Contents

Chapter 1: Preface and how to use this book 7
Chapter 2: What “Embedded Linux” actually is 15
Chapter 3: Host environment setup 31
Chapter 4: ARMv7-A and the Cortex-A7, for the MCU engineer 56
Chapter 5: A tour of the i.MX6ULL SoC 77
Chapter 6: The toolchain 95
Chapter 7: The Boot ROM, IVT, DCD, and BootData 119
Chapter 8: Hardware bring-up checklist 135
Chapter 9: First LED, pure assembly 150
Chapter 10: C + startup.S + linker script 162
Chapter 11: Hand-building a Boot ROM-acceptable image 175
Chapter 12: UART driver and printf 184
Chapter 13: CCM clock tree bring-up 197
Chapter 14: DDR3 initialization with MMDC 208
Chapter 15: Exceptions and the GIC 223
Chapter 16: Timers, EPIT and GPT 235
Chapter 17: MMU and caches 244
Chapter 18: Optional bare-metal peripherals 259
Chapter 18A: Project organization 274
Chapter 18B: Button input and beep 285
Chapter 18C: Bare-metal RTC 295
Chapter 19: U-Boot from source, first boot 306
Chapter 20: U-Boot SPL: the missing link 317
Chapter 21: U-Boot internals 327
Chapter 22: Porting U-Boot to the MINI 340
Chapter 23: bootcmd, bootargs, FIT images 358
Chapter 24: Workflows, TFTP, NFS, USB-OTG 373
Chapter 24A: Building an i.MX6ULL teaching port from scratch 386
Chapter 24B: U-Boot board policy with GPIO and I2C 456
Chapter 24C: Ethernet fallback boot in U-Boot 469
Chapter 24D: Board identity and variant selection in U-Boot 481
Chapter 24E: Multi-variant FIT images and DT overlays 494

Chapter 24F: Watchdog, bootcount, and rollback in U-Boot 505
Chapter 24G: Factory and recovery modes in U-Boot 518
Chapter 24H: PMIC and power policy in U-Boot 530
Chapter 24I: U-Boot display and boot screen 542
Chapter 24J: SPI LCD and I2C OLED displays in U-Boot 553
Chapter 25: Building mainline Linux for i.MX6ULL 567
Chapter 26: Booting the kernel from U-Boot 577
Chapter 26A: Kernel boot failure playbook 586
Chapter 27: Device Tree: the contract between firmware and kernel 593
Chapter 27A: DT bindings YAML and dt_binding_check 608
Chapter 27B: Device Tree for a custom i.MX6ULL board 618
Chapter 28: Kernel startup, traced 629
Chapter 29: Initramfs from scratch 642
Chapter 29A: Initramfs as a recovery system 650
Chapter 30: Kernel configuration deep-dive 659
Chapter 30A: Kernel lifecycle: mainline, stable, LTS, vendor BSPs 669
Chapter 30B: Product kernel configs 678
Chapter 31: A root filesystem, by hand 686
Chapter 32: /proc, /sys, devtmpfs 698
Chapter 33: Init systems 708
Chapter 34: libc, dynamic linking, and the loader 716
Chapter 35: Buildroot, after you can do it by hand 726
Chapter 35D: Bootable SD and eMMC image layout 737
Chapter 35I: Boot from SPI NOR flash 743
Chapter 35E: Buildroot product board directory 751
Chapter 35A: Ubuntu-base rootfs as a peer to BusyBox/Buildroot 757
Chapter 35F: Your application as a service 767
Chapter 35G: Networking, time, and logging 772
Chapter 35B: Read-only rootfs + overlayfs 778
Chapter 35H: Rootfs security basics 787
Chapter 35C: Container runtimes on embedded 793
Appendix — Userspace tooling reference 803
Chapter 36: Your first kernel module 818
Chapter 37: A character driver, by hand 828
Chapter 38: Auto-creating /dev/ nodes 839

Chapter 39: Platform drivers + device tree 848
Chapter 40: The misc framework 859
Chapter 41: Concurrency in the kernel 865
Chapter 42: Sleeping, waiting, polling 877
Chapter 43: Interrupts 887
Chapter 44: GPIO subsystem + pinctrl 898
Chapter 45: Input subsystem 909
Chapter 46: I²C drivers 918
Chapter 47: SPI drivers 929
Chapter 48: PWM and RTC subsystems 938
Chapter 49: IIO subsystem (ADCs, sensors) 946
Chapter 50: regmap 955
Chapter 51: DMA 964
Chapter 51A: Watchdog 971
Chapter 51B: Power management 977
Chapter 52: Network driver: FEC + KSZ8081 985
Chapter 52A: PREEMPT_RT 993
Chapter 53: Sound: ALSA and ASoC 999
Chapter 54: LCD framebuffer and DRM/KMS 1006
Chapter 54A: MTD / UBI for raw NAND 1013
Chapter 54B: V4L2 + GStreamer 1019
Chapter 55: USB gadget 1025
Chapter 55A: Kernel timers and hrtimers 1031
Chapter 55B: Async notification (SIGIO) 1036
Chapter 55C: CAN bus (SocketCAN + FlexCAN) 1040
Chapter 55D: Block device drivers 1045
Chapter 55E: WiFi + wpa_supplicant 1050
Chapter 55F: Cellular modems 1056
Chapter 55G: Multi-touch (GT911) 1061
Chapter 55H: RGB-to-HDMI bridge (sii902x) 1065
Chapter 55I: Rust for Linux 1069
Chapter 64: QSPI NOR flash 1076
Chapter 65: I²C / SPI EEPROM 1096
Chapter 66: SD card and eMMC deep dive 1111
Chapter 67: Temperature / humidity / pressure 1124

Chapter 68: Light & color sensors 1142
Chapter 69: Air quality, gas, particulate matter 1154
Chapter 70: I²C IMUs 1168
Chapter 71: SPI IMUs 1186
Chapter 72: Distance & proximity sensors 1200
Chapter 73: Magnetometer / compass 1214
Chapter 74: Hall-effect & rotary position sensors 1225
Chapter 75: Current & power monitoring 1235
Chapter 76: Battery fuel gauge + charger 1250
Chapter 77: 1-Wire sensors 1263
Chapter 78: MEMS microphones 1274
Chapter 79: Health sensors 1283
Chapter 80: External ADCs 1296
Chapter 81: External DACs + clock generators 1307
Chapter 82: RGB parallel LCD on LCDIF 1318
Chapter 83: SPI LCD 1330
Chapter 84: QSPI LCD 1341
Chapter 85: OLED & e-paper 1347
Chapter 86: Touch input ICs 1359
Chapter 87: Parallel CSI cameras 1371
Chapter 88: USB UVC cameras 1385
Chapter 89: I²S audio codecs 1392
Chapter 90: Digital class-D amplifiers 1403
Chapter 91: SDIO WiFi 1412
Chapter 92: USB WiFi 1421
Chapter 93: Hosted WiFi via ESP32 / ESP8266 1428
Chapter 94: WiFi+BT combo modules 1436
Chapter 95: HCI Bluetooth over UART/USB 1443
Chapter 96: AT-command BLE modules 1452
Chapter 97: BLE Mesh 1458
Chapter 98: LoRa 1465
Chapter 99: Sub-GHz proprietary 1484
Chapter 100: ZigBee / Thread / 802.15.4 1501
Chapter 101: UWB ranging 1512
Chapter 102: USB 4G LTE modems 1523

Chapter 103: UART AT-command modems 1534
Chapter 104: NB-IoT / Cat-M1 1543
Chapter 105: RFID / NFC 1551
Chapter 106: Fingerprint sensors 1563
Chapter 107: GPS / GNSS + PPS 1573
Chapter 108: RS-485 + Modbus RTU 1583
Chapter 109: LIN bus 1592
Chapter 110: CAN deep dive 1601
Chapter 111: Quadrature encoders & rotary 1611
Chapter 112: Stepper & DC motor drivers 1620
Chapter 113: WS2812 / SK6812 / APA102 addressable LEDs 1628
Chapter 114: Beepers, relays, SSRs 1637
Chapter 115: Dual FEC + hosted Ethernet 1645
Chapter 116: PMICs and regulator framework 1654
Chapter 117: External RTC 1663
Chapter 118: JTAG, OpenOCD, GDB at every layer 1672
Chapter 119: Kernel debugging without JTAG 1682
Chapter 120: User-space debugging 1690
Chapter 120A: Mainline patch submission workflow 1699
Chapter 121: Custom board port as a system design 1710
Chapter 121A: CI/CD for embedded Linux 1720
Chapter 122: Build your own cross-toolchain 1729
Chapter 122A: BSP → mainline migration playbook 1740
Chapter 123: Yocto vs Buildroot, an honest comparison 1749
Chapter 123A: Yocto layer development in depth 1759
Chapter 124: Secure boot (HAB) and OP-TEE 1769
Chapter 125: Field updates 1781
Chapter 125A: VSCode + gdbserver remote-debug workflow 1791
Chapter 126: Closing: what to read next 1798
Chapter 127: Why embedded products use hypervisors 1804
Chapter 128: QEMU as a virtual hardware model 1822
Chapter 129: Build and boot tiny Linux in QEMU 1832
Chapter 130: U-Boot in QEMU 1843
Chapter 131: HYP mode, stage-2 MMU, and virtual interrupts 1852
Chapter 132: Xen in QEMU 1865

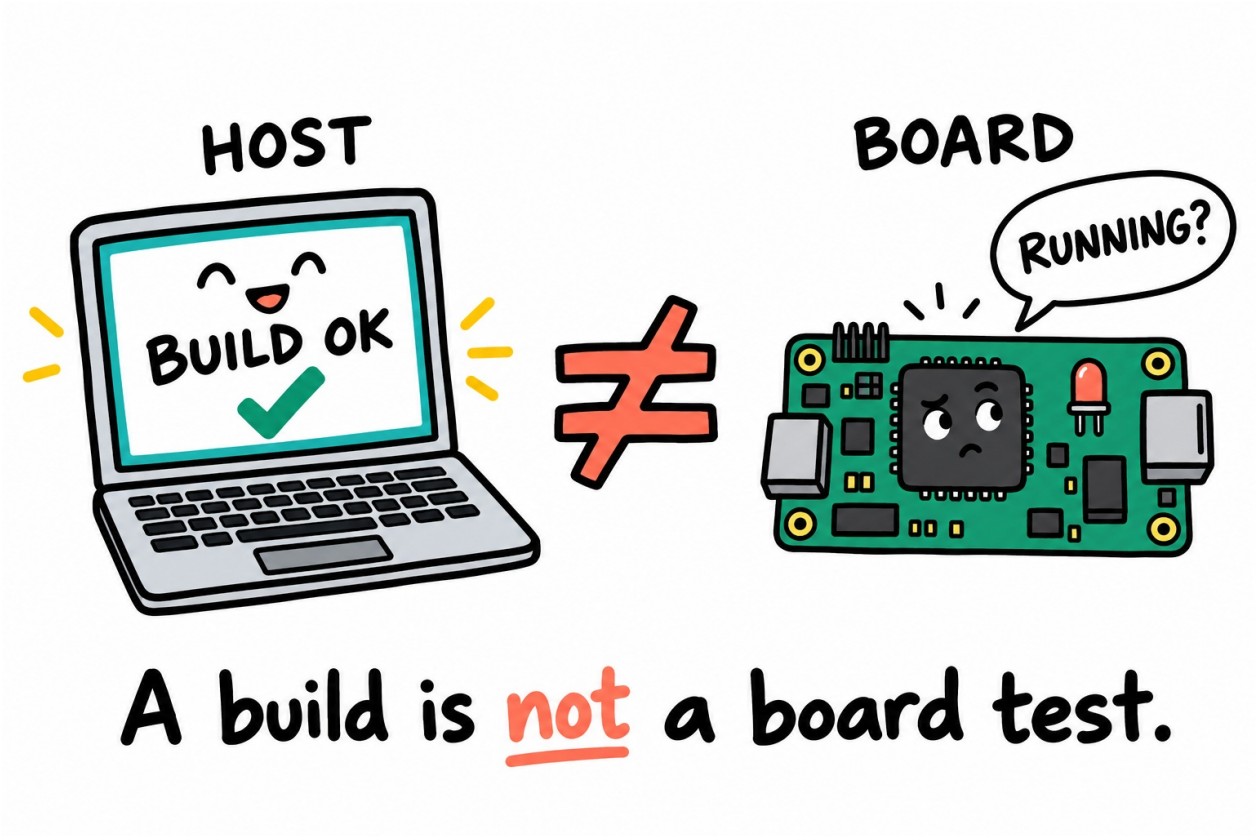
Chapter 133: First DomU Linux 1875
Chapter 134: Xen on i.MX6ULL 1884
Chapter 135: DomU Linux on i.MX6ULL 1896
Chapter 136: Devices, memory, and DMA boundaries 1905
Chapter 137: Jailhouse in QEMU ARM64 1913
Chapter 138: Bare-metal or Zephyr inmate cell 1930
Chapter 139: STM32MP1 Linux plus RTOS, the production pattern 1945
Embedded Linux on i.MX6ULL — From First Boot to First Driver 1961
Status 2029

Chapter 1: Preface and how to use this book

1.1 Why this book exists

Imagine this: the build finishes without an error, you prepare the image as instructed, and the board gives you nothing. No Linux prompt. No useful serial message. The files on the host look convincing, but you cannot yet tell whether the processor reached your code.

If you have brought up a microcontroller board, you already have a way to approach that silence. Follow reset to startup, startup to `main()`, and `main()` to the GPIO register. There may be a mistake somewhere, but the path is small enough to inspect. On an embedded Linux board, several programs run before your application gets a turn. Memory must be prepared, images must be loaded, and hardware must be described. Which of those steps did the board actually reach?



The laptop is pleased with itself. We still need an answer from the board. A clean build checks the host’s work. A response from our program, such as an LED change or a printed message, tells us it ran on the board.

A vendor board-support package, usually shortened to **BSP**, can give you a working system quickly. Install its tools, build an image, write an SD card, and you may have a Linux prompt. That is a useful starting point. The harder moment comes when you

change the design: a different DDR chip, a UART on different pins, a board that no longer matches the supplied image. Now you need to know which setting belongs to the bootloader, which belongs to Linux, and which assumption came from the old hardware.

This book follows those questions on the i.MX6ULL. The first example is modest: an LED controlled by a program small enough to follow from its first instruction. Startup code, memory layout, and the Boot ROM's header each have a job. Understanding those jobs matters more than reproducing the listing.

The same board then takes us further. A page table changes what an address means. A **device tree** describes hardware to Linux. U-Boot prepares a handoff, and a root filesystem supplies the programs that run afterward. Even when the software becomes large, we keep asking who owns a resource, what an interface promises, and which assumption lets the next layer work.

Implementation tools, including AI assistants, can help produce code. They do not choose the right failure boundary, establish a board's wiring, or turn an unchecked assumption into evidence. The aim here is to understand a design well enough to choose it, explain it, and assess an implementation of it.

Buildroot enters in Chapter 35, toolchain construction in Chapter 122, Yocto in Chapter 123, and secure boot in Chapter 124. Their inputs make more sense once the pieces and their responsibilities are clear. A large build may still fail. The useful difference is knowing which layer to investigate instead of starting over and hoping.

1.2 Who this book is for

If you can read C, find a register in a reference manual, and follow a signal through a schematic, you have the starting point this book needs. Experience with a Cortex-M part, such as an STM32, gives us something familiar to return to when the Cortex-A7 behaves differently.

Linux experience is not required. The terminal may be new to you; the hardware is not. Perhaps you have built a vendor image but still cannot explain why the directory contains both `vmlinux` and `zImage`. You do not need to settle that question before opening Chapter 2. We will meet those files when we have a reason to inspect them.

Chapter 3 uses a host workspace to explain the build environment. When a command appears, its important consequence is what changes: a file, this terminal's environment, or a persistent host setting. Commands illustrate those boundaries rather than form assignments to complete before reading on.

You can read the architecture without connecting a board. The worked hardware references are:

- The Point Atom MINI v2.2 baseboard and its supplied core-board schematic. Another fitted revision may have different wiring and components.

- The matching baseboard/core schematics and the supplier’s power instructions for that combination. The schematics supplied with this book are not a validated power-connection guide. Obtain that guide from the board supplier before attaching a supply, USB cable, or adapter.
- The x86_64 Linux host model described in Chapter 3. Host paths, target addresses and device names belong to different systems.

If you independently explore an example on hardware, the actual assembly, documented supply, measurement equipment and cable arrangement must be established first. Storage, JTAG and peripheral add-ons are not reading prerequisites. Explanatory code is not permission to write a disk, change a rail or program a fuse.

1.3 What “raw” means in this book

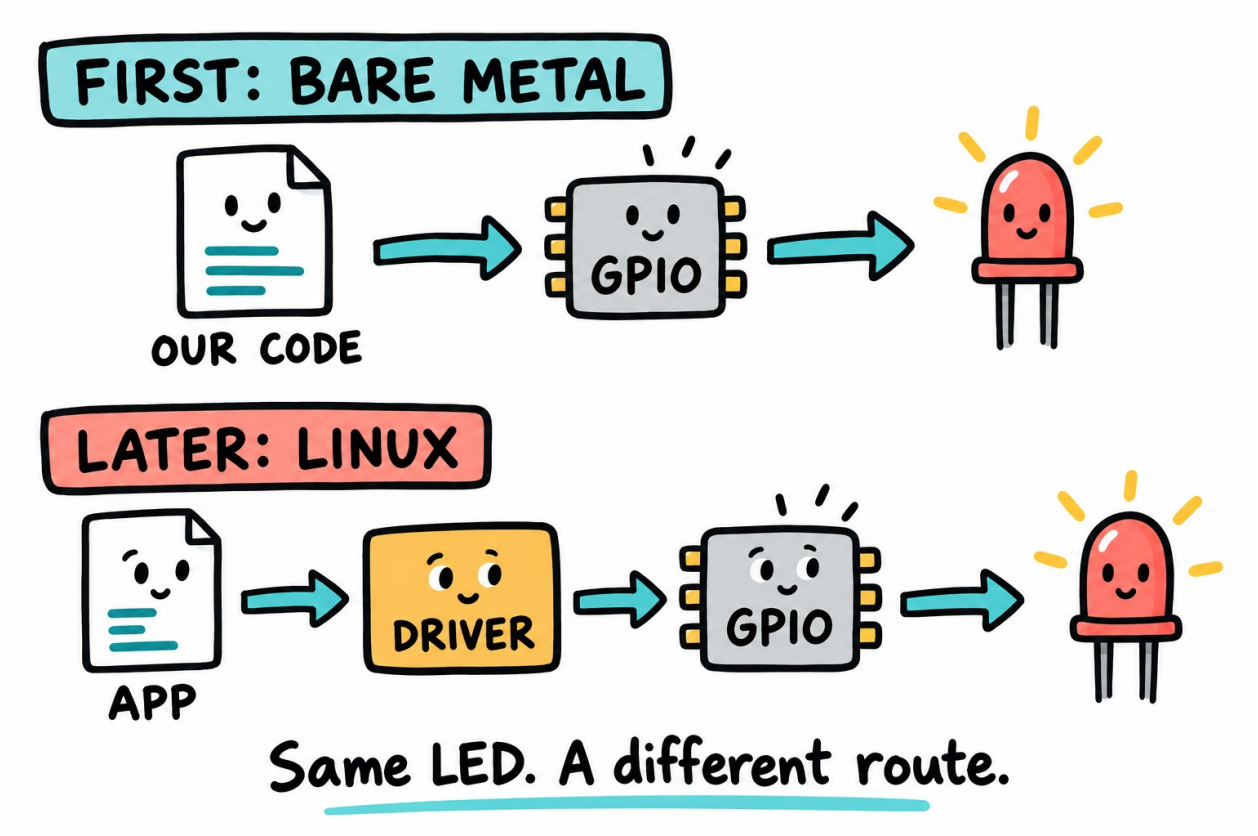
Here, “raw” means being able to point to the thing that did the work. Which compiler produced the instructions? Which file described where they belong in memory? Which header told the chip where to begin? We start with two official prebuilt Arm compilers, one for bare metal and one for Linux. They stay in the project directory, and we choose between them explicitly. Building a compiler is a separate subject, covered in Chapter 122.

Our worked board is the **Point Atom i.MX6ULL MINI**, using the supplied MINI V2.2 and core-board schematics. Part III studies Point Atom’s vendor source as a board baseline and a separate, pinned **mainline** tree for current U-Boot internals and migration. An EVK comparison is not MINI firmware. The Linux track uses mainline sources, with board-specific support considered separately. The later BSP-to-mainline migration chapter returns to that transition. Configuration examples explain the fields that express a design choice. Yocto and Buildroot follow the underlying components so that their abstractions have something concrete to refer to.

Pinned source versions make the examples specific enough to discuss. They are not a claim that every path has been qualified on hardware. When assessing a design, distinguish what the source describes, what a build checks, and what only measurements on the intended system can establish.

1.4 Scope and optional extensions

The main route follows one i.MX6ULL board from bare metal to Linux drivers. The same LED appears under direct register control and under Linux’s GPIO ownership. The UART moves from a small register interface to a kernel-managed device. Returning to familiar hardware lets us see what the new software layer changes. The reasoning is useful on other boards, but register addresses, DDR settings, and electrical requirements are board-specific.



We will visit this LED twice. First, our small program does the register work. Later, a Linux application uses a driver interface. The two rows are different experiments on the same hardware, not two programs trying to drive it at once.

There are optional routes into PREEMPT_RT, containers, and Part IX's virtualization and mixed-core designs. These address different timing, isolation and deployment needs. Read them when that design question matters, not because every product needs every layer.

This is not a Linux application-programming book. We use a shell and small C test programs, but do not teach POSIX threads or `select`/`epoll` in depth. Kubernetes deployment and Android system builds are outside the route.

1.5 How each chapter is organized

Read the early chapters with a question in mind. Why does an application need a driver? Why does a boot image contain addresses before its code? Why are there several files called a kernel? Names and formats become useful when they answer a concrete question.

A chapter develops the mechanism, the choices it enables, and the constraints those choices introduce. Code, commands and diagrams make the interfaces visible. **Key Notes** collect the architectural points worth retaining: ownership, dependencies, tradeoffs and failure boundaries. Pitfalls explain where a plausible design or implementation can go wrong. There are no assigned labs or completion tests.

The references at the end are for a second pass. You do not need to read every linked manual or source file before continuing. Use them when you want a fuller explanation or need to settle a specific question.

1.6 Key Notes

- **Layers have owners.** ROM, bootloader, kernel and application make different decisions. A failure becomes easier to reason about when the responsible layer is identified.
- **Interfaces carry assumptions.** Image headers, calling conventions, Device Tree properties and driver APIs are contracts between components, not arbitrary syntax.
- **Design choices have costs.** A simpler boot path, a richer filesystem or stronger isolation changes memory, recovery, maintenance and integration requirements.
- **Code is evidence of an intended mechanism, not of its outcome.** Compilation, simulation and physical measurement answer different questions.
- **Implementation assistance does not remove engineering judgment.** Generated code still has to fit the chosen ownership, timing, resource and failure model.
- **Hardware limits remain real.** Electrical compatibility, storage identity and irreversible operations require specific evidence regardless of how the code was produced.

1.7 Code listings

Code is **included in the chapters** to expose a mechanism or interface. A startup file can look uninviting at first. Read it in the groups explained beside it: prepare the stack, clear a memory range, call the C function. The valuable question is why each operation belongs there and what depends on its order, not whether you can type every line.

Pseudocode shows an idea, a fragment shows part of a larger implementation, and illustrative output shows what a tool may report. Some longer examples show how the pieces fit. None replaces board qualification or a complete product design. A retained command sequence is an explanatory example, not a reading assignment.

1.8 Conventions

Prompts

Once the board runs software, you will often have two terminals open: one on the host and one connected to the board. The prompt tells you where a command belongs:

```
$        a regular user prompt on the host PC
#        a root prompt on the host PC (used sparingly)
=>       the U-Boot prompt
target$  a regular user prompt on the i.MX6ULL board
target#  a root prompt on the i.MX6ULL board
```

Type the command after the prompt, not the prompt itself. For example, `$ sudo picocom ...` means enter `sudo picocom ...` in your normal host terminal. The `#` in a root-prompt example is different from the `#` that begins a comment inside a shell script. Blocks without prompts are file contents or command blocks identified in the surrounding text.

Registers and bits

Registers are written in uppercase with the bank prefix from the reference manual:

```
CCM_CCGR5 |= (3u << 24);    /* UART1 CG12: bits 25:24 = 11 */
```

This is register pseudocode, not a complete UART setup. UART1's two-bit gate is in `CCM_CCGR5` at `0x020C407C`; `11` enables it in run and WAIT modes, but not STOP [RM 18.6.28]. MMIO definitions, pin mux, and the UART root clock are supplied later.

When a bit field is named, square brackets denote a field rather than C array syntax:

```
CCM_CCGR5[CG12] = 3;       /* the same gate field, notation only */
```

Numeric notation

Hex values are written with the C `0x` prefix everywhere except inside hex dumps. Megabytes and gigabytes use the IEC binary prefixes (MiB, GiB) when precision matters. "MB" is shorthand for the marketed quantity (board has "512 MB DDR3", the chip is actually 512 MiB).

Addresses

When we cite a memory address, we cite the *physical* address unless we are discussing MMU mappings. Physical addresses on i.MX6ULL are 32 bits. In Chapter 17, virtual addresses are shown in the kernel and user ranges being discussed.

Citations

References to the i.MX6ULL Reference Manual are written as **[RM §28.5.3]**. This means Chapter 28, Section 5.3 of the *i.MX 6ULL Applications Processor Reference Manual*, revision 1, 11/2017. Linux source citations look like **[linux: drivers/gpio/gpio-mxc.c: 142]** and refer to `v6.6` unless noted.

Diagrams

The small concept sketches pause over one idea at a time. Their captions explain the important distinction in ordinary text, too. A smiling chip is a character in the explanation, not an extra component to find on your board; these drawings are not wiring instructions or to-scale memory maps.

Detailed reference diagrams are also drawn with text, so they remain readable in the Markdown source as well as on the website. The sketches are ordinary image files; you do not need a separate diagram-rendering tool to read or build the book.

1.9 How the chapters depend on each other

On a first reading, Chapters 1-8 establish the host/target distinction, the core, and the boot-image contract. Part II uses a small LED/UART program to expose startup, clocks, interrupts and memory ownership. Part III shows how U-Boot organizes those responsibilities. The kernel, root filesystem and driver parts then explain the next layers. Understanding an earlier mechanism helps with a later one, but completing a program is not a condition for continuing.

Readers interested primarily in the boot stack can move from the foundations to Chapter 19’s U-Boot source study, then return to Part II when a register-level detail needs explanation. Physical use of those examples has separate board prerequisites; reading the architecture does not require a connected board.

The table below suggests routes by the design question you want to answer:

If you…	Read	Skim	Skip
Want the full architectural progression	Numbered core chapters	supplementary detail as needed	unrelated optional designs
Have MCU experience but are new to Linux	1-17, 19-24, kernel/ rootfs and driver foundations	familiar C/build detail	unrelated peripheral add-ons
Want the Linux boot architecture first	1-8, 19-30, 31-35	9-17	optional peripheral implementations
Already shipped Linux on a different SoC, want i.MX6ULL specifics	1, 5, 7, 19-24, 27	25-35	9-18

If you…	Read	Skim	Skip
Maintain an existing BSP, want driver depth	1, 27, 36+	25-35	2-24

When starting in the middle, use the chapter's opening and Key Notes to identify its assumptions. Knowing a similar mechanism on another chip does not supply that board's wiring, timing or memory configuration.

1.10 A note on the i.MX 6ULL Reference Manual

Keep revision 1 (11/2017) of the *i.MX 6ULL Applications Processor Reference Manual* nearby. At 4,127 pages, it is a poor place to start with "I should learn all of this." Start with "I need the clock gate for UART1." That question takes you to a few relevant pages and gives you a reason to read them carefully.

For a new peripheral, start with these five questions:

1. Where are its registers? Start with the system memory map.
2. Which clock reaches the block? Follow the CCM clock path and gate.
3. Which pads carry its signals? Check the IOMUXC tables against the schematic.
4. Which interrupt ID identifies it? Look in the GIC SPI table if the block generates interrupts.
5. In what order should it be initialized? Look for the manufacturer's sequence near the start of the block's chapter.

The answers connect the software interface to the schematic. Chapter 5 works through this search for UART1, and later peripheral chapters use it in context. The goal is not to finish the whole manual. It is to find the evidence that explains a particular design.

Before following the boot sequence, there is one change to the MCU picture worth understanding. You know how to write a GPIO register. Why would Linux put a driver interface between your application and that perfectly ordinary operation? That is where we begin.

> Next: Chapter 2: What embedded Linux is. The full table of contents and draft status are reference pages, not prerequisites before Chapter 1.

Chapter 2: What “Embedded Linux” actually is

You already know how to turn on an LED. Choose the pad, enable the clock, set the direction, and write the GPIO register. If your board runs MCU firmware, your application can usually follow that path itself.

Now imagine putting the same register-writing function in a Linux application. The C may compile. The physical address may be correct. Neither fact gives the application permission or a mapping to use that register.

The LED has not become more complicated. The board has become a shared machine. A shell, a network service, and your application can run independently, and one bad pointer should not casually overwrite the others’ memory. Linux changes the rules around the operation so those programs can coexist. Let us follow what that means, starting with the firmware model you know.

2.1 The system you already understand

Think back to the startup code in a Cortex-M project. At reset, the CPU obtains its initial stack pointer and reset-handler address from the device’s vector arrangement. The vector table contains those values; it is not itself a sequence of instructions. The reset handler prepares RAM, copies `.data`, clears `.bss`, and calls `main()`. Your program then enters its main loop or starts an RTOS scheduler.

In a typical project, the application, its drivers, and the RTOS are linked into one image. Tasks may have different stacks and priorities, but they commonly share one physical address space. Pass a buffer pointer to an I2C driver, and both pieces of code use the same address to reach the same bytes. Both can use the pointer directly.

Most such projects also let application code access peripheral registers directly. Cortex-M does support privileged and unprivileged execution, and an MPU can add protection on parts that have one, but many projects do not use that separation. A call such as `i2c_read(addr, buf, len)` goes straight into the driver’s code.

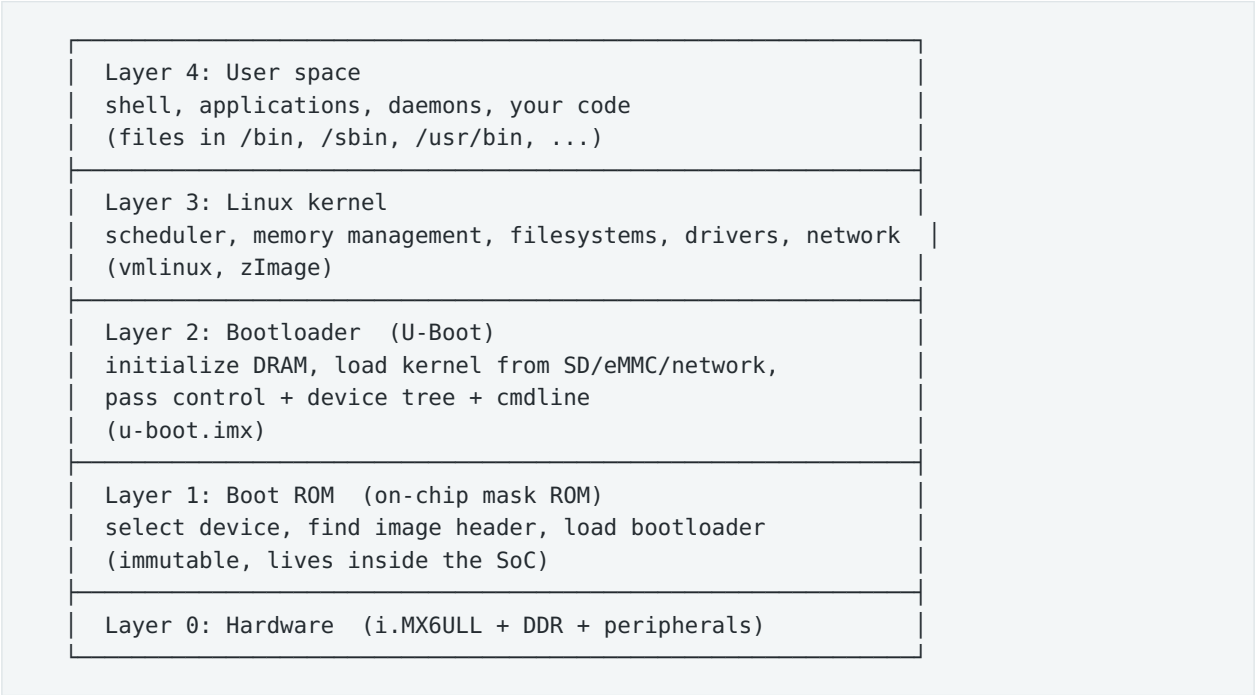
Add FatFS or littlefs, a network stack, and several RTOS tasks, and the project may already look quite capable. Linux does not make those ideas new. What happens if we want programs built independently, with private memory and controlled access to the devices they share? That is where the familiar direct-call model starts to change.

2.2 The four layers

An embedded Linux system can be viewed as four software layers above the hardware. During boot, control moves from the Boot ROM to the bootloader, then to the kernel, which starts user-space programs.

Keep three things separate: the **source** we edit, the **files** produced by a build, and the **running software** with its registers, memory, and state. An executable file in `/bin` is not yet a process. The root filesystem is the target’s directory tree mounted at `/`; it supplies programs, configuration, and any required libraries. The target does not need the kernel’s C source tree installed to run Linux.

The picture names the main roles and some typical input files, not a compulsory one-file boot recipe. A bootloader can have stages such as SPL and U-Boot. On this SoC, the ROM can also apply image-supplied hardware settings before entering the loaded code; Chapter 7 explains that route.



Read the diagram from the bottom upward, as a sequence of handoffs. At reset, there is no shell waiting to launch your application. The lower stages must make that possible first.

The Boot ROM runs first. NXP puts this small firmware inside the chip during manufacture. We cannot replace it, so our first image must use the header and placement it expects. Chapter 7 explains that format and how the ROM finds our code.

The bootloader prepares and loads the system. We use U-Boot. Its early code configures clocks and DDR, and its drivers can fetch files from storage or the network. Much of this will look familiar from MCU work. Its destination, however, is another program: the Linux kernel. In Part II, our own small bare-metal programs temporarily occupy this place in the boot sequence.

The kernel manages the running system. Once U-Boot hands over control, Linux schedules work, manages memory, and controls access to devices. Drivers are part of this layer. They turn requests such as “send these bytes” into the register operations needed by a particular UART or network controller.

User space is where applications run. It includes your program, the shell, and background services, often called *daemons*. These programs use interfaces provided by the kernel rather than assuming that every peripheral register is available to them.

By the time your application starts, the ROM and bootloader have already done their work. Your ordinary device requests go to the running kernel, not back down through every boot stage. For our LED, the important boundary is now the last one in the diagram: the application is in user space, and the driver is in the kernel. How does a request cross that boundary?

2.3 The user/kernel split, made concrete

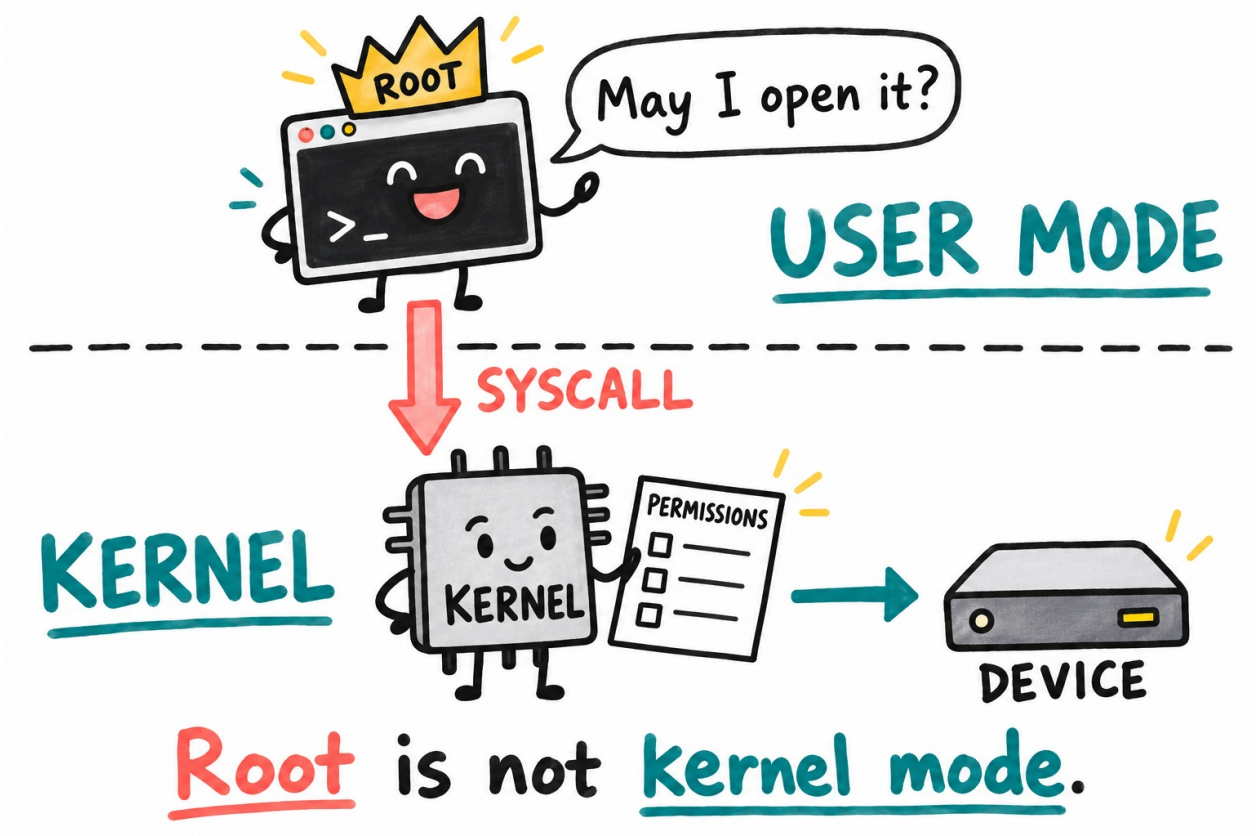
On our Linux system, application instructions and kernel instructions run with different CPU privileges. The Cortex-A7 calls ordinary application execution **USR mode**. Kernel exception handlers use privileged modes such as **SVC** and **IRQ**; Chapter 4 explains those names in detail.

- **Kernel execution** can use privileged system-control operations and kernel memory mappings.
- **User execution** cannot use privileged instructions or access protected kernel memory. An application also has no automatic mapping of peripheral registers; the kernel decides which interfaces or mappings it may use.

The CPU enforces this distinction. A user instruction that tries to write to protected kernel memory causes an exception, rather than completing the write. The kernel handles that exception and normally terminates the offending process. We will meet processes shortly; for now, think of one as a running application with its own resources.

There is an easy trap here. If `sudo` gives a program root privileges, has it crossed into kernel mode? No. **Root** is a Linux user identity; **kernel mode** is a CPU execution state. Later, `sudo picocom` gives the serial terminal credentials to open a restricted device, but its own instructions still run in user mode. The kernel checks those credentials when the program requests access. The need for `sudo` comes from device permissions, not from a rule that hardware always needs root.

An application requests a kernel service through a **system call**, usually shortened to *syscall*. On ARMv7-A, the `svc` instruction raises a Supervisor Call exception. The CPU enters the kernel handler in SVC mode, where Linux reads the syscall number from `r7` and its arguments from `r0` onward. After handling the request, the kernel returns a result to the application.



Nice crown. Still user mode. Root credentials can change the kernel’s permission decision, but they do not turn the application’s instructions into kernel instructions. The arrow follows a device request, not every instruction the application executes.

Here is a simplified path for opening an example I2C bus interface. `/dev/i2c-0` is a device-interface pathname. An **fd**, or file descriptor, is the small integer handle returned on success; Section 2.6 follows its lifetime. **libc** is the application’s C library; glibc is one implementation. The application calls its ordinary C function, and the library wrapper performs the transition into the kernel:

```
user-space process                kernel
──────────────────────────        ──────────────────────────

fd = open("/dev/i2c-0",..)
    ↓ glibc wrapper builds
    ↓ syscall args
    svc #0          ──────►       exception handler
                                  → sys_openat()
                                  → kernel parses path
                                  → finds i2c_dev driver
                                  → driver's .open() runs
                                  → returns fd
                    ◄──────       return from exception
fd in r0, back in user mode
```

Opening `/dev/i2c-0` only obtains a bus handle; it does not select an I2C peripheral or transfer bytes yet. The adapter number is illustrative, not a promise about this board. Later requests supply the address and operation.

Most application requests to the kernel take this shape. Reading a file uses `read()`. Allocating memory may use `brk()` or `mmap()`. Sleeping may use `nanosleep()`. To control an LED, an application may call `write()` on a device interface or use `ioctl()` on a device node. These operations cross the user/kernel boundary through syscalls and return a result or error to the application, which resumes in user mode. Non-root programs make syscalls too.

Not every C function needs a syscall every time it runs. `malloc()` can reuse memory the library already obtained, for example. If a driver authorizes a memory mapping, later loads and stores through that mapping do not each require a syscall. The kernel sets up and controls the access; it does not have to execute every instruction on the application’s behalf.

Linux documents its syscall interfaces in section 2 of the manual. On a Linux host, `man 2 read` opens the description of `read()`. You do not need to learn the syscall list now. First understand the request, the transition into the kernel, and the return to the caller.

Why the split exists

Suppose a network service has a bad pointer. You would rather lose that service than let it overwrite the application’s state or the kernel’s scheduler. Suppose two programs want the I2C bus. Their requests need coordination, even though neither program knows the other’s code. Linux’s separation addresses these problems:

1. **Robustness.** A bad pointer in an ordinary application is normally contained by its mappings and permissions, rather than overwriting kernel data or another program’s private memory.
2. **Access policy.** A driver can check which operations a caller is allowed to request, rather than giving every program unrestricted control of its hardware.
3. **Resource arbitration.** Many processes want the I²C bus, the CPU, or the network. The kernel coordinates supported transfers and schedules work. It does not automatically turn several separate application requests into one atomic device protocol; drivers and applications still need appropriate ownership or locking.

For one tightly controlled firmware image, a smaller RTOS and direct driver calls may be the right design. Linux takes on a different job: managing independent programs and the resources between them. Its driver boundary starts to make sense once more than your one application needs the board.

2.4 Virtual memory, in one section

Start with bare-metal code using physical addresses, with the MMU disabled. This is **symbolic pseudocode, not a register write to run**:

```
*(volatile uint32_t *)PHYSICAL_REGISTER_ADDRESS = value;
```

The manual supplies a block base and each register’s offset; together they give the physical register address. For example, `0x020E0000` is the i.MX6ULL IOMUXC **block base**, not a pad register to write at offset zero. A valid register access must also follow that register’s width and access rules.

Carry that same number into a Linux application and a less obvious question appears: is it still the address of that register?

With the MMU enabled, a user-space pointer is usually a **virtual address**, not a direct physical bus address. Before the CPU can load or store memory, the MMU translates:

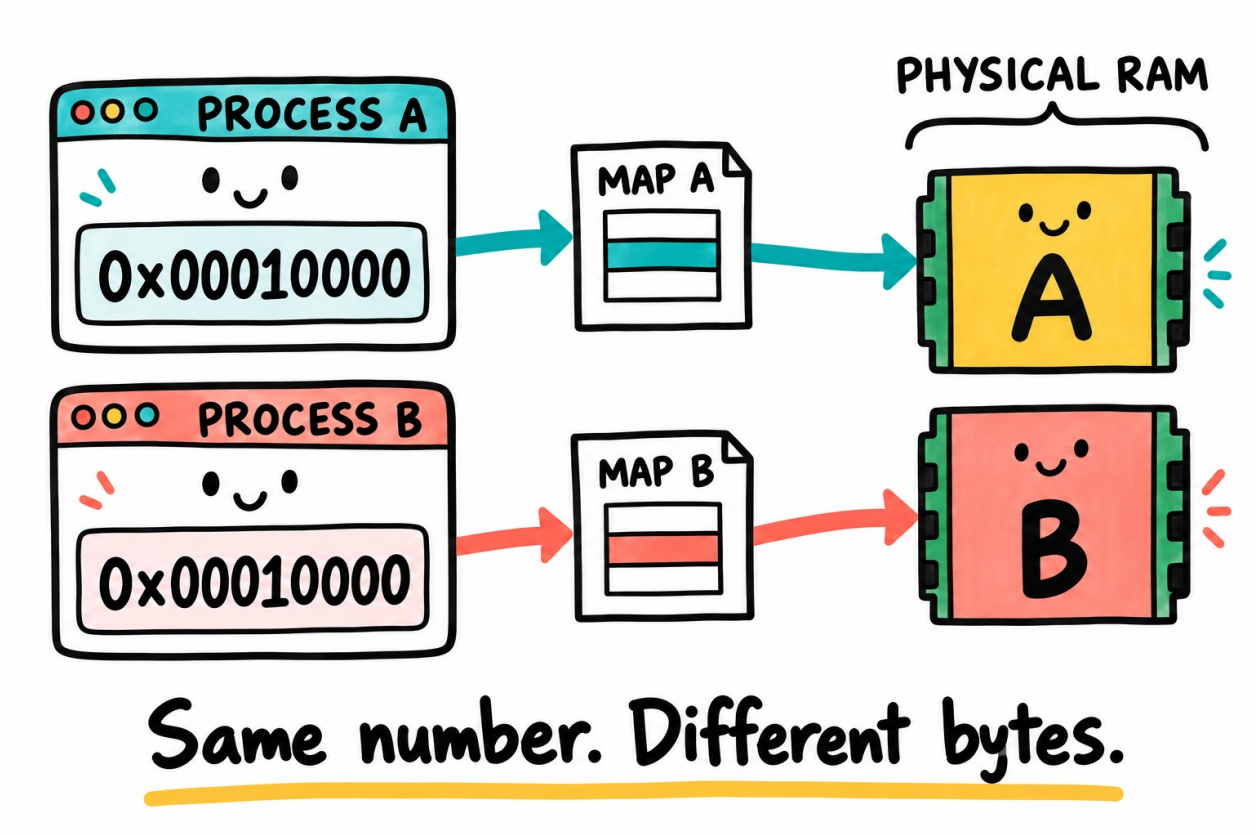
```
virtual address used by the program
        |
        v
MMU looks in the current process page table
        |
        v
physical address in RAM or in a device register block
```

Consider two applications that both have a pointer with the value `0x00010000`. In the shared MCU model, the same address would normally reach the same bytes. With separate process mappings, it need not. The following illustrative addresses show the difference; they are not a required Linux memory layout:

```
Process A:
  virtual 0x00010000  -> physical RAM for process A's code
  virtual 0x000B0000  -> physical RAM for process A's data
  virtual 0xBE000000  -> physical RAM for process A's stack

Process B:
  virtual 0x00010000  -> physical RAM for process B's code
  virtual 0x000A0000  -> physical RAM for process B's data
  virtual 0xBE000000  -> physical RAM for process B's stack
```

Look at `0x00010000` in both lists. The numbers match, but the physical RAM can be different. The current process’s mapping gives the pointer its meaning. Each application can therefore work with its own code, globals, and stack without first finding a free physical address above every other application’s allocations.



Matching numbers need not mean matching bytes. Here, each process has a private mapping to a different RAM region. Those regions can belong to the same chip. Shared mappings can instead lead to the same physical pages.

The **page table** describes this translation and its access permissions. A **page** is a fixed-size unit of memory; the ordinary small-page mappings in this route use 4 KiB pages. You can think of the table as a map owned by the kernel:

```
For process A:
  virtual page X -> physical page Y, readable and executable
  virtual page Z -> physical page W, readable and writable
  some pages     -> not mapped at all
```

What if a translation is missing, or its permissions do not allow the attempted access? The CPU raises a memory-access exception that Linux handles as a **page fault**. The name sounds like a program has already gone wrong, but that is not always the case. Linux must decide whether the access is valid and can be completed.

The kernel may decide:

- This address is valid but not loaded yet, so allocate RAM and continue.
- This address belongs to a file mapping, so attach an already cached page or fetch the file data if needed, then continue.
- This is a permitted write to a private copy-on-write page, so supply a private copy and retry the instruction.

- This access is invalid or prohibited, so deliver `SIGSEGV`, whose default action terminates the process.

This also separates an address range from the RAM currently backing it. A process can have a valid range for which physical pages are supplied as needed. The kernel decides whether a fault can be resolved or whether the access must be rejected.

Several later Linux features build on this mechanism. Treat the names below as examples of where the idea leads, not as APIs to memorize before Chapter 3:

- **Process isolation.** An ordinary store can reach only memory mapped for that process with the required permissions. Private mappings protect another process’s private state. Deliberate shared mappings and authorized cross-process operations are different cases; process separation is not an unconditional sandbox.
- **Memory-mapped files.** For a regular file, `mmap()` can give the program a virtual range instead of a separate `read()` buffer. Missing translations may fault; resident, permitted pages need no syscall or storage read for each access. The same API can also create anonymous memory or a driver-authorized device mapping.
- **fork().** A child initially shares many physical pages with its parent. Private writable pages can be mapped read-only so that a write faults; Linux makes a private copy when required and resumes the instruction. This is **copy-on-write**. Sharing a physical page does not mean either process may silently change the other’s private view.
- **Shared libraries.** A library such as `libc.so` contains common code used by many programs. Linux can map the same physical code pages into many processes, while each process still has its own private stack and heap.
- **Swap.** Swap means the kernel can move idle memory pages out of RAM and onto storage, then bring them back later. Embedded systems often disable swap, but it uses the same page-table and page-fault machinery.

Now return to the i.MX6ULL IOMUXC register block at physical address `0x020E0000`. A normal user process does not automatically have that physical address in its page table. If the process tries to treat `0x020E0000` as a pointer, the MMU interprets it as a virtual address. Unless the kernel deliberately mapped that virtual page for the process, the access faults.

We can now explain the opening puzzle. The physical GPIO address from the manual is not automatically a usable pointer in the application. The normal route is to request an operation through a driver interface. The driver reaches the hardware using the appropriate mappings and register accesses. The address was correct; our assumption about who could use it was not.

Chapter 17 follows a minimal first-level page table on bare metal, making the translation and permission decisions visible without Linux’s larger memory-management machinery.

2.5 Processes, threads, and where they live

An RTOS task gives you a familiar starting point: execution needs registers, a stack, and scheduler state. In Linux, that execution is a **thread**. A **process** groups one or more threads together with the resources they share.

A process has:

- A unique **PID** (process ID).
- A **virtual address space** (its own page table).
- A set of **open file descriptors** (more on these in a moment).
- A **current working directory**, a user ID and group ID (**UID**/**GID**), signal handlers, resource limits, and other state visible under `/proc/<pid>/`.

Threads within the process share its address space and file descriptors, but each has its own registers, stack, and execution state. Linux represents a schedulable thread as a task with a `struct task_struct`. The distinction is useful: two threads in one application can share a global variable directly, while two independent processes need an agreed sharing mechanism.

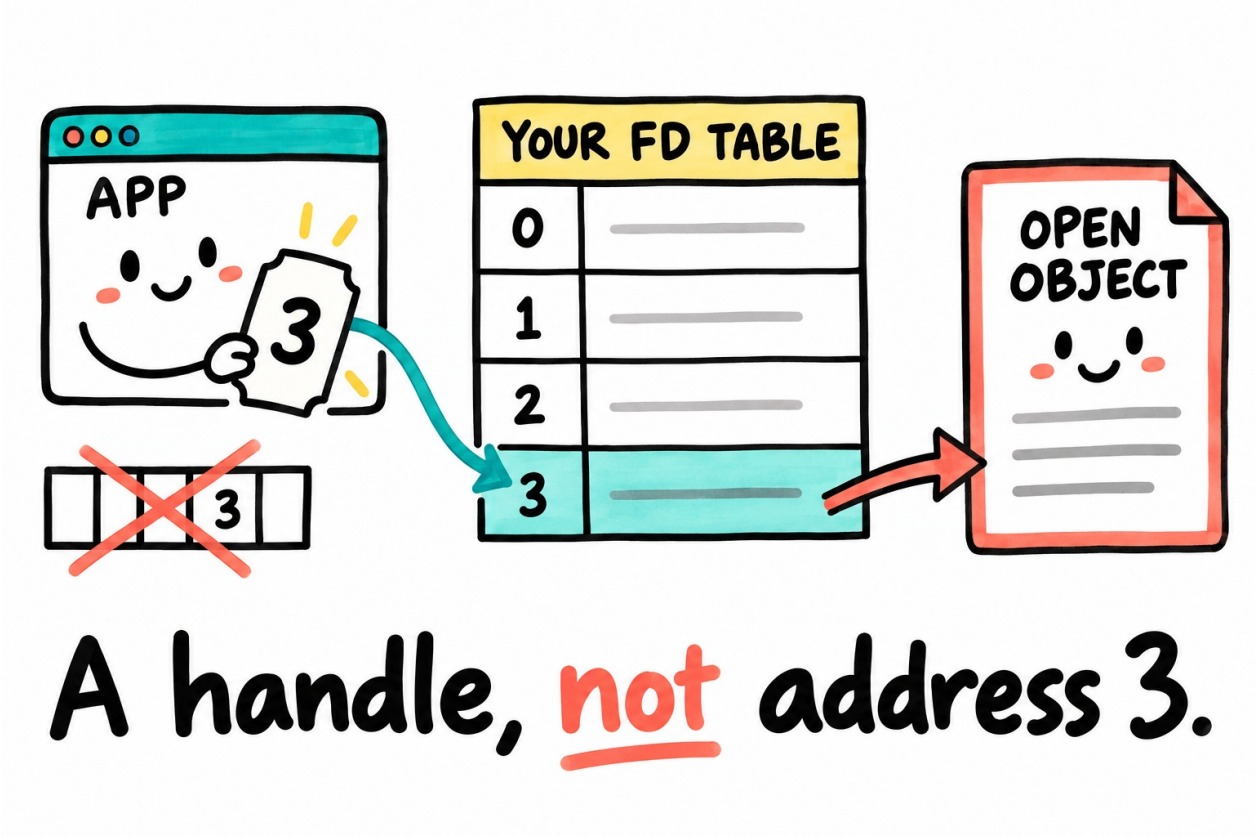
What about ISRs? A Linux **hard interrupt handler** runs in kernel interrupt context and must not sleep or make blocking allocations. Its stack arrangement depends on architecture and configuration. A threaded interrupt handler has different rules because it runs in a schedulable kernel thread. Chapter 43 explains that distinction.

2.6 Names you will meet in the next chapters

We now have a place for the remaining names: some describe the handles applications use, some describe the kernel’s internal objects, and some describe the files we build. You can return to this section when a name appears in a later example.

File descriptor (fd)

Opening a device does not hand its register address to the application. Linux gives the process a **file descriptor**, a small non-negative integer that identifies the open object in its fd table. Suppose the result is 3. A later `read(3, ...)` means “read from the object I opened,” not “read memory at address 3.” The kernel can find the object and its operations from that handle.



The number is a ticket into a table, not the object itself. The kernel manages that table for the process. Two unrelated processes can both have fd 3 for different open objects. Copying that number alone does not transfer the descriptor.

Later calls such as `read(fd, ...)` tell the kernel which open object to use. That object may be a regular file, a socket, a pipe, or a device. Other descriptor-creating APIs, including `socket()`, `pipe()`, and `eventfd()`, can wait until we need them.

An `open()` through the C library returns `-1` on failure and sets `errno` to describe the error. Do not use that result as a valid descriptor. On success, keep the fd while using the object, then call `close(fd)` to release that reference. The number can be reused for a later open; other descriptors may still refer to the original open object.

By convention, fd 0 is stdin, 1 is stdout, and 2 is stderr. An open gets the lowest free number, even 0, 1, or 2 if its old descriptor was closed. Stdout can lead to a terminal, a file, or a pipe; it is not permanently wired to a UART.

This common handle explains the phrase “everything is a file.” The phrase is not literally true, but many interfaces use the same `read()`, `write()`, and `ioctl()` model. A serial port and a text file behave differently, yet an application can refer to each through an fd.

syscall, libc, glibc, musl

The C functions in your application are not the kernel itself. **libc**, the user-space C library, implements standard C facilities and wraps many syscalls in ordinary function calls. Glibc and musl are two implementations we will encounter. The syscall underneath is the numbered kernel operation invoked through `svc` on this board.

When you call `printf()` from a C program, the path is roughly as follows. The VFS named here is the kernel’s common file-I/O interface; its filesystem details are in the optional reading at the end.

```
printf("hi\n")
   → glibc formats the string into a buffer
   → glibc calls write(1, buf, 3)
       → glibc's write() wrapper places 1, buf, 3 into registers
       → executes svc #0 with syscall number for write
           → kernel exception handler
               → sys_write()
                   → VFS layer
                       → tty driver's write callback
                           → UART register access
```

This is a simplified path when stdout is a UART terminal. Buffering and the selected libc can change the calls. The libc and kernel implementations are available as source; Chapter 28 follows kernel startup, while the driver chapters explain device operations.

Process tree, init

Who starts the first application? The kernel runs the selected **init** program, commonly `/sbin/init` on a mounted root filesystem or `/init` in an **initramfs**, a small initial filesystem unpacked into RAM. It becomes PID 1 and starts the ordinary user-space process tree. Kernel threads have a different origin.

BusyBox init and systemd are two possible init implementations, but the first experiment can be much smaller. In Chapter 29, our statically-linked PID 1 prints a message and deliberately reboots. It must not simply return from `main()`: the system’s PID 1 exiting causes a kernel panic.

Kernel module (LKM)

A driver does not always have to be built into the kernel image. A **kernel module** stores kernel code in a separate `.ko` file that can be loaded while the system is running. `insmod foo.ko` loads it; `rmmod foo` unloads it when unloading is permitted.

The `.ko` suffix can make a module look like one more program file, but loading it puts its code inside the kernel. A faulty module can damage the whole system; it does not get an application’s private-memory protection simply because it was loaded separately. We begin working with modules in Chapter 36.

2.7 Linux storage and memory use

All these layers raise a fair MCU-engineer question: does this small board need the memory budget of a desktop? A desktop installation includes many programs our board will not need. An embedded system selects the kernel features, libraries, and applications for its particular job. The useful comparison is between those selected pieces, not between the board and your entire PC installation.

There is no single “Linux memory budget” to copy into a design. The table locates the costs we will later measure; it is not an additive size estimate. Build options, compression, stripped/debug files, and the workload change the numbers.

Component	Resource	What determines the cost
`zImage`	Stored compressed boot file	Kernel configuration and compression
Running kernel	RAM	Loaded sections, kernel allocations, and workload; these categories can overlap
Device tree blob	Stored input file	Hardware description and included nodes
BusyBox	Stored executable	Selected applets, libc, and static versus shared linking
libc	Stored library or code linked into executables	Implementation, ABI, build options, and stripping
Root filesystem	Stored tree/archive/filesystem	Programs, libraries, configuration, compression, and filesystem overhead
Running user space	RAM	Processes, private/shared mappings, and working data

The MINI core used here has 512 MiB of DRAM. That gives us room to learn without treating the first build as a size contest. A smaller product needs measurements of its actual image, boot-time peak, and running workload, not a sum of unrelated example ranges.

User-space libraries and frameworks can use more storage than the kernel. Common examples are glibc, the C++ runtime, Qt, and Python. This is why Yocto and Buildroot spend much of their work selecting and packaging user-space components.

2.8 What the rest of this book builds, in order

The following table shows the first major stages of the book and where each artifact belongs in the four-layer model.

Chapter range	Layer	Artifact
3-8	host / 0 / 1	Workspace and understanding of Boot ROM and IVT/DCD
9-17	**our bare-metal code** as a Layer 2 substitute	LED, DDR, and MMU. Your code performs the bootloader’s early work.
18	optional	bare-metal I²C/SPI/LCD
19-24	Layer 2	U-Boot from source, ported and understood
25-30	Layers 3 and 4	Linux kernel and Chapter 29’s first minimal userspace/initramfs
31-35	Layer 4	Expand the root filesystem and userspace, then automate with Buildroot
36-55	Layer 3	Device drivers and kernel subsystems
Later parts	all	Debugging, product development, build systems, security, and advanced topics

Return to the four-layer diagram when a later experiment fails. If the Boot ROM has not loaded a valid image, changing an application cannot help. If Linux is running but a device cannot be opened, the image header is probably not the first place to look. The diagram gives us a way to choose the next question.

2.9 Before we move on

We have not configured a GPIO yet, but we can explain why the opening register-writing function cannot simply be carried into a Linux application. Its pointer belongs to a virtual address space, and the physical register is not automatically mapped there. The application normally asks a driver for the operation. A library wrapper, a syscall, and an fd are now recognizable parts of that request rather than three unrelated names.

At the end of the route, someone still writes the GPIO register. Linux has changed who may request it and how the device is shared, not the electrical job of turning on the LED. Keep that distinction. We will return to this same hardware operation when we write bare-metal code and, much later, a Linux driver.

2.10 Key Notes

- Boot is a sequence of ownership transfers: the ROM selects and loads an image, the bootloader prepares Linux’s inputs, and the kernel starts user space. Ordinary application requests do not return through the bootloader.
- Root credentials affect access policy, not CPU execution mode. A syscall enters a privileged kernel handler and returns the application to user mode with a result or error.
- Virtual pointers and file descriptors derive their meaning from the process’s mappings and descriptor table. Equal numbers in unrelated processes need not identify the same bytes or open object.
- Drivers own supported hardware operations and resource arbitration. A physical register address alone supplies neither an application mapping nor permission, and separate requests do not automatically form an atomic device protocol.
- A minimal root filesystem must supply PID 1 and its runtime dependencies. Static linking can remove the shared-loader requirement, but it does not supply UART support, a console, or permission for PID 1 to exit normally.
- Process isolation normally contains an invalid user-space access. Kernel modules run inside the kernel, where a fault can corrupt shared state or stop the system. Deliberately crashing the host is not a safe way to explore that difference.
- Stored images, virtual address ranges, and working RAM are different budgets. Their costs depend on the selected kernel, libraries, processes, and workload rather than one universal Linux size.

2.11 Pitfalls

- **Looking for an “embedded mode” in Linux.** We configure the kernel for the target and choose a suitable userspace: perhaps fewer daemons, a smaller libc, and a read-only root. There is no single switch that turns a desktop distribution into the system our board needs.
- **Treating `bootz` as proof of Linux entry.** A failed command can return to the U-Boot prompt. After a successful handoff, ordinary U-Boot code is no longer in control; Linux does not use it as a runtime service. Data such as the device tree can remain in memory.
- **Believing `/proc/cpuinfo` always reflects physical hardware.** It reports what the kernel detected or was told through the device tree. A virtual machine such as QEMU may report virtual hardware instead.
- **Trying to debug user-space problems with kernel tools and vice versa.** Each layer has its own toolset. First identify which layer contains the bug, then choose tools for that layer.

2.12 Going deeper

Names and filesystem objects, optional reference

A filename is how we look something up. Two names can refer to the same object through hard links. An **inode** is the kernel’s representation of that filesystem object, including its type, permissions, and owner. Directory entries connect names to inodes.

Keep this separate from an open file. An open *file object* records state such as the current offset, and an fd refers to that open object. How an inode locates content depends on the filesystem; a procfs entry, for instance, need not correspond to blocks on a disk.

An application does not need to ask whether a file is on ext4, FAT, or tmpfs before reading it. The **virtual filesystem**, or VFS, provides the common kernel interface above those filesystems, as well as procfs and devtmpfs. Each supplies operations the VFS can call. Character and block devices also connect to this framework.

Further reading

- *The Design of the Unix Operating System*, Maurice Bach (1986), for historical background on Unix processes and filesystem design. Its implementation details are not current Linux behavior.
- *Linux Kernel Development*, Robert Love (3rd ed., 2010), for a high-level kernel tour. Compare implementation details with the selected Linux source version.

- execve(2), for how an executable file becomes the program image of a running process.
- Start with `man 2 intro`, `man 2 open`, `man 2 read`, `man 2 mmap`, and `man 7 pthreads`. See mmap(2) and pthreads(7) for the mapping and shared-resource distinctions above.
- The Linux source tree’s `Documentation/admin-guide/` and `Documentation/process/`.

The early examples use a host cross-compiler to produce the board’s programs. Linux can also run a native compiler on a sufficiently equipped target, but that is not our initial route. A compiler name in a host terminal is only a starting point: which compiler file does the shell actually find? Chapter 3 makes that choice visible.

Chapter 3: Host environment setup

A new terminal can look exactly like the last one and still choose a different compiler. It can even report “command not found” while the compiler files are sitting where you left them. The files and the terminal’s knowledge of them are two different things.

An MCU IDE usually keeps that distinction out of sight by choosing tools with the project. Here we make the choice ourselves. The computer on your desk is the **host**: it edits and builds the code, inspects the result, and sends it to the i.MX6ULL **target** board. Our first job is to give the files a known home and make each terminal’s tool selection visible.

Part II’s examples use a workspace, two compilers, a serial-terminal program, and a USB download tool. All can be prepared with the board disconnected. The sections below separate project-local tool selection from hardware access and persistent host services. Reading the examples does not require installing or operating them:

When	Sections	Result to keep
Host-only tool selection	3.1-3.4, then 3.5’s introduction, 3.8, 3.11, and 3.12	Compiler paths, a project-local uuu, and terminal-local exports
Chapter 8	The serial/ROM reference commands in 3.5 and 3.8	Board connections and device identities, after the power checks
Before Chapter 11	3.9	An identified spare card; no write in this chapter
Network-boot examples, starting in Chapter 24	3.10 first, then 3.6 and 3.7	A stable address plan and deliberately started development servers

The Windows-tool alternatives in 3.5a-3.5b are optional. Neither an SD card nor a network server is a prerequisite for Chapter 9’s USB-loaded LED experiment.

3.1 Choosing the host

We use **x86_64 Ubuntu 22.04 LTS Desktop** and its default Bash terminal as the reference host. The compiler archives in this chapter run on x86_64 Linux, so they cannot be launched directly in Windows or on an ARM64 host. Start with Ubuntu installed, either natively or in a dedicated virtual machine; installing the OS is outside this chapter. Ubuntu’s Desktop installation tutorial explains the installer, but select the

reference 22.04 release rather than assuming every current release has identical packages. Server/minimal installations may use a different network manager and are not the reference recipe here.

Open Ubuntu's Terminal application. This is where the host commands in the book belong. A `$` at the start of a listing is a prompt marker, not a character to type; following lines without that marker are example output, except where a command continues onto the next line. Most commands run as your normal user. When a command needs authorization to change a system file or access a restricted device, we show `sudo` explicitly. It may ask for your login password; the terminal displays no characters while you type it.

Check the installed host before changing packages:

```
$ uname -m
x86_64
$ cat /etc/os-release
... Ubuntu 22.04 ...
```

`uname -m` reports the host machine architecture; `cat` displays a file's contents. The second listing is abbreviated: look for Ubuntu and `VERSION_ID="22.04"` in your actual output. If these differ, resolve the host choice before following this recipe.

A dedicated VM keeps package and service changes inside that lab system. Plan for these connections, but do not attach the board in Chapter 3:

- USB attachment to the guest for the serial bridge and ROM SDP device; a card reader is needed later for SD boot. *Attachment* means giving Ubuntu, rather than Windows, ownership of that USB device. Chapter 8 checks its appearance inside Ubuntu and reattachment after a USB identity changes.
- A bridged or passed-through Ethernet adapter on the board's lab link. Default VM NAT alone does not make guest TFTP/NFS servers reachable by the board.
- Only one owner for each serial port: close a Windows terminal before attaching its bridge to the guest.

We can keep compiler files and compiler selection local to the project. Ubuntu packages and the optional service settings under `/etc` are different: they change the host system and remain after the terminal closes. Use the dedicated VM if you want those changes kept off your everyday installation. The later NFS recipe requires a disposable lab VM with no personal files or shared host folders. A validated power guide for the exact baseboard/core revision is still required before Chapter 8; the supplied schematic alone is not a universal power/jumper recipe.

3.2 Workspace layout

Consider the next time a build fails and you need to find the source you edited. It helps if that file is not buried among yesterday's generated objects and downloaded archives. We use `~/imx6ull` to keep those roles separate. The layout below will stay with us as the projects grow:

```
$ mkdir -p ~/imx6ull/{src,build,boot,rootfs,scripts,toolchains,notes}
$ cd ~/imx6ull
$ ls
.
├── boot       # bootable artefacts staged here, then dd'd to SD
├── build      # all out-of-tree build outputs (kernel, U-Boot, BusyBox)
├── notes      # configuration and diagnostic records
├── rootfs     # exported over NFS to the target
├── scripts    # helpers, shared between chapters
├── src        # upstream sources: linux, u-boot, busybox, your bare-metal code
└── toolchains # prebuilt Arm compilers kept local to this project
```

Read the first command before entering it. ~ means your home directory, and Bash expands `{src,build,...}` into one path for each name. `mkdir -p` creates those directories if they are missing. Then `cd` moves this terminal into the workspace and `ls` lists what is there. The annotated tree shows what the names mean; ordinary `ls` does not print that tree or its comments.

You will also see `$HOME`, the shell-variable form of your home-directory path. It lets the environment file below refer to your directory without embedding a particular username. A path without a leading `/` is relative to this terminal's current directory; `pwd` prints that directory. A `#` starts a shell comment, such as the notes in the illustrative tree.

Two distinctions will save confusion later:

1. **Sources and outputs have different roles.** Edit tracked sources in `src/`, including your own bare-metal files. Git records source changes against upstream. Where a project supports `O=`, place generated output in `build/`; this does not move or apply source patches for you.

2. **`rootfs/` is the future NFS root.** In the later NFS-root example, Linux uses files from this exported host directory without reflashing storage. Creating the directory alone does not configure NFS or boot the target.

3.3 Host packages

Not every program used during an Arm build runs on Arm. The host also runs tools that process configuration, generate source, and package files. Ubuntu supplies these helpers through its package manager, `apt`. The long installation below prepares those host jobs; our two Arm compilers will still be downloaded separately into the workspace.

`apt update` refreshes the package catalog; `apt install` installs the named packages and their dependencies. Read the proposed installation before answering its confirmation prompt. Packages ending in `-dev` generally supply headers and libraries needed to compile another program. These installations change Ubuntu, rather than just the project directory.

The trailing `\` below joins the next line to the same command. It must be the last character on the line, with no space after it; the indented package names are continuation lines to enter, not output. Run each command in order and resolve errors before entering dependent commands. A returned prompt alone does not mean success.

```
$ sudo apt update
$ sudo apt install \
    build-essential bison flex libssl-dev libncurses-dev \
    bc kmod cpio rsync wget curl git unzip xz-utils xxd \
    device-tree-compiler u-boot-tools \
    minicom picocom \
    qemu-user-static binfmt-support \
    gdb-multiarch \
    pkg-config libusb-1.0-0-dev libftdi1-dev \
    libgmp-dev libmpfr-dev libmpc-dev libisl-dev \
    fakeroot dosfstools mtools parted nano usbutils python3
```

The list is long because it covers several later builds. You do not need to learn every package now, but the main groups have recognizable jobs:

- **`build-essential`, `bison`, `flex`, `libssl-dev`, `libncurses-dev`** provide tools and libraries needed to build the kernel and U-Boot. The kernel uses OpenSSL during build for features such as module signing.
- **`bc`** provides arithmetic used by parts of the kernel build.
- **`xxd`** prints a file's bytes in hexadecimal; Chapter 7 uses it to inspect a boot image.
- **`device-tree-compiler`** provides `dtc`, which compiles a hardware description for the kernel: devices, their addresses, and how they are connected.
- **`u-boot-tools`** provides `mkimage`, `mkenvimage`, `dumpimage`, and `mkeficapsule`.
- **`nfs-kernel-server`, `tftpd-hpa`** provide network boot services; install them later in Sections 3.6-3.7 when needed, because package installation can start daemons.
- **`minicom`, `picocom`** are serial terminals. This book uses `picocom`.
- **`qemu-user-static`, `binfmt-support`** let the host emulate ARM user-space programs, not boot an ARM kernel. They become useful during root-filesystem preparation with `chroot`, which gives a process a different top-level directory for file lookup.
- **`gdb-multiarch`** is a GDB build that supports multiple architectures, including ARM.

- **`libusb-1.0-0-dev`, `libftdi1-dev`** are needed when building USB and JTAG tools such as `imx_usb_loader` and OpenOCD.
- **OpenOCD** is the host program that controls a JTAG adapter and exposes a GDB server.
- **`fakeroot`, `dosfstools`, `mtools`, `parted`** manipulate filesystem and SD-card images.

If `apt` cannot find a package on the reference Ubuntu release, stop and record the exact error and release. Check that `apt update` succeeded; do not substitute a similarly named package without checking what it provides.

3.4 The cross toolchain

The name `gcc` tells you which compiler family you called, not which processor its output will run on. Ubuntu's ordinary `gcc` builds for the host PC. For our board, we need a compiler that runs on the PC but produces Arm instructions. That is a **cross-compiler**. Its linker, assembler, and inspection tools form a **toolchain**. Chapter 6 will make this difference visible by building the same C source for two targets.

We use two official prebuilt Arm toolchains because the target environments differ:

- **Linux target toolchain:** `arm-none-linux-gnueabihf-` Builds U-Boot, the Linux kernel, BusyBox, and target user-space programs. It targets 32-bit Arm Linux with the hard-float glibc ABI.
- **Bare-metal toolchain:** `arm-none-eabi-` Builds the small no-OS experiments in Part II. It does not assume Linux, glibc, processes, or a dynamic loader.

Both archives go into the workspace, with their original directory names. We will not install another Arm compiler through `apt` or rename these folders. Keeping the two paths visible makes it easier to check which tool a build actually used.

Use **13.2.Rel1** for both targets in this edition. A fixed release gives the examples a consistent starting point; it is not a claim that every example has been tested on hardware with that release. The 13.2 release notes identify the host and target packages. The archive directory in this layout is:

```
$ mkdir -p ~/imx6ull/src/toolchains
$ cd ~/imx6ull/src/toolchains
```

In Ubuntu's browser, download these four files from Arm and save them in that directory, retaining their exact names. If the browser saves to `~/Downloads`, move them with the file manager. These release-specific links avoid having to find an older release on Arm's changing download index:

- Linux archive and its SHA256 manifest.
- Bare-metal archive and its SHA256 manifest.

Before extraction, return to the archive directory and check both files. `sha256sum --check` reads the filename and expected hash from a manifest, computes that archive's hash, and reports whether they match:

```
$ cd ~/imx6ull/src/toolchains
$ sha256sum --check arm-gnu-toolchain-13.2.rel1-x86_64-arm-none-linux-
gnueabihf.tar.xz.sha256asc
arm-gnu-toolchain-13.2.rel1-x86_64-arm-none-linux-gnueabihf.tar.xz: OK
$ sha256sum --check arm-gnu-toolchain-13.2.rel1-x86_64-arm-none-eabi.tar.xz.sha256asc
arm-gnu-toolchain-13.2.rel1-x86_64-arm-none-eabi.tar.xz: OK
```

Stop if either reports a missing file or mismatch; do not extract it. Check the download and matching manifest, then repeat verification. Keep both archives and manifests for your installation record. Here and later, `<you>` or `<version>` in illustrative output means a value to substitute, never literal angle brackets to type in a command.

Then extract them into `~/imx6ull/toolchains`, not a system directory such as `/opt`. The compiler files remain part of the workspace, and removing this workspace does not remove a compiler another project installed globally.

```
$ mkdir -p ~/imx6ull/toolchains
$ tar -xf arm-gnu-toolchain-13.2.rel1-x86_64-arm-none-linux-gnueabihf.tar.xz \
    -C ~/imx6ull/toolchains
$ tar -xf arm-gnu-toolchain-13.2.rel1-x86_64-arm-none-eabi.tar.xz \
    -C ~/imx6ull/toolchains
```

After extraction, keep the directory names Arm supplied, including their capitalization. The two compiler executables are inside the corresponding `bin` directories:

```
~/imx6ull/toolchains/arm-gnu-toolchain-<version>-x86_64-arm-none-linux-gnueabihf/bin/arm-
none-linux-gnueabihf-gcc
~/imx6ull/toolchains/arm-gnu-toolchain-<version>-x86_64-arm-none-eabi/bin/arm-none-eabi-
gcc
```

If a later build reports the wrong compiler, these are the paths to check first.

Decoding the triplets

The long prefixes tell us more than "this is an Arm compiler." They describe the environment the output is intended to use. **ABI**, or Application Binary Interface, means the agreement about calling conventions, register use, binary format, and runtime libraries that lets separately built code work together.

- `arm` means the target CPU family is 32-bit Arm.
- The middle `none` is the vendor field. Here it means no specific silicon vendor.
- `linux` means the generated program expects a Linux target environment.
- `gnu` means GNU userland and glibc ABI.

- `eabi` means Embedded ABI v5.
- `hf` means hard-float: floating-point arguments are passed in VFP registers.

You do not need to reconstruct these names at every build. For this book, the selection is:

- Use `arm-none-linux-gnueabihf-` for this book's U-Boot, kernel, and Linux user-space builds. Only user-space builds use its glibc runtime; U-Boot/kernel choose their own freestanding build rules.
- Use `arm-none-eabi-` for our no-OS Part II experiments. Keep these two selections explicit even though some freestanding objects can be ABI-compatible across compiler families.

Environment script

Bash finds external programs by searching the directories listed in `PATH`, not the whole disk. One small file, `env.sh`, adds our compiler directories for the current terminal.

Leave `~/.bashrc` unchanged. Putting the setup there would select these tools automatically in future terminals, including terminals used for other work. Instead, we will make the selection visible each time we begin a book session.

Create the file with the editor:

```
$ nano ~/imx6ull/scripts/env.sh
```

Enter the following contents. The first part names the two selected release directories and checks their compiler files; the remaining lines give this terminal the paths and names used by later builds:

```
#!/bin/bash

IMX6ULL_HOME="$HOME/imx6ull"
linux_dir="$IMX6ULL_HOME/toolchains/arm-gnu-toolchain-13.2.Rel1-x86_64-arm-none-linux-
gnueabihf"
bare_dir="$IMX6ULL_HOME/toolchains/arm-gnu-toolchain-13.2.Rel1-x86_64-arm-none-eabi"

if [ ! -x "$linux_dir/bin/arm-none-linux-gnueabihf-gcc" ]; then
    echo "Missing executable Linux compiler in $linux_dir/bin."
    return 1
fi
if [ ! -x "$bare_dir/bin/arm-none-eabi-gcc" ]; then
    echo "Missing executable bare-metal compiler in $bare_dir/bin."
    return 1
fi

export IMX6ULL_HOME
export ARM_LINUX_TOOLCHAIN="$linux_dir"
export ARM_BAREMETAL_TOOLCHAIN="$bare_dir"
```

```
export PATH="$ARM_LINUX_TOOLCHAIN/bin:$ARM_BAREMETAL_TOOLCHAIN/bin:$IMX6ULL_HOME/build/
mfgtools/uuu:$PATH"

export ARCH=arm
export CROSS_COMPILE=arm-none-linux-gnueabihf-
export BAREMETAL_CROSS_COMPILE=arm-none-eabi-

export TFTPROOT=/srv/tftp
export NFSROOT="$IMX6ULL_HOME/rootfs"

# Default direct-link lab addresses. If you use your home/office router
# instead, replace these with the real LAN addresses from Section 3.10.
export BOARD_IP=192.168.7.2
export HOST_IP=192.168.7.1
```

Use this file with **Bash**, not `sh`. Notice the capital `R` in the extracted `13.2.Rel1` directory names; the archive filenames use lowercase `13.2.rel1`. Linux paths are case-sensitive. We keep Arm's names unchanged. These exact paths select this edition's release even if another release is present. If a check fails, resolve the missing extraction or executable before building. The file stops before changing `PATH`; it does not clear an earlier selection in this terminal.

There is no need to learn Bash scripting in full to understand this file. Read its parts in order:

- `name=value` assigns a variable; `$name` reads it. Quotes keep a path with spaces together.
- `[ ... ]` tests a condition. `-x` checks for an executable and `!` means "not". The `if` body therefore runs when that compiler is missing or not executable. `return 1` stops this sourced file with failure.
- `export` makes a variable available to programs started by this terminal. `PATH` is the colon-separated list of directories searched for commands, in order. The `uuu` directory is populated in Section 3.8.
- `TFTPROOT` and `NFSROOT` are convenient shell names; they do not configure the servers. `/etc` configuration must still be edited manually.

In `nano`, press Ctrl-O and then Enter to save. Press Ctrl-X to return to the terminal.

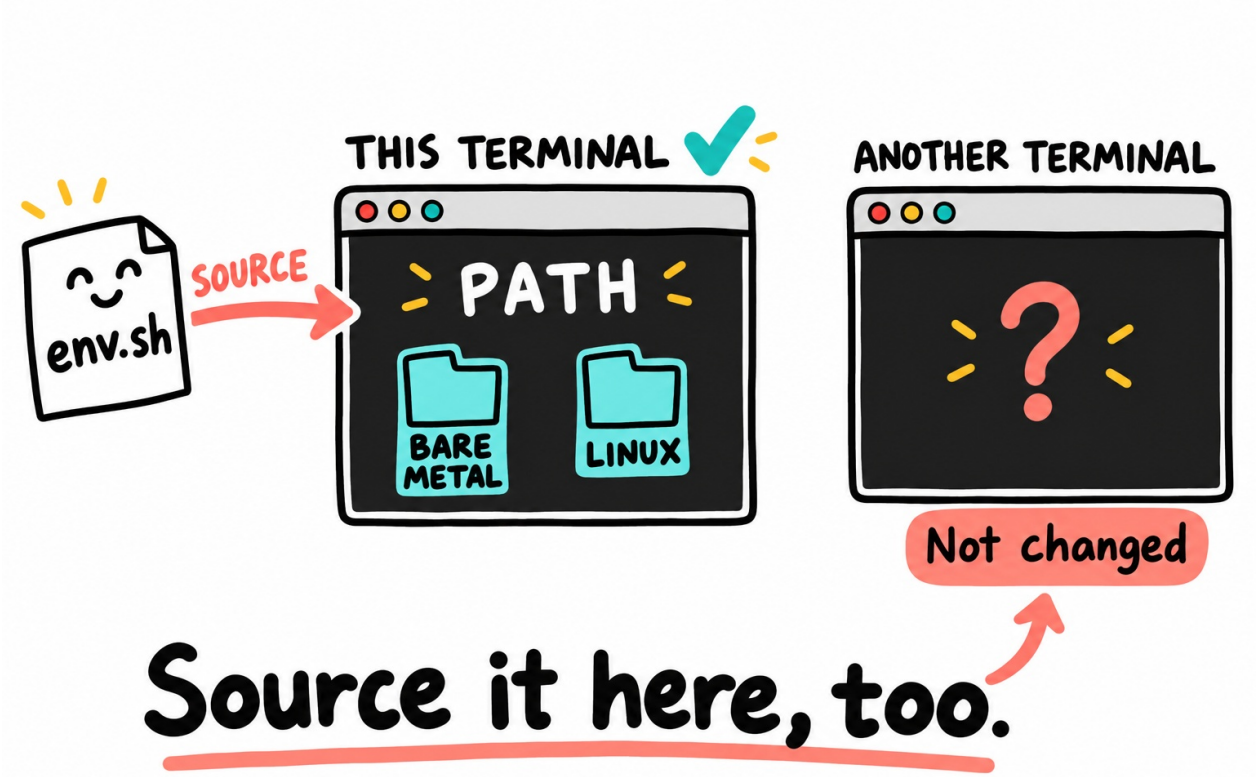
Every time you open a new terminal for this book, run:

```
$ . ~/imx6ull/scripts/env.sh
$ echo "$?"
0
```

The leading dot matters. It tells Bash to read the file **into this shell**, which is called *sourcing* the file. Its exported values then apply to commands you run in this terminal. `$?` is the previous command's exit status: zero means success here. Read it immediately after sourcing, before another command replaces it. If it is nonzero, stop, fix the named problem, and start a fresh terminal before trying again.

Source once when starting a terminal. Repeating it in the same terminal adds duplicate PATH entries but does not reinstall anything; after changing toolchain paths, use a fresh terminal rather than trying to clean up a stale PATH by hand.

If you instead enter `bash ~/imx6ull/scripts/env.sh`, the file runs in a child shell. That child's settings cannot change the terminal you return to afterward. We also do not need to make `env.sh` executable: sourcing reads it as a file rather than launching it as a separate program.



The folder symbols stand for directory names in PATH, not files copied into a terminal. Both terminals can access the same compiler files on disk. Enter `. ~/imx6ull/scripts/env.sh` in each independently opened terminal. A child shell can inherit exported settings, but a shell that was already open will not receive the change.

After building `uuu`, use the single path/version acceptance block in Section 3.11. `command -v` reports what this shell would select, which can be a program, alias, or function. For our compilers, expect full paths into the selected workspace directories. If the answer is surprising, `type -a` lists all definitions and locations of that command name.

The trailing hyphen in each prefix is intentional. Build systems append names such as `gcc` or `objcopy` to it. U-Boot, Linux, and BusyBox use `CROSS_COMPILE`; our own bare-metal Makefiles use the separate `BAREMETAL_CROSS_COMPILE` name.

3.5 Serial console

The serial console will be our way to see what the board prints. The MINI already has a USB-to-TTL bridge connected to its debug UART, so normal setup does not need an external adapter or jumper wires. Use the port labeled **USB-TTL** or **DEBUG USB**, not USB-OTG.

For now, the serial-terminal package is enough. **Leave the board disconnected.** Return to the commands below during Chapter 8, after its power-arrangement check. A USB cable carries power as well as data; installing a terminal does not make a guessed power connection safe. For the host-only route, continue at Section 3.8.

After connecting it, check which serial device appeared:

```
$ ls /dev/ttyUSB*
/dev/ttyUSB0
```

If that file does not exist, check `dmesg | tail`:

```
$ sudo dmesg | tail
[...] usb 1-1.2: new full-speed USB device number 5 using xhci_hcd
[...] usb 1-1.2: New USB device found, idVendor=1a86, idProduct=7523
[...] ch341 1-1.2:1.0: ch341-uart converter detected
[...] usb 1-1.2: ch341-uart converter now attached to ttyUSB0
```

This is illustrative CH340/ch341 output for the reference bridge. Other adapters may use `cp210x` or another driver. `|` sends `dmesg` output to `tail`, which selects the last lines; do not type the displayed log as commands. Match the newly attached bridge to its actual node, using the attachment log and `/dev/serial/by-id/` where available. The suffix `0` is not a board identity; substitute the observed node in the command below.

Finding the device and having permission to open it are separate steps. On this host, access is normally restricted to root and a group such as `dialout`. We leave your groups unchanged and show `sudo` for each serial session. Ubuntu may also restrict the kernel log read by `dmesg`.

```
$ sudo picocom -b 115200 /dev/ttyUSB0
picocom v3.1
port is        : /dev/ttyUSB0
flowcontrol    : none
baudrate is    : 115200
parity is      : none
databits are   : 8
stopbits are   : 1
```

```
...
Terminal ready
```

Quit with Ctrl-A then Ctrl-X. Plain Ctrl-C sends the interrupt character to the target; Ctrl-A then Ctrl-C instead toggles local echo. These keys follow picocom's manual.

"Terminal ready" means the host opened the port; it does not promise a message from the board. Our own bootable image does not exist yet, so silence is normal. A pre-installed factory image may print something; save that output if it does. Later, when our UART program is supposed to print, unreadable text is a reason to check 115200 8N1, the device path, and the USB cable.

If a board revision has no built-in bridge, an appropriate **3.3 V** USB-TTL adapter is an alternative. Use the matching header definition: board TX to adapter RX, board RX to adapter TX, and GND to GND; leave adapter VCC disconnected. VCC is not the only back-powering path: a driven TX/RX line can feed an unpowered board through its input protection. Disconnect the adapter for a cold cycle and follow the supplier's permitted connection order; do not drive an unpowered endpoint.

3.5a Windows-side serial terminals (for Windows-mainly readers)

For an optional Windows-side serial session, PuTTY provides a serial connection: select the bridge's COM port in Device Manager, choose Serial, and set 115200 8N1 with no flow control. This does not substitute for attaching USB devices to Ubuntu when using the Linux build/transfer commands.

Other optional terminals include:

- **MobaXterm** (`mobaxterm.mobatek.net`, free Home Edition) combines SSH, serial, X server, saved sessions, and SFTP.
- **SecureCRT** (`vandyke.com`, commercial) provides fast scrollback, saved sessions, and a configurable keymap.
- **PuTTY** (`putty.org`, free) provides a small serial and SSH client.
- **Tera Term** (`teratermproject.github.io`, free) provides serial access and a macro language.

The board's integrated bridge may use a CH340 or CP2102 device. Windows may need the corresponding driver from `wch.cn` or `silabs.com`. Linux normally includes both drivers.

When configuring any of these tools, the settings are the same we used for `picocom`: **115200 8N1, no flow control**.

3.5b Source Insight as a kernel-source navigation aid (optional)

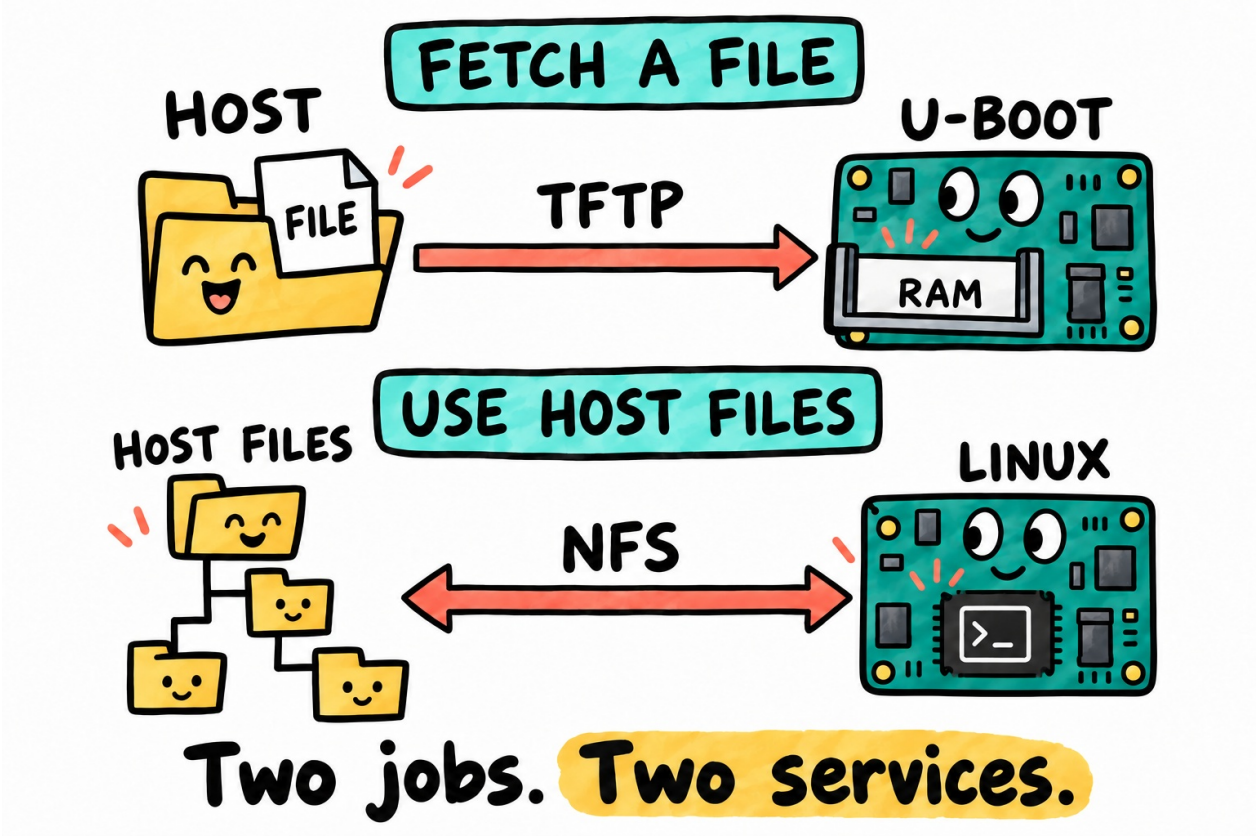
When we reach the kernel sources, following a function across many files can be slow in an ordinary editor. A source-navigation tool can help with that later; it is not another prerequisite for host setup.

- **Source Insight 4** (`sourceinsight.com`, commercial Windows) provides source indexing, Go to Definition, and call graphs.
- **VS Code + C/C++ extension** (Microsoft) is free and cross-platform. Use `compile_commands.json` from a kernel build so IntelliSense follows the correct include paths.
- **`cscope` + `ctags`** provide terminal-based source navigation and can be scripted.
- **`elixir.bootlin.com`** provides a web-based Linux source cross-reference without a local installation.

Use whichever of these you already know, or defer the choice until you need it. For the small files in Part I, `nano` is enough.

3.6 TFTP server

Once U-Boot is running, think about the next ten kernel builds. Moving the card from board to host and back for every image soon becomes more work than the small code change you wanted to test. **TFTP** lets U-Boot fetch the file from the host over Ethernet instead. This section prepares that host service; it can wait until Chapter 24.



A TFTP file travels to the board. With NFS, the directory stays on the host and the board accesses its files over the network. Linux may cache some of that data locally. These are the roles we choose for the later labs, not exclusive capabilities of either protocol. Neither service is needed for the first LED experiment.

Unlike `env.sh`, a network service continues independently of this terminal. Keep a record of the existing configuration, preserve unrelated settings, and use an isolated lab link. Work through the address plan in Section 3.10 before restarting a server bound to that address. Do not expose these services on a public or shared network.

Record `systemctl is-active tftpd-hpa` and `systemctl is-enabled tftpd-hpa` before installation. A missing unit is normal on a fresh host. A server package can start immediately and enable future startup; `stop` alone does not undo that. Install on the isolated lab system, not a machine running a shared TFTP service. For a newly installed, lab-only server, stop it and disable automatic startup before editing its configuration:

Install the server package if you have not already:

```
$ sudo apt install tftpd-hpa tftp-hpa
$ sudo systemctl disable --now tftpd-hpa
```

On the reference host, systemctl also coordinates this package's legacy startup arrangement. Record the command's confirmation and inspect `is-active`/`is-enabled` again; do not infer reboot behavior from current inactivity. Restore the original state if the service predated this example.

Now open the server configuration:

```
$ SUDO_EDITOR=nano sudoedit /etc/default/tftpd-hpa
```

`SUDO_EDITOR=nano` chooses the familiar editor for this command only. `sudoedit` lets you edit a temporary copy as your normal user, then writes it back with authorization. Save and exit with the same nano keys used earlier; your shell's default editor is not changed.

Set the four fields as follows for the direct-link address plan:

```
TFTP_USERNAME="tftp"
TFTP_DIRECTORY="/srv/tftp"
TFTP_ADDRESS="192.168.7.1:69"
TFTP_OPTIONS="--secure"
```

Each line answers one setup question:

- `TFTP_USERNAME="tftp"` runs the daemon as the unprivileged `tftp` user.
- `TFTP_DIRECTORY="/srv/tftp"` is the directory U-Boot will read files from.
- `TFTP_ADDRESS="192.168.7.1:69"` listens only on the host lab address and standard UDP port. Configure that address using Section 3.10 before restarting; router mode must use the actual host address.
- `TFTP_OPTIONS="--secure"` roots requests inside `/srv/tftp`. This book downloads existing files, so it does not enable new-file uploads with `--create`; that option would not grant filesystem write permission anyway.

Create the directory, make your normal user its owner, and keep it readable by the TFTP daemon:

```
$ sudo mkdir -p /srv/tftp
$ sudo chown "$USER:$(id -gn)" /srv/tftp
$ chmod 755 /srv/tftp
```

There are two users involved here, and they want different things. You need to put new images in the directory. The `tftp` service needs to read them, but does not need your authority to change the host. The ownership and permission commands arrange that distinction:

- `/srv` is a system directory. Without `sudo`, a normal user usually cannot create `/srv/tftp`.
- After `sudo mkdir`, the new directory is owned by `root`, so your normal user would need `sudo` every time you copy a kernel, device tree, or U-Boot image into it.
- `sudo chown "$USER:$(id -gn)" /srv/tftp` selects your user and actual primary group. `$()` substitutes command output; `id -gn` prints that group. Now you can stage files with normal `cp` commands. Change ownership only for this dedicated lab directory, not an existing shared service directory.

- The TFTP server does not run as your user. `TFTP_USERNAME="tftp"` means it runs as the low-privilege `tftp` user, so a bug in the TFTP server has less power on the host.
- `chmod 755 /srv/tftp` grants the owner read/write/execute and others read/execute. For a directory, read lists names and execute permits traversal to a named file; the file itself must separately be readable. Other ordinary users cannot add files, though root still can.

Files you copy into `/srv/tftp` also need to be readable by the TFTP daemon. Normal files created by `cp` or `echo` are usually readable already. If U-Boot gets “permission denied” from TFTP, check with:

```
$ ls -l /srv/tftp
```

If a staged file is not readable by `tftp`, change that file’s mode, not the whole tree. The smoke test below demonstrates `chmod 644`: owner read/write, others read.

With the address and configuration checked, start the service deliberately:

```
$ sudo systemctl start tftpd-hpa
$ systemctl is-active tftpd-hpa
active
```

Before involving the board, retrieve one small file from the host’s own server:

```
$ echo "hello tftp" > /srv/tftp/test.txt
$ chmod 644 /srv/tftp/test.txt
$ mkdir -p ~/imx6ull/build/tftp-check
$ cd ~/imx6ull/build/tftp-check
$ tftp 192.168.7.1 -c get test.txt
$ cat test.txt
hello tftp
```

`echo` prints the text; the shell’s `>` writes it to a file, replacing existing contents. Use this dedicated test filename. The TFTP client saves its download in this terminal’s current directory, then `cat` displays it. Use your actual host address in router mode.

If this works, the local service can read and serve the file. The board-to-host cable, firewall path, and VM reachability still need a target transfer. We make that check after U-Boot runs in Chapter 24.

Firewall: keep the existing firewall enabled. Inspect `sudo ufw status`; where UFW is active, allow the board only on the dedicated lab interface, substituting the interface name found in Section 3.10:

```
$ sudo ufw allow in on enp0s31f6 from 192.168.7.2 to 192.168.7.1 port 69 proto udp
```

TFTP uses additional UDP transfer ports; UDP/69 alone is not proof that the whole exchange is allowed. If a restart or transfer fails, first check the host address, then `systemctl status tftpd-hpa --no-pager` and `sudo journalctl -u tftpd-hpa -n 30 --no-pager`. Next inspect the file permissions, VM attachment, and firewall policy. Do not disable workstation protection to make the symptom disappear.

After a development session, stop the server and recheck both its active and enabled states. Restore only the prior configuration and firewall rules you changed. A service installed just for these examples should remain disabled; an existing shared service needs its recorded original state instead.

3.7 NFS server

TFTP saves us a trip with the SD card, but it still transfers individual files. What if Linux could use a directory on the host as its root filesystem? **NFS**, the Network File System, allows that. During development, the host can hold the target's files, and the board can read an updated file without needing a new SD-card image.

The later kernel and root-filesystem chapters explain NFS-root boot. This host-side example describes which directory the board may access and the trust that export grants.

Use a disposable lab VM for this recipe, with no personal files or shared host folders. During installation, use only the VM's outbound NAT connection without incoming port forwards, not a bridged public/shared uplink. Before starting the lab server, disconnect that uplink and use the dedicated target link, or an isolated lab router. Do not run this trusting development export on your everyday home/work LAN.

Record the existing active/enabled states of `nfs-kernel-server` and related RPC units. If they serve other work, use a separate VM rather than reconfiguring them. Packages can start services immediately; on a new lab-only VM, stop and disable the newly installed server before configuration:

Install the server package if needed:

```
$ sudo apt install nfs-kernel-server
$ sudo systemctl disable --now nfs-kernel-server
```

Open the export table:

```
$ SUDO_EDITOR=nano sudoedit /etc/exports
```

Add one line at the end. Replace `<you>` with your Linux username:

```
/home/<you>/imx6ull/rootfs 192.168.7.2(rw,sync,no_root_squash,no_subtree_check)
```

Then apply and verify:

```
$ sudo exportfs -ar
$ sudo systemctl restart nfs-kernel-server
$ sudo showmount -e localhost
Export list for localhost:
/home/<you>/imx6ull/rootfs 192.168.7.2
```

What the commands do:

- `exportfs -ar` asks the NFS server to re-read `/etc/exports` and apply the export table.
- `systemctl restart nfs-kernel-server` restarts the NFS daemon so the kernel-side service is using the current config.
- `showmount -e localhost` checks the advertised export, not whether the target can mount or boot it. Target acceptance also depends on the kernel, network path, boot arguments, and root filesystem described later.

The options in the export line determine what that access means:

- `rw` lets the target write to the exported filesystem.
- `sync` commits writes before the server replies. This is slower but reduces the chance of losing recent writes.
- `no_root_squash` maps the target's root user to host UID 0. This is convenient for a development root filesystem but unsafe on an untrusted network.
- `no_subtree_check` avoids subtree checks that can complicate renamed-file handling. It is a development tradeoff, not a claim that the directory is a secure boundary.

Security: this recipe trusts the target. Its IP address is not authentication, and `no_root_squash` accepts its root identity as host UID 0. A subdirectory export is not a strong sandbox; filehandle attacks can reach outside the intended tree under some configurations. Do not reassure yourself that personal files outside `rootfs/` are protected by this path. Do not execute target-controlled files on the host. See exports(5), Subdirectory Exports.

Export authorization, listener addresses, and firewall rules are separate checks. NFS/RPC can listen on interfaces other than the lab link. Inspect `sudo ss -lntup`, `rpcinfo -p localhost`, and the existing firewall policy before and after startup. NFSv3 uses RPC services and may require additional ports; allowing only 2049 is not a complete recipe. If the isolated topology or allowed listener path is uncertain, leave the server stopped. Check `systemctl status nfs-kernel-server --no-pager` and `sudo journalctl -u nfs-kernel-server -n 30 --no-pager` before blaming a target mount failure.

To undo this example export, remove only its line from `/etc/exports` with `sudoedit`, then run `sudo exportfs -ar`. Stop the lab-only server and inspect activity, startup enablement, and remaining RPC listeners; stopping NFS does not promise every RPC socket is gone. Restore recorded states for pre-existing services. In the disposable VM, disable newly

installed lab-only services, including `rpcbind.service`/`rpcbind.socket` if no other RPC work needs them. Do not erase a pre-existing export table or stop services other users require.

3.8 USB-OTG flashing tools

Our first program cannot depend on a bootloader we have not built yet. Fortunately, there is already code in the chip: the Boot ROM. In USB boot mode it exposes **SDP**, the Serial Download Protocol, through USB-OTG and waits for a host tool to send an image. We use NXP's `uuu` to reach that starting point, even when storage has no working image.

`uuu` (Universal Update Utility)

Build NXP's uuu_1.5.201 release inside the workspace. Selecting a release keeps these instructions tied to known sources. The package-install command needs `sudo`; the source checkout and build run as your normal user:

```
$ cd ~/imx6ull/src
$ git clone --branch uuu_1.5.201 --depth 1 https://github.com/nxp-imx/mfgtools
$ sudo apt install libusb-1.0-0-dev zlib1g-dev libbz2-dev pkg-config cmake libzstd-dev
libtinyxml2-dev
$ cmake -S ~/imx6ull/src/mfgtools -B ~/imx6ull/build/mfgtools
$ cmake --build ~/imx6ull/build/mfgtools -j "$(nproc)"
$ . ~/imx6ull/scripts/env.sh
$ command -v uuu
/home/<you>/imx6ull/build/mfgtools/uuu/uuu
$ uuu -h
... help text and the selected release version ...
```

`cmake -S ... -B ...` reads sources from one directory and generates the build in another. The next command performs that build; `nproc` supplies the available CPU count for parallel jobs. The result stays under `build/mfgtools`, with no installation into `/usr/local/bin`. The dependency list comes from this release's CMake requirements, including zlib and TinyXML2 development files.

The final path is worth noticing: the build directory contains a subdirectory named `uuu`, and the executable inside it is also named `uuu`. For restricted USB access, we leave groups and udev rules unchanged and use `sudo`. Give it the full local path because its command search path may not include our workspace:

```
$ sudo "$IMX6ULL_HOME/build/mfgtools/uuu/uuu" -lsusb
```

Run this device check only after Chapter 8's power and USB-mode checks. `15a2:0080` is the ROM SDP identity. Enumeration proves device visibility, not successful image transfer or code execution. Read any flashing command before running it with root credentials.

3.9 SD card preparation for later chapters

The SD-boot examples assume a spare 4-32 GB SD card, class 10 or better, with no data to preserve. Raw image writes overwrite its contents. This section explains device identification only, not an image-writing operation.

Identify which device it is, **carefully**:

```
$ lsblk -o NAME,PATH,SIZE,MODEL,SERIAL,TRAN,RM,TYPE,MOUNTPOINTS
```

Read your actual output: `PATH` is the device path, `TYPE` distinguishes a whole `disk` from a `part` partition, and `MOUNTPOINTS` shows what is mounted. The later examples use `/dev/sdc` with a `/dev/sdc1` partition, but your card may have different names.

Here, a command finishing successfully can be the wrong result. `dd` can write perfectly good image bytes to a perfectly valid device path that happens to be your system disk. It cannot decide which disk you meant. Compare identity, size, and mount points before every write, not just during this first setup.

Compare the device list before and after insertion. Match size, model/serial, and the newly appearing reader/card; `RM=1` alone is not proof. Reject a device containing `/`, `/boot`, swap, or unrelated mounted data. A host disk can be `/dev/sdb`; a card can be `/dev/sda` or `/dev/mmcblkN`. Disk letters do not establish safety.

Do not write or repartition the card here. Chapter 11 explains the image and its placement contract. A whole-disk path and a partition path are not interchangeable, and a raw `.imx` payload is not a partitioned Linux card image. The later case study distinguishes its selected format from the verification of written bytes; a partition list alone cannot verify raw boot code.

Before any later write or physical removal, close programs using the card and unmount **every mounted card partition**. If an unmount fails or a mount point remains, stop; do not use forced/lazy unmount to get past the warning. During an actual write, stop on any write error and require successful completion before flushing or removing the card. `sync` succeeding after a failed write does not make that write complete. We keep identification and writing manual, without a disk-selection script.

3.10 Host IP plan

For TFTP, NFS, and U-Boot experiments, the board needs an address for the host. That address must stay stable so a working command does not suddenly point to the wrong machine.

Choose the arrangement that matches your desk. The direct link is the isolated lab setup used in our examples; a router-connected setup needs its own consistent addresses.

Option A: direct host-to-board link

Use this when your computer has a spare Ethernet port, a USB-to-Ethernet adapter, or Wi-Fi for internet plus Ethernet for the board.

For the direct link, our example addresses are:

- Host: **192.168.7.1**
- Board: **192.168.7.2**

Use this private `192.168.7.0/24` subnet only if it does not overlap an existing LAN or VPN route. It avoids router DHCP changes when assigned to a dedicated adapter.

`/24` means the first 24 address bits identify the network. Inspect `ip -4 route` before choosing it: an existing LAN or broader VPN route may already cover these addresses. In a VM, inspect the Ubuntu guest's routes and adapters, not Windows's. Correlate `ip -br link` with the adapter you physically assigned to the lab; an interface name alone does not identify its cable.

If you use NetworkManager:

```
$ sudo nmtui
```

In the text UI:

1. Choose **Edit a connection**.
2. Create a separate Ethernet connection named `imx-link` rather than overwriting your everyday connection. In its **Device** field, select the identified lab adapter, not your internet adapter.
3. Set **IPv4 CONFIGURATION** to **Manual**.
4. Add address `192.168.7.1/24`.
5. Leave gateway and DNS empty for this direct board link.
6. Save, return to **Activate a connection**, and activate `imx-link`. This creates a persistent profile. After the lab, deactivate it and reactivate the recorded previous profile through the same menu; remove only `imx-link` if it is no longer needed.

`enp0s31f6` below is an example host interface name, not the board interface. Substitute your adapter's actual name.

Verify:

```
$ ip -4 addr show enp0s31f6
... inet 192.168.7.1/24 ...
```

```
$ ip -4 route get 192.168.7.2
... dev enp0s31f6 src 192.168.7.1 ...
```

Require the expected lab device and source address in your actual route result before starting a bound server. Leaving the IPv4 gateway blank avoids adding an IPv4 default route for this direct link; it does not remove an existing VPN route or prove service isolation.

Option B: board and host on your existing router

Use this when your computer has only one Ethernet port and it already connects to your Wi-Fi modem or home router. In that case, do **not** force the host to `192.168.7.1`. Leave the host on the router's LAN, usually something like `192.168.1.x`, and plug the i.MX6ULL board into the same router or switch.

For the network-server examples, use an isolated development router/network rather than opening the trusting NFS recipe on the household or office LAN. Being on the same subnet establishes neither trust nor a safe listener/firewall boundary. If the required isolation is unavailable, postpone those services and use the non-network boot route.

Example:

- Router: **192.168.1.1**
- Host: **192.168.1.23**
- Board: **192.168.1.50**

Find the host's current LAN address:

```
$ ip -4 addr
```

Look for the address on the interface connected to the router. In later U-Boot commands, this host address becomes `serverip`.

Reserve the host address as well as the board address in the router, or use documented unused static addresses outside its DHCP pool. Update `HOST_IP`/`BOARD_IP` in `env.sh`, then source it in a fresh shell. Update the TFTP bind address, NFS client address, and any lab firewall rule's interface/source/destination consistently; remove only the superseded lab rule. For the board address:

- Reserve a fixed DHCP address for the board in your router.
- Let U-Boot request DHCP, then read the assigned address.
- Choose an unused static address outside the router's DHCP pool.

Router mode is practical, but it has two drawbacks:

- DHCP can change the board address unless you reserve it.

- Some routers isolate clients, especially guest Wi-Fi networks. If TFTP or ping fails even though both devices have `192.168.1.x` addresses, check client isolation and firewall settings.

Throughout the book, commands may show the direct-link values:

```
serverip=192.168.7.1
ipaddr=192.168.7.2
```

If you use router mode, substitute your real LAN values instead:

```
serverip=<your host IP, for example 192.168.1.23>
ipaddr=<your board IP, for example 192.168.1.50>
```

Chapter 8 checks physical Ethernet presence only. Target ping and TFTP tests wait until U-Boot runs; NFS-root acceptance waits until Linux and a complete rootfs exist.

3.11 Sanity check

Return to a fresh terminal and follow the selection from beginning to end: source `env.sh`, find both compilers, and find `uuu`. The listings below show the shape of the output. Your own paths and version strings are the evidence to keep; the placeholders are not output you need to reproduce.

```
$ . ~/imx6ull/scripts/env.sh

$ command -v arm-none-linux-gnueabihf-gcc
/home/<you>/imx6ull/toolchains/arm-gnu-toolchain-13.2.Rel1-x86_64-arm-none-linux-
gnueabihf/bin/arm-none-linux-gnueabihf-gcc

$ arm-none-linux-gnueabihf-gcc --version | head -1
arm-none-linux-gnueabihf-gcc (Arm GNU Toolchain 13.2.rel1 (Build arm-13.7)) 13.2.1
20231009

$ command -v arm-none-eabi-gcc
/home/<you>/imx6ull/toolchains/arm-gnu-toolchain-13.2.Rel1-x86_64-arm-none-eabi/bin/arm-
none-eabi-gcc

$ arm-none-eabi-gcc --version | head -1
arm-none-eabi-gcc (Arm GNU Toolchain 13.2.rel1 (Build arm-13.7)) 13.2.1 20231009

$ command -v dtc mkimage picocom uuu
/usr/bin/dtc
/usr/bin/mkimage
/usr/bin/picocom
/home/<you>/imx6ull/build/mfgtools/uuu/uuu

$ ls -d ~/imx6ull/{src,build,boot,rootfs,scripts,toolchains,notes}
/home/<you>/imx6ull/boot
/home/<you>/imx6ull/build
/home/<you>/imx6ull/notes
```

```
/home/<you>/imx6ull/rootfs
/home/<you>/imx6ull/scripts
/home/<you>/imx6ull/src
/home/<you>/imx6ull/toolchains
```

`| head -1` sends the version output to `head`, selecting its first line. Require both paths to name the selected `13.2.Rel1` directories and both banners to identify `13.2.rel1`. The case difference is Arm's naming, not a reason to rename anything. An unexpected alias/function or system path is a reason to inspect `type -a`, not continue with whichever compiler happens to run.

This completes the host-only check. During Chapter 8, identify the serial bridge and confirm ROM USB enumeration after the power checks. Card identity is recorded before SD boot, without writing it here.

For a network-boot configuration, each service has its own state: `systemctl is-active tftpd-hpa` and `systemctl is-active nfs-kernel-server` report them separately. A combined multi-unit query can return success when only one unit is active. The address/route, host-local TFTP retrieval, export, listener isolation, and startup state describe different host-side properties. None replaces target transfers or boots, and these services are not required by Chapter 9's USB route.

3.12 Key Notes

- Compiler installation and compiler selection are separate state. `. ~/imx6ull/scripts/env.sh` changes the current Bash shell and its future children, not another already-open terminal. Running the file in a child shell cannot change its parent.
- The two Arm 13.2.Rel1 archives remain project-local with Arm's extracted names. `CROSS_COMPILE` selects the Linux-target tools, while `BAREMETAL_CROSS_COMPILE` selects the no-OS tools. A prefix has meaning only when command lookup reaches the intended executable.
- Sources, generated outputs, staged boot files, and the target root filesystem have different owners and lifetimes. Creating `rootfs/` does not export it, and exporting it does not supply the kernel or a usable PID 1.
- Package installation, network profiles, and services persist beyond a terminal. TFTP transfers boot files into target RAM, whereas NFS supplies ongoing access to a host tree. Listener binding, firewall policy, and export authorization are separate controls.
- The illustrated `no_root_squash` NFS export trusts target root as host UID 0. An IP address is not authentication, and a subdirectory is not a strong sandbox. This configuration belongs on an isolated disposable system without personal files or shared host folders.

- Normal builds need no root credentials. Explicit `sudo` is confined to restricted devices or host configuration. A serial port opening, ROM USB enumeration, and a host-local transfer each establish only their own interface boundary.
- Storage paths identify current kernel objects, not a permanent physical card. Whole disks and partitions are distinct targets. Raw writes require fresh identity and unmount checks, and a successful `sync` cannot repair a failed write.

3.13 Pitfalls

- **`tftp` blocked by firewall.** A blocking policy/rule can drop the request or later transfer traffic. Inspect `sudo ufw status` and the actual policy; enabled UFW is not by itself proof that this traffic is blocked.
- **NFS-root failure.** Prefer the wired lab link. For “VFS: Unable to mount root fs”, inspect the full preceding error, `root=`/`nfsroot=`/`ip=` arguments, built-in NFS/network support, host exports/firewall, and rootfs contents. That message alone does not identify a timeout.
- **Forgot to source `env.sh`.** If `arm-none-linux-gnueabihf-gcc` or `arm-none-eabi-gcc` is not found, run `. ~/imx6ull/scripts/env.sh` in that terminal.
- **Wrong compiler on `PATH`.** `command -v arm-none-linux-gnueabihf-gcc` and `command -v arm-none-eabi-gcc` must both point into the selected workspace release. If either selects a system path, alias, or function, fix the environment before building.
- **Wrong storage destination.** Stop if identity is uncertain. No disk-letter filter can distinguish a spare card from a host disk; repeat identification and unmount checks at each write.
- **Building with `sudo`.** Build as the normal user. Plain `sudo make` may lose inherited compiler selection and creates root-owned output. Explicit assignments such as `sudo CROSS_COMPILE=...` can work subject to sudo policy, but are not a reason to build as root or disable `env_reset`. Use `sudo` only for the specific restricted operation.

3.14 Going deeper

- `man 8 exportfs`, `man 5 exports`, and `man 8 tftpd` explain the optional services configured in this chapter.
- In picocom, `-l` disables locking, `-i` skips initialization, and `-t` sends an initialization string. They are not needed for the normal console command; do not add options without checking the manual.
- *The TCP/IP Guide* (Charles Kozierok) on TFTP and NFS protocols if you want to know what is on the wire.

- For frequent cross-builds, `ccache` can reuse previous compilation results. Leave it out of the first experiments so the compile commands and rebuild decisions remain easy to follow. Add it later only after checking how the project’s build system selects its compiler.

We can now trace a compiler name to a real file and explain why we source the environment script at the start of a session. The next uncertainty is on the other end of the build. The Cortex-A7 accepts familiar-looking instructions, but it does not enter an interrupt like a Cortex-M. That difference is where Chapter 4 begins.

Chapter 4: ARMv7-A and the Cortex-A7, for the MCU engineer

An interrupt handler can return to the right instruction and still break the program it interrupted. If it overwrote a register without preserving the old value, returning to the right address does not recover that lost value.

On Cortex-M, hardware saves a useful part of that context before your handler begins. Our Cortex-A7 does not make the same stack push. It selects some different registers on exception entry and leaves software to preserve the rest. Familiar-looking C and assembly therefore sit on top of a different arrangement. We need to know what the core did before our first handler instruction, and what it expects us to do afterward.

Follow the integer registers, SVC and IRQ modes, banked SP/LR, and physical versus virtual addresses on a first reading. They explain the early startup and exception work. Caches, timers, security, and virtualization have their own later uses; their tables can remain reference material until the corresponding design appears.

4.1 Where the Cortex-A7 sits in the ARM lineup

Cortex-A7, Cortex-M7: the matching number invites an assumption that the cores are close relatives with similar software requirements. The letter matters more here. Arm names describe two different things:

- **Profile letter.** `A` for “Application” (smartphones, set-top boxes, embedded Linux), `R` for “Real-time” (storage controllers, automotive), `M` for “Microcontroller” (the Cortex-M0/M3/M4/M7/M33 you have worked with).
- **Architecture version.** v6, v7, v8, v9. The version determines the instruction set and which features are present. The core implementation determines pipeline depth, cache topology, and clock ceiling.

Our i.MX6ULL has one Cortex-A7, using the ARMv7-A architecture. It includes the VFPv4 floating-point unit and NEON vector instructions, but no Cortex-M companion. Some other i.MX6 parts, such as SoloX, include an M4; do not infer the contents of this chip from that family name.

The core name also does not specify the board’s clock frequency. Clock limits come from the fitted silicon’s speed grade and electrical conditions. The supplied industrial datasheet lists 528 and 792 MHz grades, while Linux v6.6’s ULL device tree lists operating points to be constrained by the part and board. A later example selects 696 MHz, but that is not a universal reset, BSP, or Linux frequency. Chapter 5 shows how to read the part marking before choosing a clock setting.

The Cortex-A7 is an in-order, dual-issue core with an 8-stage pipeline. We do not need to study the pipeline to write the first program. Its register and exception model, however, gives us a manageable place to learn the ARMv7-A mechanisms later used by U-Boot and Linux.

4.2 Comparing the Cortex-M and Cortex-A7 models

Use the following table to place familiar MCU features beside the Cortex-A7 equivalents. It is a comparison, not a specification for every Cortex-M part:

Feature	Cortex-M (typical)	Cortex-A7
Address space	Single, flat, physical	Virtual, per-context, via MMU
MMU	No (some have MPU)	Short-descriptor tables for our examples; LPAE also supported
Privilege levels	Privileged / Unprivileged	PL0/PL1, plus PL2 for HYP; modes below
Banked registers	A few (MSP, PSP)	Selected registers are banked in exception modes
Caches	L1 I/D on M7+, sometimes none	L1 I/D mandatory. L2 is integrated inside the Cortex-A7 MPCore platform (128 KB on i.MX6ULL), with no separate PL310 controller
TLB (Translation Lookaside Buffer, the MMU’s address-translation cache)	No	Yes
Generic timer	No (SysTick)	Yes, architected
Interrupt controller	NVIC (in-core, vectored)	**GIC**, an external Generic Interrupt Controller block. Prioritized, not auto-vectored
Exception model	Tail-chained, automatic stacking	Mode switch, software-saved context

Feature	Cortex-M (typical)	Cortex-A7
FPU	Optional VFP variant	VFPv4 (mandatory in i.MX6ULL)
SIMD	DSP extensions (limited)	NEON (64-/128-bit)
Atomic ops	Exclusives on M3/M4/M7; not M0/M0+	LDREX/STREX
Instruction set	Thumb; M0/M0+ have a smaller subset than M3/M4/M7	ARM and Thumb-2 for this book

Start with the exception-model row. It explains how the opening handler could return correctly yet leave the program damaged. Then look at the MMU row, which explains Chapter 2's separate process mappings. These are changes to how software uses the machine, not just extra peripherals in a longer feature list. Timer and NEON support are useful later, but their presence does not mean every Linux configuration uses them.

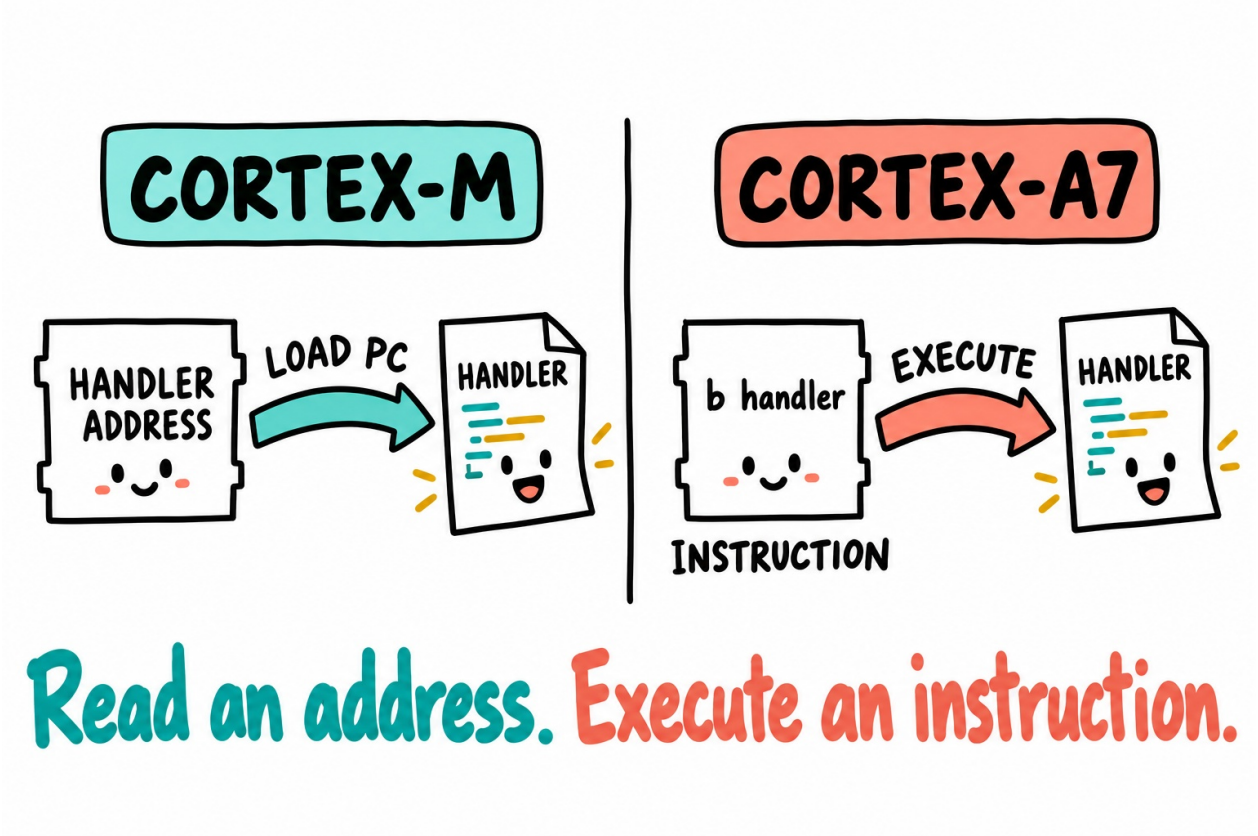
4.3 Exception modes and banked registers

Think of the first line in a Cortex-M C interrupt handler. It looks like an ordinary function, but the processor has already done some of the work that function relies on:

On Cortex-M, the usual sequence is:

1. Hardware automatically saves R0-R3, R12, LR, PC, and xPSR to the current stack.
2. Hardware loads PC from the vector table.
3. Your ISR runs.
4. `BX LR` (with the special EXC_RETURN value in LR) tells the CPU to unstack and resume.

There is also a difference in the vector itself. A Cortex-M exception vector contains a handler address. In the ARM-state vector table we will use on Cortex-A7, each slot contains an instruction, commonly a branch or a load into `pc`. The CPU starts executing that slot; it does not treat it as a Cortex-M handler pointer. See the Arm Cortex-A Programmer's Guide, Section 11.1.3.



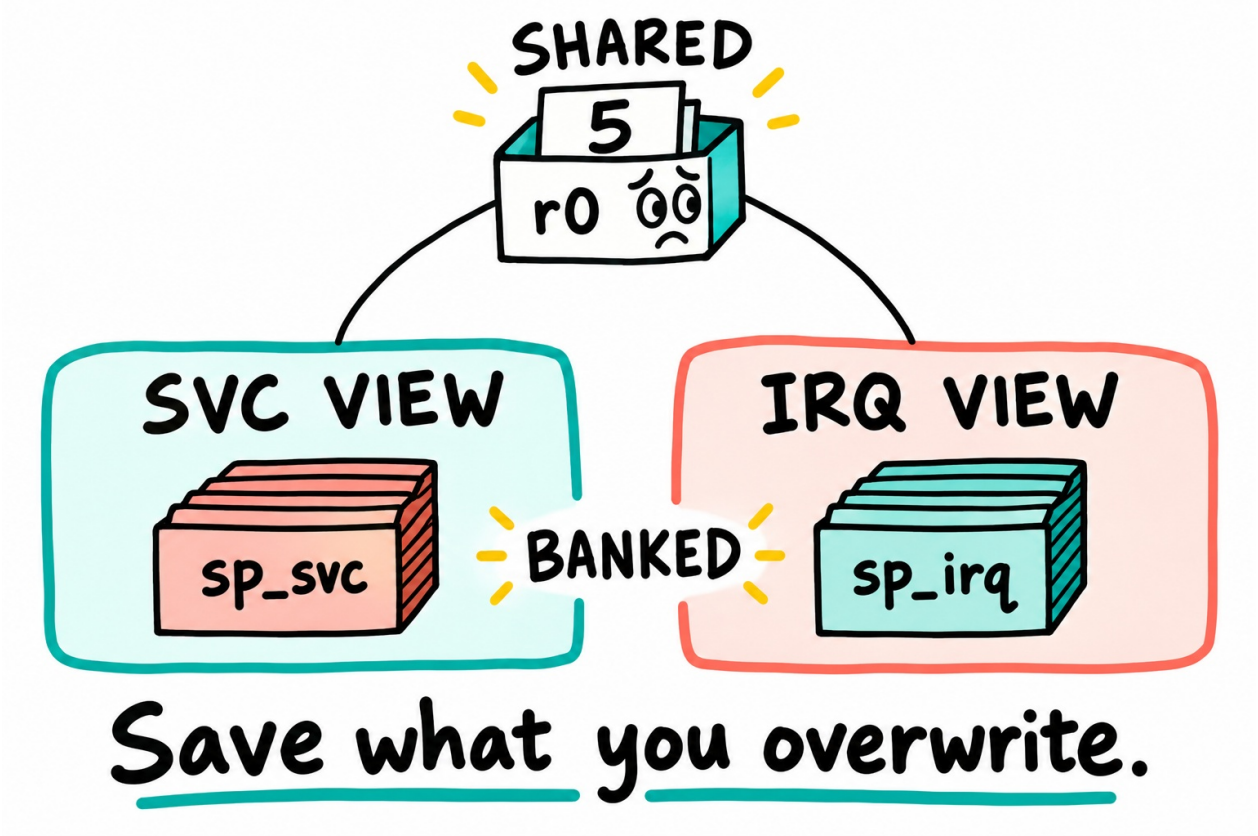
An address tells the core where to go. A branch is an instruction it executes to get there. This sketch compares handler-vector entries. The Cortex-M table's first word is a separate case: it supplies the initial stack pointer. `b handler` is an example, not a complete vector table or interrupt handler.

Our Cortex-A7 does not do that automatic stack push. Instead, an exception changes the **processor mode** and selects the register view for that mode. Some register names now refer to different physical registers. These separate copies are called **banked registers**.

Watch `sp` when an IRQ interrupts SVC code. Before the interrupt it refers to `sp_svc`; afterward the same name refers to `sp_irq`. The old stack pointer was not pushed anywhere. Its physical register is still there, out of the handler's current view. The IRQ stack must already have been initialized. Now watch `r0`: it has no corresponding IRQ bank. If the handler uses it, the interrupted value must be saved and restored by software.

The tables below describe this Cortex-A7's nine modes. Other A-profile cores need not implement exactly the same set. First use the tables to follow SVC and IRQ; the extra security and hypervisor modes can wait.

Before reading every row, hold the value 5 in `r0` and imagine an IRQ interrupting SVC code. The CPU saves status in `SPSR_irq`, records an exception link in `lr_irq`, selects `sp_irq`, and enters the IRQ vector. It does **not** create an IRQ copy of `r0`. A handler that overwrites the 5 must save it and restore it before returning.



Two SP registers, but one shared `r0`. The paper piles represent the stacks selected by those pointers. **BANKED refers to the SP registers, not the stack memory**. The connecting lines show that sharing, not a copy operation. IRQ entry pushes nothing automatically.

There is one LR/SPSR bank per exception mode, not a fresh frame for each interrupt. Preserve the return state before a C call overwrites LR or nesting can re-enter that mode. Ordinary IRQ entry masks IRQ; enabling nesting requires deliberate save/restore work. Reset is not a returnable saved pre-reset context. Chapter 15 implements the full entry and return path.

The registers you are looking at

Ordinary integer instructions use `r0` through `r15`. Several have familiar aliases: `sp` for the stack pointer, `lr` for the link register, and `pc` for the program counter. `CPSR` records the current status and mode; an exception mode also has an `SPSR` for saved status. The table connects these names to their roles in C calls and exception handling:

Register	Common alias	What it does	Beginner note
`r0-r3`	argument / result registers	Initial core argument/ result registers under AAPCS32; four ordinary 32-bit integer/pointer arguments fit here. Wider or aligned values follow the ABI's allocation rules.	Exception handlers must preserve interrupted values they overwrite, including through a C call.
`r4-r8`	general saved registers	General-purpose registers that C functions normally preserve for their caller.	`r8` is ordinary in most modes, but FIQ mode has its own banked `r8_fiq`.
`r9`	platform register	ABI-defined register. Some systems use it for a static base, thread pointer, or other platform role.	Treat it as "do not assume" in hand-written assembly that calls C.
`r10`	`sl`	General saved register. Some older ABIs used it as a stack limit.	Usually another callee-saved register in Linux code.
`r11`	`fp`	Frame pointer when frame pointers are enabled.	Useful when reading backtraces or compiler-generated assembly.
`r12`	`ip`	Scratch register used during function calls. Linker-generated branch stubs may use it.	Caller-saved. Do not expect it to survive a function call. FIQ also has `r12_fiq`.
`r13`	`sp`	Stack pointer. Points at the current mode's stack.	Banked in exception modes, so `sp_irq` and `sp_svc` are different physical registers.
`r14`	`lr`	Link register. Holds a subroutine return address, or an exception return address after an exception.	Banked in most exception modes, so `lr_irq` is not the same as user `lr`.

Register	Common alias	What it does	Beginner note
`r15`	`pc`	Program counter, the address of the instruction stream.	Writing to `pc` causes a branch. Exception return often restores a value into `pc`.
`CPSR`	current status	Holds condition flags, interrupt masks, Thumb/ARM state, and current processor mode.	The CPU changes CPSR when it enters an exception.
`SPSR_<mode>`	saved status	Snapshot of CPSR taken when an exception entered that mode.	Used on exception return to restore the old mode, flags, and interrupt-mask state. User/System mode do not have an SPSR.
`ELR_hyp`	hypervisor exception link	Return address used by HYP mode.	HYP is special: it uses `ELR_hyp` instead of a banked `lr_hyp` for exception return.

Knowing what a register does is one question. Knowing which physical copy is visible is another. The next two tables separate those questions by putting modes in columns and register names in rows.

First, the Cortex-M view:

Register	Thread mode	Handler mode
`r0-r12`	Same physical registers.	Same physical registers, exception entry stacks `r0-r3` and `r12` automatically.
`r13` / `sp`	`MSP` or `PSP`, selected by `CONTROL.SPSEL`.	`MSP`.
`r14` / `lr`	Normal function return address.	`EXC_RETURN`, the special value used by `BX LR` to leave the exception.

Register	Thread mode	Handler mode
r15 / pc	Current instruction stream.	Handler instruction stream loaded from the vector table.
status	Active xPSR.	Active xPSR. The previous xPSR is stacked automatically.

Here the general-purpose registers are shared. Hardware saves the listed interrupted values on entry and restores them on exception return. Thread mode may use MSP or PSP, while Handler mode uses MSP.

Now the Cortex-A7 view:

Register	USR	SYS	FIQ	IRQ	SVC	ABT	UND	MON	HYP
r0-r7	shared	shared	shared	shared	shared	shared	shared	shared	shared
r8-r12	shared	shared	r8_fiq-r12_fiq	shared	shared	shared	shared	shared	shared
r13 / sp	sp_usr	sp_usr	sp_fiq	sp_irq	sp_svc	sp_abt	sp_und	sp_mon	sp_hyp
r14 / lr	lr_usr	lr_usr	lr_fiq	lr_irq	lr_svc	lr_abt	lr_und	lr_mon	lr_usr*
r15 / pc	shared	shared	shared	shared	shared	shared	shared	shared	shared
CPSR	shared	shared	shared	shared	shared	shared	shared	shared	shared
SPSR	none	none	SPSR_fiq	SPSR_irq	SPSR_svc	SPSR_abt	SPSR_und	SPSR_mon	SPSR_hyp
HYP return address	none	none	none	none	none	none	none	none	ELR_hyp

`shared` means the same physical register is visible in that mode. For `r8-r12`, FIQ uses its own banked copies while the other modes share the ordinary registers. A suffix such as `_irq` means that mode has its own banked physical register. HYP mode shares `lr_usr` and uses `ELR_hyp` for exception return, as marked by `*` in the table.

The nine modes

Mode	Abbreviation	Entered on	Banked regs	`M[4:0]`
User	USR	(normal program execution)	none	`10000`
FIQ	FIQ	Fast interrupt	R8-R12, R13, R14, SPSR	`10001`
IRQ	IRQ	Normal interrupt	R13, R14, SPSR	`10010`
Supervisor	SVC	Reset, `svc` instruction	R13, R14, SPSR	`10011`
Monitor	MON	Secure Monitor Call (`smc`)	R13, R14, SPSR	`10110`
Abort	ABT	Memory/prefetch abort	R13, R14, SPSR	`10111`
Hyp	HYP	Hypervisor (virtualization)	R13, ELR_hyp, SPSR	`11010`
Undefined	UND	Undefined instruction	R13, R14, SPSR	`11011`
System	SYS	Privileged mode using the user register view	none	`11111`

The mode bits live in `CPSR.M[4:0]`. A returnable exception saves status in the destination mode's `SPSR`. Reset does not supply a restorable pre-reset frame. For C/assembly interfaces, consult AAPCS32's parameter-passing rules, not a one-register-per-C-argument assumption.

MON serves security-state transitions, and HYP serves virtualization. This Cortex-A7 supports those extensions, but our first bring-up path does not use them. Keep USR, SVC, IRQ, ABT, and UND in view first. Part IX returns to HYP.

Total physical register count exposed by Cortex-A7: **34 general-purpose**, **8 status (CPSR + 7×SPSR)**, plus `ELR_hyp`. That is 43 registers. A normal non-HYP mode sees at most 18 at once: `r0`-`r15`, `CPSR`, and its `SPSR`. HYP sees `ELR_hyp` as its extra exception-return address.

The opening puzzle now has a precise cause: banking preserves selected exception state, not every value the handler might overwrite. HYP has its own return arrangement using `ELR_hyp`; it is not the PL1 IRQ return described below.

What this means in practice

Put the two worlds side by side:

Moment	Cortex-M	Cortex-A / ARMv7-A
Interrupt accepted	Hardware picks the vector and starts exception entry.	Hardware switches to IRQ mode and branches through the exception vector.
Registers saved automatically	`r0`-`r3`, `r12`, `lr`, `pc`, xPSR are pushed to the current stack.	Nothing is pushed. `CPSR` is copied to `SPSR_irq`, and the return address goes into `LR_irq`.
Hardware entry-frame stack	The interrupted context’s active MSP or PSP.	No hardware stack frame is pushed.
SP used by handler instructions	MSP.	Initially the banked `sp_irq`, which must already point at a valid IRQ stack.
What `lr` means inside the handler	A special `EXC_RETURN` value that tells the CPU how to unstack.	The banked `LR_irq`, holding an exception return address that often needs an offset adjustment.
Handler prologue	Often no assembly is needed, a C ISR can start immediately.	Assembly must save enough state before calling C.

Moment	Cortex-M	Cortex-A / ARMv7-A
Return	`BX LR` with EXC_RETURN triggers hardware unstacking.	Software restores saved registers, then uses an appropriate exception-return instruction to restore CPSR and PC together.

For an ordinary PL1 IRQ with the original IRQ link restored, `subs pc, lr, #4` is one return form: it adjusts the IRQ link and restores status from `SPSR_irq` with the return PC. This is not a complete handler. SVC, abort, and HYP returns have different rules; an ordinary `BX LR` is not an A-profile EXC_RETURN operation. Do not restore CPSR first and then branch as two independent steps. Linux's `entry-armv.S` contains its entry/exit machinery.

PL0 vs PL1 vs PL2

Across the nine modes, ARM defines three **privilege levels**:

- **PL0** (unprivileged, like Cortex-M unprivileged Thread mode): only USR mode. This is the mode applications run in.
- **PL1** (privileged): most modes, including SVC, IRQ, FIQ, ABT, UND, and SYS. This is the level the kernel runs at.
- **PL2** (hypervisor): only HYP mode. It can intercept selected PL1 operations and transfer control to the hypervisor.

ARM TrustZone adds a separate **Security state**: Normal World or Secure World, with restricted combinations. On ARMv7-A, MON exists only in Secure state and HYP/PL2 only in Non-secure state; most other modes can exist in either. MON handles transitions between the worlds. Security state is not another privilege-level number.

MMU/control and cache-maintenance operations generally require privileged execution. Some CP15 registers, such as user thread-ID registers, are user-accessible; access rights are defined per register.

You will also meet the name **CP15** in startup assembly. It is the ARMv7-A access path for system-control registers, not another chip on the board. Instructions such as `mrc` and `mcr` access controls including the MMU enable bits in `SCTLR`, page-table bases in `TTBR0` and `TTBR1`, and the exception-vector base in `VBAR`. Cache maintenance and generic-timer access also use this path. AArch64 uses named system-register instructions instead.

Linux runs ordinary user instructions in USR/PL0 and uses PL1 modes for kernel execution. Much kernel code runs in SVC mode; exception entry can use IRQ or another mode before an entry veneer changes it. A syscall or interrupt can trigger the hardware transition. Do not equate every kernel instruction with the first mode selected by its exception vector.

HYP mode is the extra layer used by a hypervisor. A guest kernel runs as if it controls PL1, but sensitive actions can be redirected to PL2. The hypervisor can then decide whether to allow the action, emulate it, or stop the guest. The matching instruction is `hvc` (Hypervisor Call), similar to `svc` but for calls into the hypervisor.

Do not confuse HYP mode with Monitor mode:

- **HYP / PL2:** virtualizes Normal World operating systems.
- **MON / Monitor mode:** switches between Secure World and Normal World for TrustZone.

Part IX uses this distinction in its Xen design: HYP mode controls guests, while OP-TEE in Chapter 124 uses Monitor/Secure-world mechanisms.

Chapter 2's syscall now has a more concrete explanation. Its `svc` instruction triggers a Supervisor Call exception from USR to SVC mode. SP and LR select their SVC copies, and the kernel handler runs with privileged access. The application requested this transition deliberately, but the core still enters an exception. The mode table describes something we have already seen from the caller's side.

4.4 The CPSR / SPSR program status registers

CPSR (Current Program Status Register) is the A-profile equivalent of M-profile's xPSR, but more is in it:

```
Selected fields only:
31 30 29 28 27 ...  9  8  7  6  5  4  3  2  1  0
N  Z  C  V  Q       E  A  I  F  T  M4 M3 M2 M1 M0
```

- **N, Z, C, V:** condition flags (the same as M-profile: negative, zero, carry, overflow)
- **Q:** saturation flag, set by saturating arithmetic instructions
- **IT[7:0]** (split bits 26:25 + 15:10): IF-THEN block state for Thumb-2 conditional execution
- **J** (bit 24), **T** (bit 5): our examples use ARM (`J=0,T=0`) or Thumb (`J=0,T=1`). Other architectural encodings include legacy ThumbEE/Jazelle. Cortex-A7's Jazelle implementation is trivial, not Java-bytecode acceleration; those encodings are not additional instruction sets used by this book.
- **GE[3:0]** (bits 19:16): flags used by packed integer instructions such as `UADD8`, not NEON comparisons.
- **E:** current data-endianness state where applicable, not a choice attached to each load/store. This book uses little-endian data.
- **A:** asynchronous abort mask

- **I:** IRQ mask (`I=1` disables IRQs)
- **F:** FIQ mask
- **T:** Thumb state (`T=1` means executing in Thumb)
- **M[4:0]:** current processor mode (encoding for USR/SVC/IRQ/…)

For a returnable exception, the CPU saves CPSR in `SPSR_<mode>` and selects the exception mode. That bank can be overwritten by another entry to the same mode. The handler can read saved status with `mrs Rn, spsr`; privileged `cps` instructions can change the current mode and interrupt masks. None makes reset a saved frame to return to.

In the early startup examples, the mode field and I/F interrupt masks describe which register bank is active and whether IRQ or FIQ may interrupt the code. The remaining status fields become useful when reading instruction behavior and more advanced exception paths.

4.5 Memory model and the MMU

Recall the two processes in Chapter 2 that used the same pointer value for different physical RAM. The CPU needs a way to decide which translation applies. On our non-virtualized path, **stage 1** performs that virtual-to-physical translation using tables in memory. Here we can identify the structures that make the earlier example possible.

The Cortex-A7 also supports a second translation stage for virtualization. Do not confuse a translation *stage* with a table *level*: two stages are guest and hypervisor translations, while two levels are successive table lookups within one translation. The initial bare-metal design needs only stage 1.

Translation table formats

ARMv7-A supports two translation table formats:

- **Short descriptor.** Our ordinary mappings use this SoC’s 32-bit physical map. Small pages use two table levels; a section can finish at Level 1. The architecture also defines extended supersection cases outside this example’s scope.
- **Long descriptor (LPAE, 40-bit physical, 3-level tables).** Used when a system needs larger physical addresses or LPAE features. Our board has at most 512 MiB of DRAM, so this book does not need LPAE.

Here are two distinct short-descriptor paths, using a full, unsplit L1 table (`TTBCR.N=0`):

```
4 KiB small page:
  VA[31:20] -> L1 entry -> L2 table indexed by VA[19:12]
                         -> physical page base + VA[11:0]

1 MiB section:
```

```
VA[31:20] -> L1 section entry
                        -> physical section base + VA[19:0]
```

For illustrative VA `0x12345678`, the small-page indices are `0x123` and `0x45`, with offset `0x678`. A section uses the same L1 index but offset `0x45678`, with no L2 lookup. A full L1 table has 4096 entries/16 KiB; a coarse L2 table has 256 entries/1 KiB. Other table-split settings change the applicable indexing rules.

Each Level-1 entry can:

- Point to a Level-2 table (resolves a 1 MB region in 4 KB pages).
- Be a **1 MB "section"** directly (no Level-2 lookup needed, saving a memory access).
- Be a **16 MB "supersection"** (less common).

The first lookup need not send the MMU to a second table: a section entry can describe the region directly. That is why the descriptor's type matters before its other fields. Section and small-page mappings also differ. For now, ask what a completed mapping must tell the MMU: the physical address, who may access it, and how accesses to that memory should behave.

The corresponding leaf fields include access permissions (AP), execute-never (XN), and memory attributes (TEX/C/B and shareability). Relevant L1 descriptors also select a domain; a domain in "client" mode applies the mapping's permissions. Chapter 17 decodes the exact section format used by its example.

The **TLB** caches recent address translations. There is also an **ASID** (Address Space ID, 8 bits) that tags TLB entries so context switches do not need a full TLB flush.

Chapter 17 follows a minimal L1-only page table, separating descriptor layout, memory attributes, and the control-register sequence that activates translation.

What the kernel does with this

Linux on ARMv7-A:

- In the v6.6 non-LPAE two-level path, uses unsplit tables and switches the process table through **TTBR0**. Other configurations can select TTBR0/TTBR1 regions; a virtual-address user/kernel split does not imply that hardware-register split. See proc-v7-2level.S.
- Splits the 4 GB virtual address space into a user range (`0x00000000-0xBFFFFFFF`) and kernel range (`0xC0000000-0xFFFFFFFF`) by default. `CONFIG_PAGE_OFFSET` controls this split.
- Uses a fixed offset for the **linear mapping of low RAM** (`PHYS_OFFSET` to `PAGE_OFFSET`), not arbitrary `vmalloc`, `ioremap`, or high-memory pointers. Drivers use mapping and DMA APIs, not subtraction from any kernel pointer. See the ARM memory layout.

With that default split, the user range occupies about 3 GiB of virtual address space. This is a layout choice, not a statement that each process has 3 GiB of physical RAM.

4.6 Caches

The CPU writes a new value to a buffer. Has that value reached the memory a peripheral will read? With a write-back cached mapping, the modified bytes can still be in the CPU's cache. A cache makes repeated accesses faster by keeping copies nearby, but low-level code must also know when those copies agree with memory. First distinguish the i.MX6ULL's two L1 caches:

Cache	Size	Ways	Line size	Organization
L1 instruction	32 KiB	2	32 bytes	VIPT: virtually indexed, physically tagged
L1 data	32 KiB	4	64 bytes	PIPT: physically indexed, physically tagged

These properties are specified separately in the Cortex-A7 TRM DDI 0464F, Chapter 6. NXP RM Chapter 12 specifies the implemented sizes and integrated 128 KiB unified L2. This SoC does not have the separate PL310 controller used by some earlier i.MX6 parts.

Two A-profile cache properties are important here:

1. **Processor reset is not our image's entry state.** Caches are disabled at architectural reset, but the Boot ROM runs before `_start` and can enable L1 instruction caching while loading [RM 8.4.4]. Startup must account for the documented ROM handoff contract and normalize the state its design requires. Do not infer "all caches off" merely from having cold-started the board. Chapter 6's skeleton is build-only, not a complete handoff implementation.
2. **I/D geometry differs.** Do not use one line size for both maintenance loops. DMA sharing and changed executable instructions require operation-specific coherency work in Chapters 17 and 51.

The maintenance names describe different actions. **Clean** writes modified, or *dirty*, data toward the point required for the other observer to see it. **Invalidate** discards a cached copy so it will not be reused. Discarding dirty data without first cleaning it can lose writes. **Clean and invalidate** performs both actions; "flush" alone does not specify which.

The encodings below are **maintenance reference**, not a complete procedure to run. Chapter 17 supplies the required ranges, attributes, synchronization, and coherency-point definitions; Linux drivers use the appropriate DMA APIs.

ARMv7-A performs cache maintenance through **CP15** operations. The assembly uses `mcr` with an operation-specific CP15 encoding:

Operation	ARMv7-A form	Architectural name
Invalidate D-cache line by virtual address	`mcr p15, 0, Rt, c7, c6, 1`	`DCIMVAC`
Clean D-cache line by virtual address	`mcr p15, 0, Rt, c7, c10, 1`	`DCCMVAC`
Clean and invalidate D-cache line by virtual address	`mcr p15, 0, Rt, c7, c14, 1`	`DCCIMVAC`
Invalidate all instruction caches to PoU	`mcr p15, 0, Rt, c7, c5, 0`	`ICIALLU`

Set/way operations are also available for whole-cache maintenance during early boot.

Chapter 17 explains a small cache-maintenance primitive. Linux's `arch/arm/mm/cache-v7.S` provides the operating system's broader implementation.

4.7 The generic timer

The familiar SysTick combines a counter with an interrupt mechanism. The **ARMv7 generic timer** separates these ideas: a shared system counter supplies time, while per-CPU comparators decide when to raise an interrupt. Cortex-A7 supports this interface, but the SoC's clocks and power integration still determine when it operates.

Key registers (CP15 access):

- `CNTFRQ`: counter frequency (Hz). Software writes this once at boot to inform the rest of the system.
- `CNTPCT`: current counter value (64-bit physical counter).
- `CNTP_CVAL`: comparator value. The timer fires when CNTPCT >= CNTP_CVAL.
- `CNTP_CTL`: enable + interrupt mask + status.

Linux's `arch_timer` supports it, but v6.6 imx6ul.dtsi disables the generic timer, and imx6ull.dtsi does not enable it. The inherited setup uses SoC timer facilities such as GPT; do not enable a node merely because the core implements its architecture. Chapter 16 uses GPT/EPIT on bare metal. After Linux boots, inspect timer/clocksource messages and `/sys/devices/system/clocksource/clocksource0/current_clocksource` to identify the actual configuration.

Keep that separation in mind when reading the register names: SysTick is a per-core 24-bit down-counter, while the generic timer uses a shared 64-bit counter and per-CPU comparator state. Reading the time and arranging a future interrupt are related operations, but not the same one.

4.8 The Generic Interrupt Controller (GIC)

The Cortex-M NVIC was inside the core. The A-profile equivalent, the **GIC**, is outside the core. The i.MX6ULL integrates a **GIC-400** (an implementation of GIC v2).

We know how IRQ entry changes the CPU state. One question is still unanswered: which peripheral caused it? The GIC supplies that part of the interrupt path. It splits the work between a system-wide distributor and a CPU interface:

GICv2 has two parts:

- **Distributor** (`GICD_*` registers, base `0x00A01000` on i.MX6ULL): one per system. It arbitrates which interrupt goes to which CPU, sets priorities, masks, and trigger types (level/edge).
- **CPU Interface** (`GICC_*` registers, base `0x00A02000`): one per CPU. The core reads acknowledgement and writes end-of-interrupt here.

Three flavors of interrupt:

Type	ID range	Purpose
SGI, Software-Generated	0-15	Software-generated interrupts, including inter-processor interrupts in SMP systems
PPI, Private Peripheral	16-31	Per-CPU peripherals. The generic timer interrupt is a PPI.
SPI, Shared Peripheral	32-1019	Architectural range for shared peripheral interrupts; not all are implemented on this SoC

The i.MX6ULL maps SoC peripheral interrupts to SPI IDs. The mapping is in the reference manual's Chapter 3, "Interrupts and DMA Events". For example, `UART1` is SPI 26 (which the GIC sees as ID 26+32 = 58).

Unlike the NVIC's vector selection, the GIC does not send the CPU directly to a different handler address for every peripheral. An IRQ enters the CPU's IRQ exception path. Software reads `GICC_IAR` to obtain the interrupt ID, dispatches to the appropriate handler, and writes `GICC_EOIR` when handling is complete. This is the other half of the interrupt story: the CPU's banked registers handle exception state; the GIC tells software which source needs attention.

4.9 Atomics, barriers, and memory ordering

Writing a buffer and then announcing that it is ready looks straightforward in C. Low-level code must also ensure that the relevant observer sees the accesses in the required order. ARMv7-A is **weakly ordered**, so source order alone is not a sufficient guarantee. Three barrier instructions recur in later startup code:

- `dsb` (Data Synchronization Barrier): waits for outstanding memory accesses to complete.
- `dmb` (Data Memory Barrier): orders accesses but does not necessarily wait.
- `isb` (Instruction Synchronization Barrier): synchronizes subsequent instruction execution with relevant context changes. It is not a complete page-table update recipe, nor required after every condition-flag change. A particular operation may require DSB, cache maintenance, TLB invalidation, and ISB in a specified sequence.

LDREX/STREX solves a different problem: updating a word without losing an intervening update. The pair attempts an exclusive load and store; if the store fails, this example retries. It assumes a suitably aligned word in normal memory, not an arbitrary device register. Atomicity by itself does not order all other accesses.

```
retry:
    ldrex   r1, [r0]      @ load-exclusive
    add     r1, r1, #1
    strex   r2, r1, [r0]  @ store-exclusive, r2=0 on success
    cmp     r2, #0
    bne     retry
```

Look at `r2` in that loop. It reports whether `strex` succeeded; it is not the new counter value. The branch retries when the exclusive store failed. Cortex-M parts that support exclusives use the same instruction family. On A-profile, the required barriers are still needed when this operation must also order other memory accesses.

4.10 NEON and VFP

VFPv4 gives you 32 double-precision FP registers and the usual IEEE-754 operations. NEON shares the same register file (viewed as 16 × 128-bit Q registers, or 32 × 64-bit D registers) and adds packed integer/float SIMD.

Ordinary kernel code must not use floating point or accidentally generate NEON instructions. Specialized paths follow Linux's Kernel Mode NEON recipe: controlled compilation units, a permitted execution context, and `kernel_neon_begin()`/`kernel_neon_end()` protection. The protected work must not sleep and must not run in hard-IRQ context. A begin/end pair alone is not permission to put DSP code in an ISR; misuse can trap, fault the kernel, or corrupt FP state. Our driver examples use integer code.

Our startup examples use integer instructions. User programs can use NEON once the operating system enables and manages its context. Whether a particular library function does so depends on that library's build and implementation; its binary, not the name `memcpy`, determines whether it uses vector instructions.

4.11 Differences between Cortex-A7 and the bigger A-cores

The following table compares Cortex-A7 with two newer Cortex-A cores:

Feature	Cortex-A7	Cortex-A53	Cortex-A72
ISA	ARMv7-A (32-bit only)	ARMv8-A (32+64-bit)	ARMv8-A
Pipeline	In-order, 8 stages	In-order, 8 stages	Out-of-order, 15 stages
L1 D-cache	32 KB PIPT	32 KB PIPT	32 KB PIPT
Generic timer	Yes (CP15)	Yes (system reg)	Yes (system reg)
GIC version	GICv2	GICv2/v3	GICv2/v3

The most important difference for this book is the instruction set. Cortex-A7 is 32-bit only, so our assembly, page tables, and system registers use ARMv7-A definitions. The same high-level concepts apply to AArch64, but system register names and many bit layouts change.

4.12 Key Notes

- A Cortex-A7 exception selects banked SP/LR and saves status, but does not push a Cortex-M-style register frame. Shared registers, usable mode stacks, and return-state preservation remain software responsibilities.
- Exception return restores the interrupted status and the correct PC together. A normal function return neither reverses the mode transition nor recovers values a handler overwrote.
- The GIC identifies and arbitrates interrupt sources independently of the CPU's exception context. Software acknowledges an ID, dispatches it, and completes handling through the GIC interface rather than receiving a separate CPU vector for each peripheral.
- Translation tables describe permissions and memory behavior as well as addresses. A 1 MiB section ends at L1, whereas a 4 KiB small page follows L2. Translation stages and table levels solve different problems.
- Cache visibility, access ordering, and atomic updates are separate requirements. Clean/invalidate operations, DMB/DSB/ISB, and LDREX/STREX are not substitutes for one another, and L1 instruction/data maintenance uses different line geometry.
- Startup inherits the Boot ROM's handoff state rather than architectural reset state. The build-only Chapter 6 skeleton does not implement the complete cache, MMU, or exception contract needed for execution.
- CPU feature support does not establish operating-system policy. Generic-timer use depends on SoC integration, NEON requires managed context, and HYP virtualization is distinct from Monitor/Secure-world transitions.

4.13 Pitfalls

- **Assuming A-profile exceptions auto-stack like M-profile.** They do not. An IRQ handler that fails to save the required registers corrupts the interrupted context.
- **Forgetting `isb` after writing system registers.** Changes to TTBR, SCTLR, and VBAR require the barriers specified by the architecture. Without them, later instructions may execute using old state.
- **Thinking the GIC is in the core.** It is a separate memory-mapped block. You configure it via loads/stores to MMIO, not CP15. **MMIO** means memory-mapped I/O, where software accesses peripheral registers through normal load and store instructions.

- **Mixing up SPI (Shared Peripheral Interrupt) with SPI (Serial Peripheral Interface bus).** Context disambiguates, but every paragraph that mentions both is hard to read. This book will say “GIC SPI” or “SPI bus” whenever it could be ambiguous.
- **Treating one cache operation as a universal recipe.** Set/way maintenance visits physical sets/ways for whole-cache work; it is not simply a scheme that misses virtual aliases. Buffer-specific maintenance needs the correct address range, line boundaries, coherency point, and synchronization. Multi-core coordination adds requirements. Follow the operation’s sequence, and use Linux DMA APIs in drivers.

4.14 Going deeper

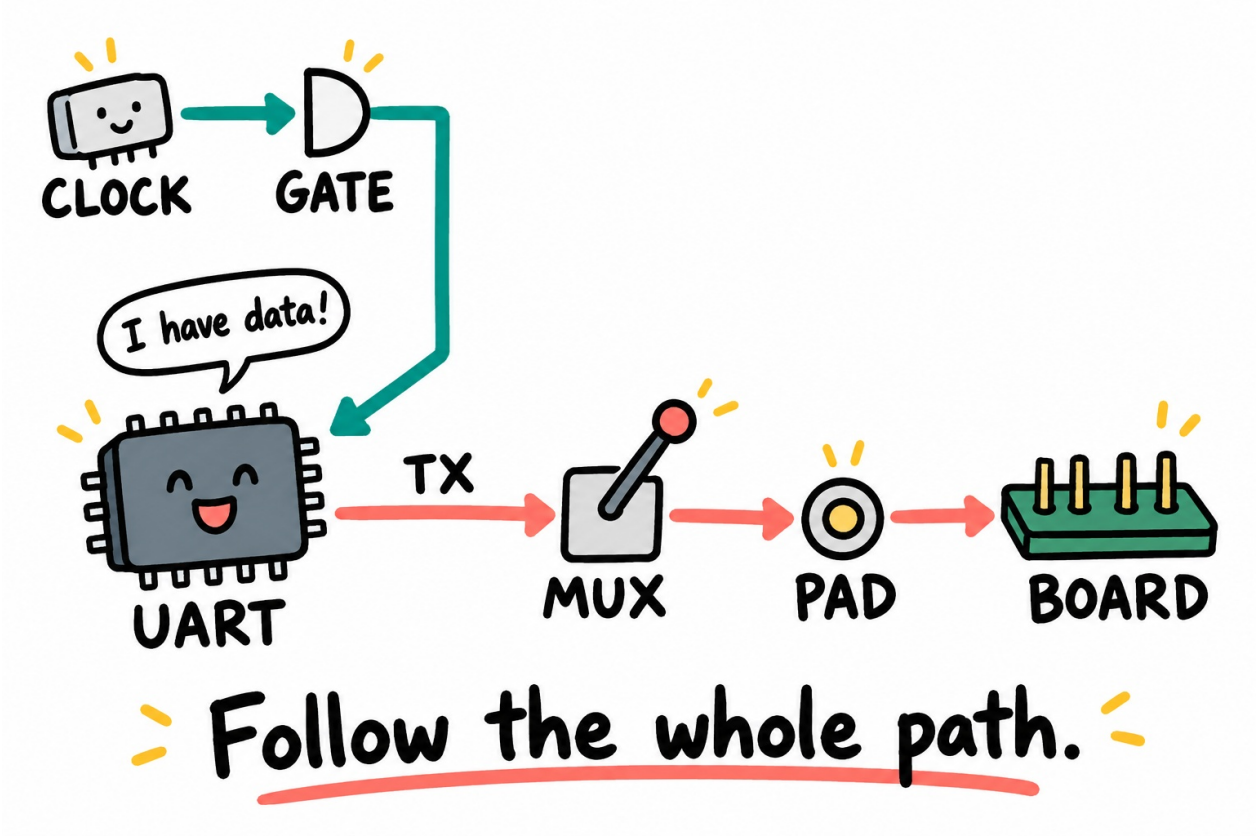
- ARM DDI 0406: *ARM Architecture Reference Manual, ARMv7-A/R*. The authoritative architecture reference.
- ARM DDI 0464: *Cortex-A7 MPCore Technical Reference Manual*. The implementation specifics.
- ARM IHI 0048B: *ARM Generic Interrupt Controller v2 Architecture Specification*.
- ARM DEN 0013: *Cortex-A Series Programmer’s Guide*. A tutorial-style overview.
- LWN: “An introduction to the ARM Generic Interrupt Controller” (2014).
- Linux source: `arch/arm/include/asm/{system,memory,page,pgtable}.h`, `arch/arm/mm/proc-v7.S`, `arch/arm/kernel/entry-armv.S`.

The opening interrupt no longer has to be mysterious: returning to the right instruction also requires restoring the state that instruction expects. Keep that habit of asking what must already be true. A GPIO write has its own prerequisites outside the CPU core: an address, a clock, and a route to the board’s pin. We follow those in Chapter 5.

Chapter 5: A tour of the i.MX6ULL SoC

A correct register write need not produce a change at a connector. A UART can be configured while its signal is routed to a different pad. A GPIO value can change while the LED's pad is assigned to another function. The instruction did its job; the path to the outside world was incomplete.

On a new MCU, you would trace that path through the reference manual and schematic. We will do the same on the i.MX6ULL: find the controller, find its clock, and follow its signal to the board. The chip is larger, but the questions are familiar. Keep an LED and a UART in mind as we move through the map; the worked UART1 lookup at the end will bring the pieces together.



"I have data!" is only the UART's part. Teal follows the clock into the UART. Coral follows TX out through the selected mux and pad to the board net. The chapter follows both paths. This sketch shows the idea, not the board's complete wiring.

5.1 What is the i.MX6ULL

The i.MX6ULL is a **system on chip**, or SoC: the CPU core is one part of a device that also includes memory controllers, clocks, and peripherals. It is a low-cost member of NXP's i.MX6 family, with one Cortex-A7 and no GPU, VPU, or PCIe controller.

Many of its peripheral names will be familiar: UART, I2C, SPI, timers, and GPIO. Others serve the larger system: the external DDR controller, Ethernet, USB, and storage interfaces. The chip also provides LCD, CSI camera, and SAI audio interfaces. These resources suit systems such as industrial control panels, meters, and gateways.

Keep the following summary nearby as we examine the map. Where a resource depends on the part variant or fitted components, check the actual board rather than relying on the MINI name alone:

- **Core:** one Cortex-A7; permitted frequency depends on the fitted part's speed grade and supplies.
- **L1 cache:** 32 KB I + 32 KB D
- **L2 cache:** 128 KB unified, integrated inside the Cortex-A7 MPCore (no external PL310 controller)
- **On-chip memory:** 128 KB **OCRAM** at `0x00900000`, plus a separate 96 KB **Boot ROM** at `0x00000000` that is mask-programmed by NXP. There is **no TCM**. A-profile systems normally use caches and system RAM instead.
- **DRAM:** 16-bit LPDDR2/DDR3L/DDR3 controller (MMDC). The Point Atom core boards used in this book have 256 MiB or 512 MiB.
- **Boot media:** SD/MMC, eMMC, NAND, SPI NOR, QSPI, parallel NOR, and USB SDP recovery
- **Process:** 28 nm
- **Package:** 289-ball VM (14 x 14 mm) or 272-ball VK (9 x 9 mm), according to the fitted variant/datasheet; not a universal interchangeable pinout.

Before we choose a clock setting, the chip marking deserves a closer look. Two boards sold under a similar name need not carry the same part. `Y` identifies i.MX6ULL, while `G` identifies i.MX6UL; those are different families, not price grades of the same part. Within ULL, `Y0`, `Y1`, and `Y2` identify baseline, reduced-feature, and full-feature variants.

For example, the supplied core schematic labels U1 `MCIMX6Y2CVM05AB`:

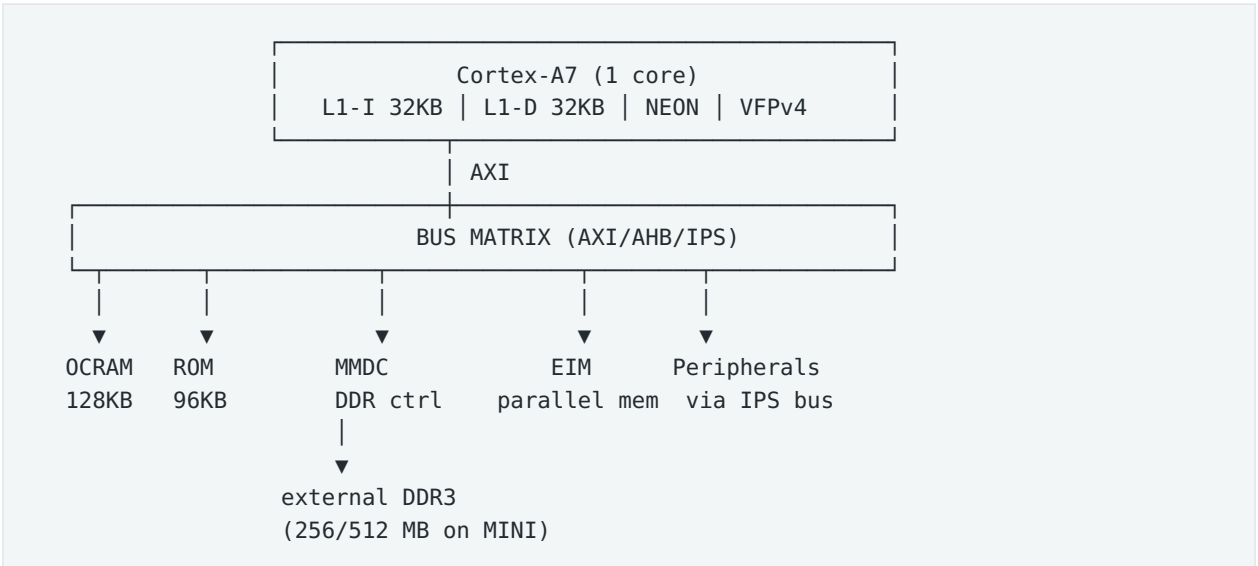
Field	Meaning in the supplied industrial datasheet
`MC`	Production qualification
`IMX6Y`	i.MX6ULL family
`2`	Full-feature differentiator

Field	Meaning in the supplied industrial datasheet
`C`	Industrial junction-temperature grade
`VM`	14 x 14 mm MAPBGA package
`05`	528 MHz speed grade
`A`	Fuse-option field
`B`	Silicon revision 1.1

The `05` in that long name changes a practical decision: this example is a 528 MHz-grade part. A clock value copied from another project is not made safe by the fact that both projects use a Cortex-A7. Check the marking actually fitted on your board, not just the schematic label. **Do not apply a 696 MHz example to a 528 MHz-grade part.** Even a higher-grade part needs the documented voltage and temperature conditions for its chosen frequency.

5.2 Block diagram

Here is a simplified view of how the major blocks connect. Some buses are omitted so we can follow the path from the CPU to memory and peripherals:



Follow a line from the CPU to OCRAM, then follow another toward the peripherals. The **bus matrix** carries those accesses from a master, such as the CPU or a DMA engine, to the selected target. A UART register write travels out of the core to the controller's block. The early examples do not configure the matrix, but it makes the next table more concrete: the physical address selects where that access goes.

5.3 System memory map

Where does a CPU load or store go? The physical address selects a region in the SoC's 4 GiB map. Some regions contain memory, others contain registers, and others are reserved. This selected-region table follows RM Tables 2-1 through 2-4; an address marked reserved is not spare space for our code.

Region	Base	Size	What's there
Boot ROM	`0x00000000`	96 KB	NXP's mask-programmed boot code (see Ch 7)
OCRAM	`0x00900000`	128 KB	On-chip SRAM
GIC distributor	`0x00A01000`	4 KB	(and CPU interface at `0x00A02000`)
External PL310 L2 controller	(absent)	-	i.MX6ULL's 128 KB L2 is integrated in the MPCore block.
AIPS-1	`0x02000000`	1 MB region	Includes GPIO, UART1, IOMUXC, CCM/analog clocks, GPT/EPIT, SNVS
AIPS-2	`0x02100000`	1 MB region	Includes USB, MMDC, USDHC and other peripheral blocks
AIPS-3	`0x02200000`	1 MB region	Includes DCP, RNGB, UART8, EPDC and related blocks

Region	Base	Size	What’s there
External Memory Interface (EIM)	`0x50000000`	128 MB	Parallel external memory; alias at `0x58000000`
QSPI	`0x60000000`	256 MB	Memory-mapped QuadSPI flash
MMDC0 (DRAM)	`0x80000000`	up to 2 GB	External DDR3

For the early boot designs, three parts of the map are especially useful. First, **DRAM begins at `0x80000000`**. A U-Boot setting such as `loadaddr=0x80800000` therefore places an image 8 MiB above that base, leaving working space around it. Remember that this is external memory and needs initialization before use.

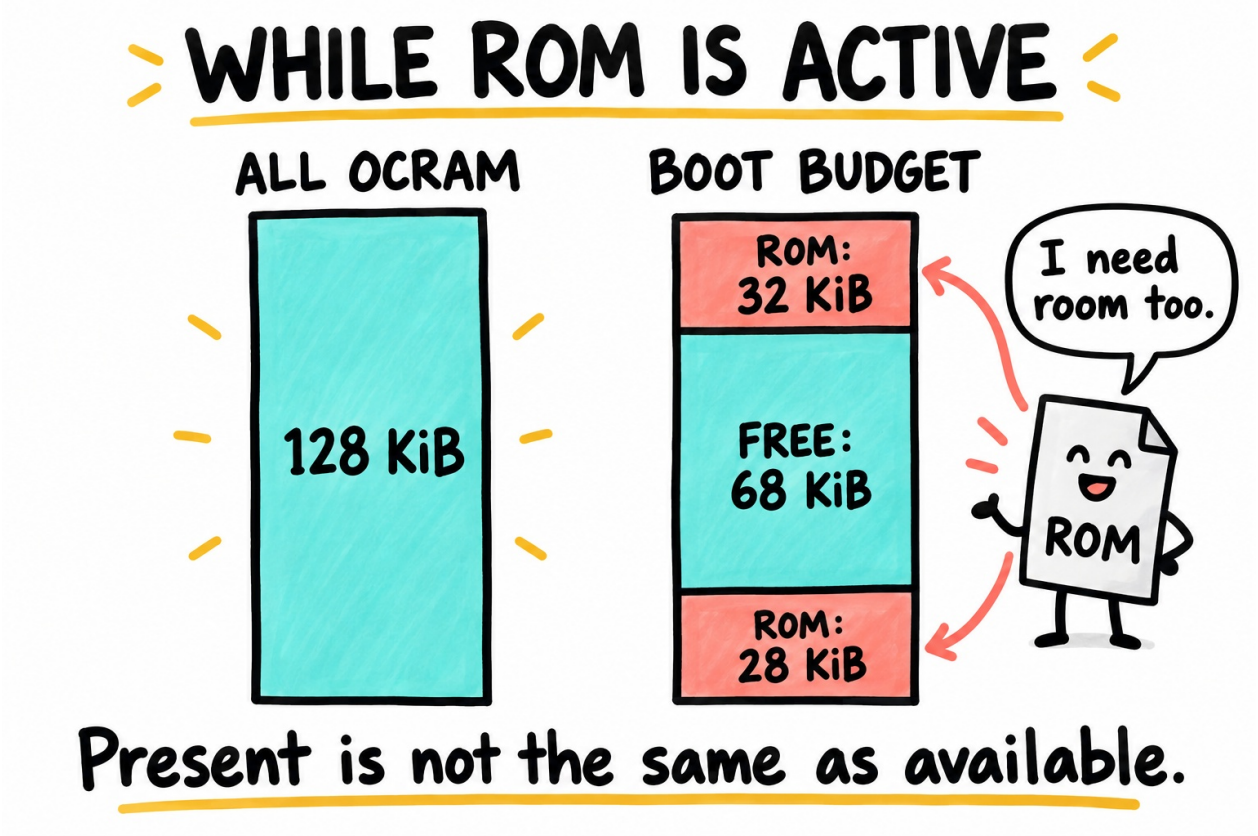
This creates a boot problem worth noticing. A board may have hundreds of megabytes of DDR, yet our first program cannot assume it is usable. Something must configure its controller first. **OCRAM begins at `0x00900000`** and gives us on-chip SRAM before DDR is ready. Our first bare-metal images run there. A small U-Boot **SPL**, or Secondary Program Loader, can also begin there, initialize DDR, and load a larger stage.

Third, most peripheral registers lie in **AIPS-1, AIPS-2, or AIPS-3**. For example, the Clock Controller Module, **CCM**, starts at `0x020C4000`, inside AIPS-1. The high address digits help you identify the containing region, but use the block’s own register table for the exact address.

The full map is RM Chapter 2. In particular, `0x00018000-0x000FFFFF`, `0x00100000-0x00107FFF`, and the region containing `0x08000000` are reserved, not the ROM aliases or EIM base sometimes listed for other configurations.

5.4 OCRAM and the boot footprint

OCRAM solves the first-memory problem, but there is a catch in its budget. The chip provides 128 KB; the Boot ROM leaves only about 68 KB free while it is loading. Both numbers are correct. The ROM is a running program with its own memory needs. RM 8.4.1, Figure 8-3 gives the boundaries below; it does not assign further subdivisions within the lower reservation.



The coral regions are RAM reserved for the ROM's work, not the ROM storage itself. While the ROM is running, our image must fit between them. The table below gives the exact addresses. The drawing is not to scale, and these reservations are not a permanent post-handoff limit.

Start	End	Approx size	During Boot ROM execution	What it means for us
`0x00900000`	`0x00906FFF`	28 KiB	Reserved.	Do not load over it while ROM is active.
`0x00907000`	`0x00917FFF`	68 KiB	OCRAM free area.	Image headers, loaded payload, and any concurrent allocations must fit.
`0x00918000`	`0x0091FFFF`	~32 KB	ROM MMU table, stack, and per-boot state.	Reserved until the ROM has handed off.

During Boot ROM execution, the usable window is the ~68 KB middle range. **After the ROM transfers control and its services are no longer needed**, software can use the entire 128 KB OCRAM range, `0x00900000-0x0091FFFF`.

That gives our loader a different limit from the physical memory limit: loaded bytes must remain below `0x00918000`, not merely below `0x00920000`. In the first padded image, we use `0x00907000` as the image base, `0x00907400` for the IVT header, and `0x00908000` for the code entry. Chapters 6 and 7 show how those choices fit together.

The time at which we use the memory matters as much as its address. We cannot borrow the ROM's stack while the ROM is still using it to load our image. Budget the loaded bytes and our own stack explicitly; the linker script in Chapter 6 will turn that boundary into a build check.

5.5 The clock tree (at one level of detail)

Return to the UART whose correct register writes did not produce output. Finding its address tells us where to write, but the block also needs a clock. The full clock tree has roughly 200 leaf signals. We can make a useful start without following all of them: trace one source through the tree to the peripheral we need.

```
External oscillators → PLLs (ANATOP) → Root clocks (CCM) → Gates (CCGR) → Peripherals
```

Layer 1, Oscillators

- **XTALOSC24M**: the 24 MHz crystal reference for the main clock tree. The Point Atom MINI has a 24 MHz crystal on Y2.
- **XTALOSC32K**: 32.768 kHz crystal for the RTC / SNVS domain. Optional. If absent, the RTC is less accurate.

Layer 2, PLLs (in the ANATOP block)

Seven PLLs:

PLL	Example output/rate	Purpose
PLL1, ARM PLL	Depends on DIV_SELECT	Feeds ARM root through downstream division; not automatically the final core rate
PLL2, System PLL	528 MHz normal intended rate	Bus clocks, peripherals; a selection field is not approval for arbitrary rates
PLL3, USB1 PLL	480 MHz normal intended rate	USB, peripheral references

PLL	Example output/rate	Purpose
PLL4, Audio PLL	variable (44.1/48 kHz multiples)	SAI
PLL5, Video PLL	variable (LCD pixel rates)	eLCDIF
PLL6, ENET PLL	500 MHz internal PLL rate	Divided reference outputs serve Ethernet; this is not a 500 MHz PHY reference
PLL7, USB2 PLL	480 MHz	Host USB

A **PLL**, or Phase-Locked Loop, produces faster clocks from a reference. PLL2 and PLL3 also expose four **PFDs**, Phase Fractional Dividers, per PLL. These provide divided outputs; PLL2_PFD2 at 396 MHz, for example, is commonly used as a source in the AHB clock path.

Layer 3, Root clocks (in CCM)

The CCM selects among PLL and PFD outputs and applies dividers to produce named **root clocks**. Names such as `AHB_CLK_ROOT`, `IPG_CLK_ROOT`, and `UART_CLK_ROOT` tell you which part of the system they serve. There are roughly 60 of these roots; we follow the one needed by a peripheral rather than configuring them all at once.

The following rates illustrate a later configuration, subject to the fitted part’s limits. They are not the frequencies to assume just after reset:

- ARM core: 696 MHz (PLL1)
- AXI bus: 198 MHz (PLL2_PFD2 / 2)
- AHB: 132 MHz
- IPG (peripheral bus): 66 MHz
- UART input: 80 MHz (PLL3 / 6)

Chapter 13 performs the explicit clock setup. Before then, our ROM-loaded program inherits the ROM’s starting state described in RM 8.4.3, not a configuration from U-Boot. When checking a frequency, follow its source and divider: a PLL’s output rate is not necessarily the rate reaching the CPU or peripheral.

Layer 4, Gates (CCGR0..CCGR6)

The relevant CCGR controls are **two-bit CG fields** in `CCM_CCGR0` through `CCM_CCGR6`, not single enable flags:

- `00`: clock off in all modes
- `01`: clock on in run mode only, off in WAIT/STOP
- `10`: reserved; do not use it as a one-bit enable value
- `11`: clock on in RUN and WAIT, off in STOP under the documented CG semantics

Bare-metal examples often select `11`. Production code selects a state according to the peripheral's run, wakeup, and low-power requirements; `01` is not a universal production setting.

RM Table 18-3, **System Clocks, Gating, and Override**, connects clocks to gate fields; the UART rows are on pages 640-641. Then read the actual CCGR register description, such as 18.6.28 for UART1's CCGR5 field.

If a peripheral's registers give unexpected values or writes seem to have no effect, check its clock gate early. An address can be correct while the block behind it is not receiving the clock it needs.

5.6 IOMUX, the universal multiplexer

Suppose we have found the UART registers and enabled the clock. There is still no reason to expect TX at just any convenient header pin. Most package **pads**, the chip's external signal connections, offer several alternate functions: GPIO, UART, I2C, Ethernet, or a timer signal. The **IOMUX**, the I/O multiplexer, selects which function reaches a pad. A controller and a connector can both be correct while the selection between them is wrong.

Do not choose a setting from another pad just because its name looks similar. Each pad has its own ALT table in the reference manual's IOMUXC chapter.

Many muxable digital pads have these controls in **IOMUXC**, the I/O Multiplexer Controller; dedicated power, analog, and memory pins do not all follow this pattern:

- A **MUX_CTL** register selecting which ALT (and a few other bits, SION, "Software Input On", which forces the pad's input buffer on even when output-driven).
- A **PAD_CTL** register controlling drive strength, slew rate, pull-up/down, hysteresis, open-drain.
- Some input functions also require a **SELECT_INPUT** register. This register chooses which eligible pad feeds the peripheral input. NXP calls this selection a "daisy chain."

The main IOMUXC block starts at `0x020E0000`. The SNVS pad group has a separate **IOMUXC_SNVS** block at `0x02290000` [RM 32.5]. For example, BEEP's `SNVS_TAMPER1`/`GPIO5_IO01` mux belongs to that group, not to a register name invented from the main-block pattern.

A typical pin setup is two writes, sometimes three. The following symbolic register pseudocode is not a standalone C program:

```
IOMUXC_SW_MUX_CTL_PAD_GPIO1_IO03 = 0x5;   /* ALT5 = GPIO1_IO03 */
IOMUXC_SW_PAD_CTL_PAD_GPIO1_IO03 = 0xB0B1; /* pad electrical settings */
/* if the pin's function has a SELECT_INPUT, write that too */
```

In this illustrative value, `ODE=0` selects push-pull rather than open-drain behavior and `SRE=1` selects fast slew. It is not a universal GPIO/UART/I2C setting. SPEED encodes pad bandwidth, not an output signal frequency; DSE is a drive-strength encoding, not a fixed current in milliamps. Chapter 9 decodes the full value before using it for its checked pad.

We can now account for the opening example. MUX_CTL selects the function, PAD_CTL sets electrical behavior, and SELECT_INPUT, where present, chooses the input route. Setting the UART's own registers does not perform these jobs for it. When a signal is missing, check this path after the clock path instead of repeatedly changing the peripheral's data register.

The IOMUX tables occupy about 300 pages, but each experiment uses only a few pads. The schematic tells you which pad to search for; its table tells you the setting to use.

5.7 Power domains and the SNVS

Clocks and pads also depend on the board's supplies. Three names recur in the power and low-power chapters:

- **VDD_SOC**: the main digital domain. Required supplies depend on the selected state: suspend/DRAM retention is not the same as SNVS-only operation. Do not translate "deep sleep" into an instruction to switch this rail off; consult the electrical datasheet's power-state requirements.
- **VDD_ARM**: the CPU domain, whose voltage/frequency control supports **DVFS**, dynamic voltage and frequency scaling. Domain names and internal regulator controls do not prove independently switchable external board rails.
- **SNVS**: Secure Non-Volatile Storage, including RTC, security state, tamper inputs, and retained registers. For example, `SNVS_LPGPR` is a 32-bit retained general-purpose register [RM 48.7.14], not a general byte-addressable SRAM. Retention requires its supply to remain present; board battery wiring matters.

This book first uses SNVS for its real-time clock and retained registers. Later chapters also discuss its security and low-power functions.

For electrical work, distinguish supply inputs such as `VDD_SOC_IN` from internal-regulator capacitor nets such as `VDD_ARM_CAP`/`VDD_SOC_CAP`. The supplied core schematic uses a shared external `VDD_ARM_SOC_IN` net. Follow the fitted board's regulator and sequencing design, not a generic three-domain drawing [industrial datasheet, Table 14 and Sections 4.1.6.2-4.1.6.3].

5.8 Fuses and identification

Some settings outlast both a reset and removal of power. The **OCOTP** block exposes one-time-programmable fuse fields and their shadow registers. NXP's programmed chip identity is different from fields a manufacturer may provision for boot, security, or a MAC address, and from the locks protecting those fields [RM Chapter 5]. A MAC slot can be unprogrammed, with the board obtaining its address elsewhere.

Reading a shadow register does not program a fuse. This symbolic example reads identity fields only:

```
uint32_t chip_id_lo = OCOTP_HW_OCOTP_CFG0;
uint32_t chip_id_hi = OCOTP_HW_OCOTP_CFG1;
```

Boot configuration can depend on both fuses and external boot-mode pins. OTP fuse programming is permanent, so this book reads OCOTP fields before it writes any. Never program boot or security fuses without checking the reference manual, the board schematic, and the recovery consequences.

5.9 Peripherals catalog

The peripherals you will touch in this book, with reference-manual chapter numbers (rev 1, 11/2017):

Block	RM Chapter	Notes
CCM	18	Clock controller, Ch 5, 13
Analog CCM (ANATOP)	18.7	PLL registers, Ch 5, 13
IOMUXC	32	Pin mux, every peripheral chapter

Block	RM Chapter	Notes
GPIO	28	Five banks with 32-bit registers; usable GPIOs depend on bonded pads, package, muxing, and board routing, Ch 9, 44
GIC	(ARM TRM)	Interrupt controller, Ch 4, 15
GPT	30	General-purpose timer, Ch 16
EPIT	24	Enhanced periodic interrupt timer, Ch 16
UART	55	8 UARTs, IrDA-capable, Ch 12
I²C	31	4 I²C controllers, Ch 46
ECSPI	20	4 SPI controllers, Ch 47
USDHC	58	SD/MMC and boot media, Ch 11
MMDC	35	DDR controller, Ch 14
FEC	22	Ethernet MAC, Ch 52
USB	56	OTG + host, Ch 55
eLCDIF	34	LCD controller, Ch 54
CSI	19	Camera input, optional
SAI	45	Audio I²S, Ch 53
ADC	13	12-bit, 10 channels, Ch 49

Block	RM Chapter	Notes
PWM	40	PWM, Ch 48
SNVS	48	RTC + secure storage; bare-metal RTC
OCOTP	37	Fuses
WDOG	59	Watchdog
GPC	27	Power controller
SRC	51	System reset controller

Use this as a lookup table when one of these names appears in a design. The chapter links explain its use in the book; the RM column takes you to the register definitions.

5.9a Board revisions and available interfaces

A block in the chip diagram is not a promise that the matching connector or device is on your desk. Check three things separately: whether the SoC supports the function, whether the needed part is fitted on the core board, and how the baseboard connects it. The table is a reference for those decisions, not a list of extra hardware to buy before reading the LED example.

Our reference documents are the MINI v2.2 schematic and the core schematic with internal project title `CL6Y2CB_V1.9`, distributed under a CORE v2.0 filename. Compare the internal titles with your PCB silkscreen. The table below identifies the hardware assumptions behind each example:

Hardware in the reference setup	What to check	Related chapter
User LED, GPIO1_IO03	Pad and active level	Assembly LED

Hardware in the reference setup	What to check	Related chapter
KEY0 / BEEP	MINI V2.2 KEY0: UART1_CTS / GPIO1_IO18, ALT5, active-low; BEEP: SNVS_TAMPER1/ GPIO5_IO01. Verify actual circuit/ revision.	Button and beep
Built-in UART1 bridge	TX/RX pads, bridge identity, power path	UART
DDR3L on core	Schematic names `NT5CC256M16EP-EK`; compare fitted marking and topology	DDR initialization
eMMC on core	Schematic names `KLM8G1GETF`; verify fitted storage	SD/eMMC
Ethernet	Actual FEC routing, PHY, reset, reference clock	FEC driver, dual Ethernet
LCD/add-on display	Connector, panel supply, timings and touch controller	RGB LCD, HDMI bridge
RTC supply	SNVS/battery wiring	Bare-metal RTC, external RTC
JTAG connector	Population and adapter wiring	JTAG debugging

ALPHA baseboards and NAND cores are revision-dependent alternatives, not verified equivalents of this schematic set. Applying a design to them requires matching schematic/BOM information and identified fitted components. Audio, CAN, light sensors, IMUs, Wi-Fi, and modems may require add-ons: see audio codecs, CAN, light sensors, SPI IMUs, SDIO Wi-Fi, and USB modems. RS485 and GPS are covered in RS485/Modbus and GPS/PPS.

Follow connector **pin numbers through both schematics**, not the core's accessory aliases alone. CORE sheet 8 labels J2 pin 49 `GPIO_1 / KEY0` and pin 48 `UART1_CTS / KEY2`. MINI V2.2 sheet 1 connects B49 to `GBC_KEY / AP_INT` and B48 to its onboard **KEY0**. Sheet 2 shows that button and its R12 pull-up. The MINI button therefore uses GPIO1_IO18; GPIO1_IO01 is a different module signal. The same core can meet baseboards with different button names.

For another board, re-derive power/reset policy, oscillator sources, DDR topology/timings, boot-storage routing, PHY wiring, and pin assignments. The board-directory pattern organizes those differences; it does not make incompatible hardware work unchanged. Do not use Chapter 14's DDR values without that comparison.

5.10 Navigating the i.MX6ULL Reference Manual

We have used several parts of revision 1 (11/2017) of IMX6ULLRM. Its 4,127 pages are easier to navigate when you give each search a purpose:

1. **Start with Chapter 2 (Memory Maps)** when you need to locate a block's address range.
2. **Use Chapter 1 (Introduction)** for the block overview and part variants.
3. **Keep Chapter 8 (System Boot) beside Chapter 7 of this book** for boot-mode and image-format questions.
4. **For each peripheral, first read:** the overview and block diagram, the initialization sequence, and the descriptions of the registers your code accesses. Read other sections when a specific question requires them.
5. **Keep Chapter 32 (IOMUXC) open in another window.** You will reference it constantly.

Worked case: UART1 from controller to bridge

The UART1 case brings the manual and schematic together. It follows an address, a clock gate, an interrupt number, and an output route. The LED and I2C cases then show why another valid SoC mux option need not be the route populated on the MINI. These are explanatory lookups, not instructions to connect devices or write registers.

Start with **where**: RM Chapter 2 places UART1 at `0x02020000`, within AIPS-1. We can now locate the controller's registers. That answers an address question, but has not yet told us how to clock the block or reach its pins.

Next find **its clock gate**. UART1 uses `CCM_CCGR5.CG12`, bits 25:24, at `0x020C407C = CCM base 0x020C4000 + 0x7C` [RM 18.6.28]. The two-bit mask is `0x03000000`. These bits select the gate, not the baud rate.

For **the interrupt**, RM Chapter 3 gives UART1 GIC SPI index 26. Adding the GIC's architectural SPI base of 32 gives interrupt ID 58, using the distinction from Chapter 4.

Finally, follow **the board connection**: CORE sheet 6's `UART1_TX_DATA`, ball K14, reaches the `UART1_TXD` net and the core/base connectors; MINI sheet 4 shows the analog-switch routing to the CH340C bridge U8's RX input. U12/U13 and their VUSB-derived control are part of that path. Check the matching power/switch arrangement rather than assuming a direct wire or inventing a switch truth table.

RM 32.6.17, page 1578, gives `IOMUXC_SW_MUX_CTL_PAD_UART1_TX_DATA` at `0x020E0084`; ALT0 selects UART1 TX. This output route does not need RX's SELECT_INPUT setting. The register name, mux choice, and schematic trace now answer different parts of the same question; this is still not a complete initialization sequence.

Worked cases: LED and I2C routing

Lookup	Checked result for this reference route
GPIO1 controller	Base `0x0209C000`; CCGR1.CG13 at `0x020C406C`, bits 27:26. The lower 16 GPIO inputs use GIC SPI index 66, architectural ID 98; no interrupt is needed for an LED output.
LED pad	GPIO1_IO03, mux `0x020E0068`, ALT5; pad-control register `0x020E02F4`. The active level and routed net must match the board schematic before using the LED example.
I2C1 controller	Base `0x021A0000`; CCGR2.CG3 at `0x020C4070`, bits 7:6; GIC SPI index 36, architectural ID 68.
Board-routed I2C1 SCL	UART4_TX_DATA, ball G17; mux `0x020E00B4`, ALT2; SELECT_INPUT `0x020E05A4`, daisy 1.
Board-routed I2C1 SDA	UART4_RX_DATA, ball G16; mux `0x020E00B8`, ALT2; SELECT_INPUT `0x020E05A8`, daisy 2.

The I2C mux choices are in RM 32.6.29/30, and its input selectors in 32.6.328/329. CORE sheets 6/8 identify the routed UART4 pads. GPIO1_IO02/03 are other SoC routes, not this board trace; selecting the IO03 I2C alternative would conflict with its LED use. Pull resistors and attached devices must be checked separately.

The pinned upstream device tree, clock implementation, and pin-function definitions provide independent cross-checks of the controller/field encodings. They do not prove what is fitted or wired on your particular board.

5.11 Key Notes

- A usable peripheral path combines a controller address, a live clock, the selected pad function, electrical settings, and board routing. Correctness at one point does not establish the rest of that path.

- The SoC memory map describes addressable regions, not initialized memory or spare space. DDR requires controller setup, and reserved ranges are not available for arbitrary code.
- OCRAM's physical capacity and the ROM-active loading budget differ. Image headers, payload, and concurrent stack allocations must respect the free window while the ROM still owns its reservations.
- Clock configuration follows sources, PLL/PFD outputs, root dividers, and two-bit gates. A PLL rate is not automatically a CPU or peripheral rate, and a valid encoding does not approve an operating point for the fitted speed grade.
- MUX_CTL, PAD_CTL, and SELECT_INPUT have distinct jobs. The MINI's routed UART4 pads serve I2C1, while selecting an alternative I2C route on GPIO1_IO03 would conflict with its LED use.
- Carrier aliases do not establish SoC identities. Connector pin traces across both schematics explain why MINI KEY0 uses GPIO1_IO18 even though a core connector labels a different signal KEY0.
- Supply-domain names and fuse shadow reads do not authorize rail switching or permanent provisioning. Power sequencing, retained-state supply, and recovery after fuse changes remain board-specific design constraints.

5.12 Pitfalls

- **Ignoring document and silicon revisions.** This book cites RM rev. 1, 11/2017. Check errata for your silicon mask; they can document behavior or workarounds without changing a field definition. Compare newer manuals explicitly rather than silently mixing section numbers.
- **Believing the i.MX6ULL has a Cortex-M4.** It does not. The bigger i.MX6 SoloX / 7Solo have one. The 6ULL is single-A7 only.
- **Using an overview as a complete wiring map.** The datasheet summarizes the device; RM Chapter 1 gives the fuller block view. The board schematic supplies its actual connections.
- **Treating UL/ULL or speed grades as interchangeable.** `G` and `Y` name different families. Decode the complete ULL marking and its feature/speed grade before applying a clock or peripheral configuration.

5.13 Going deeper

- **IMX6ULLRM**: *i.MX 6ULL Applications Processor Reference Manual*, rev. 1, 11/2017.
- **IMX6ULLIEC**: *i.MX 6ULL Industrial Electrical Characteristics* (timings, IO drive characteristics).

- **IMX6ULLHDG**: *i.MX6ULL Hardware Development Guide*, available in NXP's i.MX6ULL documentation. Check the document's actual title/revision before using a cited application-note number.
- The **Point Atom MINI schematic** (provided with your board). You will look at this constantly.

An address such as `0x00908000` no longer needs to be an unexplained number copied from a project. It belongs to OCRAM, with space around it that must be budgeted for the ROM and our stack. How do we get our first instruction to that address? Chapter 6 takes the source file through the tools that make that placement.

Chapter 6: The toolchain

Look inside a typical firmware build directory: object files, an ELF, perhaps a raw binary. They all came from the project, but they are not interchangeable copies. One still needs linking. Another contains addresses and debugging information. The raw file has shed much of that information, even though it contains the bytes we may eventually load.

The Build button in an MCU IDE collected those steps for you. Here the examples separate them so their results can be inspected. A tiny Linux-target program and a build-only bare-metal skeleton have different startup and linking contracts. Neither example requires board execution or completion of a build to follow the explanation. One small variable in the skeleton gives us a particularly useful question: how can it occupy four bytes in RAM without storing those four initialization bytes in the raw image?

6.1 gcc is not one program

Keep the application small enough that it cannot hide the build process from us. Source Chapter 3's environment file, make a directory for the experiment, and open `hello.c`:

```
$ . ~/imx6ull/scripts/env.sh
$ mkdir -p ~/imx6ull/src/ch06-hello
$ cd ~/imx6ull/src/ch06-hello
$ nano hello.c
```

Save this complete source:

```
#include <stdio.h>
int main(void) { puts("hello"); return 0; }
```

Build it for Arm Linux with the Linux-target compiler. We will inspect this result on the PC, not run it directly on the x86_64 host:

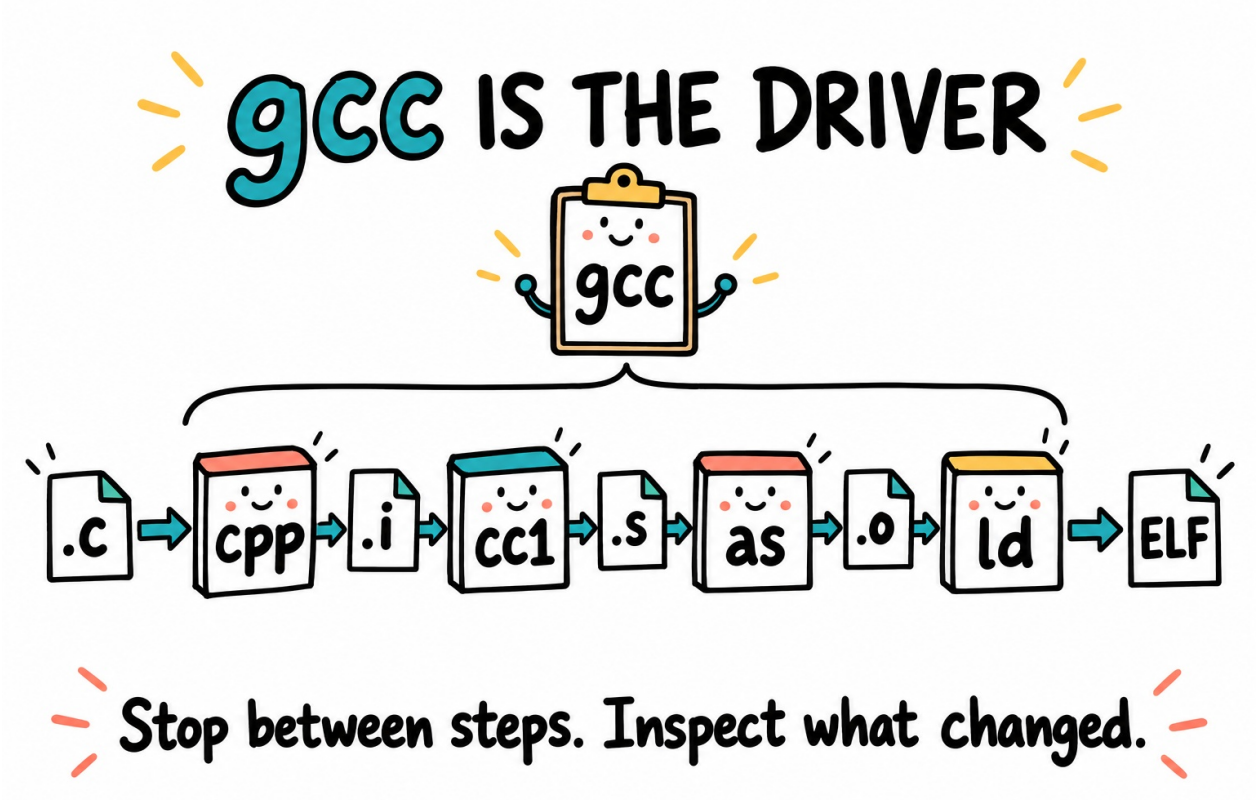
```
$ arm-none-linux-gnueabihf-gcc -O2 -o hello hello.c
$ echo $?
```

Run one command at a time. Immediately after GCC, `echo $?` prints its exit status: `0` means success; a nonzero value means the build failed. If it fails, stop and fix the first error before inspecting `hello`; an older file can remain after a failed rebuild. Chapter 3's pinned Linux compiler defaults to an Armv7-A instruction set suitable for this small Cortex-A7 example. The bare-metal build below selects its CPU and ABI explicitly.

We gave the terminal one command and got one file, `hello`. That tidy exchange hides several stages. The `gcc` command is a **driver**: it selects tools and passes each stage's result onward. The finished file uses **ELF**, the Executable and Linkable Format. It describes the target and layout as well as carrying the code. It is not just a sequence of the instructions for `main()`.

The main build stages are:

1. **Preprocessing.** Resolves `#include`, expands macros, and normally strips comments. GCC commonly performs this inside `cc1`, rather than launching a separate `cpp`. Try `-E` to inspect expanded C; its output retains source-location markers and may retain pragmas.
2. **`cc1`, the C compiler proper.** Parses C, builds an internal IR (RTL/GIMPLE), optimizes, lowers to target assembly. Output: a `.s` file. Try `-S` to stop here.
3. **`as` (from binutils), assembler.** Turns `.s` into a relocatable `.o` (ELF object file). Try `-c` to stop here.
4. **Linking.** `ld` (from binutils) combines `.o` files and libraries, resolves symbol references, applies the linker script's address layout, and writes the final ELF executable. This GCC build may invoke it through `collect2`, a wrapper that also helps arrange constructor initialization; that is not a separate build stage.



The Build button had a team behind it. This is a map of stages, not a promise of four separate processes: `cc1` can also perform preprocessing, and a GCC build may invoke the linker through `collect2`. Intermediate files may be temporary unless we ask the driver to stop or save them.

For a dynamically linked program, the final ELF also names a dynamic loader. That loader runs later on the target when the program starts. It is not a stage of the host compile command.

To see the commands actually executed, rebuild with `-v` and save the complete diagnostic stream:

```
$ arm-none-linux-gnueabihf-gcc -v -O2 -o hello hello.c 2> hello-build.log
$ echo $?
$ less hello-build.log
```

`2>` sends standard error, where GCC writes diagnostics and verbose commands, to the named file. Check the exit status before continuing: on failure, read the log to find the first error, then fix it and rebuild. In `less`, scroll through the complete log and press `q` to return to the shell. Find the `cc1`, `as`, and `collect2`/linker command lines; the compiler's identity banner alone does not show the chain. Do not pipe a live build into `head`: closing its output early can interrupt the build.

For a command plan without compiling, use `arm-none-linux-gnueabihf-gcc -### -O2 -o hello hello.c`. It prints the selected commands but creates no new executable. See GCC's overall options.

The distinction becomes useful the first time a build fails. A missing header points toward preprocessing or compilation. An undefined function reference points toward linking: the compiler may already have produced a valid object, but the linker could not find the function's implementation. Before changing flags, find which stage actually stopped.

6.2 The binutils inventory

An executable is no longer editable C, but it is not beyond inspection. **Binutils** supplies tools that assemble, link, convert, and examine binary files. Want to know where `_start` ended up? Ask `nm`. Want the instructions? Ask `objdump`. The prefix selects the target family, just as it did for GCC. The table collects these tools for later lookup:

Tool	What it does	When you reach for it
`as`	Assemble `.s` into `.o`	Usually invoked through `gcc`
`ld`	Link object files and libraries into an ELF	Bare-metal builds using `-T linker.ld`
`objcopy`	Convert formats and extract sections	Convert an ELF into a raw binary

Tool	What it does	When you reach for it
`objdump`	Disassemble code and display headers or sections	Inspect generated instructions
`nm`	List symbols in an object	Find where a symbol is defined or exported
`readelf`	Display ELF metadata	Inspect sections, segments, and dynamic information
`strip`	Remove symbols/debug info	Producing the shipped binary.
`ar`	Build/dissect `.a` archives (static libraries)	When making your own libs.
`addr2line`	Map address → file:line	Decoding crash addresses, oopses.
`size`	Print section sizes	Quick sanity check on memory budget.

There are two example artifacts in this chapter: the Linux `hello` from Section 6.1 and `led.elf` from the build-only skeleton. The inspection commands assume the corresponding successful build. They describe the artifacts' interfaces, not a prerequisite for continuing to read.

`objdump -d` on the build-only skeleton

```
$ cd ~/imx6ull/src/ch06-skeleton
$ arm-none-eabi-objdump -d led.elf
```

In this skeleton's disassembly, `_start` belongs at `0x00908000`. The stack-pointer setup, BSS-clearing loop, and branch to `main` come from the startup source in Section 6.4.1. Exact instruction addresses and bytes depend on the build; they need not match a representative listing from another build.

`readelf -l` to see segments

```
$ cd ~/imx6ull/src/ch06-hello
$ arm-none-linux-gnueabihf-readelf -l hello
```

Find the `LOAD` rows and, for a dynamically linked result, the `INTERP` row. The output identifies the required interpreter, commonly `/lib/ld-linux-armhf.so.3` for this glibc hard-float target. Record the actual output; header counts/addresses vary with toolchain options.

That `INTERP` filename explains one otherwise puzzling failure: an executable can be present on the board and still fail to start because the loader it requests is missing. The target kernel needs that loader to arrange shared libraries before startup reaches `main`. Successfully producing the executable did not supply the target's whole runtime.

Our bare-metal ELF has no `INTERP` segment. There is no Linux loader to prepare its runtime. We must arrange startup ourselves, and the boot image must tell the ROM where to enter that code.

6.3 Section, segment, VMA, LMA

The inspection tools seem to disagree about the shape of our file. One lists `.text` and `.bss`; another lists `LOAD` segments. They are looking at it for different jobs. The linker organizes contents, while a loader needs ranges of memory to prepare. Keep those two readers of the file in mind as we distinguish sections, segments, and the addresses called VMA and LMA.

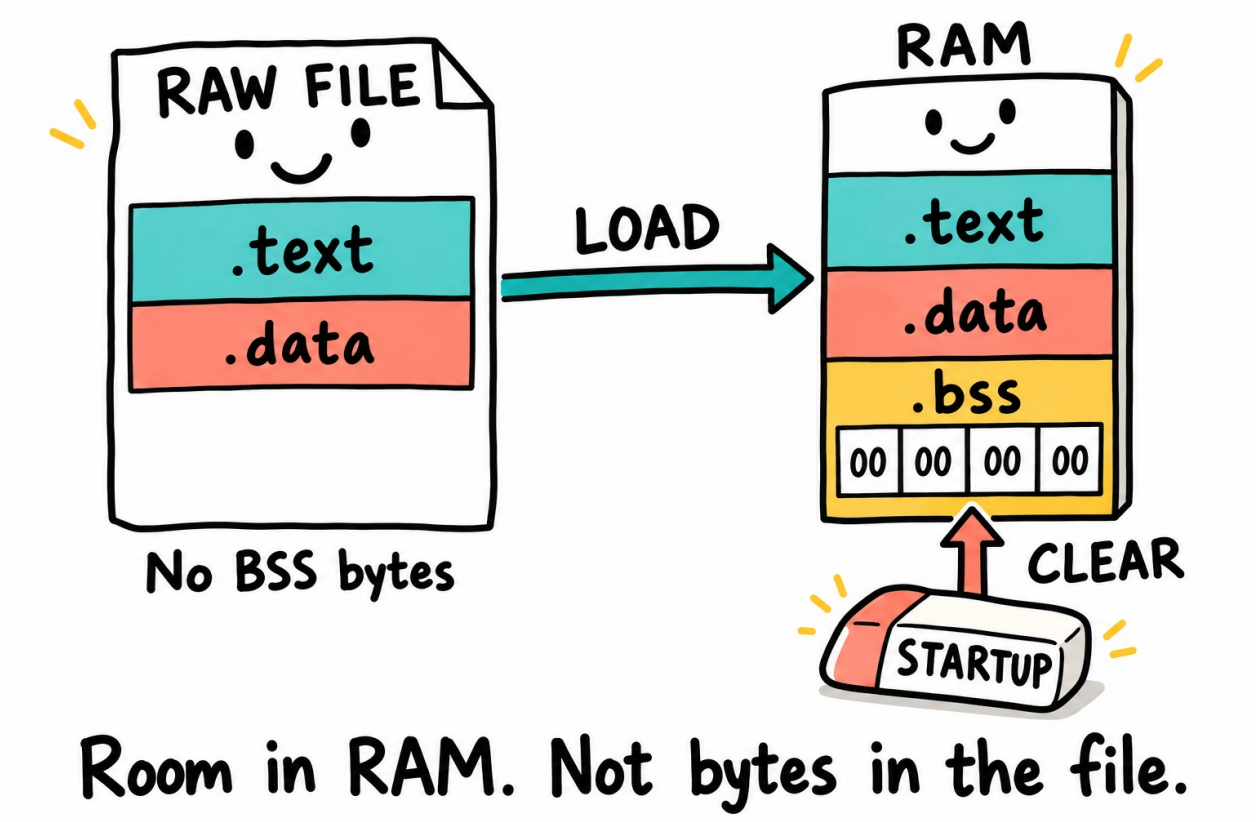
Word	Who mainly cares?	Meaning
Section	Compiler and linker	A named bucket of related bytes: `.text`, `.rodata`, `.data`, `.bss`, `.debug_*`.
Segment	Loader	An ELF program-header description. `PT_LOAD` describes a loadable memory range that can contain several sections; other types describe the interpreter, metadata, or runtime policy.
VMA, virtual memory address	CPU at runtime	The memory address the code/data expects to have when it is being used.

Word	Who mainly cares?	Meaning
LMA, load memory address	Loader/startup code	The memory address at which the section's initial bytes are loaded or stored, possibly different from its runtime address.

If you come from MCU work, start with **sections**. You already know these:

- `.text`: executable instructions.
- `.rodata`: constants and string literals.
- `.data`: globals/statics with non-zero initial values, such as `int led = 1;`.
- `.bss`: globals/statics that start as zero, such as `int counter;`.

There is a clue to the opening variable puzzle in that list. `.bss` describes zero-initialized storage; it does not need to store one zero byte for every runtime byte. Something at startup must make the storage zero. The linker arranges these sections, while the loader works with larger ranges it can load or map with permissions.



Startup clears the counter's four RAM bytes. In this raw bare-metal workflow, the ROM loads bytes but does not read an ELF `.bss` description to clear that storage. The pictured `.data` is an example of initialized data. The build-only skeleton's counter-only program has none. The labels name byte ranges, not a section table stored in the raw file.

Put those sections beside a possible Linux loader layout:

```
sections:  .text  .rodata  .data  .bss  .debug_*
             |       |       |      |
             v       v       v      v
segments:  LOAD R-X        LOAD RW       debug is not loaded
```

So a **segment** is the loader's view. For a Linux process, the kernel and dynamic linker use the ELF program headers to create mappings for `LOAD` segments. Our raw bare-metal image takes a different route: `objcopy` extracts bytes from the ELF, and the ROM image wrapper supplies the loading information. The ROM does not read our ELF section table.

Now separate the two address questions:

- **VMA** answers: "Where will this section live when the CPU uses it?"
- **LMA** answers: "At what memory address are the initial bytes loaded or stored?"

A **file offset** is a different coordinate: it locates bytes inside a stored file. Neither VMA nor LMA is a file offset. For example, `.data` can have an LMA in MCU Flash and a VMA in RAM; the ELF has its own file offset for those initial bytes. See GNU ld's basic concepts.

In a Linux process, a VMA is normally a virtual address. In early bare-metal code, before the MMU is enabled, it is usually the physical address from which the CPU executes or accesses data. For this chapter, think of VMA as the **runtime address**.

Simple OCRAM code can have VMA = LMA. In the first-image layout, the IVT is at `0x00907400`, but code begins at `0x00908000`. The header is not the first instruction:

```
image in OCRAM:
  IVT    at 0x00907400, not executable code
  .text  VMA = LMA = 0x00908000
  .data  VMA = LMA = its linked address after code/constants
```

The distinction becomes useful when **where bytes are stored** differs from **where they are used**. A familiar Cortex-M Flash-and-RAM project already has such a case:

```
Flash image                          RAM after startup
-----------                          -----------------
.text   runs from Flash              .data  variables live here
.rodata stays in Flash               .bss   zeroed here
.data  initial values  ----copy---> .data  runtime values
```

For `.data` in that system:

Address kind	Example meaning
VMA	RAM address where `led` lives when C code reads/writes it.
LMA	Flash address where the initial value of `led` was stored in the image.

That is why Flash/RAM startup copies `.data` from LMA to VMA, then zeros `.bss`. `AT(addr)` gives a section a different load address. Our OCRAM-only skeleton does not need that copy: its initialized data is loaded at its runtime address. Later OCRAM-to-DDR relocation makes the distinction useful again.

`objdump -h` displays VMA and LMA side by side. They agree for this OCRAM skeleton; later relocation into DDR may require them to differ. Its output also exposes a third coordinate, the ELF file offset:

Coordinate	Where the first code byte is
ELF file offset	The `.text` file offset reported by `objdump -h`, which is build-specific.
Raw `led.bin` file offset	`0`: `objcopy -O binary` starts at the lowest copied section's LMA and drops ELF metadata.
Wrapped `.imx` file offset in Chapter 7's layout	`0x1000`: the wrapper adds padding and ROM headers before the raw code.
Runtime memory address	`_start = 0x00908000`, as linked.

6.4 Linker scripts

A linker script (`.ld` file) is a small text file that tells `ld`:

1. What memory regions exist and their attributes.
2. Which sections go into which regions.
3. Where the entry point is.
4. What symbols to export (`_etext`, `_sdata`, `_edata`, `_sbss`, `_ebss`).

This complete `link.ld` is one input to the build-only skeleton. It aligns code with Chapter 9's entry address but does not create a ROM header:

```
ENTRY(_start)

MEMORY
{
    OCRAM (rwx) : ORIGIN = 0x00908000, LENGTH = 0x0000F800
    STACK (rw)  : ORIGIN = 0x00917800, LENGTH = 0x00000800
}

SECTIONS
{
    .text   : { KEEP(*(.text.startup)) *(.text*) } > OCRAM
    .rodata : { *(.rodata*) } > OCRAM
    .data   : { *(.data*) } > OCRAM
    .bss (NOLOAD) :
    {
        . = ALIGN(4);
        _sbss = .;
        *(.bss*) *(COMMON)
        . = ALIGN(4);
        _ebss = .;
    } > OCRAM

    _image_end = .;
    _stack_limit = ORIGIN(STACK);
    _stack_top = ORIGIN(STACK) + LENGTH(STACK);
    ASSERT(_image_end <= _stack_limit, "Image overlaps reserved stack")
}
```

Read the script from top to bottom. `ENTRY(_start)` records the entry symbol for ELF tools and debuggers. Later, the Boot ROM will use the image header's IVT entry instead, so the two descriptions must agree about where our code begins.

`MEMORY` divides the available space into a 62 KiB code/data region starting at `0x00908000` and a separate 2 KiB stack region starting at `0x00917800`. The stack ends at `0x00918000`, the ROM-active free-window limit from Chapter 5. Headers and padding belong below the code; the physical OCRAM extending above that limit is still in use by the ROM during loading.

`SECTIONS` places input sections into the output. `> OCRAM` assigns them to successive addresses in our OCRAM region. Within BSS, the dot means the current placement address, so `_sbss = .;` and `_ebss = .;` give startup the bounds it must clear.

Naming `_stack_top` does not reserve memory by itself. The separate memory budget and `ASSERT` do the important work of keeping linked sections out of the downward-growing stack's region. A larger program that crosses the limit should fail at link time, where we can read the error, rather than first reveal the overlap as strange behavior on the board. `_stack_top` is 8-byte aligned for public function calls, and the BSS bounds are aligned for clearing whole words.

6.4.1 A build-only startup skeleton

The skeleton increments a variable rather than operating an LED. It supplies instructions, startup work, and a BSS object to explain the file-versus-memory distinction. Its four input files belong together in the example directory `~/imx6ull/src/ch06-skeleton`: `link.ld` above, `startup.S` and `main.c` below, and the complete Makefile in Section 6.7.8. No generated file from an earlier assignment is required.

The example names its outputs `led.elf` and `led.bin` to match the later image discussion. The name does not imply LED behavior. With Chapter 3's terminal-local environment selected, `make` in this directory would assemble and compile the sources, link the ELF, and extract the raw binary. The preceding Make syntax fragments are not additional input files.

`startup.S` establishes the limited C-entry contract modeled here:

```
.syntax unified
.cpu cortex-a7
.arm
.section .text.startup, "ax"
.global _start
_start:
    cpsid   if
    ldr     sp, =_stack_top
    ldr     r0, =_sbss
    ldr     r1, =_ebss
    mov     r2, #0
1:
    cmp     r0, r1
    bhs     2f
    str     r2, [r0], #4
    b       1b
2:
    bl      main
3:
    b       3b
.ltorg
```

First, `cpsid if` masks IRQ/FIQ and the first `ldr` sets SP to the linker-defined stack address. The next loads obtain the BSS bounds, and `mov` supplies the zero used to clear it. The loop compares the current address with `_ebss` and clears one word at a time. Numeric labels can be reused: `1b` means the previous label `1`, while `2f` means the next label `2`.

Finally, `bl main` calls C and places a return address in LR. If the function returns, the last branch loops in place. `.ltorg` emits the literal constants needed by the address loads. The linker script's alignment and bounds are therefore an interface used by startup, not merely reporting symbols.

`main.c` supplies the zero-initialized object:

```
volatile unsigned int counter;
int main(void)
```

```
{
    while (1) {
        counter++;
    }
}
```

For this target, `counter` occupies four bytes in BSS. `volatile` keeps its loads and stores present. Startup supplies the initial zeros, while `objcopy` supplies no BSS initialization bytes in `led.bin`. In `readelf -lW led.elf`, BSS contributes to the loadable range's `MemSiz` without contributing those four bytes to `FileSiz`.

This is a build-only skeleton, not a boot candidate. Do not transfer or try to boot `led.bin`. It has no ROM header, observable LED operation, full exception setup, or complete ROM-handoff normalization. Chapter 9 explains a distinct assembly image and its wrapper.

The minimal linker script can produce an RWX load-segment warning because text and writable BSS share one ELF loadable range. That describes the ELF layout, not hardware permissions or a suitable Linux process layout. A successful link establishes only that these inputs satisfy the selected build rules. A failed rebuild may leave older output files, which are not evidence of its success.

6.5 The ABI: what makes function calls work

Suppose one source file calls a function compiled from another file. The compiler cannot choose argument registers independently for each one. Both sides need the same **Application Binary Interface**, or ABI. In particular, they must agree on:

- Which register contains the first argument? The second? Where do return values live?
- Which registers must `bar()` preserve, and which can it freely clobber?
- How is the stack aligned?
- Where do floating-point arguments go, in integer registers, or in FPU registers?
- How does a function return a structure rather than a single integer?

The **EABI** (Embedded ABI) for ARM specifies all of this. ARMv7-A Linux uses the **AAPCS** (ARM Architecture Procedure Call Standard) plus the EABI's runtime conventions. The relevant variant for us is **AAPCS-VFP**, also called "hard-float," in which floating-point parameters use `s0-s15` / `d0-d7` rather than integer registers.

The core rules (simplified):

Register	AAPCS role
r0-r3	First four integer arguments / return value (`r0`, optionally `r0`,`r1` for 64-bit). Caller-saved.
r4-r11	Callee-saved (must be preserved). R9 has a platform-specific role; see Chapter 4.
r12 (ip)	Intra-procedure-call scratch. Caller-saved.
r13 (sp)	Stack pointer. 8-byte aligned at function boundary.
r14 (lr)	Link register (return address).
r15 (pc)	Program counter.
s0-s15 / d0-d7	Applicable hard-float arguments/return values; allocation depends on types and signature, with different variadic-call rules. Caller-saved.
s16-s31 / d8-d15	Callee-saved FP.

Chapter 4 showed that returning to the right instruction is not enough if the interrupted state was damaged. Ordinary function calls have a related, but different, agreement: the caller knows which registers may change and which the callee must preserve. If handwritten assembly overwrites a callee-saved value and returns without restoring it, the caller resumes with the wrong state. Compiled C follows the ABI; our assembly must keep its side of the agreement too.

Hard-float vs soft-float

Floating-point options describe both the instructions used and how arguments cross a function boundary. GCC provides three selections:

- **soft-float** (`-mfloat-abi=soft`) uses software floating-point operations and the base calling convention, with floating-point arguments passed through core registers or the stack as applicable.
- **softfp** (`-mfloat-abi=softfp`) permits hardware floating-point instructions while retaining the base calling convention.

- **hard-float** (`-mfloat-abi=hard`) permits hardware instructions and uses the VFP calling-convention variant for applicable arguments.

The practical difference is compatibility, not a promise that one option always makes a program faster. See GCC's Arm floating-point options.

The Linux triplet suffix normally distinguishes `gnueabi` from hard-float `gnueabihf`. Objects/libraries with incompatible floating-point calling conventions must not be combined; the linker often diagnoses their attributes. A prefix alone does not prove that every freestanding object is incompatible. Check `readelf -A` and the matching runtime, and use one consistent compiler/ABI selection per build.

The Linux toolchain selected for this book uses the hard-float ABI. All Linux user-space objects and libraries that we combine must use the same ABI.

6.6 The C library, or its absence

Where did the code for `puts()` come from? We wrote a call and included a header, but supplied no function body. The header provided declarations, and a library provided the implementation. That arrangement is convenient for Linux user space. Our first bare-metal program has no operating system or runtime prepared for such a call, so Part II begins with **no libc**. Later we write the small `memcpy` and `printf` routines our experiments need.

The C library, or libc, is **separate from GCC**. GCC is the compiler driver and compiler. Libc is a runtime library plus headers that the compiler links against when you build Linux user-space programs. A complete cross-toolchain usually ships all of these together:

Piece	Example	Job
Compiler	`arm-none-linux-gnueabihf-gcc`	Turns C into object files.
Binutils	`as`, `ld`, `objcopy`, `readelf`	Assembles, links, converts, and inspects binaries.
libc headers	`stdio.h`, `unistd.h`, `pthread.h`	Tell the compiler what user-space APIs look like.
libc binaries	`libc.so`, `libc.a`, startup files such as `crt1.o`	Provide the code that implements the C/POSIX runtime.

Piece	Example	Job
libgcc	`libgcc.a`	Small helper routines that GCC itself may need, such as integer division helpers.

The Linux-target headers and libraries live in the toolchain's **sysroot**, a directory resembling a target root filesystem. Glibc is packaged alongside our official Arm compiler, not inside its executable. Inspect the selected installation with `arm-none-linux-gnueabihf-gcc -print-sysroot` and `-print-file-name=libc.a`.

For Linux user-space code, we use a libc. Three options:

libc	Notes
glibc	Common on general-purpose Linux distributions. Broad compatibility.
musl	Small, MIT-licensed implementation often used in compact systems.
uClibc-ng	Maintained fork of uClibc, available in Buildroot and used by some embedded distributions.

The selected official Arm Linux archive supplies glibc. Chapter 34 compares libc choices; a musl build needs matching headers/libraries and toolchain configuration, not merely a flag on a glibc build.

What libc actually provides

A libc bundles:

- **Wrappers around syscalls** (`open`, `read`, `write`, `mmap`, …) so you can call them as C functions.
- **Memory allocator** (`malloc`, `free`, internally calling `brk`/`mmap`).
- **Standard I/O** (`fopen`, `printf`), buffered layers atop the syscall wrappers.
- **Math** (`sin`, `sqrt`, …), in `libm.so`.
- **POSIX threads** (`pthread_*`), sometimes a separate `libpthread.so`, sometimes folded in.

- **Locale, time, network, etc.**

Knowing a function's name is not enough to make it available. In a bare-metal build, `malloc`, `printf`, or `errno` needs an implementation from our code or a deliberately selected library. This is why runtime dependencies matter as much as the target instruction set.

6.7 Make, in working depth

`make` runs the builds for the bare-metal projects in Part II and for U-Boot, Linux, and Buildroot later in the book. This section explains the parts used by those builds.

The first build is only half the work. On the second build, what changed? If you edited `main.c`, the startup assembly did not need recompiling. If you edited the linker script, neither object may need recompiling, but the final layout must be recreated. `make` records those relationships so we do not have to reconstruct the whole command sequence after each edit.

Sections 6.7.1, 6.7.3, and 6.7.4 explain the rules used by the build-only skeleton and its complete Makefile in Section 6.7.8. Variable-assignment variants, functions, conditionals, and parallelism provide later-build reference. The syntax fragments in Sections 6.7.1-6.7.6 are reading examples, not additions to the skeleton's Makefile.

`make` decides which build commands need to run. It does not know C, assembly, ELF, or ARM by itself. You teach it:

1. Which files you want to create.
2. Which input files each output depends on.
3. Which shell command creates the output from the inputs.

With those relationships recorded, `make` can answer: **which outputs need to be rebuilt now?**

It answers by looking at files and timestamps:

- If the output file does not exist, build it.
- If an input file is newer than the output file, rebuild it.
- If the output exists and all inputs are older, skip it.

For our bare-metal skeleton, the main dependency chain looks like this:



Follow a change to `main.c` through the arrows. It reaches `main.o`, then `led.elf`, then `led.bin`; it never reaches `startup.o`. A change to `link.ld` enters further along: reuse the objects, but rebuild the linked ELF and raw binary. The complete Makefile also makes both objects and the ELF depend on `Makefile` itself, so an edited flag rebuilds them. An unchanged second build has no compilation, linking, or conversion to perform.

A `Makefile` records those dependency relationships and the shell commands that produce each output. `make` decides whether to run the commands and in what order.

When you run plain `make`, it reads a file named `Makefile` in the current directory and builds the first target in that file. In our examples, the first target is usually `all`, and `all` depends on the final file we want, such as `led.bin`.

6.7.1 Rule shape

```
target ...: prerequisite ...
<TAB>command
<TAB>command
```

In a rule, the *target* is the file you want to create. The *prerequisites* are the files it depends on. The indented commands are the recipe that creates or updates the target.

`make` builds the target when the target does not exist or when a prerequisite is newer than the target. Recipe commands **must** be indented with a literal `TAB`. Spaces do not work.

6.7.2 Variables: four flavors of assignment

```
CC     = arm-none-linux-gnueabihf-gcc         # 1) recursive ("deferred")
CC    := arm-none-linux-gnueabihf-gcc         # 2) simple ("immediate")
CFLAGS ?= -O2                                 # 3) only if not already set
OBJS  += extra.o                              # 4) append
```

The two examples show how changing `name` affects `greet`. They differ by one character in the assignment:

```
name = world
greet = hello $(name)
name = there
$(info $(greet))   # prints "hello there"   ← deferred expansion
```

```
name := world
greet := hello $(name)
name := there
$(info $(greet))   # prints "hello world"   ← immediate expansion
```

With `=`, the reference to `name` is resolved when `greet` is used, after `name` became `there`. With `:=`, `greet` kept the value formed at assignment time, `hello world`. Our small Makefile uses `:=` where we want that immediate expansion. Larger projects use both deliberately; ask when a value needs to become fixed.

6.7.3 Pattern rules and automatic variables

A pattern rule with `%` matches every file fitting the pattern:

```
%.o: %.c
    $(CC) $(CFLAGS) -c -o $@ $<
```

Inside the recipe, **automatic variables** carry the per-instance pieces:

Var	Meaning
`$@`	The target being built
`$<`	The first prerequisite
`$^`	All prerequisites (de-duplicated, space-separated)
`$+`	All prerequisites (with duplicates)
`$?`	Prerequisites newer than the target
`$*`	The stem matched by `%`

For `main.o`, the rule substitutes `main.c` for `$<` and `main.o` for `$@`. The same recipe can then build other C objects without repeating the compiler command for each filename. Together with the assembly rule and final link rule, this is enough for our small project.

6.7.4 Phony targets

```
.PHONY: all clean install
```

Tells `make` that `all` / `clean` / `install` are **not** filenames. Without `.PHONY`, a file named `clean` would make `make clean` consider the target up to date and skip the recipe. With `.PHONY`, the recipe always runs.

6.7.5 Useful functions

Larger Makefiles often need to derive one filename list from another. These functions use the form `$(name args,...)`. Keep the table as a reading aid; the complete skeleton below does not require learning every function first.

Function	What it does	Example
`$(wildcard PATTERN)`	List files matching a glob (no quoting)	`$(wildcard *.c)` → `a.c b.c`
`$(patsubst PAT,REPL,LIST)`	Pattern substitution	`$(patsubst %.c,%.o,a.c b.c)` → `a.o b.o`
`$(subst FROM,TO,STR)`	Plain string substitution	`$(subst .,_,a.b.c)` → `a_b_c`; a space before `_` would be part of the replacement
`$(dir NAMES)`	Directory part	`$(dir src/a.c)` → `src/`
`$(notdir NAMES)`	Filename part	`$(notdir src/a.c)` → `a.c`
`$(basename NAMES)`	Drop the extension	`$(basename src/a.c)` → `src/a`
`$(addsuffix S,LIST)` / `$(addprefix P,LIST)`	Append / prepend	`$(addsuffix .o,a b)` → `a.o b.o`
`$(filter PAT,LIST)` / `$(filter-out PAT,LIST)`	Keep / remove matching	`$(filter %.c,a.c b.h)` → `a.c`
`$(sort LIST)`	Sort + de-duplicate	`$(sort c a b a)` → `a b c`
`$(shell CMD)`	Run a shell command, capture stdout	`$(shell uname -m)` → `x86_64`

A common idiom collects root-level C files and C files one directory below `bsp`; it does not search the tree recursively:

```
SRCS := $(wildcard bsp/*/*.c) $(wildcard *.c)
OBJS := $(patsubst %.c,%.o,$(SRCS))
```

6.7.6 Conditionals

```
ifeq ($(ARCH),arm)
  CFLAGS += -mcpu=cortex-a7
else ifeq ($(ARCH),aarch64)
  CFLAGS += -mcpu=cortex-a53
else
  $(error Unsupported ARCH=$(ARCH))
endif

ifdef DEBUG
  CFLAGS += -O0 -g3
else
  CFLAGS += -O2
endif
```

These Make directives are evaluated when Make reads the file. They are not tab-indented shell recipe commands, and automatic variables such as `$@` are not available as recipe-time values there. Compare:

```
show:
ifeq ($(DEBUG),1)
    @echo debug selected while reading the Makefile
else
    @echo release selected while reading the Makefile
endif
    @if [ -f main.c ]; then echo main.c exists at recipe time; fi
```

The `ifeq`/`else`/`endif` lines start at column one; the shell `if` starts with a tab. See GNU Make conditional syntax.

6.7.7 Parallelism

```
$ make -j$(nproc)            # use all available cores
$ make -j8                    # 8 jobs in parallel
```

`-j` lets Make run several ready recipes at once. That can shorten a large build, but each job also needs memory, so use fewer jobs on a small VM. If a build fails only in parallel, check the dependencies: one recipe may be using a file another has not finished creating.

6.7.8 A complete Makefile for the build-only skeleton

This complete `Makefile` belongs with the three sources defined in Sections 6.4 and 6.4.1. It is a single build description, unlike the preceding syntax fragments. Recipe lines use literal tabs.

```
CROSS  := arm-none-eabi-
CC     := $(CROSS)gcc
```

```
OC     := $(CROSS)objcopy

CFLAGS := -mcpu=cortex-a7 -marm -mfloat-abi=soft \
          -ffreestanding -fno-builtin -fno-pie -fno-stack-protector \
          -fno-unwind-tables -fno-asynchronous-unwind-tables -O2 -g -Wall
LDFLAGS := -nostdlib -Wl,-T,link.ld

SRCS   := startup.S main.c
OBJS   := startup.o main.o

all: led.bin

%.o: %.S Makefile
    $(CC) $(CFLAGS) -MMD -MP -c -o $@ $<

%.o: %.c Makefile
    $(CC) $(CFLAGS) -MMD -MP -c -o $@ $<

led.elf: $(OBJS) link.ld Makefile
    $(CC) $(CFLAGS) $(LDFLAGS) -o $@ $(OBJS)

led.bin: led.elf
    $(OC) -O binary $< $@

clean:
    rm -f startup.o main.o startup.d main.d led.elf led.bin

-include $(OBJS:.o=.d)

.PHONY: all clean
```

Here the tool selections, flags, and rules finally meet in one file. Follow `all` to `led.bin`, then `led.elf`, then the two objects. The flags explain what environment we are asking the compiler to assume:

- `-mcpu=cortex-a7`: generate code that uses Cortex-A7 features.
- `-marm -mfloat-abi=soft`: this integer-only startup skeleton uses ARM instructions and does not enable VFP. Hard-float code generation alone would not enable the hardware in startup.
- `-ffreestanding`: “I do not have a hosted C environment.” Disables the assumption that `main` is the standard entry, etc.
- `-fno-builtin`: disables compiler’s optimization of calls like `printf` into special builtins.
- `-fno-pie -fno-stack-protector`: do not require a position-independent runtime or stack-check runtime in this freestanding skeleton.
- `-fno-unwind-tables -fno-asynchronous-unwind-tables`: no runtime stack-unwinding tables are needed here.
- `-nostdlib` is a **link** option in `LDFLAGS`: the GCC driver adds no default startup/libc/libgcc. `-Wl,-T,link.ld` forwards the script selection to its linker. This deliberately uses GCC for linking so a later explicit `-lgcc` can select its matching runtime.

- `-MMD -MP` writes `.d` header dependencies; `-include` reads them on the next build. A header edit then rebuilds the affected objects.
- `-O2 -g`: optimize but keep debug info.
- `-Wall`: turn on the warnings everyone should be using.

Adding `Makefile` as a prerequisite means edits to its flags or recipes invalidate the affected outputs. The source remains the first prerequisite, so `$<` still means the source file; the link recipe still passes only `$(OBJS)`, not the script or Makefile, to GCC. Make does not compare command-line variable values with the previous build: after changing `CFLAGS` or `CROSS` on the command line, run `make clean` before rebuilding.

6.8 Static vs dynamic linking (for Linux user-space)

Two ways to combine your code with libraries:

- **Static linking** includes the selected library code in the executable at link time. It does not need those libraries as `.so` files on the target, but the executable is usually larger.
- **Dynamic linking** leaves references to shared libraries that must be available on the target. The dynamic loader, such as `/lib/ld-linux-armhf.so.3`, resolves them when the process starts.

Linux distributions normally use dynamic linking. Small embedded systems may use either model. Static linking simplifies deployment for a few standalone programs, while dynamic linking saves storage when many programs share the same libraries.

Return to the hello directory before these Linux user-space comparisons. To force static:

```
$ cd ~/imx6ull/src/ch06-hello
$ arm-none-linux-gnueabihf-gcc -g -static -o hello hello.c
$ file hello
hello: ELF 32-bit LSB executable, ARM, EABI5 version 1 (SYSV),
       statically linked, BuildID[sha1]=..., with debug_info, not stripped
```

Compare sizes:

```
$ arm-none-linux-gnueabihf-gcc -g -O2 -o hello-dyn hello.c
$ arm-none-linux-gnueabihf-gcc -g -O2 -static -o hello-stc hello.c
$ ls -l hello-dyn hello-stc
$ arm-none-linux-gnueabihf-strip -o hello-dyn.stripped hello-dyn
$ arm-none-linux-gnueabihf-strip -o hello-stc.stripped hello-stc
$ ls -l hello-dyn.stripped hello-stc.stripped
```

Check each compile succeeds before using its output. The two builds use matching optimization/debug flags; only the static-link selection differs. `strip -o` writes separate copies, preserving both debug originals. A static link can also bring debug information

from library archives, so an unstripped hello may occupy megabytes while its stripped copy is much smaller. Record both pairs of actual sizes rather than expecting a fixed libc-size baseline. The smaller dynamic executable still relies on shared libraries on the target; include those dependencies in the complete system's storage budget.

Also keep the target environment in view: a statically-linked Linux program still needs Linux. Some library features still use external files or configuration. `-static` does not turn `hello` into firmware that the Boot ROM can start by itself.

6.9 ELF structure needed for this book

An ELF file has:

```
┌──────────────────────────────────┐
│ ELF Header                       │  ← architecture, type (REL/EXEC/DYN), entry point
├──────────────────────────────────┤
│ Program Header Table             │  ← segments (loader's view)
├──────────────────────────────────┤
│ .text                            │
│ .rodata                          │
│ .data                            │
│ .bss (no actual bytes)           │
│ .symtab, .strtab                 │
│ .debug_*  (DWARF)                │
│ ...                              │
├──────────────────────────────────┤
│ Section Header Table             │  ← sections (linker/debugger's view)
└──────────────────────────────────┘
```

A few key facts:

- **Type `REL`** (relocatable, `.o`), produced by the assembler, fed to the linker.
- **Type `EXEC`** is an executable type, not proof of static linking: a dynamically linked non-PIE executable can also be `EXEC`.
- **Type `DYN`** can describe a shared library, dynamic PIE, or static PIE. Type and runtime-library dependencies are separate questions.
- **`.bss` occupies no file bytes.** It only declares "give me N bytes of zero at runtime." The startup code (or the kernel) zeroes it.
- **DWARF** is the debug-info format used in `.debug_*` sections. `gdb`, `objdump -S`, and `addr2line` read it.

Use `file` for a quick summary, `readelf -l` for an interpreter/program headers, and `readelf -d` for dynamic entries/dependencies. `readelf -S` checks `.debug_*`; `nm` checks ordinary symbols. Missing DWARF can simply mean the program was built without `-g`; it does not prove stripping. See elf(5).

6.10 Key Notes

- The GCC driver coordinates preprocessing, compilation, assembly, and linking. A valid object file does not establish that symbol resolution or final placement succeeded, and an old output file can survive a failed rebuild.
- Toolchain selection includes target instructions, ABI, and runtime assumptions. Linux `hello` uses a libc and Linux services. The bare-metal skeleton supplies its own startup and links without the default runtime. Static Linux linking does not make a program ROM-bootable.
- ELF sections organize linker content, while program headers describe loader ranges and dependencies. Raw conversion removes this metadata. The ROM wrapper must supply the loading and entry contract independently.
- BSS reserves runtime storage without storing its zero initialization bytes in the raw image. The skeleton's startup owns clearing the linker-defined range before C accesses `counter`.
- VMA, LMA, and file offset are different coordinates. Linker-defined entry and stack bounds constrain placement, but a linker assertion cannot prove runtime stack depth or correct ROM handoff.
- AAPCS defines argument locations, saved registers, and stack alignment across independently built code. Floating-point instruction selection and floating-point argument passing are related but distinct compatibility decisions.
- Make records dependency edges and timestamps rather than understanding source semantics. Source, header, linker-script, and Makefile dependencies determine what is rebuilt. Command-line flag changes are not automatically compared with a previous invocation.

6.11 Pitfalls

- **Mixing incompatible toolchain outputs.** Keep the book's Linux and bare-metal builds on their separately selected toolchains. Some freestanding objects can be ABI-compatible across prefixes, but incompatible calling conventions or no-OS runtime dependencies cannot be reused in Linux user space. Check object attributes and required libraries, not just compiler names.
- **`-nostdlib` also removes the automatic libgcc link.** If code uses an operation such as 64-bit integer division, GCC may emit a call to `__aeabi_uldivmod` from `libgcc`. Add `-lgcc` explicitly after your object files when required.
- **Linker-script order matters.** Place a specific startup section before a broader wildcard such as `*(.text*)` when startup must appear first. Chapter 9 shows the required ordering.

- **`.bss` must be zeroed.** If startup does not clear `.bss`, uninitialized globals contain old memory values and program behavior can change between boots.
- **Wrong `-march`/`-mcpu`.** Toolchain defaults vary. The bare-metal Makefile explicitly uses `-mcpu=cortex-a7` for instruction selection and scheduling. The small Linux hello examples use the pinned toolchain's Armv7-A defaults; for a board-specific project, choose compatible CPU/FPU/ABI flags deliberately rather than assuming another compiler's defaults match.
- **`strip` on the binary you wanted to debug.** Keep an unstripped copy. A useful convention in your Makefile: `$(NAME).elf` is unstripped (for `gdb`/`objdump`). `$(NAME).stripped.elf` is the smaller deliverable.

6.12 Going deeper

- *Linkers and Loaders* by John Levine. A detailed explanation of linkers and loaders.
- ELF for the Arm Architecture (AAELF32), alongside the generic ELF specification. AAPCS32 defines calling conventions, not the whole ELF format.
- The GCC manual, at least the section on language-independent options.
- *Procedure Call Standard for the Arm Architecture* (AAPCS32), ARM IHI 0042.
- `man 5 elf`, `man 1 ld`, `man 8 ld.so`.
- LWN: "How programs get run" (the kernel `exec` path. Relevant when you write a `binfmt`).

The disassembly and linker symbols now locate our first instruction at `0x00908000`. Chapter 7 adds the ROM's loading description and entry address around the raw bytes.

Chapter 7: The Boot ROM, IVT, DCD, and BootData

The name `_start` meant something to the linker in Chapter 6. The raw bytes we give the chip do not carry that symbol's name. How does the chip know that our first instruction belongs at `0x00908000`, and that it should begin executing there?

On a microcontroller, the reset-vector arrangement often provides the starting address. On the i.MX6ULL, NXP's Boot ROM expects an image description before it hands over control. Valid instructions are only part of the delivery. They must arrive at the right place, with a header the ROM can interpret. If those descriptions disagree, our program may never reach the code that could print an error.

Keep two questions beside the worked example: where is this byte in the file, and where will it be in RAM? The numbers will differ even when they describe the same header. Following that one image gives the names **IVT**, **BootData**, and **DCD** jobs to do, rather than leaving us with three abbreviations to memorize.

7.1 What the Boot ROM is

The Boot ROM is code NXP puts into the chip during manufacture. Its documented 96 KiB region begins at physical `0x00000000` and includes a protected area. It is not user-programmable flash.

Here we follow normal cold boot. The ROM selects a source, reads the image description, loads code, and hands over control. Low-power wake paths and calls into ROM services are separate subjects. Hardware, boot settings, and security policy determine which loading and recovery paths are available.

Three facts help us choose what to inspect when that first handoff fails:

1. **The expected format is documented.** RM Chapter 8, System Boot, describes the ROM's boot modes and image requirements. Keep it beside the worked example rather than treating a `.imx` filename as evidence of a valid image.
2. **Board setup matters even with the same ROM.** Chips of a given silicon revision have the same ROM; board wiring, boot pins, fuse provisioning, and storage contents can still produce different boot behavior.
3. **SDP is a recovery path.** With functional power/reset/USB hardware and a permitted image, USB Serial Download Protocol can load code without working storage. Chapter 8 checks enumeration; Chapter 9 transfers our first image. Closed-mode authentication is not bypassed by USB mode.

7.2 The boot sequence, step by step

POR_B is the active-low power-on-reset signal. On the cold-boot path, releasing reset lets the ROM begin its work. The following sequence connects that event to our `_start` entry. **CCM** is the Clock Controller Module introduced in Chapter 5, and **SDP** is the Serial Download Protocol used by `uuu`.

1. **Internal initialization.** Set up the watchdog, the ROM's own stack at the top of OCRAM, and a few CCM defaults.
2. **Sample boot mode and configuration.** BOOT_MODE is the first decision; the meaning of `BT_FUSE_SEL` depends on that mode. See the decision table below [RM 8.2.3 and Table 8-2]. No fuse programming is required here.
3. **Select the source or downloader.** Internal boot uses either fuse configuration or the sampled GPIO override according to that decision, not a global "clear means pins" rule.
4. *(Internal boot only)* **Probe the selected device.** SD card, eMMC, NAND, SPI-NOR, QSPI-NOR, or parallel NOR, each has a different probing path.
5. **Acquire the header using the device's boot rules.** RM Table 8-25 supplies the ordinary NOR/OneNAND/SD/MMC/SPI offsets; QSPI has separate configuration requirements:
   - **SD / eMMC / eSD / SDXC**: IVT at **`0x400`**; the initial read covers 4 KiB.
   - **SPI EEPROM (SPI-NOR)**: IVT at **`0x400`**; initial read 4 KiB.
   - **QSPI NOR**: the pinned U-Boot tool uses IVT offset **`0x1000`**; RM 8.6.2-8.6.3 also requires a configuration block at `0x400` and distinguishes XIP/non-XIP. This is not the SD recipe with a changed write offset.
   - **Parallel NOR / EIM**: IVT at offset **`0x1000`** (4 KB).
   - **OneNAND**: IVT at **`0x100`**; initial read 1 KiB.
   - **Raw NAND** (non-OneNAND): handled via the **FCB (Firmware Configuration Block)**, the IVT does *not* live at a fixed offset. It is reached after the ROM parses the FCB. We do not cover raw-NAND boot in detail in this book.
6. **Validate the IVT.** Check its tag byte (`0xD1`), version, and self-pointer.
7. **Execute the DCD** if the IVT points to one. The DCD contains register operations that the ROM performs before loading the main image. It commonly configures clocks and DDR so the image can be loaded into DRAM.
8. **Load the required image bytes.** BootData describes the image base and total extent; the ROM applies the selected device's loading rules. This description is not a byte-for-byte trace of USB upload coverage.

9. **Apply HAB authentication policy.** Closed mode rejects unauthenticated images; open mode can report authentication failures without preventing execution. SDP follows authentication policy too.
10. **Jump to `IVT.entry`.** Control transfers to your image's entry point. From here on, your code controls the machine.

Notice how much happens before our first instruction. None of these ROM steps prints a progress banner to the debug UART. A message starts only when loaded code configures that UART and prints. The silent board from the preface could therefore be waiting for an image, rejecting its layout, or executing code that simply has no serial output. We need evidence more specific than silence.

BOOT_MODE[1:0]	BT_FUSE_SEL clear	BT_FUSE_SEL set
`00`, Boot From Fuses	Enter Serial Downloader	Boot using fuse configuration
`01`, Serial Downloader	Enter SDP	Enter SDP, subject to security policy
`10`, Internal Boot	Use GPIO boot-configuration override	Use fuse boot configuration
`11`	Reserved	Reserved

A board label such as "USB" is a convenient name for a switch pattern that selects SDP through these signals. Chapter 8 matches the reference schematic's switch numbering to that mode.

For SD internal boot, the ROM must obtain IVT, BootData, and any DCD **inside the initial 4 KiB** before it can perform the later loading work. Absolute final pointers do not make structures outside that first read available early. Our IVT starts at `0x400` and BootData ends at exclusive offset `0x42C`, below `0x1000`. A later DCD must also fit the initial region and its separate size limit [RM pp. 288, 309-312]. USB SDP acquires bytes through its protocol instead; do not confuse this media-read limit with a USB packet size.

7.3 The IVT, Image Vector Table

The first description is small: eight 32-bit words, **32 bytes** in total. The **Image Vector Table**, or IVT, identifies the header and points to the entry and associated structures. Despite the name, it is not the CPU's exception vector table. Read it as the ROM's description of this image:

Offset	Field	Description
+0x00	header	4 bytes: 0xD1 tag, 0x00 0x20 big-endian length, and 0x40 or 0x41 version
+0x04	entry	Absolute address the ROM jumps to after image is loaded.
+0x08	reserved1	Must be 0x00000000.
+0x0C	dcd	Absolute address of the DCD, or 0 if none.
+0x10	boot_data	Absolute address of the BootData structure.
+0x14	self	The IVT's intended absolute address; pointer differences locate accompanying structures.
+0x18	csf	Address of the Command Sequence File (HAB signatures), or 0 if unsigned.
+0x1C	reserved2	Must be 0x00000000.

Read the fields in two groups. `header` identifies the structure and its format. The pointer fields describe where the image's pieces will be in memory:

- **The `header` byte sequence `0xD1 0x00 0x20 0x40`** is the signature the ROM looks for. If you write the wrong byte order at offset 0, the ROM rejects the image with no diagnostic. You will see this byte pattern at offset `0x400` of every bootable SD card in this book.
- **`self` is required.** The ROM can initially read a header into temporary storage. A structure's relative position is found through its absolute-pointer difference from `self`. For this file, `structure_file_offset = 0x400 + (structure_pointer - self)`.
- **IVT pointers are absolute physical addresses**, not file offsets. Final `entry` must agree with linked code at the final destination. This header handling does not promise relocation of arbitrary machine code or repair of its embedded references.

A worked example

Our code is already linked at `0x00908000`. We need to add a header without putting it where the first instruction belongs. Leave room before the code and follow the same bytes in two views: their positions in the stored file, and their addresses after loading into OCRAM.

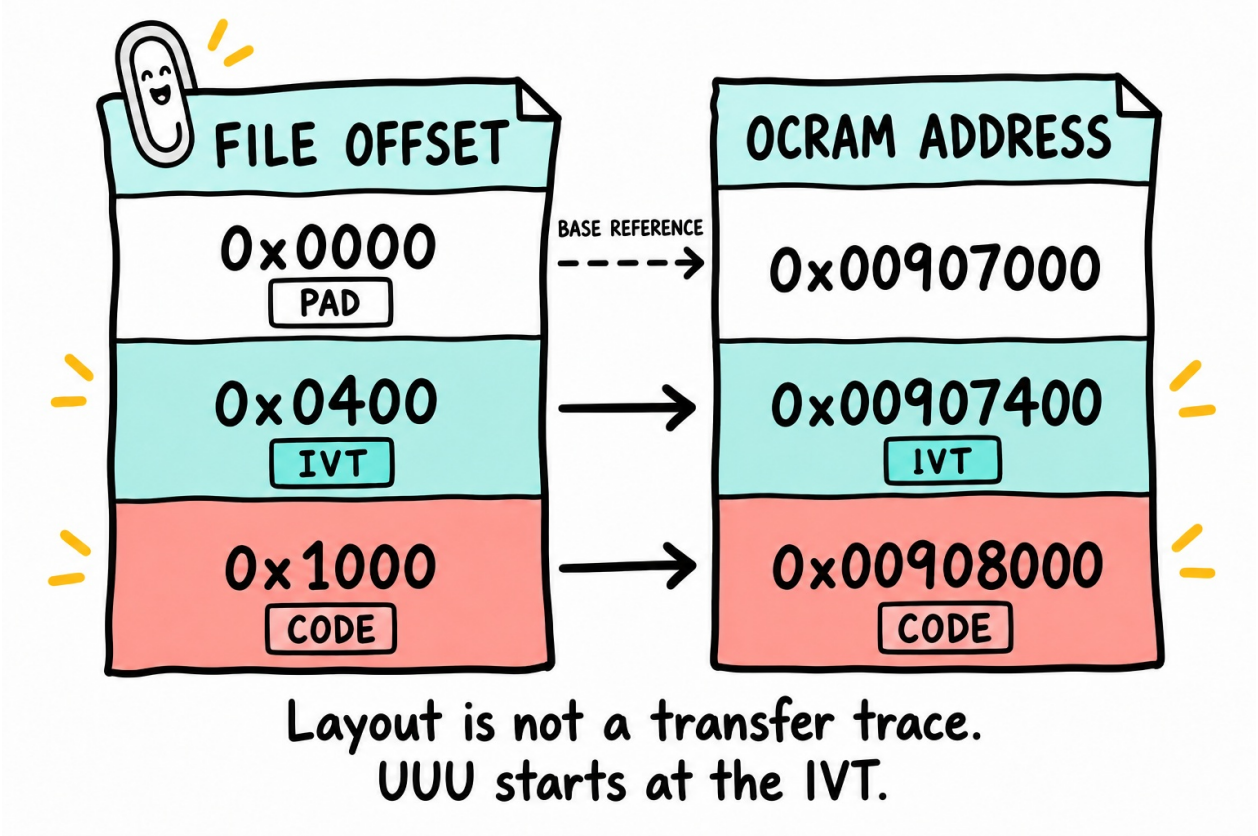
First, the file layout for this small image:

```
SD-card image file
  offset 0x0000..0x03FF   padding / unused by the ROM
  offset 0x0400           IVT begins here
  offset 0x0420           BootData begins here
  offset 0x1000           our code begins here
```

Choose `0x00907000` as the RAM coordinate for file offset zero. Adding each file offset gives its intended RAM address. Before reading on, calculate the IVT address and the BootData address. They should be `0x20` bytes apart.

Now check those values against the ROM's work:

1. The ROM reads from SD-card offset `0x400` and checks for the IVT header bytes `D1 00 20 40`.
2. The IVT says `self = 0x00907400`, so the ROM knows the IVT is meant to live at OCRAM address `0x00907400`.
3. The IVT points to BootData at `0x00907420`. That is `0x00907400 + 0x20`: the IVT starts at `0x00907400` and is 32 bytes (`0x20`) long, so BootData sits immediately after it.
4. If the IVT has a DCD pointer, the ROM runs those register writes. For this small OCRAM-only example, we use `dcd = 0`, so there is no DCD work.
5. BootData says `start = 0x00907000` and `length = total file size`. The `self - start` difference is `0x400`, matching the IVT's offset in the file.
6. The ROM jumps to the IVT's `entry` address, `0x00908000`.



Each row pairs a file offset with its intended RAM address. `base + file_offset` describes the layout, not the bytes actually transferred. In our later UUU path, the upload starts at the IVT and skips the leading `0x400` padding bytes. For uploaded bytes, `RAM = self + (file_offset - 0x400)`. The rows are not to scale. These arrows are not MMU translations.

Here is the same correspondence, including BootData:

File view	Intended RAM coordinate
SD offset `0x0000`	OCRAM address `0x00907000`
SD offset `0x0400`	OCRAM address `0x00907400`
SD offset `0x0420`	OCRAM address `0x00907420`
SD offset `0x1000`	OCRAM address `0x00908000`

In OCRAM at address `0x00907400`, the IVT contains:

```
Offset    Field      Value
+0x00     header     0xD1 0x00 0x20 0x40
+0x04     entry      0x00908000
+0x08     reserved1  0x00000000
```

```
+0x0C     dcd         0x00000000   (no DCD for this tiny OCRAM-only image)
+0x10     boot_data   0x00907420   (BootData starts immediately after IVT)
+0x14     self        0x00907400   ← the address of this IVT
+0x18     csf         0x00000000   (no signature)
+0x1C     reserved2   0x00000000
```

The linker placed `_start` at `0x00908000`; the IVT's entry field gives that address to the ROM. Chapter 9 supplies the first wrapper with this no-DCD layout, and Chapter 11 generalizes it. The whole image must still fit in Chapter 5's ROM-active OCRAM window.

7.4 BootData, telling the ROM how big the image is

The IVT gives the ROM a pointer to **BootData**. This next structure answers two practical questions: where should the image be loaded, and how many bytes belong to it? It occupies **12 bytes**:

Offset	Field	Description
`+0x00`	`start`	Physical address to load the image to.
`+0x04`	`length`	Number of bytes to load (including the IVT, DCD, and padding).
`+0x08`	`plugin`	0 = normal image. A nonzero value selects a plugin image. We will not use plugins.

In our padded file, `start` is the RAM address corresponding to file offset zero. `length` covers the complete file, including its padding and headers, not just `.text`. If you count only the instructions, you can leave later initialized data out of the transfer. A different image generator may use a different file layout, so derive these values from that layout rather than the extension.

7.5 The DCD, Device Configuration Data

OCRAM let our small image avoid a circular problem: code destined for DDR cannot first rely on unconfigured DDR to run the code that prepares it. For a larger image, someone must initialize the controller before that transfer. Who can do the work while the main program is not yet loaded?

Device Configuration Data, or DCD, is one answer. It contains commands the ROM performs before loading the main image, commonly register writes for clocks and DDR initialization. These are data records interpreted by the ROM, not Arm instructions compiled from our application.

Some commonly used command tags are:

Opcode	Name	Description
`0xCC`	WRITE	Write value(s) to register(s)
`0xCF`	CHECK	Poll register until condition is met
`0xC0`	NOP	No-op

The reference manual defines additional commands. WRITE and CHECK are the commands most relevant to the examples in this book.

WRITE format

```
0xCC <length:2-bytes-BE> <flags:1-byte> <addr0:4> <val0:4> <addr1:4> <val1:4> ...
```

The parameter uses width in **bits 2:0** as a byte count; **bit 3** is Mask and **bit 4** is Set [RM Tables 8-29/8-30]. All address/value pairs in one command use that parameter:

```
bits:        7 6 5 | 4   | 3    | 2 1 0
             0 0 0 | Set | Mask | width
0x04:        0 0 0 | 0   | 0    | 1 0 0   plain 32-bit write
0x0C:        0 0 0 | 0   | 1    | 1 0 0   clear selected word bits
0x1C:        0 0 0 | 1   | 1    | 1 0 0   set selected word bits
```

Addresses and values in these command records are big-endian, unlike the IVT pointer words. DCD is not arbitrary reset-time MMIO: each address must be in the SoC's permitted ranges [RM Table 8-31], aligned for the selected width, with a fitting value/mask. Invalid parameters can reject an entire multiwrite command. A familiar GPIO or watchdog write is not automatically permitted just because it works in later bare-metal code.

CHECK format

```
0xCF <length:2-bytes-BE> <flags:1-byte> <addr:4> <mask:4> [count:4-bytes-BE]
```

CHECK reads `addr`, applies `mask`, and tests the selected condition. Its flags do not mean the same operation as WRITE's set/clear flags. For a 32-bit word [RM Tables 8-32 through 8-34]:

Parameter	Condition on the selected bits
`0x04`	All clear
`0x14`	All set
`0x0C`	At least one clear
`0x1C`	At least one set

Without `count`, this 12-byte command polls indefinitely. With the optional word it is 16 bytes and polling is bounded; count zero performs no polling. A count is not a documented duration in milliseconds. For example, a PLL-lock wait needs the correct bitmask, condition, and permitted register, not merely a CHECK tag copied after a write.

A minimal DCD

Format-only, not a runnable DDR setup: this byte record would write `0x12345678` to `0x80000000`. Do not submit it on a freshly reset board: DDR at that address has not been initialized. The first OCRAM image uses no DCD.

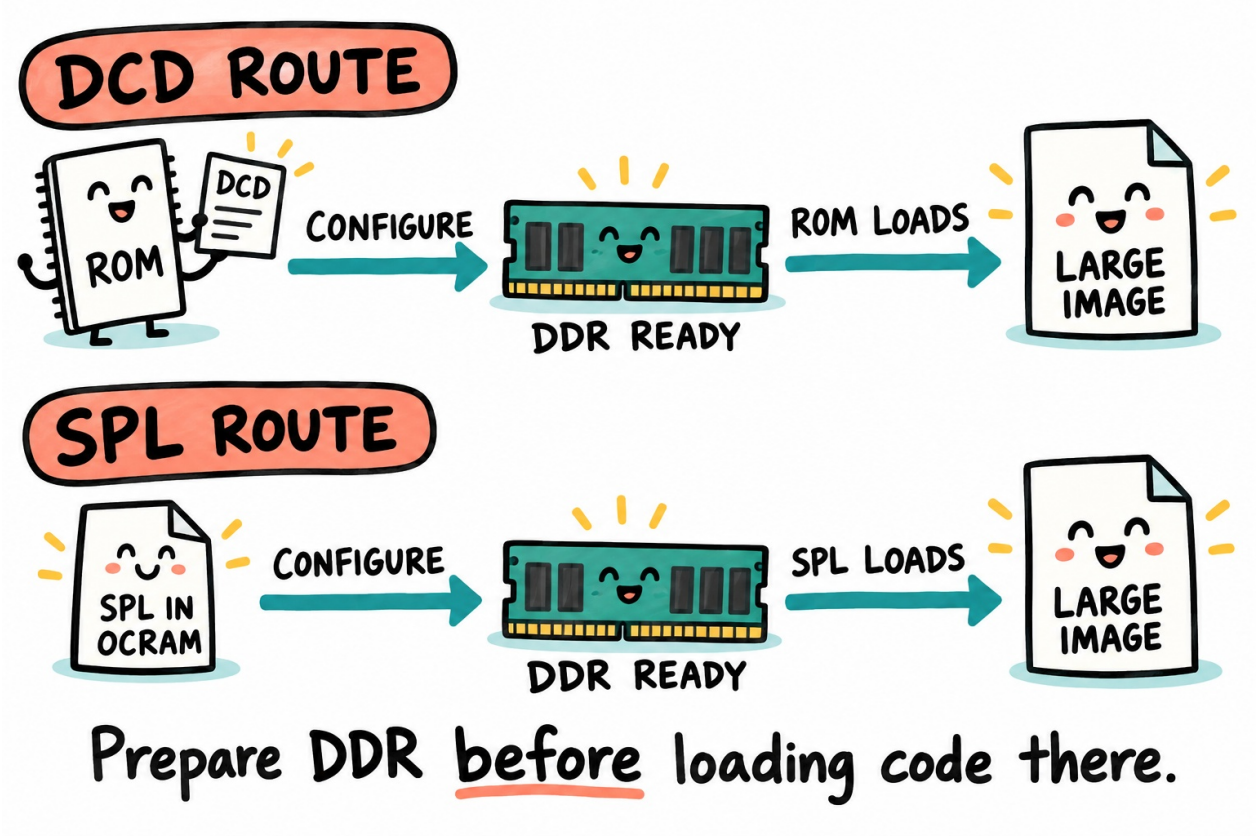
```
0xD2 0x00 0x10 0x41      # RM-format DCD header, 16 bytes total
0xCC 0x00 0x0C 0x04      # WRITE command, 12 bytes, 32-bit width
0x80 0x00 0x00 0x00      # address = 0x80000000
0x12 0x34 0x56 0x78      # value
```

This DCD is 16 bytes total: a 4-byte DCD header followed by one 12-byte WRITE command. A DDR initialization DCD is often several hundred bytes because it contains many register writes.

This format-only record follows the RM's documented version `0x41`. U-Boot v2024.01's image constants use `0x40`; the exact tool/silicon combination must be validated before turning a DCD into a runnable setup. Our first image uses no DCD and does not depend on resolving that compatibility question. The documented maximum DCD size is 1,768 bytes.

Why DCD exists

There are two ways out of that ordering problem. Give the ROM a DCD so it can prepare DDR before transferring the main image there. Or give it a small **Secondary Program Loader (SPL)** that fits in OCRAM; the SPL initializes DDR and loads the larger stage itself. The register work can be similar, but who performs it and when are different.



DDR must be ready before the larger program is loaded there. In the DCD route, the ROM reads configuration records. In the SPL route, a small program running in OCRAM does the work. The two rows show alternatives. Our first OCRAM-only image needs neither a DCD nor an SPL.

We set the first image's DCD pointer to zero. Later code configures DDR when we reach that experiment; Chapter 11 keeps the OCRAM/no-DCD design. Do not copy a DDR DCD from an unrelated board as a compulsory header block.

7.6 Boot modes, in concrete detail

The header cannot help if the ROM is looking on a different device. The boot-mode signals answer that earlier question: where should it look? Keep the decision table in Section 7.2 beside the two routes below, and remember that a board's switch label represents a particular signal pattern.

Internal Boot (BOOT_MODE = 10)

The ROM reads `BOOT_CFG1[7:0]`, `BOOT_CFG2[7:0]`, `BOOT_CFG4[7:0]` from the boot-mode pins (or from fuses if `BT_FUSE_SEL` is set). The bit patterns encode:

Device-class field	Meaning	Additional configuration
`BOOT_CFG1[7:6] = 01`	SD/MMC class	Controller, width, speed/protocol and other fields in RM SD/MMC tables
`BOOT_CFG1[7] = 1`	NAND class	Geometry and NAND-specific boot configuration
`BOOT_CFG1[7:4] = 0011`	Serial ROM through ECSPI	Port/chip-select/addressing fields, RM Table 8-20
`BOOT_CFG1[7:4] = 0001`	QuadSPI class	Interface and other fields, RM Table 8-22; interface bit 3 value 1 is reserved

These entries identify device classes. They are not complete board settings: controller selection, storage details, and board wiring supply other bits. Chapter 8 decodes the reference selector, so do not try to assemble a switch pattern from this class table alone or change boot fuses for these examples.

The Point Atom MINI boot selector exposes four labeled modes: **SD**, **eMMC**, **NAND**, and **USB**. SD, eMMC, and NAND are internal-boot configurations. USB selects Serial Downloader mode. The fitted storage depends on the core-board variant.

Serial Downloader (BOOT_MODE = 01)

The i.MX6ULL USB downloader identity is VID `0x15A2`, PID `0x0080`, corroborated by NXP's UUU device table. The RM also describes SDP through UART1/UART2. Keep serial receive activity quiet during USB startup: a UART data-ready event, including noise, can select the UART downloader [RM 8.9.2]. We use USB for the early downloads and the UART later for our program's text.

In SDP mode the ROM accepts a small command set:

- `0x0101` READ_REGISTER
- `0x0202` WRITE_REGISTER
- `0x0404` WRITE_FILE, push bytes to a target address
- `0x0505` ERROR_STATUS
- `0x0A0A` DCD_WRITE
- `0x0B0B` JUMP_ADDRESS, process a previously-loaded image's IVT

The identifiers come from RM Table 8-42. You will not need to type them or implement their USB exchange; `uuu` does that work. For our image, it uploads the IVT and following bytes, then asks the ROM to process the IVT. The header still supplies the final branch destination through `IVT.entry`.

USB mode provides the recovery path used throughout this book. If the Boot ROM enters SDP and accepts commands, software can still be loaded without relying on the installed flash contents.

7.7 The .imx image format

The name `.imx` tells us the file is intended as an i.MX boot artifact, but not which padding convention its generator used. Our **padded raw image** has the no-DCD layout below and is written at card byte zero:

```
File offset    Content
0x0000         1 KB padding; no partition table in the tiny Part II image
0x0400         ┌─ IVT (32 bytes)
0x0420         │  BootData (12 bytes)
0x042C         │  padding (a later DCD layout must define its own location)
0x1000         │  Application image proper (.text, .rodata, .data)
   ...         │
               └─  end after BootData.length bytes
```

The layout must satisfy both early acquisition and final placement: applicable IVT/BootData/DCD structures fit the device's initial-read region; `self` describes the IVT destination; `entry` matches the linked code; and `length` covers the image's complete defined extent. Final pointer arithmetic alone is not sufficient.

Here is an easy mistake to catch on paper. Our padded file already puts the IVT at file offset `0x400`. Writing that file at card offset `0x400` would put the IVT at `0x800`, not where the ROM expects it for this SD layout. Another generator may put the IVT first and require the `0x400` card write offset. Both conventions can be correct; mixing them is not:

Artifact	IVT file offset	Card write offset	Resulting IVT card offset
Book's padded raw image	`0x400`	`0`	`0x400`
Conventional IVT-first `mkimage` boot blob	`0`	`0x400`	`0x400`

Check the actual generator/version before writing. The small Part II card has no partitions. When a later Linux image adds partitions, reserve the boot region explicitly; ordinary GPT puts its primary partition-entry array at LBA 2, conflicting with an IVT at byte `0x400`. A later-starting MBR partition layout is a different contract. See UEFI GPT layout.

Tools that can generate IMX boot artifacts:

- `mkimage -T imximage -n image.cfg -d app.bin app.imx`: syntax illustration only, not a runnable alternative here. It needs a supplied configuration, entry/load settings, and a placement check. See U-Boot v2024.01 imximage.c.
- **Our own wrapper in Chapter 11.** A small Python program constructs the header fields and combines them with the application bytes, without using `mkimage`.

By writing the small wrapper ourselves, we can inspect exactly how its fields were chosen. When we later use `mkimage` or U-Boot's image configuration, the tool will be automating a format we have already followed by hand.

7.8 HAB, High Assurance Boot, briefly

A correctly laid-out image tells the ROM where its pieces belong; it does not establish whether they are trusted. A production device may also require an answer to "who signed this?" **HAB**, High Assurance Boot, is NXP's ROM secure-boot scheme. The IVT's `csf` field points to its **Command Sequence File** authentication data, produced with the Code Signing Tool, **CST**. HAB uses a fuse-stored **Super Root Key** (SRK) hash, certificates, and signatures to apply the provisioned trust policy.

Open-mode authentication failures do not necessarily block execution. Closed-mode policy requires valid authentication; closing the device is irreversible. An incorrectly signed image can sometimes be replaced through an authenticated recovery path using the provisioned trust configuration. Lost signing keys or unusable trust provisioning are different, potentially unrecoverable failures. SDP does not bypass closed-mode authentication.

We will not enable HAB in Parts I-VI of this book. Chapter 124 covers the full HAB workflow, including how to sign U-Boot and the kernel and how to extend verification to the root filesystem.

For the unsigned teaching images here, leave `IVT.csf = 0`. Do not touch SEC_CONFIG fuses. Image layout and image trust are separate questions; we only need to solve the first one for these early experiments.

7.9 How to inspect an .imx image

After Chapter 11 creates an `.imx` file, inspect it with:

```
$ xxd -s 0x400 -l 32 app.imx
```

For the worked example in Section 7.3, the 32 IVT bytes at file offset `0x400` are:

```
d1 00 20 40  00 80 90 00  00 00 00 00  00 00 00 00
20 74 90 00  00 74 90 00  00 00 00 00  00 00 00 00
```

The header stores its length bytes in big-endian order, as required by the IVT format. The pointer fields are 32-bit little-endian values:

Bytes	Field	Decoded value
`d1 00 20 40`	header	tag `0xD1`, length 32, version `0x40`
`00 80 90 00`	entry	`0x00908000`
`00 00 00 00`	reserved1	0
`00 00 00 00`	dcd	no DCD
`20 74 90 00`	boot_data	`0x00907420`
`00 74 90 00`	self	`0x00907400`
`00 00 00 00`	csf	unsigned image
`00 00 00 00`	reserved2	0

Pick the entry field out of the dump: `00 80 90 00` decodes to the `0x00908000` address we chose earlier. The layout has become actual bytes we can inspect. Compare Chapter 11’s output field by field, remembering that the header length and the pointer words use different byte-order rules. A file appearing in the directory is only the beginning of that check.

Do not use `dumpimage -l` as a promised validator for this custom padded/no-DCD file: tool support depends on header placement and format. Inspect these exact bytes and pointer relationships first; successful structural checks still do not prove execution.

7.10 Key Notes

- The Boot ROM is a loader with a fixed contract. The IVT names the entry point and associated structures, BootData describes the loaded extent, and DCD supplies initialization before application code can run.
- File offsets, media offsets and physical RAM addresses are different coordinate systems. Pointer differences must agree with the image layout, but a valid-looking header does not prove that the board’s DDR settings are correct.
- The worked OCRAM image has no DCD because its first instructions do not depend on external memory. A DDR-based image needs a documented initialization path, whether that responsibility belongs to ROM-executed DCD or to SPL.
- Boot-mode pins select a ROM path at the documented reset boundary. Switch markings alone do not reveal the sampled bit values, and a warm reset is not a substitute for the specified cold-start procedure.
- HAB changes which images the ROM accepts. Image inspection is reversible, but closing security fuses is not an ordinary configuration step.

7.11 Pitfalls

- **`IVT.self` confused with a file offset.** An inconsistent self-pointer can break the ROM’s interpretation of the associated structures. Check `self` against the IVT’s intended physical address after loading, and check the pointer differences against the file layout. A silent board does not tell you which of those checks failed.
- **`BootData.length` shorter than the image.** Tail of your image is not loaded. `.data` initial values become whatever was in RAM.
- **Unbounded DCD CHECK.** Without a poll count, a condition that never changes can block progress indefinitely. A bounded count does not by itself prove valid hardware initialization. To recover, disconnect boot-power paths, select USB while unpowered, and use Chapter 8’s documented cold-start sequence before loading a corrected image. Security policy still applies.
- **Wrong DCD lengths or byte order.** Header/command lengths are big-endian. A malformed record can fail parsing/validation or misidentify the following records, preventing boot. Decode its lengths and boundaries before transfer; DCD is not unchecked executable code.
- **Mixing file and media offsets.** The book’s padded file goes at card byte zero; an IVT-first boot blob uses another write offset. Generic partitioning, especially GPT, can overwrite the boot region. Recheck the exact layout before writing.
- **Closing HAB by accident.** It is a one-way fuse. Do not write to OCOTP_CFG5 unless you have read Chapter 124 carefully and have a key-management plan.

7.12 Going deeper

- **IMX6ULLRM**, *Chapter 8, System Boot*. The authoritative reference.
- For NAND and QSPI, start with the device-specific sections of IMX6ULLRM Chapter 8 and verify any application note's actual title and scope.
- **AN4581**: *i.MX 6 Series Boot Process*.
- U-Boot `tools/mkimage.c` and `tools/imximage.c`, which implement the production image-generation path.
- U-Boot v2024.01 image constants, for this tool's media offsets and header encodings. Do not substitute the separate later-family `imx-mkimage` container workflow for this i.MX6ULL route.

We have followed an instruction from its linked address to the header field that names it, and followed that header back to its position in the file. The numbers agree on paper. Next we need a physical connection to the chip that will read them. The Boot ROM can expose its USB downloader before our program runs; Chapter 8 checks that path on the actual board.

Chapter 8: Hardware bring-up checklist

Put the board on the desk with every cable still disconnected. After several chapters of addresses and registers, we can look at the actual connectors and switch markings those descriptions depend on. The first question is practical: how should this particular revision be connected and powered?

Then we can look for a result that does not require our own program. Chapter 7’s Boot ROM is already in the chip. In the appropriate USB mode, the host should be able to see its downloader interface even though we have not built a bootable SD image. That observation will give us a known connection to return to when a later download fails.

8.1 What we can and cannot prove yet

We have the Ubuntu tools from Chapter 3 and a Point Atom MINI board. The tempting test is to connect everything and wait for a Linux prompt. That would mix too many unanswered questions together: the supply, cables, boot mode, storage image, and software. Separate them first.

Without a program of our own, we can check that:

- The board has no obvious physical damage.
- An unpowered resistance screen shows no suspected short; this is not proof that every rail is healthy.
- The built-in USB-TTL serial bridge appears on the host.
- The requested USB switch pattern agrees with the matching schematic.
- The i.MX6ULL Boot ROM enumerates as a USB SDP device.
- You can identify the SD-card device on the host without guessing.

Each item answers a smaller question than “does the system boot?” A running Linux system, a successful image transfer, and an LED responding to our code are later tests. Keeping them separate lets us recognize real progress without mistaking a working serial bridge for a working target program.

8.2 Unbox and inspect

Set the unpowered board on an anti-static mat and look over both sides. This quiet inspection saves awkward work later: a cable can cover the switch numbering, and a second USB connector can look much like the first. Find the labels while they are still easy to see.

1. **Visible damage.** Look at every connector. Are any pins bent? Are any solder joints cracked or incomplete? Are any capacitors discolored? Did any screw hole damage a trace? Reject and return the board if you find serious damage.

2. **Connectors.** Locate USB-OTG, the USB-TTL debug port, Ethernet, microSD, the reference MINI's 48-pin/2x24 P4 expansion header, LCD connector, JTAG, and power input. A similar-looking header on another board need not have a compatible pinout.

3. **Boot-mode selector.** Match your PCB revision to the MINI v2.2 schematic. Record switch numbering/ON direction and compare the patterns in Section 8.5. Do not infer positions from mode names alone.

Photograph the top and bottom, including the printed PCB revision and switch numbering. As you identify the boot settings, photograph those too. Later, when the board is under cables, a clear photo is often quicker to consult than turning the whole assembly over.

8.3 Power rails: unpowered checks, then powered measurements

Choose the power arrangement before any cable

There is an important difference between "this board has a 5 V rail" and "this input accepts my 5 V supply." The MINI v2.2 power sheet (sheet 3, PDF page 4) shows a DC converter, `USB_TTL` VBUS (`VUSB`), OTG VBUS circuitry, K1, **JP1/VCC3.3**, and **JP2/VCC5**. JP3 belongs to the RS485 signal routing, not this pair of rail headers. `DCDC_5V` names a rail, not the barrel-input rating. Do not plug in a guessed 5 V/12 V supply.

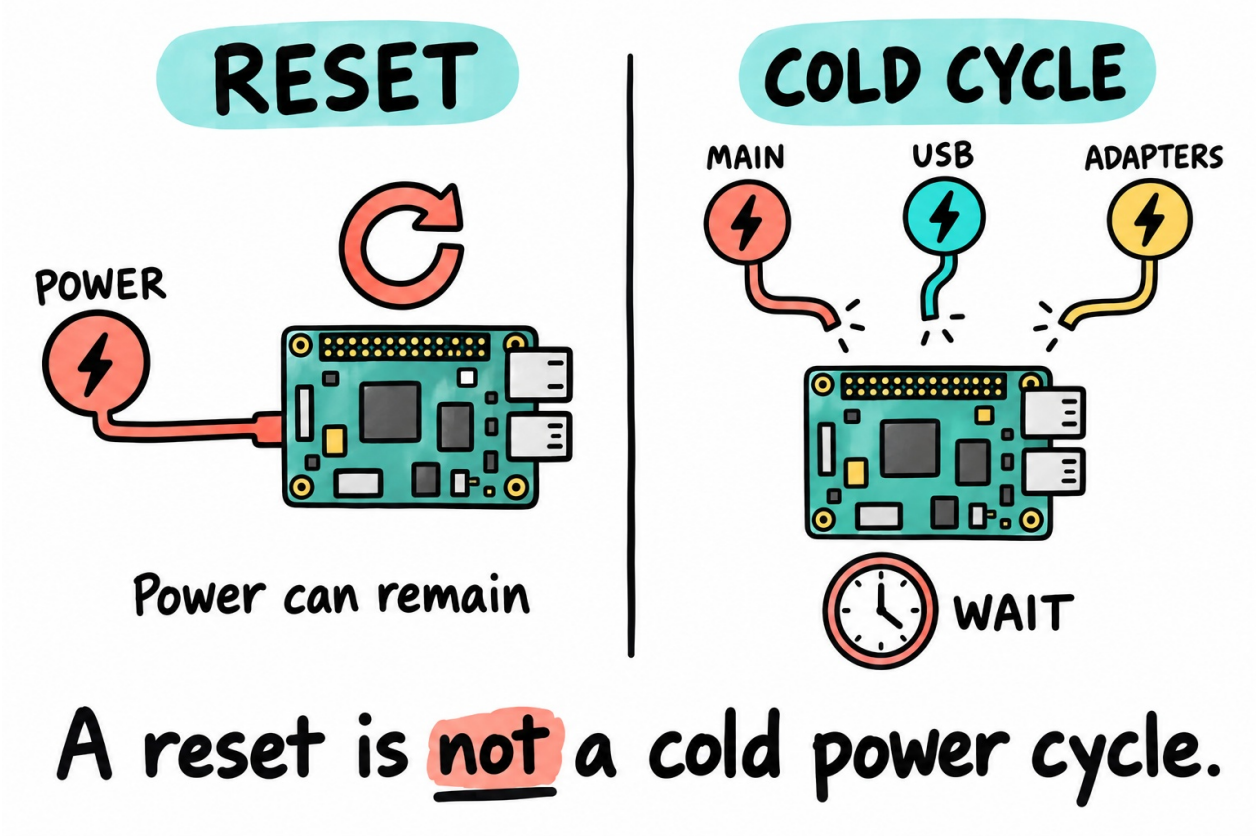
Use the supplier's arrangement for your **exact baseboard and core revision**. Record the input connector, rated voltage/current/polarity, K1 position, and any jumper settings before applying power. Attach the two USB data cables only if that arrangement documents their VBUS paths. If you have only the schematic and cannot establish those details, stop and obtain the matching board power guide; the book does not provide an electrically validated universal jumper recipe.

Connection	Role	Power check before attachment
Supplier-approved supply input	Main board supply in the selected arrangement	Rating, polarity, selector/jumpers match the guide
USB-TTL / DEBUG USB	Host serial bridge	Cable carries VBUS; determine whether VUSB can feed the board in the selected K1 state
USB-OTG	ROM SDP data link	Cable also carries VBUS; check the documented OTG isolation/ power path
External TTL adapter, if used	UART data only	Correct I/O voltage; VCC disconnected; no powered output driving an unpowered I/O domain

A USB cable deserves the same attention as a supply cable: it carries power as well as data. The board’s power switch may therefore leave the serial bridge or other circuitry partly powered. When we say **cold power cycle**, we mean disconnecting the main input and every USB/power-capable adapter, allowing the rails to discharge, then reconnecting in the documented order. Never move power jumpers on a powered board. Simultaneous-input behavior has not been measured here, so the matching board guide remains essential.

Inventory everything attached. For this first check, leave USB_HOST, expansion/LCD/ camera/SDIO add-ons, powered probes, and rail-header connections unused. They can introduce supply paths omitted from a simple two-USB picture. A directly connected adapter’s TX/RX can back-power an unpowered target even with VCC disconnected; remove its signal connection unless the exact interface documents safe isolation.

Here a cold cycle means cycling the **main boot-power domains**. A coin cell may retain SNVS/RTC power if the matching design permits it; that is different from powering every domain down. Do not remove a battery or short rails to discharge them without the documented sequence. If a retained source can feed the circuit being resistance-tested, establish safe isolation first or stop; a partly powered board is not an unpowered resistance test.



Reset need not remove power. The clock means waiting for the relevant main rails to discharge, not a requirement to erase the backup domain. Follow the board guide's source inventory and sequence. This generic drawing is not a connector layout or permission to attach several supplies together.

With the power arrangement recorded, make an unpowered resistance check before connecting any cable:

1. Disconnect all supplies, USB cables, and adapters; allow capacitors to discharge. Use known labeled test points, not guessed adjacent pins.
2. Select **resistance** mode and measure 3V3-to-GND, then 5V-to-GND where accessible. Record settled values. A brief continuity beep can be capacitors charging; the beep threshold is meter-dependent.
3. A persistent near-zero reading suggests a short: stop and investigate. No normal-resistance threshold is specified here without measurements for this revision. A higher reading alone does not certify every rail.

Lift both probes after the resistance check. **While the board is still disconnected**, put the black lead in COM and the red lead in the voltage/resistance jack, not A/mA, then select an appropriate **DC-voltage** range. Only then connect the documented supply arrangement. Attach the black probe to known GND and secure it so neither probe can slip across nearby pins. Never measure resistance on a powered circuit.

Now measure:

1. Probe **3V3** to **GND**. Expected: about 3.30 V.

2. Probe **5V** to **GND** if accessible. Expected: about 5 V.

A lit power LED is an observation, not a measurement of every rail. Record the voltages so a later reset or inconsistent result has something to compare against. These accessible rails do not validate core/DDR sequencing or every supply. If a reading is unexpected, disconnect power before investigating.

8.4 Built-in USB-TTL serial console

The debug UART is the connection through which our programs will eventually send text to the host. The MINI already has a USB-to-TTL bridge wired for this purpose, so normal use does not require an external CP2102, CH340, FTDI, or a set of jumper wires.

After Section 8.3's power-path check, attach USB-TTL/DEBUG USB. It is separate from the USB-OTG port; check the silkscreen.

On Linux, check which serial device appeared:

```
$ sudo dmesg | tail -20
```

Look for a message identifying the newly attached serial device, for example:

```
ch341-uart converter now attached to ttyUSB0
```

Here the device name is `ttyUSB0`, so its path is `/dev/ttyUSB0`. Use the node from your actual attachment log, at **115200 8N1**: eight data bits, no parity, one stop bit, and no flow control.

```
$ sudo picocom -b 115200 /dev/ttyUSB0
```

If your host reports `/dev/ttyACM0`, use that instead:

```
$ sudo picocom -b 115200 /dev/ttyACM0
```

The terminal opens, but nothing is printed. Should we change the baud rate? Not yet: the Boot ROM prints no UART banner, and we have supplied no printing program. Opening the bridge confirms host access, not target UART configuration. **Do not type to wake it up.** For the USB-ROM check, quit locally with Ctrl-A then Ctrl-X; plain Ctrl-C can send a character to the board. After a USB unplug/replug, identify the bridge again before reopening it. Chapter 12 gives us our own UART output.

Your board may already contain a factory image in eMMC or NAND. If it prints in its storage mode, keep the output in your notes; there is no need to erase it for this check. If the bridge appears on Windows but not inside an Ubuntu VM, attach it to the guest and close any Windows-side serial session. As in Chapter 3, we use explicit `sudo` for restricted logs and devices instead of changing host groups.

Unreadable characters later

If you later see unreadable output after our code starts printing, the common causes are:

- Wrong baud rate. Use 115200 8N1.
- Opening the wrong `/dev/ttyUSBx` device.
- Bad USB cable or unstable power.

Return to these checks when a program that should print produces the wrong text. Before we have such a program, silence alone tells us little about the UART.

8.5 Boot-mode selector

Chapter 7 explained that the Boot ROM needs to know where to look for an image. The MINI's boot selector supplies that choice through the boot-configuration pins. Its four selections are:

Board mode	What the Boot ROM tries	When we use it
SD	Reads a boot image from the microSD card	Bare-metal examples and removable development images
eMMC	Reads a boot image from onboard eMMC	An installed system on an eMMC board
NAND	Reads a boot image from onboard raw NAND flash	An installed system on a NAND board
USB	Requests Serial Downloader mode / Serial Download Protocol (SDP)	Recovery and early bring-up

The board normally has either eMMC or NAND fitted, depending on the core-board version. A storage mode cannot boot if that device is not fitted or does not contain a valid image. Record all four switch patterns even if your core board does not contain both storage types.

For the **MINI v2.2 schematic's printed D1-D8 order**, its BOOT table gives:

Board selection	D1	D2	D3	D4	D5	D6	D7	D8
USB	OFF	ON	OFF	OFF	OFF	OFF	OFF	OFF
MicroSD	ON	OFF	OFF	OFF	OFF	OFF	ON	OFF
eMMC	ON	OFF	ON	OFF	OFF	ON	ON	OFF
NAND	ON	OFF	OFF	OFF	ON	OFF	OFF	OFF

Read the row using the switch numbers printed on the PCB, not left-to-right positions in a photograph. Match D1 to switch number 1 and use the printed ON mark; rotating the board does not reverse the numbering. D1/D2 route BOOT_MODE1/0, while the remaining switches configure storage-related straps. USB must result in SoC mode `01`; storage selections use Internal Boot with additional configuration.

The table is transcribed from the MINI v2.2 schematic, not hardware-tested for every MINI/ALPHA revision. Check your revision's routing and the core's fuse state rather than assuming that every board is unprovisioned and wired alike.

BOOT_MODE0/1 are sampled on **POR_B deassertion**, its rising edge [RM 8.2.1]. Keep the selector stable through the documented power-up/reset sequence; storage configuration has its own latching details [RM 51.6.3.1]. A software/SRC warm reset is not a board power cycle. A physical button's effect depends on its routing, so do not classify it solely from the word RESET. Change selectors with the main boot-power domains unpowered, following Section 8.3.

8.6 Confirm USB mode and SDP enumeration

The UART can be quiet while USB tells us something useful. The ROM's downloader interface does not wait for us to install a Linux image or configure our own UART. Follow the power checks and USB-mode sequence below, then look for its USB identity on the host.

Use a **Bash host shell**, not the terminal currently owned by picocom. In a newly opened Ubuntu terminal, run `. ~/imx6ull/scripts/env.sh` and require a successful status before using `$IMX6ULL_HOME`. Keep target UART receive quiet: no keystrokes, break signal, automated serial probing, or other sender during this ROM USB test. A UART data-ready event can select UART SDP [RM 8.9.2].

1. Disconnect all sources for a cold cycle as in Section 8.3.

2. Set the boot-mode switch to **USB** mode.
3. Verify the power-guide arrangement permits the USB-OTG cable’s VBUS connection.
4. Reconnect the documented power/USB arrangement with the switch already set.
5. On the host, run:

```
$ lsusb | grep 15a2
```

Expected output:

```
Bus 001 Device 008: ID 15a2:0080 Freescale Semiconductor, Inc. i.MX 6ULL in Serial
Downloader Mode
```

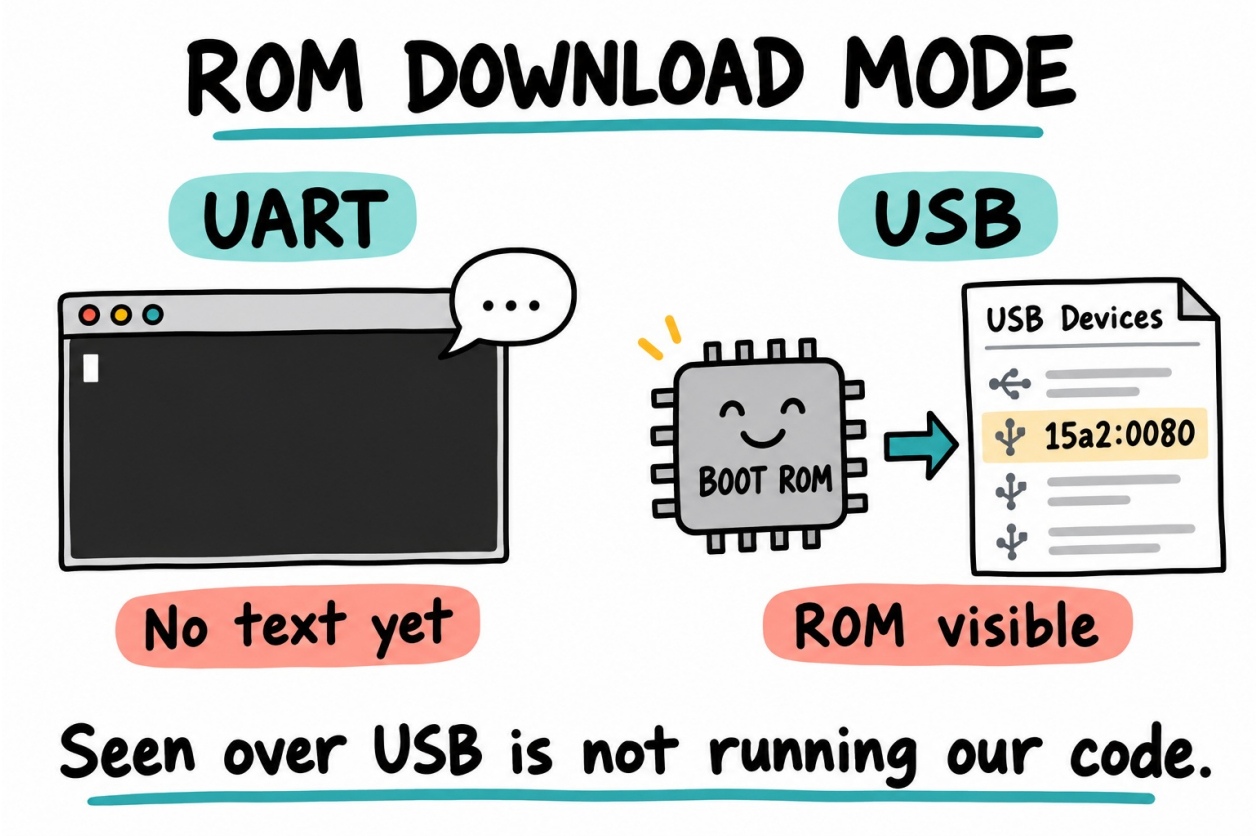
The bus and device numbers may change, and the descriptive text can vary. Look for **`15a2:0080`**, the vendor/product identity of the i.MX6ULL ROM downloader. It tells us that the host can see the ROM’s SDP interface.

It does not prove that the selector alone caused entry: fuse settings or fallback from a failed storage boot can also reach SDP. Record the requested switch pattern and observed identity separately. No fuse changes or unvalidated boot-register reads are needed here.

You can also ask `uuu` to list visible i.MX devices:

```
$ sudo "$IMX6ULL_HOME/build/mfgtools/uuu/uuu" -lsusb
```

If your actual output shows that identity, one of our questions has an answer: the host can see the ROM interface without a Linux image running on the board. Keep that output. It does not yet prove image transfer, authentication, or entry-point execution. Chapter 9 will add a different kind of evidence: the LED’s response to our program.



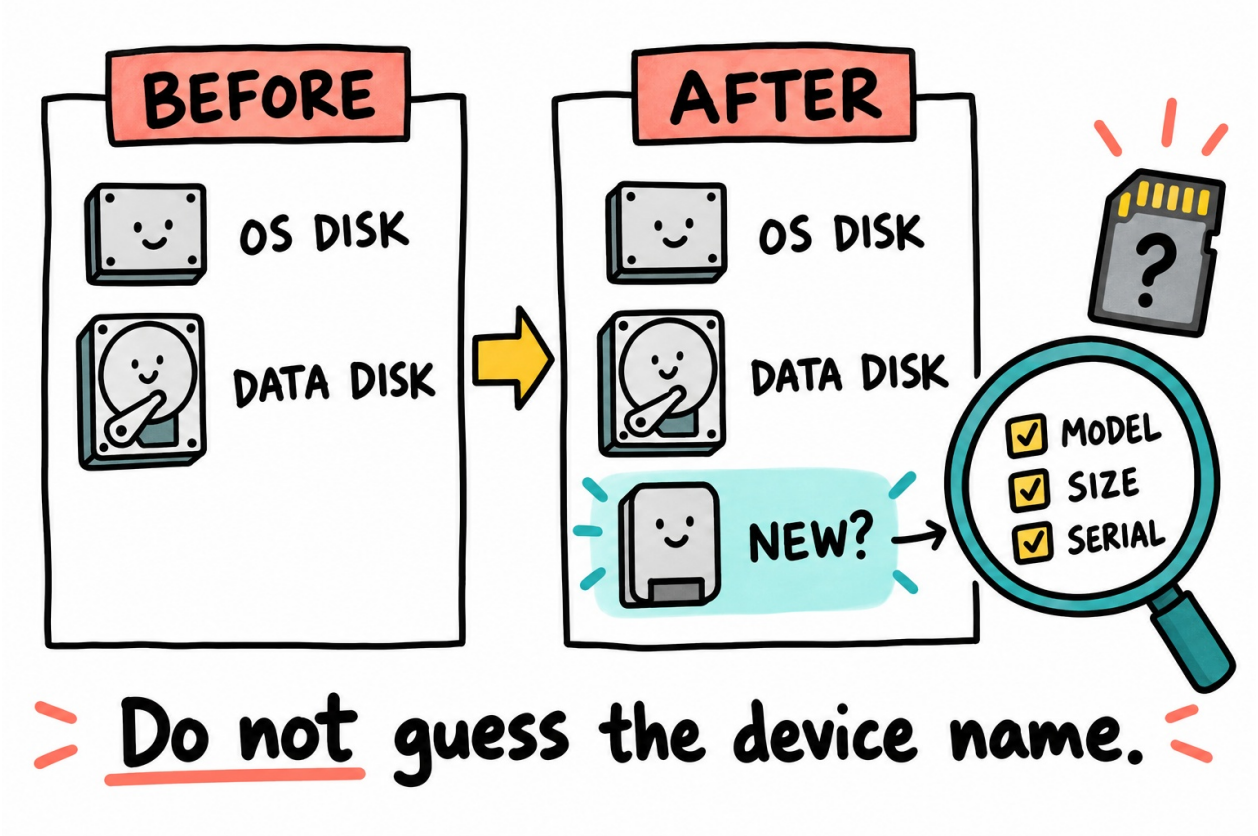
No text yet, but not no information. In the documented USB-mode cold-start state, `15a2:0080` tells us the host sees the ROM downloader. It does not prove a transfer, successful authentication, or execution of our image. These are two observations, not a cable or power-connection diagram.

If you do not see `15a2:0080`:

1. Confirm the boot switch is in the **USB** position.
2. Confirm you are using the USB-OTG port, not the USB-TTL debug port.
3. Try another USB data cable.
4. Return to the full documented cold-start sequence with USB already selected and UART input quiet; a reset-button press is not a substitute.
5. Check `sudo dmesg` for USB errors and, in a VM, confirm the ROM device is attached to Ubuntu rather than Windows.

8.7 Prepare the SD-card workflow

USB downloading will let us try the first program without preparing a boot card. In Chapter 11 we will use SD boot too. Before we reach the write command, learn how your host names the card and how to distinguish it from the host's own disks.



A new row is a question, not a write target. Model/serial fields may describe the USB reader rather than the inserted card. Match the physical spare card to the current whole-disk device using the connection, media/size changes, and before/after output. Missing or ambiguous identity is a reason to stop, not invent a serial number.

With the card not yet inserted, record a baseline; then insert the physical spare card, allow enumeration to finish, and repeat:

```
$ lsblk -o NAME,PATH,SIZE,MODEL,SERIAL,TRAN,RM,TYPE,MOUNTPOINTS
```

Read the `PATH`, `TYPE`, and `MOUNTPOINTS` columns in your actual output. For a card observed as `/dev/sdc`, the whole card is that `disk` row, not a `part` row such as `/dev/sdc1`. This is an example name, not a promised result.

Record the capacity, available identity fields, physical card, reader connection, and current device name. USB model/serial/by-id can identify the reader rather than uniquely identifying the medium; record unavailable fields as unavailable. An empty reader may remain listed, so insertion can change its size/media/partitions without creating a new disk row. If the first listing is incomplete, wait and repeat it. Device names can change; re-identify before every write.

An **illustrative** tree might look like this; these names are not destinations to copy:

```
NAME     PATH       SIZE TYPE MOUNTPOINTS
sda      /dev/sda   512G disk
  sda1   /dev/sda1  512G part /
```

```
sdc      /dev/sdc    16G disk
  sdc1   /dev/sdc1   16G part /media/you/CARD
```

Reject `sda`: its child contains the running system. `sdc` is only a candidate until its identity matches the physical spare card. `sdc1` is a partition, and its mount point must be cleared before removal or a raw write. Native cards can instead appear as `/dev/mmcblkN` with `/dev/mmcblkNp1` partitions. Neither a disk letter nor `RM=1` proves safety.

Do **not** run `dd` casually. A wrong device name can erase your host disk. Before every SD write in this book:

1. Take the before-insertion snapshot. If the known card is already inserted and must be removed for comparison, close its users and successfully unmount all its mounted partitions **before removal**. Stop on a busy/error result; do not force or lazily unmount it.
2. Insert the card, wait for discovery, and take the second `lsblk` snapshot.
3. Match the physical card and current media/size/identity clues; reject disks containing the host system, swap, or unrelated data.
4. Before a later raw write, close users, successfully unmount every mounted card partition, and recheck that all its mount points are empty. Revalidate its identity immediately before writing.
5. Write only a known image using its reviewed layout and a freshly identified medium. Stop on any write error. Successful `sync` is not proof that a failed write completed, nor a substitute for successful unmounting before removal.

Host enumeration and device identification are not an I/O, sustained-capacity, or write/readback test. Chapter 11 explains an actual boot image and its placement. Reading that explanation does not require writing a card.

8.8 What uuu will do later

`uuu` can list devices or talk to the Boot ROM over SDP to send an image. With the board in USB mode, an image transfer has this shape once a suitable `led.imx` exists:

```
$ sudo "$IMX6ULL_HOME/build/mfgtools/uuu/uuu" led.imx
```

For our early OCRAM image, the operation has two stages:

1. Sends an image file over USB-OTG into OCRAM.
2. Tells the Boot ROM to jump to the image entry point.

The two stages connect our earlier work. The bytes cross the USB connection checked here; the header from Chapter 7 tells the ROM how to handle the image. The code at the entry address must then do something observable. That last part belongs to the next experiment, not to seeing a USB device in the list.

You may encounter MfgTool in older board instructions. It is NXP's older Windows manufacturing GUI. We use `uuu` throughout this book, so there is only one downloader to set up and follow.

8.9 JTAG, optional but useful

If the LED does not blink, being able to stop the CPU and inspect its registers can save a great deal of guessing. That is what a JTAG debugger adds: hardware breakpoints, single-stepping, and register inspection. It is optional for Part II. The early examples use observable LED behavior, `uuu` and serial output to explain different levels of evidence.

Locate the JTAG header now. The important signals are:

- TMS
- TCK
- TDI
- TDO
- nTRST
- RESET
- GND
- 3V3 sense

When selecting an optional adapter, verify its exact model, voltage, target support, cable/pinout, and licensing:

- An OpenOCD-supported JTAG adapter may use FT2232H; an arbitrary FT2232H board is not automatically a compatible probe.
- J-Link models differ. EDU licensing is educational/non-profit, not a general work-use entitlement; check the exact SEGGER model and terms. A commercially licensed model still needs compatible target/voltage support.

We set up the debugger in JTAG/OpenOCD/GDB. The signals above are functions to verify, not a pin-ordered wiring list. Optional/reset components may be unpopulated. For now, locate the header only; voltage sense is normally a reference input, not permission to power the board from a probe.

8.10 Ethernet, only a physical check for now

The MINI’s Ethernet connector is easy to inspect now, even though IP networking will wait until U-Boot and Linux are running. This check is optional:

1. With the approved board power arrangement, connect the Ethernet cable to a powered host/switch link partner. Do not assume PoE powering is supported.
2. Confirm the connector fits firmly.
3. Observe whether the link LED lights.

A link indication is not proof of IP configuration, packet transfers, or running Linux. A dark LED is not yet a board-failure diagnosis: the PHY may need software setup. Actual networking tests come later.

8.11 What connection evidence establishes

Evidence	What it establishes	What remains separate
Base/core revision and matching guide	Which documented connections apply	Whether the fitted board matches those assumptions
Reviewed supply, selectors and USB power paths	A specific intended power arrangement	Electrical behavior under load and safe test limits
Resistance screening and accessible rail readings	Limited observations at stated points	Complete rail, reset and sequencing qualification
Identified serial bridge and opened port	A host serial channel	Target code execution or correctly configured UART output
Observed ROM VID:PID `15a2:0080` in USB mode	A reachable ROM interface	Image acceptance, execution or DDR health
Fresh removable-medium identification	Which host path currently belongs to the card	Capacity, durability and successful image readback

These are different evidence boundaries, not a completion worksheet. A physical transfer still requires its actual board, artifact and connection prerequisites. JTAG and Ethernet are optional to that first USB-loaded example.

8.12 Key Notes

- Bring-up evidence belongs to a particular board revision, fitted core and power arrangement. A matching product name is not enough to transfer another board’s connector, voltage or selector assumptions.
- USB-TTL provides a serial channel, while USB-OTG can expose the ROM downloader. ROM USB enumeration proves that connection in the selected boot mode, not that our image executes or that DDR is healthy.
- Power, reset, transport and software are separate failure boundaries. UART silence before a valid program runs is normal and does not identify which boundary failed.
- An unpowered short check is only screening. Powered rail measurements need documented test points, limits and a safe supply arrangement, including every possible USB power path.
- Removable storage identity must be established again before a write. A device name recorded earlier is not durable evidence that the same card still owns that host path.

8.13 Pitfalls

- **Confusing USB-TTL with USB-OTG.** USB-TTL is the serial console. USB-OTG is the Boot ROM recovery path. They are different functions and may be different connectors.
- **Expecting serial output too early.** With no valid image, silence is normal. The Boot ROM does not print progress messages on the debug UART.
- **Changing boot switches while powered.** Follow Section 8.3’s source inventory and cold-cycle procedure. BOOT_MODE pins are sampled at POR_B deassertion; an arbitrary warm reset is not a substitute.
- **Using a charge-only USB cable.** The board may power up but never enumerate. Use a data cable.
- **Writing to the wrong SD device.** Always compare `lsblk` before and after inserting the card.
- **Assuming a data cable cannot supply power.** USB-TTL and OTG carry VBUS. Use only the revision-specific documented combination; unplug all supply paths for a real cold cycle.
- **Assuming Ethernet is broken because no link LED appears.** Full Ethernet testing waits until software configures the PHY.

8.14 Going deeper

- The Point Atom MINI schematic. Print the pages for power, boot mode, USB-OTG, USB-TTL, and SD card.
- **IMX6ULLHDG**, *i.MX6ULL Hardware Development Guide*, in NXP's documentation catalog. It is a SoC guide, not the MINI barrel-supply specification.
- The `uuu` README at https://github.com/nxp-imx/mfgtools.
- An oscilloscope is optional for the checks here. Select probes/bandwidth for a later specific signal measurement; a generic scope rating does not certify DDR or every interface.

Chapter 9 follows a small assembly program from source to a visible LED change. The interesting connection is between compiler output, the ROM image header, the CPU's entry conditions and a GPIO owner. The board can demonstrate that connection, but the design can be followed without running the example.

Chapter 9: First LED, pure assembly

The host can already see the board in USB downloader mode. Now we want one observation that comes from our own program: the user LED changing state. No Linux console is involved. The ROM loads a small assembly payload into OCRAM, then gives it control.

An LED seems like a modest reward for all that preparation. It is also a useful first witness. To blink, the program must reach its entry point, access a GPIO controller and route that controller to the right pad. We can inspect each of those steps without first building a C runtime or initializing DDR.

Keep the Chapter 8 power and connector checks in place for any board operation. This example assumes an open development device and a fresh ROM downloader boot, not a jump from U-Boot or another running program. We do not change security fuses. The listings explain that execution path without requiring a board run to continue reading.

9.1 What we are about to build

A program with the following structure:

```
_start:                          ; ROM jumps here
    set SP to top of OCRAM
    enable clock to GPIO1        ; one write to CCM_CCGR1
    set pin GPIO1_IO03 to ALT5    ; one write to IOMUXC
    configure pad properties     ; one write to IOMUXC
    preload LED-off level        ; set bit 3 in GPIO1_DR
    set GPIO1_IO03 as output     ; one write to GPIO1_GDIR
loop:
    toggle GPIO1_IO03            ; toggle bit 3 of GPIO1_DR
    delay  (busy loop)
    branch loop
```

There is no scheduler and no hidden GPIO initialization. The listing below contains the instructions and constants we need. Its payload is small, but the padded `.imx` file will exceed 4 KiB. Do not confuse the code size with the upload file size.

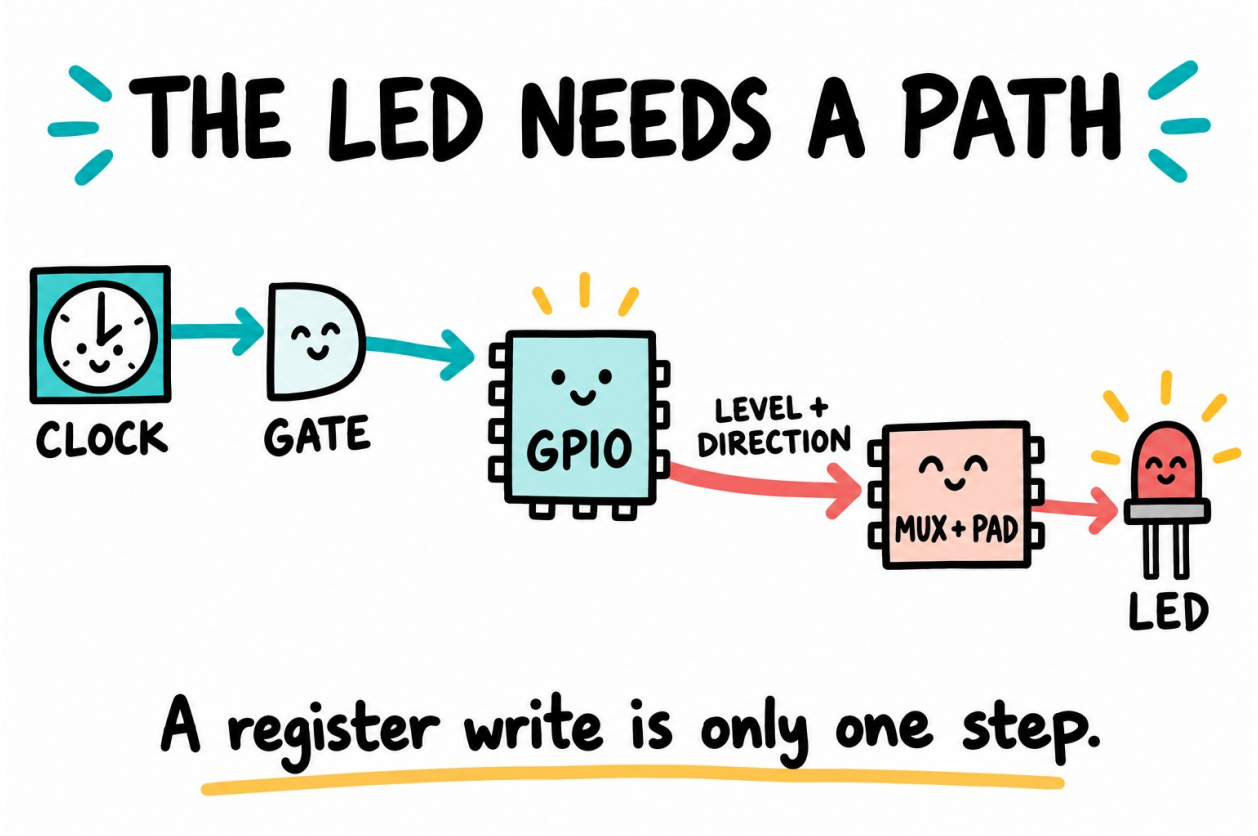
No linker script this chapter. The program is small enough to hand-place. Chapter 10 introduces the linker script as soon as we want C.

On the supplied Point Atom MINI schematic, the user LED is on **GPIO1_IO03**. It is active-low: its anode receives 3.3 V through a resistor, and the GPIO sinks current to light it. A high output turns it off. Confirm the pad and net on your exact board before uploading. For another pin, look up its mux, pad and GPIO registers again. The clock-gate register may remain the same if the pin belongs to the same bank.

The vendor Linux device tree confirms this mapping. It names the device `sys-led` and declares GPIO bank 1, bit 3, active-low.

9.2 Clock, pad and output

Think in three jobs rather than three literal writes. This example uses more than three register writes:



Follow the clock into GPIO, then follow the output toward the LED. Both paths must be ready. The blocks show responsibilities, not separate chips or a count of register writes.

1. **Enable the GPIO controller's clock**, using its two-bit field in `CCM_CCGRx`. Do not rely on reads or writes while its required clock is stopped.
2. **Route the pin to its GPIO function**, by writing the ALT number to `IOMUXC_SW_MUX_CTL_PAD_<padname>`. Without this, the pin still belongs to whatever default function the silicon picked at reset (often a different peripheral).
3. **Choose the output level and direction.** Preload the LED-off level in `GPIO<bank>_DR`, then set the bit in `GPIO<bank>_GDIR` to enable the output. This avoids deliberately exposing an old latch value when the direction changes.

You can also write `IOMUXC_SW_PAD_CTL_PAD_<padname>` to set drive strength, slew rate, and pulls. We use `0x17059`, the pad value from the vendor Linux device tree for this LED pin.

Addresses for GPIO1_IO03, from the Reference Manual:

Register	Address	Purpose
`CCM_CCGR1`	`0x020C406C`	Bits 26:27 (CG13) = GPIO1 clock gate
`IOMUXC_SW_MUX_CTL_PAD_GPIO1_IO03`	`0x020E0068`	MUX select for pin GPIO1_IO03
`IOMUXC_SW_PAD_CTL_PAD_GPIO1_IO03`	`0x020E02F4`	Pad properties for pin GPIO1_IO03
`GPIO1_DR`	`0x0209C000`	GPIO1 data register (bit 3 = our pin, i.e. `1 << 3` for GPIO1_IO03)
`GPIO1_GDIR`	`0x0209C004`	GPIO1 direction register. Bit 3 = direction for GPIO1_IO03. 1 = output.

The value to write into `MUX_CTL` for GPIO function is **5**. The IOMUX table in RM Chapter 32 says, for the pad `GPIO1_IO03`, ALT5 is `GPIO1_IO03`.

CCM_CCGR encoding (2 bits per gate)

Every CCM_CCGRx register holds **16 clock gates × 2 bits each** = 32 bits. The 2-bit field per gate is:

Bits	Meaning
`00`	**Clock off** in all CPU run modes, peripheral cannot be accessed
`01`	Clock on in RUN mode, **off** in WAIT and STOP, low-power-friendly
`10`	*Reserved*, do not program this value
`11`	Clock on in RUN and WAIT, off in STOP

For this running program, write `0b11` into CG13. Its field occupies bits 26-27, so the mask is `0b11 << 26 = 0x0C000000`. OR that mask into the existing value to preserve the other fields. Writing `0xFFFFFFFF` would also modify unrelated and reserved fields. Chapter 13 follows the clock upstream. Here we only need its gate.

9.3 The assembly source

In a new host terminal, run `. ~/imx6ull/scripts/env.sh`, then create `~/imx6ull/src/ch09-asm-led` and open it in your editor. Save the following as `led.S` there. Save the Makefile from Section 9.4 beside it. Lines beginning with `$` in command examples show the shell prompt. Do not type the `$`.

```
    .syntax unified
    .cpu    cortex-a7
    .arm
    .text
    .global _start

_start:
    cpsid   if, #0x13             @ SVC mode, IRQ and FIQ masked
    /* ------------------------------------------------------------
     *  1. Establish a stack.  The Boot ROM has used part of OCRAM
     *     for its own bookkeeping. This program does not use the stack.
     *     OCRAM ends at 0x00920000 (128 KB starting at 0x00900000).
     *     We set SP just below that.  An LED blink doesn't actually
     *     touch the stack. The ROM has already handed off before this runs.
     * ------------------------------------------------------------ */
    ldr     sp, =0x0091FFF0

    /* ------------------------------------------------------------
     *  2. Enable the GPIO1 clock gate.
     *     CCM_CCGR1 @ 0x020C406C, CG13 = bits 26:27 = 0b11 (RUN/WAIT)
     * ------------------------------------------------------------ */
    ldr     r0, =0x020C406C         @ &CCM_CCGR1
    ldr     r1, [r0]
    orr     r1, r1, #(3 << 26)      @ set CG13 = 0b11
    str     r1, [r0]

    /* ------------------------------------------------------------
     *  3. IOMUX: select ALT5 (GPIO function) for pad GPIO1_IO03.
     * ------------------------------------------------------------ */
    ldr     r0, =0x020E0068         @ &IOMUXC_SW_MUX_CTL_PAD_GPIO1_IO03
    mov     r1, #5
    str     r1, [r0]

    /* ------------------------------------------------------------
     *  4. Configure the pad with the value used by the vendor Linux
     *     device tree for this board's LED pin.
     * ------------------------------------------------------------ */
    ldr     r0, =0x020E02F4         @ &IOMUXC_SW_PAD_CTL_PAD_GPIO1_IO03
    ldr     r1, =0x00017059
    str     r1, [r0]

    /* ------------------------------------------------------------
     *  5. Preload the active-low LED's off level, then enable output.
```

```
     * ------------------------------------------------------------ */
    ldr     r0, =0x0209C000         @ &GPIO1_DR
    ldr     r1, [r0]
    orr     r1, r1, #(1 << 3)       @ high = LED off
    str     r1, [r0]

    ldr     r0, =0x0209C004         @ &GPIO1_GDIR
    ldr     r1, [r0]
    orr     r1, r1, #(1 << 3)
    str     r1, [r0]

    /* ------------------------------------------------------------
     *  6. Blink loop. Toggle bit 3 in GPIO1_DR, delay, repeat.
     *     This is only a rough delay. It is intentionally long enough
     *     to remain visible when the CPU clock is faster than expected.
     * ------------------------------------------------------------ */
    ldr     r4, =0x0209C000         @ &GPIO1_DR
    ldr     r5, [r4]                @ current value
    mov     r6, #(1 << 3)           @ bit mask for pin 3

blink:
    eor     r5, r5, r6              @ toggle bit 3 in our saved copy of GPIO1_DR
    str     r5, [r4]                @ write back

    ldr     r7, =50000000         @ long crude delay, not a calibrated interval
1:  subs    r7, r7, #1
    bne     1b

    b       blink

    .end
```

A few notes on what's there and what isn't:

- **No private exception vectors.** We explicitly mask IRQ and FIQ, but those masks do not prevent undefined-instruction or abort exceptions. This first listing assumes valid instructions and accesses. Chapter 10 adds placeholder vectors, and Chapter 15 adds handlers.
- **No `.data`, no `.bss`.** Every value we use is an immediate or computed at run time. Therefore no startup code is needed to copy or zero anything.
- **No `main()`.** `_start` is the entry. It never returns. An assembly program has no caller to return to. You must explicitly loop forever.
- **`ldr r0, =0x...`** asks GNU assembler to load a constant. It may use an encodable immediate instruction or a PC-relative load from a literal pool. Inspect the disassembly to see which it chose. Arbitrary 32-bit constants do not all fit one ARM immediate instruction.
- **`1:` is a local label.** `1b` means "branch to the nearest `1` label going backward." This is a GAS convention for local loops. It avoids us inventing new names.

- **`.syntax unified`** says “use the modern ARM/Thumb-unified mnemonics”, which lets us write `orr r1, r1, ...` even in ARM mode without surprises.

The delay count is deliberately large. A short loop that looked slow on an MCU may make this LED appear steady. CPU clock, instruction-cache state and branch timing all affect the interval, so do not label it as milliseconds. Reducing the count changes the interval without establishing a calibrated frequency. Chapter 16 supplies a measured timebase.

9.4 Building the image

Two files: `led.S` and a small `Makefile`.

```
# Makefile
CROSS := arm-none-eabi-

all: led.bin

led.o: led.S Makefile
    $(CROSS)gcc -mcpu=cortex-a7 -marm -ffreestanding -c -o $@ $<

led.elf: led.o Makefile
    $(CROSS)ld -Ttext=0x00908000 -e _start -o $@ $<

led.bin: led.elf
    $(CROSS)objcopy -O binary --only-section=.text $< $@

clean:
    rm -f led.o led.elf led.bin led.imx

.PHONY: all clean
```

What is going on:

- **`-c`** assembles `led.S` into `led.o` without linking it.
- **`-Ttext=0x00908000`** tells the linker that `.text` will run from address `0x00908000`. The image wrapper below places the first byte of `led.bin` at exactly that address.
- **`-e _start`** records `_start` as the ELF entry point. The i.MX IVT also uses this same address.
- **`--only-section=.text`** copies only our code and its literal pool into `led.bin`.

From this project directory, build. The commands below the prompt are representative output. The byte count depends on the listing and assembler:

```
$ make
arm-none-eabi-gcc -mcpu=cortex-a7 -marm -ffreestanding -c -o led.o led.S
arm-none-eabi-ld -Ttext=0x00908000 -e _start -o led.elf led.o
```

```
arm-none-eabi-objcopy -O binary --only-section=.text led.elf led.bin
$ wc -c led.bin
```

Check the ELF before wrapping it:

```
$ arm-none-eabi-readelf -h led.elf | grep 'Entry point'
  Entry point address:               0x908000

$ arm-none-eabi-objdump -d led.elf | head
00908000 <_start>:
```

Both commands must show `0x00908000`. If they show another address, stop and fix the build before using `uuu`.

9.5 Wrapping the .bin in an .imx

`led.bin` is raw machine code. The Boot ROM in SDP mode does *not* execute raw bins, it executes images that present an IVT (Chapter 7). We need to wrap.

For this chapter, a short Python script places each structure at an exact file offset. Chapter 11 explains and extends this image builder.

Save as `wrap.py`:

```
#!/usr/bin/env python3
import struct
from pathlib import Path

IMAGE_START = 0x00907000
IVT_OFFSET  = 0x00000400
CODE_OFFSET = 0x00001000

IVT_ADDR  = IMAGE_START + IVT_OFFSET   # 0x00907400
ENTRY_ADDR = IMAGE_START + CODE_OFFSET # 0x00908000
BOOT_ADDR  = IVT_ADDR + 0x20           # 0x00907420

code = Path('led.bin').read_bytes()
image_size = CODE_OFFSET + len(code)
if not code or IMAGE_START + image_size > 0x00918000:
    raise SystemExit('Payload must be nonempty and fit the ROM-active OCRAM window')

# IVT header: tag, 16-bit big-endian length, version.
ivt = struct.pack('>BHB', 0xD1, 0x0020, 0x40)
ivt += struct.pack('<IIIIIII',
    ENTRY_ADDR, # entry
    0,          # reserved1
    0,          # dcd, none for this OCRAM program
    BOOT_ADDR,  # boot_data
    IVT_ADDR,   # self
    0,          # csf, image is not signed
    0)          # reserved2

# BootData describes the complete image, including the bytes before the IVT.
boot_data = struct.pack('<III', IMAGE_START, image_size, 0)
```

```
image = bytearray(b'\x00' * CODE_OFFSET)
image[IVT_OFFSET:IVT_OFFSET + len(ivt)] = ivt
image[IVT_OFFSET + 0x20:IVT_OFFSET + 0x20 + len(boot_data)] = boot_data
image += code

# UUU loads file offset 0x400 at IVT_ADDR. Therefore file offset 0x1000
# lands at IVT_ADDR + (0x1000 - 0x400), which must equal ENTRY_ADDR.
loaded_code_addr = IVT_ADDR + (CODE_OFFSET - IVT_OFFSET)
assert loaded_code_addr == ENTRY_ADDR
assert image[CODE_OFFSET:] == code

Path('led.imx').write_bytes(image)
print(f'IVT:  file 0x{IVT_OFFSET:04X} -> RAM 0x{IVT_ADDR:08X}')
print(f'code: file 0x{CODE_OFFSET:04X} -> RAM 0x{ENTRY_ADDR:08X}')
print(f'size: {len(image)} bytes')
```

```
$ python3 wrap.py
IVT:  file 0x0400 -> RAM 0x00907400
code: file 0x1000 -> RAM 0x00908000
size: ... bytes
```

Your size may differ slightly. The two addresses must match the output above.

The important layout is:

File offset	Loaded RAM address	Content
`0x0000` to `0x03FF`	Not uploaded by UUU	Padding used by SD and eMMC boot images
`0x0400`	`0x00907400`	IVT
`0x0420`	`0x00907420`	BootData
`0x042C` to `0x0FFF`	`0x0090742C` onward	Padding
`0x1000`	`0x00908000`	First instruction in `led.bin`, `_start`

If you decode the IVT now you should see exactly the values we set:

```
$ xxd -s 0x400 -l 32 led.imx
00000400: d100 2040 0080 9000 0000 0000 0000 0000  .. @............
00000410: 2074 9000 0074 9000 0000 0000 0000 0000   t...t..........
```

Tag `D1 00 20 40`, entry `00 80 90 00` (little-endian → `0x00908000`), dcd zero, boot_data `20 74 90 00` (→ `0x00907420`), self `00 74 90 00` (→ `0x00907400`). Matches.

9.6 Pushing to the board with uuu

1. Follow Chapter 8's checked power-down and power-up procedure. Set the boot-mode switch to **SDP** (USB-Downloader) while the board is unpowered. Account for all attached power sources and UART back-power paths.
2. Connect the USB-OTG cable to the host.
3. Power on.
4. Confirm enumeration:

```
$ lsusb | grep 15a2
Bus 001 Device 010: ID 15a2:0080 Freescale SemiConductor Inc i.MX 6 SystemOnChip in
RecoveryMode
```

1. Push the image:

```
$ sudo "$IMX6ULL_HOME/build/mfgtools/uuu/uuu" led.imx
... release/banner output varies; the following is the expected command shape ...

Success 1    Failure 0
3:2-         2/ 2 [Done                                  ] SDP: done
```

When the argument is an image instead of a UUU command file, `uuu` automatically creates this command:

```
SDP: boot -f <image>
```

UUU then performs these steps:

1. Find the IVT at file offset `0x400`.
2. Upload from that offset to `IVT.self`, which is `0x00907400`.
3. Ask the ROM to process the IVT at `0x00907400`.
4. The ROM reads `IVT.entry` and branches to `_start` at `0x00908000`.

`Success 1 Failure 0` means that the USB transfer and ROM commands succeeded. It does not prove that `_start` contains valid code at the entry address.

Watch the LED. It should blink.

If it does not:

1. **Check all three execution addresses.** `readelf`, `objdump`, and `wrap.py` must all show `_start` at `0x00908000`.
2. **Check the image layout.** `wrap.py` must report code at file offset `0x1000` and RAM address `0x00908000`.
3. **Check the LED's pin.** This code controls `GPIO1_IO03`. Confirm that this is the user LED in your exact board revision.
4. **Check the blink delay.** A very short delay makes the LED look continuously on or dim. The constant is not a promised half-second interval.
5. **Power-cycle and retry.** After the ROM jumps to your program, it cannot accept another SDP upload until the board resets.

9.7 What happened, step by step

If you got the blink, this sequence ran on real silicon:

```
Power on
  → Boot ROM runs from internal ROM at 0x00000000
    → reads BOOT_MODE pins, sees SDP
    → enumerates as USB device 15a2:0080
    → waits for host commands
host: uuu pushes led.imx over USB
  → UUU skips the file's first 0x400 bytes
  → ROM receives the IVT and following bytes at RAM 0x00907400
  → ROM receives JUMP_ADDRESS: 0x00907400
ROM:
  → finds IVT signature 0xD1 at 0x00907400
  → reads IVT.dcd (zero, skip DCD)
  → reads IVT.entry = 0x00908000, jumps there
Your code:
  → _start sets SP
  → enables GPIO1 clock gate
  → sets pin ALT5
  → sets pin direction = output
  → enters blink loop
LED blinks. The ROM executed first. From the handoff into `_start`, the instructions
controlling this LED are ours.
```

No software layer sits between this program and the chip. The next chapters add layers on top of this direct register-access model.

9.8 Key Notes

- The ROM owns loading and the initial handoff. The assembly payload owns the GPIO configuration after entry, with no C runtime, scheduler or DDR dependency.

- The clock gate, mux selection and output latch/direction serve different purposes. A writable GPIO register does not establish that its signal reaches the intended pad.
- Preloading the active-low LED's off level before enabling output avoids exposing a stale latch value. Active-low polarity does not imply an open-drain driver.
- The ELF entry, IVT entry and loaded code location must agree. USB command success establishes a transfer result, not execution of the intended instructions.
- A busy-loop delay depends on clock and instruction timing. Its count is not a duration, and rapid toggling can look like a steady LED.
- A header pin is not automatically a spare output. Its voltage domain, connected loads and boot role constrain use. Never drive a supply, connected output or boot strap as though it were an LED connection.

9.9 Pitfalls

- **Forgetting the CCGR write.** Confirm the required gate before interpreting register accesses. A stopped-clock access does not have one universal, safe failure value.
- **Wrong IOMUX ALT.** Symptom: writes to GPIO_DR succeed but the pin doesn't move. Some pads default to "GPIO" in their reset ALT. Many do not. Always set ALT explicitly.
- **Entry address does not match code location.** Symptom: `uuu` reports success, but the board does nothing. The host cannot tell whether valid instructions exist at `IVT.entry`. Check the file-to-RAM map in Section 9.5.
- **Delay is too short.** The LED may look continuously on even though the pin is toggling quickly.
- **Resetting after a successful upload.** This image is in RAM. A new downloader boot waits for another upload. The SDP path leaves persistent storage unchanged. Switching to SD requires a separately prepared and verified bootable card as described in Chapter 11.
- **Mistaking active-low for open-drain.** Active-low describes which level lights the LED. Open-drain describes the output driver. They are different properties. Our pad setting uses a push-pull output.
- **Optimization eating your loop.** GCC with `-O2` may unroll or completely eliminate a delay loop with no side-effects. We avoided this here by leaving the loop in raw asm. If you port to C, mark the counter `volatile`.

9.10 Going deeper

- **IMX6ULLRM Chapter 28, GPIO**: Specifically Table 28-1 (register summary) and Table 28-3 (GPIOx_DR bit layout).

- **IMX6ULLRM Chapter 32, IOMUXC**: Look up GPIO1_IO03 in the IOMUX table.
- **IMX6ULLRM Chapter 18, CCM**: Table 18-3 (gate encoding) and Section 18.6.24 (`CCM_CCGR1`). Table 18-5 in Chapter 8 is the ROM's disabled-clock list, not the gate-encoding table.
- **ARM DDI 0406** Section A8.8.62, `LDR (literal)` form, which is what `ldr Rn, =const` expands into.
- The GNU Assembler manual, "ARM Dependent Features", `.syntax unified`, `.cpu`, `.global`, literal pools.
- Your **Point Atom MINI schematic**, the only authoritative source for which LED is on which pin on *your* board.

The LED is a simple execution witness in this design. Chapter 10 keeps the same output path and changes the language, supplying the stack and memory initialization that C expects before `main()` can run.

Chapter 10: C + startup.S + linker script

Chapter 9's assembly model reaches the LED without a C runtime. Rewriting its loop in C ought to be a small change, but C arrives with expectations: a usable stack, initialized globals and zero-initialized static storage. The compiler generates code on that basis. It does not arrange those conditions for this flat, ROM-loaded image.

We will keep the same LED and supply the missing work before calling `main()`. The linker script describes where code and objects belong. `startup.S` uses that description to prepare memory. Seeing those two files agree is more useful than treating startup as a block to paste into every project.

10.1 What an initialized global needs

Consider a C file with three globals:

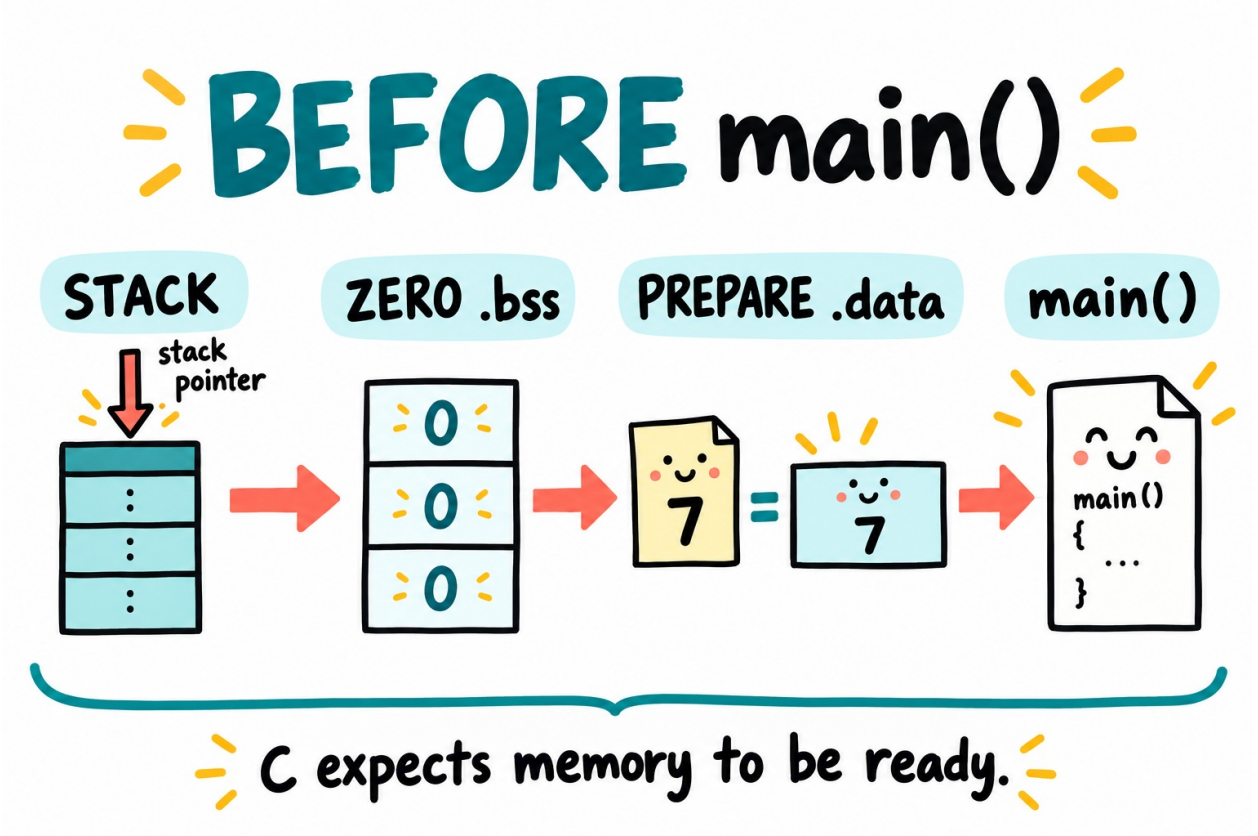
```
int   x = 7;           // initialized → .data
int   y;               // uninitialized → .bss
const int z = 42;      // const + initialized → .rodata
```

On Linux, the loader establishes the ELF's loadable memory segments and zero-filled storage before the C runtime calls `main()`. Here the Boot ROM loads our image, but it does not interpret the ELF or initialize C objects. We give it a flat binary wrapped in an IVT.

So we, ourselves, must:

- **Set an aligned stack pointer.** A C function may use the stack even when its source has no obvious local array or function call.
- **Zero `.bss`.** Otherwise `y` is whatever was in OCRAM when we arrived.
- **Copy `.data` from its load location to its run location when those differ.** This is the LMA-versus-VMA distinction from Chapter 6.
- **Branch to `main`.**

Optionally, also: set up exception vectors, configure caches, enable the FPU. We do these later as we need them.



These are promises made to C before it runs. Preparing data means putting the initial value at its runtime address. In our first layout it is already there, so a copy is not yet a relocation test.

10.2 A linker script worth keeping

Create `~/imx6ull/src/ch10-c-startup` in your editor. Save the four complete listings in this chapter as `link.ld`, `startup.S`, `main.c` and `Makefile`. Copy Chapter 9's `wrap.py` into this directory. In each new build terminal, run `. ~/imx6ull/scripts/env.sh` before entering the directory.

The Chapter 9 program needed only `.text`. Now we also describe initialized storage, zero-initialized storage and space for a stack. Save this as `link.ld`:

```
ENTRY(_vectors)

MEMORY
{
    OCRAM (rwx) : ORIGIN = 0x00908000, LENGTH = 0x00018000  /* 96 KB */
}

SECTIONS
{
    . = ORIGIN(OCRAM);

    .text ALIGN(32) : {
        KEEP(*(.vectors))      /* room for vector table later (Ch 15) */
        *(.text*)
        *(.rodata*)
        . = ALIGN(4);
```

```
    } > OCRAM

    /* Mark the end of .text -- LMA of .data begins here. */
    _etext = .;

    /*
     * .data : runtime in OCRAM, image-time directly after .text.
     * Because OCRAM and our image both live in the same region, LMA == VMA
     * for this layout. The startup copy writes each word back to itself.
     * Keep the source and destination explicit for a later split layout.
     */
    .data ALIGN(4) : AT(_etext) {
        _sdata = .;
        *(.data*)
        . = ALIGN(4);
        _edata = .;
    } > OCRAM
    _sidata = LOADADDR(.data);

    .bss ALIGN(4) (NOLOAD) : {
        _sbss = .;
        *(.bss*)
        *(COMMON)
        . = ALIGN(4);
        _ebss = .;
    } > OCRAM

    _stack_top = ORIGIN(OCRAM) + LENGTH(OCRAM);
    _stack_limit = _stack_top - 0x1000;  /* reserve 4 KiB after ROM handoff */
    ASSERT(_ebss <= _stack_limit, "Objects overlap the reserved stack")
    ASSERT(_edata <= 0x00918000, "Loaded bytes overlap ROM-active RAM")

    /DISCARD/ : { *(.note*) *(.comment) *(.ARM.exidx*) *(.ARM.extab*) }
}
```

Decoded line by line:

- **`ENTRY(_vectors)`** records the ELF entry at our first instruction, `b _start`, at `0x00908000`. The wrapper uses the same address in the IVT. `_start` itself follows the eight vector slots. The ROM reads the IVT, not this ELF field.
- **`MEMORY { OCRAM ... }`**: describes our one available region. `ORIGIN` is where the wrapper places the first program byte. `LENGTH` covers the remaining 96 KB up to the end of OCRAM at `0x00920000`.
- **`. = ORIGIN(OCRAM);`**: the location counter starts at the region's base.
- **`.text` section**: gathers all `.text*`, `.rodata*`, plus a `KEEP(*(.vectors))` placeholder for a future vector table. `KEEP` tells the linker not to remove this as unused, even if no symbol references it. The final `. = ALIGN(4);` makes `_etext` word-aligned.
- **`_etext = .;`**: captures the location counter. This is where `.text` ends. It is also where the `.data` *load image* will be placed (see next line).

- **`.data ALIGN(4) : AT(_etext)`**: `ALIGN(4)` aligns the runtime address. `AT(_etext)` sets the load address. Notice their positions around the colon. GNU `ld` rejects `.data : ALIGN(4) AT(_etext)` because `AT(...)` must come before a post-colon `ALIGN(...)`.
- **`_sidata = LOADADDR(.data)`**: asks the linker for `.data`'s actual load address. Startup code uses this symbol as its copy source.
- **`_sdata` / `_edata`**: boundary symbols our startup uses to know how much to copy.
- **`.bss (NOLOAD)`** reserves zero-initialized storage without carrying its contents in our load image. Startup clears the range between `_sbss` and `_ebss`. We round both data-section ends to four bytes because our loops copy or clear whole words.
- **`_stack_top` / `_stack_limit`** reserve 4 KiB for a descending stack. This is a budget, not an automatic stack-overflow guard. Loaded bytes must fit below `0x00918000` while ROM is active. Our own stack uses the upper OCRAM only after handoff, without returning to ROM services.
- **`/DISCARD/`**: throws away ELF notes and attributes that have no place in a bare-metal binary.

Four layout decisions explain common startup failures:

1. **Putting `ALIGN` and `AT` in the wrong order.** Use `.data ALIGN(4) : AT(_etext)`. Linker-script keywords have a fixed grammar, and `ld` reports only a line number when this order is wrong.
2. **Forgetting `KEEP` around the vector table.** When you later link with `-gc-sections`, the linker removes the table because nothing in C references it. `KEEP` prevents this.
3. **Assuming the linker moves bytes.** `AT(...)` describes a load address. It does not perform the startup copy. Without an explicit LMA, GNU ld uses layout-dependent rules, not a universal promise that it equals the VMA.
4. **Assuming `.bss` needs stored zeros.** Ordinary ELF `.bss` is normally `NOBITS` even without `NOLOAD`. Check its type and the raw binary size. The explicit `NOLOAD` makes our layout's intention clear.

10.3 startup.S, the bridge from reset to `main`

```
    .syntax unified
    .cpu    cortex-a7
    .arm
    .section .vectors, "ax"
    .align  5                       @ vector table must be 32-byte aligned
    .global _vectors
_vectors:
    b       _start                  @ Reset, replaced in Ch 15
    b       .                       @ Undef
    b       .                       @ SVC
    b       .                       @ Prefetch abort
```

```
    b       .                       @ Data abort
    b       .                       @ Reserved
    b       .                       @ IRQ
    b       .                       @ FIQ

    .section .text.startup, "ax"
    .global _start
_start:
    /* Direct entry from a fresh open-device ROM boot, not U-Boot.
       Mask asynchronous interrupts and select our working mode. */
    cpsid   if, #0x13               @ mode = SVC, mask IRQ+FIQ

    /* Use our aligned placeholder vectors instead of ROM handlers. */
    mrc     p15, 0, r0, c1, c0, 0
    bic     r0, r0, #(1 << 13)      @ SCTLR.V = 0, use VBAR
    mcr     p15, 0, r0, c1, c0, 0
    ldr     r0, =_vectors
    mcr     p15, 0, r0, c12, c0, 0
    isb

    /*  Stack: top of OCRAM, defined by the linker. */
    ldr     sp, =_stack_top

    /*  Zero .bss : for (p = &_sbss; p < &_ebss; p++) *p = 0; */
    ldr     r0, =_sbss
    ldr     r1, =_ebss
    mov     r2, #0
1:  cmp     r0, r1
    strlo   r2, [r0], #4
    blo     1b

    /*  Copy .data from LMA to VMA. In our current layout they are equal,
        so each word is copied onto itself. When .data moves to another
        memory region, this same loop performs the required relocation. */
    ldr     r0, =_sidata            @ source (LMA)
    ldr     r1, =_sdata             @ destination (VMA)
    ldr     r2, =_edata
2:  cmp     r1, r2
    ldrlo   r3, [r0], #4
    strlo   r3, [r1], #4
    blo     2b

    /* main(void) takes no arguments. */
    bl      main

    /*  main() should never return.  If it does, halt cleanly. */
hang:
    wfi
    b       hang
```

A few notes on the assembly choices:

- **`cpsid if, #0x13`** is a `cps` instruction with the side effect of setting the mode bits to `0b10011` (SVC) and the I and F mask bits. One instruction. Three guarantees.

- **`strlo r2, [r0], #4`** stores a word, then advances the pointer by four bytes when the preceding unsigned comparison says `r0 < r1`. `lo` is the condition-code spelling for unsigned lower, not an AAPCS-specific flag.
- **`bl main`** records a return address in LR and branches. If `main` returns, `hang` repeatedly executes `wfi`. This supplies a defined place to stop, not a full power-management configuration.
- **`.section .text.startup`** puts our startup code in a named subsection. The linker script's `*(.text*)` matches `.text.startup` and pulls it in early. We could put it in plain `.text`, but the explicit name makes the startup code easier to find.
- The vector table contains a reset-slot branch followed by seven self-loops. We install its address in VBAR here, but do not enable IRQ/FIQ. Chapter 15 replaces the placeholders with useful handlers.

Do not read CPU reset defaults as ROM handoff defaults. RM 8.4.4 says the ROM enables the instruction cache during download and disables the data caches and MMU after authentication. This startup is for that direct ROM path. A jump from an existing bootloader may require cache cleanup, MMU changes and peripheral reinitialization that this listing does not supply. Chapter 17 handles our own memory attributes and caches.

10.4 `main.c`, the LED, again

```
#include <stdint.h>

#define REG(addr) (*(volatile uint32_t *)(addr))

#define CCM_CCGR1   0x020C406C
#define IOMUX_MUX   0x020E0068
#define IOMUX_PAD   0x020E02F4
#define GPIO1_DR    0x0209C000
#define GPIO1_GDIR  0x0209C004
#define LED_BIT     (1u << 3)

static void delay(volatile uint32_t n)
{
    while (n--) { asm volatile ("nop"); }
}

int main(void)
{
    REG(CCM_CCGR1) |= (3u << 26);   /* GPIO1 clock on */
    REG(IOMUX_MUX) = 5;             /* ALT5 = GPIO */
    REG(IOMUX_PAD) = 0x17059;       /* vendor LED pad setting */
    REG(GPIO1_DR) |= LED_BIT;       /* preload LED off */
    REG(GPIO1_GDIR) |= LED_BIT;     /* output */

    for (;;) {
        REG(GPIO1_DR) ^= LED_BIT;
        delay(50000000u);
```

```
    }
}
```

Three things that look small but matter:

- **`volatile` on the cast.** Without `volatile`, the optimizer is free to assume `REG(GPIO1_DR)` reads always return the same value, and to elide the second read entirely in a tight loop. With `volatile`, the compiler emits a real load-store every time. Every MMIO access in this book is `volatile`.
- **The busy delay.** The volatile counter and `asm volatile ("nop")` keep observable work in the loop. Neither makes the duration precise. A `nop` is not a memory barrier. Chapter 16 replaces instruction counting with a timer.
- **`(3u << 26)`.** Unsigned masks make bitwise intent explicit. This particular signed shift would also fit a 32-bit `int`, but higher-bit masks may not.

10.5 The Makefile

```
CROSS    := arm-none-eabi-
CC       := $(CROSS)gcc
LD       := $(CROSS)ld
OC       := $(CROSS)objcopy
SIZE     := $(CROSS)size

ARCH     := -mcpu=cortex-a7 -marm -mfloat-abi=soft -mgeneral-regs-only
CFLAGS   := $(ARCH) \
            -ffreestanding -fno-builtin -nostdlib \
            -fno-common -fno-unwind-tables -fno-asynchronous-unwind-tables \
            -MMD -MP -O2 -g -Wall -Wextra -Werror=implicit-function-declaration

LDFLAGS  := $(ARCH) -T link.ld -nostdlib

OBJS     := startup.o main.o
DEPS     := $(OBJS:.o=.d)

all: led.bin

%.o: %.S Makefile
    $(CC) $(CFLAGS) -c -o $@ $<

%.o: %.c Makefile
    $(CC) $(CFLAGS) -c -o $@ $<

led.elf: $(OBJS) link.ld Makefile
    $(CC) $(LDFLAGS) -o $@ $(OBJS) -lgcc
    $(SIZE) $@

led.bin: led.elf
    $(OC) -O binary $< $@

clean:
    rm -f $(OBJS) $(DEPS) led.elf led.bin led.imx
```

```
-include $(DEPS)

.PHONY: all clean
```

A couple of flags worth highlighting:

- **`-marm -mfloat-abi=soft -mgeneral-regs-only`** keeps these early integer-only C examples in ARM state and prevents GCC from using VFP/NEON registers before we enable that hardware. Use the same architecture/ABI options at link time. We explicitly add `-lgcc` after our objects for compiler helpers such as integer division. This is not libc.
- **`-MMD -MP` and `Makefile` prerequisites** rebuild objects when included headers or build flags change. Otherwise an old object can survive an apparently successful `make`.
- **`-fno-common`**: forces every uninitialized global into `.bss` instead of “common” symbols. Without this, two files declaring `int foo;` would merge without a clear warning. That is convenient on hosted Linux and dangerous on bare-metal.
- **`-Werror=implicit-function-declaration`**: we never tolerate “I forgot to include the header.” It is one of the cheapest bugs to prevent.
- **`-O2 -g`** together, optimize but keep DWARF. The `-g` does not affect the binary. It only enlarges the ELF.

10.6 Building and running

```
$ make
$ arm-none-eabi-size led.elf
$ arm-none-eabi-readelf -h led.elf
$ arm-none-eabi-nm -n led.elf
$ wc -c led.bin
```

A few observations:

- **`text` grows.** The image now includes the vector slots, memory-initialization loops and compiled C. Read your own size output rather than expecting one fixed byte count.
- **`data` is 0.** No initialized globals in our C.
- **`bss` is 0.** No uninitialized globals.

The loops are present, but their ranges are empty. A worked variant uses an initialized object instead of the `LED_BIT` macro:

```
static volatile uint32_t led_mask = (1u << 3);   /* stored in .data */
```

In that variant, `led_mask` replaces each use of `LED_BIT`. The volatile qualifier keeps GCC from replacing this demonstration object with a constant. The section-inspection commands are:

```
$ arm-none-eabi-size led.elf
$ arm-none-eabi-readelf -SW led.elf
$ arm-none-eabi-nm -n led.elf
```

The object occupies four bytes. Compare `_sidata` and `_sdata`: in this layout they are equal. The ROM has already placed the initial value at its runtime address. The loop copies it onto itself, so disabling the copy would not demonstrate a failure. A separate source and destination are needed to test relocation, as in the later DDR work.

Check `_vectors` and the ELF entry at `0x00908000`. `_start` is after the vector slots. If GNU ld reports an RWX load-segment warning, it describes this small combined code/data layout. It is not a report that the MMU has installed page permissions.

Wrap into `.imx` with the same `wrap.py` from Chapter 9 and push:

```
$ python3 wrap.py
$ sudo "$IMX6ULL_HOME/build/mfgtools/uuu/uuu" led.imx
```

An LED blink from this image would show that startup reached compiled C. That observation alone does not test empty `.bss` or a relocated `.data` section. Section 10.10 distinguishes those memory-initialization guarantees.

10.7 Stepping through with `objdump`

It is worth reading the disassembled startup once. After `make`:

```
$ arm-none-eabi-objdump -d led.elf | head -80
```

Find `_start`. Follow the working-mode and vector setup, then the stack, clear, copy and call blocks:

1. The mode-setting `cpsid` instruction.
2. The `ldr sp, =_stack_top` literal load.
3. The `.bss` zero loop.
4. The `.data` copy loop.
5. The `bl main`.

The literal pool follows the function. You can see the resolved addresses there.

A different linker-script `OCRAM` origin changes every code address. For example, an origin of `0x00909000` would no longer agree with the wrapper's default entry at `0x00908000`. The image wrapper's entry and code placement must change with the linker layout, not independently of it.

10.8 What if `main()` returns?

In `startup.S`, after `bl main`, we fall through to a `hang` loop. In normal bare-metal code, `main()` should never return. During development, it can happen by accident, for example from an unintended `return` or `if (...) return;`. The `hang` loop gives you a stable failure state instead of letting execution continue into unknown memory.

You can make the dependency explicit by giving `main` the `__attribute__((noreturn))`:

```
__attribute__((noreturn)) int main(void) { ... }
```

GCC then warns if `main` has a code path that returns. Optional but informative.

10.9 Why `volatile`, one more time

A common bug:

```
*(uint32_t *)CCM_CCGR1 |= (3u << 26);
```

Without `volatile`, the compiler is allowed to:

- assume that `*(uint32_t *)CCM_CCGR1` does not change between reads,
- merge consecutive accesses to the same address,
- remove or combine accesses when ordinary-memory rules permit it.

In the LED program, these freedoms produce code that happens to work, because we touch each register exactly once. But the moment you write code like:

```
while ((REG(UART_STATUS) & TX_EMPTY) == 0) {}
```

…without `volatile`, the compiler treats `UART_STATUS` as constant inside the loop, reads it once before the loop, and spins forever. Most embedded engineers hit this bug once. Avoid it by reflex.

Use volatile access for MMIO, but keep reading the register definition. `REG(x) |= mask` is still a read-modify-write operation. It can be wrong for write-one-to-clear status bits or registers with read side effects. Volatile also does not supply atomicity, cache maintenance or CPU memory ordering. Those are separate decisions.

10.10 Key Notes

- The linker describes placement and exports boundaries. Startup establishes the aligned stack and object contents before calling C. Neither the compiler nor the IVT supplies that runtime preparation.
- `.bss` reserves storage without carrying initialization bytes in the flat image. A nonempty clear range is necessary to establish C's zero-initialization guarantee, regardless of OCRAM's previous contents.
- Equal `.data` LMA and VMA make the copy loop a self-copy. The initialized `led_mask` case explains stored data but cannot demonstrate relocation or expose an omitted copy.
- In a deterministic clear case, a four-byte `.bss` counter containing `0xA5A5A5A5` becomes zero after startup clears its range. A naturally zero SRAM value would not distinguish a working clear from an omitted one.
- The reserved stack and loaded-byte limits protect different lifetimes: ROM-active loading and post-handoff execution. Link-time bounds do not provide a runtime stack-overflow guard.
- `volatile` preserves MMIO accesses at the compiler boundary. Register side effects, read-modify-write safety, CPU ordering and cache attributes remain separate responsibilities.

10.11 Pitfalls

- **`bss` not zeroed.** Symptom: nondeterministic startup behavior across resets. Cause: forgot the loop, or got the `_sbss`/`_ebss` symbols wrong in the linker script.
- **`.data` not copied in a split layout.** When LMA differs from VMA, the initial values must reach the runtime destination before C reads them. The current equal-address layout cannot expose a missing copy.
- **Stack not aligned at function entry.** AAPCS requires SP to be 8-byte aligned at every public function entry. `_stack_top = ORIGIN + LENGTH` aligns naturally as long as LENGTH is a multiple of 8. Change LENGTH to an odd value and expect crashes inside libgcc helpers.
- **`-fno-common` not set.** Two `int foo;` declarations in two `.c` files merge into one symbol without a clear warning. Sometimes the result works, sometimes it corrupts memory. Always enable.
- **Forgot `volatile`.** Discussed above.
- **Linker script does not declare `.rodata`.** GCC may emit string literals into `.rodata`, which falls through to the next region. We folded `.rodata` into `.text` here. If you split them out, make sure both are placed in OCRAM.

- **Misaligned section boundaries.** Our word loops require aligned starts and rounded-up ends. An unrounded byte-sized object can otherwise make the final word store cross a section boundary. Do not rely on a particular alignment-fault configuration to catch it.

10.12 Going deeper

- GNU ld: output-section attributes and load addresses.
- LLVM's `lld` manual has a much shorter introduction to the same concepts, useful for the second-time reader.
- `arm-none-eabi-gcc -E -P -x c /dev/null -include stdint.h`: inspect the target's integer typedefs. Do not put shell redirection brackets around the header name.
- GCC: Arm options and volatile accesses.
- The U-Boot source's `arch/arm/lib/crt0.S`, read it after this chapter. The patterns are the same.

Sidebar, `REG(addr)` macro vs the NXP SDK header

We use raw addresses to keep the register lookup visible. A matching, verified vendor header can instead provide typed register layouts and bit masks. The following illustrates that style, not an extra dependency for this example. Such a header must match the exact vendor SDK or board package, SoC and revision:

```
#include "MCIMX6Y2.h"

UART1->UCR1 = 0;                  // typed access; the struct knows offsets
UART1->UCR2 |= UART_UCR2_TXEN_MASK;
GPIO1->GDIR |= (1u << 3);    // GPIO1_IO03 = output (LED0 cathode side)
```

With matching addresses and types, both styles can produce the same register-access instructions. The trade-offs:

	`REG(addr)` (this book)	NXP SDK header
Explicitness	The address is in your face	Hidden inside the struct
Risk of typos	High, `0x020E0068` vs `0x020E006B`	Low, autocomplete saves you
Portability	One `.h` per SoC family at most	One `.h` per exact part

	`REG(addr)` (this book)	NXP SDK header
Debugger view	`*(uint32_t *)0x020E0068`	`IOMUXC->SW_MUX_CTL_PAD_GPIO1_IO03`
Reading focus	Address and bit-field lookup	Named peripheral layout, still checked against the RM

Chapter 18A separates register definitions from driver policy. Raw addresses and typed headers can both be appropriate. Neither removes the need to verify offsets, reserved bits and register access semantics.

> Next chapter: **Chapter 11: Hand-building a Boot ROM-acceptable image.** The reusable image builder connects IVT fields to file offsets, RAM addresses and primary SD-card placement.

Chapter 11: Hand-building a Boot ROM-acceptable image

The preceding examples use a host upload. Can the same program start with only an SD card in the board?

The ROM does not need a filesystem for this small image. It looks for a boot structure at a particular media offset, reads the image into RAM and uses the entry field to start execution. Following those bytes explains why the first instruction lands where the linker put it. No card write is required to understand the layout.

This remains an unsigned, OCRAM-only image for an open development device. A dedicated spare card is required for the SD route. USB loading can be completed without writing any card.

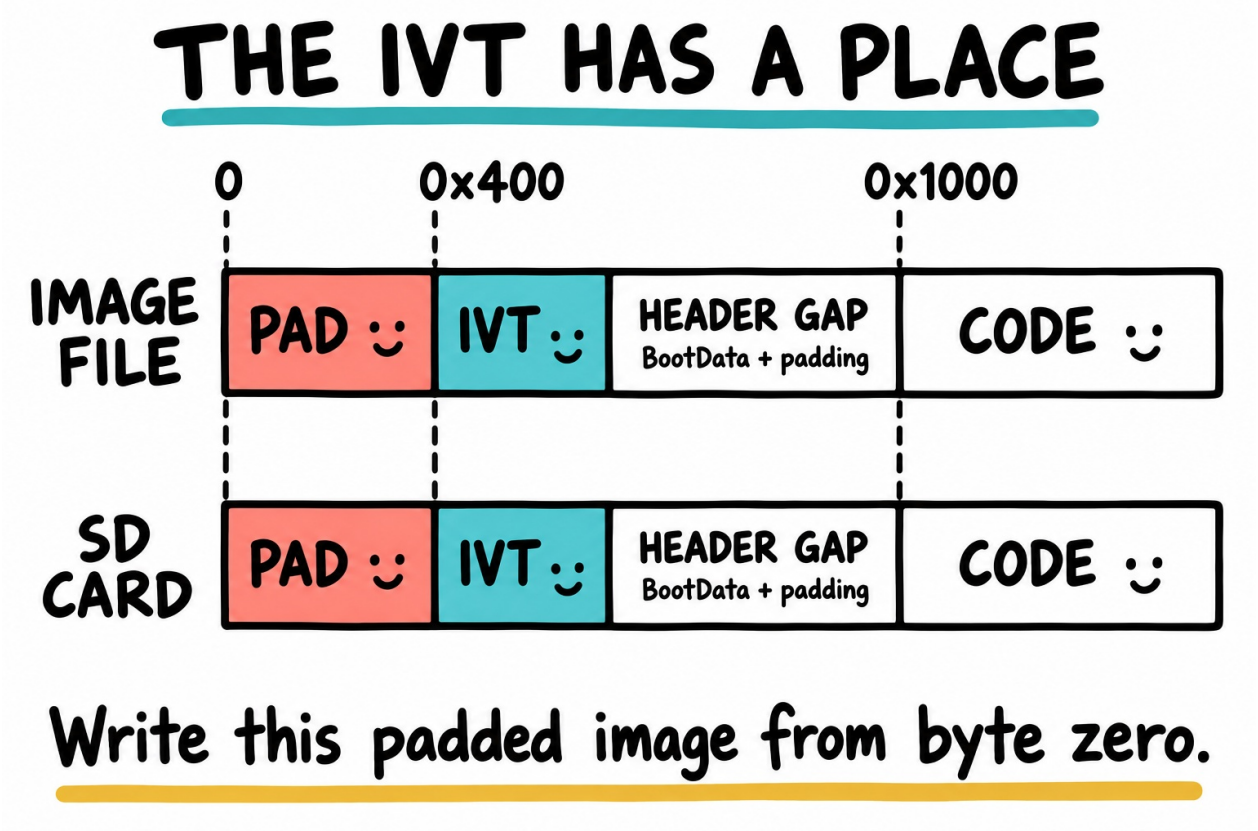
11.1 What we produced last chapter, in detail

Recap the structure of `led.imx` from Chapter 9, viewed as a sequence of file offsets:

```
file offset    content                                  size
0x0000         (pad)                                    0x400 bytes
0x0400         IVT header                               4 bytes
0x0404         IVT.entry                                4 bytes
0x0408         IVT.reserved1                            4 bytes
0x040C         IVT.dcd                                  4 bytes
0x0410         IVT.boot_data                            4 bytes
0x0414         IVT.self                                 4 bytes
0x0418         IVT.csf                                  4 bytes
0x041C         IVT.reserved2                            4 bytes
0x0420         BootData.start                           4 bytes
0x0424         BootData.length                          4 bytes
0x0428         BootData.plugin                          4 bytes
0x042C         (pad)                                    0xBD4 bytes
0x1000         _start                                   first instruction
0x1000+N       end of code
```

In SDP mode, `uuu` finds the IVT, skips the first `0x400` bytes, and uploads the IVT to its `self` address. In SD-card mode, we write the whole file at byte 0 of the card. The `0x400` bytes of padding in the file place the IVT at LBA 2, where the ROM expects it.

For this primary SD boot layout, the file starts at media byte zero and its IVT lands at byte `0x400`. Other boot areas, secondary-image settings and devices need their own placement checks. Do not turn this example into a rule for every eMMC configuration.



The file already supplies the gap before its IVT. Adding another gap at the card would shift the structure away from the expected offset. These are file and media coordinates, not RAM addresses. Block widths are not to scale.

11.2 `mkimx.py`, our own image builder

Save as `~/imx6ull/scripts/mkimx.py`:

```
#!/usr/bin/env python3
"""
mkimx.py -- build an i.MX6ULL boot image from a flat binary.

The output is a file that:
  - Has a 0x400-byte leading pad before the primary SD IVT.
  - Then an IVT at file offset 0x400 (= byte 1024).
  - Then BootData immediately after the IVT.
  - Then padding to offset 0x1000.
  - Then the user binary at file offset 0x1000.

Usage:
  mkimx.py <input.bin> <output.imx>
"""
import argparse
import struct
from pathlib import Path

IVT_TAG       = 0xD1
IVT_LENGTH    = 0x0020   # 32 bytes, big-endian per spec
IVT_VERSION   = 0x40

IVT_OFFSET    = 0x400    # IVT offset from the beginning of the file
```

```
CODE_OFFSET   = 0x1000   # user binary offset from the beginning of the file

def ivt_header():
    # The IVT header is 4 bytes:
    #   byte 0 = tag (0xD1)
    #   byte 1-2 = length (BIG endian, 16-bit)
    #   byte 3 = version (0x40 = HAB v4)
    return struct.pack('>BHB', IVT_TAG, IVT_LENGTH, IVT_VERSION)

def build(input_bin: str, output_imx: str, image_start: int):
    with open(input_bin, 'rb') as f:
        code = f.read()

    if Path(input_bin).resolve() == Path(output_imx).resolve():
        raise ValueError('Input and output paths must differ')
    if image_start < 0x00907000 or image_start % 32:
        raise ValueError('Use a 32-byte-aligned base in the free OCRAM window')
    if not code or image_start + CODE_OFFSET + len(code) > 0x00918000:
        raise ValueError('Payload must be nonempty and fit ROM-active OCRAM')

    ivt_addr        = image_start + IVT_OFFSET
    bootdata_addr   = ivt_addr + 0x20
    entry_addr      = image_start + CODE_OFFSET
    csf_addr        = 0
    dcd_addr        = 0
    image_size      = CODE_OFFSET + len(code)

    # IVT: header (4) + 7 little-endian words = 32 bytes
    ivt  = ivt_header()
    ivt += struct.pack('<IIIIIII',
        entry_addr,
        0,                  # reserved1
        dcd_addr,
        bootdata_addr,
        ivt_addr,           # self
        csf_addr,
        0)                  # reserved2

    assert len(ivt) == 0x20, len(ivt)

    # BootData: start, length, plugin
    bootdata = struct.pack('<III', image_start, image_size, 0)
    assert len(bootdata) == 12

    # Lay out the complete file header, including the leading 0x400 bytes.
    header = bytearray(CODE_OFFSET)
    header[IVT_OFFSET:IVT_OFFSET + 0x20] = ivt
    header[IVT_OFFSET + 0x20:IVT_OFFSET + 0x2C] = bootdata

    out = bytes(header) + code

    # UUU uploads the IVT at ivt_addr. The code must then land at entry_addr.
    loaded_code_addr = ivt_addr + (CODE_OFFSET - IVT_OFFSET)
    assert loaded_code_addr == entry_addr

    with open(output_imx, 'wb') as f:
        f.write(out)

    print(f"  image  = 0x{image_start:08X}")
```

```
    print(f"  entry  = 0x{entry_addr:08X}")
    print(f"  IVT    @ 0x{ivt_addr:08X}  (file offset 0x{IVT_OFFSET:04X})")
    print(f"  bdata  @ 0x{bootdata_addr:08X}")
    print(f"  code   @ 0x{entry_addr:08X}  (file offset 0x{CODE_OFFSET:04X})")
    print(f"  total  = {len(out)} bytes")
    print(f"  wrote  {output_imx}")

def main():
    ap = argparse.ArgumentParser()
    ap.add_argument('input')
    ap.add_argument('output')
    ap.add_argument('--image-start', type=lambda s: int(s, 0),
                    default=0x00907000,
                    help='RAM address corresponding to file offset 0')
    args = ap.parse_args()
    try:
        build(args.input, args.output, args.image_start)
    except (OSError, ValueError) as exc:
        ap.error(str(exc))

if __name__ == '__main__':
    main()
```

Run the script with `python3`, so it needs no executable-permission change. It builds the same simple layout as Chapter 9, with input and output filenames supplied explicitly. DCD, DDR-resident images and signatures are outside this tool's scope.

- **The length field in the IVT header is big-endian.** Everything else in the IVT is little-endian. The `struct.pack('>BHB', ...)` line handles this difference. This is the most common "I wrote my own mkimage and the ROM rejects it" bug.
- **`BootData.start` is `0x00907000`.** File offset `0` corresponds to that address. The IVT at file offset `0x400` therefore corresponds to `0x00907400`.
- **`BootData.length` includes all bytes through the end of the code.** It is not only the code size. If you forget the `CODE_OFFSET` part, the ROM stops loading before your `.text` begins.
- **`csf_addr = 0`** says this image provides no Command Sequence File for HAB authentication. It does **not** disable the device's security policy or bypass authentication on a closed device. This unsigned example depends on an open development configuration. Do not program fuses to make it run. Chapter 124 covers signed boot.

11.3 Building and inspecting

Rebuild Chapter 10's LED:

```
$ cd ~/imx6ull/src/ch10-c-startup
$ make
$ python3 ~/imx6ull/scripts/mkimx.py led.bin led.imx
  image  = 0x00907000
```

```
  entry  = 0x00908000
  IVT    @ 0x00907400  (file offset 0x0400)
  bdata  @ 0x00907420
  code   @ 0x00908000  (file offset 0x1000)
  total  = ... bytes
  wrote  led.imx
```

Source `~/imx6ull/scripts/env.sh` first in a new build terminal. The total size depends on your `led.bin`. The address lines must match this default layout. The builder keeps its offsets consistent, but it cannot discover how the input binary was linked. Check the ELF entry and first instruction as well. In Chapter 10, that instruction is the vector-slot branch to `_start`.

Verify the IVT with raw `xxd`:

```
$ xxd -s 0x400 -l 32 led.imx
00000400: d100 2040 0080 9000 0000 0000 0000 0000  .. @............
00000410: 2074 9000 0074 9000 0000 0000 0000 0000   t...t..........
$ xxd -s 0x420 -l 4 led.imx
00000420: 0070 9000                                .p..
```

Decode:

Word	Bytes (LE)	Value	Field
0x400	D1 00 20 40	tag/length/version (big-endian length!)	IVT header
0x404	00 80 90 00	0x00908000	entry ✓
0x408	00 00 00 00	0	reserved1
0x40C	00 00 00 00	0	dcd (none)
0x410	20 74 90 00	0x00907420	boot_data ✓
0x414	00 74 90 00	0x00907400	self ✓
0x418	00 00 00 00	0	csf
0x41C	00 00 00 00	0	reserved2

Word	Bytes (LE)	Value	Field
0x420	00 70 90 00	0x00907000	BootData.start ✓
0x424	Varies	Complete file size	BootData.length
0x428	00 00 00 00	0	BootData.plugin

`BootData.length` must equal the complete file size. It covers the leading pad, IVT, BootData, padding, and program code.

11.4 Path A, SDP push, again

Push to the board exactly as in Chapter 9:

```
$ sudo "$IMX6ULL_HOME/build/mfgtools/uuu/uuu" led.imx
1:18    1/ 1 [Done                                ] SDP: boot -f led.imx
```

If the LED blinks, record that observation as an SDP execution check for your board. A valid-looking header alone would only prove the file structure.

11.5 Path B, SD card boot, the real thing

> **Storage safety:** Before any command that names /dev/sdX, run lsblk -o NAME,SIZE,MODEL,TRAN,TYPE,MOUNTPOINTS. Verify the removable card using insertion/removal, capacity and available identity fields. A USB reader's model or serial may identify the reader, not the card. Unmount its partitions and stop if the path is uncertain. Writing the wrong device can destroy the host disk.

The SD path changes where the ROM obtains the image, not the payload's C startup contract.

On the i.MX6ULL with `BOOT_CFG` set for SD card, the ROM looks for an IVT at **LBA 2**, which is byte offset `0x400`. Our `.imx` file already contains `0x400` bytes before its IVT. Therefore the file must be written starting at byte 0 of the card.

Use Chapter 8's manual identification flow, not a disk-letter filter. Run `lsblk -o NAME,PATH,SIZE,MODEL,SERIAL,TRAN,RM,TYPE,MOUNTPOINTS` before and after insertion. Match the spare card by identity and capacity. Reject host, system and data disks. Recheck

every time: `/dev/sdc` below is only an example. Unmount each mounted card partition with its actual path, then inspect `lsblk` again. Stop if any identity or mountpoint is uncertain.

From the directory containing the built `led.imx`, check the input with `ls -l led.imx`, then write from byte zero to the **identified whole card**:

```
$ sudo dd if=led.imx of=/dev/sdc bs=1M conv=fsync status=progress
$ sync
```

Do not add `seek=1`. The image already includes the first 1 KB. Adding another 1 KB would move the IVT to byte offset `0x800`, and the ROM would not find it.

After flushing, remove/reinsert the card and identify it again. Compare exactly the image's byte count, substituting the newly confirmed device path:

```
$ sudo cmp -n "$(stat -c %s led.imx)" led.imx /dev/sdc
```

`stat -c %s` prints the file size; `cmp -n` compares that many bytes. Success is silent with exit status zero. A difference/read error means stop and investigate. This tiny raw image creates no filesystem or Linux partitions, so a partition listing is not its verification. Readback proves stored bytes, not ROM execution. Use a dedicated spare card; this layout is not a generic GPT-compatible image.

Before installing the card, follow Chapter 8's power-down procedure and leave the board unpowered:

1. Eject the SD card.
2. Insert into the board.
3. Set boot-mode switch to **SD**.
4. Power on.
5. Watch the LED.

If it blinks, you have booted an i.MX6ULL from an SD card you produced byte by byte. No U-Boot, no `mkimage`, and no Yocto were involved.

11.6 Key Notes

- File offsets, media offsets and RAM addresses are different coordinates. In this primary SD layout, writing the already-padded file at media byte zero places the IVT at `0x400`. An extra seek would shift it out of place.

- SDP omits the leading file padding and loads the IVT at `IVT.self`. The offset difference from IVT to payload must still land the first instruction at the linked entry address.
- The IVT header length is big-endian while its address words are little-endian. `D1 20 00 40` encodes the wrong length even if the remaining fields look plausible.
- `BootData.length` covers the complete image, not only `.text`. The builder owns the wrapper layout but cannot infer the input binary's link address.
- An entry mismatch is a file-consistency defect, not a useful execution experiment. Intentionally corrupt images must not be uploaded to branch into unintended instructions.
- A valid header, successful USB transaction, card readback and payload execution establish different properties. None substitutes for the others, and this OCRAM-only builder supplies neither DDR setup nor signed-boot authentication.

11.7 Pitfalls

- **Endianness of IVT header length.** Big-endian. The rest of the IVT is little-endian. Easy to miss.
- **`BootData.length` shorter than the file.** Tail bytes are not loaded. We always set it to "everything from start of image to end of code, including the 4 KB header gap."
- **Changing one offset without changing the address calculation.** Keep the assertion in the script. It catches a code-location and entry mismatch before the image reaches the board.
- **Writing to the wrong block device.** Repeat Chapter 8's manual identification and mount checks at the moment of writing. There is no helper that turns an uncertain device into a safe target.
- **`sync` forgotten after `dd`.** Linux's page cache is fast. A "complete" `dd` may still have a buffer in RAM. Always `sync` (or `dd conv=fsync`) before pulling the card.
- **Booting the same SD card on a different SoC.** This image is i.MX6ULL-specific. Reusing it on another i.MX6 variant may or may not work. The IVT is the same format but load addresses change. Build per board.

11.8 Going deeper

- **IMX6ULLRM Chapter 8, Section 8.7**: program-image structures, including IVT and BootData. Section 8.1 describes open-versus-closed HAB behavior.
- **U-Boot source: `tools/imximage.c`**: the reference C implementation. Compare against `mkimx.py`. You'll see we covered the simple case correctly.

- **NXP `imx-mkimage` source**: `<https://github.com/nxp-imx/imx-mkimage>`. Its newer-SoC image recipes are not the i.MX6ULL image format taught here. Chapter 22 does not require a TF-A stage for this Cortex-A7 board.
- **`uuu` script reference**: `man uuu.1` or the README in `mfgtools`. Especially the SDP commands list.

The two boot paths can lead to the same LED payload through different loading mechanisms. Chapter 12 adds UART text, so a program can report which step it reached rather than leaving everything to a blink.

Chapter 12: UART driver and `printf`

An LED can tell us that a program reached a checkpoint. It is less helpful when we want the value of a register or the name of the next step. We could invent a blink code, but the board already has a better route to the host: UART1 and its USB-to-serial bridge.

The design starts with character transport, then strings, then a small formatter in front of that transport. Keeping those jobs separate means that a garbled number does not immediately send us back to the pinmux table. The examples explain each layer without requiring a completed serial session.

12.1 Which UART, and on which pins

The i.MX6ULL has eight UART controllers, UART1 through UART8. The Point Atom MINI uses **UART1** as its debug console. The UART1 TX and RX pads are connected to the board's integrated USB-to-TTL bridge, which then connects to the host through the **USB-TTL** or **DEBUG USB** port described in Chapter 8.

The signal path is:

```
i.MX6ULL UART1 pads -> onboard USB-TTL bridge -> debug USB connector -> host serial
device
```

No external CP2102, CH340, FTDI adapter, or jumper wiring is needed. The bridge chip may itself be a CH340 or similar device, but it is already installed on the board.

For our purposes:

- Module: **UART1**, base address `0x02020000`.
- Pads: `UART1_TX_DATA` (ALT0 = UART1_TX_DATA), `UART1_RX_DATA` (ALT0 = UART1_RX_DATA).
- Pad IOMUXC registers: `IOMUXC_SW_MUX_CTL_PAD_UART1_TX_DATA` at `0x020E0084`, RX at `0x020E0088`. (Verify against your RM.)
- Daisy-chain register: `IOMUXC_UART1_RX_DATA_SELECT_INPUT` at `0x020E0624`. For the `UART1_RX_DATA` pad used here, the required daisy value is `3`.

Why RX needs a daisy-chain register

Many i.MX6ULL peripheral inputs can arrive through more than one package pad. UART1 RX is one example. Several pads have an alternate function that can feed the UART1 receiver. The UART therefore needs two separate selections:

1. **Pad MUX selection:** choose which function the physical pad performs.

2. **Peripheral input selection:** choose which eligible pad the UART1 receiver listens to.

The second selection is the **daisy-chain register**. Despite the name, signals are not passed through devices in a serial chain. It is an input multiplexer inside the SoC:

```
candidate pad A ----\
candidate pad B -----+--> SELECT_INPUT mux --> UART1 RX logic
UART1_RX_DATA pad ---/
                        ^
                        daisy value selects one path
```

For our board, both settings are required:

```
REG(IOMUX_MUX_RX) = 0;  /* ALT0: this pad performs UART1_RX_DATA */
REG(IOMUX_DAISY)  = 3;  /* UART1 listens to this pad's input path */
```

These two numbers belong to different registers. Daisy value `3` does **not** mean ALT3. The pad uses **ALT0**, while the UART input selector uses **candidate 3**.

The upstream Linux pin-function header describes the same route with this five-value tuple:

```
MX6UL_PAD_UART1_RX_DATA__UART1_DCE_RX  0x0088 0x0314 0x0624 0 3
```

The tuple means:

Value	Meaning
`0x0088`	MUX_CTL register offset
`0x0314`	PAD_CTL register offset
`0x0624`	SELECT_INPUT register offset
`0`	pad mux mode, ALT0
`3`	input daisy value

TX usually does not need this extra selection. Once the TX pad is muxed to UART1 TX, the UART drives that pad outward. RX travels inward, so the SoC must know which possible pad to connect to the receiver.

If the daisy value is wrong, TX can still print correctly while RX receives nothing. A voltage transition may reach the physical RX pad, but the UART receiver is connected internally to a different candidate path.

The UART1 gate is **CCM_CCGR5**, bits 24-25 (CG12). The baud calculation below needs an **80 MHz** module clock. Our driver explicitly selects PLL3's divided-by-six path and a root divider of one. It assumes the normal direct ROM boot has supplied a running 480 MHz PLL3. It is not a clock recipe for an arbitrary previous program's state. Chapter 13 follows the wider clock tree.

12.2 Baud rate, the i.MX way

Most UART chips compute baud as `f_in / (16 × divisor)`. I.MX is the same shape but with two divisor stages, so it can hit awkward baud rates:

```
baud = (f_uart_clk / 16) × (UBIR + 1) / (UBMR + 1)
```

- `UBIR` is a 16-bit numerator register (Baud Rate Numerator).
- `UBMR` is a 16-bit denominator register (Baud Rate Modulator).
- The factor of 16 is the oversampling rate, fixed.
- `UFCR.RFDIV` is an encoded integer divider between the module clock and the reference clock used in this formula. We choose divide by one. Its field value is `5`, not `1`.

For our case:

- `f_uart_clk = 80 MHz`
- Target baud = 115200
- We want `(UBIR + 1) / (UBMR + 1) = 115200 × 16 / 80 000 000 = 0.02304`

Multiply the ratio by 25,000: the numerator becomes 576. Both numbers fit the 16-bit registers, giving an exact nominal ratio:

- `UBIR = 575`
- `UBMR = 24999`

The calculation is `80,000,000 / 16 * 576 / 25,000 = 115,200`. There is no nominal divider error in this choice. The actual clock can still have oscillator error. With other clock/baud combinations, integer register values introduce an error that must be evaluated rather than assuming the UART will correct it. Receiver tolerance depends on both ends and the frame format.

12.3 Register map (the ones we actually use)

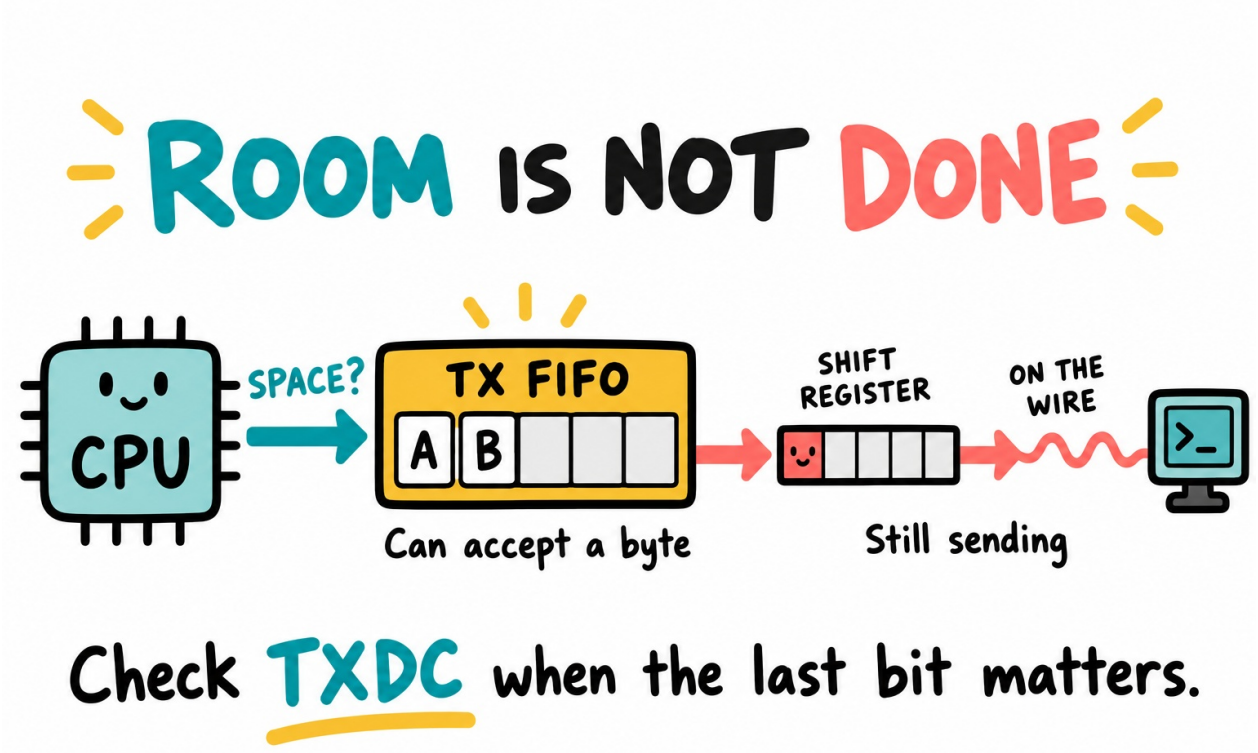
This is the working register map for the driver. Keep it beside the listing:

Register	Offset	Purpose
URXD	+0x000	Receive data (read)
UTXD	+0x040	Transmit data (write)
UCR1	+0x080	Control 1 (enable)
UCR2	+0x084	Control 2 (TX/RX/8N1)
UCR3	+0x088	Control 3 (various)
UCR4	+0x08C	Control 4 (DMA off, RX threshold)
UFCR	+0x090	FIFO control + clock div
USR1	+0x094	Status 1 (TRDY = FIFO at/below its configured TX threshold)
USR2	+0x098	Status 2 (TXDC = TX complete, RDR = RX data ready)
UESC	+0x09C	Escape character (we ignore)
UTIM	+0x0A0	Escape timer (we ignore)
UBIR	+0x0A4	Baud numerator
UBMR	+0x0A8	Baud denominator
UTS	+0x0B4	UART test register. The TX FIFO full bit lives here.

The full list is RM Table 55-3. We will not visit most of them.

The main control and status bits are:

- **`UCR1.UARTEN`** (bit 0), overall UART enable.
- **`UCR2.SRST`** (bit 0), software reset. Write zero to request reset, then wait for hardware to set it again. Writing one does not manually release it.
- **`UCR2.TXEN`** / **`UCR2.RXEN`** (bits 2 / 1), transmitter and receiver enables.
- **`USR1.TRDY`** (bit 13), the transmit FIFO has reached its configured refill threshold. It is not the same test as `UTS.TXFULL` (bit 4).
- **`USR2.RDR`** (bit 0), receive data ready.



Room in the FIFO lets us queue another byte. It does not say the last byte has left the pin. `UTS.TXFULL` answers the space question. `USR2.TXDC` answers the completion question.

12.4 The driver, top to bottom

Copy the four Chapter 10 project files into `~/imx6ull/src/ch12-uart-printf` using your editor or file manager. Keep a working copy of Chapter 10. Save `uart.h`, `uart.c` and `mini_printf.c` beside them, then replace `main.c` with Section 12.6's listing. Add `uart.o` `mini_printf.o` to `OBJS` in the copied Makefile. Its `-lgcc` supplies the integer-division helpers used by the formatter.

`uart.h`:

```
#ifndef UART_H
#define UART_H

#include <stdint.h>

void uart_init(void);
void uart_putc(char c);
void uart_puts(const char *s);
int  uart_getc(void);     /* -1 if no data */

#endif
```

uart.c:

```
#include "uart.h"

#define REG(addr) (*(volatile uint32_t *)(addr))

#define UART1_BASE   0x02020000
#define UART_URXD    (UART1_BASE + 0x000)
#define UART_UTXD    (UART1_BASE + 0x040)
#define UART_UCR1    (UART1_BASE + 0x080)
#define UART_UCR2    (UART1_BASE + 0x084)
#define UART_UCR3    (UART1_BASE + 0x088)
#define UART_UCR4    (UART1_BASE + 0x08C)
#define UART_UFCR    (UART1_BASE + 0x090)
#define UART_USR1    (UART1_BASE + 0x094)
#define UART_USR2    (UART1_BASE + 0x098)
#define UART_UBIR    (UART1_BASE + 0x0A4)
#define UART_UBMR    (UART1_BASE + 0x0A8)
#define UART_UTS     (UART1_BASE + 0x0B4)

#define CCM_CCGR5    0x020C407C
#define CCM_CSCDR1   0x020C4024
#define IOMUX_MUX_TX 0x020E0084
#define IOMUX_MUX_RX 0x020E0088
#define IOMUX_PAD_TX 0x020E0310
#define IOMUX_PAD_RX 0x020E0314
#define IOMUX_DAISY  0x020E0624

#define UTS_TXFULL   (1u << 4)
#define USR2_RDR     (1u << 0)

void uart_init(void)
{
    /* 1.  Gate the clock to UART1.  CG12 (bits 24-25) of CCGR5 = 0b11. */
    REG(CCM_CCGR5) |= (3u << 24);
    REG(UART_UCR1) = 0;
    /* Shared UART root: PLL3/6, divider 1. No other UART is in use here. */
    REG(CCM_CSCDR1) &= ~0x7Fu;

    /* 2.  Pinmux: ALT0 on both TX and RX pads. */
    REG(IOMUX_MUX_TX) = 0;
    REG(IOMUX_MUX_RX) = 0;
    REG(IOMUX_PAD_TX) = 0x000010B0;   /* push-pull, keeper enabled */
    REG(IOMUX_PAD_RX) = 0x000130B1;   /* pull enabled, fast slew */
    REG(IOMUX_DAISY)  = 3;            /* select UART1_RX_DATA input path */
```

```
    /* 3.  Soft-reset the UART (SRST is active-low: clear to assert). */
    REG(UART_UCR2) = 0;
    while (!(REG(UART_UCR2) & 1)) { /* hardware completes reset */ }

    /* 4.  Disable while configuring. */
    REG(UART_UCR1) = 0;

    /* 5. No hardware flow control; 8N1; RX+TX enable. */
    REG(UART_UCR2) = (1u << 14)   /* IRTS (ignore RTS) */
                   | (1u << 5)    /* WS = 8 data bits */
                   | (1u << 2)    /* TXEN */
                   | (1u << 1)    /* RXEN */
                   | (1u << 0);   /* writing 1 leaves SRST unchanged */

    /* 6. RM 55.15.5 requires RXDMUXSEL for this chip's muxed input. */
    REG(UART_UCR3) = (1u << 2);

    /* 7.  No DMA, no escape detection. */
    REG(UART_UCR4) = 0;           /* keep receive-ready and other IRQs disabled */

    /* 8.  FIFO control: RX trigger = 1, TX trigger = 2, RFDIV = /1.
          UFCR fields:
            RXTL  bits  5:0     RX FIFO trigger level
            RFDIV bits  9:7     reference freq divider (0b101 = /1 on i.MX6ULL)
            TXTL  bits 15:10    TX FIFO trigger level
    */
    REG(UART_UFCR) = (2u << 10)         /* TXTL = 2 */
                   | (5u << 7)          /* RFDIV = /1 */
                   | (1u << 0);         /* RXTL = 1 */

    /* 9. 115200 from an 80 MHz reference. Write UBIR before UBMR. */
    REG(UART_UBIR) = 575;
    REG(UART_UBMR) = 24999;

    /* 10. Enable UART. */
    REG(UART_UCR1) = (1u << 0);    /* UARTEN */
}

void uart_putc(char c)
{
    while (REG(UART_UTS) & UTS_TXFULL) { /* spin while FIFO is full */ }
    REG(UART_UTXD) = (uint8_t)c;
}

void uart_puts(const char *s)
{
    while (*s) {
        if (*s == '\n') uart_putc('\r');
        uart_putc(*s++);
    }
}

int uart_getc(void)
{
    if (!(REG(UART_USR2) & USR2_RDR)) return -1;
    return (int)(REG(UART_URXD) & 0xFF);
}
```

Follow the initialization in groups: clock and pads, controller reset, frame settings, baud, then enable. Three details explain common half-working results:

- **Write UBIR before UBMR.** RM 55.5 requires both writes in that order to update the rate multiplier. Reversing them does not produce one predictable percentage error.
- **The `\n` to `\r\n` translation in `uart_puts`** supplies both a new line and a return to column zero without depending on a particular terminal mapping.
- **`UCR3.RXDMUXSEL = 1`** is a requirement in the register description, not an unexplained erratum workaround. It is separate from the IOMUX daisy value.

This small driver has blocking transmit/reset waits, no timeout and no receive-error reporting. A stopped reference clock can leave it waiting forever. It illustrates an early console, not a fault-tolerant production interface.

12.5 A small `printf`

The formatter is deliberately limited so we can follow a number from its digits to `uart_putc()`. A maintained formatter with appropriate tests is a better starting point for a product. This one accepts only the formats listed below and returns zero rather than a standard `printf` character count.

`mini_printf.c`:

```
#include <stdarg.h>
#include <stdint.h>
#include "uart.h"

static void emit_char(char c)            { uart_putc(c); }
static void emit_str(const char *s)
{
    if (!s) s = "(null)";
    while (*s) emit_char(*s++);
}

static void emit_uint(unsigned long v, unsigned base, int width, char pad,
                      int negative)
{
    char buf[sizeof(v) * 8];
    const char *digits = "0123456789abcdef";
    int i = 0;
    if (v == 0) buf[i++] = '0';
    while (v) { buf[i++] = digits[v % base]; v /= base; }
    int padding = width - i - negative;
    if (pad == ' ') while (padding-- > 0) emit_char(' ');
    if (negative) emit_char('-');
    if (pad == '0') while (padding-- > 0) emit_char('0');
    while (i--) emit_char(buf[i]);
}

static void emit_int(int v, int width, char pad)
```

```
{
    unsigned magnitude = v < 0 ? 0u - (unsigned)v : (unsigned)v;
    emit_uint(magnitude, 10, width, pad, v < 0);
}

int mini_vprintf(const char *fmt, va_list ap)
{
    while (*fmt) {
        if (*fmt != '%') { emit_char(*fmt++); continue; }
        fmt++;                                  /* skip '%' */
        char pad = ' ';
        int  width = 0;
        if (*fmt == '0') { pad = '0'; fmt++; }
        /* Cap field width at 64 without writing padding into the digit buffer. */
        while (*fmt >= '0' && *fmt <= '9') {
            width = width * 10 + (*fmt++ - '0');
            if (width > 64) width = 64;
        }
        if (!*fmt) { emit_char('%'); break; }

        switch (*fmt) {
        case 'c':  emit_char((char)va_arg(ap, int));         break;
        case 's':  emit_str(va_arg(ap, const char *));       break;
        case 'd':  emit_int (va_arg(ap, int),  width, pad);  break;
        case 'u':  emit_uint(va_arg(ap, unsigned), 10, width, pad, 0); break;
        case 'x':  emit_uint(va_arg(ap, unsigned), 16, width, pad, 0); break;
        case 'p':  emit_str("0x"); emit_uint((uintptr_t)va_arg(ap, void*), 16,
                                       sizeof(uintptr_t) * 2, '0', 0); break;
        case '%':  emit_char('%');                           break;
        default:   emit_char('%'); emit_char(*fmt);          break;
        }
        if (*fmt) fmt++;
    }
    return 0;
}

int printf(const char *fmt, ...)
{
    va_list ap;
    va_start(ap, fmt);
    int r = mini_vprintf(fmt, ap);
    va_end(ap);
    return r;
}
```

Features we support: `%c %s %d %u %x %p %%`. Numeric field width up to 64, space/zero padding and negative `%d`. Width is ignored for strings and characters. Padding is emitted separately from the digit buffer, so a wide field cannot overflow that buffer. Unsigned subtraction handles `INT_MIN` without signed-negation overflow.

Features we do **not** support: `%f` (we have no floats in the kernel of this book), `%lld`, `%ll`, locales, precision (`%.5s`), left-justification (`%-5d`), `%n`. Cover them when you need them.

Variadic arguments follow C's default promotions. On this target, an `unsigned char` or `unsigned short` promotes to `int`, because `int` can represent all its values. Cast a small value to `unsigned` when passing it to `%u` or `%x`. Pass an `int` to `%d` and a `void *` to `%p`. Equal type sizes do not make an arbitrary `va_arg` type correct. Keep length modifiers such as `%lu` out of this formatter's calls.

12.6 `main()` that actually says hello

```
#include "uart.h"
int printf(const char *fmt, ...);

int main(void)
{
    uart_init();

    printf("\r\nHello, i.MX6ULL bare-metal world!\r\n");
    printf("CPU running at boot-default clock.\r\n");
    printf("This text travels at 115200 baud.\r\n");
    printf("printf supports %%d=%d %%u=%u %%x=0x%08x %%s=\"%s\" %%c=%c\r\n",
           -42, 0xCAFEu, 0xDEADBEEFu, "ready", 'Z');

    /* Echo loop so you can confirm RX works. */
    printf("\r\nType characters. They will echo back.\r\n> ");
    for (;;) {
        int c = uart_getc();
        if (c >= 0) uart_putc((char)c);
    }
}
```

Keep the board's integrated USB-TTL port connected. In one host terminal, open the serial device found in Chapter 8:

```
$ sudo picocom -b 115200 /dev/ttyUSB0
```

If the bridge appeared as `/dev/ttyACM0`, use that path instead. This connection carries UART text. The separate USB-OTG connection carries SDP commands from `uuu`.

In another host terminal, build and load the image through the board's USB-OTG port:

```
$ . ~/imx6ull/scripts/env.sh
$ cd ~/imx6ull/src/ch12-uart-printf
$ make
$ python3 ~/imx6ull/scripts/mkimx.py led.bin led.imx
$ sudo "$IMX6ULL_HOME/build/mfgtools/uuu/uuu" led.imx
```

In the terminal connected through the integrated USB-TTL bridge:

```
Hello, i.MX6ULL bare-metal world!
CPU running at boot-default clock.
This text travels at 115200 baud.
printf supports %d=-42 %u=51966 %x=0xdeadbeef %s="ready" %c=Z
```

```
Type characters. They will echo back.
> hello
```

A transport-only case consists of `uart_init(); uart_puts("UART only\n");` followed by an infinite loop. It bypasses the formatter, separating transport failures from format handling. The transcript above is representative, not a recorded board result. Terminal local echo must be off before returned characters can serve as evidence of the RX path.

12.7 Why polled UART, not interrupt-driven

We are deliberately using polling. Reasons:

- **No interrupt controller yet.** The GIC will be set up properly in Chapter 15.
- **Polling is sufficient for this small console.** An 8N1 character needs ten bit periods, about 87 microseconds at 115200 baud. The FIFO can accept several bytes sooner, but a long stream is limited by that wire rate. This is not a worst-case bound on our wait loops, especially if the clock stops.
- **Polling shows the status bits directly.** The interrupt version follows the same hardware flow, but the FIFO threshold triggers an ISR.

Chapter 15 uses an interrupt-driven echo to explain how the GIC changes event delivery without changing the UART’s data path.

12.8 Key Notes

- UART1 transport and number formatting are separate interfaces. The integrated USB-TTL bridge carries console bytes, while USB-OTG carries ROM downloader commands.
- RX requires both the pad’s ALT selection and the peripheral’s input daisy selection. Correct TX output does not establish a working RX route.
- Baud derives from the UART reference clock, encoded RFDIV and UBIR/UBMR ratio, not the ARM core rate. An exact nominal ratio still leaves oscillator error and receiver tolerance to consider.
- FIFO space permits another queued byte. It does not mean transmission is complete on the wire, so clock or power transitions need the appropriate completion condition.
- Polling simplifies the early console but occupies the CPU and can wait indefinitely if the clock stops. Without flow control, incoming data can also exceed the receiver’s service capacity.
- The formatter supports a defined subset of formats and requires matching promoted argument types. Unsupported formats are an interface limitation, not a UART fault.

Chip-identity diagnostics use documented OCOTP shadow-register reads under their clock/access requirements. They do not require programming registers or fuse writes.

12.9 Pitfalls

- **Wrong RFDIV in UFCR.** Setting `RFDIV = 0` divides by 6, not 1. Symptom: baud rate is six times too slow. The encoding is: 000=/6, 001=/5, 010=/4, 011=/3, 100=/2, 101=/1. Always `0b101`.
- **Reset never completes.** After requesting reset, wait for hardware to restore `SRST`. Writing one cannot force completion. Check the module clock if the loop never exits.
- **Wrong daisy-chain (SELECT_INPUT).** Symptom: TX works through the onboard bridge, but typed characters do not echo. `UART1_RX_DATA_SELECT_INPUT` must select the pad physically connected to the bridge.
- **Using the wrong USB connector.** The USB-TTL port appears as `/dev/ttyUSBx` or `/dev/ttyACMx` and carries console text. The USB-OTG port appears as the i.MX6ULL SDP device and is used by `uuu`.
- **CRLF versus LF.** Terminal mappings can differ. `uart_puts()` inserts `\r` before `\n`, while this formatter emits characters directly, so its example strings use explicit `\r\n`. Do not insert an extra carriage return in both layers.
- **`printf` with `float`s.** Compiles, runs, and prints wrong output because we never wrote `%f`. Do not pass floats to this `printf`.
- **UBIR after UBMR.** Discussed in §12.4. Write UBIR first.
- **Forgot the CCGR.** If the UART is silent, check the clock gate before debugging the UART registers.

12.10 Going deeper

- **IMX6ULLRM Chapter 55**: UART. Read once cover-to-cover. You'll come back.
- **AN3956**: *Configuring the i.MX UART Module*. Concise. Useful.
- **`mpaland/printf`** at `<https://github.com/mpaland/printf>`, an MIT-licensed formatter to evaluate with your own tests and required format support.
- **The 16550 UART datasheet**: every embedded engineer should read this once. It's the platonic UART.
- **Linux source: `drivers/tty/serial/imx.c`**: the same hardware, the same registers, vastly more sophisticated driver. Read it after Chapter 12 here. You'll recognize every bit.

Text output gives us a way to report the next investigation. In Chapter 13 we trace its clock source and the CPU clock through CCM, the Clock Controller Module, rather than assuming the ROM’s choices suit every peripheral.

Chapter 13: CCM clock tree bring-up

A UART can print perfectly while your clock calculation is wrong. Its baud rate depends on the UART root, not directly on the ARM core. Before changing speed, answer a smaller question: which path supplies each clock right now?

This chapter builds a **read-only clock audit**, then explains what a clock transition requires. Actual frequency/voltage changes remain part of the board-qualified startup sequence. Begin with the handoff state: a ROM, DCD, or loader may have changed the reset configuration.

> **Operating-point prerequisite:** identify the fitted CPU speed grade, supply voltage, temperature range, and board revision using Chapter 5. PLL lock does not authorize 696 or 792 MHz. The `05` marking in the supplied core schematic is a 528 MHz-grade part. Do not raise frequency to discover whether a part survives it.

13.1 What we want to set up

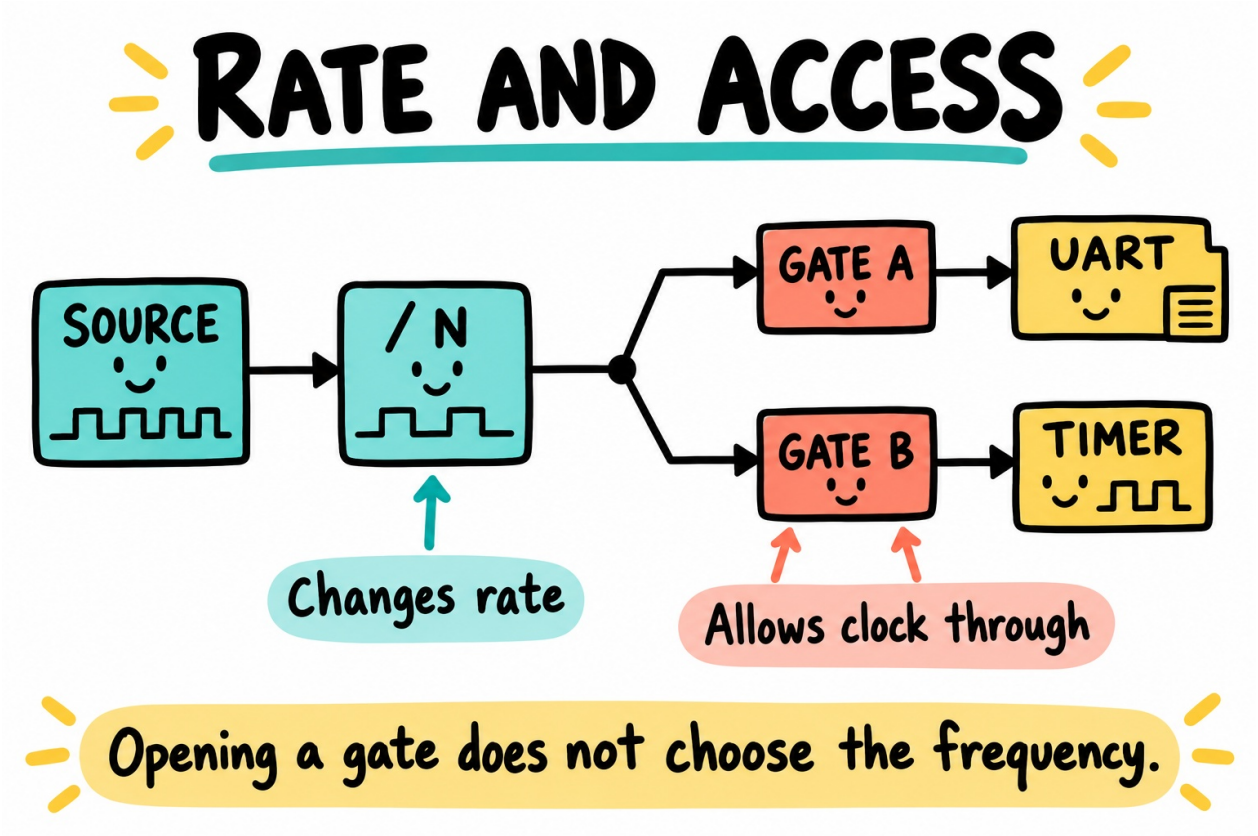
This example plan names the parent and divider behind each domain. It is a worked clock tree, not an instruction to apply these frequencies.

Domain	Example frequency	Conditions needed
ARM core	396 MHz	PLL1 = 792 MHz, ARM divider = 2; allowed operating point
AHB	132 MHz	`periph_clk` = PLL2_PFD2 = 396 MHz, AHB divider = 3
IPG	66 MHz	AHB / 2
AXI	198 MHz	AXI selects `periph_clk`, divider = 2
UART	80 MHz	PLL3 = 480 MHz, fixed /6 path, UART divider = 1
MMDC	396 MHz	**`periph2_clk`** selects PLL2_PFD2, fabric/MMDC divider = 1

Notice the extra `2` in `periph2_clk`. On i.MX6ULL, MMDC is not simply the AHB parent divided again. Sharing PLL2_PFD2 still creates a dependency: changing that PFD can disturb both buses and working DDR.

For a hypothetical, permitted 696 MHz ARM operating point, PLL1 divider 58 and ARM divider 1 give `24 MHz * 58 / 2 = 696 MHz`. That arithmetic says nothing about the regulator setting or safe transition sequence.

13.2 The ANATOP block, PLLs and PFDs



A gate controls whether a clock passes. It does not select its rate. The shared branch is a conceptual example, not the actual UART and timer root topology. Follow each real root in the register map below.

Analog clock registers occupy the region at `0x020C8000`. The supplied reference manual describes them in **Chapter 18**, alongside CCM.

Register	Offset	Role
`CCM_ANALOG_PLL_ARM`	`+0x000`	PLL1
`CCM_ANALOG_PLL_USB1`	`+0x010`	PLL3

Register	Offset	Role
CCM_ANALOG_PLL_USB2	+0x020	PLL7
CCM_ANALOG_PLL_SYS	+0x030	PLL2
CCM_ANALOG_PLL_VIDEO	+0x0A0	PLL5
CCM_ANALOG_PLL_ENET	+0x0E0	PLL6
CCM_ANALOG_PFD_480	+0x0F0	PLL3 PFDs
CCM_ANALOG_PFD_528	+0x100	PLL2 PFDs

Registers that document SET/CLR/TOG aliases have them at +4/+8/+12. Observe each register's reserved-bit rules. An atomic write cannot make an unsafe clock transition safe.

PLL1 (ARM PLL)

Bits	Field	Meaning
6:0	DIV_SELECT	Range 54..108; `f_PLL1 = f_ref * DIV_SELECT / 2`
12	POWERDOWN	1 powers down the PLL
13	ENABLE	Enables output
15:14	BYPASS_CLK_SRC	0 selects the 24 MHz reference
16	BYPASS	Selects bypass instead of PLL output
31	LOCK	Lock status, not operating-point approval

The ARM root also selects PLL1 or `step_clk` in CCSR. Reading only the PLL divider and CACRR misses that mux and PLL bypass.

PLL2 (System PLL)

Normal operation uses 528 MHz. The register also offers 480 MHz: DIV_SELECT=0 selects `20 * f_ref`; 1 selects `22 * f_ref` (528 MHz). Reversing those encodings gives plausible but wrong answers.

A PFD generates `f_PLL * 18 / FRAC`, with FRAC in the documented 12..35 range. For a 528 MHz parent:

PFD	Example FRAC	Calculated frequency
PFD0	27	352 MHz
PFD1	16	594 MHz
PFD2	24	396 MHz
PFD3	32	297 MHz

Each eight-bit PFD lane contains FRAC, a stable indication, and a gate. Before writing FRAC, follow the documented gating sequence and detach or quiesce consumers. An OCRAM function must not casually edit a live DDR source.

PLL3 (USB1 PLL, also feeds peripherals)

Normal operation uses 480 MHz. Its POWER bit has the opposite sense from PLL1/PLL2 POWERDOWN: bit 12 must be **1** for power. PLL3 has PFDs, but the UART mux selects **PLL3 / 6 or the oscillator**, not a PFD. The UART post-divider then divides the selected source.

13.3 The CCM block, root clocks and gates

CCM base is `0x020C4000`.

Register	Offset	Purpose
`CCM_CCR`	`+0x00`	Oscillator control

Register	Offset	Purpose
`CCM_CCSR`	`+0x0C`	ARM step/PLL selection
`CCM_CACRR`	`+0x10`	ARM post-divider
`CCM_CBCDR`	`+0x14`	Bus dividers and final muxes
`CCM_CBCMR`	`+0x18`	Bus parent muxes
`CCM_CSCMR1`	`+0x1C`	PERCLK and other roots, not UART selection
`CCM_CSCDR1`	`+0x24`	UART source/divider, USDHC dividers
`CCM_CDHIPR`	`+0x48`	Clock transition busy flags
`CCM_CCGR0..6`	`+0x68..80`	Peripheral gates

Divider fields usually encode `divisor - 1`. Handshakes have separate bits:

CDHIPR bit	Transition
0	AXI divider
1	AHB divider
2	MMDC divider
3	`periph2_clk` mux
5	`periph_clk` mux

CDHIPR bit	Transition
16	ARM divider

A transition routine must wait for its **own** busy bit, with a failure path. Polling bits 0..2 does not wait for ARM. Some mux changes require consumers to be inactive and gated; a handshake poll does not waive those requirements (RM sections 18.6.6, 18.6.7, 18.6.17).

13.4 The bring-up code

These files are a compilable **audit module**, not a clock initializer. Use Chapter 10's integer-only flags: `-mcpu=cortex-a7 -marm -mfloat-abi=soft -mgeneral-regs-only -ffreestanding`. Use matching architecture flags when linking and link `-lgcc` after objects for the wide division helper. Add `clocks.o` to the existing build.

Preconditions: MMU-off physical access, a 24 MHz reference, privileged execution, and a stable clock tree. Keep IRQs masked during the audit. Zero means *unsupported or unavailable*, not necessarily a stopped clock. This deliberately rejects external PLL references, the secondary ARM step path, and MMDC's audio-PLL path instead of guessing their rates.

`clocks.h`:

```
#ifndef CLOCKS_H
#define CLOCKS_H
#include <stdint.h>
uint32_t clocks_get_arm_hz(void);
uint32_t clocks_get_ahb_hz(void);
uint32_t clocks_get_ipg_hz(void);
uint32_t clocks_get_uart_hz(void);
uint32_t clocks_get_mmdc_hz(void);
#endif
```

`clocks.c`:

```
#include "clocks.h"
#define REG(a) (*(volatile uint32_t *)(uintptr_t)(a))
#define CCM_CCSR    0x020C400Cu
#define CCM_CACRR   0x020C4010u
#define CCM_CBCDR   0x020C4014u
#define CCM_CBCMR   0x020C4018u
#define CCM_CSCDR1  0x020C4024u
#define CCM_CDHIPR  0x020C4048u
#define PLL_ARM    0x020C8000u
#define PLL_USB1   0x020C8010u
#define PLL_SYS    0x020C8030u
#define PFD_528    0x020C8100u
```

```
#define XTAL_HZ    24000000u

static uint32_t pll_hz(uint32_t address)
{
    uint32_t v = REG(address);
    if (address == PLL_USB1 && ((REG(CCM_CCSR) & 1u) || (v & 3u) > 1u))
        return 0; /* unsupported PLL3 test switch/reserved divider */
    if ((v >> 14) & 3u) return 0; /* unsupported reference */
    if (v & (1u << 16)) return XTAL_HZ;
    if (!(v & (1u << 13)) || !(v & (1u << 31))) return 0;
    if (address == PLL_USB1) {
        if (!(v & (1u << 12))) return 0;
    } else if (v & (1u << 12)) {
        return 0;
    }
    if (address == PLL_ARM) {
        uint32_t div = v & 0x7Fu;
        if (div < 54 || div > 108) return 0;
        return XTAL_HZ * div / 2u;
    }
    return XTAL_HZ * ((v & 1u) ? 22u : 20u);
}

static uint32_t pfd_hz(unsigned lane)
{
    uint32_t v = (REG(PFD_528) >> (8u * lane)) & 0xFFu;
    uint32_t frac = v & 0x3Fu;
    uint32_t parent = pll_hz(PLL_SYS);
    if ((v & 0x80u) || !(v & 0x40u) || frac < 12 || frac > 35)
        return 0;
    return (uint32_t)((uint64_t)parent * 18u / frac);
}

static uint32_t periph_hz(void)
{
    uint32_t d = REG(CCM_CBCDR), m = REG(CCM_CBCMR);
    if (d & (1u << 25)) {
        uint32_t sel = (m >> 12) & 3u;
        uint32_t parent = sel == 0 ? pll_hz(PLL_USB1) :
                          sel == 1 ? XTAL_HZ :
                          sel == 2 && !(REG(PLL_SYS) & (3u << 14)) ? XTAL_HZ : 0;
        return parent / (((d >> 27) & 7u) + 1u);
    }
    switch ((m >> 18) & 3u) {
    case 0: return pll_hz(PLL_SYS);
    case 1: return pfd_hz(2);
    case 2: return pfd_hz(0);
    default: return pfd_hz(2) / 2u;
    }
}

uint32_t clocks_get_arm_hz(void)
{
    if (REG(CCM_CDHIPR)) return 0;
    uint32_t s = REG(CCM_CCSR);
    uint32_t parent = (s & (1u << 2)) ?
                      ((s & (1u << 8)) ? 0 : XTAL_HZ) : pll_hz(PLL_ARM);
    return parent / ((REG(CCM_CACRR) & 7u) + 1u);
}
```

```
uint32_t clocks_get_ahb_hz(void)
{
    if (REG(CCM_CDHIPR)) return 0;
    return periph_hz() / (((REG(CCM_CBCDR) >> 10) & 7u) + 1u);
}

uint32_t clocks_get_ipg_hz(void)
{
    return clocks_get_ahb_hz() / (((REG(CCM_CBCDR) >> 8) & 3u) + 1u);
}

uint32_t clocks_get_uart_hz(void)
{
    if (REG(CCM_CDHIPR)) return 0;
    uint32_t v = REG(CCM_CSCDR1);
    uint32_t parent = (v & (1u << 6)) ? XTAL_HZ : pll_hz(PLL_USB1) / 6u;
    return parent / ((v & 0x3Fu) + 1u);
}

uint32_t clocks_get_mmdc_hz(void)
{
    if (REG(CCM_CDHIPR)) return 0;
    uint32_t d = REG(CCM_CBCDR), m = REG(CCM_CBCMR), parent;
    if (d & (1u << 26)) {
        parent = (m & (1u << 20)) ? XTAL_HZ : pll_hz(PLL_USB1);
        parent /= (d & 7u) + 1u;
    } else {
        switch ((m >> 21) & 3u) {
        case 0: parent = pll_hz(PLL_SYS); break;
        case 1: parent = pfd_hz(2); break;
        case 2: parent = pfd_hz(0); break;
        default: return 0; /* audio PLL path not decoded here */
        }
    }
    return parent / (((d >> 3) & 7u) + 1u);
}
```

Each peripheral initializer owns its CCGR gates. For later integration, keep the two APIs separate: these getters inspect clocks; a board-owned startup sequence changes them. This module has no `clocks_init()` entry.

13.5 Verifying with `printf`

This **main-body fragment** reports the audit through the Chapter 12 UART interface. Its use assumes that console’s clock and initialization contract:

```
printf("ARM=%u AHB=%u IPG=%u UART=%u MMDC=%u Hz\r\n",
       clocks_get_arm_hz(), clocks_get_ahb_hz(), clocks_get_ipg_hz(),
       clocks_get_uart_hz(), clocks_get_mmdc_hz());
```

For section 13.1’s example tree, the calculated values would be 396000000, 132000000, 66000000, 80000000, and 396000000. These are **representative calculations**, not captured board output or a promised ROM state.

For the IPG calculation, SYS DIV_SELECT=1, PFD2 FRAC=24, PRE_PERIPH_CLK_SEL=1, PERIPH_CLK_SEL=0, AHB_PODF=2, IPG_PODF=1 give: `24 * 22 * 18 / 24 / 3 / 2 = 66 MHz`. The final mux matters too.

13.6 Verifying with hardware

The decoder calculates frequency from fields. It does not measure the oscillator or establish electrical margin.

Quick check: blink rate

A busy-loop blink is a coarse comparison. Optimization, caches, memory stalls, and peripheral bus speed affect it. Going from 396 to 696 MHz is about **1.76**, not 2; a GPIO loop need not scale by that ratio.

Better check: count cycles with PMU

This **privileged C fragment** enables and resets CCNT and clears its divide-by-64 setting. It does not grant user-mode access. Include `stdint.h`.

```
static inline void pmu_cycles_init(void)
{
    uint32_t v;
    __asm__ volatile ("mrc p15, 0, %0, c9, c12, 0" : "=r"(v));
    v = (v & ~(1u << 3)) | 1u | (1u << 2); /* D=0, E=1, C=1 */
    __asm__ volatile ("mcr p15, 0, %0, c9, c12, 0" :: "r"(v) : "memory");
    v = 1u << 31;
    __asm__ volatile ("mcr p15, 0, %0, c9, c12, 1\n\tisb"
                      :: "r"(v) : "memory");
}

static inline uint32_t pmu_ccnt(void)
{
    uint32_t v;
    __asm__ volatile ("isb\n\tmrc p15, 0, %0, c9, c13, 0"
                      : "=r"(v) :: "memory");
    return v;
}
```

CCNT tells you cycles, **not Hz**. The same cycle count at two operating points says nothing about seconds without a time reference. At 696 MHz a 32-bit counter wraps in about 6.17 seconds. Account for interrupts, read overhead, and security/debug controls. For memory-completion timing, place an appropriate `dsb` before the endpoint; `isb` alone does not drain stores.

Hardware check: scope a GPIO

Use a known, safely routed output and a low-rate waveform. Do not infer CPU cycles from C statement count or try a 70 MHz LED-pin waveform. GPIO read-modify-write travels through the peripheral bus and may be the bottleneck. A timer output or approved CCM clock-output route is more useful if mux, divider, loading, and measurement limits are recorded.

13.7 Why we set up clocks before DDR

Software converts nanoseconds and minimum **DDR CK cycle** requirements into MMDC register encodings. Choosing the clock comes first; the encoded timings describe that choice.

At 396 MHz, `tCK = 2.525... ns`. A hypothetical 15 ns minimum requires `ceil(15 / tCK) = 6` cycles, before applying that field’s encoding and minimum-cycle rules. AHB/IPG govern bus/register access, not this conversion.

Establish the MMDC parent/divider **before** enabling DRAM. A live DDR clock change requires the documented quiesce/self-refresh transition and errata handling, with code, data, stacks, and other masters kept away from DDR as required. It is not an extra write in this audit module.

13.8 Key Notes

- A frequency belongs to a complete selected path: reference, PLL or bypass, PFD where used, muxes and dividers. CCGR gates control availability, not rate.
- The audit getters inspect a stable tree. Board startup owns transitions, voltage sequencing and consumer coordination. A zero from this decoder can mean an unsupported path rather than a stopped clock.
- Shared parents create shared risks. Changing PLL2_PFD2 can affect buses and MMDC even when the ARM core has moved to another source.
- SYS DIV_SELECT=0 with a 24 MHz reference gives 480 MHz. With PFD2 FRAC=24, that becomes 360 MHz rather than the 396 MHz of the 528 MHz parent case.
- PLL lock is not operating-point approval. Safe transitions also depend on speed grade, rail voltage, temperature, consumer state and the correct handshake’s bounded failure path.
- MMDC timings describe a chosen DDR CK. Changing a live DDR clock requires the documented quiesce/self-refresh protocol, not an isolated divider write.

- Register decoding calculates nominal frequency and PMU CCNT counts cycles. Neither establishes elapsed time or electrical margin without an appropriate independent reference or measurement.

13.9 Pitfalls

- **Lock as permission.** Lock does not establish speed grade, voltage, thermal limits, or stable DDR.
- **Reading only a divider.** Mux and bypass may select another parent.
- **Changing a shared PFD.** MMDC can fail even while ARM uses the oscillator.
- **Wrong handshake.** ARM uses CDHIPR bit 16; buses and muxes differ.
- **Wrong PFD formula.** It is `f_PLL * 18 / FRAC`; use a wide intermediate.
- **Undoing a DCD.** DDR may already be initialized. Audit it before choosing which component owns further changes.

13.10 Going deeper

- **IMX6ULLRM Chapter 18**, sections 18.6.4..7, 18.6.17 and 18.7: muxes, handshakes, PLL/PFD fields.
- The fitted part's **i.MX6ULL data sheet**: frequency/voltage/temperature conditions. PLL divider range is not a speed-grade table.
- U-Boot v2025.01 clock.c: compare `get_periph_clk`, `get_mmdc_ch0_clk`, and `get_uart_clk`; note the i.MX6UL/ULL-specific branches.
- Linux v6.12 clk-imx6ul.c: the hardware expressed as clock parents and operations.
- **Cortex-A7 TRM**, performance-monitor chapter: counter controls and access restrictions; this CPU PMU is not the SoC regulator block.

> Next chapter: **Chapter 14: DDR3 initialization with MMDC.** Use the actual clock and fitted geometry to understand a board-specific initialization, without treating copied constants as a working BSP.

Chapter 14: DDR3 initialization with MMDC

A store to OCRAM needs no memory-device initialization. The same store to `0x80000000` depends on an external chip, its power and clock, controller timings, and the board's routing. A correct address is only the beginning.

This chapter traces those dependencies through a board-specific initialization record and a small destructive-test example. The supplied core schematic shows a **single x16 Nanya NT5CC256M16EP-EK on CS0**, nominally 512 MiB. That device is the record's starting point, not confirmation of the part fitted to an uninspected board.

> **Execution boundary:** this chapter does **not** contain a qualified DDR initialization table. Obtain the exact part data sheet, board-matched BSP initialization, and i.MX6ULL-compatible NXP tools before attempting bring-up. Do not substitute illustrative timings or calibration values. If DDR is already initialized by a DCD/loader, do not reinitialize it under running code. You may study the sequence and use a verified loader configuration instead.

14.1 This chapter takes time

Keep the loader, UART diagnostics, constants, stack, and exception vectors in OCRAM while DDR is uncertain. A failed DDR read may hang rather than return a conveniently printable error. Print a checkpoint **before** each new phase; a last message can narrow the failing step.

Retain Chapter 10's map and assertions: ROM-active loaded bytes must end at or below `0x00918000`; post-handoff owned stack space may extend up to `0x00920000` only when it no longer overlaps ROM use. Include every added table and embedded payload in the load-size check.

MMU and D/L2 cache state must be established before destructive memory tests. Do not infer all caches are off from reset: RM section 8.4.4 describes ROM use of I-cache and disabling D/L2/MMU following authentication, not an unconditional I-cache-off handoff.

14.1a RAM/ROM/SRAM/SDRAM/DDR, the lineage

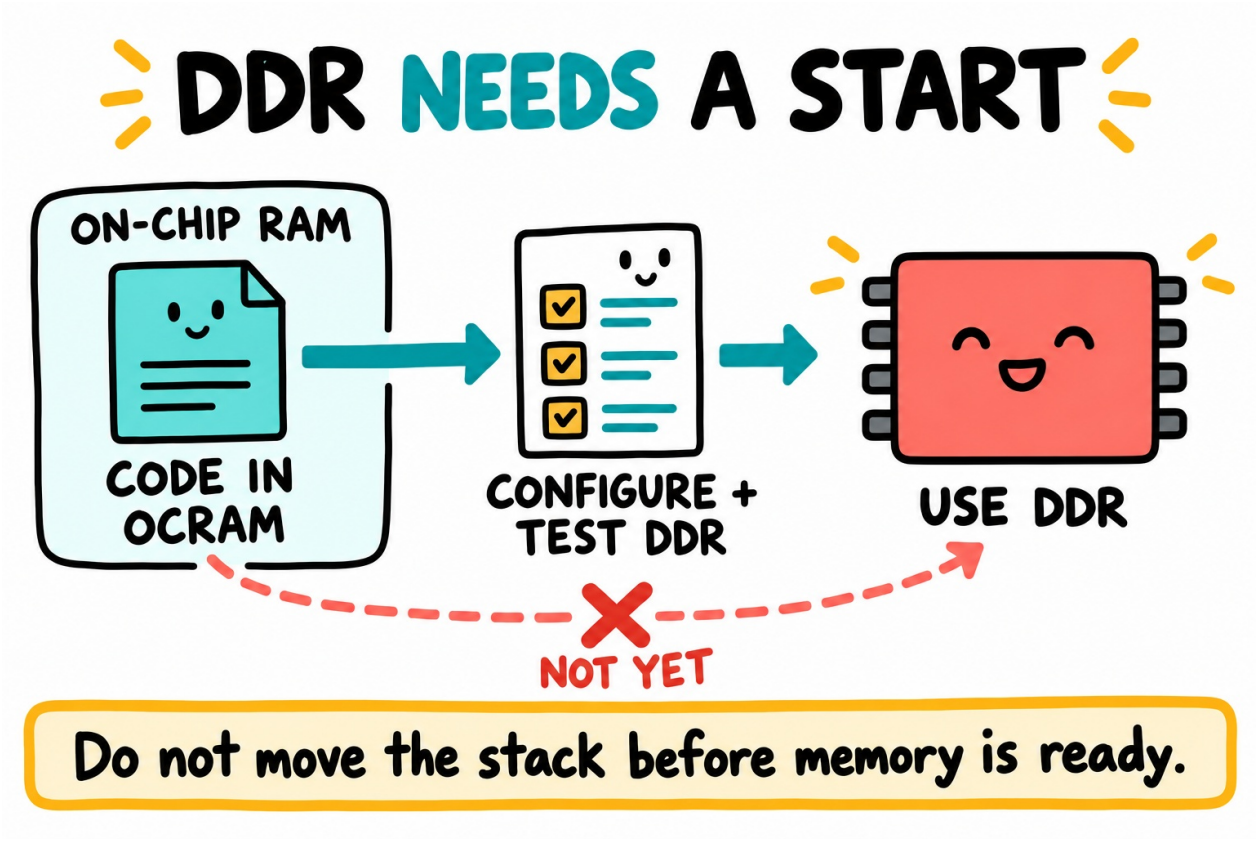
The useful distinction is whether software must keep the memory alive:

- **SRAM** retains data while powered without DRAM-style refresh. OCRAM and CPU caches use SRAM; cost and area limit their capacity.

- **DRAM** stores charge and needs refresh. **SDRAM** synchronizes commands to a clock; DDR is also synchronous DRAM.
- **DDR** transfers data on both clock edges. DDR2/DDR3 increase prefetch depth and change electrical/protocol requirements.
- **DDR3** normally uses 1.5 V; **DDR3L** supports the low-voltage 1.35 V operating specification. Confirm the fitted part and actual rail.
- **LPDDR2** is a different supported protocol. **LPDDR3, DDR4, and DDR5** are not supported by this i.MX6ULL controller configuration.

MMDC supports DDR3/DDR3L x16 and LPDDR2 x16, up to 400 MHz DDR CK (RM 35.1). A 396 MHz CK transfers at **792 MT/s**, not 396 MT/s: `396 million * 2 edges * 2 bytes = 1.584 GB/s` theoretical data-bus peak. Command overhead, refresh, and access patterns reduce usable bandwidth. CK is not the DRAM cell-array clock.

14.2 DDR3, enough detail for bring-up



Keep the code and stack in known memory while investigating the new memory. The arrows show prerequisites, not an automatic copy operation. The DDR icon is generic. The fitted device and its required settings come from the board record.

The controller activates a row in one of eight banks, reads or writes columns in that open row, then precharges it before a different activation. Refresh commands restore charge across the array over a refresh window; one command does not refresh every row at once.

Symbol	Requirement to find in the exact part data sheet
`tRCD`	Activate-to-read/write delay
`tRP`	Precharge-to-activate delay
`tRAS`	Minimum row-active time
`tRC`	Minimum row-cycle time
`tRFC`	Refresh command busy time, dependent on density
`tWR`	Write recovery; also encoded in MR0
`tFAW`	Four-activate window
`CL` / `tAA`	Permitted CAS latency and minimum access time at chosen CK
`CWL`	Permitted CAS write latency at chosen CK; not universally CL-2
`tREFI`	Refresh interval, including temperature-dependent requirements

For a minimum in nanoseconds, start with `cycles = ceil(t_min_ns / tCK_ns)`, then apply minimum-cycle requirements and the particular MMDC field encoding. Not every field is simply N-1.

For example, a **hypothetical** 15 ns minimum at 396 MHz needs six CK cycles. This is arithmetic, not a Nanya timing recommendation. A speed-bin label such as DDR3-1600 describes operation at 1600 MT/s (800 MHz CK); its latency numbers cannot be copied unchanged into a 396 MHz setup.

A timing record connects the data-sheet page, minimum ns, minimum cycles, chosen cycles, MMDC field and encoded value. A separate entry for the matching DRAM mode-register field exposes a controller/DRAM latency mismatch before the first access.

The supplied schematic names NT5CC256M16EP-EK. Obtain its manufacturer data sheet and confirm density, row/column geometry, speed suffix, temperature range, and low-frequency limits. A replacement device needs a fresh timing record, even when the pinout matches.

14.3 Two chips, one bus, channels, ranks, banks

A **channel** is a controller data interface. A **rank** is the set of devices selected together by one chip-select. **Banks** are internal arrays within a DRAM device.

Two x8 devices can form a x16 rank, but the supplied core schematic instead shows **one x16 device** on CS0. Both byte lanes come from that chip.

```
256 M locations * 16 bits = 4096 Mbits = 512 MiB
one x16 device, one rank, eight internal banks
```

For a candidate 15-row-bit, 10-column-bit, eight-bank x16 geometry: `2^15 * 2^10 * 8 * 2 bytes = 512 MiB`. Check those counts in the exact data sheet before encoding MDCTL and MDASP. A 14-row-bit version would have half that capacity. A low-address test cannot distinguish them.

14.4 The MMDC register groups

MMDC base is `0x021B0000`; its register descriptions are in **RM Chapter 35**.

Register/group	Offset	Responsibility
MDCTL	+0x000	Geometry, bus width, chip-select enable
MDPDC	+0x004	Power-down configuration
MDOTC	+0x008	ODT timing
MDCFG0	+0x00C	Includes tRFC, tXS, tXP, tXPDLL, tFAW, tCL
MDCFG1	+0x010	Includes tRCD, tRP, tRC, tRAS, tWR, tMRD, tCWL
MDCFG2	+0x014	Includes tDLLK, tRTP, tWTR, tRRD

Register/group	Offset	Responsibility
MDMISC	+0x018	Memory type and controller options
MDSCR	+0x01C	Configuration request/ acknowledge and commands
MDREF	+0x020	Refresh configuration
MDRWD	+0x02C	Read/write turnaround
MDOR	+0x030	Reset/CKE wait timing
MDASP	+0x040	Chip-select address-space partition
MAARCR	+0x400	AXI reordering
MAPSR	+0x404	Power saving/status
MPZQHWCTRL	+0x800	PHY ZQ calibration
MPWLGCR	+0x808	Write-leveling control
MPWLDECTRL0	+0x80C	Write-leveling delays for byte lanes 0 and 1
MPDGCTRL0	+0x83C	Read DQS gating for the active lanes
MPRDDLCTL	+0x848	Per-byte read delay
MPWRDLCTL	+0x850	Per-byte write delay

Register/group	Offset	Responsibility
MPRDDQBY0DL..1DL	`+0x81C..820`	Active byte-lane read DQ bit-delay controls
MPMUR0	`+0x8B8`	PHY measure-unit control/status

The register map contains descriptions inherited from wider MMDC variants. Mark the two active byte lanes before following those descriptions.

MDSCR.CON_REQ blocks new AXI accesses and drains outstanding work. Poll CON_ACK before permitted configuration/commands, then clear CON_REQ after completion (RM 35.12.8). Command fields in MDSCR issue the DRAM commands; ordinary memory traffic waits while configuration is in progress. Polls need bounded failure handling that remains entirely outside DDR.

14.5 IOMUX before MMDC

DDR pads need the correct electrical configuration before command traffic. The dedicated DDR pad/group registers are in IOMUXC (RM Chapter 32). They select the memory interface’s electrical behavior, separately from the familiar GPIO alternate-function setup.

Record exact addresses and decoded fields for clock, address/control, DQS, DQ byte groups, CKE, ODT, reset, DDR type, and calibration reference settings from the board-matched configuration. Check reserved bits and the actual 1.35 V rail. Keep addresses and decoded fields together so each write can be checked against its register description.

Drive strength, slew, and termination interact with trace impedance and loading. Too much or too little drive can reduce the signal margin. A memtest mismatch is a starting point for diagnosis; waveform and margin measurements answer the electrical questions the schematic leaves open.

14.6 The JEDEC DDR3 initialization sequence

The part data sheet and MMDC procedure together define the actual sequence. For the DDR3 power-up outline:

1. Establish supplies as specified and hold RESET# low for the required power-up interval (at least 200 us for the usual DDR3 sequence).

2. Release RESET# while keeping **CKE low** for the required interval (usually at least 500 us after RESET# rises). Establish a valid clock for the specified time before CKE rises.
3. **Assert CKE high**, issue NOP/deselect, and wait tXPR before MRS.
4. Load MR2, MR3, MR1, then MR0 with DLL reset, observing tMRD and the final tMOD requirement.
5. Issue ZQCL and observe the specified ZQ initialization and DLL-lock waits before normal traffic.
6. Complete the controller/PHY calibration flow, restore operating modes and refresh, and release configuration request.

Follow the CKE rising edge closely: the reset-to-CKE wait happens before it, and tXPR happens afterward. MMDC MDOR and command sequencing implement parts of the procedure; verify their clock basis and encodings. Micron’s DDR3 FAQ also explains reset and ZQ behavior; it is background, not the fitted Nanya part’s timing authority.

Mode Register encodings (relevant fields)

Register	Fields to decode from the fitted part’s data sheet
MR0	BL at A1:A0; split CL field at A6:A4 and A2; DLL reset at A8; discrete tWR encoding at A11:A9
MR1	DLL disable at A0; output impedance at A5/A1; RTT_NOM at A9/A6/A2; write leveling at A7
MR2	CWL at A5:A3; RTT_WR at A10:A9; refresh-related options
MR3	MPR/operating options required during calibration and normal use

Choose supported CL/CWL values for the actual CK, not the maximum-speed bin. Compute DRAM MR values separately from MMDC timing encodings. Then encode the MR address and bank selection into MDSCR as documented. A bare MR value is not an entire MDSCR command.

Each MR’s source, chosen cycles and encoding belong to that same timing record.

14.7 Preparing the board-specific initialization

The timing record, pad configuration and calibration results together define the board’s initialization contract. The following outline names the phases that a board-owned implementation must supply:

```
Verified board-specific initialization:
  establish approved clock/voltage state and MMDC root
  configure exact DDR pads and groups
  enter/confirm MMDC configuration state
  program geometry, partition, timing, reset waits, ODT and PHY controls
  perform reset/CKE/MRS/ZQ sequence with required waits
  apply qualified calibration or run the complete calibration procedure
  restore refresh and normal operating options
  release configuration request; confirm completion
  test only an explicitly reserved, destructive scratch range
```

Start from the matching board BSP and NXP RPA, then check the fitted part and silicon errata. Keep the generated initialization, timing record, tool version, and calibration log together: they describe one configuration.

The test below **is compilable**. It requires already initialized DDR, an aligned, exclusively reserved range, MMU-off access with D-cache/L2 disabled under the startup policy, and all code/data/stacks in OCRAM. It overwrites every word twice. Volatile forces accesses in C; it does not bypass a cache.

`ddr.h`:

```
#ifndef DDR_H
#define DDR_H
#include <stdint.h>
int ddr_init(void); /* board-owned implementation; 0 = completed */
uint32_t ddr_test_words(volatile uint32_t *base, uint32_t words);
#endif
```

`ddr-test.c` (there is deliberately no generic `ddr_init()` definition):

```
#include "ddr.h"

uint32_t ddr_test_words(volatile uint32_t *base, uint32_t words)
{
    uint32_t errors = 0;
    for (uint32_t pass = 0; pass < 2; pass++) {
        uint32_t mask = pass ? 0x5A5A5A5Au : 0xA5A5A5A5u;
        for (uint32_t i = 0; i < words; i++) base[i] = i ^ mask;
        __asm__ volatile ("dsb sy" ::: "memory");
        for (uint32_t i = 0; i < words; i++)
            if (base[i] != (i ^ mask)) errors++;
    }
    return errors;
}
```

Limit the range so the pointer arithmetic is valid and the two-pass error count cannot overflow. The caller supplies the capacity and reserved-region checks. Zero errors mean this range passed these patterns on this run; wider coverage comes in section 14.11.

14.8 Calibration: write leveling, DQS gating, read/write delay

Calibration asks where to sample or drive data **within** a clock period:

- **Write leveling** aligns write DQS with CK at the memory.
- **DQS gating** finds the read-response strobe window.
- **Read/write delay calibration** positions sampling/driving within the data window. The main delay registers operate per byte; separate DQ registers provide bit adjustments.

The hardware calibration procedure controls memory mode, configuration state, refresh, starting delays, completion polls, error checks, and restoration. Follow RM 35.11 and the tool’s matching user guide in order.

Recording the active-lane delays

On the i.MX6ULL x16 interface, MPWLDECTRL0 covers both active byte lanes. The corresponding registers with suffix 1 describe additional lanes in wider layouts. Keep that distinction beside the tool’s output:

Register	Address	Lane meaning
MPWLDECTRL0	`0x021B080C`	Write leveling, bytes 0 and 1
MPWLDECTRL1	`0x021B0810`	Wider-layout bytes 2 and 3, not active x16 bytes
MPDGCTRL0	`0x021B083C`	DQS gating for active x16 lanes
MPDGCTRL1	`0x021B0840`	Wider-layout additional lanes
MPRDDLCTL	`0x021B0848`	Read delay; use active byte fields

Register	Address	Lane meaning
MPWRDLCTL	0x021B0850	Write delay; use active byte fields

Record the active-lane results with board revision, part marking, CK, voltage, temperature, tool version, and test settings. Use the matching i.MX6ULL procedure to decide which fields belong in the initialization.

Qualifying supported operating points

Qualify the chosen configuration at supported operating points across the specified voltage/temperature envelope, using appropriate equipment. For each point, record test coverage, duration, errors, and calibration margin. Stay within both the SoC and DRAM ratings; an overclock result cannot replace this qualification.

Review errata **ERR005778** before DDR operation below 100 MHz (measure-unit workaround must execute outside DDR). **ERR009596** requires MAARCR's ARCR_GUARD bits to remain zero. Include those requirements in the same initialization record.

14.9 Calling from `main()` and the OCRAM → DRAM jump

Integrate this **main-body fragment** with the board-owned `ddr_init()`, working UART/printf, Chapter 10 startup, and the test module. If a DCD/loader already initialized DDR, retain that state and skip the initialization call.

```
/* In main(), while everything still lives in OCRAM: */
printf("Starting board DDR initialization\r\n");
if (ddr_init() != 0) {
    printf("DDR initialization did not complete\r\n");
    for (;;) {}
}
/* Example only: reserve 0x81000000..0x813FFFFF before using it. */
uint32_t errors = ddr_test_words((volatile uint32_t *)0x81000000u,
                                 4u * 1024u * 1024u / 4u);
printf("DDR scratch test: %u mismatches\r\n", errors);
if (errors) for (;;) {}
```

That scratch span is 16 MiB above the base. Confirm it exists and reserve it in the memory map; it stays clear of the separate payload proposed below. Report the tested span and mismatch count rather than total-capacity success.

If a checkpoint disappears, investigate power/reset, CK, pads, geometry, timings, commands, refresh, and calibration. There is no evidence-based ranking of those causes in this chapter.

14.10 Copying ourselves to DRAM

Copying a fixed-address executable to a new base does **not** relocate it. Literal addresses, global references, pointers, and vector entries still refer to the old link addresses. A zero-initialized static flag usually belongs to BSS, which is not stored in the raw payload. Printing `&main` also prints a link-time address, not a measurement of the current PC.

A simpler design uses an OCRAM loader plus a **separately linked DRAM payload**. Each image has its own link addresses and startup. Their ownership boundary is explicit:

- The example payload is linked at `0x80100000`, with vectors first and `ENTRY(_vectors)` as in Chapter 10. Its first ARM instruction branches to its own startup.
- Payload startup masks interrupts, initializes its own SVC/IRQ stacks and data/BSS, installs VBAR with the Chapter 15 policy and enters its own main. It must **not reinitialize DDR**.
- The loader holds the raw payload in its read-only section, bounded by start/end symbols. Those bytes count toward the ROM-active OCRAM load bound, which may exclude a larger two-image bundle.
- The loader tests an unrelated scratch range and copies exactly the payload file length into its linked destination. Both loader and payload regions remain excluded from subsequent destructive tests.
- The handoff synchronizes the new instruction bytes before branching to the payload entry.

Embedding **assembly fragment** in a loader source file:

```
    .section .rodata.dram_payload, "a"
    .balign 4
    .global _payload_start, _payload_end
_payload_start:
    .incbin "dram-payload.bin"
_payload_end:
```

Copy **C fragment**; include `stdint.h`. Its symbols require the assembly and linker integration above:

```
extern const unsigned char _payload_start[], _payload_end[];
extern void enter_dram_payload(uint32_t entry) __attribute__((noreturn));

void copy_and_enter_payload(void)
{
    uintptr_t bytes = (uintptr_t)_payload_end - (uintptr_t)_payload_start;
    volatile unsigned char *dst = (volatile unsigned char *)0x80100000u;
    /* Verify bytes and destination bounds against the payload map first. */
    for (uintptr_t i = 0; i < bytes; i++) dst[i] = _payload_start[i];
```

```
    enter_dram_payload(0x80100000u);
}
```

`enter-dram.S`, callable from that fragment under the explicit **MMU-off, D/L2-disabled, privileged SVC, ARM-state** policy:

```
    .syntax unified
    .cpu cortex-a7
    .arm
    .text
    .global enter_dram_payload
    .type enter_dram_payload, %function
enter_dram_payload:
    cpsid if
    dsb sy
    mov r1, #0
    mcr p15, 0, r1, c7, c5, 0   @ invalidate I-cache to PoU
    mcr p15, 0, r1, c7, c5, 6   @ invalidate branch predictor
    dsb sy
    isb
    bx r0
    .size enter_dram_payload, . - enter_dram_payload
```

I-cache may still be enabled from ROM; synchronization is therefore explicit. If D-cache/L2 is enabled, this stub is **insufficient**: dirty destination data needs the correct clean-to-PoU/coherency procedure first. Do not add a casual cache-disable instruction and discard dirty data.

Use the payload map/disassembly and, when available, a debugger’s current PC to check execution in DRAM. Its stack, globals, and VBAR need separate checks. The later IRQ/timer examples can stay in OCRAM. Their mechanisms do not depend on relocating this payload or completing DDR qualification.

14.11 Sanity tests beyond the basic memtest

All tests are destructive. Keep the tester outside the tested range and disable other masters that might use it.

Walking ones / zeros

A walking-bit test writes each `1u << bit` and its complement to a reserved word, then compares the readback. It checks data-bit behavior at that location, not every cell or address line. All-zero/all-one and alternating patterns at multiple locations extend the pattern coverage.

Address-as-data

An address-as-data test writes distinct address-derived patterns across a reserved span before reading any back. A dedicated power-of-two address-alias test addresses a different coverage gap: a 4 MiB sweep cannot find every high address bit or a wrongly configured 256/512 MiB boundary.

Long memtest

A wider test uses multiple patterns and retention intervals within the available verified capacity, excluding loader, stacks, vectors, logs, payload and tool working areas. A tester running in DRAM cannot overwrite “all 512 MiB” safely. Its result is meaningful only with the range, duration, conditions and first failing address/value.

Linux `memtester` later tests memory allocated to a user process; it is not a bare-metal program or a proof that every physical byte was tested.

14.12 Why the DCD is the elegant alternative

A DCD lets the ROM prepare DDR before loading a payload into it. Chapter 7 describes its write, check/poll, and NOP commands, format limits, and register-validation constraints. General branching and calibration algorithms require executable initialization code.

A validated DCD carries a fixed board configuration. An SPL can instead execute richer initialization code. Choose the boot design first, then express the board’s initialization in the form that design supports. Chapter 11’s `python3 mkimx.py` tool is deliberately OCRAM-only and unsigned; it does not build a DDR image or accept a DCD. The two-image design above still wraps an OCRAM loader. For a ROM-loaded DDR image, follow the actual Chapter 19 image/build route with a qualified DCD-capable configuration.

14.13 NXP’s DDR Stress Tool

Use NXP’s **i.MX 6/7 DDR Stress Test Tool** and the matching **i.MX6UL/ULL/ULZ Register Programming Aid (RPA)**. Select the package by its supported SoCs; similarly named tools for i.MX8M serve a different controller.

The RPA derives an initialization script from the selected part geometry, timings, clock, and board configuration. The stress tool loads a helper, runs calibration/testing, and reports results. Supply the RPA-derived starting configuration before asking the tool to find delay margins. NXP’s tool release page identifies the supported SoCs and separate roles of the two tools.

Before a hardware session, read the matching package guide, check board power/USB/reset prerequisites from Chapter 8, and verify the selected SoC. Archive inputs and results. This chapter records no tool run or board test.

14.14 Key Notes

- DDR availability depends on a matched clock, geometry, electrical setup, timing/command sequence, refresh and calibration. A valid address or a copied delay table establishes none of that configuration by itself.
- MMDC fields and DRAM mode registers encode the same latency choices differently. Minimum nanoseconds, minimum CK cycles and field encodings must agree for the fitted part at the selected operating point.
- Configuration request/acknowledge separates setup from normal AXI traffic. Failure handling must stay in OCRAM, because uncertain DDR cannot safely hold diagnostics, stacks or the recovery path.
- Destructive tests require exclusive scratch ownership and a known cache state. A successful low-address sweep does not establish full capacity, high-address non-aliasing or voltage/temperature margin.
- Calibration positions the active byte lanes within the data window. Its results belong to a particular board, DRAM part, clock and set of conditions, not to every board sharing the SoC.
- A separately linked DRAM payload keeps loading distinct from relocation. It owns its startup, stack, globals and vectors, while the OCRAM loader owns placement and instruction synchronization. IRQ/timer examples need not move.
- This chapter supplies no qualified DDR initialization table or board result. Temperature/voltage qualification requires controlled equipment within ratings. Do not use a hairdryer, hot-air station or DDR overclock as a stress experiment.

14.15 Pitfalls

- **Copied constants without provenance.** Calibration is not a complete clock/pad/timing/refresh configuration.
- **Wrong capacity.** A small test may pass while high addresses alias.
- **Mismatched CL/CWL.** Controller and DRAM have different encodings but must agree on the chosen latencies.
- **Live MMDC reclocking.** Timings are encoded for a particular CK.
- **Leaving CON_REQ asserted.** Normal AXI traffic remains blocked.
- **Cache-only success.** Volatile is not uncached memory.
- **Naive copy-and-call relocation.** Link-time addresses do not move.
- **Blind retries.** Follow the board’s approved reset/power sequence, including source/back-power checks; an arbitrary five-second wait is not proof of a clean reset.

14.16 Going deeper

- The **exact fitted DRAM manufacturer’s data sheet**, including operating ranges, low-frequency limits, reset/MRS/ZQ timings, and MR encodings.
- **IMX6ULLRM Chapter 35**, especially 35.11 and 35.12; Chapter 18 for clocks and Chapter 32 for pads.
- **i.MX6ULL errata**, ERR005778 and ERR009596.
- NXP i.MX 6/7 DDR tools, matching RPA and packaged stress-tool user guide.
- **AN4467, i.MX 6 Series DDR Calibration**: background on the procedures; check applicability alongside the matching device/tool guide.
- Board-matched BSP DCD/SPL: trace each value back to a worksheet or measured result rather than copying a different evaluation board’s memory table.

> Next chapter: **Chapter 15: Exceptions and the GIC.** We can build the IRQ path in OCRAM with a known stack and vector table, independently of the optional DRAM payload.

Chapter 15: Exceptions and the GIC

A received character can be sitting in the UART FIFO while the CPU sleeps. To wake the program, three things must agree: the UART must request an interrupt, the GIC must route it, and the CPU must have a usable vector and stack. We will follow one character through that path.

This is a single-core, non-nested, **Secure SVC/IRQ, Group-0 IRQ** teaching configuration for the i.MX6ULL GIC-400 (GICv2). SVC mode alone does not prove Secure state. Confirm the boot/handoff policy before using these writes; they are not a drop-in Non-secure, Hyp, Monitor, RTOS, or Linux initializer. IRQ routing must enter IRQ mode rather than Monitor/Hyp. The program owns the controller exclusively and begins without active IRQs.

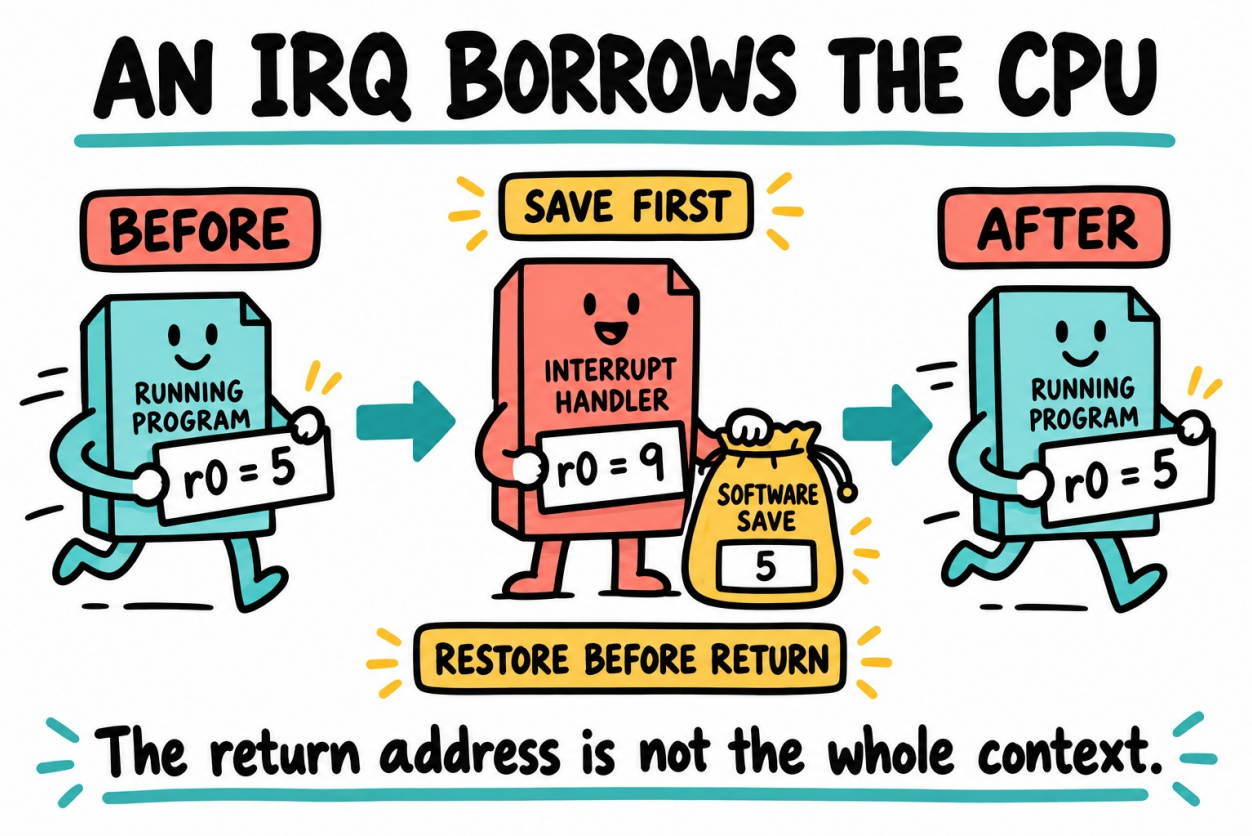
15.1 What is different from Cortex-M

On Cortex-M, exception entry stacks core registers on the interrupted context’s stack; the handler itself uses MSP. The vector slots contain handler addresses (apart from the initial SP), and EXC_RETURN tells hardware how to return.

On Cortex-A7, the hardware saves CPSR into SPSR_irq, banks IRQ SP/LR, and chooses a vector instruction. It does **not** save the shared general registers. Software must preserve them and perform an exception return.

A C ISR is an ordinary function behind our assembly wrapper. The compiler does not know that its caller represents an interrupted program. We must provide a valid AAPCS stack and preserve anything the handler can clobber.

15.2 The exception vector table



Returning to the right instruction is not enough if its registers have changed. The saved pouch represents our assembly wrapper’s work. Cortex-A7 does not automatically stack these shared registers on IRQ entry.

For our ARM-state exception policy, the vector entries are eight instructions, four bytes each, with a 32-byte-aligned base.

Offset	Exception	Cause
`+0x00`	Reset	CPU reset vector; ROM normally owns actual reset
`+0x04`	Undefined instruction	Unsupported instruction
`+0x08`	SVC	`svc` instruction
`+0x0C`	Prefetch abort	Instruction-fetch fault
`+0x10`	Data abort	Data-access fault

Offset	Exception	Cause
+0x14	Reserved	Unused vector slot
+0x18	IRQ	Routed interrupt request
+0x1C	FIQ	Fast interrupt request

A branch or PC-relative load of a handler address is common. This is not the Cortex-M function-pointer table.

With SCTLR.V=0, VBAR supplies the normal vector base; historical low vectors correspond to VBAR=0. SCTLR.V=1 selects high vectors at `0xFFFF0000` instead. The security state selects the corresponding banked VBAR. Hyp and Monitor have separate vector-base mechanisms.

Setting VBAR alone is therefore insufficient. Section 15.4 selects ARM-state, little-endian exceptions and clears high-vector selection. With the MMU off, our VBAR value is a physical address.

15.3 The new vector table

`vectors.S` is a complete assembly module. Its first instruction remains `b _start`, compatible with Chapter 10's `ENTRY(_vectors)` and IVT entry at `0x00908000`. Hardware reset still enters ROM; this first slot also serves our image's software entry.

```
    .syntax unified
    .cpu cortex-a7
    .arm
    .section .vectors, "ax"
    .balign 32
    .global _vectors
    .type _vectors, %function
_vectors:
    b       _start
    ldr     pc, =undef_handler
    ldr     pc, =svc_handler
    ldr     pc, =prefetch_handler
    ldr     pc, =data_handler
    ldr     pc, =unused_handler
    ldr     pc, =irq_entry
    ldr     pc, =fiq_handler
    .size _vectors, . - _vectors
    .ltorg                          @ handler literals follow all eight slots

    .text
undef_handler:
```

```
prefetch_handler:
data_handler:
unused_handler:
svc_handler:
fiq_handler:
    b       .                       @ fail-stop; no C call or stack required

    .global irq_entry
    .type irq_entry, %function
irq_entry:
    sub     lr, lr, #4              @ architectural IRQ resume address
    srsdb   sp!, #0x12              @ {resume PC, SPSR_irq} on IRQ stack
    push    {r0-r12, lr}            @ 56 bytes; total frame = 64 bytes
    bl      c_irq_dispatch          @ stays in IRQ mode, IRQs still masked
    pop     {r0-r12, lr}
    rfeia   sp!                     @ restore PC and CPSR from return frame
    .size irq_entry, . - irq_entry
```

The return frame is eight bytes, followed by 56 bytes of registers. Starting with an aligned IRQ SP, the C call is eight-byte aligned even if the interrupted SVC function’s SP was only four-byte aligned at that point. The SVC stack and banked SVC LR are never touched.

For an IRQ, `LR_irq - 4` is the architectural **resume address**. Do not derive it from the PC value visible in a debugger or describe every IRQ as retrying an instruction. RFE restores both execution address and saved CPSR, including flags, mode, ARM/Thumb state, and the old interrupt mask.

This wrapper saves core registers only. Compile **all reachable ISR code** with `-mcpu=cortex-a7 -marm -mfloat-abi=soft -mgeneral-regs-only`; do not call FP/NEON routines or libraries with a mismatched ABI. IRQ nesting is disabled, FIQs remain masked, and fault handlers halt. There is no scheduler/context switch or general exception recovery here.

15.4 Setting up VBAR and a separate IRQ stack

Integrate this **startup fragment** before any IRQ unmask, after establishing Chapter 10’s MMU/data-cache policy. Enter it in privileged Secure SVC, ARM state; the existing SVC stack must already be initialized.

```
    cpsid   if
    cps     #0x12
    ldr     sp, =_irq_stack_top
    cps     #0x13

    mrc     p15, 0, r0, c1, c0, 0
    bic     r0, r0, #(1 << 13)      @ SCTLR.V=0: use VBAR
    bic     r0, r0, #(1 << 30)      @ SCTLR.TE=0: ARM-state exception entry
    bic     r0, r0, #(1 << 25)      @ SCTLR.EE=0: little-endian exceptions
    mcr     p15, 0, r0, c1, c0, 0
    isb
```

```
    dsb     sy
    mov     r0, #0
    mcr     p15, 0, r0, c7, c5, 0  @ ROM may leave I-cache enabled
    mcr     p15, 0, r0, c7, c5, 6  @ synchronize branch predictor state
    dsb     sy
    ldr     r0, =_vectors
    mcr     p15, 0, r0, c12, c0, 0
    isb
```

This does not disable I-cache or change the other SCTLR controls. If vectors were written through a data cache, clean their bytes to the required point before instruction invalidation. Our integration policy has MMU/D/L2 disabled; do not silently apply it to a different state.

Inside the existing linker `SECTIONS`, retain the first-position `KEEP(*(.vectors))`. Add this **linker fragment** after BSS and outside the existing SVC stack reservation:

```
.irq_stack (NOLOAD) : ALIGN(8) {
    _irq_stack_bottom = .;
    . += 4096;
    _irq_stack_top = .;
} > OCRAM
ASSERT((_irq_stack_top & 7) == 0, "IRQ stack must be 8-byte aligned")
ASSERT(_irq_stack_top <= 0x00920000, "IRQ stack exceeds owned OCRAM")
ASSERT(_irq_stack_top <= _stack_limit, "IRQ stack overlaps SVC stack")
```

Keep the original load-end assertion at `0x00918000`, the SVC stack’s 4 KiB reservation/assertions, and non-overlap checks for **both** stacks. NOLOAD does not allocate more physical RAM. Use the map to verify all reservations; a 4 KiB IRQ stack is a budget, not a measured worst case.

15.5 The GIC v2 distributor + CPU interface

GIC-400 exposes two regions:

- **Distributor**, `0x00A01000`: enable, grouping, target, priority, trigger configuration, and pending/active state.
- **CPU interface**, `0x00A02000`: priority filtering, acknowledge, EOI.

Distributor (`GICD_*`)

Register	Offset	Purpose
CTLR	`+0x000`	Group enable controls; view depends on security

Register	Offset	Purpose
TYPER	`+0x004`	Implemented interrupt register range
IGROUPRn	`+0x080 + 4n`	Secure grouping: 0 = Group 0
ISENABLERn	`+0x100 + 4n`	Write-one enable
ICENABLERn	`+0x180 + 4n`	Write-one disable
ISPENDRn / ICPENDRn	`+0x200 / +0x280 + 4n`	Pending state / write-one clear
IPRIORITYR byte	`+0x400 + INTID`	Priority byte per interrupt
ITARGETSR byte	`+0x800 + INTID`	CPU target mask for SPIs
ICFGRn	`+0xC00 + 4n`	Two bits per INTID; bit 1 selects edge vs level

CPU Interface (`GICC_*`)

Register	Offset	Purpose
CTLR	`+0x000`	Interface/group enable, IRQ/FIQ and EOI policy
PMR	`+0x004`	Only priorities numerically **less than** PMR pass
BPR	`+0x008`	Group/subpriority split; implemented resolution is limited
IAR	`+0x00C`	Read acknowledges one interrupt, returns a token

Register	Offset	Purpose
EOIR	`+0x010`	Write matching acknowledge token
RPR / HPPIR	`+0x014 / +0x018`	Running/highest-pending priority

The selected policy sets Group 0, enables it in both blocks, leaves FIQEn clear so Group 0 arrives as IRQ, and leaves split EOI mode off. Another security policy needs different access/group handling, not just `cpsie i`.

The path is: peripheral condition -> pending SPI -> priority/target filter -> CPU IRQ exception -> IAR read -> peripheral service -> **source deassertion** -> EOI -> exception return. Clearing GIC pending alone does not clear a level-sensitive peripheral condition.

15.6 GIC bring-up code

These are complete modules for the policy above, but not a standalone boot image. Add `gic.o` and `vectors.o` to Chapter 10's project, replace its placeholder vector module rather than linking two `_vectors` definitions, and integrate section 15.4. If the placeholder is inside `startup.S`, remove only its old `.vectors` block, retaining `_start` and runtime initialization. BSS clearing initializes the handler table.

`gic.h`:

```
#ifndef GIC_H
#define GIC_H
#include <stdint.h>
typedef void (*irq_handler_t)(void);
void gic_init(void);
void gic_register(uint32_t irq, irq_handler_t fn);
void gic_enable_irq(uint32_t irq);
void gic_disable_irq(uint32_t irq);
void c_irq_dispatch(void);
static inline void irq_enable(void)
{
    __asm__ volatile ("dsb sy\n\tcpsie i\n\tisb" ::: "memory");
}
static inline void irq_disable(void)
{
    __asm__ volatile ("cpsid i" ::: "memory");
}
#endif
```

`gic.c`:

```
#include "gic.h"
#define REG(a) (*(volatile uint32_t *)(uintptr_t)(a))
#define BYTE(a) (*(volatile uint8_t *)(uintptr_t)(a))
#define GICD 0x00A01000u
#define GICC 0x00A02000u
#define MAX_IRQ 192u
static irq_handler_t handlers[MAX_IRQ];
static uint32_t num_lines;

void gic_init(void)
{
    irq_disable();                    /* FIQ must already be masked */
    REG(GICC + 0x000) = 0;
    REG(GICD + 0x000) = 0;
    __asm__ volatile ("dsb sy" ::: "memory");
    num_lines = ((REG(GICD + 0x004) & 31u) + 1u) * 32u;
    if (num_lines > MAX_IRQ) num_lines = MAX_IRQ;
    for (uint32_t i = 0; i < num_lines; i++) handlers[i] = 0;
    /* Disable PPIs; SGIs are not used by this example. */
    REG(GICD + 0x180) = 0xFFFF0000u;
    REG(GICD + 0x280) = 0xFFFF0000u;
    REG(GICD + 0x080) = 0;             /* Secure Group 0 */
    for (uint32_t i = 32; i < num_lines; i += 32) {
        REG(GICD + 0x180 + 4u * (i / 32u)) = 0xFFFFFFFFu;
        REG(GICD + 0x280 + 4u * (i / 32u)) = 0xFFFFFFFFu;
        REG(GICD + 0x080 + 4u * (i / 32u)) = 0;
    }
    for (uint32_t i = 32; i < num_lines; i++) {
        BYTE(GICD + 0x400 + i) = 0xA0;
        BYTE(GICD + 0x800 + i) = 1;     /* CPU0 */
    }
    /* All SPIs disabled: level-sensitive defaults for our UART/EPIT. */
    for (uint32_t i = 32; i < num_lines; i += 16)
        REG(GICD + 0xC00 + 4u * (i / 16u)) = 0;
    REG(GICC + 0x004) = 0xFF;
    REG(GICC + 0x008) = 0;
    __asm__ volatile ("dsb sy" ::: "memory");
    REG(GICD + 0x000) = 1;             /* EnableGrp0 */
    REG(GICC + 0x000) = 1;             /* Group0 IRQ, combined EOI */
    __asm__ volatile ("dsb sy\n\tisb" ::: "memory");
}

void gic_register(uint32_t irq, irq_handler_t fn)
{
    if (irq >= 32 && irq < num_lines) handlers[irq] = fn;
}

void gic_enable_irq(uint32_t irq)
{
    if (irq >= 32 && irq < num_lines) {
        __asm__ volatile ("dsb sy" ::: "memory");
        REG(GICD + 0x100 + 4u * (irq / 32u)) = 1u << (irq & 31u);
    }
}

void gic_disable_irq(uint32_t irq)
{
    if (irq >= 32 && irq < num_lines)
```

```
        REG(GICD + 0x180 + 4u * (irq / 32u)) = 1u << (irq & 31u);
}

void c_irq_dispatch(void)
{
    uint32_t iar = REG(GICC + 0x00C);
    uint32_t irq = iar & 0x3FFu;
    if (irq >= 1020) return;            /* special IDs: no normal EOI */
    if (irq >= 32 && irq < num_lines && handlers[irq])
        handlers[irq]();
    else
        gic_disable_irq(irq);           /* quarantine an unhandled SPI */
    __asm__ volatile ("dsb sy" ::: "memory"); /* peripheral clear before EOI */
    REG(GICC + 0x010) = iar;            /* preserve complete IAR token */
    __asm__ volatile ("dsb sy" ::: "memory");
}
```

Register/unregister and enable/disable sources only with CPU IRQs masked in this simple API. It supports SPIs, not SGI messaging or generic PPIs. It does not reset inherited active interrupts; use a clean, approved boot state rather than applying it under an existing interrupt owner.

Priority/target accesses are bytes. The hardware determines the number of implemented priorities and INTIDs; our handler table caps the supported range. PMR=0xFF admits the chosen 0xA0 priority, not literally every possible priority value.

15.7 Hooking up the UART1 IRQ

RM Table 3-1 lists UART1 as **SPI offset 26**, so its GIC INTID is `32 + 26 = 58`. Add 32 only to SPI offsets, not to an already absolute INTID or a PPI number.

The following is a **fragment inside Chapter 12's UART implementation**, where its private register definitions are visible. Export the install function in `uart.h`. Preserve the 80 MHz UART root prerequisite, `UCR4=0` polling baseline, and exact 115200 baud settings; do not globally change unrelated UART control bits.

```
#include "gic.h"
static volatile uint32_t rx_count;

static void uart1_isr(void)
{
    while (REG(UART_USR2) & (1u << 0)) { /* RDR: data available */
        uint32_t rx = REG(UART_URXD);
        uart_putc((int)(rx & 0xFFu));    /* deliberately polling echo */
        rx_count++;
    }
}

void uart1_install_isr(void)
{
    /* CPU IRQs masked; uart_init() and gic_init() already completed. */
    REG(UART_UCR1) &= ~(1u << 9);       /* RRDYEN off during setup */
    REG(UART_UFCR) = (REG(UART_UFCR) & ~0x3Fu) | 1u; /* RXTL=1 */
```

```
    gic_register(58, uart1_isr);
    gic_enable_irq(58);
    REG(UART_UCR1) |= 1u << 9;          /* RRDYEN */
}
```

RX-ready depends on **UFCR.RXTL**, not merely RDR. RXTL=1 makes one received byte enough to trigger. Reading URXD drains the FIFO and removes the level condition; do not write a fictitious W1C receive-ready bit. A production receiver must inspect URXD error flags and handle overruns; this echo drops that information deliberately.

Main-body fragment after vectors/stacks and UART are ready:

```
gic_init();
printf("Interrupt-driven echo\r\n"); /* finish foreground TX first */
uart1_install_isr();
irq_enable();
for (;;) __asm__ volatile ("wfi" ::: "memory");
```

WFI is a wait hint; an interrupt or another permitted wake condition can complete it. The loop has no foreground work, so an interrupt immediately before WFI does not lose queued main-thread work. A queue-based application needs a proper check-and-sleep protocol.

Polling TX inside the ISR is acceptable only for this bounded-rate demo. IRQs remain masked during echo, so it delays every other IRQ. Do not share an unprotected printf/TX stream with foreground code after enabling it.

15.8 What happens when you type a character

1. UART1 shifts in the byte. The FIFO reaches RXTL=1 and RRDY is asserted.
2. RRDYEN enables its level-sensitive interrupt request.
3. GIC sees SPI 26 / INTID 58, enabled and targeted to CPU0.
4. Priority 0xA0 passes PMR=0xFF and the running-priority/group filters.
5. CPU saves CPSR to SPSR_irq, selects IRQ SP/LR, and enters VBAR+0x18.
6. The vector load reaches irq_entry. It saves a 64-byte core/return frame on the IRQ stack and calls the dispatcher without enabling nesting.
7. IAR acknowledges INTID 58. The dispatcher calls the UART ISR.
8. The ISR drains URXD and echoes; draining removes the receive-ready source.
9. The dispatcher orders device writes before writing the full IAR token to EOIR.
10. The wrapper restores registers; RFE restores the resume PC and CPSR.

This is the hardware/privileged entry path. Linux adds its own entry, IRQ-domain mapping and dispatch machinery. User space does not run the GIC ISR; it can receive data later through a driver and system call.

15.9 Key Notes

- Cortex-A7 banks IRQ SP/LR and saves CPSR, but software owns shared-register preservation, stack alignment and exception return. A C handler alone is not an exception-entry interface.
- The vector table is executable instructions, not Cortex-M-style handler pointers. VBAR, SCTLR vector/entry controls and the current security state together select the actual path.
- UART, GIC distributor, CPU interface and CPSR masks each gate delivery. This example owns the GIC exclusively under a Secure Group-0, non-nested policy. It is not an initializer for an inherited interrupt owner.
- A level-sensitive source must be deasserted before EOI. Draining the UART FIFO removes its receive condition, while EOIR consumes the full IAR token. One IRQ can service several bytes, so byte counts and IRQ counts differ.
- Polling TX inside the ISR delays other IRQs while nesting is disabled. A queued alternative needs bounded capacity, queue-full policy, critical sections and TX-ready disabling when empty. Neither makes printf reentrant.
- WFI changes the waiting behavior, not the interrupt’s ownership. A foreground queue needs a check-and-sleep protocol, and a power-saving claim requires a measurement rather than the presence of WFI alone.
- IRQ’s resume rule is exception-specific. The current SVC and abort handlers fail-stop. A returning service needs its own valid wrapper and stack. Address 1 is not a guaranteed fault trigger, and calling C from ABT without a context wrapper is unsafe.

15.10 Pitfalls

- **VBAR with SCTLR.V still set.** High vectors override normal VBAR.
- **Stale instruction bytes.** ROM may leave I-cache active; synchronize a replaced vector table under the actual cache policy.
- **Wrong security/group view.** The Secure Group-0 setup is not a generic Non-secure recipe.
- **Invalid IRQ stack.** Initialize its banked SP and verify its bounds.
- **FP/NEON in the call tree.** Core-register preservation is not FP context preservation.

- **Wrong exception return rule.** IRQ uses LR-4; other exceptions have different rules. A data-abort retry without fixing the fault just faults again.
- **EOI before clearing the source.** A level request becomes pending again.
- **Unhandled SPI left enabled.** It can create an interrupt storm.
- **Nesting or PL0 mask writes.** This example supports neither nested IRQs nor user-mode control of CPSR.I.

15.11 Going deeper

- Arm IHI 0048B.b, GICv2 specification: grouping, acknowledge tokens, special INTIDs, priorities, and EOI.
- **Arm DDI 0464, Cortex-A7 TRM**, and **Arm DDI 0406, ARMv7-A/R Architecture Reference Manual**: exception entry, VBAR/SCTLR, SRS/RFE.
- Arm AAPCS32: core-register and stack-alignment requirements.
- **IMX6ULLRM Table 3-1** and system memory map: SoC SPI offsets and GIC-400 addresses.
- Linux `arch/arm/kernel/entry-armv.S` and `drivers/irqchip/irq-gic.c`: production paths, not code to splice into this minimal wrapper.
- **i.MX6ULL ERR008961**: Hyp exceptions and Monitor-routed mask behavior, outside the restricted SVC/IRQ policy here.

> Next chapter: **Chapter 16: Timers (EPIT and GPT).** A periodic source will make interrupt counts observable without relying on incoming UART traffic.

Chapter 16: Timers, EPIT and GPT

An LED delay loop changes when the compiler changes. A timer lets us ask for an interval without guessing how many instructions the loop takes. We will use GPT1 to read elapsed time and EPIT1 to request periodic service.

The examples require a verified **IPG clock**, not an assumed 696 MHz CPU. They run in the Chapter 10/15 privileged, MMU-off, integer-only environment and remain in normal run mode. DDR and a higher ARM frequency are not prerequisites.

16.1 Two timers, two jobs

- **GPT1** counts up freely. We read it for delays and elapsed-time tests.
- **EPIT1** counts down and reloads. Its compare event requests an IRQ.

A free-running counter and an interrupt source solve different problems: “how much time passed?” and “when should software run?” Linux calls these clocksource and clockevent roles. This is a teaching split, not a claim that every i.MX6ULL Linux configuration uses GPT plus EPIT. The generic timer or a GPT driver may provide those roles, depending on the kernel and device tree.

Use the Chapter 13 audit before initializing these modules. If IPG changes later, stop and reconfigure both timers under the documented procedure; their units are no longer valid.

16.2 GPT, free-running counter

GPT1 base is `0x02098000`; the supplied RM describes GPT in **Chapter 30**.

Register	Offset	Role
GPT_CR	`+0x00`	Control/source/mode
GPT_PR	`+0x04`	Prescaler
GPT_SR	`+0x08`	Compare/capture/rollover status
GPT_IR	`+0x0C`	Interrupt enables

Register	Offset	Role
GPT_OCR1..3	+0x10..18	Output compare
GPT_ICR1..2	+0x1C..20	Input capture
GPT_CNT	+0x24	Current counter

Create `timers.h`:

```
#ifndef TIMERS_H
#define TIMERS_H
#include <stdint.h>
int gpt_init(uint32_t ipg_hz);
int epit_init(uint32_t ipg_hz);
uint32_t gpt_now_us(void);
void udelay(uint32_t us);
void mdelay(uint32_t ms);
uint32_t tick_ms(void);
#endif
```

The following is the **first half of `timers.c`**; append section 16.3's second half to it. Both initializers return 0 on completion, -1 if their clock input is unsuitable or reset polling exhausts its budget. That budget counts software iterations, not calibrated microseconds.

```
#include "timers.h"
#include "gic.h"
#define REG(a) (*(volatile uint32_t *)(uintptr_t)(a))
#define CCM_CCGR1 0x020C406Cu
#define GPT1_BASE 0x02098000u
#define GPT_CR    (GPT1_BASE + 0x00u)
#define GPT_PR    (GPT1_BASE + 0x04u)
#define GPT_SR    (GPT1_BASE + 0x08u)
#define GPT_IR    (GPT1_BASE + 0x0Cu)
#define GPT_CNT   (GPT1_BASE + 0x24u)

static int wait_clear(uint32_t address, uint32_t mask)
{
    for (uint32_t budget = 1000000u; budget != 0; budget--)
        if (!(REG(address) & mask)) return 0;
    return -1;
}

int gpt_init(uint32_t ipg_hz)
{
    if (ipg_hz == 0 || ipg_hz % 1000000u != 0) return -1;
    uint32_t divisor = ipg_hz / 1000000u;
    if (divisor == 0 || divisor > 4096) return -1;
```

```
    /* CCGR1 CG10 bus gate and CG11 serial gate; RUN/WAIT, not STOP. */
    REG(CCM_CCGR1) |= (3u << 20) | (3u << 22);
    REG(GPT_CR) = 0;                    /* EN=0 before source/reset changes */
    REG(GPT_CR) = 1u << 15;             /* SWR */
    if (wait_clear(GPT_CR, 1u << 15)) return -1;
    REG(GPT_IR) = 0;                    /* GPT is polled, no IRQ */
    REG(GPT_SR) = 0x3Fu;                /* W1C all status flags */
    REG(GPT_PR) = divisor - 1u;
    REG(GPT_CR) = (1u << 9)             /* FRR */
                | (1u << 6)           /* CLKSRC=001: ipg_clk */
                | (1u << 1)           /* ENMOD: reset on enable */
                | 1u;                 /* EN */
    __asm__ volatile ("dsb sy" ::: "memory");
    return 0;
}

uint32_t gpt_now_us(void)
{
    return REG(GPT_CNT);
}

void udelay(uint32_t us)
{
    while (us != 0) {
        uint32_t chunk = us > 1000000u ? 1000000u : us;
        uint32_t start = gpt_now_us();
        /* One guard tick avoids returning early due to tick phase. */
        while ((uint32_t)(gpt_now_us() - start) < chunk + 1u) {}
        us -= chunk;
    }
}

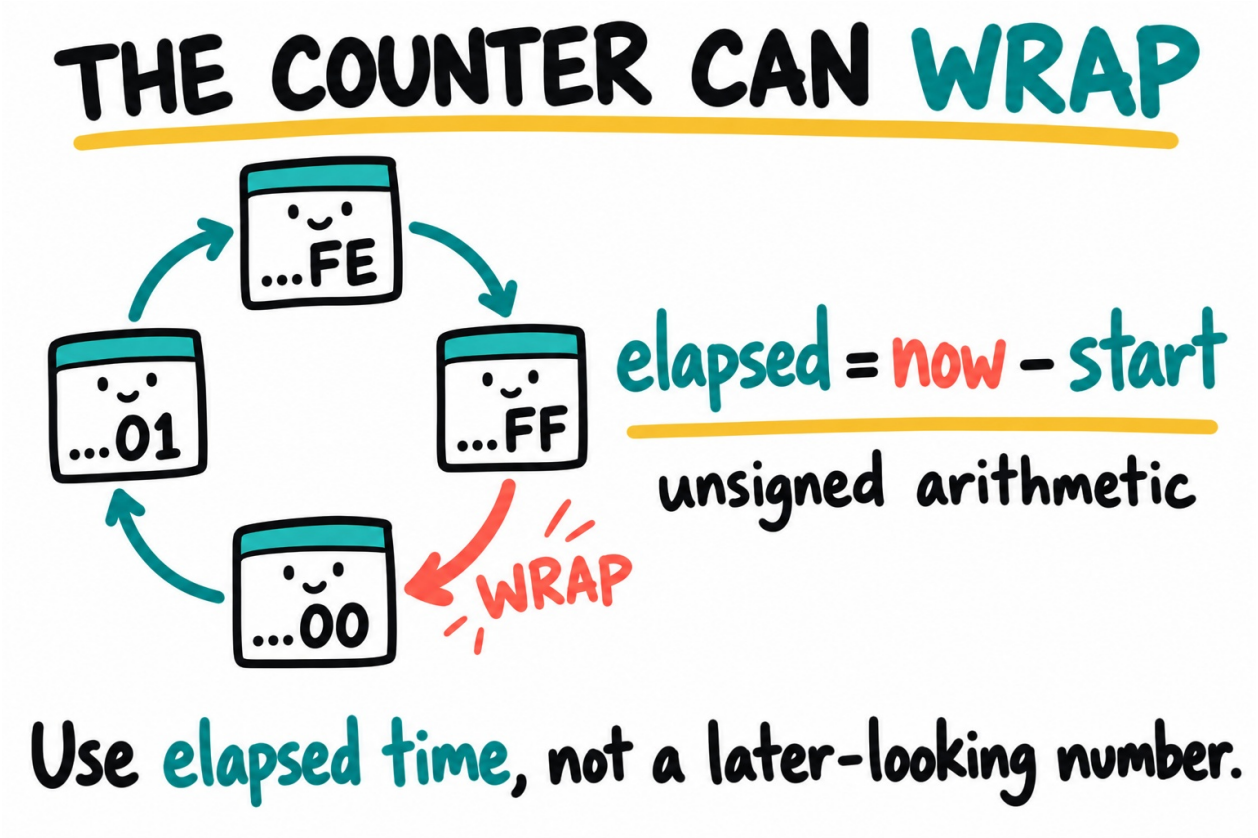
void mdelay(uint32_t ms)
{
    while (ms != 0) {
        udelay(1000u);
        ms--;
    }
}
```

With verified IPG=66 MHz, PR=65 divides by 66, producing 1 MHz. CLKSRC=001 selects `ipg_clk`; CLKSRC=010 selects the separate high-frequency reference/PERCLK path. Do not interchange them.

Unsigned subtraction handles one counter wrap. At 1 MHz, wrap occurs about every 71.58 minutes. Each delay chunk is only one second, but the polling code must still run often enough not to miss a full counter period. It requires a successfully started timer that does not stop during the wait. It has no fallback if that clock disappears.

The unit is nominally one microsecond, not a one-microsecond accuracy guarantee. Quantization, crystal error, MMIO latency, loop overhead, and IRQ service add uncertainty. The guard tick favors minimum delay; `udelay(1)` repeated a million times will **not** take exactly one second.

16.3 EPIT, periodic interrupt



Zero can come after a large count. For a 32-bit timer, the full wrap is from `0xFFFFFFFF` to `0x00000000`. The drawing abbreviates those digits. Unsigned subtraction works for an interval shorter than one full period, provided we do not miss that period while waiting.

EPIT1 base is `0x020D0000`; the supplied RM describes EPIT in **Chapter 24**.

Register	Offset	Role
EPIT_CR	`+0x00`	Source/prescaler/reload/control
EPIT_SR	`+0x04`	Compare flag, write-one-to-clear
EPIT_LR	`+0x08`	Reload value
EPIT_CMPR	`+0x0C`	Compare value
EPIT_CNR	`+0x10`	Current down-counter

The steady reload sequence includes zero: with LR=4 it counts `4,3,2,1,0,4,...`, five source ticks per period (RM Figure 24-3). For 66 MHz and 1 kHz, use **LR=65999**, not 66000, with CMPR=0. The initial enable-to-first-compare interval has its own starting phase; measure steady consecutive events when checking the period.

Append this **second half of `timers.c`**:

```
#define EPIT1_BASE 0x020D0000u
#define EPIT_CR    (EPIT1_BASE + 0x00u)
#define EPIT_SR    (EPIT1_BASE + 0x04u)
#define EPIT_LR    (EPIT1_BASE + 0x08u)
#define EPIT_CMPR  (EPIT1_BASE + 0x0Cu)
#define EPIT1_INTID 88u                 /* SPI offset 56 + 32 */
static volatile uint32_t jiffies_ms;

static void epit_isr(void)
{
    REG(EPIT_SR) = 1u;                   /* W1C OCIF; source before EOI */
    jiffies_ms++;
}

int epit_init(uint32_t ipg_hz)
{
    if (ipg_hz < 2000u || ipg_hz % 1000u != 0) return -1;
    /* gic_init() completed; CPU IRQs remain masked during setup. */
    gic_disable_irq(EPIT1_INTID);
    REG(CCM_CCGR1) |= 3u << 12;          /* CG6 */
    REG(EPIT_CR) = 0;
    REG(EPIT_CR) = 1u << 16;             /* SWR */
    if (wait_clear(EPIT_CR, 1u << 16)) return -1;
    uint32_t control = (1u << 24)        /* CLKSRC=01: ipg_clk */
                     | (1u << 3)        /* RLD: reload from LR */
                     | (1u << 2)        /* OCIEN */
                     | (1u << 1);       /* ENMOD: load on enable */
    REG(EPIT_CR) = control;              /* source selected with EN=0 */
    REG(EPIT_LR) = ipg_hz / 1000u - 1u;
    REG(EPIT_CMPR) = 0;
    REG(EPIT_SR) = 1u;
    jiffies_ms = 0;
    gic_register(EPIT1_INTID, epit_isr);
    gic_enable_irq(EPIT1_INTID);
    REG(EPIT_CR) = control | 1u;
    __asm__ volatile ("dsb sy" ::: "memory");
    return 0;
}

uint32_t tick_ms(void)
{
    return jiffies_ms;
}
```

CLKSRC is bits 25:24; PRESCALER is 15:4 (zero means divide by one). IOVW is bit 17, but this sequence does not need it: ENMOD loads LR on enable. Disabling before changing sources and clearing OCIF are intentional.

The handler is installed before the timer starts and CPU IRQs are enabled last. Chapter 15’s dispatcher performs the barrier before GIC EOI. EPIT’s source is level-sensitive; clearing its flag is not optional.

On this single-core build, the aligned 32-bit volatile read is one single-copy access. It can return the old or new value around an IRQ. It is not a general synchronization primitive. The unsigned count wraps in about 49.71 days; use unsigned differences for short intervals.

More importantly, this counts **serviced events**, not guaranteed elapsed milliseconds. OCIF is a flag, not a queue: multiple matches while IRQs are blocked can coalesce. A 64-bit version would need a defined update protocol and masked snapshot or sequence counter, not an arbitrary low/high read order.

16.4 Putting it together

This `main.c` integrates the complete modules with Chapter 12’s UART/printf and Chapter 15’s vectors/startup/GIC. Add `timers.o`, `clocks.o`, `gic.o`, and `vectors.o` to the existing project and preserve its linker bounds. Use matching integer-only architecture flags when compiling and linking; put `-lgcc` after objects. This is not an independent board loader.

No generic `clocks_init()` is called. The board/ROM configuration must be approved first; the audit supplies the actual supported IPG rate. Do not install the UART echo ISR for this example: it would share TX with foreground printf and lengthen timer latency.

```
#include "uart.h"
#include "clocks.h"
#include "gic.h"
#include "timers.h"
int printf(const char *fmt, ...);

int main(void)
{
    uart_init();                          /* Chapter 12 root prerequisite */
    uint32_t ipg_hz = clocks_get_ipg_hz();
    gic_init();                           /* leaves CPU IRQs masked */
    if (gpt_init(ipg_hz) || epit_init(ipg_hz)) {
        printf("Timer setup failed; IPG=%u Hz\r\n", ipg_hz);
        for (;;) {}
    }
    irq_enable();
    printf("Timers running; IPG=%u Hz\r\n", ipg_hz);
    uint32_t t0 = gpt_now_us();
    udelay(10000u);
    uint32_t elapsed = gpt_now_us() - t0;
    printf("10 ms request: %u timer us\r\n", elapsed);
    for (uint32_t i = 0;; i++) {
        printf("[%u serviced ticks] heartbeat %u\r\n", tick_ms(), i);
        mdelay(1000u);
```

```
    }
}
```

A representative run would show a delay of **at least** 10000 nominal timer microseconds, then increasing heartbeat/tick counts. It will not print exact 0/1000/2000 timestamps: setup, the initial delay, UART transmission, and the guarded repeated delays all take time. Measuring a delay with the same GPT validates software behavior, not the oscillator's absolute accuracy.

If the tick stops, follow the path in order:

- Does EPIT_CNR change? Check IPG, CCGR1 CG6, EN and CLKSRC.
- Does OCIF become set? Check LR/CMPR and the current count.
- Can INTID 88 pass the GIC enable, level, group, target and priority policy?
- Are VBAR, IRQ stack, and CPSR.I correct?
- Does the handler clear OCIF before EOI? Avoid stopping in the ISR with a breakpoint and interpreting debugger-induced loss as normal timing.

Run mode is deliberate. GPT/EPIT WAITEN/STOPEN and CCM low-power controls are not configured here. Do not add WFI and assume these clocks keep running in every low-power state.

16.5 Profiling with PMU CCNT and GPT

Chapter 13's PMU fragments explicitly enable CCNT and clear divide-by-64. Keep them in the profiling translation unit or a shared header; do not define the inline reader a second time.

PMU gives CPU cycles; GPT gives nominal microseconds while IPG is unchanged. Changing MMDC alone does not change GPT's time base. Memory changes can, however, change how many cycles an operation takes.

For an operation lasting well above GPT's one-tick resolution, use this **main-body fragment** after both counters are ready:

```
uint32_t c0 = pmu_ccnt();
uint32_t u0 = gpt_now_us();
/* Perform the operation here; keep all destructive tests in reserved RAM. */
__asm__ volatile ("dsb sy" ::: "memory");
uint32_t c1 = pmu_ccnt();
uint32_t u1 = gpt_now_us();
printf("%u CPU cycles across %u nominal us\r\n", c1 - c0, u1 - u0);
```

Keep the interval below one CCNT wrap at the actual frequency. Endpoint reads are not simultaneous; measure their overhead, repeat runs, and state whether interrupts/cache effects are included. A long-run cycles-per-microsecond ratio can cross-check the

decoded ARM/IPG ratio, but both may share the same oscillator, so it is not independent proof of absolute Hz. A mismatch is a question to investigate, not proof that the clock initializer is wrong.

Instruction count alone does not determine memtest throughput or elapsed time. Pipeline behavior, loop instructions, memory, cache and interrupt service all matter. A 1 MHz GPT cannot directly resolve a sub-microsecond interval.

16.6 Key Notes

- GPT’s free-running count represents elapsed ticks. EPIT requests periodic service. Their clocksource/clockevent roles separate time measurement from scheduling, without prescribing Linux’s eventual timer selection.
- Both initializers depend on a known, stable IPG rate and the selected source encoding. A later clock change invalidates their units until reconfiguration.
- EPIT’s repeating reload period includes zero. With 66 MHz IPG, LR=65999 gives 66000 source ticks per period. Its initial phase need not match the interval between steady consecutive events.
- OCIF is a flag, not an event queue. Matches can coalesce while IRQs are masked, so `tick_ms()` counts serviced interrupts rather than guaranteed elapsed milliseconds. Short ISR work preserves other sources’ service time.
- Unsigned subtraction crosses rollover for intervals shorter than one full period. From `0xFFFFFFF0` to `0x00000020` the difference is 48 ticks. That arithmetic says nothing about whether a physical timer kept running.
- A guard tick and polling overhead favor minimum-delay behavior, not exact duration. Low-power clock continuity and a stopped-counter failure path are separate design requirements not supplied by these run-mode delay loops.
- PMU cycles and GPT ticks describe different units. Their ratio can check a clock-tree calculation, but a shared oscillator prevents it from being an independent absolute-frequency measurement.

16.7 Pitfalls

- **Only one GPT gate.** CCGR1 CG10 is bus, CG11 serial; EPIT1 is CG6.
- **IPG vs PERCLK.** Source encodings select different roots.
- **Assumed IPG rate.** The divisor must match the audited, stable input.
- **Reload off by one.** The repeating EPIT sequence includes zero.
- **Missing W1C.** OCIF must be cleared at the source before GIC EOI.

- **Long masked interval.** A flag cannot count every missed match.
- **Delay timer stopped.** The polling loop cannot finish without a running counter, even though its wrap arithmetic is correct.
- **64-bit counter without a protocol.** Use a coherent snapshot/update scheme; volatile alone does not prevent a torn multiword read.
- **Low-power assumptions.** Gate encoding 11 excludes STOP; timer and CCM mode controls still determine behavior.

16.8 Going deeper

- **IMX6ULLRM Chapter 30 (GPT), Chapter 24 (EPIT)**: source encodings, reset exceptions, low-power controls, and Figure 24-3.
- **IMX6ULLRM Chapter 18**, CCGR1: separate GPT bus/serial gates.
- **Cortex-A7 TRM**, generic timer and performance-monitor chapters: alternative time bases and cycle-counter controls.
- Linux `drivers/clocksource/timer-imx-gpt.c`: clocksource/clockevent integration; check the actual device tree before assuming it is selected.
- POSIX `clock_gettime(CLOCK_MONOTONIC)`: the user-space abstraction, not a direct read of our EPIT software count.

> Next chapter: **Chapter 17: MMU and caches.** Memory attributes and cache maintenance add new requirements to the MMIO and executable-copy paths. Keep this chapter’s explicit memory/cache policy with those examples.

Chapter 17: MMU and caches

A UART status register and a byte in OCRAM are both reached through pointers. Yet only one should be remembered in a cache. If the CPU keeps yesterday’s UART status, your familiar polling loop can wait forever. If it keeps a recently used RAM byte, the same mechanism can save a bus transaction.

The Memory Management Unit (MMU) translates virtual addresses to physical addresses and checks access permissions. It also tells the CPU which kind of memory a pointer reaches. The example uses an identity map, so the pointer values stay familiar while their permissions and memory attributes change. A cache keeps copies of instructions or data closer to the CPU.

The design keeps code, table and small benchmark in OCRAM, provided the linked image and reserved stacks fit. DDR is an optional extension, not a requirement for understanding the map. The example uses one privileged address space and 1 MiB sections. Its startup sequence is limited to the book’s fresh-ROM path, with IRQ/FIQ masked during the transition rather than inheriting Chapter 16’s enabled IRQ state.

17.1 What we are not doing

Our first-level table has 4096 entries: one for each possible 1 MiB virtual region. Having a slot for every region does not mean we should grant access to all of them.

- **No second-level tables:** sections are enough for this first map. Smaller pages would let us protect code, data and holes within a section separately.
- **No address-space switching:** entries are global (`nG=0`); we do not use per-process ASIDs (address-space identifiers).
- **No process isolation:** everything here runs at PL1, the privileged level. PL0 is the unprivileged level used by Linux applications.
- **No LPAE:** the short-descriptor format is sufficient for this map. Large Physical Address Extension is a separate translation format, not something required just to enable caches.
- **No stage-2 translation:** there is no active hypervisor mapping beneath our stage-1 map.

We will map the GIC and AIPS peripheral windows as Device memory and OCRAM as Normal memory. The GIC routes interrupts; MMDC, within the peripheral windows, controls DDR. Neither controller’s registers belong in a RAM cache. If you later qualify DDR on your exact board, you can add only that qualified capacity to the map.

The table occupies 16 KiB. TTBR0, the Translation Table Base Register, points to its physical address. Once translation is enabled, the regions we chose still have virtual address = physical address. The benefit we will investigate comes from caching, not from changing the numbers in pointers.

17.2 Short-descriptor first-level entry, decoded

For a 1 MiB section, the 32-bit descriptor has these fields:

Bits	Field	Choice in this chapter
`31:20`	Physical section base	A 1 MiB-aligned physical address
`19`	NS	Zero; we do not change the current security state
`18`	Supersection	Zero: a section, not a 16 MiB supersection
`17`	nG	Zero: global translation
`16`	S	Zero for our single-core Normal RAM mapping; see Device encoding below
`15`	AP[2]	Zero
`14:12`	TEX[2:0]	Memory-type encoding
`11:10`	AP[1:0]	`01`: PL1 read/write, PL0 no access, with `SCTLR.AFE=0`
`9`	Implementation-defined	Zero
`8:5`	Domain	Zero, checked through DACR
`4`	XN	One for registers, zero for executable RAM in this simplified map

Bits	Field	Choice in this chapter
3	C	Part of the memory-type encoding
2	B	Part of the memory-type encoding
1	Section selector	One
0	PXN on implementations supporting it	Zero here, giving low bits 10

Do not reuse this layout for a page-table pointer or a second-level page. Low bits 00 mean a fault entry; 01 points to a second-level table. Bit 18 distinguishes sections from supersections. On implementations supporting PXN (Privileged Execute Never), low bits 11 can also describe a section/supersection with PXN set. We use 10 throughout.

Two control bits make the following interpretation possible: SCTLR.TRE=0 disables TEX remapping, and SCTLR.AFE=0 selects the AP permission interpretation above. With AFE set, AP[0] is an access flag instead. In Non-secure tables NS must be zero; in Secure tables it selects the Secure/Non-secure physical address space. This example does not switch between those worlds.

Region	TEX	C	B	XN	AP[2:0]	Meaning with TRE=0
Device registers	000	0	1	1	001	Shareable Device, PL1 RW, no execution
Normal cacheable RAM	001	1	1	0	001	Inner/outer write-back, write-allocate; S=0 here
Normal non-cacheable RAM	001	0	0	As needed	001	Ordinary RAM without caching

Region	TEX	C	B	XN	AP[2:0]	Meaning with TRE=0
Strongly-ordered reference	000	0	0	1	001	Separate memory type, not a variety of Normal memory

The Device encoding `000/0/1` selects Shareable Device even though our descriptor's S bit is zero. Non-shareable Device has a different TEX encoding. Shareable does **not** mean that a peripheral automatically snoops CPU caches.

For a worked decode, the OCRAM descriptor `0x0090140E` selects physical section `0x00900000`, AP=`001`, TEX=`001`, C=B=1 and XN=0. Its low bits are `10`. A Device descriptor for `0x02000000` is `0x02000416`. The authoritative layouts and attribute encodings are in Arm DDI 0406C.d, Figure B3-4 and Table B3-10.

17.3 Building the L1 table

This fragment follows the book's **direct, freshly reset Boot ROM path on an open development device**, not entry from U-Boot, Linux or an arbitrary debugger session. Establish these conditions before calling it:

1. Code, constants, cleared BSS, table, benchmark buffers and every reserved stack fit in Chapter 10's application OCRAM interval, `0x00908000..0x0091FFFF`. The table is 16 KiB and the two benchmark arrays total 8 KiB; account for alignment gaps and other objects in the link map. Reserve their space and keep unrelated tests away from it.

2. The OCRAM mapping covers Chapter 10's `_vectors` entry at `0x00908000`, the installed VBAR and its handlers. Execution is little-endian at PL1, IRQ/FIQ are masked during the transition, and no DMA agent owns these buffers. Resolve any CP15 security restrictions before this point. For an optional DDR payload, also verify that its code, stacks and VBAR lie in qualified, mapped DDR.

3. `SCTLR.M=0` and `SCTLR.C=0`; the application has not yet enabled data caching. I-cache may already be enabled. IMX6ULLRM §8.4.4 says ROM enables I-cache for image download and disables D-cache/L2/MMU following authentication. That is not a claim that every cache is off at entry.

The invalidate-only routine below discards inherited cache contents under this fresh-ROM contract, before the application creates dirty cacheable data. It is **not** a handoff repair routine. If an earlier application or bootloader could own dirty lines, clean and invalidate them while the old mappings still work, using a separately reviewed transition. Clearing C alone can leave the only current copy of your stack in cache.

Use Chapter 10's integer-only flags for these fragments: `-mcpu=cortex-a7 -marm -mfloat-abi=soft -mgeneral-regs-only`, also at link time, with `-lgcc`. That avoids introducing VFP/NEON use before its startup policy is established.

`DDR_MIB=0` is the OCRAM example's default. The only other accepted values are 256 and 512, valid only after board-specific DDR initialization and an owned-range memory test have qualified that **exact installed capacity**, beginning at `0x80000000`. The build-time guard checks the value, not the hardware. Mapping DDR does not initialize it, and a successful test of a small buffer does not establish the full capacity.

`mmu.c`, a single-core initialization fragment for that contract:

```
#include <stdint.h>

#ifndef DDR_MIB
#define DDR_MIB 0                 /* OCRAM-only default */
#endif
#if DDR_MIB != 0 && DDR_MIB != 256 && DDR_MIB != 512
#error "DDR_MIB must be 0, 256, or 512; qualify the exact board capacity"
#endif

#define L1_SECTION       2u
#define L1_AP_PL1_RW     (1u << 10)
#define L1_XN            (1u << 4)
#define L1_DEVICE        (1u << 2)  /* TEX=000, C=0, B=1 */
#define L1_NORMAL_WBWA   ((1u << 12) | (1u << 3) | (1u << 2))
#define ATTR_DEVICE     (L1_SECTION | L1_AP_PL1_RW | L1_DEVICE | L1_XN)
#define ATTR_NORMAL     (L1_SECTION | L1_AP_PL1_RW | L1_NORMAL_WBWA)

static uint32_t l1_table[4096] __attribute__((aligned(16384)));

void mmu_build_table(void)
{
    for (uint32_t i = 0; i < 4096; ++i)
        l1_table[i] = 0;             /* Unlisted regions fault. */

    l1_table[0x009] = 0x00900000u | ATTR_NORMAL; /* OCRAM section */
    l1_table[0x00A] = 0x00A00000u | ATTR_DEVICE; /* Cortex-A7/GIC */
    l1_table[0x020] = 0x02000000u | ATTR_DEVICE; /* AIPS-1 */
    l1_table[0x021] = 0x02100000u | ATTR_DEVICE; /* AIPS-2, MMDC */
    l1_table[0x022] = 0x02200000u | ATTR_DEVICE; /* AIPS-3, IOMUXC_SNVS */

    /* Opt-in only: one section per qualified MiB, beginning at 0x80000000. */
#if DDR_MIB > 0
    for (uint32_t i = 0x800; i < 0x800u + DDR_MIB; ++i)
        l1_table[i] = (i << 20) | ATTR_NORMAL;
#endif
}

static inline uint32_t read_sctlr(void)
{
    uint32_t v;
    __asm__ volatile ("mrc p15, 0, %0, c1, c0, 0" : "=r"(v));
    return v;
}
```

```
/* Invalidate-only: valid ONLY when no dirty data needs preserving. */
static void invalidate_data_caches(void)
{
    uint32_t clidr;
    __asm__ volatile ("dsb sy" ::: "memory");
    __asm__ volatile ("mrc p15, 1, %0, c0, c0, 1" : "=r"(clidr));

    for (uint32_t level = 0; level < 7; ++level) {
        uint32_t type = (clidr >> (3u * level)) & 7u;
        if (type == 0) break;
        if (type < 2 || type > 4) continue; /* No data/unified cache. */

        uint32_t selector = level << 1;    /* Data/unified side */
        uint32_t ccsidr;
        __asm__ volatile ("mcr p15, 2, %0, c0, c0, 0"
                          :: "r"(selector) : "memory");
        __asm__ volatile ("isb" ::: "memory");
        __asm__ volatile ("mrc p15, 1, %0, c0, c0, 0" : "=r"(ccsidr));
        uint32_t line_shift = (ccsidr & 7u) + 4u;
        uint32_t ways = ((ccsidr >> 3) & 0x3FFu) + 1u;
        uint32_t sets = ((ccsidr >> 13) & 0x7FFFu) + 1u;
        uint32_t way_shift = ways > 1 ? __builtin_clz(ways - 1u) : 0;

        for (uint32_t way = 0; way < ways; ++way)
            for (uint32_t set = 0; set < sets; ++set) {
                uint32_t sw = selector | (set << line_shift);
                if (ways > 1) sw |= way << way_shift;
                __asm__ volatile ("mcr p15, 0, %0, c7, c6, 2"
                                  :: "r"(sw) : "memory"); /* DCISW */
            }
        /* Finish this level before changing levels: ERR008958. */
        __asm__ volatile ("dsb sy" ::: "memory");
    }
    uint32_t zero = 0;
    __asm__ volatile ("mcr p15, 2, %0, c0, c0, 0" :: "r"(zero) : "memory");
    __asm__ volatile ("isb" ::: "memory");
}

int mmu_enable(void)
{
    uint32_t sctlr = read_sctlr();
    if (sctlr & ((1u << 0) | (1u << 2))) return -1;

    invalidate_data_caches();
    mmu_build_table();             /* Written while data caching is off. */
    __asm__ volatile ("dsb sy" ::: "memory");

    uint32_t zero = 0, domain0_client = 1;
    uint32_t table_pa = (uint32_t)(uintptr_t)l1_table;
    __asm__ volatile ("mcr p15, 0, %0, c2, c0, 2"
                      :: "r"(zero) : "memory"); /* TTBCR: EAE=0, N=0 */
    __asm__ volatile ("mcr p15, 0, %0, c2, c0, 0"
                      :: "r"(table_pa) : "memory"); /* TTBR0, NC table walks */
    __asm__ volatile ("mcr p15, 0, %0, c3, c0, 0"
                      :: "r"(domain0_client) : "memory");
    __asm__ volatile ("isb" ::: "memory");
    __asm__ volatile ("mcr p15, 0, %0, c8, c7, 0"
                      :: "r"(zero) : "memory"); /* TLBIALL, current state */
    __asm__ volatile ("mcr p15, 0, %0, c7, c5, 0"
```

```
                    :: "r"(zero) : "memory"); /* ICIALLU */
    __asm__ volatile ("mcr p15, 0, %0, c7, c5, 6"
                    :: "r"(zero) : "memory"); /* BPIALL */
    __asm__ volatile ("dsb sy; isb" ::: "memory");

    /* Direct TEX/AP encoding, little-endian tables, writable example code. */
    sctlr &= ~((1u << 28) | (1u << 29) | (1u << 25)
               | (1u << 19) | (1u << 20)); /* TRE, AFE, EE, WXN, UWXN */
    sctlr |= (1u << 0) | (1u << 2) | (1u << 12) | (1u << 11);
    __asm__ volatile ("mcr p15, 0, %0, c1, c0, 0"
                    :: "r"(sctlr) : "memory"); /* M, C, I, Z */
    __asm__ volatile ("isb" ::: "memory");
    return 0;
}
```

Pause at the table's first entry: it is zero, not a Device mapping for address zero. An accidental pointer should fault rather than reach an unknown bus target.

- **Section granularity has a cost.** OCRAM occupies `0x00900000..0x0091FFFF`, but this descriptor covers through `0x009FFFFF`. It does not create RAM in the holes. Likewise, the peripheral sections include reserved addresses; do not probe them.
- **Alignment is part of the descriptor contract.** With N=0 the table base must be 16 KiB aligned. TTBR0's low bits include table-walk attributes, not address bits; passing an unaligned pointer can corrupt those attributes as well as select the wrong base. Check the linked ELF, not just the C declaration.
- **Check the application bounds before loading.** With Chapter 10's layout, loaded bytes must respect its ROM-active limit (`_edata <= 0x00918000`), BSS must stay below the reserved stack (`_ebss <= _stack_limit`), and the complete allocation ends no later than `0x00920000` (exclusive). Include any extra mode stacks added in Chapter 15. Inspect the link map and retain the linker assertions; a small raw binary can still have large BSS reservations. The table and arrays alone do not prove the whole image fits.
- **DACR controls domains.** Here `DACR=1` makes domain 0 a client and all other domains inaccessible. Clients enforce AP permissions. Manager domains bypass AP checks; they are not needed for this example.
- **L2 is real.** i.MX6ULL has 32 KiB L1 instruction/data caches and a 128 KiB integrated unified L2. The loop reads CLIDR/CCSIDR rather than invalidating only a hardcoded L1. ERR008958 requires a DSB before changing levels in set/way cleaning sequences; that boundary is also kept here.
- **The table is static after enable.** TTBR0 uses non-cacheable walks for simplicity. Editing a now-cached table later requires cleaning the descriptors, completing that maintenance, invalidating affected TLB entries and synchronizing. Some mapping changes require break-before-make; changing a live entry is not an ordinary array update.

This map deliberately leaves RAM writable and executable. A later design should separate read-only executable code from XN data. We are exposing the mechanism, not establishing Linux-style protection.

17.4 Calling `mmu_enable()`

After the OCRAM startup and link-layout checks above, use the same owned test buffers before and after enable. No relocation is required. If you choose the Chapter 14 DDR extension, establish its initialization, capacity and payload-placement contract separately; never run a destructive memory test over the live image, stack, BSS or table.

Application integration fragment; `bench_bytes()` is defined in §17.8 and `printf` is your existing UART output routine:

```
void bench_bytes(void);

int main(void)
{
    /* Board/UART/timer setup and startup contract established earlier. */
    bench_bytes();
    if (mmu_enable() != 0) {
        printf("MMU setup refused: M or C already set\r\n");
        for (;;) {}             /* Intentional fatal stop, not a status wait. */
    }
    printf("MMU setup returned; checking the same workload\r\n");
    bench_bytes();
    for (;;) {}
}
```

A returned function and working UART are useful first checks. They are not proof of every permission or every RAM address. Record the actual timings, clock configuration, build flags and checksum; no timing transcript in this chapter represents a measured board run.

17.5 What happened

For the populated entries, virtual and physical addresses still match. Loads and stores to Device registers are not cached or speculatively performed like Normal-memory accesses. OCRAM can now be served through L1 and the integrated L2, and instruction fetches can use I-cache. With `DDR_MIB=0`, DDR entries remain faults; an accidental DDR pointer has not been given access to uninitialized memory.

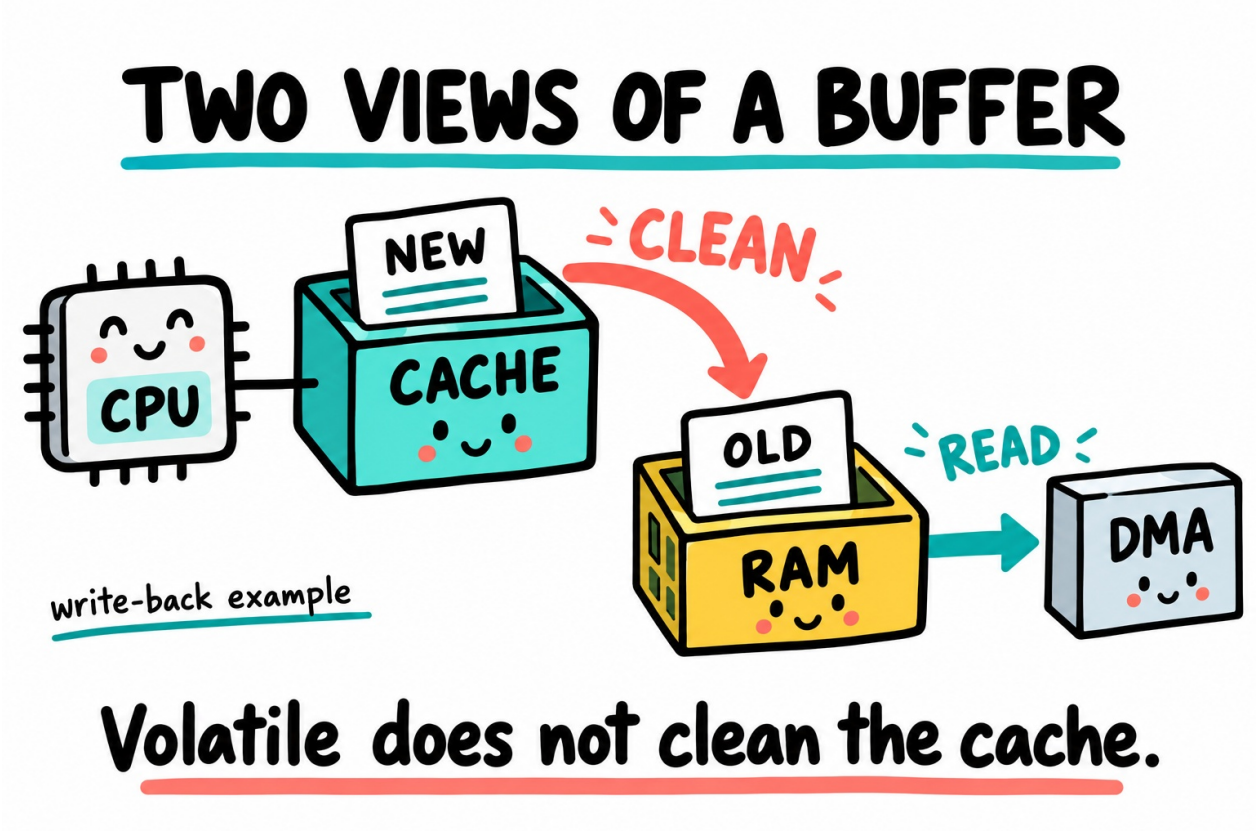
Device memory is not a replacement for all ordering rules. A buffer in Normal RAM and a start register in Device memory are different parts of a transaction. You still need barriers and, where applicable, cache maintenance between preparing that buffer and

starting hardware. A barrier can complete a register write; it cannot prove that the operation requested by that write has finished. Check the peripheral's completion condition.

Chapter 18's I2C and SPI fragments can use these Device register windows and small Normal buffers in OCRAM. Its optional LCD example needs a much larger framebuffer in separately qualified and mapped DDR. The same Device/Normal distinction matters when the LCD begins reading that buffer without the CPU's help.

17.6 Cache maintenance, when you must intervene

Suppose the CPU fills a buffer and then asks a peripheral to read it. The pointer is correct, but the newest bytes may exist only in dirty cache lines. DMA (Direct Memory Access) is another bus master; on this path it does not snoop those CPU copies.



The picture shows the state before cleaning. The clean arrow is work still to do. Complete that work and the required barrier before starting the device, then keep the submitted buffer unchanged until the transfer finishes.

Three verbs describe different operations:

Operation	What it does	Main risk
Clean	Writes dirty data toward the specified coherency point; keeps a valid copy	Cleaning too late can overwrite data a device has just produced
Invalidate	Discards a cached copy	Dirty CPU data can be lost
Clean + invalidate	Writes back, then discards	Still needs ownership and completion ordering

Use a buffer owned exclusively by one side at a time:

- **Device reads RAM:** CPU prepares buffer/descriptors, cleans to the Point of Coherency (PoC), executes DSB, then publishes the device-visible address and starts the transfer. The CPU must not change the submitted buffer until the device finishes.
- **Device writes RAM:** before submission, clean+invalidate the dedicated receive buffer to PoC and execute DSB. Keep the CPU away from it while DMA owns it. After confirmed device completion, invalidate again and execute DSB before CPU reads. The second invalidate removes any stale copy brought in during the transfer. Do not clean stale lines *after* DMA has written new data.
- **CPU writes instructions:** clean the changed D-cache lines to the Point of Unification (PoU), DSB, invalidate the corresponding I-cache lines and any required branch-predictor state, DSB, then ISB before executing. This is distinct from DMA maintenance to PoC.

PoU is where instruction and data views meet; PoC is where the relevant memory observers agree. On this integrated Cortex-A7 hierarchy, maintenance to PoC must account for L2 too, not merely put bytes into L2 while a peripheral reads DRAM.

These are single-line primitives, **not** complete transfer operations:

```
#include <stdint.h>
#include <stddef.h>

static inline void dcache_clean_va(uintptr_t p)
{
    __asm__ volatile ("mcr p15, 0, %0, c7, c10, 1"
                      :: "r"(p) : "memory"); /* DCCMVAC: PoC */
}

static inline void dcache_invalidate_va(uintptr_t p)
{
    __asm__ volatile ("mcr p15, 0, %0, c7, c6, 1"
                      :: "r"(p) : "memory"); /* DCIMVAC: PoC */
}
```

```
static inline void dcache_clean_inv_va(uintptr_t p)
{
    __asm__ volatile ("mcr p15, 0, %0, c7, c14, 1"
                      :: "r"(p) : "memory"); /* DCCIMVAC: PoC */
}

static inline void icache_invalidate_va(uintptr_t p)
{
    __asm__ volatile ("mcr p15, 0, %0, c7, c5, 1"
                      :: "r"(p) : "memory"); /* ICIMVAU: PoU */
}
```

The `"memory"` clobber constrains the compiler. DSB/ISB constrain hardware. Neither substitutes for the other, and `volatile` MMIO alone does not order ordinary buffer stores around a start-register write.

Cortex-A7 L1 D-cache and L2 lines are 64 bytes; L1 **I-cache lines are 32 bytes**. This clean-range helper is therefore for the data side only:

```
/* Same translation/ownership assumptions as the primitives above. */
int dcache_clean_range(const void *start, size_t len)
{
    if (len == 0) return 0;
    uintptr_t first = (uintptr_t)start;
    if (len - 1u > UINTPTR_MAX - first) return -1;
    uintptr_t last = (first + len - 1u) & ~(uintptr_t)63u;
    uintptr_t a = first & ~(uintptr_t)63u;
    for (;;) {
        dcache_clean_va(a);
        if (a == last) break;
        a += 64u;
    }
    __asm__ volatile ("dsb sy" ::: "memory");
    return 0;
}
```

Rounding out a **clean** range preserves neighbouring bytes. Blindly rounding an **invalidate** range can discard unrelated dirty objects. For DMA, align and pad dedicated buffers to complete 64-byte lines; ownership includes the padding. A four-byte receive object sharing a line with your counter is not a safe DMA buffer.

In Linux, use the DMA API, including mapping-error checks and the appropriate sync/unmap operations, rather than calling these bare-metal CP15 helpers. A CPU virtual pointer, a physical address and a DMA address are not generally interchangeable.

17.7 Why MMIO must be Device, not Normal Cacheable

Consider `while (!(UART_STATUS & READY)) {}`. Its `volatile` load asks the compiler to read each time. If the register window is Normal Cacheable, the CPU can still answer those reads from a cache. Compiler correctness does not repair the memory type.

The consequences go beyond slow write visibility: Normal accesses can be speculative, merged or reordered in ways a register protocol does not permit. A read with side effects may be particularly destructive. Do not test this by making a live peripheral cacheable.

Our table grants Device mappings only to the register windows we use and leaves other sections faulting. This catches some mistakes without claiming that every address inside a broad Device section is safe. A DSB also cannot repair a wrong memory type or clear a peripheral’s status flag for you.

17.8 Performance measurement, in more detail

A benchmark needs an observable result. Otherwise the compiler may remove the copying that you intended to time. The following integration fragment uses volatile bytes to make each load/store happen. It is deliberately a **byte-copy workload**, not an optimized `memcpy` or a DRAM-bandwidth test.

Provide your existing UART `printf` and a free-running hardware-timer helper `gpt_now_us()` that really returns modulo-2^32 microseconds. A raw GPT counter is not automatically in microseconds; derive its tick rate as in Chapter 16. This test does not require timer interrupts.

```
#include <stdint.h>

void bench_bytes(void)
{
    static volatile uint8_t src[4096], dst[4096];
    for (unsigned i = 0; i < 4096; ++i) src[i] = (uint8_t)i;

    __asm__ volatile ("dsb sy" ::: "memory");
    uint32_t before = gpt_now_us();
    for (unsigned n = 0; n < 1000; ++n)
        for (unsigned i = 0; i < 4096; ++i) dst[i] = src[i];
    __asm__ volatile ("dsb sy" ::: "memory");
    uint32_t elapsed = gpt_now_us() - before;

    uint32_t sum = 0;
    for (unsigned i = 0; i < 4096; ++i) sum += dst[i];
    printf("4096000 bytes: %u us, checksum %u\r\n",
           (unsigned)elapsed, (unsigned)sum);
}
```

The expected checksum is **522240**, calculated from sixteen copies of the byte values 0 through 255. That is a data check, not an invented timing result. The elapsed subtraction handles a counter wrap provided the interval is shorter than one full timer period.

Configuration to record	What the comparison tells you
Startup contract: MMU/D-cache off; record I-cache state	Baseline for your actual startup state
MMU and caches enabled, same buffers and build	Effect of the combined configuration
Separate cold run, test RAM Normal non-cacheable	Helps separate translation from data caching

The arrays total 8 KiB and are copied repeatedly. Most repetitions can hit cache; they do not stream through 4 MiB of distinct DRAM. Report 4,096,000 copied bytes, access pattern, CPU/timer clocks, compiler flags and cache state. If you use the PMU cycle counter instead, first configure its enable/divide-by-64 controls and account for its wrap. Do not convert counts to MB/s using an assumed CPU clock.

17.9 Key Notes

- Identity mapping preserves pointer values but changes access checks and memory behavior. Device/XN register mappings and Normal RAM mappings are policy choices, not consequences of `volatile`.
- A 16 KiB-aligned table and 1 MiB sections simplify translation at the cost of coarse protection. A mapped section can include holes that are not physical RAM, and writable executable RAM is not a code/data protection policy.
- Enable is a startup transition with mapped code, stacks, vectors and registers, masked IRQ/FIQ and a defined inherited cache state. Invalidate-only maintenance is unsafe when earlier software may own dirty data.
- Translation and data caching have separable effects. The repeated 8 KiB byte-copy workload can hit cache and has a calculated checksum of 522240. It is neither a DRAM-bandwidth test nor a promised speed-up.
- Aliases to the same physical RAM must use compatible attributes. Cortex-A7’s physically indexed, physically tagged D-cache does not make conflicting memory types safe.
- DMA ownership covers complete data-cache lines, including padding. Zero-length cleaning touches no line, an aligned 64-byte clean touches one, and a boundary-crossing range touches at least two. Invalidating a shared dirty line can discard a neighboring object’s bytes.

- Cleaning/invalidation, barriers and device completion solve different problems. DMA needs visibility to PoC and exclusive ownership. New instructions need synchronization to PoU and the instruction stream. A CPU-only readback cannot establish device coherency.

17.10 Pitfalls

- **Treating MMU-off as reset state.** Check both the control bits and the direct fresh-ROM contract. Never invalidate dirty live data merely because C is now zero.
- **Maintaining L1 but forgetting L2.** Read cache geometry and cover the hierarchy. Apply the DSB-between-levels requirement in supplied erratum ERR008958, especially in clean+invalidate handoff code.
- **Unmapped execution path.** Include the current PC, return address, stack, vector table and its handlers, table and UART/timer registers before setting M. Check VBAR and SCTLR.V; the example does not relocate vectors.
- **Wrong descriptor interpretation.** AFE, TRE and TTBCR.EAE change how the fields are interpreted. N=0 selects a full TTBR0 table; TTBR low bits are walk attributes, not a spare part of the base address.
- **Missing compiler or hardware barriers.** A bare `asm volatile` is not a compiler memory barrier. DSB completes the relevant accesses/maintenance; ISB synchronizes the instruction stream after control changes.
- **Changing a DMA-owned buffer.** Clean/invalidate operations cannot make simultaneous CPU/device modification safe. Confirm completion before reclaiming ownership.
- **Broad executable RAM mappings.** Useful for exposing the mechanism, not a protection policy. Leave unknown sections faulting and use finer granularity when you need code/data separation.

17.11 Going deeper

- Arm DDI 0406C.d: B3.5 descriptor formats, B3.7 access control, B3.8 memory attributes and B4 cache/TLB operations.
- Cortex-A7 TRM, DDI 0464B: cache line sizes, PIPT D-cache/VIPT I-cache and CP15 controls. Pair it with the i.MX6ULL-specific configuration in IMX6ULLRM Chapter 12.
- Supplied **IMX6ULLRM Rev. 1**, §8.4.4 for ROM cache use; supplied **IMX6ULL errata Rev. 1.2**, ERR008958 and ERR010449. Authentication/cache errata are not an invitation to change fuses for this example.
- Linux v6.12 `proc-v7-2level.S` and `mmu.c`: compare the policies and platform handling with our static table.

- Linux DMA API guide: ownership, coherent versus streaming mappings, CPU pointers versus device addresses, and barriers even for coherent memory.

> Next: **Chapter 18, Optional bare-metal peripherals.** Follow one I2C transaction, one SPI command and an LCD framebuffer handoff. The supplementary Chapters 18A-18C examine project boundaries, button timing and retained counters. None is a prerequisite for U-Boot in Part III.

Chapter 18: Optional bare-metal peripherals

A flash can receive four perfectly timed bytes and still ignore the command. If chip-select rises between the command and its reply, the flash sees separate transactions. A framebuffer can contain the right pixels while the LCD reads an older copy from DRAM. Neither problem is solved by finding another register address.

This chapter follows those boundaries: the whole bus transaction, then the whole buffer handoff. The controller logic will feel familiar from MCU work; the clock routes, pad inputs and cache ownership need a closer look.

We will trace an I2C single-byte random read, a SPI JEDEC-ID command and an eLCDIF color-bar handoff. The code supplies transaction fragments, with board setup and hardware validation still required; the display initialization remains a panel-specific sketch. Linux later coordinates these clock routes, status flags and buffers with other users of the hardware.

18.1 Why this chapter is optional

You can continue to **Part III, U-Boot** after Chapter 17. Part II has exposed the main boot-time mechanisms. Its example programs are not a production bootloader.

Read this chapter when you want to trace a request all the way to a peripheral. That helps distinguish a Linux configuration problem from a controller or wiring problem. A bare-metal probe is only appropriate when Linux and any other firmware are no longer competing for the same hardware; it is not a second owner you run beside a live driver.

The supplied **MINI V2.2 base** and **CORE V2.0** schematics show I2C routes, but do not establish an on-board `0x50` EEPROM or an ECSPI1 W25Q32 flash. The examples below therefore require a separately documented attachment, or a different board revision whose populated parts and routes you have checked. Do not infer a fitted device from a controller's presence in the SoC.

Before connecting an attachment, use the Chapter 8 power/disconnection checks. Confirm its voltage, ground, connector pinout, pull-ups and shared-net conflicts with the exact board documentation. Do not wire or swap modules while powered. A nominally 3.3 V bus is not permission to attach any EEPROM or flash variant.

Chapters **18A** (Project organization), **18B** (Button + beep) and **18C** (Bare-metal RTC) are also supplementary design cases. None is a prerequisite for beginning U-Boot.

18.2 I²C, read a byte from EEPROM

Choose an attached EEPROM whose datasheet specifies an **8-bit word address** and the combined random-read transaction below. For example, an AT24C02D uses that address width; larger EEPROMs may need two address bytes or bank bits. `0x50` is a 7-bit bus address for an appropriate address-pin configuration, not an EEPROM memory location. See the AT24C01D/AT24C02D datasheet for the chosen part's protocol.

The supplied reference-board I2C1 route is:

Signal	Pad and mux	Input selector
I2C1 SCL	UART4_TX_DATA, ALT2; mux `0x020E00B4`	`0x020E05A4`, daisy 1
I2C1 SDA	UART4_RX_DATA, ALT2; mux `0x020E00B8`	`0x020E05A8`, daisy 2
I2C1 clock gate	CCGR2.CG3, bits 7:6 at `0x020C4070`	Gate only; does not set the bit rate

These are the UART4 pads, not the other valid I2C routes on GPIO1_IO02/03. Configure both mux registers with SION (Software Input On), the input daisies, and suitable open-drain pad settings. I2C must observe SDA and SCL even while driving them. MINI sheet 1 shows 4.7 kOhm pull-up components on the I2C nets; verify fitted options, rail voltage and the attached module's additional pull-ups. UART4 cannot simultaneously own these pads.

I2C1 base is `0x021A0000`. Its registers are **16-bit**, spaced four bytes apart; the fragment uses halfword accesses and does not access the reserved halfwords between them.

Register	Offset	Purpose
`IADR`	`+0x00`	Controller's own slave address, not the target's address
`IFDR`	`+0x04`	Non-linear divider code
`I2CR`	`+0x08`	Enable, master/receive/transmit controls

Register	Offset	Purpose
`I2SR`	`+0x0C`	Busy, completion, arbitration and ACK status
`I2DR`	`+0x10`	Data; receive reads also advance the state machine

The bit rate is derived from **PERCLK_ROOT**, not simply the CPU or IPG frequency. Inspect CCM_CSCMR1's source and divider without changing the shared root under a running GPT. Table 31-3 gives these standard-mode examples:

Actual PERCLK_ROOT	IFDR	Divider	Nominal SCL
24 MHz	`0x0F`	240	100 kHz
49.5 MHz	`0x37`	512	About 96.7 kHz
66 MHz	`0x16`	768	About 85.9 kHz

These are calculations, not measured waveforms. Rise times and controller sampling affect the bus. In particular, `0x1E` means **3072**, and `0x15` means **640**. For a fast-mode design, supplied erratum **ERR007805** limits the chosen rate to 384 kHz or less to meet SCL-low time. Do not assume that programming 400 kHz meets the specification.

A one-byte random read has one continuous transaction:

```
START -> address+write -> ACK -> word address -> ACK -> repeated START
      -> address+read -> ACK -> receive one byte -> master NACK -> STOP
```

In receive mode the first, dummy read of I2DR **starts receiving**; it is not the wanted byte. After completion, request STOP before reading the final byte, so that this read does not initiate another receive cycle. Set TXAK before the dummy read because this one byte is also the last byte.

`i2c.c`, a **single-master polled fragment**, with no IRQ/DMA or automatic stuck-bus recovery. Supply a working, free-running `gpt_now_us()` as in Chapter 16; it must return microseconds and must not depend on an IRQ we have masked. All waits also have an iteration guard, so a stopped timer does not make them infinite. The iteration guard is a fail-safe, not a calibrated time interval.

```
#include <stdint.h>

#define I2C16(off) (*(volatile uint16_t *)(uintptr_t)(0x021A0000u + (off)))
#define IADR I2C16(0x00u)
#define IFDR I2C16(0x04u)
#define I2CR I2C16(0x08u)
#define I2SR I2C16(0x0Cu)
#define I2DR I2C16(0x10u)
#define CR_IEN  0x80u
#define CR_MSTA 0x20u
#define CR_MTX  0x10u
#define CR_TXAK 0x08u
#define CR_RSTA 0x04u
#define SR_IBB  0x20u
#define SR_IAL  0x10u
#define SR_IIF  0x02u
#define SR_RXAK 0x01u
#define POLL_GUARD 1000000u
#define I2C_WAIT_US 10000u

enum { I2C_TIMEOUT = -1, I2C_NACK = -2, I2C_ARB_LOST = -3,
       I2C_STOP_FAILED = -4, I2C_BAD_ARG = -5 };
uint32_t gpt_now_us(void); /* Existing timer helper, not defined here. */

static int delay_checked(uint32_t us)
{
    uint32_t t = gpt_now_us();
    for (unsigned n = 0; n < POLL_GUARD; ++n)
        if ((uint32_t)(gpt_now_us() - t) >= us) return 0;
    return I2C_TIMEOUT;
}

static int i2c_wait_bus(int busy)
{
    uint32_t t = gpt_now_us();
    for (unsigned n = 0; n < POLL_GUARD; ++n) {
        uint16_t s = I2SR;
        if (s & SR_IAL) return I2C_ARB_LOST;
        if (!!(s & SR_IBB) == busy) return 0;
        if ((uint32_t)(gpt_now_us() - t) >= I2C_WAIT_US) break;
    }
    return I2C_TIMEOUT;
}

static int i2c_wait_byte(uint16_t *status)
{
    uint32_t t = gpt_now_us();
    for (unsigned n = 0; n < POLL_GUARD; ++n) {
        uint16_t s = I2SR;
        if (s & SR_IAL) return I2C_ARB_LOST;
        if (s & SR_IIF) {
            *status = s;          /* Save ACK evidence before clearing. */
            I2SR = SR_IAL;        /* W0C: clear IIF, preserve IAL. */
            return 0;
        }
        if ((uint32_t)(gpt_now_us() - t) >= I2C_WAIT_US) break;
    }
    return I2C_TIMEOUT;
```

```
}

static int i2c_send(uint8_t byte)
{
    uint16_t s;
    I2DR = byte;
    int rc = i2c_wait_byte(&s);
    if (rc) return rc;
    return (s & SR_RXAK) ? I2C_NACK : 0;
}

int i2c_init(uint32_t perclk_hz)
{
    uint16_t divider;
    switch (perclk_hz) {
    case 24000000u: divider = 0x0Fu; break;
    case 49500000u: divider = 0x37u; break;
    case 66000000u: divider = 0x16u; break;
    default: return I2C_BAD_ARG;   /* Derive another code from Table 31-3. */
    }
    /* Clock/pads already configured, controller exclusively owned. */
    I2CR = 0;                    /* Disable/reset local controller. */
    IADR = 0;
    IFDR = divider;
    I2SR = 0;                    /* W0C: deliberately clear IIF and IAL. */
    I2CR = CR_IEN;
    __asm__ volatile ("dsb sy" ::: "memory");
    int rc = delay_checked(50);   /* Conservative enable-settling interval. */
    if (rc) I2CR = 0;
    return rc;
}

int i2c_read_byte(uint8_t addr7, uint8_t word_addr)
{
    if (addr7 < 0x08u || addr7 > 0x77u) return I2C_BAD_ARG;
    if (!(I2CR & CR_IEN)) return I2C_BAD_ARG;
    int rc = i2c_wait_bus(0);
    if (rc) return rc;            /* Do not start on an already-busy bus. */
    I2SR = 0;
    I2CR = CR_IEN | CR_MSTA | CR_MTX;
    rc = i2c_wait_bus(1);         /* Confirm START took effect. */
    if (rc) goto fail;
    rc = i2c_send((uint8_t)(addr7 << 1));
    if (rc) goto fail;
    rc = i2c_send(word_addr);
    if (rc) goto fail;

    I2CR = CR_IEN | CR_MSTA | CR_MTX | CR_RSTA;
    __asm__ volatile ("dsb sy" ::: "memory");
    rc = delay_checked(2);       /* RM 31.6: >= two module clocks after RSTA. */
    if (rc) goto fail;
    rc = i2c_send((uint8_t)((addr7 << 1) | 1u));
    if (rc) goto fail;

    I2CR = CR_IEN | CR_MSTA | CR_TXAK;
    (void)I2DR;                  /* Dummy read starts the receive. */
    uint16_t s;
    rc = i2c_wait_byte(&s);       /* RXAK is not a slave ACK check here. */
    if (rc) goto fail;
```

```
    I2CR = CR_IEN;              /* Request STOP before consuming final byte. */
    uint8_t value = (uint8_t)I2DR;
    rc = i2c_wait_bus(0);
    if (rc) { I2CR = 0; return I2C_STOP_FAILED; }
    return value;

fail:
    /* MSTA is cleared by hardware on arbitration loss: do not force STOP. */
    if (rc == I2C_ARB_LOST || (I2SR & SR_IAL)) {
        I2CR = 0;
        return I2C_ARB_LOST;
    }
    I2CR = CR_IEN;              /* Release mastership on NACK/timeout. */
    if (i2c_wait_bus(0)) {
        I2CR = 0;
        return I2C_STOP_FAILED;
    }
    return rc;
}
```

The `fail:` label is part of the C fragment. The timeout policy is deliberately simple: return a distinguishable error, release local mastership where possible, and disable the controller if release fails. Disabling it cannot make an external device release SDA/SCL. Investigate the bus before retrying; multi-master arbitration and GPIO clock-pulse recovery need additional policy.

IIF and IAL are **write-zero-to-clear (W0C)** on this i.MX6ULL controller. The direct IIF-clear write sends zero to IIF and one to IAL, leaving IAL unchanged. A read-modify-write of a status register can lose an event that arrives between the read and write. Compare this with SPI's W1C flag in the next section.

Application fragment, after clock/pad preparation and derivation of `board_perclk_hz`:

```
int rc = i2c_init(board_perclk_hz);
if (rc != 0) printf("I2C setup error %d\r\n", rc);
else {
    int v = i2c_read_byte(0x50, 0x00);
    if (v < 0) printf("EEPROM transaction error %d\r\n", v);
    else printf("EEPROM[0] = 0x%02x\r\n", (unsigned)v);
}
```

A successful read returns the stored byte. `0xFF` is a possible value, not proof of a blank EEPROM or even a correctly identified part. If there is no attached compatible device at `0x50`, a NACK is an ordinary result to report, not a reason to wait forever.

18.3 SPI, read flash JEDEC ID

ECSPI has separate transmit and receive FIFOs and four selectable channels. This example uses **ECSPI1 channel 0**, only if your documented attachment actually reaches those signals. MINI's schematic also labels ECSPI3 nets; those do not become ECSPI1 by

changing the base address in a driver. Trace the chosen controller, pad mux, input daisy, connector and flash pin as one route. Check for camera or other ownership conflicts on alternate pads.

ECSPI1 base is `0x02008000`; its gate is CCGR1.CG0, bits 1:0 at `0x020C406C`. The serial root comes through CCM_CSCDR2's ECSPI_CLK_SEL/PODF, from `pll3_60m` or the oscillator, not an assumed 80 MHz clock. This fragment requires an already established **24 MHz ECSPI root**, with the root configured while affected controllers are gated. Do not change a shared root beneath another active controller.

Register	Offset	Purpose
RXDATA	+0x00	Pop one RX FIFO word; read only when RR is set
TXDATA	+0x04	Push a TX FIFO word
CONREG	+0x08	Enable, channel/master, dividers, burst length, start control
CONFIGREG	+0x0C	Per-channel clock phase/polarity and chip-select waveform
INTREG	+0x10	Interrupt enables
DMAREG	+0x14	DMA enables
STATREG	+0x18	FIFO status; TC/RO are W1C
PERIODREG	+0x1C	Inter-burst timing
TESTREG	+0x20	Test controls, including loopback

Here is the transaction we want:

```
/CS low -> send 9F -> clock manufacturer -> clock type -> clock capacity -> /CS high
            8 bits          8 bits           8 bits          8 bits
```

Four independent eight-bit bursts would normally release `/CS` four times. Instead, load one word `0x9F000000` and request a **32-bit burst**. The controller shifts MSB first, keeping `/CS` asserted for the command and three response bytes. The first received byte corresponds to the command phase and is discarded.

Controller fragment; it reuses `gpt_now_us()`, `delay_checked()` and `POLL_GUARD` from §18.2, which can be shared helpers even if you skip the I2C transaction:

```
#define SPI32(off) (*(volatile uint32_t *)(uintptr_t)(0x02008000u + (off)))
#define SPI_RX       SPI32(0x00u)
#define SPI_TX       SPI32(0x04u)
#define SPI_CON      SPI32(0x08u)
#define SPI_CONFIG   SPI32(0x0Cu)
#define SPI_INT      SPI32(0x10u)
#define SPI_DMA      SPI32(0x14u)
#define SPI_STAT     SPI32(0x18u)
#define SPI_PERIOD   SPI32(0x1Cu)
#define SPI_TC       (1u << 7)
#define SPI_RO       (1u << 6)
#define SPI_RR       (1u << 3)
#define SPI_XCH      (1u << 2)
/* 24 MHz / ((11+1) * 2^1) = 1 MHz; CH0 master, EN=1, SMC=0. */
#define SPI_CON_IDLE ((31u << 20) | (11u << 12) | (1u << 8) \
                      | (1u << 4) | 1u)

int spi_init(void)
{
    /* Root, gate and all four pad routes prepared; /CS has a pull-up. */
    SPI_CON = 0;                 /* Reset FIFOs/internal logic. */
    SPI_CON = SPI_CON_IDLE;
    SPI_INT = 0;
    SPI_DMA = 0;
    SPI_PERIOD = 0;
    SPI_CONFIG = 0;              /* Mode 0, active-low SS0, SS_CTL0=0. */
    SPI_STAT = SPI_TC | SPI_RO;  /* W1C, clear only the two named flags. */
    __asm__ volatile ("dsb sy" ::: "memory");
    int rc = delay_checked(2);   /* Allow two 1 MHz SCLK periods to settle. */
    if (rc) SPI_CON = 0;
    return rc;
}

int spi_read_jedec(uint8_t out[3])
{
    if (!out || SPI_CON != SPI_CON_IDLE) return -1;
    /* Fresh init/previous completed read: RX FIFO must be empty. */
    if (SPI_STAT & SPI_RR) { SPI_CON = 0; return -1; }
    if (delay_checked(2)) { SPI_CON = 0; return -1; } /* /CS-high gap */
    SPI_STAT = SPI_TC | SPI_RO;
    SPI_TX = 0x9F000000u;
    SPI_CON = SPI_CON_IDLE | SPI_XCH; /* Manual start, one complete burst. */
    uint32_t t = gpt_now_us();
    for (unsigned n = 0; n < POLL_GUARD; ++n) {
        uint32_t s = SPI_STAT;
        if (s & SPI_RO) break;
        if ((s & (SPI_TC | SPI_RR)) == (SPI_TC | SPI_RR)
            && !(SPI_CON & SPI_XCH)) {
            uint32_t rx = SPI_RX;
```

```
            SPI_STAT = SPI_TC;   /* Direct W1C; no status RMW. */
            out[0] = (uint8_t)(rx >> 16);
            out[1] = (uint8_t)(rx >> 8);
            out[2] = (uint8_t)rx;
            return 0;
        }
        if ((uint32_t)(gpt_now_us() - t) >= 10000u) break;
    }
    SPI_CON = 0;                /* Abort/reset; reinitialize before retry. */
    return -1;
}
```

`SMC` selects automatic start on TX FIFO writes; it does **not** mean chip-select hold. We leave it clear and explicitly set XCH after queuing the word. `SS_CTL0` is bit 8, `SS_POL0` bit 12, and `SCLK_PHA0` bit 0 of CONFIGREG. All three are zero here: one burst, active-low select, phase 0. The checked delay also leaves a conservative select-high interval between commands; verify setup, hold and deselect requirements for your actual part. An interrupt handler must not consume these FIFOs while the polled fragment owns them.

Supplied erratum **ERR009606** forbids master burst lengths of `32n+1` **bits**, such as 33 bits. BURST_LENGTH holds bits-minus-one; our field value 31 requests 32 bits and avoids that case. ERR009535 concerns slave-mode burst termination, not this master transaction.

Application fragment:

```
uint8_t id[3];
if (spi_init() != 0 || spi_read_jedec(id) != 0)
    printf("SPI transaction failed\r\n");
else
    printf("JEDEC ID: %02x %02x %02x\r\n",
           (unsigned)id[0], (unsigned)id[1], (unsigned)id[2]);
```

The Winbond W25Q32JV Rev. J datasheet lists `EF 40 16` for IQ/JQ variants and `EF 70 16` for IM/JM variants. Check the full part marking. This is expected identification data, not a recorded board result. All-zero/all-one responses merit checks of MISO, `/CS`, device state and power; a completed controller transfer alone does not prove a flash answered.

18.4 eLCDIF, draw a color bar

An RGB panel adds a second clocked system: it consumes pixels continuously. The supplied MINI has LCD-related nets, but their existence does not identify a connected panel, its timing or its power sequence. This design must not be operated without a matching panel/carrier datasheet and approved wiring/power arrangement.

The panel defines the controller’s requirements:

- Supplies, reset, display-enable, backlight control, signal voltage and pad ownership constrain startup. The backlight/display must remain disabled through setup as required by the panel.
- The pixel clock may come from PLL5. CCM_CSCDR2 selects the LCDIF pre-source/pre-divider, CCM_CBCMR contains LCDIF1_PODF, and CCGR2.CG14 gates the LCD clocks. These controls are not all in one divider register.
- Sync polarities, pulse widths, porches, totals and active size define the scan. In DOTCLK RGB mode, VDCTRL0-4 describe that timing. LCDIF_TIMING instead serves the MPU/VSYNC interfaces.
- **Memory pixel format**, byte packing/swizzle and output bus width must agree. A `uint32_t` per pixel is not packed three-byte RGB888.
- The handoff publishes a reserved, DMA-visible framebuffer address before scanout starts. Controller start and panel enable follow the panel’s documented power sequence.

This example chooses an 800 by 480 **XRGB8888-style memory layout**: one word `0x00RRGGBB` per pixel, lower three bytes valid, little-endian/no swizzle, WORD_LENGTH=3 (24-bit input), BYTE_PACKING_FORMAT=7 and a 24-bit RGB output bus. Verify channel routing against the actual panel. The X byte is ignored here, not an alpha channel used for blending.

Framebuffer preparation fragment, not LCD initialization:

```
#include <stdint.h>

static uint32_t framebuffer[800 * 480] __attribute__((aligned(64)));

void fill_color_bars(void)
{
    static const uint32_t colors[8] = {
        0x00FFFFFF, 0x00FF00FF, 0x00FFFF00, 0x0000FFFF,
        0x00FF0000, 0x0000FF00, 0x000000FF, 0x00000000
    };
    for (unsigned y = 0; y < 480; ++y)
        for (unsigned x = 0; x < 800; ++x)
            framebuffer[y * 800 + x] = colors[x / 100];
}
```

It occupies **1,536,000 bytes**, about 1.46 MiB, with a 3200-byte row. Reserve that space in initialized DRAM without overlapping the image, stacks, table or test buffers. The 64-byte alignment also meets LCDIF’s double-word address alignment requirement.

The remaining sequence is **non-runnable pseudocode**, because panel-specific timing, reset/clock sequencing and completion checks are intentionally not supplied:

```
with LCDIF stopped and no scanout owning the framebuffer:
    configure board clocks/pads and bounded reset completion
    configure panel timing, DOTCLK/master mode, format and transfer count
    fill_color_bars()
    if dcache_clean_range(framebuffer, sizeof framebuffer) fails: stop
    # clean_range completes maintenance to PoC with DSB
    CUR_BUF  (0x021C8040) <- framebuffer's device-visible address
    NEXT_BUF (0x021C8050) <- same address
    order register setup, then set RUN using the documented control/SET alias
    enable panel/backlight according to the panel's power sequence
```

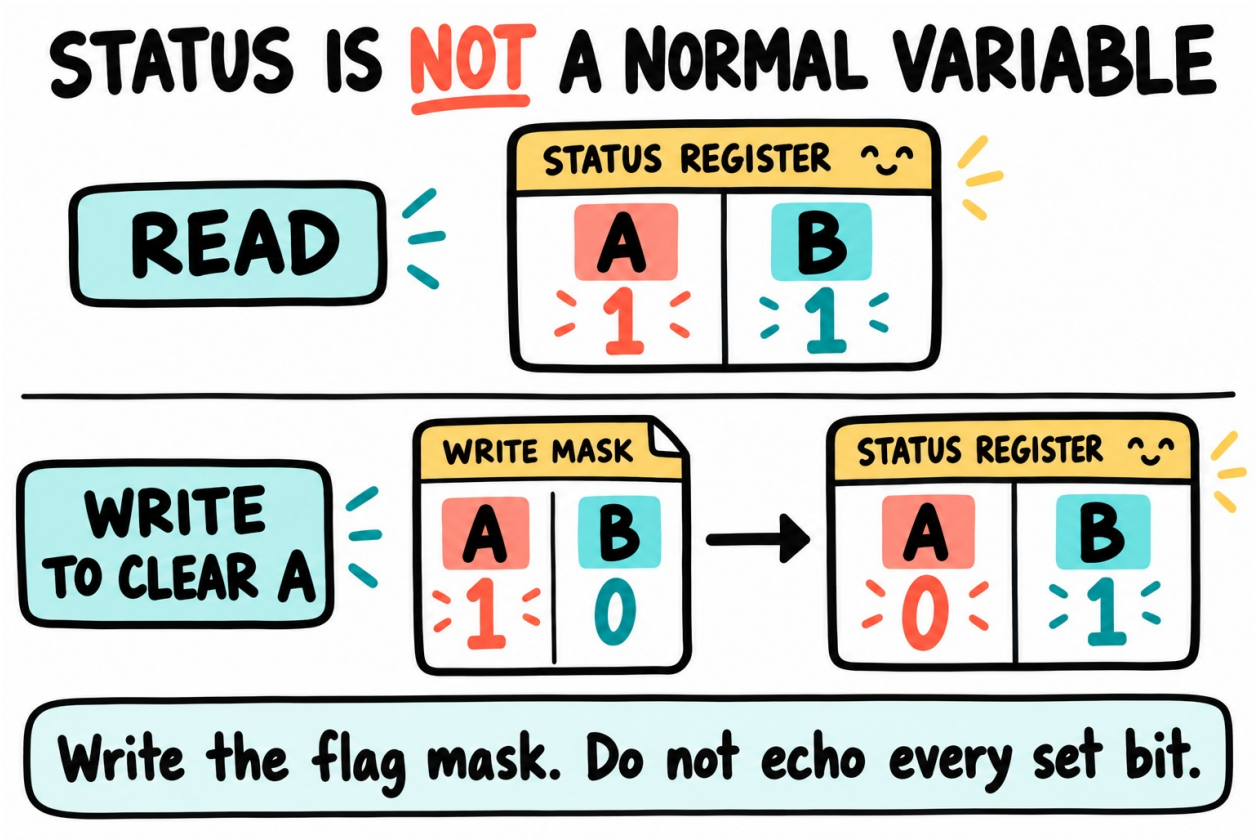
CUR_BUF and NEXT_BUF contain addresses, not CPU pointers in a general virtual-memory system. This LCD example requires separately qualified DDR and the matching `DDR_MIB` opt-in from Chapter 17. The OCRAM-only MMU example does not provide space for this framebuffer. A cast works here only with the identity mapping and a verified i.MX6ULL DMA route to that physical RAM.

Cache visibility and ownership are separate checks. LCDIF does not snoop the CPU’s cache on this path. Clean cached framebuffer writes to PoC and complete that maintenance **before** publishing a new buffer address. Alternatively, reserve appropriately mapped **Normal non-cacheable** RAM; do not use Device memory as the default type for a framebuffer. Do not change a live section’s type or create a cached/non-cacheable alias without the required transition.

For animation, fill and clean a buffer not being scanned, then queue it in NEXT_BUF. In DOTCLK mode LCDIF adopts NEXT_BUF at a frame boundary. Confirm the switch/completion condition before reusing the old buffer. Cleaning an actively scanned buffer may make new pixels visible but does not prevent tearing, and a 16 ms delay does not establish ownership or a measured refresh period.

Under Linux, `dma_alloc_coherent()` returns a CPU pointer and a device DMA address with a coherency contract; it is not universally just a non-cached mapping. Streaming DMA mappings use their own sync/unmap rules. Coherent memory still needs correct publication barriers and display-buffer lifetime handling. See the Linux DMA API guide.

18.5 The driver shape that repeats



For a W1C register, the write is a request to clear flags, not a replacement status value. Write only the intended clear mask. ECSPI's TC and RO use this behavior. The I2C flags above are W0C and need a different clear value.

A useful checklist is broader than the happy-path register writes:

```
1. establish ownership and validate board/device requirements
2. prepare power, clock source/gate and pad/input/electrical routes
3. reset or quiesce the controller, with bounded completion checks
4. configure mode, rate, format and clear only the intended status
5. transfer: preserve transaction boundaries and handle errors
6. confirm completion, return buffer ownership, and release resources
```

An ordinary exclusively owned RW configuration field may permit masked read-modify-write. A FIFO read, a self-clearing command or a W1C/W0C status field has a different contract. Read the field description before choosing the C operator.

Linux separates these responsibilities. An I2C adapter driver operates the controller; an EEPROM client driver requests transactions through it. SPI controller and flash drivers have a similar split. Clock, pinctrl, regulator, locking and DMA APIs coordinate shared resources. You will recognize the hardware sequence, but not every subsystem callback writes a controller register, and not every driver has an identical initialization order.

18.6 Why the bare-metal model ends here

Part II gives you a reference model for the boot path: image layout, UART, clocks, DDR, exceptions, timers and translation/cache setup. This optional chapter adds two transaction fragments and a display sketch. Building them is not evidence that a complete production stack has run on your board.

U-Boot now supplies a maintained bootloader structure around these mechanisms. Linux adds process protection, resource ownership and subsystem interfaces. Its early boot does not simply repeat every U-Boot DDR or pin setup. The useful question changes from “which register do I write?” to “which layer owns this request, and what state does it require?”

You could continue building bare-metal networking, USB or filesystems, or adopt an RTOS. Those are substantial separate projects, not a few remaining register examples. For this book’s Linux goal, Part III is the next dependency.

The supplementary choices remain:

- **18A, Project organization:** separate register definitions, board choices, driver interfaces and build dependencies.
- **18B, Button + beep:** distinguish GPIO levels, qualified events and output timing.
- **18C, Bare-metal RTC:** examine SNVS timekeeping and the documented backup-power arrangement.

Read any combination, or proceed to Chapter 19. Keep the small transactions here available as something you can trace when a higher-level request fails.

18.7 Key Notes

- The controller’s presence does not establish an attached device. EEPROM address width, flash command set and panel timing belong to the actual populated part and routed board, not a generic i.MX6ULL peripheral name.
- An I2C random read preserves one transaction across a repeated START. Address `0x50` becomes `0xA0`/`0xA1` on the wire, and TXAK requests the master’s final NACK rather than reporting a slave ACK.
- I2C completion, NACK, arbitration loss and failed STOP have distinct recovery implications. Bounded polling limits waiting but cannot force an external device to release a stuck bus.
- SPI command and reply share the required chip-select interval. The single 32-bit MSB-first JEDEC burst preserves that boundary, and its low received byte is the last response byte. Controller completion alone does not establish device identity.

- W0C and W1C flags require different clear writes. Ordinary configuration read-modify-write is not a general rule for status registers or FIFO accesses.
- LCD scan timing, memory packing and device addressing must agree with the panel and memory map. Cache cleaning makes pixels visible, while frame-boundary switching and confirmed completion determine which buffer the CPU may reuse.
- Linux controller, client and resource-management layers divide these responsibilities among owners. A bare-metal fragment needs exclusive hardware ownership and is not safe beside a live Linux driver.

The EEPROM and flash transactions above are reads, not update procedures. EEPROM writes require a disposable part or explicitly approved scratch location, preservation of old contents, and write-protect, page-boundary, write-cycle and bounded ACK-polling handling. Do not overwrite byte zero of an unidentified board EEPROM or power-cycle during a write. Do not short or disconnect a powered bus to force a timeout.

18.8 Pitfalls

- **I2C flag semantics:** IIF/IAL are W0C here. ECSPI TC/RO are W1C. Using the wrong clear value can leave a stale flag or erase unrelated evidence.
- **I2C early return:** a NACK path still needs to release mastership; arbitration loss is different because hardware has already cleared MSTA. A stuck bus needs recovery policy, not endless retries.
- **I2C repeated START:** obey RM §31.6’s delay before writing I2DR. A barrier orders accesses but does not itself promise two elapsed module-clock cycles.
- **SPI transaction boundaries:** SMC is start control, not chip-select hold. A flash command and reply must share the required select interval.
- **SPI FIFO/byte order:** read RXDATA only when RR is set. For our 32-bit MSB-first burst, response bytes occupy bits 23:16, 15:8 and 7:0; masking with `0xFF` gives the last one.
- **LCD visibility versus tearing:** maintenance, physical/device addressing, scan timing and buffer ownership all matter. None is replaced by an arbitrary delay.
- **Missing route or clock:** a gate does not choose the source/rate; a mux does not necessarily configure the input daisy or electrical properties. Check the entire path before blaming the peripheral protocol.
- **Busy waits without an exit:** this chapter bounds status polling and treats timer failure as an error. The separate fatal idle loops in the application skeleton are intentional, not peripheral-completion loops.

18.9 Going deeper

- Supplied **IMX6ULLRM Rev. 1**: Chapter **31** (I2C), **20** (ECSPI), **34** (eLCDIF), plus **18** (CCM) and **32** (IOMUXC). The book's chapter numbers are not the RM's chapter numbers.
- Supplied **MINI V2.2 schematic**, sheet 1 for I2C nets/pull-ups; **CORE V2.0 schematic**, sheets 6/8 for the UART4/I2C route. Check the actual populated revision before using these routes.
- Supplied **IMX6ULL errata Rev. 1.2**: ERR007805 (I2C low period), ERR009606 (SPI master burst length) and ERR009535 (SPI slave burst termination).
- Linux v6.12 `i2c-imx.c`: controller state, timeouts, recovery and SoC-specific flag semantics.
- Linux v6.12 `spi-imx.c`: controller transfers, FIFO handling, clock setup and chip-select policy.
- Linux v6.12 DRM `mxsfb`: display modes and scanout-buffer handling, rather than a universal list of panel writes.

The core bare-metal model is complete.

You now have mechanisms to inspect, and explicit boundaries where board evidence is still required. Carry that habit into U-Boot: trace what the code owns, what it assumes at entry and what its completion check actually proves.

> **Next, choose:** Chapters **18A-18C** for supplementary design cases, or **Part III, Chapter 19, U-Boot, from source, first boot.**

Chapter 18A: Project organization

You want to add a button, but opening `main.c` also brings up clock setup, UART registers, and the interrupt handler. Which parts belong to the new feature, and which must remain untouched? A project layout should make that decision easier without changing what the board does.

The example layout puts drivers into a small **board support package (BSP)** and gives their register definitions one home. This is the same separation you may have used with an MCU HAL: application code calls a driver; the driver knows the controller and the board wiring. No Linux service or device tree is involved in these bare-metal programs.

This chapter examines a refactor of the Chapters 10-17 design, not a new standalone firmware download or a required migration. Startup, linker layout and boot-image wrapper form a contract that remains intact when the C code moves. The following header is a useful subset, not every register needed by those earlier chapters.

18A.1 The problem we are solving

A project using the Chapter 16 or 17 mechanisms may have:

- `main.c`: `main()` plus inline UART init, GPIO init, CCM writes
- `startup.S`, `link.ld`, unchanged
- A growing list of `#define UART_UCR1 0x02020080` and friends, scattered across files
- Function names like `uart_init`, `gpio_init`, `epit_init`, whose declarations need a consistent home.

Two specific kinds of pain start to appear:

1. **Duplicated hardware facts.** `uart.c` defines `UART_UCR1`; `main.c` independently defines it with a typo. Separate C files are separate translation units, so both can compile and use different addresses. In a single translation unit, an incompatible macro redefinition normally produces a diagnostic. Neither situation is a reason to maintain two copies.
2. **Reuse friction.** To use the Chapter 18 I²C driver in a new project, you copy `i2c.c`, plus the relevant `#define`s from `main.c`, plus the CCM gate bit, plus the IOMUX writes. That can mean five files per peripheral.

Separate **the SoC register map** (`imx6ull.h`) from **driver behavior** (one folder per peripheral). Keep board choices, such as which pad drives KEY0, named separately from generic GPIO offsets. In this small project they can be a clearly marked board section of the same header. A new board then changes those choices rather than pretending that all i.MX6ULL boards are wired alike.

18A.2 Target layout

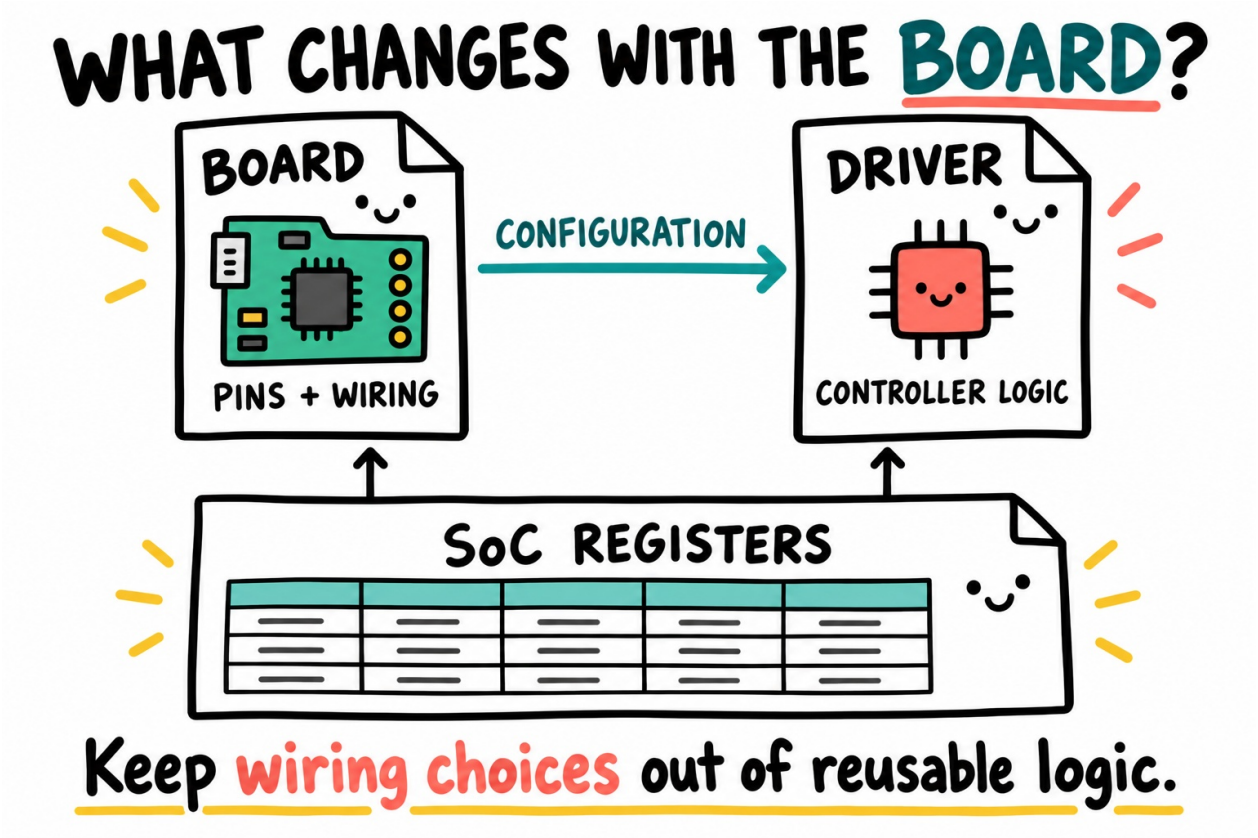
```
bare-metal/
├── Makefile                    # top-level build
├── link.ld                     # unchanged
├── startup.S                   # unchanged
├── imx6ull.h                   # shared register definitions subset
├── main.c                      # only application logic
└── bsp/
    ├── clk/
    │   ├── bsp_clk.h
    │   └── bsp_clk.c           # was clocks.c
    ├── gpio/
    │   ├── bsp_gpio.h
    │   └── bsp_gpio.c
    ├── uart/
    │   ├── bsp_uart.h
    │   └── bsp_uart.c
    ├── int/                    # interrupts / GIC
    │   ├── bsp_int.h
    │   ├── bsp_int.c
    │   └── irq_entry.S         # the IRQ entry asm from Ch 15
    ├── gpt/
    │   ├── bsp_gpt.h
    │   └── bsp_gpt.c
    ├── epit/
    │   ├── bsp_epit.h
    │   └── bsp_epit.c
    └── delay/
        ├── bsp_delay.h
        └── bsp_delay.c
```

Conventions:

- **`bsp_<peripheral>.h`** has only public-facing declarations: the API functions, and the *enum types* the API uses. No raw register addresses.
- **`bsp_<peripheral>.c`** has the function bodies. Includes `imx6ull.h` for register addresses.
- **`imx6ull.h`** is the *only* place that names registers. Every other file uses those names.

Include each driver's own header in its implementation too. That lets the compiler compare the public declaration with the definition. A driver may also include another driver's public header when it actually uses that API; for example, a buzzer driver may depend on the delay driver.

18A.3 Writing `imx6ull.h`



The register map describes the SoC. The board chooses how its signals are wired. Keep those choices visible when moving code into a driver, even if this small project stores both sets of definitions in one header.

The file is long but mechanical. Section it by peripheral block:

```
#ifndef IMX6ULL_H
#define IMX6ULL_H

#include <stdint.h>
#define REG(addr) (*(volatile uint32_t *)(uintptr_t)(addr))

/* ============================================================
 * CCM — Clock Controller Module (RM ch. 18)
 * Base: 0x020C4000
 * ============================================================ */
#define CCM_BASE        0x020C4000U
#define CCM_CCR         (CCM_BASE + 0x00)
#define CCM_CACRR       (CCM_BASE + 0x10)
#define CCM_CBCDR       (CCM_BASE + 0x14)
#define CCM_CBCMR       (CCM_BASE + 0x18)
#define CCM_CSCMR1      (CCM_BASE + 0x1C)
#define CCM_CSCDR1      (CCM_BASE + 0x24)
#define CCM_CCGR0       (CCM_BASE + 0x68)
#define CCM_CCGR1       (CCM_BASE + 0x6C)
#define CCM_CCGR2       (CCM_BASE + 0x70)
#define CCM_CCGR3       (CCM_BASE + 0x74)
#define CCM_CCGR4       (CCM_BASE + 0x78)
#define CCM_CCGR5       (CCM_BASE + 0x7C)
```

```
#define CCM_CCGR6       (CCM_BASE + 0x80)
#define CCM_CDHIPR      (CCM_BASE + 0x48)

/* CCGR per-peripheral gates -- 2 bits each, 16 gates per CCGRx register */
#define CCGR_GPIO1_GATE   (3u << 26)   /* CCGR1[27:26] */
#define CCGR_GPIO5_GATE   (3u << 30)   /* CCGR1[31:30], see note below */
#define CCGR_GPT1_GATE    (15u << 20)  /* CCGR1[23:20]: serial AND bus */
#define CCGR_EPIT1_GATE   (3u << 12)   /* CCGR1[13:12] */
#define CCGR_UART1_GATE   (3u << 24)   /* CCGR5[25:24] */
#define CCGR_I2C1_GATE    (3u << 6)    /* CCGR2[7:6]   */

/* ============================================================
 * CCM analog PLLs and PFDs (RM ch. 18)
 * Base: 0x020C8000
 * ============================================================ */
#define ANATOP_BASE     0x020C8000U
#define ANATOP_PLL_ARM  (ANATOP_BASE + 0x000)
#define ANATOP_PLL_SYS  (ANATOP_BASE + 0x030)
#define ANATOP_PFD_528  (ANATOP_BASE + 0x100)
#define ANATOP_PFD_480  (ANATOP_BASE + 0x0F0)

/* ============================================================
 * GPIO1..GPIO5 (RM ch. 28)
 * ============================================================ */
#define GPIO1_BASE      0x0209C000U
#define GPIO2_BASE      0x020A0000U
#define GPIO3_BASE      0x020A4000U
#define GPIO4_BASE      0x020A8000U
#define GPIO5_BASE      0x020AC000U

/* Per-bank register offsets */
#define GPIO_DR_OFS     0x000
#define GPIO_GDIR_OFS   0x004
#define GPIO_PSR_OFS    0x008
#define GPIO_ICR1_OFS   0x00C
#define GPIO_ICR2_OFS   0x010
#define GPIO_IMR_OFS    0x014
#define GPIO_ISR_OFS    0x018
#define GPIO_EDGE_OFS   0x01C

/* Helper for indexed access */
#define GPIO_DR(bank)   REG((bank) + GPIO_DR_OFS)
#define GPIO_GDIR(bank) REG((bank) + GPIO_GDIR_OFS)
#define GPIO_PSR(bank)  REG((bank) + GPIO_PSR_OFS)

/* ============================================================
 * UART1 and UART2 subset (RM ch. 55)
 * ============================================================ */
#define UART1_BASE      0x02020000U
#define UART2_BASE      0x021E8000U
#define UART_URXD_OFS   0x000
#define UART_UTXD_OFS   0x040
#define UART_UCR1_OFS   0x080
#define UART_UCR2_OFS   0x084
#define UART_UCR3_OFS   0x088
#define UART_UCR4_OFS   0x08C
#define UART_UFCR_OFS   0x090
#define UART_USR1_OFS   0x094
#define UART_USR2_OFS   0x098
```

```
#define UART_UBIR_OFS   0x0A4
#define UART_UBMR_OFS   0x0A8
#define UART_UTS_OFS    0x0B4

/* ============================================================
 * GPT1, EPIT1, EPIT2 (RM ch. 30, 24)
 * ============================================================ */
#define GPT1_BASE       0x02098000U
#define EPIT1_BASE      0x020D0000U
#define EPIT2_BASE      0x020D4000U

/* ============================================================
 * IOMUXC (RM ch. 32)
 * ============================================================ */
#define IOMUXC_BASE     0x020E0000U
/* Pad mux + pad ctl offsets vary per pad; use the pad-specific addresses
 * directly, looked up from the RM IOMUX tables. */

/* ============================================================
 * GIC v2 (Cortex-A7 internal at 0x00A01000 / 0x00A02000)
 * ============================================================ */
#define GICD_BASE       0x00A01000U
#define GICC_BASE       0x00A02000U

#endif /* IMX6ULL_H */
```

Bring across the remaining definitions used by your existing drivers; do not delete one merely because it is absent from this subset. `REG` expresses a volatile MMIO access, not a memory barrier. If you retained Chapter 17's MMU, the peripheral ranges must still use appropriate Device memory attributes.

Two details deserve a source check. GPT1 has **bus and serial gates**, so the combined mask covers both. The supplied IMX6ULLRM Rev. 1 calls CCGR1 CG15 reserved, whereas the Linux v6.12 i.MX6UL/ULL clock driver defines GPIO5 at offset `0x6C`, shift 30. The GPIO5 mask above follows that implementation; it is not a literal transcription of the supplied RM table. Gate value `11` enables a clock in RUN and WAIT, not STOP.

18A.4 A peripheral driver, refactored

In this layout, Chapter 13's **read-only clock audit** belongs in `bsp/clk/bsp_clk.c`, includes `bsp_clk.h` instead of `clocks.h`, and uses register names consolidated in `imx6ull.h`. The following selective gate helper shares that module. It does not replace the audit or introduce a generic PLL transition routine:

```
#include "bsp_clk.h"
#include "imx6ull.h"

void clk_enable_lab_gates(void)
{
    REG(CCM_CCGR1) |= CCGR_GPIO1_GATE | CCGR_GPT1_GATE | CCGR_EPIT1_GATE;
    REG(CCM_CCGR5) |= CCGR_UART1_GATE;
}
```

The corresponding header:

```
#ifndef BSP_CLK_H
#define BSP_CLK_H
#include <stdint.h>

void clk_enable_lab_gates(void);
uint32_t clocks_get_arm_hz(void);
uint32_t clocks_get_ahb_hz(void);
uint32_t clocks_get_ipg_hz(void);
uint32_t clocks_get_uart_hz(void);
uint32_t clocks_get_mmdc_hz(void);

#endif
```

The audit function bodies still come from Chapter 13; a declaration alone does not supply them. In particular, the timer initializer needs the actual supported IPG rate, not an assumed 66 MHz value. The gate helper does not set UART baud rate or GPT prescalers, and enabling a gate is not permission to change a live clock root. Retain the approved ROM/board clock configuration for these examples; do not call a leftover `clk_init_main()` or `clocks_init()` that reprograms it without a board-qualified transition.

18A.5 The top-level Makefile

Use the Chapter 3 environment in a **Linux terminal** (including your configured WSL shell), not PowerShell. Keep the official Arm 13.2.Rel1 `arm-none-eabi` toolchain in its existing project-local directory; do not rename it, install a replacement system compiler, or edit `.bashrc`.

```
$ . ~/imx6ull/scripts/env.sh
$ command -v arm-none-eabi-gcc
$ arm-none-eabi-gcc --version
$ cd ~/imx6ull/src/bare-metal
```

Check that the path and version identify the selected no-OS compiler before building. The directory above is the layout in Section 18A.2; substitute your own project directory if you used a different name. Build as your normal user. Save the following as `Makefile`, with literal tabs at the start of recipe lines:

```
CROSS    ?= arm-none-eabi-
CC       := $(CROSS)gcc
OC       := $(CROSS)objcopy
SIZE     := $(CROSS)size

ARCHFLAGS := -mcpu=cortex-a7 -marm -mfloat-abi=soft -mgeneral-regs-only
CFLAGS    := $(ARCHFLAGS) -std=gnu11 -ffreestanding -fno-builtin \
             -fno-pie -fno-stack-protector -fno-common \
             -fno-unwind-tables -fno-asynchronous-unwind-tables \
             -O2 -g -Wall -Wextra -Werror=implicit-function-declaration
ASFLAGS   := $(ARCHFLAGS) -g
LDFLAGS   := $(ARCHFLAGS) -nostdlib -Wl,-T,link.ld,-Map,app.map
```

```
LDLIBS    := -lgcc

# One directory level of BSP sources; keep experiments outside this tree.
BSP_DIRS := $(patsubst %/,%,$(wildcard bsp/*/))
CPPFLAGS := -I. $(addprefix -I,$(BSP_DIRS))

BSP_C    := $(wildcard bsp/*/*.c)
BSP_S    := $(wildcard bsp/*/*.S)
TOP_C    := $(wildcard *.c)
TOP_S    := startup.S

OBJS     := $(BSP_C:.c=.o) $(BSP_S:.S=.o) $(TOP_C:.c=.o) $(TOP_S:.S=.o)

all: app.bin

%.o: %.c Makefile
    $(CC) $(CPPFLAGS) $(CFLAGS) -MMD -MP -c -o $@ $<

%.o: %.S Makefile
    $(CC) $(CPPFLAGS) $(ASFLAGS) -MMD -MP -c -o $@ $<

app.elf: $(OBJS) link.ld Makefile
    $(CC) $(LDFLAGS) -o $@ $(OBJS) $(LDLIBS)
    $(SIZE) $@

app.bin: app.elf Makefile
    $(OC) -O binary $< $@

clean:
    rm -f $(OBJS) $(OBJS:.o=.d) app.elf app.bin app.map

-include $(OBJS:.o=.d)

.PHONY: all clean
```

With unchanged inputs and outputs present, a second `make` invocation should not compile or link anything. `app.bin` is still only the payload. Any board transfer requires the matching ROM wrapper and its placement/size checks.

What the rules now account for:

- **Source discovery and includes agree.** Both cover `bsp/<driver>/`, one level deep. A new `bsp/key/` directory supplies both sources and an include path. Do not leave two alternative implementations or two `main` functions inside the discovered tree; avoid a `.c` and `.S` with the same basename, since both would map to one `.o`.
- **Header changes rebuild their users.** GCC's `-MMD -MP` creates one `.d` dependency file per object; `-include` reads it after the default target. A driver includes its own header and the headers of APIs it calls. Include-path visibility does not enforce architectural layering. GNU make's prerequisite guidance explains why header dependencies are needed.

- **Edited build flags are inputs too.** Objects and outputs depend on `Makefile`. Command-line overrides and a changed compiler executable/path are not automatically tracked: use `make clean` before rebuilding with those changes. Removing or renaming a discovered source also warrants a clean rebuild, since an old ELF does not record the previous source list for Make.
- **The integer-only contract is explicit.** ARM state, soft-float ABI, and `-mgeneral-regs-only` keep these C drivers out of VFP/NEON registers. Apply that policy to all C objects, including interrupt handlers; do not mix in hard-float objects. The same architecture flags reach the link step. `-nostdlib` removes automatic runtime libraries, so `-lgcc` follows the objects for compiler-generated helpers. It does not supply `printf`, `memcpy`, or a C startup.

The linker and startup remain part of the refactor's contract:

- For the fresh-ROM OCRAM route, place `.vectors` first at `0x00908000`, use `ENTRY(_vectors)`, and make its first instruction branch to `_start`. Set the wrapper's entry to that same vector address. Install VBAR with the required alignment and synchronization; keep the exception-base selection consistent with it.
- Keep `.data` and `.bss` ends padded to four bytes for the word-copy/word-clear loops. Reserve the Chapter 10 **4 KiB stack** explicitly and retain its overlap assertions. If IRQ support is included, retain its separately allocated mode stack and 8-byte C-call alignment too.
- Keep the complete ROM-loaded image, including wrapper overhead, within the chosen limit ending at or below `0x00918000`. Linker assertions for payload RAM alone do not check wrapper bytes; rerun the Chapter 11 image-size checks.
- Expect a fresh-ROM handoff, not a jump out of an arbitrary running Linux/U-Boot image. Do not assume that all caches are off. Preserve the startup's documented handoff preparation and, when retaining the MMU example, the peripheral memory attributes and cache/table maintenance. Moving a source file does not repair a handoff protocol.

If that maintenance uses set/way operations across L1 and L2, retain the DSB between cache levels required by supplied chip erratum **ERR008958**. An organized source tree does not remove silicon-specific sequencing.

18A.6 Cost of the refactor

You gain more files and more explicit dependencies. In return, a UART change has a predictable location, and an application can include `bsp_uart.h` without copying its register map. The tradeoff is clearer ownership at the cost of more files. There is no fixed line-count or time saving to promise.

A public-header change illustrates the build contract: every affected implementation must rebuild. Without that dependency, an organized layout still permits stale code. The `.d` files therefore belong to the refactor itself, not to a later cleanup.

18A.7 Sidebar, the NXP SDK alternative

An i.MX6ULL SDK package may provide `MCIMX6Y2.h`, with structures describing register offsets. Check the selected package and device, not just the header filename. SDK access and available releases can change; start at the NXP SDK portal. The abbreviated structure below illustrates the technique; it is **not a substitute SDK header**:

```
typedef struct {
    __IO uint32_t CCR;        /* 0x000 */
    __IO uint32_t CCDR;       /* 0x004 */
    __IO uint32_t CSR;        /* 0x008 */
    __IO uint32_t CCSR;       /* 0x00C */
    __IO uint32_t CACRR;      /* 0x010 */
    __IO uint32_t CBCDR;      /* 0x014 */
    __IO uint32_t CBCMR;      /* 0x018 */
    __IO uint32_t CSCMR1;     /* 0x01C */
    /* More fields are required, including padding before CCGR1 at 0x06C. */
} CCM_Type;

#define CCM_BASE          (0x020C4000u)
#define CCM               ((CCM_Type *)CCM_BASE)
```

With the complete device header, driver code becomes:

```
CCM->CCGR1 |= (3u << 26);   /* enable GPIO1 clock */
```

Compared to our:

```
REG(CCM_CCGR1) |= (3u << 26);
```

Both styles can generate the same volatile load/modify/store sequence when addresses, types, and flags match. Compare that sequence rather than assuming the entire ELF will be byte-identical. The struct version:

- Checks member names and access types: `CCM->CGGR1` is a compile error. An undefined macro name is also a compile error; neither style catches a wrong numeric offset merely by compiling.
- Plays nicely with debuggers: GDB displays `CCM` as a struct with named fields.
- Keeps offsets in the type definition. Use `offsetof(CCM_Type, CCGR1)` or a debugger to inspect them; `%p`, if your formatter supports it, requires a `void *` argument.

Vendor headers reduce transcription work, but they are not a guarantee that every field, revision, or board assumption is correct. Review their release notes and errata, and verify the few offsets your driver uses. Also carry over the header's required type definitions, include paths, and license. Introducing a header does not initialize clocks, pads, stacks, or the FPU.

18A.8 Key Notes

- SoC offsets, board wiring and driver behavior describe different ownership domains. A shared hardware header prevents duplicate definitions without making every board's routes interchangeable.
- Public headers define driver interfaces and are included by their own implementations. Include-path visibility makes names available but does not enforce dependency direction or prevent architectural cycles.
- Moving files does not replace startup, vector placement, stack budgets, image wrapping or cache-handoff requirements. Those contracts survive the refactor unchanged.
- Source discovery and object naming constrain the build. Two alternative implementations in the discovered tree, or a C and assembly source mapping to the same object name, create ambiguity rather than modularity.
- Header dependency files and Makefile prerequisites keep outputs tied to inputs. Compiler-path changes, command-line flag overrides and removed sources need separate clean-rebuild handling because these rules do not track them automatically.
- Raw macros and verified SDK structs can express equivalent register accesses. Correct offsets, access types and semantics matter more than identical ELF bytes, which can change with link order and alignment.
- The integer-only ABI and explicit libgcc link are system-wide contracts, including ISR callees. A gate helper supplies neither clock-root transitions nor a C runtime.

18A.9 Pitfalls

- **Confusing folder names with layering.** `lib/`, `drv/`, and `bsp/` are valid conventions. Here `bsp/` names our board support code; consistent ownership and includes matter more than the spelling.
- **Including drivers from `imx6ull.h`.** It reverses the intended dependency direction and can create include cycles. Keep the hardware description independent of driver APIs.

- **Assuming `static inline` always bloats code.** A small accessor can be appropriate; unused inline functions need not be emitted. Keep substantial driver behavior in `.c` so it has one implementation and a clear API.
- **Forgetting `-I` for every BSP folder.** Symptom: `bsp_clk.h: No such file or directory`. The `$(addprefix -I,$(BSP_DIRS))` line saves you.
- **Conflicting definitions during SDK migration.** The styles can coexist behind clear module boundaries, but duplicate base names and incompatible register types cannot. Convert one implementation deliberately; it is not a blind global replacement.
- **Duplicate vectors or IRQ entries.** Moving `irq_entry.S` must remove its old definition, not add another. Keep vector ordering, VBAR setup, and mode-stack allocation in the linked result.

18A.10 Going deeper

- **Linux kernel** `arch/arm/include/asm/io.h` and `arch/arm/include/asm/hardware/`, the kernel uses the same “one header per controller” pattern at scale.
- **U-Boot** `arch/arm/include/asm/arch-mx6/imx-regs.h`, register addresses for i.MX6 family in U-Boot.
- **A verified NXP i.MX6ULL SDK package**: inspect its `MCIMX6Y2.h`, supporting headers, release notes, and license before adopting the struct-based map.
- **GNU make manual, automatic prerequisites**: the header-rebuild mechanism used above. Check GCC’s `-MMD` and `-MP` descriptions alongside it.

> Next chapter: **Chapter 18B: Button input and beep.** The layout separates a board-specific input/output path from event filtering and timing policy.

Chapter 18B: Button input and beep

A held button can produce thousands of low samples. The application usually wants one **press event**, followed later by one **release event**. Before debouncing those transitions, follow the switch's wire to its GPIO input. Then each sample has a clear meaning: the level at that pad, at that moment.

The Chapter 18A layout separates the input and buzzer circuits' drivers from an application that samples the button at a fixed interval. Foreground polling keeps the event path visible. Interrupt-driven sampling is an alternative design, not a required implementation step.

18B.1 The hardware on the Point Atom MINI

These assignments are for the supplied **MINI v2.2** baseboard and the core schematic whose internal project title is **CL6Y2CB_V1.9**. Compare them with your fitted boards. They are not an assertion that every ALPHA revision has the same wiring. The Chapter 5 board table uses this same reference route.

Signal	Schematic path	Software meaning
KEY0	MINI sheet 2: switch to GND, R12 10 kOhm to DCDC_3V3; MINI sheet 1: IMX1 B48, UART1_CTS; CORE sheet 8: J2 pin 48	Pad UART1_CTS, ALT5, GPIO1 bit 18; released = 1, pressed = 0
BEEP	CORE sheet 8: J2 pin 5, SNVS_TAMPER1 / BEEP; MINI sheet 2: R21 1 kOhm to Q1 base, Q1 S8550 emitter to DCDC_3V3, collector to BEEP1 positive terminal, other terminal to GND	Pad SNVS_TAMPER1, ALT5, GPIO5 bit 1; low enables the PNP, high disables it
LED0	CORE sheet 8: J2 pin 47, GPIO_3 / LED0; MINI sheet 1: IMX1 B47; sheet 2: LED and R6 to DCDC_3V3	GPIO1 bit 3, active-low, as in Chapter 9

CORE J2 pin 48 carries the accessory alias `KEY2`, but MINI sheet 1 connects that pin to its onboard **KEY0**. Pin 49's core alias `KEY0` instead reaches MINI's `GBC_KEY / AP_INT` module signal. Follow physical connector pin numbers into the baseboard circuit before trusting an inherited button name. UART1 TX/RX remain our console; do not enable CTS hardware flow control on this pad while using it as the button input.

The S8550 is a **PNP high-side switch**. Pulling its base down through R21 allows load current to flow from 3V3 through the transistor; the GPIO sinks base current rather than supplying the buzzer directly. Idle must be high. The schematic names BEEP1 only as BEEP: it does **not** establish whether the fitted buzzer has an internal oscillator or whether it is piezoelectric. Check its part marking/BOM. An active buzzer needs an on/off envelope; the square-wave example in Section 18B.3 requires a verified passive transducer and suitable drive circuit.

Add these board-specific names **inside** `imx6ull.h`, before its closing `#endif`. GPIO_PSR, GPIO bank bases, and CCM names come from Section 18A.3:

```
/* MINI v2.2 / CL6Y2CB_V1.9 board routes. */
#define KEY_BIT             (1u << 18)
#define BEEP_BIT            (1u << 1)
#define LED_BIT             (1u << 3)
#define KEY_MUX             0x020E008Cu
#define KEY_PAD             0x020E0318u
#define BEEP_MUX            0x0229000Cu
#define BEEP_PAD            0x02290050u
#define LED_MUX             0x020E0068u
#define LED_PAD             0x020E02F4u
```

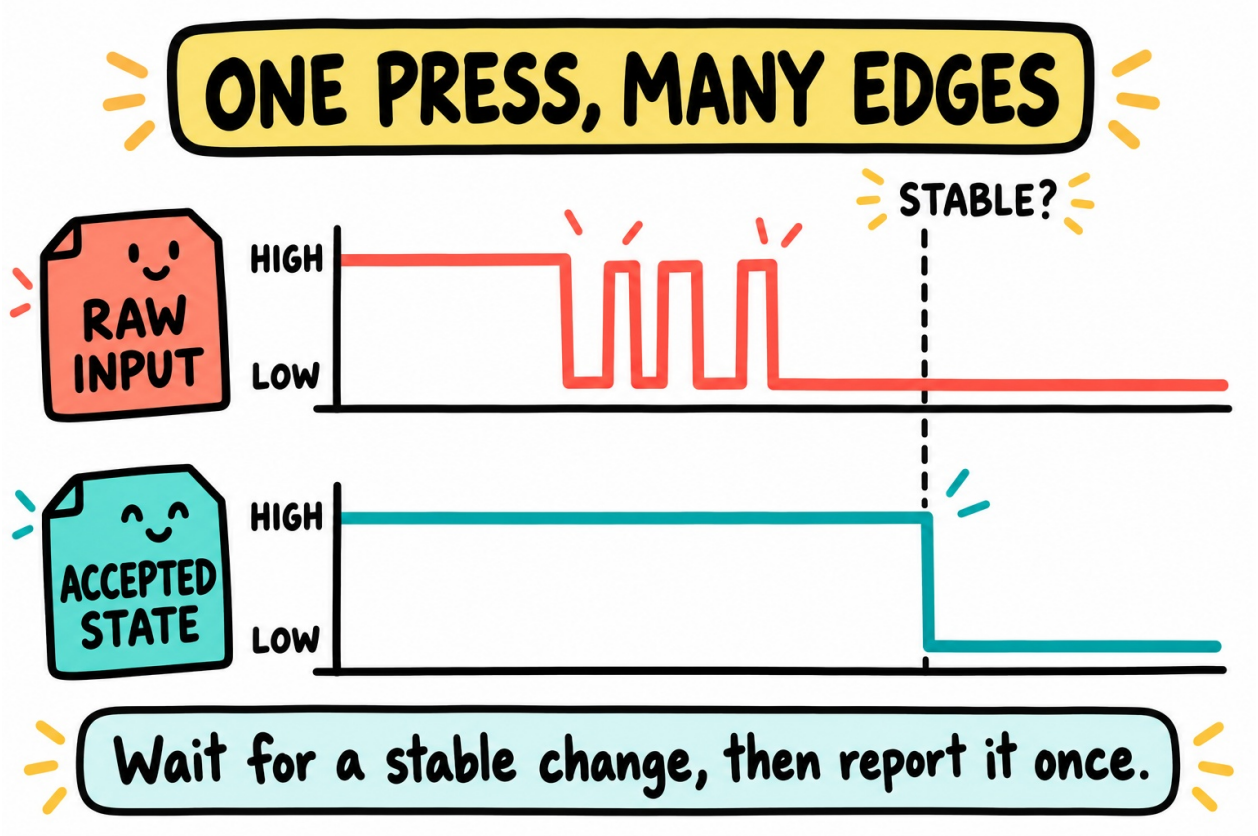
Find KEY0's `SW_MUX_CTL_PAD_UART1_CTS_B` and `SW_PAD_CTL_PAD_UART1_CTS_B` registers in RM Chapter 32; BEEP's registers are in Sections 32.5.4 and 32.5.21. IOMUXC_SNVS is a separate pad-control block, not the SNVS RTC block. GPIO5's SNVS-powered pads do not mean that its CPU register interface remains usable with main power removed. Its interface gate is CCGR1[31:30], as implemented by the Linux i.MX6UL/ULL clock driver; the supplied RM marks that field reserved.

RM 32.5.4 also qualifies ALT5 by the TAMPER_PIN_DISABLE fuse setting. Use a board already provisioned for that GPIO route. **Do not program fuses or disable security/tamper protections to use this output.** Stop if the board's ownership or provisioning is unknown. Keep the Chapter 8 power and back-power checks in force before any hardware test.

18B.2 Button driver, polled with software debounce

Read the **pad status register (PSR)** for the input level; use DR for the output latch. This conditional is an explanatory fragment, not an event detector:

```
if ((GPIO_PSR(GPIO1_BASE) & KEY_BIT) == 0u) {
    /* The pad is low now. A held key also satisfies this condition. */
}
```



One physical press can produce several electrical edges. Our filter qualifies observed samples before reporting one change. The drawing has no time scale and does not prove the level stayed constant between samples.

Contact bounce adds short transitions around a real change. Two matching samples separated by 20 ms do not prove that the signal stayed steady between them. They are a cheap filter, useful when blocking the caller is acceptable:

```
int key_read_debounced(void)
{
    int first = (GPIO_PSR(GPIO1_BASE) & KEY_BIT) == 0u;
    mdelay(20);
    int second = (GPIO_PSR(GPIO1_BASE) & KEY_BIT) == 0u;
    return first == second ? second : -1;
}
```

Here -1 means **the samples disagree**, not a diagnosis of the physical cause. Returning 1 is a level report; repeatedly calling it while the key is held still returns 1. The next driver turns qualified level changes into events without delaying inside the sampling function.

A consecutive-sample filter

Create `bsp/key/bsp_key.h`:

```
#ifndef BSP_KEY_H
#define BSP_KEY_H
```

```
typedef enum {
    KEY_NONE,
    KEY_PRESS,
    KEY_RELEASE
} key_event_t;

void key_init(void);
key_event_t key_tick(void);    /* One call per real 10 ms sample. */

#endif
```

Create `bsp/key/bsp_key.c`:

```
#include <stdint.h>
#include "bsp_key.h"
#include "imx6ull.h"

static uint8_t history;
static unsigned down;

void key_init(void)
{
    REG(CCM_CCGR1) |= CCGR_GPIO1_GATE;
    REG(CCM_CCGR4) |= (3u << 2);    /* IOMUXC interface gate. */
    GPIO_GDIR(GPIO1_BASE) &= ~KEY_BIT;
    REG(KEY_PAD) = (1u << 16);     /* Hysteresis; external R12 supplies pull-up. */
    REG(KEY_MUX) = 5u;
    history = 0u;
    down = 0u;
}

key_event_t key_tick(void)
{
    unsigned raw = (GPIO_PSR(GPIO1_BASE) & KEY_BIT) == 0u;
    history = (uint8_t)((history << 1) | raw);
    if (history == 0xFFu && down == 0u) {
        down = 1u;
        return KEY_PRESS;
    }
    if (history == 0u && down != 0u) {
        down = 0u;
        return KEY_RELEASE;
    }
    return KEY_NONE;
}
```

Eight consecutive down samples produce one press; further down samples produce none until eight up samples qualify a release. With a 10 ms cadence, the decision arrives roughly 70-80 ms after a steady transition, depending on its position between samples. A key held at startup is treated as a new press after qualification. Hysteresis improves input threshold behavior but does not remove mechanical bounce.

This is a shift-register filter, not an up/down integrator. Its window is a starting choice to measure on your switch, not a universal safe minimum. Short taps may be rejected. If you move `key_tick()` into an EPIT interrupt service routine, queue its events for foreground consumption: do not call `printf` or a blocking beep from the ISR. A shared queue needs explicit producer/consumer and overflow rules; `volatile` alone does not supply them.

Hardware and software alternatives

Hardware filtering is an alternative, not automatically the best design. An RC network with a suitable Schmitt input can suppress contact chatter, but its thresholds and time constants need calculation. The supplied MINI KEY0 circuit has a pull-up and switch, not a separate RC/Schmitt filter. GPIO interrupts can also be combined with software debounce. Later, Chapter 45’s Linux gpio-keys driver presents qualified changes to the input subsystem; it does not make the board’s electrical design irrelevant.

18B.3 Buzzer, a square wave on GPIO5_IO01 (active-low via PNP)

First make the output safely idle, independently of the buzzer type. Create `bsp/beep/bsp_beep.h`:

```
#ifndef BSP_BEEP_H
#define BSP_BEEP_H
#include <stdint.h>

void beep_init(void);
void beep_set(unsigned on);    /* Active buzzer: on/off control. */
int beep_tone(uint32_t hz, uint32_t ms); /* Verified passive load only. */

#endif
```

Create `bsp/beep/bsp_beep.c`:

```
#include "bsp_beep.h"
#include "bsp_delay.h"
#include "imx6ull.h"

void beep_init(void)
{
    REG(CCM_CCGR1) |= CCGR_GPIO5_GATE;
    GPIO_DR(GPIO5_BASE) |= BEEP_BIT;    /* Preload OFF before enabling output. */
    GPIO_GDIR(GPIO5_BASE) |= BEEP_BIT;
    REG(BEEP_PAD) = 0xB0u;             /* Push-pull, DSE=6, slow slew; no pull. */
    REG(BEEP_MUX) = 5u;
}

void beep_set(unsigned on)
{
    if (on != 0u)
```

```
        GPIO_DR(GPIO5_BASE) &= ~BEEP_BIT;
    else
        GPIO_DR(GPIO5_BASE) |= BEEP_BIT;
}

int beep_tone(uint32_t hz, uint32_t ms)
{
    beep_set(0u);
    /* Input limits bound the multiplication and reject division by zero. */
    if (hz < 200u || hz > 5000u || ms == 0u || ms > 1000u)
        return -1;

    uint32_t half_us = 500000u / hz;
    uint32_t cycles = hz * ms / 1000u;
    for (uint32_t i = 0u; i < cycles; ++i) {
        beep_set(1u);
        udelay(half_us);
        beep_set(0u);
        udelay(half_us);
    }
    beep_set(0u);
    return 0;
}
```

The initial DR write avoids enabling an output with its old latch value low. It does not guarantee a silent power-up before firmware runs; that also depends on the fitted circuit and pad reset state. The SNVS pad's SPEED field is read-only in the RM. `0xB0` selects the writable output settings; do not interpret it as a promised current rating.

For a verified passive load, `beep_tone(1000, 200)` requests approximately 200 cycles at 1 kHz. Integer rounding, GPIO access time, interrupt latency, and the GPT clock affect the waveform and duration. A 1 us timer tick is resolution, not a guarantee of 1 us waveform accuracy. Measure the output with suitable equipment before treating pitch or duty cycle as calibrated. Only this foreground driver should modify GPIO5's output latch in the example: read/modify/write operations can lose a concurrent writer's update.

Why not only a GPIO toggle in a tight loop?

The delay-based loop still occupies the CPU for the whole note. A hardware timer or PWM can generate a waveform while the application does other work, but only if its output can reach the load. RM 32.5.4 lists GPIO5_IO01, not a PWM alternate function, for SNVS_TAMPER1. Chapter 48's PWM framework cannot reroute the reference BEEP wire by software. A hardware PWM design needs a separately verified PWM-capable pad and load connection. Keep that as a different hardware setup, not a promised replacement on this net.

18B.4 Putting it together

Use the refactored clock audit, UART/mini-`printf`, and GPT/delay code from your earlier project. Declare them in their BSP headers; retain Chapter 16's `int gpt_init(uint32_t ipg_hz)` contract and the free-running 1 MHz `gpt_now_us()`. Check initialization before

either delay function is used. Keep the approved ROM/board clock tree, not an assumed frequency or an unqualified clock-switching routine. Chapter 18A's Makefile discovers the two new driver folders automatically.

LED0 driver

Chapter 10 initialized LED0 directly in `main`. Move that same board-specific sequence into `bsp/gpio/bsp_gpio.c`, with its declaration in `bsp/gpio/bsp_gpio.h`. This supplies the `led_init()` used below rather than assuming an earlier driver already exists.

`bsp/gpio/bsp_gpio.h`:

```
#ifndef BSP_GPIO_H
#define BSP_GPIO_H

void led_init(void);

#endif
```

`bsp/gpio/bsp_gpio.c`:

```
#include "bsp_gpio.h"
#include "imx6ull.h"

void led_init(void)
{
    REG(CCM_CCGR1) |= CCGR_GPIO1_GATE;
    REG(LED_MUX) = 5u;
    REG(LED_PAD) = 0x17059u;
    GPIO_DR(GPIO1_BASE) |= LED_BIT;      /* Preload active-low LED OFF. */
    GPIO_GDIR(GPIO1_BASE) |= LED_BIT;
}
```

The clock, mux, pad, OFF-latch, and output-direction order matches Chapter 10. Keep its reference board and fresh-ROM prerequisites; the pad setting is not a generic value for other loads or boards. Only the GPIO1 bit-3 latch/direction changes here, leaving KEY0's bit-18 input configuration intact.

Application

This first integration uses no EPIT callback, GIC setup, or `wfi`: sampling and event handling share the foreground loop. For the later interrupt version, Chapter 16's `epit_init(ipg_hz)` takes a clock rate, not a sampling period; count ten of its 1 ms ticks between key samples. Leave IRQ/FIQ masked for this polled example. `printf` below means the earlier bare-metal implementation, not an automatically linked libc.

```
#include "bsp_clk.h"
#include "bsp_gpio.h"
#include "bsp_uart.h"
#include "bsp_gpt.h"
```

```
#include "bsp_delay.h"
#include "bsp_key.h"
#include "bsp_beep.h"
#include "imx6ull.h"

int printf(const char *fmt, ...);

int main(void)
{
    clk_enable_lab_gates();
    uart_init();
    uint32_t ipg_hz = clocks_get_ipg_hz();
    if (gpt_init(ipg_hz) != 0) {
        printf("GPT setup failed; IPG=%u Hz\r\n", (unsigned)ipg_hz);
        for (;;) { }
    }
    led_init();
    key_init();
    beep_init();

    printf("Button input ready.\r\n");
    uint32_t sampled = gpt_now_us();
    for (;;) {
        uint32_t now = gpt_now_us();
        if ((uint32_t)(now - sampled) < 10000u)
            continue;
        sampled = now;              /* Do not invent catch-up samples. */
        key_event_t event = key_tick();
        if (event == KEY_PRESS) {
            GPIO_DR(GPIO1_BASE) ^= LED_BIT;
            printf("press\r\n");
        } else if (event == KEY_RELEASE) {
            printf("release\r\n");
        }
    }
}
```

A startup-sound variant keeps sound **before** the initial `sampled` value. For an identified active buzzer, the envelope is `beep_set(1u); mdelay(100); beep_set(0u);`. For a verified passive load, `beep_tone(2000u, 100u)` requests a tone and can report failure. A tone inside the polling loop pauses button sampling. Interrupt sampling with a foreground event queue separates those responsibilities when that pause is unacceptable.

The application’s intended behavior is that a qualified press toggles LED0 and prints press; holding it does not repeat the event; a qualified release prints release. A build alone does not establish those observations.

18B.5 Key Notes

- A pad level and an application event are different interfaces. PSR reports the input now. The history filter reports one qualified press or release rather than repeating while a level is held.

- Eight matching samples at a real 10 ms cadence qualify the transition. Sparse sampling can miss bounce and short taps, while immediate catch-up calls create no new observations. Hysteresis does not remove mechanical chatter.
- Board connector numbers and circuit polarity govern the driver. MINI KEY0 uses UART1_CTS/GPIO1_IO18, while BEEP is an active-low PNP-switched load on GPIO5_IO01. A shared alias or SoC controller name does not establish the route.
- The fitted buzzer type determines whether an on/off envelope or a driven waveform is appropriate. Timer resolution, GPIO overhead and interrupt latency limit software-tone accuracy. The reference BEEP pad has no PWM route to select.
- Blocking sound delays foreground sampling. An interrupt-based alternative needs a bounded event queue, producer/consumer rules and overflow handling, with UART and sound outside the ISR. GPIO read-modify-write also needs one writer or protected updates.
- Gesture timing, such as a double tap, needs defined endpoints between qualified events and wrap-safe elapsed comparisons. Missed samples cannot be reconstructed from the number of elapsed timer ticks.
- An idle design needs a race-aware event check and a working wake source. Whole-board current includes regulators, peripherals and LEDs, not just the CPU. Do not change meter wiring on a powered board.

18B.6 Pitfalls

- **Wrong key route.** MINI V2.2's onboard KEY0 is GPIO1 bit 18. The core's `KEY0` alias on pin 49 names a different baseboard signal here; follow both connector drawings.
- **Wrong SNVS pad offsets.** BEEP uses `0x0229000C` and `0x02290050`, not offsets 0 and 0x18. A plausible-looking wrong address can change another pad.
- **Reading a latch instead of a pad.** Use PSR for this input. Check GDIR is input and UART1_CTS is muxed to GPIO1_IO18 before tuning debounce.
- **Noisy or floating level.** Check R12 and the fitted circuit with power off; a steady read alone is not proof that the pull-up is correctly fitted. Do not substitute a new pull configuration without considering other owners of the pad.
- **Calling every loop iteration a tick.** The history window counts samples, not milliseconds. Late foreground work stretches the window; repeated immediate catch-up calls falsely claim extra observations.
- **Unexpected sound.** Check buzzer type, polarity, idle-high initialization, clock gates, and GPT setup. Do not assume an oscillator-equipped buzzer will play requested notes.

- **Shared GPIO writes.** A foreground read/modify/write can overwrite an ISR’s change to another bit of the same bank. Give the bank one writer or protect the update deliberately.

18B.7 Going deeper

- **Supplied MINI v2.2 schematic**, sheet 2, and **CORE schematic**, sheets 6 and 8: transistor circuit, package pad, and connector route.
- **IMX6ULLRM Rev. 1**, Chapters 28 and 32: GPIO PSR/GDIR and the exact mux/pad fields, including the tamper-pad qualification.
- Pinned UART1_CTS pin-function definition: offset `0x008C` for mux, `0x0318` for pad, ALT5 for GPIO1_IO18. Add the main IOMUXC base `0x020E0000` to obtain the addresses used here.
- **Jack Ganssle, A Guide to Debouncing**: measured contact behavior and filter design. Use measurements to choose a window rather than adopting one value for every switch.
- **Linux drivers/input/keyboard/gpio_keys.c**: the event/debounce path we meet in Chapter 45. **drivers/pwm/pwm-imx27.c** illustrates the PWM controller, not proof of a PWM route to BEEP.

> Next chapter: **Chapter 18C: Bare-metal RTC.** Separate a retained counter from the main-powered interface that lets the CPU read it.

Chapter 18C: Bare-metal RTC

After a restart, the ordinary uptime counter begins again at zero. A log timestamp should not. Which counter can keep advancing while the CPU and DDR have no power, and how does the CPU read it after returning?

The **Secure Non-Volatile Storage (SNVS)** block has separate high-power (HP) and low-power (LP) sections. We use its LP real-time counter and one retained general-purpose register. “Always-on” is a supply requirement, not a promise: if the LP supply disappears, so does its retained state.

This is a bare-metal counter/retention design case. Setting a counter to zero does not set a wall clock, and deliberately causing a brown-out is not needed. Keep Chapter 18A’s startup, linker, toolchain, and fresh-ROM handoff policy; initialize Chapter 16’s GPT delay source before using the driver below.

18C.1 What the SNVS provides

The supplied **IMX6ULLRM Rev. 1, Chapter 48** describes:

- An HP real-time counter and its alarm, which lose state with system power.
- An LP section with the retained counter, control/security state, monotonic counter, and general-purpose register.
- A **32-bit LPGPR at offset 0x68** described in Section 48.7.14: retained space for one marker or counter, not a general byte-addressable buffer.
- A monotonic counter incremented by permitted **writes**, not automatically by every chip reset. It has security and fuse-era behavior; we do not use it as a reboot counter or program its fuses.

The public RM describes the LP counter but omits its register fields. The starred entries in Section 18C.3 use the Linux v6.12 SNVS RTC implementation as an additional reference. NXP’s migration guide, Table 1 and clarification of the omission explain the UL-to-ULL relationship. Obtain matching NXP security/reference documentation for production use. This example does not provision security.

18C.2 Powering SNVS

Follow the actual nets, rather than a generic VBAT label:

Reference sheet	What it shows
MINI v2.2 sheet 4	CR1, labelled 1220, feeding VDD_COIN_3V
CORE sheet 8, internal project CL6Y2CB_V1.9	VDD_COIN_3V on J1 pin 41
CORE sheet 2	D1 from VDD_SNVS_3V3 and D2 from VDD_COIN_3V feeding VDD_SNVS_IN; R34 1.5 kOhm is also drawn across the coin-cell diode path
CORE sheet 6	Y1 32.768 kHz at RTC_XTALI/RTC_XTALO, separate from the 24 MHz main reference

These connections are not a battery compatibility or lifetime guarantee. Check the fitted circuit, cell chemistry/polarity, and supplier-approved arrangement, including any current through R34. Do not assume every “1220” holder accepts a rechargeable cell or that a supercapacitor can be substituted.

The supplied industrial electrical datasheet Sections 4.1.6.3 and 4.2 require the SNVS supply first on and last off, and forbid externally driving an unpowered I/O domain. Follow the matching datasheet and Chapter 8’s power/back-power inventory. CPU register access still needs the main-powered HP interface and its clocks; a coin cell does not keep the Cortex-A7 or that bus interface running.

An ordinary main-power cycle can retain LP state when its supply remains valid. It is different from LP power-on reset or LP software reset. With no working backup path, warm reset may preserve the count while complete power removal does not. Measure and document the supply conditions before interpreting a retention result.

18C.3 Register map (relevant subset)

SNVS base is `0x020CC000`. Offsets here are relative to that base, **not** to the LP subrange used by some Linux device-tree nodes.

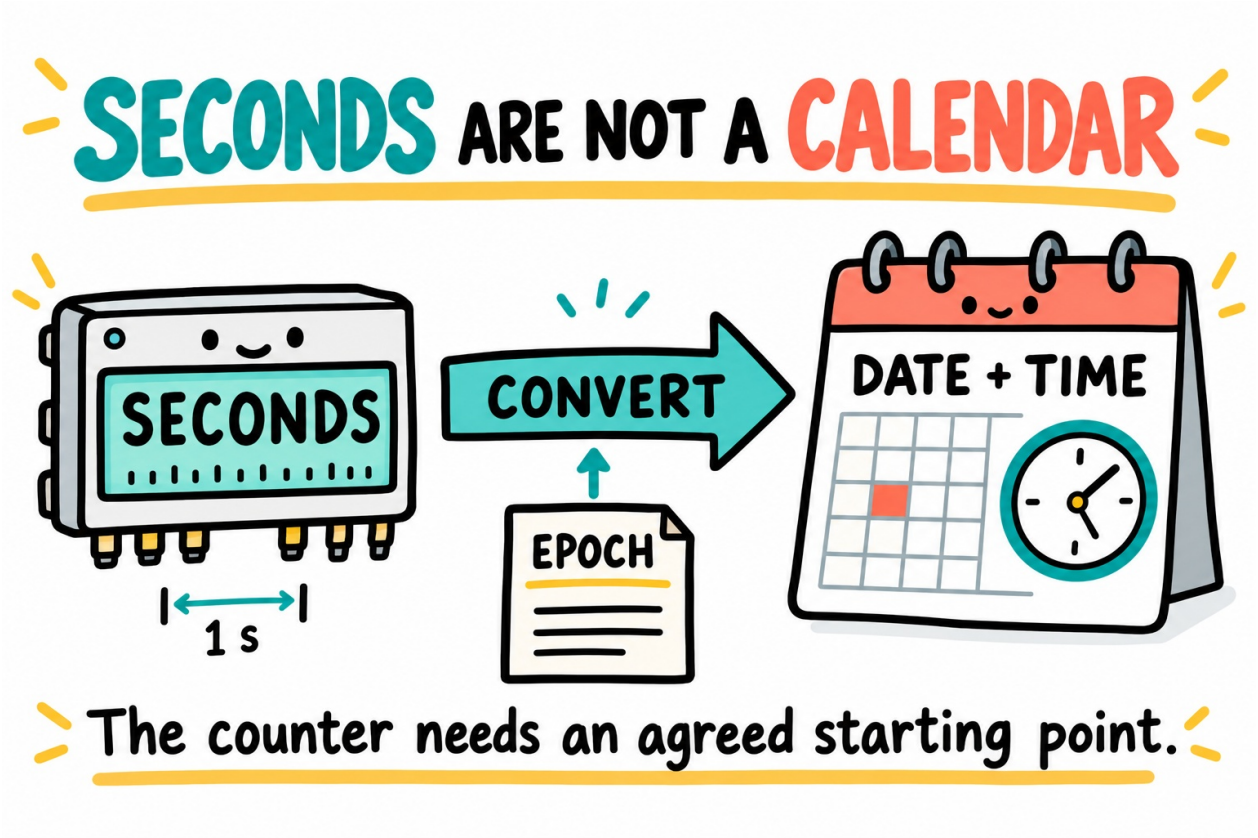
Register	Offset	Purpose
SNVS_HPLR	+0x00	HP lock register; do not set locks for this example
SNVS_HPCOMR	+0x04	HP command/access policy; not an innocuous initialization write

Register	Offset	Purpose
SNVS_LPLR	+0x34	LP lock register
SNVS_LPCR	+0x38	LP control; SRTC_ENV is bit 0*
SNVS_LPSR	+0x4C	LP status, including write-one-to-clear fields
SNVS_LPSRTCMR*	+0x50	Counter bits 46:32 in register bits 14:0
SNVS_LPSRTCLR*	+0x54	Counter bits 31:0
SNVS_LPTAR*	+0x58	LP alarm’s 32-bit seconds value; not armed in this example
SNVS_LPGPR	+0x68	One 32-bit retained general-purpose register

The LP counter is **47 bits of 32.768 kHz ticks**: 32 whole-second bits and 15 fractional bits. Form `((uint64_t)(hi & 0x7FFF) << 32) | lo`, then shift right by 15. Equivalently, seconds are `((hi & 0x7FFF) << 17) | (lo >> 15)`. At 32768 ticks, seconds must be 1; at 65536 ticks, 2. These are useful arithmetic checks without a board.

The counter stores a number, not a calendar or a timezone. Software chooses the epoch by setting that number. An unsigned 32-bit Unix-seconds choice reaches its limit in 2106; a signed 32-bit software representation has a different 2038 limit. Neither is a 64-bit-seconds hardware RTC.

18C.4 Driver



The raw SNVS counter counts ticks. Here, “seconds” means the value after removing its fractional bits. Turning that number into a calendar needs an agreed epoch and software conversion. A running counter alone does not establish a valid date.

Add the following definitions inside `imx6ull.h`, keeping the hardware map out of `bsp_rtc.c`. The starred offsets/bit use the implementation evidence identified above:

```
#define SNVS_BASE           0x020CC000u
#define SNVS_LPCR           (SNVS_BASE + 0x38u)
#define SNVS_LPSRTCMR       (SNVS_BASE + 0x50u)
#define SNVS_LPSRTCLR       (SNVS_BASE + 0x54u)
#define SNVS_LPGPR          (SNVS_BASE + 0x68u)
#define LPCR_SRTC_ENV       (1u << 0)
#define SRTC_MSB_MASK       0x7FFFu
#define CCGR_SNVS_HP_GATE   (3u << 18) /* CCGR5[19:18] */
#define CCGR_SNVS_LP_GATE   (3u << 20) /* CCGR5[21:20] */
```

Create `bsp/rtc/bsp_rtc.h`:

```
#ifndef BSP_RTC_H
#define BSP_RTC_H
#include <stdint.h>

int rtc_init(void);
int rtc_get_seconds(uint32_t *seconds);
int rtc_set_seconds(uint32_t seconds);
uint32_t rtc_scratch_read(void);
```

```
int rtc_scratch_write(uint32_t value);

#endif
```

Create `bsp/rtc/bsp_rtc.c`:

```
#include "bsp_rtc.h"
#include "bsp_delay.h"
#include "imx6ull.h"

static int rtc_enable(unsigned enable)
{
    uint32_t control = REG(SNVS_LPCR);
    if (enable != 0u)
        control |= LPCR_SRTC_ENV;
    else
        control &= ~LPCR_SRTC_ENV;
    REG(SNVS_LPCR) = control;

    for (unsigned attempt = 0u; attempt < 1000u; ++attempt) {
        unsigned running = (REG(SNVS_LPCR) & LPCR_SRTC_ENV) != 0u;
        if (running == (enable != 0u))
            return 0;
        udelay(10u);
    }
    return -1;
}

int rtc_init(void)
{
    REG(CCM_CCGR5) |= CCGR_SNVS_HP_GATE | CCGR_SNVS_LP_GATE;
    return rtc_enable(1u);           /* Preserve the existing counter value. */
}

static uint64_t rtc_raw(void)
{
    uint32_t hi = REG(SNVS_LPSRTCMR) & SRTC_MSB_MASK;
    uint32_t lo = REG(SNVS_LPSRTCLR);
    return ((uint64_t)hi << 32) | lo;
}

int rtc_get_seconds(uint32_t *seconds)
{
    if (seconds == 0)
        return -1;

    /* Compare complete samples, not only the upper word. */
    uint64_t previous = rtc_raw();
    for (unsigned attempt = 0u; attempt < 100u; ++attempt) {
        uint64_t current = rtc_raw();
        if (current == previous) {
            *seconds = (uint32_t)(current >> 15);
            return 0;
        }
        previous = current;
    }
    return -1;                       /* No output update on failure. */
}
```

```
int rtc_set_seconds(uint32_t seconds)
{
    if (rtc_enable(0u) != 0)
        return -1;                  /* Never write a running/locked counter. */

    REG(SNVS_LPSRTCLR) = seconds << 15;
    REG(SNVS_LPSRTCMR) = seconds >> 17;
    return rtc_enable(1u);
}

uint32_t rtc_scratch_read(void)
{
    return REG(SNVS_LPGPR);
}

int rtc_scratch_write(uint32_t value)
{
    REG(SNVS_LPGPR) = value;
    for (unsigned attempt = 0u; attempt < 1000u; ++attempt) {
        if (REG(SNVS_LPGPR) == value)
            return 0;
        udelay(10u);
    }
    return -1;
}
```

The setter clears SRTC_ENV, waits for the observed state, writes the low word and high word with fractional bits zero, then re-enables and waits again. It preserves the other LPCR fields. At most 1000 failed polls request 10 us delays: roughly 10 ms plus GPT guard ticks and software overhead, not a hard wall-clock deadline. This assumes GPT delays already work. A failed restart can leave the counter stopped. Report the error and inspect access/lock/clock state rather than proceeding as though time was set.

RM 48.6.2 describes both asynchronous sampling and a split-register update hazard. A high-low-high read addresses carry between words but not every partially synchronized value. This small driver requires two identical **whole-counter samples**, with a bounded retry. Linux instead accepts a small forward difference between full samples. Equality is practical for fast polled reads between 32 kHz ticks, but can time out on a slow or interrupted access path. Neither a retry count nor compilation certifies the hardware’s clock/synchronization behavior.

A stopped counter can also return a stable value. The getter checks read consistency; enable-state readback and successive time readings tell us whether it is running and advancing.

This driver deliberately does **not** set HPCOMR.NPSWA_EN. RM 48.7.2 says that bit broadens non-privileged access; privileged execution does not make the write a harmless no-op. Access rights, security state, and locks must already permit this development-board use. The upstream driver also initializes the power-glitch detector and clears status during probe. We do not copy its blanket status clear into a bare-metal security policy: pending tamper/security indications must not be erased just to obtain a

successful counter read. A cold-board failure may require vendor-documented SNVS initialization for the exact device/provisioning. Stop there, without guessed security-bit writes, LP reset, or fuse changes.

LPGPR can be locked or cleared by documented reset/tamper behavior. The write helper checks readback and reports failure. That is evidence of a matching read, not proof that a previously equal value was newly written or that it will survive power loss. Use only a board on which this register is not owned by boot/security software. Do not interpret arbitrary retained data as tamper-proof storage.

18C.5 Counter application

This program first reads the existing counter. It does **not** reset time on every boot or infer time validity from a marker. `SET_RTC_ON_THIS_BOOT=0` preserves the existing count. The alternative value 1 selects a deliberate initialization operation, not a normal restart policy. Any retention comparison requires the read-only-time version and no intervening bootloader or Linux RTC driver that resets the counter.

The application uses the Chapter 12 mini-printf formats `%u` and `%02u`; its formatter does not support `%lu` or a 64-bit length modifier. Cast variadic arguments to the unsigned type it expects.

```
#include <stdint.h>
#include "bsp_clk.h"
#include "bsp_uart.h"
#include "bsp_gpt.h"
#include "bsp_delay.h"
#include "bsp_rtc.h"

#define SET_RTC_ON_THIS_BOOT 0

int printf(const char *fmt, ...);

static void stop(const char *message)
{
    printf("%s\r\n", message);
    for (;;) { }
}

static void print_seconds(uint32_t seconds)
{
    uint32_t days = seconds / 86400u;
    uint32_t rest = seconds % 86400u;
    printf("%u days, %02u:%02u:%02u",
           (unsigned)days, (unsigned)(rest / 3600u),
           (unsigned)((rest / 60u) % 60u), (unsigned)(rest % 60u));
}

int main(void)
{
    clk_enable_lab_gates();
    uart_init();
```

```
    uint32_t ipg_hz = clocks_get_ipg_hz();
    if (gpt_init(ipg_hz) != 0)
        stop("GPT setup failed; inspect the clock audit.");
    if (rtc_init() != 0)
        stop("RTC enable timed out; inspect clocks, access, and locks.");

    if (SET_RTC_ON_THIS_BOOT != 0) {
        if (rtc_set_seconds(0u) != 0)
            stop("RTC set failed.");
        if (rtc_scratch_write(0x52544331u) != 0)
            stop("LPGPR readback failed; do not claim retention.");
    }
    printf("LPGPR = 0x%08x\r\n", (unsigned)rtc_scratch_read());

    for (;;) {
        uint32_t seconds;
        if (rtc_get_seconds(&seconds) != 0)
            stop("RTC read did not stabilize.");
        printf("[RTC = %u s] ", (unsigned)seconds);
        print_seconds(seconds);
        printf("\r\n");
        mdelay(1000u);
    }
}
```

With the one-time zero setting, the display means elapsed time since that setting, including time the CPU was off. It is not this boot's uptime. If you instead set Unix seconds, the same day/hour decomposition remains a numeric interval; it is not yet a date conversion. UART output and the foreground delay also mean lines are not an exact one-second measurement cadence. Compare counter differences with an independent elapsed-time source.

18C.6 Retention across main-power loss

The case is **controlled main-power removal**, not an induced brown-out. The backup supply must remain valid throughout. Undervoltage and rail manipulation are not part of this retention model.

Before removing anything, confirm the supplier-approved backup arrangement, the board revision, and all main-power/back-power paths. Follow the Chapter 8 checks and the datasheet's supply order. The supplied schematics alone do not establish a universal unplugging recipe.

A one-time initialization establishes zero and a marker. Subsequent images use `SET_RTC_ON_THIS_BOOT=0`, so startup observes rather than replaces that state. With the verified SNVS supply maintained, controlled main-power loss removes CPU access without necessarily stopping the LP counter. DC input, USB-TTL, USB-OTG, debug adapters and powered add-ons all matter to this power boundary. Disconnecting one VBUS cable does not establish that main rails are off.

After main power returns, the same read-only-time image reaches the counter through the fresh-ROM route. The counter difference corresponds to **all elapsed time between readings**, including shutdown, the off interval, startup and download. An independent time reference and a separate LPGPR reading distinguish elapsed-time continuity from marker retention. This is an explanatory case, not a recorded physical test.

A retained marker and an advancing counter test different things. A marker can remain while the counter stops; a running counter does not prove that every retained register is valid. Startup and download time cannot be omitted from the comparison. If the readings disagree, investigate the supply, clock, resets, and intervening firmware.

18C.7 Key Notes

- The LP counter’s 47 bits represent 32.768 kHz ticks, including 15 fractional bits. Whole seconds still need a software-chosen epoch and calendar conversion. Zero is not a valid wall-clock setting by itself.
- LP retention and CPU access have different power requirements. A valid backup supply can retain state while the main-powered HP interface is unavailable. Main-power cycling, warm reset and LP POR are not interchangeable events.
- Split-register reads cross a synchronization boundary. Two identical whole-counter samples check consistency, while successive values and enable state address whether the counter is advancing. A stable value alone can belong to a stopped counter.
- Setting time is a state transition: observed disable, low/high writes and observed re-enable. GPT-backed polling requires its delay source first, and failure can leave the counter stopped rather than establish a new time.
- LPGPR is one retained 32-bit word with an owner. A marker and a software boot count cannot independently occupy it. A count would represent executions reaching the update, not every reset or tamper-proof history.
- Retention comparisons include the entire read-to-read interval and require firmware that does not reset time. Marker survival does not establish oscillator accuracy or counter continuity. Access-policy writes and security-status clears are not harmless ways to obtain a successful read.
- A Unix-seconds interface needs bounded parsing and an explicit UTC convention before calendar/timezone presentation. Backup loss also needs an invalid-state policy. Physical backup removal requires the approved supply order with all other sources off. Never pull a cell from a powered board or short a supply to force reset.

18C.8 Pitfalls

- **Confusing 47-bit ticks with seconds.** The high register contributes 15 bits, the low 32; the lowest 15 bits are fractional. One tick is not one second.
- **Using an LP-relative offset with the full base.** Linux may add 0x34 before its LP-local offsets. Here +0x50 and +0x54 are already relative to 0x020CC000.
- **Reading only high-low-high.** Carry protection alone does not address asynchronous sampling. Compare full samples and report exhausted retries.
- **Writing before observed disable.** Do not merely clear SRTC_ENV and immediately write the counter. The transition crosses clock domains, and locks/access state can prevent it.
- **Treating access-policy writes as harmless.** NPSWA_EN changes who can access privileged registers. Do not broaden permissions, clear security status, or program locks/fuses for this example.
- **Assuming six scratch words.** This subset uses only the documented 32-bit LPGPR. Other SoC variants and separate GPR blocks are not evidence for consecutive scratch RAM here.
- **A marker that proves too much.** It is application data, not proof of battery health, RTC accuracy, or resistance to resets/tampering.
- **A frozen counter with a readable interface.** Enable readback alone is not proof that the 32 kHz timebase advances. Check successive seconds as well as control state.

18C.9 Going deeper

- **IMX6ULLRM Rev. 1, Chapter 48**: HP/LP domains, synchronization guidance, locks, access policy, and LPGPR. Note its LP RTC field omissions.
- **Supplied industrial/commercial datasheets**, Sections 4.1.6 and 4.2, plus **MINI sheet 4 / CORE sheets 2, 6, 8**: supply states, ordering, crystal, and the real backup route.
- **AN5350, i.MX 6ULL Migration Guide**, and the linked NXP clarification: the UL-to-ULL SNVS relationship and the missing public register fields.
- **Linux drivers/rtc/rtc-snvs.c**: the LP offsets, 47-bit conversion, full-sample retry, enable/disable acknowledgement, and separate alarm setup. Chapter 48 introduces the Linux RTC interface.
- **POSIX time(), gmtime(), localtime()**: software epoch/calendar/timezone services. They do not supply a bare-metal RTC driver merely by being declared.

Chapters 18A-18C supplement Part II. The button and RTC implementations use the Chapter 18A layout and earlier startup, UART, and delay code; they are not independent copy-and-boot projects. From here, Chapter 19 moves to U-Boot, which prepares hardware and loads the next image before Linux runs.

> Next chapter: **Chapter 19: U-Boot from source, first boot.**

Chapter 19: U-Boot from source, first boot

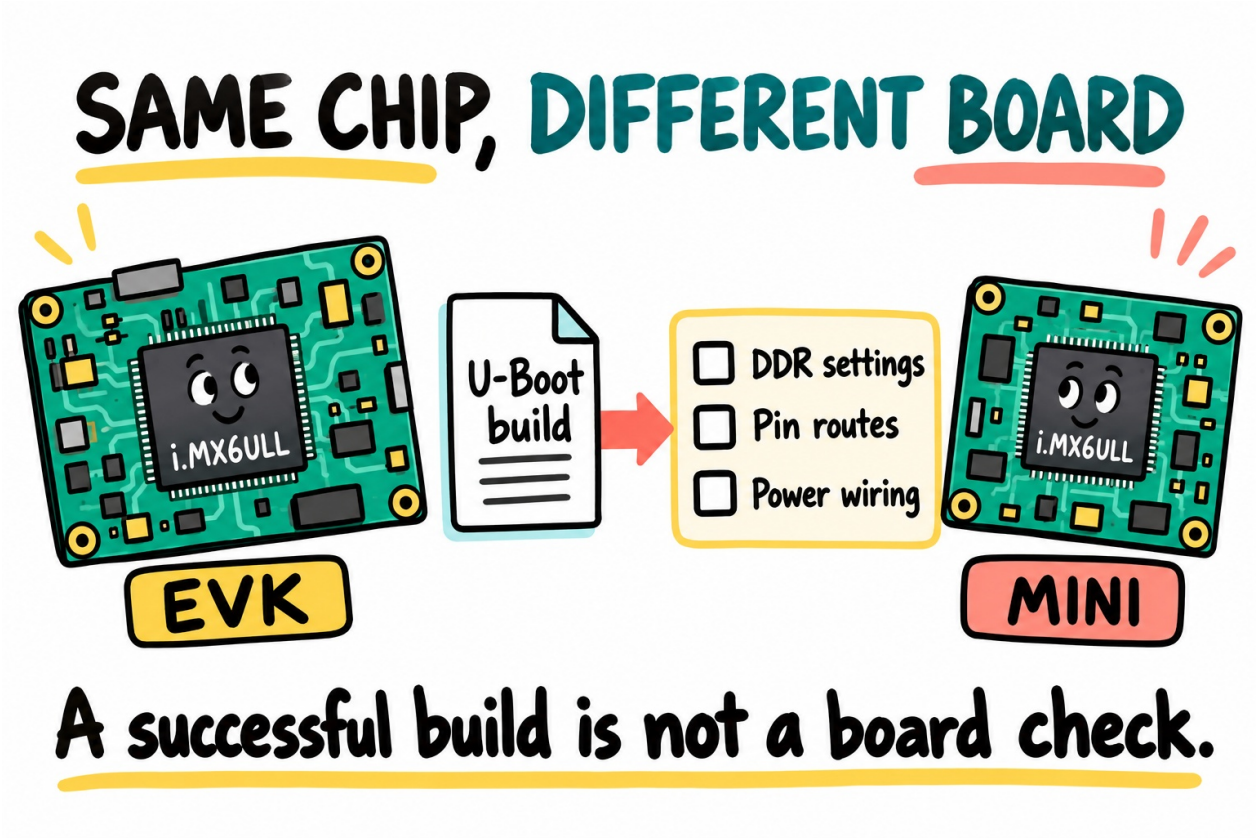
Our LED program had a modest job. It reached one pin, changed its level, and stayed there. Now imagine extending it until it can find a file on an SD card, read a device tree, choose a boot policy, and start Linux. The register writes would be only a small part of the work.

U-Boot already provides that larger framework. This time, we want its prompt on **our Point Atom i.MX6ULL MINI**, not a successful build for somebody else's evaluation board. We will begin with the board vendor's source, keep its DDR and storage setup visible, and build a deliberately quiet initial image. It will wait for us instead of trying to start an old Linux installation.

The hardware baseline is the supplied **MINI V2.2 baseboard with the 512 MiB DDR3L / 8 GB eMMC core**. The core schematic names `NT5CC256M16EP-EK` memory. The vendor guide explains that MINI shares ALPHA's software for the peripherals present on MINI. That is why an `alientek` configuration can be the right starting point even when its filename does not say `mini`.

Check the markings on your own core and baseboard before following the board steps. A NAND core or another memory arrangement needs its own configuration. Keep the factory image and use a spare SD card first. We have checked the source and host build described here, **not performed a physical MINI boot test**.

19.1 Start with the MINI’s board support



The chip name tells us which SoC logic is shared. The board source supplies the memory and wiring details that the chip name cannot. These drawings are not real board layouts.

There are three useful levels of source:

Source	What it contributes
Upstream U-Boot	The shared project and current frameworks
NXP’s BSP	Release-specific support for NXP platforms
Point Atom’s board source	The core memory setup and board connections used by the tutorial hardware

For our first MINI image, use Point Atom’s public tutorial repository at commit **`edb7ca5ac4be2d978be60a2a12c61e0b6d1f7feb`**. It is based on U-Boot 2016.03. We use this old tree as a board bring-up baseline, not as a recommendation for a new production product or an Internet-facing recovery service. It still needs maintenance, security review, and hardware testing.

One naming difference matters. Section 30.2 of the supplied V1.81 guide uses `mx6ull_14x14_ddr512_emmc_defconfig` from its factory archive. **That name is not present in this public tutorial snapshot.** Here the matching tutorial target is `mx6ull_alientek_emmc_defconfig`. Do not combine the archive's command with a different source tree, and do not substitute an EVK target just because it compiles.

Chapters 20-24J also examine upstream **v2026.04** in a separate study tree. That newer tree lets us learn current APIs and work through a modern MINI migration. Its EVK and PICO builds are comparisons on the host, not replacement MINI firmware. We will introduce that tree when we need it. For now, one board target and one output directory are enough.

19.2 Clone and look around

Use the lab Ubuntu environment from Chapter 3. `$` means the host terminal. Later, `=>` means the U-Boot prompt on the MINI. Source the environment script in every new host terminal:

```
$ . ~/imx6ull/scripts/env.sh
$ cd ~/imx6ull/src
$ git clone https://github.com/alientek-openedv/uboot-imx-
rel_imx_4.1.15_2.1.0_ga_alientek.git uboot-mini-vendor
$ cd uboot-mini-vendor
$ git checkout -b mini-first-lab edb7ca5ac4be2d978be60a2a12c61e0b6d1f7feb
$ git rev-parse HEAD
edb7ca5ac4be2d978be60a2a12c61e0b6d1f7feb
```

The local branch gives our coming edits a home. If the directory already exists, inspect its revision and preserve local changes. Do not clean somebody else's checkout to make a command match this page.

Open these files in your editor before building:

Path in the vendor snapshot	Question it answers
`configs/mx6ull_alientek_emmc_defconfig`	Which board target and image configuration are selected?
`board/freescale/mx6ull_alientek_emmc/Kconfig`	Which directory and configuration header belong to that target?
`board/freescale/mx6ull_alientek_emmc/mx6ull_alientek_emmc.c`	How are UART1, SD, eMMC, and the board's other devices connected?

Path in the vendor snapshot	Question it answers
`board/freescale/mx6ull_alientek_emmc/imximage.cfg`	Which DDR writes does the ROM execute?
`include/configs/mx6ull_alientek_emmc.h`	What memory size, environment backend, and boot defaults are compiled in?
`arch/arm/cpu/armv7/start.S`, `arch/arm/lib/crt0.S`	How does execution enter the shared ARM runtime?

The old directory name `freescale` is correct for this snapshot. The newer upstream tree uses different paths and configuration machinery. Follow the file in the tree you are actually editing.

Notice that this target does not enable SPL. The ROM executes a **DCD**, the Device Configuration Data table in the image, to prepare DDR. It then loads full U-Boot into that memory. A missing SPL output will not be a build error.

19.3 A quiet initial image

The factory defaults are intended to boot a complete system. Our initial example needs less: a console prompt, no automatic boot, and no persistent environment changes. We can express that with a small, visible edit.

Open `include/configs/mx6ull_alientek_emmc.h`. Immediately before its **last** `#endif`, after the Android-support conditional has closed, insert:

```
/* First MINI lab: stay at the prompt and use a RAM-only environment. */
#undef CONFIG_BOOTDELAY
#define CONFIG_BOOTDELAY -1
#undef CONFIG_ENV_IS_IN_MMC
#define CONFIG_ENV_IS_NOWHERE
```

`-1` disables automatic boot in this release. `CONFIG_ENV_IS_NOWHERE` selects the nonpersistent environment backend. Compiled defaults still exist, and `setenv` can change their RAM copy, but this build will not load an old MMC environment or save a new one. We are changing our project, not a host system file and not the MINI's eMMC contents.

In that same header, find the `#define CONFIG_VIDEO` line immediately before `#ifdef CONFIG_VIDEO`. Replace only that definition with a comment:

```
/* Display disabled for the first UART/SD lab. */
#ifdef CONFIG_VIDEO
```

Leave the enclosed definitions and closing conditionals in place. For this initial console example, no display support is compiled. This matters even without a screen attached: the old display helper drives GPIO5_IO09 as LCD reset, but the supplied core puts that pad in the shared reset network. Chapter 24I examines the mismatch before any display migration. An inherited helper is not permission to pulse the board's reset line.

Still in the header, find `#define CONFIG_CMD_USB` immediately before `#ifdef CONFIG_CMD_USB`. Replace that definition with:

```
/* USB host disabled for the first UART/SD lab. */
#ifdef CONFIG_CMD_USB
```

Again leave the conditional block in place. This old header enables a USB Ethernet adapter driver alongside USB host commands. We disable the whole group before removing networking, rather than leaving unresolved network dependencies. The old header definition would override an unset defconfig entry, so changing only the defconfig would not disable this USB group.

Open `configs/mx6ull_alientek_emmc_defconfig` and add this line at the end:

```
# CONFIG_CMD_NET is not set
```

For this UART/storage example we leave networking and USB host commands out. The header enables the FEC/PHY code under `CONFIG_CMD_NET`; disabling it avoids running the old PHY-specific hook before we have reconciled V2.2's different PHY. Later network examples need that support deliberately migrated and tested. ROM USB recovery is independent of these U-Boot commands and remains available.

There is also a real C lifetime bug in this old source that GCC 13 diagnoses. Open `fs/fat/fat.c` and find `do_fat_read_at()`. Move the declaration `dir_entry dent;` from inside its `while (isdir)` loop to the function's local declarations, directly after `dir_entry *dentptr = NULL;`. Do not leave a second copy in the loop. The relevant result is:

```
fsdata *mydata = &datablock;
dir_entry *dentptr = NULL;
dir_entry dent;
```

The loop stores `&dent` in `dentptr`, then code after the loop uses that pointer. A loop-local `dent` has already reached the end of its lifetime by then. Moving the object to function scope keeps it alive through the final file read. We fix that bug instead of silencing the compiler warning. A downloadable patch containing these same edits is available for comparison with your editor changes.

Now build with Chapter 3's existing Linux-target Arm compiler:

```
$ make O="$HOME/imx6ull/build/uboot-mini-vendor" ARCH=arm \
    CROSS_COMPILE="$CROSS_COMPILE" mx6ull_alientek_emmc_defconfig
```

```
$ make O="$HOME/imx6ull/build/uboot-mini-vendor" ARCH=arm \
    CROSS_COMPILE="$CROSS_COMPILE" -j4
```

Why repeat `CROSS_COMPILE` here if the environment script already exported it? This vendor Makefile assigns an old compiler prefix itself. A command-line assignment takes precedence over that Makefile assignment. We deliberately pass the prefix selected by our environment instead of installing another compiler or renaming a toolchain directory.

U-Boot uses freestanding target build rules, not Linux user-space startup or glibc. `HOSTCC` still uses Ubuntu's native compiler for programs that run on the host. `O=` keeps generated files in a separate output directory. Repeat that same argument on every build command. `-j4` allows four parallel jobs. Use `-j1` if the lab machine is short of memory.

This snapshot, including the visible edits above, builds with the book's Arm GNU 13.2.Rel1 compiler. It produces warnings from old code and assembly, so a successful exit is not a claim of warning-free or production-qualified source. Do not ignore a new error or the dangling-pointer warning we just fixed.

File under `~/imx6ull/build/uboot-mini-vendor`	**Use**
`u-boot`	Target ELF with symbols for inspection and debugging
`u-boot.map`	Linked sections and symbol addresses
`.config` and `include/autoconf.mk`	Expanded configuration, including old header-defined settings
`u-boot.bin`	Main binary without the ROM image wrapper
`u-boot.imx`	The ROM-facing image with IVT and vendor DCD
`tools/mkimage`	Host image builder used during packaging

Inspect the beginning of **`u-boot.imx`**:

```
$ xxd -l 32 ~/imx6ull/build/uboot-mini-vendor/u-boot.imx
```

The IVT starts at file byte zero. Its first four bytes are `d1 00 20 40`, and the selected entry address is `0x87800000`. Decode the IVT and BootData fields as in Chapter 7. The old `mkimage -l` auto-detection can misidentify this image as a GP header, so do not use that output as proof of an i.MX load address.

This file has no leading 1 KiB pad. Chapter 11's builder did include one. The media placement must account for that difference exactly once.

19.4 Flash to SD

Our first boot uses a **dedicated spare SD card**, not an eMMC update. Confirm that the core is the 512 MiB eMMC variant, check Chapter 8's MINI power and connector procedure, and retain the known-good factory boot path. Do not use these commands with the NAND core or an arbitrary memory replacement.

Location	Meaning
Image file byte `0`	Start of the IVT in `u-boot.imx`
SD-card byte `0x400`, or 1 KiB	Required location of that IVT on this boot medium
Raw range at `0xC0000`, size `0x2000`	Vendor MMC environment layout, if persistent storage is later re-enabled

Our RAM-only example build does not use that environment region. Still leave the space reserved: check that the image length plus its 1 KiB placement ends before `0xC0000`, and keep filesystem partitions outside these raw ranges. A dedicated DOS/MBR example layout differs from GPT, whose primary entries overlap this boot placement. Do not combine this write with an arbitrary existing partition table.

Repeat Chapter 11's card-identification procedure before writing. Confirm the whole-device path by its size and physical removal/reinsertion, inspect its mounted partitions, and unmount only the partitions belonging to that card. `/dev/sdX` below is a placeholder for the confirmed device. Writing the wrong destination can destroy another disk's data.

```
$ sudo dd if="$HOME/imx6ull/build/uboot-mini-vendor/u-boot.imx" \
    of=/dev/sdX bs=1K seek=1 conv=notrunc,fsync status=progress
```

`bs=1K seek=1` places file byte zero at card byte 1024. There is one image write, not an SPL write followed by another image at 69 KiB. Do not feed `u-boot.bin` to the old `imxdownload` helper here: it would introduce another packaging path instead of writing the image we have just inspected.

Read back and compare exactly the written bytes:

```
$ image="$HOME/imx6ull/build/uboot-mini-vendor/u-boot.imx"
$ bytes=$(stat -c %s "$image")
$ sudo dd if=/dev/sdX bs=1K skip=1 count="$bytes" iflag=count_bytes \
    status=none | cmp -n "$bytes" "$image" -
```

`skip=1` skips 1 KiB. `iflag=count_bytes` makes `count` a byte count. `cmp` prints nothing and returns success when the bytes match. A mismatch is a stop point. Matching media bytes establish a correct copy, not tested DDR margins.

19.5 First boot

With board power off, insert the spare card and set the **MINI's documented SD boot switches** as established in Chapter 8. Do not use an EVK switch diagram. Remove an optional SDIO Wi-Fi module for this example: MINI shares its USDHC1 signals between that connector and the TF-card socket.

Connect the MINI's USB-TTL console, not its USB-OTG port. Open the confirmed serial device at 115200 baud, 8 data bits, no parity, and one stop bit before powering the board. The CH340C can enumerate while the core is off, so a host serial device alone does not prove that U-Boot is running.

Our example edit disables the autoboot countdown. If startup reaches the command interpreter, it should wait at `=>`. The source's board banner says `MX6ULL ALIENTEK EMMC`. The factory guide shows a different banner from a different snapshot. Do not require its timestamp, display name, or complete log to match this build.

Record the actual `version` output and boot log. A prompt is a useful milestone: the ROM, vendor DCD, loaded program, runtime, and UART have worked far enough to answer us. It is not a test of every DDR location, temperature corner, or peripheral.

There should be no required `U-Boot SPL` banner for this target. If the board still counts down or starts the factory Linux image, first check the running version, switch setting, image placement, and whether our header edit was compiled. Do not assume you are running the new SD image.

For a silent board, keep the checks in order: power and boot selection, ROM-readable media placement, core-matched DCD, then the console route. A UART change cannot repair DDR setup that prevented entry into U-Boot.

19.6 First commands

Begin with inspection:

```
=> version
=> help
=> printenv bootcmd bootargs loadaddr fdt_addr fdt_file
```

```
=> bdinfo
=> mmc list
```

`version` identifies the running program, not the directory on the host. `printenv` shows the current RAM environment. Some variables may be absent. Our initial environment backend does not load saved MMC settings. The old header can still contain environment strings referring to commands we disabled. A printed `netboot` or display variable does not enable that feature. Do not run the inherited `bootcmd` in this inspection example.

In `bdinfo`, record the DRAM start and size, relocation address, and relocation offset. For the intended core, the declared memory size is 512 MiB starting at `0x80000000`. The startup memory probe is not a full DDR stress test.

The vendor board code maps USDHC1 to U-Boot MMC device 0 (the TF card) and USDHC2 to device 1 (the core's eMMC). Confirm that with your actual output:

```
=> mmc dev 0
=> mmc info
=> mmc dev 1
=> mmc info
```

Selecting and identifying a device does not write its contents. These indices are not host `/dev/sdX` names and do not guarantee Linux `/dev/mmcblkN` ordering. Do not run `mmc write`, `mmc erase`, or `saveenv` in this first example.

MINI has **one RJ45 connector on ENET2**. Older baseboards use LAN8720, while the supplied V2.2 schematic uses SR8201F at PHY address 1. This public tutorial snapshot selects the SMSC PHY driver, so its build alone does not establish correct SR8201F initialization. Our initial configuration disables that network path. Leave Ethernet for the revision-aware checks in Chapters 22 and 24. A `FEC1` log label is a software index, not a second RJ45 socket that the MINI must have.

`md` reads memory, but only use an intentionally chosen valid RAM range or a register whose read behavior you understand. `mw` and `mtest` write memory. They can overwrite U-Boot's code, stack, heap, device tree, or loaded images. Do not sweep a large live RAM range with a command copied from another log. Chapter 14's memory ownership rules still apply.

19.7 Recognizing Chapter 14 in the ROM's DCD

Open **our vendor target's** `board/freescale/mx6ull_alientek_emmc/imximage.cfg`. Two of its MMDC entries are:

```
DATA 4 0x021B000C 0x676B52F3
DATA 4 0x021B0010 0xB66D0B63
```

The `4` means a 32-bit write. The addresses select `MDCFG0` and `MDCFG1`. Their meaning comes from the reference manual and Chapter 14's DDR timing analysis. Do not stop at "these numbers also appear in an EVK file." The provenance here is the vendor's board configuration for the documented core, not a guess made from the SoC name.

The host image builder encodes those textual entries into the DCD. The ROM performs the writes before loading and entering the DDR-resident program. Full U-Boot then establishes its runtime and reports memory. Chapter 24A returns to this same vendor sequence when constructing a teaching port.

The vendor table also enables broad clock-gate masks. That explains its baseline behavior, but it does not make all-ones clock writes a good default for Part II's smaller experiments. Nor does a vendor table replace testing after changing a DDR part, routing, frequency, or power design.

19.8 Key Notes

- The pinned vendor target describes the documented 512 MiB eMMC core. Shared ALPHA/MINI software does not make every carrier connection or PHY revision interchangeable.
- ROM executes the image's DCD before full U-Boot enters DDR. This target has no SPL, and `dram_init()` cannot rescue an initializer that prevented execution.
- The quiet configuration separates console/storage inspection from automatic boot and persistent policy. RAM environment changes do not modify saved MMC settings.
- Disabling inherited video, network and USB-host groups prevents unrelated hooks from running before their MINI-specific assumptions are reviewed.
- `u-boot.imx` begins with its IVT at file byte zero. Its placement at SD byte `0x400` is a distinct media-layout decision, with boot and environment ranges kept outside partitions.
- Compilation, image inspection, media readback and a UART prompt establish different boundaries. None alone qualifies DDR margins, every storage sector or Ethernet behavior.

19.9 Pitfalls

- **A matching SoC is not matching board support.** The EVK is a useful comparison, not our first MINI firmware target.
- **Mixing source snapshots.** Factory-archive defconfig names and logs need not exist in the public tutorial snapshot. Follow the pinned tree.

- **Forgetting the vendor Makefile's compiler assignment.** Pass the environment's `CROSS_COMPILE` explicitly to `make`; do not install another cross compiler.
- **Treating a build as a board test.** The source and image checks here do not claim a measured boot or DDR qualification.
- **Double-wrapping the image.** Write the inspected `u-boot.imx` at 1 KiB. Do not combine it with a helper that adds another image header or pad.
- **Assuming the PHY never changed.** V2.2's SR8201F needs a revision-aware driver check, even though ENET2 and PHY address 1 remain familiar.
- **Saving settings to silence a warning.** Our initial image is deliberately nonpersistent. Investigate the selected image and backend instead.

19.10 Going deeper

- Pinned vendor defconfig.
- Vendor board implementation.
- Vendor ROM/DCD configuration.
- Supplied V1.81 guide: applicability table on pages 7-8, MINI/core description in Sections 5.2.3-5.2.5, MINI schematics in Section 5.4, and U-Boot Chapters 30-33.

Next we ask why some U-Boot images have an SPL and ours does not. Keep the MINI build: it is the concrete result against which we will compare that alternative route.

Chapter 20: U-Boot SPL: the missing link

Chapter 19 left us with a large U-Boot image and a small question: how could the ROM load it into DDR before U-Boot's C code had initialized that DDR?

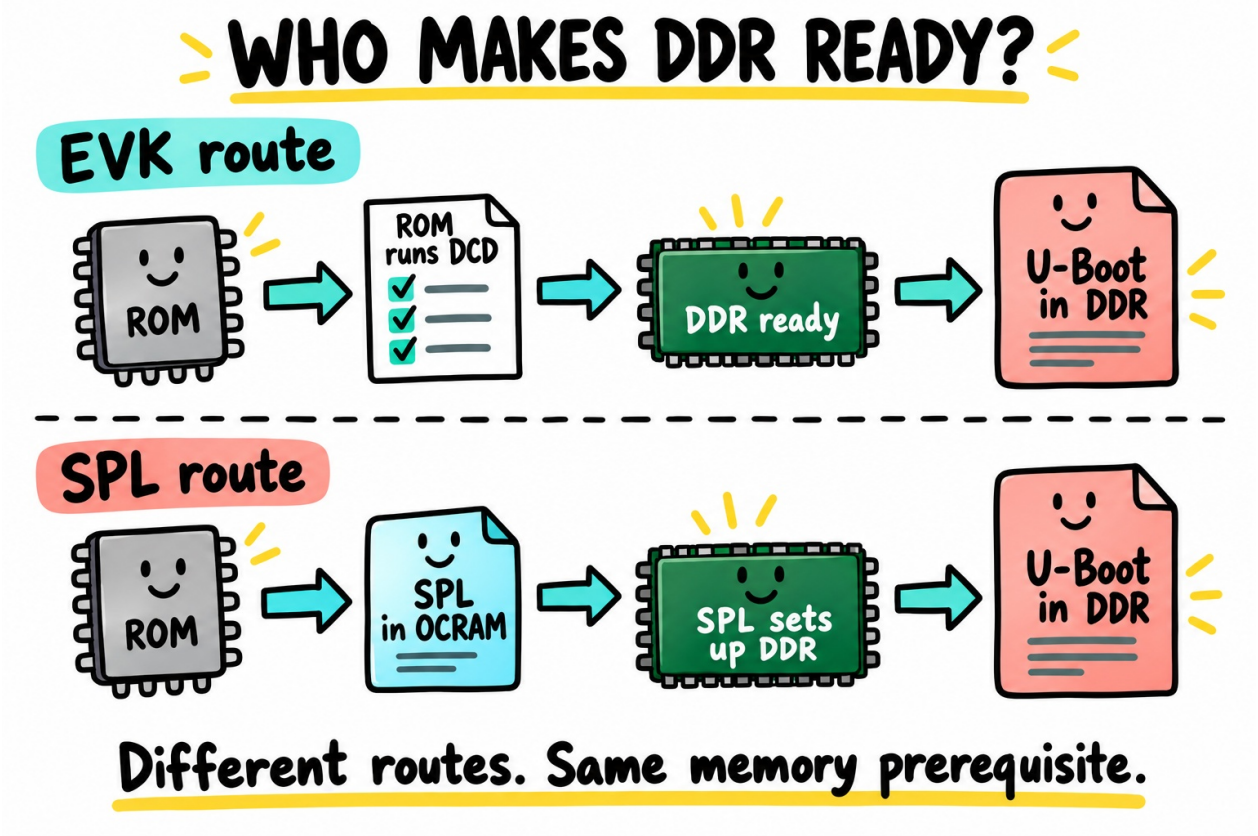
For our MINI vendor configuration, the answer was the DCD. The ROM interpreted its register writes first. Another board can give that job to a small program instead. That program is **SPL**, the Secondary Program Loader.

SPL is an alternative boot stage, not a file missing from our MINI build. Knowing which route you have is more useful than memorizing a universal "ROM, SPL, U-Boot" diagram. We will read an actual SPL implementation, then compare its responsibilities with the MINI's ROM-driven setup. The comparison uses another board's source on the host. It does not replace our MINI firmware.

20.1 Why two stages

DDR cannot hold a useful stack or executable image until its controller, pads, clocks, and memory device have been configured. The startup program must therefore begin in memory that is already usable.

Two routes solve that dependency on this SoC family:



Our MINI uses the upper route too. The EVK label names another example of ROM-interpreted DDR setup, not the board we need to buy. The lower route assigns that work to SPL code.

Route	Who prepares DDR?	Where the first application code runs
MINI ROM + DCD	The Boot ROM executes the image's DDR setup data	Full U-Boot begins in DDR
ROM + SPL	A small executable runs its board-specific DDR setup	SPL begins in internal RAM, then loads full U-Boot into DDR

Both routes still need correct board-specific DDR values. Moving the writes from a DCD into C does not remove the electrical or calibration work from Chapter 14. It changes where that knowledge is expressed and how much software can run before the large image is loaded.

Some SPL configurations can select among memory arrangements, perform more complex initialization, authenticate a payload, or boot an OS directly. Those are configured features, not promises made by the name SPL.

20.2 What SPL is responsible for

For a typical i.MX6-family SPL loading full U-Boot, the responsibilities are:

1. Establish a usable CPU mode, early stack, and global-data area.
2. Prepare the clock and pad paths needed by its console, DDR, and boot medium.
3. Initialize the fitted DDR arrangement.
4. Initialize the selected SD, eMMC, SPI, or other loader path.
5. Read and interpret the configured payload format.
6. Place that payload at its load address and transfer control to its entry.

A console is often made available before DDR setup so that an early failure can be reported. The exact order belongs to the board implementation. Do not reorder the calls just to make them resemble a generic diagram.

Likewise, “SPL cannot use networking” is too broad. Some builds include it. The usual engineering question is whether a feature is needed before the larger runtime is available, and whether its memory and dependencies fit.

20.3 The size budget

The i.MX6ULL has 128 KiB of OCRAM. That physical capacity is not a promise that all 128 KiB are available to a loaded program. ROM activity, image placement, the early stack, global data, and any early allocation pool also need space. Use the selected image format, ROM constraints, configuration, and linker map together. Do not infer a free window by subtracting an approximate reservation quoted for another board.

For the reference SPL used below, `CONFIG_SPL_MAX_SIZE` is `0x10000`, or 64 KiB. The Kconfig help describes this as the image limit **excluding BSS**. That is different from a limit on `text + data + bss + stack`.

In this family, an SPL may place BSS in DDR and clear it only after DDR setup. The early code must not use that BSS while DDR is unavailable. Its OCRAM code and stack budget and its later DDR budget are separate checks.

Record these fields from the generated configuration rather than copying another build’s size report:

Field or file	Check
`CONFIG_SPL_TEXT_BASE`	Where SPL code is linked

Field or file	Check
`CONFIG_SPL_MAX_SIZE`	Which loadable image limit the linker checks
SPL stack selection	Where the early stack begins and what lies below it
`CONFIG_SPL_BSS_START_ADDR` and `CONFIG_SPL_BSS_MAX_SIZE`	Where BSS lives and when that memory becomes usable
`spl/u-boot-spl.map`	Actual linked ranges and reservations
`spl/u-boot-spl`	ELF section sizes and addresses
`SPL`	The ROM-facing packaged file, including its header

Not every feature has a one-to-one `CONFIG_SPL_FOO` switch. Dependencies, shared framework code, and stage-specific defaults matter too. Inspect the expanded configuration after changing a feature.

20.4 Where SPL lives in the source

Our MINI baseline has no SPL to inspect. For this comparison and the modern internals examples that follow, create a **separate upstream v2026.04 study tree**. Keep the vendor tree and its MINI image from Chapter 19 unchanged:

```
$ cd ~/imx6ull/src
$ git clone --branch v2026.04 https://source.denx.de/u-boot/u-boot.git u-boot
$ cd u-boot
$ git rev-parse HEAD
88dc2788777babfd6322fa655df549a019aa1e69
```

If that directory already exists, inspect its revision and preserve local changes instead of cloning over it. This fixed release gives Chapters 20-24J one set of modern APIs to discuss. Their C patches belong here, **not** in the 2016 vendor tree. Chapter 22 maps the MINI's hardware into this newer framework.

We also need a full-U-Boot reference for Chapter 21's map and command example:

```
$ . ~/imx6ull/scripts/env.sh
$ make O="$HOME/imx6ull/build/uboot-evk" mx6ull_14x14_evk_defconfig
$ make O="$HOME/imx6ull/build/uboot-evk" -j4
```

This is a host-side comparison build, **not an image to flash to MINI**. Its host tools may need `libgnutls28-dev` in the Chapter 3 lab Ubuntu environment. That dependency does not change either project-local Arm toolchain.

Now use a real SPL-enabled configuration. In this v2026.04 tree, use **`pico-imx6ul_defconfig`** as a separate host-side reference. It describes a TechNexion PICO i.MX6UL platform, not the i.MX6ULL EVK and not our MINI. **Do not flash its outputs to either board.**

```
$ . ~/imx6ull/scripts/env.sh
$ cd ~/imx6ull/src/u-boot
$ make O="$HOME/imx6ull/build/uboot-pico-study" pico-imx6ul_defconfig
$ make O="$HOME/imx6ull/build/uboot-pico-study" -j4
$ arm-none-linux-gnueabihf-size \
    ~/imx6ull/build/uboot-pico-study/spl/u-boot-spl
```

The different `O=` directory keeps this configuration out of both the EVK study build and the MINI vendor build. The Arm-prefixed `size` program reads the target ELF on the host. It does not execute it. Its BSS count is useful, but does not describe stack usage or all packaging overhead. We are borrowing a source example, not changing the MINI's documented boot route.

Read these source files with the generated `.config` beside them:

Path	Responsibility
`board/technexion/pico-imx6ul/spl.c`	This board's early setup, DDR data, and MMC setup
`arch/arm/cpu/armv7/start.S`	Shared ARMv7 CPU startup
`arch/arm/lib/crt0.S`	Early C-runtime arrangements and stage-dependent paths
`common/spl/spl.c`	SPL framework and boot-device selection
`common/spl/spl_mmc.c`	MMC payload loading
`arch/arm/mach-imx/mx6/ddr.c`	Shared MMDC configuration routines
`common/spl/Kconfig`	Stage limits, formats, and loader defaults

An SPL file appears only when the build includes that stage. Looking for TI's `MLO`, or guessing an `.imx` filename, is not a way to repair a no-SPL EVK build.

20.5 Reading `start.S`

Open `arch/arm/cpu/armv7/start.S`, then follow its branch into `_main` in `arch/arm/lib/crt0.S`. The names are familiar from Part II, but the framework must accommodate more than our one experiment.

Startup work	What to look for
Preserve incoming boot information	`save_boot_params` and its return label
Select or respect CPU mode, mask IRQ/FIQ	The CPSR manipulation and conditional mode check
Establish architecture state	`cpu_init_cp15` and the selected critical-init path
Make early C calls possible	Initial stack selection, alignment, and global-data reservation in `_main`
Enter board setup	The call to `board_init_f`

The generic mode check is not evidence that this board normally enters in HYP mode. The handoff policy depends on the ROM or preceding firmware and the selected platform. Similarly, masking IRQ and FIQ does not make bad memory accesses harmless.

Notice the explicit `r9` global-data setup in `crt0.S`. A stack alone is not the whole early environment. The framework gives early code a place to retain state even before its normal runtime is ready.

In v2026.04, assembly and shared code use `CONFIG_XPL_BUILD` for these small loader stages. Older excerpts may say `CONFIG_SPL_BUILD`. Read the pinned tree's conditions, not just a function with a familiar name.

20.6 `board_init_f`, the "before relocation" stage

The PICO implementation's sequence can be summarized as:

```
clock gates
    -> architecture and board early setup
    -> timer
```

```
    -> preloader console
    -> board DDR configuration
    -> BSS clear
    -> SPL board_init_r
```

This is a reading map, not replacement code. Open the actual function to see the calls and the DDR data they use.

The order explains something important: this implementation clears BSS after DDR setup because its BSS location is in DDR. That region is not available for an early timeout counter merely because the C declaration compiled.

Also notice that this board's `board_init_f` calls `board_init_r` directly. Other SPL implementations return to `crt0.S`, which completes the stage's runtime setup. **The chosen board path decides.** A direct call is not a universal rule that every function named `board_init_f` never returns.

Find the PICO DDR geometry and calibration structures, then follow the call to `mx6_dram_cfg`. They are examples of how one supported board supplies data to common MMDC code. They are not calibration results for our MINI.

20.7 `board_init_r`, the "after relocation" stage

Full U-Boot uses this name for its post-relocation runtime. SPL keeps the name for its later framework phase, even when its code has not relocated.

Follow `board_init_r` in `common/spl/spl.c` into boot-order selection, registered loaders, and the selected image-loading function. The framework is not simply a hard-coded call with `BOOT_DEVICE_MMC1` on every board.

For the PICO raw-MMC reference, inspect `CONFIG_SYS_MMCSD_RAW_MODE_U_BOOT_SECTOR`. The family default is `0x8A`:

```
0x8A = 138 sectors
138 * 512 bytes = 70656 bytes = 69 KiB
```

That explains an offset you may have seen in other i.MX6 SPL instructions. It does **not** change the MINI image placement in Chapter 19. The loader's expected payload format matters as well: a legacy `u-boot.img`, a FIT, and a ROM-facing DCD image are not interchangeable just because they contain U-Boot.

The loader records a load address, entry address, and size in `struct spl_image_info`. On success, the selected handoff enters that payload. On failure, the configured boot order may try another supported loader or stop. Find the actual error path before interpreting a later boot message.

20.8 Comparing SPL to your Ch 11 image-builder

Our Chapter 11 file included a leading pad, an IVT at file offset `0x400`, BootData, a header gap, and an OCRAM payload. U-Boot's packaged `SPL` has its own placement convention. Inspect the artifact rather than assuming that its name implies the same leading pad.

```
$ xxd -l 32 ~/imx6ull/build/uboot-pico-study/SPL
$ ~/imx6ull/build/uboot-pico-study/tools/mkimage -l \
    ~/imx6ull/build/uboot-pico-study/SPL
```

For this packaged SPL, examine the IVT at file byte zero. Its entry and self pointers refer to internal RAM rather than the DDR entry of the EVK image. Use the image tool and selected configuration to interpret the remaining fields. Do not substitute the entry from a full-U-Boot `.imx` dump.

The tool may print `Mode: DCD` for both files. In this implementation that label distinguishes the non-plugin header mode, not proof that a DDR setup table is present. The PICO wrapper selected by `arch/arm/mach-imx/spl_sd.cfg` does not supply the EVK's DDR writes. Check the actual DCD pointer and table before deciding who initialized memory.

An OCRAM-loaded SPL does not need the ROM to initialize DDR just to load the SPL itself. Its own board code can do that afterward. A DDR-loaded full image does need DDR before it can execute, which is why the EVK supplies a DCD.

The meaningful comparison is the dependency, not an identical filename or an identical amount of padding:

> Which memory must already work before the next instruction can be fetched?

20.9 The SPL-to-U-Boot handshake

`spl_image_info` is the SPL loader's record. It tells the handoff where an image was placed and where to enter it. It is not automatically an argument that full U-Boot reads from the SPL's old stack.

The payload also depends on established state:

- Its code and required data must actually be present in usable memory.
- Its load and entry addresses must agree with the selected format and build.
- Memory writes must be visible to the instruction-fetch path when needed.
- CPU mode, caches, MMU, and device state must satisfy the selected entry path.

The exact cache-maintenance and jump behavior belongs to the selected loader, architecture, and configuration. Do not infer that every SPL automatically performs `cleanup_before_linux()` before every possible handoff. Read the selected `jump_to_image_no_args` implementation and any board override.

Some configurations provide additional handoff records through facilities such as a blob list. Those are separate interfaces. For our reading example, trace the normal full-U-Boot entry into `crt0.S`; it establishes its own runtime instead of treating the preceding program's stack as its permanent stack.

20.10 Key Notes

- DCD and SPL solve the same early-memory dependency with different owners: ROM interprets a table, while SPL executes board-specific initialization code.
- The PICO SPL and EVK full-U-Boot sources are comparison cases, not MINI firmware. Their payload formats and raw-MMC offsets do not replace Chapter 19's layout.
- A loadable-image size limit excludes other runtime consumers. OCRAM code, stack and early allocation, plus any later DDR-resident BSS, have separate lifetimes and budgets.
- The PICO path clears its DDR-resident BSS only after DDR initialization. A compiled C variable is not usable before its backing memory exists.
- `board_init_f` and `board_init_r` names do not impose universal return or relocation behavior. The selected board and stage determine the actual handoff.
- `spl_image_info` records placement and entry information. The next stage also depends on CPU, cache and memory state, not merely a function-pointer jump.

20.11 Pitfalls

- **Looking for SPL in every build.** Check `CONFIG_SPL` first. Our EVK does not use it, and a missing file there is expected.
- **Treating 64 KiB as the whole runtime budget.** The loadable image limit, BSS location, early stack, header, and allocation pool are different facts.
- **Accessing DDR-resident BSS before DDR setup.** Successful compilation says nothing about whether that address is usable at that moment.
- **Copying another board's DDR structures.** They describe its memory and calibration, not a generic i.MX6UL/ULL memory device.
- **Using full-U-Boot DT assumptions for SPL.** Stage filtering and available drivers determine what an SPL can bind and probe. Inspect the stage's output.

- **Guessing handoff cleanup.** Follow the actual path and its ownership rules. A function-pointer call does not by itself synchronize cached instructions.

20.12 Going deeper

- SPL development documentation.
- Pinned PICO configuration.
- PICO board SPL.
- SPL Kconfig limits and loader settings.
- The common ARM runtime entry.

Next we return to full U-Boot. Whether the ROM or an SPL made DDR usable, the large program still has to arrange its own stack, relocation, devices, and shell.

Chapter 21: U-Boot internals

At the => prompt, U-Boot looks like a small terminal. Type a command and it answers. Behind that simple exchange, the program has already changed its stack, arranged a heap, moved its code, located its devices, and loaded an environment.

We do not need to understand all of U-Boot at once. We need a route from a visible result to the code responsible for it. Start with the prompt. Work backward to the runtime that made it possible, then follow one command forward.

Keep the **v2026.04 study tree and EVK comparison build from Section 20.4** open. The MINI image from Chapter 19 stays in its separate vendor output directory. We use the newer tree here to learn current source organization and APIs, not to substitute an EVK image for the MINI. Its ROM + DCD route has already made DDR usable before full U-Boot executes. Chapter 20's SPL route can reach the same full-U-Boot entry through a different earlier stage. Neither route changes the distinction between the linked address and the address at which the relocated program will run.

21.1 The full-U-Boot boot flow, end to end

For the selected ARMv7 build, use this reading map:

```
vectors.S: _start -> reset
    start.S: CPU setup
        crt0.S: _main
            early stack and global data
            board_init_f
                common/board_f.c: ordered early init calls
            return to crt0.S
            relocate_code, then resume at the relocated label
            vector and final runtime setup, BSS clear
            board_init_r
                common/board_r.c: later init calls
                main_loop -> autoboot or command interpreter
```

This is a map, not an assembly listing to paste into a port. Conditional features add work to the actual paths. The useful habit is to locate a name, read its caller, and check which branch your `.config` selects.

Two facts will matter throughout the chapter:

- On this ARM full-U-Boot path, **`board_init_f` returns to `crt0.S`**. The assembly then arranges the relocation handoff.
- `board_init_r` reaches the main loop and is not expected to return to an earlier runtime. These two functions do not have identical return behavior.

21.1a The linker script and the named address-range symbols

Before following the CPU, look at the address ranges it will use. The build preprocesses `arch/arm/cpu/u-boot.lds` into `u-boot.lds` in the output directory. The linked ELF and `u-boot.map` show the resulting addresses.

```
$ . ~/imx6ull/scripts/env.sh
$ arm-none-linux-gnueabihf-nm -n ~/imx6ull/build/uboot-evk/u-boot \
    | grep -E '__image_copy_|__rel_dyn_|_image_binary_end|__bss_'
$ arm-none-linux-gnueabihf-readelf -S ~/imx6ull/build/uboot-evk/u-boot
```

These commands inspect target files on the host. The first filters the symbol list so that the ranges are easy to compare. Read addresses from your output, not a sample map from an unrelated compiler.

Symbol	Meaning in this ARM linker layout
`_start`, `__image_copy_start`	Beginning of the main image, linked at `CONFIG_TEXT_BASE`
`__image_copy_end`	End of the region that the normal relocation copy loop copies
`__rel_dyn_start`, `__rel_dyn_end`	Relocation records consumed while fixing the copied image
`_image_binary_end`	End marker for the linked binary layout
`__bss_start`, `__bss_end`	Runtime zero-initialized storage range

The EVK text base is `0x87800000`. The other values depend on the build. Also notice an unusual economy in the linker script: BSS overlays the relocation-table region. These bytes have different jobs at different times. The relocation records are needed first. BSS is cleared afterward, once that old role is finished. A simple drawing that places BSS after every file-backed section misses this lifetime relationship.

This is why the map matters more than a familiar-looking `size` total. Section types, overlays, copied ranges, and runtime reservations all contribute to the memory story.

In this build the overlay also affects the inspection tools: `size` can report zero BSS, and `nm` can mark its boundary symbols with `?`. That does not remove the runtime zeroing job. Check the actual `__bss_start`/`__bss_end` range and the linker section flags rather than treating that one summary column as the whole memory requirement.

21.2 `_start` and `_main`

`_start` belongs to `arch/arm/lib/vectors.S`. Its reset vector reaches the ARMv7 startup in `arch/arm/cpu/armv7/start.S`. That distinction helps when a debugger lands at the first instruction and you want to identify the next one.

`start.S` handles the selected CPU setup and reaches `_main` in `arch/arm/lib/crt0.S`. Early in `_main`, the code obtains an initial stack address, aligns it, reserves early memory, establishes `r9` as the global-data pointer, and calls `board_init_f`.

Global data, `struct global_data`, is commonly called **`gd`**. It holds state such as memory information, relocation addresses, flags, and pointers to runtime structures. On this ARM path, a fixed register gives code access to it. It is not a general substitute for a fully initialized C runtime.

Walking the SP arithmetic for the EVK

The board header uses `CFG_SYS_INIT_RAM_ADDR` and `CFG_SYS_INIT_RAM_SIZE` for OCRAM. `include/system-constants.h` derives `SYS_INIT_SP_ADDR` using the generated global-data size, unless a custom address is selected.

The default relationship is:

```
initial top = OCRAM base + OCRAM size - GENERATED_GBL_DATA_SIZE
align initial top down to an 8-byte boundary
reserve the selected early allocation pool, if present
reserve struct global_data and align it down to a 16-byte boundary
establish the new SP and gd pointer
```

Read the generated size in `include/generated/generic-asm-offsets.h`, the selected allocation size in `.config`, and the allocation function in `common/init/board_init.c`. Do not assume that GD is always 248 bytes or that subtracting GD alone gives the final SP. Configuration changes affect both GD and its neighboring reservations.

The EVK's code can already be in DDR while this early stack is in OCRAM. Code location and stack location are separate decisions. Later, the framework reserves the final stack and GD in DDR and switches to them.

Early code must respect this limited runtime. Do not introduce ordinary BSS state into a function just because its name contains `init`. Follow the entry path and the moment that storage becomes valid.

21.3 `board_init_f` and the init-function sequence

Open `common/board_f.c` and find `initcall_run_f`. In v2026.04 it is an ordered sequence of `INITCALL(...)` operations, with configuration conditions and events between them. It is not the older `init_sequence_f[]` array you may see in another guide.

The macro in `include/initcall.h` calls the named function. A nonzero result reports the failed call and stops through `hang()`. The order therefore tells you both the dependencies and where an early failure can stop progress.

Use these landmarks rather than trying to memorize the whole function:

Landmark	Responsibility
`fdtdec_setup`	Locate U-Boot's control device tree
`initf_malloc`, `initf_dm`	Arrange early allocation and driver-model state
`arch_cpu_init`, board early hook, `timer_init`	Selected architecture, board, and timing setup
`env_init`, baud and serial calls	Establish early environment state and console access
`display_options`, board/CPU reporting	Make some of the established state visible
`dram_init`	Establish available DRAM information for this board path
Reservation calls	Plan space for code, heap, board info, GD, FDT, and stacks
`setup_reloc`	Retain the addresses and offset required for the next phase

For the EVK DCD route, DDR was already initialized by the ROM. `dram_init` does not need to replay a generic DDR training recipe. On another platform, the same hook name may do more. Read the board implementation before assigning it a universal meaning.

Likewise, `env_init` is early environment initialization. It does not imply that the saved MMC environment has already been imported. The later `initr_env` path in `common/board_r.c` handles that normal load/import step.

`board_init_f` returns with a plan in GD: relocation destination, new stack, new GD location, and associated reservations. `crt0.S` then changes SP and GD, adjusts its return label by the relocation offset, and passes the destination in `r0` to `relocate_code`.

That return is important. The C function planned the move. The assembly performs the move and resumes execution in the new copy.

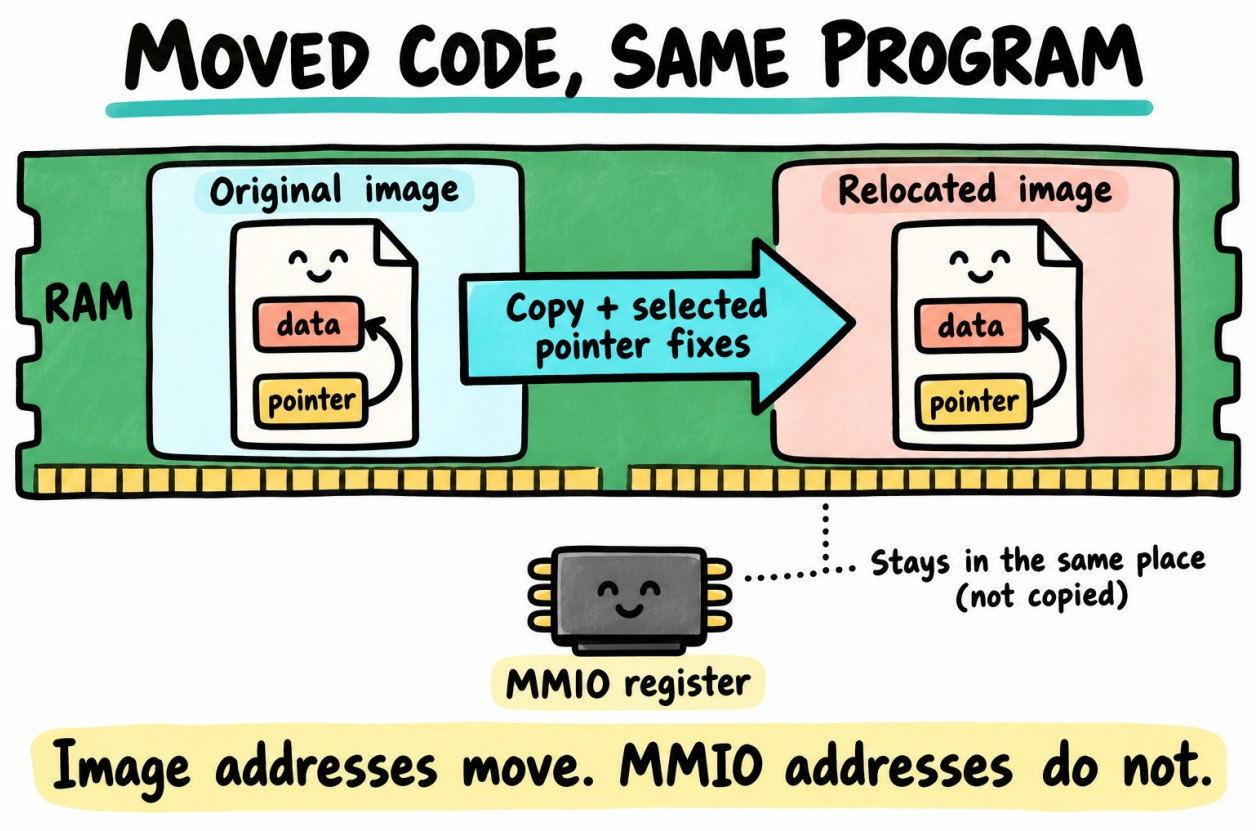
21.4 Relocation, the trick that confuses everyone

U-Boot is already in DDR, so why move it again?

The ROM image selected a load location before the full runtime had arranged the board's memory reservations. The runtime now knows more: usable DRAM, space needed for its heap, device-tree data, stack, and possibly a framebuffer. It can place itself so that later payload loading has a separate workspace.

Relocation does not make every low address automatically safe for the kernel. Loaded images, reserved regions, expansion space, and U-Boot's own allocations still need a non-overlapping plan.

How relocation works mechanically



The RAM strip is a conceptual container, not a drawing of the board's memory hardware. Selected relocation records adjust image-relative addresses. Moving U-Boot does not move the peripheral registers it accesses.

Read `arch/arm/lib/relocate.S`. At its entry, `r0` is the destination. The code derives the source range from linker-defined offsets, copies `__image_copy_start` through `__image_copy_end`, and walks the relocation records.

For an `R_ARM_RELATIVE` record, its work can be expressed as:

```
new location of the word = old location of the word + relocation offset
new value stored there  = old value stored there + relocation offset
```

These are two different additions. One finds the word in the copied image. The other corrects the address stored in that word. Not every instruction or arbitrary integer is patched, and not every relocation type is handled the same way. Follow the selected type test in the actual loop.

Finally, the routine returns through the already adjusted link register. Execution resumes at `here` in the relocated copy of `crt0.S`. The selected runtime setup follows, including BSS clearing, before entering `board_init_r`. The linker's overlay now makes sense: those storage bytes can take on their later BSS role after the relocation records have been consumed.

What you observe

`bdinfo` reports `relocaddr` and `reloc off`. Compare them with the build's `CONFIG_TEXT_BASE`. For the normal linked-load path, the relationship is:

```
relocation offset = relocated address - linked address
```

Do the subtraction using values from your session. Do not compare against `0x80800000` simply because an older board example used that text base.

Also, `relocaddr` in `bdinfo` is not automatically an environment variable named `${relocaddr}`. A report field and a shell variable are different things. There is no need to poke either address to understand the relationship.

Relocation also does not instantly erase the old image. Its bytes can remain until another operation reuses that range. Old bytes are not evidence that the CPU is still executing that copy.

21.5 The command system

Once the shell has parsed a line, command processing finds a registered command, checks its argument count, and invokes its callback. Trace `common/cli.c`, `common/command.c`, and `include/command.h` for your selected parser.

`U_BOOT_CMD(...)` declares a command entry through U-Boot's linker-list mechanism. The linker gathers entries, and `find_cmd` searches that list. There is no handwritten central array to update for every command.

Adding a `hello` command

This small command does no hardware access. We give it a distinctive name, `book_hello`, rather than risk colliding with another custom command.

First preserve the pinned revision on a local working branch:

```
$ cd ~/imx6ull/src/u-boot
$ git switch -c book-uboot-labs
```

Create `cmd/book_hello.c` with your editor:

```
#include <command.h>
#include <stdio.h>

static int do_book_hello(struct cmd_tbl *cmdtp, int flag,
                         int argc, char *const argv[])
{
    if (argc == 2)
        printf("Hello, %s.\n", argv[1]);
    else
        puts("Hello from U-Boot.\n");

    return CMD_RET_SUCCESS;
}

U_BOOT_CMD(
    book_hello, 2, 1, do_book_hello,
    "print a book-lab greeting",
    "[name]"
);
```

The `2` is the maximum argument count **including the command name**. The `1` marks the command repeatable. The callback returns a command status, which matters when a script uses it inside `if`.

Add this entry to `cmd/Kconfig`, beside the other command options:

```
config CMD_BOOK_HELLO
    bool "book_hello - Print a book-lab greeting"
    depends on CMDLINE
    help
      Enable a greeting command with no hardware side effects.
```

Add its object selection to `cmd/Makefile`:

```
obj-$(CONFIG_CMD_BOOK_HELLO) += book_hello.o
```

Finally, add `CONFIG_CMD_BOOK_HELLO=y` to the EVK study defconfig in your working tree, then rerun the two Section 20.4 EVK build commands with the same output directory. This edits the configuration input, rather than relying on a hand-edited generated `.config` that a later defconfig command replaces.

Only run the resulting board image after the applicable board/image checks. MINI readers can compile and inspect the ELF now, then carry this command into their qualified Chapter 22 port. Expected command behavior is:

```
=> book_hello
Hello from U-Boot.
=> book_hello reader
Hello, reader.
```

These are expected strings from the callback, not a claimed board test. `book_hello one two` exceeds the registered argument limit and should produce usage instead. The dispatcher rejects it before the callback runs.

21.6 The environment

The environment has two lives: a set of variables currently in RAM and, when a storage backend is configured, a saved representation that can survive reset. Changing the first does not automatically change the second.

What it is and where it lives

The common stored data uses NUL-separated `name=value` strings with integrity metadata such as a CRC. A redundant layout adds its own selection information. Backend choices include raw MMC storage, flash, filesystems, and other platform arrangements. The environment does not have to live on the boot medium or outside every filesystem.

For the EVK reference, the defconfig selects MMC storage, size `0x2000`, offset `0xC0000`, and default MMC device index 1. The selected runtime device can also depend on board/SoC hooks. Record the actual backend, device, hardware partition, offset, size, and any redundant copy before enabling a persistent write.

Compiled defaults may come from a board `.env` file or settings such as `CFG_EXTRA_ENV_SETTINGS`; this EVK uses the latter in `include/configs/mx6ullevk.h`. A valid saved environment can replace them. Rebuilding a default value does not guarantee that an existing board will start using it.

Reading and writing

Try a RAM-only variable first:

```
=> setenv book_lab visible
=> printenv book_lab
=> setenv book_lab
```

Giving `setenv` no value removes that variable. `env save` or its `saveenv` alias is a separate storage operation. For a first experiment, leaving changes unsaved is deliberate, not an error.

Before a persistence test, inspect the configured storage layout and reserve both environment copies if the build uses redundancy. Use a qualified spare lab device and retain a known-good recovery route. A bad CRC may indicate a blank region, a layout mismatch, corruption, or a failed read; saving over the region is not a general diagnosis.

From U-Boot scripts and from C code

For a harmless script example:

```
=> setenv book_steps 'echo first; echo second'
=> run book_steps
```

Single quotes store the command text. `${name}` references within such a stored script are expanded when the script later runs. Chapter 23 uses that distinction for boot policy. It also explains why a semicolon alone does not stop a boot sequence after an earlier load fails.

The C interfaces are declared in `<env.h>`:

```
const char *ip = env_get("ipaddr");
ulong address = env_get_hex("loadaddr", 0x82000000);
int ret = env_set("autoload", "no");
```

This is an API illustration, not a complete command. `ip` can be `NULL`. Check `ret` before relying on a change. An `env_get` pointer refers to the environment’s storage; do not retain it across mutations without considering its lifetime.

From Linux user-space

On the running target, `fw_printenv` and `fw_setenv` can use a matching `/etc/fw_env.config` to access the saved representation. Their storage device and byte ranges must match U-Boot’s actual backend and layout, including redundancy. A guessed configuration can overwrite unrelated data.

These are target Linux tools in this workflow, not an instruction to point a host PC at whatever `/dev/mmcblk0` it happens to have. Chapter 24F returns to their role in an update policy. A CRC is an integrity check, not authorization to let an untrusted writer control boot settings.

21.7 The driver model (DM)

Suppose board code needs a UART without knowing which controller implements it. It needs a shared interface, a driver that implements that interface, and an instance representing this particular controller.

Concept	Example
`struct udevice`	The instance associated with a selected UART node
`struct driver`	The implementation that knows this UART controller
A uclass, such as `UCLASS_SERIAL`	The category and common operations used by callers

Device Tree properties and a driver's compatible table help DM **bind** an instance to a driver. Binding is not the same as successful **probing**. Probing activates the device and can fail while obtaining clocks, pads, power, or parent-bus resources. It commonly happens when a caller first needs the device, rather than immediately for every node in the tree.

Read the real UART driver, `drivers/serial/serial_mxc.c`, and the serial uclass beside it. Find its compatible table, `U_BOOT_DRIVER` declaration, private-data allocation, and operations. Those entries connect framework requests to the controller behavior you learned in Chapter 12.

Serial operations also have a contract. For example, a nonblocking receive operation can return `-EAGAIN` when no character is ready. Returning a fabricated zero byte instead changes the meaning seen by the caller.

Early DM needs the devices, drivers, data, and allocations appropriate to the pre-relocation stage. A node that is usable later is not automatically usable before relocation. SPL filtering introduces another stage distinction. Follow the actual phase properties and driver flags in the pinned tree.

U-Boot's **control FDT** describes devices U-Boot uses. The **OS FDT** supplied to a kernel describes that kernel's hardware view. They may share source material, but they are separate objects with separate lifetimes. A reported `fdt_blob` or `fdtcontroladdr` is not automatically the blob that `bootz` passes to Linux.

21.8 Reading a real boot, end to end

Use your own qualified board log, or study the source without claiming a run. For the modern EVK comparison route, map these kinds of messages to their owners:

Observation	Source area to inspect
U-Boot banner	Early reporting, including `display_options`
CPU/model/reset information	Selected architecture and board-reporting hooks
DRAM announcement	`announce_dram_init`, board `dram_init`, and later reporting
MMC devices	Selected MMC initialization and device probes
Loading a saved environment	Post-relocation `initr_env` and the chosen backend
Serial input/output selection	Console initialization and environment choices
Autoboot countdown	Main loop and autoboot policy
=> prompt	Command interpreter

No PMIC message is guaranteed for every board. No Ethernet line proves that a transfer to your server will work. No SPL banner is expected for our no-SPL EVK configuration. Read the messages as observations about the selected path, not a checklist of lines every board must imitate.

When something stops, find the last established dependency. A console that works before relocation and goes quiet afterward suggests a different set of questions from a ROM load that never reaches UART setup.

21.9 Key Notes

- The ARM full-U-Boot path divides work between C planning and assembly handoff. `board_init_f` records reservations in GD, then `crt0.S` changes stack/GD and relocates execution.
- Early stack and global data do not imply a complete C runtime. Generated sizes, alignment and the early allocation pool determine the actual OCRAM footprint.

- Relocation copies a defined image range and interprets relocation records. BSS reuses the relocation-table storage only after those records have served their purpose.
- Command registration joins a callback, argument/status contract, Kconfig selection and linked object. `U_BOOT_CMD` contributes to linker lists rather than a manually maintained central array.
- RAM environment, compiled defaults and imported persistent settings are distinct states. Rebuilding defaults neither replaces saved settings nor authorizes a storage write.
- Driver-model binding identifies an implementation, while probing acquires and initializes its dependencies. U-Boot's control FDT remains separate from the OS tree prepared for Linux.

21.10 Pitfalls

- **Assuming every init hook has the same role on every board.** Follow the selected caller, configuration, and implementation together.
- **Using normal global state too early.** Early stack and GD setup do not guarantee that BSS or every writable-data assumption is valid yet.
- **Patching every integer by `reloc_off`.** The relocation loop interprets selected records. An MMIO address is not a pointer into U-Boot's copied image.
- **Expecting new defaults to override a saved environment.** Inspect the currently imported values before changing storage or blaming the build.
- **Confusing control and OS device trees.** Do not edit or hand off U-Boot's active control tree as though it were an unused kernel load buffer.
- **Registering duplicate command names.** Lookup order and abbreviations are not a conflict-resolution mechanism. Choose a distinct name.
- **Reflashing to clear an environment.** The environment often occupies a separate range. Replacing the executable need not replace saved settings.

21.11 Going deeper

- ARM runtime entry.
- Relocation implementation.
- Early init sequence.
- Environment documentation.

- Driver-model design.

Chapter 22 changes the board description. We can now separate what belongs to our MINI from the common runtime that gets every configured board to its prompt.

Chapter 22: Porting U-Boot to the MINI

Chapter 19 builds PointAtom's existing support for the MINI's core. Why port anything now? Understanding that vendor recipe and maintaining the same hardware on a newer upstream release are different jobs. Ask **which settings describe our core, and which describe another carrier or PHY?** A successful EVK build cannot answer that question.

Keep two source workspaces. The vendor tree, `~/imx6ull/src/uboot-mini-vendor`, is pinned to `edb7ca5ac4be2d978be60a2a12c61e0b6d1f7feb` with `mx6ull_alientek_emmc_defconfig`. It is the documented starting recipe for the **512 MiB DDR3L / 8 GB eMMC core shared by ALPHA and MINI**. The migration in this chapter uses `~/imx6ull/src/u-boot`, based on **upstream v2026.04**, commit `88dc2788777babfd6322fa655df549a019aa1e69`. Old vendor header macros and modern Kconfig APIs are not interchangeable.

The hardware references are the supplied **MINI V2.2 carrier and CORE V2.0 schematics**. The guide's PDF pages 7-8 and 236-239 establish shared tutorial/core support; pages 263-283 describe MINI circuits, and Chapter 33 (860-911) explains the original vendor port. Its older LAN8720A procedure is not automatically support for V2.2's SR8201F. When carrier prose and the supplied schematic disagree, follow the schematic and populated revision.

> **Port status:** This is a concrete, host-buildable modern MINI candidate, not an EVK image with a new name. It implements documented UART1, SD, eight-bit eMMC and ENET2 wiring, with a source-checked SR8201F driver candidate. No board boot, DDR qualification, Ethernet test or media write is claimed. Keep Chapter 19's vendor UART/storage build as the baseline; replacing a persistent bootloader requires the gates in Section 22.8.

22.1 What "porting" means

On an MCU you compare clock, pad and peripheral tables. Here ordering adds another constraint: ROM initializes DDR from **DCD**, Device Configuration Data, before full U-Boot runs there. Both selected board paths use ROM+DCD, **not SPL**.

Create a port ledger in `~/imx6ull/notes/mini-port.md`. This is its document-checked starting content, conditional on matching supplied revisions and populated components:

Chain	Hardware evidence	Implementation decision
DDR	Core sheet 3: one NT5CC256M16EP-EK x16 DDR3L, 256M x 16 bits = 512 MiB; sheet 1: `DRAM_1V35`	Compare the published vendor DCD; modern DT reports 512 MiB. Physical qualification remains separate.
Power	Core sheet 1 and MINI sheet 4: discrete converters/LDOs, `PMIC_ON_REQ` sequencing	No invented EVK PFUZE3000 or GPIO-controlled peripheral rail. Actual rails must be ready before DCD.
Console	MINI sheet 4: UART1 TX/RX through analog switches to CH340C and `USB_TTL`	UART1 DCE TX/RX, no flow control; kernel example console ttymxc0. `USB_TTL` is not ROM USB download.
SD	MINI sheets 1-2: SD1 CLK/CMD/DATA0-3, CD on `UART1_RTS_B` / `GPIO1_IO19`, VDD from `DCDC_3V3`	USDHC1, four bits, active-low CD; no copied EVK `GPIO1_IO09` power switch.
eMMC	Core sheet 4: KLM8G1GETF, SD2 CLK/CMD/DATA0-7 on NAND pads; `SD2_nRST` on `NAND_ALE` / `GPIO4_IO10`; sheet 1: `NVCC_NAND` at 3.3 V	USDHC2, eight bits, non-removable, no 1.8 V switching. Vendor `USDHC2_PWR_GPIO` is reset, not a rail enable.
Ethernet	MINI sheet 3: one SR8201F, ENET2 RMII, MDIO address 1, reset `SNVS_TAMPER8` / `GPIO5_IO08`	One MAC, with actual PHY-ID/driver matching, not copied Micrel/SMSC assumptions.
Ethernet clock	MINI sheet 3: PHY 25 MHz crystal, CLK_CTL pull-up for TXC input, `ENET2_TX_CLK` routed to TXC	SoC supplies the separate 50 MHz RMII reference. Crystal frequency is not RMII reference frequency.
Optional I/O	MINI sheets 1-2: LED0 `GPIO1_IO03`; `KEY0` `UART1_CTS` / `GPIO1_IO18`, active low; active buzzer `SNVS_TAMPER1`	Exclude these, display/touch, camera, CAN, Wi-Fi and USB from the minimum boot port. TF and optional SDIO Wi-Fi share USDHC1.

Inspect actual DDR/eMMC/PHY markings before using this ledger on hardware. A NAND-populated core, another DDR population or an older carrier is a different target. The guide identifies **LAN8720A before MINI V2.2 and SR8201F from V2.2 onward**, both on ENET2/address 1. An address does not identify a driver.

MINI has one RJ45 despite a copied ALPHA “two interfaces” sentence. On the supplied schematic, `KEY0` is connector B48 / `UART1_CTS` / `GPIO1_IO18` with R12 and an active-low switch. `SNVS_TAMPER0` is `WIFI_REG_ON`, not `KEY0`; `GPIO1_IO01` at B49 is `GBC_KEY/AP_INT` for the ATK module. Do not confuse those three nets. CAN1 uses `UART3 CTS/RTS`, not the console’s pads.

22.2 The files that connect a board to U-Boot

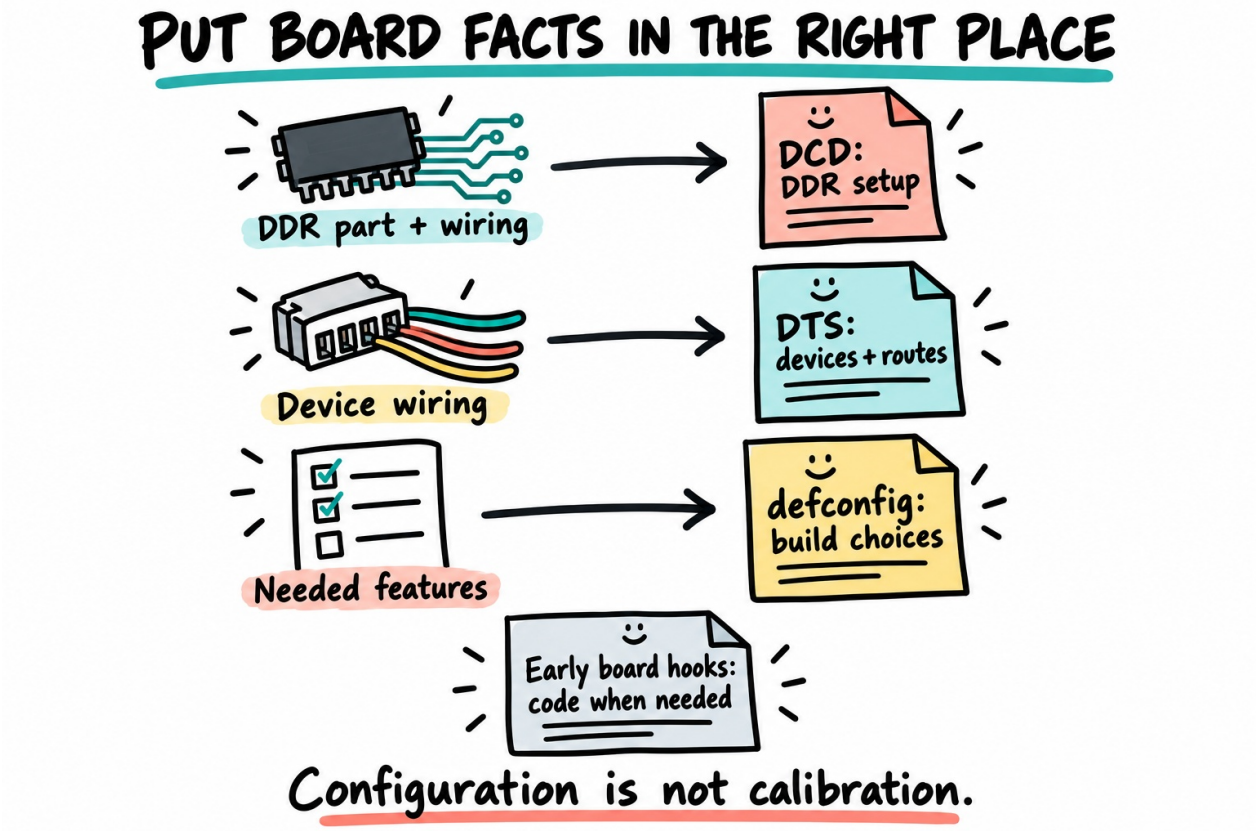
The vendor support already exists in `board/freescale/mx6ull_alientek_emmc/`, with `include/configs/mx6ull_alientek_emmc.h`. Its small defconfig passes an old `IMX_CONFIG` path through `CONFIG_SYS_EXTRA_OPTIONS`. In modern v2026.04 the reference is **`board/nxp/mx6ullevk/`**, and board Kconfig supplies `IMX_CONFIG`. Neither path is a universal naming rule.

Our modern local port uses:

```
board/myorg/mx6ull_pa_mini/
    Kconfig
    Makefile
    mx6ull_pa_mini.c
    imximage.cfg
    MAINTAINERS
include/configs/mx6ull_pa_mini.h
configs/mx6ull_pa_mini_defconfig
arch/arm/dts/imx6ull-pa-mini.dts
arch/arm/dts/imx6ull-pa-mini-u-boot.dtsi
```

`myorg` is a teaching namespace, not a registered DT vendor prefix. Underscores name the target/header; hyphens name the DT. A real upstream submission needs an appropriate prefix/binding and actual maintainers.

Source the board Kconfig from `arch/arm/mach-imx/mx6/Kconfig` and list its DTB in `arch/arm/dts/Makefile`. This follows the selected reference’s legacy-DTS route, not a silent migration to `dts/upstream/`.



Board facts belong to different files. In this ROM+DCD route, DDR setup must be qualified before U-Boot can use it. A new filename or a successful build does not supply that qualification.

Read the pinned vendor board source beside the modern reference. Carry forward hardware evidence, not every inherited optional hook.

22.3 Step 1, Create the modern MINI target

Work in your own modern checkout. Chapter 21 may already have added commits, so HEAD need not equal the release tag:

```
$ . ~/imx6ull/scripts/env.sh
$ cd ~/imx6ull/src/u-boot
$ git rev-parse 'v2026.04^{commit}'
$ git merge-base --is-ancestor v2026.04 HEAD
$ echo $?
$ git status --short
$ git log --oneline v2026.04..HEAD
```

The tag must resolve to `88dc2788777babfd6322fa655df549a019aa1e69`; the ancestry check must return zero. The immediately following `echo $?` reads its silent result. Stop on failure instead of resetting your work. Review extra commits and uncommitted changes too: ancestry establishes the base, not today's entire source. See Git's ancestry check.

For destinations that do not already exist:

```
$ mkdir -p board/myorg
$ cp -a board/nxp/mx6ullevk board/myorg/mx6ull_pa_mini
$ mv board/myorg/mx6ull_pa_mini/mx6ullevk.c board/myorg/mx6ull_pa_mini/mx6ull_pa_mini.c
$ cp include/configs/mx6ullevk.h include/configs/mx6ull_pa_mini.h
$ cp configs/mx6ull_14x14_evk_defconfig configs/mx6ull_pa_mini_defconfig
```

Inspect existing files rather than overwriting them. The new C file is untracked, so ordinary `mv` is appropriate. Preserve copied licence/copyright notices. We create the DTS afresh instead of retaining the EVK board include and its unrelated regulators, codec and shift register.

In the board-choice block of `arch/arm/mach-imx/mx6/Kconfig`, add:

```
config TARGET_MX6ULL_PA_MINI
    bool "PointAtom MINI V2.2 / eMMC core (migration candidate)"
    depends on MX6ULL
    select BOARD_LATE_INIT
    select DM
    select DM_THERMAL
    select IOMUX_LPSR
    imply CMD_DM
```

Near its existing board source lines add:

```
source "board/myorg/mx6ull_pa_mini/Kconfig"
```

Replace the copied board Kconfig with:

```
if TARGET_MX6ULL_PA_MINI

config SYS_BOARD
    default "mx6ull_pa_mini"

config SYS_VENDOR
    default "myorg"

config SYS_CONFIG_NAME
    default "mx6ull_pa_mini"

config IMX_CONFIG
    default "board/myorg/mx6ull_pa_mini/imximage.cfg"

endif
```

In the copied Makefile preserve notices and replace the object line:

```
obj-y := mx6ull_pa_mini.o
```

No SPL object is added. Update the inactive PLUGIN path in `imximage.cfg` to `board/myorg/mx6ull_pa_mini/plugin.bin`; keep `CONFIG_USE_IMXIMG_PLUGIN` disabled. Section 22.6 checks the actual DDR sequence rather than leaving copied writes unexplained.

22.4 Step 2, Edit the defconfig and header

Replace the copied defconfig with this complete modern minimum:

```
CONFIG_ARM=y
CONFIG_ARCH_MX6=y
CONFIG_TEXT_BASE=0x87800000
CONFIG_SYS_MALLOC_LEN=0x1000000
CONFIG_NR_DRAM_BANKS=1
CONFIG_ENV_SIZE=0x2000
CONFIG_MX6ULL=y
CONFIG_TARGET_MX6ULL_PA_MINI=y
CONFIG_DM_GPIO=y
CONFIG_DEFAULT_DEVICE_TREE="imx6ull-pa-mini"
CONFIG_SUPPORT_RAW_INITRD=y
CONFIG_USE_BOOTCOMMAND=y
CONFIG_BOOTCOMMAND=""
CONFIG_BOOTDELAY=-1
CONFIG_SYS_PROMPT="pa-mini=> "
CONFIG_SYS_PBSIZE=532
CONFIG_BOARD_EARLY_INIT_F=y
CONFIG_HUSH_PARSER=y
CONFIG_SYS_MAXARGS=32
CONFIG_CMD_BOOTZ=y
CONFIG_CMD_GPIO=y
CONFIG_CMD_MMC=y
CONFIG_CMD_PART=y
CONFIG_PARTITION_UUIDS=y
CONFIG_CMD_DHCP=y
CONFIG_CMD_PING=y
CONFIG_CMD_CACHE=y
CONFIG_CMD_EXT2=y
CONFIG_CMD_EXT4=y
CONFIG_CMD_FAT=y
CONFIG_CMD_FS_GENERIC=y
CONFIG_OF_CONTROL=y
CONFIG_ENV_IS_NOWHERE=y
CONFIG_BOUNCE_BUFFER=y
CONFIG_FSL_USDHC=y
CONFIG_PHYLIB=y
CONFIG_PHY_REALTEK=y
CONFIG_DM_ETH_PHY=y
CONFIG_FEC_MXC=y
CONFIG_MII=y
CONFIG_PINCTRL=y
CONFIG_PINCTRL_IMX6=y
CONFIG_DM_SERIAL=y
CONFIG_MXC_UART=y
CONFIG_IMX_THERMAL=y
```

No copied Micrel PHY, QSPI/NAND, SPI shift register, I2C PMIC, video or USB stack remains. `CONFIG_PHY_REALTEK` is the SR8201F ID-match candidate examined below, not tested networking. Older LAN8720A carriers need their own reviewed driver selection.

Environment changes are RAM-only. `ENV_SIZE` is capacity, not reserved media. `CONFIG_TEXT_BASE` is a link address, not a media offset. There is no SPL; FIT/script support is a separate extension in Chapter 23.

Keep the header's notices and replace its guard/body with:

```
#ifndef __MX6ULL_PA_MINI_CONFIG_H
#define __MX6ULL_PA_MINI_CONFIG_H

#include <asm/arch/imx-regs.h>
#include <linux/sizes.h>
#include "mx6_common.h"

#define PHYS_SDRAM_SIZE SZ_512M
#define PHYS_SDRAM MMDC0_ARB_BASE_ADDR
#define CFG_SYS_SDRAM_BASE PHYS_SDRAM
#define CFG_SYS_INIT_RAM_ADDR IRAM_BASE_ADDR
#define CFG_SYS_INIT_RAM_SIZE IRAM_SIZE
#define CFG_MXC_UART_BASE UART1_BASE
#define CFG_SYS_FSL_ESDHC_ADDR USDHC2_BASE_ADDR
#define CFG_SYS_FSL_USDHC_NUM 2
#define CFG_FEC_ENET_DEV 1

#define CFG_EXTRA_ENV_SETTINGS \
    "fdtfile=imx6ull-pa-mini.dtb\0" \
    "console=ttymxc0\0"

#endif
```

`CFG_FEC_ENET_DEV=1` names the second FEC hardware instance in our clock hook, not Linux's interface number. Modern constants use `CFG_*` where appropriate; the vendor still uses many `CONFIG_SYS_*` macros. Check their consumers instead of globally renaming symbols.

22.5 Step 3, Describe the actual MINI wiring

This DTS becomes U-Boot's **control FDT**, used by driver model. It is not the kernel DTB loaded in Chapter 23. The vendor config instead configures many pads directly in C; its Linux DTB filename does not establish a U-Boot control tree.

Create `arch/arm/dts/imx6ull-pa-mini.dts`. Include only the SoC description, not `imx6ul-14x14-evk.dtsi`:

```
// SPDX-License-Identifier: (GPL-2.0 OR MIT)
/dts-v1/;
#include "imx6ull.dtsi"
```

```
/ {
    model = "PointAtom MINI V2.2 with 512MiB eMMC core";
    compatible = "myorg,imx6ull-pa-mini", "fsl,imx6ull";

    aliases {
        /delete-property/ ethernet1;
        ethernet0 = &fec2;
        mmc0 = &usdhc1;
        mmc1 = &usdhc2;
    };
    chosen {
        stdout-path = "serial0:115200n8";
    };
    memory@80000000 {
        device_type = "memory";
        reg = <0x80000000 0x20000000>;
    };
};

&cpu0 {
    clock-frequency = <528000000>;
    /delete-property/ operating-points;
    /delete-property/ fsl,soc-operating-points;
};

&clks {
    assigned-clocks = <&clks IMX6UL_CLK_PLL3_PFD2>;
    assigned-clock-rates = <320000000>;
};

&uart1 {
    pinctrl-names = "default";
    pinctrl-0 = <&pinctrl_uart1>;
    status = "okay";
};

&usdhc1 {
    pinctrl-names = "default";
    pinctrl-0 = <&pinctrl_usdhc1>;
    bus-width = <4>;
    cd-gpios = <&gpio1 19 GPIO_ACTIVE_LOW>;
    no-1-8-v;
    max-frequency = <25000000>;
    status = "okay";
};

&usdhc2 {
    pinctrl-names = "default";
    pinctrl-0 = <&pinctrl_usdhc2>;
    bus-width = <8>;
    non-removable;
    no-1-8-v;
    max-frequency = <25000000>;
    status = "okay";
};

&fec1 {
    status = "disabled";
};
```

```
&fec2 {
    pinctrl-names = "default";
    pinctrl-0 = <&pinctrl_enet2 &pinctrl_enet2_reset>;
    phy-mode = "rmii";
    phy-handle = <&mini_phy>;
    phy-reset-gpios = <&gpio5 8 GPIO_ACTIVE_LOW>;
    phy-reset-duration = <100>;
    phy-reset-post-delay = <100>;
    status = "okay";

    mdio {
        #address-cells = <1>;
        #size-cells = <0>;
        mini_phy: ethernet-phy@1 {
            compatible = "ethernet-phy-ieee802.3-c22";
            reg = <1>;
        };
    };
};

&iomuxc {
    pinctrl_uart1: uart1grp {
        fsl,pins = <
            MX6UL_PAD_UART1_TX_DATA__UART1_DCE_TX 0x1b0b1
            MX6UL_PAD_UART1_RX_DATA__UART1_DCE_RX 0x1b0b1
        >;
    };
    pinctrl_usdhc1: usdhc1grp {
        fsl,pins = <
            MX6UL_PAD_SD1_CLK__USDHC1_CLK 0x17059
            MX6UL_PAD_SD1_CMD__USDHC1_CMD 0x17059
            MX6UL_PAD_SD1_DATA0__USDHC1_DATA0 0x17059
            MX6UL_PAD_SD1_DATA1__USDHC1_DATA1 0x17059
            MX6UL_PAD_SD1_DATA2__USDHC1_DATA2 0x17059
            MX6UL_PAD_SD1_DATA3__USDHC1_DATA3 0x17059
            MX6UL_PAD_UART1_RTS_B__GPIO1_IO19 0x1b0b0
        >;
    };
    pinctrl_usdhc2: usdhc2grp {
        fsl,pins = <
            MX6UL_PAD_NAND_RE_B__USDHC2_CLK 0x17059
            MX6UL_PAD_NAND_WE_B__USDHC2_CMD 0x17059
            MX6UL_PAD_NAND_DATA00__USDHC2_DATA0 0x17059
            MX6UL_PAD_NAND_DATA01__USDHC2_DATA1 0x17059
            MX6UL_PAD_NAND_DATA02__USDHC2_DATA2 0x17059
            MX6UL_PAD_NAND_DATA03__USDHC2_DATA3 0x17059
            MX6UL_PAD_NAND_DATA04__USDHC2_DATA4 0x17059
            MX6UL_PAD_NAND_DATA05__USDHC2_DATA5 0x17059
            MX6UL_PAD_NAND_DATA06__USDHC2_DATA6 0x17059
            MX6UL_PAD_NAND_DATA07__USDHC2_DATA7 0x17059
        >;
    };
    pinctrl_enet2: enet2grp {
        fsl,pins = <
            MX6UL_PAD_GPIO1_IO07__ENET2_MDC 0x1b0b0
            MX6UL_PAD_GPIO1_IO06__ENET2_MDIO 0x1b0b0
            MX6UL_PAD_ENET2_RX_EN__ENET2_RX_EN 0x1b0b0
            MX6UL_PAD_ENET2_RX_ER__ENET2_RX_ER 0x1b0b0
```

```
            MX6UL_PAD_ENET2_RX_DATA0__ENET2_RDATA00 0x1b0b0
            MX6UL_PAD_ENET2_RX_DATA1__ENET2_RDATA01 0x1b0b0
            MX6UL_PAD_ENET2_TX_EN__ENET2_TX_EN 0x1b0b0
            MX6UL_PAD_ENET2_TX_DATA0__ENET2_TDATA00 0x1b0b0
            MX6UL_PAD_ENET2_TX_DATA1__ENET2_TDATA01 0x1b0b0
            MX6UL_PAD_ENET2_TX_CLK__ENET2_REF_CLK2 0x4001b031
        >;
    };
};

&iomuxc_snvs {
    pinctrl_enet2_reset: enet2resetgrp {
        fsl,pins = <
            MX6ULL_PAD_SNVS_TAMPER8__GPIO5_IO08 0x1b0b0
        >;
    };
};
```

The 25 MHz storage cap is an initial conservative policy, not a measured maximum. We omit high-speed/UHS modes and voltage switching. The core's `SD1_VSELECT` circuit can switch its I/O rail, while the carrier socket remains on 3.3 V; do not request a 1.8 V mode without implementing and validating the complete chain. An already powered rail needs no fabricated switch GPIO.

The SoC include contains generic CPU operating points not established by the core schematic's MCIMX6Y2CVM05AB label. The override removes those tables and labels 528 MHz; it does not program or qualify CPU clocks/voltage. Check actual silicon and clock setup independently. This control tree is not a production Linux CPU-frequency/power description.

The PHY compatible permits actual MDIO discovery rather than forcing a guessed Micrel/SMSC ID. Reset durations are conservative candidate values, not measured timings. These **MAC-node reset properties** are consumed by this release's FEC driver; a property at another location is not automatically equivalent.

Create `arch/arm/dts/imx6ull-pa-mini-u-boot.dtsi`:

```
// SPDX-License-Identifier: GPL-2.0+
&pinctrl_uart1 {
    bootph-all;
};
```

Add this separate line in `arch/arm/dts/Makefile`:

```
dtb-$(CONFIG_MX6ULL) += imx6ull-pa-mini.dtb
```

Optional nodes can follow after their dependencies are reviewed. For example, MINI P4 exposes I2C1 SCL on **pin 43 / `UART4_TXD`** and SDA on **pin 42 / `UART4_RXD`**, not a guessed adjacent row pair; pin 41 is `ENET1_RXER`, pin 44 `UART5_RXD`/I2C2 SDA, pin 45 `UART5_TXD`/I2C2 SCL. These connectors do not require I2C in the minimum bootloader. Review Linux's own tree/bindings separately.

22.6 Step 4, DDR config in DCD

There is no `spl.c` in this target. ROM applies the DCD before entering U-Boot; `dram_init()` reports the configured memory through `imx_ddr_size()`, not a new timing initialization.

We have more evidence than “both boards have DDR”: the supplied core shows the x16 512 MiB DDR3L part/rail, and the pinned vendor publishes its initialization recipe. Compare the **ordered DATA entries** against the copied modern image config:

```
$ cd ~/imx6ull/src/u-boot
$ diff -u \
    <(awk '$1 == "DATA" {sub(/\r$/, "", $4); print $1, $2, $3, $4}' \
      ../uboot-mini-vendor/board/freescale/mx6ull_alientek_emmc/imximage.cfg) \
    <(awk '$1 == "DATA" {sub(/\r$/, "", $4); print $1, $2, $3, $4}' \
      board/myorg/mx6ull_pa_mini/imximage.cfg)
```

This is Bash process substitution, run in the Linux shell. It removes a possible Windows carriage return from the final field, but preserves hexadecimal case and entry order. For the pinned inputs, the DATA sequences match. The candidate therefore carries forward the **document-compared vendor sequence**, not a newly invented calibration table. Keep modern conditional names such as `CONFIG_USE_IMXIMG_PLUGIN` and `CONFIG_IMX_HAB`; do not replace the whole file with the old preprocessor wrapper.

Identical bytes establish provenance, not electrical qualification. Confirm populated DDR, core/PCB revision and rail sequencing, then use the matching NXP DDR tooling/test procedure from Chapter 14. Preserve tool/input versions, calibration output and environmental test scope. No measurements of these rails or DDR margins have been performed for this text.

DCD syntax is `DATA <width-in-bytes> <register-address> <value>`; ordering and initialization commands matter. Review the generated image’s DCD too. A DT memory size cannot repair incorrect initialization. Chapter 20’s PICO SPL study is a different board/architecture, not a MINI substitute.

22.7 Step 5, Board hooks the drivers still need

Why not describe every operation in DT? A binding works only if the selected driver consumes it. This release’s USDHC driver does **not** call the `mmc-pwrseq-emmc` implementation. Adding that node alone would leave the reset pulse unexecuted. We use a small hook for `NAND_ALE` / `GPIO4_IO10` instead, following the vendor route without calling it “power.”

Preserve the copied C file’s licence/copyright notices and replace its implementation with:

```
#include <init.h>
#include <asm/arch/clock.h>
#include <asm/arch/crm_regs.h>
#include <asm/arch/imx-regs.h>
#include <asm/arch/iomux.h>
#include <asm/arch/mx6-pins.h>
#include <asm/arch/sys_proto.h>
#include <asm/global_data.h>
#include <asm/gpio.h>
#include <asm/io.h>
#include <asm/mach-imx/iomux-v3.h>
#include <env.h>
#include <linux/delay.h>
#include <linux/sizes.h>
#include <miiphy.h>

DECLARE_GLOBAL_DATA_PTR;

int dram_init(void)
{
    gd->ram_size = imx_ddr_size();
    return 0;
}

int board_early_init_f(void)
{
    return 0;
}

static int mini_emmc_reset(void)
{
    const unsigned int reset = IMX_GPIO_NR(4, 10);
    int ret;

    imx_iomux_v3_setup_pad(MX6_PAD_NAND_ALE__GPIO4_IO10 |
                           MUX_PAD_CTRL(0x1b0b0));
    ret = gpio_request(reset, "mini-emmc-reset");
    if (ret)
        return ret;
    ret = gpio_direction_output(reset, 0);
    if (ret)
        return ret;
    udelay(500);
    ret = gpio_set_value(reset, 1);
    if (ret)
        return ret;
    mdelay(1);
    return 0;
}

#ifdef CONFIG_FEC_MXC
static int mini_fec_clock(void)
{
    struct iomuxc *regs = (struct iomuxc *)IOMUXC_BASE_ADDR;
    int ret;

    /* Internal ENET2 reference; REF_CLK2 drives the PHY's TXC input. */
    clrsetbits_le32(&regs->gpr[1], IOMUX_GPR1_FEC2_MASK,
```

```
                    IOMUX_GPR1_FEC2_CLOCK_MUX1_SEL_MASK);
    ret = enable_fec_anatop_clock(CFG_FEC_ENET_DEV, ENET_50MHZ);
    if (ret)
        return ret;
    enable_enet_clk(1);
    return 0;
}

int board_phy_config(struct phy_device *phydev)
{
    if (phydev->drv->config)
        return phydev->drv->config(phydev);
    return 0;
}
#endif

int board_init(void)
{
    int ret;

    gd->bd->bi_boot_params = PHYS_SDRAM + 0x100;
    ret = mini_emmc_reset();
    if (ret)
        return ret;
#ifdef CONFIG_FEC_MXC
    return mini_fec_clock();
#else
    return 0;
#endif
}

int board_late_init(void)
{
    return env_set("board_name", "PA-MINI");
}

int checkboard(void)
{
    puts("Board: PointAtom MINI V2.2 migration candidate\n");
    return 0;
}
```

UART1’s pre-relocation pinctrl belongs to the control DT, not a second C pad table. USDHC bus pads also stay in DT; only eMMC reset is owned by this hook. It requests the GPIO, asserts low and releases it, propagating API failures. Assertion follows the vendor’s 500 microseconds, with an initial conservative release delay. Whether the device honors hardware reset also depends on its existing configuration; do not change irreversible eMMC reset-enable fields as a lab shortcut.

ENET2 retains the internal 50 MHz reference path needed by the supplied SR8201F input strap. The PHY’s 25 MHz crystal is a separate circuit. PHY reset belongs to the FEC driver’s DT path, not this C hook.

Check the driver by ID. The CoreChips SR8201F-VB datasheet, register tables 13-14, gives ID words 0x001c and 0xc816, combined **0x001cc816**. This matches the v2026.04 RTL8201F entry, UID 0x1cc816/mask 0xffffff. It is a **datasheet/source match**, not an observed board ID or proof of complete silicon compatibility. Confirm actual suffix/revision and MDIO ID before qualifying the link.

Leave `CONFIG_RTL8201F_PHY_S700_RMII_TIMINGS` disabled: it chooses another platform's adjustments. The normal driver uses generic autonegotiation/startup. Real reset recovery, clock direction and packet transfers remain hardware tests.

This board hook removes the inherited unconditional PHY write **`0x1f = 0x8190`**. On SR8201F that address is a page selector, not a universal board register. It returns the selected driver's config status instead of discarding it. Do not transplant the guide's LAN8720A global-reset workaround into modern generic PHY code.

The old vendor header enables **`CONFIG_PHY_SMSC` only**. That supports its older intended LAN8720A path, not a known-working SR8201F network claim. Generic fallback is not qualification. Chapter 19's initial example build explicitly disables video and networking, so its inherited display and PHY hooks are not executed. It also disables USB commands to avoid the old USB/network dependency. Any later re-enablement needs carrier-specific review; this modern candidate replaces the inherited PHY write rather than copying it.

22.8 Step 6, Build and inspect the candidate

First build the modern migration in a fresh output directory, without cleaning either source workspace:

```
$ . ~/imx6ull/scripts/env.sh
$ cd ~/imx6ull/src/u-boot
$ command -v arm-none-linux-gnueabihf-gcc
$ make O="$HOME/imx6ull/build/u-boot-mini-2026.04" \
    ARCH=arm CROSS_COMPILE=arm-none-linux-gnueabihf- mx6ull_pa_mini_defconfig
$ make O="$HOME/imx6ull/build/u-boot-mini-2026.04" \
    ARCH=arm CROSS_COMPILE=arm-none-linux-gnueabihf- -j"$(nproc)"
$ grep -E 'CONFIG_(TARGET_MX6ULL_PA_MINI|IMX_CONFIG|DEFAULT_DEVICE_TREE|TEXT_BASE|
ENV_IS_NOWHERE|PHY_REALTEK|SPL)=' \
    ~/imx6ull/build/u-boot-mini-2026.04/.config
$ ls -l ~/imx6ull/build/u-boot-mini-2026.04/u-boot-dtb.imx
$ ~/imx6ull/build/u-boot-mini-2026.04/tools/mkimage -l \
    ~/imx6ull/build/u-boot-mini-2026.04/u-boot-dtb.imx
```

Keep Chapter 3's compiler directory name, `arm-gnu-toolchain-13.2.Rel1-x86_64-arm-none-linux-gnueabihf`, and prefix `arm-none-linux-gnueabihf-`. No `.bashrc`/global path edits are needed. Old-source compatibility patches from Chapter 19 belong to the vendor build, not automatically to this release.

Inspect final config, generated `arch/arm/dts/imx6ull-pa-mini.dtb`, linked board object and image DCD/header. Grep above selects assignments; unset SPL may appear as a comment or be absent. Check actual artifacts too. After menu changes, run `make O=... savedefconfig` and manually compare its output before updating your source defconfig.

The modern product is `u-boot-dtb.imx`, distinct from the vendor build's `u-boot.imx`. For this modern release's `tools/imximage.c`, the non-plugin v2 **Load Address** label prints the IVT's BootData pointer, not `BootData.start`. See the upstream inspection in Chapter 24A Section 21; do not assume the vendor tool displays the same fields. Inspect actual fields to calculate the ROM range.

Flashing is a separate decision. The modern EVK documentation's 1 KiB SD offset is an EVK-route fact, not a blanket eMMC-area instruction. Compare the modern image's actual format/length with Chapter 19's MINI vendor layout evidence. There is no SPL-at-1-KiB / full-U-Boot-at-69-KiB recipe here.

Before a later physical write, establish all of these:

- Matching core/carrier population, DCD/power/pad evidence and qualified boot/memory map.
- Actual medium/eMMC area, ROM offset, image length, partition map and reserved environment regions, without overlap.
- Host device identity, mounts and backup; U-Boot MMC indices are not Linux host device names.
- A qualified board-specific recovery image/path without fuse changes.
- Reviewed write/read-back procedure, not a copied block count.

No raw write command or target execution is supplied here. A successful build completes the source/API check, not the hardware gate.

22.9 Verify per-peripheral

After the hardware/recovery gate, interrupt autoboot and record real identity. Start with UART/storage; Ethernet is not a prerequisite for reaching the first prompt.

Qualified-board check	Evidence to record	Limit
`version`, `bdinfo`, `printenv`, `help`	Build, reported memory/relocation, active defaults/commands	Not DDR margin or proof that saved settings fit the new build

Qualified-board check	Evidence to record	Limit
`mmc list`, verified `mmc dev / mmc info`	SD/eMMC routing and enumerated properties	Not image-layout/write safety
Read-only listing/load into reserved RAM	Prepared filesystem files and load status	Wrong addresses/sizes can overwrite live RAM
ENET2 MDIO ID inspection using an actually available command	Actual address 1/ID, reset/clock evidence and driver	Management-bus response is not packet communication
Isolated `ping`, then controlled TFTP checks	Actual link and transfer results	Not Linux PHY readiness, NFS-root or reliability qualification

Normal vendor mapping is MMC 0 for USDHC1/SD and MMC 1 for USDHC2/eMMC. Modern aliases preserve that intent. Still record enumeration: vendor fuse-based mapping and Linux aliases are additional layers, not permission to guess `/dev/mmcblkN`.

Compare the real PHY ID with Section 22.7, inspect RMII reference/reset if needed, and preserve failure logs. Stop if the observed device differs. A banner saying “FEC” is not a successful network test.

Do not run `mtest` over a guessed live-DDR range. It can overwrite U-Boot, stack/heap, control FDT or images. DDR qualification uses a dedicated setup with known ownership and a documented test plan.

22.10 MAINTAINERS file

Replace copied EVK ownership. Before submission, fill in actual maintainer details:

```
MX6ULL_PA_MINI BOARD
M: Your Name <you@example.com>
S: Maintained
F: board/myorg/mx6ull_pa_mini/
F: include/configs/mx6ull_pa_mini.h
F: configs/mx6ull_pa_mini_defconfig
F: arch/arm/dts/imx6ull-pa-mini.dts
F: arch/arm/dts/imx6ull-pa-mini-u-boot.dtsi
```

This routes review, not certification. Keep hardware revisions, DCD provenance and actual tests beside the port.

22.11 Key Notes

- A board port connects build selection, control DT, image configuration and board hooks. A matching SoC target alone does not describe the MINI carrier or populated core.
- Ordered vendor DCD writes provide traceable DDR initialization input. The modern DT’s memory size describes a bank but cannot initialize it or establish electrical margins.
- DT properties work only through their consumers. In this release, eMMC reset needs the explicit board hook because the USDHC driver does not invoke `mmc-pwrseq-emmc`.
- The single MINI RJ45 uses ENET2. PHY address, PHY ID, 25 MHz crystal and 50 MHz RMII reference describe different interfaces and cannot substitute for one another.
- Optional connectors can share pads and controllers. TF/SDIO sharing and the distinct KEY0, Wi-Fi and accessory nets constrain what may be enabled together.
- The modern result is a host-built `u-boot-dtb.imx` candidate with RAM-only environment and no SPL. Persistent replacement still requires identified media, a complete range map and a qualified recovery path.

22.12 Pitfalls

- **ALPHA prose used as MINI wiring.** Follow supplied schematics and the populated revision.
- **Vendor source treated as universal support.** Shared core support does not establish the changed carrier PHY.
- **DCD matching confused with validation.** Published-sequence provenance is not measured DDR margin.
- **Eight-bit eMMC treated as four-bit SD.** Include DATA4-7 and the actual reset; do not turn reset into rail control.
- **Unused DT power-sequence node.** Check driver consumers; this USDHC implementation needs the explicit hook.
- **PHY names/addresses used as driver matches.** Compare real ID/mask, then qualify reset/clock/data behavior.
- **Wrong address space.** Link, DCD registers, RAM buffers and media offsets have different purposes.
- **Saving inherited policy.** Modern `ENV_IS_NOWHERE` and the vendor’s MMC backend are different; neither permits unreviewed persistent writes.

22.13 Going deeper

Follow the v2026.04 DT control guide, USDHC driver, FEC driver and PHY driver alongside the schematic. Find each property’s consumer before relying on it.

Vendor Chapter 33 explains the original pad/storage/LAN8720A port, but its old symbols, global-PHY workaround and unconditional boot strings are not modern patches. Keep the supplied schematics, matched device datasheets and i.MX6ULL reference manual from Parts I-II beside both trees.

> Next chapter: **Chapter 23: `bootcmd`, `bootargs`, FIT images.** Inspect the vendor storage policy, replace unsafe load chains with explicit gates, and study FIT separately in the modern workspace.

Chapter 23: `bootcmd`, `bootargs`, FIT images

A U-Boot prompt tells you that the bootloader reached its command loop. What must happen next? It must place a kernel and the right hardware description in usable RAM, choose where Linux will find its files, then transfer control. These are separate operations. A successful file transfer does not prove the kernel can mount root, and a correct command line does not repair a wrong DTB.

On an MCU, the linker script and startup code often fix those decisions at build time. Here, part of the policy is text in U-Boot's **environment**, a set of named string values. We will follow that text into Linux, then package the kernel and DTB into a FIT, or Flattened Image Tree.

Start with Chapter 19's **vendor MINI build**, source commit `edb7ca5ac4be2d978be60a2a12c61e0b6d1f7feb`, config `mx6ull_alientek_emmc_defconfig`, in `~/imx6ull/src/uboot-mini-vendor`. The FAT/eMMC helper below follows that command set. The subsequent SD/PARTUUID, control-FDT and FIT examples use Chapter 22's separate **upstream v2026.04 migration**, commit `88dc2788777babfd6322fa655df549a019aa1e69`, in `~/imx6ull/src/u-boot`. Do not assume modern features exist in the vendor build.

Vendor policy names Linux's `imx6ull-alientek-emmc.dtb`; modern examples use local `imx6ull-pa-mini.dtb`. Both must be **kernel DTBs from matched Part IV builds**, checked for the actual MINI revision. Neither permits an EVK DTB or Chapter 22's U-Boot control DTB as the kernel tree.

> **Example boundary:** These builds provide source/API evidence, not board validation. Target examples require qualified DDR, bootloader and memory map. Host packaging needs no board. No media writes, persistent settings, key enrollment or fuse programming are needed.

23.1 `bootcmd`, U-Boot's autoboot

`bootcmd` is a string evaluated by U-Boot's command interpreter during autoboot. `bootdelay` is the runtime delay setting, with `CONFIG_BOOTDELAY` as its build-time default; negative values have special meanings. The Chapter 19 example patches and Chapter 22 migration disable autoboot. Typing `run emmcboot` or `run sdboot` still executes a helper manually.

Before writing a helper, answer three questions: which storage device/partition has the files, where may they fit in RAM, and what should happen if a load fails? The following addresses are a **worked layout**, not a universal i.MX6ULL memory map:

Buffer	Example start	Planned file window
`kernel_addr_r`	`0x82000000`	Less than 16 MiB before the next buffer
`fdt_addr_r`	`0x83000000`	Less than 1 MiB, with room for later DT expansion/relocation
`scriptaddr`	`0x84000000`	A separate small script buffer
`fit_addr_r`	`0x85000000`	Less than 16 MiB for the complete FIT

Use these numbers only after checking actual DDR banks and reserved ranges, U-Boot relocation/stack/heap/control FDT, kernel decompression workspace and the uncompressed kernel destination. `bdinfo` helps inspect U-Boot's layout but is not a complete kernel-memory planner. Host file sizes must fit their planned windows **before** loading; a successful transfer can already have overwritten another object if the file was too large.

Start with the vendor's eMMC/FAT route

Normal vendor mapping is MMC 0 for USDHC1/SD and MMC 1 for USDHC2/eMMC. Published policy uses eMMC partition 1 for boot files and `/dev/mmcblk1p2` for Linux root. Confirm `mmc list`, contents and kernel enumeration: U-Boot indices do not guarantee Linux names. The vendor enables older `CONFIG_SYS_HUSH_PARSER`, FAT loading and `bootz`; Chapter 19's FAT compatibility fix belongs to that old build.

After verifying the RAM plan above, define a temporary helper requiring both files and argument construction:

```
=> setenv kernel_addr_r 0x82000000
=> setenv fdt_addr_r 0x83000000
=> setenv fdtfile imx6ull-alientek-emmc.dtb
=> setenv emmcargs 'setenv bootargs console=ttymxc0,115200 root=/dev/mmcblk1p2
rootfstype=ext4 ro rootwait'
=> setenv emmcboot 'if mmc dev 1 && mmc rescan && fatload mmc 1:1 ${kernel_addr_r} zImage
&& fatload mmc 1:1 ${fdt_addr_r} ${fdtfile} && run emmcargs; then bootz ${kernel_addr_r}
- ${fdt_addr_r}; else echo eMMC load or arguments failed; false; fi'
=> run emmcboot
```

The root path is the documented vendor assumption, not universal advice. Stop if it is not the intended device/partition. Inspect `help fatload`, `help bootz` and active environment. The helper neither writes media nor saves settings. Chapter 19's example patches use RAM environment/no autoboot; the unmodified vendor header selects a persistent MMC backend.

Modern SD route with PARTUUID

The rest of this section uses the modern migration. Chapter 22 enables **`CONFIG_CMD_PART=y`** and **`CONFIG_PARTITION_UUIDS=y`** for `part uuid`. Keep hush, generic filesystem/MMC support and `CONFIG_CMD_BOOTZ=y`. Inspect final config and `help part` before relying on it; do not infer support from the vendor command set.

For the next example only, assume verified U-Boot MMC device `0`, boot-files partition `1`, and an ext4 root partition `2`. These are not guaranteed to be the same indices Linux later uses. The board's routed console must match `ttymxc0` at 115200 baud. Enter the commands at U-Boot, without copying the `=>` prompt:

```
=> setenv kernel_addr_r 0x82000000
=> setenv fdt_addr_r 0x83000000
=> setenv fdtfile imx6ull-pa-mini.dtb
=> setenv sdargs 'setenv bootargs console=ttymxc0,115200 root=PARTUUID=${rootuuid}
rootfstype=ext4 ro rootwait'
=> setenv sdboot 'if mmc dev 0 && mmc rescan && part uuid mmc 0:2 rootuuid && load mmc
0:1 ${kernel_addr_r} zImage && load mmc 0:1 ${fdt_addr_r} ${fdtfile} && run sdargs; then
bootz ${kernel_addr_r} - ${fdt_addr_r}; else echo SD load or arguments failed - staying
in U-Boot; false; fi'
=> run sdboot
```

`part uuid` obtains the root partition's identifier instead of guessing `/dev/mmcblk0p2`. It requires the partition/UUID command support in your build. Confirm that the identified partition is actually the intended root filesystem. `load` reads a filesystem file; it is not a raw-sector loader. Its available filesystems depend on the config.

The example uses the **hush parser** (`CONFIG_HUSH_PARSER=y`). Single quotes preserve `${...}` while defining a helper, so values expand when `run` executes it. `&&` continues only after success. The `else` branch deliberately returns failure with `false`, making it useful inside another helper too.

Argument construction is a prerequisite as well: `run sdargs` must succeed before `bootz`. Writing it as `run sdargs; bootz ...` would allow a failed environment update to leave an old command line in use. The FIT example uses the same gate before `bootm`.

Why not `load ...; load ...; bootz ...`? A semicolon continues after a failed command. RAM may still contain yesterday's kernel or DTB, or a partial new transfer. A failed load must block the boot command, even if the address looks plausible. The v2026.04 load documentation describes its status and `filesize` result.

A successful `bootz` normally does not return to the prompt. If it fails before handoff, U-Boot remains available for inspection. Keep helpers temporary while debugging. Only after the complete chain is understood would `setenv bootcmd 'run sdboot'` change this session's autoboot policy; it does not make that policy persistent.

Read inherited policy before replacing it

The pinned vendor header, not its small defconfig alone, defines default policy. Trace `findfdt`, `loadimage`, `loadfdt`, `mmcboot` and `netboot`. Its **`fdt_file=imx6ull-alientek-emmc.dtb`** is a Linux filename, not a carrier/PHY detector. Storage helpers use FAT and `mmcdev` starts from the environment device, normally 1.

Published policy repeats `mmc dev`, permits DT-less fallback through `boot_fdt=try`, and does not fully gate the network kernel load. These are reasons for our helper, not patterns to retain. Chapter 19's example patches disable automatic use and networking; inspect the binary rather than assuming patches were applied elsewhere.

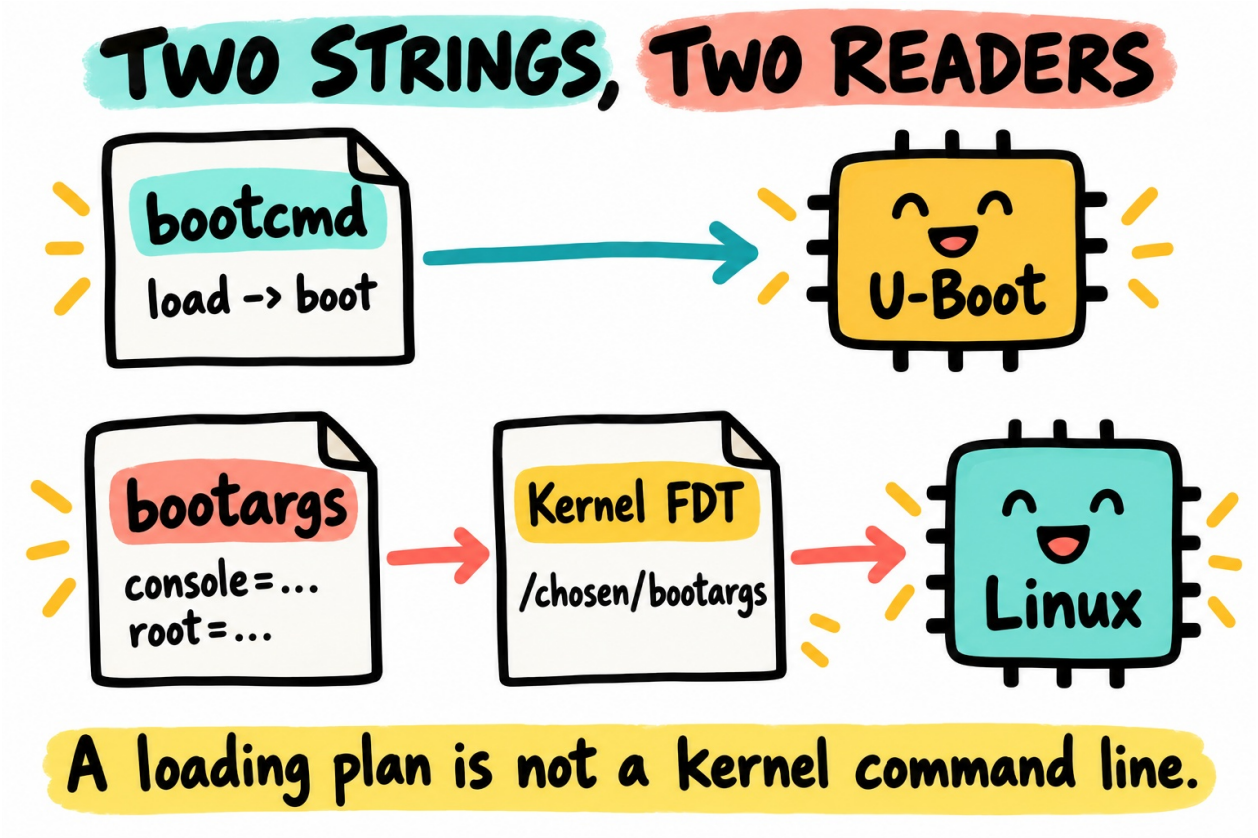
For comparison only, `configs/mx6ull_14x14_evk_defconfig` in v2026.04 supplies this EVK command, reformatted for reading. It is not the modern MINI default; Chapter 22 replaces it with an empty command:

```
run findfdt;
mmc dev ${mmcdev};
if mmc rescan; then
    if run loadbootscript; then
        run bootscript;
    else
        if run loadimage; then
            run mmcboot;
        else
            run netboot;
        fi;
    fi;
else
    run netboot;
fi
```

EVK helpers live in `include/configs/mx6ullevk.h`. Its `findfdt` uses **`fdt_file`**, not our **`fdtfile`**, and chooses EVK/ULZ-EVK filenames from board-name/revision strings. Neither these nor similarly named vendor helpers detect the safe kernel tree for a changed carrier.

A `boot.scr` is a packaged command script loaded by `loadbootscript` and executed by `source`; it is not plain text that U-Boot silently reads on every board. A stored environment from an old image can also override compiled defaults. On qualified hardware, inspect `printenv bootcmd` and each referenced variable. On the host, trace them in the versioned source without claiming that your board has those defaults.

23.2 bootargs, the kernel command line



Two strings have different readers. U-Boot executes the loading plan, then prepares the kernel tree and command line for Linux. This kernel FDT is not U-Boot's control tree.

`bootcmd` belongs to U-Boot. `bootargs` supplies a candidate Linux command line. In the normal DT boot path, U-Boot places it in **`/chosen/bootargs`**, a string property of the **kernel** FDT. Linux interprets the words, subject to its own build-time command-line policy. A board hook or forced/extended `CONFIG_CMDLINE` policy can change the final result. Once Linux is running, `/proc/cmdline` is the evidence for what it received.

For an ext4 root partition, our `sdargs` helper constructs:

```
console=ttymxc0,115200 root=PARTUUID=<actual-root-partition-id> rootfstype=ext4 ro
rootwait
```

The bracketed identifier above is explanatory, not literal input. Each field answers a different question:

Token	Question it answers	Prerequisite or limit
`console=ttymxc0,115200`	Where should the normal serial console appear?	Matched DT UART route and built-in i.MX console driver; not required for every possible Linux system

Token	Question it answers	Prerequisite or limit
`root=PARTUUID=...`	Which partition is mounted as `/`?	Correct partition identifier and built-in storage/partition support, unless an initramfs supplies it
`rootfstype=ext4`	Which filesystem is expected?	Does not add the ext4 driver; it must be available before root mount
`ro` / `rw`	Initial read-only or read-write mount?	Userspace may later remount; `ro` is useful for inspection
`rootwait`	Wait for discovery of the block root device?	Does not repair a wrong identifier, unsupported filesystem or NFS timeout

Useful diagnostic tokens are not universal fixes:

Token	Meaning and boundary
`init=/bin/sh`	Use that executable after mounting the real root. It must exist with its interpreter/libraries. It cannot recover an unmountable root.
`rdinit=/init`	Select an executable in the initramfs. Its presence changes the early-userspace path.
`single`	A request to userspace init; the selected init implementation decides what it means.
`quiet` / `loglevel=8`	Reduce console messages or permit debug-level messages. `ignore_loglevel` can help diagnostics.
`panic=10`	Request a reboot after a panic delay; omit while you need to read a failure.
`earlycon=ec_imx6q,0x02020000`	An early polled i.MX UART console, only if that UART is already initialized and matches the board route and kernel support.

Token	Meaning and boundary
`ip=dhcp`	Ask the kernel's built-in IP autoconfiguration to use DHCP, if enabled and a server exists. It is not U-Boot DHCP.
`root=/dev/nfs nfsroot=...`	Ask Linux to mount network root. Ethernet, IP and NFS prerequisites are developed in Chapter 24.

The Linux parameter reference documents these kernel-side options. Check it against the kernel release/config selected in Part IV; pinning U-Boot does not also pin Linux.

Do not append a debug token and assume it survives `run bootcmd`. A helper such as `sdargs` can overwrite `bootargs` immediately before boot. Change the helper that constructs it, or perform a controlled manual boot without calling that helper. Keep the previous value in your notes and restore it after the experiment. No `saveenv` is needed for one-shot testing.

Where bootargs ends up in the DT

In v2026.04, the normal path reaches `fdt_chosen()` in `boot/fdt_support.c`, via OS FDT preparation in `boot/image-fdt.c`. The weak `board_fdt_chosen_bootargs()` defaults to returning the environment's `bootargs`. The vendor uses older `common/fdt_support.c` and does not configure this board with the modern control-FDT setup. Trace each release's callers rather than assuming identical paths. The resulting kernel property looks like this:

```
chosen {
    bootargs = "console=ttymxc0,115200 root=PARTUUID=... rootfstype=ext4 ro rootwait";
};
```

Do not edit the control FDT to change the kernel command line. `fdt addr -c` queries U-Boot's control tree. `fdt addr ${fdt_addr_r}` selects a loaded **working FDT**, which can become the kernel tree. The `fdt` command is not itself a file loader, and boot preparation may later relocate/fix up the tree. The v2026.04 FDT command documentation explains both roles.

After a successful, size-checked kernel-DTB load, a gated inspection is:

```
=> if load mmc 0:1 ${fdt_addr_r} ${fdtfile}; then fdt addr ${fdt_addr_r}; fdt print /
model; fdt print /chosen; else echo DTB load failed; false; fi
```

This shows the loaded tree **before** boot-time fixups, so a missing or old `bootargs` property at this stage is not proof that Linux will receive no command line. Do not change the control address or expand a tree into unreserved live RAM as a diagnostic shortcut.

23.3 `bootm`, `bootz`, `booti`, what's the difference

Choose by image format and enabled architecture support, not by filename extension alone:

Command	Input	This board path
`bootz`	A Linux `zImage`, with optional raw initramfs and FDT	The ordinary 32-bit ARM route here; needs `CONFIG_CMD_BOOTZ=y`
`bootm`	A FIT or supported legacy uImage	FIT needs `CONFIG_FIT=y`; a legacy wrapper needs `CONFIG_LEGACY_IMAGE_FORMAT=y`
`booti`	Supported Linux `Image` format, including configured decompression support	Used on architectures such as arm64/RISC-V, not an alternative for this ARMv7-A zImage

For a raw initramfs, `bootz` needs **address:size**. For example, after all three loads succeed and their regions have been reserved:

```
bootz ${kernel_addr_r} ${ramdisk_addr_r}:${ramdisk_size} ${fdt_addr_r}
```

Save `ramdisk_size` from `${filesize}` **immediately after the successful initramfs load**; a later DTB load replaces `filesize`. Use `-` in that position if no initramfs is supplied. These details are in the v2026.04 bootz reference. A valid gzip/cpio initramfs also requires the kernel's initrd/decompressor support and an executable `/init` with its dependencies.

A FIT uses its own named subimages/configurations. It is not a zImage renamed to `.itb`, and `bootm` does not invoke `bootz` as a shell helper. For our FIT below, U-Boot starts an ARM zImage payload that performs its own decompression; `compression = "none"` means no additional outer compression for U-Boot to undo.

23.4 Boot scripts, a slightly nicer layer

Suppose you want the same reviewed commands on two lab cards without retyping them. A script can do that, but only if the boot policy explicitly loads and executes it. A script is executable policy, not a harmless configuration file.

Open `~/imx6ull/boot/boot.cmd` in an editor. Use the approved address map and storage assumptions from Section 23.1. The following complete script gates both loads and obtains the root partition ID:

```
if mmc dev 0 && mmc rescan && part uuid mmc 0:2 rootuuid && load mmc 0:1 ${kernel_addr_r}
zImage && load mmc 0:1 ${fdt_addr_r} ${fdtfile} && setenv bootargs console=ttymxc0,115200
root=PARTUUID=${rootuuid} rootfstype=ext4 ro rootwait; then
    bootz ${kernel_addr_r} - ${fdt_addr_r}
else
    echo Storage, image load or argument construction failed
fi
exit 1
```

The final **`exit 1`** is outside the conditional. It reports failure if any prerequisite fails or if `bootz` returns instead of handing control to Linux. A successful kernel handoff never reaches it. Do not rely on a trailing `false`, or an exit buried in a nested conditional, to supply the final status of this multiline old-hush script. The v2026.04 exit reference describes leaving the innermost script. Single-line prompt helpers above still use `false` for their failure branch.

Use the **v2026.04 build's** host tool, not an unrelated system `mkimage`:

```
$ . ~/imx6ull/scripts/env.sh
$ cd ~/imx6ull/boot
$ ~/imx6ull/build/u-boot-mini-2026.04/tools/mkimage \
    -A arm -O linux -T script -C none -n 'MINI SD lab boot' -d boot.cmd boot.scr
$ ~/imx6ull/build/u-boot-mini-2026.04/tools/mkimage -l boot.scr
```

This creates a legacy script image. On a qualified board, with the file already staged by an approved procedure and `scriptaddr` reserved separately:

```
=> setenv scriptaddr 0x84000000
=> if load mmc 0:1 ${scriptaddr} boot.scr; then source ${scriptaddr}; else echo Script
load failed; false; fi
```

`CONFIG_CMD_SOURCE=y` and legacy image support are required for this example format. The script must already have `kernel_addr_r`, `fdt_addr_r` and `fdtfile` set; inspect them before sourcing it. A plain `boot.cmd` cannot be executed by this legacy `source` path. The v2026.04 source reference also describes FIT scripts. Neither a legacy CRC nor an unsigned script gives authentication; do not allow it to bypass a product's verified-boot policy.

23.5 FIT, Flattened Image Tree

With separate files, how do you record that a particular kernel belongs with a particular DTB? FIT gives them names inside one container and lets a **configuration** choose the combination. It can also carry a ramdisk or other firmware, but those are optional. One bundle simplifies release bookkeeping; it does not make the underlying storage update atomic.

First make the build support explicit. In your own v2026.04 build, use `make O=... menuconfig`, inspect/save the resulting `.config`, and ensure `CONFIG_FIT=y`, `CONFIG_SHA256=y` and `CONFIG_CMD_BOOTM=y`. Keep `CONFIG_CMD_BOOTZ`, hush, filesystem and DT command support for the preceding examples. Do not assume vendor or EVK configs enable FIT just because `bootm` exists. The unsigned example below does not need `CONFIG_FIT_SIGNATURE` or keys.

This FIT example belongs to the **modern build**, not the initial vendor image. Host tool capabilities alone do not establish target FIT support.

A FIT image source file (.its)

In a new staging directory under `~/imx6ull/boot`, place matched kernel `zImage` and **kernel** `imx6ull-pa-mini.dtb` artifacts. Record their build/config/board revision and sizes. The following `boot.its` is a complete **kernel + DTB, hash-only** example; it does not depend on a rootfs archive from a chapter you have not reached yet:

```
/dts-v1/;

/ {
    description = "MINI lab kernel and kernel DTB";
    #address-cells = <1>;

    images {
        kernel-1 {
            description = "ARM Linux zImage";
            data = /incbin/("zImage");
            type = "kernel";
            arch = "arm";
            os = "linux";
            compression = "none";
            load = <0x82000000>;
            entry = <0x82000000>;
            hash-1 {
                algo = "sha256";
            };
        };

        fdt-1 {
            description = "Matched MINI kernel DTB";
            data = /incbin/("imx6ull-pa-mini.dtb");
            type = "flat_dt";
            arch = "arm";
            compression = "none";
            load = <0x83000000>;
            hash-1 {
                algo = "sha256";
            };
        };
    };

    configurations {
        default = "conf-mini";
        conf-mini {
            description = "Qualified board revision recorded with artifacts";
```

```
            kernel = "kernel-1";
            fdt = "fdt-1";
        };
    };
};
```

The `load`/`entry` values must match the approved RAM/decompression plan, not just the source filenames. Build and list the container from that staging directory:

```
$ . ~/imx6ull/scripts/env.sh
$ ~/imx6ull/build/u-boot-mini-2026.04/tools/mkimage -f boot.its boot.itb
$ ~/imx6ull/build/u-boot-mini-2026.04/tools/mkimage -l boot.itb
$ stat -c '%n %s bytes' zImage imx6ull-pa-mini.dtb boot.itb
$ sha256sum zImage imx6ull-pa-mini.dtb boot.itb
```

Listing the FIT checks packaging/metadata, not whether either payload can run. Load the complete FIT into its **separate** approved buffer. Set the root/console command line for the intended rootfs before boot, for example by running `sdargs` after obtaining `rootuuid` for the same verified root partition:

```
=> setenv fit_addr_r 0x85000000
=> if mmc dev 0 && mmc rescan && part uuid mmc 0:2 rootuuid && load mmc 0:1 ${fit_addr_r}
boot.itb && run sdargs; then bootm ${fit_addr_r}#conf-mini; else echo FIT load, root
identification or arguments failed; false; fi
```

The `#conf-mini` suffix chooses a FIT configuration; it is **not** a signing option. Keep it adjacent to the address. Do not load the container at the kernel's `load` address: extracting the kernel there could overwrite the DTB and other still-needed data inside the FIT. The v2026.04 bootm reference documents configuration selection.

What do the SHA-256 nodes buy you? With verification enabled, U-Boot can detect payload corruption against the stored hashes. An attacker who can replace the FIT can replace both payload and hash. **Integrity hashes are not authentication.** Authentication requires signature support, trusted keys held outside the untrusted FIT, required verification and a policy that prevents unsigned alternate boot paths. NXP HAB authenticating the ROM-loaded bootloader and U-Boot authenticating a FIT are distinct links in a trust chain. See the v2026.04 signature documentation. this example creates no keys, provisions no trust anchor and changes no fuses.

An optional initramfs belongs in a `ramdisk` subimage and the selected configuration must reference it. For a `rootfs.cpio.gz` payload, do not blindly tell U-Boot to decompress it before Linux receives it: use the outer-compression convention appropriate to the boot path, commonly `compression = "none"`, with Linux doing the gzip unpacking. Audit `CONFIG_BLK_DEV_INITRD`, gzip support, `/init`, libraries, size and RAM reservations first. A configuration with an executable initramfs `/init` is not the same boot experiment as direct NFS-root.

Why this matters in Chapter 24E

Several configurations can reference different kernel DTBs. Chapter 24E develops board-identity selection. Choosing a configuration is a hardware decision: an EEPROM/strap value and a DTB must describe the same qualified revision. A valid FIT structure cannot prove that match. Keep this chapter to one reviewed configuration before adding variants.

23.6 Practical boot-command idioms

Quick recovery boot (sh as init)

If root mounts but normal init fails, temporarily change `sdargs` to include `init=/bin/sh`, then use the same gated `sdboot`. The shell must exist and be executable, including its dynamic loader/libraries. Prefer a read-only root while inspecting.

That shell is PID 1, not an ordinary login session. Exiting it can panic the kernel; shutdown/reboot tools may assume a normal init service that is absent. Use the rootfs's documented recovery procedure and synchronize any intentional writes before a controlled reset. Do not use `init=/bin/sh` as a cure for `root=/dev/nonsense`: root mounting happens first.

NFS-root development loop

Keep the loader and root choice separate. A TFTP loader can hand the same kernel a local-root or NFS-root command line. Chapter 24 defines a gated `devel_boot` and builds `bootargs` immediately before it. U-Boot `ipaddr`/`serverip` configure **U-Boot**, while Linux needs its own `ip=`/`nfsroot=` and built-in drivers. Modules on an NFS root cannot provide the driver needed to reach that root in the first place.

Boot from a USB stick

USB mass storage is a **host-mode** path, unlike ROM USB serial download. The vendor header enables host/storage code, but that is not board validation. Chapter 22's modern minimum deliberately excludes USB and has no ready-to-use host command/controller. Enable and validate host/storage support, connector role and power first. With the same approved RAM map and a prepared stick whose device/partition mapping you have checked:

```
=> if usb start && load usb 0:1 ${kernel_addr_r} zImage && load usb 0:1 ${fdt_addr_r} $
{fdtfile}; then bootz ${kernel_addr_r} - ${fdt_addr_r}; else echo USB load failed; false;
fi
```

Set `bootargs` beforehand for the intended root. Reading the kernel from USB does not imply Linux's root is also on USB. `usb start` initializes hardware; it is not a host-only syntax test.

Boot the same kernel with two different DTBs

Only compare DTBs appropriate for the **same physical board/revision**, such as a reviewed peripheral-description change. After reset, reload both artifacts through a gated helper and select the intended `fdtfile`. Do not substitute arbitrary MINI revisions as a safe experiment: a wrong DTB can drive incorrect regulators, pads or clocks even if Linux accepts the blob’s syntax. Record source revisions so you can attribute the observed difference.

23.7 Diagnose the last completed boot stage

What was the last operation that actually completed? A mount failure is a different problem from a failed image load. Use that evidence before changing another command-line token:

Observation	Checks before another boot	Why a cmdline-only fix may fail
Loader reports failure or wrong image format	File path, command support, image metadata, approved sizes and load status	No kernel should be started after this failure
No text after handoff	Correct kernel DTB, console UART/driver, early-console support, DDR and entry/decompression map	A wrong pad/clock or corrupt image can look like a console typo
Root device cannot be found	Actual partition ID, built-in MMC/storage/partition drivers and discovery messages	`rootwait` cannot create a missing device
Root device found but mount fails	Filesystem type/support and image contents	`rootfstype=ext4` selects a driver; it does not build one
Root mounts but init fails	Init path, execute permissions, interpreter, architecture/ABI and libraries	`init=/bin/sh` still requires a usable shell/runtime
NFS-root waits or times out	Kernel Ethernet/IP/NFS support, link, server address/export and RPC/firewall path	A U-Boot ping tests a different network stack

Observation	Checks before another boot	Why a cmdline-only fix may fail
FIT hash verification fails	Restage the reviewed artifacts/ container and keep the failure evidence	Do not disable verification to make it boot

23.8 Key Notes

- `bootcmd` is U-Boot execution policy, while `bootargs` becomes a candidate Linux command line through the OS FDT. Helpers and kernel configuration can alter what `/proc/cmdline` ultimately reports.
- Each load and argument-construction step must succeed before handoff. Semicolons alone can continue after failure and reuse stale payloads or a stale command line.
- RAM ownership includes staging buffers, kernel decompression, FDT growth and U-Boot's runtime reservations. File-size checking belongs before the transfer, not after possible overlap.
- `bootz`, `bootm` and `booti` select supported image/architecture paths. A filename extension or metadata string does not convert a payload's format or architecture.
- FIT configurations bind named payloads into a selected set. Hash-only verification detects corruption but does not authenticate the producer or enforce an allowed board combination.
- Root discovery, filesystem mounting and starting init are separate Linux dependencies. An `init=/bin/sh` choice cannot repair an unreachable root and still requires a usable shell runtime.

23.9 Pitfalls

- **Expansion at the wrong time.** Single-quote helper definitions. A plain `setenv` assignment is not a recursive environment evaluator; `run` executes the stored command string.
- **A later helper overwrites diagnostics.** Inspect the command that actually constructs `bootargs`, not just its earlier printed value.
- **Stale RAM and stale `filesize`.** Gate every load; preserve an initramfs size immediately. A failed load is not permission to reuse old data.

- **Confusing the two FDTs.** The control tree describes U-Boot's hardware/drivers and can hold trust keys. The working/kernel tree is prepared for Linux. Do not modify the control tree casually.
- **Saving too early.** `setenv` is transient. `saveenv` writes a configured storage backend, and Chapter 22's modern migration has none; Chapter 19's example patches also use RAM-only storage, unlike the unmodified vendor header. Production persistence needs a reviewed layout/recovery policy.
- **Wrong format or architecture.** `bootm` consumes FIT/legacy images, not an arbitrary raw zImage. `arch = "arm64"` metadata does not convert a 32-bit ARM payload.
- **Hashes mistaken for authorization.** A valid hash-only FIT is not signed, and an optional signature is not necessarily enforced. Do not treat a successful example boot as verified boot.

23.10 Going deeper

Use the versioned bootm, bootz, source and FDT references alongside the actual config. FIT documentation lives in `doc/usage/fit/` in this release, not an assumed old `doc/uImage.FIT/` path.

For the Linux half, follow the kernel command-line reference and the matched kernel source/config in Part IV. The NFS-root reference helps distinguish direct kernel mounting from initramfs-controlled early userspace.

> Next chapter: **Chapter 24: Workflows: TFTP, NFS, USB-OTG.** We will use the same load/handoff contract with files served from the host, while keeping network-root and recovery prerequisites explicit.

Chapter 24: Workflows, TFTP, NFS, USB-OTG

Suppose you change one driver and want to try it again. Must that also mean rewriting the bootloader and moving a card between machines? Once a board has a qualified U-Boot and wired Ethernet path, it can load new kernel artifacts into RAM. Linux can then mount a root filesystem served by the host. The bootloader, kernel and userspace no longer have to share one update mechanism.

There is a catch worth understanding before setting this up: **U-Boot's Ethernet driver stops being responsible at kernel handoff**. Linux must bring up its own driver, configure its own address and reach the NFS server. A successful TFTP transfer is useful evidence, but it is not evidence that NFS-root will work.

Chapter 19's first vendor MINI build deliberately disables networking and video: it is a UART/SD/eMMC baseline, not a TFTP bootloader. The pinned vendor header selects SMSC support for older LAN8720A carriers; **MINI V2.2 uses SR8201F**. Shared core support does not establish a working link.

Network examples here use Chapter 22's separate **upstream v2026.04 migration**, commit `88dc2788777babfd6322fa655df549a019aa1e69`, with ENET2/address-1 wiring and the source-checked RTL8201F ID-match candidate. They remain optional until actual PHY ID, reset, 50 MHz reference and transfers are validated on this carrier. Keep local storage from Chapter 23 while those tests remain undone.

Target commands also require Chapter 23's approved RAM plan and matched kernel/DT/rootfs. Modern examples name the **kernel** `imx6ull-pa-mini.dtb`; a vendor-kernel set instead may contain `imx6ull-alientek-emmc.dtb` at another output path. Choose the matched pair explicitly, never rename an EVK or U-Boot control DTB. Without artifacts and board validation, follow host/policy examples and return after Parts IV-V.

> **Execution boundary:** The host checks assume the isolated lab services from Chapter 3 are already configured. Stop if that setup is missing rather than changing unrelated host exports or firewall rules. Target examples are optional until the board and artifacts are qualified. Keep environment edits temporary, preserve a known-good recovery path, and do not write media or change fuses for these examples.

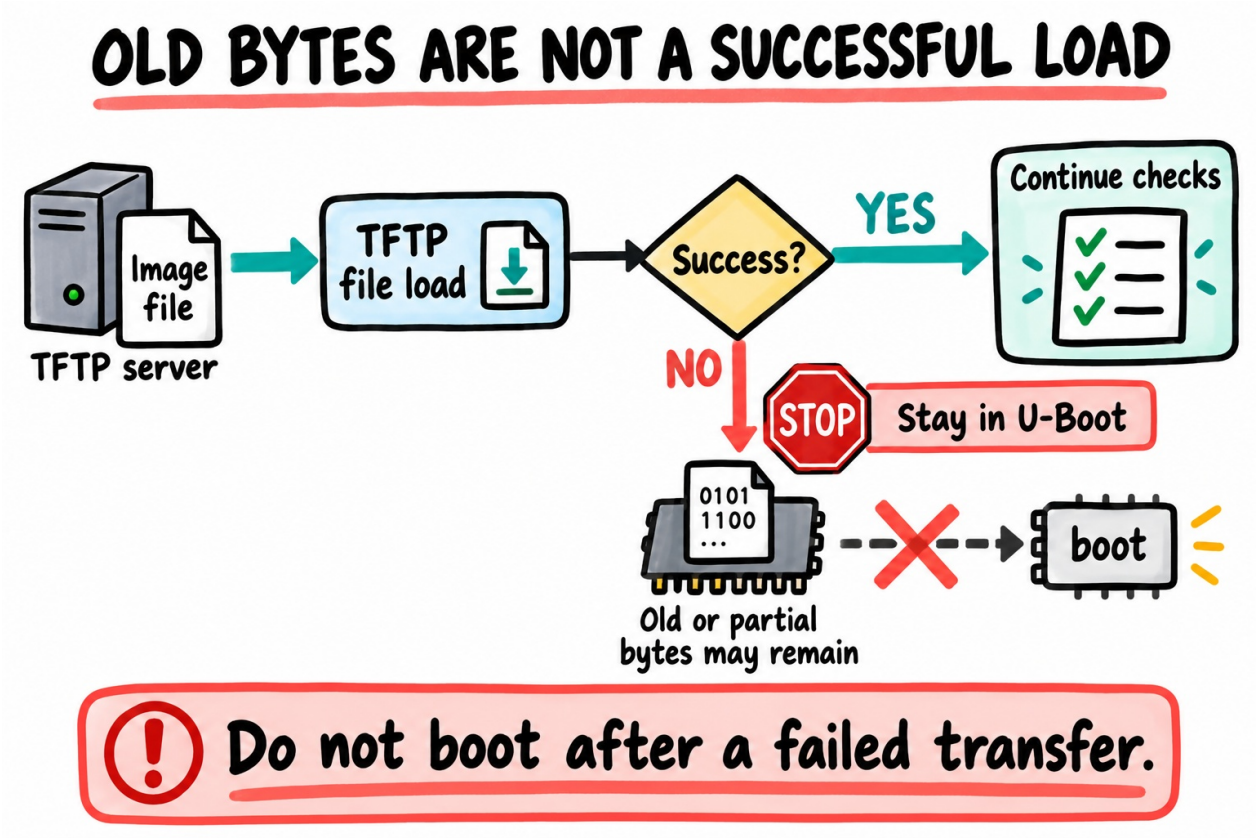
24.1 Three transports, three jobs

Transport	Job in this workflow	Important boundary
TFTP, Trivial File Transfer Protocol	U-Boot retrieves `zImage` and kernel DTB, or one FIT, into RAM	No authentication/encryption; load failure must block boot
NFS, Network File System	Linux mounts the host's exported directory as root	Needs Linux drivers and server access before userspace exists
USB serial download using `uuu`	ROM loads a board-compatible recovery bootloader	Later script steps may write persistent media; not automatically RAM-only

U-Boot itself may still come from SD/eMMC. Both the selected vendor core recipe and modern candidate use ROM+DCD followed by full U-Boot: **no SPL in these configs**. A network-loaded kernel does not remove the need for a valid first-stage image or a separately qualified USB recovery path.

Keep the lab on an isolated wired link. The example address plan is host `192.168.7.1/24`, board `192.168.7.2/24`, with no gateway required on that direct subnet. Use it only if Chapter 3's actual host interface and routing match. WSL's private NAT address is not automatically the host address a physical board can reach; USB visibility and NFS-server support also depend on the WSL/host arrangement.

24.2 TFTP for the kernel



A failed transfer does not make its destination empty or trustworthy. Stop that boot path instead of using the bytes left behind. Every required file needs its own successful load.

Stage layout on the host

Where does the server look for a filename? In Chapter 3's secure `tftpd-hpa` setup, `/srv/tftp` is the server root. The board requests a **relative** name such as `mini-lab-r1/zImage`, not a path into your home directory.

Do not symlink from that secure root to `~/imx6ull/src/linux`. A chrooted server cannot follow that link outside its root, and user-directory permissions may block it even without a chroot. Instead stage a complete, identifiable artifact set in a new directory:

```
/srv/tftp/
    mini-lab-r1/
        zImage
        imx6ull-pa-mini.dtb
        artifacts.sha256
    known-good-r1/
        zImage
        imx6ull-pa-mini.dtb
```

`known-good-r1` means artifacts with a recorded successful test on this exact board, not an EVK image renamed as a rescue set. The directory name records a lab artifact revision, not a Linux or U-Boot version.

After the matched kernel build and qualified MINI kernel DT from Part IV exist, use the build output directory rather than guessing an in-source path. These commands assume the **modern kernel layout**, `~/imx6ull/build/kernel/arch/arm/boot/dts/nxp/imx/`, and a writable lab staging root from Chapter 3. An old vendor kernel commonly puts DTBs directly under `arch/arm/boot/dts/`; inspect the selected output and use its actual MINI filename/path, not these paths unchanged:

```
$ . ~/imx6ull/scripts/env.sh
$ test -r ~/imx6ull/build/kernel/arch/arm/boot/zImage
$ test -r ~/imx6ull/build/kernel/arch/arm/boot/dts/nxp/imx/imx6ull-pa-mini.dtb
$ mkdir /srv/tftp/mini-lab-r1
$ cp ~/imx6ull/build/kernel/arch/arm/boot/zImage /srv/tftp/mini-lab-r1/zImage
$ cp ~/imx6ull/build/kernel/arch/arm/boot/dts/nxp/imx/imx6ull-pa-mini.dtb /srv/tftp/mini-
lab-r1/imx6ull-pa-mini.dtb
$ cd /srv/tftp/mini-lab-r1
$ stat -c '%n %s bytes' zImage imx6ull-pa-mini.dtb
$ sha256sum zImage imx6ull-pa-mini.dtb > artifacts.sha256
$ sha256sum --check artifacts.sha256
```

Stop on any failure. `mkdir` deliberately fails if that revision already exists: choose a new name instead of replacing artifacts during a load. Finish staging both files, check their sizes against the RAM plan, verify server readability, then select this directory at the board. The checksum manifest is a host record of the artifact pair, not automatic TFTP authentication or target verification. A FIT can keep the pair inside one container but is still unauthenticated unless signature policy is enforced.

TFTP from U-Boot

At a qualified U-Boot prompt, check `help tftpboot`, the selected Ethernet interface, and the temporary network settings. The v2026.04 command is `tftpboot`; abbreviated `tftp` is common, but the explicit name makes the helper easier to inspect:

```
=> setenv ipaddr 192.168.7.2
=> setenv serverip 192.168.7.1
=> setenv netmask 255.255.255.0
=> setenv gatewayip
=> setenv bootdir mini-lab-r1
=> setenv fdtfile imx6ull-pa-mini.dtb
=> setenv kernel_addr_r 0x82000000
=> setenv fdt_addr_r 0x83000000
=> ping ${serverip}
```

Clearing `gatewayip` is appropriate only for this direct-subnet example. MAC allocation, PHY routing, clocks and resets must already be qualified; do not copy an `ethaddr` from another board. The modern control DT aliases its sole enabled MAC, ENET2, as Ethernet 0; Linux interface naming is separate. Inspect actual enumeration. Inherited EVK `ethprime=eth1` or vendor `ethprime=FEC` is not board validation.

For a transfer-only check, set a helper whose second load is gated by the first:

```
=> setenv netload 'if tftpboot ${kernel_addr_r} ${bootdir}/zImage && setenv kernel_size $
{filesize}; then if tftpboot ${fdt_addr_r} ${bootdir}/${fdtfile} && setenv fdt_size $
{filesize}; then true; else echo DTB transfer or size capture failed; false; fi; else
echo Kernel transfer or size capture failed; false; fi'
=> run netload
```

Record the actual transfer status and sizes; no sample speed or success log is promised. `filesize` is replaced by each successful load. On failure, old data and size variables are not trustworthy. Do not issue a manual `bootz` just because buffers still contain something. Section 24.6 supplies the gated **load + arguments + boot** helper.

Speed

Measure on your actual link if transfer time matters. PHY link negotiation, TFTP options/block sizes, server scheduling, USB Ethernet adapters and file size all affect it. A rebuild still needs a new transfer and kernel restart; TFTP does not make an already running kernel change in place.

24.3 NFS for the rootfs

Export from the host

The **rootfs** is the filesystem mounted at `/`, containing programs, libraries and configuration. An NFS root moves its backing files onto the host. It does not turn arbitrary host binaries into Arm programs or supply missing target runtime libraries.

Use the rootfs preparation procedure from Part V before this step. An archive must be trusted and its numeric ownership, modes, symlinks and target ABI preserved in a dedicated tree. Extracting it as a normal user can lose ownership or fail on special files; extracting an unreviewed archive as root is unsafe. A directory listing containing `bin` and `etc` is not a functional-rootfs test. Check the selected init and its dependencies.

Inspect the **existing** host service configuration and active exports. These are inspection commands, not instructions to reconfigure or restart the host here:

```
$ systemctl status tftpd-hpa --no-pager
$ systemctl status nfs-kernel-server --no-pager
$ cat /etc/exports
$ sudo exportfs -v
$ ip -br address
$ ip route
```

A narrowly scoped export might look like this illustrative configuration, with `you` replaced by the real absolute path:

```
/home/you/imx6ull/rootfs 192.168.7.2(rw,sync,no_subtree_check,root_squash)
```

`root_squash` maps client root to an unprivileged server identity. It can work for readable files but prevents many root-owned writes. If the chosen development root requires client-root writes, `no_root_squash` is a deliberate exception granting the board substantial authority over that exported tree. Limit it to the single board address and isolated link, never `*`, and use a disposable dedicated rootfs. Do not export your home directory itself or weaken host-wide permissions to get past a mount error. See the server's exports reference.

Changing exported file contents does not require restarting NFS or re-running `exportfs`. Changing the export definition does require an administrator's reviewed reload, with existing exports preserved. Do not perform that reload merely to publish a rebuilt application. Keep a separate rootfs per concurrently running target so their runtime writes do not collide.

NFS-root from the kernel cmdline

Can Linux mount NFS before it loads a module from that same root? No. For **direct kernel NFS-root without an initramfs**, at least these capabilities and their dependencies must be built in (`=y`), not modules:

Kernel setting/capability	Why it is needed before userspace
`CONFIG_NET`, `CONFIG_INET`, `CONFIG_IP_PNP`	Networking, IPv4 and early address configuration
`CONFIG_NFS_FS`, `CONFIG_ROOT_NFS`, `CONFIG_NFS_V3` for this v3 example	NFS client and root mounting
`CONFIG_FEC`, PHY support and the actual PHY driver	Linux's i.MX Ethernet path, independent of U-Boot's driver
Required pinctrl, clocks, regulator/reset providers and matched kernel DT	Hardware dependencies for that Ethernet path
`CONFIG_IP_PNP_DHCP`, if choosing DHCP	A DHCP request cannot work from an option string alone

For MINI V2.2, check the Linux driver against the actual SR8201F ID and this carrier's reset/clock route. U-Boot's Realtek ID match is not a test of the selected kernel driver. There may be further board/kernel dependencies. Verify them in the selected Part IV kernel `.config` and boot messages; the U-Boot config is not a substitute. An initramfs can supply modules and perform a different mount procedure, but then its `/init` owns that logic. An executable `/init` can prevent the direct `nfsroot=` path from being used.

For our static, direct-subnet example the final kernel command line is:

```
console=ttymxc0,115200 root=/dev/nfs nfsroot=192.168.7.1:/home/you/imx6ull/
rootfs,vers=3,tcp ip=192.168.7.2:192.168.7.1::255.255.255.0:pa-mini:eth0:off rw
```

Replace the host path and confirm that `eth0` is Linux's intended interface. The first seven positional fields are:

```
ip=<client-ip>:<server-ip>:<gateway-ip>:<netmask>:<hostname>:<device>:<autoconf>
```

The empty gateway is intentional for this direct link. `off` disables automatic address discovery, not Ethernet. This is Linux configuration; it is not inherited automatically from U-Boot `ipaddr`. Optional later fields can specify DNS/NTP addresses. The Linux NFS-root reference explains the format and early-userspace caveat.

After a real boot, establish what mounted rather than looking for a fabricated welcome banner:

```
target# cat /proc/cmdline
target# cat /proc/mounts
target# cat /proc/net/pnp
target# uname -r
```

Find the `/` entry in `/proc/mounts` and check its source/type/options. The `target#` prefix means a target root shell, not a host command. Do not enter these at U-Boot.

Which files are needed before NFS is reachable?

Almost any file can be stored in an exported directory. The useful question is **when that file becomes reachable**:

- The bootloader must run before this workflow can reach the NFS root. It comes from its qualified local/USB boot path.
- U-Boot loads the kernel and kernel DTB using TFTP here. Their later copies in the NFS tree cannot retroactively change the running kernel or its initial DT.
- Modules can live under `/lib/modules/<kernel-release>` on NFS **after** root is mounted. They must match the running kernel; a module required to reach root must instead be built in or supplied by early userspace.
- Applications/configuration can be updated in the exported tree. NFS caching and application behavior determine when a process observes a change. A running executable does not become new code because its file changed; restart it deliberately.

Do not replace or delete the backing export directory while a target uses it. New module/rootfs sets belong in a staged tree and a controlled reboot plan, not a blind `modules_install` over a running system.

Speed and reliability

A wired development root keeps the network path independent of any Wi-Fi driver you are developing. If the link/server fails, a hard NFS mount may block filesystem accesses. Have serial access and a known-good local-root alternative; do not promise that every edit is visible immediately.

For a mount timeout, first check Linux link/PHY discovery, IP configuration, server route, export path/client permissions and RPC reachability. NFSv3 commonly needs rpcbind/mountd as well as the NFS port. TFTP also uses a transfer port beyond the initial UDP/69 request. A blanket rule for only UDP/69 and TCP/2049 may therefore be insufficient. Review interface/client-scoped firewall rules; do not disable the host firewall.

A temporary **`nfsrootdebug`** kernel parameter can expose the direct NFS-root choices. It is not `nfsroot=...,debug`, and `nfs.callback_tcpport=0` is not a general v3 mount repair. `rootwait` does not repair a network link or NFS timeout. Version 3 here is an explicit lab choice, not a claim that NFSv4 is inherently unreliable.

24.4 USB-OTG recovery via uuu

If the local bootloader cannot start, TFTP cannot help: there is no U-Boot network stack running yet. The i.MX serial-download path can provide an earlier entry through ROM, provided the exact board has a routed USB download port, correct power/boot-mode straps and a compatible, qualified image. Switch labels and USB connector roles come from the supplied board schematic/manual, not the EVK’s switch positions.

Review a recovery recipe

There is no universal complete flashing recipe for a MINI here. Instead, review one supplied by the board owner before using it. NXP’s official examples distinguish downloading an i.MX6/7 bootloader from built-in scripts that write SD/eMMC. Our selected DCD/no-SPL image is not the SPL/SDPU sequence from a different board.

Use this review checklist, not a runnable placeholder script:

Stage	Evidence required
ROM discovery/SDP	Exact SoC protocol, download-port routing, boot-mode selection, host USB access and cable

Stage	Evidence required
Boot image download	Board-qualified DCD/power/pads, image format, load/entry layout and any existing authentication policy
U-Boot protocol transition	Actual gadget/controller and command support; reaching a prompt does not automatically start Fastboot
Optional persistent programming	Explicit storage controller/area, partition map, offsets, actual byte/block counts, backup and read-back procedure
Return to normal boot	Documented strap restoration and a known-good image for this physical revision

Fastboot commands need a bootloader built and configured for that USB gadget path. Neither Chapter 19's baseline nor Chapter 22's modern minimum is a ready-made manufacturing image. The modern minimum excludes USB; the vendor host stack is not proof of a configured Fastboot gadget. Check NXP's U-Boot requirements and the actual v2026.04 config rather than assuming `uuu` adds missing firmware features.

A script containing `FB: flash`, `mmc write`, erases, or persistent environment saves changes storage even if its first operation only downloaded into RAM. Inspect every stage and any built-in recipe. Do not copy arbitrary block counts, wipe the first megabyte to simulate a fault, or burn fuses as part of a recovery example.

When to use it

Prepare recovery before changing the persistent bootloader. USB serial download is useful when the board's qualified recovery path has been established; it is not a cure for an incorrect DDR sequence or a power fault. During ordinary kernel/userspace iteration, keep using the known-good bootloader and change only the intended artifacts.

A non-destructive reader example is to document how to select ROM download mode and identify the device without sending an image. Record the actual SoC/USB identifiers and host permissions. No USB transfer or recovery success is claimed in this chapter.

24.5 The canonical development loop

There are two loops now, not one magical live-update mechanism:

```
Kernel/DT source -> matched build -> new TFTP artifact directory
    -> gated U-Boot transfer -> kernel restart -> inspect actual logs

Userspace source -> target-ABI application/rootfs staging
    -> controlled update in the exported tree -> restart/read as appropriate
```

For a kernel change, reuse the selected config and output directory. `CONFIG_LOCALVERSION` is a useful deliberate identity set in the editor/menuconfig, unlike appending invalid text to a driver. Build kernel, DTBs and modules together:

```
$ . ~/imx6ull/scripts/env.sh
$ cd ~/imx6ull/src/linux
$ make O="$HOME/imx6ull/build/kernel" ARCH=arm \
    CROSS_COMPILE=arm-none-linux-gnueabihf- -j"$(nproc)" zImage dtbs modules
$ make O="$HOME/imx6ull/build/kernel" ARCH=arm \
    CROSS_COMPILE=arm-none-linux-gnueabihf- kernelrelease
```

Record the release, config, source revision and artifact hashes. Stage the next TFTP directory using Section 24.2. If modules changed, prepare them in a **separate** rootfs staging tree with the Part V procedure. For example, this writes only a chosen module staging directory, not the running export:

```
$ make O="$HOME/imx6ull/build/kernel" ARCH=arm \
    CROSS_COMPILE=arm-none-linux-gnueabihf- \
    INSTALL_MOD_PATH="$HOME/imx6ull/build/modules-next" modules_install
```

Review that output's ownership/dependencies before deploying it. `INSTALL_MOD_PATH` is a host destination, not a target path. Compare the running target's `uname -r` and module ABI/vermagic with the intended build after the controlled reboot.

For a configuration-file example, create a noncritical marker in the dedicated exported rootfs using your editor, then read it from the target. Do not choose credentials, init scripts or a currently executing binary for the first experiment. Record when the new content is observed; caching and a program's own reload policy matter. No fixed transfer or edit-to-boot timing is promised.

24.6 Some useful U-Boot env helpers

Here is the full temporary network-root chain. These are **U-Boot prompt commands**, not `env.sh` contents. The example assumes the verified addresses, console route and Linux `eth0` mapping already discussed. Replace `nfspath` with the actual existing export, and set `bootdir` only after its artifact pair is complete:

```
=> setenv serverip 192.168.7.1
=> setenv ipaddr 192.168.7.2
=> setenv netmask 255.255.255.0
=> setenv gatewayip
=> setenv hostname pa-mini
```

```
=> setenv netdev eth0
=> setenv console ttymxc0
=> setenv nfspath /home/you/imx6ull/rootfs
=> setenv bootdir mini-lab-r1
=> setenv fdtfile imx6ull-pa-mini.dtb
=> setenv kernel_addr_r 0x82000000
=> setenv fdt_addr_r 0x83000000
=> setenv nfsargs 'setenv bootargs console=${console},115200 root=/dev/nfs nfsroot=$
{serverip}:${nfspath},vers=3,tcp ip=${ipaddr}:${serverip}:${gatewayip}:${netmask}:$
{hostname}:${netdev}:off rw'
=> setenv devel_boot 'if run netload && run nfsargs; then bootz ${kernel_addr_r} - $
{fdt_addr_r}; else echo Network load or arguments failed - staying in U-Boot; false; fi'
=> run devel_boot
```

Define `netload` exactly as in Section 24.2 first. `netdev` here supplies the **Linux** `ip=` field; it does not select U-Boot's Ethernet device. `nfsargs` is a command helper, so `run nfsargs` expands its variables at execution time. An environment string containing `${serverip}` is not recursively expanded just because another value refers to it.

Both `run netload` and `run nfsargs` must return success before `bootz` is reached. If constructing `bootargs` fails, stop at the prompt rather than booting with a previous command line. Keep the `&&` gate when extending this helper.

For a later session-level autoboot choice:

```
=> setenv bootcmd 'run devel_boot'
```

Keep that assignment unsaved during bring-up. It does not alter Chapter 22's disabled-autoboot delay on its own. Persistent autoboot is a separate product decision requiring a qualified environment backend, failure/recovery policy and deliberate delay settings. Chapter 22's `ENV_IS_NOWHERE` migration cannot persist it; Chapter 19's example patches also use RAM environment, unlike the unmodified vendor header.

For a modern one-shot local-root comparison, retain Chapter 23's `sdargs`/`sdboot` and use `run sdboot`. For the vendor baseline, use its separate `emmcargs`/`emmcboot`, restoring the matched vendor-kernel DTB filename and verified root mapping. That helper obtains a real root partition UUID and gates all loads. Do not silently replace the NFS arguments with guessed `/dev/mmcblk0p2` numbering or retain NFS arguments while testing a local root.

Failed-DTB case: If the DTB transfer fails while an earlier DTB remains in RAM, `netload` returns failure. `devel_boot` prints the error and stays in U-Boot. The old bytes are irrelevant to the decision.

24.7 Key Notes

- TFTP loads boot payloads into RAM, NFS backs Linux’s root filesystem, and ROM USB download enters an earlier recovery stage. They solve different failures and have different prerequisites.
- Linux owns Ethernet and IP configuration after handoff. Direct NFS-root needs the driver/provider, IP and NFS path before root is reachable, either built in or supplied by a deliberate initramfs design.
- An immutable staged kernel/DTB set avoids mixed revisions during transfer. Hash records identify artifacts but do not authenticate TFTP or qualify the board they target.
- Host exports, service bindings and firewall rules are persistent host policy. A narrow isolated export limits authority, especially when client-root writes justify `no_root_squash`.
- Kernel replacement requires a new load and restart. NFS file changes have caching and process-lifetime consequences, and modules must match the running kernel.
- `uuu` may cross from ROM download into writable bootloader protocols. A RAM download at the first stage does not make later flash, erase or environment-save commands harmless.

24.8 Pitfalls

- **The server works from the host but not from the board.** Verify the physical-board route, service bind address and scoped firewall policy. WSL NAT and USB forwarding are separate concerns.
- **A secure-root symlink points into home.** Stage real readable files under the TFTP root. `--secure` changes the server’s filesystem root; it is not a setting that forbids all subdirectories.
- **U-Boot ping passed, so Linux networking must work.** Linux starts its own driver and IP setup. Check built-in dependencies and the kernel DT.
- **NFS root permissions fixed with a wildcard exception.** `no_root_squash` is an authority grant, not a universal mount prerequisite. Keep any justified exception to one board on an isolated link.
- **Drivers are modules on the unreachable root.** Build the boot-critical path in, or use an explicitly designed initramfs. An `/init` changes who mounts root.
- **Old artifacts are mistaken for a new build.** New directory names and hashes make staging traceable. Check running release/module compatibility, not a fabricated log date.

- `uuu` **recipe assumed harmless.** Inspect protocol transitions and writes. A ROM download and a Fastboot storage write are different operations with different prerequisites.
- **Network timeouts treated as block-device delay.** `rootwait` is not a PHY/NFS fix. Investigate the failing network stage and retain serial/recovery access.

24.9 Going deeper

Use the Linux NFS-root guide for `ip=`, `nfsrootdebug` and early-userspace behavior. Read your installed tftpd-hpa manual and exports manual alongside the actual Chapter 3 service files; the linked manuals describe options, not proof of your installed service version/configuration.

NXP's UUU examples and U-Boot requirements explain the tool/protocol boundary. Check a board-owner recipe against your pinned U-Boot build before any future recovery operation.

The core path now has a concrete test for each boundary: a versioned bootloader build, a hardware-qualified port, checked artifact loads, a kernel handoff and an observed root mount. The source/build examples alone do not establish all five. Keep their remaining evidence requirements visible as you move into Linux and rootfs development.

> Next chapter: **Chapter 24A: Building i.MX6ULL U-Boot from nothing.** That optional deep dive changes the level of platform work. It is not a requirement to replace the working, qualified bootloader during ordinary kernel iteration.

Chapter 24A: Building an i.MX6ULL teaching port from scratch

On an MCU, you can often reach `main()` by supplying a vector table, a linker script, and a few clock writes. Here the Boot ROM must first make external memory usable, load a much larger executable, and transfer control to it. U-Boot then builds its own runtime: a stack, global data, device discovery, relocation, and finally a command interpreter. A serial message at one stage does not prove that the later stages work.

Chapter 19's first practical MINI build uses the official Alientek vendor source at commit `edb7ca5ac4be2d978be60a2a12c61e0b6d1f7feb` with `mx6ull_alientek_emmc_defconfig`. Keep that baseline for first board work. This optional chapter takes a deeper educational route on upstream U-Boot: keep the generic ARMv7 startup and frameworks, but give UART1, GPT1, and USDHC2 their own small register-level implementations. The silicon is not new to upstream U-Boot. Our new `ARCH_IMX6ULL` selection is a separate teaching route, not a replacement for the normal `ARCH_MX6` port and not a qualified full-chip port.

The supplied listings build against **upstream v2026.04, commit `88dc2788777babfd6322fa655df549a019aa1e69`** and include the vendor-documented DDR initializer for the supplied 512 MiB eMMC core-board design. Section 24A.14 identifies its exact provenance, and section 24A.21 creates the actual ROM+DCD image without an omitted file. Source compilation and image inspection are verified checkpoints, not board-test results. No DDR calibration measured by us or successful boot of this custom port is claimed here.

This chapter follows one rule:

> Each supplied new file is shown in full. For existing files, insertion fragments identify the manual edit location; they are not standalone `git apply` patches. DDR constants come from a named vendor source, not from a guessed table. Framework code is named where it enters our boot path.

We reuse U-Boot's architecture startup, driver model, MMC protocol state machine, block layer, and command shell. Our three small drivers make the peripheral boundary visible. We do not implement power management, all clock roots, security setup, or every silicon erratum. Treat those omissions as reasons to use the upstream platform for product work, not as proof that they are unnecessary.

24A.1 The exact hardware used in this chapter

A real low-level port cannot use a fictional chip. Register values that are "close enough" produce a silent board.

This chapter uses this exact target:

Item	Value used here
Board	Point Atom MINI i.MX6ULL board
CPU	NXP i.MX6ULL, one Cortex-A7 core
DDR	Supplied core V2.0: one x16 512 MiB DDR3L `NT5CC256M16EP-EK`, using the documented eMMC-core vendor DCD
First boot medium	Removable SD card
On-board storage to inspect later	eMMC on USDHC2, eight wired data lines; begin in 1-bit legacy mode
Console	UART1 through the board's built-in USB-to-TTL circuit
Console settings	115200 baud, 8 data bits, no parity, 1 stop bit
U-Boot source	Upstream U-Boot v2026.04
Cross compiler prefix	`arm-none-linux-gnueabihf-`

`ARCH_IMX6ULL` is our new architecture symbol. We deliberately do not select `ARCH_MX6` or link `arch/arm/mach-imx/`. In v2026.04, the standard `mx6ull_14x14_evk_defconfig` uses `board/nxp/mx6ullevk/`; it has a ROM+DCD image and no SPL. We inspect that source to check APIs and formats, but do not borrow its EVK DDR values as MINI board data.

The supplied core V2.0 and MINI baseboard V2.2 schematics, the vendor guide's MINI applicability table, and its eMMC-core DDR description support this documented target. Compare them with the actual assembly before hardware work: the guide explicitly allows future DDR substitutions. Keep a hardware ledger with board revision, fitted DDR/eMMC markings, rails, BSP revision, inherited DDR clock, and calibration/test records. The complete source image can be built now, but its suitability for a particular physical assembly and its runtime stability remain hardware checks.

Values that must change on another board

Do not use this image unchanged on an i.MX6ULL board with a different DDR part or PCB layout.

Value	Why it is board-specific
DDR calibration registers	They compensate for signal delay on this PCB and DDR layout.
DDR geometry and timing	They describe the fitted memory device.
UART pads	Another board may route UART1 to different pads.
eMMC pads and bus width	Another board may use USDHC1, a 4-bit bus, or no eMMC.
Boot switches	Their physical positions are board-specific.

The SoC register addresses do not change between boards that use the same i.MX6ULL silicon.

24A.2 The complete first boot path

The i.MX6ULL has 128 KiB of OCRAM. A normal full U-Boot image does not fit there. We have two possible designs:

1. The Boot ROM loads a small SPL into OCRAM. SPL initializes DDR and then loads full U-Boot.
2. The Boot ROM reads a DCD table from the image header, performs the DDR register writes itself, and then loads full U-Boot directly into DDR.

We use the second design. The intended milestone is a U-Boot prompt after a qualified ROM+DCD load; adding SPL would introduce a different executable and load contract to debug.

```
SD card
  offset 0x00000400: i.MX image header, IVT, Boot Data, and DCD
  following bytes:   u-boot.bin
          |
          v
i.MX6ULL Boot ROM
  1. Finds the image at SD offset 0x400
```

```
  2. Copies the small header and DCD into internal memory
  3. Performs every DCD register write in order
  4. DDR is usable only if that board's DCD is correct
  5. Loads u-boot.bin to 0x87800000 in DDR
  6. Jumps to the entry address 0x87800000
          |
          v
Our IMX6ULL platform code
  1. Selects simple, known clock sources
  2. Initializes UART1 and prints an early marker
  3. Continues board_init_f: early DM/board marker, timer and console
  4. Declares the conditional DDR bank and reserves relocation space
          |
          v
Generic ARM handoff in crt0.S
  1. board_init_f returns; switch stack/global data to reserved DDR
  2. relocate_code copies/fixes the executable and resumes there
  3. Set final runtime state, clear BSS, and enter board_init_r
          |
          v
U-Boot after relocation
  1. Rebuilds driver-model state and the normal console
  2. Binds the USDHC2 host; card identification occurs when requested
  3. Shows the imx6ull=> prompt
```

There is no SPL in this design. The ROM executes the documented board-specific DCD before the first U-Boot instruction. `dram_init()` later reports a memory bank; it cannot repair an incorrect DCD because its own code is already running from DDR.

Memory map used by this port

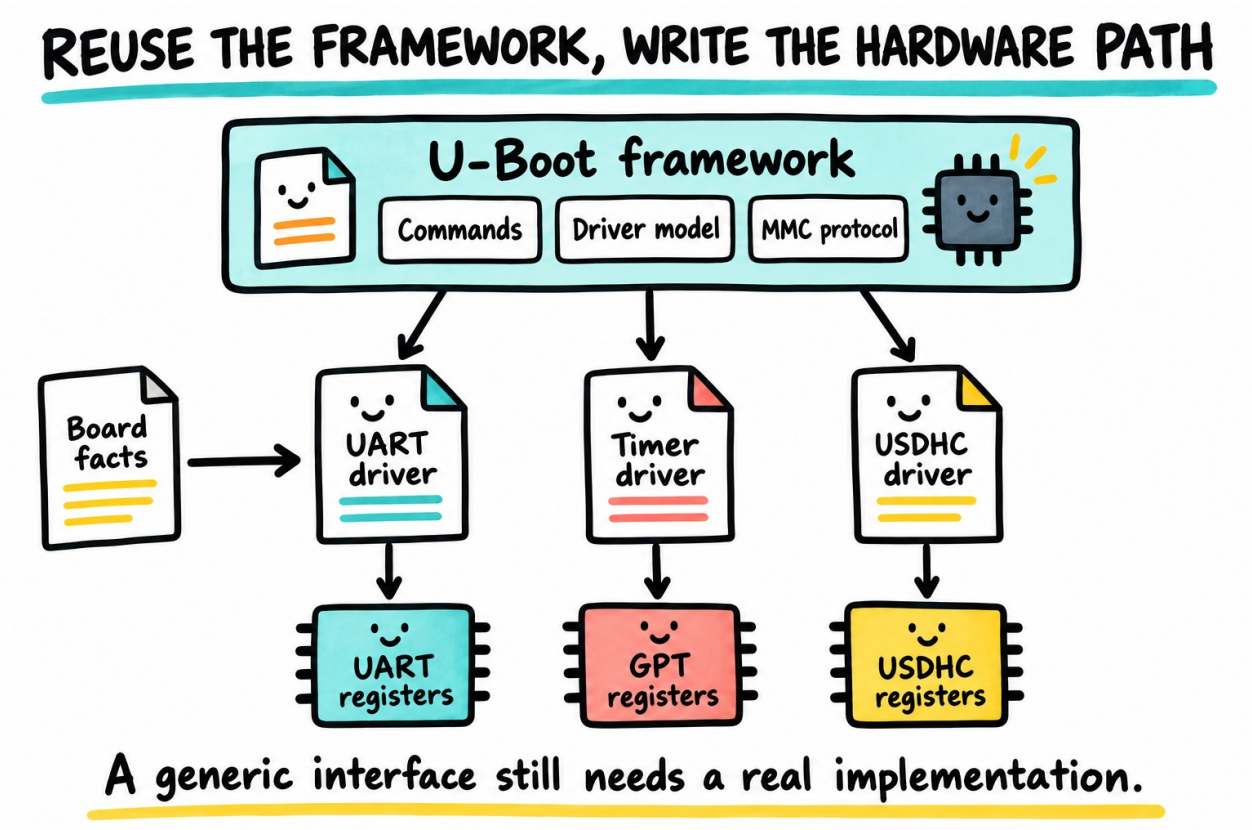
Address or range	Use
`0x00900000` to `0x0091FFFF`	128 KiB OCRAM used for the first stack and global data after ROM handoff
`0x80000000` to `0x9FFFFFFF`	Configured 512 MiB DDR bank for the documented core, subject to physical validation
`0x80000100`	Linux boot-parameter address reported by the board code
`0x82000000`	Default kernel or test-file load address
`0x83000000`	Default Device Tree load address for Linux

Address or range	Use
`0x87800000`	U-Boot link address, ROM load address, and first entry address

`CONFIG_TEXT_BASE` and the `mkimage -e` argument must both be `0x87800000`. If they differ, the ROM can load the bytes correctly and still jump to the wrong place.

OCRAM is 128 KiB, not the larger aliased address window above it. The ROM reserves parts of OCRAM while booting; the reference manual permits its use after normal boot handoff. This teaching port does not call back into the ROM. Its early stack and global data use OCRAM, while the executable and appended DTB already reside in DDR. Relocation later reserves U-Boot, stack, malloc, and other runtime data near the top of the reported bank. The load addresses in this table are conventions, not ownership reservations.

24A.3 What we write, edit, and reuse



Reuse the framework, but make the hardware boundary explicit. A common interface cannot talk to a new controller until a driver implements its register-level contract. The icons represent software layers, not physical board wiring.

Create these files:

```
arch/arm/mach-imx6ull/
|-- Kconfig
|-- Makefile
|-- clock.c
|-- cpu.c
|-- early_uart.c
`-- include/mach/
    |-- clock.h
    |-- hardware.h
    `-- uart.h

board/point-atom/imx6ull-mini/
|-- Kconfig
|-- MAINTAINERS
|-- Makefile
|-- board.c
`-- imximage.cfg

arch/arm/dts/
|-- imx6ull-from-scratch.dtsi
|-- imx6ull-point-atom-mini-from-scratch.dts
`-- imx6ull-point-atom-mini-from-scratch-u-boot.dtsi

drivers/serial/serial_imx6ull.c
drivers/timer/imx6ull_gpt_timer.c
drivers/mmc/imx6ull_usdhc.c
configs/imx6ull_point_atom_mini_defconfig
include/configs/imx6ull_point_atom_mini.h
```

Edit these existing files:

```
arch/arm/Kconfig
arch/arm/Makefile
arch/arm/dts/Makefile
drivers/serial/Kconfig
drivers/serial/Makefile
drivers/timer/Kconfig
drivers/timer/Makefile
drivers/mmc/Kconfig
drivers/mmc/Makefile
```

Reuse these existing U-Boot files without changing their C code:

Existing file	**What we reuse**
`arch/arm/cpu/armv7/start.S` and `arch/arm/lib/crt0.S`	CPU entry/state setup, early stack/global data, and entry into `board_init_f()`
`arch/arm/lib/relocate.S`	Copies U-Boot to its final DDR location and fixes addresses

Existing file	What we reuse
`common/board_f.c`	Initialization before relocation
`common/board_r.c`	Initialization after relocation and command loop
`drivers/mmc/mmc.c`	The hardware-independent MMC and eMMC command sequence
`drivers/mmc/mmc-uclass.c`	Connects our host driver to U-Boot's MMC uclass
`drivers/block/blk-uclass.c`	Exposes the discovered eMMC as a block device
`tools/mkimage`	Builds the i.MX IVT, Boot Data, and DCD image header

Before relocation, `board_init_f()` builds enough runtime state to size DDR and reserve the destination. `crt0.S` and `relocate.S` move to that destination; `board_init_r()` runs with writable globals and the full runtime. Driver model (DM) matches each Device Tree node to a driver and allocates its per-device data. A uclass is the common interface, such as serial or MMC, that the rest of U-Boot calls. This is the point where the MCU habit of calling one peripheral function directly becomes a framework callback contract.

24A.4 Start from a clean U-Boot tree

Use a separate checkout for this alternative port. Keep Chapter 21's command-example branch and Chapter 22's normal-MX6 board work in their existing checkout; do not check that edited tree back to the tag to make a version check pass. Run these commands on the Linux build host:

```
$ . ~/imx6ull/scripts/env.sh
$ cd ~/imx6ull/src
$ git clone --branch v2026.04 https://source.denx.de/u-boot/u-boot.git u-boot-24a
$ cd u-boot-24a
$ UBOOT_BASE=88dc2788777babfd6322fa655df549a019aa1e69
$ git describe --tags --exact-match HEAD
$ test "$(git rev-parse --verify 'v2026.04^{commit}')" = "$UBOOT_BASE"
$ test "$(git rev-parse HEAD)" = "$UBOOT_BASE"
$ git rev-parse HEAD
$ git switch -c imx6ull-from-scratch
```

Confirm the compiler before editing anything:

```
$ arm-none-linux-gnueabihf-gcc --version
$ make --version
$ dtc --version
```

Before making custom commits, the tag's resolved commit and `HEAD` must both match the pin above. Stop on any failed check. If `u-boot-24a` already exists, inspect its history and local changes instead of overwriting it. Once custom commits exist, `HEAD` is expected to differ and an exact-tag query on it is no longer the right check; section 24A.20 checks the pinned base's ancestry instead. Neither check says that uncommitted files are unchanged, so always inspect `git status --short` as well.

The compiler must be Chapter 3's project-local Linux-target Arm GNU 13.2.Rel1 compiler, with prefix `arm-none-linux-gnueabihf-`. Preserve the separate `arm-none-eabi-` installation used for bare-metal work. Source the environment in each build shell; do not edit `.bashrc`, system paths, or global Git configuration. Native GCC builds host utilities, but cannot replace the ARM cross compiler for the target executable.

Use Chapter 19's host build dependencies as well: Make, native GCC, Bison, Flex, M4, `bc`, and the host development libraries required by U-Boot's image tools. U-Boot builds its own DTC. The standalone `dtc` check is useful for inspecting trees, but does not select a target compiler.

Create the directories:

```
$ mkdir -p arch/arm/mach-imx6ull/include/mach
$ mkdir -p board/point-atom/imx6ull-mini
```

The other parent directories already exist in U-Boot.

24A.5 Build a hardware ledger before writing C

The table below is the bridge from the reference manual and Part II into U-Boot.

Hardware	Address	Facts used by our code
OCRAM	`0x00900000`	128 KiB
DDR	`0x80000000`	512 MiB only after the qualified DCD and geometry agree
UART1	`0x02020000`	i.MX UART register layout

Hardware	Address	Facts used by our code
GPT1	`0x02098000`	32-bit up-counter with a 24 MHz source
WDOG1	`0x020BC000`	Used by `reset`
CCM	`0x020C4000`	Clock selectors, dividers, and gates
IOMUXC	`0x020E0000`	Pad mux, pad electrical control, and input daisy selection
MMDC	`0x021B0000`	DDR controller
USDHC2	`0x02194000`	eMMC host controller

The first visible code uses these UART1 registers:

Register	Offset	Purpose
`URXD`	`0x00`	Received byte and receive status
`UTXD`	`0x40`	Byte to transmit
`UCR1`	`0x80`	Main UART enable
`UCR2`	`0x84`	Reset, TX, RX, word length, and flow-control settings
`UCR3`	`0x88`	RX input path setting
`UCR4`	`0x8C`	Additional control settings
`UFCR`	`0x90`	FIFO thresholds and reference-clock divider

Register	Offset	Purpose
USR2	0x98	Transmit-complete and receive-ready status
UBIR	0xA4	Baud-rate numerator
UBMR	0xA8	Baud-rate denominator
UTS	0xB4	FIFO empty and full status

For first bring-up, `arch_cpu_init()` selects the 24 MHz oscillator as the UART root clock. This is slower than the 80 MHz PLL-derived clock used in Chapter 12, but it removes a PLL dependency. The baud-rate registers are calculated from the actual selected rate.

24A.6 Connect the new platform to the ARM build

Edit `arch/arm/Kconfig`

Search for `config ARCH_KIRKWOOD`. Insert this entry immediately above it, inside the same ARM platform choice:

```
+config ARCH_IMX6ULL
+   bool "Learning i.MX6ULL platform"
+   select CPU_V7A
+   select SUPPORT_OF_CONTROL
+   select SKIP_LOWLEVEL_INIT_ONLY
+   help
+     Build the from-scratch IMX6ULL teaching port for the Point Atom
+     MINI board. This deliberately does not use arch/arm/mach-imx.
```

Near the bottom of the same file, insert our source line after the HiSTB line:

```
 source "arch/arm/mach-histb/Kconfig"
+source "arch/arm/mach-imx6ull/Kconfig"
 source "arch/arm/mach-integrator/Kconfig"
```

The two changes do different jobs:

Change	Effect
`config ARCH_IMX6ULL`	Creates the platform option and says that the CPU implements ARMv7-A.
Machine Kconfig source line	Lets Kconfig read our board target and board directory settings.

`CPU_V7A` makes U-Boot compile the generic Cortex-A7-compatible ARMv7 code. We do not write a reset vector.

`SKIP_LOWLEVEL_INIT_ONLY` skips the generic low-level platform hook because ROM+DCD already supplied DDR. Unlike `SKIP_LOWLEVEL_INIT`, it retains `cpu_init_cp15()`: the generic entry invalidates TLB/instruction state and disables the MMU and data cache before entering this runtime. The image assumes the normal ROM handoff, not an arbitrary jump from another firmware with dirty data caches. The two cache-off defconfig options also prevent later cache enabling. They do not, by themselves, clean an unknown predecessor's dirty cache.

Edit `arch/arm/Makefile`

Add one line to the sorted machine list:

```
 machine-$(CONFIG_ARCH_HISTB)          += histb
+machine-$(CONFIG_ARCH_IMX6ULL)        += imx6ull
 machine-$(CONFIG_ARCH_IPQ40XX)        += ipq40xx
```

When `CONFIG_ARCH_IMX6ULL=y`, this line adds `arch/arm/mach-imx6ull/` to the build and adds its `include/` directory to the compiler's header search path.

Create `arch/arm/mach-imx6ull/Kconfig`

```
if ARCH_IMX6ULL

config TARGET_IMX6ULL_POINT_ATOM_MINI
    bool "Point Atom MINI with 512 MiB DDR3L"
    select BOARD_EARLY_INIT_F
    help
      Build U-Boot for the Point Atom MINI i.MX6ULL board. The first
      image boots from SD and accesses the on-board eMMC through USDHC2.

source "board/point-atom/imx6ull-mini/Kconfig"

endif
```

There is only one board today, so this is a `bool`, not a `choice`. A future second board can turn this section into a choice.

`BOARD_EARLY_INIT_F` tells common U-Boot that our board supplies `board_early_init_f()`. That hook runs before relocation. We use it only for an early progress message.

Create `arch/arm/mach-imx6ull/Makefile`

```
# SPDX-License-Identifier: GPL-2.0+

obj-y += clock.o
obj-y += cpu.o
obj-y += early_uart.o
```

Each object is built into full U-Boot whenever `ARCH_IMX6ULL` is selected.

24A.7 Define the hardware addresses

Create `arch/arm/mach-imx6ull/include/mach/hardware.h`:

```
/* SPDX-License-Identifier: GPL-2.0+ */
#ifndef __IMX6ULL_HARDWARE_H
#define __IMX6ULL_HARDWARE_H

#include <linux/sizes.h>

#define IMX6ULL_OCRAM_BASE             0x00900000UL
#define IMX6ULL_OCRAM_SIZE             SZ_128K

#define IMX6ULL_DDR_BASE               0x80000000UL
#define IMX6ULL_DDR_SIZE               SZ_512M

#define IMX6ULL_UART1_BASE             0x02020000UL
#define IMX6ULL_GPT1_BASE              0x02098000UL
#define IMX6ULL_WDOG1_BASE             0x020BC000UL
#define IMX6ULL_WDOG2_BASE             0x020C0000UL
#define IMX6ULL_WDOG3_BASE             0x021E4000UL
#define IMX6ULL_SRC_SCR                0x020D8000UL
#define IMX6ULL_CCM_BASE               0x020C4000UL
#define IMX6ULL_IOMUXC_BASE            0x020E0000UL
#define IMX6ULL_MMDC_BASE              0x021B0000UL
#define IMX6ULL_USDHC2_BASE            0x02194000UL

#define IMX6ULL_CCM_CSCMR1             0x020C401CUL
#define IMX6ULL_CCM_CSCDR1             0x020C4024UL
#define IMX6ULL_CCM_CCGR1              0x020C406CUL
#define IMX6ULL_CCM_CCGR3              0x020C4074UL
#define IMX6ULL_CCM_CCGR5              0x020C407CUL
#define IMX6ULL_CCM_CCGR6              0x020C4080UL

#define IMX6ULL_ANATOP_PLL2            0x020C8030UL
#define IMX6ULL_ANATOP_PFD_528         0x020C8100UL

#define IMX6ULL_UART1_TX_MUX            0x020E0084UL
```

```
#define IMX6ULL_UART1_RX_MUX            0x020E0088UL
#define IMX6ULL_UART1_TX_PAD            0x020E0310UL
#define IMX6ULL_UART1_RX_PAD            0x020E0314UL
#define IMX6ULL_UART1_RX_SELECT         0x020E0624UL

#endif
```

This header contains addresses and sizes only. It does not initialize anything. The `UL` suffix makes each constant an unsigned long, which avoids signed-address warnings on 32-bit ARM.

24A.8 Write the clock code

Create `arch/arm/mach-imx6ull/include/mach/clock.h`:

```
/* SPDX-License-Identifier: GPL-2.0+ */
#ifndef __IMX6ULL_CLOCK_H
#define __IMX6ULL_CLOCK_H

#include <linux/types.h>

void imx6ull_clock_init(void);
u32 imx6ull_get_uart_clock(void);
u32 imx6ull_get_usdhc2_clock(void);

#endif
```

Both drivers written in this chapter call these functions directly. There is no existing i.MX clock API between our code and the registers.

Create `arch/arm/mach-imx6ull/clock.c`:

```
// SPDX-License-Identifier: GPL-2.0+
#include <asm/io.h>
#include <asm/arch/clock.h>
#include <asm/arch/hardware.h>
#include <linux/bitops.h>

#define CCM_CSCDR1_UART_CLK_PODF_MASK   0x3f
#define CCM_CSCDR1_UART_CLK_SEL         BIT(6)

#define CCM_CSCMR1_USDHC2_CLK_SEL       BIT(17)
#define CCM_CSCDR1_USDHC2_PODF_MASK     (0x7 << 16)
#define CCM_CSCDR1_USDHC2_PODF_DIV2     (0x1 << 16)

#define CCM_CCGR1_GPT1_MASK             ((0x3 << 20) | (0x3 << 22))
#define CCM_CCGR3_WDOG1_MASK            (0x3 << 16)
#define CCM_CCGR5_UART1_MASK            (0x3 << 24)
#define CCM_CCGR6_USDHC2_MASK           (0x3 << 4)

#define OSCILLATOR_HZ                    24000000U
#define PLL2_DIV_SELECT                 BIT(0)
#define PLL2_POWERDOWN                  BIT(12)
#define PLL2_ENABLE                     BIT(13)
```

```
#define PLL2_BYPASS                     BIT(16)
#define PLL2_LOCK                       BIT(31)
#define PFD2_FRAC_SHIFT                 16
#define PFD2_FRAC_MASK                  (0x3f << PFD2_FRAC_SHIFT)
#define PFD2_CLKGATE                    BIT(23)

void imx6ull_clock_init(void)
{
    /* UART root = 24 MHz oscillator, divider = 1. */
    clrsetbits_le32((void *)IMX6ULL_CCM_CSCDR1,
              CCM_CSCDR1_UART_CLK_SEL |
              CCM_CSCDR1_UART_CLK_PODF_MASK,
              CCM_CSCDR1_UART_CLK_SEL);

    /* Stop this unused host while switching its root, not the DDR PLL. */
    clrbits_le32((void *)IMX6ULL_CCM_CCGR6, CCM_CCGR6_USDHC2_MASK);
    /* USDHC2 root = existing PLL2 PFD2, divided by 2. */
    clrbits_le32((void *)IMX6ULL_CCM_CSCMR1,
             CCM_CSCMR1_USDHC2_CLK_SEL);
    clrsetbits_le32((void *)IMX6ULL_CCM_CSCDR1,
              CCM_CSCDR1_USDHC2_PODF_MASK,
              CCM_CSCDR1_USDHC2_PODF_DIV2);

    /* Value 3 in a CCGR field enables the clock in every run mode. */
    setbits_le32((void *)IMX6ULL_CCM_CCGR1, CCM_CCGR1_GPT1_MASK);
    setbits_le32((void *)IMX6ULL_CCM_CCGR3, CCM_CCGR3_WDOG1_MASK);
    setbits_le32((void *)IMX6ULL_CCM_CCGR5, CCM_CCGR5_UART1_MASK);
    setbits_le32((void *)IMX6ULL_CCM_CCGR6, CCM_CCGR6_USDHC2_MASK);
}

u32 imx6ull_get_uart_clock(void)
{
    u32 cscdr1 = readl((void *)IMX6ULL_CCM_CSCDR1);
    u32 divider = (cscdr1 & CCM_CSCDR1_UART_CLK_PODF_MASK) + 1;
    if (!(cscdr1 & CCM_CSCDR1_UART_CLK_SEL))
        return 0; /* This port only decodes the oscillator path. */
    return OSCILLATOR_HZ / divider;
}

u32 imx6ull_get_usdhc2_clock(void)
{
    u32 pll2_control;
    u32 pfd_register;
    u32 pll2_rate;
    u32 pfd2_fraction;
    u32 usdhc2_divider;
    u32 cscdr1;

    pll2_control = readl((void *)IMX6ULL_ANATOP_PLL2);
    if (!(pll2_control & PLL2_LOCK) || !(pll2_control & PLL2_ENABLE) ||
        (pll2_control & (PLL2_POWERDOWN | PLL2_BYPASS)))
        return 0;
    pll2_rate = OSCILLATOR_HZ *
            (20 + ((pll2_control & PLL2_DIV_SELECT) << 1));

    pfd_register = readl((void *)IMX6ULL_ANATOP_PFD_528);
    pfd2_fraction = (pfd_register & PFD2_FRAC_MASK) >>
            PFD2_FRAC_SHIFT;
    if ((pfd_register & PFD2_CLKGATE) ||
```

```
        pfd2_fraction < 12 || pfd2_fraction > 35)
        return 0;

    cscdr1 = readl((void *)IMX6ULL_CCM_CSCDR1);
    if (readl((void *)IMX6ULL_CCM_CSCMR1) & CCM_CSCMR1_USDHC2_CLK_SEL)
        return 0; /* PFD0 is outside this port's supported clock path. */
    usdhc2_divider = ((cscdr1 >> 16) & 0x7) + 1;

    return ((u64)pll2_rate * 18 / pfd2_fraction) / usdhc2_divider;
}
```

The UART calculation deliberately accepts only the oscillator path we selected. It does not guess an 80 MHz rate when a register says otherwise. GPT1 needs both its bus and serial gates enabled; setting only the bus gate can leave register access working while the counter does not advance.

USDHC2 uses an **inherited** PLL2/PFD2 path. We do not retune or power-cycle it: PLL2 also participates in the memory clock tree, and this code executes from DDR. The getter checks lock, enable, bypass, powerdown, PFD gate/fraction, and root selection. An unsupported state produces zero and the host probe fails. A successful ROM SD load alone does not prove every later USDHC2 clock assumption. With PLL2 at 528 MHz, fraction 24, and divider two, the calculation is 198 MHz. The host then chooses divisors that do not exceed the requested card clock. Record the inherited register state during later hardware qualification.

24A.9 Expose the early UART code

Create `arch/arm/mach-imx6ull/include/mach/uart.h`:

```
/* SPDX-License-Identifier: GPL-2.0+ */
#ifndef __IMX6ULL_UART_H
#define __IMX6ULL_UART_H

#include <linux/types.h>

struct imx6ull_uart {
    u32 rxd;
    u32 reserved0[15];
    u32 txd;
    u32 reserved1[15];
    u32 cr1;
    u32 cr2;
    u32 cr3;
    u32 cr4;
    u32 fcr;
    u32 sr1;
    u32 sr2;
    u32 esc;
    u32 tim;
    u32 bir;
    u32 bmr;
    u32 brc;
    u32 onems;
```

```
    u32 ts;
};

int imx6ull_uart_hw_init(struct imx6ull_uart *uart, u32 clock,
              u32 baudrate);
void imx6ull_early_uart_putc(char ch);
void imx6ull_early_uart_puts(const char *text);

#endif
```

The reserved arrays are not unused memory. They preserve the gaps in the hardware register map. For example, `txd` must land at offset `0x40`, not directly after `rxd` at offset `0x04`.

Create `arch/arm/mach-imx6ull/early_uart.c`:

```
// SPDX-License-Identifier: GPL-2.0+
#include <asm/io.h>
#include <asm/arch/hardware.h>
#include <asm/arch/uart.h>
#include <errno.h>
#include <linux/bitops.h>

#define UCR1_UARTEN                   BIT(0)

#define UCR2_SRST                     BIT(0)
#define UCR2_RXEN                     BIT(1)
#define UCR2_TXEN                     BIT(2)
#define UCR2_WS                       BIT(5)
#define UCR2_IRTS                     BIT(14)

#define UCR3_RXDMUXSEL                BIT(2)
#define UCR3_ADNIMP                   BIT(7)

#define UFCR_RFDIV_DIV2               (4 << 7)
#define UFCR_TXTL_2                   (2 << 10)
#define UFCR_RXTL_1                   1

#define UTS_TXFULL                    BIT(4)
#define USR2_TXDC                     BIT(3)

int imx6ull_uart_hw_init(struct imx6ull_uart *uart, u32 clock,
              u32 baudrate)
{
    u32 divisor;
    u32 budget = 100000;

    if (!clock || !baudrate || baudrate > clock / 2)
        return -EINVAL;
    divisor = (clock / 2 + baudrate / 2) / baudrate;
    if (!divisor || divisor > 65536)
        return -EINVAL;

    writel(0, &uart->cr1);
    writel(0, &uart->cr2);

    /* No timer exists yet: this is a finite poll budget, not a duration. */
```

```
    while (!(readl(&uart->cr2) & UCR2_SRST)) {
        if (!--budget)
            return -ETIMEDOUT;
    }

    writel(0x704 | UCR3_ADNIMP | UCR3_RXDMUXSEL, &uart->cr3);
    writel(0x8000, &uart->cr4);
    writel(0x2b, &uart->esc);
    writel(0, &uart->tim);
    writel(0, &uart->ts);

    writel(UFCR_RFDIV_DIV2 | UFCR_TXTL_2 | UFCR_RXTL_1,
           &uart->fcr);
    writel(0xf, &uart->bir);
    writel(divisor - 1, &uart->bmr);

    writel(UCR2_WS | UCR2_IRTS | UCR2_RXEN | UCR2_TXEN | UCR2_SRST,
           &uart->cr2);
    setbits_le32(&uart->cr3, UCR3_RXDMUXSEL);
    writel(UCR1_UARTEN, &uart->cr1);
    return 0;
}

void imx6ull_early_uart_putc(char ch)
{
    struct imx6ull_uart *uart =
        (struct imx6ull_uart *)IMX6ULL_UART1_BASE;
    u32 budget = 100000;

    while (readl(&uart->ts) & UTS_TXFULL) {
        if (!--budget)
            return; /* Missing marker is a failure, not permission to continue. */
    }

    writel(ch, &uart->txd);
}

void imx6ull_early_uart_puts(const char *text)
{
    struct imx6ull_uart *uart =
        (struct imx6ull_uart *)IMX6ULL_UART1_BASE;
    u32 budget = 100000;

    while (*text) {
        if (*text == '\n')
            imx6ull_early_uart_putc('\r');
        imx6ull_early_uart_putc(*text++);
    }
    /* Complete the marker before DM can reset the same UART's FIFO. */
    while (!(readl(&uart->sr2) & USR2_TXDC)) {
        if (!--budget)
            return;
    }
}
```

This is the bare-metal UART code. U-Boot’s console is not running when it prints the first marker. The function writes directly to UART1 in the same way as Chapter 12.

The baud formula used by this UART is:

```
baud = reference_clock / 16 * (UBIR + 1) / (UBMR + 1)
```

`UFCR` divides the 24 MHz input by two. `UBIR` is 15, so `(UBIR + 1)` cancels the `/16`. Round `12000000 / 115200` to 104, then program **`UBMR = 104 - 1 = 103`**. The resulting rate is about 115385 baud. Forgetting the minus one programs a different divider, not merely a different representation. UART reset deasserts after module-clock cycles; the early bounded loop cannot promise a millisecond timeout before GPT exists. Early `putc()` waits for FIFO space, then `puts()` drains the marker with a bounded transmit-complete check before the later DM initialization can reset the FIFO. DM `setbrg()` and `probe()` propagate initialization failures.

24A.10 Write the SoC entry hooks

Create `arch/arm/mach-imx6ull/cpu.c`:

```
// SPDX-License-Identifier: GPL-2.0+
#include <asm/io.h>
#include <asm/arch/clock.h>
#include <asm/arch/hardware.h>
#include <asm/arch/uart.h>
#include <init.h>
#include <linux/bitops.h>
#include <linux/types.h>
#include <stdio.h>

#define WDOG_WCR_WDE                    BIT(2)
#define WDOG_WCR_WDA                    BIT(5)

struct imx6ull_watchdog_regs {
    u16 wcr;
    u16 wsr;
    u16 wrsr;
};

int arch_cpu_init(void)
{
    struct imx6ull_uart *uart =
        (struct imx6ull_uart *)IMX6ULL_UART1_BASE;
    int ret;

    imx6ull_clock_init();
    /* WMCR.PDE is separate from the normal watchdog-enable bit. */
    writew(0, (void *)(IMX6ULL_WDOG1_BASE + 0x08));
    writew(0, (void *)(IMX6ULL_WDOG2_BASE + 0x08));
    writew(0, (void *)(IMX6ULL_WDOG3_BASE + 0x08));
    /* Use cold reset, matching the normal ROM+DCD re-entry contract. */
    clrbits_le32((void *)IMX6ULL_SRC_SCR, BIT(0));
    ret = imx6ull_uart_hw_init(uart, imx6ull_get_uart_clock(), 115200);
    if (ret)
        return ret;
    imx6ull_early_uart_puts("\n[imx6ull] arch_cpu_init reached\n");
```

```
    return 0;
}

int print_cpuinfo(void)
{
    puts("CPU:   NXP i.MX6ULL, ARM Cortex-A7\n");
    return 0;
}

void reset_cpu(void)
{
    struct imx6ull_watchdog_regs *wdog =
        (struct imx6ull_watchdog_regs *)IMX6ULL_WDOG1_BASE;
    /* WCR.SRS is active low; leave WDA high for an internal reset. */
    u16 value = WDOG_WCR_WDE | WDOG_WCR_WDA;

    /* Three writes are required by i.MX6 erratum ERR004346. */
    writew(value, &wdog->wcr);
    writew(value, &wdog->wcr);
    writew(value, &wdog->wcr);

    while (1)
        ;
}
```

Common U-Boot calls `arch_cpu_init()` from `common/board_f.c` before relocation. If the terminal shows the marker and then stops, we know all of these things already worked:

- The Boot ROM found our image.
- The IVT entry address was valid.
- The DCD completed well enough for the ROM to load U-Boot into DDR.
- Generic ARMv7 startup reached C code.
- The stack is usable.
- The CCM and UART register addresses are correct.
- The board’s integrated USB-to-TTL connection is working.

That line proves a limited executable path, not all 512 MiB of DDR. It can appear despite marginal timing or address aliasing elsewhere in the bank. The reset routine uses a 16-bit register access and the ERR004346 repeated-write workaround; `SRS=0` asserts the system reset, whereas `SRS=1` does nothing. Its clock gate is enabled explicitly. The early hook clears the three watchdogs’ separate power-down enable bits, with their bus clocks available under this image’s broad DCD gating policy, and chooses cold-reset routing through SRC. It does not disable an already enabled normal watchdog: this first ROM entry assumes no inherited active normal watchdog that needs servicing. A chainloaded image would need a different audited handoff. Reset routing and repeated board power-cycle behavior remain hardware checks.

24A.11 Add the board directory

The register layout belongs to the SoC. The oscillator, inherited clock state, UART wiring, and reset assumptions still belong to a boot contract. The board files now add a **conditional** DDR bank and the two early checkpoints.

Create `board/point-atom/imx6ull-mini/Kconfig`

```
if TARGET_IMX6ULL_POINT_ATOM_MINI

config SYS_BOARD
    default "imx6ull-mini"

config SYS_VENDOR
    default "point-atom"

config SYS_SOC
    default "imx6ull"

config SYS_CONFIG_NAME
    default "imx6ull_point_atom_mini"

endif
```

These strings tell the build system where the board objects and legacy configuration header live:

Setting	Result
`SYS_VENDOR="point-atom"`	First part of `board/point-atom/imx6ull-mini/`
`SYS_BOARD="imx6ull-mini"`	Last part of `board/point-atom/imx6ull-mini/`
`SYS_CONFIG_NAME="imx6ull_point_atom_mini"`	Includes `include/configs/imx6ull_point_atom_mini.h`
`SYS_SOC="imx6ull"`	Gives tools and generated configuration a readable SoC name

Create `board/point-atom/imx6ull-mini/Makefile`

```
# SPDX-License-Identifier: GPL-2.0+

obj-y += board.o
```

Create `board/point-atom/imx6ull-mini/board.c`

```
// SPDX-License-Identifier: GPL-2.0+
#include <asm/global_data.h>
#include <asm/arch/hardware.h>
#include <asm/arch/uart.h>
#include <linux/sizes.h>
#include <stdio.h>

DECLARE_GLOBAL_DATA_PTR;

int board_early_init_f(void)
{
    imx6ull_early_uart_puts("[imx6ull] board_early_init_f reached\n");
    return 0;
}

int dram_init(void)
{
    gd->ram_size = IMX6ULL_DDR_SIZE;
    return 0;
}

int dram_init_banksize(void)
{
    gd->bd->bi_dram[0].start = IMX6ULL_DDR_BASE;
    gd->bd->bi_dram[0].size = IMX6ULL_DDR_SIZE;
    return 0;
}

int board_init(void)
{
    gd->bd->bi_boot_params = IMX6ULL_DDR_BASE + 0x100;
    return 0;
}

int checkboard(void)
{
    puts("Board: Point Atom MINI, IMX6ULL teaching port\n");
    return 0;
}
```

`dram_init()` does not initialize or probe DDR. It assigns the bank size assumed by this teaching configuration. A printed `512 MiB` line therefore proves only that the assignment ran. Use this configuration only after the fitted geometry, DCD, and full-bank validation agree with that size; otherwise change `IMX6ULL_DDR_SIZE` and the DTS memory node together.

`dram_init_banksize()` fills bank 0 in the board-information structure. Linux and U-Boot commands use this bank description later.

Create `board/point-atom/imx6ull-mini/MAINTAINERS`

```
IMX6ULL POINT ATOM MINI
M:      Your Name <you@example.com>
S:      Maintained
F:      arch/arm/dts/imx6ull-from-scratch.dtsi
F:      arch/arm/dts/imx6ull-point-atom-mini-from-scratch*
F:      arch/arm/mach-imx6ull/
F:      board/point-atom/imx6ull-mini/
F:      configs/imx6ull_point_atom_mini_defconfig
F:      drivers/mmc/imx6ull_usdhc.c
F:      drivers/serial/serial_imx6ull.c
F:      drivers/timer/imx6ull_gpt_timer.c
F:      include/configs/imx6ull_point_atom_mini.h
```

Replace the name and email before sending a real patch. `MAINTAINERS` tells `get_maintainer.pl` who owns these files. It does not affect the binary.

24A.12 Turn the UART code into a U-Boot driver

The early UART proves the hardware works. U-Boot's command shell cannot call that private print function directly. The serial uclass expects a driver with `putc`, `getc`, `pending`, and `setbrg` operations.

Create `drivers/serial/serial_imx6ull.c`:

```
// SPDX-License-Identifier: GPL-2.0+
#include <dm.h>
#include <errno.h>
#include <serial.h>
#include <asm/io.h>
#include <asm/arch/clock.h>
#include <asm/arch/uart.h>
#include <linux/bitops.h>

#define URXD_RX_DATA                    0xff
#define USR2_TXDC                       BIT(3)
#define USR2_RDR                        BIT(0)
#define UTS_RXEMPTY                     BIT(5)
#define UTS_TXFULL                      BIT(4)

struct imx6ull_serial_plat {
    struct imx6ull_uart *uart;
};

static int imx6ull_serial_setbrg(struct udevice *dev, int baudrate)
{
    struct imx6ull_serial_plat *plat = dev_get_plat(dev);

    return imx6ull_uart_hw_init(plat->uart, imx6ull_get_uart_clock(), baudrate);
}

static int imx6ull_serial_probe(struct udevice *dev)
{
```

```
    struct imx6ull_serial_plat *plat = dev_get_plat(dev);

    return imx6ull_uart_hw_init(plat->uart, imx6ull_get_uart_clock(), 115200);
}

static int imx6ull_serial_putc(struct udevice *dev, const char ch)
{
    struct imx6ull_serial_plat *plat = dev_get_plat(dev);

    if (readl(&plat->uart->ts) & UTS_TXFULL)
        return -EAGAIN;

    writel(ch, &plat->uart->txd);
    return 0;
}

static int imx6ull_serial_getc(struct udevice *dev)
{
    struct imx6ull_serial_plat *plat = dev_get_plat(dev);

    if (readl(&plat->uart->ts) & UTS_RXEMPTY)
        return -EAGAIN;

    return readl(&plat->uart->rxd) & URXD_RX_DATA;
}

static int imx6ull_serial_pending(struct udevice *dev, bool input)
{
    struct imx6ull_serial_plat *plat = dev_get_plat(dev);
    u32 sr2 = readl(&plat->uart->sr2);

    if (input)
        return !!(sr2 & USR2_RDR);

    return !(sr2 & USR2_TXDC);
}

static int imx6ull_serial_of_to_plat(struct udevice *dev)
{
    struct imx6ull_serial_plat *plat = dev_get_plat(dev);
    fdt_addr_t address = dev_read_addr(dev);

    if (address == FDT_ADDR_T_NONE)
        return -EINVAL;

    plat->uart = (struct imx6ull_uart *)address;
    return 0;
}

static const struct dm_serial_ops imx6ull_serial_ops = {
    .putc = imx6ull_serial_putc,
    .pending = imx6ull_serial_pending,
    .getc = imx6ull_serial_getc,
    .setbrg = imx6ull_serial_setbrg,
};

static const struct udevice_id imx6ull_serial_ids[] = {
    { .compatible = "fsl,imx6ull-uart" },
    { }
```

```
};

U_BOOT_DRIVER(serial_imx6ull) = {
    .name = "serial_imx6ull",
    .id = UCLASS_SERIAL,
    .of_match = imx6ull_serial_ids,
    .of_to_plat = imx6ull_serial_of_to_plat,
    .plat_auto = sizeof(struct imx6ull_serial_plat),
    .probe = imx6ull_serial_probe,
    .ops = &imx6ull_serial_ops,
    .flags = DM_FLAG_PRE_RELOC,
};
```

The important control flow is:

```
imx6ull-point-atom-mini-from-scratch.dts
  compatible = "fsl,imx6ull-uart"
        |
        v
imx6ull_serial_ids[] matches the node
        |
        v
imx6ull_serial_of_to_plat() reads reg = <0x02020000 0x4000>
        |
        v
imx6ull_serial_probe() initializes that UART
        |
        v
common console code calls imx6ull_serial_putc() and getc()
```

Returning `-EAGAIN` is part of the serial driver contract. It means “the FIFO cannot accept or provide a byte yet.” The serial uclass retries. A driver-model operation must not spin forever inside `putc()` or `getc()`.

Edit `drivers/serial/Kconfig`

Search for `config MXC_UART` in the serial-driver menu. Insert this new entry immediately above it:

```
+config IMX6ULL_SERIAL
+   bool "IMX6ULL UART driver"
+   depends on DM_SERIAL && ARCH_IMX6ULL
+   help
+     Enable the i.MX-style UART used by the from-scratch IMX6ULL port.
```

Edit `drivers/serial/Makefile`

Add this line beside the other serial-driver object lines:

```
+obj-$(CONFIG_IMX6ULL_SERIAL) += serial_imx6ull.o
```

Kconfig decides whether the feature exists. The Makefile decides which object implements it. Enabling only one side does not work.

24A.13 Write the timer driver

Common U-Boot needs a monotonically increasing counter for delays, timeouts, and commands such as `sleep`. GPT1 can use the 24 MHz oscillator independently of the main CPU clock.

Its special 24 MHz prescaler is only four bits wide. Dividing by eight is valid, so we run the U-Boot timer at 3 MHz:

```
24,000,000 / (7 + 1) = 3,000,000 ticks per second
```

Create `drivers/timer/imx6ull_gpt_timer.c`:

```
// SPDX-License-Identifier: GPL-2.0+
#include <dm.h>
#include <errno.h>
#include <timer.h>
#include <asm/io.h>
#include <linux/bitops.h>

#define GPT_CR_EN                       BIT(0)
#define GPT_CR_FRR                      BIT(9)
#define GPT_CR_EN_24M                   BIT(10)
#define GPT_CR_SWR                      BIT(15)
#define GPT_CR_CLKSRC_24M               (5 << 6)

#define GPT_PR_PRESCALER24M_SHIFT       12
#define GPT_24M_PRESCALER               7
#define GPT_COUNTER_RATE                3000000U

struct imx6ull_gpt_regs {
    u32 cr;
    u32 pr;
    u32 sr;
    u32 ir;
    u32 ocr1;
    u32 ocr2;
    u32 ocr3;
    u32 icr1;
    u32 icr2;
    u32 cnt;
};

struct imx6ull_gpt_priv {
    struct imx6ull_gpt_regs *regs;
};

static u64 imx6ull_gpt_get_count(struct udevice *dev)
{
    struct imx6ull_gpt_priv *priv = dev_get_priv(dev);
```

```
    return timer_conv_64(readl(&priv->regs->cnt));
}

static int imx6ull_gpt_probe(struct udevice *dev)
{
    struct imx6ull_gpt_priv *priv = dev_get_priv(dev);
    struct timer_dev_priv *uc_priv = dev_get_uclass_priv(dev);
    fdt_addr_t address = dev_read_addr(dev);
    u32 budget = 100000;

    if (address == FDT_ADDR_T_NONE)
        return -EINVAL;

    priv->regs = (struct imx6ull_gpt_regs *)address;
    writel(0, &priv->regs->ir);
    writel(0x3f, &priv->regs->sr);
    uc_priv->clock_rate = GPT_COUNTER_RATE;
    /* Preserve the timebase when DM probes a new instance after relocation. */
    if (readl(&priv->regs->cr) ==
        (GPT_CR_CLKSRC_24M | GPT_CR_EN_24M | GPT_CR_FRR | GPT_CR_EN) &&
        readl(&priv->regs->pr) ==
        (GPT_24M_PRESCALER << GPT_PR_PRESCALER24M_SHIFT))
        return 0;

    setbits_le32(&priv->regs->cr, GPT_CR_SWR);
    while (readl(&priv->regs->cr) & GPT_CR_SWR) {
        if (!--budget)
            return -ETIMEDOUT;
    }

    writel(GPT_24M_PRESCALER << GPT_PR_PRESCALER24M_SHIFT,
           &priv->regs->pr);
    writel(GPT_CR_CLKSRC_24M | GPT_CR_EN_24M | GPT_CR_FRR,
           &priv->regs->cr);
    setbits_le32(&priv->regs->cr, GPT_CR_EN);

    uc_priv->clock_rate = GPT_COUNTER_RATE;
    return 0;
}

static const struct timer_ops imx6ull_gpt_ops = {
    .get_count = imx6ull_gpt_get_count,
};

static const struct udevice_id imx6ull_gpt_ids[] = {
    { .compatible = "fsl,imx6ull-gpt" },
    { }
};

U_BOOT_DRIVER(imx6ull_gpt_timer) = {
    .name = "imx6ull_gpt_timer",
    .id = UCLASS_TIMER,
    .of_match = imx6ull_gpt_ids,
    .probe = imx6ull_gpt_probe,
    .priv_auto = sizeof(struct imx6ull_gpt_priv),
    .ops = &imx6ull_gpt_ops,
    .flags = DM_FLAG_PRE_RELOC,
};
```

`GPT1` has a 32-bit counter. At 3 MHz it wraps after about 1,432 seconds. `timer_conv_64()` detects a wrap when a sampled count falls below the previous sample. It cannot reconstruct multiple wraps while nobody reads the timer. GPT reset also uses a finite poll budget: calling a timer-backed delay from inside this timer's own probe would create a circular dependency. The probe sets the uclass rate to the **counter** rate, not the oscillator rate. Marking this driver pre-relocation and keeping its DTS node available lets common code obtain time before relocation.

Edit `drivers/timer/Kconfig`

Search for `config IMX_GPT_TIMER`. Insert this entry immediately above it:

```
+config IMX6ULL_GPT_TIMER
+   bool "IMX6ULL GPT timer"
+   depends on TIMER && ARCH_IMX6ULL
+   help
+     Use GPT1 with its 24 MHz oscillator input as U-Boot's timer.
```

Edit `drivers/timer/Makefile`

Add this line beside the other timer-driver object lines:

```
+obj-$(CONFIG_IMX6ULL_GPT_TIMER) += imx6ull_gpt_timer.o
```

24A.14 Use the documented MINI core-board DCD

DDR is board-specific, but that does not require inventing an initializer. The official Alientek repository supplies one in [board/freescale/mx6ull_alientek_emmc/imximage.cfg](), at **commit `edb7ca5ac4be2d978be60a2a12c61e0b6d1f7feb`**. Its `mx6ull_alientek_emmc_defconfig` selects that file. We copy its seven clock-gate writes and all 53 DDR IOMUX/MMDC writes, preserving their values and order. Only the UART1 and USDHC2 pad/reset/daisy groups after the DDR block are additions for our teaching drivers. No vendor runtime driver or old configuration header is imported into v2026.04.

The supplied *I.MX6U Embedded Linux Driver Development Guide V1.81* marks both its DDR chapter 23 and U-Boot porting chapter 33 as MINI-compatible on page 7. Pages 531-533 and 537-543 identify the eMMC core's `NT5CC256M16EP-EK`: one x16, 4 Gbit device, 15 row bits, 10 column bits, eight internal banks, and one chip select. That is `2^15 * 2^10 * 8 * 2 = 536870912` bytes, or 512 MiB, matching the supplied core V2.0 schematic. The NAND core's 256 MiB configuration is not interchangeable with it.

The guide's DDR worksheet uses a nominal 400 MHz clock, with page 537 describing the normal ROM's 396 MHz PLL2/PFD2 path. Our DCD copies the vendor timing/geometry/calibration data without retuning that memory clock. Pages 545-550 explain calibration and stress testing, but their example results are **not** the constants in this pinned vendor DCD. For example, the vendor uses `MPDGCTRL0=0x41640158`, `MPRDDLCTL=0x40403237`, and

`MPWRDLCTL=0x40403C33`. Do not replace those with the guide's example calibration or an image-format fixture. Neither the guide nor this source comparison proves margins on the reader's individual board.

The source provenance is now complete. Keep its evidence separate from the physical acceptance record:

Evidence	What it establishes or still requires
Vendor commit and selected image file	Traceable ordered initializer, reproduced in full below.
Guide and core V2.0 schematic	Documented 512 MiB x16 eMMC-core design and MINI applicability, not an inspection of fitted silicon.
Core revision, fitted marking, and rails	Confirm the actual assembly matches before trying the image. A replacement part needs its own matching BSP or DDR work.
DDR clock and calibration record	Vendor values are reused, not newly measured. Record the inherited clock and validate the active byte lanes on hardware.
Independent memory validation	Cover aliasing, the whole usable bank, cold/warm starts, and relevant voltage/temperature conditions. A banner is insufficient.
Bank size	Keep `IMX6ULL_DDR_SIZE` and the DTS memory node consistent with the accepted geometry.

There is no separate DDR include to supply and no macro claiming qualification. Part II's DDR timing analysis explains the preserved write order, timing units, mode-register commands, refresh, ZQ, and calibration. A successful compile checks software contracts; `mkimage` checks syntax and layout. Neither evaluates electrical timing.

Create `board/point-atom/imx6ull-mini/imximage.cfg` with the complete content below:

```
/*
 * Copyright (C) 2016 Freescale Semiconductor, Inc.
 * SPDX-License-Identifier: GPL-2.0+
 * Clock/DDR data: Alientek edb7ca5ac4be2d978be60a2a12c61e0b6d1f7feb,
 * board/freescale/mx6ull_alientek_emmc/imximage.cfg (non-plugin DCD).
 * Documented 512 MiB x16 eMMC core, not our own bench calibration.
 */
```

```
IMAGE_VERSION 2
BOOT_FROM sd

/* Broad clock gating for initial bring-up, not a product clock policy. */
DATA 4 0x020C4068 0xFFFFFFFF
DATA 4 0x020C406C 0xFFFFFFFF
DATA 4 0x020C4070 0xFFFFFFFF
DATA 4 0x020C4074 0xFFFFFFFF
DATA 4 0x020C4078 0xFFFFFFFF
DATA 4 0x020C407C 0xFFFFFFFF
DATA 4 0x020C4080 0xFFFFFFFF

/* Vendor DDR block: preserve every address, value, and write order. */
DATA 4 0x020E04B4 0x000C0000
DATA 4 0x020E04AC 0x00000000
DATA 4 0x020E027C 0x00000030
DATA 4 0x020E0250 0x00000030
DATA 4 0x020E024C 0x00000030
DATA 4 0x020E0490 0x00000030
DATA 4 0x020E0288 0x000C0030
DATA 4 0x020E0270 0x00000000
DATA 4 0x020E0260 0x00000030
DATA 4 0x020E0264 0x00000030
DATA 4 0x020E04A0 0x00000030
DATA 4 0x020E0494 0x00020000
DATA 4 0x020E0280 0x00000030
DATA 4 0x020E0284 0x00000030
DATA 4 0x020E04B0 0x00020000
DATA 4 0x020E0498 0x00000030
DATA 4 0x020E04A4 0x00000030
DATA 4 0x020E0244 0x00000030
DATA 4 0x020E0248 0x00000030
DATA 4 0x021B001C 0x00008000
DATA 4 0x021B0800 0xA1390003
DATA 4 0x021B080C 0x00000004
DATA 4 0x021B083C 0x41640158
DATA 4 0x021B0848 0x40403237
DATA 4 0x021B0850 0x40403C33
DATA 4 0x021B081C 0x33333333
DATA 4 0x021B0820 0x33333333
DATA 4 0x021B082C 0xF3333333
DATA 4 0x021B0830 0xF3333333
DATA 4 0x021B08C0 0x00944009
DATA 4 0x021B08B8 0x00000800
DATA 4 0x021B0004 0x0002002D
DATA 4 0x021B0008 0x1B333030
DATA 4 0x021B000C 0x676B52F3
DATA 4 0x021B0010 0xB66D0B63
DATA 4 0x021B0014 0x01FF00DB
DATA 4 0x021B0018 0x00201740
DATA 4 0x021B001C 0x00008000
DATA 4 0x021B002C 0x000026D2
DATA 4 0x021B0030 0x006B1023
DATA 4 0x021B0040 0x0000004F
DATA 4 0x021B0000 0x84180000
DATA 4 0x021B0890 0x00400000
DATA 4 0x021B001C 0x02008032
DATA 4 0x021B001C 0x00008033
DATA 4 0x021B001C 0x00048031
```

```
DATA 4 0x021B001C 0x15208030
DATA 4 0x021B001C 0x04008040
DATA 4 0x021B0020 0x00000800
DATA 4 0x021B0818 0x00000227
DATA 4 0x021B0004 0x0002552D
DATA 4 0x021B0404 0x00011006
DATA 4 0x021B001C 0x00000000

/* UART1 on UART1_TX_DATA and UART1_RX_DATA pads. */
DATA 4 0x020E0084 0x00000000
DATA 4 0x020E0088 0x00000000
DATA 4 0x020E0310 0x000010B0
DATA 4 0x020E0314 0x000130B1
DATA 4 0x020E0624 0x00000003

/* USDHC2 eMMC mux registers. ALT1 selects USDHC2 on the NAND pads. */
DATA 4 0x020E0178 0x00000001
DATA 4 0x020E017C 0x00000001
DATA 4 0x020E0180 0x00000001
DATA 4 0x020E0184 0x00000001
DATA 4 0x020E0188 0x00000001
DATA 4 0x020E018C 0x00000001
DATA 4 0x020E0190 0x00000001
DATA 4 0x020E0194 0x00000001
DATA 4 0x020E0198 0x00000001
DATA 4 0x020E019C 0x00000001
/* NAND_ALE: GPIO4_IO10 (ALT5), hold eMMC RESET_N inactive. */
DATA 4 0x020E01A0 0x00000005

/* Candidate <=20 MHz 3.3 V pad settings; validate on the fitted board. */
/* CLK: no pull/keeper; CMD/DAT/RESET: pull-up, DSE=3, SPEED=1, SRE=1. */
DATA 4 0x020E0404 0x00000059
DATA 4 0x020E0408 0x00017059
DATA 4 0x020E040C 0x00017059
DATA 4 0x020E0410 0x00017059
DATA 4 0x020E0414 0x00017059
DATA 4 0x020E0418 0x00017059
DATA 4 0x020E041C 0x00017059
DATA 4 0x020E0420 0x00017059
DATA 4 0x020E0424 0x00017059
DATA 4 0x020E0428 0x00017059
DATA 4 0x020E042C 0x00017059

/* Preload GPIO4 DR high, then enable output; preserve other GPIO bits. */
SET_BIT 4 0x020A8000 0x00000400
SET_BIT 4 0x020A8004 0x00000400

/* USDHC2 input daisy selectors for clock, command, and data lines. */
DATA 4 0x020E0670 0x00000002
DATA 4 0x020E0678 0x00000002
DATA 4 0x020E067C 0x00000002
DATA 4 0x020E0680 0x00000002
DATA 4 0x020E0684 0x00000001
DATA 4 0x020E0688 0x00000002
DATA 4 0x020E068C 0x00000001
DATA 4 0x020E0690 0x00000001
DATA 4 0x020E0694 0x00000001
DATA 4 0x020E0698 0x00000001
```

Each `DATA 4 address value` line tells the Boot ROM to perform one 32-bit write. The ROM does not understand “DDR3L” as a high-level concept. It writes these values in this order, exactly as bare-metal C would do with `writel(value, address)`.

What each DCD group does

Group	Why it must happen before U-Boot starts
CCGR writes	Give the IOMUX and MMDC blocks working clocks during setup.
UART pad writes	Connect UART1 to the pads wired to the board’s built-in USB-to-TTL circuit.
USDHC2 pad writes	Select the schematic’s NAND-pad route; electrical/reset validation is still required.
DDR IOMUX writes	Set DDR signal voltage, drive strength, and pad behavior.
MMDC calibration writes	Apply the vendor’s stored read/write delay settings, not calibration measured by this example.
MMDC timing writes	Describe memory geometry, row timing, refresh, and controller behavior.
JEDEC commands	Reset and configure the DDR3L device itself.

The input daisy selector is a second mux on signals entering the SoC. The pad mux connects a physical pin to a peripheral function. The daisy register then tells the peripheral which possible input path to listen to. TX is an output, so UART1 TX needs no daisy value. UART1 RX is an input, so it needs both the pad mux and `IOMUXC_UART1_RX_DATA_SELECT_INPUT = 3`.

For USDHC2, `NAND_RE_B` selects CLK, `NAND_WE_B` selects CMD, and `NAND_DATA00..07` select DAT0..7 on ALT1. `NAND_ALE` could select the host RESET_B output on ALT1, but here it selects **GPIO4_IO10 on ALT5**. The supplied schematic has a 10 kOhm pull-down on eMMC RESET_N. A 22 kOhm internal pull-up alone would not establish an inactive high level against it, so the DCD preloads GPIO4’s output latch high, then enables that output. `SET_BIT` is a ROM read/modify/write command, not a full-register replacement. Do not pulse the line or change the card’s potentially one-time `RST_N_FUNCTION` field.

The offsets and daisy values agree with the pinned `mx6ul_pins.h` definitions. They establish routing, not signal integrity. The candidate `0x17059` enables hysteresis, a 22 kOhm pull-up, pull selection/enable, medium speed, drive code three, and fast slew. CLK instead uses `0x59` without a pull/keeper. Check the actual VCC, VCCQ, NVCC_NAND rails and external resistors; a pull-up register cannot change rail voltage. Confirm RESET_N is actually high and that power-on/reset recovery satisfies the fitted eMMC. The GPIO setting is an explicit boot prerequisite, not a tested reset sequence.

The same DCD operation written as C

The DCD is not magic firmware. The following helper expresses its operation in bare-metal C:

```
struct dcd_write {
    unsigned long address;
    unsigned int value;
};

static void apply_dcd(const struct dcd_write *table, unsigned int count)
{
    unsigned int i;

    for (i = 0; i < count; i++)
        writel(table[i].value, (void *)table[i].address);
}
```

This helper explains the `DATA` writes, not the `SET_BIT` read/modify/write commands, and is not another file to add. A real SPL initializer must also implement the required waits, checks, and sequencing. Here the ROM is the table executor, and the complete documented DCD above is the input to section 24A.21. Reusing that input makes image creation reproducible, not board qualification automatic.

24A.15 Describe the SoC and board with Device Tree

The Device Tree describes hardware instances and board wiring. It does not initialize DDR. U-Boot needs DDR before it can safely access the Device Tree appended to `u-boot.bin`.

Create `arch/arm/dts/imx6ull-from-scratch.dtsi`

```
/ {
    compatible = "fsl,imx6ull-from-scratch";
    #address-cells = <1>;
    #size-cells = <1>;

    soc {
        compatible = "simple-bus";
        #address-cells = <1>;
        #size-cells = <1>;
        ranges;
```

```
        uart1: serial@2020000 {
            compatible = "fsl,imx6ull-uart";
            reg = <0x02020000 0x4000>;
            status = "disabled";
            bootph-all;
        };

        gpt1: timer@2098000 {
            compatible = "fsl,imx6ull-gpt";
            reg = <0x02098000 0x4000>;
            status = "disabled";
            bootph-all;
        };

        usdhc2: mmc@2194000 {
            compatible = "fsl,imx6ull-usdhc";
            reg = <0x02194000 0x4000>;
            status = "disabled";
        };
    };
};
```

The `.dtsi` says that every IMX6ULL SoC has these controller instances. Their status is disabled because a board may not route them to usable pins.

`bootph-all` keeps UART1 and GPT1 available before and after relocation. Without it, U-Boot's Device Tree filtering can remove an early device from a reduced pre-relocation tree.

Create `arch/arm/dts/imx6ull-point-atom-mini-from-scratch.dts`

```
/dts-v1/;

#include "imx6ull-from-scratch.dtsi"

/ {
    model = "Point Atom MINI, IMX6ULL teaching port";
    compatible = "point-atom,imx6ull-mini", "fsl,imx6ull-from-scratch";

    aliases {
        serial0 = &uart1;
        mmc0 = &usdhc2;
    };

    chosen {
        stdout-path = "serial0:115200n8";
        tick-timer = &gpt1;
    };

    memory@80000000 {
        device_type = "memory";
        reg = <0x80000000 0x20000000>;
    };
};
```

```
&uart1 {
    status = "okay";
};

&gpt1 {
    status = "okay";
};

&usdhc2 {
    bus-width = <1>;
    max-frequency = <20000000>;
    non-removable;
    no-1-8-v;
    status = "okay";
};
```

The board file enables only devices that are physically connected. The `mmc0` alias gives USDHC2 device number 0 in U-Boot, so the command is `mmc dev 0`. The hardware base remains the real USDHC2 address, `0x02194000`.

Eight data lines are wired, but this first configuration advertises only one and caps the clock at 20 MHz. `no-1-8-v` is a capability restriction, not a voltage regulator: the board must actually supply compatible 3.3 V signaling rails. Our custom driver also has no 1.8 V capabilities or switching code. This teaching DT describes U-Boot's private drivers, not a DTB to pass unchanged to Linux.

Create `arch/arm/dts/imx6ull-point-atom-mini-from-scratch-u-boot.dtsi`

```
// SPDX-License-Identifier: GPL-2.0+

/ {
};
```

This intentionally empty file takes precedence over the fallback `imx6ull-u-boot.dtsi`. Our UART and timer nodes already contain `bootph-all`. Without the board-specific file, fallback include selection could introduce labels and assumptions from the normal i.MX tree into this independent teaching tree. Keep the explicit empty override so the DT source boundary is reproducible.

Edit `arch/arm/dts/Makefile`

Add this line beside the other 32-bit ARM DTB selections:

```
+dtb-$(CONFIG_ARCH_IMX6ULL) += imx6ull-point-atom-mini-from-scratch.dtb
```

This line tells U-Boot's build which DTB belongs to our architecture. The defconfig later chooses this exact filename as the default Device Tree.

24A.16 Write the USDHC2 eMMC driver from scratch

U-Boot’s MMC core knows the standard MMC and eMMC protocol. It knows that identification starts with CMD0, CMD1, CMD2, and CMD3. It does not know how the i.MX6ULL USDHC2 controller sends one of those commands. Our driver must provide that hardware layer.

The Boot ROM, not U-Boot, reads the first image from the removable SD card. By the time our code starts, the complete U-Boot payload is already in DDR. This first U-Boot driver controls USDHC2 for the on-board eMMC. Adding SD access later means enabling USDHC1 pads and clocks and creating a second Device Tree node for the same driver design.

The first driver uses programmed I/O, usually shortened to PIO. The CPU copies every 32-bit word between the USDHC FIFO and DDR. PIO is slower than DMA, but every transfer is visible and there are no cache-coherency or descriptor problems during first bring-up.

The listing implements these paths, which still need controller/card testing:

- Command transmission and 48-bit or 136-bit responses
- Commands that return a busy signal on DAT0
- Single-block and multi-block reads and writes
- 1-bit, 4-bit, and 8-bit bus widths
- Clock division for identification and legacy transfer speed
- Command, data, FIFO, and DAT0 timeouts
- Driver-model MMC binding

The default is **1-bit legacy eMMC, at most 20 MHz**, not 52 MHz high speed. The width register paths for four/eight bits are included so a later wiring check does not require a second driver, but they are not advertised by the initial DTS. No high-speed, DDR transfer, DMA, voltage switching, HS200, HS400, or tuning is advertised. This is also not a removable-SD driver: card detect, write protect, supplies, and the other host’s root clock are absent.

The USDHC registers used here

Offset	Register	What our driver uses it for
`0x04`	`BLKATTR`	Block size and number of blocks

Offset	Register	What our driver uses it for
0x08	CMDARG	Command argument
0x0C	XFERTYP	Command number, response type, and command checks
0x10 to 0x1C	CMDRSP0 to CMDRSP3	Response words returned by the card
0x20	DATPORT	PIO FIFO data
0x24	PRSSTAT	Command, data, FIFO, clock, and DAT0 state
0x28	PROCTL	Data bus width
0x2C	SYSCTL	Reset, timeout, and SD clock divisors
0x30	IRQSTAT	Command and data completion or error status
0x34	IRQSTATEN	Chooses which status bits the controller records
0x38	IRQSIGEN	Interrupt output enable, left zero because we poll
0x44	WML	FIFO read and write watermark
0x48	MIXCTRL	USDHC data-transfer flags
0xC0	VENDORSPEC	Internal, host, peripheral, and SD clock gates

Offset	Register	What our driver uses it for
0xC4	MMCBOOT	eMMC boot mode, cleared for normal commands

Create `drivers/mmc/imx6ull_usdhc.c`

```
// SPDX-License-Identifier: GPL-2.0+
#include <dm.h>
#include <errno.h>
#include <mmc.h>
#include <string.h>
#include <time.h>
#include <asm/io.h>
#include <asm/arch/clock.h>
#include <linux/bitops.h>
#include <linux/delay.h>

#define USDHC_BLKATTR                   0x04
#define USDHC_CMDARG                    0x08
#define USDHC_XFERTYP                   0x0c
#define USDHC_CMDRSP0                   0x10
#define USDHC_CMDRSP1                   0x14
#define USDHC_CMDRSP2                   0x18
#define USDHC_CMDRSP3                   0x1c
#define USDHC_DATPORT                   0x20
#define USDHC_PRSSTAT                   0x24
#define USDHC_PROCTL                    0x28
#define USDHC_SYSCTL                    0x2c
#define USDHC_IRQSTAT                   0x30
#define USDHC_IRQSTATEN                 0x34
#define USDHC_IRQSIGEN                  0x38
#define USDHC_WML                       0x44
#define USDHC_MIXCTRL                   0x48
#define USDHC_VENDORSPEC                0xc0
#define USDHC_MMCBOOT                   0xc4

#define PRSSTAT_DAT0                    BIT(24)
#define PRSSTAT_BREN                    BIT(11)
#define PRSSTAT_BWEN                    BIT(10)
#define PRSSTAT_SDSTB                   BIT(3)
#define PRSSTAT_DLA                     BIT(2)
#define PRSSTAT_CICHB                   BIT(1)
#define PRSSTAT_CIDHB                   BIT(0)

#define PROCTL_DTW_MASK                 (0x3 << 1)
#define PROCTL_DTW_4                    BIT(1)
#define PROCTL_DTW_8                    BIT(2)
#define PROCTL_INIT                     BIT(5)

#define SYSCTL_CLOCK_MASK               0x0000fff0
#define SYSCTL_TIMEOUT_MASK             0x000f0000
#define SYSCTL_TIMEOUT_MAX              (14 << 16)
#define SYSCTL_RSTA                     BIT(24)
```

```
#define SYSCTL_RSTC                     BIT(25)
#define SYSCTL_RSTD                     BIT(26)
#define SYSCTL_RSTT                     BIT(28)

#define IRQSTAT_CC                      BIT(0)
#define IRQSTAT_TC                      BIT(1)
#define IRQSTAT_BWR                     BIT(4)
#define IRQSTAT_BRR                     BIT(5)
#define IRQSTAT_CTOE                    BIT(16)
#define IRQSTAT_CCE                     BIT(17)
#define IRQSTAT_CEBE                    BIT(18)
#define IRQSTAT_CIE                     BIT(19)
#define IRQSTAT_DTOE                    BIT(20)
#define IRQSTAT_DCE                     BIT(21)
#define IRQSTAT_DEBE                    BIT(22)

#define IRQSTAT_CMD_ERROR               (IRQSTAT_CTOE | IRQSTAT_CCE | \
                      IRQSTAT_CEBE | IRQSTAT_CIE)
#define IRQSTAT_DATA_ERROR              (IRQSTAT_DTOE | IRQSTAT_DCE | \
                      IRQSTAT_DEBE)
#define IRQSTAT_USED                    (IRQSTAT_CC | IRQSTAT_TC | \
                      IRQSTAT_BWR | IRQSTAT_BRR | \
                      IRQSTAT_CMD_ERROR | \
                      IRQSTAT_DATA_ERROR)

#define XFERTYP_CMD(index)              (((index) & 0x3f) << 24)
#define XFERTYP_CMDTYP_ABORT            (0x3 << 22)
#define XFERTYP_DPSEL                   BIT(21)
#define XFERTYP_CICEN                   BIT(20)
#define XFERTYP_CCCEN                   BIT(19)
#define XFERTYP_RSPTYP_136              BIT(16)
#define XFERTYP_RSPTYP_48               BIT(17)
#define XFERTYP_RSPTYP_48_BUSY          (0x3 << 16)
#define XFERTYP_MSBSEL                  BIT(5)
#define XFERTYP_DTDSEL_READ             BIT(4)
#define XFERTYP_BCEN                    BIT(1)
#define MIXCTRL_TRANSFER_MASK           0x7f

#define VENDORSPEC_INIT                 0x20007809
#define VENDORSPEC_IPGEN                BIT(11)
#define VENDORSPEC_HCKEN                BIT(12)
#define VENDORSPEC_PEREN                BIT(13)
#define VENDORSPEC_CKEN                 BIT(14)
#define VENDORSPEC_FRC_SDCLK_ON         BIT(8)

#define WML_READ_ONE_WORD               1
#define WML_WRITE_ONE_WORD              (1 << 16)

#define COMMAND_TIMEOUT_MS              1000
#define DATA_TIMEOUT_MS                 5000

struct imx6ull_usdhc_plat {
    fdt_addr_t address;
    u32 bus_width;
    bool non_removable;
    struct mmc_config cfg;
    struct mmc mmc;
};
```

```
struct imx6ull_usdhc_priv {
    u8 __iomem *base;
    u32 input_clock;
};

static void __iomem *usdhc_reg(struct imx6ull_usdhc_priv *priv,
                   u32 offset)
{
    return priv->base + offset;
}

static int usdhc_wait_mask(struct imx6ull_usdhc_priv *priv, u32 offset,
               u32 mask, bool want_set, ulong timeout_ms)
{
    ulong start = get_timer(0);

    while (!!(readl(usdhc_reg(priv, offset)) & mask) != want_set) {
        if (get_timer(start) >= timeout_ms)
            return -ETIMEDOUT;
    }

    return 0;
}

static int usdhc_wait_irq(struct imx6ull_usdhc_priv *priv, u32 events,
              ulong start, ulong timeout_ms)
{
    while (!(readl(usdhc_reg(priv, USDHC_IRQSTAT)) & events)) {
        if (get_timer(start) >= timeout_ms)
            return -ETIMEDOUT;
    }

    return 0;
}

static int usdhc_reset_lines(struct imx6ull_usdhc_priv *priv, bool data)
{
    u32 mask = SYSCTL_RSTC;

    if (data)
        mask |= SYSCTL_RSTD;

    setbits_le32(usdhc_reg(priv, USDHC_SYSCTL), mask);
    return usdhc_wait_mask(priv, USDHC_SYSCTL, mask, false, 100);
}

static u32 usdhc_build_xfertyp(struct mmc_cmd *cmd,
                   struct mmc_data *data)
{
    u32 value = XFERTYP_CMD(cmd->cmdidx);

    if (cmd->resp_type & MMC_RSP_CRC)
        value |= XFERTYP_CCCEN;
    if (cmd->resp_type & MMC_RSP_OPCODE)
        value |= XFERTYP_CICEN;

    if (cmd->resp_type & MMC_RSP_136)
        value |= XFERTYP_RSPTYP_136;
    else if (cmd->resp_type & MMC_RSP_BUSY)
```

```
        value |= XFERTYP_RSPTYP_48_BUSY;
    else if (cmd->resp_type & MMC_RSP_PRESENT)
        value |= XFERTYP_RSPTYP_48;

    if (cmd->cmdidx == MMC_CMD_STOP_TRANSMISSION)
        value |= XFERTYP_CMDTYP_ABORT;

    if (data) {
        value |= XFERTYP_DPSEL;
        if (data->flags & MMC_DATA_READ)
            value |= XFERTYP_DTDSEL_READ;
        if (data->blocks > 1)
            value |= XFERTYP_MSBSEL | XFERTYP_BCEN;
    }

    return value;
}

static int usdhc_data_error(u32 status)
{
    if (status & IRQSTAT_DTOE)
        return -ETIMEDOUT;
    if (status & IRQSTAT_DCE)
        return -EILSEQ;
    if (status & IRQSTAT_DEBE)
        return -EIO;
    return 0;
}

static int usdhc_wait_fifo(struct imx6ull_usdhc_priv *priv, u32 ready,
               ulong start)
{
    for (;;) {
        u32 status = readl(usdhc_reg(priv, USDHC_IRQSTAT));
        int ret = usdhc_data_error(status);

        if (ret)
            return ret;
        if (get_timer(start) >= DATA_TIMEOUT_MS)
            return -ETIMEDOUT;
        if (readl(usdhc_reg(priv, USDHC_PRSSTAT)) & ready)
            return 0;
    }
}

static int usdhc_transfer_pio(struct imx6ull_usdhc_priv *priv,
                  struct mmc_data *data, ulong start)
{
    u32 bytes = data->blocks * data->blocksize;
    u32 words = bytes / sizeof(u32);
    u32 i;
    int ret;

    if (data->flags & MMC_DATA_READ) {
        char *destination = data->dest;

        for (i = 0; i < words; i++) {
            u32 word;
```

```
            ret = usdhc_wait_fifo(priv, PRSSTAT_BREN, start);
            if (ret)
                return ret;
            word = readl(usdhc_reg(priv, USDHC_DATPORT));
            memcpy(destination + i * sizeof(word), &word, sizeof(word));
        }
    } else {
        const char *source = data->src;

        for (i = 0; i < words; i++) {
            u32 word;

            ret = usdhc_wait_fifo(priv, PRSSTAT_BWEN, start);
            if (ret)
                return ret;
            memcpy(&word, source + i * sizeof(word), sizeof(word));
            writel(word, usdhc_reg(priv, USDHC_DATPORT));
        }
    }

    return 0;
}

static void usdhc_read_response(struct imx6ull_usdhc_priv *priv,
                struct mmc_cmd *cmd)
{
    if (cmd->resp_type & MMC_RSP_136) {
        u32 response3 = readl(usdhc_reg(priv, USDHC_CMDRSP3));
        u32 response2 = readl(usdhc_reg(priv, USDHC_CMDRSP2));
        u32 response1 = readl(usdhc_reg(priv, USDHC_CMDRSP1));
        u32 response0 = readl(usdhc_reg(priv, USDHC_CMDRSP0));

        cmd->response[0] = (response3 << 8) | (response2 >> 24);
        cmd->response[1] = (response2 << 8) | (response1 >> 24);
        cmd->response[2] = (response1 << 8) | (response0 >> 24);
        cmd->response[3] = response0 << 8;
    } else if (cmd->resp_type & MMC_RSP_PRESENT) {
        cmd->response[0] = readl(usdhc_reg(priv, USDHC_CMDRSP0));
    }
}

static int imx6ull_usdhc_wait_dat0(struct udevice *dev, int state,
                   int timeout_us)
{
    struct imx6ull_usdhc_priv *priv = dev_get_priv(dev);
    u32 saved = readl(usdhc_reg(priv, USDHC_VENDORSPEC));
    int ret = -ETIMEDOUT;

    if (timeout_us < 0)
        return -EINVAL;
    setbits_le32(usdhc_reg(priv, USDHC_VENDORSPEC),
             VENDORSPEC_FRC_SDCLK_ON);
    for (;;) {
        ret = usdhc_data_error(readl(usdhc_reg(priv, USDHC_IRQSTAT)));
        if (ret)
            break;
        if (!!(readl(usdhc_reg(priv, USDHC_PRSSTAT)) & PRSSTAT_DAT0) ==
            !!state)
            break;
```

```
        if (!timeout_us--) {
            ret = -ETIMEDOUT;
            break;
        }
        udelay(1);
    }
    if (!(saved & VENDORSPEC_FRC_SDCLK_ON))
        clrbits_le32(usdhc_reg(priv, USDHC_VENDORSPEC),
                 VENDORSPEC_FRC_SDCLK_ON);
    return ret;
}

static int imx6ull_usdhc_send_cmd(struct udevice *dev,
                  struct mmc_cmd *cmd,
                  struct mmc_data *data)
{
    struct imx6ull_usdhc_priv *priv = dev_get_priv(dev);
    u32 inhibit = PRSSTAT_CIDHB;
    u32 transfer_type;
    u32 status;
    ulong data_start;
    int ret;

    if (cmd->cmdidx != MMC_CMD_STOP_TRANSMISSION)
        inhibit |= PRSSTAT_CICHB | PRSSTAT_DLA;

    /* Reject unsupported lengths/directions before issuing a command. */
    if (data && (!data->blocks || data->blocks > 128 ||
             !data->blocksize || data->blocksize > 512 ||
             data->blocksize % sizeof(u32) ||
             (data->flags != MMC_DATA_READ && data->flags != MMC_DATA_WRITE)))
        return -EINVAL;

    ret = usdhc_wait_mask(priv, USDHC_PRSSTAT, inhibit, false,
                  COMMAND_TIMEOUT_MS);
    if (ret)
        goto error;

    writel(0xffffffff, usdhc_reg(priv, USDHC_IRQSTAT));

    if (data) {
        writel((data->blocks << 16) | data->blocksize,
               usdhc_reg(priv, USDHC_BLKATTR));
        clrsetbits_le32(usdhc_reg(priv, USDHC_SYSCTL),
                  SYSCTL_TIMEOUT_MASK, SYSCTL_TIMEOUT_MAX);
    }

    transfer_type = usdhc_build_xfertyp(cmd, data);
    clrsetbits_le32(usdhc_reg(priv, USDHC_MIXCTRL),
              MIXCTRL_TRANSFER_MASK,
              transfer_type & MIXCTRL_TRANSFER_MASK);

    writel(cmd->cmdarg, usdhc_reg(priv, USDHC_CMDARG));
    writel(transfer_type & 0xffff0000,
           usdhc_reg(priv, USDHC_XFERTYP));

    ret = usdhc_wait_irq(priv, IRQSTAT_CC | IRQSTAT_CMD_ERROR,
                 get_timer(0), COMMAND_TIMEOUT_MS);
    if (ret)
```

```
        goto error;

    status = readl(usdhc_reg(priv, USDHC_IRQSTAT));
    if (status & IRQSTAT_CTOE) {
        ret = -ETIMEDOUT;
        goto error;
    }
    if (status & IRQSTAT_CCE) {
        ret = -EILSEQ;
        goto error;
    }
    if (status & (IRQSTAT_CEBE | IRQSTAT_CIE)) {
        ret = -EIO;
        goto error;
    }

    usdhc_read_response(priv, cmd);

    if (!data && (cmd->resp_type & MMC_RSP_BUSY)) {
        ret = imx6ull_usdhc_wait_dat0(dev, 1, DATA_TIMEOUT_MS * 1000);
        if (ret)
            goto error;
    }

    if (data) {
        data_start = get_timer(0);
        ret = usdhc_transfer_pio(priv, data, data_start);
        if (ret)
            goto error;

        ret = usdhc_wait_irq(priv, IRQSTAT_TC | IRQSTAT_DATA_ERROR,
                     data_start, DATA_TIMEOUT_MS);
        if (ret)
            goto error;

        status = readl(usdhc_reg(priv, USDHC_IRQSTAT));
        ret = usdhc_data_error(status);
        if (ret)
            goto error;
    }

    writel(0xffffffff, usdhc_reg(priv, USDHC_IRQSTAT));
    return 0;

error:
    /* A failed busy response also leaves the data state machine active. */
    if (usdhc_reset_lines(priv, data != NULL || (cmd->resp_type & MMC_RSP_BUSY)))
        ret = -ETIMEDOUT;
    writel(0xffffffff, usdhc_reg(priv, USDHC_IRQSTAT));
    return ret;
}

static int usdhc_set_clock(struct imx6ull_usdhc_priv *priv,
                struct mmc *mmc, u32 requested)
{
    u32 pre_divider = 1;
    u32 divider = 1;
    u32 encoded;
    int ret;
```

```
    if (!requested) {
        clrbits_le32(usdhc_reg(priv, USDHC_VENDORSPEC),
                 VENDORSPEC_CKEN);
        mmc->clock = 0;
        return 0;
    }

    while (priv->input_clock / (16 * pre_divider) > requested &&
           pre_divider < 256)
        pre_divider *= 2;

    while (priv->input_clock / (pre_divider * divider) > requested &&
           divider < 16)
        divider++;

    mmc->clock = priv->input_clock / pre_divider / divider;
    encoded = ((pre_divider >> 1) << 8) | ((divider - 1) << 4);

    clrbits_le32(usdhc_reg(priv, USDHC_VENDORSPEC), VENDORSPEC_CKEN);
    clrsetbits_le32(usdhc_reg(priv, USDHC_SYSCTL),
              SYSCTL_CLOCK_MASK, encoded);

    ret = usdhc_wait_mask(priv, USDHC_PRSSTAT, PRSSTAT_SDSTB,
                  true, 100);
    if (ret)
        return ret;

    setbits_le32(usdhc_reg(priv, USDHC_VENDORSPEC),
             VENDORSPEC_PEREN | VENDORSPEC_CKEN);
    return 0;
}

static int imx6ull_usdhc_set_ios(struct udevice *dev)
{
    struct imx6ull_usdhc_plat *plat = dev_get_plat(dev);
    struct imx6ull_usdhc_priv *priv = dev_get_priv(dev);
    struct mmc *mmc = &plat->mmc;
    u32 width;
    int ret;

    if (mmc->ddr_mode || mmc->clock > mmc->cfg->f_max ||
        (mmc->signal_voltage != MMC_SIGNAL_VOLTAGE_000 &&
         mmc->signal_voltage != MMC_SIGNAL_VOLTAGE_330))
        return -EINVAL;
    if (mmc->bus_width > plat->bus_width)
        return -EINVAL;
    ret = usdhc_set_clock(priv, mmc, mmc->clk_disable ? 0 : mmc->clock);
    if (ret)
        return ret;

    switch (mmc->bus_width) {
    case 1:
        width = 0;
        break;
    case 4:
        width = PROCTL_DTW_4;
        break;
    case 8:
```

```
        width = PROCTL_DTW_8;
        break;
    default:
        return -EINVAL;
    }

    clrsetbits_le32(usdhc_reg(priv, USDHC_PROCTL),
              PROCTL_DTW_MASK, width);
    return 0;
}

static int imx6ull_usdhc_get_cd(struct udevice *dev)
{
    struct imx6ull_usdhc_plat *plat = dev_get_plat(dev);

    return plat->non_removable ? 1 : 0;
}

static int imx6ull_usdhc_get_wp(struct udevice *dev)
{
    /* The soldered eMMC has no mechanical write-protect switch. */
    (void)dev;
    return 0;
}

static int imx6ull_usdhc_hw_init(struct imx6ull_usdhc_priv *priv,
                 struct mmc *mmc)
{
    int ret;

    setbits_le32(usdhc_reg(priv, USDHC_SYSCTL),
             SYSCTL_RSTA | SYSCTL_RSTT);
    ret = usdhc_wait_mask(priv, USDHC_SYSCTL,
                  SYSCTL_RSTA | SYSCTL_RSTT, false, 100);
    if (ret)
        return ret;

    writel(0, usdhc_reg(priv, USDHC_MMCBOOT));
    writel(0, usdhc_reg(priv, USDHC_MIXCTRL));
    writel(0, usdhc_reg(priv, 0x60)); /* DLL_CTRL */
    writel(0, usdhc_reg(priv, 0x68)); /* CLK_TUNE_CTRL_STATUS */
    writel(VENDORSPEC_INIT, usdhc_reg(priv, USDHC_VENDORSPEC));
    setbits_le32(usdhc_reg(priv, USDHC_VENDORSPEC),
             VENDORSPEC_HCKEN | VENDORSPEC_IPGEN);
    writel(PROCTL_INIT, usdhc_reg(priv, USDHC_PROCTL));
    writel(SYSCTL_TIMEOUT_MAX, usdhc_reg(priv, USDHC_SYSCTL));
    writel(IRQSTAT_USED, usdhc_reg(priv, USDHC_IRQSTATEN));
    writel(0, usdhc_reg(priv, USDHC_IRQSIGEN));
    writel(0xffffffff, usdhc_reg(priv, USDHC_IRQSTAT));
    writel(WML_READ_ONE_WORD | WML_WRITE_ONE_WORD,
           usdhc_reg(priv, USDHC_WML));

    ret = usdhc_set_clock(priv, mmc, 400000);
    if (ret)
        return ret;

    setbits_le32(usdhc_reg(priv, USDHC_VENDORSPEC),
             VENDORSPEC_FRC_SDCLK_ON);
    udelay(1000);
```

```
    clrbits_le32(usdhc_reg(priv, USDHC_VENDORSPEC),
             VENDORSPEC_FRC_SDCLK_ON);

    return 0;
}

static int imx6ull_usdhc_of_to_plat(struct udevice *dev)
{
    struct imx6ull_usdhc_plat *plat = dev_get_plat(dev);

    plat->address = dev_read_addr(dev);
    if (plat->address == FDT_ADDR_T_NONE)
        return -EINVAL;

    plat->bus_width = dev_read_u32_default(dev, "bus-width", 1);
    plat->non_removable = dev_read_bool(dev, "non-removable");
    if (!plat->non_removable || (plat->bus_width != 1 &&
        plat->bus_width != 4 && plat->bus_width != 8))
        return -EINVAL;
    return 0;
}

static int imx6ull_usdhc_probe(struct udevice *dev)
{
    struct mmc_uclass_priv *uclass = dev_get_uclass_priv(dev);
    struct imx6ull_usdhc_plat *plat = dev_get_plat(dev);
    struct imx6ull_usdhc_priv *priv = dev_get_priv(dev);
    struct mmc *mmc = &plat->mmc;
    struct blk_desc *block;
    int ret;

    priv->base = (u8 __iomem *)plat->address;
    priv->input_clock = imx6ull_get_usdhc2_clock();
    if (!priv->input_clock)
        return -EINVAL;

    plat->cfg.name = "i.MX6ULL USDHC2 PIO";
    plat->cfg.voltages = MMC_VDD_32_33 | MMC_VDD_33_34;
    plat->cfg.host_caps = MMC_MODE_1BIT | MMC_CAP_NONREMOVABLE;
    if (plat->bus_width >= 4)
        plat->cfg.host_caps |= MMC_MODE_4BIT;
    if (plat->bus_width >= 8)
        plat->cfg.host_caps |= MMC_MODE_8BIT;
    plat->cfg.f_min = 400000;
    plat->cfg.f_max = dev_read_u32_default(dev, "max-frequency", 20000000);
    if (plat->cfg.f_max < plat->cfg.f_min || plat->cfg.f_max > 20000000)
        return -EINVAL;
    plat->cfg.b_max = 128;

    mmc->cfg = &plat->cfg;
    mmc->dev = dev;
    uclass->mmc = mmc;
    block = mmc_get_blk_desc(mmc);
    if (block && plat->non_removable)
        block->removable = 0;

    ret = imx6ull_usdhc_hw_init(priv, mmc);
    if (ret)
        return ret;
```

```
    return 0;
}

static int imx6ull_usdhc_bind(struct udevice *dev)
{
    struct imx6ull_usdhc_plat *plat = dev_get_plat(dev);

    return mmc_bind(dev, &plat->mmc, &plat->cfg);
}

static int imx6ull_usdhc_reinit(struct udevice *dev)
{
    struct imx6ull_usdhc_plat *plat = dev_get_plat(dev);
    struct imx6ull_usdhc_priv *priv = dev_get_priv(dev);

    return imx6ull_usdhc_hw_init(priv, &plat->mmc);
}

static const struct dm_mmc_ops imx6ull_usdhc_ops = {
    .get_cd = imx6ull_usdhc_get_cd,
    .get_wp = imx6ull_usdhc_get_wp,
    .send_cmd = imx6ull_usdhc_send_cmd,
    .set_ios = imx6ull_usdhc_set_ios,
    .wait_dat0 = imx6ull_usdhc_wait_dat0,
    .reinit = imx6ull_usdhc_reinit,
};

static const struct udevice_id imx6ull_usdhc_ids[] = {
    { .compatible = "fsl,imx6ull-usdhc" },
    { }
};

U_BOOT_DRIVER(imx6ull_usdhc) = {
    .name = "imx6ull_usdhc",
    .id = UCLASS_MMC,
    .of_match = imx6ull_usdhc_ids,
    .of_to_plat = imx6ull_usdhc_of_to_plat,
    .bind = imx6ull_usdhc_bind,
    .probe = imx6ull_usdhc_probe,
    .ops = &imx6ull_usdhc_ops,
    .priv_auto = sizeof(struct imx6ull_usdhc_priv),
    .plat_auto = sizeof(struct imx6ull_usdhc_plat),
};
```

Edit `drivers/mmc/Kconfig`

Insert this entry immediately above `config FSL_ESDHC_IMX` in the MMC menu:

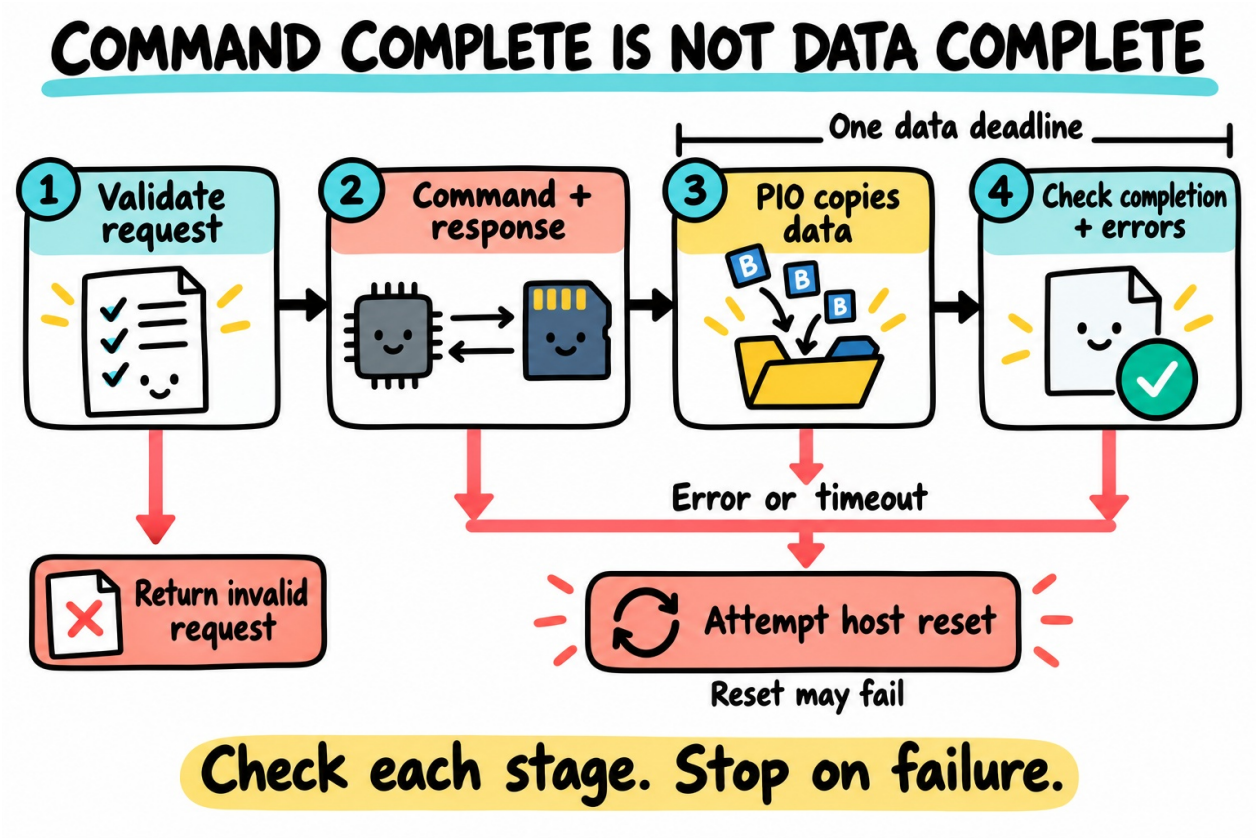
```
config IMX6ULL_USDHC
    bool "i.MX6ULL USDHC PIO driver"
    depends on ARCH_IMX6ULL && DM_MMC
    help
      Build the polling USDHC driver written by the from-scratch i.MX6ULL
      tutorial. It supports eMMC identification and PIO block transfers.
```

Edit `drivers/mmc/Makefile`

Add:

```
+obj-$(CONFIG_IMX6ULL_USDHC) += imx6ull_usdhc.o
```

Follow one block read through the code



Command completion is not data completion. The driver copies FIFO words, checks completion and returns any error. Host reset is a separate recovery attempt, not proof of card-write integrity. Card and buffer icons are conceptual.

Step	Function	What happens
1	MMC core	Creates CMD17 or CMD18 and a destination buffer.
2	`imx6ull_usdhc_send_cmd()`	Validates the request, waits for the command/data path, clears stale status, and sets block attributes.

Step	Function	What happens
3	`usdhc_build_xfertyp()` and caller	Encodes the command and transfer flags; the caller sets `MIXCTRL`, writes `CMDARG`, then launches with `XFERTYP`.
4	USDHC2 hardware	Sends the command to the eMMC and records completion in `IRQSTAT`.
5	`usdhc_read_response()`	Copies the controller response registers into `cmd->response[]`.
6	`usdhc_transfer_pio()`	Waits for `BREN` and copies each FIFO word into the destination buffer.
7	Host driver	Requires transfer complete and checks data timeout, CRC, and end-bit errors.
8	MMC core	Handles the protocol's stop/status sequence and exposes the completed block read.

The protocol decisions remain in U-Boot's MMC core. The i.MX6ULL register work is entirely in the driver shown above.

Read the driver as a state machine. First validate the request, then wait for the command path to become available. Clear old write-one-to-clear status bits, program block attributes and transfer mode, and write `XFERTYP` last to launch the command. Command completion is not data completion. For an R1b response without a buffer, DAT0 must release while card clock runs; for a buffered command, every FIFO word and the final `TC` must complete within one data budget.

The controller's `CICHB` flag inhibits the data path; `CIDHB` inhibits the command path. CMD12 must still be issuable when the data path is active, so it waits only for the command inhibit. This driver does not enable Auto CMD12; the MMC core sends the stop where its protocol requires it. R2 responses are shifted/reassembled because the controller omits the final CRC/end byte. CRC checking remains enabled only for response types that actually contain CRC: CMD1's R3 response is not one of them.

`IRQSTATEN` enables status latching even though `IRQSIGEN=0` disables the interrupt output. Watermarks are one word, matching the one-word polling loop. A FIFO-ready bit does not override a latched error. Data CRC returns `-EILSEQ`, timeout returns `-ETIMEDOUT`, and other transfer framing errors return `-EIO`. The same deadline covers all FIFO words and final completion; restarting a five-second timeout for each word could otherwise stretch a stalled request enormously. `memcpy()` avoids imposing alignment on a caller's buffer while the MMIO FIFO accesses stay 32-bit.

On failure the driver resets command state and, for buffered or busy commands, data state. A reset failure is returned, not discarded. This is host recovery only: it does not prove that a partially written card block is intact. `reinit()` resets the host for an explicit rescan, and `set_ios()` respects `mmc->clk_disable`. No board/card execution was performed to validate these paths. The first configuration disables MMC writes at the core, even though the host listing shows the write FIFO path for study.

24A.17 Add the small legacy configuration header

Create `include/configs/imx6ull_point_atom_mini.h`:

```
/* SPDX-License-Identifier: GPL-2.0+ */
#ifndef __IMX6ULL_POINT_ATOM_MINI_CONFIG_H
#define __IMX6ULL_POINT_ATOM_MINI_CONFIG_H

#include <asm/arch/hardware.h>

#define CFG_SYS_SDRAM_BASE              IMX6ULL_DDR_BASE

#define CFG_SYS_INIT_RAM_ADDR           IMX6ULL_OCRAM_BASE
#define CFG_SYS_INIT_RAM_SIZE           IMX6ULL_OCRAM_SIZE

#define CFG_EXTRA_ENV_SETTINGS \
    "kernel_addr_r=0x82000000\0" \
    "fdt_addr_r=0x83000000\0" \
    "console=ttymxc0,115200\0"

#endif
```

Most feature settings belong in Kconfig, not in this header. Three old-style `CFG_*` values are still needed here:

Macro	Use
`CFG_SYS_SDRAM_BASE`	Lowest usable DDR address
`CFG_SYS_INIT_RAM_ADDR`	Start of internal RAM used before relocation

Macro	Use
`CFG_SYS_INIT_RAM_SIZE`	Lets generic ARM code place the early stack near the top of OCRAM

The initial stack address is calculated by common U-Boot as:

```
OCRAM base + OCRAM size - generated global-data size
```

We do not choose an unexplained stack constant.

24A.18 Create the complete defconfig

Create `configs/imx6ull_point_atom_mini_defconfig`:

```
CONFIG_ARM=y
CONFIG_ARCH_IMX6ULL=y
CONFIG_TARGET_IMX6ULL_POINT_ATOM_MINI=y
CONFIG_TEXT_BASE=0x87800000
CONFIG_SYS_MALLOC_LEN=0x01000000
CONFIG_NR_DRAM_BANKS=1
CONFIG_SYS_MALLOC_F_LEN=0x4000
CONFIG_DEFAULT_DEVICE_TREE="imx6ull-point-atom-mini-from-scratch"
CONFIG_OF_CONTROL=y
CONFIG_OF_SEPARATE=y
# CONFIG_OF_UPSTREAM is not set
# CONFIG_CLK is not set
# CONFIG_PINCTRL is not set
CONFIG_SYS_ICACHE_OFF=y
CONFIG_SYS_DCACHE_OFF=y
CONFIG_BAUDRATE=115200
CONFIG_BOOTDELAY=3
CONFIG_USE_BOOTCOMMAND=y
CONFIG_BOOTCOMMAND="echo U-Boot is ready;"
CONFIG_SYS_PROMPT="imx6ull=> "
CONFIG_SYS_PBSIZE=512
CONFIG_DISPLAY_CPUINFO=y
CONFIG_DISPLAY_BOARDINFO=y
CONFIG_HUSH_PARSER=y
CONFIG_SYS_MAXARGS=32
CONFIG_SYS_LOAD_ADDR=0x82000000
CONFIG_SYS_MEMTEST_START=0x81000000
CONFIG_SYS_MEMTEST_END=0x81800000
CONFIG_CMD_MEMORY=y
CONFIG_CMD_MEMTEST=y
CONFIG_CMD_MMC=y
CONFIG_CMD_FAT=y
CONFIG_CMD_EXT4=y
CONFIG_CMD_FS_GENERIC=y
CONFIG_CMD_DM=y
CONFIG_CMD_SLEEP=y
```

```
CONFIG_ENV_IS_NOWHERE=y
CONFIG_ENV_SIZE=0x2000
CONFIG_NO_NET=y
# CONFIG_NET is not set
# CONFIG_EFI_LOADER is not set
# CONFIG_TOOLS_MKEFICAPSULE is not set
CONFIG_DM_SERIAL=y
CONFIG_IMX6ULL_SERIAL=y
CONFIG_TIMER=y
CONFIG_IMX6ULL_GPT_TIMER=y
CONFIG_MMC=y
CONFIG_DM_MMC=y
# CONFIG_MMC_WRITE is not set
CONFIG_IMX6ULL_USDHC=y
```

What every configuration line does

Configuration	Why it is enabled or assigned
`ARM`	Builds the ARM architecture instead of another U-Boot architecture.
`ARCH_IMX6ULL`	Adds our machine directory and enables our platform Kconfig.
`TARGET_IMX6ULL_POINT_ATOM_MINI`	Selects our board directory and board configuration header.
`TEXT_BASE`	Links U-Boot to execute at `0x87800000`, the same address used by the Boot ROM image.
`SYS_MALLOC_LEN`	Reserves 16 MiB in relocated DDR for U-Boot's dynamic allocations.
`NR_DRAM_BANKS`	Says the board reports one contiguous DDR bank.
`SYS_MALLOC_F_LEN`	Reserves 16 KiB of early allocation space for pre-relocation driver model, taken below the initial stack/global-data area.
`DEFAULT_DEVICE_TREE`	Selects `imx6ull-point-atom-mini-from-scratch.dtb` and appends it to `u-boot.bin`.

Configuration	Why it is enabled or assigned
`OF_CONTROL`	Makes U-Boot discover UART, timer, and MMC from Device Tree.
`OF_SEPARATE`, `OF_UPSTREAM` disabled	Appends our local DTB rather than selecting a DT from the upstream Linux DT subtree.
`CLK` disabled	Uses the three explicit clock functions in `clock.c` instead of assuming a complete clock-controller driver exists.
`PINCTRL` disabled	Uses the complete pad and daisy writes in the DCD instead of assuming a pin-controller driver exists.
`SYS_ICACHE_OFF`	Leaves the instruction cache off during first bring-up.
`SYS_DCACHE_OFF`	Leaves the data cache and MMU mapping off during first bring-up.
`BAUDRATE`	Sets the normal U-Boot console to 115200 baud.
`BOOTDELAY`	Waits three seconds before executing `bootcmd`.
`USE_BOOTCOMMAND`	Allows this defconfig to provide a fixed first-stage `bootcmd`.
`BOOTCOMMAND`	Prints one harmless line. It does not attempt to boot Linux yet.
`SYS_PROMPT`	Makes our command prompt easy to recognize.
`SYS_PBSIZE`	Allocates a 512-byte console print buffer.
`DISPLAY_CPUINFO`	Allows U-Boot to print the CPU information line when available.

Configuration	Why it is enabled or assigned
`DISPLAY_BOARDINFO`	Calls `checkboard()` and prints our board name.
`HUSH_PARSER`	Enables U-Boot's normal shell parser for variables and command lists.
`SYS_MAXARGS`	Allows a command to have up to 32 arguments.
`SYS_LOAD_ADDR`	Gives load commands a default destination at `0x82000000`.
`SYS_MEMTEST_START`	First byte used by our deliberate DDR test.
`SYS_MEMTEST_END`	End of the 8 MiB DDR test window.
`CMD_MEMORY`	Enables `md`, `mw`, `cp`, and other memory commands.
`CMD_MEMTEST`	Enables the `mtest` DDR test command.
`CMD_MMC`	Enables `mmc list`, `mmc info`, `mmc read`, and related commands.
`CMD_FAT`	Enables commands for FAT filesystems.
`CMD_EXT4`	Enables commands for ext4 filesystems.
`CMD_FS_GENERIC`	Enables generic `load`, `ls`, and `fstype` commands.
`CMD_DM`, `CMD_SLEEP`	Explicitly includes the device-tree and timer checkpoint commands used later.
`ENV_IS_NOWHERE`	Uses a compiled default environment and never writes persistent storage.

Configuration	Why it is enabled or assigned
`ENV_SIZE`	Sizes the environment buffer; it is not a media reservation when the backend is nowhere.
`NO_NET`, `NET` disabled, `EFI_LOADER` disabled, `TOOLS_MKEFICAPSULE` disabled	Selects the no-network member of the networking choice and excludes UEFI/capsule tooling. Merely unsetting `NET` would leave Kconfig free to select its default networking member. OpenSSL remains needed by the default image-tool build.
`DM_SERIAL`	Enables the driver-model serial uclass.
`IMX6ULL_SERIAL`	Compiles the UART driver we wrote in this chapter.
`TIMER`	Enables the driver-model timer uclass.
`IMX6ULL_GPT_TIMER`	Compiles our GPT1 timer driver.
`MMC`	Enables the common MMC, SD, and eMMC protocol layer and block-device support.
`DM_MMC`	Enables the driver-model MMC uclass.
`MMC_WRITE` disabled	Removes block write/erase support from the MMC core for first discovery. It does not forbid all card configuration commands.
`IMX6ULL_USDHC`	Compiles the USDHC2 PIO driver written in this chapter.

Cache is disabled for first bring-up. Generic ARMv7 entry still establishes the expected CP15 state; later cache enabling is suppressed. Enabling data cache on ARMv7 also entails MMU/page-table and device-memory attributes. Do not copy an ARM64 `mem_map[]` prescription into this ARMv7 port. Review the ARMv7 cache implementation, valid RAM ranges, MMIO attributes, alignment, and Linux handoff together, then test cache enabling as a separate change.

`ENV_IS_NOWHERE` is equally deliberate. A wrong environment offset can overwrite an SD partition or eMMC boot area. Persistent environment storage belongs after block access and the storage layout are verified.

24A.19 Check the source tree before building

The final port-owned tree should be:

```
$ find arch/arm/mach-imx6ull board/point-atom/imx6ull-mini -type f | sort
arch/arm/mach-imx6ull/Kconfig
arch/arm/mach-imx6ull/Makefile
arch/arm/mach-imx6ull/clock.c
arch/arm/mach-imx6ull/cpu.c
arch/arm/mach-imx6ull/early_uart.c
arch/arm/mach-imx6ull/include/mach/clock.h
arch/arm/mach-imx6ull/include/mach/hardware.h
arch/arm/mach-imx6ull/include/mach/uart.h
board/point-atom/imx6ull-mini/Kconfig
board/point-atom/imx6ull-mini/MAINTAINERS
board/point-atom/imx6ull-mini/Makefile
board/point-atom/imx6ull-mini/board.c
board/point-atom/imx6ull-mini/imximage.cfg
```

Also confirm the remaining created files:

```
arch/arm/dts/imx6ull-from-scratch.dtsi
arch/arm/dts/imx6ull-point-atom-mini-from-scratch.dts
arch/arm/dts/imx6ull-point-atom-mini-from-scratch-u-boot.dtsi
configs/imx6ull_point_atom_mini_defconfig
drivers/serial/serial_imx6ull.c
drivers/timer/imx6ull_gpt_timer.c
include/configs/imx6ull_point_atom_mini.h
drivers/mmc/imx6ull_usdhc.c
```

Run U-Boot’s whitespace check before compiling:

```
$ git diff --check
```

No output means the tracked diff passed that whitespace check. It does not inspect untracked new files, resolve Kconfig, or validate register values. Review `git status --short` against the two file inventories; do not accidentally include unrelated experiments. The DDR block is part of the supplied `imximage.cfg`, so the inventory has no missing board-specific include.

24A.20 Configure and build U-Boot

Build out of tree. A new output directory prevents a previous board’s configuration from leaking into this one and keeps the edited source easy to inspect:

```
$ . ~/imx6ull/scripts/env.sh
$ UBOOT_BASE=88dc2788777babfd6322fa655df549a019aa1e69
$ test "$(git rev-parse --verify 'v2026.04^{commit}')" = "$UBOOT_BASE"
$ git merge-base --is-ancestor "$UBOOT_BASE" HEAD
$ git log --oneline "$UBOOT_BASE..HEAD"
$ git status --short
$ mkdir -p ~/imx6ull/build
$ UBOOT_OUT=$(mktemp -d ~/imx6ull/build/uboot-24a-v2026.04.XXXXXX)
$ make O="$UBOOT_OUT" CROSS_COMPILE=arm-none-linux-gnueabihf- \
      imx6ull_point_atom_mini_defconfig
$ make O="$UBOOT_OUT" CROSS_COMPILE=arm-none-linux-gnueabihf- \
      -j$(nproc)
```

Run this block from `~/imx6ull/src/u-boot-24a`. The tag check still verifies the upstream identity; the ancestry check accepts either the base itself or your custom descendant commits. Stop if either fails. Review the displayed commits and local changes: ancestry alone does not prove that only the intended chapter edits are present. Do not use `git describe --exact-match HEAD` as a post-feature-commit requirement, or reset away Chapter 21's custom command just to obtain that output.

The configure command resolves the defconfig through Kconfig into `$UBOOT_OUT/.config`. The build compiles target code with the ARM compiler and host tools with native GCC. Keep `UBOOT_OUT` in this shell for the remaining commands. No image-header target is wired into our custom machine Makefile: this build produces `u-boot.bin`, not the standard EVK's `u-boot-dtb.imx`.

Confirm the important generated values:

```
$ grep -E 'CONFIG_(ARCH_IMX6ULL|TEXT_BASE|DEFAULT_DEVICE_TREE|IMX6ULL_SERIAL|
IMX6ULL_GPT_TIMER|IMX6ULL_USDHC)=' "$UBOOT_OUT/.config"
```

Required configuration values, not a captured board log:

```
CONFIG_ARCH_IMX6ULL=y
CONFIG_TEXT_BASE=0x87800000
CONFIG_DEFAULT_DEVICE_TREE="imx6ull-point-atom-mini-from-scratch"
CONFIG_IMX6ULL_SERIAL=y
CONFIG_IMX6ULL_GPT_TIMER=y
CONFIG_IMX6ULL_USDHC=y
```

Confirm that the main artifacts exist:

```
$ ls -l "$UBOOT_OUT"/u-boot "$UBOOT_OUT"/u-boot.bin \
        "$UBOOT_OUT"/u-boot.map "$UBOOT_OUT"/dts/dt.dtb \
        "$UBOOT_OUT"/tools/mkimage
$ arm-none-linux-gnueabihf-readelf -h "$UBOOT_OUT/u-boot"
$ arm-none-linux-gnueabihf-nm -n "$UBOOT_OUT/u-boot" | head
```

`u-boot` is the ELF file with symbols. `u-boot.bin` is the flat executable with the selected DTB appended. `u-boot.map` shows where every function and section was linked. `dts/dt.dtb` is the compiled board Device Tree.

Check that the ELF is ARM and `_start` is at `0x87800000`. In `.config`, also check `SKIP_LOWLEVEL_INIT_ONLY=y`, `SYS_ICACHE_OFF=y`, `SYS_DCACHE_OFF=y`, the three custom drivers, and the disabled MMC-write/high-speed features. Decompile the built DTB with `$UBOOT_OUT/scripts/dtc/dtc -I dtb -O dts "$UBOOT_OUT/dts/dt.dtb"` and inspect its selected console, timer, one-bit MMC width, and 20 MHz ceiling. A disabled Kconfig option can disappear during dependency resolution; reading the generated file is more reliable than assuming the defconfig line took effect.

The complete listings, including the documented vendor DDR block, were compiled in a fresh private copy of the pinned upstream source with Arm GNU 13.2.Rel1 during this review. The actual image was then generated and its encoded DCD compared with the vendor source. These establish source and image milestones only. This custom port was not executed on a board.

Read build failures literally

Error	Most likely missing step
`can't open file arch/arm/mach-imx6ull/Kconfig`	The `source` line exists, but the file or directory name is wrong.
`No rule to make target imx6ull_usdhc.o`	The MMC Makefile line exists, but the driver filename is wrong or missing.
`asm/arch/hardware.h: No such file`	The machine line in `arch/arm/Makefile` is missing or misspelled.
`undefined reference to imx6ull_get_uart_clock`	`clock.o` is absent from the machine Makefile.
`undefined reference to imx6ull_get_usdhc2_clock`	The USDHC driver is enabled, but `clock.o` is missing from the machine Makefile.
`FDT_ERR_NOTFOUND` or missing DTB target	The DTS Makefile line or `DEFAULT_DEVICE_TREE` name is wrong.
Missing `openssl/...`, parser generator, or native utility	Host build dependency or its private include/tool path, not an ARM peripheral bug.

Do not respond to the first compiler error by enabling unrelated Kconfig symbols. Follow the filename and symbol named by the error.

24A.21 Build the i.MX Boot ROM image

`u-boot.bin` alone is not bootable on this SoC. It has no i.MX IVT, Boot Data, or DCD. Build those around it in two visible commands.

Section 24A.14 now supplies every directive, including the documented DDR block. Preprocess the configuration to remove C comments. This uses the actual board image configuration, not a substitute header-format fixture:

```
$ cpp -P -x c board/point-atom/imx6ull-mini/imximage.cfg \
      -o "$UBOOT_OUT/u-boot.cfgout"
```

Then run the U-Boot host tool:

```
$ "$UBOOT_OUT/tools/mkimage" \
      -n "$UBOOT_OUT/u-boot.cfgout" \
      -T imximage \
      -e 0x87800000 \
      -d "$UBOOT_OUT/u-boot.bin" \
      "$UBOOT_OUT/u-boot-imx6ull.imx"
```

Every argument has a specific job:

Argument	Meaning
`-n u-boot.cfgout`	Read our image version, SD boot type, and DCD writes.
`-T imximage`	Generate the i.MX Boot ROM image format.
`-e 0x87800000`	Put `0x87800000` in the IVT entry field and calculate the load layout from it.
`-d u-boot.bin`	Use our complete U-Boot binary as the payload.
`u-boot-imx6ull.imx`	Name of the resulting SD-card image.

Inspect the generated header:

```
$ "$UBOOT_OUT/tools/dumpimage" -l "$UBOOT_OUT/u-boot-imx6ull.imx"
```

The report should identify an i.MX image and show the `0x87800000` entry address. Also check that the image is larger than `u-boot.bin` because it contains the ROM header and DCD:

```
$ ls -l "$UBOOT_OUT/u-boot.bin" "$UBOOT_OUT/u-boot-imx6ull.imx"
```

Read the actual IVT and Boot Data fields as well. This small host-side check decodes little-endian words and resolves the Boot Data pointer relative to `self`:

```
$ python3 - "$UBOOT_OUT/u-boot-imx6ull.imx" "$UBOOT_OUT/u-boot.bin" <<'PY'
import struct
import sys
from pathlib import Path

image = Path(sys.argv[1]).read_bytes()
payload = Path(sys.argv[2]).read_bytes()
assert image[:4] == bytes.fromhex("d1 00 20 40"), "Not this v2 IVT"
entry, reserved, dcd, boot_data, self, csf, reserved2 = struct.unpack_from("<7I", image,
4)
start, size, plugin = struct.unpack_from("<3I", image, boot_data - self)
assert (entry, self, boot_data, dcd, start, plugin, csf) == (
    0x87800000, 0x877FF400, 0x877FF420, 0x877FF42C, 0x877FF000, 0, 0)
assert image[0xC00:0xC00 + len(payload)] == payload, "Payload differs"
assert size >= 0x1000 + len(payload), "Boot Data load size is too small"
for name, value in (("entry", entry), ("self", self), ("DCD", dcd),
                    ("Boot Data pointer", boot_data), ("Boot Data start", start),
                    ("Boot Data size", size)):
    print(f"{name}: 0x{value:08X}")
PY
```

These assertions are specific to this chapter’s non-plugin, unsigned SD image. They detect a changed layout or payload, not DDR electrical faults. The encoded DCD must also retain section 24A.14’s ordered vendor block and teaching pad/reset additions.

What `mkimage` adds

```
u-boot-imx6ull.imx
|-- IVT header
|-- entry = 0x87800000
|-- DCD pointer
|-- Boot Data pointer
|-- self pointer
|-- DCD command table from u-boot.cfgout
|-- padding and payload metadata
`-- u-boot.bin
```

The IVT field named `self` contains the RAM address where the ROM sees that IVT. It does not mean that the IVT executes code. The ROM reads the structure and follows its pointers.

Do not confuse three address spaces: an offset inside the `.imx` file, a byte offset on the card, and a RAM address from the IVT. For this v2 SD format, `include/imximage.h` sets an initial load region of `0x1000` and an IVT media offset of `0x400`. The generated file starts **at the IVT**, so its payload starts at file offset `0xC00`; placing the file at card byte `0x400` puts the payload at card byte `0x1000`.

Item	File offset / value for this layout
IVT	File `0x000`; media `0x400`; `self = 0x877FF400`
Boot Data	File `0x020`; IVT pointer `0x877FF420`
DCD	File `0x02C`; IVT pointer `0x877FF42C` when entries exist
Payload	File `0xC00`; media `0x1000`; execution address `0x87800000`
Boot Data start	`0x877FF000`, not the executable entry

These relationships follow the pinned `tools/imximage.c` generator. Check them again if the format, boot medium, plugin mode, or header sizing changes. In this release, `dumpimage`'s v2 line labelled `Load Address` prints the **Boot Data pointer**, so it is not sufficient evidence for the payload load address. Boot Data size includes the image load layout and alignment; do not substitute the raw binary length. Image inspection proves pointer/layout consistency, not that the DCD will initialize the physical DDR.

24A.22 Write the image to an SD card

This is a later hardware procedure, not part of the source-build example. Confirm the fitted core matches the documented DDR target and complete the image/layout checks before using a dedicated expendable card. Establish a power-off/recovery plan for this untested teaching runtime, and retain Chapter 19's vendor baseline. A write beginning at 1 KiB can still overwrite a GPT entry array or an existing bootloader. Use a reviewed raw/MBR layout with an explicitly reserved boot gap; this offset is not a general guarantee of partition safety.

Identify the whole removable device by size, model, transport, and physical insertion/removal before doing anything destructive:

```
$ lsblk -o NAME,SIZE,MODEL,TRAN,MOUNTPOINTS
```

In the commands below, `/dev/sdX` means the whole card. Replace `sdX` with the real device. Do not use a partition such as `/dev/sdX1`.

Record the resolved whole-device path and verify it is the intended card, not a system disk. Back up anything needed. Unmount each mounted partition by its exact name and stop on any error; do not hide failures behind a wildcard and `|| true`. Confirm the image fits wholly within the reserved boot gap before proceeding. The following `SD_DEVICE` is a placeholder that must be assigned to that verified whole device, never a partition:

```
$ SD_DEVICE=/dev/sdX
$ sudo dd if="$UBOOT_OUT/u-boot-imx6ull.imx" of="$SD_DEVICE" \
          bs=1K seek=1 conv=fsync,notrunc status=progress
$ sync
```

`bs=1K seek=1` skips 1 KiB, which is offset `0x400`. This matches this image's SD IVT position. Writing the same file at zero puts the IVT in the wrong place. Do not add another 1 KiB of padding to a file that already begins with the IVT.

Read back and compare the **whole image**, not just its first 64 bytes:

```
$ IMAGE_SIZE=$(stat -c %s "$UBOOT_OUT/u-boot-imx6ull.imx")
$ sudo dd if="$SD_DEVICE" of="$UBOOT_OUT/sd-readback.imx" \
      iflag=skip_bytes,count_bytes skip=1024 count="$IMAGE_SIZE" status=none
$ cmp "$UBOOT_OUT/u-boot-imx6ull.imx" "$UBOOT_OUT/sd-readback.imx"
```

`cmp` must exit successfully with no differences. This confirms stored bytes, not ROM acceptance or board operation. No card writes or readbacks were performed during this review.

24A.23 Connect the built-in console and boot

The board already contains the USB-to-TTL bridge. Do not connect an external USB-to-TTL adapter.

1. Connect the board's USB debug or serial connector to the host.
2. Find the new device with `dmesg` or `ls /dev/ttyUSB* /dev/ttyACM*`.
3. Open it at 115200 baud.
4. Set the board boot switches to SD mode.
5. Insert the prepared SD card.
6. Power-cycle the board.

For example:

```
$ picocom -b 115200 /dev/ttyUSB0
```

Use the actual device name. The terminal settings are 115200 baud, 8 data bits, no parity, and 1 stop bit. Hardware flow control must be off.

Capture the complete terminal log from power-on. This chapter does not supply a fabricated successful boot transcript. Look for the literal strings our code emits: `[imx6ull] arch_cpu_init reached`, then `[imx6ull] board_early_init_f reached`. Later, a banner and `imx6ull=>` establish further checkpoints. The `512 MiB` report is the configured bank assignment, not a DDR measurement. Preserve missing, partial, or garbled output as evidence rather than adjusting the log to match an example.

24A.24 Test one subsystem at a time

Test the timer

```
imx6ull=> sleep 1
imx6ull=>
```

The prompt should return after about one second. An immediate return or a permanent hang points to GPT clock, reset, prescaler, or Device Tree selection.

Inspect the memory map

```
imx6ull=> bdinfo
```

Check these values in the output:

```
DRAM bank   = 0x00000000
-> start    = 0x80000000
-> size     = 0x20000000
```

The relocation address should be inside DDR and must not overlap the test window at `0x81000000` to `0x817FFFFF`.

Test DDR without overwriting U-Boot

```
imx6ull=> mtest 0x81000000 0x81800000 0x00000000 1
```

This is a half-open 8 MiB window with the default quick-test implementation. Check the compiled test variant before interpreting an endpoint: alternative tests and optional bit-flip checks have different accesses. Even this aligned range is destructive and is not automatically free just because it lies below U-Boot’s relocation address. Verify the full runtime reservations and any loaded buffers first, then stop on any mismatch. Do not extend a running U-Boot memory test over its own stack, heap, DTB, or executable. A small successful scratch test does not qualify the whole DDR bank or its calibration.

Test eMMC discovery

```
imx6ull=> mmc list
imx6ull=> mmc dev 0
imx6ull=> mmc info
imx6ull=> mmc part
```

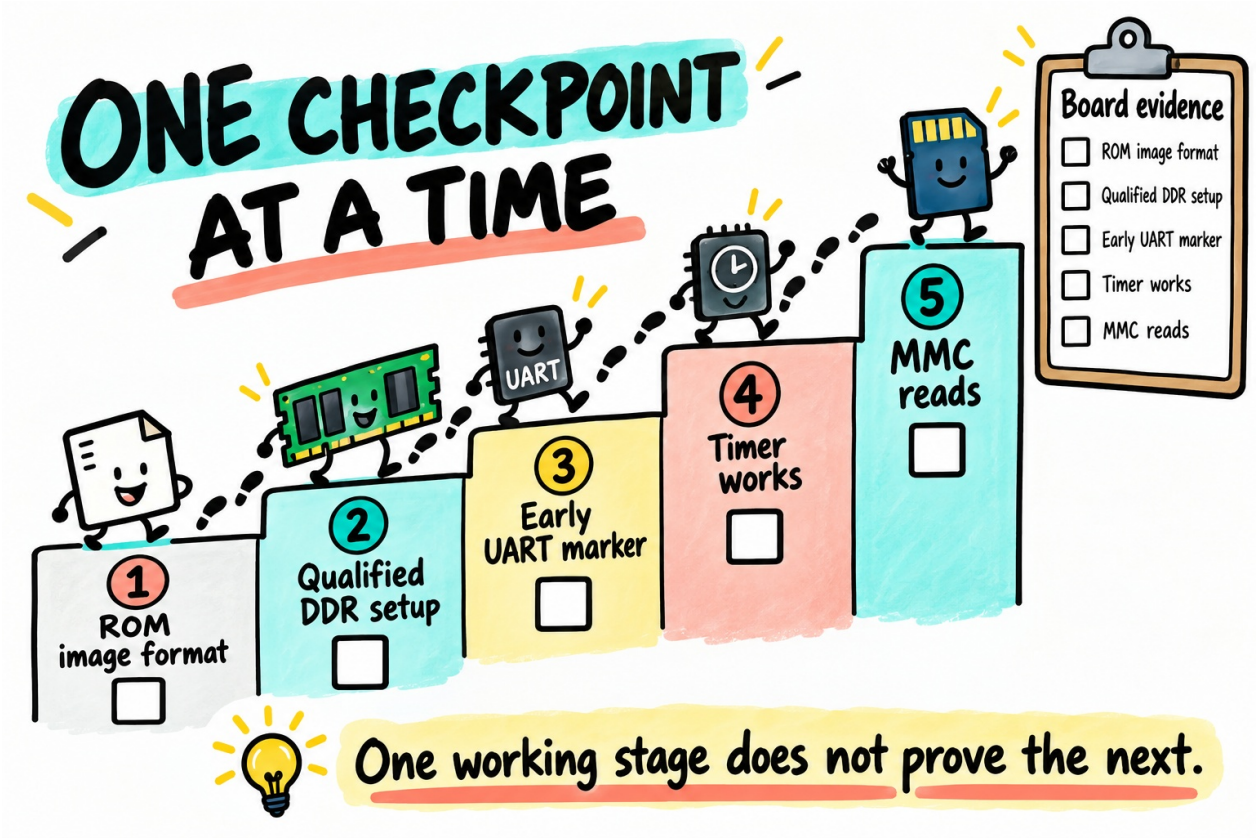
What to record if discovery succeeds:

- `mmc list` shows `i.MX6ULL USDHC2 PIO`.
- `mmc dev 0` selects it without a timeout.
- `mmc info` identifies MMC/eMMC rather than an SD card and shows the negotiated legacy mode, one-bit bus, and actual divided clock.
- `mmc part` prints the existing partition table, if one is present.

Do not run `mmc write` during first discovery. Reading identification and partition data proves the controller path without changing storage.

Compare CID manufacturer/product/revision and capacity against the fitted part's record. CSD/EXT_CSD capacity and mode bits distinguish what the card supports from what the host currently advertises. `non-removable` means no mechanical card detect; it does not prove that a chip is connected. A missing partition table is not an identification failure. A valid partition listing examples only a few sectors, not the whole medium. Initialization still sends volatile card configuration commands even with block writes disabled; do not use partition, boot configuration, RPMB, erase, or reset-function commands here.

24A.25 Diagnose silence by the last completed stage



Gather evidence one stage at a time. An early marker narrows the search, but does not qualify DDR or prove a later peripheral works. The memory and card drawings are conceptual symbols, not a board layout.

Last visible result	What has already worked	Check next
No serial device appears on host	Nothing about the SoC yet	USB cable, board power, and built-in USB-to-TTL bridge
Serial device appears, but no text	Host connection only	SD boot switch, image at offset `0x400`, IVT entry, and DCD
Garbled text	UART transmits	Baud rate, 24 MHz UART selection, terminal flow control
`arch_cpu_init` marker only	ROM, DDR load, ARM startup, clocks, and early UART	`BOARD_EARLY_INIT_F`, board object, early init return

Last visible result	What has already worked	Check next
Both markers, no U-Boot banner	Board early hook works	timer probe, Device Tree inclusion, relocation, BSS or DDR corruption
Banner, but wrong DRAM size	Pre-relocation console works; relocation is not yet proven	`dram_init()`, DTS memory node, exact DDR hardware fitted
Prompt works, `sleep` hangs	Most of U-Boot works	GPT node, GPT clock gate, `tick-timer`, timer rate
Prompt works, `mmc dev 0` times out	DDR, timer, and console work	eMMC pad mux, daisy values, USDHC2 clock, reset line, voltage

This table is why the early markers are direct register writes. They remain available even if driver-model serial or relocation fails.

Inspect the final Device Tree from U-Boot

At the prompt:

```
imx6ull=> dm tree
```

Look for the serial, timer, and MMC uclasses, with nodes at the expected addresses and their probe indicators. `dm tree` commonly displays **node/device names**, such as `serial@2020000`, not necessarily the `U_BOOT_DRIVER` identifier. `dm drivers` can help inspect registered driver names. Binding means a compatible driver was found; probing means its initialization callback succeeded; neither proves successful eMMC identification.

If a device is absent, first check its Kconfig symbol and Device Tree `compatible`. If it is present but not probed, inspect its `reg`, status, and required clock setup.

24A.26 What is complete and what is deliberately absent

The supplied source covers the teaching runtime and can be built against the pinned release. It is not a claimed board-qualified image. The source includes:

- A new ARM machine selection
- A new board target
- Direct early clock and UART code

- A driver-model serial driver
- A driver-model timer driver
- A driver-model USDHC2 PIO driver
- A complete ROM image configuration with the pinned vendor’s documented 512 MiB eMMC-core DDR initializer
- A SoC `.dtsi` and board `.dts`
- A complete defconfig
- Reproducible image creation, later SD provisioning, and separate physical verification procedures

These features are not silently assumed. They are deliberately postponed:

Feature not added	Reason to add it later
SPL	A different boot-chain design, not a missing stage of this MINI ROM+DCD flow. Any Chapter 20 SPL example is a separate source/build-only reference, not a MINI image.
Data cache and MMU	Need a tested memory-region map. Enable them after the basic port is stable.
Pin controller driver	The first DCD performs the exact pad writes. A reusable pinctrl driver becomes useful when many peripherals and runtime pin states are added.
Clock controller driver	The first port has three explicit clock consumers. A driver-model clock tree becomes useful as the peripheral count grows.
Wider/faster MMC, DMA, and tuning	First prove legacy one-bit reads. Wider buses need line checks; high speed needs qualified pad/timing settings. DMA and advanced modes add cache, descriptor, voltage, and sampling work.
Persistent environment	Needs a reviewed eMMC or SD offset and erase/write policy.

Feature not added	Reason to add it later
Ethernet, USB, NAND, display	They do not help prove the minimum boot chain. Add one driver and one visible test at a time.
Linux boot command	Chapter 23 builds `bootcmd`, `bootargs`, and image loading after U-Boot itself is trustworthy.

Postponed does not mean optional forever. It means the feature is outside the first dependency chain and has a named later step.

Qualification also remains outside the source result: we did not measure DDR calibration, full-bank stability, inherited clocks/watchdogs, boot straps, eMMC rails/reset, or controller/card error recovery. The DDR values have vendor-source provenance, while our peripheral drivers and additional pad/reset policy remain untested on hardware. Product hardening would additionally review security state, silicon revision/errata, reset/power management, watchdog servicing, and kernel handoff. Do not turn an unchecked item into a successful result by changing a chapter label from draft to complete.

24A.27 The porting method to carry to a truly new SoC

The names and register values will change on another chip, but the method remains:

1. Write the exact Boot ROM load contract and memory map.
2. Prove DDR with visible low-level code or a visible ROM configuration table.
3. Add the architecture Kconfig and machine Makefile connection.
4. Provide the early stack addresses and DDR bank description.
5. Print one direct UART marker before relying on the console framework.
6. Add a monotonic timer before enabling timeout-dependent drivers.
7. Describe devices in Device Tree and match each `compatible` to one driver.
8. Implement the smallest polling driver that proves each new peripheral before adding DMA, interrupts, or tuning.
9. Explain every defconfig option by the code or behavior it enables.
10. Build the exact ROM image format and verify it after writing the boot medium.
11. Test one subsystem at the prompt before adding the next one.

The core skill is not copying a vendor directory. It is making every dependency visible, then proving those dependencies in an order that leaves useful evidence when the board stops.

24A.28 Key Notes

- The port joins several contracts: ROM image layout, early ARM runtime, U-Boot relocation, Device Tree discovery and peripheral framework callbacks. A correct interface at one stage does not establish the next.
- An early UART can use bounded private polling before the driver model exists. The later serial callback must follow its framework's readiness semantics rather than block indefinitely.
- MMC response type, transfer deadlines and recovery ordering belong to the controller/protocol boundary. Extending a deadline after every word is not the same policy as bounding the whole transfer.
- The vendor DCD supplies named DDR initialization values for a documented core-board design. Source provenance and successful compilation are not measured calibration or physical memory qualification.
- File placement, media placement and loaded RAM pointers must agree in the ROM image. Header readback and a printed memory size cannot prove full storage integrity or DDR stability.
- This teaching route intentionally omits platform responsibilities. A product port still needs power, reset, security, errata and failure recovery ownership beyond the three small drivers.

Primary sources for this chapter

- Official Alientek eMMC-core DCD and its defconfig establish the copied initializer. The supplied guide V1.81 page 7, DDR pages 525-550, and chapter 33 pages 860-911 establish MINI applicability and the documented eMMC-core design. The supplied core V2.0 schematic confirms the labelled DDR and wiring, not our own bench validation.
- Pinned U-Boot source: `arch/arm/cpu/armv7/start.S`, `arch/arm/lib/crt0.S`, `common/board_f.c`, and `common/board_r.c` establish startup and relocation.
- Image generator and format definitions establish the IVT/DCD layout. EVK documentation describes that board's DCD image, not a MINI qualification.
- MMC API, MMC core, and upstream USDHC implementation are comparison evidence for callback/protocol contracts, not hidden linked drivers in this custom port.

- NXP i.MX6ULL documentation, especially IMX6ULLRM Rev. 1 chapters 8 (ROM), 18 (CCM), 30 (GPT), 32 (IOMUXC), 35 (MMDC), 55 (UART), 58 (USDHC), and 59 (WDOG). The supplied core/baseboard schematics establish candidate wiring; fitted-part and calibration evidence must still come from the board record.

Previous: Chapter 24: Workflows: TFTP, NFS, USB-OTG

Next: Chapter 24B: U-Boot board policy with GPIO and I2C

Chapter 24B: U-Boot board policy with GPIO and I2C

A bootloader can read a temperature register correctly and still make the wrong decision. A truncated negative value, a stale saved `bootcmd`, or a recovery check followed unconditionally by normal boot can undo the intended gate. The important part is carrying the input all the way to the boot decision.

We will add a once-per-boot policy to the Chapter 22 MINI migration port. These examples target upstream **U-Boot v2026.04**, commit `88dc2788777babfd6322fa655df549a019aa1e69`, with the established names:

```
board/myorg/mx6ull_pa_mini/mx6ull_pa_mini.c
configs/mx6ull_pa_mini_defconfig
arch/arm/dts/imx6ull-pa-mini.dts
include/configs/mx6ull_pa_mini.h
```

The supplied MINI V2.2 schematic gives us a real starting point: the stock **KEY0** button and two routed I2C buses. The TMP102 at `0x48` is an **optional external add-on**, not an onboard temperature sensor. We will study its error handling on the host, then use KEY0 for a policy that needs no added sensor.

These are modern driver-model examples, not patches to paste into Chapter 19’s vendor U-Boot. The vendor baseline establishes the documented MINI starting point; compiling v2026.04 helpers does not establish a working modern MINI image. Keep the two source trees, configurations and build outputs separate.

MINI V2.2 resource	Documented connection	Use here
KEY0	UART1_CTS pad / GPIO1_IO18; switch to ground, R12 10 kOhm to 3.3 V	Stock recovery input.
I2C1	UART4 TX/RX pads; P4 pin 43 is SCL, pin 42 is SDA when muxed to I2C	Optional TMP102, later identity EEPROM.
I2C2	UART5 TX/RX pads; P4 pin 45 is SCL, pin 44 is SDA, also RGBLCD J1 and camera P1	Leave to fitted display/touch/ camera hardware.

MINI V2.2 resource	Documented connection	Use here
ON/OFF, RESET, BOOT_CFG1	Power control, shared hardware reset, ROM boot selection	Not recovery GPIOs.

This mapping comes from MINI schematic sheets 1-3 and CORE sheet 6. CORE connector aliases describe several possible baseboards; the **MINI** sheet decides what is actually connected. In particular, MINI SNVS_TAMPER0 is `WIFI_REG_ON`, not KEY0. The guide's pp7-8 applicability table excludes the stock ALPHA I2C experiments on MINI; that is not the absence of an I2C controller or connector.

These checks run in full U-Boot after DDR initialization. The pinned mainline EVK reference uses ROM plus DCD and `u-boot-dtb.imx`, not SPL at a second storage offset. A late check cannot protect hardware during ROM/DCD initialization.

24B.1 What belongs in U-Boot policy

On an MCU you may own a peripheral for the application's lifetime. Here U-Boot owns it only long enough to decide what to start. Linux initializes its own drivers afterward.

Decision	Suitable scope
Hold at the prompt because a measured condition forbids boot	A bounded check, with a defined error policy.
Select recovery while a key is held	Read a dedicated input before dispatching either path.
Choose an OS DTB from a board ID	Validate identity first; Chapter 24D develops this.
Maintain fan speed after Linux starts	Linux thermal/PWM drivers, not a U-Boot polling loop.
Manage a complex update transaction	Usually a Linux recovery system; loading an image is a smaller job.

Stopping at the prompt leaves the board powered. It is not thermal shutdown, and one sensor does not measure every junction. A safety requirement may need hardware supervision, power removal and a qualified thermal design. The 50 C threshold is a teaching value, not a board rating.

24B.2 Files changed in this chapter

File	Responsibility
`configs/mx6ull_pa_mini_defconfig`	Select interfaces, commands and the late hook.
`arch/arm/dts/imx6ull-pa-mini.dts`	Describe MINI wiring and fitted add-ons in U-Boot's **control DT**.
`board/myorg/mx6ull_pa_mini/mx6ull_pa_mini.c`	Read inputs and dispatch policy once.
`include/configs/mx6ull_pa_mini.h`	Extend default environment without duplicate keys.

The control DT describes hardware U-Boot itself uses. It is not the OS DTB later loaded at `fdt_addr_r`. Adding a sensor only to the OS DTB does not make it available to U-Boot's driver model.

24B.3 Enable the needed U-Boot features

Merge these into the existing defconfig; this is not a complete defconfig:

```
CONFIG_BOARD_LATE_INIT=y
CONFIG_DM_GPIO=y
CONFIG_MXC_GPIO=y
CONFIG_DM_GPIO_LOOKUP_LINE_NAME=y
CONFIG_CMD_GPIO=y
CONFIG_DM_I2C=y
CONFIG_SYS_I2C_MXC=y
CONFIG_CMD_I2C=y
CONFIG_HUSH_PARSER=y
CONFIG_HUSH_OLD_PARSER=y
CONFIG_USE_BOOTCOMMAND=y
CONFIG_BOOTCOMMAND="run normal_boot"
```

`SYS_I2C_MXC` is the pinned controller symbol. Driver-model interfaces do not replace controller drivers or pad setup. Retain Chapter 22's `OF_CONTROL`, i.MX pinctrl and other board settings. The command options add diagnostics; the C reader does not need those commands. No `setexpr` is needed.

Use the existing project setup in an Ubuntu Bash terminal:

```
. ~/imx6ull/scripts/env.sh
# Continue only if setup succeeded, from your writable U-Boot source tree.
make O=../build-pa-policy ARCH=arm CROSS_COMPILE=arm-none-linux-gnueabihf-
mx6ull_pa_mini_defconfig
```

```
grep -E '^CONFIG_(BOARD_LATE_INIT|DM_GPIO|MXC_GPIO|DM_I2C|SYS_I2C_MXC|HUSH_PARSER)=' ../
build-pa-policy/.config
```

Use the Linux-target compiler, not `arm-none-eabi-`. Keep the original Arm toolchain directory names; do not change global PATH or `.bashrc`. Inspect the generated `.config`: dependencies determine what is enabled.

24B.4 Add the I2C sensor to the U-Boot Device Tree

This **DTS integration fragment** uses the documented MINI I2C1 route: UART4_TX at P4 pin **43** is SCL, UART4_RX at P4 pin **42** is SDA. These are not the two pins in one connector row. Pin 41 is ENET1_RXER, and pin 44 is UART5_RX/I2C2_SDA. A separately wired, 3.3 V-compatible TMP102 at 7-bit `0x48` is the proposed add-on. The MINI already has R42/R57 4.7 kOhm pull-ups to `DCDC_3V3` on these nets. Check the module's additional pull-ups and total loading; do not attach 5 V pull-ups or enable UART4 on the same pads. Merge existing nodes rather than duplicating labels.

```
/ {
    aliases {
        i2c0 = &i2c1;
    };
};

&i2c1 {
    pinctrl-names = "default";
    pinctrl-0 = <&pinctrl_i2c1>;
    clock-frequency = <100000>;
    status = "okay";

    temperature-sensor@48 {
        compatible = "ti,tmp102";
        reg = <0x48>;
    };
};

&iomuxc {
    pinctrl_i2c1: i2c1grp {
        fsl,pins = <
            MX6UL_PAD_UART4_TX_DATA__I2C1_SCL 0x4001b8b0
            MX6UL_PAD_UART4_RX_DATA__I2C1_SDA 0x4001b8b0
        >;
    };
};
```

Both macros exist in the pinned `imx6ul-pinfunc.h`: TX maps to SCL and RX to SDA. The value includes SION and electrical pad-control bits. Decode it against the SoC manual and wiring; it is not a universal I2C setting. I2C2 has its own R40/R41 pull-ups and shared connectors; do not move the example there without auditing attached modules and addresses.

The alias intends U-Boot sequence 0 for I2C1. Inspect the built control DT and `i2c bus`, rather than assuming Linux adapter numbering. Behind an I2C mux, select the **child bus**, with its mux driver and DT enabled; an upstream bus number is insufficient.

The raw reader does not need a TMP102-specific U-Boot driver. The I2C uclass can bind a generic chip for register access. A compatible string does not prove that hardware is fitted or that a sensor driver exists.

24B.5 Test I2C before writing policy code

These are commands for a later test **with the TMP102 add-on fitted**, not captured board output. Reads and address probes create bus traffic. Do not scan another bus or assume an ACK from a different module is the sensor.

```
=> i2c bus
=> i2c dev 0
=> i2c probe 48
=> i2c md 48 01.1 2
=> i2c md 48 00.1 2
```

`.1` specifies a one-byte register pointer, not a one-byte result. These addresses and lengths are hexadecimal. The configuration register is `0x01`; temperature is `0x00`. An ACK is not a chip identity check.

As a calculation, not an observed reading, `19 00` encodes `0x190 = 400` signed counts: `400 / 16 = 25 C`. Keep fractional counts for comparisons.

24B.6 Add default policy variables

This header **fragment** deliberately has non-booting defaults, so it can be integrated before loading commands are ready:

```
#define PA_MINI_POLICY_ENV \
    "normal_boot=echo No normal boot installed; false\0" \
    "recovery_boot=echo No recovery boot installed; false\0"
```

Append it to the existing `CFG_EXTRA_ENV_SETTINGS` once. Do not duplicate `normal_boot`. Chapter 24C replaces that path; a qualified recovery path remains separate. `CONFIG_BOOTCOMMAND` supplies default `bootcmd`; do not add a conflicting header string.

`env_set()` changes RAM, not storage. Saved values can override compiled defaults, so a rebuild alone does not change every variable on an already configured board. This chapter neither calls `saveenv` nor resets the whole environment.

24B.7 Add the temperature read code

These are complete helpers to add to the existing board file. Their contract is a genuine TMP102, normal 12-bit mode, continuous conversion, stable supply, and no concurrent configuration writer. Rejecting shutdown and extended mode prevents misinterpretation.

```
#include <dm.h>
#include <env.h>
#include <i2c.h>
#include <stdio.h>
#include <linux/delay.h>
#include <linux/errno.h>

#define PA_MINI_TEMP_BUS       0
#define PA_MINI_TEMP_ADDR      0x48
#define PA_MINI_TEMP_LIMIT_16C (50 * 16)

static int pa_mini_read_tmp102_16c(int *temp)
{
    struct udevice *chip;
    u8 config[2], raw[2];
    int sample, ret;

    ret = i2c_get_chip_for_busnum(PA_MINI_TEMP_BUS,
                                  PA_MINI_TEMP_ADDR, 1, &chip);
    if (ret)
        return ret;
    ret = i2c_set_chip_offset_len(chip, 1);
    if (ret)
        return ret;
    ret = dm_i2c_read(chip, 0x01, config, sizeof(config));
    if (ret)
        return ret;
    if ((config[0] & 0x01) || (config[1] & 0x10))
        return -EOPNOTSUPP;

    /* Allow the first conversion after a stable power-up supply. */
    mdelay(20);
    ret = dm_i2c_read(chip, 0x00, raw, sizeof(raw));
    if (ret)
        return ret;
    if (raw[1] & 0x0f)
        return -EINVAL;
    sample = ((int)raw[0] << 4) | (raw[1] >> 4);
    if (sample & 0x800)
        sample -= 0x1000;
    if (sample < -40 * 16 || sample > 125 * 16)
        return -ERANGE;
    *temp = sample;
    return 0;
}

static int pa_mini_check_temperature(void)
{
    char value[16];
    int temp, ret;
```

```
    ret = pa_mini_read_tmp102_16c(&temp);
    if (ret)
        return ret;
    snprintf(value, sizeof(value), "%d", temp);
    if (env_set("board_temp_16c", value))
        return -EIO;
    printf("Temperature sample: %d units of 1/16 C\n", temp);
    return temp >= PA_MINI_TEMP_LIMIT_16C ? -ERANGE : 0;
}
```

The TI TMP102 datasheet, sections 5.5, 6.4 and 6.5, specifies signed 0.0625 C counts, normal/extended formats, shutdown behavior, and a maximum conversion time of 15 ms in the cited revision. The 20 ms wait assumes supply is already stable; it is not power sequencing or a freshness guarantee after brownout.

`i2c_get_chip_for_busnum()` takes a pointer width. For an already bound chip, explicitly setting that width avoids relying on its DT default. Every lookup, probe and transfer error propagates, including allocation failure during generic binding. Never dereference `chip` after a failed lookup. See the pinned I2C implementation.

The fixed threshold removes loose environment parsing from the decision. Samples outside the sensor's specified -40..125 C range are rejected. The diagnostic value is signed decimal text; `env_set_ulong()` is inappropriate for negative temperatures.

24B.8 Why `board_late_init()` is the right hook

In the pinned post-relocation sequence, environment loading precedes `board_late_init`, and network initialization and the main loop follow it. Driver model is available, but devices still probe on demand and may fail.

The following is the **temperature-add-on hook**. Install it only in a build whose hardware contract includes that sensor. On an unmodified MINI, use the stock KEY0 hook in 24B.10 instead. Merge into the existing hook rather than creating a second definition:

```
int board_late_init(void)
{
    int ret;

    if (env_set("bootcmd", "echo Boot blocked by board policy; false"))
        return -EIO;
    ret = pa_mini_check_temperature();
    if (ret) {
        printf("Temperature gate blocked boot: %d\n", ret);
        return 0;
    }
    if (env_set("bootcmd", "run normal_boot"))
        return -EIO;
    return 0;
}
```

The first assignment replaces stale saved `bootcmd` with a closed gate. Only success opens it. An assignment error returns from the init hook; this init sequence then halts rather than entering autoboot with uncertain state. An ordinary sensor failure leaves a blocked command and a reachable prompt.

This is operational policy, not a security boundary. Command access allows manual boot. Audit `preboot`, boot menus, bootstd and other entry points in the final configuration; an alternate command can bypass a `bootcmd`-only gate.

24B.9 Build and test

From the same writable source tree and prepared Bash environment:

```
make O=../build-pa-policy ARCH=arm CROSS_COMPILE=arm-none-linux-gnueabihf- -j"$(nproc)"
```

Do not run `distclean` in a shared tree or repeat storage writes to test logic. The new defconfig must first exist in the Chapter 22 port; it is not present upstream.

Synthetic input	Expected result
`19 00`	400 counts, below the teaching limit.
`31 f0`	799 counts, below 50 C without truncation.
`32 00`	800 counts, blocked.
`ff f0`	-1 count, reported as signed text.
I2C failure, shutdown, extended mode	Blocked; no uninitialized sample.
Environment update failure	Error returned; normal dispatch not installed.

Test these on the host first. For a later qualified board test, interrupt autoboot and inspect `board_temp_16c` and `bootcmd`. Do not heat the board. A temporary `setenv` does **not** survive `reset`; inject failures in a separate example build or host fixture, not by saving a threshold.

24B.10 Add a GPIO recovery button

Use a dedicated, externally biased input. A raw GPIO number does not establish mux, polarity, ownership or debounce. Do not use `gpio input` on another driver's PHY reset, PMIC enable or boot strap: it changes direction. `gpio status -a` is inspection, not permission to take a pin.

Use the stock MINI V2.2 **KEY0**, not a new button: schematic sheet 2 shows the ground switch and R12 pull-up, and sheet 1 routes `KEY0` to UART1_CTS / GPIO1_IO18. UART1 TX/RX still serve the serial console, but CTS must not simultaneously be UART hardware flow control or another GPIO consumer.

This **control-DT fragment** names that line. Merge the name into index 18 of an existing line-name array, preserving other names; the empty entries below are only for a new array:

```
&gpio1 {
    gpio-line-names =
        "", "", "", "", "", "", "", "",
        "", "", "", "", "", "", "", "",
        "", "", "PA_RECOVERY_KEY";
};
```

The board's pad setup must select `MX6UL_PAD_UART1_CTS_B__GPIO1_IO18` before the helper runs; a line-name lookup does not automatically select pinctrl. Retain the external R12 pull-up and qualify the input pad settings. A line name does not encode polarity either: the C descriptor explicitly marks KEY0 active low.

The CORE connector's `KEY0` accessory alias on J2 pin 49 names GPIO1_IO01, but MINI uses that signal for `GBC_KEY / AP_INT`, not its onboard button. SNVS_TAMPER0 is another separate signal, `WIFI_REG_ON`, and must not be borrowed as a recovery key. SNVS/tamper pads belong to **IOMUXC-SNVS**, not an arbitrary group under `&iomuxc`. The actual KEY0 route above is a main-IOMUXC UART pad.

Add this complete helper and its includes to the same C file:

```
#include <asm/gpio.h>

static int pa_mini_recovery_key(bool *pressed)
{
    struct gpio_desc key;
    int first, value, i, ret, release_ret;

    ret = dm_gpio_lookup_name("PA_RECOVERY_KEY", &key);
    if (ret)
        return ret;
    ret = dm_gpio_request(&key, "pa-recovery");
    if (ret)
        return ret;
    ret = dm_gpio_set_dir_flags(&key, GPIOD_IS_IN | GPIOD_ACTIVE_LOW);
    if (ret)
```

```
        goto release;
    first = dm_gpio_get_value(&key);
    ret = first < 0 ? first : 0;
    for (i = 0; !ret && i < 4; i++) {
        mdelay(5);
        value = dm_gpio_get_value(&key);
        if (value < 0)
            ret = value;
        else if (value != first)
            ret = -EAGAIN;
    }
release:
    release_ret = dm_gpio_free(NULL, &key);
    if (!ret)
        ret = release_ret;
    if (!ret)
        *pressed = first != 0;
    return ret;
}
```

Lookup does not claim the pin. A successful request owns it until release, including when direction setup or a read fails. A failed request must not free someone else's claim. `dm_gpio_get_value()` returns **logical active** with descriptor polarity applied: a pressed active-low key gives 1. Five equal samples over 20 ms are a bounded teaching debounce policy, not a switch specification. An unstable sample blocks this example; a product may choose a bounded retry.

For a **stock MINI without a temperature sensor**, this is the complete replacement hook. It uses the same blocked-first dispatch and the helper above:

```
int board_late_init(void)
{
    bool pressed;
    int ret;

    if (env_set("bootcmd", "echo Boot blocked by board policy; false"))
        return -EIO;
    ret = pa_mini_recovery_key(&pressed);
    if (ret) {
        printf("Recovery key check blocked boot: %d\n", ret);
        return 0;
    }
    if (env_set("bootcmd", pressed ? "run recovery_boot" : "run normal_boot"))
        return -EIO;
    return 0;
}
```

For the **sensor-equipped example**, instead replace the final dispatch assignment in 24B.8 with this **hook fragment**, adding `bool pressed;` to its declarations. The temperature gate then remains mandatory:

```
    ret = pa_mini_recovery_key(&pressed);
    if (ret) {
        printf("Recovery key check blocked boot: %d\n", ret);
```

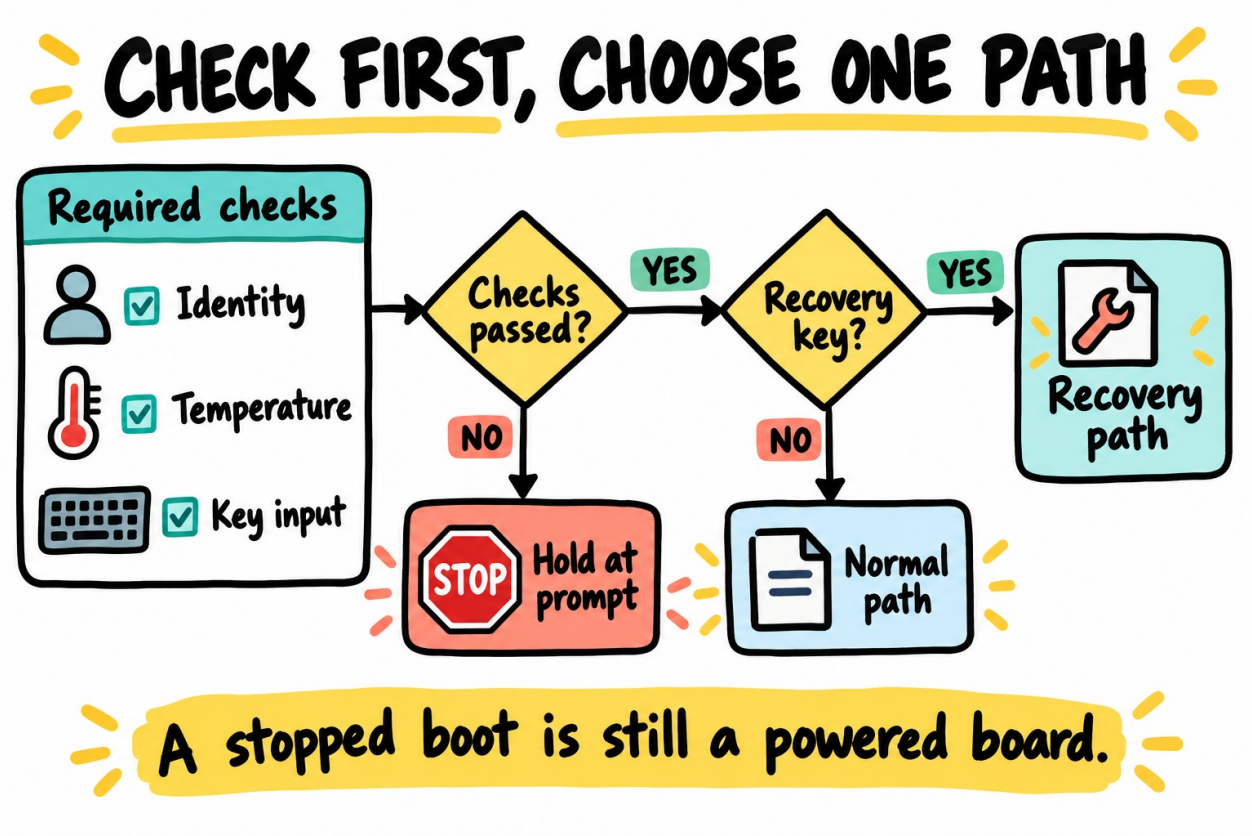
```
        return 0;
    }
    if (env_set("bootcmd", pressed ? "run recovery_boot" : "run normal_boot"))
        return -EIO;
```

Recovery and normal boot are exclusive. Changing `bootcmd` inside a running command does not cancel its remaining commands. `run recovery_check; run normal_boot` still executes normal boot.

24B.11 Fail-open or fail-closed

Check	This example’s action	Qualification needed
Temperature at/above limit	Hold at prompt	Thermal requirement and sensor placement.
Mandatory sensor unavailable	Hold at prompt	Error does not establish a safe temperature.
Recovery input unreadable/ unstable	Hold at prompt	Dedicated pin and another service path.
Invalid board identity	Hold in Chapter 24D	No invented “safe” DTB.
Optional peripheral absent	Product-specific	Do not silently weaken a mandatory check.

MINI sheet 4 and CORE sheet 1 show discrete regulators, not an I2C PFUZE3000 PMIC. The vendor board file’s inherited “I2C1 for PMIC and EEPROM” comment is not a MINI parts list. A future PMIC fault input would need its own documented polarity, latching and power behavior; there is none to read in this example. Boot policy is not a substitute for hardware protection.



Open only one path after its prerequisites pass. This diagram shows the combined sensor/identity add-on policy developed across these chapters. A stock MINI uses KEY0 without those absent devices. It is not a hardware safety circuit. Holding at the prompt does not remove power.

24B.12 Key Notes

- `board_late_init()` runs after environment import but before normal dispatch. A blocked-first hook prevents stale saved commands from opening the gate before required inputs are valid.
- The stock MINI input is KEY0 on UART1_CTS/GPIO1_IO18. The TMP102 is an optional add-on, so its mandatory temperature gate belongs only to a sensor-equipped hardware contract.
- TMP102 comparison uses signed sixteenth-degree counts, preserving negative values and fractional thresholds. Transport errors and unsupported sensor modes are not valid samples.
- GPIO lookup, claim, mux, polarity and release are separate responsibilities. Descriptor logical activity already includes active-low interpretation, while bounded stable samples define the example’s debounce policy.
- Recovery and normal dispatch are exclusive choices. Changing `bootcmd` inside a running script does not cancel commands that follow in that script.

- A late software hold leaves power applied and permits operator commands. It is neither ROM-time protection nor a security or thermal-shutdown boundary, and it requires no persistent environment or fuse changes.

24B.13 Pitfalls

- **Pointer width.** Sensor register pointers and EEPROM addresses can have different widths.
- **Bus selection.** A U-Boot sequence is not a Linux adapter number; muxes introduce child buses.
- **Unsigned temperature.** Preserve signed counts for comparison and reporting.
- **Environment errors.** Do not open the gate after a failed assignment.
- **Pin ownership.** Direction changes on shared reset/power pins affect hardware immediately.
- **Gate bypass.** Mutable environment and prompt access are not trusted boot policy.
- **Powered hold.** A board at the prompt can continue to heat.

24B.14 Going deeper

- I2C API and uclass: binding, pointer widths and errors.
- GPIO descriptors and uclass: claims, polarity and release.
- Board initialization and autoboot: detection before execution.
- Environment API: volatile updates and return values.

Previous: Chapter 24A: Building i.MX6ULL U-Boot from nothing

Next: Chapter 24C: Ethernet fallback boot in U-Boot

Chapter 24C: Ethernet fallback boot in U-Boot

Suppose U-Boot is running, but the boot partition has no kernel. Ethernet can supply a replacement kernel and DTB without repairing that partition first. The trap is that a failed load leaves old bytes in RAM: a script separated only by semicolons can boot those bytes instead of taking its fallback.

This chapter builds a local-first loading policy for **U-Boot v2026.04**, commit `88dc2788777babfd6322fa655df549a019aa1e69`. Each path reaches `bootz` only after both files load. The commands are for an isolated development network, not an authenticated production recovery service.

The modern port name remains `mx6ull_pa_mini`, its header `mx6ull_pa_mini.h`, and its base DTS `imx6ull-pa-mini.dts`. The supplied MINI V2.2 schematic establishes **one RJ45, on ENET2**, with an **SR8201F at MDIO address 1**. Earlier MINI revisions use LAN8720 at the same address. We will use that wiring, not the EVK's two-port assumptions.

Hardware documentation is not a claim of a working link. Chapter 19's vendor source selects an SMSC driver, and our initial build disables that network path. Section 22.7 compares the SR8201F-VB datasheet's `0x001cc816` ID with the modern RTL8201F driver's match table and enables `CONFIG_PHY_REALTEK` as a concrete candidate. Confirm the fitted PHY's actual ID and behavior before a V2.2 network test. A source/datasheet ID match is stronger than a similar name or generic fallback, but still does not establish complete silicon compatibility, RMII timing or measured packet transfers.

24C.1 The real use case

The policy has three outcomes:

1. Local kernel and DTB load, and U-Boot hands control to Linux.
2. Local loading or a returning `bootz` fails; U-Boot tries the network path.
3. The network path also fails; U-Boot returns to the prompt.

This cannot recover a board on which U-Boot itself does not start. Nor does it detect that Linux later panics or cannot mount root: once control passes to the kernel, this script is no longer running. Watchdog and boot-attempt recovery are a separate subject in Chapter 24F.

24C.2 What this chapter is not

TFTP supplies **kernel and DTB only**. The example root filesystem remains on MMC. If that rootfs is broken, loading the kernel by Ethernet does not repair it. Chapter 24's NFS-root workflow changes that assumption; a self-contained recovery initramfs is another design.

There is also no authenticity check on loose `zImage` and DTB files. TFTP has no image authentication. Chapter 24E explains why hashes alone do not solve this.

24C.3 Files changed in this chapter

File	Responsibility
`configs/mx6ull_pa_mini_defconfig`	Network, FEC/PHY, filesystem and scripting support.
`arch/arm/dts/imx6ull-pa-mini.dts`	U-Boot control-DT wiring for FEC, MDIO and PHY.
`include/configs/mx6ull_pa_mini.h`	Guarded local/network environment helpers.
Existing host TFTP root	Matching `zImage` and `imx6ull-pa-mini.dtb`.

The port must already initialize MINI Ethernet clocks and pads and qualify the revision's PHY. This is v2026.04 migration work, separate from the old vendor configuration. A network command's existence does not establish that the hardware is usable.

24C.4 Enable U-Boot network commands

Merge this configuration fragment into the Chapter 22 port:

```
CONFIG_NET=y
CONFIG_DM_ETH=y
CONFIG_FEC_MXC=y
CONFIG_PHYLIB=y
CONFIG_DM_ETH_PHY=y
CONFIG_CMD_NET=y
CONFIG_CMD_PING=y
CONFIG_CMD_DHCP=y
CONFIG_CMD_TFTPBOOT=y
CONFIG_HUSH_PARSER=y
CONFIG_HUSH_OLD_PARSER=y
CONFIG_CMD_MMC=y
CONFIG_CMD_FS_GENERIC=y
CONFIG_CMD_BOOTZ=y
CONFIG_CMD_FAT=y
```

```
CONFIG_FS_FAT=y
CONFIG_CMD_EXT4=y
CONFIG_FS_EXT4=y
```

For a **preV2.2 LAN8720 MINI**, `CONFIG_PHY_SMSC=y` selects the named LAN8710/LAN8720 driver; clocks, reset and negotiation still need testing. For the supplied **V2.2 SR8201F design**, retain Chapter 22's source-checked `CONFIG_PHY_REALTEK=y` candidate, then compare the actual MDIO ID and test the chip-specific behavior. Do not enable another platform's RTL8201F timing adjustment merely to make a link appear. In particular, do not transplant the vendor hook's unchecked register `0x1f` write of `0x8190` as a universal PHY fix.

`CMD_FS_GENERIC` provides the generic `load` used below; FAT/ext4 support handles the chosen filesystem. `CMD_BOOTZ` starts the loose ARM zImage. `CMD_DHCP` is optional for the static example, but permits the separate test in 24C.10. Retain 24B's DM GPIO and i.MX pinctrl support for reset ownership.

In the prepared Ubuntu Bash terminal, run `. ~/imx6ull/scripts/env.sh`, require success, and use the established `arm-none-linux-gnueabihf-` compiler. Reconfigure and build out of tree as in 24B.3/24B.9; do not edit global shell setup.

24C.5 Device Tree check: FEC, PHY, and reset

The MINI V2.2 route is documented in schematic sheets 1 and 3, CORE sheets 5-6, and guide pp274-275:

Signal	MINI route
RMII MAC	ENET2; ENET1 signals are exposed on P4, not a second onboard RJ45.
MDIO / MDC	GPIO1_IO06 / GPIO1_IO07 muxed to ENET2 management signals.
PHY address	1, from straps; unchanged across the documented PHY replacement.
Active-low reset	SNVS_TAMPER8 / GPIO5_IO08, net `ENET2_RST`.
RMII reference	ENET2_TX_CLK to the PHY TXC input; vendor setup supplies 50 MHz from the SoC.

The RJ45 symbol and some differential-side nets retain `ENET1` names on sheet 3. Follow the **MAC-side ENET2 nets** into U11; the connector label is not a second MAC connection. Likewise, the guide's "two RJ45" sentence is an ALPHA carry-over, not MINI wiring.

Chapter 22 already supplies these routes and uses the FEC driver's **MAC-node** reset properties. Keep that working integration as the starting point. The fragment below is an **alternative PHY-uclass reset arrangement**, not another reset owner to paste beside it. To study it, remove the MAC's `phy-reset-*` properties first, then merge these properties into the existing `mini_phy` node. Do not duplicate PHY nodes or change reset ownership on hardware without a reviewed reason.

```
/ {
    aliases {
        ethernet0 = &fec2;
    };
};

&fec1 {
    status = "disabled";
};

&fec2 {
    pinctrl-names = "default";
    pinctrl-0 = <&pinctrl_enet2 &pinctrl_enet2_reset>;
    phy-mode = "rmii";
    phy-handle = <&mini_phy>;
    status = "okay";

    mdio {
        #address-cells = <1>;
        #size-cells = <0>;

        mini_phy: ethernet-phy@1 {
            reg = <1>;
            reset-gpios = <&gpio5 8 GPIO_ACTIVE_LOW>;
            reset-assert-us = <100000>;
        };
    };
};

&iomuxc_snvs {
    pinctrl_enet2_reset: enet2-reset-grp {
        fsl,pins = <
            MX6ULL_PAD_SNVS_TAMPER8__GPIO5_IO08 0x1b0b0
        >;
    };
};
```

`pinctrl_enet2` must already contain the documented RMII and management signal muxing. The fragment omits that group and clock initialization deliberately; a DT node alone does not configure the SoC clock direction. The 100000 us assertion mirrors the vendor code's 100 ms low pulse, **not a verified SR8201F timing requirement**. No

post-release delay is invented here: add the fitted PHY's required delay after checking its specification and measured supply/reference-clock startup. The reset pad-control value also requires electrical qualification.

In v2026.04, the Ethernet PHY uclass consumes these PHY-node reset properties with `DM_ETH_PHY` and `DM_GPIO`. FEC also supports the older **MAC-node** `phy-reset-gpios`, `phy-reset-duration` and `phy-reset-post-delay`, whose delays are milliseconds. Do not specify both paths for the same GPIO or mix their units.

GPIO5_IO08 is an SNVS/tamper pad: use the i.MX6ULL SNVS pin-function header and **IOMUXC-SNVS**, as above, not the main controller's older i.MX6UL pad offsets. The reset line must not also be a recovery button, strap reader or C routine's independently requested output. Choose one reset owner when migrating away from the vendor's raw-GPIO routine.

24C.6 MAC address: do not ignore it

A valid Ethernet MAC is six bytes, nonzero and unicast. All-ones is broadcast and is invalid. Valid syntax does not imply ownership or uniqueness.

The modern fragment deliberately makes **ENET2 Ethernet sequence 0**, using `ethaddr`, with ENET1 disabled. Hardware MAC number 2 does not mean environment sequence 1. Sequence 1 uses `eth1addr`; inspect the built aliases and selected device rather than carrying over an EVK `ethprime`. The vendor header's legacy `CONFIG_FEC_ENET_DEV=1` selects the second hardware FEC, not this driver-model sequence contract.

Prefer an assigned, unique factory address from a documented source. A read-only SoC address may be usable if it is actually provisioned; a chip UID is not automatically a MAC allocation. An EEPROM/secure element/PMIC is only an identity source when the board has one with a defined record and provisioning process.

For an isolated lab, this is an **example locally administered unicast address**, not a shared default for every board:

```
=> setenv ethaddr 02:7a:60:00:00:01
```

Bit 1 of the first byte marks local administration; bit 0 is clear for unicast. Allocate a different address for each board on the lab network and check for collisions. Do not copy a vendor's public prefix, and do not `saveenv` merely to run a network test. Existing environment protection may reject changing an already assigned `ethaddr`; do not enable blanket overwrite as a workaround. Chapter 24D uses an explicit identity policy and checks assignment errors.

24C.7 Manual Ethernet test

Use an isolated subnet that does not conflict with the host's VPN or existing adapters. `192.168.7.1` for the host and `.2` for one board are examples, not universal addresses.

Inspect the host's current adapters using `ip addr show`. Use Chapter 24's existing service setup rather than reinstalling packages or changing the host globally. With a qualified board and an interruptible autoboot:

```
=> setenv ipaddr 192.168.7.2
=> setenv serverip 192.168.7.1
=> setenv netmask 255.255.255.0
=> ping ${serverip}
```

There is no predicted board log here. Record the actual selected device and result. A failed ping can also be host ICMP filtering; it does not uniquely diagnose hardware. If initialization reports no controller/PHY, fix the driver, clock, reset, address and mux before attempting fallback.

24C.8 Manual TFTP test

Place the matching kernel and OS DTB in the **already configured** TFTP root from Chapter 24. `imx6ull-pa-mini.dtb` is the modern example's OS-tree name; the vendor environment instead chooses `imx6ull-alientek-emmc.dtb`. Neither U-Boot's control DT nor a renamed unrelated DTB substitutes for a kernel-compatible MINI OS tree. Creating a directory does not configure the server to use it. Do not disable the host firewall globally; diagnose the service and its allowed traffic. TFTP uses UDP port 69 for the initial request and a server-selected transfer port afterward.

The example RAM slots are:

Slot	Address	Qualification
Kernel	`0x82000000`	Kernel payload and self-decompression must not overlap reserved data.
OS DTB	`0x83000000`	DTB and later growth/relocation need space.

Check `bdinfo`, actual file sizes, available DDR, U-Boot relocation/stack/malloc and boot memory limits. These addresses are not guarantees for every memory size or large image. A successful load does not mean the buffers were safely placed.

For a manual read-to-RAM test:

```
=> setenv kernel_addr_r 0x82000000
=> setenv fdt_addr_r 0x83000000
=> tftpboot ${kernel_addr_r} zImage
=> tftpboot ${fdt_addr_r} imx6ull-pa-mini.dtb
```

Inspect each result before proceeding. `filesize` describes the most recent successful load, not both files; failure may leave old RAM and old metadata. The automated scripts below use return status, not stale `filesize`, to decide whether boot is permitted.

24C.9 Local boot command

The complete header block in 24C.11 defines the commands. Read its local path first:

```
if mmc dev ${mmcdev} && mmc rescan; then
    if load mmc ${mmcdev}:${bootpart} ${kernel_addr_r} zImage && load mmc ${mmcdev}:$
{bootpart} ${fdt_addr_r} ${fdtfile}; then
        if run set_local_args; then
            bootz ${kernel_addr_r} - ${fdt_addr_r}
        else
            false
        fi
    else
        echo Local image load failed; false
    fi
else
    echo MMC unavailable; false
fi
```

This is the **body of `local_boot`**, not a Bash program or a prompt assignment. `&&` stops after the first failure. Each failure branch ends in `false`, so an `echo` cannot accidentally convert failure into success. A `bootz` that returns reports its own status to the caller.

The `-` means no initrd. This example chooses the **MINI TF slot on USDHC1** as local media, conventionally U-Boot `mmcdev=0` in the vendor board code. The eMMC core's USDHC2 is conventionally `mmcdev=1`; a NAND core does not gain eMMC merely by changing that variable. Confirm the modern port's device sequences before use. Optional SDIO WiFi at P2 shares USDHC1 with TF, so do not fit/use that module while relying on this TF path.

The illustrative `rootdev=/dev/mmcblk0p2` in the header must be checked against **Linux's** MMC numbering. U-Boot `mmcdev=0` does not prove Linux `mmcblk0`. A measured PARTUUID is often clearer; obtain it from the intended root partition, never invent one. To use eMMC instead, coordinate device, partitions, root arguments and matching kernel/DTB; do not merely change one number.

24C.10 Network boot command

The default network path deliberately uses static settings. It reapplies all three addresses before attempting TFTP, so no earlier DHCP attempt silently supplies different values. Both network loads must succeed before arguments are set and `bootz` is called.

DHCP is an optional **separate** test:

```
=> setenv autoload no
=> dhcp
```

`autoload=no` requests network configuration without the automatic boot-file transfer. If DHCP fails, printing “using static IP” does not restore overwritten or cleared variables. Explicitly reset `ipaddr`, `serverip`, `netmask` and any stale `gatewayip` before a static retry, as `net_boot` does below. If DHCP succeeds, its TFTP server and boot filename still need to match your intended server/payload; do not silently trust arbitrary DHCP boot options.

24C.11 Fallback boot command

This is a complete macro definition for the two loading paths and their dispatcher. Merge it into the existing header, retaining unrelated board defaults. Replace the `normal_boot` placeholder from 24B, and remove duplicate address/filename keys. Keep the non-booting `recovery_boot` until a real recovery image exists.

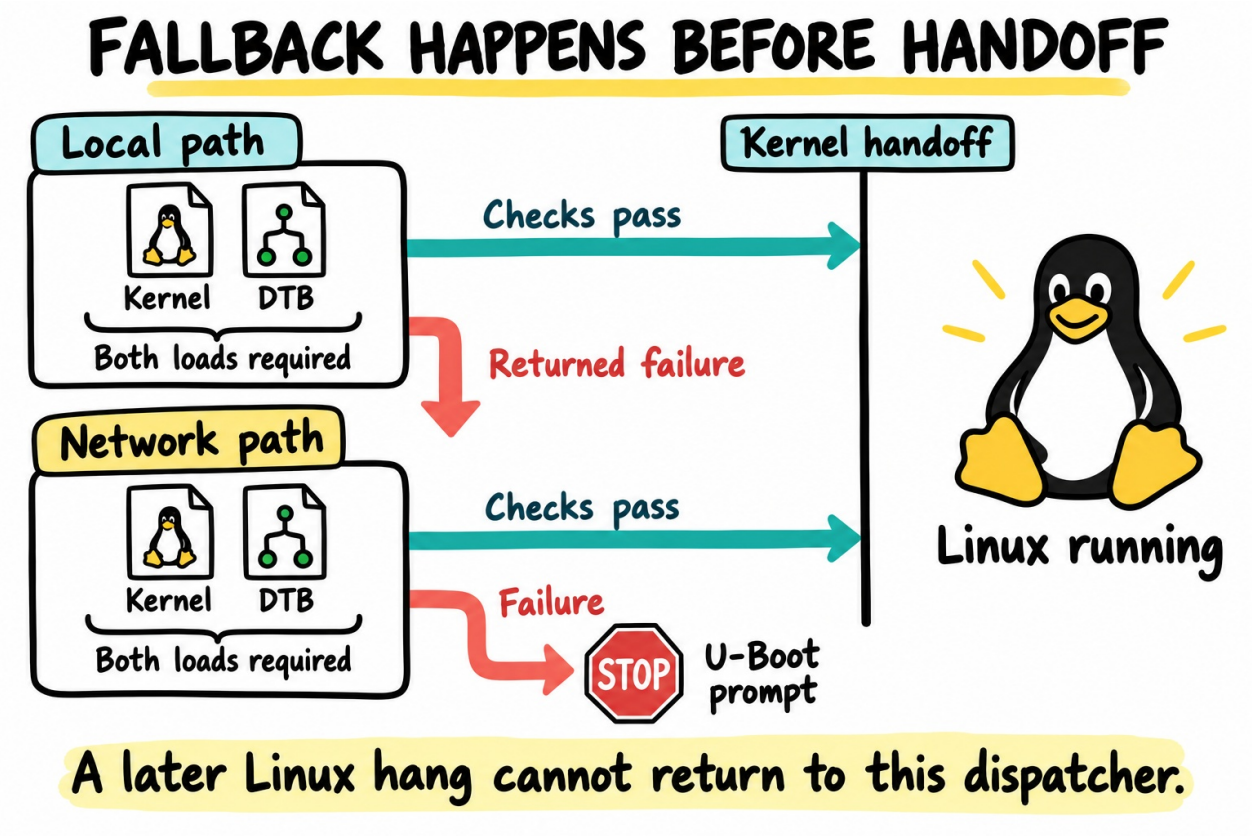
```
#define PA_MINI_NET_FALLBACK_ENV \
    "kernel_addr_r=0x82000000\0" \
    "fdt_addr_r=0x83000000\0" \
    "fdtfile=imx6ull-pa-mini.dtb\0" \
    "bootfile=zImage\0" \
    "mmcdev=0\0" \
    "bootpart=1\0" \
    "rootdev=/dev/mmcblk0p2\0" \
    "lab_ip=192.168.7.2\0" \
    "lab_server=192.168.7.1\0" \
    "lab_mask=255.255.255.0\0" \
    "set_local_args=setenv bootargs console=ttymxc0,115200 " \
        "root=${rootdev} rw rootwait\0" \
    "local_boot=echo Trying local MMC; " \
        "if mmc dev ${mmcdev} && mmc rescan; then " \
            "if load mmc ${mmcdev}:${bootpart} ${kernel_addr_r} zImage && " \
                "load mmc ${mmcdev}:${bootpart} ${fdt_addr_r} ${fdtfile}; then " \
                "if run set_local_args; then " \
                    "bootz ${kernel_addr_r} - ${fdt_addr_r}; " \
                "else false; fi; " \
            "else echo Local image load failed; false; fi; " \
        "else echo MMC unavailable; false; fi\0" \
    "net_boot=echo Trying static-IP TFTP; " \
        "if setenv ipaddr ${lab_ip} && setenv serverip ${lab_server} && " \
            "setenv netmask ${lab_mask} && setenv gatewayip; then " \
            "if tftpboot ${kernel_addr_r} ${bootfile} && " \
                "tftpboot ${fdt_addr_r} ${fdtfile}; then " \
                "if run set_local_args; then " \
```

```
                    "bootz ${kernel_addr_r} - ${fdt_addr_r}; " \
                "else false; fi; " \
            "else echo Network image load failed; false; fi; " \
        "else echo Network setup failed; false; fi\0" \
    "fallback_boot=if run local_boot; then true; " \
        "else echo Local path returned failure; run net_boot; fi\0" \
    "normal_boot=run fallback_boot\0"
```

Append this macro to `CFG_EXTRA_ENV_SETTINGS` once. Use `CONFIG_USE_BOOTCOMMAND=y` and `CONFIG_BOOTCOMMAND="run normal_boot"` from 24B. When board policy is enabled, its late hook selects either `run normal_boot`, `run recovery_boot`, or a blocked command. **Do not replace that hook's gate with an unconditional fallback bootcmd.**

Successful kernel handoff does not return. Fallback can run after a failed load or a `bootz` error returned before handoff; it cannot run after a Linux failure. The `true` branch preserves success in a host mock or another returning command; it is not a "Linux finished" message.

24C.12 Make failure visible



Fallback handles failures returned before kernel handoff. Both paths must load their complete image pair and pass argument checks. A later Linux hang requires a separate reset and recovery policy.

Log attempts at path boundaries, then keep the underlying load/PHY error. The code emits "Trying local MMC", "Local image load failed" or "MMC unavailable", and "Trying static-IP TFTP" as appropriate. Those are source strings, not promised serial output from this board.

Distinguish three observations in your notes: which path was attempted, whether each payload loaded, and whether handoff occurred. A TFTP transfer alone establishes none of the kernel's rootfs, driver or application results.

24C.13 Optional: fetch a boot script

A legacy `boot.scr` is executable command input. Its CRC is not authentication. A trusted lab may use one, but fetching a script from an untrusted server grants that server whatever commands the build exposes, including potential storage writes. A signed kernel FIT does not authenticate an unrelated script.

For an isolated, deliberate lab only, the following is the complete content of `boot.cmd`, to create in the editor:

```
if tftpboot ${kernel_addr_r} zImage && tftpboot ${fdt_addr_r} imx6ull-pa-mini.dtb && run
set_local_args; then
    bootz ${kernel_addr_r} - ${fdt_addr_r}
else
    echo Script image load or argument setup failed
fi
exit 1
```

The final `exit 1` is outside the conditional. It reports failure if either load or argument setup fails, or if `bootz` returns instead of handing control to Linux. Successful kernel handoff never reaches it. In this pinned old-hush multiline `source` path, a nested conditional ending in `false` can still return status 0 to its caller, even though it correctly blocks handoff. Do not use that structure to report packaged-script failure. The pinned exit reference describes leaving the innermost script; the single-line `run` helpers in 24C.11 are unchanged.

Using the host `mkimage` built from the pinned source:

```
mkimage -A arm -T script -C none -n "pa-mini lab net script" -d boot.cmd boot.scr
```

With `CONFIG_CMD_SOURCE` and `CONFIG_LEGACY_IMAGE_FORMAT` deliberately enabled for this legacy lab script, a **single guarded command** is:

```
=> if tftpboot ${scriptaddr} boot.scr; then source ${scriptaddr}; else echo Script load
failed; false; fi
```

Define and qualify a separate `scriptaddr` first, for example `0x84000000` only after the RAM map is checked. Do not default to executing remotely supplied scripts in production. Required FIT signatures, trusted configuration policy and a verified boot chain require a separate security design, not simply this `source` command with a different image nearby.

24C.14 Key Notes

- Local-first fallback responds to load failures or a boot command returning before handoff. It cannot diagnose a later Linux failure or recover a bootloader that never starts.
- Every path gates both payload loads and argument setup. Explicit failure status prevents an error message's successful `echo` from making stale RAM bootable.
- TFTP supplies the kernel and DTB here, not the MMC root filesystem. Network loading does not repair a corrupt root or authenticate loose images.
- ENET2 hardware numbering differs from driver-model Ethernet sequence numbering. The sole-interface `ethaddr` contract, unique MAC allocation and revision-specific PHY/reset/clock path remain distinct requirements.
- Static retry must restore actual address variables after DHCP. A message announcing static fallback does not restore state that an earlier attempt changed.
- Filesystem loads need a qualified RAM and root-partition mapping even when they do not write storage. Remote legacy scripts carry executable authority and can bypass a separate signed-image path.

24C.15 Pitfalls

- **Semicolon-only loads.** A later successful command can mask failure and boot stale data.
- **Last-command status.** `echo` returns success; explicit failure branches must end in failure.
- **DHCP retry state.** A message does not restore static settings.
- **Rootfs assumption.** TFTP kernel/DTB does not fix MMC rootfs corruption.
- **Reset domains and units.** PHY-node microseconds differ from legacy FEC-node milliseconds.
- **Public or duplicate MAC.** Use an owned factory address, or unique locally administered lab addresses.
- **Authentication.** Loose payloads and legacy boot scripts are not trusted merely because they transfer successfully.

24C.16 Going deeper

- Command Kconfig: `load`, networking, `bootz`, hush and legacy script options.
- Network commands and BOOTP/DHCP: return paths and `autoload` behavior.

- TFTP implementation: transfer ports, loading and retries. The pinned tree has no separate `dhcp.rst` or `tftpboot.rst`; inspect `cmd/net.c` for those commands.
- Environment documentation: network variables and defaults.
- FEC driver and PHY uclass: actual consumers of reset properties.
- SMSC PHY driver and PHY matching/fallback: named LAN8720 support is not evidence of SR8201F qualification.
- Vendor board header and board setup: historical controller, clock and PHY assumptions to audit during migration, not a tested V2.2 Ethernet result.

Previous: Chapter 24B: U-Boot board policy with GPIO and I2C

Next: Chapter 24D: Board identity and variant selection in U-Boot

Chapter 24D: Board identity and variant selection in U-Boot

An EEPROM byte can select a display DTB. It can also select the wrong one after a partial factory write or a read from the wrong bus. Before that byte controls boot, we need to know what record it belongs to, whether the record is complete, and whether the combination describes hardware we support.

We will define a small binary identity record, validate it, and publish a DTB/FIT choice only after validation succeeds. This continues the **U-Boot v2026.04** baseline, commit `88dc2788777babfd6322fa655df549a019aa1e69`, and the `mx6ull_pa_mini` port. Chapter 24E packages those choices in one FIT.

The supplied MINI V2.2 and CORE schematics contain **no factory identity EEPROM**. Our 64-byte record is a proposed example format for an **optional external EEPROM** on the documented I2C1 route, not a vendor record to discover on an unmodified board. The display connector and available panel models are documented MINI hardware options. No example here programs EEPROM or fuses.

This is a v2026.04 host-study/modern-migration example. Chapter 19’s vendor board names and old I2C API are not interchangeable with this port’s names and driver-model helpers. A stock MINI can use an explicitly selected, matching OS DTB without pretending that it has this EEPROM.

24D.1 The real problem

Fix one MINI baseboard revision and **one core assembly** first. Within that assembly, the proposed late selector changes only the optional display description, not DDR, boot storage or the single ENET2 port. The guide documents these panel choices at RGBLCD J1 (pp267-268):

LCD option	Selected MINI attachment	OS DTB to implement	FIT configuration
0	No display	`imx6ull-pa-mini-no-lcd.dtb`	`conf-no-lcd`
1	Optional 4.3-inch **ID4342**, 480 x 272	`imx6ull-pa-mini-lcd43.dtb`	`conf-lcd43`
2	Optional 7-inch **ID7016**, 1024 x 600	`imx6ull-pa-mini-lcd70.dtb`	`conf-lcd70`

These are filenames for OS trees to develop and test, not supplied or bench-qualified DTBs. The same-size **ID4384** (4.3-inch, 800 x 480) and **ID7084** (7-inch, 800 x 480) are different options and are **not** accepted by these two entries. No size alone determines full display timing, supply sequence or the attached touch controller's address/driver.

MINI sheet 2 documents RGB888, I2C2 touch signals, backlight `BLT_PWM` on GPIO1_IO08 and a hardware `RESET` shared with the SoC. Do not copy the vendor file's inherited software LCD reset on GPIO5_IO09 onto this board. Its three SGM3157 switches isolate LCD_DATA7/15/23, which also serve ROM boot straps and panel ID signals. The proposed EEPROM avoids implementing that separate panel-ID switching sequence; its LCD byte is an assembly declaration, **not an automatic measurement of the fitted screen**.

The guide pp238-239 documents NAND cores with 256 MiB DDR and eMMC cores with 512 MiB DDR; the supplied CORE schematic shows the storage alternatives and the 512 MiB DDR part. Choose the early image/DCD for the actual core before these late examples. Do not combine those assemblies into the display table or treat baseboard V2.2 as a RAM size.

A late EEPROM read cannot choose DDR timing that the Boot ROM already needed. The pinned EVK path initializes DDR using the image's DCD, before `board_late_init`. Families with different DDR may need distinct early images or an independently engineered early-selection mechanism. Do not claim this late hook enables arbitrary RAM variants in one binary.

24D.2 Three ways to identify a board

Source	Useful information	What must be established
Dedicated GPIO straps	A few assembly options	Mux, input ownership, pull resistors, settled readings and reserved states.
I2C EEPROM	Revision, options, MAC, serial	Actual part, address scheme, record format and factory process.
Documented read-only chip identity/fuse data	Provisioned chip identity	Board mapping and authorized allocation of fields. A UID alone does not encode peripherals.

Our proposed add-on is an **AT24C02-class 256-byte device**, at 7-bit `0x50`, using a one-byte offset. It connects to MINI P4 pin **43 for I2C1 SCL** and pin **42 for I2C1 SDA**, with 24B's control-DT alias making that bus U-Boot sequence 0. Pin 44 belongs to I2C2 SDA,

not I2C1. UART4 must not own those pads. The baseboard already pulls these nets up to 3.3 V; choose a compatible device/module and account for any additional pull-ups. The address and record are our design assumptions, not detected factory hardware.

Microchip's AT24C02C addressing specification describes that part's eight-bit word address and package-dependent address pins. A 24C04 may put upper address bits in the slave address; larger parts commonly use two-byte offsets. Check the exact part and strapped address rather than substituting a generic EEPROM name. The optional TMP102 at `0x48` can share this bus if both modules' electrical/address contracts are satisfied; neither is fitted by default.

For a muxed bus, use the child bus sequence or resolve the intended bus by its control-DT node. No code below reads or writes a PMIC.

24D.3 Do not let the identity format grow by accident

A format is a byte-level contract, not a C structure written to EEPROM. This proposed version occupies exactly 64 bytes starting at offset zero:

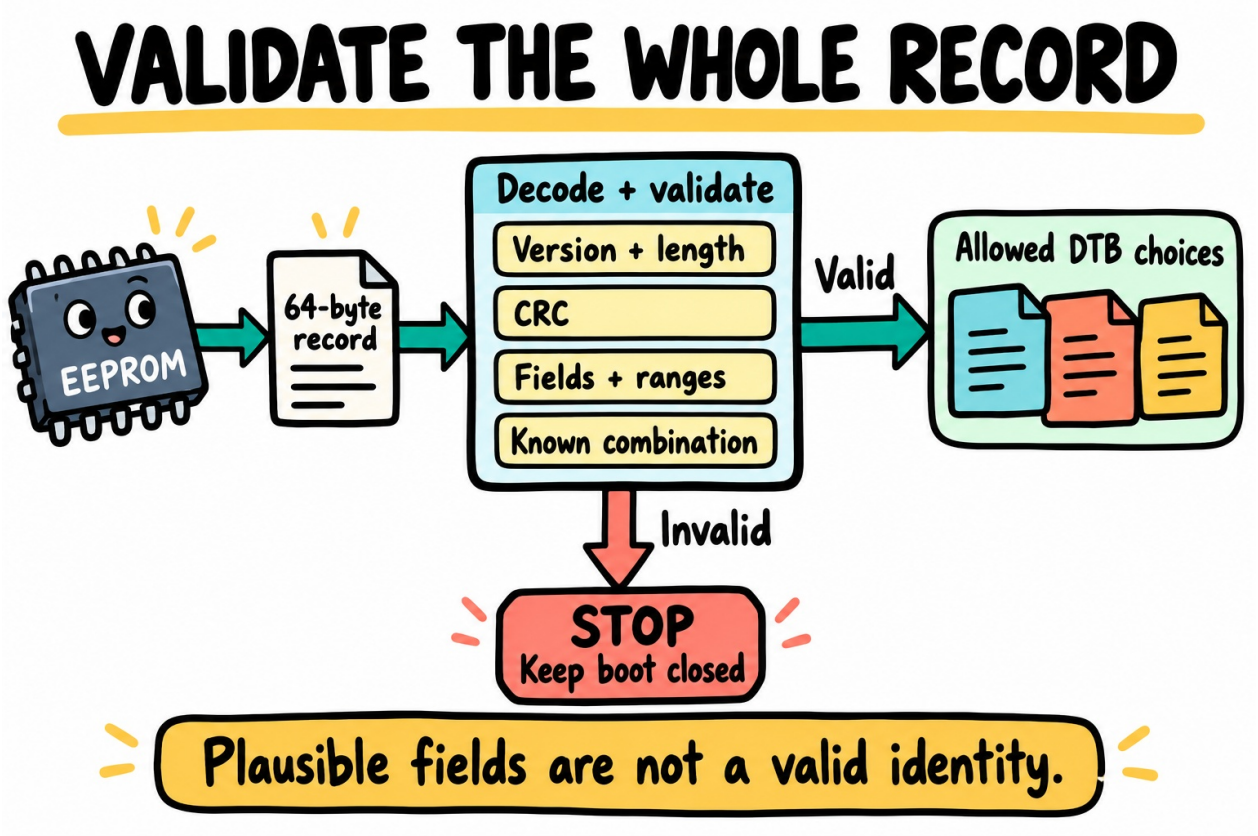
Offset	Bytes	Encoding / rule
`0x00`	8	Exact bytes `PAMID01` followed by NUL.
`0x08`	1	Format version 1.
`0x09`	1	Reserved, zero.
`0x0a`	2	Record length 64, little-endian.
`0x0c`	1	Example hardware-profile revision 1 only; not the MINI PCB silkscreen revision.
`0x0d`	1	LCD option 0, 1 or 2.
`0x0e`	1	Ethernet option 1: the single onboard MINI ENET2 RJ45.
`0x0f`	1	Reserved, zero.

Offset	Bytes	Encoding / rule
`0x10`	6	MAC in transmitted byte order.
`0x16`	10	Reserved, all zero.
`0x20`	16	Nonempty ASCII serial: 1-15 letters/digits/hyphens, NUL terminated, remaining bytes zero.
`0x30`	12	Reserved, all zero.
`0x3c`	4	Little-endian CRC-32 over bytes `0x00..0x3b`.

`hw_rev=1` names this proposed fixed-core/fixed-baseboard profile. It is not a conversion of “V2.2” to an integer, and `eth_opt=1` does not mean ENET1 or identify the PHY silicon. The profile’s bill of materials must specify the baseboard revision, core and supported PHY software separately. This record alone cannot establish working SR8201F support.

The CRC is U-Boot `crc32(0, data, 60)`: reflected CRC-32/ISO-HDLC, polynomial `0xedb88320`, standard initial/final complement. The known check value for ASCII `123456789` is `0xcbf43926`. State coverage, seed and byte order so the factory generator and bootloader agree.

CRC detects accidental corruption; anyone able to rewrite the record can recompute it. This is not authenticated identity. Factory programming, readback verification, power-failure handling and write protection are separate procedures. An incomplete record is invalid even if some fields look plausible; do not salvage an unverified MAC or revision from it.



Validate the complete record before trusting any field to select hardware. CRC detects corruption, not authenticated identity. The EEPROM is an optional lab addition, absent from the supplied stock MINI schematics.

24D.4 Files changed in this chapter

File	Responsibility
`configs/mx6ull_pa_mini_defconfig`	Late init and driver-model I2C; commands only for diagnosis.
`arch/arm/dts/imx6ull-pa-mini.dts`	EEPROM bus/pads in U-Boot's control DT.
`board/myorg/mx6ull_pa_mini/mx6ull_pa_mini.c`	Decode, validate, map and publish the identity.
`include/configs/mx6ull_pa_mini.h`	Boot helpers consume the selected filename/ configuration.

Keep manufacturing identity separate from user boot preferences. The code overwrites the volatile selection on each boot; it does not save auto-detected values.

24D.5 Enable the useful configs

Retain 24B's `BOARD_LATE_INIT`, `DM_I2C`, `SYS_I2C_MXC`, control-DT/pinctrl support and hush. The MAC helper example also assumes the network/environment support from 24C. `CMD_I2C` is a diagnostic convenience, not required by the decoder.

For **separate DTBs**, retain `CMD_BOOTZ`, `CMD_MMC`, `CMD_FS_GENERIC` and the chosen filesystem from 24C. For **FIT**, additionally enable:

```
CONFIG_FIT=y
CONFIG_FIT_FULL_CHECK=y
CONFIG_SHA256=y
CONFIG_CMD_BOOTM=y
```

For manual overlays, additionally enable:

```
CONFIG_CMD_FDT=y
CONFIG_OF_LIBFDT_OVERLAY=y
```

Neither FIT nor overlays are needed merely to read identity. No legacy `i2c_read()`/global-bus API or `setexpr` byte reinterpretation is used. Configure in a private out-of-tree build with `. ~/imx6ull/scripts/env.sh` and `CROSS_COMPILE=arm-none-linux-gnueabihf-`, as in 24B.

24D.6 Manual EEPROM read first

For a later read-only test **with the proposed EEPROM add-on fitted** and the documented I2C1 route enabled:

```
=> i2c bus
=> i2c dev 0
=> i2c probe 50
=> i2c md 50 00.1 40
```

The hexadecimal length `40` reads all 64 bytes, including CRC. `.1` matches only the proposed one-byte-offset part. Probe only the documented address; do not scan an unknown bus. An ACK at `0x50` does not prove a particular EEPROM or record exists.

Record the actual dump, then compare it with the format table. Erased `ff` bytes, all-zero bytes, a missing terminator or bad CRC should fail. There is no fabricated factory dump to copy and no `i2c mw` example.

24D.7 Board code to read identity

These are complete helpers for the proposed format, to add to the existing board C file. Merge their includes with 24B's includes at the top, once each. Parsing uses bytes explicitly, avoiding host structure padding, unaligned loads and implicit endianness.

```
#include <dm.h>
#include <env.h>
#include <i2c.h>
#include <net.h>
#include <stdio.h>
#include <string.h>
#include <linux/errno.h>
#include <u-boot/crc.h>

#define PA_MINI_ID_LEN 64

struct pa_mini_id {
    u8 hw_rev;
    u8 lcd_opt;
    u8 eth_opt;
    u8 mac[6];
    char serial[16];
};

static int pa_mini_decode_id(const u8 *raw, size_t len,
                             struct pa_mini_id *id)
{
    struct pa_mini_id decoded = { 0 };
    u32 stored_crc;
    unsigned int record_len;
    size_t i, serial_len;

    if (!raw || !id || len != PA_MINI_ID_LEN)
        return -EINVAL;
    if (memcmp(raw, "PAMID01\0", 8) || raw[8] != 1)
        return -EINVAL;
    record_len = (unsigned int)raw[10] | ((unsigned int)raw[11] << 8);
    if (record_len != PA_MINI_ID_LEN)
        return -EINVAL;
    stored_crc = (u32)raw[60] | ((u32)raw[61] << 8) |
                 ((u32)raw[62] << 16) | ((u32)raw[63] << 24);
    if (stored_crc != crc32(0, raw, 60))
        return -EBADMSG;
    if (raw[9] || raw[15])
        return -EINVAL;
    for (i = 22; i < 32; i++)
        if (raw[i])
            return -EINVAL;
    for (i = 48; i < 60; i++)
        if (raw[i])
            return -EINVAL;
    if (raw[12] != 1 || raw[13] > 2 || raw[14] != 1)
        return -ENODEV;
    if (!is_valid_ethaddr(raw + 16))
        return -EINVAL;

    for (serial_len = 0; serial_len < 16 && raw[32 + serial_len];
         serial_len++) {
        u8 c = raw[32 + serial_len];

        if (!((c >= 'A' && c <= 'Z') || (c >= 'a' && c <= 'z') ||
              (c >= '0' && c <= '9') || c == '-'))
            return -EINVAL;
```

```
    }
    if (!serial_len || serial_len == 16)
        return -EINVAL;
    for (i = serial_len; i < 16; i++)
        if (raw[32 + i])
            return -EINVAL;

    decoded.hw_rev = raw[12];
    decoded.lcd_opt = raw[13];
    decoded.eth_opt = raw[14];
    memcpy(decoded.mac, raw + 16, 6);
    memcpy(decoded.serial, raw + 32, serial_len);
    *id = decoded;
    return 0;
}

static int pa_mini_read_id(struct pa_mini_id *id)
{
    struct udevice *chip;
    u8 raw[PA_MINI_ID_LEN];
    int ret;

    ret = i2c_get_chip_for_busnum(0, 0x50, 1, &chip);
    if (ret)
        return ret;
    ret = i2c_set_chip_offset_len(chip, 1);
    if (ret)
        return ret;
    ret = dm_i2c_read(chip, 0, raw, sizeof(raw));
    if (ret)
        return ret;
    return pa_mini_decode_id(raw, sizeof(raw), id);
}

static int pa_mini_apply_id(const struct pa_mini_id *id)
{
    static const char *const configs[] = {
        "conf-no-lcd", "conf-lcd43", "conf-lcd70"
    };
    static const char *const dtbs[] = {
        "imx6ull-pa-mini-no-lcd.dtb",
        "imx6ull-pa-mini-lcd43.dtb",
        "imx6ull-pa-mini-lcd70.dtb"
    };
    const char *serial;
    u8 existing_mac[6];

    if (id->hw_rev != 1 || id->lcd_opt > 2 || id->eth_opt != 1)
        return -ENODEV;
    serial = env_get("serial#");
    if (serial && strcmp(serial, id->serial))
        return -EEXIST;
    if (!serial && env_set("serial#", id->serial))
        return -EIO;
    if (env_get("ethaddr")) {
        if (!eth_env_get_enetaddr("ethaddr", existing_mac) ||
            memcmp(existing_mac, id->mac, 6))
            return -EEXIST;
    } else if (eth_env_set_enetaddr("ethaddr", id->mac)) {
```

```
        return -EIO;
    }
    if (env_set_ulong("board_rev", id->hw_rev) ||
        env_set_ulong("lcd_opt", id->lcd_opt) ||
        env_set_ulong("eth_opt", id->eth_opt) ||
        env_set("fdtfile", dtbs[id->lcd_opt]) ||
        env_set("fitconf", configs[id->lcd_opt]))
        return -EIO;
    return env_set("identity_ready", "1") ? -EIO : 0;
}

static int pa_mini_identify(void)
{
    struct pa_mini_id id;
    int ret;

    if (env_set("identity_ready", "0") || env_set("fdtfile", NULL) ||
        env_set("fitconf", NULL))
        return -EIO;
    ret = pa_mini_read_id(&id);
    if (ret)
        return ret;
    return pa_mini_apply_id(&id);
}
```

The output structure is assigned only on complete success. The stack buffer is fixed at 64 bytes; an untrusted length never controls an allocation or read size. The CRC is mandatory and covers options, MAC, serial and all reserved bytes. All I2C/binding failures propagate.

Publication is not an atomic environment transaction: earlier diagnostic assignments can remain if a later assignment fails. `identity_ready` is therefore cleared first and set **last**. The boot gate must remain closed on any error. Inspect diagnostics as provisional until readiness is 1.

The MAC validator rejects zero and multicast/broadcast, not duplicates or addresses you do not own. Factory allocation/readback must enforce uniqueness and address ownership. For lab fixtures use unique locally administered unicast addresses, never a copied public MAC. A record with missing serial or invalid MAC fails as a whole under this example’s contract. A different product can define an explicit optional-field version; it should not silently weaken this one.

The helper assumes the sole interface is Ethernet sequence 0 and therefore sets `ethaddr`, as in 24C. A port using sequence 1 must deliberately map its address to `eth1addr`; a multi-port board needs a record with separately allocated addresses and its own validated mapping, not reuse of this one-port record.

Existing `serial#`/`ethaddr` must match, or selection stops. This catches stale or conflicting saved identity without requiring `CONFIG_ENV_OVERWRITE`. Clearing/reconciling those protected variables is a deliberate service operation, not a default example step.

In the **EEPROM-equipped example only**, insert this fragment into the **single** `board_late_init()` from 24B, immediately after successfully installing its blocked `bootcmd` and before any enabled temperature/key dispatch. Do not install this mandatory reader on a stock MINI with no EEPROM:

```
    ret = pa_mini_identify();
    if (ret) {
        printf("Board identity blocked boot: %d\n", ret);
        return 0;
    }
```

The order is now: close boot gate, identify, check temperature **if that add-on is part of this build**, read stock KEY0, then open exactly one path. Never let a later successful check undo an earlier failure. Network initialization occurs after this hook, so a successful identity can supply its MAC before normal Ethernet initialization.

24D.8 Boot with separate DTB files

Keep 24C's load-gated path; it already uses `${fdtfile}`. A stock MINI uses a deliberately chosen DTB matching its known core/baseboard/panel, without adding `identity_ready`. For the **EEPROM-identified example**, replace its `normal_boot` definition with this complete environment entry:

```
"normal_boot=if test \"${identity_ready}\" = \"1\"; then run fallback_boot; " \
    "else echo Identity not ready; false; fi\0"
```

Use the pinned hush `test` command. The readiness variable is always set to 0/1 by the identification helper; it is not a cryptographic authorization token. Loose DTB filenames on MMC and TFTP must match the allow-list, and every load must remain guarded. Do not boot whatever stale DTB happens to occupy `fdt_addr_r`.

24D.9 Boot with a FIT configuration

The Chapter 24E FIT uses the same `conf-no-lcd`, `conf-lcd43` and `conf-lcd70` names. Set `fitfile=multi.itb` and a separate qualified staging address such as `fit_addr_r=0x85000000`; see 24E's memory assumptions. Its guarded command is conceptually:

```
if test "${identity_ready}" = "1"; then
    if load mmc ${mmcdev}:${bootpart} ${fit_addr_r} ${fitfile}; then
        bootm ${fit_addr_r}#${fitconf}
    else
        echo FIT load failed; false
    fi
else
    echo Identity not ready; false
fi
```

This is a **command-body fragment**; 24E.6 supplies the complete environment macro, including argument setup, verification setting and MMC initialization. `${fitconf}` comes from the allow-list, not arbitrary EEPROM text. There is no `conf-safe` default for unknown hardware.

24D.10 Boot with a base DTB plus overlay

Use overlays only for qualified optional hardware on a common base. Identity selects an allow-listed overlay set; a filename from unvalidated EEPROM must not become a command.

U-Boot's control DT is already in use by driver model. Apply overlays to a separately loaded **OS DTB** selected as the working FDT, never to `fdt addr -c` or `fdtcontroladdr`. An OS overlay does not retroactively change U-Boot's devices or early DDR setup.

Chapter 24E supplies a complete, harmless host overlay pair and a guarded manual flow. Both label-referencing base and overlay must be compiled with `dtc -@`. Reserve room to grow the base without touching the overlay/kernel. In the pinned command, `fdt resize 2000` means **hexadecimal `0x2000`**, or 8192 extra bytes.

On apply failure, treat both buffers as unusable and reload both originals before retrying. Do not boot the base as an automatic fallback after an overlay error; a product needs an independently qualified no-overlay path to make that choice.

24D.11 What if identity is missing?

Failure	This example's behavior
Bus/read/probe error	Closed boot gate, error reported.
Wrong magic, version, length or CRC	No record fields published as trusted selections.
Unknown revision/options	Stop; do not infer a nearest variant.
Missing/invalid MAC or serial	Incomplete record rejected.
Conflicting saved identity	Stop for deliberate service reconciliation.
Environment publication failure	Keep the gate closed; do not boot partially updated state.

This table applies to a build that **requires the add-on record**. On a stock MINI, the expected absence of an EEPROM is not a damaged factory identity: omit this reader and select the documented fixed hardware explicitly. Do not invent a valid record, salvage partial fields, or force `identity_ready=1` to conceal an error in the add-on mode.

A known recovery image may be an alternative, but only if it is electrically compatible with **every** board that can reach it. Calling an image “safe” does not establish that property. The prompt itself permits bypass by an operator; production trust requires more than this mutable environment policy.

24D.12 Key Notes

- The 64-byte record is a proposed optional-EEPROM interface, not a stock MINI factory format. Fixed hardware can select its matching OS tree explicitly without invented identity data.
- Byte offsets, length, endianness, reserved bytes, bounded serial and CRC coverage form one contract. A C structure layout or plausible magic prefix is not sufficient validation.
- The decoder publishes its output only on complete success. Environment publication can be partial, so `identity_ready` is cleared first and set last while dispatch remains closed on error.
- CRC detects corruption but not forgery. MAC syntax validation also does not establish ownership or uniqueness, which belong to provisioning and allocation policy.
- Late identity can select an allow-listed display DTB or FIT configuration for a fixed early platform. It cannot choose DDR timing that ROM already used.
- Unknown or conflicting identity has no automatically safe nearest variant. Factory programming, persistent settings and any universal recovery image need independently reviewed contracts.

24D.13 Pitfalls

- **C structure as storage format.** Padding and endianness are not a wire contract.
- **Magic-only validation.** A plausible prefix says nothing about completeness or corruption.
- **Unbounded serial.** Never pass an unterminated EEPROM array to string APIs.
- **Wrong EEPROM address scheme.** Pointer width and bank-in-address behavior are part-specific.
- **Late RAM selection.** DDR is already working when this hook runs.

- **Partial publication.** Use a final readiness marker and keep dispatch closed on errors.
- **CRC mistaken for trust.** A writable record with a recomputable CRC is not authenticated.

24D.14 Going deeper

- I2C declarations and implementation: bus lookup, offsets and binding failures.
- CRC implementation: the exact factory/bootloader CRC contract.
- Ethernet validity helpers and environment API: validity versus ownership, and assignment status.
- Manual overlays and FIT overlays: OS tree construction, not control-DT replacement.

Previous: Chapter 24C: Ethernet fallback boot in U-Boot

Next: Chapter 24E: Multi-variant FIT images and DT overlays

Chapter 24E: Multi-variant FIT images and DT overlays

A FIT configuration is a set of references: this kernel, this DTB, and optionally this ramdisk or overlay sequence. Several configurations can share a kernel without duplicating it. One release artifact is possible, but every supported hardware combination still needs testing.

We will package Chapter 24D's three display choices for **one fixed MINI core/baseboard assembly**: no panel, optional ID4342, or optional ID7016 at RGBLCD J1. The board connector and panel options are documented; the matching OS trees and automatic EEPROM selector are examples to implement and qualify. We then examine overlays separately.

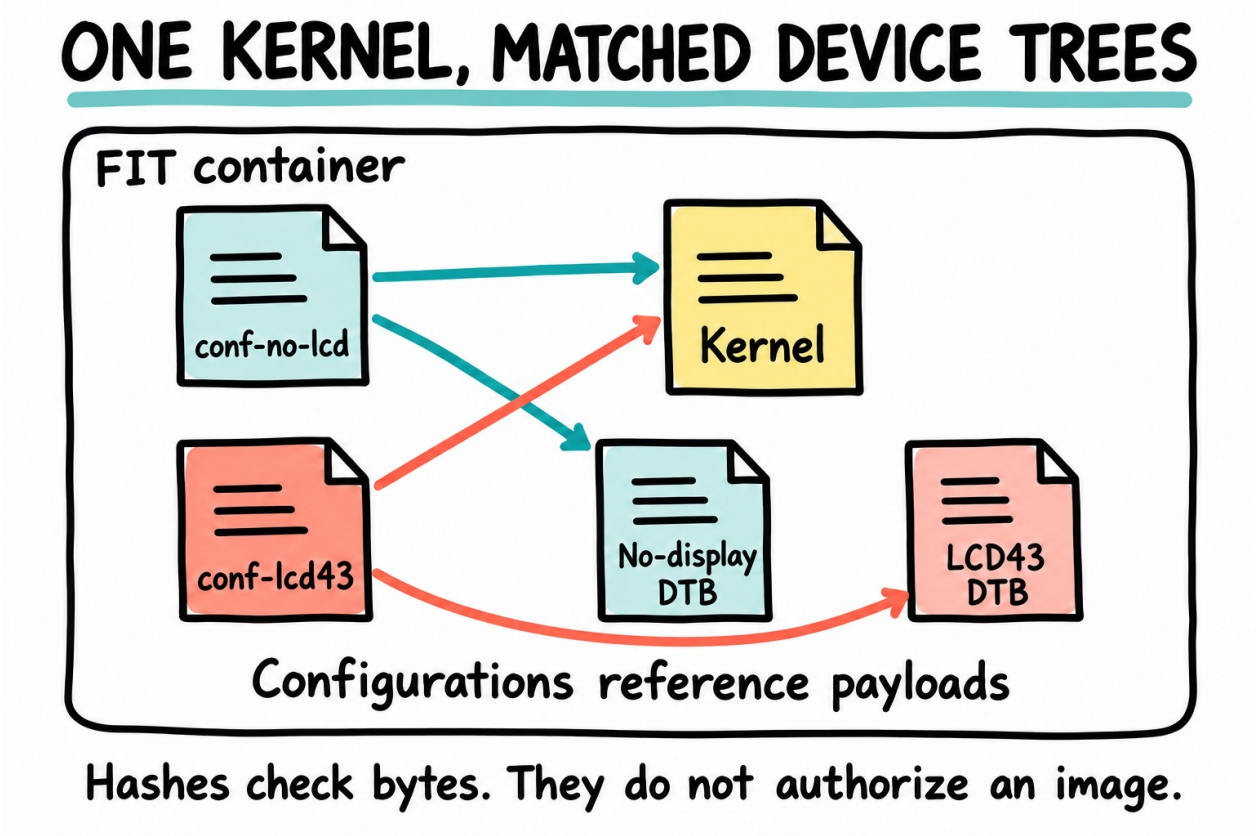
The baseline is upstream **U-Boot v2026.04**, commit `88dc2788777babfd6322fa655df549a019aa1e69`, for host study and modern MINI migration. These APIs and security options are not a promise about Chapter 19's older vendor build. FIT means Flattened Image Tree. The DT-shaped container is not the OS hardware DTB stored inside it.

24E.1 The scenario

Example LCD option / MINI attachment	FIT configuration	Payload DTB
0 / no panel	`conf-no-lcd`	`imx6ull-pa-mini-no-lcd.dtb`
1 / ID4342, 480 x 272	`conf-lcd43`	`imx6ull-pa-mini-lcd43.dtb`
2 / ID7016, 1024 x 600	`conf-lcd70`	`imx6ull-pa-mini-lcd70.dtb`

All three must share the **same chosen early DDR/storage setup**, kernel and MMC rootfs. Do not include the guide's 256 MiB NAND core and 512 MiB eMMC core in this late display selector: ROM/DCD initialization has already happened. This example includes **no initramfs**; its arguments still point to MMC. One FIT is not automatically a full recovery image or an OTA transaction.

Use the MINI RGB888/I2C2/backlight/shared-reset wiring described in 24D, then complete timing, power and touch-controller details from the fitted module’s specification. The same-size ID4384 and ID7084 panels are not aliases for the two choices above. These files are not shipped MINI DTBs and have not been bench-tested. Late FIT selection cannot change the ROM/DCD initialization already performed.



Configurations reference payloads rather than duplicating the kernel. Two of our three proposed combinations are shown here. Hashes check bytes against stored values, but trusted required signatures and boot policy are needed for authentication.

24E.2 The .its file with three configurations

Place matching `zImage` and qualified MINI OS DTBs beside this complete `multi.its`. No fallback default is specified: an explicit fixed-hardware choice or a validated add-on identity must select a supported configuration.

```
/dts-v1/;

/ {
    description = "mx6ull_pa_mini fixed-core display lab";
    #address-cells = <1>;

    images {
        kernel-1 {
            description = "Shared ARM Linux zImage";
            data = /incbin/("zImage");
            type = "kernel";
            arch = "arm";
```

```
            os = "linux";
            compression = "none";
            load = <0x82000000>;
            entry = <0x82000000>;
            hash-1 { algo = "sha256"; };
        };
        fdt-no-lcd {
            description = "MINI no-display OS DTB";
            data = /incbin/("imx6ull-pa-mini-no-lcd.dtb");
            type = "flat_dt";
            arch = "arm";
            compression = "none";
            hash-1 { algo = "sha256"; };
        };
        fdt-lcd43 {
            description = "MINI ID4342 480x272 OS DTB";
            data = /incbin/("imx6ull-pa-mini-lcd43.dtb");
            type = "flat_dt";
            arch = "arm";
            compression = "none";
            hash-1 { algo = "sha256"; };
        };
        fdt-lcd70 {
            description = "MINI ID7016 1024x600 OS DTB";
            data = /incbin/("imx6ull-pa-mini-lcd70.dtb");
            type = "flat_dt";
            arch = "arm";
            compression = "none";
            hash-1 { algo = "sha256"; };
        };
    };

    configurations {
        conf-no-lcd {
            description = "No display";
            kernel = "kernel-1";
            fdt = "fdt-no-lcd";
        };
        conf-lcd43 {
            description = "ID4342 4.3-inch display";
            kernel = "kernel-1";
            fdt = "fdt-lcd43";
        };
        conf-lcd70 {
            description = "ID7016 7-inch display";
            kernel = "kernel-1";
            fdt = "fdt-lcd70";
        };
    };
};
```

In the prepared Bash environment (`. ~/imx6ull/scripts/env.sh`, successful status), use host tools built from the pinned source:

```
mkimage -f multi.its multi.itb
mkimage -l multi.itb
stat -c '%n %s bytes' multi.itb
```

These write host artifacts, not board storage. Size depends on the actual inputs; there is no promised megabyte count. Listing the FIT is not a boot or signature-validation test.

For the unsigned example, enable `FIT`, `FIT_FULL_CHECK`, `SHA256`, `CMD_BOOTM` and the filesystem/hush support in 24D. Retain the port's ARM Linux boot support. Disable `FIT_BEST_MATCH` if omission of an explicit configuration should fail rather than attempt compatible matching. Our policy always names the configuration.

24E.3 Booting a specific configuration

Stage the FIT separately from its kernel destination: here `fit_addr_r=0x85000000`, while the kernel loads/enters at `0x82000000`. Do not stage this FIT at its own kernel load address and rely on an overlapping copy.

For a later qualified board test **with an ID4342 panel and the selected fixed core**, with correct root arguments already established:

```
=> if load mmc ${mmcdev}:${bootpart} ${fit_addr_r} multi.itb; then bootm ${fit_addr_r}
#conf-lcd43; else echo FIT load failed; false; fi
```

24E.6 defines the variables and arguments. Manual explicit selection can also study FIT on a stock MINI with no identity EEPROM: choose `conf-no-lcd` when no panel is fitted, or the configuration for the actual module. Do not manufacture `identity_ready=1` to bypass the EEPROM-mode helper. The attached `#` suffix is part of the `bootm` argument, not a shell comment. An unknown configuration fails, rather than selecting a nearby name. Without explicit selection, FIT defaults and matching options affect behavior; this FIT has no `default` property.

U-Boot resolves the selected image references, checks them according to verification policy, loads the kernel at its declared destination and prepares the OS DTB. `compression="none"` describes the outer payload; the zImage still decompresses itself. The DTB has no fixed `load` here, so U-Boot's FDT relocation policy applies. Its final address is not necessarily the FIT staging address.

Qualify usable DDR, actual sizes, kernel decompression, FDT growth/relocation, and U-Boot stack/malloc/reservations using the real RAM map and `bdinfo`. A reserved FIT window of, for example, 32 MiB at `0x85000000` is a packaging constraint, not a runtime bounds checker. These addresses do not guarantee safety for another memory size or large image.

Hashes and trusted selection

SHA-256 detects changed payload bytes when verification runs. It does not authenticate the producer: an attacker can replace payload **and hash**. Missing hash children are not equivalent to an enforced required signature. Keep example verification enabled and inspect hash nodes, but do not call an unsigned FIT trusted boot.

A product needs required signatures checked against trusted public keys in U-Boot's control DT, a verified chain to that U-Boot, and a policy binding the allowed kernel/DTB/overlay combination. `CONFIG_FIT_SIGNATURE=y` alone neither installs trusted keys nor requires signatures. Individually signed images without authenticated configuration policy can permit unwanted recombination. A valid signed configuration also does not prove it is allowed on this board; trusted identity and selection must decide that.

Mutable environment, alternate unsigned boot paths, prompt access, writable EEPROM identity and rollback need separate treatment. No key-enrollment, production-signing or fuse-programming procedure is provided. Read the pinned signature design for image versus configuration authentication.

24E.4 Detecting which variant we are running on

Automatic add-on selection happens before `bootm`, in the single 24B/24D board hook. A stock MINI has neither our proposed EEPROM nor dedicated variant straps; use an explicit fixed-hardware selection until such an identification design exists. Use `fitconf` consistently, not a second `variant` variable with different names.

Pattern A, Strap pin

Request dedicated input descriptors, check errors, allow levels to settle and sample for stability. Follow 24B.10's ownership/polarity/cleanup method, but qualify strap timing separately from mechanical-key debounce.

This is a **proposed dedicated add-on strap** mapping table, not MINI wiring or pin-reading code:

Qualified logical state	Selection
00	`conf-no-lcd`
01	`conf-lcd43`
10	`conf-lcd70`
11, changing readings or read failure	Closed gate; unsupported identity.

No GPIO1_IO09/IO10 wiring is asserted. Do not repurpose MINI BOOT_CFG1, KEY0, ENET2 reset or the LCD_DATA7/15/23 panel-ID/boot-strap circuit as generic free inputs. Reading the stock panel ID requires its own SGM3157 switching/pad-ownership sequence; this table does not implement that. Floating straps cannot be fixed with a software default. GPIO5/SNVS pads need IOMUXC-SNVS configuration if actually selected.

Pattern B, EEPROM ID

With the **optional EEPROM on MINI I2C1/P4** fitted, use 24D's DM-I2C reader and full 64-byte validation: magic, version, length, CRC, options, valid MAC and bounded serial before publishing `fitconf` and `identity_ready`. It propagates binding/allocation errors and explicitly sets pointer width. The stock schematics do not contain this EEPROM; the record's profile revision is not a MINI silkscreen or an automatic panel probe.

Do not substitute legacy global-bus `i2c_read()` or one unvalidated byte. Reading one byte to RAM and then using `setexpr.l` consumes a whole word, including three unrelated bytes. There is no reason to use the future kernel/FIT buffer as identity scratch space.

Factory provisioning and readback remain separate. Reprogramming live identity is not a normal boot step or a lab example here.

Pattern C, eFuse

Already-provisioned, documented read-only fields may identify a chip. They require the exact fuse map, field ownership and board mapping. A one-time value is not automatically confidential or authenticated board-option data, or proof of tamper resistance.

There is no generic "spare OCOTP word" in this example and no burn guidance. Reserved/security/boot fields are not interchangeable with identity storage. Use host records or schematic-qualified straps/EEPROM reads for the example.

24E.5 DT overlays, the alternative

An overlay adds or changes nodes/properties in a base DT. It suits optional blocks on a qualified common base. For MINI, keep the chosen core, storage and onboard ENET2 PHY in the base; a display or camera add-on must describe its real connector and resources. LCD touch and camera share I2C2, and display ID pins share boot-strap circuitry, so "optional" does not mean independent or conflict-free. An overlay cannot supply missing SR8201F driver support or fix an already executed DCD.

It is not a general runtime deletion mechanism: DTS source deletion directives are not arbitrary deletion operations carried through `fdt apply`.

This complete **host-only pair** changes a demonstration property, not display/PHY/power wiring. Never pass the toy base to Linux as the MINI hardware DTB.

`lab-base.dts`:

```
/dts-v1/;
/ {
    model = "Host overlay fixture, not a board DTB";
    compatible = "myorg,overlay-fixture";

    lab: lab-options {
```

```
        display-option = <0>;
    };
};
```

`lab-lcd43.dtso`:

```
/dts-v1/;
/plugin/;

&lab {
    display-option = <1>;
};
```

Compile and merge on the host:

```
dtc -@ -I dts -O dtb -o lab-base.dtb lab-base.dts
dtc -@ -I dts -O dtb -o lab-lcd43.dtbo lab-lcd43.dtso
fdtoverlay -i lab-base.dtb -o lab-combined.dtb lab-lcd43.dtbo
fdtget -t x lab-combined.dtb /lab-options display-option
```

The resulting property should be 1. Both label-referencing inputs use `-@`; base symbols and overlay fixups allow the label to resolve. Real DTS files using preprocessor includes/macros need their normal build preprocessing, not an unqualified direct `dtc` call.

For optional read-to-RAM board inspection, qualify nonoverlapping `fdt_addr_r`/`fdtoverlay_addr_r` slots with `0x2000` extra bytes after the base:

```
if load mmc ${mmcdev}:${bootpart} ${fdt_addr_r} lab-base.dtb; then
    if load mmc ${mmcdev}:${bootpart} ${fdtoverlay_addr_r} lab-lcd43.dtbo; then
        if fdt addr ${fdt_addr_r} && fdt resize 2000 && fdt apply ${fdtoverlay_addr_r};
then
            fdt print /lab-options
        else
            echo Overlay failed - reload both originals; false
        fi
    else
        echo Overlay load failed; false
    fi
else
    echo Base load failed; false
fi
```

This is a complete compound U-Boot command, shown across lines for reading, not a Bash program. It deliberately does **not** boot. `fdt addr` without `-c` selects the **working FDT**; for real boot that must be a separately loaded OS DTB, never U-Boot's live control DT at `fdtcontroladdr`.

`fdt resize 2000` parses hexadecimal: 8192 extra bytes, not decimal 2000. It changes the declared buffer size, not RAM allocation or neighboring-image bounds. Too little space causes an error, not silent truncation. Apply mutates its inputs; libfdt invalidates them on failure. Reload **both** originals before retrying. Do not boot after failure or assume the base remains untouched.

Trade-offs vs separate DTBs

Choice	Separate DTBs	Base plus overlays
Source maintenance	Share `.dtsi`; full DTBs need not duplicate source	Maintain shared labels and ordered sets.
Failure handling	Select a qualified DTB	Stop on merge failure; reload inputs.
Validation	Each complete board tree	Each supported final combination and ordering.
Shipping	Several FDT image nodes	Base/overlays can also be FIT image nodes.

There is no universal variant-count threshold. Overlays help only when shared hardware and combinations justify their additional contracts.

For **automatic FIT overlays**, define the compiled base and overlays as separate `type="flat_dt"` image nodes, each with a hash. With `OF_LIBFDT_OVERLAY`, this is a **replacement configuration fragment**, not a complete ITS:

```
conf-lcd43 {
    kernel = "kernel-1";
    fdt = "fdt-base", "overlay-lcd43";
};
```

Define `fdt-base` and `overlay-lcd43` under `images` with actual data. The first FDT is the base; later entries apply in order. The pinned interface also supports `bootm <addr>#<base-config>#<extra-config>` for deliberately selected add-ons. Do not accept arbitrary extra names from unauthenticated identity. A signed design must authenticate and permit the complete selected combination.

24E.6 Putting it together, the full multi-variant boot script

For the **EEPROM-equipped example**, use 24D's C identification and 24B's gate ordering. This complete macro replaces 24C's normal path with **local FIT boot**; it does not silently fall back to unsigned network payloads. A stock fixed-hardware MINI uses the explicit configuration route in 24E.3 instead, not this mandatory-EEPROM dispatcher.

```
#define PA_MINI_MULTI_FIT_ENV \
    "fit_addr_r=0x85000000\0" \
    "fitfile=multi.itb\0" \
    "mmcdev=0\0" \
    "bootpart=1\0" \
    "rootdev=/dev/mmcblk0p2\0" \
    "set_fit_args=setenv bootargs console=ttymxc0,115200 " \
        "root=${rootdev} rw rootwait\0" \
    "boot_multi=if test \"${identity_ready}\" = \"1\"; then " \
        "if mmc dev ${mmcdev} && mmc rescan; then " \
            "if load mmc ${mmcdev}:${bootpart} ${fit_addr_r} ${fitfile}; then " \
                "if setenv verify yes && run set_fit_args; then " \
                    "bootm ${fit_addr_r}#${fitconf}; " \
                "else echo FIT argument setup failed; false; fi; " \
            "else echo FIT load failed; false; fi; " \
        "else echo MMC unavailable; false; fi; " \
    "else echo Identity not ready; false; fi\0" \
    "normal_boot=run boot_multi\0"
```

Merge once into `CFG_EXTRA_ENV_SETTINGS`, removing conflicting definitions of the same keys. Keep `CONFIG_BOOTCOMMAND="run normal_boot"`; the late hook still installs blocked, normal or recovery dispatch. There is no default `fitconf` in this macro.

The example follows 24C's TF/USDHC1 path, without an SDIO WiFi module on the shared bus. Confirm the modern port's device sequence and Linux's actual root partition; U-Boot `${mmcdev}` does not establish its `/dev/mmcblk` name. eMMC/USDHC2 requires coordinated media/root arguments on an eMMC core. Adding a FIT ramdisk later requires coordinated ITS, argument and memory changes, not simply an extra `rootfs.cpio.gz` reference.

Mutable `verify=yes` enables corruption checking in this unsigned example. It is not required-signature enforcement or a bypass-resistant boot policy. Alternate boot routes must independently obey the final product's trust requirements.

24E.7 Key Notes

- FIT configurations share payloads through named references. One bundle simplifies release bookkeeping but does not reduce the set of hardware combinations that need qualification or make updates atomic.

- Explicit fixed-hardware selection and validated add-on identity are different dispatch modes. This container supplies no default configuration for unknown hardware.
- FIT staging, kernel destination/decompression and FDT growth have separate memory ownership. Metadata can describe destinations without checking that the chosen windows are safe.
- Required signatures, trusted keys and authenticated configuration policy address different trust questions from payload hashes. A signed combination still needs authorization for the actual board.
- Overlays mutate a separately loaded OS tree, not U-Boot's live control FDT. Symbols/fixups and application order are interfaces, and a failed apply requires fresh base and overlay inputs.
- Separate complete DTBs trade larger artifact sets for simpler final-tree selection. Overlays reduce shared-source repetition only when their combination and failure contracts justify the extra machinery.

24E.8 Pitfalls

- **One artifact mistaken for one test.** Validate every supported final combination.
- **Overlapping staging/destination.** Keep FIT clear of kernel decompression and relocated U-Boot.
- **Unknown identity mapped to a default.** No variant is automatically safe for unknown hardware.
- **Hash mistaken for authentication.** Payloads and hashes can be replaced together.
- **Control DT mistaken for OS DTB.** Do not mutate driver model's live tree.
- **Failed overlay reused.** Reload base and overlay originals.
- **Source deletion mistaken for merge behavior.** Inspect the actual resulting tree.

24E.9 Going deeper

- FIT source format and implementation: references, selection and hashes.
- FIT overlays: ordered FDT lists and extra configurations.
- Manual overlays, FDT command and libfdt apply: working-tree mutation and errors.
- FIT signatures and boot Kconfig: support versus required trusted signatures.

Previous: Chapter 24D: Board identity and variant selection in U-Boot

Next: Chapter 24F: Watchdog, bootcount, and rollback in U-Boot

Chapter 24F: Watchdog, bootcount, and rollback in U-Boot

The new kernel starts, prints a few lines, and hangs before the application runs. On an MCU you might already have a watchdog that resets this failure. But a reset alone repeats the same bad image. Something must remember that this image was being tried, and choose a different one next time.

We will give U-Boot that piece of memory. Linux will confirm success only after checking the system it actually booted. This is an **A/B policy demonstration**, not a production updater: authentication, interrupted image writes, metadata durability, and rescue remain separate requirements.

Start with the **MINI vendor baseline from Chapter 19**: identify the fitted core, reach UART1 through the board's USB_TTL connector, and establish SD or eMMC access. This chapter then develops the policy against upstream **U-Boot v2026.04**. The `mx6ull_pa_mini` target and `imx6ull-pa-mini.dts` are the book's modern migration work, not names for the vendor firmware. Do not paste these APIs/configs into the older vendor tree unchanged.

The modern path uses ROM + board-matched DCD + full U-Boot, not SPL. Chapter 22 supplies a complete, host-buildable MINI source/image candidate with documented UART/storage wiring and a source-checked PHY-driver candidate. Its physical operation remains unqualified. `CONFIG_ENV_IS_NOWHERE=y` means its **environment is RAM-only**, not that the whole firmware is merely a mock; it cannot yet implement persistent A/B. Keep the vendor baseline available and pass the separate storage/persistence gate in Section 24F.3.

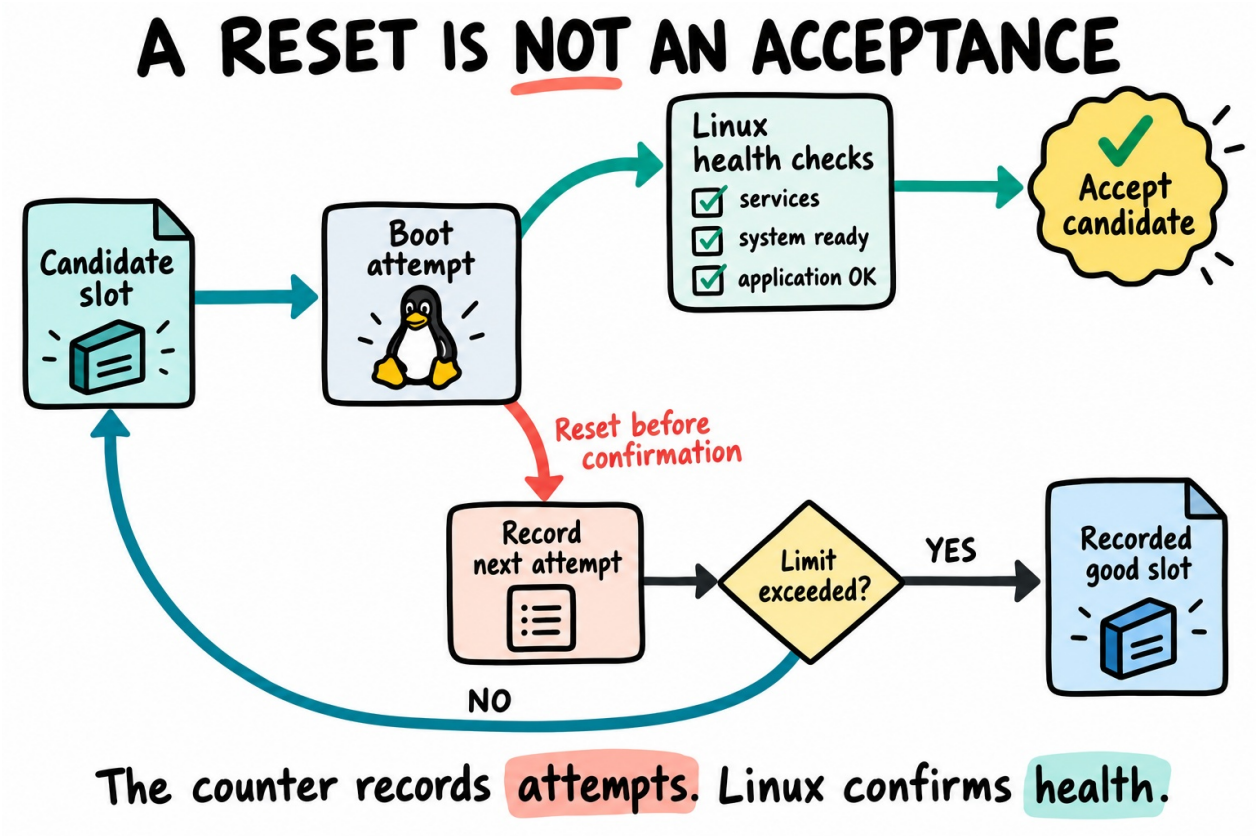
24F.1 The problem in one timeline

After storage and persistence are qualified, suppose A is known good and B is a newly written candidate:

```
persist candidate B, with attempt counter = 0
    |
next U-Boot: persist attempt 1, then try B
    |
Linux hangs before health confirmation
    |
watchdog reset: next U-Boot persists attempt 2, then tries B
    |
after the permitted attempts: choose known-good A
```

The counter records **attempts**, not detected kernel failures. A manual reset or power loss before confirmation also consumes an attempt. A hang without a reset gives U-Boot no opportunity to choose another slot.

Decision	Owner
Which complete image set to try	U-Boot policy and persistent metadata
Whether an attempt was recorded	Bootcount backend
Whether the running image is healthy	Linux health service
Whether a stalled system resets	Hardware watchdog and feeding policy



Reset gives U-Boot another decision point, not proof of success. Every candidate attempt needs recorded metadata, and Linux must identify and check the running image before accepting it. This is a policy model, not a guarantee that storage saves are durable.

24F.2 Terms

Term	Meaning here
Slot A or B	A matching kernel, DTB, and root filesystem set

Term	Meaning here
`active_slot`	Next slot U-Boot intends to boot; not proof of what Linux is running
`good_slot`	Last slot accepted by the health service
Candidate	Installed but not yet accepted slot
`bootcount`	Attempts since the candidate was armed, with backend-specific persistence
`bootlimit`	Limit compared with the incremented count using **greater than**
`altbootcmd`	Command selected instead of `bootcmd` when the limit is exceeded

With a counter initially zero and `bootlimit=3`, counts 1, 2, and 3 try the candidate. Count 4 selects `altbootcmd`. Zero or a missing `bootlimit` disables the limit check. Without the alternate command the intended automatic rollback cannot happen. See the bootcount documentation.

24F.3 Example storage layout

The supplied MINI V2.2 baseboard has a **TF/microSD socket on USDHC1**. The core-board design offers **eMMC on USDHC2 or NAND**, depending on the fitted variant; the two flash drawings do not mean both chips are installed. The SDIO Wi-Fi connector shares USDHC1 with the TF socket, so do not use both during this example. These routes are visible on MINI schematic sheets 1-2 and core sheet 4, and explained in the guide's MINI Sections 5.2.3 and 5.4.16-17.

Use an identified spare microSD as the first disposable A/B medium, leaving the vendor baseline and accepted installation recoverable. An eMMC-core owner can later qualify the eMMC user area separately. A NAND-core owner can still consider the microSD example: **the MMC/PARTUUID policy below is not a NAND or UBI updater**. Do not translate its partition numbers into NAND offsets.

Record the baseline's `mmc list`, `mmc info`, `part list`, environment settings, and Linux storage identity before planning changes. Hardware USDHC1/2 names do not establish U-Boot sequence numbers. Chapter 22's modern candidate deliberately uses

`ENV_IS_NOWHERE`: environment changes, candidate flags, slot choice and bootcount cannot survive reset. `ENV_SIZE` alone reserves no media range. Neither a vendor storage driver nor the modern MINI candidate allocates the new A/B layout for you.

Mandatory persistence and layout gate before hardware A/B:

1. Qualify the actual MINI boot path, DDR, storage wiring and device identity. Record the U-Boot device/hardware partition and the corresponding Linux device; controller numbering is not interchangeable between them.
2. Review a complete byte-range map for that medium: ROM boot image, partition-table metadata, environment copy or copies, and both slots. Reserve non-overlapping ranges explicitly, including any backup partition table. No MINI offset or partition allocation is supplied by this chapter.
3. Select a persistent environment backend supported by that storage and configure its approved device, hardware partition, offset, size and any redundancy. Replace `ENV_IS_NOWHERE` only in this qualified hardware variant; retain it for host modeling and the nonpersistent candidate.
4. Under the approved storage-qualification procedure, establish successful environment writes, readback and reload across the required reset/power cycles, plus a defined invalid-environment/recovery path. Merely seeing a RAM variable change or compiling `BOOTCOUNT_ENV` does not pass this gate.
5. Before Linux can confirm an attempt, establish its matching `fw_env.config` from that same approved layout and compare read-only results with U-Boot. Keep the modern example build's autoboot disabled until its boot command, load ranges and recovery policy have also been reviewed.

Without this evidence, stop at source inspection and host fixtures. This is a prerequisite for the hardware demonstration, not production power-cut qualification. Only after it passes may a **disposable, identified** lab medium use the proposed layout below. The table is not a disk creation recipe, and does not establish that any of these partitions already exist.

U-Boot partition	Name	Content
`mmc 0:1`	`boot_a`	A's `zImage` and `imx6ull-pa-mini.dtb`
`mmc 0:2`	`rootfs_a`	A's ext4 root filesystem
`mmc 0:3`	`boot_b`	B's `zImage` and `imx6ull-pa-mini.dtb`

U-Boot partition	Name	Content
`mmc 0:4`	`rootfs_b`	B’s ext4 root filesystem

Confirm device identity with `mmc list`, `mmc dev 0`, `mmc info`, and `part list mmc 0`. U-Boot MMC numbering is not Linux `/dev/mmcblkN` numbering. We obtain the root partition’s **PARTUUID**, its partition-table identity, instead of assuming `/dev/mmcblk0p2`.

Initially A and B may have identical contents, but then kernel version alone cannot distinguish them. Root PARTUUID and slot markers must still differ. Later use uniquely identified kernel builds.

24F.4 Files changed in this chapter

File or area	Responsibility
`configs/mx6ull_pa_mini_defconfig`	Bootcount/commands; a separately qualified persistent environment backend for the hardware variant
`arch/arm/dts/imx6ull-pa-mini.dts`	Watchdog availability and board reset wiring
`include/configs/mx6ull_pa_mini.h`	A/B defaults, merged into the existing environment once
Linux rootfs	Environment tools, matching storage description, per-slot identity, health service

In the qualified persistent variant, compiled defaults do not replace a saved environment. Inspect `printenv` after installing a build. Import only reviewed variables; do not erase unrelated identity/network settings with a blanket environment reset.

24F.5 Enable the U-Boot features

The v2026.04 fragment is:

```
CONFIG_BOOTCOUNT_LIMIT=y
CONFIG_BOOTCOUNT_ENV=y
CONFIG_WDT=y
CONFIG_IMX_WATCHDOG=y
CONFIG_CMD_WDT=y
```

```
# CONFIG_WATCHDOG_AUTOSTART is not set
CONFIG_HUSH_PARSER=y
CONFIG_CMD_BOOTZ=y
CONFIG_CMD_MMC=y
CONFIG_CMD_PART=y
CONFIG_PARTITION_UUIDS=y
CONFIG_CMD_FS_GENERIC=y
CONFIG_CMD_EXT4=y
CONFIG_CMD_FAT=y
```

`IMX_WATCHDOG`, not `WDT_IMX2`, selects this release's i.MX driver. `WDT` enables the driver-model watchdog API. Leave autostart off for initial command/policy work; later qualify the watchdog handoff deliberately. Check the generated `.config`, including filesystem support selected by commands.

`BOOTCOUNT_ENV` is not an environment-storage driver. This fragment does not turn `ENV_IS_NOWHERE` into persistent storage; a RAM-only build can host the policy example but not retain attempts. Complete Section 24F.3's gate before hardware trials, even if the vendor baseline already saves its own environment. Its storage map must be reconciled with the modern build and the new slots. Do not restore EVK MMC settings as assumed MINI values. In bootcount_env.c, `upgrade_available` nonzero enables loading the stored count and saving each increment through `env_save()`. With it zero, the backend loads zero and does not save increments; a RAM `bootcount` may still appear. This backend **does not propagate environment-save errors**. Failed persistence can defeat the attempt bound.

Build your reviewed v2026.04 MINI migration source in its own checkout, separate from Chapter 19's vendor source:

```
$ . ~/imx6ull/scripts/env.sh
$ cd ~/imx6ull/src/u-boot
$ make O="$IMX6ULL_HOME/build/uboot-ab" ARCH=arm CROSS_COMPILE="$CROSS_COMPILE"
mx6ull_pa_mini_defconfig
$ make O="$IMX6ULL_HOME/build/uboot-ab" ARCH=arm CROSS_COMPILE="$CROSS_COMPILE" -j "$
(nproc)"
```

Build success does not validate the MINI's DDR or reset circuit.

24F.6 Counter evolution across attempts

The counter is a state-machine input, not a Linux-health measurement. With an initial count of zero, an active upgrade and `bootlimit=3`, the intended selection is:

Attempt	Count after increment	Dispatch
First	1	Candidate command

Attempt	Count after increment	Dispatch
Second	2	Candidate command
Third	3	Candidate command
Fourth	4	Alternate command

This model assumes the count survives each reset and no health-confirmation step clears it. Interrupting autoboot still occurs after U-Boot's increment, so the counter does not identify why the previous attempt stopped.

Chapter 22's `ENV_IS_NOWHERE` candidate cannot provide this persistence. Section 24F.3 defines the storage and recovery evidence needed for a separate hardware variant. The table explains dispatch without requiring reader resets, environment writes or a particular storage allocation.

24F.7 A/B environment variables

Append this demonstration macro to the existing `CFG_EXTRA_ENV_SETTINGS` in `mx6ull_pa_mini.h`. Remove duplicate `bootcmd`/`altbootcmd` definitions; they need one owner. Keep the port's `CONFIG_BOOTCOMMAND` default consistent. Use it as a host fixture before qualification, or in the hardware variant only after Section 24F.3's gate. Do not enable the candidate's autoboot merely because this macro compiles.

```
#define PA_MINI_AB_ENV \
    "active_slot=A\0" \
    "good_slot=A\0" \
    "upgrade_available=0\0" \
    "bootlimit=3\0" \
    "bootcount=0\0" \
    "ab_mmc=0\0" \
    "select_slot=if test \"${active_slot}\" = \"A\"; then " \
        "setenv bootpart 0:1 && setenv rootpart 0:2; " \
    "elif test \"${active_slot}\" = \"B\"; then " \
        "setenv bootpart 0:3 && setenv rootpart 0:4; " \
    "else echo Invalid active slot; false; fi\0" \
    "boot_selected=if run select_slot && mmc dev ${ab_mmc} && " \
        "mmc rescan && part uuid mmc ${rootpart} rootuuid; then " \
        "if setenv bootargs console=ttymxc0,115200 " \
            "root=PARTUUID=${rootuuid} rw rootwait ab.slot=${active_slot}; then " \
        "if load mmc ${bootpart} ${kernel_addr_r} zImage && " \
            "load mmc ${bootpart} ${fdt_addr_r} imx6ull-pa-mini.dtb; then " \
            "bootz ${kernel_addr_r} - ${fdt_addr_r}; " \
        "else echo Slot image load failed; false; fi; " \
        "else echo Boot arguments unavailable; false; fi; " \
    "else echo Slot selection or MMC failed; false; fi\0" \
```

```
    "rollback=if test \"${upgrade_available}\" = \"1\" && " \
        "test \"${active_slot}\" != \"${good_slot}\"; then " \
        "if test \"${good_slot}\" = \"A\" || test \"${good_slot}\" = \"B\"; then " \
            "if setenv active_slot ${good_slot} && " \
                "setenv upgrade_available 0 && setenv bootcount 0 && " \
                "saveenv; then run boot_selected; " \
            "else echo Rollback metadata save failed; false; fi; " \
        "else echo Invalid good slot; false; fi; " \
    "else echo No valid pending rollback; false; fi\0" \
    "bootcmd=run boot_selected\0" \
    "altbootcmd=run rollback\0"
```

Supply qualified, non-overlapping `kernel_addr_r` and `fdt_addr_r` for the actual hardware variant. Chapter 22 supplies a source memory layout, but its load ranges still require physical DDR and overlap qualification for this A/B workflow. No additional RAM addresses are guessed here. Device 0 is fixed in both partition mappings: changing `ab_mmc` alone does not change those mappings.

The `&&` operators matter. A failed kernel or DTB load must not call `bootz` with stale bytes from a previous attempt. If `bootz` returns an error, the command returns to U-Boot, without accepting the image. This demo does not immediately reset or try another slot after a load failure; manual recovery or a qualified watchdog supplies the next decision opportunity.

The alternate path checks that a candidate is pending before changing slot state. Recovery and power policy in Chapters 24G/H must override **both** commands, not just the normal path.

24F.8 Mark slot B as candidate

Only after Section 24F.3's gate passes and all B components are written, flushed, and verified, with accepted A preserved, arm B:

```
pa-mini=> setenv active_slot B
pa-mini=> setenv good_slot A
pa-mini=> setenv upgrade_available 1
pa-mini=> setenv bootcount 0
pa-mini=> saveenv
```

Reset only after a successful save. `good_slot A` is valid here because the lab already accepted A. On an A update while B is good, reverse the roles; never hard-code A as the accepted slot in an updater.

One save groups these fields into one environment blob. It does **not** make the storage write power-cut atomic. Candidate metadata is the final step, not the first step before copying files.

24F.9 Linux marks the boot good

First prove that Linux's environment tools access the same bytes as U-Boot. The modern NOWHERE candidate has no persistent environment bytes for these tools; do not install or run a hardware writer against a guessed location. There is a concrete vendor starting point to inspect: the supplied public vendor tree's `include/configs/mx6ull_alientek_emmc.h` defines fallback MMC environment device 1 (commented USDHC2), hardware partition 0 (user area), byte offset `12 * 64 KiB = 0xC0000`, and size `8 KiB = 0x2000` in its MMC branch. These are **that source target's defaults**, not authorization to use them on a microSD, NAND variant, changed build, or repartitioned eMMC. In that vendor tree, `mmc_get_env_dev()` in `arch/arm/cpu/armv7/mx6/soc.c` derives the device from the boot controller when booted from SD/MMC; the board mapping can adjust it. Thus an SD-booted session need not save to the header's device 1. Check the generated configuration and this runtime mapping before deriving Linux's path. The `/etc/fw_env.config` shape below requires the approved matching layout:

```
<matching Linux block device/hardware partition> <approved byte offset> <approved
environment byte size>
```

Derive the real device, hardware partition, offset, size, and redundant copy from your generated config and `env/mmc.c` board overrides. Offsets are bytes, not sectors; a partition path changes the origin. eMMC boot partitions are not user-area partitions. Compare read-only `fw_printenv` with U-Boot `printenv` before enabling a writer. Stop on CRC/default-environment warnings or a mismatch.

The rootfs builder must install three files under `/etc/boot-health/`: `slot` (A or B), `root-partuuid` (the deployed root partition identity), and `kernel-release` (intended `uname -r`). Give distinct builds distinct releases. This script requires `findmnt`, `blkid`, `flock`, environment tools, and a product-specific `/usr/libexec/product-health`; a minimal BusyBox image may not contain them.

```
#!/bin/sh
# /usr/sbin/mark-boot-good; run as root after product readiness.
set -efu
die() { echo "mark-boot-good: $*" >&2; exit 1; }
exec 9>/run/boot-metadata.lock
flock -x 9
slot=
for word in $(cat /proc/cmdline); do
    case "$word" in
        ab.slot=*)
            [ -z "$slot" ] || die "duplicate slot token"
            slot=${word#ab.slot=}
            ;;
    esac
done
case "$slot" in A|B) ;; *) die "invalid booted slot" ;; esac
[ "$slot" = "$(cat /etc/boot-health/slot)" ] || die "rootfs slot mismatch"
[ "$(uname -r)" = "$(cat /etc/boot-health/kernel-release)" ] || die "kernel mismatch"
rootdev=$(findmnt -n -o SOURCE --target /)
```

```
case "$rootdev" in /dev/*) ;; *) die "unsupported root device" ;; esac
uuid=$(blkid -s PARTUUID -o value "$rootdev")
[ -n "$uuid" ] || die "root has no PARTUUID"
[ "$uuid" = "$(cat /etc/boot-health/root-partuuid)" ] || die "wrong mounted root"
[ "$slot" = "$(fw_printenv -n active_slot)" ] || die "metadata slot mismatch"
pending=$(fw_printenv -n upgrade_available)
case "$pending" in 0) exit 0 ;; 1) ;; *) die "invalid pending flag" ;; esac
test -x /usr/libexec/product-health || die "health check missing"
/usr/libexec/product-health || die "product unhealthy"
batch=$(mktemp /run/boot-good.XXXXXX)
trap 'rm -f "$batch"' EXIT HUP INT TERM
printf 'good_slot %s\nupgrade_available 0\nbootcount 0\n' "$slot" > "$batch"
fw_setenv -s "$batch" || die "metadata write failed"
[ "$(fw_printenv -n good_slot)" = "$slot" ] || die "good slot readback failed"
[ "$(fw_printenv -n upgrade_available)" = 0 ] || die "pending readback failed"
[ "$(fw_printenv -n bootcount)" = 0 ] || die "count readback failed"
```

The batch requests one update rather than three independent writes. All updater/health writers must share the lock; this script cannot serialize a writer that ignores it. The checks suit a direct block-device ext4 root, not NFS, overlay, or device-mapper roots without adaptation.

These identity checks catch integration mistakes, not malicious substitution. Two kernels sharing a release cannot be cryptographically distinguished this way. Signed images and a trusted update framework provide that stronger contract. Install the script through the rootfs build with executable permissions, invoke it from the actual init system after readiness, and propagate failure. Merely waiting some seconds is not a health test.

24F.10 Watchdog role

Read the reset circuit and watchdog DT properties before starting it. On the supplied core schematic, `SNVS_TAMPER9` is labeled `nWDOG` and joins the reset network; MINI’s RESET button also reaches that network. A net label is not proof that the i.MX watchdog peripheral is muxed onto that pad. Retain the established baseline reset behavior while reviewing the modern driver. UART1_CTS/GPIO1_IO18 belongs to the MINI KEY0 switch, and GPIO1_IO08 to RGB backlight. GPIO1_IO01 instead reaches GBC_KEY/AP_INT on this baseboard; an inherited EVK watchdog-output pinmux there still needs its actual net and accessory ownership reviewed, not a copied KEY0 assumption. `fsl,ext-reset-output` changes reset routing, not generic reliability. The v2026.04 i.MX driver has **no stop operation**; `wdt stop` cannot be relied on to disable a started device.

For an approved idle-board reset test, select the **name** printed by the command, not a numeric device index:

```
pa-mini=> wdt list
pa-mini=> wdt dev <name printed by wdt list>
pa-mini=> wdt expire
```

The last command deliberately resets the board. Never run it during writes or useful product work. `wdt start 10000; sleep 20` is not a starvation proof: U-Boot's cyclic scheduler can keep feeding during sleep or console waits. See wdt usage and `drivers/watchdog/wdt-uclass.c`.

Qualify the whole handoff: U-Boot starts the watchdog, Linux's built-in i.MX driver services the same device soon enough, and userspace takes ownership under its required `nowayout`/close policy. Choose timeouts from measured worst-case paths, including recovery. A driver/daemon may feed independently of application health: verify a stuck product really stops feeding. Acceptance is a one-time event; runtime health supervision continues later.

24F.11 Environment storage warning

With a qualified persistent backend, each candidate attempt rewrites the environment. Save errors do not stop this backend's bootcount logic.

Storage choice	What still needs proof
RAM-only `ENV_IS_NOWHERE`	The complete modern candidate can use RAM variables, but cannot retain hardware rollback metadata across reset
Single environment	Torn write can lose policy and count; no production guarantee
Redundant environment	Separate ranges, CRC/selection rules, matching Linux config, power-cut behavior
Filesystem bootcount	Exact backend/config and interrupted filesystem write recovery; a file is not automatically atomic
RTC/SNVS counter	Available backend, retained-register ownership, reset/backup-loss behavior; no automatic MINI backend assumed
Dedicated metadata	Journal/copy protocol, generation/integrity checks, ordering, invalid-state recovery

Redundancy does not prove controller caches are durable or copies cannot fail together. Preserve the accepted image and commit candidate metadata last. An environment recovered from defaults must lead to a deliberately chosen safe path, not an accidental slot choice.

24F.12 Key Notes

- Bootcount measures attempts, not diagnosed kernel failures. With an initial zero and limit three, counts one through three use the candidate and count four selects the alternate command.
- `ENV_IS_NOWHERE` cannot retain slot/count state across reset. `BOOTCOUNT_ENV` depends on a separate persistent backend and ignores save errors, so compilation does not establish an attempt bound.
- Candidate metadata is committed after the complete image set is written and verified, with the accepted slot preserved. One environment save groups fields but does not make a torn storage write atomic.
- Linux acceptance checks the running root PARTUUID, kernel release, slot identity and product readiness. Metadata's intended slot or an elapsed delay is not proof of health.
- Watchdog reset creates the next selection opportunity. U-Boot, Linux and userspace feeding ownership must agree, and routine feeding can conceal a stuck application.
- Recovery and power vetoes cover both `bootcmd` and `altbootcmd`. Temporary overrides also interact with whole-environment bootcount saves and must not become persistent policy accidentally.
- A/B rollback is not rescue or power-cut qualification. Both slots, metadata and the load path can fail, requiring an independently recoverable design.

24F.13 Pitfalls

- **Calling attempts detected failures.** Reset reasons need a separate design.
- **Booting stale RAM.** Gate both loads before the kernel jump.
- **Clearing pending from the wrong system.** Check running kernel and root, not just metadata's intent.
- **Missing resets.** A hung Linux stays hung without a reset mechanism.
- **Assuming counts persist.** This backend ignores save errors.
- **Treating RAM variables as boot metadata.** `ENV_IS_NOWHERE` cannot retain state across reset; a complete firmware build does not qualify persistence.
- **Treating A/B as rescue.** Both slots and metadata can fail; keep recovery.
- **Overriding only normal boot.** Safety choices must cover `altbootcmd` too.

24F.14 Going deeper

- Supplied `IMX6ULL_MINI_V2.2_(Mini_base_plate_schematic).pdf`, sheets 1-2, and `IMX6ULL_CORE_V2.0_(core_board_schematic).pdf`, sheets 1, 4, 5-6: storage routes, fitted-variant choices, and reset/pad ownership.
- Vendor eMMC target header: a baseline map to inspect, not a modern A/B allocation.
- Bootcount API, include/bootcount.h, and common/autoboot.c: increment and selection order.
- MMC environment backend and tool config format.
- Linux watchdog API: feeding, timeout, and close semantics, with driver-specific limitations.
- Part VIII's update-framework chapters: authenticated images and durable metadata beyond this demonstration.

Previous: Chapter 24E: Multi-variant FIT images and DT overlays

Next: Chapter 24G: Factory and recovery modes in U-Boot

Chapter 24G: Factory and recovery modes in U-Boot

The serial prompt still works, but the kernel file is gone. You do not need a complete Linux system to repair that failure. U-Boot already has a storage driver and, with the right USB controller support, can offer a service port to a host computer.

The service port is also a new way to damage the board. A host that sees a writable disk can overwrite the accepted slot, credentials, or boot metadata. We will distinguish recovery levels first, then restrict what the host can reach. Begin with Chapter 19's **MINI vendor baseline and UART1 console**. The UMS/DFU configurations and board code below target upstream **U-Boot v2026.04**, in the separate `mx6ull_pa_mini` migration, not the older vendor tree. Keep the baseline boot/recovery route available while adding service.

The MINI already has the necessary connector routes; it does not need a generic EVK carrier. Chapter 22 now supplies a complete modern MINI source candidate for UART/storage and source-checked ENET2 support, with physical operation still unqualified. It deliberately omits USB, so integrate the device-mode path here and plan a writable service target explicitly. Its `ENV_IS_NOWHERE` environment is nonpersistent; that fact is separate from USB support or the safety of exposing media.

Chapter 19's initial vendor example also disables `CMD_USB`. The USB_TTL serial bridge does not need that command; UMS/DFU below are later feature builds, not services promised by that initial UART/storage binary.

24G.1 Recovery levels

Level	Code that must already work	Boundary
Boot ROM SDP	i.MX6ULL ROM and its selected USB boot path	Can load a suitable image without working U-Boot on storage; host procedure determines whether anything is written
U-Boot USB recovery	U-Boot, DDR, USB, and storage drivers	Repairs selected later-stage contents, not a bootloader that cannot start
Linux recovery image	A separate bootable Linux image	Richer diagnostics/networking, with more dependencies

Level	Code that must already work	Boundary
Main Linux	Normal kernel and rootfs	Normal operation and controlled updates

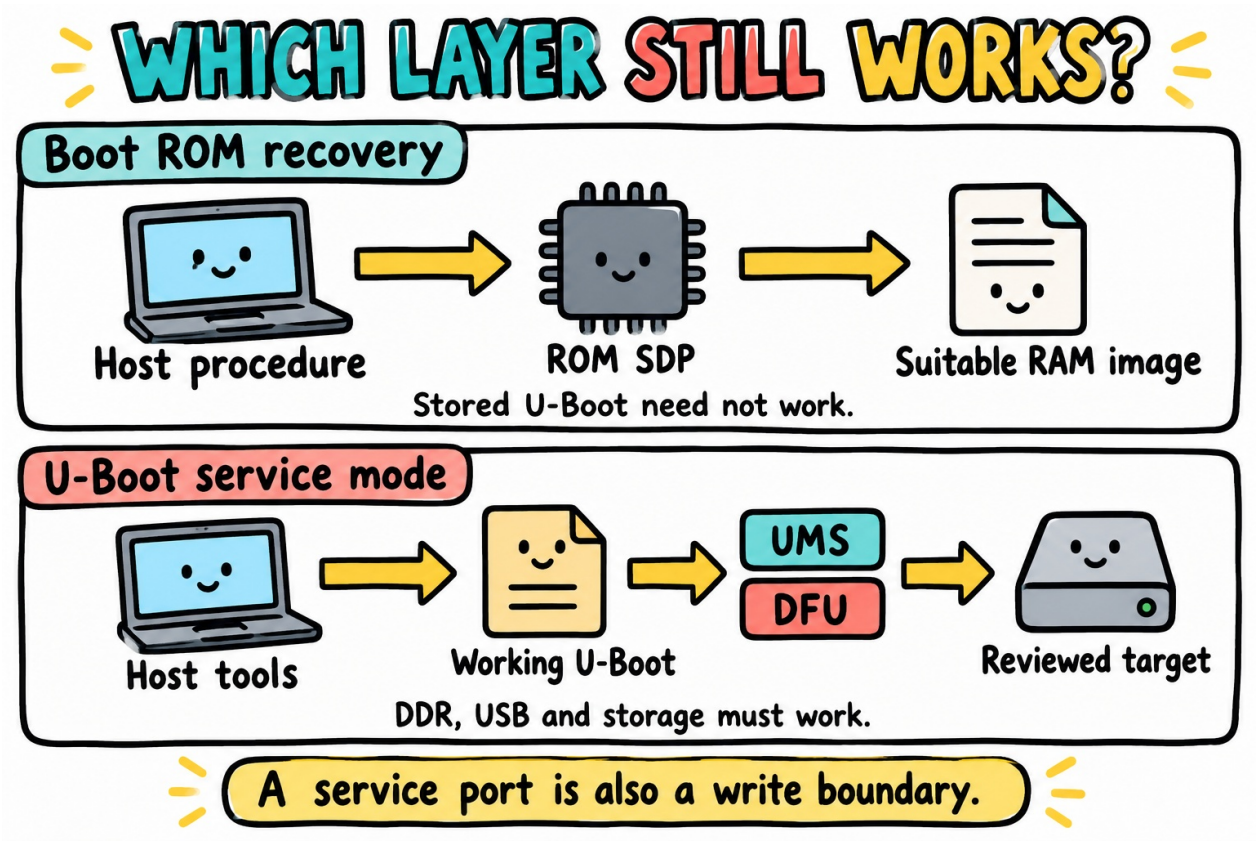
SDP is not UMS or DFU. ROM mode is selected by the board's documented boot-mode procedure. UMS and DFU are commands run by an already working U-Boot. NXP's `uuu` can orchestrate multiple stages/protocols; its presence on the host does not identify which stage the target is running. Keep Chapter 8's exact ROM access procedure documented independently.

On **MINI V2.2**, the similarly shaped USB sockets have different jobs (supplied baseboard schematic, sheet 4; guide Sections 5.4.3, 5.4.10-11):

Board connector	Actual route	Use here
`USB_TTL`, Type-C	CH340C USB-to-UART bridge, then UART1	Serial console; not ROM SDP, UMS or DFU
`USB_0TG`, Type-C	i.MX6ULL USB_OTG1 D+/D-, ID/VBUS circuitry	ROM USB boot and the intended U-Boot peripheral service connection
`USB_H0ST`, Type-A	USB_OTG2, with board 5 V on VBUS	Host peripherals; not the host-PC service cable

Use a data-capable host-to-USB_OTG cable and keep the independent USB_TTL serial console. Apply Chapter 8's reviewed supply arrangement; connecting several powered cables is not a substitute for tracing the VBUS paths. ROM entry uses the MINI's BOOT_CFG1 switch bank and USB setting from its own sheet-1 table/silkscreen, not the recovery key or EVK switch positions. KEY0 service entry only works after U-Boot reaches the key hook.

The modern image path uses ROM DCD initialization and `u-boot-dtb.imx`, with no SPL. Preserve the **MINI-matched DDR setup** established during migration. Do not copy an `spl` DFU target, raw offsets, or the EVK DCD from another board into the MINI repair procedure.



Recovery begins at the layer that still works. ROM SDP and U-Boot's UMS or DFU are different protocols with different prerequisites. The host procedure and exposed target determine which writes are possible.

24G.2 Files changed in this chapter

File	Responsibility
`configs/mx6ull_pa_mini_defconfig`	USB device controller, UMS/DFU, GPIO commands
`arch/arm/dts/imx6ull-pa-mini.dts`	USB role/PHY/power and recovery GPIO/pinctrl
`board/myorg/mx6ull_pa_mini/mx6ull_pa_mini.c`	Read the key and override both automatic boot paths
`include/configs/mx6ull_pa_mini.h`	Reviewed recovery command defaults

These are modern-port files, not vendor target filenames. Merge into the existing port. A second `board_late_init()` is a duplicate symbol, not an additional callback. Compiled defaults also do not overwrite an existing saved environment.

24G.3 Choose the recovery method

Method	Host view	Main risk
UMS	A block device, either whole medium or selected partition	Unrestricted writes within the exposed range; host automount
DFU	Named alternates through `dfu-util`	Names do not authenticate images or make writes atomic
Fastboot	Fastboot protocol and board-specific targets	Layout and commands are a separate reviewed implementation
TFTP	Files fetched over Ethernet	Network trust and verification; not a USB service

Start with enumeration and inspection on a spare medium. Prefer a bounded partition over a whole disk. DFU becomes useful when factory tools need a specific target, but an accepted image must remain outside that target. Neither protocol is a production update policy by itself.

For MINI, choose USB plus local TF/eMMC access before relying on network recovery. There is **one RJ45 on ENET2**, with PHY address 1. The guide (pp. 274-275) identifies LAN8720 before V2.2 and SR8201F on V2.2 and later; the supplied V2.2 sheet 3 shows SR8201F. The public vendor eMMC header enables `CONFIG_PHY_SMSC`, which is not evidence of correct SR8201F support. Do not promise working TFTP on V2.2. A modern network path requires the actual PHY ID, reset and RMII clock behavior to be confirmed on the board. Chapter 22 has completed the **datasheet/source ID comparison**: SR8201F-VB ID words `0x001c`/`0xc816` give `0x001cc816`, matching v2026.04's RTL8201F UID/mask entry. This makes `PHY_REALTEK` a concrete driver candidate, not an observed MDIO ID or a bench-compatible link. Similar naming alone would not establish this match. The comparison uses the CoreChips datasheet, Tables 13-14 and the pinned driver entry. Chapter 19's initial vendor example disables networking before the legacy network conditionals; it does not run an unreviewed PHY hook or require Ethernet for the first UART/storage example.

24G.4 Enable USB recovery configs

For the MINI USB_OTG1 ChipIdea path in v2026.04, review this fragment with the migrated controller/PHY setup. A vendor USB implementation is a wiring and behavior reference, not a drop-in implementation of these modern APIs:

```
CONFIG_USB=y
CONFIG_DM_USB=y
CONFIG_USB_EHCI_HCD=y
CONFIG_USB_EHCI_MX6=y
CONFIG_USB_GADGET=y
# CONFIG_DM_USB_GADGET is not set
CONFIG_CI_UDC=y
CONFIG_USB_GADGET_DOWNLOAD=y
CONFIG_CMD_USB_MASS_STORAGE=y
CONFIG_DFU=y
CONFIG_DFU_OVER_USB=y
CONFIG_DFU_MMC=y
CONFIG_CMD_DFU=y
CONFIG_DM_GPIO=y
CONFIG_CMD_GPIO=y
CONFIG_CMD_GPIO_READ=y
CONFIG_BOARD_LATE_INIT=y
```

`CI_UDC` is the device-controller implementation. In this release it depends on `! DM_USB_GADGET`; that is distinct from `DM_USB`. The ChipIdea gadget path uses i.MX EHCI setup through `usb_setup_ehci_gadget()`. Generic USB/gadget options alone are insufficient. Check the generated `.config` and the controller implementation.

Select USB_OTG1 for the service connection, with the intended peripheral role; inspect its DT `dr_mode`, PHY, VBUS handling, and any board hooks. Sheet 4 shows the OTG port's MT9700 supply switch and VBUS/ID circuitry; do not invent a spare GPIO-based VBUS enable or treat USB_OTG2 as the gadget. Do not drive host VBUS while connected as a device. Chapter 8's power and back-power checks apply to USB and the serial adapter too.

Set `USB_GADGET_MANUFACTURER`, `USB_GADGET_VENDOR_NUM`, and `USB_GADGET_PRODUCT_NUM` to identities authorized for your board/workflow. No borrowed ROM VID/PID is assigned here: ROM enumeration and a U-Boot gadget are different devices. `CMD_GPIO_READ` is required for `gpio read`.

24G.5 Manual UMS test

UMS is writable. Linux being absent does not make it safe. Before starting, identify the target medium and selected partition, ensure no target command will also access it, and disable host automount through the host's normal per-session controls. Use a dedicated lab host if that cannot be guaranteed.

Use the MINI's identified spare TF card as the first service medium; the core's eMMC is a separate target, and the MMC commands below do not expose NAND. Inventory the medium on the UART console first. Chapter 24F's table proposes a layout, not one installed by the vendor baseline or Chapter 22. Before service, qualify every exposed byte range. If using A/B, pass 24F.3's persistence/layout gate too. UMS itself does not

require a persistent environment, but RAM-only environment support does not make storage exposure safe. On that qualified disposable layout, exposing B's boot partition is bounded:

```
pa-mini=> mmc list
pa-mini=> mmc dev 0
pa-mini=> mmc info
pa-mini=> part list mmc 0
pa-mini=> ums 0 mmc 0:3
```

Do not run this until device 0 and partition 3 have been confirmed. The first `0` is the USB gadget controller argument. `0:3` is the MMC device/partition; `ums 0 mmc 0` instead exposes the **whole** medium, including raw boot and environment ranges. See UMS syntax.

On the host, compare before/after inventories, without mounting or writing:

```
$ lsblk -o NAME,PATH,SIZE,MODEL,SERIAL,TRAN,TYPE,MOUNTPOINTS
```

A partition exposed as one LUN can appear as a whole host block device containing a filesystem directly, not as `/dev/sdX1`. Disk letters and USB transport alone are not identity proof. Do not provide a guessed path to `mount`, a formatter, or an image writer.

For later authorized copying, use the exact identified LUN, only one host filesystem owner, and the approved repair procedure. Before leaving UMS, require successful writes and flushes, unmount/eject on the host, and confirm no remaining mounts. If unmount fails, stop; do not force it. Then press `Ctrl-C` in U-Boot. Never boot Linux, reset the target, or access its filesystem from another U-Boot command while the host still owns it.

24G.6 Manual DFU test

Begin with a **file target**, not raw bootloader sectors. On the same separately qualified layout, assuming B's boot partition is FAT:

```
pa-mini=> setenv dfu_alt_info 'splash.bmp fat 0 3'
pa-mini=> dfu 0 mmc 0 list
pa-mini=> dfu 0 mmc 0
```

Here `splash.bmp` is the actual target filename, not a label for an entire partition. If the partition is ext4, the target grammar must use `ext4` and the build needs the corresponding write support. FAT writes need `CONFIG_FAT_WRITE`; enable it only for an approved write workflow.

On the host, list descriptors only:

```
$ dfu-util -l
```

Require the intended device identity and alternate name in actual output. Enumeration is not a write/verification test. No `dfu-util -D`, raw offset, or bootloader write recipe is supplied here. End the session with the console's `Ctrl-C` or the approved host detach procedure, after transfers are complete. Do not assume it automatically resets or boots Linux.

DFU documentation describes `raw`, `part`, `fat`, and `ext4` forms. For MMC, raw offset/size are interpreted as block ranges by the backend; they are not interchangeable with byte addresses or `.imx` file offsets. Review `drivers/dfu/dfu_mmc.c` for the exact medium. Named targets still allow damaging writes, and `script` targets execute downloaded commands. Do not expose scripts, fuses, or secrets as a beginner recovery target.

24G.7 Add recovery environment commands

After the bounded manual test, add defaults to the existing header:

```
#define PA_MINI_RECOVERY_ENV \
    "recovery_ums=if test \"${update_allowed}\" = \"1\"; then " \
        "ums 0 mmc 0:3; else echo Recovery writes not authorized; false; fi\0" \
    "dfu_alt_info=splash.bmp fat 0 3\0" \
    "recovery_dfu=if test \"${update_allowed}\" = \"1\"; then " \
        "dfu 0 mmc 0; else echo Recovery writes not authorized; false; fi\0" \
    "recovery_boot=echo Recovery service selected; " \
        "run recovery_ums; echo Recovery service ended; false\0"
```

These are **disposable-layout defaults**, not a dynamic inactive-slot selector. Their partitions must actually exist in the separately qualified medium; they are not allocations supplied by Chapter 22. They expose partition 3 even if B later becomes accepted. Before real deployment, derive the allowed target from trusted policy or use a separate service partition. Never present this fixed mapping as A/B-safe.

The deliberate `false` leaves the automatic path stopped after service exit/failure. It must not fall through into a second normal/network boot. To choose DFU in RAM, set `recovery_boot` to the same wrapper with `run recovery_dfu`. Keep Chapter 24F's normal and alternate commands; do not add another compiled `bootcmd` definition here.

Both USB defaults are writable, so they require a fresh `update_allowed=1` decision from Chapter 24H's reviewed power policy, not merely permission to boot. Keep its fail-closed default at zero, once in the combined environment. For a supervised spare-medium lab without a monitor, an operator may set that variable **in RAM only**, after the approved stable-supply checks; this is an explicit lab authorization, not invented voltage telemetry. An unset/zero flag refuses the service. A product must refresh permission before service and handle power deterioration during it; the shell gate is not authentication or a durable-write guarantee.

24G.8 Detect the recovery key in board code

On **MINI V2.2, KEY0 connects through IMX1 B48 to UART1_CTS**, muxed here as **GPIO1_IO18, GPIO1 bit 18, active-low**. Sheet 2's R12 supplies the 10 kOhm pull-up and the switch pulls the net to ground. This agrees with the guide's MINI Section 5.4.13. Trace the baseboard connection, not the core sheet's inherited KEY aliases: B49/GPIO1_IO01 instead carries `GBC_KEY/AP_INT`, and is not this KEY0 switch.

Retain UART1_TXD/RXD for the USB_TTL console, but disable any competing UART1 CTS hardware-flow-control ownership on this pad. KEY0 must remain an input; do not use `gpio set` or `gpio clear` on it.

Add a board-owned property to the existing `/chosen` node:

```
/ {
    chosen {
        myorg,recovery-gpios = <&gpio1 18 GPIO_ACTIVE_LOW>;
    };
};

&iomuxc {
    pinctrl_recovery: recoverygrp {
        fsl,pins = <MX6UL_PAD_UART1_CTS_B__GPIO1_IO18 0x1b0b0>;
    };
};
```

Include the GPIO binding definitions and append `&pinctrl_recovery` to the port's **existing IOMUXC default pinctrl hog**. Do not replace its other groups. The GPIO property does not apply pinmux by itself. The pad value is an example to review with the board pull-up and electrical requirements. The custom property is consumed by this board code, not a new upstream binding.

```
#include <dm.h>
#include <dm/ofnode.h>
#include <env.h>
#include <hang.h>
#include <stdio.h>
#include <asm/gpio.h>
#include <linux/delay.h>
#include <linux/errno.h>

static int pa_mini_recovery_key_pressed(void)
{
    struct gpio_desc key;
    ofnode node = ofnode_path("/chosen");
    int first, second, ret, free_ret;

    if (!ofnode_valid(node))
        return -ENOENT;
    ret = gpio_request_by_name_nodev(node, "myorg,recovery-gpios", 0,
                                     &key, GPIOD_IS_IN);
    if (ret)
        return ret;
```

```
    first = dm_gpio_get_value(&key);
    mdelay(20); /* Two separated samples; not a qualified debounce filter. */
    second = dm_gpio_get_value(&key);
    free_ret = dm_gpio_free(NULL, &key);
    if (first < 0)
        return first;
    if (second < 0)
        return second;
    if (free_ret)
        return free_ret;
    return first == 1 && second == 1;
}

static void pa_mini_apply_recovery_policy(void)
{
    int pressed = pa_mini_recovery_key_pressed();
    const char *command;

    if (!pressed)
        return;
    if (pressed < 0) {
        printf("Recovery key read failed: %d\n", pressed);
        command = "echo Key state unknown; false";
    } else {
        command = "run recovery_boot";
    }
    if (env_set("upgrade_available", "0") ||
        env_set("bootcmd", command) || env_set("altbootcmd", command))
        hang();
}
```

`dm_gpio_get_value()` applies `GPIO_ACTIVE_LOW`: a pressed physical low becomes logical **1**. The error path blocks automatic boot instead of interpreting an I/O error as a released key. Call this helper from the existing `board_late_init()` only after critical power policy allows service; do not save its temporary overrides. Chapter 24F's increment/limit selection occurs later, so both commands must be overridden. In the nonpersistent modern candidate there is no persistent candidate metadata to preserve, and reset reloads defaults. Once 24F.3's gate has selected persistent storage, with `BOOTCOUNT_ENV` the temporary `upgrade_available=0` is essential: it suppresses the later **whole-environment save**, so these overrides are not persisted as part of a candidate attempt. The stored candidate flag/count remain unchanged; a service-only entry does not consume an attempt in this example. Do not invoke another `saveenv` in this path without separately reconciling that state. Recovery still depends on usable DDR and storage; it cannot repair a failure before this callback runs.

24G.9 Environment-only recovery key

For manual input inspection, v2026.04's i.MX GPIO bank names use `GPIO1_`. Confirm the actual name through `gpio status -a`, then read the physical level:

```
pa-mini=> setenv key_level
pa-mini=> gpio read key_level GPIO1_18
pa-mini=> printenv key_level
```

A pressed key should give 0 and a released key 1 here. Unlike a DT descriptor, this command does not know the board's active-low meaning. Check command success and the value, not `gpio input`'s return status, which historically returns the level and can confuse a high input with failure.

An environment script that changes `bootcmd` *inside the already executing bootcmd* does not replace the remaining commands in that execution. For a lab-only wrapper, directly branch to `run recovery_boot` or to the recorded normal command. The alternate path must also branch through that check. Use the C hook above for the integrated example; no fragile key script is installed automatically.

24G.10 Factory test mode

Factory mode checks an identified assembly; recovery repairs storage. Do not turn an I2C address scan into a pass result: an ACK does not prove chip identity, and probing every address can have side effects. GPIO status does not establish continuity or safe output operation either.

A read-only MMC inventory can be gated honestly:

```
pa-mini=> setenv factory_mmc 'if mmc dev 0 && mmc rescan && mmc info; then echo
TEST:MMC:ENUMERATED; else echo TEST:MMC:FAIL; false; fi'
pa-mini=> run factory_mmc
```

`ENUMERATED` is deliberately narrower than a destructive storage test. A fixture's PASS must correspond to a defined measurement, expected identity, and limits. Keep command status as well as printed tokens. Never print `TEST:I2C:PASS` merely because `i2c probe` ran.

A MINI factory policy separates UART1 console, identified TF/eMMC enumeration, KEY0 input, and the optional display's observations. LED0 is GPIO1_IO03, active-low; BEEP is SNVS_TAMPER1/GPIO5 bit 1, active-low through sheet 2's S8550 PNP high-side switch. Neither should be driven before reviewing its ownership/startup state. An I2C OLED or PMIC is not fitted on this baseboard, so do not require its ACK as a stock-board PASS. Ethernet remains a separate V2.2 PHY qualification, not an assumed fixture dependency. Keep serial output as the primary diagnostic channel.

24G.11 Recovery safety rules

Rule	Why
Deliberate physical entry and explicit serial message	Avoid accidental service exposure
Critical power veto before USB/storage service	Repair writes also need reliable power
Narrow, reviewed storage target	Whole-disk UMS includes raw boot/environment data
No simultaneous host/target ownership	Filesystem caches are not coordinated
Preserve an accepted image and metadata	A service protocol is not rollback
Define exit, disconnect, and watchdog behavior	A timeout/reset during transfer can tear writes
Document ROM SDP separately	U-Boot cannot repair itself if it cannot run
Restrict physical service access	A button is not authentication; storage may contain secrets

24G.12 Key Notes

- ROM SDP, U-Boot USB service and Linux recovery require progressively more working software. A U-Boot gadget cannot repair an image that prevents U-Boot from starting.
- USB_TTL is the UART bridge, USB_OTG is the intended peripheral-service route, and USB_HOST supplies host peripherals. Connector shape does not establish role or safe power routing.
- UMS grants writable block access within its exposed range. One host or target filesystem owner is required, with completed flush/unmount before ownership returns to U-Boot.
- DFU alternates describe target grammar and ranges, not authenticated or atomic updates. A file target narrows exposure, while raw block ranges and downloaded scripts enlarge it.

- Fixed partition-3 service defaults are not a dynamic inactive-slot selector. Once B becomes accepted, that same mapping can expose the accepted image.
- KEY0 dispatch overrides normal and alternate commands and suppresses the later bootcount save through a temporary candidate flag. Service exit deliberately does not fall through to boot.
- Fresh write-power permission and restricted service access are separate requirements. Enumeration proves neither storage repair nor a factory PASS, and a physical button is not authentication.

24G.13 Pitfalls

- **Confusing ROM SDP with a U-Boot gadget.** Identify who is executing.
- **Borrowed SPL offsets.** This port has no SPL; raw layouts are board-specific.
- **Wrong KEY0 bit or polarity.** Descriptor logical 1 means pressed; raw 0 does.
- **Assuming Linux absence makes UMS safe.** Host writes can still destroy data.
- **Exiting with host caches dirty.** Release the host LUN before target access.
- **Overriding only `bootcmd`.** An exceeded counter selects `altbootcmd` instead.
- **Treating enumeration as a production test.** Match each PASS to real evidence.

24G.14 Going deeper

- Supplied MINI V2.2 schematic, sheets 1-4, and the guide's **MINI** Sections 5.4.3, 5.4.10-13, 5.4.15-19: connector roles, KEY0/BEEP routes, TF/Wi-Fi sharing and the V2.2 PHY change. Use the revision's schematic when the guide's copied prose disagrees.
- UMS command and cmd/usb_mass_storage.c.
- DFU grammar and MMC backend.
- GPIO command and descriptor API.
- NXP UUU sources/documentation: read the selected script's protocols and writes before invoking it.

Previous: Chapter 24F: Watchdog, bootcount, and rollback

Next: Chapter 24H: PMIC and power policy in U-Boot

Chapter 24H: PMIC and power policy in U-Boot

The boot log stops when the storage device starts drawing current. Replacing the kernel will not fix a supply that collapses at that moment. Before calling it a software failure, ask which rail powers the device, what can observe that rail, and when the observation is possible.

On the supplied **MINI V2.2 + core V2.0 design, power comes from discrete regulators, not an I2C PMIC**. There is no fitted fuel gauge or external I2C rail monitor in these schematics. The useful stock-board example is to trace that real power tree and its reset/enable paths, then establish the stable supply arrangement for Chapter 19's vendor baseline. A PMIC or monitor is an optional hardware extension, not something to discover by guessing an I2C address.

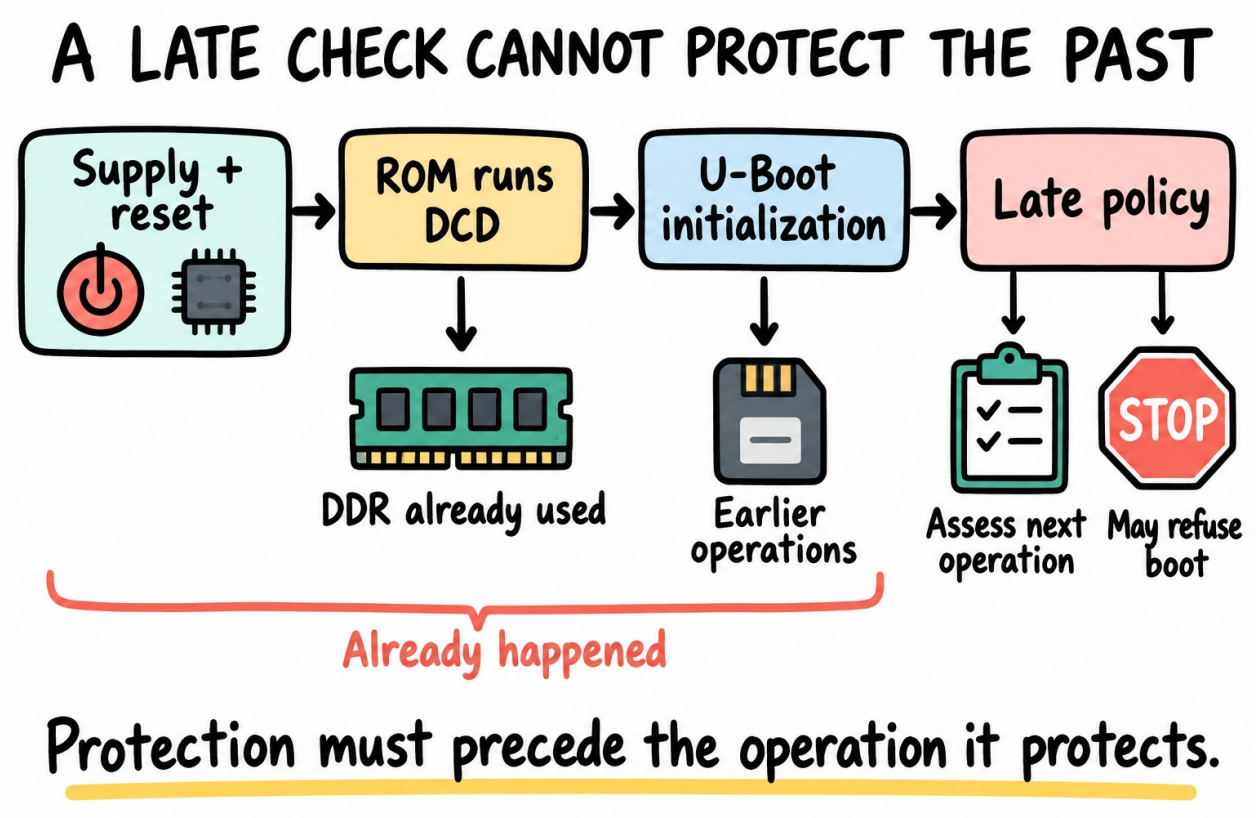
For that extension we use upstream **U-Boot v2026.04** APIs in the separate `mx6ull_pa_mini` migration. The policy can first be tested in host fixtures, without changing real rails. It is not a drop-in patch for the older vendor firmware, and the refusal stub below must not block an ordinary stock MINI. Any hardware A/B integration still needs Section 24F.3's persistence/layout gate; stable power alone does not allocate environment or slot storage.

24H.1 What U-Boot should check

Condition	Useful decision	Earliest required protection
Core or DDR supply outside specifications	Do not attempt unsafe initialization	Hardware/reset/ROM-time design, before using DDR
Storage supply unavailable	Do not load or write that device	Before storage access, possibly before board late init
Verified PMIC critical fault	Refuse further automatic boot/ service	Before the affected load or operation
Energy insufficient for an update	Block writes while normal boot may remain possible	Rechecked immediately before and during updating
Optional LCD supply fault	Leave LCD/backlight off, retain serial path if safe	Before display initialization

A PMIC status bit is not necessarily a measurement of a rail. An enable register describes a request, not proof that voltage has settled. An ADC reading also has accuracy, sample age, scaling, and operating limits.

The boot phase matters. The MINI image path is **ROM + board-matched DCD + full U-Boot**, with DDR initialized before `board_late_init()`. A late check cannot protect that first DDR access or diagnose every brownout: the CPU may already have reset before it can print. It also cannot undo storage reads already performed while loading the environment.



Put protection before the operation it protects. In this ROM+DCD path, a late power veto cannot protect the first DDR access or undo earlier device activity. The icons show phases, not fitted monitoring hardware.

24H.2 Three hardware patterns

Hardware	What software can reasonably know
Supported I2C PMIC	Documented status/faults and regulator controls; only some chips expose voltage ADCs
Fixed regulator with GPIO enable	Intended enable state and, if wired, a separate power-good input

Hardware	What software can reasonably know
ADC or battery gauge	A documented conversion/energy estimate, not automatic authorization for every load

Here is the actual starting point, from MINI sheet 4 and core sheets 1-2:

Fitted circuit	Route/responsibility	What it does not provide
MINI U6, JW5060T	DC input conversion to the 5 V supply path	No I2C voltage/status registers
MINI U4, JW5060T	DCDC_5V to DCDC_3V3; EN receives `PMIC_ON_REQ` through R51	No programmable PMIC merely because of the net name
Core U8, TMI3211	Local DCDC_3V3; its `DCDC_3V3_PG` feeds core supply enables	Not an automatically readable GPIO/voltage sample
Core U6/U11, MT3420B	ARM/SoC and DDR supplies, respectively	Not permission to adjust resistor-set rails from an I2C command
Core U10, TMI809F and reset network	Supply supervision/reset path	No bootloader log guaranteed during a brownout
MINI CR1, CR1220 backup cell	VDD_COIN_3V to the core SNVS supply path	Does not power Linux, storage or LCD; not an update-energy battery
MINI U7, XC6206	USB-powered CH340C UART supply	Serial enumeration does not prove the core rails are powered

Trace DC_IN, K1, the two USB VBUS paths, the core supply inputs, and the approved probing points before choosing the Chapter 8 supply arrangement. The USB_TTL bridge has its own supply; SGM3157 switches isolate its UART lines when the board’s 3.3 V is absent. A live host serial device or lit power LED is therefore not a complete rail qualification.

For the **stock MINI**, finish that circuit audit and keep the established vendor regulator/clock setup. Software has no documented input-voltage telemetry here. Do not add `pmic@2d` or turn an enable net into an invented “voltage OK” measurement. The vendor board source even contains PFUZE3000 code guarded for the **9x9 EVK**; that inherited code is not fitted MINI hardware.

For an **optional monitored product**, add a documented ADC/gauge/PMIC circuit with appropriate scaling, protection, address straps and fault wiring. P4 pin 43 is UART4_TXD/I2C1 SCL; pin 42 is UART4_RXD/I2C1 SDA. Pin 41 is ENET1_RXER, not SCL: these I2C signals are not a same-row header pair. MINI sheet 1 shows 4.7 kOhm pull-ups to DCDC_3V3. Check voltage compatibility, loading and UART4 ownership before using it. This gives a real bus starting point, not a specified monitor model or register map.

24H.3 Files changed in this chapter

File or area	Responsibility
`configs/mx6ull_pa_mini_defconfig`	Required I2C/GPIO/regulator support for real fitted devices
`arch/arm/dts/imx6ull-pa-mini.dts`	Real compatible, bus, pinmux, rail and consumer relationships
`board/myorg/mx6ull_pa_mini/mx6ull_pa_mini.c`	Measurement adapter and automatic-boot veto
`include/configs/mx6ull_pa_mini.h`	Fail-closed update and failure-message defaults
Board design records	Limits, timing, fault meanings, policy and test evidence

These file names belong to the modern migration. The adapter in Section 24H.7 deliberately refuses operation until implemented. Keep it in a host fixture or an explicit monitor-extension build; do not call it from the stock MINI’s boot hook and label `-ENOSYS` a detected hardware fault.

24H.4 Enable useful configs

The following is the **optional I2C monitor/regulator integration toolbox**, not a requirement to enable PMIC support on the stock MINI. Select only what the actual design needs:

```
CONFIG_BOARD_LATE_INIT=y
CONFIG_DM_I2C=y
CONFIG_SYS_I2C_MXC=y
CONFIG_CMD_I2C=y
CONFIG_DM_GPIO=y
CONFIG_CMD_GPIO=y
CONFIG_DM_REGULATOR=y
CONFIG_DM_REGULATOR_FIXED=y
```

`SYS_I2C_MXC` is the i.MX I2C-controller symbol in v2026.04. It does not select a PMIC chip driver. `DM_REGULATOR` supplies the regulator framework, and `DM_REGULATOR_FIXED` supports fixed-regulator descriptions; neither creates rails absent from the board. Enable the exact PMIC/regulator drivers after checking their Kconfig dependencies and compatible matches.

`CONFIG_CMD_POWEROFF` is optional. It adds a command only when supported by the configuration; a working platform poweroff implementation is still required. Do not assume a command can remove board input power. A supported poweroff must be qualified for rail ordering and watchdog behavior. We use a serial-visible automatic-boot stop for the policy example instead.

Build using `. ~/imx6ull/scripts/env.sh`, `ARCH=arm`, and the selected `arm-none-linux-gnueabihf-` compiler, with a separate `O=` directory as in Chapter 24F. No `.bashrc` or host-global settings are needed.

24H.5 Example Device Tree for a PMIC

Device Tree is a hardware description, not a place to invent a register map. An added PMIC's description depends on the fitted design and matching chip documentation. The table shows that evidence contract. On the stock discrete-power MINI, the correct result is **no PMIC node**:

Field	Required evidence
Bus and address	Schematic nets, address straps, 7-bit address, verified controller sequence
Compatible	A match in the selected v2026.04 PMIC driver
Register address width	Driver/datasheet: one byte, two bytes, or another protocol
Fault read	Width, polarity, latching, read-to-clear/W1C side effects

Field	Required evidence
Voltage read	Whether an ADC exists; byte order, coherent sample method, scale and units
Regulators	Real rail names, permissible ranges, enable polarity, consumers
Timing	Startup, settling, power-good, reset and shutdown order

A fictitious `myorg,teaching-pmic` compatible has no upstream driver. A node with such a name cannot make `dm_i2c_reg_read()` understand the chip. Prefer the supported PMIC driver's API and regulator bindings when available.

For a real GPIO-enabled rail, `regulator-fixed` uses the board's actual voltage, GPIO, polarity, and consumer phandle. Do not borrow GPIO4 bit 10 or label an SD rail "eMMC" without evidence. Preserve existing fixed rails and pinctrl in the modern tree. In particular, do not describe the core `PMIC_ON_REQ` supply chain as an independently switchable display regulator. `regulator-always-on` is a software constraint, not proof of voltage; `regulator-boot-on` describes boot expectations, not a measurement. Neither property grants permission to change a core/DDR rail.

24H.6 Manual PMIC read first

There is no stock MINI PMIC read to perform. Use this section only after installing the documented optional monitor/PMIC; otherwise continue with the stock circuit audit and the host policy example.

Begin with `i2c bus` and inspect the controller's actual alias/sequence. The schematic's "I2C1" need not be U-Boot bus 1. Select only the identified bus. A broad `i2c probe` sends bus transactions and is not guaranteed harmless for every device, so prefer a reviewed read to a known fitted address.

The command shape below is **not runnable until placeholders are resolved**:

```
pa-mini=> i2c dev <verified U-Boot bus sequence>
pa-mini=> i2c md <verified 7-bit address> <safe status register>.1 <byte count>
```

`.1` means a one-byte register offset; it is not the data width. A device with a two-byte offset needs the corresponding `.2` protocol, and some chips need a driver-specific transfer instead. Do not read a latch-clearing register merely to inspect it, and do not

write an unknown bit to “clear the error”. Record raw bytes and datasheet interpretation separately. No device address, fault bit, voltage conversion, or successful read is supplied as a MINI fact.

24H.7 Example power policy code

There are two contracts. The **adapter** produces one validated snapshot using chip-specific rules. The **policy** decides whether that snapshot allows continued automatic boot or update. A communication error is not an ordinary voltage value.

This board-local example is compilable **monitor-extension policy scaffolding**, not a stock MINI PMIC driver:

```
#include <env.h>
#include <hang.h>
#include <stdio.h>
#include <linux/errno.h>

struct pa_power_snapshot {
    int critical_ok;
    int update_ok;
};

static int pa_read_power_snapshot(struct pa_power_snapshot *sample)
{
    sample->critical_ok = 0;
    sample->update_ok = 0;
    /* Replace only after the board-specific measurement contract is reviewed. */
    return -ENOSYS;
}

static void pa_block_power_boot(const char *reason)
{
    printf("Automatic boot blocked: %s\n", reason);
    if (env_set("power_fail_reason", reason) ||
        env_set("upgrade_available", "0") ||
        env_set("bootcmd", "run power_fail_boot") ||
        env_set("altbootcmd", "run power_fail_boot"))
        hang();
}

static int pa_mini_apply_power_policy(void)
{
    struct pa_power_snapshot sample = { 0 };
    int ret;

    if (env_set("update_allowed", "0"))
        hang();
    ret = pa_read_power_snapshot(&sample);
    if (ret) {
        printf("Power snapshot unavailable: %d\n", ret);
        pa_block_power_boot("critical power state unknown");
        return 0;
    }
    if (sample.critical_ok != 1) {
```

```
        pa_block_power_boot("critical power condition failed");
        return 0;
    }
    if (sample.update_ok == 1 && env_set("update_allowed", "1"))
        hang();
    return 1;
}
```

Here a successful adapter return means all **required** observations were obtained coherently and interpreted against reviewed limits. Preserve negative I2C errors; do not merge two reads into a voltage until both succeed. If a multi-byte ADC can change between reads, use its documented latch/burst method. Account for signed fields, byte order, unit scaling, overflow, and sample age. A raw integer assembled as `(hi << 8) | lo` is not automatically millivolts.

For a documented register-based chip, v2026.04 offers `i2c_get_chip_for_busnum(bus, addr, offset_len, &dev)`, `dm_i2c_read(dev, reg, bytes, len)`, and `dm_i2c_reg_read(dev, reg)`. The last returns an integer byte or negative error, not an arbitrary multi-byte measurement. Set/check the actual offset length; an existing bound chip may already have its own configuration. Prefer its PMIC driver rather than overriding a device protocol behind that driver's back.

In the implemented monitor-extension build, merge the policy at the **actual operation boundary**. At board late init, if it returns zero, do not run recovery or another hook that reinstates boot. If it returns one, identity and recovery policy may continue. Earlier storage or video initialization needs its own earlier veto, or must be deferred; merely calling this helper first inside `board_late_init()` is not sufficient.

A power observation does not authorize raising CPU/DDR clocks. Such a change requires the exact silicon speed grade, voltage/frequency limits, validated sequencing, and clock/DDR design. None is established by this chapter.

24H.8 Default environment for power failure

The update-permission default is also useful on a stock MINI: it prevents writable USB service from being enabled merely by holding KEY0. Merge the following defaults into the modern environment:

```
#define PA_MINI_POWER_ENV \
    "update_allowed=0\0" \
    "power_fail_reason=not checked\0" \
    "power_fail_boot=echo Power check failed: ${power_fail_reason}; " \
        "echo Use the approved stable-power procedure before retrying; false\0"
```

Both automatic commands point to this failure path when blocked. These are RAM overrides. With `ENV_IS_NOWHERE` there is no stored candidate state; reset reloads defaults. After Chapter 24F's persistence/layout gate, clearing `upgrade_available` in RAM prevents `BOOTCOUNT_ENV`'s later counter increment from saving the **whole** environment, including

these temporary failure commands. Persistent candidate metadata remains unchanged. Do not call `saveenv` in this path. An allocation failure invokes `hang()` instead of continuing with old boot commands.

On a stock MINI supervised example build, reset `update_allowed` to `0` in the existing board hook on **every entry**, checking the `env_set()` result, but do not call the unimplemented snapshot adapter. Normal boot remains under its established policy. For a spare-medium service session, the operator may set permission to `1` **in RAM only** after the approved stable-supply checks from Chapter 8. Do not save it. This is manual authorization without voltage telemetry; unattended power-qualified updates need an implemented monitor or another reviewed hardware power contract.

Stopping automatic boot leaves a development console, not a secure or hardware-level safety barrier. A person can still issue dangerous commands, and the watchdog may reset a stopped system. Define whether the design requires a safe powered state, supervisor-held reset, or qualified shutdown. Do not loop rapid resets into an unstable supply as an assumed remedy.

24H.9 Low battery update block

A system may have enough energy for read-only operation but not for updating. This is an **optional battery-powered product extension**. The MINI's CR1220 SNVS backup cell is not that system battery and cannot authorize an image write. On a stock externally powered MINI, use the supervised supply contract above instead of fabricating battery percentages. `update_allowed` starts at 0 and becomes 1 only from a fresh qualified snapshot in the monitored design, or explicit supervised lab authorization on the stock design. It is not a persisted authorization token.

In the separately qualified persistent variant, on a normal candidate boot, `BOOTCOUNT_ENV` may save its RAM value along with the attempt. The next boot must still reset it to zero and reassess power before any updater/service uses it. Never trust a stored `update_allowed=1`.

A command wrapper can check the decision:

```
safe_update=if test "${update_allowed}" = "1"; then run do_update; else echo Update
blocked; false; fi
```

This is **only a gate**, not a supplied updater. `do_update` must be an independently reviewed implementation that refreshes the power assessment immediately before writing, monitors the update as required, verifies data, and handles interruption. Do not add a generic flashing command here.

A one-time boot voltage does not predict energy for an entire update. Battery state, source impedance, temperature, write current, hold-up time, and measurement uncertainty all affect the decision. Determine limits from the actual design; no universal 4.7/4.8 V thresholds are provided. Writable UMS also bypasses this shell gate, so block service entry or enforce write policy at the backend when power is unsuitable.

24H.10 When to fail open

Observation	Defensible policy
Required critical monitor unreadable	Treat state as unknown; block the operation that requires it
Documented critical fault asserted	Follow the board’s defined safe-state procedure
DDR power cannot be guaranteed before ROM DCD	Fix hardware/startup sequencing; late policy is too late
Optional battery gauge absent on a non-battery design	Continue only under a documented non-battery power contract
Optional display rail unavailable	Keep display off if it is safe for the rest of the board
No PMIC/monitor fitted	No invented telemetry; document hardware guarantees and software limits

Specify these rules in the board design record. “Fail closed” here describes software refusing an operation; it is not proof that stopping software makes the electrical state safe.

24H.11 Key Notes

- The documented MINI/core uses discrete regulators with no fitted I2C PMIC or rail monitor. `PMIC_ON_REQ` is a sequencing net, and the CR1220 backs SNVS rather than powering updates.
- A measurement adapter owns protocol, side effects, coherence, units and sample validity. Policy consumes a validated snapshot, not raw bytes or a transport error disguised as voltage.

- Protection must precede the operation it guards. A DDR-resident late hook cannot protect ROM/DCD accesses or undo earlier environment/storage reads.
- Boot-safe and update-safe are different decisions. Fresh `update_allowed` permission belongs at the write/service boundary and cannot be trusted merely because it was saved earlier.
- Critical power vetoes outrank key recovery and cover both automatic commands. Suppressing a whole-environment attempt save prevents temporary failure commands from becoming persistent.
- The unimplemented adapter belongs only to the monitor-extension design. Its `-ENOSYS` is not a stock-board fault, and a software stop is not proof of an electrically safe powered state.
- Rail limits and hold-up requirements come from the actual design. Lowering supplies, changing regulator setpoints or cutting power is not required to understand these policy cases.

24H.12 Pitfalls

- **A late “DDR check” advertised as early protection.** DDR was already used.
- **Invented addresses or bit meanings.** Use fitted-chip documentation.
- **Read equals harmless.** Some status reads clear evidence or require latches.
- **Enable state equals voltage.** Requests and measured power-good differ.
- **Unknown treated as zero millivolts.** Keep transport errors separate.
- **Recovery overriding a power block.** Apply priority to both boot paths.
- **Boot-time permission reused for later writes.** Refresh before the operation.
- **Optional monitor treated as mandatory hardware.** Specify the actual design.

24H.13 Going deeper

- Supplied MINI V2.2 schematic, sheet 4, and core V2.0 schematic, sheets 1-2; guide MINI Sections 5.4.3, 5.4.7, 5.4.20-21: discrete supplies, serial isolation, and SNVS backup, not an I2C PMIC.
- v2026.04 PMIC drivers and regulator drivers: supported compatibles and APIs.
- I2C API and i.MX controller.
- EVK image configuration: concrete ROM/DCD initialization, not a MINI power qualification.

- The fitted chip’s official datasheet and board supply/rail timing records; those documents are prerequisites, not values this generic chapter can supply.

Previous: Chapter 24G: Factory and recovery modes in U-Boot

Next: Chapter 24I: U-Boot display and boot screen

Chapter 24I: U-Boot display and boot screen

A lit backlight can make a completely uninitialized LCD look alive. The opposite is possible too: valid pixels are being scanned out, but the backlight is off and the screen appears black. Before adding a logo, separate those two paths: data reaches the panel through eLCDIF; light comes from a board-specific power and backlight circuit.

The supplied **MINI V2.2 baseboard already has a 40-pin RGBLCD connector, J1**, with 24-bit RGB, sync/clock, backlight control, and touch signals. What you add is a compatible RGB panel module and its cable, not a hypothetical LCD carrier. An SPI/8080 MCU-interface module cannot plug into this RGB path merely because its size matches.

Start with Chapter 19’s MINI vendor baseline and identify the actual panel module before enabling a separate display build. Chapter 19’s first vendor lab deliberately disables video; no panel is required there, and its legacy LCD-reset helper must not run during the UART/SD/eMMC starting example. We then prepare the modern boot screen using upstream **v2026.04** in the separate `mx6ull_pa_mini` migration. The old board display helpers and modern DM video APIs are not interchangeable. Without a panel, you can still inspect J1’s routes and complete the host BMP/memory examples; with a panel, its timing and power specification determine the remaining values.

24I.1 What a U-Boot boot screen really is

A framebuffer is a RAM region containing pixels. eLCDIF repeatedly reads those pixels and produces the panel’s clock, sync, and data signals. U-Boot loads a BMP into a **different** RAM region, decodes it, and writes the result into that framebuffer.

```
trusted BMP on storage -> bounded image buffer -> BMP decoder -> framebuffer
                                                                    |
                                                          eLCDIF + board wiring
                                                                    |
                                                             powered RGB panel
```

Stage	Display owner
U-Boot	Its video driver, framebuffer, and optional text/BMP output
Linux after its display driver probes	Its own driver and buffers

Linux may reset the controller/panel and redraw. Preserving the image across handoff is a separate integration project, not a property of `bmp display`. Changing Linux Device Tree does not fix U-Boot's own control DT or driver.

The MINI connection is concrete (baseboard sheet 2, core sheets 5/8; guide MINI Section 5.4.4, pp. 267-268):

MINI route	Meaning for this example
LCD_DATA16..23 -> R0..7; DATA8..15 -> G0..7; DATA0..7 -> B0..7	RGB888 wiring; the three highest color bits pass through switches
LCD_PCLK, LCD_HSYNC, LCD_VSYNC, LCD_DE -> J1	eLCDIF pixel/sync path, not SPI transfers
DCDC_5V and ground -> J1	Module supply connection; panel/module electrical limits still apply
BLT_PWM -> J1 pin 34, GPIO1_IO08 on the SoC	Backlight control input to the module, not direct LED power
RESET -> J1 pin 40	Shared board reset network, not an independent arbitrary LCD-reset GPIO
I2C2 SCL/SDA, CT_INT, CT_RST -> J1	Touch-related routes; not required for drawing a BMP

U1-U3 are SGM3157 switches on LCD_DATA23/15/7. These pads also carry boot configuration bits, while compatible ATK panels use the corresponding lines for panel ID. R11 connects `SGM_CTRL` to LCD_VSYNC in the supplied schematic; R10's DE option and the R15-R17 bypasses are marked DNP. Preserve that startup isolation and switching arrangement. Do not bypass the switches or change strap resistors as a display troubleshooting shortcut.

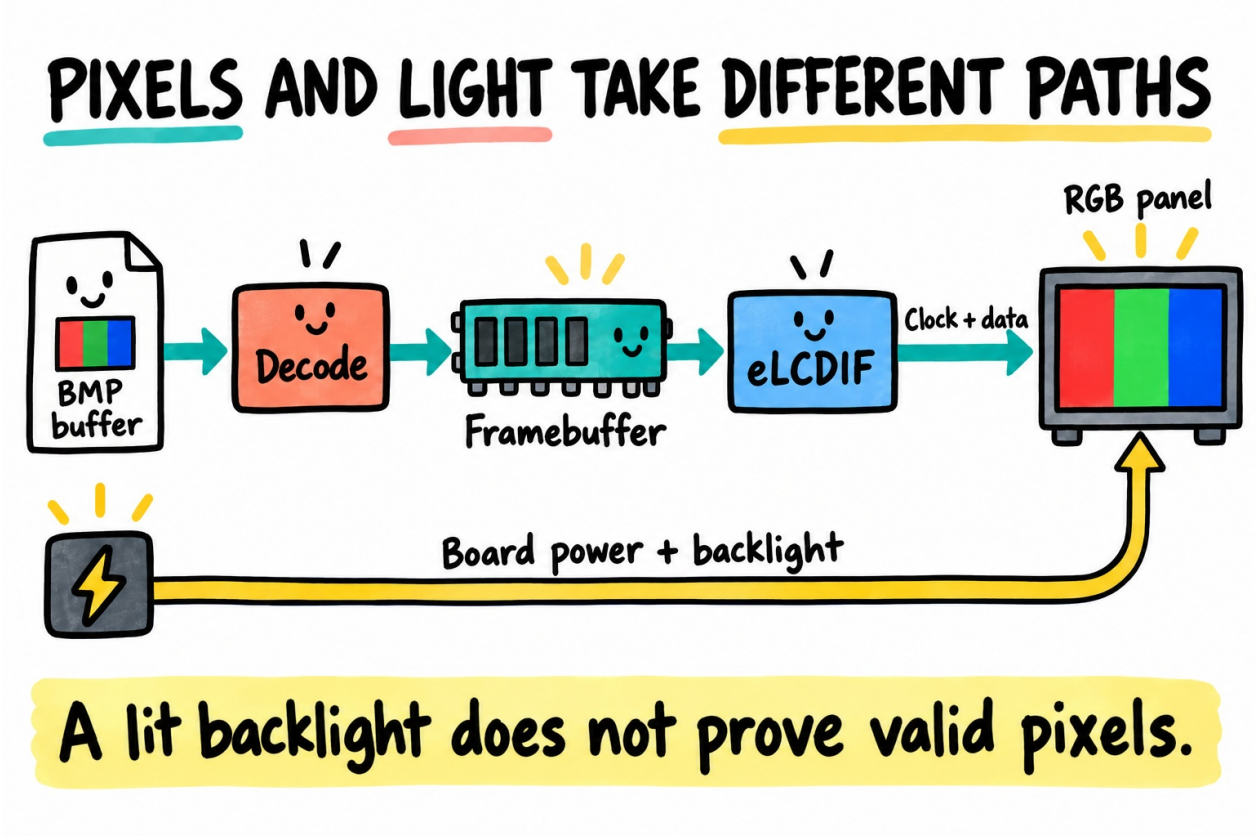


Image bytes, scanout pixels, and panel light follow different paths. A black screen can therefore have more than one cause. The drawing is a data-flow model, not a carrier schematic or proof of display operation.

24I.2 Files changed in this chapter

File or area	Responsibility
`configs/mx6ull_pa_mini_defconfig`	DM video, MXS/eLCDIF driver, BMP formats, filesystem commands
`arch/arm/dts/imx6ull-pa-mini.dts`	Verified display timing/pinmux description
Board display integration	Rail, reset, and backlight sequencing, before the required operation
`include/configs/mx6ull_pa_mini.h`	Optional splash command, without replacing safety policy
Identified boot partition	Trusted, validated `splash.bmp`

This extends the same board port. It does not create a second board or change its ROM/DCD boot path. These are modern-port filenames, not vendor target names. Establish UART1 and SD/eMMC first; then add video with a matched panel, verified DDR, display clocks, memory reservations, and power.

The guide lists ATK RGB module IDs such as 4342 (480 x 272), 4384/7084 (800 x 480), 7016 (1024 x 600), and 1018 (1280 x 800). Choose by the fitted module, not by screen diagonal alone. The public vendor eMMC board source currently defines a `TFT7016` mode and `detect = NULL`: neither a default `panel` string nor that table proves automatic MINI panel-ID detection. Its helper also treats GPIO5_IO09 as LCD reset, whereas the supplied core labels SNVS_TAMPER9 `nWDOG` in the shared reset network. Review that difference before enabling or migrating the helper; do not toggle it as a harmless independent panel reset. Keep the exact matched mode and observed baseline behavior as comparison evidence, rather than advertising every kit screen as supported by the same binary.

24I.3 Enable the display-related configs

In v2026.04 use:

```
CONFIG_VIDEO=y
CONFIG_VIDEO_MXS=y
CONFIG_VIDEO_BPP32=y
CONFIG_CMD_BMP=y
CONFIG_BMP=y
CONFIG_BMP_24BPP=y
CONFIG_CMD_FAT=y
CONFIG_FS_FAT=y
CONFIG_CMD_ITEST=y
```

`VIDEO` is the driver-model video option in this release; there is no separate `DM_VIDEO` switch to add. `VIDEO_MXS` selects `drivers/video/mxsfb.c` for this LCDIF family. The inherited i.MX6UL DT compatible includes `fsl,imx6sx-lcdif`, which this driver matches. Its 24/18-bit output paths use **32-bit framebuffer storage**. `BMP_24BPP` enables decoding a 24-bit source BMP; the source-file format and framebuffer format are not the same setting.

`SPLASH_SCREEN`/`SPLASH_SCREEN_ALIGN` are optional framework choices, not requirements for the manual `bmp display` path used here. Enabling them alone does not load a file. Automatic framework loading requires its own preparation hook/source configuration. `CMD_CLS` is optional for console experiments.

Build and inspect a separate output directory:

```
$ . ~/imx6ull/scripts/env.sh
$ cd ~/imx6ull/src/u-boot
$ make O="$IMX6ULL_HOME/build/uboot-video" ARCH=arm CROSS_COMPILE="$CROSS_COMPILE"
mx6ull_pa_mini_defconfig
$ make O="$IMX6ULL_HOME/build/uboot-video" ARCH=arm CROSS_COMPILE="$CROSS_COMPILE" -j "$
(nproc)"
```

```
$ grep -E 'CONFIG_(VIDEO|VIDEO_MXS|VIDEO_BPP32|BMP|BMP_24BPP|CMD_BMP|CMD_FAT|CMD_ITEST)='
"$IMX6ULL_HOME/build/uboot-video/.config"
```

The grep is an inspection after configuration, not a substitute for reading Kconfig dependencies or a successful build. The unmodified EVK defconfig has no enabled video path; it is not a ready-made MINI LCD configuration.

The ULL EVK header also does not supply the base alias required by this driver. In the MINI header, retain the architecture register definitions and add the same alias used by the upstream UL/Colibri-ULL ports:

```
#include <asm/arch/imx-regs.h>
#define MXS_LCDIF_BASE MX6UL_LCDIF1_BASE_ADDR
```

Without it, enabling `VIDEO_MXS` on the ULL EVK-derived port fails to compile at `MXS_LCDIF_BASE`. This is a source integration requirement, not permission to write controller registers or a hardware validation result.

24I.4 Add the LCD to the U-Boot Device Tree

The MXS driver reads an LCDIF `display` phandle, then `bits-per-pixel` and the **first** `display-timings` entry in that referenced node. In this version it does not use `bus-width` to choose the wire format or `native-mode` to select another timing entry. Match its actual parser, not just a Linux binding.

The following is an **800 x 480 timing-format example**, not the vendor's TFT7016 mode or a timing selected from MINI panel-ID lines. For a fitted 4384/7084 module, compare every clock/porch/polarity with its own timing specification; sharing a resolution does not qualify these values. For 7016 or another module, replace the geometry too. It assumes 24-bit RGB wiring and active-high DE:

```
&lcdif {
    pinctrl-names = "default";
    pinctrl-0 = <&pinctrl_lcdif>;
    display = <&display0>;
    status = "okay";

    display0: display {
        bits-per-pixel = <24>;
        display-timings {
            timing0 {
                clock-frequency = <33000000>;
                hactive = <800>;
                vactive = <480>;
                hfront-porch = <40>;
                hback-porch = <88>;
                hsync-len = <48>;
                vfront-porch = <13>;
                vback-porch = <32>;
                vsync-len = <3>;
                hsync-active = <0>;
```

```
                vsync-active = <0>;
                de-active = <1>;
                pixelclk-active = <0>;
            };
        };
    };
};
```

Supply the board-specific `pinctrl_lcdif` group under the existing IOMUXC node. For 24-bit wiring that normally includes LCD_CLK, ENABLE/DE, HSYNC, VSYNC, and DATA00..23 with the appropriate pad macros. An 18-bit panel needs its documented signal mapping and driver format; do not arbitrarily remove six wires. On MINI, retain the SGM3157/ boot-strap arrangement described above and account for the shared reset and BLT_PWM pin ownership. Pad drive strength and slew are electrical choices, not universal `0x79` settings.

The example's horizontal total is `800+40+88+48=976` clocks and vertical total `480+13+32+3=528` lines. Divide the pixel clock by their product to obtain the nominal frame rate. This is arithmetic, not a measured/calibrated clock or a panel-approved refresh rate. Verify actual clock capability, edge semantics, and DE polarity; this MXS implementation sets ENABLE polarity high during setup, so a DE-low panel needs source-level investigation rather than a DT promise alone.

The driver does **not** resolve a generic panel/backlight phandle and perform all module power/reset sequencing for you. GPIO1_IO08 can provide GPIO enable or PWM1 output according to the matched module's BLT_PWM contract; the net name alone does not establish polarity, duty limits or sequencing. Provide that reviewed integration before probe where needed: stabilize rails, apply reset timing, prepare valid scanout, then enable the backlight in the panel-approved order. A `gpio-backlight` node on its own is not proof that this path invokes it. Critical power checks from Chapter 24H must precede the operation they guard, not merely run later in `board_late_init()`.

24I.5 Prepare the BMP file

Use one trusted, uncompressed Windows-style BMP with a 40-byte DIB header, 24-bit BGR pixels, positive height (bottom-up rows), no palette, and dimensions that fit the verified panel. Start with clear color bars and a border; they make swapped colors and shifted edges visible without a complex logo.

For the **800 x 480 format example**, each row is padded to a multiple of four bytes:

```
row bytes = ((800 * 3 + 3) / 4 rounded down) * 4 = 2400
pixel bytes = 2400 * 480 = 1152000
file bytes = 14 + 40 + 1152000 = 1152054
framebuffer bytes for the driver's 32-bit path = 800 * 480 * 4 = 1536000
```

These are calculated sizes for that format, not observed output. Inspect the actual file's header, offset, dimensions, compression, and total length on the host. U-Boot's `bmp info` reads header fields; it does not fully validate a file against a supplied buffer length. Do not feed arbitrary recovery-uploaded BMPs to a bootloader decoder as if it were a hardened image service.

Put the checked file on the **identified** intended boot partition using the already approved storage-copy workflow. Do not use a guessed `/dev/sdX1` mount/copy recipe. If using Chapter 24G UMS, the host must own the exposed range exclusively and release it before U-Boot loads the file.

24I.6 Manual U-Boot display test

First establish the image-buffer reservation using `bdinfo`, the port's memory layout, and the video framebuffer reservation. The image buffer must fit entirely inside qualified DDR and avoid U-Boot relocation/stack/malloc, control DT, framebuffer, kernel/DTB/initrd ranges, and decompression space. `0x88000000` is not safe merely because it is in DDR: video reserves memory near the top and the allocation depends on the build and RAM size.

These placeholders must be replaced by the reviewed layout:

```
pa-mini=> bdinfo
pa-mini=> dm tree
pa-mini=> setenv splashimage <reserved image-buffer start>
pa-mini=> setenv splash_max <reserved buffer size in hexadecimal bytes>
pa-mini=> setenv splashpart <identified MMC device:boot-partition>
pa-mini=> fatsize mmc ${splashpart} splash.bmp
pa-mini=> printenv filesize
```

Confirm a nonzero file size no larger than `splash_max`, then load and inspect it. No host/other target writer may alter the file between size-check and load.

```
pa-mini=> fatload mmc ${splashpart} ${splashimage} splash.bmp
pa-mini=> bmp info ${splashimage}
pa-mini=> bmp display ${splashimage} 0 0
```

Require actual successful commands and sensible dimensions/bit depth. A successful display command means the software path accepted the image; the panel observation is another checkpoint. Record both rather than copying an invented success log. If U-Boot console output also targets video, later text or scrolling may overwrite the splash. Retain serial as the debug console; change display console routing only after diagnosing it explicitly.

24I.7 Add the splash command to the default environment

Only after the manual path works, merge this macro with the existing defaults. It deliberately leaves buffer address/size and partition to the reviewed board layout:

```
#define PA_MINI_SPLASH_ENV \
    "splashfile=splash.bmp\0" \
    "show_splash=if test -n \"${splashimage}\" && " \
        "test -n \"${splash_max}\" && test -n \"${splashpart}\"; then " \
        "if fatsize mmc ${splashpart} ${splashfile}; then " \
            "if itest ${filesize} -gt 0 && itest ${filesize} -le ${splash_max}; then " \
                "if fatload mmc ${splashpart} ${splashimage} ${splashfile}; then " \
                    "bmp display ${splashimage} 0 0; " \
                "else echo Splash load failed; false; fi; " \
            "else echo Splash exceeds reserved buffer or is empty; false; fi; " \
        "else echo Splash file unavailable; false; fi; " \
    "else echo Splash buffer not configured; false; fi\0"
```

`fatsize` produces hexadecimal `filesize`; `itest` handles the numeric comparison. The command bounds the file load, not malformed BMP header reads. Use only the prevalidated trusted asset. Do not replace this with a truncated load and then try to decode an incomplete file.

For Chapter 24F’s policy, a reviewed normal-path wrapper is:

```
normal_with_splash=run show_splash; run boot_selected
bootcmd=run normal_with_splash
```

Make that the single normal command definition, with matching compiled boot command settings. Keep `altbootcmd` and Chapter 24G/H overrides intact. Do not introduce a new `normal_boot=run mmcboot` that bypasses A/B selection or safety. Updating defaults does not replace saved variables; reconcile them manually without deleting unrelated settings. No `saveenv` is needed for initial RAM-only trials.

24I.8 If the display is black

Observation/check	Next evidence
Backlight dark	Approved rail/enable measurement; a flashlight may reveal faint pixels
File not loaded	Storage identity, `fatsize`, load status, actual asset path

Observation/check	Next evidence
Header implausible	Reinspect trusted host file, do not keep decoding it
No video device	Generated config, compatible match, control DT and probe errors
Lit but blank/shifted panel	Reset sequence, pinmux, signal routing, clock/ porches/polarities
Wrong colors	Wire RGB order, framebuffer format, BMP format/ conversion
Image changes after display	Video-console output or Linux reinitialization
Corruption elsewhere	Image/framebuffer/load/decompression memory overlap

Work outward from a known observation. A lit backlight does not prove pixel traffic, and a framebuffer in RAM does not prove electrical timing. Use only approved probing points and instruments; never change supply voltages to try to make the image brighter.

24I.9 Make the splash optional

`run show_splash; run boot_selected` intentionally ignores the first command's failure and still attempts the recorded normal boot policy. The second command's success/failure remains the wrapper's result. A missing logo does not choose another slot or mark a candidate good.

For a factory image test, use a different wrapper:

```
factory_splash=if run show_splash; then echo TEST:SPLASH:COMMAND_OK; else echo
TEST:SPLASH:FAIL; false; fi
```

`COMMAND_OK` is not a panel inspection PASS. The fixture still needs optical or electrical evidence if display output is part of acceptance. Avoid an unconditional trailing success message that hides a failed BMP command.

24I.10 Production choices

Asset location	Tradeoff
Boot-partition BMP	Easy replacement, but file validity/access must be controlled
Embedded asset	Fixed with the U-Boot build; requires a defined decode/display integration
Verified FIT/partition	Authentication can cover the asset, only if it is actually checked before decoding

A file residing inside a signed container is not automatically verified by `fatload` followed by `bmp display`. Keep image-size validation even when integrity is authenticated. Optional visual feedback must not weaken the recovery/power decision or delay watchdog service unpredictably.

24I.11 Key Notes

- BMP source data, framebuffer pixels and RGB wires are separate formats. The MXS 24/18-bit output paths use 32-bit framebuffer storage, independent of a 24-bit BMP decoder.
- eLCDIF scanout and backlight power are separate paths. The fitted module’s timing, sequencing and BLT_PWM contract determine whether valid RAM pixels become a visible image.
- MINI J1 includes shared reset and SGM3157 isolation on display/boot-strap lines. An inherited independent LCD-reset helper is not permission to drive the shared reset network.
- U-Boot’s actual parser determines which timing and power properties take effect. A Linux binding or backlight node does not guarantee that the selected bootloader driver consumes it.
- Trusted BMP validation and separate image/framebuffer reservations precede decoding. `bmp info` header output is not a complete bounded-input validation service.
- The optional splash wrapper can report failure while preserving normal A/B boot, but must not bypass recovery or power vetoes. A displayed logo neither accepts a candidate nor establishes product readiness.

- Linux owns its own display initialization after handoff. Seamless image preservation requires an explicit cross-stage contract, not merely a successful `bmp display` call.

24I.12 Pitfalls

- **Backlight equals initialized panel.** They are separate paths.
- **Linux binding equals U-Boot behavior.** Read this version's actual parser.
- **Missing `BMP_24BPP`.** Generic BMP support alone is not every source format.
- **Universal timing/pad settings.** MINI routing and the actual module determine them.
- **Ignoring the SGM3157 switches or shared reset.** They are part of this board, not optional decoration.
- **Loading over the framebuffer or kernel.** Reserve the complete range first.
- **Decoding an untrusted/truncated image.** Header output is not full validation.
- **A splash wrapper replacing safety policy.** Preserve A/B and both veto paths.
- **Expecting seamless Linux takeover.** Reinitialization is a separate contract.

24I.13 Going deeper

- Supplied MINI V2.2 schematic, sheets 1-2, core schematic sheets 5/8, and guide MINI Section 5.4.4 (pp. 267-268): J1 wiring, panel choices, SGM3157 isolation, shared reset, and GPIO1_IO08 backlight control.
- Vendor eMMC board display implementation: inspect the configured mode and reset assumptions before migration.
- v2026.04 MXS video driver and video Kconfig.
- BMP command and video BMP conversion.
- Splash preparation and splash code, for the optional framework rather than this manual path.
- The fitted panel's official timing/electrical specification and the carrier schematic are prerequisites for a hardware example.

Previous: Chapter 24H: PMIC and power policy in U-Boot

Next: Chapter 24J: SPI LCD and I2C OLED displays in U-Boot

Chapter 24J: SPI LCD and I2C OLED displays in U-Boot

An SPI transfer succeeds, yet the screen does not change. That is possible: SPI reports whether the controller moved bits, not whether those bits were the panel's commands, used the right D/C level, or reached initialized pixel memory. An I2C ACK is similarly narrower than a visible OLED image.

This chapter follows those bytes to a small status display. We use upstream **U-Boot v2026.04**, distinguish its drivers from Linux's, and provide a bounded SSD1306 command for one documented module profile. The supplied **MINI V2.2 has no fitted SPI LCD or I2C OLED**. Its RGBLCD socket belongs to Chapter 24I; a small serial display is an added module. We use the actual P4 expansion routes below, with a shared P1/I2C2 alternative for an OLED. The guide's ALPHA I2C/SPI sensor labs are not evidence that their devices exist on MINI.

Begin with Chapter 19's MINI vendor baseline and UART1 console. The bus mapping below comes from the supplied MINI/core schematics; the code targets the separate **v2026.04 `mx6ull_pa_mini` migration**, not a drop-in vendor patch. Keep UART1 for diagnostics while bringing up the added module.

24J.1 First decision: what kind of display is it?

Display	Bus/memory	Path in this chapter
Parallel RGB panel	eLCDIF scans a RAM framebuffer	Chapter 24I's DM video path
ST7735/ST7789/ILI9341-type color module	SPI commands and controller pixel RAM	Board-specific transport/init unless a verified driver implements the required path
SSD1306 monochrome module	I2C commands and page-organized display RAM	Bounded board-local status command
SH1106 module	Similar-looking but different addressing/geometry	Its own driver/profile, not the SSD1306 command below
Smart UART display	Module-specific serial protocol	Separate product integration

The v2026.04 `drivers/video/` tree has no matching ST7735, ST7789, ILI9341, SSD1306, or SH1106 driver for these examples. A Linux-compatible string does not create a U-Boot driver. In particular, a panel-initialization driver is not necessarily a DM video driver that allocates a framebuffer and flushes pixels over SPI.

Use a full framebuffer only when you need it and have the complete transport integration. A rectangle indicating service mode needs far less code, but still needs correct initialization and errors. Calling the routine “tiny” does not exempt it from those requirements.

For a first MINI extension, a documented **128 x 64 I2C SSD1306 module with 3.3 V-compatible supply/logic and the specified charge-pump profile** needs fewer control wires than a color SPI LCD. Use I2C1 on P4 as described in Section 24J.8. A color module additionally needs a reviewed D/C, reset, backlight and initialization path; the transport helper alone will not display an image. Neither module is part of the stock-board acceptance test.

24J.2 What belongs in U-Boot?

Useful early states include boot attempt, recovery service, refusal reason, and a defined factory result. Show update progress only after verified work actually advances; transferring bytes into a buffer is not a committed flash update. Do not display PASS before the check succeeds.

Keep the serial console as the authoritative diagnostic path. A display may itself be the failed optional peripheral. Rendering it must not enable an unsafe rail, bypass a power veto, or block watchdog service indefinitely. Linux takes over the display after its own driver initializes it; U-Boot’s status routine is not the product UI.

24J.3 SPI LCD hardware checklist

Signal/condition	Evidence needed
SCLK/MOSI/CS	Actual ECSPI controller, pad routing, chip select, SPI mode and allowed frequency
MISO	Whether connected and whether reads are supported; write-only panels cannot prove identity by readback
D/C	Command/data polarity and setup time relative to CS/clock

Signal/condition	Evidence needed
RESET	Assertion polarity, pulse width, delay after release
Backlight	Separate driver/enable/PWM polarity and safe startup order
VCC/IOVCC	Correct voltage levels, power sequence, no powered GPIO back-feed
Geometry	Controller RAM size, visible window offsets, rotation and RGB/BGR order

MINI sheet 3's **P4** exposes a useful ECSPI3 route, verified against the v2026.04 i.MX6UL pin-function definitions:

P4 pin	Schematic net / SoC pad	ECSPI3 function
17	UART2_TXD / UART2_TX_DATA	SS0 (native chip select)
18	UART2_RXD / UART2_RX_DATA	SCLK
19	UART2_RTS / UART2_RTS_B	MISO
20	UART2_CTS / UART2_CTS_B	MOSI

The UART labels name physical routes, not the selected function. Disable UART2 ownership and configure the corresponding `__ECSPI3_*` pinmux plus the ECSPI3 controller in U-Boot's control DT. Do not use ALPHA's sensor connector wiring or switch ENET2 pads away from the MINI's onboard PHY. Confirm P4 pin 1/orientation on the actual assembly before attaching wires.

P4 pin 47 is ground, 48 DCDC_3V3, and 46 DCDC_5V. These are board supply nets, not interchangeable module voltages; check the module's supply and IO levels, pull-ups, current budget and power-off behavior. D/C/reset and backlight still need separately assigned GPIOs/module circuits. UART1_CTS/GPIO1_IO18 is MINI KEY0; GPIO1_IO01 carries GBC_KEY/AP_INT, not that switch. GPIO1_IO03 is LED0, GPIO1_IO08 RGB backlight, and SNVS_TAMPER1/GPIO5 bit 1 BEEP through an active-low PNP switch.

GPIO1_IO04 also reaches camera PWDN, and GPIO1_IO02 camera RESET. A header pin is not automatically unowned: review accessories and the control DT before selecting the additional outputs.

A typical controller sequence includes reset, sleep exit and its delay, power/gamma configuration, pixel format, memory orientation, a visible column/row window, pixel data, display-on, and backlight. The exact command bytes and delays come from the **controller and module** specification. ST7789, ST7735, and ILI9341 sequences are not interchangeable.

24J.4 SPI LCD: Device Tree shape

Describe the real ECSPI bus, pinmux, and chip select. A display child needs its actual `reg` chip-select index and `spi-max-frequency`; mode properties, D/C/reset GPIOs, and any power/backlight links must match the driver you are actually writing. Do not add `sitronix,st7789v` and expect v2026.04 to bind a missing driver.

Property/group	What it establishes
Bus pinctrl	SCLK/MOSI and any MISO route; separate D/C/reset/CS GPIO pad groups where needed
`cs-gpios` or native CS	The verified selected device, not an assumed GPIO4 bit 26
Child `reg`	SPI chip select, not a memory address
`spi-max-frequency` and mode	A reviewed bus limit, not a calibrated transfer rate
Driver-consumed GPIO properties	Ownership/polarity only if the implementation requests them
Width/height/rotation	Useful only if the chosen driver parses them

For an initial command-driven prototype, bind a reviewed generic SPI child or the new board-local driver and request GPIO descriptors explicitly. Generic SPI binding does not reset the LCD or render pixels. A future video driver must provide geometry/storage, draw/flush behavior, and error handling; a panel node alone is not that implementation.

24J.5 SPI LCD: useful configs

The i.MX bus foundation is:

```
CONFIG_SPI=y
CONFIG_DM_SPI=y
CONFIG_MXC_SPI=y
CONFIG_CMD_SPI=y
CONFIG_DM_GPIO=y
```

`CMD_SPI` adds **`sspi`**, not a command named `spi`. Check `help sspi` on the actual build. Its arguments include a bit count, not a byte count. Its small command buffer is not a frame uploader; do not paste unbounded pixel hex strings. A transfer also needs the right D/C state, which `sspi` does not configure for you.

If implementing a real DM video driver, use `CONFIG_VIDEO` and the required BMP formats as in Chapter 24I; **`CONFIG_DM_VIDEO` is not a v2026.04 symbol**. Do not enable generic video and claim it supplies the absent panel driver. Build the same MINI port with Chapter 3's sourced environment and a separate `O=` directory. Bus compilation does not validate the added module wiring.

24J.6 SPI LCD manual bring-up order

First keep the backlight disabled and review the electrical/reset sequence. Verify the bus/pad route, then use an approved analyzer connection to inspect a small, documented command transaction. A write-only transfer returning zero cannot distinguish an absent panel from a correctly connected one.

This **transport helper**, not a complete panel driver, sends one command and up to 512 data bytes with CS held across the D/C change. It assumes a probed SPI **child** device and an already requested D/C descriptor whose logical 0 is command and 1 is data:

```
#include <dm.h>
#include <spi.h>
#include <asm/gpio.h>
#include <linux/errno.h>
#include <linux/types.h>

static int lcd_command(struct udevice *child, struct gpio_desc *dc,
                       u8 command, const u8 *data, unsigned int len)
{
    int ret;

    if (len > 512 || (len && !data))
        return -EINVAL;
    ret = dm_spi_claim_bus(child);
    if (ret)
        return ret;
    ret = dm_gpio_set_value(dc, 0);
    if (ret)
        goto release;
```

```
    ret = dm_spi_xfer(child, 8, &command, NULL,
                      SPI_XFER_BEGIN | (len ? 0 : SPI_XFER_END));
    if (ret)
        goto end;
    if (len) {
        ret = dm_gpio_set_value(dc, 1);
        if (ret)
            goto end;
        ret = dm_spi_xfer(child, len * 8, data, NULL, SPI_XFER_END);
        if (ret)
            goto end;
    }
    goto release;
end:
    /* Best effort to release CS; preserve the original error. */
    dm_spi_xfer(child, 0, NULL, NULL, SPI_XFER_END);
release:
    dm_spi_release_bus(child);
    return ret;
}
```

The byte bound makes `len * 8` safe and keeps the buffer contract explicit. The supplied buffer must really contain `len` bytes. A cleanup error cannot prove CS returned idle: report the original error and stop that display path, then use the board's approved reset/recovery procedure. The helper does not supply reset, delays, bus speed/mode, or rail initialization. Add required D/C setup/hold delays if the selected controller's specification needs them.

After a documented initialization, set a small column/row window using the controller's required high-byte/low-byte order, then write a known pixel pattern. For an RGB565 mode, red is the byte pair `f8 00`, green `07 e0`, and blue `00 1f` **if** the selected mode is high-byte-first RGB. Sending a native little-endian `u16` array reverses those bytes. RGB/BGR settings can change the visible colors too.

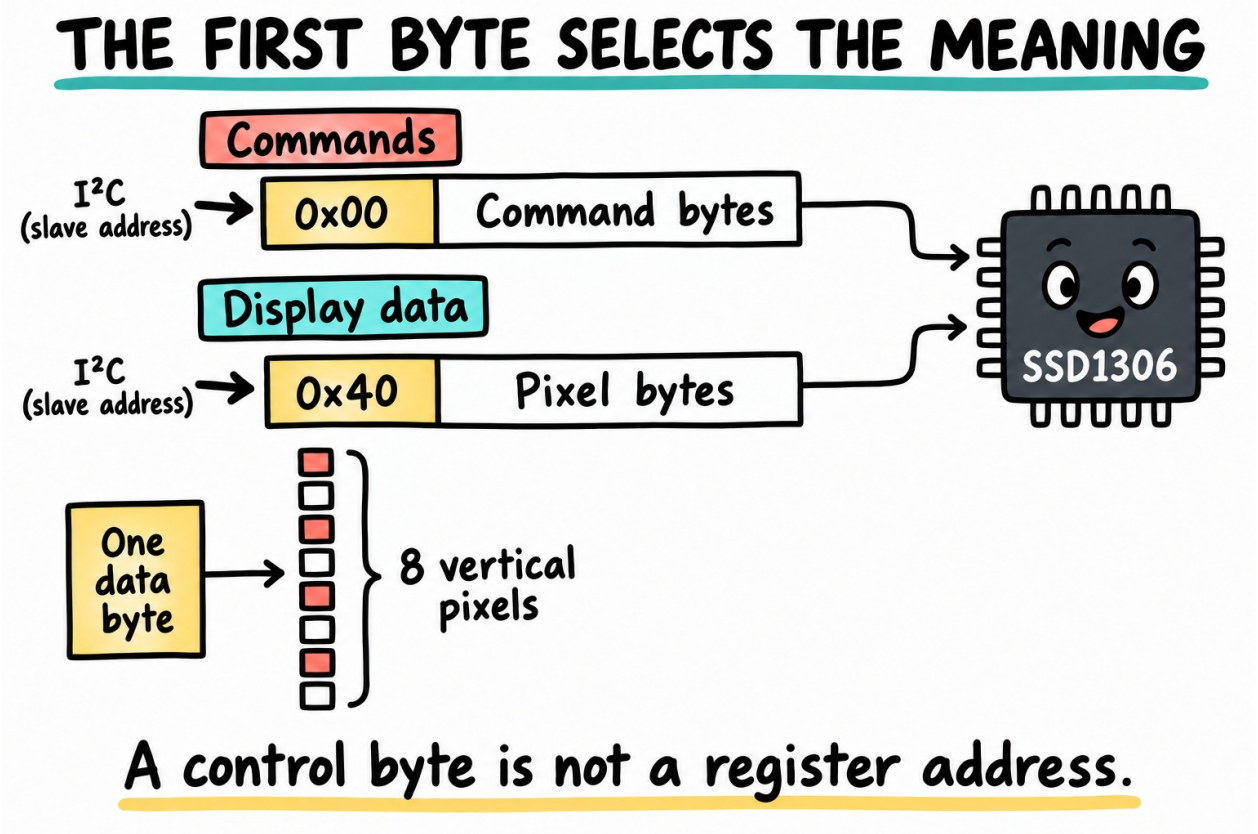
The bounded helper is not a whole-screen fill engine. A 16 x 16 RGB565 window needs 512 pixel bytes; a larger frame needs a reviewed streaming/chunking routine that preserves controller address state and watchdog scheduling. Do not restart RAM-write at the origin for every chunk and call that a fill. Prove color/window mapping before adding text. Without module-specific init, this SPI path remains an implementation example, not a ready-to-use driver.

24J.7 I2C OLED: why it feels different

An SSD1306 has display RAM too. In its page organization, one byte controls eight vertical pixels at one column. A 128 x 64 image needs `128 * (64 / 8) = 1024` data bytes. There is no separate backlight: OLED pixels emit light.

In I2C mode, a control byte precedes the payload. `0x00` selects a command stream; `0x40` selects display data. These are **not register addresses**. The first payload byte must not be preceded by an extra register-offset byte inserted by a generic I2C helper.

SSD1306 modules may use 7-bit address `0x3c` or `0x3d`, but verify the fitted strap/module. Some modules are SPI, 128 x 32, externally supplied, or contain SH1106 instead. A matching outline or an ACK does not establish this profile.



The control byte selects how the SSD1306 interprets the following bytes. It is not a register address, and no extra offset byte belongs before it. The pixel column illustrates page organization without claiming a fitted MINI display.

24J.8 I2C OLED Device Tree shape

Linux's `solomon,ssd1306fb-i2c` binding does not imply a matching U-Boot driver. Keep the two control trees and ownership stages distinct. The board command below directly obtains a generic I2C chip on a **verified** bus/address.

On MINI, use **I2C1 via P4 pin 43 UART4_TXD (SCL), pin 42 UART4_RXD (SDA)**, and the reviewed supply/ground connections. Pin 41 is ENET1_RXER, not SCL; pin 44 is UART5_RXD/I2C2 SDA, not I2C1 SDA. Do not infer UART or bus pairs from adjacent header rows. MINI sheet 1 shows R42/R57 4.7 kOhm pull-ups to DCDC_3V3. Core sheet 6 and the upstream pin definitions map these pads to `MX6UL_PAD_UART4_TX_DATA__I2C1_SCL` and `MX6UL_PAD_UART4_RX_DATA__I2C1_SDA`. Disable UART4 on these pads, enable `&i2c1`, and supply its IOMUXC pinctrl group with the board's reviewed pad settings. Account for the module's own pull-ups rather than adding another pair automatically. Keep UART1/USB_TTL as the console.

Use `i2c bus` to map this **hardware I2C1** to its U-Boot sequence, then set the command's `OLED_BUS` accordingly. The `0` below is an example sequence, not an assertion that every migrated DT assigns I2C1 sequence zero. I2C2 is also routed to the RGB touch/camera connectors, so it is not an isolated spare bus on an accessorized MINI. Prefer the explicitly reviewed P4 route for this first module instead of probing an unknown shared bus.

The layout also labels **P1 OLED/CAMERA**. Sheet 2 routes P1 pin 1 to DCDC_3V3, pin 2 to ground, pin 4 to I2C2 SCL and pin 6 to I2C2 SDA. A documented OLED adapter can use those wires after checking the shared camera/touch bus and module address. This is an alternative to P4/I2C1, not a promise that a generic four-pin OLED plugs into P1 directly; the connector's other pins carry camera signals. Select the corresponding I2C2 pinmux and actual U-Boot bus sequence if taking this route.

For a dedicated U-Boot generic child, explicitly describing `u-boot,i2c-offset-len = <0>` prevents register-offset insertion. Its chosen compatible/driver must be reviewed; do not attach this property to an unrelated chip or override a PMIC driver's protocol. The command also sets and checks zero offset length because the chip may already be bound. Linux should receive its own supported display node and reset/power wiring.

24J.9 I2C OLED configs

Enable the actual bus and command support:

```
CONFIG_DM_I2C=y
CONFIG_SYS_I2C_MXC=y
CONFIG_CMD_I2C=y
CONFIG_DM_GPIO=y
```

`BOARD_LATE_INIT` is needed only if the integration uses that callback, not for the command itself. First inspect `i2c bus`, bus aliases/pinmux, supply, pull-ups, and reset wiring. Do not indiscriminately scan a shared PMIC bus. A reviewed known-address check may establish an ACK, not chip identity or successful display initialization.

24J.10 A tiny SSD1306 U-Boot command

This example covers **only a verified 128 x 64 SSD1306 I2C module with the internal charge-pump profile** described below. Before running it, the module must be powered and reset with its documented timing. Its segment/COM orientation, electrical settings, contrast, and power sequence must match its specification. Do not substitute a 128 x 32 or SH1106 module. The P4/I2C1 route above is documented MINI hardware; the particular module, address strap, reset and supply profile still need matching.

Create `board/myorg/mx6ull_pa_mini/oled_ssd1306.c` in your port. Change the bus/address constants only after mapping the actual controller and strap:

```
// SPDX-License-Identifier: GPL-2.0+
#include <command.h>
#include <dm.h>
#include <i2c.h>
#include <stdio.h>
#include <linux/errno.h>
#include <linux/kernel.h>
#include <linux/string.h>

#define OLED_BUS       0       /* Must match the verified U-Boot bus sequence. */
#define OLED_ADDR      0x3c    /* Must match the module's 7-bit address. */
#define OLED_WIDTH     128
#define OLED_PAGES     8
#define OLED_CHUNK     16

static int oled_packet(struct udevice *dev, u8 control,
                       const u8 *data, unsigned int len)
{
    u8 buf[OLED_CHUNK + 1];

    if (!len || len > OLED_CHUNK || !data)
        return -EINVAL;
    buf[0] = control;
    memcpy(buf + 1, data, len);
    return dm_i2c_write(dev, 0, buf, len + 1);
}

static int oled_commands(struct udevice *dev, const u8 *data, unsigned int len)
{
    return oled_packet(dev, 0x00, data, len);
}

static int oled_init(struct udevice *dev)
{
    static const struct {
        u8 bytes[2];
        u8 len;
    } init[] = {
        { { 0xae, 0 }, 1 },       /* Display off until RAM is cleared. */
        { { 0xd5, 0x80 }, 2 },    /* Module profile: clock. */
        { { 0xa8, 0x3f }, 2 },    /* 64-row multiplex. */
        { { 0xd3, 0x00 }, 2 },    /* Display offset. */
        { { 0x40, 0 }, 1 },       /* Start line. */
        { { 0x8d, 0x14 }, 2 },    /* Internal charge pump profile. */
        { { 0x20, 0x02 }, 2 },    /* Page addressing, matching drawing below. */
        { { 0xa1, 0 }, 1 },       /* Segment remap. */
        { { 0xc8, 0 }, 1 },       /* COM scan direction. */
        { { 0xda, 0x12 }, 2 },    /* 128x64 COM pin profile. */
        { { 0x81, 0x7f }, 2 },    /* Profile contrast, not a measured optimum. */
        { { 0xd9, 0xf1 }, 2 },    /* Charge-pump precharge profile. */
        { { 0xdb, 0x20 }, 2 },    /* Documented VCOMH encoding. */
        { { 0xa4, 0 }, 1 },       /* Display RAM, not all-pixels-on. */
        { { 0xa6, 0 }, 1 },       /* Normal polarity. */
        { { 0x2e, 0 }, 1 },       /* Scrolling disabled. */
    };
    unsigned int i;
    int ret;
```

```
    for (i = 0; i < ARRAY_SIZE(init); ++i) {
        ret = oled_commands(dev, init[i].bytes, init[i].len);
        if (ret)
            return ret;
    }
    return 0;
}

static int oled_draw(struct udevice *dev, int bar)
{
    u8 pixels[OLED_CHUNK];
    unsigned int page, col;
    int ret;

    for (page = 0; page < OLED_PAGES; ++page) {
        u8 position[] = { 0xb0 + page, 0x00, 0x10 };

        ret = oled_commands(dev, position, sizeof(position));
        if (ret)
            return ret;
        memset(pixels, bar && page < 2 ? 0xff : 0, sizeof(pixels));
        for (col = 0; col < OLED_WIDTH; col += OLED_CHUNK) {
            ret = oled_packet(dev, 0x40, pixels, sizeof(pixels));
            if (ret)
                return ret;
        }
    }
    return 0;
}

static int do_oled(struct cmd_tbl *cmdtp, int flag, int argc, char *const argv[])
{
    struct udevice *dev;
    const u8 on = 0xaf;
    int bar, ret;

    if (argc != 2)
        return CMD_RET_USAGE;
    if (!strcmp(argv[1], "clear"))
        bar = 0;
    else if (!strcmp(argv[1], "bar"))
        bar = 1;
    else
        return CMD_RET_USAGE;

    ret = i2c_get_chip_for_busnum(OLED_BUS, OLED_ADDR, 0, &dev);
    if (ret)
        goto fail;
    ret = i2c_set_chip_offset_len(dev, 0);
    if (ret)
        goto fail;
    ret = oled_init(dev);
    if (ret)
        goto fail;
    ret = oled_draw(dev, bar);
    if (ret)
        goto fail;
    ret = oled_commands(dev, &on, 1);
    if (ret)
```

```
        goto fail;
    return CMD_RET_SUCCESS;
fail:
    printf("OLED transfer failed on bus %d address 0x%02x: %d\n",
           OLED_BUS, OLED_ADDR, ret);
    return CMD_RET_FAILURE;
}

U_BOOT_CMD(oled, 2, 0, do_oled,
           "draw a checked SSD1306 module status pattern",
           "clear - initialize, clear RAM, then enable display\n"
           "oled bar - draw a 16-pixel-high band (not update progress)");
```

Each command and its argument stay in one bounded packet. Drawing uses **page addressing** (`0x20,0x02`) followed by the matching page/column commands. Each page contains eight 16-byte data packets. The first two pages form a 16-pixel band; this is a recognizable marker, not a percentage gauge. The entire RAM is written before display-on, avoiding stale pixels at startup. If any transfer fails, stop and return failure; partially changed pixels must not be interpreted as a valid result.

The command encodings were checked against Solomon Systech's SSD1306 Rev 1.1 command tables and its charge-pump application note, available in this manufacturer document hosted by Focus LCDs. The document specifies page mode, control bytes, and the VCOMH `0x20` encoding; it does not qualify the supply/reset design of an unknown module.

Add the object to the existing board Makefile only for the selected optional feature, or for a dedicated example build:

```
obj-y += oled_ssd1306.o
```

Then build with the selected Linux Arm compiler as in Chapter 24F. On an approved matching module, `oled clear` and `oled bar` are the manual checks. `oled typo` or missing arguments must return usage **before any bus access**. There is no successful hardware log promised here. A host transfer fixture can verify bytes and failures; it cannot qualify OLED voltages or reset timing.

24J.11 Showing status automatically

Only after manual qualification, add a RAM-only optional wrapper to the existing normal A/B path. Use it only from an approved normal-boot console, not one stopped by a recovery or power veto:

```
pa-mini=> setenv show_oled 'oled bar'
pa-mini=> setenv normal_with_oled 'run show_oled; run boot_selected'
pa-mini=> setenv bootcmd 'run normal_with_oled'
pa-mini=> run normal_with_oled
```

If the RGB splash is also used, integrate both optional commands into one normal wrapper. Do not replace the port with an assumed `distro_bootcmd`, and do not blindly `saveenv` an experimental display hook. Retain the alternate rollback command and the recovery/power overrides. Optional display failure may be reported without blocking normal boot if the hardware is safe.

The last command tries Linux in this session. Resetting without saving reloads the earlier stored policy. A compiled board hook runs at a different time: review its interaction with the whole-environment bootcount save, as in 24G/H.

For a power fault, serial-only reporting is the default: initializing an optional display can draw exactly the current the policy is refusing. A rollback/recovery-specific symbol needs a defined renderer and qualified hook; `oled bar` alone does not distinguish those states. Do not call a static helper from another board source file without providing a proper interface and linking it deliberately.

24J.12 Which path should you choose?

Requirement	Implementation boundary
Full RGB logo	Chapter 24I’s verified video driver and trusted BMP path
SPI LCD with no upstream driver	Module-specific reset/init plus bounded transactions, then a real draw/flush implementation
SSD1306 marker on the exact profile	The board command above, after hardware qualification
Text, fonts, percentage progress	Additional renderer, clipping/bounds, real state source and transfer/error tests
Shared Linux display	Separate Linux driver/DT initialization and intentional handoff
Safety-critical indication	Defined indication semantics and independent failure behavior; not merely a colored shape

The first useful result is a small, explainable pattern with known bytes. Do not call it a working graphics stack or production status display before its missing layers have been built and tested.

24J.13 Pitfalls

- **Linux driver names used as U-Boot support evidence.** Check this release's tree.
- **SPI success treated as panel ACK.** SPI has no automatic device ACK.
- **Byte count passed as bit count.** `dm_spi_xfer` takes bits.
- **Native `u16` pixels on a high-byte-first wire.** Construct the actual bytes.
- **D/C or CS broken across a command/data pair.** Inspect the whole transaction.
- **Extra I2C offset byte.** SSD1306 requires its control byte first.
- **Horizontal mode mixed with page commands.** Initialization and drawing must agree.
- **SH1106 or 128x32 treated as this module.** Geometry and commands differ.
- **A display hook bypassing A/B or safety.** Optional feedback must retain policy.

24J.14 Key Notes

- The stock MINI has neither serial display, and v2026.04 lacks the named panel drivers used in these examples. A compatible string or working bus driver does not supply reset, initialization or rendering.
- SPI command/data transfers depend on D/C and CS across the entire transaction. The bounded helper owns bus claim/release and length checks, but not panel timing or a whole-screen streaming engine.
- `dm_spi_xfer` counts bits, while RGB565 wire order is a byte-level contract. Native little-endian pixel arrays do not automatically match high-byte-first panel modes.
- SSD1306 I2C packets start with command/data control bytes, not register offsets. Zero offset length, page-mode initialization and page/column writes must agree for all 1024 pixel bytes.
- The supplied OLED command covers only its documented 128 x 64 charge-pump profile. An ACK, similar module shape or SH1106 name does not establish that profile.
- P4 bus routes share pads with UART functions, and P1/I2C2 shares accessories. Module voltage, pull-ups and separate GPIO ownership remain hardware contracts beyond byte-transaction correctness.

- Optional indication retains serial diagnostics and boot/recovery/power precedence. Partial pixels after a transfer failure and a decorative bar are not verified status or committed update progress.

24J.15 Going deeper

- Supplied MINI V2.2 schematic, sheets 1-3, and core sheets 5-6/8: P4 bus routes, existing GPIO owners and I2C pull-ups. The guide's applicability table (pp. 7-8) excludes the stock MINI from its ALPHA I2C/SPI sensor labs.
- v2026.04 i.MX6UL pin-function definitions: UART2-to-ECSPI3 and UART4-to-I2C1 alternate functions.
- v2026.04 video drivers and SPI API.
- I2C API and I2C uclass: offset insertion and errors.
- Solomon Systech SSD1306 product documentation, plus the fitted module's documented initialization/power specification.
- The actual ST7735/ST7789/ILI9341 or SH1106 manufacturer's controller manual, matched to the module rather than a generic internet initialization list.

End of Part III.

We can now separate image choice from acceptance, service entry from ROM recovery, and a display indication from the operation it describes. The next part follows the kernel after U-Boot hands it the selected image and DTB.

Previous: Chapter 24I: U-Boot display and boot screen

Next: Chapter 25: Building mainline Linux for i.MX6ULL

Chapter 25: Building mainline Linux for i.MX6ULL

> **GIC:** ARM's Generic Interrupt Controller, the Cortex-A interrupt router roughly analogous to NVIC on Cortex-M. **MCU bridge:** Think of the GIC like the Cortex-M NVIC scaled up for Cortex-A: it routes peripheral interrupts to CPU cores and has separate distributor and CPU-interface blocks. **ELF:** Executable and Linkable Format, the standard Linux object and executable file format. **ABI:** Application Binary Interface: the calling convention, register use, binary format, and library contract that let separately built code run together. **GPIO:** General-Purpose Input/Output, a pin controlled as a digital input, output, or interrupt source. **MCU bridge:** Think of Linux GPIO like the same pin set/reset block you used on STM32, but accessed through a kernel subsystem that owns numbering, direction, interrupts, and user-space exposure. **TFTP:** Trivial File Transfer Protocol, a simple network protocol U-Boot commonly uses to fetch kernels from the host. **MCU bridge:** Think of U-Boot like a much larger boot stub plus debug monitor: it initializes hardware, loads the next image, and gives you commands before Linux starts.

> **What:** how a mainline Linux source tree, configuration and toolchain produce the kernel image, device tree blobs and modules for i.MX6ULL. Chapter 26 explains the bootloader handoff.
>
> **Why:** later chapters refer to these source and artifact layouts. Understanding their roles does not require completing a build.
>
> **Focus:** the four build artefacts you actually use (`vmlinux`, `zImage`, `*.dtb`, `*.ko`) and the four directories you will visit most (`arch/arm/`, `drivers/`, `include/`, `Documentation/`).

25.1 Why mainline

The Linux kernel ships under several release tracks:

- **Mainline** at `git.kernel.org/torvalds/linux.git`, Linus's tree. The current development tip. New releases roughly every 9 weeks (the "x.y" releases like 6.6, 6.7).
- **Stable**: Greg KH backports bug fixes to each mainline release for about 6 weeks. Tags look like `6.6.1`, `6.6.2`, and so on.
- **Long-Term Support (LTS)**: selected mainline releases get fix backports for years. Check `kernel.org/category/releases.html` before choosing for a product. As of June 2026, active longterm lines include `6.18`, `6.12`, `6.6`, `6.1`, `5.15`, and `5.10`.

- **Vendor BSPs**: NXP, ST, TI, and other silicon vendors ship forks pinned to a specific kernel minor with thousands of patches on top. NXP forks mainline into `linux-imx`. Active branches are pinned to `5.15` and `6.6`. Older branches exist for legacy products.

We build from upstream Linux, not from a vendor BSP. For source-reading stability, the chapter examples pin to the `v6.6` tag. For a shipping product, start from the current longterm branch that supports your hardware. The i.MX6ULL has had full support in mainline since v4.10 (released 2017). Chapter 30A goes deeper on when each track is appropriate.

25.2 Clone the source

```
$ cd ~/imx6ull/src
$ git clone --depth=1 --branch v6.6 \
      https://git.kernel.org/pub/scm/linux/kernel/git/stable/linux.git linux-v6.6
$ cd linux-v6.6
$ git describe --tags --always
v6.6
```

`v6.6` is an exact tag, so every reader sees the same source layout and similar line numbers. If you want the maintained longterm branch instead, clone `--branch linux-6.6.y` or, better, the newest longterm branch listed on kernel.org. That branch receives fixes, so line numbers drift over time.

The chapter examples assume `v6.6` unless otherwise noted. Newer longterm kernels work similarly for our purposes, but command output and source line numbers will differ.

Directory layout (the parts you'll touch)

```
linux/
├── arch/                # CPU architecture support
│   └── arm/             # 32-bit ARM, includes i.MX6ULL
│       ├── boot/        # boot wrapper + device tree compiler
│       │   ├── dts/     # device tree source files (*.dts, *.dtsi)
│       │   └── compressed/
│       ├── configs/     # *_defconfig files
│       ├── include/asm/ # ARM-specific kernel headers
│       ├── kernel/      # ARM-specific kernel entry (start.S, head.S)
│       ├── mach-imx/    # i.MX SoC family code
│       └── mm/          # ARM memory management
├── block/               # block device layer
├── crypto/              # cryptography subsystem
├── Documentation/       # Sphinx-rendered kernel docs
├── drivers/             # all device drivers, grouped by subsystem
│   ├── clk/imx/         # i.MX clock drivers
│   ├── gpio/            # GPIO drivers
│   ├── i2c/             # I²C bus + slave drivers
│   ├── input/           # input subsystem
│   ├── irqchip/         # interrupt controllers (incl. GIC)
│   ├── mmc/             # MMC / SD card
│   ├── net/             # network drivers
```

```
│   ├── pinctrl/         # pin control (incl. freescale/)
│   ├── rtc/             # real-time clocks
│   ├── spi/             # SPI bus
│   ├── tty/serial/      # UART drivers (incl. imx.c)
│   └── usb/             # USB host + gadget
├── fs/                  # file systems (ext4, fat, tmpfs, ...)
├── include/             # kernel-wide headers
│   ├── linux/           # the main public kernel API
│   ├── uapi/            # user-space ABI headers
│   └── dt-bindings/     # DT binding constants (clocks, gpios, IRQs)
├── init/                # kernel init: start_kernel, kernel_init
├── ipc/                 # System V IPC
├── kernel/              # core kernel: scheduler, signals, sysctl, locking
├── lib/                 # generic kernel utility code
├── mm/                  # memory management (page allocator, slab, ...)
├── net/                 # network stack
├── samples/             # example code
├── scripts/             # build scripts (Kconfig, kbuild, dtc)
├── security/            # LSMs (selinux, apparmor, ...)
├── sound/               # ALSA (SoC audio in sound/soc/)
├── tools/               # user-space companion tools
└── usr/                 # initramfs cpio packager
```

Every subsystem follows the same layout: top-level is the subsystem, next level is the vendor, and the lowest level is the SoC or board. The i.MX6ULL UART driver lives at `drivers/tty/serial/imx.c`. Its DT binding is at `Documentation/devicetree/bindings/serial/fsl-imx-uart.yaml`. Its register definitions are inside the driver file. This pattern repeats for every subsystem.

The four directories you will spend the most time in over the rest of this book:

- `arch/arm/boot/dts/`: every chapter from 27 onward
- `drivers/<subsystem>/`: every driver chapter in Part VI
- `Documentation/devicetree/bindings/`: DT binding schemas (Ch 27A)
- `include/dt-bindings/`: constants shared between DT source and driver source

25.3 Defconfig and the kernel’s config system

Like U-Boot, the kernel uses Kconfig + a `.config` file. The `arch/arm/configs/` directory holds default starting points:

> **U-Boot:** the bootloader that initializes enough hardware to load and start the Linux kernel.

```
$ ls arch/arm/configs/ | grep imx
imx_v6_v7_defconfig
mxs_defconfig
```

`imx_v6_v7_defconfig` (formerly `imx_v7_defconfig`) is the omnibus i.MX configuration that, on v6.6, covers i.MX31/35/27 (ARMv6), i.MX5 (selected boards), i.MX6 (all variants including ULL), and i.MX7. One config builds for all of them. A single `zImage` boots on any board that has a matching DT. This is the mainline style.

```
$ . ~/imx6ull/scripts/env.sh
$ make imx_v6_v7_defconfig
#
# configuration written to .config
#
```

All `make` commands in Part IV assume `env.sh` exported:

```
export ARCH=arm
export CROSS_COMPILE=arm-none-linux-gnueabihf-
```

If you did not source that script, write those two variables directly on each `make` command.

Inspect `.config`:

```
$ grep -E '^CONFIG_(ARCH|SOC|MACH|ARM|EABI|VFP)' .config | head -20
CONFIG_ARCH_MULTIPLATFORM=y
CONFIG_ARCH_MXC=y
CONFIG_SOC_IMX6=y
CONFIG_SOC_IMX6UL=y          # ← i.MX6UL family (includes 6ULL)
CONFIG_SOC_IMX6Q=y
...
```

Every entry in `.config` is either compiled in (`=y`) or compiled as a loadable module (`=m`). The set of options is enormous (~7000 for a v6.6 kernel). For now we trust `imx_v6_v7_defconfig`'s defaults. Chapter 30 returns to specific knobs.

25.4 Build

Single-shot build of everything we need:

```
$ make -j$(nproc) zImage modules dtbs
```

Three artifact groups (`make` builds them in dependency order from the same object tree):

- **`zImage`**: the compressed kernel image. ~6 MB. This is what U-Boot will `bootz`.
- **`modules`**: every `=m` driver, compiled to `.ko` files. ~hundreds in a default config. Installed separately.
- **`dtbs`**: every device tree blob the architecture defines. Includes `imx6ull-14x14-evk.dtb` for the NXP EVK and ~20 other i.MX6ULL variants.

On a modern host with `-j$(nproc)`, the first build takes 5–10 minutes. Incremental rebuilds finish in seconds.

What just got produced

```
$ ls -lh arch/arm/boot/zImage
-rw-r--r-- 1 you you 6.0M Jan 22 14:42 zImage

$ ls arch/arm/boot/dts/nxp/imx/imx6ull*.dtb
imx6ull-14x14-evk.dtb
imx6ull-9x9-evk.dtb
imx6ull-colibri-eval-v3.dtb
imx6ull-colibri-iris.dtb
imx6ull-colibri-wifi-iris.dtb
...

$ find . -name '*.ko' | wc -l
432

$ ls -lh vmlinux
-rwxr-xr-x 1 you you 145M Jan 22 14:42 vmlinux
```

Four artefacts, four roles:

File	Type	Used for
`vmlinux`	ELF with full debug info	gdb / `addr2line` / oops decoding
`arch/arm/boot/Image`	Raw kernel binary, uncompressed	rarely used on ARM32 (used on AArch64)
`arch/arm/boot/zImage`	Compressed kernel + decompressor stub	**what U-Boot loads**
`arch/arm/boot/dts/*.dtb`	Compiled device tree blobs	one per board variant
`**/*.ko`	Loadable kernel modules	each `=m` driver

You do *not* ship `vmlinux` to the target, it is 20× the size of `zImage`. You *do* keep `vmlinux` around on the host because it has symbols `zImage` lacks (Chapter 57 uses it to decode panics).

Module installation to the rootfs

> **Privilege boundary:** $ means normal user. # or sudo means root and can change host or target state. After a privileged command, verify the expected device, service, or file appears before continuing. Roll back by undoing the config change or stopping the service you just enabled.

The `.ko` files are scattered across the build tree. Before they are useful on the target, they need to be collected into a `/lib/modules/<version>/` hierarchy:

```
$ make INSTALL_MOD_PATH=~/imx6ull/rootfs modules_install
$ ls ~/imx6ull/rootfs/lib/modules/
6.6.0/
$ ls ~/imx6ull/rootfs/lib/modules/6.6.0/kernel/drivers/ | head
acpi
ata
auxdisplay
base
block
bluetooth
...
```

`INSTALL_MOD_PATH=~/imx6ull/rootfs` is the path that becomes `/` on the target, usually your NFS-exported rootfs directory (Chapter 24). The `make` rule also generates `modules.dep`, `modules.alias`, and a few other index files so `modprobe` works on the target.

> **MCU bridge:** Think of the rootfs as the firmware image's file-backed runtime environment. On an MCU you link everything into flash. On Linux, programs and config live in this mounted tree. **NFS:** Network File System, which lets the target mount a host directory over Ethernet during development. **rootfs:** root filesystem, the directory tree mounted at / that contains /bin, /etc, /dev, and libraries.

25.5 zImage vs vmlinux vs Image, what compresses what

A short sketch of the wrapping:

```
              vmlinux (ELF with symbols, 145 MB)
                       │
                 strip + objcopy
                       ▼
       arch/arm/boot/Image (raw binary, ~16 MB)
                       │
                 gzip compression
                       ▼
                 compressed/piggy.gzip (~6 MB)
                       │
```

```
   prepend decompressor stub (compressed/head.S + misc.c)
                       ▼
        arch/arm/boot/zImage (~6 MB)
```

When U-Boot `bootz`'s a `zImage`:

1. U-Boot loads the `zImage` into DRAM at the address told (typically `0x82000000`).
2. U-Boot transfers control to the first instruction of `zImage`, which is the **decompressor stub** in `arch/arm/boot/compressed/head.S`.
3. The stub copies itself out of the way, sets up a small workspace, and decompresses the gzipped kernel into the final run-from address.
4. The stub jumps to the decompressed kernel's entry point `stext` (Chapter 28). You see the famous *"Uncompressing Linux... Done, booting the kernel."* message.
5. The decompressor stub is now discarded. The kernel runs from where the stub decompressed it to.

This is why the kernel works regardless of whether you store it compressed or not, the kernel image carries its own decompressor.

25.6 Sanity check the build

```
$ file arch/arm/boot/zImage
arch/arm/boot/zImage: Linux kernel ARM boot executable zImage (little-endian)

$ file vmlinux
vmlinux: ELF 32-bit LSB executable, ARM, EABI5 version 1 (SYSV),
         statically linked, BuildID[sha1]=..., with debug_info, not stripped

$ arm-none-linux-gnueabihf-readelf -h vmlinux | head
ELF Header:
  ...
  Entry point address:               0x80008000
  ...

$ ls arch/arm/boot/dts/nxp/imx/imx6ull-14x14-evk.dtb
arch/arm/boot/dts/nxp/imx/imx6ull-14x14-evk.dtb

$ dtc -I dtb -O dts arch/arm/boot/dts/nxp/imx/imx6ull-14x14-evk.dtb | head -30
/dts-v1/;
/ {
    #address-cells = <0x01>;
    #size-cells = <0x01>;
    interrupt-parent = <0x01>;
    compatible = "fsl,imx6ull-14x14-evk", "fsl,imx6ull";
    model = "Freescale i.MX6 ULL 14x14 EVK Board";
    ...
};
```

The `dtc` reverse-compile is a useful sanity check that the DT compiled correctly. We will walk this DT in detail in Chapter 27.

25.7 Make this fit your workspace

A practical workspace recipe:

```
$ mkdir -p ~/imx6ull/{src,build,rootfs}
$ cd ~/imx6ull/src
$ git clone --depth=1 --branch v6.6 \
      https://git.kernel.org/pub/scm/linux/kernel/git/stable/linux.git linux-v6.6
$ cd linux-v6.6

# Build out-of-tree to keep the source clean (optional but recommended)
$ make O=~/imx6ull/build/kernel imx_v6_v7_defconfig
$ make O=~/imx6ull/build/kernel -j$(nproc) zImage modules dtbs

# Symlink artefacts into TFTP and rootfs
$ ln -sf ~/imx6ull/build/kernel/arch/arm/boot/zImage /srv/tftp/zImage
$ ln -sf ~/imx6ull/build/kernel/arch/arm/boot/dts/nxp/imx/imx6ull-14x14-evk.dtb /srv/
tftp/imx6ull.dtb

# Install modules into NFS-exported rootfs
$ make O=~/imx6ull/build/kernel INSTALL_MOD_PATH=~/imx6ull/rootfs modules_install
```

`O=...` puts every generated file in a sibling directory. The source tree stays bit-identical to what `git` checked out. Cleanups are `rm -rf ~/imx6ull/build/kernel`.

25.8 Key Notes

- A kernel build is defined by its source revision, resolved `.config` and toolchain. An output directory separates generated state from source, but does not make different configurations interchangeable.
- `vmlinux` carries symbols for host-side diagnosis, while `zImage` packages the ARM32 kernel and its decompressor for loading. Their different sizes reflect different roles, not different kernel features.
- One multiplatform kernel can support several boards through separate DTBs. An EVK or Colibri DTB is a hardware description for that board, not evidence of a matching MINI description.
- Built-in features are present before rootfs access. Loadable `.ko` files require a matching kernel and a reachable module tree, with indexes used by `modprobe`.
- `INSTALL_MOD_PATH` selects the target rootfs staging tree. Omitting it crosses into the host's module installation path rather than preparing the target.

- Successful artifact generation establishes build output, not correct board wiring, DDR behavior or a successful Linux boot.

25.9 Pitfalls

> **Driver choice:** Use the in-tree, maintained driver first. Use out-of-tree, spidev, or custom-driver paths only after you accept the kernel-version maintenance cost and document who owns updates.

- **Forgetting the Chapter 3 environment.** If you do not run `. ~/imx6ull/scripts/env.sh`, `make` will build for the host x86-64 machine and fail somewhere deep in arch code. Source the script before invoking `make`.
- **Building from the source tree without `O=`.** Works, but `git status` becomes useless because every `make` populates the source tree with `.o` files. Out-of-tree builds keep the source pristine.
- **Wrong defconfig.** `make imx_v6_v7_defconfig` not `make x86_64_defconfig`. The latter happens when you forget to export `ARCH=arm`, Linux helpfully picks the host default.
- **Old gcc-toolchain miscompile.** Mainline kernels usually require a fairly recent gcc (≥ 5.1 for v6.x. ≥ 4.9 for older). The Arm GNU Toolchain from Chapter 3 is fine. Custom-built ancient toolchains sometimes miscompile RCU or AAPCS-sensitive code paths.
- **`make modules_install` to a system location.** By default `make modules_install` writes to `/lib/modules/$(uname -r)/`. **Always** pass `INSTALL_MOD_PATH=...` when cross-building or you will overwrite your host's modules.
- **Mismatch between `zImage` and `modules`.** Modules built against kernel version X will refuse to load on a running kernel built from version Y (they check the version's "vermagic" string). If you rebuild the kernel, rebuild + reinstall modules.

25.10 Going deeper

- **`Documentation/admin-guide/README.rst`** in the kernel tree, the upstream-maintained README. Read it once.
- **`Documentation/process/`**: how the community works (`coding-style.rst`, `submitting-patches.rst`, `4.Coding.rst`).
- **`Documentation/kbuild/`**: the kernel build system. `kconfig.rst` and `makefiles.rst` are the most useful.
- **`Documentation/arch/arm/`**: ARM-specific docs, including the `.dts` → `.dtb` flow.
- **kernelnewbies.org**: the friendliest entry point for new kernel hackers.

- **The kernel mailing list archive** at `lore.kernel.org`, search `[PATCH] imx6ull` to read every recent i.MX6ULL change discussion.

> Next chapter: **Chapter 26: Booting the kernel from U-Boot.** The boot case study follows a `zImage` through the firmware handoff and interprets the kernel's early output.

Chapter 26: Booting the kernel from U-Boot

> **IRQ:** interrupt request, the signal path that tells the CPU or interrupt controller that hardware needs service. **DMA:** Direct Memory Access. Hardware moves data to or from memory without the CPU copying each byte. **Buildroot:** a configuration-driven build system that produces a complete root filesystem and related images.

> **What:** the U-Boot-to-Linux handoff, illustrated by the `zImage` and EVK DTB example from Chapter 25, and the meaning of early UART output. The EVK example is not a MINI boot qualification. **TFTP:** Trivial File Transfer Protocol, a simple network protocol U-Boot commonly uses to fetch kernels from the host. **U-Boot:** the bootloader that initializes enough hardware to load and start the Linux kernel.
>
> **Why:** after the handoff, Linux owns initialization. Its messages expose which interfaces and subsystems have become usable.
>
> **Focus:** the **kernel boot log** as a diagnostic instrument. Every line means something specific. Every successful boot prints predictable lines in predictable order. If you can recognise the first 30 lines, you can recognise which of them is missing or wrong on a board that’s not booting.

26.1 The pre-boot contract

The kernel expects three things at the moment U-Boot transfers control:

Condition	Value	How U-Boot establishes it
CPU state	SVC mode, IRQ/FIQ masked, MMU off, caches off	`bootz` calls `cleanup_before_linux()` before the jump
`r0`	0 (or boot-magic on some platforms; 0 is safe)	`bootz` sets
`r1`	“Machine number” for legacy ATAGS path; ignored on DT systems	`bootz` sets

Condition	Value	How U-Boot establishes it
r2	**Physical address of the DTB**	`bootz` sets to the DTB load address

The first instruction the kernel executes is `stext` (in `arch/arm/kernel/head.S`). It begins by reading `r2` to find the DTB. If `r2` is wrong, the kernel cannot parse its hardware description and dies silently (no UART, no output, no diagnostic, because UART has not been initialised yet).

This is the *one* hardware contract every ARM Linux boot relies on. If `r2` is correct, the kernel almost always boots. If `r2` is wrong, you see nothing on the UART.

26.2 The three-command boot

From the U-Boot prompt:

```
=> tftp 0x82000000 zImage
=> tftp 0x83000000 imx6ull.dtb
=> bootz 0x82000000 - 0x83000000
```

What each does:

1. `tftp 0x82000000 zImage`, pulls the kernel from your TFTP server into DRAM at `0x82000000`. The address was chosen for two reasons: it’s far enough above the DRAM base (`0x80000000`) that U-Boot’s image (currently relocated near the top of DRAM) doesn’t conflict, and it’s far enough below that the kernel has room to decompress upward into.
2. `tftp 0x83000000 imx6ull.dtb`, pulls the DT blob to a second location. ~50 KB.
3. `bootz 0x82000000 - 0x83000000`, start a zImage at `0x82000000`, no initrd (`-`), DTB at `0x83000000`. This is the key step. U-Boot sets `r2 = 0x83000000`, jumps to the kernel, and is done.

You can save these as an env one-shot:

```
=> setenv bootnet 'tftp 0x82000000 zImage && tftp 0x83000000 imx6ull.dtb && bootz
0x82000000 - 0x83000000'
=> setenv bootcmd 'run bootnet'
=> saveenv
```

Now every power-on auto-boots over the network.

26.3 The cmdline

Before `bootz`, set `bootargs`:

```
=> setenv bootargs 'console=ttymxc0,115200 earlycon root=/dev/mmcblk0p2 rw rootwait'
=> saveenv
```

Token by token:

- **`console=ttymxc0,115200`**: once the i.MX UART driver loads, route `printk` to UART1 at 115200 baud. *If this token is wrong, you see no kernel output.* The driver name `ttymxc0` is the i.MX-specific convention. Other SoCs use `ttyS0`, `ttyAMA0`, etc.
- **`earlycon`**: very early UART printk *before* the full driver loads. Reads the DT's `chosen.stdout-path` to find which UART. Without `earlycon`, the first ~10 boot lines stay in a buffer. You see them only when the regular console driver loads.
- **`root=/dev/mmcblk0p2`**: what device holds the rootfs. We will return to this in Part V. A panic at this stage can show that the kernel reached root mounting, but does not establish a complete system boot.

> **MCU bridge:** Think of the rootfs as the firmware image's file-backed runtime environment. On an MCU you link everything into flash. On Linux, programs and config live in this mounted tree. **rootfs:** root filesystem, the directory tree mounted at / that contains /bin, /etc, /dev, and libraries.

- **`rw`**: mount the root read-write.
- **`rootwait`**: don't panic if `root=` isn't immediately ready. Wait. Always safe to include.

A development cmdline, with NFS root, looks like:

> **NFS:** Network File System, which lets the target mount a host directory over Ethernet during development.

```
console=ttymxc0,115200 earlycon
root=/dev/nfs nfsroot=192.168.7.1:/home/you/imx6ull/rootfs,vers=3,nolock,tcp
ip=192.168.7.2:192.168.7.1:192.168.7.1:255.255.255.0::eth0:off
rw rootwait
```

Chapter 31 uses this form to explain a network-backed rootfs.

26.4 What you should see

```
=> run bootnet
Using FEC0 device
TFTP from server 192.168.7.1; our IP address is 192.168.7.2
Filename 'zImage'.
Load address: 0x82000000
Loading: ########################  4.5 MiB/s  done
Bytes transferred = 6291456 (600000 hex)
Using FEC0 device
TFTP from server 192.168.7.1; our IP address is 192.168.7.2
Filename 'imx6ull.dtb'.
Load address: 0x83000000
Loading: #  100 KiB/s  done
Bytes transferred = 56320 (dc00 hex)
## Flattened Device Tree blob at 83000000
   Booting using the fdt blob at 0x83000000
   Loading Device Tree to 8ffec000, end 8ffffdff ... OK

Starting kernel ...

[    0.000000] Booting Linux on physical CPU 0x0
[    0.000000] Linux version 6.6.0 (you@host) (arm-none-linux-gnueabihf-gcc 11.4.0) ...
[    0.000000] CPU: ARMv7 Processor [410fc075] revision 5 (ARMv7), cr=10c5387d
[    0.000000] CPU: div instructions available: patching division code
[    0.000000] CPU: PIPT / VIPT nonaliasing data cache, VIPT aliasing instruction cache
[    0.000000] OF: fdt: Machine model: Freescale i.MX6 ULL 14x14 EVK Board
[    0.000000] Memory policy: Data cache writealloc
[    0.000000] efi: UEFI not found.
[    0.000000] cma: Reserved 64 MiB at 0x0000000094000000
[    0.000000] Zone ranges:
[    0.000000]   Normal   [mem 0x0000000080000000-0x000000009fffffff]
[    0.000000]   HighMem  empty
[    0.000000] Movable zone start for each node
[    0.000000] Early memory node ranges
[    0.000000]   node   0: [mem 0x0000000080000000-0x000000009fffffff]
[    0.000000] Initmem setup node 0 [mem 0x0000000080000000-0x000000009fffffff]
[    0.000000] percpu: Embedded 13 pages/cpu s24336 r8192 d20720 u53248
[    0.000000] Built 1 zonelists, mobility grouping on.  Total pages: 130048
[    0.000000] Kernel command line: console=ttymxc0,115200 earlycon root=/dev/mmcblk0p2
rw rootwait
[    0.000000] Dentry cache hash table entries: 65536 (order: 6, 262144 bytes, linear)
[    0.000000] Inode-cache hash table entries: 32768 (order: 5, 131072 bytes, linear)
[    0.000000] mem auto-init: stack:all(zero), heap alloc:off, heap free:off
[    0.000000] Memory: 444184K/524288K available (10240K kernel code, 1112K rwdata,
                3144K rodata, 1024K init, 268K bss, 14760K reserved, 65536K cma-reserved,
                0K highmem)
[    0.000000] SLUB: HWalign=64, Order=0-3, MinObjects=0, CPUs=1, Nodes=1
...
```

Each line exposes a different part of the boot state:

Line	What it tells you
`Booting Linux on physical CPU 0x0`	CPU0 has booted (SMP would say `0x0–0xN`). On i.MX6ULL there's only one core.
`Linux version 6.6.0 ...`	The kernel version + the toolchain that built it. Cross-check this against your build.
`CPU: ARMv7 Processor [410fc075]`	The MIDR (Main ID Register) of the core. `0x410FC075` decodes as: implementer `0x41` (ARM), variant `0xF`, architecture `0xC`, primary part `0xC07` (Cortex-A7), revision `r0p5`.
`CPU: div instructions available`	The CPU has hardware integer division. Some Cortex-A profiles don't.
`CPU: PIPT / VIPT nonaliasing data cache, VIPT aliasing instruction cache`	The cache aliasing properties. Matters for DMA correctness.
`OF: fdt: Machine model: Freescale i.MX6 ULL 14 x14 EVK Board`	**The model string from the DT root node.** Confirms the right DTB loaded.
`Memory policy: Data cache writealloc`	The cache write policy the kernel chose.
`cma: Reserved 64 MiB at 0x94000000`	Contiguous Memory Allocator carved out 64 MiB. Used for big DMA buffers (framebuffer, camera).
`Zone ranges: Normal [mem 0x80000000-0x9fffffff]`	The 512 MiB of DRAM, mapped as Normal cacheable memory (Ch 17).
`Kernel command line: ...`	The literal `bootargs` U-Boot passed. **Verify this matches what you set.**
`Memory: 444184K/524288K available ...`	Of 524 288 KB DRAM, 444 184 KB is usable for user memory; the rest is kernel code/data/reserved.
`SLUB: HWalign=64, Order=0-3`	The slab allocator is up. CPU's cache-line size is 64 bytes.

After Memory: SLUB, output continues for another 50–100 lines as drivers probe. The kernel reaches user-space when it prints:

```
[    2.158972] EXT4-fs (mmcblk0p2): mounted filesystem with ordered data mode...
[    2.193456] VFS: Mounted root (ext4 filesystem) on device 179:2.
[    2.198321] Run /sbin/init as init process

Welcome to Buildroot
buildroot login:
```

That last “Mounted root” line is the threshold: the kernel has finished its own initialisation and is now executing user-space. From here, Part V takes over.

26.5 When it doesn’t boot, what to look for

If you see *nothing at all* after `Starting kernel ...`:

- **The DTB address in `r2` is wrong.** Usually `bootz 0x82000000 - 0x83000000` is correct. If you type `bootz 0x82000000 0x83000000` (no `-`), U-Boot reads `0x83000000` as the initrd address. The kernel then gets no DTB. Symptom: silence.
- **Wrong DTB for the board.** Kernel finds *a* DTB but it describes hardware the actual board doesn’t have. Symptom: silence after `Starting kernel ...`. Cross-check the DT model line by trying earlycon (see below).
- **`console=` token wrong.** Kernel boots fine. UART driver loads. But printk is redirected somewhere else. Symptom: nothing after `Starting kernel ...`. Add `earlycon` to bootargs to see *very* early printk before the driver loads, if those appear, the regular console is the problem.
- **DDR not all working.** The kernel does an early memtest of sorts. If DRAM has bit errors it usually panics early but the panic might not reach the UART. Rerun the U-Boot `mtest` first.

> **DDR:** external DRAM that must be configured and trained before most software can run from it.

If you see *some output then silence*:

- **Driver hang.** Look at the *last* line printed. The next subsystem to probe is likely hanging. A common cause is the PMIC on I²C. If I²C is broken, the regulators stay off. Devices fail to enumerate, and the kernel hangs.

> **MCU bridge:** Think of a PMIC like a programmable power-tree supervisor: it replaces discrete enables and LDO assumptions with sequenced rails the kernel can model. **PMIC:** Power Management IC, a chip that sequences and regulates the board's voltage rails.

- **VFS panic** ("Cannot open root device 'mmcblkXpY'"): rootfs not found. The panic message is clear. Fix the `root=` argument.
- **`Kernel panic - not syncing: VFS: Unable to mount root fs`**: same as above. The kernel says exactly what's wrong.

26.6 Verifying with earlycon

`earlycon` activates a tiny inline UART driver during `start_kernel()` (Chapter 28), long before the regular `console=` driver loads. It's invaluable for diagnosing "kernel boots but I see no output" issues, because earlycon is much harder to misconfigure.

Add to bootargs:

```
earlycon
```

...or explicitly:

```
earlycon=ec_imx6q,0x02020000
```

The first form reads the UART address from the DT's `chosen.stdout-path` (an i.MX6ULL DTB already sets this). The second form pins it explicitly to the UART1 register base.

With earlycon active, you'll see ~5 extra lines printed *before* the normal "Booting Linux on physical CPU 0x0", these are emitted by setup_arch() before the regular console driver loads.

26.7 Key Notes

- U-Boot owns image loading and the CPU handoff state. Linux then owns its memory mappings, interrupt handling and driver initialization.
- `bootz` has separate kernel, initrd and DTB arguments. The `-` placeholder expresses the absence of an initrd and keeps the DTB in the correct argument position.
- Kernel, DTB and decompression workspace must occupy compatible memory regions. A valid DTB can still be corrupted by an overlapping image or workspace.

- `earlycon`, the regular `console=` argument and DT `stdout-path` describe related but distinct console paths. Missing UART output alone does not establish that the CPU stopped.
- The model, memory and command-line messages identify what Linux was given. They do not prove that the DT accurately describes the populated MINI board.
- Root mounting and executing init are later boundaries than kernel entry. A VFS panic narrows the failure stage, but is not a successful boot to user space.

26.8 Pitfalls

- **DT load address conflicts with kernel decompression area.** If the kernel decompresses to a region that overlaps where you loaded the DTB, the DT gets corrupted partway through boot and the kernel hangs at a random point. The address `0x83000000` for DTB is safe because the kernel decompresses from `0x82000000` upward but stops well before 16 MiB (the kernel is < 16 MiB). For *very* large kernels (CONFIG_DEBUG_INFO, huge configs), use `0x88000000` for DTB instead.
- **Forgetting `cleanup_before_linux()`.** U-Boot's `bootz` does this automatically. If you wrote your own jump-to-kernel code (don't), you need to flush caches and disable MMU before transferring.

> **MMU:** Memory Management Unit, hardware that translates virtual addresses to physical addresses and enforces permissions.

- **Kernel built for a different ARM revision.** A `zImage` built with `CONFIG_ARCH_MULTI_V6_V7` or `CONFIG_ARCH_MULTI_V7` runs on Cortex-A7. (`CONFIG_ARCH_MULTI_V7_ONLY` is *not* a mainline symbol, earlier drafts of this chapter listed it. Ignore.) A 64-bit kernel (`CONFIG_ARM64`) will not run on Cortex-A7. Symptom: undefined instruction at `stext`. A Thumb-2-only kernel (`CONFIG_THUMB2_KERNEL=y`) requires the bootloader to enter in Thumb state. If your bootloader hands off in ARM state to a Thumb kernel, you fault on the very first instruction.
- **Wrong board's DTB.** Loading the i.MX8MP EVK DTB on an i.MX6ULL board: the kernel reads the DT's `compatible` root property, looks for `fsl,imx6ull` (or the matching SoC), doesn't find it, and panics in `setup_machine_fdt()`. Sometimes silently.
- **`root=` pointing at something not ready by the time VFS mounts root.** USB-stick root devices need `rootwait` because USB enumeration is slow. SD cards are usually fast enough that you can skip `rootwait`, but always safe to add.

26.9 Going deeper

- **`arch/arm/boot/compressed/head.S`**: read the decompressor stub. It is short and educational.

- **`init/main.c`**: the file `start_kernel()` lives in. We trace it line-by-line in Chapter 28.
- **`Documentation/admin-guide/kernel-parameters.txt`**: every cmdline parameter the kernel understands. ~1500 lines. Skim the headers. You'll come back for specific tokens.
- **`Documentation/arch/arm/booting.rst`**: the boot contract (`r0`/`r1`/`r2`) in the canonical place.
- **The kernel's `printk` format**: `<5>` (KERN_NOTICE), `<6>` (KERN_INFO), `<7>` (KERN_DEBUG) prefix codes. Mostly invisible at boot. Visible when you use `dmesg --level=info` etc.

> Next chapter: **Chapter 26A: Kernel boot failure playbook.** Before we move deeper into Device Tree, we learn how to debug the common failures between `bootz` and the first shell.

Chapter 26A: Kernel boot failure playbook

> **What:** a practical checklist for the failures that happen between `bootz` and the first shell.
>
> **Why:** when Linux does not boot, the symptom is often vague: silence, a panic, or one last line that looks unrelated. This chapter gives you a repeatable debug order.
>
> **Result:** you can look at the last visible line and choose the next test instead of changing random settings.
>
> **Focus:** classify the failure first: no output, early output then silence, VFS panic, init panic, or driver probe failure.

The cases use the kernel artifacts described in Chapter 25 and the boot command structure from Chapter 26. They explain diagnostic boundaries without requiring a completed board boot.

26A.1 Keep one known-good boot command

Before debugging anything, save a known-good command that uses `&&`. If a load fails, the boot stops instead of jumping into stale RAM.

```
=> setenv boot_good 'tftp 0x82000000 zImage && tftp 0x83000000 imx6ull.dtb && bootz
0x82000000 - 0x83000000'
=> setenv bootargs_good 'console=ttymxc0,115200 earlycon root=/dev/mmcblk0p2 rw rootwait'
=> saveenv
```

Then boot with:

```
=> setenv bootargs ${bootargs_good}
=> run boot_good
```

If you later break `bootcmd`, you still have a known-good path.

26A.2 Symptom table

Symptom	Most likely area	First test
Nothing after `Starting kernel ...`	DTB address, early console, CPU entry state	Add explicit `earlycon=ec_imx6q,0x02020000`
A few early lines, then silence	Console handoff or driver hang	Add `ignore_loglevel loglevel=8 initcall_debug`
`VFS: Cannot open root device`	`root=` or storage driver	Boot initramfs from Chapter 29
`No working init found`	rootfs content or init path	Try `init=/bin/sh` or `rdinit=/init`
Ethernet/NFS root hangs	network driver, IP args, server export	Boot SD root first, then test NFS
Random panics	DDR, bad DT memory node, kernel/module mismatch	Run U-Boot `mtest`; check `Memory:` line

The last visible line locates a failure boundary, which makes the table useful for choosing a diagnostic path.

26A.3 Case 1: nothing after `Starting kernel ...`

Start with the three boot arguments that maximize early visibility:

```
=> setenv bootargs 'console=ttymxc0,115200 earlycon=ec_imx6q,0x02020000 ignore_loglevel
loglevel=8 root=/dev/mmcblk0p2 rw rootwait'
=> run boot_good
```

If output appears now, the kernel was running. Your normal console path was wrong.

Check:

```
=> fdt addr 0x83000000
=> fdt print /chosen
=> fdt print /aliases
```

You want `/chosen/stdout-path` to name the same UART you use for serial.

Common causes:

- Wrong third argument to `bootz`.
- Missing `-` when there is no initrd.
- Wrong DTB for the board.
- Bad `stdout-path`.
- U-Boot loaded the DTB to an address overwritten by kernel decompression.

The correct no-initrd shape is:

```
bootz kernel_addr - dtb_addr
```

The correct initramfs shape is:

```
bootz kernel_addr initrd_addr:initrd_size dtb_addr
```

26A.4 Case 2: early output, then silence

When early messages appear but normal output disappears, the early console worked and the regular console did not.

Compare three values:

Place	What to check
U-Boot `bootargs`	`console=ttymxc0,115200`
DT `/chosen`	`stdout-path = &uart1;` or `serial0:115200n8`
Linux driver	i.MX UART driver enabled and UART node `status = "okay"`

Useful bootargs:

```
console=ttymxc0,115200 earlycon ignore_loglevel loglevel=8
```

If the log stops near a specific driver, add:

```
initcall_debug
```

Then search the log:

```
target# dmesg | grep initcall
```

The last initcall before the stop is your suspect.

26A.5 Case 3: VFS cannot mount root

The kernel booted. It reached the point where it needs a root filesystem.

Typical panic:

```
VFS: Cannot open root device "mmcblk0p2" or unknown-block(179,2)
Kernel panic - not syncing: VFS: Unable to mount root fs
```

Read it literally:

Message part	Meaning
`mmcblk0p2`	The requested root device.
`unknown-block(179,2)`	The block major/minor the kernel tried.
`Unable to mount root fs`	Device missing, driver missing, wrong filesystem, or partition not ready.

Checklist:

1. Add `rootwait`.
2. Confirm the DT enables the SD/eMMC controller.
3. Confirm `CONFIG_MMC` and `CONFIG_MMC_SDHCI_ESDHC_IMX`.
4. Confirm `CONFIG_EXT4_FS` if the root partition is ext4.
5. Boot the Chapter 29 initramfs and run `cat /proc/partitions`.

If initramfs boots but SD root does not, the kernel is fine. The problem is storage, partitioning, or `root=`.

26A.6 Case 4: no working init

Typical panic:

```
Run /sbin/init as init process
Failed to execute /sbin/init (error -2)
Kernel panic - not syncing: No working init found.
```

Error `-2` is `ENOENT`: file not found.

Try a shell:

```
=> setenv bootargs 'console=ttymxc0,115200 earlycon root=/dev/mmcblk0p2 rw rootwait
init=/bin/sh'
=> run boot_good
```

If `/bin/sh` runs, your rootfs exists but its init system is missing or broken.

If `/bin/sh` also fails:

- The rootfs may not contain `/bin/sh`.
- The binary may be for the wrong architecture.
- Dynamic libraries may be missing.
- `/dev/console` may be missing if devtmpfs is disabled.

Check from an initramfs:

```
# mount /dev/mmcblk0p2 /mnt
# ls -l /mnt/sbin/init /mnt/bin/sh
# file /mnt/bin/sh
# ls /mnt/lib
```

26A.7 Case 5: NFS root hangs

First prove local boot works. Then prove network from U-Boot. Then try NFS root.

Known-good NFS bootargs shape:

```
console=ttymxc0,115200 earlycon
root=/dev/nfs
nfsroot=192.168.7.1:/home/you/imx6ull/rootfs,vers=3,nolock,tcp
ip=192.168.7.2:192.168.7.1:192.168.7.1:255.255.255.0:pa-mini:eth0:off
rw rootwait
```

Common mistakes:

- Host firewall blocks NFS.
- `/etc/exports` path is wrong.
- Missing `no_root_squash` on a development export.

- Kernel lacks `CONFIG_IP_PNP` or NFS root support.
- Wrong Ethernet DT pinctrl or PHY reset GPIO.

For development, use NFSv3 first. It is simpler to debug on small embedded boards.

26A.8 Minimum debug bootargs

Keep these three sets.

Normal:

```
console=ttymxc0,115200 root=/dev/mmcblk0p2 rw rootwait
```

Verbose:

```
console=ttymxc0,115200 earlycon ignore_loglevel loglevel=8 root=/dev/mmcblk0p2 rw
rootwait
```

Probe timing:

```
console=ttymxc0,115200 earlycon ignore_loglevel loglevel=8 initcall_debug root=/dev/
mmcblk0p2 rw rootwait
```

Use the smallest one that answers the question.

26A.9 Key Notes

- A boot failure has a stage: image handoff, console availability, root mounting, init execution or device probing. Similar visible symptoms can originate in different stages.
- Chained loads with `&&` prevent a failed transfer from being followed by a jump into stale RAM. A diagnostic baseline also needs a matching image, DTB and command line.
- Early output followed by silence can indicate console handoff trouble rather than a stopped kernel. `initcall_debug` adds evidence about initialization progress, not proof that the last printed driver caused the failure.
- A VFS panic concerns the path to the root device and filesystem support. `rootwait` handles delayed discovery, but cannot supply a missing driver or correct a wrong partition.

- Init execution depends on the executable path, architecture and, for dynamic binaries, interpreter and libraries. Finding the file in the rootfs is only one of these conditions.
- An initramfs separates early user-space availability from the normal rootfs path. NFS adds independent dependencies on Linux networking, host exports and firewall policy.

26A.10 Pitfalls

- **Changing three things at once.** Change one variable, boot, save the log.
- **Ignoring the last line.** The last printed line is often the clue.
- **Using semicolons in boot scripts.** Use `&&` for commands that must stop on failure.
- **Assuming silence means dead CPU.** Often the kernel is running, but the console path is wrong.
- **Debugging rootfs before kernel boot.** If you have not seen the `VFS:` lines, you are not debugging rootfs yet.

26A.11 Going deeper

- `Documentation/admin-guide/kernel-parameters.txt`, for bootargs.
- `Documentation/arch/arm/booting.rst`, for the ARM boot contract.
- `init/do_mounts.c`, for root filesystem mounting.
- `init/main.c`, for init selection and `kernel_init()`.

Previous: Chapter 26: Booting the kernel from U-Boot

Next: Chapter 27: Device Tree: the contract between firmware and kernel

Chapter 27: Device Tree: the contract between firmware and kernel

> **What:** the **Device Tree**, its origin, grammar and standard properties, how it is compiled (`dtc`) and consumed (`of_*` APIs), and how drivers bind through `compatible`. The EVK and optional I²C sensor cases illustrate this hardware-description interface. **Device Tree:** a data file that describes board hardware to the Linux kernel instead of hard-coding it in C.
>
> **Why:** DT is the biggest mental shift in this chapter. There is no longer a hand-written `board-*.c` with platform device tables. There is a `.dts` file that describes the hardware, and the kernel matches drivers to nodes by string at runtime. Understanding this dynamic-binding model is the prerequisite for every chapter in Part VI.
>
> **Focus:** the **`compatible` string** as the keystone. Compatible-strings in DT nodes are matched against compatible-strings in driver source code. That single mechanism is how every driver in mainline finds its hardware. Once you have this, the rest of DT is just grammar.

27.1 Why the Device Tree exists

Before the Device Tree, every ARM board in mainline shipped a hand-written C file under `arch/arm/mach-<soc>/board-<name>.c` containing struct-array declarations like:

```
static struct platform_device smdk2440_devices[] = {
    &s3c_device_ohci,
    &s3c_device_lcd,
    &s3c_device_wdt,
    &s3c_device_i2c0,
    &s3c_device_iis,
};
static struct s3c2410_uartcfg smdk2440_uartcfgs[] = {
    [0] = { .hwport = 0, .flags = 0, .ucon = 0x3c5, ... },
    ...
};
```

Each board got its own ~500-line C file, hand-written, compiled into the kernel. By 2010 the ARM `arch/` tree held thousands of such files and Linus Torvalds publicly complained that ARM was “a fucking pain in the ass”. The community’s response was to adopt the **Device Tree**, which had been used on PowerPC since the early 2000s. PowerPC in turn borrowed it from Open Firmware on Sun and Apple machines.

The premise of DT is simple: instead of describing hardware in C code that gets compiled into the kernel, describe it in a structured *text* file (`.dts`) that gets compiled separately into a *binary* blob (`.dtb`). The kernel reads the blob at boot time, builds an in-memory representation, and matches drivers to nodes dynamically. One kernel binary now supports thousands of boards because the per-board description lives outside the kernel.

The consequence: when you support a new board variant, you don't recompile the kernel. You write a DTS file. When you change which I²C chip is on which bus, you edit DT, not C. One ARM kernel binary now works across many boards.

27.2 DTS, DTB, DTC, DTSI

Four file extensions you will see:

- **`.dts`**: Device Tree Source. Human-readable text. One file describes one board.
- **`.dtsi`**: Device Tree Source Include. A `.dts` fragment that gets `#include`'d. Used for SoC-wide content shared by every board with that SoC.
- **`.dtb`**: Device Tree Blob. Binary form. What the kernel actually consumes at boot.
- **`.dtbo`**: Device Tree Overlay. A binary fragment that patches a base `.dtb` at runtime (Ch 24E).

The compiler is `dtc` (Device Tree Compiler), in `scripts/dtc/`:

```
$ make dtbs           # compile every DTS the current arch defines
$ make dtbs_check     # additionally validate against YAML schemas (Ch 27A)
```

On v6.6 and newer, the i.MX DTS files live under `arch/arm/boot/dts/nxp/imx/`, and each board DTB is listed in `arch/arm/boot/dts/nxp/imx/Makefile`:

```
dtb-$(CONFIG_SOC_IMX6ULL) += \
    imx6ull-14x14-evk.dtb \
    imx6ull-colibri-eval-v3.dtb \
    imx6ull-9x9-evk.dtb \
    ...
```

To add a board, you add the `.dts` file and one line here.

27.3 Anatomy of a DTS file

Let's read `imx6ull-14x14-evk.dts` from the top. (Specific line numbers vary by kernel version. The structure is stable.)

```
/dts-v1/;
```

```
#include <dt-bindings/input/input.h>
#include "imx6ull.dtsi"

/ {
    model = "Freescale i.MX6 ULL 14x14 EVK Board";
    compatible = "fsl,imx6ull-14x14-evk", "fsl,imx6ull";

    chosen {
        stdout-path = &uart1;
    };

    memory@80000000 {
        device_type = "memory";
        reg = <0x80000000 0x20000000>;   /* 512 MiB */
    };

    reg_sd1_vmmc: regulator-sd1-vmmc {
        compatible = "regulator-fixed";
        regulator-name = "VSD_3V3";
        regulator-min-microvolt = <3300000>;
        regulator-max-microvolt = <3300000>;
        gpio = <&gpio1 9 GPIO_ACTIVE_HIGH>;
        enable-active-high;
    };

    /* ... more nodes ... */
};

&uart1 {
    pinctrl-names = "default";
    pinctrl-0 = <&pinctrl_uart1>;
    status = "okay";
};

&fec1 {
    pinctrl-names = "default";
    pinctrl-0 = <&pinctrl_enet1>;
    phy-mode = "rmii";
    phy-handle = <&ethphy0>;
    status = "okay";
    /* ... */
};
```

Six things going on:

1. **`/dts-v1/;`**: Version marker. Always present.
2. **`#include`**: Yes, DTS supports C-preprocessor includes. `dt-bindings/input/input.h` is a header that defines named constants (`KEY_ENTER`, etc.) used in DT.
3. **`#include "imx6ull.dtsi"`**: Pulls in the SoC-wide DT fragment. *This is where most nodes actually live.* The board-level `.dts` mostly references and patches them.
4. **`/ { ... };`**: The **root node**. Every DT has exactly one. Inside it are all the other nodes (in a tree).

5. **`&uart1 { ... };`**: A **reference** to a node defined in the included `.dtsi`. The `&label` syntax says "modify the node previously labelled `uart1`". This is the canonical pattern: SoC `.dtsi` declares the node disabled by default. Board `.dts` references it and sets `status = "okay"` plus board-specific pinctrl.
6. **Properties**: The `key = value;` pairs inside each node.

27.4 Node syntax

Every node has the form:

```
[label:] name[@unit-address] {
    property1 = value;
    property2;
    child-node {
        ...
    };
};
```

Each component:

- **`label`** (optional, before `:`), a phandle/reference target. Lets other parts of the DT refer to this node by `&label`.
- **`name`** (required), a human-readable name. Convention: same as the kind of device (`uart1`, `i2c2`, `ethphy0`).
- **`@unit-address`** (when applicable), the device's base address on its parent bus. For an MMIO peripheral like UART1, this is the register-block base address: `serial@2020000`. The unit-address is for human readability and uniqueness. The actual address used by the kernel comes from the `reg` property.

> **MMIO:** memory-mapped I/O, where software accesses peripheral registers through normal load and store instructions.

Example:

```
uart1: serial@2020000 {
    compatible = "fsl,imx6ul-uart", "fsl,imx21-uart";
    reg = <0x2020000 0x4000>;
    interrupts = <GIC_SPI 26 IRQ_TYPE_LEVEL_HIGH>;
    clocks = <&clks IMX6UL_CLK_UART1_IPG>, <&clks IMX6UL_CLK_UART1_SERIAL>;
    clock-names = "ipg", "per";
    status = "disabled";
};
```

`uart1` is the label. `serial@2020000` is `name@unit-address`. Inside: five properties.

27.5 Property data types

DT properties take five basic data types:

Type	Syntax	Example
Empty (boolean flag)	`key;`	`interrupt-controller;`
String	`key = "value";`	`model = "Freescale i.MX6 ULL ...";`
String list	`key = "v1", "v2";`	`compatible = "fsl,imx6ul-uart", "fsl,imx21-uart";`
Cell (32-bit integer)	`key = <0x123>;`	`reg = <0x2020000 0x4000>;`
Byte sequence	`key = [00 11 22 33];`	`mac-address = [00 04 9f 01 30 ad];`

A “cell” is exactly 32 bits. Multiple cells in one property are space-separated inside `< >`. Properties of arbitrary complexity are built from these primitives.

References use the angle-bracket form together with the `&label` shortcut. They can be richer:

```
clocks = <&clks IMX6UL_CLK_UART1_IPG>, <&clks IMX6UL_CLK_UART1_SERIAL>;
```

This says: two clock entries. Each entry is a reference to the `clks` node plus an integer index. The clock provider (the node labelled `clks`) decides what that integer means. The number of cells per entry comes from a `#clock-cells` property on the provider node (described in the next section).

27.6 Standard properties

The DT specification standardizes a dozen properties that almost every node uses. We will see them in every chapter from here on.

`compatible`, the binding key

```
compatible = "fsl,imx6ul-uart", "fsl,imx21-uart";
```

A list of strings, most-specific first. Each string has the format `vendor,model`. The kernel matches this against drivers' `of_device_id[]` arrays:

```
/* drivers/tty/serial/imx.c */
static const struct of_device_id imx_uart_dt_ids[] = {
    { .compatible = "fsl,imx6q-uart",  .data = &imx_uart_devdata[IMX6Q_UART] },
    { .compatible = "fsl,imx53-uart",  .data = &imx_uart_devdata[IMX53_UART] },
    { .compatible = "fsl,imx1-uart",   .data = &imx_uart_devdata[IMX1_UART]  },
    { .compatible = "fsl,imx21-uart",  .data = &imx_uart_devdata[IMX21_UART] },
    { /* sentinel */ }
};

static struct platform_driver serial_imx_driver = {
    .driver = {
        .name = "imx-uart",
        .of_match_table = imx_uart_dt_ids,
        ...
    },
    .probe = imx_uart_probe,
    ...
};
```

At boot, the kernel walks the DT. For each node with a `compatible` property, it walks through every registered driver's `of_match_table` looking for a match. The first match (matching against the *first* compatible string in the node, then the second, then …) wins. The driver's `probe()` is invoked. *This is how the kernel matches drivers to hardware at boot.*

Reading our example:

- The DT says `compatible = "fsl,imx6ul-uart", "fsl,imx21-uart";`
- The driver's match table doesn't have "fsl,imx6ul-uart" (per the snippet above), so that one doesn't bind.
- It does have "fsl,imx21-uart", so the **second** compatible matches. The `.data` field tells the driver "this is the imx21-uart variant" so a single driver can support multiple SoC revisions with minor parameter differences.

The vendor prefix (`fsl,`) is registered in `Documentation/devicetree/bindings/vendor-prefixes.yaml`. The model name follows. Together they form a globally unique key.

reg, register addresses

```
reg = <0x2020000 0x4000>;          /* one range: 0x2020000, size 0x4000 */
reg = <0x2020000 0x4000>,
      <0x2024000 0x4000>;          /* two ranges, e.g., for split MMIO */
```

A `reg` property is a list of *(base, size)* pairs. **How many 32-bit cells are used for *base* and how many for *size* is determined by the parent node's `#address-cells` and `#size-cells`**, see next section.

`#address-cells` and `#size-cells`

These are properties on **parent** nodes that tell `reg` properties on **child** nodes how many cells to use:

```
soc {
    #address-cells = <1>;   /* every child's reg base is 1 cell (32 bits) */
    #size-cells = <1>;      /* every child's reg size is 1 cell */

    uart1: serial@2020000 {
        reg = <0x2020000 0x4000>;   /* 1 cell base, 1 cell size */
    };
};
```

If `#address-cells = <2>`, the base is a 64-bit value (split as two 32-bit cells, high then low):

```
memory@0 {
    #address-cells = <2>;
    #size-cells = <2>;
    reg = <0x0 0x80000000 0x0 0x20000000>;   /* base = 0x80000000, size = 0x20000000 */
};
```

On i.MX6ULL (32-bit), most ranges fit in 1 cell each. On i.MX8M (64-bit), 2 cells is more common.

`interrupts` and `interrupt-parent`

```
interrupts = <GIC_SPI 26 IRQ_TYPE_LEVEL_HIGH>;
```

The number of cells per IRQ entry is dictated by the **`#interrupt-cells`** property on the *interrupt parent* (the node listed in `interrupt-parent`, which defaults to the root node's `interrupt-parent`). For the GIC, `#interrupt-cells = <3>`, encoding *(IRQ type, IRQ number, trigger type)*. The constants `GIC_SPI` and `IRQ_TYPE_LEVEL_HIGH` come from `dt-bindings/interrupt-controller/`.

> **MCU bridge:** Think of the GIC like the Cortex-M NVIC scaled up for Cortex-A: it routes peripheral interrupts to CPU cores and has separate distributor and CPU-interface blocks. **IRQ:** interrupt request, the signal path that tells the CPU or interrupt controller that hardware needs service. **GIC:** ARM's Generic Interrupt Controller, the Cortex-A interrupt router roughly analogous to NVIC on Cortex-M.

`clocks` and `clock-names`

```
clocks = <&clks IMX6UL_CLK_UART1_IPG>, <&clks IMX6UL_CLK_UART1_SERIAL>;
clock-names = "ipg", "per";
```

Two clocks. The driver gets them with `devm_clk_get(dev, "ipg")` and `devm_clk_get(dev, "per")`. The integer constants `IMX6UL_CLK_*` are named in `include/dt-bindings/clock/imx6ul-clock.h`.

`pinctrl-0`, `pinctrl-1`, …, `pinctrl-names`

```
pinctrl-names = "default", "sleep";
pinctrl-0 = <&pinctrl_uart1>;
pinctrl-1 = <&pinctrl_uart1_sleep>;
```

Each `pinctrl-N` points to a pin-configuration node, and `pinctrl-names` gives each one a symbolic name. The driver picks a state with `pinctrl_select_state(p, "default")`. Default is automatically activated when the driver probes.

`status`

```
status = "okay";      /* device is functional, probe it */
status = "disabled";  /* device exists in DT but kernel should ignore it */
```

Used to enable/disable a device without removing its DT node. SoC `.dtsi` files declare every peripheral as `status = "disabled"` by default. Board `.dts` files set `status = "okay"` on the ones present on that board.

`aliases` and `chosen`

Two special nodes at the root level:

```
/ {
    aliases {
        serial0 = &uart1;
        serial1 = &uart2;
        ethernet0 = &fec1;
    };

    chosen {
        stdout-path = &uart1;
        bootargs = "console=ttymxc0,115200 root=...";
    };
};
```

`aliases` names devices by a stable identifier (“serial0” is always the first UART. “serial1” always the second), regardless of where they sit in the tree.

`chosen` carries arguments to the kernel that aren’t *about* hardware:

- `bootargs`: the kernel cmdline (Ch 26’s `setenv bootargs ...` ends up here).
- `stdout-path`: which device `earlycon` should use.
- `linux,initrd-start` / `linux,initrd-end`, where an initrd lives (Ch 29).

- `kaslr-seed`: random seed for kernel address-space randomization.

27.7 Reading `imx6ull.dtsi`

`imx6ull.dtsi` is the SoC-wide DT. It includes the per-SoC-family fragment `imx6ul.dtsi` and adds the few i.MX6ULL-specific tweaks (some clock gates differ. The SAI audio block has different pinmuxing).

The general structure:

```
/ {
    interrupt-parent = <&gpc>;     /* the General Power Controller, on i.MX6, gates IRQs
*/

    aliases { /* ... */ };

    cpus {
        #address-cells = <1>;
        #size-cells = <0>;
        cpu0: cpu@0 {
            compatible = "arm,cortex-a7";
            device_type = "cpu";
            reg = <0>;
            clocks = <&clks IMX6UL_CLK_ARM>;
            operating-points-v2 = <&cpu0_opp_table>;
            /* ... */
        };
    };

    intc: interrupt-controller@a01000 {
        compatible = "arm,gic-400";   /* current mainline; "arm,cortex-a7-gic" is the
older fallback */
        #interrupt-cells = <3>;
        interrupt-controller;
        reg = <0xa01000 0x1000>,
              <0xa02000 0x100>;
    };

    clocks { /* fixed clocks */ };

    soc {
        compatible = "simple-bus";
        #address-cells = <1>;
        #size-cells = <1>;
        interrupt-parent = <&gpc>;
        ranges;          /* identity-mapped: child addresses == parent addresses */

        aips1: aips-bus@2000000 {
            compatible = "fsl,aips-bus", "simple-bus";
            #address-cells = <1>;
            #size-cells = <1>;
            reg = <0x2000000 0x100000>;
            ranges;

            uart1: serial@2020000 { /* ... */ status = "disabled"; };
            uart2: serial@21e8000 { /* ... */ };
```

```
            i2c1:  i2c@21a0000 { /* ... */ };
            /* ... ~30 more peripherals ... */
        };

        aips2: aips-bus@2100000 { /* ... */ };
        aips3: aips-bus@2200000 { /* ... */ };

        ocram: sram@900000 {
            compatible = "mmio-sram";
            reg = <0x00900000 0x20000>;
            clocks = <&clks IMX6UL_CLK_OCRAM>;
        };

        ddr_pmu: ddr-pmu@21b0000 {
            compatible = "fsl,imx6ull-ddr-pmu";
            reg = <0x21b0000 0x10000>;
            interrupts = <GIC_SPI 119 IRQ_TYPE_LEVEL_HIGH>;
        };
    };
};
```

Five nodes deserve special attention:

- **`cpus`**: under it, one or more `cpu@N` nodes. For i.MX6ULL (single-core), one. The kernel reads this to know how many cores to bring up.
- **`intc`**: the interrupt controller. The GIC v2 on Cortex-A7 inside the i.MX6ULL. `interrupt-controller;` (empty property) marks it as the IRQ source for any node that doesn't specify `interrupt-parent` otherwise.
- **`clocks`**: fixed-frequency oscillators (the 24 MHz XTAL, the 32 kHz RTC source).
- **`soc`**: a container for the on-SoC peripheral buses. `compatible = "simple-bus"` is the magic that tells the kernel to *automatically* recurse into child nodes and probe them (otherwise it would only probe nodes the parent's driver explicitly enumerated).
- **`aips1` / `aips2` / `aips3`**: the three AIPS bridges from Chapter 5. Each is `simple-bus` too, so the kernel descends into them.

27.8 How a peripheral comes alive

Walking the end-to-end binding for UART1 on the EVK:

1. **Boot.** U-Boot has loaded `imx6ull-14x14-evk.dtb` to `0x83000000` and called `bootz`.

> **MCU bridge:** Think of U-Boot like a much larger boot stub plus debug monitor: it initializes hardware, loads the next image, and gives you commands before Linux starts. **U-Boot:** the bootloader that initializes enough hardware to load and start the Linux kernel.

1. **`stext` reads `r2 = 0x83000000`** and stashes it.
2. **`setup_arch()` → `setup_machine_fdt(0x83000000)`** parses the DT blob.
3. **`unflatten_device_tree()`** builds the in-memory tree of `struct device_node`.
4. **`of_platform_default_populate()`** walks the tree, for each `compatible = "simple-bus"` parent, creates platform devices for each child.
5. For each created platform device, the kernel walks every registered platform_driver looking for `compatible` matches. The `imx-uart` driver matches via `fsl,imx21-uart`.
6. **`imx_uart_probe()` is called** with a `struct platform_device *pdev` for that DT node.
7. Inside probe: `clk_get(dev, "ipg")` (via DT's `clocks` + `clock-names`), `clk_prepare_enable(clk)`, `ioremap` of the `reg` range, `request_irq` of the IRQ from `interrupts`, `devm_pinctrl_get` to apply pinmux from `pinctrl-0`, register a TTY device, done.
8. **The driver is now bound to the hardware.** `/dev/ttymxc0` becomes available.

That nine-step flow happens for **every** peripheral. The variation is which `compatible` matches which driver. Every driver chapter in Part VI walks this same flow.

27.9 Writing a DT overlay

Suppose you wire a new I²C sensor (a Texas Instruments TMP102 thermometer) to I²C2 of your board, at address `0x48`. You don't want to recompile the kernel. You want to add a DT node at runtime via an overlay.

`tmp102-overlay.dtso`:

```
/dts-v1/;
/plugin/;

&i2c2 {
    status = "okay";

    tmp102@48 {
        compatible = "ti,tmp102";
        reg = <0x48>;
    };
};
```

`/plugin/;` marks this as an overlay (vs a standalone DT). The `&i2c2` references the I²C2 node in the base DT (defined in `imx6ull.dtsi`). The new `tmp102@48` becomes a child of `i2c2`.

Compile and apply (from U-Boot, see Ch 24E):

```
=> load mmc 0:1 0x84000000 tmp102-overlay.dtbo
=> fdt addr 0x83000000
=> fdt resize 4096
=> fdt apply 0x84000000
=> bootz 0x82000000 - 0x83000000
```

Or from a Linux user-space with ConfigFS (in newer kernels):

```
# mkdir /sys/kernel/config/device-tree/overlays/tmp102
# cat tmp102.dtbo > /sys/kernel/config/device-tree/overlays/tmp102/dtbo
```

After overlay application, the kernel re-walks the DT, finds the new `tmp102@48` node, looks for a driver with `compatible = "ti,tmp102"` (the upstream driver is `drivers/hwmon/tmp102.c`), probes it, and `/sys/class/hwmon/hwmon<N>/temp1_input` becomes readable.

In this case, an overlay changes the hardware description without rebuilding the kernel. The matching driver and required controller support still need to be available.

> **MCU bridge:** Think of the rootfs as the firmware image's file-backed runtime environment. On an MCU you link everything into flash. On Linux, programs and config live in this mounted tree. **rootfs:** root filesystem, the directory tree mounted at / that contains /bin, /etc, /dev, and libraries.

This is why DT exists.

27.10 OF API, accessing DT from driver code

When your driver's `probe()` is called, it gets a `struct platform_device *pdev`. From there it reads DT properties via the **OF API** (Open Firmware API, the historical name. Today everyone says "DT API" but the C symbols still start with `of_`):

```
struct device_node *np = pdev->dev.of_node;

const char *name;
of_property_read_string(np, "label", &name);

u32 freq;
of_property_read_u32(np, "clock-frequency", &freq);

u32 vals[4];
of_property_read_u32_array(np, "vals", vals, 4);
```

```
if (of_property_read_bool(np, "interrupt-controller"))
    /* this node is an IRQ controller */;

/* Look up a phandle (a &label reference). */
struct device_node *phy = of_parse_phandle(np, "phy-handle", 0);

/* Find a child by name. */
struct device_node *child = of_get_child_by_name(np, "led0");

/* Iterate all children. */
for_each_child_of_node(np, child) {
    /* ... */
}
```

About twenty `of_*` helpers cover every DT-shaped read. They are documented in `include/linux/of.h` and `Documentation/devicetree/usage-model.rst`. We will see most of them in Part VI.

27.11 The single biggest mental shift

For an MCU engineer used to:

```
void uart1_init(void) {
    CCM_CCGR5 |= (3 << 24);   // clock gate
    IOMUXC_SW_MUX_CTL_PAD_UART1_TX = 0;   // pinmux
    // ... 40 more register writes ...
}
```

...the Linux pattern is:

```
&uart1 {
    pinctrl-0 = <&pinctrl_uart1>;
    clocks = <&clks IMX6UL_CLK_UART1_IPG>, <&clks IMX6UL_CLK_UART1_SERIAL>;
    status = "okay";
};
```

And then *the driver*, written once for every board with this UART, reads those properties and does the equivalent register writes for you. The board engineer's job is to *describe* what's present, not to *do* the bring-up. The driver author's job is to handle every property correctly so any board can use the driver.

This is the Linux model. Once this clicks, Part VI is much easier.

27.12 Key Notes

- The SoC `.dtsi` describes controllers and shared resources. The board `.dts` describes connections, supplies and enabled devices, so a shared SoC does not make two boards' DTBs interchangeable.

- `compatible` identifies a hardware binding and connects a node to driver support. It is not driver code and cannot make an absent device respond.
- A property's meaning depends on its context. Parent bus cell counts define `reg` encoding, while clock, GPIO and interrupt providers define their own specifier cells.
- Phandles express resource relationships. Clock names and entries, pinctrl states and regulator supplies must agree with both the binding and the board wiring.
- DT compilation resolves includes and references into a blob. Kernel device creation and driver probe are later stages with their own dependencies and failures.
- Overlays modify a base hardware description. They require compatible base nodes and available driver support, and are not a substitute for checking an optional device's actual wiring.
- `/chosen` carries boot-time information such as command line and console selection. Those values are distinct from the persistent description of hardware resources.

27.13 Pitfalls

- **`compatible` typo.** Off-by-one-character compatible strings are the most common DT bug. Kernel parses the DT, sees no driver match, the device silently doesn't probe. Symptom: device file missing in `/dev/`. Nothing in `dmesg` about that device. Fix: `dtc -I dtb -O dts your.dtb | grep compatible` and verify every string is exact.
- **Missing `#address-cells` / `#size-cells` on a parent.** Kernel logs warning ("missing or invalid reg property") but might or might not fail. Always set these on any parent node that has child nodes with `reg`.
- **Hex without `0x`.** `reg = <2020000 4000>;` is decimal, not hex! Always use `0x` for register addresses: `reg = <0x2020000 0x4000>;`.
- **Forgotten `;` at end of property.** DTC error messages for missing `;` are sometimes misleading. Check the line above the error first.
- **Reference vs definition.** `uart1: serial@2020000 { ... };` *defines* the node. `&uart1 { ... };` *references and modifies* it. If you write `uart1 { ... };` (no label, just the bare name in a separate `/ { uart1 { ... } }`), the DTC creates a *new* node `uart1`, which is almost never what you want.
- **`clocks` order matters.** `clock-names` is positional. `clock-names = "ipg", "per"` means the *first* `clocks` entry is "ipg", the *second* is "per". Swap the order and the driver fails to find the right clock.
- **`status = "okay"` typo.** Some old templates use `"ok"` (without "ay"). The DT spec mandates `"okay"`. Older kernels accepted `"ok"` for backward compatibility, but modern `dtbs_check` rejects it as invalid against the schema.

27.14 Going deeper

- **`Documentation/devicetree/usage-model.rst`**: the canonical “how DT works” document.
- **`Documentation/devicetree/bindings/`**: every binding’s YAML schema. Where you go to look up “what properties does this kind of device want?”
- **`include/linux/of.h`**: every OF API function declared.
- **`drivers/of/`**: the kernel’s DT subsystem implementation.
- **`elinux.org/Device_Tree_Reference`**: community tutorial. Pleasant introduction.
- **The DeviceTree Specification (v0.4)** at `devicetree.org/specifications/`, the canonical spec. ~80 pages.
- **`scripts/dtc/`**: the DT compiler source. Worth a skim if you want to know exactly what DTC does.

> Next chapter: **Chapter 27A: DT bindings YAML and `dt_binding_check`.** Now that we know DTS, we look at the *contract* it has with drivers: machine-checkable JSON-schema descriptions of every binding, and the build target that validates them.

Chapter 27A: DT bindings YAML and dt_binding_check

> **DMA:** Direct Memory Access. Hardware moves data to or from memory without the CPU copying each byte. **MCU bridge:** Think of DMA like the MCU DMA controller you used for UART or SPI, but with cache coherency, scatter-gather descriptors, and kernel ownership rules added.

> **What:** the mainline kernel's machine-checkable description of every Device Tree binding, a JSON-Schema document (in YAML form) that says *exactly* which properties a node should have, of what type, with what constraints. Plus the `make dt_binding_check` / `make dtbs_check` targets that validate your DTS against them. **MCU bridge:** Think of Device Tree like a board-level hardware description table that replaces hard-coded #define LED_PORT GPIOA decisions. Unlike an MCU header, the kernel parses it at boot and matches it to drivers. **Device Tree:** a data file that describes board hardware to the Linux kernel instead of hard-coding it in C.
>
> **Why:** Since kernel v4.18 (mid-2018), every new binding *must* ship a YAML schema, and existing bindings are being migrated. Without a schema, your patch will not be accepted upstream. Without a schema check on your CI, your binding can silently drift between board variants and you'll only discover it when something breaks.
>
> **Focus:** the **schema as a contract**. A binding YAML is the source of truth for what a node *should* look like. The DTS files are checked against it. Drivers are documented by it. Read one binding YAML carefully and the rest follow the same pattern.

27A.1 Why bindings need schemas

For the first ~15 years of Device Tree's life, "bindings" lived as free-form `.txt` files in `Documentation/devicetree/bindings/`. A typical pre-2018 binding looked like:

```
* Freescale i.MX UART

Required properties:
- compatible: should be "fsl,<soc>-uart"
- reg: address and length of the register region
- interrupts: should contain the UART interrupt

Optional properties:
- fsl,dte-mode: indicates the UART is in DTE mode
```

Two problems:

1. **Not machine-checkable.** A DTS file with a typo (`compatible = "fsl,imx6ull-art";`, missing the `u`) or with a missing property silently passes compilation. Kernel just doesn't probe the device at runtime, and you go hunting.
2. **Inconsistent.** Two binding files written by two people, even for closely-related hardware, would describe things in subtly different ways. Some used `reg` for one purpose, some another. No way to enforce style.

Since 2018, every new binding ships as a **YAML file containing a JSON-Schema**. Schemas are validated automatically. DTS files are linted against them. CI rejects bindings that fail the lint. The `.txt` bindings are being migrated, with a deadline that keeps slipping but is real: any *new* binding without YAML is rejected.

27A.2 Anatomy of a binding YAML

Open `Documentation/devicetree/bindings/serial/fsl-imx-uart.yaml` (the i.MX UART binding). Lightly elided:

```
# SPDX-License-Identifier: (GPL-2.0-only OR BSD-2-Clause)
%YAML 1.2
---
$id: http://devicetree.org/schemas/serial/fsl-imx-uart.yaml#
$schema: http://devicetree.org/meta-schemas/core.yaml#

title: Freescale i.MX Universal Asynchronous Receiver/Transmitter (UART)

maintainers:
  - Fabio Estevam <festevam@gmail.com>

allOf:
  - $ref: serial.yaml#

properties:
  compatible:
    oneOf:
      - enum:
          - fsl,imx1-uart
          - fsl,imx21-uart
      - items:
          - enum:
              - fsl,imx25-uart
              - fsl,imx27-uart
              - fsl,imx31-uart
              - fsl,imx35-uart
              - fsl,imx50-uart
              - fsl,imx51-uart
              - fsl,imx53-uart
              - fsl,imx6q-uart
          - const: fsl,imx21-uart
      - items:
          - enum:
              - fsl,imx6sl-uart
```

```
              - fsl,imx6sll-uart
              - fsl,imx6ul-uart
          - const: fsl,imx6q-uart
          - const: fsl,imx21-uart

  reg:
    maxItems: 1

  interrupts:
    maxItems: 1

  clocks:
    items:
      - description: IPG clock for the UART
      - description: Per-module clock for the UART

  clock-names:
    items:
      - const: ipg
      - const: per

  dmas:
    items:
      - description: DMA channel for RX
      - description: DMA channel for TX

  dma-names:
    items:
      - const: rx
      - const: tx

  fsl,uart-has-rtscts:
    type: boolean
    description: |
      Indicates the UART has RTS and CTS lines, that are mostly required to do
      hardware flow control. Deprecated, use uart-has-rtscts instead.

  uart-has-rtscts: true

  fsl,dte-mode:
    type: boolean
    description: |
      Indicate the uart works in DTE mode. The uart works in DCE mode by default.

required:
  - compatible
  - reg
  - interrupts
  - clocks
  - clock-names

unevaluatedProperties: false

examples:
  - |
    #include <dt-bindings/clock/imx6sx-clock.h>
    aliases {
        serial0 = &uart1;
    };
```

```
    uart1: serial@2020000 {
        compatible = "fsl,imx6sx-uart", "fsl,imx21-uart";
        reg = <0x02020000 0x4000>;
        interrupts = <GIC_SPI 26 IRQ_TYPE_LEVEL_HIGH>;
        clocks = <&clks IMX6SX_CLK_UART_IPG>,
                 <&clks IMX6SX_CLK_UART_SERIAL>;
        clock-names = "ipg", "per";
        uart-has-rtscts;
        fsl,dte-mode;
    };
```

Section-by-section:

- **`$id`**: globally unique URL for this schema. Used by `$ref` to pull schemas in from other files.
- **`title`**: human-readable. Shows up in generated docs.
- **`maintainers`**: who signs off on changes.
- **`allOf: [$ref: serial.yaml#]`**: inherits constraints from a parent schema. Every UART binding includes the generic-serial schema, which adds e.g. `current-speed` and `rs485-*` properties.
- **`properties`**: the meat. Each property gets a sub-schema saying what kind of value it accepts.
  - `compatible.oneOf`: three legal forms: just `fsl,imx1-uart` or `fsl,imx21-uart`. A SoC string followed by `fsl,imx21-uart` as fallback. Or a three-string list ending in `fsl,imx21-uart`. Anything else is rejected.
  - `reg.maxItems: 1`: exactly one address-range, no more.
  - `clocks.items`: exactly two entries. The schema even documents what each represents.
  - `clock-names.items`: must be the literal strings “ipg” and “per”, in that order.
  - `uart-has-rtscts: true`: equivalent to “this is a boolean flag” (the value `true` after a name is the YAML shorthand for “this is a valid property with no additional constraints”).
- **`required`**: properties that MUST be present.
- **`unevaluatedProperties: false`**: *no other properties* are allowed. This catches typos. If you wrote `clock-name` (singular), the schema rejects it as an unknown property.
- **`examples`**: a working DT fragment that must validate against the schema. Doubly useful: it documents usage *and* it’s lint-checked as part of `dtbs_check`.

This one file replaces ~30 lines of English prose, and it can be checked by tools.

27A.3 Running the checks

Two `make` targets:

```
$ make ARCH=arm CROSS_COMPILE=arm-none-linux-gnueabihf- dt_binding_check
$ make ARCH=arm CROSS_COMPILE=arm-none-linux-gnueabihf- dtbs_check
```

What each does:

- **`dt_binding_check`** validates every binding YAML file *against the meta-schema*. Catches errors *in the bindings themselves*: malformed YAML, references to non-existent parent schemas, missing required meta-fields. Run this when you write or edit a binding.
- **`dtbs_check`** compiles every DTS for the current arch, then validates each *compiled DTB* against the matching binding schema. Catches errors *in the DT source*: wrong property types, missing required properties, extra unknown properties. Run this when you edit a DTS.

Prerequisites:

```
$ pip install --user --upgrade dtschema yamllint
```

`dtschema` is the validator that does the actual matching. `yamllint` catches YAML formatting issues.

A typical `dtbs_check` run on the i.MX tree produces ~100 warnings as of v6.6, many existing DTS files have minor issues that haven't been cleaned up. New code should add *zero* new warnings. The upstream maintainers will require that.

27A.4 Writing your first binding

Consider a GPIO-driven LED as a small schema case study. The fictional "my-pa-led" binding below illustrates constraints, while the existing `gpio-leds` binding is the appropriate interface for a normal GPIO LED.

> **MCU bridge:** Think of Linux GPIO like the same pin set/reset block you used on STM32, but accessed through a kernel subsystem that owns numbering, direction, interrupts, and user-space exposure. **GPIO:** General-Purpose Input/Output, a pin controlled as a digital input, output, or interrupt source.

`Documentation/devicetree/bindings/leds/myorg,pa-led.yaml`:

```
# SPDX-License-Identifier: (GPL-2.0-only OR BSD-2-Clause)
%YAML 1.2
---
```

```
$id: http://devicetree.org/schemas/leds/myorg,pa-led.yaml#
$schema: http://devicetree.org/meta-schemas/core.yaml#

title: My-Org Point Atom LED

maintainers:
  - Your Name <you@example.org>

description: |
  A single GPIO-driven LED on the Point Atom MINI / ALPHA boards.
  Active-low (cathode connected to the GPIO; anode through resistor to 3V3).

properties:
  compatible:
    const: myorg,pa-led

  gpios:
    maxItems: 1
    description: GPIO connected to the LED's cathode

  default-state:
    enum: [on, off, keep]
    default: off

required:
  - compatible
  - gpios

additionalProperties: false

examples:
  - |
    #include <dt-bindings/gpio/gpio.h>

    led0 {
        compatible = "myorg,pa-led";
        gpios = <&gpio1 3 GPIO_ACTIVE_LOW>;
        default-state = "off";
    };
```

The corresponding scoped binding check is:

```
$ make ARCH=arm CROSS_COMPILE=arm-none-linux-gnueabihf- dt_binding_check \
    DT_SCHEMA_FILES=leds/myorg,pa-led.yaml
```

If the schema is well-formed, the example inside it compiles cleanly and the binding-check passes. If you have a typo, you'll get a precise error pointing at the line.

Then, to add the matching DT node to a board:

```
/ {
    led0 {
        compatible = "myorg,pa-led";
        gpios = <&gpio1 3 GPIO_ACTIVE_LOW>;
        default-state = "off";
```

```
    };
};
```

The `make ARCH=arm dtbs_check` target validates the board node against the corresponding schema.

27A.5 Common schema patterns

A short library of patterns you'll re-use:

Property whose value is one of a fixed set

```
my-mode:
  enum: [polled, irq, dma]
  default: polled
```

Property whose value is a number in a range

```
my-freq-hz:
  $ref: /schemas/types.yaml#/definitions/uint32
  minimum: 1000000
  maximum: 50000000
```

Property whose value is a string matching a pattern

```
label:
  pattern: '^[a-z0-9-]+$'
```

Property whose value depends on `compatible`

```
allOf:
  - if:
      properties:
        compatible:
          contains:
            const: myorg,pa-board-rev-b
    then:
      required:
        - my-rev-b-only-property
```

A pair of related arrays whose lengths must match

```
foo:
  $ref: /schemas/types.yaml#/definitions/uint32-array
  minItems: 1
  maxItems: 8
```

```
foo-names:
  minItems: 1
  maxItems: 8

# Then in the example, foo and foo-names have the same number of items.
```

Most real bindings use 5-10 patterns from this library. Reading existing bindings in `Documentation/devicetree/bindings/` is the fastest way to learn them.

27A.6 Inheriting from base schemas

A binding rarely starts from scratch. Almost every binding `allOf`:-includes one or more parents:

- **`serial.yaml`** for UARTs
- **`i2c-controller.yaml`** for I²C buses
- **`spi-controller.yaml`** for SPI buses
- **`leds.yaml`** for LED-class devices
- **`hwmon.yaml`** for hardware monitoring

These parents define standard properties common to the *class* (e.g., `current-speed` for serial, `clock-frequency` for I²C buses). Inheriting from them means your binding picks up that vocabulary automatically and is validated consistently with everyone else's.

27A.7 Key Notes

- A binding is the contract between hardware description and driver interpretation. YAML expresses that contract using JSON Schema constraints rather than replacing it with implementation details.
- `dt_binding_check` checks the binding and its examples. `dtbs_check` checks compiled board descriptions against applicable bindings, so the two targets address different error sources.
- Required properties, permitted values and ordered resource names make assumptions explicit. Unknown-property rejection can expose a typo that ordinary DTS compilation accepts.
- Shared schemas carry class-wide vocabulary. Closing a composed schema with `unevaluatedProperties` accounts for inherited properties rather than treating them all as unknown.
- A schema example illustrates a node's interface, not a new driver. The fictional LED binding here does not replace the maintained `gpio-leds` implementation.

- Validation can reject malformed descriptions without detecting every wrong address, pin connection or voltage assumption. Schema success is not physical-board qualification.

27A.8 Pitfalls

- **Schema-validation passes but DT still doesn’t work at runtime.** The schema only catches *syntactic* errors, wrong types, wrong arity, missing required props. Semantic errors (a `reg` value pointing at the wrong physical address) pass schema but fail at boot. The schema catches some bugs but not all. You still need to test on real hardware.
- **Forgetting to install `dtschema`.** Symptom: `dt_binding_check` reports zero warnings on a tree that clearly has issues. Means the validator silently isn’t running. Verify: `pip show dtschema`.
- **Schema example doesn’t validate.** Common when you write a binding without testing the example. Run `dt_binding_check` against your own binding first. Only then propose it upstream.
- **`additionalProperties: false` too strict.** If you forget that the inherited base schema allows certain extra properties, the strict-mode rejection of “unknown” properties will reject legal usage. Use `unevaluatedProperties: false` (which respects `allOf` inheritance) instead.
- **`maxItems` vs `items`.** `maxItems: 1` means up to one entry of any type. `items: [- description: foo]` means exactly one entry, described as foo. These are different, use the second form when you want the description to appear in the docs.

27A.9 Going deeper

- **`Documentation/devicetree/writing-schema.rst`**: the canonical tutorial.
- **`Documentation/devicetree/bindings/example-schema.yaml`**: a fully-annotated example schema. Read it after this chapter.
- **`dtschema` source**: `github.com/devicetree-org/dt-schema`. Read for ground truth on what’s actually validated.
- **JSON Schema spec** at `json-schema.org/specification`. DT bindings use JSON Schema vocabulary. This is the underlying spec.
- **`grep -r "unevaluatedProperties" Documentation/devicetree/bindings/`**: read a few real-world bindings end-to-end before writing your own.

> Next chapter: **Chapter 27B: Device Tree for a custom i.MX6ULL board.** With DTS grammar and bindings understood, we now turn a board schematic into a Linux board `.dts`.

Chapter 27B: Device Tree for a custom i.MX6ULL board

> **What:** a custom-board Device Tree case study covering UART console, SD/eMMC, Ethernet, I2C, SPI, LEDs, keys, regulators, aliases and `/chosen`.
>
> **Why:** Chapter 27 teaches the grammar. This chapter teaches the job: turn a schematic into a `.dts` file that the kernel can boot.
>
> **Result:** a view of how schematic connections become kernel resource relationships, and why compilation, schema checks and peripheral operation establish different things.
>
> **Focus:** do not start by copying the whole EVK and hoping. Start with a small bootable board description, then add one peripheral at a time.

The case uses the image-loading interface described in Chapter 26. Its fragments are illustrative, not a qualified description of the fitted MINI board, and a working port is not a reading prerequisite.

27B.1 The board DTS is not the SoC DTS

The i.MX6ULL SoC file already exists:

```
arch/arm/boot/dts/nxp/imx/imx6ull.dtsi
```

It describes hardware inside the chip:

- CPU.
- GIC.
- UART controllers.
- I2C controllers.
- SPI controllers.
- USDHC controllers.
- GPIO banks.
- CCM clocks.
- IOMUXC pin controller.

- OCRAM.

Your board file describes what is connected outside the chip and which SoC blocks are actually used:

```
arch/arm/boot/dts/nxp/imx/imx6ull-pa-mini.dts
```

Board DTS files answer questions like:

- Which UART is the console?
- Which pins are routed to the SD slot?
- Which regulator powers the SD card?
- Which PHY is connected to FEC Ethernet?
- Which I2C address is the temperature sensor?
- Which GPIO drives the user LED?

27B.2 Files changed

File	What changes
`arch/arm/boot/dts/nxp/imx/imx6ull-pa-mini.dts`	New board description.
`arch/arm/boot/dts/nxp/imx/Makefile`	Add `imx6ull-pa-mini.dtb` to the build.
U-Boot environment	Set `fdtfile=imx6ull-pa-mini.dtb` or load it directly.

No kernel driver is written in this chapter.

27B.3 Start with the smallest bootable DTS

Create `arch/arm/boot/dts/nxp/imx/imx6ull-pa-mini.dts`:

```
// SPDX-License-Identifier: (GPL-2.0+ OR MIT)
/dts-v1/;

#include "imx6ull.dtsi"

/ {
    model = "Point Atom MINI i.MX6ULL";
    compatible = "point-atom,imx6ull-mini", "fsl,imx6ull";
```

```
    chosen {
        stdout-path = &uart1;
    };

    memory@80000000 {
        device_type = "memory";
        reg = <0x80000000 0x20000000>; /* 512 MiB */
    };
};

&uart1 {
    pinctrl-names = "default";
    pinctrl-0 = <&pinctrl_uart1>;
    status = "okay";
};

&iomuxc {
    pinctrl_uart1: uart1grp {
        fsl,pins = <
            MX6UL_PAD_UART1_TX_DATA__UART1_DCE_TX 0x1b0b1
            MX6UL_PAD_UART1_RX_DATA__UART1_DCE_RX 0x1b0b1
        >;
    };
};
```

This is enough for:

- Kernel to identify the machine.
- Kernel to know RAM size.
- Kernel to find the console UART.
- Kernel to print early boot logs.

It is not enough for SD rootfs, Ethernet, GPIO LEDs, or I2C. Add those later.

27B.4 Add it to the kernel build

Edit `arch/arm/boot/dts/nxp/imx/Makefile`:

```
dtb-$(CONFIG_SOC_IMX6ULL) += \
    imx6ull-pa-mini.dtb
```

Keep the list sorted near other `imx6ull-*` boards if the file already has a long block.

Build:

```
$ make dtbs
$ ls arch/arm/boot/dts/nxp/imx/imx6ull-pa-mini.dtb
```

Boot it:

```
=> tftp 0x82000000 zImage
=> tftp 0x83000000 imx6ull-pa-mini.dtb
=> setenv bootargs 'console=ttymxc0,115200 earlycon root=/dev/mmcblk0p2 rw rootwait'
=> bootz 0x82000000 - 0x83000000
```

The first proof is the model line:

```
OF: fdt: Machine model: Point Atom MINI i.MX6ULL
```

If you see that line, the kernel consumed your DTB.

27B.5 Add fixed regulators

Most boards have always-on rails or GPIO-controlled rails. Describe them explicitly. Drivers use these nodes to know when power is available.

Example SD-card 3.3 V rail:

```
/ {
    reg_sd1_vmmc: regulator-sd1-vmmc {
        compatible = "regulator-fixed";
        regulator-name = "VSD_3V3";
        regulator-min-microvolt = <3300000>;
        regulator-max-microvolt = <3300000>;
        gpio = <&gpio1 9 GPIO_ACTIVE_HIGH>;
        enable-active-high;
    };
};
```

If the rail is truly always on and has no GPIO:

```
/ {
    reg_3v3: regulator-3v3 {
        compatible = "regulator-fixed";
        regulator-name = "3V3";
        regulator-min-microvolt = <3300000>;
        regulator-max-microvolt = <3300000>;
        regulator-always-on;
    };
};
```

Do not invent regulators just to silence warnings. Each regulator node should correspond to a real rail in the schematic.

27B.6 Add SD or eMMC

Example SD slot on USDHC1:

```
&usdhc1 {
    pinctrl-names = "default", "state_100mhz", "state_200mhz";
    pinctrl-0 = <&pinctrl_usdhc1>;
    pinctrl-1 = <&pinctrl_usdhc1_100mhz>;
    pinctrl-2 = <&pinctrl_usdhc1_200mhz>;
    cd-gpios = <&gpio1 19 GPIO_ACTIVE_LOW>;
    keep-power-in-suspend;
    vmmc-supply = <&reg_sd1_vmmc>;
    bus-width = <4>;
    status = "okay";
};
```

Pin group:

```
&iomuxc {
    pinctrl_usdhc1: usdhc1grp {
        fsl,pins = <
            MX6UL_PAD_SD1_CMD__USDHC1_CMD     0x17059
            MX6UL_PAD_SD1_CLK__USDHC1_CLK     0x10071
            MX6UL_PAD_SD1_DATA0__USDHC1_DATA0 0x17059
            MX6UL_PAD_SD1_DATA1__USDHC1_DATA1 0x17059
            MX6UL_PAD_SD1_DATA2__USDHC1_DATA2 0x17059
            MX6UL_PAD_SD1_DATA3__USDHC1_DATA3 0x17059
        >;
    };
};
```

Use the actual pins from your schematic. The pad names must match `imx6ul-pinfunc.h`.

Boot proof:

```
mmc0: new high speed SDHC card at address 0001
mmcblk0: mmc0:0001 SD16G 14.8 GiB
```

If you do not see `mmcblk0`, check pinctrl, card-detect GPIO, regulator, and `CONFIG_MMC_SDHCI_ESDHC_IMX`.

27B.7 Add Ethernet

Ethernet needs more than a MAC node. You need MAC pins, MDIO, PHY address, reset GPIO, clocking, and sometimes a regulator.

Example:

```
&fec1 {
    pinctrl-names = "default";
    pinctrl-0 = <&pinctrl_enet1>;
    phy-mode = "rmii";
    phy-handle = <&ethphy0>;
    phy-supply = <&reg_3v3>;
    status = "okay";
```

```
    mdio {
        #address-cells = <1>;
        #size-cells = <0>;

        ethphy0: ethernet-phy@0 {
            compatible = "ethernet-phy-ieee802.3-c22";
            reg = <0>;
            reset-gpios = <&gpio5 7 GPIO_ACTIVE_LOW>;
            reset-assert-us = <10000>;
            reset-deassert-us = <80000>;
        };
    };
};
```

Boot proof:

```
fec 2188000.ethernet eth0: registered PHC device 0
libphy: fec_enet_mii_bus: probed
```

Runtime proof:

```
target# ip link show eth0
target# udhcpc -i eth0
```

If the PHY is not found, check:

- `reg = <N>` matches the PHY strap address.
- MDC/MDIO pins are correct.
- PHY reset is deasserted.
- 50 MHz RMII reference clock is present.

27B.8 Add I2C devices

Enable the controller:

```
&i2c1 {
    clock-frequency = <100000>;
    pinctrl-names = "default";
    pinctrl-0 = <&pinctrl_i2c1>;
    status = "okay";

    temp@48 {
        compatible = "ti,tmp102";
        reg = <0x48>;
    };
};
```

Pin group:

```
&iomuxc {
    pinctrl_i2c1: i2c1grp {
        fsl,pins = <
            MX6UL_PAD_UART4_TX_DATA__I2C1_SCL 0x4001b8b0
            MX6UL_PAD_UART4_RX_DATA__I2C1_SDA 0x4001b8b0
        >;
    };
};
```

The `0x40000000` bit selects open-drain behavior in the i.MX pin config. I2C also needs pull-ups on the board.

Boot proof:

```
tmp102 0-0048: initialized
```

Runtime proof:

```
target# cat /sys/class/hwmon/hwmon*/temp1_input
```

27B.9 Add SPI devices

Enable ECSPI:

```
&ecspi1 {
    pinctrl-names = "default";
    pinctrl-0 = <&pinctrl_ecspi1>;
    cs-gpios = <&gpio4 26 GPIO_ACTIVE_LOW>;
    status = "okay";

    spidev@0 {
        compatible = "rohm,dh2228fv";
        reg = <0>;
        spi-max-frequency = <10000000>;
    };
};
```

For learning, `spidev` is useful. For production, bind the real device to a real driver. Do not ship product logic that depends on a fake compatible string.

Runtime proof:

```
target# ls /dev/spidev*
```

27B.10 Add LEDs and keys

LED:

```
/ {
    leds {
        compatible = "gpio-leds";

        user_led {
            label = "user";
            gpios = <&gpio1 3 GPIO_ACTIVE_LOW>;
            default-state = "off";
        };
    };
};
```

Button:

```
/ {
    gpio-keys {
        compatible = "gpio-keys";

        recovery {
            label = "recovery";
            gpios = <&gpio1 5 GPIO_ACTIVE_LOW>;
            linux,code = <KEY_RESTART>;
            wakeup-source;
        };
    };
};
```

Add the include for key constants:

```
#include <dt-bindings/input/input.h>
```

Runtime proof:

```
target# ls /sys/class/leds
target# echo 1 > /sys/class/leds/user/brightness
target# cat /proc/bus/input/devices
```

27B.11 Add aliases and chosen

Make Linux names stable:

```
/ {
    aliases {
        serial0 = &uart1;
        ethernet0 = &fec1;
        mmc0 = &usdhc1;
        i2c0 = &i2c1;
        spi0 = &ecspi1;
    };

    chosen {
```

```
        stdout-path = &uart1;
    };
};
```

Aliases affect numbering. If you want `/dev/mmcblk0` to stay predictable, use aliases.

27B.12 Build and validate

Build only the DTB:

```
$ make dtbs
```

Run schema checks:

```
$ make dtbs_check DT_SCHEMA_FILES=imx
```

Reverse-compile for inspection:

```
$ dtc -I dtb -O dts \
      arch/arm/boot/dts/nxp/imx/imx6ull-pa-mini.dtb \
      > /tmp/imx6ull-pa-mini.expanded.dts
```

Search the expanded file:

```
$ grep -n 'Point Atom MINI' /tmp/imx6ull-pa-mini.expanded.dts
$ grep -n 'serial@2020000' /tmp/imx6ull-pa-mini.expanded.dts
$ grep -n 'ethernet-phy' /tmp/imx6ull-pa-mini.expanded.dts
```

The expanded DTS is what the kernel effectively sees after includes.

27B.13 Bring-up order

The dependency order for a staged board description is:

1. Memory and UART.
2. SD/eMMC.
3. Rootfs boot.
4. Ethernet.
5. LEDs and keys.
6. I2C.
7. SPI.

8. Displays, audio, sensors, and optional devices.

A log from each known state separates a new peripheral dependency from a regression in an earlier one.

27B.14 Key Notes

- A board DTS owns external topology, not the SoC register description or driver implementation. Included SoC nodes provide the base that board-specific references refine.
- Memory and console form an early observation path. Storage and filesystem support are additional dependencies before a normal rootfs can become usable.
- Each enabled controller needs more than `status = "okay"`. Pinmux, supplies, bus width, reset signals and attached-device addresses must describe the same physical circuit.
- Ethernet spans MAC, MDIO, PHY, reset and reference clock. An interface name or model string alone does not establish a usable physical link or correct MINI wiring.
- LEDs, keys and sensors use existing subsystem bindings where possible. A fake SPI compatible string introduces a description mismatch rather than a production device interface.
- Build registration selects which board blob is produced, while bootloader selection decides which blob Linux receives. Either can be correct while the other is wrong.
- The fragments describe architectural patterns, not verified carrier mappings. A fitted board requires schematic-to-pad tracing and runtime evidence beyond compilation and schema checks.

27B.15 Pitfalls

- **Copying the EVK DTS blindly.** It may describe pins and regulators your board does not have.
- **Enabling a controller without pinctrl.** The driver probes, but pins stay muxed to something else.
- **Wrong GPIO polarity.** `GPIO_ACTIVE_LOW` vs `GPIO_ACTIVE_HIGH` changes behavior completely.
- **Wrong PHY address.** Ethernet will probe MDIO and find nothing.
- **Forgetting `vmmc-supply`.** MMC may work sometimes and fail under load.

- **Using DT to fix driver bugs.** DT describes hardware. Do not encode software policy unless the binding explicitly asks for it.

27B.16 Going deeper

- `Documentation/devicetree/bindings/`, for each device class.
- `arch/arm/boot/dts/nxp/imx/`, for i.MX board examples.
- `scripts/dtc/`, for the compiler.
- `drivers/of/`, for how the kernel creates devices from DT.

Previous: Chapter 27A: DT bindings YAML and `dt_binding_check`

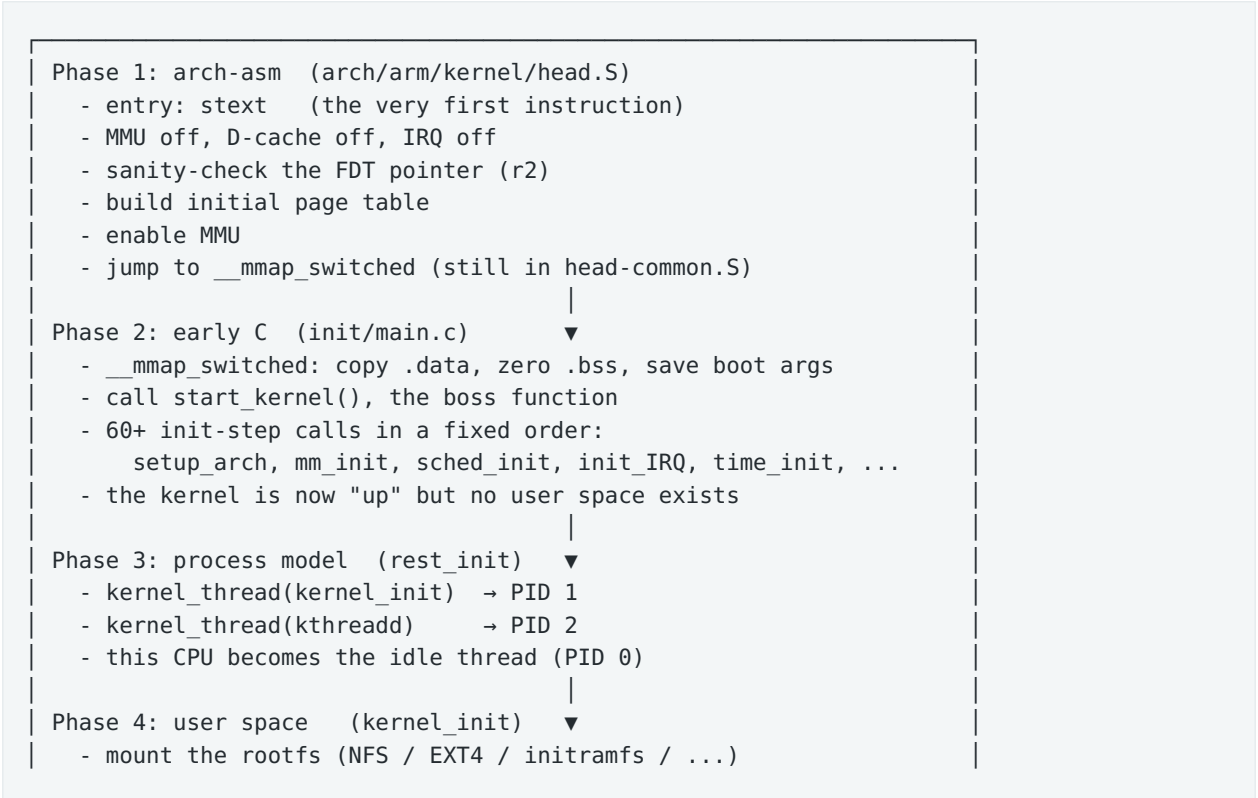
Next: Chapter 28: Kernel startup, traced

Chapter 28: Kernel startup, traced

> **IRQ:** interrupt request, the signal path that tells the CPU or interrupt controller that hardware needs service. **MCU bridge:** Think of an IRQ like an EXTI/NVIC interrupt path, except Linux splits the hard interrupt from deferred work and must share lines across drivers. **NFS:** Network File System, which lets the target mount a host directory over Ethernet during development.

> **What:** the kernel path from `stext` to executing `/sbin/init`, with source locations connecting the boot log from Chapter 26 to subsystem initialization.
>
> **Why:** The boot path is long but readable. Each line you trace becomes one less thing that surprises you when something breaks. By the time you have walked `stext → __mmap_switched → start_kernel → rest_init → kernel_init` once, you can debug "why is my system not booting?" with confidence.
>
> **Focus:** the **four phases** of kernel startup: (1) architecture-specific assembly that runs *before* virtual memory, (2) early C in `start_kernel()` that brings up subsystems in a fixed order, (3) `rest_init()` which forks PID 1 and PID 2, (4) `kernel_init` which exec's user-space. Each phase has a clean handoff to the next.

28.1 The four phases



```
│   - run /sbin/init  (or /etc/init / /bin/init / /bin/sh fallback)   │
│   - if exec succeeds, kernel_init's job is done                      │
└──────────────────────────────────────────────────────────────────────┘
```

We walk each phase in turn. Source-file paths are relative to the kernel source root. Line numbers reference v6.6. ±5 lines on other versions.

28.2 Phase 1, `stext` (arch/arm/kernel/head.S)

`stext` is the entry point, the very first instruction the kernel executes. The linker script `arch/arm/kernel/vmlinux.lds.S` declares it as the entry:

```
ENTRY(stext)
```

`stext` is defined in `arch/arm/kernel/head.S`. Before the kernel takes control, U-Boot has prepared:

> **MCU bridge:** Think of U-Boot like a much larger boot stub plus debug monitor: it initializes hardware, loads the next image, and gives you commands before Linux starts. **U-Boot:** the bootloader that initializes enough hardware to load and start the Linux kernel.

```
MMU       = off
D-cache   = off
I-cache   = don't care (kernel will flush)
r0        = 0  (or boot-magic; ignored on DT systems)
r1        = machine number  (legacy ATAGS; ignored on DT systems)
r2        = physical address of the DTB
```

This contract is documented in `Documentation/arch/arm/booting.rst`, the page to consult if you ever doubt what register holds what.

`stext` walked, with our annotations:

```
ENTRY(stext)
    safe_svcmode_maskall r9    @ ensure SVC mode, IRQ+FIQ masked

    mrc p15, 0, r9, c0, c0      @ read processor MIDR
    bl  __lookup_processor_type @ find proc_info for this MIDR in __proc_info_begin..end
    movs r10, r5                @ r5 = procinfo struct ptr (or 0 = not supported)
    beq __error_p               @ if not supported, hang in __error_p

    bl  __vet_atags             @ sanity-check the FDT/ATAGS pointer in r2

    bl  __create_page_tables    @ build a minimal page table in OCRAM/early DRAM

    /* compute the address to jump to AFTER the MMU comes up */
    ldr r13, =__mmap_switched
    adr lr, 1f                  @ set up return for the __cpu_setup call
```

```
    mov r8, r4
    ldr r12, [r10, #PROCINFO_INITFUNC]
    add r12, r12, r10
    ret r12                     @ call CPU-specific __cpu_setup (arch/arm/mm/proc-v7.S)

1:  b   __enable_mmu            @ enable MMU; flow continues at r13 (=__mmap_switched)
ENDPROC(stext)
```

Four points to keep in mind:

- **`__lookup_processor_type`** walks a linker-supplied array (`__proc_info_begin..__proc_info_end`) of `struct proc_info_list`. Each entry says "for MIDR mask X = value Y, this is your `cpu_setup`, `cpu_cache_fns`, etc." The Cortex-A7 entry lives in `arch/arm/mm/proc-v7.S`. If your CPU isn't recognised, the kernel hangs in `__error_p` (you see no output because UART isn't initialised yet, Chapter 26 §26.5 covers this failure mode).
- **`__vet_atags`** does a quick byte-pattern check on the data at `r2`. If it looks like a DTB (magic bytes `0xD00DFEED`) or ATAGS, it's accepted. Otherwise the address is zeroed and the kernel will later boot with no DT (almost certainly panicking).
- **`__create_page_tables`** builds a flat identity-mapped 1-MiB-section page table just big enough to cover the kernel image + early reservations. Real page tables come later in `paging_init()`.
- **`__enable_mmu`** sets `SCTLR.M=1`. The next instruction it executes is via virtual addresses. The jump-via-`r13` lands at `__mmap_switched`, which lives at a virtual address in the kernel's mapped region.

28.3 Phase 2 begins, `__mmap_switched` (arch/arm/kernel/head-common.S)

The first C-callable function after the MMU comes up. It does the C runtime setup:

> **MCU bridge:** Think of the MMU as a hardware address translator in front of every load/store. Cortex-M usually runs physical addresses directly. Linux relies on virtual addresses and page permissions. **MMU:** Memory Management Unit, hardware that translates virtual addresses to physical addresses and enforces permissions.

```
__mmap_switched:
    adr r3, __mmap_switched_data

    /* copy .data from its initial location to its run location (if any) */
    ldmia r3!, {r4, r5, r6, r7}
1:  cmp r5, r6
    ldrne fp, [r4], #4
    strne fp, [r5], #4
    bne 1b
```

```
    /* zero .bss */
    mov fp, #0
1:  cmp r6, r7
    strcc fp, [r6], #4
    bcc 1b

    /* save processor ID, machine number, atags/dtb pointer for early kernel code */
    ldmia r3, {r4, r5, r6, r7, sp}
    str r9, [r4]   @ processor ID
    str r1, [r5]   @ machine number
    str r2, [r6]   @ atags/dtb pointer

    b start_kernel  @ never returns
ENDPROC(__mmap_switched)
```

The two stash-saves at the end are why `r9`, `r1`, `r2` were preserved through Phase 1: they get parked at well-known kernel addresses so the C code can find them.

After `b start_kernel`, we are running C with a stack, BSS zeroed, and the boot arguments tucked away. **This is the boundary between assembly and C.** Everything from here is in `init/main.c` or files it calls.

28.4 Phase 2 main, `start_kernel()` (init/main.c)

`start_kernel()` is ~200 lines of sequential setup calls. Each call brings one subsystem from “uninitialised” to “minimally functional.” The order matters, many calls depend on earlier ones. Read it from the top:

```
asmlinkage __visible void __init start_kernel(void)
{
    char *command_line;
    char *after_dashes;

    set_task_stack_end_magic(&init_task);
    smp_setup_processor_id();
    debug_objects_early_init();
    init_vmlinux_build_id();
    cgroup_init_early();

    local_irq_disable();
    early_boot_irqs_disabled = true;

    boot_cpu_init();
    page_address_init();
    pr_notice("%s", linux_banner);          /* the famous "Linux version ..." */
    early_security_init();
    setup_arch(&command_line);              /* the heavy lifter — see below */
    setup_boot_config();
    setup_command_line(command_line);
    setup_nr_cpu_ids();
    setup_per_cpu_areas();
    smp_prepare_boot_cpu();
    boot_cpu_hotplug_init();
```

```
    build_all_zonelists(NULL);
    page_alloc_init();
    pr_notice("Kernel command line: %s\n", saved_command_line);

    /* parameter parsing */
    after_dashes = parse_args("Booting kernel",
                              static_command_line, __start___param,
                              __stop___param - __start___param,
                              -1, -1, NULL, &unknown_bootoption);

    setup_log_buf(0);
    vfs_caches_init_early();
    sort_main_extable();
    trap_init();
    mm_core_init();                         /* the memory manager */
    poking_init();
    ftrace_init();
    early_trace_init();
    sched_init();                           /* the scheduler */
    radix_tree_init();
    maple_tree_init();
    housekeeping_init();
    workqueue_init_early();
    rcu_init();
    trace_init();
    context_tracking_init();
    early_irq_init();
    init_IRQ();                             /* the interrupt subsystem */
    tick_init();
    rcu_init_nohz();
    init_timers();
    srcu_init();
    hrtimers_init();
    softirq_init();
    timekeeping_init();
    kfence_init();
    time_init();

    /* enable interrupts — kernel can now respond to IRQs */
    local_irq_enable();
    early_boot_irqs_disabled = false;

    /* ... many more init calls ... */

    console_init();                         /* now printk reaches the UART for real */
    ...
    rest_init();                            /* hand off to phase 3 */
}
```

A few calls earn their own attention.

`setup_arch(&command_line)`, `arch/arm/kernel/setup.c`

The biggest single call in `start_kernel()` on ARM. The main steps are:

1. **`setup_machine_fdt(__atags_pointer)`**: Parses the DT blob (passed in `r2` and saved by Phase 1). Calls `unflatten_device_tree()` which converts the flat DTB to the in-memory tree of `struct device_node`. Reads `/chosen/bootargs` and stores it in `boot_command_line`.
2. **`parse_early_param()`**: Handles a small set of cmdline tokens that need to be processed before most subsystems exist (`earlycon=`, `debug=`, `nokaslr`, `mem=`).
3. **`paging_init()`**: Builds the *real* page tables now that we know how much DRAM exists (from the DT) and where the kernel needs to map peripherals.
4. **`request_standard_resources()`**: Populates `/proc/iomem` with the kernel-code / kernel-data / DRAM regions.
5. **`smp_init_cpus()`**: Initializes per-CPU data structures.

After `setup_arch` returns, *the kernel knows what hardware it's on*, and the command line is parsed.

`mm_core_init()`, `mm/mm_init.c`

Brings up the memory subsystem: the page allocator (`buddy`), the slab allocator (`SLUB`), the vmalloc address space. After this, `kmalloc()` works.

`sched_init()`, `kernel/sched/core.c`

Initializes the scheduler's data structures and creates the boot CPU's runqueue. After this, `schedule()` works, but there's still only one task (the boot thread).

`init_IRQ()`, calls into `arch/arm/kernel/irq.c` → `irqchip_init()`

Walks the DT looking for nodes with `compatible = "arm,cortex-a7-gic"` (or whichever interrupt controller the SoC uses), and probes the GIC driver. After this, IRQs from devices can be registered with `request_irq()`.

> **MCU bridge:** Think of the GIC like the Cortex-M NVIC scaled up for Cortex-A: it routes peripheral interrupts to CPU cores and has separate distributor and CPU-interface blocks. **GIC:** ARM's Generic Interrupt Controller, the Cortex-A interrupt router roughly analogous to NVIC on Cortex-M.

`time_init()`, `drivers/clocksource/`

Walks the DT for clocksource and clockevent providers (the generic ARM timer, or i.MX GPT, depending on configuration). Establishes the kernel's notion of "the current time" and "how to schedule future events." After this, `jiffies` advances, `udelay()` works, `hrtimer`s work.

`console_init()`, `drivers/tty/`

Now binds the *real* console driver to the UART. Until this point, all `printk` output went to one of two places. Either it sat in the `printk` ring buffer for `dmesg` to read later, or it was pushed to the UART by `earlycon` if the bootloader configured that. After `console_init()`, every later `printk` reaches the UART in real time.

Roughly 30 more init calls

`vfs_caches_init`, `proc_root_init`, `cgroup_init`, `taskstats_init_early`, `cpuset_init`, `kthread_init`, `late_time_init`, … each brings up one subsystem. Don't memorise the order. Do know that it's a fixed sequence that you can trace in source.

At the end, `start_kernel()` calls `rest_init()` and never returns.

28.5 Phase 3, `rest_init()` (init/main.c)

```
noinline void __ref rest_init(void)
{
    struct task_struct *tsk;
    int pid;

    rcu_scheduler_starting();

    /*
     * We need to spawn init first so that it obtains pid 1, however
     * the init task will end up wanting to create kthreads, which,
     * if we schedule it before we create kthreadd, will OOPS.
     */
    pid = user_mode_thread(kernel_init, NULL, CLONE_FS);
    rcu_read_lock();
    tsk = find_task_by_pid_ns(pid, &init_pid_ns);
    tsk->flags |= PF_NO_SETAFFINITY;
    set_cpus_allowed_ptr(tsk, cpumask_of(smp_processor_id()));
    rcu_read_unlock();

    numa_default_policy();
    pid = kernel_thread(kthreadd, NULL, NULL, CLONE_FS | CLONE_FILES);
    rcu_read_lock();
    kthreadd_task = find_task_by_pid_ns(pid, &init_pid_ns);
    rcu_read_unlock();

    system_state = SYSTEM_SCHEDULING;
    complete(&kthreadd_done);
```

```
    /* Call into cpu_idle with preempt disabled */
    schedule_preempt_disabled();
    cpu_startup_entry(CPUHP_ONLINE);
}
```

Three things happen:

1. **PID 1 created.** `user_mode_thread(kernel_init)` creates a task running `kernel_init()` as PID 1. This task will eventually `exec` user space.
2. **PID 2 created.** `kernel_thread(kthreadd)` creates a task running `kthreadd()` as PID 2. `kthreadd` is the kernel-thread daemon: every subsequent `kthread_create()` is dispatched through it. There is a separate task for this because creating kthreads needs certain locks that the boot thread cannot easily take.
3. **The boot CPU becomes the idle thread (PID 0).** `cpu_startup_entry(CPUHP_ONLINE)` enters `do_idle()`, which is the per-CPU idle loop. When no other task is runnable, the CPU runs idle, which on ARM eventually executes `wfi` (wait for interrupt).

After `rest_init()` returns to the boot CPU's task, that task *is* PID 0 doing idle.

You can verify on a running system:

```
target# ps -A | head
  PID TTY          TIME CMD
    1 ?        00:00:01 init
    2 ?        00:00:00 kthreadd
    3 ?        00:00:00 rcu_gp
    4 ?        00:00:00 rcu_par_gp
    ...
```

PID 1 is init. PID 2 is kthreadd. PID 0 (the idle task) doesn't show in `ps` because it's a kernel-internal thread.

28.6 Phase 4, `kernel_init()` (init/main.c)

PID 1 starts here:

```
static int __ref kernel_init(void *unused)
{
    int ret;

    /*
     * Wait until kthreadd is all set-up.
     */
    wait_for_completion(&kthreadd_done);

    kernel_init_freeable();             /* device init, SMP wakeup, rootfs mount */
    async_synchronize_full();
    kprobe_free_init_mem();
    ftrace_free_init_mem();
    kgdb_free_init_mem();
```

```
    exit_boot_config();
    free_initmem();                     /* free .init.* sections — done with them */
    mark_readonly();

    /*
     * Kernel mappings are now finalized - update the userspace page-table
     * to finalize PTI.
     */
    pti_finalize();

    system_state = SYSTEM_RUNNING;
    numa_default_policy();

    rcu_end_inkernel_boot();

    do_sysctl_args();

    if (ramdisk_execute_command) {
        ret = run_init_process(ramdisk_execute_command);
        if (!ret)
            return 0;
        pr_err("Failed to execute %s (error %d)\n",
               ramdisk_execute_command, ret);
    }

    /*
     * We try each of these until one succeeds.
     *
     * The Bourne shell can be used instead of init if we are
     * trying to recover a really broken machine.
     */
    if (execute_command) {
        ret = run_init_process(execute_command);
        if (!ret)
            return 0;
        panic("Requested init %s failed (error %d).",
              execute_command, ret);
    }

    if (CONFIG_DEFAULT_INIT[0] != '\0') {
        ret = run_init_process(CONFIG_DEFAULT_INIT);
        if (ret)
            pr_err("Default init %s failed (error %d)\n",
                   CONFIG_DEFAULT_INIT, ret);
        else
            return 0;
    }

    if (!try_to_run_init_process("/sbin/init") ||
        !try_to_run_init_process("/etc/init")  ||
        !try_to_run_init_process("/bin/init")  ||
        !try_to_run_init_process("/bin/sh"))
        return 0;

    panic("No working init found.  Try passing init= option to kernel. "
          "See Linux Documentation/admin-guide/init.rst for guidance.");
}
```

In English:

1. **Wait for `kthreadd` to be ready** (so the rest of init can spawn kthreads).
2. **`kernel_init_freeable()`**: Finishes device probing, mounts the rootfs (per `root=` and `rootfstype=` from cmdline), opens `/dev/console`. This is the call that emits the *"VFS: Mounted root (ext4 filesystem) on device 179:2."* boot-log line.
3. **`free_initmem()`**: Frees the `.init.*` linker sections. The kernel's setup code has run. It's no longer needed and gets returned to the page allocator. You see the famous *"Freeing unused kernel image (initmem) memory: 1024K"* line.
4. **Fall through the init-binary search**:
   - If `rdinit=` was on the cmdline → run that (initramfs case. See Ch 29).
   - Else if `init=` was on the cmdline → run that.
   - Else if `CONFIG_DEFAULT_INIT` is set → run that.
   - Else try `/sbin/init`, `/etc/init`, `/bin/init`, `/bin/sh` in order.
5. **`run_init_process()`** calls `kernel_execve()` which `exec`s the chosen binary. **On a successful `exec`, the calling task's image is replaced**, `kernel_init()`'s code is unmapped, the new program runs. From the kernel's perspective, PID 1 is now /sbin/init (which lives in user space). `kernel_init` "returns" only in the sense that it never returns from `kernel_execve`.

After this point, the kernel is in steady state. User-space processes run. The kernel responds to syscalls and interrupts. The boot is done.

28.7 Mapping boot-log lines to source

For every memorable boot-log line, you can now name the source location. Spot-checks:

Boot log line	Source file	Function
`Booting Linux on physical CPU 0x0`	`arch/arm/kernel/setup.c`	`setup_arch` calling `pr_info`
`Linux version 6.6.0 (you@host) ...`	`init/version.c`	`start_kernel` printing `linux_banner`
`OF: fdt: Machine model: ...`	`drivers/of/fdt.c`	`early_init_dt_scan`

Boot log line	Source file	Function
`Kernel command line: ...`	`init/main.c`	`start_kernel` printing `saved_command_line`
`Memory: 444184K/524288K available ...`	`mm/mm_init.c`	`mem_init_print_info` (moved out of `mm/page_alloc.c` in v6.2+)
`clocksource: arm_global_timer: ...`	`drivers/clocksource/arm_global_timer.c`	clocksource registration
`imx-uart 2020000.serial: ...`	`drivers/tty/serial/imx.c`	`imx_uart_probe`
`Freeing unused kernel image (initmem) memory: 1024K`	`mm/page_alloc.c`	`free_initmem`
`Run /sbin/init as init process`	`init/main.c`	`run_init_process`

The table connects visible messages to initialization stages. Matching source and image versions matters more than memorizing line numbers.

28.8 Key Notes

- Architecture entry establishes CPU recognition and the initial address mappings before ordinary kernel C services exist. Failures there may precede a usable console.
- `start_kernel()` orders dependent subsystem initialization. Allocation, scheduling, interrupts and timekeeping cannot be treated as independent startup callbacks.
- `setup_arch()` translates boot-time hardware and memory information into kernel state. The DTB pointer and command line belong to the firmware-to-kernel interface.
- `rest_init()` creates the task that becomes PID 1 and the kernel-thread manager. The boot task becomes idle rather than becoming the user-space init program.
- Executing init changes PID 1’s program image without changing its PID. Root mounting and executable availability are separate requirements for this transition.
- Initialization-only sections are reclaimed after boot. Pointers to `__initdata` cannot remain part of a driver’s steady-state data ownership.

- Boot logs and initcall diagnostics expose progress, but diagnosis needs source and symbols from the matching build. More output is not a substitute for locating the failed phase.

28.9 Pitfalls

- **Confusing `vmlinux` with `zImage`.** When debugging a panic, you want symbols. The symbols are in `vmlinux`, not `zImage`. Always have `vmlinux` from the *same build* alongside your `zImage`.
- **Thinking init runs in kernel space.** PID 1 is *kernel_init* until `kernel_execve("/sbin/init")` returns successfully. Thereafter PID 1 is the `/sbin/init` user-space binary. The transition is invisible in `ps` output but real in process address space.
- **`free_initmem` recycles `.init.text` and `.init.data`.** Function names like `init_IRQ`, `setup_arch`, `start_kernel` themselves get freed, you cannot call them after boot. The compiler enforces this via the `__init` attribute, which places them in the `.init.text` section.
- **`__init` data referenced after boot.** A subtler version of the above. If a driver's probe routine stashes a pointer to a global tagged `__initdata`, that pointer becomes dangling after `free_initmem`. Symptom: crash on first access to the data, much later. Lint catches most of these. Some slip through.
- **Console output disappearing mid-boot.** Happens when the early `earlycon` is active, then the regular console-driver probes but mismatches the port, and the regular driver "takes over" without working. Symptom: log goes silent partway through. Fix: ensure your `console=` and DT `/chosen/stdout-path` agree.
- **Reading kernel source on the wrong version.** Always check what `cat /proc/version` reports on your target and read the matching tag in your local source tree. V6.6 and v6.7 can differ in startup flow.

28.10 Going deeper

- **`init/main.c`**: the file `start_kernel` lives in. Read it cover to cover. ~1000 lines, mostly the function we walked.
- **`arch/arm/kernel/head.S`** and **`arch/arm/kernel/head-common.S`**, the assembly entry. Short and educational.
- **`Documentation/arch/arm/booting.rst`**: the boot contract.
- **`Documentation/admin-guide/init.rst`**: what cmdline parameters `kernel_init` consults.
- **`Documentation/core-api/printk-formats.rst`**: printk format specifiers (`%pK`, `%pS`, `%px`, …) that you'll see all through the kernel.

- **The Bootlin “Embedded Linux kernel” training material** (free, public), covers the same startup path with different emphasis.

> Next chapter: **Chapter 29: Initramfs from scratch.** A small cpio archive illustrates the earliest user-space environment that the kernel can hand control to.

Chapter 29: Initramfs from scratch

> **What:** two small early user-space designs: a statically linked program in a cpio archive that prints “hello” and reboots, and a BusyBox-based initramfs with a shell.
>
> **Why:** the standard rootfs path (`root=/dev/mmcblk0p2`) hides a lot. An initramfs exposes the kernel-to-user-space handoff, the requirements on `/init`, and how a cpio archive becomes a populated filesystem. Buildroot and Ubuntu-base in Part V expand the runtime environment with more pieces. **MCU bridge:** Think of the rootfs as the firmware image’s file-backed runtime environment. On an MCU you link everything into flash. On Linux, programs and config live in this mounted tree. **rootfs:** root filesystem, the directory tree mounted at / that contains /bin, /etc, /dev, and libraries. **Buildroot:** a configuration-driven build system that produces a complete root filesystem and related images.
>
> **Focus:** the **cpio archive as a filesystem image** that the kernel unpacks into the initial tmpfs. Once that model is clear, the rest is just commands.

29.1 What an initramfs is

An **initramfs** is a small filesystem image that the kernel loads into RAM before any “real” disk filesystem is mounted. The kernel mounts the initramfs as `/`, runs `/init`, and from there `/init` can do whatever it wants, usually pivot to a real rootfs on disk, but for embedded systems the initramfs *is* the rootfs.

The image format is **cpio** (the venerable Unix archive format), optionally compressed with gzip/bzip2/xz/zstd. The kernel has a built-in cpio extractor that runs very early in boot.

Two ways to get the cpio archive into kernel memory:

1. **Built into the kernel image.** The kernel’s `usr/initramfs_data.cpio.gz` gets linked into `vmlinux` (and therefore into `zImage`). The kernel knows the archive’s location. On boot it extracts it.
2. **Loaded separately by the bootloader.** U-Boot reads `initramfs.cpio.gz` into RAM at some address, passes that address via the DT’s `/chosen/linux,initrd-start` and `linux,initrd-end` properties (or the legacy ATAGS), and the kernel extracts from there.

> **MCU bridge:** Think of U-Boot like a much larger boot stub plus debug monitor: it initializes hardware, loads the next image, and gives you commands before Linux starts. **U-Boot:** the bootloader that initializes enough hardware to load and start the Linux kernel.

Option 1 is simpler for tiny images. Option 2 allows changes to the rootfs without rebuilding the kernel. The examples compare both packaging choices.

29.2 The smallest possible initramfs

A single binary that prints “hello”, waits a moment, and reboots:

`hello.c`:

```
#include <stdio.h>
#include <unistd.h>
#include <sys/reboot.h>
#include <sys/syscall.h>
#include <linux/reboot.h>

int main(void)
{
    /* Open the kernel console so printf goes somewhere visible. */
    /* (kernel_init opens /dev/console as fd 0,1,2 before running /init.) */

    puts("\n*** hello from a one-binary initramfs ***\n");

    for (int i = 5; i > 0; i--) {
        printf("rebooting in %d...\n", i);
        sleep(1);
    }

    /* Use the raw syscall — we don't depend on libc init having done anything. */
    syscall(__NR_reboot,
            LINUX_REBOOT_MAGIC1, LINUX_REBOOT_MAGIC2,
            LINUX_REBOOT_CMD_RESTART, NULL);

    return 0;   /* unreachable on a successful reboot */
}
```

Build statically so we have no library dependencies on the target:

```
$ arm-none-linux-gnueabihf-gcc -static -Os -o init hello.c
$ arm-none-linux-gnueabihf-strip init
$ ls -lh init
-rwxr-xr-x 1 you you 480K Jan 22 init
```

480 KB statically linked against glibc. That’s ten times the size of the program itself. Glibc is fat. With **musl** instead:

```
$ musl-gcc -static -Os -o init hello.c   # or use arm-linux-musleabihf-gcc
$ arm-none-linux-gnueabihf-strip init
$ ls -lh init
-rwxr-xr-x 1 you you  30K Jan 22 init
```

30 KB. That's roughly the minimum a statically-linked C program reaches.

Build the cpio archive:

```
$ mkdir -p initramfs
$ cp init initramfs/init                # MUST be named "init" at the root
$ cd initramfs
$ find . | cpio -o -H newc | gzip > ../initramfs.cpio.gz
$ ls -lh ../initramfs.cpio.gz
-rw-r--r-- 1 you you 14K Jan 22 ../initramfs.cpio.gz
```

`-H newc` selects the SVR4 / portable ASCII cpio format the kernel expects. The result is a 14 KB compressed image containing one file (`/init`, executable). The kernel will extract this into a tmpfs, execute `/init`, and we'll see the hello message.

29.3 Booting it from U-Boot

Drop `initramfs.cpio.gz` into your TFTP server, then:

> **TFTP:** Trivial File Transfer Protocol, a simple network protocol U-Boot commonly uses to fetch kernels from the host.

```
=> tftp 0x82000000 zImage
=> tftp 0x83000000 imx6ull.dtb
=> tftp 0x84000000 initramfs.cpio.gz
=> setenv initrd_size ${filesize}
=> setenv bootargs 'console=ttymxc0,115200 earlycon rdinit=/init'
=> bootz 0x82000000 0x84000000:${initrd_size} 0x83000000
```

Notice the new things:

- **`tftp 0x84000000 initramfs.cpio.gz`**: load the initramfs at a third DRAM address.
- **`rdinit=/init`**: tells `kernel_init` to run `/init` from the initramfs (instead of `/sbin/init` from a disk rootfs).
- **`setenv initrd_size ${filesize}`**: after `tftp`, U-Boot stores the loaded byte count in `filesize`. Raw initramfs images need both address and size.
- **`bootz 0x82000000 0x84000000:${initrd_size} 0x83000000`**: the second argument is now `initrd_address:initrd_size` (no longer `-`). U-Boot writes both `linux,initrd-start` and `linux,initrd-end` into the DT.

You should see:

```
[   2.087xxx] Freeing unused kernel image (initmem) memory: 1024K
[   2.110xxx] Run /init as init process

*** hello from a one-binary initramfs ***

rebooting in 5...
rebooting in 4...
rebooting in 3...
rebooting in 2...
rebooting in 1...
[   7.214xxx] reboot: Restarting system
```

That is the smallest user space that boots Linux on this board: 30 KB compiled.

29.4 A BusyBox-based initramfs (real shell)

`hello` is just a demo. A practical initramfs has a shell and some utilities, plus an init system. **BusyBox** packs all of these into one statically-linked binary of about 600 KB. It exposes hundreds of applets, most of the common Unix utilities.

```
$ cd ~/imx6ull/src
$ wget https://busybox.net/downloads/busybox-1.36.1.tar.bz2
$ tar xf busybox-1.36.1.tar.bz2
$ cd busybox-1.36.1
$ make ARCH=arm CROSS_COMPILE=arm-none-linux-gnueabihf- defconfig

# Enable static linking
$ make ARCH=arm CROSS_COMPILE=arm-none-linux-gnueabihf- menuconfig
# Settings → Build static binary (no shared libs) → [*]
# Save and exit.

$ make ARCH=arm CROSS_COMPILE=arm-none-linux-gnueabihf- -j$(nproc)
$ ls -lh busybox
-rwxr-xr-x 1 you you 1.2M Jan 22 busybox
$ arm-none-linux-gnueabihf-strip busybox
$ ls -lh busybox
-rwxr-xr-x 1 you you 580K Jan 22 busybox
```

580 KB statically linked against glibc. With musl, ~450 KB.

Now build a rootfs around it:

```
$ cd ~/imx6ull
$ rm -rf initramfs
$ mkdir -p initramfs/{etc/init.d,proc,sys,dev,tmp,var,root}

# Install BusyBox and all applet symlinks.
# This uses the BusyBox build system, so the host never tries to execute
# the ARM busybox binary.
$ make -C ~/imx6ull/src/busybox-1.36.1 \
      ARCH=arm CROSS_COMPILE=arm-none-linux-gnueabihf- \
```

```
      CONFIG_PREFIX=$PWD/initramfs install

# If your BusyBox config did not install /sbin/init, add it.
$ test -e initramfs/sbin/init || ln -s /bin/busybox initramfs/sbin/init

# A minimal inittab for busybox's init
$ cat > initramfs/etc/inittab <<'EOF'
::sysinit:/etc/init.d/rcS
::respawn:-/bin/sh
::ctrlaltdel:/sbin/reboot
::shutdown:/bin/umount -a -r
EOF

# A minimal startup script
$ mkdir -p initramfs/etc/init.d
$ cat > initramfs/etc/init.d/rcS <<'EOF'
#!/bin/sh
mount -t proc none /proc
mount -t sysfs none /sys
mount -t devtmpfs none /dev
echo "*** BusyBox initramfs is up ***"
EOF
$ chmod +x initramfs/etc/init.d/rcS

# Repackage
$ cd initramfs
$ find . | cpio -o -H newc | gzip > ../initramfs.cpio.gz
$ ls -lh ../initramfs.cpio.gz
-rw-r--r-- 1 you you 250K Jan 22 ../initramfs.cpio.gz
```

250 KB compressed. Load and boot as before, but **drop `rdinit=/init`** from bootargs, busybox-init wants to run as `/sbin/init`, which is the default search order:

```
=> setenv bootargs 'console=ttymxc0,115200 earlycon'
=> tftp 0x84000000 initramfs.cpio.gz
=> setenv initrd_size ${filesize}
=> bootz 0x82000000 0x84000000:${initrd_size} 0x83000000
```

After kernel boots, you should see:

```
[   2.298xxx] Run /sbin/init as init process
*** BusyBox initramfs is up ***

Please press Enter to activate this console.

# ls /
bin   dev  etc   lib   proc  root  sbin  sys  tmp  usr  var
# cat /proc/cpuinfo
processor  : 0
model name : ARMv7 Processor rev 5 (v7l)
...
```

In this example, BusyBox supplies a shell and the enabled applets. Their availability depends on its configuration, not on the presence of a full desktop distribution.

29.5 Embedded vs separate

Two architectures for shipping the initramfs:

Embedded in the kernel

```
# In kernel's .config:
CONFIG_INITRAMFS_SOURCE="/home/you/imx6ull/initramfs.cpio.gz"

$ make ARCH=arm zImage     # zImage now includes initramfs internally
```

The `zImage` grows by the initramfs size. No separate bootloader step. Convenient for a *fixed* initramfs that's part of the kernel's release artifact. Used by single-purpose appliances and by recovery kernels.

Separate file loaded by bootloader

In the separate-file example above, the kernel is one file (`zImage`) and the initramfs is another (`initramfs.cpio.gz`). The bootloader loads both. The advantage is *changing the rootfs doesn't require rebuilding the kernel*.

Choose based on how often the rootfs changes vs the kernel. For early development, embedded is faster (one artefact). For production, separate is more flexible.

29.6 Init systems compared

Once you have user space running, what runs as PID 1? Three common choices for embedded:

Init	Footprint	What you get
BusyBox init	< 1 KB on top of busybox	One inittab; classic Unix `respawn` / `sysinit` semantics. Sufficient for 95% of embedded
`sysvinit`	~80 KB	The traditional System V `/etc/rc.d/init.d/*` scripts. Familiar to anyone from before 2010
`systemd`	~5 MB + dependencies	Service unit files, sockets, journal, timers. Powerful, big, dependency-heavy

For embedded, **BusyBox init** is the default. We use it in this book through Chapter 31 and switch to discussing systemd in Chapter 33.

29.7 Key Notes

- An initramfs supplies files before the normal storage-backed root is available. The kernel unpacks a cpio archive, then early user space owns the next boot or recovery decision.
- `/init` must be executable for the target architecture, with its interpreter and libraries present when needed. Static linking removes shared-library dependencies, not the requirements for a console or filesystem mounts.
- A separately loaded raw archive needs its address and byte count in the bootloader handoff. The saved size must describe the initramfs transfer, not another image loaded afterward.
- Embedding the archive ties its contents to the kernel artifact. Loading it separately permits independent replacement but adds another image to the bootloader's release and memory-layout contract.
- BusyBox combines selected applets in one binary. Its shell and utilities still need startup policy, mount points and device-node availability.
- procfs exposes kernel and process information, sysfs exposes device-model attributes, and devtmpfs supplies device nodes. Their mounts are distinct from the files stored in the cpio archive.
- PID 1 is a system-lifetime process. The rebooting example is a demonstration of that boundary, not a service manager or a qualified recovery system.

29.8 Pitfalls

- **`/init` must exist at the root and be executable.** Forget either and the kernel panics with "No filesystem could mount root, tried: ramfs". Cpio archives don't error on missing init.
- **`/init` linked dynamically.** If `/init` depends on `libc.so.6` and `libc.so.6` isn't in the cpio, exec fails silently. Either statically link (recommended) or include the needed `.so` files in `/lib/` of the initramfs.
- **`init=` vs `rdinit=`.** `init=path` tells the kernel to look on the *root filesystem* (the one specified by `root=`). `rdinit=path` tells it to look on the *initramfs*. For initramfs-only boots, use `rdinit=` or just rely on the default `/init` lookup.
- **Forgetting the initramfs size in `bootz`.** For a raw `initramfs.cpio.gz`, use `bootz kernel_addr initrd_addr:initrd_size fdt_addr`. The size usually comes from U-Boot's `${filesize}` after `tftp` or `load`.

- **cpio archive built without `-H newc`.** Default cpio format isn't what the kernel expects. The unpacker reports an error and gives up. Always `-H newc`.
- **Trailing slash on `find ..`** `find .` gives relative paths like `./init`, which is what cpio wants. `find /home/you/initramfs` gives absolute paths, so the archive ends up with `/home/you/initramfs/init` and the kernel cannot find `/init`. Always `cd` into the rootfs first.
- **BusyBox not statically linked.** Built dynamic by default. Forgetting to set static causes the binary to need glibc shared objects you don't have in the initramfs. Symptom: `Kernel panic - not syncing: Attempted to kill init!` because `exec` fails.
- **No `/dev/console` before `kernel_init` opens it.** Kernel handles this automatically via devtmpfs auto-mount, but if you disable that in `.config`, you'll see `Warning: unable to open an initial console` and lose all stdio in your init. Keep `CONFIG_DEVTMPFS=y` and `CONFIG_DEVTMPFS_MOUNT=y`.

29.9 Going deeper

- **`Documentation/filesystems/ramfs-rootfs-initramfs.rst`**: canonical kernel doc on the initramfs mechanism.
- **`Documentation/admin-guide/initrd.rst`**: older initrd mechanism (predecessor to initramfs). Useful context.
- **BusyBox manual** at `busybox.net/about.html` and the per-applet `--help`.
- **`init/initramfs.c`**: the kernel's cpio extractor. Short and readable.
- **`klibc`**: an even smaller libc-replacement than musl, designed specifically for in-kernel-cpio-initramfs static binaries.

> Next chapter: **Chapter 29A: Initramfs as a recovery system.** The same early user-space mechanism can host diagnostics and a controlled recovery path independently of the main rootfs.

Chapter 29A: Initramfs as a recovery system

Chapter 29 described a tiny initramfs as the kernel's early PID 1 environment.

In real products, initramfs is often more useful than that. It becomes a small recovery system that can boot even when the real root filesystem is broken. This chapter builds that idea in a direct way.

The goal is not to make a pretty rescue UI. The goal is to make a system that can:

- boot without the main root filesystem,
- print useful logs,
- mount the real root filesystem read-only,
- collect files for debugging,
- enter a shell,
- run a controlled update or factory reset path.

That is a practical embedded Linux tool.

29A.1 Where the recovery initramfs sits

Normal boot:

```
U-Boot -> kernel + DTB -> real rootfs -> /sbin/init
```

Recovery boot:

```
U-Boot -> kernel + DTB + initramfs -> /init inside initramfs
```

The recovery initramfs is loaded by U-Boot together with the kernel. The kernel unpacks it into RAM and runs `/init`.

Because it lives in RAM, it can still boot when the SD card, eMMC partition, or root filesystem has a problem.

Piece	Normal boot	Recovery boot
Kernel	Same kernel, usually	Same kernel, or a smaller recovery kernel
DTB	Board DTB	Same board DTB
Root filesystem	SD/eMMC/NFS root	initramfs in RAM
PID 1	`/sbin/init` from rootfs	`/init` from initramfs
Main purpose	Run the product	Repair, debug, update, or reset

29A.2 Start from the BusyBox rootfs

The recovery case uses the BusyBox layout from Chapter 29 as its base. The paths below illustrate that layout rather than requiring an earlier build.

```
$ cd ~/imx6ull/code/ch29
$ cp -a initramfs initramfs-recovery
```

The directory should look like this:

```
initramfs-recovery/
  bin/
  dev/
  etc/
  proc/
  sbin/
  sys/
  tmp/
  usr/
```

Now replace the simple `/init` with a recovery script.

29A.3 A small recovery `/init`

Save this as `initramfs-recovery/init`:

```
#!/bin/sh

export PATH=/sbin:/bin:/usr/sbin:/usr/bin
```

```
mount -t proc proc /proc
mount -t sysfs sysfs /sys
mount -t devtmpfs devtmpfs /dev 2>/dev/null || mount -t tmpfs tmpfs /dev
mount -t tmpfs tmpfs /tmp

echo
echo "i.MX6ULL recovery initramfs"
echo "Kernel command line:"
cat /proc/cmdline
echo

mount_real_root_ro()
{
    mkdir -p /mnt/root

    echo "Trying /dev/mmcblk0p2 as real rootfs..."
    if mount -o ro /dev/mmcblk0p2 /mnt/root; then
        echo "Mounted real rootfs read-only at /mnt/root"
        return 0
    fi

    echo "Could not mount /dev/mmcblk0p2"
    return 1
}

collect_logs()
{
    mkdir -p /tmp/recovery-logs

    dmesg > /tmp/recovery-logs/dmesg.txt
    cat /proc/cmdline > /tmp/recovery-logs/cmdline.txt
    mount > /tmp/recovery-logs/mounts.txt

    if [ -d /mnt/root/var/log ]; then
        cp -a /mnt/root/var/log /tmp/recovery-logs/root-var-log
    fi

    echo "Logs collected in /tmp/recovery-logs"
}

factory_reset()
{
    echo "Factory reset example"
    echo "This book does not erase anything automatically."
    echo "In a real product, erase only the data partition, not the bootloader."
}

while true; do
    echo
    echo "Recovery menu"
    echo "1) Shell"
    echo "2) Mount real rootfs read-only"
    echo "3) Collect logs"
    echo "4) Factory reset placeholder"
    echo "5) Reboot"
    printf "> "
    read choice

    case "$choice" in
```

```
        1)
            sh
            ;;
        2)
            mount_real_root_ro
            ;;
        3)
            grep -q " /mnt/root " /proc/mounts || mount_real_root_ro
            collect_logs
            ;;
        4)
            factory_reset
            ;;
        5)
            reboot -f
            ;;
        *)
            echo "Unknown choice"
            ;;
    esac
done
```

Make it executable:

```
$ chmod +x initramfs-recovery/init
```

This script is plain on purpose. In recovery code, boring is good. You want behavior you can read at 2 AM on a failing board.

29A.4 Repack the recovery initramfs

```
$ cd initramfs-recovery
$ find . -print0 | cpio --null -ov --format=newc | gzip -9 > ../initramfs-
recovery.cpio.gz
$ cd ..
```

Copy it to the TFTP directory:

```
$ cp initramfs-recovery.cpio.gz ~/imx6ull/tftp/
```

29A.5 Boot it from U-Boot

Use the same kernel and DTB, but pass the recovery initramfs as the second `bootz` argument.

```
=> setenv serverip 192.168.1.10
=> setenv ipaddr 192.168.1.50
=> tftp 0x82000000 zImage
=> tftp 0x83000000 imx6ull-your-board.dtb
=> tftp 0x84000000 initramfs-recovery.cpio.gz
```

```
=> setenv initrd_size ${filesize}
=> setenv bootargs 'console=ttymxc0,115200 earlycon loglevel=8 rdinit=/init'
=> bootz 0x82000000 0x84000000:${initrd_size} 0x83000000
```

The `addr:size` form matters. A raw initramfs file does not carry its own size in a way `bootz` can always infer. We save `${filesize}` immediately after the initramfs TFTP command, then pass it to `bootz`.

Expected result:

```
i.MX6ULL recovery initramfs
Kernel command line:
console=ttymxc0,115200 earlycon loglevel=8 rdinit=/init

Recovery menu
1) Shell
2) Mount real rootfs read-only
3) Collect logs
4) Factory reset placeholder
5) Reboot
>
```

29A.6 Add a real-root handoff

Sometimes recovery should repair something and then continue into the normal root filesystem.

The usual flow is:

1. Mount the real root filesystem.
2. Mount `proc`, `sys`, and `dev` under it.
3. Replace the initramfs process with the real init process.

Example:

```
mount /dev/mmcblk0p2 /mnt/root || {
    echo "Cannot mount real rootfs"
    sh
}

mount --move /proc /mnt/root/proc
mount --move /sys  /mnt/root/sys
mount --move /dev  /mnt/root/dev

exec switch_root /mnt/root /sbin/init
```

Use this only when the real root filesystem is healthy enough to boot. If the point is recovery, keep the menu path available.

29A.7 Network recovery

For factory or development boards, recovery often needs Ethernet.

If BusyBox was built with `ip`, `udhcpc`, `wget`, and `tftp`, you can do:

```
ip link set eth0 up
udhcpc -i eth0
```

Or use a fixed address:

```
ip addr add 192.168.1.51/24 dev eth0
ip link set eth0 up
ip route add default via 192.168.1.1
```

Then fetch an update:

```
tftp -g -r rootfs.ext4.gz -l /tmp/rootfs.ext4.gz 192.168.1.10
tftp -g -r rootfs.ext4.gz.sha256 -l /tmp/rootfs.ext4.gz.sha256 192.168.1.10
cd /tmp
sha256sum -c rootfs.ext4.gz.sha256
```

Do not write the update until the checksum passes.

29A.8 Safe update rules

Recovery code can save a product, but it can also destroy one if it writes the wrong block device.

Use rules like these:

Rule	Why
Print the target block device before writing	Makes mistakes visible on the console
Verify checksum before flashing	Detects broken downloads
Do not update the bootloader unless necessary	A bad bootloader update can remove recovery
Do not write a mounted filesystem	Avoids corruption

Rule	Why
Keep factory data separate from rootfs	Reset should not erase MAC address, serial number, or calibration
Prefer A/B rootfs for products	Allows rollback after a failed update

For example:

```
TARGET=/dev/mmcblk0p2

echo "About to write rootfs to $TARGET"
echo "Type YES to continue:"
read answer
[ "$answer" = "YES" ] || exit 1

mount | grep -q "$TARGET" && {
    echo "$TARGET is mounted. Refusing to write."
    exit 1
}

sha256sum -c /tmp/rootfs.ext4.gz.sha256 || exit 1
gunzip -c /tmp/rootfs.ext4.gz > "$TARGET"
sync
```

This is not fancy, but it prevents many expensive mistakes.

29A.9 U-Boot recovery trigger

Recovery is usually selected by U-Boot.

Common triggers:

Trigger	Example
Button held at power-on	Read GPIO in U-Boot, boot recovery if pressed
Bootcount exceeded	If Linux fails too many times, boot recovery
Environment variable	`setenv bootmode recovery`
Factory command	A production tool asks U-Boot to boot recovery

Trigger	Example
Missing rootfs	U-Boot cannot load normal rootfs, so it loads recovery

Example U-Boot logic:

```
if test "${bootmode}" = "recovery"; then
    run boot_recovery;
else
    run boot_normal;
fi
```

Recovery is most useful when U-Boot can choose it before Linux is involved.

29A.10 Key Notes

- Recovery independence depends on the whole boot path: firmware must still load a working kernel, board DTB and archive. Running from RAM avoids reliance on the main rootfs, not every storage or hardware dependency.
- The recovery `/init` owns diagnostics and the decision to remain in recovery or hand off to normal init. `exec switch_root` changes that ownership rather than starting a second normal system beside the first.
- Read-only inspection and log collection are different operations from repair or image replacement. The factory-reset placeholder deliberately performs no erase.
- Network recovery requires both kernel device support and selected BusyBox applets. U-Boot network access alone does not establish a usable Linux network path.
- A downloaded checksum detects damaged bytes, not an untrusted sender. Target identity, mount state, image compatibility and update authorization remain separate write boundaries.
- Factory identity and calibration belong outside partitions erased by reset or rootfs replacement. Protecting the bootloader preserves the path that makes recovery possible.
- Logs collected in `/tmp` live in RAM and are lost at reboot unless exported. A recovery menu appearing is not evidence of safe updates or power-loss recovery.

29A.11 Common mistakes

Symptom	Likely cause
Kernel ignores initramfs	`bootz` did not receive `addr:size` for the raw initramfs
Kernel says `No working init found`	`/init` is missing, not executable, or built for the wrong architecture
Shell commands are missing	BusyBox applets were not enabled
`/dev/mmcblk0p2` does not exist	MMC driver, device tree, or partition table problem
Update writes the wrong partition	Script used a hard-coded device name without checking
Recovery loses serial number	Factory data was stored inside an erased partition

Recovery is not only a debugging trick. It is part of product design.

Previous chapter: Chapter 29 - Initramfs from scratch

Next chapter: Chapter 30 - Kernel configuration

Chapter 30: Kernel configuration deep-dive

> **rootfs:** root filesystem, the directory tree mounted at / that contains /bin, /etc, /dev, and libraries. **MCU bridge:** Think of the rootfs as the firmware image's file-backed runtime environment. On an MCU you link everything into flash. On Linux, programs and config live in this mounted tree.

> **What:** the kernel's Kconfig system, configuration editors, `.config`, defconfig snapshots and the options that shape an i.MX6ULL image's capabilities and boot dependencies.
>
> **Why:** earlier examples used `imx_v6_v7_defconfig` as a starting point. Product configuration makes choices about image size, scheduling, device support and engineering diagnostics explicit. **PREEMPT_RT:** the Linux real-time patch set that makes more kernel paths preemptible and reduces latency.
>
> **Focus:** `.config` is the resolved configuration. `menuconfig`, `xconfig` and other editors operate within Kconfig dependency rules rather than independently choosing what can compile.

30.1 The Kconfig system

The kernel has ~7000 configurable options across hundreds of `Kconfig` files scattered through the source tree. Each is a `CONFIG_FOO` symbol that ends up as `=y` (compiled in), `=m` (loadable module), or absent (compiled out) in the final `.config`.

Five `make` targets cover 95% of use:

Target	What it does
`make defconfig`	Use the default config (`arch/$ARCH/configs/defconfig`)
`make <name>_defconfig`	Use a named per-board/per-arch starting point (e.g., `imx_v6_v7_defconfig`)
`make menuconfig`	Interactive ncurses-based config editor

Target	What it does
`make oldconfig`	Update an existing `.config` against new Kconfig options; prompt for each new one
`make savedefconfig`	Distill the current `.config` down to the minimum diff from defaults and save to `defconfig`

Other UIs exist (`nconfig`, `xconfig`, `gconfig`), same functionality, different UIs. `menuconfig` is the one everyone uses.

30.2 `make menuconfig` tour

From a configured tree:

```
$ make ARCH=arm CROSS_COMPILE=arm-none-linux-gnueabihf- menuconfig
```

You'll get a full-screen ncurses interface. Navigation:

- **Arrow keys**: move
- **Enter**: descend into a submenu
- **Space**: cycle `=y` / `=m` / disabled
- **`?`**: show help for the current option (very useful, every Kconfig symbol has descriptive help)
- **`/`**: search for an option by name (case-insensitive substring. Multiple matches show their location in the menu tree)
- **`esc esc`**: go back
- **`Q`**: quit. Prompts to save

A representative top-level menu:

```
  General setup  --->
  System Type  --->
  Bus support  --->
  Kernel Features  --->
  Boot options  --->
  CPU Power Management  --->
  Floating point emulation  --->
  Userspace binary formats  --->
  Power management options  --->
  [*] Networking support  --->
  Device Drivers  --->
```

```
  File systems  --->
  Security options  --->
-*- Cryptographic API  --->
  Library routines  --->
  Kernel hacking  --->
```

Each `--->` is a submenu. Each `[*]` / `<*>` / `<M>` / `< >` is a yes/module/no option. Each `(...)` is a string or integer option. The structure is consistent: subsystems group their options together.

30.3 The `.config` file

> **Privilege boundary:** $ means normal user. # or sudo means root and can change host or target state. After a privileged command, verify the expected device, service, or file appears before continuing. Roll back by undoing the config change or stopping the service you just enabled.

After saving in `menuconfig`, the entire state lives in `.config` at the kernel-tree root. It is plain text. Each line is one of: `CONFIG_FOO=y`, `CONFIG_FOO=m`, `CONFIG_FOO="string"`, `CONFIG_FOO=10`, or `# CONFIG_FOO is not set`.

```
$ wc -l .config
2147 .config

$ head -10 .config
#
# Automatically generated file; DO NOT EDIT.
# Linux/arm 6.6.0 Kernel Configuration
#
CONFIG_CC_VERSION_TEXT="arm-none-linux-gnueabihf-gcc (Ubuntu 11.4.0-1ubuntu1~22.04)
11.4.0"
CONFIG_GCC_VERSION=110400
CONFIG_CLANG_VERSION=0
CONFIG_AS_IS_GNU=y
CONFIG_AS_VERSION=24200
...
```

You **can** edit `.config` by hand, but the comment is honest: changes you make can be silently undone the next time `make` runs, because the Kconfig dependencies might reject your edit. Always go through `make olddefconfig` or `make oldconfig` after a manual edit to let the system reconcile.

30.4 defconfigs, capturing your customisation

A *defconfig* is the **minimum** set of options that, applied on top of the architecture's defaults, reproduces your `.config`. It's how you preserve a customisation without saving the full 2000-line `.config`.

```
$ make ARCH=arm savedefconfig
$ ls defconfig
defconfig
$ wc -l defconfig
324 defconfig
```

That `defconfig` file (324 lines vs 2147 for full `.config`) contains *only* the lines that differ from the architecture default. Save it as `arch/arm/configs/myboard_defconfig`, commit it to git, and anyone can reproduce your build:

```
$ make ARCH=arm myboard_defconfig
$ make ARCH=arm -j$(nproc) zImage
```

With those two commands, the second engineer has the same kernel. This is why upstream ships `imx_v6_v7_defconfig` instead of full `.config`s.

30.5 The dozen knobs that matter most

Reading every help text in `menuconfig` takes hours and is not the fastest way to learn what matters. Here are the dozen options that matter most on the path from `defconfig` to an i.MX6ULL image:

Preemption model, `CONFIG_PREEMPT_*`

```
General setup
  Preemption Model  --->
    ( ) No Forced Preemption (Server)
    (X) Voluntary Kernel Preemption (Desktop)     ← default
    ( ) Preemptible Kernel (Low-Latency Desktop)
    ( ) Fully Preemptible Kernel (Real-Time)
```

Determines how long kernel code can hold the CPU before letting another thread run. **`PREEMPT_NONE`** maximizes throughput, **`PREEMPT_VOLUNTARY`** (the default) is a reasonable compromise, **`PREEMPT`** improves desktop responsiveness, `PREEMPT_RT` turns the kernel into a real-time kernel with bounded, low latency (Chapter 52A is dedicated to this).

For a typical embedded product, `PREEMPT_VOLUNTARY` is fine. For motion control or audio with hard latency budgets, `PREEMPT_RT`. For a router pushing packets, `PREEMPT_NONE`.

Tick rate, `CONFIG_HZ`

```
Kernel Features
  Timer frequency  --->
    ( ) 100 Hz
    (X) 250 Hz
    ( ) 300 Hz
    ( ) 1000 Hz
```

How often the kernel scheduler tick fires. Higher = better latency, more overhead. 250 Hz is the modern default for non-server workloads. For battery-powered devices, lower. For low-latency, 1000 Hz.

Tickless idle, `CONFIG_NO_HZ_*`

```
General setup
  Timers subsystem
    Timer tick handling  --->
      ( ) Periodic timer ticks (constant rate, no dynticks)
      (X) Idle dynticks system (tickless idle)
      ( ) Full dynticks system (tickless)
```

`NO_HZ_IDLE` lets the CPU sleep longer when idle by stopping the tick. Saves power. The default for v6.x.

Devtmpfs, `CONFIG_DEVTMPFS` + `CONFIG_DEVTMPFS_MOUNT`

```
Device Drivers
  Generic Driver Options
    [*] Maintain a devtmpfs filesystem to mount at /dev
    [*]   Automount devtmpfs at /dev, after the kernel mounted the rootfs
```

Required for the modern `/dev` model: kernel populates `/dev/*` from device probes. Without it, you need `udev` or `mdev` in userspace. **Always enable both.**

> **udev:** the user-space device manager that reacts to kernel device events and creates policy-driven /dev nodes.

Initramfs, `CONFIG_BLK_DEV_INITRD` + `CONFIG_INITRAMFS_SOURCE`

```
General setup
  [*] Initial RAM filesystem and RAM disk (initramfs/initrd) support
  ()  Initramfs source file(s)            ← optional, embeds an initramfs
```

These options support the initramfs designs discussed in Chapter 29. The source field embeds a cpio archive directly into `zImage`.

Modules, `CONFIG_MODULES` and `CONFIG_MODULE_UNLOAD`

```
[*] Enable loadable module support  --->
    [*]   Module unloading
    [*]   Forced module unloading      ← off in production usually
```

Enables `=m`-style driver loading. Useful for development (load/unload a driver without reboot). Off if you want a monolithic kernel.

`console=` defaults via DT, already covered

The kernel reads `chosen.stdout-path` from DT plus the `console=` cmdline. Both are independent of `.config`.

USB, `CONFIG_USB` + `CONFIG_USB_EHCI_HCD` + `CONFIG_USB_GADGET`

```
Device Drivers
  [*] USB support  --->
       <*>   Support for Host-side USB
       <*>   EHCI HCD (USB 2.0) support
       [*]   USB Gadget Support
```

EHCI host = USB host port on i.MX6ULL. Gadget = device-mode (Chapter 55).

Network drivers, `CONFIG_FEC` for the i.MX FEC

```
Device Drivers
  [*] Network device support  --->
       Ethernet driver support  --->
         <*>   Freescale devices  --->
                 <*>   FEC (Freescale FEC and i.MX6UL/ULL)
```

Without `CONFIG_FEC`, your Ethernet won't work. Default `imx_v6_v7_defconfig` enables this. If you start from a stripped-down config you may have to add it back.

MMC / SD, `CONFIG_MMC` + `CONFIG_MMC_SDHCI_ESDHC_IMX`

```
Device Drivers
  <*> MMC/SD/SDIO card support  --->
        <*>   Secure Digital Host Controller Interface support
        <*>     Freescale eSDHC/uSDHC i.MX controller
```

Required to access SD card and eMMC.

Filesystem support, `CONFIG_EXT4_FS`, `CONFIG_F2FS_FS`, `CONFIG_VFAT_FS`

```
File systems  --->
  <*> The Extended 4 (ext4) filesystem
  <*> F2FS filesystem support
  <*> DOS/FAT/EXFAT/NT Filesystems  --->
        <*>   VFAT (Windows-95) fs support
```

Need at least one filesystem matching your `root=`. Most embedded systems use ext4 on eMMC, F2FS on eMMC for flash longevity, FAT for boot partitions.

Kernel debug options, `CONFIG_DEBUG_KERNEL` + `CONFIG_DEBUG_INFO`

```
Kernel hacking  --->
  [*] Kernel debugging
      Compile-time checks and compiler options  --->
        Debug information --->
          (X) Generate DWARF Version 5 debuginfo
```

For development, keep these on, gdb stack traces and oops decoding need DWARF symbols. For production, off, they bloat `vmlinux` by 100+ MB (though `zImage` stays small).

30.6 Building a smaller kernel

Default `imx_v6_v7_defconfig` produces a ~6 MB `zImage`. For a tighter image:

```
$ make ARCH=arm imx_v6_v7_defconfig
$ make ARCH=arm menuconfig
```

Aggressive trimming:

- **Disable unused architectures/SoCs.** `System Type → Multiple platform selection`: turn off `ARM_AT91`, `ARM_OMAP`, `ARCH_BCM`, etc., keep only `ARCH_MXC`. Saves ~500 KB.
- **Disable unused drivers.** `Device Drivers → Sound card support → ALSA for SoC audio support`, off if you don't have audio. Saves ~200 KB.

> **ALSA:** Linux's kernel and user-space audio stack.

- **Disable unused filesystems.** Only ext4 needed? Disable F2FS, FAT, etc. Saves ~100 KB.
- **Disable IPv6 / IPv4 sub-features you don't need.** TCP / UDP / unicast routing, yes. Multicast routing / fib_rules / netfilter, no. Saves ~500 KB.
- **Disable debug info.** `Kernel hacking → Kernel debugging → off`. Cuts `vmlinux` size dramatically. `zImage` shrinks by ~1 MB. Keep your dev build separate.

A trimmed image for i.MX6ULL can reach ~3 MB. Below that requires turning off things you usually want (e.g., loadable modules → +200 KB savings, but you lose `modprobe`).

30.7 Useful Kconfig idioms

Find an option by name

```
( ) Inside menuconfig, press / and type CONFIG_FOO
```

The search results include the menu path. Quote: "`Symbol: PREEMPT_RT [=n]. Defined at kernel/Kconfig.preempt:64. Location: Main menu → General setup → Preemption Model → ..`". You can jump there with Enter.

See dependencies

```
( ) Highlight an option and press ? for help; the "Depends on:" lines list its
preconditions
```

If `CONFIG_FOO` depends on `BAR && BAZ`, enabling FOO requires both BAR and BAZ. Menuconfig hides FOO if its dependencies aren't met. Search to find the missing dependency.

Strip a feature from a defconfig

```
$ make ARCH=arm imx_v6_v7_defconfig
$ ./scripts/config --disable AUDIT
$ make ARCH=arm olddefconfig
```

`scripts/config` is the headless command-line equivalent of menuconfig, invaluable for CI.

Combine multiple defconfigs

```
$ make ARCH=arm imx_v6_v7_defconfig
$ ./scripts/config --enable PREEMPT_RT
$ make ARCH=arm olddefconfig
```

You start from a known baseline and apply targeted changes. Easier to maintain than committing a full bespoke defconfig.

30.8 Key Notes

- Kconfig resolves requested options against dependencies and defaults. A hand-edited symbol does not necessarily survive resolution or become compiled code.
- `.config` records the complete resolved state, while a defconfig records a compact starting point relative to defaults. Its meaning is tied to the architecture and kernel revision.
- Built-in support is available before module loading. Root storage and filesystem dependencies cannot live only in modules on the rootfs that they are needed to mount.

- Configuration enables capabilities, DT describes hardware, and bootargs select behavior for a boot. Changing a root device or console argument is not the same as enabling its driver.
- Trimming removes functionality as well as bytes. Recovery storage, consoles and optional maintenance paths remain requirements even when the normal product does not use them.
- Preemption, timer and idle choices balance latency, overhead and power. An option name alone does not establish a measured real-time bound.
- Debug information and runtime diagnostics serve different purposes. Host-side symbols and development-only interfaces need explicit ownership across product configurations.

30.9 Pitfalls

- **Editing `.config` directly and not running `make oldconfig`.** Your edits may be silently reverted on next build because of dependency rules. Always go through Kconfig.
- **Forgetting `ARCH=arm`.** Without it, `make menuconfig` shows you the *host's* Kconfig (x86 or arm64), not the cross target's. Always export.
- **`make defconfig` overwrites `.config`.** If you've made customisations and run `make defconfig`, those changes are gone. Save with `make savedefconfig` first.
- **Different defconfigs in different parts of the tree.** `arch/arm/configs/imx_v6_v7_defconfig` is for ARM 32-bit i.MX6/i.MX7. `arch/arm64/configs/defconfig` is for everything ARM 64-bit. Don't cross them.
- **Disabling `CONFIG_MMU`.** There is a `CONFIG_MMU` symbol you can clear. Don't. The result is `nommu` Linux for parts without an MMU, which won't run on Cortex-A7 anyway.

> **MCU bridge:** Think of the MMU as a hardware address translator in front of every load/store. Cortex-M usually runs physical addresses directly. Linux relies on virtual addresses and page permissions. **MMU:** Memory Management Unit, hardware that translates virtual addresses to physical addresses and enforces permissions.

- **Disabling `CONFIG_PREEMPT_VOLUNTARY` to "make it faster".** If you turn off `CONFIG_PREEMPT_VOLUNTARY`, the build falls back to `CONFIG_PREEMPT_NONE`, which is the server profile (higher latency, higher throughput). On an interactive system this is a regression. Read the help on each Preemption Model option before changing.

30.10 Going deeper

- **`Documentation/kbuild/kconfig-language.rst`**: the canonical Kconfig syntax reference.
- **`Documentation/kbuild/kconfig.rst`**: operational guide for `make menuconfig` and friends.
- **`scripts/kconfig/`**: the Kconfig parser and UI source. Worth a skim if you write your own Kconfig.
- **`Documentation/admin-guide/`**: runtime-tunable options many of which correspond to `.config` choices.

> Next chapter: **Chapter 30A: Kernel lifecycle decision framework.** Configuration choices sit within a longer-lived decision: which source and maintenance track can support the product's field life? **BSP:** Board Support Package: vendor patches, configs, bootloader files, and scripts needed to boot one board.

Chapter 30A: Kernel lifecycle: mainline, stable, LTS, vendor BSPs

> **ABI:** Application Binary Interface: the calling convention, register use, binary format, and library contract that let separately built code run together.

> **What:** a decision framework for one of the most important architectural choices in any Linux-based product: *which* kernel release do we ship? Mainline tip, the latest stable, an LTS, a vendor BSP frozen years ago, or something curated between them? **BSP:** Board Support Package: vendor patches, configs, bootloader files, and scripts needed to boot one board.
>
> **Why:** the choice determines your security-fix cadence, your driver-update cost, your hardware-support range, and your migration burden for the next five-to-ten years. If you choose well, updates ship easily for years. If you choose poorly, three years from now you are backporting six years of CVEs onto a fork no one upstream maintains.
>
> **Focus: maintenance economics**, not features. Whatever you pick, you commit to maintaining the gap between it and what the world ships next. If you cannot maintain it yourself, you pay someone who will.

30A.1 The six release tracks

The kernel has more than one “version”. When someone says “use Linux 6.6”, they could mean any of six related things:

Track	What it is	Maintainer	Release cadence	Lifetime
Mainline	Linus’s tree at `git.kernel.org/torvalds/linux.git`	Linus Torvalds	One “x.y” release every ~9 weeks	“Tip” only; superseded by next release
Stable	Per-release backport tree	Greg Kroah-Hartman	Daily/weekly through ~6 weeks after release	~6 weeks per minor

Track	What it is	Maintainer	Release cadence	Lifetime
LTS	Selected mainline releases extended	Greg KH + Sasha Levin	Weekly tagged releases	**2 years** (regular) or **6 years** (extended)
Vendor BSP	Forked mainline + thousands of vendor patches	NXP / TI / ST / etc.	Vendor-internal cadence	Until the vendor abandons the SoC
Yocto / Buildroot kernel	A curated combination of an LTS + a small layer of fixes	Yocto / Buildroot maintainers	Aligned with Yocto release cadence	Tied to the Yocto release's lifetime
Distribution kernel	Debian, Ubuntu, Fedora kernel packages	Distro maintainers	Distro release cadence	Tied to distro release lifetime

For embedded systems, the three that matter are **Mainline**, **LTS**, and **Vendor BSP**. The other three are either too volatile (mainline tip), too short-lived (stable), or too desktop-oriented (distro) for most embedded products.

30A.2 Mainline

`git.kernel.org/torvalds/linux.git`, the canonical tree. Every kernel feature, fix, and driver ultimately lives here.

Releases: the merge window opens 1 week after each release, accepts new features for 2 weeks, then there are 6-8 weekly `-rcN` candidates before the next `x.y` final. The cycle is roughly:

```
6.5 release  ──────────►  6.6-rc1 (window closes)
                (1 week)    │
                            └─ 8 × rc cycles, ~1 week each ─►  6.6 release  ─►  6.7-rc1 ...
```

In 2025, mainline runs at roughly 6.6 → 6.7 → 6.8 → 6.9 across a calendar year, with each minor having ~9-week cadence.

Lifetime: none. The day 6.6 ships, 6.7-rc1 is already absorbing the next merge window. There is no concept of "support" for plain mainline. If you ship `6.6.0` and there's a CVE in `6.6.1`, you must either (a) take `6.6.1` (the stable update), (b) cherry-pick the fix yourself, or (c) decline the fix.

When to choose mainline:

- **Development.** You're building the next product. You want every recent driver fix and every new feature.
- **Pre-production hardware bring-up.** Mainline is where the i.MX6ULL gained DT bindings, gained mainline EVK support, gained subsequent fixes. Tracking mainline means you can be confident about what works.
- **Demo / proof-of-concept devices.** Lifetime doesn't matter. Recent features do.

When *not* to choose mainline:

- **Shipping product.** A device in a customer's hands needs an upstream commitment longer than 9 weeks. Mainline gives you no such commitment.

30A.3 Stable

`git.kernel.org/stable/linux.git`, branches `linux-6.6.y`, `linux-6.5.y`, etc.

Greg KH manages this tree. Stable releases backport **bug fixes and security patches only** from mainline to a specific x.y. So `linux-6.6.y` accumulates fixes from mainline that apply cleanly (or with light tweaks) to 6.6's codebase.

Releases come out daily-to-weekly with version tags like `v6.6.1`, `v6.6.2`, ..., `v6.6.42`.

Lifetime: historically ~6 weeks per minor (one cycle of mainline). The current minor's stable tree continues until the next minor's stable tree starts. The older one is then EOL'd.

Exception: selected stable trees are promoted to **LTS** (next section) and continue for years.

When to choose stable (non-LTS):

- You want the latest known-good kernel, with all minor-version fixes, and you're going to upgrade to the next minor every 9 weeks anyway. Mostly: development boards, where you regularly rebase.

When *not* to choose stable:

- Shipping product, again. You'll lose support in 6 weeks.

30A.4 Long-Term Support (LTS)

Selected mainline releases are designated **LTS** and get stable-tree backports for years instead of weeks. The current list changes, so always check kernel.org before choosing for a product. As of June 2026, kernel.org lists:

LTS	Released	Projected EOL	Notes
6.18	Nov 2025	Dec 2028	Newest longterm line at this revision.
6.12	Nov 2024	Dec 2028	Strong default for new products if your BSP and drivers are ready.
6.6	Oct 2023	Dec 2027	Good teaching baseline and still common in embedded.
6.1	Dec 2022	Dec 2027	Heavily used by Android and several distros.
5.15	Oct 2021	Dec 2026	Still common, but aging.
5.10	Dec 2020	Dec 2026	Old baseline, mostly for legacy products.

LTS lifetime is the lifetime of the product, in practice. A product shipping in 2026 should normally start from the newest longterm line that supports the hardware and vendor stack. If you ship on an older LTS, write the migration date into the product plan before release.

The “extended LTS” track (sometimes called “Civil Infrastructure Platform” or CIP) backports security fixes for **6+ years** on selected LTS releases, funded by industrial users (Toshiba, Siemens, others). Less drama-prone than mainline LTS but slower-moving.

When to choose LTS:

- **Any shipping product with a multi-year field life.** This is the default answer.
- Cost-sensitive products that can’t afford a dedicated kernel maintainer. LTS gives you free upstream backports.
- Products that need security-fix updates after end-of-development.

When *not* to choose LTS:

- You need a brand-new feature only in mainline that hasn’t been backported.

- Your hardware vendor's BSP is pinned to a non-LTS release and you can't afford to forward-port their patches.

30A.5 Vendor BSPs

Every silicon vendor (NXP, TI, ST, Rockchip, Broadcom, ...) maintains a downstream fork of mainline with their own additions. For i.MX6ULL, that's NXP's `linux-imx` at `github.com/nxp-imx/linux-imx`.

A vendor BSP typically contains:

1. **A pinned mainline base.** NXP's recent linux-imx branches are based on `5.15.71` or `6.6.23`, not on mainline tip.
2. **Vendor patches.** Thousands. Drivers for proprietary silicon, performance tweaks, customer-requested features, security fixes the vendor hasn't yet upstreamed.
3. **Vendor BSP-only drivers.** Drivers that exist *only* in the BSP, never in mainline. Frequently for GPU, video codec, ISP, or other IP blocks the vendor considers a trade secret.

The vendor releases new tags periodically (NXP about quarterly). Old tags get little attention.

The upside. Every peripheral on the vendor's reference board has a working driver. The vendor's tool, Yocto layer (`meta-imx`), demo image, evaluation kit, assumes their BSP.

> **Yocto:** a metadata-driven build system for producing custom Linux distributions.

The downside. Three things you must factor in:

- **Security-fix latency.** The vendor's quarterly cadence means CVEs in your BSP base are typically 1-3 months behind mainline. Some critical fixes never get backported to the vendor's older releases.
- **Mainline drift.** Every quarter, the vendor's tree drifts further from mainline. Any patch you write against the BSP needs porting before it works against mainline. The longer you wait, the harder that port becomes.
- **Vendor abandonment risk.** When NXP stops supporting i.MX6ULL (they will, eventually), `linux-imx` will stop getting updates for that SoC. You're then maintaining the BSP yourself or porting to mainline.

When to choose a vendor BSP:

- **You need a vendor driver that isn't in mainline.** Common for GPUs, hardware codecs, vendor-specific ISP/camera pipelines.

- **Time-to-market is paramount and the product lifecycle is < 3 years.** Vendor BSPs ship fast. If you don't need to maintain it for 10 years, the migration cost may never arrive.
- **You're under contractual pressure to use the vendor's reference design.** This happens.

When *not* to choose a vendor BSP:

- **Multi-year product life and security matters.** The BSP is a long-term liability.
- **Mainline already supports your hardware completely.** For i.MX6ULL with stock peripherals, mainline is fully sufficient. Going vendor adds future cost without benefit.

30A.6 The decision framework

A flowchart-as-text:

```
Question 1: Does your hardware work on mainline?
   ├─ Yes  ─► Question 2
   └─ No   ─► Vendor BSP is forced.  Plan a migration to mainline within 1-2 years.
                 See Chapter 122A for the playbook.

Question 2: What's the product's expected field life?
   ├─ < 1 year         ─► Mainline tip is fine; rebase to next LTS before manufacturing.
   ├─ 1-3 years        ─► Current LTS.
   ├─ 3-7 years        ─► Current LTS, with explicit budget for one LTS migration mid-lifecycle.
   └─ 7+ years         ─► Extended LTS (CIP) or self-maintained LTS branch.

Question 3: Do you have a kernel maintainer on the team?
   ├─ Yes              ─► You can run mainline if Q1 says you can.
   └─ No               ─► Stick with LTS.  Don't try to maintain a fork.
```

For our i.MX6ULL on Point Atom MINI:

- **Q1: Does it work on mainline?** Yes. Full support since v4.10.
- **Q2: Product life?** Assume 5 years for an industrial product.
- **Q3: Maintainer?** This book exists to teach you to be one.

→ **Current LTS, with a mid-lifecycle migration plan.** As of this revision, that usually means 6.12 or 6.18 for a new product, unless your vendor stack requires 6.6. If you choose 6.6, plan the next LTS migration before manufacturing.

30A.7 The 4.1.15 trap

Here is a common scenario. You inherit a vendor BSP pinned to Linux 4.1.15 (NXP's i.MX BSP from 2017). The hardware works and the board boots, so you are tempted to ship it.

What you would actually be shipping:

- **A kernel from 2017.** Five-plus years of CVE backlog. Even cherry-picking critical fixes is a part-time job.
- **A driver model from 2017.** Many subsystems (DT bindings YAML, modern clk framework, regmap-everywhere, devm-_* helpers, etc.) have evolved. Code you write against 4.1.15 doesn't transfer to mainline without rewrite.

> **MCU bridge:** Think of regmap like a typed wrapper around your read_reg() and write_reg() helpers, with caching, locking, and bus differences handled centrally. **regmap:** a kernel helper that wraps register reads and writes over I2C, SPI, or MMIO.

- **Toolchain constraints.** 4.1.15 doesn't build with modern gcc (>= 11) without patches. You're committing to an old gcc too.
- **No upstream support.** Linux 4.1 has been EOL for years. Issues you file get closed as "fixed in 5.x".

The 4.1.15 trap is real. It works *today*. It costs increasingly more *every year*. If you find yourself inheriting one, treat the migration to mainline as a P1 work item, not a P3 "someday". Chapter 122A walks the migration.

The same applies to any sufficiently-old vendor BSP: 4.9, 4.14, 4.19, 5.4, 5.10. Each year past the LTS EOL the situation worsens.

30A.8 Worked example: shipping decisions

Three scenarios:

Scenario A, Consumer IoT smart-light, 3-year product life

- Need: low-power Wi-Fi, BLE, a custom LED-strip driver, OTA updates.
- All hardware supported in mainline (the Wi-Fi via standard mainline drivers, BLE via Bluetooth subsystem, LED strip via SPI).
- **Decision: current LTS.** Pick the newest longterm line that supports the Wi-Fi and BLE stack. Set up an OTA pipeline (Ch 63) so kernel upgrades reach the field.

Scenario B, Industrial PLC, 10-year product life, IEC 61131 certified

- Need: hard real-time (< 100 µs jitter), CAN bus, EtherCAT master, vendor-specific ISP for camera.
- ISP driver only in NXP's BSP. Never upstreamed.

- **Decision: Vendor BSP, for now, with mandatory migration plan.** Track NXP's LTS-aligned branch (6.6-based). Allocate budget in year 3 to begin upstreaming the ISP driver. Aim to be on mainline LTS by year 5. Otherwise: pay an external Linux company to extended-LTS your kernel for the remaining lifespan.

Scenario C, Hobbyist single-board computer kit

- Need: just works out of the box. Users will run their own kernels too.
- **Decision: Mainline tip, refresh per Yocto release.** Hobbyists expect to be current. Nobody's running this kit for 5 years.

The pattern across all three: **the product's field life and the security level required together determine the kernel track.**

30A.9 Key Notes

- A release tag fixes source identity, while a maintained branch supplies an ongoing stream of changes. A teaching baseline and a supported shipping baseline serve different purposes.
- Stable and longterm maintenance primarily carry fixes, not every new feature. Needed hardware support must be assessed in the selected tree rather than inferred from mainline development.
- The product team owns qualification and delivery of updates even when upstream provides backports. A support window does not automatically deliver fixes to deployed devices.
- A vendor BSP can meet a hardware dependency while adding downstream maintenance obligations. Custom drivers, bindings and toolchain constraints contribute to the cost of moving that dependency later.
- Field life can outlast a kernel's maintenance window. Migration or extended support is therefore a lifecycle resource decision, not only a version-number choice.
- The worked shipping cases illustrate different constraints, not fixed recommendations for every product. Dated support tables and MINI hardware claims need current evidence before a release decision.
- Build coverage across source tracks can reveal configuration or API drift, but cannot establish runtime compatibility, security-update safety or board support by itself.

30A.10 Pitfalls

- **"Just stay on 5.4 forever."** 5.4 is already outside the current kernel.org longterm list. That is not forever.

- **“We’ll upgrade the kernel later.”** “Later” never comes if your code is so deeply intertwined with vendor BSP internals that migration is a major rewrite. Budget the migration *now*, even if you delay execution.
- **“LTS = no breaking changes.”** LTS gets bug fixes and security patches. Behavior fixes can change semantics. Test before deploying.
- **“Mainline is unstable.”** Modern mainline is *very* stable. What’s unstable is mainline tip *between releases*. Pick a tagged release (`v6.6` not `master`) and you have what was tested.
- **“We don’t need security fixes. The device isn’t on the internet.”** This is less and less true. Even isolated devices receive USB sticks. Even airgapped devices face supply-chain attacks. Apply the fixes.
- **“Our customer requires the vendor BSP.”** Sometimes true (compliance, IP). Often a habit. Push back on the contractual requirement when the technical justification is weak.

30A.11 Going deeper

- **`kernel.org/category/releases.html`**: current state of every kernel release.
- **`lwn.net/Articles/...`**: weekly Linux Weekly News articles, the best source for keeping current.
- **The `linux-cip` (Civil Infrastructure Platform) project**: extended-LTS for industrial applications.
- **Bootlin’s “Embedded Linux from Scratch” training** (free online materials), covers a similar decision framework.
- **`Documentation/process/stable-kernel-rules.rst`**: what does and doesn’t get backported to stable trees.
- **`drm/kernel-doc-rst` discussions on `linux-rt-users` mailing list** for PREEMPT_RT-specific lifecycle considerations.

> **PREEMPT_RT:** the Linux real-time patch set that makes more kernel paths preemptible and reduces latency.

> Next chapter: **Chapter 30B: Product kernel configs.** Before leaving the kernel, we split development, production, and recovery configs so kernel configuration becomes a maintained product asset. **Buildroot:** a configuration-driven build system that produces a complete root filesystem and related images.

Chapter 30B: Product kernel configs

Chapter 30 explained how kernel configuration works. Chapter 30A explained how to choose and maintain a kernel version.

This chapter is about using configuration in a product. A product rarely has only one kernel config. You usually need at least:

- a development config,
- a production config,
- a recovery config.

They should be related, but not identical.

30B.1 Why one config is not enough

During bring-up, you want many debug features:

- verbose logs,
- debug filesystems,
- extra drivers,
- symbols,
- tracing.

In production, you usually want less:

- smaller image,
- shorter boot time,
- fewer attack paths,
- fewer accidental interfaces.

In recovery, you want a kernel that can boot when the normal system is broken:

- storage driver built in,
- console built in,
- initramfs support,
- enough networking or USB for update,
- fewer optional product features.

Trying to use one `.config` for all three jobs makes the file confusing.

30B.2 Use fragments

Do not hand-edit one huge `.config` forever.

Keep small config fragments in your board repository:

```
kernel-configs/
  base.config
  dev.config
  prod.config
  recovery.config
```

Example `base.config`:

```
CONFIG_ARCH_MXC=y
CONFIG_SOC_IMX6ULL=y
CONFIG_SERIAL_IMX=y
CONFIG_SERIAL_IMX_CONSOLE=y
CONFIG_MMC=y
CONFIG_MMC_SDHCI=y
CONFIG_MMC_SDHCI_ESDHC_IMX=y
CONFIG_EXT4_FS=y
CONFIG_DEVTMPFS=y
CONFIG_DEVTMPFS_MOUNT=y
```

Example `dev.config`:

```
CONFIG_DEBUG_KERNEL=y
CONFIG_DYNAMIC_DEBUG=y
CONFIG_DEBUG_FS=y
CONFIG_KALLSYMS=y
CONFIG_KALLSYMS_ALL=y
CONFIG_FTRACE=y
CONFIG_FUNCTION_TRACER=y
```

Example `prod.config`:

```
# CONFIG_DEBUG_FS is not set
# CONFIG_DYNAMIC_DEBUG is not set
CONFIG_STRICT_KERNEL_RWX=y
CONFIG_STRICT_MODULE_RWX=y
```

Example `recovery.config`:

```
CONFIG_BLK_DEV_INITRD=y
CONFIG_INITRAMFS_SOURCE=""
CONFIG_USB_SUPPORT=y
CONFIG_USB=y
CONFIG_USB_STORAGE=y
```

```
CONFIG_VFAT_FS=y
CONFIG_FAT_FS=y
```

These are examples, not a final product policy. The important idea is that each fragment has a purpose.

30B.3 Build separate output directories

Use one output directory per config.

```
$ cd ~/imx6ull/src/linux-v6.6
$ mkdir -p build-dev build-prod build-recovery
```

Merge the development config:

```
$ ./scripts/kconfig/merge_config.sh \
      -O build-dev \
      arch/arm/configs/imx_v6_v7_defconfig \
      kernel-configs/base.config \
      kernel-configs/dev.config
$ make O=build-dev olddefconfig
```

Merge the production config:

```
$ ./scripts/kconfig/merge_config.sh \
      -O build-prod \
      arch/arm/configs/imx_v6_v7_defconfig \
      kernel-configs/base.config \
      kernel-configs/prod.config
$ make O=build-prod olddefconfig
```

Merge the recovery config:

```
$ ./scripts/kconfig/merge_config.sh \
      -O build-recovery \
      arch/arm/configs/imx_v6_v7_defconfig \
      kernel-configs/base.config \
      kernel-configs/recovery.config
$ make O=build-recovery olddefconfig
```

Then build each one:

```
$ make O=build-dev -j$(nproc) zImage dtbs modules
$ make O=build-prod -j$(nproc) zImage dtbs modules
$ make O=build-recovery -j$(nproc) zImage dtbs
```

The `O=` directory keeps generated files separate. That makes it possible to compare configs and rebuild only the one you need.

30B.4 Built-in or module?

For embedded boot, this decision matters.

Driver	Built-in (=y)	Module (=m)
UART console	Usually yes	No, too late for early boot logs
MMC rootfs driver	Yes if rootfs is on MMC	Only if initramfs can load it
Ethernet	Depends on boot needs	Fine if rootfs can load modules
USB storage recovery	Yes for recovery kernel	Fine for normal product
Optional sensor	Usually no	Good module candidate
Filesystem for rootfs	Yes	Only if initramfs loads it first

A simple rule:

If the kernel needs it before the real root filesystem is mounted, build it in.

30B.5 Compare configs

The kernel tree includes a config comparison tool:

```
$ ./scripts/diffconfig build-dev/.config build-prod/.config
```

Example output:

```
-DEBUG_FS y
-DYNAMIC_DEBUG y
-FTRACE y
 STRICT_KERNEL_RWX y
 STRICT_MODULE_RWX y
```

Use this before release. It catches surprises like debug features left enabled in production.

30B.6 Save a small defconfig

After merging and checking a config, save the minimal form:

```
$ make O=build-prod savedefconfig
$ cp build-prod/defconfig kernel-configs/imx6ull_product_defconfig
```

The full `.config` is generated output. It is useful for builds, but it is noisy in code review.

Commit the small defconfig or fragments. Regenerate the full `.config` during the build.

30B.7 Name the kernel clearly

Set a local version so the running board tells you what it is using.

In a config fragment:

```
CONFIG_LOCALVERSION="-imx6ull-prod"
```

Or from the command line:

```
$ make O=build-prod LOCALVERSION=-imx6ull-prod zImage
```

On the board:

```
# uname -a
Linux imx6ull 6.6.0-imx6ull-prod ...
```

This small detail helps distinguish several kernel variants in the same development environment.

30B.8 Keep bootargs different from config

Kernel config decides what the kernel can do.

Bootargs decide what this boot should do.

Examples:

Setting	Usually belongs in
Enable MMC driver	Kernel config

Setting	Usually belongs in
Select root device	Bootargs
Enable ext4 support	Kernel config
Pass `rootwait`	Bootargs
Enable initramfs support	Kernel config
Pass `rdinit=/init`	Bootargs
Enable dynamic debug support	Kernel config
Turn on one driver’s debug messages	Bootargs or debugfs

Do not rebuild the kernel just to change `root=/dev/mmcblk0p2`. That is a bootargs job.

30B.9 A small CI matrix

Even a small embedded team should build more than one kernel in automation.

Example:

Build	Config	Must prove
`dev`	base + dev	Boots, has debugfs and symbols
`prod`	base + prod	Boots normal rootfs, debug features absent
`recovery`	base + recovery	Boots initramfs recovery menu
`dtbs_check`	board DTS	Device tree still validates

The CI does not need to run on real hardware for every commit, but hardware boot tests should happen before release.

30B.10 Release checklist

Before handing a kernel to manufacturing or customers:

- `uname -a` shows the expected version string.
- The DTB in the boot partition matches the kernel release.
- `scripts/diffconfig` was checked against the previous release.
- Debug-only features are intentionally enabled or disabled.
- Rootfs storage, console, and filesystem drivers are built in when needed.
- Recovery boot was tested, not only normal boot.
- The exact kernel source tag and config fragments are recorded.

30B.11 Key Notes

- Development, production and recovery kernels share hardware needs but serve different operational roles. A diagnostic interface useful in development is not automatically a production requirement.
- A base fragment captures common support, while role fragments express differences. Kconfig resolution determines the final state, so the resulting `.config` matters as well as the requested fragments.
- Separate `O=` directories isolate generated state for each variant. They prevent one role's configuration and artifacts from silently replacing another's.
- Boot-critical drivers must be reachable before the normal rootfs mounts. Modules are viable at that stage only when an earlier environment can provide and load them.
- Kernel configuration defines capability, bootargs select boot policy, and DT defines hardware resources. These interfaces need coordinated releases without being collapsed into one configuration file.
- Local version strings and config diffs expose artifact identity and policy changes. They complement source revision records rather than replacing them.
- A CI build checks a different boundary from normal or recovery boot tests. Manufacturing readiness includes recovery operation, not merely the existence of three successfully built images.

Part IV connects image artifacts, firmware handoff, hardware description, recovery and configuration ownership. Part V examines the user-space environment that depends on those interfaces.

Previous chapter: Chapter 30A - Kernel lifecycle

Next part: Part V - Root filesystem and user space

Chapter 31: A root filesystem, by hand

> **udev:** the user-space device manager that reacts to kernel device events and creates policy-driven /dev nodes.

> **What:** the composition of a small root filesystem: BusyBox, the FHS directory tree, runtime libraries, `/etc/inittab`, `/etc/init.d/rcS`, `/etc/fstab` and an NFS-backed root. **NFS:** Network File System, which lets the target mount a host directory over Ethernet during development.
>
> **Why:** Chapter 29 isolated early user space in an initramfs. This case adds startup configuration, shared-library dependencies and a host-backed development root. These pieces reveal responsibilities that an automated image builder later assembles. **MCU bridge:** Think of the rootfs as the firmware image's file-backed runtime environment. On an MCU you link everything into flash. On Linux, programs and config live in this mounted tree. **rootfs:** root filesystem, the directory tree mounted at / that contains /bin, /etc, /dev, and libraries.
>
> **Focus:** the **handful of files in `/etc/`** that everything else depends on, `inittab`, `init.d/rcS`, `fstab`, `passwd`, `group`. Once you know these files, Buildroot, Yocto, and Ubuntu-base look like the same content generated by tools. **Yocto:** a metadata-driven build system for producing custom Linux distributions. **Buildroot:** a configuration-driven build system that produces a complete root filesystem and related images.

31.1 The FHS in 30 seconds

The Filesystem Hierarchy Standard says what each top-level directory is for. The embedded subset:

```
/               # root directory
├── bin/        # essential user binaries (ls, mv, sh, ...)
├── sbin/       # essential system binaries (init, mount, fsck, ...)
├── usr/        # non-essential user binaries; mirrors bin/ + sbin/ + lib/ structure
│   ├── bin/
│   ├── sbin/
│   └── lib/
├── lib/        # shared libraries that bin/ and sbin/ need
├── etc/        # configuration files (text, no binaries)
├── dev/        # device files (mostly auto-populated by devtmpfs)
├── proc/       # mount point for procfs (Ch 32)
├── sys/        # mount point for sysfs (Ch 32)
├── tmp/        # temporary files; usually tmpfs
├── var/        # variable data: logs, mailboxes, runtime state
│   ├── log/
```

```
│   └── run/
├── root/        # root user's home directory
├── home/        # user homes (often empty on embedded)
└── mnt/         # ad-hoc mount points
```

We create each of these directories. The ones that should contain files at boot (`bin/`, `sbin/`, `lib/`, `etc/`) we populate now. The ones the kernel or daemons fill (`proc/`, `sys/`, `dev/`, `tmp/`, `var/run/`) stay empty until mount time.

31.2 The plan

```
host                                    target
────────────────────────────────────    ──────────────────────────────────

~/imx6ull/rootfs/                       / (mounted via NFS from host)
  ├── bin/busybox + symlinks            same
  ├── sbin/init → /bin/busybox          same
  ├── etc/inittab, rcS, fstab           same
  ├── lib/*.so.* (from cross-toolchain) same
  ├── proc/  (empty; mount point)       proc filesystem mounted here
  ├── sys/   (empty)                    sysfs mounted here
  └── dev/   (empty)                    devtmpfs mounted here

NFS server on host exports ~/imx6ull/rootfs/.
Kernel boots with bootargs: root=/dev/nfs nfsroot=192.168.7.1:...
```

In this arrangement, the target sees a host-exported filesystem as its root. Understanding that relationship does not require setting up an NFS server or building the example.

31.3 Build BusyBox statically

```
$ cd ~/imx6ull/src
$ wget https://busybox.net/downloads/busybox-1.36.1.tar.bz2
$ tar xjf busybox-1.36.1.tar.bz2
$ cd busybox-1.36.1

$ make ARCH=arm CROSS_COMPILE=arm-none-linux-gnueabihf- defconfig
$ make ARCH=arm CROSS_COMPILE=arm-none-linux-gnueabihf- menuconfig
```

In menuconfig, enable two settings explicitly:

```
Settings  --->
    [*]   Build static binary (no shared libs)
    [*]   Enable Unicode support
    [*]     Check $LC_ALL, $LC_CTYPE and $LANG environment variables
```

Build:

```
$ make ARCH=arm CROSS_COMPILE=arm-none-linux-gnueabihf- -j$(nproc)
$ arm-none-linux-gnueabihf-strip busybox
$ ls -lh busybox
-rwxr-xr-x 1 you you 580K busybox
```

580 KB statically linked. (~450 KB with musl-gcc.)

> **Static or dynamic?** Static is simpler to deploy, no libraries needed on the target. The drawback is **DNS resolution over the network doesn't work** with static glibc, because glibc's NSS (Name Service Switch, the pluggable backend behind `gethostbyname` / `getaddrinfo`) is *dlopen'd* at runtime even from a "static" binary. A static glibc binary *can* still resolve names listed in `/etc/hosts`. What it can't do is real DNS over the network. If you need real DNS, build BusyBox dynamically and copy the toolchain libraries into the rootfs. This chapter does both: static for the first boot, dynamic in §31.10.

31.4 Populate the rootfs

```
$ cd ~/imx6ull
$ mkdir -p rootfs/{bin,sbin,etc/init.d,lib,proc,sys,dev,tmp,var/
{log,run},root,home,mnt,usr/{bin,sbin,lib}}
$ cd rootfs

# Install BusyBox into the rootfs
$ make -C ~/imx6ull/src/busybox-1.36.1 \
       ARCH=arm CROSS_COMPILE=arm-none-linux-gnueabihf- \
       CONFIG_PREFIX=$PWD install
```

`make install` reads BusyBox's config and creates symlinks for every applet:

```
$ ls bin/ | head
[
[[
ash
base32
base64
busybox
cat
chattr
chgrp
chmod
...
$ ls bin/ | wc -l
246
```

246 applets, all pointing at `bin/busybox`. The single 580 KB binary acts as 246 commands. Which one it acts as depends on `argv[0]`.

```
$ ls -l sbin/init
lrwxrwxrwx 1 you you 14 ... sbin/init -> ../bin/busybox
```

`sbin/init` → busybox. When the kernel `execs` `/sbin/init`, busybox examines `argv[0]` = `/sbin/init` and acts as init.

31.5 Create `/etc/inittab`

BusyBox init reads `/etc/inittab` to know what to do. Format:

```
<id>:<runlevels>:<action>:<process>
```

For BusyBox, `<id>` is the controlling tty (empty for “use the system default console”), `<runlevels>` is ignored. The action determines when the process runs.

```
$ cat > etc/inittab <<'EOF'
# /etc/inittab — busybox init configuration

# System initialization. Runs once at boot, before anything else.
::sysinit:/etc/init.d/rcS

# Spawn a shell on the console. respawn = restart automatically when it exits.
console::respawn:-/bin/sh

# What to do on Ctrl-Alt-Del. (No keyboard? Harmless to keep.)
::ctrlaltdel:/sbin/reboot

# What to do on shutdown.
::shutdown:/bin/umount -a -r
::shutdown:/sbin/swapoff -a
EOF
```

The leading `-` on `-/bin/sh` makes it a login shell (reads `/etc/profile` and `~/.profile`). The eight standard BusyBox init actions:

Action	When the process runs
`sysinit`	Once, at boot, before anything else
`wait`	Once, init waits for it to finish before continuing
`once`	Once, init does not wait
`respawn`	Started; whenever it exits, restart it

Action	When the process runs
`askfirst`	Like respawn, but prints “Please press Enter to activate” and waits
`restart`	When init receives `SIGHUP` (i.e. asked to re-read inittab)
`ctrlaltdel`	When init receives `SIGINT` (Ctrl-Alt-Del on a real keyboard)
`shutdown`	On halt/poweroff/reboot

For a development board you typically need only `sysinit`, `respawn`, and `shutdown`.

31.6 Create `/etc/init.d/rcS`

The shell script `/etc/inittab`’s `sysinit` line runs. This is where most of the per-boot setup lives:

```
$ cat > etc/init.d/rcS <<'EOF'
#!/bin/sh

# Search paths for the rest of this session.
PATH=/sbin:/bin:/usr/sbin:/usr/bin
LD_LIBRARY_PATH=/lib:/usr/lib
export PATH LD_LIBRARY_PATH

# Mount the virtual filesystems described in /etc/fstab.
/bin/mount -a

# devtmpfs is mounted automatically by the kernel if CONFIG_DEVTMPFS_MOUNT=y.
# If yours isn't, mount it manually:
mountpoint -q /dev || /bin/mount -t devtmpfs none /dev

# devpts gives us pseudo-terminals for things like ssh, screen, etc.
mkdir -p /dev/pts
/bin/mount -t devpts none /dev/pts

# Hook mdev into the kernel's hotplug mechanism. (mdev is BusyBox's udev.)
echo /sbin/mdev > /proc/sys/kernel/hotplug

# Populate /dev/ from /sys/ for devices already enumerated before mdev started.
/sbin/mdev -s

# Set a hostname.
/bin/hostname pa-mini
```

```
# Start any background daemons here (sshd, ntpd, your app, ...).

echo "*** rootfs ready ***"
EOF
$ chmod +x etc/init.d/rcS
```

The `chmod +x` matters. Without execute permission, BusyBox init skips the script silently. No `/proc`, no `/sys`, no `/dev/pts`. This is the most common first-time-rootfs bug.

31.7 Create `/etc/fstab`

`mount -a` consults `/etc/fstab` to know what to mount. Five-column format:

```
<device> <mountpoint> <fs-type> <options> <dump-freq> <fsck-order>
```

```
$ cat > etc/fstab <<'EOF'
# device      mountpoint    type      options     dump  pass
proc          /proc         proc      defaults    0     0
sysfs         /sys          sysfs     defaults    0     0
tmpfs         /tmp          tmpfs     defaults    0     0
tmpfs         /run          tmpfs     defaults    0     0
EOF
```

Four virtual filesystems:

- **`proc`**: the procfs (Ch 32) at `/proc`.
- **`sysfs`**: the sysfs (Ch 32) at `/sys`.
- **`tmpfs` at `/tmp`**: RAM-backed tmpfs for `/tmp`. Wipes on reboot, which is what `/tmp` is supposed to do.
- **`tmpfs` at `/run`**: RAM-backed tmpfs for daemon PID/socket files.

No physical disk mounts, the rootfs *itself* is mounted by the kernel before this script runs, via the `root=` cmdline. Other partitions (data, log) would be additional `/etc/fstab` lines.

31.8 Create `/etc/passwd` and `/etc/group`

Even a single-user system needs minimal versions:

```
$ cat > etc/passwd <<'EOF'
root:x:0:0:root:/root:/bin/sh
EOF
$ cat > etc/group <<'EOF'
root:x:0:
EOF
$ cat > etc/shadow <<'EOF'
root::0:0:99999:7:::
```

```
EOF
$ chmod 600 etc/shadow
```

`root::` (empty password field in shadow) means `root` logs in with no password. Insecure. Fine for an early dev image. Replace with a real hash before production.

31.9 Create `/etc/profile`

Read by login shells. Sets the prompt, default permissions, etc.:

```
$ cat > etc/profile <<'EOF'
# /etc/profile — login-shell configuration

export PATH=/sbin:/bin:/usr/sbin:/usr/bin
export LD_LIBRARY_PATH=/lib:/usr/lib

umask 022

# A nicer prompt: [user@host:cwd]#
PS1='[\u@\h:\w]\$ '
export PS1

# Useful aliases.
alias ll='ls -lh'
alias la='ls -la'
alias ..='cd ..'
EOF
```

After login, you see `[root@pa-mini:~]#` instead of just `#`. Small but pays back the first time you have two ssh sessions open.

31.10 Copy libraries (for dynamic binaries)

The static BusyBox example in §31.3 avoids ordinary shared-library dependencies. A dynamically linked application, `ssh` or another tool needs the shared libraries present at the runtime paths the loader expects.

The libraries live inside your cross-toolchain installation:

```
$ . ~/imx6ull/scripts/env.sh
$ TOOLCHAIN=$ARM_LINUX_TOOLCHAIN
$ cd ~/imx6ull/rootfs

# The "main" libraries (libc, libm, libpthread, libdl, ...) and the dynamic linker
$ cp -d $TOOLCHAIN/arm-none-linux-gnueabihf/libc/lib/*.so* lib/
$ cp -d $TOOLCHAIN/arm-none-linux-gnueabihf/libc/lib/*.a lib/

# The dynamic linker itself MUST be the real file, not a symlink.
$ rm lib/ld-linux-armhf.so.3
$ cp $TOOLCHAIN/arm-none-linux-gnueabihf/libc/lib/ld-linux-armhf.so.3 lib/
```

```
# usr/lib for less-common libraries
$ cp -d $TOOLCHAIN/arm-none-linux-gnueabihf/libc/usr/lib/*.so* usr/lib/
```

Two things to know:

1. **`-d` preserves symlinks.** Most `.so.N` files are symlinks to `.so.N.M`. Without `-d`, `cp` follows the symlink and copies the target. Both files end up as identical full copies. With `-d`, the symlink stays a symlink and you save space.
2. **`ld-linux-armhf.so.3` is the dynamic linker itself.** In the toolchain it's a symlink to the real `ld-2.31.so`. On the target it *must* be a real file at the path the ELF binaries' INTERP section points to (`/lib/ld-linux-armhf.so.3`). The `rm` + `cp` (without `-d`) dance after the wildcard copy forces the real file.

> **ELF:** Executable and Linkable Format, the standard Linux object and executable file format.

Total size: ~60 MB unstripped for glibc, ~5–10 MB after `strip` and removing the locales / NSS modules you don't need. ~5 MB for a comparable musl install. The glibc bulk is mostly NSS modules, locale data, and unused libraries, all of which you'd strip in a production build.

31.11 Export over NFS, boot

Add to `/etc/exports` on the host:

```
/home/you/imx6ull/rootfs *(rw,sync,no_root_squash,no_subtree_check)
```

Restart NFS:

```
$ sudo exportfs -ar
$ showmount -e localhost
Export list for localhost:
/home/you/imx6ull/rootfs *
```

In U-Boot:

> **MCU bridge:** Think of U-Boot like a much larger boot stub plus debug monitor: it initializes hardware, loads the next image, and gives you commands before Linux starts. **U-Boot:** the bootloader that initializes enough hardware to load and start the Linux kernel.

```
=> setenv bootargs 'console=ttymxc0,115200 earlycon \
                    root=/dev/nfs nfsroot=192.168.7.1:/home/you/imx6ull/
```

```
rootfs,vers=3,nolock,tcp \
                    ip=192.168.7.2:192.168.7.1:192.168.7.1:255.255.255.0::eth0:off \
                    rw rootwait'
=> setenv bootcmd 'tftp 0x82000000 zImage && tftp 0x83000000 imx6ull.dtb && bootz
0x82000000 - 0x83000000'
=> saveenv
=> boot
```

After kernel boot:

```
[    3.412345] VFS: Mounted root (nfs filesystem) on device 0:16.
[    3.428901] Run /sbin/init as init process
*** rootfs ready ***

Please press Enter to activate this console.

[Enter]

[root@pa-mini:~]#
```

The example's shell uses `/` provided by the host over NFS, rather than a local root partition.

31.12 The development loop

Now the win:

```
# On the host:
$ echo 'echo "hello from the host"' > ~/imx6ull/rootfs/usr/bin/hello
$ chmod +x ~/imx6ull/rootfs/usr/bin/hello

# On the target — no reboot:
[root@pa-mini:~]# hello
hello from the host
```

The host's filesystem *is* the target's filesystem. You edit, the target sees. This is what makes embedded Linux iteration feel reasonable.

For installing new kernel modules:

```
# On the host after a kernel rebuild:
$ make -C ~/imx6ull/src/linux INSTALL_MOD_PATH=~/imx6ull/rootfs modules_install

# On the target (no reboot for already-loaded modules; modprobe sees them):
[root@pa-mini:~]# modprobe my-driver
```

31.13 Verify everything

A sanity check sweep:

```
[root@pa-mini:~]# uname -a
Linux pa-mini 6.6.0 #1 SMP ... armv7l GNU/Linux

[root@pa-mini:~]# cat /proc/cpuinfo | head -5
processor   : 0
model name  : ARMv7 Processor rev 5 (v7l)
BogoMIPS    : 24.00
Features    : half thumb fastmult vfp edsp neon vfpv3 tls vfpv4 idiva idivt vfpd32 ...

[root@pa-mini:~]# ls /dev/ | head
console
fb0
mem
mmcblk0
null
random
ttymxc0
urandom
zero

[root@pa-mini:~]# mount
/dev/nfs on / type nfs (rw,...)
proc on /proc type proc (rw,...)
sysfs on /sys type sysfs (rw,...)
tmpfs on /tmp type tmpfs (rw,...)
devtmpfs on /dev type devtmpfs (rw,...)

[root@pa-mini:~]# free -h
              total        used        free      shared
Mem:           448M         12M        434M          0
```

All five check out: kernel running, CPU detected, devices enumerated, mounts active, memory roughly what we expect (512 MiB total minus kernel reservation).

31.14 Key Notes

- A rootfs supplies executables, libraries and configuration. Directories such as `/proc`, `/sys` and `/dev` are mount points whose runtime contents are not produced by copying ordinary files.
- BusyBox selects applets through invocation names, so its binary, enabled applets and installed links jointly define the available commands and init behavior.
- `inittab` selects startup and respawn actions, `rcS` performs per-boot setup, and `fstab` describes mounts. Their responsibilities are related but not interchangeable.
- Account files and file modes define access policy. An empty root password and an unrestricted development shell are not production authentication designs.
- Dynamic executables depend on a matching target loader and libraries, not the host’s library tree. Copying a binary alone does not establish a usable runtime.

- NFS makes the host-exported tree part of the target's runtime dependency. Export permissions, Linux networking and persistent host configuration remain independent concerns.
- Storage location determines persistence: RAM-backed `/tmp` and `/run` disappear at reboot, while data on an NFS export belongs to the host-backed tree. Paths alone do not guarantee durable logging.

31.15 Pitfalls

- **Forgetting `chmod +x` on `rcS`.** Silent failure. Nothing in `/proc` or `/sys`.
- **`/sbin/init` not pointing at BusyBox.** Kernel finds no init, kernel panics, you scratch your head. Verify with `ls -l rootfs/sbin/init`.
- **`/etc/fstab` has tabs vs spaces inconsistency.** `mount -a` is lenient but some tools aren't. Use either consistently.
- **NFS `root_squash`.** Without `no_root_squash`, the target's root user maps to nobody and `chmod 600 /etc/shadow` fails. The export option matters for embedded development.
- **`/dev/console` missing.** If `CONFIG_DEVTMPFS_MOUNT` is off and `rcS` doesn't manually mount devtmpfs, the *console itself* may be missing and `kernel_init` prints a warning. Always keep devtmpfs auto-mounted unless you have a specific reason.
- **Library version mismatch.** If your cross-toolchain uses glibc 2.31 but your application was built against glibc 2.34, the application's required symbols (`memcpy@GLIBC_2.34`) won't resolve. Fix: rebuild the app against the rootfs's libc, or upgrade the rootfs's libc to match the app.
- **NFS over Wi-Fi.** Sometimes works. Often drops packets and freezes the target. *Always* NFS over wired Ethernet.

31.16 Going deeper

- **The FHS specification** at `refspecs.linuxfoundation.org/FHS_3.0/`. Concise and prescriptive.
- **BusyBox manual** at `busybox.net/about.html` and the per-applet `--help`.
- **`/etc/inittab` documentation** in `examples/inittab` inside the BusyBox source tree.
- **`man 8 mount`**, **`man 5 fstab`**, **`man 5 inittab`**, the canonical references.
- **The `nss-musl` and `nss-static`** trick, if you really need static binaries AND DNS, musl static + a simple resolver works where glibc static doesn't.

> Next chapter: **Chapter 32: /proc, /sys, devtmpfs.** These virtual filesystems expose kernel state and device interfaces within the user-space filesystem view.

Chapter 32: /proc, /sys, devtmpfs

> **Privilege boundary:** $ means normal user. # or sudo means root and can change host or target state. After a privileged command, verify the expected device, service, or file appears before continuing. Roll back by undoing the config change or stopping the service you just enabled.

> **What:** the three virtual filesystems through which user space sees and pokes the kernel, `procfs` (process & system info), `sysfs` (the modern device model), and `devtmpfs` (device nodes). Each is RAM-backed and populated by the kernel. **sysfs:** a kernel-generated filesystem under /sys that exposes devices, drivers, and attributes.
>
> **Why:** every later chapter pokes `/proc` or `/sys` somewhere, to read a sensor, to set a GPIO, to inspect a driver. Knowing which virtual filesystem holds what is what makes the difference between following a tutorial and debugging an unfamiliar problem. **MCU bridge:** Think of Linux GPIO like the same pin set/reset block you used on STM32, but accessed through a kernel subsystem that owns numbering, direction, interrupts, and user-space exposure. **GPIO:** General-Purpose Input/Output, a pin controlled as a digital input, output, or interrupt source.
>
> **Focus: the file-as-interface pattern.** In Unix everything is a file. The kernel takes that literally. `cat /proc/cpuinfo` reads CPU info. `echo 1 > /sys/class/leds/led0/brightness` turns on an LED. `cat /proc/interrupts` shows IRQ counts. Once you know this idiom, a lot of debugging needs no code. **MCU bridge:** Think of an IRQ like an EXTI/NVIC interrupt path, except Linux splits the hard interrupt from deferred work and must share lines across drivers. **IRQ:** interrupt request, the signal path that tells the CPU or interrupt controller that hardware needs service.

32.1 Three virtual filesystems, three jobs

Filesystem	Mount point	Source of content	Best at
procfs	`/proc`	Process info + miscellaneous kernel data	Per-process state; legacy kernel knobs
sysfs	`/sys`	The kernel's device model (drivers, buses, devices)	Modern device introspection and control

Filesystem	Mount point	Source of content	Best at
devtmpfs	`/dev`	Device nodes auto-created by the kernel	Actually reading from / writing to devices

All three are **virtual**, RAM-backed, no physical storage. They are populated by the kernel on the fly. Every entry corresponds to a kernel data structure. Writing to a file usually calls a kernel callback that parses the bytes.

32.2 procfs, the process FS that grew

`/proc` was originally a way for `ps` to list processes. Every running process gets a directory named for its PID. That part is unchanged in 30 years:

```
[root@pa-mini:~]# ls /proc/
1   116  2    33   45   685  9          consoles    cpuinfo   ...
```

Numbered directories are processes. `1` is `init`. `2` is `kthreadd`. Later entries are everything else.

Inside a process directory:

```
[root@pa-mini:~]# ls /proc/1/
cmdline      comm         cwd          environ      exe          fd/
maps         mem          mountinfo    mounts       net/         oom_score
root         stat         statm        status       task/        wchan
```

File	Contents
`cmdline`	The args the process was exec'd with, null-separated. `cat /proc/1/cmdline` → `/sbin/init`
`comm`	Just the program's short name
`exe`	Symlink to the binary on disk: `ls -l /proc/1/exe` → `/sbin/init`
`cwd`	Symlink to current working directory
`root`	Symlink to the process's root (changes if `chroot`'d)

File	Contents
`environ`	Environment variables, null-separated
`fd/`	Directory of symlinks for each open file descriptor
`maps`	The process's memory map, every region with addresses and permissions
`mem`	The process's address space, as a file (mostly inaccessible without `ptrace`)
`status`	Human-readable summary: state, uid, memory, signals
`stat`	Machine-readable: 50+ fields used by tools like `top`

Useful command-line idioms:

```
# How many open file descriptors does init have?
[root@pa-mini:~]# ls /proc/1/fd | wc -l
8

# What environment did init inherit from the kernel?
[root@pa-mini:~]# cat /proc/1/environ | tr '\0' '\n'
HOME=/
TERM=linux

# What's the memory layout of busybox?
[root@pa-mini:~]# cat /proc/$(pidof busybox)/maps | head
00010000-00050000 r-xp 00000000 00:00 1234567 /bin/busybox
0005f000-00060000 r--p 0003f000 00:00 1234567 /bin/busybox
00060000-00061000 rw-p 00040000 00:00 1234567 /bin/busybox
...
```

The non-PID entries in `/proc/` are global system info:

Path	Contents
`/proc/cpuinfo`	CPU model, features, BogoMIPS

Path	Contents
`/proc/meminfo`	Total/free/buffers/cached memory
`/proc/loadavg`	Load averages: 1-min, 5-min, 15-min
`/proc/uptime`	System uptime in seconds (since boot)
`/proc/version`	The full `linux_banner` string
`/proc/cmdline`	The bootargs the kernel received (echo of what `chosen.bootargs` from DT carried)
`/proc/interrupts`	Per-IRQ counts. Useful for `is my IRQ firing?`
`/proc/iomem`	The physical memory map: kernel code, kernel data, peripherals
`/proc/devices`	Registered character and block major numbers
`/proc/filesystems`	Filesystems the kernel knows about (loadable + built-in)
`/proc/mounts`	Live mount table (newer than `/etc/mtab`)
`/proc/modules`	Currently loaded kernel modules
`/proc/sys/`	A whole subtree of *tunable* kernel parameters (more below)
`/proc/<tid>/`	Per-thread variants (a thread is a task whose pid != tgid)

`/proc/sys/` is special, most files there are **writable** and tweak kernel behavior live:

```
# Read the current value
[root@pa-mini:~]# cat /proc/sys/kernel/hostname
pa-mini
```

```
# Change it
[root@pa-mini:~]# echo new-hostname > /proc/sys/kernel/hostname

# Lots of these. The full set is huge.
[root@pa-mini:~]# ls /proc/sys/
abi/  debug/  dev/  fs/  kernel/  net/  user/  vm/
```

Most production code uses `sysctl` (a wrapper) instead of writing to `/proc/sys/` directly:

```
[root@pa-mini:~]# sysctl -a | head
kernel.hostname = pa-mini
kernel.osrelease = 6.6.0
...
[root@pa-mini:~]# sysctl -w net.ipv4.ip_forward=1
```

The values are the same. `sysctl` adds value-validation and persistence support via `/etc/sysctl.conf`.

32.3 sysfs, the modern device model

`/sys` is newer (kernel 2.5/2.6, ~2003). It exposes the **kernel device model**, every `struct device`, `struct device_driver`, `struct bus_type`, `struct class`, etc., as a directory tree.

Top-level layout:

```
[root@pa-mini:~]# ls /sys/
block/   bus/      class/    dev/    devices/   firmware/   fs/   hypervisor/
kernel/  module/   power/
```

Each is a different *view* of the same underlying graph:

- **`/sys/devices/`**: the master tree. Every device the kernel knows about lives here exactly once, organized by physical topology (which bus is on which controller is on which CPU complex). Hierarchical and verbose.
- **`/sys/bus/`**: devices grouped by *bus type* (`i2c`, `spi`, `platform`, `usb`, …). Each bus lists its devices and the drivers bound to them.
- **`/sys/class/`**: devices grouped by *function* (`leds`, `gpio`, `tty`, `net`, …). Best for “I want all the LEDs” or “all the network interfaces.”
- **`/sys/block/`**: block devices (SD card, eMMC, USB sticks).
- **`/sys/dev/`**: devices indexed by `major:minor` number.
- **`/sys/module/`**: every loaded kernel module, with its parameters.
- **`/sys/kernel/`**: kernel-internal stuff (security, slab, debug).

- **`/sys/firmware/`**: info from firmware (the DT lives here as `/sys/firmware/devicetree/base/`).

Walking a device

The on-chip ADC (Chapter 49 will be all about this. Here is a teaser):

```
[root@pa-mini:~]# ls /sys/bus/iio/devices/
iio:device0

[root@pa-mini:~]# ls /sys/bus/iio/devices/iio:device0/
dev               in_voltage1_raw    name        of_node        scan_elements
in_voltage0_raw   in_voltage_scale   power       subsystem      uevent

[root@pa-mini:~]# cat /sys/bus/iio/devices/iio:device0/name
2198000.adc

[root@pa-mini:~]# cat /sys/bus/iio/devices/iio:device0/in_voltage_scale
0.806884765

[root@pa-mini:~]# cat /sys/bus/iio/devices/iio:device0/in_voltage0_raw
1453
```

That's the ADC reading channel 0 as a raw 12-bit value, with a known scale factor to convert to volts: `1453 × 0.806884765 / 1000 = 1.172 V`. No code. One `cat`.

Controlling a device

GPIO LEDs:

```
[root@pa-mini:~]# ls /sys/class/leds/
led0  mmc0::

[root@pa-mini:~]# cat /sys/class/leds/led0/brightness
0
[root@pa-mini:~]# cat /sys/class/leds/led0/max_brightness
1
[root@pa-mini:~]# echo 1 > /sys/class/leds/led0/brightness
# LED lights up
[root@pa-mini:~]# echo 0 > /sys/class/leds/led0/brightness
# LED turns off
```

Other useful patterns to look up:

```
[root@pa-mini:~]# ls /sys/class/net/
eth0  lo

[root@pa-mini:~]# cat /sys/class/net/eth0/address
00:04:9f:01:30:ad

[root@pa-mini:~]# cat /sys/class/net/eth0/operstate
up
```

```
[root@pa-mini:~]# echo down > /sys/class/net/eth0/operstate
# eth0 brought administratively down
```

Device-tree introspection

The currently-active DT lives at `/sys/firmware/devicetree/base/`:

```
[root@pa-mini:~]# ls /sys/firmware/devicetree/base/
aliases/             chosen/        cpus/        firmware/    interrupt-
controller@a01000/
clocks/              compatible     model        soc/         ...

[root@pa-mini:~]# cat /sys/firmware/devicetree/base/model
Freescale i.MX6 ULL 14x14 EVK Board

[root@pa-mini:~]# cat /sys/firmware/devicetree/base/soc/aips-bus@2000000/serial@2020000/
compatible
fsl,imx6ul-uartfsl,imx6q-uartfsl,imx21-uart
```

Each DT property is a file. Each node is a directory. Mirrors the DT structure exactly.

32.4 devtmpfs, where device nodes live

Historically, `/dev/` was a regular directory populated at install time with `mknod` calls. Every device that *might* exist had a static node. Modern Linux replaces this with **`devtmpfs`**, a tmpfs that the kernel auto-populates whenever a device probes.

Enable in kernel `.config`:

```
CONFIG_DEVTMPFS=y
CONFIG_DEVTMPFS_MOUNT=y
```

With both set, the kernel mounts devtmpfs at `/dev` early in boot, before init runs. Every `device_create()` or platform-device probe with a `class` adds a node.

What you see:

```
[root@pa-mini:~]# ls /dev/
console     mtd0          mtdblock0   ram0          ttymxc0     ttymxc4
fb0         mtd0ro        null        random        ttymxc1     urandom
i2c-0       mtd1          ptmx        rtc0          ttymxc2     vcs
i2c-1       mtd1ro        pts/        snd/          ttymxc3     vcs1
[root@pa-mini:~]# ls -l /dev/console
crw------- 1 root root 5, 1 Jan 22 12:34 /dev/console
```

The `c` at the start of the permissions means **character device**. `5, 1` is the major and minor number. The kernel routes any open/read/write on this file to the driver registered with that major number.

When a USB device gets plugged in, devtmpfs *immediately* gains entries for it:

```
# Before plugging in a USB stick
[root@pa-mini:~]# ls /dev/sd* 2>/dev/null
[root@pa-mini:~]#
# (Plug in USB stick)
[    142.512345] usb 1-1: new high-speed USB device number 5
[    142.789012] sd 0:0:0:0: [sda] 30277632 512-byte logical blocks
[    142.812345] sda: sda1
[root@pa-mini:~]# ls /dev/sd*
/dev/sda  /dev/sda1
```

No user-space help needed. The kernel did all of it.

What devtmpfs *doesn't* do

It creates device *nodes*, not the metadata around them. For things like:

- Setting permissions per-device (e.g., `/dev/i2c-1` should be group-readable by users in `dialout`)
- Creating symlinks (e.g., `/dev/serial/by-id/usb-FTDI...`)
- Running scripts when a device appears (e.g., auto-mount a USB stick)

…you need either **`udev`** (full-featured but heavy) or **`mdev`** (BusyBox's tiny alternative).

> **udev:** the user-space device manager that reacts to kernel device events and creates policy-driven /dev nodes.

`mdev`, BusyBox's user-space helper

Chapter 31's `rcS` example uses this interface:

```
echo /sbin/mdev > /proc/sys/kernel/hotplug
/sbin/mdev -s
```

The first line registers mdev as the kernel's hotplug agent: whenever a device appears or disappears, the kernel `fork`+`exec`s `/sbin/mdev` with environment variables describing the event. Mdev then applies rules from `/etc/mdev.conf`.

The second line says "scan `/sys` for everything that already exists and create any missing nodes", useful right after boot for devices that were enumerated before mdev was wired up.

A minimal `/etc/mdev.conf`:

```
# pattern             user:group  mode  command

# Default: root:root, 0660.
```

```
# Make audio nodes group-writable so members of "audio" can play sound.
snd/[!c].*             root:audio  0660

# I²C buses readable by "i2c" group.
i2c-[0-9]+             root:i2c    0660

# Run a script when an SD card appears or vanishes.
mmcblk[0-9]p[0-9]      root:root   0660  @/etc/mdev/auto-mount.sh
```

The `@` prefix runs the command *after* the node is created (`$` runs *before*. `*` both).

32.5 Key Notes

- procfs exposes process and system state, sysfs exposes the device model, and devtmpfs supplies device nodes. Their common file-shaped interface does not give them the same purpose.
- A virtual-file read obtains kernel-generated information. A permitted write can invoke a driver or kernel callback, so it is an operation on live state rather than an ordinary text-file edit.
- `/proc/<pid>/maps` describes address-space layout, while `/proc/<pid>/mem` represents memory bytes with stronger access controls. A readable process directory does not authorize unrestricted memory access.
- sysfs bus, class and topology paths are views of related kernel objects. Device identity and documented attributes matter more than incidental enumeration numbers.
- Device nodes connect major/minor numbers to driver interfaces. mdev or udev adds permission, naming and event policy beyond devtmpfs's basic node creation.
- Runtime Device Tree properties show the description Linux received, not proof of the circuit attached to it. Likewise, an ADC raw value needs its documented scale and electrical context before it represents a voltage.
- Mount availability, driver registration and access permissions are separate failure boundaries. A missing or unwritable path cannot be diagnosed from its filename alone.

32.6 Pitfalls

- **`/proc/` writes that don't take effect.** Some `/proc/sys/` entries are read-only on certain configurations. Symptom: `echo 1 > /proc/sys/...` succeeds, but `cat` still shows the old value. Use `sysctl -w` and check the return code.

- **`sysfs` attribute file write that hangs.** If a `store` callback in the driver does something blocking (e.g., reset the chip), the `echo` shell command appears to hang. It's not hung, it's waiting for the kernel callback to complete. Normal.
- **devtmpfs not mounted.** Without `CONFIG_DEVTMPFS_MOUNT=y`, the kernel doesn't auto-mount at boot. Either set that config or mount manually in early `rcS`. Otherwise `/dev/console` may not exist and you get the dreaded "Warning: unable to open an initial console" message.
- **Confusing `/proc/<pid>/mem` with `/proc/<pid>/maps`.** `maps` is the *layout* (text, addresses, permissions). `mem` is the raw bytes. Reading `mem` without `ptrace` is usually denied.
- **`/proc/sys/kernel/hotplug` overwritten.** If you `echo /sbin/mdev > /proc/sys/kernel/hotplug` and later run something that sets it to something else (rare but possible), mdev stops working. Check the file's value at runtime.
- **Forgetting that `/sys` paths are case-sensitive.** `/sys/class/Leds/` won't work. It's `leds`.
- **Sysfs path stability assumptions.** Don't hard-code paths under `/sys/devices/`. They rename across kernel versions. Use `/sys/class/...` in scripts.

32.7 Going deeper

- **`Documentation/filesystems/proc.rst`**: comprehensive procfs reference.
- **`Documentation/filesystems/sysfs.rst`** and `Documentation/driver-api/driver-model/`.
- **`Documentation/admin-guide/sysctl/`**: all the `/proc/sys/` knobs.
- **`man 5 proc`**: concise but complete procfs guide.
- **`man udev`** and **`man udevadm`**, when you outgrow mdev. Most embedded systems can stick with mdev.
- **The `iio_utils` user-space tools**: for richer ADC/sensor interaction than `cat`/`echo`.

> Next chapter: **Chapter 33: Init systems.** With sysfs/proc/devtmpfs understood, we look at PID 1's job in detail and compare BusyBox init, sysvinit, and systemd.

Chapter 33: Init systems

> **What:** PID 1, what it does, what it should do, and the three real choices for an embedded Linux system: BusyBox `init` (tiny, traditional), `sysvinit` (the classical desktop init from the 90s), and `systemd` (the modern service manager that runs on basically every desktop distro).
>
> **Why:** PID 1 is special: the kernel panics if it dies, and every other process on the system descends from it. The choice you make here determines how you write boot scripts, how you crash-restart services, how logs are collected, and how much disk and RAM the system uses just to "be up."
>
> **Focus:** the **trade-off triangle**: simplicity, capability, and footprint. BusyBox wins on simplicity and footprint. Systemd wins on capability. Sysvinit is the historical middle. For most embedded products in 2025, BusyBox init is the right answer. Knowing *why* is the goal of this chapter.

33.1 What PID 1 actually does

The minimum PID-1 job description:

1. **Run startup scripts** (mount filesystems, configure network, start daemons).
2. **Spawn login prompts** (one per console / serial line).
3. **Reap zombies.** When a process's parent dies, its children get reparented to PID 1. When *those* children eventually exit, PID 1 must `wait()` for them or they become zombies forever.
4. **Handle signals.** `SIGTERM` → shut down cleanly. `SIGINT` (Ctrl-Alt-Del on a physical keyboard) → reboot or do whatever the policy says.
5. **Coordinate shutdown.** Stop services in reverse start order, unmount filesystems, sync disks, then `reboot()` or `poweroff()`.

That's it. Any program that does these five things is a legitimate PID 1. The choice is *how* much beyond this it does.

33.2 BusyBox init

The examples since Chapter 29 use BusyBox init. It is **~1500 lines of C**, included in the BusyBox binary as one of its applets. Its configuration is `/etc/inittab` (Chapter 31 §31.5).

Features it has:

- `inittab`-based: `sysinit`, `respawn`, `askfirst`, `once`, `wait`, `shutdown`, etc. (the 8 actions from Ch 31)
- Reads `/etc/inittab` once at boot. Re-reads on SIGHUP
- Reaps zombies. Signals shutdown properly

Features it does *not* have:

- No service dependency tracking. If service A needs service B running, you have to encode that in shell scripts yourself.
- No automatic restart count limit. If a service crashes 1000 times in 1 second, busybox-init dutifully restarts it 1000 times.
- No socket activation. (Systemd's headline feature. Nice to have for embedded? Rarely.)
- No structured logging. `printf` to the console. That's it.
- No cgroup-based resource isolation per service.

When to choose BusyBox init:

- Single-purpose appliance. The system runs one or two daemons + a shell.
- Boot time matters. BusyBox init starts within 100 ms of `kernel_init`.
- RAM and flash budget are tight. The init itself adds 0 bytes (it's inside busybox).
- You want to *understand* every line that runs at boot. You can read all the init code in an hour.

For most i.MX6ULL-class embedded products, this is the right answer.

33.3 sysvinit

The classical Unix init from the 80s/90s. Still around, still on some Debian systems if you uninstall systemd.

The model:

- `/etc/inittab` (different syntax from BusyBox. Runs `/etc/rc.d/rc <N>` for each runlevel)
- `/etc/init.d/`: one shell script per service. Each script accepts `start`, `stop`, `restart`, `status` as arguments.
- `/etc/rc<N>.d/`: symlinks to `/etc/init.d/` scripts, named `S<NN><name>` (start) or `K<NN><name>` (kill). Init runs them in numeric order when entering runlevel N.

Runlevels: a number 0-6 representing system states. By convention:

- 0 = halt
- 1 = single-user maintenance
- 2 = multi-user, no networking (rarely used)
- 3 = multi-user, networking, text console
- 5 = multi-user, networking, graphical login
- 6 = reboot

You change runlevel with `init <N>` or `telinit <N>`. The current level is in `/var/run/utmp`.

A sysvinit service script:

```
#!/bin/sh
### BEGIN INIT INFO
# Provides:          my-daemon
# Required-Start:    $network
# Required-Stop:     $network
# Default-Start:     2 3 4 5
# Default-Stop:      0 1 6
# Short-Description: My daemon
### END INIT INFO

case "$1" in
    start)
        start-stop-daemon --start --background --make-pidfile \
            --pidfile /var/run/my-daemon.pid \
            --exec /usr/bin/my-daemon
        ;;
    stop)
        start-stop-daemon --stop --pidfile /var/run/my-daemon.pid
        ;;
    restart) $0 stop; $0 start ;;
    *) echo "Usage: $0 {start|stop|restart}"; exit 1 ;;
esac
```

The `LSB info` block in the comment header gives dependency hints, sysvinit can read these and compute service order (or you arrange the symlink names manually).

When to choose sysvinit:

- You’re maintaining an existing system that already uses it.
- You explicitly want runlevels and you’d find shell scripts comforting.

When *not* to choose sysvinit:

- You're starting fresh in 2025. BusyBox init does the same job with one tenth the bytes. Systemd does *more* if you need it. Sysvinit has little reason to exist in a new design.

33.4 systemd

Systemd is the giant in the room. Every major desktop distro (Debian, Ubuntu, Fedora, Arch) ships systemd by default. It is *the* default for general-purpose Linux in 2025.

What you get:

- **Unit files** instead of shell scripts. Declarative, in `/etc/systemd/system/`:

```
[Unit]
Description=My daemon
After=network.target

[Service]
ExecStart=/usr/bin/my-daemon
Restart=on-failure
RestartSec=5s

[Install]
WantedBy=multi-user.target
```

- **Dependency-driven ordering.** `After=`, `Before=`, `Requires=`, `Wants=`, `Conflicts=`. Systemd computes the right parallel start order.
- **Per-service resource limits** via cgroups: `CPUQuota=`, `MemoryMax=`, `IOWeight=`.
- **Sandbox options**: `PrivateTmp=true`, `ProtectSystem=strict`, `NoNewPrivileges=true`, `CapabilityBoundingSet=`.
- **Socket activation**: a service starts when something connects to its socket, not at boot.
- **Timer units**: like cron jobs, but unit-managed.
- **`journald`**: a binary, queryable log database. `journalctl -u my-daemon` shows your service's logs.
- **A few hundred more features.**

The cost is footprint. Systemd itself plus its required satellites (`udev`, `systemd-journald`, `systemd-logind`, …) is **~6 MB on disk and ~30 MB RAM at idle** on a minimal install. On i.MX6ULL's 512 MB DRAM that's tolerable but not negligible.

> **udev:** the user-space device manager that reacts to kernel device events and creates policy-driven /dev nodes.

When to choose systemd:

- Your rootfs is **Ubuntu-base** or **Debian** (Chapter 35A). They already use systemd. Fighting it is more work than embracing it.
- You need socket activation, advanced sandboxing, or per-service cgroup limits.
- Your team is Linux-distro-experienced and `systemctl status` is muscle memory.

When *not* to choose systemd:

- You're on a Buildroot or yocto-core-image-minimal-style image. They default to BusyBox init for good reason.

> **Buildroot:** a configuration-driven build system that produces a complete root filesystem and related images.

- Your boot-time budget is < 2 seconds. Systemd needs 3-5 seconds on i.MX6ULL just for itself.
- RAM is tight (< 256 MB).

33.5 Comparing them at a glance

	BusyBox init	sysvinit	systemd
Lines of code	~1.5 K	~5 K	~600 K
Disk footprint	0 (part of busybox)	~100 KB	~6 MB
RAM footprint (idle)	< 100 KB	~1 MB	~30 MB
Boot time on i.MX6ULL	~100 ms	~300 ms	~3-5 s
Config format	`inittab` (4-column)	`inittab` + shell scripts	unit files (ini-style)

	BusyBox init	sysvinit	systemd
Service dependencies	manual (script order)	LSB headers	declarative `After=`/`Requires=`
Auto-restart	yes (respawn)	manual	yes (`Restart=`)
Resource limits	no	no	yes (cgroup)
Sandbox features	no	no	yes
Logging	console	console + syslog	journald (binary)
Industry default for embedded	yes	rare	growing

33.6 Why embedded might want *no* init system at all

For some products the simplest answer is: **don't have one**. The kernel's `init=/path/to/myapp` cmdline argument tells the kernel to run *your application* as PID 1. No shell, no inittab, no service manager. Your application:

```
int main(void) {
    /* set up signal handlers for SIGTERM */
    /* mount /proc, /sys, /dev */
    /* configure network */
    /* run forever, waiting on hardware */
}
```

This is the embedded equivalent of an MCU's `main()`. Boot time: under 500 ms total (kernel + your app). Footprint: whatever your app is. No `/etc/`, no `/sbin/init`, no nothing.

When this works:

- The system has one job (industrial controller, single-purpose sensor).
- No need to ssh in, no need to inspect anything live.
- Failure mode is "reboot", your app crashes, PID 1 crashes, kernel panics, watchdog (Ch 51A) reboots.

When this doesn’t work:

- You need diagnostics. Without `/bin/sh` you can’t poke around.
- Multi-service. If you have two daemons, you need *something* to start the second one.

For a developer board you almost always want a shell. For a shipping single-purpose appliance, “no init” is a real choice and surprisingly common.

33.7 Recommendation

For the rest of this book, and for most readers’ real products, **BusyBox init** is the default. We use it in every chapter from here through Part VI. When we need to talk about a feature only systemd has (Chapter 51A’s watchdog daemon, Chapter 35C’s container manager), we’ll note it explicitly. When we get to Chapter 35A (Ubuntu-base), we’ll meet systemd in its natural habitat.

33.8 Key Notes

- PID 1 has system-lifetime responsibilities, including child reaping and shutdown coordination. Running an application directly as init transfers those responsibilities to the application rather than removing them.
- BusyBox init describes actions in `inittab`. Startup ordering, service readiness and richer restart policy must be supplied by the surrounding scripts or service design.
- sysvinit organizes services through scripts and runlevels. Its `inittab` semantics are not interchangeable with BusyBox’s despite the shared filename.
- systemd expresses service ordering, activation and dependencies as units, with resource limits and sandboxing through additional kernel interfaces. Those capabilities bring dependencies and policy to maintain.
- A process restart policy addresses termination, not whether a still-running service is useful. Readiness and health require their own application-to-supervisor contract.
- Footprint and startup cost depend on the complete service set and configuration. The chapter’s comparison does not establish a measured boot-time budget for a MINI product.
- Replacing init with a shell is a diagnostic change to boot policy, not an equivalent production service lifecycle.

33.9 Pitfalls

- **Zombies accumulating.** Some daemons double-fork and detach. The grandchild then gets reparented to PID 1. If PID 1's `wait()` loop is correct, the kernel hands it the SIGCHLD and the child is reaped. BusyBox init does this correctly. Custom PID-1 binaries often forget the `wait()` loop. The symptom is `<defunct>` processes piling up in `ps`.
- **Respawn storm.** A `respawn` line for a service that immediately exits causes infinite restart loop, burning CPU. BusyBox init doesn't rate-limit. Add a `sleep 5` to your service or use systemd's `RestartSec=`.
- **`/etc/inittab` syntax differences.** sysvinit uses runlevels in the second field. BusyBox ignores that field entirely. Don't copy/paste between init implementations.
- **`init=` cmdline overrides everything.** Even if `/sbin/init` exists, if you boot with `init=/bin/sh`, the kernel runs the shell directly. Useful for recovery. Surprising if you forgot you set it.
- **systemd in a 256 MB system.** It will boot but everything will be sluggish. Choose BusyBox or sysvinit instead.
- **systemd unit-file ordering bugs.** Putting both `After=` and `Wants=` on a unit can produce unexpected orderings if the targets aren't carefully chosen. When in doubt, read `systemd.unit(5)`.

33.10 Going deeper

- **BusyBox init source**: `init/init.c` in the BusyBox tree. ~1500 lines. Read it.
- **`Documentation/admin-guide/initrd.rst`** (kernel) for the init handoff details.
- **`man systemd.unit`**, **`man systemd.service`**, canonical systemd reference.
- **`Lennart Poettering`'s "Rethinking PID 1"** blog post, the original systemd design rationale. Worth reading for context.
- **`Rich Felker`'s posts on musl + init systems**: the minimalist's counter-argument.

> Next chapter: **Chapter 34: libc, dynamic linking, and the loader.** Down one more level: what the libraries are that everything else depends on, and how the kernel hands control to user space's first instruction.

Chapter 34: libc, dynamic linking, and the loader

> **What:** the C library that user-space programs link against (glibc, musl, uClibc-ng), the ELF dynamic-linker that resolves shared-library references at runtime (`/lib/ld-linux-armhf.so.3`), and the bookkeeping (PLT, GOT, `LD_LIBRARY_PATH`, `RPATH`) that makes `hello-world` actually find `printf`. **ELF:** Executable and Linkable Format, the standard Linux object and executable file format.
>
> **Why:** every dynamically-linked program on the target depends on this machinery. When it works it's invisible. When it breaks you get `No such file or directory` on a file that does exist. Knowing what the loader does demystifies these failures.
>
> **Focus: ld-linux's job.** When the kernel `exec`s a dynamically-linked program, the first thing that runs is *not* `main()`, it's the dynamic linker, which loads every required shared library, fixes up addresses, and only *then* jumps to your code. Once you've traced this sequence you can debug most `libfoo.so.X: cannot open shared object file` problems.

34.1 Three C libraries

Embedded Linux lets you pick the C library. On a desktop you get glibc and nothing else.

	glibc	musl	uClibc-ng
Origin	GNU project, ~1988	Rich Felker, ~2011	Fork of uClibc, 2014
License	LGPL-2.1	MIT	LGPL-2.1
Static linking	discouraged (NSS won't work)	first-class	first-class
Static-linked "hello world"	~700 KB	~30 KB	~50 KB
Dynamic-linked "hello world"	8 KB + ~2 MB libs	8 KB + 600 KB libs	8 KB + 700 KB libs

	glibc	musl	uClibc-ng
Feature completeness	most complete	very complete (POSIX, most GNU exts)	complete enough for embedded
Performance	optimized aggressively	conservative, predictable	adequate
Used in mainstream	Debian, Ubuntu, Fedora	Alpine, Void	OpenWrt, Buildroot

For embedded Linux **musl is the default**. Reasons:

- One sixth the size of glibc for the same program.
- Static linking actually works (no NSS / dlopen surprises).
- Tighter, more predictable behavior (no surprise heuristics).
- MIT-licensed (LGPL is fine for most people but not all).

glibc remains right for: anything that ships large dynamic apps (Qt, Python, Java), or any rootfs that's a distro derivative (Ubuntu-base / Debian).

> **MCU bridge:** Think of the rootfs as the firmware image's file-backed runtime environment. On an MCU you link everything into flash. On Linux, programs and config live in this mounted tree. **rootfs:** root filesystem, the directory tree mounted at / that contains /bin, /etc, /dev, and libraries.

34.2 Anatomy of a dynamically-linked ELF

Compile a trivial program against glibc:

```
$ cat > hello.c <<'EOF'
#include <stdio.h>
int main(void) { puts("hello"); return 0; }
EOF
$ arm-none-linux-gnueabihf-gcc -o hello hello.c
$ file hello
hello: ELF 32-bit LSB executable, ARM, EABI5 version 1 (SYSV),
       dynamically linked, interpreter /lib/ld-linux-armhf.so.3, ...
```

Two things to notice from `file`:

- **dynamically linked**: references shared libraries that aren't part of the binary.

- **interpreter /lib/ld-linux-armhf.so.3**: when the kernel `exec`s this binary, it actually runs the *interpreter* first, passing the binary as an argument.

That second point is the heart of dynamic linking and worth understanding precisely.

The ELF program headers

```
$ arm-none-linux-gnueabihf-readelf -l hello | head -25

Elf file type is DYN (Position-Independent Executable file)
Entry point 0x4c0
There are 9 program headers, starting at offset 52

Program Headers:
  Type           Offset   VirtAddr   PhysAddr   FileSiz MemSiz   Flg Align
  PHDR           0x000034 0x00000034 0x00000034 0x00120 0x00120  R   0x4
  INTERP         0x000154 0x00000154 0x00000154 0x00019 0x00019  R   0x1
      [Requesting program interpreter: /lib/ld-linux-armhf.so.3]
  LOAD           0x000000 0x00000000 0x00000000 0x004f4 0x004f4  R   0x10000
  LOAD           0x000ed4 0x00010ed4 0x00010ed4 0x0011c 0x0011c  RW  0x10000
  DYNAMIC        0x000eec 0x00010eec 0x00010eec 0x000d0 0x000d0  RW  0x4
  ...
```

The **INTERP** segment is a single string: `/lib/ld-linux-armhf.so.3`. The kernel reads this string out of the ELF, opens that file, and `exec`s *it*, passing your `hello` as a regular argument to the linker. The dynamic linker is the *real* program PID-wise. Your `hello` is its workload.

What the linker does

`ld-linux-armhf.so.3` runs and:

1. **Reads `hello`'s DYNAMIC segment.** This contains a table of needed libraries:

   ```
   $ arm-none-linux-gnueabihf-readelf -d hello | head
    0x00000001 (NEEDED) Shared library: [libc.so.6]
    0x0000001d (RUNPATH) Library runpath: [/path/to/runtime/libs]   (if set)
    ...
   ```

2. **Resolves each NEEDED library.** Search order:
   - `LD_LIBRARY_PATH` environment variable (colon-separated)
   - `RPATH` / `RUNPATH` from the ELF
   - `/etc/ld.so.cache` (built by `ldconfig`)
   - `/lib`, `/usr/lib` (default fallback)
3. **`mmap`s each `.so` into memory** at addresses chosen for that process (ASLR shuffles them if enabled).

4. **Repeats recursively**: Libc may need other libraries (pthread, ld, ...).
5. **Performs symbol resolution.** For every undefined symbol in `hello` and the loaded libraries, find the defining library. Rewrite the address tables (PLT / GOT) so that calls go to the right place.
6. **Calls the program's entry point** (`_start`, which eventually calls `main`).

All of that happens before your `main()` runs. On embedded i.MX6ULL hardware, dynamic linking adds ~5-50 ms to startup depending on how many libraries are pulled in.

34.3 PLT and GOT

The PLT and GOT are the two tables that make dynamic linking efficient.

GOT, Global Offset Table

The GOT holds runtime addresses of every external symbol the program references. The compiler emits `LOAD r0, [GOT+offset]` instead of a fixed address.

```
$ arm-none-linux-gnueabihf-readelf -r hello

Relocation section '.rel.dyn' at offset 0x444 contains 4 entries:
 Offset     Info    Type            Sym.Value  Sym. Name
00010eec  00000017 R_ARM_RELATIVE
00010ef0  00000017 R_ARM_RELATIVE
00010ff0  00000115 R_ARM_GLOB_DAT    00000000   __libc_start_main@GLIBC_2.34
00010ff4  00000215 R_ARM_GLOB_DAT    00000000   __cxa_finalize@GLIBC_2.4

Relocation section '.rel.plt' at offset 0x464 contains 1 entry:
 Offset     Info    Type            Sym.Value  Sym. Name
00011000  00000316 R_ARM_JUMP_SLOT   00000000   puts@GLIBC_2.4
```

`R_ARM_GLOB_DAT` entries are data-section relocations. The loader writes the resolved address of `__libc_start_main` into the GOT slot at `0x10ff0`.

PLT, Procedure Linkage Table

The PLT is a tiny stub per function that jumps via the GOT. The first time you call `puts()`:

```
puts@plt:
    ldr  r12, [pc, #plt_offset_to_GOT_entry]
    bx   r12              @ → resolved address of puts, or to the resolver stub
```

The first time, the GOT slot still points at a "resolver" routine inside ld-linux. The resolver looks up `puts` for real, writes the real address into the GOT slot, and jumps there. **Subsequent calls** go directly via the GOT, fast.

This trick is called **lazy binding**. The dynamic linker doesn't resolve every function at startup. It resolves them on first use. You can disable lazy binding with `LD_BIND_NOW=1`, useful for security-hardened systems (no resolver stub at runtime).

34.4 Where libraries actually come from

On the target:

```
[root@pa-mini:~]# ls /lib/
ld-2.31.so              libnss_files.so.2
ld-linux-armhf.so.3     libnss_files-2.31.so
libc-2.31.so            libpthread-2.31.so
libc.so.6               libpthread.so.0
libdl-2.31.so           libresolv-2.31.so
libdl.so.2              libresolv.so.2
libm-2.31.so            librt-2.31.so
libm.so.6               librt.so.1
```

Notice:

- `libc.so.6` is a *symlink* to `libc-2.31.so`. The "6" is the ABI version. "2.31" is the implementation version.

> **ABI:** Application Binary Interface: the calling convention, register use, binary format, and library contract that let separately built code run together.

- `ld-linux-armhf.so.3` is the dynamic linker, a *real file*, not a symlink. (See Ch 31 §31.10's gotcha.)
- The `nss_files` library is for Name Service Switch, loaded dynamically by glibc when you call `gethostbyname`, `getpwuid`, etc. This is the part that breaks under static linking.

The dynamic linker also consults `/etc/ld.so.cache`, a pre-indexed map of `library-name → file-path`:

```
$ ldconfig -p | head
21 libs found in cache `/etc/ld.so.cache'
   libz.so.1 (libc6,hard-float) => /lib/libz.so.1
   libuClibc-1.0.so (libc6) => /lib/libuClibc-1.0.so
   ...
```

`ldconfig` rebuilds this cache from `/etc/ld.so.conf` and `/etc/ld.so.conf.d/*.conf`. On embedded systems running BusyBox, `ldconfig` is often skipped, the search defaults (`/lib`, `/usr/lib`) cover everything.

34.5 Looking inside a real program

`ldd` shows what a binary depends on:

```
[root@pa-mini:~]# ldd /bin/busybox
        not a dynamic executable
```

(Static, no dependencies.) But:

```
[root@pa-mini:~]# ldd /usr/bin/some-dynamic-binary
        libpthread.so.0 => /lib/libpthread.so.0 (0xb6f00000)
        libm.so.6 => /lib/libm.so.6 (0xb6e80000)
        libc.so.6 => /lib/libc.so.6 (0xb6d80000)
        /lib/ld-linux-armhf.so.3 (0xb6f80000)
```

Right column is *load addresses*, where each `.so` was `mmap`'d into the process's address space.

For a deeper look:

```
[root@pa-mini:~]# LD_DEBUG=libs my-binary 2>&1 | head -20
      6: find library=libpthread.so.0 [0]; searching
      6:  search cache=/etc/ld.so.cache
      6:   trying file=/lib/libpthread.so.0
      6:
      6: find library=libc.so.6 [0]; searching
      6:  search cache=/etc/ld.so.cache
      6:   trying file=/lib/libc.so.6
      6:
      6: calling init: /lib/libpthread.so.0
      6: calling init: /lib/libc.so.6
      6: initialize program: my-binary
      6: transferring control: my-binary
```

`LD_DEBUG=libs` (`LD_DEBUG=help` for the full list of categories) is the diagnostic tool for "why isn't this library being found?" Use `LD_DEBUG=libs` before guessing.

34.6 RPATH and friends

Sometimes you ship a binary that depends on libraries *not* in `/lib`. Three common ways to point the loader at them:

LD_LIBRARY_PATH

```
[root@pa-mini:~]# LD_LIBRARY_PATH=/opt/myapp/lib /opt/myapp/bin/my-binary
```

Most general. User can override per-invocation. Downside: easy to forget. Security-sensitive binaries (setuid) ignore it for safety.

`RPATH` baked into the binary

```
$ arm-none-linux-gnueabihf-gcc -Wl,-rpath,/opt/myapp/lib -o my-binary my.c
```

The binary now searches `/opt/myapp/lib/` automatically. Visible in `readelf -d` as a `RUNPATH` entry. Best for self-contained apps that bundle their own libs.

`$ORIGIN` is a special token that expands to the binary's own directory:

```
$ arm-none-linux-gnueabihf-gcc -Wl,-rpath,'$ORIGIN/../lib' -o my-binary my.c
```

Binary in `/opt/myapp/bin/` searches `/opt/myapp/lib/`, regardless of where the package is installed.

`/etc/ld.so.conf.d/myapp.conf`

```
$ echo /opt/myapp/lib > /etc/ld.so.conf.d/myapp.conf
$ ldconfig
```

System-wide. Affects all binaries. Use sparingly.

34.7 Static vs dynamic for embedded, the real trade-off

	Static	**Dynamic**
Binary size	Big (KB to MB per program)	Small (KB)
RAM per process	High (no library sharing)	Low (shared mmaps)
Disk for whole image	Variable; if you have 1 binary, smaller. 10 binaries: larger.	One copy of each lib regardless of binary count
Update granularity	Replace binary	Replace library; binaries pick it up
`dlopen()` works	No (in practice for glibc)	Yes
DNS / NSS works	No (glibc) / Yes (musl)	Yes

	Static	Dynamic
Boot time	Slightly faster (no linker startup)	Slightly slower

For embedded with **N small binaries**, dynamic linking is almost always smaller in total. For embedded with **1 large binary**, static can win.

`busybox`, our biggest single binary, is the canonical case for static: 580 KB static vs ~580 KB binary + ~2 MB glibc dynamic. Choosing static for busybox saves 2 MB. Choosing dynamic means dozens of other small dynamic binaries can also be on the system without adding more libc copies.

The decision is per-binary, not per-system. Mix as needed.

34.8 Key Notes

- A dynamically linked ELF declares an interpreter path and needed libraries. The target must provide a compatible loader and dependency set before application startup can succeed.
- An executable can exist while execution reports a missing file because its interpreter is absent. A found library can still fail through missing symbols or incompatible symbol versions.
- GOT entries and PLT stubs support relocation and indirect calls to shared code. Lazy binding moves some function resolution to first use rather than eliminating it.
- Library search paths are part of deployment policy. `LD_LIBRARY_PATH` is per-environment state, a binary's runpath is embedded metadata, and the loader cache is system-wide state.
- `$ORIGIN` is interpreted by the loader relative to the ELF object, not as a shell variable for a wrapper script. Quoting preserves it while linking.
- Static linking simplifies some deployment dependencies but duplicates linked code across binaries and changes update granularity. Dynamic linking shares libraries while requiring coordinated ABI compatibility.
- libc choices affect runtime behavior as well as size. Naming the same C API does not make glibc, musl and uClibc-ng binaries interchangeable.

34.9 Pitfalls

- **`libfoo.so.X: cannot open shared object file`**: the dynamic linker can't find a NEEDED library. Diagnose with `LD_DEBUG=libs`. Fix by copying the library into `/lib` or adding to `LD_LIBRARY_PATH` / `RPATH`.
- **`relocation error: undefined symbol`**: the library was found but doesn't have a symbol the binary needs. Usually means library *version* mismatch. The binary was built against newer libc. The runtime has older.
- **Mixed glibc and musl on one rootfs.** Mixing glibc and musl on one rootfs is possible but easy to get wrong. Glibc uses SONAME `libc.so.6` and loader `/lib/ld-linux-armhf.so.3`. Musl uses its own loader `/lib/ld-musl-armhf.so.1` and its own libc. They can live in separate prefixes. The failure mode is when both want to own the same `/lib/libc.so.6` symlink. Pick one libc per rootfs, or put musl binaries under their own prefix with the loader path baked in via `RPATH`.
- **Static glibc + getaddrinfo.** Returns "Temporary failure in name resolution" with no obvious cause. NSS modules are dlopen'd at runtime even for "static" binaries. If the .so files aren't on disk you lose DNS. Either ship the NSS .so files alongside your "static" binary or switch to musl.
- **`LD_LIBRARY_PATH` and setuid binaries.** Ignored for setuid binaries (security). Don't rely on it for system binaries.
- **`ldconfig` not run after copying libraries.** Some glibc setups won't find a library until you `ldconfig`. Symptom: works after `ldconfig`, breaks on fresh boot. Fix: run `ldconfig` from `rcS` or include the library path in `/etc/ld.so.conf.d/`.
- **`$ORIGIN` rpath doesn't work in shell scripts.** It's resolved by the *dynamic linker*, not the shell. Only matters for the ELF binary itself. If you invoke the binary via a wrapper script, `$ORIGIN` is relative to the wrapper's location only if the wrapper itself is dynamic.

34.10 Going deeper

- **`man ld.so`**: the canonical loader reference.
- **`man ldd`**, **`man ldconfig`**, **`man dlopen`**.
- **`Documentation/admin-guide/dynamic-debug-howto.rst`** for kernel-side dlopen-like patterns.
- **`Drepper, "How to Write Shared Libraries"`**: long PDF, gold standard.
- **musl's website**, `musl.libc.org`, design rationale, comparison tables.
- **`The ELF Specification`** (System V ABI, ARM supplement) for the gritty details of dynamic sections.

> Next chapter: **Chapter 35: Buildroot, after you can do it by hand.** Buildroot automates the rootfs composition whose BusyBox, libc and init interfaces we have examined. **Buildroot:** a configuration-driven build system that produces a complete root filesystem and related images.

Chapter 35: Buildroot, after you can do it by hand

> **What:** Buildroot's configuration-driven assembly of a root filesystem, optional boot artifacts, packages and product customizations. The EVK configuration is a build-system case, not a qualified MINI image. **MCU bridge:** Think of the rootfs as the firmware image's file-backed runtime environment. On an MCU you link everything into flash. On Linux, programs and config live in this mounted tree. **rootfs:** root filesystem, the directory tree mounted at / that contains /bin, /etc, /dev, and libraries. **Buildroot:** a configuration-driven build system that produces a complete root filesystem and related images.
>
> **Why:** Chapter 31 separated the files and runtime responsibilities in a small rootfs. Buildroot turns that composition into maintained configuration and package recipes. A completed hand-built rootfs is not a prerequisite for understanding the model.
>
> **Focus: the `output/` tree**, every artefact Buildroot produces lives in one place under `output/`, with a predictable layout. Once you know `output/`, debugging a failed build is a directed search instead of a hunt.

35.1 What Buildroot is

Buildroot is a **make-driven** build system written almost entirely in GNU Make + shell. Architecturally:

- One Kconfig tree describes ~3000 packages plus the BusyBox / kernel / toolchain configuration.
- Each package has a small `package.mk` that says "download from URL X, extract, configure, build, install."
- A top-level `Makefile` orchestrates: select packages → fetch tarballs → build toolchain → build each package → assemble rootfs.
- Output is a `rootfs.tar` (or `.ext4` / `.cpio.gz` / `.ubifs`, your choice).

Three things Buildroot is *good* at:

1. **Tiny minimal images.** A no-extras BusyBox rootfs from Buildroot is ~3 MB. Same Yocto build is ~30 MB.

> **Yocto:** a metadata-driven build system for producing custom Linux distributions.

1. **Reproducible builds.** Same `.config` + same source versions → bit-identical output (usually).
2. **Fast development cycle.** Type one command, get a `rootfs.tar` in 10–30 minutes.

Three things Buildroot is *not* good at:

1. **Per-package customisation.** Patching a package is doable but awkward.
2. **Concurrent multi-config builds.** Each `make` is one config. Switching configs means rebuilding.
3. **Long-term maintenance of many products.** Yocto / OpenEmbedded scales to many product variants better.

Buildroot is the right tool for learning, and for single-purpose products with simple package needs. Yocto is the right tool for commercial product lines with many variants. Most engineers learn Buildroot first.

35.2 Get the source

```
$ cd ~/imx6ull/src
$ wget https://buildroot.org/downloads/buildroot-2025.02.15.tar.xz
$ tar xJf buildroot-2025.02.15.tar.xz
$ cd buildroot-2025.02.15
```

Buildroot has LTS and short stable branches. As of June 2026, `2025.02.x` is the active LTS line, supported until March 2028. `2026.05.x` is the latest stable line, supported for a shorter window. Use LTS for a product unless you need a package or fix that only exists in a newer stable branch.

```
$ ls
arch/      configs/   docs/      Makefile      support/
board/     dl/        fs/        package/      system/
boot/      DEVELOPERS Kbuild     README        toolchain/
CHANGES    Config.in  linux/     Makefile.legacy utils/
```

Top-level layout:

Directory	Purpose
`arch/`	CPU architecture support (most users never touch this)

Directory	Purpose
`board/`	Per-board files: defconfigs and post-build scripts
`boot/`	Bootloader recipes (`uboot/`, `barebox/`, `grub2/`)
`configs/`	Pre-canned configurations (`*_defconfig`)
`dl/`	Downloaded source tarballs (cached across builds)
`docs/`	Manual, read `docs/manual/manual.html` after this chapter
`fs/`	Filesystem-image builders (ext4, cpio, squashfs, ubi, ...)
`linux/`	Kernel recipe
`package/`	One subdirectory per package, ~3000 entries
`support/`	Support scripts (kconfig wrapper, checksum tools)
`system/`	System-skeleton files (default `/etc/inittab`, etc.)
`toolchain/`	Cross-toolchain recipes

The first time you build, `dl/` is empty and Buildroot downloads every source tarball. Subsequent builds reuse the cache. Net `dl/` size for a typical build: 200 MB to 1 GB depending on packages.

35.3 First build, the i.MX6UL EVK defconfig

Buildroot ships a defconfig for the NXP i.MX6UL/ULL EVK:

```
$ make list-defconfigs | grep -i imx6
freescale_imx6sololiteevk_defconfig - ...
freescale_imx6sxsabresd_defconfig   - ...
```

```
freescale_imx6ulevk_defconfig       - NXP i.MX 6UL EVK
freescale_imx6ullevk_defconfig      - NXP i.MX 6ULL EVK   ← this one
```

Configure and build:

```
$ make freescale_imx6ullevk_defconfig
$ make -j$(nproc) 2>&1 | tee build.log
```

What happens, in order:

1. `make` parses Kconfig + `.config`, builds a list of selected packages.
2. Downloads every required tarball to `dl/`.
3. Builds the cross-toolchain (the first build only. About 8 minutes).
4. Extracts and builds each package, in dependency order.
5. Assembles the rootfs in `output/target/`.
6. Packages it into `output/images/rootfs.<format>`.

First build on a 4-core machine: 30-60 minutes. Subsequent rebuilds: seconds (only changed packages rebuild).

When done:

```
$ ls output/images/
imx6ull-14x14-evk.dtb   rootfs.cpio   rootfs.cpio.gz   rootfs.ext2
rootfs.ext4 → rootfs.ext2               rootfs.tar    SPL    u-boot-dtb.imx   zImage
```

The whole stack, SPL, U-Boot, kernel, DTB, rootfs in multiple formats, produced by one `make`. The defconfig enables `BR2_TARGET_UBOOT` and `BR2_LINUX_KERNEL`. Turn both off when you only need the rootfs.

> **MCU bridge:** Think of SPL like the tiny early startup code that runs from internal SRAM before DDR is usable. **MCU bridge:** Think of U-Boot like a much larger boot stub plus debug monitor: it initializes hardware, loads the next image, and gives you commands before Linux starts. **SPL:** Secondary Program Loader, a tiny first U-Boot stage that fits in OCRAM and initializes DDR. **U-Boot:** the bootloader that initializes enough hardware to load and start the Linux kernel.

35.4 The output/ tree

The single most useful thing to learn about Buildroot is its output layout:

```
output/
├── build/                          # extracted, configured, built packages
│   ├── busybox-1.36.1/
│   │   ├── .stamp_downloaded
│   │   ├── .stamp_extracted
│   │   ├── .stamp_configured
│   │   ├── .stamp_built
│   │   ├── .stamp_target_installed
│   │   └── ...source files extracted here, build artefacts mixed in...
│   ├── host-gcc-arm-...
│   ├── linux-headers-...
│   └── ...one dir per package...
├── host/                           # host-side tools that Buildroot built
│   ├── bin/                        # qemu, mkimage, mkfs.ext4, ...
│   ├── arm-buildroot-linux-...-/   # the cross-toolchain
│   └── ...
├── images/                         # the FINAL ARTEFACTS
│   ├── rootfs.tar
│   ├── rootfs.ext4
│   ├── zImage
│   ├── imx6ull-14x14-evk.dtb
│   └── SPL, u-boot-dtb.imx
├── staging → ./host/arm-buildroot-linux-.../sysroot  # symlink for convenience
└── target/                         # the rootfs staging area (pre-packaging)
    ├── bin/
    ├── sbin/
    ├── etc/
    ├── lib/
    └── ...
```

Three subdirectories you'll visit:

- **`output/target/`**: the rootfs being assembled. You can `chroot` into it for inspection. **Don't edit it directly**, your changes are wiped on next build. Use the post-build script mechanism (§35.7) for persistent customisations.
- **`output/host/`**: host-side binaries. The `mkimage` Buildroot built for you lives at `output/host/bin/mkimage`. Useful when you want a known-version tool without polluting the host.
- **`output/build/<package>/`**: where each package was unpacked and built. When a build fails inside a package, this is where you go.

The `.stamp_*` files are Buildroot's idea of "what stage of the build is this package in?" If you delete `.stamp_built` and re-run `make <package>`, only that package rebuilds.

35.5 Reading the defconfig

```
$ head -40 .config
#
# Automatically generated file; DO NOT EDIT.
# Buildroot 2025.02 Configuration
#
```

```
BR2_HAVE_DOT_CONFIG=y
BR2_HOST_GCC_AT_LEAST_4_9=y
...
BR2_ARCH_HAS_TOOLCHAIN_BUILDROOT=y
BR2_arm=y
BR2_ARM_CPU_HAS_VFPV4=y
BR2_arm_cortex_a7=y
BR2_ARM_FPU_NEON_VFPV4=y
BR2_ARM_EABIHF=y
...
BR2_TOOLCHAIN_BUILDROOT_GLIBC=y      # use Buildroot-built glibc
BR2_TOOLCHAIN_BUILDROOT_CXX=y         # with C++
...
BR2_PACKAGE_BUSYBOX=y
BR2_PACKAGE_BUSYBOX_SHOW_OTHERS=y
...
BR2_LINUX_KERNEL=y
BR2_LINUX_KERNEL_CUSTOM_VERSION=y
BR2_LINUX_KERNEL_CUSTOM_VERSION_VALUE="6.6"
BR2_LINUX_KERNEL_DEFCONFIG="imx_v6_v7"
BR2_LINUX_KERNEL_DTS_SUPPORT=y
BR2_LINUX_KERNEL_INTREE_DTS_NAME="nxp/imx/imx6ull-14x14-evk"   # v6.5+ path; pre-v6.5 was
just "imx6ull-14x14-evk"
...
BR2_TARGET_UBOOT=y
BR2_TARGET_UBOOT_BOARDNAME="mx6ull_14x14_evk"
BR2_TARGET_UBOOT_CUSTOM_VERSION=y
BR2_TARGET_UBOOT_CUSTOM_VERSION_VALUE="2026.04"
BR2_TARGET_UBOOT_FORMAT_IMX=y
```

The file is dense but readable. Section by section:

- **`BR2_arm` + `BR2_arm_cortex_a7`**: target CPU.
- **`BR2_TOOLCHAIN_BUILDROOT_*`**, Buildroot builds its own glibc-based toolchain.
- **`BR2_PACKAGE_BUSYBOX`**: include BusyBox in the rootfs.
- **`BR2_LINUX_KERNEL`**: build the kernel as part of this Buildroot run, using mainline 6.6 with `imx_v7` defconfig and the `imx6ull-14x14-evk` DTS.
- **`BR2_TARGET_UBOOT`**: build U-Boot 2023.10 as part of this run.

To suppress kernel + U-Boot builds (you're building those separately):

```
$ make menuconfig
# Linux  →  [ ] Linux Kernel        ← uncheck
# Bootloaders  →  [ ] U-Boot         ← uncheck
$ make
```

Or programmatically:

```
$ ./utils/config --disable LINUX_KERNEL
$ ./utils/config --disable TARGET_UBOOT
```

```
$ make olddefconfig
$ make
```

35.6 Adding a package

Buildroot ships ~3000 packages. To pick one (e.g., `nano` editor):

```
$ make menuconfig
# Target packages  →  Text editors and viewers  →  [*] nano
$ make
```

Or:

```
$ ./utils/config --enable BR2_PACKAGE_NANO
$ make olddefconfig
$ make
```

Buildroot:

1. Downloads `nano-7.2.tar.xz` to `dl/nano/`.
2. Extracts to `output/build/nano-7.2/`.
3. Runs the package's `./configure` with the cross-compiler.
4. `make` and `make install` into `output/staging/` and `output/target/`.
5. Re-rolls `rootfs.tar`.

Re-deploy and `nano` is now on the target. Total time for adding a small package: ~30 seconds.

35.7 Customising without forking, `BR2_ROOTFS_OVERLAY` and post-build scripts

You almost always need to add *your own* files to the rootfs (custom `/etc/inittab`, `/etc/init.d/S99myapp`, your application binary). Three mechanisms, in order of complexity:

Overlay directory

```
$ mkdir -p board/myorg/overlay/etc/init.d
$ cat > board/myorg/overlay/etc/init.d/S99myapp <<'EOF'
#!/bin/sh
echo "starting my app"
/usr/bin/myapp &
EOF
$ chmod +x board/myorg/overlay/etc/init.d/S99myapp
```

In `.config`:

```
BR2_ROOTFS_OVERLAY="board/myorg/overlay"
```

Now every Buildroot build copies the overlay tree on top of `output/target/` before packaging. Your `S99myapp` ships in the rootfs.

Post-build script

For things you need to *generate* (interpolated config files, version strings, etc.):

```
$ cat > board/myorg/post-build.sh <<'EOF'
#!/bin/sh
TARGET_DIR=$1
echo "build $(date) on $(hostname)" > "$TARGET_DIR/etc/build-info"
EOF
$ chmod +x board/myorg/post-build.sh
```

In `.config`:

```
BR2_ROOTFS_POST_BUILD_SCRIPT="board/myorg/post-build.sh"
```

The script receives `$TARGET_DIR` as `$1` and runs after the overlay is applied but before packaging.

Custom package

For shipping your own software *as a Buildroot package* with a real Makefile-driven build:

```
$ mkdir package/myapp
$ cat > package/myapp/Config.in <<'EOF'
config BR2_PACKAGE_MYAPP
    bool "myapp"
    help
      The myapp daemon.
EOF
$ cat > package/myapp/myapp.mk <<'EOF'
MYAPP_VERSION = 1.0
MYAPP_SITE = $(TOPDIR)/../my-app-source
MYAPP_SITE_METHOD = local

define MYAPP_BUILD_CMDS
    $(MAKE) CC="$(TARGET_CC)" -C $(@D)
endef

define MYAPP_INSTALL_TARGET_CMDS
    $(INSTALL) -D -m 0755 $(@D)/myapp $(TARGET_DIR)/usr/bin/myapp
endef

$(eval $(generic-package))
EOF
```

Then in the top-level `package/Config.in`, add `source "package/myapp/Config.in"`. After `make menuconfig && make`, your `myapp` is in the rootfs.

For a single product, the overlay mechanism handles 90% of needs. For multi-product BSPs you graduate to packages.

35.8 Saving the defconfig

After your `menuconfig` changes:

```
$ make savedefconfig BR2_DEFCONFIG=configs/myproduct_defconfig
$ git add configs/myproduct_defconfig
```

That file is ~200 lines and contains only the differences from defaults. Anyone with the Buildroot tree + the defconfig can reproduce your build.

35.9 Comparing to the hand-built rootfs from Chapter 31

	Chapter 31 (by hand)	**Chapter 35 (Buildroot)**
Number of commands typed	~20	~3
Number of files created by you	~6	0 (everything templated)
BusyBox config	Started from busybox `defconfig`	Same; Buildroot wraps it
Library copying	Manual (`cp -d ... lib/`)	Automatic
`/etc/inittab`	You wrote it	Buildroot provided default; overlay if you want custom
Reproducibility	Notes in your head	`defconfig` file in git
First-build time	~5 minutes once you've practised	~30 minutes
Rebuild time	Edit + retest in seconds (NFS)	Edit + rerun build (~1 minute for an overlay change)

	Chapter 31 (by hand)	Chapter 35 (Buildroot)
New package	Hunt down + build + install yourself	One menuconfig click

The hand-built case explains the structure. Buildroot provides an automated image-oriented approach, with product-specific configuration still owned by the product team.

35.10 Key Notes

- Buildroot owns the dependency-ordered assembly of selected packages into images. Its configuration describes the intended system rather than a sequence of edits performed on a live target.
- `output/build` holds package build state, `output/host` holds host tools, `output/staging` provides the target sysroot, and `output/images` holds deployable artifacts. `output/target` is an intermediate tree, not the maintained product source.
- A rootfs overlay supplies static product files, a post-build script generates or adjusts staged content, and a package recipe owns software build and installation. These mechanisms solve different customization needs.
- Persistent changes belong in defconfigs, overlays, recipes and patches. Editing generated output hides ownership and can lose the change during regeneration.
- Integrated kernel and bootloader builds require one clear artifact authority. Mixing separately built images with another build's modules or DTB creates release mismatches.
- A reference-board defconfig describes that reference board. Its SPL, U-Boot and DTB artifacts are not automatically MINI firmware.
- Image creation and board operation are separate boundaries. A repeatable configuration still needs source, toolchain and product-file identity, plus runtime qualification before deployment.

35.11 Pitfalls

- **`make clean` is not enough.** Buildroot has many levels of "clean":
  - `make clean`: remove `output/`. Tarballs in `dl/` remain.
  - `make distclean`: remove `output/` and `.config`.
  - `make <pkg>-dirclean`: clean *one* package.

    - `make <pkg>-rebuild`: force rebuild without dirty clean.

- **Network access required for first build.** Buildroot downloads from upstream. Behind a corporate firewall, you may need `BR2_PRIMARY_SITE=` to point to a mirror. The `dl/` directory caches between builds.
- **Building as root.** Buildroot refuses to build as root (good safety check). Use a regular user.
- **Modifying files under `output/target/` directly.** They will be overwritten the next time `make` runs. Use overlay or post-build scripts.
- **Linux kernel built by Buildroot may not match yours.** If you enabled `BR2_LINUX_KERNEL` and pointed it at a custom branch, but also reused your own kernel binary outside Buildroot, your `modules_install` may collide. Disable Buildroot's kernel build OR commit to using only Buildroot's.
- **`/etc/inittab` from Buildroot's skeleton is different from §31.5.** Buildroot uses `system/skeleton/etc/inittab`. If your overlay's inittab doesn't completely replace it, Buildroot may keep parts of its own.
- **Package version pinning.** Buildroot pins each package to a specific version (in the `.mk` file). To upgrade, you edit the `.mk` to bump the version and possibly update the checksum file. Don't try to upgrade the package from inside `output/build/`. The next clean rebuild wipes it.

35.12 Going deeper

- **`docs/manual/manual.html`** in the Buildroot tree, the canonical reference.
- **`docs/manual/adding-packages-*.txt`** for the package-creation tutorial.
- **`http://buildroot.org/#community`**: mailing list (`buildroot@buildroot.org`) and IRC (`#buildroot` on Libera Chat) for help.
- **`Bootlin's Buildroot training`**: free online materials, very thorough.
- **`Yocto vs Buildroot: A Comparison`** articles on LWN, when you outgrow Buildroot's limitations, this is the case for switching to Yocto.

> Next chapter: **Chapter 35D: Bootable SD and eMMC image layout.** NFS is for development. A product needs a real boot partition, rootfs partition, and data partition.

Chapter 35D: Bootable SD and eMMC image layout

Chapters 31 and 35 got a root filesystem running over NFS. NFS is perfect for development, but a product needs local storage.

This chapter builds the storage layout you will use again and again:

```
SD/eMMC
  p1  FAT32   boot files: zImage, DTB, optional initramfs
  p2  ext4    root filesystem
  p3  ext4    persistent data
```

The exact partition sizes change per product. The pattern does not.

35D.1 Why split boot, rootfs, and data?

Do not put everything in one writable ext4 partition.

Partition	Mounted at	Purpose
Boot	`/boot` or not mounted	Files U-Boot reads: kernel, DTB, initramfs
Rootfs	`/`	Operating system files
Data	`/data`	App config, logs, databases, user files

This gives you clean boundaries:

- You can replace the rootfs without erasing user data.
- You can mount rootfs read-only later.
- You can factory reset by clearing `/data` or the overlay partition.
- You can update boot files separately from application data.

35D.2 Use labels, not guessed device names

On one board, the SD card may appear as `/dev/mmcblk0`. On another boot path, eMMC may be `/dev/mmcblk1`.

Use filesystem labels:

```
root=LABEL=rootfs rootwait
```

Then the kernel searches for the partition labeled `rootfs`.

Create labels when formatting:

```
$ sudo mkfs.vfat -F 32 -n BOOT /dev/sdb1
$ sudo mkfs.ext4 -L rootfs /dev/sdb2
$ sudo mkfs.ext4 -L data /dev/sdb3
```

Labels make bootargs easier to read and harder to break.

35D.3 Manual image build

Manual partitioning is useful once. It teaches what the image builder later automates.

First, identify the target card:

```
$ lsblk -o NAME,SIZE,MODEL,TRAN,TYPE,MOUNTPOINTS
```

Assume the SD card is `/dev/sdb`. Replace this with the correct device on your host.

Create partitions:

```
$ sudo parted /dev/sdb --script \
      mklabel msdos \
      mkpart primary fat32 1MiB 129MiB \
      set 1 boot on \
      mkpart primary ext4 129MiB 641MiB \
      mkpart primary ext4 641MiB 100%
```

Format:

```
$ sudo mkfs.vfat -F 32 -n BOOT /dev/sdb1
$ sudo mkfs.ext4 -L rootfs /dev/sdb2
$ sudo mkfs.ext4 -L data /dev/sdb3
```

Mount:

```
$ mkdir -p /tmp/imx6ull-card/boot /tmp/imx6ull-card/rootfs /tmp/imx6ull-card/data
$ sudo mount /dev/sdb1 /tmp/imx6ull-card/boot
```

```
$ sudo mount /dev/sdb2 /tmp/imx6ull-card/rootfs
$ sudo mount /dev/sdb3 /tmp/imx6ull-card/data
```

Copy boot files:

```
$ sudo cp ~/imx6ull/src/linux-v6.6/arch/arm/boot/zImage /tmp/imx6ull-card/boot/
$ sudo cp ~/imx6ull/src/linux-v6.6/arch/arm/boot/dts/nxp/imx/imx6ull-your-board.dtb \
      /tmp/imx6ull-card/boot/
```

Install the rootfs:

```
$ sudo tar -xf ~/imx6ull/buildroot/output/images/rootfs.tar \
      -C /tmp/imx6ull-card/rootfs
```

Create data directories:

```
$ sudo mkdir -p /tmp/imx6ull-card/data/myapp /tmp/imx6ull-card/data/log
```

Unmount:

```
$ sync
$ sudo umount /tmp/imx6ull-card/boot
$ sudo umount /tmp/imx6ull-card/rootfs
$ sudo umount /tmp/imx6ull-card/data
```

Now the card has a boot partition, rootfs partition, and data partition.

35D.4 Bootargs for local rootfs

In U-Boot:

```
=> setenv bootargs 'console=ttymxc0,115200 earlycon root=LABEL=rootfs rw rootwait'
=> fatload mmc 0:1 0x82000000 zImage
=> fatload mmc 0:1 0x83000000 imx6ull-your-board.dtb
=> bootz 0x82000000 - 0x83000000
```

Save it:

```
=> setenv bootcmd 'fatload mmc 0:1 0x82000000 zImage && fatload mmc 0:1 0x83000000
imx6ull-your-board.dtb && bootz 0x82000000 - 0x83000000'
=> saveenv
```

Use `&&` so a failed load stops the boot.

35D.5 Mount /data

In the rootfs, add this to `/etc/fstab`:

```
LABEL=data  /data  ext4  defaults,noatime  0  2
```

Create the mountpoint:

```
$ sudo mkdir -p /tmp/imx6ull-card/rootfs/data
```

On target:

```
[root@pa-mini:~]# mount /data
[root@pa-mini:~]# touch /data/test
[root@pa-mini:~]# reboot
[root@pa-mini:~]# ls /data/test
```

If the file survives, persistent data works.

35D.6 Automate with genimage

Manual partitioning is not release engineering. For repeatable images, use `genimage`.

Buildroot can run `genimage` as a post-image step. The core file is `genimage.cfg`:

```
image sdcard.img {
  hdimage {
  }

  partition boot {
    partition-type = 0x0C
    bootable = true
    image = "boot.vfat"
    size = 128M
  }

  partition rootfs {
    partition-type = 0x83
    image = "rootfs.ext4"
    size = 512M
  }

  partition data {
    partition-type = 0x83
    size = 512M
  }
}

image boot.vfat {
  vfat {
    files = {
      "zImage",
      "imx6ull-your-board.dtb"
    }
  }
```

```
  size = 128M
}
```

The result is one flashable file:

```
$ sudo dd if=sdcard.img of=/dev/sdb bs=4M conv=fsync status=progress
```

Buildroot board directories usually keep this under:

```
board/myorg/myboard/genimage.cfg
```

Chapter 35E later uses that structure.

35D.7 Verify the card

Before booting, check:

```
$ lsblk -f /dev/sdb
NAME   FSTYPE LABEL  SIZE
sdb
├─sdb1 vfat   BOOT   128M
├─sdb2 ext4   rootfs 512M
└─sdb3 ext4   data   ...
```

After boot:

```
[root@pa-mini:~]# cat /proc/cmdline
[root@pa-mini:~]# findmnt /
[root@pa-mini:~]# findmnt /data
[root@pa-mini:~]# df -h
```

You want:

- `/` mounted from `LABEL=rootfs`,
- `/data` mounted from `LABEL=data`,
- boot files loaded from the FAT partition.

35D.8 Key Notes

- Boot files, the root filesystem and application data have different owners and update lifetimes. Separate partitions make those boundaries visible but do not by themselves provide rollback.

- Partition identity and device enumeration are separate concerns. A label or PARTUUID can describe the intended root, while the selected init and kernel still need support for resolving that form.
- The ROM boot region is outside the filesystem-level view. Image packaging must keep bootloader placement, partition tables and later filesystem writes from overlapping.
- Root mounting depends on storage discovery and filesystem availability. A correct filename in U-Boot does not prove that the kernel can mount the selected root.
- Service startup must follow the availability of persistent data. Otherwise an application can accidentally write into the underlying root directory instead of the intended data partition.

35D.9 Pitfalls

- **Writing the wrong `/dev/sdX`.** Always run `lsblk` before `dd`, `parted`, or `mkfs`.
- **Using `/dev/mmcblk0p2` in bootargs.** It may change when boot source changes. Prefer `LABEL=` or `PARTUUID=`.
- **Forgetting `rootwait`.** MMC may not be ready when the kernel tries to mount root.
- **Putting mutable app data in `/etc`.** Use `/data` for runtime data.
- **Mounting `/data` too late.** If your app starts before `/data` is mounted, it may create files on the rootfs instead.

Previous chapter: Chapter 35 - Buildroot

Next chapter: Chapter 35I - Boot from SPI NOR flash

Chapter 35I: Boot from SPI NOR flash

Chapter 35D used SD/eMMC. That is the normal bulk-storage path.

SPI NOR flash is different:

- small, often 4 MB to 32 MB,
- soldered to the board,
- accessed through the MTD subsystem, not the block layer,
- erase-before-write,
- good for bootloaders, kernels, DTBs, rescue images, and small read-only root filesystems.

This chapter is about the system layout, not the flash command protocol. The low-level NOR protocol and Linux `spi-nor` driver are covered later in Chapter 64.

35I.1 First question: can your board boot from it?

The i.MX6ULL Boot ROM can boot from SPI-NOR and QSPI-NOR when the boot pins select that mode and the board wiring matches the ROM boot path.

That last sentence matters.

Some development boards expose only SD, eMMC, NAND, and USB boot modes in their DIP switches. They may still have an SPI flash connected to ECSPI for storage, but the Boot ROM may not be strapped to boot from it on that board.

So there are two cases:

Case	What works
Board boot pins support SPI/QSPI NOR	Boot ROM loads U-Boot directly from SPI flash
Board has SPI flash but no SPI boot strap	U-Boot/Linux can use the flash after boot, but ROM boot must still come from SD/eMMC/NAND/USB

This chapter is most useful for a custom product board or a dev board whose boot switches expose SPI/QSPI boot.

35I.2 SPI NOR is not eMMC

Do not use `fdisk` on SPI NOR.

eMMC and SD are block devices:

```
/dev/mmcblk0
/dev/mmcblk0p1
/dev/mmcblk0p2
```

SPI NOR is an MTD device:

```
/dev/mtd0        raw character device
/dev/mtdblock0   block wrapper, useful only for read-only filesystems
```

The partition table is usually not stored on the flash itself. It is described by:

- Device Tree fixed partitions, preferred,
- U-Boot `mtdparts`,
- kernel command line `mtdparts=...`.

For a product, use Device Tree fixed partitions so Linux and U-Boot agree.

35I.3 A realistic 16 MB layout

For a 16 MB SPI NOR, a practical layout is tight:

```
Offset      Size      Name        Content
0x000000    1 MB      bootloader  U-Boot image, or SPL + U-Boot
0x100000    128 KB    env         U-Boot environment
0x120000    128 KB    env-r       Redundant U-Boot environment
0x140000    6 MB      kernel      zImage
0x740000    128 KB    dtb         board DTB
0x760000    8 MB      rootfs      SquashFS root filesystem
0xF60000    640 KB    data        tiny config/log area, optional
```

This is not a universal layout. It is a teaching layout.

The key rules:

- Align partitions to erase-block boundaries.
- Leave room for kernel growth.
- Keep rootfs read-only and compressed.
- Do not put large logs in SPI NOR.
- Keep U-Boot environment away from kernel/rootfs partitions.

For a 32 MB NOR, give rootfs and recovery more room. For a 4 MB NOR, store only bootloader + recovery loader, and keep rootfs elsewhere.

35I.4 Device Tree fixed partitions

For QSPI NOR, your board DTS will contain a flash node under the QSPI controller. Exact controller node names vary by kernel and board.

The important part is the `fixed-partitions` block:

```
flash@0 {
    compatible = "jedec,spi-nor";
    reg = <0>;
    spi-max-frequency = <50000000>;
    #address-cells = <1>;
    #size-cells = <1>;

    partitions {
        compatible = "fixed-partitions";
        #address-cells = <1>;
        #size-cells = <1>;

        bootloader@0 {
            label = "bootloader";
            reg = <0x000000 0x100000>;
            read-only;
        };

        env@100000 {
            label = "env";
            reg = <0x100000 0x020000>;
        };

        env-redundant@120000 {
            label = "env-r";
            reg = <0x120000 0x020000>;
        };

        kernel@140000 {
            label = "kernel";
            reg = <0x140000 0x600000>;
        };

        dtb@740000 {
            label = "dtb";
            reg = <0x740000 0x020000>;
        };

        rootfs@760000 {
            label = "rootfs";
            reg = <0x760000 0x800000>;
        };

        data@f60000 {
            label = "data";
            reg = <0xF60000 0x0A0000>;
```

```
        };
    };
};
```

After Linux boots:

```
[root@pa-mini:~]# cat /proc/mtd
dev:    size   erasesize  name
mtd0: 00100000 00010000 "bootloader"
mtd1: 00020000 00010000 "env"
mtd2: 00020000 00010000 "env-r"
mtd3: 00600000 00010000 "kernel"
mtd4: 00020000 00010000 "dtb"
mtd5: 00800000 00010000 "rootfs"
mtd6: 000a0000 00010000 "data"
```

The partition numbers come from the DT order.

35I.5 Kernel config

The kernel needs SPI NOR, MTD, and the root filesystem format.

Example fragment:

```
CONFIG_MTD=y
CONFIG_MTD_BLOCK=y
CONFIG_MTD_SPI_NOR=y
CONFIG_SPI=y
CONFIG_SPI_IMX=y
CONFIG_SPI_FSL_QUADSPI=y
CONFIG_SQUASHFS=y
CONFIG_SQUASHFS_XZ=y
```

Use the controller option that matches your hardware:

Hardware path	Likely kernel area
ECSPI + SPI NOR	SPI controller + `spi-nor`
QSPI NOR	QSPI controller + `spi-nor`

If you store the rootfs in SPI NOR, build the flash driver and SquashFS support into the kernel, not as modules. The kernel needs them before rootfs is mounted.

35I.6 Buildroot rootfs for SPI NOR

For a small NOR rootfs, use SquashFS:

```
Filesystem images  --->
  [*] squashfs root filesystem
      Compression method = xz
```

Useful config symbols:

```
BR2_TARGET_ROOTFS_SQUASHFS=y
BR2_TARGET_ROOTFS_SQUASHFS_XZ=y
BR2_PACKAGE_MTD=y
```

Output:

```
output/images/rootfs.squashfs
```

A SquashFS rootfs is read-only. That is a feature here. Put writable data somewhere else:

- small JFFS2 partition in the same SPI NOR,
- external eMMC/SD `/data`,
- RAM-only tmpfs for stateless products.

For SPI NOR, avoid ext4 on `/dev/mtdblockN`. It is not a real block device with wear leveling.

35I.7 Flash from U-Boot

Load images over TFTP, then write them to SPI NOR.

Probe flash:

```
=> sf probe
```

Write kernel:

```
=> tftp 0x82000000 zImage
=> sf update 0x82000000 0x140000 ${filesize}
```

Write DTB:

```
=> tftp 0x82000000 imx6ull-your-board.dtb
=> sf update 0x82000000 0x740000 ${filesize}
```

Write rootfs:

```
=> tftp 0x82000000 rootfs.squashfs
=> sf update 0x82000000 0x760000 ${filesize}
```

`sf update` erases the needed sectors and writes the new data. If your U-Boot does not have `sf update`, use `sf erase` followed by `sf write`, but make sure erase length is sector-aligned.

Be careful with the bootloader region. During bring-up, keep a USB SDP or SD-card recovery path available before overwriting the flash region that contains U-Boot.

35I.8 Boot from SPI NOR

In U-Boot:

```
=> sf probe
=> sf read 0x82000000 0x140000 0x600000
=> sf read 0x83000000 0x740000 0x020000
=> setenv bootargs 'console=ttymxc0,115200 earlycon root=/dev/mtdblock5
rootfstype=squashfs ro'
=> bootz 0x82000000 - 0x83000000
```

Save:

```
=> setenv bootargs 'console=ttymxc0,115200 earlycon root=/dev/mtdblock5
rootfstype=squashfs ro'
=> setenv bootcmd 'sf probe && sf read 0x82000000 0x140000 0x600000 && sf read 0x83000000
0x740000 0x020000 && bootz 0x82000000 - 0x83000000'
=> saveenv
```

Why `mtdblock5`? In the example DT, the `rootfs` partition is the sixth partition, so it becomes `mtd5`, and the block wrapper is `/dev/mtdblock5`.

This is why fixed partition order matters.

35I.9 Kernel + DTB in SPI, rootfs elsewhere

A common product compromise:

```
SPI NOR:  U-Boot, env, kernel, DTB, recovery initramfs
eMMC:     main rootfs, data
```

Bootargs:

```
root=LABEL=rootfs rw rootwait
```

Boot flow:

```
Boot ROM -> U-Boot from SPI NOR
U-Boot -> kernel + DTB from SPI NOR
Kernel -> rootfs from eMMC
```

This gives fast, soldered boot firmware, while leaving the large rootfs on eMMC.

For many i.MX6ULL products, this is more practical than putting the whole OS in a 16 MB NOR.

35I.10 Writable data in SPI NOR

Small writable data can live in SPI NOR, but treat it carefully.

Options:

Option	Use
U-Boot env	Boot variables only
JFFS2	Tiny writable config partition
Raw MTD	Fixed records, counters, calibration
eMMC `/data`	Better for logs and databases

Do not write logs continuously to SPI NOR. NOR has erase-cycle limits, and log traffic can wear a small partition quickly.

35I.11 Key Notes

- SPI NOR offers eraseblock-based storage through MTD, not an eMMC-style block device. Filesystem and update choices must account for erase geometry and limited capacity.
- ROM bootability, U-Boot access and Linux access are separate capabilities. A working SPI driver does not prove that the fitted flash and wiring are a valid ROM boot source.
- Bootloader, environment, kernel, DTB and rootfs regions share one address space. Their offsets, sizes and erase boundaries must remain consistent across packaging and update tools.

- A small immutable rootfs can live in NOR while bulk or mutable data lives elsewhere. That split changes boot dependencies and recovery behavior.
- Environment saves and application writes consume erase cycles. Repeated metadata updates need a wear and power-failure strategy, not merely an available write command.

35I.12 Pitfalls

- **Board can use SPI flash, but ROM cannot boot from it.** Check boot straps, not only the schematic.
- **Wrong IVT offset.** Boot ROM image placement differs between SPI-NOR and QSPI-NOR. Chapter 7 explains the ROM offsets. Verify against your SoC and U-Boot image format.
- **Rootfs too large.** A normal glibc rootfs will not fit in 16 MB. Use BusyBox + SquashFS or keep rootfs on eMMC.
- **Using ext4 on MTD.** MTD is not a normal block device. Use SquashFS for read-only or JFFS2 for tiny writable areas.
- **Forgetting to erase before write.** NOR can change bits from 1 to 0 by programming, but needs erase to return 0 to 1.
- **Overlapping U-Boot env.** If rootfs overlaps the env sector, `saveenv` corrupts the filesystem.
- **Saving environment every boot.** Each `saveenv` erases a sector. Do not write U-Boot env as a normal boot counter unless you designed for wear.

Previous chapter: Chapter 35D - Bootable SD and eMMC image layout

Next chapter: Chapter 35E - Buildroot product board directory

Chapter 35E: Buildroot product board directory

Chapter 35 showed Buildroot. Chapter 35D showed the SD/eMMC image layout. Chapter 35I showed the SPI NOR layout. This chapter turns those pieces into a product-style board directory.

The goal is a tree like this:

```
buildroot/
  configs/pa_mini_defconfig
  board/myorg/pa-mini/
    rootfs-overlay/
    post-build.sh
    post-image.sh
    genimage.cfg
    linux.fragment
    busybox.fragment
```

That is the shape you can commit, review, rebuild, and hand to another engineer.

35E.1 What belongs in the board directory

File	Purpose
`configs/pa_mini_defconfig`	The saved Buildroot configuration
`rootfs-overlay/`	Files copied directly into the target rootfs
`post-build.sh`	Final rootfs edits before image creation
`post-image.sh`	Create SD/eMMC image after Buildroot creates filesystems
`genimage.cfg`	Partition/image description
`linux.fragment`	Kernel config additions
`busybox.fragment`	BusyBox applet/config additions

Keep product-specific files here. Do not edit `output/target/` by hand.

35E.2 Create the directory

```
$ cd ~/imx6ull/src/buildroot-2025.02.15
$ mkdir -p board/myorg/pa-mini/rootfs-overlay/etc/init.d
$ mkdir -p board/myorg/pa-mini/rootfs-overlay/data
```

Add a message of the day:

```
$ cat > board/myorg/pa-mini/rootfs-overlay/etc/motd <<'EOF'
Point Atom MINI i.MX6ULL
Buildroot product rootfs
EOF
```

Add a first boot script:

```
$ cat > board/myorg/pa-mini/rootfs-overlay/etc/init.d/S99hello <<'EOF'
#!/bin/sh
echo "hello from product overlay"
EOF
$ chmod +x board/myorg/pa-mini/rootfs-overlay/etc/init.d/S99hello
```

35E.3 Add a post-build script

`post-build.sh` runs after Buildroot has assembled `output/target/`, before the filesystem image is created.

```
$ cat > board/myorg/pa-mini/post-build.sh <<'EOF'
#!/bin/sh
set -eu

TARGET_DIR="$1"

mkdir -p "$TARGET_DIR/data" "$TARGET_DIR/etc"

cat > "$TARGET_DIR/etc/build-info" <<INFO
board=pa-mini
date=$(date -u +%Y-%m-%dT%H:%M:%SZ)
git=$(git -C "$BR2_EXTERNAL" rev-parse --short HEAD 2>/dev/null || echo unknown)
INFO

chmod 0644 "$TARGET_DIR/etc/build-info"
EOF
$ chmod +x board/myorg/pa-mini/post-build.sh
```

This is where you generate simple files. Keep it deterministic when making releases. For a final release, avoid `date` unless you intentionally want the image to differ every build.

35E.4 Add genimage

Create `board/myorg/pa-mini/genimage.cfg`:

```
image boot.vfat {
  vfat {
    files = {
      "zImage",
      "imx6ull-pa-mini.dtb"
    }
  }
  size = 128M
}

image sdcard.img {
  hdimage {
  }

  partition boot {
    partition-type = 0x0C
    bootable = true
    image = "boot.vfat"
    size = 128M
  }

  partition rootfs {
    partition-type = 0x83
    image = "rootfs.ext4"
    size = 512M
  }

  partition data {
    partition-type = 0x83
    size = 512M
  }
}
```

Add `post-image.sh`:

```
$ cat > board/myorg/pa-mini/post-image.sh <<'EOF'
#!/bin/sh
set -eu

BOARD_DIR="$(dirname "$0")"
GENIMAGE_TMP="${BUILD_DIR}/genimage.tmp"

rm -rf "$GENIMAGE_TMP"

genimage \
  --rootpath "$TARGET_DIR" \
  --tmppath "$GENIMAGE_TMP" \
  --inputpath "$BINARIES_DIR" \
  --outputpath "$BINARIES_DIR" \
  --config "$BOARD_DIR/genimage.cfg"
```

```
EOF
$ chmod +x board/myorg/pa-mini/post-image.sh
```

Enable host `genimage` in Buildroot:

```
BR2_PACKAGE_HOST_GENIMAGE=y
```

35E.5 Connect it in Buildroot config

Open menuconfig:

```
$ make menuconfig
```

Set:

```
System configuration  --->
  Root filesystem overlay directories = board/myorg/pa-mini/rootfs-overlay

System configuration  --->
  Custom scripts to run before creating filesystem images = board/myorg/pa-mini/post-
build.sh

Filesystem images  --->
  [*] ext2/3/4 root filesystem
      ext2/3/4 variant = ext4

Host utilities  --->
  [*] host genimage

System configuration  --->
  Custom scripts to run after creating filesystem images = board/myorg/pa-mini/post-
image.sh
```

Then save:

```
$ make savedefconfig BR2_DEFCONFIG=configs/pa_mini_defconfig
```

Now the product can be rebuilt with:

```
$ make pa_mini_defconfig
$ make -j$(nproc)
```

35E.6 Kernel and BusyBox fragments

Buildroot can merge kernel config fragments:

```
BR2_LINUX_KERNEL_CONFIG_FRAGMENT_FILES="board/myorg/pa-mini/linux.fragment"
```

Example `linux.fragment`:

```
CONFIG_DEVTMPFS=y
CONFIG_DEVTMPFS_MOUNT=y
CONFIG_OVERLAY_FS=y
CONFIG_BLK_DEV_INITRD=y
```

BusyBox fragments work similarly:

```
BR2_PACKAGE_BUSYBOX_CONFIG_FRAGMENT_FILES="board/myorg/pa-mini/busybox.fragment"
```

Example `busybox.fragment`:

```
CONFIG_ASH=y
CONFIG_MDEV=y
CONFIG_SWITCH_ROOT=y
CONFIG_FEATURE_EDITING=y
```

Fragments are better than hand-editing generated `.config` files.

35E.7 What to commit

Commit:

- `configs/pa_mini_defconfig`
- `board/myorg/pa-mini/rootfs-overlay/`
- `board/myorg/pa-mini/post-build.sh`
- `board/myorg/pa-mini/post-image.sh`
- `board/myorg/pa-mini/genimage.cfg`
- kernel and BusyBox fragments

Do not commit:

- `output/`
- `dl/`
- `host/`
- generated images

Those are build outputs or caches.

35E.8 Key Notes

- A product board directory is the reviewable source of image policy. Generated output is a build result, not the place to preserve product configuration.
- A rootfs overlay supplies static files, post-build processing prepares target contents, and post-image processing packages artifacts. Keeping these phases distinct makes dependencies easier to inspect.
- The image layout and boot configuration form one contract. A reproducible rootfs is still unusable if the bootloader, partition names or kernel arguments disagree with it.
- Relative paths, pinned inputs and configuration fragments make a build portable. Absolute host paths and edits under `output/target/` hide dependencies on one workstation.
- Build identity is useful evidence of provenance, not proof of hardware qualification or release approval.

35E.9 Pitfalls

- **Editing `output/target/`.** It feels fast but disappears on rebuild.
- **Putting generated files in overlay.** Use post-build for generated files.
- **Using absolute host paths.** Keep paths relative to the Buildroot tree or board directory.
- **Forgetting host genimage.** `post-image.sh` fails if `genimage` is not available.
- **Committing `output/`.** It is huge and not reproducible review material.

Previous chapter: Chapter 35I - Boot from SPI NOR flash

Next chapter: Chapter 35A - Ubuntu-base rootfs

Chapter 35A: Ubuntu-base rootfs as a peer to BusyBox/Buildroot

> **sysfs:** a kernel-generated filesystem under /sys that exposes devices, drivers, and attributes.

> **What:** an Ubuntu rootfs on the i.MX6ULL. Ubuntu-the-distro with GNOME is too heavy. We use **Ubuntu-base** instead, the same Debian-family userland your laptop has, in a 80 MB tarball that runs `apt` and `bash` natively. You unpack it, `chroot` into it via `qemu-user-static`, install packages from the host's network, then NFS-boot the target into it. **NFS:** Network File System, which lets the target mount a host directory over Ethernet during development. **rootfs:** root filesystem, the directory tree mounted at / that contains /bin, /etc, /dev, and libraries.
>
> **Why:** for projects where the developer's familiarity matters more than image footprint, Ubuntu-base wins. Glibc, full coreutils, systemd, an actual `bash`, all present, in exchange for ~80 MB on disk and ~30 MB RAM at idle vs. BusyBox's < 5 MB / < 10 MB.
>
> **Focus:** the **`qemu-user-static` + `chroot` trick**, installing armhf packages from x86_64 host into the armhf rootfs by transparently running ARM binaries on the host CPU through QEMU emulation. Apt-get doesn't notice and the workflow works.

35A.1 When this is the right answer

Reason	Buildroot wins	Ubuntu-base wins
Smallest possible image	✓	
Fastest boot (<2 s)	✓	
Compatible with team's `apt-get` muscle memory		✓
Easy to `apt install foo` for a quick fix in the field		✓

Reason	Buildroot wins	Ubuntu-base wins
Reproducible byte-for-byte builds	✓	
Long-term security backports for free		✓ (Ubuntu LTS)
Lowest RAM	✓	
Best for prototypes / dev boards		✓
Best for shipping products	✓ (usually)	

The headline feature is `apt install <anything>`, Ubuntu's ~100 000-package archive available, no recompilation. For early development this is hard to beat.

The headline cost is ~25× the disk and ~3× the RAM vs Buildroot. On i.MX6ULL with 512 MiB DRAM and an 8 GB eMMC, both are fine. On a 32 MiB device, only Buildroot fits.

> **Buildroot:** a configuration-driven build system that produces a complete root filesystem and related images.

35A.2 Get the rootfs tarball

```
$ cd ~/imx6ull
$ wget https://cdimage.ubuntu.com/ubuntu-base/releases/26.04/release/ubuntu-base-26.04-
base-armhf.tar.gz
$ mkdir ubuntu-rootfs
$ sudo tar -xzf ubuntu-base-26.04-base-armhf.tar.gz -C ubuntu-rootfs/
$ ls ubuntu-rootfs/
bin   dev  etc   home  lib    media  mnt  opt
proc  root  run   sbin  srv    sys    tmp  usr
var
```

That's already a minimal Ubuntu 26.04 LTS rootfs for ARM. Total size before extra packages is around a few dozen MB compressed and about 80 MB after extraction. Use `24.04` if you need the older LTS, same workflow.

Three useful Ubuntu LTS choices:

- **22.04 LTS**: standard security maintenance through May 2027.

- **24.04 LTS**: standard security maintenance through May 2029.
- **26.04 LTS**: standard security maintenance through May 2031.

Pick the most-recent LTS unless something specific requires older.

35A.3 Install qemu-user-static on the host

The trick that makes the whole flow work:

```
$ sudo apt install qemu-user-static binfmt-support
$ ls /usr/bin/qemu-arm-static
/usr/bin/qemu-arm-static
```

`qemu-user-static` runs one ARM binary at a time on your x86_64 host. `binfmt-support` registers it with the kernel via `binfmt_misc`. When the kernel sees `exec` of an ARM ELF, it transparently invokes `qemu-arm-static` to run it.

> **ELF:** Executable and Linkable Format, the standard Linux object and executable file format.

This means: inside the `chroot`, when you type `apt install nano`, `apt` is an ARM binary, `dpkg` is an ARM binary, every `.postinst` script's binary callouts are ARM, and they *all run on your x86_64 host* via QEMU emulation. The chroot doesn't know the difference. Neither do the binaries.

Copy the QEMU binary into the rootfs so the chroot has it:

```
$ sudo cp /usr/bin/qemu-arm-static ubuntu-rootfs/usr/bin/
```

35A.4 The mount-and-chroot script

Inspired by the PA workflow, but adapted. Save as `~/imx6ull/mount-ubuntu.sh`:

```
#!/bin/bash
set -e

ROOTFS=$HOME/imx6ull/ubuntu-rootfs

echo "Mounting host filesystems into $ROOTFS..."

sudo mount -t proc /proc      "$ROOTFS/proc"
sudo mount -t sysfs /sys      "$ROOTFS/sys"
sudo mount --bind /dev        "$ROOTFS/dev"
sudo mount --bind /dev/pts    "$ROOTFS/dev/pts"

# Copy host resolv.conf so DNS works in chroot
sudo cp /etc/resolv.conf "$ROOTFS/etc/resolv.conf"
```

```
# Drop into the rootfs
sudo chroot "$ROOTFS" /bin/bash
```

And the unmount counterpart `~/imx6ull/unmount-ubuntu.sh`:

```
#!/bin/bash
ROOTFS=$HOME/imx6ull/ubuntu-rootfs
echo "Unmounting..."
sudo umount "$ROOTFS/dev/pts" || true
sudo umount "$ROOTFS/dev"     || true
sudo umount "$ROOTFS/sys"     || true
sudo umount "$ROOTFS/proc"    || true
```

Make both executable:

```
$ chmod +x mount-ubuntu.sh unmount-ubuntu.sh
```

35A.5 Configure the rootfs from inside the chroot

```
$ ./mount-ubuntu.sh
# (host shell becomes ARM shell — transparently, via qemu-arm-static)

root@host:/# uname -m
armv7l                  # ← we are inside the chroot, claiming to be ARM
```

You are now inside a fake ARM machine. Whatever you run here runs on QEMU. Whatever you write to the filesystem is in the rootfs.

Set up apt sources

The default `sources.list` in `ubuntu-base` is empty. Add the official Ubuntu ports archive (ports.ubuntu.com hosts the armhf packages):

```
root@host:/# cat > /etc/apt/sources.list <<'EOF'
deb http://ports.ubuntu.com/ubuntu-ports resolute main restricted universe multiverse
deb http://ports.ubuntu.com/ubuntu-ports resolute-updates main restricted universe
multiverse
deb http://ports.ubuntu.com/ubuntu-ports resolute-security main restricted universe
multiverse
EOF
```

(`resolute` is the codename for 26.04. Use `noble` for 24.04 or `jammy` for 22.04.)

For users in China who find the official mirrors slow, the Tsinghua or USTC mirror is much faster:

```
deb http://mirrors.ustc.edu.cn/ubuntu-ports resolute main restricted universe multiverse
deb http://mirrors.ustc.edu.cn/ubuntu-ports resolute-updates main restricted universe
```

```
multiverse
deb http://mirrors.ustc.edu.cn/ubuntu-ports resolute-security main restricted universe
multiverse
```

Install essential packages

> **Privilege boundary:** $ means normal user. # or sudo means root and can change host or target state. After a privileged command, verify the expected device, service, or file appears before continuing. Roll back by undoing the config change or stopping the service you just enabled.

```
root@host:/# apt update
root@host:/# apt install -y \
    sudo vim openssh-server kmod net-tools \
    ifupdown iputils-ping rsyslog less htop \
    language-pack-en-base
```

That’s the minimum for: a non-root user (sudo), an editor, ssh in, kernel module loading, networking, ping, syslog. ~50 MB of additional packages.

For the curious, this is what runs:

1. `apt update`, downloads package metadata. Runs on QEMU. HTTP calls go to the host’s network.
2. `apt install`, for each package: download `.deb`, dpkg-extract files into `/`, run pre/post-install scripts. The scripts themselves are ARM binaries running under QEMU.

Set root password and create a user

```
root@host:/# passwd root
New password: <something>

root@host:/# adduser dev
# Creates /home/dev, sets password

root@host:/# usermod -aG sudo dev
```

Set hostname

```
root@host:/# echo pa-mini > /etc/hostname
root@host:/# cat > /etc/hosts <<'EOF'
127.0.0.1   localhost
127.0.1.1   pa-mini
EOF
```

Enable the serial console for systemd

Ubuntu uses systemd. We need a getty on `ttymxc0` for serial login:

```
root@host:/# systemctl enable serial-getty@ttymxc0.service
```

`systemctl` here just creates a symlink, it doesn't actually start anything (we're in a chroot). The symlink ensures the service is enabled on first boot of the target.

Done, exit and unmount

```
root@host:/# exit
# (back to host x86_64 shell)
$ ./unmount-ubuntu.sh
```

35A.6 Boot it from NFS

Export `~/imx6ull/ubuntu-rootfs/` over NFS (same as Ch 31, different rootfs path). In U-Boot:

> **MCU bridge:** Think of U-Boot like a much larger boot stub plus debug monitor: it initializes hardware, loads the next image, and gives you commands before Linux starts. **U-Boot:** the bootloader that initializes enough hardware to load and start the Linux kernel.

```
=> setenv bootargs 'console=ttymxc0,115200 earlycon \
                    root=/dev/nfs nfsroot=192.168.7.1:/home/you/imx6ull/ubuntu-
rootfs,vers=3,nolock,tcp \
                    ip=192.168.7.2:192.168.7.1:192.168.7.1:255.255.255.0::eth0:off \
                    rw rootwait'
=> run bootnet
```

After kernel boot:

```
[OK] Started Serial Getty on ttymxc0.

Ubuntu 26.04 LTS pa-mini ttymxc0

pa-mini login: dev
Password:
Welcome to Ubuntu 26.04 LTS (GNU/Linux 6.6.0 armv7l)
...

dev@pa-mini:~$ uname -a
Linux pa-mini 6.6.0 #1 SMP ... armv7l armv7l armv7l GNU/Linux

dev@pa-mini:~$ sudo apt install screen
[sudo] password for dev:
```

```
...

dev@pa-mini:~$ which python3
/usr/bin/python3

dev@pa-mini:~$ python3 -c 'print("hello from an ARM Ubuntu")'
hello from an ARM Ubuntu
```

A full Ubuntu shell. `apt install` works, `python3` is there, every command you're used to is there.

Boot time on i.MX6ULL: ~12-15 seconds from `bootz` to login prompt (vs ~3 s for BusyBox). Most of that is systemd's startup.

35A.7 Persistence, burning to eMMC

NFS-rooted Ubuntu is great for development but you eventually want it on the target's flash.

```
# On the host, pack the configured rootfs:
$ cd ubuntu-rootfs
$ sudo tar -czf ../ubuntu-rootfs.tar.gz .
$ cd ..

# Format an SD card or eMMC partition (assume /dev/sdb1 is the rootfs partition):
$ sudo mkfs.ext4 /dev/sdb1
$ sudo mount /dev/sdb1 /mnt/rootfs
$ sudo tar -xzf ubuntu-rootfs.tar.gz -C /mnt/rootfs/
$ sudo umount /mnt/rootfs

# Configure U-Boot to boot from eMMC:
# bootargs: root=/dev/mmcblk1p2 rw rootwait ...
```

The same rootfs works whether served over NFS or mounted from eMMC.

35A.8 Comparison: BusyBox vs Buildroot vs Ubuntu-base

	BusyBox by hand (Ch 31)	**Buildroot (Ch 35)**	**Ubuntu-base (Ch 35A)**
Initial rootfs size	~5 MB	~10 MB	~80 MB
RAM at idle	~10 MB	~15 MB	~30 MB
Boot to login	< 3 s	< 5 s	~12 s

	BusyBox by hand (Ch 31)	Buildroot (Ch 35)	Ubuntu-base (Ch 35A)
Customisation method	Edit `/etc/` directly	Overlay + defconfig	`apt install`
New package	Cross-compile + copy	menuconfig + rebuild	`apt install`
Reproducibility	Notes	Defconfig	Snapshot of `apt` state
Long-term security	You backport	Buildroot package version	Ubuntu LTS team
Best for	Learning, tiny	Most embedded products	Dev boards, prototypes

For our continuing work in this book, **Buildroot** is the default. **Ubuntu-base** is a great option when (a) you're spinning up a new dev board, (b) you need a weird package that's in Ubuntu but not Buildroot, or (c) you'd rather pay 70 MB and 10 seconds of boot for the comfort of `apt install`.

35A.9 Key Notes

- Ubuntu-base trades a small, tightly controlled image for familiar packages and a larger maintenance surface. The choice depends on product storage, memory and update policy, not on whether a shell can start.
- QEMU user-mode emulation executes ARM user-space binaries during host-side preparation. It does not emulate MINI hardware or qualify the target kernel, bootloader or peripheral drivers.
- A chroot changes the apparent filesystem root. Bind mounts, name resolution and host kernel services remain separate dependencies that need deliberate setup and cleanup.
- The rootfs architecture must match the target ABI. A working host-side package installation is not evidence that the target can boot that filesystem.
- Services, writable state and package updates need a product policy. An NFS-mounted development rootfs and a locally shipped image have different availability and persistence assumptions.

35A.10 Pitfalls

- **`qemu-arm-static` not copied into the rootfs.** Symptom: `chroot` fails with `Exec format error` because the kernel doesn't know how to run an ARM binary. Fix: copy `/usr/bin/qemu-arm-static` into `ubuntu-rootfs/usr/bin/`.
- **`binfmt-support` not installed on the host.** Same symptom. Fix: `apt install binfmt-support` and restart the service.
- **Forgetting `/etc/resolv.conf`.** Inside the chroot, `apt update` will fail with "Temporary failure resolving 'ports.ubuntu.com'". Fix: copy host's `/etc/resolv.conf` into the rootfs before `chroot`.
- **`mount --bind /dev` without `--bind /dev/pts`.** Symptom: `passwd` and other terminal-needing commands fail in the chroot. Fix: bind both.
- **Forgetting to unmount.** If you reboot the host without `unmount-ubuntu.sh`, the host's `/dev/pts/` is still bind-mounted under the rootfs. Later operations may misbehave. Always pair mount and unmount.
- **Choosing the wrong armhf.** Ubuntu publishes both `armhf` (32-bit, hard-float) and `arm64` (64-bit). I.MX6ULL is `armhf`. Downloading `arm64` and trying to run it on a 32-bit i.MX6ULL is a slow, confusing failure.
- **Mirror URL with HTTPS but `apt` can't validate certs.** Older Ubuntu base images may not have `ca-certificates` installed. Use `http://` mirrors initially, install `ca-certificates` first, then switch to `https://`.
- **`systemctl enable` in chroot is harmless but `systemctl start` is not.** Inside a chroot, `start` may try to manipulate cgroups or dbus and fail. Only use `enable` / `disable` for one-shot config in the chroot.

35A.11 Going deeper

- **`debootstrap`**: Debian's equivalent of `ubuntu-base.tar.gz`. Builds a fresh Debian rootfs from package archives. More flexible. More complex.
- **`multistrap`**: debootstrap with support for non-Debian package sources. Used by Yocto when assembling Debian-based images.

> **Yocto:** a metadata-driven build system for producing custom Linux distributions.

- **`schroot`**: manage chroot environments without writing your own mount scripts.
- **`Ubuntu Core`**: Ubuntu's official "embedded" variant. Uses snaps instead of apt. Immutable rootfs. Different philosophy from this chapter. Worth knowing about.

- **`Yocto-meta-ubuntu`** layer, combines Yocto-style builds with Ubuntu's package archive.

> Next chapter: **Chapter 35F: Your application as a service.** Now that you have several rootfs choices, we make the actual product application start, stop, log, and restart cleanly.

Chapter 35F: Your application as a service

An embedded Linux product is not finished when the shell boots. The product starts when your application starts automatically, logs somewhere useful, restarts if it crashes, and shuts down cleanly.

This chapter shows the three common service paths:

- BusyBox init script,
- systemd unit,
- Buildroot overlay packaging.

35F.1 The application contract

Write the app so the init system can manage it.

Good behavior:

- run in the foreground unless the service wrapper backgrounds it,
- print logs to stdout/stderr or syslog,
- exit nonzero on failure,
- handle `SIGTERM`,
- keep runtime data in `/data` or `/var`, not in `/usr`,
- read config from `/etc/myapp.conf` or `/data/myapp/config`.

Bad behavior:

- double-forking without telling the init system,
- silently creating files under `/`,
- ignoring termination signals,
- writing logs forever without rotation.

35F.2 A tiny example app

Save as `myapp.c`:

```
#include <signal.h>
#include <stdio.h>
#include <stdbool.h>
#include <unistd.h>

static volatile sig_atomic_t running = 1;

static void stop(int sig)
{
    (void)sig;
    running = 0;
}

int main(void)
{
    signal(SIGTERM, stop);
    signal(SIGINT, stop);

    puts("myapp: started");
    fflush(stdout);

    while (running) {
        puts("myapp: tick");
        fflush(stdout);
        sleep(5);
    }

    puts("myapp: stopping");
    return 0;
}
```

Build with the same target toolchain:

```
$ arm-none-linux-gnueabihf-gcc -O2 -Wall -o myapp myapp.c
```

Install it in the rootfs:

```
$ install -D -m 0755 myapp rootfs/usr/bin/myapp
```

35F.3 BusyBox init script

For BusyBox init, create `/etc/init.d/S99myapp`:

```
#!/bin/sh

case "$1" in
  start)
    echo "Starting myapp"
    start-stop-daemon -S -b -m -p /var/run/myapp.pid \
      -x /usr/bin/myapp -- >> /var/log/myapp.log 2>&1
    ;;
  stop)
    echo "Stopping myapp"
```

```
    start-stop-daemon -K -p /var/run/myapp.pid
    ;;
  restart)
    "$0" stop
    sleep 1
    "$0" start
    ;;
  *)
    echo "Usage: $0 {start|stop|restart}"
    exit 1
    ;;
esac
```

Make it executable:

```
$ chmod +x rootfs/etc/init.d/S99myapp
```

In a Buildroot overlay:

```
board/myorg/pa-mini/rootfs-overlay/etc/init.d/S99myapp
```

BusyBox runs scripts in lexical order if your `rcS` does that. `S99myapp` starts late, after filesystems and networking.

35F.4 systemd unit

On Ubuntu-base or a systemd Buildroot image, create `/etc/systemd/system/myapp.service`:

```
[Unit]
Description=My embedded application
After=network-online.target
Wants=network-online.target

[Service]
Type=simple
ExecStart=/usr/bin/myapp
Restart=always
RestartSec=5
StandardOutput=journal
StandardError=journal

[Install]
WantedBy=multi-user.target
```

Enable:

```
[root@pa-mini:~]# systemctl daemon-reload
[root@pa-mini:~]# systemctl enable --now myapp.service
[root@pa-mini:~]# systemctl status myapp.service
```

Read logs:

```
[root@pa-mini:~]# journalctl -u myapp.service -f
```

If the app exits, systemd restarts it.

35F.5 Config and data

Use this layout:

Path	Purpose
`/usr/bin/myapp`	executable, from the rootfs image
`/etc/myapp.conf`	default config, version-controlled
`/data/myapp/`	persistent runtime data
`/var/log/myapp.log`	logs, maybe tmpfs or persistent depending on product
`/run/myapp/`	pid files and sockets, tmpfs

Do not write to `/usr`. In production it may be read-only.

35F.6 Health checks

At minimum, add a status command:

BusyBox:

```
[root@pa-mini:~]# ps | grep myapp
[root@pa-mini:~]# tail -f /var/log/myapp.log
```

systemd:

```
[root@pa-mini:~]# systemctl is-active myapp.service
[root@pa-mini:~]# journalctl -u myapp.service -n 50
```

For a product, pair this with a watchdog in Part VIII. The service restarts the app. The watchdog restarts the board if the whole system gets stuck.

35F.7 Key Notes

- A service has a lifecycle contract: dependencies, readiness, shutdown, failure handling and ownership of its files. Installing an executable establishes only one part of that contract.
- The supervisor and application should have clear responsibilities. Foreground execution makes process ownership visible, while restart policy should not conceal a permanent configuration error.
- Configuration, persistent data and transient files belong in different locations. A read-only rootfs works only when the application's write behavior agrees with that layout.
- A running process is not necessarily a healthy service. Health should reflect useful progress, while graceful shutdown protects state and releases resources.
- Network availability can change after startup. Service ordering helps establish initial conditions, but the application still needs a recovery policy for later disconnections.

35F.8 Pitfalls

- **App daemonizes itself under systemd.** Use `Type=forking` only if the app really forks. Prefer foreground mode.
- **Logs disappear.** If `/var/log` is tmpfs, logs vanish at reboot. Decide intentionally.
- **App starts before network.** Add ordering (`After=network-online.target`) or make the app handle network loss.
- **Writing config to `/etc` on read-only rootfs.** Store runtime changes under `/data`.
- **No graceful shutdown.** Handle `SIGTERM`, close files, and flush data.

Previous chapter: Chapter 35A - Ubuntu-base rootfs

Next chapter: Chapter 35G - Networking, time, and logging

Chapter 35G: Networking, time, and logging

The rootfs boots. Your app starts. Now the normal field problems begin:

- It has no IP address.
- DNS does not work.
- Time is 1970.
- Logs vanish after reboot.
- The network comes up after your app already failed.

This chapter gives the practical checklist.

35G.1 Bring up Ethernet manually

On a small BusyBox rootfs:

```
[root@pa-mini:~]# ip link set eth0 up
[root@pa-mini:~]# ip addr add 192.168.1.50/24 dev eth0
[root@pa-mini:~]# ip route add default via 192.168.1.1
[root@pa-mini:~]# echo 'nameserver 8.8.8.8' > /etc/resolv.conf
```

Test:

```
[root@pa-mini:~]# ping -c 3 192.168.1.1
[root@pa-mini:~]# ping -c 3 8.8.8.8
[root@pa-mini:~]# ping -c 3 example.com
```

If the first works but the second fails, routing is wrong.

If the second works but the third fails, DNS is wrong.

35G.2 DHCP with BusyBox

BusyBox provides `udhcpc`.

```
[root@pa-mini:~]# udhcpc -i eth0
```

For boot, add to `/etc/init.d/rcS`:

```
ip link set eth0 up
udhcpc -i eth0 -q
```

`-q` exits after it gets a lease. Without it, `udhcpc` stays in the foreground unless scripted differently.

35G.3 Static IP config file

For a simple product, keep one config file:

```
# /etc/network.conf
IFACE=eth0
ADDR=192.168.1.50/24
GATEWAY=192.168.1.1
DNS=192.168.1.1
```

Read it from `rcS`:

```
if [ -f /etc/network.conf ]; then
    . /etc/network.conf
    ip link set "$IFACE" up
    ip addr add "$ADDR" dev "$IFACE"
    ip route add default via "$GATEWAY"
    echo "nameserver $DNS" > /etc/resolv.conf
else
    udhcpc -i eth0 -q
fi
```

Move the runtime-editable copy to `/data/network.conf` if rootfs is read-only.

35G.4 Hostname and `/etc/hosts`

Set:

```
echo pa-mini > /etc/hostname
```

And:

```
127.0.0.1   localhost
127.0.1.1   pa-mini
```

On boot:

```
hostname -F /etc/hostname
```

Some tools behave strangely when hostname is missing.

35G.5 Time

Many i.MX6ULL boards do not have a battery-backed RTC installed. Without network time, the system may boot at 1970.

Check:

```
[root@pa-mini:~]# date
```

Use BusyBox NTP:

```
[root@pa-mini:~]# ntpd -n -q -p pool.ntp.org
```

For boot:

```
ntpd -p pool.ntp.org
```

On Ubuntu-base, use systemd-timesyncd or chrony:

```
[root@pa-mini:~]# timedatectl status
[root@pa-mini:~]# systemctl status systemd-timesyncd
```

If the device has no internet, set time from:

- RTC,
- GPS,
- local NTP server,
- manufacturing tool,
- signed update metadata.

35G.6 Logging paths

Decide where logs live:

Path	Storage	Use
`/run/log`	tmpfs	boot-only logs
`/var/log`	tmpfs or persistent	system logs

Path	Storage	Use
`/data/log`	persistent data partition	field logs
`journalctl`	systemd journal	Ubuntu/systemd images

For BusyBox syslog:

```
syslogd -n -O /var/log/messages
```

For background:

```
syslogd -O /var/log/messages
klogd
```

If `/var/log` is tmpfs, logs are lost on reboot. That is okay for development. For field debugging, copy important logs to `/data/log`.

35G.7 Log rotation

Without rotation, logs fill storage.

Simple BusyBox rotation:

```
LOG=/data/log/messages

if [ -f "$LOG" ] && [ "$(wc -c < "$LOG")" -gt 1048576 ]; then
    mv "$LOG" "$LOG.1"
    : > "$LOG"
fi
```

Run it from cron or before starting syslog.

Buildroot can also include `logrotate`.

35G.8 Debug checklist

When the board says “network broken”:

```
ip link
ip addr
ip route
cat /etc/resolv.conf
ping -c 3 <gateway>
ping -c 3 8.8.8.8
```

```
ping -c 3 example.com
dmesg | grep -i eth
```

When time is wrong:

```
date
hwclock -r
timedatectl status 2>/dev/null || true
ps | grep ntp
```

When logs are missing:

```
mount | grep ' /var\| /data'
ps | grep 'syslog\|journald'
ls -lh /var/log /data/log 2>/dev/null
```

35G.9 Key Notes

- Link state, IP configuration, routing and DNS are separate layers. Success at one layer narrows a diagnosis but does not establish that the next layer works.
- Network configuration needs one clear owner. Competing static settings, DHCP clients and service scripts can overwrite addresses or resolver state.
- Wall-clock time supports certificates and useful logs, while elapsed-time measurements need a clock that is not changed by synchronization.
- Log retention is a storage policy. Tmpfs limits persistence, while flash-backed logs need capacity and wear controls.
- A service should cope with delayed or lost connectivity. One successful boot-time connection is not a lifetime availability guarantee.

35G.10 Pitfalls

- **App assumes network is ready.** Network may come up after init starts your app. Retry inside the app.
- **DNS file overwritten.** DHCP clients may replace `/etc/resolv.conf`.
- **No RTC.** TLS, package managers, and certificates fail if time is far wrong.
- **Persistent logs fill flash.** Rotate or cap them.
- **Logging too much to eMMC.** Flash has wear. Use tmpfs for noisy logs and persistent storage only for useful summaries.

Previous chapter: Chapter 35F - Your application as a service

Next chapter: Chapter 35B - Read-only rootfs and overlayfs

Chapter 35B: Read-only rootfs + overlayfs

> **sysfs:** a kernel-generated filesystem under /sys that exposes devices, drivers, and attributes.

> **What:** separate an immutable base system from writable runtime state. A **read-only root filesystem** and **overlayfs** can place changes in a per-boot tmpfs or a persistent upper layer. This reduces writes to the base image, but is not a blanket power-loss guarantee for the storage device or writable data. **rootfs:** root filesystem, the directory tree mounted at / that contains /bin, /etc, /dev, and libraries.
>
> **Why:** an unexpected power cut can interrupt a write or leave application state inconsistent. Keeping the base system immutable limits what ordinary runtime writes can change and gives recovery a stable starting point. Writable state still needs a home: tmpfs for disposable data, a separate data partition for persistent records, or an overlay for changes presented inside the root tree. Its durability and recovery behavior remain design decisions.
>
> **Focus:** the **three-tier model**, `lowerdir` (immutable rootfs), `upperdir` (where changes accumulate), `workdir` (overlay's scratch space). Once you understand those three, every overlayfs setup follows the same shape.

35B.1 The problem this solves

A standard development rootfs is mounted `rw` (read-write). A successful write can still leave data buffered in memory. If power disappears before that data reaches persistent storage, the next boot may see an older file or an incomplete update.

The filesystem and the application protect different things. Ext4's default journal protects metadata updates, not complete application transactions. A filesystem can recover its structure while a configuration file remains unusable. Journal replay and any boot-time filesystem check also depend on the filesystem and boot policy. A repair tool cannot reconstruct application data that never reached storage. See the ext4 journal documentation.

The risk belongs to both the filesystem and the application: a filesystem can remain mountable while a partially updated configuration is unusable.

One strategy is **not to write to the base rootfs at runtime**. That removes a class of interrupted updates to the base image. It does not remove the need for durable application transactions, storage qualification and a recoverable update process.

35B.2 Two patterns

There are two common ways to ship a read-only rootfs:

Pattern A, RO rootfs + tmpfs for writable paths

The simplest. Your application doesn't need writes to persist. It just needs *some* place to put `/var/log/`, `/tmp/`, and any temp state. Make those paths point to a tmpfs (RAM-backed, lost on reboot, exactly what you want for `/tmp/`):

```
/        ext4 ro       (root filesystem; immutable)
/tmp     tmpfs rw      (RAM; gone on reboot)
/var/log tmpfs rw      (RAM; gone on reboot)
/var/run tmpfs rw      (RAM; gone on reboot)
/run     tmpfs rw      (RAM; gone on reboot)
```

Add to `/etc/fstab`:

```
# device     mountpoint   type     options                   dump  pass
proc         /proc        proc     defaults                  0     0
sysfs        /sys         sysfs    defaults                  0     0
tmpfs        /tmp         tmpfs    defaults,nosuid,nodev     0     0
tmpfs        /var/log     tmpfs    defaults,size=8M          0     0
tmpfs        /var/run     tmpfs    defaults,size=2M          0     0
tmpfs        /run         tmpfs    defaults,size=2M          0     0
```

And set bootargs:

```
root=/dev/mmcblk1p2 ro rootwait    # ← note the "ro"
```

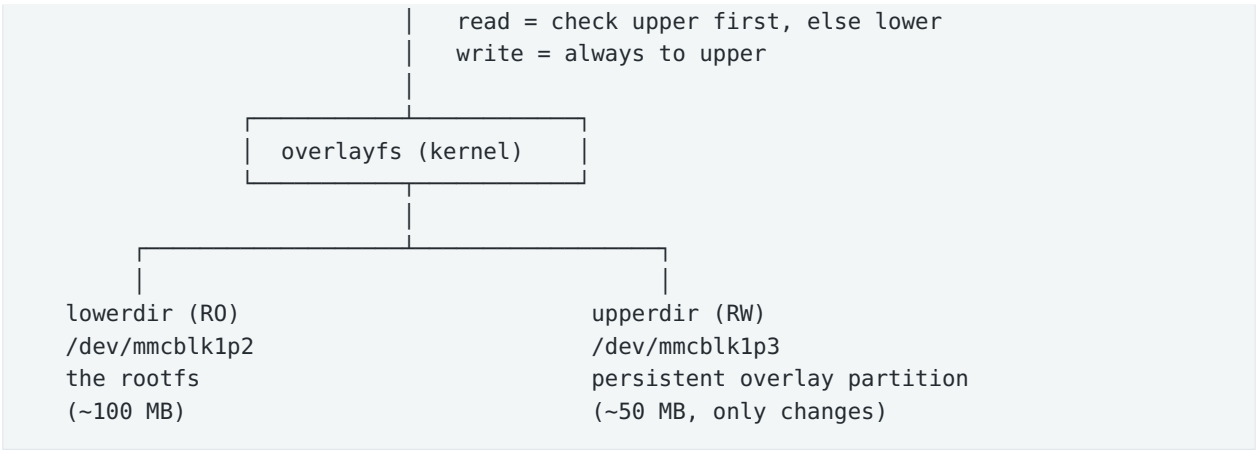
That's it. The rootfs is mounted RO. The tmpfs mounts give you writable paths.

The downside: anything in `/etc/` you'd want to modify (e.g., `/etc/network/interfaces` to change IP, `/etc/hostname` to set device serial) requires *rebuilding the rootfs* or some out-of-band update mechanism.

Pattern B, RO rootfs + overlay for selective persistence

If you want some changes to persist (e.g., the device's serial number, configuration, learned data), use **overlayfs**. The rootfs stays read-only. An overlay on top gives the illusion of writability and stores changes to a separate persistent partition.

```
                user-space sees a normal "writable /"
                           ▲
```



When the application writes to `/etc/hostname`, overlayfs copies `/etc/hostname` from `lowerdir` to `upperdir` ("copy-up"), then applies the write. From then on, reads of `/etc/hostname` come from `upperdir`. When `upperdir` is on a persistent partition, the change survives reboots.

The deeper trade-off:

- **More flexible** than Pattern A, you can change config files post-deployment.
- **More complex**: early boot must mount the overlay and handle an unreadable upper layer. Falling back to base defaults is safe only if the product can operate without the settings or identity stored in that layer.

The following examples show where each pattern puts its writable state. Pattern A has fewer boot dependencies, so it comes first.

35B.3 Pattern A in practice

Assume the rootfs we built in Chapter 31 or Chapter 35 lives on `/dev/mmcblk1p2` (the second eMMC partition. `p1` is FAT for boot, `p2` is ext4 root).

The rootfs build owns the `/etc/fstab` entries shown in §35B.2. Runtime services must agree with those mount boundaries.

Searching configuration files can help identify expected writers, but a text search cannot prove that every service respects a read-only root:

```
# What writes to /etc/ at runtime?
$ grep -rn "open.*WRONLY\|fopen.*\"[wa]" $TARGET_DIR/etc/ | head
```

Common culprits to fix:

- `/etc/resolv.conf` updated by `dhclient`, link to `/run/resolv.conf` (tmpfs) instead.
- `/etc/adjtime` updated by `hwclock`, same fix, or use `--noadjfile`.

- `/etc/machine-id` (systemd), needs to be persistent. Pre-generate at install time, then it's not written at boot.
- `/var/log/*`: ensure your tmpfs has enough space (`size=8M` in our fstab) and you have log rotation. Otherwise the tmpfs fills up.

The `ro` boot argument expresses the initial root-mount policy. Persistent environment changes are a separate bootloader design decision, not a requirement for this example:

> **MCU bridge:** Think of U-Boot like a much larger boot stub plus debug monitor: it initializes hardware, loads the next image, and gives you commands before Linux starts. **U-Boot:** the bootloader that initializes enough hardware to load and start the Linux kernel.

```
=> setenv bootargs 'console=ttymxc0,115200 earlycon root=/dev/mmcblk1p2 ro rootwait ...'
```

An illustrative shell transcript shows the distinction:

```
[root@pa-mini:~]# mount | head -1
/dev/mmcblk1p2 on / type ext4 (ro,relatime,...)

[root@pa-mini:~]# touch /etc/test
touch: /etc/test: Read-only file system

[root@pa-mini:~]# touch /var/log/test
[root@pa-mini:~]# ls /var/log/test
/var/log/test                          ← writable, in tmpfs
```

This output illustrates the intended write boundary: `/` rejects ordinary writes and the tmpfs paths accept disposable state. It does not establish power-loss resilience for the complete system.

35B.4 Pattern B, overlayfs

For changes that need to persist, this example overlays `/etc/`, `/var/`, and `/home/` on the read-only rootfs. A separate eMMC partition holds the upper layer.

Partition layout

> **Storage safety:** Before any command that names /dev/sdX, run lsblk -o NAME,SIZE,MODEL,TRAN,TYPE,MOUNTPOINTS. Verify the removable card by size and model, unmount its partitions, and stop if the path is not the target card. Writing the wrong /dev node can destroy the host disk.

```
/dev/mmcblk1p1   FAT  boot files (zImage, dtb, optional)
/dev/mmcblk1p2   ext4 rootfs (RO, ~200 MB)
/dev/mmcblk1p3   ext4 overlay upper + data (RW, ~500 MB)
```

These names describe an example target layout, not disks to format on the host. Provisioning must identify the actual medium and separate image installation from the runtime mounting policy.

Initial population

The installed base contains the rootfs and its mount points. The persistent partition contains an upper/work directory pair for each overlaid tree: `upper-etc` with `work-etc`, and equivalent pairs for `var` and `home`. Each work directory starts empty and shares a filesystem with its upper directory. Data kept outside the overlay has its own retention policy.

Boot-time overlay setup

The overlay must exist before services start using the affected paths. An initramfs can establish those mounts before handing control to the real rootfs. A late overlay risks hiding files that a service has already opened.

The following `/init` fragment illustrates that handoff. It assumes the layout and directories described above. A production implementation also needs checked mount results, a recovery branch and device discovery appropriate to the actual board:

```
#!/bin/sh
# Early init: set up overlay, then exec real /sbin/init

# Mount essential virtual filesystems
mount -t proc      none /proc
mount -t sysfs     none /sys
mount -t devtmpfs  none /dev

# Mount the RO rootfs
mkdir -p /rofs
mount -o ro /dev/mmcblk1p2 /rofs

# Mount the overlay upper partition
mkdir -p /overlay
mount /dev/mmcblk1p3 /overlay

# Build an overlay for /etc, /var, /home
for d in etc var home; do
    mkdir -p /merged/$d
    mount -t overlay overlay \
        -o lowerdir=/rofs/$d,upperdir=/overlay/upper-$d,workdir=/overlay/work-$d \
        /merged/$d
done

# Build the merged root: lower = rofs, upper = merged dirs
mkdir -p /merged-root
```

```
mount --bind /rofs /merged-root

# Bind-mount the overlayed subdirs over the read-only rootfs
mount --bind /merged/etc  /merged-root/etc
mount --bind /merged/var  /merged-root/var
mount --bind /merged/home /merged-root/home

# Move the virtual filesystems into the new root.
mkdir -p /merged-root/proc /merged-root/sys /merged-root/dev
mount --move /proc /merged-root/proc
mount --move /sys  /merged-root/sys
mount --move /dev  /merged-root/dev

# Exec the real init
exec switch_root /merged-root /sbin/init
```

That `/init` is small but does the whole dance: mount RO rootfs, mount overlay storage, overlay-mount the writable subdirs, move `/proc /sys /dev`, switch root, exec real init.

A simpler approach (no initramfs) that works for some setups: a small `S00-overlay` script in `/etc/init.d/` that runs *before* `S01-mountall` (BusyBox runs them alphabetically). But mounting overlay over a partial filesystem is fragile. Initramfs is cleaner.

From inside the running system

```
[root@pa-mini:~]# mount | grep overlay
overlay on /etc type overlay (rw,lowerdir=/rofs/etc,upperdir=/overlay/upper-etc,workdir=/
overlay/work-etc)
overlay on /var type overlay (rw,lowerdir=/rofs/var,upperdir=/overlay/upper-var,workdir=/
overlay/work-var)
overlay on /home type overlay (rw,lowerdir=/rofs/home,upperdir=/overlay/upper-
home,workdir=/overlay/work-home)

[root@pa-mini:~]# touch /etc/test
[root@pa-mini:~]# ls /overlay/upper-etc/
test
```

You wrote to `/etc/test`. The kernel copy-up'd to `/overlay/upper-etc/test`. The original `/rofs/etc/` is untouched.

Reboot. `/etc/test` survives. The RO rootfs is unchanged.

35B.5 Power-loss failure boundaries

Imagine the product updating its network settings when power disappears. The base image may still be usable, but an incomplete setting in the upper layer can hide its default. Recovery therefore needs more than a read-only mount.

State	Design responsibility
Base image	Keep runtime changes out of the shipped system and retain a recovery path if the image cannot be read.
Tmpfs state	Treat its contents as disposable and recreate them during startup.
Persistent upper layer	Define application commit boundaries and a response to incomplete or unreadable state.
Storage device	Qualify the actual medium and power behavior. Separate partitions are not independent physical devices.

There is no universal “last two seconds” loss window or failure percentage. Durability depends on the application, filesystem and storage stack. Synchronization must be explicit when persistence matters, with errors handled rather than ignored. Overlayfs does not make several application writes one transaction. Its durability documentation explains that boundary.

Even `ro` needs context: ext4 can replay its journal during a read-only mount. Suppressing replay is not a general repair strategy for an inconsistent filesystem. See the ext4 mount documentation.

Power-loss qualification belongs to a controlled hardware validation plan, not a required reading exercise. This chapter provides no measured MINI power-cut results.

35B.6 Factory reset

With Pattern B, an empty upper layer exposes the base defaults again. That makes reset possible without reflashing the base image, but deciding what to discard is a product responsibility.

User settings may be disposable. Device identity, credentials, calibration and retained records may not be. Keep those lifetimes separate instead of putting everything into one erasable layer.

A recovery environment can prepare replacement upper/work directories while the overlay is unmounted. Deleting backing files under a live overlay is not valid: the kernel documents undefined behavior for such changes.

The reset design also needs a recoverable state if power disappears halfway through, and a path that works when the old upper layer cannot be mounted. A button gesture requests this policy. It does not, by itself, implement it or promise a fixed reset time.

35B.7 Key Notes

- An immutable rootfs separates the shipped system from changing data. Read-only mounting reduces exposure to interrupted writes to that filesystem, but does not make the storage device or the entire product immune to power loss.
- Tmpfs gives disposable writable state at the cost of RAM. A persistent data partition or overlay retains changes, so its consistency, capacity and wear remain part of the design.
- Overlayfs combines a lower filesystem with an upper layer and work directory. The work and upper directories must share a filesystem, and the apparent merged tree can hide which layer owns a change.
- Factory reset is a data-lifecycle decision: disposable settings may be cleared while device identity, credentials and records may need to survive.
- Power-loss resilience requires evidence for the actual hardware, filesystem and application write pattern. A fixed number of restarts is not a universal durability guarantee.

35B.8 Pitfalls

- **`/etc/resolv.conf` and `/etc/adjtime`.** Both written at runtime in default Ubuntu/Debian setups. Either symlink them to `/run/` (tmpfs) or use overlay. If they end up trying to write to a RO mount, things like DHCP and NTP silently misbehave.
- **systemd's `/etc/machine-id`.** Generated on first boot, written to `/etc/`. On RO root, systemd may regenerate it every boot, breaking journal continuity. Fix: pre-generate at flash time.
- **Forgetting `ro` in `bootargs`.** Without `ro` in `bootargs` the whole scheme is defeated. Verify with `mount | head -1` after boot.
- **Overlay `workdir` must be on the same filesystem as `upperdir`.** Different filesystems for `workdir` and `upperdir` is an immediate mount failure.
- **Tmpfs filling up.** `/var/log` on tmpfs without log rotation, plus a chatty daemon, eats your RAM. Set `size=` explicitly and use `logrotate` (or just `> /var/log/messages` from cron).

- **Apps writing to `/etc/` expecting persistence.** If your app does `fopen("/etc/myapp.conf", "w")` to save settings, with Pattern A those settings are lost on reboot. Either use Pattern B (overlay) or move the writable file to `/data/` (a real persistent partition).
- **`switch_root` missing.** The initramfs must include a `switch_root` applet, usually from BusyBox. Without it, the overlay setup works but cannot hand off to the real rootfs.

35B.9 Going deeper

- **`Documentation/filesystems/overlayfs.rst`** in the kernel tree, the canonical reference.
- **`man 8 mount.overlay`**: overlay mount options in depth.
- **`erofs`**: Enhanced Read-Only File System, an alternative to ext4-RO with better compression. Used in modern Android.
- **`squashfs`**: another RO filesystem, slower but more compact than erofs. Common on initramfs images.
- **`A/B partition schemes`**: pair this chapter with Ch 63 (Field updates). Two RO rootfs partitions, switch atomically on update.

> Next chapter: **Chapter 35H: Rootfs security basics.** A read-only rootfs helps reliability. Before containers or release images, we remove the development shortcuts that should not ship.

Chapter 35H: Rootfs security basics

This chapter is not a full security book. It is the minimum checklist before a rootfs leaves the lab.

The goal:

- no empty root password,
- no unnecessary network service,
- app does not run as root unless it must,
- secrets are not baked into public images,
- writable data is separated from immutable system files.

35H.1 Development settings are not release policy

During bring-up, we often do this:

```
root::0:0:root:/root:/bin/sh
```

That means root has no password. Fine for a closed lab board. Bad for anything connected to a network.

Before release:

- set a password or lock root,
- prefer SSH keys,
- create a non-root service user,
- disable unused gettys and network daemons.

35H.2 Lock root login

In `/etc/shadow`, a locked root account starts with `!`:

```
root:!:19000:0:99999:7:::
```

Or on target:

```
[root@pa-mini:~]# passwd -l root
```

If you need emergency root access, use a recovery mode that requires physical access.

35H.3 SSH keys, not passwords

For Dropbear or OpenSSH, install an authorized key:

```
mkdir -p /home/dev/.ssh
chmod 700 /home/dev/.ssh
cat > /home/dev/.ssh/authorized_keys <<'EOF'
ssh-ed25519 AAAA... dev@host
EOF
chmod 600 /home/dev/.ssh/authorized_keys
chown -R dev:dev /home/dev/.ssh
```

Disable password login if the product allows it.

For OpenSSH:

```
PasswordAuthentication no
PermitRootLogin no
```

For Dropbear, pass options from its init script. The exact flags depend on the Buildroot Dropbear version, so check:

```
dropbear -h
```

35H.4 Run the app as a user

Create a service user:

```
myapp:x:200:200:myapp:/var/empty:/sbin/nologin
```

In `/etc/group`:

```
myapp:x:200:
```

BusyBox init script:

```
start-stop-daemon -S -b -c myapp -x /usr/bin/myapp
```

systemd unit:

```
[Service]
User=myapp
Group=myapp
ExecStart=/usr/bin/myapp
```

If the app needs GPIO, I2C, SPI, or video access, give access to that specific device or group. Do not run everything as root just because one file needed permission.

35H.5 File permissions

Useful defaults:

```
/usr/bin/myapp        0755 root:root
/etc/myapp.conf       0644 root:root
/data/myapp/          0750 myapp:myapp
/data/myapp/secret    0600 myapp:myapp
```

Check:

```
find / -perm -4000 -type f 2>/dev/null
find / -writable -type d 2>/dev/null | head
```

Setuid binaries deserve review. World-writable directories should be rare.

35H.6 Secrets

Do not bake these into a public rootfs image:

- private SSH keys,
- API tokens,
- customer certificates,
- production passwords,
- signing keys.

Better places:

- provision during manufacturing,
- store under `/data/secure/`,
- derive from a hardware unique key if available,
- use a secure element or TPM for private keys.

At minimum, each shipped board should not share the same private identity.

35H.7 Reduce services

List listening sockets:

```
ss -lntup
```

On BusyBox-only systems without `ss`, use:

```
netstat -lntup
```

Ask for each service:

- Why does this need to listen?
- Is it bound to all interfaces?
- Does it need authentication?
- Can it be disabled in production?

Common development-only services:

- SSH with password login,
- telnet,
- FTP,
- debug HTTP servers,
- gdbserver,
- unrestricted MQTT broker.

35H.8 Read-only rootfs helps security

Chapter 35B used read-only rootfs for power-loss safety. It also helps security:

- malware cannot easily modify `/usr/bin`,
- accidental writes fail early,
- factory reset can clear data without reflashing system files.

But read-only rootfs is not enough. If `/data` is writable and the app executes files from `/data`, attackers can still persist. Mount data with conservative options when possible:

```
LABEL=data  /data  ext4  defaults,noatime,nodev,nosuid  0  2
```

Use `noexec` only if your product never runs programs from `/data`.

35H.9 Release checklist

- Root password locked or intentionally set.
- SSH root login disabled.
- Password SSH disabled if keys are available.
- App runs as non-root.
- `/data` permissions reviewed.
- No private keys in the image.
- `ss -lntup` shows only expected services.
- Read-only rootfs tested.
- Factory reset does not erase device identity unless intended.

35H.10 Key Notes

- Security begins with authority: which account owns a service, which files it can change, and which interfaces it exposes. Running everything as root erases useful boundaries.
- Device identity and credentials have a different lifecycle from ordinary configuration. Provisioning, backup, update and factory reset must agree on what survives.
- An image shared across a fleet should not contain a shared private credential. Public build artifacts and secret provisioning belong to different trust domains.
- Removing unnecessary services reduces exposure, but an immutable rootfs does not replace authentication, authorization or an update process.
- Development convenience settings are not release policy. The relevant question is whether each privilege and listening service has a product purpose and an accountable owner.

35H.11 Pitfalls

- **One shared password across all boards.** This becomes a fleet-wide secret.
- **Private keys in Git.** Never put production keys in the repository.
- **App runs as root forever.** Convenient in the lab, expensive later.
- **Factory reset wipes identity.** Serial number, MAC address, and certificates often need to survive.

- **Security added after release.** It is much harder after field deployment.

Previous chapter: Chapter 35B - Read-only rootfs and overlayfs

Next chapter: Chapter 35C - Container runtimes on embedded

Chapter 35C: Container runtimes on embedded

> **What:** running OCI containers on an i.MX6ULL with Podman. By the end you'll have an Alpine Linux container running a small Python web server, accessing the host's GPIO from inside the container. **MCU bridge:** Think of Linux GPIO like the same pin set/reset block you used on STM32, but accessed through a kernel subsystem that owns numbering, direction, interrupts, and user-space exposure. **GPIO:** General-Purpose Input/Output, a pin controlled as a digital input, output, or interrupt source.
>
> **Why:** modern embedded products separate "the base system" (kernel + bootloader + minimal rootfs, updated rarely) from "the application" (one or more containers, updated whenever a customer-facing change is needed). Containers give you reproducible app deployment, image-based updates, and the ability to roll back in seconds. The cost is some RAM and some complexity. The benefit is a deployment story that scales from one device or a fleet of a million. **MCU bridge:** Think of the rootfs as the firmware image's file-backed runtime environment. On an MCU you link everything into flash. On Linux, programs and config live in this mounted tree. **rootfs:** root filesystem, the directory tree mounted at / that contains /bin, /etc, /dev, and libraries.
>
> **Focus:** Three kernel features make containers work: namespaces (process isolation), cgroups (resource limits), and overlayfs (storage). All three have been in mainline Linux for years. You just need them turned on in `.config`.

35C.1 When containers make sense on embedded

Reason	Yes	No
App + base system have different update cadences	✓	
Single product variant, single binary		✓
Multiple apps that mustn't interfere	✓	

Reason	Yes	No
Need to ship third-party software unmodified	✓	
Need 100 % minimum disk footprint		✓
Need < 50 ms cold-start latency		✓
Need to roll back app-only without touching kernel	✓	

For a single-purpose appliance with one in-house binary, containers are overhead. For a smart-camera platform that ships one base image to thousands of customers, with each customer running 1-5 of *their* applications on top, containers are how you keep your sanity.

The i.MX6ULL is on the small side for containers, but it's tractable: a single Alpine-based container costs ~30 MB additional disk and ~15 MB RAM at idle.

35C.2 What containers actually are

A "container" is just a Linux process group with three things layered on:

1. **Namespaces**: The kernel maintains separate "views" per namespace type: PID, network, mount, UTS (hostname), IPC, user, cgroup. A process in a PID namespace sees only processes in that namespace. It thinks it's PID 1.
2. **Cgroups (control groups)**: Resource limits per group: CPU, memory, IO. Enforced by the kernel.
3. **Overlay filesystem**: The container's root is an overlayfs over the image's read-only layers + a writable upper. Same `overlayfs` from Chapter 35B.

There is no container-runtime magic. A container is a process with namespaces, cgroups, and a careful mount setup. Docker, Podman, containerd, and CRI-O all do the same thing. They differ in UI, daemons, and ecosystem.

For embedded, **Podman** wins:

- **Daemonless.** Docker has a privileged daemon that runs as root and listens on a socket. Podman has no daemon, `podman run` directly spawns the container process. One less moving part.

- **Rootless mode.** Containers can run as a normal user, using user namespaces. Better security story than Docker.
- **Drop-in CLI-compatible with Docker.** `podman pull`, `podman run`, `podman ps` all work like `docker pull`, etc.

35C.3 Kernel configuration

The kernel features containers need. Check your `.config`:

```
CONFIG_NAMESPACES=y
CONFIG_UTS_NS=y
CONFIG_IPC_NS=y
CONFIG_USER_NS=y
CONFIG_PID_NS=y
CONFIG_NET_NS=y
CONFIG_CGROUPS=y
CONFIG_MEMCG=y
CONFIG_CPUSETS=y
CONFIG_CGROUP_DEVICE=y
CONFIG_CGROUP_FREEZER=y
CONFIG_CGROUP_PIDS=y
CONFIG_CGROUP_SCHED=y
CONFIG_CGROUP_NET_PRIO=y
CONFIG_BRIDGE=y                # for the container network bridge
CONFIG_NETFILTER=y
CONFIG_NETFILTER_XT_MATCH_ADDRTYPE=y
CONFIG_NF_CONNTRACK=y
CONFIG_OVERLAY_FS=y
CONFIG_SECCOMP=y
CONFIG_VETH=y                  # virtual ethernet pair for container networking
```

`imx_v6_v7_defconfig` enables most of these. `CONFIG_USER_NS` is sometimes off for size and needs flipping on.

After confirming and rebuilding the kernel:

```
[root@pa-mini:~]# zgrep -E "CONFIG_(NAMESPACES|CGROUPS|OVERLAY_FS|USER_NS)" /proc/
config.gz
CONFIG_NAMESPACES=y
CONFIG_USER_NS=y
CONFIG_CGROUPS=y
CONFIG_OVERLAY_FS=y
```

(If `/proc/config.gz` is empty, your kernel was built without `CONFIG_IKCONFIG_PROC=y`. The check still works by inspecting `.config` on the build host.)

35C.4 Install Podman

On a Buildroot rootfs, enable `BR2_PACKAGE_PODMAN`. On Ubuntu-base:

> **Buildroot:** a configuration-driven build system that produces a complete root filesystem and related images.

```
$ ./mount-ubuntu.sh   # chroot in
root@host:/# apt update
root@host:/# apt install -y podman
```

Verify:

```
[root@pa-mini:~]# podman --version
podman version 4.3.1

[root@pa-mini:~]# podman info | head
host:
  arch: arm
  buildahVersion: 1.28.2
  cgroupManager: systemd        ← or "cgroupfs" for non-systemd setups
  cgroupVersion: v2
  ...
```

If `cgroupManager` is shown but errors are reported, you may need to switch to `cgroupfs`. On a BusyBox rootfs:

```
cat > /etc/containers/containers.conf <<'EOF'
[engine]
cgroup_manager = "cgroupfs"
events_logger = "file"
EOF
```

35C.5 Pull and run a first container

```
[root@pa-mini:~]# podman pull docker.io/library/alpine:latest
Trying to pull docker.io/library/alpine:latest...
Getting image source signatures
Copying blob sha256:abcdef... done
Copying config sha256:fedcba... done
Writing manifest to image destination
abcdefabcdef...

[root@pa-mini:~]# podman run --rm alpine sh -c "echo hello from $(uname -m)"
hello from armv7l
```

A real Alpine Linux 3.x container, pulled, run, exited, cleaned up, in roughly 30 seconds plus download time. Alpine images for armv7 are ~3-5 MB compressed, ~10 MB on disk.

Run it interactively:

```
[root@pa-mini:~]# podman run --rm -it alpine sh
/ # cat /etc/os-release
```

```
NAME="Alpine Linux"
ID=alpine
VERSION_ID=3.19.1
...
/ # apk add python3
/ # python3 -c 'print("hi")'
hi
/ # exit
```

You just ran Alpine's apk inside a container running on Ubuntu (or BusyBox), all on top of the same kernel. From inside, it looks like a complete Alpine system.

35C.6 Container that talks to GPIO

The interesting case for embedded: a container that can access host hardware.

Goal: a container running a Python script that toggles the LED via `/sys/class/leds/led0/brightness`.

`Dockerfile`:

```
FROM alpine:latest

RUN apk add --no-cache python3

COPY blink.py /blink.py

CMD ["python3", "/blink.py"]
```

`blink.py`:

```
#!/usr/bin/env python3
import time

LED = "/sys/class/leds/led0/brightness"

while True:
    with open(LED, "w") as f:
        f.write("1")
    time.sleep(0.5)
    with open(LED, "w") as f:
        f.write("0")
    time.sleep(0.5)
```

Build:

```
[root@pa-mini:~]# podman build -t led-blinker .
```

Run, with `/sys/class/leds/` bind-mounted into the container:

```
[root@pa-mini:~]# podman run --rm -v /sys/class/leds:/sys/class/leds:rw led-blinker
```

The LED blinks. Inside the container, the Python script writes to `/sys/class/leds/led0/brightness`. Thanks to the bind mount, those writes hit the host's actual sysfs.

> **sysfs:** a kernel-generated filesystem under /sys that exposes devices, drivers, and attributes.

A more careful invocation passes only the *specific* device, not the entire sysfs hierarchy:

```
[root@pa-mini:~]# podman run --rm \
    -v /sys/class/leds/led0:/sys/class/leds/led0:rw \
    led-blinker
```

That gives the container access to *only* `led0`'s files. Other LEDs (and the rest of `/sys/`) remain invisible.

For more privileged access patterns:

- **`--device /dev/i2c-0`** to expose `/dev/i2c-0` to the container.
- **`--cap-add SYS_RAWIO`** to grant a capability the container needs.
- **`--privileged`** (last resort) to give the container full host access. Bad practice except for development.

35C.7 Storage and rootfs layout

Podman stores images and containers under `/var/lib/containers/` (rootful) or `~/.local/share/containers/` (rootless). On embedded systems where `/var/` may be tmpfs (Ch 35B), redirect this:

```
$ cat > /etc/containers/storage.conf <<'EOF'
[storage]
driver = "overlay"
runroot = "/run/containers/storage"
graphroot = "/data/containers/storage"
EOF
```

`graphroot` should be on persistent storage. `runroot` (the per-run temp) on tmpfs is fine.

35C.8 When *not* to use containers

Be honest about the costs:

- **RAM.** Each container adds ~10-20 MB at minimum. On 512 MB i.MX6ULL with three containers, that's 50 MB just to *be* in containers. Not always a problem. Sometimes prohibitive.
- **Boot time.** `podman.service` plus container-pull-and-start adds 2-5 seconds.
- **Complexity.** "Why isn't my hardware working?" answers gain a layer: "Is it the device driver? The bind mount? The cgroup device controller? The user-namespace mapping?"
- **Determinism.** A pulled container's exact bytes depend on the registry server. For a closed-supply-chain product, you'd mirror the registry or build images locally and bake them into the rootfs.

For one-purpose appliances, just ship a binary. For dynamic product lines that update frequently, containers earn their cost.

35C.9 Production patterns

A few real-world patterns:

Pre-loaded image, no registry pulls

For products without reliable internet, you don't `podman pull` on every boot, you bake the image into the rootfs at build time:

```
# At build time on the host:
$ podman save -o myapp.tar localhost/myapp:v1.0
$ cp myapp.tar $TARGET_ROOTFS/opt/preloaded-images/

# At first boot on the target:
[root@pa-mini:~]# podman load -i /opt/preloaded-images/myapp.tar
[root@pa-mini:~]# podman run -d --restart=always myapp:v1.0
```

Quadlet systemd units

For systemd-based products, prefer **Quadlet**. A Quadlet file is a small declarative file that systemd turns into a Podman service.

Save this as `/etc/containers/systemd/myapp.container`:

```
[Unit]
Description=My application container
Wants=network-online.target
After=network-online.target
```

```
[Container]
Image=localhost/myapp:v1.0
ContainerName=myapp
Volume=/data/myapp:/data:rw

[Service]
Restart=always
RestartSec=5

[Install]
WantedBy=multi-user.target
```

Then enable it:

```
[root@pa-mini:~]# systemctl daemon-reload
[root@pa-mini:~]# systemctl enable --now myapp.service
[root@pa-mini:~]# systemctl status myapp.service
```

The systemd unit handles auto-start, dependencies, log capture, and restart-on-failure. Older examples use `podman generate systemd`; that command still exists, but Podman now marks it deprecated and recommends Quadlet for new systems.

Updates via image swap

```
# New version available:
podman pull registry.example.com/myapp:v1.1
podman stop myapp
podman rm myapp
podman run -d --name myapp --restart=always registry.example.com/myapp:v1.1

# If it crashes immediately, return to the previous tag:
podman stop myapp || true
podman rm myapp || true
podman run -d --name myapp --restart=always registry.example.com/myapp:v1.0
```

Image swap is atomic at the container level: either the new image is running or the old one is. Much cleaner than “extract a tarball into /opt/myapp/” updates.

35C.10 Key Notes

- Containers package applications separately from the base system, but share its kernel. Application rollback and kernel or bootloader rollback are different update boundaries.
- Namespaces shape what a process can see, cgroups limit resource use, and container storage layers manage filesystem changes. None substitutes for a hardware isolation boundary.

- Device access is an explicit authority grant. A bind mount or privileged mode can expose host resources, so the application interface should be narrower than the whole host filesystem.
- Images, writable application data and transient state have different lifetimes. Persistent storage placement matters as much as the runtime's ability to launch a process.
- Container deployment is worthwhile only when its update and dependency benefits justify the memory, storage and operational complexity on the target.

35C.11 Pitfalls

- **`USER_NS` disabled.** Symptom: Podman rootless mode fails with "no subuid map". Fix: rebuild kernel with `CONFIG_USER_NS=y`.
- **cgroup v1 vs v2 mismatch.** Some older systemd setups expect v1 hierarchies. Podman 4.x prefers v2. Force v2 in the kernel cmdline: `cgroup_no_v1=all systemd.unified_cgroup_hierarchy=1`.
- **Network bridge not created.** Without `CONFIG_BRIDGE` and `CONFIG_VETH`, containers have no network. Symptom: container can't reach the internet. Fix: kernel config.
- **`/var/lib/containers/` on tmpfs.** Loses every image on reboot. Always redirect `graphroot` to persistent storage.
- **Trying to `podman pull` from a registry behind a corporate proxy.** Set `HTTPS_PROXY=...` in `/etc/containers/registries.conf` or globally.
- **Image platform mismatch.** `podman pull alpine` might fetch x86_64 by default if the registry doesn't auto-select. Force: `podman pull --platform linux/arm/v7 alpine`.
- **Bind-mount permissions.** Containers run as a different UID than the host's. Bind-mounting `/sys/class/gpio/` may work for reads but not writes if UID-mapping isn't right. The `--privileged` workaround is too coarse. The correct fix is `--user 0:0` plus careful capability grants.

35C.12 Going deeper

- **Podman documentation** at `docs.podman.io/`, comprehensive and well-organised.
- **`man podman-run`** for the full set of options.
- **`runc`**: the lower-level container runtime that Podman invokes. Smaller, fewer dependencies. Sometimes used directly on resource-constrained systems.
- **`containerd` and `nerdctl`**: alternative runtime + CLI, slightly closer to "what Kubernetes uses." Heavier than Podman but ecosystem-richer.

- **`Buildah`**: image-building companion to Podman without needing a daemon.
- **OCI Image Specification** at `github.com/opencontainers/image-spec`, the standard image format Podman, Docker, and friends all consume.
- **`balenaEngine`**: a stripped-down Docker variant specifically for embedded. Used by balena.io's commercial fleet-management platform.

End of Part V. You can now build a rootfs three ways (by hand, Buildroot, Ubuntu-base), harden it for production (RO + overlayfs), and ship containerised applications on top.

> Next chapter: **Chapter 36: Your first kernel module.** With the user-space picture complete, we turn back to the kernel to write the driver that gives `/sys/class/leds/led0/brightness` its behavior.

Appendix — Userspace tooling reference

> **What this is.** Throughout the book, chapters reach for `bluetoothctl`, `nmcli`, `gst-launch-1.0`, `alsamixer`, `wpa_supplicant`, `i2c-tools`, `mosquitto-clients`, `tcpdump`, `iperf3`, `gpsd`, `chrony`, `libnfc`, `mtd-utils`, `gdbserver`, … without saying where they come from. They come from **whatever rootfs you built or chose**. This appendix collects every tool referenced anywhere in the book and tells you how to install it on the three rootfs paths.
>
> **TL;DR.** During *learning*, use **Ubuntu-base (Ch 35A)** and `apt install ...`. When you *ship*, use **Buildroot (Ch 35)** and flip the corresponding Kconfig flag. **BusyBox-only (Ch 31)** does not have most of these.

A.1 Three rootfs paths, briefly recapped

Rootfs	Size	Get tools by	Best for
BusyBox hand-built (Ch 31)	~3 MB	hand-cross-compile each tool; mostly not worth it	"Hello, world" + learning the boot pipeline
Buildroot (Ch 35)	30–80 MB	`make menuconfig` → flip `BR2_PACKAGE_*` → rebuild	Production shipping
Ubuntu-base (Ch 35A)	~600 MB	`apt install <name>` on the target	Learning, prototyping, dev kit

The choice is not permanent. Many teams develop a feature on Ubuntu-base, validate, then port the same configuration to Buildroot for the shipping image.

A.2 Why BusyBox-only doesn't get you very far

BusyBox is one binary with ~240 *applets* (`ls`, `sh`, `mount`, `ip`, `mdev`, …). It covers the GNU coreutils + a slice of net-tools + sysvinit. What it does **not** contain is *anything specialised*: no `alsa-utils`, no `bluez`, no `gstreamer`, no `chrony`, no `gdbserver`. The applet list is fixed at compile time. Adding `bluetoothctl` to a BusyBox-only system means cross-compiling BlueZ separately and installing it next to BusyBox — which is what Buildroot does for you with one Kconfig flag.

Reach for BusyBox-only when the goal is to *understand* what an init does and what a rootfs minimum looks like. Once you want to *do something* with hardware (audio, BT, WiFi, modem, GPS, …) you need either Buildroot or Ubuntu-base.

A.3 Tool index — every name used in this book

For each tool: **what it is**, **chapters that use it**, **Ubuntu-base install**, **Buildroot Kconfig**, and a short note.

Networking + iproute2

Tool	Chapters	Ubuntu-base	Buildroot
`ip` (iproute2)	52, 55C, 91–94, 102, 108, 115	preinstalled	`BR2_PACKAGE_IPROUTE2=y`
`dhclient`, `dhcpcd`	24, 52, 91–93	preinstalled / `apt install isc-dhcp-client`	`BR2_PACKAGE_DHCPCD=y`
`ifupdown` (`ifup`, `ifdown`)	various	`apt install ifupdown`	`BR2_PACKAGE_IFUPDOWN=y`
`NetworkManager` (`nmcli`)	91–94, 102	`apt install network-manager`	`BR2_PACKAGE_NETWORK_MANAGER=y`
`tcpdump`	24, 52, 102, 108, 115	`apt install tcpdump`	`BR2_PACKAGE_TCPDUMP=y`
`iperf3`	52, 55E, 91–94, 115	`apt install iperf3`	`BR2_PACKAGE_IPERF3=y`
`ethtool`	52, 115	`apt install ethtool`	`BR2_PACKAGE_ETHTOOL=y`
`ss` (sockstat)	52, 102, 120	preinstalled	`BR2_PACKAGE_IPROUTE2=y` (bundled)
`nfs-kernel-server` (host)	24, 31, 35A	`apt install nfs-kernel-server`	n/a (host-only)
`tftpd-hpa` (host)	24, 26	`apt install tftpd-hpa`	n/a (host-only)

WiFi + wireless

Tool	Chapters	Ubuntu-base	Buildroot
`wpa_supplicant`	55E, 91–94	`apt install wpasupplicant`	`BR2_PACKAGE_WPA_SUPPLICANT=y`
`iw`	91–94	`apt install iw`	`BR2_PACKAGE_IW=y`
`hostapd` (AP mode)	91	`apt install hostapd`	`BR2_PACKAGE_HOSTAPD=y`
`wireless-regdb`	91	preinstalled	`BR2_PACKAGE_WIRELESS_REGDB=y`
`crda` (legacy regdb tool)	91	preinstalled (deprecated, merged into kernel)	typically not needed on modern kernels
WiFi firmware blobs	91–94	`apt install firmware-realtek firmware-brcm80211 …`	`BR2_PACKAGE_RTL_BT_FIRMWARE=y`, etc. (one per chip)

Bluetooth

Tool	Chapters	Ubuntu-base	Buildroot
BlueZ stack (`bluetoothd`)	95–97	`apt install bluez`	`BR2_PACKAGE_BLUEZ5_UTILS=y`
`bluetoothctl`	95–97	bundled with bluez	bundled
`btmon`, `btmgmt`, `hciattach`	95–97	`apt install bluez-tools`	`BR2_PACKAGE_BLUEZ5_UTILS_DEPRECATED=y` (legacy `hciconfig`/`hcitool`)
`bluetooth-meshd` + `mesh-cfgclient`	97	`apt install bluez-meshd` (Ubuntu 22.04+)	`BR2_PACKAGE_BLUEZ5_UTILS=y` + experimental mesh support

Tool	Chapters	Ubuntu-base	Buildroot
Pre-built BT firmware	95–97	`apt install firmware-misc-nonfree`	per-chip firmware package

Cellular (modems)

Tool	Chapters	Ubuntu-base	Buildroot
`pppd` + `chat`	24, 55F, 103	`apt install ppp`	`BR2_PACKAGE_PPP=y`
`ModemManager` (`mmcli`)	55F, 102	`apt install modemmanager`	`BR2_PACKAGE_MODEM_MANAGER=y`
`libqmi` (`qmicli`)	55F, 102	`apt install libqmi-utils`	`BR2_PACKAGE_LIBQMI=y`
`libmbim` (`mbimcli`)	55F, 102	`apt install libmbim-utils`	`BR2_PACKAGE_LIBMBIM=y`
`gpsd` + `gpspipe`, `cgps`, `gpsmon`	107	`apt install gpsd gpsd-clients`	`BR2_PACKAGE_GPSD=y`
`quectel-CM` (vendor)	102	not in apt; build from NXP/Quectel repo	not in mainline Buildroot

Audio (ALSA, PulseAudio, GStreamer)

Tool	Chapters	Ubuntu-base	Buildroot
`alsa-utils` (`alsamixer`, `aplay`, `arecord`)	53, 78, 89, 90	`apt install alsa-utils`	`BR2_PACKAGE_ALSA_UTILS=y`
ALSA libraries (`libasound2`)	53, 89, 90	preinstalled / `apt install libasound2-dev`	`BR2_PACKAGE_ALSA_LIB=y`

Tool	Chapters	Ubuntu-base	Buildroot
`pulseaudio`	optional (Ch 53 sidebar)	`apt install pulseaudio`	`BR2_PACKAGE_PULSEAUDIO=y`
`pipewire`	optional (Ch 53 sidebar)	`apt install pipewire`	`BR2_PACKAGE_PIPEWIRE=y`
`gstreamer1.0-tools` (`gst-launch-1.0`)	54B, 87, 88	`apt install gstreamer1.0-tools`	`BR2_PACKAGE_GSTREAMER1=y`
GStreamer plugins (base/good/bad/ugly)	54B, 87, 88	`apt install gstreamer1.0-plugins-{base,good,bad}`	`BR2_PACKAGE_GST1_PLUGINS_BASE/GOOD/BAD=y`
`v4l-utils` (`v4l2-ctl`, `qv4l2`)	54B, 87	`apt install v4l-utils`	`BR2_PACKAGE_V4L_UTILS=y`

GPIO / I²C / SPI / sensors / IIO

Tool	Chapters	Ubuntu-base	Buildroot
`libgpiod` (`gpioget`, `gpioset`, `gpiomon`, `gpioinfo`)	18B, 44, 78, 99, 111	`apt install gpiod`	`BR2_PACKAGE_LIBGPIOD=y`
`i2c-tools` (`i2cdetect`, `i2cdump`, `i2cset`, `i2cget`)	18, 46, 64–117 (most cookbook)	`apt install i2c-tools`	`BR2_PACKAGE_I2C_TOOLS=y`
`spidev-test`	47	`apt install spi-tools` (older) / build manually	`BR2_PACKAGE_SPI_TOOLS=y`
`evtest` (input subsystem)	45, 55G, 86, 106	`apt install evtest`	`BR2_PACKAGE_EVTEST=y`
`libinput-tools`	86	`apt install libinput-tools`	`BR2_PACKAGE_LIBINPUT=y`

Tool	Chapters	Ubuntu-base	Buildroot
`tslib` (resistive touch)	86	`apt install libts-bin`	`BR2_PACKAGE_TSLIB=y`
IIO sysfs (`/sys/bus/iio/devices/`)	all sensor chapters	kernel-provided; no userspace install	kernel `CONFIG_IIO=y`
`iio-utils` (`iio_readdev`, `iio_info`)	various	`apt install libiio-utils`	`BR2_PACKAGE_LIBIIO_TOOLS=y`

Storage / filesystem / flash

Tool	Chapters	Ubuntu-base	Buildroot
`mtd-utils` (`flash_erase`, `nandwrite`, `flashcp`, `mtdinfo`)	54A, 64	`apt install mtd-utils`	`BR2_PACKAGE_MTD=y`
`ubi-utils` (`ubinfo`, `ubiformat`, `ubinize`)	54A	bundled with `mtd-utils`	bundled
`parted`, `fdisk`, `sfdisk`	19, 31, 121	preinstalled	`BR2_PACKAGE_PARTED=y` / `BR2_PACKAGE_UTIL_LINUX_FDISK=y`
`e2fsprogs` (`mkfs.ext4`, `e2fsck`, `tune2fs`)	19, 31, 35B	preinstalled	`BR2_PACKAGE_E2FSPROGS=y`
`dosfstools` (`mkfs.vfat`)	121, 125	preinstalled	`BR2_PACKAGE_DOSFSTOOLS=y`
`cryptsetup` (LUKS)	124 (sidebar)	`apt install cryptsetup`	`BR2_PACKAGE_CRYPTSETUP=y`
`veritysetup` (dm-verity)	124	bundled with cryptsetup	bundled
`overlayfs` (kernel feature, not a tool)	35B	n/a	kernel `CONFIG_OVERLAY_FS=y`

Time + NTP + PPS

Tool	Chapters	Ubuntu-base	Buildroot
`chrony` (`chronyc`)	107, 117	`apt install chrony`	`BR2_PACKAGE_CHRONY=y`
`ntpd` (legacy alternative)	117	`apt install ntp`	`BR2_PACKAGE_NTP=y`
`pps-tools` (`ppstest`)	107	`apt install pps-tools`	`BR2_PACKAGE_PPS_TOOLS=y`
`hwclock`	18C, 117	preinstalled (util-linux)	`BR2_PACKAGE_UTIL_LINUX_HWCLOCK=y`

CAN / RS-485 / industrial

Tool	Chapters	Ubuntu-base	Buildroot
`can-utils` (`candump`, `cansend`, `cangen`, `canplayer`, `cansniffer`, `isotpsend`, `isotprecv`, `cangw`)	55C, 110	`apt install can-utils`	`BR2_PACKAGE_CAN_UTILS=y`
`libsocketcan`	55C, 110	`apt install libsocketcan2 libsocketcan-dev`	`BR2_PACKAGE_LIBSOCKETCAN=y`
`libmodbus` (`modbus_*`)	108	`apt install libmodbus5 libmodbus-dev`	`BR2_PACKAGE_LIBMODBUS=y`
`pymodbus` (Python)	108	`pip install pymodbus`	`BR2_PACKAGE_PYTHON3_PYMODBUS=y`

Identification / NFC / Fingerprint

Tool	Chapters	Ubuntu-base	Buildroot
`libnfc-bin` (`nfc-list`, `nfc-mfultralight`, …)	105	`apt install libnfc-bin libnfc-dev`	`BR2_PACKAGE_LIBNFC=y`
`neard` (NFC daemon, `neardctl`)	105	`apt install neard`	`BR2_PACKAGE_NEARD=y`
`mfoc` / `mfcuk` (Mifare Classic key recovery; security research)	105 sidebar	`apt install mfoc mfcuk`	not in mainline Buildroot
`fprintd` + `libfprint`	106	`apt install fprintd libfprint-2-2`	`BR2_PACKAGE_FPRINTD=y`
`pamu2fcfg` (2FA)	106	`apt install libpam-u2f`	`BR2_PACKAGE_LIBPAM_*`

Smart-home / mesh / IoT

Tool	Chapters	Ubuntu-base	Buildroot
`mosquitto` broker	100, 114, all MQTT examples	`apt install mosquitto`	`BR2_PACKAGE_MOSQUITTO=y`
`mosquitto-clients` (`mosquitto_pub`, `mosquitto_sub`)	as above	`apt install mosquitto-clients`	bundled when mosquitto enabled
`zigbee2mqtt`	100	Node 18+: `apt install nodejs npm`; then `npm install -g zigbee2mqtt`	non-trivial; usually run from a separate container
OpenThread Border Router (`otbr-agent`)	100	build from `openthread/ot-br-posix` source	non-trivial; typically self-built

Tool	Chapters	Ubuntu-base	Buildroot
`chip-tool` (Matter)	100	build from `project-chip/connectedhomeip`	non-trivial

Debug / tracing / profiling

Tool	Chapters	Ubuntu-base	Buildroot
`gdb-multiarch` (host)	118, 120, 125A	`apt install gdb-multiarch`	n/a (host-only)
`gdbserver` (target)	120, 125A	`apt install gdbserver`	`BR2_PACKAGE_GDB=y` + `BR2_PACKAGE_GDB_SERVER=y`
`strace`	120	`apt install strace`	`BR2_PACKAGE_STRACE=y`
`ltrace`	120	`apt install ltrace`	`BR2_PACKAGE_LTRACE=y`
`perf` (`linux-tools`)	120	`apt install linux-tools-generic` (kernel-version-matched)	`BR2_PACKAGE_LINUX_TOOLS_PERF=y`
`valgrind` (memcheck)	120 sidebar	`apt install valgrind`	`BR2_PACKAGE_VALGRIND=y`
`trace-cmd` + `kernelshark`	119	`apt install trace-cmd kernelshark`	`BR2_PACKAGE_TRACE_CMD=y` (kernelshark builds separately on host)
`bcc`, `bpftrace`	119	`apt install bpfcc-tools bpftrace`	`BR2_PACKAGE_BCC=y` / `BR2_PACKAGE_BPFTRACE=y` (kernel must have BPF + BTF)
`coredumpctl` (systemd)	120	bundled with systemd	requires systemd init
OpenOCD (host)	118	`apt install openocd`	n/a (host-only)

Build infrastructure (host)

Tool	Chapters	Ubuntu-base (host)
Cross toolchains (`arm-none-linux-gnueabihf-gcc`, `arm-none-eabi-gcc`)	3, 6, 122	Arm GNU Toolchain tarballs installed under `~/imx6ull/toolchains/`
`crosstool-NG` (build your own toolchain)	122	`apt install autoconf bison flex texinfo unzip`; clone `crosstool-ng`
`bison`, `flex`, `bc`, `libssl-dev` (kernel build)	25, 122	`apt install build-essential bison flex bc libssl-dev`
`device-tree-compiler` (`dtc`)	27, 27A	`apt install device-tree-compiler`
`u-boot-tools` (`mkimage`, `mkenvimage`)	11, 19, 23	`apt install u-boot-tools`
`qemu-user-static` (chroot foreign arch)	35A	`apt install qemu-user-static`
`binfmt-support`	35A	`apt install binfmt-support`
`imx_usb_loader`, `uuu` (i.MX SDP flashing)	3, 8, 24, 121A	`apt install imx-usb-loader`; build `uuu` from `nxp-imx/mfgtools`

OTA / images / production

Tool	Chapters	Ubuntu-base	Buildroot
RAUC (`rauc`)	125	`apt install rauc`	`BR2_PACKAGE_RAUC=y`
SWUpdate	125	build from source (sbabic/swupdate)	`BR2_PACKAGE_SWUPDATE=y`

Tool	Chapters	Ubuntu-base	Buildroot
Mender (`mender-client`, `mender-cli`)	125	`apt install mender-client`; `mender-cli` separate	`BR2_PACKAGE_MENDER=y` (community); commercial via Mender
`casync` (delta-update chunking)	125	`apt install casync`	`BR2_PACKAGE_CASYNC=y`
`wic` (Yocto image creator)	123A	bundled with Yocto / bitbake	n/a

Secure boot / TEE

Tool	Chapters	Ubuntu-base	Buildroot
NXP CST (Code Signing Tool)	124	download from NXP site; non-redistributable	not in Buildroot
`openssl` (key generation)	124	preinstalled	`BR2_PACKAGE_OPENSSL=y`
OP-TEE (`tee-supplicant`)	124	build from `OP-TEE/optee_os` and `OP-TEE/optee_client`	`BR2_PACKAGE_OPTEE_CLIENT=y`
`imx_habtool` (HAB CSF helper)	124	from NXP CST tarball	not in Buildroot

A.4 Quick-start “install everything” cheat sheets

Ubuntu-base one-liner

For a *learning* dev kit (any chapter in the book usable after this):

```
sudo apt update
sudo apt install -y \
    i2c-tools spi-tools gpiod libgpiod-dev \
    alsa-utils libasound2-dev \
    gstreamer1.0-tools gstreamer1.0-plugins-base gstreamer1.0-plugins-good gstreamer1.0-
plugins-bad \
```

```
    v4l-utils libv4l-dev \
    bluez bluez-tools \
    wpasupplicant iw hostapd wireless-regdb \
    network-manager modemmanager libqmi-utils libmbim-utils \
    ppp \
    gpsd gpsd-clients pps-tools chrony \
    can-utils libsocketcan-dev libmodbus5 libmodbus-dev \
    libnfc-bin libnfc-dev neard \
    fprintd libfprint-2-2 \
    mosquitto mosquitto-clients \
    mtd-utils dosfstools parted cryptsetup \
    iperf3 tcpdump ethtool \
    gdb-multiarch gdbserver strace ltrace valgrind \
    trace-cmd \
    rauc casync \
    device-tree-compiler u-boot-tools
```

That’s about 400 MB of additional disk on the target. Drop the lines you don’t need.

Buildroot Kconfig fragment

Drop into your `configs/myboard_defconfig` (or merge via `make menuconfig`):

```
BR2_PACKAGE_I2C_TOOLS=y
BR2_PACKAGE_LIBGPIOD=y
BR2_PACKAGE_LIBGPIOD_TOOLS=y
BR2_PACKAGE_ALSA_UTILS=y
BR2_PACKAGE_ALSA_LIB=y
BR2_PACKAGE_GSTREAMER1=y
BR2_PACKAGE_GST1_PLUGINS_BASE=y
BR2_PACKAGE_GST1_PLUGINS_GOOD=y
BR2_PACKAGE_GST1_PLUGINS_BAD=y
BR2_PACKAGE_V4L_UTILS=y
BR2_PACKAGE_BLUEZ5_UTILS=y
BR2_PACKAGE_BLUEZ5_UTILS_DEPRECATED=y
BR2_PACKAGE_WPA_SUPPLICANT=y
BR2_PACKAGE_IW=y
BR2_PACKAGE_HOSTAPD=y
BR2_PACKAGE_NETWORK_MANAGER=y
BR2_PACKAGE_MODEM_MANAGER=y
BR2_PACKAGE_LIBQMI=y
BR2_PACKAGE_LIBMBIM=y
BR2_PACKAGE_PPP=y
BR2_PACKAGE_GPSD=y
BR2_PACKAGE_PPS_TOOLS=y
BR2_PACKAGE_CHRONY=y
BR2_PACKAGE_CAN_UTILS=y
BR2_PACKAGE_LIBSOCKETCAN=y
BR2_PACKAGE_LIBMODBUS=y
BR2_PACKAGE_LIBNFC=y
BR2_PACKAGE_NEARD=y
BR2_PACKAGE_FPRINTD=y
BR2_PACKAGE_MOSQUITTO=y
BR2_PACKAGE_MTD=y
BR2_PACKAGE_DOSFSTOOLS=y
BR2_PACKAGE_E2FSPROGS=y
BR2_PACKAGE_PARTED=y
```

```
BR2_PACKAGE_CRYPTSETUP=y
BR2_PACKAGE_IPERF3=y
BR2_PACKAGE_TCPDUMP=y
BR2_PACKAGE_ETHTOOL=y
BR2_PACKAGE_GDB=y
BR2_PACKAGE_GDB_SERVER=y
BR2_PACKAGE_STRACE=y
BR2_PACKAGE_LTRACE=y
BR2_PACKAGE_VALGRIND=y
BR2_PACKAGE_TRACE_CMD=y
BR2_PACKAGE_RAUC=y
BR2_PACKAGE_CASYNC=y
BR2_PACKAGE_OPTEE_CLIENT=y
```

This will bloat the image by ~40 MB. That’s expected for a dev-image; production images trim to only what they ship.

A.5 Per-Part cheat sheets

If you only want the tools for the Part you’re reading right now:

Part VI — Driver development

```
# Ubuntu-base on target
apt install i2c-tools gpiod alsa-utils v4l-utils gstreamer1.0-tools \
            bluez wpasupplicant iw modemmanager libqmi-utils \
            mtd-utils can-utils evtest
```

Part VII — Device cookbook

Same as Part VI, plus:

```
apt install libnfc-bin neard fprintd \
            libmodbus-dev mosquitto-clients \
            gpsd gpsd-clients pps-tools chrony \
            iperf3 tcpdump
```

Part VIII — Debug, production, advanced

```
# On the host
apt install gdb-multiarch openocd build-essential bison flex bc libssl-dev \
            qemu-user-static binfmt-support u-boot-tools device-tree-compiler

# On the target (debug image only — strip for production)
apt install gdbserver strace ltrace valgrind trace-cmd perf \
            rauc casync
```

Part IX — Applied virtualization

```
# On the host
apt install qemu-system-arm qemu-system-aarch64 gdb-multiarch \
            device-tree-compiler u-boot-tools bridge-utils

# Hypervisor builds also need the normal kernel/U-Boot build tools above.
# Xen, Jailhouse, Zephyr, and STM32MP1-specific tooling are pinned inside
# their chapters because version drift matters.
```

A.6 Pitfalls

- **libgpiod v1 vs v2.** The CLI changed: `gpioget --chip=gpiochip0 18` (v2) vs `gpioget gpiochip0 18` (v1). Ubuntu 24.04+ ships v2; older Buildroot trees ship v1. Match documentation to your version.
- **GStreamer plugin sets.** "`base`" + "`good`" + "`bad`" cover ~95 % of pipelines; "`ugly`" adds patented-codec elements (h264, mp3). Add `gstreamer1.0-libav` for ffmpeg-backed elements. Ubuntu separates these into named packages; Buildroot uses `BR2_PACKAGE_GST1_PLUGINS_*=y`.
- **WiFi firmware blobs are non-redistributable.** Ubuntu ships them in `firmware-iwlwifi`, `firmware-realtek`, etc.; Buildroot has corresponding packages but you may need `BR2_LEGAL_INFO=y` and to accept the licenses.
- **`bluez` 5.x vs `bluetooth-meshd`.** Mesh support is a *separate daemon* (`bluetooth-meshd`); installing `bluez` alone does **not** give you mesh.
- **`mosquitto` broker + clients.** "Clients" without "broker" still works — you just connect to a remote broker. Drop `BR2_PACKAGE_MOSQUITTO=y` and keep `BR2_PACKAGE_MOSQUITTO_CLIENTS=y` if you don't host the broker.
- **`zigbee2mqtt` is Node.js, not a C package.** Run from npm, or containerize.
- **OpenThread / Matter require building from source.** No clean apt or Buildroot package; expect 30–90 min of build time and tracking upstream master.
- **NXP CST (Code Signing Tool) is not redistributable.** Download from the NXP site after registration; do not check it into a public repo.
- **`mfoc`/`mfcuk` are offensive-security tools.** Many jurisdictions restrict cloning RFID cards. Use for authorized assessment only.
- **systemd-only utilities.** `journalctl`, `coredumpctl`, `systemd-analyze` only work if your init is `systemd`. BusyBox-init / runit / sysvinit systems use `dmesg`, `gdb on /var/lib/cores/*`, and their own benchmarking instead.

- **`pip install` on production images.** Convenient on dev kit; ruins reproducibility on shipping firmware. Use Buildroot's `BR2_PACKAGE_PYTHON3_*` packages or vendor your wheels.

A.7 Going deeper

- **Buildroot manual, "Package recipes" chapter** — the canonical reference; lists every `BR2_PACKAGE_*` symbol.
- **Yocto Project Layer Index** (https://layers.openembedded.org/) — search any tool name to find the recipe.
- **Debian / Ubuntu packages.ubuntu.com** — search to verify the exact package name for any tool.
- **`apt-file`** (`apt install apt-file && apt-file update`) — search "which package ships this binary?": `apt-file search bin/gpioget`.
- **Per-chapter "Tooling" boxes** — most chapters that depend on a specific tool now carry a one-line install hint at the top of the chapter; this appendix is the canonical full reference.

Chapter 36: Your first kernel module

> **Privilege boundary:** $ means normal user. # or sudo means root and can change host or target state. After a privileged command, verify the expected device, service, or file appears before continuing. Roll back by undoing the config change or stopping the service you just enabled.

> **What:** the smallest possible Linux kernel module, a `.ko` file with an entry point, an exit point, and a license tag, and the build system (`Kbuild`) that turns C source into it. The example connects module loading, kernel logging, and unloading to their roles in the module lifecycle.
>
> **Why:** every driver you'll ever write, character, block, network, platform, I²C, SPI, sound, is a kernel module at its core. The wrapper around the interesting code is the same: module_init / module_exit / MODULE_LICENSE. Master the trivial case once and the only thing left to learn for each subsystem is its own API.
>
> **Focus: what gets linked into what, and when**. `hello.ko` is a relocatable ELF that the kernel loader patches into the kernel's address space at insmod time. Understanding that, that there is no fresh process, no separate memory, just dynamic linking into the running kernel, explains 80 % of "why is this allowed?" and "why is that not?" questions. **ELF:** Executable and Linkable Format, the standard Linux object and executable file format.

36.1 The driver mindset shift

If your last decade was MCU work, the first thing to internalise is this:

MCU firmware	Linux kernel module
`main()` runs from reset	No `main()`; you register callbacks
Your code owns the CPU	Your code is invoked when the kernel decides
Stack is whatever you allocate	Stack is ~16 KB and shared with whoever called you
Memory is whatever flash holds	Memory comes from `kmalloc()` / `vmalloc()`

MCU firmware	Linux kernel module
You print to a UART you set up	You call `printk` and it routes itself
Crash → board resets	Crash → kernel panics → user is unhappy

A kernel module is a library that the kernel dynamically links into itself. The kernel calls your `module_init` once when loaded, and your `module_exit` once when unloaded. In between, your code waits, then runs when something calls into it, a system call, an interrupt, a work queue, a kthread, or whichever subsystem registered it.

This is more than a coding-style change. The failure modes change too. A wild pointer in firmware corrupts your `.bss`. A wild pointer in a kernel module corrupts the kernel itself. That can mean kernel threads, other drivers, or the scheduler, whatever happens to be running. A bug that would have been a crashed loop is now an unbootable system.

36.2 The smallest module that compiles

`hello.c`:

```
#include <linux/init.h>
#include <linux/module.h>
#include <linux/kernel.h>

static int __init hello_init(void)
{
    pr_info("hello: loaded\n");
    return 0;
}

static void __exit hello_exit(void)
{
    pr_info("hello: unloaded\n");
}

module_init(hello_init);
module_exit(hello_exit);

MODULE_LICENSE("GPL");
MODULE_AUTHOR("you@example.com");
MODULE_DESCRIPTION("Smallest possible kernel module");
MODULE_VERSION("0.1");
```

About twenty lines. Walk through each.

Headers.

- `<linux/init.h>`: defines `__init` and `__exit` macros, plus `module_init` and `module_exit`.

- `<linux/module.h>`: defines the `MODULE_*` metadata macros and ties into the module-loading infrastructure.
- `<linux/kernel.h>`: pulls in `printk`, `pr_info`, and friends.

Note **none** of these are `<stdio.h>` or `<stdlib.h>`. **The kernel has no libc.** `printf`, `malloc`, `strcmp`, all unavailable. The kernel provides its own versions, sometimes with the same name (`strcmp`), sometimes renamed (`kmalloc` instead of `malloc`, `printk` instead of `printf`).

`__init` annotation. `hello_init` is the function that runs when the module loads. Marking it `__init` tells the linker to put it in a special section (`.init.text`) that the kernel can **free** after the module has loaded. For a built-in module (statically linked into the kernel image), this means ~10 KB of init code is reclaimed once the system boots. For a `.ko`, the section is freed after `module_init` returns. **Implication:** code in `__init` functions can call other `__init` functions but not be called from runtime code. The linker warns if you violate this.

Return value. `hello_init` returns `int`. Returning 0 means "module loaded successfully." A negative `errno` (e.g., `-ENODEV`, `-EINVAL`) means "load failed", the kernel unloads the module and `insmod` reports the error.

`__exit` annotation. Similar to `__init`: `.exit.text` is freed in the built-in case (since a built-in module is never unloaded). For a `.ko`, `module_exit` runs when `rmmod` is called.

`module_init(hello_init).` This isn't a function call, it's a macro that expands to a special section entry telling the loader: "the entry point is `hello_init`." Without it, the loader can't find your code.

`MODULE_LICENSE("GPL").` Without this, two things happen: the kernel taints itself (sets a "this kernel has loaded non-GPL code" flag visible in `dmesg` and `/proc/sys/kernel/tainted`), and **many GPL-only exported symbols become unavailable to your module.** If you try to call `gpiod_get` (which is `EXPORT_SYMBOL_GPL`) from a non-GPL module, you get `Unknown symbol gpiod_get`. There is no way around this short of using a GPL license tag. Other accepted strings: `"GPL v2"`, `"GPL and additional rights"`, `"Dual BSD/GPL"`, `"Dual MIT/GPL"`, `"Dual MPL/GPL"`, `"Proprietary"`.

The rest of the `MODULE_*` macros are just metadata. `modinfo hello.ko` displays them. They're not load-time gates (except `MODULE_LICENSE`).

36.3 The `Kbuild` Makefile

> **Driver choice:** Use the in-tree, maintained driver first. Use out-of-tree, spidev, or custom-driver paths only after you accept the kernel-version maintenance cost and document who owns updates.

You don't compile a kernel module with `arm-none-linux-gnueabihf-gcc hello.c -o hello.ko`. Kbuild does a lot of work for you. It generates per-module ELF sections, applies the kernel's own `CFLAGS` (including `-fno-stack-protector` and many others), and dynamically discovers the right `vmlinux` symbols. You build out-of-tree modules by *invoking the kernel's own Makefile* and pointing at your source.

`Makefile`:

```
obj-m += hello.o

# Path to the kernel source (must match the kernel we'll run on)
KDIR ?= /home/$(USER)/linux-imx6ull/build

# Architecture and cross-compiler for the target
ARCH         ?= arm
CROSS_COMPILE ?= arm-none-linux-gnueabihf-

all:
    $(MAKE) -C $(KDIR) ARCH=$(ARCH) CROSS_COMPILE=$(CROSS_COMPILE) M=$(PWD) modules

clean:
    $(MAKE) -C $(KDIR) ARCH=$(ARCH) CROSS_COMPILE=$(CROSS_COMPILE) M=$(PWD) clean
```

Run:

```
$ make
make -C /home/you/linux-imx6ull/build M=/home/you/hello modules
  CC [M]  /home/you/hello/hello.o
  MODPOST /home/you/hello/Module.symvers
  CC [M]  /home/you/hello/hello.mod.o
  LD [M]  /home/you/hello/hello.ko
```

Five lines of `make` output. What happened:

1. **`CC [M] hello.o`**: Your `hello.c` compiled to `hello.o`. Standard so far.
2. **`MODPOST`**: Module post-processing. This scans `hello.o` for **undefined symbols** (functions you call but didn't define, `printk`, in our case) and looks them up in the kernel's exported symbol table (`Module.symvers` from the kernel build). For each symbol it finds, it records the symbol's hash. If a symbol is **not** exported, modpost emits a warning. If a symbol is `EXPORT_SYMBOL_GPL` and you're not GPL, modpost emits a warning.
3. **`CC [M] hello.mod.o`**: Modpost generates a small `hello.mod.c` containing version magic, module metadata, and symbol-version records. This compiles to `hello.mod.o`.
4. **`LD [M] hello.ko`**: Final link. `hello.o` + `hello.mod.o` → `hello.ko`. The `.ko` is a **relocatable ELF**, not a shared object. The kernel does the relocations itself at load time.

Inspect the output:

```
$ file hello.ko
hello.ko: ELF 32-bit LSB relocatable, ARM, EABI5 version 1 (SYSV), ...

$ arm-none-linux-gnueabihf-readelf -h hello.ko | head -5
  Class:                             ELF32
  Data:                              2's complement, little endian
  Type:                              REL (Relocatable file)
  Machine:                           ARM

$ modinfo hello.ko
filename:       /home/you/hello/hello.ko
version:        0.1
description:    Smallest possible kernel module
author:         you@example.com
license:        GPL
srcversion:     ABCD1234EF567890
depends:
vermagic:       6.6.0 SMP mod_unload modversions ARMv7
```

The **`vermagic`** is critical. It encodes the exact kernel version (`6.6.0`) and the config options that affect binary compatibility (`SMP`, `mod_unload`, etc.). When you `insmod hello.ko`, the kernel compares `hello.ko`'s vermagic to its own. If they differ, the load is refused. You can't take a module built against 6.6.0 and load it on 6.6.1, even if the API is identical. This is a feature, not a bug, it prevents subtle ABI breakage.

> **ABI:** Application Binary Interface: the calling convention, register use, binary format, and library contract that let separately built code run together.

36.4 Loading and unloading

Copy `hello.ko` to the target. Then on the target:

```
[root@pa-mini:~]# insmod hello.ko
[root@pa-mini:~]# dmesg | tail -1
[   42.123456] hello: loaded

[root@pa-mini:~]# lsmod
Module                  Size  Used by
hello                  16384  0

[root@pa-mini:~]# rmmod hello
[root@pa-mini:~]# dmesg | tail -1
[   50.234567] hello: unloaded
```

That's the whole loop. The module is now part of the kernel's address space. Its code is reachable via the symbol table.

`/sys/module/hello/` exposes runtime information:

```
[root@pa-mini:~]# ls /sys/module/hello/
coresize      holders       initsize    refcnt    srcversion    uevent
initsize      initstate     notes/      sections/ taint         version
```

- `refcnt`: how many other things use this module. As long as it's > 0, `rmmod` refuses.
- `srcversion`: hash of the source. Useful for detecting "did I actually copy the new build?"
- `sections/`: load address of each section. Useful when debugging crashes. You need the section addresses to symbolicate addresses in the `dmesg` backtrace.
- `taint`: does this module taint the kernel?

`/proc/modules` has the same info but flatter, suitable for scripting.

36.5 Module parameters

Hardcoding settings is fine for "hello world" but real drivers need to be tunable. Linux gives you `module_param`:

```
#include <linux/moduleparam.h>

static int howmany = 1;
static char *whom = "world";

module_param(howmany, int, 0644);
MODULE_PARM_DESC(howmany, "Number of greetings");

module_param(whom, charp, 0644);
MODULE_PARM_DESC(whom, "Who to greet");

static int __init hello_init(void)
{
    int i;
    for (i = 0; i < howmany; i++)
        pr_info("hello, %s\n", whom);
    return 0;
}
```

The third argument to `module_param` is the **permission** for the sysfs file at `/sys/module/hello/parameters/howmany`. `0644` means readable by all, writable by root. `0` means not exposed in sysfs at all.

> **sysfs:** a kernel-generated filesystem under /sys that exposes devices, drivers, and attributes.

Use at load time:

```
[root@pa-mini:~]# insmod hello.ko howmany=3 whom=earth
[root@pa-mini:~]# dmesg | tail -3
[  ... ] hello, earth
[  ... ] hello, earth
[  ... ] hello, earth
```

Or change at runtime via sysfs:

```
[root@pa-mini:~]# cat /sys/module/hello/parameters/whom
earth
[root@pa-mini:~]# echo mars > /sys/module/hello/parameters/whom
[root@pa-mini:~]# cat /sys/module/hello/parameters/whom
mars
```

Note: writing to the sysfs file **does not re-run `hello_init`.** It just updates the variable. Your driver's runtime code needs to read the variable each time to see the new value.

Supported types: `bool`, `byte`, `short`, `ushort`, `int`, `uint`, `long`, `ulong`, `charp` (a `char *`), and arrays via `module_param_array`.

36.6 `printk` and its message levels

`printk` is `printf`'s kernel cousin. The signature is the same. The difference is that the first character of the format string (a special `KERN_*` byte) sets the **message level**. There are eight levels:

Macro	Value	Used for
`KERN_EMERG`	0	System is unusable, panic imminent
`KERN_ALERT`	1	Action must be taken immediately
`KERN_CRIT`	2	Critical conditions (e.g., hard hardware errors)
`KERN_ERR`	3	Error conditions (driver init failed, etc.)
`KERN_WARNING`	4	Warnings (the default, unmarked `printk` uses this)

Macro	Value	Used for
KERN_NOTICE	5	Normal but significant
KERN_INFO	6	Informational (load/unload, link up, etc.)
KERN_DEBUG	7	Debug noise

Modern code uses the convenience wrappers, which are clearer:

```
pr_emerg("...");      // KERN_EMERG
pr_alert("...");      // KERN_ALERT
pr_crit("...");
pr_err("...");
pr_warn("...");
pr_notice("...");
pr_info("...");
pr_debug("...");
```

`pr_debug` is special: it expands to a `printk` only if `DEBUG` is defined when the file is compiled, **or** if dynamic debug (`CONFIG_DYNAMIC_DEBUG`) is enabled and the user has turned this site on at runtime. We use it heavily. Dynamic debug is one of the most underused gems of kernel development.

What you see in `dmesg` depends on the **console log level**. Read it:

```
[root@pa-mini:~]# cat /proc/sys/kernel/printk
4    4    1    7
```

Four numbers: current console log level, default for unmarked messages, minimum allowed, boot default. The console shows messages with priority **lower than or equal to** the first number, so `4` means levels `0–3` print to console, levels `4–7` only go to the ring buffer. Raise it:

```
[root@pa-mini:~]# echo 8 > /proc/sys/kernel/printk
```

Now everything prints to console. Useful for debugging. Noisy in production.

36.7 Key Notes

- A loadable module joins the kernel’s address space. Its callbacks have kernel privileges and no process-level fault isolation.
- Kbuild supplies the running kernel’s build contract, including configuration and exported symbols. A relocatable `.ko` is not a user-space executable.
- Initialization acquires resources and publishes callbacks. Failed initialization must unwind acquired resources before returning an error.
- `__init` code is temporary. Runtime callbacks cannot depend on code or data reclaimed after initialization.
- Module parameters expose configuration, not a second initialization path. Runtime changes need a defined effect on active driver state.
- Unloading requires both reference management and removal of asynchronous users. An Oops is a kernel fault, not a safe recovery mechanism or evidence of board reliability.

36.8 Pitfalls

- **`vermagic` mismatch.** “Module hello: version magic ‘…’ should be ‘…’” The two strings are right there, diff them. Usually you built against the wrong kernel tree, or the running kernel was updated and you haven’t rebuilt.
- **`Unknown symbol foo (err -2).`** A function you reference isn’t exported by the running kernel. `nm hello.ko | grep " U "` shows your undefined references. `cat /proc/kallsyms | grep foo` shows what the kernel has. The fix is usually: GPL-license the module, or pick a different API.
- **`__init` data referenced at runtime.** Compile warning: “section mismatch.” Your `__init` function called by something that lives in `.text` (runtime). Either drop the `__init` annotation or restructure so the call only happens during init.
- **Forgetting `MODULE_LICENSE`.** Build succeeds. Modpost warns “missing MODULE_LICENSE”. Load fails with “taints kernel” and many APIs unavailable. Always include it.
- **Out-of-tree modules and kernel upgrades.** Your `hello.ko` is tied to one exact kernel build. Plan for either rebuilding the module each time you rebuild the kernel, or adopt **DKMS** (Dynamic Kernel Module Support) which rebuilds automatically on kernel upgrade. For an embedded product with a fixed kernel, just build once and ship the `.ko` alongside `zImage`.
- **`rmmod` says “Resource busy”.** Something is using your module. The `refcnt` in `/sys/module/<name>/refcnt` tells you how many. The `holders/` directory inside lists which other modules reference yours. You can’t unload until they all go away.

- **`insmod` says nothing, but `dmesg` does.** `insmod` only reports errors to stderr. The success or failure detail (and any `printk` calls from your init function) go to the kernel log. Always check `dmesg` after every load.

36.9 Going deeper

- **`Documentation/kbuild/modules.rst`**: the canonical reference for the out-of-tree module build system.
- **`Documentation/admin-guide/dynamic-debug-howto.rst`**: turn `pr_debug` calls on and off at runtime, by module, file, line, or function. Indispensable.
- **`Linux Device Drivers, 3rd edition`** (LDD3, Corbet/Rubini/Kroah-Hartman), old but still the best long-form introduction to kernel modules. Chapters 2 and 3 cover modules and char devices.
- **`Documentation/process/submitting-patches.rst`**: when your driver is good enough to upstream.
- **`scripts/checkpatch.pl`**: run it on every change. Catches whitespace, style, and many common bugs.

> Next chapter: **Chapter 37: A character driver, by hand.** The module lifecycle provides the setting for `file_operations`, the interface through which user-space can `open`, `read`, `write`, and `close` `/dev/hello`.

Chapter 37: A character driver, by hand

> **What:** a character device driver, the kind that backs `/dev/ttyS0`, `/dev/i2c-1`, `/dev/hidraw0`, and most other "stream of bytes you read and write" device files in `/dev/`. A small buffer-backed example shows device-number allocation, `cdev` registration, `open`/`read`/`write`/`release`, and the user/kernel data boundary.
>
> **Why:** character drivers are how the vast majority of embedded peripheral drivers expose themselves to user-space. UARTs, GPIO chips, I²C/SPI controllers, sensors, fingerprint readers, sound cards' control interfaces, almost all are character devices under the hood. The pattern is identical every time. What changes is the body of `open` / `read` / `write`. Master the pattern in this chapter. Everything in Part VI is variations on it. **MCU bridge:** Think of Linux GPIO like the same pin set/reset block you used on STM32, but accessed through a kernel subsystem that owns numbering, direction, interrupts, and user-space exposure. **GPIO:** General-Purpose Input/Output, a pin controlled as a digital input, output, or interrupt source.
>
> **Focus: the user/kernel boundary**. The single most common bug class in driver code is dereferencing a user-space pointer directly. `copy_to_user` and `copy_from_user` are not just safer, they are *correct*. A direct dereference will fault, silently corrupt, or be a security hole. By the end of this chapter the `__user` annotation should feel like a load-bearing part of every function signature.

37.1 The picture

When a process does `fd = open("/dev/hello", O_RDWR)` and then `write(fd, "hi", 2)`, this chain of events happens:

```
user-space                kernel-space
──────────                ────────────
write(fd, "hi", 2)
     │
     │  glibc wraps it,
     │  invokes SVC                sys_write(fd, ...)
     │                                   │
     │                             VFS:  lookup file*
     │                                   │
     │                             dispatch to f_op->write
     │                                   │
     │                             your_driver_write(filp, buf, 2, &pos)
     │                                   │
     │                             copy_from_user(kbuf, buf, 2)
     │                                   │
     │                             do something useful
     │                                   │
```

```
        |                              return 2 (bytes written)
   2  <-------------------------------  return up the stack
```

Your driver provides a **`file_operations`** struct. The kernel's VFS layer looks up the right `file_operations` for a given device number, then calls your function pointers. Everything else is plumbing.

What the user thinks is "writing to a file" is whatever your `write` callback decides to do, send UART bytes, toggle GPIOs, fill a buffer for next read. The "file" is a façade. You decide what's behind it.

37.2 Device numbers

A device file in `/dev/` has a **major** and **minor** number:

```
[root@pa-mini:~]# ls -l /dev/null /dev/ttymxc0 /dev/i2c-0
crw-rw-rw- 1 root root  1, 3 May 24  2026 /dev/null
crw--w---- 1 root tty 207, 0 May 24  2026 /dev/ttymxc0
crw-rw---- 1 root i2c  89, 0 May 24  2026 /dev/i2c-0
```

The two numbers after the size (`1, 3`, `207, 0`, `89, 0`) are *major, minor*. The kernel uses the major number to dispatch to the right driver. The driver uses the minor number to distinguish among devices that share the driver (e.g., `i2c-0`, `i2c-1`, `i2c-2`).

On Linux, the combined number is a 32-bit `dev_t`: 12 bits major, 20 bits minor. The split is opaque to most code. Use the macros:

```
dev_t devid = MKDEV(major, minor);
unsigned int maj = MAJOR(devid);
unsigned int min = MINOR(devid);
```

Picking a major number, don't

The old way was to pick an unused major from a documented list (`Documentation/admin-guide/devices.txt`, the list of all officially registered major numbers). The modern way is to ask the kernel for one:

```
dev_t devid;
int err = alloc_chrdev_region(&devid, 0, 1, "hello");
//                             ▲      ▲  ▲   ▲
//                             |      |  |   name shown in /proc/devices
//                             |      |  count of consecutive minors
//                             |      base minor (almost always 0)
//                             out: assigned dev_t
```

The kernel finds an unused major and returns the `dev_t` to you. Always prefer this for new drivers. Hard-coding majors is a 1990s pattern. Don't do it.

After `alloc_chrdev_region`, you'll see your device in `/proc/devices`:

```
[root@pa-mini:~]# cat /proc/devices
Character devices:
...
240 hello
...
```

When you unload the driver, balance this with `unregister_chrdev_region(devid, 1)`.

37.3 The cdev structure

A `cdev` is the kernel's representation of a character device. You build one in your driver and register it:

```
#include <linux/cdev.h>

struct cdev mycdev;

cdev_init(&mycdev, &my_fops);   /* attach file_operations */
mycdev.owner = THIS_MODULE;
err = cdev_add(&mycdev, devid, 1);   /* register; now reachable */
```

After `cdev_add`, opening `/dev/hello` (assuming the device file exists with the right major/minor) routes through your `my_fops`. `cdev_init` and `cdev_add` are really one logical step. The kernel splits them so it can tell apart initialization from registration. Treat them as two lines next to each other.

To remove: `cdev_del(&mycdev)`.

37.4 The full structure of a chardev driver

Here is the layout of the buffer-backed example, `hello_chrdev.c`:

```
#include <linux/init.h>
#include <linux/module.h>
#include <linux/kernel.h>
#include <linux/fs.h>
#include <linux/cdev.h>
#include <linux/slab.h>
#include <linux/uaccess.h>

#define HELLO_BUFSIZE  256

struct hello_dev {
    struct cdev cdev;
    dev_t devid;
    char *buffer;
    size_t buf_len;
    struct mutex lock;
};
```

```
static struct hello_dev *hd;

/* ──────────────────────────── file_operations ──────────────────────────── */

static int hello_open(struct inode *inode, struct file *filp)
{
    struct hello_dev *dev = container_of(inode->i_cdev,
                                         struct hello_dev, cdev);
    filp->private_data = dev;
    pr_info("hello: open\n");
    return 0;
}

static int hello_release(struct inode *inode, struct file *filp)
{
    pr_info("hello: release\n");
    return 0;
}

static ssize_t hello_read(struct file *filp, char __user *ubuf,
                          size_t count, loff_t *ppos)
{
    struct hello_dev *dev = filp->private_data;
    ssize_t ret;

    if (mutex_lock_interruptible(&dev->lock))
        return -ERESTARTSYS;

    if (*ppos >= dev->buf_len) {
        ret = 0;   /* EOF */
        goto out;
    }
    if (*ppos + count > dev->buf_len)
        count = dev->buf_len - *ppos;

    if (copy_to_user(ubuf, dev->buffer + *ppos, count)) {
        ret = -EFAULT;
        goto out;
    }
    *ppos += count;
    ret = count;

out:
    mutex_unlock(&dev->lock);
    return ret;
}

static ssize_t hello_write(struct file *filp, const char __user *ubuf,
                           size_t count, loff_t *ppos)
{
    struct hello_dev *dev = filp->private_data;
    ssize_t ret;

    if (mutex_lock_interruptible(&dev->lock))
        return -ERESTARTSYS;

    if (count > HELLO_BUFSIZE)
        count = HELLO_BUFSIZE;
```

```
    if (copy_from_user(dev->buffer, ubuf, count)) {
        ret = -EFAULT;
        goto out;
    }
    dev->buf_len = count;
    *ppos = count;
    ret = count;

out:
    mutex_unlock(&dev->lock);
    return ret;
}

static const struct file_operations hello_fops = {
    .owner   = THIS_MODULE,
    .open    = hello_open,
    .release = hello_release,
    .read    = hello_read,
    .write   = hello_write,
};

/* ──────────────────────────── init / exit ──────────────────────────── */

static int __init hello_init(void)
{
    int err;

    hd = kzalloc(sizeof(*hd), GFP_KERNEL);
    if (!hd)
        return -ENOMEM;

    hd->buffer = kzalloc(HELLO_BUFSIZE, GFP_KERNEL);
    if (!hd->buffer) {
        err = -ENOMEM;
        goto free_dev;
    }
    mutex_init(&hd->lock);

    err = alloc_chrdev_region(&hd->devid, 0, 1, "hello");
    if (err)
        goto free_buf;

    cdev_init(&hd->cdev, &hello_fops);
    hd->cdev.owner = THIS_MODULE;

    err = cdev_add(&hd->cdev, hd->devid, 1);
    if (err)
        goto unreg_region;

    pr_info("hello: major=%d minor=%d  (mknod /dev/hello c %d 0)\n",
            MAJOR(hd->devid), MINOR(hd->devid), MAJOR(hd->devid));
    return 0;

unreg_region:
    unregister_chrdev_region(hd->devid, 1);
free_buf:
    kfree(hd->buffer);
free_dev:
```

```
    kfree(hd);
    return err;
}

static void __exit hello_exit(void)
{
    cdev_del(&hd->cdev);
    unregister_chrdev_region(hd->devid, 1);
    kfree(hd->buffer);
    kfree(hd);
    pr_info("hello: unloaded\n");
}

module_init(hello_init);
module_exit(hello_exit);

MODULE_LICENSE("GPL");
MODULE_DESCRIPTION("Hello character device");
```

That’s 100-some lines. Let’s pull out the four ideas that matter.

Idea 1: `container_of` and `private_data`

```
struct hello_dev *dev = container_of(inode->i_cdev, struct hello_dev, cdev);
filp->private_data = dev;
```

The inode passed to `open` knows it has a `cdev` inside it (`inode->i_cdev`). But what we *want* is the enclosing `hello_dev` struct that has all our state. `container_of` is the kernel’s “given a pointer to a struct member, recover the pointer to the containing struct” macro. It’s a compile-time trick, no runtime cost.

We stash the `hello_dev *` in `filp->private_data` so subsequent `read`/`write`/`release` calls can find it without redoing the lookup. **`filp->private_data` is the standard idiom** for per-open state.

Idea 2: `__user` and copy_to/from_user

```
static ssize_t hello_read(..., char __user *ubuf, ...)
{
    if (copy_to_user(ubuf, dev->buffer + *ppos, count))
        return -EFAULT;
}
```

The `__user` annotation on `ubuf` is a marker for `sparse` (a static analyzer) saying “this pointer is in user-space’s address space. Do not dereference it directly.” On i.MX6ULL there is no MMU domain protection, so a direct dereference might *appear* to work. But it only works when the user buffer is paged in and reachable from kernel mode, not always the case. On systems with **PAN** (Privileged Access Never, an ARMv8 feature. Not on i.MX6ULL but on many newer SoCs) a direct dereference faults immediately.

> **MCU bridge:** Think of the MMU as a hardware address translator in front of every load/store. Cortex-M usually runs physical addresses directly. Linux relies on virtual addresses and page permissions. **MMU:** Memory Management Unit, hardware that translates virtual addresses to physical addresses and enforces permissions.

`copy_to_user` (and its sibling `copy_from_user`) do three things:

1. **Validate the address** is within user-space (`access_ok`).
2. **Handle page faults gracefully**: If the user-space page is paged out, the function brings it in.
3. **Return the number of bytes NOT copied**: Zero = success. Nonzero = partial copy. **Most drivers convert any nonzero result to `-EFAULT`.**

Don't bypass `copy_to/from_user`. If you find yourself thinking "I just need to peek at one byte," use `get_user(byte, p)` (single byte) or `put_user(byte, p)` (single write). Same safety guarantees, smaller code.

Idea 3: Locking

```
if (mutex_lock_interruptible(&dev->lock))
    return -ERESTARTSYS;
```

Multiple processes can have your device open at the same time. Two threads doing `write()` concurrently can race on `dev->buffer`. A `mutex` serializes them.

We use `mutex_lock_interruptible` rather than `mutex_lock`. The difference: if a signal is pending while we wait for the lock, `_interruptible` returns `-ERESTARTSYS` and the kernel rolls back the syscall so it can restart after the signal handler. `mutex_lock` (uninterruptible) can leave a process unkillable if the lock is held by a buggy other path.

`-ERESTARTSYS` is the conventional return code for "signal pending. Please restart me." The VFS handles it correctly.

Idea 4: Goto-based unwind

```
err = alloc_chrdev_region(...);
if (err)
    goto free_buf;
err = cdev_add(...);
if (err)
    goto unreg_region;
...
unreg_region:
    unregister_chrdev_region(...);
free_buf:
    kfree(hd->buffer);
free_dev:
```

```
    kfree(hd);
    return err;
```

This `goto` cascade is **idiomatic kernel C**. Each label cleans up *exactly* what's been allocated up to that point. It is the kernel's idiomatic error-path style. After a dozen drivers it becomes natural. Read it carefully: each error-path target only does the cleanup for resources that were successfully acquired *before* this point.

37.5 Building, loading, testing

```
$ make
$ scp hello_chrdev.ko target:~
```

On the target:

```
[root@pa-mini:~]# insmod hello_chrdev.ko
[root@pa-mini:~]# dmesg | tail -1
hello: major=240 minor=0  (mknod /dev/hello c 240 0)

[root@pa-mini:~]# mknod /dev/hello c 240 0
[root@pa-mini:~]# echo "ping" > /dev/hello
[root@pa-mini:~]# cat /dev/hello
ping
```

It works. We wrote "ping" into the kernel buffer. `cat` read it back.

A few things to notice:

- The major number (240) is **whatever the kernel picked**. It's not stable across reboots. Next chapter, we'll automate device-file creation with udev/mdev so you don't have to `mknod` by hand.

> **udev:** the user-space device manager that reacts to kernel device events and creates policy-driven /dev nodes.

- `echo "ping"` writes 5 bytes (4 + newline). `cat` reads all of them. Our buffer correctly tracks `buf_len`.
- Each new `cat` invocation gets a fresh open, so `*ppos` resets to 0. Within one `cat`, the first `read` returns 5 bytes and the second returns 0 (EOF).

37.6 Testing edge cases

> **Privilege boundary:** $ means normal user. # or sudo means root and can change host or target state. After a privileged command, verify the expected device, service, or file appears before continuing. Roll back by undoing the config change or stopping the service you just enabled.

Test	Expected	Why
`dd if=/dev/zero of=/dev/hello bs=1M count=1`	Truncates to 256 bytes	Our `count > HELLO_BUFSIZE` clamp
`dd if=/dev/hello of=/dev/null bs=1` (after write)	Reads exactly `buf_len` bytes	Our `*ppos + count > buf_len` clamp
Two `cat /dev/hello > x &` in parallel	Both succeed, neither corrupts	Mutex serializes
Kill a stuck read with Ctrl-C	Returns immediately	`mutex_lock_interruptible`
`rmmod hello_chrdev` while a process has it open	"Resource busy"	`THIS_MODULE` owner causes refcount > 0

The last one is worth elaborating. Set `cdev.owner = THIS_MODULE`. The kernel auto-increments your module's reference count for every `open` on a device file under your `cdev`. As long as one process has `/dev/hello` open, `rmmod` refuses. This is the safety mechanism that prevents "user has fd open → driver unloaded → user's next read crashes" bugs.

37.7 `mknod` and why it's temporary

Notice we manually `mknod /dev/hello c 240 0` to create the device file. That's tedious and fragile:

- The major number is dynamic, pick by `alloc_chrdev_region`, but `mknod` requires you to know it.
- Reboots may renumber.
- On a tmpfs `/dev` (almost always true now. See Ch 32), the manually-`mknod`'d node disappears at reboot, you'd have to recreate it each boot.

Chapter 38 fixes this entirely: with `class_create` + `device_create`, the kernel **broadcasts a hot-plug event** when your driver loads, and udev (or mdev) creates the right file in `/dev/` automatically. Same when you unload: the file disappears.

For now, `mknod` is fine. Just know it's a stopgap.

37.8 Key Notes

- The device number selects a `cdev`, and the VFS dispatches operations through its `file_operations`. The pathname alone does not define device behavior.
- `private_data` connects an open file to driver state. Shared device state and per-open state have different lifetimes and synchronization needs.
- User pointers remain untrusted across the syscall boundary. User-copy helpers report incomplete transfers without making those pointers ordinary kernel memory.
- Byte counts, offsets, short transfers, EOF, and negative errors form the user-space contract. A bounded buffer must not report bytes it did not accept.
- A mutex can serialize buffer access in syscall context. It does not make a separate write followed by a read one indivisible transaction.
- Module ownership keeps callback code resident while it is referenced. Device-state lifetime still needs a separate plan for unbinding or removal.

37.9 Pitfalls

- **Forgetting `THIS_MODULE` in `cdev.owner` or `file_operations.owner`.** The kernel won't increment your module's refcount on open. `rmmod` while a process has the device open → kernel crashes when it tries to call into freed code. Always set both.
- **Calling user-space functions inside the kernel.** Kernel code does not have access to glibc. No `printf`, no `malloc`, no `memcpy_s`. Use `printk`/`pr_*`, `kmalloc`/`kfree`, `memcpy` (which exists in the kernel, slightly different optimisation profile).
- **Stack overflow.** Kernel stacks are **8 KB on ARM32 i.MX6ULL** (16 KB on x86_64 / arm64). Don't put large arrays on the stack. If you need a 4 KB scratch buffer, use `kmalloc(4096, GFP_KERNEL)` and free it at the end of the function.
- **Allocating with the wrong flag.** `kmalloc(..., GFP_KERNEL)` may sleep, fine in syscall context, not fine in interrupt context. In an IRQ handler, use `GFP_ATOMIC`. We'll cover this in Ch 43.

> **MCU bridge:** Think of an IRQ like an EXTI/NVIC interrupt path, except Linux splits the hard interrupt from deferred work and must share lines across drivers. **IRQ:** interrupt request, the signal path that tells the CPU or interrupt controller that hardware needs service.

- **Returning the wrong type.** `read` and `write` return `ssize_t`. Don't return `int` (compile warning), don't return `size_t` (may hide negative values), and don't return success when you mean count.
- **Holding a mutex across `copy_to_user`.** `copy_to_user` can sleep (it may need to page in user memory). Sleeping while holding a mutex is fine in principle, but if you hold the mutex too long, every other reader/writer is blocked. For most chardevs this is acceptable.
- **Not handling `*ppos` correctly.** A misbehaving driver that ignores `*ppos` reads the buffer-from-the-start every read, leading `cat` into an infinite loop. Always advance `*ppos` by the number of bytes you returned.

37.10 Going deeper

- **`Documentation/filesystems/vfs.rst`**: how the VFS dispatches operations.
- **`Documentation/process/coding-style.rst`**: the kernel's coding standards. `checkpatch.pl` enforces them.
- **`drivers/char/mem.c`**: `/dev/null`, `/dev/zero`, `/dev/random`. Real chardev implementations in canonical style.
- **`drivers/char/misc.c`**: the misc device framework (next chapter, Ch 40). Useful to read once you understand chardev basics.
- **LDD3 Chapter 3**: much more on the chardev driver model.

> Next chapter: **Chapter 38: Auto-creating `/dev/` nodes.** With udev/mdev hot-plug, you stop calling `mknod` by hand. We add `class_create` and `device_create` to the driver.

Chapter 38: Auto-creating /dev/ nodes

> **Privilege boundary:** $ means normal user. # or sudo means root and can change host or target state. After a privileged command, verify the expected device, service, or file appears before continuing. Roll back by undoing the config change or stopping the service you just enabled.

> **What:** `class_create` + `device_create`, the two calls that let your driver tell the kernel "I have a new device. Please broadcast a hot-plug event so user-space creates `/dev/<name>` for me." With these in place you never `mknod` by hand again.
>
> **Why:** real drivers don't burden users with manual `mknod` steps after every `insmod`. Modern Linux uses the **uevent** mechanism, the kernel broadcasts a netlink message describing the new device, and a user-space agent (udev on workstations, mdev on embedded) reacts by creating the right file in `/dev/`, setting permissions, and possibly running scripts. Your driver's only responsibility is to register the device and let the framework do the rest. **udev:** the user-space device manager that reacts to kernel device events and creates policy-driven /dev nodes.
>
> **Focus: the relationship between `/sys/class/...` and `/dev/...`.** The class hierarchy in sysfs is the *source of truth*, that's where the kernel describes what devices exist. The `/dev/` tree is a **shadow** of sysfs maintained by the hot-plug agent. Get this picture right and most "why is my device file missing?" debugging becomes trivial.

38.1 The hot-plug pipeline

When your driver calls `device_create(...)`, this happens:

```
driver: device_create(class, NULL, devid, NULL, "hello")
             │
             ▼
kernel: creates /sys/class/hello/hello/  with attributes:
           ├─ dev              (the "240:0" string)
           ├─ uevent           (writable trigger)
           └─ subsystem        (symlink back up)
             │
             ▼
kernel: emits a uevent of type "add"
           ACTION=add
           DEVNAME=hello
           MAJOR=240
           MINOR=0
           SUBSYSTEM=hello
             │
```

```
                ▼ (netlink broadcast)
   udev/mdev: receives the event, looks up rules
                │
                ▼
   udev/mdev: mknod /dev/hello c 240 0
              chmod 0660 /dev/hello
              chown root:plugdev /dev/hello
```

The driver call (`device_create`) doesn't itself create `/dev/hello`. It creates the **sysfs entry** at `/sys/class/hello/hello/`, which the kernel uses to broadcast the uevent. The actual `/dev/hello` is created by the **listener**.

This is different from how you might imagine it. The kernel does not maintain `/dev/`. It publishes events. User-space decides what to do with them. Different listeners can make wildly different choices (udev creates rich-permission nodes with named symlinks. Mdev creates minimal nodes. Both work).

(Aside: there is a fallback. If `CONFIG_DEVTMPFS=y` and the kernel mounts devtmpfs on `/dev/`, Ch 32, the *kernel itself* auto-creates the device node, no user-space listener required. Then udev/mdev's job becomes just refining permissions and creating symlinks. We'll assume devtmpfs is on, which it is in 99% of modern setups.)

38.2 Adding to the chardev driver

The Chapter 37 example gains a class, a device, and matching cleanup. These additions connect character-device dispatch to the device model.

```
#include <linux/device.h>

struct hello_dev {
    struct cdev cdev;
    dev_t devid;
    struct class *class;
    struct device *device;
    char *buffer;
    size_t buf_len;
    struct mutex lock;
};
```

In `hello_init`, after `cdev_add` succeeds:

```
    hd->class = class_create("hello");
    if (IS_ERR(hd->class)) {
        err = PTR_ERR(hd->class);
        goto del_cdev;
    }

    hd->device = device_create(hd->class, NULL, hd->devid, NULL, "hello");
    if (IS_ERR(hd->device)) {
        err = PTR_ERR(hd->device);
```

```
        goto destroy_class;
    }
```

And the new cleanup labels:

```
destroy_class:
    class_destroy(hd->class);
del_cdev:
    cdev_del(&hd->cdev);
```

In `hello_exit`, balance them (in reverse order):

```
    device_destroy(hd->class, hd->devid);
    class_destroy(hd->class);
    cdev_del(&hd->cdev);
    unregister_chrdev_region(hd->devid, 1);
```

That’s it. Build, load:

```
[root@pa-mini:~]# insmod hello_chrdev.ko
[root@pa-mini:~]# ls -l /dev/hello
crw-rw---- 1 root root 240, 0 May 24 09:30 /dev/hello
[root@pa-mini:~]# echo "ping" > /dev/hello
[root@pa-mini:~]# cat /dev/hello
ping
[root@pa-mini:~]# rmmod hello_chrdev
[root@pa-mini:~]# ls -l /dev/hello
ls: cannot access '/dev/hello': No such file or directory
```

No `mknod` step. The file appears at load and disappears at unload.

38.3 Anatomy of class and device

`class_create`

```
struct class *class_create(struct module *owner, const char *name);
```

Creates a directory `/sys/class/<name>/`. A *class* is a group of devices that share a role, LED, RTC, GPIO chip, network interface, sound card. The class directory holds one entry per device in that group. It also publishes group-level attributes that udev/mdev rules can match on.

> **MCU bridge:** Think of Linux GPIO like the same pin set/reset block you used on STM32, but accessed through a kernel subsystem that owns numbering, direction, interrupts, and user-space exposure. **GPIO:** General-Purpose Input/Output, a pin controlled as a digital input, output, or interrupt source.

The kernel ships dozens of standard classes:

```
[root@pa-mini:~]# ls /sys/class/
backlight   gpio        leds       power_supply  rtc
block       i2c-adapter mdio_bus   pwm           sound
bluetooth   input       net        regulator     spi_master
...
```

When you create your own class (`"hello"`), `/sys/class/hello/` appears. New custom-driver chardevs that *don't fit* an existing class do this, make a class with the driver's name. Drivers that fit an existing class skip `class_create` and register with the subsystem framework directly. For example, an LED driver belongs in `leds` and an RTC in `rtc`. Ch 44–48 cover these subsystems.

`device_create`

```
struct device *device_create(struct class *class, struct device *parent,
                             dev_t devt, void *drvdata,
                             const char *fmt, ...);
```

- **`class`**: which class this device belongs to.
- **`parent`**: the device's parent in the device hierarchy. `NULL` is fine for top-level chardev. Real subsystem drivers set this to the platform device or USB device that hosts them, so sysfs reflects the bus topology.
- **`devt`**: the `dev_t` (major:minor). The kernel writes `MAJOR:MINOR` into the device's `dev` attribute, which is what udev/mdev reads.
- **`drvdata`**: a `void *` stored in the `device`'s `driver_data` field. Use it (or `dev_set_drvdata` later) to attach your own state.
- **`fmt, ...`**: `printf`-style device name. Becomes the directory name in `/sys/class/<class>/<name>/` *and* (via the uevent's `DEVNAME`) the filename in `/dev/`.

A few naming conventions to know:

- For a single-instance device, use `"hello"`.
- For multiple instances, use a format like `"hello%d", index`. The result is `/dev/hello0`, `/dev/hello1`, etc.
- For sub-devices, use a slash: `"input/event%d"` puts the result at `/dev/input/eventN`.

Inspect what got created:

```
[root@pa-mini:~]# ls /sys/class/hello/hello/
dev    power/    subsystem    uevent

[root@pa-mini:~]# cat /sys/class/hello/hello/dev
```

```
240:0

[root@pa-mini:~]# cat /sys/class/hello/hello/uevent
DEVNAME=hello
DEVTYPE=
MAJOR=240
MINOR=0
SUBSYSTEM=hello
```

The `uevent` file is special: reading it prints the current state, **writing** to it re-broadcasts the event. Writing `echo add > uevent` re-triggers the event. This is useful for replaying events on a system that booted before udev was running.

38.4 Picking permissions and ownership

By default, udev/mdev creates `/dev/hello` with permissions 0600 (root-only). That's safe but unhelpful, your test programs running as a regular user can't open the device.

Three places to set device permissions, in order of preference:

A. Udev rule (cleanest)

Create `/etc/udev/rules.d/99-hello.rules`:

```
KERNEL=="hello", MODE="0660", GROUP="plugdev"
```

Reload:

```
[root@pa-mini:~]# udevadm control --reload
[root@pa-mini:~]# udevadm trigger --subsystem-match=hello
```

Now `/dev/hello` is mode 0660, owned by `root:plugdev`. Add your user to the `plugdev` group and they can read/write it.

B. Mdev rule (embedded with BusyBox)

`/etc/mdev.conf`:

```
hello   0:plugdev 0660
```

`mdev`'s syntax is positional, not key=value: name, owner:group, mode. Mdev applies these on every new device event. (Re-trigger by `mdev -s` to apply to already-created files.)

C. Devnode callback in the driver

For permissions that *must* be a property of the driver (because user-space rule files might not exist), set a `devnode` callback on the class:

```
static char *hello_devnode(struct device *dev, umode_t *mode)
{
    if (mode)
        *mode = 0660;   /* readable by group */
    return NULL;
}

/* In init: */
hd->class->devnode = hello_devnode;
```

The kernel's devtmpfs runs `devnode` when creating the device and respects the returned mode. This is the most reliable but least configurable approach, most production systems use udev rules instead.

38.5 Multiple devices in one driver

If your driver controls *N* identical devices (e.g., 4 LEDs, 8 GPIO chips), you want a single driver creating multiple `/dev/` nodes. The pattern:

```
#define N_DEVICES 4

static struct hello_dev *hds[N_DEVICES];
static struct class *hello_class;
static dev_t hello_base_devid;

static int __init hello_init(void)
{
    int i, err;

    err = alloc_chrdev_region(&hello_base_devid, 0, N_DEVICES, "hello");
    if (err) return err;

    hello_class = class_create("hello");
    if (IS_ERR(hello_class)) {
        err = PTR_ERR(hello_class);
        goto unreg;
    }

    for (i = 0; i < N_DEVICES; i++) {
        hds[i] = kzalloc(sizeof(*hds[i]), GFP_KERNEL);
        if (!hds[i]) { err = -ENOMEM; goto unwind; }
        hds[i]->devid = MKDEV(MAJOR(hello_base_devid), i);
        cdev_init(&hds[i]->cdev, &hello_fops);
        hds[i]->cdev.owner = THIS_MODULE;
        err = cdev_add(&hds[i]->cdev, hds[i]->devid, 1);
        if (err) goto unwind;

        device_create(hello_class, NULL, hds[i]->devid, NULL, "hello%d", i);
    }
    return 0;

unwind:
    while (--i >= 0) {
        device_destroy(hello_class, hds[i]->devid);
        cdev_del(&hds[i]->cdev);
```

```
        kfree(hds[i]);
    }
    class_destroy(hello_class);
unreg:
    unregister_chrdev_region(hello_base_devid, N_DEVICES);
    return err;
}
```

After load:

```
[root@pa-mini:~]# ls /dev/hello*
/dev/hello0  /dev/hello1  /dev/hello2  /dev/hello3
```

Each is an independent chardev sharing the same `file_operations`. Use `iminor(filp->f_inode)` to figure out which one a given open is referring to.

38.6 Exposing custom sysfs attributes

A device file is one way to talk to your driver. Sysfs attributes are another, and they're often *better* for set-once configuration or one-shot commands.

Add a `state` attribute:

```
static ssize_t state_show(struct device *dev, struct device_attribute *attr,
                          char *buf)
{
    return sysfs_emit(buf, "loaded\n");   /* bounds-checked since 5.10; prefer over
sprintf */
}

static ssize_t state_store(struct device *dev, struct device_attribute *attr,
                           const char *buf, size_t count)
{
    pr_info("hello: state set to %.*s", (int)count, buf);
    return count;
}

static DEVICE_ATTR_RW(state);   /* creates `state` read+write attr */

/* In init, after device_create succeeds: */
err = device_create_file(hd->device, &dev_attr_state);
```

Now:

```
[root@pa-mini:~]# cat /sys/class/hello/hello/state
loaded
[root@pa-mini:~]# echo run > /sys/class/hello/hello/state
[root@pa-mini:~]# dmesg | tail -1
hello: state set to run
```

`sysfs` files are limited to **PAGE_SIZE** (4 KB on i.MX6ULL) per show callback, but that's plenty for status / configuration. They're great for things like:

- Read-only stats (`bytes_processed`, `irq_count`).
- One-shot commands (`echo reset > /sys/class/.../control`).
- Toggles (`echo 1 > /sys/class/leds/led0/brightness`).

For things like streaming data, stick to the chardev `read`/`write`. Sysfs is for control, not bandwidth.

For multiple attributes, group them and use `device_create_file` per attribute, *or* `sysfs_create_group(&dev->kobj, &attr_group)` for atomic group creation/deletion.

38.7 Key Notes

- `cdev` registration provides operation dispatch, while class/device registration describes the instance in sysfs. Publishing a node before its backing interface is ready creates an access race.
- With devtmpfs, the kernel can create basic device nodes. Udev or mdev supplies user-space naming and access policy, rather than implementing the driver.
- Dynamic major/minor numbers identify an instance, not a permanent product name. Stable names and permissions belong to an explicit device-management policy.
- A shared class can contain multiple devices, but each instance needs its own state and complete failure unwind. Partial registration is a lifetime problem, not merely a naming error.
- Sysfs attributes suit small status and configuration values. Streaming data belongs in an interface with an explicit transfer and readiness contract.
- Removal withdraws published interfaces before releasing their backing state. Hot-plug events do not by themselves synchronize every application that may already hold a device open.

38.8 Pitfalls

- **Calling `device_create` before `cdev_add`.** Device node appears but `open` on it returns `-ENXIO` because no cdev is registered for that major. Always `cdev_add` first, then `device_create`. Cleanup in reverse order.
- **Forgetting `device_destroy` in cleanup.** The `/dev/` node lingers (kernel side), udev/mdev never gets the "remove" event. Next load creates `/dev/hello` *again*, double-registered. Eventually you'll trip over name conflicts.

- **`IS_ERR(class)` vs `class == NULL`.** `class_create` returns an `ERR_PTR` on failure, not `NULL`. Test with `IS_ERR(class)` and recover with `PTR_ERR(class)`. Same for `device_create`.
- **Race between insmod and udev.** If your user-space code does `insmod hello.ko && cat /dev/hello`, the second part may run before udev has created `/dev/hello`. Mitigation: use `udevadm settle` between `insmod` and the access, or have your user-space code retry with a short delay.
- **Class name collisions.** Two drivers trying to register a class with the same name fail loudly. Use unique names. If you're adding a driver to an existing subsystem, use that subsystem's `register` API (e.g., `led_classdev_register`) instead of creating a competing class.
- **Sysfs attribute permissions.** `DEVICE_ATTR_RW` creates 0644 attributes. For root-only writes, use `DEVICE_ATTR(name, 0640, show, store)` or one of `DEVICE_ATTR_RO`, `DEVICE_ATTR_WO`, `DEVICE_ATTR_ADMIN_RO`.
- **Returning >PAGE_SIZE from a `show` callback.** Silently truncated by sysfs. Don't use sysfs to stream large data. Use a chardev or `debugfs` instead.

38.9 Going deeper

- **`Documentation/driver-api/driver-model/overview.rst`**: the device model is much richer than what we use here. Classes are one slice of it.
- **`Documentation/admin-guide/devices.rst`**: udev/mdev rule syntax in detail.
- **`drivers/leds/led-class.c`**: a real subsystem class, fully fleshed out. Read it once.
- **`Documentation/filesystems/sysfs.rst`**: the sysfs design rationale. Explains why each file is limited to PAGE_SIZE and how the kobject hierarchy works.

> Next chapter: **Chapter 39: Platform drivers + device tree.** With manual device registration understood, we move to the *real* way Linux drivers describe themselves: a `platform_driver` that gets matched to a device-tree `compatible` string, with `probe`/`remove` doing what `module_init`/`module_exit` did up until now.

Chapter 39: Platform drivers + device tree

> **What:** the `platform_driver` model, the canonical way Linux describes *on-SoC* peripherals (UART, I²C controllers, GPIO blocks, PWM, ADC, all the things memory-mapped into the SoC's address space). A platform driver registers with the kernel saying "I drive devices that match `compatible = "vendor,part"`". The kernel walks the device tree, finds matching nodes, and invokes the driver's `probe()` once per match. **MCU bridge:** Think of Linux PWM like an MCU timer output channel, except the driver exposes period, duty cycle, polarity, and enable state through a subsystem. **MCU bridge:** Think of Linux GPIO like the same pin set/reset block you used on STM32, but accessed through a kernel subsystem that owns numbering, direction, interrupts, and user-space exposure. **PWM:** Pulse-Width Modulation, a timer output whose duty cycle controls average power or encodes timing. **GPIO:** General-Purpose Input/Output, a pin controlled as a digital input, output, or interrupt source.
>
> **Why:** the kernel doesn't probe address ranges blindly looking for hardware (that's how PC BIOSes work, and it doesn't scale to SoCs with no buses to enumerate). It needs to be **told** what devices exist and where their registers live, that's exactly what the device tree does. The `platform_driver` API is the kernel-side half of the DT contract: you describe the driver, the DT describes the device, the kernel matches them.
>
> **Focus: driver and device are separate**. The driver is a `.ko` (or built-in code) that knows *how* to talk to a hardware block. The device is a DT node that says *where the block is* (registers, IRQs, clocks, pins). The bus (platform bus, here) matches them by `compatible` string. Once this trinity clicks, every subsystem driver in the kernel looks the same.

39.1 Why "platform" exists

> **Privilege boundary:** $ means normal user. # or sudo means root and can change host or target state. After a privileged command, verify the expected device, service, or file appears before continuing. Roll back by undoing the config change or stopping the service you just enabled.

You've already seen platform devices in passing. Open any imx6ull DT and you see things like:

```
gpio1: gpio@209c000 {
    compatible = "fsl,imx6ul-gpio", "fsl,imx35-gpio";
```

```
    reg = <0x209c000 0x4000>;
    interrupts = <0 66 IRQ_TYPE_LEVEL_HIGH>, <0 67 IRQ_TYPE_LEVEL_HIGH>;
    clocks = <&clks IMX6UL_CLK_GPIO1>;
    gpio-controller;
    #gpio-cells = <2>;
    interrupt-controller;
    #interrupt-cells = <2>;
};
```

This is a **platform device**: a hardware block on the SoC, described by a DT node, with no enumerable bus connecting it (compared to USB, PCI, or even I²C, where a discovery protocol enumerates children). The CPU just has memory-mapped registers at a fixed physical address and an IRQ line connected to the GIC.

> **MCU bridge:** Think of the GIC like the Cortex-M NVIC scaled up for Cortex-A: it routes peripheral interrupts to CPU cores and has separate distributor and CPU-interface blocks. **IRQ:** interrupt request, the signal path that tells the CPU or interrupt controller that hardware needs service. **GIC:** ARM's Generic Interrupt Controller, the Cortex-A interrupt router roughly analogous to NVIC on Cortex-M.

The **platform bus** in the kernel is a virtual abstraction over this "no bus" case. It exists only to give the device model something to attach to. At boot, every DT node whose parent does not name a real bus becomes a platform device. In practice that means everything directly under the SoC node. When you `insmod` a `platform_driver`, the platform bus walks the device list looking for matches.

There are two kinds of buses in Linux:

1. **Enumerable**: USB, PCI, I²C (devices can be discovered by polling). The bus driver enumerates. Child devices appear automatically.
2. **Non-enumerable**: Platform, plus a few others. Devices must be described externally (DT, ACPI, board file). The platform bus is the catch-all for SoC peripherals.

Almost everything on i.MX6ULL is a platform device: GPIO blocks, UARTs, I²C/SPI/eCSPI controllers, PWM, ADC, timers, FlexCAN, Ethernet MAC, USB OTG, LCDIF. (The devices on an I²C bus are I²C-bus children, not platform devices.)

> **MAC:** Media Access Control in networking and radio chapters. It is the layer that owns framing and medium access.

39.2 The pieces

To write a platform driver for an on-SoC peripheral, you need three things:

1. **A device-tree node** describing the hardware (`compatible`, `reg`, `interrupts`, `clocks`, pinctrl). This usually exists already, the SoC's vendor wrote it.
2. **A `platform_driver` struct** in your code that declares which `compatible` strings it handles and points to your `probe` / `remove` functions.
3. **A `probe()` function** that does what `module_init` did before: claim resources, request IRQs, register the chardev/class, set up internal state. Returns 0 on success, negative `errno` on failure.

When the kernel finds a DT node whose `compatible` matches your driver, the bus calls your `probe(struct platform_device *pdev)`. Your `probe` finds resources via `pdev->dev.of_node` (the DT node) or via `platform_get_resource()`. When the driver is unloaded, or the device is removed (rare on SoC peripherals. Common on hotpluggable hardware), `remove()` runs.

39.3 A minimal example: a "demo" platform driver

The following platform-driver case shows matching, resource lookup, and managed register mapping. The DT address is illustrative, not a MINI hardware resource or a prerequisite for later chapters.

Device description

The example device description is:

```
&{/} {
    demo0: demo@1000 {
        compatible = "linuxlearn,demo";
        reg = <0x00001000 0x100>;
        status = "okay";
    };
};
```

The address `0x1000` is a placeholder. It does not identify safe unused memory on a physical board. A real resource must describe identified hardware and must not overlap another owner's address range.

For this description, the corresponding DT view would contain:

```
[root@pa-mini:~]# ls /sys/firmware/devicetree/base/demo@1000/
compatible  name  reg  status
```

```
[root@pa-mini:~]# cat /sys/firmware/devicetree/base/demo@1000/compatible
linuxlearn,demo
```

The DT node exists, but no driver has claimed it, `dmesg` shows nothing yet because no `platform_driver` matches.

Driver implementation

`demo.c`:

```
#include <linux/init.h>
#include <linux/module.h>
#include <linux/platform_device.h>
#include <linux/of.h>
#include <linux/io.h>

struct demo_priv {
    void __iomem *base;
    int irq;
};

static int demo_probe(struct platform_device *pdev)
{
    struct demo_priv *priv;
    struct resource *res;

    dev_info(&pdev->dev, "probe: matched compatible '%s'\n",
             pdev->dev.of_node->name);

    priv = devm_kzalloc(&pdev->dev, sizeof(*priv), GFP_KERNEL);
    if (!priv)
        return -ENOMEM;

    res = platform_get_resource(pdev, IORESOURCE_MEM, 0);
    if (!res)
        return dev_err_probe(&pdev->dev, -EINVAL, "no memory resource\n");

    priv->base = devm_ioremap_resource(&pdev->dev, res);
    if (IS_ERR(priv->base))
        return PTR_ERR(priv->base);

    dev_info(&pdev->dev, "registers at %pa, mapped to %p\n",
             &res->start, priv->base);

    platform_set_drvdata(pdev, priv);
    return 0;
}

static void demo_remove(struct platform_device *pdev)
{
    struct demo_priv *priv = platform_get_drvdata(pdev);
    dev_info(&pdev->dev, "remove\n");
    (void)priv;   /* nothing to free; devm_* handles all */
}
```

```
static const struct of_device_id demo_of_match[] = {
    { .compatible = "linuxlearn,demo" },
    { /* sentinel */ }
};
MODULE_DEVICE_TABLE(of, demo_of_match);

static struct platform_driver demo_driver = {
    .driver = {
        .name           = "demo",
        .of_match_table = demo_of_match,
    },
    .probe  = demo_probe,
    .remove = demo_remove,
};

module_platform_driver(demo_driver);

MODULE_LICENSE("GPL");
MODULE_DESCRIPTION("Demo platform driver");
```

That is the full template. Look at the four pieces that matter.

Piece A: `of_match_table` + `MODULE_DEVICE_TABLE`

```
static const struct of_device_id demo_of_match[] = {
    { .compatible = "linuxlearn,demo" },
    { /* sentinel */ }
};
MODULE_DEVICE_TABLE(of, demo_of_match);
```

The `of_device_id` table lists every `compatible` string this driver handles. The kernel's DT matcher compares each entry against every DT node's `compatible` (which may be a list. First match wins).

`MODULE_DEVICE_TABLE(of, demo_of_match)` exposes the table to `depmod`. When the user runs `modprobe`, `depmod` knows which `.ko` to autoload for a given `compatible` string. This is how the kernel can auto-load drivers at boot: parse DT → find unmatched compatible → search through modules' DEVICE_TABLE → modprobe the right one.

If you omit `MODULE_DEVICE_TABLE`, your driver still works when manually `insmod`'d, but **auto-loading from DT will silently fail**. Always include it.

Piece B: `platform_driver` + `module_platform_driver`

```
static struct platform_driver demo_driver = {
    .driver = {
        .name           = "demo",
        .of_match_table = demo_of_match,
    },
    .probe  = demo_probe,
    .remove = demo_remove,
};
```

```
module_platform_driver(demo_driver);
```

The `platform_driver` struct ties the matching table to your callbacks. The `module_platform_driver()` macro expands to a `module_init` + `module_exit` pair that calls `platform_driver_register` / `platform_driver_unregister`. **You no longer write `module_init` / `module_exit` by hand**, the macro does it. (Look it up. It's literally a one-liner each way.)

Piece C: `devm_*` (managed) allocations

```
priv = devm_kzalloc(&pdev->dev, sizeof(*priv), GFP_KERNEL);
...
priv->base = devm_ioremap_resource(&pdev->dev, res);
```

The `devm_` prefix is a kernel pattern that means **device-managed**: resources allocated this way are *automatically freed when the device goes away*. No `kfree`, no `iounmap`, no `free_irq` in your `remove()` function, it's all handled.

`devm_*` is the biggest readability gain in modern kernel code. Compare:

```
/* Without devm_ */
priv = kzalloc(sizeof(*priv), GFP_KERNEL);
if (!priv) return -ENOMEM;
priv->base = ioremap(res->start, resource_size(res));
if (!priv->base) { kfree(priv); return -ENOMEM; }
priv->buffer = kmalloc(4096, GFP_KERNEL);
if (!priv->buffer) { iounmap(priv->base); kfree(priv); return -ENOMEM; }
...
/* And in remove(): */
kfree(priv->buffer);
iounmap(priv->base);
kfree(priv);
```

vs.

```
/* With devm_ */
priv = devm_kzalloc(&pdev->dev, sizeof(*priv), GFP_KERNEL);
if (!priv) return -ENOMEM;
priv->base = devm_ioremap_resource(&pdev->dev, res);
if (IS_ERR(priv->base)) return PTR_ERR(priv->base);
priv->buffer = devm_kmalloc(&pdev->dev, 4096, GFP_KERNEL);
if (!priv->buffer) return -ENOMEM;
...
/* remove() can be empty (or omitted entirely if there's nothing to do). */
```

Every common resource has a `devm_` variant: `devm_kmalloc`, `devm_kzalloc`, `devm_kasprintf`, `devm_ioremap_resource`, `devm_request_irq`, `devm_clk_get`, `devm_regulator_get`, `devm_reset_control_get`, etc. Use them. The error-path goto cascade from Ch 37 mostly disappears.

Piece D: `dev_err_probe`

```
return dev_err_probe(&pdev->dev, -EINVAL, "no memory resource\n");
```

`dev_err_probe` is two functions in one: it logs the message **and** returns the errno. If the errno is `-EPROBE_DEFER` (a special “try again later” return), it logs at debug level instead of error to avoid spam (because probe deferrals are usually transient). For all other errno values, it logs at error level. Always prefer it to `dev_err(...) + return -EINVAL;`.

39.4 Building and loading

`Makefile` (same as Ch 36’s):

```
obj-m += demo.o
KDIR ?= /home/$(USER)/linux-imx6ull/build
all:
    $(MAKE) -C $(KDIR) ARCH=arm CROSS_COMPILE=arm-none-linux-gnueabihf- M=$(PWD) modules
```

On the target:

```
[root@pa-mini:~]# insmod demo.ko
[root@pa-mini:~]# dmesg | tail -2
demo demo@1000: probe: matched compatible 'demo'
demo demo@1000: registers at 0x00001000, mapped to (ptrval)

[root@pa-mini:~]# rmmod demo
[root@pa-mini:~]# dmesg | tail -1
demo demo@1000: remove
```

The probe fired automatically, no manual `mknod` or device registration. The DT node was there. The driver matched it. `probe()` ran with all the right context.

Verify in sysfs

```
[root@pa-mini:~]# ls /sys/bus/platform/drivers/demo/
bind  module  uevent  unbind  demo@1000

[root@pa-mini:~]# ls /sys/bus/platform/devices/demo@1000/
driver  driver_override  modalias  of_node  power  subsystem  uevent

[root@pa-mini:~]# readlink /sys/bus/platform/devices/demo@1000/driver
../../../../bus/platform/drivers/demo

[root@pa-mini:~]# cat /sys/bus/platform/devices/demo@1000/modalias
of:Ndemo@1000T(null)Clinuxlearn,demo
```

`/sys/bus/platform/drivers/demo/` lists devices currently bound to this driver (`demo@1000` here). `/sys/bus/platform/devices/demo@1000/` shows the device, its driver, and a `modalias` that `depmod` uses to autoload the right module.

39.5 Manual bind / unbind

Sysfs lets you unbind a device from its driver and rebind later, without unloading the module:

> **sysfs:** a kernel-generated filesystem under /sys that exposes devices, drivers, and attributes.

```
[root@pa-mini:~]# echo demo@1000 > /sys/bus/platform/drivers/demo/unbind
[root@pa-mini:~]# dmesg | tail -1
demo demo@1000: remove

[root@pa-mini:~]# echo demo@1000 > /sys/bus/platform/drivers/demo/bind
[root@pa-mini:~]# dmesg | tail -1
demo demo@1000: probe: matched compatible 'demo'
```

Useful in development for re-probing a device after a hardware glitch, without a reboot. Also useful in production: unbind unused hardware to drop its clocks.

39.6 Getting more from the DT

`platform_get_resource` returns one resource (`reg` or `interrupt` etc). For richer DT data, use the `of_*` API directly:

```
/* Read a u32 property */
u32 val;
if (of_property_read_u32(pdev->dev.of_node, "linuxlearn,speed-hz", &val))
    val = 100000;   /* default */

/* Read a string */
const char *mode;
if (of_property_read_string(pdev->dev.of_node, "linuxlearn,mode", &mode))
    mode = "polled";

/* A GPIO descriptor */
struct gpio_desc *reset_gpio;
reset_gpio = devm_gpiod_get(&pdev->dev, "reset", GPIOD_OUT_LOW);
if (IS_ERR(reset_gpio))
    return PTR_ERR(reset_gpio);

/* A clock */
struct clk *clk;
clk = devm_clk_get(&pdev->dev, "main");
if (IS_ERR(clk))
    return PTR_ERR(clk);
clk_prepare_enable(clk);
```

Notice these are all subsystem APIs (`of_*`, `gpiod_*`, `clk_*`), not raw DT-parsing code. The DT layer is the data layer. Subsystems above it provide *typed* access. We'll see clocks in Ch 50, GPIOs in Ch 44, etc.

39.7 Two probe-related patterns to know

Pattern: `probe_defer` for ordering

A driver's `probe()` may depend on something that isn't ready yet, e.g., the regulator the driver wants hasn't been registered yet because its own driver hasn't probed. The driver returns `-EPROBE_DEFER`:

```
priv->vcc = devm_regulator_get(&pdev->dev, "vcc");
if (IS_ERR(priv->vcc))
    return dev_err_probe(&pdev->dev, PTR_ERR(priv->vcc), "no vcc regulator\n");
```

That's the whole pattern. `dev_err_probe` already handles `-EPROBE_DEFER` quietly (debug-level log instead of error-level), so the manual `if (PTR_ERR(...) == -EPROBE_DEFER) return -EPROBE_DEFER;` check that older drivers carry is redundant, drop it.

The kernel notes the deferred device and retries it after every other probe attempt, until all probes have stabilised. This means you don't have to manage init order manually across drivers. The kernel does it for you.

Pattern: shutdown vs remove

`remove()` is called when the driver is unloaded or the device is unbound. `shutdown()` is called during system shutdown/reboot, only on devices that need to be quiesced (DMA stopped, watchdog disabled, etc.). For most drivers `remove()` suffices. For drivers that own DMA engines or watchdogs, add a `.shutdown = my_shutdown`. It runs in atomic context, keep it short.

> **MCU bridge:** Think of DMA like the MCU DMA controller you used for UART or SPI, but with cache coherency, scatter-gather descriptors, and kernel ownership rules added. **DMA:** Direct Memory Access. Hardware moves data to or from memory without the CPU copying each byte.

39.8 Key Notes

- Device Tree describes a hardware instance, the driver supplies behavior, and the platform bus matches the two. One driver can bind to several devices with separate private state.
- `probe()` establishes an instance's resources and interfaces. Module registration is a different lifetime from a successful device binding.

- `compatible`, resource ranges, interrupts, clocks, and GPIO properties are binding contracts. A matching string does not establish that the described hardware exists or is safe to access.
- `-EPROBE_DEFER` represents a supplier dependency that is not ready. It is not a general-purpose timer for retrying an arbitrary condition.
- Managed allocations simplify release on probe failure and unbind. They do not replace stopping DMA, draining work, or preventing callbacks from using released state.
- Unbind removes an instance without necessarily unloading the driver code. Open interfaces and asynchronous activity must respect that distinction.

39.9 Pitfalls

- **DT node has wrong `status`.** A node with `status = "disabled"` is skipped by the platform bus. Many vendor DTs ship peripherals as `disabled`. You must overlay `status = "okay";` to activate.
- **`compatible` typo.** A typo in either the DT or the driver's `of_match_table` is silent: no probe, no error message. Always cross-check both spellings.
- **Forgetting `MODULE_DEVICE_TABLE`.** Driver works manually but won't auto-load. Symptom: must `modprobe demo` by hand at every boot. Fix is a one-liner.
- **Calling `kfree` on a `devm_kmalloc` pointer.** Double-free. Symptom: memory corruption that may take hours to manifest. Pick one allocator and stay consistent. If you `devm_kmalloc`, never `kfree` it. If you `kmalloc`, never let it slip past `remove` without `kfree`.
- **Calling sleeping functions in atomic context.** `dev_err_probe` is fine. `devm_kmalloc(... GFP_KERNEL)` is fine in probe. But if `probe` itself is called from atomic context (rare for platform drivers but real for some buses), use `GFP_ATOMIC`. Misuse triggers `BUG: sleeping function called from invalid context` at runtime.
- **Driver and device names with hyphens vs underscores.** Some subsystems are picky. The convention: driver `.name` and `compatible` use **hyphens** ("snps,dwc-mshc"). Avoid underscores. The kernel won't reject them but tools may treat them differently.
- **Trying to mix `platform_driver` and direct chardev registration.** It works, many real drivers do both: probe registers the chardev, but the **order matters**. Always do chardev/class/device_create *inside* `probe`, not at module load time. Otherwise a probe failure leaves a partially-registered chardev with no backing.

39.10 Going deeper

- **`Documentation/driver-api/driver-model/platform.rst`**: the platform bus's official documentation.

- **`Documentation/devicetree/bindings/`**: the YAML bindings (Ch 27A). The source-of-truth for what each `compatible` expects.
- **`drivers/gpio/gpio-mxc.c`**: i.MX GPIO controller driver. A small, clean platform driver. Read it.
- **`Documentation/driver-api/devres.rst`**: full list of `devm_*` helpers.
- **`MAINTAINERS`**: when your driver is upstream-quality, this is where you find which subsystem maintainer it belongs to.

> **End of foundation chapters (Ch 36–39).** With LKM, chardev, hot-plug, and platform-driver patterns understood, you have the skeleton every subsequent driver in Part VI hangs off. The next chapters (40–43) add the *behaviors* that real drivers need: the misc-device shortcut for trivial chardevs, concurrency primitives, sleeping/polling, and interrupts. **LKM:** Loadable Kernel Module, kernel code compiled as a .ko file and inserted at runtime.

> Next chapter: **Chapter 40: The misc framework.** For dead-simple chardevs that don’t deserve their own class, `miscdevice` is a one-call shortcut that handles dev_t allocation, class registration, and device-node creation in one shot.

Chapter 40: The misc framework

> **What:** `miscdevice`, a one-call shortcut that turns 80 lines of “allocate major, register cdev, create class, create device” boilerplate into 10 lines. Use it for simple character devices that don’t fit a standard subsystem.
>
> **Why:** the Ch 37–38 pattern (alloc_chrdev_region + cdev_init + cdev_add + class_create + device_create) is correct but verbose. The misc framework is the kernel’s pre-canned version. Many real drivers, `/dev/watchdog`, `/dev/hwrng`, `/dev/rfkill`, `/dev/btrfs-control`, `/dev/loop-control`, use it. Knowing when to use it saves you the chardev boilerplate.
>
> **Focus: misc is just chardev with shared major 10**. There’s no new mechanism, the kernel reserves major 10 for “miscellaneous” devices and the misc framework hands out minor numbers within that major. Your driver provides just minor + name + fops. The rest is done for you.

40.1 When to use misc (and when not to)

Use misc when:

- You need a chardev for a single device (or a small fixed number of them).
- The device doesn’t fit any existing subsystem (LED, RTC, GPIO, input, sound, etc.). If it does fit, register with that subsystem instead, you get richer integration (sysfs attributes, common ioctls, user-space tooling).

> **MCU bridge:** Think of Linux GPIO like the same pin set/reset block you used on STM32, but accessed through a kernel subsystem that owns numbering, direction, interrupts, and user-space exposure. **GPIO:** General-Purpose Input/Output, a pin controlled as a digital input, output, or interrupt source. **sysfs:** a kernel-generated filesystem under /sys that exposes devices, drivers, and attributes.

- You don’t need to publish custom class-level attributes (those go on `/sys/class/<your-class>/`).

Don’t use misc when:

- You need *many* dynamically-numbered instances (loop devices, USB serial ports). Misc minor space is limited.

- You’re writing a driver that should integrate with a framework (LED → `leds-class`. Input device → `input_register_device`. Sound → ALSA/ASoC. Network → `netdev`). The Part VI subsystem chapters will cover these.

> **ASoC:** ALSA System-on-Chip, the embedded audio layer that connects CPU audio ports, codecs, and board wiring. **ALSA:** Linux’s kernel and user-space audio stack.

For these in-between cases, simple chardev, one or two instances, no matching framework, misc fits.

40.2 The API

```
#include <linux/miscdevice.h>
```

```
struct miscdevice {
    int                 minor;      /* MISC_DYNAMIC_MINOR for "let kernel pick" */
    const char          *name;      /* device name, becomes /dev/<name> */
    const struct file_operations *fops;
    struct list_head    list;
    struct device       *parent;
    struct device       *this_device;
    const struct attribute_group **groups;
    const char          *nodename;  /* optional: name override for /dev */
    umode_t             mode;       /* permission mode */
};

int  misc_register(struct miscdevice *misc);
void misc_deregister(struct miscdevice *misc);
```

Two functions. The full API.

40.3 Hello, misc

The Chapter 37 chardev can expose the same operations through a misc device. The manual registration path is:

```
/* Ch 37 — by hand */
err = alloc_chrdev_region(&hd->devid, 0, 1, "hello");      /* 1 line */
cdev_init(&hd->cdev, &hello_fops);                          /* 2 */
hd->cdev.owner = THIS_MODULE;                               /* 3 */
err = cdev_add(&hd->cdev, hd->devid, 1);                    /* 4 */
hd->class = class_create("hello");             /* 5 */
hd->device = device_create(hd->class, NULL, hd->devid, NULL, "hello");  /* 6 */
/* +cleanup labels, +reverse-order unwind */
```

The after, using misc:

```
/* Ch 40 — using misc */
static struct miscdevice hello_misc = {
    .minor = MISC_DYNAMIC_MINOR,
    .name  = "hello",
    .fops  = &hello_fops,
    .mode  = 0660,
};

static int __init hello_init(void)
{
    return misc_register(&hello_misc);
}

static void __exit hello_exit(void)
{
    misc_deregister(&hello_misc);
}
```

Six lines to register, one to deregister. The kernel:

- Allocates a free minor in the misc major (10).
- Creates `/dev/hello` automatically via the device-model hot-plug.
- Sets the mode (0660 here, so the device is group-readable/writable).
- Increments your module's refcount on every open.

Load:

```
[root@pa-mini:~]# insmod hello_misc.ko
[root@pa-mini:~]# ls -l /dev/hello
crw-rw---- 1 root root 10, 122 May 24 09:30 /dev/hello
```

The major is `10` (the misc-class major), the minor is whatever the kernel picked. `cat /proc/devices` confirms:

```
[root@pa-mini:~]# cat /proc/devices | head -10
Character devices:
...
 10 misc
...
```

Inside `/sys/class/misc/`:

```
[root@pa-mini:~]# ls /sys/class/misc/
device-tree-id  hello  hpet  hwrng  loop-control  rfkill  ...
```

Your device sits alongside the kernel's other misc devices.

40.4 Real-world examples to study

The kernel has dozens of misc drivers, small files, easy to read. The instructive ones:

Driver	File	What it does
`loop-control`	`drivers/block/loop.c`	Helper to create/free loop devices
`watchdog`	`drivers/watchdog/watchdog_dev.c`	The `/dev/watchdog` interface (Ch 51A)
`hwrng`	`drivers/char/hw_random/core.c`	The hardware RNG framework
`rfkill`	`net/rfkill/core.c`	WiFi/Bluetooth kill-switch interface

Read any of them: you'll see `misc_register` used exactly as we used it, surrounded by the subsystem's own initialization. Misc is *one piece* of a real driver, not the whole driver.

40.5 Comparison table

Feature	Manual chardev (Ch 37–38)	misc
Boilerplate lines	~30	~6
Major number	Dynamic (yours)	Fixed (10)
Minor numbers	All yours, 0..N-1	One from misc's pool
Custom class in sysfs	Yes (you control name)	No (lives under `/sys/class/misc/`)
Multiple instances of same driver	Easy	Each needs its own `miscdevice` struct

Feature	Manual chardev (Ch 37–38)	misc
Right for:	“Custom subsystem feel”; many instances	Simple one-off chardev

40.6 Key Notes

- Misc devices share major 10 and obtain an individual minor. The framework reduces registration plumbing without changing the `file_operations` contract.
- A standard subsystem is preferable when it already represents the device’s role. Misc does not supply input, sensor, audio, or networking semantics.
- Each registered instance needs a persistent `miscdevice` and backing state. A stack-local registration object cannot outlive the function that created it.
- Platform binding and misc registration solve different problems: the first connects hardware to a driver, the second exposes a user-space endpoint.
- Deregistration prevents new access through the registered endpoint. Existing open files and asynchronous users require driver-owned lifetime handling before state is freed.

40.7 Pitfalls

- **`miscdevice` must outlive the registration**, because the misc layer holds a pointer to it. Don’t put it on the stack of `init()`, make it `static` or allocate it with `kmalloc`.
- **Multiple modules registering with the same `name`.** `misc_register` returns `-EBUSY`. The first one wins. Pick a unique name.
- **Forgetting `.mode`.** Default is 0600 (root-only). Set 0660 (or whatever) explicitly if you want non-root access.
- **Need a custom class but using misc.** Misc devices all live in `/sys/class/misc/`. If you need your own class with shared attributes across instances, drop back to manual chardev (Ch 37–38), misc isn’t the right tool.
- **Hot-add/hot-remove during shutdown.** If you `misc_deregister` while a `/dev/hello` is still open, the kernel safely refuses the unregister via refcount. Cleanly close all fds first.

40.8 Going deeper

- **`Documentation/driver-api/miscellaneous.rst`**: the misc framework's official short doc.
- **`include/linux/miscdevice.h`**: the small header with all the well-known minor numbers reserved (e.g., minor 130 = `/dev/watchdog`).
- **`drivers/char/misc.c`**: implementation. ~250 lines. Easy read.

> Next chapter: **Chapter 41: Concurrency in the kernel.** With a working chardev (whether manual or misc) we now look at the synchronization primitives every driver needs: atomics, spinlocks, mutexes, and when to use which.

Chapter 41: Concurrency in the kernel

> **What:** the kernel synchronization toolbox, `atomic_t`, `spinlock_t`, `mutex`, `rwlock_t`, `semaphore`, RCU, and the per-CPU and memory-barrier primitives that back them. By the end you can answer "what lock should I use?" by asking three questions.
>
> **Why:** every driver that touches shared state in two contexts (a process and an interrupt. A process and a timer. Two processes via `open(2)`) has a race. Pick the wrong primitive and you hit one of two failure modes: a silent data-corruption race, or a lockup so deep that `dmesg` cannot tell you what happened.
>
> **Focus: the three questions**, *who else can be running this code at the same time?* (process, softirq, hardirq, multiple CPUs). *is the critical section allowed to sleep?* (mutex if yes, spinlock if no). *is the access read-mostly?* (RCU if yes). Get these three answers right and the API choice is mechanical.

41.1 Why the kernel is concurrent

Unlike a single-core MCU running a single firmware loop, the kernel is concurrent on multiple axes:

1. **Multi-CPU (SMP).** The i.MX6ULL is single-core, but the kernel is built `CONFIG_SMP=y` and *behaves* as if it were SMP. Your driver may run on different CPUs at different times.
2. **Preemption.** A process running your driver code can be preempted by a higher-priority process. Your local variables are safe. Your shared variables aren't.
3. **Interrupts.** A hardware interrupt can preempt your driver mid-instruction. If both your driver and its interrupt handler touch the same variable, you have a race.
4. **Softirqs and tasklets.** A "bottom half", softirq, tasklet, or work queue, can run on the same CPU as your driver, interleaving at quantum boundaries.
5. **Multiple syscalls.** Two processes both calling `read(fd)` on your device file are racing inside your driver simultaneously.

Concurrency is the default. Every shared variable needs a plan. Every variable that's read or written from more than one of the contexts above needs locking, atomics, or one of the lock-free patterns we'll get to.

41.2 The decision tree

Three questions. Answer them and you've picked your primitive.

Question 1: What contexts can access the data?

- **Process only** (no IRQ handler, no softirq) → mutex or semaphore.

> **MCU bridge:** Think of an IRQ like an EXTI/NVIC interrupt path, except Linux splits the hard interrupt from deferred work and must share lines across drivers. **IRQ:** interrupt request, the signal path that tells the CPU or interrupt controller that hardware needs service.

- **Process + IRQ** → spinlock (and the IRQ variant, `spin_lock_irqsave`).
- **Process + softirq/tasklet** → spinlock with `_bh` variant.
- **IRQ only**, single CPU → no lock needed (interrupts are serialized on one CPU). With SMP, still need a spinlock.

Question 2: Is the critical section allowed to sleep?

- **Yes** (calls `kmalloc(GFP_KERNEL)`, `copy_to_user`, anything else that can block) → must be mutex (or other sleeping lock). Spinlocks forbid sleeping while held.
- **No** (just arithmetic, register read/write) → spinlock is fine.

Question 3: Is the access read-mostly?

- **Yes** (e.g., a list updated rarely but read in every IRQ) → consider RCU.
- **No** → standard lock is fine.

That's it. Three questions, one primitive. Let's see them in action.

Context cheat sheet, what may sleep, what may not

This is *the* table to memorize. Every later driver chapter (i2c, SPI, IIO, regmap, DMA, USB, etc.) assumes you know it. We will not repeat it.

> **MCU bridge:** Think of DMA like the MCU DMA controller you used for UART or SPI, but with cache coherency, scatter-gather descriptors, and kernel ownership rules added. **MCU bridge:** Think of regmap like a typed wrapper around your read_reg() and write_reg() helpers, with caching, locking, and bus differences handled centrally. **DMA:** Direct Memory Access. Hardware moves data to or from memory without the CPU copying each byte. **IIO:** Industrial I/O, Linux's subsystem for sensors, ADCs, DACs, and buffered sampled data. **regmap:** a kernel helper that wraps register reads and writes over I2C, SPI, or MMIO.

Context	Examples	May sleep?	Locks you may take	Locks you must avoid
Process (syscall, `probe()`, file_operations)	`read`, `write`, `ioctl`, `open`	**yes**	`mutex_lock`, `spin_lock`, `down`, `wait_event`, `i2c_smbus_*`, `spi_sync`, `kmalloc(GFP_KERNEL)`	,
Threaded IRQ (`request_threaded_irq` bottom half)	the `_threaded_fn` argument	**yes**	same as process context	,
Workqueue (`schedule_work`, `delayed_work`)	`INIT_WORK` handlers	**yes**	same as process context	,
Softirq / tasklet / timer	`softirq` callbacks, `mod_timer` callbacks	**no**	`spin_lock`, `atomic_*`, `mutex_trylock` (only)	`mutex_lock`, `i2c_smbus_*`, `msleep`, `kmalloc(GFP_KERNEL)`
Hard IRQ (top half of `request_irq`)	the `_handler` argument when there's no threaded fn	**no, strictly atomic**	`spin_lock_irqsave`, `atomic_*`	everything that may sleep, including i2c/spi transfers, `printk` with KERN_DEBUG at high rate
Holding a spinlock	inside `spin_lock` … `spin_unlock`	**no**	nested spinlocks (different order rule)	everything that may sleep

Rule of thumb for sensor / bus drivers: if you need to do an `i2c_smbus_*` or `spi_sync` (both can block on bus contention), you must be in **process context** or a **threaded IRQ**, never in a hard IRQ handler or under a spinlock. Use `request_threaded_irq` and put the bus access in the threaded half.

We refer back to this table from many places. When a later chapter says "this runs in atomic context," consult the rows above.

41.3 Atomic operations

> **Security boundary:** Do not burn fuses, enroll production keys, or sign release images when exploring example code. Use throwaway keys and back up the unsigned image plus the key directory before testing irreversible security flows.

Sometimes you just want to increment a counter from multiple contexts. A lock is overkill. The kernel exposes **atomic types** that compile to a single bus-locked instruction.

```
#include <linux/atomic.h>

atomic_t counter = ATOMIC_INIT(0);

atomic_inc(&counter);
atomic_dec(&counter);
atomic_add(7, &counter);
int n = atomic_read(&counter);

/* Test-and-modify */
if (atomic_dec_and_test(&counter)) {
    /* counter just hit zero */
}

/* Compare and swap */
int old = atomic_cmpxchg(&counter, 0, 1);
if (old == 0) {
    /* we set 0 → 1, exclusive use */
}
```

Atomics are signed 32-bit (`atomic_t`) or signed 64-bit (`atomic64_t`). They're guaranteed atomic against all contexts, all CPUs. They don't take a lock. They use the CPU's atomic-instruction support (ldrex/strex on ARM).

Use atomics for:

- Reference counts.
- Statistics counters (e.g., packets transmitted).
- Simple flags (one bit at a time, but `set_bit/clear_bit/test_bit` are a better API for that).

Don't use atomics for:

- Anything that requires multi-step consistency (e.g., "increment `a` and decrement `b`", you need a lock).
- Pointers that need ordering with respect to other writes (use RCU instead).

Bit operations

For single-bit flags:

```
unsigned long flags = 0;

set_bit(BIT_RUNNING, &flags);
clear_bit(BIT_RUNNING, &flags);
if (test_bit(BIT_RUNNING, &flags)) ...

/* Atomic test-and-set; returns the OLD value */
if (!test_and_set_bit(BIT_BUSY, &flags))
    /* we got the lock */
```

These compile to the same atomic instructions as `atomic_t`. They're idiomatic for "a small set of independent state bits."

41.4 Spinlocks

A spinlock is the simplest lock: a CPU that can't acquire it *spins* (busy-loops) until it can. No sleeping, no scheduling, perfect for protecting state shared with IRQ handlers.

```
#include <linux/spinlock.h>

static DEFINE_SPINLOCK(my_lock);    /* static initializer */

spin_lock(&my_lock);
shared_data = new_value;
spin_unlock(&my_lock);
```

Three rules for spinlocks:

1. **Hold time must be short.** While you hold a spinlock, the kernel will not preempt the current task. In the IRQ-safe variant, IRQs are disabled on the holding CPU too. Other CPUs trying to grab the lock burn cycles. "Short" means microseconds, not milliseconds.
2. **No sleeping while held.** Don't call `kmalloc(GFP_KERNEL)`, `copy_to_user`, `mutex_lock`, or anything that might sleep. Kernel debug builds (`CONFIG_DEBUG_ATOMIC_SLEEP=y`) will catch and shame you for these.
3. **Don't reschedule.** Don't call `schedule()`, don't call `cond_resched()`, don't call user-space syscalls.

Variants

```
spin_lock(&lock);          spin_unlock(&lock);          /* base */
spin_lock_bh(&lock);       spin_unlock_bh(&lock);       /* disable softirqs */
```

```
spin_lock_irq(&lock);      spin_unlock_irq(&lock);      /* disable IRQs */
spin_lock_irqsave(&lock, flags);  spin_unlock_irqrestore(&lock, flags);
```

When to use which:

- **Process context only**: `spin_lock` is enough.
- **Process + softirq/tasklet**: `spin_lock_bh`. Otherwise a softirq can fire between your lock and unlock and try to grab the same lock, deadlock.
- **Process + IRQ**: `spin_lock_irqsave`. The `_save` variant preserves the previous IRQ state, which matters because you might already be in an IRQ-disabled context (nested locks).

The IRQ variants must be matched: `spin_lock_irqsave` ↔ `spin_unlock_irqrestore`, never mix.

Example: protecting a shared counter

```
static DEFINE_SPINLOCK(stats_lock);
static u64 packets;

/* From process context (driver write callback) */
void user_send(void)
{
    unsigned long flags;
    spin_lock_irqsave(&stats_lock, flags);
    packets++;
    spin_unlock_irqrestore(&stats_lock, flags);
}

/* From IRQ handler */
irqreturn_t my_irq(int irq, void *dev)
{
    spin_lock(&stats_lock);   /* IRQs already off in IRQ handler */
    packets++;
    spin_unlock(&stats_lock);
    return IRQ_HANDLED;
}
```

(In practice, `atomic64_t` would be simpler for a single counter. Spinlocks shine when there's more than one variable to keep consistent.)

41.5 Mutexes

A mutex is a **sleeping** lock. If the lock is unavailable, the requesting task is put to sleep and woken when the lock is released. This makes them perfect for protecting longer critical sections, anything that calls `kmalloc(GFP_KERNEL)`, `copy_to_user`, etc.

```
#include <linux/mutex.h>
```

```
static DEFINE_MUTEX(my_mutex);

mutex_lock(&my_mutex);
/* may sleep */
do_long_work();
mutex_unlock(&my_mutex);
```

Variants:

```
mutex_lock(&m);              /* uninterruptible — can't be killed */
mutex_lock_interruptible(&m); /* returns -EINTR on signal; preferred */
mutex_lock_killable(&m);     /* only fatal signals interrupt */
mutex_trylock(&m);           /* nonblocking; returns 1 if got it, 0 if not */
```

Use `mutex_lock_interruptible` in driver fops so a stuck process can be killed with `Ctrl-C` instead of being unkillable.

Mutexes have one nice property over semaphores: the kernel tracks *who* holds them. Lockdep (Linux's lock-debug subsystem, `CONFIG_PROVE_LOCKING=y`) uses this for deadlock detection, if you ever take lock A while holding B in one place and B while holding A elsewhere, lockdep prints a screaming warning to dmesg with both stack traces.

41.6 Read-write locks

When a resource is **read often, written rarely**, a read-write lock lets multiple readers share access while writers get exclusive ownership.

```
#include <linux/rwlock.h>

static DEFINE_RWLOCK(my_rwlock);

/* Readers */
read_lock(&my_rwlock);
/* read shared data */
read_unlock(&my_rwlock);

/* Writers */
write_lock(&my_rwlock);
/* modify shared data */
write_unlock(&my_rwlock);
```

Same _irq, _bh, _irqsave variants as spinlocks apply. Also exists as sleeping variant: `rwsem` (`down_read`/`down_write` API).

The catch: rwlocks have a famously bad performance profile on heavily contended scenarios, readers are cheap, writers are expensive, and starvation is possible. For most "many-read, rare-write" cases, **RCU is the better choice**.

41.7 RCU, the read-mostly secret weapon

Read-Copy-Update is the kernel's read-mostly trick: readers pay zero synchronization cost, no atomic operations, no lock acquisition, no memory barriers in the fast path. Writers copy, modify, and publish atomically. Old readers see the old version until they finish. New readers see the new.

```
#include <linux/rcupdate.h>

struct config {
    int speed;
    int mode;
};

static struct config __rcu *cur_config;

/* Reader */
void some_read_path(void)
{
    struct config *c;
    rcu_read_lock();
    c = rcu_dereference(cur_config);
    if (c)
        do_something(c->speed, c->mode);
    rcu_read_unlock();
}

/* Writer */
void update_config(int s, int m)
{
    struct config *old, *new;
    new = kmalloc(sizeof(*new), GFP_KERNEL);
    new->speed = s;
    new->mode = m;
    old = rcu_dereference_protected(cur_config, lockdep_is_held(&write_lock));
    rcu_assign_pointer(cur_config, new);
    synchronize_rcu();      /* wait for all readers to finish with `old` */
    kfree(old);
}
```

`rcu_read_lock()`/`rcu_read_unlock()` are basically free, they expand to nothing on most kernel configurations (`CONFIG_PREEMPT_NONE`) or to a per-CPU counter increment (`CONFIG_PREEMPT_RCU`). No spinning, no atomics in the read path.

The catch: writes are expensive (`synchronize_rcu()` can take milliseconds) and you can only protect *pointer* updates this way. RCU is heavy machinery, not for a simple counter. But for "lookup-then-use" data on every packet, it is dramatically faster than any lock, that's why almost every networking-fast-path data structure in the kernel is RCU-protected.

We won't go deeper here. If you find yourself wanting RCU, read `Documentation/RCU/whatisRCU.rst` and the references. It has subtleties.

41.8 Per-CPU data, the lock-free shortcut

Sometimes you can sidestep locking entirely by *splitting* the data. If you have a “packets sent” counter, instead of one shared `atomic64_t`, have **per-CPU counters**. Each CPU increments its own (no contention). Read combines them.

```
#include <linux/percpu.h>

static DEFINE_PER_CPU(u64, packets);

/* Increment on this CPU (no atomic needed if preemption disabled) */
void incr_local(void)
{
    preempt_disable();
    this_cpu_inc(packets);
    preempt_enable();
}

/* Sum across all CPUs */
u64 sum_all(void)
{
    u64 total = 0;
    int cpu;
    for_each_possible_cpu(cpu)
        total += per_cpu(packets, cpu);
    return total;
}
```

Per-CPU data works well when reads are rare relative to writes (the opposite of RCU’s case). Linux’s `getrusage` accounting, networking statistics, scheduler runqueue load, all use per-CPU.

On a single-core CPU (the i.MX6ULL), the per-CPU array has one slot and `this_cpu_inc` is just a normal increment. Still useful, because you’ve written code that scales to multi-core too.

41.9 Lockdep, your friend

Build the kernel with `CONFIG_PROVE_LOCKING=y`. (It costs ~20% performance, so disable in production, but enable during development.) Lockdep watches every lock acquisition and:

- Detects A-B-B-A deadlocks across all CPUs and contexts.
- Catches “sleeping while atomic” violations.
- Flags forgotten unlocks.
- Highlights wait-context mismatches (RT, IRQ context, etc.).

When it triggers, you get a wall of dmesg output. Two stack traces, one per lock acquisition path, and a verdict like “deadlock possible.” Read it carefully, it tells you which locks, in which order, from which functions.

Turn it on during development. Disable for production.

41.10 A worked example: thread-safe ring buffer

Let’s solidify with a small example, a ring buffer that a process writes to from `write(2)` and a tasklet reads from. Two contexts (process, softirq), shared state.

```
#include <linux/spinlock.h>

#define RBUF_SIZE 4096

struct rbuf {
    char data[RBUF_SIZE];
    size_t head, tail;
    spinlock_t lock;
};

static struct rbuf rb;

void rb_init(void) { spin_lock_init(&rb.lock); }

/* Producer: from write() */
ssize_t rb_put(const char __user *u, size_t n)
{
    unsigned long flags;
    char buf[RBUF_SIZE];
    if (n > RBUF_SIZE) n = RBUF_SIZE;
    if (copy_from_user(buf, u, n)) return -EFAULT;

    spin_lock_irqsave(&rb.lock, flags);   /* lock for shared structure access */
    for (size_t i = 0; i < n; i++) {
        size_t next = (rb.head + 1) % RBUF_SIZE;
        if (next == rb.tail) break;       /* full */
        rb.data[rb.head] = buf[i];
        rb.head = next;
    }
    spin_unlock_irqrestore(&rb.lock, flags);
    return n;
}

/* Consumer: from a tasklet (softirq context) */
size_t rb_consume(char *out, size_t max)
{
    unsigned long flags;
    size_t i;
    spin_lock_irqsave(&rb.lock, flags);
    for (i = 0; i < max && rb.head != rb.tail; i++) {
        out[i] = rb.data[rb.tail];
        rb.tail = (rb.tail + 1) % RBUF_SIZE;
    }
    spin_unlock_irqrestore(&rb.lock, flags);
```

```
    return i;
}
```

Things to notice:

- We `copy_from_user` *outside* the lock, because `copy_from_user` can sleep. With the lock, that would be illegal.
- The shared-state access (the actual put/consume) is short and under spinlock.
- We use `spin_lock_irqsave` because the consumer runs in softirq context.

This pattern, copy outside, lock around the structure mutation only, is fundamental. Locks should protect the **minimum** of code that needs protection.

41.11 Key Notes

- Concurrency depends on access paths, not only CPU count. Syscalls, interrupts, timers, and deferred work can interleave on a single-core system.
- The protected invariant determines the lock's scope. An atomic counter does not make a buffer, its length, and its ownership consistent as a group.
- Sleeping operations belong outside atomic critical sections. The ring-buffer example separates user copying from mutation of shared queue state for this reason.
- IRQ-safe locking prevents a local interrupt from attempting a lock held by the interrupted path. All paths sharing the state must follow a compatible locking discipline.
- A consistent acquisition order prevents lock-order cycles. Lockdep can expose exercised dependency problems but does not prove every possible execution safe.
- RCU separates publication from reclamation, while per-CPU state reduces shared writes. Both trade simpler contention paths for additional update, lifetime, or aggregation rules.

41.12 Pitfalls

- **Sleeping inside a spinlock.** The kernel may not crash immediately. It may deadlock minutes or hours later when the *other* CPU tries to take the same lock. `CONFIG_DEBUG_ATOMIC_SLEEP=y` catches this immediately at the offending call site.
- **Forgetting `spin_lock_irqsave` when sharing with an IRQ.** Process takes lock → IRQ fires on same CPU → IRQ tries lock → deadlock. Use `_irqsave` whenever IRQ context can hit the same lock.

- **Recursive locking.** Linux mutexes are **not recursive**. Taking the same mutex twice from the same task = deadlock (or BUG with debug enabled). Use `mutex_trylock` if you might already hold it, or restructure to never re-enter.
- **`mutex_lock` in `read`/`write` instead of `mutex_lock_interruptible`.** A process holding your mutex becomes unkillable. Always use interruptible in syscall context.
- **Long-held mutexes.** `mutex_lock` doesn't sleep-spin, it actually puts the thread to sleep. For very short critical sections (<a few hundred cycles), a spinlock is faster than the schedule overhead. For long ones, mutex is far better. Profile if unsure. The dividing line is ~tens of cycles vs ~microseconds.
- **Mixing atomic and locked access.** `atomic_inc(&x); spin_lock(&l); some_other_code(); spin_unlock(&l);`, what consistency are you trying to maintain? If `x` is only consistent with the lock, use the lock everywhere. Mixed access usually means a bug.
- **Using `volatile`.** `volatile` is almost always wrong in kernel code. Use atomics or memory barriers. The kernel coding style explicitly bans `volatile` except for compile-time-known special cases. See `Documentation/process/volatile-considered-harmful.rst`.

41.13 Going deeper

- **`Documentation/locking/`**: the kernel's lock-by-lock documentation. Read `mutex-design.rst` and `rt-mutex-design.rst`.
- **`Documentation/locking/lockdep-design.rst`**: what lockdep watches for and how to read its reports.
- **`Documentation/RCU/whatisRCU.rst`**: the friendly RCU intro.
- **`Documentation/memory-barriers.txt`**: when you absolutely need to think about memory ordering. (Almost never in driver code. Usually only in lock primitives themselves.)
- **`Is Parallel Programming Hard, And, If So, What Can You Do About It?`** by Paul McKenney, the definitive textbook on kernel concurrency, free PDF online.

> Next chapter: **Chapter 42: Sleeping, waiting, polling.** With locking understood, we add the missing piece for blocking I/O: wait queues, `wait_event_interruptible`, and the `poll`/`select` integration.

Chapter 42: Sleeping, waiting, polling

> **What: wait queues** (`wait_queue_head_t`, `wait_event_interruptible`, `wake_up`), the `.poll` file_operations callback, and the `O_NONBLOCK` machinery. Together these let a `read(2)` or `write(2)` syscall block until data is ready, wake exactly the right process when it is, and integrate with `select(2)` / `poll(2)` / `epoll`.
>
> **Why:** drivers that produce data on their own schedule (UART, keyboard, sensor, network) need a way to make a reader wait without polling. Without wait queues, your `read` callback either returns "no data, try again" (caller burns CPU spinning) or blocks the CPU itself (kernel hangs). Wait queues are how Linux makes blocking I/O efficient. The thread sleeps. The scheduler runs something else. An interrupt or timer wakes the thread when its data is ready.
>
> **Focus: the sleep/wake protocol**. The reader registers itself on a wait queue, checks a condition, and sleeps if not met. The producer modifies state then calls `wake_up`. The kernel guarantees no missed wakeups via a careful prepare-and-check sequence. Get the sequence right and your driver's blocking I/O is correct. Get it wrong and reads sometimes hang forever.

42.1 The two ways to wait

Two patterns exist in user-space:

1. **Blocking read.** `read(fd, buf, n)` doesn't return until data is available. The kernel puts the caller to sleep and wakes it when data arrives.
2. **Polling / `select`.** The caller registers interest in `fd` becoming readable/writable, then sleeps in `select(fds, ...)` until *any* registered fd is ready. The same `fd` may be one of many.

Both eventually rest on the same kernel primitive: a **wait queue**. The driver maintains a `wait_queue_head_t` per logical "thing to wait for." Threads add themselves and sleep. Producers wake everyone who's waiting.

42.2 Wait queue API

```
#include <linux/wait.h>
```

```
wait_queue_head_t my_wq;
init_waitqueue_head(&my_wq);
/* or: */
DECLARE_WAIT_QUEUE_HEAD(my_wq);

/* Wait until a condition is true */
```

```
wait_event_interruptible(my_wq, condition);

/* Wake everyone waiting on this queue */
wake_up_interruptible(&my_wq);
```

`condition` is **evaluated multiple times** inside `wait_event_interruptible`. The macro:

1. Adds the current task to the wait queue.
2. Sets state to `TASK_INTERRUPTIBLE`.
3. Checks `condition`. If true, removes the task and returns.
4. If false, calls `schedule()`, the task sleeps.
5. On wake, jumps back to step 3.

This loop is what prevents the “lost wakeup” race: the producer might `wake_up` between our check of `condition` and our `schedule()`. The wait_event_* macros handle the atomicity carefully.

Variants

Macro	What
`wait_event(wq, cond)`	Uninterruptible, task can’t be killed
`wait_event_interruptible(wq, cond)`	Killable; returns `-ERESTARTSYS` on signal
`wait_event_timeout(wq, cond, t)`	Times out after `t` jiffies; returns remaining time
`wait_event_interruptible_timeout(...)`	Both interruptible and timeout
`wait_event_killable(wq, cond)`	Like interruptible, but only fatal signals interrupt

For driver `read`/`write` callbacks: **use `wait_event_interruptible` or `wait_event_interruptible_timeout`**. Never use the uninterruptible variants. A stuck driver with uninterruptible waiters is the classic D-state hang, the user cannot kill the process. Only a reboot fixes it.

Wake variants

Function	What
`wake_up(&wq)`	Wakes one or more, sets `TASK_RUNNING`
`wake_up_interruptible(&wq)`	Wakes one or more that are in `TASK_INTERRUPTIBLE`
`wake_up_all(&wq)`	Wakes everyone
`wake_up_interruptible_all(&wq)`	Wakes everyone in `TASK_INTERRUPTIBLE`

For driver code, match the variant to your wait: if you `wait_event_interruptible`, you `wake_up_interruptible`. Mismatch isn't broken per se, wake_up wakes everyone, but the pairing is conventional and clear.

42.3 A blocking read example

A kernel-thread "producer" illustrates how the Chapter 40 misc-device interface can support a blocking `read` that waits for data.

```
#include <linux/wait.h>
#include <linux/kthread.h>
#include <linux/delay.h>

static char data_buf[64];
static int  data_len;
static DECLARE_WAIT_QUEUE_HEAD(read_wq);
static struct mutex data_lock;
static struct task_struct *producer;

/* Producer kernel thread: every 2 seconds, generate data and wake readers */
static int producer_fn(void *arg)
{
    int counter = 0;
    while (!kthread_should_stop()) {
        msleep(2000);
        mutex_lock(&data_lock);
        data_len = snprintf(data_buf, sizeof(data_buf),
                            "event %d\n", counter++);
        mutex_unlock(&data_lock);
        wake_up_interruptible(&read_wq);
    }
    return 0;
}

static ssize_t my_read(struct file *filp, char __user *ubuf,
```

```
                        size_t count, loff_t *ppos)
{
    ssize_t ret;

    /* Block until data_len > 0; respect O_NONBLOCK */
    if (filp->f_flags & O_NONBLOCK) {
        if (data_len == 0)
            return -EAGAIN;
    } else {
        if (wait_event_interruptible(read_wq, data_len > 0))
            return -ERESTARTSYS;
    }

    mutex_lock(&data_lock);
    if (count > data_len)
        count = data_len;
    if (copy_to_user(ubuf, data_buf, count)) {
        ret = -EFAULT;
        goto out;
    }
    data_len = 0;   /* consumed */
    ret = count;
out:
    mutex_unlock(&data_lock);
    return ret;
}
```

Test:

```
[root@pa-mini:~]# insmod waiting.ko
[root@pa-mini:~]# cat /dev/waiting
event 0       ← appears after ~2 seconds, blocks here
event 1
event 2
...
```

The read blocks. The producer thread wakes the wait queue every 2 seconds. The reader wakes, returns one event's worth of data, then re-enters the wait if `cat` continues.

O_NONBLOCK

```
if (filp->f_flags & O_NONBLOCK) {
    if (data_len == 0)
        return -EAGAIN;
}
```

If the user opened the device with `O_NONBLOCK`, we *never* sleep, we return `-EAGAIN` (= "would block. Try again later") immediately if no data. This is what `epoll` and similar event loops want.

`O_NONBLOCK` is a per-open flag. The user can change it later via `fcntl(fd, F_SETFL, O_NONBLOCK)`. Always check it.

42.4 poll / select / epoll

`select(2)` and friends let user-space block on multiple fds at once. To support them, your driver implements a `.poll` callback in `file_operations`:

```
static __poll_t my_poll(struct file *filp, poll_table *wait)
{
    __poll_t mask = 0;

    /* Register our wait queue with the poll system */
    poll_wait(filp, &read_wq, wait);

    /* Report current readability/writability */
    if (data_len > 0)
        mask |= EPOLLIN | EPOLLRDNORM;

    /* If we had writable state, we'd also: */
    /* if (have_space) mask |= EPOLLOUT | EPOLLWRNORM; */

    return mask;
}
```

Add to `file_operations`:

```
static const struct file_operations my_fops = {
    .owner   = THIS_MODULE,
    .read    = my_read,
    .poll    = my_poll,
    /* ... */
};
```

The `poll_wait` call **doesn’t block**. It just registers our wait queue with the kernel’s poll machinery, if the caller’s `select` decides to sleep, it’ll be woken whenever any of the queues it registered with is woken.

After registering, we **immediately** report current readability. If data is available, `EPOLLIN | EPOLLRDNORM` says “this fd is readable now.”

The flow from user-space:

```
   user: select({read=fd}, ..., timeout=10s)
            |
            ▼
   kernel: dispatches to my_poll(filp, &poll_table)
            |
            ▼
   my_poll: poll_wait(filp, &read_wq, &poll_table)
            return mask
            |
            ▼
   kernel: if mask != 0, return immediately
            else, sleep on the registered wait queues
            |
```

```
          ▼
   producer: wake_up_interruptible(&read_wq)
          │
          ▼
   kernel: woken; rerun all the poll callbacks
          if any returns non-zero mask, return to user-space
```

The `wake_up_interruptible(&read_wq)` does double duty: it wakes blocking `read()` callers *and* triggers a recheck for `select` callers. One wait queue, two consumers, exactly what we want.

Testing with `select`

User-space test (`test_poll.c`):

```
#include <sys/select.h>
#include <stdio.h>
#include <fcntl.h>
#include <unistd.h>

int main(void)
{
    int fd = open("/dev/waiting", O_RDONLY | O_NONBLOCK);
    char buf[64];
    fd_set rfds;
    struct timeval tv;

    while (1) {
        FD_ZERO(&rfds);
        FD_SET(fd, &rfds);
        tv.tv_sec = 5;
        tv.tv_usec = 0;
        int r = select(fd + 1, &rfds, NULL, NULL, &tv);
        if (r == 0) {
            printf("[timeout]\n");
        } else if (r > 0 && FD_ISSET(fd, &rfds)) {
            int n = read(fd, buf, sizeof(buf) - 1);
            buf[n] = 0;
            printf("[%d bytes] %s", n, buf);
        }
    }
}
```

Build for ARM and run on target:

```
[root@pa-mini:~]# ./test_poll
[64 bytes] event 0
[64 bytes] event 1
[timeout]      ← if no events for 5 seconds
...
```

`select` correctly sleeps and wakes only when our driver signals data ready.

42.5 msleep, msleep_interruptible, schedule_timeout

Driver code sometimes needs to “wait N milliseconds.” The options:

```
msleep(100);                  /* sleep 100 ms; uninterruptible */
msleep_interruptible(100);    /* sleep 100 ms; signal can wake */
ssleep(2);                    /* sleep 2 seconds */
udelay(50);                   /* busy-wait 50 µs (cannot sleep) */
mdelay(2);                    /* busy-wait 2 ms (cannot sleep) */
usleep_range(50, 100);        /* sleep ~50–100 µs, can sleep */
schedule_timeout(jiffies);    /* low-level; you set the task state first */
```

Quick guide:

- **Process context, can sleep, exact delay not critical:** `msleep`.
- **Process context, can sleep, want to be killable:** `msleep_interruptible`.
- **Process context, sleeping but short:** `usleep_range`. Kernel may bundle short sleeps to reduce wake-ups.
- **Atomic context (IRQ handler, spinlock held):** `udelay` or `mdelay` only, these busy-wait. Do not `mdelay` more than ~10 ms, you stall every other task on a single-core system.

> **MCU bridge:** Think of an IRQ like an EXTI/NVIC interrupt path, except Linux splits the hard interrupt from deferred work and must share lines across drivers. **IRQ:** interrupt request, the signal path that tells the CPU or interrupt controller that hardware needs service.

- **Need to wait for a condition with a timeout:** `wait_event_interruptible_timeout`.

42.6 Tasks state machine

Quick sidebar on what “sleep” means.

A task in Linux has a state:

- `TASK_RUNNING`: on a CPU or in a runqueue waiting for one.
- `TASK_INTERRUPTIBLE`: sleeping. Can be woken by a signal.
- `TASK_UNINTERRUPTIBLE`: sleeping. Only the thing it’s waiting for can wake it. (This is the dreaded “D state” you see in `ps`.)
- `TASK_KILLABLE`: uninterruptible but fatal signals (SIGKILL) wake it. Compromise between INT and UNINT.
- `TASK_STOPPED`: paused by SIGSTOP.

- `TASK_TRACED`: paused by ptrace.

`wait_event_interruptible` sets state to `TASK_INTERRUPTIBLE`. `wait_event` (no _interruptible) sets `TASK_UNINTERRUPTIBLE`. **Avoid `TASK_UNINTERRUPTIBLE`** for driver-level waits, if your driver bugs out, the user can't kill the stuck process. The system feels unresponsive, `kill -9` doesn't help, only reboot does.

Times where `TASK_UNINTERRUPTIBLE` is appropriate:

- Waiting for filesystem I/O.
- Waiting for a hardware operation that *must* complete (DMA finish, etc.).

> **MCU bridge:** Think of DMA like the MCU DMA controller you used for UART or SPI, but with cache coherency, scatter-gather descriptors, and kernel ownership rules added. **DMA:** Direct Memory Access. Hardware moves data to or from memory without the CPU copying each byte.

- Holding important locks that signals could destabilize.

For chardev `read`/`write`, always use interruptible.

42.7 Key Notes

- A wait queue records interest in a condition. The producer owns state changes and wakeups, while the reader remains responsible for checking whether its condition is satisfied.
- `wait_event_*` handles the sleep/wake registration protocol. Shared data still needs synchronization, and a wakeup does not grant exclusive ownership to one reader.
- Blocking reads, nonblocking reads, and `.poll` must describe the same readiness state. `-EAGAIN` means no transfer is currently possible, not EOF.
- `.poll` registers interest before reporting current readiness. Readiness is an observation, not a reservation of data against another consumer.
- Signal interruption and timeouts bound waits that would otherwise trap an application behind a failed producer. The syscall's restart policy is part of its observable behavior.
- Removal needs a terminal condition and a wakeup path for blocked callers. Stopping the producer alone can leave readers asleep on state that will never change.

42.8 Pitfalls

- **Race: condition check vs schedule.** If you write the pattern by hand instead of using `wait_event_interruptible`, you may have a window where the producer sets the condition, calls wake, and you fall asleep *after* the wake, sleeping forever. **Always use the `wait_event_*` macros.** They handle this race correctly.
- **Forgetting to wake.** Producer changes state but never calls `wake_up`. Reader sleeps forever. Symptom: works once (initial check passes), hangs after.
- **`wait_event` instead of `wait_event_interruptible` in `read`/`write` callbacks.** Process becomes unkillable when stuck. Always use interruptible variants in fops.
- **Returning `-EINTR` instead of `-ERESTARTSYS`.** Both are valid responses to a signal during a sleep, but `-ERESTARTSYS` causes the kernel to re-execute the syscall after the signal handler returns (if the signal handler permits). `-EINTR` returns the error directly to user-space, requiring the app to retry. Prefer `-ERESTARTSYS`. It's friendlier.
- **`poll_wait` called after returning the mask.** Order matters: register the wait first, then return the mask. Reverse it and the kernel may register no wait, so `select` busy-loops.
- **Calling `msleep` in an IRQ handler.** IRQ handlers are atomic. Use `udelay` or `mdelay` (busy-wait, no sleep), or schedule a workqueue/tasklet to do the sleeping work.
- **Memory-barrier wishful thinking.** Producer writes data buffer, then wakes. Reader is woken, then reads buffer. The `wake_up` family has implicit barriers, wake_up implies a full barrier, and the woken task's resumption implies a barrier too. You usually don't need explicit `smp_wmb()`/`smp_rmb()`. But if you're doing fancy lockless work, double-check `Documentation/memory-barriers.txt`.

42.9 Going deeper

- **`Documentation/scheduler/sched-domains.rst`**: how the scheduler picks which CPU to wake a task on.
- **`Documentation/filesystems/poll.rst`**: the kernel's poll subsystem and how user-space `epoll` is implemented in terms of it.
- **`drivers/tty/`**: UART and TTY drivers. Real-world examples of mixed blocking-read + poll + ioctl driver code.
- **`drivers/input/evdev.c`**: the input-event chardev. Read it as a reference for poll-based event delivery.
- **LDD3 Chapter 6**: blocking I/O, still mostly accurate after all these years.

> Next chapter: **Chapter 43: Interrupts.** With locking and waiting in place, we wire actual hardware events into the driver: `request_irq`, top-half/bottom-half split, tasklets, workqueues, threaded IRQs.

Chapter 43: Interrupts

> **What:** `request_irq`, the **top-half / bottom-half split**, and the four standard bottom halves, softirqs, tasklets, work queues, and threaded IRQs. The examples distinguish interrupt acknowledgement from deferred processing and show how the execution context constrains each callback. **IRQ:** interrupt request, the signal path that tells the CPU or interrupt controller that hardware needs service.
>
> **Why:** interrupts are how hardware tells the kernel something happened: data arrived, DMA finished, a button was pressed, a timer expired. Get the IRQ-handler design wrong and you hit one of two failures: *missed interrupts* (handler too slow or wrong polarity) or *IRQ storms* (handler does not acknowledge, hardware re-asserts continuously, system hangs). The rules below give you the right design every time. **MCU bridge:** Think of DMA like the MCU DMA controller you used for UART or SPI, but with cache coherency, scatter-gather descriptors, and kernel ownership rules added. **DMA:** Direct Memory Access. Hardware moves data to or from memory without the CPU copying each byte.
>
> **Focus: the IRQ contract is "fast, atomic, and minimal."** Your top-half runs with interrupts disabled, in atomic context (no sleeping, no `kmalloc(GFP_KERNEL)`, no `copy_to_user`). Anything that takes more than a few microseconds *must* be deferred to a bottom half. Once you accept this constraint, the API choices below follow naturally.

43.1 How the i.MX6ULL gets an interrupt to your code

The chain, end to end:

```
hardware peripheral (e.g., GPIO pin transition)
     │
     ▼
peripheral IRQ output line
     │
     ▼
GIC (Generic Interrupt Controller — Cortex-A IRQ multiplexer)
     │ assigns IRQ number (e.g., 99)
     ▼
CPU's IRQ exception vector
     │
     ▼
kernel's gic_handle_irq → generic_handle_irq(virq)
     │
     ▼
IRQ domain mapping (DT-based) → your handler
     │
```

```
        ▼
   your_irq_handler(irq, dev_id) — runs with IRQ off
```

Two things to notice:

1. **IRQ numbers in DT and `request_irq` are virtual.** The DT line `interrupts = <0 99 IRQ_TYPE_LEVEL_HIGH>` carries the GIC hardware number. At boot, the kernel maps it to a virtual IRQ (a *virq*). Your `request_irq` uses this virq. You usually do not see the mapping happen, the framework hands you the virq.
2. **GIC is the multiplexer.** The CPU has one IRQ line. The GIC has up to ~160 inputs (i.MX6ULL specific) and figures out which is firing. The kernel's GIC driver demultiplexes and routes to your handler.

43.2 The top half

Your top half is the function the kernel calls when the IRQ fires. The contract:

- **It runs in atomic context.** No sleeping. No `kmalloc(GFP_KERNEL)`. No `mutex_lock`. No `copy_to_user`. No `printk` with `KERN_INFO` (well, `printk` works but is rate-limited. Minimize).
- **It runs with that IRQ disabled.** The GIC won't re-fire the same IRQ on the same CPU until you return. (Other CPUs *can* see it. That's how SMP works.)
- **It runs with kernel preemption off.** No context switch until you return.
- **It returns `IRQ_HANDLED`** if it processed the IRQ, `IRQ_NONE` if not (used in shared-IRQ scenarios, "this wasn't mine").
- **It must acknowledge the hardware.** Otherwise the IRQ line stays asserted and the IRQ fires again immediately ⇒ IRQ storm ⇒ system hang.

A canonical top-half:

```
static irqreturn_t my_irq_handler(int irq, void *dev_id)
{
    struct my_dev *dev = dev_id;
    u32 status;

    /* 1. Read status; figure out what fired */
    status = readl(dev->base + STATUS_REG);
    if (!(status & MY_IRQ_FLAG))
        return IRQ_NONE;          /* not for us — shared IRQ */

    /* 2. Acknowledge the hardware (write-1-clear pattern) */
    writel(MY_IRQ_FLAG, dev->base + STATUS_REG);

    /* 3. Capture minimal data; defer the rest */
    dev->raw_value = readl(dev->base + DATA_REG);
    dev->irq_count++;
```

```
    /* 4. Wake the bottom half (workqueue, tasklet, or wait queue) */
    queue_work(dev->wq, &dev->work);

    return IRQ_HANDLED;
}
```

Five lines of real work. Read status, ack, snapshot, defer, return. Under 1 µs on i.MX6ULL.

If your top-half is doing anything more than this, parsing protocol bytes, looking up tables, doing math, it's too long. Move it to a bottom half.

43.3 Requesting an IRQ

```
#include <linux/interrupt.h>
```

```
int request_irq(unsigned int irq, irq_handler_t handler,
                unsigned long flags, const char *name, void *dev);

void free_irq(unsigned int irq, void *dev);
```

For platform drivers, the `irq` number comes from the DT via `platform_get_irq()`:

```
int virq = platform_get_irq(pdev, 0);
if (virq < 0)
    return virq;

err = devm_request_irq(&pdev->dev, virq, my_irq_handler,
                       IRQF_TRIGGER_RISING | IRQF_ONESHOT,
                       "myhw", priv);
```

The `devm_request_irq` variant auto-frees on driver-unbind. Always prefer it.

Flags

Flag	Meaning
`IRQF_TRIGGER_RISING`	Edge-triggered, rising edge
`IRQF_TRIGGER_FALLING`	Edge-triggered, falling edge
`IRQF_TRIGGER_HIGH`	Level-triggered, active high
`IRQF_TRIGGER_LOW`	Level-triggered, active low

Flag	Meaning
`IRQF_SHARED`	Multiple handlers may share this IRQ line
`IRQF_ONESHOT`	Don’t re-enable IRQ until threaded handler completes
`IRQF_NO_THREAD`	Force top-half-only (don’t run as kernel thread)

The trigger flag is **usually omitted** for platform drivers because the DT specifies it (in the `interrupts` property’s third cell). The kernel’s IRQ subsystem reads from DT.

43.4 Bottom halves, four choices

Top half done. Now you need to do the real work outside the atomic constraints. Four options, in order of “easiest” to “most flexible”:

1. Threaded IRQ, the modern default

`request_threaded_irq` is the cleanest pattern: the kernel calls your *primary* (top-half) function with IRQs off, then schedules your *threaded* function as a kernel thread that runs with normal kernel context, can sleep, take mutexes, do `copy_to_user`, everything.

```
static irqreturn_t my_primary(int irq, void *dev_id)
{
    struct my_dev *dev = dev_id;
    /* Read status, ack hw — atomic context */
    writel(...);
    return IRQ_WAKE_THREAD;
}

static irqreturn_t my_threaded(int irq, void *dev_id)
{
    struct my_dev *dev = dev_id;
    /* Process at leisure — full kernel context */
    mutex_lock(&dev->lock);
    /* ... */
    mutex_unlock(&dev->lock);
    wake_up(&dev->wq);
    return IRQ_HANDLED;
}

/* In probe: */
err = devm_request_threaded_irq(&pdev->dev, virq,
                                my_primary, my_threaded,
                                IRQF_TRIGGER_RISING | IRQF_ONESHOT,
                                "myhw", priv);
```

The `IRQF_ONESHOT` flag is important: it keeps the IRQ masked from when the primary returns `IRQ_WAKE_THREAD` until the threaded handler completes. Without it, the IRQ could re-fire and re-schedule before you've finished processing.

You can pass `NULL` for the primary, in which case the kernel installs a default that just returns `IRQ_WAKE_THREAD`. Then your threaded handler is the only thing you wrote.

Use threaded IRQ for ~80% of new driver code. It's the cleanest model.

2. Work queues, explicit deferral

A work queue is a kernel thread that processes a queue of work items. Schedule a work item from your top-half. The work runs later in a normal kernel thread.

```
#include <linux/workqueue.h>

static struct workqueue_struct *my_wq;
static struct work_struct my_work;

static void my_work_fn(struct work_struct *w)
{
    /* Runs in process context. Can sleep. */
    struct my_dev *dev = container_of(w, struct my_dev, work);
    /* Process accumulated data */
}

static irqreturn_t my_irq(int irq, void *dev_id)
{
    /* Top half */
    queue_work(my_wq, &my_work);
    return IRQ_HANDLED;
}

/* In probe */
my_wq = alloc_workqueue("myhw", WQ_UNBOUND, 0);
INIT_WORK(&my_work, my_work_fn);
```

For most drivers, the shared system workqueue is fine and you don't need to allocate your own:

```
INIT_WORK(&my_work, my_work_fn);
/* In IRQ: */
schedule_work(&my_work);
```

When to use work queues over threaded IRQs:

- You want to *coalesce* multiple IRQs into one bottom-half execution (work is idempotent. Queueing it again is a no-op if already queued).
- You don't want a dedicated kthread per IRQ.
- You're piggy-backing on existing workqueue infrastructure.

3. Tasklets, legacy

A tasklet runs in **softirq context** (atomic, can't sleep, but with all IRQs enabled). It's faster to schedule than a workqueue but lives in atomic context.

```
#include <linux/interrupt.h>

/* Modern form (since v5.9): DECLARE_TASKLET(name, fn) where fn takes (struct
tasklet_struct *).
 * DECLARE_TASKLET_OLD is the backward-compat macro for the legacy (unsigned long)
callback shown
 * here. New drivers should prefer DECLARE_TASKLET — but better still, prefer workqueues
 * (tasklets are being phased out across the tree). */
static DECLARE_TASKLET_OLD(my_tasklet, my_tasklet_fn);

static void my_tasklet_fn(unsigned long data)
{
    /* Softirq context. Can't sleep. */
}

static irqreturn_t my_irq(int irq, void *dev_id)
{
    tasklet_schedule(&my_tasklet);
    return IRQ_HANDLED;
}
```

Tasklets are **discouraged** in new code. The kernel is migrating away from them, they're an obstacle for `PREEMPT_RT` (real-time kernel. Ch 52A). Use threaded IRQs unless you have a strong reason for atomic-context bottom-half processing.

> **PREEMPT_RT:** the Linux real-time patch set that makes more kernel paths preemptible and reduces latency.

4. Softirqs, kernel-internal only

Softirqs are the lowest-level deferred mechanism. Used internally for networking, timers, and block-I/O completion. **Driver authors don't write softirqs**. We use tasklets/ workqueues/threaded IRQs which are built on top of softirq machinery.

Picking among them, table

Bottom half	Context	Can sleep?	Coalesces?	Scheduling cost	When to use
Threaded IRQ	Process (kthread)	Yes	No	~10 µs (wake kthread)	Default modern choice

Bottom half	Context	Can sleep?	Coalesces?	Scheduling cost	When to use
Workqueue (shared)	Process (kthread)	Yes	Yes (if queued)	~10 µs	Coalescing multi-IRQ
Workqueue (dedicated)	Process (kthread)	Yes	Yes	~10 µs	Want isolation
Tasklet	Softirq	No	Yes	< 1 µs	Legacy; latency-sensitive

43.5 GPIO interrupts, the everyday case

The most common reason embedded drivers want IRQs is to react to a GPIO transition (button press, sensor data-ready, etc.). The mechanics:

> **MCU bridge:** Think of Linux GPIO like the same pin set/reset block you used on STM32, but accessed through a kernel subsystem that owns numbering, direction, interrupts, and user-space exposure. **GPIO:** General-Purpose Input/Output, a pin controlled as a digital input, output, or interrupt source.

DT side

```
my_button {
    compatible = "linuxlearn,button";
    interrupt-parent = <&gpio4>;
    interrupts = <14 IRQ_TYPE_EDGE_FALLING>;
    button-gpios = <&gpio4 14 GPIO_ACTIVE_LOW>;
};
```

The `interrupts` property names the GPIO bank (via `interrupt-parent`) and the pin within that bank. `IRQ_TYPE_EDGE_FALLING` says trigger on the high-to-low transition.

Driver side

```
static int button_probe(struct platform_device *pdev)
{
    struct gpio_desc *gpio;
    int virq, err;

    gpio = devm_gpiod_get(&pdev->dev, "button", GPIOD_IN);
    if (IS_ERR(gpio))
```

```
        return PTR_ERR(gpio);

    virq = gpiod_to_irq(gpio);
    if (virq < 0)
        return virq;

    err = devm_request_threaded_irq(&pdev->dev, virq, NULL, button_thread,
                                    IRQF_TRIGGER_FALLING | IRQF_ONESHOT,
                                    "button", priv);
    return err;
}

static irqreturn_t button_thread(int irq, void *dev_id)
{
    /* Button pressed; do something in process context */
    pr_info("button: pressed\n");
    return IRQ_HANDLED;
}
```

Two new things:

- **`gpiod_to_irq`** converts a GPIO descriptor into a virq we can pass to `request_irq`. This is the bridge.
- **NULL primary handler** in `request_threaded_irq`, when you don't have any atomic-context work to do, pass NULL and the kernel installs a default that just returns `IRQ_WAKE_THREAD`.

That's it. The threaded handler runs whenever the button is pressed. Sleep, mutex, copy_to_user, all fine.

43.6 Shared IRQs

Multiple devices can share one IRQ line on some hardware (PCI is the canonical case. Some SoC peripherals also support it). To handle:

```
err = request_irq(virq, my_handler, IRQF_SHARED, "myhw", priv);
```

Each handler examines hardware status to see if *its* device fired. Returns `IRQ_HANDLED` if so, `IRQ_NONE` if not. The kernel calls all registered handlers in turn until one returns `IRQ_HANDLED` (or all return `IRQ_NONE`, in which case it's a spurious IRQ).

On i.MX6ULL, GPIO interrupts naturally share (all 32 pins of a bank share one GIC line), so the kernel's GPIO IRQ controller multiplexes for you. You don't need `IRQF_SHARED` for those, the GPIO subsystem handles demuxing.

43.7 IRQ-related debug

When IRQs misbehave:

```
[root@pa-mini:~]# cat /proc/interrupts
           CPU0
  17:          0     GIC-0  29 Edge      ...
  19:        842     GIC-0  30 Edge      arch_timer
  21:         12     GIC-0  31 Edge      arch_timer
  29:       4221     GIC-0  68 Level     2020000.serial
  46:          0  gpio-mxc  14 Falling   button
...
```

`/proc/interrupts` shows every active IRQ, total count, controller, hardware number, trigger type, and owner. Look for:

- **Count stuck at 0**: IRQ never fires, wrong trigger polarity, hardware not configured, line not wired up.
- **Count exploding**: IRQ storm, handler not acknowledging hardware properly.
- **“None” in the owner column**: someone requested it then `free_irq`’d but didn’t claim ownership. Usually a bug.

To debug timing:

```
$ echo function > /sys/kernel/debug/tracing/current_tracer
$ echo my_irq_handler my_threaded > /sys/kernel/debug/tracing/set_ftrace_filter
$ echo 1 > /sys/kernel/debug/tracing/tracing_on
... trigger IRQs ...
$ cat /sys/kernel/debug/tracing/trace
```

You’ll see timestamps for every entry/exit of your handler, in microseconds. Chapter 119 covers ftrace properly.

43.8 Key Notes

> **Privilege boundary:** $ means normal user. # or sudo means root and can change host or target state. After a privileged command, verify the expected device, service, or file appears before continuing. Roll back by undoing the config change or stopping the service you just enabled.

- IRQ domains translate controller-specific interrupt descriptions into the virtual IRQ used by a driver. A DT hardware number is not a portable `request_irq` argument.
- The hard handler identifies and services the source with bounded, non-sleeping work. Bus transactions that can block belong in a threaded handler or other sleeping context.

- Threaded IRQs keep deferred service associated with the interrupt, while workqueues suit independently scheduled or coalesced work. Coalescing requires a plan for retaining every event that matters.
- Trigger polarity and device acknowledgement are separate contracts. A level source left asserted can storm, so intentional misconfiguration is not a safe recovery demonstration.
- Shared handlers claim only events indicated by their own hardware status. The `dev_id` cookie also connects registration, instance state, and release.
- Teardown must prevent new events and drain handlers and deferred work before freeing state. Managed IRQ registration does not automatically cancel separately queued work.

43.9 Pitfalls

- **Sleeping in a top-half.** `kmalloc(GFP_KERNEL)`, `mutex_lock`, `copy_to_user`, all forbidden. `CONFIG_DEBUG_ATOMIC_SLEEP=y` catches at the call site. Use `GFP_ATOMIC` if you really must allocate from IRQ context. Otherwise pre-allocate.
- **Forgetting to ack.** Level-triggered IRQ + no acknowledge = continuous re-fire = lockup. The kernel will eventually detect the storm and disable the IRQ, printing “spurious IRQ disabled” to dmesg.
- **Wrong trigger polarity in DT.** Symptom: IRQ never fires. Always cross-check the device’s datasheet against the DT’s `IRQ_TYPE_*` value.
- **Returning `IRQ_NONE` from a non-shared handler.** The kernel treats this as a spurious IRQ. After enough of these, the IRQ is disabled.
- **Calling `request_irq` then `free_irq` with mismatched `dev` pointers.** `free_irq` is keyed on the cookie. Mismatch ⇒ silently fails to free the right handler.
- **Not using `IRQF_ONESHOT` with threaded IRQs.** The hardware can re-fire while the thread is still running, causing a queue of pending threaded calls. Almost always you want `IRQF_ONESHOT`.
- **Wrong context for memory allocation.** In an IRQ handler, `kmalloc(GFP_KERNEL)` may sleep waiting for memory reclaim. Use `GFP_ATOMIC` in top-halves and bottom-halves running in softirq context. Top tip: pre-allocate at probe time so you never alloc in IRQ context.
- **Forgetting `dev_id` parameter.** `request_irq` takes a cookie. You can pass a pointer to your private state. **Don’t pass NULL** even if you don’t need it, the kernel won’t free the handler later, because the cookie is part of the identity for `free_irq`.

- **Not handling shared IRQ correctly.** If you register with `IRQF_SHARED` but always return `IRQ_HANDLED`, other handlers on the same line are starved. Inspect *your* hardware status before claiming the IRQ.

43.10 Going deeper

- **`Documentation/core-api/genericirq.rst`**: the kernel's generic IRQ framework.
- **`Documentation/devicetree/bindings/interrupt-controller/`**: the binding for declaring IRQ controllers in DT.
- **`Documentation/locking/lockdep-design.rst`**: IRQ-context vs process-context locking rules (important once you have shared state).
- **`drivers/gpio/gpio-mxc.c`**: the i.MX GPIO driver. Shows how a chained IRQ controller works: the GPIO bank handler demuxes 32 pin events into per-pin virqs.
- **`Documentation/PCI/MSI-HOWTO.rst`**: PCI Message Signaled Interrupts. Different mechanism, same API on the receiving side.
- **`drivers/spi/spi-imx.c`**: a real i.MX driver using threaded IRQ + DMA.

> **End of foundation chapters (Ch 36–43).** You now have the full kernel-module driver vocabulary: load/unload, chardev, hot-plug, platform binding, locking, blocking I/O, and interrupts. The chapters that follow (44–51 + insertions) take this vocabulary and apply it to specific subsystems: GPIO, input, I²C, SPI, PWM/RTC, IIO, regmap, DMA, network, sound, LCD/DRM. Each chapter follows the same pattern, the subsystem provides a registration API, you fill in callbacks, the framework handles the rest. **MCU bridge:** Think of Linux PWM like an MCU timer output channel, except the driver exposes period, duty cycle, polarity, and enable state through a subsystem. **MCU bridge:** Think of regmap like a typed wrapper around your read_reg() and write_reg() helpers, with caching, locking, and bus differences handled centrally. **PWM:** Pulse-Width Modulation, a timer output whose duty cycle controls average power or encodes timing. **IIO:** Industrial I/O, Linux's subsystem for sensors, ADCs, DACs, and buffered sampled data. **regmap:** a kernel helper that wraps register reads and writes over I2C, SPI, or MMIO.

> Next chapter: **Chapter 44: GPIO subsystem.** The `gpiod_*` API and how character drivers integrate with the GPIO framework.

Chapter 44: GPIO subsystem + pinctrl

> **What:** the two halves of Linux's pin handling, **pinctrl** (which decides what *function* a pin has, GPIO vs UART vs I²C, plus electrical properties: drive strength, pull-up, slew rate) and **gpiod** (the modern descriptor-based API for the pins that *do* end up as GPIOs: direction and value). By the end you can request a GPIO from DT, configure pull-up, drive it, watch it for IRQs, all without ever touching an MMIO register. **GPIO:** General-Purpose Input/Output, a pin controlled as a digital input, output, or interrupt source. **MMIO:** memory-mapped I/O, where software accesses peripheral registers through normal load and store instructions.
>
> **Why:** every real driver eventually wants a GPIO. A reset pin on a peripheral chip. A power-enable on a regulator. A `data-ready` line from a sensor. Hard-coding the MMIO writes (as we did in Part II bare-metal) couples the driver to one specific SoC. The kernel's `gpiod_*` API gives you a portable, DT-described abstraction: "this driver wants the GPIO whose DT property is `reset-gpios`," and the gpiod subsystem figures out which bank, which pin, and which register to touch.
>
> **Focus: the descriptor abstraction**. `struct gpio_desc *` hides the bank, the pin offset, the polarity (`ACTIVE_LOW`), and the SoC-specific register layout behind one opaque handle. Once you accept that, and stop thinking in "GPIO numbers", every GPIO-using driver in Linux looks the same.

44.1 The two-step pin model

Almost every pin on an SoC is *multiplexed*: it can act as one of several functions. The i.MX6ULL's `UART1_RTS_B` ball, for example, can be:

Mux value	Function
0	UART1 RTS (default, the pin's name)
1	ENET1 transmit error
2	USDHC1 card-detect
3	CSI camera data bit 5

Mux value	Function
4	Ethernet 1588 event output
5	**GPIO1 IO19**
8	USDHC2 card-detect

Picking *which* function the pin performs is **pin multiplexing** (pinmux). Setting drive strength, pull-up, slew rate, etc. is **pin configuration** (pinconf). Together they are managed by the **pinctrl subsystem**.

Only *after* you've muxed a pin as GPIO does it become a GPIO. Then a separate API, the **gpiod** subsystem, handles its direction (input/output) and value (high/low).

The two-step model is fixed across Linux:

```
   DT says:        pinctrl-0 = <&pinctrl_my_button>;
                   button-gpios = <&gpio1 19 GPIO_ACTIVE_LOW>;
                                  └──┬──┘ └┬┘ └──────┬──────┘
                                     │     │     polarity
                                     │     pin within bank
                                    bank phandle

   step 1:  pinctrl sets the iomux register so pin GPIO1_IO19 is in GPIO mode
            (mux value 5; conf_reg gets 0x17059: pull-up, drive strength, etc.)

   step 2:  driver calls devm_gpiod_get(dev, "button", GPIOD_IN);
            gpiod subsystem looks up the descriptor, configures direction
```

If you forget step 1, leave the pin in its default UART function, step 2 reads garbage and your driver thinks the button is always pressed. The two are coupled but managed separately. Most boards do step 1 once in DT (via a `pinctrl-0` reference), then drivers ask for descriptors as needed.

44.2 pinctrl in the device tree

The pinctrl entries live under the SoC's `iomuxc` node. For i.MX6ULL:

```
&iomuxc {
    pinctrl-names = "default";
    pinctrl-0 = <&pinctrl_hog>;

    pinctrl_hog: hoggrp {
        fsl,pins = <
            MX6UL_PAD_UART1_RTS_B__GPIO1_IO19  0x17059
            MX6UL_PAD_GPIO1_IO09__GPIO1_IO09   0x17059
```

```
        >;
    };

    pinctrl_button: buttongrp {
        fsl,pins = <
            MX6UL_PAD_UART1_RTS_B__GPIO1_IO19  0x17059
        >;
    };
};
```

Two things going on:

Pin groups. `pinctrl_hog` and `pinctrl_button` are pin **groups**. A group is a named bundle of pin configurations that can be applied as a unit. Drivers reference groups by name.

The hog. A `hog` is special: pinctrl applies it at boot time, not at any driver's probe. Pins owned by the core (e.g., a `card-detect` line that nothing else will claim) go in the hog. Pins owned by a specific peripheral go in that peripheral's group, which the peripheral's driver activates.

The macro. `MX6UL_PAD_UART1_RTS_B__GPIO1_IO19` is defined in `arch/arm/boot/dts/imx6ul-pinfunc.h`. It expands to five integers (`mux_reg, conf_reg, input_reg, mux_mode, input_val`). The trailing `0x17059` is the **electrical configuration** (the conf_reg value): pull-up enabled, fast speed, drive strength = 40 Ω, etc. The format of that 32-bit word is documented in the i.MX6ULL reference manual under `IOMUXC_SW_PAD_CTL_PAD_*`.

You don't compute these by hand for every pin. NXP publishes a tool called **Pins Tool for i.MX** with a GUI for picking mux and electrical settings. Or you read the macros that other in-tree boards have already used (`imx6ul-evk.dts`, `imx6ull-14x14-evk.dts`) and copy the ones that match your hardware.

Multiple states

A peripheral can declare *multiple* pinctrl states, typically `default` (active) and `sleep` (low-power):

```
my_uart {
    pinctrl-names = "default", "sleep";
    pinctrl-0 = <&pinctrl_uart_active>;
    pinctrl-1 = <&pinctrl_uart_sleep>;
};
```

The driver switches states via `pinctrl_pm_select_default_state(dev)` and `pinctrl_pm_select_sleep_state(dev)`. The sleep state typically muxes the pins to GPIO with pull-down enabled, so the peripheral's input pins don't float when the peripheral itself is powered off. We'll return to this in Ch 51B (power management).

44.3 GPIO in the device tree

A driver that wants a GPIO declares it in its DT node:

```
my_device {
    compatible = "linuxlearn,my-device";
    reset-gpios = <&gpio5 11 GPIO_ACTIVE_LOW>;
    button-gpios = <&gpio1 19 GPIO_ACTIVE_LOW>;
    led-gpios = <&gpio4 14 GPIO_ACTIVE_HIGH>;
};
```

Each entry is three cells:

1. **The bank phandle** (`&gpio5`), names which GPIO controller bank the pin lives on. I.MX6ULL has 5 banks (`gpio1`–`gpio5`). Most are 32-pin. **`gpio5` exposes only 12 pins (0–11)** because of package pin-count, and `gpio4` is partial on some packages too. Check the IOMUX table before assuming a pin number is wired out.

> **MCU bridge:** Think of IOMUX like STM32 alternate-function selection, but with separate pad electrical settings and board-level ownership by Device Tree. **IOMUX:** the pin multiplexer that decides which peripheral function appears on each package pin.

1. **The pin number** within the bank (0–31).
2. **Flags**: Usually `GPIO_ACTIVE_HIGH` or `GPIO_ACTIVE_LOW`. The polarity is **part of the abstraction**. Code below operates on logical “asserted” / “deasserted”. Physical level is hidden.

The property name pattern is `<purpose>-gpios`. The kernel strips the `-gpios` suffix to form the *connection ID* drivers use to fetch the descriptor.

44.4 The gpiod descriptor API

```
#include <linux/gpio/consumer.h>
```

Fetching a GPIO:

```
struct gpio_desc *reset;

reset = devm_gpiod_get(&pdev->dev, "reset", GPIOD_OUT_HIGH);
if (IS_ERR(reset))
    return PTR_ERR(reset);
```

`devm_gpiod_get(dev, "reset", GPIOD_OUT_HIGH)` says: “look up the `reset-gpios` property on this device’s DT node, claim the pin, set it as an output, and initialise it to **asserted** (`GPIOD_OUT_HIGH`) or **deasserted** (`GPIOD_OUT_LOW`).” Because the DT declared `GPIO_ACTIVE_LOW`, “asserted” means physical low, the API hides that.

Flag options:

- `GPIOD_ASIS`: don’t set direction. Just acquire the descriptor.
- `GPIOD_IN`: input.
- `GPIOD_OUT_LOW`: output, initial value deasserted.
- `GPIOD_OUT_HIGH`: output, initial value asserted.

The `devm_` prefix gives us automatic release on driver-unbind, same as Ch 39.

Reading and writing:

```
int val = gpiod_get_value(reset);     /* returns 0 or 1, logical */
gpiod_set_value(reset, 1);             /* asserts (physical low if ACTIVE_LOW) */

/* From sleeping context only (these may sleep on I²C-based GPIO expanders): */
val = gpiod_get_value_cansleep(reset);
gpiod_set_value_cansleep(reset, 0);
```

Two variants per operation. The plain version is safe to call from atomic context (IRQ handler, spinlock held). The `_cansleep` version is required when the underlying GPIO chip is on an I²C or SPI bus (where the bus transaction itself may sleep). **Use `_cansleep` in process context**, it works for both bus-backed and direct GPIOs.

> **MCU bridge:** Think of an IRQ like an EXTI/NVIC interrupt path, except Linux splits the hard interrupt from deferred work and must share lines across drivers. **IRQ:** interrupt request, the signal path that tells the CPU or interrupt controller that hardware needs service.

Direction change at runtime:

```
gpiod_direction_input(desc);
gpiod_direction_output(desc, 1);   /* output, value = asserted */
```

Plus the helpers we saw in Ch 43:

```
int irq = gpiod_to_irq(desc);
err = devm_request_threaded_irq(dev, irq, NULL, my_thread,
                                IRQF_TRIGGER_RISING | IRQF_ONESHOT,
                                "my-button", priv);
```

44.5 A complete example: button + LED platform driver

DT:

```
my_blinker {
    compatible = "linuxlearn,blinker";
    button-gpios = <&gpio1 19 GPIO_ACTIVE_LOW>;
    led-gpios = <&gpio4 14 GPIO_ACTIVE_HIGH>;
    status = "okay";
};
```

The pinctrl side (in the iomuxc block):

```
&iomuxc {
    pinctrl_blinker: blinkergrp {
        fsl,pins = <
            MX6UL_PAD_UART1_RTS_B__GPIO1_IO19  0x17059  /* button: pull-up, schmitt */
            MX6UL_PAD_NAND_CE1_B__GPIO4_IO14   0x10b0   /* LED: drive strength 40Ω */
        >;
    };
};
```

...and the consumer references it:

```
my_blinker {
    /* ... */
    pinctrl-names = "default";
    pinctrl-0 = <&pinctrl_blinker>;
};
```

Driver:

```
#include <linux/init.h>
#include <linux/module.h>
#include <linux/platform_device.h>
#include <linux/gpio/consumer.h>
#include <linux/interrupt.h>

struct blinker {
    struct gpio_desc *button;
    struct gpio_desc *led;
    int irq;
};

static irqreturn_t blinker_thread(int irq, void *dev_id)
{
    struct blinker *b = dev_id;
    int val = gpiod_get_value_cansleep(b->led);
    gpiod_set_value_cansleep(b->led, !val);   /* toggle LED */
    return IRQ_HANDLED;
}
```

```
static int blinker_probe(struct platform_device *pdev)
{
    struct blinker *b;
    int err;

    b = devm_kzalloc(&pdev->dev, sizeof(*b), GFP_KERNEL);
    if (!b)
        return -ENOMEM;

    b->led = devm_gpiod_get(&pdev->dev, "led", GPIOD_OUT_LOW);
    if (IS_ERR(b->led))
        return dev_err_probe(&pdev->dev, PTR_ERR(b->led), "no led gpio\n");

    b->button = devm_gpiod_get(&pdev->dev, "button", GPIOD_IN);
    if (IS_ERR(b->button))
        return dev_err_probe(&pdev->dev, PTR_ERR(b->button), "no button gpio\n");

    b->irq = gpiod_to_irq(b->button);
    if (b->irq < 0)
        return dev_err_probe(&pdev->dev, b->irq, "irq lookup failed\n");

    err = devm_request_threaded_irq(&pdev->dev, b->irq, NULL, blinker_thread,
                                    IRQF_TRIGGER_FALLING | IRQF_ONESHOT,
                                    "blinker", b);
    if (err)
        return err;

    platform_set_drvdata(pdev, b);
    dev_info(&pdev->dev, "ready\n");
    return 0;
}

static const struct of_device_id blinker_of_match[] = {
    { .compatible = "linuxlearn,blinker" },
    { }
};
MODULE_DEVICE_TABLE(of, blinker_of_match);

static struct platform_driver blinker_driver = {
    .driver = {
        .name = "linuxlearn-blinker",
        .of_match_table = blinker_of_match,
    },
    .probe = blinker_probe,
};
module_platform_driver(blinker_driver);

MODULE_LICENSE("GPL");
```

Build, load, press the button: the LED toggles. About 90 lines, zero MMIO writes, portable to any SoC with a Linux GPIO driver.

44.6 User-space access, libgpiod

For one-off scripting and prototyping, you can drive GPIOs from user-space *without* a kernel driver. The traditional way (`/sys/class/gpio/`) is **deprecated** since 4.8 and gone in some distros, never write new code for it. The current way is the **character-device API** at `/dev/gpiochipN`, accessed via the `libgpiod` library and its CLI tools.

```
[root@pa-mini:~]# gpiodetect
gpiochip0 [209c000.gpio] (32 lines)
gpiochip1 [20a0000.gpio] (32 lines)
gpiochip2 [20a4000.gpio] (32 lines)
gpiochip3 [20a8000.gpio] (32 lines)
gpiochip4 [20ac000.gpio] (24 lines)

[root@pa-mini:~]# gpioinfo gpiochip0
gpiochip0 - 32 lines:
        line   0:      unnamed       unused   input  active-high
        line   1:      unnamed       unused   input  active-high
        ...
        line  19:      unnamed   "blinker"   input  active-low [used]
        ...
```

Reading and writing:

```
# Read pin 19 of bank 1
[root@pa-mini:~]# gpioget gpiochip0 19
1   ← button not pressed (active-low; physical high = logical 1 if asked as inverted)

# Set GPIO4 pin 14 to 1
[root@pa-mini:~]# gpioset gpiochip3 14=1
[root@pa-mini:~]# gpioset gpiochip3 14=0

# Monitor for events
[root@pa-mini:~]# gpiomon --rising-edge gpiochip0 19
event:  RISING EDGE offset: 19 timestamp: [   1234.567890123]
```

`libgpiod` is the same API your driver code uses, but from user-space. Great for board bring-up: confirm a GPIO works at the hardware level before writing a driver around it.

44.7 GPIO expanders, when GPIOs come over a bus

What if you need more GPIOs than the SoC has? Solution: a **GPIO expander** chip on I²C or SPI. The MCP23017 (I²C, 16 GPIOs), PCA9555 (I²C, 16 GPIOs), and PCF8575 are common choices.

From the kernel's perspective, an I²C GPIO expander registers itself as a `gpio_chip` just like the SoC banks. Once registered, `gpiochip5` (or whatever number) appears in `/sys/class/gpio/` and the descriptor API works identically. Drivers don't know or care that the GPIO is "remote."

DT:

```
&i2c1 {
    mcp23017: gpio@20 {
        compatible = "microchip,mcp23017";
        reg = <0x20>;
        gpio-controller;
        #gpio-cells = <2>;
        interrupt-parent = <&gpio4>;
        interrupts = <14 IRQ_TYPE_LEVEL_LOW>;
    };
};

my_device {
    /* GPIOs come from the expander */
    reset-gpios = <&mcp23017 5 GPIO_ACTIVE_LOW>;
};
```

The driver code is *unchanged*, `devm_gpiod_get(&pdev->dev, "reset", ...)` works the same way. The only difference is that operations like `gpiod_set_value` may now sleep (they trigger an I²C transaction), so you must use `gpiod_set_value_cansleep` when the GPIO might be on an expander. **For portable driver code, always use `_cansleep` in process context.**

44.8 Key Notes

- Pinctrl owns mux and pad configuration, while GPIO consumers own direction and logical value through descriptors. Successful descriptor acquisition does not establish correct board wiring.
- `<purpose>-gpios` connects a consumer role to a controller and line. Active-low flags translate logical assertion into electrical polarity and must agree with the attached circuit.
- GPIO ownership prevents competing consumers from treating the same line as independent resources. A kernel driver and a user-space line request are alternatives, not two simultaneous control paths.
- Bus-backed GPIOs may sleep during access. Portable consumers use sleeping-context operations rather than assuming every descriptor maps to local MMIO.
- Active and sleep pinctrl states are electrical design choices. Pulls, output levels, and wake routing must remain safe when the peripheral’s power or clocks change.
- The chapter’s example pin assignments illustrate the interface, not verified MINI button or LED wiring. Physical use requires the fitted carrier/core mapping and electrical limits to be identified.

44.9 Pitfalls

- **Forgetting the pinctrl group.** Pin is still in its default mux (for example, UART). `gpiod_get` succeeds, the GPIO controller has no idea the pin is muxed elsewhere. But the GPIO seems "stuck": reads and writes hit the GPIO register, while the IOMUX routes the pin to UART. Always declare a pinctrl group that muxes the pin as GPIO, and reference it from `pinctrl-0`.
- **`GPIO_ACTIVE_LOW` confusion.** The kernel's logical "asserted" hides physical polarity. If you read raw via `/sys/class/gpio/`, you see physical level. If you read via `gpiod_get_value`, you see logical. Match your DT flag to your hardware schematic.
- **Wrong bank phandle.** `<&gpio1 ...>` vs `<&gpio2 ...>`, typo costs you hours. `gpioinfo` is your friend after boot.
- **Calling `gpiod_set_value` (atomic) on an I²C-backed GPIO.** Kernel BUG: "scheduling while atomic." Always use `_cansleep` unless you're 100 % sure the GPIO is direct (and even then, for new code, use `_cansleep` for portability).
- **Hog vs driver-owned pin.** Don't hog a pin that a driver will claim. The driver's `pinctrl_select_state` will fail. Hog only ownerless pins.
- **Using GPIO sysfs.** Old `/sys/class/gpio/export` interface is deprecated. Use `libgpiod` and `gpioset/gpioget/gpiomon` for user-space access. Sysfs may be missing entirely on newer kernels.
- **Pin number arithmetic.** "GPIO1 IO19 = global GPIO number 19" is wrong in some places, right in others. Global numbers are legacy. The descriptor API doesn't care about numbers. Just use the DT phandle + pin offset.
- **Forgetting `MODULE_DEVICE_TABLE`.** Driver works manually, doesn't autoload. Easy to miss. Always include it.

44.10 Going deeper

- **`Documentation/driver-api/pin-control.rst`**: the pinctrl subsystem's official architecture document.
- **`Documentation/driver-api/gpio/`**: gpiod consumer and provider APIs.
- **`Documentation/devicetree/bindings/pinctrl/fsl,imx-pinctrl.txt`**: the i.MX-specific pinctrl binding.
- **`arch/arm/boot/dts/imx6ul-pinfunc.h`**: all `MX6UL_PAD_*` macros defined here. Skim for an hour and you'll start to recognise the patterns.
- **`drivers/pinctrl/freescale/pinctrl-imx6ul.c`**: the i.MX6UL pinctrl driver (and yes, it works for 6ULL too).

- **`drivers/gpio/gpio-mxc.c`**: the i.MX GPIO controller driver. Demuxes 32 pins per bank into per-pin virqs, handles the chained IRQ.
- **`tools/gpio/`**: small helper utilities. Useful for one-off scripts.
- **`libgpiod` source on kernel.org**: the canonical user-space API.

> Next chapter: **Chapter 45: Input subsystem.** With GPIOs and IRQs in hand, the natural next step is “turn a button into a key-press event the OS recognises.” That’s the `input_dev` framework, and `gpio-keys` is its perfect canonical example.

Chapter 45: Input subsystem

> **What:** the **input subsystem**, the kernel framework that turns “a GPIO went low” or “an I²C read returned a touch coordinate” into a standardised event stream consumed by `evdev`, X11, Wayland, framebuffer toolkits, and command-line tools. A small button-driver case connects the IRQ handler to `evtest` reading `/dev/input/eventN` and contrasts it with the maintained `gpio-keys` driver. **MCU bridge:** Think of an IRQ like an EXTI/NVIC interrupt path, except Linux splits the hard interrupt from deferred work and must share lines across drivers. **MCU bridge:** Think of Linux GPIO like the same pin set/reset block you used on STM32, but accessed through a kernel subsystem that owns numbering, direction, interrupts, and user-space exposure. **IRQ:** interrupt request, the signal path that tells the CPU or interrupt controller that hardware needs service. **GPIO:** General-Purpose Input/Output, a pin controlled as a digital input, output, or interrupt source.
>
> **Why:** every input device on a Linux box, keyboard, mouse, touchscreen, joystick, IR remote, goes through the input subsystem. Once you understand `input_register_device` and `input_event`, *every* input driver in the kernel looks familiar. The framework handles event multiplexing, queueing, sysfs/`evdev` integration, autorepeat, and userspace device-node creation, your driver just calls `input_report_key()` and `input_sync()`. **sysfs:** a kernel-generated filesystem under /sys that exposes devices, drivers, and attributes.
>
> **Focus: type, code, value**, the three-element tuple that describes every input event. Once that triple makes sense, `EV_KEY` + `KEY_ENTER` + `1` means “Enter was pressed”, the rest of the input subsystem (abs axes, relative motion, multi-touch slots) is just different combinations of type/code/value.

45.1 The picture

When you press a key on a USB keyboard:

```
physical key down
     |
USB-HID interrupt-in transfer reports scan-code 0x28 (Enter)
     |
usbhid driver: hid_input_report → input_report_key(dev, KEY_ENTER, 1)
     |
input core:  queue an event {EV_KEY, KEY_ENTER, 1} on every evdev handler
     |
/dev/input/event3 becomes readable
     |
user-space: read() returns 24 bytes — a struct input_event
     |
X11 / Wayland / your application: "Enter was pressed"
```

The driver's only job is to call `input_report_*()` and `input_sync()`. The core handles queueing, multiplexing, and user-space delivery. **Your driver feeds events into the type/code/value protocol**. The input core delivers them to user-space. You never talk to user-space directly.

45.2 Event types and codes

> **Security boundary:** Do not burn fuses, enroll production keys, or sign release images when exploring example code. Use throwaway keys and back up the unsigned image plus the key directory before testing irreversible security flows.

`#include <linux/input.h>`, defines hundreds of `EV_*`, `KEY_*`, `BTN_*`, `ABS_*`, `REL_*`, `SW_*`, `LED_*`, `SND_*`, `MSC_*` constants.

Common event types:

Type	Meaning	Typical codes
`EV_SYN`	Synchronization, end-of-event-group marker	`SYN_REPORT` (always 0)
`EV_KEY`	Key/button pressed/released	`KEY_A`...`KEY_Z`, `KEY_ENTER`, `BTN_LEFT`, `BTN_TOUCH`
`EV_REL`	Relative axis (mouse motion, scroll wheel)	`REL_X`, `REL_Y`, `REL_WHEEL`
`EV_ABS`	Absolute axis (touchscreen, joystick, IMU)	`ABS_X`, `ABS_Y`, `ABS_PRESSURE`, `ABS_MT_SLOT`
`EV_MSC`	Miscellaneous (scancode, raw value pass-through)	`MSC_SCAN`, `MSC_RAW`
`EV_SW`	Switch (lid open/closed, headphone jack, dock)	`SW_LID`, `SW_HEADPHONE_INSERT`
`EV_LED`	LED state output (driver consumes from userspace)	`LED_NUML`, `LED_CAPSL`
`EV_SND`	Sound output (PC speaker beep)	`SND_BELL`, `SND_TONE`

Each event has a **value** appropriate to its type:

- `EV_KEY` value: `0` = released, `1` = pressed, `2` = autorepeat.
- `EV_REL` value: signed delta (+1, -3, etc.).
- `EV_ABS` value: absolute position, in whatever range the driver declared.
- `EV_SYN` value: usually 0.

A coherent group of events ends with `EV_SYN`/`SYN_REPORT`. The input core delivers all events between two `SYN_REPORT`s atomically to userspace, important when a single touch update sends multiple coordinates that must arrive together.

45.3 The mainline `gpio-keys` driver

The kernel already has `gpio-keys`, the in-tree driver that exposes DT-described GPIOs as keyboard keys. It is the normal implementation choice when its capabilities match the product. The DT looks like this:

```
gpio_keys {
    compatible = "gpio-keys";

    key-enter {
        label = "Enter";
        linux,code = <KEY_ENTER>;
        gpios = <&gpio1 19 GPIO_ACTIVE_LOW>;
        debounce-interval = <50>;
        wakeup-source;
    };

    key-up {
        label = "Up";
        linux,code = <KEY_UP>;
        gpios = <&gpio4 14 GPIO_ACTIVE_LOW>;
        debounce-interval = <50>;
    };
};
```

Set `CONFIG_KEYBOARD_GPIO=y` in the kernel config, boot. Pressing the GPIO19 button now generates real `KEY_ENTER` events on `/dev/input/eventN`.

The smaller version below isolates the framework's registration and reporting interfaces without requiring a separate implementation from the reader.

45.4 Writing our own button driver

`button_input.c`:

```
#include <linux/init.h>
#include <linux/module.h>
```

```
#include <linux/platform_device.h>
#include <linux/gpio/consumer.h>
#include <linux/input.h>
#include <linux/interrupt.h>

struct button_dev {
    struct input_dev *input;
    struct gpio_desc *gpio;
    int irq;
};

static irqreturn_t button_irq(int irq, void *dev_id)
{
    struct button_dev *bd = dev_id;
    int pressed = gpiod_get_value_cansleep(bd->gpio);
    /* The gpiod API has already applied ACTIVE_LOW polarity:
       pressed=1 here means logically pressed. */
    input_report_key(bd->input, KEY_ENTER, pressed);
    input_sync(bd->input);
    return IRQ_HANDLED;
}

static int button_probe(struct platform_device *pdev)
{
    struct button_dev *bd;
    int err;

    bd = devm_kzalloc(&pdev->dev, sizeof(*bd), GFP_KERNEL);
    if (!bd)
        return -ENOMEM;

    bd->gpio = devm_gpiod_get(&pdev->dev, "button", GPIOD_IN);
    if (IS_ERR(bd->gpio))
        return dev_err_probe(&pdev->dev, PTR_ERR(bd->gpio), "no button gpio\n");

    bd->irq = gpiod_to_irq(bd->gpio);
    if (bd->irq < 0)
        return bd->irq;

    bd->input = devm_input_allocate_device(&pdev->dev);
    if (!bd->input)
        return -ENOMEM;

    bd->input->name = "linuxlearn-button";
    bd->input->phys = "button/input0";
    bd->input->id.bustype = BUS_HOST;

    /* Declare what events this device can emit */
    input_set_capability(bd->input, EV_KEY, KEY_ENTER);

    err = input_register_device(bd->input);
    if (err)
        return dev_err_probe(&pdev->dev, err, "input register failed\n");

    err = devm_request_threaded_irq(&pdev->dev, bd->irq, NULL, button_irq,
                                    IRQF_TRIGGER_RISING | IRQF_TRIGGER_FALLING
                                    | IRQF_ONESHOT,
                                    "linuxlearn-button", bd);
    if (err)
```

```
        return err;

    platform_set_drvdata(pdev, bd);
    return 0;
}

static const struct of_device_id button_of_match[] = {
    { .compatible = "linuxlearn,input-button" },
    { }
};
MODULE_DEVICE_TABLE(of, button_of_match);

static struct platform_driver button_driver = {
    .driver = {
        .name = "linuxlearn-input-button",
        .of_match_table = button_of_match,
    },
    .probe = button_probe,
};
module_platform_driver(button_driver);

MODULE_LICENSE("GPL");
```

The flow:

1. `devm_input_allocate_device()` allocates an `input_dev` and ties its lifetime to the parent device.
2. We fill in identification, `name` (visible in `evtest`), `phys` (physical-bus path), `id.bustype` (origin family).
3. `input_set_capability(input, EV_KEY, KEY_ENTER)` declares “this device can emit `EV_KEY` events with code `KEY_ENTER`.” User-space tools query this via the EVIOCGBIT ioctl to learn what events to expect.
4. `input_register_device()` registers with the core. A `/dev/input/eventN` node appears.
5. In the IRQ handler, `input_report_key()` + `input_sync()` deliver one event.

DT:

```
my_button {
    compatible = "linuxlearn,input-button";
    button-gpios = <&gpio1 19 GPIO_ACTIVE_LOW>;
    pinctrl-names = "default";
    pinctrl-0 = <&pinctrl_button>;
};
```

Build, load:

```
[root@pa-mini:~]# insmod button_input.ko
[root@pa-mini:~]# cat /proc/bus/input/devices
I: Bus=0019 Vendor=0000 Product=0000 Version=0000
N: Name="linuxlearn-button"
```

```
P: Phys=button/input0
S: Sysfs=/devices/platform/my_button/input/input2
H: Handlers=evdev event2
B: PROP=0
B: EV=3
B: KEY=10000000 ... (sparse bit; KEY_ENTER is bit 28)

[root@pa-mini:~]# evtest /dev/input/event2
... waits ...
Press button: Event: time 4242.12, type 1 (EV_KEY), code 28 (KEY_ENTER), value 1
            Event: time 4242.12, ------------ SYN_REPORT -----------
Release:    Event: time 4242.45, type 1 (EV_KEY), code 28 (KEY_ENTER), value 0
            Event: time 4242.45, ------------ SYN_REPORT -----------
```

Done. Button is a real keyboard key.

45.5 Auto-repeat, debounce, and key mapping

The `input` core provides **autorepeat** for free: register `EV_REP` capability and the core will generate repeat events (value=2) while a key is held. `gpio-keys` does this automatically. Our minimal driver doesn't bother.

Debounce filters electrical transitions into a stable logical state. A delayed-work design postpones sampling until the debounce interval has elapsed, then reports the stable press or release state. Further transitions restart the settling interval. The minimal driver above omits this policy, while `gpio-keys` provides debounce handling.

Key mapping. The `KEY_*` codes are *logical*, `KEY_ENTER` is the same value regardless of whether the user's keymap is US QWERTY or Dvorak. The user-space keymap translates `KEY_*` to characters. For embedded devices with only a few buttons, you pick meaningful `KEY_*` codes:

- `KEY_VOLUMEUP` / `KEY_VOLUMEDOWN` for media buttons.
- `KEY_POWER` for the power button (the kernel and systemd both recognise this).
- `KEY_HOME`, `KEY_BACK`, `KEY_MENU` for navigation.
- `KEY_WAKEUP` for a wakeup-from-suspend button.

The full list is in `include/uapi/linux/input-event-codes.h`. Pick a code that matches the *role* of the button. User-space will know what to do with it.

45.6 Absolute axes, touchscreens and joysticks

For a touchscreen or joystick, you have *coordinates*, not key presses. Declare `EV_ABS` capabilities and report values:

```
input_set_abs_params(input, ABS_X, 0, 4095, 0, 0);
input_set_abs_params(input, ABS_Y, 0, 4095, 0, 0);
input_set_abs_params(input, ABS_PRESSURE, 0, 255, 0, 0);
```

```
/* In the IRQ/work handler when a sample arrives: */
input_report_abs(input, ABS_X, x_coord);
input_report_abs(input, ABS_Y, y_coord);
input_report_abs(input, ABS_PRESSURE, pressure);
input_report_key(input, BTN_TOUCH, 1);
input_sync(input);
```

`input_set_abs_params(dev, code, min, max, fuzz, flat)`, `min/max` is the valid range, `fuzz` is the noise floor (changes ≤ fuzz are suppressed), `flat` is the center deadzone for joysticks.

For **multi-touch**, the protocol is more involved, the MT-B (slot-based) protocol uses `ABS_MT_SLOT` + per-finger `ABS_MT_POSITION_X/Y`. We'll cover that in Ch 55G (GT911 multi-touch driver).

45.7 Polled vs interrupt-driven

Some input devices don't have IRQs, accelerometers configured for continuous mode, for instance, or a button on a slow I²C expander where IRQ wiring isn't practical. The `input_polled_dev` framework (now subsumed into the regular `input_dev` via `input_setup_polling`) polls a device at a fixed rate:

```
input_setup_polling(input, my_poll_callback);
input_set_poll_interval(input, 20);   /* 20 ms = 50 Hz */
input_register_device(input);
```

The core calls `my_poll_callback(input)` every 20 ms. Inside, sample the hardware, `input_report_*`, `input_sync`. No IRQ wiring needed.

45.8 User-space, evdev

`/dev/input/event*` is the chardev that streams `struct input_event` records to user-space:

```
struct input_event {
    struct timeval time;
    __u16 type;
    __u16 code;
    __s32 value;
};
```

Read one record per event. The most common tool is `evtest` (interactive) for debugging. For production, applications either use `libevdev` (a thin wrapper) or higher-level libraries:

- **`libinput`**: used by Wayland and modern X11. Handles gesture recognition, palm rejection, tap-to-click, etc.
- **`libevdev`**: minimal wrapper for reading raw events.

- **`/dev/input/eventN` directly**: fine for embedded with one app.

For a quick test from shell:

```
$ sudo evtest /dev/input/event2     # interactive
$ sudo cat /dev/input/event2 | hexdump -C
```

45.9 Key Notes

- An input driver translates device-specific observations into declared type/code/value events. Evdev and higher-level user-space consumers own delivery and interpretation.
- Capabilities describe the device's event vocabulary before registration. `input_sync()` marks the end of a coherent update, such as a key transition or coordinate sample.
- Press and release are both state changes. Debounce and autorepeat are distinct policies, and a simple edge handler does not supply both automatically.
- `gpio-keys` is an implementation choice, while the custom button case exposes the same architectural boundary with fewer features. Neither DT examples nor event listings qualify MINI wiring.
- Logical key codes describe roles, not characters or mandatory system actions. Power-button shutdown policy belongs to the user-space system configuration.
- IRQs, polling, and delayed work must stop reporting before the input device's state is released. Framework registration does not remove the need for coordinated teardown.

45.10 Pitfalls

- **Forgetting `input_sync`.** Events are buffered until `SYN_REPORT`. Without it, user-space never sees them. After any group of `input_report_*` calls, call `input_sync`.
- **Reporting an unsupported event.** If you `input_report_key(input, KEY_ENTER, 1)` but didn't `input_set_capability(..., EV_KEY, KEY_ENTER)`, the event is silently dropped. Always declare capabilities first.
- **Not using `devm_input_allocate_device`.** Forgetting `input_free_device` in error paths leaks the device. `devm_` handles it.
- **Calling `input_register_device` before setting capabilities.** Capabilities must be set *before* register. Order: alloc → set_capability → register.

- **Mixing `input_allocate_device` with separate `input_register_device`.** Both can fail, at different points. Use standard `goto` cleanup, or just use `devm_input_allocate_device` to avoid the problem.
- **Confusing absolute and relative axes.** Mice use `EV_REL` (delta motion). Touchscreens use `EV_ABS` (absolute position). Mixing them gives weird user-space behavior.
- **Multi-touch with single-touch protocol.** Don't try to emit `ABS_X` for multiple fingers, that's not how it works. Use the MT-B slot protocol (Ch 55G).
- **Repeating events that haven't actually changed.** The core does *not* dedupe. Every `input_report_key(..., 1)` followed by `input_sync` is one event. Polling a held button without state-tracking spams the queue.
- **IRQ flag `IRQF_TRIGGER_FALLING` without `IRQF_TRIGGER_RISING`**: you only get press events, not release. For a press/release-capable button, request *both* edges (`IRQF_TRIGGER_RISING | IRQF_TRIGGER_FALLING`).

45.11 Going deeper

- **`Documentation/input/`**: the input subsystem's full documentation. Read `input.rst`, `event-codes.rst`, `multi-touch-protocol.rst`.
- **`include/uapi/linux/input-event-codes.h`**: the canonical list of `EV_*`, `KEY_*`, `ABS_*`, etc.
- **`drivers/input/keyboard/gpio_keys.c`**: the in-tree gpio-keys driver. Read it. ~600 lines and covers debounce, wakeup, autorepeat, runtime PM, polling, every feature you'd add to a production button driver.
- **`drivers/input/evdev.c`**: the evdev "handler" that exposes input events as `/dev/input/eventN`.
- **`libevdev` source** (freedesktop.org), for high-level user-space input access.
- **`Documentation/input/input.rst`**: overview of the input architecture, including the handler/handle/dev triangle.

> Next chapter: **Chapter 46: I²C drivers.** With GPIO and input behind us, we move to a real bus: the i.MX6ULL's I²C controllers, the `i2c_client` / `i2c_driver` model, and how a single I²C bus accommodates a half-dozen sensors at different addresses.

Chapter 46: I²C drivers

What: the Linux **I²C subsystem**, `i2c_adapter` (the controller), `i2c_client` (one chip on the bus), `i2c_driver` (the per-chip driver), and `i2c_msg` (a transfer message). The examples connect DT matching and bus transactions to a chip-specific interface exposed through a subsystem or sysfs. **sysfs:** a kernel-generated filesystem under /sys that exposes devices, drivers, and attributes.

Why: I²C is the most common slow bus in embedded systems: temp sensors, EEPROMs, GPIO expanders, RTCs, audio codecs, touch controllers, PMICs, battery gauges, half of Part VII's cookbook chapters are I²C devices. The Linux I²C model is a *clean* example of the bus/driver/device split first introduced in Ch 39, and every I²C driver looks the same once you know the shape. **MCU bridge:** Think of Linux GPIO like the same pin set/reset block you used on STM32, but accessed through a kernel subsystem that owns numbering, direction, interrupts, and user-space exposure. **GPIO:** General-Purpose Input/Output, a pin controlled as a digital input, output, or interrupt source.

Focus: i2c_msg is the universal transaction. `i2c_transfer(adapter, msgs, count)` sends a sequence of `i2c_msg` structs. Each is one read or write, and adjacent ones share the bus without a STOP between them. Get this primitive right and you can talk to any I²C chip: write-then-read, repeated-start, 10-bit addressing, SMBus quirks.

Tooling. This chapter uses `i2c-tools` (`i2cdetect`, `i2cdump`, `i2cset`, `i2cget`).

- **Ubuntu-base (target):** `apt install i2c-tools`
- **Buildroot:** `BR2_PACKAGE_I2C_TOOLS=y` **Buildroot:** a configuration-driven build system that produces a complete root filesystem and related images.
- Full per-tool reference: Userspace tooling appendix. **MCU bridge:** Think of the rootfs as the firmware image's file-backed runtime environment. On an MCU you link everything into flash. On Linux, programs and config live in this mounted tree. **rootfs:** root filesystem, the directory tree mounted at / that contains /bin, /etc, /dev, and libraries.

46.1 The split: adapter vs client vs driver

Three players:

```
i2c_adapter      i2c_client       i2c_driver
─────────────    ──────────       ──────────
the controller   one chip on      the code that
(SoC peripheral) the bus           knows the chip's
```

```
                                        registers
   "i2c-imx"        "BME280 at 0x76"  "bme280"
   on bus #1
```

- **`i2c_adapter`** is the I²C controller driver, i.MX6ULL has 4 (I2C1, I2C2, I2C3, I2C4). NXP wrote `drivers/i2c/busses/i2c-imx.c`. You don't touch this unless you're porting Linux to a new SoC. It exposes the bus as `/dev/i2c-N` and registers with the kernel's I²C core.
- **`i2c_client`** describes one chip: bus number, 7-bit address, name. The kernel creates these from DT nodes, for every child node of an `&i2c1` block, you get one `i2c_client`.
- **`i2c_driver`** is what you write. It declares "I handle chips whose DT `compatible` is X" and provides `probe()` / `remove()` callbacks.

When the kernel parses DT, it sees:

```
&i2c1 {
    bme280@76 {
        compatible = "bosch,bme280";
        reg = <0x76>;
    };
};
```

...and creates an `i2c_client` with `addr = 0x76`, `name = "bme280"`. When your `i2c_driver` registers, the I²C core walks every client on every adapter. For each one whose `compatible` is in your `of_match_table`, it calls your `probe()`.

46.2 Device tree for I²C

The i.MX6ULL has 4 I²C controllers. Enable one in your board DT:

```
&i2c1 {
    clock-frequency = <100000>;    /* 100 kHz standard mode */
    pinctrl-names = "default";
    pinctrl-0 = <&pinctrl_i2c1>;
    status = "okay";

    /* children = devices on the bus */
    eeprom: at24@50 {
        compatible = "atmel,24c02";
        reg = <0x50>;
        pagesize = <8>;
    };

    bme280: bme280@76 {
        compatible = "bosch,bme280";
        reg = <0x76>;
    };

    expander: gpio@20 {
        compatible = "microchip,mcp23017";
```

```
        reg = <0x20>;
        gpio-controller;
        #gpio-cells = <2>;
    };
};
```

Three rules:

1. **`reg` is the I²C address** (7-bit, low 7 bits, so 0x76 means address pattern `1110110`). Not a memory address.
2. **The unit-name** (`bme280@76`) must match `reg` in hex. This is a DT lint rule. Mismatch produces a warning.
3. **`clock-frequency`** sets the bus speed: 100 kHz (standard), 400 kHz (fast), 1 MHz (fast-plus), or 3.4 MHz (high-speed). Limited by your slowest device on the bus.

46.3 Anatomy of an I²C driver

The minimal skeleton:

```
#include <linux/init.h>
#include <linux/module.h>
#include <linux/i2c.h>
#include <linux/of.h>

struct mychip_priv {
    struct i2c_client *client;
    /* device state */
};

static int mychip_probe(struct i2c_client *client)
{
    struct mychip_priv *priv;
    int chip_id;

    priv = devm_kzalloc(&client->dev, sizeof(*priv), GFP_KERNEL);
    if (!priv)
        return -ENOMEM;
    priv->client = client;
    i2c_set_clientdata(client, priv);

    /* Read chip-id register to verify the chip is present */
    chip_id = i2c_smbus_read_byte_data(client, 0xD0);   /* chip-id reg */
    if (chip_id < 0)
        return dev_err_probe(&client->dev, chip_id, "read chip-id failed\n");
    if (chip_id != 0x60)
        return dev_err_probe(&client->dev, -ENODEV,
                             "unexpected chip-id 0x%02x\n", chip_id);

    dev_info(&client->dev, "mychip ready at 0x%02x\n", client->addr);
    return 0;
}

static void mychip_remove(struct i2c_client *client)
```

```
{
    /* devm_* cleans up; nothing to do */
}

static const struct i2c_device_id mychip_id[] = {
    { "mychip", 0 },
    { }
};
MODULE_DEVICE_TABLE(i2c, mychip_id);

static const struct of_device_id mychip_of_match[] = {
    { .compatible = "linuxlearn,mychip" },
    { }
};
MODULE_DEVICE_TABLE(of, mychip_of_match);

static struct i2c_driver mychip_driver = {
    .driver = {
        .name = "mychip",
        .of_match_table = mychip_of_match,
    },
    .probe = mychip_probe,
    .remove = mychip_remove,
    .id_table = mychip_id,
};
module_i2c_driver(mychip_driver);

MODULE_LICENSE("GPL");
```

Key things to notice:

`module_i2c_driver(mychip_driver)`, like `module_platform_driver`, this is a macro that generates the `module_init`/`module_exit` pair. One-liner registration.

`i2c_set_clientdata` / `i2c_get_clientdata`, store and retrieve your private struct pointer on the client. Used to find the priv struct from later callbacks.

Two match tables, `i2c_device_id` for non-DT systems (the `id_table`) *and* `of_device_id` for DT. Modern systems use DT. The `i2c_device_id` is the historical fallback. Include both for portability.

`MODULE_DEVICE_TABLE(i2c, mychip_id)`, exposes the table to depmod so the right module autoloads when an I²C device with that name is detected.

46.4 SMBus API, the simple case

For one-byte, one-word, and block transfers, use the **SMBus** API:

```
int  i2c_smbus_read_byte_data(struct i2c_client *client, u8 reg);
int  i2c_smbus_write_byte_data(struct i2c_client *client, u8 reg, u8 val);
int  i2c_smbus_read_word_data(struct i2c_client *client, u8 reg);
int  i2c_smbus_write_word_data(struct i2c_client *client, u8 reg, u16 val);
int  i2c_smbus_read_i2c_block_data(struct i2c_client *client, u8 reg, u8 len, u8 *buf);
```

```
int  i2c_smbus_write_i2c_block_data(struct i2c_client *client, u8 reg, u8 len, const u8
*buf);
```

Each function does the “write the register address, then read N bytes” pattern (or write equivalent) in one call. They return the read value (or negative errno) for the read variants. 0 or negative for writes.

A typical sensor read:

```
/* Read temperature: register 0xFA, 3 bytes */
u8 raw[3];
int err = i2c_smbus_read_i2c_block_data(client, 0xFA, 3, raw);
if (err < 0)
    return err;
int temp_raw = (raw[0] << 12) | (raw[1] << 4) | (raw[2] >> 4);
```

SMBus is technically a subset of I²C with stricter timing, but the SMBus helpers work on any standard I²C bus. Almost all chips with “register N, read M bytes” semantics work with SMBus calls.

46.5 Raw `i2c_transfer`, when SMBus isn’t enough

Some chips need transactions SMBus doesn’t model directly, for example, a write of 5 bytes followed by a read of 200 bytes, or a write with no repeated-start. For those, drop down to the lowest-level API:

```
struct i2c_msg msgs[2] = {
    {
        .addr  = client->addr,
        .flags = 0,                  /* write */
        .len   = 2,
        .buf   = (u8[]){ 0xFA, 0x00 }, /* 2-byte register address */
    },
    {
        .addr  = client->addr,
        .flags = I2C_M_RD,           /* read; repeated-start before this msg */
        .len   = 16,
        .buf   = rxbuf,
    },
};

int ret = i2c_transfer(client->adapter, msgs, ARRAY_SIZE(msgs));
if (ret != 2)
    return ret < 0 ? ret : -EIO;
```

The kernel issues the entire message group atomically, START, addr+W, two write bytes, repeated-START, addr+R, 16 read bytes, STOP. Between two `i2c_msg`s there’s a *repeated start* (no STOP). If you wanted separate transactions with bus release in between, two separate `i2c_transfer` calls.

`i2c_msg.flags` bits:

- `I2C_M_RD`: this message is a read.
- `I2C_M_TEN`: 10-bit address (rare).
- `I2C_M_NOSTART`: don't issue START for this message (concatenates with previous. Rare).
- `I2C_M_IGNORE_NAK`: don't treat NAK as an error (for "best-effort" writes).

46.6 A complete example: AT24C02 EEPROM-backed sysfs

Let's make this concrete with a worked example: a driver for the AT24C02 (256-byte EEPROM) that exposes its contents via a sysfs `data` binary attribute. (Note: the real `at24` driver in the kernel does this and more. This is a simplified version for learning.)

```
#include <linux/init.h>
#include <linux/module.h>
#include <linux/i2c.h>
#include <linux/sysfs.h>

struct at24_priv {
    struct i2c_client *client;
    struct bin_attribute attr;
    size_t size;
};

static ssize_t at24_read(struct file *filp, struct kobject *kobj,
                         struct bin_attribute *attr,
                         char *buf, loff_t off, size_t count)
{
    struct at24_priv *p = dev_get_drvdata(kobj_to_dev(kobj));
    struct i2c_msg msgs[2];
    u8 addr = off;

    if (off + count > p->size)
        count = p->size - off;
    if (count == 0)
        return 0;

    msgs[0] = (struct i2c_msg){
        .addr = p->client->addr, .flags = 0, .len = 1, .buf = &addr,
    };
    msgs[1] = (struct i2c_msg){
        .addr = p->client->addr, .flags = I2C_M_RD, .len = count, .buf = buf,
    };

    int n = i2c_transfer(p->client->adapter, msgs, 2);
    return n == 2 ? count : -EIO;
}

static ssize_t at24_write(struct file *filp, struct kobject *kobj,
                          struct bin_attribute *attr,
                          char *buf, loff_t off, size_t count)
```

```
{
    struct at24_priv *p = dev_get_drvdata(kobj_to_dev(kobj));
    u8 tx[1 + 8];      /* 1 addr + page-size data */
    size_t done = 0;

    if (off + count > p->size)
        count = p->size - off;

    while (done < count) {
        /* AT24C02 has 8-byte pages; can't cross a page boundary in one write */
        size_t pageoff = (off + done) % 8;
        size_t chunk = min_t(size_t, count - done, 8 - pageoff);
        tx[0] = off + done;
        memcpy(tx + 1, buf + done, chunk);

        struct i2c_msg msg = {
            .addr = p->client->addr, .flags = 0, .len = chunk + 1, .buf = tx,
        };
        int n = i2c_transfer(p->client->adapter, &msg, 1);
        if (n != 1) return -EIO;

        msleep(5);     /* AT24C02 internal write cycle: ≤5 ms */
        done += chunk;
    }
    return count;
}

static int at24_probe(struct i2c_client *client)
{
    struct at24_priv *p;
    int err;

    p = devm_kzalloc(&client->dev, sizeof(*p), GFP_KERNEL);
    if (!p) return -ENOMEM;

    p->client = client;
    p->size = 256;     /* AT24C02 = 256 bytes */

    sysfs_bin_attr_init(&p->attr);
    p->attr.attr.name = "data";
    p->attr.attr.mode = 0660;
    p->attr.size      = p->size;
    p->attr.read      = at24_read;
    p->attr.write     = at24_write;

    dev_set_drvdata(&client->dev, p);

    err = sysfs_create_bin_file(&client->dev.kobj, &p->attr);
    if (err)
        return dev_err_probe(&client->dev, err, "sysfs create failed\n");

    dev_info(&client->dev, "AT24C02 %zu-byte EEPROM ready\n", p->size);
    return 0;
}

static void at24_remove(struct i2c_client *client)
{
    struct at24_priv *p = dev_get_drvdata(&client->dev);
    sysfs_remove_bin_file(&client->dev.kobj, &p->attr);
```

```
}

static const struct of_device_id at24_of_match[] = {
    { .compatible = "linuxlearn,at24c02" },
    { }
};
MODULE_DEVICE_TABLE(of, at24_of_match);

static struct i2c_driver at24_driver = {
    .driver = {
        .name = "linuxlearn-at24",
        .of_match_table = at24_of_match,
    },
    .probe  = at24_probe,
    .remove = at24_remove,
};
module_i2c_driver(at24_driver);

MODULE_LICENSE("GPL");
```

Build, load. From user-space:

```
[root@pa-mini:~]# ls /sys/bus/i2c/devices/1-0050/
data  driver/  name  ...

[root@pa-mini:~]# echo -n "hello world" > /sys/bus/i2c/devices/1-0050/data
[root@pa-mini:~]# dd if=/sys/bus/i2c/devices/1-0050/data bs=11 count=1
hello world
```

11 bytes wrote, 11 bytes read. The `dd` triggered an `at24_read` that issued a write-then-read transaction. The `echo` triggered an `at24_write` that paged out the data 8 bytes at a time with 5 ms waits between pages.

The mainline `at24` driver (`drivers/misc/eeprom/at24.c`) does this and more, nvmem integration, large-EEPROM support, write-protect-GPIO, etc. Read it.

46.7 /dev/i2c-N, user-space access

For development, prototyping, and quick chip inspection, user-space can talk directly to any I²C bus via the `i2c-dev` chardev:

```
[root@pa-mini:~]# modprobe i2c-dev
[root@pa-mini:~]# ls /dev/i2c-*
/dev/i2c-0  /dev/i2c-1  /dev/i2c-2  /dev/i2c-3
```

Install the `i2c-tools` package and you get `i2cdetect`, `i2cget`, `i2cset`, `i2cdump`, `i2ctransfer`:

```
[root@pa-mini:~]# i2cdetect -y 1
     0  1  2  3  4  5  6  7  8  9  a  b  c  d  e  f
00:                         -- -- -- -- -- -- -- --
10: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
20: 20 -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
```

```
30: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
40: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
50: 50 -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
60: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
70: -- -- -- -- -- -- 76 -- -- -- -- -- -- -- -- --

[root@pa-mini:~]# i2cget -y 1 0x76 0xD0
0x60                                    ← BME280 chip-id read

[root@pa-mini:~]# i2cset -y 1 0x50 0x00 0xAB
                                        ← write 0xAB to AT24 address 0
```

If a kernel driver has already bound to the device, `i2c-tools` will refuse to touch it (you would race the driver). Pass `-y -f` to override, only for known-safe testing.

46.8 Key Notes

- The adapter owns controller operation, the client identifies one bus address, and the chip driver owns device semantics. A DT client description is not proof of a responding chip.
- SMBus helpers suit defined SMBus-style operations, while `i2c_transfer` describes message sequences such as a repeated-start register read. The device protocol and adapter capabilities constrain the choice.
- `i2c_transfer` returns a completed-message count, not a byte count. A nonnegative partial result cannot be treated as completion of the requested sequence.
- Bus serialization protects an individual transfer group. Multi-transfer operations, such as an EEPROM write and its completion wait, may also need device-level serialization.
- EEPROM page boundaries, capacity, write-cycle timing, and write protection are separate constraints. A valid bus transaction can still overwrite the wrong data or race a protection change.
- Direct `i2c-dev` access is a separate control path. Forcing access to a driver-owned address can invalidate the driver's assumptions and is not a substitute for coordinated ownership.

46.9 Pitfalls

- **Wrong unit-name vs `reg`.** `bme280@76` with `reg = <0x77>` doesn't match, kernel warns, but the device may still probe at 0x77 (the `reg` wins). DT lint catches it. Check your DT compilation log.
- **Missing pinctrl.** SDA/SCL pins not muxed to I²C function. Symptom: `i2cdetect` shows nothing. Use `gpioinfo` to confirm pins aren't claimed as GPIO instead.

- **Pull-up resistors missing.** I²C requires external pull-ups (typically 4.7 kΩ to 3.3 V). The SoC's internal pull-ups (~50 kΩ) usually work at 100 kHz but fail at 400 kHz. Schematic problem, not software.
- **Bus contention with multiple drivers.** Two drivers each claiming the same address → second one fails to probe. `dmesg` says `"address 0x76 already in use"`. Check for hidden DT children.
- **Wrong clock-frequency.** Some chips can't handle 400 kHz. The bus runs as fast as `clock-frequency` says. Slower devices NAK or corrupt data. Drop to 100 kHz when in doubt.
- **Calling SMBus block read with the wrong size.** SMBus has `read_i2c_block_data` (variable length) and `read_block_data` (size sent by chip). They look similar. Check your chip's protocol.
- **Mixing `i2c_master_send/recv` and `i2c_transfer` styles.** They work, but `i2c_transfer` is more flexible. New code should prefer it.
- **Forgetting `MODULE_DEVICE_TABLE`.** Auto-loading silently fails.
- **Address aliasing.** Some chips support multiple addresses based on address-pin pull. If two of the same chip are on the bus, configure them with different pull strappings. Otherwise both respond to the same address.

46.10 Going deeper

- **`Documentation/i2c/`**: full I²C subsystem documentation. Read `summary.rst`, `writing-clients.rst`, `instantiating-devices.rst`.
- **`Documentation/devicetree/bindings/i2c/`**: DT bindings for the I²C subsystem.
- **`drivers/i2c/busses/i2c-imx.c`**: the i.MX I²C controller driver. ~1200 lines. Worth reading once.
- **`drivers/misc/eeprom/at24.c`**: production-grade EEPROM driver with nvmem, regmap, and write-protect support.

> **MCU bridge:** Think of regmap like a typed wrapper around your read_reg() and write_reg() helpers, with caching, locking, and bus differences handled centrally. **regmap:** a kernel helper that wraps register reads and writes over I2C, SPI, or MMIO.

- **`drivers/iio/pressure/bmp280-i2c.c`** + `bmp280-core.c`, clean modern I²C IIO driver. Pairs nicely with Ch 49.

> **IIO:** Industrial I/O, Linux's subsystem for sensors, ADCs, DACs, and buffered sampled data.

- **`drivers/i2c/i2c-core.c`**: the I²C core itself. Hundreds of pages of API to skim.

> Next chapter: **Chapter 47: SPI drivers.** Same shape as I²C: a controller subsystem we don't write, a `spi_device` per chip, a `spi_driver` per chip-class, but now with full-duplex transactions and per-CS configuration.

Chapter 47: SPI drivers

> **What:** the Linux **SPI subsystem**, `spi_master` (the controller, again written by the SoC vendor), `spi_device` (one chip on a chip-select), `spi_driver` (the per-chip code), `spi_message` (a sequence of transfers). Same shape as I²C but with full-duplex transactions and per-CS independent configuration (mode, speed, word size).
>
> **Why:** SPI carries the high-bandwidth peripherals, NOR flash, LCDs, fast ADCs, IMUs at >1 kHz, Ethernet PHYs over SPI (W5500, ENC28J60), CAN controllers over SPI (MCP2515). Speeds from 1 MHz to 50 MHz are common, vs I²C's 100–400 kHz. The Linux model lets each chip-select have its own mode and clock without the driver caring about the controller's specifics.
>
> **Focus: `spi_message` is a list of `spi_transfer`s, executed back-to-back without releasing CS unless you ask**. Mastering this lets you build any chip's command sequence, "write 1 cmd byte, then read 4 data bytes, holding CS through both" is just a two-transfer message.

47.1 How SPI differs from I²C

Property	I²C	SPI
Wires	2 (SDA, SCL)	4 (MOSI, MISO, SCK, CS) + 1 per chip-select
Topology	Multi-drop on one pair	Star: separate CS per chip
Addressing	7-bit on the wire	Implicit (CS asserted)
Direction	Half-duplex (one direction at a time)	Full-duplex (both at once)
Speed	100 kHz, 1 MHz typical	1, 50 MHz typical
Bus arbitration	Built-in (NAK, arbitration loss)	None (only one master per CS)
Protocol overhead	Address + ACK per byte	None, just clock bits

For an SPI bus, you pick one **chip-select** per device. The controller asserts CS, clocks bytes, deasserts CS. Each CS has its own configuration: clock speed, polarity (CPOL), phase (CPHA), word size (usually 8 bits, sometimes 16/32 for DMA-friendly).

> **MCU bridge:** Think of DMA like the MCU DMA controller you used for UART or SPI, but with cache coherency, scatter-gather descriptors, and kernel ownership rules added. **DMA:** Direct Memory Access. Hardware moves data to or from memory without the CPU copying each byte.

The i.MX6ULL has 4 **eCSPI** controllers (eCSPI1–4), each with up to 4 chip-select lines (so up to 16 SPI devices on the SoC in theory).

47.2 Device tree for SPI

```
&ecspi3 {
    pinctrl-names = "default";
    pinctrl-0 = <&pinctrl_ecspi3>;
    cs-gpios = <&gpio4 5 GPIO_ACTIVE_LOW>;   /* optional GPIO-CS */
    status = "okay";

    flash@0 {
        compatible = "winbond,w25q128", "jedec,spi-nor";
        reg = <0>;                              /* chip-select 0 */
        spi-max-frequency = <50000000>;         /* 50 MHz */
        spi-tx-bus-width = <1>;
        spi-rx-bus-width = <1>;
    };

    adc@1 {
        compatible = "linuxlearn,fastadc";
        reg = <1>;                              /* chip-select 1 */
        spi-max-frequency = <10000000>;         /* 10 MHz */
        spi-cpol;                               /* CPOL=1, CPHA=0 (mode 2) */
    };
};
```

Differences from I²C:

- **`reg` is the chip-select index**, not an address.
- **`spi-max-frequency`** is per-device. The controller uses the lower of (its max, this).
- **`spi-cpol` / `spi-cpha`** set the SPI mode (omit both = mode 0. Both = mode 3).
- **`cs-gpios`**: optionally specify GPIO-based CS lines instead of the controller's native CS pins. Common when the controller's native CS doesn't quite behave the way you want (e.g., toggling between messages).

> **MCU bridge:** Think of Linux GPIO like the same pin set/reset block you used on STM32, but accessed through a kernel subsystem that owns numbering, direction, interrupts, and user-space exposure. **GPIO:** General-Purpose Input/Output, a pin controlled as a digital input, output, or interrupt source.

47.3 An SPI driver skeleton

```
#include <linux/init.h>
#include <linux/module.h>
#include <linux/spi/spi.h>
#include <linux/of.h>

struct fastadc_priv {
    struct spi_device *spi;
    struct mutex lock;
};

static int fastadc_read_word(struct fastadc_priv *p, u16 *out)
{
    u8 tx[2] = { 0x80, 0x00 };   /* read command */
    u8 rx[2];
    struct spi_transfer xfer = {
        .tx_buf = tx,
        .rx_buf = rx,
        .len    = 2,
        .speed_hz = 10000000,    /* override device default if desired */
    };
    int err = spi_sync_transfer(p->spi, &xfer, 1);
    if (err < 0)
        return err;
    *out = ((u16)rx[0] << 8) | rx[1];
    return 0;
}

static int fastadc_probe(struct spi_device *spi)
{
    struct fastadc_priv *p;
    u16 sample;
    int err;

    p = devm_kzalloc(&spi->dev, sizeof(*p), GFP_KERNEL);
    if (!p)
        return -ENOMEM;
    p->spi = spi;
    mutex_init(&p->lock);

    /* Configure SPI bus parameters */
    spi->mode          = SPI_MODE_0;
    spi->bits_per_word = 8;
    err = spi_setup(spi);
    if (err)
        return dev_err_probe(&spi->dev, err, "spi_setup failed\n");

    err = fastadc_read_word(p, &sample);
    if (err)
```

```
        return dev_err_probe(&spi->dev, err, "test read failed\n");

    dev_info(&spi->dev, "fastadc ready (test sample = %u)\n", sample);
    spi_set_drvdata(spi, p);
    return 0;
}

static void fastadc_remove(struct spi_device *spi)
{
    /* devm_* handles cleanup */
}

static const struct of_device_id fastadc_of_match[] = {
    { .compatible = "linuxlearn,fastadc" },
    { }
};
MODULE_DEVICE_TABLE(of, fastadc_of_match);

static const struct spi_device_id fastadc_id[] = {
    { "fastadc", 0 },
    { }
};
MODULE_DEVICE_TABLE(spi, fastadc_id);

static struct spi_driver fastadc_driver = {
    .driver = {
        .name = "linuxlearn-fastadc",
        .of_match_table = fastadc_of_match,
    },
    .probe    = fastadc_probe,
    .remove   = fastadc_remove,
    .id_table = fastadc_id,
};
module_spi_driver(fastadc_driver);

MODULE_LICENSE("GPL");
```

Mirror image of the I²C driver from Ch 46, same idioms: `module_spi_driver`, two match tables, `devm_kzalloc`, `dev_err_probe`.

47.4 Transfers and messages

The data model is two levels:

- **`spi_transfer`**: one back-and-forth burst. Has `tx_buf` (or NULL for read-only), `rx_buf` (or NULL for write-only), `len`, optional per-transfer overrides (`speed_hz`, `bits_per_word`, `cs_change`, `delay`).
- **`spi_message`**: a list of `spi_transfer`s executed atomically. CS asserts before the first transfer, deasserts after the last (unless overridden).

For simple sequences, **`spi_sync_transfer(spi, xfers, n)`** is a one-call wrapper that builds the `spi_message`, submits it, waits, and returns.

For a complex case, say, write a register address then read 16 bytes, with no STOP between (CS held continuously):

```
u8 tx_cmd[2] = { 0x80, reg_addr };
u8 rx_data[16];
struct spi_transfer xfers[2] = {
    {
        .tx_buf = tx_cmd,
        .len    = 2,
    },
    {
        .rx_buf = rx_data,
        .len    = 16,
    },
};
err = spi_sync_transfer(spi, xfers, 2);
```

CS is asserted for the entire 18-byte sequence. The first transfer sends 2 bytes (clocking nothing in particular into `rx`), the second reads 16 bytes (clocking dummy 0s out of `tx`).

Need to deassert CS between transfers? Set `xfers[0].cs_change = 1`. Need a delay? `xfers[0].delay.value = 10. xfers[0].delay.unit = SPI_DELAY_UNIT_USECS;.`

Async transfers

For high-rate sampling, `spi_async()` submits a message without waiting. A callback fires on completion. Drivers that sample continuously (audio codecs, fast IMUs) use this. Initialise:

```
static void my_complete(void *context)
{
    struct my_priv *p = context;
    /* runs in workqueue context after transfer finishes */
}

struct spi_message msg;
spi_message_init(&msg);
spi_message_add_tail(&xfer, &msg);
msg.complete = my_complete;
msg.context  = priv;
spi_async(spi, &msg);
```

47.5 Half-duplex helpers

For chips that don’t really use full-duplex, the helpers `spi_write`, `spi_read`, and `spi_write_then_read` are simpler than building transfers by hand:

```
err = spi_write(spi, txbuf, len);                       /* TX only, RX discarded */
err = spi_read(spi, rxbuf, len);                        /* RX only, TX = 0 */
err = spi_write_then_read(spi, txbuf, tlen, rxbuf, rlen); /* TX then RX, CS held */
```

`spi_write_then_read` is what you reach for 90% of the time. It builds two transfers internally with CS held throughout.

47.6 /dev/spidevN, user-space access

> **Template warning:** This block contains placeholder values. Replace compatible strings, GPIO numbers, addresses, and paths with values from your board before using it.

Like `/dev/i2c-N`, there's a user-space chardev for SPI: `/dev/spidev<bus>.<cs>`. To enable, add the binding to your DT:

```
&ecspi3 {
    spidev@0 {
        compatible = "rohm,dh2228fv";   /* historical placeholder for spidev */
        reg = <0>;
        spi-max-frequency = <10000000>;
    };
};
```

Why `"rohm,dh2228fv"`? The kernel maintainers will not accept `"spidev"` as a generic compatible, spidev is not a chip, just a user-space access mechanism. The `dh2228fv` is one of several chip names "registered" to spidev as a workaround for development. For production, use the actual chip's compatible string with a real driver.

> **Kernel warning since v4.15.** If you use this placeholder in a DT, `spidev_probe` prints `WARNING: Probing spidev with broken DT entry` and refuses to bind unless `CONFIG_SPI_SPIDEV` overrides are set. Modern best practice: either pick a real-chip compatible that already appears in `drivers/spi/spidev.c`'s `spidev_dt_ids[]` (e.g., `"semtech,sx1301"` for a known SPI radio), or write a proper DT overlay that adds a `compatible` your kernel build accepts. The cookbook chapters (Ch 98, 99, 101, 105, 106) use `rohm,dh2228fv` as shorthand. In production swap for the chip's real string + a tiny accepting driver, or build with the `spidev_compatible_array` patch.

The spidev userspace API uses ioctl:

```
int fd = open("/dev/spidev2.0", O_RDWR);
u8 mode = SPI_MODE_0;
ioctl(fd, SPI_IOC_WR_MODE, &mode);
u32 speed = 10000000;
ioctl(fd, SPI_IOC_WR_MAX_SPEED_HZ, &speed);

struct spi_ioc_transfer xfer = {
    .tx_buf = (uintptr_t)tx, .rx_buf = (uintptr_t)rx, .len = 4,
};
ioctl(fd, SPI_IOC_MESSAGE(1), &xfer);
```

Useful for bring-up. Production code should be a real `spi_driver`.

47.7 A worked example, driving an MCP3008 ADC

The MCP3008 is a popular SPI ADC: 10-bit, 8 channels, single-ended. Communication protocol:

- Write a 3-byte command: `0x01`, `(0x80 | channel<<4)`, `0x00`.
- Read 3 bytes back. The 10-bit value lives in `rx[1] & 0x03` (high 2 bits) and `rx[2]` (low 8 bits).

A minimal driver function:

```
static int mcp3008_read_channel(struct mcp3008_priv *p, int channel, u16 *out)
{
    u8 tx[3] = { 0x01, (u8)(0x80 | (channel << 4)), 0x00 };
    u8 rx[3];
    struct spi_transfer xfer = {
        .tx_buf = tx, .rx_buf = rx, .len = 3,
    };
    int err = spi_sync_transfer(p->spi, &xfer, 1);
    if (err < 0)
        return err;
    *out = ((rx[1] & 0x03) << 8) | rx[2];
    return 0;
}
```

Wrap with IIO (Ch 49) and you have an 8-channel ADC exposed via `/sys/bus/iio/devices/iio:device0/in_voltage*_raw`. The mainline `drivers/iio/adc/mcp320x.c` does this for the entire MCP320x family.

47.8 Key Notes

- The SPI controller owns bus scheduling and signaling, while each `spi_device` carries chip-select and configuration constraints. The chip driver owns command framing and interpretation.
- A `spi_message` groups transfers into one bus operation. Chip-select continuity, delays, and mode must match the peripheral protocol rather than merely the desired byte count.
- Full-duplex clocks transmit and receive together. Command bytes, dummy clocks, and meaningful receive bytes occupy different positions in the same exchange.
- Synchronous submission waits for completion in a sleeping context. Asynchronous submission extends the lifetime of the message, transfers, buffers, and callback state until completion.

- A maximum clock is a limit, not proof of a reliable electrical interface. Controller divisors, chip timing, wiring, and loading constrain usable speed.
- Spidev exposes raw transactions without chip semantics. A subsystem driver can instead supply a stable sensor or storage contract and centralize ownership, power management, and error handling.

47.9 Pitfalls

- **Wrong CS polarity.** If a chip wants active-high CS, the controller's default (active-low) won't drive it correctly. Use `spi-cs-high` in DT.
- **CPOL/CPHA wrong.** Symptom: reads return 0xFF or junk. Cross-check the chip's datasheet *mode* against your DT. Mode 0 is most common. Mode 3 next.
- **Speed too high for layout.** Long traces, missing termination, no ground reference plane → garbage at 20 MHz that works fine at 1 MHz. Start slow, ramp up.
- **Sending a single read transfer with `len` larger than your `rx_buf`.** Buffer overflow → kernel panic. The `len` field is the SPI clock count, so you need at least that many bytes in `rx_buf`.
- **Calling `spi_sync_transfer` from atomic context.** It sleeps. Use `spi_async` from atomic context, or defer to a workqueue.
- **Native CS vs GPIO-CS subtleties.** The i.MX eCSPI native CS asserts and deasserts for *each* `spi_transfer`. To hold CS across multiple transfers, either (a) put them all in one `spi_message`, or (b) use GPIO-based CS via `cs-gpios`, software holds GPIO-CS for the whole message.
- **`bits_per_word` != 8.** Most chips use 8-bit words. If you set 16, the controller packs two bytes per "word", and the byte order may not be what you expect. Stay at 8 unless you have a reason.
- **Forgetting `spi_setup` after changing mode/speed.** Changes to `spi->mode`, `spi->bits_per_word`, etc., don't take effect until you call `spi_setup`. Probe should always call it once.

47.10 Going deeper

- **`Documentation/spi/`**: SPI subsystem documentation.
- **`Documentation/devicetree/bindings/spi/`**: DT bindings for SPI controllers and devices.
- **`drivers/spi/spi-imx.c`**: the i.MX SPI controller driver.
- **`drivers/iio/adc/mcp320x.c`**: production MCP3008 driver as a clean IIO reference.
- **`drivers/spi/spidev.c`**: the spidev chardev driver. How `/dev/spidevN` is implemented.

- **`Documentation/spi/spidev.rst`**: when/how to use spidev (and the controversial `dh2228fv` workaround explained).

> Next chapter: **Chapter 48: PWM and RTC.** Two short but practical subsystems for ubiquitous embedded needs: dimming backlights, generating tones, and keeping time across power cycles. **MCU bridge:** Think of Linux PWM like an MCU timer output channel, except the driver exposes period, duty cycle, polarity, and enable state through a subsystem. **PWM:** Pulse-Width Modulation, a timer output whose duty cycle controls average power or encodes timing.

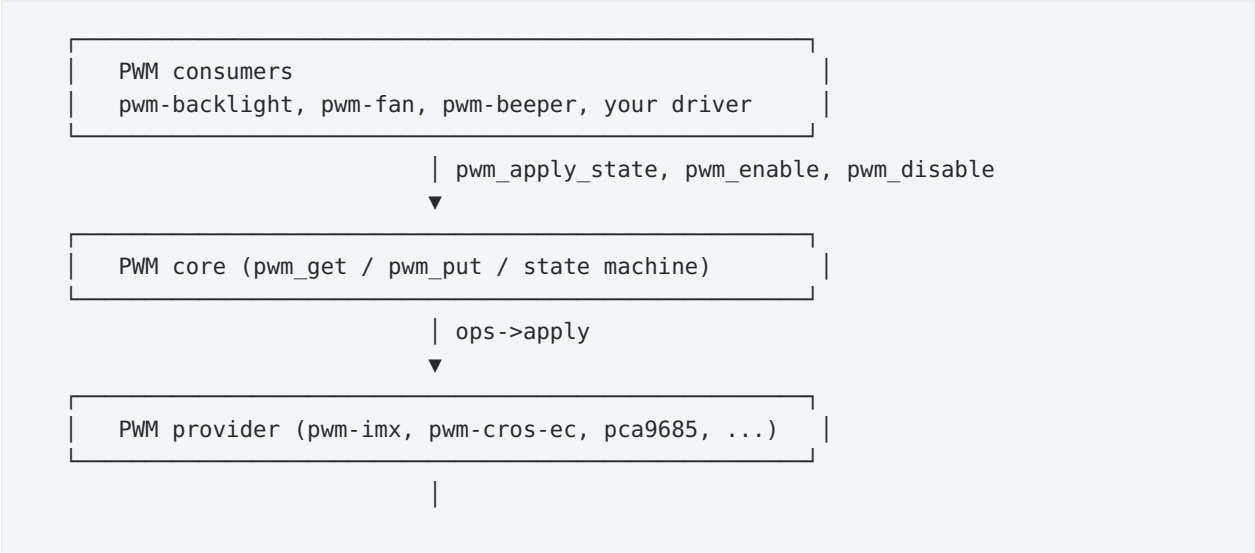
Chapter 48: PWM and RTC subsystems

> **MMIO:** memory-mapped I/O, where software accesses peripheral registers through normal load and store instructions.

> **What:** two short, unrelated subsystems combined here, each is small enough on its own, and the patterns reinforce each other. **PWM**, the `pwm_*` API and the `pwm-backlight` / `pwm-fan` / `pwm-beeper` consumers. **RTC**, the `rtc_class` framework, sysfs `/sys/class/rtc/`, the `hwclock` user-space tool, and how an external RTC chip plugs into Linux's wall-clock time. **PWM:** Pulse-Width Modulation, a timer output whose duty cycle controls average power or encodes timing. **sysfs:** a kernel-generated filesystem under /sys that exposes devices, drivers, and attributes.
>
> **Why:** *backlight dimming, fan speed, audible beeper, servo control* all use PWM, and every product that doesn't have continuous network access needs an RTC to keep time across reboots. These are subsystems you'll touch on almost every embedded project. Knowing the consumer-side API saves you from re-inventing it.
>
> **Focus: consumer vs provider model**. PWM and RTC both expose two APIs: one for the *producer* (chip driver that owns the PWM controller or RTC silicon) and one for the *consumer* (driver/code that wants a PWM signal or a wall-clock read). You almost always write *consumers*. The SoC vendor wrote the producers. Knowing which side you're on tells you which API to look up.

48.1 PWM subsystem

48.1.1 Architecture



```
                              ▼ MMIO / I²C / SPI
                            hardware
```

The provider talks to the hardware. The consumer asks for a `period` and `duty_cycle`. The core mediates and enforces invariants (e.g., duty ≤ period, period ≤ chip max).

i.MX6ULL has 8 PWM channels (PWM1–PWM8). The mainline `pwm-imx27` driver covers them.

48.1.2 DT representation

In the SoC DT:

```
&pwm1 {
    pinctrl-names = "default";
    pinctrl-0 = <&pinctrl_pwm1>;
    status = "okay";
};
```

Consumer side, e.g., a backlight on PWM1:

```
backlight: backlight {
    compatible = "pwm-backlight";
    pwms = <&pwm1 0 5000000 0>;
    /*       ^   ^   ^         ^
             |   |   |         flags (0 = normal polarity)
             |   |   period in ns (5 ms = 200 Hz)
             |   channel 0 of this PWM controller
             phandle to pwm1
     */
    brightness-levels = <0 4 8 16 32 64 128 255>;
    default-brightness-level = <5>;
    power-supply = <&reg_lcd_3v3>;
};
```

The `pwms` property is the binding. It's analogous to `gpios`, references the controller phandle, channel number within the controller, the period in nanoseconds, and flags. The consumer ("pwm-backlight") looks it up with `devm_pwm_get(&pdev->dev, NULL)`.

48.1.3 Consumer API

```
#include <linux/pwm.h>
```

```
struct pwm_device *pwm = devm_pwm_get(&pdev->dev, NULL);
if (IS_ERR(pwm))
    return PTR_ERR(pwm);

struct pwm_state state = {
    .period = 5000000,           /* 5 ms = 200 Hz */
    .duty_cycle = 2500000,       /* 50 % */
    .polarity = PWM_POLARITY_NORMAL,
    .enabled = true,
```

```
};
err = pwm_apply_state(pwm, &state);
```

That’s the whole API for static use. For dynamic adjustment:

```
/* Adjust brightness; period stays fixed */
pwm_get_state(pwm, &state);
state.duty_cycle = (state.period * percent) / 100;
pwm_apply_state(pwm, &state);
```

Convenience helpers for the most common case:

```
pwm_config(pwm, duty_ns, period_ns);    /* set duty + period */
pwm_enable(pwm);
pwm_disable(pwm);
```

`pwm_config` / `pwm_enable` are wrappers over `pwm_apply_state`. New code prefers the explicit state struct. Legacy code uses the simpler form.

48.1.4 Built-in consumer drivers

You almost never need to write a custom PWM consumer. Use the in-tree generics:

Driver	Binding	Use case
`pwm-backlight`	`compatible = "pwm-backlight"`	LCD backlight
`pwm-fan`	`compatible = "pwm-fan"`	Cooling fan
`pwm-beeper`	`compatible = "pwm-beeper"`	Active or passive beeper
`pwm-vibrator`	`compatible = "pwm-vibrator"`	Haptic motor
`pwm-leds`	`compatible = "pwm-leds"`	LED brightness via PWM
`pwm-ir-tx`	`compatible = "pwm-ir-tx"`	IR transmitter

Pick the one that matches and configure via DT. The generic drivers expose user-space interfaces (e.g., `/sys/class/backlight/`, `/sys/class/leds/`).

48.1.5 /sys/class/pwm, sysfs access

For prototyping, export PWMs to user-space:

```
[root@pa-mini:~]# ls /sys/class/pwm/
pwmchip0  pwmchip1  pwmchip2  pwmchip3  pwmchip4  pwmchip5  pwmchip6  pwmchip7

[root@pa-mini:~]# cd /sys/class/pwm/pwmchip0
[root@pa-mini:~]# echo 0 > export       ← claim channel 0
[root@pa-mini:~]# cd pwm0
[root@pa-mini:~]# echo 1000000 > period
[root@pa-mini:~]# echo 500000  > duty_cycle    ← 50 %
[root@pa-mini:~]# echo 1 > enable
```

A 1 kHz, 50 % duty PWM is now on the corresponding pin. Useful for quick bring-up. Production drivers should use the consumer API.

48.2 RTC subsystem

48.2.1 Two RTCs to know about

Most boards have *two* sources of timekeeping:

1. **The SoC's internal RTC.** On i.MX6ULL this is the **SNVS_LP** block, a low-power domain with its own RTC, optionally backed by a coin-cell battery on the `VBAT` pin. The mainline driver is `rtc-snvs`. The internal RTC is "free", no extra BOM, but has the SoC's main XTAL accuracy (~50 ppm uncompensated).
2. **An external RTC chip.** DS3231 (TCXO, ±2 ppm), PCF8563, MCP79410, etc. Better accuracy, battery-backed, talks I²C or SPI. Mainline drivers in `drivers/rtc/`.

Boards often have one, sometimes both. The kernel uses **the first registered RTC as `/dev/rtc0`** and exposes the rest as `rtc1`, `rtc2`, etc. The Real-Time Clock that *systemd* / *busybox-init* read is `/dev/rtc0` by default, choose carefully.

48.2.2 DT for an external RTC

DS3231 on I²C:

```
&i2c1 {
    rtc@68 {
        compatible = "maxim,ds3231", "dallas,ds1307";
        reg = <0x68>;
        interrupt-parent = <&gpio1>;
        interrupts = <22 IRQ_TYPE_EDGE_FALLING>;
        wakeup-source;
    };
};
```

The `wakeup-source` flag lets the RTC's alarm wake the SoC from suspend (Ch 51B).

Once probed, `/dev/rtc1` (or whichever number) appears, and `/sys/class/rtc/rtcN/` exposes attributes.

48.2.3 rtc_class consumer API

For *most* drivers, you don't talk to the RTC directly. User-space's `hwclock` reads `/dev/rtcN`. The kernel's time-of-day comes from the RTC at boot (`hctosys`).

If you do need to read or set time from a kernel driver:

```
#include <linux/rtc.h>

struct rtc_device *rtc = rtc_class_open("rtc0");
if (!rtc) return -ENODEV;

struct rtc_time tm;
err = rtc_read_time(rtc, &tm);     /* tm is in UTC */
pr_info("now: %d-%02d-%02d %02d:%02d:%02d\n",
        tm.tm_year + 1900, tm.tm_mon + 1, tm.tm_mday,
        tm.tm_hour, tm.tm_min, tm.tm_sec);

rtc_class_close(rtc);
```

For alarms (e.g., wake at a specific time):

```
struct rtc_wkalrm alm = {
    .enabled = 1,
    .time = future_tm,
};
rtc_set_alarm(rtc, &alm);
```

The kernel coordinates with PM to use the RTC alarm as the wakeup source.

48.2.4 RTC provider (driver), quick sketch

If you ever do need to write an RTC driver (e.g., for a chip without a mainline driver), the shape is:

```
static int my_rtc_read_time(struct device *dev, struct rtc_time *tm) { ... }
static int my_rtc_set_time(struct device *dev, struct rtc_time *tm) { ... }
static int my_rtc_alarm_irq_enable(struct device *dev, unsigned int enabled) { ... }

static const struct rtc_class_ops my_rtc_ops = {
    .read_time   = my_rtc_read_time,
    .set_time    = my_rtc_set_time,
    .alarm_irq_enable = my_rtc_alarm_irq_enable,
    /* ... read_alarm, set_alarm, etc */
};

/* In probe: */
struct rtc_device *rtc = devm_rtc_allocate_device(&pdev->dev);
```

```
if (IS_ERR(rtc))
    return PTR_ERR(rtc);
rtc->ops = &my_rtc_ops;
rtc->range_min = RTC_TIMESTAMP_BEGIN_2000;
rtc->range_max = RTC_TIMESTAMP_END_2099;
err = devm_rtc_register_device(rtc);
```

The core handles `/dev/rtcN` and `/sys/class/rtc/` for you.

48.2.5 User-space, hwclock and timedatectl

```
# Read RTC into system time at boot (standard init scripts do this automatically)
[root@pa-mini:~]# hwclock -s

# Set RTC from system time
[root@pa-mini:~]# date -s "2026-05-26 14:30:00"
[root@pa-mini:~]# hwclock -w

# Read RTC directly
[root@pa-mini:~]# hwclock
2026-05-26 14:30:42.123456+00:00

# systemd-aware tooling
[root@pa-mini:~]# timedatectl
                Local time: Tue 2026-05-26 14:30:42 UTC
            Universal time: Tue 2026-05-26 14:30:42 UTC
                  RTC time: Tue 2026-05-26 14:30:42
                 Time zone: Etc/UTC
 System clock synchronized: yes
               NTP service: active
```

For a fleet product, run `chrony` or `systemd-timesyncd` to sync system time to NTP. Then write the RTC periodically, via systemd's `systemd-time-sync-target`, or an init `-11` hook.

48.2.6 Alarms for wake-from-suspend

Combine the RTC with PM (Ch 51B) to wake the system at a specific time:

```
# Wake the system 60 seconds from now
$ echo $(($(date +%s) + 60)) > /sys/class/rtc/rtc0/wakealarm
$ echo mem > /sys/power/state         ← suspend
# … 60 seconds later, system resumes ...
```

This is the foundation of low-power data-logger products: sleep deeply, wake on RTC, sample sensors, log, sleep again.

48.3 Key Notes

- A PWM provider implements hardware timing, while a consumer owns the signal’s purpose. Backlight, fan, and beeper frameworks add role-specific policy above raw period and duty cycle.
- Period, duty cycle, polarity, and enable state are one configuration contract. Hardware resolution and electrical behavior constrain what a requested state can achieve.
- One owner controls a PWM channel at a time. Safe inactive levels and suspend behavior belong to the consumer’s design, not merely to resource allocation.
- An RTC preserves wall-clock information, while the running system maintains its own clock. Boot-time selection and synchronization policy determine which source is trusted.
- An external RTC is optional hardware, and backup power determines whether time survives power loss. Register access alone does not establish accuracy or persistence.
- Wake alarms require both RTC alarm support and a working wake route through power management. An alarm setting does not by itself demonstrate suspend/resume behavior on the MINI.

48.4 Pitfalls

- **PWM period too short.** The provider’s hardware has a max-period and a frequency resolution. Asking for 1 ns period or 1 Hz frequency may snap to the nearest achievable value silently. Always read back the actual state with `pwm_get_state`.
- **Polarity inversion forgotten.** Some backlights are active-low (full brightness = 0% duty). Use `PWM_POLARITY_INVERTED` in `pwm_state` or `pwms = <..., PWM_POLARITY_INVERTED>` in DT.
- **PWM stops when consumer driver unloads.** `pwm_put` (or `devm_*` cleanup) disables the PWM. If you want the signal to keep running after unload, that’s a design choice you must explicitly handle.
- **Multiple consumers fighting over one PWM.** Only one consumer per PWM. Verify with `/sys/class/pwm/pwmchipN/pwmN/`.
- **RTC time-zone confusion.** RTC by convention stores UTC. Some legacy systems store local time. `timedatectl set-local-rtc 0` to enforce UTC.
- **No backup battery on SNVS_LP.** SoC’s internal RTC loses time on power-loss without `VBAT`. Symptom: every reboot starts in 1970. Wire up a CR2032. If you cannot, accept the limit and sync via NTP at boot.

- **Multiple RTCs, hctosys reads the wrong one.** `CONFIG_RTC_HCTOSYS_DEVICE="rtc0"` (default) picks the first registered. If you have both SoC RTC (registers first) and DS3231 (registers later, more accurate), you get the wrong one. Either rename via udev or change kernel config.

> **udev:** the user-space device manager that reacts to kernel device events and creates policy-driven /dev nodes.

- **DS3231 alarm-mask register quirk.** The alarm fires for the first match across multiple fields. Misconfiguring the mask gives a once-per-second wake instead of once-per-day. Read the datasheet carefully.

48.5 Going deeper

- **`Documentation/pwm.rst`**: the PWM subsystem documentation.
- **`Documentation/admin-guide/rtc.rst`**: RTC subsystem and `hwclock`.
- **`drivers/pwm/pwm-imx27.c`**: i.MX PWM driver. Small and clean.
- **`drivers/rtc/rtc-snvs.c`**: i.MX SNVS RTC driver.
- **`drivers/rtc/rtc-ds1307.c`**: handles DS1307, DS1338, DS1340, DS3231, DS3232, MCP7940x, one driver for the whole family. A good reference for handling chip-family variants.
- **`Documentation/devicetree/bindings/pwm/`** and **`/rtc/`**, DT bindings.

> Next chapter: **Chapter 49: IIO subsystem.** ADCs, DACs, light/temp/pressure/IMU sensors, they all live in IIO, the “Industrial I/O” subsystem. Once you internalise IIO, every sensor in Part VII’s cookbook becomes “DT + driver registers channels + user-space reads /sys/bus/iio/devices/.” **IIO:** Industrial I/O, Linux’s subsystem for sensors, ADCs, DACs, and buffered sampled data.

Chapter 49: IIO subsystem (ADCs, sensors)

> **What: Industrial I/O** (IIO), the kernel framework that everything sensor-related lives in. ADCs, DACs, temperature/humidity/pressure/light/proximity sensors, IMUs (accel + gyro + mag), color sensors, particulate-matter sensors, current sensors, all expose themselves through one consistent API: `/sys/bus/iio/devices/iio:deviceN/in_<type>_<index>_raw` for one-shot reads, `/dev/iio:deviceN` for streamed buffers. **IIO:** Industrial I/O, Linux's subsystem for sensors, ADCs, DACs, and buffered sampled data.
>
> **Why:** before IIO (~2011), every sensor driver invented its own sysfs layout. Reading an ADXL345 was completely different from reading an LIS3DH despite both being 3-axis accelerometers. IIO standardised the interface: every accelerometer reports `in_accel_x_raw` in the same units after `_scale` is applied. User-space tools (`iio-utils`, libiio, gnuplot wrappers) work generically. **Every chip in Part VII's sensor cookbook is an IIO driver. sysfs:** a kernel-generated filesystem under /sys that exposes devices, drivers, and attributes.
>
> **Focus: channels, scale, and triggers**. A *channel* is one measurable thing (accel-x, temp, ADC-in-3). A *scale* converts raw value to engineering units. A *trigger* is what causes a coordinated sample to be taken (a timer, an IRQ, a sysfs poke). Get those three concepts right and the rest of IIO follows. **MCU bridge:** Think of an IRQ like an EXTI/NVIC interrupt path, except Linux splits the hard interrupt from deferred work and must share lines across drivers. **IRQ:** interrupt request, the signal path that tells the CPU or interrupt controller that hardware needs service.

49.1 Architecture



```
        ▼
    hardware
```

Drivers declare a list of `iio_chan_spec` (channel specifications) and provide `read_raw` / `write_raw` callbacks. The core handles user-space exposure.

49.2 The channel, IIO's atom

An IIO channel describes *one measurable quantity*. Examples:

Channel type	Direction	Modifier	Common index	Sysfs file
`IIO_VOLTAGE`	input	(none)	0..7	`in_voltage0_raw`
`IIO_ACCEL`	input	X/Y/Z	,	`in_accel_x_raw`
`IIO_TEMP`	input	(none)	,	`in_temp_raw`
`IIO_HUMIDITYRELATIVE`	input	(none)	,	`in_humidityrelative_raw`
`IIO_PRESSURE`	input	(none)	,	`in_pressure_raw`
`IIO_LIGHT`	input	(none)	,	`in_illuminance_raw`
`IIO_PROXIMITY`	input	(none)	,	`in_proximity_raw`
`IIO_INTENSITY`	input	`IR / BOTH / RED / GREEN / BLUE`	,	`in_intensity_ir_raw`
`IIO_ANGL_VEL`	input	X/Y/Z	,	`in_anglvel_x_raw`
`IIO_MAGN`	input	X/Y/Z	,	`in_magn_x_raw`
`IIO_VOLTAGE`	output	(none)	0..N	`out_voltage0_raw` (DAC)

A channel reports a *raw* integer plus a *scale* and (optionally) an *offset*:

```
real_value_in_SI_units = (raw + offset) × scale
```

For example, a temperature sensor might report `in_temp_raw = 25420`, `in_temp_scale = 0.001`, `in_temp_offset = -2048`. The cooked value is `(25420 + (-2048)) × 0.001 = 23.372` °C.

User-space reads either:

- The raw value and computes itself: `cat in_temp_raw → 25420. cat in_temp_scale → 0.001. ....`
- A processed value if available: `cat in_temp_input → 23372` (in milli-units).

49.3 Defining channels in a driver

```
#include <linux/iio/iio.h>

static const struct iio_chan_spec my_channels[] = {
    {
        .type = IIO_TEMP,
        .info_mask_separate = BIT(IIO_CHAN_INFO_RAW) |
                              BIT(IIO_CHAN_INFO_SCALE) |
                              BIT(IIO_CHAN_INFO_OFFSET),
    },
    {
        .type = IIO_PRESSURE,
        .info_mask_separate = BIT(IIO_CHAN_INFO_RAW) | BIT(IIO_CHAN_INFO_SCALE),
    },
    {
        .type = IIO_HUMIDITYRELATIVE,
        .info_mask_separate = BIT(IIO_CHAN_INFO_RAW) | BIT(IIO_CHAN_INFO_SCALE),
    },
};
```

Three channels: temp, pressure, humidity. `info_mask_separate` declares which per-channel attributes exist (raw, scale, offset). The core auto-creates `in_temp_raw`, `in_temp_scale`, `in_temp_offset`, etc.

The driver's `read_raw` callback dispatches by channel and `info`:

```
static int my_read_raw(struct iio_dev *idev,
                       struct iio_chan_spec const *chan,
                       int *val, int *val2, long info)
{
    struct my_priv *p = iio_priv(idev);

    switch (info) {
    case IIO_CHAN_INFO_RAW:
        switch (chan->type) {
        case IIO_TEMP:
```

```
            *val = read_temp_raw(p);  /* a raw integer from the chip */
            return IIO_VAL_INT;
        case IIO_PRESSURE:
            *val = read_pressure_raw(p);
            return IIO_VAL_INT;
        case IIO_HUMIDITYRELATIVE:
            *val = read_hum_raw(p);
            return IIO_VAL_INT;
        default:
            return -EINVAL;
        }

    case IIO_CHAN_INFO_SCALE:
        switch (chan->type) {
        case IIO_TEMP:
            *val = 0; *val2 = 10000;     /* 0.01 °C per raw */
            return IIO_VAL_INT_PLUS_MICRO;
        /* ... pressure, hum ... */
        }

    case IIO_CHAN_INFO_OFFSET:
        if (chan->type == IIO_TEMP) {
            *val = -2048;
            return IIO_VAL_INT;
        }
        return -EINVAL;
    }

    return -EINVAL;
}

static const struct iio_info my_iio_info = {
    .read_raw = my_read_raw,
};
```

Return values:

- `IIO_VAL_INT`: `*val` holds an integer.
- `IIO_VAL_INT_PLUS_MICRO`: value is `*val + *val2/1000000`. Used for fractional scales.
- `IIO_VAL_INT_PLUS_NANO`: `*val + *val2/1e9`. For very precise scales (e.g., picoteslas).
- `IIO_VAL_FRACTIONAL_LOG2`: `*val / 2^*val2`. Common for ADC scales like 1/4096.

49.4 Probing and registering

```
static int my_probe(struct i2c_client *client)
{
    struct iio_dev *idev;
    struct my_priv *p;
    int err;

    idev = devm_iio_device_alloc(&client->dev, sizeof(*p));
    if (!idev)
        return -ENOMEM;
```

```
    p = iio_priv(idev);
    p->client = client;
    i2c_set_clientdata(client, idev);

    idev->name = "mysensor";
    idev->info = &my_iio_info;
    idev->modes = INDIO_DIRECT_MODE;
    idev->channels = my_channels;
    idev->num_channels = ARRAY_SIZE(my_channels);

    err = devm_iio_device_register(&client->dev, idev);
    if (err)
        return dev_err_probe(&client->dev, err, "iio register failed\n");

    dev_info(&client->dev, "mysensor ready\n");
    return 0;
}
```

`devm_iio_device_alloc(&client->dev, sizeof(*p))` allocates both the `iio_dev` and your private struct in one block. `iio_priv(idev)` recovers the priv pointer.

`INDIO_DIRECT_MODE` is the simple mode, user-space `cat in_temp_raw` directly invokes your `read_raw`. The advanced mode (`INDIO_BUFFER_HARDWARE` etc.) enables high-rate buffered capture with triggers.

49.5 User-space sees this

```
[root@pa-mini:~]# ls /sys/bus/iio/devices/iio:device0/
in_humidityrelative_raw   in_pressure_raw     in_temp_offset
in_humidityrelative_scale in_pressure_scale   in_temp_raw
                                              in_temp_scale
name                      ...

[root@pa-mini:~]# cat /sys/bus/iio/devices/iio:device0/name
mysensor

[root@pa-mini:~]# cat /sys/bus/iio/devices/iio:device0/in_temp_raw
25420

[root@pa-mini:~]# cat /sys/bus/iio/devices/iio:device0/in_temp_scale
0.010000
```

To convert: `awk 'BEGIN{print (25420 + (-2048)) * 0.01}'` → `23.372` °C. Or use a tool that knows IIO conventions:

```
[root@pa-mini:~]# iio_attr -d iio:device0 in_temp_raw
in_temp_raw: 25420
[root@pa-mini:~]# iio_readdev -t mysensor in_temp
23.372 °C
```

49.6 Triggers and buffered capture

For high-rate sensors (IMU at 1 kHz, ADC at 100 kHz), per-sample sysfs reads are too slow. IIO's *buffer* infrastructure lets the driver push samples into a kfifo, and user-space reads them as a stream from `/dev/iio:deviceN`.

The orchestration:

1. Driver registers a **buffer** (`devm_iio_kfifo_buffer_setup`).
2. User-space writes `1` to `scan_elements/in_<channel>_en` for each channel to enable (e.g., `in_accel_x_en`, `in_accel_y_en`).
3. User-space writes `buffer/length` to set the kfifo depth.
4. User-space binds a **trigger** via `trigger/current_trigger`, typically `hrtimer-N` (a kernel high-resolution timer firing at a set rate).
5. User-space writes `buffer/enable = 1`.
6. The trigger fires. The driver's trigger handler reads a coordinated set of samples and pushes them.
7. User-space `read(/dev/iio:deviceN, ...)` returns the bytes.

```
[root@pa-mini:~]# echo 1 > /sys/bus/iio/devices/iio:device0/scan_elements/in_accel_x_en
[root@pa-mini:~]# echo 1 > /sys/bus/iio/devices/iio:device0/scan_elements/in_accel_y_en
[root@pa-mini:~]# echo 1 > /sys/bus/iio/devices/iio:device0/scan_elements/in_accel_z_en
[root@pa-mini:~]# echo 1024 > /sys/bus/iio/devices/iio:device0/buffer/length
[root@pa-mini:~]# echo "hrtimer-0" > /sys/bus/iio/devices/iio:device0/trigger/
current_trigger
[root@pa-mini:~]# echo 1 > /sys/bus/iio/devices/iio:device0/buffer/enable
[root@pa-mini:~]# dd if=/dev/iio:device0 of=samples.bin bs=$((6*100)) count=10
```

Six bytes per sample (3 axes × 2 bytes), 100 samples per read, 10 reads. The trigger drives the cadence. The driver pushes. User-space drains.

We come back to triggers and buffers in Ch 70/71 (IMUs), where they become essential.

49.7 ADC drivers, a special case

ADCs work the same as sensors but with `IIO_VOLTAGE` channels and an `_indexed` flag so multiple channels can be numbered 0..7:

```
#define ADC_CHANNEL(idx) {                                \
    .type = IIO_VOLTAGE,                                   \
    .indexed = 1,                                          \
    .channel = (idx),                                      \
    .info_mask_separate = BIT(IIO_CHAN_INFO_RAW),          \
    .info_mask_shared_by_type = BIT(IIO_CHAN_INFO_SCALE),\
    .scan_index = (idx),                                   \
```

```
    .scan_type = {                                          \
        .sign = 'u', .realbits = 10, .storagebits = 16,  \
    },                                                      \
}

static const struct iio_chan_spec mcp3008_channels[] = {
    ADC_CHANNEL(0), ADC_CHANNEL(1), ADC_CHANNEL(2), ADC_CHANNEL(3),
    ADC_CHANNEL(4), ADC_CHANNEL(5), ADC_CHANNEL(6), ADC_CHANNEL(7),
};
```

`indexed = 1` causes the sysfs files to be named `in_voltage0_raw`, `in_voltage1_raw`, etc. `info_mask_shared_by_type` means the scale is shared (one `in_voltage_scale` for all channels), common for ADCs where every channel has the same reference voltage.

49.8 Key Notes

- Channels describe measurable quantities and their attributes. The sensor driver supplies acquisition and conversion semantics, while IIO supplies a common user-space interface.
- Raw values, offset, scale, and the `IIO_VAL_*` representation form a unit-conversion contract. Plausible numbers do not establish correct units or calibration.
- Direct reads suit occasional observations. Triggered buffers separate acquisition cadence from user-space scheduling and reduce per-sample interface overhead.
- Buffered records are a binary contract defined by enabled channels, scan order, storage width, signedness, endianness, and alignment. They are not simply a concatenation of displayed sysfs values.
- A trigger requests acquisition but does not guarantee the sensor has a new sample or that several axes were captured simultaneously. The hardware sampling model constrains coherence.
- Buffer enable/disable, direct access, interrupts, and removal must coordinate device state. Slow consumers and finite queues require an explicit overflow or loss policy.

49.9 Pitfalls

> **Security boundary:** Do not burn fuses, enroll production keys, or sign release images when exploring example code. Use throwaway keys and back up the unsigned image plus the key directory before testing irreversible security flows.

- **Wrong `realbits` / `storagebits` in `scan_type`.** Buffered reads return junk. User-space can’t decode. For a 10-bit signed value stored in 16 bits, `sign='s', realbits=10, storagebits=16, shift=0`.

- **Forgetting `INDIO_DIRECT_MODE`.** A driver with no `modes` set behaves as “buffer-only”. Sysfs `_raw` reads return -EBUSY when no trigger is active. For sensors you want polled, always set `INDIO_DIRECT_MODE`.
- **`indexed=1` vs `indexed=0`.** Without `indexed=1`, the sysfs file is `in_voltage_raw` (singular). With it and `.channel = 3`, it’s `in_voltage3_raw`. Use indexed for multi-channel ADCs.
- **Returning the wrong `IIO_VAL_*` from `read_raw`.** User-space sees a non-numeric “0.000000” or garbled value. Cross-check the value type with what you store in `*val`/`*val2`.
- **Trigger / channel ordering mismatch.** With buffered capture, the bytes in `/dev/iio:deviceN` are packed in the order of `scan_index`. Make sure your driver pushes in that order.
- **Forgetting `iio_priv()`.** Recovering your private struct via `idev->dev.parent_data` or whatever doesn’t work. Use `iio_priv(idev)`. The alignment of priv data is also handled correctly only via the `_alloc(sizeof(priv))` form.
- **Two drivers competing for the same DT node.** Sometimes both `hwmon` and IIO drivers exist for the same chip. IIO is the modern choice. Pick one and disable the other in kconfig.

49.10 Going deeper

- **`Documentation/iio/`**: the IIO subsystem documentation.
- **`drivers/iio/`**: hundreds of drivers, mostly straightforward. Best ones to read: `pressure/bmp280-i2c.c` + `bmp280-core.c` (BME280 family), `imu/mpu6050/`, `adc/mcp320x.c`.
- **`Documentation/ABI/testing/sysfs-bus-iio*`**, official sysfs ABI for each channel/info combo.

> **ABI:** Application Binary Interface: the calling convention, register use, binary format, and library contract that let separately built code run together.

- **`libiio`** at https://github.com/analogdevicesinc/libiio, high-level user-space library. Bindings for Python, C++, etc.
- **`Documentation/iio/iio_configfs.rst`**: how to add a software-only IIO device (useful for trigger configuration).

> Next chapter: **Chapter 50: regmap.** Almost every chip with registers is now talked to via the regmap abstraction. Once you know regmap, writing the I²C / SPI / MMIO half of a driver becomes mechanical, you declare a register layout and the framework handles the rest. **MCU bridge:** Think of regmap like a typed wrapper

> around your read_reg() and write_reg() helpers, with caching, locking, and bus differences handled centrally. **MMIO:** memory-mapped I/O, where software accesses peripheral registers through normal load and store instructions. **regmap:** a kernel helper that wraps register reads and writes over I2C, SPI, or MMIO.

Chapter 50: regmap

> **What: regmap**, the register-access layer that sits between your driver and the bus (I²C, SPI, MMIO, or a custom one). You describe the chip's register layout once. Regmap gives you `regmap_read(rm, reg, &val)` and `regmap_write(rm, reg, val)` that just work, with optional caching, locking, debugging, and bulk transfers all handled for you. **MMIO:** memory-mapped I/O, where software accesses peripheral registers through normal load and store instructions. **regmap:** a kernel helper that wraps register reads and writes over I2C, SPI, or MMIO.
>
> **Why:** before regmap (~2011), every driver duplicated the same boilerplate: an I²C wrapper, an SPI wrapper, register-cache invalidation, byte-swap dances, mutex protection. The same 40 lines were copy-pasted across hundreds of drivers, with subtle bugs each time. Regmap factored it out. A modern driver, especially an audio codec or PMIC with hundreds of registers, uses regmap exclusively and is half as long as it would have been pre-regmap. **MCU bridge:** Think of a PMIC like a programmable power-tree supervisor: it replaces discrete enables and LDO assumptions with sequenced rails the kernel can model. **PMIC:** Power Management IC, a chip that sequences and regulates the board's voltage rails.
>
> **Focus: declare-then-use**. You provide a `regmap_config` describing your chip's registers (bit widths, ranges, which are volatile vs cached, which are read-only) and a one-call regmap_init for your bus. From there every register access goes through the same two functions. Get the config right and the rest is bookkeeping.

50.1 Why regmap exists

A typical pre-regmap driver had code like this:

```
/* Read register `reg` over I²C */
static int my_i2c_read(struct my_priv *p, u8 reg, u8 *val)
{
    struct i2c_msg msgs[2] = {
        { .addr = p->client->addr, .flags = 0,        .len = 1, .buf = &reg },
        { .addr = p->client->addr, .flags = I2C_M_RD, .len = 1, .buf = val  },
    };
    int n = i2c_transfer(p->client->adapter, msgs, 2);
    return n == 2 ? 0 : -EIO;
}

static int my_i2c_write(struct my_priv *p, u8 reg, u8 val)
{
    u8 tx[2] = { reg, val };
    return i2c_master_send(p->client, tx, 2) == 2 ? 0 : -EIO;
}
```

```
/* Now repeat for SPI, with mode 0, MSB first, etc., for the SPI variant of the chip...
*/
```

…repeated for every chip. Then you'd add:

- A mutex around register accesses (otherwise two threads racing the bus = corruption).
- A cache of writeable registers so you don't re-read after every write.
- Bulk-write support to set 4 registers in one transaction.
- Bit-field helpers to modify a single bit of a register.
- Endianness handling for chips with mixed-width registers.

That is a hundred lines of identical-looking code. Regmap factors it out: declare *what your chip looks like*, and regmap handles *how to talk to it*.

50.2 The minimal regmap

For a typical I²C chip with 8-bit registers and 8-bit register addresses:

```
#include <linux/regmap.h>

static const struct regmap_config my_regmap_config = {
    .reg_bits = 8,
    .val_bits = 8,
    .max_register = 0xFF,
};

static int my_probe(struct i2c_client *client)
{
    struct regmap *regmap;
    unsigned int val;
    int err;

    regmap = devm_regmap_init_i2c(client, &my_regmap_config);
    if (IS_ERR(regmap))
        return PTR_ERR(regmap);

    err = regmap_read(regmap, 0xD0, &val);
    if (err)
        return err;
    dev_info(&client->dev, "chip-id = 0x%02x\n", val);

    err = regmap_write(regmap, 0xF4, 0x27);    /* configure */
    if (err)
        return err;

    return 0;
}
```

That's the whole pattern. `devm_regmap_init_i2c(client, &config)` creates a regmap bound to this I²C client. `regmap_read` / `regmap_write` do all the bus dancing.

For SPI, the only change is the init call:

```
regmap = devm_regmap_init_spi(spi, &my_regmap_config);
```

For MMIO (a memory-mapped peripheral on the SoC):

```
regmap = devm_regmap_init_mmio(&pdev->dev, base, &my_regmap_config);
```

The rest of the driver, `regmap_read`, `regmap_write`, `regmap_update_bits`, is *identical*. The driver no longer cares which bus it sits on.

50.3 Variations of the config

Real chips have quirks. The config struct accommodates them.

Different register widths

```
.reg_bits = 16,           /* 16-bit register address */
.val_bits = 16,           /* 16-bit value */
.reg_format_endian = REGMAP_ENDIAN_BIG,
.val_format_endian = REGMAP_ENDIAN_BIG,
```

Audio codecs commonly have 16-bit register space and 16-bit values, big-endian wire format.

SPI command bits

Many SPI chips encode read/write in the top bit of the register address:

```
.reg_bits = 8,
.val_bits = 8,
.read_flag_mask  = 0x80,    /* OR'd into reg for reads */
.write_flag_mask = 0x00,    /* OR'd into reg for writes */
```

When you call `regmap_read(rm, 0x40, &val)`, regmap actually sends `0xC0` on the wire (0x80 | 0x40). You don't think about it.

Volatile / writable / readable ranges

Not every register is the same. ID registers are read-only. Status registers change without you writing. Some addresses are reserved. Regmap can cache writeable, non-volatile registers, saving bus traffic.

```
static bool my_readable(struct device *dev, unsigned int reg)
{
    return reg <= 0x7F;
}

static bool my_writeable(struct device *dev, unsigned int reg)
{
    return reg >= 0x40 && reg <= 0x7F;
}

static bool my_volatile(struct device *dev, unsigned int reg)
{
    return reg == 0x00 || reg == 0x01;     /* status regs */
}

static const struct regmap_config my_regmap_config = {
    .reg_bits = 8,
    .val_bits = 8,
    .max_register = 0xFF,
    .readable_reg = my_readable,
    .writeable_reg = my_writeable,
    .volatile_reg = my_volatile,
    .cache_type = REGCACHE_RBTREE,
};
```

With `cache_type = REGCACHE_RBTREE`, regmap caches every non-volatile, non-read-only register in a red-black tree. A `regmap_read` of a cached register returns the cached value instantly. Only volatile registers hit the bus. A `regmap_write` updates both the cache and the bus. After resume from suspend, `regcache_sync(regmap)` flushes the cache back to the chip.

This last point, `regcache_sync`, is a huge win for power management. After resume, instead of re-reading every config register, the driver calls `regcache_sync` and regmap replays only the registers whose cached value differs from defaults.

Three cache types:

- `REGCACHE_NONE`: no caching. Every access hits the bus. Default.
- `REGCACHE_RBTREE`: red-black tree. Sparse register space, good for chips with thousands of registers used sparsely.
- `REGCACHE_FLAT`: flat array. Dense register space, fastest, good for chips with <128 registers all used.

Default values

Combined with caching, you can provide the chip's register power-on defaults. Regmap uses these to decide what to push to hardware on `regcache_sync`:

```
static const struct reg_default my_defaults[] = {
    { 0x40, 0x10 },
    { 0x41, 0x00 },
```

```
    { 0x42, 0xFF },
    /* ... */
};

static const struct regmap_config my_regmap_config = {
    /* ... */
    .reg_defaults = my_defaults,
    .num_reg_defaults = ARRAY_SIZE(my_defaults),
};
```

50.4 The API, what you'll actually call

```
int regmap_read(struct regmap *rm, unsigned int reg, unsigned int *val);
int regmap_write(struct regmap *rm, unsigned int reg, unsigned int val);

int regmap_update_bits(struct regmap *rm, unsigned int reg,
                       unsigned int mask, unsigned int val);
/* atomic read-modify-write: val = (cur & ~mask) | (val & mask) */

int regmap_bulk_read(struct regmap *rm, unsigned int reg, void *val, size_t count);
int regmap_bulk_write(struct regmap *rm, unsigned int reg, const void *val, size_t
count);
/* sequence of count registers starting at reg */

int regmap_multi_reg_write(struct regmap *rm, const struct reg_sequence *regs, int num);
/* a list of (reg, val) pairs, written in one go */
```

For one-shot ops, `regmap_read` and `regmap_write`. For bit twiddling, `regmap_update_bits`. For initialisation of many registers in one go, `regmap_multi_reg_write` with a static `reg_sequence` array.

Example, a chip init sequence:

```
static const struct reg_sequence my_init_sequence[] = {
    { 0x40, 0x00 },        /* disable */
    { 0x41, 0x80 },        /* enable mode */
    { 0x42, 0x33 },        /* set sample rate */
    { 0x43, 0x05 },        /* set gain */
    { 0x40, 0x01 },        /* enable */
};

err = regmap_multi_reg_write(regmap, my_init_sequence,
                             ARRAY_SIZE(my_init_sequence));
```

Six bus transactions but one line of driver code.

50.5 regmap + IIO + interrupts, the full pattern

Here's a sketch of a complete modern driver, say, an environmental sensor, combining everything from Ch 36–49:

```
struct mysensor {
    struct regmap *regmap;
    int irq;
};

static const struct regmap_config mysensor_regmap_config = {
    .reg_bits = 8,
    .val_bits = 8,
    .max_register = 0xFF,
    .cache_type = REGCACHE_RBTREE,
};

static int mysensor_read_raw(struct iio_dev *idev, ...)
{
    struct mysensor *p = iio_priv(idev);
    unsigned int hi, lo;

    switch (info) {
    case IIO_CHAN_INFO_RAW:
        regmap_read(p->regmap, REG_DATA_HI, &hi);
        regmap_read(p->regmap, REG_DATA_LO, &lo);
        *val = (hi << 8) | lo;
        return IIO_VAL_INT;
    /* ... */
    }
}

static irqreturn_t mysensor_irq(int irq, void *data)
{
    struct iio_dev *idev = data;
    /* read data, push to buffer ... */
    iio_trigger_notify_done(idev->trig);
    return IRQ_HANDLED;
}

static int mysensor_probe(struct i2c_client *client, ...)
{
    struct iio_dev *idev = devm_iio_device_alloc(&client->dev, sizeof(struct mysensor));
    struct mysensor *p = iio_priv(idev);

    p->regmap = devm_regmap_init_i2c(client, &mysensor_regmap_config);
    /* configure chip */
    regmap_multi_reg_write(p->regmap, mysensor_init_seq, ...);

    /* request irq */
    devm_request_threaded_irq(&client->dev, gpiod_to_irq(...), NULL, mysensor_irq, ...);

    idev->name = "mysensor";
    idev->info = &mysensor_iio_info;
    idev->channels = mysensor_channels;
    /* ... */
    return devm_iio_device_register(&client->dev, idev);
}
```

Around 200 lines total. The same chip without regmap or IIO would be 600+. The frameworks save you that code.

> **IIO:** Industrial I/O, Linux's subsystem for sensors, ADCs, DACs, and buffered sampled data.

50.6 Debug: /sys/kernel/debug/regmap

With `CONFIG_REGMAP_DEBUG=y`, every regmap is exposed via debugfs:

```
[root@pa-mini:~]# ls /sys/kernel/debug/regmap/
1-0076

[root@pa-mini:~]# cd /sys/kernel/debug/regmap/1-0076
[root@pa-mini:~]# ls
cache_only  cache_bypass  name  range  registers  rbtree

[root@pa-mini:~]# cat name
1-0076

[root@pa-mini:~]# cat registers
00: 60
01: 00
02: 00
...
F4: 27
```

`registers` dumps the cached or read register values. `cache_only = 1` makes subsequent regmap accesses return cached values without touching the bus (useful for debugging "what does my driver *think* the chip looks like?"). `cache_bypass = 1` forces every access to the bus (useful for "is the bus actually working?").

For interactive driver debugging during bring-up, this is the tool you reach for.

50.7 Key Notes

> **Privilege boundary:** $ means normal user. # or sudo means root and can change host or target state. After a privileged command, verify the expected device, service, or file appears before continuing. Roll back by undoing the config change or stopping the service you just enabled.

- Regmap separates register semantics from bus encoding. Address width, value width, endianness, and command flags must describe the actual chip protocol.
- Readable, writable, volatile, and side-effect registers need distinct policies. Caching status or replaying command registers can change device behavior rather than merely improve performance.

- Regmap locking serializes accesses through that map. A larger multi-register state transition may still require a driver-level lock and a hardware-specific sequence.
- Read-modify-write helpers are appropriate only for registers whose read and write semantics permit them. Write-one-to-clear and clear-on-read registers need particular care.
- Power loss can invalidate the relationship between cached configuration and hardware. Resume must restore power and bus access before synchronizing suitable cached state.
- Direct bus access outside the map can leave its cache stale. Debug views and cache controls are observations of the access layer, not proof of a functioning peripheral.

50.8 Pitfalls

- **Wrong `reg_bits` or `val_bits`.** Symptom: writes appear to "work" but reads come back wrong, or two-register chips return garbage. Check the chip's datasheet bus-protocol section carefully.
- **`read_flag_mask` for SPI.** If the chip uses the top bit as R/W, you must set this. Symptom: reads return zeros (the chip thinks every transaction is a write).
- **Caching a volatile register.** Subsequent reads return stale data. Always mark status/event registers as volatile.
- **Not marking the read-only registers as such.** `regmap_write` to a read-only register silently fails on the chip, but regmap still caches the value. Next read returns the cached (wrong) value. Always declare readability/writability.
- **`regcache_sync` without `regcache_mark_dirty` first.** `sync` only pushes registers that have been marked dirty since the last sync. After a real power-loss/restore, mark dirty first.
- **Mixing regmap and direct bus access on the same chip.** Don't. Pick one. The cache will diverge from the chip.
- **Forgetting to free the regmap.** With `devm_regmap_init_*`, you don't, but without `devm_`, you must `regmap_exit` in remove.
- **Endianness mismatch.** Chip is big-endian, driver assumes little-endian. Symptom: 16-bit values appear byte-swapped. Set `reg_format_endian = REGMAP_ENDIAN_BIG` in config.

50.9 Going deeper

- **`Documentation/driver-api/regmap.rst`**: the regmap kernel documentation.
- **`include/linux/regmap.h`**: the API. Skim once for the full set of functions.

- **`drivers/base/regmap/`**: implementation. `regcache-rbtree.c`, `regcache-flat.c` are short and instructive.
- **`drivers/iio/pressure/bmp280-core.c`**: clean regmap + IIO example.
- **`drivers/mfd/`**: MFD (multi-function device) drivers heavily use regmap for chip-wide register layouts shared across cell drivers.
- **`sound/soc/codecs/wm8960.c`**: a real audio codec driver showing what 200+ registers looks like with regmap.

> **End of Phase 2 (Ch 44–50).** You now have the seven foundational subsystems: GPIO/pinctrl, input, I²C, SPI, PWM/RTC, IIO, regmap. With these, almost every peripheral driver in the kernel becomes legible. The remaining Part VI chapters (51–55I) layer on DMA, watchdog, power management, networking, sound, display, USB, kernel timers, and PREEMPT_RT, but the *shape* of every one of those drivers is now familiar. **MCU bridge:** Think of Linux PWM like an MCU timer output channel, except the driver exposes period, duty cycle, polarity, and enable state through a subsystem. **MCU bridge:** Think of Linux GPIO like the same pin set/reset block you used on STM32, but accessed through a kernel subsystem that owns numbering, direction, interrupts, and user-space exposure. **DMA:** Direct Memory Access. Hardware moves data to or from memory without the CPU copying each byte. **PWM:** Pulse-Width Modulation, a timer output whose duty cycle controls average power or encodes timing. **GPIO:** General-Purpose Input/Output, a pin controlled as a digital input, output, or interrupt source. **PREEMPT_RT:** the Linux real-time patch set that makes more kernel paths preemptible and reduces latency.

> Next chapter: **Chapter 51: DMA.** When CPU-driven memcpy is too slow, the DMA controller takes over. The kernel's `dmaengine` framework gives drivers a portable API to set up and submit transfers across any SoC's DMA hardware.

Chapter 51: DMA

> **MMIO:** memory-mapped I/O, where software accesses peripheral registers through normal load and store instructions.

> **What:** Linux's `dmaengine` framework, the portable way to ask a hardware DMA controller to move bytes between memory and a peripheral (or memory and memory). The examples explain channel acquisition, descriptor submission, buffer ownership, and completion notification. **DMA:** Direct Memory Access. Hardware moves data to or from memory without the CPU copying each byte.
>
> **Why:** The CPU is bad at bulk data moves. At 10 Mbps SPI, one IRQ per byte burns a large fraction of the i.MX6ULL CPU. SDMA does the same job at zero CPU cost. Any driver that streams data, SPI, I²S audio, CSI camera, LCDIF, eMMC, uses DMA. Once you know the consumer API, most DMA-using drivers read the same way. **MCU bridge:** Think of an IRQ like an EXTI/NVIC interrupt path, except Linux splits the hard interrupt from deferred work and must share lines across drivers. **IRQ:** interrupt request, the signal path that tells the CPU or interrupt controller that hardware needs service.
>
> **Focus: the four standard steps**, request a channel, configure direction & widths, prepare a descriptor, submit + issue + wait. Once you know these four steps, most DMA drivers look the same.

51.1 When and why

i.MX6ULL has a Smart DMA controller (**SDMA**), a programmable peripheral DMA engine separate from the Cortex-A7. It runs little "scripts" that move data between system DRAM and peripheral FIFOs (UART RX, SPI TX, I²S, etc.), interrupting the CPU only on completion or error.

Without DMA, an SPI driver writes one byte to TX, waits for "TX empty," writes the next byte, at a few MHz that's tens of thousands of IRQs per second. With DMA, you tell SDMA "move these 4096 bytes to SPI TX, then notify me." Zero CPU between start and end.

The trade-off:

- **Setup cost.** Configuring an SDMA transfer costs ~1 µs. For 4-byte transfers, PIO is faster.

- **Memory pinning.** DMA needs physically-contiguous, cache-coherent buffers. Use `dma_alloc_coherent` (slower, smaller pool) or `dma_map_single` (faster, manages cache). The MMU/cache material from Ch 4 matters here.

> **MCU bridge:** Think of the MMU as a hardware address translator in front of every load/store. Cortex-M usually runs physical addresses directly. Linux relies on virtual addresses and page permissions. **MMU:** Memory Management Unit, hardware that translates virtual addresses to physical addresses and enforces permissions.

- **Debug pain.** A misconfigured DMA writes to random memory. Bugs are harder to diagnose than PIO bugs.

Rule of thumb: use DMA for transfers ≥ 64 bytes, polling/PIO for shorter.

51.2 The two halves of dmaengine

```
   driver (SPI controller, UART, sound, etc.)
        │  dma_request_chan, dmaengine_prep_*, dmaengine_submit, dma_async_issue_pending
        ▼
   ┌──────────────────────────────────────────────────────────┐
   │   dmaengine core                                         │
   │   - channel allocator (round-robin among free channels)  │
   │   - descriptor management                                │
   │   - completion callbacks                                 │
   └──────────────────────────────────────────────────────────┘
        │
        ▼
   provider (SDMA controller driver — drivers/dma/imx-sdma.c)
        │ MMIO + IRQ
        ▼
   hardware
```

You don't write a provider unless you're porting Linux to new SoC silicon. You write *consumers*: drivers that request a channel and submit transfers.

51.3 Device tree, declaring DMA channels for a peripheral

A peripheral that wants DMA declares it in DT via the standard `dmas` + `dma-names` pair:

```
&ecspi3 {
    pinctrl-names = "default";
    pinctrl-0 = <&pinctrl_ecspi3>;
    dmas = <&sdma 7 7 1>, <&sdma 8 7 2>;
    dma-names = "rx", "tx";
    status = "okay";
};
```

Each `<&sdma N M K>` triple is provider-specific. For i.MX SDMA: `<&sdma <channel> <type> <priority>>`. `channel` and `type` come from the SDMA event-mux table (e.g., ECSPI3 RX = event 7). The SoC datasheet has the mapping.

`dma-names` gives each channel a *symbolic* name. Drivers ask for `"rx"` or `"tx"` (not channel number 7). Same pattern as PWM and clocks.

> **MCU bridge:** Think of Linux PWM like an MCU timer output channel, except the driver exposes period, duty cycle, polarity, and enable state through a subsystem.
> **PWM:** Pulse-Width Modulation, a timer output whose duty cycle controls average power or encodes timing.

51.4 The four standard steps

In a peripheral driver's probe, plus the runtime transfer path:

Step 1, Request the channel

```
struct dma_chan *rx_chan, *tx_chan;

rx_chan = dma_request_chan(&pdev->dev, "rx");
if (IS_ERR(rx_chan))
    return dev_err_probe(&pdev->dev, PTR_ERR(rx_chan), "no RX dma\n");

tx_chan = dma_request_chan(&pdev->dev, "tx");
if (IS_ERR(tx_chan)) {
    err = PTR_ERR(tx_chan);
    goto release_rx;
}
```

Returns a `struct dma_chan *`, your handle. Don't forget `dma_release_channel(chan)` in remove (or use the `devm_` style: `devm_get_free_pages` etc. Don't have a DMA equivalent for channel requests, so manual cleanup).

Step 2, Configure slave parameters

For peripheral DMA ("slave" DMA, meaning the engine is slave to your peripheral's pacing), tell the engine where the peripheral FIFO is and how wide each transfer is:

```
struct dma_slave_config cfg = {
    .direction       = DMA_DEV_TO_MEM,            /* receive */
    .src_addr        = peripheral_phys + RX_FIFO,
    .src_addr_width  = DMA_SLAVE_BUSWIDTH_1_BYTE,
    .src_maxburst    = 1,
};
dmaengine_slave_config(rx_chan, &cfg);

cfg.direction      = DMA_MEM_TO_DEV;
cfg.dst_addr       = peripheral_phys + TX_FIFO;
```

```
cfg.dst_addr_width = DMA_SLAVE_BUSWIDTH_1_BYTE;
cfg.dst_maxburst   = 1;
dmaengine_slave_config(tx_chan, &cfg);
```

Set once at probe (or whenever it changes).

Step 3, Prepare a descriptor

For each transfer:

```
struct dma_async_tx_descriptor *desc;
dma_cookie_t cookie;
dma_addr_t  dma_buf_phys;

dma_buf_phys = dma_map_single(&pdev->dev, my_buffer, len, DMA_FROM_DEVICE);
if (dma_mapping_error(&pdev->dev, dma_buf_phys))
    return -ENOMEM;

desc = dmaengine_prep_slave_single(rx_chan, dma_buf_phys, len,
                                   DMA_DEV_TO_MEM,
                                   DMA_PREP_INTERRUPT | DMA_CTRL_ACK);
if (!desc) {
    dma_unmap_single(&pdev->dev, dma_buf_phys, len, DMA_FROM_DEVICE);
    return -ENOMEM;
}

desc->callback        = rx_done_cb;
desc->callback_param  = priv;
```

`dma_map_single` pins the buffer in physical memory, flushes the CPU cache appropriately, and returns the bus-visible physical (or IOVA) address. The kernel handles the cache flush/invalidate sequence, your CPU might have modified the buffer, so caches are flushed *to memory* before DMA reads from it (`DMA_TO_DEVICE`), or invalidated *from memory* after DMA writes to it (`DMA_FROM_DEVICE`).

For scatter-gather (multiple non-contiguous chunks): `dmaengine_prep_slave_sg(chan, sg_list, sg_count, dir, flags)`.

For ring-buffer continuous capture (audio, CSI): `dmaengine_prep_dma_cyclic(chan, dma_addr, total, period, dir, flags)`.

Step 4, Submit, issue, wait

```
cookie = dmaengine_submit(desc);

dma_async_issue_pending(rx_chan);

/* In rx_done_cb (runs in tasklet context): */
static void rx_done_cb(void *param)
{
    struct my_priv *p = param;
    dma_unmap_single(p->dev, p->dma_buf_phys, p->len, DMA_FROM_DEVICE);
    complete(&p->rx_done);     /* wake the waiter */
```

```
}

/* In the original context, wait: */
wait_for_completion(&p->rx_done);
```

`dmaengine_submit` queues the descriptor. `dma_async_issue_pending` actually kicks the engine. Most drivers separate them so they can queue multiple descriptors before starting (chained transfers).

The callback runs in *tasklet context*, no sleeping, no `kmalloc(GFP_KERNEL)`, no `mutex_lock`. Use it to release the mapping and signal a completion that a sleeping waiter can pick up.

51.5 Cyclic transfers (audio, camera)

For continuous streaming where the same buffer is read in a circle:

```
/* Allocate a coherent ring buffer of 8 periods × 4096 bytes */
size_t period = 4096, total = 8 * period;
void *cpu_buf;
dma_addr_t dma_buf_phys;

cpu_buf = dma_alloc_coherent(&pdev->dev, total, &dma_buf_phys, GFP_KERNEL);

desc = dmaengine_prep_dma_cyclic(rx_chan, dma_buf_phys, total, period,
                                 DMA_DEV_TO_MEM,
                                 DMA_PREP_INTERRUPT | DMA_CTRL_ACK);
desc->callback       = period_elapsed_cb;
desc->callback_param = priv;

dmaengine_submit(desc);
dma_async_issue_pending(rx_chan);
```

The callback fires once per *period*, every 4096 bytes. User-space drains via separate API (ALSA’s snd_pcm_indirect or similar).

> **ALSA:** Linux’s kernel and user-space audio stack.

`dma_alloc_coherent` allocates from the kernel’s coherent-DMA pool, guaranteed cache-coherent (writes from the CPU are immediately visible to DMA and vice versa), at the cost of being uncached for the CPU. For audio ring buffers this is perfect.

51.6 Memory-to-memory DMA

For pure memcpy acceleration:

```
struct dma_chan *chan = dma_request_chan_by_mask(&dma_cap_zero(mask));
dma_cap_set(DMA_MEMCPY, mask);
```

```
desc = dmaengine_prep_dma_memcpy(chan, dst_dma, src_dma, len, flags);
/* ... submit, issue, wait ... */
```

Memcpy DMA is rare on i.MX6ULL (CPU is fast enough at small sizes). Useful on bigger SoCs where the DMA engine has multiple lanes and can outpace a single CPU.

51.7 Cache coherency, the trap

A common bug: User-space fills a buffer. The driver hands the kernel pointer to DMA. DMA reads stale data, the CPU's recent writes are still in L1 cache.

`dma_map_single(dev, ptr, size, DMA_TO_DEVICE)` solves this by **flushing** (writing back) the relevant cache lines to memory before the DMA reads them. The matching `dma_unmap_single` does nothing for `TO_DEVICE`. For `FROM_DEVICE`, it **invalidates** the cache so subsequent CPU reads come from the (DMA-written) memory.

The kernel handles cache coherency for you, but only if you use the APIs. If you cast a pointer to `dma_addr_t` and skip `dma_map_*`, you get random data corruption. Whether it appears depends on whether the cache line was evicted between operations. *Always use dma_map_ or dma_alloc_coherent, never cast.**

51.8 Key Notes

- The DMA provider owns the controller, while a consumer owns transfer intent, buffers, and completion handling. A channel request must match the peripheral's routing and capabilities.
- DMA mapping establishes a device-visible address and cache-ownership rules. A CPU pointer cast is not a DMA mapping, even when its numeric value appears plausible.
- Streaming mappings require direction-specific ownership transitions. Coherent allocation simplifies cache visibility but does not remove ordering or lifetime requirements.
- Preparation, submission, and issuing work are distinct stages. A queued descriptor is not necessarily an active transfer, and every failure stage needs balanced cleanup.
- Buffers and callback state remain alive until completion or synchronized termination. Timeout and removal paths cannot free memory while the engine can still access it.
- Cyclic transfers repeatedly hand periods between hardware and software. Period size and queue depth trade notification overhead against latency and overrun margin, rather than guaranteeing unlimited throughput.

51.9 Pitfalls

- **Forgetting to call `dma_async_issue_pending`.** Descriptor sits queued. Callback never fires. Driver hangs. The split between submit and issue is intentional (for chaining) but easy to get wrong.
- **Wrong direction in dma_map_single.** `TO_DEVICE` for outgoing (driver writes, DMA reads). `FROM_DEVICE` for incoming (DMA writes, driver reads). `BIDIRECTIONAL` if both. Wrong direction = stale data.
- **Buffer alignment / size constraints.** Some DMA engines require power-of-2 sizes or specific alignment. Check the engine's `device_caps`.
- **DMA on stack buffers.** `dma_map_single(&stack_var, ...)`, the stack isn't necessarily DMA-able memory. Use `kmalloc(GFP_DMA | GFP_KERNEL, ...)` or `dma_alloc_coherent`.
- **Sleeping in the callback.** Tasklet context. Use `complete()`, `wake_up()`, or schedule a workqueue.
- **Not releasing the channel.** `dma_release_channel` in remove (or DMA channels leak. Eventually you run out).
- **Two consumers requesting the same channel.** First wins. Second fails with -EBUSY. DT specifies which event line each peripheral uses. Conflicts mean rewiring.
- **Cache flushing on bidirectional transfers.** Subtle. For `BIDIRECTIONAL` the kernel flushes before *and* invalidates after, slower. Use one direction whenever possible.

51.10 Going deeper

- **`Documentation/driver-api/dmaengine/`**: the dmaengine framework documentation. `client.rst` is the consumer guide.
- **`drivers/dma/imx-sdma.c`**: the i.MX SDMA controller driver. Big (~2500 lines). Worth skimming once.
- **`drivers/spi/spi-imx.c`**: a clean dmaengine consumer.
- **`sound/soc/fsl/imx-sdma.c`** + `imx-pcm-dma.c`, audio's cyclic DMA usage.
- **`Documentation/core-api/dma-api-howto.rst`**: DMA buffer ownership rules and cache coherency, the canonical reference.

> Next chapter: **Chapter 51A: Watchdog.** A simple but critical subsystem. Every shipping product needs to recover automatically from a hung kernel or stuck application.

Chapter 51A: Watchdog

> **What:** the **watchdog subsystem**, `/dev/watchdog`, the `watchdog_device` framework, and the user-space keepalive policy. Timer expiry requests a hardware reset, while the feeder's health criteria determine which failures can lead to that reset.
>
> **Why:** Most shipping products need this. A kernel oops in some other subsystem. A deadlock in your driver. A user-space process stuck in an infinite loop. Without a watchdog, the device becomes a brick that needs a manual power-cycle. With one, the device reboots within seconds, logs the event, and is back in service.
>
> **Focus: the keepalive contract**. Some user-space process *must* write to `/dev/watchdog` (or call the right ioctl) before the timer expires, forever. If that process dies, hangs, or gets stuck on disk I/O, the watchdog fires and the system resets. Picking *which* process should hold this responsibility, and what "alive" means to it, is the design decision.

51A.1 Hardware vs software watchdog

Two kinds exist:

Hardware watchdog, a SoC peripheral or external IC. Independent of the CPU. Counts down a register. On zero, asserts the system's RESET signal. Survives kernel hangs because it's not running kernel code. I.MX6ULL has two: WDOG1 and WDOG2 (in SNVS). External ICs like TPS3823, MAX6369 provide an even more independent watchdog (someone else's silicon, on a separate power rail).

Software watchdog (`softdog` module), a kernel timer-based watchdog. Useful for testing but useless against a kernel that's truly stuck (the timer is part of the kernel that hung). Don't ship products with only softdog.

Always use a hardware watchdog in production. Use softdog only for dev-host testing.

51A.2 Enable WDOG1 in DT

```
&wdog1 {
    pinctrl-names = "default";
    pinctrl-0 = <&pinctrl_wdog>;
    fsl,ext-reset-output;
    status = "okay";
};
```

`fsl,ext-reset-output` exports the WDOG_B pin so an external chip (PMIC) can react to the reset. Required if your PMIC supplies CPU power and needs an explicit reset signal.

> **MCU bridge:** Think of a PMIC like a programmable power-tree supervisor: it replaces discrete enables and LDO assumptions with sequenced rails the kernel can model. **PMIC:** Power Management IC, a chip that sequences and regulates the board's voltage rails.

The driver is `imx2_wdt` (mainline). Once enabled, `/dev/watchdog` appears (also `/dev/watchdog0`).

51A.3 The user-space ioctl interface

`/dev/watchdog` follows the legacy "any write keeps it alive" convention plus a richer ioctl interface:

```
#include <linux/watchdog.h>

int fd = open("/dev/watchdog", O_WRONLY);

/* Configure timeout */
int timeout = 30;
ioctl(fd, WDIOC_SETTIMEOUT, &timeout);

/* Get current timeout */
ioctl(fd, WDIOC_GETTIMEOUT, &timeout);

/* Keepalive */
ioctl(fd, WDIOC_KEEPALIVE, NULL);
/* equivalently: */
write(fd, "x", 1);

/* Disarm (gracefully — driver-dependent) */
ioctl(fd, WDIOC_SETOPTIONS, &(int){ WDIOS_DISABLECARD });

/* On close: by default, the watchdog stays armed → reset after timeout
 * unless your kernel is built with CONFIG_WATCHDOG_NOWAYOUT=n and you write "V" */
write(fd, "V", 1);    /* "magic close" — disable watchdog on close */
close(fd);
```

Magic close: With `CONFIG_WATCHDOG_NOWAYOUT=y` (the usual default), once `/dev/watchdog` is opened, it cannot be safely closed. To disable it on close, write a `'V'` first. Closing without the `'V'` leaves it armed. This prevents a buggy daemon from accidentally disabling the watchdog by exiting.

For production builds, leave `NOWAYOUT=y`, your keepalive process is supposed to live forever. If it dies, you *want* the watchdog to fire.

51A.4 Picking a keepalive process

Three common patterns:

Pattern A, systemd's built-in watchdog

If you use systemd, the simplest answer is `WatchdogSec` in your unit files. The PID 1 systemd opens `/dev/watchdog`, sets the timeout, and the kernel resets unless systemd itself keeps writing.

```
[Service]
ExecStart=/usr/bin/my-app
WatchdogSec=30s
Restart=always
```

The unit becomes responsible: it must call `sd_notify(0, "WATCHDOG=1")` periodically. If it doesn't, systemd assumes the unit is hung and restarts it. Systemd as a whole is hung → kernel resets.

Layered watchdog: hardware → systemd → application.

Pattern B, busybox `watchdog` daemon

For non-systemd setups, BusyBox has a `watchdog` command:

```
# In /etc/init.d/rcS or equivalent:
/sbin/watchdog -t 5 /dev/watchdog
```

`-t 5` says "feed every 5 seconds." It runs forever, writing to `/dev/watchdog`. Simple but dumb, it only checks if `watchdog` itself is running. It can't tell if the application is hung.

Pattern C, application-aware feeder

Better: a tiny daemon that periodically checks the application is healthy, then feeds the watchdog. "Healthy" might mean: pings reachable, /proc//status is "S" not "D", a heartbeat counter in shared memory advances.

```
while (1) {
    if (app_is_healthy()) {
        write(wdfd, "x", 1);
    }
    sleep(5);
}
```

This catches the case where the application is *running* but stuck, perhaps in a deadlock or infinite loop, which dumb keepalive misses.

51A.5 What to do with watchdog reset events

A reset by watchdog should be **observable** so you can debug. Two mechanisms:

1. **Boot-reason register.** Most SoCs (including i.MX6ULL via SRC_SRSR) record why the system reset. On boot, `imx2_wdt` reads this and reports via `dmesg` / `/proc/sys/kernel/last_reboot_reason` if available.
2. **Persistent log on reset.** Use a `pstore` / `ramoops` region, a small chunk of DRAM marked “preserved across warm reset”, to save kernel oops and console buffer. On the next boot, the recovered data lives in `/sys/fs/pstore/`.

```
reserved-memory {
    #address-cells = <1>;
    #size-cells = <1>;
    ranges;

    ramoops@9c000000 {
        compatible = "ramoops";
        reg = <0x9c000000 0x100000>;     /* 1 MB */
        record-size  = <0x4000>;
        console-size = <0x4000>;
        ftrace-size  = <0x0>;
    };
};
```

After a watchdog reset, `ls /sys/fs/pstore/` shows `dmesg-ramoops-N`, `console-ramoops-N`, the last KB of dmesg before the hang. Very useful for debugging field failures.

51A.6 Writing a watchdog driver (for completeness)

If you ever needed to support a chip not in mainline:

```
#include <linux/watchdog.h>

static int my_wdt_start(struct watchdog_device *wdd) { /* enable HW */ }
static int my_wdt_stop(struct watchdog_device *wdd)  { /* disable HW */ }
static int my_wdt_ping(struct watchdog_device *wdd)  { /* feed timer */ }
static int my_wdt_set_timeout(struct watchdog_device *wdd, unsigned int t) { /* program
timeout */ }

static const struct watchdog_ops my_wdt_ops = {
    .owner = THIS_MODULE,
    .start = my_wdt_start,
    .stop  = my_wdt_stop,
    .ping  = my_wdt_ping,
    .set_timeout = my_wdt_set_timeout,
};

static const struct watchdog_info my_wdt_info = {
    .options = WDIOF_SETTIMEOUT | WDIOF_KEEPALIVEPING | WDIOF_MAGICCLOSE,
    .identity = "my watchdog",
};
```

```
/* In probe: */
struct watchdog_device *wdd = devm_kzalloc(...);
wdd->ops = &my_wdt_ops;
wdd->info = &my_wdt_info;
wdd->min_timeout = 1;
wdd->max_timeout = 128;
wdd->timeout = 30;
watchdog_set_drvdata(wdd, priv);

return devm_watchdog_register_device(&pdev->dev, wdd);
```

The core handles `/dev/watchdog`, ioctls, and sysfs. You just implement the four ops.

> **sysfs:** a kernel-generated filesystem under /sys that exposes devices, drivers, and attributes.

51A.7 Key Notes

- A hardware watchdog provides a reset path that is less dependent on kernel scheduling than a software timer. Its actual clock, power, and reset dependencies bound its independence.
- The feeder owns the definition of progress. Feeding because one daemon still runs can conceal failure of the application it is meant to supervise.
- Application supervision and hardware keepalive are separate layers. Service restart policy and whole-system reset policy need distinct owners and time budgets.
- Timeout selection includes scheduling delays, storage stalls, boot handoff, and suspend behavior. A timeout that catches hangs can also interrupt legitimate slow operation.
- `nowayout`, magic-close support, and driver capabilities determine whether an armed watchdog can stop. Closing a descriptor is not a universally safe disarm operation.
- Reset-reason information and persistent logs help explain recovery, but require supported hardware and memory preservation. Watchdog reset can interrupt storage writes and does not establish successful recovery of the MINI or its attached devices.

51A.8 Pitfalls

- **`CONFIG_WATCHDOG_NOWAYOUT=n` in production.** A buggy daemon `close()`s the device → watchdog disabled → product hangs forever. Always `=y` in shipped kernels.

- **Forgetting to feed during heavy I/O.** If your keepalive process gets stuck on disk I/O (D state), it can't feed. Worst case: watchdog fires during normal operation. Tune timeout to be longer than longest expected I/O burst.
- **Pre-init watchdog**: The bootloader (U-Boot) can start the watchdog before the kernel boots. If the kernel takes longer to boot than the timeout, watchdog fires during boot. Either U-Boot disables it before jumping, or kernel takes over fast.

> **MCU bridge:** Think of U-Boot like a much larger boot stub plus debug monitor: it initializes hardware, loads the next image, and gives you commands before Linux starts. **U-Boot:** the bootloader that initializes enough hardware to load and start the Linux kernel.

- **Multiple processes opening `/dev/watchdog`.** First open arms it. Later opens get -EBUSY (in most drivers). Stick to one feeder process.
- **Watchdog during suspend.** Suspended kernel can't feed. Most watchdog drivers stop the timer on suspend automatically. Verify your specific driver's behavior.
- **Short timeouts.** A 2-second timeout has no slack for slow user-space ops. 30 seconds is a saner default. Some products use 5–10 minutes (high-availability gear with long expected blocking ops).
- **Forgetting ramoops region from kernel cmdline.** `mem=...` cuts off the reserved area. Tune carefully so ramoops's physical address is *inside* the visible-to-kernel memory map.

51A.9 Going deeper

- **`Documentation/watchdog/`**: the watchdog subsystem documentation.
- **`drivers/watchdog/imx2_wdt.c`**: i.MX2/3/5/6/7 watchdog driver. ~300 lines.
- **`Documentation/admin-guide/ramoops.rst`**: ramoops setup and usage.
- **`Documentation/admin-guide/pstore-blk.rst`**: pstore on block devices for systems with non-volatile storage but no preserved RAM.
- **`man systemd.service` → `WatchdogSec`**: the systemd-watchdog integration.

> Next chapter: **Chapter 51B: Power management.** Runtime PM, DVFS, suspend/resume. Watchdog + PM interact: a suspended system isn't feeding the watchdog, so PM must coordinate the watchdog driver's behavior.

Chapter 51B: Power management

> **regmap:** a kernel helper that wraps register reads and writes over I2C, SPI, or MMIO. **MCU bridge:** Think of regmap like a typed wrapper around your read_reg() and write_reg() helpers, with caching, locking, and bus differences handled centrally.

> **What:** Linux's three power-management layers, **runtime PM** (drivers autonomously gate clocks and rails when idle), **DVFS** (CPU frequency-and-voltage scaling under load) and **system sleep** (suspend-to-RAM / standby / hibernation). By the end your driver participates in runtime PM, the system suspends to RAM cleanly, and the CPU clocks down when idle.
>
> **Why:** On battery-powered products, PM tuning is a large fraction of the work. Saving 50 mA in idle means 5× battery life on a 1 Ah cell. For mains-powered embedded, it's still meaningful: less heat, smaller heatsinks, lower fan noise. Linux's PM framework is rich and *opt-in*, drivers that don't implement it don't suspend. The whole device fails to enter suspend until you fix them.
>
> **Focus: the three layers are mostly independent**. Runtime PM is "this peripheral is idle. Gate its clock now." DVFS is "the CPU isn't busy. Scale to 396 MHz." System sleep is "the user pressed the suspend button. Stop everything safely, resume on a wake source." Implement them one at a time. Don't tangle them up.

51B.1 Runtime PM

Runtime PM is the kernel's automatic per-device idle/active state machine. Each device has an idle and an active state. The framework counts users with a refcount. When the refcount hits zero it calls `runtime_suspend`. When it goes from zero to one it calls `runtime_resume`.

```
#include <linux/pm_runtime.h>

static int my_runtime_suspend(struct device *dev)
{
    struct my_priv *p = dev_get_drvdata(dev);
    clk_disable_unprepare(p->clk);    /* gate the clock */
    regulator_disable(p->vcc);         /* drop the rail */
    return 0;
}

static int my_runtime_resume(struct device *dev)
{
    struct my_priv *p = dev_get_drvdata(dev);
    int err = regulator_enable(p->vcc);
    if (err) return err;
```

```
    return clk_prepare_enable(p->clk);
}

static const struct dev_pm_ops my_pm_ops = {
    SET_RUNTIME_PM_OPS(my_runtime_suspend, my_runtime_resume, NULL)
};

static struct platform_driver my_driver = {
    .driver = {
        .name = "my-driver",
        .pm   = &my_pm_ops,
    },
    /* ... */
};

/* In probe: */
pm_runtime_set_autosuspend_delay(&pdev->dev, 1000);  /* 1 s after last use */
pm_runtime_use_autosuspend(&pdev->dev);
pm_runtime_enable(&pdev->dev);

/* In your hot path (read/write/IRQ handler entry): */
pm_runtime_get_sync(&pdev->dev);     /* wakes if suspended; bumps refcount */
do_the_work();
pm_runtime_mark_last_busy(&pdev->dev);
pm_runtime_put_autosuspend(&pdev->dev);
```

Sequence:

1. Driver calls `pm_runtime_get_sync` before using the device. If already runtime-active, this just bumps the refcount. If suspended, it calls `runtime_resume` first, then bumps.
2. Driver does work.
3. Driver calls `pm_runtime_mark_last_busy` + `pm_runtime_put_autosuspend`. The refcount drops. When it hits zero, after the autosuspend delay (1 s here), `runtime_suspend` runs.

The result: between bursts of use, the device's clock is gated and rail is dropped, automatically. No user-space involvement.

Inspect from sysfs:

> **sysfs:** a kernel-generated filesystem under /sys that exposes devices, drivers, and attributes.

```
[root@pa-mini:~]# cat /sys/devices/platform/.../power/runtime_status
suspended
[root@pa-mini:~]# cat /sys/devices/platform/.../power/runtime_active_time
12345
[root@pa-mini:~]# cat /sys/devices/platform/.../power/runtime_suspended_time
876543
```

51B.2 DVFS, CPU frequency scaling

The i.MX6ULL CPU can run at 396, 528, or 696 MHz. Higher frequency = more performance + more power. Lower = the reverse. **DVFS** (Dynamic Voltage and Frequency Scaling) picks the frequency dynamically based on load.

CPU operating points in DT:

```
&cpu0 {
    operating-points-v2 = <&cpu0_opp_table>;
    cpu-supply = <&reg_arm>;
};

cpu0_opp_table: opp-table {
    compatible = "operating-points-v2";

    opp-396000000 {
        opp-hz = /bits/ 64 <396000000>;
        opp-microvolt = <1125000>;
    };
    opp-528000000 {
        opp-hz = /bits/ 64 <528000000>;
        opp-microvolt = <1175000>;
    };
    opp-696000000 {
        opp-hz = /bits/ 64 <696000000>;
        opp-microvolt = <1275000>;
    };
};
```

Each OPP is a frequency + the voltage required to run at that frequency. The DVFS framework reads this table, picks an OPP, and applies the change. When raising the clock, the regulator goes up first, then the clock. When lowering, the reverse. This ensures the CPU is always supplied with enough voltage for its current speed.

User-space pick the **governor** (the algorithm that decides when to scale):

```
[root@pa-mini:~]# cat /sys/devices/system/cpu/cpu0/cpufreq/scaling_available_governors
performance powersave userspace ondemand conservative schedutil

[root@pa-mini:~]# cat /sys/devices/system/cpu/cpu0/cpufreq/scaling_governor
schedutil

[root@pa-mini:~]# echo performance > /sys/devices/system/cpu/cpu0/cpufreq/
scaling_governor
[root@pa-mini:~]# cat /sys/devices/system/cpu/cpu0/cpufreq/scaling_cur_freq
696000
```

Governors:

- **performance**: always max frequency. Lowest latency.
- **powersave**: always min frequency. Lowest power.

- **schedutil**: modern default. Uses scheduler hints (utilisation) to scale.
- **ondemand**: older. Reacts to CPU load.
- **conservative**: slower-reacting version of ondemand.

For embedded products, `schedutil` is almost always right. Performance for benchmarks, powersave for “device is on standby waiting for an event.”

51B.3 System sleep, suspend-to-RAM

Three system-level sleep states (described in `/sys/power/state`):

- **freeze**: userspace frozen, devices left running. Lowest latency wake (microseconds), modest power savings.
- **standby**: devices suspended, CPU clock gated. Medium savings.
- **mem**: full suspend-to-RAM. RAM in self-refresh, CPU off, only PMIC and SoC suspend domain active. Lowest power, ~1–2 second wake latency.

> **MCU bridge:** Think of a PMIC like a programmable power-tree supervisor: it replaces discrete enables and LDO assumptions with sequenced rails the kernel can model. **PMIC:** Power Management IC, a chip that sequences and regulates the board’s voltage rails.

Trigger:

```
[root@pa-mini:~]# echo mem > /sys/power/state
```

The kernel:

1. Freezes all userspace processes (sends SIGSTOP-equivalent).
2. For each device (in topological order, leaves first), calls `pm_ops.suspend`. Drivers save state, gate clocks, idle hardware.
3. The CPU is parked. Only the wakeup-capable peripherals remain powered.
4. A wakeup event fires (RTC alarm, GPIO IRQ, etc.).

> **MCU bridge:** Think of an IRQ like an EXTI/NVIC interrupt path, except Linux splits the hard interrupt from deferred work and must share lines across drivers. **MCU bridge:** Think of Linux GPIO like the same pin set/reset block you used on STM32, but accessed through a kernel subsystem that owns numbering, direction,

> interrupts, and user-space exposure. **IRQ:** interrupt request, the signal path that tells the CPU or interrupt controller that hardware needs service. **GPIO:** General-Purpose Input/Output, a pin controlled as a digital input, output, or interrupt source.

1. Resume in reverse order: devices first, then userspace thaws.

For your driver to participate:

```
static int my_suspend(struct device *dev)
{
    struct my_priv *p = dev_get_drvdata(dev);
    /* save any state that won't survive (chip registers via regmap_mark_dirty);
       gate clocks; tell the chip to enter low-power mode */
    return 0;
}

static int my_resume(struct device *dev)
{
    struct my_priv *p = dev_get_drvdata(dev);
    /* restore — typically by re-running probe-like init or regcache_sync(p->regmap) */
    return 0;
}

static const struct dev_pm_ops my_pm_ops = {
    SET_SYSTEM_SLEEP_PM_OPS(my_suspend, my_resume)
    SET_RUNTIME_PM_OPS(my_runtime_suspend, my_runtime_resume, NULL)
};
```

System-suspend callbacks run when user-space writes `mem` to `/sys/power/state`. Runtime-suspend callbacks run automatically when the device goes idle.

Wakeup sources

For the system to wake from `mem`, *something* must be allowed to fire an IRQ in the suspended state. Configure via DT (`wakeup-source` boolean) and userspace:

```
[root@pa-mini:~]# echo enabled > /sys/class/wakeup/.../wakeup
[root@pa-mini:~]# cat /sys/power/wakeup_count    ← bumps each wake
```

Common wakeup sources: RTC alarm, USB plug, GPIO from button or sensor data-ready, network wake-on-LAN.

Set an RTC alarm wake:

```
[root@pa-mini:~]# echo $(($(date +%s) + 30)) > /sys/class/rtc/rtc0/wakealarm
[root@pa-mini:~]# echo mem > /sys/power/state
# … 30 s later, system wakes ...
```

51B.4 Measuring impact

You changed something to save power. How do you measure?

External: a USB current meter (e.g., USB Type-A inline meter) or a benchtop power analyzer at the SoC supply. Best for absolute numbers.

Internal: `/sys/class/power_supply/`, `INA226` driver readouts (Ch 75), or `powertop`:

```
[root@pa-mini:~]# powertop --time=10
Summary: 23.4 wakeups/second
       Usage    Events/s   Category   Description
        5.4%        0.0    Process    /usr/bin/myapp
        ...
```

`powertop` shows which userspace process is keeping the CPU awake. Goal: idle the device under test → see “kernel sleep” at 95%+. Wakeups should be < 10/sec at idle.

51B.5 The full optimisation playbook

The order of operations for “make this product 3× longer-lasting”:

1. **Measure idle current.** Get a baseline number.
2. **Use `powertop`** to find spurious wakeups. Disable / tune.
3. **Enable runtime PM in all peripheral drivers.** Each one off saves a few mA.
4. **Configure DVFS to powersave when idle.** Or schedutil, let scheduler decide.
5. **Use `mem` suspend during long idle periods.** A few seconds of wake-up latency for a battery-life multiplier.
6. **Set wakeup-only inputs.** Don’t poll sensors. Let them IRQ on data-ready.
7. **Drop USB/Ethernet PHY power when unused.** `ip link set eth0 down` actually drops the PHY’s clock.

> **PHY:** physical-layer block or chip that converts digital MAC signals to electrical or radio signals. **MAC:** Media Access Control in networking and radio chapters. It is the layer that owns framing and medium access.

1. **Compile out unused subsystems.** A smaller kernel boots faster, holds less in cache.

Each step saves a few mA. Together they take a 1 Ah cell from 4 hours to 4 days.

51B.6 Key Notes

- Runtime PM follows a device's activity, DVFS changes operating points under load, and system sleep coordinates the whole platform. These are distinct policies with different dependency and wakeup requirements.
- A driver owns the ordering of register restoration, clocks, rails and in-flight work. Suspend is safe only when those resources can be quiesced and resumed consistently.
- Register caching is useful only when software's cached state remains aligned with hardware after reset or power removal. Resume must restore the state that the device actually lost.
- Lower current in one state is not the same as lower energy for a workload. Transition cost, time to completion and wakeup frequency belong in the comparison.
- A valid operating point depends on the fitted silicon, regulator and board conditions. A nominal frequency in an example is not permission to select it on every MINI.

51B.7 Pitfalls

- **`pm_runtime_get_sync` from atomic context.** It may sleep waiting for resume. Use `pm_runtime_get` (async. Check status) instead.
- **Forgetting to enable runtime PM.** `pm_runtime_enable(dev)` must be called once at probe. Without it, all the get/put calls are no-ops.
- **Driver that never calls `pm_runtime_get/put`.** The device is permanently "active" from PM's view. Runtime suspend never runs.
- **System suspend on a driver without suspend ops.** Kernel logs "noop_suspend" and proceeds, but your peripheral may be left in a bad state on resume. Always provide ops or use `pm_ptr` to inherit reasonable defaults.
- **Clock left enabled across suspend.** PMIC may try to drop rails while clock is still toggling, which can violate the chip's spec sheet. Order: gate clock first, then drop rail, on the way down. Reverse on resume.
- **DVFS without proper regulator support.** If `cpu-supply` is wrong, the kernel may raise the clock before voltage settles. Execution becomes unreliable. Always specify `operating-points-v2` with matched microvolt entries.
- **Wakeup source enabled but `IRQ_TYPE_LEVEL_HIGH` instead of `IRQ_TYPE_EDGE`.** Some SoCs only wake on edge IRQs from certain banks. Confirm against datasheet.
- **`wakealarm` set in the past.** Kernel ignores. System suspends indefinitely. Always compute as `$(date +%s) + N`.

51B.8 Going deeper

- **`Documentation/power/`**: the entire PM kernel documentation.
- **`Documentation/power/runtime_pm.rst`**: runtime PM in depth.
- **`Documentation/admin-guide/pm/cpufreq.rst`**: CPU frequency scaling.
- **`drivers/cpufreq/imx6q-cpufreq.c`**: i.MX6 cpufreq driver.
- **`drivers/regulator/anatop-regulator.c`**: Anatop regulator driver used for CPU voltage on i.MX.
- **`powertop` source**: kernel.org. Read for inspiration on how to measure.
- **`Documentation/power/states.rst`**: sleep state semantics.

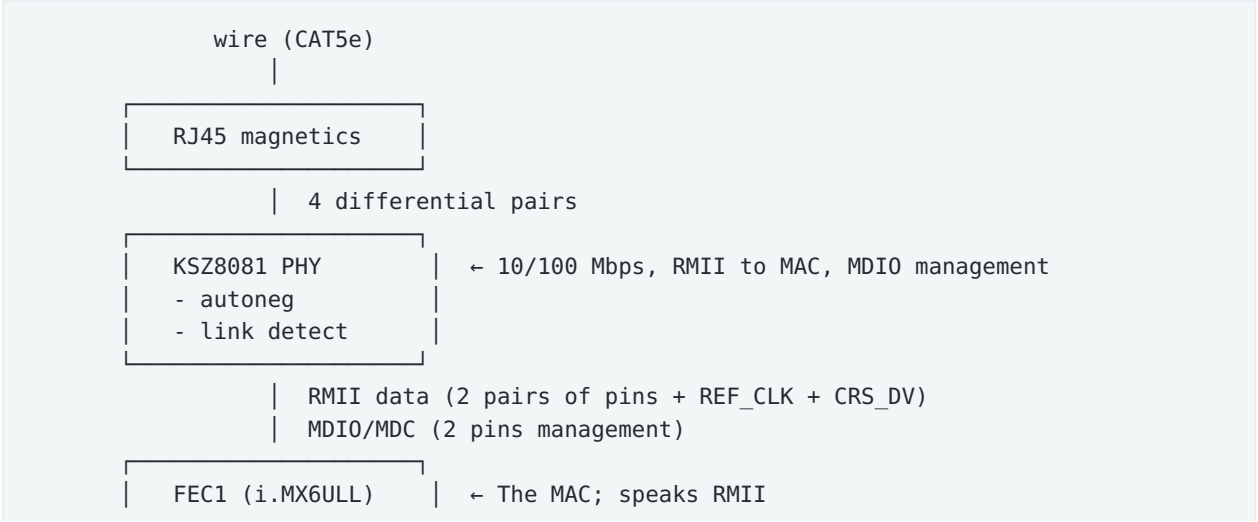
> Next chapter: **Chapter 52: Network driver (FEC + KSZ8081 PHY).** With PM understood, we tackle a real-world driver: i.MX6ULL's FEC Ethernet MAC and the KSZ8081 PHY that completes the gigabit Ethernet stack on Point Atom boards.

Chapter 52: Network driver: FEC + KSZ8081

> **DMA:** Direct Memory Access. Hardware moves data to or from memory without the CPU copying each byte. **MCU bridge:** Think of DMA like the MCU DMA controller you used for UART or SPI, but with cache coherency, scatter-gather descriptors, and kernel ownership rules added. **DDR:** external DRAM that must be configured and trained before most software can run from it.

> **What:** the i.MX6ULL's **FEC** (Fast Ethernet Controller) and the **KSZ8081** RMII PHY that nearly every Point Atom board uses. The kernel's network-device framework (`netdev`), the PHY library (`phylib`), MDIO bus operations, RMII vs MII timing, the full path from MAC to `eth0`. **MAC:** Media Access Control in networking and radio chapters. It is the layer that owns framing and medium access. **PHY:** physical-layer block or chip that converts digital MAC signals to electrical or radio signals.
>
> **Why:** Ethernet is one of the most-debugged peripherals on any embedded board. Wrong PHY ID, wrong RMII clock direction, wrong delay-line settings, and you spend a week wondering why your `ping` drops every fifth packet. The mainline `fec_main.c` + `phylib` + `kszphy.c` stack is mature. Understanding what it expects from DT and how to verify timing turns a one-week bug-hunt into a one-hour bring-up.
>
> **Focus: the FEC ↔ PHY ↔ Linux pipeline**. The FEC is the MAC (Media Access Controller). The PHY is the SerDes that turns digital frames into wire signals. The MDIO bus is the management interface between them. Linux's `netdev` exposes the result as `eth0`.

52.1 The pipeline



```
        │   - DMA              │
        │   - Tx/Rx queues     │
        └──────────────────────┘
                   │  AXI / DMA → DDR
        ┌──────────────────────┐
        │   Linux: fec_main.c  │  ← netdev driver
        │   phylib             │  ← talks to KSZ8081 via MDIO
        └──────────────────────┘
                   │
                   ▼
       Linux: eth0 (network stack)
```

i.MX6ULL has **two FEC instances**, FEC1 and FEC2. Some boards wire both (giving dual Ethernet). Point Atom MINI typically has one wired, ALPHA may have two.

52.2 Device tree

The FEC node:

```
&fec1 {
    pinctrl-names = "default", "sleep";
    pinctrl-0 = <&pinctrl_enet1>;
    pinctrl-1 = <&pinctrl_enet1_sleep>;
    phy-mode = "rmii";
    phy-handle = <&ethphy0>;
    phy-supply = <&reg_enet_3v3>;
    fsl,magic-packet;
    status = "okay";

    mdio {
        #address-cells = <1>;
        #size-cells = <0>;

        ethphy0: ethernet-phy@2 {
            compatible = "ethernet-phy-ieee802.3-c22";
            reg = <2>;
            micrel,led-mode = <1>;
            clocks = <&clks IMX6UL_CLK_ENET_REF>;
            clock-names = "rmii-ref";
        };
    };
};
```

Critical fields:

- **`phy-mode = "rmii"`**: the data interface between MAC and PHY. RMII (2-pair, 50 MHz) is what i.MX6ULL boards use. RGMII is for gigabit. Not on i.MX6ULL.
- **`phy-handle`**: points to the PHY node. The MAC driver uses phylib to talk to it.
- **`reg = <2>` (in the PHY node)**: the PHY's MDIO address. Set by board strapping (PHYAD pins).

- **`micrel,led-mode = <1>`**: Micrel/Microchip-specific tweak (link-on-bicolor vs blink-on-activity).
- **`clocks` and `clock-names = "rmii-ref"`** on the PHY, tell the PHY driver which clock provides the 50 MHz RMII reference. This is the most common bring-up bug on i.MX6ULL boards. See §52.5.

52.3 netdev framework, what the driver provides

A network driver implements `net_device_ops`:

```
static const struct net_device_ops fec_netdev_ops = {
    .ndo_open       = fec_enet_open,        /* ifconfig eth0 up */
    .ndo_stop       = fec_enet_close,        /* ifconfig eth0 down */
    .ndo_start_xmit = fec_enet_start_xmit,   /* TX one packet */
    .ndo_set_mac_address = eth_mac_addr,
    .ndo_validate_addr   = eth_validate_addr,
    .ndo_tx_timeout      = fec_timeout,
    .ndo_get_stats64     = fec_enet_get_stats,
    /* ... */
};

/* In probe: */
struct net_device *ndev = alloc_etherdev_mqs(sizeof(struct fec_enet_private),
num_tx_queues, num_rx_queues);
ndev->netdev_ops = &fec_netdev_ops;
ndev->ethtool_ops = &fec_enet_ethtool_ops;
SET_NETDEV_DEV(ndev, &pdev->dev);
register_netdev(ndev);
```

`alloc_etherdev_mqs` allocates a `net_device` with Ethernet defaults plus private storage. `register_netdev` creates the `eth0` interface and starts ifupdown / network manager hooks.

The driver receives packets in `napi_poll` (NAPI: New API. The polled receive model used since Linux 2.6) and transmits in `ndo_start_xmit`. NAPI batches RX interrupts. The driver gets one IRQ, then polls until the RX queue is empty, then re-arms the IRQ. This avoids one IRQ per packet at high rates.

> **MCU bridge:** Think of an IRQ like an EXTI/NVIC interrupt path, except Linux splits the hard interrupt from deferred work and must share lines across drivers. **IRQ:** interrupt request, the signal path that tells the CPU or interrupt controller that hardware needs service.

52.4 phylib, the PHY library

PHYs all speak (mostly) the same management protocol: **MDIO/MDC clauses 22 and 45** define a small register space (32 registers in C22, 65536 in C45) accessed over a 2-wire bus.

The MAC driver doesn't directly talk to the PHY. It uses **phylib**:

```
phydev = of_phy_connect(ndev, phy_handle, fec_link_state_change_cb,
                        0, PHY_INTERFACE_MODE_RMII);
```

The PHY's per-chip quirks live in a vendor PHY driver (`drivers/net/phy/micrel.c` for Microchip KSZ). The MAC driver just calls phylib functions:

- `phy_start(phydev)`: start autonegotiation.
- `phy_stop(phydev)`: halt.
- `phy_print_status(phydev)`: print link state.

The kernel autoloads the right PHY driver based on the PHY's identification registers (read via MDIO at boot).

```
[root@pa-mini:~]# dmesg | grep -i fec
fec 20b4000.ethernet: Falling back to support FF-MAC address
Micrel KSZ8081 RNB stepping 1.4: probed
fec 20b4000.ethernet eth0: registered PHC device 0
[root@pa-mini:~]# ip link show eth0
2: eth0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc fq_codel state UP mode DEFAULT
group default qlen 1000
    link/ether 00:11:22:33:44:55 brd ff:ff:ff:ff:ff:ff
```

52.5 The RMII clock direction trap

The KSZ8081 can operate in two RMII modes:

- **PHY supplies REF_CLK** (50 MHz oscillator inside the PHY package).
- **MAC supplies REF_CLK** (i.MX6ULL's ENET_TX_CLK pin drives the PHY).

Which? **Depends on the board.** Point Atom boards historically use *MAC-supplied* clock (i.MX provides the 50 MHz to the PHY). Other boards do the reverse.

If the clock direction is wrong, you get no link, no MDIO communication, and confusing symptoms. The DT must declare:

- The PHY's `clocks = <&clks IMX6UL_CLK_ENET_REF>;` if MAC supplies clock.
- Omit the `clocks` and instead have the PHY's `clock-names = "rmii-ref";` if PHY supplies clock, but the PHY chip itself must be ordered with the "RNB" or "RND" variant.

Cross-check schematic ↔ DT ↔ PHY chip ordering code. Mismatch is the #1 cause of "Ethernet doesn't work" on new i.MX6ULL boards.

52.6 RMII pinmux

The RMII interface uses 7 pins:

Signal	Direction	Purpose
TXD0, TXD1	MAC → PHY	Outgoing data
TX_EN	MAC → PHY	Transmit enable
RXD0, RXD1	PHY → MAC	Incoming data
CRS_DV	PHY → MAC	Carrier sense / data valid
REF_CLK	MAC↔PHY (one direction)	50 MHz reference

Plus MDIO and MDC (2 pins for PHY management). All 9 must be muxed correctly:

```
pinctrl_enet1: enet1grp {
    fsl,pins = <
        MX6UL_PAD_GPIO1_IO07__ENET1_MDC      0x1b0b0
        MX6UL_PAD_GPIO1_IO06__ENET1_MDIO     0x1b0b0
        MX6UL_PAD_ENET1_RX_DATA0__ENET1_RDATA00 0x1b0b0
        MX6UL_PAD_ENET1_RX_DATA1__ENET1_RDATA01 0x1b0b0
        MX6UL_PAD_ENET1_RX_EN__ENET1_RX_EN   0x1b0b0
        MX6UL_PAD_ENET1_RX_ER__ENET1_RX_ER   0x1b0b0
        MX6UL_PAD_ENET1_TX_DATA0__ENET1_TDATA00 0x1b0b0
        MX6UL_PAD_ENET1_TX_DATA1__ENET1_TDATA01 0x1b0b0
        MX6UL_PAD_ENET1_TX_EN__ENET1_TX_EN   0x1b0b0
        MX6UL_PAD_ENET1_TX_CLK__ENET1_REF_CLK1 0x4001b031
    >;
};
```

The last line's `0x4001b031` is the conf_reg value for the RMII reference clock, it sets the SION (Software Input ON) bit, which is required for the clock to be both an input and output. Forgetting this is a classic bug.

52.7 MAC address sources

> **Examples and production:** Do not burn fuses, enroll production keys, or sign release images while using these examples. Use throwaway keys and back up the unsigned image plus the key directory before testing irreversible security flows.

Where does `eth0`'s MAC address come from?

1. **FEC's MAC register** (preserved across warm reset, set by bootloader).
2. **OCOTP fuses** (i.MX6ULL has MAC fuses. Written once at factory).
3. **DT property `mac-address`**.
4. **Random** (last resort. Address with locally-administered bit).

The mainline `fec_main.c` checks in this order: DT mac-address → OCOTP fuse → MAC register → random. For production: program the OCOTP at factory test (one-time, indelible). For development: pass via U-Boot's `bootargs` (`eth0=...`).

> **MCU bridge:** Think of U-Boot like a much larger boot stub plus debug monitor: it initializes hardware, loads the next image, and gives you commands before Linux starts. **U-Boot:** the bootloader that initializes enough hardware to load and start the Linux kernel.

52.8 Bringing it up

After kernel boot with correct DT:

```
[root@pa-mini:~]# ip link show eth0
2: eth0: <BROADCAST,MULTICAST> mtu 1500 ...

[root@pa-mini:~]# ip link set eth0 up
[root@pa-mini:~]# dmesg | tail -2
fec 20b4000.ethernet eth0: Link is Up - 100Mbps/Full - flow control rx/tx

[root@pa-mini:~]# ip addr add 192.168.1.100/24 dev eth0
[root@pa-mini:~]# ping 192.168.1.1
PING 192.168.1.1 (192.168.1.1) 56(84) bytes of data.
64 bytes from 192.168.1.1: icmp_seq=1 ttl=64 time=0.452 ms
```

`ethtool` for inspection:

```
[root@pa-mini:~]# ethtool eth0
Settings for eth0:
        Supported ports: [ TP MII ]
        Supported link modes:   10baseT/Half 10baseT/Full
                                100baseT/Half 100baseT/Full
        Auto-negotiation: on
        Speed: 100Mb/s
        Duplex: Full
        Link detected: yes
```

`ethtool -S eth0` for hardware statistics (RX errors, TX collisions, etc.).

52.9 Dual FEC

For a board with both FEC1 and FEC2:

```
&fec1 { ... phy-handle = <&ethphy0>; ... };
&fec2 { ... phy-handle = <&ethphy1>; ... };
```

Two separate PHYs, two MDIO buses (one per FEC). Two independent `net_device`s appear (eth0 and eth1). They can be:

- Independent subnets (router scenario).
- Bonded (`ip link add bond0 type bond ...`) for redundancy.
- Bridged (`ip link add br0 type bridge ...`) for switching.

We'll go deeper on dual FEC in Part VII Ch 115 (Device Cookbook: Dual FEC + W5500).

52.10 Key Notes

- Ethernet divides responsibility between the MAC, PHY and Linux network stack. MDIO manages PHY state, while RMII carries the frame data and clock relationship.
- Device Tree describes board wiring, supplies, reset and clock direction. It cannot compensate for a physical strap or timing arrangement that contradicts those properties.
- A network interface appearing, link negotiation succeeding and packets flowing are different milestones. Throughput is a system result, not a fixed property of the driver alone.
- MAC address provenance is part of device identity. A temporary or duplicated address can work locally while failing fleet-level networking expectations.
- Link loss is a normal state transition. Driver, PHY management and application reconnect policy must cooperate rather than assume the cable remains present.

52.11 Pitfalls

- **RMII clock direction reversed.** No link. The most common, most painful. Confirm against schematic and PHY chip ordering code.
- **SION bit not set on REF_CLK pad.** Same symptom. Use `0x4001b031` (or whatever your conf_reg should be with `IMX_PAD_SION` set) for the REF_CLK pinmux.
- **Wrong PHY MDIO address.** PHY chip's `PHYAD` strapping pin determines its MDIO address. DT's `reg = <N>` must match. Default for KSZ8081 with PHYAD=1 strapped high is 1, but some boards strap it to 2.

- **Missing PHY supply.** PHY needs ~100 mA from a 3.3V rail. If reg_enet_3v3 isn't enabled at the right time, MDIO reads return 0xFFFF.
- **MII vs RMII pinmux swap.** Pads with similar names but different signals. Cross-check `imx6ul-pinfunc.h` macros.
- **MAC address all zeros.** Set OCOTP at factory, or pass via bootargs. Linux's "random" fallback won't survive reboots, breaking DHCP-leased systems.
- **Forgot phy-mode.** Without `phy-mode = "rmii"`, default may be MII (different signals). Link won't come up.
- **NAPI weight too low** for high-rate traffic. Default 64 is fine for 100 Mbps. Bump higher for gigabit (not on i.MX6ULL).
- **Cable issue.** Half the "FEC not working" reports turn out to be a bent CAT5e or a dead RJ45 jack. Try a different cable before debugging software.

52.12 Going deeper

- **`drivers/net/ethernet/freescale/fec_main.c`**: the mainline FEC driver. Important read.
- **`drivers/net/phy/micrel.c`**: KSZ8081 / KSZ9031 / etc. PHY drivers.
- **`Documentation/networking/`**: kernel networking documentation.
- **`Documentation/devicetree/bindings/net/ethernet-controller.yaml`**: generic Ethernet binding.
- **`Documentation/devicetree/bindings/net/fsl-fec.yaml`**: FEC-specific binding.
- **`Documentation/networking/phy.rst`**: phylib internals.
- **`tools/testing/selftests/net/`**: kernel's network self-tests. Useful templates for your own.

> Next chapter: **Chapter 52A: PREEMPT_RT.** When latency matters more than throughput, the real-time kernel patches turn Linux into a viable hard-real-time platform. We cover what they do, how to measure, and the trade-offs.
> **PREEMPT_RT:** the Linux real-time patch set that makes more kernel paths preemptible and reduces latency.

Chapter 52A: PREEMPT_RT

> **Driver choice:** Use the in-tree, maintained driver first. Use out-of-tree, spidev, or custom-driver paths only after you accept the kernel-version maintenance cost and document who owns updates.

> **What: PREEMPT_RT**, fully merged into the mainline kernel since **v6.12** (December 2024), is the kernel configuration that turns Linux into a hard-real-time OS, where worst-case interrupt-to-thread latency is measured in tens of microseconds on a Cortex-A7 instead of milliseconds. No out-of-tree patch is needed on v6.12 or later. We cover the four core changes (preemptible spinlocks, threaded IRQs by default, priority inheritance, high-resolution timers), how to enable it on the i.MX6ULL, and how to measure latency with `cyclictest`. **PREEMPT_RT:** the Linux real-time patch set that makes more kernel paths preemptible and reduces latency.
>
> **Why:** standard Linux has a few-millisecond worst-case scheduling latency under load. That's fine for general computing but disqualifies it from motor control, audio processing, industrial PLCs, and anything that needs deterministic response. PREEMPT_RT bridges that gap. Many industrial products run PREEMPT_RT Linux today, CNCs, robotic arms, real-time camera inference.
>
> **Focus: the latency budget**. PREEMPT_RT reduces sources of kernel scheduling latency, but a configuration option does not guarantee every application deadline. Interrupt behavior, driver work, priorities, memory access and hardware interference all contribute to the bound a product must justify.

52A.1 What "real-time" means here

"Real-time" doesn't mean "fast." It means "*deterministic*." On a standard kernel, your callback runs in about 100 µs on average. But every 1000th time, it takes 5 ms because some other kernel code held a non-preemptible lock. For audio sampling at 48 kHz (20.8 µs/sample) or a motor control loop at 5 kHz (200 µs/sample), that worst case is fatal.

PREEMPT_RT trades a few percent of throughput for bounded worst case. With it enabled and tuned on i.MX6ULL Cortex-A7, you can expect:

- **Standard kernel under load**: ~100 µs typical, 5–10 ms worst case.
- **PREEMPT_RT under load**: ~30 µs typical, ~150 µs worst case.

A 30× drop in the worst case makes hard-RT applications viable.

52A.2 What PREEMPT_RT changes

Four big changes:

1. Preemptible spinlocks

In standard Linux, holding a `spin_lock` *disables preemption*. A higher-priority task can't run until the lock is released. PREEMPT_RT converts most spinlocks to "rt_mutex", sleeping locks that *can be preempted*. A high-priority task can interrupt a lower-priority task even mid-lock.

The exception: `raw_spinlock`, the few critical locks that genuinely need to disable preemption (the scheduler's own lock, IRQ disable code). These stay non-preemptible. PREEMPT_RT's correctness depends on the kernel using `raw_spinlock_t` *only* where strictly required, and `spinlock_t` (now preemptible) everywhere else. The conversion has been mostly upstreamed. What remains is a small bounded set.

> **MCU bridge:** Think of an IRQ like an EXTI/NVIC interrupt path, except Linux splits the hard interrupt from deferred work and must share lines across drivers. **IRQ:** interrupt request, the signal path that tells the CPU or interrupt controller that hardware needs service.

2. Threaded interrupts by default

Standard Linux runs IRQ handlers in IRQ context: atomic, fast, but blocking other IRQs at the same priority. PREEMPT_RT runs every IRQ handler as a kernel thread. The scheduler treats them like any other thread. A real-time thread can preempt an IRQ handler thread. SCHED_FIFO priorities determine order.

You already use `request_threaded_irq` (Ch 43). PREEMPT_RT extends this to *every* IRQ, even ones registered with `request_irq`. The primary handler becomes vestigial.

3. Priority inheritance for all mutexes

Without priority inheritance, priority inversion happens. Low-priority task A holds mutex M. High-priority task B wants M and blocks. Medium-priority task C runs and preempts A. B now waits behind C indefinitely. PREEMPT_RT's mutexes implement PI: when B blocks on M, the kernel temporarily boosts A's priority to B's. A runs through to release M, B unblocks, normal priorities restored.

This single feature avoids the Mars Pathfinder bug.

4. High-resolution timers everywhere

The `hrtimer` framework gives ns-resolution timers (Ch 55A). PREEMPT_RT makes all kernel-internal timers use hrtimer-class timing, so scheduling decisions happen with µs precision.

52A.3 Enabling PREEMPT_RT

On **v6.12 or later**, PREEMPT_RT is in mainline, no patch needed. Just enable the option in menuconfig:

```
[host]$ make menuconfig
General setup --->
    Preemption Model
        ( ) No Forced Preemption (Server)
        ( ) Voluntary Kernel Preemption (Desktop)
        ( ) Preemptible Kernel (Low-Latency Desktop)
        (X) Fully Preemptible Kernel (Real-Time)
```

On earlier kernels (v6.1 LTS, older 6.6 stable points before the merge completion) the option may be missing for some architectures and you still need the out-of-tree patch:

```
$ wget https://cdn.kernel.org/pub/linux/kernel/projects/rt/6.6/older/patch-6.6.20-
rt19.patch.gz
$ zcat patch-6.6.20-rt19.patch.gz | patch -p1
$ make ARCH=arm imx_v6_v7_defconfig
$ make menuconfig    # enable Full Preemption
$ make ARCH=arm CROSS_COMPILE=arm-none-linux-gnueabihf- zImage modules dtbs -j8
```

For new work in 2025+, target v6.12 LTS or newer and skip the patch step entirely. Boot the new kernel. `uname -a` will show `PREEMPT_RT` in the version string.

52A.4 Measuring latency with cyclictest

`cyclictest` is the standard benchmark, it spawns a high-priority thread that sleeps for a fixed interval and measures actual wake-time deviation.

```
[root@pa-mini:~]# cyclictest -t1 -p99 -i1000 -l100000
# /dev/cpu_dma_latency set to 0us
policy: fifo: loadavg: 0.21 0.06 0.02 1/53 230

T: 0 (  223) P:99 I:1000 C: 100000 Min:      6 Act:    8 Avg:    12 Max:      85
```

What this means:

- **T: 0**: thread 0.
- **P:99**: priority 99 (highest FIFO).

- **I:1000**: wakeup interval = 1000 µs (1 ms).
- **C: 100000**: 100,000 wake-ups completed.
- **Min: 6 µs**: fastest measured wake.
- **Max: 85 µs**: worst case.
- **Avg: 12 µs**: average.

For PREEMPT_RT on i.MX6ULL Cortex-A7, a tuned configuration typically gets:

- Max < 100 µs.
- Avg < 20 µs.

Standard kernel on the same hardware: max often > 5000 µs (5 ms).

Run cyclictest with system load:

```
[root@pa-mini:~]# (stress-ng --cpu 1 --io 4 --vm 1 --vm-bytes 50M --timeout 60s &) ;
cyclictest -t1 -p99 -i1000 -l60000
```

This is the test that matters, latency *under load*, not at idle.

52A.5 Configuration tuning

PREEMPT_RT alone is not enough. You also need to tune the kernel, the cmdline, and userspace:

Kernel config

- `CONFIG_HZ=1000` (already default). Higher HZ = finer scheduler granularity.
- `CONFIG_NO_HZ_FULL` for tickless operation on dedicated CPUs (more relevant for multi-core).
- Disable everything non-essential. Each enabled debug option costs latency.

Kernel cmdline

- **`isolcpus=`** (multi-core only), reserve specific CPUs for RT threads. I.MX6ULL is single-core, so not applicable.
- **`nohz_full=`**: disable timer ticks on specified CPUs.
- **`mce=off`**: disable machine-check exceptions.
- **`processor.max_cstate=0`**: disable CPU C-states (lower latency, higher idle power).

Userspace

- **Set RT priority** for your real-time thread: `pthread_setschedparam(SCHED_FIFO, prio=80)`.
- **Lock memory**: `mlockall(MCL_CURRENT | MCL_FUTURE)` to prevent page-fault latency.
- **Pre-fault pages**: write to every page of your stack at startup.

```
#include <sys/mman.h>
#include <pthread.h>
#include <sched.h>

int main(void) {
    struct sched_param p = { .sched_priority = 80 };
    pthread_setschedparam(pthread_self(), SCHED_FIFO, &p);
    mlockall(MCL_CURRENT | MCL_FUTURE);

    /* Pre-fault 256 KB of stack */
    char stack[256 * 1024];
    memset(stack, 0, sizeof(stack));

    /* Now the RT loop */
    while (1) { ... }
}
```

52A.6 Pitfalls

- **Mixing SCHED_FIFO at priority 99 with the kernel's own RT threads.** Your code can starve kernel watchdogs / scheduler maintenance. Cap user RT priorities below 90.
- **Driver still has a raw_spinlock with too much code inside it.** A long raw_spinlock critical section blocks RT. Mostly upstream code is clean. Out-of-tree drivers are the usual culprits.
- **`IRQF_NO_THREAD` flag on `request_irq`.** Forces top-half-only handling. Don't use unless absolutely necessary.
- **`spin_lock` in code that calls `kmalloc(GFP_KERNEL)`.** Works under standard Linux (kmalloc tries not to sleep). Under PREEMPT_RT, the kmalloc *can* sleep, causing weirdness. Always use `GFP_ATOMIC` in critical sections.
- **VFS / page fault latency on first access.** Read a file, the kernel may fault pages from storage. Mlockall + warmup avoids this.
- **Hardware quirks.** Some i.MX6ULL peripherals (SDMA, USB) introduce latency. Profile with `ftrace` to find the culprit.

52A.7 Key Notes

- Real-time design concerns deadlines and worst-case interference, not average throughput. PREEMPT_RT reduces sources of kernel scheduling latency but does not establish a universal deadline guarantee.
- Threaded interrupts and changed locking behavior alter where work runs and who can preempt it. Priority choices must include kernel service threads, not just the application.
- Page faults, storage, device drivers and hardware behavior remain part of the latency budget. Kernel configuration alone is not a complete timing argument.
- A latency measurement describes a specific kernel, board, workload and observation period. Its maximum is evidence for those conditions, not proof of an absolute bound.
- The i.MX6ULL has one Cortex-A7 core. Multicore isolation recipes do not create a spare core, so shared CPU demand must fit the design.

52A.8 Going deeper

- **`Documentation/locking/`**: preemptible locks under PREEMPT_RT.
- **https://wiki.linuxfoundation.org/realtime/start**: the canonical real-time Linux community wiki.
- **`Documentation/admin-guide/sysctl/kernel.rst`**: RT-throttling and related sysctls.
- **`tools/rt-tests/`**: cyclictest, oslat, hackbench source.
- **Open Source Automation Development Lab (OSADL) latency archives**: long-running latency plots across many hardware platforms.

> Next chapter: **Chapter 53: Sound (ALSA + ASoC).** Ethernet behind us, audio next: one of the most layered subsystems in the kernel, with three drivers (machine, codec, CPU-DAI) cooperating to make a single `aplay` work. **ASoC:** ALSA System-on-Chip, the embedded audio layer that connects CPU audio ports, codecs, and board wiring. **ALSA:** Linux’s kernel and user-space audio stack.

Chapter 53: Sound: ALSA and ASoC

What: the kernel's audio framework, **ALSA** (Advanced Linux Sound Architecture) and **ASoC** (ALSA System on Chip, the embedded refinement). ASoC splits an audio chain into three drivers. The **CPU-DAI** is the SoC's I²S/SAI controller. The **codec** is the analog chip (WM8960, SGTL5000). The **machine driver** wires them together for one specific board. By the end you can `aplay test.wav` over a WM8960 on the Point Atom ALPHA. **ASoC:** ALSA System-on-Chip, the embedded audio layer that connects CPU audio ports, codecs, and board wiring. **ALSA:** Linux's kernel and user-space audio stack.

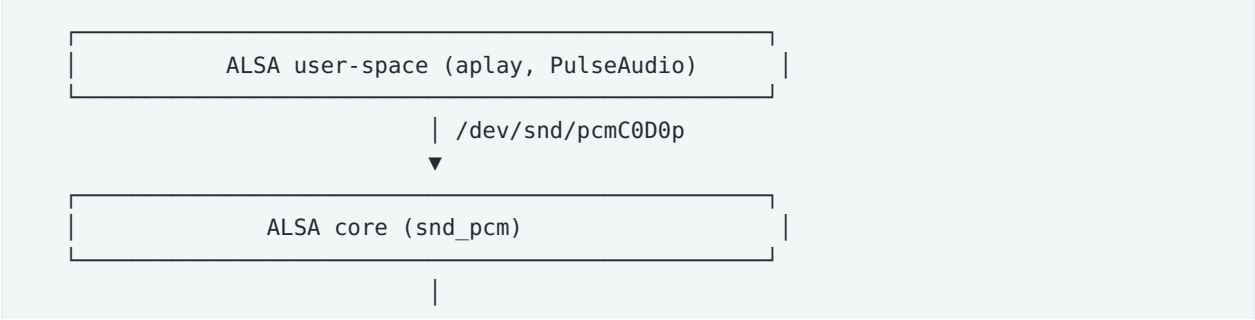
Why: audio is one of the tightest real-time loops in any embedded system. 48000 samples/second × 2 channels × 16 bits = a 96 KB/s data stream that must never glitch. ASoC's three-way split is the kernel's solution to *not* re-writing a complete audio driver for every new SoC + codec combination, you reuse the CPU-DAI and codec drivers, only writing a small machine driver per board.

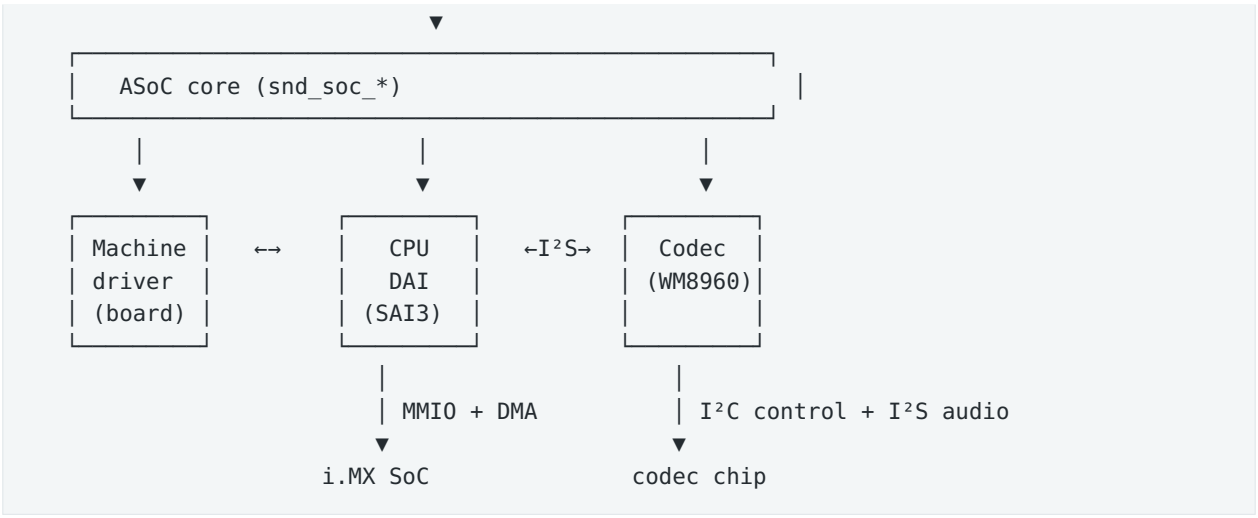
Focus: the three drivers cooperate via `snd_soc_dai_link`. The machine driver declares "CPU-DAI X drives codec Y over format Z at clock W." Once you understand this binding, ASoC drivers become readable.

Tooling. This chapter uses `alsa-utils` (`alsamixer`, `aplay`, `arecord`, `amixer`).

- **Ubuntu-base (target):** `apt install alsa-utils libasound2-dev`
- **Buildroot:** `BR2_PACKAGE_ALSA_UTILS=y BR2_PACKAGE_ALSA_LIB=y` **Buildroot:** a configuration-driven build system that produces a complete root filesystem and related images.
- Full per-tool reference: Userspace tooling appendix. **MCU bridge:** Think of the rootfs as the firmware image's file-backed runtime environment. On an MCU you link everything into flash. On Linux, programs and config live in this mounted tree. **rootfs:** root filesystem, the directory tree mounted at / that contains /bin, /etc, /dev, and libraries.

53.1 Three drivers cooperating





- **CPU-DAI**: the SoC's audio peripheral driver. For i.MX6ULL: SAI (Synchronous Audio Interface). The driver is `drivers/sound/soc/fsl/fsl_sai.c`. Speaks I²S to the codec.
- **Codec**: the codec chip's driver, e.g., `sound/soc/codecs/wm8960.c`. Talks I²C for control (volume, mute, sample rate). Receives/sends I²S audio data.
- **Machine**: the board-specific glue. E.g., `sound/soc/fsl/imx-wm8960.c`. Declares the DAI link and any board-specific routing (which speaker/headphone outputs are wired, which mics are connected).

53.2 Device tree for WM8960 audio

```
&sai2 {
    pinctrl-names = "default";
    pinctrl-0 = <&pinctrl_sai2>;
    assigned-clocks = <&clks IMX6UL_CLK_SAI2_SEL>,
                      <&clks IMX6UL_CLK_SAI2>;
    assigned-clock-parents = <&clks IMX6UL_CLK_PLL4_AUDIO_DIV>;
    assigned-clock-rates = <0>, <24576000>;
    status = "okay";
};

&i2c2 {
    wm8960: codec@1a {
        compatible = "wlf,wm8960";
        reg = <0x1a>;
        wlf,shared-lrclk;
        clocks = <&clks IMX6UL_CLK_SAI2>;
        clock-names = "mclk";
    };
};

sound {
    compatible = "fsl,imx-audio-wm8960";
    model = "wm8960-audio";
    audio-cpu = <&sai2>;
    audio-codec = <&wm8960>;
    audio-routing =
        "Headphone Jack", "HP_L",
```

```
        "Headphone Jack", "HP_R",
        "Ext Spk", "SPK_LP",
        "Ext Spk", "SPK_LN",
        "Ext Spk", "SPK_RP",
        "Ext Spk", "SPK_RN",
        "LINPUT1", "Mic Jack",
        "LINPUT3", "Mic Jack",
        "Mic Jack", "MICB";
    mux-int-port = <2>;
    mux-ext-port = <6>;
    hp-det-gpio = <&gpio5 4 GPIO_ACTIVE_HIGH>;
};
```

Three nodes:

- **`&sai2`**: the SoC's SAI2 audio peripheral. Provides the I²S link.
- **`&i2c2` / `wm8960`**: the codec on I²C2. The `clocks = <&clks IMX6UL_CLK_SAI2>` is the master clock supplying the codec.
- **`sound`**: the machine driver binding. `compatible = "fsl,imx-audio-wm8960"` picks the i.MX-WM8960 machine driver. `audio-routing` declares which codec outputs are connected to which physical jacks.

53.3 How a sample plays

```
  aplay test.wav
      │
      │ open(/dev/snd/pcmC0D0p)
      │ ioctl(SNDRV_PCM_IOCTL_HW_PARAMS, format=S16_LE, rate=48000, channels=2)
      │ write() loop ── packed into period-sized chunks ──
      │
      ▼
  ALSA core fills a periodic DMA ring buffer
      │
      │ via dmaengine – cyclic transfer (Ch 51.5)
      ▼
  SDMA continuously transfers from ring buffer → SAI2 TX FIFO
      │
      │ I²S serial clock + frame sync + data
      ▼
  WM8960 sigma-delta DAC ──→ analog audio ──→ amp ──→ speaker
```

DMA fires an IRQ every period (typically 1024 samples, about 21 ms at 48 kHz). ALSA refills that period from the user-space buffer. The cycle continues. If user-space writes fast enough to avoid an underrun, playback continues without glitches.

> **MCU bridge:** Think of an IRQ like an EXTI/NVIC interrupt path, except Linux splits the hard interrupt from deferred work and must share lines across drivers. **MCU bridge:** Think of DMA like the MCU DMA controller you used for UART or SPI, but with cache coherency, scatter-gather descriptors, and kernel ownership rules added.

> **IRQ:** interrupt request, the signal path that tells the CPU or interrupt controller that hardware needs service. **DMA:** Direct Memory Access. Hardware moves data to or from memory without the CPU copying each byte.

If userspace can't keep up: **xrun** (underrun). ALSA logs it. Sound clicks or pauses. Reasons: CPU too loaded, period too short, user-space process scheduling jitter (Ch 52A's PREEMPT_RT helps).

> **PREEMPT_RT:** the Linux real-time patch set that makes more kernel paths preemptible and reduces latency.

53.4 Writing a machine driver

The machine driver is the thing you'd write for a new board. The CPU-DAI and codec drivers come from mainline. Sketch:

```
#include <sound/soc.h>

static const struct snd_soc_dapm_widget my_dapm_widgets[] = {
    SND_SOC_DAPM_HP("Headphone Jack", NULL),
    SND_SOC_DAPM_SPK("Ext Spk", NULL),
    SND_SOC_DAPM_MIC("Mic Jack", NULL),
};

static int my_init(struct snd_soc_pcm_runtime *rtd)
{
    struct snd_soc_card *card = rtd->card;
    /* Optional: clk routing, ALC settings, etc. */
    return 0;
}

SND_SOC_DAILINK_DEFS(hifi,
    DAILINK_COMP_ARRAY(COMP_CPU("imx-sai2")),
    DAILINK_COMP_ARRAY(COMP_CODEC("wm8960.1-001a", "wm8960-hifi")),
    DAILINK_COMP_ARRAY(COMP_PLATFORM("imx-sai2")));

static struct snd_soc_dai_link my_dai_link = {
    .name = "HiFi",
    .stream_name = "HiFi",
    .init = my_init,
    SND_SOC_DAILINK_REG(hifi),
    .dai_fmt = SND_SOC_DAIFMT_I2S
             | SND_SOC_DAIFMT_NB_NF
             | SND_SOC_DAIFMT_CBS_CFS,
};

static struct snd_soc_card my_card = {
    .name = "my-board-audio",
    .owner = THIS_MODULE,
    .dai_link = &my_dai_link,
    .num_links = 1,
```

```
    .dapm_widgets = my_dapm_widgets,
    .num_dapm_widgets = ARRAY_SIZE(my_dapm_widgets),
};

static int my_probe(struct platform_device *pdev)
{
    my_card.dev = &pdev->dev;
    return devm_snd_soc_register_card(&pdev->dev, &my_card);
}
```

The DAI link declares:

- **CPU side**: which SAI peripheral.
- **Codec side**: which I²C-bound chip + which DAI on that chip (codecs may have multiple DAIs).
- **DAI format**: I²S protocol, normal bit clock + frame clock polarity (`NB_NF`), codec is bit-clock-slave + frame-clock-slave (`CBS_CFS`, meaning the SoC is the master).

That's 99% of what a machine driver does. The hard work is in the codec driver and CPU-DAI driver, which are upstream.

53.5 User-space, ALSA tools

```
[root@pa-mini:~]# aplay -l                          # list playback devices
**** List of PLAYBACK Hardware Devices ****
card 0: wm8960audio [wm8960-audio], device 0: HiFi wm8960-hifi-0 []

[root@pa-mini:~]# arecord -l                        # list capture devices
**** List of CAPTURE Hardware Devices ****
card 0: wm8960audio [wm8960-audio], device 0: HiFi wm8960-hifi-0 []

[root@pa-mini:~]# alsamixer                         # interactive mixer
[root@pa-mini:~]# aplay -D plughw:0,0 -f S16_LE -r 48000 -c 2 test.wav
[root@pa-mini:~]# arecord -D plughw:0,0 -f S16_LE -r 48000 -c 2 -d 5 rec.wav
[root@pa-mini:~]# amixer set 'Headphone' 80%
```

For higher-level audio (Bluetooth, network), you'd stack PulseAudio or PipeWire on top. For embedded one-app systems, raw ALSA is fine.

53.6 Key Notes

- ASoC separates the CPU audio interface, codec and board-specific connection. The machine description joins reusable drivers rather than replacing their responsibilities.
- Sample format, channel count, frame clocks and master-clock relationships form an end-to-end contract. A successful control-bus probe does not establish a valid audio stream.

- DMA periods and buffers trade response latency against tolerance for scheduling delays. An xrun means the producer or consumer did not meet that buffering contract.
- DAPM represents powered audio routes. Digital samples can move while the analog path remains muted or unpowered.
- Codec presence and wiring are board-specific. An ALPHA audio example is not evidence that the same codec or signal path is fitted to the MINI.

53.7 Pitfalls

- **Wrong I²S format.** Codec expects “left-justified,” driver sends “I²S”: you hear silence or noise. Fix `.dai_fmt = SND_SOC_DAIFMT_LEFT_J` etc.
- **Master/slave clock mismatch.** Both SoC and codec think they’re masters → no clock at all. One must be slave (`CBS_CFS` or `CBM_CFM`).
- **MCLK rate wrong.** Codec needs e.g., 256× sample rate as master clock. Set `assigned-clock-rates` in DT for the codec’s MCLK input.
- **DAPM routes incomplete.** Headphone outputs muted because DAPM thinks they’re not connected. Trace `cat /sys/kernel/debug/asoc/.../dapm/*`.
- **Wrong codec address.** `&i2c2 { wm8960: codec@1a { reg = <0x1a> ... } }`, verify with `i2cdetect`.
- **MASTER clock not enabled.** Codec needs MCLK to be running before any I²C command. Tie the codec’s clock to the SAI’s clock-gate so they enable together.
- **No DAPM widgets for jacks.** Audio “plays” (DMA happily transfers) but mute amps because DAPM doesn’t know to route. Always declare physical jacks as widgets.

53.8 Going deeper

- **`Documentation/sound/`**: ALSA documentation.
- **`Documentation/sound/soc/`**: ASoC documentation. Start with `overview.rst`.
- **`sound/soc/fsl/imx-wm8960.c`**: i.MX-WM8960 machine driver (~600 lines). Read this to understand a complete machine driver.
- **`sound/soc/codecs/wm8960.c`**: production codec driver (~1500 lines). Excellent example of regmap + ASoC + DAPM.
- **`sound/soc/fsl/fsl_sai.c`**: i.MX SAI CPU-DAI driver.
- **`Documentation/devicetree/bindings/sound/`**: DT bindings.

> Next chapter: **Chapter 54: LCD framebuffer and DRM/KMS.** Audio done, video next. From “Linux has /dev/fb0” to “Linux has DRM/KMS with mode-setting and Wayland-compatible output.”

Chapter 54: LCD framebuffer and DRM/KMS

> **What:** the kernel's display stack, the legacy **fbdev** (`/dev/fb0`) API and the modern **DRM/KMS** (Direct Rendering Manager / Kernel Mode Setting) framework. Both still ship. New drivers target DRM. We cover the i.MX6ULL **LCDIF** controller, the `panel-simple` driver that handles dozens of RGB-parallel panels, and how mainline kernels expose displays to user-space.
>
> **Why:** LCDIF + RGB parallel LCDs are the bread and butter of i.MX6ULL HMI products. The framework has shifted significantly in the last 5 years (away from fbdev, toward DRM) and being current matters, DRM brings page-flipping, atomic mode-setting, fence-based synchronisation, and Wayland compatibility.
>
> **Focus:** For most boards, panel-simple plus correct DT timings is enough. For 90 % of products you don't write a panel driver. You describe panel timings in DT, point at `panel-simple`, and the LCDIF driver handles the rest. The remaining 10 % is custom panels needing a chip-specific init sequence, covered in Ch 80–84 of the cookbook.

54.1 fbdev vs DRM

	fbdev	**DRM/KMS**
Era	1990s–early 2010s	2008-present
API style	mmap `/dev/fb0`, ioctl	Object-based (CRTCs, planes, encoders, connectors)
Multi-display	Awkward	Native
Atomic mode-set	No	Yes
Page-flipping	Hack	First-class
Wayland	No	Yes

	fbdev	DRM/KMS
User-space typical client	direct mmap	libdrm + Mesa / Cairo / GTK / Qt

For embedded HMI with one panel and a single fullscreen Qt app, fbdev still works. For anything with multiple outputs, GPU acceleration, or Wayland, DRM is the only option. Both APIs are present on modern kernels. Fbdev is emulated on top of DRM (`fbdev_emulation`). A fullscreen Qt app opens `/dev/fb0`, but DRM is doing the work underneath.

54.2 i.MX6ULL LCDIF

The LCDIF is a DPI / RGB-parallel controller. It outputs:

- **Pixel clock** (PCLK. 6–80 MHz typical).
- **HSYNC** (horizontal sync).
- **VSYNC** (vertical sync).
- **DE** (data enable).
- **24 data lines** (R[7:0], G[7:0], B[7:0]), typically wired as 18-bit RGB666 or 24-bit RGB888.

The driver is `drivers/gpu/drm/mxsfb/`. It's a DRM driver. Fbdev users get `/dev/fb0` via emulation.

54.3 RGB-parallel panel timings

Every RGB-parallel panel has:

- **Resolution**: e.g., 800 × 480.
- **Pixel clock frequency**: e.g., 33 MHz.
- **Horizontal porches**: `hsync-len`, `hback-porch`, `hfront-porch`.
- **Vertical porches**: `vsync-len`, `vback-porch`, `vfront-porch`.
- **Polarity bits**: HS, VS, DE, PCLK active edges.

In DT, these go in a `panel-timing` subnode:

```
panel {
    compatible = "panel-simple";
    backlight = <&backlight>;
    enable-gpios = <&gpio5 11 GPIO_ACTIVE_HIGH>;
```

```
    power-supply = <&reg_lcd_3v3>;

    port {
        panel_in: endpoint {
            remote-endpoint = <&lcdif_out>;
        };
    };

    panel-timing {
        clock-frequency = <33300000>;
        hactive = <800>;
        vactive = <480>;
        hfront-porch = <210>;
        hback-porch = <46>;
        hsync-len = <1>;
        vback-porch = <23>;
        vfront-porch = <22>;
        vsync-len = <1>;
        hsync-active = <0>;
        vsync-active = <0>;
        de-active = <1>;
        pixelclk-active = <0>;
    };
};
```

These numbers come from the panel datasheet's *AC characteristics* page. Get them right. Get them slightly wrong and you get a "rolling" image or no image at all.

The LCDIF side:

```
&lcdif {
    pinctrl-names = "default";
    pinctrl-0 = <&pinctrl_lcdif_dat>, <&pinctrl_lcdif_ctrl>;
    display = <&display0>;
    status = "okay";

    port {
        lcdif_out: endpoint {
            remote-endpoint = <&panel_in>;
        };
    };
};
```

The `port`/`endpoint` graph wires LCDIF's output to panel's input. This is the `of_graph` binding, used throughout DRM for "this output goes to that input."

54.4 panel-simple

`drivers/gpu/drm/panel/panel-simple.c` is a database-driven panel driver: it has a *database* of dozens of supported panels (vendor-id + part-number), each with a struct of timings hardcoded. For a panel that's already in the database, you just add `compatible = "vendor,partno"`:

```
panel {
    compatible = "ampire,am-480272h3tmqw-t01h", "panel-dpi";
    /* timings derived from the compatible */
};
```

If your panel isn't in the database, use `compatible = "panel-dpi"` and supply timings yourself (§54.3).

54.5 User-space usage

fbdev style (legacy)

```
[root@pa-mini:~]# ls /dev/fb*
/dev/fb0

[root@pa-mini:~]# fbset
mode "800x480-60"
    geometry 800 480 800 480 32
    timings 30030 210 46 23 22 1 1
    ...

[root@pa-mini:~]# cat /dev/urandom > /dev/fb0   # paint random pixels
```

For a Qt app: just run it with `-platform linuxfb` and Qt mmaps `/dev/fb0`.

DRM style (modern)

```
[root@pa-mini:~]# modetest -M mxsfb
Encoders:
id    crtc    type    possible crtcs    possible clones
35    33      DPI     0x00000001        0x00000000

Connectors:
id      encoder status          name            size (mm)       modes   encoders
36      35      connected       DPI-1           0x0             1       35
        modes:
            #0 800x480 60.00 800 1010 1056 1066 480 502 524 547 33300 flags: nhsync,
nvsync; type: preferred, driver

[root@pa-mini:~]# modetest -M mxsfb -s 36:800x480   # set 800x480 mode
```

For user-space drawing: libdrm + GBM (Generic Buffer Management) + a 2D library like Cairo. For Qt: `-platform eglfs` (full GPU, not on i.MX6ULL, no GPU) or `-platform linuxfb` (falls through to fbdev emulation).

54.6 Backlight

A typical RGB panel needs:

- VLED rail (12 V or boosted).
- PWM dimming via a transistor driver.

> **MCU bridge:** Think of Linux PWM like an MCU timer output channel, except the driver exposes period, duty cycle, polarity, and enable state through a subsystem. **PWM:** Pulse-Width Modulation, a timer output whose duty cycle controls average power or encodes timing.

DT:

```
backlight: backlight {
    compatible = "pwm-backlight";
    pwms = <&pwm1 0 5000000 0>;
    brightness-levels = <0 4 8 16 32 64 128 255>;
    default-brightness-level = <6>;
    power-supply = <&reg_lcd_vled>;
};
```

Ch 48 covered pwm-backlight. The LCDIF driver references it via `backlight = <&backlight>;` to coordinate power-on/off ordering.

54.7 Bring-up checklist

When a new panel doesn’t display:

1. **Verify the pixel clock is generated.** Scope PCLK. Should be at the rate in DT.
2. **Verify HSYNC and VSYNC.** Polarity matches DT (`hsync-active = <0>` = active-low).
3. **Verify data lines** with a scope. Pattern depends on what’s in `/dev/fb0`. `cat /dev/zero > /dev/fb0` gives all-zero data.
4. **Verify enable pin.** Some panels need a “panel-on” GPIO held high.

> **MCU bridge:** Think of Linux GPIO like the same pin set/reset block you used on STM32, but accessed through a kernel subsystem that owns numbering, direction, interrupts, and user-space exposure. **GPIO:** General-Purpose Input/Output, a pin controlled as a digital input, output, or interrupt source.

1. **Verify backlight.** Even if the panel is showing correctly, with backlight off you see nothing.

2. **Verify timings against the datasheet.** Off-by-one in front-porch is the most common error.

`drm.debug=15` on the kernel cmdline fills dmesg with DRM trace output. Very useful for modeset bugs.

54.8 Key Notes

- DRM/KMS separates framebuffer storage, scanout, mode selection and output connections. Legacy fbdev exposure does not describe the whole display pipeline.
- Panel timing, controller limits and board wiring must agree. A panel's advertised mode does not mean the LCDIF or memory system can sustain it.
- Panel power and backlight are separate from pixel transport. A blank screen can come from either path even when a modeset is accepted.
- Device Tree describes the physical display chain and timing requirements. Existing panel drivers are preferable when they represent the actual hardware.
- Buffer ownership and presentation timing matter above the driver as well. Drawing pixels and displaying the intended complete frame are different operations.

54.9 Pitfalls

- **Pixel clock too high.** Above about 80 MHz the LCDIF gives no output. Use lower pixel clock or smaller resolution.
- **Wrong polarity.** Image inverted vertically or shifted. Classic "porch off by one" symptom.
- **Backlight forgot.** "Display broken!", backlight is off. Always check in good ambient light first.
- **Pinmux clash.** LCDIF uses many pins (24 data + 4 control). Conflict with another peripheral muxed onto the same pin → garbled output.
- **DT enable-gpios missing or wrong polarity.** Panel power-up sequence broken.
- **Wrong DRM mode.** modetest's "preferred" mode may not be what you want. Specify explicitly.
- **`fbcon` console** drawing on top of your Qt app. Disable: `vt.global_cursor_default=0` + `fbcon=null`.

54.10 Going deeper

- **`Documentation/gpu/drm-internals.rst`**: DRM architecture.

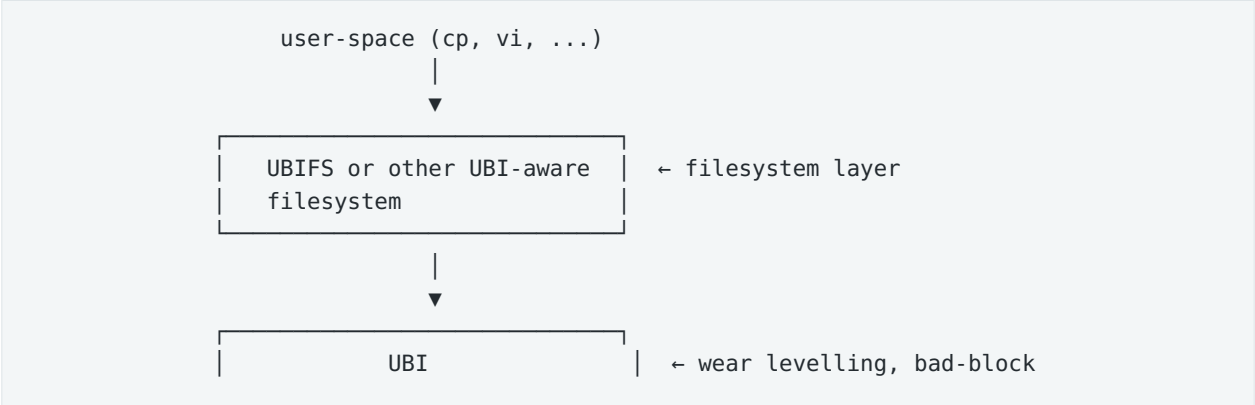
- **`Documentation/gpu/drm-kms.rst`**: KMS (Kernel Mode Setting) detailed.
- **`drivers/gpu/drm/mxsfb/`**: i.MX LCDIF DRM driver.
- **`drivers/gpu/drm/panel/panel-simple.c`**: panel database.
- **`Documentation/devicetree/bindings/display/panel/`**: panel bindings.
- **`libdrm`** at `kernel.org`, user-space DRM library and `modetest` source.

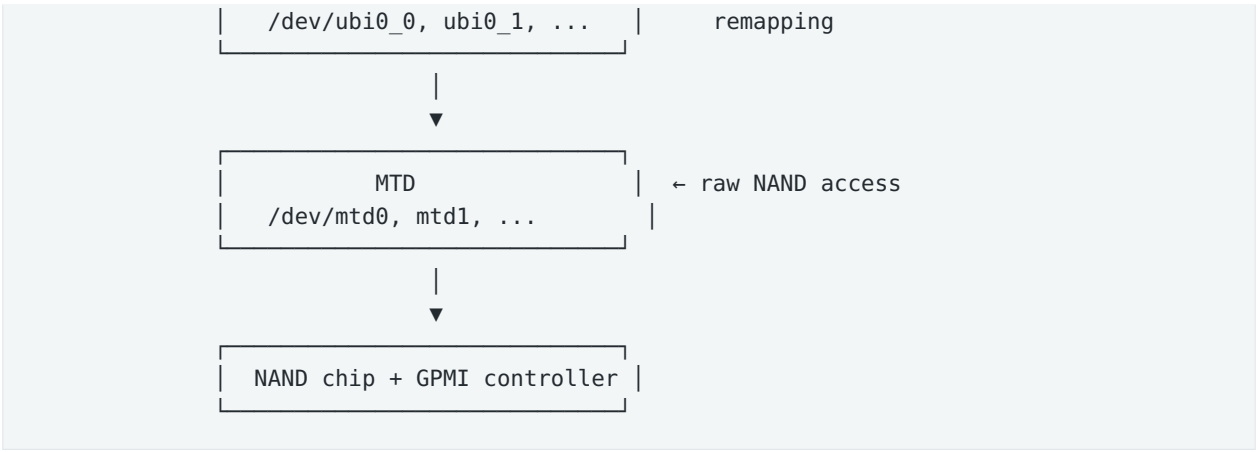
> Next chapter: **Chapter 54A: MTD/UBI for raw NAND.** When your storage isn’t an eMMC/SD card but raw NAND, MTD partitions it and UBI manages wear levelling above MTD. **UBI:** Unsorted Block Images, a flash-management layer over raw NAND that handles wear leveling and bad blocks. **MTD:** Memory Technology Device, Linux’s raw flash subsystem for eraseblock-based storage.

Chapter 54A: MTD / UBI for raw NAND

> **What:** the **MTD** (Memory Technology Devices) subsystem and the **UBI** (Unsorted Block Images) layer that sits on top of it. MTD partitions and exposes raw NAND/NOR flash to the kernel. UBI handles wear-levelling, bad-block management, and exposes UBI *volumes* that look like static block devices. **UBI:** Unsorted Block Images, a flash-management layer over raw NAND that handles wear leveling and bad blocks. **MTD:** Memory Technology Device, Linux's raw flash subsystem for eraseblock-based storage.
>
> **Why:** Raw NAND is common in industrial embedded, cheaper per GB than eMMC, longer-lived if managed correctly. But NAND is not a block device. It has erase blocks (about 128 KB) and pages (about 2 KB). Bad blocks appear over the device's lifetime. Erase cycles are limited. MTD/UBI is the kernel's solution.
>
> **Focus: the three layers**, MTD (NAND geometry), UBI (wear leveling + bad-block remapping), UBIFS (filesystem). Keep them separate in your head, each solves a different problem.
>
> **Tooling.** This chapter uses `mtd-utils` (`flash_erase`, `nandwrite`, `flashcp`, `mtdinfo`, `ubinfo`, `ubinize`, `ubiformat`).
>
> - **Ubuntu-base (target):** `apt install mtd-utils`
> - **Buildroot:** `BR2_PACKAGE_MTD=y` **Buildroot:** a configuration-driven build system that produces a complete root filesystem and related images.
> - Full per-tool reference: Userspace tooling appendix. **MCU bridge:** Think of the rootfs as the firmware image's file-backed runtime environment. On an MCU you link everything into flash. On Linux, programs and config live in this mounted tree. **rootfs:** root filesystem, the directory tree mounted at / that contains /bin, /etc, /dev, and libraries.

54A.1 Three layers





You can use MTD without UBI for read-only partitions (kernel image). For writable storage, always use UBI.

54A.2 NAND on i.MX6ULL

i.MX6ULL has the **GPMI** (General Purpose Media Interface) NAND controller with BCH error correction. Connects to standard 8-bit NAND chips.

DT:

```
&gpmi {
    pinctrl-names = "default";
    pinctrl-0 = <&pinctrl_gpmi_nand>;
    nand-on-flash-bbt;
    fsl,no-blockmark-swap;
    status = "okay";

    partition@0 {
        label = "u-boot";
        reg = <0x0 0x400000>;          /* 4 MB */
    };
    partition@1 {
        label = "kernel";
        reg = <0x400000 0x800000>;     /* 8 MB */
    };
    partition@2 {
        label = "rootfs";
        reg = <0xc00000 0x0>;          /* rest of NAND */
    };
};
```

Each partition becomes `/dev/mtdN` (and a character-device `/dev/mtdNchar`).

54A.3 MTD operations

```
[root@pa-mini:~]# cat /proc/mtd
dev:    size   erasesize  name
mtd0: 00400000 00020000 "u-boot"
mtd1: 00800000 00020000 "kernel"
```

```
mtd2: 0f400000 00020000 "rootfs"

[root@pa-mini:~]# flash_erase /dev/mtd1 0 0       # erase entire kernel partition
[root@pa-mini:~]# nandwrite -p /dev/mtd1 zImage   # write kernel image
[root@pa-mini:~]# nanddump /dev/mtd1 > backup.bin # dump
```

`mtd-utils` (`flash_erase`, `flashcp`, `nandwrite`, `nanddump`, `flash_eraseall`) is the standard toolset.

54A.4 UBI on top

For the rootfs partition, layer UBI:

```
[root@pa-mini:~]# ubiformat /dev/mtd2 -O 2048 -s 2048   # format mtd2 for UBI
[root@pa-mini:~]# ubiattach -m 2 -d 0                    # attach as ubi0
[root@pa-mini:~]# ubimkvol /dev/ubi0 -N root -m         # max-size volume named "root"
[root@pa-mini:~]# ubimkvol /dev/ubi0 -N data -s 100MiB  # 100 MB volume "data"

[root@pa-mini:~]# cat /proc/partitions
major minor  #blocks  name
   31     0     4096  mtdblock0
   31     1     8192  mtdblock1
   31     2   249856  mtdblock2
  ...
```

UBI volumes appear as `/dev/ubi0_0` (root) and `/dev/ubi0_1` (data). To user-space they look like character devices. With a UBI-aware filesystem (UBIFS) on top, they look like filesystems.

```
[root@pa-mini:~]# mkfs.ubifs -m 2048 -e 126976 -c 1000 -o root.ubifs /path/to/staging
[root@pa-mini:~]# ubiupdatevol /dev/ubi0_0 root.ubifs
[root@pa-mini:~]# mount -t ubifs ubi0:root /mnt
```

The `-m` (min I/O size, equals NAND page size), `-e` (logical eraseblock size), and `-c` (max LEB count) come from `ubinfo /dev/ubi0`.

54A.5 Boot config

Kernel cmdline:

```
ubi.mtd=2 root=ubi0:root rootfstype=ubifs
```

This says: attach mtd2 as UBI, then mount the UBI volume named “root” as the rootfs.

U-Boot:

> **MCU bridge:** Think of U-Boot like a much larger boot stub plus debug monitor: it initializes hardware, loads the next image, and gives you commands before Linux starts. **U-Boot:** the bootloader that initializes enough hardware to load and start the Linux kernel.

```
setenv bootargs 'console=ttymxc0,115200 ubi.mtd=2 root=ubi0:root rootfstype=ubifs rw'
nand read 80800000 kernel
bootz 80800000 - 81000000
```

54A.6 Wear levelling

NAND erase blocks survive ~10,000–100,000 erase cycles (chip-dependent). UBI tracks per-block erase counts and migrates rarely-erased blocks toward “hot” data, averaging wear across the whole device.

```
[root@pa-mini:~]# ubinfo -a /dev/ubi0
ubi0
Volumes count:                           2
Logical eraseblock size:                 126976 bytes, 124.0 KiB
Total amount of logical eraseblocks:     1888 (239708672 bytes, 228.6 MiB)
Amount of available logical eraseblocks: 0 (0 bytes)
Maximum count of volumes                 128
Count of bad physical eraseblocks:       3
Count of reserved physical eraseblocks:  40
Current maximum erase counter value:     145
Minimum input/output unit size:          2048 bytes
Character device major/minor:            246:0
Present volumes:                         0, 1
```

Bad blocks (3 here) are remapped to reserved blocks (40 set aside). Max erase counter is 145, far below the 10k+ chip limit, so the device is healthy.

54A.7 UBIFS, the filesystem

UBIFS is a journalling filesystem designed for UBI. Features:

- Atomic operations (power-loss safe).
- Compression (LZO or zstd), typically 1.5–2× compression of typical embedded filesystems.
- Read/write performance ~2–3× ext4-on-eMMC for typical workloads (NAND fundamentals. Ext4 not designed for NAND).

When *not* to use:

- Random small writes, UBIFS is bad at this.

- Large databases, consider a separate ext4-on-eMMC slot if you have both.

54A.8 Key Notes

- MTD exposes raw flash geometry and operations, UBI manages logical eraseblocks and wear, and UBIFS provides a filesystem above UBI. These layers solve different problems.
- Raw NAND needs ECC and bad-block handling that match the controller and chip. A filesystem cannot repair a mismatched low-level storage contract.
- Image generation, partitioning and boot arguments must agree on geometry and volume identity. A familiar `/dev/` name is not enough to identify a safe erase target.
- Wear management and power-failure recovery are design requirements, not consequences of a successful mount.
- Formatting, erase and bad-block manipulation can destroy boot or user data. Those operations belong to a qualified storage plan, not to conceptual exploration.

54A.9 Pitfalls

- **Wrong page/OOB size.** `ubiformat` defaults may not match your chip. Specify `-O 2048 -e 131072` explicitly to match `cat /proc/mtd` output.
- **Mounting UBIFS over ext4.** “Why does my UBIFS feel slow?”, UBIFS atop a block layer that’s atop NAND is double-translation. Use UBIFS directly on a UBI volume.
- **No bbt (Bad Block Table) reservation.** Pre-existing factory bad blocks become “good” in UBI’s view, then fail unpredictably. Always `nand-on-flash-bbt` or `fsl,use-bbt` in DT.
- **Erasing the wrong partition.** `flash_erase /dev/mtd0` erases U-Boot. Always confirm partition number. Back up before erase.
- **Power-loss during write.** UBIFS handles this well. Ext4-on-mtdblock does not. Use UBIFS for writable partitions.
- **`ubi.mtd=` mismatch with DT partition number.** If you add/remove partitions in DT, the numbering shifts. Kernel cmdline gets stale. Lock both at the same time.

54A.10 Going deeper

- **`Documentation/filesystems/ubifs.rst`**: UBIFS documentation.
- **`Documentation/ABI/stable/sysfs-bus-ubi`**: UBI sysfs ABI.

> **ABI:** Application Binary Interface: the calling convention, register use, binary format, and library contract that let separately built code run together. **sysfs:** a kernel-generated filesystem under /sys that exposes devices, drivers, and attributes.

- **`drivers/mtd/nand/raw/gpmi-nand/`**: i.MX GPMI NAND driver.
- **`drivers/mtd/ubi/`**: UBI implementation.
- **`mtd-utils` source**: user-space tools.
- **http://linux-mtd.infradead.org/**: the canonical MTD/UBI website.

> Next chapter: **Chapter 54B: V4L2 + GStreamer for the CSI camera.** Capture video frames from the i.MX6ULL CSI parallel camera interface and pipe them through GStreamer for processing or display.

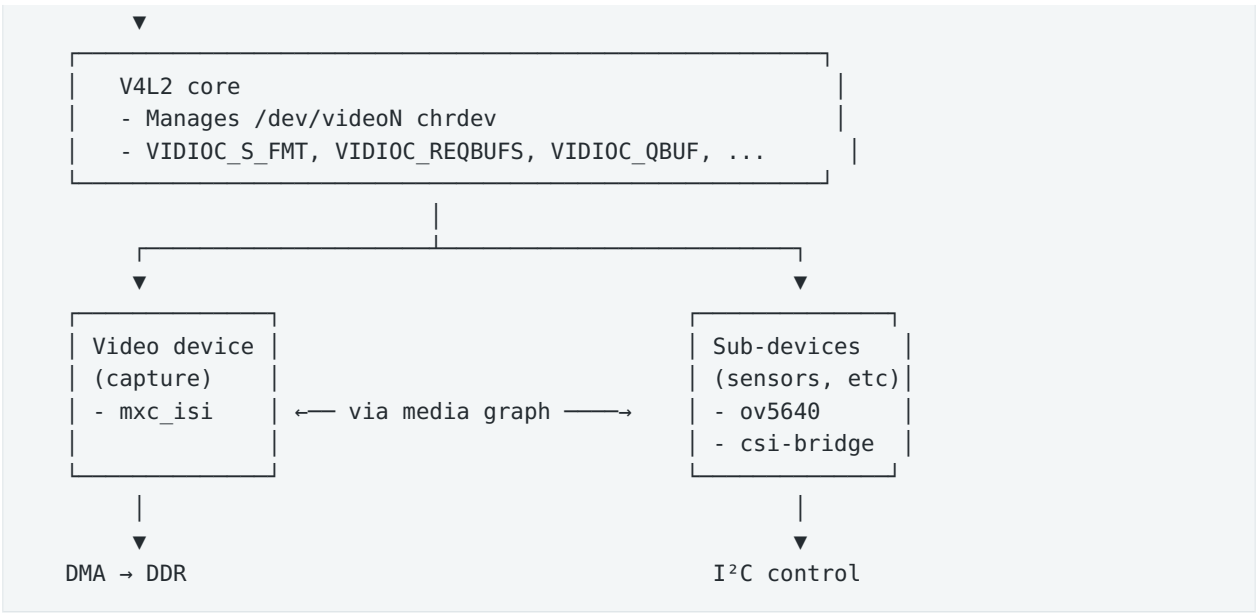
Chapter 54B: V4L2 + GStreamer

> **DMA:** Direct Memory Access. Hardware moves data to or from memory without the CPU copying each byte. **MCU bridge:** Think of DMA like the MCU DMA controller you used for UART or SPI, but with cache coherency, scatter-gather descriptors, and kernel ownership rules added. **DDR:** external DRAM that must be configured and trained before most software can run from it.

> **What:** the **V4L2** (Video for Linux 2) subsystem, the kernel framework that abstracts video capture and output devices behind `/dev/videoN`, and **GStreamer**, the user-space pipeline framework that consumes V4L2 frames. We focus on the i.MX6ULL's **CSI** (Camera Serial Interface) for parallel cameras (OV5640, OV7725, GC2145), the only camera interface on this SoC.
>
> **Why:** every IP camera, dashcam, smart-doorbell, machine-vision product runs this stack. The complexity comes from V4L2's flexibility: sub-devices (the sensor, the CSI, the IPU) compose into pipelines that can be configured at runtime. Once the pipeline model clicks, the rest of the imaging stack reads easily.
>
> **Focus: the V4L2 subdev graph + GStreamer's `v4l2src`.** The kernel exposes the capture pipeline as a graph of sub-devices. GStreamer's `v4l2src` element grabs frames from `/dev/video0`. After that, the rest is image processing.
>
> **Tooling.** This chapter uses `v4l-utils` + `gstreamer1.0-tools` + base/good/bad plugins.
>
> - **Ubuntu-base (target):** `apt install v4l-utils gstreamer1.0-tools gstreamer1.0-plugins-base gstreamer1.0-plugins-good gstreamer1.0-plugins-bad`
> - **Buildroot:** `BR2_PACKAGE_V4L_UTILS=y BR2_PACKAGE_GSTREAMER1=y BR2_PACKAGE_GST1_PLUGINS_BASE=y BR2_PACKAGE_GST1_PLUGINS_GOOD=y BR2_PACKAGE_GST1_PLUGINS_BAD=y` **Buildroot:** a configuration-driven build system that produces a complete root filesystem and related images.
> - Full per-tool reference: Userspace tooling appendix. **MCU bridge:** Think of the rootfs as the firmware image's file-backed runtime environment. On an MCU you link everything into flash. On Linux, programs and config live in this mounted tree. **rootfs:** root filesystem, the directory tree mounted at / that contains /bin, /etc, /dev, and libraries.

54B.1 V4L2 architecture

```
user-space:  GStreamer pipeline / OpenCV / your app
     │ open /dev/video0, ioctl, mmap, queue/dequeue buffers
```



A V4L2 *device* is what user-space opens (`/dev/video0`). It connects to *sub-devices*, the sensor (OV5640) and the CSI bridge (i.MX CSI/ISI), through a **media graph**. User-space sets the format (resolution, pixelformat) on both the video device and on each subdev.

54B.2 Device tree

```
&i2c2 {
    ov5640: camera@3c {
        compatible = "ovti,ov5640";
        reg = <0x3c>;
        pinctrl-names = "default";
        pinctrl-0 = <&pinctrl_csi1>;
        clocks = <&clks IMX6UL_CLK_CSI>;
        clock-names = "xclk";
        powerdown-gpios = <&gpio4 24 GPIO_ACTIVE_HIGH>;
        reset-gpios = <&gpio4 25 GPIO_ACTIVE_LOW>;

        port {
            ov5640_to_csi: endpoint {
                remote-endpoint = <&csi_from_ov5640>;
                clock-lanes = <0>;
                data-lanes = <1>;
            };
        };
    };
};

&csi {
    status = "okay";

    port {
        csi_from_ov5640: endpoint {
            remote-endpoint = <&ov5640_to_csi>;
        };
    };
};
```

Two things to notice:

- **Endpoints + remote-endpoint** form the graph. The OV5640's output endpoint links to the CSI's input endpoint, forming a media-controller pipeline.
- The OV5640 needs a master clock (XCLK) from the SoC, powerdown and reset GPIOs, and I²C for control.

54B.3 V4L2 user-space, the bare minimum

```
int fd = open("/dev/video0", O_RDWR);

/* 1. Set the desired format */
struct v4l2_format fmt = {
    .type = V4L2_BUF_TYPE_VIDEO_CAPTURE,
    .fmt.pix = {
        .width = 640, .height = 480,
        .pixelformat = V4L2_PIX_FMT_YUYV,
        .field = V4L2_FIELD_NONE,
    },
};
ioctl(fd, VIDIOC_S_FMT, &fmt);

/* 2. Request buffers */
struct v4l2_requestbuffers reqbufs = {
    .count = 4,
    .type = V4L2_BUF_TYPE_VIDEO_CAPTURE,
    .memory = V4L2_MEMORY_MMAP,
};
ioctl(fd, VIDIOC_REQBUFS, &reqbufs);

/* 3. Map them, queue them */
struct v4l2_buffer buf;
void *mappings[4];
for (int i = 0; i < 4; i++) {
    buf = (struct v4l2_buffer){
        .type = V4L2_BUF_TYPE_VIDEO_CAPTURE,
        .memory = V4L2_MEMORY_MMAP,
        .index = i,
    };
    ioctl(fd, VIDIOC_QUERYBUF, &buf);
    mappings[i] = mmap(NULL, buf.length, PROT_READ|PROT_WRITE, MAP_SHARED, fd,
buf.m.offset);
    ioctl(fd, VIDIOC_QBUF, &buf);
}

/* 4. Start streaming */
int type = V4L2_BUF_TYPE_VIDEO_CAPTURE;
ioctl(fd, VIDIOC_STREAMON, &type);

/* 5. Dequeue / process / re-queue loop */
while (1) {
    ioctl(fd, VIDIOC_DQBUF, &buf);
    process_frame(mappings[buf.index], buf.bytesused);
```

```
    ioctl(fd, VIDIOC_QBUF, &buf);
}
```

100 lines for a complete capture loop. Tedious but predictable.

For everyday work, use **libv4l2** or **GStreamer**, which wrap this.

54B.4 GStreamer in 30 seconds

GStreamer is a *pipeline* engine: elements connected with `!` pipes form a data flow.

```
# Display camera at 640x480
[root@pa-mini:~]# gst-launch-1.0 v4l2src device=/dev/video0 \
    ! video/x-raw,width=640,height=480,framerate=30/1 \
    ! videoconvert ! fbdevsink

# Record to file
[root@pa-mini:~]# gst-launch-1.0 v4l2src device=/dev/video0 \
    ! video/x-raw,width=640,height=480 \
    ! jpegenc ! avimux ! filesink location=out.avi

# Stream to network
[root@pa-mini:~]# gst-launch-1.0 v4l2src device=/dev/video0 \
    ! video/x-raw,width=640,height=480 \
    ! jpegenc ! rtpjpegpay ! udpsink host=192.168.1.100 port=5000
```

Elements:

- **`v4l2src`**: V4L2 capture source.
- **`videoconvert`**: color-space conversion (YUYV → RGB if needed).
- **`fbdevsink`**: output to `/dev/fb0`.
- **`jpegenc`**: JPEG-encode each frame.
- **`avimux`**: wrap in AVI container.
- **`rtpjpegpay`**: RTP packetize.

i.MX6ULL has no GPU/VPU, so video encoding is software (slow). Useful up to ~5–10 fps at QVGA. For higher framerates and resolutions you need a different SoC.

54B.5 Controls, exposure, gain, white balance

```
[root@pa-mini:~]# v4l2-ctl --list-ctrls
                     brightness 0x00980900 (int)    : min=0 max=255 step=1 default=128
value=128
                       contrast 0x00980901 (int)    : min=0 max=255 step=1 default=32
value=32
                     saturation 0x00980902 (int)    : min=0 max=255 step=1 default=64
value=64
```

```
...

[root@pa-mini:~]# v4l2-ctl --set-ctrl=brightness=200
[root@pa-mini:~]# v4l2-ctl --set-ctrl=exposure_auto=1     # manual
[root@pa-mini:~]# v4l2-ctl --set-ctrl=exposure_absolute=300
```

The sensor driver (ov5640) exposes a stack of controls. User-space tunes them. Auto-exposure and auto-white-balance are good enough for general use.

54B.6 Key Notes

- The capture pipeline joins sensor, interface and memory buffers through V4L2 and, where applicable, the media graph. GStreamer consumes that exposed stream in user space.
- Format and timing must agree across the pipeline. Pixel-format conversion may solve an interface mismatch at a significant CPU and memory cost.
- Queued buffers transfer ownership between application and driver. Enough buffering absorbs scheduling variation, while more buffering can add latency.
- Sensor control access does not prove pixel delivery. Power, reset, input clock and parallel bus timing remain separate dependencies.
- The i.MX6ULL’s camera path and processing budget constrain the system. A pipeline syntax example does not establish a sustained frame rate or hardware encoder capability.

54B.7 Pitfalls

- **Sensor reset/powerdown GPIO polarity.** Wrong polarity → I²C-detect fails for the sensor.

> **MCU bridge:** Think of Linux GPIO like the same pin set/reset block you used on STM32, but accessed through a kernel subsystem that owns numbering, direction, interrupts, and user-space exposure. **GPIO:** General-Purpose Input/Output, a pin controlled as a digital input, output, or interrupt source.

- **Wrong XCLK rate.** OV5640 wants 12–27 MHz. Outside that range, sensor doesn’t enumerate.
- **Pixel format mismatch.** Sensor outputs YUYV, but you request RGB. GStreamer can convert, but it costs CPU.
- **Buffer underrun.** 4 buffers minimum for smooth capture. Fewer cause dropped frames.

- **Concurrent open.** Only one user-space client per `/dev/video0`. Either GStreamer or your custom app, not both.
- **Frame size > VPU/memory budget.** 5 MP at 30 fps needs about 140 MB/s of memory bandwidth. This is close to the i.MX6ULL's practical limit.

54B.8 Going deeper

> **Driver choice:** Use the in-tree, maintained driver first. Use out-of-tree, spidev, or custom-driver paths only after you accept the kernel-version maintenance cost and document who owns updates.

- **`Documentation/userspace-api/media/`**: V4L2 user-space API.
- **`Documentation/userspace-api/media/v4l/`**: V4L2 user-space programming reference (the bible).
- **`drivers/media/i2c/ov5640.c`**: production OmniVision driver.
- **`drivers/staging/media/imx/`**: i.MX CSI/IPU drivers (historically out-of-tree).
- **https://gstreamer.freedesktop.org/documentation/**: GStreamer manual.
- **`v4l-utils` package**: `v4l2-ctl`, `media-ctl`, `qv4l2`.

> Next chapter: **Chapter 55: USB gadget.** With the host side covered earlier, the gadget side turns the i.MX6ULL into a USB device: USB mass storage, USB serial, USB Ethernet, custom HID.

Chapter 55: USB gadget

> **What:** the **USB gadget** framework, turning the i.MX6ULL's USB OTG controller into a USB *device* (instead of a host). The mainline **ConfigFS** gadget interface lets user-space compose USB devices from "functions" (mass storage, serial, Ethernet, HID) without writing kernel code.
>
> **Why:** USB gadget runs on Android phones, Raspberry Pi Zero in USB-Pi mode, smart meters that expose data over USB-serial, and many other devices. For embedded products: USB-as-device is how your board talks to a PC for debug, firmware update, or as a remote sensor.
>
> **Focus: functions composed into a configuration**. A gadget has one *configuration* with one or more *functions*. ConfigFS exposes this as a filesystem: `mkdir` a function, `echo` settings into its files, then bind to a UDC. No kernel code.

55.1 USB roles on i.MX6ULL

i.MX6ULL has 2× USB OTG controllers. Each can be:

- **Host**: Linux runs the USB host stack. Devices plug into it.
- **Device** (gadget), the SoC *is* a USB device that gets plugged into something else.
- **OTG**: auto-detect host/device via the ID pin.

Configure in DT:

```
&usbotg1 {
    dr_mode = "peripheral";        /* device-only */
    /* or "host", or "otg" */
    vbus-supply = <&reg_usb_otg1_vbus>;
    pinctrl-names = "default";
    pinctrl-0 = <&pinctrl_usbotg1>;
    status = "okay";
};
```

For "otg" mode you also wire the ID pin to a GPIO.

> **MCU bridge:** Think of Linux GPIO like the same pin set/reset block you used on STM32, but accessed through a kernel subsystem that owns numbering, direction, interrupts, and user-space exposure. **GPIO:** General-Purpose Input/Output, a pin controlled as a digital input, output, or interrupt source.

55.2 ConfigFS gadget overview

ConfigFS is the current way to compose a USB gadget. From userspace:

```
# Mount configfs
mount -t configfs none /sys/kernel/config

# Create gadget
cd /sys/kernel/config/usb_gadget
mkdir mygadget
cd mygadget

# Device descriptor
echo 0x1d6b > idVendor      # Linux Foundation
echo 0x0104 > idProduct     # Multifunction Composite Gadget
echo 0x0100 > bcdDevice
echo 0x0200 > bcdUSB        # USB 2.0

# Strings
mkdir strings/0x409          # English (US)
echo "Linuxlearn" > strings/0x409/manufacturer
echo "MyGadget"   > strings/0x409/product
echo "ABCD1234"   > strings/0x409/serialnumber

# Functions: a serial port, a network interface, mass storage
mkdir functions/acm.GS0
mkdir functions/ecm.usb0
mkdir functions/mass_storage.0
echo /dev/loop0 > functions/mass_storage.0/lun.0/file

# Configuration
mkdir configs/c.1
ln -s functions/acm.GS0          configs/c.1/
ln -s functions/ecm.usb0         configs/c.1/
ln -s functions/mass_storage.0   configs/c.1/
mkdir configs/c.1/strings/0x409
echo "Conf 1" > configs/c.1/strings/0x409/configuration
echo 250 > configs/c.1/MaxPower

# Bind to a UDC (USB Device Controller)
ls /sys/class/udc/                                  # find the udc name
echo 2184000.usb > UDC
```

The last line binds the gadget by writing the UDC name. Plug a USB cable from the i.MX6ULL's OTG port into a PC. The PC sees a composite USB device with serial, Ethernet, and mass storage.

55.3 Common function types

Function	Description	Linux usage
`acm`	CDC-ACM serial port	`/dev/ttyACM0` on host
`ecm`	CDC Ethernet	`usb0` network interface
`ncm`	NCM Ethernet (faster)	same
`rndis`	RNDIS Ethernet (Windows-compatible)	same
`mass_storage`	USB mass storage	block device on host
`hid`	HID device (keyboard, mouse, custom)	input device on host
`uvc`	UVC webcam	webcam on host
`midi`	USB MIDI	midi device on host

Each is a kernel module: `g_acm.ko`, `usb_f_ecm.ko`, etc. Built-in or modular.

55.4 Real-world examples

USB-serial console for a headless device

```
mkdir -p /sys/kernel/config/usb_gadget/console
cd $_
echo 0x1d6b > idVendor
echo 0x0104 > idProduct
mkdir strings/0x409
echo "Linuxlearn" > strings/0x409/manufacturer
echo "Headless"  > strings/0x409/product
mkdir functions/acm.0
mkdir configs/c.1
ln -s functions/acm.0 configs/c.1/
echo 2184000.usb > UDC
```

Now from the host PC: `/dev/ttyACM0` is the i.MX6ULL's UART/console.

USB Ethernet for SSH

```
mkdir -p /sys/kernel/config/usb_gadget/ssh
cd $_
echo 0x1d6b > idVendor; echo 0x0104 > idProduct
mkdir functions/ecm.usb0
echo aa:bb:cc:dd:ee:01 > functions/ecm.usb0/host_addr
echo aa:bb:cc:dd:ee:02 > functions/ecm.usb0/dev_addr
mkdir configs/c.1; ln -s functions/ecm.usb0 configs/c.1/
echo 2184000.usb > UDC
```

`usb0` appears on both sides. Assign IPs:

```
# On i.MX6ULL:
ip addr add 192.168.10.2/24 dev usb0
ip link set usb0 up

# On host PC:
ip addr add 192.168.10.1/24 dev usb0
ip link set usb0 up
ssh root@192.168.10.2
```

USB mass storage from a file

```
dd if=/dev/zero of=/tmp/disk.img bs=1M count=64
mkfs.vfat /tmp/disk.img
losetup /dev/loop0 /tmp/disk.img

mkdir -p /sys/kernel/config/usb_gadget/ms
cd $_
echo 0x1d6b > idVendor; echo 0x0104 > idProduct
mkdir functions/mass_storage.0
echo /tmp/disk.img > functions/mass_storage.0/lun.0/file
mkdir configs/c.1; ln -s functions/mass_storage.0 configs/c.1/
echo 2184000.usb > UDC
```

The host PC sees a 64 MB USB stick.

55.5 Writing a custom function

For specialised use cases (e.g., a custom protocol over USB), you can write a kernel function driver. But before doing so, check whether **FunctionFS** (a userspace-driven generic function) fits your needs. With FunctionFS, your gadget function lives in user space, the kernel just relays bytes between endpoints and your daemon. Much less work than a kernel function.

55.6 Key Notes

- A USB gadget is the device side of a connection, not a USB host. The controller role, connector wiring and VBUS handling must support that role.
- ConfigFS composes functions into configurations and binds the result to a USB device controller. Descriptor identity and host compatibility are part of the interface contract.
- Only one gadget can own a controller at a time. Reconfiguration requires an explicit disconnect and rebinding lifecycle.
- Mass-storage export gives the host access to a backing store. Host and target must not independently mount and modify the same filesystem at the same time.
- A composite gadget combines useful interfaces but also exposes more authority. Recovery, networking and storage functions need separate access and data-ownership decisions.

55.7 Pitfalls

- **Forgetting `vbus-supply`.** USB device mode still needs VBUS sensing. Without it, the controller never detects a “plugged” event.
- **`dr_mode = "host"` when you want device.** Symptom: nothing plugs in on the host. Always check.
- **Wrong VID/PID.** Windows binds drivers based on these. Wrong values → host loads wrong driver → device fails to enumerate.
- **`mass_storage` backing file is too small / wrong format.** Host complains about corrupted filesystem.
- **Two gadgets bound to one UDC.** Only one bind per UDC at a time. Disconnect first.
- **ACM not appearing on Windows.** Windows needs a `.inf` driver file before it will bind to CDC-ACM. Linux/macOS hosts are fine.

55.8 Going deeper

- **`Documentation/usb/gadget_configfs.rst`**: the canonical ConfigFS gadget doc.
- **`drivers/usb/gadget/`**: gadget framework and functions.
- **`drivers/usb/gadget/function/`**: individual function drivers (one per `.c` file).
- **`Documentation/usb/functionfs.rst`**: FunctionFS for user-space-driven gadgets.

- **`Documentation/devicetree/bindings/usb/`**: USB controller bindings.

> Next chapter: **Chapter 55A: Kernel timers + hrtimers.** Beyond `mdelay` / `msleep`, the kernel offers precise timers for scheduling delayed work and periodic actions.

Chapter 55A: Kernel timers and hrtimers

> **What:** the kernel's two timer families, **timer_list** (jiffies-granular, ~1 ms on i.MX6ULL with HZ=1000) and **hrtimer** (high-resolution, nanosecond-granular). Used for "do X in N ms" patterns inside drivers, debouncing buttons, polling status, scheduling periodic samples.
>
> **Why:** `msleep` and `mdelay` block the calling thread. Sometimes you need "fire a callback in 50 ms without blocking", that's what these timers are for. Common driver patterns use them: timeouts, periodic polling, rate limiting, deferred work.
>
> **Focus: timer_list for ms granularity, hrtimer for µs/ns**.

55A.1 timer_list

The classic jiffies-based timer:

```
#include <linux/timer.h>

struct timer_list my_timer;

static void my_timer_cb(struct timer_list *t)
{
    /* runs in softirq context — no sleeping! */
    pr_info("timer fired\n");
}

/* In probe: */
timer_setup(&my_timer, my_timer_cb, 0);

/* Schedule: fire 100 ms from now */
mod_timer(&my_timer, jiffies + msecs_to_jiffies(100));

/* Cancel: */
del_timer_sync(&my_timer);
```

- **Granularity**: 1 jiffy ≈ 1 ms (HZ=1000).
- **Context**: callback runs in softirq context (atomic). No sleeping, no `mutex_lock`, no `kmalloc(GFP_KERNEL)`.
- **`mod_timer`**: re-schedules a running timer to a new expiry. If the timer is not active, it arms it.
- **`del_timer_sync`**: deletes and waits for any in-flight callback to finish. Use in `remove`.

Typical pattern, button debounce (Ch 45 example):

```
static void debounce_cb(struct timer_list *t)
{
    struct my_button *b = from_timer(b, t, debounce_timer);
    int val = gpiod_get_value(b->gpio);
    input_report_key(b->input, KEY_ENTER, val);
    input_sync(b->input);
}

static irqreturn_t button_irq(int irq, void *dev_id)
{
    struct my_button *b = dev_id;
    mod_timer(&b->debounce_timer, jiffies + msecs_to_jiffies(20));
    return IRQ_HANDLED;
}
```

Each press triggers an IRQ that re-arms the timer. If the button bounces, every bounce resets the timer. Only after 20 ms of silence does the timer fire and report the press.

> **MCU bridge:** Think of an IRQ like an EXTI/NVIC interrupt path, except Linux splits the hard interrupt from deferred work and must share lines across drivers. **IRQ:** interrupt request, the signal path that tells the CPU or interrupt controller that hardware needs service.

55A.2 hrtimer

When you need µs precision:

```
#include <linux/hrtimer.h>

struct hrtimer my_hrtimer;

static enum hrtimer_restart my_hr_cb(struct hrtimer *t)
{
    /* still softirq context */
    pr_info("hrtimer fired\n");
    /* For periodic: */
    hrtimer_forward_now(t, ms_to_ktime(100));
    return HRTIMER_RESTART;
    /* For one-shot: return HRTIMER_NORESTART; */
}

/* In probe: */
hrtimer_init(&my_hrtimer, CLOCK_MONOTONIC, HRTIMER_MODE_REL);
my_hrtimer.function = my_hr_cb;

/* Arm: fire 500 µs from now */
hrtimer_start(&my_hrtimer, us_to_ktime(500), HRTIMER_MODE_REL);
```

```
/* Cancel: */
hrtimer_cancel(&my_hrtimer);
```

- **Granularity**: nanoseconds, limited by hardware (i.MX6ULL's GPT has ~30 ns resolution).
- **Context**: same, softirq, no sleeping.
- **Periodic loops**: return `HRTIMER_RESTART` after `hrtimer_forward_now`. It drifts less than recomputing `now + 1ms` each time.

Common pattern, periodic sampling at 1 kHz:

```
hrtimer_init(&p->hr, CLOCK_MONOTONIC, HRTIMER_MODE_REL);
p->hr.function = sample_cb;
hrtimer_start(&p->hr, ms_to_ktime(1), HRTIMER_MODE_REL);
```

Each callback: do the work, then `hrtimer_forward_now(t, ms_to_ktime(1)); return HRTIMER_RESTART;`. The forwarding (vs. recomputing now+1ms) keeps the schedule cumulative without drift.

55A.3 Workqueue vs timer

Both are deferred mechanisms but in different contexts:

	timer/hrtimer	**workqueue**
Callback context	softirq (atomic)	process (sleeping OK)
Precision	high	low (~1 ms typical, more under load)
Use for	timeouts, fast polling	longer work that may sleep

Combine them: timer fires (atomic), schedules a workqueue (process context for the heavy work).

```
static void my_timer_cb(struct timer_list *t)
{
    struct my_priv *p = from_timer(p, t, timer);
    schedule_work(&p->work);
    mod_timer(&p->timer, jiffies + msecs_to_jiffies(100));  /* re-arm */
}

static void my_work_fn(struct work_struct *w)
{
```

```
    struct my_priv *p = container_of(w, struct my_priv, work);
    /* heavy work, can sleep */
}
```

55A.4 Delayed work, a workqueue with built-in timer

For the common “do this in N ms, in process context” pattern:

```
#include <linux/workqueue.h>

struct delayed_work my_dwork;

static void my_dwork_fn(struct work_struct *w)
{
    /* process context — can sleep */
}

INIT_DELAYED_WORK(&my_dwork, my_dwork_fn);

schedule_delayed_work(&my_dwork, msecs_to_jiffies(100));
cancel_delayed_work_sync(&my_dwork);
```

`delayed_work` combines a timer + workqueue. Most “do this later” use cases want this rather than raw timers.

55A.5 Key Notes

- Kernel timers schedule callbacks, not private threads. Their execution context determines which operations are legal, while workqueues provide a place for work that may sleep.
- Timer resolution and observed callback latency are different quantities. A high-resolution timer does not make GPIO transitions or periodic work perfectly deterministic.
- Periodic scheduling needs an explicit phase and overrun policy. Re-arming relative to completion and maintaining an intended time sequence have different drift behavior.
- Cancellation is part of object lifetime. Removal must prevent callbacks or deferred work from accessing released device state.
- A hardware timer, PWM block or DMA engine can be a better owner of precise signal timing than a software callback.

55A.6 Pitfalls

- **Sleeping in a timer callback.** Softirq context, `kmalloc(GFP_KERNEL)`, `mutex_lock` are forbidden. Use workqueue if you need to sleep.

- **`del_timer` without `_sync` in remove.** Race: timer fires during cleanup. Always use `del_timer_sync`.
- **Forgetting to re-arm a periodic timer.** It fires once and stops. Either `mod_timer` in the callback or use `delayed_work`.
- **hrtimer drift with manual re-arming.** `hrtimer_start(t, now + 1ms)` drifts because of callback latency. `hrtimer_forward_now(t, 1ms)` doesn't.
- **Timer fires after device is unregistered.** Without `del_timer_sync`, the callback can run after probe-cleanup, touching freed memory. Synchronize cleanup.
- **Too many timers in flight.** Each adds to the kernel timer wheel. Thousands cost CPU. For mass-scheduled events, consider a single timer + a list of work items.

55A.7 Going deeper

- **`Documentation/timers/`**: kernel timer documentation.
- **`include/linux/timer.h`** and **`hrtimer.h`**, APIs.
- **`kernel/time/`**: implementation. `hrtimer.c`, `timer.c`.
- **LDD3 Chapter 7**: timers and workqueues in depth.

> Next chapter: **Chapter 55B: Async notification via SIGIO.** A "signal me when data is ready" mechanism that's useful for legacy POSIX-style apps that don't use poll/epoll.

Chapter 55B: Async notification (SIGIO)

> **IRQ:** interrupt request, the signal path that tells the CPU or interrupt controller that hardware needs service. **MCU bridge:** Think of an IRQ like an EXTI/NVIC interrupt path, except Linux splits the hard interrupt from deferred work and must share lines across drivers.

> **What:** the **fasync / SIGIO** mechanism, a driver tells its user-space process "data is ready" by sending it a Unix signal, instead of the process polling or blocking on read. The driver implements `.fasync` in `file_operations`. User-space arms it with `fcntl(fd, F_SETOWN, getpid()); fcntl(fd, F_SETFL, O_ASYNC);`.
>
> **Why:** for very rare events ("button pressed" or "thermal alarm"), forcing user-space to `poll()` continuously or block on `read()` is wasteful. SIGIO lets the application do other things. The signal arrives when something happens. Linux's input layer doesn't use it (poll/epoll is preferred), but legacy POSIX-style apps and some custom devices do.
>
> **Focus: registers a process for delivery. Driver triggers**. The driver maintains a `fasync_struct`. When an event happens, it calls `kill_fasync` and the kernel sends the registered process a SIGIO.

55B.1 Driver side

Add `.fasync` to your `file_operations`:

```
#include <linux/fs.h>

struct my_priv {
    struct fasync_struct *fasync;
    /* ... */
};

static int my_fasync(int fd, struct file *filp, int mode)
{
    struct my_priv *p = filp->private_data;
    return fasync_helper(fd, filp, mode, &p->fasync);
}

static const struct file_operations my_fops = {
    .owner   = THIS_MODULE,
    /* ... */
    .fasync  = my_fasync,
};

/* When an event occurs (e.g., IRQ → data ready): */
```

```
static void notify_userspace(struct my_priv *p)
{
    if (p->fasync)
        kill_fasync(&p->fasync, SIGIO, POLL_IN);
}

/* In release(): */
static int my_release(struct inode *inode, struct file *filp)
{
    my_fasync(-1, filp, 0);    /* clean up the fasync registration */
    return 0;
}
```

`fasync_helper(fd, filp, mode, &list)` manages the list of registered processes. `kill_fasync(&list, signal, band)` sends the signal to every registered process.

55B.2 User-space side

```
#include <fcntl.h>
#include <signal.h>
#include <unistd.h>

static volatile sig_atomic_t got_event;

static void on_sigio(int sig)
{
    got_event = 1;
}

int main(void)
{
    int fd = open("/dev/myevent", O_RDONLY);

    /* Register our signal handler */
    struct sigaction sa = {0};
    sa.sa_handler = on_sigio;
    sigaction(SIGIO, &sa, NULL);

    /* Tell the kernel: send signals to *this* process */
    fcntl(fd, F_SETOWN, getpid());

    /* Enable async notification */
    int flags = fcntl(fd, F_GETFL);
    fcntl(fd, F_SETFL, flags | O_ASYNC);

    while (1) {
        pause();    /* sleep until any signal arrives */
        if (got_event) {
            char buf[64];
            int n = read(fd, buf, sizeof(buf) - 1);
            buf[n] = 0;
            printf("event: %s\n", buf);
            got_event = 0;
        }
```

```
    }
}
```

Three steps. First, install the signal handler. Second, `F_SETOWN` tells the kernel which process gets the signal. Third, `F_SETFL | O_ASYNC` enables delivery.

55B.3 When to use SIGIO (and when not to)

Use when:

- The event is rare (≤ a few per second).
- The application is already signal-handling (legacy code).
- Polling overhead would be wasteful.

Don’t use when:

- Events come at > ~100 Hz (signal delivery is expensive).
- You can use `poll()` / `select()` / `epoll` instead, those are more efficient and don’t have signal-handler restrictions.
- You need to know *which* fd fired (SIGIO carries the band but not the fd unless you use `F_SETSIG` for realtime signals with `siginfo`).

For modern code, `poll()` / `epoll` are preferred for everything. SIGIO is worth knowing about, but rarely the right tool today.

55B.4 Key Notes

- SIGIO is a notification that device state may need attention, not the data itself. The application’s queue or read interface remains the source of events.
- Signal ownership, enabling asynchronous delivery and device release form one registration lifecycle. Stale registrations can outlive the intended recipient.
- Ordinary signals can merge. A recipient must drain available work rather than assume one signal corresponds to one hardware event.
- Signal-handler context restricts what the application can safely do. Complex processing belongs outside the handler.
- `poll()` and `epoll()` also provide event-driven waiting without continuous busy polling. The choice is an application interface decision, not a requirement to use signals for efficiency.

55B.5 Pitfalls

- **Signal handler doing too much.** Signal handlers run in arbitrary context. Only async-signal-safe functions allowed. Set a flag. Do real work in the main loop.
- **Forgetting `F_SETOWN`.** Without it, the kernel doesn't know who to signal.
- **Race: signal arrives before sigaction installed.** Install handler before opening the device.
- **Signal merging.** If your driver fires SIGIO twice before user-space handles either, only one delivery happens. User-space must drain everything that was queued, not just handle one event.
- **`fasync_helper(-1, ...)` not called in release.** Stale fasync entry. Later signals go to a dead pid.

55B.6 Going deeper

- The relevant kernel docs are sparse, see LDD3 Chapter 6 and `man 2 fcntl` instead.
- **`drivers/char/`**: many older char drivers use SIGIO.
- **`kernel/signal.c`**, **`fs/fcntl.c`**, implementation.

> Next chapter: **Chapter 55C: CAN + FlexCAN.** SocketCAN, kernel framework, FlexCAN driver, transceivers.

Chapter 55C: CAN bus (SocketCAN + FlexCAN)

> **What: SocketCAN**, Linux's abstraction that exposes a CAN interface as a network device (`can0`) and CAN frames as `struct sockaddr_can` / `struct can_frame` over a normal socket. The **FlexCAN** driver covers i.MX6ULL's 2 FlexCAN controllers. User-space speaks the socket API. By the end you can `cansend can0 123#DEADBEEF` and watch the frame on a scope.
>
> **Why:** CAN is the dominant bus in automotive and a strong second in industrial automation. The SocketCAN abstraction means you write CAN apps with `socket()` / `sendto()` / `recvmsg()`, same APIs as TCP/UDP. No proprietary library. Tools work across all CAN hardware on Linux.
>
> **Focus: CAN looks like a network device.** Once `can0` is "up," everything is generic, Wireshark, tcpdump-equivalent (`candump`), `iproute2` configuration, even SO_TIMESTAMP for nanosecond-accurate receive timestamps.
>
> **Tooling.** This chapter uses `can-utils` + `iproute2` (`ip link set canX type can ...`).
>
> - **Ubuntu-base (target):** `apt install can-utils iproute2`
> - **Buildroot:** `BR2_PACKAGE_CAN_UTILS=y BR2_PACKAGE_IPROUTE2=y` **Buildroot:** a configuration-driven build system that produces a complete root filesystem and related images.
> - Full per-tool reference: Userspace tooling appendix. **MCU bridge:** Think of the rootfs as the firmware image's file-backed runtime environment. On an MCU you link everything into flash. On Linux, programs and config live in this mounted tree. **rootfs:** root filesystem, the directory tree mounted at / that contains /bin, /etc, /dev, and libraries.

55C.1 CAN basics

CAN 2.0 is a differential, multi-master, contention-resolved bus:

- 2 wires (CAN_H, CAN_L), 60 Ω termination each end.
- 1 Mbps max (high-speed CAN).
- 11-bit (standard) or 29-bit (extended) frame identifier.
- 0–8 data bytes per frame.

CAN-FD (Flexible Data-rate) extends payload to 64 bytes and allows 5 Mbps data-phase. I.MX6ULL FlexCAN supports CAN-FD on the newer revisions.

Physical layer needs a *transceiver* between SoC and bus: TJA1051 (5V), TJA1463 (CAN-FD), MCP2562. The SoC speaks 3.3V TTL CAN_TX/CAN_RX. The transceiver speaks differential CAN_H/CAN_L.

55C.2 i.MX FlexCAN in DT

```
&can1 {
    pinctrl-names = "default";
    pinctrl-0 = <&pinctrl_flexcan1>;
    xceiver-supply = <&reg_can_3v3>;
    status = "okay";
};

&can2 {
    pinctrl-names = "default";
    pinctrl-0 = <&pinctrl_flexcan2>;
    status = "okay";
};
```

Once enabled, `can0` and `can1` appear:

```
[root@pa-mini:~]# ip link show | grep can
3: can0: <NOARP,ECHO> mtu 16 qdisc noop state DOWN ...
4: can1: <NOARP,ECHO> mtu 16 qdisc noop state DOWN ...
```

55C.3 Bringing up the interface

```
[root@pa-mini:~]# ip link set can0 type can bitrate 500000
[root@pa-mini:~]# ip link set can0 up
[root@pa-mini:~]# ip -s link show can0
3: can0: <NOARP,UP,LOWER_UP,ECHO> mtu 16 qdisc fq_codel state UP qlen 10
    link/can
    RX: bytes  packets  errors  dropped overrun mcast
    0          0        0       0       0       0
    TX: bytes  packets  errors  dropped carrier collsns
    0          0        0       0       0       0
```

Send and receive:

```
[root@pa-mini:~]# cansend can0 123#11.22.33.44       # send frame, ID 0x123
[root@pa-mini:~]# candump can0                       # passive monitor
  can0  123   [4]  11 22 33 44

# CAN-FD with bit-rate switch:
[root@pa-mini:~]# ip link set can0 type can bitrate 500000 dbitrate 2000000 fd on
[root@pa-mini:~]# cansend can0 123##1.11.22.33.44.55.66.77.88
```

55C.4 Programming with sockets

```
#include <sys/socket.h>
#include <linux/can.h>
#include <linux/can/raw.h>
#include <net/if.h>

int sock = socket(PF_CAN, SOCK_RAW, CAN_RAW);

struct ifreq ifr;
strncpy(ifr.ifr_name, "can0", IFNAMSIZ);
ioctl(sock, SIOCGIFINDEX, &ifr);

struct sockaddr_can addr = {
    .can_family = AF_CAN,
    .can_ifindex = ifr.ifr_ifindex,
};
bind(sock, (struct sockaddr *)&addr, sizeof(addr));

/* Send */
struct can_frame frame = {
    .can_id  = 0x123,
    .can_dlc = 4,
    .data    = { 0x11, 0x22, 0x33, 0x44 },
};
write(sock, &frame, sizeof(frame));

/* Receive */
struct can_frame rxframe;
int n = read(sock, &rxframe, sizeof(rxframe));
printf("rx id=%x dlc=%d\n", rxframe.can_id, rxframe.can_dlc);
```

For filtering, only receive frames with specific IDs:

```
struct can_filter rfilter[2];
rfilter[0].can_id   = 0x123;
rfilter[0].can_mask = CAN_SFF_MASK;
rfilter[1].can_id   = 0x200;
rfilter[1].can_mask = 0x700;     /* match 0x200–0x2FF */
setsockopt(sock, SOL_CAN_RAW, CAN_RAW_FILTER, &rfilter, sizeof(rfilter));
```

The kernel filters in software, or in hardware where the controller supports it. FlexCAN has message-buffer (MB) filtering. High-throughput receivers should always set filters. Otherwise, every frame on the bus is delivered to every socket.

55C.5 Higher protocols

CAN-RAW is the bottom layer. Real applications use one of:

- **ISO-TP** (ISO-15765-2), fragmentation/reassembly for >8-byte payloads. `linux/can/isotp.h`. Used by OBD-II and UDS automotive diagnostics.
- **J1939**: heavy-duty truck/agricultural protocol. `linux/can/j1939.h`.

- **CAN BCM** (Broadcast Manager), kernel-side periodic frame TX/RX with filtering. Reduces user-space wakeups.

```
/* ISO-TP socket */
int sock = socket(PF_CAN, SOCK_DGRAM, CAN_ISOTP);
struct sockaddr_can addr = {
    .can_family = AF_CAN,
    .can_ifindex = ifr.ifr_ifindex,
    .can_addr.tp = { .tx_id = 0x7E0, .rx_id = 0x7E8 },
};
bind(sock, (struct sockaddr *)&addr, sizeof(addr));
write(sock, "Hello, this is an ISO-TP message longer than 8 bytes", 53);
```

55C.6 Error frames and bus health

CAN reports “bus-off” when error count exceeds 255. SocketCAN exposes these as special error frames with `can_id` flag `CAN_ERR_FLAG`. Enable:

```
can_err_mask_t err_mask = CAN_ERR_TX_TIMEOUT | CAN_ERR_LOSTARB | CAN_ERR_CRTL |
CAN_ERR_PROT | CAN_ERR_TRX | CAN_ERR_BUSOFF;
setsockopt(sock, SOL_CAN_RAW, CAN_RAW_ERR_FILTER, &err_mask, sizeof(err_mask));
```

When the bus goes bus-off, restart it:

```
ip link set can0 type can restart-ms 100      # auto-restart 100 ms after bus-off
```

Or do it manually with `ip link set can0 down. ip link set can0 up;` after sorting the wiring/termination.

55C.7 Key Notes

- SocketCAN exposes CAN through Linux networking while FlexCAN owns controller behavior. The external transceiver and bus wiring remain essential physical dependencies.
- Bit timing and termination are shared bus contracts. A local loopback result does not establish correct communication with physical nodes.
- CAN arbitration, filtering and higher-level protocols solve different problems. ISO-TP adds transport behavior above frames rather than changing the electrical bus.
- Bus-off indicates a fault boundary requiring a deliberate recovery policy. Automatically restarting without addressing a persistent fault can repeat the failure.
- Classic CAN on the i.MX6ULL is not CAN FD. Protocol and controller capabilities must match every node on the intended segment.

55C.8 Pitfalls

- **Missing/wrong terminations.** Without 120 Ω termination at each end of the bus (60 Ω total), signal integrity collapses. For a single-node bench setup, put one 120 Ω resistor across CAN_H/CAN_L. Robustness is lower but it works.
- **Wrong bitrate on one end.** Symptom: all frames error. Both sides must agree.
- **CAN_TX/RX swapped at transceiver.** Symptom: no transmit. Verify schematic.
- **No transceiver supply.** Many transceivers need their own VCC. Without it, no signaling.
- **Bus-off and no restart-ms.** Bus stuck off after first error storm. Set `restart-ms`.
- **Different CAN-FD speeds.** Old nodes can't handle CAN-FD speed-shift frames. Bus collapses. Use CAN-FD only on segments where all nodes support it.

55C.9 Going deeper

- **`Documentation/networking/can.rst`**: SocketCAN documentation.
- **`drivers/net/can/flexcan.c`**: i.MX FlexCAN driver.
- **`can-utils`**: `cansend`, `candump`, `cangen`, `canbusload`, `isotpdump`. Required.
- **`linux/can/isotp.h`**, **`linux/can/j1939.h`**, higher-protocol headers.
- **OpenXC** project, open-source automotive data over CAN.

> Next chapter: **Chapter 55D: Block device drivers.** The other half of "storage", `gendisk`, request queues, blk-mq.

Chapter 55D: Block device drivers

> **What:** the **block** layer, `gendisk`, request queues, `blk-mq` (multi-queue block), bio. Most embedded systems consume block devices (eMMC, SD card via MMC subsystem. Raw NAND via MTD+UBI). Occasionally you need to *write* one, typically a RAM disk, a loop-style virtual device, or a translation layer over a custom storage chip. **UBI:** Unsorted Block Images, a flash-management layer over raw NAND that handles wear leveling and bad blocks. **MTD:** Memory Technology Device, Linux's raw flash subsystem for eraseblock-based storage.
>
> **Why:** less common to write than char drivers, but worth knowing because (a) the request-queue model differs significantly from "byte stream" and (b) understanding block lets you debug performance of any storage layer above (filesystem latency, fsync behavior).
>
> **Focus: bio is the universal request**. A user-space `read(fd, ..., 4096)` becomes one or more `struct bio`s submitted to a `gendisk`. Drivers either submit one I/O per bio or batch into hardware-specific request structures.

55D.1 The path from read() to your driver

```
user-space: read(fd, buf, 8192)
     |
VFS: dispatches via filesystem (ext4, ubifs, ...) → submit_bio()
     |
block layer: builds a struct bio, schedules
     |
blk-mq: per-CPU multi-queue; merges adjacent bios when possible
     |
driver: receives a queue_rq() callback per request
     |
hardware
```

`bio` describes: which device, what direction (read/write), what sector range, and a vector of memory buffers. Drivers can process bios one at a time, or convert them to hardware-native request structures.

55D.2 Minimal: RAM disk

```
#include <linux/blk-mq.h>
#include <linux/blkdev.h>
#include <linux/genhd.h>

#define DISK_SIZE_MB 64
#define SECTOR_SIZE  512
```

```
struct ramdisk {
    void *data;
    sector_t capacity_sectors;
    struct gendisk *disk;
    struct blk_mq_tag_set tag_set;
    struct request_queue *queue;
};

static blk_status_t ramdisk_queue_rq(struct blk_mq_hw_ctx *hctx,
                                     const struct blk_mq_queue_data *bd)
{
    struct request *req = bd->rq;
    struct ramdisk *r = req->q->queuedata;
    struct bio_vec bvec;
    struct req_iterator iter;
    sector_t pos = blk_rq_pos(req);

    blk_mq_start_request(req);

    rq_for_each_segment(bvec, req, iter) {
        void *user_buf = kmap_local_page(bvec.bv_page) + bvec.bv_offset;
        void *disk_buf = r->data + pos * SECTOR_SIZE;

        if (rq_data_dir(req) == READ)
            memcpy(user_buf, disk_buf, bvec.bv_len);
        else
            memcpy(disk_buf, user_buf, bvec.bv_len);

        kunmap_local(user_buf);
        pos += bvec.bv_len / SECTOR_SIZE;
    }

    blk_mq_end_request(req, BLK_STS_OK);
    return BLK_STS_OK;
}

static const struct blk_mq_ops ramdisk_mq_ops = {
    .queue_rq = ramdisk_queue_rq,
};

static const struct block_device_operations ramdisk_fops = {
    .owner = THIS_MODULE,
};

static int __init ramdisk_init(void)
{
    struct ramdisk *r = kzalloc(sizeof(*r), GFP_KERNEL);
    if (!r) return -ENOMEM;

    r->capacity_sectors = (DISK_SIZE_MB * 1024 * 1024) / SECTOR_SIZE;
    r->data = vmalloc(DISK_SIZE_MB * 1024 * 1024);
    if (!r->data) { kfree(r); return -ENOMEM; }

    r->tag_set.ops          = &ramdisk_mq_ops;
    r->tag_set.nr_hw_queues = 1;
    r->tag_set.queue_depth  = 128;
    r->tag_set.numa_node    = NUMA_NO_NODE;
    r->tag_set.cmd_size     = 0;
```

```
    r->tag_set.flags        = BLK_MQ_F_SHOULD_MERGE;
    r->tag_set.driver_data  = r;
    blk_mq_alloc_tag_set(&r->tag_set);

    r->disk = blk_mq_alloc_disk(&r->tag_set, r);
    r->queue = r->disk->queue;
    r->queue->queuedata = r;

    strscpy(r->disk->disk_name, "myram0", DISK_NAME_LEN);
    r->disk->major = 240;
    r->disk->first_minor = 0;
    r->disk->minors = 1;
    r->disk->fops = &ramdisk_fops;
    set_capacity(r->disk, r->capacity_sectors);

    add_disk(r->disk);
    return 0;
}
```

After `insmod`:

```
[root@pa-mini:~]# ls /dev/myram0
[root@pa-mini:~]# mkfs.ext4 /dev/myram0
[root@pa-mini:~]# mount /dev/myram0 /mnt
```

55D.3 Real block drivers, what’s added

The ramdisk above is minimal. Production block drivers add:

- **DISCARD / TRIM** (`BLK_STS_OK` on `REQ_OP_DISCARD`), letting the device free unused sectors. EMMCs and SSDs care.
- **FLUSH** (`REQ_OP_FLUSH`), ensure writes are durable. Critical for ACID-style apps.
- **Zoned device support** (`REQ_OP_ZONE_*`), SMR drives, ZNS NVMe.
- **Read-only handling**: block O_WRONLY when device is RO.
- **Partition table parsing**: automatic GPT/MBR scan, creating partitions as separate minors.

The kernel’s MMC subsystem is the canonical example for SD/eMMC. Read `drivers/mmc/core/`.

55D.4 Performance metrics

```
[root@pa-mini:~]# dd if=/dev/myram0 of=/dev/null bs=1M count=64
67108864 bytes (64 MB, 64 MiB) copied, 0.04 s, 1.7 GB/s

[root@pa-mini:~]# fio --name=randread --filename=/dev/myram0 \
  --rw=randread --bs=4k --runtime=10 --time_based --ioengine=psync
```

```
  ...
  read: IOPS=350k, BW=1366MiB/s ...
```

For comparison, an eMMC HS200 hits ~120 MB/s sequential, ~10k IOPS random. NAND flash via UBI hits ~10 MB/s.

55D.5 Key Notes

- The block layer describes sector-oriented I/O through bios, requests and queues. A block driver must honor offsets, lengths and completion semantics rather than act like an unstructured byte stream.
- Queue ownership ends only when the request is completed or failed through the expected interface. Incorrect completion accounting can affect both data integrity and scheduling.
- A RAM disk explains the block contract without implementing storage durability. Its contents depend on the backing allocation’s lifetime.
- Filesystem behavior depends on error and flush semantics beneath it. A successful write call is not automatically evidence of persistence across power loss.
- Removal and resource cleanup must account for open or mounted users. Device lifetime is part of the interface, not just module initialization.

55D.6 Pitfalls

> **Storage safety:** Before any command that names /dev/sdX, run lsblk -o NAME,SIZE,MODEL,TRAN,TYPE,MOUNTPOINTS. Verify the removable card by size and model, unmount its partitions, and stop if the path is not the target card. Writing the wrong /dev node can destroy the host disk.

- **Forgetting to `blk_mq_start_request`.** Driver does work, calls `end_request`, but the request was never marked started, corrupted statistics, possible deadlock.
- **Block-aligned constraint violations.** Modern kernels require sector-aligned bios. Requests smaller than a sector get split. Honor `bvec` offsets.
- **Holding spinlocks across long memcpy.** `queue_rq` is expected to return quickly. Defer long work to a workqueue.
- **`mkfs.ext4` complaints about a tiny disk.** Some FS minimums apply (~8 MB for ext4 with default settings).
- **Unloading while disk is mounted.** Kernel panics. Always `umount` before `rmmod`.

55D.7 Going deeper

- **`Documentation/block/`**: block layer documentation.
- **`drivers/block/`**: many real block drivers, mostly readable.
- **`drivers/block/loop.c`**: the loop driver, a good “real but simple” example.
- **`drivers/mmc/core/block.c`**: MMC block device.
- **LDD3 Chapter 16**: block drivers (some details outdated, but the model is the same).

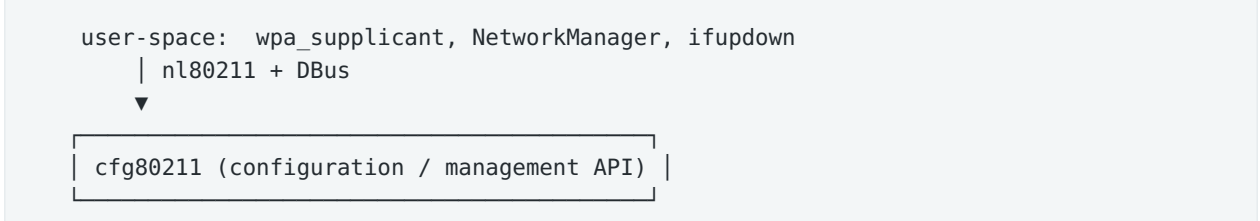
> Next chapter: **Chapter 55E: WiFi + wpa_supplicant.** Bringing up an SDIO or USB WiFi module on Linux, end-to-end.

Chapter 55E: WiFi + wpa_supplicant

> **DMA:** Direct Memory Access. Hardware moves data to or from memory without the CPU copying each byte. **MCU bridge:** Think of DMA like the MCU DMA controller you used for UART or SPI, but with cache coherency, scatter-gather descriptors, and kernel ownership rules added.

> **What:** Linux's **WiFi stack**, `mac80211` (the kernel's 802.11 implementation), `cfg80211` (the configuration API), per-chip drivers (`brcmfmac`, `rtl8xxxu`, etc.), and the user-space **wpa_supplicant** that handles WPA2/WPA3 authentication. We focus on SDIO WiFi (AP6212 / RTL8189 on Point Atom-class boards) as the most common embedded case.
>
> **Why:** WiFi is a vertical stack, driver, kernel WiFi core, supplicant, network manager. If any layer is wrong, nothing works. Knowing the layers and how to debug each one turns "WiFi doesn't work" from a multi-day debug into a methodical bring-up.
>
> **Focus: firmware + nvram + supplicant**. The driver loads vendor firmware from `/lib/firmware/`, plus a per-board nvram (channel list, antennas, regulatory). Wpa_supplicant handles the 4-way handshake. All three must be right.
>
> **Tooling.** This chapter uses `wpa_supplicant`, `iw`, optional `hostapd`. Chip firmware blob.
>
> - **Ubuntu-base (target):** `apt install wpasupplicant iw hostapd firmware-realtek firmware-brcm80211`
> - **Buildroot:** `BR2_PACKAGE_WPA_SUPPLICANT=y BR2_PACKAGE_IW=y BR2_PACKAGE_HOSTAPD=y` **Buildroot:** a configuration-driven build system that produces a complete root filesystem and related images.
> - Full per-tool reference: Userspace tooling appendix. **MCU bridge:** Think of the rootfs as the firmware image's file-backed runtime environment. On an MCU you link everything into flash. On Linux, programs and config live in this mounted tree. **rootfs:** root filesystem, the directory tree mounted at / that contains /bin, /etc, /dev, and libraries.

55E.1 The stack



```
        |
        ▼
   ┌──────────────────────────────────────────────┐
   | mac80211 (kernel 802.11 implementation)   |
   |   - filters, queues, rate control         |
   |   - regulatory domain enforcement         |
   └──────────────────────────────────────────────┘
        |
        ▼
   ┌──────────────────────────────────────────────┐
   | chip driver: brcmfmac, rtl8xxxu, ...       |
   |   - SDIO / USB / SPI transport             |
   |   - firmware loading                        |
   └──────────────────────────────────────────────┘
        |
        ▼
   wifi hardware (AP6212 / RTL8189 / etc.)
```

For “full-MAC” chips (most embedded modules), much of the 802.11 logic is in chip firmware. The driver is thin. For “soft-MAC” chips, mac80211 does more.

> **MAC:** Media Access Control in networking and radio chapters. It is the layer that owns framing and medium access.

55E.2 AP6212 on SDIO (representative case)

DT:

```
&usdhc2 {
    pinctrl-names = "default";
    pinctrl-0 = <&pinctrl_usdhc2>;
    bus-width = <4>;
    non-removable;
    keep-power-in-suspend;
    wakeup-source;
    vmmc-supply = <&reg_wlan>;
    mmc-pwrseq = <&wifi_pwrseq>;
    status = "okay";

    wifi@1 {
        compatible = "brcm,bcm4329-fmac";
        reg = <1>;
        interrupt-parent = <&gpio5>;
        interrupts = <8 IRQ_TYPE_LEVEL_HIGH>;
    };
};

wifi_pwrseq: wifi_pwrseq {
    compatible = "mmc-pwrseq-simple";
    reset-gpios = <&gpio5 9 GPIO_ACTIVE_LOW>;
    clocks = <&clks IMX6UL_CLK_OSC>;
```

```
    clock-names = "ext_clock";
};
```

Three things to notice:

- **SDIO bus configured** with `bus-width = 4`, no card-detect, `non-removable`.
- **Power sequence** drives the WL_REG_ON reset GPIO high to enable the chip.

> **MCU bridge:** Think of Linux GPIO like the same pin set/reset block you used on STM32, but accessed through a kernel subsystem that owns numbering, direction, interrupts, and user-space exposure. **GPIO:** General-Purpose Input/Output, a pin controlled as a digital input, output, or interrupt source.

- **`brcm,bcm4329-fmac`** matches the brcmfmac driver.

55E.3 Firmware and nvram

brcmfmac loads two files from `/lib/firmware/brcm/`:

```
brcmfmac43430-sdio.bin          ← firmware blob
brcmfmac43430-sdio.txt          ← nvram (per-board calibration & config)
```

The exact filename depends on chip ID. Find current names at https://git.kernel.org/pub/scm/linux/kernel/git/firmware/linux-firmware.git/tree/brcm.

Without the nvram, the driver probes but channels do not enumerate correctly (or at all). The nvram is per-board. Copying from another board gives the wrong antenna config, the wrong regulatory domain, and broken behavior. Always get the matching nvram from your board vendor.

55E.4 Bring-up trace

```
[root@pa-mini:~]# dmesg | grep -iE 'brcm|wifi|mmc'
mmc1: SDHCI controller on 2194000.usdhc [2194000.usdhc] using DMA
mmc1: new high speed SDIO card at address 0001
brcmfmac: brcmf_fw_alloc_request: using brcm/brcmfmac43430-sdio
brcmfmac: F1 signature read @0x18000000=0x4040a9a6
brcmfmac: brcmf_fw_complete_request: Found firmware (...)
brcmfmac: bcm4330-fmac initialized
[root@pa-mini:~]# ip link
3: wlan0: <BROADCAST,MULTICAST> mtu 1500 ...
```

`wlan0` appears. Now scan and connect:

```
[root@pa-mini:~]# ip link set wlan0 up
[root@pa-mini:~]# iw wlan0 scan | grep -E 'SSID|signal'
```

```
        SSID: MyHomeWiFi
        signal: -45.00 dBm
        SSID: GuestNet
        signal: -67.00 dBm
```

55E.5 wpa_supplicant

`/etc/wpa_supplicant.conf`:

```
ctrl_interface=/var/run/wpa_supplicant
update_config=1

network={
    ssid="MyHomeWiFi"
    psk="my-password"
    key_mgmt=WPA-PSK
}
```

Start:

```
# Manual:
wpa_supplicant -B -i wlan0 -c /etc/wpa_supplicant.conf -D nl80211

# Then get an IP:
udhcpc -i wlan0
# or:
dhclient wlan0
```

WPA3 (SAE, Simultaneous Authentication of Equals):

```
network={
    ssid="MyWPA3Net"
    sae_password="my-password"
    key_mgmt=SAE
    ieee80211w=2
}
```

55E.6 Useful tools

```
[root@pa-mini:~]# iw dev                  # show interfaces and modes
[root@pa-mini:~]# iw wlan0 link            # current connection
[root@pa-mini:~]# iw wlan0 station dump    # signal, throughput, retries
[root@pa-mini:~]# wpa_cli                  # interactive supplicant control
[root@pa-mini:~]# iwconfig wlan0           # legacy tool; still useful
```

55E.7 Key Notes

- Wi-Fi spans bus discovery, chip firmware, kernel wireless interfaces, authentication and IP networking. Failure at one layer should not be diagnosed as a failure of the whole stack.
- Board-specific calibration and firmware are part of the hardware support contract. A driver name alone does not establish that a module is correctly supported.
- `wpa_supplicant` owns authentication policy, while the driver and firmware own radio operations. DHCP and routing begin after that connection boundary.
- Regulatory configuration and supported authentication modes constrain deployment. Connectivity in one environment is not universal regional or security compatibility.
- Roaming, power saving and reconnect behavior need application-level availability assumptions beyond a single successful association.

55E.8 Pitfalls

- **Missing firmware.** Symptom: “Direct firmware load failed -2.” Copy the right blob to `/lib/firmware/brcm/`.
- **Wrong nvram.** WiFi enumerates but signal is terrible or country code wrong. Get vendor’s nvram.
- **MMC pwrseq not triggering.** Reset GPIO not toggled before SDIO init. Symptom: chip doesn’t respond at all. Check the `mmc-pwrseq-simple` node.
- **32 KHz clock missing.** Most BCM and Realtek modules need a 32 kHz LPO clock. Without it, sleep modes fail and the chip becomes unreliable.
- **regdb missing.** Kernel won’t tune above channel 11 without a regulatory database loaded. Install `wireless-regdb`.
- **WPA3 / SAE not supported by chip firmware.** Older AP6212 firmware lacks SAE. Upgrade firmware or fall back to WPA2.

55E.9 Going deeper

- **`Documentation/networking/cfg80211.rst`**: cfg80211 API.
- **`Documentation/networking/mac80211_hwsim.rst`**: for testing without hardware.
- **`drivers/net/wireless/broadcom/brcm80211/brcmfmac/`**: brcmfmac driver source.
- **`hostap.git/wpa_supplicant/`**: wpa_supplicant source and configuration examples.
- **https://wireless.wiki.kernel.org/**: the canonical kernel wireless wiki.

> Next chapter: **Chapter 55F: Cellular modems.** USB 4G modems (EC25) and the QMI/MBIM modes. UART AT-command modems.

Chapter 55F: Cellular modems

> **What:** integrating a 4G/LTE modem with embedded Linux. The two big paths: **USB modems** with QMI/MBIM/RNDIS data interfaces (Quectel EC20/EC25, SimCom SIM7600), and **UART AT-command modems** with PPP (older parts and NB-IoT chips). The user-space orchestrator is **ModemManager**, supplemented by `qmicli` / `mbimcli` or, for AT modems, raw chat scripts.
>
> **Why:** every IoT device that ships without WiFi+wired backhaul has a cellular modem. Getting the bring-up right early avoids weeks of customer-side debugging.
>
> **Focus: picking the data mode**. The EC25 alone exposes four USB modes: RNDIS, ECM, QMI, MBIM. The wrong choice produces no working interface. QMI is the standard mode for Linux today. Use it unless you have a specific reason not to.
>
> **Tooling.** This chapter uses `ppp` (`pppd`, `chat`), `ModemManager` (`mmcli`), `libqmi-utils` (`qmicli`).
>
> - **Ubuntu-base (target):** `apt install ppp modemmanager libqmi-utils`
> - **Buildroot:** `BR2_PACKAGE_PPP=y BR2_PACKAGE_MODEM_MANAGER=y BR2_PACKAGE_LIBQMI=y` **Buildroot:** a configuration-driven build system that produces a complete root filesystem and related images.
> - Full per-tool reference: Userspace tooling appendix. **MCU bridge:** Think of the rootfs as the firmware image's file-backed runtime environment. On an MCU you link everything into flash. On Linux, programs and config live in this mounted tree. **rootfs:** root filesystem, the directory tree mounted at / that contains /bin, /etc, /dev, and libraries.

55F.1 Hardware connection

USB modems plug into a USB host port (i.MX6ULL USB OTG configured as host). They draw 1–2 A during TX bursts, your power supply must handle it. Most bench-test failures are power-related.

UART modems wire to a UART (typically 115200 baud + flow control). Plus an "enable" GPIO and a "status" GPIO.

> **MCU bridge:** Think of Linux GPIO like the same pin set/reset block you used on STM32, but accessed through a kernel subsystem that owns numbering, direction, interrupts, and user-space exposure. **GPIO:** General-Purpose Input/Output, a pin controlled as a digital input, output, or interrupt source.

55F.2 USB EC25 (QMI mode)

After plugging in, `lsusb` shows:

```
[root@pa-mini:~]# lsusb
Bus 001 Device 002: ID 2c7c:0125 Quectel Wireless Solutions Co., Ltd. EC25 LTE modem
```

The driver `qmi_wwan` handles QMI. `option` provides AT-command serial ports.

```
[root@pa-mini:~]# dmesg | tail -20
usb 1-1: New USB device found, idVendor=2c7c, idProduct=0125
qmi_wwan 1-1:1.4 wwan0: register 'qmi_wwan' at usb-...
option 1-1:1.0: GSM modem (1-port) converter detected
usb 1-1: GSM modem (1-port) converter now attached to ttyUSB0
usb 1-1: GSM modem (1-port) converter now attached to ttyUSB1
...
```

`/dev/ttyUSB0–/dev/ttyUSB3` are AT-command channels. `wwan0` is the data interface. `/dev/cdc-wdm0` is the QMI control channel.

55F.3 ModemManager, the easy path

```
[root@pa-mini:~]# systemctl start ModemManager
[root@pa-mini:~]# mmcli -L
    /org/freedesktop/ModemManager1/Modem/0 [Quectel] EC25

[root@pa-mini:~]# mmcli -m 0
  ------------------------
  Hardware |   manufacturer: Quectel
           |          model: EC25
           |       revision: EC25EFAR05A04M4G
  System   |         device: /sys/devices/...
  ...
  Status   |  unlock retries: sim-pin (3), sim-puk (10)
           |          state: registered
           |    power state: on
           |        signal: 75 %
```

Set up a connection:

```
[root@pa-mini:~]# mmcli -m 0 --simple-connect="apn=internet"
successfully connected the modem
[root@pa-mini:~]# ip link show wwan0
wwan0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc fq_codel state UP qlen 1000
[root@pa-mini:~]# dhclient wwan0
[root@pa-mini:~]# ping 8.8.8.8
```

For NetworkManager / nmcli users:

```
nmcli connection add type gsm con-name lte apn internet
nmcli connection up lte
```

55F.4 Manual QMI

If you don’t want ModemManager:

```
# Set raw mode for the wwan0 interface
[root@pa-mini:~]# qmicli -d /dev/cdc-wdm0 --dms-set-operating-mode='online'
[root@pa-mini:~]# qmicli -d /dev/cdc-wdm0 --wda-set-data-format='raw-ip'
[root@pa-mini:~]# ip link set wwan0 down
[root@pa-mini:~]# echo Y > /sys/class/net/wwan0/qmi/raw_ip
[root@pa-mini:~]# ip link set wwan0 up

# Start data session
[root@pa-mini:~]# qmicli -d /dev/cdc-wdm0 --wds-start-network="apn='internet',ip-type=4"
--client-no-release-cid
        Packet data handle: '...'
        CID: '...'

[root@pa-mini:~]# udhcpc -i wwan0
```

Manual setup is fragile. Use ModemManager when you can.

55F.5 UART AT-command modem (e.g., A7670C)

Wire UART TX/RX to a UART on the i.MX6ULL. Power-on sequence: assert PWRKEY GPIO for ~500 ms.

```
[root@pa-mini:~]# echo 1 > /sys/class/gpio/gpioN/value
[root@pa-mini:~]# sleep 0.5
[root@pa-mini:~]# echo 0 > /sys/class/gpio/gpioN/value

[root@pa-mini:~]# minicom -D /dev/ttymxc1 -b 115200
AT
OK
AT+CSQ
+CSQ: 22,99
OK
AT+CGREG?
+CGREG: 0,1
OK
```

For data: PPP via `pppd` + chat script.

`/etc/ppp/peers/a7670`:

```
/dev/ttymxc1
115200
crtscts
connect '/usr/sbin/chat -v -f /etc/ppp/chat-a7670'
```

```
defaultroute
usepeerdns
noauth
```

`/etc/ppp/chat-a7670`:

```
ABORT BUSY
ABORT 'NO CARRIER'
'' AT
OK ATD*99#
CONNECT ''
```

Then `pppd call a7670`. A `ppp0` interface appears, with internet routed through the modem.

PPP is slow (~10 Mbit/s max) and adds latency. Use only for old modems without USB.

55F.6 Key Notes

- Cellular integration separates modem control, the data interface and host network configuration. QMI, MBIM, Ethernet-like modes and PPP imply different drivers and management paths.
- One manager should own modem state. Competing AT sessions or network managers can change a configuration another component expects to retain.
- Registration, a data session, an IP address and usable routing are separate states. An `OK` reply to an AT command proves none of the later states.
- Carrier provisioning, antenna design and supply transients are system dependencies. USB enumeration does not qualify the RF or power path.
- Failover needs a route and retry policy that considers cost, availability and session disruption, not just whether one ping succeeded.

55F.7 Pitfalls

- **USB power.** EC25 spikes to >1.5 A on TX bursts. 5V/2A is the minimum. Less = USB resets at the worst moments.
- **Wrong USB mode.** EC25 boots in RNDIS by default in some firmware. Switch to QMI: `AT+QCFG="usbnet",0`.
- **APN wrong / missing.** Carriers reject your registration silently. Cross-check with your SIM provider.
- **Regulatory.** Some bands are illegal in some countries. Use a regional-specific modem variant.

- **Antenna SWR.** Bad antenna match = poor signal. Cellular antennas matter as much as WiFi antennas.
- **`raw_ip` mode trap.** EC25 needs `raw_ip` mode. ModemManager sets it. Manual QMI must too.

55F.8 Going deeper

- **https://www.freedesktop.org/wiki/Software/ModemManager/**: ModemManager docs.
- **`libqmi` / `libmbim`**: the C libraries beneath qmicli/mbimcli.
- **`drivers/net/usb/qmi_wwan.c`**: kernel QMI driver.
- **`drivers/usb/serial/option.c`**: AT-command channel exposure.
- **3GPP TS 27.007**: AT command set standard.

> Next chapter: **Chapter 55G: Multi-touch GT911.** Touchscreens, MT-B slot protocol, calibration.

Chapter 55G: Multi-touch (GT911)

> **What:** the **MT-B** (slot-based multi-touch) protocol, the kernel's standard for reporting per-finger touch coordinates, and the **Goodix GT911**, the common I²C capacitive touch controller that ships with most off-the-shelf RGB-parallel LCDs (ATK4384, ATK7016, ATK10261).
>
> **Why:** for any product with a touch panel, this is the input. The mainline `goodix` driver covers GT911, GT9110, GT9271 and other variants. You usually just configure DT correctly. Calibration is rarely needed for capacitive (unlike resistive). The panel's coordinate frame is wired in DT.
>
> **Focus: MT-B is slot-based**. Each tracked finger gets a *slot*. The driver reports per-slot position. Older code uses MT-A. Current code uses MT-B.

55G.1 MT-B vs MT-A

MT-A (legacy): each touch group is bracketed by `MT_SYNC` events. The kernel tracks which contact is which by proximity in successive frames. Order-dependent, confusing.

MT-B (current): the driver assigns each contact a *slot* number. As long as the same finger is down, its slot doesn't change. Cleaner code, cleaner user-space.

A two-finger touch event in MT-B:

```
EV_ABS  ABS_MT_SLOT       0
EV_ABS  ABS_MT_TRACKING_ID  42
EV_ABS  ABS_MT_POSITION_X 100
EV_ABS  ABS_MT_POSITION_Y 200

EV_ABS  ABS_MT_SLOT       1
EV_ABS  ABS_MT_TRACKING_ID  43
EV_ABS  ABS_MT_POSITION_X 300
EV_ABS  ABS_MT_POSITION_Y 400

EV_SYN  SYN_REPORT
```

Finger lift: write `ABS_MT_TRACKING_ID = -1` to that slot.

55G.2 GT911 in DT

```
&i2c2 {
    gt911@5d {
        compatible = "goodix,gt911";
        reg = <0x5d>;
        interrupt-parent = <&gpio1>;
```

```
        interrupts = <9 IRQ_TYPE_EDGE_FALLING>;
        irq-gpios = <&gpio1 9 GPIO_ACTIVE_HIGH>;
        reset-gpios = <&gpio5 2 GPIO_ACTIVE_HIGH>;
        touchscreen-size-x = <1024>;
        touchscreen-size-y = <600>;
        touchscreen-inverted-x;
        touchscreen-inverted-y;
        touchscreen-swapped-x-y;
    };
};
```

Key properties:

- **`reg = <0x5d>` or `<0x14>`**: the GT911 has two I²C addresses. The IRQ pin level at reset selects which: 0x5d when IRQ is low, 0x14 when IRQ is high.
- **`reset-gpios`**: pulsed at probe.
- **`touchscreen-size-x/y`**: physical resolution to report. Usually matches the LCD.
- **`touchscreen-inverted-x/y`, `touchscreen-swapped-x-y`**: rotate/flip the touch coordinate frame to match the LCD orientation.

55G.3 Wiring

GT911 needs:

- VDD (3.3V).
- GND.
- SDA, SCL.
- INT (touch-event interrupt. Also doubles as I²C-address-select at reset).
- RST.

The driver bring-up sequence for RST and INT selects the I²C address. The driver handles this for you.

55G.4 Verify it works

```
[root@pa-mini:~]# dmesg | grep -i goodix
Goodix-TS 1-005d: ID 911, version: 1060
[root@pa-mini:~]# ls /dev/input/event*
/dev/input/event0  /dev/input/event1
[root@pa-mini:~]# evtest /dev/input/event1
Supported events:
  Event type 1 (EV_KEY)
    Event code 330 (BTN_TOUCH)
  Event type 3 (EV_ABS)
    Event code 47 (ABS_MT_SLOT)
```

```
        Value      0
        Min        0
        Max        4
    Event code 53 (ABS_MT_POSITION_X)
        Min        0
        Max     1024
    ...

[Press a finger]
Event: time 12345.67, type 3 (EV_ABS), code 57 (ABS_MT_TRACKING_ID), value 0
Event: time 12345.67, type 3 (EV_ABS), code 53 (ABS_MT_POSITION_X), value 312
Event: time 12345.67, type 3 (EV_ABS), code 54 (ABS_MT_POSITION_Y), value 450
Event: time 12345.67, type 1 (EV_KEY), code 330 (BTN_TOUCH), value 1
Event: time 12345.67, ------------ SYN_REPORT ---------
```

Touch is now reported through standard input events. User-space tools (X11, Wayland, Qt eglfs/linuxfb) consume them.

55G.5 Coordinate frame fixes

If touch is “mirrored” or “rotated” relative to the visible LCD:

- **Inverted X** (left/right swapped): `touchscreen-inverted-x;`
- **Inverted Y** (top/bottom swapped): `touchscreen-inverted-y;`
- **Swapped X-Y** (touched a horizontal line, axis is vertical): `touchscreen-swapped-x-y;`

Try combinations until the touch matches the cursor on screen. For resistive touchscreens (XPT2046 etc.), this isn’t enough, you need software calibration with `xinput_calibrator`, since the touch values are not linearly mapped to display pixels.

55G.6 Firmware

GT911 boots from internal ROM but accepts firmware updates via I²C. Some boards ship a firmware file. The driver auto-loads from `/lib/firmware/goodix_911.fw` if present. Without it, the chip uses its ROM firmware, usually adequate.

55G.7 Key Notes

- MT-B reports contacts through slots and tracking IDs. Finger lifetime is different from the number of coordinate samples received.
- Reset and interrupt wiring affect both startup and address selection on the GT911. Those details belong to the fitted panel assembly, not merely its advertised resolution.
- Touch coordinates and displayed pixels are separate coordinate systems. Rotation, inversion and axis swapping should follow a defined mapping.

- The kernel reports contacts, while user-space input policy interprets long presses and gestures. Keeping that boundary clear avoids embedding application behavior in a hardware driver.
- A working single contact does not establish correct tracking, release behavior or latency under several simultaneous contacts.

55G.8 Pitfalls

- **Wrong I²C address.** RST/INT timing at reset selects 0x5d vs 0x14. If the driver's expected address doesn't match, no probe.
- **Missing reset-gpios.** Chip never wakes up. I²C-detect fails.
- **Polling instead of IRQ.** Without `interrupts`, driver polls (slow, laggy). Always wire and declare the IRQ.
- **`touchscreen-size-x/y` wrong.** Touch coordinates reported in wrong range. User-space scales weirdly.
- **Touch latency from kernel.** Default `goodix` driver is good. If you have weird latency, check `evtest` timestamps against vsync.
- **Two touchscreens conflict.** XPT2046 + GT911 both on the same panel? Pick one in DT.

55G.9 Going deeper

- **`Documentation/input/multi-touch-protocol.rst`**: MT-B protocol reference.
- **`drivers/input/touchscreen/goodix.c`**: the GT911 driver.
- **`Documentation/devicetree/bindings/input/touchscreen/goodix.yaml`**: DT binding.
- **`tools/testing/selftests/input/`**: input test tooling.

> Next chapter: **Chapter 55H: RGB-to-HDMI bridge (sii902x).** Adding HDMI output to an i.MX6ULL by hanging an HDMI transmitter chip off the LCDIF parallel RGB.

Chapter 55H: RGB-to-HDMI bridge (sii902x)

> **What:** the **Silicon Image SiI902x** family of RGB-parallel-to-HDMI transmitter chips, and the kernel's **DRM bridge** subsystem. The i.MX6ULL has no native HDMI. An SiI9022/SiI9024 chip on the LCDIF parallel output gives you HDMI. The mainline `sii902x.c` DRM bridge driver handles config and EDID parsing.
>
> **Why:** any product that needs an external display (HMI, kiosk, signage) usually wants HDMI compatibility. SiI902x is a small, low-cost (~$2) chip that takes 24-bit RGB + HSYNC/VSYNC/PCLK and outputs HDMI 1.4 at up to 1080p60. It works on any i.MX6ULL board with LCDIF pinmux available.
>
> **Focus: the bridge concept**. A DRM "bridge" sits between a CRTC (LCDIF) and a connector (HDMI port). DRM chains bridges automatically. You describe the chain in DT and the driver does the rest.

55H.1 Hardware

SiI902x has:

- **Input**: 24 parallel RGB + HSYNC + VSYNC + PCLK + DE. From i.MX6ULL LCDIF.
- **Output**: 4 differential pairs + DDC (HPD + I²C). To HDMI connector.
- **I²C control**: 0x39 typically.
- **HPD (Hot Plug Detect)** interrupt: tells the bridge when an HDMI cable is attached.

Connect LCDIF parallel out → SiI902x in, HDMI cable → SiI902x out, I²C2 + HPD-IRQ → host. The bridge handles EDID negotiation with the sink (TV/monitor).

> **MCU bridge:** Think of an IRQ like an EXTI/NVIC interrupt path, except Linux splits the hard interrupt from deferred work and must share lines across drivers. **IRQ:** interrupt request, the signal path that tells the CPU or interrupt controller that hardware needs service.

55H.2 Device tree

```
&i2c2 {
    sii902x: hdmi@39 {
        compatible = "sil,sii9022";
        reg = <0x39>;
```

```
        reset-gpios = <&gpio2 5 GPIO_ACTIVE_LOW>;
        interrupt-parent = <&gpio2>;
        interrupts = <6 IRQ_TYPE_EDGE_FALLING>;
        #sound-dai-cells = <0>;

        ports {
            #address-cells = <1>;
            #size-cells = <0>;

            port@0 {
                reg = <0>;
                sii902x_in: endpoint {
                    remote-endpoint = <&lcdif_out>;
                };
            };

            port@1 {
                reg = <1>;
                sii902x_out: endpoint {
                    remote-endpoint = <&hdmi_connector_in>;
                };
            };
        };
    };
};

&lcdif {
    status = "okay";
    port {
        lcdif_out: endpoint {
            remote-endpoint = <&sii902x_in>;
        };
    };
};

hdmi_connector: hdmi-connector {
    compatible = "hdmi-connector";
    type = "a";
    port {
        hdmi_connector_in: endpoint {
            remote-endpoint = <&sii902x_out>;
        };
    };
};
```

Three DT nodes form the graph: LCDIF, SiI902x, and the HDMI connector. The DRM bridge chain wires them together.

55H.3 How it works

At boot:

1. SiI902x's `sii902x_probe` registers a DRM bridge.
2. LCDIF's DRM driver sees a downstream bridge instead of a panel.

3. SiI902x's `attach` callback runs. It queries the connector for HPD state.
4. If HPD is asserted (cable plugged), it reads EDID from the sink (via DDC I²C on the HDMI connector).
5. Available modes derived from EDID become valid DRM modes.

`modetest` shows the modes:

```
[root@pa-mini:~]# modetest -M mxsfb
Connectors:
id      encoder status          name            size (mm)       modes   encoders
42      40      connected       HDMI-A-1        598x336         18      40
        modes:
            #0 1920x1080 60.00 1920 2008 2052 2200 1080 1084 1089 1125 148500 flags:
phsync, pvsync; type: preferred
            #1 1280x720 60.00 1280 1390 1430 1650 720 725 730 750 74250 flags: phsync,
pvsync;
            ...
```

Set a mode:

```
[root@pa-mini:~]# modetest -M mxsfb -s 42:1920x1080
```

However, the i.MX6ULL LCDIF tops out at about 80 MHz pixel clock. 1080p60 is 148.5 MHz. Doesn't work. Practical max on i.MX6ULL via SiI902x is 720p (74.25 MHz). 480p, 576p, 720p all work.

55H.4 Audio over HDMI

SiI902x also carries audio. With `#sound-dai-cells = <0>;`, the chip exposes an audio DAI. Build an ASoC machine driver (Ch 53) that connects SAI → SiI902x → HDMI. Most products skip this and use HDMI for video only.

> **ASoC:** ALSA System-on-Chip, the embedded audio layer that connects CPU audio ports, codecs, and board wiring.

55H.5 Key Notes

- A DRM bridge transforms one display interface into another between the controller and connector. The i.MX6ULL has no native HDMI, so the external bridge and its wiring are part of the design.
- I2C control, parallel pixel input, DDC and hot-plug detection serve different roles. A successful bridge probe does not prove that a sink can receive pixels.

- EDID describes sink modes, but the controller, bridge and board must all support the selected timing. A monitor’s mode list is not permission to exceed source limits.
- HDMI audio is an additional path with its own clocks and routing. Video support alone does not establish audio support.
- Disconnects and invalid modes are normal states that the display pipeline should represent explicitly.

55H.6 Pitfalls

- **No HPD wiring.** Bridge thinks nothing connected. Mode list is empty.
- **Reset-gpios polarity wrong.** Chip held in reset forever.
- **DDC pull-ups missing.** EDID read fails. Schematic check.
- **Pixel clock > 80 MHz.** LCDIF chokes. Stay ≤ 720p on i.MX6ULL.
- **No frame-rate match.** Some monitors require specific timings. The EDID list filters out invalid ones but check the kernel debug output.
- **Modeset reports OK but the screen stays blank.** Often the HDMI sink is expecting a different pixel format than the bridge sends. SiI902x defaults to RGB. Some old TVs want YCbCr. Force RGB via property if needed.

55H.7 Going deeper

- **`drivers/gpu/drm/bridge/sii902x.c`**: the SiI902x DRM bridge driver.
- **`Documentation/gpu/drm-kms.rst`**: KMS bridge chain.
- **SiI9022 datasheet**: register map, EDID handling, audio config.

> Next chapter: **Chapter 55I: Rust-for-Linux.** A small kernel-module written in Rust, demonstrating the upcoming second kernel language.

Chapter 55I: Rust for Linux

> **What:** the **Rust-for-Linux** project, Rust as a second supported language inside the kernel since Linux 6.1. We cover the toolchain setup (rustc + bindgen + a specific Rust edition), what kernel APIs are *exposed* to Rust today, what kind of drivers can already be written, and how to compile a “hello world” kernel module in Rust.
>
> **Why:** Rust’s memory-safety guarantees apply at compile time. A whole class of C kernel bugs becomes impossible to write in safe Rust: use-after-free, double-free, data races on shared memory, integer overflows. The kernel community has accepted this as worth integrating because these bug classes account for a large share of kernel CVEs. As of 2026, Rust is still small in the kernel (drivers, no core subsystems) but growing.
>
> **Focus: the borrow checker, applied to kernel code**. The trade-off: more compile errors, fewer runtime errors. For a chapter on i.MX6ULL device drivers, the value is “you can do it for new drivers if you want, with caveats.”

55I.1 Status as of late 2025 / early 2026

- **In mainline**: since 6.1 (October 2022).
- **Supported architectures**: x86_64, arm64, RISC-V, LoongArch64. **ARM32 (i.MX6ULL’s architecture) is NOT yet supported.**
- **What’s written in Rust**: a small handful of drivers (NVMe, GPU, etc.). Adoption is slow.
- **What it takes**: nightly-ish Rust toolchain, bindgen, `CONFIG_RUST=y`.

On the i.MX6ULL specifically, Rust is not usable yet, ARM32 is not supported. This chapter is about *the model*. You’d apply it on a different SoC (i.MX8M, Raspberry Pi 4, etc.) where Rust-for-Linux works today.

55I.2 Toolchain

```
# A specific Rust version that matches the kernel's expectation
$ rustup toolchain install 1.74.0
$ rustup default 1.74.0
$ rustup component add rust-src

# bindgen for generating Rust bindings from C headers
$ cargo install --locked --version 0.65.1 bindgen-cli
```

The kernel's `Documentation/rust/quick-start.rst` lists the exact versions required for a given kernel. Mismatch = build fails.

Check the kernel supports it:

```
$ cd linux/
$ make ARCH=arm64 CROSS_COMPILE=aarch64-linux-gnu- LLVM=1 rustavailable
Rust is available!
```

If not, the message tells you what's missing.

Enable in `.config`:

```
General setup --->
    [*] Rust support
```

55I.3 Hello world in Rust

`rust_hello.rs`:

```
// SPDX-License-Identifier: GPL-2.0

//! A trivial Rust kernel module.

use kernel::prelude::*;

module! {
    type: RustHello,
    name: "rust_hello",
    author: "LinuxLearn",
    description: "Hello world from Rust",
    license: "GPL",
}

struct RustHello;

impl kernel::Module for RustHello {
    fn init(_module: &'static ThisModule) -> Result<Self> {
        pr_info!("Hello from Rust!\n");
        Ok(RustHello)
    }
}

impl Drop for RustHello {
    fn drop(&mut self) {
        pr_info!("Goodbye from Rust!\n");
    }
}
```

`Makefile`:

```
obj-m += rust_hello.o

KDIR ?= /lib/modules/$(shell uname -r)/build

all:
    $(MAKE) -C $(KDIR) M=$(PWD) modules

clean:
    $(MAKE) -C $(KDIR) M=$(PWD) clean
```

Build, load:

```
$ make
  CC [M] rust_hello.o
  ...

# insmod rust_hello.ko
# dmesg | tail -1
[ ... ] Hello from Rust!

# rmmod rust_hello
# dmesg | tail -1
[ ... ] Goodbye from Rust!
```

That's it. The `module!` macro expands to all the C boilerplate (module_init, MODULE_LICENSE, etc.) under the hood.

55I.4 What Rust gets you

Memory safety at compile time:

- Use-after-free: caught by the borrow checker.
- Double-free: caught by ownership rules.
- Data races on shared mutable state: caught by the borrow checker + `Send`/`Sync` traits.

Type-state programming: force certain sequences of operations via the type system. Example, a "driver" that *must* be initialised before any I/O:

```
struct Uninit;
struct Init;

struct MyDevice<S> { _state: PhantomData<S>, ... }

impl MyDevice<Uninit> {
    fn new() -> Self { ... }
    fn init(self) -> MyDevice<Init> { ... }   /* consumes; returns initialised */
}

impl MyDevice<Init> {
```

```
    fn read(&self) -> u32 { ... }   /* only callable on initialised */
}
```

The compiler refuses to compile a call to `.read()` on an uninitialised device. The “did you forget to init?” bug class is no longer expressible.

Error handling via `Result`:

```
fn read_register(reg: u32) -> Result<u32> {
    if reg > MAX_REG {
        return Err(EINVAL);
    }
    Ok(unsafe { read_volatile(... as *const u32) })
}

let val = read_register(0x40)?;     // `?` propagates errors automatically
```

You cannot silently ignore an error, the compiler will not let you.

55I.5 What’s *exposed* today (the limits)

Kernel APIs are exposed to Rust as Rust modules under `kernel::`. Currently available:

- `kernel::pr_info!`, `pr_err!`, etc. (printk).
- `kernel::sync::{Mutex, SpinLock, ...}`: locks.
- `kernel::workqueue::Work`: workqueues.
- `kernel::time::Ktime`: time.
- `kernel::error::{Result, Error}`: error handling.
- `kernel::miscdev::Registration`: misc device chardev.
- `kernel::file`: file_operations bindings.
- `kernel::pci::Driver`, `kernel::platform::Driver`, bus drivers.

Not yet exposed (as of early 2026):

- Most subsystem-specific APIs (input, sound, drm, networking).
- Lots of platform-specific helpers.

The list grows monthly. Check `rust/kernel/` in mainline.

55I.6 Trade-offs

Pros:

- Memory safety classes of bug eliminated.
- Type-driven API design.
- Better build-time errors.

Cons:

- Smaller community inside the kernel.
- Tooling churn, Rust version requirements change with each kernel.
- Toolchain disk cost (rustc + bindgen ~1.5 GB).
- C interop requires `unsafe { }` blocks where you cross the boundary. Within those blocks, all the same C bugs are possible, Rust safety depends on the `unsafe` blocks being correct. Bugs inside `unsafe` are no different from C bugs.
- More upfront design for the type system.

For an embedded project: if you're writing a new driver for a chip that doesn't exist in mainline, and you control the build environment, Rust is worth considering on Rust-supported architectures. For maintaining 4.1 BSP code or anything on ARM32, not yet.

> **BSP:** Board Support Package: vendor patches, configs, bootloader files, and scripts needed to boot one board.

55I.7 Key Notes

- Rust changes how memory ownership and some concurrency rules are expressed, but a driver still needs a correct hardware and kernel-interface design.
- Safe abstractions depend on justified unsafe boundaries. Wrapping a C API does not remove the need to establish lifetime, context and locking rules.
- Kernel Rust does not use the ordinary user-space standard library. The selected kernel's exposed abstractions and supported target determine what can actually be implemented.
- Toolchain compatibility belongs to the chosen kernel source version. A generic Rust installation is not evidence that its kernel integration is available.
- Language-level safety does not qualify register values, DMA behavior, timing or recovery policy. Those remain architectural and hardware responsibilities.

55I.8 Pitfalls

- **Toolchain version mismatch.** Each kernel pins specific rustc/bindgen versions. Use exactly what `Documentation/rust/quick-start.rst` says.
- **`unsafe` overuse.** Wrapping every C call in `unsafe { }` defeats the safety goal. Write thin safe wrappers around C APIs.
- **Trying to use std**: Kernel Rust is `#![no_std]`. Heap allocations go through specific kernel allocators.
- **Long compile times.** First build of a Rust module is slow (~30 s). Incremental is fast.
- **Linking issues with mixed C/Rust modules.** Currently uncommon. If you need it, follow the kernel maintainers' examples.

55I.9 Going deeper

> **Driver choice:** Use the in-tree, maintained driver first. Use out-of-tree, spidev, or custom-driver paths only after you accept the kernel-version maintenance cost and document who owns updates.

- **`Documentation/rust/`**: the Rust-for-Linux documentation.
- **`rust/kernel/`**: the in-kernel Rust support crate.
- **https://github.com/Rust-for-Linux**: historical out-of-tree work.
- **"Rust for Linux" talks from Linux Plumbers, Open Source Summit**: videos online.
- **Asahi Linux's Apple-GPU driver**: large real-world Rust kernel code.

> **End of Part VI, Driver Development.** From the smallest kernel module (Ch 36) to Rust-for-Linux (Ch 55I), you now have the full vocabulary: load/unload, chardev, hot-plug, platform binding, locking, blocking I/O, interrupts, GPIO, input, I²C, SPI, PWM/RTC, IIO, regmap, DMA, watchdog, PM, network, RT, audio, LCD, MTD/UBI, V4L2, USB, timers, async, CAN, block, WiFi, cellular, multi-touch, HDMI, Rust. **MCU bridge:** Think of DMA like the MCU DMA controller you used for UART or SPI, but with cache coherency, scatter-gather descriptors, and kernel ownership rules added. **MCU bridge:** Think of Linux PWM like an MCU timer output channel, except the driver exposes period, duty cycle, polarity, and enable state through a subsystem. **MCU bridge:** Think of Linux GPIO like the same pin set/reset block you used on STM32, but accessed through a kernel subsystem that owns numbering, direction,

> interrupts, and user-space exposure. **MCU bridge:** Think of regmap like a typed wrapper around your read_reg() and write_reg() helpers, with caching, locking, and bus differences handled centrally. **DMA:** Direct Memory Access. Hardware moves data to or from memory without the CPU copying each byte. **PWM:** Pulse-Width Modulation, a timer output whose duty cycle controls average power or encodes timing. **IIO:** Industrial I/O, Linux's subsystem for sensors, ADCs, DACs, and buffered sampled data. **UBI:** Unsorted Block Images, a flash-management layer over raw NAND that handles wear leveling and bad blocks. **MTD:** Memory Technology Device, Linux's raw flash subsystem for eraseblock-based storage. **GPIO:** General-Purpose Input/Output, a pin controlled as a digital input, output, or interrupt source. **regmap:** a kernel helper that wraps register reads and writes over I2C, SPI, or MMIO.
>
> The next Part is the **Device Cookbook (Part VII)**, Ch 64 onwards. It applies these subsystem interfaces to representative chips, comparing wiring, Device Tree, driver responsibilities and failure modes. Its Key Notes focus on the decisions behind each integration.

> Next chapter: **Chapter 64: QSPI NOR flash** (Winbond W25Q128 / Macronix MX25L256 / Micron MT25Q).

Chapter 64: QSPI NOR flash

> **UBI:** Unsorted Block Images, a flash-management layer over raw NAND that handles wear leveling and bad blocks.

> **Naming convention used across Part VII.** Shell prompts shown as `[root@pa-mini:~]#` are illustrative target prompts using the hostname `pa-mini`. They are not captured MINI test results. A hostname does not establish that a peripheral is fitted, correctly wired or supported on a particular board revision.

> **What:** how a QSPI NOR flash chip actually works on the wire, how the mainline `spi-nor` driver implements it, and how to write your own minimal driver from scratch for one specific chip. Three chips compared, **Winbond W25Q128** (16 MB), **Macronix MX25L25645G** (32 MB), **Micron MT25QL256ABA** (32 MB), but the from-scratch driver targets the W25Q128 to keep the example concrete.
>
> **Why:** the philosophy of this book is “raw, build it yourself, understand it forever.” For QSPI flash that means: command bytes on the wire, status-register polling, page-program timing, JEDEC ID parsing. After this chapter you can read the mainline `spi-nor` source and know exactly what each function is hiding. If you encounter a chip that is not in the database, you can add an entry, or replace the framework with about 200 lines of your own.
>
> **Focus: a NOR flash is a state machine driven by single-byte commands**. `0x9F` = read JEDEC ID. `0x06` = write-enable. `0x20` = sector erase. `0x02` = page program. `0x03` = read. Send the right bytes in the right order and you can read, erase, program, and identify any standard NOR flash with about 100 lines of code. The mainline driver wraps this in abstractions and a parameter database, but the wire protocol itself is small.

64.1 Why QSPI NOR vs eMMC vs SD vs raw NAND

	QSPI NOR	eMMC	SD card	Raw NAND
Typical size	8–32 MB	4–64 GB	4 GB–1 TB	256 MB–4 GB
Read speed	50 MB/s	250 MB/s	90 MB/s	30 MB/s

	QSPI NOR	eMMC	SD card	Raw NAND
Write speed	0.5 MB/s	100 MB/s	60 MB/s	5 MB/s
Erase block	4–64 KB	invisible	invisible	128 KB
Erase cycles	100,000	1k–10k	varies	10,000
XIP-capable	✔	✗	✗	✗
Wear-leveling in HW	none, driver problem	yes	yes	none, UBI
Cost (volume)	$1–5	$5–20	$3–10	$2–10
Best for	Small soldered boot device	Main consumer storage	Removable / dev	Mid-size industrial

QSPI NOR fits when your storage need is under 32 MB, when you want a fast, deterministic boot, when you want a soldered device that resists theft, and when there is little user data to store. Many industrial i.MX6ULL designs boot from QSPI NOR.

64.2 Chip comparison

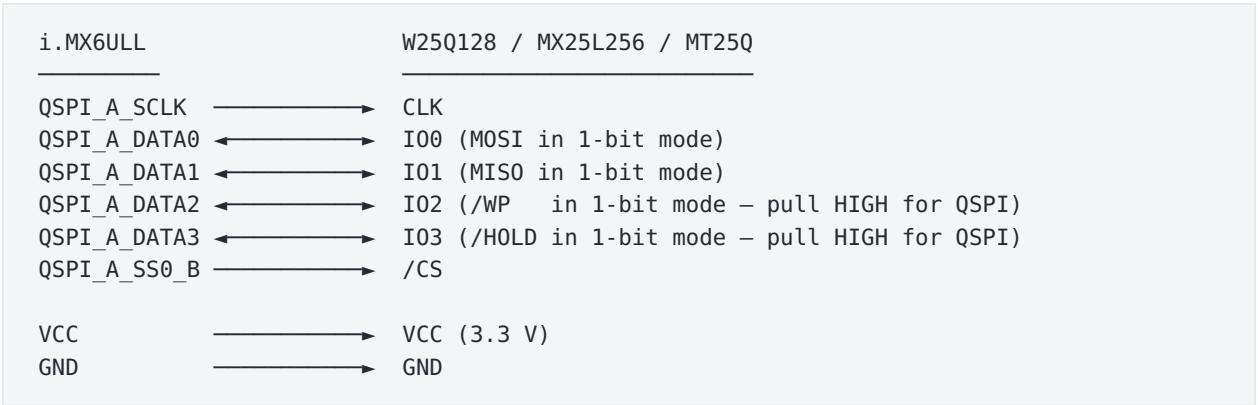
	W25Q128 (Winbond)	MX25L25645G (Macronix)	MT25QL256ABA (Micron)
Capacity	16 MB	32 MB	32 MB
Max SPI clock	104 MHz	133 MHz	166 MHz
Max QSPI clock	80 MHz	104 MHz	90 MHz
Page program size	256 B	256 B	256 B

	W25Q128 (Winbond)	MX25L25645G (Macronix)	MT25QL256ABA (Micron)
Sector erase	4 KB	4 KB	4 KB
Address bytes	3 (auto 4-byte cmds also avail.)	3 + 4-byte mode	3 + 4-byte mode
Erase cycles	100k	100k	100k (rated)
Temp grade	-40 to +85 °C	-40 to +85 °C	-40 to +105 °C AEC-Q100
Volume price	$1.20–1.80	$2.50–4.00	$5–7
JEDEC ID	`ef 40 18`	`c2 20 19`	`20 ba 19`

All three speak the same core command set. The chips differ mainly in capacity, top speed, and the 4-byte-address handshake. The from-scratch driver below works on all three (with the address-byte size adjusted).

64.3 Schematic

The minimum is six wires:



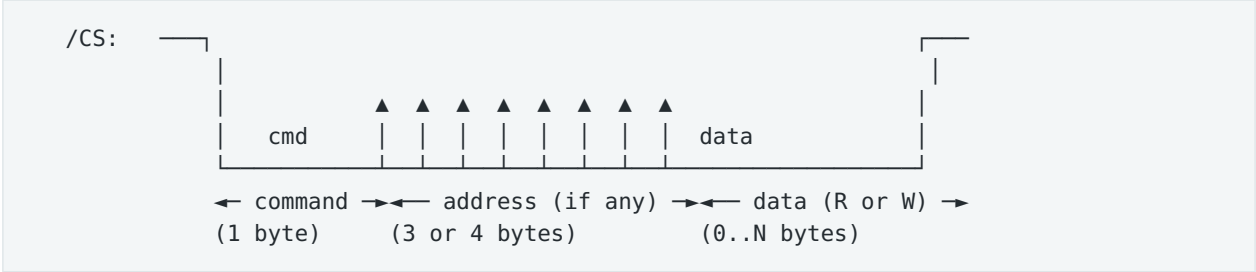
Decoupling: 100 nF + 4.7 µF close to VCC. Page programming draws sharp ~20 mA pulses. Insufficient decoupling causes spurious resets.

Layout: ≤ 5 mm length-mismatch between SCLK and IO[0:3] traces at 80 MHz. Series termination 33 Ω on each line is common. Keep traces ≤ 4 cm.

Pull-ups: 10 kΩ on IO2 and IO3. These pull-ups keep IO2 and IO3 HIGH while the chip is still in single-IO mode at boot. Quad mode is not enabled yet, so IO2 acts as /WP and IO3 as /HOLD. Both are active-low, so “HIGH” means “not asserted.”

64.4 The protocol, what’s on the wire

NOR flash chips speak a small command set. Each transaction is the same shape:



The host:

1. Drives /CS low.
2. Clocks out a 1-byte command.
3. (Optionally) clocks out the address bytes, 3 bytes for chips ≤ 16 MB, 4 bytes for larger or when explicitly in 4-byte mode.
4. Either clocks out write data (program) or clocks in read data.
5. Drives /CS high.

The command set you’ll use

Op	Command byte	Address?	Data direction	Notes
Read JEDEC ID	`0x9F`	none	3 bytes in (mfr+type+capacity)	identify chip
Read status register 1	`0x05`	none	1 byte in	bit 0 = BUSY
Write enable	`0x06`	none	none	required before any write/erase
Write disable	`0x04`	none	none	optional cleanup

Op	Command byte	Address?	Data direction	Notes
Read data (slow)	`0x03`	3 bytes	N bytes in	≤ 50 MHz
Fast read	`0x0B`	3 bytes + 1 dummy	N bytes in	up to chip max
Page program	`0x02`	3 bytes	up to 256 B out	within one 256-B page
Sector erase (4 KB)	`0x20`	3 bytes	none	takes ~50 ms
Block erase (64 KB)	`0xD8`	3 bytes	none	takes ~150 ms
Chip erase	`0xC7`	none	none	tens of seconds
4-byte address mode	`0xB7`	none	none	switch to 4-byte addressing

That is the full interface. Some chips add quad-IO commands (`0xEB` for read, `0x32` for program). Same idea but with data spread over 4 IO lanes for ~4× throughput.

Three invariants that catch beginners

1. **Every write/erase must be preceded by `0x06` (write-enable).** The chip silently ignores writes without it. After a successful write/erase, the chip auto-clears write-enable, so it must be re-issued each time.
2. **The BUSY bit (status register bit 0) is set during write/erase and clears when done.** The host must poll until it clears before issuing the next command. A page program can take 700 µs–3 ms. A sector erase takes 50–250 ms.
3. **You can only program 1→0.** Bits cannot be flipped 0→1 except via erase. So “writing” a byte first requires the relevant sector to have been erased to all-0xFF.

64.5 Reading JEDEC ID, the smallest useful transaction

Before anything else, identify the chip:

```
   Host                    Chip
   ----                    ----
   /CS↓
   send 0x9F  (read JEDEC ID command)
   receive 3 bytes:
       byte 0: manufacturer (0xEF Winbond, 0xC2 Macronix, 0x20 Micron, ...)
       byte 1: memory type   (e.g., 0x40 for W25Q SPI flash family)
       byte 2: capacity      (0x18 = 2^24 bytes = 16 MB; 0x19 = 32 MB; 0x1A = 64 MB)
   /CS↑
```

From these three bytes the chip is uniquely identified (in 99% of cases). The mainline `spi-nor` driver does exactly this and then looks the triple up in its database.

64.6 How the mainline `spi-nor` driver works internally

Source: `drivers/mtd/spi-nor/`.

> **MTD:** Memory Technology Device, Linux's raw flash subsystem for eraseblock-based storage.

```
drivers/mtd/spi-nor/
├── core.c        ← framework: read/write/erase loops, sector tracking
├── sfdp.c        ← SFDP (Serial Flash Discoverable Parameters) parser
├── swp.c         ← software write-protect helpers
├── otp.c         ← one-time-programmable region support
├── debugfs.c     ← /sys/kernel/debug/spi-nor/
├── winbond.c     ← parameter table for Winbond chips
├── macronix.c    ← Macronix table
├── micron-st.c   ← Micron + ST + GigaDevice + ISSI tables
├── ... (15+ vendor files)
```

The driver is two-layer: a vendor-agnostic core that handles read / erase / write loops, plus per-vendor parameter tables describing each chip's quirks (size, fast-read opcode choice, quad-enable bit position, etc.).

Driver flow at probe

```
/* drivers/mtd/spi-nor/core.c — simplified */
static int spi_nor_probe(struct spi_mem *spimem)
{
    struct spi_nor *nor = ...;

    /* 1. Issue 0x9F, read 3 bytes */
    err = spi_nor_read_id(nor, /* offset */ 0, /* len */ 3, id);

    /* 2. Walk the database for a match */
    for (each registered chip in winbond_nor_parts, macronix_nor_parts, ...) {
        if (memcmp(id, info->id, 3) == 0) {
            nor->info = info;
            break;
```

```
        }
    }

    /* 3. Optionally read SFDP for runtime discovery of params not in static table */
    if (info->parse_sfdp || !info)
        spi_nor_parse_sfdp(nor);

    /* 4. Enable quad mode if the chip supports it (chip-specific QE bit) */
    spi_nor_quad_enable(nor);

    /* 5. Register an MTD device */
    nor->mtd.name      = "nor";
    nor->mtd.size      = info->sector_size * info->n_sectors;
    nor->mtd._read     = spi_nor_read;
    nor->mtd._write    = spi_nor_write;
    nor->mtd._erase    = spi_nor_erase;
    return mtd_device_parse_register(&nor->mtd, ...);
}
```

That's the architecture: identify → look up parameters → register as MTD device. The rest is MTD's problem.

How `spi_nor_read` works

```
/* Simplified */
static int spi_nor_read(struct mtd_info *mtd, loff_t from, size_t len,
                        size_t *retlen, u_char *buf)
{
    struct spi_nor *nor = mtd_to_spi_nor(mtd);

    while (len > 0) {
        size_t chunk = min(len, /* controller transfer limit */ 4096);

        struct spi_mem_op op = SPI_MEM_OP(
            SPI_MEM_OP_CMD(nor->read_opcode, 1),       /* cmd byte */
            SPI_MEM_OP_ADDR(nor->addr_nbytes, from, 1),/* address */
            SPI_MEM_OP_DUMMY(nor->read_dummy, 1),       /* dummy cycles for fast-read */
            SPI_MEM_OP_DATA_IN(chunk, buf, 1));         /* read data */

        err = spi_mem_exec_op(nor->spimem, &op);
        if (err) return err;

        buf  += chunk;
        from += chunk;
        len  -= chunk;
        *retlen += chunk;
    }
    return 0;
}
```

The `spi_mem_op` struct describes a NOR-style command in four fields (cmd, addr, dummy, data), `spi_mem_exec_op` translates that into whatever the underlying SPI/QSPI controller wants. For a controller like i.MX QSPI that has hardware support for "command +

address + data” transactions, it programs a few registers and DMA-receives the data. For a plain SPI controller, it falls back to bit-banging the same sequence through ordinary `spi_messages`.

> **MCU bridge:** Think of DMA like the MCU DMA controller you used for UART or SPI, but with cache coherency, scatter-gather descriptors, and kernel ownership rules added. **DMA:** Direct Memory Access. Hardware moves data to or from memory without the CPU copying each byte.

How `spi_nor_write` works

Write is *much* more involved because of erase-before-write, page boundaries, and BUSY polling:

```
static int spi_nor_write(struct mtd_info *mtd, loff_t to, size_t len,
                         size_t *retlen, const u_char *buf)
{
    struct spi_nor *nor = mtd_to_spi_nor(mtd);

    while (len > 0) {
        /* 1. Compute how much we can write in this page (256 B max, less if not aligned)
*/
        size_t page_offset = to & (nor->page_size - 1);
        size_t chunk       = min(len, nor->page_size - page_offset);

        /* 2. Issue write-enable */
        spi_nor_write_enable(nor);    /* sends 0x06 */

        /* 3. Issue page program: 0x02 + addr + up to 256 data bytes */
        struct spi_mem_op op = SPI_MEM_OP(
            SPI_MEM_OP_CMD(nor->program_opcode, 1),
            SPI_MEM_OP_ADDR(nor->addr_nbytes, to, 1),
            SPI_MEM_OP_NO_DUMMY,
            SPI_MEM_OP_DATA_OUT(chunk, buf, 1));
        err = spi_mem_exec_op(nor->spimem, &op);

        /* 4. Poll status until BUSY clears */
        err = spi_nor_wait_till_ready(nor);   /* sends 0x05, reads until bit0 = 0 */

        buf += chunk; to += chunk; len -= chunk; *retlen += chunk;
    }
    return 0;
}
```

Step 1’s page-boundary clamp matters because NOR chips wrap *within a page*. If you program at address 0xFE with 4 bytes of data, the last 2 bytes go to 0xFE and 0xFF, then the next 2 bytes wrap back to 0x00 and 0x01 of the same page. Silent corruption. The driver splits long writes at page boundaries.

How erase works

```
static int spi_nor_erase_sector(struct spi_nor *nor, u32 addr)
{
    spi_nor_write_enable(nor);

    struct spi_mem_op op = SPI_MEM_OP(
        SPI_MEM_OP_CMD(nor->erase_opcode, 1),       /* 0x20 = sector erase */
        SPI_MEM_OP_ADDR(nor->addr_nbytes, addr, 1),
        SPI_MEM_OP_NO_DUMMY,
        SPI_MEM_OP_NO_DATA);
    spi_mem_exec_op(nor->spimem, &op);

    return spi_nor_wait_till_ready(nor);    /* may take 50 ms */
}
```

For a longer range, the MTD-level `erase` callback walks the range and issues one of `0x20` (4 KB) / `0x52` (32 KB) / `0xD8` (64 KB) per granularity that aligns, minimizing the total number of erase commands.

Status polling

```
static int spi_nor_wait_till_ready(struct spi_nor *nor)
{
    unsigned long deadline = jiffies + msecs_to_jiffies(40000);  /* 40 s max */
    while (time_before(jiffies, deadline)) {
        u8 sr;
        struct spi_mem_op op = SPI_MEM_OP(
            SPI_MEM_OP_CMD(SPINOR_OP_RDSR, 1),     /* 0x05 */
            SPI_MEM_OP_NO_ADDR, SPI_MEM_OP_NO_DUMMY,
            SPI_MEM_OP_DATA_IN(1, &sr, 1));
        spi_mem_exec_op(nor->spimem, &op);
        if (!(sr & 1))      /* BUSY bit clear */
            return 0;
        cond_resched();
    }
    return -ETIMEDOUT;
}
```

This polling loop is what makes the framework work on Linux: it yields the CPU between checks. For a chip-erase (tens of seconds), the loop runs for a long time but the kernel stays responsive because of `cond_resched`.

64.7 Writing a NOR-flash driver from scratch

To prove we understand the chip, let’s write a minimal driver that bypasses the `spi-nor` framework entirely. Just a plain `spi_driver` that exposes a `/dev/myflash` char device with `read` / `write` / `ioctl` for erase. ~250 lines.

`myflash.c`:

```
#include <linux/init.h>
#include <linux/module.h>
#include <linux/spi/spi.h>
#include <linux/cdev.h>
#include <linux/fs.h>
#include <linux/uaccess.h>
#include <linux/slab.h>
#include <linux/delay.h>

#define CHIP_SIZE     (16 * 1024 * 1024)   /* 16 MB = W25Q128 */
#define SECTOR_SIZE   4096
#define PAGE_SIZE_N   256

#define CMD_READ_ID       0x9F
#define CMD_READ_STATUS   0x05
#define CMD_WRITE_ENABLE  0x06
#define CMD_READ          0x03
#define CMD_PAGE_PROGRAM  0x02
#define CMD_SECTOR_ERASE  0x20

#define IOC_ERASE_SECTOR  _IO('M', 1)

struct myflash {
    struct spi_device *spi;
    struct cdev cdev;
    dev_t devid;
    struct class *class;
    struct mutex lock;
};

/* === Low-level: send one command + (optional) address + (optional) data === */

static int mf_xfer(struct myflash *m, u8 cmd, u32 addr, bool has_addr,
                   u8 *tx_data, u8 *rx_data, size_t data_len)
{
    u8 hdr[5];
    int hdr_len = 1;
    struct spi_transfer xfers[2] = {0};
    struct spi_message msg;

    hdr[0] = cmd;
    if (has_addr) {
        hdr[1] = (addr >> 16) & 0xff;
        hdr[2] = (addr >>  8) & 0xff;
        hdr[3] =  addr        & 0xff;
        hdr_len = 4;
    }

    spi_message_init(&msg);

    xfers[0].tx_buf = hdr;
    xfers[0].len    = hdr_len;
    spi_message_add_tail(&xfers[0], &msg);

    if (data_len > 0) {
        xfers[1].tx_buf = tx_data;
        xfers[1].rx_buf = rx_data;
        xfers[1].len    = data_len;
```

```
        spi_message_add_tail(&xfers[1], &msg);
    }

    return spi_sync(m->spi, &msg);
}

static int mf_read_id(struct myflash *m, u8 id[3])
{
    return mf_xfer(m, CMD_READ_ID, 0, false, NULL, id, 3);
}

static int mf_write_enable(struct myflash *m)
{
    return mf_xfer(m, CMD_WRITE_ENABLE, 0, false, NULL, NULL, 0);
}

static int mf_read_status(struct myflash *m, u8 *sr)
{
    return mf_xfer(m, CMD_READ_STATUS, 0, false, NULL, sr, 1);
}

static int mf_wait_ready(struct myflash *m)
{
    int retries = 1000;     /* ~1 s @ ~1 ms each */
    while (retries-- > 0) {
        u8 sr;
        int err = mf_read_status(m, &sr);
        if (err) return err;
        if (!(sr & 1))      /* BUSY clear */
            return 0;
        msleep(1);
    }
    return -ETIMEDOUT;
}

/* === Mid-level: read N bytes / program 1 page / erase 1 sector === */

static int mf_read(struct myflash *m, u32 addr, u8 *buf, size_t len)
{
    return mf_xfer(m, CMD_READ, addr, true, NULL, buf, len);
}

static int mf_page_program(struct myflash *m, u32 addr, const u8 *buf, size_t len)
{
    int err;
    if (len == 0 || len > PAGE_SIZE_N) return -EINVAL;
    if ((addr & (PAGE_SIZE_N - 1)) + len > PAGE_SIZE_N) return -EINVAL;

    err = mf_write_enable(m);                           if (err) return err;
    err = mf_xfer(m, CMD_PAGE_PROGRAM, addr, true,
                  (u8 *)buf, NULL, len);                if (err) return err;
    return mf_wait_ready(m);
}

static int mf_erase_sector(struct myflash *m, u32 addr)
{
    int err;
    if (addr & (SECTOR_SIZE - 1)) return -EINVAL;
    err = mf_write_enable(m);                           if (err) return err;
```

```
    err = mf_xfer(m, CMD_SECTOR_ERASE, addr, true, NULL, NULL, 0);
    if (err) return err;
    return mf_wait_ready(m);
}

/* === Char-device interface === */

static int mf_open(struct inode *inode, struct file *filp)
{
    struct myflash *m = container_of(inode->i_cdev, struct myflash, cdev);
    filp->private_data = m;
    return 0;
}

static ssize_t mf_fops_read(struct file *filp, char __user *u,
                            size_t count, loff_t *ppos)
{
    struct myflash *m = filp->private_data;
    u8 *kbuf;
    int err;

    if (*ppos >= CHIP_SIZE) return 0;
    if (*ppos + count > CHIP_SIZE) count = CHIP_SIZE - *ppos;
    if (count > 65536) count = 65536;   /* cap per call */

    kbuf = kmalloc(count, GFP_KERNEL);
    if (!kbuf) return -ENOMEM;

    mutex_lock(&m->lock);
    err = mf_read(m, *ppos, kbuf, count);
    mutex_unlock(&m->lock);
    if (err) { kfree(kbuf); return err; }

    if (copy_to_user(u, kbuf, count)) { kfree(kbuf); return -EFAULT; }
    kfree(kbuf);
    *ppos += count;
    return count;
}

static ssize_t mf_fops_write(struct file *filp, const char __user *u,
                             size_t count, loff_t *ppos)
{
    struct myflash *m = filp->private_data;
    u8 *kbuf;
    int err;
    size_t done = 0;

    if (*ppos >= CHIP_SIZE) return -ENOSPC;
    if (*ppos + count > CHIP_SIZE) count = CHIP_SIZE - *ppos;

    kbuf = kmalloc(count, GFP_KERNEL);
    if (!kbuf) return -ENOMEM;
    if (copy_from_user(kbuf, u, count)) { kfree(kbuf); return -EFAULT; }

    mutex_lock(&m->lock);
    while (done < count) {
        size_t page_off = (*ppos + done) & (PAGE_SIZE_N - 1);
        size_t chunk    = min((size_t)(PAGE_SIZE_N - page_off), count - done);
        err = mf_page_program(m, *ppos + done, kbuf + done, chunk);
```

```
        if (err) break;
        done += chunk;
    }
    mutex_unlock(&m->lock);
    kfree(kbuf);
    if (err) return err;

    *ppos += done;
    return done;
}

static long mf_fops_ioctl(struct file *filp, unsigned int cmd, unsigned long arg)
{
    struct myflash *m = filp->private_data;
    int err;

    if (cmd == IOC_ERASE_SECTOR) {
        u32 addr = (u32)arg;
        mutex_lock(&m->lock);
        err = mf_erase_sector(m, addr);
        mutex_unlock(&m->lock);
        return err;
    }
    return -ENOTTY;
}

static loff_t mf_fops_llseek(struct file *filp, loff_t off, int whence)
{
    return fixed_size_llseek(filp, off, whence, CHIP_SIZE);
}

static const struct file_operations mf_fops = {
    .owner   = THIS_MODULE,
    .open    = mf_open,
    .read    = mf_fops_read,
    .write   = mf_fops_write,
    .llseek  = mf_fops_llseek,
    .unlocked_ioctl = mf_fops_ioctl,
};

/* === Probe / Remove === */

static int mf_probe(struct spi_device *spi)
{
    struct myflash *m;
    u8 id[3];
    int err;

    m = devm_kzalloc(&spi->dev, sizeof(*m), GFP_KERNEL);
    if (!m) return -ENOMEM;
    m->spi = spi;
    mutex_init(&m->lock);

    spi->mode = SPI_MODE_0;
    spi->bits_per_word = 8;
    err = spi_setup(spi);
    if (err) return dev_err_probe(&spi->dev, err, "spi_setup failed\n");

    err = mf_read_id(m, id);
```

```
    if (err) return dev_err_probe(&spi->dev, err, "read-id failed\n");
    dev_info(&spi->dev, "JEDEC ID: %02x %02x %02x\n", id[0], id[1], id[2]);

    if (id[0] != 0xEF || id[2] != 0x18)
        return dev_err_probe(&spi->dev, -ENODEV,
                             "expected W25Q128 (EF xx 18), got %02x %02x %02x\n",
                             id[0], id[1], id[2]);

    /* Register chardev */
    err = alloc_chrdev_region(&m->devid, 0, 1, "myflash");
    if (err) return err;
    cdev_init(&m->cdev, &mf_fops);
    m->cdev.owner = THIS_MODULE;
    err = cdev_add(&m->cdev, m->devid, 1);
    if (err) goto unreg;

    m->class = class_create("myflash");
    if (IS_ERR(m->class)) { err = PTR_ERR(m->class); goto del_cdev; }
    device_create(m->class, NULL, m->devid, NULL, "myflash");

    spi_set_drvdata(spi, m);
    return 0;

del_cdev:
    cdev_del(&m->cdev);
unreg:
    unregister_chrdev_region(m->devid, 1);
    return err;
}

static void mf_remove(struct spi_device *spi)
{
    struct myflash *m = spi_get_drvdata(spi);
    device_destroy(m->class, m->devid);
    class_destroy(m->class);
    cdev_del(&m->cdev);
    unregister_chrdev_region(m->devid, 1);
}

static const struct of_device_id mf_of_match[] = {
    { .compatible = "linuxlearn,myflash" },
    { }
};
MODULE_DEVICE_TABLE(of, mf_of_match);

static struct spi_driver mf_driver = {
    .driver = {
        .name = "myflash",
        .of_match_table = mf_of_match,
    },
    .probe  = mf_probe,
    .remove = mf_remove,
};
module_spi_driver(mf_driver);

MODULE_LICENSE("GPL");
```

DT to test it:

```
&ecspi3 {
    flash@0 {
        compatible = "linuxlearn,myflash";
        reg = <0>;
        spi-max-frequency = <20000000>;
    };
};
```

Note we're using ordinary SPI (`ecspi3`), not QSPI. The raw protocol works at single-IO mode up to ~50 MHz. Using QSPI would require either an MMIO-driver for the i.MX QSPI controller's command-mode registers (more involved) or going through `spi_mem` (which is exactly what we said we were skipping). For this teaching example, ordinary SPI is the right choice.

> **MMIO:** memory-mapped I/O, where software accesses peripheral registers through normal load and store instructions.

Example build and access:

```
[root@pa-mini:~]# insmod myflash.ko
[root@pa-mini:~]# dmesg | tail
myflash spi3.0: JEDEC ID: ef 40 18
[root@pa-mini:~]# ls /dev/myflash
/dev/myflash

# Read sector 0 (4 KB at offset 0):
[root@pa-mini:~]# dd if=/dev/myflash bs=4096 count=1 | hexdump -C | head
00000000  ff ff ff ff ff ff ff ff ...    (virgin sector)

# Erase, write, read back:
[root@pa-mini:~]# echo -n "Hello, NOR!" > /tmp/msg
[root@pa-mini:~]# ./myflash-ioctl /dev/myflash --erase 0          # uses ioctl
IOC_ERASE_SECTOR
[root@pa-mini:~]# dd if=/tmp/msg of=/dev/myflash bs=11 conv=notrunc
[root@pa-mini:~]# dd if=/dev/myflash bs=11 count=1
Hello, NOR!
```

(The `myflash-ioctl` is a tiny userspace helper that does `ioctl(fd, _IO('M', 1), addr)`. Build alongside.)

What we got, in ~250 lines:

- Identify the chip by JEDEC ID.
- Read arbitrary offsets.
- Page-aligned writes (with the in-driver split-at-page-boundary loop).
- Sector erase via ioctl.
- A chardev `/dev/myflash`.

- Mutex protection against concurrent access.

What we *skipped* compared to `spi-nor`:

- MTD integration (so no partitions, no `fw_setenv`, no boot-from-flash for U-Boot).

> **MCU bridge:** Think of U-Boot like a much larger boot stub plus debug monitor: it initializes hardware, loads the next image, and gives you commands before Linux starts. **U-Boot:** the bootloader that initializes enough hardware to load and start the Linux kernel.

- Quad-IO mode (4× faster).
- SFDP auto-discovery (works for any chip, not just W25Q128).
- 4-byte address mode for > 16 MB chips.
- Software write-protect ranges.
- Sleep/resume PM.
- DMA for transfers > one SPI controller burst.

That gap is what the mainline framework gives you. For a one-off chip, the from-scratch version is enough. For any product going to market, use the mainline driver.

64.8 Now: how to use the mainline driver

With the protocol and the from-scratch driver demystified, the mainline driver is just convenience layers over the same operations.

DT:

```
&qspi {
    pinctrl-names = "default";
    pinctrl-0 = <&pinctrl_qspi>;
    status = "okay";

    flash@0 {
        compatible = "winbond,w25q128", "jedec,spi-nor";
        reg = <0>;
        spi-max-frequency = <80000000>;
        spi-rx-bus-width = <4>;       /* QSPI quad */
        spi-tx-bus-width = <4>;

        partitions {
            compatible = "fixed-partitions";
            #address-cells = <1>;
            #size-cells = <1>;

            partition@0 {
```

```
                label = "u-boot";
                reg = <0x000000 0x100000>;       /* 1 MB */
                read-only;
            };
            partition@100000 {
                label = "u-boot-env";
                reg = <0x100000 0x010000>;       /* 64 KB */
            };
            partition@110000 {
                label = "dtb";
                reg = <0x110000 0x010000>;
            };
            partition@120000 {
                label = "kernel-a";
                reg = <0x120000 0x600000>;       /* 6 MB */
            };
            partition@720000 {
                label = "kernel-b";
                reg = <0x720000 0x600000>;
            };
            partition@d20000 {
                label = "user-data";
                reg = <0xd20000 0x2e0000>;
            };
        };
    };
};
```

For Macronix / Micron:

```
compatible = "macronix,mx25l25645g", "jedec,spi-nor";
/* or */
compatible = "micron,mt25ql256", "jedec,spi-nor";
```

The `"jedec,spi-nor"` fallback means the driver runs and reads JEDEC ID at probe. If the first compatible isn’t recognised, the database fallback identifies the chip from the ID anyway.

After boot:

```
[root@pa-mini:~]# cat /proc/mtd
dev:    size   erasesize  name
mtd0: 00100000 00010000 "u-boot"
mtd1: 00010000 00010000 "u-boot-env"
mtd2: 00010000 00010000 "dtb"
mtd3: 00600000 00010000 "kernel-a"
mtd4: 00600000 00010000 "kernel-b"
mtd5: 002e0000 00010000 "user-data"
```

Read/write/erase from user-space:

```
[root@pa-mini:~]# nanddump /dev/mtd2 > my-dtb.bin     # back up the DTB
[root@pa-mini:~]# flash_erase /dev/mtd3 0 0           # erase kernel-A slot
```

```
[root@pa-mini:~]# nandwrite -p /dev/mtd3 zImage       # program kernel
[root@pa-mini:~]# flashcp -v new-dtb.dtb /dev/mtd2    # erase+write atomically
```

For U-Boot env on `mtd1`:

```
[root@pa-mini:~]# cat /etc/fw_env.config
# Device         Offset    Env size  Sector size
/dev/mtd1        0x0       0x10000   0x10000

[root@pa-mini:~]# fw_setenv bootcmd 'sf probe; sf read ${loadaddr} 0x120000 0x600000;
bootz'
[root@pa-mini:~]# fw_printenv bootcmd
bootcmd=sf probe; sf read ${loadaddr} 0x120000 0x600000; bootz
```

64.9 XIP from QSPI

NOR's signature trick: **eXecute In Place**. The CPU reads instructions directly from QSPI without first copying them to DRAM. I.MX6ULL QSPI maps to a memory window (typically at 0x60000000) where reads transparently fetch from the flash.

- **Pro**: smaller RAM footprint. Faster boot (no copy).
- **Con**: slow (max ~50 MB/s vs. DDR3's ~800 MB/s). Higher power. Instruction-cache misses are expensive.

In practice on i.MX6ULL, XIP is used only for U-Boot (small, fast-boot critical). The kernel and rootfs always run from DRAM. We covered XIP in detail in Ch 11.

> **MCU bridge:** Think of the rootfs as the firmware image's file-backed runtime environment. On an MCU you link everything into flash. On Linux, programs and config live in this mounted tree. **rootfs:** root filesystem, the directory tree mounted at / that contains /bin, /etc, /dev, and libraries.

64.10 Boot from QSPI NOR

U-Boot config sets where to load from:

```
boot_qspi=sf probe; \
  sf read 0x80800000 0x120000 0x600000; \
  sf read 0x83000000 0x110000 0x10000; \
  bootz 0x80800000 - 0x83000000

bootcmd=run boot_qspi
```

- `sf probe` initialises the QSPI controller (U-Boot's own driver).
- `sf read <ram_addr> <flash_offset> <length>` copies from QSPI to RAM.

- `bootz <kernel> - <dtb>` boots.

For A/B updates, define `boot_qspi_a` and `boot_qspi_b` with different `<flash_offset>` values. A boot counter (`fw_setenv boot_count`) + an A/B flag selects which slot.

64.11 Key Notes

- NOR programming, erase and read are different operations with different alignment and timing rules. Write-enable and bounded busy polling belong to the device state machine, not just to a convenient API.
- Page-program boundaries and eraseblock boundaries are separate constraints. Splitting a write correctly does not preserve unrelated data in a sector that must be erased.
- Chip capacity, address mode and supported commands must agree across ROM, U-Boot and Linux. An addressing mode left by one stage can become a dependency of the next.
- MTD partitions and environment tools share a storage-layout contract. Driver access does not make an arbitrary erase or environment save safe.

64.12 Pitfalls

- **Forgetting `write-enable`.** Every write/erase silently no-ops. Status register's WEL bit (bit 1) tells you if write-enable is currently set. Check there if writes mysteriously don't take.
- **Writing without erasing first.** NOR can only flip 1→0. Writing 0xAA to an erased (0xFF) byte gives 0xAA. Writing 0x55 over that 0xAA gives `0xAA & 0x55 = 0x00`, not 0x55. Always erase the sector first.
- **Page-boundary wrap.** Page program wraps within a 256-B page. The split-at-page-boundary loop is mandatory.
- **Status-poll forgotten.** Issuing the next command before BUSY clears = silent failure or chip-state corruption. Always wait.
- **Quad mode enable.** Some chips need their QE (Quad Enable) status bit set before quad-IO commands work. The `spi-nor` driver handles this for known chips. With `"jedec,spi-nor"` only it may not. Symptom: garbage when using quad-mode commands. Add the specific compatible.
- **4-byte addressing for > 16 MB.** Chips > 16 MB need 4 address bytes. The third address byte (high) of a 3-byte command becomes part of the data sent to the chip, silent corruption at offsets > 16 MB. The mainline driver auto-switches via `0xB7` (enter 4-byte mode). Old U-Boots may not.

- **Partition off the chip.** A typo in `reg = <offset size>` past the chip end. Kernel warns. Verify dmesg.
- **U-Boot env partition mismatch with `fw_env.config`.** Both reference the same offset. If they disagree, `fw_setenv` writes to areas U-Boot expects to be unwritten. Always cross-check.
- **U-Boot env wear.** Each `saveenv` erases + writes the env sector (~50 ms wear cycle). NOR rated 100k cycles. If your boot script `saveenv`s every boot, you wear out in months. Use `redundant_env` (two ping-ponged copies) or avoid auto-saving.

64.13 Going deeper

- **`drivers/mtd/spi-nor/core.c`**: read it cover to cover. ~2000 lines. Notice how `spi_nor_read`, `spi_nor_write_data`, `spi_nor_erase_sector` map to the same protocol the from-scratch driver implements.
- **`drivers/mtd/spi-nor/winbond.c`**: the parameter table for your chip. Compare its `n_sectors`, `page_size`, `flags` against the datasheet.
- **`drivers/spi/spi-fsl-qspi.c`**: i.MX QSPI controller driver. Note how it uses an LUT (look-up table) of pre-baked command sequences to accelerate.
- **`Documentation/devicetree/bindings/mtd/jedec,spi-nor.yaml`**: DT binding details.
- **JEDEC SFDP standard (JESD216)**: Serial Flash Discoverable Parameters. The runtime discovery protocol the mainline driver uses for chips not in its static database.
- **W25Q128JV datasheet**, **MX25L25645G datasheet**, **MT25Q datasheet**, protocol reference for the three chips. The command table is on page 1 of each.

> Next chapter: **Chapter 65: I²C / SPI EEPROM.** For when you need just a few KB of persistent storage. Same depth: protocol on the wire, the `at24` driver's internals, a from-scratch I²C-EEPROM driver, then DT enablement.

Chapter 65: I²C / SPI EEPROM

> **What:** small persistent storage chips, bytes addressable, no erase-before-write needed, ~1M write cycles. We'll walk the chip-side protocol byte-by-byte, dissect how the mainline `at24` driver actually works, then write a tiny from-scratch I²C-EEPROM driver. Three chips compared, **Microchip AT24C02** (I²C, 256 B), **AT24C512** (I²C, 64 KB), **25LC512** (SPI, 64 KB).
>
> **Why:** EEPROM is the place embedded boards store small permanent facts about themselves, MAC address, board serial, calibration. The protocol is small. Writing your own driver in 100 lines is realistic and worth the time. After this chapter the kernel's `at24.c` will read as ordinary code, not a mystery. **MAC:** Media Access Control in networking and radio chapters. It is the layer that owns framing and medium access.
>
> **Focus: two protocol gotchas**, (a) page-aligned writes (writing across a page boundary silently wraps within the same page), and (b) the ACK-poll loop (the chip NACKs while internally programming, you poll until it ACKs). Get those two right and the rest is byte arithmetic.

65.1 When EEPROM beats flash, OTP fuses, NVRAM

Use case	Best fit	Why
Permanent board ID, read-only after factory	**OTP / eFuse** (i.MX6ULL OCOTP)	Indestructible; can't be erased
MAC address, serial number, calibration, rare writes	**EEPROM** (I²C / SPI)	~1M cycles, byte-addressable
Frequently updated config (boot env, A/B slot)	**NOR flash** (QSPI; redundant pair)	Larger writes, faster
Bulk storage (firmware, user data)	**NAND flash / eMMC**	GB-scale

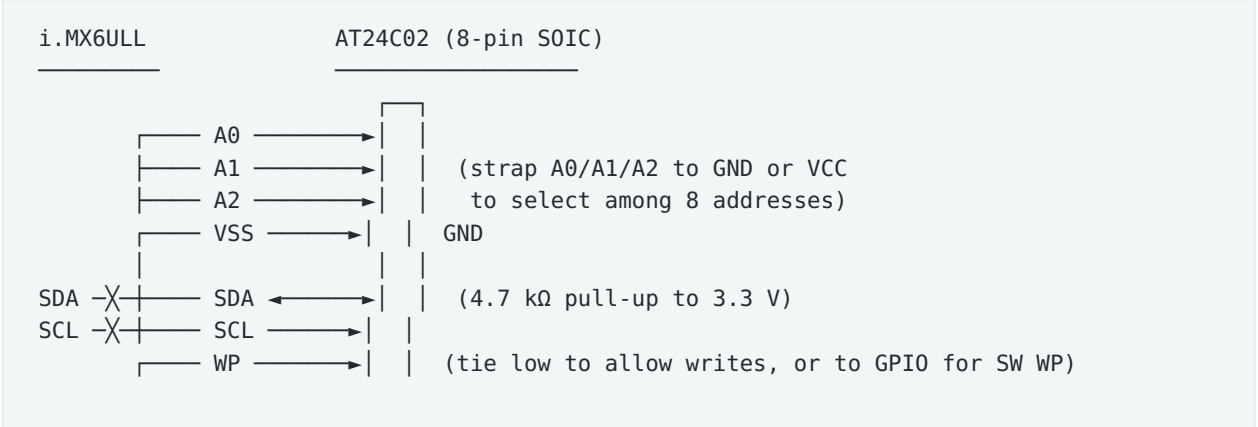
The EEPROM niche: small, byte-addressable, rewriteable but rarely written, kept on the same I²C bus as your sensors anyway. ~$0.30 BOM cost for AT24C02.

65.2 Chip comparison

	AT24C02	AT24C512	25LC512
Bus	I²C	I²C	SPI
Capacity	256 B (2 Kbit)	64 KB (512 Kbit)	64 KB (512 Kbit)
Page size	8 B	128 B	128 B
Write cycle	≤ 5 ms	≤ 5 ms	≤ 5 ms
Endurance	1 M cycles	1 M cycles	1 M cycles
Max bus clock	1 MHz (1.7+ V)	1 MHz	20 MHz
I²C addr	0x50–0x57 (A0/A1/A2 strap)	0x50–0x57	,
Addressing	1 byte	2 bytes	2 bytes
Price	$0.30	$1.00	$1.50

Pick AT24C02 for tiny ID storage (MAC, serial), AT24C512 for calibration tables, 25LC512 for SPI bus or factory-speed bulk programming.

65.3 Schematic, AT24C02 (I²C)



```
        └──── VCC ──────┤  │  3.3 V
                        └──┘
```

For 25LC512 (SPI): MOSI / MISO / SCK / /CS (4 wires) + /WP + /HOLD.

65.4 The I²C EEPROM protocol on the wire

An I²C transaction looks like this for a “read 4 bytes from offset 0x40”:

```
Master:   START | 0xA0 |   |   | 0x40 |   | START | 0xA1 |   |   |   |   |   | STOP
                        ACK ↑         ACK             ACK ↑   ACK  ACK  ACK  NACK
Slave:                 ←┘           ←┘               ←┘   ↓    ↓    ↓    ↓
                                                         D0   D1   D2   D3
```

- `0xA0` is the AT24’s 7-bit address (0x50) shifted left + write bit (0): writing the **register pointer**.
- `0x40` is the byte address inside the EEPROM (since AT24C02 has only 256 bytes, one address byte suffices).
- A *repeated start*, then `0xA1` = address + read bit, then reads 4 bytes with the master ACKing each except the last (NACK signals “I’m done”).

For a write of 4 bytes to offset 0x40:

```
Master:   START | 0xA0 |   | 0x40 |   | D0 |   | D1 |   | D2 |   | D3 |   | STOP
                       ACK         ACK    ACK    ACK    ACK    ACK ↑
Slave:                ←┘          ←┘     ←┘     ←┘     ←┘     ←┘
```

After the data bytes, the chip starts an *internal write cycle* of about 5 ms. During this time it NACKs every I²C transaction. The host keeps issuing address-write transactions. Each NACK means “still writing,” and the first ACK means “done.” This is called **ACK polling**.

For AT24C512 (and larger), the byte address is **2 bytes**: send 0xA0, ACK, addr_high, ACK, addr_low, ACK, then data. Same protocol, one more address byte.

Page boundaries, the bite-you-once trap

The EEPROM’s internal write buffer is one page. Writing 4 bytes starting at offset 0x06 of an 8-byte-page chip:

- Bytes 0x06 and 0x07 go to the buffer.
- Address auto-increments, but **wraps within the page**, not to the next page.
- So byte indexed “0x08” actually writes to **0x00**, and “0x09” to **0x01**.

The fix: split writes at page boundaries. A write of N bytes starting at offset O becomes:

- First chunk: bytes `[O, page_end)` where `page_end = (O + page_size) & ~(page_size - 1)`.
- Wait for ACK after that chunk.
- Next chunk: bytes `[page_end, page_end + page_size)`.
- And so on.

This is the AT24 driver's most important loop. Get it wrong = silent data corruption.

65.5 How the mainline at24 driver works internally

Source: `drivers/misc/eeprom/at24.c` (~1000 lines).

The driver is structured around a **regmap** abstraction (Ch 50) over either `regmap_init_i2c` or `regmap_init_smbus` (for the 1-byte-address case). The regmap config encodes the address width and page size. Then the driver implements two callbacks for `nvmem` (Ch 65.7 below) and that's most of it.

> **MCU bridge:** Think of regmap like a typed wrapper around your read_reg() and write_reg() helpers, with caching, locking, and bus differences handled centrally. **regmap:** a kernel helper that wraps register reads and writes over I2C, SPI, or MMIO.

Probe walk

```
/* drivers/misc/eeprom/at24.c — simplified */
static int at24_probe(struct i2c_client *client)
{
    struct at24_data *at24;
    const struct at24_chip_data *cdata;

    /* 1. Resolve chip parameters: byte_len, page_size, flags, addr_width */
    cdata = at24_get_chip_data(client);  /* from match table or DT */

    at24 = devm_kzalloc(&client->dev, sizeof(*at24), GFP_KERNEL);
    at24->byte_len  = cdata->byte_len;
    at24->page_size = cdata->page_size;

    /* 2. Build a regmap matching this chip's protocol */
    struct regmap_config regmap_config = {
        .val_bits = 8,
        .reg_bits = cdata->flags & AT24_FLAG_ADDR16 ? 16 : 8,
        .disable_locking = true,
    };
    at24->regmap = devm_regmap_init_i2c(client, &regmap_config);

    /* 3. Allocate "regions" — one regmap per I²C address when chip spans multiple */
```

```
    /* (large AT24s like 64 KB take multiple I²C addresses; the driver creates
       a regmap per address and routes reads/writes by offset) */
    at24->bank_addr_shift = ...;
    /* ... */

    /* 4. Set up nvmem-config */
    struct nvmem_config nvmem_config = {
        .name        = ...,
        .type        = NVMEM_TYPE_EEPROM,
        .read_only   = is_writeprotected(at24),
        .word_size   = 1,
        .size        = at24->byte_len,
        .reg_read    = at24_read,
        .reg_write   = at24_write,
        .priv        = at24,
    };
    at24->nvmem = devm_nvmem_register(&client->dev, &nvmem_config);

    return 0;
}
```

The read callback

```
static int at24_read(void *priv, unsigned int off, void *val, size_t count)
{
    struct at24_data *at24 = priv;

    while (count) {
        struct regmap *regmap;
        size_t addr;
        size_t chunk;

        /* Pick the right regmap (i.e., the right I²C address)
         * for chips that span multiple I²C addresses. */
        regmap = at24_select_regmap(at24, &off);
        addr = off & 0xFF;       /* mask to the device's address space */

        /* Cap each chunk to whatever the controller can do in one go */
        chunk = min(count, AT24_MAX_READ);

        int err = regmap_bulk_read(regmap, addr, val, chunk);
        if (err) return err;

        val   += chunk;
        off   += chunk;
        count -= chunk;
    }
    return 0;
}
```

Reads don’t need to respect page boundaries, only writes do. The wrap-within-page behaviour applies to programming, not reading. So `at24_read` is a simple linear loop.

The write callback (the interesting one)

```
static int at24_write(void *priv, unsigned int off, void *val, size_t count)
{
    struct at24_data *at24 = priv;
    int err;

    mutex_lock(&at24->lock);

    while (count) {
        size_t page_offset = off & (at24->page_size - 1);
        size_t in_page     = at24->page_size - page_offset;
        size_t chunk       = min(count, in_page);

        struct regmap *regmap = at24_select_regmap(at24, &off);
        unsigned int addr = off & 0xFF;

        /* Bulk-write up to one page worth at this offset */
        err = regmap_bulk_write(regmap, addr, val, chunk);
        if (err) break;

        /* ACK-poll: issue dummy reads of register 0 until they succeed.
         * The chip NACKs until its internal write cycle finishes (~5 ms). */
        unsigned long deadline = jiffies + msecs_to_jiffies(at24->write_max_ms);
        do {
            u8 dummy;
            err = regmap_read(regmap, 0, &dummy);
            if (err == 0) break;   /* ACK received: write cycle done */
            usleep_range(100, 500);
        } while (time_before(jiffies, deadline));

        if (err) break;

        val   += chunk;
        off   += chunk;
        count -= chunk;
    }

    mutex_unlock(&at24->lock);
    return err;
}
```

Two things to notice:

1. **The page-boundary split**: `chunk = min(count, at24->page_size - page_offset)` ensures we never cross a page boundary in one transaction.
2. **ACK polling**: the loop with `regmap_read(regmap, 0, &dummy)` tests whether the chip ACKs *yet*. While the chip is writing internally, it NACKs every transaction. The loop spins (yielding with `usleep_range`) until it gets an ACK.

That is the whole driver. Around 50 lines of real code. The rest is parameter tables, DT plumbing, and edge cases (multi-address chips, write-protect GPIOs).

65.6 Writing an I²C EEPROM driver from scratch

Let’s prove we understand by writing the minimal version. ~150 lines, no regmap, no nvmem, just an `i2c_driver` + chardev. Targets AT24C02 specifically.

`myeeprom.c`:

```
#include <linux/init.h>
#include <linux/module.h>
#include <linux/i2c.h>
#include <linux/cdev.h>
#include <linux/fs.h>
#include <linux/uaccess.h>
#include <linux/delay.h>

#define EEPROM_SIZE     256
#define EEPROM_PAGE     8
#define WRITE_MAX_MS    25      /* upper bound on internal write cycle + slack */

struct myeeprom {
    struct i2c_client *client;
    struct cdev cdev;
    dev_t devid;
    struct class *class;
    struct mutex lock;
};

/* === Low-level: write reg-address, then read N bytes === */

static int me_read(struct myeeprom *e, u8 off, u8 *buf, size_t count)
{
    struct i2c_msg msgs[2] = {
        { .addr = e->client->addr, .flags = 0,        .len = 1, .buf = &off },
        { .addr = e->client->addr, .flags = I2C_M_RD, .len = count, .buf = buf },
    };
    int n = i2c_transfer(e->client->adapter, msgs, 2);
    return n == 2 ? 0 : (n < 0 ? n : -EIO);
}

/* === Mid-level: write one page (caller must ensure no boundary crossing) === */

static int me_write_page(struct myeeprom *e, u8 off, const u8 *buf, size_t count)
{
    u8 tx[1 + EEPROM_PAGE];
    struct i2c_msg msg = {
        .addr = e->client->addr, .flags = 0,
        .len = count + 1, .buf = tx,
    };
    int n;
    unsigned long deadline;

    if (count == 0 || count > EEPROM_PAGE) return -EINVAL;
    if ((off & (EEPROM_PAGE - 1)) + count > EEPROM_PAGE) return -EINVAL;

    tx[0] = off;
    memcpy(tx + 1, buf, count);
```

```
    n = i2c_transfer(e->client->adapter, &msg, 1);
    if (n != 1) return n < 0 ? n : -EIO;

    /* ACK-poll: keep issuing 1-byte writes (just the address byte) until ACK */
    deadline = jiffies + msecs_to_jiffies(WRITE_MAX_MS);
    while (time_before(jiffies, deadline)) {
        u8 zero = 0;
        struct i2c_msg poll = { .addr = e->client->addr, .flags = 0,
                                .len = 1, .buf = &zero };
        n = i2c_transfer(e->client->adapter, &poll, 1);
        if (n == 1) return 0;
        usleep_range(200, 500);
    }
    return -ETIMEDOUT;
}

/* === Char-device fops === */

static int me_open(struct inode *inode, struct file *filp)
{
    struct myeeprom *e = container_of(inode->i_cdev, struct myeeprom, cdev);
    filp->private_data = e;
    return 0;
}

static ssize_t me_fops_read(struct file *filp, char __user *u,
                            size_t count, loff_t *ppos)
{
    struct myeeprom *e = filp->private_data;
    u8 kbuf[EEPROM_SIZE];
    int err;

    if (*ppos >= EEPROM_SIZE) return 0;
    if (*ppos + count > EEPROM_SIZE) count = EEPROM_SIZE - *ppos;

    mutex_lock(&e->lock);
    err = me_read(e, *ppos, kbuf, count);
    mutex_unlock(&e->lock);
    if (err) return err;

    if (copy_to_user(u, kbuf, count)) return -EFAULT;
    *ppos += count;
    return count;
}

static ssize_t me_fops_write(struct file *filp, const char __user *u,
                             size_t count, loff_t *ppos)
{
    struct myeeprom *e = filp->private_data;
    u8 kbuf[EEPROM_SIZE];
    int err = 0;
    size_t done = 0;

    if (*ppos >= EEPROM_SIZE) return -ENOSPC;
    if (*ppos + count > EEPROM_SIZE) count = EEPROM_SIZE - *ppos;

    if (copy_from_user(kbuf, u, count)) return -EFAULT;

    mutex_lock(&e->lock);
```

```
    while (done < count) {
        size_t page_off = (*ppos + done) & (EEPROM_PAGE - 1);
        size_t chunk    = min((size_t)(EEPROM_PAGE - page_off), count - done);
        err = me_write_page(e, *ppos + done, kbuf + done, chunk);
        if (err) break;
        done += chunk;
    }
    mutex_unlock(&e->lock);
    if (err) return err;

    *ppos += done;
    return done;
}

static loff_t me_fops_llseek(struct file *filp, loff_t off, int whence)
{
    return fixed_size_llseek(filp, off, whence, EEPROM_SIZE);
}

static const struct file_operations me_fops = {
    .owner   = THIS_MODULE,
    .open    = me_open,
    .read    = me_fops_read,
    .write   = me_fops_write,
    .llseek  = me_fops_llseek,
};

/* === Probe / Remove === */

static int me_probe(struct i2c_client *client)
{
    struct myeeprom *e;
    u8 probe_byte;
    int err;

    e = devm_kzalloc(&client->dev, sizeof(*e), GFP_KERNEL);
    if (!e) return -ENOMEM;
    e->client = client;
    mutex_init(&e->lock);

    /* Sanity read: does anything respond at this address? */
    err = me_read(e, 0, &probe_byte, 1);
    if (err) return dev_err_probe(&client->dev, err, "EEPROM not responding\n");
    dev_info(&client->dev, "myeeprom alive at 0x%02x (byte0=0x%02x)\n",
             client->addr, probe_byte);

    err = alloc_chrdev_region(&e->devid, 0, 1, "myeeprom");
    if (err) return err;
    cdev_init(&e->cdev, &me_fops);
    e->cdev.owner = THIS_MODULE;
    err = cdev_add(&e->cdev, e->devid, 1);
    if (err) goto unreg;

    e->class = class_create("myeeprom");
    if (IS_ERR(e->class)) { err = PTR_ERR(e->class); goto del_cdev; }
    device_create(e->class, NULL, e->devid, NULL, "myeeprom");

    i2c_set_clientdata(client, e);
    return 0;
```

```
del_cdev:
    cdev_del(&e->cdev);
unreg:
    unregister_chrdev_region(e->devid, 1);
    return err;
}

static void me_remove(struct i2c_client *client)
{
    struct myeeprom *e = i2c_get_clientdata(client);
    device_destroy(e->class, e->devid);
    class_destroy(e->class);
    cdev_del(&e->cdev);
    unregister_chrdev_region(e->devid, 1);
}

static const struct of_device_id me_of_match[] = {
    { .compatible = "linuxlearn,myeeprom" },
    { }
};
MODULE_DEVICE_TABLE(of, me_of_match);

static const struct i2c_device_id me_id[] = {
    { "myeeprom", 0 },
    { }
};
MODULE_DEVICE_TABLE(i2c, me_id);

static struct i2c_driver me_driver = {
    .driver = {
        .name = "myeeprom",
        .of_match_table = me_of_match,
    },
    .probe    = me_probe,
    .remove   = me_remove,
    .id_table = me_id,
};
module_i2c_driver(me_driver);

MODULE_LICENSE("GPL");
```

DT:

```
&i2c1 {
    eeprom@50 {
        compatible = "linuxlearn,myeeprom";
        reg = <0x50>;
    };
};
```

Example build and access:

```
[root@pa-mini:~]# insmod myeeprom.ko
[root@pa-mini:~]# dmesg | tail -1
myeeprom 1-0050: myeeprom alive at 0x50 (byte0=0xff)
```

```
[root@pa-mini:~]# echo -n "Hello world!" > /dev/myeeprom
[root@pa-mini:~]# dd if=/dev/myeeprom bs=12 count=1
Hello world!
12+0 records in
12+0 records out

[root@pa-mini:~]# hexdump -C /dev/myeeprom | head -1
00000000  48 65 6c 6c 6f 20 77 6f  72 6c 64 21 ff ff ff ff  |Hello world!....|
```

The page-aligned write loop handled the “Hello world!” (12 bytes starting at offset 0, which crosses the 8-byte page boundary) correctly, split into [0..7] and [8..11], each its own page-program with ACK-poll.

What we got, in ~150 lines:

- I²C read/write split correctly at page boundaries.
- ACK polling for write completion.
- Mutex-protected chardev.
- Probe-time sanity check.

What we *skipped* compared to `at24`:

- nvmem integration (so the FEC can’t read MAC from us, we don’t appear under `/sys/bus/nvmem/`).
- Multi-address chips (AT24C512 spans 4 I²C addresses).
- 2-byte addressing (we hardcoded 1-byte AT24C02).
- Write-protect GPIO.

> **MCU bridge:** Think of Linux GPIO like the same pin set/reset block you used on STM32, but accessed through a kernel subsystem that owns numbering, direction, interrupts, and user-space exposure. **GPIO:** General-Purpose Input/Output, a pin controlled as a digital input, output, or interrupt source.

- Sysfs binary attribute (we exposed via /dev only).

> **sysfs:** a kernel-generated filesystem under /sys that exposes devices, drivers, and attributes.

- read-only mode for DT `read-only` boolean.

65.7 Now: the mainline driver and nvmem

DT for the mainline driver:

```
&i2c1 {
    eeprom: eeprom@50 {
        compatible = "atmel,24c02";
        reg = <0x50>;
        pagesize = <8>;
        #address-cells = <1>;
        #size-cells = <1>;

        /* nvmem cells — typed sub-regions */
        mac_address: mac@0 {
            reg = <0x0 0x6>;     /* 6 bytes at offset 0 */
        };
        serial_number: serial@10 {
            reg = <0x10 0x10>;   /* 16 bytes at offset 0x10 */
        };
    };
};

&fec1 {
    nvmem-cells = <&mac_address>;
    nvmem-cell-names = "mac-address";
    /* ... */
};
```

The mainline `at24` driver registers an nvmem provider. The FEC driver consumes the `mac-address` cell at probe, six bytes from offset 0 become eth0's MAC address. **No board-specific kernel code**. Production: write the MAC in the factory test, the kernel picks it up automatically every boot.

For SPI 25LC512:

```
&ecspi3 {
    eeprom@1 {
        compatible = "microchip,25lc512", "atmel,at25";
        reg = <1>;
        spi-max-frequency = <10000000>;
        size = <65536>;
        pagesize = <128>;
        address-width = <16>;
    };
};
```

The `"atmel,at25"` fallback covers most SPI EEPROMs with the same protocol. The explicit `size`, `pagesize`, `address-width` tell the driver the geometry.

After probe, the EEPROM exposes a binary attribute (sysfs-bin):

```
[root@pa-mini:~]# ls /sys/bus/i2c/devices/1-0050/
driver/  eeprom  name  ...
```

```
[root@pa-mini:~]# hexdump -C /sys/bus/i2c/devices/1-0050/eeprom | head
00000000  ff ff ff ff ff ff ff ff  ff ff ff ff ff ff ff ff  |................|

[root@pa-mini:~]# echo -n "SN12345" > /tmp/serial
[root@pa-mini:~]# dd if=/tmp/serial of=/sys/bus/i2c/devices/1-0050/eeprom bs=1 seek=0
[root@pa-mini:~]# dd if=/sys/bus/i2c/devices/1-0050/eeprom bs=1 count=7
SN12345
```

65.8 Factory programming workflow

```
# 1. Generate per-unit serial + MAC
SERIAL=$(uuidgen | tr -d - | cut -c1-12)
MAC=$(printf '02:%02x:%02x:%02x:%02x:%02x' \
      $((RANDOM % 256)) $((RANDOM % 256)) $((RANDOM % 256)) \
      $((RANDOM % 256)) $((RANDOM % 256)))
echo "Serial: $SERIAL  MAC: $MAC"

# 2. Write MAC (6 bytes at offset 0)
echo "$MAC" | awk -F: '{for(i=1;i<=NF;i++) printf "%c", strtonum("0x"$i)}' \
    > /sys/bus/i2c/devices/1-0050/eeprom

# 3. Write serial (16 bytes at offset 0x10)
printf "%-16s" "$SERIAL" | dd of=/sys/bus/i2c/devices/1-0050/eeprom \
    bs=1 seek=16 conv=notrunc 2>/dev/null

# 4. Verify
xxd /sys/bus/i2c/devices/1-0050/eeprom | head -2

# 5. (Optional, 25LC512) drive WP pin high
echo 1 > /sys/class/gpio/.../wp_value
```

After factory: WP held high, field firmware can read but not write. Reboot → kernel’s nvmem cells pull the MAC into the FEC → networking comes up with the right address.

65.9 Key Notes

- EEPROM is useful for small persistent records such as identity and calibration. Its byte-oriented interface still has page-program, completion and endurance constraints.
- Address width, page size and bus addressing belong to the selected chip. Similar part names do not establish identical transaction framing.
- Nvmem separates storage access from the consumers of named data cells. Factory programming and Device Tree must agree on record offsets, lengths and encoding.
- Write protection, interrupted updates and record validation are design choices. A readback of one value is not a complete identity or durability policy.

65.10 Pitfalls

- **Wrong pagesize.** Writes across page boundaries wrap silently. Always set `pagesize = <N>` in DT (mainline) or hardcode correctly (from-scratch). The AT24C512's 128-byte pages are not the same as AT24C02's 8-byte pages.
- **Address-width confusion.** AT24C02 uses 1-byte address. AT24C512 uses 2-byte. The mainline driver derives this from chip size. The from-scratch driver hardcodes one or the other.
- **5 ms write cycle not waited for.** Issuing the next command before the cycle finishes → NACK → kernel error. The ACK-poll loop is mandatory.
- **Multiple EEPROMs on one bus.** Strap each chip's A0/A1/A2 differently to give unique addresses (0x50–0x57). If two chips share an address, neither responds correctly.
- **WP pin floating.** Reads return all-0xFF and writes silently fail. Tie WP low, or wire it to a GPIO that defaults to low.
- **Wrong nvmem cell offset.** Driver reads garbage as the MAC. Cross-check `reg = <offset size>` against your factory-write script.
- **`/WP` and software write-protect register coexist.** Some chips have *both* a hardware /WP pin *and* a software status-register write-protect. Make sure both allow writes.
- **The "WC" pin.** Some EEPROMs name it WC (write control) instead of WP. Same idea, different polarity sometimes, read the datasheet.

65.11 Going deeper

- **`drivers/misc/eeprom/at24.c`**: the production driver. ~1000 lines. Read after writing the from-scratch version above, you'll recognise every block.
- **`drivers/misc/eeprom/at25.c`**: SPI EEPROM driver.
- **`Documentation/devicetree/bindings/eeprom/at24.yaml`**: DT binding.
- **`Documentation/devicetree/bindings/nvmem/nvmem.yaml`**: nvmem provider binding.
- **`Documentation/ABI/testing/sysfs-bus-nvmem`**: nvmem sysfs.

> **ABI:** Application Binary Interface: the calling convention, register use, binary format, and library contract that let separately built code run together.

- **AT24C02 datasheet** (Microchip), has timing diagrams for the ACK-poll loop on page 11.

> Next chapter: **Chapter 66: SD card and eMMC deep dive.** A different beast: you wouldn't write an MMC host controller driver from scratch in 200 lines (the protocol has 40+ commands, multiple state machines, signal-voltage switching, tuning). Instead we'll trace a single `read()` through the kernel's MMC stack, host driver → core → block layer, to see how the layers fit, and inspect EXT_CSD to manage device life. That gets you "understand the framework" without "rewrite the framework."

Chapter 66: SD card and eMMC deep dive

> **Examples and production:** Do not burn fuses, enroll production keys, or sign release images while using these examples. Use throwaway keys and back up the unsigned image plus the key directory before testing irreversible security flows.

> **What:** the **MMC subsystem** that backs both SD cards and eMMC. Speed modes (DS, HS, HS200, HS400, SDR104), the **EXT_CSD** register (the eMMC's metadata block), boot partitions, RPMB, wear monitoring, and the "removable card in a production product" antipattern. Three configurations compared: **µSD on a card slot**, **soldered eMMC at HS200**, **soldered eMMC with secure boot via RPMB**.
>
> **Why:** for most i.MX6ULL products with > 32 MB storage need, the choice is "SD or eMMC." Picking wrong leads to field failures or wasted engineering effort. EMMC is the right choice for production. SD is fine for dev boards. This chapter is mostly the why behind that statement, plus the bring-up + monitoring details. **Compare**: removable SD (cheap, accessible, dies first), soldered eMMC HS200 (200 MB/s, rated for 5+ years of continuous service in industrial-grade parts), eMMC with RPMB (~10 % overhead, replay-protected for secure boot).

66.1 SD card vs eMMC, the production reality

SD cards fail often, in ways that surprise engineers used to flash chips:

- **No wear levelling guarantees**: The card's controller does *some*, but cheap cards (the kind whose price you negotiate down) skimp on it.
- **Power-loss corruption**: A write in progress when power drops can corrupt the entire FAT/ext4 metadata, not just the in-flight sector. EMMC has better resilience (built-in caching with battery backup in some packages), but SD cards are notoriously fragile.
- **Temperature**: Industrial-grade SD exists (-40 to +85 °C) but consumer ones spec 0–70.
- **Counterfeit**: The single biggest reliability issue. A "32 GB SanDisk" purchased on a forum reseller might be a 4 GB chip with the controller faked to report 32 GB. Writes beyond 4 GB silently fail. Even reputable distributors get hit with fakes.

For a dev board or a product where the user *expects* removable media (a kiosk that takes SD-card content updates), SD is fine. **For any product that's supposed to "just work" for years untouched, solder an eMMC.**

eMMC advantages:

- Soldered = no socket reliability issues.
- Built-in wear leveling, bad-block management, controller-level health monitoring (EXT_CSD).
- Faster (HS200 ~200 MB/s vs SD DDR50 ~50 MB/s).
- Authenticated boot partitions (RPMB).
- Industrial grade available, AEC-Q100 for automotive.

The cost difference is < $1 in volume. Decide between SD and eMMC at schematic time, not after the first field unit fails.

66.2 i.MX6ULL MMC hardware

The SoC has **two uSDHC controllers**: uSDHC1 and uSDHC2. Each supports:

- SD cards up to SDR104 (~104 MB/s).
- eMMC up to HS200 (~200 MB/s).
- SDIO (used by SDIO WiFi modules, Ch 91).
- 1, 4, or 8-bit data widths.

Typically Point Atom boards wire uSDHC1 to the SD slot, uSDHC2 to onboard WiFi or eMMC.

66.3 Speed modes

Mode	Clock	Data lines	Bandwidth (theoretical)	Required pinmux
DS (Default Speed)	25 MHz	4-bit	12.5 MB/s	basic
HS (High Speed)	50 MHz	4-bit	25 MB/s	basic

Mode	Clock	Data lines	Bandwidth (theoretical)	Required pinmux
DDR50	50 MHz double-rate	4-bit	50 MB/s	basic
SDR50	100 MHz	4-bit	50 MB/s	basic
SDR104	200 MHz	4-bit	100 MB/s	UHS-I voltage switch
HS200 (eMMC)	200 MHz SDR	8-bit	200 MB/s	8-bit, 1.8 V signal
HS400 (eMMC)	200 MHz DDR	8-bit	400 MB/s	strobe-aware (i.MX6ULL does NOT support HS400)

i.MX6ULL supports up to HS200 (200 MB/s on 8-bit eMMC) and SDR104 (104 MB/s on 4-bit SD). HS400 is i.MX8M+ territory.

66.4 Device tree, uSDHC2 with eMMC

```
&usdhc2 {
    pinctrl-names = "default", "state_100mhz", "state_200mhz";
    pinctrl-0 = <&pinctrl_usdhc2>;
    pinctrl-1 = <&pinctrl_usdhc2_100mhz>;
    pinctrl-2 = <&pinctrl_usdhc2_200mhz>;
    bus-width = <8>;
    non-removable;
    mmc-hs200-1_8v;
    keep-power-in-suspend;
    no-sd;
    no-sdio;
    vmmc-supply = <&reg_emmc_3v3>;
    vqmmc-supply = <&reg_emmc_1v8>;
    status = "okay";
};
```

Critical pieces:

- **`bus-width = <8>`**: eMMC’s 8 data lines. Gets HS200’s full bandwidth.
- **`non-removable`**: kernel knows not to poll for card removal.

- **`mmc-hs200-1_8v`**: declares HS200 mode is supported (requires switching VQMMC to 1.8 V).
- **`vqmmc-supply`**: the I/O voltage rail. Needs to support 1.8 V for HS200.
- **`pinctrl-1` and `-2`**: different pin slew rates at higher speeds (pull strengths change).
- **`no-sd`, `no-sdio`**: speeds up probing. We know this is eMMC.

For an SD slot on uSDHC1:

```
&usdhc1 {
    pinctrl-names = "default", "state_100mhz", "state_200mhz";
    pinctrl-0 = <&pinctrl_usdhc1>;
    pinctrl-1 = <&pinctrl_usdhc1_100mhz>;
    pinctrl-2 = <&pinctrl_usdhc1_200mhz>;
    bus-width = <4>;
    cd-gpios = <&gpio1 19 GPIO_ACTIVE_LOW>;
    keep-power-in-suspend;
    vmmc-supply = <&reg_sd1_vmmc>;
    sd-uhs-sdr104;
    no-mmc;
    status = "okay";
};
```

`cd-gpios` declares the card-detect line (a switch in the SD socket).

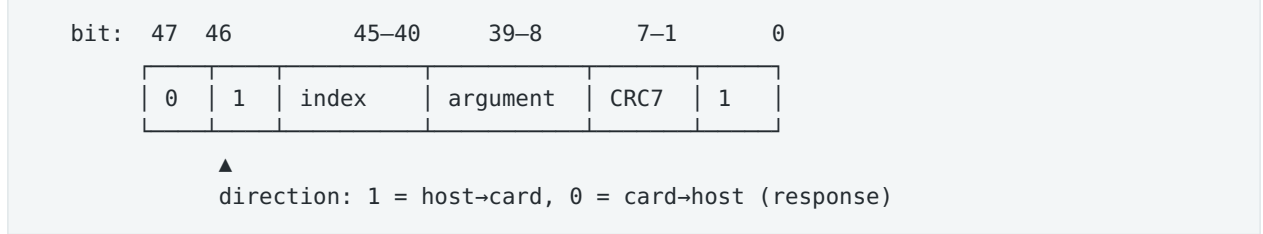
66.5 The MMC protocol, what's actually on the wire

Unlike QSPI and EEPROM (Ch 64/65), where a from-scratch driver was tractable in ~200 lines, an MMC/SD **host controller** driver is genuinely a different scale. The SD spec is ~700 pages. The eMMC spec is ~400. There are ~60 commands, multi-stage state machines, signal-voltage switching, tuning windows, CRC validation, and physical-layer subtleties. **Writing one from scratch in a single chapter is not realistic.**

What we *can* do is **trace a single `read()` through the layers** so you understand exactly what the kernel does, and you can read the existing host driver's source after.

The protocol vocabulary

MMC/SD commands are 48-bit packets:



A handful of commands you'll see in dmesg or in driver code:

Cmd	Name	Purpose
CMD0	GO_IDLE_STATE	Reset card to idle
CMD8	SEND_IF_COND	Check SD 2.0 voltage range (SD vs eMMC discrimination)
ACMD41	SD_SEND_OP_COND	SD initialization, voltage window
CMD1	SEND_OP_COND	eMMC initialization equivalent
CMD2	ALL_SEND_CID	Read card identification
CMD3	SET_RELATIVE_ADDR	Assign a relative address (RCA)
CMD7	SELECT_CARD	Select the addressed card for I/O
CMD8 (eMMC)	SEND_EXT_CSD	Read 512-byte EXT_CSD block
CMD17	READ_SINGLE_BLOCK	Read one block (512 B)
CMD18	READ_MULTIPLE_BLOCK	Read consecutive blocks until CMD12
CMD23	SET_BLOCK_COUNT	Pre-declare a multi-block count (skip CMD12)
CMD24	WRITE_BLOCK	Write one block
CMD25	WRITE_MULTIPLE_BLOCK	Write consecutive blocks
CMD12	STOP_TRANSMISSION	End a CMD18/CMD25 sequence
CMD13	SEND_STATUS	Read card status register

Cmd	Name	Purpose
CMD6	SWITCH	Set EXT_CSD fields (mode change, partition switch)
CMD21 (eMMC)	SEND_TUNING_BLOCK	HS200/HS400 tuning pattern

Data transfers happen on the data lines in parallel with commands on the CMD line. A successful read of 4 blocks is roughly:

```
Host → Card:  CMD17 (READ_SINGLE_BLOCK) | start_block_addr
Card → Host:  R1 response (status, OK)
Card → Host:  DAT lines: 512 bytes of data + CRC16
```

For multi-block (CMD18), one command triggers a stream of consecutive blocks until the host issues CMD12 to stop, much more efficient than per-block CMD17s.

The state machine

Cards live in one of these states: **Idle → Ready → Identification → Standby → Transfer → Sending-data / Receive-data / Programming / Disconnected**. Commands move the card between states. Only certain commands are valid in each state. The host's job is to track this state and never issue an illegal command.

This is why a from-scratch MMC driver isn't tractable in a chapter, it's not the wire protocol (relatively simple), it's the state-machine management plus the voltage switching plus the tuning plus the CRC plus the error recovery.

66.6 The kernel's MMC subsystem, three layers

```
user-space: cat /dev/mmcblk1p1
     │ read() syscall
     ▼
┌──────────────────────────────────────────────────┐
│  block layer (Ch 55D)                            │
│  builds a struct bio → struct request            │
└──────────────────────────────────────────────────┘
     │ mmc_blk_request_fn (queue_rq callback)
     ▼
┌──────────────────────────────────────────────────┐
│  MMC block driver (drivers/mmc/core/block.c)     │
│  - converts bio offsets → MMC block addresses    │
│  - builds struct mmc_request                     │
└──────────────────────────────────────────────────┘
     │ mmc_wait_for_req(host, mrq)
     ▼
┌──────────────────────────────────────────────────┐
```

```
    │  MMC core (drivers/mmc/core/core.c)                │
    │  - state-machine tracking                          │
    │  - CMD/data sequencing                              │
    │  - retry / error recovery                          │
    └────────────────────────────────────────────────────┘
         │ host->ops->request(host, mrq)
         ▼
    ┌────────────────────────────────────────────────────┐
    │  Host driver (drivers/mmc/host/sdhci-esdhc-imx.c) │
    │  - programs hardware registers                     │
    │  - configures DMA / PIO                            │
    │  - waits for completion IRQ                        │
    │  - reads response register, returns to core        │
    └────────────────────────────────────────────────────┘
         │ MMIO + IRQ
         ▼
   uSDHC peripheral → physical bus → card
```

The `mmc_host_ops` contract

A host driver provides this struct of callbacks:

```
/* drivers/mmc/host/sdhci-esdhc-imx.c — simplified */
static const struct mmc_host_ops sdhci_esdhc_ops = {
    .request          = sdhci_request,         /* execute one command + opt. data */
    .set_ios          = sdhci_set_ios,         /* change clock, bus width, voltage */
    .get_cd           = sdhci_esdhc_get_cd,    /* card-present? */
    .get_ro           = sdhci_esdhc_get_ro,    /* write-protect? */
    .enable_sdio_irq  = sdhci_enable_sdio_irq, /* for SDIO devices */
    .execute_tuning   = sdhci_esdhc_executing_tuning, /* HS200 tuning */
    .start_signal_voltage_switch = sdhci_start_signal_voltage_switch, /* 3.3→1.8 V */
    /* ... */
};
```

That is the abstraction. The core asks the host driver to “execute this request” or “switch to bus width 8” without caring whether the controller is uSDHC, sdhci-pci, dw-mshc, or anything else. The host driver translates abstract requests into specific MMIO writes for its hardware.

> **MMIO:** memory-mapped I/O, where software accesses peripheral registers through normal load and store instructions.

Tracing a single 4-KB read

You run `dd if=/dev/mmcblk1p1 bs=4096 count=1 of=/tmp/x`. Here’s what happens:

1. **VFS** sends `read(fd, buf, 4096)` to the block-device file’s chardev.
2. **Block layer** receives a 4 KB bio at logical-block-address (LBA) = wherever in p1. Splits into `struct request` (one or more, depending on splitting policy).

3. **`mmc_blk_request_fn`** (in `block.c`) is called as the queue's `queue_rq`. It:
    - Calculates the **MMC block address** (LBA / 512, since MMC blocks are 512 B even when the FS uses larger).
    - Decides between **single-block (CMD17)** for 1 block or **multi-block (CMD18 / CMD23+CMD18)** for 2+ blocks. For 4 KB = 8 blocks: multi-block with CMD23 prefix.
    - Builds a **`struct mmc_request`** containing the command, data-direction, sg-list for the DMA destination, and a completion callback.

> **MCU bridge:** Think of DMA like the MCU DMA controller you used for UART or SPI, but with cache coherency, scatter-gather descriptors, and kernel ownership rules added. **DMA:** Direct Memory Access. Hardware moves data to or from memory without the CPU copying each byte.

1. **`mmc_wait_for_req(host, mrq)`** in the core: sends to host_ops->request, waits on the completion.
2. **`sdhci_request`** in the host driver:
    - Programs the eSDHC's command register (CMD18 opcode), argument register (start block), block-count register (8).
    - Sets up SDMA: scatter-gather list pointing into the user-space buffer's pinned pages.
    - Enables IRQs for "command complete" and "transfer complete."
    - Writes the "start" bit. Hardware now drives the bus.
3. **Hardware** drives CMD18 onto the CMD line. EMMC responds with R1. EMMC begins streaming 8 × 512 bytes on the 8-bit DAT bus at 200 MHz (HS200) into DDR via DMA.

> **DDR:** external DRAM that must be configured and trained before most software can run from it.

1. **IRQ "command complete"** fires after the CMD line transaction. Host reads R1 response register, captures status.

> **MCU bridge:** Think of an IRQ like an EXTI/NVIC interrupt path, except Linux splits the hard interrupt from deferred work and must share lines across drivers. **IRQ:** interrupt request, the signal path that tells the CPU or interrupt controller that hardware needs service.

1. **IRQ "transfer complete"** fires after the data phase finishes. Host driver calls `mmc_request_done(host, mrq)`.
2. **MMC core** wakes the waiter. `mmc_blk_request_fn` checks for errors, retries if needed, calls `blk_mq_end_request(req, status)`.
3. **Block layer** returns to user-space.

End-to-end at HS200, 4 KB takes about ~25 µs (10 µs of bus time, ~15 µs of kernel + IRQ overhead). For a properly-sized buffer (~64 KB or larger), the kernel overhead amortises and you approach raw bus bandwidth.

A host driver's skeleton

If you ever did need to write one (say, porting to a new SoC):

```
static int sdhci_xxx_probe(struct platform_device *pdev)
{
    struct sdhci_host *host;
    struct resource *iomem;

    /* SDHCI is the common framework — most SDHCI-compatible host drivers
       reuse the SDHCI core and only set quirks + IOMEM */
    host = sdhci_pltfm_init(pdev, &sdhci_xxx_pdata, sizeof(struct sdhci_xxx));
    if (IS_ERR(host)) return PTR_ERR(host);

    /* Get clocks, regulators, IOMEM */
    sdhci_get_of_property(pdev);

    /* Custom: i.MX has non-standard register layout for some bits */
    host->ops = &sdhci_xxx_ops;        /* override SDHCI defaults */
    host->quirks |= SDHCI_QUIRK_BROKEN_DMA;
    host->quirks2 |= SDHCI_QUIRK2_NO_1_8_V;
    host->mmc->caps |= MMC_CAP_NONREMOVABLE | MMC_CAP_8_BIT_DATA;

    return sdhci_add_host(host);
}
```

The **SDHCI framework** (`drivers/mmc/host/sdhci.c`) handles the standard cases. Vendor drivers like `sdhci-esdhc-imx.c` add quirks and override specific operations. That's why the i.MX driver is ~1500 lines (mostly quirks) rather than 5000+, the heavy lifting is in `sdhci.c`.

For a completely custom controller (not SDHCI-compatible), you'd implement `mmc_host_ops` from scratch, that's more work but the structure is the same.

66.7 EXT_CSD, the eMMC’s health report card

eMMCs maintain a 512-byte **Extended Card-Specific Data** register full of metadata. Read it from userspace:

```
[root@pa-mini:~]# mmc extcsd read /dev/mmcblk1
============================================
  Extended CSD rev 1.7 (MMC 5.0)
============================================

Card supported command sets [S_CMD_SET: 0x01]
HPI Features [HPI_FEATURE: 0x01]: implementation based on CMD13
Background operations support [BKOPS_SUPPORT: 0x01]
...

Device life time estimation type A [DEVICE_LIFE_TIME_EST_TYP_A: 0x01]
Device life time estimation type B [DEVICE_LIFE_TIME_EST_TYP_B: 0x01]
Pre EOL information [PRE_EOL_INFO: 0x01]

eMMC Life Time Estimation A: 0%–10% device life time used
eMMC Life Time Estimation B: 0%–10% device life time used
Pre EOL information: Normal
```

The three key fields:

- **`DEVICE_LIFE_TIME_EST_TYP_A`**: SLC-cell wear, 0–9 representing 10% bands (0 = 0–10% used).
- **`DEVICE_LIFE_TIME_EST_TYP_B`**: MLC-cell wear, same scale.
- **`PRE_EOL_INFO`**: 0x01 = Normal, 0x02 = Warning (80%+ wear), 0x03 = Urgent (replacement needed).

Production firmware should periodically read these, log via MQTT/syslog to a fleet management system. When you see PRE_EOL warning across many units of the same age, you’ve quantified your hardware lifetime, invaluable for warranty planning.

66.8 eMMC boot partitions

eMMCs typically have:

- **Boot partition 1** (typically 4 MB).
- **Boot partition 2** (typically 4 MB).
- **RPMB** (Replay-Protected Memory Block, ~4 MB).
- **User partition** (the bulk).

```
[root@pa-mini:~]# ls /dev/mmcblk1*
/dev/mmcblk1         /dev/mmcblk1p1
```

```
/dev/mmcblk1boot0    /dev/mmcblk1boot1
/dev/mmcblk1rpmb
```

`mmcblk1boot0` is the active boot partition, i.MX6ULL boots from it (with the right fuse setting) instead of looking at the main partition's MBR. Use it for U-Boot:

> **MCU bridge:** Think of U-Boot like a much larger boot stub plus debug monitor: it initializes hardware, loads the next image, and gives you commands before Linux starts. **U-Boot:** the bootloader that initializes enough hardware to load and start the Linux kernel.

```
# Force write to mmcblk1boot0
[root@pa-mini:~]# echo 0 > /sys/block/mmcblk1boot0/force_ro
[root@pa-mini:~]# dd if=u-boot.imx of=/dev/mmcblk1boot0
[root@pa-mini:~]# echo 1 > /sys/block/mmcblk1boot0/force_ro

# Select boot partition 1 as the active partition
[root@pa-mini:~]# mmc bootpart enable 1 1 /dev/mmcblk1
```

Why? Two boot partitions enable atomic boot-loader updates: write to boot0 while booting from boot1. On success, swap. Crash mid-update = boot1 still works.

66.9 RPMB, replay-protected secure storage

RPMB is a small (~4 MB) partition that requires HMAC authentication for every write. The eMMC controller verifies that a key (programmed once at factory) matches before allowing writes. Reads are authenticated too, you know the data hasn't been tampered with.

Use cases:

- Secure-boot rollback prevention (store "minimum allowed firmware version").
- DRM keys.
- Counters that need replay protection (anti-replay nonce).

Programming the RPMB key is one-shot, once written, that's it forever. **Don't experiment with RPMB on production boards.**

66.10 Performance test

```
[root@pa-mini:~]# dd if=/dev/zero of=/data/big.bin bs=1M count=512 conv=fsync
512+0 records in
512+0 records out
536870912 bytes (537 MB) copied, 4.5 s, 119 MB/s
```

```
[root@pa-mini:~]# fio --name=randwr --filename=/data/test.bin --rw=randwrite \
    --bs=4k --runtime=30 --time_based --ioengine=psync --iodepth=1 --size=100M
   write: IOPS=2500, BW=10MiB/s
```

HS200 eMMC: 100–150 MB/s sequential, ~2500 IOPS random 4k write. Compare to a budget SD card: 30 MB/s sequential, ~200 IOPS random.

66.11 Key Notes

- The MMC core manages card protocols, the host driver manages the controller, and the block layer exposes storage requests. A failure can occur at any of these boundaries.
- Signaling voltage, bus width, controller capability and board wiring limit usable speed modes. A mode supported by a card is not automatically supported by the MINI.
- User areas, boot partitions and RPMB have different access and protection rules. They are not interchangeable locations for an image.
- Managed flash hides internal erase and wear behavior. Health indicators and throughput measurements inform a design but do not guarantee lifetime or power-loss safety.

66.12 Pitfalls

- **`bus-width = <4>`** on a chip with 8 data lines wired. You get DS or HS speeds at best. HS200 needs 8-bit. Check schematic ↔ DT.
- **Missing `vqmmc-supply`** for HS200. Driver can't switch to 1.8 V signaling. Falls back to HS50. Look for "fall back" messages in dmesg.
- **`non-removable` on an SD slot.** The card-detect signal is ignored, so the system keeps trying to talk to the slot after the card is removed.
- **`cd-gpios` polarity wrong.** Empty slot reports as "card present" (or vice versa). `GPIO_ACTIVE_LOW` is typical for card-detect switches.
- **eMMC tuning fails.** HS200 requires per-card calibration ("tuning"). Some eMMCs require specific tuning patterns. Mainline supports this. If you see "tuning failed" in dmesg, the eMMC chip is buggy (not common but happens, usually fixable by `mmc-ddr-1_8v` instead of HS200).
- **fsync slow.** eMMC `fsync` does a real flush-to-flash, which takes 10 to 100 ms. If your application calls `fsync` after every write, throughput drops sharply. Batch your writes.
- **Write amplification.** Even with TRIM, eMMC's GC writes amplify your data ~2–5×. A 10 GB/day app actually writes 30 GB/day to flash. Plan lifetime accordingly.

- **`force_ro` on boot partitions**: Default is RO. You must clear it to write. Don't forget to re-arm.
- **Power-fail mid-erase.** eMMC's internal erase block is invisible. A power loss can corrupt a wider area than you actually wrote. Industrial eMMCs (Micron, KIOXIA) include PFAIL protection. Consumer parts do not.

66.13 Going deeper

- **`Documentation/mmc/`**: MMC subsystem documentation.
- **`drivers/mmc/core/core.c`**: MMC core. Look at `mmc_wait_for_req`, `mmc_attach_sd`, `mmc_attach_mmc` for the state-machine code.
- **`drivers/mmc/core/block.c`**: the block-driver glue. `mmc_blk_mq_issue_rq` is the per-request entry point.
- **`drivers/mmc/host/sdhci.c`**: SDHCI common framework.
- **`drivers/mmc/host/sdhci-esdhc-imx.c`**: i.MX uSDHC driver. ~1500 lines, mostly quirks over `sdhci.c`.
- **`Documentation/devicetree/bindings/mmc/`**: MMC bindings.
- **JEDEC eMMC 5.1 standard (JESD84-B51)**: the eMMC specification.
- **SD Physical Layer Specification**: at sdcard.org (free Simplified version, ~250 pages).
- **`mmc-utils`**: `mmc`, `mmc extcsd`, RPMB tools.
- **`fio`**: the storage benchmarking tool.

> **End of Group A, Storage (Ch 64–66).** You now have the three storage stacks covered: QSPI NOR for small boot, EEPROM for tiny metadata, eMMC/SD for bulk. Pick by capacity + speed + reliability requirements.

> Next chapter: **Chapter 67: Temperature / humidity / pressure sensors.** Group B opens with the environmental sensors trio, BME280, SHT3x, AHT20, and the IIO drivers that expose them. **IIO:** Industrial I/O, Linux's subsystem for sensors, ADCs, DACs, and buffered sampled data.

Chapter 67: Temperature / humidity / pressure

> **IRQ:** interrupt request, the signal path that tells the CPU or interrupt controller that hardware needs service. **MCU bridge:** Think of an IRQ like an EXTI/NVIC interrupt path, except Linux splits the hard interrupt from deferred work and must share lines across drivers.

> **What:** three I²C environmental sensors, dissected: **Bosch BME280** (T+H+P, the workhorse), **Sensirion SHT3x** (T+H, lab-grade accuracy), **ASAir AHT20** (T+H, cheap-and-good). For each: register map, the bytes on the wire, how the mainline IIO driver works, and, for BME280, the most complex, a from-scratch IIO driver implemented from the datasheet. **IIO:** Industrial I/O, Linux's subsystem for sensors, ADCs, DACs, and buffered sampled data.
>
> **Why:** environmental sensors are the most common I²C peripherals in IoT and HMI products. They're also the canonical sensors for understanding the IIO subsystem: a small driver, a clear data path, real compensation math that exposes why "raw" and "scale" are separate IIO attributes. Writing one from scratch teaches you both the chip and IIO at the same time.
>
> **Focus: calibration math turns raw ADC bins into engineering units**. The BME280 ships per-chip calibration coefficients in non-volatile memory. The driver reads them at probe and applies a published polynomial to each raw measurement. Understanding this, that the compensation formula lives in the *driver*, not the *chip*, is the key insight.

67.1 Sensor comparison

	Bosch BME280	Sensirion SHT3x-DIS	ASAir AHT20
Measures	T, H, P	T, H	T, H
Interface	I²C or SPI	I²C	I²C
I²C address	0x76 / 0x77	0x44 / 0x45	0x38

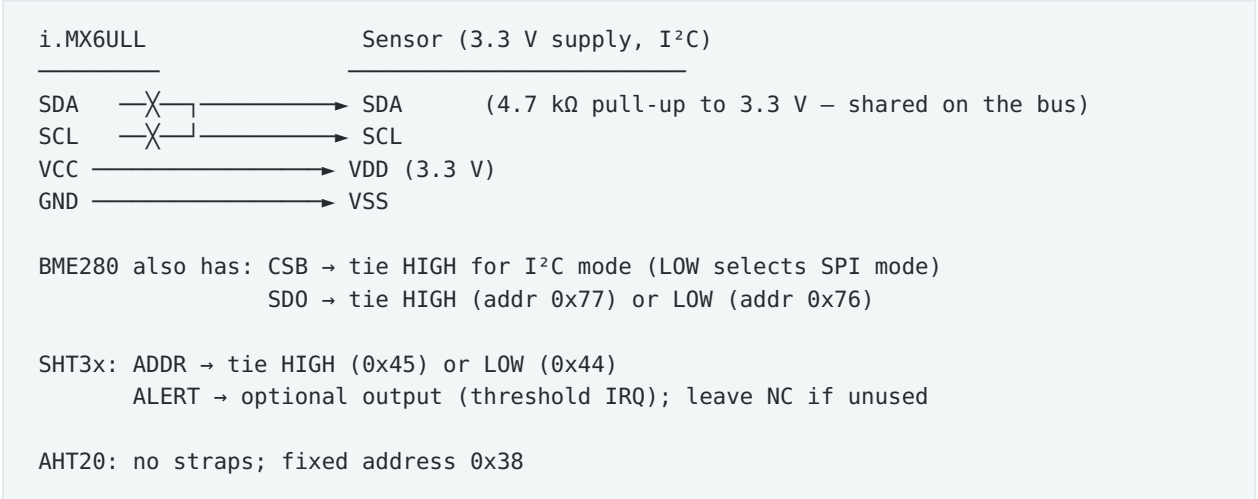
	Bosch BME280	Sensirion SHT3x-DIS	ASAir AHT20
Resolution (T)	0.01 °C	0.015 °C	0.01 °C
Accuracy (T)	±1.0 °C	±0.2 °C	±0.3 °C
Resolution (H)	0.008 %RH	0.01 %RH	0.024 %RH
Accuracy (H)	±3 %RH	±1.5 %RH	±2 %RH
Resolution (P)	0.18 Pa	,	,
Calibration data	per-chip NVM (read at startup)	factory-trimmed in firmware (no NVM read needed)	factory-trimmed in firmware
CRC on data	no	yes (8-bit polynomial)	no
Datasheet length	60 pp	30 pp	20 pp
Volume price	$2–4	$4–7	$0.40–0.80
Mainline driver	`bmp280-i2c.c` + `bmp280-core.c`	`sht3x.c` (hwmon) + `humidity/shtc1.c`-style IIO not yet for SHT3x in IIO; `sht3x` is hwmon	`aht20.c` (recent kernels)

Pick guide:

- **BME280**: when you want pressure (altitude approximation), or when you're already in the BMP/BME ecosystem.
- **SHT3x**: when you need humidity accuracy (HVAC, agriculture, calibration-reference work).
- **AHT20**: when cost matters and ±2 % RH is fine. The default cheap humidity sensor on hobbyist boards.

67.2 Schematic

All three are 4–6 pin packages:



The classic mistake: forgetting CSB strap on the BME280, left floating, the chip oscillates between I²C and SPI modes. Always tie CSB high explicitly.

67.3 Protocol, BME280 on the wire

BME280 has a register-bank model just like the EEPROMs in Ch 65:

Reg	Name	Purpose
0xD0	id	Chip ID (always 0x60 for BME280, 0x58 for BMP280)
0xE0	reset	Write 0xB6 to soft-reset
0xF2	ctrl_hum	Humidity oversampling
0xF4	ctrl_meas	Temp/pressure oversampling + power mode
0xF5	config	Filter, standby duration
0xF7..0xF9	press_msb/lsb/xlsb	Raw pressure (20-bit)
0xFA..0xFC	temp_msb/lsb/xlsb	Raw temperature (20-bit)

Reg	Name	Purpose
0xFD..0xFE	hum_msb/lsb	Raw humidity (16-bit)
0x88..0xA1	dig_T*, dig_P*, dig_H1	Calibration coefficients (factory NVM)
0xE1..0xE7	dig_H2..dig_H6	More calibration

To take a measurement:

1. Write to `ctrl_hum` (0xF2): humidity oversampling (1× / 2× / 4× / 8× / 16×).
2. Write to `ctrl_meas` (0xF4): temp + pressure oversampling, power mode.
   - Mode 0b01 = forced (take one measurement, return to sleep).
   - Mode 0b11 = normal (continuous).
3. Wait for measurement to complete (typically 8 ms with default oversampling).
4. Read 8 bytes from 0xF7, the entire measurement block in one transaction.
5. Apply compensation formulas using the calibration coefficients read at probe.

The compensation formulas (~30 lines of fixed-point arithmetic per axis) are *published in the datasheet* and identical for every BME280. They convert the raw 20-bit ADC values to °C × 100, Pa × 256, and %RH × 1024.

A worked read

To read once with default oversampling:

```
   START | 0xEC | 0xF4 | 0x25 | STOP        (write 0x25 to ctrl_meas: 1×T, 1×P, forced
mode)
   ... wait ~8 ms ...
   START | 0xEC | 0xF7 | START | 0xED |     (write reg address 0xF7, repeated start,
read)
        D[0..7] (msb,lsb,xlsb of P; msb,lsb,xlsb of T; msb,lsb of H)
   STOP
```

`0xEC` = (0x76 << 1) | 0 (write). `0xED` = (0x76 << 1) | 1 (read).

The output bytes pack as:

```
press_raw = (D[0] << 12) | (D[1] << 4) | (D[2] >> 4);   // 20-bit
temp_raw  = (D[3] << 12) | (D[4] << 4) | (D[5] >> 4);   // 20-bit
hum_raw   = (D[6] << 8) | D[7];                          // 16-bit
```

These three integers go into the compensation functions (datasheet §4.2.3 / 8.1 / 8.2).

67.4 How the mainline driver works

Source: `drivers/iio/pressure/bmp280-core.c` (~1500 lines) + `bmp280-i2c.c` (~150 lines) + `bmp280-spi.c`.

The driver covers BMP180 / BMP280 / BME280 / BMP380 / BMP580, five chips of similar lineage. The *bus* (I²C vs SPI) is decoupled via regmap. The *chip* is identified at probe via the `0xD0` ID register, and the right `chip_info` table is selected.

> **MCU bridge:** Think of regmap like a typed wrapper around your read_reg() and write_reg() helpers, with caching, locking, and bus differences handled centrally. **regmap:** a kernel helper that wraps register reads and writes over I2C, SPI, or MMIO.

Architecture

```
/* drivers/iio/pressure/bmp280.h — simplified */
struct bmp280_chip_info {
    unsigned int   id_reg;          /* 0xD0 */
    const unsigned int *chip_id;
    const struct iio_chan_spec *channels;
    int num_channels;

    int (*read_calib)(struct bmp280_data *);
    int (*chip_config)(struct bmp280_data *);
    int (*read_press)(struct bmp280_data *, u32 *);
    int (*read_temp)(struct bmp280_data *, s32 *);
    int (*read_humid)(struct bmp280_data *, u32 *);
    ...
};
```

The `chip_info` is a vtable: for each chip in the family, the driver assigns the right callbacks. BME280 gets `bme280_read_humid` (with humidity). BMP280 (no humidity sensor) gets a stub.

Probe

```
static int bmp280_common_probe(struct device *dev, struct regmap *regmap,
                               const struct bmp280_chip_info *chip_info,
                               const char *name, int irq)
{
    struct bmp280_data *data;
```

```
    struct iio_dev *indio_dev;
    unsigned int chip_id;
    int err;

    indio_dev = devm_iio_device_alloc(dev, sizeof(*data));
    data = iio_priv(indio_dev);
    data->regmap = regmap;
    data->chip_info = chip_info;

    /* 1. Identify the chip */
    err = regmap_read(regmap, chip_info->id_reg, &chip_id);
    if (chip_id != chip_info->chip_id[0]) return -EINVAL;

    /* 2. Read the per-chip calibration coefficients into data->calib */
    err = chip_info->read_calib(data);

    /* 3. Configure default operating mode */
    err = chip_info->chip_config(data);

    /* 4. Register with IIO */
    indio_dev->name = name;
    indio_dev->info = &bmp280_info;
    indio_dev->modes = INDIO_DIRECT_MODE;
    indio_dev->channels = chip_info->channels;
    indio_dev->num_channels = chip_info->num_channels;
    return devm_iio_device_register(dev, indio_dev);
}
```

The I²C glue is tiny, just creates a regmap and calls the common probe:

```
/* drivers/iio/pressure/bmp280-i2c.c — simplified */
static int bmp280_i2c_probe(struct i2c_client *client)
{
    struct regmap *regmap = devm_regmap_init_i2c(client, &bmp280_regmap_config);
    const struct bmp280_chip_info *chip_info = ...;  /* from id_table */
    return bmp280_common_probe(&client->dev, regmap, chip_info, ...);
}
```

read_raw, the IIO callback

```
static int bmp280_read_raw(struct iio_dev *indio_dev,
                           struct iio_chan_spec const *chan,
                           int *val, int *val2, long mask)
{
    struct bmp280_data *data = iio_priv(indio_dev);
    int ret;

    mutex_lock(&data->lock);
    switch (mask) {
    case IIO_CHAN_INFO_PROCESSED:
        switch (chan->type) {
        case IIO_TEMP:
            ret = data->chip_info->read_temp(data, val);     /* returns mC */
            break;
        case IIO_PRESSURE:
            ret = data->chip_info->read_press(data, (u32 *)val);  /* Pa */
```

```
            break;
        case IIO_HUMIDITYRELATIVE:
            ret = data->chip_info->read_humid(data, (u32 *)val);   /* milli-percent */
            break;
        }
        if (ret == 0) ret = IIO_VAL_INT;
        break;
    }
    mutex_unlock(&data->lock);
    return ret;
}
```

When user-space reads `/sys/bus/iio/devices/iio:device0/in_temp_input`, the driver issues a "forced measurement," waits about 8 ms, reads the 8 raw bytes, applies the compensation formula, and returns `23420` (millidegrees Celsius).

The compensation formula (where the magic happens)

For temperature, from the BME280 datasheet §8.1:

```
static s32 bme280_compensate_temp(struct bmp280_data *data, s32 adc_T)
{
    s32 var1, var2;

    var1 = (((adc_T >> 3) - ((s32)data->t_fine_calib.dig_T1 << 1)) *
            (s32)data->t_fine_calib.dig_T2) >> 11;

    var2 = (((((adc_T >> 4) - (s32)data->t_fine_calib.dig_T1) *
              ((adc_T >> 4) - (s32)data->t_fine_calib.dig_T1)) >> 12) *
              (s32)data->t_fine_calib.dig_T3) >> 14;

    data->t_fine = var1 + var2;       /* saved for use in pressure compensation */
    return (data->t_fine * 5 + 128) >> 8;  /* temp in mC */
}
```

These formulas are not approximations. They are lifted byte-for-byte from page 25 of the BME280 datasheet. The driver carries it as-published, same code in every BME280 driver across all OSes.

`dig_T1`, `dig_T2`, `dig_T3` are the calibration coefficients read from chip NVM at probe. Each chip has slightly different ones (silicon process variation).

The pressure formula is longer (~30 lines) and uses 64-bit arithmetic to avoid overflow. Humidity formula uses the saved `t_fine` to compensate for temperature.

67.5 Writing a BME280 IIO driver from scratch

Goal: a working IIO driver that exposes temp + humidity + pressure as `in_temp_input` / `in_pressure_input` / `in_humidityrelative_input`. No regmap, no chip-family abstraction, just BME280, raw `i2c_smbus_*`, and the Bosch math.

`mybme280.c`:

```
#include <linux/init.h>
#include <linux/module.h>
#include <linux/i2c.h>
#include <linux/delay.h>
#include <linux/iio/iio.h>
#include <linux/of.h>

#define BME280_CHIP_ID         0x60
#define REG_ID                 0xD0
#define REG_RESET              0xE0
#define REG_CTRL_HUM           0xF2
#define REG_CTRL_MEAS          0xF4
#define REG_CONFIG             0xF5
#define REG_DATA_START         0xF7   /* 8 bytes: P, T, H */

/* Calibration registers */
#define REG_CALIB_TP           0x88   /* 24 bytes: T1..T3, P1..P9 */
#define REG_CALIB_H1           0xA1   /* 1 byte */
#define REG_CALIB_H2           0xE1   /* 7 bytes: H2..H6 */

struct mybme {
    struct i2c_client *client;
    struct mutex lock;

    /* Calibration coefficients (read once at probe) */
    u16 T1; s16 T2, T3;
    u16 P1; s16 P2, P3, P4, P5, P6, P7, P8, P9;
    u8  H1, H3;
    s16 H2, H4, H5;
    s8  H6;

    s32 t_fine;   /* set by compensate_temp, used by compensate_press/humid */
};

/* Read N bytes from a register */
static int mb_read_block(struct mybme *m, u8 reg, u8 *out, int n)
{
    int r = i2c_smbus_read_i2c_block_data(m->client, reg, n, out);
    if (r < 0) return r;
    if (r != n) return -EIO;
    return 0;
}

static int mb_read_calib(struct mybme *m)
{
    u8 buf[26];
    int err;

    /* Read T1..T3, P1..P9 (24 bytes) starting at 0x88 */
    err = mb_read_block(m, REG_CALIB_TP, buf, 24);
    if (err) return err;
    m->T1 = buf[0] | (buf[1] << 8);
    m->T2 = (s16)(buf[2] | (buf[3] << 8));
    m->T3 = (s16)(buf[4] | (buf[5] << 8));
    m->P1 = buf[6] | (buf[7] << 8);
    m->P2 = (s16)(buf[8] | (buf[9] << 8));
    m->P3 = (s16)(buf[10] | (buf[11] << 8));
    m->P4 = (s16)(buf[12] | (buf[13] << 8));
```

```
    m->P5 = (s16)(buf[14] | (buf[15] << 8));
    m->P6 = (s16)(buf[16] | (buf[17] << 8));
    m->P7 = (s16)(buf[18] | (buf[19] << 8));
    m->P8 = (s16)(buf[20] | (buf[21] << 8));
    m->P9 = (s16)(buf[22] | (buf[23] << 8));

    /* H1 at 0xA1 (1 byte) */
    err = i2c_smbus_read_byte_data(m->client, REG_CALIB_H1);
    if (err < 0) return err;
    m->H1 = err;

    /* H2..H6 at 0xE1 (7 bytes) — encoding is *weird* per datasheet */
    err = mb_read_block(m, REG_CALIB_H2, buf, 7);
    if (err) return err;
    m->H2 = (s16)(buf[0] | (buf[1] << 8));
    m->H3 = buf[2];
    m->H4 = (s16)(((s8)buf[3] << 4) | (buf[4] & 0x0F));
    m->H5 = (s16)(((s8)buf[5] << 4) | (buf[4] >> 4));
    m->H6 = (s8)buf[6];
    return 0;
}

/* ------ Compensation formulas (copy-pasted from Bosch BME280 datasheet §8.1–8.2) ------
*/

static s32 compensate_T(struct mybme *m, s32 adc_T)
{
    s32 var1, var2;
    var1 = ((((adc_T >> 3) - ((s32)m->T1 << 1))) * (s32)m->T2) >> 11;
    var2 = (((((adc_T >> 4) - (s32)m->T1) *
              ((adc_T >> 4) - (s32)m->T1)) >> 12) * (s32)m->T3) >> 14;
    m->t_fine = var1 + var2;
    return (m->t_fine * 5 + 128) >> 8;   /* mC */
}

static u32 compensate_P(struct mybme *m, s32 adc_P)
{
    s64 var1, var2, p;
    var1 = (s64)m->t_fine - 128000;
    var2 = var1 * var1 * (s64)m->P6;
    var2 = var2 + ((var1 * (s64)m->P5) << 17);
    var2 = var2 + (((s64)m->P4) << 35);
    var1 = ((var1 * var1 * (s64)m->P3) >> 8) + ((var1 * (s64)m->P2) << 12);
    var1 = (((((s64)1) << 47) + var1)) * ((s64)m->P1) >> 33;
    if (var1 == 0) return 0;
    p = 1048576 - adc_P;
    p = (((p << 31) - var2) * 3125) / var1;
    var1 = (((s64)m->P9) * (p >> 13) * (p >> 13)) >> 25;
    var2 = (((s64)m->P8) * p) >> 19;
    p = ((p + var1 + var2) >> 8) + (((s64)m->P7) << 4);
    return (u32)p;        /* Pa × 256 (driver convention) */
}

static u32 compensate_H(struct mybme *m, s32 adc_H)
{
    s32 v;
    v = m->t_fine - 76800;
    v = ((((adc_H << 14) - (((s32)m->H4) << 20) - (((s32)m->H5) * v)) + 16384) >> 15)
        * (((((((v * ((s32)m->H6)) >> 10) *
```

```
               (((v * ((s32)m->H3)) >> 11) + 32768)) >> 10) + 2097152) *
            ((s32)m->H2) + 8192) >> 14);
    v = v - (((((v >> 15) * (v >> 15)) >> 7) * ((s32)m->H1)) >> 4);
    if (v < 0) v = 0;
    if (v > 419430400) v = 419430400;
    return (u32)(v >> 12);   /* %RH × 1024 */
}

/* ------ Do a forced measurement, read all 8 bytes, compensate ------ */

static int mb_measure(struct mybme *m, s32 *t_mC, u32 *p_Pa_x256, u32 *h_xRH_x1024)
{
    u8 raw[8];
    s32 adc_T, adc_P, adc_H;
    int err;

    /* humidity oversampling = 1× */
    err = i2c_smbus_write_byte_data(m->client, REG_CTRL_HUM, 0x01);
    if (err) return err;

    /* temp + press oversampling = 1×; mode = forced (0b01) */
    err = i2c_smbus_write_byte_data(m->client, REG_CTRL_MEAS,
                                    (0x01 << 5) | (0x01 << 2) | 0x01);
    if (err) return err;

    msleep(10);   /* worst-case ~8 ms with 1× oversampling */

    err = mb_read_block(m, REG_DATA_START, raw, 8);
    if (err) return err;

    adc_P = (raw[0] << 12) | (raw[1] << 4) | (raw[2] >> 4);
    adc_T = (raw[3] << 12) | (raw[4] << 4) | (raw[5] >> 4);
    adc_H = (raw[6] << 8) | raw[7];

    *t_mC      = compensate_T(m, adc_T);   /* call first — sets t_fine */
    *p_Pa_x256 = compensate_P(m, adc_P);
    *h_xRH_x1024 = compensate_H(m, adc_H);
    return 0;
}

/* ------ IIO callback ------ */

static int mb_read_raw(struct iio_dev *idev, struct iio_chan_spec const *chan,
                       int *val, int *val2, long info)
{
    struct mybme *m = iio_priv(idev);
    s32 t_mC; u32 p_x256, h_x1024;
    int err;

    if (info != IIO_CHAN_INFO_PROCESSED) return -EINVAL;

    mutex_lock(&m->lock);
    err = mb_measure(m, &t_mC, &p_x256, &h_x1024);
    mutex_unlock(&m->lock);
    if (err) return err;

    switch (chan->type) {
    case IIO_TEMP:
        *val = t_mC;                        /* mC */
```

```
        return IIO_VAL_INT;
    case IIO_PRESSURE:
        *val = p_x256 / 256;                  /* Pa */
        return IIO_VAL_INT;
    case IIO_HUMIDITYRELATIVE:
        *val = h_x1024 * 1000 / 1024;         /* milli-percent */
        return IIO_VAL_INT;
    default:
        return -EINVAL;
    }
}

static const struct iio_chan_spec mb_channels[] = {
    { .type = IIO_TEMP,             .info_mask_separate = BIT(IIO_CHAN_INFO_PROCESSED) },
    { .type = IIO_PRESSURE,         .info_mask_separate = BIT(IIO_CHAN_INFO_PROCESSED) },
    { .type = IIO_HUMIDITYRELATIVE, .info_mask_separate = BIT(IIO_CHAN_INFO_PROCESSED) },
};

static const struct iio_info mb_iio_info = {
    .read_raw = mb_read_raw,
};

/* ------ Probe / Remove ------ */

static int mb_probe(struct i2c_client *client)
{
    struct iio_dev *idev;
    struct mybme *m;
    int chip_id, err;

    idev = devm_iio_device_alloc(&client->dev, sizeof(*m));
    if (!idev) return -ENOMEM;
    m = iio_priv(idev);
    m->client = client;
    mutex_init(&m->lock);

    /* Identify */
    chip_id = i2c_smbus_read_byte_data(client, REG_ID);
    if (chip_id < 0) return chip_id;
    if (chip_id != BME280_CHIP_ID)
        return dev_err_probe(&client->dev, -ENODEV,
                             "unexpected chip-id 0x%02x (want 0x60)\n", chip_id);

    /* Soft reset, wait for the chip to come back */
    i2c_smbus_write_byte_data(client, REG_RESET, 0xB6);
    msleep(5);

    err = mb_read_calib(m);
    if (err) return dev_err_probe(&client->dev, err, "read-calib failed\n");

    idev->name = "mybme280";
    idev->info = &mb_iio_info;
    idev->modes = INDIO_DIRECT_MODE;
    idev->channels = mb_channels;
    idev->num_channels = ARRAY_SIZE(mb_channels);

    return devm_iio_device_register(&client->dev, idev);
}
```

```
static const struct of_device_id mb_of_match[] = {
    { .compatible = "linuxlearn,mybme280" },
    { }
};
MODULE_DEVICE_TABLE(of, mb_of_match);

static const struct i2c_device_id mb_id[] = {
    { "mybme280", 0 },
    { }
};
MODULE_DEVICE_TABLE(i2c, mb_id);

static struct i2c_driver mb_driver = {
    .driver = {
        .name = "mybme280",
        .of_match_table = mb_of_match,
    },
    .probe    = mb_probe,
    .id_table = mb_id,
};
module_i2c_driver(mb_driver);

MODULE_LICENSE("GPL");
```

DT:

```
&i2c1 {
    bme280@76 {
        compatible = "linuxlearn,mybme280";
        reg = <0x76>;
    };
};
```

Example build and access:

```
[root@pa-mini:~]# insmod mybme280.ko
[root@pa-mini:~]# ls /sys/bus/iio/devices/iio:device0/
in_humidityrelative_input  in_pressure_input  in_temp_input  name  ...

[root@pa-mini:~]# cat /sys/bus/iio/devices/iio:device0/name
mybme280

[root@pa-mini:~]# cat /sys/bus/iio/devices/iio:device0/in_temp_input
23420
[root@pa-mini:~]# cat /sys/bus/iio/devices/iio:device0/in_pressure_input
100327
[root@pa-mini:~]# cat /sys/bus/iio/devices/iio:device0/in_humidityrelative_input
56000
```

Values: 23.42 °C (temp = 23420 mC), 100.3 kPa (= 100327 Pa = 1003.27 mbar = local barometric pressure at sea level), 56.0 % RH.

Driver is ~300 lines and reports calibrated values via IIO. The compensation math is the longest part, and that's *just the Bosch published formulas* copy-pasted from the datasheet.

What we *skipped* compared to mainline:

- Filter and standby-time configuration (we use minimums).
- Buffered/triggered capture (we only do `INDIO_DIRECT_MODE` sysfs reads).

> **sysfs:** a kernel-generated filesystem under /sys that exposes devices, drivers, and attributes.

- Power management (`runtime_suspend` to drop to chip sleep mode).
- Multi-chip support (we only handle BME280. Mainline handles BMP180/280/380/580 too).

Those are framework features, not chip-understanding features. The chip is what we set out to teach.

67.6 SHT3x, a different design philosophy

SHT3x doesn't have a register map. Instead it uses **2-byte commands** sent on I²C, optionally followed by a data block with **CRC**. Sensirion's design choice: keep state in the host, not the chip.

Commands

16-bit command	What
`0x2C 06`	One-shot measurement, high repeatability, with clock-stretching
`0x24 00`	One-shot measurement, high repeatability, no clock-stretching
`0x21 30`	Periodic mode, 1 mps, high repeatability
`0xE0 00`	Read measurement (after periodic mode)
`0x30 41`	Reset

16-bit command	What
0xF3 2D	Read status register

A one-shot read of T+H:

```
START | 0x88 | 0x2C | 0x06 | STOP       (write command, addr 0x44 << 1 = 0x88)
... wait ~15 ms ...
START | 0x89 | T_msb | T_lsb | T_crc | H_msb | H_lsb | H_crc | STOP
```

The CRC bytes use Sensirion's polynomial `0x31` (CRC-8) over each 2-byte word. The driver validates them. If either CRC is bad, retry.

Conversion math (much simpler than BME280)

```
T_mC  = -45000 + ((175000 * temp_raw) >> 16);
RH_milli_pct = (100000 * hum_raw) >> 16;
```

That's it. Two linear formulas. No per-chip calibration to read, Sensirion factory-trims each chip's coefficients into firmware before shipping, so every chip reports the same scale.

Mainline driver

`drivers/iio/humidity/sht3x.c` (~700 lines, but most is alert/threshold handling). The probe sends a reset command, sets up periodic mode, and registers an IIO device. The `read_raw` callback sends `0xE0 00`, parses 6 bytes (T,T,CRC,H,H,CRC), validates, returns the linear-mapped result.

To convert the from-scratch BME280 driver to SHT3x:

- Drop the calibration reads and compensation math.
- Replace register-pointer writes with 2-byte command writes.
- Add CRC-8 validation.
- Replace `mb_measure` with a "send command, sleep, read 6 bytes, CRC-check, linear-convert" function.

We won't re-implement it. The structure is now clear.

67.7 AHT20, even simpler

AHT20 is the cheap-and-cheerful option: ASAir's clone of the DHT-family with proper I²C.

Commands

Command	What
`0xBE 08 00`	Initialise (write once after power-up)
`0xAC 33 00`	Start measurement (trigger)
(no command)	Read 7 bytes, status, H[3], T[3]
`0xBA`	Soft reset

A one-shot read:

```
   START | 0x70 | 0xAC | 0x33 | 0x00 | STOP    (start measurement)
   ... wait ~80 ms ...
   START | 0x71 | S | H0 | H1 | H2/T0 | T1 | T2 | CRC | STOP
                   ↑
                   bit 7 of byte S is "busy" (1 = still measuring)
```

The H and T raw values are 20 bits each. They share a middle byte: the top 4 bits go to H, the bottom 4 bits to T.

Conversion:

```
RH_milli_pct = (h_raw * 100000) >> 20;
T_mC = ((t_raw * 200000) >> 20) - 50000;
```

Mainline driver: `drivers/iio/humidity/aht20.c` (in newer kernels). ~250 lines.

To convert the from-scratch driver: same structure as BME280, drop compensation, use AHT20's command sequence. After the BME280 driver, converting to AHT20 is mostly mechanical command-table substitution.

67.8 Now: the mainline driver

DT for the mainline BME280:

```
&i2c1 {
    bme280@76 {
        compatible = "bosch,bme280";
        reg = <0x76>;
        /* No other properties needed; the driver defaults are sensible */
```

```
    };
};
```

Enable `CONFIG_BMP280=y` and `CONFIG_BMP280_I2C=y` in kernel config. Boot:

```
[root@pa-mini:~]# ls /sys/bus/iio/devices/iio:device0/
in_humidityrelative_input
in_humidityrelative_oversampling_ratio
in_humidityrelative_oversampling_ratio_available
in_pressure_input
in_pressure_oversampling_ratio
in_temp_input
in_temp_oversampling_ratio
name
sampling_frequency
sampling_frequency_available

[root@pa-mini:~]# cat name
bme280
```

Notice extra knobs the mainline driver provides (oversampling ratio, sampling frequency), these tune the chip’s internal averaging.

```
[root@pa-mini:~]# echo 16 > in_temp_oversampling_ratio
[root@pa-mini:~]# echo 16 > in_humidityrelative_oversampling_ratio
```

Sets 16× oversampling: chip averages 16 samples internally, giving ~4× better noise at the cost of ~16× more measurement time (~130 ms instead of ~8 ms). Same data path, just longer averaging.

67.9 User-space access

For a logging app:

```
#!/usr/bin/env python3
import time
def read_iio(path):
    with open(path) as f: return int(f.read().strip())
while True:
    t = read_iio("/sys/bus/iio/devices/iio:device0/in_temp_input") / 1000.0
    h = read_iio("/sys/bus/iio/devices/iio:device0/in_humidityrelative_input") / 1000.0
    p = read_iio("/sys/bus/iio/devices/iio:device0/in_pressure_input")
    print(f"T={t:.2f}°C H={h:.1f}%RH P={p}Pa")
    time.sleep(10)
```

Pipe to MQTT, Grafana, SQLite, whatever the product needs.

67.10 Key Notes

- Raw sensor counts become useful measurements only through the correct scale or compensation path. BME280 calibration coefficients belong to the individual chip.
- Conversion completion, register read order and byte encoding are part of the measurement contract. A responding I2C address does not prove that a sample is fresh or valid.
- Oversampling and measurement cadence trade noise, response time and energy. Continuous sampling can also affect the temperature being observed.
- The driver should expose consistent units and validity, while application policy decides logging, trends and alarms. A demonstration reading is not a calibrated system-accuracy claim.

67.11 Pitfalls

- **CSB strap floating on BME280**: Chip oscillates between I²C and SPI modes. Tie CSB high explicitly (10 kΩ to VCC).
- **Wrong I²C address.** BME280: 0x76 if SDO low, 0x77 if SDO high. Check schematic.
- **Forgetting to set humidity oversampling** (`ctrl_hum` write). Default at power-on is 0x00 = humidity disabled. You read 0x8000 forever and wonder why.
- **`ctrl_hum` write order**: The chip only acts on `ctrl_hum` *after* the next `ctrl_meas` write. So always write `ctrl_hum` before `ctrl_meas`.
- **Self-heating.** Continuous-mode at 16× oversampling makes the chip's own current dissipation warm the sensor by ~0.5 °C. Use forced-mode + sleep, or accept the offset.
- **Calibration coefficient endianness.** They're little-endian on the wire. If you mis-cast `(s16)((buf[1] << 8) | buf[0])` vs the reverse, math is garbage. Cross-check against datasheet table.
- **CRC ignored on SHT3x**: CRC byte != 0 doesn't *break* but indicates data corruption, chip retried internally but the read got noisy. Log and retry. If persistent, bus signal-integrity issue.
- **AHT20's "busy" bit**: If status bit 7 is set, the chip is still measuring. You got the bytes too early. Re-trigger or wait longer.
- **Reading without the `t_fine` set first** (BME280). `compensate_P` and `compensate_H` use `t_fine` set by `compensate_T`. Call temp compensation first. Otherwise pressure & humidity are wrong.

- **All-`0x80 0x00` reads**: Forgot to write `ctrl_meas` with the forced-mode bit. Chip is asleep.

67.12 Going deeper

- **`drivers/iio/pressure/bmp280-core.c`**: the mainline driver in full. Compare every block to the from-scratch version.
- **`drivers/iio/pressure/bmp280-i2c.c`**: the tiny I²C glue.
- **`drivers/iio/humidity/sht3x.c`**: SHT3x.
- **`drivers/iio/humidity/aht20.c`**: AHT20 (newer kernels).
- **BME280 datasheet (Bosch BST-BME280-DS001)**: §4 register map. §8 compensation formulas. Read once, refer back forever.
- **SHT3x datasheet**: command list with CRC polynomial.
- **AHT20 datasheet**: short, English version sometimes hard to find. Chinese version on aosong.com has the bit-packing diagrams.
- **`Documentation/ABI/testing/sysfs-bus-iio*`**, IIO sysfs ABI. Tells you what `_processed` vs `_raw` + `_scale` mean.

> **ABI:** Application Binary Interface: the calling convention, register use, binary format, and library contract that let separately built code run together.

> Next chapter: **Chapter 68: Light & color sensors.** Photodiodes meet I²C. BH1750 / TSL2561 / VEML7700, three approaches to “convert photons to lux.”

Chapter 68: Light & color sensors

> **What:** four I²C ambient-light sensors, dissected: **Rohm BH1750** (the simplest), **AMS TSL2561** (dual-channel for IR rejection), **Vishay VEML7700** (low-power, modern). Plus the bonus **AMS TCS34725** (RGB+clear color sensor). For each: protocol on the wire, the mainline IIO driver internals, and a from-scratch IIO driver for BH1750. **IIO:** Industrial I/O, Linux's subsystem for sensors, ADCs, DACs, and buffered sampled data.
>
> **Why:** measuring light is harder than it looks. A photodiode's current is roughly proportional to incident photon flux, but the human eye's "lux" response is wavelength-weighted (CIE photopic curve). Different sensors solve this differently. BH1750 uses an analog filter. TSL2561 measures a broadband channel and an IR channel and subtracts. VEML7700 uses an integrated correction. After this chapter you can pick a sensor by its trade-offs, and write a driver for any of them.
>
> **Focus: integration time controls both noise floor and saturation point**. Light sensors are integrators, current × time → digital count. Long integration: low-light accuracy. Short integration: high-light range. Pick integration time for the range you care about. The IIO `integration_time` attribute exposes this directly.

68.1 Sensor comparison

	Rohm BH1750	AMS TSL2561	Vishay VEML7700	AMS TCS34725
Channels	1 (lux)	2 (broad + IR)	1 (corrected lux)	4 (R, G, B, clear)
Resolution	1 lx (low res) / 0.5 lx (high res)	16-bit ADC × 2	16-bit	16-bit × 4
Range	1–65535 lx	0.1–40000 lx (auto-gain)	0.05–120 klx	0.04–1000 klx (gain-dep)
I²C address	0x23 / 0x5C	0x29 / 0x39 / 0x49	0x10	0x29
Max bus clock	400 kHz	400 kHz	400 kHz	400 kHz

	Rohm BH1750	AMS TSL2561	Vishay VEML7700	AMS TCS34725
Integration time	120 ms (high-res) / 16 ms (low-res), fixed	13.7 / 101 / 402 ms	25 / 50 / 100 / 200 / 400 / 800 ms	2.4 ms × 1..256
Has IR rejection	analog filter (built-in)	computed (broad − IR)	analog filter	analog filter
Idle current	1 µA	0.6 µA	0.5 µA	0.5 µA
Volume price	$0.80–1.50	$1.50–2.50	$1.50–2.50	$2.50–4.00
Mainline driver	`bh1750.c`	`tsl2561.c` (in `staging`)	`veml7700.c`	`tcs3472.c`

Pick guide:

- **BH1750**: simplest protocol, fine accuracy. Default for “I just need lux.”
- **TSL2561**: when ambient has significant IR component (incandescent lighting, sunlight through glass).
- **VEML7700**: low-power product, or when you need wide dynamic range (klx + dark room).
- **TCS34725**: RGB color sensing (color matching, white-balance, paper-color detection).

68.2 Why lux is hard, the photometric response

Light meters claim “lux”, but a photodiode just produces current proportional to total photon flux. To convert photon flux to lux, you must weight by the eye’s wavelength sensitivity (peaks at 555 nm green, drops to near-zero at 400 nm violet and 700 nm red, dead-zero in IR/UV).

Three sensor strategies:

1. **Optical filter** (BH1750, VEML7700, TCS34725 clear channel): a colored glass cover on the die that approximates the photopic curve. Cheap, fixed.

2. **Multi-channel + math** (TSL2561): one broadband channel (visible + IR) + one IR-only channel. Compute lux = `(broad − IR) × calibrated_curve`. More accurate, more software.
3. **R+G+B sensors** (TCS34725): measure each band. Can compute lux *and* report color.

The mainline driver does whichever math the chip needs, and presents user-space with `in_illuminance_input` in lux. The user never sees the wavelength weighting. The driver does it. Just `cat in_illuminance_input` gives lux.

68.3 Protocol, BH1750 on the wire

BH1750 is unusual: it has no register map. You send single-byte **opcodes**, and the chip either acts or starts returning data.

Opcode	Meaning
0x00	Power down
0x01	Power on
0x07	Reset (clears data register)
0x10	Continuous, high-res mode (1 lx, 120 ms)
0x11	Continuous, high-res mode 2 (0.5 lx, 120 ms)
0x13	Continuous, low-res mode (4 lx, 16 ms)
0x20	One-shot, high-res mode
0x21	One-shot, high-res mode 2
0x23	One-shot, low-res mode
0x4N	Change sensitivity (N = MTREG / 8; trim integration time)

A typical sequence:

```
   START | 0x46 | 0x10 | STOP     (write opcode 0x10 — high-res continuous)
   ... wait 120 ms (per datasheet) ...
   START | 0x47 | D[hi] | D[lo] | STOP  (read 2 bytes — 16-bit count)
```

(`0x46` = 0x23<<1, write. `0x47` = 0x23<<1 | 1, read.)

To convert the 16-bit count to lux:

```
lux = count / 1.2   (datasheet typo-prone; check §"How to Calculate lx")
```

For the high-resolution mode 2 (0x11): `lux = count / (1.2 * 2)`.

Two bytes on the wire and one division, that is the whole protocol. It fits on a page.

68.4 How the mainline `bh1750` driver works

Source: `drivers/iio/light/bh1750.c` (~250 lines).

```
/* Simplified */
struct bh1750_data {
    struct i2c_client *client;
    struct mutex lock;
    int mtreg;             /* sensitivity trim, default 69 */
    const struct bh1750_chip_info *chip_info;
};

struct bh1750_chip_info {
    u16 mtreg_min, mtreg_max, mtreg_default;
    int inc_per_us_lo, inc_per_us_hi;   /* timing constants */
};

static int bh1750_change_mtreg(struct bh1750_data *data, int mtreg)
{
    int err;
    /* MTREG is split across two registers (3 high bits + 5 low bits) */
    err = i2c_smbus_write_byte(data->client, BH1750_MTREG_HI | (mtreg >> 5));
    if (err) return err;
    return i2c_smbus_write_byte(data->client, BH1750_MTREG_LO | (mtreg & 0x1F));
}

static int bh1750_read(struct bh1750_data *data, int *val)
{
    u8 buf[2];
    int err, delay;

    err = i2c_smbus_write_byte(data->client, BH1750_ONE_SHOT_H);
    if (err) return err;

    /* Wait for measurement: typical 120ms, scales with mtreg */
    delay = data->chip_info->inc_per_us_hi * data->mtreg;
    usleep_range(delay, delay + 5000);
```

```
    err = i2c_master_recv(data->client, buf, 2);
    if (err < 0) return err;

    *val = (buf[0] << 8) | buf[1];
    return 0;
}

static int bh1750_read_raw(struct iio_dev *idev,
                           struct iio_chan_spec const *chan,
                           int *val, int *val2, long mask)
{
    struct bh1750_data *data = iio_priv(idev);
    int err;

    mutex_lock(&data->lock);
    switch (mask) {
    case IIO_CHAN_INFO_RAW:
        err = bh1750_read(data, val);
        if (err) goto out;
        err = IIO_VAL_INT;
        break;
    case IIO_CHAN_INFO_SCALE:
        /* Scale = mtreg-normalised count per lux */
        *val = 0;
        *val2 = data->chip_info->inc_per_us_hi * 1000 / data->mtreg;
        err = IIO_VAL_INT_PLUS_MICRO;
        break;
    }
out:
    mutex_unlock(&data->lock);
    return err;
}
```

User-space reads `in_illuminance_raw` (the raw 16-bit count) and `in_illuminance_scale` (the conversion factor). The actual lux = `raw * scale`.

Some mainline drivers add a third attribute `_processed` that returns the result already converted to lux × 1000, saving user-space the math. BH1750's driver opts for the raw+scale pattern (more flexible).

68.5 Writing a BH1750 driver from scratch

`mybh1750.c`:

```
#include <linux/init.h>
#include <linux/module.h>
#include <linux/i2c.h>
#include <linux/delay.h>
#include <linux/iio/iio.h>

#define OP_POWER_ON       0x01
#define OP_RESET          0x07
#define OP_ONESHOT_HIRES  0x20
#define OP_POWER_DOWN     0x00
```

```
struct mybh {
    struct i2c_client *client;
    struct mutex lock;
};

static int mb_measure(struct mybh *m, u16 *out_count)
{
    u8 buf[2];
    int err;

    err = i2c_smbus_write_byte(m->client, OP_ONESHOT_HIRES);
    if (err < 0) return err;

    msleep(180);   /* per datasheet: high-res = 120ms typ., 180 ms max */

    err = i2c_master_recv(m->client, buf, 2);
    if (err < 0) return err;
    if (err != 2) return -EIO;

    *out_count = (buf[0] << 8) | buf[1];
    return 0;
}

static int mb_read_raw(struct iio_dev *idev,
                       struct iio_chan_spec const *chan,
                       int *val, int *val2, long mask)
{
    struct mybh *m = iio_priv(idev);
    u16 count;
    int err;

    if (chan->type != IIO_LIGHT) return -EINVAL;

    switch (mask) {
    case IIO_CHAN_INFO_PROCESSED:
        mutex_lock(&m->lock);
        err = mb_measure(m, &count);
        mutex_unlock(&m->lock);
        if (err) return err;
        /* lux = count / 1.2 — report lux × 1000 */
        *val = ((u32)count * 1000) / 12 * 10;     /* count/1.2 × 1000 */
        return IIO_VAL_INT;
    case IIO_CHAN_INFO_SCALE:
        /* Raw is in 0.833 lx units; scale = 1/1.2 */
        *val = 0; *val2 = 833333;
        return IIO_VAL_INT_PLUS_MICRO;
    case IIO_CHAN_INFO_RAW:
        mutex_lock(&m->lock);
        err = mb_measure(m, &count);
        mutex_unlock(&m->lock);
        if (err) return err;
        *val = count;
        return IIO_VAL_INT;
    }
    return -EINVAL;
}

static const struct iio_chan_spec mb_channels[] = {
    {
```

```
        .type = IIO_LIGHT,
        .info_mask_separate = BIT(IIO_CHAN_INFO_PROCESSED)
                            | BIT(IIO_CHAN_INFO_RAW)
                            | BIT(IIO_CHAN_INFO_SCALE),
    },
};

static const struct iio_info mb_iio_info = {
    .read_raw = mb_read_raw,
};

static int mb_probe(struct i2c_client *client)
{
    struct iio_dev *idev;
    struct mybh *m;
    int err;

    idev = devm_iio_device_alloc(&client->dev, sizeof(*m));
    if (!idev) return -ENOMEM;
    m = iio_priv(idev);
    m->client = client;
    mutex_init(&m->lock);

    /* Power on, then sanity reset */
    err = i2c_smbus_write_byte(client, OP_POWER_ON);
    if (err < 0) return dev_err_probe(&client->dev, err, "power-on failed\n");
    err = i2c_smbus_write_byte(client, OP_RESET);
    if (err < 0) return dev_err_probe(&client->dev, err, "reset failed\n");
    msleep(10);

    idev->name = "mybh1750";
    idev->info = &mb_iio_info;
    idev->modes = INDIO_DIRECT_MODE;
    idev->channels = mb_channels;
    idev->num_channels = ARRAY_SIZE(mb_channels);

    return devm_iio_device_register(&client->dev, idev);
}

static void mb_remove(struct i2c_client *client)
{
    i2c_smbus_write_byte(client, OP_POWER_DOWN);
}

static const struct of_device_id mb_of_match[] = {
    { .compatible = "linuxlearn,mybh1750" },
    { }
};
MODULE_DEVICE_TABLE(of, mb_of_match);

static const struct i2c_device_id mb_id[] = {
    { "mybh1750", 0 },
    { }
};
MODULE_DEVICE_TABLE(i2c, mb_id);

static struct i2c_driver mb_driver = {
    .driver = {
        .name = "mybh1750",
```

```
        .of_match_table = mb_of_match,
    },
    .probe    = mb_probe,
    .remove   = mb_remove,
    .id_table = mb_id,
};
module_i2c_driver(mb_driver);

MODULE_LICENSE("GPL");
```

DT:

```
&i2c1 {
    bh1750@23 {
        compatible = "linuxlearn,mybh1750";
        reg = <0x23>;
    };
};
```

Build, load, test:

```
[root@pa-mini:~]# insmod mybh1750.ko
[root@pa-mini:~]# cat /sys/bus/iio/devices/iio:device0/in_illuminance_processed
410000        ← 410 lx (room with overhead lights)

# Cover the sensor:
[root@pa-mini:~]# cat /sys/bus/iio/devices/iio:device0/in_illuminance_processed
80           ← 0.08 lx (effectively dark)
```

Driver is ~120 lines. Bottom-up: opcode write, sleep, read 2 bytes, divide. The whole protocol fit in a paragraph in the datasheet.

68.6 TSL2561, two channels, software fusion

TSL2561 has two photodiodes:

- **Channel 0**: broadband (visible + IR).
- **Channel 1**: IR-only.

The chip’s claim to fame: subtract Ch1 from Ch0 to get the visible-only response. The math:

```
ratio = Ch1 / Ch0;

if (ratio < 0.5):   lux = 0.0304 * Ch0 - 0.062 * Ch0 * ratio^1.4
elif ratio < 0.61:  lux = 0.0224 * Ch0 - 0.031 * Ch1
elif ratio < 0.80:  lux = 0.0128 * Ch0 - 0.0153 * Ch1
elif ratio < 1.30:  lux = 0.00146 * Ch0 - 0.00112 * Ch1
else:               lux = 0
```

Piecewise polynomial. The coefficients come from chip characterisation across a range of light sources (incandescent, fluorescent, daylight), the chip vendor has done the empirical work.

Mainline driver: `drivers/staging/iio/light/tsl2x7x.c` covers TSL2561, TSL2563, TSL2x7x family. Reads both channels, applies the formula, exposes `in_illuminance_input`.

68.7 VEML7700, auto-ranging done right

VEML7700 has one channel with **6 integration times** (25 / 50 / 100 / 200 / 400 / 800 ms) and **4 gains** (1x, 2x, 1/4x, 1/8x). The driver can auto-range: start at low gain + short integration. If count saturates, reduce gain. If count is too low, increase integration time.

```
/* drivers/iio/light/veml7700.c — simplified auto-range */
static int veml7700_read_lux(struct veml7700_data *data, int *lux)
{
    u16 raw;
    int err = veml7700_read_als(data, &raw);

    while (raw < 100 && data->it_idx < ITERATIONS_MAX) {
        data->it_idx++;
        veml7700_set_integration_time(data);
        err = veml7700_read_als(data, &raw);
    }
    while (raw > 50000 && data->it_idx > 0) {
        data->it_idx--;
        veml7700_set_integration_time(data);
        err = veml7700_read_als(data, &raw);
    }
    *lux = raw * veml7700_lux_per_count(data);
    return 0;
}
```

Why this matters: with auto-ranging, the chip covers 0.05 lx (dark hallway) to 120 klx (direct sunlight), six decades, without saturating. Without auto-ranging, you'd have to pick a tradeoff at design time.

68.8 TCS34725, RGB + clear

The bonus chip: color sensing. Four channels, R, G, B, and clear (broadband, similar to TSL2561 Ch0). Each is a 16-bit count over a programmable integration time.

```
/* IIO channels for TCS34725 */
static const struct iio_chan_spec tcs3472_channels[] = {
    { .type = IIO_INTENSITY, .modified = 1, .channel2 = IIO_MOD_LIGHT_CLEAR,
      .info_mask_separate = BIT(IIO_CHAN_INFO_RAW) },
    { .type = IIO_INTENSITY, .modified = 1, .channel2 = IIO_MOD_LIGHT_RED,
      .info_mask_separate = BIT(IIO_CHAN_INFO_RAW) },
    { .type = IIO_INTENSITY, .modified = 1, .channel2 = IIO_MOD_LIGHT_GREEN,
      .info_mask_separate = BIT(IIO_CHAN_INFO_RAW) },
```

```
    { .type = IIO_INTENSITY, .modified = 1, .channel2 = IIO_MOD_LIGHT_BLUE,
      .info_mask_separate = BIT(IIO_CHAN_INFO_RAW) },
};
```

User-space sees `in_intensity_clear_raw`, `in_intensity_red_raw`, `_green_raw`, `_blue_raw`. To compute lux: use the clear channel + R/G/B-weighted formula in the datasheet. To compute color temperature (Kelvin): from R/G/B + clear, the McCamy approximation or the chromaticity-coords method.

Mainline driver: `drivers/iio/light/tcs3472.c`.

68.9 Now: enable the mainline drivers

DT for BH1750:

```
&i2c1 {
    bh1750@23 {
        compatible = "rohm,bh1750";
        reg = <0x23>;
    };
};
```

Kernel config: `CONFIG_BH1750=y`. Boot:

```
[root@pa-mini:~]# cat /sys/bus/iio/devices/iio:device0/name
bh1750
[root@pa-mini:~]# cat /sys/bus/iio/devices/iio:device0/in_illuminance_raw
493
[root@pa-mini:~]# cat /sys/bus/iio/devices/iio:device0/in_illuminance_scale
0.833333
# → lux = 493 × 0.833 = 410 lx
```

For TSL2561 / VEML7700 / TCS34725: substitute the appropriate compatible:

```
tsl2561@39 { compatible = "amstaos,tsl2561"; reg = <0x39>; };
veml7700@10 { compatible = "vishay,veml7700"; reg = <0x10>; };
tcs34725@29 { compatible = "amstaos,tcs34725"; reg = <0x29>; };
```

68.10 Key Notes

- Light sensors do not all observe the same spectral response. Lux conversion, color channels and infrared rejection are device-specific design choices.
- Gain and integration time trade sensitivity against saturation and response delay. Reading faster than conversion does not create new measurements.
- The enclosure and optical window become part of calibration. A bare-board comparison cannot establish the finished product’s response.

- An automatic backlight policy needs filtering and hysteresis above the measurement interface, rather than reacting to every raw count.

68.11 Pitfalls

- **BH1750 reading times wrong.** Datasheet says 120 ms typical, 180 ms max for high-res. Use 180 ms to be safe. Or check the busy-flag (chip will NACK reads during measurement).
- **TSL2561 saturated**: If Ch0 or Ch1 is 0xFFFF, the chip is saturated. Either drop the gain (HIGH → LOW) or reduce integration time. Auto-range or the user will report “lux = 0” in bright sun.
- **Sensor cover material.** Glass with strong IR-cut coating distorts readings. Use clear glass or a known-spec optical window.
- **Tinted enclosures.** Dark-tinted plastic over the sensor cuts visible light unevenly. Calibrate against a known reference *with the enclosure in place*.
- **Direct light vs reflected.** A sensor pointed at the sky reads sky brightness, not ambient. For “what’s the light on my desk?” point at the desk, or use a diffuser cover.
- **VEML7700 auto-range hysteresis.** Switching back and forth between integration times causes flicker in the reported lux. The mainline driver hysteresis suppresses this. Rolling your own, leave deadbands.
- **TCS34725 IR contamination.** Even with IR-rejection filter, sunlight’s high R-channel reading isn’t pure red, there’s IR leak. For color-match work, use indoor LED light.
- **Integration time ≠ sampling rate.** If you read every 100 ms but integration is 800 ms, you get the same value four times in a row. Match the cadence.

68.12 Going deeper

- **`drivers/iio/light/bh1750.c`**: the production driver.
- **`drivers/staging/iio/light/tsl2x7x.c`**: TSL2561/2563/2x7x family.
- **`drivers/iio/light/veml7700.c`**: auto-ranging.
- **`drivers/iio/light/tcs3472.c`**: RGB + clear.
- **BH1750 datasheet (Rohm)**: opcode table on p.5.
- **TSL2561 datasheet (AMS)**: lux formula on p.16.
- **VEML7700 datasheet (Vishay)**: page on gain/integration trade-offs.

- **TCS34725 datasheet (AMS)**: color-temperature derivation formula in app note AN1078.
- **CIE photopic luminosity function**: the eye-response curve all these sensors approximate.

> Next chapter: **Chapter 69: Air quality, gas, particulate matter.** Three classes of "what's in the air" sensor: NDIR CO_2 (SCD30), metal-oxide eCO_2/TVOC (CCS811), and laser-scattering PM (PMS5003). Different bus, different protocol, different physics, different accuracy stories.

Chapter 69: Air quality, gas, particulate matter

> **IRQ:** interrupt request, the signal path that tells the CPU or interrupt controller that hardware needs service. **MCU bridge:** Think of an IRQ like an EXTI/NVIC interrupt path, except Linux splits the hard interrupt from deferred work and must share lines across drivers.

> **What:** three radically different "what's in the air" sensors: **Sensirion SCD30** (NDIR CO_2, I^2C with clock-stretching), **AMS CCS811** (metal-oxide TVOC + eCO_2, I^2C with interrupt), **Plantower PMS5003** (laser-scattering PM, UART). Each represents a different sensing physics, a different bus, a different protocol shape. For each: physics, protocol, the mainline driver, plus a from-scratch UART-based PMS5003 driver (since it's the most pedagogically interesting and the existing IIO support is fragmented). **IIO:** Industrial I/O, Linux's subsystem for sensors, ADCs, DACs, and buffered sampled data.
>
> **Why:** the air-quality market is exploding (post-pandemic. IAQ in offices. Outdoor pollution dashboards). These three sensors together cover the dimensions that matter: CO_2 (occupancy, ventilation), VOCs (cleaning chemicals, paint, formaldehyde), particulate matter (combustion, dust, wildfire smoke). Knowing which chip claims to measure what, and what it *actually* measures, separates a useful product from a placebo.
>
> **Focus: NDIR measures physics directly. Metal-oxide infers from a correlation. Laser scatter counts particles. Three very different things wearing the label 'air quality sensor.'** NDIR (CO_2): direct absorbance measurement, traceable to the molecule. MOX (CCS811): a tin-oxide film whose resistance changes with reducing gases, the "eCO_2" is *inferred* from VOC trends, **not** actually measured. PM: literally counting particles flowing through a laser beam. Once you see this hierarchy, an "eCO_2" reading reads as "rough VOC trend that someone scaled into CO_2-looking numbers," not as a CO_2 measurement.

69.1 Sensor comparison

	Sensirion SCD30	AMS CCS811	Plantower PMS5003
Measures	CO_2 (NDIR), T, H	TVOC + eCO_2 (inferred)	PM1.0, PM2.5, PM10

	Sensirion SCD30	AMS CCS811	Plantower PMS5003
Physics	IR absorbance at 4.26 µm	Metal-oxide resistance	Laser scattering + counting
Interface	I²C (clock-stretch) or Modbus or UART	I²C with /WAKE pin	UART
Range	400–10000 ppm	TVOC 0–32k ppb, eCO_2 400–32k ppm	0–500 µg/m³
Accuracy	±(30 ppm + 3 %)	“indicative”, not certified	±10 µg/m³ + 10 %
Warm-up	< 5 s for first reading	20 min for stability; 48 h burn-in	30 s for stable fan
Idle current	19 mA average	0.6 mA average	100 mA (laser+fan)
Calibration	ASC (auto self-calibration) using 7-day low	factory; baseline drift	factory only
Volume price	$50–80	$5–15	$20–40
Mainline driver	`scd30_core.c` + `scd30_i2c.c`	`ccs811.c`	none in mainline; SerDev driver in some BSPs

Pick guide:

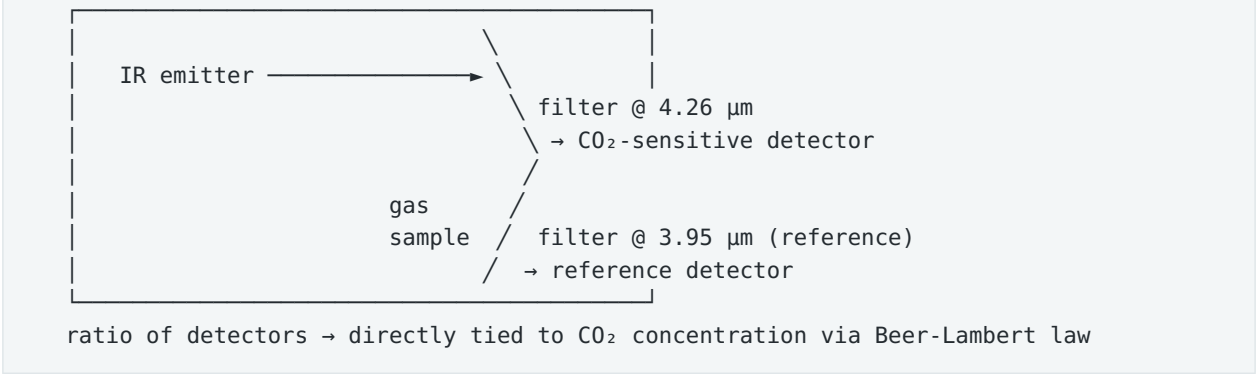
- **SCD30**: when you need **real CO_2**, traceable to a standard, for ventilation control or occupancy detection. Expensive but accurate.
- **CCS811**: when you want **a trend** in air quality (VOC build-up from cleaning chemicals, paint, body odour). Don’t label CCS811 output as “CO_2” to end users.
- **PMS5003**: when you need **PM monitoring**, air quality monitor product, wildfire-smoke alert, HVAC filter health.

You often combine all three. A complete IAQ (indoor air quality) sensor stack is: temp/humidity (Ch 67) + real CO_2 (SCD30) + VOC trend (CCS811) + PM (PMS5003).

69.2 Physics, what each sensor actually measures

SCD30, NDIR (Non-Dispersive InfraRed)

CO_2 absorbs strongly at 4.26 µm. The chip contains an IR emitter, a small chamber, and two photodetectors with bandpass filters:



The chamber is ~10 cm long. Gas diffuses in passively (no pump). The two-detector design rejects emitter intensity drift and dust. **The output is calibrated CO_2 ppm**, traceable to NIST CO_2 standards.

CCS811, MOX (Metal-OXide gas sensor)

A heated SnO_2 film. In dry air, oxygen ions adsorb on the film, raising its resistance. Reducing gases (CO, H_2, ethanol, formaldehyde, VOCs collectively) react with the surface oxygen, freeing electrons, lowering resistance.

```
resistance(VOC) = R₀ / (1 + k × [VOC])
```

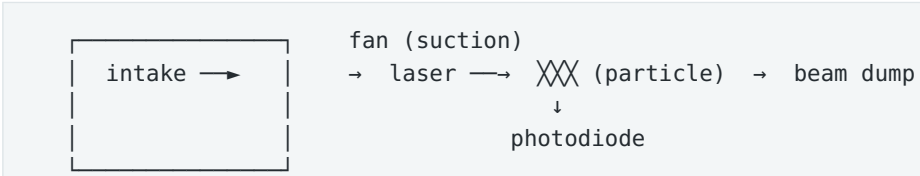
The chip's firmware maps resistance to two output numbers:

- **TVOC** (Total Volatile Organic Compounds, ppb).
- **eCO_2** (equivalent CO_2, ppm), *not* CO_2. It's an estimate based on the assumption that human-occupancy CO_2 rise tracks human-occupancy VOC rise. **Inaccurate by construction** whenever a VOC source other than human breath is present, cooking, cleaning, painting all corrupt it.

The film "burns in" over the first 48 hours (chemistry stabilises), drifts over months (poisoning), and ages over years.

PMS5003, Laser scattering

A small laser shines across a sample stream pulled through by a fan. Particles passing through scatter light. A photodiode at an angle catches the scatter. The chip's controller analyses pulse height (→ particle size) and pulse count rate (→ particle density). Output: counts per size bin, plus three "summary" estimates (PM1.0, PM2.5, PM10 µg/m³).



Cheap, accurate enough for trend, *not* certified-quality (which requires beta-attenuation or gravimetric monitors costing $20k+).

69.3 Protocol, SCD30 I²C with clock stretching

SCD30 uses a Sensirion-style command set: 2-byte commands with optional 2-byte payload + CRC-8.

Command	Payload	Purpose
`0x00 10`	2 bytes (0 = ambient pressure)	Trigger continuous measurement
`0x01 04`	none	Stop continuous
`0x46 00`	2 bytes (interval in seconds)	Set measurement interval
`0x02 02`	none, returns 2 bytes	Get data-ready status
`0x03 00`	none, returns 18 bytes	Read measurement
`0xD0 04`	none	Reset

Key gotcha: SCD30 uses **I²C clock stretching**, it holds SCL low for tens of milliseconds while doing internal work. Many SoC I²C controllers handle this fine. Some (notably Broadcom's BCM2835 on Raspberry Pi) do not. **i.MX6ULL handles clock-stretching correctly**, but verify your `clock-frequency` is ≤ 400 kHz.

A read of CO_2+T+H:

```
Host: START | 0xC2 | 0x03 | 0x00 | STOP                          (cmd 0x0300)
... wait 3 ms ...
Host: START | 0xC3 | (18 bytes) | STOP

18 bytes = CO₂_msb | CO₂_lsb | CRC | CO₂_xmsb | CO₂_xlsb | CRC |
           T_msb   | T_lsb   | CRC | T_xmsb   | T_xlsb   | CRC |
           H_msb   | H_lsb   | CRC | H_xmsb   | H_xlsb   | CRC
```

Each measurement is a **32-bit IEEE-754 float** split across 4 data bytes (with CRC after every 2). Decode:

```
uint32_t bits = (raw[0]<<24) | (raw[1]<<16) | (raw[3]<<8) | raw[4];
float co2_ppm = *(float*)&bits;
```

Indices 2 and 5 are CRC bytes, skipped here for the value extraction, but check them and retry on mismatch.

Mainline SCD30 driver

`drivers/iio/chemical/scd30_core.c` (~700 lines) + `scd30_i2c.c` (~150 lines).

```
/* Simplified */
static int scd30_command_read(struct scd30_state *state, enum scd30_cmd cmd,
                              u16 *out, int nresp)
{
    /* Send the 2-byte command */
    err = scd30_write(state, cmd, NULL, 0);
    /* Wait for chip's internal processing */
    msleep(state->cmd[cmd].timeout_ms);
    /* Read 2*nresp bytes (each result is 2 bytes + 1 CRC) */
    err = scd30_read_resp(state, out, nresp);
    /* Validate each CRC */
    return crc8_check(...);
}
```

The IIO `read_raw` callback issues `0x0300` (read measurement), parses 18 bytes into three floats, returns CO_2 in PPM, T in mC, H in mRH.

The CRC verification is what makes SCD30/SHT3x drivers heavier than BME280, every 2-byte word is CRC-checked, every read either succeeds with all-good CRCs or retries.

69.4 Protocol, CCS811 I²C with /WAKE pin

CCS811 has a register map (similar to BME280) but a quirk: the **/WAKE pin must be held low** when communicating. Internal sleep saves power. The host wakes the chip with /WAKE = low, transacts, then releases /WAKE.

Register	Purpose
0x00 STATUS	Bit 7 = APP_RUNNING, bit 3 = DATA_READY
0x01 MEAS_MODE	Drive mode (0 = idle, 1 = 1 reading/sec, 2 = 10 sec, 3 = 60 sec, 4 = 250 ms raw)

Register	Purpose
0x02 ALG_RESULT_DATA	8 bytes: eCO_2[2], TVOC[2], STATUS, ERROR_ID, RAW_DATA[2]
0x05 ENV_DATA	Set host's T/H (so chip can compensate baseline drift)
0xB0 SW_RESET	Write `0x11 0xE5 0x72 0x8A` to reset
0xF4 APP_START	"Boot from firmware"; mandatory at startup

Bring-up sequence:

1. Power on. Chip is in boot mode.
2. Wait ≥ 70 ms.
3. Issue `0xF4` (no data), switch to app mode.
4. Wait 70 ms.
5. Write `0x01 = 1` to MEAS_MODE.
6. From now on, the chip measures every 1 s. Poll STATUS for DATA_READY bit.
7. Read `0x02` (8 bytes) on data-ready. Bytes 0-1 = eCO_2 ppm, bytes 2-3 = TVOC ppb.

Mainline CCS811 driver

`drivers/iio/chemical/ccs811.c` (~600 lines).

```
static int ccs811_get_measurement(struct ccs811_data *data)
{
    int err, count = 0;
    u8 status;

    /* Wait for DATA_READY (poll up to 1s) */
    do {
        err = i2c_smbus_read_byte_data(client, CCS811_REG_STATUS);
        if (err < 0) return err;
        status = err;
        if (status & STATUS_DATA_READY) break;
        msleep(10);
    } while (count++ < 100);

    return i2c_smbus_read_i2c_block_data(client, CCS811_REG_ALG_RESULT,
                                         8, data->buffer);
```

```
}

static int ccs811_read_raw(struct iio_dev *idev, ...,  int *val, int *val2, long mask)
{
    struct ccs811_data *data = iio_priv(idev);
    err = ccs811_get_measurement(data);
    switch (chan->type) {
    case IIO_CONCENTRATION:
        if (chan->channel2 == IIO_MOD_CO2)
            *val = (data->buffer[0] << 8) | data->buffer[1];   /* eCO2 ppm */
        else
            *val = (data->buffer[2] << 8) | data->buffer[3];   /* TVOC ppb */
        return IIO_VAL_INT;
    }
}
```

User-space: `cat /sys/bus/iio/devices/iio:device0/in_concentration_co2_input` returns eCO_2 ppm.

The driver also implements an **environmental compensation** hook, feed it temperature and humidity from your BME280, and the chip adjusts its baseline. Without compensation, the eCO_2/TVOC drift over the seasons.

69.5 PMS5003, UART protocol

> **Driver choice:** Use the in-tree, maintained driver first. Use out-of-tree, spidev, or custom-driver paths only after you accept the kernel-version maintenance cost and document who owns updates.

PMS5003 isn't I^2C, it's a 9600-baud UART with a custom binary framing. No mainline IIO driver as of writing (a SerDev-based driver exists out-of-tree, but mainline support is fragmented). Perfect for a from-scratch implementation.

Frame format

Every frame is 32 bytes:

```
   offset  bytes  meaning
   0       2      0x42 0x4D     (sync: "BM")
   2       2      frame length (= 28 = 0x001C)
   4       2      PM1.0 standard (µg/m³)
   6       2      PM2.5 standard
   8       2      PM10 standard
   10      2      PM1.0 atmospheric
   12      2      PM2.5 atmospheric  ← the most often-quoted PM2.5
   14      2      PM10 atmospheric
   16      2      particle count > 0.3 µm in 0.1 L
   18      2      particle count > 0.5 µm
   20      2      particle count > 1.0 µm
   22      2      particle count > 2.5 µm
   24      2      particle count > 5.0 µm
```

```
   26      2      particle count > 10.0 µm
   28      2      reserved
   30      2      checksum (sum of bytes 0..29)
```

The chip auto-sends a frame every ~800 ms when active. So a driver doesn't *request* data. It listens for frames.

A from-scratch UART driver

We'll use the kernel's **SerDev** subsystem, `serdev_device_driver`, which lets a driver bind to a UART port via DT and receive raw bytes. The frame parsing is a small state machine.

`mypms5003.c`:

```
#include <linux/init.h>
#include <linux/module.h>
#include <linux/serdev.h>
#include <linux/iio/iio.h>
#include <linux/of.h>

#define PMS_FRAME_LEN 32

struct mypms {
    struct serdev_device *serdev;
    struct mutex lock;

    /* Latest values */
    u16 pm1_0, pm2_5, pm10;

    /* Frame-assembly state */
    u8  buf[PMS_FRAME_LEN];
    int idx;

    struct iio_dev *idev;
};

/* Called by serdev with raw RX bytes — runs in IRQ-thread context */
static int mp_receive_buf(struct serdev_device *serdev,
                          const unsigned char *data, size_t count)
{
    struct mypms *m = serdev_device_get_drvdata(serdev);
    size_t i;

    for (i = 0; i < count; i++) {
        u8 c = data[i];

        /* Sync state machine */
        if (m->idx == 0) {
            if (c == 0x42) { m->buf[m->idx++] = c; }
            continue;
        }
        if (m->idx == 1) {
            if (c != 0x4D) { m->idx = 0; continue; }   /* resync */
            m->buf[m->idx++] = c;
            continue;
```

```
        }
        m->buf[m->idx++] = c;
        if (m->idx < PMS_FRAME_LEN) continue;

        /* Full frame — validate checksum */
        u16 sum = 0;
        for (int j = 0; j < 30; j++) sum += m->buf[j];
        u16 frame_sum = (m->buf[30] << 8) | m->buf[31];
        if (sum != frame_sum) {
            m->idx = 0;
            continue;
        }

        /* Decode (big-endian 16-bit fields) */
        mutex_lock(&m->lock);
        m->pm1_0 = (m->buf[10] << 8) | m->buf[11];
        m->pm2_5 = (m->buf[12] << 8) | m->buf[13];
        m->pm10  = (m->buf[14] << 8) | m->buf[15];
        mutex_unlock(&m->lock);

        m->idx = 0;
    }
    return count;
}

static const struct serdev_device_ops mp_serdev_ops = {
    .receive_buf = mp_receive_buf,
};

/* IIO callback */
static int mp_read_raw(struct iio_dev *idev,
                       struct iio_chan_spec const *chan,
                       int *val, int *val2, long mask)
{
    struct mypms *m = iio_priv(idev);

    if (mask != IIO_CHAN_INFO_PROCESSED) return -EINVAL;
    if (chan->type != IIO_MASSCONCENTRATION) return -EINVAL;

    mutex_lock(&m->lock);
    switch (chan->channel2) {
    case IIO_MOD_PM1:   *val = m->pm1_0; break;
    case IIO_MOD_PM2P5: *val = m->pm2_5; break;
    case IIO_MOD_PM10:  *val = m->pm10;  break;
    default:            mutex_unlock(&m->lock); return -EINVAL;
    }
    mutex_unlock(&m->lock);
    return IIO_VAL_INT;
}

#define PMS_CHAN(idx, _mod) {                                   \
    .type = IIO_MASSCONCENTRATION,                               \
    .modified = 1,                                               \
    .channel2 = (_mod),                                          \
    .info_mask_separate = BIT(IIO_CHAN_INFO_PROCESSED),          \
    .scan_index = (idx),                                         \
}

static const struct iio_chan_spec mp_channels[] = {
```

```
    PMS_CHAN(0, IIO_MOD_PM1),
    PMS_CHAN(1, IIO_MOD_PM2P5),
    PMS_CHAN(2, IIO_MOD_PM10),
};

static const struct iio_info mp_iio_info = {
    .read_raw = mp_read_raw,
};

/* Probe / Remove */

static int mp_probe(struct serdev_device *serdev)
{
    struct iio_dev *idev;
    struct mypms *m;
    int err;

    idev = devm_iio_device_alloc(&serdev->dev, sizeof(*m));
    if (!idev) return -ENOMEM;
    m = iio_priv(idev);
    m->serdev = serdev;
    m->idev = idev;
    mutex_init(&m->lock);

    serdev_device_set_drvdata(serdev, m);
    serdev_device_set_client_ops(serdev, &mp_serdev_ops);

    err = serdev_device_open(serdev);
    if (err) return err;

    serdev_device_set_baudrate(serdev, 9600);
    serdev_device_set_flow_control(serdev, false);
    serdev_device_set_parity(serdev, SERDEV_PARITY_NONE);

    idev->name = "mypms5003";
    idev->info = &mp_iio_info;
    idev->modes = INDIO_DIRECT_MODE;
    idev->channels = mp_channels;
    idev->num_channels = ARRAY_SIZE(mp_channels);

    return devm_iio_device_register(&serdev->dev, idev);
}

static void mp_remove(struct serdev_device *serdev)
{
    serdev_device_close(serdev);
}

static const struct of_device_id mp_of_match[] = {
    { .compatible = "linuxlearn,mypms5003" },
    { }
};
MODULE_DEVICE_TABLE(of, mp_of_match);

static struct serdev_device_driver mp_driver = {
    .driver = {
        .name = "mypms5003",
        .of_match_table = mp_of_match,
    },
```

```
    .probe  = mp_probe,
    .remove = mp_remove,
};
module_serdev_device_driver(mp_driver);

MODULE_LICENSE("GPL");
```

DT, attach to UART2:

```
&uart2 {
    pinctrl-names = "default";
    pinctrl-0 = <&pinctrl_uart2>;
    status = "okay";

    pms5003 {
        compatible = "linuxlearn,mypms5003";
        current-speed = <9600>;
    };
};
```

Example build and access:

```
[root@pa-mini:~]# insmod mypms5003.ko
[root@pa-mini:~]# ls /sys/bus/iio/devices/iio:device0/
in_massconcentration_pm10_input
in_massconcentration_pm1_input
in_massconcentration_pm2p5_input
name

[root@pa-mini:~]# cat /sys/bus/iio/devices/iio:device0/in_massconcentration_pm2p5_input
12
[root@pa-mini:~]# # → 12 µg/m³ (clean indoor air)
```

In wildfire smoke: same code reads “180” or “220” µg/m³, the standard “very unhealthy” range.

Driver is ~150 lines including IIO. The whole protocol is a state machine over a UART stream.

Why SerDev?

SerDev, kernel/SerDev, is the “I want a UART-attached device, not a tty” subsystem. Before SerDev, drivers either polled a `/dev/ttyN` from user space (ugly), or set up a custom `line discipline` (painful kernel API). SerDev lets a kernel driver bind to a DT-described UART port. The framework handles the tty plumbing, exposes a small API (`set_baudrate`, `set_flow_control`, `receive_buf`), and the driver gets clean byte streams.

Many UART-protocol devices fit this mould: GPS NMEA receivers, Bluetooth HCI, LoRa modules, Nextion displays, PMS5003. SerDev makes them all tractable.

69.6 Putting it together, a full IAQ stack

For an indoor-air-quality monitor product:

```
&i2c1 {
    bme280@76 { compatible = "bosch,bme280"; reg = <0x76>; };
    scd30@61  { compatible = "sensirion,scd30"; reg = <0x61>; };
    ccs811@5a { compatible = "ams,ccs811"; reg = <0x5a>;
                wake-gpios = <&gpio4 14 GPIO_ACTIVE_LOW>; };
};

&uart2 {
    pms5003 { compatible = "plantower,pms7003"; };   /* or your linuxlearn one */
};
```

User-space reads all of them via IIO + serdev, feeds an MQTT topic, plots in Grafana. Bonus: feed the BME280's T+H to the CCS811's ENV_DATA register for compensation.

69.7 Key Notes

- NDIR CO2, metal-oxide gas response and optical particle sensing measure different quantities. An inferred eCO2 value must not be presented as a direct CO2 measurement.
- Framing, CRCs, readiness and wake control are part of obtaining valid samples. A plausible number alone does not establish that the transaction succeeded.
- Baseline algorithms and calibration depend on the intended environment. Their assumptions can fail in enclosed spaces or changing gas mixtures.
- Units, validity and sensor status should remain visible to the application. A combined air-quality display should not hide warmup, saturation or airflow failures.

69.8 Pitfalls

- **Calling eCO_2 "CO_2".** It isn't. CCS811's eCO_2 is calibrated to track human-occupancy *if and only if* there are no other VOC sources. Cooking, paint, cleaning products, smoking, all corrupt it. If your product advertises "CO_2 sensor," use SCD30, not CCS811.
- **CCS811 24-hour burn-in.** Brand-new chip reports nonsense for the first 20 minutes. Reasonable after 24 hours of continuous operation. Document this. Don't show users readings during burn-in.
- **SCD30 ASC corrupting calibration.** Auto self-calibration assumes your room reaches outdoor CO_2 (400 ppm) sometime each week, fails for hermetically sealed environments. Disable ASC via the FRC command if the room never opens up.

- **PMS5003 fan failure.** Particle count silently goes to zero or near-zero. Detect by: count is identically zero for > 10 frames in a row → likely fan failure. Most products bring out a “fan power” pin. Toggle it off and on to reset the fan.
- **PMS5003 in dust storms.** The chip’s saturation limit is ~500 µg/m³. In actual dust storms (Sahara, wildfire heart) values can hit 1500+. PMS5003 will report 500 forever.
- **SCD30 clock-stretching beyond timeout.** I²C controllers vary in their max-stretch tolerance. I.MX6ULL is fine at 400 kHz. Some bridges fail. Use scope to check SCL low durations if reads are flaky.
- **CCS811 /WAKE polarity.** Active low. Tying it permanently low works (no sleep), but increases power. For battery products, GPIO it.
- **PM2.5 standard vs atmospheric**: Two columns in the frame. The “atmospheric” one is calibrated against standard EPA reference. Use that one. The “standard” column is uncalibrated.
- **Mixing PM2.5 µg/m³ with AQI.** AQI is a country-specific transformation (US EPA different from China MEE). Don’t confuse the two.

69.9 Going deeper

- **`drivers/iio/chemical/scd30_core.c`** + **`scd30_i2c.c`**, the SCD30 mainline driver.
- **`drivers/iio/chemical/ccs811.c`**: CCS811.
- **`drivers/iio/chemical/sps30_i2c.c`**: Sensirion SPS30 PM driver (the higher-end alternative to PMS5003. Full IIO support).
- **SCD30 Interface Description (Sensirion)**: protocol reference. Command list with CRCs.
- **CCS811 datasheet (AMS)**: register map. Warm-up notes.
- **PMS5003 / PMS7003 manual** (search “PMS5003 transport protocol”), frame format and command list.
- **`Documentation/networking/serdev.rst`**: SerDev framework reference.
- **EPA AQI definition**: for converting µg/m³ to the human-friendly index.

> **End of Group B, Environmental sensors (Ch 67–69).** You now have temperature/humidity/pressure (Ch 67), light (Ch 68), and gas + PM (Ch 69), the three pillars of “what’s the environment doing?” Each with protocol, mainline driver internals, and at least one from-scratch implementation.

> Next chapter: **Chapter 70: I²C IMUs (MPU6050 / MPU9250 / ICM-20948).** From environmental into motion: 6- and 9-axis IMUs, the IIO trigger/buffer mechanism, and sensor-fusion math at 1 kHz.

Chapter 70: I²C IMUs

> **regmap:** a kernel helper that wraps register reads and writes over I2C, SPI, or MMIO. **MCU bridge:** Think of regmap like a typed wrapper around your read_reg() and write_reg() helpers, with caching, locking, and bus differences handled centrally.

> **What:** three I²C inertial measurement units, dissected: **InvenSense MPU6050** (6-axis, the classic), **MPU9250** (9-axis with an AK8963 magnetometer hiding inside via I²C-master mode), **ICM-20948** (modern 9-axis, replaced MPU9250). For each: register map, the sampling-rate trade-offs, the IIO **trigger + buffer** mechanism for high-rate capture, and a from-scratch MPU6050 driver including IIO buffer support. **IIO:** Industrial I/O, Linux's subsystem for sensors, ADCs, DACs, and buffered sampled data.
>
> **Why:** IMUs are everywhere, drones, e-scooters, VR headsets, fitness wearables, industrial vibration monitors. They're also the canonical IIO example of *high-rate buffered capture*: a 1 kHz IMU produces 6–9 measurements per sample, and one sysfs read per sample isn't going to work. The IIO trigger/buffer framework is the answer. Once you understand it, the same pattern works for any high-rate sensor. **sysfs:** a kernel-generated filesystem under /sys that exposes devices, drivers, and attributes.
>
> **Focus: Trigger + buffer is how IIO scales to thousands of samples per second.** A `trigger` (timer or IRQ) tells the driver "now". The driver atomically samples *all enabled channels*. Pushes the coordinated sample into a kfifo. User-space drains the kfifo from `/dev/iio:deviceN`. The whole pipeline is asynchronous and survives microsecond jitter. **MCU bridge:** Think of an IRQ like an EXTI/NVIC interrupt path, except Linux splits the hard interrupt from deferred work and must share lines across drivers. **IRQ:** interrupt request, the signal path that tells the CPU or interrupt controller that hardware needs service.

70.1 Chip comparison

	MPU6050	MPU9250	ICM-20948
Accel	3-axis, 16-bit, ±2/4/8/16 g	3-axis, 16-bit, ±2/4/8/16 g	3-axis, 16-bit, ±2/4/8/16 g

	MPU6050	MPU9250	ICM-20948
Gyro	3-axis, 16-bit, ±250/500/1k/2k °/s	3-axis, 16-bit, ±250/500/1k/2k °/s	3-axis, 16-bit, ±250/500/1k/2k °/s
Magnetometer	none	AK8963 internal (3-axis, ±4900 µT)	AK09916 internal (3-axis, ±4900 µT)
Max ODR	8 kHz (gyro), 1 kHz (accel)	8 kHz / 4 kHz	9 kHz / 4.5 kHz
FIFO	1 kB	1 kB	4 kB
I²C address	0x68 / 0x69	0x68 / 0x69	0x68 / 0x69
Max bus clock	400 kHz	400 kHz	400 kHz (Fast mode), 1 MHz (Fast+)
Onboard DMP	yes (closed-source firmware blob)	yes	yes (newer firmware)
Idle current	0.4 mA	3.5 mA (mag on)	1.2 mA
Lifecycle	EOL but ubiquitous on hobbyist boards	EOL since 2018	active
Volume price	$1.50–2.50	$3–6	$4–8
Mainline driver	`inv_mpu6050_*.c` family	same	same

Pick guide:

- **MPU6050**: cheap hobby projects, learning, where 6-axis is enough.
- **MPU9250**: legacy product maintenance, don’t design in new.
- **ICM-20948**: new designs needing magnetometer. Same I²C/SPI register model as MPU family (InvenSense legacy compatibility).

70.2 Why “9-axis” and what the magnetometer adds

A 6-axis IMU (accel + gyro) can compute **orientation drift-free in roll and pitch** by sensing gravity. But it has *no absolute reference for yaw*, rotate around the vertical axis and the gyro integration drifts seconds-of-arc per second.

Adding a magnetometer gives an Earth-field reference: the chip measures the geomagnetic vector (~50 µT, pointing roughly north + downward). Combined with the accel’s gravity vector, the fusion algorithm can lock down all three rotation angles, roll, pitch, *and* yaw. The result: **drift-free orientation in all three axes**.

For drones, AR/VR, robotic-arm control: 9-axis is mandatory. For tap-detection, fall-detection, vibration logging: 6-axis is enough.

70.3 Protocol, MPU6050 register map

> **Examples and production:** Do not burn fuses, enroll production keys, or sign release images while using these examples. Use throwaway keys and back up the unsigned image plus the key directory before testing irreversible security flows.

Register layout (most-used subset):

Reg	Name	Purpose
0x6B	PWR_MGMT_1	Reset, sleep, clock source select
0x6C	PWR_MGMT_2	Per-axis disable
0x19	SMPLRT_DIV	Sample rate = gyro_output / (1 + DIV)
0x1A	CONFIG	DLPF (digital low-pass filter) config
0x1B	GYRO_CONFIG	Full-scale range select
0x1C	ACCEL_CONFIG	Full-scale range select
0x38	INT_ENABLE	Interrupt mask

Reg	Name	Purpose
0x3A	INT_STATUS	Interrupt status (data-ready bit)
0x3B..0x40	ACCEL_OUT (X/Y/Z, 2 bytes each)	16-bit signed accel
0x41..0x42	TEMP_OUT	Die temperature
0x43..0x48	GYRO_OUT (X/Y/Z, 2 bytes each)	16-bit signed gyro
0x75	WHO_AM_I	Always 0x68
0x37	INT_PIN_CFG	Bypass enable (for MPU9250 magnetometer access)

To read a single 6-axis sample:

```
   Host: START | 0xD0 | 0x3B | START | 0xD1 | (14 bytes) | STOP
                              ↑
   (write reg pointer = 0x3B, repeated-start, read 14 bytes:
    AX_h AX_l AY_h AY_l AZ_h AZ_l TMP_h TMP_l GX_h GX_l GY_h GY_l GZ_h GZ_l)
```

Each value is 16-bit signed, big-endian. To convert raw to physical units:

```
/* Accel: ±2g full scale ⇒ 16384 LSB/g */
accel_g_x = (s16)((raw[0] << 8) | raw[1]) / 16384.0;

/* Gyro: ±250 °/s ⇒ 131 LSB / (°/s) */
gyro_dps_x = (s16)((raw[8] << 8) | raw[9]) / 131.0;

/* Temperature: see datasheet eq. */
temp_c = (s16)((raw[6] << 8) | raw[7]) / 340.0 + 36.53;
```

Bring-up sequence:

1. Read WHO_AM_I (0x75). Verify it returns 0x68 (or 0x71 for MPU9250, 0xEA for ICM-20948).
2. Write 0x80 to PWR_MGMT_1: soft reset.
3. Wait ~100 ms.
4. Write 0x00 to PWR_MGMT_1: wake from sleep, internal 8 MHz clock.

5. Write 0x01 to PWR_MGMT_1: wake, PLL with X-gyro reference (lower noise).

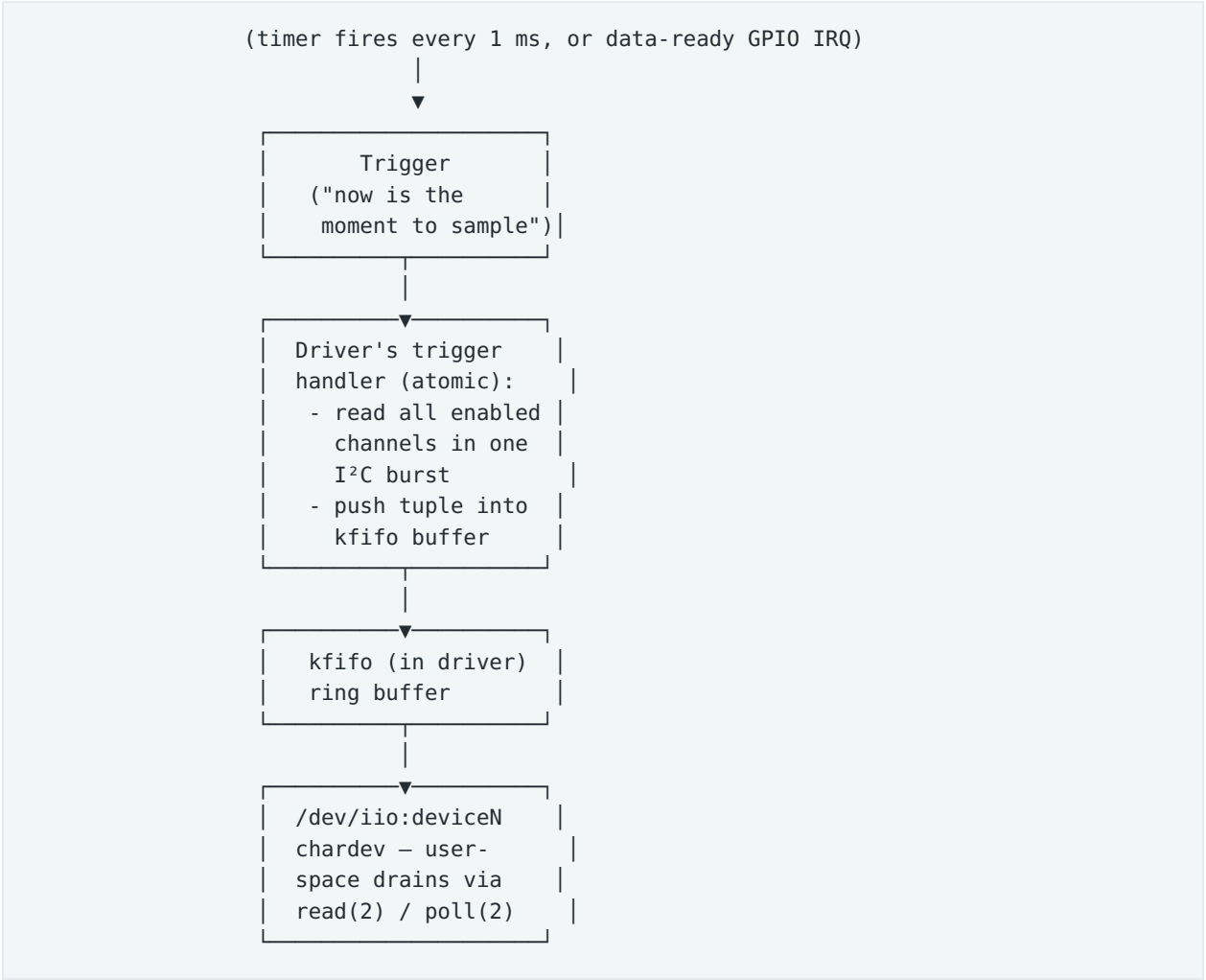
> **MCU bridge:** Think of a PLL like the clock multiplier setup you used on STM32, but with more clock roots, gates, and consumers that Linux later needs to describe.
> **PLL:** Phase-Locked Loop, a clock block that multiplies a reference clock to create faster clocks.

1. Configure DLPF, sample rate, ranges as needed.
2. Read at the configured cadence, or set up an IRQ on data-ready (bit 0 of INT_STATUS).

70.4 IIO trigger + buffer, the high-rate model

For a 1 kHz IMU, a one-sample-per-sysfs-read loop crosses the syscall boundary 1000 times per second per axis. That's ~30 µs per sysfs read × 6 axes × 1000 Hz = 18 % of one CPU just on the syscall overhead. Not workable.

The IIO solution: **triggers** + **buffers**.



A user-space app does:

```
# Enable channels to be captured
echo 1 > /sys/bus/iio/devices/iio:device0/scan_elements/in_accel_x_en
echo 1 > .../in_accel_y_en
echo 1 > .../in_accel_z_en
echo 1 > .../in_anglvel_x_en
echo 1 > .../in_anglvel_y_en
echo 1 > .../in_anglvel_z_en

# Configure buffer
echo 512 > /sys/bus/iio/devices/iio:device0/buffer/length

# Bind a trigger
echo "hrtimer-0" > .../trigger/current_trigger
echo 1000 > /sys/bus/iio/devices/trigger0/sampling_frequency

# Start streaming
echo 1 > .../buffer/enable

# Drain (each sample = 6 × 2 bytes = 12 bytes; 12000 B/s)
dd if=/dev/iio:device0 of=samples.bin bs=12 count=10000
```

10 000 atomic samples land in `samples.bin`. The data layout matches the order of `scan_elements/*_en` toggled on. Each sample's bytes are packed. User-space parses with the driver-declared `scan_type` (bits per sample, byte order).

Triggers

A *trigger* is its own IIO object. Two common kinds:

- **hrtimer**: kernel high-resolution timer firing at a programmable rate. Drift-free, suitable for steady sampling. Backed by `drivers/iio/trigger/iio-trig-hrtimer.c`.
- **interrupt**: an IRQ on a GPIO connected to the chip's INT pin (the chip asserts when it has data ready). Synchronized exactly to the chip's sample clock.

> **MCU bridge:** Think of Linux GPIO like the same pin set/reset block you used on STM32, but accessed through a kernel subsystem that owns numbering, direction, interrupts, and user-space exposure. **GPIO:** General-Purpose Input/Output, a pin controlled as a digital input, output, or interrupt source.

Drivers may also publish their *own* trigger ("data-ready trigger") that consumer code can bind. The MPU6050 driver does this, its INT pin's IRQ becomes an IIO trigger named `mpu6050-dev0`, and you can bind it to its own buffer or to *another* device's buffer (sync sampling across chips).

70.5 How the mainline `inv_mpu6050` driver is structured

Source: `drivers/iio/imu/inv_mpu6050/`, 5 files, ~3000 lines total. Covers MPU6050, MPU6500, MPU9150, MPU9250, ICM20608, ICM20602, ICM20690 (the entire InvenSense MPU family).

```
drivers/iio/imu/inv_mpu6050/
├── inv_mpu_core.c     ← chip-agnostic IIO logic
├── inv_mpu_ring.c     ← buffer/trigger callbacks
├── inv_mpu_trigger.c  ← data-ready trigger
├── inv_mpu_i2c.c      ← I²C bus glue (creates regmap)
├── inv_mpu_spi.c      ← SPI bus glue (creates regmap)
```

Probe walk

```
/* Simplified */
static int inv_mpu_i2c_probe(struct i2c_client *client)
{
    struct regmap *regmap = devm_regmap_init_i2c(client, &inv_mpu_regmap_config);
    return inv_mpu_core_probe(regmap, client->irq, name,
                              inv_mpu_i2c_aux_bus_setup, /* aux bus for mag */
                              chip_type);
}

/* inv_mpu_core_probe (~simplified): */
static int inv_mpu_core_probe(struct regmap *regmap, int irq, const char *name,
                              int (*setup_aux)(...), int chip_type)
{
    struct iio_dev *indio_dev;
    struct inv_mpu6050_state *st;

    indio_dev = devm_iio_device_alloc(...);
    st = iio_priv(indio_dev);
    st->map = regmap;

    /* Identify */
    err = regmap_read(regmap, INV_MPU6050_REG_WHO_AM_I, &whoami);
    if (whoami != expected) return -ENODEV;

    /* Initial chip config: reset, set clock source, defaults */
    err = inv_mpu6050_init_config(indio_dev);

    /* Optional: bring up auxiliary I²C bus to talk to a magnetometer */
    if (setup_aux) setup_aux(st);

    /* Set up trigger + buffer infrastructure */
    err = devm_iio_triggered_buffer_setup(...,
              iio_pollfunc_store_time,        /* pre-handler */
              inv_mpu6050_read_fifo,           /* main handler */
              &inv_mpu6050_buffer_setup_ops);

    /* Set up the data-ready trigger */
    err = inv_mpu6050_probe_trigger(indio_dev);

    /* Register */
```

```
    indio_dev->channels = inv_mpu_channels;
    indio_dev->num_channels = ARRAY_SIZE(inv_mpu_channels);
    indio_dev->info = &mpu_info;
    return devm_iio_device_register(...);
}
```

Channel definitions

```
#define INV_MPU6050_CHAN(_type, _channel2, _index) {  \
    .type = (_type),                                   \
    .modified = 1,                                     \
    .channel2 = (_channel2),                           \
    .info_mask_separate = BIT(IIO_CHAN_INFO_RAW),       \
    .info_mask_shared_by_type = BIT(IIO_CHAN_INFO_SCALE), \
    .scan_index = (_index),                             \
    .scan_type = {                                     \
        .sign = 's', .realbits = 16, .storagebits = 16,\
        .shift = 0, .endianness = IIO_BE,              \
    },                                                 \
}

static const struct iio_chan_spec inv_mpu_channels[] = {
    IIO_CHAN_SOFT_TIMESTAMP(0),
    {  /* gyro temp */
        .type = IIO_TEMP, .info_mask_separate = BIT(IIO_CHAN_INFO_RAW) | ...,
        .scan_index = -1,
    },
    INV_MPU6050_CHAN(IIO_ANGL_VEL, IIO_MOD_X, 1),
    INV_MPU6050_CHAN(IIO_ANGL_VEL, IIO_MOD_Y, 2),
    INV_MPU6050_CHAN(IIO_ANGL_VEL, IIO_MOD_Z, 3),
    INV_MPU6050_CHAN(IIO_ACCEL,    IIO_MOD_X, 4),
    INV_MPU6050_CHAN(IIO_ACCEL,    IIO_MOD_Y, 5),
    INV_MPU6050_CHAN(IIO_ACCEL,    IIO_MOD_Z, 6),
};
```

`scan_index` is the in-buffer position. `scan_type` tells user-space “16-bit signed, big-endian.” `IIO_CHAN_SOFT_TIMESTAMP(0)` adds a 64-bit timestamp per sample, essential when you need to time-align across sensors.

The trigger handler

```
static irqreturn_t inv_mpu6050_read_fifo(int irq, void *p)
{
    struct iio_poll_func *pf = p;
    struct iio_dev *indio_dev = pf->indio_dev;
    struct inv_mpu6050_state *st = iio_priv(indio_dev);
    size_t bytes_per_datum, fifo_count;

    mutex_lock(&st->lock);

    /* Read the FIFO count register to see how many samples are queued */
    regmap_bulk_read(st->map, INV_MPU6050_REG_FIFO_COUNT_H, &fifo_count, 2);
    fifo_count = be16_to_cpu(fifo_count);
    bytes_per_datum = compute_packet_size(st);
```

```
    while (fifo_count >= bytes_per_datum) {
        u8 sample_buf[INV_MPU6050_OUTPUT_DATA_SIZE];
        regmap_noinc_read(st->map, INV_MPU6050_REG_FIFO_R_W, sample_buf,
bytes_per_datum);

        iio_push_to_buffers_with_timestamp(indio_dev, sample_buf,
                                           iio_get_time_ns(indio_dev));
        fifo_count -= bytes_per_datum;
    }

    mutex_unlock(&st->lock);
    iio_trigger_notify_done(indio_dev->trig);
    return IRQ_HANDLED;
}
```

Each trigger event: read FIFO count, drain N samples from FIFO via a single bulk I²C read, push each into the IIO buffer with a kernel timestamp. The driver doesn't care whether the trigger came from a timer or from the chip's own data-ready IRQ, same handler either way.

Two-stage IRQ path for hardware trigger

When the data-ready trigger is bound:

1. Chip's INT pin goes high → GPIO IRQ on i.MX6ULL.
2. Kernel calls the IIO trigger's primary handler (the chip driver's "I have data" callback).
3. Primary handler returns `IRQ_WAKE_THREAD`. The trigger framework schedules the trigger handler (above) as a kernel thread.
4. Thread runs at SCHED_FIFO priority, drains FIFO, pushes to buffer.

At 1 kHz the CPU wakes once per sample for a few microseconds. Compare to `read()` polling, which is 30 µs per syscall and 6 channels = 180 µs per sample, or 18 % of one CPU. Trigger+buffer is closer to 5 µs per sample, around 0.5 %.

70.6 Writing an MPU6050 driver from scratch (with buffer support)

Goal: a working IIO driver that supports both sysfs reads (`in_accel_x_raw`) AND triggered buffered capture (via `/dev/iio:device0`). ~350 lines.

`mympu6050.c`:

```
#include <linux/init.h>
#include <linux/module.h>
#include <linux/i2c.h>
#include <linux/delay.h>
```

```
#include <linux/iio/iio.h>
#include <linux/iio/buffer.h>
#include <linux/iio/triggered_buffer.h>
#include <linux/iio/trigger_consumer.h>

#define REG_SMPLRT_DIV   0x19
#define REG_CONFIG       0x1A
#define REG_GYRO_CONFIG  0x1B
#define REG_ACCEL_CONFIG 0x1C
#define REG_INT_ENABLE   0x38
#define REG_INT_STATUS   0x3A
#define REG_ACCEL_XOUT_H 0x3B
#define REG_GYRO_XOUT_H  0x43
#define REG_PWR_MGMT_1   0x6B
#define REG_WHO_AM_I     0x75

#define WHO_AM_I_VAL     0x68

struct mympu {
    struct i2c_client *client;
    struct mutex lock;
};

/* === Low-level I²C ops === */

static int mp_read_block(struct mympu *m, u8 reg, u8 *buf, int n)
{
    int r = i2c_smbus_read_i2c_block_data(m->client, reg, n, buf);
    return r == n ? 0 : (r < 0 ? r : -EIO);
}

static int mp_write_byte(struct mympu *m, u8 reg, u8 val)
{
    return i2c_smbus_write_byte_data(m->client, reg, val);
}

/* === Bring-up === */

static int mp_init(struct mympu *m)
{
    int err;

    /* Reset */
    err = mp_write_byte(m, REG_PWR_MGMT_1, 0x80); if (err) return err;
    msleep(100);
    /* Wake, PLL with X-gyro ref */
    err = mp_write_byte(m, REG_PWR_MGMT_1, 0x01); if (err) return err;
    /* Sample rate = 1 kHz / (1+99) = 10 Hz initially (we'll override) */
    err = mp_write_byte(m, REG_SMPLRT_DIV, 99);   if (err) return err;
    /* DLPF: 44 Hz BW (CONFIG bits 2:0 = 3) */
    err = mp_write_byte(m, REG_CONFIG, 0x03);      if (err) return err;
    /* Gyro ±250 °/s */
    err = mp_write_byte(m, REG_GYRO_CONFIG, 0x00); if (err) return err;
    /* Accel ±2g */
    err = mp_write_byte(m, REG_ACCEL_CONFIG, 0x00); if (err) return err;

    return 0;
}
```

```
/* === Single-sample reads (for sysfs INFO_RAW) === */

static int mp_read_accel_axis(struct mympu *m, int axis, s16 *val)
{
    u8 buf[2];
    int err = mp_read_block(m, REG_ACCEL_XOUT_H + axis*2, buf, 2);
    if (err) return err;
    *val = (s16)((buf[0] << 8) | buf[1]);
    return 0;
}

static int mp_read_gyro_axis(struct mympu *m, int axis, s16 *val)
{
    u8 buf[2];
    int err = mp_read_block(m, REG_GYRO_XOUT_H + axis*2, buf, 2);
    if (err) return err;
    *val = (s16)((buf[0] << 8) | buf[1]);
    return 0;
}

/* === IIO read_raw === */

static int mp_read_raw(struct iio_dev *idev,
                       struct iio_chan_spec const *chan,
                       int *val, int *val2, long mask)
{
    struct mympu *m = iio_priv(idev);
    s16 raw;
    int err;

    switch (mask) {
    case IIO_CHAN_INFO_RAW:
        mutex_lock(&m->lock);
        if (chan->type == IIO_ACCEL)
            err = mp_read_accel_axis(m, chan->scan_index - 1, &raw);
        else if (chan->type == IIO_ANGL_VEL)
            err = mp_read_gyro_axis(m, chan->scan_index - 4, &raw);
        else err = -EINVAL;
        mutex_unlock(&m->lock);
        if (err) return err;
        *val = raw;
        return IIO_VAL_INT;
    case IIO_CHAN_INFO_SCALE:
        /* Accel: 1/16384 g/LSB = 9.80665 / 16384 m/s² per LSB */
        /* Gyro: 1/131 °/s/LSB = (π/180) / 131 rad/s per LSB */
        if (chan->type == IIO_ACCEL) {
            *val = 0; *val2 = 598;     /* 9.80665/16384 ≈ 0.000598 m/s²/LSB */
            return IIO_VAL_INT_PLUS_MICRO;
        } else {
            *val = 0; *val2 = 133;     /* π/180/131 ≈ 0.000133 rad/s/LSB */
            return IIO_VAL_INT_PLUS_MICRO;
        }
    }
    return -EINVAL;
}

/* === Channel table === */

#define ACCEL_CH(axis, idx) {                                       \
```

```
    .type = IIO_ACCEL, .modified = 1, .channel2 = (axis),           \
    .info_mask_separate = BIT(IIO_CHAN_INFO_RAW),                   \
    .info_mask_shared_by_type = BIT(IIO_CHAN_INFO_SCALE),           \
    .scan_index = (idx),                                            \
    .scan_type = { .sign='s', .realbits=16, .storagebits=16,        \
                   .endianness=IIO_BE },                             \
}
#define GYRO_CH(axis, idx) {                                       \
    .type = IIO_ANGL_VEL, .modified = 1, .channel2 = (axis),        \
    .info_mask_separate = BIT(IIO_CHAN_INFO_RAW),                   \
    .info_mask_shared_by_type = BIT(IIO_CHAN_INFO_SCALE),           \
    .scan_index = (idx),                                            \
    .scan_type = { .sign='s', .realbits=16, .storagebits=16,        \
                   .endianness=IIO_BE },                             \
}

static const struct iio_chan_spec mp_channels[] = {
    ACCEL_CH(IIO_MOD_X, 1),
    ACCEL_CH(IIO_MOD_Y, 2),
    ACCEL_CH(IIO_MOD_Z, 3),
    GYRO_CH (IIO_MOD_X, 4),
    GYRO_CH (IIO_MOD_Y, 5),
    GYRO_CH (IIO_MOD_Z, 6),
    IIO_CHAN_SOFT_TIMESTAMP(7),
};

/* === Buffered capture: trigger handler reads 14 bytes (accel+temp+gyro), pushes 12
bytes (no temp) === */

static irqreturn_t mp_trigger_handler(int irq, void *p)
{
    struct iio_poll_func *pf = p;
    struct iio_dev *idev = pf->indio_dev;
    struct mympu *m = iio_priv(idev);
    u8 raw[14];
    u8 sample[12 + 8];    /* 6 channels × 2 bytes + 64-bit timestamp */
    int err;

    mutex_lock(&m->lock);
    err = mp_read_block(m, REG_ACCEL_XOUT_H, raw, 14);
    mutex_unlock(&m->lock);

    if (err == 0) {
        /* Repack: accel 6 bytes, then gyro 6 bytes (skip temp at raw[6..7]) */
        memcpy(sample,     raw,        6);     /* accel X/Y/Z */
        memcpy(sample + 6, raw + 8,    6);     /* gyro X/Y/Z */
        iio_push_to_buffers_with_timestamp(idev, sample,
                                           iio_get_time_ns(idev));
    }

    iio_trigger_notify_done(idev->trig);
    return IRQ_HANDLED;
}

static const struct iio_info mp_iio_info = {
    .read_raw = mp_read_raw,
};

/* === Probe / Remove === */
```

```
static int mp_probe(struct i2c_client *client)
{
    struct iio_dev *idev;
    struct mympu *m;
    int who, err;

    idev = devm_iio_device_alloc(&client->dev, sizeof(*m));
    if (!idev) return -ENOMEM;
    m = iio_priv(idev);
    m->client = client;
    mutex_init(&m->lock);

    who = i2c_smbus_read_byte_data(client, REG_WHO_AM_I);
    if (who < 0) return who;
    if (who != WHO_AM_I_VAL)
        return dev_err_probe(&client->dev, -ENODEV,
                             "unexpected WHO_AM_I 0x%02x\n", who);

    err = mp_init(m);
    if (err) return err;

    idev->name     = "mympu6050";
    idev->info     = &mp_iio_info;
    idev->modes    = INDIO_DIRECT_MODE | INDIO_BUFFER_TRIGGERED;
    idev->channels = mp_channels;
    idev->num_channels = ARRAY_SIZE(mp_channels);

    err = devm_iio_triggered_buffer_setup(&client->dev, idev,
                                          iio_pollfunc_store_time,
                                          mp_trigger_handler,
                                          NULL);
    if (err) return err;

    return devm_iio_device_register(&client->dev, idev);
}

static const struct of_device_id mp_of_match[] = {
    { .compatible = "linuxlearn,mympu6050" },
    { }
};
MODULE_DEVICE_TABLE(of, mp_of_match);

static const struct i2c_device_id mp_id[] = {
    { "mympu6050", 0 },
    { }
};
MODULE_DEVICE_TABLE(i2c, mp_id);

static struct i2c_driver mp_driver = {
    .driver = {
        .name = "mympu6050",
        .of_match_table = mp_of_match,
    },
    .probe    = mp_probe,
    .id_table = mp_id,
};
module_i2c_driver(mp_driver);
```

```
MODULE_LICENSE("GPL");
```

DT:

```
&i2c1 {
    mpu6050@68 {
        compatible = "linuxlearn,mympu6050";
        reg = <0x68>;
    };
};
```

Test, sysfs first:

```
[root@pa-mini:~]# insmod mympu6050.ko
[root@pa-mini:~]# cat /sys/bus/iio/devices/iio:device0/in_accel_z_raw
16384         ← chip flat on the table, +1 g on Z
[root@pa-mini:~]# cat /sys/bus/iio/devices/iio:device0/in_accel_x_scale
0.000598
[root@pa-mini:~]# # 16384 × 0.000598 = 9.80 m/s², exactly 1 g. Good.
```

Then triggered buffered capture at 1 kHz using an hrtimer:

```
# Find a stock hrtimer trigger (some boards have one; if not, configfs-create one)
echo "hrtimer-0" > /sys/bus/iio/devices/iio:device0/trigger/current_trigger
echo 1000 > /sys/bus/iio/devices/trigger0/sampling_frequency

# Enable channels
for ax in x y z; do
    echo 1 > /sys/bus/iio/devices/iio:device0/scan_elements/in_accel_${ax}_en
    echo 1 > /sys/bus/iio/devices/iio:device0/scan_elements/in_anglvel_${ax}_en
done
echo 1 > /sys/bus/iio/devices/iio:device0/scan_elements/in_timestamp_en

# Buffer + start
echo 1024 > /sys/bus/iio/devices/iio:device0/buffer/length
echo 1 > /sys/bus/iio/devices/iio:device0/buffer/enable

# Drain 5000 samples (each = 6*2 + 8 = 20 bytes)
dd if=/dev/iio:device0 of=imu.bin bs=20 count=5000
```

Five thousand atomic samples, accel, gyro, and 64-bit timestamp, captured in 5 seconds. From here, user-space can FFT for vibration analysis or feed a Madgwick filter for orientation.

What we got, ~350 lines:

- Sysfs INFO_RAW per-axis reads.
- INFO_SCALE per axis-type for cooked-units conversion.
- Triggered buffered capture with timestamps.

- Drift-free protocol for 1 kHz capture.

What we *skipped* compared to mainline:

- Multiple range/ODR runtime configuration (we hardcoded ±2g / ±250 °/s / ~1 kHz).
- DMP firmware upload (closed-source, complex).
- Magnetometer aux-bus (MPU9250-only).
- Self-test, interrupt-on-motion, FIFO overflow detection.
- Runtime PM (chip sleep when idle).

70.7 MPU9250, the magnetometer-via-aux-bus quirk

MPU9250 contains an InvenSense MPU6500 (6-axis) **plus** an AsahiKASEI AK8963 magnetometer in the same package. The AK8963 isn't directly on the host I²C bus, it's accessible via the MPU6500's internal "auxiliary I²C master."

Two modes:

1. **Bypass mode**: bit 1 of INT_PIN_CFG (0x37) = 1. The MPU6500 ties its aux I²C lines directly to the host's I²C lines. Host then sees AK8963 at address 0x0C and talks to it directly.
2. **Aux-bus master mode**: the MPU6500 itself reads AK8963 registers periodically and stores results into its EXT_SENS_DATA registers (0x49..0x60). Host reads those.

Bypass is simpler. Aux-master mode is needed when the chip's internal sample-aligned-with-mag synchronisation matters.

The mainline driver supports both via the `inv_mpu_aux` helper, presenting `/sys/bus/iio/devices/iio:device0/in_magn_*_raw` even though physically the magnetometer is on a hidden bus.

70.8 ICM-20948, the modern replacement

ICM-20948 reorganised the register space into **banks**, 4 banks of 256 registers each, switched via a `REG_BANK_SEL` register. The bring-up sequence is 10 % longer (need bank-select before each access), but the chip itself has lower noise, lower idle current, and an updated DMP.

Mainline support: same `inv_mpu6050` driver, with `inv_icm20948_*` callbacks for bank handling.

70.9 Now: the mainline driver

DT for MPU6050:

```
&i2c1 {
    mpu6050@68 {
        compatible = "invensense,mpu6050";
        reg = <0x68>;
        interrupt-parent = <&gpio1>;
        interrupts = <14 IRQ_TYPE_EDGE_RISING>;   /* INT pin */
    };
};
```

Kernel config: `CONFIG_INV_MPU6050_IIO=y`, `CONFIG_INV_MPU6050_I2C=y`.

For MPU9250: `compatible = "invensense,mpu9250";`. For ICM-20948: `compatible = "invensense,icm20948";`.

The mainline driver gives you the same `/sys/bus/iio/...` interface plus extra knobs:

- `in_accel_sampling_frequency_available`: list of supported ODRs.
- `in_anglvel_scale_available`: list of supported ranges.
- `in_anglvel_calibbias_*`: write a bias offset.
- Data-ready trigger publishes as `mpu6050-dev0`, bindable as the sampling trigger (instead of hrtimer).

70.10 Sensor fusion in user-space

The IMU gives raw measurements. Orientation comes from a fusion algorithm. Most common: **Madgwick filter** (a complementary filter with quaternion gradient descent). User-space implementation in ~200 lines of C:

```
void madgwick_update(quat *q, vec3 gyro, vec3 accel, float dt, float beta)
{
    /* Normalize accel */
    accel = normalize(accel);

    /* Gradient descent step from accel: find quaternion that makes
       the predicted accel match the measured */
    vec4 grad = compute_gradient(q, accel);
    grad = normalize(grad);

    /* Quaternion derivative from gyro */
    quat qdot = quat_mul(*q, (quat){0, gyro.x, gyro.y, gyro.z}) * 0.5;

    /* Combine: gyro integration corrected by accel gradient, scaled by beta */
    qdot -= beta * grad;

    /* Integrate */
    *q = quat_add(*q, quat_scale(qdot, dt));
    *q = normalize(*q);
}
```

Run once per IMU sample (1 kHz). `beta` tunes the trust-gyro-vs-trust-accel balance (~0.1 is typical). Output: a quaternion describing the chip's orientation in world frame. Drift-free in roll/pitch.

For yaw, add magnetometer. The algorithm extends to "Madgwick AHRS."

Fusion belongs in user-space, not in the driver. The driver delivers clean samples. The application turns them into orientation.

70.11 Key Notes

- An IIO trigger and buffer decouple capture from user-space reads. Channel layout, scale and timestamps form the sample interface consumed by later processing.
- The configured output rate is not necessarily the rate of distinct samples delivered to the application. Bus time, FIFO behavior and buffer consumption constrain that path.
- Accelerometer, gyroscope and magnetometer data have different biases and coordinate frames. Sensor fusion needs those definitions in addition to a filter implementation.
- A magnetometer behind an IMU's auxiliary bus has a different ownership path from a directly attached I2C device. Bypass and internal-master modes cannot be treated as unrelated accesses.

70.12 Pitfalls

- **Wrong I²C address.** AD0 strap pin: low = 0x68, high = 0x69. Check schematic.
- **Forgetting the wake from sleep.** Default after reset is sleep mode (bit 6 of PWR_MGMT_1). Without writing PWR_MGMT_1 = 0, all reads return 0 or stale data.
- **Wrong byte order.** All IMU output is *big-endian* on the wire. If you misread as little-endian, accel-X and accel-Y appear swapped or scaled wrong.
- **Scale factor wrong.** Each range setting has a different LSB/g or LSB/(°/s). ±2g = 16384 LSB/g. ±4g = 8192. ±8g = 4096. ±16g = 2048. Off-by-2 = factor-of-2 wrong.
- **DLPF too high BW.** Default 256 Hz BW with ODR 1 kHz means you alias high-frequency noise into your signal. Set DLPF to 1/4 of ODR or lower.
- **Gyro bias drift not calibrated.** Gyros drift with temperature. Calibrate at startup (chip stationary for 5 seconds. Average ⇒ bias). Subtract bias from every sample.
- **Self-heating.** Continuous-mode chip rises 1-2 °C above ambient. If your application uses chip temperature as a thermometer (don't), this is an offset.

- **Magnetometer interference.** A buck regulator within 5 cm of MPU9250's mag corrupts readings. Schematic-stage planning matters.
- **Buffer overrun.** If user-space drains slower than the trigger rate, the IIO buffer overflows (older samples dropped). Check `/sys/bus/iio/devices/iio:device0/buffer/length` is big enough. Check `dmesg` for buffer-overrun warnings.

70.13 Going deeper

- **`drivers/iio/imu/inv_mpu6050/inv_mpu_core.c`**: the production driver. Compare to your from-scratch version.
- **`drivers/iio/imu/inv_mpu6050/inv_mpu_ring.c`**: buffer + trigger glue.
- **MPU6050 register map (InvenSense PS-MPU6000A-00 rev 3.4)**: the canonical reference.
- **MPU9250 product specification**: magnetometer-aux-bus details on page 50+.
- **ICM-20948 datasheet**: bank-selection model.
- **Sebastian Madgwick's thesis (2010)**: derivation of the AHRS algorithm. Math-heavy but illuminating.
- **`Documentation/iio/iio_configfs.rst`**: how to create a configfs hrtimer trigger if none exist by default.

> Next chapter: **Chapter 71: SPI IMUs.** When 1 kHz I²C isn't enough, LSM6DSO, ICM-42688, ADXL345. Bus contention, FIFO-watermark IRQs, and a from-scratch ADXL345 SPI driver.

Chapter 71: SPI IMUs

What: three SPI inertial sensors at increasing complexity: **Analog Devices ADXL345** (3-axis accel only, the textbook case), **STMicro LSM6DSO** (6-axis with internal FIFO and finite-state-machine), **InvenSense ICM-42688** (6-axis, low-noise, large FIFO). For each: SPI command framing (R/W bit + register address), FIFO+watermark IRQ patterns, and a from-scratch ADXL345 SPI driver with FIFO support. **MCU bridge:** Think of an IRQ like an EXTI/NVIC interrupt path, except Linux splits the hard interrupt from deferred work and must share lines across drivers. **IRQ:** interrupt request, the signal path that tells the CPU or interrupt controller that hardware needs service.

Why: beyond ~400 Hz per axis, you run out of I²C bandwidth: 400 kHz divided by ~10 bits per byte does not leave room for many channels. SPI runs at 10+ MHz, so an ICM-42688 streaming all 6 axes at 8 kHz fits comfortably. SPI also gives **per-CS configuration** (different IMUs on the same bus with different speeds and CPOL/CPHA), which makes multi-IMU systems straightforward to wire.

Focus: the FIFO + watermark IRQ pattern. Instead of taking one IRQ per sample (8000/s, far too many), configure the chip's internal FIFO with a watermark threshold. The chip raises its IRQ only when N samples have accumulated. The driver then drains them in a single SPI burst. The CPU wakes 100×/sec instead of 8000×/sec, while still capturing every sample.

71.1 Chip comparison

	ADXL345	LSM6DSO	ICM-42688
Axes	3 (accel)	6 (accel+gyro)	6 (accel+gyro)
Max ODR	3.2 kHz	6.66 kHz	8 kHz / 32 kHz (oversampled)
FIFO	32 samples	9 KB	2 KB
Noise floor (accel)	~290 µg/√Hz	70 µg/√Hz	60 µg/√Hz
Max SPI clock	5 MHz	10 MHz	24 MHz

	ADXL345	LSM6DSO	ICM-42688
Special features	tap, double-tap, activity	FSM (finite-state machines), MLC (machine-learning core)	UI + AUX (two SPI ports), anti-aliasing
Volume price	$4–7	$4–7	$5–9
Mainline driver	`adxl345_core.c` + `adxl345_spi.c`	`st_lsm6dsx_*`	`inv_icm42600_*`

Pick guide:

- **ADXL345**: cheap accel-only. Tap detection. Legacy.
- **LSM6DSO**: machine-learning core (FSM + MLC), useful for “detect a specific motion” without CPU involvement.
- **ICM-42688**: when noise floor matters (industrial vibration, audio-rate sampling).

For most new designs: **ICM-42688 if 6-axis SPI**, ADXL345 if 3-axis accel is enough.

71.2 Why SPI

	I²C @ 400 kHz	SPI @ 10 MHz
Bits per byte (overhead)	9 (start, 8 data, ACK)	8
Max effective throughput	~40 kB/s	~1.2 MB/s
6 axes × 2 bytes × N Hz	sustainable up to ~3 kHz	sustainable up to ~100 kHz
Per-CS config	shared bus settings	per-device mode/speed
Multi-drop	yes (addressed)	star (one CS per chip)

I²C breaks at high rates because the protocol overhead dominates. SPI doesn’t address (CS is implicit), doesn’t ACK each byte, and runs much faster. For a 1 kHz IMU, either bus works. For 8 kHz, SPI is mandatory.

71.3 ADXL345 SPI protocol

ADXL345 has a 64-byte register map. SPI command framing:

```
Host: /CS↓ | R/W | MB | addr[5:0] | data... | /CS↑
           ↑     ↑
           0=W   1 = multi-byte (auto-increment)
           1=R
```

A *read* of register 0x32 (DATAX0) for 6 bytes (X/Y/Z, 2 bytes each):

```
/CS↓
send: 0b11_110010   (R=1, MB=1, addr=0x32)
read: 6 bytes
/CS↑
```

A *write* of value 0x08 to register 0x2D (POWER_CTL):

```
/CS↓
send: 0b00_101101   (R=0, MB=0, addr=0x2D)
send: 0x08
/CS↑
```

The “MB” (multi-byte) flag tells the chip to auto-increment the register pointer between bytes, efficient way to dump consecutive registers.

Key registers:

Reg	Name	Purpose
0x00	DEVID	Always 0xE5
0x2C	BW_RATE	Output data rate + low-power mode
0x2D	POWER_CTL	Sleep / standby / measure mode
0x2E	INT_ENABLE	Interrupt sources
0x2F	INT_MAP	Route interrupts to INT1 or INT2 pin
0x30	INT_SOURCE	Interrupt status

Reg	Name	Purpose
0x31	DATA_FORMAT	Range, full-resolution, justification
0x32..0x37	DATAX0/X1/Y0/Y1/Z0/Z1	16-bit signed axis data
0x38	FIFO_CTL	FIFO mode + watermark
0x39	FIFO_STATUS	FIFO level

Bring-up:

1. Read DEVID (0x00). Verify 0xE5.
2. Write DATA_FORMAT (0x31) = 0x08 (full-res ±2g. +0x01 for ±4g, etc.).
3. Write BW_RATE (0x2C) = 0x0A (100 Hz default. See datasheet table for other rates).
4. Write FIFO_CTL (0x38) = (mode << 6) | (trigger << 5) | watermark.
    - Mode 1 = FIFO mode, 2 = Stream mode, 3 = Trigger mode.
    - Watermark = number of samples (0..31).
5. Write INT_ENABLE (0x2E) = 0x02 (watermark interrupt).
6. Write POWER_CTL (0x2D) = 0x08 (measure mode).

The chip now samples at 100 Hz. The FIFO accumulates samples. When level reaches watermark, INT1 asserts. Host drains, level resets, repeat.

71.4 How the mainline `adxl345` driver works

Source: `drivers/iio/accel/adxl345_core.c` (~600 lines) + `adxl345_spi.c` (~80 lines).

> **IIO:** Industrial I/O, Linux's subsystem for sensors, ADCs, DACs, and buffered sampled data.

```
/* drivers/iio/accel/adxl345_spi.c — simplified */
static int adxl345_spi_probe(struct spi_device *spi)
{
    struct regmap *regmap = devm_regmap_init_spi(spi, &adxl345_spi_regmap_config);
    if (IS_ERR(regmap))
```

```
        return PTR_ERR(regmap);
    return adxl345_core_probe(&spi->dev, regmap, /* fifo_delay_ns */ 0, /* name */);
}

/* The regmap config encodes the R/W + MB bit-stuffing automatically */
static const struct regmap_config adxl345_spi_regmap_config = {
    .reg_bits = 8,
    .val_bits = 8,
    .read_flag_mask  = 0x80 | 0x40,   /* R=1, MB=1 → OR'd into the address */
    .write_flag_mask = 0x00,
    .max_register    = 0x39,
};
```

The regmap layer takes care of OR’ing 0xC0 into addresses for reads. The core code just calls `regmap_read(regmap, reg, &val)`, same code that worked for I²C now works for SPI, courtesy of regmap.

> **MCU bridge:** Think of regmap like a typed wrapper around your read_reg() and write_reg() helpers, with caching, locking, and bus differences handled centrally.
> **regmap:** a kernel helper that wraps register reads and writes over I2C, SPI, or MMIO.

Probe flow

```
static int adxl345_core_probe(struct device *dev, struct regmap *regmap,
                              u32 fifo_delay_ns, const char *name)
{
    struct adxl345_data *data;
    struct iio_dev *indio_dev;
    unsigned int devid;

    indio_dev = devm_iio_device_alloc(dev, sizeof(*data));
    data = iio_priv(indio_dev);
    data->regmap = regmap;

    /* Verify DEVID */
    err = regmap_read(regmap, ADXL345_REG_DEVID, &devid);
    if (devid != ADXL345_DEVID) return -ENODEV;

    /* Configure: ±2g full-res, 100 Hz, FIFO stream mode */
    regmap_write(regmap, ADXL345_REG_DATA_FORMAT, ADXL345_FULL_RES | ADXL345_RANGE_2G);
    regmap_write(regmap, ADXL345_REG_BW_RATE, ADXL345_BW_100);
    regmap_write(regmap, ADXL345_REG_FIFO_CTL, ADXL345_FIFO_STREAM | 25);

    /* Enter measurement mode */
    regmap_update_bits(regmap, ADXL345_REG_POWER_CTL,
                       ADXL345_POWER_CTL_MEASURE, ADXL345_POWER_CTL_MEASURE);

    /* Register triggered buffer */
    err = devm_iio_triggered_buffer_setup(dev, indio_dev,
                                          iio_pollfunc_store_time,
                                          adxl345_trigger_handler,
                                          NULL);
```

```
    /* If there's an IRQ on the watermark line, set it up */
    if (irq > 0)
        err = devm_request_threaded_irq(dev, irq, NULL, adxl345_irq_handler,
                                        IRQF_TRIGGER_HIGH | IRQF_ONESHOT,
                                        name, indio_dev);

    return devm_iio_device_register(dev, indio_dev);
}
```

The IRQ + drain pattern

```
static irqreturn_t adxl345_irq_handler(int irq, void *p)
{
    struct iio_dev *indio_dev = p;
    struct adxl345_data *data = iio_priv(indio_dev);
    unsigned int int_source;
    unsigned int entries;
    int err;

    err = regmap_read(data->regmap, ADXL345_REG_INT_SOURCE, &int_source);
    if (err) return IRQ_NONE;

    if (int_source & ADXL345_INT_WATERMARK) {
        /* Read FIFO_STATUS for the count */
        regmap_read(data->regmap, ADXL345_REG_FIFO_STATUS, &entries);
        entries &= 0x3F;

        /* Drain N samples — each is 6 bytes */
        u8 buf[6 * 32];
        regmap_noinc_read(data->regmap, ADXL345_REG_DATAX0, buf, entries * 6);

        for (int i = 0; i < entries; i++)
            iio_push_to_buffers_with_timestamp(indio_dev, &buf[i * 6],
                                               iio_get_time_ns(indio_dev));
    }
    return IRQ_HANDLED;
}
```

Compare to the per-sample-IRQ alternative: at 800 Hz with watermark = 16, the IRQ fires 50× per second instead of 800× per second. CPU load drops 16-fold, and the captured data is the same.

71.5 Writing an ADXL345 SPI driver from scratch (with FIFO + watermark IRQ)

`myadxl345.c`:

```
#include <linux/init.h>
#include <linux/module.h>
#include <linux/spi/spi.h>
#include <linux/delay.h>
#include <linux/iio/iio.h>
```

```
#include <linux/iio/buffer.h>
#include <linux/iio/triggered_buffer.h>
#include <linux/iio/trigger_consumer.h>
#include <linux/iio/trigger.h>
#include <linux/interrupt.h>
#include <linux/gpio/consumer.h>

#define REG_DEVID        0x00
#define REG_BW_RATE      0x2C
#define REG_POWER_CTL    0x2D
#define REG_INT_ENABLE   0x2E
#define REG_INT_MAP      0x2F
#define REG_INT_SOURCE   0x30
#define REG_DATA_FORMAT  0x31
#define REG_DATAX0       0x32
#define REG_FIFO_CTL     0x38
#define REG_FIFO_STATUS  0x39

#define DEVID_VAL        0xE5

#define WATERMARK        16

struct myadxl {
    struct spi_device *spi;
    struct mutex lock;
    int irq;
    struct iio_trigger *trig;
};

/* === Low-level SPI read/write with R/W and MB bits === */

static int ma_read(struct myadxl *m, u8 reg, u8 *buf, int n)
{
    u8 cmd = reg | 0x80 | (n > 1 ? 0x40 : 0);  /* R=1, MB if multi-byte */
    struct spi_transfer xfers[2] = {
        { .tx_buf = &cmd, .len = 1 },
        { .rx_buf = buf,  .len = n },
    };
    struct spi_message msg;
    spi_message_init(&msg);
    spi_message_add_tail(&xfers[0], &msg);
    spi_message_add_tail(&xfers[1], &msg);
    return spi_sync(m->spi, &msg);
}

static int ma_write(struct myadxl *m, u8 reg, u8 val)
{
    u8 buf[2] = { reg, val };    /* R=0, MB=0 implicit */
    return spi_write(m->spi, buf, 2);
}

/* === Per-sample reads (sysfs INFO_RAW) === */

static int ma_read_axis(struct myadxl *m, int axis, s16 *out)
{
    u8 buf[2];
    int err = ma_read(m, REG_DATAX0 + axis*2, buf, 2);
    if (err) return err;
    *out = (s16)(buf[0] | (buf[1] << 8));   /* little-endian on the wire */
```

```
    return 0;
}

/* === Bring-up === */

static int ma_init(struct myadxl *m)
{
    u8 devid;
    int err;

    err = ma_read(m, REG_DEVID, &devid, 1);
    if (err) return err;
    if (devid != DEVID_VAL) return -ENODEV;

    /* Standby first */
    ma_write(m, REG_POWER_CTL, 0x00);

    /* ±2g, full-resolution */
    ma_write(m, REG_DATA_FORMAT, 0x08);

    /* 100 Hz ODR */
    ma_write(m, REG_BW_RATE, 0x0A);

    /* FIFO: stream mode (mode 2), watermark 16 */
    ma_write(m, REG_FIFO_CTL, (2 << 6) | WATERMARK);

    /* Map watermark IRQ to INT1, enable */
    ma_write(m, REG_INT_MAP, 0x00);    /* all to INT1 */
    ma_write(m, REG_INT_ENABLE, 0x02); /* watermark only */

    /* Measure mode */
    ma_write(m, REG_POWER_CTL, 0x08);
    return 0;
}

/* === IIO read_raw === */

static int ma_read_raw(struct iio_dev *idev,
                       struct iio_chan_spec const *chan,
                       int *val, int *val2, long mask)
{
    struct myadxl *m = iio_priv(idev);
    s16 raw;
    int err;

    switch (mask) {
    case IIO_CHAN_INFO_RAW:
        mutex_lock(&m->lock);
        err = ma_read_axis(m, chan->scan_index, &raw);
        mutex_unlock(&m->lock);
        if (err) return err;
        *val = raw;
        return IIO_VAL_INT;
    case IIO_CHAN_INFO_SCALE:
        /* Full-res: 4 mg/LSB = 4×9.80665/1000 ≈ 0.0392 m/s²/LSB */
        *val = 0; *val2 = 39226;
        return IIO_VAL_INT_PLUS_MICRO;
    }
    return -EINVAL;
```

```
}

#define ACCEL_CH(axis, idx) {                                          \
    .type = IIO_ACCEL, .modified = 1, .channel2 = (axis),               \
    .info_mask_separate = BIT(IIO_CHAN_INFO_RAW),                       \
    .info_mask_shared_by_type = BIT(IIO_CHAN_INFO_SCALE),               \
    .scan_index = (idx),                                                \
    .scan_type = { .sign='s', .realbits=16, .storagebits=16,            \
                   .endianness=IIO_LE },                                 \
}

static const struct iio_chan_spec ma_channels[] = {
    ACCEL_CH(IIO_MOD_X, 0),
    ACCEL_CH(IIO_MOD_Y, 1),
    ACCEL_CH(IIO_MOD_Z, 2),
    IIO_CHAN_SOFT_TIMESTAMP(3),
};

static const struct iio_info ma_iio_info = {
    .read_raw = ma_read_raw,
};

/* === Watermark IRQ handler — runs in kernel thread === */

static irqreturn_t ma_irq_thread(int irq, void *p)
{
    struct iio_dev *idev = p;
    struct myadxl *m = iio_priv(idev);
    u8 int_src, fifo_status;
    int err;

    mutex_lock(&m->lock);

    err = ma_read(m, REG_INT_SOURCE, &int_src, 1);
    if (err) goto out;

    if (int_src & 0x02) {     /* watermark */
        ma_read(m, REG_FIFO_STATUS, &fifo_status, 1);
        int entries = fifo_status & 0x3F;

        for (int i = 0; i < entries; i++) {
            u8 sample[6];
            ma_read(m, REG_DATAX0, sample, 6);
            iio_push_to_buffers_with_timestamp(idev, sample,
                                               iio_get_time_ns(idev));
        }
    }
out:
    mutex_unlock(&m->lock);
    return IRQ_HANDLED;
}

/* === Probe / Remove === */

static int ma_probe(struct spi_device *spi)
{
    struct iio_dev *idev;
    struct myadxl *m;
    int err;
```

```
    idev = devm_iio_device_alloc(&spi->dev, sizeof(*m));
    if (!idev) return -ENOMEM;
    m = iio_priv(idev);
    m->spi = spi;
    mutex_init(&m->lock);

    spi->mode = SPI_MODE_3;
    spi->bits_per_word = 8;
    err = spi_setup(spi);
    if (err) return dev_err_probe(&spi->dev, err, "spi_setup failed\n");

    err = ma_init(m);
    if (err) return dev_err_probe(&spi->dev, err, "init failed\n");

    idev->name     = "myadxl345";
    idev->info     = &ma_iio_info;
    idev->modes    = INDIO_DIRECT_MODE | INDIO_BUFFER_TRIGGERED;
    idev->channels = ma_channels;
    idev->num_channels = ARRAY_SIZE(ma_channels);

    err = devm_iio_triggered_buffer_setup(&spi->dev, idev,
                                          NULL, NULL, NULL);
    if (err) return err;

    if (spi->irq > 0) {
        err = devm_request_threaded_irq(&spi->dev, spi->irq, NULL, ma_irq_thread,
                                        IRQF_TRIGGER_HIGH | IRQF_ONESHOT,
                                        "myadxl345", idev);
        if (err) return err;
    }

    return devm_iio_device_register(&spi->dev, idev);
}

static const struct of_device_id ma_of_match[] = {
    { .compatible = "linuxlearn,myadxl345" },
    { }
};
MODULE_DEVICE_TABLE(of, ma_of_match);

static const struct spi_device_id ma_id[] = {
    { "myadxl345", 0 },
    { }
};
MODULE_DEVICE_TABLE(spi, ma_id);

static struct spi_driver ma_driver = {
    .driver = {
        .name = "myadxl345",
        .of_match_table = ma_of_match,
    },
    .probe    = ma_probe,
    .id_table = ma_id,
};
module_spi_driver(ma_driver);

MODULE_LICENSE("GPL");
```

DT:

```
&ecspi3 {
    adxl345@0 {
        compatible = "linuxlearn,myadxl345";
        reg = <0>;
        spi-max-frequency = <5000000>;
        spi-cpha;
        spi-cpol;                                   /* mode 3 */
        interrupt-parent = <&gpio4>;
        interrupts = <14 IRQ_TYPE_LEVEL_HIGH>;     /* watermark IRQ */
    };
};
```

Test:

```
[root@pa-mini:~]# insmod myadxl345.ko
[root@pa-mini:~]# cat /sys/bus/iio/devices/iio:device0/in_accel_z_raw
253        ← ~1 g (1000 mg / 4 mg/LSB = 250), close to 253 LSB
[root@pa-mini:~]# cat /sys/bus/iio/devices/iio:device0/in_accel_x_scale
0.039226
```

Then enable buffered capture, same workflow as Ch 70 (`scan_elements/*_en`, `buffer/enable`). At 100 Hz with watermark 16, IRQs fire ~6×/sec, draining 16 samples each. CPU overhead negligible.

What we got, ~280 lines:

- SPI command framing with R/W + MB bits.
- IIO INFO_RAW sysfs.

> **sysfs:** a kernel-generated filesystem under /sys that exposes devices, drivers, and attributes.

- Triggered buffered capture *driven by the chip's own watermark IRQ*.

What we skipped:

- Multi-rate ODR support (we hardcoded 100 Hz).
- Tap / double-tap / activity / inactivity detection (ADXL345's interesting features).
- Self-test.

71.6 LSM6DSO, the FSM and MLC

LSM6DSO contains a **finite-state-machine engine** (FSM) and a **machine-learning core** (MLC), runtime-programmable accelerators that detect specific motion patterns *without CPU involvement*:

- **FSM**: a small bytecode language (~256 instructions, configurable). You write a state machine ("if x_accel > 0.5 g for 100 ms then z_accel > -0.5 g for 200 ms then trigger"). The chip runs it at the IMU sample rate and emits an IRQ on match. Detect "doorbell pressed" or "drone has crashed" with zero CPU.
- **MLC**: a decision-tree classifier (8 trees, depth 8). Compiled from a Python tool with sample-labeled training data. Detect "walking vs running vs cycling" with ~90 % accuracy at < 1 % CPU.

These are special. When you need FSM or MLC, no other current-production part offers the same. Mainline support: `drivers/iio/imu/st_lsm6dsx/` includes FSM and MLC firmware-loading via the IIO config interface.

For ordinary use (just sample at 1 kHz), LSM6DSO is a normal SPI IMU, same model as ADXL345 with more channels and a bigger FIFO.

71.7 ICM-42688, the noise winner

ICM-42688 has the lowest accel noise floor in this category (60 µg/√Hz), meaningful for vibration analysis where you want to see micro-g signals. Its 2 KB FIFO supports streaming at 8 kHz with sane IRQ rates.

Register-set is bank-organised (like ICM-20948). Mainline driver: `drivers/iio/imu/inv_icm42600/`.

Its distinguishing feature is two SPI ports, UI (Userspace Interface) for normal samples, AUX for an external magnetometer pass-through. The MPU9250's aux-bus idea but cleaner.

71.8 Now: the mainline drivers

DT for ADXL345:

```
&ecspi3 {
    adxl345@0 {
        compatible = "adi,adxl345";
        reg = <0>;
        spi-max-frequency = <5000000>;
        spi-cpha; spi-cpol;
        interrupt-parent = <&gpio4>;
        interrupts = <14 IRQ_TYPE_LEVEL_HIGH>;
```

```
    };
};
```

For LSM6DSO: `compatible = "st,lsm6dso";`. For ICM-42688: `compatible = "invensense,icm42688";`.

Mainline drivers expose richer attributes than our from-scratch:

- `in_accel_sampling_frequency_available`: full ODR list.
- `in_accel_scale_available`: full range list.
- `events/in_accel_thresh_rising_value`: tap/threshold events.
- FIFO-watermark configurable via `buffer/watermark`.

71.9 Key Notes

- A FIFO watermark batches samples to reduce interrupt and bus overhead. Larger batches also increase delivery latency and the amount of data at risk during a stall.
- SPI framing includes mode, chip-select continuity and device-specific command bits. A common bus does not imply a common register protocol.
- An interrupt timestamp describes when service was requested, not automatically when every FIFO sample was measured. Sample timing needs an explicit reconstruction model.
- Overflow handling is part of the data interface. A pipeline that silently loses samples can produce convincing but incorrect motion or vibration results.

71.10 Pitfalls

- **SPI mode wrong.** ADXL345 is mode 3 (CPOL=1, CPHA=1). LSM6DSO is mode 0 or 3. ICM-42688 is mode 0. Each datasheet's "SPI timing" diagram tells you. Wrong mode → garbage reads.
- **CS asserted across the wrong byte count.** Reading 6 bytes but the SPI controller deasserts CS after byte 1 → chip resets pointer and you re-read register 0x32 six times. Use a single `spi_message` with all transfers chained.
- **R/W bit position.** Bit 7. MB bit at 6. Different per chip, ICM-42688 uses different bits. Read the datasheet's "SPI protocol" section.
- **Endianness mismatch.** ADXL345 puts data out little-endian. MPU6050 puts it out big-endian. Easy to swap by accident.

- **FIFO overrun.** If user-space drains too slowly, the FIFO overflows and you lose samples silently. Detect via the OVERRUN bit in INT_SOURCE (FIFO_STATUS for some chips).
- **Self-test forgotten.** Each chip has a self-test mode (forces internal mechanical stimulation). Run on power-up to verify the chip is functional. Ship products with this in startup self-check.
- **Pull-ups on /CS during reset.** Some boards leave /CS floating during SoC reset. Chip enters undefined state. Tie /CS HIGH at idle (10 kΩ to VCC or controller-default).
- **Multi-chip SPI with shared GPIO IRQ.** Multiple IMUs sharing a watermark-IRQ GPIO. Decode in handler by reading each chip's INT_SOURCE. Only the one with bit set wants service.

> **MCU bridge:** Think of Linux GPIO like the same pin set/reset block you used on STM32, but accessed through a kernel subsystem that owns numbering, direction, interrupts, and user-space exposure. **GPIO:** General-Purpose Input/Output, a pin controlled as a digital input, output, or interrupt source.

71.11 Going deeper

- **`drivers/iio/accel/adxl345_core.c`** + `adxl345_spi.c`, production ADXL345.
- **`drivers/iio/imu/st_lsm6dsx/`**: LSM6DSO + family. FSM/MLC firmware loading.
- **`drivers/iio/imu/inv_icm42600/`**: ICM-42688.
- **ADXL345 datasheet (Analog Devices Rev G)**: register map + FIFO modes.
- **LSM6DSO datasheet (STMicro DS12140)**: FSM/MLC sections.
- **ICM-42688-P datasheet (InvenSense DS-000347)**: anti-aliasing filter design.
- **`Documentation/iio/iio_devbuf.rst`**: buffered IIO model.

> **End of Group C, Motion sensors (Ch 70–71).** I²C IMUs cover up to ~1 kHz, SPI IMUs the rest of the way to ~10 kHz. Both use IIO triggered buffers, both support watermark-IRQ patterns when ODR is high.

> Next chapter: **Chapter 72: Distance & proximity sensors (VL53L0X / HC-SR04 / GP2Y0A).** Three approaches to "how far away is that object?" Each with different physics and very different drivers.

Chapter 72: Distance & proximity sensors

What: three radically different “how far away is that object” sensors: **STMicro VL53L0X** (I²C, laser time-of-flight, mm precision, requires firmware-blob upload at probe), **HC-SR04** (GPIO, ultrasonic, famously hard to time accurately under Linux), **Sharp GP2Y0A** (analog IR, ADC-fed). For each: physics, protocol, the mainline driver, and a from-scratch driver for VL53L0X (the most interesting case) plus a clear-eyed look at why HC-SR04 is hard on Linux. **MCU bridge:** Think of Linux GPIO like the same pin set/reset block you used on STM32, but accessed through a kernel subsystem that owns numbering, direction, interrupts, and user-space exposure. **GPIO:** General-Purpose Input/Output, a pin controlled as a digital input, output, or interrupt source.

Why: distance sensing is in every robot, every parking assist, every smart-lighting fixture. The three classes cover the practical price/accuracy spectrum: $0.50 IR analog → $3 ultrasonic → $8 ToF laser. Knowing the trade-offs lets you pick correctly and not promise users a ranging accuracy you cannot actually deliver.

Focus: Time-of-flight measures with electronics. Ultrasonic measures with sound. IR measures with reflected brightness. Three different physics. ToF measures photon round-trip directly (mm-accurate, fast, expensive). Ultrasonic measures sound round-trip (cm-accurate, slow, cheap). IR measures reflected intensity then maps to a non-linear curve (poor accuracy, very cheap). Each driver’s complexity tracks the physics.

72.1 Sensor comparison

	STMicro VL53L0X	HC-SR04	Sharp GP2Y0A21YK
Physics	940 nm laser ToF	40 kHz ultrasonic	850 nm IR triangulation
Range	30 mm, 2000 mm	20 mm, 4000 mm	100 mm, 800 mm
Accuracy	±3% (≤ 1.2 m), ±7% (1.2, 2 m)	±3 mm	±5% non-linear
Update rate	50 Hz (default)	~20 Hz	~25 Hz

	STMicro VL53L0X	HC-SR04	Sharp GP2Y0A21YK
Interface	I²C (0x29)	GPIO (TRIG + ECHO)	Analog 0, 3 V
Beam cone	~25°	~30°	~5°
Sunlight tolerance	poor (IR ambient washout)	excellent	poor
Volume price	$5–8	$1–2	$3–5
Mainline driver	`vl53l0x-i2c.c`	none (GPIO + user-space or sleeping-bitbang)	none (analog → IIO ADC)

Pick guide:

- **VL53L0X**: indoor robotics, gesture detection, gauging, where mm precision matters.
- **HC-SR04**: low-cost cm-precision. Outdoor (it tolerates sun). Cheap robots, parking sensors.
- **GP2Y0A**: rough proximity (“is there a wall in front?”), trip-wire, line-following robots.

72.2 The three physics

VL53L0X, Time-of-Flight

A pulsed 940 nm VCSEL (vertical-cavity surface-emitting laser) emits ~10 ns pulses. A Single-Photon Avalanche Diode (SPAD) array detects returning photons. The chip times the round-trip with picosecond resolution → distance = c × t / 2.

Range is 2 m in indoor light. In direct sunlight it drops to ~0.6 m because 940 nm ambient noise dominates. The chip is accurate and fast, but more complex than the alternatives.

HC-SR04, Ultrasonic

A transducer emits 8 pulses of 40 kHz ultrasound on the TRIG pin’s rising edge. A receiver listens for the echo and pulses the ECHO line for as long as the round-trip takes. Distance = (echo_pulse_us / 58) cm, or in metric: `t_us × 0.000343 / 2` m.

Robust to light, fails on soft / angled / tiny targets (poor acoustic reflection). Slow: at 4 m max range, one measurement takes ~24 ms (longer if no echo, you need a timeout).

GP2Y0A, IR triangulation

An IR LED emits a 5 ms pulse. A linear PSD (position-sensitive detector) measures *where* the reflected spot lands on the sensor (not how bright). The angle of return → distance via triangulation.

Non-linear output curve (output voltage *not* monotonic with distance, has a peak around 80 mm). Datasheet provides a piecewise table. Cheap and good enough for “object near” detection. Bad for precise ranging.

72.3 Protocol, VL53L0X

VL53L0X is unusual: it needs a long initial register-write sequence, effectively a firmware blob, uploaded at every probe. Unlike most I²C devices that have a fixed register-set behavior, VL53L0X needs to be initialized by uploading **160 separate register writes** at probe, calibration constants, internal-state-machine setup, and tuning parameters. STMicro’s API ships these as a long list in their reference code. The kernel driver embeds them too.

Register map (just the headlines):

Reg	Name	Purpose
0xC0	IDENTIFICATION_MODEL_ID	Always 0xEE (or 0xEEAA depending on rev)
0xC1	IDENTIFICATION_REV_ID	Silicon revision
0x88	POWER_MANAGEMENT_GO1	Power mode
0x80	(multi-purpose)	Various commands
0x00	SYSRANGE_START	Write 0x01 → start a measurement
0x13	RESULT_INTERRUPT_STATUS	Bit 0 set when new measurement ready

Reg	Name	Purpose
0x14	RESULT_RANGE_STATUS + RANGE_MM_HI/LO	Status + 16-bit distance
0x0A	SYSTEM_INTERRUPT_CONFIG_GPIO	What triggers INT1
0x60..0xA0	various magic registers	Tuning, calibration, undocumented

The init sequence

```
1. Read 0xC0 (verify chip-id = 0xEE).
2. Write to ~30 "magic" registers to enable internal voltage references.
3. Read SPAD info from a private register sequence.
4. Apply the SPAD calibration map (10+ writes).
5. Load default tuning settings (~80 register writes — basically opaque to user).
6. Configure interrupt: INT on new-sample-ready, active-low.
7. Perform reference calibration (VHV and phase).
8. Start measurement mode.
```

Step 5 is the “tuning settings load”, the famous register-pair dump:

```
static const u8 vl53l0x_default_tuning[] = {
    0xFF, 0x01,    0x00, 0x00,
    0xFF, 0x00,    0x09, 0x00,
    0x10, 0x00,    0x11, 0x00,
    0x24, 0x01,    0x25, 0xFF,
    ... (about 80 pairs) ...
    0xFF, 0x00,    0x80, 0x00,
};
```

Each pair is `(register, value)`. The driver writes them sequentially. The values are STMicro’s IP, they’re a calibrated “factory good” state-machine config for the SPAD array. Don’t try to derive them from datasheet. They aren’t documented.

A measurement cycle

```
Host: write 0x00 = 0x01    (SYSRANGE_START — single-shot)
... wait for interrupt or poll bit 0 of 0x13 ...
Host: read 2 bytes from 0x1E    (RANGE_MM_HI/LO)
Host: write 0x0B = 0x01    (clear interrupt)
```

Distance = `(buf[0] << 8) | buf[1]` mm. Range status (0x14) tells you about errors: signal too weak (target too far), sigma too high (noisy ambient).

72.4 How the mainline `vl53l0x` driver works

Source: `drivers/iio/proximity/vl53l0x-i2c.c` (~600 lines).

> **IIO:** Industrial I/O, Linux's subsystem for sensors, ADCs, DACs, and buffered sampled data.

Surprisingly compact for a chip with a firmware-blob-equivalent. The driver embeds STMicro's tuning data as a const u8 array and writes it at probe. The init code uses `regmap_multi_reg_write` for the tuning dump.

```
/* Simplified probe */
static int vl53l0x_probe(struct i2c_client *client)
{
    struct vl53l0x_data *data;
    struct iio_dev *indio_dev;
    u8 chip_id;
    int err;

    indio_dev = devm_iio_device_alloc(&client->dev, sizeof(*data));
    data = iio_priv(indio_dev);
    data->client = client;
    mutex_init(&data->lock);

    /* Optionally toggle XSHUT to force reset */
    if (data->xshut_gpio)
        gpiod_set_value_cansleep(data->xshut_gpio, 0);

    /* Verify chip-id */
    err = i2c_smbus_read_byte_data(client, VL53L0X_REG_IDENTIFICATION_MODEL_ID);
    if (err != 0xEE) return -ENODEV;

    /* Big init: ~160 register writes from a tuning blob */
    err = vl53l0x_init(data);
    if (err) return err;

    /* IIO setup */
    indio_dev->name = "vl53l0x";
    indio_dev->info = &vl53l0x_iio_info;
    indio_dev->modes = INDIO_DIRECT_MODE;
    indio_dev->channels = vl53l0x_channels;
    indio_dev->num_channels = ARRAY_SIZE(vl53l0x_channels);

    return devm_iio_device_register(&client->dev, indio_dev);
}

static int vl53l0x_init(struct vl53l0x_data *data)
{
    int err;
    /* Data init */
    err = i2c_smbus_write_byte_data(data->client, 0x88, 0x00); ...
    /* Static init */
    err = vl53l0x_load_tuning(data);   /* writes the 80-pair blob */
    /* Reference calibration */
```

```
    err = vl53l0x_perform_ref_calibration(data);
    /* Set measurement timing budget */
    err = vl53l0x_set_measurement_timing_budget(data, 33000);   /* µs */
    return 0;
}

static int vl53l0x_read_proximity(struct vl53l0x_data *data, int *val)
{
    int err;
    u8 buf[2];
    int retries = 100;

    /* Start single-shot */
    err = i2c_smbus_write_byte_data(data->client, 0x00, 0x01);
    if (err) return err;

    /* Wait for completion */
    while (retries--) {
        int status = i2c_smbus_read_byte_data(data->client, 0x13);
        if (status & 0x07) break;     /* new sample ready */
        msleep(1);
    }
    if (retries < 0) return -ETIMEDOUT;

    /* Read range */
    err = i2c_smbus_read_i2c_block_data(data->client, 0x1E, 2, buf);
    if (err < 0) return err;
    *val = (buf[0] << 8) | buf[1];

    /* Clear interrupt */
    i2c_smbus_write_byte_data(data->client, 0x0B, 0x01);
    return 0;
}
```

What looks complex in the driver is mostly the tuning-blob loop. The actual measurement is a single-shot trigger, busy-poll, read 2 bytes.

72.5 Writing a VL53L0X driver from scratch

Goal: working driver exposing `in_distance_input` in mm. We'll include a minimal-but-functional init sequence (enough to get readings, omitting the deepest calibration tweaks).

`myvl53l0x.c`:

```
#include <linux/init.h>
#include <linux/module.h>
#include <linux/i2c.h>
#include <linux/delay.h>
#include <linux/iio/iio.h>

#define REG_MODEL_ID         0xC0
#define REG_SYSRANGE_START   0x00
#define REG_RESULT_INT_STATUS 0x13
#define REG_RESULT_RANGE     0x1E
```

```
#define REG_INT_CLEAR        0x0B

#define EXPECTED_MODEL_ID    0xEE

/* Minimal tuning blob — first 20-ish writes that get us a working chip.
 * Full STMicro blob is ~80 pairs; this is enough for default range. */
static const u8 vl_min_tuning[][2] = {
    {0x88, 0x00},
    {0x80, 0x01},
    {0xFF, 0x01},
    {0x00, 0x00},
    {0x91, 0x00},   /* stop variable */
    {0x00, 0x01},
    {0xFF, 0x00},
    {0x80, 0x00},
    /* enable signal-rate-msrc-limit defaults */
    {0x44, 0xFF},
    {0x46, 0x06},
    {0x60, 0x00},
    {0x46, 0xFF},
    /* set measurement-timing-budget for ~33 ms */
    {0x70, 0x04},
    {0x71, 0x08},
    /* misc */
    {0x4E, 0x12},
    {0x4F, 0x00},
    {0x01, 0xFF},
};

struct myvl {
    struct i2c_client *client;
    struct mutex lock;
};

static int mv_write(struct myvl *m, u8 reg, u8 val)
{
    return i2c_smbus_write_byte_data(m->client, reg, val);
}

static int mv_read(struct myvl *m, u8 reg)
{
    return i2c_smbus_read_byte_data(m->client, reg);
}

static int mv_init(struct myvl *m)
{
    int err, i;

    /* Verify chip */
    err = mv_read(m, REG_MODEL_ID);
    if (err < 0) return err;
    if (err != EXPECTED_MODEL_ID)
        return -ENODEV;

    /* Apply minimal tuning */
    for (i = 0; i < ARRAY_SIZE(vl_min_tuning); i++) {
        err = mv_write(m, vl_min_tuning[i][0], vl_min_tuning[i][1]);
        if (err) return err;
    }
```

```
    return 0;
}

static int mv_measure(struct myvl *m, int *out_mm)
{
    int retries = 100, status;
    u8 buf[2];
    int err;

    err = mv_write(m, REG_SYSRANGE_START, 0x01);
    if (err) return err;

    while (retries-- > 0) {
        status = mv_read(m, REG_RESULT_INT_STATUS);
        if (status < 0) return status;
        if (status & 0x07) break;
        usleep_range(500, 1500);
    }
    if (retries <= 0) return -ETIMEDOUT;

    err = i2c_smbus_read_i2c_block_data(m->client, REG_RESULT_RANGE, 2, buf);
    if (err < 0) return err;
    *out_mm = (buf[0] << 8) | buf[1];

    /* Clear interrupt */
    mv_write(m, REG_INT_CLEAR, 0x01);
    return 0;
}

static int mv_read_raw(struct iio_dev *idev,
                       struct iio_chan_spec const *chan,
                       int *val, int *val2, long mask)
{
    struct myvl *m = iio_priv(idev);
    int mm, err;

    if (mask != IIO_CHAN_INFO_PROCESSED) return -EINVAL;
    if (chan->type != IIO_DISTANCE) return -EINVAL;

    mutex_lock(&m->lock);
    err = mv_measure(m, &mm);
    mutex_unlock(&m->lock);
    if (err) return err;
    *val = mm;
    return IIO_VAL_INT;
}

static const struct iio_chan_spec mv_channels[] = {
    { .type = IIO_DISTANCE,
      .info_mask_separate = BIT(IIO_CHAN_INFO_PROCESSED) },
};

static const struct iio_info mv_iio_info = {
    .read_raw = mv_read_raw,
};

static int mv_probe(struct i2c_client *client)
{
```

```
    struct iio_dev *idev;
    struct myvl *m;
    int err;

    idev = devm_iio_device_alloc(&client->dev, sizeof(*m));
    if (!idev) return -ENOMEM;
    m = iio_priv(idev);
    m->client = client;
    mutex_init(&m->lock);

    err = mv_init(m);
    if (err) return dev_err_probe(&client->dev, err, "init failed\n");

    idev->name = "myvl53l0x";
    idev->info = &mv_iio_info;
    idev->modes = INDIO_DIRECT_MODE;
    idev->channels = mv_channels;
    idev->num_channels = ARRAY_SIZE(mv_channels);

    return devm_iio_device_register(&client->dev, idev);
}

static const struct of_device_id mv_of_match[] = {
    { .compatible = "linuxlearn,myvl53l0x" },
    { }
};
MODULE_DEVICE_TABLE(of, mv_of_match);

static const struct i2c_device_id mv_id[] = { { "myvl53l0x", 0 }, { } };
MODULE_DEVICE_TABLE(i2c, mv_id);

static struct i2c_driver mv_driver = {
    .driver = {
        .name = "myvl53l0x",
        .of_match_table = mv_of_match,
    },
    .probe = mv_probe,
    .id_table = mv_id,
};
module_i2c_driver(mv_driver);

MODULE_LICENSE("GPL");
```

DT:

```
&i2c1 {
    vl53l0x@29 {
        compatible = "linuxlearn,myvl53l0x";
        reg = <0x29>;
    };
};
```

Test:

```
[root@pa-mini:~]# insmod myvl53l0x.ko
[root@pa-mini:~]# cat /sys/bus/iio/devices/iio:device0/in_distance_input
312       ← 312 mm to nearest reflective surface
```

Wave a hand in front: number drops. Hold further: rises. ~50 Hz update rate.

The minimal-tuning blob gives “good enough” readings (~5 % accuracy in indoor light). For production with mm precision and reliable behavior in edge cases (cover dust, glass, polarisation), use the full mainline driver, which has the complete STMicro blob plus the reference-calibration routines.

72.6 HC-SR04, why it’s hard on Linux

The HC-SR04 protocol is dead simple in concept:

```
TRIG → drive HIGH for ≥ 10 µs, then LOW.
ECHO ← pulse stays high for (round-trip time) µs.
distance_cm = pulse_us / 58
```

But measuring a pulse of variable duration (60 µs to ~23 ms) with µs accuracy from Linux GPIO is *hard*:

- Standard kernel preemption: any other ISR or higher-priority thread can delay your edge measurement by 100+ µs → 17 mm error.
- The GPIO IRQ → user-space wakeup latency is typically 50–200 µs, even more under load.

> **MCU bridge:** Think of an IRQ like an EXTI/NVIC interrupt path, except Linux splits the hard interrupt from deferred work and must share lines across drivers. **IRQ:** interrupt request, the signal path that tells the CPU or interrupt controller that hardware needs service.

- `gpiomon` from libgpiod has the same problem.

The honest options:

1. **PREEMPT_RT + threaded IRQ + ktime_get_ns** in driver: latency ~20 µs typical, ~150 µs worst case. → 5 mm worst-case error. Usable.

> **PREEMPT_RT:** the Linux real-time patch set that makes more kernel paths preemptible and reduces latency.

1. **Capture-input timer hardware** (a TIM block on the SoC configured to capture-compare on the ECHO edge). I.MX6ULL's GPT has this. Very accurate (~10 ns), but requires writing a driver for the GPT capture mode, substantial work.
2. **Dedicated MCU helper** (an ESP32 or STM32 doing the timing, talking back to i.MX6ULL via I²C). The right answer for production systems.
3. **PRU/Cortex-M co-processor** on SoCs that have one (TI Sitara, NXP i.MX7/8). I.MX6ULL has Cortex-M4 in some variants. Not in i.MX6ULL.

A from-scratch HC-SR04 driver for Linux that gives reasonable but unspectacular results:

```
static int sr04_measure(struct sr04 *s, int *out_cm)
{
    ktime_t t_start, t_end;
    int err;

    /* TRIG high for 12 µs */
    gpiod_set_value(s->trig, 1);
    udelay(12);
    gpiod_set_value(s->trig, 0);

    /* Wait for ECHO high (busy-wait with timeout) */
    t_start = ktime_get();
    while (!gpiod_get_value(s->echo)) {
        if (ktime_to_us(ktime_sub(ktime_get(), t_start)) > 5000)
            return -ETIMEDOUT;
        cpu_relax();
    }
    t_start = ktime_get();    /* echo went high; reset clock */

    /* Wait for ECHO low */
    while (gpiod_get_value(s->echo)) {
        if (ktime_to_us(ktime_sub(ktime_get(), t_start)) > 30000)
            return -ETIMEDOUT;
        cpu_relax();
    }
    t_end = ktime_get();

    *out_cm = ktime_to_us(ktime_sub(t_end, t_start)) / 58;
    return 0;
}
```

The kernel busy-waits in two loops here. That keeps one CPU pinned for the full ~25 ms measurement. With PREEMPT_RT and a SCHED_FIFO priority, accuracy improves. Without, it's still ±2 cm in the typical case.

In short: do not ship products with HC-SR04 wired directly to Linux GPIO. Either use a co-processor or pick a different sensor.

72.7 GP2Y0A, analog needs an ADC

GP2Y0A21YK outputs 0–3 V proportional to distance via a non-linear curve. You read with an ADC. In Linux that means an external I²C/SPI ADC (Ch 80) or the SoC's internal ADC.

i.MX6ULL has ADC1/ADC2, 12-bit, ~1 MS/s, mainline driver `drivers/iio/adc/vf610_adc.c`. Wire GP2Y0A's output to an ADC channel. In IIO:

```
[root@pa-mini:~]# cat /sys/bus/iio/devices/iio:device0/in_voltage0_raw
1834
[root@pa-mini:~]# cat /sys/bus/iio/devices/iio:device0/in_voltage_scale
0.732421875
# → V = 1834 × 0.732 / 1000 = 1.343 V
```

User-space converts voltage to mm via the datasheet's piecewise table or polynomial. The non-linearity peaks ~80 mm. Below that distance, voltage *decreases* again, a single voltage maps to two distances. **Always combine with a hard minimum bracket** (mechanically prevent the target from being closer than 100 mm).

This is the only case where the "driver" is just the IIO ADC driver. The sensor-specific math lives in user-space.

72.8 Now: the mainline driver

DT for VL53L0X:

```
&i2c1 {
    vl53l0x@29 {
        compatible = "st,vl53l0x";
        reg = <0x29>;
        interrupt-parent = <&gpio4>;
        interrupts = <10 IRQ_TYPE_EDGE_FALLING>;
        xshut-gpios = <&gpio4 11 GPIO_ACTIVE_LOW>;
    };
};
```

`xshut-gpios` is the chip's external-shutdown pin, letting the driver reset the chip if it gets stuck. The `interrupts` line is GPIO1 (the chip's data-ready IRQ) which lets the driver wake on sample-ready instead of polling.

Kernel config: `CONFIG_VL53L0X_I2C=y`.

For multi-chip setups (3 VL53L0X looking forward/left/right), the chips share I²C address 0x29 by default. Use XSHUT to hold all-but-one off at boot. Each one is brought up sequentially and reassigned to a unique address before the next is woken. This sequencing happens in the driver via `xshut-gpios`.

72.9 Key Notes

- Distance sensors use different physical mechanisms, so target shape, ambient conditions and range affect them differently. Software cannot make their limitations interchangeable.
- Pulse-based ultrasonic ranging needs a timing owner that can capture edges reliably. User-space scheduling latency is not equivalent to signal flight time.
- Analog ranging includes the ADC, reference and nonlinear conversion path. Some parts of the response curve cannot uniquely identify distance.
- Invalid or out-of-range results need an explicit status. Returning a convenient zero can turn a measurement failure into a false nearby-object indication.

72.10 Pitfalls

- **VL53L0X under sunlight.** Range collapses to ~60 cm. If outdoor use is required, pick ultrasonic.
- **VL53L0X behind glass.** The chip’s emitter reflects off the inner surface of the glass, and you read 0 mm forever. Use a recessed window or tilt the cover slightly.
- **VL53L0X minimum range.** Below 30 mm, readings are nonsense. Don’t trust them.
- **HC-SR04 narrow targets.** Sound wave is ~25° cone. A thin pole reflects little, readings drop out. Hold a flat board for testing.
- **HC-SR04 echo from the floor.** In open setups, the floor reflects ultrasound. You read floor distance, not target. Angle the sensor slightly upward.
- **GP2Y0A double-valued zone.** Voltage isn’t monotonic with distance below ~80 mm. Constrain mechanically.
- **GP2Y0A ambient light.** Strong IR (sunlight, incandescent bulb) saturates the receiver. Indoor use only.
- **VL53L0X I²C address conflict.** Default 0x29. If your board has another chip there, sequence with XSHUT.
- **Pulsing TRIG too fast on HC-SR04.** Min 60 ms between measurements. Faster = sensor is still listening for previous echo. Readings get confused.

72.11 Going deeper

- **`drivers/iio/proximity/vl53l0x-i2c.c`**: production VL53L0X driver. Read the tuning blob comments.

- **STMicro VL53L0X API source** at https://www.st.com/en/embedded-software/stsw-img005.html. The full reference C implementation. ~10000 lines. Useful for understanding what the kernel driver's minimal init omits.
- **VL53L0X datasheet (STMicro DS33054)**: register summary, calibration overview.
- **HC-SR04 module documentation** (multiple vendors, all roughly the same), timing diagram.
- **GP2Y0A21YK datasheet (Sharp)**: non-linear curve table (page 5).
- **`Documentation/iio/iio_devbuf.rst`**: for buffered ranging-over-time applications.

> Next chapter: **Chapter 73: Magnetometer / compass.** From "how far" to "which way", the magnetic vector of the Earth, the hard-iron and soft-iron calibration that nobody tells you about.

Chapter 73: Magnetometer / compass

> **What:** three I²C magnetometers: **Honeywell HMC5883L** (legacy classic, EOL but ubiquitous on hobbyist boards), **QST QMC5883L** (cheap clone with quirky register-set differences), **Memsic MMC5983MA** (modern low-noise). Most of this chapter is not driver code. It is calibration: **hard-iron and soft-iron calibration**, the universal "my compass points 23° wrong" problem and how to fix it.
>
> **Why:** any product that needs to know which way it's facing, drone, robot vacuum, AR headset, GPS-assisted navigation, needs a magnetometer. The math is simple. The *calibration* is what separates a useful compass from a useless one. Many products ship without calibration. Their compass is off by 10–30°, and the IMU often gets blamed.
>
> **Focus:** Calibration runs in user-space. The driver's job is to deliver raw X/Y/Z in stable, scaled units. The driver reports raw `µT × scale`. User-space collects samples, fits an ellipsoid model, computes hard-iron (offset) and soft-iron (skew matrix). After applying the correction, raw 3-axis readings become Earth-magnetic-field vectors with < 1° error. Without calibration: 10–30° error is typical, depending on what's mounted near the sensor.

73.1 Chip comparison

	Honeywell HMC5883L	QST QMC5883L	Memsic MMC5983MA
Range	±0.88 / 1.3 / 1.9 / 2.5 / 4 / 4.7 / 5.6 / 8.1 Gauss	±2 / 8 Gauss	±8 Gauss
Resolution	12-bit (in ±0.88 G), ~0.7 mG/LSB	16-bit, ~30 µG/LSB	18-bit, ~0.06 mG/LSB
Noise floor	2 mG RMS	2 mG RMS	0.4 mG RMS
Update rate	0.75, 75 Hz	10/50/100/200 Hz	up to 1 kHz
I²C address	0x1E	0x0D	0x30

	Honeywell HMC5883L	QST QMC5883L	Memsic MMC5983MA
Reg map	“real” HMC layout	shifted (different from HMC)	new
Compatible?	yes mainline (`hmc5843_i2c.c`)	yes mainline (`qmc5883.c`)	yes (`mmc56x3.c`, recent kernels)
Lifecycle	EOL since 2018	active (cheap China supply)	active
Volume price	$4–6 (eBay clones $1)	$0.50–1.50	$4–6

Watch for this: every $1 eBay “HMC5883L” breakout is actually a *QMC5883L*. The pinout is similar. The protocol is different. If your “HMC5883L” doesn’t probe at 0x1E, try QMC5883L at 0x0D.

Pick guide:

- **HMC5883L** for legacy maintenance only.
- **QMC5883L** for cheap projects. Understand the register-set quirk.
- **MMC5983MA** for serious products needing low-noise + high-rate.

73.2 The physics, why magnetometers care so much about the environment

A magnetometer measures the **local magnetic field vector** in three orthogonal axes. The Earth produces ~50 µT (0.5 Gauss). Local sources can be much stronger:

- **Hard iron** (permanent magnets, magnetised steel): adds a fixed DC offset to each axis.
- **Soft iron** (any ferromagnetic material): bends the field unevenly, different axes see different scale factors, axes may not be exactly orthogonal anymore.

If you take the chip out of the box and read raw, the data lies on a *3D ellipsoid* (not a sphere centered on origin) with the ellipsoid’s center *offset* from origin. The ellipse-fitting calibration recovers two corrections:

- **Offset vector b**: subtract from each raw reading to center on origin (hard-iron).
- **3×3 transformation matrix A**: multiply the centered reading to make the ellipsoid a sphere (soft-iron).

Calibrated reading = A × (raw − b). The radius of the resulting sphere equals the magnitude of Earth's field (~50 µT) at your location. Heading angle from the X/Y components is then accurate to ~1°.

Without calibration: heading error of 10–30°, wrong enough to be useless for navigation.

73.3 Protocol, HMC5883L

Register map (the "real" HMC):

Reg	Name	Purpose
0x00	CONFIG_A	Output rate, oversampling, measurement mode
0x01	CONFIG_B	Gain/range select
0x02	MODE	Continuous / single / idle
0x03..0x08	X_MSB, X_LSB, Z_MSB, Z_LSB, Y_MSB, Y_LSB	16-bit signed values (note: Z then Y!)
0x09	STATUS	DATA_READY bit
0x0A..0x0C	ID_A/B/C	"H43" identification

Bring-up:

1. Read identification bytes (0x0A..0x0C), should be 'H', '4', '3'.
2. Write CONFIG_A (0x00) = 0x70: 8-sample averaging, 15 Hz output rate.
3. Write CONFIG_B (0x01) = 0x20: gain ±1.3 Gauss, 1090 LSB/G.
4. Write MODE (0x02) = 0x00: continuous measurement.
5. After ~7 ms (per datasheet), data is ready. Read 6 bytes from 0x03.
6. Apply scale: `field_uT = raw / 1090 * 100` (to get µT from 1090 LSB/Gauss).

Each measurement read is 6 bytes back-to-back. Note the **X/Z/Y order** (HMC's idiosyncrasy, most other chips do X/Y/Z).

73.4 Protocol, QMC5883L (different!)

QMC5883L uses a *different* register layout despite the similar-looking name and pinout. The differences that matter:

Reg	QMC	HMC
Data start	0x00	0x03
Status	0x06	0x09
Temperature	0x07–0x08	none
Control 1	0x09	(different bits)
Control 2	0x0A	(different)
Period (refresh)	0x0B	none
I²C address	0x0D	0x1E

Mode bits in QMC's control register are also different. HMC5883L code does not work on a QMC5883L, the chip will simply not respond. This is the #1 reason "my $1 HMC5883L doesn't work", it's not an HMC5883L.

QMC bring-up:

1. Write 0x0B = 0x01: set the "period" register (mandatory. Chip won't work without it).
2. Write 0x09 = 0x1D: 200 Hz output rate, ±8 G range, 512 oversampling, continuous mode.
3. Read 6 bytes from 0x00, X / Y / Z order, *little-endian* per axis (vs HMC's big-endian).

73.5 How the mainline drivers work

`drivers/iio/magnetometer/hmc5843_i2c.c` + `hmc5843_core.c` covers HMC5843 + HMC5883 + HMC5883L. Standard regmap-based pattern: read ID, configure, register IIO channels for `IIO_MAGN` with X/Y/Z modifiers.

> **MCU bridge:** Think of regmap like a typed wrapper around your read_reg() and write_reg() helpers, with caching, locking, and bus differences handled centrally. **IIO:** Industrial I/O, Linux's subsystem for sensors, ADCs, DACs, and buffered sampled data. **regmap:** a kernel helper that wraps register reads and writes over I2C, SPI, or MMIO.

`drivers/iio/magnetometer/qmc5883.c` is its own driver. Can't share with HMC due to the register-set incompatibility.

`drivers/iio/magnetometer/mmc56x3.c` covers MMC5983MA and MMC56x3 family.

All three expose:

```
in_magn_x_raw
in_magn_y_raw
in_magn_z_raw
in_magn_scale            ← LSB to Gauss (or µT) conversion
sampling_frequency
sampling_frequency_available
```

User-space reads `raw × scale` to get the field in physical units. The driver doesn't do calibration, that's user-space's job.

73.6 Writing a QMC5883L driver from scratch

We'll target QMC5883L since it's what most people actually have. ~180 lines.

`myqmc5883.c`:

```
#include <linux/init.h>
#include <linux/module.h>
#include <linux/i2c.h>
#include <linux/delay.h>
#include <linux/iio/iio.h>

#define REG_DATA      0x00
#define REG_STATUS    0x06
#define REG_CONTROL1  0x09
#define REG_CONTROL2  0x0A
#define REG_PERIOD    0x0B

struct myqmc {
    struct i2c_client *client;
    struct mutex lock;
};

static int mq_write(struct myqmc *m, u8 reg, u8 val)
{
    return i2c_smbus_write_byte_data(m->client, reg, val);
}

static int mq_read_block(struct myqmc *m, u8 reg, u8 *buf, int n)
```

```
{
    int r = i2c_smbus_read_i2c_block_data(m->client, reg, n, buf);
    return r == n ? 0 : (r < 0 ? r : -EIO);
}

static int mq_init(struct myqmc *m)
{
    int err;
    /* Soft reset */
    err = mq_write(m, REG_CONTROL2, 0x80);
    if (err) return err;
    msleep(20);
    /* Period register — mandatory magic value */
    err = mq_write(m, REG_PERIOD, 0x01);
    if (err) return err;
    /* CONTROL1: OSR=512(00), RNG=±8G(10), ODR=200Hz(11), MODE=continuous(01)
       binary: 0001_1101 = 0x1D */
    err = mq_write(m, REG_CONTROL1, 0x1D);
    if (err) return err;
    return 0;
}

static int mq_read_axes(struct myqmc *m, s16 axes[3])
{
    u8 buf[6];
    int err;

    err = mq_read_block(m, REG_DATA, buf, 6);
    if (err) return err;
    /* QMC: little-endian per axis */
    axes[0] = (s16)(buf[0] | (buf[1] << 8));   /* X */
    axes[1] = (s16)(buf[2] | (buf[3] << 8));   /* Y */
    axes[2] = (s16)(buf[4] | (buf[5] << 8));   /* Z */
    return 0;
}

static int mq_read_raw(struct iio_dev *idev,
                       struct iio_chan_spec const *chan,
                       int *val, int *val2, long mask)
{
    struct myqmc *m = iio_priv(idev);
    s16 axes[3];
    int err;

    switch (mask) {
    case IIO_CHAN_INFO_RAW:
        mutex_lock(&m->lock);
        err = mq_read_axes(m, axes);
        mutex_unlock(&m->lock);
        if (err) return err;
        switch (chan->channel2) {
        case IIO_MOD_X: *val = axes[0]; return IIO_VAL_INT;
        case IIO_MOD_Y: *val = axes[1]; return IIO_VAL_INT;
        case IIO_MOD_Z: *val = axes[2]; return IIO_VAL_INT;
        }
        return -EINVAL;
    case IIO_CHAN_INFO_SCALE:
        /* ±8 G range, 16-bit signed ⇒ 32768 LSB / 8 G = 4096 LSB/G
         * Convert to Tesla: 1 G = 100 µT, so 1 LSB = 100/4096 µT */
```

```
        *val = 0; *val2 = 24414;     /* ~24.4 nT/LSB */
        return IIO_VAL_INT_PLUS_NANO;
    }
    return -EINVAL;
}

#define MAGN_CH(axis) {                                              \
    .type = IIO_MAGN, .modified = 1, .channel2 = (axis),              \
    .info_mask_separate = BIT(IIO_CHAN_INFO_RAW),                     \
    .info_mask_shared_by_type = BIT(IIO_CHAN_INFO_SCALE),             \
    .scan_index = 0,                                                  \
    .scan_type = { .sign='s', .realbits=16, .storagebits=16,          \
                   .endianness=IIO_LE },                               \
}

static const struct iio_chan_spec mq_channels[] = {
    MAGN_CH(IIO_MOD_X),
    MAGN_CH(IIO_MOD_Y),
    MAGN_CH(IIO_MOD_Z),
};

static const struct iio_info mq_iio_info = {
    .read_raw = mq_read_raw,
};

static int mq_probe(struct i2c_client *client)
{
    struct iio_dev *idev;
    struct myqmc *m;
    int err;

    idev = devm_iio_device_alloc(&client->dev, sizeof(*m));
    if (!idev) return -ENOMEM;
    m = iio_priv(idev);
    m->client = client;
    mutex_init(&m->lock);

    err = mq_init(m);
    if (err) return dev_err_probe(&client->dev, err, "init failed\n");

    idev->name = "myqmc5883";
    idev->info = &mq_iio_info;
    idev->modes = INDIO_DIRECT_MODE;
    idev->channels = mq_channels;
    idev->num_channels = ARRAY_SIZE(mq_channels);

    return devm_iio_device_register(&client->dev, idev);
}

static const struct of_device_id mq_of_match[] = {
    { .compatible = "linuxlearn,myqmc5883" },
    { }
};
MODULE_DEVICE_TABLE(of, mq_of_match);

static const struct i2c_device_id mq_id[] = { { "myqmc5883", 0 }, { } };
MODULE_DEVICE_TABLE(i2c, mq_id);

static struct i2c_driver mq_driver = {
```

```
    .driver = {
        .name = "myqmc5883",
        .of_match_table = mq_of_match,
    },
    .probe = mq_probe,
    .id_table = mq_id,
};
module_i2c_driver(mq_driver);

MODULE_LICENSE("GPL");
```

DT:

```
&i2c1 {
    qmc5883@d {
        compatible = "linuxlearn,myqmc5883";
        reg = <0x0d>;
    };
};
```

Test:

```
[root@pa-mini:~]# insmod myqmc5883.ko
[root@pa-mini:~]# cat /sys/bus/iio/devices/iio:device0/in_magn_x_raw
1042
[root@pa-mini:~]# cat /sys/bus/iio/devices/iio:device0/in_magn_y_raw
-832
[root@pa-mini:~]# cat /sys/bus/iio/devices/iio:device0/in_magn_z_raw
-1620

# Calibrated heading (raw, no soft/hard-iron correction!):
[root@pa-mini:~]# awk "BEGIN { print atan2(-832, 1042) * 180 / 3.14159 }"
-38.6      ← "compass says 38.6° west of north"... but it's wrong because no calibration
```

73.7 Calibration, the part most products skip

A user-space calibration script collects samples while you slowly rotate the sensor in all 3D orientations (cover an imaginary sphere). The samples fall on an ellipsoid. From the ellipsoid you compute two things:

1. The **offset** (center of the ellipsoid), hard-iron correction.
2. The **3x3 matrix** to rotate-and-scale the ellipsoid into a sphere, soft-iron correction.

A simple Python sketch:

```
#!/usr/bin/env python3
import numpy as np, time, sys

def read_axis(axis):
    return int(open(f"/sys/bus/iio/devices/iio:device0/in_magn_{axis}_raw").read())
```

```
# Phase 1: collect ~1000 samples while the user rotates the sensor
samples = []
print("Rotate the sensor in all directions for 30 seconds...")
t0 = time.time()
while time.time() - t0 < 30:
    samples.append([read_axis(a) for a in "xyz"])
    time.sleep(0.03)
samples = np.array(samples, dtype=float)

# Phase 2: fit ellipsoid → extract center (hard-iron) and matrix (soft-iron)
# Simplified: just center + scale-axis-by-axis (ignores rotation)
center = (samples.max(axis=0) + samples.min(axis=0)) / 2
ranges = (samples.max(axis=0) - samples.min(axis=0)) / 2
avg_range = ranges.mean()
scale = avg_range / ranges    # per-axis scale

# Phase 3: save calibration
np.savez("magcal.npz", center=center, scale=scale)
print(f"Hard-iron offset: {center}")
print(f"Soft-iron scale:  {scale}")
```

Apply at runtime:

```
cal = np.load("magcal.npz")
def read_calibrated():
    raw = np.array([read_axis(a) for a in "xyz"])
    return cal["scale"] * (raw - cal["center"])

import math
m = read_calibrated()
heading_deg = math.atan2(m[1], m[0]) * 180 / math.pi
if heading_deg < 0: heading_deg += 360
print(f"Heading: {heading_deg:.1f}°")
```

This is the *simplified* calibration. The proper version fits a full ellipsoid model (10 parameters: center 3 + axes 3 + rotation 3 + radius 1) using least-squares. See `Calibration of triaxial magnetometers` literature for the math. Libraries like **MotionCal** (Adafruit's GUI tool) do this. The resulting offset + 3x3 matrix is portable.

After proper calibration: heading error < 1°, indoor or out.

73.8 Mainline driver enablement

DT for QMC5883L:

```
&i2c1 {
    qmc5883@d {
        compatible = "qst,qmc5883l";
        reg = <0x0d>;
    };
};
```

For HMC5883L: `compatible = "honeywell,hmc5883l"; reg = <0x1e>;`.

For MMC5983MA: `compatible = "memsic,mmc5983ma"; reg = <0x30>;`.

The mainline drivers expose richer sysfs:

> **sysfs:** a kernel-generated filesystem under /sys that exposes devices, drivers, and attributes.

```
[root@pa-mini:~]# ls /sys/bus/iio/devices/iio:device0/
in_magn_scale
in_magn_scale_available
in_magn_x_raw
in_magn_y_raw
in_magn_z_raw
sampling_frequency
sampling_frequency_available
```

Plus buffered capture via trigger (Ch 70) for high-rate logging.

73.9 Key Notes

- A magnetometer measures a field vector, not an already-correct compass heading. Scale, axis orientation and calibration belong in the interpretation path.
- Hard-iron offset and soft-iron distortion are different effects. Calibration must represent the installed assembly and its operating environment.
- A heading derived from only two axes assumes a level sensor. A tilting product needs a defined coordinate frame and tilt compensation.
- Chip identity and register layout must be verified independently of breakout-board branding. A responding address is not sufficient evidence of software compatibility.

73.10 Pitfalls

- **Mistaking QMC5883L for HMC5883L.** #1 hobbyist trap. If it doesn’t respond at 0x1E, try 0x0D.
- **No calibration.** Heading off by 10–30°. Calibrate, every time, in-place.
- **Calibration baked into firmware but environment changed.** A magnetometer on a robot calibrated on workbench, then mounted near a motor: recalibrate on the robot.
- **XY-only heading assumes the sensor is held level.** If the device can tilt, combine the magnetometer with the accelerometer and project the magnetic vector onto the horizontal plane. This is “tilt compensation” and it is mandatory for any device a user holds in their hand.

- **Buck converters within 5 cm.** Switching power supplies emit strong RF magnetic noise. Magnetometer reads garbage. Layout: keep mag far from switchers, or filter heavily.
- **Iron rich PCB substrate.** Cheap PCBs sometimes have ferromagnetic impurities. Affects calibration repeatability across boards.
- **Phone case magnets.** A 30-cm gap to a magnet still measures 100s of µT. Test your product in its real-world envelope.
- **Sensor saturation.** A strong nearby field (relay coil during switch) can saturate the magnetometer. Readings stick at ±MAX for a while afterward. Verify range is wide enough.
- **Slow rotation during calibration.** Need to rotate slowly enough that samples cover the sphere uniformly. Too fast = patches uncovered = poor ellipsoid fit.

73.11 Going deeper

- **`drivers/iio/magnetometer/hmc5843_core.c`** + `hmc5843_i2c.c`, HMC family.
- **`drivers/iio/magnetometer/qmc5883.c`**: QMC.
- **`drivers/iio/magnetometer/mmc56x3.c`**: MMC.
- **HMC5883L datasheet (Honeywell)**: register layout.
- **QMC5883L datasheet (QST)**: for comparison with HMC. Note the differences.
- **MMC5983MA datasheet (Memsic)**: set-reset cycle (a calibration improvement specific to MMC).
- **“Markovsky-Van Huffel ellipsoid fitting”**: the math behind proper soft-iron calibration.
- **MotionCal tool** by Adafruit, interactive calibration capture + analysis.
- **NOAA WMM (World Magnetic Model)**: for converting magnetic-north to true-north (declination varies by location and year).

> Next chapter: **Chapter 74: Hall-effect & rotary position sensors.** Magnetic sensing applied to mechanical position, AS5048 (absolute rotary), A1324 (linear Hall), TLE5012 (high-rate angular).

Chapter 74: Hall-effect & rotary position sensors

What: three Hall-effect-based position sensors at different abstraction levels: **AMS AS5048A** (SPI, 14-bit absolute rotary, “magnet-on-axis” encoder), **Allegro A1324** (analog linear Hall sensor), **Infineon TLE5012B** (SPI, high-rate, dual-die for safety-critical motor control). For each: physics, protocol, mainline driver, plus a from-scratch SPI driver for AS5048A.

Why: measuring rotary position without mechanical contact is the foundation of brushless motor control, robotic joints, throttle position sensors, steering angle, knob inputs on appliances. Hall-on-magnet sensors replace optical encoders. They cost less, last longer (no slip-rings, no aging photo-emitters), and tolerate oil and dust.

Focus: The magnet matters as much as the chip, get it wrong and the chip reads garbage. AS5048 needs a *diametrically magnetised* 2-pole magnet, axially mounted, 0.5–3 mm above the chip die. Wrong magnet, wrong distance, wrong polarisation = the chip reports nonsense or a low-resolution mess. Most “AS5048 doesn’t work” threads online trace back to the wrong magnet.

74.1 Sensor comparison

	AMS AS5048A	Allegro A1324	Infineon TLE5012B
Output	14-bit absolute angle	analog 0–3 V	15-bit absolute angle
Interface	SPI (also I²C version: AS5048B)	analog	SPI (SSC mode)
Update rate	11.25 kHz	continuous (analog)	42 µs latency (24 kHz)
Magnet	diametric, on-axis	linear flux	diametric, on-axis
Magnet distance	0.5, 3 mm	physical contact / very close	0.5, 3 mm

	AMS AS5048A	Allegro A1324	Infineon TLE5012B
Programmable zero	yes	no	yes
ABI/UVW output	yes	no	yes
Safety features	none	none	dual-die for ASIL-B
Volume price	$3–6	$0.50–1.50	$7–12

Pick guide:

- **AS5048A**: general motor control, joystick/knob, robotics. The general-purpose choice.
- **A1324**: simple “is metal nearby?” or linear-distance from a magnet. ADC-based.
- **TLE5012B**: automotive / safety-rated motor control. Has redundancy.

74.2 The physics, Hall and magnetoresistive

A Hall sensor outputs a voltage proportional to the magnetic-field component perpendicular to its die. Rotate a 2-pole magnet near the chip: the field above the chip’s center rotates with the magnet. Two perpendicular Hall sensors on the same die (at 0° and 90°) give sin(θ) and cos(θ), perfect for `atan2()` to extract angle.

AS5048 and TLE5012 actually use **giant magnetoresistance** (GMR) elements arranged in Wheatstone bridges, slightly different from raw Hall but the same principle: cos/sin → angle.

The **magnet** must be:

- **Diametrically magnetised**: poles on opposite sides of the cylinder, not top/bottom. This makes the field rotate with the magnet.
- **On-axis**: the chip’s center aligned with the magnet’s rotational axis (within ±0.5 mm).
- **At the right distance**: 0.5–3 mm gap. Too close = chip saturates. Too far = noise dominates.

Common Chinese eBay “AS5048A modules” ship without a magnet or with the wrong (axial) magnet. Buy a small (6 mm × 2.5 mm) **diametrically-magnetised** disc-magnet separately.

74.3 Protocol, AS5048A SPI

AS5048A uses SPI with a peculiar **command-then-result** sequence. Each SPI frame is 16 bits:

```
   bit 15:    parity (even parity over remaining 15 bits)
   bit 14:    R/W (1 = read, 0 = write)
   bits 13:0: register address (or data)
```

You send a *command frame* in the current transaction. The matching *response* arrives in the next transaction.

To read the angle register (0x3FFF):

```
   Transaction 1:
       Host → chip: 0xFFFF  (parity=1, R=1, addr=0x3FFF)
       Chip → host: (irrelevant — whatever's in the chip's response buffer)

   Transaction 2:
       Host → chip: 0x0000  (NOP — or queue next command)
       Chip → host: 0x_AAAA where AAAA = angle data
```

The response's bit layout:

```
   bit 15:    parity
   bit 14:    error flag (1 = error, read 0x4001 to clear)
   bits 13:0: 14-bit angle (0..16383)
```

Convert to degrees: `angle_deg = (raw / 16384.0) * 360.0`.

Parity check

Even parity over bits 14:0. The chip rejects frames with bad parity. Implementation:

```
static u16 as5048_parity(u16 v)
{
    v ^= v >> 8;
    v ^= v >> 4;
    v ^= v >> 2;
    v ^= v >> 1;
    return v & 1;
}

u16 frame_for_read(u16 addr)
{
    u16 cmd = (1 << 14) | addr;   /* R=1 */
    if (as5048_parity(cmd)) cmd |= (1 << 15);   /* set parity to make even */
    return cmd;
}
```

Key registers

Reg	Name	Purpose
0x0000	NOP	dummy frame
0x0001	Clear error flag	reading returns error code, clears flag
0x0003	Programming control	for OTP burn (one-time-programmable)
0x0016	OTP zero position high	8 bits
0x0017	OTP zero position low	6 bits
0x3FFD	Diagnostics + AGC	magnet too weak / too strong / etc.
0x3FFE	Magnitude	strength of field (sanity check)
0x3FFF	Angle	the answer

74.4 Writing an AS5048A driver from scratch

`myas5048.c`:

```
#include <linux/init.h>
#include <linux/module.h>
#include <linux/spi/spi.h>
#include <linux/iio/iio.h>

#define REG_ANGLE       0x3FFF
#define REG_DIAG        0x3FFD
#define REG_MAGNITUDE   0x3FFE
#define REG_CLEAR_ERR   0x0001

struct myas {
    struct spi_device *spi;
    struct mutex lock;
};

static u16 parity_even(u16 v)
```

```
{
    v ^= v >> 8; v ^= v >> 4; v ^= v >> 2; v ^= v >> 1;
    return v & 1;
}

static u16 build_read_cmd(u16 addr)
{
    u16 cmd = (1 << 14) | (addr & 0x3FFF);
    if (parity_even(cmd)) cmd |= 1 << 15;
    return cmd;
}

/* Two-frame transaction: send cmd this frame, get result in next */
static int ma_read_reg(struct myas *m, u16 addr, u16 *out)
{
    u16 cmd = build_read_cmd(addr);
    u16 nop = build_read_cmd(0x0000);
    u8 tx[4], rx[4];
    struct spi_transfer xfer = {
        .tx_buf = tx,
        .rx_buf = rx,
        .len = 4,
    };
    int err;

    /* Big-endian on the wire */
    tx[0] = cmd >> 8; tx[1] = cmd & 0xFF;
    tx[2] = nop >> 8; tx[3] = nop & 0xFF;

    err = spi_sync_transfer(m->spi, &xfer, 1);
    if (err) return err;

    /* Result is in the second 16 bits */
    u16 result = (rx[2] << 8) | rx[3];

    /* Check parity (we don't enforce, just log) */
    if (parity_even(result))
        pr_debug("myas5048: bad parity on read\n");

    /* Check error flag */
    if (result & (1 << 14)) {
        pr_warn("myas5048: error flag set\n");
        /* Issue a clear-error transaction */
    }

    *out = result & 0x3FFF;
    return 0;
}

static int ma_read_raw(struct iio_dev *idev,
                       struct iio_chan_spec const *chan,
                       int *val, int *val2, long mask)
{
    struct myas *m = iio_priv(idev);
    u16 raw;
    int err;

    if (chan->type != IIO_ANGL) return -EINVAL;
```

```
    switch (mask) {
    case IIO_CHAN_INFO_RAW:
        mutex_lock(&m->lock);
        err = ma_read_reg(m, REG_ANGLE, &raw);
        mutex_unlock(&m->lock);
        if (err) return err;
        *val = raw;
        return IIO_VAL_INT;
    case IIO_CHAN_INFO_SCALE:
        /* 14-bit = 16384 LSB per full turn = 2π rad
         * 1 LSB = 2π / 16384 ≈ 383.5 µrad */
        *val = 0; *val2 = 383495;     /* nano-radians per LSB */
        return IIO_VAL_INT_PLUS_NANO;
    }
    return -EINVAL;
}

static const struct iio_chan_spec ma_channels[] = {
    {
        .type = IIO_ANGL,
        .info_mask_separate = BIT(IIO_CHAN_INFO_RAW),
        .info_mask_shared_by_type = BIT(IIO_CHAN_INFO_SCALE),
    },
};

static const struct iio_info ma_iio_info = {
    .read_raw = ma_read_raw,
};

static int ma_probe(struct spi_device *spi)
{
    struct iio_dev *idev;
    struct myas *m;
    u16 magnitude;
    int err;

    idev = devm_iio_device_alloc(&spi->dev, sizeof(*m));
    if (!idev) return -ENOMEM;
    m = iio_priv(idev);
    m->spi = spi;
    mutex_init(&m->lock);

    spi->mode = SPI_MODE_1;     /* CPOL=0, CPHA=1 per AS5048 datasheet */
    spi->bits_per_word = 8;
    err = spi_setup(spi);
    if (err) return dev_err_probe(&spi->dev, err, "spi_setup failed\n");

    /* Sanity-check magnet by reading magnitude */
    err = ma_read_reg(m, REG_MAGNITUDE, &magnitude);
    if (err) return err;
    /* Read again to get the actual answer (first frame is throwaway) */
    err = ma_read_reg(m, REG_MAGNITUDE, &magnitude);
    if (err) return err;
    dev_info(&spi->dev, "AS5048 magnitude: %u (typical 5000-6000 with good magnet)\n",
             magnitude);
    if (magnitude < 1000)
        dev_warn(&spi->dev, "weak magnet — check distance/orientation\n");

    idev->name = "myas5048a";
```

```
    idev->info = &ma_iio_info;
    idev->modes = INDIO_DIRECT_MODE;
    idev->channels = ma_channels;
    idev->num_channels = ARRAY_SIZE(ma_channels);

    return devm_iio_device_register(&spi->dev, idev);
}

static const struct of_device_id ma_of_match[] = {
    { .compatible = "linuxlearn,myas5048a" },
    { }
};
MODULE_DEVICE_TABLE(of, ma_of_match);

static const struct spi_device_id ma_id[] = { { "myas5048a", 0 }, { } };
MODULE_DEVICE_TABLE(spi, ma_id);

static struct spi_driver ma_driver = {
    .driver = {
        .name = "myas5048a",
        .of_match_table = ma_of_match,
    },
    .probe = ma_probe,
    .id_table = ma_id,
};
module_spi_driver(ma_driver);

MODULE_LICENSE("GPL");
```

DT:

```
&ecspi3 {
    as5048a@0 {
        compatible = "linuxlearn,myas5048a";
        reg = <0>;
        spi-max-frequency = <10000000>;
        spi-cpha;                       /* mode 1 */
    };
};
```

Test:

```
[root@pa-mini:~]# insmod myas5048a.ko
[root@pa-mini:~]# dmesg | tail -2
myas5048a spi3.0: AS5048 magnitude: 5482 (typical 5000-6000 with good magnet)

[root@pa-mini:~]# cat /sys/bus/iio/devices/iio:device0/in_angl_raw
8192       ← 180° (8192 / 16384 × 360°)

[root@pa-mini:~]# # Rotate the magnet 90°:
[root@pa-mini:~]# cat /sys/bus/iio/devices/iio:device0/in_angl_raw
12288     ← 270°

[root@pa-mini:~]# # In radians:
[root@pa-mini:~]# awk "BEGIN { print 12288 * 0.000383495 }"
4.71239    ← 270° = 3π/2 rad ≈ 4.71
```

Driver is ~150 lines. The 14-bit absolute angle is in IIO, ready for any application that needs it.

> **IIO:** Industrial I/O, Linux's subsystem for sensors, ADCs, DACs, and buffered sampled data.

74.5 A1324, analog linear Hall

Allegro A1324 is a 3-pin part: VCC, GND, OUT. Output voltage:

```
V_OUT = V_OUT_Q + sensitivity × B
       (V_OUT_Q ≈ VCC/2 at zero field; sensitivity ≈ 5 mV/Gauss)
```

To read: wire to ADC (Ch 80 or i.MX6ULL internal ADC). The "driver" is just the ADC driver. User-space converts voltage to Gauss (or position, given a known magnet's field-vs-distance curve).

For "is there a magnet nearby?" (lid open/closed, latch position): A1324 + ADC + a threshold is enough.

74.6 TLE5012B, the safety variant

> **Driver choice:** Use the in-tree, maintained driver first. Use out-of-tree, spidev, or custom-driver paths only after you accept the kernel-version maintenance cost and document who owns updates.

TLE5012B is functionally similar to AS5048 but with two independent dies on the same package. Each gets its own SPI access. You cross-check the two readings. Any divergence indicates fault. ASIL-B (SIL-2) capable.

Mainline driver: `drivers/iio/position/iqs62x.c` covers some Iqs sensors. TLE5012 has out-of-tree drivers from Infineon. The protocol is more elaborate (16-bit SPI with a "SSC" auto-mode option for continuous streaming).

For non-safety-critical use, AS5048 is the cheaper, equally-accurate choice.

74.7 Mainline driver enablement

`drivers/iio/position/as5048.c` is the mainline AS5048 driver (covers AS5048A SPI and AS5048B I²C variants).

DT:

```
&ecspi3 {
    as5048a@0 {
        compatible = "ams,as5048a";
        reg = <0>;
        spi-max-frequency = <10000000>;
        spi-cpha;
    };
};
```

After load: `/sys/bus/iio/devices/iio:device0/in_angl_raw` plus `_scale` for converting to radians.

74.8 Key Notes

- Rotary magnetic sensing depends on magnet type, alignment and field strength as well as electronics. Mechanical geometry is part of the measurement design.
- Absolute angle wraps at one revolution. Position, velocity and accumulated turns require different state and interpretation above that value.
- The AS5048A's command and response span successive frames. The driver must preserve that relationship rather than associate a reply with the wrong request.
- Parity and diagnostic status help identify invalid data, but do not establish a qualified motor-control or safety system.

74.9 Pitfalls

- **Wrong magnet.** Axial magnetisation = chip sees constant field, no rotation signal. Diametric is mandatory.
- **Magnet off-axis.** Even 0.5 mm off-axis adds significant non-linearity (>1° error). Mechanical fixturing matters.
- **Magnet too close.** Chip saturates. Angle clamps or wraps. The datasheet specifies a typical air gap of 1–3 mm.
- **Magnet too far / weak.** Magnitude register low. Angle is noisy. Use a stronger magnet or move closer.
- **SPI mode wrong.** AS5048A is mode 1 (CPOL=0, CPHA=1). Mode 0 returns 0xFFFF every read.
- **Forgetting the two-frame protocol.** A read result is in the *next* frame. First read returns junk. Second returns the answer.
- **Parity ignored on critical applications.** The chip can return bad data due to bus noise. Parity is your sanity check. Always validate in safety-critical code.

- **Magnetic interference from motor.** If the AS5048 is on the motor's shaft, motor magnets/coils may bleed through. Use shielding or magnetic isolation.
- **Hot-plug / start-up race.** The chip needs about 10 ms after VCC rises. A read earlier returns junk. The mainline driver handles this delay. The from-scratch driver must too.

74.10 Going deeper

- **`drivers/iio/position/as5048.c`**: production AS5048 driver.
- **AS5048A datasheet (AMS)**: SPI protocol, magnet selection appendix, programming OTP.
- **A1324 datasheet (Allegro)**: output transfer function.
- **TLE5012B datasheet (Infineon)**: SSC protocol, safety features.
- **AMS app note AN53048-A1**: magnet selection and mechanical mounting recommendations.
- **`Documentation/ABI/testing/sysfs-bus-iio` (channel type IIO_ANGL)**: angle channel ABI.

> **ABI:** Application Binary Interface: the calling convention, register use, binary format, and library contract that let separately built code run together. **sysfs:** a kernel-generated filesystem under /sys that exposes devices, drivers, and attributes.

> **End of Group D, Position & distance (Ch 72–74).** From "how far" (ToF, ultrasonic, IR) to "which way" (magnetometer) to "what angle" (Hall rotary). Three different physics, three different driver shapes.

> Next chapter: **Chapter 75: Current and power monitoring (INA219 / INA226 / INA3221).** Low-side and high-side current measurement, the calibration register that nobody understands at first, and the `hwmon` framework that sometimes coexists with IIO for the same chip.

Chapter 75: Current & power monitoring

> **regmap:** a kernel helper that wraps register reads and writes over I2C, SPI, or MMIO. **MCU bridge:** Think of regmap like a typed wrapper around your read_reg() and write_reg() helpers, with caching, locking, and bus differences handled centrally.

> **What:** three I²C high-side current/voltage monitors from Texas Instruments: **INA219** (12-bit, the classic), **INA226** (16-bit, modern, programmable averaging), **INA3221** (3-channel, simultaneous-sample 3-rail monitor). For each: physics, register map, the *calibration register* that is the most common bug, and a from-scratch INA219 driver. Plus the **hwmon** subsystem, the sibling of IIO that current monitors usually live in. **IIO:** Industrial I/O, Linux's subsystem for sensors, ADCs, DACs, and buffered sampled data.
>
> **Why:** every device that draws power benefits from knowing how much. Production telemetry (per-rail consumption logged to fleet management), fault detection (overcurrent → shutdown), low-power optimisation (which subsystem ate the budget?), battery-life prediction. INA219 in particular costs $1.50 and lets you watch any 0–26 V rail at 1 mA resolution.
>
> **Focus:** The shunt converts current to voltage. The chip's ADC converts voltage to a count. The calibration register tells the chip the shunt's value, so the chip can report current directly. Get the shunt size right (low enough to not waste power. High enough to get good resolution) and program the calibration register to match, the chip then reports current directly in amperes. Without the calibration register, the Current and Power registers read zero (or numbers in unknown units, depending on the chip).

75.1 Chip comparison

	TI INA219	TI INA226	TI INA3221
Channels	1	1	3
Bus voltage range	0–26 V	0–36 V	0–26 V

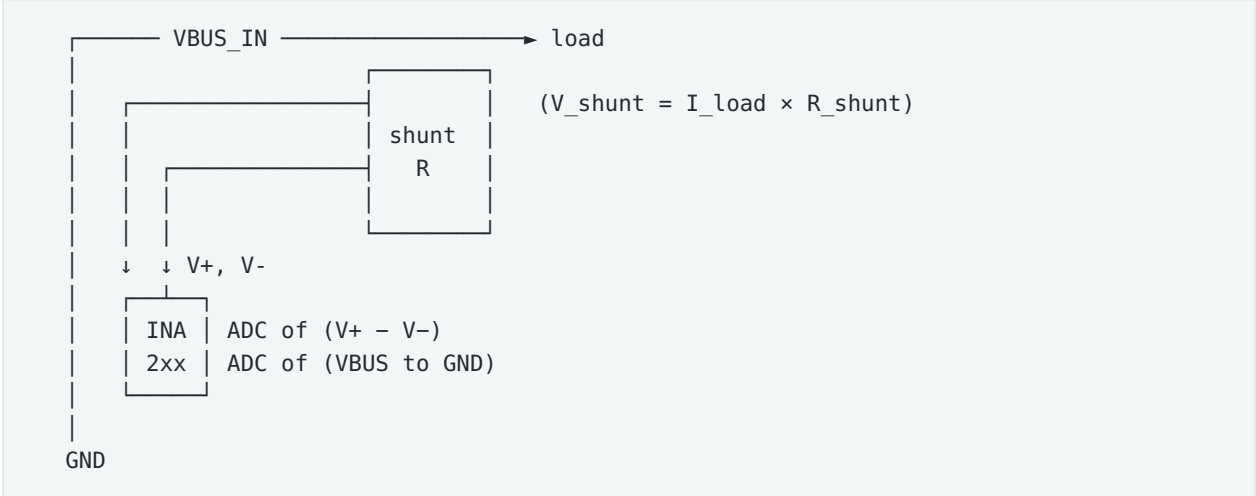
	TI INA219	TI INA226	TI INA3221
Shunt voltage resolution	10 µV (LSB, 12-bit)	2.5 µV (LSB, 16-bit)	40 µV (LSB, 13-bit)
Max shunt voltage	±320 mV	±81.92 mV	±163.8 mV
Bus voltage LSB	4 mV	1.25 mV	8 mV
ADC conversion time	532 µs (12-bit)	8.244 ms (16-bit, 1024 avg)	configurable
Average filter	none	up to 1024-sample average	up to 1024 average per chan
I²C address	0x40 (4 pin-strapped variants → up to 16 chips on one bus)	0x40 (16 variants)	0x40 (4 variants)
I²C clock	100 kHz / 400 kHz / 2.94 MHz (HS)	up to 2.94 MHz	up to 2.94 MHz
Alert pin	none	overcurrent / under/over voltage / power threshold	one per channel
Volume price	$1.50–2.50	$3–5	$4–6
Linux driver	`hwmon/ina2xx.c`	`hwmon/ina2xx.c`	`hwmon/ina3221.c`

Pick guide:

- **INA219**: cheap, fast (~500 µs), one rail. 1 mA resolution typically.
- **INA226**: better noise, programmable averaging, alert pin. Power-budget profiling.
- **INA3221**: 3 rails at once with their own alerts. Multi-rail system telemetry.

75.2 The physics, shunt + amplifier

You measure current with a low-value precision resistor (the **shunt**) in series with the load:



The INA's two inputs straddle the shunt. The chip's ADC measures the differential voltage. A separate ADC measures the bus voltage relative to GND.

Shunt selection is the design decision:

- Too small (< 0.01 Ω): poor resolution. Noise limits accuracy.
- Too large (> 0.5 Ω): wastes power. Voltage drop on the rail.
- Sweet spot: choose `R_shunt = V_shunt_max / I_max` where V_shunt_max ≈ 50 mV (well below INA's 320 mV max).

Example: monitoring a rail expected to draw up to 2 A. R_shunt = 50 mV / 2 A = 25 mΩ. Power dissipation: I² × R = 4 × 0.025 = 100 mW (use a 1/4 W resistor with margin). Resolution: 10 µV / 25 mΩ = 0.4 mA per LSB.

75.3 Protocol, INA219

Register map:

Reg	Name	Purpose
0x00	Configuration	bus range, gain, ADC resolution, mode
0x01	Shunt Voltage	signed 16-bit, LSB = 10 µV

Reg	Name	Purpose
0x02	Bus Voltage	bits 15:3 = bus voltage / 4 mV; bit 1 = CNVR; bit 0 = OVF
0x03	Power	calculated by chip = current × bus_voltage (LSB = 20 × current_LSB)
0x04	Current	calculated by chip = shunt_voltage × CAL >> 12
0x05	Calibration	the magic register

Each register is 16 bits, **big-endian** on the wire. A read of register 0x02 looks like:

```
   Host: START | 0x80 | 0x02 | START | 0x81 | (2 bytes MSB,LSB) | STOP
```

(0x40 << 1 = 0x80 for write. 0x40 << 1 | 1 = 0x81 for read.)

The calibration register, finally explained

The chip's internal `Current` register doesn't measure current directly. It computes:

```
Current_register = (Shunt_voltage × Calibration_register) / 4096
```

You program the Calibration register with a value that makes Current_register read out in your chosen units (mA, 100 µA, etc.).

The formula (from datasheet):

```
Calibration = trunc(0.04096 / (Current_LSB × R_shunt))
```

Where:

- `Current_LSB` is the unit you want for the current reading. E.g., 0.0001 A/LSB = 100 µA per count.
- `R_shunt` is in ohms.

Example: 25 mΩ shunt, want 100 µA/LSB.

```
Cal = trunc(0.04096 / (0.0001 × 0.025))
    = trunc(0.04096 / 0.0000025)
```

```
    = trunc(16384)
    = 16384
```

Write 16384 (0x4000) to register 0x05. Now `Current_register` reads in units of 100 µA. A reading of 1234 → 1234 × 100 µA = 123.4 mA.

Power register (0x03) auto-computes as `(Current × BusVoltage) >> 11` with LSB = `20 × Current_LSB`. If `Current_LSB = 100 µA`, `Power_LSB = 2 mW`.

The fixed `4096` and `0.04096` and `2048` (for power) come from the ADC's internal scaling, they're not adjustable, they're physical constants of the chip's design. The Calibration register is just a multiplier that maps "raw shunt voltage" to "current in your units."

This is the part most people miss on the first try. Without programming Calibration:

- The Shunt Voltage register *does* work, it reads in 10 µV units regardless.
- The Bus Voltage register works.
- The Current register reads zero (Cal = 0 ⇒ Current = 0).
- The Power register reads zero.

So a "first-light" sanity check that just reads Shunt Voltage will work fine and seem to confirm everything… And then you wonder why Current reads zero.

75.4 How the mainline `ina2xx` driver works

Source: `drivers/hwmon/ina2xx.c` (~700 lines). Covers INA219, INA220, INA226, INA230, INA231, the whole INA family.

The driver auto-detects which chip based on `compatible` strings, looks up its parameters from a per-chip `ina2xx_config` table, and registers an hwmon device. The hwmon framework exposes `/sys/class/hwmon/hwmon0/in0_input`, `curr1_input`, `power1_input`, same shape across all hwmon-class drivers.

```
/* Simplified */
struct ina2xx_config {
    u16  config_default;
    int  calibration_value;   /* default; user can override via shunt-resistor + current-
lsb */
    int  registers;            /* number of valid registers */
    int  shunt_div;            /* divisor mapping raw shunt voltage to µV */
    int  bus_voltage_shift;    /* shift for bus-voltage register */
    int  bus_voltage_lsb;      /* µV/LSB */
    int  power_lsb_factor;    /* multiplier to current_lsb */
};

static const struct ina2xx_config ina2xx_config[] = {
    [ina219] = {
        .config_default     = 0x399F,         /* 32 V, ±320 mV, 12-bit, continuous */
        .calibration_value  = 4096,
```

```
        .shunt_div          = 100,            /* 10 µV/LSB ⇒ multiply by 10 to get µV  */
        .bus_voltage_shift  = 3,
        .bus_voltage_lsb    = 4000,           /* 4 mV/LSB */
        .power_lsb_factor   = 20,             /* power_lsb = 20 × current_lsb */
    },
    [ina226] = {
        .config_default     = 0x4127,
        .calibration_value  = 2048,
        .shunt_div          = 400,            /* 2.5 µV/LSB */
        .bus_voltage_shift  = 0,
        .bus_voltage_lsb    = 1250,
        .power_lsb_factor   = 25,
    },
};

static int ina2xx_init(struct ina2xx_data *data, struct device *dev)
{
    int err;

    /* Write configuration register (set ranges, gain, conversion mode) */
    err = regmap_write(data->regmap, INA2XX_CONFIG, data->config->config_default);

    /* Compute calibration based on user-supplied shunt resistor */
    /* shunt_resistor in DT is in µΩ; current_lsb auto-computed */
    u64 cal = data->config->calibration_value * 1000000;
    do_div(cal, data->shunt_uohms);
    /* cal now has dimension that matches what's required */

    err = regmap_write(data->regmap, INA2XX_CALIBRATION, cal);
    return 0;
}

static ssize_t ina2xx_show(struct device *dev, struct device_attribute *attr, char *buf)
{
    struct ina2xx_data *data = dev_get_drvdata(dev);
    struct sensor_device_attribute *sda = to_sensor_dev_attr(attr);
    int reg = sda->index;
    unsigned int raw;
    int val;

    regmap_read(data->regmap, reg, &raw);
    val = ina2xx_get_value(data, reg, raw);    /* scale by per-register factor */
    return sprintf(buf, "%d\n", val);
}

/* Attribute table */
static SENSOR_DEVICE_ATTR_RO(in0_input,   ina2xx, INA2XX_SHUNT_VOLTAGE);
static SENSOR_DEVICE_ATTR_RO(in1_input,   ina2xx, INA2XX_BUS_VOLTAGE);
static SENSOR_DEVICE_ATTR_RO(curr1_input, ina2xx, INA2XX_CURRENT);
static SENSOR_DEVICE_ATTR_RO(power1_input, ina2xx, INA2XX_POWER);
```

The hwmon framework’s standard attribute names: `in*_input` = voltage in millivolts, `curr*_input` = current in milliamps, `power*_input` = power in microwatts, `temp*_input` = temperature in milli-degrees. User-space tools (`lm-sensors`, `sensors`, Grafana exporters) all consume this conventional naming.

75.5 Writing an INA219 driver from scratch

We'll write a from-scratch driver that follows the hwmon convention (not IIO this time, hwmon is the canonical home for power monitoring). ~250 lines.

`myina219.c`:

```
#include <linux/init.h>
#include <linux/module.h>
#include <linux/i2c.h>
#include <linux/hwmon.h>
#include <linux/hwmon-sysfs.h>
#include <linux/of.h>

#define REG_CONFIG       0x00
#define REG_SHUNT_V      0x01
#define REG_BUS_V        0x02
#define REG_POWER        0x03
#define REG_CURRENT      0x04
#define REG_CALIBRATION  0x05

#define CFG_DEFAULT      0x399F     /* 32V range, ±320mV, 12-bit, continuous */

struct myina {
    struct i2c_client *client;
    struct mutex lock;
    u32 shunt_uohms;       /* shunt resistor in micro-ohms (from DT) */
    s32 current_lsb_uA;    /* µA per LSB */
};

/* === Low-level: 16-bit big-endian I²C registers === */

static int mi_read16(struct myina *m, u8 reg, s16 *out)
{
    int err;
    err = i2c_smbus_read_word_swapped(m->client, reg);
    if (err < 0) return err;
    *out = (s16)err;
    return 0;
}

static int mi_write16(struct myina *m, u8 reg, u16 val)
{
    return i2c_smbus_write_word_swapped(m->client, reg, val);
}

/* === Bring-up: program config + calibration === */

static int mi_init(struct myina *m)
{
    int err;
    u32 cal;

    /* Config register: 32 V range, gain /8 (±320 mV), 12-bit, continuous */
    err = mi_write16(m, REG_CONFIG, CFG_DEFAULT);
    if (err) return err;
```

```
    /* Auto-compute calibration:
       Cal = 0.04096 / (Current_LSB × R_shunt)
       Pick Current_LSB so the chip's 16-bit register covers our expected max:
       For 2 A max: Current_LSB = 2 A / 32767 ≈ 61 µA, round up to 100 µA
       Then Cal = 0.04096 / (100e-6 × shunt_ohms)
                = 0.04096 / (current_lsb_uA × 1e-6 × shunt_uohms × 1e-6)
                = 0.04096 / (current_lsb_uA × shunt_uohms × 1e-12)
                = 0.04096 × 1e12 / (current_lsb_uA × shunt_uohms)
                = 40960000 / (current_lsb_uA × shunt_uohms)  (need integer)

       More carefully: Cal = trunc(40960000 × 1000 / (current_lsb_uA × shunt_uohms))
       ... after working through units cleanly with the mainline algorithm. */
    m->current_lsb_uA = 100;     /* 100 µA per LSB — works for shunts up to a few hundred
mΩ */
    cal = 40960000u / (m->current_lsb_uA * (m->shunt_uohms / 1000));
    /* Simpler: with shunt = 25 mΩ = 25000 µΩ, current_lsb = 100 µA:
       Cal = 40960000 / (100 × 25) = 16384 ✓ */

    if (cal > 0xFFFF) cal = 0xFFFF;
    err = mi_write16(m, REG_CALIBRATION, cal);
    if (err) return err;

    dev_info(&m->client->dev,
             "INA219 config: shunt=%u µΩ, current_lsb=%d µA, cal=%u\n",
             m->shunt_uohms, m->current_lsb_uA, cal);
    return 0;
}

/* === Read functions returning values in hwmon units === */

static int mi_read_shunt_uV(struct myina *m, s32 *uV)
{
    s16 raw;
    int err = mi_read16(m, REG_SHUNT_V, &raw);
    if (err) return err;
    /* INA219 raw LSB = 10 µV */
    *uV = (s32)raw * 10;
    return 0;
}

static int mi_read_bus_mV(struct myina *m, s32 *mV)
{
    s16 raw;
    int err = mi_read16(m, REG_BUS_V, &raw);
    if (err) return err;
    /* bits 15:3 = bus voltage in 4 mV units */
    *mV = (raw >> 3) * 4;
    return 0;
}

static int mi_read_current_mA(struct myina *m, s32 *mA)
{
    s16 raw;
    int err = mi_read16(m, REG_CURRENT, &raw);
    if (err) return err;
    /* current_uA = raw × current_lsb_uA */
    *mA = ((s32)raw * m->current_lsb_uA) / 1000;
    return 0;
}
```

```
static int mi_read_power_uW(struct myina *m, s32 *uW)
{
    s16 raw;
    int err = mi_read16(m, REG_POWER, &raw);
    if (err) return err;
    /* Power LSB = 20 × current_LSB */
    *uW = (s32)raw * m->current_lsb_uA * 20;
    return 0;
}

/* === hwmon-style sysfs attributes === */

static ssize_t in0_input_show(struct device *dev, struct device_attribute *a, char *buf)
{
    struct myina *m = dev_get_drvdata(dev);
    s32 uV;
    int err;
    mutex_lock(&m->lock);
    err = mi_read_shunt_uV(m, &uV);
    mutex_unlock(&m->lock);
    if (err) return err;
    /* hwmon convention: in*_input is in mV; shunt voltage is small, report in mV */
    return sprintf(buf, "%d\n", uV / 1000);
}

static ssize_t in1_input_show(struct device *dev, struct device_attribute *a, char *buf)
{
    struct myina *m = dev_get_drvdata(dev);
    s32 mV;
    int err;
    mutex_lock(&m->lock);
    err = mi_read_bus_mV(m, &mV);
    mutex_unlock(&m->lock);
    if (err) return err;
    return sprintf(buf, "%d\n", mV);
}

static ssize_t curr1_input_show(struct device *dev, struct device_attribute *a, char
*buf)
{
    struct myina *m = dev_get_drvdata(dev);
    s32 mA;
    int err;
    mutex_lock(&m->lock);
    err = mi_read_current_mA(m, &mA);
    mutex_unlock(&m->lock);
    if (err) return err;
    return sprintf(buf, "%d\n", mA);
}

static ssize_t power1_input_show(struct device *dev, struct device_attribute *a, char
*buf)
{
    struct myina *m = dev_get_drvdata(dev);
    s32 uW;
    int err;
    mutex_lock(&m->lock);
    err = mi_read_power_uW(m, &uW);
```

```
    mutex_unlock(&m->lock);
    if (err) return err;
    return sprintf(buf, "%d\n", uW);
}

static DEVICE_ATTR_RO(in0_input);
static DEVICE_ATTR_RO(in1_input);
static DEVICE_ATTR_RO(curr1_input);
static DEVICE_ATTR_RO(power1_input);

static struct attribute *mi_attrs[] = {
    &dev_attr_in0_input.attr,
    &dev_attr_in1_input.attr,
    &dev_attr_curr1_input.attr,
    &dev_attr_power1_input.attr,
    NULL,
};
ATTRIBUTE_GROUPS(mi);

/* === Probe / Remove === */

static int mi_probe(struct i2c_client *client)
{
    struct myina *m;
    struct device *hwmon_dev;
    u32 shunt_uohms = 25000;     /* default 25 mΩ if DT doesn't say */
    int err;

    m = devm_kzalloc(&client->dev, sizeof(*m), GFP_KERNEL);
    if (!m) return -ENOMEM;
    m->client = client;
    mutex_init(&m->lock);

    of_property_read_u32(client->dev.of_node, "shunt-resistor-micro-ohms",
                         &shunt_uohms);
    m->shunt_uohms = shunt_uohms;

    err = mi_init(m);
    if (err) return err;

    hwmon_dev = devm_hwmon_device_register_with_groups(&client->dev,
                                                       "myina219", m, mi_groups);
    return PTR_ERR_OR_ZERO(hwmon_dev);
}

static const struct of_device_id mi_of_match[] = {
    { .compatible = "linuxlearn,myina219" },
    { }
};
MODULE_DEVICE_TABLE(of, mi_of_match);

static const struct i2c_device_id mi_id[] = { { "myina219", 0 }, { } };
MODULE_DEVICE_TABLE(i2c, mi_id);

static struct i2c_driver mi_driver = {
    .driver = {
        .name = "myina219",
        .of_match_table = mi_of_match,
    },
```

```
    .probe = mi_probe,
    .id_table = mi_id,
};
module_i2c_driver(mi_driver);

MODULE_LICENSE("GPL");
```

DT:

```
&i2c1 {
    ina219@40 {
        compatible = "linuxlearn,myina219";
        reg = <0x40>;
        shunt-resistor-micro-ohms = <25000>;    /* 25 mΩ */
    };
};
```

Test:

```
[root@pa-mini:~]# insmod myina219.ko
[root@pa-mini:~]# ls /sys/class/hwmon/hwmon0/
curr1_input  in0_input  in1_input  name  power1_input

[root@pa-mini:~]# cat /sys/class/hwmon/hwmon0/in1_input
5023          ← bus voltage = 5.023 V

[root@pa-mini:~]# cat /sys/class/hwmon/hwmon0/curr1_input
124           ← current = 124 mA

[root@pa-mini:~]# cat /sys/class/hwmon/hwmon0/power1_input
622000        ← power = 622 mW = 0.622 W

[root@pa-mini:~]# sensors
myina219-i2c-1-40
Adapter: i.MX6UL I2C adapter
in0:           1.24 mV    (shunt voltage)
in1:           5.02 V     (bus voltage)
curr1:       124.00 mA
power1:        0.62 W
```

`sensors` (from lm-sensors package) auto-discovers and prints all hwmon devices. Production telemetry scrapes `/sys/class/hwmon/` directly.

75.6 hwmon vs IIO, when to use which

Both expose sensor readings via sysfs. Conventions and audience differ:

> **sysfs:** a kernel-generated filesystem under /sys that exposes devices, drivers, and attributes.

	hwmon	IIO
Primary use	System health monitoring	General-purpose sensing
Typical readers	`sensors`, fan-control daemons, Grafana	Custom user-space apps, scientific apps
Units	Fixed (mV, mA, mW, mC)	Raw + scale; user multiplies
Buffered capture	no	yes (triggers + buffers)
Multi-axis sensors	awkward (would need many channels)	first-class (X/Y/Z modifiers)
Naming	`in*_input`, `curr*_input`	`in_<type>_<modifier>_raw`

Current monitors live in hwmon. IMUs, ADCs, and environmental sensors live in IIO. A few chips have both drivers, pick one in your kernel config and disable the other.

75.7 INA226, improved sibling

INA226's interface is nearly identical to INA219. The registers are mostly the same. Improvements:

- **16-bit shunt-voltage ADC** with 2.5 µV LSB (vs INA219's 10 µV).
- **Programmable averaging**: 1, 4, 16, 64, 128, 256, 512, 1024 samples averaged per result.
- **Alert pin** + alert configuration: assert ALERT on overcurrent, undervoltage, etc.
- **Mask/enable register** to select which alert sources fire.

Driver-wise: same `ina2xx.c`, just with `compatible = "ti,ina226"` and different per-register multipliers. From-scratch, you'd copy the INA219 driver and update the constants.

75.8 INA3221, 3-channel

INA3221 is **three independent measurement channels** in one chip, with three pairs of shunt-voltage / bus-voltage registers (channels 1, 2, 3). Plus three alert pins.

Mainline driver: `drivers/hwmon/ina3221.c`. The DT specifies channels via subnodes:

```
&i2c1 {
    ina3221@40 {
        compatible = "ti,ina3221";
        reg = <0x40>;
        #address-cells = <1>;
        #size-cells = <0>;

        input@0 {
            reg = <0>;
            label = "VBUS_5V";
            shunt-resistor-micro-ohms = <10000>;
        };
        input@1 {
            reg = <1>;
            label = "VBUS_3V3";
            shunt-resistor-micro-ohms = <50000>;
        };
        input@2 {
            reg = <2>;
            label = "VBUS_1V8";
            shunt-resistor-micro-ohms = <100000>;
        };
    };
};
```

Result: `/sys/class/hwmon/hwmon0/in*_label`, `in*_input`, `curr*_input` × 3 channels. Production-grade 3-rail monitoring on one chip with one I²C address.

75.9 Key Notes

- The shunt, analog input range and calibration setting define the current measurement. Digital resolution alone does not determine useful accuracy.
- A larger shunt improves small-signal voltage at the cost of dissipation and voltage drop. Placement and common-mode limits constrain the circuit.
- Averaged telemetry and instantaneous peak protection serve different purposes. A slow monitor can report a reasonable mean while missing a harmful transient.
- Hwmon exposes monitoring quantities, while IIO is suited to sampled-data interfaces. The framework choice should follow how the system uses the measurements.

75.10 Pitfalls

- **Calibration register left at zero.** Current and Power read zero forever. The #1 INA gotcha. You’ll lose an hour to it.

- **Shunt placement on low-side.** INA219 is *high-side* (V+ and V− both near VBUS). Low-side measurement is also possible (V+ at shunt-load side, V− at GND) but limits range to bus voltages near GND. For motor PWM where you want to keep GND clean, high-side is mandatory.

> **MCU bridge:** Think of Linux PWM like an MCU timer output channel, except the driver exposes period, duty cycle, polarity, and enable state through a subsystem.
> **PWM:** Pulse-Width Modulation, a timer output whose duty cycle controls average power or encodes timing.

- **Shunt too small.** 1 mΩ shunt + 100 mA load = 100 µV shunt voltage = 10 LSB on INA219 = 5–10 LSB of noise. Get a bigger shunt or use INA226.
- **Shunt too big.** 1 Ω shunt + 1 A load = 1 W dissipated in the resistor. Resistor heats, drift, power waste.
- **Bus voltage > 26 V.** INA219's max bus voltage. Higher = damage. INA226 goes to 36 V. Even higher needs different chips (INA138, INA260).
- **Common-mode voltage limit.** Both INA pins must sit within the chip's input range. On a high-side shunt at 24 V, V+ and V- are both close to 24 V. INA219 is rated up to 26 V, so that is fine. On a 36 V rail, use INA226.
- **PWM-controlled load measurement.** Switching loads at kHz rates produce ripple that the INA's slow ADC averages, you get the mean current, not peak. For peak measurement, use a faster current-sense amplifier + scope.
- **DC blocking on dynamic loads.** If your "current monitor" is reading near zero and you're sure load is drawing power, check if there's a series capacitor isolating DC, capacitor blocks DC current entirely. The shunt-amplifier reads zero.
- **Mismatched A0/A1 strapping in DT vs hardware.** DT says `reg = <0x40>` but board pinned to 0x44. Silent failure.

75.11 Going deeper

- **`drivers/hwmon/ina2xx.c`**: production driver covering the INA family.
- **`drivers/hwmon/ina3221.c`**: multi-channel.
- **INA219 datasheet (TI SBOS448G)**: calibration formula appendix. Current_LSB choice guide.
- **INA226 datasheet (TI SBOS547A)**: averaging mode tradeoffs. Alert configuration.
- **`Documentation/hwmon/`**: hwmon framework documentation. ABI. Convention.

> **ABI:** Application Binary Interface: the calling convention, register use, binary format, and library contract that let separately built code run together.

- **`Documentation/hwmon/sysfs-interface.rst`**: the canonical attribute naming reference.
- **`sensors-detect` / `sensors`**: lm-sensors user-space tooling.

> Next chapter: **Chapter 76: Battery fuel gauge + charger.** Linux’s `power_supply_class` framework, MAX17048 ModelGauge for SoC estimation, TP4056 single-cell charger, BQ24074 path-managed charger.

Chapter 76: Battery fuel gauge + charger

> **What:** the three pieces of a battery-powered embedded product: a **fuel gauge** that tracks state-of-charge (Maxim MAX17048, I²C, “ModelGauge” algorithm), a **charger** that manages the CC/CV cycle (TI TP4056, analog, simple. Or TI BQ24074, I²C-configurable, path-managed), and the **`power_supply_class`** framework that ties them into Linux. For each: physics, protocol, mainline driver, plus a from-scratch MAX17048 driver implementing the `power_supply` provider model.
>
> **Why:** any battery-powered product needs to report “percent full” to the user accurately. The naive approach, voltage divider into ADC, lookup table, is wrong: Li-ion voltage doesn’t track SoC linearly, and load voltage drops badly bias the reading. A fuel gauge chip does the right thing: integrates current (coulomb counting) or models the cell (impedance tracking) to get sub-2 % SoC accuracy. Plus a charger that knows when to terminate.
>
> **Focus:** The kernel exposes battery state to user-space through `power_supply_class`, the same framework used by laptops, phones, and embedded boards. Drivers register as `power_supply` providers. User-space (UPower, systemd-battery-monitor, your custom app) reads from `/sys/class/power_supply/`. Same shape for laptop batteries, e-bike packs, phones, embedded devices.

76.1 Chip comparison

	Maxim MAX17048	TI TP4056	TI BQ24074
Function	1-cell fuel gauge	1-cell Li-ion charger	1-cell path-managed charger
Interface	I²C (0x36)	none (analog control)	I²C (0x6B)
Algorithm	ModelGauge (impedance)	linear CC/CV	linear CC/CV, dynamic-power-path mgmt
Charge current setting	n/a	resistor (≤ 1 A)	resistor (≤ 1.5 A)
Charge termination	n/a	C/10	C/10 + safety timer

	Maxim MAX17048	TI TP4056	TI BQ24074
Power-path	n/a	no, battery powers load while charging	yes, input powers load directly, battery only when input absent
Idle current	23 µA	none	100 µA
Volume price	$2.50–4.00	$0.30–0.80	$2–4
Mainline driver	`power/supply/max17040_battery.c`	none (no I²C; status detected via GPIO)	`power/supply/bq2415x_charger.c`

Pick guide:

- **MAX17048**: lowest-cost real fuel gauge. Use unless you need coulomb-counting precision.
- **TP4056**: cheap charger. Fine for “low-power device that mostly runs from battery.” No power-path = device loses power when battery dies, even with USB plugged.
- **BQ24074**: production-quality charger with power-path. Device stays on with input even when battery is removed. Charger negotiates input current. The standard choice for production-grade hardware.

A complete battery system typically combines a charger + a gauge: TP4056 + MAX17048 is the cheap stack. BQ24074 + MAX17048 is the production stack.

76.2 The physics, why a fuel gauge is needed

A naïve “voltage → %” approach fails for two reasons:

1. **Voltage isn’t linear with SoC.** A Li-ion cell goes 4.2 V (100 %) → 3.7 V (50 %) → 3.4 V (10 %) → 3.0 V (cutoff). The middle plateau is flat, a small voltage range covers most of the capacity.
2. **Voltage drops under load.** The cell’s internal resistance (50–500 mΩ depending on age and chemistry) causes voltage to drop by `I × R_internal`. A 500 mA load on a 100 mΩ cell drops the measured voltage by 50 mV, equivalent to ~5 % SoC on the plateau. If you read while a high-current load is active, you may tell the user 60 % when the resting SoC is actually 65 %.

Two real-world approaches:

- **Coulomb counting** (TI BQ27xxx, MAX17042). Measure current with a shunt. Integrate over time. Subtract from a known-full capacity. Pros: accurate. Cons: needs full charge cycle to calibrate. Drift over time as full capacity changes with age.
- **Impedance tracking / ModelGauge** (MAX17048). Build an internal model of the cell's V/I/T/SoC relationship. Use measured V and I (in MAX17048's case, just V, it estimates I from V swings) to look up SoC. Pros: no shunt needed. No full-charge required to calibrate. Cons: needs a per-chemistry pre-loaded model.

MAX17048's main selling point: 23 µA standby current, no shunt resistor required, a factory-loaded "typical Li-ion" model, and SoC accurate enough for most consumer products.

76.3 Protocol, MAX17048

Register map (every register is 16 bits, big-endian):

Reg	Name	Purpose
0x02	VCELL	Cell voltage, 78.125 µV/LSB
0x04	SOC	State of charge, bits 15:8 = %, bits 7:0 = fractional 256ths
0x06	MODE	Quick-start trigger
0x08	VERSION	IC version (0x0011 for MAX17048)
0x0A	HIBRT	Hibernate threshold
0x0C	CONFIG	Rcomp + alert threshold
0x14	VALRT	Voltage alert min/max
0x16	CRATE	C-rate estimate (current as fraction of capacity)

Reg	Name	Purpose
0x18	VRESET / ID	Soft-reset trigger; chip ID
0x1A	STATUS	Alert flags

A reading

```
   Host: START | 0x6C | 0x02 | START | 0x6D | (2 bytes MSB,LSB) | STOP
   (0x36 << 1 = 0x6C for write; ... | 1 = 0x6D for read)
```

`raw = (buf[0] << 8) | buf[1]; vcell_uV = raw × 78.125;`, voltage in microvolts. Divide by 1000 for mV.

For SoC: `soc_pct = buf[0]; soc_frac = buf[1];` so a reading of `0x52 0xC0` is 82 + 192/256 = 82.75 %.

Bring-up

Practically no init needed for default operation:

1. Read VERSION (0x08). Verify chip is alive.
2. (Optional) write MODE = 0x4000 to issue “quick-start”, forces a fresh SoC estimate, useful after a brand-new pack is connected.
3. Start reading VCELL and SOC.

The “ModelGauge” runs continuously inside the chip on its own 30-second cycle. You just read the output.

76.4 power_supply_class, how Linux models batteries

`power_supply_class` (in `drivers/power/supply/power_supply_core.c`) is the framework that ties together battery / charger / AC-adapter into one consistent user-space view.

A driver registers as one or more **power_supply** objects, each with a type (BATTERY, MAINS, USB, etc.) and a set of *properties* it can report:

```
enum power_supply_property {
    POWER_SUPPLY_PROP_STATUS,             /* CHARGING / DISCHARGING / FULL / NOT_CHARGING
*/
    POWER_SUPPLY_PROP_HEALTH,              /* GOOD / OVERHEAT / DEAD / OVERVOLTAGE / etc.
*/
    POWER_SUPPLY_PROP_PRESENT,             /* battery present (0/1) */
    POWER_SUPPLY_PROP_VOLTAGE_NOW,         /* µV */
```

```
    POWER_SUPPLY_PROP_CURRENT_NOW,         /* µA */
    POWER_SUPPLY_PROP_CHARGE_NOW,          /* µAh — coulomb counters */
    POWER_SUPPLY_PROP_CHARGE_FULL,         /* µAh capacity */
    POWER_SUPPLY_PROP_CAPACITY,            /* % (0..100) */
    POWER_SUPPLY_PROP_CAPACITY_LEVEL,      /* CRITICAL/LOW/NORMAL/HIGH/FULL */
    POWER_SUPPLY_PROP_TEMP,                /* deci-degrees C */
    POWER_SUPPLY_PROP_TIME_TO_EMPTY_NOW,   /* seconds */
    POWER_SUPPLY_PROP_TIME_TO_FULL_NOW,
    /* ... 40+ more */
};
```

The driver provides a single `get_property` callback that the framework calls with each property the user-space requests. User-space sees `/sys/class/power_supply/<name>/`:

```
[root@pa-mini:~]# ls /sys/class/power_supply/
battery   ac

[root@pa-mini:~]# cat /sys/class/power_supply/battery/capacity
82
[root@pa-mini:~]# cat /sys/class/power_supply/battery/voltage_now
3852000
[root@pa-mini:~]# cat /sys/class/power_supply/battery/status
Discharging
```

The naming is standardised across phones, laptops, and embedded boards.

76.5 Writing a MAX17048 power_supply driver from scratch

Goal: register the chip as a `POWER_SUPPLY_BATTERY` and report at least capacity + voltage + status. ~200 lines.

`mymax17048.c`:

```
#include <linux/init.h>
#include <linux/module.h>
#include <linux/i2c.h>
#include <linux/power_supply.h>

#define REG_VCELL    0x02
#define REG_SOC      0x04
#define REG_VERSION  0x08

struct mymax {
    struct i2c_client *client;
    struct power_supply *psy;
    struct mutex lock;
};

/* === Low-level I²C: 16-bit big-endian register reads === */

static int mm_read_reg(struct mymax *m, u8 reg, u16 *val)
{
```

```
    int err = i2c_smbus_read_word_swapped(m->client, reg);
    if (err < 0) return err;
    *val = err;
    return 0;
}

static int mm_read_voltage_uV(struct mymax *m, int *uV)
{
    u16 raw;
    int err = mm_read_reg(m, REG_VCELL, &raw);
    if (err) return err;
    /* MAX17048: 78.125 µV/LSB.
     * voltage_uV = raw × 78.125 = (raw × 78125) / 1000 */
    *uV = ((u32)raw * 78125) / 1000;
    return 0;
}

static int mm_read_soc_permille(struct mymax *m, int *permille)
{
    u16 raw;
    int err = mm_read_reg(m, REG_SOC, &raw);
    if (err) return err;
    /* High byte = percent; low byte = fractional 256ths.
     * Return as permille (parts per thousand) — power_supply_class uses integer %
usually,
     * but we'll return tenths and let CAPACITY round. */
    int pct = (raw >> 8) & 0xFF;
    int frac256 = raw & 0xFF;
    /* permille = pct × 10 + frac256 × 10 / 256  */
    *permille = pct * 10 + frac256 * 10 / 256;
    return 0;
}

/* === power_supply_class callback === */

static enum power_supply_property mm_props[] = {
    POWER_SUPPLY_PROP_PRESENT,
    POWER_SUPPLY_PROP_VOLTAGE_NOW,
    POWER_SUPPLY_PROP_CAPACITY,
    POWER_SUPPLY_PROP_STATUS,
};

static int mm_get_property(struct power_supply *psy,
                           enum power_supply_property prop,
                           union power_supply_propval *val)
{
    struct mymax *m = power_supply_get_drvdata(psy);
    int err = 0;
    int v;

    mutex_lock(&m->lock);
    switch (prop) {
    case POWER_SUPPLY_PROP_PRESENT:
        val->intval = 1;       /* if we're talking to it, it's present */
        break;
    case POWER_SUPPLY_PROP_VOLTAGE_NOW:
        err = mm_read_voltage_uV(m, &v);
        if (!err) val->intval = v;
        break;
```

```
    case POWER_SUPPLY_PROP_CAPACITY:
        err = mm_read_soc_permille(m, &v);
        if (!err) val->intval = v / 10;     /* round to % */
        if (val->intval > 100) val->intval = 100;
        break;
    case POWER_SUPPLY_PROP_STATUS:
        /* MAX17048 doesn't know charge direction directly — would need to
         * monitor SoC trend or read CRATE.  Stub: report UNKNOWN. */
        val->intval = POWER_SUPPLY_STATUS_UNKNOWN;
        break;
    default:
        err = -EINVAL;
    }
    mutex_unlock(&m->lock);
    return err;
}

static const struct power_supply_desc mm_desc = {
    .name           = "battery",
    .type           = POWER_SUPPLY_TYPE_BATTERY,
    .properties     = mm_props,
    .num_properties = ARRAY_SIZE(mm_props),
    .get_property   = mm_get_property,
};

/* === Probe / Remove === */

static int mm_probe(struct i2c_client *client)
{
    struct mymax *m;
    struct power_supply_config psy_cfg = {};
    u16 version;
    int err;

    m = devm_kzalloc(&client->dev, sizeof(*m), GFP_KERNEL);
    if (!m) return -ENOMEM;
    m->client = client;
    mutex_init(&m->lock);

    err = mm_read_reg(m, REG_VERSION, &version);
    if (err) return dev_err_probe(&client->dev, err, "version read failed\n");
    if ((version & 0xFFF0) != 0x0010)
        return dev_err_probe(&client->dev, -ENODEV,
                             "unexpected version 0x%04x\n", version);
    dev_info(&client->dev, "MAX17048 ready, version 0x%04x\n", version);

    psy_cfg.drv_data = m;
    psy_cfg.of_node  = client->dev.of_node;

    m->psy = devm_power_supply_register(&client->dev, &mm_desc, &psy_cfg);
    if (IS_ERR(m->psy))
        return dev_err_probe(&client->dev, PTR_ERR(m->psy),
                             "power_supply register failed\n");

    i2c_set_clientdata(client, m);
    return 0;
}

static const struct of_device_id mm_of_match[] = {
```

```
    { .compatible = "linuxlearn,mymax17048" },
    { }
};
MODULE_DEVICE_TABLE(of, mm_of_match);

static const struct i2c_device_id mm_id[] = { { "mymax17048", 0 }, { } };
MODULE_DEVICE_TABLE(i2c, mm_id);

static struct i2c_driver mm_driver = {
    .driver = {
        .name = "mymax17048",
        .of_match_table = mm_of_match,
    },
    .probe    = mm_probe,
    .id_table = mm_id,
};
module_i2c_driver(mm_driver);

MODULE_LICENSE("GPL");
```

DT:

```
&i2c1 {
    fuelgauge@36 {
        compatible = "linuxlearn,mymax17048";
        reg = <0x36>;
    };
};
```

Test:

```
[root@pa-mini:~]# insmod mymax17048.ko
[root@pa-mini:~]# ls /sys/class/power_supply/
battery
[root@pa-mini:~]# cat /sys/class/power_supply/battery/capacity
82
[root@pa-mini:~]# cat /sys/class/power_supply/battery/voltage_now
3852023
[root@pa-mini:~]# cat /sys/class/power_supply/battery/present
1
[root@pa-mini:~]# upower -i /org/freedesktop/UPower/devices/battery_battery
  native-path:          battery
  power supply:         yes
  state:                unknown
  energy-percentage:    82%
  voltage:              3.85 V
```

`upower` (the desktop daemon) auto-discovers the battery and presents it as a standard battery device. Same applies to systemd’s logind, fwupd, and many other userspace consumers.

To improve: implement STATUS by reading CRATE (register 0x16), a positive C-rate means charging, negative means discharging. Add it as a third property type the driver supports.

76.6 TP4056, the analog charger

TP4056 is a simple linear Li-ion charger:

```
    USB 5V ──► VIN  ┌──────────┐  BAT ──► Li-ion cell
                    │          │
                    │ TP4056   │  CHRG (active-low LED indicator)
                    │          │  STDBY (active-low LED indicator)
                    │          │  PROG (resistor to GND sets charge current)
                    └──────────┘
```

Charge current is set by `R_PROG`:

```
I_charge = 1200 / R_prog   amps
```

R = 1.2 kΩ → 1 A. R = 2.4 kΩ → 500 mA, etc.

The chip does:

1. **Trickle charge** (8 % of programmed current) until cell > 2.9 V.
2. **Constant current** at programmed current up to 4.2 V.
3. **Constant voltage** at 4.2 V. Current tapers.
4. **Termination** when current drops to 10 % of programmed.

No I²C, no control. CHRG and STDBY pins drive LEDs (or GPIOs into the SoC for status detection).

To integrate with Linux: wire CHRG to a GPIO. In DT, declare a `power_supply` of type AC with this GPIO. The mainline framework `gpio-charger.c` does this generic pattern:

> **MCU bridge:** Think of Linux GPIO like the same pin set/reset block you used on STM32, but accessed through a kernel subsystem that owns numbering, direction, interrupts, and user-space exposure. **GPIO:** General-Purpose Input/Output, a pin controlled as a digital input, output, or interrupt source.

```
charger {
    compatible = "gpio-charger";
    charger-type = "mains";
    gpios = <&gpio4 5 GPIO_ACTIVE_LOW>;     /* CHRG LED line */
    charge-status-gpios = <&gpio4 6 GPIO_ACTIVE_LOW>;
};
```

`/sys/class/power_supply/main_charger/online` reads 1 when charging.

TP4056 limitations: No power-path. When battery is empty *and* USB is plugged in, the chip charges the battery but does not separately power the load. Voltage on BAT depends on cell state. For a Linux SoC needing 4.2 V min at 500+ mA, this can cause boot loops on a deeply-discharged battery.

76.7 BQ24074, production-quality

BQ24074 adds:

- **Power-path**: input feeds VOUT directly when present. Battery feeds VOUT only when input absent. Device stays on regardless of battery state, as long as input is present.
- **Dynamic Power-Path Management (DPPM)**: monitors input voltage. If input collapses below the threshold (overloaded USB port), the chip reduces charge current to keep the load alive.
- **Configurable via I²C**: programmed charge current, termination voltage, safety timer.

Pick BQ24074 for any product that must come up the moment a USB cable is plugged in, regardless of whether the battery is alive.

Mainline driver: `drivers/power/supply/bq24190_charger.c` covers the BQ241xx family. DT:

```
&i2c1 {
    charger@6b {
        compatible = "ti,bq24074";
        reg = <0x6b>;
        ti,charge-current = <500000>;       /* 500 mA */
        ti,input-current-limit = <1000000>;  /* 1 A USB-3 capable port */
        interrupt-parent = <&gpio4>;
        interrupts = <12 IRQ_TYPE_EDGE_FALLING>;
    };
};
```

`/sys/class/power_supply/bq24074-charger/`:

```
input_current_limit
charge_current_limit
status                    ← Charging / Not Charging / Full
online                    ← input present
voltage_now
```

A user-space battery daemon (UPower, your own) combines this with the MAX17048's capacity to give a complete view.

76.8 Integrating: charger + gauge → battery status

To synthesise a full picture, your application reads:

```
charger_online = read("/sys/class/power_supply/charger/online")
charger_status = read("/sys/class/power_supply/charger/status")
battery_capacity = read("/sys/class/power_supply/battery/capacity")
battery_voltage  = read("/sys/class/power_supply/battery/voltage_now")

if charger_online:
    if battery_capacity == 100:
        print("Plugged in, fully charged")
    else:
        print(f"Charging: {battery_capacity}%")
else:
    if battery_capacity < 10:
        print(f"Low battery: {battery_capacity}%, shut down soon")
    else:
        print(f"On battery: {battery_capacity}%")
```

In product UI, do *time-averaged* SoC display, instantaneous readings vary by ±1 % when the load is unsteady. EMA with $\tau = 30$ s smooths the indicator.

76.9 Key Notes

- Fuel estimation, charging, cell protection and power-path management are separate responsibilities. A Linux battery percentage is not a charging-safety mechanism.
- A gauge’s model must match the cell and operating conditions. Voltage under load is not a direct, universal mapping to remaining capacity.
- The charger and system load share an electrical topology that determines behavior when external power changes. Driver properties cannot repair an unsuitable power path.
- Low-battery shutdown needs margin, hysteresis and a persistence policy. An estimated percentage alone does not guarantee enough energy to finish a filesystem write.

76.10 Pitfalls

- **Quick-start without justification.** Writing 0x4000 to MODE clears the chip’s internal model and forces a fresh estimate. Useful on fresh-pack-insertion. Harmful if done routinely (degrades SoC accuracy).
- **Reading SoC immediately after power-on.** Chip needs ~250 ms to initialise. The first read returns the cached value from last shutdown.

- **Trusting MAX17048 SoC on a cell-chemistry mismatch.** Factory-loaded model is “typical Li-ion 3.7 V nominal.” $LiFePO_4$ (3.2 V nominal) reads wildly wrong. Use BQ27xxx with the right chemistry profile for non-standard cells.
- **TP4056 without thermal management.** At 1 A charge, the chip dissipates ~1 W. Without thermal vias and a ground plane, it overheats and reduces charge current, slower charging.
- **TP4056 in deep-discharge cells.** Cells below 2.5 V might fail TP4056’s pre-charge. Use a chip with adjustable pre-charge current.
- **BQ24074 without I²C in the boot path.** If the chip needs config at boot to set higher input current, U-Boot must initialize it, otherwise it defaults to 100 mA input, which can’t power the SoC.

> **MCU bridge:** Think of U-Boot like a much larger boot stub plus debug monitor: it initializes hardware, loads the next image, and gives you commands before Linux starts. **U-Boot:** the bootloader that initializes enough hardware to load and start the Linux kernel.

- **Power-path absent on production-bound product.** User unplugs charger. Device dies. They expected the battery to take over. Verify your topology before laying out.
- **Forgetting `present` property.** Some user-space code refuses to talk to a battery whose `present` is 0. Always report 1 if the chip is responding.
- **Capacity hysteresis.** Naïve daemons toggle “5% low warning” at 4.9 % then back at 5.0 %. Add hysteresis (warn at 5 %, clear at 8 %).

76.11 Going deeper

- **`drivers/power/supply/max17040_battery.c`**: production MAX17048/40/44 driver.
- **`drivers/power/supply/bq27xxx_battery.c`**: coulomb-counting gauges from TI.
- **`drivers/power/supply/gpio-charger.c`**: generic “charging status from GPIO” driver.
- **`drivers/power/supply/bq2415x_charger.c`** + `bq24190_charger.c`, TI charger family.
- **`Documentation/power/power_supply_class.rst`**: framework documentation.
- **`Documentation/ABI/testing/sysfs-class-power`**: full sysfs attribute list.

> **ABI:** Application Binary Interface: the calling convention, register use, binary format, and library contract that let separately built code run together. **sysfs:** a kernel-generated filesystem under /sys that exposes devices, drivers, and attributes.

- **MAX17048 datasheet (Maxim)**: register map. Quickstart guide.
- **TP4056 datasheet (NanJing Top Power)**: application circuit.
- **BQ24074 datasheet (TI SLUS818)**: power-path explained.
- **UPower source** at https://gitlab.freedesktop.org/upower/upower, how user-space consumes power_supply_class.

> **End of Group E, Power & current (Ch 75–76).** Together with Ch 51B (power management) you now have all three pieces of low-power product engineering: PM saves power, INA reports it, fuel gauge predicts remaining time.

> Next chapter: **Chapter 77: 1-Wire sensors (DS18B20 / DHT22).** Maxim’s odd “one wire + ground” protocol. Slow but charming. The kernel’s w1 subsystem. Why DHT22 is a worse fit for Linux than for an MCU.

Chapter 77: 1-Wire sensors

> **What:** Maxim's **1-Wire** protocol, one digital pin (plus ground) carries bidirectional half-duplex data with timing-based bit framing. The well-supported case: **DS18B20** (digital thermometer, real 1-Wire, kernel `w1` subsystem). The lookalike: **DHT22** (single-wire T/H, *not* 1-Wire, hostile to Linux GPIO timing). For DS18B20: protocol, the `w1` master / slave architecture, mainline driver internals, and a from-scratch w1-slave driver. For DHT22: a clear-eyed look at why DHT22 is a poor fit for Linux plus what to do instead. **MCU bridge:** Think of Linux GPIO like the same pin set/reset block you used on STM32, but accessed through a kernel subsystem that owns numbering, direction, interrupts, and user-space exposure. **GPIO:** General-Purpose Input/Output, a pin controlled as a digital input, output, or interrupt source.
>
> **Why:** 1-Wire is the cheap, long-cable, parasitically-powered alternative to I²C. A 30-meter cable with 10 DS18B20s on it works. *Real* 1-Wire devices (with proper protocol implementations) are kernel-friendly. DHT22 uses the same physical wiring (one signal line plus ground, with parasitic power) but a different, incompatible bit-framing scheme, and that timing requires µs-accurate edge detection that Linux GPIO can't reliably deliver.
>
> **Focus: the master must generate tightly-timed pulse widths** (15 µs reset, 60 µs slot, 1 µs sample window). For DS18B20 this is done by the **w1 master** driver, usually a "GPIO bit-bang" master with PREEMPT_RT helping, or a hardware UART repurposed as a w1 master. The slave devices live in `drivers/w1/slaves/`. Once the master is reliable, writing a slave driver is short, the w1 core does the work. **PREEMPT_RT:** the Linux real-time patch set that makes more kernel paths preemptible and reduces latency.

77.1 1-Wire protocol, what's on the wire

1-Wire uses a single GPIO with a ~4.7 kΩ pull-up to 3.3 V. Idle = high. The master pulls low for specific durations to send bits and to query slaves. Slaves also pull low to respond, sharing the same wire.

Reset / Presence

```
master pulls LOW for ≥ 480 µs            ← RESET
master releases (pull-up takes line HIGH)
... 15..60 µs settling ...
if a slave is present, it pulls LOW       ← PRESENCE
for 60..240 µs, then releases.
```

The master samples the line at ~60 µs after release: low = at least one slave present.

Write 1 / Write 0 bits

```
Write 1:
    master pulls LOW for 1..15 µs, then releases.
    Slave samples at ~30 µs after master pulls low; sees HIGH → reads 1.

Write 0:
    master pulls LOW for 60..120 µs.
    Slave samples at ~30 µs after master pulls low; still LOW → reads 0.
```

The slave's sample point (~30 µs) discriminates 1 vs 0 based on whether the line has recovered.

Read bit

```
master pulls LOW for 1..15 µs, then releases.
If slave wants to send 0, it keeps the line LOW for ~30 µs.
If slave wants to send 1, it leaves the line HIGH (pull-up).
Master samples at ~13 µs after starting the pulse:
    LOW = 0, HIGH = 1.
```

The whole protocol is timing-based. Detecting that an edge happened is not the same as knowing *when* it happened. The host must drive transitions with about ±5 µs accuracy.

Standard commands

After a reset/presence, the master sends 8-bit commands LSB-first:

Command	Hex	Purpose
READ ROM	0x33	Read slave's 64-bit ROM ID (only one slave on bus)
SKIP ROM	0xCC	Broadcast to all slaves (skip addressing)
MATCH ROM	0x55	Address one slave by ROM ID (8 bytes)
SEARCH ROM	0xF0	Enumerate slaves (multi-drop)

After ROM-addressing, slave-specific function commands follow (e.g., DS18B20's 0x44 = Convert Temperature).

77.2 DS18B20, a proper 1-Wire device

DS18B20 is a 9–12 bit programmable thermometer. Power 3.0–5.5 V, range −55 to +125 °C, accuracy ±0.5 °C at typical room temperature, ~750 ms for a 12-bit conversion.

Sequence to read temperature:

```
1. Reset + presence pulse.
2. SKIP ROM (0xCC) — assuming only one slave or broadcasting.
3. CONVERT T (0x44) — start a temperature conversion.
4. Wait 750 ms (or poll the bus: while the chip is converting, it holds the
   bus LOW; release means done).
5. Reset + presence pulse.
6. SKIP ROM (0xCC).
7. READ SCRATCHPAD (0xBE).
8. Read 9 bytes: temp_lsb, temp_msb, TH, TL, config, reserved×3, CRC.
9. Validate CRC-8.
10. Decode: temp_raw = (msb << 8) | lsb; temp_C = temp_raw / 16.0  (signed!).
```

Multiple DS18B20s on the same bus: SEARCH ROM enumerates them. MATCH ROM addresses each by 64-bit ID. Otherwise SKIP ROM broadcasts to all.

77.3 The Linux `w1` subsystem

Source: `drivers/w1/`.

```
drivers/w1/
├── w1.c            ← core (slave registration, search)
├── w1_io.c         ← bus primitives (reset, read/write byte, sample)
├── w1_int.c        ← internal interfaces
├── w1_netlink.c    ← user-space notification
├── masters/        ← bus-master drivers
│   ├── w1-gpio.c
│   ├── ds2482.c     (I²C-to-1-Wire bridge)
│   ├── ds2490.c     (USB-to-1-Wire dongle)
│   └── omap_hdq.c   (TI's hardware w1 controller)
└── slaves/         ← slave drivers
    ├── w1_therm.c   (DS18B20, DS1822, DS28EA00)
    ├── w1_ds2406.c
    ├── w1_ds2438.c
    └── ...
```

Two layers: a *master* implements bus primitives. *slaves* are registered after enumeration.

A master's contract

```
struct w1_bus_master {
    void *data;
    u8  (*read_byte) (void *);
    void (*write_byte)(void *, u8);
```

```
    u8  (*read_bit)  (void *);
    void (*write_bit) (void *, u8);
    u8  (*touch_bit) (void *, u8);
    u8  (*reset_bus) (void *);
    /* ... */
};
```

A master can choose to implement either bit-level or byte-level primitives. The core synthesises the missing operations. A *GPIO bit-bang master* implements `reset_bus`, `read_bit`, `write_bit` using `gpiod_*` + `udelay`/`ndelay`. The core composes those into byte reads/writes.

`w1-gpio` master

```
/* drivers/w1/masters/w1-gpio.c — simplified */

static u8 w1_gpio_read_bit(void *data)
{
    struct w1_gpio_platform_data *pdata = data;
    u8 bit;

    /* Pull low for ~6 µs */
    gpiod_direction_output(pdata->gpiod, 0);
    udelay(6);
    /* Release; let pull-up recover */
    gpiod_direction_input(pdata->gpiod);
    udelay(9);    /* sample at ~15 µs after low pulse start */
    bit = gpiod_get_value(pdata->gpiod);
    udelay(55);   /* finish the slot */
    return bit;
}

static void w1_gpio_write_bit(void *data, u8 bit)
{
    struct w1_gpio_platform_data *pdata = data;
    if (bit) {
        gpiod_direction_output(pdata->gpiod, 0);
        udelay(6);
        gpiod_direction_input(pdata->gpiod);
        udelay(64);
    } else {
        gpiod_direction_output(pdata->gpiod, 0);
        udelay(60);
        gpiod_direction_input(pdata->gpiod);
        udelay(10);
    }
}

static u8 w1_gpio_reset_bus(void *data)
{
    struct w1_gpio_platform_data *pdata = data;
    u8 presence;

    gpiod_direction_output(pdata->gpiod, 0);
    udelay(480);
    gpiod_direction_input(pdata->gpiod);
```

```
    udelay(70);
    presence = gpiod_get_value(pdata->gpiod);
    udelay(410);
    return presence;
}
```

`udelay()` is **busy-wait**, not sleep. The whole sequence holds the CPU for ~70 µs (worst case) per bit. Reading the 9-byte scratchpad takes ~5 ms of busy-wait, during which other userspace can't preempt. Fine for "read every 5 seconds." Not fine for thousands of reads per second.

Crucially, `udelay` and the GPIO writes happen *with preemption disabled*. Without that protection, a scheduler tick mid-pulse would distort timing, bit becomes garbage. The `w1-gpio` driver wraps the bit operations in `local_irq_disable()` / `local_irq_enable()` around the timing-critical region.

This is why `w1-gpio` *works* on standard Linux: the master driver explicitly disables interrupts during each bit. The timing tolerance (1-Wire allows ±15 µs slop) absorbs the latency of one tick worth of pending IRQs.

Slave enumeration

After registering a master, the w1 core starts a kthread that:

1. Issues SEARCH ROM (0xF0) periodically.
2. Walks the binary-tree search algorithm to enumerate all slave ROM IDs.
3. For each new ROM ID, looks up the *family code* (top byte), 0x28 for DS18B20, 0x26 for DS2438, etc.
4. Looks up the corresponding slave driver. Calls its `add_slave` callback.
5. The slave driver registers per-device sysfs attributes.

> **sysfs:** a kernel-generated filesystem under /sys that exposes devices, drivers, and attributes.

User-space sees:

```
/sys/bus/w1/devices/28-0000054321ab/   ← one DS18B20
  ├── temperature       (read this)
  ├── ext_power
  ├── eeprom_cmd
  ├── name
  └── ...

/sys/bus/w1/devices/w1_bus_master1/    ← the master
  ├── w1_master_slave_count
```

```
   ├── w1_master_slaves
   └── w1_master_search
```

A read of `/sys/bus/w1/devices/28-.../temperature` triggers the conversion + read sequence, returning ASCII milli-degrees:

```
[root@pa-mini:~]# cat /sys/bus/w1/devices/28-0000054321ab/temperature
23187
[root@pa-mini:~]# # → 23.187 °C
```

77.4 Writing a w1 slave driver from scratch

We won't rewrite the master (the bit-bang master is well-engineered already and re-doing it teaches little). Instead we'll write a **slave driver** for an imaginary family, say, a custom sensor with family code 0xA5 that returns 4 bytes when commanded with 0xCC, 0xBE.

The w1 core handles enumeration. We provide a `w1_family` with `add_slave` / `remove_slave` callbacks:

`my_w1_slave.c`:

```
#include <linux/init.h>
#include <linux/module.h>
#include <linux/slab.h>
#include <linux/sysfs.h>
#include <linux/w1.h>

#define W1_FAMILY_LL_CUSTOM 0xA5
#define CMD_READ_DATA       0xBE   /* slave-specific function command */

static ssize_t value_show(struct device *dev, struct device_attribute *attr, char *buf)
{
    struct w1_slave *sl = dev_to_w1_slave(dev);
    u8 cmd[2];
    u8 rx[4];
    u32 value;

    mutex_lock(&sl->master->bus_mutex);

    if (w1_reset_select_slave(sl)) {
        mutex_unlock(&sl->master->bus_mutex);
        return -EIO;
    }

    cmd[0] = CMD_READ_DATA;
    w1_write_block(sl->master, cmd, 1);

    w1_read_block(sl->master, rx, 4);

    mutex_unlock(&sl->master->bus_mutex);

    /* Big-endian on the wire */
```

```
    value = (rx[0] << 24) | (rx[1] << 16) | (rx[2] << 8) | rx[3];
    return sprintf(buf, "%u\n", value);
}
static DEVICE_ATTR_RO(value);

static struct attribute *my_w1_attrs[] = {
    &dev_attr_value.attr,
    NULL,
};
ATTRIBUTE_GROUPS(my_w1);

static const struct w1_family_ops my_w1_fops = {
    .groups = my_w1_groups,
};

static struct w1_family my_w1_family = {
    .fid = W1_FAMILY_LL_CUSTOM,
    .fops = &my_w1_fops,
};

module_w1_family(my_w1_family);

MODULE_LICENSE("GPL");
MODULE_ALIAS("w1-family-" __stringify(W1_FAMILY_LL_CUSTOM));
```

That’s it, 50 lines. The w1 core handles enumeration. When a slave with family code 0xA5 is discovered, the core calls our driver’s `groups`, creating `/sys/bus/w1/devices/a5-XXXXXX/value`.

Three w1-core helpers used:

- **`w1_reset_select_slave(sl)`**: reset bus, issue MATCH ROM with this slave’s ID. Returns 0 on success.
- **`w1_write_block(master, buf, n)`**: write N bytes (each as 8 individual bit-write operations).
- **`w1_read_block(master, buf, n)`**: read N bytes.

These wrap the master’s bit-level primitives. The slave driver doesn’t see GPIO toggles at all, clean abstraction.

For comparison, the **mainline `w1_therm.c`** (DS18B20 slave) is ~1200 lines because it handles: family codes for multiple chips (DS18B20, DS1822, DS18S20, MAX31850), CRC validation, conversion timing with bus-power detection, resolution programming, alarm thresholds, EEPROM read/write, async conversions, multiple temperature-format conversions. The shape is the same as our 50-line version, just multiplied by features.

77.5 Setting up `w1-gpio` master in DT

```
onewire {
    compatible = "w1-gpio";
    gpios = <&gpio4 14 GPIO_ACTIVE_HIGH>;
```

```
    /* Optional: external pull-up control, parasitic mode, etc. */
};
```

`w1-gpio` master accepts a single `gpios` property (the data pin). The pull-up is assumed to be external (4.7 kΩ to 3.3 V). With slow polling and PREEMPT_RT, this works at 1-Wire's nominal timing on any GPIO that supports a few microseconds of `local_irq_disable()`.

Kernel config: `CONFIG_W1=y`, `CONFIG_W1_MASTER_GPIO=y`, `CONFIG_W1_SLAVE_THERM=y`.

Test:

```
[root@pa-mini:~]# ls /sys/bus/w1/devices/
28-0000054321ab    w1_bus_master1
[root@pa-mini:~]# cat /sys/bus/w1/devices/28-0000054321ab/temperature
23187
[root@pa-mini:~]# cat /sys/bus/w1/devices/w1_bus_master1/w1_master_slave_count
1
```

Long cables (10 m+) with multiple DS18B20s on a single GPIO, works.

77.6 DHT22, the imposter, and why Linux is wrong

DHT22 ("AM2302") *uses the same physical wiring* as 1-Wire but **invents its own incompatible protocol**:

1. Master pulls low for 1–10 ms.
2. Master releases. Pull-up takes line high.
3. Sensor pulls low for 80 µs (acknowledgment).
4. Sensor pulls high for 80 µs.
5. Sensor sends 40 bits of data, each as:
    - Low for **50 µs** (start of bit).
    - High for **26 µs** = 0, **70 µs** = 1.

The host must time the duration of each "high" segment to discriminate 0 from 1. **40 bits per measurement** = the host must accurately measure 40 short pulses (26 µs vs 70 µs, a 44 µs delta).

Why Linux struggles:

- Standard kernel preemption: any other ISR can delay your GPIO read by 100+ µs → bit misread.
- Even with PREEMPT_RT, scheduling jitter can be 50 µs+.

- Once one bit is misread, the whole 40-bit frame is wrong, there is no resync point until the next measurement starts.
- DHT22's CRC catches it, but you just retry, every read potentially fails.

The honest options:

1. **Don't.** Use SHT3x or AHT20 (proper I²C) instead. They cost the same.
2. **PREEMPT_RT + busy-wait GPIO** in a driver. Works ~80 % of the time. You retry until success.
3. **MCU helper.** An ATtiny / ESP8266 does the DHT22 timing and exposes the result via I²C / UART to Linux. The right answer if you must support DHT22.
4. **PRU coprocessor** on chips that have one (TI AM335x, Beaglebone Black). Not on i.MX6ULL.

The mainline `dht11.c` driver (`drivers/iio/humidity/dht11.c`) takes approach (2), uses high-resolution timers and IRQ-on-edge to measure pulse widths. It works on RPi-class hardware most of the time. Reliability varies by load.

> **MCU bridge:** Think of an IRQ like an EXTI/NVIC interrupt path, except Linux splits the hard interrupt from deferred work and must share lines across drivers. **IRQ:** interrupt request, the signal path that tells the CPU or interrupt controller that hardware needs service.

Verdict: if you see DHT22 on someone's product schematic, replace it with SHT3x.

77.7 Other 1-Wire devices worth knowing

Device	Family	Purpose
DS18B20	0x28	Temperature
DS18S20	0x10	Older temperature, 9-bit only
DS1822	0x22	Lower-precision DS18B20 sibling
DS2406	0x12	1-Wire GPIO extender (2 ports)

Device	Family	Purpose
DS2438	0x26	Battery monitor (V, T, current via shunt)
DS2431	0x2D	1-Kbit EEPROM
DS2433	0x23	4-Kbit EEPROM
DS2401	0x01	Unique 48-bit silicon serial number (= ROM ID only)
MAX31850	0x3B	K-type thermocouple-to-1-Wire

DS2401 is interesting for embedded products: it's a unique serial number you can solder onto a board and read over a single GPIO. Anti-counterfeiting, asset tracking. ~$0.30.

77.8 Key Notes

- Maxim 1-Wire and the DHT22's single-signal protocol are not the same interface. Similar wiring does not imply compatible timing or drivers.
- The 1-Wire master owns electrical timing and discovery, while slave drivers interpret device-specific commands and data. A hardware master can move timing responsibility away from scheduled software.
- Cable capacitance, topology and pull-up behavior constrain the bus. Successful communication on a short lead does not establish an arbitrary cable length.
- Parasitic powering creates an additional conversion-current requirement. CRC validation and device identity remain necessary even after enumeration succeeds.

77.9 Pitfalls

- **Missing pull-up.** No pull-up = bus floats = no devices enumerate. 4.7 kΩ is standard. Longer cables may need 2.2 kΩ.
- **Insufficient parasitic-power current.** DS18B20 in parasitic mode (3-pin connection, VDD tied to GND, draws power from the bus) needs the master to drive a strong pull-up *during conversion* (750 ms). `w1-gpio` doesn't do this. Use a 3-pin parasitic config only with the master that supports "strong pull-up."

- **GPIO open-drain capability.** 1-Wire requires the GPIO to switch between output-low and input (with pull-up). Some SoC GPIOs are limited. I.MX6ULL is fine.
- **Search ROM with bus contention.** With many slaves on a long cable, signal integrity degrades. Search may fail to find all slaves. Reduce cable length or use a ds2482 hardware master.
- **CRC errors.** w1-therm's `temperature` sysfs file returns "-1" or stale on CRC failure. Check `dmesg` for `w1_slave: crc mismatch`. Increase pull-up strength.
- **DHT22 expecting tight Linux GPIO timing.** It won't work reliably. See §77.6.
- **w1 polling thread CPU usage.** The w1 core re-enumerates every 10 seconds by default. On a busy bus this is ~50 ms of CPU. Adjustable via `/sys/bus/w1/devices/w1_bus_master1/w1_master_timeout`.

77.10 Going deeper

- **`drivers/w1/w1.c`** + `w1_io.c`, w1 core.
- **`drivers/w1/masters/w1-gpio.c`**: bit-bang master.
- **`drivers/w1/slaves/w1_therm.c`**: DS18B20 driver (~1200 lines, includes everything).
- **`drivers/w1/slaves/w1_ds2438.c`**: battery-monitor slave for comparison.
- **`drivers/iio/humidity/dht11.c`**: the "best-effort" DHT22 driver.
- **`Documentation/w1/`**: w1 framework documentation.
- **Maxim 1-Wire app notes** at maximintegrated.com (now analog.com), `AN187`, `AN126`, `AN148`.
- **DS18B20 datasheet (Maxim)**: protocol reference.
- **`drivers/w1/masters/ds2482.c`**: I²C-to-1-Wire bridge driver for systems where GPIO timing isn't tolerable.

> Next chapter: **Chapter 78: MEMS microphones.** Digital I²S microphones (INMP441, ICS-43434), capture audio with no analog audio chain.

Chapter 78: MEMS microphones

What: digital MEMS microphones, the chips that replaced analog electret mics in everything from phones to wearables. We focus on **I²S** mics (TDK InvenSense **INMP441**, TDK **ICS-43434**) and contrast with **PDM** mics. Plus the ASoC machine-driver pattern needed to wire one to the i.MX6ULL SAI, since this is one case where you really do write a small “machine driver” but not a chip driver. **ASoC:** ALSA System-on-Chip, the embedded audio layer that connects CPU audio ports, codecs, and board wiring.

Why: every smart speaker, voice-assistant, voice-controlled IoT device, dashcam, drone for FAA-broadcast, they all have one or more digital microphones. The mic outputs already-digitized PCM (or PDM). No separate ADC or codec required. The driver structure is unusual: there’s no codec chip with registers, just a simple I²S DAI. The ASoC `simple-card` machine driver handles this exact pattern.

Focus: A digital MEMS mic is, from Linux’s view, an I²S DAI without any control interface, clocks in, samples out. It samples audio internally, outputs PCM on SD when WS/LR-clock + BCLK are running. To the SAI driver, the mic is just an I²S slave. Wire SAI to the mic in DT via `simple-audio-card`, then `arecord` captures the audio. That is the whole audio-input pipeline.

Tooling. This chapter uses `alsa-utils` (`arecord`, `aplay`), `i2c-tools`. **ALSA:** Linux’s kernel and user-space audio stack.

- **Ubuntu-base (target):** `apt install alsa-utils i2c-tools`
- **Buildroot:** `BR2_PACKAGE_ALSA_UTILS=y BR2_PACKAGE_I2C_TOOLS=y` **Buildroot:** a configuration-driven build system that produces a complete root filesystem and related images.
- Full per-tool reference: Userspace tooling appendix. **MCU bridge:** Think of the rootfs as the firmware image’s file-backed runtime environment. On an MCU you link everything into flash. On Linux, programs and config live in this mounted tree. **rootfs:** root filesystem, the directory tree mounted at / that contains /bin, /etc, /dev, and libraries.

78.1 Sensor comparison

	TDK InvenSense INMP441	TDK ICS-43434	TDK ICS-41350 (PDM)
Interface	I²S	I²S	PDM (1-bit pulse-density)
Sample rate	7.19, 52.7 kHz	7.19, 51.6 kHz	1.024, 3.072 MHz PDM (decimates to ~48 kHz)
Resolution	24-bit	24-bit	1-bit out, 16-bit after decimation
SNR	61 dBA	65 dBA	64 dBA
Sensitivity	-26 dBFS @ 94 dB SPL	-26 dBFS	-26 dBFS
Acoustic overload (AOP)	120 dB SPL	116 dB SPL	130 dB SPL
Idle current	0.5 mA	0.6 mA	0.6 mA
Stereo (L/R select)	LR pin selects L or R channel	LR pin	LR pin
Package	6-pin LGA	6-pin LGA	5-pin LGA
Volume price	$1.50–2.50	$2–3.50	$1.50–2.50

Pick guide:

- **INMP441**: cheap, common, fine for voice. Default choice.
- **ICS-43434**: lower noise. Better for music or low-volume signal.
- **PDM**: more compact wiring (1-bit data). But needs the SoC's PDM-decoder hardware. I.MX6ULL's SAI has *only I²S*, no native PDM. So PDM mics are awkward on i.MX6ULL, skip.

78.2 I²S protocol primer

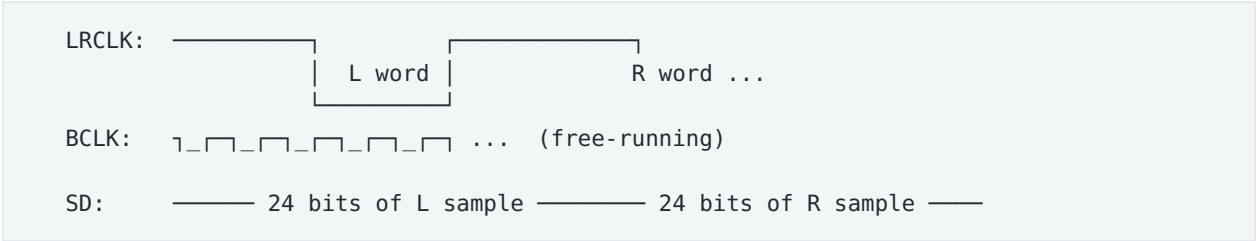
I²S is a 3-wire serial audio standard:

- **BCLK** (bit clock): one transition per bit.
- **LRCLK** (word/frame clock): toggles once per sample, marking left vs right channel.
- **SD** (serial data): the audio bits.

For a 48 kHz / 24-bit stereo stream:

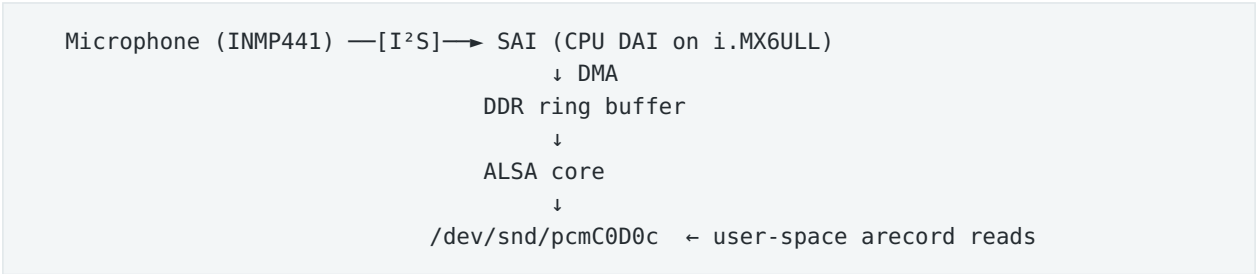
- LRCLK = 48 kHz square wave (low = L word, high = R word).
- BCLK = 48 kHz × 64 = 3.072 MHz (64 bits per stereo sample, conventional even for 24-bit data).
- SD = MSB-first, 24 bits per word, padded with zeros to fill the 32-bit slot.

The “master” generates BCLK + LRCLK. The “slave” follows. For a microphone, the SoC is master, mic is slave. The mic drives SD whenever LRCLK matches its programmed channel.



The mic outputs its sample MSB-first during its assigned LRCLK phase. Single mic → mono. Two mics with LR-select pin strapped opposite → stereo.

78.3 How the data flows in Linux



There’s no I²C control, the mic has no registers. The wires alone (BCLK, LRCLK, SD, LR-select strap) determine its behavior.

What the kernel needs

The kernel needs an ASoC sound card consisting of:

1. **CPU DAI**: the i.MX SAI driver (mainline, no work).
2. **Codec DAI**: a stub representing the mic. The mainline driver for this is called **`dmic`** (digital mic) or, for I²S mics specifically, there's no chip driver because the mic has no registers. The *ASoC machine driver* uses a fake codec.
3. **Machine driver**: wires the two DAIs together. Use **`simple-audio-card`** for this.

`simple-audio-card` (in `sound/soc/generic/simple-card.c`) is a generic ASoC machine driver. You describe the audio topology in DT. The driver builds a working sound card from the description. No coding required.

78.4 Device tree for INMP441 → SAI2

> **Template warning:** This block contains placeholder values. Replace compatible strings, GPIO numbers, addresses, and paths with values from your board before using it.

```
&sai2 {
    pinctrl-names = "default";
    pinctrl-0 = <&pinctrl_sai2>;
    assigned-clocks = <&clks IMX6UL_CLK_SAI2_SEL>,
                      <&clks IMX6UL_CLK_SAI2>;
    assigned-clock-parents = <&clks IMX6UL_CLK_PLL4_AUDIO_DIV>;
    assigned-clock-rates = <0>, <24576000>;
    status = "okay";
};

/* The "codec" is a fake — INMP441 has no control interface */
dmic_codec: dmic-codec {
    compatible = "dmic-codec";
    #sound-dai-cells = <0>;
};

sound {
    compatible = "simple-audio-card";
    simple-audio-card,name = "imx-inmp441";
    simple-audio-card,format = "i2s";
    simple-audio-card,bitclock-master = <&cpu_dai>;
    simple-audio-card,frame-master = <&cpu_dai>;

    cpu_dai: simple-audio-card,cpu {
        sound-dai = <&sai2>;
    };

    simple-audio-card,codec {
        sound-dai = <&dmic_codec>;
```

```
    };
};
```

Three nodes:

1. **`&sai2`**: enable the i.MX SAI2 peripheral. The mainline `fsl_sai.c` handles it.
2. **`dmic_codec`**: a *fake* codec node. The mainline `sound/soc/codecs/dmic.c` driver (`compatible = "dmic-codec"`) is a generic “digital microphone” placeholder, no registers, no control, just claims to be an ASoC codec DAI.
3. **`sound`**: the machine. `simple-audio-card` reads this and builds the sound card by wiring sai2’s DAI to dmic_codec’s DAI. `bitclock-master = <&cpu_dai>` tells the framework that the SoC generates BCLK.

After boot:

```
[root@pa-mini:~]# arecord -l
**** List of CAPTURE Hardware Devices ****
card 0: imxinmp441 [imx-inmp441], device 0: 308b000.sai-dmic-hifi dmic-hifi-0 []
```

The mic appears as a capture device. Now record:

```
[root@pa-mini:~]# arecord -D plughw:0,0 -f S32_LE -r 48000 -c 1 -d 5 test.wav
Recording WAVE 'test.wav' : Signed 32 bit Little Endian, Rate 48000 Hz, Mono
```

The `S32_LE` format is required because INMP441 outputs 24 bits in a 32-bit slot. ALSA can downconvert to 16-bit via `plughw` (the `plug` prefix). Writing directly to `hw:0,0` requires S32_LE.

Listen back on the host machine: it’ll be quiet (single mic, low input) but speech should be audible.

78.5 How the SAI + DMA + ALSA pipeline works

Worth understanding even when you do not write the drivers yourself:

1. **SAI** is the i.MX's I²S peripheral. The mainline `fsl_sai.c` driver:
    - Configures BCLK and LRCLK from a parent clock (the PLL4_AUDIO_DIV in our DT, at 24.576 MHz, exactly 512 × 48 kHz, a "perfect" rate for audio).
    - Programs its TX/RX FIFO settings.
    - Sets up an SDMA channel to copy RX FIFO → DDR (Ch 51).

> **DDR:** external DRAM that must be configured and trained before most software can run from it.

- Implements the ASoC `snd_soc_dai_ops`: `set_fmt` (I²S vs left-justified), `hw_params` (sample rate, channels), `trigger` (start/stop).

1. **ASoC core** binds the SAI DAI (CPU side) to the `dmic-codec` DAI (codec side). At `hw_params` time, both DAIs negotiate format and sample rate.
2. **PCM substream**: ALSA creates a substream backed by a DMA-coherent ring buffer in DDR. The SAI's SDMA channel writes into it cyclically (Ch 51.5).

> **MCU bridge:** Think of DMA like the MCU DMA controller you used for UART or SPI, but with cache coherency, scatter-gather descriptors, and kernel ownership rules added. **DMA:** Direct Memory Access. Hardware moves data to or from memory without the CPU copying each byte.

1. **User-space**: opens `/dev/snd/pcmC0D0c`, configures format/rate via ioctl, reads samples. Reads block until enough data is available. ALSA copies from the ring buffer.

In the data path: mic → BCLK timing → SD bits → SAI's RX FIFO → SDMA → DDR ring buffer → memcpy → user-space buffer. Zero CPU between mic and SDMA. One memcpy per `read()`.

78.6 A "machine driver" you might write

`simple-audio-card` handles most use cases. But for unusual topologies (multiple mics with different formats. On-the-fly clock-rate changes. DAPM widgets representing mute relays), you may write a custom machine driver. The shape:

```
/* sound/soc/fsl/my-mic-machine.c — sketch */
```

```
static struct snd_soc_dai_link my_dai = {
    .name           = "mic-link",
    .stream_name    = "mic-capture",
    .cpus           = SND_SOC_DAILINK_REGn(cpu, "30030000.sai"),
    .codecs         = SND_SOC_DAILINK_REGn(codec, "dmic-codec"),
    .platforms      = SND_SOC_DAILINK_REGn(platform, "30030000.sai"),
    .dai_fmt        = SND_SOC_DAIFMT_I2S
                    | SND_SOC_DAIFMT_NB_NF
                    | SND_SOC_DAIFMT_CBC_CFC,
    .ops            = &my_dai_ops,
};

static struct snd_soc_card my_card = {
    .name      = "imx-inmp441-custom",
    .owner     = THIS_MODULE,
    .dai_link  = &my_dai,
    .num_links = 1,
};

static int my_probe(struct platform_device *pdev)
{
    my_card.dev = &pdev->dev;
    return devm_snd_soc_register_card(&pdev->dev, &my_card);
}
```

Roughly 70 lines. Same shape as Ch 53's WM8960 machine driver, only without the codec controls, DAPM widgets, and jack-detection.

For most cases: Most projects do not need a custom machine driver. Use `simple-audio-card` in DT.

78.7 Stereo with two mics

Two INMP441s on the same bus, one strapped LR=GND (Left), one strapped LR=VDD (Right):

```
   ┌── SD ──────────────────► both mics share SD
   ├── BCLK ────────────────► both
   ├── LRCLK ───────────────► both
   ┌── INMP441 #1, LR → GND  (drives SD during left phase)
   └── INMP441 #2, LR → VDD  (drives SD during right phase)
```

Both mics monitor LRCLK. Each drives SD only during its assigned phase. The SoC sees stereo without any extra wires.

DT: change `simple-audio-card,routing` to declare two channels. The kernel doesn't need to know how many mics, that's a wiring choice.

```
[root@pa-mini:~]# arecord -D plughw:0,0 -f S32_LE -r 48000 -c 2 -d 5 stereo.wav
```

The captured file plays back in stereo on the host.

78.8 Key Notes

- An I2S microphone supplies digital samples through a clocked audio interface. ASoC joins that interface to the CPU DAI and DMA path without inventing a register-controlled codec.
- Sample width, slot width and channel selection are separate parts of the stream format. A 32-bit container does not mean the microphone has 32 meaningful audio bits.
- The clock source and microphone channel straps must agree. Two physical microphones are required for genuinely different left and right signals.
- Buffering protects against scheduling variation, while acoustic placement, enclosure and noise determine the useful captured signal.

78.9 Pitfalls

- **Wrong format.** INMP441 outputs 24-bit MSB-first I²S. Configure for S32_LE in ALSA. The 24 audio bits sit in the high 24 bits of the 32-bit slot. The bottom 8 bits are zero. S24_LE *may* work depending on ASoC version.
- **MCLK not provided.** Some I²S mics need an MCLK in addition to BCLK + LRCLK. INMP441 does *not*. ICS-43434 also doesn't. But other I²S codecs do. Verify against datasheet.
- **Master/slave mismatch.** Both SoC and mic configured as slaves → no clock generated. INMP441 is always a slave (chip can't generate clocks).
- **WS polarity wrong.** Some chips expect LRCLK = HIGH for left. Others use LOW. `simple-audio-card,format = "i2s"` defaults to "left = LOW" (LJ vs I²S subtly differ).
- **LR-strap floating.** INMP441 reads middle state. Outputs nothing or noisy data. Strap explicitly.
- **Single mic, stereo requested.** ALSA gives you a stereo stream with the second channel duplicated (or zero, depending on plug setup). For real stereo, wire two physical mics.
- **DC-offset / wind noise.** MEMS mics pick up low-frequency rumble. Apply a high-pass filter in user-space (`sox` has a `highpass 100` effect).
- **Self-noise floor not what's specified.** A 61 dBA SNR INMP441 in a quiet room measures ~33 dB SPL noise floor, anything quieter is masked. To capture whispers, choose ICS-43434 (4 dB lower noise).

> **MCU bridge:** Think of SPL like the tiny early startup code that runs from internal SRAM before DDR is usable. **SPL:** Secondary Program Loader, a tiny first U-Boot stage that fits in OCRAM and initializes DDR.

- **dmic-codec compatible string.** Must be exactly `"dmic-codec"`. If the kernel was built without `CONFIG_SND_SOC_DMIC=y`, the placeholder driver is missing and the card fails to probe.

78.10 Going deeper

- **`sound/soc/generic/simple-card.c`**: read the parsing logic to understand what `simple-audio-card` accepts.
- **`sound/soc/codecs/dmic.c`**: the dmic placeholder driver (~150 lines).
- **`sound/soc/fsl/fsl_sai.c`**: the i.MX SAI driver.
- **`Documentation/devicetree/bindings/sound/simple-card.yaml`**: DT binding reference.
- **INMP441 datasheet (TDK InvenSense)**: timing diagram, LR-strap behavior.
- **ICS-43434 datasheet**: similar, with the SNR improvement.
- **ALSA documentation** at https://www.alsa-project.org/ for advanced topics (ringbuffer tuning, sample-rate conversion).

> Next chapter: **Chapter 79: Health sensors (MAX30100 / MAX30102).** PPG-based heart-rate and SpO_2 measurement. The I^2C interface with FIFO + the user-space DSP to extract HR/SpO_2.

Chapter 79: Health sensors

What: PPG (photoplethysmography) sensors, Maxim **MAX30100** (the original) and **MAX30102** (the improved successor). Both: red + IR LED + photodiode + FIFO + I²C. They give you raw light-intensity samples from a finger (or earlobe, forehead). Your code extracts **heart rate** and **SpO₂** (blood-oxygen saturation) from those samples. Protocol, FIFO mechanics, from-scratch IIO driver, and a sketch of the HR/SpO₂ extraction algorithm. **IIO:** Industrial I/O, Linux's subsystem for sensors, ADCs, DACs, and buffered sampled data.

Why: "wellness" features (fitness bands, smart watches, baby monitors, medical IoT) all use PPG. The chip is cheap, the wiring trivial, the principle is simple, but the work is in the signal processing on the host side. This chapter covers the chip end-to-end and points at what user-space must do.

Focus: The chip delivers light-intensity samples. Your code turns those samples into heart rate and SpO₂, the chip cannot do this for you. Heart rate comes from counting peaks in the IR signal. SpO₂ comes from the ratio of red-AC/red-DC to IR-AC/IR-DC mapped through an empirical curve. The driver delivers raw 18-bit samples at 100 Hz. User-space filters, finds peaks, computes. Without good signal processing, the readings are unreliable. The chip cannot compensate for bad code on the host.

79.1 Chip comparison

	Maxim MAX30100	Maxim MAX30102	Maxim MAX30105
Channels	red + IR	red + IR	red + IR + green (particle sense)
Resolution	14-bit	18-bit	18-bit
Sample rates	50/100/167/200/400/600/800/1000 Hz	50–3200 Hz	50–3200 Hz
FIFO	16 samples	32 samples	32 samples
Sensitivity	per-LED current 0–50 mA	0–51 mA	0–51 mA

	Maxim MAX30100	Maxim MAX30102	Maxim MAX30105
Operating voltage	1.7, 2.0 V (sensor) + 3.3 V (I/O)	1.7, 2.0 V + 3.3 V	1.7, 2.0 V + 3.3 V
I²C address	0x57	0x57	0x57
Idle current	0.6 µA shutdown	0.7 µA	0.7 µA
Status	EOL, replaced by MAX30102	active	active
Volume price	$4 (older modules)	$6–8	$10–12

Pick guide:

- **MAX30102** for new designs. Better signal, larger FIFO.
- **MAX30100** for legacy / module-already-on-hand.
- **MAX30105** when you need a third LED (green for “is there a finger present?” via particle sensing).

79.2 The physics, photoplethysmography

Shine light into tissue. Some of it is absorbed by blood. The rest reflects or scatters back to a photodiode. As your heart beats, the volume of blood in capillaries fluctuates → light absorption fluctuates → photodiode current fluctuates. That AC component is the **PPG signal**.

The pulsatile signal is small, about 1 % of the DC level. Most of the light hitting the photodiode is the constant amount that passes through tissue without modulation. The pulsatile AC component is the signal we actually want.

For **SpO_2**, you measure with two wavelengths:

- **Red (660 nm)**: oxygenated hemoglobin absorbs *less* red light.
- **IR (940 nm)**: deoxygenated hemoglobin absorbs *less* IR light.

Compute:

```
R = (AC_red / DC_red) / (AC_IR / DC_IR)
SpO₂ ≈ 110 - 25·R    (empirical for fingertip PPG; calibration-dependent)
```

The “110 − 25·R” is fitted to clinical pulse-oximeter data. Different chips, different fits, Maxim publishes a recommended table for each.

For **heart rate**, you find peaks in the IR-AC signal. Peak-to-peak time × 60 = BPM. Modern algorithms use FFT-based methods or autocorrelation for robustness against motion artifacts.

79.3 Protocol, MAX30102

Register map (the highlights):

Reg	Name	Purpose
0x00	Interrupt Status 1	Bit 7 = FIFO almost full, bit 6 = new sample, bit 0 = power-ready
0x02	Interrupt Enable 1	Mask of interrupts
0x04	FIFO Write Pointer	Index where chip writes next sample
0x05	FIFO Overflow Counter	How many samples lost
0x06	FIFO Read Pointer	Reader-controlled pointer; chip ignores beyond this
0x07	FIFO Data Register	Read here to get samples
0x08	FIFO Configuration	Sample averaging, rollover, almost-full threshold
0x09	Mode Configuration	Reset, shutdown, mode (HR=0x02, SpO_2=0x03, multi-LED=0x07)
0x0A	SpO_2 Configuration	ADC range, sample rate, pulse width
0x0C	LED1 (red) Pulse Amplitude	0..255, controls LED current

Reg	Name	Purpose
0x0D	LED2 (IR) Pulse Amplitude	0..255
0xFF	Part ID	0x15 for MAX30102

Bring-up sequence

1. Write 0x09 = 0x40 → reset.
2. Wait ~10 ms.
3. Write 0x02 = 0xC0 → enable FIFO-almost-full + new-data IRQs.
4. Write 0x08 = 0x4F → average 4 samples per FIFO entry, FIFO rollover OFF, almost-full at 17 samples (=32-15).
5. Write 0x09 = 0x03 → SpO_2 mode (both LEDs).
6. Write 0x0A = 0x27 → ADC range 4096 nA, sample rate 100 Hz, pulse width 411 µs (18-bit).
7. Write 0x0C = 0x24 → LED1 (red) current ~7 mA.
8. Write 0x0D = 0x24 → LED2 (IR) current ~7 mA.
9. Wait for IRQ or poll. Each FIFO entry is 6 bytes: 3 bytes red + 3 bytes IR.

> **MCU bridge:** Think of an IRQ like an EXTI/NVIC interrupt path, except Linux splits the hard interrupt from deferred work and must share lines across drivers. **IRQ:** interrupt request, the signal path that tells the CPU or interrupt controller that hardware needs service.

Reading the FIFO

```
Host: write 0x07 to set register pointer, then read N×6 bytes back.
       N = (write_ptr - read_ptr) mod 32
```

Each 3-byte sample is *18 bits packed in 24 bits*, the top 6 bits are zero. To extract:

```
u32 red_sample = ((buf[0] << 16) | (buf[1] << 8) | buf[2]) & 0x3FFFF;
u32 ir_sample  = ((buf[3] << 16) | (buf[4] << 8) | buf[5]) & 0x3FFFF;
```

The chip increments its write pointer. You increment your read pointer (write to 0x06). FIFO empty when write_ptr == read_ptr.

79.4 How a mainline driver would work

As of v6.6, mainline does include IIO drivers, `drivers/iio/health/max30100.c` and `drivers/iio/health/max30102.c`, that handle the basics (FIFO drain via IRQ, raw red/IR/green samples to a triggered buffer). Most ecosystems still ship Arduino-style ports with the HR/SpO_2 math built in. The mainline drivers deliberately leave the DSP for user space. A complete in-kernel driver would:

1. Read part-id at probe (verify 0x15 for MAX30102).
2. Configure via DT properties (sample rate, LED current).
3. Register an IIO device with two `IIO_INTENSITY` channels (red, IR).
4. Set up triggered buffered capture using either the chip's data-ready IRQ or an hrtimer.
5. Push samples into the IIO buffer on FIFO-almost-full IRQ.

User-space then sees `/dev/iio:device0` streaming 6-byte samples at 100 Hz, processes them.

The MAX30102's data-ready IRQ pin connects to a GPIO. The driver uses `request_threaded_irq` (Ch 43) → drains the FIFO → pushes to IIO buffer (Ch 49/70). It's an IIO driver with a FIFO. Structurally identical to the IMU drivers in Ch 70–71, just with intensity channels instead of accel/gyro.

> **MCU bridge:** Think of Linux GPIO like the same pin set/reset block you used on STM32, but accessed through a kernel subsystem that owns numbering, direction, interrupts, and user-space exposure. **GPIO:** General-Purpose Input/Output, a pin controlled as a digital input, output, or interrupt source.

79.5 Writing a MAX30102 driver from scratch

We'll implement the full driver with triggered buffered IIO capture driven by the chip's data-ready IRQ. ~300 lines.

`mymax30102.c`:

```
#include <linux/init.h>
#include <linux/module.h>
#include <linux/i2c.h>
#include <linux/delay.h>
#include <linux/iio/iio.h>
```

```
#include <linux/iio/buffer.h>
#include <linux/iio/triggered_buffer.h>
#include <linux/iio/trigger_consumer.h>
#include <linux/interrupt.h>

#define REG_INT_STATUS_1  0x00
#define REG_INT_ENABLE_1  0x02
#define REG_FIFO_WR_PTR   0x04
#define REG_FIFO_OVF      0x05
#define REG_FIFO_RD_PTR   0x06
#define REG_FIFO_DATA     0x07
#define REG_FIFO_CONFIG   0x08
#define REG_MODE_CONFIG   0x09
#define REG_SPO2_CONFIG   0x0A
#define REG_LED1_PA       0x0C
#define REG_LED2_PA       0x0D
#define REG_PART_ID       0xFF

#define PART_ID_VAL       0x15

struct mymax30102 {
    struct i2c_client *client;
    struct mutex lock;
};

static int mh_init(struct mymax30102 *m)
{
    int err, part_id;

    part_id = i2c_smbus_read_byte_data(m->client, REG_PART_ID);
    if (part_id < 0) return part_id;
    if (part_id != PART_ID_VAL) return -ENODEV;

    /* Reset */
    err = i2c_smbus_write_byte_data(m->client, REG_MODE_CONFIG, 0x40);
    if (err) return err;
    msleep(10);

    /* Enable data-ready IRQ + FIFO-almost-full */
    err = i2c_smbus_write_byte_data(m->client, REG_INT_ENABLE_1, 0xC0);
    if (err) return err;

    /* FIFO config: avg=4 samples per entry, rollover OFF, almost-full at 15 */
    err = i2c_smbus_write_byte_data(m->client, REG_FIFO_CONFIG, 0x4F);
    if (err) return err;

    /* Mode: SpO2 (both LEDs) */
    err = i2c_smbus_write_byte_data(m->client, REG_MODE_CONFIG, 0x03);
    if (err) return err;

    /* SpO2 config: ADC 4096 nA, sample rate 100 Hz, pulse width 411 µs (18-bit) */
    err = i2c_smbus_write_byte_data(m->client, REG_SPO2_CONFIG, 0x27);
    if (err) return err;

    /* LED currents (~7 mA each — about right for a fingertip; reduce for earlobe) */
    err = i2c_smbus_write_byte_data(m->client, REG_LED1_PA, 0x24);
    if (err) return err;
    err = i2c_smbus_write_byte_data(m->client, REG_LED2_PA, 0x24);
    if (err) return err;
```

```
    /* Reset FIFO pointers */
    i2c_smbus_write_byte_data(m->client, REG_FIFO_WR_PTR, 0);
    i2c_smbus_write_byte_data(m->client, REG_FIFO_OVF, 0);
    i2c_smbus_write_byte_data(m->client, REG_FIFO_RD_PTR, 0);

    return 0;
}

static int mh_read_fifo_count(struct mymax30102 *m)
{
    int wr = i2c_smbus_read_byte_data(m->client, REG_FIFO_WR_PTR);
    int rd = i2c_smbus_read_byte_data(m->client, REG_FIFO_RD_PTR);
    if (wr < 0 || rd < 0) return -EIO;
    int count = wr - rd;
    if (count < 0) count += 32;
    return count;
}

static int mh_read_one_sample(struct mymax30102 *m, u32 *red, u32 *ir)
{
    u8 buf[6];
    int err = i2c_smbus_read_i2c_block_data(m->client, REG_FIFO_DATA, 6, buf);
    if (err < 0) return err;
    if (err != 6) return -EIO;
    *red = ((buf[0] << 16) | (buf[1] << 8) | buf[2]) & 0x3FFFF;
    *ir  = ((buf[3] << 16) | (buf[4] << 8) | buf[5]) & 0x3FFFF;
    return 0;
}

/* === IIO INFO_PROCESSED (one-shot read; drains entire FIFO, returns latest) === */

static int mh_read_raw(struct iio_dev *idev,
                       struct iio_chan_spec const *chan,
                       int *val, int *val2, long mask)
{
    struct mymax30102 *m = iio_priv(idev);
    u32 red, ir;
    int err, n;

    if (mask != IIO_CHAN_INFO_RAW) return -EINVAL;

    mutex_lock(&m->lock);
    n = mh_read_fifo_count(m);
    if (n <= 0) {
        mutex_unlock(&m->lock);
        return -EAGAIN;
    }
    /* Drain all but the latest */
    while (n-- > 1) mh_read_one_sample(m, &red, &ir);
    err = mh_read_one_sample(m, &red, &ir);
    mutex_unlock(&m->lock);

    if (err) return err;

    switch (chan->channel2) {
    case IIO_MOD_LIGHT_RED:  *val = red; return IIO_VAL_INT;
    case IIO_MOD_LIGHT_IR:   *val = ir;  return IIO_VAL_INT;
    }
```

```
    return -EINVAL;
}

#define MAX_CHAN(_mod, _idx) {                                          \
    .type = IIO_INTENSITY, .modified = 1, .channel2 = (_mod),           \
    .info_mask_separate = BIT(IIO_CHAN_INFO_RAW),                       \
    .scan_index = (_idx),                                               \
    .scan_type = { .sign='u', .realbits=18, .storagebits=32,            \
                   .endianness=IIO_BE },                                \
}

static const struct iio_chan_spec mh_channels[] = {
    MAX_CHAN(IIO_MOD_LIGHT_RED, 0),
    MAX_CHAN(IIO_MOD_LIGHT_IR,  1),
    IIO_CHAN_SOFT_TIMESTAMP(2),
};

static const struct iio_info mh_iio_info = {
    .read_raw = mh_read_raw,
};

/* === IRQ handler: drain FIFO and push samples to IIO buffer === */

static irqreturn_t mh_irq_thread(int irq, void *p)
{
    struct iio_dev *idev = p;
    struct mymax30102 *m = iio_priv(idev);
    int int_status, count;
    u32 red, ir;
    u32 sample[2 + 2];     /* 2 u32 channels + 8 bytes timestamp */

    mutex_lock(&m->lock);

    int_status = i2c_smbus_read_byte_data(m->client, REG_INT_STATUS_1);
    if (int_status < 0) goto out;

    /* Either FIFO_A_FULL (bit 7) or PPG_RDY (bit 6) brought us here */
    count = mh_read_fifo_count(m);

    while (count-- > 0) {
        if (mh_read_one_sample(m, &red, &ir)) break;
        sample[0] = red;
        sample[1] = ir;
        iio_push_to_buffers_with_timestamp(idev, sample,
                                           iio_get_time_ns(idev));
    }
out:
    mutex_unlock(&m->lock);
    return IRQ_HANDLED;
}

/* === Probe / Remove === */

static int mh_probe(struct i2c_client *client)
{
    struct iio_dev *idev;
    struct mymax30102 *m;
    int err;
```

```
    idev = devm_iio_device_alloc(&client->dev, sizeof(*m));
    if (!idev) return -ENOMEM;
    m = iio_priv(idev);
    m->client = client;
    mutex_init(&m->lock);

    err = mh_init(m);
    if (err) return dev_err_probe(&client->dev, err, "init failed\n");

    idev->name = "mymax30102";
    idev->info = &mh_iio_info;
    idev->modes = INDIO_DIRECT_MODE | INDIO_BUFFER_TRIGGERED;
    idev->channels = mh_channels;
    idev->num_channels = ARRAY_SIZE(mh_channels);

    err = devm_iio_triggered_buffer_setup(&client->dev, idev,
                                          NULL, NULL, NULL);
    if (err) return err;

    if (client->irq > 0) {
        err = devm_request_threaded_irq(&client->dev, client->irq, NULL,
                                        mh_irq_thread,
                                        IRQF_TRIGGER_FALLING | IRQF_ONESHOT,
                                        "mymax30102", idev);
        if (err) return err;
    }

    return devm_iio_device_register(&client->dev, idev);
}

static const struct of_device_id mh_of_match[] = {
    { .compatible = "linuxlearn,mymax30102" },
    { }
};
MODULE_DEVICE_TABLE(of, mh_of_match);

static const struct i2c_device_id mh_id[] = { { "mymax30102", 0 }, { } };
MODULE_DEVICE_TABLE(i2c, mh_id);

static struct i2c_driver mh_driver = {
    .driver = {
        .name = "mymax30102",
        .of_match_table = mh_of_match,
    },
    .probe = mh_probe,
    .id_table = mh_id,
};
module_i2c_driver(mh_driver);

MODULE_LICENSE("GPL");
```

DT:

```
&i2c1 {
    max30102@57 {
        compatible = "linuxlearn,mymax30102";
        reg = <0x57>;
        interrupt-parent = <&gpio4>;
```

```
        interrupts = <13 IRQ_TYPE_EDGE_FALLING>;     /* data-ready IRQ */
    };
};
```

Test (one-shot via sysfs):

> **sysfs:** a kernel-generated filesystem under /sys that exposes devices, drivers, and attributes.

```
[root@pa-mini:~]# insmod mymax30102.ko
[root@pa-mini:~]# # Put your finger on the sensor:
[root@pa-mini:~]# cat /sys/bus/iio/devices/iio:device0/in_intensity_red_raw
102842
[root@pa-mini:~]# cat /sys/bus/iio/devices/iio:device0/in_intensity_ir_raw
108391
[root@pa-mini:~]# # Remove finger:
[root@pa-mini:~]# cat /sys/bus/iio/devices/iio:device0/in_intensity_red_raw
3214        ← ambient light only
```

Streaming via buffer:

```
# Enable both intensity channels + timestamp
for ch in red ir; do
    echo 1 > /sys/bus/iio/devices/iio:device0/scan_elements/in_intensity_${ch}_en
done
echo 1 > /sys/bus/iio/devices/iio:device0/scan_elements/in_timestamp_en

echo 256 > /sys/bus/iio/devices/iio:device0/buffer/length
echo 1   > /sys/bus/iio/devices/iio:device0/buffer/enable

# Each sample: 4 (red u32) + 4 (ir u32) + 8 (timestamp) = 16 bytes
dd if=/dev/iio:device0 of=ppg.bin bs=16 count=3000
```

3000 samples in 30 seconds (100 Hz). Parse offline.

79.6 Extracting heart rate + SpO_2 in user-space

The driver gives raw samples. The interesting work is now:

```
import numpy as np, struct, sys
from scipy.signal import butter, filtfilt, find_peaks

# Load samples
data = np.fromfile("ppg.bin", dtype=np.uint32).reshape(-1, 4)
red = data[:, 0].astype(float)
ir  = data[:, 1].astype(float)
# (columns 2,3 are the 64-bit timestamp split as two u32; ignore for now)

# 1. Band-pass filter the IR signal: keep 0.5–4 Hz (30–240 BPM)
fs = 100.0    # sample rate
b, a = butter(4, [0.5/(fs/2), 4.0/(fs/2)], btype='band')
```

```
ir_ac = filtfilt(b, a, ir)
red_ac = filtfilt(b, a, red)

# 2. Find peaks for heart rate
peaks, _ = find_peaks(ir_ac, distance=fs * 0.4)    # min 400 ms between peaks
if len(peaks) > 1:
    intervals = np.diff(peaks) / fs   # seconds between beats
    bpm = 60.0 / intervals.mean()
    print(f"Heart rate: {bpm:.1f} BPM")

# 3. SpO2 from R-ratio
red_dc = red.mean()
ir_dc  = ir.mean()
red_ac_rms = np.sqrt(np.mean(red_ac**2))
ir_ac_rms  = np.sqrt(np.mean(ir_ac**2))
R = (red_ac_rms / red_dc) / (ir_ac_rms / ir_dc)
spo2 = 110 - 25 * R
print(f"SpO2: {spo2:.1f}% (R-ratio {R:.3f})")
```

That’s the textbook approach. Production systems add:

- **Motion artifact rejection** (accelerometer cross-correlation. Gate readings during walking).
- **Auto-gain** (adjust LED currents to keep DC within the ADC’s sweet spot).
- **Finger-detection** (if ir_dc < threshold, no finger present. Report “no signal”).
- **FFT/autocorrelation** instead of peak-finding for noisy signals.

Maxim’s reference code includes the algorithm. Apple Watch’s algorithm is patented and proprietary. For a hobby/demo: peak-finding + a 10-second running average works.

79.7 Key Notes

- The sensor supplies optical samples, while signal processing estimates pulse-related quantities. Raw red and infrared counts are not heart rate or oxygen saturation.
- Motion, ambient light, contact and perfusion affect signal quality. Invalid measurements need to remain distinguishable from a reassuring numeric result.
- FIFO framing and timestamps preserve the input to the algorithm. Lost or mismatched samples can invalidate the estimate even when the bus continues to work.
- The example is an architecture illustration, not a medically qualified instrument. An empirical formula and comparison reading do not establish clinical accuracy or suitability.

79.8 Pitfalls

> **Driver choice:** Use the in-tree, maintained driver first. Use out-of-tree, spidev, or custom-driver paths only after you accept the kernel-version maintenance cost and document who owns updates.

- **Sensor in direct sun.** Photodiode saturated by ambient IR. DC level pinned at max. No pulsatile signal. Use indoor or covered.
- **Loose finger placement.** Tiny finger movements look like enormous “pulses.” Mechanical fixturing matters, clip designs work, finger-on-flat-board doesn’t.
- **Cold fingers.** Reduced peripheral perfusion → weak AC signal. SpO_2 readings unreliable. Warm hands. Or use earlobe.
- **No DC tracking.** Naive AC extraction (just band-pass) fails when DC drifts with finger pressure. Real algorithms track DC adaptively.
- **R-ratio calibration off.** The 110-25·R formula is for one specific chip’s geometry. Use Maxim’s chip-specific table for accuracy. Don’t claim “medical-grade SpO_2” without proper calibration against a reference oximeter.
- **FIFO overflow.** If user-space drains slower than 100 Hz × 6 bytes = 600 B/s, FIFO overflows. Check REG_FIFO_OVF (0x05) and warn.
- **IRQ pin polarity.** Open-drain output, active-low. Pull-up to 3.3V required.
- **18-bit data in 24-bit container.** Top 6 bits are zero, they’re padding. Mask with 0x3FFFF or the math goes wrong.
- **No mainline IIO driver.** Multiple out-of-tree implementations exist. Quality varies. Writing your own from §79.5 is reasonable.

79.9 Going deeper

- **MAX30102 datasheet (Maxim 19-7740 Rev 1)**: full register map.
- **Maxim app note AN6409**: “PPG signal processing on MAX30102.”
- **Maxim’s reference C library** (Arduino-flavored but algorithmically clear), at `github.com/MaximIntegratedRefDesTeam/`.
- **`drivers/iio/light/`**: IIO INTENSITY-channel drivers for comparison.
- **`scipy.signal`** documentation, Butterworth, peak-finding.
- **PhysioNet**: open-access ECG/PPG signal databases for testing algorithms.

- **WHO SpO_2 measurement guidelines**: for understanding what “medical-grade” means.

> **End of Group F, Specialty sensors (Ch 77–79).** Three sensors that each represent a different driver pattern: 1-Wire’s bit-bang master + slave-driver split (Ch 77), I²S audio capture without a chip driver (Ch 78), and an IIO chip with FIFO + user-space DSP (Ch 79).

> Next chapter: **Chapter 80: External ADCs (ADS1115 / ADS1256 / MCP3008 / AD7606).** When the SoC’s internal ADC isn’t accurate enough, fast enough, or has the wrong number of channels.

Chapter 80: External ADCs

> **regmap:** a kernel helper that wraps register reads and writes over I2C, SPI, or MMIO. **MCU bridge:** Think of regmap like a typed wrapper around your read_reg() and write_reg() helpers, with caching, locking, and bus differences handled centrally.

> **What:** four external analog-to-digital converters spanning the price/precision spectrum: **TI ADS1115** (16-bit, I²C, programmable-gain, 4-channel), **TI ADS1256** (24-bit, SPI, ultra-low-noise, 8-channel), **Microchip MCP3008** (10-bit, SPI, cheap, 8-channel), **Analog Devices AD7606** (16-bit, 8-channel *simultaneous-sampling*). For each: protocol, the IIO ADC channel model, and a from-scratch ADS1115 IIO driver. Plus ratiometric measurement (load cells, RTDs), the trick that cancels reference-voltage error. **IIO:** Industrial I/O, Linux's subsystem for sensors, ADCs, DACs, and buffered sampled data.
>
> **Why:** the i.MX6ULL's internal ADCs (two 12-bit SAR blocks, each multiplexing up to 10 pins, ~1 MS/s aggregate) are ±a few LSB noisy and share the SoC's noisy power rails. For precision measurement, a load-cell scale, a 4-20 mA industrial loop, a thermocouple, simultaneous 3-phase power sampling, you need an external ADC with a clean reference, more bits, or true simultaneity. Knowing which external ADC fits saves you from chasing noise that the silicon will never let you remove.
>
> **Focus:** Bits, speed, channels, and simultaneity are independent design axes. **ADS1115** is high-bit, slow, and multiplexed. **MCP3008** is low-bit, medium-speed, and cheap. **ADS1256** is very-high-bit, low-noise, and slow. **AD7606** is high-bit, fast, and *simultaneous* (all channels sampled at the same instant, critical for phase measurement). Pick by which axis matters most for your application.

80.1 Chip comparison

	TI ADS1115	TI ADS1256	Microchip MCP3008	ADI AD7606
Resolution	16-bit	24-bit	10-bit	16-bit
Channels	4 single / 2 diff	8 single / 4 diff	8 single	8 simultaneous
Max sample rate	860 SPS	30 kSPS	200 kSPS	200 kSPS/ch (all at once)

	TI ADS1115	TI ADS1256	Microchip MCP3008	ADI AD7606
Interface	I²C	SPI	SPI	parallel or SPI
Built-in PGA	yes (2/3×, 16×)	yes (1×, 64×)	none	none (±10 V / ±5 V range pins)
Reference	internal	external	VDD (ratiometric)	internal 2.5 V
ENOB (effective bits)	~15.5	~22	~9.5	~15.5
Simultaneous?	no (mux)	no (mux)	no (mux)	**yes**
Volume price	$3–5	$10–15	$2–3	$20–30
Mainline driver	`ti-ads1015.c`	`ti-ads1256` (recent)	`mcp320x.c`	`ad7606.c`

Pick guide:

- **MCP3008**: cheapest 8-channel. 10-bit is fine for “read a potentiometer / light sensor / battery divider.”
- **ADS1115**: 16-bit, PGA, I²C, the standard precision choice. Load cells, 4-20 mA loops.
- **ADS1256**: 24-bit, lowest noise, strain gauges, lab instruments, weigh scales needing sub-gram resolution.
- **AD7606**: when channels must sample *at the same instant*, 3-phase power analysis, vibration with multiple accelerometers, phase-sensitive detection.

80.2 Why not use the SoC’s internal ADC?

The i.MX6ULL has **2 ADC blocks** (ADC1, ADC2), each a 12-bit SAR with up to 10 external input pins muxed in. They’re fine for “read a battery voltage divider” but limited:

- **12-bit / ~10 ENOB**: ~3 mV resolution on a 3.3 V range. A load cell’s signal might be 1 mV full-scale, invisible.

- **Shared noisy rails**: the ADC reference is the SoC's analog supply, polluted by digital switching. The bottom 2 bits are noise.
- **Only two simultaneous conversions**: even though each ADC block has many input pins, only one channel per block samples at a time, so a multi-sensor product is bottlenecked at the *block* count, not the channel count.
- **No PGA**: can't amplify a small signal before conversion.
- **No simultaneity within a block**: SAR ADCs mux. Channels on the same ADC are sampled at different instants.

An external ADC with a clean reference, a PGA, and more bits lets you see signals the SoC's internal ADC cannot. The cost is a chip + an I²C/SPI transaction per sample.

80.3 Protocol, ADS1115

ADS1115 has just 4 registers, addressed by a 1-byte pointer:

Pointer	Register	Purpose
0x00	Conversion	16-bit last result (read-only)
0x01	Config	mux, PGA, mode, data rate, comparator
0x02	Lo_thresh	comparator low threshold
0x03	Hi_thresh	comparator high threshold

To take a single-shot conversion of channel 0 (AIN0 vs GND):

```
1. Write Config (0x01) = a 16-bit word:
   bit 15:    OS = 1 (start single conversion)
   bits 14:12: MUX = 100 (AIN0 vs GND)
   bits 11:9:  PGA = 010 (±2.048 V full scale)
   bit 8:     MODE = 1 (single-shot)
   bits 7:5:  DR = 100 (128 SPS)
   bits 4:0:  comparator config (disabled = 00011)
   = 0xC383 typical
2. Poll bit 15 of Config (OS bit): 1 = busy, 0 = done.
   (Or wait the conversion time: 1/DR seconds.)
3. Read Conversion register (0x00): 16-bit signed result.
4. Convert: voltage = raw × (full_scale / 32768).
   For PGA ±2.048 V: voltage = raw × 2.048 / 32768 = raw × 62.5 µV.
```

Each register is 16 bits, big-endian on the wire. The MUX field selects which input pair. To switch channels you re-write Config, the chip handles one conversion at a time, since it is multiplexed.

The PGA is the killer feature: ±0.256 V full-scale range gives 7.8 µV/LSB, read a thermocouple directly.

80.4 How the mainline `ti-ads1015` driver works

Source: `drivers/iio/adc/ti-ads1015.c` (~1000 lines). Covers ADS1015 (12-bit) and ADS1115 (16-bit), and the 4-channel variants.

The driver registers an IIO device with one channel per input configuration (4 single-ended + 4 differential = 8 logical channels). Each channel has a `scale_available` listing the PGA ranges, and a `sampling_frequency_available` listing the data rates.

```
/* Simplified */
static int ads1015_read_raw(struct iio_dev *indio_dev,
                            struct iio_chan_spec const *chan,
                            int *val, int *val2, long mask)
{
    struct ads1015_data *data = iio_priv(indio_dev);

    switch (mask) {
    case IIO_CHAN_INFO_RAW:
        mutex_lock(&data->lock);
        /* Set the mux to this channel, set PGA, trigger conversion */
        ads1015_set_conv_mode(data, ADS1015_SINGLESHOT);
        regmap_update_bits(data->regmap, ADS1015_CFG_REG,
                           ADS1015_CFG_MUX_MASK,
                           chan->address << ADS1015_CFG_MUX_SHIFT);
        /* Wait for conversion */
        ads1015_get_adc_result(data, chan->address, val);
        mutex_unlock(&data->lock);
        return IIO_VAL_INT;

    case IIO_CHAN_INFO_SCALE:
        /* Return the volts-per-LSB for the current PGA setting */
        *val = ads1015_fullscale_range[data->channel_data[chan->address].pga];
        *val2 = chan->scan_type.realbits - 1;   /* 2^15 */
        return IIO_VAL_FRACTIONAL_LOG2;

    case IIO_CHAN_INFO_SAMP_FREQ:
        *val = ads1015_data_rate[data->channel_data[chan->address].data_rate];
        return IIO_VAL_INT;
    }
    return -EINVAL;
}
```

The driver also supports **buffered/continuous mode** with a data-ready IRQ (the ALERT/RDY pin), and a software-comparator that can fire an IIO event when the input crosses a threshold.

User-space sees:

```
in_voltage0_raw            (AIN0 vs GND, single-ended)
in_voltage0-voltage1_raw   (AIN0 vs AIN1, differential)
in_voltage_scale
in_voltage_scale_available  (the PGA ranges)
sampling_frequency
sampling_frequency_available
```

80.5 Writing an ADS1115 IIO driver from scratch

`myads1115.c`:

```
#include <linux/init.h>
#include <linux/module.h>
#include <linux/i2c.h>
#include <linux/delay.h>
#include <linux/iio/iio.h>

#define REG_CONVERSION  0x00
#define REG_CONFIG      0x01

/* Config field positions */
#define CFG_OS_SINGLE   (1 << 15)
#define CFG_MUX_SHIFT   12
#define CFG_PGA_2_048   (0x2 << 9)   /* ±2.048 V */
#define CFG_MODE_SINGLE (1 << 8)
#define CFG_DR_128SPS   (0x4 << 5)
#define CFG_COMP_DIS    0x0003

struct myads {
    struct i2c_client *client;
    struct mutex lock;
};

static int ma_read16(struct myads *m, u8 reg, u16 *val)
{
    int r = i2c_smbus_read_word_swapped(m->client, reg);   /* big-endian */
    if (r < 0) return r;
    *val = r;
    return 0;
}

static int ma_write16(struct myads *m, u8 reg, u16 val)
{
    return i2c_smbus_write_word_swapped(m->client, reg, val);
}

/* mux 4..7 = AIN0..AIN3 vs GND (single-ended) */
static int ma_read_channel(struct myads *m, int chan, s16 *out)
{
    u16 config, status;
    int err, retries = 50;

    config = CFG_OS_SINGLE
```

```
            | ((4 + chan) << CFG_MUX_SHIFT)
            | CFG_PGA_2_048
            | CFG_MODE_SINGLE
            | CFG_DR_128SPS
            | CFG_COMP_DIS;

    err = ma_write16(m, REG_CONFIG, config);
    if (err) return err;

    /* Poll OS bit (bit 15) — 1 = busy, 0 = done */
    do {
        usleep_range(1000, 2000);
        err = ma_read16(m, REG_CONFIG, &status);
        if (err) return err;
    } while (!(status & CFG_OS_SINGLE) && retries--);
    /* note: OS reads 1 when conversion is DONE in single-shot; check datasheet */

    err = ma_read16(m, REG_CONVERSION, (u16 *)out);
    return err;
}

static int ma_read_raw(struct iio_dev *idev,
                       struct iio_chan_spec const *chan,
                       int *val, int *val2, long mask)
{
    struct myads *m = iio_priv(idev);
    s16 raw;
    int err;

    switch (mask) {
    case IIO_CHAN_INFO_RAW:
        mutex_lock(&m->lock);
        err = ma_read_channel(m, chan->channel, &raw);
        mutex_unlock(&m->lock);
        if (err) return err;
        *val = raw;
        return IIO_VAL_INT;
    case IIO_CHAN_INFO_SCALE:
        /* ±2.048 V over 2^15 = 62.5 µV/LSB */
        *val = 0; *val2 = 62500;     /* nano-volts per LSB → IIO_VAL_INT_PLUS_NANO */
        return IIO_VAL_INT_PLUS_NANO;
    }
    return -EINVAL;
}

#define ADS_CHAN(idx) {                                                 \
    .type = IIO_VOLTAGE, .indexed = 1, .channel = (idx),                \
    .info_mask_separate = BIT(IIO_CHAN_INFO_RAW),                       \
    .info_mask_shared_by_type = BIT(IIO_CHAN_INFO_SCALE),               \
    .scan_index = (idx),                                                \
    .scan_type = { .sign='s', .realbits=16, .storagebits=16 },          \
}

static const struct iio_chan_spec ma_channels[] = {
    ADS_CHAN(0), ADS_CHAN(1), ADS_CHAN(2), ADS_CHAN(3),
};

static const struct iio_info ma_iio_info = {
    .read_raw = ma_read_raw,
```

```
};

static int ma_probe(struct i2c_client *client)
{
    struct iio_dev *idev;
    struct myads *m;
    s16 test;
    int err;

    idev = devm_iio_device_alloc(&client->dev, sizeof(*m));
    if (!idev) return -ENOMEM;
    m = iio_priv(idev);
    m->client = client;
    mutex_init(&m->lock);

    /* Sanity: a read should succeed */
    err = ma_read_channel(m, 0, &test);
    if (err) return dev_err_probe(&client->dev, err, "test read failed\n");

    idev->name = "myads1115";
    idev->info = &ma_iio_info;
    idev->modes = INDIO_DIRECT_MODE;
    idev->channels = ma_channels;
    idev->num_channels = ARRAY_SIZE(ma_channels);

    return devm_iio_device_register(&client->dev, idev);
}

static const struct of_device_id ma_of_match[] = {
    { .compatible = "linuxlearn,myads1115" },
    { }
};
MODULE_DEVICE_TABLE(of, ma_of_match);

static const struct i2c_device_id ma_id[] = { { "myads1115", 0 }, { } };
MODULE_DEVICE_TABLE(i2c, ma_id);

static struct i2c_driver ma_driver = {
    .driver = {
        .name = "myads1115",
        .of_match_table = ma_of_match,
    },
    .probe = ma_probe,
    .id_table = ma_id,
};
module_i2c_driver(ma_driver);

MODULE_LICENSE("GPL");
```

DT:

```
&i2c1 {
    ads1115@48 {
        compatible = "linuxlearn,myads1115";
        reg = <0x48>;
    };
};
```

Test:

```
[root@pa-mini:~]# insmod myads1115.ko
[root@pa-mini:~]# cat /sys/bus/iio/devices/iio:device0/in_voltage0_raw
16384
[root@pa-mini:~]# cat /sys/bus/iio/devices/iio:device0/in_voltage_scale
0.000062
[root@pa-mini:~]# # → 16384 × 62.5 µV = 1.024 V on AIN0
```

Driver is ~180 lines, gives 16-bit single-ended reads on 4 channels via IIO. The full mainline driver adds differential channels, runtime PGA selection, continuous mode, comparator events.

80.6 MCP3008, SPI, cheap, 10-bit

MCP3008 we already met in Ch 47 (SPI drivers). Its protocol: a 3-byte SPI transaction encodes the channel and returns a 10-bit result. The mainline driver `drivers/iio/adc/mcp320x.c` covers the MCP320x/MCP330x family.

```
/* The 3-byte command from Ch 47 */
u8 tx[3] = { 0x01, (u8)(0x80 | (channel << 4)), 0x00 };
u8 rx[3];
/* result = ((rx[1] & 0x03) << 8) | rx[2]; */
```

DT:

```
&ecspi3 {
    adc@0 {
        compatible = "microchip,mcp3008";
        reg = <0>;
        spi-max-frequency = <1000000>;
        vref-supply = <&reg_3v3>;       /* ratiometric: scale = vref / 1024 */
    };
};
```

`vref-supply` is important: MCP3008 is *ratiometric*, its full-scale equals VREF (typically VDD). The driver reads the regulator's voltage and computes the scale. So `in_voltage_scale` = VREF / 1024.

80.7 ADS1256, 24-bit, low-noise

> **Driver choice:** Use the in-tree, maintained driver first. Use out-of-tree, spidev, or custom-driver paths only after you accept the kernel-version maintenance cost and document who owns updates.

ADS1256 is the precision SPI ADC: 24-bit, programmable gain to 64×, 30 kSPS max, but with an excellent noise floor (~22 ENOB at low data rates). For a load cell measuring a few-mV signal, this is the chip.

Protocol: SPI commands (RDATA, WREG, RREG, SYNC), a register set for gain/rate/mux, and a DRDY pin that goes low when a conversion is ready. The mainline driver is `drivers/iio/adc/ti-ads1256.c` (recent kernels) or out-of-tree variants.

The complexity: ADS1256 has a strict timing relationship between DRDY, the command, and the data read. You must wait for DRDY, issue RDATA, wait t6 (~6.5 × master clock period), then clock out 3 bytes. Get the timing wrong and you read stale or corrupt data.

80.8 AD7606, simultaneous sampling

AD7606 is unique here: **all 8 channels sample at the same instant**. A single CONVST (convert-start) pulse triggers all 8 sample-and-hold circuits simultaneously. Then you read the 8 results sequentially (parallel bus or SPI).

Why this matters: for 3-phase power measurement, you need voltage and current of all three phases captured at the *same* moment to compute true power and phase angle. A multiplexed ADC samples them microseconds apart, at 50/60 Hz that's a fraction of a degree of phase error, but for harmonics and transients it matters.

```
CONVST↓ ──► all 8 S/H freeze simultaneously
BUSY goes high during conversion (~4 µs)
BUSY↓ ──► read 8 × 16-bit results over parallel bus or SPI
```

The mainline driver `drivers/iio/adc/ad7606.c` uses a GPIO for CONVST, a GPIO IRQ for BUSY, and either parallel-bus or SPI read. The DT specifies range pins, oversampling pins, etc. This is a more involved driver because of the parallel-bus option and the strict CONVST/BUSY handshake.

> **MCU bridge:** Think of Linux GPIO like the same pin set/reset block you used on STM32, but accessed through a kernel subsystem that owns numbering, direction, interrupts, and user-space exposure. **GPIO:** General-Purpose Input/Output, a pin controlled as a digital input, output, or interrupt source.

80.9 Ratiometric measurement, the noise-cancellation trick

For sensors that are *resistive dividers excited by the ADC's reference*, load cells, RTDs, potentiometers, there's a useful trick: make the measurement *ratiometric*.

A load cell is a Wheatstone bridge. Excite it with voltage Vexc. The output is `Vout = Vexc × (sensitivity × load)`. If you also use Vexc as the ADC's reference, then:

```
ADC_reading = Vout / Vref = Vout / Vexc = sensitivity × load
```

The Vexc *cancels*. Any noise or drift in the excitation voltage also cancels. The reading depends only on the load, not on the absolute excitation. This is why precision scales use ratiometric ADCs (ADS1256, HX711): the reference and the bridge excitation are the same rail.

To do this: wire the ADC's REF+ / REF− to the same rail that excites the bridge. In DT, the `vref-supply` points at the excitation regulator. The ADC's `scale` becomes meaningless in absolute volts but the *ratio* is rock-stable.

80.10 Key Notes

- Resolution, conversion rate, input range and simultaneous sampling are independent selection criteria. A larger nominal bit count does not ensure more useful information.
- A multiplexed ADC observes channels at different times. Phase-sensitive measurements require an acquisition architecture that accounts for that difference.
- The analog front end, reference, grounding and settling time are part of the conversion contract. Software scale cannot correct clipping or an unsafe input voltage.
- Ratiometric measurement cancels a shared excitation/reference variation only when the circuit actually has that relationship. It is not a general substitute for a stable reference.

80.11 Pitfalls

- **ADS1115 OS-bit polarity confusion.** In single-shot: writing OS=1 *starts* a conversion. Reading OS=1 means *idle/done*, OS=0 means *converting*. Easy to get backwards. The datasheet's "Operational Status" description is counterintuitive, read it carefully.
- **PGA range vs input voltage.** If your signal exceeds the PGA range, the reading clips at ±32767. ADS1115's ±0.256 V range clips anything above 256 mV. Pick the range to fit your signal with margin.
- **Input above VDD.** ADS1115 inputs must be within GND−0.3 V to VDD+0.3 V. A 5 V signal into a 3.3 V-powered ADS1115 damages it. Use a divider.
- **Ratiometric misunderstanding.** Ratiometric works only when the sensor is excited *by the same reference*. A 4-20 mA loop is *not* ratiometric (it's a current source). Use absolute reference there.

- **MCP3008 vref vs vdd.** MCP3008 has separate VDD and VREF pins. If VREF < VDD, the usable input range is limited to VREF. Tie them together for full-range.
- **SPI clock too fast for MCP3008.** Max 3.6 MHz at 5 V, 1.35 MHz at 2.7 V. Exceeding it gives noise. Stay at 1 MHz to be safe.
- **ADS1256 DRDY timing.** Must wait for DRDY low before reading. Must respect t6 delay after RDATA command. Race conditions give corrupt data.
- **AD7606 oversampling pins.** OS[2:0] pins set hardware oversampling. If left floating, behavior is undefined. Strap or GPIO them.
- **Grounding.** External ADCs need a clean analog ground separate from digital ground, joined at one point (star ground). A noisy ground negates the precision you paid for.

80.12 Going deeper

- **`drivers/iio/adc/ti-ads1015.c`**: production ADS1015/1115 driver.
- **`drivers/iio/adc/mcp320x.c`**: MCP3008 family.
- **`drivers/iio/adc/ad7606.c`**: simultaneous-sampling ADC with parallel + SPI variants.
- **ADS1115 datasheet (TI SBAS444)**: config register bit-fields. OS-bit semantics.
- **ADS1256 datasheet (TI SBAS288)**: timing diagrams for the DRDY/RDATA handshake.
- **AD7606 datasheet (ADI)**: CONVST/BUSY handshake.
- **TI app note SLYT423**: “How delta-sigma ADCs work” (for understanding ADS1256’s 24-bit precision).
- **`Documentation/devicetree/bindings/iio/adc/`**: DT bindings for each.

> Next chapter: **Chapter 81: External DACs + clock generators.** Analog *output* (MCP4725, AD5663) and programmable clock generation (Si5351), the inverse of this chapter, plus the clk-framework integration.

Chapter 81: External DACs + clock generators

> **What:** analog *output* and clock *generation*, the inverse of Chapter 80. **Microchip MCP4725** (12-bit I²C DAC with EEPROM), **Analog Devices AD5663** (16-bit dual SPI DAC), and **SiLabs Si5351** (programmable 3-output clock generator). For each: protocol, the IIO `out_voltage` model (DACs) or the `clk` framework (Si5351), and a from-scratch MCP4725 IIO DAC driver. **IIO:** Industrial I/O, Linux's subsystem for sensors, ADCs, DACs, and buffered sampled data.
>
> **Why:** embedded systems sometimes need to *produce* an analog voltage (control a VCO, set a programmable power-supply setpoint, generate a waveform) or *produce a precise clock* (drive an SDR mixer, clock an external ADC, generate a reference frequency). The SoC has neither a DAC nor flexible clock outputs in most cases. These three chips cover the common cases, and they introduce two new frameworks: IIO's output channels and the kernel's `clk` provider model.
>
> **Focus:** A DAC is an IIO channel that flows out instead of in. A clock generator is a `clk` provider. DAC: `out_voltage0_raw` is *writable*. Writing it sets the output voltage. Clock gen: the chip registers as a `clk` in the kernel clock tree, and other devices (or user-space via `/sys/.../clk`) consume its output. These are two different frameworks. Both are useful.

81.1 Chip comparison

	Microchip MCP4725	ADI AD5663	SiLabs Si5351A
Function	12-bit DAC	dual 16-bit DAC	3-output clock generator
Interface	I²C	SPI	I²C
Channels	1	2	3 (clock outputs)
Output range	0, VDD	0, VREF	2.5 kHz, 200 MHz
Settling time	6 µs	5 µs	n/a

	Microchip MCP4725	ADI AD5663	SiLabs Si5351A
EEPROM (power-on default)	yes	no	optional (Si5351B)
I²C / SPI address	0x60–0x67	(SPI CS)	0x60 / 0x61
Volume price	$1–2	$5–8	$1.50–3
Mainline driver	`mcp4725.c`	`ad5446.c` family	`clk-si5351.c`

Pick guide:

- **MCP4725**: cheap single-channel analog output. Set a control voltage, generate slow waveforms.
- **AD5663**: dual-channel, 16-bit, faster, stereo control, precision setpoints.
- **Si5351**: programmable clocks for RF, SDR, clocking external chips. Up to 200 MHz, arbitrary frequencies.

81.2 MCP4725, the simplest DAC

MCP4725 takes a 12-bit value and outputs `V = (value / 4096) × VDD`. Two write modes:

- **Fast write**: 2 bytes, just the 12-bit value. Updates output immediately.
- **Write + EEPROM**: 3 bytes, value + store to EEPROM (so the chip powers up at this value next time).

Fast-write protocol:

```
   START | 0xC0 | (D11..D8 in low nibble) | (D7..D0) | STOP
   Actually: byte0 = (mode << 4) | (D11..D8); byte1 = D7..D0
   Fast mode: byte0 = 0x0_ where high nibble bits select mode (00 = normal)
```

Specifically, fast write:

```
u8 buf[2];
buf[0] = (value >> 8) & 0x0F;     /* upper 4 bits, mode = normal (00) */
buf[1] = value & 0xFF;            /* lower 8 bits */
i2c_master_send(client, buf, 2);
```

Write with EEPROM (3 bytes):

```
u8 buf[3];
buf[0] = 0x60;                    /* command: write DAC + EEPROM */
buf[1] = (value >> 4) & 0xFF;     /* D11..D4 */
buf[2] = (value << 4) & 0xF0;     /* D3..D0 in upper nibble */
i2c_master_send(client, buf, 3);
```

(The two formats pack the 12 bits differently, fast write splits 4+8, EEPROM write splits 8+4. Datasheet figures 6-1 and 6-2.)

81.3 IIO output channels

DACs use IIO too, but with `output = 1` channels. The key difference: the channel exposes a *writable* `out_voltageN_raw`.

```
static const struct iio_chan_spec dac_channels[] = {
    {
        .type = IIO_VOLTAGE,
        .indexed = 1,
        .channel = 0,
        .output = 1,                /* ← this makes it an output */
        .info_mask_separate = BIT(IIO_CHAN_INFO_RAW)
                            | BIT(IIO_CHAN_INFO_SCALE),
    },
};
```

And the driver implements `write_raw` (in addition to or instead of `read_raw`):

```
static int dac_write_raw(struct iio_dev *idev,
                         struct iio_chan_spec const *chan,
                         int val, int val2, long mask)
{
    /* val is the new raw DAC code; program it into the chip */
}
```

User-space:

```
[root@pa-mini:~]# echo 2048 > /sys/bus/iio/devices/iio:device0/out_voltage0_raw
[root@pa-mini:~]# # → output = 2048/4096 × VDD = VDD/2 = 1.65 V
```

81.4 Writing an MCP4725 IIO DAC driver from scratch

`mymcp4725.c`:

```
#include <linux/init.h>
#include <linux/module.h>
#include <linux/i2c.h>
#include <linux/iio/iio.h>

struct mymcp {
```

```
    struct i2c_client *client;
    struct mutex lock;
    u16 last_value;          /* cache for read-back */
    u32 vref_mV;             /* reference (= VDD) in mV */
};

/* Fast write: 2 bytes, mode = normal */
static int mc_set(struct mymcp *m, u16 value)
{
    u8 buf[2];
    int err;

    value &= 0x0FFF;
    buf[0] = (value >> 8) & 0x0F;
    buf[1] = value & 0xFF;

    err = i2c_master_send(m->client, buf, 2);
    if (err != 2) return err < 0 ? err : -EIO;
    m->last_value = value;
    return 0;
}

static int mc_read_raw(struct iio_dev *idev,
                       struct iio_chan_spec const *chan,
                       int *val, int *val2, long mask)
{
    struct mymcp *m = iio_priv(idev);

    switch (mask) {
    case IIO_CHAN_INFO_RAW:
        *val = m->last_value;     /* return the cached DAC code */
        return IIO_VAL_INT;
    case IIO_CHAN_INFO_SCALE:
        /* scale = VDD / 4096, expressed as mV-per-LSB */
        *val = m->vref_mV;
        *val2 = 12;               /* 2^12 */
        return IIO_VAL_FRACTIONAL_LOG2;
    }
    return -EINVAL;
}

static int mc_write_raw(struct iio_dev *idev,
                        struct iio_chan_spec const *chan,
                        int val, int val2, long mask)
{
    struct mymcp *m = iio_priv(idev);
    int err;

    if (mask != IIO_CHAN_INFO_RAW) return -EINVAL;
    if (val < 0 || val > 4095) return -EINVAL;

    mutex_lock(&m->lock);
    err = mc_set(m, val);
    mutex_unlock(&m->lock);
    return err;
}

static const struct iio_chan_spec mc_channels[] = {
    {
```

```
        .type = IIO_VOLTAGE,
        .indexed = 1,
        .channel = 0,
        .output = 1,
        .info_mask_separate = BIT(IIO_CHAN_INFO_RAW) | BIT(IIO_CHAN_INFO_SCALE),
    },
};

static const struct iio_info mc_iio_info = {
    .read_raw  = mc_read_raw,
    .write_raw = mc_write_raw,
};

static int mc_probe(struct i2c_client *client)
{
    struct iio_dev *idev;
    struct mymcp *m;
    int err;

    idev = devm_iio_device_alloc(&client->dev, sizeof(*m));
    if (!idev) return -ENOMEM;
    m = iio_priv(idev);
    m->client = client;
    m->vref_mV = 3300;        /* assume VDD = 3.3 V; could read a regulator */
    mutex_init(&m->lock);

    of_property_read_u32(client->dev.of_node, "vref-millivolt", &m->vref_mV);

    /* Set output to mid-scale at startup */
    err = mc_set(m, 2048);
    if (err) return dev_err_probe(&client->dev, err, "initial set failed\n");

    idev->name = "mymcp4725";
    idev->info = &mc_iio_info;
    idev->modes = INDIO_DIRECT_MODE;
    idev->channels = mc_channels;
    idev->num_channels = ARRAY_SIZE(mc_channels);

    return devm_iio_device_register(&client->dev, idev);
}

static const struct of_device_id mc_of_match[] = {
    { .compatible = "linuxlearn,mymcp4725" },
    { }
};
MODULE_DEVICE_TABLE(of, mc_of_match);

static const struct i2c_device_id mc_id[] = { { "mymcp4725", 0 }, { } };
MODULE_DEVICE_TABLE(i2c, mc_id);

static struct i2c_driver mc_driver = {
    .driver = {
        .name = "mymcp4725",
        .of_match_table = mc_of_match,
    },
    .probe = mc_probe,
    .id_table = mc_id,
};
module_i2c_driver(mc_driver);
```

```
MODULE_LICENSE("GPL");
```

DT:

```
&i2c1 {
    dac@60 {
        compatible = "linuxlearn,mymcp4725";
        reg = <0x60>;
        vref-millivolt = <3300>;
    };
};
```

Test:

```
[root@pa-mini:~]# insmod mymcp4725.ko
[root@pa-mini:~]# echo 0 > /sys/bus/iio/devices/iio:device0/out_voltage0_raw
[root@pa-mini:~]# # measure output: 0 V
[root@pa-mini:~]# echo 4095 > /sys/bus/iio/devices/iio:device0/out_voltage0_raw
[root@pa-mini:~]# # measure output: ~3.3 V
[root@pa-mini:~]# echo 2048 > /sys/bus/iio/devices/iio:device0/out_voltage0_raw
[root@pa-mini:~]# # measure output: ~1.65 V
```

Generate a slow sine wave from user-space:

```
#!/bin/sh
while true; do
  for i in $(seq 0 36); do
    # 12-bit sine: 2048 + 2047*sin(i*10°)
    v=$(awk "BEGIN{print int(2048 + 2047*sin($i*0.1745))}")
    echo $v > /sys/bus/iio/devices/iio:device0/out_voltage0_raw
    usleep 5000
  done
done
```

~5 Hz sine on a scope. For faster/cleaner waveforms, write a kernel-side waveform generator or use the AD5663 (faster SPI).

Driver is ~120 lines. Gives writable IIO output. Mainline `mcp4725.c` adds EEPROM persistence, power-down modes, and read-back of the actual chip register.

81.5 AD5663, dual 16-bit SPI DAC

AD5663 is two 16-bit DACs in one package, SPI-controlled. Each write is a 24-bit SPI frame: 8 command/address bits + 16 data bits.

```
   bits 23:22: reserved
   bits 21:19: command (e.g., 011 = write + update DAC)
   bits 18:17: address (which DAC: 00=A, 01=B, 11=both)
   bits 16:0:  16-bit data
```

The mainline `drivers/iio/dac/ad5446.c` covers the AD5446/AD5663 family. Two `out_voltage` channels (`out_voltage0_raw`, `out_voltage1_raw`).

Use AD5663 over MCP4725 when you need: 16-bit precision, two synchronized channels, faster update (SPI at 10+ MHz vs I²C at 400 kHz), or external precision reference.

81.6 Si5351, programmable clock generator, and the clk framework

Si5351 is different from the DACs: it produces *clocks*, not voltages. It has a 25 MHz crystal, two internal PLLs, and three output dividers, letting it synthesize almost any frequency from 2.5 kHz to 200 MHz on each of three outputs.

This integrates with the kernel's **clk framework**, the same framework that manages the SoC's internal clock tree (Ch 13, Ch 25). The Si5351 registers as a **clock provider**. Its outputs become entries in the kernel clock tree that other devices can consume.

How clocks flow

```
   25 MHz XTAL ──► Si5351 PLL_A ──► Output divider 0 ──► CLK0 (e.g., 100 MHz)
                          PLL_A ──► Output divider 1 ──► CLK1 (e.g., 13.56 MHz)
                          PLL_B ──► Output divider 2 ──► CLK2 (e.g., 27 MHz)
```

The frequency synthesis: `f_out = f_xtal × (PLL_mult) / (output_divider)`. The PLL multiplier is a fractional value (a + b/c), giving fine resolution. The math to derive (a, b, c, divider) for a target frequency is non-trivial, the driver does it.

> **MCU bridge:** Think of a PLL like the clock multiplier setup you used on STM32, but with more clock roots, gates, and consumers that Linux later needs to describe.
> **PLL:** Phase-Locked Loop, a clock block that multiplies a reference clock to create faster clocks.

DT

```
&i2c1 {
    si5351: clock-generator@60 {
        compatible = "silabs,si5351a";
        reg = <0x60>;
        #clock-cells = <1>;
        clocks = <&ref25m>;            /* 25 MHz crystal input */
        clock-names = "xtal";

        /* Per-output config via child nodes */
        clkout0 {
            reg = <0>;
            silabs,drive-strength = <8>;        /* mA */
            silabs,multisynth-source = <0>;     /* PLL A */
```

```
            silabs,clock-source = <0>;
            clock-frequency = <100000000>;       /* 100 MHz */
        };
        clkout2 {
            reg = <2>;
            silabs,multisynth-source = <1>;     /* PLL B */
            clock-frequency = <27000000>;        /* 27 MHz */
        };
    };
};

/* A device consuming the Si5351's CLK0 */
some_device {
    clocks = <&si5351 0>;          /* phandle + output index */
    clock-names = "ref-clock";
};
```

The consuming device, in its driver, does:

```
struct clk *clk = devm_clk_get(&pdev->dev, "ref-clock");
clk_prepare_enable(clk);
unsigned long rate = clk_get_rate(clk);    /* 100000000 */
clk_set_rate(clk, 48000000);                /* retune to 48 MHz */
```

So the Si5351's outputs become first-class kernel clocks, the same `clk_get` / `clk_set_rate` API used for SoC clocks. A consumer driver doesn't know or care that the clock comes from an external I²C chip.

User-space access

```
[root@pa-mini:~]# cat /sys/kernel/debug/clk/clk_summary | grep si5351
   si5351_clkout0    1   1   100000000   ...
   si5351_clkout2    1   1    27000000   ...
```

The `clk_summary` debugfs file lists every clock in the tree, including the Si5351's outputs.

Mainline driver

`drivers/clk/clk-si5351.c` implements frequency-synthesis math, PLL configuration, and clock-provider registration. The architectural question is how a requested output frequency becomes a valid combination of PLL and divider settings, and how changing one output affects the others. The maintained driver is the reference for that device-specific logic. The simpler skeleton below illustrates the clock-framework contract, not a replacement for the Si5351 driver.

A from-scratch clk provider *skeleton* (for a fixed-frequency case) looks like:

```
static unsigned long my_clk_recalc_rate(struct clk_hw *hw, unsigned long parent)
{
    struct my_clk *c = to_my_clk(hw);
    return c->current_rate;
```

```
}

static int my_clk_set_rate(struct clk_hw *hw, unsigned long rate,
                           unsigned long parent)
{
    struct my_clk *c = to_my_clk(hw);
    /* Compute and program the chip's PLL + divider registers for `rate` */
    c->current_rate = rate;
    return 0;
}

static const struct clk_ops my_clk_ops = {
    .recalc_rate = my_clk_recalc_rate,
    .set_rate    = my_clk_set_rate,
    .round_rate  = my_clk_round_rate,
};

/* In probe: */
struct clk_init_data init = {
    .name = "my-clkout0",
    .ops = &my_clk_ops,
    .parent_names = (const char *[]){ "xtal" },
    .num_parents = 1,
};
c->hw.init = &init;
err = devm_clk_hw_register(&client->dev, &c->hw);
of_clk_add_hw_provider(client->dev.of_node, of_clk_hw_simple_get, &c->hw);
```

That registers a clk that consumers can `clk_get`. The hard part, the actual PLL math in `set_rate`, is chip-specific.

81.7 Key Notes

- A DAC is an analog-output provider, while a clock generator is a clock provider. They share a control bus but expose different contracts to Linux consumers.
- Output range, reference stability and load drive determine useful DAC behavior. A numeric write alone does not guarantee the requested voltage at the load.
- Volatile output updates and persistent power-on settings have different lifetimes and endurance costs. EEPROM programming should not be hidden inside every waveform update.
- A programmable clock’s requested rate must satisfy its divider and PLL constraints. Consumers need the actual achievable rate and a clear ownership policy for changes.

81.8 Pitfalls

- **MCP4725 bit-packing.** Fast-write packs 12 bits as 4+8. EEPROM-write packs as 8+4. Mixing them up produces a value 16× off. Datasheet figures 6-1 / 6-2.

- **DAC output loading.** MCP4725 can source or sink only a few mA. Driving a low-impedance load directly causes the output voltage to drop. Buffer with an op-amp follower for current.
- **DAC output range = VDD.** MCP4725's full-scale is VDD, not a fixed reference. If VDD is noisy (shared digital rail), the output is noisy. Use a clean rail or AD5663 with external reference.
- **EEPROM write endurance.** MCP4725's EEPROM is rated ~1M cycles. Don't write EEPROM on every output change (use fast-write). Reserve EEPROM-write for "set the power-on default."
- **Si5351 PLL constraints.** Each PLL must run between 600 and 900 MHz internally. The output dividers are 4 to 2048. Not every target frequency is achievable on every output. The driver's solver picks the closest valid combination. Verify the actual rate via `clk_get_rate`.
- **Si5351 output drive vs load.** Output drive strength (2/4/6/8 mA) must match the load (50 Ω termination etc.). Wrong drive = distorted clock or ringing.
- **Clock consumer ordering.** If a device's `clk_get` happens before the Si5351 driver probes, it gets `-EPROBE_DEFER`. The kernel retries. Usually fine. But circular clock dependencies deadlock.
- **#clock-cells mismatch.** Si5351 has `#clock-cells = <1>` (the output index is the cell). A consumer referencing `<&si5351>` without the index fails. Always `<&si5351 N>`.

81.9 Going deeper

- **`drivers/iio/dac/mcp4725.c`**: production MCP4725/4726 driver with EEPROM + power-down.
- **`drivers/iio/dac/ad5446.c`**: AD5446/AD5663 family.
- **`drivers/clk/clk-si5351.c`**: the Si5351 clk provider. Study the `si5351_*_set_rate` math.
- **`Documentation/driver-api/clk.rst`**: the clk framework provider/consumer model.
- **`drivers/clk/clk.c`**: the clk core.
- **MCP4725 datasheet (Microchip)**: write-format figures.
- **AD5663 datasheet (ADI)**: 24-bit SPI frame layout.
- **Si5351 datasheet + AN619 (SiLabs)**: "Manually generating an Si5351 register map", the PLL math reference.
- **`Documentation/devicetree/bindings/clock/silabs,si5351.yaml`**: DT binding.

> **End of Group G, Analog conversion & clock generation (Ch 80–81).** ADCs in (Ch 80), DACs + clocks out (Ch 81). Both wrap into IIO (`out_voltage`) or the clk framework. The Si5351 is also our first encounter with writing a kernel *clock provider*.

> Next chapter: **Chapter 82: RGB parallel LCD on LCDIF.** Group H (Displays) opens with the i.MX6ULL's native parallel-RGB display interface: panel timings, `panel-simple`, the DRM bridge, and bringing up a real ATK panel.

Chapter 82: RGB parallel LCD on LCDIF

> **What:** “dumb” RGB-parallel TFT panels driven by the i.MX6ULL’s **LCDIF** controller. Three panels compared: **ATK4384** (4.3” 480×272), **ATK7016** (7” 1024×600), **ATK10261** (10.1” 1280×800). The panel has no controller, it just accepts pixel clock + sync + 24 data lines. The “driver” is therefore a *timing description* + the DRM panel framework. We cover panel timings deeply, how `panel-simple` works internally, and how to add a custom panel (two ways: DT-only via `panel-dpi`, or a real `drm_panel` driver from scratch).
>
> **Why:** parallel-RGB is the standard HMI display interface for i.MX6ULL. A parallel panel has no frame buffer of its own. The SoC streams pixels at the pixel clock and refreshes the glass 60 times a second. Get the timings right and the panel works. Get one porch wrong and the image rolls, tears, or stays blank. This chapter is mostly about *reading a panel datasheet’s timing table and translating it to DT*.
>
> **Focus:** A working panel needs a pixel clock, six porch numbers, and three polarities. That is the entire job. The LCDIF is a raster generator: it outputs HSYNC/VSYNC/DE/PCLK and 24 RGB bits, scanning left-to-right, top-to-bottom, forever. The panel datasheet gives you the exact timing. You transcribe it into a `panel-timing` DT node. There is no negotiation. The numbers must match the glass.

82.1 Panel comparison

	ATK4384	ATK7016	ATK10261
Size	4.3”	7”	10.1”
Resolution	480×272	1024×600	1280×800
Pixel clock	9 MHz	51.2 MHz	71.1 MHz
Color depth	24-bit (RGB888)	24-bit	24-bit
Interface	RGB parallel (DPI)	RGB parallel	RGB parallel
Backlight	LED, PWM dimming	LED, PWM	LED, PWM

	ATK4384	ATK7016	ATK10261
Touch	resistive or capacitive option	capacitive (GT911)	capacitive (GT911)
i.MX6ULL feasible?	✓ easily	✓ (near LCDIF max)	⚠ 71 MHz pclk exceeds safe LCDIF range (~70 MHz)

i.MX6ULL LCDIF pixel-clock ceiling is ~70 MHz in practice. The ATK4384 (9 MHz) and ATK7016 (51 MHz) are comfortable. ATK10261 at 71 MHz is at the limit. It usually works but you may need to drop to a lower refresh rate.

82.2 How a parallel-RGB panel works

A parallel TFT has no memory. The SoC must *continuously* send every pixel, every frame, forever:

```
for each frame (60×/sec):
  VSYNC pulse
  vertical back porch (blank lines)
  for each visible line (e.g., 272 or 600):
    HSYNC pulse
    horizontal back porch (blank pixels)
    for each visible pixel (e.g., 480 or 1024):
      drive R[7:0], G[7:0], B[7:0]; assert DE; tick PCLK
    horizontal front porch (blank pixels)
  vertical front porch (blank lines)
```

The signals:

- **PCLK** (pixel clock): one tick per pixel. The panel latches RGB on each edge.
- **HSYNC**: marks the start of each horizontal line.
- **VSYNC**: marks the start of each frame.
- **DE** (data enable): high during the visible region. The panel only shows pixels when DE is high.
- **R/G/B[7:0]**: 24 data lines (18 for RGB666 panels).

The “porches” are blanking intervals. They are a CRT legacy, the electron beam needed time to fly back. LCDs kept the timing model. The panel datasheet specifies exact porch widths.

The timing math

Total horizontal pixels = hsync + hbackporch + hactive + hfrontporch. Total vertical lines = vsync + vbackporch + vactive + vfrontporch.

```
pixel_clock = (h_total) × (v_total) × refresh_rate
```

For ATK7016 (1024×600 @ 60 Hz):

```
h_total = 1 (hsync) + 46 (hbp) + 1024 (active) + 210 (hfp) = 1281
v_total = 1 (vsync) + 23 (vbp) + 600 (active) + 22 (vfp) = 646
pclk = 1281 × 646 × 60 ≈ 49.6 MHz
```

Close to the datasheet's 51.2 MHz (the difference is the exact refresh rate they spec). The point: **the porches and pixel clock are interdependent**, and the panel datasheet gives the authoritative numbers.

82.3 Device tree, the timing description

The LCDIF side:

```
&lcdif {
    pinctrl-names = "default";
    pinctrl-0 = <&pinctrl_lcdif_dat>, <&pinctrl_lcdif_ctrl>;
    display = <&display0>;
    status = "okay";

    display0: display0 {
        bits-per-pixel = <24>;
        bus-width = <24>;

        display-timings {
            native-mode = <&timing_atk7016>;
            timing_atk7016: timing0 {
                clock-frequency = <51200000>;
                hactive = <1024>;
                vactive = <600>;
                hfront-porch = <210>;
                hback-porch = <46>;
                hsync-len = <1>;
                vback-porch = <23>;
                vfront-porch = <22>;
                vsync-len = <1>;
                hsync-active = <0>;   /* HSYNC active-low */
                vsync-active = <0>;   /* VSYNC active-low */
                de-active = <1>;      /* DE active-high */
                pixelclk-active = <0>; /* latch on falling edge */
            };
        };
    };
};
```

For ATK4384 substitute: `clock-frequency = <9000000>; hactive = <480>; vactive = <272>; hfront-porch = <5>; hback-porch = <40>; hsync-len = <1>; vback-porch = <8>; vfront-porch = <8>; vsync-len = <1>;`.

These numbers come directly from the panel datasheet's "AC Timing Characteristics" / "Display Timing" table. Transcribing them is the entire job.

The modern DRM way: panel-simple + of_graph

The above uses the older `display-timings` binding. The modern DRM/KMS approach uses `panel-simple` and the `of_graph` port/endpoint binding:

```
&lcdif {
    status = "okay";
    port {
        lcdif_out: endpoint {
            remote-endpoint = <&panel_in>;
        };
    };
};

panel {
    compatible = "panel-dpi";
    backlight = <&backlight>;
    enable-gpios = <&gpio5 11 GPIO_ACTIVE_HIGH>;
    power-supply = <&reg_lcd_3v3>;

    width-mm = <154>;
    height-mm = <86>;

    panel-timing {
        clock-frequency = <51200000>;
        hactive = <1024>;
        vactive = <600>;
        hfront-porch = <210>;
        hback-porch = <46>;
        hsync-len = <1>;
        vback-porch = <23>;
        vfront-porch = <22>;
        vsync-len = <1>;
        hsync-active = <0>;
        vsync-active = <0>;
        de-active = <1>;
        pixelclk-active = <0>;
    };

    port {
        panel_in: endpoint {
            remote-endpoint = <&lcdif_out>;
        };
    };
};
```

`panel-dpi` is `panel-simple`'s generic "describe-your-own-timing" variant. The `port`/`endpoint` graph wires the LCDIF output to the panel input.

82.4 How `panel-simple` works internally

Source: `drivers/gpu/drm/panel/panel-simple.c` (~5000 lines, but most is the panel *database*).

The driver is a **`drm_panel`** provider. It has:

1. A giant database of known panels (vendor+part → hardcoded `display_mode` struct).
2. A generic `panel-dpi` / `panel-lvds` path that reads timings from DT.
3. Power-sequencing logic: enable regulator, wait, assert enable-gpio, wait, turn on backlight.

> **MCU bridge:** Think of Linux GPIO like the same pin set/reset block you used on STM32, but accessed through a kernel subsystem that owns numbering, direction, interrupts, and user-space exposure. **GPIO:** General-Purpose Input/Output, a pin controlled as a digital input, output, or interrupt source.

```
/* Simplified */
struct panel_simple {
    struct drm_panel base;
    struct regulator *supply;
    struct gpio_desc *enable_gpio;
    struct backlight_device *backlight;
    const struct panel_desc *desc;   /* timings + delays */
};

static int panel_simple_prepare(struct drm_panel *panel)
{
    struct panel_simple *p = to_panel_simple(panel);

    regulator_enable(p->supply);
    if (p->desc->delay.prepare)
        msleep(p->desc->delay.prepare);
    gpiod_set_value_cansleep(p->enable_gpio, 1);
    if (p->desc->delay.enable)
        msleep(p->desc->delay.enable);
    return 0;
}

static int panel_simple_get_modes(struct drm_panel *panel,
                                  struct drm_connector *connector)
{
    struct panel_simple *p = to_panel_simple(panel);
    struct drm_display_mode *mode = drm_mode_create(connector->dev);

    /* Copy the timing from desc (database) or DT into a drm_display_mode */
    drm_display_mode_from_videomode(&p->vm, mode);
    mode->type = DRM_MODE_TYPE_DRIVER | DRM_MODE_TYPE_PREFERRED;
    drm_mode_probed_add(connector, mode);
    return 1;
}
```

```
static const struct drm_panel_funcs panel_simple_funcs = {
    .prepare    = panel_simple_prepare,
    .enable     = panel_simple_enable,    /* turns on backlight */
    .disable    = panel_simple_disable,
    .unprepare  = panel_simple_unprepare,
    .get_modes  = panel_simple_get_modes,
};
```

The LCDIF DRM driver (`mxsfb`) is the **CRTC + encoder**. The panel is the **connector's** mode source. At modeset time, DRM:

1. Calls `panel->prepare` (power on).
2. Reads modes via `panel->get_modes`.
3. Programs the LCDIF's timing registers from the chosen mode.
4. Calls `panel->enable` (backlight on).
5. Streams pixels.

82.5 Adding a custom panel, three approaches

Approach 1: DT-only via panel-dpi (no code)

If your panel isn't in the database, use `compatible = "panel-dpi"` and put the timing in DT (§82.3). Zero code. **This is the right answer for 90 % of custom panels.**

Approach 2: Add to the panel-simple database (small patch)

If you want a clean `compatible = "vendor,part"` and intend to upstream:

```
/* In panel-simple.c */
static const struct drm_display_mode atk7016_mode = {
    .clock = 51200,
    .hdisplay = 1024,
    .hsync_start = 1024 + 210,
    .hsync_end = 1024 + 210 + 1,
    .htotal = 1024 + 210 + 1 + 46,
    .vdisplay = 600,
    .vsync_start = 600 + 22,
    .vsync_end = 600 + 22 + 1,
    .vtotal = 600 + 22 + 1 + 23,
    .flags = DRM_MODE_FLAG_NHSYNC | DRM_MODE_FLAG_NVSYNC,
};

static const struct panel_desc atk7016 = {
    .modes = &atk7016_mode,
    .num_modes = 1,
    .bpc = 8,
    .size = { .width = 154, .height = 86 },
    .bus_format = MEDIA_BUS_FMT_RGB888_1X24,
```

```
    .bus_flags = DRM_BUS_FLAG_DE_HIGH | DRM_BUS_FLAG_PIXDATA_DRIVE_NEGEDGE,
};

/* Add to of_device_id table: */
{ .compatible = "alientek,atk7016", .data = &atk7016 },
```

Notice the DRM `drm_display_mode` uses *cumulative* values (hsync_start = hdisplay + front_porch) rather than the DT's separate porch fields. The conversion is simple but easy to get wrong.

Approach 3: A full drm_panel driver from scratch

Some panels need custom power sequencing or init commands. This is rare for dumb RGB panels but common for panels with an init controller. For those, write a `drm_panel` driver:

```
#include <drm/drm_panel.h>
#include <drm/drm_modes.h>
#include <linux/platform_device.h>
#include <linux/gpio/consumer.h>
#include <linux/regulator/consumer.h>

struct mypanel {
    struct drm_panel panel;
    struct gpio_desc *enable_gpio;
    struct regulator *supply;
    struct backlight_device *backlight;
};

#define to_mypanel(p) container_of(p, struct mypanel, panel)

static const struct drm_display_mode mypanel_mode = {
    .clock = 51200,
    .hdisplay = 1024, .hsync_start = 1234, .hsync_end = 1235, .htotal = 1281,
    .vdisplay = 600,  .vsync_start = 622,  .vsync_end = 623,  .vtotal = 646,
    .flags = DRM_MODE_FLAG_NHSYNC | DRM_MODE_FLAG_NVSYNC,
};

static int mypanel_prepare(struct drm_panel *panel)
{
    struct mypanel *p = to_mypanel(panel);
    int err = regulator_enable(p->supply);
    if (err) return err;
    msleep(20);
    gpiod_set_value_cansleep(p->enable_gpio, 1);
    msleep(50);
    return 0;
}

static int mypanel_unprepare(struct drm_panel *panel)
{
    struct mypanel *p = to_mypanel(panel);
    gpiod_set_value_cansleep(p->enable_gpio, 0);
    regulator_disable(p->supply);
    return 0;
}
```

```
static int mypanel_get_modes(struct drm_panel *panel, struct drm_connector *connector)
{
    struct drm_display_mode *mode = drm_mode_duplicate(connector->dev, &mypanel_mode);
    if (!mode) return -ENOMEM;
    mode->type = DRM_MODE_TYPE_DRIVER | DRM_MODE_TYPE_PREFERRED;
    drm_mode_probed_add(connector, mode);
    connector->display_info.width_mm = 154;
    connector->display_info.height_mm = 86;
    return 1;
}

static const struct drm_panel_funcs mypanel_funcs = {
    .prepare   = mypanel_prepare,
    .unprepare = mypanel_unprepare,
    .get_modes = mypanel_get_modes,
};

static int mypanel_probe(struct platform_device *pdev)
{
    struct mypanel *p;
    int err;

    p = devm_kzalloc(&pdev->dev, sizeof(*p), GFP_KERNEL);
    if (!p) return -ENOMEM;

    p->supply = devm_regulator_get(&pdev->dev, "power");
    if (IS_ERR(p->supply)) return PTR_ERR(p->supply);

    p->enable_gpio = devm_gpiod_get(&pdev->dev, "enable", GPIOD_OUT_LOW);
    if (IS_ERR(p->enable_gpio)) return PTR_ERR(p->enable_gpio);

    drm_panel_init(&p->panel, &pdev->dev, &mypanel_funcs, DRM_MODE_CONNECTOR_DPI);

    err = drm_panel_of_backlight(&p->panel);   /* finds backlight from DT */
    if (err) return err;

    drm_panel_add(&p->panel);
    platform_set_drvdata(pdev, p);
    return 0;
}

static void mypanel_remove(struct platform_device *pdev)
{
    struct mypanel *p = platform_get_drvdata(pdev);
    drm_panel_remove(&p->panel);
}

static const struct of_device_id mypanel_of_match[] = {
    { .compatible = "linuxlearn,mypanel" },
    { }
};
MODULE_DEVICE_TABLE(of, mypanel_of_match);

static struct platform_driver mypanel_driver = {
    .driver = {
        .name = "linuxlearn-mypanel",
        .of_match_table = mypanel_of_match,
    },
```

```
    .probe = mypanel_probe,
    .remove = mypanel_remove,
};
module_platform_driver(mypanel_driver);

MODULE_LICENSE("GPL");
```

DT:

```
panel {
    compatible = "linuxlearn,mypanel";
    power-supply = <&reg_lcd_3v3>;
    enable-gpios = <&gpio5 11 GPIO_ACTIVE_HIGH>;
    backlight = <&backlight>;
    port {
        panel_in: endpoint {
            remote-endpoint = <&lcdif_out>;
        };
    };
};
```

About 120 lines. It registers a `drm_panel` with the timing and power sequencing. The LCDIF DRM driver finds the panel through the of_graph link and uses its mode. For a *dumb* panel this is overkill (use panel-dpi), but for a panel that needs an init sequence (e.g., a panel with an embedded controller requiring SPI commands before it accepts RGB, covered in Ch 83), this is the structure you extend.

82.6 Backlight + power sequencing

A panel needs:

1. Logic power (3.3 V), `power-supply`.
2. An enable/standby GPIO, `enable-gpios`.
3. Backlight (LED, PWM-dimmed), `backlight` phandle to a `pwm-backlight` node (Ch 48).

> **MCU bridge:** Think of Linux PWM like an MCU timer output channel, except the driver exposes period, duty cycle, polarity, and enable state through a subsystem.
> **PWM:** Pulse-Width Modulation, a timer output whose duty cycle controls average power or encodes timing.

The power sequence matters: most panels want logic power *before* enabling, and the backlight *last* (so you don't show garbage during init). The `prepare`/`enable` split in `drm_panel_funcs` enforces this, `prepare` powers logic, `enable` turns on backlight.

```
backlight: backlight {
    compatible = "pwm-backlight";
    pwms = <&pwm1 0 5000000 0>;
```

```
    brightness-levels = <0 4 8 16 32 64 128 255>;
    default-brightness-level = <6>;
    power-supply = <&reg_lcd_vled>;
};
```

82.7 User-space

> **Privilege boundary:** $ means normal user. # or sudo means root and can change host or target state. After a privileged command, verify the expected device, service, or file appears before continuing. Roll back by undoing the config change or stopping the service you just enabled.

After bring-up:

```
[root@pa-mini:~]# cat /sys/class/graphics/fb0/modes
U:1024x600p-60

[root@pa-mini:~]# fbset
mode "1024x600-60"
    geometry 1024 600 1024 600 32
    timings 19531 46 210 23 22 1 1
endmode

[root@pa-mini:~]# cat /dev/urandom > /dev/fb0      # noise on screen
[root@pa-mini:~]# modetest -M mxsfb -s <connector_id>:1024x600   # DRM modeset test
```

For a Qt app: `./app -platform linuxfb`. For backlight: `echo 4 > /sys/class/backlight/backlight/brightness`.

82.8 Key Notes

- A parallel RGB panel consumes a continuously timed pixel stream from LCDIF. Active pixels, blanking intervals and polarities must agree with its electrical specification.
- Pixel clock and memory bandwidth constrain the complete mode. Resolution alone does not determine whether the controller can drive a panel reliably.
- Panel preparation, pixel enable and backlight enable are different stages. Their ordering avoids displaying unstable data during power transitions.
- Framebuffer pixel format and physical bus width are different properties. Existing DRM panel support is useful only when it accurately describes the fitted assembly.

82.9 Pitfalls

- **One porch wrong = rolling/torn image.** The classic symptom. Re-check every porch value against the datasheet.
- **Polarity wrong.** `hsync-active`, `vsync-active`, `de-active`, `pixelclk-active` must match the panel. Inverted DE = black screen. Inverted pixelclk = data latched on wrong edge = garbled.
- **Pixel clock exceeds LCDIF max.** ~70 MHz ceiling on i.MX6ULL. Above this, the LCDIF can't generate stable timing. Use a lower-resolution or lower-refresh mode.
- **Backlight forgotten.** "Display broken!", backlight off. Always test in daylight first. The LCD content is there even with backlight off (faintly visible).
- **Pinmux clash.** LCDIF uses 24 data + 4 control = 28 pins. Conflicts with other peripherals muxed onto those pins → garbled or no output. Audit pinmux.
- **bits-per-pixel vs bus-width.** A 24-bit framebuffer on an 18-bit (RGB666) bus needs dithering or truncation. Set `bus-width = <18>` for RGB666 panels.
- **enable-gpio polarity.** Active-high vs active-low. Wrong → panel stays in standby, blank.
- **Power sequencing race.** Backlight on before logic power stabilises → flash of garbage. Use prepare/enable split.
- **DE-only vs sync mode.** Some panels use DE-only timing (ignore HSYNC/VSYNC, rely on DE). Set `de-active` and ensure the panel's mode-select pins agree.

82.10 Going deeper

- **`drivers/gpu/drm/mxsfb/`**: the i.MX LCDIF DRM driver.
- **`drivers/gpu/drm/panel/panel-simple.c`**: the panel database + panel-dpi/lvds generic paths.
- **`Documentation/devicetree/bindings/display/panel/`**: panel bindings. `panel-dpi.yaml`, `simple-panel.yaml`.
- **`include/drm/drm_panel.h`**: the drm_panel API.
- **`Documentation/gpu/drm-kms.rst`**: CRTC/encoder/connector/panel model.
- **ATK7016 / ATK4384 datasheets**: the timing tables.
- **`drivers/video/of_display_timing.c`**: how DT `display-timings` are parsed.

> Next chapter: **Chapter 83: SPI LCD.** Smart panels (ST7789, ILI9341) with an embedded controller and frame buffer, driven over SPI with command/data framing. The DRM “tiny” driver framework and `mipi-dbi`.

Chapter 83: SPI LCD

> **PWM:** Pulse-Width Modulation, a timer output whose duty cycle controls average power or encodes timing. **MCU bridge:** Think of Linux PWM like an MCU timer output channel, except the driver exposes period, duty cycle, polarity, and enable state through a subsystem.

> **What:** “smart” TFT panels with an embedded controller and frame buffer, driven over SPI. Two controllers compared: **Sitronix ST7789** (240×240 / 240×320), **Ilitek ILI9341** (240×320). Unlike the dumb parallel panels in Ch 82, these have their own RAM. You send an init sequence and pixel data over SPI. The controller refreshes the glass on its own. We cover the MIPI-DBI command model, the DRM **tiny** driver framework (`mipi-dbi`), and a from-scratch DRM tiny driver for ST7789.
>
> **Why:** SPI LCDs are cheap and small. They need 4–5 wires versus 28 for parallel, and they appear in smartwatches, thermostats, handheld instruments, and hobby gadgets. The trade-off is bandwidth. A 240×240 16-bit frame is 115 KB. At 40 MHz SPI that takes ~23 ms, about 40 fps for a full refresh. For static or partial-update UIs, that is plenty.
>
> **Focus:** the MIPI-DBI command/data model and partial updates. The controller speaks a standardized command set (MIPI Display Bus Interface): `0x2A` set column range, `0x2B` set row range, `0x2C` write pixels. A D/C (data/command) GPIO distinguishes command bytes from data bytes. Send only the changed rectangle. That keeps refresh fast enough. **MCU bridge:** Think of Linux GPIO like the same pin set/reset block you used on STM32, but accessed through a kernel subsystem that owns numbering, direction, interrupts, and user-space exposure. **GPIO:** General-Purpose Input/Output, a pin controlled as a digital input, output, or interrupt source.

83.1 Controller comparison

	Sitronix ST7789	Ilitek ILI9341	Sitronix ST7735
Resolution	240×240 / 240×320	240×320	128×160
Color	16/18-bit (RGB565/666)	16/18-bit	16-bit
Interface	SPI (3/4-wire), parallel	SPI, parallel	SPI

	Sitronix ST7789	Ilitek ILI9341	Sitronix ST7735
Frame buffer	yes (internal GRAM)	yes	yes
Max SPI clock	~62 MHz	~10 MHz (read), ~62 MHz (write)	~15 MHz
Command set	MIPI-DBI-ish	MIPI-DBI-ish	MIPI-DBI-ish
Volume price	$3–6	$4–8	$2–4
Mainline driver	`drm/tiny/st7789v` (via panel-mipi-dbi)	`drm/tiny/ili9341.c`	`drm/tiny/st7735r.c`

Pick guide:

- **ST7789**: modern default. Common on 1.3”–2” round and square modules.
- **ILI9341**: classic 2.4”–2.8” modules. Slightly slower SPI.
- **ST7735**: tiny 0.96”–1.8” displays.

83.2 The MIPI-DBI command model

These controllers descend from the MIPI **DBI** (Display Bus Interface) Type C spec. Communication is command + parameters + pixel data, with a **D/C** (Data/Command) GPIO selecting which:

```
D/C = 0 → the byte is a COMMAND
D/C = 1 → the byte is DATA (command parameter or pixel)
```

Wiring:

```
i.MX6ULL                ST7789
────────                ──────
SPI MOSI  ──────────►  SDA (data in)
SPI SCLK  ──────────►  SCL
SPI CS    ──────────►  CS
GPIO      ──────────►  DC (data/command select)
GPIO      ──────────►  RESET (active-low)
GPIO/PWM  ──────────►  BLK (backlight)
3.3V      ──────────►  VCC
GND       ──────────►  GND
```

A 3-wire mode also exists. The D/C bit becomes a 9th bit per byte, so there is no DC pin. Most SPI controllers cannot generate the 9-bit frame, so 4-wire (a separate DC GPIO) is the standard.

Key commands

Command	Hex	Parameters	Purpose
SWRESET	0x01	none	Software reset
SLPOUT	0x11	none	Sleep out (wake)
COLMOD	0x3A	1 byte	Pixel format (0x55 = RGB565, 0x66 = RGB666)
MADCTL	0x36	1 byte	Memory access control (rotation, RGB/BGR order)
CASET	0x2A	4 bytes	Column address range (x_start, x_end)
RASET	0x2B	4 bytes	Row address range (y_start, y_end)
RAMWR	0x2C	pixel data	Write pixels into the addressed window
DISPON	0x29	none	Display on
INVON	0x21	none	Invert (many ST7789 modules need this)

Drawing a pixel rectangle

```
1. CASET (0x2A) with x_start, x_end          (D/C=0 for cmd, D/C=1 for the 4 param bytes)
2. RASET (0x2B) with y_start, y_end
3. RAMWR (0x2C)                                (D/C=0)
4. stream (x_end-x_start+1) × (y_end-y_start+1) pixels  (D/C=1)
   each pixel = 2 bytes (RGB565: RRRRRGGG GGGBBBBB)
```

The controller auto-increments its internal address pointer as you stream pixels, wrapping within the window. So a full-screen update is: set window to (0,0)-(239,239), RAMWR, stream 57600 pixels.

83.3 The DRM “tiny” framework

For these small SPI displays, DRM has a dedicated lightweight subsystem: `drivers/gpu/drm/tiny/`. It provides `mipi_dbi`, a helper library that implements the CASET/RASET/RAMWR dance, dirty-rectangle tracking, and the DRM plumbing (CRTC, connector, framebuffer). A tiny driver only needs to provide the chip’s init sequence.

```
drivers/gpu/drm/tiny/
├── mi0283qt.c     (ILI9341-based)
├── ili9341.c
├── ili9486.c
├── st7586.c
├── st7735r.c
├── st7789v.c       (via panel-mipi-dbi)
├── repaper.c       (e-paper — Ch 85)
└── ...
drivers/gpu/drm/drm_mipi_dbi.c    ← the shared mipi_dbi helper
```

The `mipi_dbi` helper exposes:

- `mipi_dbi_spi_init()`: set up the SPI + DC-GPIO transport.
- `mipi_dbi_command(dbi, cmd, params...)`: send a command + parameters.
- `mipi_dbi_buf_copy()` + the `fb_dirty` callback, the partial-update path that sends only changed pixels.
- `mipi_dbi_dev_init()`: register the DRM device with a given mode.

83.4 Writing an ST7789 DRM tiny driver from scratch

Goal: a working DRM driver exposing `/dev/fb0` (via DRM fbdev emulation) for a 240×240 ST7789. ~200 lines, leveraging the `mipi_dbi` helper for the heavy lifting.

`myst7789.c`:

```
#include <linux/module.h>
#include <linux/spi/spi.h>
#include <linux/gpio/consumer.h>
#include <linux/delay.h>
#include <linux/property.h>

#include <drm/drm_atomic_helper.h>
#include <drm/drm_drv.h>
#include <drm/drm_fbdev_generic.h>
#include <drm/drm_gem_atomic_helper.h>
#include <drm/drm_gem_dma_helper.h>
```

```
#include <drm/drm_managed.h>
#include <drm/drm_mipi_dbi.h>
#include <drm/drm_modeset_helper.h>

/* The display's native timing — for an SPI display, the "mode" is just
 * the pixel dimensions; there are no porches (the controller self-refreshes). */
static const struct drm_display_mode myst7789_mode = {
    DRM_SIMPLE_MODE(240, 240, 28, 28),    /* 240×240, ~28×28 mm */
};

/* The init + pipe-enable callback: runs once when the display is turned on */
static void myst7789_pipe_enable(struct drm_simple_display_pipe *pipe,
                                 struct drm_crtc_state *crtc_state,
                                 struct drm_plane_state *plane_state)
{
    struct mipi_dbi_dev *dbidev = drm_to_mipi_dbi_dev(pipe->crtc.dev);
    struct mipi_dbi *dbi = &dbidev->dbi;
    int ret, idx;

    if (!drm_dev_enter(pipe->crtc.dev, &idx))
        return;

    /* Hardware reset via the reset GPIO (mipi_dbi handles it) */
    mipi_dbi_hw_reset(dbi);

    /* --- The ST7789 init sequence (from the datasheet / module vendor) --- */
    mipi_dbi_command(dbi, MIPI_DCS_SOFT_RESET);
    msleep(150);
    mipi_dbi_command(dbi, MIPI_DCS_EXIT_SLEEP_MODE);
    msleep(10);

    /* COLMOD = RGB565 (16-bit) */
    mipi_dbi_command(dbi, MIPI_DCS_SET_PIXEL_FORMAT, 0x55);

    /* MADCTL: orientation + BGR/RGB order */
    mipi_dbi_command(dbi, MIPI_DCS_SET_ADDRESS_MODE, 0x00);

    /* Porch, gate, voltage control — module-specific magic from the datasheet */
    mipi_dbi_command(dbi, 0xB2, 0x0C, 0x0C, 0x00, 0x33, 0x33);  /* PORCTRL */
    mipi_dbi_command(dbi, 0xB7, 0x35);                           /* GCTRL */
    mipi_dbi_command(dbi, 0xBB, 0x19);                           /* VCOMS */
    mipi_dbi_command(dbi, 0xC0, 0x2C);                           /* LCMCTRL */
    mipi_dbi_command(dbi, 0xC2, 0x01);                           /* VDVVRHEN */
    mipi_dbi_command(dbi, 0xC3, 0x12);                           /* VRHS */
    mipi_dbi_command(dbi, 0xC4, 0x20);                           /* VDVS */
    mipi_dbi_command(dbi, 0xC6, 0x0F);                           /* FRCTRL2 */
    mipi_dbi_command(dbi, 0xD0, 0xA4, 0xA1);                     /* PWCTRL1 */

    /* Most cheap ST7789 modules need display inversion ON */
    mipi_dbi_command(dbi, MIPI_DCS_ENTER_INVERT_MODE);

    mipi_dbi_command(dbi, MIPI_DCS_SET_DISPLAY_ON);
    msleep(50);

    /* mipi_dbi's helper flushes the framebuffer to the panel */
    mipi_dbi_enable_flush(dbidev, crtc_state, plane_state);

    drm_dev_exit(idx);
}
```

```
static const struct drm_simple_display_pipe_funcs myst7789_pipe_funcs = {
    .enable    = myst7789_pipe_enable,
    .disable   = mipi_dbi_pipe_disable,
    .update    = mipi_dbi_pipe_update,
};

DEFINE_DRM_GEM_DMA_FOPS(myst7789_fops);

static struct drm_driver myst7789_driver = {
    .driver_features    = DRIVER_GEM | DRIVER_MODESET | DRIVER_ATOMIC,
    .fops               = &myst7789_fops,
    DRM_GEM_DMA_DRIVER_OPS_VMAP,
    .name               = "myst7789",
    .desc               = "ST7789 SPI panel (linuxlearn)",
    .date               = "20260101",
    .major              = 1,
    .minor              = 0,
};

static int myst7789_probe(struct spi_device *spi)
{
    struct device *dev = &spi->dev;
    struct mipi_dbi_dev *dbidev;
    struct drm_device *drm;
    struct mipi_dbi *dbi;
    struct gpio_desc *dc;
    int ret;

    dbidev = devm_drm_dev_alloc(dev, &myst7789_driver,
                                struct mipi_dbi_dev, drm);
    if (IS_ERR(dbidev))
        return PTR_ERR(dbidev);

    dbi = &dbidev->dbi;
    drm = &dbidev->drm;

    /* D/C GPIO (data/command select) */
    dc = devm_gpiod_get(dev, "dc", GPIOD_OUT_LOW);
    if (IS_ERR(dc))
        return dev_err_probe(dev, PTR_ERR(dc), "no dc gpio\n");

    /* Reset GPIO */
    dbi->reset = devm_gpiod_get(dev, "reset", GPIOD_OUT_HIGH);
    if (IS_ERR(dbi->reset))
        return dev_err_probe(dev, PTR_ERR(dbi->reset), "no reset gpio\n");

    /* Backlight (optional) */
    dbidev->backlight = devm_of_find_backlight(dev);
    if (IS_ERR(dbidev->backlight))
        return PTR_ERR(dbidev->backlight);

    /* Initialize the SPI + DC transport */
    ret = mipi_dbi_spi_init(spi, dbi, dc);
    if (ret)
        return ret;

    /* Register the DRM device with our mode + pipe funcs */
    ret = mipi_dbi_dev_init(dbidev, &myst7789_pipe_funcs, &myst7789_mode, 0);
```

```
    if (ret)
        return ret;

    drm_mode_config_reset(drm);

    ret = drm_dev_register(drm, 0);
    if (ret)
        return ret;

    /* Enable fbdev emulation so /dev/fb0 appears */
    drm_fbdev_generic_setup(drm, 16);     /* 16 bpp */

    spi_set_drvdata(spi, drm);
    return 0;
}

static void myst7789_remove(struct spi_device *spi)
{
    struct drm_device *drm = spi_get_drvdata(spi);
    drm_dev_unplug(drm);
    drm_atomic_helper_shutdown(drm);
}

static const struct of_device_id myst7789_of_match[] = {
    { .compatible = "linuxlearn,myst7789" },
    { }
};
MODULE_DEVICE_TABLE(of, myst7789_of_match);

static const struct spi_device_id myst7789_spi_id[] = {
    { "myst7789", 0 },
    { }
};
MODULE_DEVICE_TABLE(spi, myst7789_spi_id);

static struct spi_driver myst7789_spi_driver = {
    .driver = {
        .name = "myst7789",
        .of_match_table = myst7789_of_match,
    },
    .id_table = myst7789_spi_id,
    .probe = myst7789_probe,
    .remove = myst7789_remove,
};
module_spi_driver(myst7789_spi_driver);

MODULE_LICENSE("GPL");
```

DT:

```
&ecspi3 {
    display@0 {
        compatible = "linuxlearn,myst7789";
        reg = <0>;
        spi-max-frequency = <40000000>;
        dc-gpios = <&gpio4 14 GPIO_ACTIVE_HIGH>;
        reset-gpios = <&gpio4 15 GPIO_ACTIVE_LOW>;
        backlight = <&backlight_lcd>;
```

```
        rotation = <0>;
    };
};
```

Build (needs the kernel's DRM headers, build against your kernel tree). Load:

```
[root@pa-mini:~]# insmod myst7789.ko
[root@pa-mini:~]# dmesg | grep -i myst7789
[drm] Initialized myst7789 1.0.0 for spi3.0 on minor 0
[root@pa-mini:~]# ls /dev/fb0
/dev/fb0
[root@pa-mini:~]# cat /sys/class/graphics/fb0/virtual_size
240,240

[root@pa-mini:~]# cat /dev/urandom > /dev/fb0      # noise on the LCD
[root@pa-mini:~]# # The mipi_dbi helper auto-flushes dirty regions to the panel
```

What the `mipi_dbi` helper did for us:

- Tracked dirty rectangles (only changed regions sent over SPI).
- Did the CASET/RASET/RAMWR command sequence per flush.
- Converted the XRGB8888 DRM framebuffer to RGB565 on the fly.
- Provided the DRM CRTC/connector/plane plumbing.
- Provided `/dev/fb0` fbdev emulation.

We provided two things: the chip-specific init sequence and the pixel dimensions. About 200 lines, mostly the init sequence.

For production: handle rotation through MADCTL. Add sleep on disable for power management. Better still, switch to the mainline `panel-mipi-dbi` generic driver and skip the custom code entirely.

83.5 The even-easier way: panel-mipi-dbi

Mainline has a *generic* DRM tiny driver, `drivers/gpu/drm/tiny/panel-mipi-dbi.c`, that reads the init sequence from a **firmware blob** rather than hardcoded C. You write the init sequence in a text file, compile it with a tool, and the generic driver runs it. No C code at all.

DT:

```
&ecspi3 {
    display@0 {
        compatible = "sitronix,st7789v", "panel-mipi-dbi-spi";
        reg = <0>;
        spi-max-frequency = <40000000>;
        dc-gpios = <&gpio4 14 GPIO_ACTIVE_HIGH>;
        reset-gpios = <&gpio4 15 GPIO_ACTIVE_LOW>;
```

```
        backlight = <&backlight_lcd>;
        width-mm = <28>;
        height-mm = <28>;

        panel-timing {
            hactive = <240>;
            vactive = <240>;
            hback-porch = <0>;
            vback-porch = <0>;
            clock-frequency = <0>;
            hfront-porch = <0>;
            hsync-len = <0>;
            vfront-porch = <0>;
            vsync-len = <0>;
        };
    };
};
```

The init sequence lives in `/lib/firmware/mydisplay.bin`, compiled from a human-readable description with the `mipi-dbi-cmd` tool (in the kernel tree). For a panel already supported by an existing init blob, you reuse it.

For most production: **use panel-mipi-dbi with a firmware init blob**. Write a custom DRM driver only if the panel needs runtime logic the firmware-blob can't express.

83.6 Performance, partial updates

A full 240×240 RGB565 frame = 115200 bytes. At 40 MHz SPI: ~23 ms = ~43 fps theoretical max for full refresh. In practice, ~30 fps with overhead.

The trick: **only send what changed**. The `mipi_dbi` helper tracks dirty rectangles. If your UI updates a small clock digit, only that rectangle is sent, a few KB, sub-millisecond. This is why SPI LCDs feel snappy for typical UIs (mostly static with small dynamic regions) even though full-screen video would be a slideshow.

For a Qt or LVGL app, the toolkit reports dirty regions to DRM. The helper sends only those. No tearing if you respect the vblank.

83.7 Key Notes

- An SPI display controller stores pixels and refreshes its glass independently. The host updates controller memory rather than generating every raster timing edge.
- Command/data selection, pixel format, window offsets and initialization are part of the module contract. A shared controller name does not establish identical panel setup.
- Partial updates reduce transferred bytes when only a region changes. The benefit depends on application damage tracking and driver behavior, not just nominal SPI speed.

- DRM helpers can own common transport and presentation mechanics while a panel description supplies device-specific requirements.

83.8 Pitfalls

- **Missing DC GPIO toggle.** Commands and data get confused. Controller does nothing or garbage. The `mipi_dbi` helper handles DC. A hand-rolled driver must toggle it correctly (low for command byte, high for parameters/pixels).
- **Wrong init sequence.** Each module vendor tweaks the gamma/voltage commands. A generic ST7789 init may give washed-out or dark colors. Use the vendor's recommended sequence.
- **INVON vs INVOFF.** Many cheap ST7789 modules have inverted pixels, black shows as white. Toggle MIPI_DCS_ENTER_INVERT_MODE.
- **BGR vs RGB.** MADCTL bit 3 swaps red and blue. If red text shows as blue, flip it.
- **Column/row offset.** Some 240×240 ST7789 modules are actually 240×320 panels with the display window offset, you must add an x/y offset in CASET/RASET or the image is shifted. Vendor-specific.
- **SPI too fast.** ST7789 tolerates ~62 MHz. ILI9341 writes at ~30 MHz reliably. Above the limit: corrupt pixels. Start at 20 MHz, ramp up.
- **No reset pulse.** Without a hardware reset at probe, the controller may be in an undefined state. Always pulse RESET low for ~10 µs at init.
- **Backlight not handled.** Display "works" (data flows) but screen dark. Wire and enable the backlight.
- **DRM headers version mismatch.** The DRM API churns between kernel versions. `mipi_dbi_dev_init` signature changed across 5.x/6.x. Match your driver to your kernel.

83.9 Going deeper

- **`drivers/gpu/drm/drm_mipi_dbi.c`**: the shared mipi_dbi helper. Read `mipi_dbi_fb_dirty` for the partial-update path.
- **`drivers/gpu/drm/tiny/ili9341.c`**: a complete tiny driver. Compare to the from-scratch version.
- **`drivers/gpu/drm/tiny/panel-mipi-dbi.c`**: the generic firmware-blob driver.
- **`Documentation/gpu/drm-kms-helpers.rst`**: the simple-display-pipe + mipi_dbi helpers.
- **`tools/` → `mipi-dbi-cmd`**: the firmware-blob compiler.
- **ST7789 datasheet (Sitronix)**: command list. Init recommendations.

- **ILI9341 datasheet (Ilitek)**: command list.
- **MIPI DBI specification**: the standard these controllers descend from.

> Next chapter: **Chapter 84: QSPI LCD.** When SPI bandwidth isn’t enough, quad-SPI displays for higher frame rates and the round LCDs popular in smartwatches.

Chapter 84: QSPI LCD

> **What:** displays driven over **quad-SPI** (4 data lanes instead of 1) for ~4× the bandwidth of plain SPI. Common on round smartwatch-style LCDs: **GC9D01** (round 160×160), **ST77916** (round 360×360), **SH8601** (AMOLED). We cover why QSPI matters for displays, the quad-mode MIPI-DBI command framing, the i.MX6ULL QSPI controller's display-mode constraints, and how the mainline `mipi-dbi` helper handles quad transfers.
>
> **Why:** a plain-SPI 360×360 16-bit display needs 259 KB per frame, at 40 MHz SPI that's 52 ms = ~19 fps full-refresh, too slow for smooth animation. Quad-SPI quadruples the data rate to about 13 ms per frame, or 75 fps. That is the difference between smooth animation and a slideshow. QSPI is newer and less common. Fewer mainline drivers exist, so you are more likely to write your own.
>
> **Focus: QSPI display = MIPI-DBI commands sent over 4 data lanes**. The command model is identical to Ch 83 (CASET/RASET/RAMWR), but each byte's 8 bits are spread across 4 IO lines (2 bits per line per clock), so pixels stream 4× faster. The challenge is that the i.MX6ULL QSPI controller is designed for *flash*, not displays, using it for a display means working within its command-LUT model.

84.1 Why QSPI for displays

Bandwidth math for a full-frame refresh at 16-bit color:

Display	Pixels	Bytes/frame	@40 MHz SPI (1-bit)	@40 MHz QSPI (4-bit)
240×240	57600	115 KB	23 ms (43 fps)	6 ms (170 fps)
360×360	129600	259 KB	52 ms (19 fps)	13 ms (77 fps)
454×454	206116	412 KB	82 ms (12 fps)	21 ms (48 fps)

For small displays plain SPI is fine. For 360×360+ round AMOLEDs (smartwatch class), QSPI is needed for smooth animation. Partial updates (Ch 83) help, but full-frame animation (a sweeping watch hand redrawing the whole face) needs the bandwidth.

84.2 Quad-SPI signaling

Plain SPI: 1 data line (MOSI), 1 bit per clock. Quad-SPI: 4 data lines (IO0–IO3), 4 bits per clock, 4× the throughput at the same clock rate.

```
Plain SPI:   MOSI — b7 b6 b5 b4 b3 b2 b1 b0   (8 clocks per byte)

Quad SPI:    IO3 — b7 b3      (2 clocks per byte)
             IO2 — b6 b2
             IO1 — b5 b1
             IO0 — b4 b0
```

For displays, a common framing (the “QSPI display” convention used by these controllers):

- **Command phase**: single-lane (IO0 only), with a leading “command marker” byte.
- **Address phase**: single-lane or quad.
- **Data phase**: quad-lane (pixels stream 4× fast).

The exact framing is controller-specific. A typical ST77916 write:

```
/CS↓
single-lane: 0x02 (write marker) + 0x00 + cmd + 0x00
quad-lane:   pixel data ...
/CS↑
```

84.3 The i.MX6ULL QSPI-for-display problem

The i.MX6ULL QSPI controller (`fsl-quadspi`) was designed for **NOR flash** (Ch 64), not displays. It works through a **LUT** (look-up table) of pre-programmed command sequences, you fill LUT entries describing “command + address + dummy + data” phases, then trigger them.

For a display, this is awkward:

- The flash-oriented driver assumes memory-mapped reads (XIP). Displays are write-only streams.
- The LUT model doesn’t naturally fit “send command on 1 lane, then stream N KB of pixels on 4 lanes.”
- Mainline `fsl-quadspi.c` (the MTD driver) doesn’t expose a display path.

> **MTD:** Memory Technology Device, Linux’s raw flash subsystem for eraseblock-based storage.

Two practical approaches on i.MX6ULL:

1. **Use the QSPI in `spi-mem` mode.** Recent kernels expose the QSPI controller via the `spi_mem` API, which `mipi_dbi` can drive for quad transfers. The mainline `spi-nxp-fspi.c` (for i.MX8) supports this. The older `fsl-quadspi.c` (i.MX6) has limited support. Check your kernel.
2. **Bit-bang or use a different SoC.** If the QSPI controller can't do display-style transfers, you're stuck with plain SPI on i.MX6ULL. QSPI displays are more at home on i.MX8M / RP2040 / ESP32-S3 which have flexible QSPI/PIO peripherals.

Honest assessment: the i.MX6ULL is *not* a great host for QSPI displays. Its QSPI is flash-centric. For a product needing a QSPI AMOLED, an i.MX8M Mini (with FlexSPI) or a dedicated display co-processor is a better fit. We still cover the topic. The displays are increasingly common, and you will likely meet them on a more capable SoC.

84.4 How mipi_dbi handles quad mode

Some SPI controllers expose `spi_mem` with quad support. On those SoCs, the `mipi_dbi` helper can issue quad transfers. The newer DRM tiny drivers (e.g., for the ST77903, SH8601 AMOLEDs) use `spi_mem_op` structures:

```
/* Conceptual — quad pixel write via spi_mem */
struct spi_mem_op op = SPI_MEM_OP(
    SPI_MEM_OP_CMD(0x32, 1),                  /* command on 1 lane */
    SPI_MEM_OP_ADDR(3, (0x002C00), 1),         /* RAMWR address, 1 lane */
    SPI_MEM_OP_NO_DUMMY,
    SPI_MEM_OP_DATA_OUT(len, pixels, 4));      /* pixels on 4 lanes! */

spi_mem_exec_op(spimem, &op);
```

The `SPI_MEM_OP_DATA_OUT(len, buf, 4)`, the trailing `4` says "4-lane (quad) data phase." The controller spreads the bytes across IO0–IO3. This is the same `spi_mem_op` abstraction we saw in the QSPI flash chapter (Ch 64.6), repurposed for display data.

A from-scratch QSPI display driver on a supporting SoC would:

1. Acquire the `spi_mem` device.
2. Send the init sequence via single-lane `spi_mem_op`s.
3. Implement the `mipi_dbi` `fb_dirty` callback to send CASET/RASET (single-lane) then RAMWR pixels (quad-lane) via `spi_mem_exec_op`.

The structure mirrors Ch 83's `myst7789.c`, but the pixel-write path uses a quad `spi_mem_op` instead of a plain `spi_sync`. We do not reproduce the full driver. It is the Ch 83 driver with the data phase changed to quad. It only works on an SoC whose controller supports quad `spi_mem` writes, stock i.MX6ULL does not.

84.5 When QSPI display makes sense

Scenario	Bus choice
240×240 static UI, occasional updates	plain SPI (Ch 83)
360×360 round watch face, full-screen animation	QSPI (on a capable SoC)
Large UI, video	parallel RGB (Ch 82) or MIPI-DSI (not on i.MX6ULL)
i.MX6ULL specifically	plain SPI or parallel RGB, QSPI display support is weak

For the i.MX6ULL reader: **prefer parallel RGB (Ch 82) for big/fast displays, plain SPI (Ch 83) for small ones.** QSPI displays are a “know it exists, use on a better SoC” topic.

84.6 Key Notes

- Quad data lanes can reduce pixel-transfer time, but command phases, overhead and refresh synchronization still affect usable frame rate.
- The i.MX6ULL QSPI controller is flash-oriented. A display design needs demonstrated compatibility between its transaction model, Linux support and the panel protocol.
- Command lane width and data lane width are separate fields of a transaction. Lane order, chip select and initialization must match the module.
- Transport speed does not remove tearing or power constraints. A plain SPI or parallel RGB design may be more appropriate when support for quad display transfers is uncertain.

84.7 Pitfalls

- **Assuming i.MX6ULL QSPI does displays well.** It doesn’t, it’s flash-centric. Verify `spi_mem` quad-out support before committing to a QSPI display on this SoC.
- **Quad-mode lane order.** The bit-to-lane mapping (which bits go on IO0 vs IO3) varies. A wrong mapping gives scrambled pixels. Match the controller’s convention.

- **Command vs data lane count.** Commands usually go on 1 lane, pixels on 4. Sending the command on 4 lanes confuses the controller. The `spi_mem_op` cmd/data buswidth fields must be set per phase.
- **Pull-ups on IO2/IO3.** In single-lane mode, IO2/IO3 may double as /WP and /HOLD. For display use they're data. Ensure no conflicting pulls.
- **Frame tearing at high fps.** Without vsync/TE-pin synchronization, fast full-frame updates tear. Many QSPI AMOLEDs have a TE (tearing-effect) output. Wire it to a GPIO IRQ and sync updates.

> **MCU bridge:** Think of an IRQ like an EXTI/NVIC interrupt path, except Linux splits the hard interrupt from deferred work and must share lines across drivers. **MCU bridge:** Think of Linux GPIO like the same pin set/reset block you used on STM32, but accessed through a kernel subsystem that owns numbering, direction, interrupts, and user-space exposure. **IRQ:** interrupt request, the signal path that tells the CPU or interrupt controller that hardware needs service. **GPIO:** General-Purpose Input/Output, a pin controlled as a digital input, output, or interrupt source.

- **Power: AMOLED inrush.** QSPI AMOLEDs can draw significant current when displaying bright white. Budget the rail accordingly.
- **Init sequence length.** AMOLED init sequences are long (50–100 commands) with vendor-specific gamma/voltage. Get the exact sequence from the module vendor.

84.8 Going deeper

- **`drivers/gpu/drm/tiny/`**: newer entries (st7571, sh8601-style) show quad approaches on capable SoCs.
- **`drivers/spi/spi-mem.c`**: the `spi_mem_op` abstraction shared between flash and these displays.
- **`drivers/spi/spi-nxp-fspi.c`**: i.MX8 FlexSPI driver with quad support (contrast with i.MX6's `fsl-quadspi.c`).
- **`Documentation/spi/spi-summary.rst`**: SPI/QSPI overview.
- **ST77916 / SH8601 datasheets**: QSPI display command framing.
- **i.MX6ULL Reference Manual, QuadSPI chapter**: the LUT model and its constraints.

> **Note on Group H so far:** parallel RGB (Ch 82) for big/fast on i.MX6ULL, plain SPI (Ch 83) for small/cheap, QSPI (Ch 84) for high-bandwidth-on-capable-SoCs. The next two chapters cover the *non-raster* display technologies: OLED character/graphic displays + e-paper (Ch 85), and the touch input that turns a display into an interface (Ch 86).

> Next chapter: **Chapter 85: OLED & e-paper (SSD1306 / SH1106 / SSD1680).** Tiny monochrome OLEDs and the radically different e-paper refresh model.

Chapter 85: OLED & e-paper

What: two non-backlit display technologies. **OLED**, self-emissive monochrome dot-matrix: **Solomon SSD1306** (128×64, I²C/SPI), **Sino Wealth SH1106** (132×64, the "almost-SSD1306"). **E-paper**, bistable reflective: **Solomon SSD1680** (the controller behind most 1.5"–2.9" e-paper modules). For each: the framebuffer-RAM model, the refresh mechanics (instant for OLED, multi-second for e-paper), the mainline driver, and a from-scratch SSD1306 fbdev driver.

Why: OLEDs are the cheapest "real display" you can buy. A 128×64 OLED costs around $2, draws ~20 mA, and has perfect contrast without a backlight. E-paper is the opposite. Zero idle power, the image persists with no power. Sunlight-readable. But slow to update. Both show up constantly in IoT status displays, instruments, smart-home panels, electronic shelf labels. They need very different driver thinking than the raster panels of Ch 82–84.

Focus: OLED uses a page-addressed bitmap. E-paper uses a two-buffer waveform replay driven by a LUT. They need very different driver code. The SSD1306 stores a 1-bit-per-pixel image in internal RAM organized as 8 "pages" of 128 bytes. You push the whole bitmap and it displays instantly. The SSD1680 stores *two* images (old + new) and replays a per-pixel voltage *waveform* (the LUT) over ~2 seconds to flip the e-ink particles, a completely different mental model.

85.1 Technology & chip comparison

	SSD1306 OLED	SH1106 OLED	SSD1680 e-paper
Tech	OLED (self-emissive)	OLED	E-ink (bistable reflective)
Resolution	128×64 / 128×32	132×64 (128 visible)	up to 296×176
Colors	mono (on/off)	mono	mono or B/W/Red
Interface	I²C (0x3C/0x3D) or SPI	I²C / SPI	SPI
RAM model	8 pages × 128 cols	8 pages × 132 cols (offset!)	dual buffer + LUT

	SSD1306 OLED	SH1106 OLED	SSD1680 e-paper
Refresh	instant (<1 ms)	instant	full ~2 s, partial ~0.3 s
Idle power	~5 µW off, ~20 mW on	similar	**0 W** (image persists)
Sunlight	poor (washes out)	poor	excellent
Lifetime	OLED burn-in over years	burn-in	~1M refresh cycles
Volume price	$1.50–3	$1.50–3	$5–15
Mainline driver	`ssd130x` (DRM tiny) / `ssd1307fb` (fbdev)	same `ssd130x`	`repaper.c`, `ssd1680` patches

Pick guide:

- **SSD1306**: default tiny status display. Ubiquitous.
- **SH1106**: nearly identical. The catch is its 132-column RAM with a 2-pixel offset, drivers must account for it.
- **SSD1680**: e-paper for “set it and forget it” displays (shelf labels, room signs, low-power dashboards).

85.2 SSD1306, page-addressed framebuffer

The SSD1306 has 128×64 = 8192 pixels = 1024 bytes of internal RAM (1 bit/pixel). It’s organized as **8 pages**, each 128 bytes wide. Within a page, *each byte is a vertical column of 8 pixels* (bit 0 = top, bit 7 = bottom):

```
Page 0: byte[0] = column 0, rows 0-7    (bit0=row0, bit7=row7)
        byte[1] = column 1, rows 0-7
        ...
        byte[127] = column 127, rows 0-7
Page 1: rows 8-15
...
Page 7: rows 56-63
```

This vertical-byte layout is unusual (most framebuffers are row-major with horizontal bytes). To set pixel (x, y): `page = y / 8; bit = y % 8; ram[page*128 + x] |= (1 << bit);`.

Commands vs data

Like the SPI LCDs (Ch 83), the SSD1306 distinguishes commands from data. Over **I²C**, the distinction is a control byte:

```
I²C write: [control byte] [data/command byte] ...
control byte 0x00 → following bytes are COMMANDS
control byte 0x40 → following bytes are DATA (pixels)
```

Over **SPI**, a D/C GPIO selects (like Ch 83).

> **MCU bridge:** Think of Linux GPIO like the same pin set/reset block you used on STM32, but accessed through a kernel subsystem that owns numbering, direction, interrupts, and user-space exposure. **GPIO:** General-Purpose Input/Output, a pin controlled as a digital input, output, or interrupt source.

Key commands

Command	Hex	Purpose
Display off / on	0xAE / 0xAF	Power the panel
Set contrast	0x81 + value	Brightness
Set memory mode	0x20 + mode	Horizontal/vertical/page addressing
Set column range	0x21 + start + end	(horizontal mode)
Set page range	0x22 + start + end	(horizontal mode)
Charge pump	0x8D + 0x14	Enable internal boost (mandatory!)
Set COM pins	0xDA + config	Panel geometry

The init sequence

A working SSD1306 init (the canonical sequence):

```
0xAE              display off
0x20 0x00         horizontal addressing mode
0xB0              page start
0xC8              COM scan direction (flip vertical)
0x00 0x10         column low/high
0x40              start line 0
0x81 0x7F         contrast
0xA1              segment remap (flip horizontal)
0xA6              normal (non-inverted)
0xA8 0x3F         multiplex 64
0xA4              output follows RAM
0xD3 0x00         display offset 0
0xD5 0xF0         clock divide
0xD9 0x22         pre-charge
0xDA 0x12         COM pins config
0xDB 0x20         VCOMH
0x8D 0x14         charge pump ON  ← without this, screen stays black
0xAF              display on
```

The `0x8D 0x14` (charge pump enable) is the #1 gotcha. The OLED needs an internal boost converter for the ~7 V it requires. Forget this command and the screen stays black.

85.3 How the mainline driver works

Two mainline drivers exist:

- **`drivers/video/fbdev/ssd1307fb.c`**: legacy fbdev driver. Exposes `/dev/fb0`. A 1-bit framebuffer.
- **`drivers/gpu/drm/solomon/ssd130x.c`** + `ssd130x-i2c.c` / `ssd130x-spi.c`, modern DRM driver. Covers SSD1305/1306/1307/1309 and SH1106.

The DRM `ssd130x` driver:

1. Reads geometry from DT (width, height, page-offset, COM config).
2. Runs the init sequence.
3. Registers a DRM device with a 1-bit (or emulated) framebuffer.
4. On framebuffer flush, converts the DRM XRGB8888 buffer to the SSD1306's packed-vertical-byte format and pushes the dirty pages.

```
/* ssd130x.c — simplified flush */
static int ssd130x_update_rect(struct ssd130x_device *ssd130x,
                               struct drm_rect *rect, u8 *buf)
{
    unsigned int x = rect->x1, y = rect->y1;
    unsigned int width = drm_rect_width(rect);
    unsigned int pages = DIV_ROUND_UP(drm_rect_height(rect), 8);

    /* Set the column + page address window */
    ssd130x_write_cmd(ssd130x, 3, 0x21, x + ssd130x->col_offset,
```

```
                      x + ssd130x->col_offset + width - 1);
    ssd130x_write_cmd(ssd130x, 3, 0x22, y / 8, y / 8 + pages - 1);

    /* Pack the rect into vertical bytes and push as data */
    for (each page in the rect) {
        ... pack 8 vertical pixels per byte ...
        ssd130x_write_data(ssd130x, packed, width);
    }
    return 0;
}
```

The DRM helper does the XRGB→1bit conversion (any non-black pixel → on). The driver packs into vertical bytes and pushes only the dirty pages.

85.4 Writing an SSD1306 fbdev driver from scratch

A DRM driver is the modern way, but for a monochrome OLED a simple **fbdev** driver is more illustrative, it's the bare "here's a framebuffer, here's how I push it to the chip" model. ~200 lines.

`myssd1306.c`:

```
#include <linux/init.h>
#include <linux/module.h>
#include <linux/i2c.h>
#include <linux/fb.h>
#include <linux/delay.h>
#include <linux/vmalloc.h>

#define WIDTH   128
#define HEIGHT  64
#define PAGES   (HEIGHT / 8)
#define FB_SIZE (WIDTH * PAGES)        /* 1024 bytes */

struct myssd {
    struct i2c_client *client;
    struct fb_info *info;
    u8 *vmem;                          /* the 1-bit framebuffer (1024 bytes) */
};

static int ms_cmd(struct myssd *m, u8 cmd)
{
    u8 buf[2] = { 0x00, cmd };          /* control 0x00 = command */
    return i2c_master_send(m->client, buf, 2);
}

static int ms_data(struct myssd *m, const u8 *data, int len)
{
    /* Prefix with control byte 0x40 = data */
    u8 *buf = kmalloc(len + 1, GFP_KERNEL);
    int ret;
    if (!buf) return -ENOMEM;
    buf[0] = 0x40;
    memcpy(buf + 1, data, len);
    ret = i2c_master_send(m->client, buf, len + 1);
```

```
    kfree(buf);
    return ret;
}

static const u8 ssd1306_init[] = {
    0xAE, 0x20, 0x00, 0xB0, 0xC8, 0x00, 0x10, 0x40,
    0x81, 0x7F, 0xA1, 0xA6, 0xA8, 0x3F, 0xA4, 0xD3,
    0x00, 0xD5, 0xF0, 0xD9, 0x22, 0xDA, 0x12, 0xDB,
    0x20, 0x8D, 0x14, 0xAF,
};

static int ms_init_panel(struct myssd *m)
{
    int i, err;
    for (i = 0; i < ARRAY_SIZE(ssd1306_init); i++) {
        err = ms_cmd(m, ssd1306_init[i]);
        if (err < 0) return err;
    }
    return 0;
}

/* Push the entire vmem framebuffer to the OLED */
static void ms_flush(struct myssd *m)
{
    /* Set full window: columns 0-127, pages 0-7 */
    ms_cmd(m, 0x21); ms_cmd(m, 0); ms_cmd(m, 127);
    ms_cmd(m, 0x22); ms_cmd(m, 0); ms_cmd(m, 7);
    ms_data(m, m->vmem, FB_SIZE);
}

/* fbdev deferred-io callback: called after the framebuffer is touched */
static void ms_deferred_io(struct fb_info *info, struct list_head *pagelist)
{
    struct myssd *m = info->par;
    ms_flush(m);
}

static struct fb_deferred_io ms_defio = {
    .delay = HZ / 30,                   /* flush at most 30×/sec */
    .deferred_io = ms_deferred_io,
};

static const struct fb_fix_screeninfo ms_fix = {
    .id = "myssd1306",
    .type = FB_TYPE_PACKED_PIXELS,
    .visual = FB_VISUAL_MONO10,
    .xpanstep = 0, .ypanstep = 0, .ywrapstep = 0,
    .line_length = WIDTH / 8,
    .accel = FB_ACCEL_NONE,
};

static const struct fb_var_screeninfo ms_var = {
    .xres = WIDTH, .yres = HEIGHT,
    .xres_virtual = WIDTH, .yres_virtual = HEIGHT,
    .bits_per_pixel = 1,
};

/* Standard fbdev ops backed by deferred-io */
static const struct fb_ops ms_fbops = {
```

```
    .owner       = THIS_MODULE,
    .fb_read     = fb_sys_read,
    .fb_write    = fb_sys_write,
    .fb_fillrect = sys_fillrect,
    .fb_copyarea = sys_copyarea,
    .fb_imageblit = sys_imageblit,
    .fb_mmap     = fb_deferred_io_mmap,
};

static int ms_probe(struct i2c_client *client)
{
    struct myssd *m;
    struct fb_info *info;
    int err;

    m = devm_kzalloc(&client->dev, sizeof(*m), GFP_KERNEL);
    if (!m) return -ENOMEM;
    m->client = client;

    m->vmem = vzalloc(FB_SIZE);
    if (!m->vmem) return -ENOMEM;

    err = ms_init_panel(m);
    if (err) goto free_vmem;

    info = framebuffer_alloc(0, &client->dev);
    if (!info) { err = -ENOMEM; goto free_vmem; }

    info->par = m;
    info->fbops = &ms_fbops;
    info->fix = ms_fix;
    info->var = ms_var;
    info->screen_buffer = m->vmem;
    info->screen_size = FB_SIZE;
    info->fbdefio = &ms_defio;
    fb_deferred_io_init(info);

    m->info = info;
    i2c_set_clientdata(client, m);

    err = register_framebuffer(info);
    if (err) goto release_fb;

    /* Clear screen */
    memset(m->vmem, 0, FB_SIZE);
    ms_flush(m);

    dev_info(&client->dev, "SSD1306 OLED at /dev/fb%d\n", info->node);
    return 0;

release_fb:
    fb_deferred_io_cleanup(info);
    framebuffer_release(info);
free_vmem:
    vfree(m->vmem);
    return err;
}

static void ms_remove(struct i2c_client *client)
```

```
{
    struct myssd *m = i2c_get_clientdata(client);
    unregister_framebuffer(m->info);
    fb_deferred_io_cleanup(m->info);
    framebuffer_release(m->info);
    vfree(m->vmem);
}

static const struct of_device_id ms_of_match[] = {
    { .compatible = "linuxlearn,myssd1306" },
    { }
};
MODULE_DEVICE_TABLE(of, ms_of_match);

static const struct i2c_device_id ms_id[] = { { "myssd1306", 0 }, { } };
MODULE_DEVICE_TABLE(i2c, ms_id);

static struct i2c_driver ms_driver = {
    .driver = {
        .name = "myssd1306",
        .of_match_table = ms_of_match,
    },
    .probe = ms_probe,
    .remove = ms_remove,
    .id_table = ms_id,
};
module_i2c_driver(ms_driver);

MODULE_LICENSE("GPL");
```

DT:

```
&i2c1 {
    oled@3c {
        compatible = "linuxlearn,myssd1306";
        reg = <0x3c>;
    };
};
```

Test:

```
[root@pa-mini:~]# insmod myssd1306.ko
[root@pa-mini:~]# dmesg | tail -1
myssd1306 1-003c: SSD1306 OLED at /dev/fb0

[root@pa-mini:~]# cat /sys/class/graphics/fb0/virtual_size
128,64

# The framebuffer is 1-bit-per-pixel, 16 bytes per line.
# Write a pattern (every byte 0xFF = alternating columns lit):
[root@pa-mini:~]# python3 -c "import sys; sys.stdout.buffer.write(bytes([0xFF,
0x00]*512))" > /dev/fb0
[root@pa-mini:~]# # → vertical stripes appear on the OLED
```

The key abstraction: **`fb_deferred_io`**. The framebuffer lives in vmalloc'd RAM. User-space writes to it via mmap or write(). The deferred-io machinery batches writes and calls `ms_deferred_io` (which calls `ms_flush`) at most 30×/second, so a flurry of pixel writes results in one I²C burst, not thousands.

What we got, ~200 lines:

- `/dev/fb0` at 128×64, 1-bit.
- Deferred-io batched flushing.
- Standard fbdev ops (fillrect, imageblit) work, so toolkits like `fbi`, `fbcon`, simple Cairo apps render.

What we skipped vs the mainline `ssd130x` DRM driver:

- DRM integration (mainline is DRM. Fbdev is legacy).
- SPI transport (we did I²C only).
- SH1106 column-offset handling.
- Contrast / inversion runtime control.
- Partial-page updates (we flush the whole frame).

85.5 SH1106, the off-by-2 sibling

SH1106 is nearly pin- and command-compatible with SSD1306, *except* its RAM is **132 columns wide** while the visible panel is 128. The visible area is centered, so columns 2–129 are shown. Columns 0–1 and 130–131 are off-screen.

The consequence: when you set the column window, you must offset by 2. A driver written for SSD1306 (offset 0) shows the SH1106 image shifted 2 pixels right, with garbage wrapping on the left edge. The mainline `ssd130x` driver reads a `col_offset` value from DT to handle this case.

For SH1106, change the column window from 0–127 to 2–129. Some SH1106 variants do not support horizontal addressing mode at all, for those, set the page and column manually for each page.

85.6 SSD1680 e-paper, a completely different model

E-paper is *bistable*: each pixel is a microcapsule of black and white particles that move under an electric field and *stay put* with no power. This gives zero-power image retention but makes refreshing slow and weird.

The dual-buffer + LUT model

The SSD1680 holds **two image buffers**: the “old” image (currently on the glass) and the “new” image. To refresh, the controller replays a **waveform**, a sequence of voltage pulses per pixel, that moves the particles from old to new state. The waveform is defined by a **LUT** (look-up table) loaded into the controller.

```
1. Write new image to the "new" buffer (SPI, like SSD1306 RAM).
2. (Optional) write old image to "old" buffer for differential update.
3. Load the waveform LUT (or use the OTP built-in LUT).
4. Trigger DISPLAY UPDATE (command 0x22 + 0x20).
5. Wait for BUSY pin to go inactive (~2 seconds for full refresh).
```

Two refresh modes:

- **Full refresh** (~2 s): flashes the whole screen black→white→target several times to clear ghosting. Clean image, slow, visible flicker.
- **Partial refresh** (~0.3 s): updates only changed pixels without the full flash. Fast, but accumulates **ghosting** (faint remnants of old images). Must do a full refresh periodically to clear.

Why this is hard for a generic framebuffer

A normal framebuffer is “write pixel, see it.” E-paper is “write image, trigger update, wait 2 seconds, see it.” The deferred-io model breaks (you can’t flush 30×/sec, each flush takes 2 s). E-paper drivers expose a custom update trigger and let user-space decide *when* to refresh.

Mainline: `drivers/gpu/drm/tiny/repaper.c` (for Pervasive Displays panels) and various SSD1680 patches. The DRM driver does a full refresh on each DRM page-flip, acceptable for “update once per minute” status displays, terrible for anything interactive.

Practical use

```
# After bring-up, a typical e-paper update from user-space:
[root@pa-mini:~]# cat dashboard.bin > /dev/fb0     # write the image
[root@pa-mini:~]# echo 1 > /sys/class/graphics/fb0/refresh   # trigger (driver-specific)
# ... 2 seconds of flicker ...
# Image now on screen, persists with power off.
```

For e-paper, you design the UI around the refresh model: update once per minute (weather, room booking), use partial refresh for a clock-style digit, full refresh periodically to de-ghost.

85.7 Key Notes

- OLED and e-paper have different refresh architectures. OLED updates controller memory for an active display, while e-paper applies a panel-specific waveform to change a retained image.
- Framebuffer layout is not necessarily row-major pixel storage. Page-addressed monochrome controllers require an explicit packing and addressing model.
- An e-paper update has a busy interval and depends on waveform and temperature assumptions. Partial refresh trades speed and appearance against accumulating artifacts.
- Power policy differs too: OLED brightness and static content affect consumption and aging, while e-paper retention does not imply that updates are free.

85.8 Pitfalls

- **Charge pump not enabled.** `0x8D 0x14` is mandatory for the internal boost. Without it, the OLED is black. The #1 SSD1306 gotcha.
- **Wrong I²C address.** SSD1306 is 0x3C (or 0x3D if the addr-select pad is bridged). Check with `i2cdetect`.
- **SH1106 treated as SSD1306.** 2-pixel horizontal shift + edge garbage. Set the column offset.
- **Vertical-byte confusion.** The SSD1306 RAM layout is column-of-8-vertical-pixels per byte. A driver that assumes row-major bytes shows a scrambled image. Pack correctly.
- **Flushing too often.** Without deferred-io, every pixel write triggers a full 1 KB I²C transfer, the bus saturates. Batch with `fb_deferred_io`.
- **OLED burn-in.** A static image (a logo, a fixed UI) burns in over months. Invert/shift periodically, or dim, for always-on displays.
- **E-paper partial-refresh ghosting.** Accumulates. Schedule periodic full refreshes.
- **E-paper update while busy.** Triggering a new update before BUSY clears corrupts the image. Always wait for BUSY.
- **E-paper temperature sensitivity.** Below ~0 °C, e-paper refreshes very slowly or not at all. The waveform LUT is temperature-dependent. Good modules include a temperature sensor and ship multiple LUTs for the controller to switch between.

85.9 Going deeper

- **`drivers/gpu/drm/solomon/ssd130x.c`**: modern DRM OLED driver (SSD1306/SH1106).

- `drivers/video/fbdev/ssd1307fb.c`: legacy fbdev OLED driver. Compare to the from-scratch version.
- `drivers/gpu/drm/tiny/repaper.c`: e-paper DRM driver (Pervasive Displays).
- `Documentation/fb/deferred_io.rst`: the deferred-io framework.
- **SSD1306 datasheet (Solomon Systech)**: command table. Page-addressing model.
- **SH1106 datasheet**: note the 132-column RAM.
- **SSD1680 datasheet**: the dual-buffer + LUT waveform model.
- `drivers/video/fbdev/core/fb_defio.c`: deferred-io implementation.

> Next chapter: **Chapter 86: Touch input ICs.** Turning a display into an interface: capacitive buttons (TTP223), capacitive matrices (MPR121), and resistive 4-wire touch (XPT2046) with its calibration story.

Chapter 86: Touch input ICs

What: three touch technologies at increasing complexity. **TTP223** (single capacitive button, GPIO output, `gpio-keys`), **MPR121** (12-channel capacitive, I²C, with IRQ), **XPT2046/ADS7846** (4-wire resistive touchscreen controller, SPI, ADC-based, needs calibration). For each: physics, protocol, the input subsystem integration, and a from-scratch XPT2046 input driver, the most interesting, since resistive touch requires reading X/Y ADC channels and software calibration. **MCU bridge:** Think of an IRQ like an EXTI/NVIC interrupt path, except Linux splits the hard interrupt from deferred work and must share lines across drivers. **MCU bridge:** Think of Linux GPIO like the same pin set/reset block you used on STM32, but accessed through a kernel subsystem that owns numbering, direction, interrupts, and user-space exposure. **IRQ:** interrupt request, the signal path that tells the CPU or interrupt controller that hardware needs service. **GPIO:** General-Purpose Input/Output, a pin controlled as a digital input, output, or interrupt source.

Why: A display without touch is a monitor. Add touch and it becomes an interface. Capacitive buttons replace mechanical ones (no wear, sealed enclosures). Capacitive matrices give you piano keys, sliders, proximity. Resistive touch is the cheap way to make any LCD interactive (works with gloves and stylus, unlike capacitive). Each is a different input-subsystem pattern, this chapter completes the input picture started in Ch 45 and the multi-touch GT911 of Ch 55G.

Focus: Capacitive touch is threshold detection, a digital touched/not-touched. Resistive touch is two ADC readings plus a calibration step. A capacitive button outputs a clean digital ‘touched’ signal. Wire it to `gpio-keys` and you are done. Resistive touch gives you two ADC readings, X and Y, that do not map directly to screen pixels. A calibration step (using `tslib` or `xinput_calibrator`) converts raw ADC counts into pixel coordinates.

Tooling. This chapter uses `evtest`, `libinput-tools`, `xinput_calibrator` (resistive), `i2c-tools`.

- **Ubuntu-base (target):** `apt install evtest libinput-tools xinput-calibrator i2c-tools`
- **Buildroot:** `BR2_PACKAGE_EVTEST=y BR2_PACKAGE_LIBINPUT=y BR2_PACKAGE_TSLIB=y BR2_PACKAGE_I2C_TOOLS=y` **Buildroot:** a configuration-driven build system that produces a complete root filesystem and related images.
- Full per-tool reference: Userspace tooling appendix. **MCU bridge:** Think of the rootfs as the firmware image’s file-backed runtime environment. On an MCU you link everything into flash. On Linux, programs and config live in this mounted tree. **rootfs:** root filesystem, the directory tree mounted at / that contains /bin, /etc, /dev, and libraries.

86.1 Technology comparison

	TTP223	MPR121	XPT2046 (ADS7846 clone)
Tech	single cap button	12-channel cap	4-wire resistive
Output	GPIO (digital touch/no-touch)	I²C (per-channel + IRQ)	SPI (X/Y/Z ADC values)
Touch type	finger only	finger only	finger, gloved, stylus, anything
Multi-touch	no	per-channel (12 buttons)	no (single point)
Position	none (button)	none (discrete buttons)	continuous X/Y
Needs calibration	no	threshold tuning	yes (per-panel)
Sealed enclosure	yes (touch through glass)	yes	no (touch surface exposed)
Volume price	$0.10–0.30	$1–2	$1–2
Mainline driver	`gpio-keys`	`mpr121_touchkey.c`	`ads7846.c`

Pick guide:

- **TTP223**: a single touch button, power, mode, wake. Cheapest possible touch input.
- **MPR121**: capacitive keypad, slider, or proximity, 12 electrodes.
- **XPT2046**: make any resistive-overlay LCD touch-interactive. Works with gloves/stylus. Needs calibration.

For *capacitive multi-touch* (a modern phone-style glass touchscreen), see Ch 55G (GT911).

86.2 TTP223, capacitive button via gpio-keys

The TTP223 is a self-contained capacitive touch sensor: a copper pad (on your PCB or a separate electrode) connects to its input. Its output pin goes high (or toggles, configurable) when touched. From Linux's view it's just a GPIO that changes state.

No driver needed, use the in-tree **`gpio-keys`** (Ch 45):

```
gpio_keys {
    compatible = "gpio-keys";

    touch_power {
        label = "PowerTouch";
        linux,code = <KEY_POWER>;
        gpios = <&gpio4 14 GPIO_ACTIVE_HIGH>;
        debounce-interval = <50>;
    };

    touch_menu {
        label = "MenuTouch";
        linux,code = <KEY_MENU>;
        gpios = <&gpio4 15 GPIO_ACTIVE_HIGH>;
    };
};
```

Each TTP223's output → a GPIO → a `gpio-keys` button. Touching the pad generates a `KEY_POWER` or `KEY_MENU` input event, which `evtest /dev/input/eventN` will show. No driver code needed.

The TTP223 has configuration pads (TOG, AHLB) you bridge to set: momentary vs toggle output, active-high vs active-low, fast vs low-power mode. These are hardware straps, not software-configurable. Set them on the PCB.

86.3 MPR121, 12-channel capacitive

The MPR121 measures capacitance on 12 electrodes. Each electrode's capacitance rises when a finger approaches (the finger adds capacitance to ground). The chip tracks a per-electrode baseline and reports "touched" when capacitance exceeds a threshold.

Register map highlights:

Reg	Name	Purpose
0x00–0x01	Touch Status	12 bits, one per electrode (1 = touched)

Reg	Name	Purpose
0x04–0x1D	Electrode filtered data	per-electrode capacitance reading
0x1E–0x2A	Baseline values	per-electrode baseline
0x41–0x5A	Touch/Release thresholds	per-electrode
0x5B–0x7F	Configuration	filter, debounce, auto-config
0x80	Soft reset	write 0x63

The IRQ pin asserts when the touch status changes. Bring-up:

1. Soft reset (write 0x63 to 0x80).
2. Configure per-electrode touch/release thresholds (typical: touch 12, release 6, hysteresis).
3. Configure filtering + auto-configuration.
4. Write Electrode Configuration (0x5E) to enable N electrodes + start.
5. On IRQ, read touch status (0x00–0x01). 12 bits tell you which electrodes are touched.

Mainline driver + input

`drivers/input/keyboard/mpr121_touchkey.c` registers each electrode as a key. DT:

```
&i2c1 {
    mpr121@5a {
        compatible = "freescale,mpr121-touchkey";
        reg = <0x5a>;
        interrupt-parent = <&gpio4>;
        interrupts = <14 IRQ_TYPE_EDGE_FALLING>;
        autorepeat;
        linux,keycodes = <KEY_0 KEY_1 KEY_2 KEY_3 KEY_4 KEY_5
                          KEY_6 KEY_7 KEY_8 KEY_9 KEY_A KEY_B>;
        vdd-supply = <&reg_3v3>;
    };
};
```

12 electrodes → 12 key codes. Touching electrode 0 emits `KEY_0`, etc. The driver reads the touch-status register on each IRQ and reports key-down/up events. `evtest` shows them.

For a slider or proximity (analog), you'd read the filtered-data registers directly (the mainline keytouch driver only does discrete keys. A custom IIO or input driver could expose the analog capacitance).

> **IIO:** Industrial I/O, Linux's subsystem for sensors, ADCs, DACs, and buffered sampled data.

86.4 XPT2046, 4-wire resistive touch

Resistive touch is two transparent resistive layers separated by spacer dots. Pressing pushes the layers together at the touch point. To find the position:

```
Measure X: drive a voltage gradient across the X layer (left=0V, right=3.3V);
           read the voltage at the touch point on the Y layer (an ADC reading
           proportional to the X position).
Measure Y: drive a voltage gradient across the Y layer (top/bottom);
           read on the X layer.
Measure Z (pressure): drive one axis, measure resistance — tells you if/how
           hard it's pressed.
```

The XPT2046 is an SPI-controlled 12-bit ADC + analog mux that automates this. You send a control byte selecting which measurement (X, Y, Z1, Z2). It drives the right layers and returns the ADC value.

Protocol

Each measurement is a 3-byte SPI transaction:

```
byte 0: control byte:
    bit 7:   start (1)
    bits 6:4: channel select (A2 A1 A0)
              101 = X position
              001 = Y position
              011 = Z1, 100 = Z2 (pressure)
    bit 3:   mode (0 = 12-bit, 1 = 8-bit)
    bit 2:   SER/DFR (0 = differential — better noise rejection)
    bits 1:0: power-down mode
bytes 1-2: read the 12-bit result (in the upper 12 of 16 bits)
```

Reading X position:

```
   tx = { 0xD0, 0x00, 0x00 };    /* 0xD0 = start + X channel + 12-bit + differential */
   rx = spi_transfer(tx, 3);
   x = ((rx[1] << 8) | rx[2]) >> 3;   /* 12-bit result, right-justified */
```

A touch IRQ (the PENIRQ pin) asserts when the panel is pressed, wire it to a GPIO IRQ to avoid polling.

86.5 Writing an XPT2046 input driver from scratch

Goal: an `input_dev` reporting `ABS_X` / `ABS_Y` / `ABS_PRESSURE` + `BTN_TOUCH`, driven by the PENIRQ. ~250 lines.

`myxpt2046.c`:

```
#include <linux/init.h>
#include <linux/module.h>
#include <linux/spi/spi.h>
#include <linux/input.h>
#include <linux/interrupt.h>
#include <linux/gpio/consumer.h>
#include <linux/delay.h>

#define CMD_X   0xD0    /* start + X + 12-bit + differential */
#define CMD_Y   0x90    /* start + Y */
#define CMD_Z1  0xB0
#define CMD_Z2  0xC0

#define MAX_ADC 4095

struct myxpt {
    struct spi_device *spi;
    struct input_dev *input;
    int irq;
    struct gpio_desc *pen_gpio;     /* PENIRQ line, also readable */
};

static int xp_read_channel(struct myxpt *x, u8 cmd)
{
    u8 tx[3] = { cmd, 0x00, 0x00 };
    u8 rx[3];
    struct spi_transfer xfer = { .tx_buf = tx, .rx_buf = rx, .len = 3 };
    int err = spi_sync_transfer(x->spi, &xfer, 1);
    if (err) return err;
    return ((rx[1] << 8) | rx[2]) >> 3;     /* 12-bit, right-justified */
}

/* Read X/Y a few times and median-filter for noise rejection */
static int xp_read_filtered(struct myxpt *x, u8 cmd)
{
    int samples[5], i, j, tmp;
    for (i = 0; i < 5; i++)
        samples[i] = xp_read_channel(x, cmd);
    /* tiny bubble sort */
    for (i = 0; i < 4; i++)
        for (j = 0; j < 4 - i; j++)
```

```
            if (samples[j] > samples[j+1]) {
                tmp = samples[j]; samples[j] = samples[j+1]; samples[j+1] = tmp;
            }
    return samples[2];      /* median */
}

static irqreturn_t xp_irq_thread(int irq, void *dev_id)
{
    struct myxpt *x = dev_id;

    /* While the pen is down (PENIRQ low), sample continuously */
    while (gpiod_get_value(x->pen_gpio) == 0) {   /* active-low PENIRQ */
        int rx = xp_read_filtered(x, CMD_X);
        int ry = xp_read_filtered(x, CMD_Y);
        int z1 = xp_read_channel(x, CMD_Z1);

        /* Crude pressure check: ignore ghost touches with z1 too low */
        if (z1 > 100) {
            input_report_abs(x->input, ABS_X, rx);
            input_report_abs(x->input, ABS_Y, ry);
            input_report_abs(x->input, ABS_PRESSURE, z1);
            input_report_key(x->input, BTN_TOUCH, 1);
            input_sync(x->input);
        }
        msleep(10);     /* ~100 Hz sampling while touched */
    }

    /* Pen up */
    input_report_key(x->input, BTN_TOUCH, 0);
    input_report_abs(x->input, ABS_PRESSURE, 0);
    input_sync(x->input);

    return IRQ_HANDLED;
}

static int xp_probe(struct spi_device *spi)
{
    struct myxpt *x;
    int err;

    x = devm_kzalloc(&spi->dev, sizeof(*x), GFP_KERNEL);
    if (!x) return -ENOMEM;
    x->spi = spi;

    spi->mode = SPI_MODE_0;
    spi->bits_per_word = 8;
    err = spi_setup(spi);
    if (err) return err;

    x->pen_gpio = devm_gpiod_get(&spi->dev, "pendown", GPIOD_IN);
    if (IS_ERR(x->pen_gpio))
        return dev_err_probe(&spi->dev, PTR_ERR(x->pen_gpio), "no pendown gpio\n");

    x->input = devm_input_allocate_device(&spi->dev);
    if (!x->input) return -ENOMEM;

    x->input->name = "myxpt2046";
    x->input->phys = "myxpt2046/input0";
    x->input->id.bustype = BUS_SPI;
```

```
    /* Report ABS_X / ABS_Y in raw 12-bit ADC range; calibration done in user-space */
    input_set_abs_params(x->input, ABS_X, 0, MAX_ADC, 0, 0);
    input_set_abs_params(x->input, ABS_Y, 0, MAX_ADC, 0, 0);
    input_set_abs_params(x->input, ABS_PRESSURE, 0, MAX_ADC, 0, 0);
    input_set_capability(x->input, EV_KEY, BTN_TOUCH);

    err = input_register_device(x->input);
    if (err) return err;

    x->irq = gpiod_to_irq(x->pen_gpio);
    err = devm_request_threaded_irq(&spi->dev, x->irq, NULL, xp_irq_thread,
                                    IRQF_TRIGGER_FALLING | IRQF_ONESHOT,
                                    "myxpt2046", x);
    if (err) return err;

    spi_set_drvdata(spi, x);
    return 0;
}

static const struct of_device_id xp_of_match[] = {
    { .compatible = "linuxlearn,myxpt2046" },
    { }
};
MODULE_DEVICE_TABLE(of, xp_of_match);

static const struct spi_device_id xp_id[] = { { "myxpt2046", 0 }, { } };
MODULE_DEVICE_TABLE(spi, xp_id);

static struct spi_driver xp_driver = {
    .driver = {
        .name = "myxpt2046",
        .of_match_table = xp_of_match,
    },
    .probe = xp_probe,
    .id_table = xp_id,
};
module_spi_driver(xp_driver);

MODULE_LICENSE("GPL");
```

DT:

```
&ecspi3 {
    touch@0 {
        compatible = "linuxlearn,myxpt2046";
        reg = <0>;
        spi-max-frequency = <2000000>;
        pendown-gpios = <&gpio4 14 GPIO_ACTIVE_LOW>;
        interrupt-parent = <&gpio4>;
        interrupts = <14 IRQ_TYPE_EDGE_FALLING>;
    };
};
```

Test:

```
[root@pa-mini:~]# insmod myxpt2046.ko
[root@pa-mini:~]# evtest /dev/input/event2
Supported events:
  EV_KEY: BTN_TOUCH
  EV_ABS: ABS_X (0-4095), ABS_Y (0-4095), ABS_PRESSURE

# Touch the panel:
Event: type 3 (EV_ABS), code 0 (ABS_X), value 1834
Event: type 3 (EV_ABS), code 1 (ABS_Y), value 2201
Event: type 1 (EV_KEY), code 330 (BTN_TOUCH), value 1
Event: ---------- SYN_REPORT ----------
```

It reports **raw ADC coordinates** (0–4095), not screen pixels. The mapping from ADC to pixels is the calibration step.

What we got, ~250 lines:

- PENIRQ-driven sampling (no polling when idle).
- Median-filtered X/Y for noise rejection.
- Pressure-gated to reject ghost touches.
- Standard `input_dev` with ABS axes.

What we skipped vs mainline `ads7846.c`:

- Hardware debounce settling-time tuning.
- The `ti,x-plate-ohms` pressure-to-resistance conversion.
- Runtime configuration of sample count, settle delay.
- Proper PENIRQ vs sample interleaving (the mainline driver disables PENIRQ during sampling to avoid spurious IRQs).

86.6 Calibration, raw ADC to screen pixels

The XPT2046 gives raw ADC values. They don’t map 1:1 to pixels:

- The touch panel’s edges don’t align with the LCD’s edges (mechanical offset).
- The ADC range isn’t the full panel (the resistive gradient has dead zones).
- X/Y may be swapped or inverted relative to the display.

Calibration computes a **3×2 affine transform** (`tslib`’s model):

```
   pixel_x = a·adc_x + b·adc_y + c
   pixel_y = d·adc_x + e·adc_y + f
```

You collect 5 calibration points (corners + center), solve for (a..f) by least-squares. The standard tools:

- **`tslib`** (`ts_calibrate`): writes a calibration file `/etc/pointercal`. Apps use `tslib` to transform raw events.
- **`xinput_calibrator`** (X11): generates an Xorg config snippet.
- The kernel's **`touchscreen` properties** in DT (`touchscreen-size-x`, `touchscreen-inverted-x`, etc.) handle simple cases (swap/invert/scale) but not the full affine.

For our driver to integrate with `tslib`:

```
[root@pa-mini:~]# export TSLIB_TSDEVICE=/dev/input/event2
[root@pa-mini:~]# ts_calibrate            # touch the 5 crosshairs
[root@pa-mini:~]# ts_test                 # verify the transform
```

After calibration, `/etc/pointercal` holds the transform. `tslib`-linked apps (or the `evdev`+`libinput` path with a calibration matrix) report pixel coordinates.

For a cleaner kernel-side approach, the mainline `ads7846` driver + the `touchscreen` DT properties + `libinput`'s calibration matrix handle it without `tslib`.

86.7 Key Notes

- A capacitive button, multi-electrode touch controller and resistive panel expose different signals. The input subsystem can unify events without erasing their physical differences.
- Resistive measurements need a transform from raw ADC coordinates into the display frame. Axis orientation, calibration and pressure validity belong in that contract.
- Thresholds, filtering and release behavior determine whether noise becomes an input event. Application gestures should remain separate from low-level measurement handling.
- Electrical straps and acquisition behavior are hardware-specific. Reusing an input driver does not establish identical polarity, key mapping or interrupt timing.

86.8 Pitfalls

- **TTP223 strap config.** Momentary vs toggle, active-high vs low, set by PCB straps (TOG, AHLB pads), not software. Get them right at layout.
- **MPR121 thresholds too sensitive.** Default thresholds may trigger on proximity, not touch. Tune touch/release thresholds with hysteresis (touch > release).

- **XPT2046 PENIRQ during sampling.** Sampling toggles the panel layers, and that can trigger PENIRQ even when no one is touching. The mainline driver masks PENIRQ during sampling. Our simple driver polls the GPIO instead, it works, but the mainline approach is cleaner.
- **Resistive touch needs calibration, always.** Raw ADC values are not pixel coordinates. Ship `ts_calibrate` or a kernel-side calibration matrix. Uncalibrated touch is unusable.
- **X/Y swapped or inverted.** Depends on panel mounting. Fix via calibration or DT `touchscreen-swapped-x-y` / `touchscreen-inverted-x`.
- **Noisy resistive readings.** Median-filter (we do 5 samples). Single-sample touch jitters badly.
- **Ghost touches at light pressure.** Gate on pressure (Z), reject touches below a Z threshold.
- **SPI clock too fast for XPT2046.** Max ~2 MHz for reliable conversion. Faster gives noisy ADC values.
- **MPR121 needs the IRQ.** Polling the touch-status register works but is laggy. Wire the IRQ.

86.9 Going deeper

- **`drivers/input/touchscreen/ads7846.c`**: the production XPT2046/ADS7846 driver. Compare to the from-scratch version. Note the PENIRQ masking.
- **`drivers/input/keyboard/mpr121_touchkey.c`**: MPR121 driver.
- **`drivers/input/keyboard/gpio_keys.c`**: for TTP223-style buttons.
- **`Documentation/devicetree/bindings/input/touchscreen/touchscreen.yaml`**: the common touchscreen properties (swap/invert/scale).
- **`tslib`** at https://github.com/libts/tslib, calibration + filtering library.
- **XPT2046 / ADS7846 datasheets**: control-byte format, differential vs single-ended.
- **MPR121 datasheet (NXP/Freescale)**: register map, auto-config.
- **`libinput` calibration matrix docs**: the modern alternative to tslib.

> **End of Group H, Displays (Ch 82–86).** The full display spectrum: parallel RGB (big/fast), SPI (small/smart), QSPI (high-bandwidth), OLED (tiny/crisp), e-paper (zero-power/slow), plus the touch input (capacitive button/matrix, resistive) that makes them interactive.

> Next chapter: **Chapter 87: Parallel CSI cameras (OV5640 / OV7725 / GC2145).** Group I (Cameras), the i.MX6ULL's parallel camera interface, the V4L2 sensor sub-device model, and bringing up a real camera sensor.

Chapter 87: Parallel CSI cameras

What: image sensors connected to the i.MX6ULL's **parallel CSI** (Camera Sensor Interface, 8-bit data + pixel/line/frame sync, *not* MIPI). Three sensors compared: **OmniVision OV5640** (5 MP, the common default), **OV7725** (0.3 MP VGA, simple), **GalaxyCore GC2145** (2 MP, budget). For each: the dual-bus model (I²C for control + parallel for pixels), the V4L2 **sub-device** architecture, the media-controller graph, and a from-scratch V4L2 sensor subdev driver.

Why: Any i.MX6ULL product with a camera uses parallel CSI. The i.MX6ULL has no MIPI-CSI. Smart doorbells, barcode scanners, and machine-vision sensors all run through this interface. The driver model has more moving parts than most subsystems. A *sensor* sub-device feeds a *CSI bridge* sub-device, which feeds a *video* device. The media controller wires them all together. Understand this graph and the rest of V4L2 falls into place.

Focus: A camera sensor is two devices in one. It has an I²C control interface and a parallel pixel stream. V4L2 models the combination as a sub-device. The sensor's driver lives on I²C (configure resolution, exposure, format via registers) but exposes a *pad* that emits a pixel-format on the parallel bus. The CSI bridge captures that stream into DRAM. The media graph connects sensor-pad → CSI-pad → video-node.

Tooling. This chapter uses `v4l-utils`, `gstreamer1.0-tools` + plugins, `i2c-tools`.

- **Ubuntu-base (target):** `apt install v4l-utils gstreamer1.0-tools gstreamer1.0-plugins-base gstreamer1.0-plugins-good gstreamer1.0-plugins-bad i2c-tools`
- **Buildroot:** `BR2_PACKAGE_V4L_UTILS=y BR2_PACKAGE_GSTREAMER1=y BR2_PACKAGE_GST1_PLUGINS_BASE=y BR2_PACKAGE_I2C_TOOLS=y` **Buildroot:** a configuration-driven build system that produces a complete root filesystem and related images.
- Full per-tool reference: Userspace tooling appendix. **MCU bridge:** Think of the rootfs as the firmware image's file-backed runtime environment. On an MCU you link everything into flash. On Linux, programs and config live in this mounted tree. **rootfs:** root filesystem, the directory tree mounted at / that contains /bin, /etc, /dev, and libraries.

87.1 Sensor comparison

	OmniVision OV5640	OmniVision OV7725	GalaxyCore GC2145
Resolution	2592×1944 (5 MP)	640×480 (VGA)	1600×1200 (2 MP)
Output formats	YUV422, RGB565, RAW, JPEG	YUV422, RGB565, RAW	YUV422, RGB565, RAW
Interface	8/10-bit parallel + MIPI	8-bit parallel	8-bit parallel + MIPI
Control	I²C (0x3C)	I²C (0x21)	I²C (0x3C)
Frame rate	30 fps @ 1080p, 15 fps @ 5 MP	60 fps @ VGA	30 fps @ UXGA
Autofocus	VCM (voice-coil motor) option	no (fixed focus)	no
Built-in ISP	yes (AWB, AE, gamma)	basic	basic
Init complexity	~250 register writes	~100	~200
Volume price	$5–10 (module)	$2–4	$3–5
Mainline driver	`ov5640.c`	`ov772x.c`	`gc2145.c` (recent)

i.MX6ULL CSI bandwidth: the parallel CSI captures 8-bit data at the sensor’s pixel clock (typically up to ~96 MHz). For 5 MP at full res, the data rate exceeds what the i.MX6ULL can comfortably DMA to DRAM, practical use is QVGA/VGA streaming or occasional full-res stills.

> **MCU bridge:** Think of DMA like the MCU DMA controller you used for UART or SPI, but with cache coherency, scatter-gather descriptors, and kernel ownership rules added. **DMA:** Direct Memory Access. Hardware moves data to or from memory without the CPU copying each byte.

Pick guide:

- **OV5640**: when you need quality + the built-in ISP (auto-exposure, auto-white-balance). The default.
- **OV7725**: VGA is enough, want simplicity and high frame rate.
- **GC2145**: cheap 2 MP. The budget option.

87.2 The dual-bus camera model

A parallel camera sensor has *two* connections to the SoC:

```
i.MX6ULL                        OV5640
────────                        ──────
I²C (SCL/SDA) ◄──────────────► SIOC/SIOD   (control: registers)
CSI_PCLK      ◄──────────────── PCLK        (pixel clock from sensor)
CSI_HSYNC     ◄──────────────── HREF        (line valid)
CSI_VSYNC     ◄──────────────── VSYNC       (frame valid)
CSI_D[7:0]    ◄──────────────── D[9:2]       (8-bit pixel data)
CSI_MCLK      ────────────────► XCLK         (master clock TO sensor, ~24 MHz)
GPIO          ────────────────► PWDN         (power down)
GPIO          ────────────────► RESET
```

Two roles:

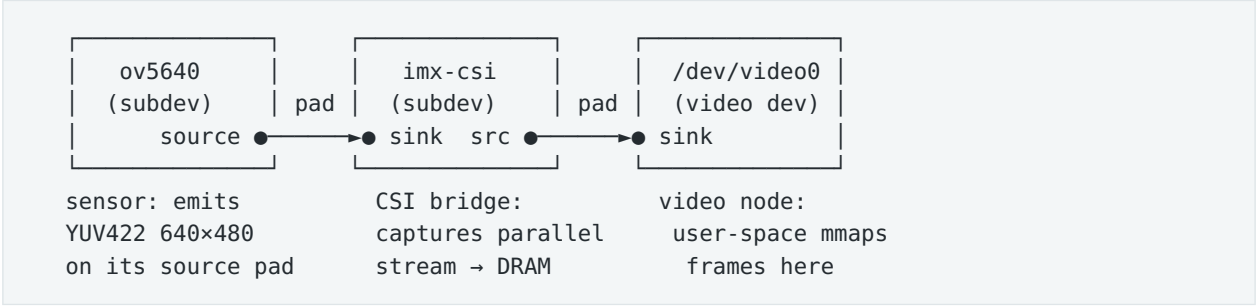
- **I²C**: the host configures the sensor, resolution, output format, exposure, gain, test patterns. Hundreds of register writes.
- **Parallel bus**: the sensor *continuously* streams pixels (like the RGB LCD of Ch 82, but inbound). The sensor is the *master* of the pixel clock (it generates PCLK). The CSI captures on each PCLK edge when HREF+VSYNC indicate valid data.
- **MCLK**: the SoC provides the sensor's master clock (typically 24 MHz). The sensor's internal PLL multiplies it up to the pixel clock.

> **MCU bridge:** Think of a PLL like the clock multiplier setup you used on STM32, but with more clock roots, gates, and consumers that Linux later needs to describe.
> **PLL:** Phase-Locked Loop, a clock block that multiplies a reference clock to create faster clocks.

The sensor needs its MCLK running *before* I²C communication works (the sensor's logic is clocked by MCLK). This is a common bring-up gotcha.

87.3 The V4L2 sub-device + media-controller model

V4L2 models a capture pipeline as a graph of **entities** connected by **pads** and **links**:



- **Sub-device** (`v4l2_subdev`): a sensor or a processing block. The OV5640 is a subdev. The CSI is a subdev. Subdevs have **pads** (source = output, sink = input).
- **Video device** (`/dev/video0`): the user-space-facing node where frames are dequeued.
- **Media controller** (`/dev/media0`): the graph topology. `media-ctl` inspects and configures links.

The user-space app:

1. Opens `/dev/media0`, inspects the graph.
2. Sets the format on each pad (sensor source, CSI sink, CSI source), they must match through the pipeline.
3. Opens `/dev/video0`, sets the buffer format, queues buffers, streams.

All this indirection, separate subdevs, explicit format propagation, feels heavy for one camera. It is what lets V4L2 handle complex pipelines (multiple sensors, ISP stages, scalers) with the same code paths.

87.4 How the mainline ov5640 driver works

Source: `drivers/media/i2c/ov5640.c` (~3700 lines, sensors are big because of the register init tables and the ISP control).

The driver is a **`v4l2_subdev`** registered on I²C. Its structure:

```
struct ov5640_dev {
    struct i2c_client *i2c_client;
    struct v4l2_subdev sd;            /* the subdev */
    struct media_pad pad;             /* its source pad */
    struct v4l2_ctrl_handler ctrls;   /* exposure, gain, AWB, etc. */
    struct clk *xclk;                  /* master clock input */
    struct gpio_desc *reset_gpio;
    struct gpio_desc *pwdn_gpio;
    struct regulator_bulk_data supplies[OV5640_NUM_SUPPLIES];
    /* current mode */
    const struct ov5640_mode_info *current_mode;
};
```

Probe

```
static int ov5640_probe(struct i2c_client *client)
{
    struct ov5640_dev *sensor;

    sensor = devm_kzalloc(&client->dev, sizeof(*sensor), GFP_KERNEL);
    sensor->i2c_client = client;

    /* Get clocks, regulators, GPIOs */
    sensor->xclk = devm_clk_get(&client->dev, "xclk");
    sensor->reset_gpio = devm_gpiod_get(&client->dev, "reset", GPIOD_OUT_HIGH);
    sensor->pwdn_gpio = devm_gpiod_get(&client->dev, "powerdown", GPIOD_OUT_HIGH);
    devm_regulator_bulk_get(&client->dev, ..., sensor->supplies);

    /* Power on + verify chip id */
    ov5640_power_on(sensor);                  /* MCLK on, GPIOs sequenced */
    ov5640_read_reg(sensor, OV5640_REG_CHIP_ID_HIGH, &id);   /* expect 0x5640 */

    /* Init V4L2 subdev */
    v4l2_i2c_subdev_init(&sensor->sd, client, &ov5640_subdev_ops);
    sensor->sd.flags |= V4L2_SUBDEV_FL_HAS_DEVNODE;
    sensor->pad.flags = MEDIA_PAD_FL_SOURCE;
    media_entity_pads_init(&sensor->sd.entity, 1, &sensor->pad);

    /* Register controls (exposure, gain, AWB, test pattern) */
    ov5640_init_controls(sensor);

    /* Register the subdev asynchronously (the CSI bridge will bind to it) */
    v4l2_async_register_subdev_sensor(&sensor->sd);
    return 0;
}
```

The subdev ops

```
static const struct v4l2_subdev_video_ops ov5640_video_ops = {
    .s_stream = ov5640_s_stream,       /* start/stop streaming */
};

static const struct v4l2_subdev_pad_ops ov5640_pad_ops = {
    .enum_mbus_code = ov5640_enum_mbus_code,    /* what formats can I emit? */
    .get_fmt        = ov5640_get_fmt,
    .set_fmt        = ov5640_set_fmt,            /* set resolution + format */
    .enum_frame_size = ov5640_enum_frame_size,
};
```

`s_stream(1)` writes the registers to start the sensor outputting pixels. `set_fmt` programs the resolution + format registers. The sensor's "pad format" tells the CSI bridge what to expect.

The register init tables

The bulk of `ov5640.c` is **mode tables**, arrays of (register, value) pairs for each supported resolution:

```
static const struct reg_value ov5640_init_setting[] = {
    {0x3103, 0x11, 0, 0}, {0x3008, 0x82, 0, 5}, {0x3008, 0x42, 0, 0},
    {0x3103, 0x03, 0, 0}, {0x3630, 0x36, 0, 0}, {0x3631, 0x0e, 0, 0},
    /* ... ~250 more ... */
};

static const struct reg_value ov5640_setting_VGA_640_480[] = {
    {0x3c07, 0x08, 0, 0}, {0x3814, 0x31, 0, 0}, {0x3815, 0x31, 0, 0},
    /* resolution-specific registers */
};
```

These come from OmniVision’s reference code. They configure the sensor’s internal ISP, PLL, timing, and pixel pipeline. Like the BME280 compensation formulas (Ch 67) or the VL53L0X tuning blob (Ch 72), these are vendor IP transcribed verbatim, you don’t derive them, you apply them.

87.5 Writing a minimal V4L2 sensor subdev from scratch

A full OV5640 driver is 3700 lines (mostly mode tables). For learning, we’ll write a *minimal* subdev for a simple sensor, a fixed-format VGA YUV422 sensor (modeling an OV7725-class part), that registers in the media graph and streams one fixed mode. ~250 lines.

`mysensor.c`:

```
#include <linux/init.h>
#include <linux/module.h>
#include <linux/i2c.h>
#include <linux/clk.h>
#include <linux/gpio/consumer.h>
#include <linux/delay.h>
#include <media/v4l2-subdev.h>
#include <media/v4l2-async.h>
#include <media/v4l2-ctrls.h>
#include <media/v4l2-fwnode.h>

#define SENSOR_WIDTH   640
#define SENSOR_HEIGHT  480
#define REG_CHIP_ID    0x0A
#define CHIP_ID_VAL    0x77

struct mysensor {
    struct i2c_client *client;
    struct v4l2_subdev sd;
    struct media_pad pad;
    struct clk *xclk;
    struct gpio_desc *reset_gpio;
    struct v4l2_ctrl_handler ctrls;
    bool streaming;
};

#define to_mysensor(s) container_of(s, struct mysensor, sd)
```

```
static int ms_read(struct mysensor *m, u8 reg)
{
    return i2c_smbus_read_byte_data(m->client, reg);
}

static int ms_write(struct mysensor *m, u8 reg, u8 val)
{
    return i2c_smbus_write_byte_data(m->client, reg, val);
}

/* A (tiny, illustrative) init sequence — real sensors have hundreds */
static const u8 ms_init_seq[][2] = {
    { 0x12, 0x80 },   /* soft reset */
    { 0x11, 0x01 },   /* clock prescaler */
    { 0x0C, 0x00 },   /* output format = YUV422 */
    { 0x12, 0x00 },   /* VGA, YUV */
    /* ... real sensors: hundreds more ... */
};

static int ms_apply_init(struct mysensor *m)
{
    int i, err;
    err = ms_write(m, 0x12, 0x80);   /* reset */
    if (err) return err;
    msleep(10);
    for (i = 0; i < ARRAY_SIZE(ms_init_seq); i++) {
        err = ms_write(m, ms_init_seq[i][0], ms_init_seq[i][1]);
        if (err) return err;
    }
    return 0;
}

/* --- subdev pad ops: tell the pipeline what format we emit --- */

static int ms_enum_mbus_code(struct v4l2_subdev *sd,
                             struct v4l2_subdev_state *state,
                             struct v4l2_subdev_mbus_code_enum *code)
{
    if (code->index != 0) return -EINVAL;
    code->code = MEDIA_BUS_FMT_YUYV8_2X8;    /* YUV422, 8-bit parallel */
    return 0;
}

static int ms_get_fmt(struct v4l2_subdev *sd,
                      struct v4l2_subdev_state *state,
                      struct v4l2_subdev_format *fmt)
{
    fmt->format.width  = SENSOR_WIDTH;
    fmt->format.height = SENSOR_HEIGHT;
    fmt->format.code   = MEDIA_BUS_FMT_YUYV8_2X8;
    fmt->format.field  = V4L2_FIELD_NONE;
    fmt->format.colorspace = V4L2_COLORSPACE_SRGB;
    return 0;
}

/* We only support one fixed format, so set_fmt = get_fmt */
#define ms_set_fmt ms_get_fmt
```

```
static int ms_enum_frame_size(struct v4l2_subdev *sd,
                              struct v4l2_subdev_state *state,
                              struct v4l2_subdev_frame_size_enum *fse)
{
    if (fse->index != 0) return -EINVAL;
    fse->min_width = fse->max_width = SENSOR_WIDTH;
    fse->min_height = fse->max_height = SENSOR_HEIGHT;
    return 0;
}

/* --- subdev video ops: start/stop streaming --- */

static int ms_s_stream(struct v4l2_subdev *sd, int enable)
{
    struct mysensor *m = to_mysensor(sd);
    int err = 0;

    if (enable == m->streaming) return 0;

    if (enable) {
        /* Write registers to start pixel output */
        err = ms_write(m, 0x12, 0x00);   /* enable output */
    } else {
        err = ms_write(m, 0x12, 0x40);   /* standby */
    }
    if (!err) m->streaming = enable;
    return err;
}

static const struct v4l2_subdev_video_ops ms_video_ops = {
    .s_stream = ms_s_stream,
};

static const struct v4l2_subdev_pad_ops ms_pad_ops = {
    .enum_mbus_code  = ms_enum_mbus_code,
    .get_fmt         = ms_get_fmt,
    .set_fmt         = ms_set_fmt,
    .enum_frame_size = ms_enum_frame_size,
};

static const struct v4l2_subdev_ops ms_subdev_ops = {
    .video = &ms_video_ops,
    .pad   = &ms_pad_ops,
};

/* --- power sequencing --- */

static int ms_power_on(struct mysensor *m)
{
    int err;
    err = clk_prepare_enable(m->xclk);    /* MCLK must run before I²C works */
    if (err) return err;
    gpiod_set_value_cansleep(m->reset_gpio, 0);   /* release reset */
    msleep(20);
    return 0;
}

static int ms_probe(struct i2c_client *client)
{
```

```
    struct mysensor *m;
    int id, err;

    m = devm_kzalloc(&client->dev, sizeof(*m), GFP_KERNEL);
    if (!m) return -ENOMEM;
    m->client = client;

    m->xclk = devm_clk_get(&client->dev, "xclk");
    if (IS_ERR(m->xclk))
        return dev_err_probe(&client->dev, PTR_ERR(m->xclk), "no xclk\n");

    m->reset_gpio = devm_gpiod_get_optional(&client->dev, "reset", GPIOD_OUT_HIGH);

    err = ms_power_on(m);
    if (err) return err;

    /* Verify chip ID (now that MCLK is running) */
    id = ms_read(m, REG_CHIP_ID);
    if (id < 0) return dev_err_probe(&client->dev, id, "chip-id read failed\n");
    if (id != CHIP_ID_VAL)
        return dev_err_probe(&client->dev, -ENODEV,
                             "unexpected chip-id 0x%02x\n", id);

    err = ms_apply_init(m);
    if (err) return err;

    /* Init the V4L2 subdev */
    v4l2_i2c_subdev_init(&m->sd, client, &ms_subdev_ops);
    m->sd.flags |= V4L2_SUBDEV_FL_HAS_DEVNODE;
    m->sd.entity.function = MEDIA_ENT_F_CAM_SENSOR;
    m->pad.flags = MEDIA_PAD_FL_SOURCE;
    err = media_entity_pads_init(&m->sd.entity, 1, &m->pad);
    if (err) return err;

    /* Register asynchronously — the CSI bridge binds to us */
    err = v4l2_async_register_subdev_sensor(&m->sd);
    if (err) {
        media_entity_cleanup(&m->sd.entity);
        return err;
    }

    dev_info(&client->dev, "mysensor VGA YUV422 ready\n");
    return 0;
}

static void ms_remove(struct i2c_client *client)
{
    struct v4l2_subdev *sd = i2c_get_clientdata(client);
    struct mysensor *m = to_mysensor(sd);
    v4l2_async_unregister_subdev(sd);
    media_entity_cleanup(&sd->entity);
    clk_disable_unprepare(m->xclk);
}

static const struct of_device_id ms_of_match[] = {
    { .compatible = "linuxlearn,mysensor" },
    { }
};
MODULE_DEVICE_TABLE(of, ms_of_match);
```

```
static struct i2c_driver ms_driver = {
    .driver = {
        .name = "mysensor",
        .of_match_table = ms_of_match,
    },
    .probe = ms_probe,
    .remove = ms_remove,
};
module_i2c_driver(ms_driver);

MODULE_LICENSE("GPL");
```

DT, the sensor *and* its link to the CSI bridge via of_graph:

```
&i2c1 {
    sensor@21 {
        compatible = "linuxlearn,mysensor";
        reg = <0x21>;
        clocks = <&clks IMX6UL_CLK_CSI>;
        clock-names = "xclk";
        reset-gpios = <&gpio4 25 GPIO_ACTIVE_LOW>;
        pinctrl-names = "default";
        pinctrl-0 = <&pinctrl_csi_mclk>;

        port {
            sensor_out: endpoint {
                remote-endpoint = <&csi_in>;
                bus-width = <8>;
                hsync-active = <1>;
                vsync-active = <1>;
                pclk-sample = <1>;
            };
        };
    };
};

&csi {
    status = "okay";
    port {
        csi_in: endpoint {
            remote-endpoint = <&sensor_out>;
        };
    };
};
```

The `port`/`endpoint` graph links `sensor_out` → `csi_in`. The endpoint properties (`bus-width`, `hsync-active`, `pclk-sample`) describe the parallel bus electrical timing, the CSI bridge needs them to capture correctly.

Build, load:

```
[root@pa-mini:~]# insmod mysensor.ko
[root@pa-mini:~]# dmesg | grep mysensor
mysensor 1-0021: mysensor VGA YUV422 ready
```

```
[root@pa-mini:~]# media-ctl -p -d /dev/media0
Entity 1: mysensor 1-0021 (1 pad, 1 link)
            type V4L2 subdev subtype Sensor
        pad0: Source [fmt:YUYV8_2X8/640x480]
                -> "imx-csi":0 [ENABLED]
Entity 2: imx-csi (2 pads, ...)
...

[root@pa-mini:~]# ls /dev/video0
/dev/video0
```

The sensor appears in the media graph, linked to the CSI bridge, with `/dev/video0` available for capture. Capture with the V4L2 ioctl sequence from Ch 54B, or:

```
[root@pa-mini:~]# v4l2-ctl --device /dev/video0 \
    --set-fmt-video=width=640,height=480,pixelformat=YUYV \
    --stream-mmap=3 --stream-count=10 --stream-to=frames.raw
```

10 VGA YUV422 frames captured to `frames.raw`.

What we got, ~250 lines:

- A V4L2 sensor subdev registered in the media graph.
- Power sequencing (MCLK before I²C).
- One fixed VGA YUV422 mode.
- Streaming start/stop.

What a real OV5640 driver adds (the other ~3500 lines):

- Dozens of resolution modes with full register tables.
- Exposure/gain/AWB/AF V4L2 controls.
- Frame-rate control.
- Test patterns.
- The full ISP configuration.

87.6 The CSI bridge side

The i.MX6ULL CSI driver (`drivers/staging/media/imx/imx7-media-csi.c` or the imx6 variant) is the *other* subdev, it captures the parallel stream into DRAM via DMA. You don’t write this (it’s the SoC’s silicon driver). It:

1. Binds to the sensor subdev (via the of_graph link + v4l2-async).
2. Reads the sensor’s pad format.

3. Configures the CSI peripheral's capture format + DMA.
4. Provides `/dev/video0` (the video device).

The bridge + sensor must agree on format: if the sensor emits YUYV8_2X8 640×480, the CSI captures that. `media-ctl --set-v4l2` can override pad formats during bring-up debugging.

87.7 GStreamer + processing

Once `/dev/video0` works (Ch 54B covers this), GStreamer pipelines do the rest:

```
# Display on the LCD (Ch 82)
gst-launch-1.0 v4l2src device=/dev/video0 ! video/x-raw,format=YUY2,width=640,height=480
\
    ! videoconvert ! fbdevsink

# JPEG-encode and save
gst-launch-1.0 v4l2src device=/dev/video0 num-buffers=1 ! jpegenc ! filesink
location=snap.jpg

# Stream over network
gst-launch-1.0 v4l2src device=/dev/video0 ! jpegenc ! rtpjpegpay ! udpsink
host=192.168.1.100 port=5000
```

i.MX6ULL has no GPU/VPU, so JPEG/H.264 encoding is software, slow. Practical: ~5–10 fps at VGA for encoded streams. Raw capture is faster.

87.8 Key Notes

- A parallel camera has a control path over I2C and a pixel path into CSI. A sensor probe proves only the first of those boundaries.
- V4L2 subdevices and media endpoints describe a pipeline whose pad formats, bus widths and synchronization must agree.
- The sensor owns exposure and output timing, the capture controller owns transfer into buffers, and user space owns later processing or presentation.
- Resolution, frame rate, pixel format and processing all consume bandwidth or CPU. The i.MX6ULL's parallel CSI path is not a MIPI receiver or a hardware video-encoding guarantee.

87.9 Pitfalls

- **MCLK not running before I²C.** The sensor's logic is clocked by MCLK. No MCLK = no I²C ACK. Enable the xclk in power-on *before* the chip-id read.

- **Reset/powerdown GPIO polarity.** Wrong polarity holds the sensor in reset. I²C-detect fails.

> **MCU bridge:** Think of Linux GPIO like the same pin set/reset block you used on STM32, but accessed through a kernel subsystem that owns numbering, direction, interrupts, and user-space exposure. **GPIO:** General-Purpose Input/Output, a pin controlled as a digital input, output, or interrupt source.

- **Pad format mismatch through the pipeline.** Sensor emits YUYV but CSI configured for RGB → no frames or garbage. Use `media-ctl` to verify each pad's format matches.
- **Parallel bus timing (hsync/vsync/pclk polarity).** Wrong polarity in the endpoint properties → torn/shifted/blank frames. Match the sensor datasheet.
- **Bandwidth exceeds CSI capability.** 5 MP at 30 fps is ~220 MB/s, i.MX6ULL can't sustain it. Use lower res or lower frame rate.
- **No GPU/VPU.** Software JPEG/H.264 on i.MX6ULL is slow. Budget for raw capture or low-fps encoded streams.
- **Sensor lens not focused.** Fixed-focus modules have a screw-adjustable lens. Ship-from-factory focus may be off. Adjust for your working distance.
- **async subdev never binds.** If the of_graph link is wrong, the CSI bridge never finds the sensor. `/dev/video0` may not appear or has no source. Check `dmesg` for v4l2-async timeout warnings.

87.10 Going deeper

- **`drivers/media/i2c/ov5640.c`**: the production OV5640 driver. The mode tables are the bulk. The subdev ops match the from-scratch version's shape.
- **`drivers/staging/media/imx/`**: the i.MX CSI/IPU capture drivers.
- **`Documentation/userspace-api/media/v4l/`**: the V4L2 API reference.
- **`Documentation/driver-api/media/v4l2-subdev.rst`**: the subdev model.
- **`Documentation/admin-guide/media/imx.rst`**: i.MX media pipeline specifics.
- **OV5640 datasheet + software application notes (OmniVision)**: the register init sequences.
- **`v4l-utils`**: `v4l2-ctl`, `media-ctl`, `qv4l2`.
- **`yavta`**: a minimal V4L2 capture test tool, good for understanding the ioctl flow.

> Next chapter: **Chapter 88: USB UVC cameras.** The opposite of bring-up effort: plug in a webcam and `uvcvideo` just works, but understanding the UVC protocol and bandwidth budgeting tells you *why* it works and when it won’t.

Chapter 88: USB UVC cameras

What: USB webcams, the standardized **UVC** (USB Video Class) devices that the kernel's `uvcvideo` driver supports out of the box. Unlike the parallel-CSI sensors in Ch 87, a UVC camera is class-compliant. Plug it in, `/dev/video0` appears, and there is no driver to write. We cover the UVC protocol model, why it "just works," USB-2.0 bandwidth budgeting (the real constraint on i.MX6ULL), MJPEG vs YUYV vs H.264 modes, and the practical gotchas.

Why: for many products, a USB webcam is the *easiest* camera, no sensor bring-up, no register tables, no CSI timing. The trade-off is USB bandwidth and the fact that the camera's quality is whatever the webcam vendor shipped. Knowing the bandwidth math tells you what resolution/frame-rate is achievable on the i.MX6ULL's USB-2.0 host.

Focus: UVC is a class driver. The protocol is standardized, so one kernel driver covers every UVC camera. The hard part is no longer per-device driver code. It is bandwidth budgeting and format selection. A USB-2.0 bus is ~480 Mbps theoretical, ~320 Mbps practical. Uncompressed 1080p30 needs ~750 Mbps, impossible. MJPEG-compressed 1080p30 fits. Get the bandwidth math right and the rest is easy.

Tooling. This chapter uses `v4l-utils`, `gstreamer1.0-tools` + plugins, `ffmpeg`.

- **Ubuntu-base (target):** `apt install v4l-utils gstreamer1.0-tools gstreamer1.0-plugins-base gstreamer1.0-plugins-good gstreamer1.0-plugins-bad ffmpeg`
- **Buildroot:** `BR2_PACKAGE_V4L_UTILS=y BR2_PACKAGE_GSTREAMER1=y BR2_PACKAGE_FFMPEG=y` **Buildroot:** a configuration-driven build system that produces a complete root filesystem and related images.
- Full per-tool reference: Userspace tooling appendix. **MCU bridge:** Think of the rootfs as the firmware image's file-backed runtime environment. On an MCU you link everything into flash. On Linux, programs and config live in this mounted tree. **rootfs:** root filesystem, the directory tree mounted at / that contains /bin, /etc, /dev, and libraries.

88.1 Why UVC is different from CSI

	Parallel CSI (Ch 87)	USB UVC
Driver	custom per sensor (~3700 lines)	`uvcvideo` handles all (class-compliant)

	Parallel CSI (Ch 87)	USB UVC
Bring-up	sensor registers, CSI timing, power seq	plug in, `/dev/video0` appears
Bus bandwidth	up to ~96 MHz parallel	USB-2.0 ~320 Mbps practical
Quality	you pick the sensor	whatever the webcam vendor shipped
Compression	raw (you encode in software)	camera does MJPEG/H.264 in hardware
Cost	sensor module $2–10	webcam $10–50
Power	~100 mW	~500 mW, 2.5 W (USB powered)

The big advantage of UVC: the camera does the compression. A UVC camera with hardware MJPEG/H.264 offloads encoding from the i.MX6ULL (which has no VPU). For the i.MX6ULL, this often makes UVC the better choice. The webcam's silicon does the encoding, and the SoC only has to receive the compressed bytes.

88.2 The UVC protocol model

UVC is a USB device class (like HID for keyboards, mass-storage for drives). A UVC camera exposes:

- **A Video Control (VC) interface**: controls, brightness, exposure, focus, zoom, white balance.
- **One or more Video Streaming (VS) interfaces**: the actual frame data, with negotiated format + resolution + frame rate.

The camera advertises its capabilities in **descriptors** read at enumeration:

- Supported formats (YUYV / MJPEG / H.264 / NV12).
- Per-format supported frame sizes.
- Per-frame-size supported frame intervals (frame rates).

The `uvcvideo` driver parses these descriptors and exposes them through V4L2, exactly the same `/dev/video0` interface as a CSI camera. From the application's view, a UVC camera and a CSI camera are identical. Both are V4L2 video devices. Only the bring-up differs.

Isochronous vs bulk transfers

UVC video runs over USB **isochronous** transfers. These guarantee bandwidth but never retransmit, a dropped microframe is simply lost. The camera reserves a slice of each USB frame's bandwidth. This is why a UVC camera "claims" bandwidth on enumeration. Plugging two high-res cameras into one USB-2.0 bus can fail because the combined isochronous bandwidth exceeds the bus.

Some cameras support **bulk** transfers (used by a few MJPEG cameras), best-effort, retransmitted, but no bandwidth guarantee.

88.3 Bandwidth budgeting, the real constraint

USB-2.0 high-speed = 480 Mbps theoretical, ~320 Mbps usable for isochronous (after protocol overhead + the spec's 80% isochronous cap).

Uncompressed (YUYV, 16 bits/pixel) bandwidth:

Resolution	@30 fps	@15 fps	Fits USB-2.0?
320×240 (QVGA)	37 Mbps	18 Mbps	✓
640×480 (VGA)	147 Mbps	74 Mbps	✓ (VGA30 tight but OK)
1280×720 (720p)	442 Mbps	221 Mbps	✗ @30, ✓ @15
1920×1080 (1080p)	996 Mbps	498 Mbps	✗

MJPEG-compressed (typically ~10:1):

Resolution	@30 fps MJPEG	Fits?
640×480	~15 Mbps	✓ easily
1280×720	~44 Mbps	✓

Resolution	@30 fps MJPEG	Fits?
1920×1080	~100 Mbps	✓

Conclusion: for anything above VGA on USB-2.0, use **MJPEG** (or H.264). Uncompressed YUYV is limited to VGA30 or 720p15. The i.MX6ULL has only USB-2.0 (no USB-3.0), so this ceiling is hard.

The CPU cost: receiving MJPEG is cheap (just DMA). *Decoding* MJPEG to raw (for processing or display) costs CPU, i.MX6ULL software JPEG decode does ~30 fps VGA, ~10 fps 720p. If you only need to *store* or *forward* the MJPEG (e.g., to a network client that decodes), no decode needed, full frame rate.

> **MCU bridge:** Think of DMA like the MCU DMA controller you used for UART or SPI, but with cache coherency, scatter-gather descriptors, and kernel ownership rules added. **DMA:** Direct Memory Access. Hardware moves data to or from memory without the CPU copying each byte.

88.4 Bring-up, there isn’t much

```
[root@pa-mini:~]# lsusb
Bus 001 Device 005: ID 046d:0825 Logitech, Inc. Webcam C270

[root@pa-mini:~]# dmesg | grep -i uvc
usb 1-1.2: new high-speed USB device number 5
uvcvideo: Found UVC 1.00 device Webcam C270 (046d:0825)
input: Webcam C270 as /devices/.../input/input5
usb 1-1.2: UVC device initialized.

[root@pa-mini:~]# ls /dev/video*
/dev/video0
```

Plug in, `/dev/video0` appears. List capabilities:

```
[root@pa-mini:~]# v4l2-ctl --device /dev/video0 --list-formats-ext
ioctl: VIDIOC_ENUM_FMT
    Index: 0  Type: Video Capture  Pixel Format: 'YUYV'
        Size: Discrete 640x480
            Interval: Discrete 0.033s (30.000 fps)
        Size: Discrete 1280x720
            Interval: Discrete 0.133s (7.500 fps)
    Index: 1  Type: Video Capture  Pixel Format: 'MJPG' (compressed)
        Size: Discrete 1280x720
            Interval: Discrete 0.033s (30.000 fps)
        Size: Discrete 1920x1080
            Interval: Discrete 0.040s (25.000 fps)
```

Notice: YUYV maxes at 720p7.5 (bandwidth-limited), but MJPG does 720p30 and 1080p25 (compression fits the bus). The camera's descriptors tell you exactly what's possible.

Capture:

```
# Raw YUYV VGA
v4l2-ctl --device /dev/video0 --set-fmt-video=width=640,height=480,pixelformat=YUYV \
    --stream-mmap=3 --stream-count=30 --stream-to=vga.raw

# MJPEG 720p (frames are already JPEG — just save them)
v4l2-ctl --device /dev/video0 --set-fmt-video=width=1280,height=720,pixelformat=MJPG \
    --stream-mmap=3 --stream-count=1 --stream-to=frame.mjpg

# GStreamer: MJPEG → decode → display
gst-launch-1.0 v4l2src device=/dev/video0 ! image/
jpeg,width=1280,height=720,framerate=30/1 \
    ! jpegdec ! videoconvert ! fbdevsink
```

88.5 Controls

UVC cameras expose standardized controls:

```
[root@pa-mini:~]# v4l2-ctl --device /dev/video0 --list-ctrls
                     brightness 0x00980900 (int)    : min=0 max=255 default=128 value=128
                       contrast 0x00980901 (int)    : min=0 max=255 default=32 value=32
                     saturation 0x00980902 (int)    : min=0 max=255 default=32 value=32
        white_balance_automatic 0x0098090c (bool)   : default=1 value=1
                  power_line_freq 0x00980918 (menu)  : ...
                exposure_auto    0x009a0901 (menu)   : ...
            exposure_time_absolute 0x009a0902 (int)  : min=1 max=10000 value=166
                     focus_auto   0x009a090c (bool)   : ...

[root@pa-mini:~]# v4l2-ctl --set-ctrl=brightness=200
[root@pa-mini:~]# v4l2-ctl --set-ctrl=exposure_auto=1     # manual
[root@pa-mini:~]# v4l2-ctl --set-ctrl=exposure_time_absolute=500
```

These map to UVC's standardized control requests. `uvcvideo` translates V4L2 control IDs to USB control transfers. Not every camera supports every control, `--list-ctrls` shows what's available.

88.6 When UVC doesn't "just work"

- **"Driver needed" cameras**: some cheap cameras claim UVC but ship non-compliant descriptors, or need a vendor driver (rare these days). `dmesg` shows enumeration errors.
- **Bandwidth contention**: a USB WiFi dongle (Ch 92) + a UVC camera on the same USB-2.0 bus compete. The camera may fail to start, or the WiFi throughput drops. The i.MX6ULL has 2 USB controllers, put bandwidth-heavy devices on separate ones.

- **Power**: a 2.5 W webcam on a bus-powered hub may brown out. Use a powered hub or wire the camera's VBUS to a beefier rail.
- **H.264 UVC cameras**: some webcams output H.264. `uvcvideo` exposes it as a pixel format. You save the H.264 stream directly (no decode needed for storage/forwarding).
- **Multiple `/dev/videoN` per camera**: many UVC cameras expose two nodes, one for video, one for metadata. The video one is usually the lower-numbered.

88.7 USB gadget side, being a UVC camera

The inverse: making the i.MX6ULL *appear* as a webcam to a host PC (Ch 55's USB gadget). The `uvc` gadget function streams frames the i.MX6ULL generates (e.g., from a CSI camera) to a host as a standard webcam. Useful for "camera-over-USB" products. The ConfigFS `uvc` function (Ch 55.3) does this.

88.8 Key Notes

- UVC describes a class-based control and streaming interface. Reusing a class driver reduces integration work but does not remove device quirks or mode constraints.
- Raw and compressed formats shift the bottleneck between USB bandwidth, storage and CPU processing. Receiving compressed frames is not the same cost as decoding them.
- Devices sharing a USB controller also share scheduling, bandwidth and power constraints. A camera's advertised mode is not a system throughput result.
- Frame loss and disconnects need explicit handling. A valid `/dev/videoN` node does not establish uninterrupted capture.

88.9 Pitfalls

- **Uncompressed above VGA on USB-2.0.** Won't fit. Use MJPEG. The bandwidth table is the design constraint.
- **Software decode is the bottleneck, not the bus.** Receiving MJPEG is cheap. Decoding it to raw for display/processing on the GPU-less i.MX6ULL is slow. Forward the MJPEG undecoded where possible.
- **Bus-power brownout.** High-res webcams draw 1–2.5 W. Bus-powered hubs may not deliver it. Use a powered hub or dedicate a USB port with a strong VBUS rail.
- **Two bandwidth-heavy USB devices on one controller.** Camera + WiFi/Ethernet contend. Spread across the i.MX6ULL's two USB controllers.

- **`uvcvideo` quirks.** Some cameras need quirk flags (`uvcvideo.quirks=`). `dmesg` hints at enumeration issues. The `uvcvideo` source has a quirks table for known-bad cameras.
- **Frame drops on isochronous.** Isochronous has no retransmission. A missed USB microframe = a dropped/corrupt frame. Under heavy system load, frames drop silently. PREEMPT_RT (Ch 52A) helps for deterministic capture.
- **MJPEG is not a true video codec.** It is per-frame JPEG with no compression between frames. Typical ratio is 10:1, against H.264's 100:1. For bandwidth-critical streams, pick an H.264 UVC camera.

88.10 Going deeper

> **Driver choice:** Use the in-tree, maintained driver first. Use out-of-tree, spidev, or custom-driver paths only after you accept the kernel-version maintenance cost and document who owns updates.

- **`drivers/media/usb/uvc/`**: the `uvcvideo` driver. `uvc_driver.c` has the quirks table.
- **`Documentation/userspace-api/media/v4l/`**: V4L2 API (same as for CSI cameras).
- **USB Video Class specification (usb.org)**: the UVC standard.
- **`v4l-utils`**: `v4l2-ctl`, `qv4l2`, `v4l2-compliance` (tests a camera's V4L2 conformance).
- **`drivers/usb/gadget/function/f_uvc.c`**: the UVC *gadget* function (being a camera).
- **`guvcview, cheese`**: desktop tools for testing UVC cameras.

> **End of Group I, Cameras (Ch 87–88).** Parallel CSI (custom driver, you control the sensor) vs USB UVC (class-compliant, plug-and-play, camera does compression). On the GPU/VPU-less i.MX6ULL, UVC's hardware compression often wins for anything above VGA.

> Next chapter: **Chapter 89: I²S audio codecs.** Group J (Audio), the codec chips (WM8960, SGTL5000, ES8388, TLV320) that give the i.MX6ULL line-in/line-out/headphone/mic, and the ASoC machine driver that wires them up. **ASoC:** ALSA System-on-Chip, the embedded audio layer that connects CPU audio ports, codecs, and board wiring.

Chapter 89: I²S audio codecs

What: the analog-front-end chips that give the i.MX6ULL real audio: DAC for playback, ADC for capture, headphone and speaker drivers, mic preamps. Four codecs compared: **Cirrus WM8960** (the i.MX favourite), **NXP SGTL5000** (on many i.MX EVKs), **Everest ES8388** (cheap, ESP32-popular), **TI TLV320AIC3104** (industrial). Builds on Ch 53 (ALSA/ASoC framework). For each: the codec's role, the register/DAPM model, and a from-scratch ASoC codec driver. **ASoC:** ALSA System-on-Chip, the embedded audio layer that connects CPU audio ports, codecs, and board wiring. **ALSA:** Linux's kernel and user-space audio stack.

Why: the i.MX6ULL's SAI is a digital I²S serializer only. It has no analog audio. A codec adds the DAC/ADC/amp/mic. Ch 53 showed the three-driver ASoC split (CPU-DAI + codec + machine) with WM8960. This chapter goes *deep on the codec side*, how a codec driver is structured, what DAPM (Dynamic Audio Power Management) actually does, and how to write one from scratch.

Focus: a codec driver has three parts: a regmap for register access, DAPM widgets and routes for the analog graph, and DAI ops for I²S format negotiation. The regmap (Ch 50) handles register access. The DAPM graph models the analog signal paths: DAC → mixer → headphone amp → jack. Each block is powered up only when it lies on an active route, which saves power and avoids switching clicks. The DAI ops handle the I²S format negotiation. Once you understand these three, every codec driver looks familiar.

Tooling. This chapter uses `alsa-utils`, `i2c-tools`.

- **Ubuntu-base (target):** `apt install alsa-utils i2c-tools`
- **Buildroot:** `BR2_PACKAGE_ALSA_UTILS=y BR2_PACKAGE_I2C_TOOLS=y` **Buildroot:** a configuration-driven build system that produces a complete root filesystem and related images.
- Full per-tool reference: Userspace tooling appendix. **MCU bridge:** Think of the rootfs as the firmware image's file-backed runtime environment. On an MCU you link everything into flash. On Linux, programs and config live in this mounted tree. **rootfs:** root filesystem, the directory tree mounted at / that contains /bin, /etc, /dev, and libraries.

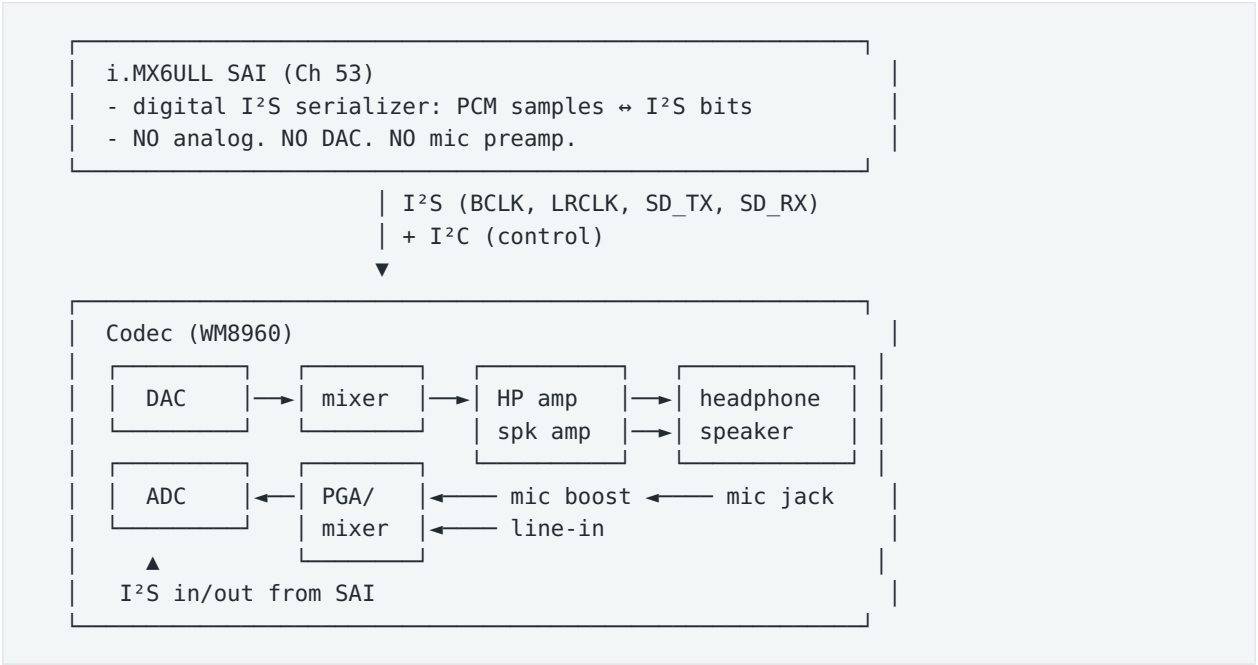
89.1 Codec comparison

	Cirrus WM8960	NXP SGTL5000	Everest ES8388	TI TLV320AIC3104
DAC SNR	98 dB	99.9 dB	96 dB	92 dB
ADC SNR	94 dB	90.5 dB	95 dB	92 dB
Outputs	stereo HP + class-D speaker	HP + line	HP + line	HP + line
Speaker amp	built-in 1 W class-D	no (needs ext amp)	no	no
Mic preamp	yes (+ boost)	yes	yes	yes
Control bus	I²C or 3-wire SPI	I²C	I²C	I²C
I²S roles	master or slave	master or slave	master or slave	master or slave
Sample rates	8–48 kHz	8–96 kHz	8–96 kHz	8–96 kHz
Volume price	$2–4	$3–5	$1–2	$3–5
Mainline driver	`wm8960.c`	`sgtl5000.c`	`es8328.c` (covers ES8388)	`tlv320aic3x.c`

Pick guide:

- **WM8960**: has a *built-in speaker amp*, drive a speaker directly, no external amp. Default for i.MX boards.
- **SGTL5000**: best SNR. On NXP EVKs. No speaker amp.
- **ES8388**: cheapest. Common on Chinese audio boards.
- **TLV320AIC3104**: industrial temperature range, robust.

89.2 What a codec does (and the SAI doesn’t)



The codec turns digital PCM into analog you can hear, and analog mic/line signals into digital PCM. The I²S carries the digital samples. I²C carries the control (volume, routing, power).

89.3 DAPM, Dynamic Audio Power Management

The single most important codec concept. A codec has many analog blocks (DAC, ADC, mixers, amps, bias generators). Powering them all on wastes power and the audible “click” when amps switch. **DAPM** models the analog signal flow as a *graph of widgets* and powers each widget only when it’s part of an active route from a running stream to a connected jack.

Widget types:

- **Endpoints**: `SND_SOC_DAPM_HP` (headphone jack), `SND_SOC_DAPM_SPK` (speaker), `SND_SOC_DAPM_MIC` (mic jack), `SND_SOC_DAPM_LINE`.
- **DAC/ADC**: `SND_SOC_DAPM_DAC`, `SND_SOC_DAPM_ADC`.
- **Mixers**: `SND_SOC_DAPM_MIXER` (combine sources).
- **PGAs / amps**: `SND_SOC_DAPM_PGA`, `SND_SOC_DAPM_OUT_DRV`.
- **Supplies**: `SND_SOC_DAPM_SUPPLY` (a shared resource like a bias generator or PLL).

> **MCU bridge:** Think of a PLL like the clock multiplier setup you used on STM32, but with more clock roots, gates, and consumers that Linux later needs to describe.
> **PLL:** Phase-Locked Loop, a clock block that multiplies a reference clock to create faster clocks.

And **routes** connecting them:

```
static const struct snd_soc_dapm_route wm8960_routes[] = {
    { "Headphone Jack", NULL, "LOUT1 PGA" },
    { "Headphone Jack", NULL, "ROUT1 PGA" },
    { "LOUT1 PGA", NULL, "Left Output Mixer" },
    { "Left Output Mixer", "PCM Playback Switch", "Left DAC" },
    { "Left DAC", NULL, "DACL" },
    ...
};
```

When you play audio: the stream activates “Left DAC”. DAPM walks the graph forward, DAC → mixer → PGA → Headphone Jack, and powers on each widget in that path. When you stop, it powers them down (in the right order to avoid pops). Blocks *not* in an active route stay off.

DAPM is why a well-written codec driver consumes µA at idle and doesn’t click, and why a *badly* written one pops on every play/stop. Getting the routes and power-sequencing right is the bulk of codec-driver effort.

89.4 How the mainline `wm8960` driver works

Source: `sound/soc/codecs/wm8960.c` (~1500 lines).

```
/* Simplified structure */
struct wm8960_priv {
    struct regmap *regmap;
    struct clk *mclk;
    struct snd_soc_component *component;
    /* PLL + clocking state */
};

/* Regmap config: WM8960 has 56 7-bit-address, 9-bit-value registers,
 * write-only (no read-back), so we use a register cache. */
static const struct regmap_config wm8960_regmap = {
    .reg_bits = 7,
    .val_bits = 9,
    .max_register = WM8960_PLL4,
    .reg_defaults = wm8960_reg_defaults,    /* the power-on values */
    .num_reg_defaults = ARRAY_SIZE(wm8960_reg_defaults),
    .cache_type = REGCACHE_RBTREE,           /* must cache — chip is write-only */
};
```

Note: WM8960 registers are *write-only* (you can't read them back). The regmap cache (Ch 50) is essential, it's the only way the driver knows the current register state. `cache_type = REGCACHE_RBTREE` + `reg_defaults` gives the driver a software shadow of the chip.

The component + DAI

```
static const struct snd_soc_dai_ops wm8960_dai_ops = {
    .hw_params  = wm8960_hw_params,     /* set sample rate, word length */
    .set_fmt    = wm8960_set_dai_fmt,    /* I²S vs LJ, master/slave */
    .set_sysclk = wm8960_set_dai_sysclk, /* MCLK config */
    .set_pll    = wm8960_set_dai_pll,    /* internal PLL for non-integer rates */
    .mute_stream = wm8960_mute,          /* soft-mute on start/stop (anti-pop) */
};

static struct snd_soc_dai_driver wm8960_dai = {
    .name = "wm8960-hifi",
    .playback = { .stream_name = "Playback", .channels_min = 1, .channels_max = 2,
                  .rates = SNDRV_PCM_RATE_8000_48000,
                  .formats = SNDRV_PCM_FMTBIT_S16_LE | SNDRV_PCM_FMTBIT_S24_LE },
    .capture  = { .stream_name = "Capture", ... },
    .ops = &wm8960_dai_ops,
    .symmetric_rate = 1,
};

static const struct snd_soc_component_driver wm8960_component = {
    .probe              = wm8960_probe,
    .set_bias_level     = wm8960_set_bias_level,   /* power state transitions */
    .controls           = wm8960_snd_controls,      /* volume, mute, etc. */
    .num_controls       = ARRAY_SIZE(wm8960_snd_controls),
    .dapm_widgets       = wm8960_dapm_widgets,
    .num_dapm_widgets   = ARRAY_SIZE(wm8960_dapm_widgets),
    .dapm_routes        = wm8960_dapm_routes,
    .num_dapm_routes    = ARRAY_SIZE(wm8960_dapm_routes),
};
```

Probe

```
static int wm8960_i2c_probe(struct i2c_client *client)
{
    struct wm8960_priv *wm8960;

    wm8960 = devm_kzalloc(&client->dev, sizeof(*wm8960), GFP_KERNEL);
    wm8960->regmap = devm_regmap_init_i2c(client, &wm8960_regmap);

    /* Reset the chip to known defaults */
    regmap_write(wm8960->regmap, WM8960_RESET, 0);

    i2c_set_clientdata(client, wm8960);

    /* Register the ASoC component + DAI */
    return devm_snd_soc_register_component(&client->dev, &wm8960_component,
                                           &wm8960_dai, 1);
}
```

That’s the whole codec driver shape: regmap + DAI ops + component (controls + DAPM). The machine driver (Ch 53) wires this codec’s DAI to the SAI’s DAI.

89.5 Writing an ASoC codec driver from scratch

We’ll write a minimal codec driver for a simplified codec (model: a stereo-DAC-only codec with a headphone output and a master-volume control). Real codecs are bigger, but the *structure* is identical. ~200 lines.

`mycodec.c`:

```
#include <linux/init.h>
#include <linux/module.h>
#include <linux/i2c.h>
#include <linux/regmap.h>
#include <sound/soc.h>
#include <sound/tlv.h>

/* Imaginary register map */
#define REG_RESET     0x00
#define REG_POWER     0x01
#define REG_DAC_CTRL  0x02
#define REG_VOL_L     0x03
#define REG_VOL_R     0x04
#define REG_FORMAT    0x05
#define REG_MAX       0x10

static const struct reg_default mycodec_reg_defaults[] = {
    { REG_POWER, 0x00 }, { REG_DAC_CTRL, 0x00 },
    { REG_VOL_L, 0x79 }, { REG_VOL_R, 0x79 },
    { REG_FORMAT, 0x02 },
};

static const struct regmap_config mycodec_regmap = {
    .reg_bits = 8,
    .val_bits = 8,
    .max_register = REG_MAX,
    .reg_defaults = mycodec_reg_defaults,
    .num_reg_defaults = ARRAY_SIZE(mycodec_reg_defaults),
    .cache_type = REGCACHE_RBTREE,
};

/* --- Controls: a stereo master volume (0..127, -73 to +6 dB) --- */
static const DECLARE_TLV_DB_SCALE(vol_tlv, -7300, 100, 0);

static const struct snd_kcontrol_new mycodec_controls[] = {
    SOC_DOUBLE_R_TLV("Master Playback Volume",
                     REG_VOL_L, REG_VOL_R, 0, 127, 0, vol_tlv),
    SOC_SINGLE("DAC Mute Switch", REG_DAC_CTRL, 3, 1, 0),
};

/* --- DAPM widgets + routes --- */
static const struct snd_soc_dapm_widget mycodec_widgets[] = {
    SND_SOC_DAPM_DAC("DAC", "Playback", REG_POWER, 0, 0),
    SND_SOC_DAPM_PGA("HP Amp", REG_POWER, 1, 0, NULL, 0),
    SND_SOC_DAPM_OUTPUT("HPOUT"),
```

```
};

static const struct snd_soc_dapm_route mycodec_routes[] = {
    { "HP Amp", NULL, "DAC" },
    { "HPOUT", NULL, "HP Amp" },
};

/* --- DAI ops --- */
static int mycodec_hw_params(struct snd_pcm_substream *substream,
                             struct snd_pcm_hw_params *params,
                             struct snd_soc_dai *dai)
{
    struct snd_soc_component *comp = dai->component;
    u8 fmt;

    /* Set word length from params */
    switch (params_width(params)) {
    case 16: fmt = 0x00; break;
    case 24: fmt = 0x02; break;
    default: return -EINVAL;
    }
    snd_soc_component_update_bits(comp, REG_FORMAT, 0x03, fmt);
    return 0;
}

static int mycodec_set_fmt(struct snd_soc_dai *dai, unsigned int fmt)
{
    struct snd_soc_component *comp = dai->component;

    /* We only support I²S, codec as slave */
    if ((fmt & SND_SOC_DAIFMT_FORMAT_MASK) != SND_SOC_DAIFMT_I2S)
        return -EINVAL;
    if ((fmt & SND_SOC_DAIFMT_CLOCK_PROVIDER_MASK) != SND_SOC_DAIFMT_CBC_CFC)
        return -EINVAL;     /* codec consumer (slave) only */
    return 0;
}

static const struct snd_soc_dai_ops mycodec_dai_ops = {
    .hw_params = mycodec_hw_params,
    .set_fmt   = mycodec_set_fmt,
};

static struct snd_soc_dai_driver mycodec_dai = {
    .name = "mycodec-hifi",
    .playback = {
        .stream_name = "Playback",
        .channels_min = 2, .channels_max = 2,
        .rates = SNDRV_PCM_RATE_8000_48000,
        .formats = SNDRV_PCM_FMTBIT_S16_LE | SNDRV_PCM_FMTBIT_S24_LE,
    },
    .ops = &mycodec_dai_ops,
};

static const struct snd_soc_component_driver mycodec_component = {
    .controls         = mycodec_controls,
    .num_controls     = ARRAY_SIZE(mycodec_controls),
    .dapm_widgets     = mycodec_widgets,
    .num_dapm_widgets = ARRAY_SIZE(mycodec_widgets),
    .dapm_routes      = mycodec_routes,
```

```
    .num_dapm_routes  = ARRAY_SIZE(mycodec_routes),
    .idle_bias_on     = 1,
    .use_pmdown_time  = 1,
    .endianness       = 1,
};

static int mycodec_probe(struct i2c_client *client)
{
    struct regmap *regmap;

    regmap = devm_regmap_init_i2c(client, &mycodec_regmap);
    if (IS_ERR(regmap))
        return PTR_ERR(regmap);

    /* Reset */
    regmap_write(regmap, REG_RESET, 0x00);

    return devm_snd_soc_register_component(&client->dev, &mycodec_component,
                                           &mycodec_dai, 1);
}

static const struct of_device_id mycodec_of_match[] = {
    { .compatible = "linuxlearn,mycodec" },
    { }
};
MODULE_DEVICE_TABLE(of, mycodec_of_match);

static const struct i2c_device_id mycodec_id[] = { { "mycodec", 0 }, { } };
MODULE_DEVICE_TABLE(i2c, mycodec_id);

static struct i2c_driver mycodec_driver = {
    .driver = {
        .name = "mycodec",
        .of_match_table = mycodec_of_match,
    },
    .probe = mycodec_probe,
    .id_table = mycodec_id,
};
module_i2c_driver(mycodec_driver);

MODULE_LICENSE("GPL");
```

Three things to notice:

1. **`SOC_DOUBLE_R_TLV`**: Declares a stereo volume control with a dB scale (`vol_tlv`). ALSA tools (`alsamixer`) show it as a slider with dB readout. The macro handles the L/R register pair.
2. **DAPM widgets reference power-control bits**: `SND_SOC_DAPM_DAC("DAC", "Playback", REG_POWER, 0, 0)` means “the DAC widget’s power is bit 0 of REG_POWER.” DAPM sets/clears that bit as the route activates/deactivates.
3. **`snd_soc_component_update_bits`**: The ASoC wrapper over regmap for read-modify-write of a register field.

The machine driver (Ch 53) binds this codec to the SAI:

```
sound {
    compatible = "fsl,imx-audio-mycodec";   /* or simple-audio-card */
    audio-codec = <&mycodec>;
    audio-cpu = <&sai2>;
    audio-routing = "Headphone Jack", "HPOUT";
};
```

Or with `simple-audio-card`:

```
sound {
    compatible = "simple-audio-card";
    simple-audio-card,name = "imx-mycodec";
    simple-audio-card,format = "i2s";
    simple-audio-card,bitclock-master = <&cpudai>;
    simple-audio-card,frame-master = <&cpudai>;
    cpudai: simple-audio-card,cpu { sound-dai = <&sai2>; };
    simple-audio-card,codec { sound-dai = <&mycodec>; };
};
```

Test:

```
[root@pa-mini:~]# insmod mycodec.ko
[root@pa-mini:~]# aplay -l
card 0: imxmycodec [imx-mycodec], device 0: ...
[root@pa-mini:~]# alsamixer        # shows "Master Playback Volume"
[root@pa-mini:~]# aplay test.wav   # plays through the codec
```

What we got, ~200 lines:

- A registered ASoC codec component.
- A master-volume control visible in `alsamixer`.
- DAPM power management of the DAC and HP amp.
- I²S format negotiation.

What WM8960 adds (the other ~1300 lines): ADC + capture path, mic preamp + boost, the speaker class-D amp, the internal PLL for non-integer sample rates, jack detection, dozens more controls, full DAPM graph with mixers.

89.6 Codec differences that matter

- **WM8960**: built-in 1 W class-D speaker amp, the `Speaker` DAPM route + the `SPK_LP/LN/RP/RN` outputs. Drive a speaker directly. Unique among these four.
- **SGTL5000**: best SNR. Needs a specific power-up sequence (the chip has analog + digital + I/O rails that must come up in order). The mainline driver enforces it.
- **ES8388**: cheap. The mainline `es8328.c` driver covers it but the ES8388 has minor register differences, check the compatible.

- **TLV320AIC3104**: highly configurable routing (a “miniDSP” in bigger siblings). Industrial temp range.

89.7 Key Notes

- A codec driver joins register control, digital audio format and analog routing. The board’s machine description connects it to the CPU audio interface.
- DAPM describes which analog blocks and outputs must be powered for an active route. A running PCM transfer can coexist with a silent physical output.
- Clock roles, sample format and accepted rates must agree at both ends. Register-cache restoration also matters when the codec loses power.
- Mute and power sequencing affect clicks, current and output safety. They belong to the audio lifecycle rather than a single playback command.

89.8 Pitfalls

- **Write-only registers without a cache.** WM8960 (and many codecs) can’t be read back. Without `cache_type` + `reg_defaults`, the driver loses track of state. Always cache.
- **Master/slave clock mismatch.** If both SAI and codec think they’re I²S master, no bit clock. One must be slave. Set `bitclock-master`/`frame-master` in the machine DT.
- **MCLK rate wrong.** Codecs need MCLK = a specific multiple of the sample rate (e.g., 256× or 512× fs). For 48 kHz: 12.288 MHz or 24.576 MHz. Wrong MCLK → wrong pitch or no audio. Configure the SAI’s clock to produce the right MCLK.
- **DAPM routes incomplete.** Audio “plays” (DMA flows) but is silent because DAPM never powered the output path. Trace the dapm debugfs.

> **MCU bridge:** Think of DMA like the MCU DMA controller you used for UART or SPI, but with cache coherency, scatter-gather descriptors, and kernel ownership rules added. **DMA:** Direct Memory Access. Hardware moves data to or from memory without the CPU copying each byte.

- **Pop/click on play/stop.** Missing soft-mute or wrong power-up/down order. Implement `mute_stream` and order DAPM events.
- **Wrong codec I²C address.** WM8960 = 0x1A, SGTL5000 = 0x0A, ES8388 = 0x10/0x11. Check `i2cdetect`.
- **SGTL5000 power sequencing.** Its rails must come up in order. Out-of-order = chip doesn’t respond. The mainline driver handles it. A hand-rolled one must too.

- **Non-integer sample rates.** 44.1 kHz needs a fractional PLL ratio from a 12 MHz MCLK. Codecs have an internal PLL. Configure `set_pll` or you only get the integer-related rates.

89.9 Going deeper

- **`sound/soc/codecs/wm8960.c`**: the production codec driver. Compare structure to the from-scratch version.
- **`sound/soc/codecs/sgtl5000.c`**: note the power-sequencing.
- **`sound/soc/codecs/es8328.c`**: ES8388/ES8328.
- **`sound/soc/codecs/tlv320aic3x.c`**: TI codec.
- **`Documentation/sound/soc/dapm.rst`**: the DAPM model, in depth.
- **`Documentation/sound/soc/codec.rst`**: the codec-driver author's guide.
- **`include/sound/soc.h`**: the `SOC_*` control macros and DAPM widget macros.
- **WM8960 datasheet (Cirrus)**: register map. The DAPM graph mirrors its block diagram.

> Next chapter: **Chapter 90: Digital class-D amplifiers.** When you don't need a full codec, just turn I²S into sound through a speaker. MAX98357A (no control at all), TAS5805M (DSP amp), PCM5102A (DAC).

Chapter 90: Digital class-D amplifiers

What: “audio output without a codec.” These chips take I²S directly and drive a speaker (class-D amp) or headphones (DAC). They have no ADC, no mic input, and no input mixer, just a digital-in, analog-out path. Three chips: **TI MAX98357A** (pure I²S → 3.2 W speaker, *no control interface at all*), **TI TAS5805M** (I²C-controlled DSP class-D amp), **TI PCM5102A** (I²S → headphone DAC, no control). For each: the ASoC component model for a control-less or simply-controlled device, and from-scratch drivers. The MAX98357A driver is extremely short. The TAS5805M shows DSP-coefficient loading. **ASoC:** ALSA System-on-Chip, the embedded audio layer that connects CPU audio ports, codecs, and board wiring.

Why: for *playback-only* products, a Bluetooth speaker, a voice-announcement system, a doorbell chime, a kiosk that plays sounds, a full codec (Ch 89) is overkill. All you need is to turn I²S into sound. These chips do exactly that, cheaper and simpler. The MAX98357A needs zero configuration. Wire I²S, pick L/R/mono with a resistor, and it plays.

Focus: an amp without a control bus is the simplest ASoC component you can write. The MAX98357A driver is ~100 lines and has no registers, it’s a DAPM widget (the amp) + a DAI (the I²S sink) + maybe an enable GPIO. The TAS5805M adds I²C control + a DSP that needs a coefficient blob loaded. Driver complexity tracks chip capability, dumb amp short, DSP amp long. **MCU bridge:** Think of Linux GPIO like the same pin set/reset block you used on STM32, but accessed through a kernel subsystem that owns numbering, direction, interrupts, and user-space exposure. **GPIO:** General-Purpose Input/Output, a pin controlled as a digital input, output, or interrupt source.

Tooling. This chapter uses `alsa-utils`, `i2c-tools`. **ALSA:** Linux’s kernel and user-space audio stack.

- **Ubuntu-base (target):** `apt install alsa-utils i2c-tools`
- **Buildroot:** `BR2_PACKAGE_ALSA_UTILS=y BR2_PACKAGE_I2C_TOOLS=y` **Buildroot:** a configuration-driven build system that produces a complete root filesystem and related images.
- Full per-tool reference: Userspace tooling appendix. **MCU bridge:** Think of the rootfs as the firmware image’s file-backed runtime environment. On an MCU you link everything into flash. On Linux, programs and config live in this mounted tree. **rootfs:** root filesystem, the directory tree mounted at / that contains /bin, /etc, /dev, and libraries.

90.1 Chip comparison

	TI MAX98357A	TI TAS5805M	TI PCM5102A
Function	I²S → speaker amp	I²S → DSP → speaker amp	I²S → headphone DAC
Power output	3.2 W (4 Ω)	2× 23 W (stereo)	line/headphone level
Control	none (config pins)	I²C (DSP + volume + EQ)	none (config pins)
DSP features	none	EQ, crossover, DRC, bass boost	none
Channels	mono (L/R/avg via pin)	stereo	stereo
DAC SNR	98 dB	110 dB	112 dB
Sample rates	8–96 kHz	32–96 kHz	8–384 kHz
Volume price	$1.50–2.50	$4–6	$2–4
Mainline driver	`max98357a.c`	`tas5805m.c`	`pcm5102a` via `pcm512x.c` family / `simple-audio-card`

Pick guide:

- **MAX98357A**: simplest possible “make sound from I²S.” Mono. Bluetooth speakers, voice prompts. *Zero config.*
- **TAS5805M**: when you want EQ / bass boost / dynamic-range control done in hardware, a real speaker product with tuning.
- **PCM5102A**: headphone/line-out DAC. Audiophile-grade SNR. No amp.

90.2 MAX98357A, the zero-config amp

The MAX98357A is as simple as it gets. It has **no I²C, no SPI, no registers**. Configuration is via *hardware pins*:

- **SD_MODE pin**: tied via a resistor to select left channel / right channel / (L+R)/2 average / shutdown. (A resistor divider on one pin encodes 4 states.)
- **GAIN_SLOT pin**: sets amp gain (3/6/9/12/15 dB) and which I²S slot to use.

Wire up I²S (BCLK, LRCLK, DIN), power, and a speaker. It plays. The “driver” just needs to:

1. Declare a DAI (the I²S sink).
2. Declare a DAPM widget (the amp output).
3. Optionally manage an enable/shutdown GPIO.

There is no register access in the driver because the chip has no registers.

90.3 Writing a MAX98357A-style driver from scratch

`mymax98357.c`:

```
#include <linux/init.h>
#include <linux/module.h>
#include <linux/platform_device.h>
#include <linux/gpio/consumer.h>
#include <sound/soc.h>
#include <sound/soc-dapm.h>

struct mymax {
    struct gpio_desc *sdmode_gpio;     /* shutdown/mode control */
};

/* DAPM: the speaker amp output, gated by the SD_MODE GPIO */
static int mymax_event(struct snd_soc_dapm_widget *w,
                       struct snd_kcontrol *kc, int event)
{
    struct snd_soc_component *comp = snd_soc_dapm_to_component(w->dapm);
    struct mymax *m = snd_soc_component_get_drvdata(comp);

    /* Power the amp on when the route activates, off when it deactivates */
    gpiod_set_value_cansleep(m->sdmode_gpio,
                             SND_SOC_DAPM_EVENT_ON(event) ? 1 : 0);
    return 0;
}

static const struct snd_soc_dapm_widget mymax_widgets[] = {
    SND_SOC_DAPM_OUT_DRV_E("Amp", SND_SOC_NOPM, 0, 0, NULL, 0,
                           mymax_event,
                           SND_SOC_DAPM_PRE_PMU | SND_SOC_DAPM_POST_PMD),
    SND_SOC_DAPM_OUTPUT("Speaker"),
```

```
};

static const struct snd_soc_dapm_route mymax_routes[] = {
    { "Amp", NULL, "HiFi Playback" },
    { "Speaker", NULL, "Amp" },
};

static const struct snd_soc_component_driver mymax_component = {
    .dapm_widgets     = mymax_widgets,
    .num_dapm_widgets = ARRAY_SIZE(mymax_widgets),
    .dapm_routes      = mymax_routes,
    .num_dapm_routes  = ARRAY_SIZE(mymax_routes),
    .idle_bias_on     = 1,
};

/* The DAI: a pure I²S sink. No hw_params logic needed — the chip
 * auto-detects sample rate from the I²S clocks. */
static struct snd_soc_dai_driver mymax_dai = {
    .name = "HiFi",
    .playback = {
        .stream_name = "HiFi Playback",
        .formats = SNDRV_PCM_FMTBIT_S16_LE | SNDRV_PCM_FMTBIT_S24_LE
                 | SNDRV_PCM_FMTBIT_S32_LE,
        .rates = SNDRV_PCM_RATE_8000_96000,
        .rate_min = 8000, .rate_max = 96000,
        .channels_min = 1, .channels_max = 2,
    },
};

static int mymax_probe(struct platform_device *pdev)
{
    struct mymax *m;

    m = devm_kzalloc(&pdev->dev, sizeof(*m), GFP_KERNEL);
    if (!m) return -ENOMEM;

    /* Optional shutdown GPIO (SD_MODE) */
    m->sdmode_gpio = devm_gpiod_get_optional(&pdev->dev, "sdmode", GPIOD_OUT_LOW);
    if (IS_ERR(m->sdmode_gpio))
        return PTR_ERR(m->sdmode_gpio);

    platform_set_drvdata(pdev, m);

    return devm_snd_soc_register_component(&pdev->dev, &mymax_component,
                                           &mymax_dai, 1);
}

static const struct of_device_id mymax_of_match[] = {
    { .compatible = "linuxlearn,mymax98357a" },
    { }
};
MODULE_DEVICE_TABLE(of, mymax_of_match);

static struct platform_driver mymax_driver = {
    .driver = {
        .name = "mymax98357a",
        .of_match_table = mymax_of_match,
    },
    .probe = mymax_probe,
```

```
};
module_platform_driver(mymax_driver);

MODULE_LICENSE("GPL");
```

DT:

```
amp: codec {
    compatible = "linuxlearn,mymax98357a";
    #sound-dai-cells = <0>;
    sdmode-gpios = <&gpio4 14 GPIO_ACTIVE_HIGH>;
};

sound {
    compatible = "simple-audio-card";
    simple-audio-card,name = "imx-spk";
    simple-audio-card,format = "i2s";
    simple-audio-card,bitclock-master = <&cpudai>;
    simple-audio-card,frame-master = <&cpudai>;
    cpudai: simple-audio-card,cpu { sound-dai = <&sai2>; };
    simple-audio-card,codec { sound-dai = <&amp>; };
};
```

Note: it's a **platform_device** (not I²C/SPI) because there's no control bus, the chip is configured by pins. Test:

```
[root@pa-mini:~]# insmod mymax98357a.ko
[root@pa-mini:~]# aplay -l
card 0: imxspk [imx-spk], device 0: ...
[root@pa-mini:~]# aplay test.wav     # plays through the speaker
```

~100 lines. The `OUT_DRV_E` widget's event callback toggles the SD_MODE GPIO with the route. The amp is on during playback and off when idle, saving power. That is the only "logic" in the driver.

The mainline `max98357a.c` is similar (~150 lines), adds the GAIN gpio and a small refinement, but it's the same shape. This is the floor of ASoC complexity.

90.4 PCM5102A, same idea, headphone DAC

The PCM5102A is also control-less (config via pins: format, de-emphasis, mute). It outputs analog line/headphone level from I²S. The "driver" is the same shape as the MAX98357A, a DAI + a DAPM output widget, no registers.

In fact, many such control-less DACs/amps don't need a *custom* driver at all, `simple-audio-card` can use a generic "spdif-transmitter" or "dummy codec" stand-in, or the chip-specific stub driver. For PCM5102A, the `pcm512x` family driver covers the controllable variants (PCM5121/5122 with I²C). The bare PCM5102A uses a fixed-function stub.

90.5 TAS5805M, the DSP amp

The TAS5805M sits at the other end. It is a 2×23 W stereo class-D amp with an **on-chip DSP** that provides:

- Parametric EQ (15 biquads per channel).
- Dynamic range control (compression/limiting).
- Bass boost / loudness.
- Crossover filters (for biamped speakers).

All of this is configured over I²C. The DSP runs a *coefficient set*. You compute biquad coefficients (for example, from a target frequency response) and load them into the chip's coefficient RAM.

The configuration model

```
1. Reset, set I²S format, sample rate.
2. Load a "book + page" of register settings (TI's tool generates these).
3. Load DSP coefficients (the EQ/DRC/crossover tuning) into coefficient pages.
4. Set volume.
5. Switch from HiZ → PLAY mode.
```

The "book + page" model: the TAS5805M's register space is paged (256 registers per page, multiple pages, organized into "books"). You select a book (write reg 0x7F), select a page (write reg 0x00), then access the page's registers. The DSP coefficients live in specific book/page locations.

TI provides **PPC3** (a GUI tool) to design the EQ/DRC and export a register-write sequence (a `.h` file or a binary blob). The driver replays this blob at probe, like the camera (Ch 87) and ToF (Ch 72) register blobs.

Mainline driver

`sound/soc/codecs/tas5805m.c` registers an ASoC component with:

- A volume control.
- The init sequence loaded from a firmware blob (`/lib/firmware/tas5805m_dsp.bin`).
- DAPM for the amp.

```
/* Simplified: load the DSP config blob */
static int tas5805m_load_config(struct tas5805m_priv *tas)
{
    const struct firmware *fw;
    int err = request_firmware(&fw, "tas5805m_dsp.bin", tas->dev);
    if (err) return err;
```

```
    /* The blob is a sequence of (reg, val) or book/page-switch commands */
    err = tas5805m_process_block(tas, fw->data, fw->size);
    release_firmware(fw);
    return err;
}
```

The blob is your speaker tuning. A from-scratch TAS5805M driver would: regmap for the paged register access, a firmware-blob loader for the DSP config, a volume control, and DAPM, roughly the WM8960 shape from Ch 89 plus the book/page paging and the firmware blob. We won't reproduce the full driver. The *new* concept beyond Ch 89 is the book/page paging + the externally-designed DSP coefficient blob.

> **MCU bridge:** Think of regmap like a typed wrapper around your read_reg() and write_reg() helpers, with caching, locking, and bus differences handled centrally.
> **regmap:** a kernel helper that wraps register reads and writes over I2C, SPI, or MMIO.

Designing the DSP config

The interesting part for a product: you measure your speaker's frequency response (with a calibrated mic), design a corrective EQ in PPC3, and export the blob. The TAS5805M then *fixes the speaker's acoustic flaws in hardware*. Good DSP tuning makes a $5 speaker sound much better than its physical parts deserve. This is why DSP class-D amps are standard in smart speakers.

90.6 The Bluetooth-speaker product pattern

A common product: a Bluetooth speaker on i.MX6ULL.

```
Phone ──[A2DP/Bluetooth]──► BlueZ (Ch 95) ──► ALSA ──► SAI (I²S) ──► TAS5805M ──► speaker
```

- BlueZ receives the A2DP stream, decodes SBC/AAC to PCM.
- PCM goes to ALSA → the SAI → I²S.
- The TAS5805M's DSP applies EQ + limiting, drives the speaker.

The i.MX6ULL does the Bluetooth + decode. The TAS5805M does the analog + acoustic tuning. No full codec needed (playback only). This is the standard "amp without a codec" product pattern.

90.7 Key Notes

- A digital amplifier can accept I2S without offering a register interface. Linux still needs a component and DAI description to connect the audio path.

- Hardware straps, enable signals and DSP configuration have different ownership paths. A zero-control-bus design is not necessarily a zero-policy design.
- Speaker load, supply capacity, thermal behavior and EMI constrain the output stage. Successful audio streaming does not qualify delivered power or acoustic performance.
- DAPM and configuration sequencing should leave the amplifier in a defined state when idle, starting, stopping or recovering from a reset.

90.8 Pitfalls

- **MAX98357A SD_MODE resistor wrong.** The resistor value encodes channel select. Wrong value = wrong channel or shutdown. Use the datasheet's resistor table exactly.
- **No DAI → no sound.** Even a control-less amp needs an ASoC DAI registered so the machine driver can bind it. The "dummy codec" must still expose a DAI.
- **MAX98357A idle hiss.** If SD_MODE stays high when idle, the amp's output stage hisses. Gate it via the DAPM event callback (amp off when no stream).
- **TAS5805M book/page confusion.** Forgetting to select the right book/page before a register access writes to the wrong location → garbage config or no sound. The paging model is error-prone. Follow TI's sequence exactly.
- **TAS5805M PLAY/HiZ state.** The amp must be explicitly switched to PLAY mode after config. Left in HiZ → silent.
- **Sample-rate mismatch.** The amp auto-detects rate from I²S clocks (MAX98357A) or must be told (TAS5805M). A mismatch → wrong pitch or no audio.
- **Speaker impedance vs amp.** A 3.2 W amp into a 4 Ω speaker is fine. Into 32 Ω headphones it's too weak. Match amp to load.
- **Class-D EMI.** Class-D amps switch at ~400 kHz. The speaker leads radiate. Keep leads short, use ferrites/filters, or fail EMC.
- **Power supply for the amp.** A 2×23 W TAS5805M needs a beefy supply (peaks > 4 A). Brownouts cause clipping/reset. Budget the rail.

90.9 Going deeper

- **`sound/soc/codecs/max98357a.c`**: the production driver. Compare to the from-scratch version, nearly identical, ~150 lines.
- **`sound/soc/codecs/tas5805m.c`**: the DSP amp driver with firmware-blob loading.
- **`sound/soc/codecs/pcm512x.c`**: PCM510x/512x family.

- **`Documentation/sound/soc/`**: ASoC framework (same as Ch 89).
- **MAX98357A datasheet (TI/Maxim)**: the SD_MODE / GAIN resistor tables.
- **TAS5805M datasheet + PPC3 tool (TI)**: the book/page model and the EQ designer.
- **TI app note SLAA738**: TAS5805M DSP configuration.

> **End of Group J, Audio (Ch 89–90).** Full codecs (Ch 89: DAC+ADC+amp+mic, the WM8960 class) vs playback-only amps/DACs (Ch 90: I²S → speaker/headphone, the MAX98357A/TAS5805M class). The driver complexity spectrum, control-less amp (~100 lines) → simple codec (~200) → full codec (~1500) → DSP amp (regmap + firmware blob), maps directly to the chip's capabilities.

> Next chapter: **Chapter 91: SDIO WiFi.** Group K (WiFi) opens with the SDIO-attached modules (AP6212, RTL8189), firmware loading, the brcmfmac/rtl driver stack, and the SDIO bring-up that trips up every new board.

Chapter 91: SDIO WiFi

IRQ: interrupt request, the signal path that tells the CPU or interrupt controller that hardware needs service. **MCU bridge:** Think of an IRQ like an EXTI/NVIC interrupt path, except Linux splits the hard interrupt from deferred work and must share lines across drivers.

What: WiFi modules attached over the **SDIO** bus (the same physical interface as an SD card, repurposed for I/O). Three modules compared: **AP6212** (Broadcom BCM43438, on many i.MX boards), **RTL8189FTV** (Realtek), **SD8801** (Marvell/NXP). Builds on Ch 55E (the WiFi stack). For each: the SDIO bring-up sequence, firmware + NVRAM loading, and, since full-MAC WiFi drivers are 30k+ lines you won't write from scratch, a *trace of how a packet flows through the stack* and how to bring up the SDIO transport (the part that actually trips up every new board). **MAC:** Media Access Control in networking and radio chapters. It is the layer that owns framing and medium access.

Why: SDIO WiFi is the standard embedded WiFi: soldered-down, low-cost, no USB port consumed. It is also the hardest peripheral to bring up on a new board. Five things must be exactly right: the SDIO transport, the power sequence, the 32 kHz clock, the per-board NVRAM, and the firmware blob. When any one is wrong, the symptom is usually "nothing in dmesg." This chapter is mostly about the bring-up sequence and debugging.

Focus: the WiFi chip is a full-MAC co-processor. The Linux driver does two things only: it loads firmware, and it shuttles SDIO packets. The chip runs the entire 802.11 stack in its own firmware. The Linux driver (brcmfmac, etc.) loads that firmware over SDIO at boot, then ferries data packets and control commands back and forth. Bring-up is two jobs: get the SDIO bus working, then supply the right firmware and NVRAM. After that, the chip handles the actual WiFi.

Tooling. This chapter uses `wpa_supplicant`, `iw`, chip firmware blob (Cypress/Realtek/Marvell).

- **Ubuntu-base (target):** `apt install wpasupplicant iw firmware-brcm80211 firmware-realtek`
- **Buildroot:** `BR2_PACKAGE_WPA_SUPPLICANT=y BR2_PACKAGE_IW=y` **Buildroot:** a configuration-driven build system that produces a complete root filesystem and related images.

- Full per-tool reference: Userspace tooling appendix. **MCU bridge:** Think of the rootfs as the firmware image's file-backed runtime environment. On an MCU you link everything into flash. On Linux, programs and config live in this mounted tree. **rootfs:** root filesystem, the directory tree mounted at / that contains /bin, /etc, /dev, and libraries.

91.1 Module comparison

Driver choice: Use the in-tree, maintained driver first. Use out-of-tree, spidev, or custom-driver paths only after you accept the kernel-version maintenance cost and document who owns updates.

	AP6212 (BCM43438)	RTL8189FTV	SD8801 (Marvell)
Vendor	Broadcom (AmPak module)	Realtek	Marvell/NXP
Standard	802.11 b/g/n (2.4 GHz)	802.11 b/g/n	802.11 b/g/n
Combo	+ Bluetooth 4.1	WiFi only	+ Bluetooth (8997 variant)
Bus	SDIO 2.0	SDIO	SDIO
Driver	`brcmfmac` (in-tree)	`rtl8189es`/`rtl8189fs` (out-of-tree) or `rtw88`	`mwifiex` (in-tree)
Firmware	yes + per-board NVRAM	yes (in driver or blob)	yes
32 kHz clock	required	required	required
Throughput (real)	~30–50 Mbps	~30–40 Mbps	~40 Mbps
Volume price	$2–4	$1.50–3	$3–5

Pick guide:

- **AP6212**: well-supported mainline (`brcmfmac`), combo BT. Default for i.MX boards.
- **RTL8189**: cheapest. But the driver is usually out-of-tree (DKMS pain, see Ch 92's lesson, applies here too).
- **SD8801**: mainline `mwifiex`, decent. Combo variants exist.

Strongly prefer modules with in-tree drivers, AP6212 with brcmfmac, SD8801 with mwifiex. Out-of-tree drivers (most RTL8189 variants) are a maintenance burden, see §91.8.

91.2 SDIO, SD card bus, repurposed

SDIO uses the same physical bus and protocol as an SD card (Ch 66). Instead of reading and writing storage blocks, the device exposes **I/O functions**, a set of registers and an interrupt line. The WiFi chip is "SDIO function 1 (and 2)". The host reads/writes its registers and data FIFOs over SDIO commands (CMD52 single-byte, CMD53 block).

```
i.MX6ULL uSDHC2 ──[SDIO 4-bit]──► AP6212
    CLK, CMD, DAT0-3                WL_SDIO
    + WL_REG_ON (GPIO reset)        WL_REG_ON
    + WL_HOST_WAKE (IRQ)            out-of-band IRQ
    + 32.768 kHz LPO clock          32K_CLK
    + 3.3V + 1.8V rails
```

Key signals beyond the 6-wire SDIO:

- **WL_REG_ON**: a GPIO that powers/resets the WiFi block. Must be pulsed to wake the chip.

> **MCU bridge:** Think of Linux GPIO like the same pin set/reset block you used on STM32, but accessed through a kernel subsystem that owns numbering, direction, interrupts, and user-space exposure. **GPIO:** General-Purpose Input/Output, a pin controlled as a digital input, output, or interrupt source.

- **32.768 kHz LPO clock**: the low-power oscillator the chip needs for its sleep timing. Without it, the chip is unreliable or won't init.
- **WL_HOST_WAKE**: an out-of-band interrupt (the chip can wake the host even when SDIO is idle).

91.3 The full WiFi stack (recap from Ch 55E)

```
user-space: wpa_supplicant, NetworkManager
     │ nl80211
```

```
        ▼
   cfg80211 (config API)
        │
        ▼
   ┌──────────────────────────────────────────────┐
   │ brcmfmac (the chip driver)                   │
   │  - loads firmware + NVRAM over SDIO          │
   │  - implements cfg80211 ops (scan, connect)   │
   │  - shuttles data frames chip ↔ network stack │
   └──────────────────────────────────────────────┘
        │ SDIO (CMD52/CMD53)
        ▼
   AP6212 firmware (runs the actual 802.11 MAC)
```

For a **full-MAC** chip like the AP6212, mac80211 is *not* used, the chip's firmware is the MAC. Brcmfmac talks cfg80211 directly. (Soft-MAC chips like some Atheros parts use mac80211. Full-MAC chips like Broadcom/Marvell don't.)

91.4 Device-tree bring-up

This is where boards fail. The DT must describe: the SDIO bus, the power sequence, the chip node, and the 32 kHz clock.

```
/* The 32.768 kHz clock the WiFi needs */
wifi_lpo_clk: wifi-lpo-clk {
    compatible = "fixed-clock";
    #clock-cells = <0>;
    clock-frequency = <32768>;
};

/* Power sequence: pulse WL_REG_ON, provide the LPO clock */
wifi_pwrseq: wifi-pwrseq {
    compatible = "mmc-pwrseq-simple";
    reset-gpios = <&gpio5 9 GPIO_ACTIVE_LOW>;     /* WL_REG_ON */
    clocks = <&wifi_lpo_clk>;
    clock-names = "ext_clock";
    post-power-on-delay-ms = <100>;
};

&usdhc2 {
    pinctrl-names = "default";
    pinctrl-0 = <&pinctrl_usdhc2>;
    bus-width = <4>;
    non-removable;                /* soldered chip, not a removable card */
    keep-power-in-suspend;
    cap-power-off-card;
    mmc-pwrseq = <&wifi_pwrseq>;
    vmmc-supply = <&reg_wlan_3v3>;
    vqmmc-supply = <&reg_wlan_1v8>;
    no-1-8-v;                     /* or allow it, depending on module */
    status = "okay";

    #address-cells = <1>;
    #size-cells = <0>;
```

```
    wifi@1 {
        compatible = "brcm,bcm4329-fmac";    /* covers BCM43438/AP6212 */
        reg = <1>;                            /* SDIO function 1 */
        interrupt-parent = <&gpio5>;
        interrupts = <8 IRQ_TYPE_LEVEL_HIGH>; /* WL_HOST_WAKE */
        interrupt-names = "host-wake";
    };
};
```

The five things that must all be right:

1. **SDIO bus config** (`bus-width`, `non-removable`, no card-detect).
2. **Power sequence** (`mmc-pwrseq-simple` pulsing WL_REG_ON).
3. **The 32 kHz clock** supplied to the pwrseq.
4. **The chip node** with the right `compatible` and SDIO function `reg = <1>`.
5. **Voltage rails** (`vmmc`, `vqmmc`).

Miss any one of these and dmesg is silent.

91.5 Firmware + NVRAM

brcmfmac loads two files from `/lib/firmware/brcm/`:

```
brcmfmac43430-sdio.bin      ← the firmware blob (the 802.11 MAC code)
brcmfmac43430-sdio.txt      ← the per-board NVRAM (calibration + config)
```

The exact filename depends on the chip ID (read over SDIO at probe). For BCM43438: `brcmfmac43430-sdio.*`.

The firmware blob is the chip's program, get it from `linux-firmware.git`. It's chip-specific but board-independent.

The NVRAM (`.txt`) is **per-board**: antenna configuration, regulatory domain, board-specific RF calibration, crystal frequency trim. It comes from the *module vendor* (AmPak for AP6212) or the board vendor. Using the wrong NVRAM gives: terrible range, wrong country code, or a chip that enumerates but won't connect.

```
[root@pa-mini:~]# ls /lib/firmware/brcm/
brcmfmac43430-sdio.bin
brcmfmac43430-sdio.txt        ← you must supply this for your board
brcmfmac43430-sdio.clm_blob   ← optional regulatory blob
```

The `.clm_blob` (Country Locale Matrix) holds regulatory limits. Newer firmware needs it.

91.6 Bring-up trace

A successful bring-up in dmesg:

```
[root@pa-mini:~]# dmesg | grep -iE 'mmc|brcm|wifi'
mmc1: SDHCI controller on 2194000.usdhc [2194000.usdhc] using ADMA
mmc1: queuing unknown CIS tuple 0x80 (2 bytes)        ← SDIO enumeration
mmc1: new high speed SDIO card at address 0001        ← chip responded!
brcmfmac: brcmf_fw_alloc_request: using brcm/brcmfmac43430-sdio for chip BCM43430/1
brcmfmac: brcmf_fw_complete_request: firmware brcm/brcmfmac43430-sdio.bin found
brcmfmac: brcmf_fw_complete_request: nvram brcm/brcmfmac43430-sdio.txt found
brcmfmac: brcmf_c_preinit_dcmds: Firmware: BCM43430/1 wl0: ...
brcmfmac_sdio mmc1:0001:1 wlan0: renamed from wlan0
```

The progression:

1. `mmc1: new high speed SDIO card`, **the SDIO transport works**, the chip responded to enumeration. If this line is missing, the problem is the SDIO bus / power sequence / 32 kHz clock, *not* WiFi.
2. `brcmfmac: ... firmware ... found`, the firmware blob loaded.
3. `nvram ... found`, the NVRAM loaded.
4. `Firmware: BCM43430/1`, the chip booted its firmware and reported its version.
5. `wlan0`, the network interface appeared.

Debugging by stage: if you don't see line 1, it's a transport problem (most common). If line 1 appears but not line 2, the firmware file is missing/misnamed. If lines 1–4 appear but WiFi behaves badly, it's the NVRAM.

Then scan + connect (Ch 55E):

```
[root@pa-mini:~]# ip link set wlan0 up
[root@pa-mini:~]# iw wlan0 scan | grep SSID
[root@pa-mini:~]# wpa_supplicant -B -i wlan0 -c /etc/wpa_supplicant.conf -D nl80211
[root@pa-mini:~]# udhcpc -i wlan0
[root@pa-mini:~]# ping 8.8.8.8
```

91.7 How a packet flows (since you won't write this driver)

Full-MAC WiFi drivers are tens of thousands of lines, you won't reimplement brcmfmac. But understanding the data path demystifies it:

Transmit (your app sends a packet)

```
1. App: send(socket, data)
2. Kernel network stack builds an skb (socket buffer), routes it to wlan0.
```

```
3. brcmfmac's ndo_start_xmit(skb, wlan0) is called.
4. brcmfmac wraps the skb with a BDC/BCMC header (Broadcom's framing).
5. It writes the framed packet to the chip's SDIO data FIFO via CMD53 block write.
6. The AP6212 firmware takes over: 802.11 framing, encryption (CCMP),
   rate selection, retransmission, the actual radio transmission.
```

The Linux side stops at “write bytes to SDIO FIFO.” Everything 802.11 happens in the chip’s firmware.

Receive

```
1. AP6212 firmware receives an 802.11 frame, decrypts it, de-frames to Ethernet.
2. It asserts the SDIO interrupt (or WL_HOST_WAKE).
3. brcmfmac's SDIO IRQ handler reads the chip's status, then reads the
   pending data from the SDIO FIFO via CMD53 block read.
4. It strips the BDC header, builds an skb.
5. netif_rx(skb) hands it to the network stack → up to the app.
```

Control (scan, connect)

```
1. wpa_supplicant: nl80211 "scan request" → cfg80211 → brcmfmac's .scan op.
2. brcmfmac sends a "WLC_SCAN" command (Broadcom's ioctl-over-SDIO protocol)
   to the chip's control FIFO.
3. The chip's firmware does the scan (off-channel sweeps), reports results back.
4. brcmfmac reads the results, converts to cfg80211 BSS entries, reports up.
```

So brcmfmac is three things: a firmware loader, an SDIO packet shuttle, and a cfg80211-to-Broadcom command translator. The source is ~15000 lines, but conceptually it is those three jobs.

91.8 In-tree vs out-of-tree, a strong recommendation

The AP6212 (brcmfmac) and SD8801 (mwifiex) have **in-tree** drivers, they ship with the kernel, build automatically, and track kernel API changes. The RTL8189FTV usually needs an **out-of-tree** driver (`rtl8189es` from a GitHub repo).

Out-of-tree WiFi drivers are a recurring maintenance nightmare:

- They break on every kernel upgrade (the kernel’s internal APIs change. Out-of-tree drivers must be patched to match).
- DKMS rebuilds them on kernel upgrade, but only if a matching version exists.
- Quality varies wildly. Many are vendor dumps with poor power management.

For a product, choose a module with an in-tree driver. Saving $0.50 on an RTL8189 versus an AP6212 is nothing compared to the engineering cost of maintaining an out-of-tree driver for an 8-year product life. (Note: `rtw88` is bringing more Realtek parts in-tree, check current status.)

91.9 Key Notes

- SDIO discovery is a transport milestone before Wi-Fi firmware, calibration and authentication can work. Each stage provides different diagnostic evidence.
- Module power sequencing, reference clocks and signaling voltage are board dependencies. Their description must match the fitted module and available controller.
- The host driver and radio firmware share a protocol and compatibility contract. Board-specific calibration is not interchangeable with an arbitrary firmware file.
- Kernel networking, authentication and IP configuration remain separate responsibilities even for a full-MAC module. Driver support also has a lifecycle cost across kernel updates.

91.10 Pitfalls

- **No “new SDIO card” in dmesg.** The transport failed. 90% of bring-up problems are here, not in WiFi. Check: WL_REG_ON pulsing (mmc-pwrseq), 32 kHz clock present, rails up, SDIO pinmux correct.
- **Missing 32 kHz clock.** The chip needs the LPO for sleep timing. Without it, it may enumerate but be flaky, or not enumerate. Provide it (a `fixed-clock` in DT + a physical 32 kHz source, often from the PMIC).

> **MCU bridge:** Think of a PMIC like a programmable power-tree supervisor: it replaces discrete enables and LDO assumptions with sequenced rails the kernel can model. **PMIC:** Power Management IC, a chip that sequences and regulates the board’s voltage rails.

- **Wrong/missing NVRAM.** Chip enumerates, firmware loads, but range is awful or it won’t associate. The NVRAM is per-board. Get the right one from the module vendor.
- **Wrong firmware filename.** brcmfmac derives the name from the chip ID. If your chip variant maps to a different filename, “firmware not found.” Check the exact name in the dmesg “using brcm/…” line.
- **`no-1-8-v` mismatch.** Some modules require 1.8 V SDIO signaling, others 3.3 V only. Mismatch = enumeration failure or corruption. Match `vqmmc` + the `no-1-8-v` flag to the module.
- **Card-detect on a soldered chip.** Forgetting `non-removable` makes the kernel poll for card removal. The chip may get powered off. Always `non-removable` for soldered WiFi.

- **regdb missing.** Without `wireless-regdb` + CRDA (or the in-kernel regdb), the chip is limited to the most restrictive channels. Install it.
- **Out-of-tree driver + kernel upgrade.** The RTL8189 out-of-tree driver breaks. Pin the kernel or use DKMS, or pick an in-tree module.
- **BT half of a combo not coming up.** The AP6212 is WiFi+BT. The BT half (Ch 94) needs separate UART bring-up. WiFi working doesn't mean BT works.

91.11 Going deeper

- **`drivers/net/wireless/broadcom/brcm80211/brcmfmac/`**: the brcmfmac driver. `sdio.c` is the SDIO transport. `core.c` the cfg80211 glue.
- **`drivers/net/wireless/marvell/mwifiex/`**: Marvell SD8801.
- **`drivers/mmc/core/sdio.c`**: the SDIO bus enumeration.
- **`Documentation/devicetree/bindings/net/wireless/brcm,bcm4329-fmac.yaml`**: the DT binding.
- **`Documentation/devicetree/bindings/mmc/mmc-pwrseq-simple.yaml`**: power sequencer.
- **`linux-firmware.git/brcm/`**: the firmware blobs.
- **AmPak AP6212 datasheet + reference schematic**: the canonical wiring + NVRAM.
- **`Documentation/networking/regulatory.rst`**: the WiFi regulatory framework.

> Next chapter: **Chapter 92: USB WiFi.** The dongle approach, RTL8188EUS, MT7601, and the in-tree-vs-out-of-tree driver saga that defines the USB-WiFi experience.

Chapter 92: USB WiFi

> **Driver choice:** Use the in-tree, maintained driver first. Use out-of-tree, spidev, or custom-driver paths only after you accept the kernel-version maintenance cost and document who owns updates.

> **What:** USB WiFi dongles, the plug-in alternative to soldered SDIO WiFi (Ch 91). Three chips compared: **Realtek RTL8188EUS** (the ubiquitous cheap dongle), **MediaTek MT7601** (common in $3 dongles), **Ralink RT5370** (older, very mainline-friendly). The big theme here is in-tree versus out-of-tree drivers. Some dongles just work. Others need a constantly-rebuilt DKMS module. Plus bandwidth contention with other USB devices on the i.MX6ULL's USB-2.0 bus.
>
> **Why:** USB WiFi is the *fastest* way to add WiFi to a board that has a spare USB port, no SDIO bring-up, no NVRAM, no 32 kHz clock. The catch is driver support: some chips have excellent in-tree drivers, others need out-of-tree modules that break on every kernel upgrade. Chip choice is most of the work.
>
> **Focus:** the chip you buy determines whether bringing up WiFi takes five minutes or five days. A `rtw88-` or `rt2800usb`-supported chip is plug-and-play. An RTL8188EUS needs the out-of-tree `rtl8188eus` driver, which you must rebuild for every kernel. This chapter is a buyer's guide as much as a driver guide.
>
> **Tooling.** This chapter uses `wpa_supplicant`, `iw`, chip firmware blob (Realtek/MediaTek/Ralink).
>
> - **Ubuntu-base (target):** `apt install wpasupplicant iw firmware-realtek firmware-misc-nonfree`
> - **Buildroot:** `BR2_PACKAGE_WPA_SUPPLICANT=y BR2_PACKAGE_IW=y` **Buildroot:** a configuration-driven build system that produces a complete root filesystem and related images.
> - Full per-tool reference: Userspace tooling appendix. **MCU bridge:** Think of the rootfs as the firmware image's file-backed runtime environment. On an MCU you link everything into flash. On Linux, programs and config live in this mounted tree. **rootfs:** root filesystem, the directory tree mounted at / that contains /bin, /etc, /dev, and libraries.

92.1 Chip comparison & driver status

	RTL8188EUS	MT7601	RT5370
Vendor	Realtek	MediaTek	Ralink (now MediaTek)
Standard	802.11n (2.4 GHz, 1×1)	802.11n (2.4 GHz)	802.11n (2.4 GHz)
Throughput	~50 Mbps	~50 Mbps	~40 Mbps
In-tree driver?	partial (`r8188eu` since 5.18, in staging)	yes (`mt7601u`)	**yes (`rt2800usb`)** ✓
Out-of-tree	`rtl8188eus` (aircrack-ng repo)	usually not needed	not needed
AP mode / monitor	yes (out-of-tree)	limited	yes
Volume price	$2–4	$3–5	$3–5

The crucial column is “in-tree driver?”:

- **RT5370** (`rt2800usb`): the gold standard. Mainline since forever, rock-solid, supports AP + monitor mode. For the lowest-hassle path, buy an RT5370 dongle.
- **MT7601** (`mt7601u`): in-tree, works well. Good second choice.
- **RTL8188EUS**: the `r8188eu` staging driver works for basic STA mode (recent kernels), but AP/monitor mode + best performance needs the out-of-tree `rtl8188eus`. The most common dongle, but the most painful driver.

92.2 The plug-and-play case (RT5370)

```
[root@pa-mini:~]# lsusb
Bus 001 Device 004: ID 148f:5370 Ralink Technology RT5370 Wireless Adapter

[root@pa-mini:~]# dmesg | grep -iE 'rt2800|usb'
usb 1-1.3: new high-speed USB device number 4
usb 1-1.3: RT chipset 5390, rev 0502 detected
ieee80211 phy0: rt2x00_set_rt: Info - RT chipset 5390, rev 0502 detected
ieee80211 phy0: Selected rate control algorithm 'minstrel_ht'
usbcore: registered new interface driver rt2800usb
```

```
[root@pa-mini:~]# ip link
3: wlan0: <BROADCAST,MULTICAST> ...
```

Plug in → `wlan0` appears. Firmware (`rt2870.bin`) is loaded automatically from `/lib/firmware/` (ships with `linux-firmware`). Then scan/connect exactly as Ch 55E:

```
[root@pa-mini:~]# ip link set wlan0 up
[root@pa-mini:~]# wpa_supplicant -B -i wlan0 -c /etc/wpa_supplicant.conf -D nl80211
[root@pa-mini:~]# udhcpc -i wlan0
```

Total bring-up is three steps. Insert the dongle. Copy the firmware if it is not present. Connect. About five minutes.

RT5370 and MT7601 are *soft-MAC* chips. They use mac80211: the kernel does the 802.11 MAC, the chip is just a radio. This is why integration is clean. Mac80211 and cfg80211 handle the protocol work, and the chip driver is a thin USB-to-radio shim.

> **MAC:** Media Access Control in networking and radio chapters. It is the layer that owns framing and medium access.

92.3 The painful case (RTL8188EUS)

Most cheap “USB WiFi” dongles sold as “for Raspberry Pi” are RTL8188EUS. They’re full-MAC-ish (Realtek’s own MAC in a half-baked driver). Two driver options:

Option A: in-tree `r8188eu` (staging)

Since kernel 5.18, there’s a staging driver `drivers/staging/r8188eu/`. Enable `CONFIG_R8188EU=m`. It handles **station mode** (connect to an AP) adequately. For a product that just needs to join a WiFi network, this is enough and it’s *in-tree* (builds with the kernel, tracks API changes).

```
[root@pa-mini:~]# dmesg | grep 8188
r8188eu 1-1.3:1.0: Firmware Version 11, SubVersion 1, Signature 0x88e0
r8188eu 1-1.3:1.0 wlan0: renamed from wlan0
```

Option B: out-of-tree `rtl8188eus`

The aircrack-ng `rtl8188eus` driver supports AP mode, monitor mode, packet injection, and better performance. But it’s **out-of-tree**:

```
# On the build host or target:
git clone https://github.com/aircrack-ng/rtl8188eus
cd rtl8188eus
make ARCH=arm CROSS_COMPILE=arm-none-linux-gnueabihf- KSRC=/path/to/kernel
# Install the resulting 8188eu.ko
```

The pain:

- **Breaks on kernel upgrade.** The driver uses internal kernel APIs that change between versions. Each kernel bump may require patching the driver.
- **DKMS** automates the rebuild but only if a compatible driver version exists for the new kernel.
- **Quality**: power management is poor (the dongle runs hot, drains battery), and it occasionally wedges requiring a re-plug.

The recommendation

For a *product*: **don't use RTL8188EUS** unless you need its AP/monitor features. Use RT5370 or MT7601 (in-tree, clean). The RTL8188EUS is fine for a hobby project or a dev board where you'll manage the driver manually, but for an 8-year-life product, the out-of-tree maintenance cost is real.

92.4 How USB WiFi differs from SDIO WiFi

	SDIO WiFi (Ch 91)	USB WiFi
Bring-up effort	high (transport, pwrseq, NVRAM, clock)	low (plug in)
Per-board config	NVRAM file required	none (firmware is generic)
Bus	dedicated SDIO	shared USB
Bandwidth contention	none	competes with other USB devices
Power	soldered, low	dongle, often higher (no PM tuning out-of-tree)
Antenna	on-board (you design it)	on the dongle (vendor's)
Form factor	soldered chip	external dongle (or soldered USB module)

USB WiFi trades bring-up ease for bus contention and (often) worse power management. For a quick bring-up or a field-swappable WiFi, it’s great. For a polished low-power product, soldered SDIO with an in-tree driver is better.

92.5 USB bandwidth contention

The i.MX6ULL has USB-2.0 (480 Mbps theoretical, ~320 practical). A USB WiFi dongle + another bandwidth-heavy USB device (a UVC camera from Ch 88, USB Ethernet) on the *same* USB controller compete:

```
[root@pa-mini:~]# # Camera + WiFi on the same hub:
[root@pa-mini:~]# # iperf3 over WiFi while streaming the camera → both degrade
```

Mitigation: the i.MX6ULL has **two USB OTG controllers**. Put the camera on one, WiFi on the other. Each gets its own 480 Mbps.

```
[root@pa-mini:~]# lsusb -t      # shows the bus topology
/:  Bus 01.Port 1: Dev 1, Class=root_hub, Driver=ci_hdrc/1p, 480M
    |__ Port 1: Dev 4, ... rt2800usb         ← WiFi on controller 1
/:  Bus 02.Port 1: Dev 1, Class=root_hub, Driver=ci_hdrc/1p, 480M
    |__ Port 1: Dev 2, ... uvcvideo          ← camera on controller 2
```

92.6 AP mode (hostapd)

USB dongles with mac80211 (RT5370, MT7601) and AP-capable firmware can be an *access point*, not just a client:

```
# /etc/hostapd.conf
interface=wlan0
driver=nl80211
ssid=MyEmbeddedAP
hw_mode=g
channel=6
wpa=2
wpa_passphrase=secretpassword
wpa_key_mgmt=WPA-PSK
rsn_pairwise=CCMP

[root@pa-mini:~]# hostapd /etc/hostapd.conf &
[root@pa-mini:~]# # Now devices can connect to "MyEmbeddedAP"
# Add a DHCP server (dnsmasq) + IP forwarding for a full hotspot.
```

This turns your i.MX6ULL into a WiFi hotspot, useful for “configure-the-device-from-your-phone” setup flows. RT5370 and MT7601 support this in-tree. RTL8188EUS needs the out-of-tree driver for AP mode.

92.7 Key Notes

- USB removes much board-specific bus wiring but does not remove firmware, driver or radio-policy dependencies.
- The actual USB identity and supported chip revision matter more than a dongle's retail name. Station, access-point and monitor capabilities must be established for the selected stack.
- Out-of-tree driver support creates an ongoing compatibility responsibility. A module that builds today does not establish support for the next kernel.
- USB power and shared-controller bandwidth can limit networking before radio conditions do. Throughput depends on the whole system and traffic pattern.

92.8 Pitfalls

- **Buying RTL8188EUS expecting it to "just work."** The most common dongle, the most painful driver. For a product, prefer RT5370/MT7601.
- **Out-of-tree driver + kernel upgrade.** Breaks. Pin the kernel, use DKMS with a compatible version, or, better, switch to an in-tree chip.
- **Missing firmware.** `rt2800usb` needs `rt2870.bin`. `mt7601u` needs `mt7601u.bin`. Copy from linux-firmware. Symptom: chip detected, no `wlan0`.
- **Bandwidth contention.** Camera + WiFi on one USB controller → both degrade. Spread across the two controllers.
- **Bus-power limits.** Some dongles draw 400+ mA on TX. A weak VBUS rail browns out → disconnects. Ensure adequate USB power.
- **Counterfeit chips.** A dongle sold as "RT5370" may contain an RTL8188 (or vice versa). Always `lsusb` to confirm the actual chip before committing a design.
- **Monitor/AP mode on the wrong chip.** Not all chips/drivers support AP or monitor mode. Check before designing a feature around it.
- **regdb / country code.** Same as SDIO, install `wireless-regdb` or you're limited to restrictive channels.
- **Soldered USB-WiFi modules.** Some "USB WiFi" is a soldered-on module, not a removable dongle. The driver situation is the same, but now you cannot swap the chip if the driver turns out to be bad. Choose the chip even more carefully.

92.9 Going deeper

- **`drivers/net/wireless/ralink/rt2x00/`**: the `rt2800usb` driver (RT5370). A clean mac80211 USB driver.

- **`drivers/net/wireless/mediatek/mt7601u/`**: MT7601.
- **`drivers/staging/r8188eu/`**: the in-tree RTL8188EU staging driver.
- **`drivers/net/wireless/realtek/rtw88/`**: newer Realtek parts coming in-tree.
- **`linux-firmware.git`**: `rt2870.bin`, `mt7601u.bin`.
- **`hostapd` documentation**: for AP mode.
- **aircrack-ng `rtl8188eus` repo**: the out-of-tree driver (and its README's kernel-compatibility notes).
- **`Documentation/networking/mac80211-injection.rst`**: monitor/injection mode.

> **Note on Group K so far:** SDIO WiFi (Ch 91) for soldered, low-power, in-tree (AP6212). USB WiFi (Ch 92) for quick/swappable but watch the driver story (prefer RT5370/MT7601 in-tree over RTL8188EUS out-of-tree). The next two chapters cover the hosted approach (Ch 93: WiFi via an ESP32 co-processor) and WiFi+BT combo modules (Ch 94).

> Next chapter: **Chapter 93: Hosted WiFi via ESP32 / ESP8266.** When the SoC has no SDIO and no spare USB, offload WiFi to an ESP co-processor over UART or SPI.

Chapter 93: Hosted WiFi via ESP32 / ESP8266

What: offloading WiFi to an **ESP32** (or ESP8266) co-processor connected over SPI or UART. Two architectures: **esp-hosted** (the ESP runs firmware that makes Linux see a normal `wlan0`, full network-stack integration), and **AT-command mode** (the ESP runs the TCP/IP stack itself. Linux talks to it like a modem). For each: the architecture, the bring-up, and how the host-side driver shuttles data.

Why: sometimes the SoC has no SDIO and no spare USB (or you want to add WiFi to an existing design without changing the SoC). An ESP32 is a $2 WiFi+BT co-processor you connect over a couple of GPIOs. It's also the answer for *MCU + Linux co-existence* designs, and for adding WiFi to legacy SoCs that predate good WiFi support. The trade-off: lower throughput than SDIO/USB, and you now have *two* firmwares to manage.

Focus: two very different offload models. Esp-hosted treats the ESP as a *dumb radio*. Linux runs the IP stack and just sends 802.11 frames through the ESP. From Linux's view there is a normal `wlan0` with wpa_supplicant on top. AT-command treats the ESP as a *smart modem*. The ESP runs its own TCP/IP stack and Linux speaks `AT+CIPSTART` to open sockets. Simple, but limited and non-standard. The choice between them shapes the rest of the design.

Tooling. This chapter uses `wpa_supplicant`, `iw`, the `esp-hosted` kernel module + firmware on the ESP.

- **Ubuntu-base (target):** `apt install wpasupplicant iw`
- **Buildroot:** `BR2_PACKAGE_WPA_SUPPLICANT=y BR2_PACKAGE_IW=y` **Buildroot:** a configuration-driven build system that produces a complete root filesystem and related images.
- Full per-tool reference: Userspace tooling appendix. **MCU bridge:** Think of the rootfs as the firmware image's file-backed runtime environment. On an MCU you link everything into flash. On Linux, programs and config live in this mounted tree. **rootfs:** root filesystem, the directory tree mounted at / that contains /bin, /etc, /dev, and libraries.

93.1 Why hosted WiFi

Scenario	Why hosted WiFi
SoC has no SDIO and no free USB	ESP over SPI/UART uses just GPIOs
Adding WiFi to a legacy design	no SoC change; bolt on an ESP module
MCU + Linux hybrid product	the ESP can also do real-time tasks
Want WiFi + BT in one cheap part	ESP32 has both
Strict EMC / certification	the ESP module is pre-certified (FCC/CE), saves you RF certification

One often-missed advantage: an ESP32 *module* (not the bare chip) ships with FCC/CE/IC modular certification. If you mount the module without changing the antenna, your product inherits the certification. You skip an expensive antenna-certification step. For low-volume products, this alone can justify hosted WiFi.

The cost: throughput tops out around 20 Mbps (SPI) or much less (UART), vs 30–50 Mbps for SDIO/USB. You also maintain the ESP firmware on top of the Linux side.

93.2 The two architectures

esp-hosted: ESP as a dumb radio

```
Linux: wpa_supplicant, full IP stack, wlan0
     │ nl80211 / netdev
     ▼
esp32_sdio / esp32_spi driver (Espressif's Linux driver)
     │ SPI or SDIO transport (a custom framed protocol)
     ▼
ESP32 running esp-hosted firmware
     │  (just relays 802.11 frames + control)
     ▼
radio
```

Linux sees a normal `wlan0`. Wpa_supplicant runs on Linux. The ESP is a transport, it moves frames and relays cfg80211 commands. This is the “proper” integration: standard tools, standard behavior, the ESP’s own IP stack is *not* used.

esp-hosted also relays Bluetooth, sending HCI over the same transport. So one ESP32 gives Linux both `wlan0` and `hci0`.

AT-command: ESP as a smart modem

```
Linux: your app sends AT commands over a UART
     │ /dev/ttymxc1 (raw UART)
     ▼
ESP8266/ESP32 running AT firmware
     │  (runs its OWN TCP/IP stack)
     ▼
radio
```

Linux talks to the ESP like a dial-up modem:

```
AT+CWJAP="MySSID","password"      ← join an AP
AT+CIPSTART="TCP","192.168.1.10",80  ← open a TCP socket
AT+CIPSEND=18                      ← send 18 bytes
GET / HTTP/1.0\r\n\r\n
```

The ESP runs the entire network stack. Linux just orchestrates via AT strings. Simple, but: no standard `wlan0`, no wpa_supplicant, no Linux sockets, your app speaks the AT dialect. Limited to a handful of simultaneous connections. Non-standard.

Choosing

- **esp-hosted**: when you want WiFi to behave like real WiFi, standard tools, multiple sockets, Linux's IP stack, TLS via OpenSSL, etc. It is the right choice for production. You need both the esp-hosted firmware on the ESP and the matching Linux driver.
- **AT-command**: when you want dead-simple, a few connections, and don't mind a non-standard interface. AT-command mode is common in quick prototypes and in MCU-style code. Avoid it for any product that needs Linux's network ecosystem (sockets, TLS, multiple connections, NetworkManager).

93.3 esp-hosted bring-up (SPI transport)

> **Driver choice:** Use the in-tree, maintained driver first. Use out-of-tree, spidev, or custom-driver paths only after you accept the kernel-version maintenance cost and document who owns updates.

Espressif's esp-hosted has two parts: firmware flashed to the ESP32, and a Linux driver (out-of-tree, from Espressif's `esp-hosted` repo).

Wiring (SPI mode):

```
   i.MX6ULL eCSPI3        ESP32
   ———————————————        ———————
   MOSI ————————————►  GPIO13 (HSPI MOSI)
   MISO ◄————————————  GPIO12
   SCLK ————————————►  GPIO14
   CS   ————————————►  GPIO15
   GPIO (handshake) ◄——  GPIO2   (ESP signals "data ready")
   GPIO (data-ready) ◄—  GPIO4
   reset GPIO ——————►  EN (reset the ESP)
```

The two extra GPIOs (handshake + data-ready) are the ESP's way of telling the host "I have data" without the host polling, essential for the SPI transport's flow control.

DT:

```
&ecspi3 {
    esp32@0 {
        compatible = "espressif,esp32-spi";
        reg = <0>;
        spi-max-frequency = <10000000>;
        reset-gpios = <&gpio4 10 GPIO_ACTIVE_LOW>;
        data-ready-gpios = <&gpio4 11 GPIO_ACTIVE_HIGH>;
        handshake-gpios = <&gpio4 12 GPIO_ACTIVE_HIGH>;
    };
};
```

Build + load Espressif's driver (out-of-tree):

```
[root@pa-mini:~]# insmod esp32_spi.ko
[root@pa-mini:~]# dmesg | grep -i esp
esp32: Resetting ESP32
esp32: ESP32 chip detected, firmware version 1.0.3
esp32: Features: WLAN + BT/BLE
esp32_spi: Network interface 'espsta0' created
esp32_spi: Bluetooth HCI interface created

[root@pa-mini:~]# ip link
3: espsta0: <BROADCAST,MULTICAST> ...     ← the WiFi interface
```

Now `espsta0` behaves like any `wlan0`, wpa_supplicant, DHCP, etc.:

```
[root@pa-mini:~]# wpa_supplicant -B -i espsta0 -c /etc/wpa_supplicant.conf -D nl80211
[root@pa-mini:~]# udhcpc -i espsta0
[root@pa-mini:~]# ping 8.8.8.8
```

Throughput: ~10–20 Mbps over SPI at 10 MHz. Adequate for IoT telemetry, MQTT, OTA updates. Not for video streaming.

93.4 How the esp-hosted driver works

The driver is a **netdev** (network device) that wraps the SPI transport:

```
/* Conceptual — esp-hosted SPI driver structure */

/* Transmit: kernel hands us an skb, we frame + SPI it to the ESP */
static netdev_tx_t esp_xmit(struct sk_buff *skb, struct net_device *ndev)
{
    struct esp_priv *priv = netdev_priv(ndev);
    /* Prepend the esp-hosted protocol header (interface type, length) */
    struct esp_payload_header *hdr = skb_push(skb, sizeof(*hdr));
    hdr->if_type = ESP_STA_IF;
    hdr->len = skb->len;
    /* Queue for the SPI worker to send on the next transaction */
    skb_queue_tail(&priv->tx_q, skb);
    schedule_work(&priv->tx_work);
    return NETDEV_TX_OK;
}

/* The SPI worker: full-duplex transaction exchanges TX and RX simultaneously */
static void esp_spi_work(struct work_struct *w)
{
    /* Wait for the ESP's data-ready/handshake GPIO */
    /* Do a full-duplex SPI transfer: send a queued TX packet,
       receive whatever the ESP has pending */
    spi_sync(priv->spi, &msg);
    /* Parse the received buffer's esp header; if it's a data frame,
       build an skb and netif_rx() it up the stack */
}

static const struct net_device_ops esp_netdev_ops = {
    .ndo_start_xmit = esp_xmit,
    .ndo_open  = esp_open,
    .ndo_stop  = esp_stop,
};
```

The ESP's transport protocol multiplexes several streams over the same SPI link: WiFi-STA frames, WiFi-AP frames, BT-HCI packets, and control commands. An `if_type` field in the header distinguishes them. The driver demuxes received packets to the right interface (`espsta0`, `hci0`, etc.).

Control (scan, connect) goes through a separate control path, the driver sends "control request" packets (a protobuf-encoded command) and the ESP firmware executes them, mimicking cfg80211 ops. So `wpa_supplicant`'s nl80211 scan request → cfg80211 → the esp driver's `.scan` → a control packet to the ESP → the ESP scans → results come back → reported up. The structure mirrors brcmfmac (Ch 91). Only the transport changes, the esp-hosted SPI protocol instead of SDIO.

93.5 AT-command mode

For the simpler AT approach, the ESP runs Espressif's AT firmware. Linux talks raw UART:

```
/* User-space, talking to the ESP over /dev/ttymxc1 */
int fd = open("/dev/ttymxc1", O_RDWR);
/* set 115200 8N1 via termios */
```

```
write(fd, "AT+CWMODE=1\r\n", 13);          /* station mode */
expect_ok(fd);
write(fd, "AT+CWJAP=\"MySSID\",\"pass\"\r\n", ...);  /* join AP */
expect("WIFI CONNECTED", fd);
write(fd, "AT+CIPSTART=\"TCP\",\"192.168.1.10\",80\r\n", ...);
expect("CONNECT", fd);
write(fd, "AT+CIPSEND=18\r\n", ...);
expect(">", fd);
write(fd, "GET / HTTP/1.0\r\n\r\n", 18);
/* read +IPD response with the HTTP reply */
```

This is *user-space* code talking to a UART, no kernel driver at all (just the standard UART tty). It's the same pattern as a cellular AT modem (Ch 103). Simple, but: parsing AT responses is fiddly, only ~5 simultaneous connections, no TLS unless the firmware supports it, no integration with Linux sockets.

If you want AT-mode to look like a network interface, the kernel's PPP driver (`drivers/net/ppp/`) plus a chat script can layer PPP over the AT link. Esp-hosted is still the better choice for a real `wlan0`, though.

93.6 esp-hosted vs AT, decision table

	esp-hosted	**AT-command**
Linux sees	`wlan0` (standard)	a UART
Network stack	Linux's (full)	the ESP's (limited)
Tools	wpa_supplicant, ip, sockets	custom AT parser
TLS	OpenSSL on Linux	only if firmware supports
Simultaneous connections	unlimited (Linux sockets)	~5
Throughput	~10–20 Mbps (SPI)	~1–5 Mbps (UART)
Complexity	out-of-tree driver + firmware	just a UART + your AT parser
Bluetooth	yes (hci0 too)	ESP-dependent

	esp-hosted	AT-command
Best for	real products	quick prototypes, MCU-style

93.7 Key Notes

- Hosted networking and AT-command networking put the IP-stack boundary in different places. That choice determines whether Linux sees a normal network interface or an application-specific socket service.
- The co-processor firmware and Linux transport driver form a versioned pair. Updating only one side can break the shared protocol.
- Handshake signals, framing and flow control govern transport reliability. Increasing bus frequency alone does not establish stable throughput.
- Adding a co-processor adds a reset, health and update lifecycle. Real-time work on that processor also competes with its own radio and firmware responsibilities.

93.8 Pitfalls

- **Firmware version mismatch.** The esp-hosted Linux driver and the ESP firmware must be compatible versions. A mismatch → the driver loads but no interface, or garbled transport. Pin both versions.
- **Missing handshake/data-ready GPIOs.** The SPI transport needs them for flow control. Without them, the host polls blindly and loses packets. Wire and declare both.
- **Out-of-tree driver maintenance.** esp-hosted's Linux driver is out-of-tree (Espressif's repo), same kernel-upgrade fragility as Ch 92's RTL8188EUS. Plan for it.
- **AT response parsing fragility.** AT firmware responses vary across firmware versions ("OK" vs "SEND OK" vs "+CIPSEND:"). A brittle parser breaks on a firmware update. Parse defensively.
- **UART throughput ceiling.** AT over 115200 baud = ~11 KB/s raw, less after framing. For anything but tiny telemetry, use SPI esp-hosted or a higher UART baud.
- **Two firmwares to manage.** You now own the ESP firmware *and* the Linux side. OTA must update both, in a safe order.
- **Shared SPI bus contention.** If the ESP shares an SPI bus with other devices, the esp-hosted transport's latency suffers. Give it a dedicated bus or chip-select with priority.

- **ESP brownout.** The ESP32 draws current spikes (~500 mA) on TX. A weak rail → brownout resets → WiFi drops. Decouple well.

93.9 Going deeper

- **Espressif `esp-hosted` repo** (github.com/espressif/esp-hosted), firmware + Linux driver + protocol docs.
- **`esp-hosted` protocol documentation**: the framed SPI/SDIO transport.
- **Espressif AT firmware documentation**: the AT command set.
- **`drivers/net/` netdev model**: how a network interface driver is structured (esp-hosted is a netdev).
- **`Documentation/networking/`**: Linux network device internals.
- **`drivers/bluetooth/`**: for the HCI side of esp-hosted (Ch 95).

> Next chapter: **Chapter 94: WiFi+BT combo modules.** One chip, two radios, one antenna, the AP6212 and RTL8723, the shared-antenna coexistence problem, and bringing up WiFi (SDIO) + Bluetooth (UART) simultaneously.

Chapter 94: WiFi+BT combo modules

What: modules that combine WiFi and Bluetooth on one chip, sharing one 2.4 GHz antenna: **AP6212** (Broadcom BCM43438, WiFi over SDIO + BT over UART), **RTL8723BS** (Realtek, WiFi over SDIO + BT over UART). Two challenges define the topic. First, you must bring up two radios on one chip over two different buses. Second, the coexistence problem: both radios share the same 2.4 GHz band and the same antenna.

Why: most embedded products that want WiFi *also* want Bluetooth (BLE for phone-app provisioning, classic BT for audio). A combo module is cheaper and smaller than two separate chips, and it solves the coexistence problem in hardware (the chip arbitrates between its own radios). Bringing up both halves takes more than twice the effort of either alone, WiFi on SDIO (Ch 91), BT on UART, then making them coexist.

Focus: one chip carries two radios on two buses. They share one antenna and are managed by two independent kernel subsystems. The WiFi half is SDIO + brcmfmac (Ch 91). The BT half is UART + the Bluetooth HCI subsystem + `btattach`/`hciattach`. The two stacks are independent. They share silicon, but nothing else in software. The coexistence (PTA, Packet Traffic Arbitration) is internal to the chip but you must enable it and wire the BT_WAKE/WL_WAKE signals.

Tooling. This chapter uses `wpa_supplicant`, `iw`, `bluez` (`bluetoothctl`, `hciattach`), combo firmware.

- **Ubuntu-base (target):** `apt install wpasupplicant iw bluez bluez-tools`
- **Buildroot:** `BR2_PACKAGE_WPA_SUPPLICANT=y BR2_PACKAGE_IW=y BR2_PACKAGE_BLUEZ5_UTILS=y` **Buildroot:** a configuration-driven build system that produces a complete root filesystem and related images.
- Full per-tool reference: Userspace tooling appendix. **MCU bridge:** Think of the rootfs as the firmware image's file-backed runtime environment. On an MCU you link everything into flash. On Linux, programs and config live in this mounted tree. **rootfs:** root filesystem, the directory tree mounted at / that contains /bin, /etc, /dev, and libraries.

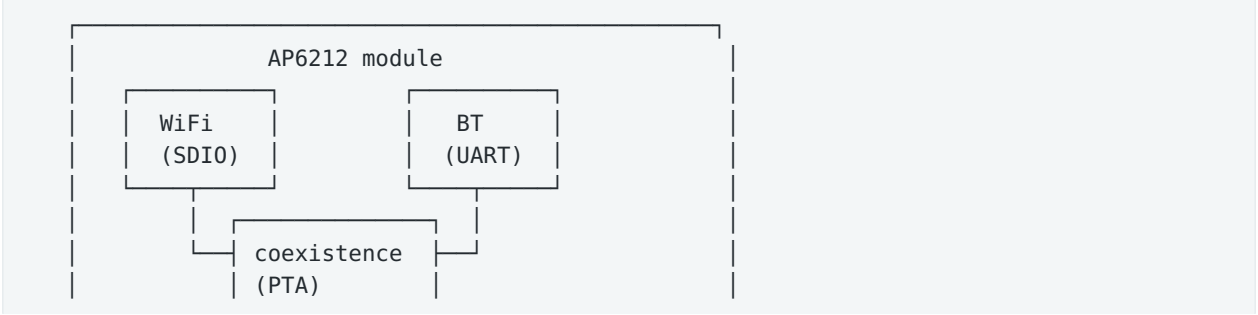
94.1 Module comparison

	AP6212 (BCM43438)	RTL8723BS	AP6256 (BCM43456)
WiFi	802.11 b/g/n 2.4 GHz	802.11 b/g/n 2.4 GHz	802.11 ac 2.4/5 GHz
Bluetooth	BT 4.1 + BLE	BT 3.0/4.0 + BLE	BT 5.0 + BLE
WiFi bus	SDIO	SDIO	SDIO
BT bus	UART (H4/H5)	UART	UART
WiFi driver	`brcmfmac` (in-tree)	`rtl8723bs` (in-tree, staging-graduated)	`brcmfmac`
BT driver	`hci_bcm` / `btbcm` (in-tree)	`hci_uart` + `btrtl` (in-tree)	`hci_bcm`
Coexistence	internal PTA	internal	internal
Volume price	$2–4	$2–3	$4–6

Pick guide:

- **AP6212**: best mainline support (brcmfmac + hci_bcm both in-tree). Default for i.MX boards.
- **RTL8723BS**: cheap, `rtl8723bs` is now in-tree. BT via btrtl. Decent.
- **AP6256**: when you need 5 GHz / 802.11ac / BT 5.0.

94.2 Two radios, two buses



```
   │            └──────┬──────┘                          │
   │                   │ shared 2.4 GHz radio + antenna  │
   └─────────┬─────────┴─────────────────────────────────┘
             │ SDIO          │ UART
             ▼               ▼
   i.MX6ULL uSDHC2    i.MX6ULL UART3
   → brcmfmac         → hci_bcm (Bluetooth HCI)
   → wlan0            → hci0
```

The two radios share the silicon and the antenna but connect to the SoC over *different* buses:

- **WiFi → SDIO** (uSDHC2) → `brcmfmac` → `wlan0`. (Exactly Ch 91.)
- **BT → UART** (UART3) → the Bluetooth HCI stack → `hci0`.

These are independent driver stacks. WiFi working doesn't mean BT works, and vice versa.

94.3 The WiFi half (recap of Ch 91)

Identical to Ch 91, SDIO bring-up, pwrseq, 32 kHz clock, firmware + NVRAM, `brcmfmac`, `wlan0`. Nothing new, the combo module's WiFi is exactly the AP6212 case from Ch 91.

94.4 The BT half, Bluetooth over UART

The BT radio connects via UART, speaking the **HCI** (Host Controller Interface) protocol, the standard host↔BT-controller protocol. The UART variant is "H4" (3-wire: TX/RX/no flow) or "H5"/"3-wire" (with software flow control) or H4 with hardware RTS/CTS.

Linux's Bluetooth stack:

```
   user-space: bluetoothctl, btmgmt, BlueZ daemon
        │
        ▼
   BlueZ (the Linux Bluetooth subsystem, net/bluetooth/)
        │ HCI commands/events
        ▼
   hci_uart line discipline + hci_bcm (vendor glue)
        │ UART (H4/H5)
        ▼
   AP6212 BT controller
```

DT for the BT half

Modern kernels use the **serdev** model (Ch 69's SerDev) to attach the BT controller to a UART:

```
&uart3 {
    pinctrl-names = "default";
```

```
    pinctrl-0 = <&pinctrl_uart3>;
    uart-has-rtscts;                /* hardware flow control — important for BT */
    status = "okay";

    bluetooth {
        compatible = "brcm,bcm43438-bt";
        max-speed = <3000000>;        /* 3 Mbps after baud-rate switch */
        shutdown-gpios = <&gpio5 10 GPIO_ACTIVE_HIGH>;   /* BT_REG_ON */
        device-wakeup-gpios = <&gpio5 11 GPIO_ACTIVE_HIGH>;  /* BT_WAKE */
        host-wakeup-gpios = <&gpio5 12 GPIO_ACTIVE_HIGH>;    /* HOST_WAKE */
        clocks = <&wifi_lpo_clk>;     /* same 32 kHz LPO as WiFi */
        clock-names = "lpo";
    };
};
```

The `hci_bcm` driver binds to this node (via the `brcm,bcm43438-bt` compatible), powers the BT block (`shutdown-gpios` = BT_REG_ON), loads the BT firmware patch, switches to high baud, and registers `hci0`.

BT firmware

Like WiFi, the BT controller needs a firmware patch (`.hcd` file for Broadcom):

```
/lib/firmware/brcm/BCM43430A1.hcd
```

`hci_bcm` loads this over HCI at init. Without it, BT works at reduced functionality (or not at all). Get it from the module vendor or linux-firmware.

Bring-up trace

```
[root@pa-mini:~]# dmesg | grep -i blue
Bluetooth: Core ver 2.22
Bluetooth: HCI device and connection manager initialized
Bluetooth: HCI UART driver
hci_uart_bcm serial1-0: supply vbat not found, using dummy regulator
Bluetooth: hci0: BCM: chip id 94
Bluetooth: hci0: BCM43430A1
Bluetooth: hci0: BCM43430A1 'brcm/BCM43430A1.hcd' Patch
Bluetooth: hci0: BCM43430A1 (...) build 0000

[root@pa-mini:~]# hciconfig
hci0:   Type: Primary  Bus: UART
        BD Address: 43:43:A1:12:34:56  ACL MTU: 1021:8  SCO MTU: 64:1
        UP RUNNING
```

`hci0` is up. Now use BlueZ:

```
[root@pa-mini:~]# bluetoothctl
[bluetooth]# power on
[bluetooth]# scan on            ← discovers nearby devices
[bluetooth]# pair AA:BB:CC:DD:EE:FF
[bluetooth]# connect AA:BB:CC:DD:EE:FF
```

For older / non-serdev setups, you attach manually:

```
[root@pa-mini:~]# btattach -B /dev/ttymxc2 -P bcm -S 3000000
# or the legacy:
[root@pa-mini:~]# hciattach /dev/ttymxc2 bcm43xx 3000000 flow
```

But the serdev DT approach (above) is preferred, the BT comes up automatically at boot.

94.5 Coexistence, the shared-antenna problem

WiFi and BT both use 2.4 GHz, and the combo module has *one* antenna. If both radios transmit simultaneously, they interfere. WiFi throughput collapses and BT audio stutters.

The chip solves this internally with **PTA** (Packet Traffic Arbitration, also called coexistence or “coex”). PTA is a hardware arbiter that time-slices the radio between WiFi and BT, prioritising by packet type. BT audio is latency-sensitive and gets high priority. WiFi bulk data can wait a few ms.

For the combo module (WiFi + BT on the *same* chip), PTA is internal, both radios are on one die, the arbiter is built in, and it “just works” once both halves are up. You don’t wire external coex signals.

For *separate* WiFi and BT chips (two dies), you’d wire 3-wire coex signals (BT_PRIORITY, BT_ACTIVE, WLAN_ACTIVE) between them, but combo modules avoid this by integrating.

What you *do* manage:

- **BT_WAKE / HOST_WAKE** (declared in DT), power-management handshakes letting BT wake the host and vice versa.
- Enabling coex in firmware (usually default-on for combo modules).
- Antenna design: one antenna, one matching network, shared by both radios. The module datasheet specifies the antenna requirements.

Observable effect: run iperf3 over WiFi while streaming BT audio. Without coex, both break. With coex (combo module), WiFi throughput drops modestly (the radio is time-shared) but both function. This drop is the cost of coexistence, typically 10 to 30 percent less WiFi throughput while BT is active.

94.6 Bringing up both, the order

1. **WiFi first** (Ch 91): SDIO + pwrseq + 32 kHz clock + firmware + NVRAM → `wlan0`. Verify it works alone.
2. **BT second**: UART + serdev BT node + BT firmware patch → `hci0`. Verify it works alone.

3. **Shared resources**: both halves share the 32 kHz LPO clock (declare it once, reference from both pwrseq and the BT node) and often share power rails. The WL_REG_ON and BT_REG_ON are usually *separate* GPIOs, power each half independently.
4. **Coexistence test**: run both simultaneously. Confirm acceptable performance.

A common mistake is to get WiFi working and assume the job is done. Then a field unit fails because the BT side was never wired correctly. Test both, separately and together.

94.7 Key Notes

- A combo module contains distinct Wi-Fi and Bluetooth paths, often using SDIO and UART respectively. Success of one interface does not prove the other works.
- Shared clocks, supplies and reset dependencies must have consistent ownership even when different kernel subsystems manage the two radios.
- Coexistence is a radio-resource problem, not merely two successful probes. Simultaneous traffic can change latency and throughput.
- Firmware, unique addresses and suspend/wakeup behavior are part of the finished module integration. A default identity or one boot-time connection is insufficient.

94.8 Pitfalls

- **WiFi works, BT forgotten.** The two halves are independent. Test both. A working `wlan0` says nothing about `hci0`.
- **Missing BT firmware patch.** `hci0` comes up but with reduced functionality or wrong BD address. Copy the `.hcd` (Broadcom) or firmware (Realtek) to `/lib/firmware/brcm/` (or `/rtl_bt/`).
- **UART without hardware flow control.** BT at 3 Mbps needs RTS/CTS. Without `uart-has-rtscts`, you get HCI packet corruption at high baud. Wire and enable flow control.
- **Shared 32 kHz clock declared twice / not at all.** Both halves need the LPO. Declare it once (a `fixed-clock`) and reference from both. Missing it → flaky or dead radios.
- **Baud-rate switch failure.** BT starts at 115200, then `hci_bcm` switches to 3 Mbps. If the UART or flow control can’t handle the switch, BT init fails after the firmware load. Check `dmesg` for the baud-switch step.
- **Default/duplicate BD address.** Many modules ship with `43:43:A1:00:00:00` or similar. Two devices with the same BD address can’t both be paired by one phone. Program a unique address at factory.

- **Coex not enabled.** On a *combo* module it's usually internal/automatic, but some firmwares need a coex-enable command. If WiFi + BT together perform terribly, check coex config.
- **Antenna shared but matched for one band.** The single antenna + matching network must work for both WiFi and BT (both 2.4 GHz, so usually fine). A poor match hurts both. Follow the module's reference antenna design.
- **BT_REG_ON vs WL_REG_ON confusion.** Separate GPIOs, separate power domains. Don't tie them together unless the module datasheet says to.

94.9 Going deeper

- **`drivers/bluetooth/hci_bcm.c`**: the Broadcom BT-over-UART driver (AP6212).
- **`drivers/bluetooth/btrtl.c`** + `hci_h5.c`, Realtek BT (RTL8723BS).
- **`drivers/bluetooth/hci_serdev.c`**: the serdev attachment glue.
- **`net/bluetooth/`**: the BlueZ kernel Bluetooth subsystem.
- **`drivers/net/wireless/broadcom/brcm80211/brcmfmac/`**: the WiFi half (Ch 91).
- **`Documentation/devicetree/bindings/net/broadcom-bluetooth.yaml`**: the BT DT binding.
- **AP6212 datasheet + reference schematic**: the canonical combo wiring, coex, antenna.
- **Bluetooth Core Specification (HCI section)**: the host-controller protocol.
- **`btattach` / `hciattach` man pages**: manual BT-UART attachment.

> **End of Group K, WiFi (Ch 91–94).** SDIO WiFi (soldered, in-tree, Ch 91), USB WiFi (swappable, watch the driver, Ch 92), hosted WiFi (ESP co-processor, Ch 93), and WiFi+BT combo (one chip, two radios, coexistence, Ch 94). Together they cover every practical way to get a 2.4 GHz radio onto an i.MX6ULL.

> Next chapter: **Chapter 95: HCI Bluetooth over UART/USB.** Group L (Bluetooth), the dedicated BT controllers (nRF52, BCM4343, CSR), the HCI protocol in depth, and the BlueZ stack for GATT/BLE.

Chapter 95: HCI Bluetooth over UART/USB

What: dedicated Bluetooth controllers and the Linux Bluetooth stack (**BlueZ**). Three controllers compared: **Nordic nRF52** (with Zephyr HCI firmware, UART), **Broadcom BCM4343** (UART), **CSR8510/Realtek RTL8761** (USB dongle). We cover the **HCI protocol** in depth and the BlueZ architecture. You will not write the HCI controller, that lives in the chip's firmware. What you do build is a **BLE GATT peripheral**, in user-space, through BlueZ's D-Bus API.

Why: Bluetooth is how embedded products talk to phones, BLE for "configure-my-device-from-an-app" provisioning, GATT for sensor data, classic BT for audio. Ch 94 brought up the combo-module BT half. This chapter goes deep on the protocol and on *building a GATT service*, the part you actually write. Understanding HCI makes the rest of the stack feel less magical. Building a GATT peripheral is the practical skill.

Focus: the controller firmware owns the radio and link layer. BlueZ provides host protocols and exposes interfaces to the application. A GATT server defines services, characteristics, access policy and notification behavior through BlueZ's D-Bus API. This separation keeps application policy distinct from radio control.

Tooling. This chapter uses `bluez` (`bluetoothctl`, `btmon`, `btmgmt`, `hciattach`), Python `dbus` for the GATT-server example.

- **Ubuntu-base (target):** `apt install bluez bluez-tools python3-dbus`
- **Buildroot:** `BR2_PACKAGE_BLUEZ5_UTILS=y BR2_PACKAGE_BLUEZ5_UTILS_DEPRECATED=y BR2_PACKAGE_PYTHON3_DBUS=y` **Buildroot:** a configuration-driven build system that produces a complete root filesystem and related images.
- Full per-tool reference: Userspace tooling appendix. **MCU bridge:** Think of the rootfs as the firmware image's file-backed runtime environment. On an MCU you link everything into flash. On Linux, programs and config live in this mounted tree. **rootfs:** root filesystem, the directory tree mounted at / that contains /bin, /etc, /dev, and libraries.

95.1 Controller comparison

	Nordic nRF52 (Zephyr HCI)	Broadcom BCM4343	CSR8510 / RTL8761
Connection	UART	UART	USB
BT version	5.x (nRF52840)	4.1/4.2	4.0 / 5.0 (RTL8761)
BLE	yes (excellent)	yes	yes
Classic BT	nRF52: no (BLE only)	yes	yes
Firmware	Zephyr `hci_uart` sample	Broadcom `.hcd` patch	in dongle
Driver	`hci_uart` (h4/h5)	`hci_bcm`	`btusb`
Use case	BLE-only products	combo / classic+BLE	quick dev dongle
Volume price	$3–6 (module)	$2–4	$3–8 (dongle)

Pick guide:

- **nRF52840**: best BLE. Zephyr HCI firmware turns it into a clean HCI controller. BLE-only.
- **BCM4343 / AP6212-BT** (Ch 94): classic BT + BLE. Combo with WiFi.
- **CSR8510/RTL8761 USB dongle**: zero-effort dev, plug into USB, `btusb` handles it.

95.2 The HCI protocol

HCI (Host Controller Interface) is the standardised boundary between two halves of any Bluetooth system: the *host* (Linux plus BlueZ) and the *controller* (the BT chip). HCI is a packet protocol. It defines four packet types:

Type	Direction	Purpose
Command	host → controller	“do something” (scan, connect, set advertising data)
Event	controller → host	“something happened” (device found, connected, data received)
ACL data	both	bulk data (GATT, L2CAP payloads)
SCO data	both	synchronous (classic BT audio)

A command packet:

```
[0x01] [opcode_lo] [opcode_hi] [param_len] [params...]
 ↑ type=command
opcode = OGF (group) << 10 | OCF (command)
e.g., LE Set Advertising Enable = OGF 0x08, OCF 0x000A
```

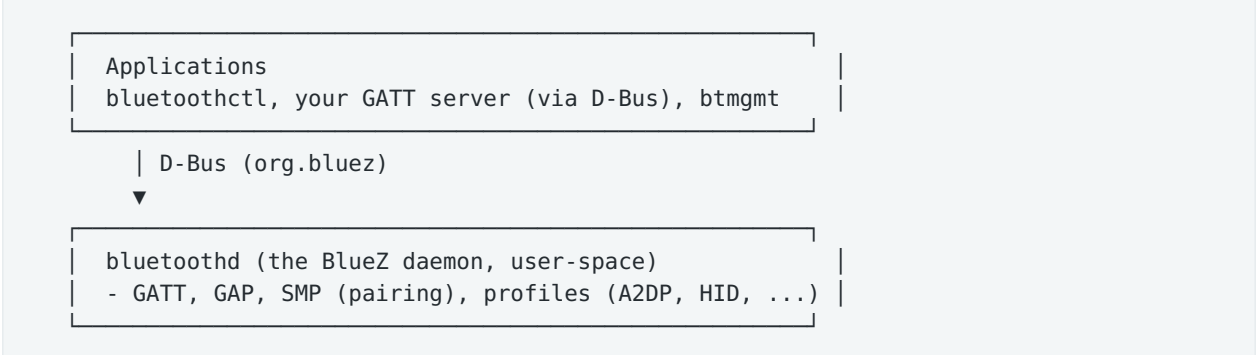
An event packet:

```
[0x04] [event_code] [param_len] [params...]
 ↑ type=event
e.g., LE Advertising Report (a scan found a device)
```

Over UART, these packets are framed with the **H4** protocol (just the type byte prefix, no checksum, relies on UART reliability) or **H5/3-wire** (adds sequence numbers + CRC + retransmission for unreliable UARTs).

You rarely send raw HCI yourself. BlueZ does it for you. Knowing the format lets you read `btmon` traces and debug.

95.3 The BlueZ architecture



```
        │ HCI sockets (AF_BLUETOOTH)
        ▼
   ┌──────────────────────────────────────────────────────┐
   │  Kernel BT subsystem (net/bluetooth/)                │
   │  - L2CAP, HCI core, socket interface                 │
   └──────────────────────────────────────────────────────┘
        │ HCI (H4/H5 over UART, or USB)
        ▼
   ┌──────────────────────────────────────────────────────┐
   │  Controller (nRF52 / BCM4343 firmware)               │
   │  - link layer, radio, encryption                     │
   └──────────────────────────────────────────────────────┘
```

Three layers:

- **Controller** (the chip firmware): radio + link layer. You don't write this.
- **Kernel BT** (`net/bluetooth/`): HCI transport, L2CAP, sockets. You don't write this.
- **bluetoothd** (BlueZ daemon): GATT, pairing, profiles, exposed over **D-Bus**. You don't write this either.
- **Your application**: talks to bluetoothd over D-Bus to define GATT services, advertise, handle reads/writes. **This is what you build.**

95.4 Bringing up the controller

nRF52 with Zephyr HCI (UART)

Flash the nRF52 with Zephyr's `hci_uart` sample (it turns the nRF52 into a pure HCI controller). Wire UART (with flow control). DT:

```
&uart3 {
    uart-has-rtscts;
    status = "okay";

    bluetooth {
        compatible = "zephyr,hci-uart";   /* or generic via btattach */
        max-speed = <1000000>;
    };
};
```

If no specific compatible exists, attach manually:

```
[root@pa-mini:~]# btattach -B /dev/ttymxc2 -S 1000000 -P h4
[root@pa-mini:~]# hciconfig
hci0:   Type: Primary  Bus: UART  ...  UP RUNNING
```

BCM4343 (UART)

As Ch 94, `hci_bcm` + the `.hcd` firmware patch, serdev DT node.

USB dongle (CSR8510 / RTL8761)

The easiest:

```
[root@pa-mini:~]# lsusb
Bus 001 Device 005: ID 0a12:0001 Cambridge Silicon Radio CSR8510 A10
[root@pa-mini:~]# dmesg | grep -i blue
Bluetooth: hci0: BCM: chip ...   (or CSR/RTL)
[root@pa-mini:~]# hciconfig hci0 up
```

`btusb` handles USB BT dongles class-compliantly, plug in, `hci0` appears.

95.5 bluetoothctl, the interactive tool

```
[root@pa-mini:~]# bluetoothctl
[bluetooth]# power on
[bluetooth]# scan on
[NEW] Device AA:BB:CC:DD:EE:FF MyPhone
[bluetooth]# scan off
[bluetooth]# pair AA:BB:CC:DD:EE:FF
[bluetooth]# connect AA:BB:CC:DD:EE:FF
[bluetooth]# menu gatt
[bluetooth]# list-attributes   ← shows the connected device's GATT services
```

`btmon` decodes the HCI traffic live, invaluable for debugging:

```
[root@pa-mini:~]# btmon
> HCI Event: LE Meta Event (0x3e) ...
        LE Advertising Report
          Address: AA:BB:CC:DD:EE:FF
          Data: ...
```

95.6 Building a BLE GATT peripheral (the from-scratch part)

The application boundary is a **GATT server** that advertises a service and exposes characteristics a phone app can read, write or subscribe to. The user-space example below uses BlueZ’s D-Bus API to illustrate that boundary.

GATT structure:

- A **service** (a UUID) groups related data.
- **Characteristics** (each a UUID) are the data points, readable, writable, or notifiable.
- A phone app discovers the service, reads/writes characteristics, subscribes to notifications.

Example: a temperature sensor peripheral. Service = “Environmental Sensing”. Characteristic = “Temperature” (notify the phone when it changes).

Here’s a minimal GATT server using Python + `dbus` (the most readable. C with GLib/sd-bus is the production path):

```
#!/usr/bin/env python3
# A minimal BLE GATT peripheral: exposes a temperature characteristic.
import dbus, dbus.service, dbus.mainloop.glib
from gi.repository import GLib
import struct, random

BLUEZ = 'org.bluez'
GATT_SERVICE_IFACE = 'org.bluez.GattService1'
GATT_CHRC_IFACE = 'org.bluez.GattCharacteristic1'
ADV_IFACE = 'org.bluez.LEAdvertisement1'

TEMP_SVC_UUID  = '0000181a-0000-1000-8000-00805f9b34fb'   # Environmental Sensing
TEMP_CHRC_UUID = '00002a6e-0000-1000-8000-00805f9b34fb'   # Temperature

class TempCharacteristic(dbus.service.Object):
    def __init__(self, bus, index, service):
        self.path = service.path + '/char' + str(index)
        self.notifying = False
        self.value = 2300   # 23.00 °C in 0.01 units
        super().__init__(bus, self.path)
        GLib.timeout_add_seconds(2, self.update_temp)

    def get_properties(self):
        return { GATT_CHRC_IFACE: {
            'Service': self.service_path,
            'UUID': TEMP_CHRC_UUID,
            'Flags': ['read', 'notify'],
        }}

    @dbus.service.method(GATT_CHRC_IFACE, in_signature='a{sv}', out_signature='ay')
    def ReadValue(self, options):
        return [dbus.Byte(b) for b in struct.pack('<h', self.value)]

    @dbus.service.method(GATT_CHRC_IFACE)
    def StartNotify(self):
        self.notifying = True

    @dbus.service.method(GATT_CHRC_IFACE)
    def StopNotify(self):
        self.notifying = False

    def update_temp(self):
        # Simulate a reading (real: read from your BME280, Ch 67)
        self.value = 2300 + random.randint(-50, 50)
        if self.notifying:
            val = [dbus.Byte(b) for b in struct.pack('<h', self.value)]
            self.PropertiesChanged(GATT_CHRC_IFACE, {'Value': val}, [])
        return True

    @dbus.service.signal('org.freedesktop.DBus.Properties', signature='sa{sv}as')
    def PropertiesChanged(self, iface, changed, invalidated):
```

```
        pass

# ... (service registration + advertisement registration boilerplate) ...
# Register the service+characteristic with org.bluez via
# GattManager1.RegisterApplication, and an advertisement via
# LEAdvertisingManager1.RegisterAdvertisement.

if __name__ == '__main__':
    dbus.mainloop.glib.DBusGMainLoop(set_as_default=True)
    # ... setup, register service + advertisement ...
    print("Advertising temperature service. Connect from a phone (nRF Connect app).")
    GLib.MainLoop().run()
```

The full example needs the service-object and advertisement-object registration boilerplate, about 250 lines in total. BlueZ ships `example-gatt-server` in `test/` and most production code starts from that.

What this gives you: a phone running the **nRF Connect** app (or your own app) discovers “Environmental Sensing,” reads the temperature, and subscribes to notifications, getting a push every 2 seconds. This is the standard pattern: a BLE sensor talking to a phone app.

Wire the `update_temp()` to a real BME280 (Ch 67) and you have a working BLE environmental sensor.

The C / production path

For a real product, write the GATT server in C against **sd-bus** (systemd) or **GDBus** (GLib), or use BlueZ’s `gdbus` C helpers. The structure mirrors the Python: define the service + characteristics as D-Bus objects, register with `org.bluez`, handle ReadValue/WriteValue/StartNotify. More code, but no Python runtime and better performance.

95.7 GAP, advertising + the connection flow

Before GATT, a BLE peripheral **advertises**: it broadcasts small packets (“I exist, here’s my name + service UUIDs”) on the 3 advertising channels. A central (phone) scans, sees the advert, connects.

```
Peripheral (i.MX6ULL):  advertise "TempSensor" + service UUID 0x181A
Central (phone):        scan → sees advert → connect
→ GATT discovery → read/subscribe characteristics
```

BlueZ exposes a `LEAdvertisingManager1` D-Bus interface that controls advertising. You register an advertisement object with the name, service UUIDs, and any manufacturer-specific data. The example-gatt-server includes this.

95.8 Key Notes

- HCI separates the radio controller from the Bluetooth host stack. BlueZ exposes application services above that boundary rather than requiring applications to implement link-layer behavior.
- A GATT service defines data and operations, while advertising lets peers discover it. Characteristic permissions and notification semantics form an application protocol.
- UART framing, flow control and firmware must support the controller's transport. The existence of `hci0` does not establish complete feature support.
- Pairing and access policy need explicit design, especially for provisioning secrets or controlling hardware. A discoverable service should not imply unrestricted authority.

95.9 Pitfalls

- **No hardware flow control on UART BT.** At 1–3 Mbps, BT HCI needs RTS/CTS. Without it, HCI packets corrupt → `hci0` flaky or dead. `uart-has-rtscts` + wire the lines.
- **Firmware patch missing.** BCM controllers need the `.hcd`. Without it, reduced functionality or a default BD address.
- **bluetoothd not running.** `bluetoothctl` and the D-Bus API need the `bluetoothd` daemon. `systemctl start bluetooth` (or run `bluetoothd` manually).
- **D-Bus permission denied.** Your GATT server needs permission to talk to `org.bluez`. Run as root, or add a D-Bus policy file granting access.
- **Advertising not enabled.** A GATT server with no advertisement is invisible, the phone can't find it. Register an `LEAdvertisement1` object.
- **Characteristic flags wrong.** A "read" characteristic the app tries to subscribe to fails. Match flags (`read`/`write`/`notify`) to intended use.
- **BD address duplicate.** Two devices with the same BD_ADDR confuse a phone. Program unique addresses (Ch 65 EEPROM).
- **Classic BT on a BLE-only controller.** nRF52 is BLE-only. Trying classic BT (A2DP, etc.) fails. Use a BCM/CSR for classic.
- **MTU too small.** Default BLE ATT MTU is 23 bytes (20 usable). Large characteristic values need MTU negotiation. Otherwise they're truncated. Negotiate a larger MTU.

95.10 Going deeper

- **`net/bluetooth/`**: the kernel Bluetooth subsystem (HCI core, L2CAP, sockets).
- **`drivers/bluetooth/`**: `hci_uart.c`, `hci_bcm.c`, `btusb.c`, `btnordic`/Zephyr-HCI support.

- **BlueZ `test/example-gatt-server` + `example-advertisement`**, adapt these for your GATT peripheral.
- **BlueZ D-Bus API docs** (`doc/gatt-api.txt`, `doc/advertising-api.txt`), the interfaces your app calls.
- **`btmon`, `btmgmt`, `bluetoothctl`**: the BlueZ tools.
- **Bluetooth Core Specification**: HCI (Vol 4 Part E), GATT (Vol 3 Part G), GAP (Vol 3 Part C).
- **Nordic nRF Connect app**: the indispensable BLE-peripheral test tool (phone-side).
- **`Documentation/devicetree/bindings/net/` (bluetooth bindings)**: the DT for UART controllers.

> Next chapter: **Chapter 96: AT-command BLE modules.** The easy path, HM-10, HC-08, BLE without a kernel stack, just UART AT commands. When simplicity beats integration.

Chapter 96: AT-command BLE modules

What: BLE modules that hide the entire Bluetooth stack behind a simple UART AT-command interface. **HM-10** (CC2540/CC2541-based), **HC-08**, **JDY-08**, you send `AT+...` strings over a UART, and the module handles advertising, connection, and a transparent serial data pipe to a connected phone. No BlueZ, no kernel BT stack, no GATT programming.

Why: for products that need “send a few bytes to/from a phone app” and nothing more, the AT-BLE module is dramatically simpler than the full HCI + BlueZ stack of Ch 95. The module is the Bluetooth stack. Linux only talks to a UART. The trade-off is real. You are stuck with the module’s fixed GATT profile, usually a single “transparent UART” characteristic. Throughput tops out at a few hundred bytes per second. The AT command set is vendor-specific and non-standard.

Focus: the module behaves like a wireless serial cable. After configuration, anything you write to the UART appears in the phone app (via a notify characteristic), and anything the phone sends appears on the UART RX. It is a wireless serial port. Linux needs no Bluetooth code, just open `/dev/ttymxc2` and call `read`/`write`.

Tooling. This chapter uses Just a UART terminal: `picocom` or `minicom`.

- **Ubuntu-base (target):** `apt install picocom minicom`
- **Buildroot:** `BR2_PACKAGE_PICOCOM=y` **Buildroot:** a configuration-driven build system that produces a complete root filesystem and related images.
- Full per-tool reference: Userspace tooling appendix. **MCU bridge:** Think of the rootfs as the firmware image’s file-backed runtime environment. On an MCU you link everything into flash. On Linux, programs and config live in this mounted tree. **rootfs:** root filesystem, the directory tree mounted at / that contains /bin, /etc, /dev, and libraries.

96.1 Module comparison

	HM-10	HC-08	JDY-08
Chip	TI CC2540/CC2541	unknown (clone)	Beken BK3431
BLE version	4.0	4.0	4.2

	HM-10	HC-08	JDY-08
Profile	transparent UART (custom service)	transparent UART	transparent UART
Default baud	9600	9600	9600
Range	~30 m	~20 m	~30 m
Config	AT commands	AT commands	AT commands
Cloned variants	many (firmware varies!)	several	several
Volume price	$3–6 (genuine), $1–2 (clone)	$1–2	$1–2

Clone variants are a real problem. There are at least five different “HM-10” modules in the market, each with different firmware and different AT command syntax. A genuine HM-10 (from Jnhuamao) responds to one command set. Clones may differ. Always verify the exact AT dialect of *your* module.

Pick guide:

- **HM-10 genuine**: best-documented, most-supported. Worth the premium for the known command set.
- **HC-08 / JDY-08**: cheaper. Verify the command set before designing around it.

96.2 The transparent-UART model

```
Linux              AT-BLE module           Phone app
─────              ─────────────           ─────────
/dev/ttymxc2  ◄──► UART ◄──► [BLE stack] ◄──► BLE ◄──► custom characteristic

Two modes:
- COMMAND mode: AT strings configure the module (name, baud, role).
- DATA mode (connected): bytes flow transparently both ways.
```

After a phone connects, the module enters **data mode**: every byte Linux writes to the UART is sent to the phone (as a GATT notification on the module’s TX characteristic). Every byte the phone writes (to the module’s RX characteristic) appears on Linux’s UART RX. It’s literally a wireless UART cable.

This is the main reason to choose an AT-BLE module. Your application is plain serial I/O, no D-Bus, no GATT objects, no BlueZ daemon.

96.3 Configuration via AT commands

In command mode (before a connection, or after entering it via a mode pin):

```
AT              → OK              (is the module alive?)
AT+NAME?        → +NAME=HMSoft    (query name)
AT+NAMEMyDevice → OK              (set name to "MyDevice")
AT+BAUD?        → +BAUD=0         (0 = 9600)
AT+BAUD4        → OK              (set to 115200)
AT+ROLE0        → OK              (0 = peripheral/slave, 1 = central/master)
AT+ADTY0        → OK              (advertising type)
AT+RESET        → OK              (apply + reboot)
AT+VERS?        → +VERS:HMSoft V540  (firmware version — tells you the clone variant)
```

Caveat on syntax. The genuine HM-10 commands omit the `\r\n` terminator and the `=` sign for set commands in older firmware (e.g., `AT+NAMEMyDevice`, not `AT+NAME=MyDevice`). Clone command sets vary. Run `AT+VERS?` first, the version response identifies your variant.

A configuration sequence from Linux:

```
int fd = open("/dev/ttymxc2", O_RDWR | O_NOCTTY);
/* set 9600 8N1 */
write(fd, "AT+NAMEMySensor", 15);   expect_ok(fd);
write(fd, "AT+BAUD4", 8);            expect_ok(fd);   /* 115200 */
write(fd, "AT+ROLE0", 8);            expect_ok(fd);   /* peripheral */
write(fd, "AT+RESET", 8);
/* reopen at 115200 */
```

96.4 Using it, just a serial port

After configuration, the module advertises automatically. When a phone connects, you’re in data mode. From Linux:

```
int fd = open("/dev/ttymxc2", O_RDWR | O_NOCTTY);
/* configured at 115200 8N1 */

/* Send a sensor reading to the phone */
char msg[32];
int temp = read_bme280();          /* Ch 67 */
snprintf(msg, sizeof(msg), "T=%d.%02d\n", temp/100, temp%100);
write(fd, msg, strlen(msg));        /* appears in the phone app */

/* Receive commands from the phone */
char buf[64];
int n = read(fd, buf, sizeof(buf));  /* whatever the phone sent */
```

```
if (n > 0 && strncmp(buf, "LED ON", 6) == 0)
    gpiod_set_value(led, 1);
```

That is the whole integration. There is no Bluetooth code, just `read` and `write` on a UART. A phone app (e.g., “Serial Bluetooth Terminal” or “LightBlue”) connects to the module’s name, opens the transparent characteristic, and exchanges text.

Compare to Ch 95’s GATT server: about 250 lines of D-Bus code versus ten lines of serial I/O. AT modules trade features for simplicity.

96.5 When AT-BLE is the right choice (and when it isn’t)

Right choice when:

- You need a simple bidirectional data pipe to a phone app.
- “A few hundred bytes per second” is enough throughput.
- You want zero Bluetooth code on the Linux side.
- The product is simple (a sensor that reports + takes a few commands).
- BOM cost matters and a $1.50 module beats a $4 controller + the dev effort.

Wrong choice when:

- You need standard GATT services (so any BLE app, not just a serial-terminal app, works).
- You need multiple characteristics, proper service UUIDs, or BLE standard profiles (HID, HRS, etc.).
- You need >few hundred bytes/sec throughput.
- You need control over advertising, security/pairing, or connection parameters.
- You’re building a polished product where the phone app is a real app (not a serial terminal).

For a real product with a custom phone app, the BlueZ GATT path (Ch 95) is better, it gives standard GATT the app can use cleanly. The AT module fits quick, simple “wireless serial cable” use cases.

96.6 Key Notes

- An AT-BLE module owns most of the Bluetooth stack and exposes a serial-facing service. Linux sees its command/data protocol, not necessarily an HCI controller managed by BlueZ.

- Command mode and connected data mode have different interpretations of the same bytes. Firmware dialect and line framing are part of the host-module contract.
- A transparent link still needs application framing, reconnect handling and flow control. Wireless delivery should not be treated as an infallible serial cable.
- Reduced integration effort comes with limits on GATT flexibility, connections, throughput and security policy. Those limits belong in the architecture decision.

96.7 Pitfalls

- **Clone command-set differences.** “HM-10” clones use different AT syntax (`AT+NAME=x` vs `AT+NAMEx`, with/without `\r\n`). Always `AT+VERS?` first and match your module’s dialect.
- **No line terminator (genuine HM-10).** Genuine HM-10 v5xx commands have *no* CR/LF. Sending `AT\r\n` to one may fail. Send bare `AT`. Clones often *require* `\r\n`. Test.
- **Command mode vs data mode confusion.** Once connected, the module is in data mode, AT commands are passed through as data, not interpreted. To reconfigure, disconnect first (or use a mode pin if the module has one).
- **Baud mismatch after AT+BAUD.** Set the baud, reset, then *reopen the UART at the new baud*. Forgetting this = no communication.
- **Throughput ceiling.** BLE’s connection interval limits throughput to a few hundred bytes/sec to a few KB/sec. Don’t expect serial-cable speeds.
- **No pairing/security by default.** Most AT modules connect without pairing, anyone can connect. For sensitive data, this is a security hole. The full BlueZ stack (Ch 95) gives proper pairing/encryption.
- **Single connection.** AT modules typically support one connection at a time. For multiple simultaneous phones, you need a real GATT server.
- **Auto-advertising can’t be fully disabled on some.** The module advertises whenever not connected, a power and security consideration.

96.8 Going deeper

- **HM-10 datasheet + AT command guide (Jnhuamao)**: the genuine command set.
- **The Linux `termios` API**: UART configuration (the only “driver” you need).
- **Ch 95**: the full BlueZ GATT path, for when AT-BLE is too limited.
- **Ch 103**: UART AT-command cellular modems use the same “talk-to-a-UART” pattern.

- **“Serial Bluetooth Terminal” (Android) / “LightBlue” (iOS)**: phone-side test apps.

> Next chapter: **Chapter 97: BLE Mesh.** Beyond point-to-point, a self-healing mesh of dozens of BLE nodes, the bluez-mesh stack, provisioning, and mesh models for lighting/sensor networks.

Chapter 97: BLE Mesh

What: Bluetooth Mesh, a many-to-many networking layer built on BLE advertising, where dozens-to-thousands of nodes relay messages for each other to cover a building. This chapter covers four things: the mesh architecture (elements, models, addresses, publish/subscribe), the **bluez-mesh** stack on Linux, the provisioning flow that adds a node to a network, and a worked lighting-control example with the i.MX6ULL as gateway and provisioner.

Why: BLE point-to-point (Ch 95) reaches one device at ~30 m. BLE Mesh covers an entire building with hundreds of nodes: smart lighting (the dominant use case), building sensors, industrial monitoring. Nodes relay each other's messages to extend coverage. It is the technology behind commercial smart-lighting systems, the kind installed in offices and warehouses. An i.MX6ULL makes a good mesh gateway, bridging mesh traffic to WiFi or the cloud. It can also act as a provisioner, adding new nodes to the network.

Focus: mesh is a publish/subscribe protocol layered on flooded BLE advertisements, with addresses tied to models. A node has *elements*, each with *models* (e.g., a "Generic OnOff Server" model for a light). Messages are *published* to *group addresses*. Nodes *subscribed* to that group act on them. "Turn off all kitchen lights" means: publish OnOff=0 to the "kitchen" group. Every light subscribed to "kitchen" responds. Flooding plus relay gives whole-building coverage without any backbone wiring.

Tooling. This chapter uses `bluez` + `bluetooth-meshd` + `mesh-cfgclient`.

- **Ubuntu-base (target):** `apt install bluez bluez-meshd`
- **Buildroot:** `BR2_PACKAGE_BLUEZ5_UTILS=y  # (mesh requires the Buildroot experimental option)` **Buildroot:** a configuration-driven build system that produces a complete root filesystem and related images.
- Full per-tool reference: Userspace tooling appendix. **MCU bridge:** Think of the rootfs as the firmware image's file-backed runtime environment. On an MCU you link everything into flash. On Linux, programs and config live in this mounted tree. **rootfs:** root filesystem, the directory tree mounted at / that contains /bin, /etc, /dev, and libraries.

97.1 Why mesh, not point-to-point

	BLE point-to-point (Ch 95)	BLE Mesh
Topology	star (1 central, N peripherals)	mesh (any-to-any via relays)
Range	~30 m (1 hop)	whole building (multi-hop relay)
Nodes	~7 simultaneous connections	thousands
Addressing	per-connection	unicast + group + virtual addresses
Model	GATT (connection-oriented)	publish/subscribe (connectionless)
Use case	one device ↔ phone	building-scale lighting/sensors
Killer app	provisioning, sensor-to-phone	smart lighting

Mesh sacrifices some point-to-point simplicity. In return you get scale and coverage. A message hops node-to-node. A light in a far room can relay a message it overhears, extending the range far beyond a single radio's reach.

97.2 Mesh architecture

```
Node (a physical device, e.g., a smart light)
 └─ Element (an addressable part; a light might have 1 element)
     └─ Models (the functional units):
         - Generic OnOff Server   (responds to on/off)
         - Light Lightness Server (responds to brightness)
         - Health Server          (reports faults)
```

Key concepts:

- **Node**: a provisioned device in the network. Has a unicast address.
- **Element**: an addressable entity within a node. A multi-gang switch has multiple elements (one per gang).
- **Model**: defines behavior, a "Generic OnOff Server" handles on/off. A "Generic OnOff Client" sends on/off. SIG-defined models (standard) or vendor models (custom).

- **Address types**:
  - **Unicast**: one element.
  - **Group**: a set of elements (“all kitchen lights”).
  - **Virtual**: a label-hashed group.
- **Publish/Subscribe**: a model *publishes* messages to an address. Models *subscribed* to that address receive them. A wall switch publishes “OnOff=0” to group “kitchen”. All kitchen lights subscribe to “kitchen” and turn off.
- **Relay**: nodes with the relay feature re-broadcast messages, extending range. This flooding is how a message reaches across a building.

Security

Mesh has two key tiers:

- **Network key (NetKey)**: shared by all nodes in the network. Encrypts at the network layer (relay nodes can relay without decrypting the application payload).
- **Application key (AppKey)**: per-application. Encrypts the payload. A light and a switch share an AppKey. Relay nodes don’t have it.

This two-tier scheme lets relays forward traffic they can’t read, important for security at scale.

97.3 Provisioning, adding a node to the network

A fresh node is **unprovisioned**, it advertises “I want to join.” A **provisioner** (a phone app, or your i.MX6ULL) runs the provisioning protocol:

```
1. Unprovisioned node beacons.
2. Provisioner discovers it, initiates provisioning.
3. ECDH key exchange (the node and provisioner derive a shared secret).
4. (Optional) Out-of-band authentication (a number to confirm, a QR code).
5. Provisioner assigns: unicast address, NetKey, an IV index.
6. Node is now provisioned — part of the network.
7. Provisioner (or a Configuration Client) binds AppKeys to the node's models
   and sets up publish/subscribe.
```

Provisioning is the security-critical step, it’s where keys are distributed. The OOB authentication prevents a rogue provisioner from hijacking nodes.

97.4 bluez-mesh on Linux

Linux's BLE Mesh stack is **bluez-mesh**, a separate daemon (`bluetooth-meshd`) from the main `bluetoothd`, with its own D-Bus API. It uses the same HCI controller (Ch 95) but runs the mesh protocol stack.

```
   Applications (your mesh node logic, mesh-cfgclient)
        │ D-Bus (org.bluez.mesh)
        ▼
   bluetooth-meshd (the mesh daemon)
        │ HCI sockets
        ▼
   Kernel BT subsystem
        │ HCI
        ▼
   Controller (must support BLE advertising + scanning)
```

The controller needs to support BLE advertising + scanning (any BLE 4.0+ controller does, nRF52, BCM4343, a USB dongle).

Start the daemon:

```
[root@pa-mini:~]# bluetooth-meshd --config /var/lib/bluetooth/mesh &
```

Tools:

- **`mesh-cfgclient`**: a provisioner + configuration client (provision nodes, bind keys, set pub/sub).
- **`meshctl`**: older combined tool.

97.5 A worked example, the i.MX6ULL as a mesh provisioner + gateway

Scenario: 5 smart lights (each a small nRF52 node running a "Generic OnOff Server"), and the i.MX6ULL as the provisioner + gateway (bridging the mesh to MQTT/cloud, so a cloud command controls the lights).

Step 1: create the network (on the i.MX6ULL)

```
[root@pa-mini:~]# mesh-cfgclient
[mesh-cfgclient]# create          ← create a new network (generates NetKey)
Created mesh network with token ...
[mesh-cfgclient]# appkey-create 0 0   ← create an AppKey, bound to NetKey 0
```

Step 2: provision each light

```
[mesh-cfgclient]# discover-unprovisioned on   ← scan for unprovisioned nodes
Scan result: device UUID aabbcc...
[mesh-cfgclient]# provision aabbcc...           ← provision this node
Assigning address 0x0002
Provisioning done
```

Repeat for each light (addresses 0x0002, 0x0003, …).

Step 3: bind the AppKey + configure pub/sub

```
[mesh-cfgclient]# menu config
[config]# target 0002                          ← configure node at 0x0002
[config]# appkey-add 0                          ← give it AppKey 0
[config]# bind 0 0 1000                          ← bind AppKey 0 to the OnOff model
(0x1000)
[config]# sub-add 0002 c000 1000                 ← subscribe its OnOff model to group
0xC000 ("all lights")
```

Now node 0x0002's OnOff model has the AppKey and is subscribed to group 0xC000.

Step 4: control the lights

```
[mesh-cfgclient]# menu onoff
[onoff]# target c000        ← address the "all lights" group
[onoff]# onoff 1            ← turn all subscribed lights ON
[onoff]# onoff 0            ← turn them all OFF
```

One message to group 0xC000 → all 5 lights respond. Add more lights, subscribe them to 0xC000, and they join the group automatically, no reconfiguration of the others. Group addressing is what makes this possible.

Step 5: bridge to MQTT (the gateway role)

Your i.MX6ULL app, via the `org.bluez.mesh` D-Bus API, subscribes to mesh status messages and publishes commands. Wire it to an MQTT client:

```
# Conceptual gateway loop
on_mqtt("home/lights/kitchen/set", lambda payload:
    mesh_publish(group=0xC000, model="OnOff", value=int(payload)))
on_mesh_status(lambda node, value:
    mqtt_publish(f"home/lights/{node}/state", value))
```

Now a cloud/phone MQTT command controls the mesh lights, and light state changes propagate to the cloud. The i.MX6ULL is the **mesh-to-IP gateway**, the typical role for a Linux device in a mesh network (the lights are cheap nRF52 nodes. The gateway is the one Linux box).

97.6 Building a mesh node application

For the i.MX6ULL to be a *node* (not just a provisioner), e.g., a mesh sensor that publishes temperature, you write an application against `org.bluez.mesh`'s `Application1` + `Element1` interfaces, declaring your models and handling incoming messages. BlueZ ships `test/test-mesh` as a starting point. The structure: declare elements + models, register with `bluetooth-meshd`, implement the model message handlers (`DevKeyMessageReceived`, `MessageReceived`).

The structure is similar to the GATT server of Ch 95 but applies to mesh models. It is more involved, and the bluez-mesh D-Bus API is less mature than the GATT one. For most products, the i.MX6ULL is the *provisioner/gateway* (using `mesh-cfgclient`), and the cheap nodes (nRF52 with Zephyr/nRF SDK mesh firmware) are the *servers*.

97.7 Key Notes

- Mesh models define application behavior, elements contain models, and publication and subscription connect them through addresses. Network membership alone does not configure useful application behavior.
- Provisioning, network keys, application keys and model binding are distinct authority steps.
- Relaying improves reach at the cost of radio traffic. More relays are not automatically better, so density, retry and message-rate assumptions matter.
- Sequence state and keys have persistent lifetimes. Reset and recovery policy must preserve or deliberately re-establish replay-protection and identity state.

97.8 Pitfalls

- **bluetooth-meshd vs bluetoothd.** They're separate daemons and can conflict over the HCI controller. Run mesh on a dedicated controller, or ensure only one daemon claims `hci0`.
- **Provisioning OOB confusion.** If OOB authentication is configured, both sides must agree on the method (number, QR, none). Mismatch = provisioning fails.
- **AppKey not bound.** A provisioned node that hasn't had an AppKey bound to its model can't decrypt application messages, it joins the network but ignores commands. Always bind the AppKey.
- **Forgetting subscription.** A node bound to an AppKey but not subscribed to the target group won't receive group messages. Both bind *and* subscribe.
- **Relay feature off.** If no nodes relay, range is limited to one hop. Enable relay on enough nodes for coverage (but not *all*, too many relays cause message storms).

- **IV index / replay protection drift.** Mesh uses sequence numbers as replay protection. If a node's stored sequence state is lost (because its flash was erased), the network may reject it. Persist mesh state correctly.
- **bluez-mesh maturity.** The Linux mesh stack and D-Bus API are less polished than GATT. Expect rough edges. Commercial mesh systems often use vendor stacks (Silicon Labs, Nordic) on the nodes and a custom gateway.
- **Provisioning capacity.** A network has a finite address space and key storage. Plan addressing (unicast ranges, group allocation) before deploying hundreds of nodes.

97.9 Going deeper

- **`bluetooth-meshd` + `mesh-cfgclient`** (BlueZ), the Linux mesh daemon + provisioner tool.
- **BlueZ `test/test-mesh`**: a sample mesh node application.
- **BlueZ `doc/mesh-api.txt`**: the `org.bluez.mesh` D-Bus API.
- **Bluetooth Mesh Profile Specification** + **Mesh Model Specification** (Bluetooth SIG), the canonical references.
- **Nordic nRF5 SDK for Mesh / Zephyr Bluetooth Mesh**: for the *node* firmware (the cheap nRF52 lights).
- **Silicon Labs Bluetooth Mesh docs**: an alternative node stack with good tutorials.
- **Ch 95**: the underlying HCI controller bring-up that mesh sits on.

> **End of Group L, Bluetooth (Ch 95–97).** Full HCI + BlueZ GATT (Ch 95, the capable path), AT-command transparent-serial BLE (Ch 96, the simple path), and BLE Mesh (Ch 97, the building-scale path). The i.MX6ULL spans roles from a single BLE sensor to a mesh gateway.

> Next chapter: **Chapter 98: LoRa.** Group M (Long-range & specialty wireless), kilometre-range sub-GHz radio with the SX127x/SX126x, LoRaWAN vs point-to-point, and the spreading-factor trade-offs.

Chapter 98: LoRa

> **sysfs:** a kernel-generated filesystem under /sys that exposes devices, drivers, and attributes.

> **What: LoRa**, a Semtech-proprietary sub-GHz long-range modulation (chirp spread spectrum) reaching multiple kilometres at sub-100 kbps. Four real radios compared: **Semtech SX1276/78** (legacy LoRa, 433/868/915 MHz), **SX1262** (current generation, lower power, FSK + LoRa), **LLCC68** (cheap SX1262 sibling, limited SF range), and **EByte E22-900M30S** (a ready-to-fly SX1262 module with PA + LNA). We dissect the radio's SPI register map, walk the kernel `sx127x` / `sx1301` candidate drivers (and why **most production LoRa stacks live in user space**), write a tiny SPI-only LoRa driver from scratch in user space (no kernel driver), then bring up a real LoRaWAN gateway with **ChirpStack**.
>
> **Why:** LoRa is the only short-message radio that crosses kilometres without infrastructure, sub-watt, and into deep building penetration. It's the workhorse for agricultural sensors, wildlife trackers, water-meter telemetry, and remote alarms. Many engineers copy the "Arduino LoRa library" without understanding the modulation, the registers, or why a bad antenna costs most of your link budget. This chapter walks the whole stack.
>
> **Focus: LoRa is a SPI radio with two state machines on top, modem and packet handler, and a tightly coupled antenna RF chain you cannot ignore**. Chirp spread spectrum (CSS) gives ~–137 dBm sensitivity at SF12/125 kHz, but the air-time at SF12 is *seconds per packet*, throttled by duty-cycle regulation (1 % on 868 MHz EU). The four-tuple of spreading factor, bandwidth, coding rate, and preamble is the whole engineering job. You must know what each one costs in air-time, sensitivity, and power. The radio is easy. The link budget is the engineering.

98.1 LoRa vs everything else short-message

	LoRa (CSS)	BLE (Ch 95–97)	Sub-GHz FSK (Ch 99)	NB-IoT (Ch 104)	WiFi
Range (open field)	5–15 km	~30 m	~1 km	cellular cell	~50 m

	LoRa (CSS)	BLE (Ch 95–97)	Sub-GHz FSK (Ch 99)	NB-IoT (Ch 104)	WiFi
Range (urban, dense)	1–3 km	~10 m	~300 m	cellular cell	~20 m
Bitrate	0.3–50 kbps	1 Mbps	1–500 kbps	30 kbps	10–600 Mbps
RX current	~10 mA	~5 mA	~13 mA	~30 mA	~80 mA
Sleep / cold	200 nA	1 µA	1 µA	1 µA (PSM)	1 mA
Infrastructure needed	optional (LoRaWAN GW)	none / phone	none	carrier cell	router
Spectrum	ISM sub-GHz	2.4 GHz	ISM sub-GHz	licensed cellular	2.4/5 GHz
Cost (radio IC)	$3–6	$1–3	$1–4	$5–10	$2–5
Air time per packet	**seconds at SF12**	1 ms	5 ms	200 ms	<1 ms
Duty cycle limit	0.1–10 % (regional)	none	0.1–10 %	none	none

The trade is direct: LoRa buys range with bitrate. At SF12/BW125, you transmit ~250 bits per *second* (yes, per second). A 50-byte payload costs **~2 seconds of air time**. EU 868 MHz lets you transmit only 1 % of the time → ~30 packets/hour. That single number drives every product decision.

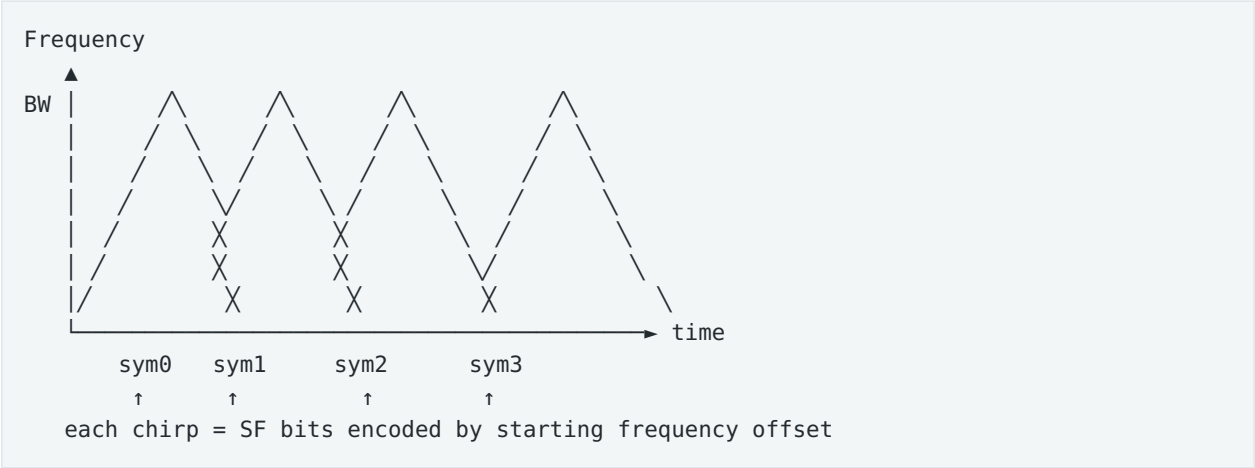
Pick guide:

- **SX1276/78**: legacy projects, abundant code, OK power. Pick SX1262 instead for anything new.
- **SX1262**: the default new design. Lower TX current (~118 mA at +22 dBm vs ~120 mA at +20 dBm for SX1276), better RX sensitivity, FSK + LoRa in one chip, +22 dBm internal PA.

- **LLCC68**: SX1262 register-compatible *almost*. Only SF5–SF11 (no SF12). Save $0.50/unit if SF12 isn't needed.
- **EByte E22**: buy this if you want a finished module with PA, LNA, SMA connector, RF shield. Slightly closed (you can't change the matching network) but eliminates 80 % of the RF risk.

98.2 How chirp spread spectrum actually works (the 5-minute primer)

LoRa modulation is **chirp spread spectrum (CSS)**: each symbol is a frequency *sweep* (a chirp) over the channel bandwidth. A symbol encodes `SF` bits, where **SF (spreading factor)** is 7..12. The chirp starts at a frequency offset proportional to the symbol value and sweeps linearly.



Key consequences:

- **Symbol time** doubles every increase of SF: `Tsym = 2^SF / BW`. At SF12/BW125: 32.768 ms *per symbol*. At SF7/BW125: 1.024 ms.
- **Bitrate** ≈ `SF × BW / 2^SF × CR`. At SF12/BW125/CR4/5: ~293 bps. At SF7/BW125/CR4/5: ~5.5 kbps.
- **Sensitivity** improves about 2.5 dB per SF step. SF7 is around −123 dBm. SF12 is around −137 dBm. That is **14 dB** of headroom, or roughly 5× the range.
- **Demodulation** is processing-gain-based: even buried in noise, the chirp correlation pulls the signal out. This is why LoRa works below the noise floor.

The four-tuple you tune for every link:

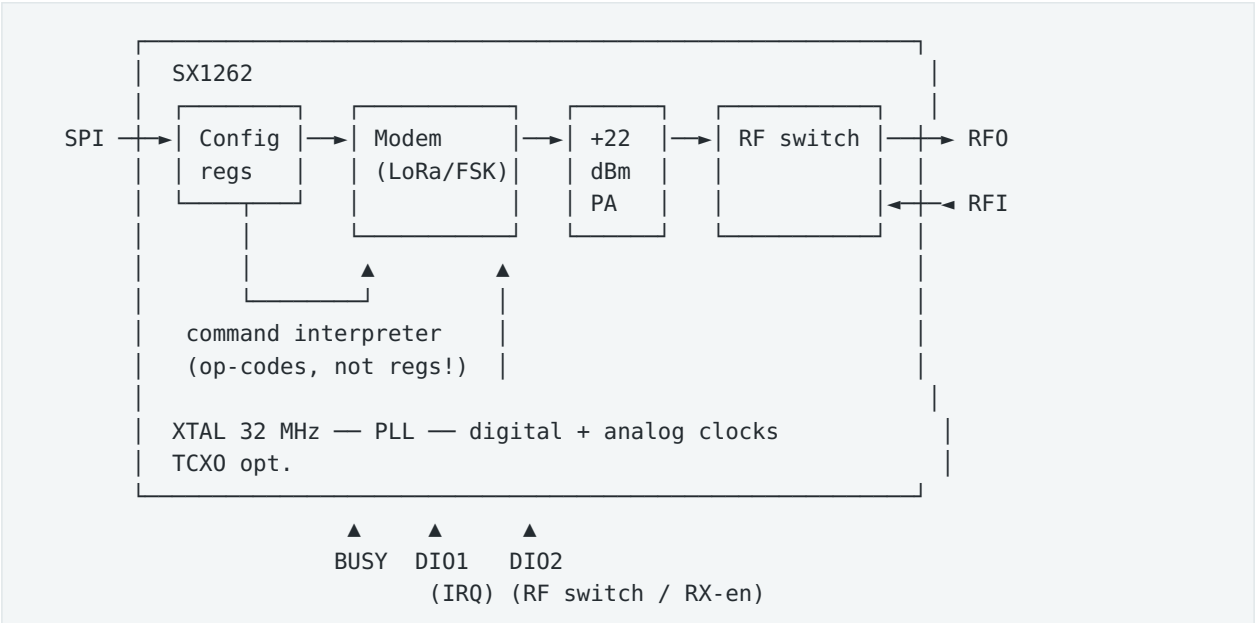
Knob	Range	What it costs
SF (spreading factor)	7–12	doubles air time per step; +2.5 dB sensitivity per step
BW (bandwidth)	7.8 / 10.4 / 15.6 / 20.8 / 31.25 / 41.7 / 62.5 / 125 / 250 / 500 kHz	wider = faster but less sensitive; 125 kHz is the universal default
CR (coding rate)	4/5, 4/6, 4/7, 4/8	extra parity → +overhead, +robustness
Preamble length	6–65535 symbols	longer = receiver wake-up time, but consumes air time

A 50-byte payload at SF12/BW125/CR4/5: ~2.3 s. At SF7/BW125/CR4/5: ~110 ms. Same payload, 20× the throughput. *But*: SF7 reaches ~3 km open field. SF12 reaches ~10 km. The product choice is in that ratio.

98.3 SX1276 and SX1262, what's in the chip

Both are single-chip transceivers: digital baseband + IF + RF front-end + PA. Antenna in/out is one or two pins (TX vs RX path selected by an internal RF switch on SX1262. External pin selection on SX1276).

Block diagram (SX1262, the one to use for new designs):



The two chips differ in interface style:

- **SX1276/78** are **register-mapped**: `WRITE_REG addr value`, `READ_REG addr` on SPI. ~120 registers documented in the datasheet. You configure by writing each register.
- **SX1262** is **command-based**: `SetPacketType`, `SetRfFrequency`, `SetTxParams`, `SetModulationParams`, `WriteBuffer`, `SetTx`. Each is an opcode + payload over SPI. The chip parses it. Easier to use, but the code looks different from SX1276's.

This is the #1 source of porting pain when moving SX1276 code to SX1262.

SX1276 register map (the parts that matter)

Addr	Name	Purpose
0x00	FIFO	TX/RX FIFO data (R/W), burst-read pulls bytes
0x01	OpMode	LoRa/FSK toggle, mode (sleep/standby/tx/rx)
0x06–08	FrfMsb/Mid/Lsb	RF frequency: `Fcarrier = Frf × 32e6 / 2^19`
0x09	PaConfig	PA selection (RFO vs PA_BOOST), output power
0x0E	FifoTxBaseAddr	where in the 256-byte chip RAM TX starts
0x0F	FifoRxBaseAddr	where RX starts
0x10	FifoAddrPtr	current FIFO read/write pointer
0x12	IrqFlags	IRQ status, write 1 to clear
0x13	RxNbBytes	bytes in the last received packet
0x1D	ModemConfig1	BW (upper 4 bits), CR (3 bits), explicit/implicit header

Addr	Name	Purpose
0x1E	ModemConfig2	SF (upper 4), CRC enable, TX continuous
0x20	PreambleMsb	preamble length (in symbols)
0x21	PreambleLsb	
0x22	PayloadLength	TX length (or expected RX in implicit mode)
0x39	SyncWord	0x12 (private) or 0x34 (LoRaWAN)
0x40	DioMapping1	which IRQ source each DIO pin signals
0x42	Version	0x12 for SX1276

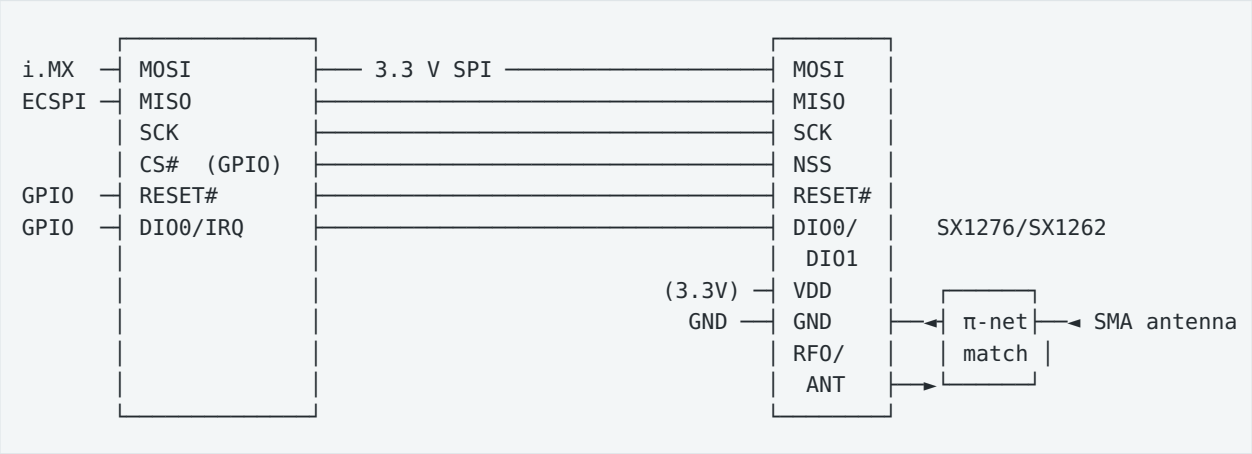
The TX sequence in pseudocode:

```
1. WRITE_REG(OpMode, LoRa|Standby)
2. WRITE_REG(FrfMsb..Lsb, freq)
3. WRITE_REG(PaConfig, PA_BOOST | output_power)
4. WRITE_REG(ModemConfig1, BW125 | CR45 | explicit_hdr)
5. WRITE_REG(ModemConfig2, SF7 | CRC_on)
6. WRITE_REG(PreambleLsb, 8)
7. WRITE_REG(PayloadLength, N)
8. WRITE_REG(FifoTxBaseAddr, 0); WRITE_REG(FifoAddrPtr, 0)
9. BURST_WRITE(FIFO, payload, N)
10. WRITE_REG(DioMapping1, DIO0=TxDone)
11. WRITE_REG(OpMode, LoRa|Tx)        ← starts TX
12. wait for DIO0 rising edge        ← TxDone interrupt
13. WRITE_REG(IrqFlags, 0xFF)          ← clear all flags
14. WRITE_REG(OpMode, LoRa|Standby)
```

That’s it. Eighty registers in the datasheet. This loop uses twelve.

98.4 Wiring, what the schematic must do

The radio is easy. The RF path is where projects die.



Mandatory rules. Every one of these has destroyed a board in the field:

1. **Antenna or 50 Ω dummy load at TX every time.** Transmitting into an open or short pin destroys the PA in milliseconds. This matters most during bring-up, when it is tempting to power the radio without an antenna just to see if it responds.
2. **VDD bulk capacitor ≥ 10 µF + 100 nF near the chip.** Each transmit pulse is ~120 mA at +22 dBm, the supply must hold up. A weak rail = power droop = the modem retransmits = battery dies overnight.
3. **Ground plane under the radio.** Single-sided protoboard works at SF12 for ~30 m. Move to a real PCB with continuous ground for anything beyond eval.
4. **TCXO recommended for SF11/SF12.** The crystal must hold ±20 ppm over temperature. A cheap XTAL drifts and the SF12 demodulator (very narrow effective bandwidth) loses the signal. SX1262 has TCXO control built in (`SetDIO3AsTCXOCtrl`).
5. **PA_BOOST vs RFO on SX1276.** Two output paths. PA_BOOST is the high-power one (up to +20 dBm). RFO is the lower one (max +14 dBm). Pick *one* and route only that to your antenna. Asserting +20 dBm on RFO destroys the chip.
6. **EByte modules** integrate the matching network and add a +30 dBm PA. They eliminate steps 1–5 if you also buy a real antenna. Total project savings: enormous.

98.5 The kernel side, and why most LoRa stacks aren’t in the kernel

> **Driver choice:** Use the in-tree, maintained driver first. Use out-of-tree, spidev, or custom-driver paths only after you accept the kernel-version maintenance cost and document who owns updates.

There's no `subsystem/lora/` in mainline. There are out-of-tree drivers (`sx127x-driver`, `sx1301-cpld-driver` for the concentrator chip used in LoRaWAN gateways), but the dominant pattern is **`spidev` + a user-space stack**. Reasons:

1. **LoRaWAN MAC is huge and stateful.** The link-layer (network/application keys, ADR, retransmission, multi-channel scanning) lives in a daemon (ChirpStack, lora-net, lorawan-stack). A kernel driver isn't where this belongs.
2. **No standard MAC.** P2P LoRa is application-defined. LoRaWAN is the most common but not the only. Every product wraps the PHY differently.

> **PHY:** physical-layer block or chip that converts digital MAC signals to electrical or radio signals.

1. **Timing is not RT-critical.** LoRa packets are tens-to-thousands of milliseconds long. User-space SPI latency (a few ms) is irrelevant.
2. **Mainline has refused most LoRa kernel patches** for these reasons. The community converged on user-space.

This is unusual in this book. Almost every other chapter says "the kernel driver does this." On LoRa, the kernel is only the SPI bus controller.

There *is* one important exception: **LoRaWAN gateway concentrators** (Semtech SX1301, SX1302, SX1303), multi-channel parallel demodulators on PCIe or SPI used in *gateways* (not nodes). Even these are bound by `spidev`, with a user-space "packet forwarder" (`lora_pkt_fwd`, ChirpStack-MP-Packet-Forwarder). The hardware is FPGA-like. The abstraction is in user space.

What an out-of-tree LoRa kernel driver looks like (for context)

The community `sx127x-driver` and `lora-net` Linux trees expose the radio as a **netdev**, `lora0` shows up like `wlan0`, you `ip link set lora0 up`, and a SOCK_RAW socket gives you packets. The driver pattern is:

```
// Simplified from lora-net out-of-tree
static const struct net_device_ops sx127x_netdev_ops = {
    .ndo_open       = sx127x_open,        /* set RX mode, enable IRQ */
    .ndo_stop       = sx127x_stop,        /* sleep mode */
    .ndo_start_xmit = sx127x_xmit,        /* push frame to FIFO, start TX */
};

static int sx127x_probe(struct spi_device *spi) {
    struct net_device *ndev = alloc_netdev(sizeof(*priv), "lora%d", ...);
    /* attach SPI, IRQ on DIO0, GPIO reset */
    request_threaded_irq(spi->irq, NULL, sx127x_irq_thread,
                         IRQF_ONESHOT, "sx127x", priv);
    /* probe by reading reg 0x42 — must be 0x12 for SX1276 */
    if (sx127x_read_reg(spi, 0x42) != 0x12) return -ENODEV;
    sx127x_reset(priv);
```

```
    sx127x_init_modem(priv, SF7, BW125, CR45);
    return register_netdev(ndev);
}
```

The threaded IRQ on DIO0 fires on `TxDone`/`RxDone`, the handler reads `IrqFlags`, pulls the FIFO, hands the skb up. It looks like every Linux netdev driver, just with chirp modulation under the hood.

> **MCU bridge:** Think of an IRQ like an EXTI/NVIC interrupt path, except Linux splits the hard interrupt from deferred work and must share lines across drivers. **IRQ:** interrupt request, the signal path that tells the CPU or interrupt controller that hardware needs service.

But the **MAC**, addressing, ADR, retransmission, LoRaWAN OTAA join, is *not* in the driver. That's the daemon's job in user space.

For this book, we follow the dominant pattern: **`spidev` + user-space driver + ChirpStack daemon**. The driver internals above are for understanding what a netdev-wrapped path would look like.

98.6 From scratch, a user-space SX1276 driver in C (transmit + receive)

We build the smallest possible LoRa driver: open `spidev`, reset the chip, configure modem, transmit a packet, receive a packet. ~200 lines, no LoRaWAN, no MAC. Pure PHY. Two boards running this can ping each other at 5 km.

sx1276_min.c:

```
/* Minimal user-space SX1276 LoRa driver — PHY only, no LoRaWAN.
 * Pair: two i.MX6ULL boards, each with an SX1276 module on ECSPI1.
 * Example: board A sends "hello %d"; board B prints received with RSSI/SNR.
 */
#include <fcntl.h>
#include <stdint.h>
#include <stdio.h>
#include <string.h>
#include <unistd.h>
#include <sys/ioctl.h>
#include <linux/spi/spidev.h>
#include <linux/gpio.h>

#define REG_FIFO         0x00
#define REG_OP_MODE      0x01
#define REG_FRF_MSB      0x06
#define REG_PA_CONFIG    0x09
#define REG_FIFO_TX_BASE 0x0E
#define REG_FIFO_RX_BASE 0x0F
#define REG_FIFO_ADDR    0x10
#define REG_FIFO_RX_CUR  0x10  /* same reg in standby */
```

```
#define REG_IRQ_FLAGS    0x12
#define REG_RX_NB_BYTES  0x13
#define REG_PKT_SNR      0x19
#define REG_PKT_RSSI     0x1A
#define REG_MODEM_CFG1   0x1D
#define REG_MODEM_CFG2   0x1E
#define REG_PREAMBLE_LSB 0x21
#define REG_PAYLOAD_LEN  0x22
#define REG_SYNC_WORD    0x39
#define REG_DIO_MAPPING1 0x40
#define REG_VERSION      0x42

#define OP_LONG_RANGE    0x80
#define OP_SLEEP         0x00
#define OP_STDBY         0x01
#define OP_TX            0x03
#define OP_RX_CONT       0x05

#define IRQ_TX_DONE      0x08
#define IRQ_RX_DONE      0x40

static int spi_fd;

static uint8_t reg_read(uint8_t addr) {
    uint8_t tx[2] = { addr & 0x7F, 0 }, rx[2];
    struct spi_ioc_transfer t = { .tx_buf=(unsigned long)tx, .rx_buf=(unsigned long)rx,
                                  .len=2, .speed_hz=1000000 };
    ioctl(spi_fd, SPI_IOC_MESSAGE(1), &t);
    return rx[1];
}

static void reg_write(uint8_t addr, uint8_t val) {
    uint8_t tx[2] = { addr | 0x80, val };
    struct spi_ioc_transfer t = { .tx_buf=(unsigned long)tx, .len=2, .speed_hz=1000000 };
    ioctl(spi_fd, SPI_IOC_MESSAGE(1), &t);
}

static void burst_read(uint8_t addr, uint8_t *buf, int n) {
    uint8_t cmd = addr & 0x7F;
    struct spi_ioc_transfer t[2] = {
        { .tx_buf=(unsigned long)&cmd, .len=1, .speed_hz=1000000 },
        { .rx_buf=(unsigned long)buf,  .len=n, .speed_hz=1000000 },
    };
    ioctl(spi_fd, SPI_IOC_MESSAGE(2), t);
}

static void burst_write(uint8_t addr, const uint8_t *buf, int n) {
    uint8_t cmd = addr | 0x80;
    struct spi_ioc_transfer t[2] = {
        { .tx_buf=(unsigned long)&cmd, .len=1, .speed_hz=1000000 },
        { .tx_buf=(unsigned long)buf,  .len=n, .speed_hz=1000000 },
    };
    ioctl(spi_fd, SPI_IOC_MESSAGE(2), t);
}

static void sx_reset(void) {
    /* assume RESET gpio handled via libgpiod or a sysfs poke — pulse low 1 ms */
    /* Platform GPIO reset adapter omitted: this sketch is not standalone. */
}
```

```
static void sx_init(uint32_t freq_hz, uint8_t sf, uint8_t bw_idx) {
    /* Sleep → switch to LoRa → standby */
    reg_write(REG_OP_MODE, OP_LONG_RANGE | OP_SLEEP);
    reg_write(REG_OP_MODE, OP_LONG_RANGE | OP_STDBY);

    /* Frequency: Frf = freq * 2^19 / 32e6 */
    uint64_t frf = ((uint64_t)freq_hz << 19) / 32000000;
    reg_write(REG_FRF_MSB,     (frf >> 16) & 0xFF);
    reg_write(REG_FRF_MSB + 1, (frf >>  8) & 0xFF);
    reg_write(REG_FRF_MSB + 2,  frf        & 0xFF);

    /* PA_BOOST, output power = 17 dBm (PaSelect=1, OutputPower=14, MaxPower=7) */
    reg_write(REG_PA_CONFIG, 0x80 | 0x70 | 0x0E);

    /* ModemConfig1: BW (bw_idx<<4) | CR=4/5 (001<<1) | explicit header (0) */
    reg_write(REG_MODEM_CFG1, (bw_idx << 4) | (0x01 << 1) | 0x00);

    /* ModemConfig2: SF (sf<<4) | CRC on (0x04) */
    reg_write(REG_MODEM_CFG2, (sf << 4) | 0x04);

    /* Preamble 8 symbols */
    reg_write(REG_PREAMBLE_LSB, 8);

    /* Private sync word (0x12); 0x34 reserved for LoRaWAN */
    reg_write(REG_SYNC_WORD, 0x12);

    /* FIFO bases: TX=0x00, RX=0x00 (we never use both at once) */
    reg_write(REG_FIFO_TX_BASE, 0x00);
    reg_write(REG_FIFO_RX_BASE, 0x00);
}

static void sx_send(const uint8_t *data, uint8_t len) {
    reg_write(REG_OP_MODE, OP_LONG_RANGE | OP_STDBY);
    reg_write(REG_FIFO_ADDR, 0x00);
    burst_write(REG_FIFO, data, len);
    reg_write(REG_PAYLOAD_LEN, len);
    reg_write(REG_DIO_MAPPING1, 0x40);  /* DIO0 = TxDone */
    reg_write(REG_OP_MODE, OP_LONG_RANGE | OP_TX);

    /* Poll IRQ flag — production would use a GPIO IRQ on DIO0 */
    while ((reg_read(REG_IRQ_FLAGS) & IRQ_TX_DONE) == 0) usleep(1000);
    reg_write(REG_IRQ_FLAGS, 0xFF);  /* clear all flags */
    reg_write(REG_OP_MODE, OP_LONG_RANGE | OP_STDBY);
}

static int sx_recv(uint8_t *buf, uint8_t maxlen, int8_t *snr, int16_t *rssi) {
    reg_write(REG_DIO_MAPPING1, 0x00);  /* DIO0 = RxDone */
    reg_write(REG_FIFO_ADDR, 0x00);
    reg_write(REG_OP_MODE, OP_LONG_RANGE | OP_RX_CONT);

    while ((reg_read(REG_IRQ_FLAGS) & IRQ_RX_DONE) == 0) usleep(10000);

    uint8_t n = reg_read(REG_RX_NB_BYTES);
    if (n > maxlen) n = maxlen;

    uint8_t rx_cur = reg_read(0x10);          /* FifoRxCurrentAddr */
    reg_write(REG_FIFO_ADDR, rx_cur);
    burst_read(REG_FIFO, buf, n);
```

```
    *snr  = ((int8_t)reg_read(REG_PKT_SNR)) / 4;
    *rssi = -157 + reg_read(REG_PKT_RSSI);    /* HF port; subtract 164 for LF */

    reg_write(REG_IRQ_FLAGS, 0xFF);
    return n;
}

int main(int argc, char **argv) {
    spi_fd = open("/dev/spidev0.0", O_RDWR);
    uint8_t mode = SPI_MODE_0;
    ioctl(spi_fd, SPI_IOC_WR_MODE, &mode);

    sx_reset();
    uint8_t ver = reg_read(REG_VERSION);
    if (ver != 0x12) { fprintf(stderr, "SX1276 not found (got 0x%02x)\n", ver); return
1; }
    printf("SX1276 detected\n");

    sx_init(868100000, /*SF*/ 7, /*BW=125 kHz idx=*/ 0x07);

    if (argc > 1 && argv[1][0] == 't') {
        for (int i = 0;; i++) {
            char msg[32];
            int n = snprintf(msg, sizeof msg, "hello %d", i);
            sx_send((uint8_t *)msg, n);
            printf("TX %s\n", msg);
            sleep(2);
        }
    } else {
        for (;;) {
            uint8_t buf[256]; int8_t snr; int16_t rssi;
            int n = sx_recv(buf, sizeof buf, &snr, &rssi);
            printf("RX %.*s  RSSI=%d dBm  SNR=%d dB\n", n, buf, rssi, snr);
        }
    }
}
```

What this driver shows that the framework hides:

- **Frequency programming.** The `Frf = freq × 2^19 / 32e6` formula appears nowhere in `lora-net`, it's hidden by `set_frequency()`. Here you see the raw three-byte divider write.
- **SPI access pattern.** A read is `addr & 0x7F`. A write is `addr | 0x80`. Burst FIFO read/write are sequential reads after one address byte.
- **FIFO is a 256-byte chip RAM with a pointer.** Set `FifoAddrPtr` to where you want to read/write. Then read/write the FIFO register repeatedly. Easy to get wrong, `FifoTxBaseAddr` is "where TX should start," `FifoAddrPtr` is "where the next byte goes/comes from."
- **IRQ flags are level-and-cleared.** Until you write back to `REG_IRQ_FLAGS`, the next operation sees a stale flag and you think the prior TX is still in progress.

- **RSSI math is asymmetric.** HF (above 525 MHz) and LF (below) subtract different offsets. Forget the offset, your “RSSI” is meaningless.
- **The whole driver is 200 lines.** With a real GPIO IRQ on DIO0 (`libgpiod`’s `gpiod_line_event_wait`) it becomes interrupt-driven. With a small framing layer (length + CRC + sequence) it becomes a usable PHY for point-to-point telemetry.

> **MCU bridge:** Think of Linux GPIO like the same pin set/reset block you used on STM32, but accessed through a kernel subsystem that owns numbering, direction, interrupts, and user-space exposure. **GPIO:** General-Purpose Input/Output, a pin controlled as a digital input, output, or interrupt source.

When you read the mainline-candidate `sx127x` driver later, you’ll recognize *exactly* this sequence inside `sx127x_tx_pkt()` and `sx127x_rx_handler()`, wrapped in a netdev shell. The framework adds netdev plumbing, threaded IRQs, and skb-based queueing, but the radio dance is the same twelve register writes.

Translating to SX1262 (the chip you should use for new designs)

The same physical actions become **opcodes**:

SX1276 register write	SX1262 command
`WriteReg(OpMode, STDBY)`	`SetStandby(STDBY_RC)`
`WriteReg(Frf*, ...)`	`SetRfFrequency(freq)`
`WriteReg(PaConfig, ...)`	`SetPaConfig(...)` + `SetTxParams(power, ramp)`
`WriteReg(ModemConfig*, ...)`	`SetModulationParams(SF, BW, CR, LDRO)`
`WriteReg(PayloadLength, n)`	`SetPacketParams(preamble, hdr_type, len, crc, iq)`
`WriteReg(FIFO, ...)` (burst)	`WriteBuffer(offset, data, n)`
`WriteReg(OpMode, TX)`	`SetTx(timeout)`
poll `IrqFlags`	`GetIrqStatus()` + `ClearIrqStatus()`

Same dozen actions, different SPI shape. The Semtech “SX126x driver” reference implementation in their open-source repo is a clean read. Port it from there.

98.7 Device tree, wiring spidev to the radio

> **Template warning:** This block contains placeholder values. Replace compatible strings, GPIO numbers, addresses, and paths with values from your board before using it.

For the user-space approach, all you need is `spidev` on the right CS:

```
&ecspi3 {
    pinctrl-names = "default";
    pinctrl-0 = <&pinctrl_ecspi3>;
    cs-gpios = <&gpio1 20 GPIO_ACTIVE_LOW>;
    status = "okay";

    lora@0 {
        compatible = "rohm,dh2228fv";        /* the universal spidev stand-in */
        reg = <0>;
        spi-max-frequency = <1000000>;
    };
};

&gpio1 {
    /* DIO0 (IRQ) and RESET are plain GPIOs accessed via libgpiod */
};
```

> *Production note: `rohm,dh2228fv` is a development-time spidev placeholder. Modern kernels print a warning when it appears in DT. See Chapter 47’s `spidev` warning for the proper DT overlay path or a real-chip compatible swap.* `/dev/spidev3.0` appears. `libgpiod` lines manage RESET and DIO0.

If you instead use the out-of-tree `sx127x-driver`:

```
lora@0 {
    compatible = "semtech,sx1276";
    reg = <0>;
    spi-max-frequency = <1000000>;
    reset-gpios = <&gpio1 21 GPIO_ACTIVE_LOW>;
    dio-gpios = <&gpio1 22 GPIO_ACTIVE_HIGH>,    /* DIO0 = TxDone/RxDone */
                <&gpio1 23 GPIO_ACTIVE_HIGH>;    /* DIO1 = RxTimeout */
    semtech,clock-frequency = <32000000>;
};
```

You then get `lora0` and (in some out-of-tree forks) a SOCK_DGRAM/SOCK_RAW socket interface. This path is *not* mainline. If you go here, pin your kernel version.

98.8 LoRaWAN vs LoRa P2P

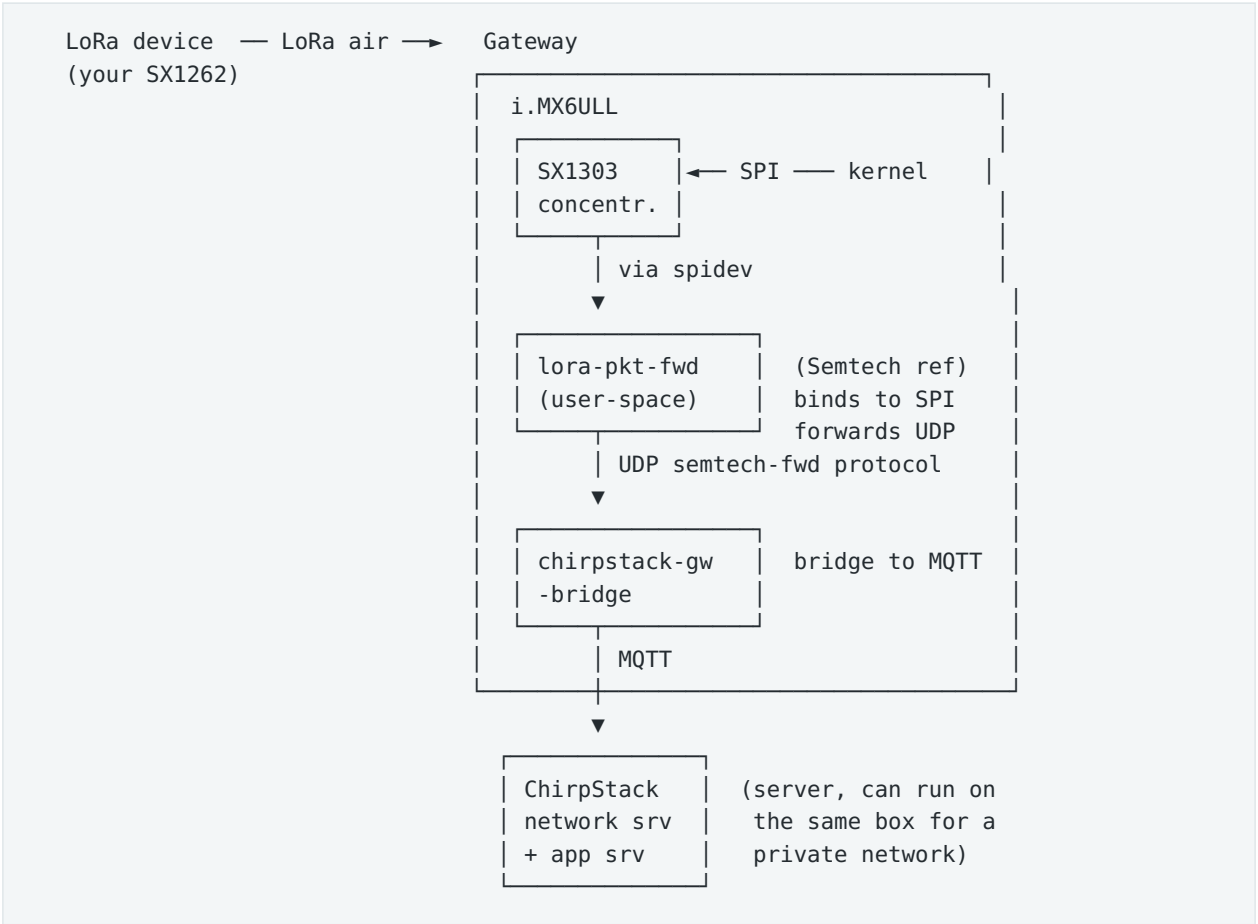
We've covered the **PHY** (LoRa modulation). The two product paths from here:

- **LoRa P2P**: your own framing on top of the PHY. Two boards talk directly. Best for a fleet of your own devices, no gateway, no infrastructure. The user-space driver above is enough.
- **LoRaWAN**: the *MAC* on top: device addressing (DevEUI/AppEUI/AppKey), OTAA join, three classes (A polling, B beacon, C continuous-RX), regional ISM channel plans, ADR (Adaptive Data Rate). Requires a **gateway** (multi-channel concentrator like SX1301/SX1303) + a **network server** (ChirpStack, The Things Stack) + an **application server**. Best for products joining a shared network (The Things Network, Helium, private deployments).

A LoRaWAN gateway on the i.MX6ULL

The gateway is a Linux box with a **concentrator card** (RAK2287/SX1303 SPI module, or a USB IC-880A). It listens to 8 channels in parallel, the SX1301/SX1303 is essentially 8 LoRa modems in one die, and forwards every received packet to a network server.

Stack:



Bring-up:

```
# Concentrator (SX1303) on SPI
echo dtoverlay=sx1303-spi >> /boot/config.txt    # or your DT equivalent
# Build sx1302_hal (Semtech's reference; works for sx1303 too)
git clone https://github.com/Lora-net/sx1302_hal && cd sx1302_hal
make
# Edit global_conf.json: region (EU868/US915), SPI device, channel plan
./packet_forwarder/lora_pkt_fwd
# In parallel:
docker run -p 1700:1700/udp chirpstack/chirpstack-gateway-bridge
docker run chirpstack/chirpstack
```

End-to-end, an SX1262 node running OTAA-joined LoRaWAN will:

1. JoinRequest → uplinked over the air → received by 8-channel concentrator
2. → forwarded UDP to gateway-bridge
3. → bridged to MQTT
4. → received by ChirpStack network server, which validates AppKey, generates DevAddr, returns JoinAccept
5. → downlinked through gateway-bridge → packet-forwarder → over the air → node receives the JoinAccept
6. Node now sends uplinks. ChirpStack decrypts. Your application server (or an MQTT subscriber) consumes them

A private LoRaWAN network on one i.MX6ULL is realistic. The kernel is involved as the SPI driver and nothing else.

98.9 Key Notes

- LoRa is a modulation and packet-radio mechanism, while LoRaWAN adds network roles, joining, security and regional behavior. A point-to-point radio is not a multichannel gateway.
- Spreading factor, bandwidth, coding rate and payload length trade airtime against sensitivity and capacity. Longer range is not free throughput.
- The radio state machine, completion signals and host timeouts must agree. A long packet can be valid even when a generic short timeout expires.
- Antenna, supply and regional transmission requirements constrain the complete link. Example frequencies and power settings are not universal deployment permissions.

98.10 Pitfalls

- **Transmitting without an antenna.** Will destroy the PA within a few transmissions. The chip's "safe" RFO output is +14 dBm, even that wants a load. Always have a real antenna or a 50 Ω dummy load attached during TX.
- **Wrong PA output selected on SX1276.** `PA_BOOST` (RFO_HF/RFO_LF pins) vs `RFO` pin. The schematic must route exactly one to the antenna. Sending +20 dBm into the wrong pin is permanent damage.
- **Sync word collision with LoRaWAN.** Private LoRa networks must use sync 0x12. The LoRaWAN value 0x34 is reserved. Using it in P2P will make every LoRaWAN gateway in earshot try to demodulate your packets and crash.
- **Bandwidth misnumbering.** The `BW` field is an *index* into a table, not the bandwidth in kHz. `0x07 = 125 kHz`, `0x08 = 250 kHz`, `0x09 = 500 kHz`. Putting `125` in the register sets ~3.9 MHz, out of spec, doesn't work.
- **SF12 timing.** A SF12/BW125 packet of 51 bytes takes ~2.3 seconds. Your `while(... & TX_DONE)` poll loop must allow for it. Many sample drivers time out at 1 s. Use 5+ seconds or compute from the air-time formula.
- **Duty cycle violations.** EU 868 g1 sub-band is **1 %**. At SF12, that's two packets per minute, period. Build a duty-cycle tracker in firmware or you'll be illegally transmitting and the regulator can fine you.
- **TCXO not started.** SX1262 with TCXO: you must call `SetDIO3AsTCXOCtrl(voltage, delay)` before any frequency operation. Skip it and the radio's PLL drifts → packets demodulate with high error or not at all on SF11/SF12.

> **MCU bridge:** Think of a PLL like the clock multiplier setup you used on STM32, but with more clock roots, gates, and consumers that Linux later needs to describe.
> **PLL:** Phase-Locked Loop, a clock block that multiplies a reference clock to create faster clocks.

- **`Sleep` does not erase state on SX127x, but on SX1262 Cold Start it does.** SX127x in Sleep mode preserves register state and consumes ~200 nA. SX1262 in Cold Start (`SetSleep(0x00)`) loses configuration, you must reconfigure on wake. Check the datasheet.
- **Single-chip "RF switch destroyed" failure.** A common module fault: the chip transmits OK at first but receive sensitivity is –80 dBm instead of –123 dBm, the internal/external RF switch was killed during a TX-into-open. Detect by measuring RSSI of a known close transmitter. If it's >40 dB worse than spec, the switch is gone.

- **Out-of-tree driver kernel-version pinning.** `sx127x-driver` is community-maintained. Major kernel updates break it. If you go the kernel-netdev route, pin your kernel until you have time to forward-port.
- **Concentrator SPI clock too fast.** SX1303 SPI tops out at 8 MHz. Some carrier boards expose 50 MHz `ecspi`. Set `spi-max-frequency = <8000000>;` or it works in the lab and fails on customer hardware.
- **Channel plan mismatch.** US915 has 64 uplink channels in 8 sub-bands. EU868 has 3 mandatory + 5 optional. A node configured for EU and a gateway for US never talk. The mistake is invisible until you check both sides' `region` parameter.

98.11 Going deeper

- **Semtech SX127x datasheet** + **SX126x datasheet**, the canonical reference for register / command behavior.
- **`semtech-prh/sx127x-driver` and `lora-net/lora-modules`**: out-of-tree Linux drivers. Read the `sx127x.c` / `sx126x.c` source for "what a netdev wrap would look like."
- **`Lora-net/sx1302_hal`**: Semtech's reference packet-forwarder for SX1301/SX1302/SX1303 concentrators. This is what every commercial LoRaWAN gateway ships.
- **ChirpStack** (`chirpstack/chirpstack`), open-source LoRaWAN network + application server, designed to run on a small Linux box.
- **LoRa Alliance LoRaWAN Specification** (1.0.3 / 1.1), the MAC standard.
- **Semtech AN1200.22, LoRa Modulation Basics**: the formal CSS math (a more rigorous version of §98.2).
- **The Things Industries documentation**: practical channel-plan and ADR notes for production fleets.
- **Ch 99**: the *non*-LoRa sub-GHz alternatives (nRF24L01, CC1101) when you don't need range but you do need throughput.
- **Ch 95**: for comparison: BLE's GATT/HCI model vs LoRaWAN's MAC/PHY split.

> **End of Chapter 98, LoRa.** Group M starts here: long-range and specialty wireless where the kernel is "just" the SPI controller and the protocol stack lives in user space. The next two chapters cover the alternatives, proprietary sub-GHz FSK (nRF24L01 / CC1101) and the IEEE 802.15.4 family (ZigBee, Thread).

> Next chapter: **Chapter 99: Sub-GHz proprietary radios (nRF24L01, CC1101).** Same sub-GHz spectrum, FSK instead of CSS, much higher throughput, no LoRaWAN-style infrastructure.

Chapter 99: Sub-GHz proprietary

> **Driver choice:** Use the in-tree, maintained driver first. Use out-of-tree, spidev, or custom-driver paths only after you accept the kernel-version maintenance cost and document who owns updates.

> **What: non-LoRa short-range radios** for fleets that need higher throughput than LoRa or have no infrastructure: **Nordic nRF24L01+** (2.4 GHz, 250 kbps, 2 Mbps GFSK, the dominant cheap mesh radio), **TI CC1101** (sub-GHz 300–928 MHz multi-mode), **TI CC1200** (newer, lower phase noise, higher data rate, better Wi-SUN candidate). We dissect each chip's SPI command/register model, the **Enhanced ShockBurst** auto-ACK in nRF24, the CC1101 state machine, write a 200-line nRF24 P2P driver from scratch in user space, then wire DT for the existing kernel drivers where they exist.
>
> **Why:** LoRa (Ch 98) buys range with bandwidth. The opposite trade, sub-second latency, multi-kbps throughput, no MAC stack, no infrastructure, $1–4 BOM, is what nRF24L01 and CC1101 give you. Every consumer remote control, garage opener, weather station, and dozens-of-nodes IoT prototype uses one of these. The kernel has *some* support (`nrf24` is out-of-tree. CC1101 has an old in-tree driver), but as with LoRa, most production stacks live in user space against `spidev`. Understanding the chip's state machine + SPI command shape is everything. **MAC:** Media Access Control in networking and radio chapters. It is the layer that owns framing and medium access.
>
> **Focus: these chips are state machines driven by SPI commands. The behavior depends on the current state, not on individual register values**. NRF24's state machine has 7 states (Power-Down, Standby-I, Standby-II, RX, TX, etc.). CC1101's has 13. Almost every "the chip doesn't work" bug is the chip being in the wrong state, TX commanded from RX, FIFO read while still receiving, command issued before the previous one's settling time elapsed. Learn the state diagram and the rest is straightforward.

99.1 Choosing the chip

	nRF24L01+	CC1101	CC1200
Frequency	2.4 GHz ISM	300–348 / 387–464 / 779–928 MHz	164–192 / 274–320 / 410–480 / 820–960 MHz

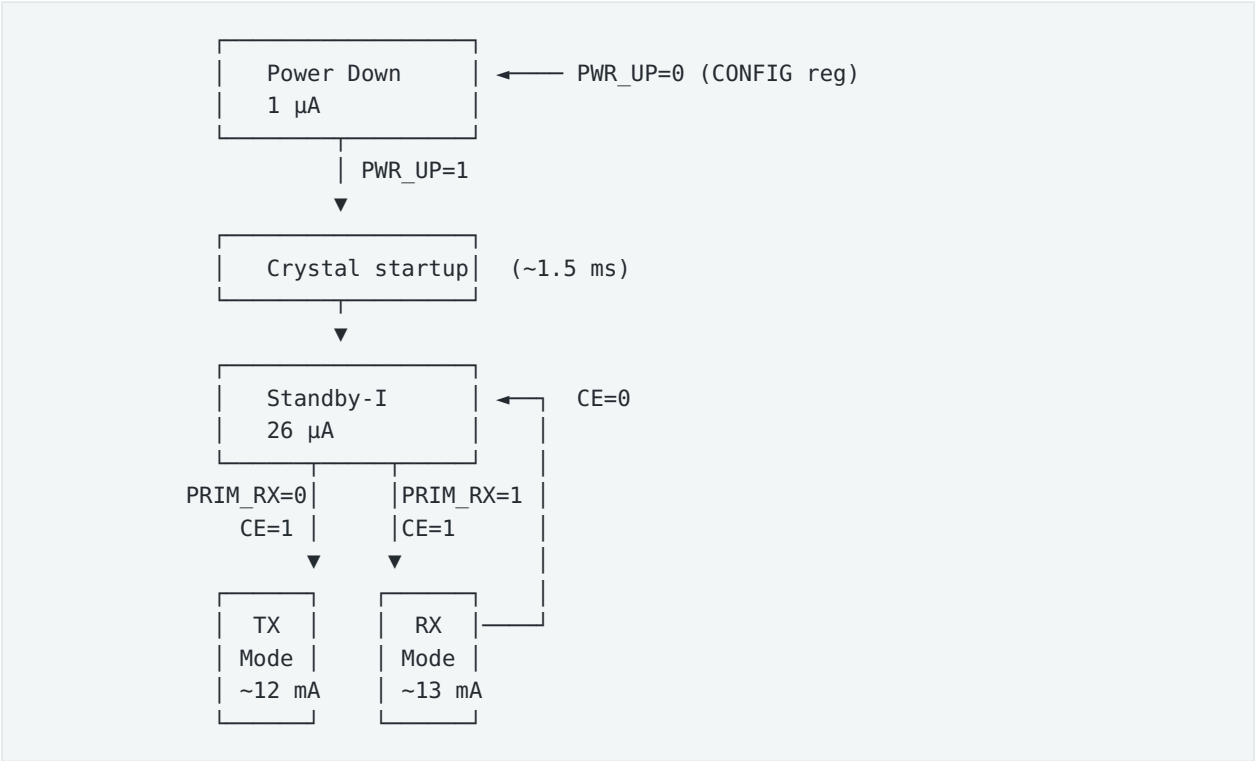
	nRF24L01+	CC1101	CC1200
Modulation	GFSK	OOK / 2-FSK / 4-FSK / GFSK / MSK	2-FSK / 4-FSK / OOK / GFSK / 4-GFSK / MSK
Data rate	250 kbps / 1 / 2 Mbps	0.6–500 kbps	0.6 kbps, 1.25 Mbps
Range (open, +0 dBm)	50–200 m	100–500 m	500 m, 1.5 km
Range (open, +20 dBm w/ PA)	1 km	1–2 km	2–5 km
RX current	12.6 mA	14.7 mA	17 mA
Sleep	900 nA	200 nA	120 nA
Address-aware?	yes (`Enhanced ShockBurst`)	no (raw FSK only)	no
Auto-ACK	yes (the killer feature)	no (must build in software)	no
Multi-receiver pipes	6 simultaneous PRX from 1 PTX	none	none
Cost	$0.80 (module)	$1.20	$3.50
Kernel driver	out-of-tree `nrf24`	in-tree `cc1101` (legacy, broken on modern kernels)	none
Use case	low-cost mesh, RC, hub-and-spoke	sub-GHz remotes, garage door, smart-home	longer-range industrial, Wi-SUN

Pick guide:

- **nRF24L01+**: when 2.4 GHz is acceptable, you have 1–32 nodes, and the auto-ACK + pipes feature saves you from building a MAC. Caveat: 2.4 GHz is crowded. WiFi will eat you in a busy office.
- **CC1101**: sub-GHz remote-control style, when 2.4 GHz coexistence is a problem, you can tolerate building your own ACK protocol, and BOM cost is critical. Most “smart” home AC remotes, weather stations, and security sensors use this.
- **CC1200**: the modern CC1101 successor. Pick this for new designs over CC1101 (better phase noise, easier registers, mainstream Wi-SUN support).

99.2 nRF24L01+, the dominant cheap radio

State machine



The two GPIO-style control pins:

> **MCU bridge:** Think of Linux GPIO like the same pin set/reset block you used on STM32, but accessed through a kernel subsystem that owns numbering, direction, interrupts, and user-space exposure. **GPIO:** General-Purpose Input/Output, a pin controlled as a digital input, output, or interrupt source.

- **CSN** (Chip Select Not), SPI CS, frames each command.

- **CE** (Chip Enable), controls active TX/RX. CE=0 means “go to Standby-I and hold there”. CE=1 means “start TX (if PRIM_RX=0) or stay in RX (if PRIM_RX=1).”

These transition rules catch every newcomer:

- To switch between TX and RX you must pass through Standby-I. Set `CE=0`, wait 130 µs, set `PRIM_RX`, then set `CE=1`.
- The TX FIFO is loaded *before* CE goes high. Pulsing CE for ≥10 µs triggers transmission of one packet.

SPI commands

The nRF24 has **commands** (not direct register writes):

Command	Hex	Meaning
`R_REGISTER addr`	`0x00 | addr`	read register addr (5 bits)
`W_REGISTER addr`	`0x20 | addr`	write register
`R_RX_PAYLOAD`	`0x61`	read RX FIFO (1–32 bytes)
`W_TX_PAYLOAD`	`0xA0`	write TX FIFO
`FLUSH_TX`	`0xE1`	empty TX FIFO
`FLUSH_RX`	`0xE2`	empty RX FIFO
`REUSE_TX_PL`	`0xE3`	resend last TX on next CE pulse
`R_RX_PL_WID`	`0x60`	get length of top RX FIFO entry (dynamic payloads)
`W_ACK_PAYLOAD pipe`	`0xA8 | pipe`	piggyback an ACK payload back to PTX
`W_TX_PAYLOAD_NOACK`	`0xB0`	TX with auto-ACK disabled for this frame

Command	Hex	Meaning
`NOP`	`0xFF`	read STATUS register only

Every SPI transaction returns the STATUS register in the first byte, useful “for free” status polling.

Key registers (5-bit addresses):

Addr	Name	Purpose
0x00	CONFIG	EN_CRC, CRCO, PWR_UP, PRIM_RX
0x01	EN_AA	per-pipe Enhanced ShockBurst auto-ACK enable
0x02	EN_RXADDR	per-pipe RX enable
0x03	SETUP_AW	address width (3/4/5 bytes)
0x04	SETUP_RETR	ARD (auto-retry delay), ARC (auto-retry count)
0x05	RF_CH	RF channel (0–125, freq = 2400 + channel MHz)
0x06	RF_SETUP	PA power (0/-6/-12/-18 dBm), data rate (1 / 2 / 250 kbps)
0x07	STATUS	RX_DR, TX_DS, MAX_RT, RX_P_NO, TX_FULL
0x08	OBSERVE_TX	retransmission + lost-packet counts
0x09	RPD (CD on non-plus)	received-power detector

Addr	Name	Purpose
0x0A–0x0F	RX_ADDR_P0..P5	per-pipe RX address (5 bytes for P0,P1; 1 byte for P2..P5)
0x10	TX_ADDR	TX address (must match PRX's RX_ADDR_P0 for auto-ACK)
0x11–0x16	RX_PW_P0..P5	static payload length per pipe (1–32)
0x17	FIFO_STATUS	TX_REUSE, TX_FULL, TX_EMPTY, RX_FULL, RX_EMPTY
0x1C	DYNPD	per-pipe dynamic-payload enable
0x1D	FEATURE	EN_DPL (dynamic payloads), EN_ACK_PAY, EN_DYN_ACK

Enhanced ShockBurst, the killer feature

ShockBurst is the auto-retransmit + auto-ACK layer built into the chip. Setup:

1. PTX writes a frame. Sets `EN_AA` on pipe 0 (or whichever).
2. PTX pulses CE. Chip transmits.
3. Chip *automatically* flips to RX, listens for an ACK (an empty frame on the same address).
4. If ACK received within ARD (`SETUP_RETR`), `TX_DS` IRQ, success.

> **MCU bridge:** Think of an IRQ like an EXTI/NVIC interrupt path, except Linux splits the hard interrupt from deferred work and must share lines across drivers. **IRQ:** interrupt request, the signal path that tells the CPU or interrupt controller that hardware needs service.

1. If not, retry up to ARC times. If still no ACK, `MAX_RT` IRQ, failure.

The PRX side automatically generates the ACK on receipt. The CPU doesn't run code for this round trip, it's all in chip silicon. Result: reliable point-to-point delivery at sub-millisecond latency, with zero MAC code on either side.

You can even piggyback an **ACK payload**: PRX's `W_ACK_PAYLOAD` queues a payload that rides back inside the ACK frame, giving 32-byte bidirectional half-duplex with one transmit + auto-ACK pair.

This is why nRF24 is the default choice when BOM cost matters and rates stay below 1 Mbps.

Six receive pipes, the multi-PRX, one-PTX pattern

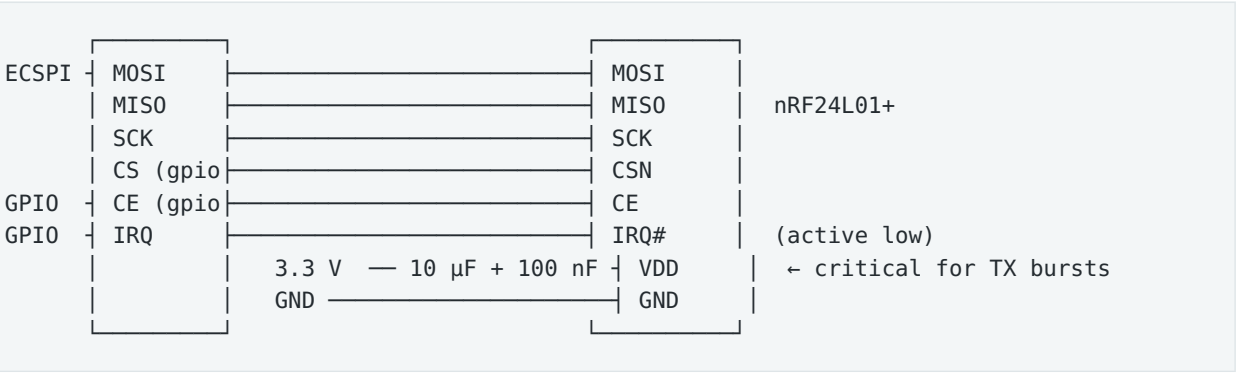
A single nRF24 in RX mode can listen for 6 different addresses simultaneously (pipes P0..P5). The canonical use:

```
Hub (one nRF24 in PRX mode)
├─ Pipe 0: 0xE7E7E7E7E7  (default broadcast)
├─ Pipe 1: 0xC2C2C2C2C2  (Node A)
├─ Pipe 2:           D3  (Node B — shares 4-byte prefix with P1)
├─ Pipe 3:           D4  (Node C)
├─ Pipe 4:           D5  (Node D)
└─ Pipe 5:           D6  (Node E)
```

Pipes 2–5 share the high 4 address bytes with pipe 1. Only the last byte differs. So you get one hub talking to 5 satellites, each individually addressed and auto-ACKed. Add ACK payloads and the hub can send commands back inside each ACK, a 6-node star network without a MAC layer.

99.3 Wiring an nRF24 to the i.MX6ULL

Six wires + power:



The IRQ pin is active-low. It asserts on `RX_DR`, `TX_DS`, or `MAX_RT`. Reading `STATUS` and writing 1 back to the bit clears the interrupt.

Power supply gotcha: the 1 Mbps TX burst draws ~12 mA peaks at the rail. Without the 10 µF bulk + 100 nF local decoupling, the supply droops and the next TX corrupts. This is the most common reason a newly bought module appears dead.

99.4 How the out-of-tree nRF24 kernel drivers actually work

There are two community kernel drivers:

- **`nrf24` by Marcin Ciupak** (out-of-tree, GitHub `nRF24/nRF24L01_plus_Linux_Driver`). Presents the chip as a char device per pipe (`/dev/nrf24-pipe0`…). Read = receive. Write = transmit.
- **`nrf24-network` patches by Andre Renaud** (one attempt to make it netdev. Never accepted upstream).

Walk of the char-device driver's TX path (paraphrased from `nrf24/nrf24.c`):

```
static ssize_t nrf24_write(struct file *f, const char __user *buf,
                           size_t len, loff_t *off) {
    struct nrf24_pipe *p = f->private_data;
    if (len > 32) return -EINVAL;
    if (copy_from_user(p->tx_buf, buf, len)) return -EFAULT;
    mutex_lock(&p->dev->tx_lock);
    nrf24_set_mode_tx(p->dev);                /* PRIM_RX=0; CE=0 (Standby-I) */
    nrf24_set_tx_addr(p->dev, p->cfg.address); /* TX_ADDR + RX_ADDR_P0 (for ACK) */
    nrf24_send_packet(p->dev, p->tx_buf, len); /* W_TX_PAYLOAD + pulse CE */
    wait_event_interruptible_timeout(
        p->dev->tx_done_wait,
        atomic_read(&p->dev->tx_status) != TX_PENDING, ARD * (ARC + 1));
    int status = atomic_read(&p->dev->tx_status);
    nrf24_set_mode_rx(p->dev);                /* PRIM_RX=1 again */
    mutex_unlock(&p->dev->tx_lock);
    return (status == TX_SUCCESS) ? len : -EIO;
}
```

The threaded IRQ handler:

```
static irqreturn_t nrf24_irq(int irq, void *data) {
    struct nrf24_device *d = data;
    uint8_t status = nrf24_cmd(d, NOP, NULL, 0, NULL, 0);
    if (status & STATUS_TX_DS)  atomic_set(&d->tx_status, TX_SUCCESS);
    if (status & STATUS_MAX_RT) atomic_set(&d->tx_status, TX_FAILED);
    if (status & STATUS_RX_DR) {
        uint8_t pipe = (status >> 1) & 0x07;
        uint8_t len;
        nrf24_cmd(d, R_RX_PL_WID, NULL, 0, &len, 1);
        nrf24_cmd(d, R_RX_PAYLOAD, NULL, 0, d->rx_buf, len);
        kfifo_in(&d->pipes[pipe].rx_fifo, d->rx_buf, len);
        wake_up_interruptible(&d->pipes[pipe].rx_wait);
    }
    nrf24_cmd(d, W_REGISTER | STATUS, &status, 1, NULL, 0);  /* clear flags */
    wake_up_interruptible(&d->tx_done_wait);
    return IRQ_HANDLED;
}
```

The architecture: one chip, six per-pipe char devices, one shared TX lock (because the chip can only TX to one address at a time), one IRQ thread, a kfifo per pipe holding received bytes. Read() blocks on the kfifo. Write() pumps the TX path.

That’s a lot of machinery to handle six addresses on one chip. The from-scratch version below skips it and uses `spidev` + libgpiod directly.

99.5 From scratch, a user-space nRF24 driver in C

nrf24_min.c:

```
/* Minimal user-space nRF24L01+ driver. PTX or PRX role at startup.
 * Pair: board A as PTX (sends "ping %d"); board B as PRX (prints with RSSI).
 */
#include <fcntl.h>
#include <stdint.h>
#include <stdio.h>
#include <string.h>
#include <unistd.h>
#include <sys/ioctl.h>
#include <linux/spi/spidev.h>
#include <gpiod.h>

#define CMD_R_REG     0x00
#define CMD_W_REG     0x20
#define CMD_R_RX_PL   0x61
#define CMD_W_TX_PL   0xA0
#define CMD_FLUSH_TX  0xE1
#define CMD_FLUSH_RX  0xE2
#define CMD_NOP       0xFF

#define REG_CONFIG    0x00
#define REG_EN_AA     0x01
#define REG_EN_RXADDR 0x02
#define REG_SETUP_AW  0x03
#define REG_SETUP_RETR 0x04
#define REG_RF_CH     0x05
#define REG_RF_SETUP  0x06
#define REG_STATUS    0x07
#define REG_RX_ADDR_P0 0x0A
#define REG_TX_ADDR   0x10
#define REG_RX_PW_P0  0x11
#define REG_DYNPD     0x1C
#define REG_FEATURE   0x1D

static int spi_fd;
static struct gpiod_line *ce_line, *irq_line;

static uint8_t cmd(uint8_t opcode, const uint8_t *tx, uint8_t *rx, int n) {
    uint8_t buf_tx[33], buf_rx[33];
    buf_tx[0] = opcode;
    if (tx) memcpy(&buf_tx[1], tx, n);
    else    memset(&buf_tx[1], 0, n);
    struct spi_ioc_transfer t = {
        .tx_buf=(unsigned long)buf_tx, .rx_buf=(unsigned long)buf_rx,
        .len=n+1, .speed_hz=4000000,
```

```
    };
    ioctl(spi_fd, SPI_IOC_MESSAGE(1), &t);
    if (rx) memcpy(rx, &buf_rx[1], n);
    return buf_rx[0];  /* STATUS comes back in the first byte, free */
}

static uint8_t reg_read(uint8_t a) {
    uint8_t v;
    cmd(CMD_R_REG | (a & 0x1F), NULL, &v, 1);
    return v;
}
static void reg_write(uint8_t a, uint8_t v) {
    cmd(CMD_W_REG | (a & 0x1F), &v, NULL, 1);
}
static void reg_write_n(uint8_t a, const uint8_t *v, int n) {
    cmd(CMD_W_REG | (a & 0x1F), v, NULL, n);
}

static void ce(int on) { gpiod_line_set_value(ce_line, on); }

static void nrf_init(int as_prx) {
    /* Reset config: CRC=2bytes, PWR_UP=1, PRIM_RX as requested */
    reg_write(REG_CONFIG, 0x0C | 0x02 | (as_prx ? 0x01 : 0x00));
    usleep(2000);                           /* crystal settling */

    reg_write(REG_SETUP_AW, 0x03);          /* 5-byte addresses */
    reg_write(REG_RF_CH, 76);               /* 2476 MHz — away from WiFi ch1/6/11 */
    reg_write(REG_RF_SETUP, 0x06);          /* 1 Mbps, 0 dBm */
    reg_write(REG_EN_AA, 0x01);             /* auto-ACK on pipe 0 */
    reg_write(REG_EN_RXADDR, 0x01);         /* RX on pipe 0 */
    reg_write(REG_SETUP_RETR, 0x3F);        /* ARD=1000us, ARC=15 retries */

    uint8_t addr[5] = { 0xE7, 0xE7, 0xE7, 0xE7, 0xE7 };
    reg_write_n(REG_RX_ADDR_P0, addr, 5);
    reg_write_n(REG_TX_ADDR,    addr, 5);
    reg_write(REG_RX_PW_P0, 32);            /* static 32-byte payload */

    cmd(CMD_FLUSH_TX, NULL, NULL, 0);
    cmd(CMD_FLUSH_RX, NULL, NULL, 0);
    reg_write(REG_STATUS, 0x70);            /* clear RX_DR/TX_DS/MAX_RT */
}

static int nrf_send(const uint8_t *data, int len) {
    ce(0);
    cmd(CMD_W_TX_PL, data, NULL, len);
    ce(1);
    usleep(15);                             /* >10 µs trigger pulse */
    ce(0);
    /* poll status until TX_DS or MAX_RT */
    for (int i = 0; i < 100; i++) {
        uint8_t st = cmd(CMD_NOP, NULL, NULL, 0);
        if (st & 0x20) { reg_write(REG_STATUS, 0x20); return 0; }   /* TX_DS */
        if (st & 0x10) {                                            /* MAX_RT */
            reg_write(REG_STATUS, 0x10);
            cmd(CMD_FLUSH_TX, NULL, NULL, 0);
            return -1;
        }
        usleep(1000);
    }
```

```
    return -2;
}

static int nrf_recv(uint8_t *buf, int maxlen) {
    ce(1);  /* enter RX */
    for (;;) {
        uint8_t st = cmd(CMD_NOP, NULL, NULL, 0);
        if (st & 0x40) {                                        /* RX_DR */
            int len = (maxlen > 32) ? 32 : maxlen;
            cmd(CMD_R_RX_PL, NULL, buf, len);
            reg_write(REG_STATUS, 0x40);
            ce(0);
            return len;
        }
        usleep(1000);
    }
}

int main(int argc, char **argv) {
    spi_fd = open("/dev/spidev0.0", O_RDWR);
    uint8_t mode = SPI_MODE_0;
    ioctl(spi_fd, SPI_IOC_WR_MODE, &mode);

    struct gpiod_chip *chip = gpiod_chip_open("/dev/gpiochip0");
    ce_line  = gpiod_chip_get_line(chip, 20);
    irq_line = gpiod_chip_get_line(chip, 21);
    gpiod_line_request_output(ce_line, "nrf24-ce", 0);
    gpiod_line_request_input(irq_line, "nrf24-irq");

    int as_prx = (argc > 1 && argv[1][0] == 'r');
    nrf_init(as_prx);

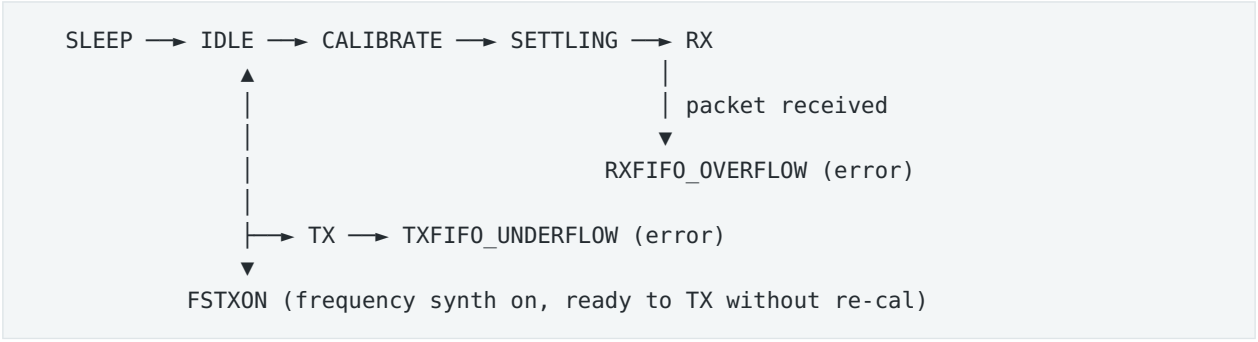
    if (as_prx) {
        printf("Listening...\n");
        for (;;) {
            uint8_t buf[32];
            int n = nrf_recv(buf, 32);
            printf("RX: %.*s\n", n, buf);
        }
    } else {
        for (int i = 0; ; i++) {
            char msg[32];
            int n = snprintf(msg, 32, "ping %d", i);
            int r = nrf_send((uint8_t *)msg, n);
            printf("TX %s [%s]\n", msg, r == 0 ? "ACK" : "no-ACK");
            sleep(1);
        }
    }
}
```

Two boards, same binary, one with the `r` argument, one without, and you have a 1 Mbps ACKed point-to-point link. Add `OBSERVE_TX` reading to see retry counts. Add multi-pipe addresses to make a hub.

99.6 CC1101, sub-GHz state-machine radio

CC1101 is harder to use than nRF24L01 because TI gives you the modulator/demodulator + framing primitives but **no auto-ACK**. You build the MAC.

State machine (13 states)



Command strobes (single-byte SPI):

- `SRES` (0x30) reset
- `SCAL` (0x33) calibrate frequency synthesizer
- `SRX` (0x34) enter RX
- `STX` (0x35) enter TX
- `SIDLE` (0x36) leave RX/TX → IDLE
- `SFRX` (0x3A) flush RX FIFO
- `SFTX` (0x3B) flush TX FIFO

Configuration is ~40 registers (IOCFG2/1/0, FIFOTHR, SYNC1/0, PKTLEN, PKTCTRL1/0, ADDR, CHANNR, FSCTRL1/0, FREQ2/1/0, MDMCFG4/3/2/1/0, DEVIATN, MCSM2/1/0, FOCCFG, BSCFG, AGCCTRL2/1/0, …). TI's **SmartRF Studio** generates the register dump for any modulation, data rate, and deviation. Almost everyone uses it for CC1101 bring-up.

Once configured, registers persist across IDLE ↔ RX/TX transitions. Strobing `SRX` or `STX` does not reset them. You can rewrite them at any time. So you configure at startup and then just strobe `SRX` / `STX` to flip mode.

Walk of the kernel CC1101 driver

The in-tree `drivers/net/wireless/ti/cc1101.c` (an out-of-tree fork, never merged) follows a pattern similar to nRF24: SPI device, GPIO IRQ (GDO0 = pkt_rx, GDO2 = chip_ready), char device. The full driver is ~1500 lines because it implements packet framing + length-byte vs fixed-length modes + address filtering, features the chip supports in hardware but the driver must enable per use case.

> *Note: the in-tree wireless directory does not contain this driver, verify against your current kernel before assuming it is present.*

A good CC1101 starting point is **`elechouse/CC1101` on GitHub** (Arduino-derived but C99), clean code that walks register init → packet send → receive. Port that to user space on `spidev` + libgpiod and you have a working sub-GHz radio in ~300 lines.

A 30-line user-space CC1101 send

```
/* Skeleton — assumes you have generated reg_table[] via SmartRF Studio. */
static const uint8_t reg_table[][2] = {
    {0x00, 0x29}, /* IOCFG2: GDO2 = chip-ready */
    {0x02, 0x06}, /* IOCFG0: GDO0 = pkt-received */
    {0x07, 0xFF}, /* PKTCTRL1: address filter + status appended */
    {0x08, 0x05}, /* PKTCTRL0: variable length, CRC, whitening */
    {0x0A, 0xAA}, /* ADDR (our address) */
    /* ... 40 lines from SmartRF Studio ... */
    {0xFF, 0xFF}, /* end */
};

static void cc1101_init(void) {
    strobe(SRES); usleep(1000);
    for (int i = 0; reg_table[i][0] != 0xFF; i++)
        reg_write(reg_table[i][0], reg_table[i][1]);
    strobe(SCAL);
}

static void cc1101_send(uint8_t dst, const uint8_t *data, int n) {
    strobe(SIDLE);
    strobe(SFTX);
    uint8_t hdr[2] = { n + 1, dst };          /* length, destination */
    burst_write(0x3F | 0x40, hdr, 2);         /* 0x3F = TX FIFO */
    burst_write(0x3F | 0x40, data, n);
    strobe(STX);
    while ((reg_read(0x35) & 0x70) != 0)      /* MARCSTATE != IDLE */
        usleep(100);
}
```

CC1200 is essentially CC1101 with a cleaner register set and higher symbol rates, the same template fits.

99.7 Device tree examples

> **Template warning:** This block contains placeholder values. Replace compatible strings, GPIO numbers, addresses, and paths with values from your board before using it.

For the user-space approach (`spidev` + libgpiod), the DT is just a `spidev` slot:

```
&ecspi3 {
    cs-gpios = <&gpio4 26 GPIO_ACTIVE_LOW>;
    status = "okay";

    nrf24@0 {
        compatible = "rohm,dh2228fv";   /* spidev stand-in */
        reg = <0>;
        spi-max-frequency = <8000000>;  /* nRF24 max 10 MHz */
    };
};

&gpio4 {
    /* CE, IRQ — accessed via libgpiod */
};
```

> *Production note: `rohm,dh2228fv` is a development-time spidev placeholder. Modern kernels print a warning when it appears in DT. See Chapter 47's `spidev` warning for the proper DT overlay path or a real-chip compatible swap.* For the `nrf24` out-of-tree driver:

```
nrf24@0 {
    compatible = "nordic,nrf24l01p";
    reg = <0>;
    spi-max-frequency = <8000000>;
    ce-gpios = <&gpio4 27 GPIO_ACTIVE_HIGH>;
    interrupts-extended = <&gpio4 28 IRQ_TYPE_EDGE_FALLING>;
};
```

99.8 When to pick CC1101 vs nRF24L01

Scenario	Pick
1-PTX, ≤6 PRX, BOM critical, 2.4 GHz OK	**nRF24L01+** (auto-ACK + pipes do the MAC)
Sub-GHz, no MAC needed (each device transmits + I'll dedupe in app)	**CC1101**
Coexistence with WiFi a worry	**CC1101** (sub-GHz, less crowded)
Need 1+ km range without LoRa	CC1101 + PA + LNA module (E07 series) or **CC1200**

Scenario	Pick
Need range *and* multi-node + auto-ACK	LoRa P2P (Ch 98), neither nRF24 nor CC1101 gives both
802.15.4 / ZigBee / Thread	not these, Ch 100

99.9 Key Notes

- These radios implement different state machines and packet services. The nRF24L01+ is a 2.4 GHz device, while CC1101 represents the sub-GHz path.
- Hardware auto-acknowledgment and retries reduce host work but do not provide application-level exactly-once delivery or security.
- FIFO ownership, standby transitions and completion status govern reliable transactions. A register write is meaningful only in the appropriate device state.
- Link budget, channel contention and energy per delivered message matter more than nominal data rate alone.

99.10 Pitfalls

- **nRF24 vs nRF24+ clones.** The “+” variant added 250 kbps data rate and is what most modules sell. Some Chinese clones (NRF24L01 RFM73 etc.) drop quirks. Verify with `CONFIG` default = 0x08 and the auto-ACK setting.
- **Bad VDD decoupling.** Without 10 µF bulk + 100 nF, TX bursts droop the rail, all packets MAX_RT fail. The single most common nRF24 problem.
- **CE pulse too short.** ≥10 µs is the spec. On a slow CPU + sysfs GPIO you’ll comfortably exceed that. On libgpiod with cached lines or an FPGA-fast GPIO it can be 1 µs and the chip never enters TX.

> **sysfs:** a kernel-generated filesystem under /sys that exposes devices, drivers, and attributes.

- **TX_ADDR ≠ RX_ADDR_P0.** The PTX’s `RX_ADDR_P0` must equal its own `TX_ADDR` for auto-ACK to work (the ACK comes back to P0). Easy to overlook.
- **TX-then-RX without going through Standby-I.** Always `CE=0`, change `PRIM_RX`, `CE=1`. Going directly puts the chip in an undefined state.

- **CC1101 register dump from wrong frequency band.** The register table for 433 MHz won't work at 868. SmartRF Studio regenerates entirely. Don't copy/paste blindly between band setups.
- **CC1101 manual calibration needed periodically.** The frequency synth needs `SCAL` after temperature changes or long sleeps. The `MCSM*` registers can be set to auto-calibrate, but watch the timing, `MCSM0 = 0x18` means "calibrate every 4th transition from IDLE to RX/TX."
- **2.4 GHz crowded.** Every WiFi, BT, microwave oven, baby monitor uses 2.4 GHz. Sub-GHz (CC1101 433/868) is 10× quieter. Test in realistic environment, not on an isolated bench.
- **Antenna critical at sub-GHz.** A quarter-wave at 868 MHz is 86 mm. A 25 mm "stub" antenna costs you 10+ dB. Use a proper helical or PCB-trace antenna designed for the band.
- **No FCC certification path with home-made matching network.** Buying a pre-certified module (E07, RAK, etc.) saves a $20k+ FCC test cycle if you're shipping a product.

99.11 Going deeper

- **Nordic nRF24L01+ Product Specification v1.0**: the canonical reference, all register and state semantics.
- **TI CC1101 Datasheet** + **TI Design Note DN500** (CC1101 in C). And **SmartRF Studio** for register-table generation.
- **TI CC1200 Datasheet** + Wi-SUN reference designs.
- **`nRF24/nRF24L01_plus_Linux_Driver`** (out-of-tree GitHub), char-device driver, useful for "what does a kernel driver for this look like."
- **`elechouse/CC1101`**: clean reference C library.
- **`tmrh20/RF24` / `RF24Network` / `RF24Mesh`**: Arduino-world but readable. The RF24Mesh layer shows how to build a multi-hop MAC over nRF24's PHY+ACK.

> **PHY:** physical-layer block or chip that converts digital MAC signals to electrical or radio signals.

- **Ch 98** for the LoRa-vs-FSK trade-off.
- **Ch 100** for IEEE 802.15.4 (Thread/ZigBee) as the certified-network alternative.

> Next chapter: **Chapter 100: ZigBee / Thread / 802.15.4**, the certified mesh-network alternatives to nRF24 ad-hoc.

Chapter 100: ZigBee / Thread / 802.15.4

> **Privilege boundary:** $ means normal user. # or sudo means root and can change host or target state. After a privileged command, verify the expected device, service, or file appears before continuing. Roll back by undoing the config change or stopping the service you just enabled.

> **What:** the **IEEE 802.15.4** family. It is the certified mesh networking stack that powers most retail smart-home meshes: Philips Hue, Aqara, Eve, IKEA Trådfri, Google Nest. We compare **TI CC2530** (legacy ZigBee 3.0), **Nordic nRF52840** (modern, OpenThread + ZigBee + BLE in one chip), and **Silicon Labs EFR32MG** (commercial-grade ZigBee/Thread). On Linux, the i.MX6ULL is the **gateway** (running zigbee2mqtt, Thread Border Router, or Home Assistant), not a node. The radio is on a coprocessor module. Linux talks to it over UART/USB as a **ZNP** (ZigBee Network Processor) or **NCP** (Network Coprocessor).
>
> **Why:** 802.15.4 is the only mesh radio with serious certification, vendor cross-compat, and consumer-product penetration. If you're building a *gateway* (smart-home hub, factory data collector, gateway-as-a-service), you're integrating with 802.15.4 modules. You do not write the nodes. You buy them. The Linux skill is **gateway integration**: pairing the radio coprocessor, serial framing of the host-controller protocol, MQTT bridging, OTA upgrade management.
>
> **Focus: the radio firmware is a black box. You talk to it over a serial protocol (EZSP, Thread Spinel, ZNP) that mirrors the network layer**. Just like BLE HCI (Ch 95), the host-controller boundary is what you debug. Once the gateway daemon (zigbee2mqtt, OpenThread Border Router) is up, MQTT/MDNS handles the rest. No kernel driver to write, `ttyUSB`/`spidev` is the chip-side interface and a user-space daemon is the brain.
>
> **Tooling.** This chapter uses Node.js 18+, `zigbee2mqtt` (via npm), Mosquitto broker. For Thread, build `openthread/ot-br-posix` from source.
>
> - **Ubuntu-base (target):** `apt install nodejs npm mosquitto mosquitto-clients`
> - **Buildroot:** `BR2_PACKAGE_NODEJS=y BR2_PACKAGE_MOSQUITTO=y  # otbr typically self-built` **Buildroot:** a configuration-driven build system that produces a complete root filesystem and related images.

> - Full per-tool reference: Userspace tooling appendix. **MCU bridge:** Think of the rootfs as the firmware image's file-backed runtime environment. On an MCU you link everything into flash. On Linux, programs and config live in this mounted tree. **rootfs:** root filesystem, the directory tree mounted at / that contains /bin, /etc, /dev, and libraries.

100.1 The three protocols at a glance

	ZigBee 3.0	Thread	802.15.4 raw
PHY	802.15.4 (2.4 GHz)	802.15.4 (2.4 GHz)	802.15.4
MAC	ZigBee	IPv6/6LoWPAN	raw
Network	mesh, ZigBee	mesh, IPv6 routers	none (you build it)
Addressing	16-bit short + 64-bit IEEE	IPv6 (linklocal + mesh-local + global)	64-bit IEEE
Ecosystem	Philips Hue, Aqara, IKEA, Sengled	Apple HomeKit, Google Nest, Matter	DIY mesh, RPL
Linux gateway	zigbee2mqtt + Z2M	OpenThread Border Router (otbr)	custom
Certification path	ZigBee Alliance	Thread Group	none
Best for	retail smart-home product	next-gen smart home (Matter)	research / one-off

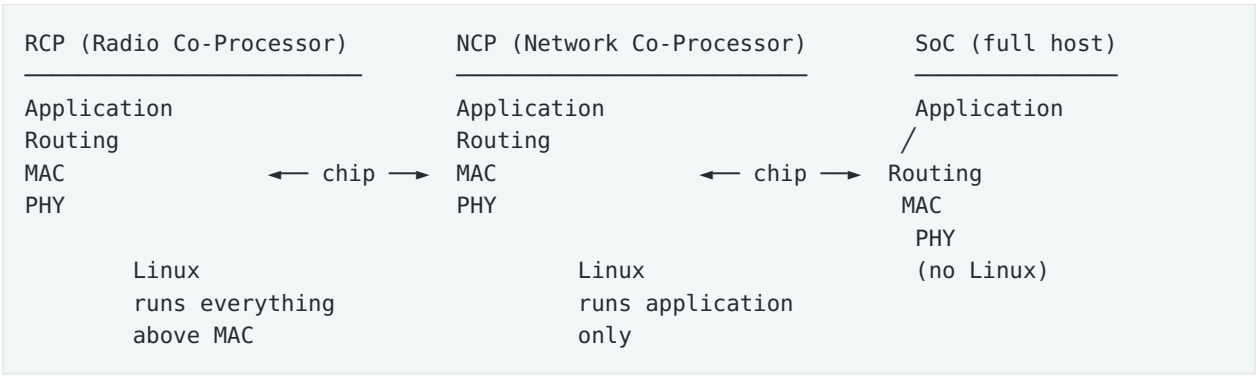
The big shift: **Matter** (the new consumer smart-home standard) runs *over* Thread (or WiFi). So new gateway designs target **Thread** with the **OpenThread Border Router** stack, while still supporting legacy ZigBee for installed-base reasons. Hence: serious products today bring up both stacks on the same coprocessor (nRF52840 can do both).

100.2 Why the radio lives on a coprocessor

A 802.15.4 PHY is timing-strict: ack-on-receive is required within 1 ms, channel hopping happens at sub-ms boundaries, and the MAC retransmit logic must run reliably. Between Linux's scheduling jitter and SPI/USB latency, running the MAC on the host CPU is not reliable.

> **MAC:** Media Access Control in networking and radio chapters. It is the layer that owns framing and medium access. **PHY:** physical-layer block or chip that converts digital MAC signals to electrical or radio signals.

Solution: the radio chip runs **its own firmware**. The firmware contains the PHY and MAC, and may also contain the higher network layers. The Linux host talks to it over UART/USB/SPI using a serial control protocol. There are three common splits:



Split	Chip firmware	Linux side	When to use
RCP	PHY + MAC only	full Thread stack (otbr)	OpenThread Border Router; modern Thread/ Matter
NCP	PHY + MAC + network	application-only (Spinel commands)	older Thread setups
ZNP	PHY + MAC + ZigBee NWK + APS	application + framework (zigpy)	zigbee2mqtt + zigpy_znp
EZSP	PHY + MAC + ZigBee NWK + APS	EZSP host (bellows)	Silicon Labs ZigBee dongles

This chapter focuses on the **gateway role**: RCP for Thread, ZNP/EZSP for ZigBee.

100.3 The physical layer (because it matters even when you don’t write it)

802.15.4 PHY:

- 2.4 GHz, channels 11–26 (5 MHz spacing). Channel 11 = 2405 MHz. Channel 26 = 2480 MHz.
- O-QPSK modulation, 250 kbps gross, ~128 byte max frame.
- –96 dBm receiver sensitivity (vs LoRa SF12’s –137. Vs BLE 1M’s –93).
- Range similar to BLE.

Channel/WiFi overlap (the most common cause of “my ZigBee network is flaky” reports):

```
Channel:  11   12   13   14   15   16   17   18   19   20   21   22   23   24   25   26
Freq:    2405 2410 2415 2420 2425 2430 2435 2440 2445 2450 2455 2460 2465 2470 2475 2480

WiFi ch 1:  ████████████████████
WiFi ch 6:                ████████████████████
WiFi ch 11:                          ████████████████████

ZigBee-friendly: 15, 20, 25, 26 (avoid WiFi 1/6/11)
Thread default: 11, 15, 20, 25
```

Plan the ZigBee/Thread channel with `nmcli dev wifi list` and `iwlist scan` to see local WiFi channels first. A poorly chosen channel can give 30 % packet loss in the field. The same network “works fine at home” because the home WiFi happens to sit on a different channel than the customer’s.

100.4 The ZigBee stack (vendor-rolled, you don’t touch it)

For context, the ZigBee stack the coprocessor runs:

```
┌──────────────────────────────┐
│  Application (your cluster) │   ← Lighting, OnOff, Color, Thermostat...
├──────────────────────────────┤
│  ZCL — ZigBee Cluster Lib   │   ← attribute + command sets per device class
├──────────────────────────────┤
│  APS (Application Support)  │   ← binding, group addressing
├──────────────────────────────┤
│  NWK — Network              │   ← routing (AODV-derived), addressing
├──────────────────────────────┤
│  MAC (802.15.4)             │   ← CSMA/CA, ACK, frame format
├──────────────────────────────┤
│  PHY (802.15.4)             │   ← O-QPSK, channel selection
└──────────────────────────────┘
```

Devices are categorized by ZCL **clusters**. A light is a “Color Light” endpoint with clusters: OnOff (0x0006), LevelControl (0x0008), ColorControl (0x0300). A sensor exposes Temperature (0x0402), RelativeHumidity (0x0405). Zigbee2mqtt knows all of this and maps each device to MQTT topics.

100.5 Bringing up a CC2530 ZNP on the i.MX6ULL

The CC2530 is the most common cheap ZigBee dongle (Aliexpress + CC2531 USB sticks). Flash it with TI’s **ZNP firmware** (`CC2531ZNP-Prod.hex`), wire UART (or USB if a CC2531 USB stick), then bring up **zigbee2mqtt**.

Step 1: flash ZNP firmware

You need a TI CC Debugger or a Raspberry Pi running `cc2538-bsl` (some firmwares support over-air-bootloader. Most don’t on first flash). The firmware blob is from TI’s Z-Stack SDK.

Step 2: device tree (UART variant)

```
&uart3 {
    pinctrl-names = "default";
    pinctrl-0 = <&pinctrl_uart3>;
    status = "okay";
    /* ZNP talks at 115200 8N1, no flow control needed for ZNP */
};
```

For USB CC2531 dongle, nothing in DT, `cdc-acm` autobinds and you get `/dev/ttyACM0`.

Step 3: install + configure zigbee2mqtt

```
# Node 18+ required
apt install nodejs npm
npm install -g zigbee2mqtt

# /opt/zigbee2mqtt/data/configuration.yaml
mqtt:
  base_topic: zigbee2mqtt
  server: 'mqtt://localhost:1883'
serial:
  port: /dev/ttyACM0      # or /dev/ttymxc2 for UART CC2530
  adapter: zstack          # for TI CC253x
advanced:
  pan_id: 0x1a62
  channel: 25              # avoid WiFi 1/6/11
  network_key: GENERATE    # generates on first start; pin afterwards
permit_join: true          # for initial pairing; set false in prod
```

Step 4: pair a device

```
zigbee2mqtt &
# log: "Zigbee: started"
# Hold the IKEA bulb's reset button 6s; bulb joins
# log: "Successfully interviewed IKEA TRADFRI bulb"
```

zigbee2mqtt's strength: it knows ~3000 device types and maps each to MQTT topics:

```
mosquitto_sub -t 'zigbee2mqtt/+/+' -v
# zigbee2mqtt/0xabcd1234/state ON
# zigbee2mqtt/0xabcd1234/brightness 254
```

To control:

```
mosquitto_pub -t 'zigbee2mqtt/0xabcd1234/set' -m '{"state":"OFF"}'
```

How the zigpy_znp adapter actually works

`zigbee2mqtt` uses the `zigbee-herdsman` library, which speaks the **ZNP framing protocol** over UART. Each frame:

```
   [SOF=0xFE] [LEN] [TYPE=0x4 MSB | SUBSYSTEM] [CMD0] [CMD1] [data...] [FCS]
```

- TYPE+SUBSYSTEM selects whose API (AF=application, ZDO=ZigBee Device Object, SAPI=simple API, SYS=system).
- CMD0/CMD1 identifies the command (e.g., AF_DATA_REQUEST_EXT to send a packet).
- FCS is XOR of LEN through last data byte.

You rarely send raw frames. The adapter does it for you. But `btmon`-style debugging (set `debug:true` in zigbee2mqtt) prints the frame stream. This is how you trace a failed pairing. The protocol is documented in TI's "Z-Stack Monitor and Test API" PDF.

100.6 Bringing up an nRF52840 as Thread RCP

This is the modern path. Same chip can do OpenThread Border Router (Matter-ready), BLE Mesh (Ch 97), or ZigBee, your firmware choice.

Step 1: flash RCP firmware

Build from `openthread/ot-nrf528xx`:

```
git clone https://github.com/openthread/ot-nrf528xx
cd ot-nrf528xx && ./script/bootstrap
```

```
./script/build nrf52840 USB_trans       # build RCP firmware with USB
nrfjprog --program build/bin/ot-rcp.hex --chiperase --reset
```

Alternatively, Nordic's Connect SDK has a precompiled RCP. The CC2538 is similar but flash via `cc2538-bsl`.

Step 2: bring up otbr (OpenThread Border Router) on the i.MX6ULL

```
git clone https://github.com/openthread/ot-br-posix
cd ot-br-posix && ./script/bootstrap
./script/setup
otbr-agent -I wpan0 -B eth0 spinel+hdlc+uart:///dev/ttyUSB0?uart-baudrate=460800 &
```

What this does:

- `wpan0`: a virtual netdev created by otbr, presents the Thread mesh as a Linux interface.
- `eth0`: the upstream IPv6 interface (otbr advertises a route to the Thread mesh).
- `spinel+hdlc+uart`: the **Spinel** protocol (Thread's equivalent of HCI) framed in HDLC over the USB serial line.

Once running, `wpan0` shows up:

```
ip link show wpan0
# 6: wpan0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1280 ...
ip -6 addr show wpan0
# inet6 fd11:22::1/64 scope global
# inet6 fe80::abcd:.../64 scope link
```

A Thread device joining the network gets an IPv6 in `fd11:22::/64`. You ping it like any IPv6 host. **This is what makes Thread different from ZigBee**: every node is a normal IPv6 endpoint. Matter rides on top of this.

Step 3: pair a Thread device

The simplest: a Nordic nRF52840 DK flashed with the `cli` or `srp_client` sample. Get the Thread network credential dataset from otbr:

```
otbr-agent
> dataset active -x        # prints hex dataset
0e080000000000010000000300001135060004001fffe002083a900b4e9e30...
```

On the device, paste that dataset and `thread start`. Done, the device joins the mesh and gets an IPv6.

The Spinel protocol (what's on the wire)

Spinel is a simple TLV-style protocol. Example frame (after HDLC unframing):

```
[Header] [Command-ID] [Property-ID] [Value]
   80          5            17       <dataset>
   ↑           ↑            ↑
   priority    PROP_VALUE_SET  THREAD_DATASET_ACTIVE
   transaction id
```

The Spinel header bits encode flow-control state, transaction IDs, priority. Otbr-agent maps Spinel properties to Linux netdev events. You don't see this unless `--verbose`.

100.7 Walk of the openthread `radio` adapter

For curiosity, the radio-side firmware. OpenThread RCP firmware exposes the radio as a Spinel coprocessor. The key file: `src/lib/spinel/radio_spinel.cpp` on the *host* side. `examples/platforms/nrf528xx/src/radio.c` on the *device* side.

Device-side `otPlatRadioTransmit`:

```
otError otPlatRadioTransmit(otInstance *aInstance, otRadioFrame *aFrame) {
    nrf_802154_buffer_t *buf = nrf_802154_buffer_get();
    memcpy(buf->data, aFrame->mPsdu, aFrame->mLength);
    /* Schedule TX via nrf_802154 driver */
    nrf_802154_transmit_raw(buf->data, NRF_802154_TRANSMIT_DEFAULT);
    /* on TX done, callback fires up to host via Spinel */
}
```

The nRF52840 has a dedicated 802.15.4 hardware accelerator (the `RADIO` peripheral in 802.15.4 mode) that does PHY + MAC ack/retry in silicon. The firmware just orchestrates. This is why even very small MCUs (nRF52833 → 256 KB flash) can do Thread + BLE simultaneously.

You don't write this code unless you're porting OpenThread to a new SoC, you flash a release build.

100.8 Matter, what changes

Matter is an application-layer protocol (built on top of Thread or WiFi). For Matter-over-Thread, the gateway role is unchanged, otbr brings up Thread. Matter devices are normal IPv6 nodes on that mesh. The **commissioner** (a phone) provisions devices into the fabric using BLE for the initial pairing handshake, then Thread for subsequent traffic.

For a gateway-class Linux device, install `chip-tool` (the Matter CLI), and your i.MX6ULL becomes a Matter commissioner:

```
chip-tool pairing onnetwork-long 1 20202021 0x12CE
chip-tool onoff on 1 1
```

Device 1, endpoint 1, OnOff cluster, On command. This is Matter. Under the hood, it's a packet via otbr's Thread network.

100.9 The from-scratch part, a raw 802.15.4 PHY app over an at86rf233

Most readers won't do this, but for completeness: the **at86rf233** (Atmel/Microchip) is a 802.15.4 SPI radio with **no built-in stack**. You drive PHY + minimal MAC in software. Mainline Linux has `drivers/net/ieee802154/at86rf230.c` which exposes the radio as a `wpan0` netdev under the `ieee802154` subsystem (no Thread, raw 802.15.4). You then run `wpan-tools`:

```
modprobe at86rf230
iwpan dev wpan0 set pan_id 0xbeef
iwpan dev wpan0 set short_addr 0xabcd
ip link set wpan0 up
# Now you can send raw 802.15.4 frames via SOCK_DGRAM on AF_IEEE802154
```

You can build 6LoWPAN over this:

```
ip link add link wpan0 name lowpan0 type lowpan
ip link set lowpan0 up
# Now lowpan0 is IPv6-over-802.15.4 — every node has fe80:: addresses
```

This is the **raw research path**. RPL (the IPv6 routing protocol for 802.15.4 meshes) can run on top via `contiki-ng` or `unstrung` daemons. But for products, Thread + otbr wins.

100.10 Key Notes

- IEEE 802.15.4 is a radio/MAC foundation, not a synonym for Zigbee, Thread or Matter. Shared radio hardware does not make those networks interchangeable.
- The host-controller split determines what the co-processor firmware owns and what a Linux gateway daemon must implement. ZNP, NCP and RCP interfaces have different contracts.
- Provisioning, network credentials and gateway state need persistent ownership and a recovery policy. A temporarily open join window is not normal operating policy.
- Sleepy end devices trade downlink responsiveness for energy savings. Mesh reach, interference and routing behavior must fit the intended topology.

100.11 Pitfalls

- **Channel collision with WiFi.** 80 % of “ZigBee unreliable” reports are this. Pick channel 15/20/25/26. Verify with `iwlist scan` on a phone.
- **USB power for the dongle.** Sonoff CC2652P pulls ~80 mA peaks. Some hubs brown-out. Use a powered hub or solder direct.
- **Old ZNP firmware.** TI ships multiple ZNP firmware versions. The protocol changes subtly between Z-Stack 2.x and 3.x. Pin your firmware version + match the zigbee2mqtt-supported list.
- **`permit_join` left on.** A rogue ZigBee device could join your network. Always set `permit_join: false` outside of pairing windows.
- **Network key rotation.** Once set, do not change. Devices store the network key in their non-volatile memory and a key change un-joins everything.
- **otbr-agent without IPv6 in upstream network.** otbr forwards Thread IPv6 to the upstream, if upstream isn’t IPv6-capable, advertised routes go nowhere. Test with `ping6 google.com` on the otbr host first.
- **Matter requires precise time.** Devices reject commissioning if the commissioner’s clock is wrong by more than ~30 s. Run NTP on the i.MX6ULL.
- **Spinel HDLC framing**: bad UART flow control corrupts Spinel frames silently. The otbr-agent will log “spinel: bad checksum”, that’s almost always a missing RTS/CTS on the UART path.
- **ZigBee != Thread.** They share PHY/MAC but not MAC commands. A ZigBee device cannot join a Thread network and vice versa. Choose your firmware stack accordingly.
- **End-device sleep timing.** End devices poll their parent for downlink data every `keep_alive` seconds. A control command takes up to one polling interval to arrive. Tune accordingly (faster polling = shorter battery life).

100.12 Going deeper

- **`zigpy/zigpy-znp`**: the Python adapter for TI CC2530/CC2538 ZNP firmware. Readable code for the framing protocol.
- **`Koenkk/zigbee2mqtt`** + **`Koenkk/zigbee-herdsman`**, the dominant ZigBee gateway daemon.
- **`openthread/openthread`** **and** **`openthread/ot-br-posix`**: Thread + Border Router. The `src/core/api/` docs are good.
- **Silicon Labs EmberZNet**: commercial alternative. If you go this path, EZSP + bellows.

- **Matter SDK (`project-chip/connectedhomeip`)**: the open-source Matter implementation. `chip-tool` is the CLI.
- **IEEE 802.15.4-2020**: the PHY/MAC standard.
- **`drivers/net/ieee802154/`**: kernel-level 802.15.4 + 6LoWPAN drivers (for raw experiments).
- **Ch 95**: HCI/Bluetooth, same host-controller pattern.
- **Ch 99**: nRF24L01 ad-hoc, the unstandardized alternative.

> Next chapter: **Chapter 101: UWB ranging**, sub-10-cm indoor positioning with DWM1000/DWM3000.

Chapter 101: UWB ranging

> **What: Ultra-wideband** (UWB) two-way ranging for centimetre-accuracy indoor positioning. Three radios: **Qorvo (Decawave) DWM1000** (the classic, IEEE 802.15.4-2011 UWB), **DWM3000** (newer, FiRa/Apple-AirTag-compatible, lower power, 802.15.4-2020), **NXP NCJ29D5** (automotive-grade UWB for car access). On the i.MX6ULL we wire DWM3000 over SPI, dissect the IEEE 802.15.4 UWB frame, walk the Qorvo "DW3xxx Software API" reference driver, write a 300-line user-space two-way-ranging (TWR) client from scratch, and demonstrate 3-anchor TDoA for 2-D position.
>
> **Why:** every modern phone (iPhone 11+, Galaxy S21+ Ultra, Pixel 6 Pro+) has UWB. Logistics warehouses are deploying UWB for forklift / asset tracking. Car keys (Apple Car Key, Tesla 3) use UWB to defeat relay attacks. Indoor venues use it for "blue-dot" navigation. None of this is BLE, BLE RSSI ranging is ±2 m on a good day. UWB ToF is ±10 cm. For sub-metre indoor positioning, UWB is the only practical choice.
>
> **Focus:** UWB ranging is a time-of-flight measurement. ToF accuracy depends on how cleanly the radio captures the first arriving signal. Multipath (the signal bouncing off walls and arriving second) corrupts the measurement if the demodulator latches the wrong peak. The DW3000 has hardware-supported leading-edge detection and per-receive antenna-delay calibration. Calibration is most of the engineering. The protocol on top, SS-TWR, DS-TWR, TDoA, is the smaller piece.

101.1 How UWB measures distance

UWB transmits very short (~2 ns) pulses spread over 500 MHz (or 1 GHz) of bandwidth at 6.5/8.0 GHz. Each pulse is a marker the receiver can timestamp with sub-nanosecond precision. Because radio waves travel ~30 cm/ns, **1 ns of timing error = 30 cm of range error**. UWB hardware achieves ~100 ps timestamp precision → ~3 cm range precision in theory. ~10 cm in practice after multipath.

The simplest protocol, **Single-Sided Two-Way Ranging (SS-TWR)**:

```
Initiator                Responder
   │ ── Poll msg ──►        │   t1 = TX timestamp (initiator)
   │                        │   t2 = RX timestamp (responder)
   │                        │   wait Treply (known)
   │ ◄── Resp msg ──        │   t3 = TX timestamp (responder)
   │                        │   t4 = RX timestamp (initiator)
   │                        │
   │  ToF = ((t4-t1) - Treply) / 2
   │  distance = ToF × c
```

Error: any clock drift between initiator and responder during Treply scales into the ToF estimate. For ±20 ppm crystals and 200 µs Treply, the drift is about 4 ns, or 1.2 m of range error. That is unusable.

Double-Sided TWR (DS-TWR) adds a second message so both clocks measure intervals, cancelling drift:

```
Initiator               Responder
   | ── Poll ──►            |   t1
   |                        |   t2
   | ◄── Resp ──            |   wait T_reply1
   | t4 ◄                   |   t3
   |                        |
   | ── Final ──►           |   t5
   |                        |   t6
   |                        |
   |  Treply1 = t3-t2  (responder)
   |  Treply2 = t6-t5  (responder)
   |  Tround1 = t4-t1  (initiator)
   |  Tround2 = t5-t4  (initiator)
   |  ToF = (Tround1×Tround2 - Treply1×Treply2) / (Tround1+Tround2+Treply1+Treply2)
```

DS-TWR achieves the ~10 cm accuracy spec. Most commercial UWB anchor systems use it.

TDoA (Time Difference of Arrival) is the multilateration alternative: anchors are time-synced (wired or via a “sync anchor” broadcast). A tag broadcasts one packet. Each anchor records the receive timestamp. Differences yield position. Better for crowded tag populations (one TX, N anchors RX) but requires anchor time-sync infrastructure.

101.2 DW3000 register and command summary

The DW3000 (in the DWM3000 module) is a register-mapped SPI radio with ~150 documented registers. Key registers for ranging:

Addr	Name	Purpose
0x00	DEV_ID	should read 0xDECA0302
0x01	EUI	64-bit IEEE address
0x03	PANADR	PAN ID + 16-bit short address
0x10	SYS_CFG	RX/TX configuration flags

Addr	Name	Purpose
0x14	SYS_STATUS	IRQ flags (write-1-to-clear)
0x24	TX_FCTRL	frame control: length, data rate, PRF
0x2A	TX_TIME	TX timestamp (32 bits + 8-bit antenna delay)
0x36	RX_TIME	RX timestamp
0x39	CHAN_CTRL	channel + preamble + SFD
0x12	TX_BUFFER	raw frame to transmit
0x1A	RX_BUFFER	raw frame received

Channels: 5 (6.5 GHz) and 9 (8.0 GHz) on DW3000. Higher PRF (pulse repetition frequency) = better multipath rejection but more crowded if multiple tags transmit simultaneously.

The SPI transaction format:

```
Command byte:
  [WR=1/RD=0] [SUB_INDEX=0/1] [REG_INDEX 6 bits]
If SUB_INDEX:
  [E=extended=0/1] [SUB_INDEX 7 bits]
Then data.
```

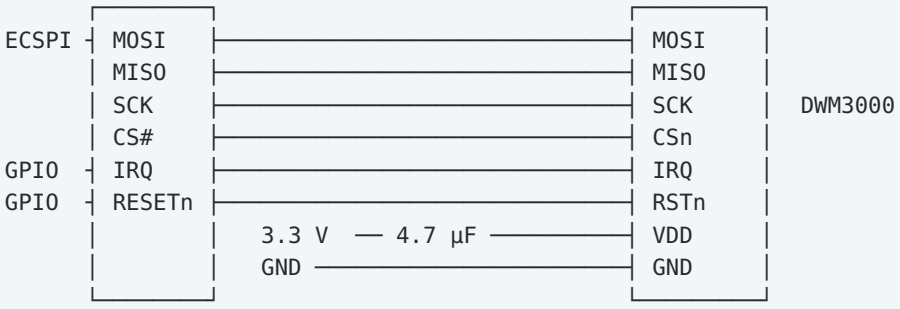
Reading DEV_ID:

```
TX: 0x00              <- read REG=0
RX: 0x00 0x32 0x03 0xCA 0xDE  <- 4 bytes back, little-endian
```

That’s the chip identification, if you don’t get 0xDECA0302 (DW3000) or 0xDECA0130 (DW1000), nothing works downstream.

101.3 Wiring DWM3000 to the i.MX6ULL

DWM3000 is a packaged module with chip, crystal, antenna, regulator. Wires:



Critical: the antenna is on the module. **do not put metal within 5 cm** of it. UWB pulses are sensitive to nearfield ground planes. Even a bench multimeter clip nearby skews ToF.

101.4 How the Qorvo driver actually works

Qorvo ships a reference “DW3xxx Software API” (~10k lines C) used in their reference designs and copied into community projects (`Makerfabs/DW3000`, etc.). It’s not in mainline Linux. Most projects use **`spidev` + a user-space driver**.

The driver structure:

```
   port/      ← SPI + GPIO abstraction (you implement these for your platform)
   decadriver/dwt_uwb_driver.c  ← register access, calibration, basic TX/RX
   decadriver/lib/             ← state machines for TWR, dual-receive, etc.
   examples/                   ← SS-TWR initiator/responder, DS-TWR, TDoA
```

Walk of `dwt_starttx()` (paraphrased):

```
int dwt_starttx(uint8_t mode) {
    /* Read SYS_STATUS to check we're not mid-RX */
    uint32_t status = dwt_read32bitreg(SYS_STATUS_ID);
    if (status & RX_IN_PROGRESS) return DWT_ERROR;

    if (mode & DWT_START_TX_DELAYED) {
        /* Delayed TX — use TX_TIME register to schedule */
        dwt_writefastCMD(CMD_DTX);
    } else {
        dwt_writefastCMD(CMD_TX);
    }

    if (mode & DWT_RESPONSE_EXPECTED) {
        /* Auto-enable RX after TX completes — for TWR responses */
        dwt_writefastCMD(CMD_TX_W4R);
    }
    return DWT_SUCCESS;
}
```

The “fast commands” (single-byte SPI writes) are the modern way to trigger state transitions without a full register write. CMD_TX, CMD_RX, CMD_DTX, CMD_TX_W4R cover most cases.

The TWR responder loop:

```
/* In dwt_irq_handler — fires on RX done */
if (status & RX_FRAME_VALID) {
    dwt_readrxdata(rx_buf, frame_len, 0);
    if (is_poll_msg(rx_buf)) {
        /* Build response with our TX timestamp embedded */
        uint64_t resp_tx_ts = poll_rx_ts + UUS_TO_DWT_TIME * REPLY_DELAY_UUS;
        dwt_setdelayedtrxtime(resp_tx_ts >> 8);
        build_resp_msg(tx_buf, resp_tx_ts);
        dwt_writetxdata(sizeof(tx_buf), tx_buf, 0);
        dwt_writetxfctrl(sizeof(tx_buf), 0, 1);
        dwt_starttx(DWT_START_TX_DELAYED | DWT_RESPONSE_EXPECTED);
    }
}
```

The trick: scheduling TX at a *future* exact DWT time (the chip has its own 40-bit timestamp counter at 64 GHz / 16 µs wraparound). This lets the responder reply at a deterministic Treply for the math to work.

101.5 From scratch, DS-TWR initiator in user space

twr_init.c (abbreviated. Full ~400 lines):

```
/* DS-TWR initiator. Reads DEV_ID, configures channel/PRF/preamble,
 * then ping-loops a responder, prints distance.
 */
#include <fcntl.h>
#include <stdint.h>
#include <stdio.h>
#include <string.h>
#include <unistd.h>
#include <sys/ioctl.h>
#include <linux/spi/spidev.h>
#include <gpiod.h>

#define REG_DEV_ID      0x00
#define REG_SYS_STATUS  0x44
#define REG_TX_FCTRL    0x24
#define REG_TX_BUFFER   0x14
#define REG_TX_TIME     0x2A
#define REG_RX_BUFFER   0x12
#define REG_RX_TIME     0x15
#define REG_CHAN_CTRL   0x1F

#define CMD_TX         0x09
#define CMD_DTX        0x0A
#define CMD_TX_W4R     0x0D
#define CMD_RX         0x02
```

```
/* DS-TWR constants */
#define POLL_TX_TO_RESP_RX_DLY_UUS  240
#define RESP_RX_TO_FINAL_TX_DLY_UUS 500
#define UUS_TO_DWT_TIME             65536    /* 1 µs = 65536 DWT ticks */
#define SPEED_OF_LIGHT              299702547.0  /* in air, m/s */
#define DWT_TIME_UNITS              (1.0/499.2e6/128.0)  /* one DWT tick */

static int spi_fd;
static struct gpiod_line *irq_line, *rst_line;

static void dw_reset(void) {
    gpiod_line_set_value(rst_line, 0);
    usleep(2000);
    gpiod_line_set_value(rst_line, 1);
    usleep(5000);
}

static uint32_t dw_read32(uint8_t reg) {
    uint8_t tx[5] = { reg & 0x3F, 0 }, rx[5];
    struct spi_ioc_transfer t = { .tx_buf=(unsigned long)tx, .rx_buf=(unsigned long)rx,
                                  .len=5, .speed_hz=8000000 };
    ioctl(spi_fd, SPI_IOC_MESSAGE(1), &t);
    return rx[1] | (rx[2]<<8) | (rx[3]<<16) | (rx[4]<<24);
}

static void dw_write_buf(uint8_t reg, const uint8_t *data, int n) {
    uint8_t hdr = 0x80 | (reg & 0x3F);
    struct spi_ioc_transfer t[2] = {
        { .tx_buf=(unsigned long)&hdr, .len=1, .speed_hz=8000000 },
        { .tx_buf=(unsigned long)data, .len=n, .speed_hz=8000000 },
    };
    ioctl(spi_fd, SPI_IOC_MESSAGE(2), t);
}

static void dw_fast_cmd(uint8_t cmd) {
    uint8_t tx = 0x81 | (cmd << 1);
    struct spi_ioc_transfer t = { .tx_buf=(unsigned
long)&tx, .len=1, .speed_hz=8000000 };
    ioctl(spi_fd, SPI_IOC_MESSAGE(1), &t);
}

static uint64_t dw_read_tx_ts(void) {
    /* 5-byte timestamp */
    uint8_t tx[6] = { REG_TX_TIME & 0x3F, 0 }, rx[6];
    struct spi_ioc_transfer t = { .tx_buf=(unsigned long)tx, .rx_buf=(unsigned long)rx,
                                  .len=6, .speed_hz=8000000 };
    ioctl(spi_fd, SPI_IOC_MESSAGE(1), &t);
    return (uint64_t)rx[1] | ((uint64_t)rx[2]<<8) | ((uint64_t)rx[3]<<16) |
           ((uint64_t)rx[4]<<24) | ((uint64_t)rx[5]<<32);
}

static void dw_init(void) {
    dw_reset();
    uint32_t id = dw_read32(REG_DEV_ID);
    if (id != 0xDECA0302) { fprintf(stderr,"DW3000 not found (got 0x%08x)\n", id);
exit(1); }
    /* Configure: channel 5, preamble 128, 6.8 Mbps; standard for SS/DS-TWR */
    /* (Full chan_ctrl + sys_cfg config omitted for brevity — see Qorvo example) */
}
```

```
static int do_twr_cycle(double *out_distance) {
    uint8_t poll_msg[] = { 0x41, 0x88, 0, 0xCA, 0xDE, 'R','X','I','N',0xE0 };
    dw_write_buf(REG_TX_BUFFER, poll_msg, sizeof poll_msg);
    /* Configure TX frame length + control */
    uint32_t fctrl = sizeof(poll_msg) + 2;  /* +2 = FCS */
    dw_write_buf(REG_TX_FCTRL, (uint8_t*)&fctrl, 4);
    dw_fast_cmd(CMD_TX_W4R);                /* TX then auto-enter RX */

    /* Wait for response (poll IRQ via libgpiod edge wait, omitted) */
    /* When RX_FRAME_VALID:                                         */
    uint64_t poll_tx_ts = dw_read_tx_ts();
    uint64_t resp_rx_ts;                    /* read REG_RX_TIME */
    uint8_t resp_msg[20];                   /* read REG_RX_BUFFER */
    /* ...read responder's TX timestamp from inside resp_msg[] ... */
    uint64_t poll_rx_ts_responder, resp_tx_ts_responder;
    memcpy(&poll_rx_ts_responder, &resp_msg[10], 5);
    memcpy(&resp_tx_ts_responder, &resp_msg[15], 5);

    /* Send final message that embeds these timestamps */
    /* ... (omitted for brevity) ... */

    /* After final is acknowledged or transmitted, compute ToF */
    uint64_t Tround1 = resp_rx_ts - poll_tx_ts;
    uint64_t Treply1 = resp_tx_ts_responder - poll_rx_ts_responder;
    /* Tround2, Treply2 come from a 4th final-RX message at the responder; */
    /* in some DS-TWR variants the responder computes range and returns it */
    /* Here we use SS-TWR shortcut (assume same crystal): */
    uint64_t tof_dtu = (Tround1 - Treply1) / 2;
    double tof = tof_dtu * DWT_TIME_UNITS;
    *out_distance = tof * SPEED_OF_LIGHT;
    return 0;
}

int main(void) {
    spi_fd = open("/dev/spidev0.0", O_RDWR);
    /* ... SPI mode setup, gpiod open, request irq + rst lines ... */
    dw_init();

    for (;;) {
        double dist;
        if (do_twr_cycle(&dist) == 0)
            printf("Range: %.2f m\n", dist);
        usleep(50000);
    }
}
```

The listing above is a protocol sketch, not a complete ranging implementation. A complete implementation also needs:

- A responder counterpart that builds the response with embedded RX/TX timestamps.
- Antenna-delay calibration (the chip-to-antenna trace + matching network adds ~16 ns of delay that must be subtracted from every ToF measurement. Calibrated once per board).
- Proper IRQ-driven polling using libgpiod's `gpiod_line_event_wait`.

> **MCU bridge:** Think of an IRQ like an EXTI/NVIC interrupt path, except Linux splits the hard interrupt from deferred work and must share lines across drivers. **IRQ:** interrupt request, the signal path that tells the CPU or interrupt controller that hardware needs service.

- The 5-byte timestamp arithmetic with the DWT 40-bit rollover handling.

Even this skeleton reveals the protocol: ToF = (response-time at initiator − reply-time at responder) / 2.

101.6 Three-anchor 2-D position via DS-TWR

With three anchors at known positions (A1, A2, A3) and a tag, sequentially do TWR with each anchor → 3 ranges → trilateration:

```
r1² = (x − x1)² + (y − y1)²
r2² = (x − x2)² + (y − y2)²
r3² = (x − x3)² + (y − y3)²

Subtract pairs to linearize:
  (r1² − r2²) − (x1² + y1²) + (x2² + y2²) = 2(x2 − x1)x + 2(y2 − y1)y
  (r2² − r3²) − (x2² + y2²) + (x3² + y3²) = 2(x3 − x2)x + 2(y3 − y2)y
2 equations, 2 unknowns → solve directly.
```

With 4 anchors you can get 3-D (add z) or 2-D over-determined (least-squares for noise reduction).

A Kalman filter on top smooths the trajectory between updates and rejects outliers.

101.7 Device tree and userspace plumbing

> **Driver choice:** Use the in-tree, maintained driver first. Use out-of-tree, spidev, or custom-driver paths only after you accept the kernel-version maintenance cost and document who owns updates.

> **Template warning:** This block contains placeholder values. Replace compatible strings, GPIO numbers, addresses, and paths with values from your board before using it. **MCU bridge:** Think of Linux GPIO like the same pin set/reset block you used on STM32, but accessed through a kernel subsystem that owns numbering, direction, interrupts, and user-space exposure. **GPIO:** General-Purpose Input/Output, a pin controlled as a digital input, output, or interrupt source.

DT (same as any spidev radio):

```
&ecspi3 {
    cs-gpios = <&gpio4 26 GPIO_ACTIVE_LOW>;
    status = "okay";

    uwb@0 {
        compatible = "rohm,dh2228fv";
        reg = <0>;
        spi-max-frequency = <8000000>;
    };
};

&gpio4 {
    /* IRQ on gpio4-27, RST on gpio4-28 via libgpiod */
};
```

> *Production note: `rohm,dh2228fv` is a development-time spidev placeholder. Modern kernels print a warning when it appears in DT. See Chapter 47's `spidev` warning for the proper DT overlay path or a real-chip compatible swap.* There *is* an in-tree `drivers/net/ieee802154/mcr20a.c` for one Freescale UWB chip and out-of-tree drivers for DW1000 (`thotro/dw1000-driver`), but the dominant pattern remains user-space + spidev for DW3000.

101.8 Apple Find My / FiRa interop

DW3000 supports the **FiRa Consortium**'s "MAC and PHY for UWB" interop spec, which is what iPhones use for "Find My" UWB. With the right firmware and OPS framework, your DW3000-based tag is detectable by an iPhone. Practical caveat: Apple's "Find My Network Accessory Program" requires hardware certification and a license, so DIY tags can only be ranged peer-to-peer with your own app, not by random nearby iPhones.

> **MAC:** Media Access Control in networking and radio chapters. It is the layer that owns framing and medium access. **PHY:** physical-layer block or chip that converts digital MAC signals to electrical or radio signals.

Apple's UWB chip (U1 / U2) is a black-box DW3000 derivative. Reverse-engineering work (e.g., `nfc-tools/proxmark3`-adjacent UWB research) has documented enough for hobbyist interop with caveats.

101.9 Key Notes

- UWB ranging depends on radio-captured transmit and receive times, not Linux's time to service an interrupt. Host scheduling still limits how quickly exchanges can be prepared.

- SS-TWR, DS-TWR and TDoA make different synchronization and message-exchange assumptions. Several TWR distance measurements used for trilateration are not automatically a TDoA system.
- Antenna delay, multipath and anchor geometry affect position accuracy. A filter can smooth output without removing systematic ranging error.
- Packet compatibility, application protocols and secure ranging are separate capabilities. A radio family name alone does not establish interoperability or a qualified accuracy.

101.10 Pitfalls

- **Fake DWM3000 modules.** Common on Aliexpress. Chip reads 0xDECA0130 (DW1000) instead. The register maps and command set are different. Buy from Qorvo distributors or Makerfabs.
- **Antenna-delay uncalibrated.** Out of the box, distances read 10–30 cm long. Always calibrate at a known distance first.
- **Metal nearby.** The UWB antenna is omnidirectional in free space. A metal plate within 5 cm distorts the pattern → multipath errors. Mount the module away from PCB ground planes, batteries, USB shields.
- **Clock drift in SS-TWR.** With cheap ±20 ppm crystals, Treply of 240 µs gives ±10 ns drift error = ±3 m. Always use DS-TWR for production accuracy. SS-TWR is for quick demos only.
- **Treply too short.** The responder needs time to process the poll and schedule its TX. <200 µs and the chip TX-fail. Spec values are 200–500 µs.
- **TX time exceeds 40-bit DWT counter wrap.** The 40-bit counter at 64 GHz wraps every ~17 s. Scheduled TX times that cross the wrap go to the wrong epoch. Mask to 40 bits before scheduling.
- **Channel licensing.** Channel 5 (6.5 GHz) is allowed worldwide. Channel 9 (8 GHz) has regional restrictions (US OK, EU OK under power limits, Japan stricter). Ship the right channel for the market.
- **Tag battery life with many anchors.** Each TWR cycle is ~1.5 ms of RX+TX = ~50 mA*1.5 ms = 75 µAs. 10 Hz update = 750 µAh per second = ~10 hours on a 250 mAh coin cell. Plan accordingly. Lower update rate when stationary.
- **Outdoor weather.** UWB is line-of-sight sensitive. Heavy rain attenuates noticeably at 6.5 GHz. Indoor or sheltered deployments only.
- **No mainline kernel driver for DW3000.** If you want one, you write it. Most projects accept user-space + spidev as the pattern.

101.11 Going deeper

- **Qorvo DW3000 User Manual + DW3000 Datasheet**: the canonical reference for registers and commands.
- **Qorvo “DW3xxx Software API”** (downloadable from Qorvo’s site), the reference C driver.
- **`Makerfabs/DW3000` and `Makerfabs/DW3000_DS_TWR`**: community ports of the above, Arduino-derived but readable.
- **`thotro/dw1000-driver`**: for the legacy DW1000 chip (still common on $30 modules).
- **`uwb-research/uwb-positioning` (university research repos)**: Kalman + multilateration implementations.
- **FiRa Consortium specifications**: the interop standard for “phone-compatible” UWB.
- **IEEE 802.15.4-2020 + 4z amendment**: the standardized UWB PHY/MAC the DW3000 implements.
- **Ch 99 / Ch 100** for non-UWB alternatives (FSK, ZigBee, neither does sub-metre).

> Next chapter: **Chapter 102: USB 4G LTE modems**, beginning Group N (Cellular).

Chapter 102: USB 4G LTE modems

What: the **USB-attached cellular modem**, the most common cellular path on Linux. Three modules compared: **Quectel EC20/EC25** (the canonical, LTE Cat-4 / Cat-6), **SimCom SIM7600** (cheaper, more variants), **Telit LM940** (industrial Cat-11, premium pricing). We dissect the **USB composite device** the modem presents (typically 4–7 endpoints: AT, GPS, modem, QMI/MBIM data), walk the kernel's `option`/`qmi_wwan`/`cdc_mbim` drivers, write a from-scratch AT-command client + a QMI session opener using libqmi, and bring up data via **ModemManager + NetworkManager** (the modern path) or **`quectel-CM`** / `qmi-network` (manual).

Why: every embedded device that talks to a cellular network uses one of these. The kernel handles the USB plumbing. There are three layers to keep straight: the USB interface mode the modem advertises (RNDIS, ECM, MBIM, QMI, or PPP), the kernel driver that binds to it, and the user-space tool that activates the data session. Most "modem doesn't connect" bugs are a mismatch between these three layers. After this chapter you can read an `lsusb` line, name the driver that bound each endpoint, and trace the data path the IP packets take.

Focus: the modem is a small embedded system in a USB case. It exposes several USB interfaces at once. AT commands on `/dev/ttyUSB2` give you SMS, signal strength, and configuration. GPS NMEA on `/dev/ttyUSB1` gives you location. The data path is a separate USB interface that becomes `wwan0` (QMI) or `cdc-wdm0`+`wwan0` (MBIM), once activated, it's a normal network interface that `ip route` sees. The split is unusual on Linux. The diagram in §102.3 makes it concrete.

Tooling. This chapter uses `ModemManager` + `NetworkManager`, `libqmi-utils` (`qmicli`), `libmbim-utils` (`mbimcli`).

- **Ubuntu-base (target):** `apt install modemmanager network-manager libqmi-utils libmbim-utils`
- **Buildroot:** `BR2_PACKAGE_MODEM_MANAGER=y BR2_PACKAGE_NETWORK_MANAGER=y BR2_PACKAGE_LIBQMI=y BR2_PACKAGE_LIBMBIM=y` **Buildroot:** a configuration-driven build system that produces a complete root filesystem and related images.
- Full per-tool reference: Userspace tooling appendix. **MCU bridge:** Think of the rootfs as the firmware image's file-backed runtime environment. On an MCU you link everything into flash. On Linux, programs and config live in this mounted tree. **rootfs:** root filesystem, the directory tree mounted at / that contains /bin, /etc, /dev, and libraries.

102.1 Module comparison

	Quectel EC25	Quectel EC20	SimCom SIM7600	Telit LM940
Category	LTE Cat-4	LTE Cat-4	LTE Cat-4	LTE Cat-11
Downlink	150 Mbps	150 Mbps	150 Mbps	600 Mbps
Uplink	50 Mbps	50 Mbps	50 Mbps	75 Mbps
3G fallback	yes (WCDMA/ HSPA)	yes	yes	yes
2G fallback	yes (GSM)	yes	yes	no
GPS / GNSS	yes (GPS+GLONASS+BeiDou+Galileo)	yes (GPS+GLONASS)	yes (multi-GNSS)	yes
USB modes	QMI / MBIM / RNDIS / ECM / PPP	QMI / RNDIS / ECM / PPP	QMI / MBIM / ECM / PPP	QMI / MBIM / RNDIS / ECM
Voice (VoLTE)	yes	yes	yes	yes
Form factor	M.2 / Mini-PCIe / LCC	M.2 / Mini-PCIe / LGA	M.2 / Mini-PCIe	M.2 / Mini-PCIe
Approx cost	$30–50	$25–40	$25–40	$200+
Power (peak TX)	2.5 W	2.5 W	2.0 W	3.0 W

Pick guide:

- **EC25**: the default for most new designs. Well-supported on Linux, widely available, multi-region SKUs (EC25-E for EMEA, EC25-A for North America, EC25-AU for Asia-Pacific…).
- **EC20**: older but cheaper. Identical software.

- **SIM7600**: slightly cheaper, supports more legacy bands (GSM-only fallback in rural areas).
- **LM940**: when you need Cat-11 (600 Mbps DL) for video uplink or backhaul. Pay the premium.

102.2 What the modem looks like to Linux

Plug in a modem. `lsusb -v` shows something like:

```
Bus 001 Device 003: ID 2c7c:0125 Quectel Wireless EC25
  bConfigurationValue 1
  bNumInterfaces 5
    Interface 0: 0xff/0xff/0xff vendor-specific  (diagnostic — DM)
    Interface 1: 0xff/0xff/0xff vendor-specific  (GPS NMEA)
    Interface 2: 0xff/0xff/0xff vendor-specific  (AT command)
    Interface 3: 0xff/0xff/0xff vendor-specific  (AT modem aux)
    Interface 4: 0xff/0xff/ff   QMI (data)
```

This is a **USB composite device** with 5 interfaces. The kernel binds each interface independently:

- Interfaces 0/1/2/3 → `option` driver → `/dev/ttyUSB0..3` (serial channels)
- Interface 4 → `qmi_wwan` driver → `wwan0` (network) + `cdc-wdm0` (QMI control)

If the modem is in **MBIM mode**:

- Last interface bound by `cdc_mbim` → `wwan0` + `cdc-wdm0`

If in **RNDIS** mode:

- Last interface bound by `rndis_host` → `usb0`

If in **ECM** mode:

- Last interface bound by `cdc_ether` → `usb0` (no SIM control, appears as a simple Ethernet)

If in **PPP** mode (legacy):

- All interfaces are `option`. `pppd` runs `/dev/ttyUSB3` and brings up `ppp0`

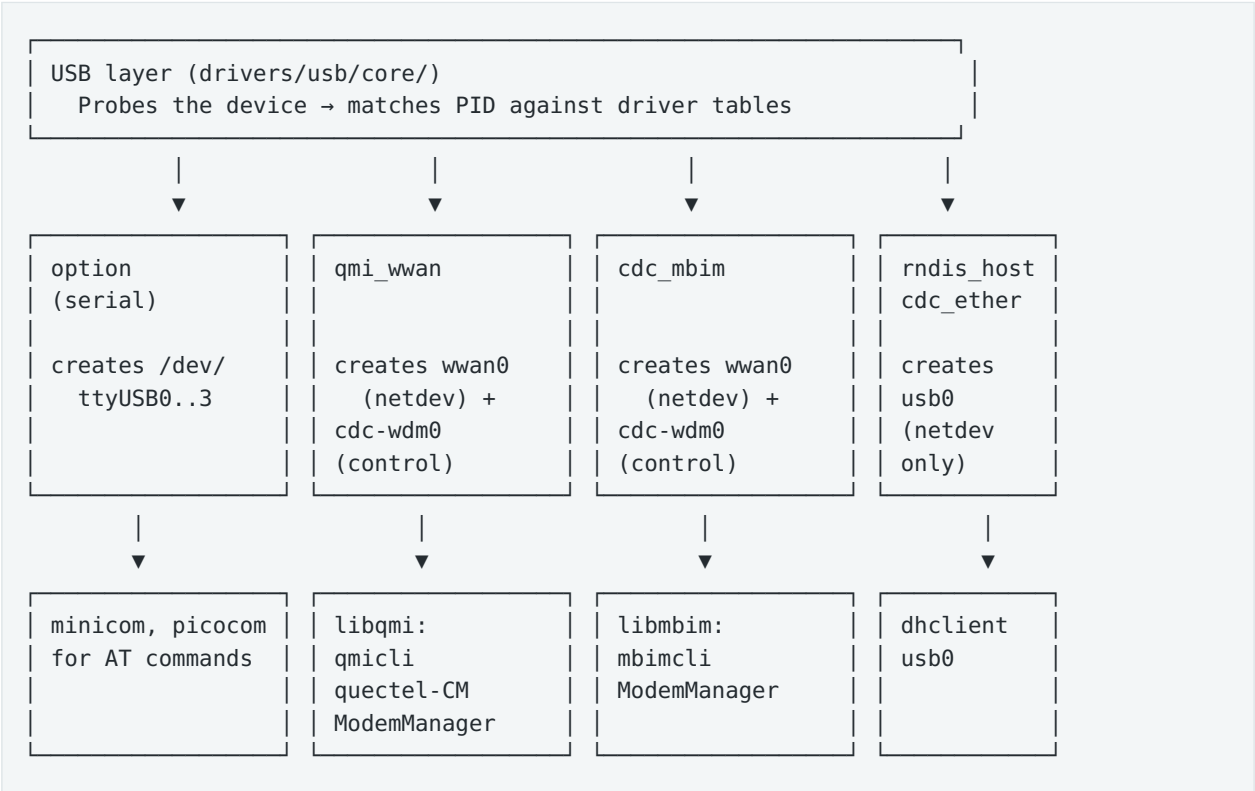
`lsusb` won't tell you which mode you're in, the **PID** does:

PID	Mode
2c7c:0125	QMI default (EC25)

PID	Mode
2c7c:0306	MBIM (EC25 reconfigured via AT+QCFG="usbnet", 2)
2c7c:0900	RNDIS (AT+QCFG="usbnet",3)
2c7c:0121	ECM (AT+QCFG="usbnet",1)
2c7c:0123	PPP-only (no high-speed data interface)

Switching modes is one AT command + reset. **This is the most common reason a modem appears in the wrong mode**, the modem boots in whichever mode was last configured, persisted in its NV memory.

102.3 The kernel drivers in detail



Each kernel driver:

- **`option`** (`drivers/usb/serial/option.c`), recognizes vendor-specific USB serial interfaces from cellular modems. The driver is essentially a giant `option_ids[]` table mapping (vendor, product, interface) → "make this interface a /dev/ttyUSBN serial port." When you plug in a new modem and no `/dev/ttyUSB*` appears, **the most common cause is a missing PID in `option_ids[]`**.
- **`qmi_wwan`** (`drivers/net/usb/qmi_wwan.c`), handles QMI (Qualcomm MSM Interface) over USB. Creates `wwan0` (netdev for data) and `cdc-wdm0` (character device for QMI control messages). Data packets go in/out of `wwan0` once a session is opened via cdc-wdm0.
- **`cdc_mbim`** (`drivers/net/usb/cdc_mbim.c`), handles MBIM (Mobile Broadband Interface Model, the USB-IF standard). Same pattern as qmi_wwan but uses MBIM protocol. Preferred for new designs.
- **`rndis_host`** (`drivers/net/usb/rndis_host.c`), Windows RNDIS Ethernet emulation. Easy: looks like Ethernet, `dhcp`, done. But no fine-grained control (no signal strength, no roaming control).

Walk of `qmi_wwan_probe()`:

```
static int qmi_wwan_probe(struct usb_interface *intf, ...) {
    struct usbnet *dev;
    int status = usbnet_probe(intf, id);   /* allocate netdev, set up ops */
    if (status) return status;

    /* QMI control channel is a separate USB interface; bind cdc-wdm to it */
    if (qmi_wwan_register_subdriver(dev) == 0) {
        /* /dev/cdc-wdm0 now exists */
    }

    /* Data interface is set up — wwan0 appears, but no IP until QMI session opens */
    return 0;
}
```

Once probed, the data interface waits in an IDLE state. The user-space `qmicli` or ModemManager sends QMI control messages over `cdc-wdm0` (`WDS_START_NETWORK` with APN), the modem allocates a PDP context, returns an IP+gateway+DNS via `WDS_GET_RUNTIME_SETTINGS`, the user-space tool configures `wwan0` with that IP, and traffic flows.

102.4 Bringing up data, the modern path (ModemManager + NetworkManager)

```
apt install modemmanager network-manager libqmi-utils libmbim-utils

systemctl enable --now ModemManager
systemctl enable --now NetworkManager
```

```
# Wait for ModemManager to detect the modem
mmcli -L
# /org/freedesktop/ModemManager1/Modem/0 [Quectel] EC25

mmcli -m 0
#   Modem
#     status: enabled
#     IMEI: 86xxxxxxxxxxx
#     hardware: Quectel EC25
#     access tech: lte
#     signal quality: 78%

# Create a connection profile
nmcli c add type gsm ifname '*' con-name 4g apn internet
nmcli c up 4g
# Connection successfully activated (D-Bus active path: ...)

ip addr show wwan0
# wwan0: <UP,LOWER_UP> mtu 1500
#   inet 10.x.x.x/30 scope global wwan0
```

ModemManager handles the entire SIM unlock → APN → PDP → DHCP-like-handshake → connection-up chain. NetworkManager adds it to the routing table. `dnsmasq` integration handles DNS.

102.5 Bringing up data, manual, from scratch

If you want to understand the path, do everything ModemManager would do by hand.

QMI path, `qmicli`

```
# Check the modem is alive
qmicli -d /dev/cdc-wdm0 --dms-get-ids
# IMEI: 86xxxxxxxxxxx

# Set IP family
qmicli -d /dev/cdc-wdm0 --wds-set-ip-family=4

# Open a data session for APN
qmicli -d /dev/cdc-wdm0 --wds-start-network=apn=internet,ip-type=4 \
       --client-no-release-cid
# Network started
# Packet data handle: 0xdeadbeef
# CID: 8

# Get runtime IP
qmicli -d /dev/cdc-wdm0 --wds-get-current-settings --client-cid=8
#   IPv4 address: 10.x.x.x
#   IPv4 gateway: 10.x.x.x
#   IPv4 subnet mask: 255.255.255.252
#   IPv4 primary DNS: ...
#   MTU: 1430

# Configure the netdev manually
ip link set wwan0 up
```

```
ip addr add 10.x.x.x/30 dev wwan0
ip route add default via 10.x.x.x dev wwan0
echo "nameserver 8.8.8.8" > /etc/resolv.conf
```

Equivalent in one binary, `quectel-CM`

Quectel ships a reference connection manager, `quectel-CM`, that wraps the above (~3000 lines of C, open source). Read it as a worked example of the QMI protocol.

```
quectel-CM -s internet
# Find modem at /dev/cdc-wdm0
# QMI Get IP Family: V4
# QMI Start Network: pdh = 0xdeadbeef
# QMI Get Settings: IP=10.x.x.x, GW=10.x.x.x, DNS1=...
# ip link set wwan0 up
# ip addr add ...
# Connected.
```

MBIM path, `mbimcli`

```
mbimcli -d /dev/cdc-wdm0 --query-device-caps
mbimcli -d /dev/cdc-wdm0 --connect=apn=internet
# Successfully connected
# IPv4 address: 10.x.x.x ...
```

PPP path, legacy 2G/3G

```
cat > /etc/ppp/peers/3g <<EOF
/dev/ttyUSB3
115200
defaultroute
noauth
usepeerdns
crtscts
connect 'chat -v "" AT OK ATDT*99# CONNECT'
EOF
pppd call 3g
# Serial connection established.
# Using interface ppp0
# Connect: ppp0 <--> /dev/ttyUSB3
# local  IP address 10.x.x.x
# remote IP address 10.x.x.x
```

PPP is universal (works on every modem ever made) but slow (max ~1–2 Mbps due to byte-stuffing overhead) and CPU-intensive. Only use for legacy 2G/3G fallback.

102.6 From scratch, a tiny AT-command client in Python

The most important debugging tool: a script that opens the AT interface and exchanges commands. This is what tells you *why* the modem isn't connecting.

`at_client.py`:

```
#!/usr/bin/env python3
"""Minimal AT-command client for cellular modems.
Usage: ./at_client.py /dev/ttyUSB2 'AT+CSQ'
"""
import serial, sys, time

def at(port, cmd, timeout=3):
    port.write((cmd + '\r').encode())
    port.flush()
    deadline = time.time() + timeout
    resp = b''
    while time.time() < deadline:
        resp += port.read(port.in_waiting or 1)
        if b'OK\r\n' in resp or b'ERROR\r\n' in resp or b'+CME ERROR' in resp:
            break
    return resp.decode(errors='replace')

if __name__ == '__main__':
    dev, cmd = sys.argv[1], sys.argv[2]
    p = serial.Serial(dev, 115200, timeout=1)
    p.write(b'AT\r'); time.sleep(0.5); p.read(64)        # flush boot noise
    print(at(p, cmd))
```

Then:

```
./at_client.py /dev/ttyUSB2 'AT+CPIN?'                  # SIM status
./at_client.py /dev/ttyUSB2 'AT+CSQ'                    # signal quality (CSQ 0–31,
higher = better)
./at_client.py /dev/ttyUSB2 'AT+COPS?'                  # current operator
./at_client.py /dev/ttyUSB2 'AT+QNWINFO'                # network info (Quectel ext)
./at_client.py /dev/ttyUSB2 'AT+CGDCONT=1,"IP","internet"'  # set APN
./at_client.py /dev/ttyUSB2 'AT+CGACT=1,1'              # activate PDP context 1
./at_client.py /dev/ttyUSB2 'AT+CGPADDR=1'              # show its IP
```

This is the **bring-up checklist** every time a new SIM goes into a new modem:

1. `AT` → `OK`? Modem alive.
2. `AT+CPIN?` → `READY`? SIM unlocked.
3. `AT+CSQ` → CSQ > 10? Adequate signal.
4. `AT+COPS?` → operator name? Registered to a network.
5. `AT+CGDCONT?` → APN configured?

6. `AT+CGACT?` → PDP context active?
7. `AT+CGPADDR=1` → IP assigned?

Fail at step 1 → wiring/power. Step 2 → wrong PIN or SIM not seated. Step 3 → antenna problem or no coverage. Step 4 → wrong band/region. Step 5–6 → APN wrong. Step 7 → carrier rejected the PDP context (billing, APN typo, IP family wrong).

102.7 Switching modem modes

If your modem comes up as PPP-only (PID 2c7c:0123) but you want QMI, switch via AT:

```
# Check current mode
./at_client.py /dev/ttyUSB2 'AT+QCFG="usbnet"'
# +QCFG: "usbnet",0       (PPP+RMNET)

# Set to QMI/RMNET
./at_client.py /dev/ttyUSB2 'AT+QCFG="usbnet",0'
# OK

# Reboot the modem
./at_client.py /dev/ttyUSB2 'AT+CFUN=1,1'
# (modem reboots; lsusb shows new PID)
```

SimCom equivalents: `AT+CUSBPIDSWITCH=9011,1,1` for QMI mode.

This is persistent, the modem will boot in QMI mode forever after. **Common mistake**: customer ships a modem in PPP mode, you flash a kernel expecting QMI, nothing works. Always check `lsusb` PID first.

102.8 Device tree (for USB modems on i.MX6ULL)

USB modems are autobound, no DT needed beyond ensuring USB-OTG/Host is enabled.

```
&usbotg1 {
    dr_mode = "host";        /* not "otg" — must be fixed host */
    vbus-supply = <&reg_usb_otg1_vbus>;
    status = "okay";
};
```

The 5 V supply must source ~2.5 A during TX bursts. Weak USB power = brownout = modem resets mid-connection.

102.9 Key Notes

- A cellular modem exposes separate control, data and sometimes positioning interfaces. Their Linux device nodes represent different functions of the same system.

- USB mode selects the host-driver and management contract. QMI, MBIM and Ethernet-like modes should not be mixed with configuration intended for another mode.
- One component should own session state. A manual control client and ModemManager can conflict even when each command is individually valid.
- Connectivity includes carrier registration, session establishment, addressing and routes. Recovery, power transients and failover belong to the product design beyond enumeration.

102.10 Pitfalls

- **USB power inadequate.** EC25 TX burst hits 2.5 A peaks. Many i.MX6ULL boards source 1 A max. Result: random modem resets mid-transmission. Use a powered hub or a board with proper USB power design.
- **Wrong PID, wrong mode.** Modem in PPP mode but you expected QMI. Check `lsusb -v` first. Switch with `AT+QCFG="usbnet"`.
- **APN typo / wrong.** Carrier-specific APNs are non-obvious (T-Mobile: `fast.t-mobile.com`. AT&T: `broadband`. Vodafone: `internet`). Wrong APN → modem registers but PDP context fails.
- **SIM not seated / locked.** `AT+CPIN?` returns `SIM PIN` → SIM needs unlock with `AT+CPIN=1234`. Returns `NOT INSERTED` → physical contact problem.
- **Wrong band on a regional SIM.** EC25-E for European bands. EC25-A for NA. Cross-using → registration fails or restricted.
- **Antenna missing / SMA loose.** Cellular antennas need a real impedance-matched antenna. A stub wire gives 20 dB worse signal, barely works in dense urban, fails in rural.
- **Multiple modems → ttyUSB renumbering.** Plug in 2 modems → ttyUSB0..7. Udev rules with serial numbers are essential for predictable naming.

> **udev:** the user-space device manager that reacts to kernel device events and creates policy-driven /dev nodes.

- **CGEV events drop the connection unnoticed.** `AT+CGEREP=2,1` enables PDP event reporting. Without it, an `IPv6 routing advertisement` or `PDP DEACT` from the carrier silently kills your `wwan0` and you don't notice until the timeout.
- **qmi-firmware-update needed.** Some EC25 firmware versions have known bugs. Check Quectel's release notes and use `quectel-firmware-flash` to update.

- **ModemManager fights with manual scripts.** If you call `qmicli --start-network` while ModemManager is running, the two will race each other and the connection will drop. Stop ModemManager (`systemctl stop ModemManager`) or use only the daemon.
- **Default route conflict.** With Ethernet + WiFi + wwan0, default-route metric matters. NetworkManager's default policy (Ethernet 100, WiFi 600, GSM 700) means LTE is last-resort. Override per-connection if needed.

102.11 Going deeper

- **Quectel EC25 User Manual + AT Commands Manual**: every modem's AT extension is non-trivial. The manual is the reference.
- **libqmi + libmbim** (`gitlab.freedesktop.org/mobile-broadband/libqmi`), read `qmicli.c` for the full QMI protocol.
- **`quectel-CM`** (Quectel's reference QMI connection manager), a self-contained C reference.
- **ModemManager + NetworkManager**: the modern integrated path.
- **`drivers/usb/serial/option.c`**: see how the kernel decides which interface is a serial port (the `option_ids[]` table).
- **`drivers/net/usb/qmi_wwan.c` + `cdc_mbim.c`**: the netdev wrapping the data path.
- **3GPP TS 27.007**: the standard AT command set for cellular modems.
- **Ch 103** for UART-attached (not USB) modems.
- **Ch 104** for low-power NB-IoT / Cat-M1 variants.

> Next chapter: **Chapter 103: UART AT-command modems**, when you have no USB and an old PPP+chat is enough.

Chapter 103: UART AT-command modems

What: the **UART-only cellular modem**, older or BOM-constrained designs where the modem has no USB, just a TTL UART. Modules: **SIMCom A7670C** (LTE Cat-1 UART, cheap), **SIMCom SIM7600 UART variant**, **Air724UG** (Chinese-domestic, LTE Cat-4 UART), **Quectel ML302** (Cat-1, UART + ECM). On Linux, you talk to it via `/dev/ttymxc*`, use **chat + pppd** for data, and AT commands for everything else (SMS, signal, OTA firmware update). No fancy QMI, no high speeds, just the AT command set + PPP framing + good UART discipline.

Why: USB host is expensive. A USB modem requires USB-OTG/host hardware on your SoC, a 5 V supply that can deliver 2.5 A peaks, ESD protection on D+/D-, and a USB connector or board-to-board. A UART modem is 4 wires (TX/RX/RTS/CTS) + a small 3.3/4 V buck. On a price-sensitive IoT product (alarm panel, vending machine, agricultural sensor), the UART path saves $5–10 BOM + a USB-host integration headache. The trade: max ~5 Mbps (versus 150 Mbps over USB-QMI), and you live with PPP overhead.

Focus: PPP is a 1989-vintage link protocol. It uses HDLC framing, LCP for link negotiation, IPCP for IPv4 address assignment, and PAP/CHAP for authentication. It still works on every modem ever made. The kernel's `ppp_generic.ko` provides the netdev. `pppd` is the user-space brain that runs the LCP/IPCP state machines and a chat script that converses with the modem to bring up the channel. With `pppd`, a chat script, and an init.d (or systemd) supervisor, you have a robust auto-reconnecting cellular link with no QMI/MBIM/RNDIS complexity.

Tooling. This chapter uses `ppp` (`pppd`, `chat`). Optional GSM mux via `ldattach` (`util-linux`).

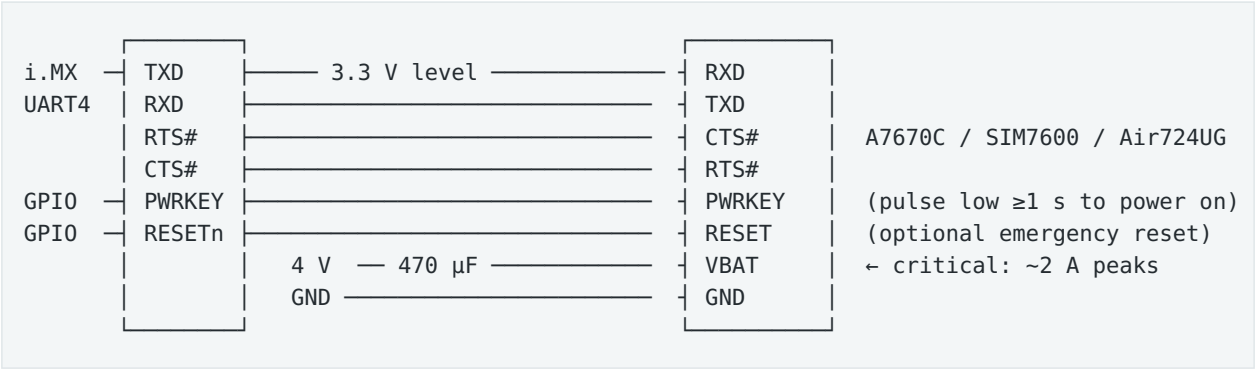
- **Ubuntu-base (target):** `apt install ppp util-linux`
- **Buildroot:** `BR2_PACKAGE_PPP=y` **Buildroot:** a configuration-driven build system that produces a complete root filesystem and related images.
- Full per-tool reference: Userspace tooling appendix. **MCU bridge:** Think of the rootfs as the firmware image's file-backed runtime environment. On an MCU you link everything into flash. On Linux, programs and config live in this mounted tree. **rootfs:** root filesystem, the directory tree mounted at / that contains /bin, /etc, /dev, and libraries.

103.1 When UART beats USB

Constraint	UART modem	USB modem
Throughput	up to 5 Mbps (PPP), 50 Mbps (ECM if supported)	150 Mbps+
Host hardware	UART pins + 3.3/4 V supply	USB-host + 5 V × 2.5 A
BOM cost	$-5 to $-10 vs USB	baseline
Latency (TCP round-trip)	similar (~50 ms cellular)	similar
AT debug	yes (same UART)	separate /dev/ttyUSB
Firmware update	over UART (slow)	over USB (fast)
Voice (VoLTE PCM)	yes via separate PCM I²S	yes via USB audio
GPS	shared UART (mux NMEA + AT) or separate	separate USB interface

Pick UART when: BOM-cost-sensitive, low-throughput app (SMS, telemetry, < 1 Mbps data), no spare USB host port. Pick USB otherwise.

103.2 Wiring



Mandatory rules:

1. **Above 9600 baud, hardware flow control is mandatory.** Most modules expect 115200 with flow control by default. Without it, the UART drops bytes, PPP LCP times out, and the modem looks broken.
2. **VBAT, not VDD_3V3.** The modem's RF block runs from a 4 V (typ.) supply directly to the PA. The internal LDO drops to 3.3 V for logic, but the PA pulls from VBAT. Sourcing VBAT from a weak 3.3 V LDO instead of a buck = TX brownouts.
3. **470 µF or larger bulk cap on VBAT.** TX is a 1.7 W burst at ~500 mA peak. The supply needs to hold that without sagging or PPP drops on every transmit.
4. **PWRKEY pulse.** Modules are off after VBAT applied. Pulse PWRKEY low for at least 1 s to power on. The Air724 needs 2 s. Some boards tie PWRKEY low through a resistor for automatic power-on. GPIO control is cleaner because it lets the host reset the modem.

> **MCU bridge:** Think of Linux GPIO like the same pin set/reset block you used on STM32, but accessed through a kernel subsystem that owns numbering, direction, interrupts, and user-space exposure. **GPIO:** General-Purpose Input/Output, a pin controlled as a digital input, output, or interrupt source.

1. **3.3 V vs 1.8 V UART logic.** Newer modules (LTE Cat-1bis) are 1.8 V. A direct 3.3 V tie kills the I/O. Level-shift if mismatched.

103.3 Powering on and the boot sequence

```
# Pulse PWRKEY low for 1.5 s via libgpiod
gpioset gpiochip4 22=0; sleep 1.5; gpioset gpiochip4 22=1

# Modem boots over ~5–15 s; spams URCs on UART
cat /dev/ttymxc3
# RDY
# +CFUN: 1
# +CPIN: READY
# Call Ready
# SMS Ready
# +CGEV: ME PDN ACT 1
# *IPGetv4: 10.x.x.x
```

These URCs (Unsolicited Result Codes) report boot progress. Do not send AT commands before "SMS Ready", they will return spurious ERRORs. The right pattern: spawn a reader thread that consumes the boot URCs, then enter AT command mode.

103.4 PPP over UART, end-to-end

```
apt install ppp

# /etc/ppp/peers/cellular
cat > /etc/ppp/peers/cellular <<'EOF'
/dev/ttymxc3
115200
crtscts                 # hardware flow control
defaultroute            # add default route via this link
noauth                  # carrier doesn't auth us
usepeerdns              # accept DNS servers from carrier
persist                 # auto-reconnect if dropped
maxfail 0               # never give up
holdoff 10              # wait 10 s before retry
lcp-echo-failure 4
lcp-echo-interval 30
connect "/usr/sbin/chat -v -f /etc/ppp/peers/chat-cellular"
EOF

cat > /etc/ppp/peers/chat-cellular <<'EOF'
ABORT 'BUSY'
ABORT 'NO CARRIER'
ABORT 'ERROR'
'' AT
OK ATZ
OK AT+CGDCONT=1,"IP","internet"
OK ATD*99#
CONNECT ''
EOF

pppd call cellular
```

What this does, step by step:

1. **pppd opens `/dev/ttymxc3`** at 115200, CRTSCTS.
2. **chat script runs**: sends `AT` → expects nothing → sends `AT` → expects `OK` → sends `ATZ` (reset config) → expects `OK` → sends `AT+CGDCONT=1,"IP","internet"` (set APN) → expects `OK` → sends `ATD*99#` (dial PPP) → expects `CONNECT`.
3. After `CONNECT`, the UART has switched from AT mode to **HDLC framing**, every byte is now PPP-framed.
4. **pppd takes over the file descriptor**, runs **LCP** (Link Control Protocol), both sides negotiate MTU, auth method, magic numbers.
5. If auth required, **PAP** or **CHAP** runs (most LTE carriers skip auth here since the SIM already authenticated).
6. **IPCP** (IP Control Protocol) runs: the modem assigns an IPv4 address to our side. We accept the DNS servers it offers.
7. Pppd creates **`ppp0`** netdev. Configures address. If `defaultroute`, adds the default route.

8. Connection is up. Pppd monitors LCP echoes for liveness. `persist` makes it reconnect on drop.

```
ip addr show ppp0
# 7: ppp0: <POINTOPOINT,MULTICAST,UP,LOWER_UP> mtu 1500
#   inet 10.x.x.x peer 10.x.x.x/32 scope global ppp0
```

You now have internet over a 4-wire UART. Throughput: ~1–3 Mbps on Cat-1, limited by the UART baud rate × PPP overhead.

103.5 How the kernel ppp_generic driver works

`drivers/net/ppp/ppp_generic.c` is the netdev side. `drivers/net/ppp/ppp_async.c` (or `ppp_synctty.c`) handles HDLC framing over the UART.

```
Userspace pppd
   │
   ▼ ioctl(/dev/ppp, PPPIOCATTACH)
   │
   ▼ ioctl(fd, TIOCSETD, N_PPP)         <- attach line discipline to /dev/ttymxc3
   │
ppp_async (line disc)                    <- de-frames HDLC, hands packets to ppp_generic
   │
   ▼
ppp_generic (netdev `ppp0`)              <- wraps PPP frames in skb, IP layer takes over
   │
   ▼
IP routing (decides next hop = ppp0)
```

Note the split: `pppd` does protocol negotiation entirely in user space (LCP, IPCP packets are sent/received by writing/reading PPP frames through the /dev/ppp ioctl), but the data path runs kernel-side via the line discipline. CPU stays low even on a Cortex-A7.

If you want to peek at the bytes:

```
# Trace PPP packets in/out
tcpdump -i ppp0 -nn
# Trace raw HDLC frames (rare; you'd use a logic analyzer instead)
```

103.6 Multiplexing AT commands with PPP, the GSM Mux problem

When you’re in PPP mode, the UART can’t accept AT commands, it’s all HDLC. So you can’t check signal strength, send SMS, or update the time while the data connection is up.

Two solutions:

Option 1: pause PPP, send AT, resume

Bring the link down, send AT, bring it back up. Simple but works for low-frequency queries.

Option 2: GSM 07.10 multiplexer (CMUX)

The 3GPP CMUX protocol multiplexes multiple virtual channels onto one UART. Modem implements: channel 1 = AT, channel 2 = PPP, channel 3 = NMEA GPS, etc. Linux kernel has `drivers/tty/n_gsm.c` to demux.

```
# Initiate CMUX on the modem
echo -e 'AT+CMUX=0\r' > /dev/ttymxc3
# Then attach n_gsm to that tty
ldattach GSM0710 /dev/ttymxc3
# /dev/gsmtty1, /dev/gsmtty2, ... appear

# PPP on channel 2
pppd /dev/gsmtty2 115200 ...
# AT commands on channel 1
echo 'AT+CSQ' > /dev/gsmtty1
```

This is the textbook embedded-cellular pattern. The kernel side (n_gsm) is small but the user-space side needs careful sequencing. Many production designs use a small daemon (e.g., `gsmmuxd`) that owns the channels and exposes them as fifos.

103.7 From scratch, a robust auto-reconnect daemon

Real products need PPP to come up at boot, retry on failure, and bring up an LED or report status. Here’s a minimal supervisor:

`celld.sh`:

```
#!/bin/sh
# Cellular link supervisor: power-cycle modem, run pppd, restart on death.

MODEM_TTY=/dev/ttymxc3
PWRKEY_GPIO_CHIP=gpiochip4
PWRKEY_GPIO_LINE=22
LED_GPIO_LINE=23

set_led() { gpioset $PWRKEY_GPIO_CHIP $LED_GPIO_LINE=$1; }
pwr_pulse() {
    gpioset $PWRKEY_GPIO_CHIP $PWRKEY_GPIO_LINE=0
    sleep 1.5
    gpioset $PWRKEY_GPIO_CHIP $PWRKEY_GPIO_LINE=1
}

while true; do
    set_led 0    # off
    pwr_pulse
    sleep 8      # wait for boot
```

```
    # Wait for "SMS Ready" URC (or timeout)
    timeout 30 sh -c "while ! grep -q 'SMS Ready' $MODEM_TTY 2>/dev/null; do sleep 1;
done" \
        < $MODEM_TTY
    set_led 1    # on (modem booted)

    # Bring up PPP
    pppd call cellular updetach
    rc=$?
    echo "pppd exited rc=$rc — restarting in 30 s"
    set_led 0
    sleep 30
done
```

Install as `/etc/init.d/celld` or a systemd unit. The link comes up at boot, recovers from any modem hang by power-cycling PWRKEY, and shows status on an LED. About 30 lines of shell, replacing several hundred lines of vendor C.

103.8 ECM mode, when the UART module has USB-style Ethernet

Newer Cat-1bis modules (Quectel ML302, EC200N) expose **USB CDC-ECM** via a built-in USB-to-UART bridge, so even though they're "UART modems," the data path is actually USB-ECM. Plug the module's microUSB into the host: `usb0` appears as Ethernet, `dhclient usb0` gets an IP, done. The UART carries only AT commands at that point.

This pattern eliminates PPP entirely while keeping the BOM low (one module, no separate USB modem chip). Recommended for any new design that can afford one USB port.

```
dmesg | grep cdc_ether
# cdc_ether 1-1.1:1.1: usb0: register 'cdc_ether' at ...
dhclient usb0
```

103.9 Key Notes

- PPP divides responsibility between modem transport, user-space negotiation and the kernel network interface. A UART stream becomes a link only after that negotiation succeeds.
- AT control and PPP data cannot independently own the same byte stream without a defined multiplexing mechanism or separate channel.
- Serial bandwidth and flow control constrain the data path. A fast cellular radio does not remove a slow UART bottleneck.
- Reconnect policy needs states, deadlines and backoff. Repeated power cycling or overlapping managers can conceal the original failure and prevent stable recovery.

103.10 Pitfalls

- **No hardware flow control.** Above 9600 baud, PPP breaks. Always wire RTS/CTS and pass `crtscts` to pppd.
- **VBAT not VDD.** Powering the modem's RF section from a weak 3.3 V LDO causes TX brownouts → connection drops on transmit.
- **PWRKEY pulse too short.** Some modules want ≥2 s for power-on (Air724) and ≥3 s for power-off. Read the AT command and hardware manuals.
- **AT commands sent during boot.** Spurious ERRORs because the modem isn't ready. Wait for "SMS Ready" URC before talking.
- **Chat script timeout too short.** Default 45 s per step. LTE registration can take 30–60 s on a cold start in poor coverage. Increase `connect-delay` and chat timeouts.
- **APN typo.** Same as Ch 102, wrong APN → PDP context fails. Carriers don't make this discoverable.
- **PPP holds the tty.** Once pppd takes the line, you can't `cat /dev/ttymxc3` to debug AT. Use CMUX or a second UART for debug.
- **NetworkManager fights pppd.** If NM is running, it may auto-take the modem. Disable NM for that tty or use NM exclusively.
- **Default route conflict.** PPP's `defaultroute` plus existing eth0 default route → routing loops. Use `replacedefaultroute` or set metric.
- **MTU 1500 vs operator MTU.** Some operators clamp at 1460 or 1430. Pppd negotiates LCP MTU but TCP path-MTU discovery can still fail. Set `mtu 1430` explicitly if you see hung-large-packet symptoms.
- **CMUX framing errors silent.** Bad CRTS/CTS on the underlying UART causes n_gsm to drop frames. The per-channel ttys appear functional but data is garbled. Verify hardware flow first.

103.11 Going deeper

- **`pppd(8)` manual**: the canonical reference. Every option matters for production reliability.
- **`chat(8)`**: the connect-script DSL.
- **`drivers/net/ppp/ppp_generic.c` + `ppp_async.c`**: the kernel side.
- **`drivers/tty/n_gsm.c`**: 3GPP 27.010 multiplexer.
- **3GPP TS 27.010**: the CMUX specification.

- **SIMCom A7670C AT Commands Manual**: module-specific extensions.
- **`gnokii`**: legacy but instructive open-source modem manager.
- **Ch 102** for USB modem comparison.
- **Ch 104** for low-power NB-IoT/Cat-M1 over UART (same toolchain, different power profile).

> Next chapter: **Chapter 104: NB-IoT / Cat-M1**, the low-power-cellular subset designed for battery-powered IoT sensors.

Chapter 104: NB-IoT / Cat-M1

> **What:** the **low-power-cellular** subset of LTE, **NB-IoT (Cat-NB1/NB2)** and **LTE-M (Cat-M1)**. Modules: **Quectel BC95-G** (NB-IoT only, ~$8), **Quectel BC26** (NB-IoT + GNSS), **SimCom SIM7080G** (NB-IoT + Cat-M1, multi-region). We cover the PHY differences from LTE Cat-1+, the PSM and eDRX features that let a 19 Ah Li-SOCl2 cell run a sensor for up to ten years, the AT command set, MQTT/CoAP profiles tailored for low data rates, and an end-to-end battery-powered sensor that hits 10 mA average over a 1-uplink-per-hour cycle. **PHY:** physical-layer block or chip that converts digital MAC signals to electrical or radio signals.
>
> **Why:** standard LTE (Ch 102/103) wakes the radio, registers, transmits, idles, total energy per uplink ~5 Joules. NB-IoT/Cat-M1, with PSM, parks the radio in “deep sleep” between uplinks while keeping its network registration alive. The result: ~1 J per uplink → years of battery life. This is the technology behind smart water meters, GPS livestock trackers, vending-machine telemetry, and rural emergency call-boxes. If your product needs cellular + battery for years (not days), this is the only path.
>
> **Focus:** PSM is the primary power saver. EDRX is a smaller secondary saving. Use PSM correctly or your battery life estimate is wrong by orders of magnitude. The cycle is: TX, wait for the downlink ACK, then enter PSM. The radio and modem are off, but the network still considers the device registered (IP, PDP context, and security keys are kept). On the next wake, by timer or external GPIO, the device resumes immediately, with no re-registration. The 19 Ah → 10 year math depends on TX every hour at 50 bytes and PSM at <5 µA between. If the AT commands are wrong, the modem stays at 100 mA always-on. Battery life drops by a factor of 1000. **MCU bridge:** Think of Linux GPIO like the same pin set/reset block you used on STM32, but accessed through a kernel subsystem that owns numbering, direction, interrupts, and user-space exposure. **GPIO:** General-Purpose Input/Output, a pin controlled as a digital input, output, or interrupt source.

104.1 NB-IoT vs LTE-M vs LTE Cat-1

	NB-IoT (Cat-NB1)	LTE-M (Cat-M1)	LTE Cat-1
Downlink	27 kbps	1 Mbps	10 Mbps
Uplink	62 kbps	1 Mbps	5 Mbps

	NB-IoT (Cat-NB1)	LTE-M (Cat-M1)	LTE Cat-1
Bandwidth	180 kHz	1.4 MHz	20 MHz
Latency	1.6–10 s	50–200 ms	50 ms
Mobility	stationary or pedestrian	pedestrian + vehicle	full
Voice (VoLTE)	no	yes	yes
PSM (Power Save)	yes (essential)	yes	yes (rare)
eDRX	yes	yes	yes
MCL (max coupling loss)	164 dB	156 dB	142 dB
Penetration	excellent (basements, deep indoor)	good	OK
Cost (per byte)	low (carrier plans optimized for IoT)	mid	high
Module cost	$5–10	$10–20	$25–40

MCL (Maximum Coupling Loss) is the critical metric: NB-IoT's 164 dB MCL means it can reach receivers 20 dB weaker than standard LTE, i.e., it penetrates concrete basements, underground meter chambers, and remote rural cells where Cat-1 has no signal.

Pick guide:

- **NB-IoT (BC95-G)**: your sensor sends < 1 kB/day, stationary, lives in a basement/manhole/field for 10 years. Cheapest.
- **Cat-M1 (SIM7080G)**: needs mobility (asset tracker on a truck), faster latency (alarm system), or occasional firmware update (1 MB OTA over the air in < 30 min vs hours on NB-IoT).
- **Multi-mode (SIM7080G)**: falls back to whichever the carrier supports. Best for products shipped to multiple regions.

104.2 PSM, eDRX, and the power model

```
Normal LTE (always-on RRC connected):
  Active ──────────────────────────────────────── Active
  100 mA, 24/7 = 2.4 Ah/day → 0.3 days on a 19 Ah pack

Cat-1 with idle DRX (sleeping radio, kept-registered):
  Active ─┐    ┌──── 1 s DRX ────┐    ┌─ Active
  10 mA   │5 mA│                 │5 mA│
  ~5 mA average → 0.12 Ah/day → 158 days

NB-IoT with PSM (deep sleep, kept-registered for hours):
  TX ─┐ idle ─┐                                  ┌─ TX
  100 mA  10 mA│         ──── PSM, 5 µA ────     │
  5 µA average → 0.12 mAh/day → > 10 years on 19 Ah
```

The two timers:

- **T3324**: “active timer” after going idle. Modem stays idle-but-listening for T3324 (so a downlink can reach it quickly). Then enters PSM.
- **T3412**: “TAU (Tracking Area Update) timer.” The modem MUST send a TAU before this expires or the network will deregister it. PSM lasts up to T3412.

Negotiation via `AT+CPSMS=1,,,"00000100","00000010"`:

- 4th arg = T3412 (granularity bits): `00000100` = 4 hours
- 5th arg = T3324: `00000010` = 4 s

Carrier-side caveat: the network may grant less than requested. Always `AT+CPSMS?` after activation to see the *granted* values.

eDRX (extended DRX) is an alternative for slightly less power saving but faster downlink response, the modem sleeps for up to 2.92 hours but wakes briefly to listen for paging. Use eDRX when you need occasional downlink commands. PSM when you only need uplinks + scheduled wake.

104.3 Bringing up NB-IoT on the BC95-G

The BC95-G presents a single UART (9600 baud default!). No USB, no PPP for data, uses an “AT-shell” data path (`AT+NSOCR` for socket-create, `AT+NSOSD` for send) which is much simpler than QMI but limited to UDP/TCP/CoAP.

```
# Power on (PWRKEY pulse + wait for "Neul" greeting)
gpioset gpiochip4 22=0; sleep 0.5; gpioset gpiochip4 22=1
sleep 10
cat /dev/ttymxc3 &
# Neul
# +CFUN: 1
# +CPIN: READY
```

```
# +IP: 10.x.x.x

# AT bring-up
echo 'AT' > /dev/ttymxc3
# OK
echo 'AT+NBAND=8' > /dev/ttymxc3      # band 8 (900 MHz, EU NB-IoT)
echo 'AT+CFUN=1' > /dev/ttymxc3       # full functionality
echo 'AT+CGATT=1' > /dev/ttymxc3      # attach
echo 'AT+CSCON?' > /dev/ttymxc3       # +CSCON: 0,1 (idle but registered)

# Configure PSM
echo 'AT+CPSMS=1,,,"00000110","00000010"' > /dev/ttymxc3
# 00000110 = T3412 24 hours; 00000010 = T3324 4 s
# Modem will enter PSM 4 s after data idle, wake every 24 h for TAU

# Send a UDP packet
echo 'AT+NSOCR=DGRAM,17,5683,1' > /dev/ttymxc3       # create UDP socket
# +NSOCR: 0
echo 'AT+NSOST=0,52.34.12.34,5683,5,68656C6C6F' > /dev/ttymxc3   # send "hello"
# +NSOST: 0,5

# Close + sleep
echo 'AT+NSOCL=0' > /dev/ttymxc3
# Modem auto-enters PSM 4 s later
```

The `+NSOST` command takes hex-encoded payload, verbose but simple. For larger payloads use `+NMGS` (Network Message Send) which queues a transmission for the next radio wake.

104.4 Cat-M1 on the SIM7080G

Cat-M1 supports IP networking like Cat-1. You can use PPP (Ch 103) or the simpler AT-socket interface. SimCom calls these `+CASOPEN`, `+CASEND`, `+CIPSEND` depending on firmware version.

```
# Power on, wait for "+CPIN: READY"
echo 'AT+CNMP=38' > /dev/ttymxc3       # LTE only (no GSM fallback)
echo 'AT+CMNB=2' > /dev/ttymxc3        # Cat-M1 preferred
echo 'AT+CGDCONT=1,"IP","iot.t-mobile"' > /dev/ttymxc3
echo 'AT+CNACT=0,1' > /dev/ttymxc3     # PDP context up
# +CNACT: 0,1,"10.x.x.x"

# MQTT publish (built-in MQTT client on SIM7080)
echo 'AT+SMCONF="URL","mqtt.example.com",1883' > /dev/ttymxc3
echo 'AT+SMCONN' > /dev/ttymxc3
echo 'AT+SMPUB="sensor/01",5,1,0' > /dev/ttymxc3
echo '23.45' > /dev/ttymxc3
echo 'AT+SMDISC' > /dev/ttymxc3
echo 'AT+CPSMS=1,,,"00000110","00000010"' > /dev/ttymxc3
echo 'AT+CFUN=0' > /dev/ttymxc3        # force minimum functionality → sleep
```

The built-in MQTT/CoAP clients of SIM7080/BC95 mean you don’t need a TCP stack on Linux, the modem firmware does it. This is invaluable for tiny MCU products, but on Linux you can also run `mosquitto_pub` over PPP if you prefer.

104.5 The 10-year battery sensor, a worked example

Sensor: BME280 temperature/humidity/pressure (Ch 67) read once per hour, value sent via UDP to a cloud receiver. Hardware budget:

- 19 Ah Li-SOCl2 D-cell (Saft LSH20), nominal 19 Ah at 3.6 V, OK to 100 mA pulse with a 47 µF tantalum across.
- BC95-G NB-IoT modem.
- STM32L0 (or i.MX6ULL in suspend) as wake controller.
- BME280 powered from a GPIO (no quiescent).

Per-cycle energy:

Phase	Time	Current	Charge (µAs)
Wake STM32 from STOP, read BME280	50 ms	5 mA	250
Wake modem from PSM	200 ms	80 mA	16,000
RRC reconnect + UL TX (50 bytes via UDP)	1.5 s	80 mA avg	120,000
Idle waiting for DL ACK	500 ms	30 mA	15,000
Re-enter PSM	100 ms	10 mA	1,000
Total per cycle	2.35 s		**~152,000 µAs = 42 µAh**
PSM between cycles (3600 s)	1 hour	5 µA	18,000 µAs = 5 µAh
Total per hour			**47 µAh**

Phase	Time	Current	Charge (µAs)
Per day			**1.13 mAh**
Per year			**413 mAh**
Battery life at 19 Ah usable			**~46 years** (but limited by self-discharge & temperature)

Real-world constraints reduce this to **8–12 years**:

- LSH20 self-discharge ~1 %/year = ~190 mAh/year baseline.
- Cold-temperature operation derates capacity 30 % at –20 °C.
- Network re-registration after every TAU can cost 200+ mAh/year of extra time-on-air.

Ten years on one D-cell with no maintenance is achievable. The engineering is enforcing PSM duty cycle on every wake.

104.6 Key Notes

- NB-IoT and LTE-M offer different coverage, latency and mobility tradeoffs. The selected module and operator must support the intended service.
- PSM and eDRX change availability as well as power consumption. Requested sleep parameters are not necessarily the parameters granted by the network.
- A battery-life model must include complete wake, registration, transfer, retry and sleep behavior. Modem sleep current alone is not a system energy budget, especially with an always-on Linux host.
- Application sessions may expire while the modem sleeps. Reconnection and delayed downlink behavior need an explicit protocol policy rather than an assumption of continuous reachability.

104.7 Pitfalls

- **PSM not granted by carrier.** Your `AT+CPSMS?` after registration shows `T3412 = default`. Some carriers (US Verizon) restrict PSM to certain plans. Without it, your “10 year battery” sensor is a “3 week” sensor.

- **NB-IoT not deployed by your carrier.** US carriers (Verizon, AT&T) have NB-IoT in some bands but not all. Check before committing.
- **Wrong band for the region.** NB-IoT band 8 (EU 900) doesn't work in NA. Need band 5/12/13 (US). Per-region SKUs avoid this.
- **PSM wakes too often for TAU.** T3412 of 24 h is the max in most networks. Carriers may grant only 1 h. Recalculate battery budget accordingly.
- **Modem-side TCP timeout drops the link.** Long PSM intervals + an open TCP socket = the network's NAT entry expires. Always close sockets before sleep. Reopen on wake. UDP is friendlier for this.
- **TX power higher than expected.** NB-IoT can transmit at up to +23 dBm (200 mW) for coverage extension. Battery budget collapses if you assume 100 mW. Measure real currents.
- **Voltage drop at cold temp.** Li-SOCl2 cells passivate. First pulse after a long sleep can drop to 2.7 V momentarily. Add a "wake-warmer", a 100 ms low-current pre-pulse before the modem TX, to depassivate.
- **No coverage indicator.** Without a +CEREG event listener, your firmware doesn't know it lost the network until a TX fails. Subscribe to URCs.
- **Firmware bugs in PSM transition.** Some BC95 firmware revisions enter PSM but don't wake on the GPIO trigger. Update modem firmware before relying on PSM hardware wake.
- **Wrong APN for IoT plan.** IoT SIMs often use a separate APN (e.g., `iot.1nce.net`). Using the default consumer APN may register but fail to get an IP.

104.8 Going deeper

- **Quectel BC95-G AT Commands Manual**: Quectel's NB-IoT extensions (`AT+NSOCR`, `AT+NMGS`).
- **SimCom SIM7080G AT Commands Manual**: built-in MQTT/HTTP/CoAP clients.
- **3GPP TS 23.682**: the PSM and eDRX specification.
- **`drivers/net/wwan/` modern subsystem**: newer kernel WWAN drivers for some Cat-M1 modules.
- **1NCE / Hologram / Soracom**: IoT-focused carriers with NB-IoT/Cat-M1 plans and global SIMs.
- **Saft LSH20 datasheet**: the canonical 19 Ah D-cell for long-life IoT.
- **Ch 102 / Ch 103** for LTE Cat-1 baseline comparison.

> Next chapter: **Chapter 105: RFID / NFC**, MFRC522 + PN532 for tag reading.

Chapter 105: RFID / NFC

> **IRQ:** interrupt request, the signal path that tells the CPU or interrupt controller that hardware needs service. **MCU bridge:** Think of an IRQ like an EXTI/NVIC interrupt path, except Linux splits the hard interrupt from deferred work and must share lines across drivers.

> **What: 13.56 MHz HF RFID and NFC**, the technology behind contactless access cards, transit passes, phone Wallet, and "tap-to-pair." Three chips compared: **NXP MFRC522** (the ubiquitous Arduino-clone SPI/I²C/UART, ISO 14443A only), **NXP PN532** (more capable: 14443A/B + FeliCa, NFC initiator and target), **ST25R3911** (longer read range, high-end). On the i.MX6ULL we read tag UIDs over SPI, authenticate a Mifare Classic 1K block, walk the kernel `pn533` driver as the canonical mainline NFC stack reference, write a 200-line user-space MFRC522 driver from scratch, then bring up `libnfc` + `neard` for high-level NFC.
>
> **Why:** Access control is one of the most common embedded Linux applications, door readers, time-and-attendance kiosks, equipment-rental lockers, EV-charger user identification. NFC tagging extends to smart-home pairing (Tap to Wi-Fi), industrial asset tracking, and consumer-product authenticity verification. The chips are cheap (MFRC522 modules cost about $1). The standards are real. The security is half-broken, Mifare Classic was cracked in 2008. And Linux has an NFC subsystem (`net/nfc/`) that most engineers do not know exists.
>
> **Focus:** At 13.56 MHz, RFID and NFC use inductive coupling. The reader's antenna generates a magnetic field. That field powers the tag's IC, and the tag sends data back by modulating its load on the field. The reader chip uses a fixed sequence of register writes: configure carrier ON, set framing (Miller-encoded for tag→reader, Manchester for reader→tag), issue protocol-level commands (REQA, ATQA, ANTICOLL, SELECT, AUTH, READ_BLOCK), parse responses. Antenna matching is critical, a 5 mm misalignment between the reference design's antenna loop and yours = 30 % less read range.
>
> **Tooling.** This chapter uses `libnfc-bin` (`nfc-list`, `nfc-mfultralight`), `neard`. Offensive-research only: `mfoc`, `mfcuk`.
>
> - **Ubuntu-base (target):** `apt install libnfc-bin libnfc-dev neard`
> - **Buildroot:** `BR2_PACKAGE_LIBNFC=y BR2_PACKAGE_NEARD=y` **Buildroot:** a configuration-driven build system that produces a complete root filesystem and related images.

> - Full per-tool reference: Userspace tooling appendix. **MCU bridge:** Think of the rootfs as the firmware image's file-backed runtime environment. On an MCU you link everything into flash. On Linux, programs and config live in this mounted tree. **rootfs:** root filesystem, the directory tree mounted at / that contains /bin, /etc, /dev, and libraries.

105.1 Chip comparison

	MFRC522	PN532	ST25R3911
Interfaces	SPI / I²C / UART	SPI / I²C / UART (HSU)	SPI
Standards	ISO 14443A	14443A/B, FeliCa, NFC initiator + target	14443A/B, FeliCa, longer range, V (ISO 15693)
Range (with standard tag)	3–5 cm	4–6 cm	8–12 cm
TX power	~100 mW	~100 mW	~1.4 W (with external antenna)
Mifare Classic Crypto1	yes	yes	yes
NFC tag emulation	no	yes	yes
P2P NFC	no	yes	yes
Kernel mainline driver	none	pn533 (drivers/nfc/)	none
Cost (module)	$1–3	$5–10	$15–25

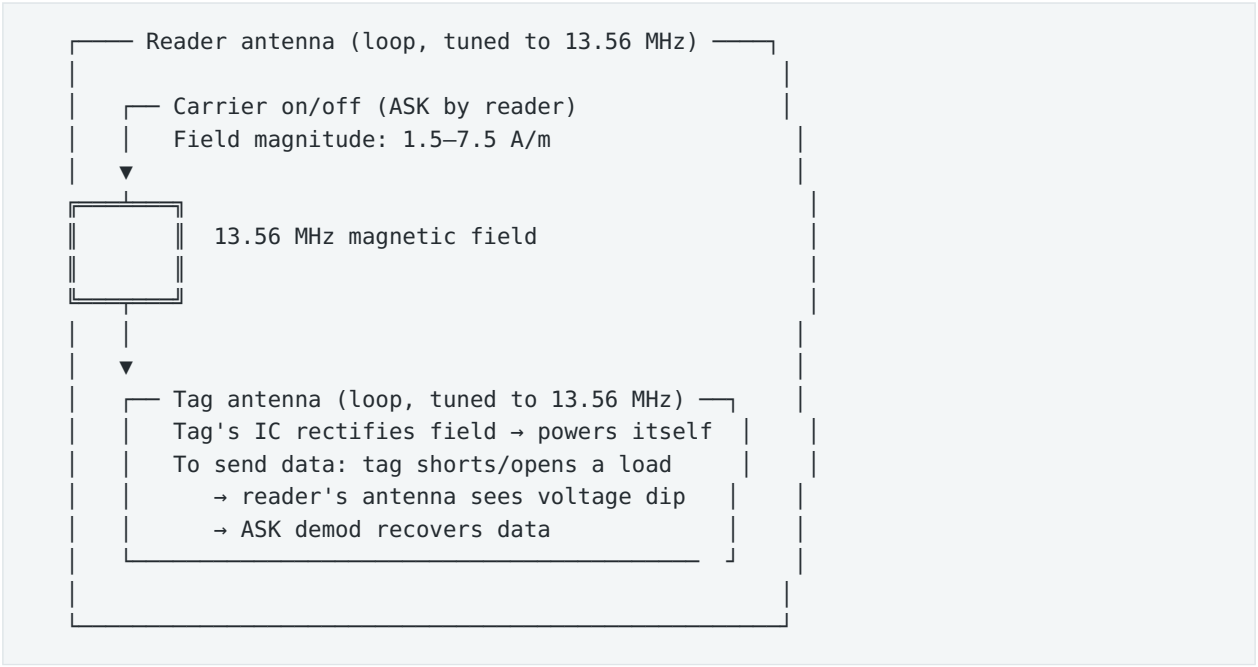
Pick guide:

- **MFRC522**: door reader, asset tracker, anything that just needs a UID + Mifare R/W. Cheap and adequate for most access-control work.
- **PN532**: when you need NFC P2P (handover, peer transfer), phone-emulation, or FeliCa support. Mainline kernel driver exists.

- **ST25R3911**: when you need range (10 cm+), ISO 15693 (long-range industrial RFID), high reliability.

105.2 How 13.56 MHz RFID actually works

Two halves: reader and tag.



The tag is **passive**, no battery. The reader's field both powers the tag's chip and carries communication. Tag-to-reader uses load modulation with an 847.5 kHz subcarrier (Type A). Reader-to-tag uses direct ASK modulation, at either 100 % or 10 % depth.

The communication frames the **ISO 14443 anticollision and select protocol**:

```
Reader: REQA (request answer)         0x26  (Type A: short frame, 7 bits)
Tag:    ATQA (answer to request)      2 bytes — tag's basic capability
Reader: ANTICOLL (cascade level 1)    0x93 0x20
Tag:    UID CL1 + BCC                 5 bytes — 4 UID bytes + BCC checksum
Reader: SELECT (cascade level 1)      0x93 0x70 UID...
Tag:    SAK (select acknowledge)      1 byte — Mifare Classic, Mifare Ultralight, ...

For Mifare Classic, then:
Reader: AUTH(Key, Sector)             0x60 0x00 ... Crypto1 challenge-response
Tag:    encrypted from now on
Reader: READ_BLOCK 0x04                read block 4
Tag:    16 bytes (encrypted)
```

The MFRC522 implements ISO 14443A framing and timing in silicon. You don't bit-bang. You issue protocol-level commands and the chip handles the air interface.

105.3 MFRC522 register and command summary

The MFRC522 has 64 registers, 16 commands. Key ones:

Register	Addr	Purpose
CommandReg	0x01	issue command (Idle, Transceive, Authent, …)
ComIEnReg / DivIEnReg	0x02 / 0x03	interrupt enable
ComIrqReg	0x04	interrupt status (RxIRq, TxIRq, ErrIRq, TimerIRq)
ErrorReg	0x06	error flags (collision, parity, CRC)
FIFODataReg	0x09	TX/RX FIFO (64 bytes)
FIFOLevelReg	0x0A	FIFO byte count
BitFramingReg	0x0D	how many bits in last byte
ModeReg	0x11	CRC preset, polarity
TxControlReg	0x14	antenna driver enable
VersionReg	0x37	0x92 = MFRC522 v2.0

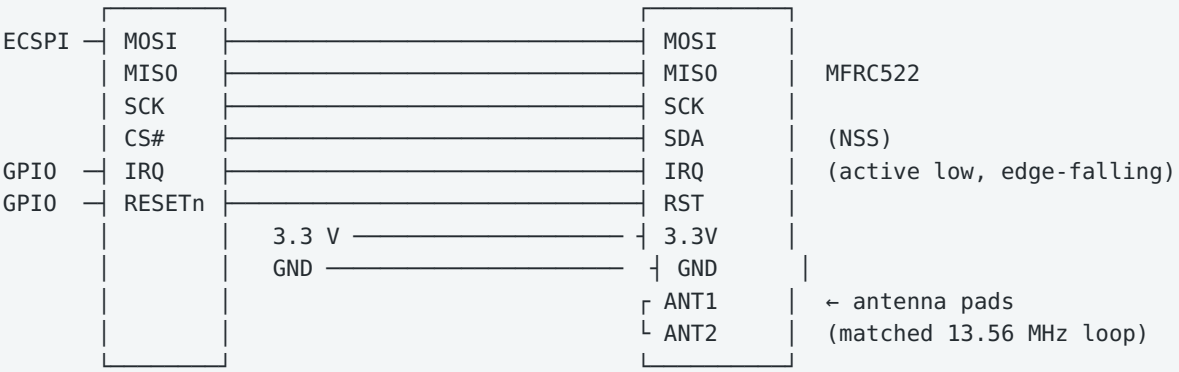
Command	Hex	Meaning
Idle	0x00	stop current command
Transceive	0x0C	send FIFO bytes + receive response
Authent	0x0E	Mifare Classic Crypto1 authentication
SoftReset	0x0F	reset the chip

The transceive flow:

1. Write payload into FIFODataReg (e.g., REQA = 0x26).
2. Write BitFramingReg with 7 (REQA is a 7-bit short frame).
3. Write CommandReg = Transceive.
4. Write BitFramingReg = 0x80 (set StartSend bit), chip TXes + receives response.
5. Wait for RxIRq or TimerIRq.
6. Read FIFODataReg until FIFOLevelReg reads 0, that's the tag's response.

That sequence, embedded in `MFRC522_TransceiveData()`, is what every Arduino library wraps. Below we write it ourselves.

105.4 Wiring MFRC522 (SPI) to the i.MX6ULL



The antenna is the critical part. A bad-design module gets 1 cm read range. The cheap-but-correct modules (the green "RC522" boards with the small ferrite antenna trace) get 3–5 cm. Do not wind your own antenna unless you have an impedance analyser. Buy a tuned module instead.

105.5 How the kernel pn533 driver works

The kernel NFC subsystem (`net/nfc/`) provides netlink socket interface + the device-specific drivers. PN532 is in `drivers/nfc/pn533/`.

```
User-space (neard, libnfc, your app)
     │ AF_NFC netlink
     ▼
net/nfc/                      <-- core, target table, polling
     │
     ▼
drivers/nfc/pn533/             <-- chip-specific
     │                         <-- USB or UART or I²C variant
```

```
        ▼
   USB / serdev / i2c bus
```

Walk of `pn533_send_cmd_frame_async()` (paraphrased):

```
static int pn533_send_cmd_frame_async(struct pn533 *dev,
                                      struct pn533_frame *out_frame,
                                      struct pn533_frame *in_frame,
                                      ...) {
    /* Build a PN532 frame: preamble + start + length + cmd + data + DCS + postamble */
    pn533_build_cmd_frame(out_frame, cmd, params, params_len);

    /* Submit URB (for USB) or write tty (for UART) */
    return dev->phy_ops->send_frame_async(dev, out_frame, in_frame, ...);
}
```

The phy_ops separation lets the same protocol layer work for USB, I²C, UART variants of the PN532.

Polling for tags:

```
static int pn533_send_poll_frame(struct pn533 *dev) {
    /* InListPassiveTarget command, 1 target max, baudrate 106 kbps Type A */
    pn533_build_cmd_frame(frame, PN533_CMD_IN_LIST_PASSIVE_TARGET,
                          (u8[]){ 1, 0 }, 2);
    return pn533_send_cmd_frame_async(dev, frame, &dev->resp_frame, ...);
}
```

When a tag enters the field, the response includes UID, ATQA, SAK. `pn533` calls into `net/nfc/core.c::nfc_targets_found()` which broadcasts netlink events to user-space. Neard receives the events and runs the higher-level NDEF/handover state machines.

User-space:

```
modprobe pn533_usb       # or pn533_i2c, pn533_uart
nfctool list             # via libnfc
neardctl tags            # via neard
```

This is the “proper” Linux path. Most projects skip the kernel NFC stack. They use `libnfc` directly against `/dev/spidev`, or one of the MFRC522 user-space drivers shown below.

105.6 From scratch, user-space MFRC522 driver

mfrc522_min.c (compressed. Full ~400 lines):

```
/* Minimal MFRC522 driver: detect tag, read UID, dump Mifare Classic block 0.
 * Build: gcc -o mfrc522 mfrc522_min.c -lgpiod
 */
#include <fcntl.h>
#include <stdint.h>
```

```
#include <stdio.h>
#include <string.h>
#include <unistd.h>
#include <sys/ioctl.h>
#include <linux/spi/spidev.h>
#include <gpiod.h>

#define REG_COMMAND       0x01
#define REG_COM_IRQ       0x04
#define REG_ERROR         0x06
#define REG_FIFO_DATA     0x09
#define REG_FIFO_LEVEL    0x0A
#define REG_CONTROL       0x0C
#define REG_BIT_FRAMING   0x0D
#define REG_MODE          0x11
#define REG_TX_MODE       0x12
#define REG_RX_MODE       0x13
#define REG_TX_CONTROL    0x14
#define REG_TX_AUTO       0x15
#define REG_T_MODE        0x2A
#define REG_T_PRESCALER   0x2B
#define REG_T_RELOAD_H    0x2C
#define REG_T_RELOAD_L    0x2D
#define REG_VERSION       0x37

#define CMD_IDLE          0x00
#define CMD_TRANSCEIVE    0x0C
#define CMD_AUTHENT       0x0E
#define CMD_SOFTRESET     0x0F

#define PICC_REQA         0x26
#define PICC_ANTICOLL_CL1 0x93
#define PICC_AUTH_A       0x60
#define PICC_AUTH_B       0x61
#define PICC_READ         0x30
#define PICC_HALT         0x50

static int spi_fd;
static struct gpiod_line *rst;

static uint8_t reg_read(uint8_t r) {
    uint8_t tx[2] = { 0x80 | ((r << 1) & 0x7E), 0 }, rx[2];
    struct spi_ioc_transfer t = { .tx_buf=(unsigned long)tx, .rx_buf=(unsigned long)rx,
                                  .len=2, .speed_hz=1000000 };
    ioctl(spi_fd, SPI_IOC_MESSAGE(1), &t);
    return rx[1];
}
static void reg_write(uint8_t r, uint8_t v) {
    uint8_t tx[2] = { (r << 1) & 0x7E, v };
    struct spi_ioc_transfer t = { .tx_buf=(unsigned long)tx, .len=2, .speed_hz=1000000 };
    ioctl(spi_fd, SPI_IOC_MESSAGE(1), &t);
}
static void set_bits(uint8_t r, uint8_t mask) { reg_write(r, reg_read(r) | mask); }
static void clr_bits(uint8_t r, uint8_t mask) { reg_write(r, reg_read(r) & ~mask); }

static void antenna_on(void) { set_bits(REG_TX_CONTROL, 0x03); }

static int transceive(uint8_t cmd, const uint8_t *send, int send_len,
                      uint8_t *recv, int *recv_len, uint8_t last_bits) {
```

```
    reg_write(REG_COMMAND, CMD_IDLE);
    reg_write(REG_COM_IRQ, 0x7F);                /* clear all flags */
    reg_write(REG_FIFO_LEVEL, 0x80);             /* flush FIFO */
    for (int i = 0; i < send_len; i++) reg_write(REG_FIFO_DATA, send[i]);
    reg_write(REG_COMMAND, cmd);
    if (cmd == CMD_TRANSCEIVE) set_bits(REG_BIT_FRAMING, 0x80);   /* StartSend */

    uint8_t irq;
    for (int i = 0; i < 200; i++) {              /* ~40 ms timeout */
        irq = reg_read(REG_COM_IRQ);
        if (irq & 0x30) break;                   /* RxIRq or IdleIRq */
        if (irq & 0x01) return -2;               /* TimerIRq */
        usleep(200);
    }
    clr_bits(REG_BIT_FRAMING, 0x80);

    if (reg_read(REG_ERROR) & 0x13) return -1;   /* BufferOvfl, ParityErr, ProtocolErr */

    int n = reg_read(REG_FIFO_LEVEL);
    if (n > *recv_len) n = *recv_len;
    for (int i = 0; i < n; i++) recv[i] = reg_read(REG_FIFO_DATA);
    *recv_len = n;
    return 0;
}

static int picc_request(uint8_t *atqa) {
    reg_write(REG_BIT_FRAMING, 0x07);            /* 7 bits last byte */
    uint8_t cmd = PICC_REQA;
    int n = 2;
    return transceive(CMD_TRANSCEIVE, &cmd, 1, atqa, &n, 7);
}

static int picc_anticoll(uint8_t *uid_out) {
    reg_write(REG_BIT_FRAMING, 0x00);
    uint8_t cmd[2] = { PICC_ANTICOLL_CL1, 0x20 };
    uint8_t resp[5];
    int n = 5;
    int r = transceive(CMD_TRANSCEIVE, cmd, 2, resp, &n, 0);
    if (r) return r;
    uint8_t bcc = resp[0] ^ resp[1] ^ resp[2] ^ resp[3];
    if (bcc != resp[4]) return -3;
    memcpy(uid_out, resp, 4);
    return 0;
}

static void mfrc_init(void) {
    /* Hardware reset */
    gpiod_line_set_value(rst, 0); usleep(2000);
    gpiod_line_set_value(rst, 1); usleep(50000);

    reg_write(REG_COMMAND, CMD_SOFTRESET);
    usleep(50000);

    /* TX/RX modulation: 100% ASK, CRC preset 6363 */
    reg_write(REG_T_MODE, 0x8D);
    reg_write(REG_T_PRESCALER, 0x3E);
    reg_write(REG_T_RELOAD_L, 30);
    reg_write(REG_T_RELOAD_H, 0);
    reg_write(REG_TX_AUTO, 0x40);
```

```
    reg_write(REG_MODE, 0x3D);
    antenna_on();
}

int main(void) {
    spi_fd = open("/dev/spidev0.0", O_RDWR);
    uint8_t mode = SPI_MODE_0;
    ioctl(spi_fd, SPI_IOC_WR_MODE, &mode);

    struct gpiod_chip *chip = gpiod_chip_open("/dev/gpiochip0");
    rst = gpiod_chip_get_line(chip, 24);
    gpiod_line_request_output(rst, "mfrc-rst", 1);

    mfrc_init();
    printf("MFRC522 v=0x%02X\n", reg_read(REG_VERSION));

    printf("Present a tag.\n");
    for (;;) {
        uint8_t atqa[2], uid[4];
        if (picc_request(atqa) == 0 && picc_anticoll(uid) == 0) {
            printf("UID: %02X %02X %02X %02X  ATQA: %02X %02X\n",
                   uid[0], uid[1], uid[2], uid[3], atqa[0], atqa[1]);
            /* HALT the tag so we don't keep re-reading */
            uint8_t halt[2] = { PICC_HALT, 0 };
            int n = 0;
            transceive(CMD_TRANSCEIVE, halt, 2, NULL, &n, 0);
            sleep(1);
        }
        usleep(100000);
    }
}
```

That’s the entire reader: ~150 lines for the core, ~50 for SPI/GPIO setup. Add Crypto1 auth + block read (another ~100 lines) and you have a working Mifare Classic 1K reader/writer.

> **MCU bridge:** Think of Linux GPIO like the same pin set/reset block you used on STM32, but accessed through a kernel subsystem that owns numbering, direction, interrupts, and user-space exposure. **GPIO:** General-Purpose Input/Output, a pin controlled as a digital input, output, or interrupt source.

What the framework hides:

- The MFRC522’s transceive command does both TX and RX in one go using its FIFO.
- The 7-bit short frame for REQA is configured via BitFramingReg before the transceive, easy to forget.
- The BCC byte after the 4-byte UID is a sanity check. Modules that compute BCC wrong report a passing read but with corrupt UIDs.
- The HALT command must be sent before the next REQA, or the tag won’t respond again (it’s already “active”).

105.7 Mifare Classic security, broken but still used

The Crypto1 cipher (proprietary, never published, reverse-engineered in 2008) protects Mifare Classic blocks. **Crypto1 is fundamentally broken**, anyone with a $30 reader and `mfoc`/`mfcuk` tools can dump all keys from a card in ~5 minutes.

Despite this, Mifare Classic dominates legacy access systems because:

- Vendor lock-in (the building's reader infrastructure).
- The keys you don't know stay encrypted enough for a casual attacker.

Modern systems use **Mifare DESFire EV2/EV3** (AES-128, properly designed) or **iCLASS SE**. If you're designing new, use DESFire. If you're integrating with existing infrastructure, you may be stuck with Classic.

For DESFire: the MFRC522 supports the framing but not the AES. You implement Crypto on the host side (or use a chip like ST25R3911 with hardware AES).

105.8 Bringing up libnfc + neard (the kernel-stack path)

For a PN532 module (USB or UART), the proper path:

```
modprobe pn533_i2c                                    # or _usb, depending on attach
apt install libnfc-bin libnfc-dev neard

# Identify devices
nfc-list
# NFC device: pn532_uart:/dev/ttymxc1 opened
# 1 ISO14443A passive target(s) found:
# ATQA (SENS_RES): 00 04
# UID (NFCID1): 12 34 56 78
# SAK (SEL_RES): 08

# Read NDEF (NFC Data Exchange Format) tags
nfc-mfultralight r tag.dump

# neard for high-level NFC: tap to pair, web URLs
neard &
neardctl tags
neardctl record-uri https://example.com
```

`neard`'s big advantage: implements the NFC Forum's connection-handover spec, so a "tap-to-pair-Wi-Fi" works without you writing any of it.

105.9 Key Notes

- Reader transport, tag protocol and application authorization are separate layers. Reading a tag UID identifies a presented value, not proof of an authorized person.
- Anticollision and tag selection govern communication when several tags are present. Different tag families can require different authentication and data formats.
- RF coupling depends on antenna design, surrounding material and placement. A functioning SPI register interface does not qualify read range.
- Keys and access-control policy need a lifecycle independent of demonstration data. UID-only or legacy-card examples are not a complete security design.

105.10 Pitfalls

- **Module quality varies wildly.** Cheap MFRC522 modules have detuned antennas. 1 cm range vs the 5 cm spec. Buy from known suppliers or accept the variance.
- **CRC preset wrong.** ModeReg = 0x3D sets CRC preset 6363h. Wrong value = tag rejects every command.
- **Tag already in HALT state.** A tag that’s been HALTed by a prior reader cycle won’t respond to REQA, use WUPA (0x52) instead.
- **Crypto1 timing.** The Authent command has tight timing requirements. On slow SPI buses (<1 MHz) you can miss the window. Bump SPI to 4+ MHz.
- **DESFire mistaken for Mifare Classic.** DESFire ATQA is 0x4403 + SAK 0x20. Trying Crypto1 on it fails. Detect by SAK and switch protocols.
- **Multiple tags in field.** ISO 14443 anticollision picks one tag. The others starve. The cascade-level-1/2/3 dance handles 7- or 10-byte UIDs (NTAG2xx). Simple drivers fail on 7-byte tags by ignoring CL2.
- **Mifare Classic key not default.** Most production cards have rotated keys. The manufacturer/vendor key is in the access-control software’s database. Don’t expect FF×6 to work on a real building card.
- **Cloning detection.** Mifare Classic clones (UID-changeable “magic cards”) are common. Security systems should check that re-reads are consistent and that the AppId block matches expectations, not just the UID.
- **Antenna detuning by metal nearby.** Mounting an MFRC522 against a metal backplate detunes the 13.56 MHz LC tank → no read. Use a ferrite shield (the NXP CLRC663 reference design shows the layout).

105.11 Going deeper

- **NXP MFRC522 Datasheet** + **MF1S50yyX_V1** (Mifare Classic 1K product data sheet).
- **NXP PN532 User Manual** (UM0701-02), the canonical reference.
- **`drivers/nfc/`** and **`net/nfc/`** in the kernel.
- **`libnfc`** **(https://github.com/nfc-tools/libnfc)**: covers most readers, including MFRC522 via the `mfrc522_spi` driver.
- **`mfoc`** / **`mfcuk`**: Mifare Classic key recovery (offensive security. Useful for understanding what *not* to rely on).
- **NFC Forum specifications**: NDEF, RTD, Connection Handover.
- **ISO/IEC 14443** (Type A and Type B), the air-interface standard.
- **`neard`** **daemon**: high-level NFC stack on Linux.
- **Ch 95**: Bluetooth handover via NFC (the “tap to pair” use case bridges the two stacks).

> Next chapter: **Chapter 106: Fingerprint sensors**, UART command protocols for R503, FPM10A, AS608.

Chapter 106: Fingerprint sensors

What: standalone fingerprint modules that do the imaging + matching internally and expose a UART command protocol. **Grow R503** (capacitive, ring-LED indicator, the modern favorite), **FPM10A / AS608** (optical, classic, cheap), **GT-521F** (capacitive, larger sensor area). This chapter also covers the libfprint path for USB fingerprint scanners, the laptop-style readers. On the i.MX6ULL we wire R503 to UART, walk the proprietary 9-byte command framing protocol, enroll a template, perform a 1:N match, store templates in flash, and integrate with PAM for “password + fingerprint” 2-factor login.

Why: Fingerprint is the dominant biometric for low-friction access control. Face recognition has privacy and enrolment problems. Iris is expensive. UART modules are *self-contained matchers*, you don’t deal with raw image processing, template extraction, or the messy algorithms. You enroll, you match, you get a yes/no + template ID. Common applications: smart locks, time-and-attendance kiosks, equipment-checkout terminals, secure-area entry. On Linux, plug the module’s UART into the i.MX6ULL, write a 300-line driver, and you have biometric auth.

Focus: modules are stateful, they remember which template ID is enrolled in which slot, and protocol commands operate on that state. Enrollment is a three-step sequence: capture image 1, capture image 2, combine into template, store at chosen ID. Matching is one command: capture, compare to all stored, return ID + score. The framing protocol is trivial (header + length + cmd + data + checksum), but the **state model** (which template is at slot N, what happens if you re-enroll into an occupied slot, how power-cycle affects state) is where most integrations fail.

Tooling. The UART examples use `picocom`. The PAM integration example needs the PAM development headers to build a custom module.

- **Ubuntu-base (target):** `apt install picocom libpam0g-dev`
- **Buildroot:** `BR2_PACKAGE_PICOCOM=y BR2_PACKAGE_LINUX_PAM=y` **Buildroot:** a configuration-driven build system that produces a complete root filesystem and related images.
- Full per-tool reference: Userspace tooling appendix. **MCU bridge:** Think of the rootfs as the firmware image’s file-backed runtime environment. On an MCU you link everything into flash. On Linux, programs and config live in this mounted tree. **rootfs:** root filesystem, the directory tree mounted at / that contains /bin, /etc, /dev, and libraries.

106.1 Module comparison

	Grow R503	FPM10A / AS608	GT-521F32
Sensor type	capacitive	optical	capacitive
Sensor area	21.5 mm dia	14 × 18 mm	16 × 16 mm
Resolution	508 DPI	508 DPI	450 DPI
Template storage	200 templates	162 (FPM10A) / 1000 (AS608)	200
Match time	~1.2 s	~1.0 s	~1.5 s
FAR (false accept)	< 0.001 %	< 0.001 %	< 0.001 %
FRR (false reject)	< 1 %	< 2 %	< 1 %
Interface	UART (TTL 3.3 V)	UART (TTL 3.3 V or 5 V)	UART
LED ring	yes (RGB, programmable)	no	no
Cost (module)	$25–35	$10–18	$20–30
Form factor	round button	flat sensor + PCB	flat module

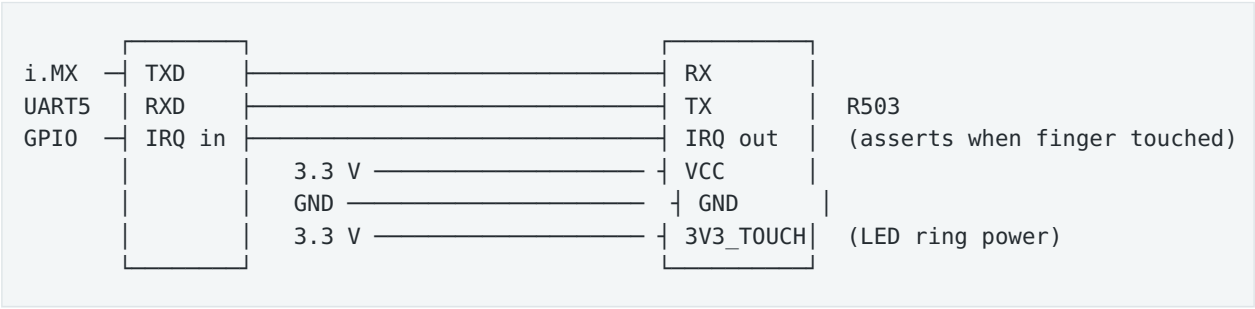
Pick guide:

- **R503**: for any consumer-facing product. The capacitive sensor is dirt/wet-tolerant. The LED ring gives user feedback (red = denied, green = approved, blue = scanning). Choose the R503 when the product needs a polished user experience.
- **AS608**: when BOM matters and the form factor allows an optical sensor (which can’t get wet). Bigger storage (1000 templates) than R503.
- **GT-521F**: for cases where you need a flat panel sensor (door reader behind a 1 mm glass).

106.2 Wiring an R503 to the i.MX6ULL

R503 has a 6-wire cable: VCC, GND, TX, RX, IRQ, 3V3_TOUCH_OUT.

> **MCU bridge:** Think of an IRQ like an EXTI/NVIC interrupt path, except Linux splits the hard interrupt from deferred work and must share lines across drivers. **IRQ:** interrupt request, the signal path that tells the CPU or interrupt controller that hardware needs service.



The IRQ pin (active high) goes high when a finger touches the sensor. Wire to a GPIO and you can wake the i.MX6ULL from sleep on touch, critical for battery-powered locks.

> **MCU bridge:** Think of Linux GPIO like the same pin set/reset block you used on STM32, but accessed through a kernel subsystem that owns numbering, direction, interrupts, and user-space exposure. **GPIO:** General-Purpose Input/Output, a pin controlled as a digital input, output, or interrupt source.

106.3 The packet framing protocol

All commands use the same frame:

```
   [Header MSB] [Header LSB] [Address 4 bytes] [Packet ID] [Length MSB] [Length LSB]
[Data...] [Sum MSB] [Sum LSB]
       0xEF         0x01     0xFF 0xFF 0xFF 0xFF     ID         N             N
payload    checksum
```

- **Header**: fixed `0xEF01`.
- **Address**: module address. Default `0xFFFFFFFF` (broadcast). Can be changed.
- **Packet ID**: frame type:
  - `0x01` = command
  - `0x02` = data (multi-part follow-up)
  - `0x07` = ack/response from module

    - `0x08` = end-of-data

- **Length**: payload + checksum length in bytes.
- **Data**: first byte is the command code. Rest is command-specific payload.
- **Checksum**: sum of bytes from Packet ID to last data byte, modulo 65536.

Example: `GetImg` command (capture an image of the finger on the sensor):

```
EF 01  FF FF FF FF  01  00 03  01  00 05
 ↑       ↑           ↑     ↑    ↑    ↑
hdr   address       cmd  len=3 GetImg checksum
```

Response (success):

```
EF 01  FF FF FF FF  07  00 03  00  00 0B
                     ↑     ↑    ↑    ↑
                   ack  len=3 OK   checksum
```

Confirmation codes:

- `0x00` = success
- `0x01` = packet error
- `0x02` = no finger
- `0x03` = failed to enroll
- `0x07` = template generation fail (too few features)
- `0x09` = no match (1:1 verify)
- `0x0A` = no template found (1:N search)

Key commands:

Hex	Name	Purpose
0x01	GetImg	capture image from sensor
0x02	GenChar	extract feature template from image, into char buffer 1 or 2
0x05	RegModel	combine char buffers 1+2 into a template

Hex	Name	Purpose
0x06	Store	store template in flash at PageID
0x07	LoadChar	load template from flash into char buffer
0x04	Search	match char buffer against all stored, return PageID + score
0x1A	Match	compare char buffers 1 and 2 (1:1 verify)
0x35	AuraLedConfig	R503-only: configure LED ring (color, mode)
0x17	ReadTemplateIndex	bitmap of which slots have stored templates
0x0C	DeleteChar	delete templates in a range
0x0D	Empty	wipe all templates

106.4 Enrollment, the 3-step dance

```
1. GetImg                  ← user puts finger on sensor
2. GenChar(buffer=1)       ← extract features → char buffer 1
3. (user lifts finger, AT-LEAST-200-MS-pause, presses again)
4. GetImg                  ← capture again
5. GenChar(buffer=2)       ← extract features → char buffer 2
6. RegModel                ← combine buffers 1+2 into a robust template
7. Store(slot=N)           ← persist to flash at slot N
```

Double-capture improves robustness. Fingerprints do not sample identically each time, so the union of two captures gives better matching tolerance. Some implementations capture 3 times for even better quality (look for FPM10A's `Enroll` shortcut command that does it all in one).

If GenChar returns "too few features," the user's finger was wet, dirty, or off-center. Retry.

106.5 Matching, 1:N search

```
   1. GetImg                  ← user puts finger
   2. GenChar(buffer=1)
   3. Search(buffer=1, start=0, count=200)
       → returns PageID (slot) + Score, or "no match"
```

Score is 0..2000 (higher = better match). The chip's threshold is typically 50, above that, declare a match. Tune for FAR/FRR trade-off.

106.6 From scratch, R503 driver in C

r503.c (compressed):

```
/* Minimal R503 fingerprint driver.
 * Commands: enroll, search, delete, list.
 */
#include <fcntl.h>
#include <stdint.h>
#include <stdio.h>
#include <string.h>
#include <termios.h>
#include <unistd.h>

static int uart_fd;

static int send_pkt(uint8_t pid, const uint8_t *data, int n) {
    uint8_t hdr[9] = {
        0xEF, 0x01,                                 /* header */
        0xFF, 0xFF, 0xFF, 0xFF,                     /* address */
        pid,                                         /* packet id */
        (n + 2) >> 8, (n + 2) & 0xFF,               /* length */
    };
    uint16_t sum = pid + ((n + 2) >> 8) + ((n + 2) & 0xFF);
    for (int i = 0; i < n; i++) sum += data[i];
    write(uart_fd, hdr, 9);
    write(uart_fd, data, n);
    uint8_t tail[2] = { sum >> 8, sum & 0xFF };
    return write(uart_fd, tail, 2);
}

static int recv_pkt(uint8_t *data_out, int maxlen) {
    uint8_t hdr[9];
    /* Read until we have a valid header */
    for (int i = 0; i < 9; ) {
        int n = read(uart_fd, &hdr[i], 9 - i);
        if (n <= 0) return -1;
        i += n;
    }
    if (hdr[0] != 0xEF || hdr[1] != 0x01) return -2;
    uint16_t len = (hdr[7] << 8) | hdr[8];
    if (len < 2 || len > maxlen + 2) return -3;
    uint8_t payload[256];
    for (int i = 0; i < len; ) {
```

```
        int n = read(uart_fd, &payload[i], len - i);
        if (n <= 0) return -4;
        i += n;
    }
    /* Verify checksum (skipped for brevity in this listing) */
    memcpy(data_out, payload, len - 2);
    return len - 2;        /* returns payload (incl confirm byte) */
}

static int cmd(uint8_t op, const uint8_t *args, int argc, uint8_t *resp, int max) {
    uint8_t buf[256];
    buf[0] = op;
    if (args) memcpy(&buf[1], args, argc);
    send_pkt(0x01, buf, argc + 1);
    return recv_pkt(resp, max);
}

/* High-level operations */
static int get_img(void) {
    uint8_t r[16];
    return (cmd(0x01, NULL, 0, r, sizeof r) > 0) ? r[0] : -1;
}
static int gen_char(int buf_num) {
    uint8_t a = buf_num, r[16];
    return (cmd(0x02, &a, 1, r, sizeof r) > 0) ? r[0] : -1;
}
static int reg_model(void) {
    uint8_t r[16];
    return (cmd(0x05, NULL, 0, r, sizeof r) > 0) ? r[0] : -1;
}
static int store(int slot) {
    uint8_t a[3] = { 0x01, slot >> 8, slot & 0xFF }, r[16];
    return (cmd(0x06, a, 3, r, sizeof r) > 0) ? r[0] : -1;
}
static int search(int *out_slot, int *out_score) {
    uint8_t a[5] = { 0x01, 0x00, 0x00, 0x00, 0xC8 }, r[16];   /* buf=1, start=0,
count=200 */
    int n = cmd(0x04, a, 5, r, sizeof r);
    if (n < 5) return -1;
    if (r[0] != 0x00) return r[0];
    *out_slot = (r[1] << 8) | r[2];
    *out_score = (r[3] << 8) | r[4];
    return 0;
}

static int enroll(int slot) {
    printf("Place finger...\n");
    while (get_img() != 0) usleep(100000);
    if (gen_char(1) != 0) return -1;
    printf("Lift finger, place again...\n");
    sleep(1);
    while (get_img() != 0) usleep(100000);
    if (gen_char(2) != 0) return -1;
    if (reg_model() != 0) return -1;
    return store(slot);
}

int main(int argc, char **argv) {
    uart_fd = open("/dev/ttymxc4", O_RDWR | O_NOCTTY);
```

```
    struct termios tio;
    tcgetattr(uart_fd, &tio);
    cfsetspeed(&tio, B57600);
    cfmakeraw(&tio);
    tcsetattr(uart_fd, TCSANOW, &tio);

    if (argc > 1 && strcmp(argv[1], "enroll") == 0) {
        int slot = atoi(argv[2]);
        if (enroll(slot) == 0) printf("Enrolled at slot %d\n", slot);
        else printf("Enroll failed\n");
    } else {
        printf("Place finger to identify...\n");
        for (;;) {
            if (get_img() != 0) { usleep(100000); continue; }
            if (gen_char(1) != 0) { printf("Bad image, retry\n"); sleep(1); continue; }
            int slot, score;
            int r = search(&slot, &score);
            if (r == 0) printf("Match: slot=%d score=%d\n", slot, score);
            else printf("No match\n");
            sleep(2);
        }
    }
}
```

What this exposes:

- The framing protocol’s checksum, length, header, easy to get wrong with no error messages.
- The two-stage capture for enrollment.
- The 0x0X confirmation codes that distinguish “no finger” from “too few features” from “match found.”
- Why the matching is *fast* (the chip’s internal DSP does the convolution) but *async* (a 1.2 s wall-clock for capture + extract + search).

106.7 PAM integration, 2-factor login

For “password + fingerprint” 2FA, write a small PAM module that calls into your R503 driver:

```
/* /lib/security/pam_r503.so */
PAM_EXTERN int pam_sm_authenticate(pam_handle_t *pamh, int flags, ...) {
    /* Open R503, capture, search */
    int slot, score, rc = r503_authenticate(&slot, &score);
    if (rc != 0 || score < 50) return PAM_AUTH_ERR;
    /* Check slot is allowed for this user */
    const char *user;
    pam_get_user(pamh, &user, NULL);
    if (!is_user_authorized(user, slot)) return PAM_AUTH_ERR;
    return PAM_SUCCESS;
}
```

`/etc/pam.d/sshd`:

```
auth required pam_unix.so       try_first_pass
auth required pam_r503.so
account required pam_unix.so
```

Now SSH (or any PAM-using service) requires password + fingerprint.

106.8 libfprint, for USB fingerprint scanners

If you have a laptop-style USB fingerprint reader (Validity, Synaptics, Goodix), the kernel side is `libusb` user-space. `libfprint` provides the protocol drivers. Used by `fprintd` daemon and GNOME/KDE settings.

```
apt install fprintd libfprint-2-2
fprintd-enroll username
fprintd-verify username
```

This is a different ecosystem from the embedded UART modules. Choose based on form factor and integration needs.

106.9 Key Notes

- The module owns image capture and matching, while the host owns enrollment authority, user identity and the action permitted after a result.
- Template slots are persistent state. Slot overwrite, deletion, backup and module replacement need a mapping that remains consistent with the host’s user records.
- A match score is an algorithm-specific result, not a universal confidence or security guarantee. Acceptance policy must consider both false acceptance and false rejection.
- Biometric records are sensitive and difficult to replace. Transport trust, storage protection, deletion and fallback access belong in the design, not only in the UART parser.

106.10 Pitfalls

- **Wrong baud rate.** Default 57600 on R503. Some clones default to 9600. Try both. Some modules persist whatever was last set.
- **First-scan warmup.** Optical sensors (FPM10A) need ~500 ms warmup after power-on. The first 2–3 scans may give “no finger detected” even when present.

- **Wet fingers on optical.** Optical sensors fail with wet, dirty, or smudged fingers. Capacitive (R503) is much more tolerant. If your product deploys outdoors, choose capacitive.
- **Sensor surface scratched.** A single deep scratch on the optical sensor's prism creates a permanent "fake fingerprint" everyone matches. Replace the module.
- **Power supply noise.** Some modules are picky about VCC ripple > 50 mV. Add an LC filter on the supply.
- **Template format incompatibility.** R503 templates are not interchangeable with FPM10A. Each vendor's algorithm produces a different template format. Don't try to migrate.
- **Slot overwrite on re-enroll.** `Store(slot=N)` silently overwrites the previous template at slot N. Use `ReadTemplateIndex` first to check.
- **Score threshold tuning.** Default 50. Too low = false accepts, too high = false rejects. Tune for your application's FAR/FRR balance.
- **Privacy / GDPR.** Fingerprint templates are biometric PII in many jurisdictions. Store encrypted. Allow deletion. Document the retention policy.
- **Aging.** Skin changes with age. Templates may need re-enrollment every ~5 years for older users.

106.11 Going deeper

- **Grow R503 User Manual**: protocol spec, all commands, error codes.
- **AS608 User Manual**: slightly different command set but similar architecture.
- **`libfprint` + `fprintd`** for USB scanners.
- **PAM Programming Guide** for writing custom auth modules.
- **ISO/IEC 19794-2**: fingerprint template interchange format (if you need cross-vendor portability).
- **NIST Fingerprint Image Quality (NFIQ)**: the standard quality score. Some R503 firmware versions expose it.
- **Ch 105 (RFID)**: RFID + fingerprint is the classic 2-factor combination for high-security access control.

> Next chapter: **Chapter 107: GPS/GNSS + PPS time sync**, u-blox + gpsd + chrony for sub-µs NTP.

Chapter 107: GPS / GNSS + PPS

What: GNSS receivers (GPS + GLONASS + BeiDou + Galileo) and the **PPS (Pulse-Per-Second)** time-discipline signal. We compare **u-blox NEO-6M** (legacy, GPS-only), **NEO-8M** (multi-constellation), **NEO-9M** (concurrent multi-band, lower power, GNSS RAW data), and the cheap **ATGM336H** (BeiDou+GPS+GLONASS). On Linux, we parse **NMEA-0183**, decode u-blox's binary **UBX** protocol, bring up **gpsd** as the central daemon, and wire the **PPS GPIO** to **chrony** for sub-microsecond NTP, turning a $5 receiver into a **stratum-1 time server**. **MCU bridge:** Think of Linux GPIO like the same pin set/reset block you used on STM32, but accessed through a kernel subsystem that owns numbering, direction, interrupts, and user-space exposure. **GPIO:** General-Purpose Input/Output, a pin controlled as a digital input, output, or interrupt source.

Why: GPS receivers do two things, both critical for embedded products:

1. **Position** for asset tracking, geo-fencing, fleet management, anti-theft.
2. **Time**, GPS atomic-clock-derived time, sub-µs precise, traceable to UTC. Telco basestations, financial exchanges, distributed databases (Spanner, CockroachDB), and any time-sensitive logging system uses GPS-disciplined clocks. A $5 chip + $20 antenna = stratum-1 NTP, no internet required.

The same PPS technique works for any time-domain measurement on Linux: synchronised audio between boards, distributed instruments, IP-connected oscilloscopes.

Focus: NMEA reports the wall-clock second, but it arrives 50–500 ms after the actual second. PPS is the nanosecond-accurate edge. A naïve "set the clock from `$GPRMC`" gets you to ±100 ms. With PPS, the kernel timestamps each GPIO edge using the hardware clock. Chrony combines two streams: NMEA, which is slow but tells you *which* second this is. PPS, which is fast but does not name the second. Together they reach ±100 ns. The PPS path is the key: GPS pin → kernel `pps_gpio` driver → `/dev/pps0` → chrony refclock. Get this right and you have sub-microsecond GPS time. Skip the PPS and you have NMEA-only ±100 ms.

Tooling. This chapter uses `gpsd` + `gpsd-clients` (`gpspipe`, `cgps`, `gpsmon`), `chrony`, `pps-tools` (`ppstest`).

- **Ubuntu-base (target):** `apt install gpsd gpsd-clients chrony pps-tools`
- **Buildroot:** `BR2_PACKAGE_GPSD=y BR2_PACKAGE_CHRONY=y BR2_PACKAGE_PPS_TOOLS=y`
  Buildroot: a configuration-driven build system that produces a complete root filesystem and related images.

> - Full per-tool reference: Userspace tooling appendix. **MCU bridge:** Think of the rootfs as the firmware image's file-backed runtime environment. On an MCU you link everything into flash. On Linux, programs and config live in this mounted tree. **rootfs:** root filesystem, the directory tree mounted at / that contains /bin, /etc, /dev, and libraries.

107.1 GNSS module comparison

	u-blox NEO-6M	NEO-8M	NEO-9M	ATGM336H
Year	2011	2014	2019	2017
Constellations	GPS	GPS + GLONASS + Galileo + BeiDou + QZSS	concurrent dual-band L1/L5	BeiDou + GPS + GLONASS
Channels	50	72	184	33
Position accuracy	~2.5 m CEP	~2.0 m	~1.5 m (multi-band 0.5 m)	~2.5 m
Cold-start time	~27 s	~26 s	~24 s	~32 s
Hot-start time	< 1 s	< 1 s	< 2 s	~1 s
Sensitivity (acq)	–148 dBm	–148 dBm	–148 dBm	–148 dBm
Current (acq)	67 mA	67 mA	~35 mA	30 mA
Current (tracking)	50 mA	50 mA	25 mA	25 mA
PPS output	yes	yes	yes	yes
PPS jitter	~30 ns	~30 ns	~20 ns	~50 ns
UART default	9600 8N1	9600 8N1	38400 8N1	9600 8N1

	u-blox NEO-6M	NEO-8M	NEO-9M	ATGM336H
Native protocol	NMEA + UBX	NMEA + UBX	NMEA + UBX + RAW measurements	NMEA only
Cost (module + antenna)	$8–15	$10–18	$40–60	$5–8

Pick guide:

- **NEO-8M**: the common choice for most projects. Multi-constellation = better urban coverage, faster TTFF (time to first fix). Well-documented + UBX binary.
- **NEO-9M**: when you need < 1 m accuracy, RAW pseudorange data (RTK-able with a base station), or lowest power.
- **ATGM336H**: when BOM matters more than UBX support. Cheap, NMEA-only.

107.2 NMEA-0183, the universal text protocol

NMEA is a 1980s-era ASCII protocol designed for marine instruments. Every GNSS receiver speaks it. Sentences look like:

```
$GPRMC,123519.00,A,4807.038,N,01131.000,E,022.4,084.4,230394,003.1,W*6A
       ↑           ↑↑          ↑          ↑     ↑     ↑          ↑
       UTC time   lat (DDMM.mmm)lon         spd  hdg   date       magvar
                  fix valid                knots deg  ddmmyy
```

Sentence	Content
`$GPRMC`	recommended minimum: time, lat/lon, speed, heading, date
`$GPGGA`	fix data with altitude, # satellites, HDOP
`$GPGSA`	active satellites + DOPs (PDOP, HDOP, VDOP)
`$GPGSV`	satellites in view (multiple sentences if > 4)

Sentence	Content
`$GPVTG`	velocity over ground
`$GPGLL`	lat/lon only
`$GNRMC` etc.	`GN` prefix = multi-constellation fix

The checksum is XOR of all bytes between `$` and `*`, in hex. Parsers should always verify it. Corrupt UART bytes (no flow control on most modules) flip bits silently.

NMEA's three weaknesses for time sync:

1. **Latent**: sentence is generated some time *after* the second. Transmission at 9600 baud takes ~70 ms.
2. **Inconsistent**: different modules emit GPRMC at different points in the second.
3. **Verbose**: parsing ASCII costs CPU you don't need.

Hence PPS for sub-second timing.

107.3 UBX, u-blox binary protocol

> **Privilege boundary:** $ means normal user. # or sudo means root and can change host or target state. After a privileged command, verify the expected device, service, or file appears before continuing. Roll back by undoing the config change or stopping the service you just enabled.

u-blox's native protocol is binary and **labels each message with the exact GPS time the position is valid for**. Frame:

```
[µB] [class] [id] [len_lo] [len_hi] [payload...] [ck_a] [ck_b]
 0xB5 0x62
```

Classes:

- `0x01` NAV (navigation results, POSLLH, VELNED, PVT, STATUS)
- `0x02` RXM (raw measurements, RAWX for RTK)
- `0x05` ACK (ack/nack from CFG operations)
- `0x06` CFG (configuration, port baud, message rates, GNSS selection)

- `0x0A` MON (monitor, HW status, jamming, RF antenna)

The single most useful message: `UBX-NAV-PVT` (Position, Velocity, Time, class 0x01, id 0x07, 92 bytes). One frame per fix, contains the iTOW (integer time of week, ms), year/month/day/hour/min/sec, lat/lon/h, velocity, accuracy estimates, fix type, # satellites. Replaces 5+ NMEA sentences.

The setup command to enable NAV-PVT at 1 Hz only:

```
B5 62 06 01 08 00 01 07 00 01 00 00 00 00 18 E1
       ↑ CFG-MSG       ↑ NAV-PVT, rate=1 on UART
```

Switching to UBX-only at startup reduces UART traffic 5× and gives you nanosecond-precise per-message timing. Use `u-center` on Windows or `ubxtool` from `gpsd-clients` to build these CFG messages.

107.4 PPS, the sub-microsecond signal

The PPS pin pulses high for ~100 ms exactly on the UTC second boundary. The receiver synchronizes its 1 kHz timepulse generator to its GNSS-derived clock. Jitter is ~20–50 ns.



Linux kernel side: `drivers/pps/`. The `pps_gpio` driver registers an IRQ on the GPIO, and on each edge records a hardware timestamp (`ktime_get_ts()`) plus the GPIO event time. User-space (chrony) reads `/dev/pps0` ioctl-style to get the latest edge timestamp and computes the offset between GPIO-edge-time and system-clock-time.

> **MCU bridge:** Think of an IRQ like an EXTI/NVIC interrupt path, except Linux splits the hard interrupt from deferred work and must share lines across drivers. **IRQ:** interrupt request, the signal path that tells the CPU or interrupt controller that hardware needs service.

DT binding:

```
pps {
    compatible = "pps-gpio";
    gpios = <&gpio5 3 GPIO_ACTIVE_HIGH>;
    assert-falling-edge;       /* or omit for rising-edge */
    pinctrl-names = "default";
    pinctrl-0 = <&pinctrl_pps>;
};
```

After `dmesg | grep pps`:

```
pps_core: LinuxPPS API ver. 1 registered
pps_core: Software ver. 5.3.6 - Copyright 2005-2007 Rodolfo Giometti
pps pps0: new PPS source pps.-1
pps pps0: Registered IRQ 67 as PPS source
```

`/dev/pps0` exists. The tool to dump edges:

```
ppstest /dev/pps0
# trying PPS source "/dev/pps0"
# source 0 - assert 1709236745.000001234, sequence: 1234
# source 0 - assert 1709236746.000001456, sequence: 1235
```

Each pulse's timestamp is captured with `getnstimeofday()` precision, typically sub-microsecond on a Cortex-A7 with hardware IRQ.

107.5 gpsd, the central daemon

gpsd opens the GPS receiver UART, parses NMEA/UBX, and exposes the parsed fix via a JSON socket protocol on `localhost:2947`. Applications (your code, chrony, navit, foxtrotgps, qmapshack) talk to gpsd, not the receiver directly.

```
apt install gpsd gpsd-clients

# /etc/default/gpsd
DEVICES="/dev/ttymxc3 /dev/pps0"
GPSD_OPTIONS="-n"               # don't wait for client; always read

systemctl restart gpsd

# Inspect live fix
gpsmon
# Or in JSON
gpspipe -w
#  {"class":"TPV","time":"2026-05-31T12:34:56.000Z","lat":52.3,"lon":13.4,...}

# Status check
cgps -s
```

For Python:

```
import gps
session = gps.gps(mode=gps.WATCH_ENABLE)
for report in session:
    if report['class'] == 'TPV':
        print(report.lat, report.lon, report.time)
```

107.6 Chrony + PPS, the stratum-1 NTP server

Once gpsd is up and `/dev/pps0` is alive, chrony combines NMEA (gives the second number) + PPS (gives the precise edge) to discipline the system clock.

```
apt install chrony

# /etc/chrony/chrony.conf — append:
refclock SHM 0 refid GPS poll 4 noselect    # gpsd's SHM segment (low precision, just for
the second)
refclock PPS /dev/pps0 refid PPS lock GPS prefer trust   # PPS (high precision)

systemctl restart chrony

# Verify
chronyc sources -v
# MS Name/IP address  Stratum Poll Reach LastRx Last sample
# #? GPS                     0    4   377    13   -45ms[  -45ms] +/-  100ms
# #* PPS                     0    4   377    14   -100ns[ -100ns] +/-   200ns
```

The `*` next to PPS means it's the chosen reference. The offset is < 200 ns. The system clock is now disciplined to GPS time at sub-microsecond accuracy. `date +%N.%9N` shows nanosecond-precise time.

To serve NTP to your LAN:

```
# chrony.conf
allow 192.168.0.0/24

# Now other Linux/Windows boxes can ntpd-sync from this i.MX6ULL
# On a client:
chronyc sources
# ^* mygpsbox.local              1   6   377    25    +12µs[  +15µs] +/-  410µs
```

Total parts cost is about $20. The result is a stratum-1 NTP server.

107.7 From scratch, UBX parser in C

Use NMEA for debugging, UBX for production. The skeleton parser:

```
/* ubx_parse.c — fragment */
struct ubx_nav_pvt {
    uint32_t iTOW;
    uint16_t year;
    uint8_t  month, day, hour, min, sec;
    uint8_t  valid;
    uint32_t tAcc;
    int32_t  nano;
    uint8_t  fixType;
    /* ... lat, lon, height, vel, etc ... */
};
```

```
static int read_ubx_frame(int fd, uint8_t *cls, uint8_t *id,
                          uint8_t *payload, int *plen) {
    uint8_t buf;
    /* Wait for 0xB5 0x62 */
    do { if (read(fd, &buf, 1) != 1) return -1; } while (buf != 0xB5);
    read(fd, &buf, 1);
    if (buf != 0x62) return -2;
    read(fd, cls, 1);
    read(fd, id, 1);
    uint8_t len_lo, len_hi;
    read(fd, &len_lo, 1); read(fd, &len_hi, 1);
    int len = len_lo | (len_hi << 8);
    if (len > 200) return -3;
    for (int i = 0; i < len; ) {
        int n = read(fd, &payload[i], len - i);
        if (n <= 0) return -4;
        i += n;
    }
    /* Read + verify Fletcher checksum */
    uint8_t ck_a = 0, ck_b = 0;
    ck_a += *cls; ck_b += ck_a;
    ck_a += *id;  ck_b += ck_a;
    ck_a += len_lo; ck_b += ck_a;
    ck_a += len_hi; ck_b += ck_a;
    for (int i = 0; i < len; i++) { ck_a += payload[i]; ck_b += ck_a; }
    uint8_t rx_a, rx_b;
    read(fd, &rx_a, 1); read(fd, &rx_b, 1);
    if (ck_a != rx_a || ck_b != rx_b) return -5;
    *plen = len;
    return 0;
}

int main(void) {
    int fd = open("/dev/ttymxc3", O_RDWR | O_NOCTTY);
    /* configure 38400 8N1 ... */
    for (;;) {
        uint8_t cls, id, payload[200]; int len;
        if (read_ubx_frame(fd, &cls, &id, payload, &len) == 0
            && cls == 0x01 && id == 0x07) {
            struct ubx_nav_pvt *p = (void *)payload;
            printf("%04d-%02d-%02dT%02d:%02d:%02d.%09d  fix=%d\n",
                   p->year, p->month, p->day, p->hour, p->min, p->sec,
                   p->nano, p->fixType);
        }
    }
}
```

Run this. You'll see each fix printed with the exact GPS-derived UTC time it was valid for. Compare with the PPS-disciplined system clock to verify they agree.

107.8 Key Notes

- GNSS messages identify time and position, while PPS provides a timing edge. Combining them requires associating that edge with the correct second and validity state.

- Receiver time, the system clock and network time distribution are different stages. A good receiver specification does not establish the target's GPIO timestamp or NTP accuracy.
- Transport framing and checksums protect message interpretation. Fix status, antenna conditions and leap-second information affect whether the resulting data is usable.
- Loss of reception needs a holdover and recovery policy. The system should distinguish synchronized time from an estimate based on an aging local clock.

107.9 Pitfalls

- **Antenna missing or shielded.** GPS needs sky view. An indoor desk position usually fails. Use a roof or window-mount antenna with a coax extension.
- **No PPS GPIO.** Some "GPS modules" omit the PPS pin or it's not bonded out. Verify before buying.
- **PPS GPIO not configured.** Without the DT `pps-gpio` node, `/dev/pps0` doesn't appear. `modprobe pps_gpio` only works if the platform driver instantiated it from DT.
- **PPS polarity wrong.** Rising-edge vs falling-edge, check the module datasheet. Wrong polarity → chrony sees no edges.
- **NMEA-only refclock = poor accuracy.** Without PPS, system clock accuracy is ~30 ms (NMEA latency). Insist on PPS for sub-µs.
- **Baud rate too low for UBX-NAV-PVT at 10 Hz.** At 9600, the 92-byte NAV-PVT plus other UBX leaves no headroom for 10 Hz updates. Switch to 38400 or 115200.
- **NMEA checksum bytes flipped.** No flow control + heavy bus traffic = bit flips. Always verify the checksum and discard bad sentences.
- **Multi-constellation overrides single-constellation in NMEA.** GNRMC, GNGGA replace GPRMC, GPGGA. Parsers must accept both prefixes.
- **u-blox jamming detection.** The MON-RF message reports interference. If jamming is detected (drone show, military jammer, RF leak), the module may report no fix. Don't blame the antenna without checking MON-RF.
- **GNSS time vs UTC leap seconds.** GPS time has no leap seconds. UTC does. Old or unprogrammed modules may emit times off by 18 s after a leap second. Use the `LeapSeconds` field if exposed.
- **TPS regulator + GPS together = noise.** Switching regulators inject noise on the GPS antenna's RF input. Use an LDO close to the antenna, or shield the regulator.

107.10 Going deeper

- **u-blox NEO-8M / NEO-9M Protocol Specification**: the canonical UBX reference (1500+ pages).
- **`gpsd` documentation**: covers many receivers, JSON protocol, refclock SHM mechanism.
- **`chrony` documentation**: refclock PPS, GPS, SHM integration.
- **`drivers/pps/`** + `Documentation/pps/pps.rst`, kernel side.
- **NMEA-0183** standard (proprietary. Many free summaries online).
- **NTPv4 + IEEE 1588 PTP**: for sub-µs over Ethernet (after you have a local stratum-1).
- **RTKLIB**: for RTK centimetre-accurate positioning using u-blox RAW data.
- **Ch 51B**: for using PPS to wake a sleeping device every second.

> Next chapter: **Chapter 108: RS-485 + Modbus RTU**, beginning Group Q (Industrial buses).

Chapter 108: RS-485 + Modbus RTU

What: RS-485, the differential half-duplex serial bus that has carried industrial data since 1983, still ubiquitous in factories, building automation, solar inverters, and HVAC. We compare **MAX485** (5 V, 5 Mbps, the canonical chip), **SP3485** (3.3 V, 10 Mbps), **ADM2483** (isolated, for noisy environments), **MAX13487** (auto-direction, eliminates the GPIO control headache). On Linux, we wire RS-485 to a UART, enable the kernel's RS-485 mode (`SER_RS485_ENABLED`), and use **`libmodbus`** to implement a **Modbus RTU** master/slave talking to real inverters, energy meters, and PLCs. **MCU bridge:** Think of Linux GPIO like the same pin set/reset block you used on STM32, but accessed through a kernel subsystem that owns numbering, direction, interrupts, and user-space exposure. **GPIO:** General-Purpose Input/Output, a pin controlled as a digital input, output, or interrupt source.

Why: every industrial site has Modbus RTU. Solar inverters, energy meters, BMS systems, irrigation controllers, VFDs (variable-frequency drives), even building HVAC, they all expose data over Modbus RTU on RS-485. Your i.MX6ULL becomes the data collector / gateway, polling 5–50 devices and bridging to MQTT/cloud. RS-485 is everywhere in industrial systems. Knowing it gives you access to the industrial IoT market that pure-WiFi devices cannot reach.

Focus: RS-485 is half-duplex differential signalling on a two-wire bus. You must switch the driver between TX and RX with sub-bit-time precision. On top of that, Modbus RTU detects frames by inter-character timeouts that scale with baud rate. The kernel's RS-485 ioctl handles DE/RE control automatically *if* your UART supports it. Otherwise you bit-bang via a GPIO with sub-microsecond latency requirements that are hard on a non-RT Linux. Auto-direction transceivers (MAX13487) solve this in hardware. Three things break a bench setup when you deploy it on a real factory floor. Termination, 120 Ω at each end of the bus. Biasing, fail-safe pull-ups so the idle line is a known logic level. Ground reference, measure the voltage between grounds before connecting devices powered from different supplies.

Tooling. This chapter uses `libmodbus5` + dev headers (C), or `pymodbus` (Python). Optional `modpoll` CLI tester.

- **Ubuntu-base (target):** `apt install libmodbus5 libmodbus-dev python3-pymodbus`
- **Buildroot:** `BR2_PACKAGE_LIBMODBUS=y BR2_PACKAGE_PYTHON3_PYMODBUS=y` **Buildroot:** a configuration-driven build system that produces a complete root filesystem and related images.

> - Full per-tool reference: Userspace tooling appendix. **MCU bridge:** Think of the rootfs as the firmware image's file-backed runtime environment. On an MCU you link everything into flash. On Linux, programs and config live in this mounted tree. **rootfs:** root filesystem, the directory tree mounted at / that contains /bin, /etc, /dev, and libraries.

108.1 RS-485 physics

```
Differential pair (twisted), 120 Ω characteristic impedance
A line: ──────────────┐ ┌────────┐ ┌────────────────
                      │ │        │ │
                       X          X        <- signal swing 1.5–5 V
                      │ │        │ │
B line: ──────────────┘ └────────┘ └────────────────

Receiver sees (A − B). Above +200 mV = "1"; below −200 mV = "0".
~24 V common-mode range. Tolerant of ~7 V ground difference.
```

Why it works for long distance:

- Differential rejects common-mode noise (motor brushes, switching supplies).
- 120 Ω matched impedance + termination resistors → no reflections at high baud / long cable.
- Multidrop: up to 32 standard receivers (more with low-load chips). One transmitter at a time.

Cable lengths:

- 100 m at 1 Mbps
- 1200 m at 100 kbps
- Up to 4 km at 9600 (the Modbus default)

Topology: daisy-chain only. No star or branch wiring. Spurs from the main bus must be shorter than 30 cm. Terminate both physical ends.

```
[Master] ──── stub ── ┬ ────────── ┬ ── stub ── [Slave 3]
                      [Slave 1]   [Slave 2]
120 Ω here ┘                                          └ 120 Ω here
           and pull-up to +5V (bias)
           and pull-down to GND (bias)
           for fail-safe (idle line = logic "1")
```

108.2 Transceiver comparison

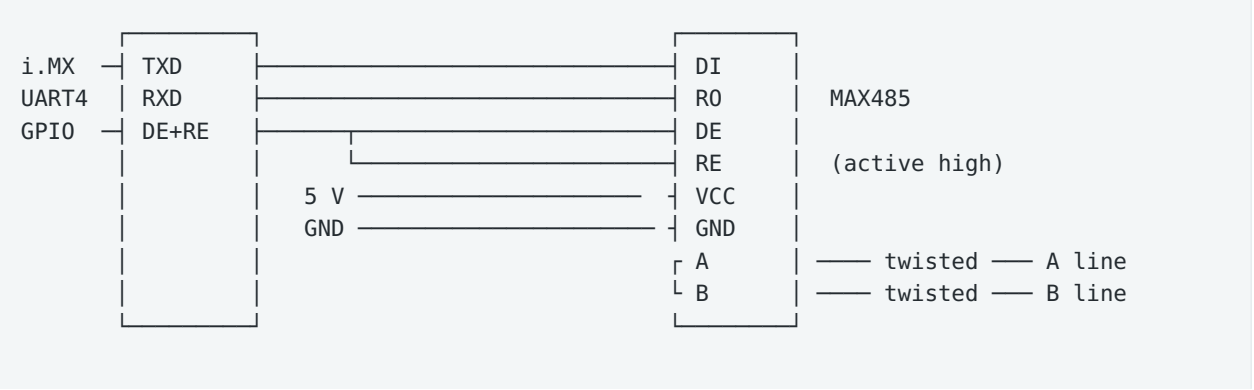
	MAX485	SP3485	ADM2483	MAX13487
Supply	5 V	3.3 V	5 V (isolated)	3.3 V or 5 V
Max speed	2.5 Mbps	10 Mbps	0.5 Mbps	16 Mbps
DE/RE control	GPIO	GPIO	GPIO	**automatic** (senses TX activity)
Isolation	none	none	2.5 kV galvanic	none
Receivers per bus	32	32	256	256
Cost	$0.50	$0.80	$5	$1.50

Pick guide:

- **MAX485 / SP3485**: bench testing, simple single-board projects. Manual DE/RE control via GPIO.
- **MAX13487**: recommended for production. Auto-direction removes the timing-critical DE/RE control.
- **ADM2483**: when your bus has 24 V power devices, motors nearby, long outdoor runs. Isolation prevents ground loops from blowing up the SoC.

108.3 Wiring RS-485 to the i.MX6ULL

For a manual-direction transceiver (MAX485):



```
                                120 Ω termination at both ends of A-B
                                +5V via 680Ω to A; GND via 680Ω to B (bias)
```

DE and RE are tied (active-high for DE, active-low for RE = same line drives both, so you assert it high to transmit).

For MAX13487 auto-direction, no DE/RE, just connect. The chip senses TX activity and switches direction.

Termination: 120 Ω across A-B at both ends of the bus. Bias resistors (680 Ω to +5 V on A, 680 Ω to GND on B) ensure the bus reads logic-1 when no one is driving, required by Modbus RTU's frame-detection.

108.4 Kernel RS-485 mode

The i.MX UART driver can manage DE/RE automatically if your DT specifies it. This is much better than bit-banging from user-space (which is jitter-prone on non-RT Linux).

```
&uart4 {
    pinctrl-names = "default";
    pinctrl-0 = <&pinctrl_uart4>;
    rts-gpios = <&gpio1 28 GPIO_ACTIVE_HIGH>;
    linux,rs485-enabled-at-boot-time;
    rs485-rts-active-high;
    rs485-rts-delay = <0 0>;       /* µs before/after frame */
    status = "okay";
};
```

The driver toggles RTS automatically around each TX. From user-space, also:

```
struct serial_rs485 conf = {0};
conf.flags = SER_RS485_ENABLED | SER_RS485_RTS_ON_SEND;
conf.delay_rts_before_send = 0;
conf.delay_rts_after_send = 0;
ioctl(fd, TIOCSRS485, &conf);
```

Verify:

```
ioctl-test /dev/ttymxc3 TIOCGRS485
# flags: enabled, RTS_ON_SEND
```

Some UARTs (i.MX has integrated "9-bit mode" with hardware DE) do this directly without GPIO toggling, check your reference manual for SER_RS485_AUTO.

If the kernel can't manage it, user-space `libmodbus` does GPIO toggling, works at 9600 (1 ms per byte = plenty of margin), struggles above 115200.

108.5 Modbus RTU framing

The protocol on the wire:

```
[SlaveAddr] [FunctionCode] [Data...] [CRC16-Lo] [CRC16-Hi]
    1B            1B          N B         2B
Frame end: 3.5-character silent gap (no bytes for 3.5 × char-time)
```

Slave addresses 1–247. 0 = broadcast. Function codes:

Code	Name	What
0x01	Read Coils	read bit-addressed outputs
0x02	Read Discrete Inputs	read bit-addressed inputs
0x03	Read Holding Registers	read 16-bit registers
0x04	Read Input Registers	read read-only 16-bit registers
0x05	Write Single Coil	write one bit
0x06	Write Single Register	write one 16-bit register
0x0F	Write Multiple Coils	write many bits
0x10	Write Multiple Registers	write many 16-bit registers
0x17	Read/Write Multiple Registers	combined

CRC: standard Modbus CRC-16, polynomial 0xA001, initial 0xFFFF.

Frame separation by **inter-character timeout**:

- Inter-character (within frame): < 1.5 char-times
- Inter-frame: ≥ 3.5 char-times

At 9600 baud, one 11-bit character takes 1.146 ms. The 3.5-character inter-frame gap is therefore about 4 ms, that is the idle time after which a receiver declares end-of-frame.

This is where Linux can struggle: the kernel buffers UART data. User-space sees it in chunks. Reconstructing the precise inter-character gaps requires the kernel to timestamp byte arrivals. `libmodbus` handles this with `select()` + read-timeout heuristics. It works fine at ≤ 38400.

108.6 Using libmodbus from C

```
#include <modbus/modbus.h>

modbus_t *ctx = modbus_new_rtu("/dev/ttymxc3", 9600, 'N', 8, 1);
modbus_rtu_set_serial_mode(ctx, MODBUS_RTU_RS485);   /* uses kernel RS485 mode */
modbus_set_slave(ctx, 1);
modbus_connect(ctx);

uint16_t regs[32];
int rc = modbus_read_registers(ctx, 0x0000, 16, regs);
if (rc == -1) {
    fprintf(stderr, "Read failed: %s\n", modbus_strerror(errno));
} else {
    for (int i = 0; i < 16; i++)
        printf("reg[0x%04X] = %u (0x%04X)\n", i, regs[i], regs[i]);
}

modbus_close(ctx);
modbus_free(ctx);
```

That’s an entire Modbus master. Replace `modbus_read_registers` with `modbus_write_register`, `modbus_read_bits` etc. For other ops.

For a Python prototype: `pymodbus` mirrors the API and is great for interactive testing.

108.7 A worked example, reading a solar inverter

Solar inverters (Growatt, Goodwe, SMA, Sungrow) speak Modbus RTU at 9600 8N1 on a built-in RS-485 port. Map (typical, vendor-specific):

Register	Meaning
0x0001	inverter status (0=waiting, 1=running, ...)
0x0003	PV1 voltage (×0.1 V)
0x0004	PV1 current (×0.1 A)
0x0007	total grid frequency (×0.01 Hz)

Register	Meaning
0x000C	total active power (×0.1 W)
0x001A	today’s energy (×0.1 kWh)

Polling loop:

```
import time, json, paho.mqtt.publish as publish
from pymodbus.client.sync import ModbusSerialClient

client = ModbusSerialClient(method='rtu', port='/dev/ttymxc3',
                            baudrate=9600, stopbits=1, parity='N', timeout=1)
client.connect()
while True:
    rr = client.read_holding_registers(0x0001, 30, unit=1)
    if rr.isError(): time.sleep(5); continue
    data = {
        'status': rr.registers[0],
        'pv1_v':  rr.registers[2] / 10.0,
        'pv1_i':  rr.registers[3] / 10.0,
        'freq':   rr.registers[6] / 100.0,
        'power':  rr.registers[11] / 10.0,
        'energy_today': rr.registers[25] / 10.0,
    }
    publish.single('home/solar/inverter', json.dumps(data), hostname='localhost')
    time.sleep(5)
```

About 30 lines of Python and $15 of hardware put the inverter on Home Assistant or Grafana.

108.8 Modbus slave, make your i.MX6ULL a peripheral

```
modbus_t *ctx = modbus_new_rtu("/dev/ttymxc3", 9600, 'N', 8, 1);
modbus_set_slave(ctx, 5);                          /* our address */
modbus_rtu_set_serial_mode(ctx, MODBUS_RTU_RS485);
modbus_connect(ctx);

modbus_mapping_t *mb = modbus_mapping_new(100, 100, 100, 100);  /* coils, discrete,
holding, input */

for (;;) {
    uint8_t query[MODBUS_RTU_MAX_ADU_LENGTH];
    int n = modbus_receive(ctx, query);
    if (n > 0) {
        /* Update mb->tab_input_registers etc. From your sensors */
        mb->tab_input_registers[0] = read_temperature();
        modbus_reply(ctx, query, n, mb);
```

```
    }
}
```

Now your i.MX6ULL is slave 5 on the bus. Any master can poll it for sensor data.

108.9 Key Notes

- RS-485 defines an electrical link, while Modbus RTU defines framing and operations above it. Good electrical signaling does not establish correct register semantics.
- Direction control belongs with reliable UART timing or suitable hardware. User-space wakeup time is not a safe substitute for a precise turnaround requirement.
- Termination, biasing, common-mode range and isolation depend on topology and environment. A short bench cable does not qualify a factory installation.
- Register numbering, function codes, units and multi-register encoding form the device-level contract. A valid CRC cannot detect a correctly framed request to the wrong register.

108.10 Pitfalls

- **No bias resistors → idle floats.** Bus reads garbage. Slaves think the master is constantly framing data. Fail-safe biasing (typically built into MAX13487 and similar) is mandatory.
- **One-end termination only.** Reflections cause CRC errors on long buses. Always terminate both ends.
- **Ground potential difference.** Long bus → different equipment grounds may differ by several volts. RS-485 tolerates ±7 V common-mode. Beyond that, transceivers latch up or die. Use isolated transceivers (ADM2483) for long outdoor runs or different-power-source bus segments.
- **GPIO DE jitter.** User-space-toggled DE has tens-of-µs jitter on non-RT Linux, at 115200 baud, one bit is 9 µs. Half a byte can be lost. Use kernel RS-485 mode or auto-direction transceivers.
- **Inter-character timeout too short.** Linux may deliver bytes in chunks every 10 ms. Libmodbus's read loop must allow for that. Default timeouts work. Very tight timeouts cause spurious "incomplete frame" errors.
- **Wrong baud / parity.** Modbus default is 9600 8N1, but devices often use 19200 8E1 or 38400 8N2. Always check the device manual. Libmodbus's `parity` parameter is 'N', 'E', or 'O'.

- **Wrong register addressing convention.** Some vendors document registers as 1-based (40001). The actual wire-level is 0-based (0x0000). The mapping is “subtract 1 and use the 0x03 function code.”
- **Holding vs input registers.** Function 0x03 = holding (R/W). 0x04 = input (R-only). Vendors mix these confusingly. Trial-and-error or careful manual reading.
- **Float-encoding ambiguity.** A 32-bit float spans two 16-bit registers. The byte/word order varies per vendor (big/little/mixed endian). Test with a known reading (e.g., grid voltage near 230 V).
- **Modbus TCP confusion.** Some devices speak Modbus TCP on Ethernet. The protocol’s data section is the same as RTU but with an MBAP header instead of CRC. Libmodbus has separate `modbus_new_tcp()` for that. Don’t mix them up.

108.11 Going deeper

- **Modicon Modbus Protocol Reference (Schneider)**: the original specification.
- **Modbus.org**: current spec downloads (free registration).
- **`libmodbus`** docs, RTU + TCP, master + slave, mature.
- **`pymodbus`**: Python alternative, great for prototyping.
- **TI / Maxim RS-485 application notes**: termination, biasing, ESD protection.
- **Drivers/tty/serial/imx.c**: the i.MX UART driver’s RS-485 implementation.
- **EIA-485 standard** (TIA-485-A), the physical layer spec.
- **Ch 109 (LIN)** + **Ch 110 (CAN)**, siblings in the “industrial bus” family.

> Next chapter: **Chapter 109: LIN bus**, sub-CAN bus for automotive bodywork.

Chapter 109: LIN bus

> **What: LIN (Local Interconnect Network)**, a single-wire, master/slave serial bus used in cars for low-cost peripherals where CAN is overkill (door modules, seat motors, mirror controls, HVAC fans, rain sensors, parking sensors). Three transceivers: **NXP TJA1020** (legacy 12 V), **TJA1027** (3.3/5 V LIN 2.x), **Microchip MCP2003B** (similar). On Linux there is no native LIN subsystem, so we drive it from a UART with custom break + parity handling, build a master node and a slave responder in C, and demonstrate talking to a real automotive LIN node (HVAC blower controller from a junkyard).
>
> **Why:** if you're integrating with automotive systems, retrofit modules, OBD diagnostic tools, custom dashboards, EV conversion kits, you'll meet LIN. Cars use 100+ LIN slaves. CAN buses delegate "is the user pressing the seat-heat button" to a LIN sub-bus underneath. LIN is also spreading into industrial actuator buses (HVAC valves, building blinds). At about $0.40 per node and one wire, it is the cheapest option for simple peripherals. Linux has no native LIN subsystem. You write the framing yourself. This is also a useful UART example.
>
> **Focus:** A LIN frame is: a UART start, a 'break' pulse, an 0x55 sync byte, a 6-bit Protected Identifier (PID) byte, one to eight data bytes, and a checksum byte. A single master schedules every frame. Slaves never transmit on their own. The break signal (≥13 dominant bits = ~1.4 ms low at 9600 LIN-baud) is *not* a normal UART feature, you either generate it with `tcsendbreak()` or by toggling baud rate momentarily. The protocol is simple. The trap is getting the timing right on a non-deterministic Linux UART.

109.1 LIN at a glance vs CAN, RS-485

	LIN	CAN	RS-485
Wires	1 + GND	2 (differential) + GND	2 (differential) + GND
Speed	1–20 kbps	125 kbps, 1 Mbps (CAN-FD: 5+ Mbps)	up to 10 Mbps
Topology	bus, 1 master + 1–15 slaves	bus, multi-master	bus, multidrop

	LIN	CAN	RS-485
Arbitration	master-scheduled (no arbitration)	CSMA/CR	master-scheduled
Cost / node	$0.40 transceiver	$1–2 transceiver	$0.80 transceiver
Use cases	comfort, body (windows, mirrors, HVAC)	powertrain, safety	industrial sensors, energy meters
Standard	ISO 17987 / LIN 2.x	ISO 11898	TIA-485
OS support on Linux	none native	`SocketCAN` (great)	`serial` + `libmodbus`

LIN's design philosophy: **deterministic schedule, no contention, cheap silicon**. The master polls each slave on a fixed schedule (e.g., poll temp sensor every 1 s, poll door switch every 100 ms). No collision, no priorities, no extra hardware.

109.2 LIN frame format

```
Master generates HEADER:
   [Break field] [Sync byte=0x55] [PID byte]
     ≥13 dom bits   0x55           ID + 2 parity bits

Slave (or master) responds with RESPONSE:
   [Data 1] [Data 2] ... [Data N] [Checksum]
     1–8 bytes               classic or enhanced CRC
```

PID byte (Protected Identifier):

```
bit 7  6  5  4  3  2  1  0
    P1 P0 ID5 ID4 ID3 ID2 ID1 ID0
P0 = ID0 ⊕ ID1 ⊕ ID2 ⊕ ID4
P1 = !(ID1 ⊕ ID3 ⊕ ID4 ⊕ ID5)
```

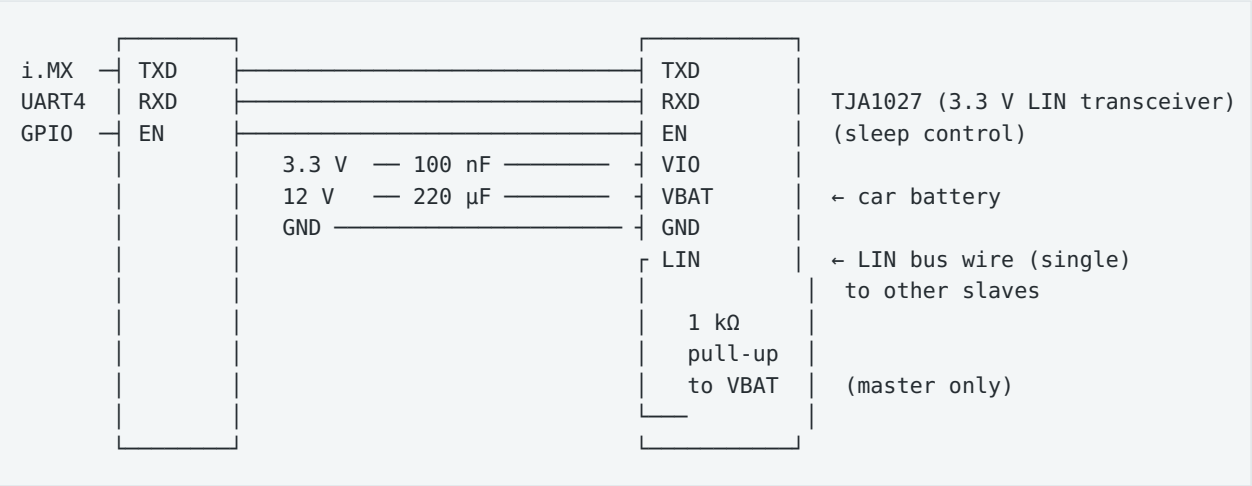
The 6-bit ID is 0..63. Some IDs are reserved (0x3C, 0x3D = master/slave request frames. 0x3E, 0x3F = reserved).

Checksum:

- **Classic** (LIN 1.x): sum of data bytes only.

- **Enhanced** (LIN 2.x): sum of PID + data bytes. The same byte position has different semantics depending on which version the bus runs.

109.3 Wiring



The LIN bus is a single wire pulled up to 12 V (typically 7–18 V in practice). Each node pulls the wire low to transmit. Idle = recessive = 12 V. Dominant = ~0 V. The master provides a 1 kΩ pull-up (slaves use 30 kΩ pull-up). The TJA1027 handles the level translation between the 12 V LIN domain and the 3.3 V UART domain.

109.4 Generating a LIN break + frame from a UART

The break field is the trick: 13+ consecutive dominant (low) bits at LIN baud. UART hardware can’t natively emit that without one of these tricks:

Trick 1, `tcsendbreak()`

```
tcsendbreak(uart_fd, 0);    /* 0 = at least 250 ms in POSIX; on Linux ~250 ms */
```

But 250 ms is way too long for a 20 kbps LIN bus. Linux Documentation/serial states the value is implementation-specific. The i.MX UART driver, by default, gives the minimum break (~13 bits at the current baud), which is exactly what we want, but you must verify with a scope.

Trick 2, baud-rate switch

Temporarily switch the UART to half-baud, send 0x00, switch back. At half baud, the 9 bit-times of 0x00’s start+8data = 18 bit-times at full baud, exceeds the 13-bit minimum.

```
struct termios t;
tcgetattr(fd, &t);
cfsetspeed(&t, B4800);                  /* 9600 / 2 */
tcsetattr(fd, TCSANOW, &t);
write(fd, "\0", 1);
```

```
tcdrain(fd);
cfsetspeed(&t, B9600);
tcsetattr(fd, TCSANOW, &t);
```

Trick 3, bit-bang with GPIO

For the most reliable timing, switch the TX pin to GPIO momentarily, pulse it low for the calculated time, switch back. Reliable but messy.

> **MCU bridge:** Think of Linux GPIO like the same pin set/reset block you used on STM32, but accessed through a kernel subsystem that owns numbering, direction, interrupts, and user-space exposure. **GPIO:** General-Purpose Input/Output, a pin controlled as a digital input, output, or interrupt source.

Trick 4, i.MX UART SEND_BREAK bit

The i.MX UART has a `SENDBRK` bit in UCR1 that asserts continuous TX while set. From user space, `ioctl(fd, TIOCSBRK)` then `usleep(1400)` then `ioctl(fd, TIOCCBRK)` gives a ~1.4 ms break at the line's baud, perfect for 9600 LIN.

109.5 From scratch, LIN master in C

lin_master.c:

```
/* Minimal LIN 2.x master. Sends headers; reads slave responses.
 * Tested against an HVAC blower controller from a 2018 VW (LIN slave at ID 0x10).
 */
#include <fcntl.h>
#include <stdint.h>
#include <stdio.h>
#include <string.h>
#include <termios.h>
#include <unistd.h>
#include <sys/ioctl.h>

static int fd;

static uint8_t pid(uint8_t id) {
    uint8_t p0 = ((id >> 0) ^ (id >> 1) ^ (id >> 2) ^ (id >> 4)) & 1;
    uint8_t p1 = (~((id >> 1) ^ (id >> 3) ^ (id >> 4) ^ (id >> 5))) & 1;
    return (p1 << 7) | (p0 << 6) | (id & 0x3F);
}

static uint8_t cksum_enhanced(uint8_t pid_byte, const uint8_t *data, int n) {
    uint16_t s = pid_byte;
    for (int i = 0; i < n; i++) s += data[i];
    while (s > 0xFF) s = (s & 0xFF) + (s >> 8);
    return (uint8_t)(~s);
}
```

```
static void send_break(void) {
    /* i.MX-specific: assert TX low for 1.4 ms (>13 LIN bits at 9600) */
    ioctl(fd, TIOCSBRK, 0);
    usleep(1500);
    ioctl(fd, TIOCCBRK, 0);
    usleep(50);                         /* inter-byte gap before sync */
}

static void send_header(uint8_t id) {
    send_break();
    uint8_t sync = 0x55;
    write(fd, &sync, 1);
    uint8_t p = pid(id);
    write(fd, &p, 1);
    tcdrain(fd);
}

/* Master sending response (master broadcasting to slaves) */
static void send_response(uint8_t id, const uint8_t *data, int n) {
    send_header(id);
    write(fd, data, n);
    uint8_t c = cksum_enhanced(pid(id), data, n);
    write(fd, &c, 1);
    tcdrain(fd);
}

/* Master polling a slave (header only; slave fills the response) */
static int poll_slave(uint8_t id, uint8_t *out, int expected_len, int timeout_ms) {
    send_header(id);
    /* Now read what the slave sent + its checksum */
    int total = expected_len + 1;
    int got = 0;
    while (got < total && timeout_ms > 0) {
        int n = read(fd, &out[got], total - got);
        if (n > 0) got += n;
        else { usleep(1000); timeout_ms--; }
    }
    if (got < total) return -1;
    uint8_t c = cksum_enhanced(pid(id), out, expected_len);
    if (c != out[expected_len]) return -2;
    return expected_len;
}

int main(void) {
    fd = open("/dev/ttymxc3", O_RDWR | O_NOCTTY);
    struct termios t = {0};
    tcgetattr(fd, &t);
    cfsetspeed(&t, B9600);
    cfmakeraw(&t);
    t.c_cflag |= CLOCAL | CREAD;
    tcsetattr(fd, TCSANOW, &t);

    /* Send command frame: ID 0x20, "fan speed = 100" */
    uint8_t cmd[2] = { 100, 0 };
    send_response(0x20, cmd, 2);

    /* Poll status frame: ID 0x10, expecting 2-byte response */
    uint8_t buf[16];
    int n = poll_slave(0x10, buf, 2, 30);
```

```
    if (n > 0)
        printf("Slave responded: %02X %02X (status=0x%02X temp=%d)\n",
               buf[0], buf[1], buf[0], buf[1]);

    return 0;
}
```

That's a working LIN 2.x master in ~80 lines. To run a periodic schedule (poll ID 0x10 every 100 ms, broadcast control on 0x20 every 500 ms), add a scheduler loop.

109.6 LIN slave on the i.MX6ULL

The i.MX6ULL is overkill as a LIN slave (the typical slave is a Cypress PSoC or a $0.30 8-bit MCU), but for prototyping or aggregating multiple sensors onto one bus it's useful.

```
/* Wait for break (UART detects framing error on receive). */
for (;;) {
    uint8_t b;
    int n = read(fd, &b, 1);
    /* Check for break via ioctl TIOCMIWAIT(TIOCM_BRK) or via TIOCGICOUNT */
    /* On break: read sync byte, PID, decide if we own this ID */
    uint8_t sync, p;
    read(fd, &sync, 1);
    if (sync != 0x55) continue;
    read(fd, &p, 1);
    uint8_t id = p & 0x3F;
    if (id == OUR_RESPONSE_ID) {
        /* We're publisher for this ID — send our 2-byte status + checksum */
        uint8_t data[2] = { sensor_status(), sensor_temp() };
        uint8_t c = cksum_enhanced(p, data, 2);
        write(fd, data, 2);
        write(fd, &c, 1);
    }
    /* Else: master broadcast we may want to subscribe to — capture data */
}
```

Detecting the break from user-space is hard on Linux. There is no dedicated API. Practical patterns:

- Look for a UART framing error followed by 0x00 byte (the break appears as 0x00 with a framing error flag).
- Poll `ioctl(fd, TIOCGICOUNT, &counts)` for `brk++` since last call.

Both methods work but neither is clean. For real products, run the LIN slave on a dedicated MCU.

109.7 LIN sleep + wake

LIN supports a deep sleep mode for low-power automotive ECUs:

- **Sleep command**: master sends ID 0x3C with data `[0x00, 0xFF×7]`. All slaves enter sleep.
- **Wake**: any node pulls the bus low for ≥250 µs. All nodes wake up.

The TJA1027 has an EN pin, when low, the transceiver itself is off (drawing ~10 µA from VBAT). Master pulls EN low. Slaves are still on the bus but the bus is idle. To wake: master pulls EN high, then drives the wake pulse.

This is how a car can leave 50+ LIN slaves on a bus that drains 100 µA quiescent.

109.8 Talking to a real automotive LIN slave

Junkyard a VW/Audi HVAC blower controller (~$15). Pinout:

- 12 V power
- GND
- LIN

Pull-up 1 kΩ from LIN to 12 V (you're the master). Wire to your TJA1027. The blower controller's slave ID for "set fan speed" is typically `0x20`. The exact ID is vendor-specific. You find it by capturing bus traffic and matching commands to behaviour.

Send: ID 0x20, data `[speed_0..255, 0x00, 0x00, ..., 0x00]` (8 bytes total for VW). The fan should spin proportional to speed.

At this point you have driven an automotive comfort actuator from Linux. The same pattern works for door-lock modules, mirror-fold motors, sunroof tilt, all LIN slaves.

109.9 Key Notes

- LIN uses one schedule owner to issue headers and coordinate responses. Adding another sender without that role agreement creates collisions.
- Break, synchronization, protected identifier and checksum have distinct jobs. A UART byte stream alone does not establish a compliant LIN transaction.
- Timing-sensitive work must have a suitable hardware or software owner. A Linux example does not prove that slave response deadlines are met under load.
- Transceiver supplies, wake behavior and application signals are device-specific. Automotive integration requires documented interfaces and a safe isolated test environment, not speculative commands on a vehicle.

109.10 Pitfalls

- **Break too short or too long.** <13 bits → slave doesn’t see it as break, treats as data → garbage frame. >50 bits → slave times out and ignores the rest.
- **No pull-up on bus.** Bus floats. Nobody reads anything coherent. The master is responsible for the 1 kΩ pull-up.
- **Master powered from 3.3 V but bus needs 12 V.** TJA1027 needs VBAT 7–18 V on the BAT pin (not the 3.3 V VIO). Without it, the transceiver doesn’t drive.
- **Mixed checksum types.** LIN 1.x checksum = sum of data only. LIN 2.x = sum of PID + data. A LIN 1.x slave on a 2.x bus rejects every frame because the checksum doesn’t include the PID.
- **PID parity wrong.** Easy to invert the parity bits. Compute and verify with a known-good table.
- **Linux UART read latency.** When polling a slave, the master sends header then must read 1–9 bytes within 50 ms or so. A loaded Linux box may delay the read. The response is buffered but you miss the timing window for the next frame. Use higher process priority or RT scheduling.
- **TIOCSBRK on some UART drivers is too long.** Always scope-verify the break length.
- **Multiple masters.** LIN has exactly one master. Two masters on one bus is undefined. Their schedules collide.
- **Bus-off via short.** A shorted-to-ground LIN line (a chafed wire in a car loom) puts the bus permanently dominant. No comms possible. Detection: the master’s read-back of its own sync byte is 0x00. Recover by raising EN to put the transceiver in low-power mode.
- **Different car manufacturers use slightly different sleep timings.** VW’s “wake-up pulse must be ≥250 µs” vs BMW’s “≥150 µs vs full break”. Follow the slave datasheet.

109.11 Going deeper

- **ISO 17987, LIN specification** (all 7 parts).
- **LIN consortium** specification archives (LIN 1.3 / 2.0 / 2.1 / 2.2A).
- **NXP TJA1027 datasheet + application note**: the standard 3.3 V LIN transceiver.
- **`drivers/tty/serial/imx.c`**: see how the i.MX UART implements TIOCSBRK / TIOCCBRK.
- **Microchip MCP2003 + MCP2004 application notes**: alternative transceivers + reference designs.

- `lin-bus.org` **forum + GitHub**: community LIN code and dumps.
- **Ch 108 (RS-485)**: the industrial cousin. Ch 110 (CAN), the heavyweight cousin.
- **Vector CANoe / LIN.SimulationKit**: commercial tools (expensive). Great for reverse-engineering automotive LIN systems if you have access.

> Next chapter: **Chapter 110: CAN deep dive**, CAN-FD, ISO-TP, SocketCAN advanced, MCP2515.

Chapter 110: CAN deep dive

What: the deep follow-up to Ch 55C's FlexCAN intro. We compare CAN transceivers, **NXP TJA1051** (5 V classic CAN), **TJA1463** (CAN-FD with fast bit timing), **Microchip MCP2562** (5 V or 3.3 V flexible), plus the **MCP2515** SPI-CAN-controller for adding CAN to a board with no spare FlexCAN. We dig into the **CAN-FD** frame format and why arbitration vs data phase have different baud rates, **ISO-TP (ISO-15765-2)** for multi-frame transport (the basis of OBD-II / UDS diagnostics), **SocketCAN** advanced features (BCM = Broadcast Manager, J1939 daemon, CAN-XL preview), and an end-to-end **OBD-II diagnostic tool** that reads engine RPM, coolant temp, and DTCs from a real car.

Why: CAN is everywhere, every car since 2008 (US OBD-II mandate), most industrial automation, every drone autopilot, half the modern medical devices, every BLDC servo. Mainline Linux's SocketCAN is the most complete CAN stack of any general-purpose OS: every interface looks like a network device, packets are skbs, you read/write via `sendto`/`recvfrom` on a `PF_CAN` socket. After this chapter you can take on most vehicle and industrial CAN integration work.

Focus: Classic CAN is a bit-stuffed differential bus with priority arbitration via CSMA/CR. CAN-FD adds a second, faster bit rate during the data phase. The result: 64 bytes through a 1 Mbps arbitration bus in about 120 µs. The kernel handles bit-timing, error recovery, bus-off detection. The hard parts you write: ISO-TP segmentation for messages >8 bytes (every diagnostic command is one), filters to subset thousands of frames/second down to what your app cares about, and BCM cyclic-broadcast for periodic frames (heart-beats, control loops). Get the bit timing right (sample point, SJW, prop and phase segments). Otherwise you will see CAN bus errors that look like a wiring fault but are caused by configuration.

Tooling. This chapter uses `can-utils` (full suite), `libsocketcan`, optional `python-can`.

- **Ubuntu-base (target):** `apt install can-utils libsocketcan-dev python3-can`
- **Buildroot:** `BR2_PACKAGE_CAN_UTILS=y BR2_PACKAGE_LIBSOCKETCAN=y BR2_PACKAGE_PYTHON3_PYTHON_CAN=y` **Buildroot:** a configuration-driven build system that produces a complete root filesystem and related images.
- Full per-tool reference: Userspace tooling appendix. **MCU bridge:** Think of the rootfs as the firmware image's file-backed runtime environment. On an MCU you link everything into flash. On Linux, programs and config live in this mounted tree. **rootfs:** root filesystem, the directory tree mounted at / that contains /bin, /etc, /dev, and libraries.

110.1 Transceiver and controller comparison

	TJA1051	TJA1463	MCP2562	MCP2515 (with TJA1051)
Type	transceiver	transceiver	transceiver	**standalone controller + transceiver**
Speed	1 Mbps (classic)	5 Mbps (CAN-FD data phase)	1 Mbps	1 Mbps
CAN-FD	no	yes	no	no
Supply	5 V	5 V (3.3 V opt)	5 V or 3.3 V	5 V controller, 3.3 V SPI
Need on-SoC controller	yes (FlexCAN)	yes (FlexCAN with FD)	yes	no, SPI from host
Use case	basic CAN	modern auto + CAN-FD	flexible designs	adding CAN to a board with no FlexCAN
Cost	$0.80	$1.50	$1.00	$4 (MCP2515) + $0.80 (TJA1051)

Pick guide:

- **TJA1051** + i.MX6ULL FlexCAN, default for classic CAN at ≤ 1 Mbps.
- **TJA1463**: when the bus uses CAN-FD (modern cars, new industrial designs).
- **MCP2515 SPI controller**: when both i.MX6ULL FlexCAN1 + FlexCAN2 are used and you need a third CAN bus.

110.2 Classic CAN frame (refresher)

```
SOF | Arbitration (ID + RTR) | Control | Data | CRC | ACK | EOF
1 bit  11 or 29 bits + 1     6 bits   0–64b  16b   2b   7b
```

- **Arbitration**: lower ID wins. Loser backs off. Winner continues uninterrupted. This is **CSMA/CR** (collision resolution, not avoidance).
- Dominant is logical 0 (actively driven). Recessive is logical 1 (idle). Dominant always overwrites recessive, so the lowest-numbered ID wins arbitration.
- **Bit-stuffing**: 5 consecutive identical bits → insert opposite bit. Receiver removes it. Adds ~20 % overhead.
- **CRC15**: protects the frame.
- **ACK slot**: a single bit. Receivers pull it dominant if they got the frame. Lack of ACK = error.

CAN-FD extends this:

- Optional second bit rate (5+ Mbps) during data phase.
- Data length up to 64 bytes (vs 8).
- 17- or 21-bit CRC.
- Bus-load math: classic CAN at 500 kbps = ~5500 frames/s. CAN-FD at 500 kbps arb / 5 Mbps data = ~9000 of-up-to-64-byte frames/s = ~10× the throughput.

110.3 Bit timing, the hidden trap

Each CAN bit is divided into segments:

```
|– Sync –|– Prop –|– Phase1 –|– Phase2 –|
    1       1–8      1–8        2–8         (TQ = Time Quanta)
                                ↑
                          sample point (typ. 75–87.5 %)
```

The clock = `CAN_clk = SoC_clk / (prescaler × TQ)`. For 500 kbps on a 33 MHz CAN clock:

- 33 MHz / 500 kbps = 66 TQ per bit → too many.
- Use prescaler 4: 33 / 4 = 8.25 MHz → 8.25 / 0.5 = 16.5 → round to 16 TQ per bit.
- Then split 16 TQ: 1 Sync + 7 Prop + 4 Phase1 + 4 Phase2 → sample at (1+7+4)/16 = 75 %.

DT for FlexCAN:

```
&can1 {
    pinctrl-names = "default";
    pinctrl-0 = <&pinctrl_can1>;
    xceiver-supply = <&reg_can_xcvr>;
    status = "okay";
};
```

The kernel SocketCAN handles bit-timing computation automatically when you set:

```
ip link set can0 type can bitrate 500000 sample-point 0.875
ip link set can0 up
```

For CAN-FD:

```
ip link set can0 type can bitrate 500000 dbitrate 2000000 fd on
ip link set can0 up
```

Verify the actual programmed timing:

```
ip -d link show can0
# 6: can0: <NOARP,UP,LOWER_UP,ECHO> mtu 16 ...
#   can state ERROR-ACTIVE (berr-counter tx 0 rx 0) restart-ms 0
#   bitrate 500000 sample-point 0.875 ...
```

A wrong sample point + long bus = sporadic errors that look intermittent. Set sample-point explicitly when troubleshooting.

110.4 Bringing up CAN with SocketCAN

```
modprobe flexcan       # for FlexCAN
modprobe can-raw
modprobe can-bcm
modprobe can-isotp     # for ISO-TP

ip link set can0 type can bitrate 500000
ip link set can0 up

ip link show can0      # state ERROR-ACTIVE = up + healthy

# Dump
candump can0
# can0  123   [8]  DE AD BE EF 00 01 02 03

# Send
cansend can0 123#DEADBEEF00010203

# Generate test traffic
cangen can0 -g 10
```

`can-utils` is the SocketCAN swiss army knife:

- `candump`: sniff
- `cansend`: send one frame
- `cangen`: generate
- `canplayer`: replay a log
- `cansniffer`: diff display (highlight changed bytes)
- `cangw`: kernel CAN-to-CAN gateway with filters
- `isotpsend` / `isotprecv`, multi-frame ISO-TP

110.5 Filters, surviving high-rate buses

A car CAN bus at 500 kbps carries 1500+ frames/s. Reading every frame on a Cortex-A7 is fine, but processing every one wastes CPU. **Filters** in the kernel drop unwanted frames before they hit user-space.

```
int s = socket(PF_CAN, SOCK_RAW, CAN_RAW);
struct sockaddr_can addr = { .can_family = AF_CAN };
struct ifreq ifr; strcpy(ifr.ifr_name, "can0"); ioctl(s, SIOCGIFINDEX, &ifr);
addr.can_ifindex = ifr.ifr_ifindex;
bind(s, (struct sockaddr*)&addr, sizeof addr);

/* Filter: accept only IDs 0x7E0..0x7EF (OBD-II range) */
struct can_filter filt[] = {
    { .can_id = 0x7E0, .can_mask = 0x7F0 },     /* match 0x7Ex */
};
setsockopt(s, SOL_CAN_RAW, CAN_RAW_FILTER, &filt, sizeof filt);

struct can_frame frame;
while (read(s, &frame, sizeof frame) > 0) {
    printf("ID=%X DLC=%d data[0]=%02X\n",
           frame.can_id, frame.can_dlc, frame.data[0]);
}
```

The kernel applies the filter. User-space only sees matching frames. Critical for performance.

110.6 ISO-TP (ISO-15765-2)

CAN frames carry 8 bytes (64 for FD). Diagnostic messages are often 30+ bytes. ISO-TP fragments them.

Three frame types:

- **SF (Single Frame)**: payload ≤ 7 bytes, sent in one CAN frame.

- **FF (First Frame)**: payload > 7 bytes, contains total length + first 6 bytes.
- **CF (Consecutive Frame)**: subsequent fragments with a 4-bit sequence number.
- **FC (Flow Control)**: receiver tells sender block size + separation time.

The kernel `can-isotp` module exposes this as a SOCK_DGRAM:

```
int s = socket(PF_CAN, SOCK_DGRAM, CAN_ISOTP);
struct sockaddr_can addr = { .can_family = AF_CAN };
addr.can_ifindex = ifr.ifr_ifindex;
addr.can_addr.tp.tx_id = 0x7E0;     /* ECM request ID */
addr.can_addr.tp.rx_id = 0x7E8;     /* ECM response ID */
bind(s, ...);

/* Send a UDS read-DID command */
uint8_t cmd[] = { 0x22, 0xF1, 0x90 };       /* Read DID: VIN */
send(s, cmd, sizeof cmd, 0);

uint8_t resp[256];
int n = recv(s, resp, sizeof resp, 0);
/* resp[0] = 0x62 (positive response); resp[1..2] = 0xF1 0x90; resp[3..19] = VIN */
```

That’s a 19-byte response (positive response + DID + 17-byte VIN) auto-fragmented and reassembled by the kernel. Without ISO-TP, you’d be writing the segmentation state machine yourself.

110.7 J1939, heavy-duty truck protocol

J1939 layers on CAN with 29-bit IDs encoding source + destination + parameter group number (PGN). Used in trucks, agricultural machinery, marine engines.

```
modprobe can-j1939
ip link set can0 type can bitrate 250000
ip link set can0 up

# Talk to source address 0x80, listen for SPN-based messages
# j1939_set_address(s, 0x80)
# ... j1939-utils provides example sender/receiver
```

J1939’s PGN dictionary lists thousands of standardized parameters (engine RPM, fuel consumption, brake pressure, …). For agricultural/marine integrations, this is standard.

110.8 BCM, Broadcast Manager for periodic frames

Need to send a frame every 20 ms for a control loop? BCM does it in kernel:

```
int s = socket(PF_CAN, SOCK_DGRAM, CAN_BCM);
connect(s, (struct sockaddr*)&addr, sizeof addr);

struct {
```

```
    struct bcm_msg_head head;
    struct can_frame frame;
} msg = {0};
msg.head.opcode = TX_SETUP;
msg.head.flags = SETTIMER | STARTTIMER;
msg.head.can_id = 0x123;
msg.head.nframes = 1;
msg.head.ival2.tv_sec = 0;
msg.head.ival2.tv_usec = 20000;        /* 20 ms cycle */
msg.frame.can_id = 0x123;
msg.frame.can_dlc = 8;
memcpy(msg.frame.data, "HEARTBT!", 8);
send(s, &msg, sizeof msg, 0);
```

Now the kernel emits this frame every 20 ms forever. User-space process can sleep. Useful for safety heartbeats, periodic control updates, slow telemetry.

110.9 MCP2515, adding CAN to a board with no FlexCAN

> **Privilege boundary:** $ means normal user. # or sudo means root and can change host or target state. After a privileged command, verify the expected device, service, or file appears before continuing. Roll back by undoing the config change or stopping the service you just enabled.

If you've used both i.MX6ULL FlexCANs and need a third bus, the MCP2515 is the standard SPI add-on. Kernel driver `drivers/net/can/spi/mcp251x.c`.

DT:

```
&ecspi3 {
    cs-gpios = <&gpio4 26 GPIO_ACTIVE_LOW>;
    status = "okay";

    can@0 {
        compatible = "microchip,mcp2515";
        reg = <0>;
        spi-max-frequency = <10000000>;
        clocks = <&clk16m>;             /* 16 MHz crystal */
        interrupts-extended = <&gpio4 27 IRQ_TYPE_LEVEL_LOW>;
    };
};

clk16m: clk16m {
    compatible = "fixed-clock";
    #clock-cells = <0>;
    clock-frequency = <16000000>;
};
```

After `modprobe mcp251x` you get `can1`, identical SocketCAN semantics to FlexCAN. Throughput limited by SPI: 10 MHz SPI handles 500 kbps CAN cleanly. Struggles near 1 Mbps under load.

110.10 OBD-II, talking to a real car

Plug an OBD-II to DB9 adapter into the car's port (under the dashboard). Connect to your i.MX6ULL's CAN1. Bitrate 500 kbps.

```
int s = socket(PF_CAN, SOCK_DGRAM, CAN_ISOTP);
struct sockaddr_can addr = { .can_family = AF_CAN };
addr.can_ifindex = if_nametoindex("can0");
addr.can_addr.tp.tx_id = 0x7DF;      /* OBD-II functional request */
addr.can_addr.tp.rx_id = 0x7E8;      /* ECU response (engine) */
bind(s, ...);

/* Mode 01 PID 0C: engine RPM */
uint8_t req[] = { 0x01, 0x0C };
send(s, req, sizeof req, 0);

uint8_t resp[8];
recv(s, resp, sizeof resp, 0);
/* resp[0]=0x41 mode echo, resp[1]=0x0C PID, resp[2]+resp[3] = RPM*4 */
int rpm = ((resp[2] << 8) | resp[3]) / 4;
printf("RPM: %d\n", rpm);
```

OBD-II mode/PID list:

- 01 0C → engine RPM
- 01 0D → vehicle speed
- 01 05 → coolant temp
- 01 0F → intake air temp
- 01 11 → throttle position
- 03 → diagnostic trouble codes (DTCs)
- 04 → clear DTCs
- 09 02 → VIN

About 200 lines of C and a Linux board are enough to build a complete OBD-II dashboard.

110.11 Key Notes

- CAN controller capability, transceiver capability and network protocol are separate choices. A CAN-FD-capable transceiver cannot turn the i.MX6ULL's classic FlexCAN controller into a CAN-FD controller.
- Arbitration priority and bus utilization shape worst-case delivery time. Nominal bitrate is not a deadline guarantee.
- SocketCAN filters, BCM, ISO-TP and J1939 address different application and transport needs. They do not replace physical termination, timing or fault handling.
- Diagnostic access and gateways can change a live system's behavior. Examples are protocol illustrations, not authorization or qualification for connecting to a vehicle.

110.12 Pitfalls

- **Wrong sample point.** A 75 % sample on a 50 m bus may sample inside the prop-delay. Bumping to 87.5 % cures "random errors." Always set explicit sample-point on long buses.
- **No termination.** Bus reflections cause CRC errors at higher speeds. 1 Mbps unterminated barely works at 1 m, breaks at 5 m.
- **Termination too close together.** 120 Ω at both physical ends. Not "120 Ω near the master + 120 Ω in the middle."
- **Mixing CAN and CAN-FD nodes.** Classic-only nodes see the FD flag bit as a form error and fault. Either all nodes do CAN-FD or none do (some modern transceivers tolerate FD frames as "noise" but not standardized).
- **MCP2515 silicon errata.** Some revs have known SPI-glitch bugs at high SPI clocks. Cap at 5 MHz on errata silicon.
- **Bit-stuffing eats throughput.** Effective bandwidth on classic CAN is 60–80 % of nominal due to stuffing + acks + interframe space.
- **ID 0 is highest priority.** A node that always sends ID 0 starves everyone else. Reserve low IDs for hard-real-time critical messages only.
- **No ACK on a single-node bus.** A node alone on the bus sends a frame. No other node ACKs. The transmitter errors and retries forever. Bus-off after ~256 errors. To debug solo: add a `cangen --rx-ack` simulator, or loopback.
- **`flexcan` driver vs CONFIG_CAN_CALC_BITTIMING.** Without CAN_CALC_BITTIMING in your kernel config, you must specify all timing parameters explicitly. Newer kernels enable it by default.

- **ISO-TP block size = 0 (no flow control).** Some ECUs send STmin and BS=0 (means “send everything as fast as you can”). Your transmit loop may overwhelm slower receivers. The receiver’s flow control says BS=1, respect it.
- **J1939 source address claims.** Multi-master J1939 requires arbitration of source addresses at startup. Out-of-the-box examples may skip this and you get address conflicts on a real bus.
- **CAN_RAW_LOOPBACK on by default.** Sent frames are echoed back to your own socket. This surprises people writing CAN code for the first time.

110.13 Going deeper

- **ISO 11898-1/2 (classic CAN), ISO 11898-1:2015 (CAN-FD)**: physical and data link layers.
- **ISO 15765-2 (ISO-TP)**: multi-frame transport for diagnostics.
- **ISO 14229 (UDS)**: Unified Diagnostic Services. The application protocol over ISO-TP.
- **SAE J1939**: heavy-duty truck protocol.
- **SAE J1979**: OBD-II legislated emissions PIDs.
- **`drivers/net/can/`** in the kernel. Especially `flexcan.c`, `mcp251x.c`, `mcp251xfd.c` (CAN-FD).
- **`net/can/`** for SOCK_RAW, SOCK_DGRAM CAN_BCM, CAN_ISOTP, CAN_J1939.
- **`can-utils`**: the canonical user-space toolkit.
- **`python-can`**: Python bindings to SocketCAN. Great for prototyping.
- **`SavvyCAN`**: open-source CAN analyzer with reverse-engineering tools.
- **Hardware: Peak PCAN, Kvaser, Vector**: reference-grade USB-CAN analyzers when you need certainty.

> Next chapter: **Chapter 111: Quadrature encoders & rotary**, Group R (Motors & encoders) starts here.

Chapter 111: Quadrature encoders & rotary

What: quadrature incremental encoders (optical or magnetic, two phase-shifted square waves), **mechanical rotary encoders** (knob-style, low-res), and **absolute magnetic encoders** (AS5048A from Ch 74, here used as a position sensor not a sensor-out). Three implementations on the i.MX6ULL: (a) software quadrature decode via two GPIO IRQs (works ≤ ~10 kHz pulse rate), (b) hardware quadrature via the i.MX **XBAR + ENC** peripheral (works up to ~1 MHz pulse rate but needs the optional ENC IP block), (c) `rotary_encoder` kernel driver for low-rate user-interface knobs. Plus the IIO `angl` channel pattern for absolute encoders. **MCU bridge:** Think of Linux GPIO like the same pin set/reset block you used on STM32, but accessed through a kernel subsystem that owns numbering, direction, interrupts, and user-space exposure. **IIO:** Industrial I/O, Linux's subsystem for sensors, ADCs, DACs, and buffered sampled data. **GPIO:** General-Purpose Input/Output, a pin controlled as a digital input, output, or interrupt source.

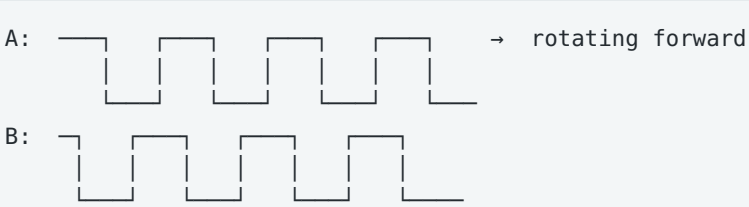
Why: every closed-loop motor needs position feedback. Every UI knob needs decoding. The i.MX6ULL has a strong NXP-BSP-only ENC peripheral that mainline Linux doesn't fully expose, so on a mainline kernel, you either do software decode (cheap, slow), use the partial mainline driver if it exists for your variant, or bridge to an external decoder IC. Understanding the four implementation tiers (software IRQ, hardware decoder, dedicated chip, sysfs `rotary_encoder` driver) lets you pick correctly for your use case. **MCU bridge:** Think of an IRQ like an EXTI/NVIC interrupt path, except Linux splits the hard interrupt from deferred work and must share lines across drivers. **IRQ:** interrupt request, the signal path that tells the CPU or interrupt controller that hardware needs service. **BSP:** Board Support Package: vendor patches, configs, bootloader files, and scripts needed to boot one board. **sysfs:** a kernel-generated filesystem under /sys that exposes devices, drivers, and attributes.

Focus: Quadrature decoding works on two channels that are 90° out of phase. The leading channel tells you the direction of rotation. A is high then B goes high = forward. B high then A high = backward. Four edges per full A+B cycle = 4× resolution multiplication. Mechanical encoders bounce badly, 5 ms or more of noise per click, and need debouncing. Optical encoders are clean but expensive. Magnetic encoders are the middle ground.

111.1 Encoder types

	Mechanical (knob)	Optical incremental	Magnetic incremental	Magnetic absolute
Resolution	12–24 ppr	100–10,000 ppr	64–4096 ppr	12–14 bits / rev
Max RPM	hand-spinning	10,000+	30,000+	60,000+
Index pulse	sometimes	yes	optional	n/a (absolute)
Bouncing	severe	none	none	none
Cost	$0.50 (Bourns PEC11)	$20–200	$5–50	$5 (AS5600) to $25 (AS5048A)
Use case	UI knob	CNC, servo	hobby/industrial servo	absolute position (joint angle)

111.2 Quadrature signal, the math




```
   On A's rising edge:  if B==0 then count++   else count--
   On A's falling edge: if B==1 then count++   else count--
   On B's rising edge:  if A==1 then count++   else count--
   On B's falling edge: if A==0 then count++   else count--
```

That’s 4× decoding (every edge of either channel = one count). The encoder spec (“100 PPR”) refers to 100 pulses per revolution of just one channel. 4× decoded → 400 counts/rev.

State-machine view (current AB state + previous AB state):

Prev	Curr	Direction
00	01	+1
01	11	+1
11	10	+1
10	00	+1
00	10	−1
10	11	−1
11	01	−1
01	00	−1
??	same	0
00	11, 01→10, etc.	INVALID (missed an edge)

Software-decode in C:

```
static volatile int32_t count = 0;
static uint8_t prev_ab = 0;
static const int8_t QDEC_TABLE[16] = {
    /* prev:curr  00 01 10 11 */
    /* prev 00 */  0, +1, -1,  0,
    /* prev 01 */ -1,  0,  0, +1,
    /* prev 10 */ +1,  0,  0, -1,
    /* prev 11 */  0, -1, +1,  0,
};

static void encoder_irq_handler(void) {
    uint8_t a = gpio_read(GPIO_A);
    uint8_t b = gpio_read(GPIO_B);
    uint8_t curr_ab = (a << 1) | b;
    count += QDEC_TABLE[(prev_ab << 2) | curr_ab];
    prev_ab = curr_ab;
}
```

Both channels call this on rising and falling edges. The 12 valid transitions produce ±1. The 4 invalid transitions (where both bits changed at once = missed edge) produce 0.

111.3 Software decode in user-space (libgpiod)

For low-rate encoders (≤10 kHz pulses, which covers most UI and slow-servo uses):

```
/* qdec_userspace.c */
#include <gpiod.h>
#include <stdio.h>
#include <poll.h>

int main(void) {
    struct gpiod_chip *chip = gpiod_chip_open("/dev/gpiochip0");
    struct gpiod_line *a = gpiod_chip_get_line(chip, 20);
    struct gpiod_line *b = gpiod_chip_get_line(chip, 21);

    struct gpiod_line_request_config cfg = {
        .consumer = "qdec",
        .request_type = GPIOD_LINE_REQUEST_EVENT_BOTH_EDGES,
        .flags = 0,
    };
    gpiod_line_request(a, &cfg, 0);
    gpiod_line_request(b, &cfg, 0);

    struct pollfd pfd[2] = {
        { .fd = gpiod_line_event_get_fd(a), .events = POLLIN },
        { .fd = gpiod_line_event_get_fd(b), .events = POLLIN },
    };
    int32_t count = 0;
    uint8_t prev = (gpiod_line_get_value(a) << 1) | gpiod_line_get_value(b);

    static const int8_t TBL[16] = {
        0,+1,-1, 0, -1, 0, 0,+1, +1, 0, 0,-1, 0,-1,+1, 0
    };

    for (;;) {
        poll(pfd, 2, -1);
        struct gpiod_line_event ev;
        if (pfd[0].revents) gpiod_line_event_read(a, &ev);
        if (pfd[1].revents) gpiod_line_event_read(b, &ev);
        uint8_t curr = (gpiod_line_get_value(a) << 1) | gpiod_line_get_value(b);
        count += TBL[(prev << 2) | curr];
        prev = curr;
        printf("\rcount=%d   ", count);
        fflush(stdout);
    }
}
```

The `gpiod_line_event_get_fd` lets you poll multiple GPIO lines via standard `poll()`. Latency per edge: ~50–500 µs on a non-RT i.MX6ULL, fine for hand-spun knobs, fails for a 10,000 ppr encoder spinning at 60 RPM (= 40 kHz edge rate, exceeds Linux's interrupt budget).

For higher rates: kernel driver, hardware decoder, or external decoder chip.

111.4 Kernel `rotary_encoder` driver, for UI knobs

For UI-style mechanical encoders (10–30 detents per turn), the kernel provides `drivers/input/misc/rotary_encoder.c`, wires two GPIOs and exposes the knob as a Linux input device.

DT:

```
rotary@0 {
    compatible = "rotary-encoder";
    pinctrl-names = "default";
    pinctrl-0 = <&pinctrl_rotary>;
    gpios = <&gpio4 20 GPIO_ACTIVE_LOW>,        /* A */
            <&gpio4 21 GPIO_ACTIVE_LOW>;        /* B */
    linux,axis = <REL_X>;                       /* report as REL_X relative event */
    rotary-encoder,steps-per-period = <4>;      /* 4 for full-quadrature */
    rotary-encoder,encoding = "gray";            /* or "binary" */
    rotary-encoder,relative-axis;
};
```

A new `/dev/input/eventN` appears. Use `evtest /dev/input/event3`:

```
Event: time 1709236745.123, type 2 (EV_REL), code 0 (REL_X), value 1
Event: time 1709236745.234, type 2 (EV_REL), code 0 (REL_X), value 1
Event: time 1709236745.345, type 2 (EV_REL), code 0 (REL_X), value -1
```

This is the right pattern for a volume knob, menu scroll, jog wheel. Bouncing? The driver does some debouncing internally. For very bouncy mechanical encoders, add an RC filter on the GPIO inputs.

111.5 i.MX hardware quadrature, the ENC peripheral (and its mainline state)

The i.MX6ULL **eXtended Quadrature Encoder (ENC)** module, 32-bit counter, hardware-accelerated direction detection, index-pulse support, count up to ~10 MHz edge rate. Documented in NXP's reference manual ch. 33-ish.

Status in mainline Linux:

- **NXP BSP kernels**: full support via `drivers/staging/iio/quadrature_encoder.c` or vendor patches.
- **Mainline**: partial. Some i.MX variants (i.MX7, i.MX8) have IIO-exposed encoder counters. I.MX6ULL coverage varies by kernel version. Check `find /sys/bus/iio/devices/ -iname '*encoder*'` after enabling `CONFIG_IIO_ST_LSM6DSX` or relevant config.

If your kernel has it, exposing via DT:

```
&qei1 {
    pinctrl-0 = <&pinctrl_qei1>;
    status = "okay";
};
```

User-space reads the count via IIO:

```
cat /sys/bus/iio/devices/iio:device0/in_count0_raw
```

If your mainline doesn't include i.MX6ULL ENC, fall back to GPIO-IRQ software decode (≤10 kHz) or an external decoder chip (LS7366R via SPI for unlimited speed).

111.6 External SPI decoder, LSI/CSI LS7366R

For "I need a fast quadrature counter and my SoC doesn't have one":

```
   Encoder A,B,Z   →  LS7366R (32-bit counter, debouncer, SPI interface)
                      i.MX6ULL ← SPI ← LS7366R
```

LS7366R is a $5 chip. 4× decode in hardware. 32-bit counter. Up to 40 MHz pulse rate. Driver: write your own ~200-line userspace SPI driver. Sequence: write MDR0 register (filter, decode mode), then `READ_CNTR` over SPI returns the current count.

No mainline driver, but SPI access is straightforward.

```
uint8_t cmd = 0x88;      /* CLR | CNTR */
write(spi, &cmd, 1);     /* clear counter */
uint8_t read_cnt[5] = { 0x60, 0,0,0,0 };  /* READ_CNTR + 4 dummy */
uint8_t resp[5];
spi_transfer(read_cnt, resp, 5);
int32_t count = (resp[1] << 24) | (resp[2] << 16) | (resp[3] << 8) | resp[4];
```

For multi-axis: chain multiple LS7366Rs on the same SPI bus with separate CS.

111.7 Absolute encoders, AS5048A pattern

AS5048A (Ch 74), magnetic absolute, 14-bit / revolution, SPI interface. Mainline IIO driver: `drivers/iio/position/ams5048.c` exposes `in_angl0_raw`:

```
cat /sys/bus/iio/devices/iio:device0/in_angl0_raw
# 8192       (out of 16384 = 180°)
cat /sys/bus/iio/devices/iio:device0/in_angl0_scale
# 0.000383   (radians per count: 2π / 16384)
```

For position control, absolute encoders are better in nearly every case, no homing, no missed-count drift, instant boot-up state. The trade: cost ($5 vs $0.50) and harder mounting (needs precisely-placed magnet on the shaft).

111.8 Closed-loop motor control, the velocity-feedback example

A 1024-ppr encoder on a brushed DC motor. Goal: maintain constant 1000 RPM regardless of load.

```
/* Velocity loop running at 100 Hz (every 10 ms) */
int32_t prev_count = 0;
double target_rpm = 1000.0;
double Kp = 0.5, Ki = 0.1;
double integral = 0;

for (;;) {
    int32_t count = read_encoder();
    int32_t delta = count - prev_count;
    prev_count = count;

    /* delta counts in 10 ms × 60/(0.01*1024) = RPM */
    double rpm = delta * 60.0 / (0.01 * 1024);
    double err = target_rpm - rpm;
    integral += err * 0.01;
    double pwm = Kp * err + Ki * integral;
    if (pwm > 100) pwm = 100;
    if (pwm < 0)   pwm = 0;
    set_pwm_duty(pwm);

    usleep(10000);
}
```

A 100 Hz control loop is adequate for a brushed motor. Mechanical dynamics are much slower than that. PWM drives the H-bridge from Ch 112.

> **MCU bridge:** Think of Linux PWM like an MCU timer output channel, except the driver exposes period, duty cycle, polarity, and enable state through a subsystem.
> **PWM:** Pulse-Width Modulation, a timer output whose duty cycle controls average power or encodes timing.

111.9 Key Notes

- Quadrature encoders report relative transitions, while absolute encoders report a position within their defined range. Homing and accumulated turns are different state problems.
- Edge rate, debounce and capture timing determine whether software decoding can preserve counts. Increasing thread priority does not establish a no-loss bound.

- A hardware counter or external decoder can own fast edge capture while Linux reads accumulated state. Its existence and support must be established for the actual SoC and board.
- Count scaling, direction, index handling and coordinate origin must agree with the control system. A plausible number is not evidence of a safe motion loop.

111.10 Pitfalls

- **GPIO interrupt latency too high.** At 40 kHz edges, the IRQ handler takes 5–20 µs on a Cortex-A7. Above that, edges are lost. Use a hardware decoder.
- **No pull-ups on encoder inputs.** Open-collector outputs from optical encoders need external 4.7 kΩ pull-ups. Without, signals float and you see ghost counts.
- **No debouncing on mechanical encoder.** 5 ms of bounce per detent → 100s of false counts per click. Use RC filter or the kernel driver's debounce.
- **Reversed A/B.** Direction is wrong. Swap two wires in software or DT.
- **Index pulse double-trigger.** Z is often longer than one count's worth of time. Trigger on its rising edge only, or you'll home twice per revolution.
- **Encoder loses counts on power loss.** Incremental encoders need re-homing after every boot. Use absolute encoders (AS5048A) or a homing routine.
- **User-space decode misses edges under CPU load.** A user-space loop on a busy Linux box loses edges when the scheduler holds it. RT priority (`chrt -f 99`) or move to a kernel module.
- **AS5048A magnet alignment.** The magnet must be on the shaft axis (radially polarized, diametric) within ±0.5 mm. Misalignment → non-linear angle errors of several degrees.
- **Index of LS7366R registers wrong.** MDR0 vs MDR1. Common mistake. Default values don't enable filters → noisy signals cause false counts.

111.11 Going deeper

- **`Documentation/devicetree/bindings/input/rotary-encoder.txt`**: kernel rotary encoder binding.
- **`drivers/input/misc/rotary_encoder.c`**: readable source.
- **NXP IMX6ULL Reference Manual, ch. 33 (ENC)**: full peripheral docs.
- **LS7366R datasheet**: 12-page chip. SPI register tour.
- **AMS AS5048A datasheet**: absolute encoder.

- **Bourns PEC11 series**: the classic UI knob.
- **`libgpiod`**: for user-space GPIO event-based code.
- **Ch 74**: magnetic position sensors (same AS5048A chip).
- **Ch 112**: pairs with this for closed-loop motor control.

> Next chapter: **Chapter 112: Stepper & DC motor drivers**, DRV8825, TMC2209, BTS7960, BLDC FOC.

Chapter 112: Stepper & DC motor drivers

What: the four motor-driver families: **stepper** (DRV8825, A4988, basic step/dir. TMC2209, silent stallGuard UART config), **DC brush** (BTS7960, 43 A H-bridge), **BLDC** (DRV8302, trapezoidal / sinusoidal / FOC), and **servo** (PWM-controlled hobby servos). On the i.MX6ULL we drive a stepper via PWM-step + GPIO-dir, configure TMC2209's RMS current and microstepping over UART, run a closed-loop velocity on a brushed motor with PWM + encoder feedback (Ch 111), and drive a BLDC with trapezoidal commutation. Emphasis on the **electrical safety + thermal limits** that are easy to overlook and fatal to ignore. **MCU bridge:** Think of Linux PWM like an MCU timer output channel, except the driver exposes period, duty cycle, polarity, and enable state through a subsystem. **MCU bridge:** Think of Linux GPIO like the same pin set/reset block you used on STM32, but accessed through a kernel subsystem that owns numbering, direction, interrupts, and user-space exposure. **PWM:** Pulse-Width Modulation, a timer output whose duty cycle controls average power or encodes timing. **GPIO:** General-Purpose Input/Output, a pin controlled as a digital input, output, or interrupt source.

Why: any product that *moves* needs one of these. 3D printers (steppers), automated blinds (DC), drone ESCs (BLDC), CNC mills (everything), conveyor belts, robotic arms, automated valves, motor control is its own discipline. Linux makes the *control* easy (Cortex-A7 has plenty of MIPS for PID). The hard parts are the silicon-protection details, the EMI from chopping inductive currents, and the mechanical resonance of the load. Get these wrong and the result ranges from an audible whine to a destroyed MOSFET.

Focus: Steppers need precise step-rate generation, usually from a PWM. DC motors need a PWM and an H-bridge, with current feedback for control. BLDC motors need commutation, switching three half-bridges in sync with rotor position. The kernel's PWM framework (Ch 48) handles step generation for steppers. For DC/BLDC closed-loop, you'll bolt together: encoder (Ch 111) + PID + PWM + current sense (INA226 from Ch 75). For absolute torque control or smooth low-RPM behaviour, you need FOC (Field-Oriented Control). The current loop runs at 10 kHz or higher, which exceeds Linux's scheduling jitter. Most engineers offload FOC to a dedicated MCU, SimpleFOC on STM32 is the open-source standard.

112.1 Driver type pick-guide

Motor	Best driver	When
NEMA17 stepper, 1 A, low noise	TMC2209	3D printer, lab automation, anywhere quiet matters
NEMA17 stepper, 1 A, BOM critical	DRV8825 / A4988	cost-sensitive, noise OK
NEMA23+ stepper, 3+ A	TMC2226 / DM542 (external driver)	larger machines
Small DC brush motor, ≤ 1 A	TB6612FNG	simple bidirectional control
Large DC brush motor, 5–40 A	BTS7960	electric scooters, automation actuators
BLDC outrunner (drone motor)	DRV8302 / ODrive ESC	low-RPM torque, smooth
RC servo (positioning)	(no driver, PWM direct)	hobby pan-tilt, robotic joints

112.2 Steppers, step/dir interface

A stepper rotates 1.8° (200 steps) or 0.9° (400 steps) per full step. Microstepping (1/2, 1/4, …, 1/256) interpolates between full steps for smoother motion and finer resolution (at lower torque per microstep).

Driver interface (DRV8825, A4988):

- **STEP** pin, rising edge advances one (micro)step.
- **DIR** pin, 0 = CW, 1 = CCW.
- **ENABLE** pin (active low), disables the outputs (motor freewheels).
- **M0, M1, M2** pins, microstep configuration (000 = full, 001 = 1/2, …, 101 = 1/32).
- **SLEEP, RESET**: typically tied high.

Wiring:

```
   i.MX6ULL PWM ──────── STEP            ┌──────── A+ ─── motor coil A ─── A–
   GPIO         ──────── DIR             │       B+ ─── motor coil B ─── B–
   GPIO         ──────── ENABLE     DRV8825         (motor terminals)
   GPIO ×3      ──────── M0/M1/M2     │       Vmot (12–45 V) + bulk cap
                                      └──────  GND
   Logic supply 3.3/5 V
```

The PWM generates step pulses. The driver chops the current. Current is set by a trimpot (or in TMC2209 via UART).

Critical: every driver has a “current limit” you set with a trimpot (sense resistor + reference voltage). Set too high → driver overheats and shuts down. Set too low → motor stalls. For NEMA17 + DRV8825 + 0.1 Ω sense → Vref = Imax × 5 × 0.1 = ~0.4 V for 0.8 A current. Measure with multimeter on the trimpot wiper. Adjust slowly.

112.3 Stepper from Linux, PWM step generation

Use the kernel PWM (Ch 48) to generate steps:

```
# Configure PWM for 1 kHz step rate
echo 0 > /sys/class/pwm/pwmchip0/export
echo 1000000 > /sys/class/pwm/pwmchip0/pwm0/period       # 1 ms = 1 kHz
echo 500000  > /sys/class/pwm/pwmchip0/pwm0/duty_cycle   # 50 %
echo 1 > /sys/class/pwm/pwmchip0/pwm0/enable

# Set direction
gpioset gpiochip4 22=0     # CW

# At 1 kHz and 1/8 microstepping, motor rotates at 1000/(8×200) = 0.625 rev/s = 37.5 RPM
```

For variable speed (acceleration profile), modify the PWM period over time:

```
/* Accelerate from 100 Hz to 5 kHz over 1 second */
for (int hz = 100; hz <= 5000; hz += 100) {
    int period_ns = 1000000000 / hz;
    int duty_ns   = period_ns / 2;
    write_file("/sys/class/pwm/pwmchip0/pwm0/period", period_ns);
    write_file("/sys/class/pwm/pwmchip0/pwm0/duty_cycle", duty_ns);
    usleep(20000);                                       /* 20 ms per step */
}
```

For coordinated multi-axis motion (CNC, 3D printer) this isn’t enough, you need a lookahead planner (Marlin, Klipper architecture). Klipper runs the motion planner on Linux and offloads step generation to an STM32 over USB serial. This split is the dominant pattern for high-end CNC and 3D printing.

112.4 TMC2209, UART configuration for silent stepping

TMC2209 (Trinamic) is the current common choice for silent stepper drivers. Same step/dir interface as DRV8825, but a **UART configuration interface** lets you set:

- Microstepping (up to 1/256, smooth interpolation)
- RMS current (in mA, very precise)
- stealthChop (silent voltage-mode chop) vs spreadCycle (fast current-mode chop)
- stallGuard (sensorless homing, detects when motor stalls into an end-stop)
- CoolStep (auto-reduce current when load is low)

The UART is single-wire (TX and RX share). Driver auto-direction-switches:

```
   i.MX UART TX ──┬──── TMC2209 PDN_UART
                  │
                  └── 1 kΩ ──── i.MX UART RX
```

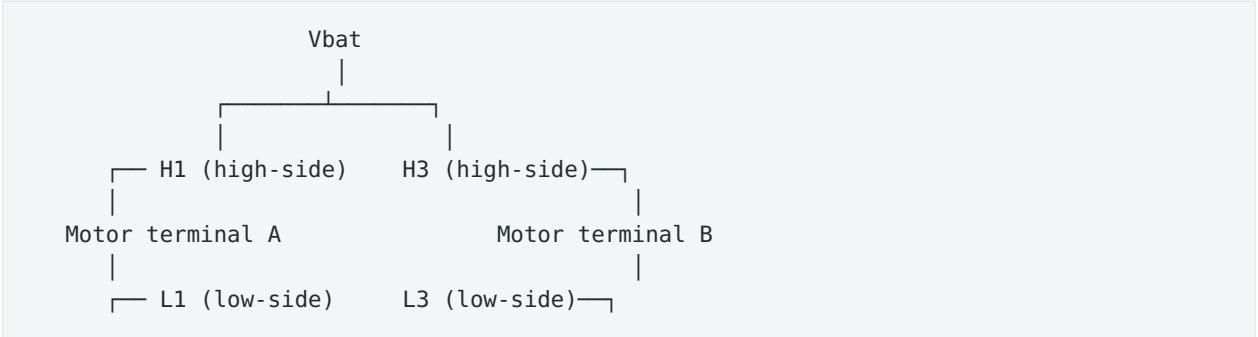
Linux driver: none in mainline (TMC's official driver is C++ and runs on STM32). For Linux, write a small user-space UART driver matching the TMC datagram format:

```
struct tmc_write {
    uint8_t sync = 0x05;       /* sync nibble + reserved nibble */
    uint8_t slave_addr;
    uint8_t reg_addr | 0x80;   /* MSB set = write */
    uint32_t data;
    uint8_t crc;               /* TMC's specific CRC8 */
};

/* Set RMS current = 800 mA */
tmc_write(addr=0, IHOLD_IRUN, (HOLD_CURRENT << 0) | (RUN_CURRENT << 8) | (IHOLDDELAY <<
16));
```

The TMC2209 datasheet (1MB PDF, very readable) walks every register. Once configured, the motor is near-silent. A 3D printer with TMC2209 drivers is roughly the volume of a fridge fan.

112.5 DC brushed motor, H-bridge + PWM



```
              │                 │
              └─────────┬───────┘
                        │
                       GND

   Forward:  H1+L3 on  (current A→B)
   Reverse:  H3+L1 on  (current B→A)
   Brake:    L1+L3 on  (both terminals shorted to GND)
   Coast:    all off (motor freewheels)
```

PWM the high-side (or both sides) to control voltage/torque. The BTS7960 packages all of this in two ICs (BTS7960 each is a half-bridge). You wire two together for full H-bridge.

```
   i.MX PWM1 → BTS7960 #1 RPWM (forward PWM)
   i.MX PWM2 → BTS7960 #2 RPWM (reverse PWM)
   i.MX GPIO → both enables
   Vbat (5–27 V) → both BTS7960 Vmot
   Motor → between the two BTS7960 outputs
```

To drive forward at 50 %:

```
echo 50% > /sys/class/pwm/pwmchip1/pwm0/duty_cycle  # forward PWM
echo 0   > /sys/class/pwm/pwmchip1/pwm1/duty_cycle  # reverse off
```

To brake: set both PWMs to 0 and assert enable. Both low-sides switch on → motor terminals shorted → fast brake.

Critical: never assert both high-sides (forward + reverse) simultaneously, that's a shoot-through short across Vbat. The BTS7960 has internal cross-conduction protection but external H-bridges from discrete MOSFETs need software interlock + dead-time.

112.6 BLDC, commutation 101

A BLDC motor has 3 stator coils and a permanent-magnet rotor. To spin, you cycle current through pairs of coils in synchronism with the rotor position:

```
   Rotor angle:    0°        60°       120°      180°      240°      300°
   Energize:       A+B-       A+C-      B+C-      B+A-      C+A-      C+B-
   (i.e., "A to B" means current from coil A's terminal to coil B's)
```

To know rotor position:

- **Hall sensors** (3 of them, integrated on the motor), gives 60° resolution → trapezoidal commutation.
- **Encoder**: finer resolution → sinusoidal commutation.
- **Sensorless back-EMF detection**: measure the un-energized coil's voltage. Zero-crossing tells you where you are. Works above ~10 % nominal speed.

DRV8302 (TI) is the gate driver for 3 half-bridges + provides current-sense amplifiers. You add 6 N-FETs and you have a BLDC ESC. The MCU (or Linux) runs commutation logic.

For Linux: the **timing is too tight for closed-loop control** at full speed. A BLDC at 10,000 RPM with 14 magnetic poles cycles 14 × 10,000 / 60 = 2,333 commutations/s. Each commutation must hit within ~50 µs of the right rotor angle, or torque ripple becomes audible. Linux interrupt jitter is typically > 100 µs.

Solution: offload commutation to a dedicated MCU. Use **SimpleFOC** (Arduino-based, STM32) or **ODrive** (a dedicated BLDC controller). Linux supervises (sends target speed via UART or USB). MCU does the kHz current loop.

For hobbyist trapezoidal commutation on a slow BLDC (< 1000 RPM):

```
/* Sensored trapezoidal commutation — 6-step */
uint8_t hall_state = (gpio_read(HALL_A) << 2) | (gpio_read(HALL_B) << 1) |
gpio_read(HALL_C);
/* Map hall_state (1..6, two invalid) to gate pattern */
static const uint8_t COMMUTATE_TBL[8] = {
    /* hall: 000  001  010  011  100  101  110  111 */
    /* gate: inv  4    2    3    6    5    1    inv  */
        0,   4,   2,   3,   6,   5,   1,   0,
};
uint8_t pattern = COMMUTATE_TBL[hall_state];
apply_gates(pattern, pwm_duty);
```

This runs on hall-IRQ → at 1000 RPM × 14 poles = 234 IRQ/s. Linux can handle that.

> **MCU bridge:** Think of an IRQ like an EXTI/NVIC interrupt path, except Linux splits the hard interrupt from deferred work and must share lines across drivers. **IRQ:** interrupt request, the signal path that tells the CPU or interrupt controller that hardware needs service.

112.7 Key Notes

- Stepper pulse generation, brushed-motor PWM and BLDC commutation have different timing contracts. Linux can supervise a system without owning every fast control loop.
- Current limiting, dead time, supply transients and thermal limits are electrical responsibilities that software commands do not replace.
- Open-loop step counts do not guarantee physical motion. Feedback and fault reporting are needed when missed motion has consequences.

- A stop command and a safety function are not the same design. Fault containment and emergency stopping need an independently justified hardware and system policy.

112.8 Pitfalls

> **Examples and production:** Do not burn fuses, enroll production keys, or sign release images while using these examples. Use throwaway keys and back up the unsigned image plus the key directory before testing irreversible security flows.

- **Stepper current too high.** DRV8825 / A4988 overheat and thermally shut down. Set Vref carefully. Add heat sink + airflow.
- **Stepper missed steps.** Acceleration too aggressive or current too low. Add longer ramp or increase current.
- **No back-EMF clamp diodes.** Switching off an inductive load (motor coil) generates voltage spikes that destroy MOSFETs. Drivers like BTS7960 have internal clamps. Discrete designs need flyback diodes.
- **PWM frequency in motor's audible range.** 1–10 kHz PWM makes motors whine. Bump to >20 kHz (ultrasonic) or use stealthChop (TMC).
- **H-bridge shoot-through.** Never enable both high and low side of the same leg simultaneously. Use a driver IC with dead-time, or software interlock.
- **Powering driver Vmot before logic Vcc.** Some drivers latch up. Check the datasheet sequence. BTS7960 is tolerant. A4988 is sensitive.
- **Insufficient bulk capacitance on Vmot.** Motor inrush sags the rail. Logic supply brown-outs. Add 470 µF+ low-ESR cap per driver.
- **EMI from motor brushes.** A DC brushed motor radiates broadband RF. Add ceramic caps (100 nF) across motor terminals + an LC filter on the supply line. WiFi and CAN buses near unfiltered motors fail intermittently.
- **TMC2209 UART CRC mismatch.** Easy to compute wrong. Use the official polynomial table or copy from the TMC library.
- **BLDC wrong phase order.** If the motor spins backward when you command forward, swap any two motor phases (e.g., A ↔ B). Don't rely on coil-color conventions, they're not standard.
- **No current limit on the supply.** A stalled motor draws stall current = Vbat / Rcoil. NEMA17 stalled: 12 V / 3 Ω = 4 A. Without a current-limited supply or a fuse, the wiring smokes.

112.9 Going deeper

- **TI DRV8825 datasheet + application notes**: the canonical stepper-driver tutorial.
- **Trinamic TMC2209 datasheet + datagram structure**: for UART config.
- **Infineon BTS7960 datasheet**: H-bridge integration.
- **TI DRV8302 datasheet**: BLDC gate driver.
- **SimpleFOC project (Arduino-based)**: readable FOC implementation. Runs on STM32 or ESP32.
- **ODrive project**: open-source BLDC controller for high-performance robotics.
- **Klipper firmware**: Linux + MCU split architecture for 3D printers. Great reference for “Linux as motion planner.”
- **Ch 48**: PWM kernel framework.
- **Ch 111**: encoder feedback (essential for closed-loop control).

> Next chapter: **Chapter 113: WS2812 / SK6812 / APA102 addressable LEDs**, Group S (Indicators & smart LEDs).

Chapter 113: WS2812 / SK6812 / APA102 addressable LEDs

What: the **addressable RGB LED strip**, daisy-chained programmable pixels where each pixel is RGB (WS2812) or RGBW (SK6812) or has independent global brightness (APA102). We cover the timing-critical 800 kHz one-wire protocol of WS2812 + SK6812, why bit-banging from Linux fails without PREEMPT_RT, the three production-quality implementations on the i.MX6ULL, (1) **SPI + DMA encoding** (the workhorse), (2) **PWM + DMA** (alternative for boards with spare PWM), (3) **bit-bang under PREEMPT_RT** (only viable for short strips). Then APA102's clean SPI native protocol that sidesteps all the timing pain. **MCU bridge:** Think of DMA like the MCU DMA controller you used for UART or SPI, but with cache coherency, scatter-gather descriptors, and kernel ownership rules added. **MCU bridge:** Think of Linux PWM like an MCU timer output channel, except the driver exposes period, duty cycle, polarity, and enable state through a subsystem. **DMA:** Direct Memory Access. Hardware moves data to or from memory without the CPU copying each byte. **PWM:** Pulse-Width Modulation, a timer output whose duty cycle controls average power or encodes timing. **PREEMPT_RT:** the Linux real-time patch set that makes more kernel paths preemptible and reduces latency.

Why: every modern indicator strip, status display, mood-lighting product, holiday string light, LED ring around a Wi-Fi-enabled doorbell, automotive interior accent, all use these three families. Hundreds of pixels per meter, $0.05 per pixel at volume, the only daisy-chained interface that scales to 1000+ LEDs on a single data line. Yet driving them from Linux is "interesting", the WS2812 wire protocol uses 350 ns and 800 ns pulses. Linux IRQ jitter is microseconds. Understanding the SPI + DMA trick lets you drive any-length strip at 30+ fps with < 5 % CPU. **IRQ:** interrupt request, the signal path that tells the CPU or interrupt controller that hardware needs service.

Focus: WS2812 and SK6812 encode bits as pulse widths on a single wire. A "0" bit is 350 ns high followed by 800 ns low. A "1" bit is 800 ns high followed by 450 ns low. A reset is 50 µs or more of low. Each LED takes 24 bits (3 channels × 8), or 32 bits for RGBW, and the whole strip is one back-to-back stream. Bit-banging from Linux user-space does not work. The IRQ jitter is microseconds. The WS2812 timing budget is hundreds of nanoseconds. The trick: each WS2812 "bit" becomes 4 SPI bits at 3.2 MHz. A "0" is encoded as `1000`, a "1" as `1110`. The SPI bitstream becomes the WS2812 waveform after DMA pushes it out hands-off. APA102 is the escape hatch, separate clock + data lines, no timing constraints.

Tooling. This chapter uses `libgpiod` (`gpioset`) for hello-world. The SPI-DMA path uses raw `/dev/spidev`. Optional `python3-spidev`.

- **Ubuntu-base (target):** `apt install gpiod libgpiod-dev python3-spidev`

- **Buildroot:** `BR2_PACKAGE_LIBGPIOD=y BR2_PACKAGE_PYTHON3_SPIDEV=y` **Buildroot:** a configuration-driven build system that produces a complete root filesystem and related images.
- Full per-tool reference: Userspace tooling appendix. **MCU bridge:** Think of the rootfs as the firmware image's file-backed runtime environment. On an MCU you link everything into flash. On Linux, programs and config live in this mounted tree. **rootfs:** root filesystem, the directory tree mounted at / that contains /bin, /etc, /dev, and libraries.

113.1 Chip comparison

	WS2812B	SK6812 (RGBW)	APA102	WS2811
Channels per pixel	RGB (3 × 8 bit)	RGBW (4 × 8 bit)	RGB + 5-bit global brightness	RGB (off-strip controller)
Protocol	one-wire, 800 kHz	one-wire, 800 kHz	SPI (clock + data)	one-wire, 800 kHz
Refresh rate (100 LEDs)	30 fps max	30 fps max	4500 fps max	30 fps
Strip current (white max)	60 mA/LED	60 mA/LED	60 mA/LED	60 mA/LED
Timing tolerance	±150 ns	±150 ns	clock-recovered (any)	±150 ns
Cost (per LED, bulk)	$0.05	$0.08	$0.15	$0.06
Power	5 V	5 V	5 V	12 V (controller) + 12 V LEDs
Reset gap	50 µs (some need 250 µs)	50 µs	n/a	50 µs

Pick guide:

- **WS2812B**: workhorse for any non-time-critical visual. Cheap, ubiquitous.

- **SK6812 RGBW**: when you need warm white (not RGB-mixed). The dedicated white LED is much truer than RGB-mixed white.
- **APA102**: when you need ultra-high refresh (POV displays, audio-reactive bars at kHz update). Also when you want to drive from a “dumb” 3.3 V SPI without level shifting tricks. More expensive.

113.2 WS2812 timing in detail

The datasheet specifies (per the chip, not the strip):

Bit	Description	T_HIGH	T_LOW
0	“0” code	0.4 µs ±150 ns	0.85 µs ±150 ns
1	“1” code	0.8 µs ±150 ns	0.45 µs ±150 ns
Reset	latch	0	≥50 µs

Each LED captures 24 bits of data (or 32 for RGBW), then forwards the rest down the chain. Data order: GRB (not RGB), the green byte first.

A 100-LED strip = 2400 bits × 1.25 µs/bit = 3 ms transfer + 50 µs reset → max ~328 fps theoretical, but the data line settling and chip latch reduce this. 30 fps is comfortable for most apps.

113.3 The SPI + DMA pattern, the standard approach in embedded Linux

Encode each WS2812 bit as 4 SPI bits at 3.2 MHz (312.5 ns per SPI bit):

- WS2812 “0” = SPI `1000` → 312.5 ns high + 937.5 ns low ≈ within spec
- WS2812 “1” = SPI `1110` → 937.5 ns high + 312.5 ns low ≈ within spec

So each WS2812 byte (8 bits) becomes 4 SPI bytes (32 bits). For 100 LEDs × 3 bytes = 300 WS2812 bytes → 1200 SPI bytes. At 3.2 MHz SPI = 3 ms transfer. DMA pushes it autonomously. CPU does the encoding once per frame and starts the SPI transfer.

```
/* Encode one WS2812 byte (8 bits) into 4 SPI bytes (32 bits). */
static void encode_byte(uint8_t in, uint8_t out[4]) {
    static const uint8_t bit_lut[4] = { 0x88, 0x8E, 0xE8, 0xEE };
    out[0] = bit_lut[(in >> 6) & 3];
    out[1] = bit_lut[(in >> 4) & 3];
```

```
    out[2] = bit_lut[(in >> 2) & 3];
    out[3] = bit_lut[(in >> 0) & 3];
}
```

This encodes 2 WS2812 bits at a time → 1 SPI byte. Each WS2812 bit is 4 SPI bits. Two WS2812 bits = 8 SPI bits = 1 SPI byte. The LUT pre-computes all 4 combinations:

- `00` → `1000 1000` → 0x88
- `01` → `1000 1110` → 0x8E
- `10` → `1110 1000` → 0xE8
- `11` → `1110 1110` → 0xEE

Full encoder for a 100-LED strip:

```
#define N_LEDS 100
uint8_t pixels[N_LEDS][3];          /* G, R, B */
uint8_t spi_buf[N_LEDS * 3 * 4];    /* 4× expansion */

static void encode_strip(void) {
    for (int i = 0; i < N_LEDS; i++) {
        encode_byte(pixels[i][0], &spi_buf[i*12 + 0]);   /* G */
        encode_byte(pixels[i][1], &spi_buf[i*12 + 4]);   /* R */
        encode_byte(pixels[i][2], &spi_buf[i*12 + 8]);   /* B */
    }
}

static void send_strip(int spi_fd) {
    encode_strip();
    struct spi_ioc_transfer t = {
        .tx_buf = (unsigned long)spi_buf,
        .len = sizeof spi_buf,
        .speed_hz = 3200000,
    };
    ioctl(spi_fd, SPI_IOC_MESSAGE(1), &t);
    /* The trailing low time of the last bit is naturally followed by SPI idle
       (line stays low after CS rises), giving the >50 µs reset for free. */
}
```

Wire MOSI to the strip's DIN. Always add a 74AHCT125 buffer to convert 3.3 V MOSI to 5 V. The WS2812 datasheet specifies VIH = 0.7 × VDD = 3.5 V. 3.3 V is below that. Some strip batches accept it. Many do not. Buffer every time.

Now you have 30+ fps RGB animations on a strip with ~3 % CPU. Add animation logic (rainbow shift, fade, audio-reactive) on top.

113.4 Alternative, PWM + DMA encoding

Some i.MX6ULL board designs free SPI for other devices. Use PWM with DMA:

```
PWM period = 1.25 µs (800 kHz)
DMA writes new duty-cycle on every PWM cycle:
  "0" bit → duty = 32 % (T_HIGH = 0.4 µs)
  "1" bit → duty = 64 % (T_HIGH = 0.8 µs)
```

Linux PWM doesn't expose dynamic per-cycle duty changes via sysfs. This needs a kernel-side DMA-fed PWM driver. The mainline `pwm-imx27.c` doesn't support this. Vendor BSPs sometimes do. Effort/reward: lower than SPI approach for most uses. SPI wins.

> **sysfs:** a kernel-generated filesystem under /sys that exposes devices, drivers, and attributes.

113.5 PREEMPT_RT bit-bang, for short strips only

With PREEMPT_RT (Ch 52A), a high-priority RT thread can busy-loop on `clock_gettime(CLOCK_MONOTONIC)` and toggle a GPIO with ~150 ns jitter. Just barely within WS2812 spec.

> **MCU bridge:** Think of Linux GPIO like the same pin set/reset block you used on STM32, but accessed through a kernel subsystem that owns numbering, direction, interrupts, and user-space exposure. **GPIO:** General-Purpose Input/Output, a pin controlled as a digital input, output, or interrupt source.

```
struct sched_param sp = { .sched_priority = 90 };
sched_setscheduler(0, SCHED_FIFO, &sp);
mlockall(MCL_CURRENT | MCL_FUTURE);

/* Pin to one CPU */
cpu_set_t mask; CPU_ZERO(&mask); CPU_SET(0, &mask);
sched_setaffinity(0, sizeof mask, &mask);

for (int led = 0; led < N_LEDS; led++) {
    for (int bit = 23; bit >= 0; bit--) {
        if (pixel[led] & (1 << bit)) {
            gpio_high(); busy_wait_ns(800);
            gpio_low();  busy_wait_ns(450);
        } else {
            gpio_high(); busy_wait_ns(400);
            gpio_low();  busy_wait_ns(850);
        }
    }
}
```

`busy_wait_ns` is a tight `clock_gettime` loop. Works for 10–30 LEDs reliably. Fails above ~50 (a single interrupt or migration jitter corrupts a bit, all subsequent LEDs latch the wrong data).

Do not ship this approach. It is useful as a demo only. Production designs should use SPI + DMA.

113.6 APA102, the timing-painless alternative

APA102 has two wires: CLOCK + DATA, normal SPI. No timing constraints (any clock rate. APA102 syncs to your clock). Wire to SPI. Bit-bang or use the kernel SPI driver, all work.

Protocol per frame:

```
   Start frame:  32 bits of zero
   For each LED: 1 byte (111 + 5-bit global brightness 0..31), then BGR (24 bits)
   End frame:    32+ bits of ones (or zeros – both work)
```

Code:

```
uint8_t buf[4 + 4*N_LEDS + 4];
memset(&buf[0], 0, 4);                                  /* start */
for (int i = 0; i < N_LEDS; i++) {
    buf[4 + i*4 + 0] = 0xE0 | (brightness & 0x1F);   /* 5-bit brightness */
    buf[4 + i*4 + 1] = pixel[i].b;
    buf[4 + i*4 + 2] = pixel[i].g;
    buf[4 + i*4 + 3] = pixel[i].r;
}
memset(&buf[sizeof buf - 4], 0xFF, 4);                 /* end */
spi_send(buf, sizeof buf);
```

That’s it. No 4× encoding, no level-shift requirement (APA102 accepts 3.3 V logic directly), no reset-gap concern. The 5-bit global brightness is a separate PWM domain at ~700 Hz, useful for low-light, but it flickers in camera footage (frame-rate beating against the 700 Hz). For studio use, disable global PWM (set brightness 0x1F, control color via the 8-bit channels).

113.7 Color and gamma

Naïvely lerping RGB doesn’t look right. The eye perceives brightness logarithmically. A “50 % bright red” set to `(128, 0, 0)` looks far too bright. Apply gamma 2.2:

```
static const uint8_t GAMMA8[256] = { /* precomputed table */ };
uint8_t corrected = GAMMA8[raw];
```

Generate the table:

```
[int((i/255.0)**2.2 * 255 + 0.5) for i in range(256)]
```

For mood-lighting or status indicators, always apply gamma. For data visualization (e.g., “32 % CPU load”), maybe not.

For RGB→HSV color manipulation:

- Hue rotates around the rainbow.
- Saturation = how vivid.
- Value (brightness) = how bright.

Animations (rainbow shift, breathing) become 5-line `for` loops over HSV.

113.8 Power budgeting, the elephant in the room

> **Examples and production:** Do not burn fuses, enroll production keys, or sign release images while using these examples. Use throwaway keys and back up the unsigned image plus the key directory before testing irreversible security flows.

Each LED at full white = 60 mA. 100 LEDs at full white = 6 A. A USB 5 V supply (1 A) can’t even light 17 LEDs. Real numbers:

Strip length	Max current at white	Required PSU
30	1.8 A	5 V × 2 A
100	6 A	5 V × 8 A
300	18 A	5 V × 20 A
1000	60 A	5 V × 70 A (specialty supply)

For long strips:

- Inject power every 50–100 LEDs (extra 5 V + GND from a beefier supply).
- Voltage drop along the strip turns “white” at the end into “yellow” (red survives the drop, blue dies first).
- Use thick wire (14 AWG for 10+ A).
- Add a fuse, a shorted addressable strip can dump 20 A through a thin wire and start a fire.

For mood lighting, cap brightness at 64/255 (~25 %), visually plenty, current draw 1/4. Software-enforced.

113.9 Key Notes

- Pulse-width-coded LEDs need a continuous waveform with the selected part’s timing. Encoding into a hardware-driven stream moves that responsibility away from scheduled bit toggles.
- Frame length, transfer continuity and reset intervals constrain the update path. DMA support alone does not prove that a driver preserves the waveform across its buffers.
- Color order, brightness mapping and visible output are different layers. A correct electrical frame can still display the wrong color or brightness.
- Power distribution, level compatibility and fault protection must scale with the strip. A fixed brightness limit is not a substitute for a qualified supply and wiring design.

113.10 Pitfalls

- **3.3 V → 5 V level shifting.** WS2812 datasheet says VIH = 0.7 × VDD = 3.5 V. A 3.3 V GPIO is *below* this. Most strips work anyway but some batches don’t. Always use a 74AHCT125 buffer for reliability.
- **First LED bright random color on power-up.** Strip powers up before MCU sends data → LEDs latch garbage. Send a frame of all-zero immediately on boot.
- **Reset gap too short.** Some clone WS2812 chips need 250 µs reset, not 50 µs. If frames intermittently corrupt mid-strip, lengthen the reset.
- **Power injection without ground tie.** Inject +5 V but forget to tie the injected supply’s GND to the strip’s GND. Now the strip floats, signals to misbehave. Always tie all GNDs.
- **Voltage droop on long strips.** Red survives, blue dies. Inject power every 100 LEDs or shorten the strip.
- **DMA SPI buffer too large.** 1000-LED strip × 4× = 12 KB DMA buffer. Some SPI drivers cap at 4 KB and silently truncate. Test full strip. Verify all LEDs animated.
- **Wrong color order.** WS2812 is GRB (Green-Red-Blue), not RGB. WS2812B is GRB. SK6812 RGBW is GRBW. APA102 is BGR. Easy to swap. “red doesn’t work” is the symptom.
- **Capacitor between V+ and GND missing.** Add 1000 µF + 100 nF at the strip’s start. The inrush spike at “all-white” stresses your PSU and causes voltage droop.
- **Fuse missing.** A shorted strip dumps PSU’s max current, start a fire. 10 A fuse for 5 m of WS2812 minimum.

- **Quality varies wildly between batches.** Cheap strips have miscalibrated chips. The same code looks subtly different (pinker reds, dim blues) on different strips. Production: buy from one reliable supplier and stick.
- **APA102 5-bit brightness PWM beats with camera frame rate.** For video shoots, max global brightness. Control via 8-bit channels.

113.11 Going deeper

- **WS2812B datasheet** (~5 pages, full timing).
- **APA102 datasheet**: clean spec. No timing pain.
- **Adafruit NeoPixel uberguide**: bible of WS2812 power, wiring, troubleshooting.
- **Tim Bauwens' (xy_pi) Linux WS2812 driver patches**: for understanding kernel-side approaches.
- **FastLED library (Arduino)**: readable C++ for color manipulation, gamma, HSV.
- **`drivers/leds/leds-pca9685.c`**: for 16-channel PWM I²C alternatives.
- **rPi DMA-WS2812 implementations (`rpi_ws281x`)**: the canonical reference (different SoC but same principles).
- **Ch 78** (MEMS mic) for audio-reactive applications.
- **Ch 52A** (PREEMPT_RT) for short-strip bit-bang viability.

> Next chapter: **Chapter 114: Beepers, relays, SSRs**, the everyday actuators.

Chapter 114: Beepers, relays, SSRs

What: the **discrete actuators** that fall outside the main subsystems but appear in every product: **passive piezo buzzers** (need PWM to make sound), **active buzzers** (fixed-frequency, GPIO on/off), **mechanical relays** (5 V or 12 V coils driving 240 V AC contacts), **MOSFETs** (DC switching, fast, no contact wear), and **SSRs (Solid State Relays)** (AC switching, opto-isolated, zero-cross, the production-grade choice for mains-load control). On the i.MX6ULL we drive each with the matching kernel framework (PWM for passive, GPIO for the rest), wire the protection circuits (flyback diodes, snubbers, isolation), and build a 4-channel home automation relay board controlled via MQTT. **MCU bridge:** Think of Linux PWM like an MCU timer output channel, except the driver exposes period, duty cycle, polarity, and enable state through a subsystem. **MCU bridge:** Think of Linux GPIO like the same pin set/reset block you used on STM32, but accessed through a kernel subsystem that owns numbering, direction, interrupts, and user-space exposure. **PWM:** Pulse-Width Modulation, a timer output whose duty cycle controls average power or encodes timing. **GPIO:** General-Purpose Input/Output, a pin controlled as a digital input, output, or interrupt source.

Why: real products take physical actions: beep on user input, switch a pump, turn on a heater, drive a solenoid valve, ring a bell. Each actuator has different electrical requirements and different safety pitfalls. Get them wrong and you damage the driver, miss a debounce, energize an AC load while the relay’s contact is half-open (arcing → contact welding → can’t turn off → fire). This chapter is short but covers the engineering details that separate a demo from a five-year shipping product.

Focus: three non-negotiable rules for the actuators in this chapter. Inductive loads (relays, solenoids, motors) need a flyback diode. AC loads need isolation. Zero-cross AC switching needs a zero-cross SSR. Non-zero-cross switching arcs, generates harmonics, and burns contacts. The Linux side is trivial, sysfs PWM or GPIO. The electrical-engineering side is most of the work. **sysfs:** a kernel-generated filesystem under /sys that exposes devices, drivers, and attributes.

114.1 Buzzers, passive vs active

	Passive piezo	Active buzzer
Construction	bare piezo disc	piezo + oscillator IC
Drive	needs AC signal (PWM)	GPIO on/off (it self-oscillates)

	Passive piezo	Active buzzer
Frequency	you choose (typ. 2–4 kHz)	fixed (typ. 2.7 kHz)
Volume control	yes (PWM duty)	no
Cost	$0.15	$0.30
Use	melodies, tones, frequency-encoded info	“beep” indicators

Passive buzzer

```
   i.MX PWM ──── 1 kΩ ──── Gate of small MOSFET (e.g. 2N7000)
                                  │
                              Drain ── one terminal of piezo
   Other piezo terminal ──────── 5 V

   Or simpler: PWM pin direct to piezo (works if piezo low-capacitance + low current)
```

Drive a 2 kHz square wave at 50 % duty:

```
echo 0 > /sys/class/pwm/pwmchip0/export
echo 500000 > /sys/class/pwm/pwmchip0/pwm0/period      # 500 µs = 2 kHz
echo 250000 > /sys/class/pwm/pwmchip0/pwm0/duty_cycle  # 50 %
echo 1 > /sys/class/pwm/pwmchip0/pwm0/enable

sleep 0.2   # beep
echo 0 > /sys/class/pwm/pwmchip0/pwm0/enable
```

Volume = duty cycle (max at 50 %. Both 0 % and 100 % = silence). Pitch = frequency.

Play a melody: change the period over time. A “Mario” tone sequence is just a list of `(freq, duration)` tuples.

Active buzzer

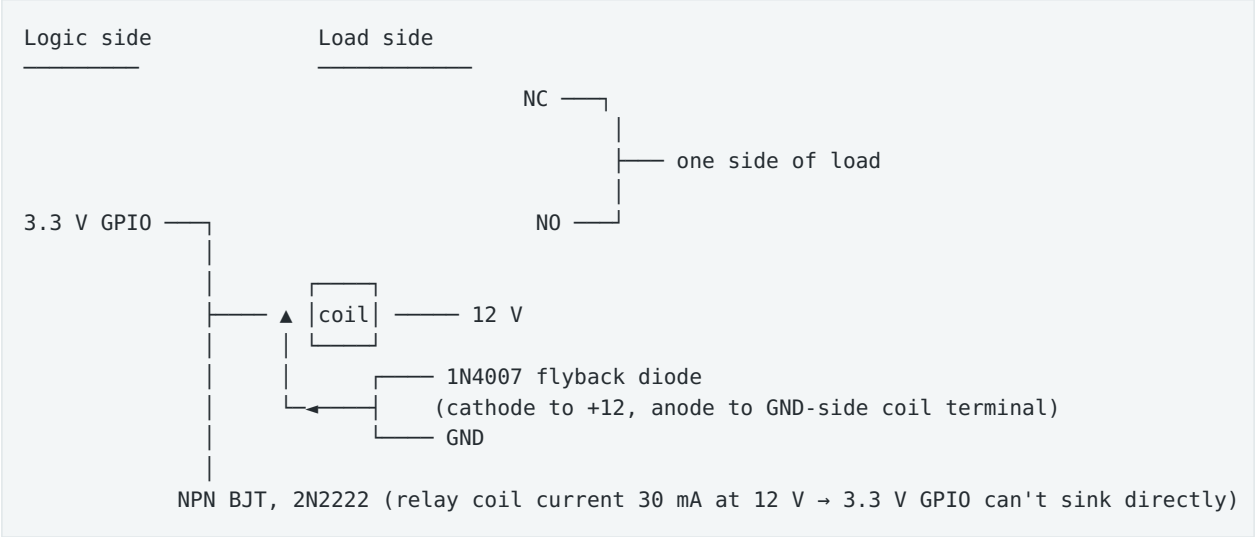
Simpler:

```
gpio_write(BUZZER, 1); usleep(100000);
gpio_write(BUZZER, 0);
```

No PWM, no MOSFET (most active buzzers tolerate direct 3.3/5 V GPIO drive at < 30 mA). Pitch is whatever the buzzer’s internal oscillator gives.

114.2 Mechanical relays, the basics

A relay = electromagnetically actuated switch. Coil (5/12/24 V DC) energizes → moves armature → switches contacts.



The flyback diode is mandatory. When the GPIO drops and the BJT turns off, the relay coil's inductance generates a back-EMF spike (hundreds of volts). Without the diode, the spike kills the BJT, propagates back, damages the SoC. The diode shorts the spike harmlessly.

Linux side:

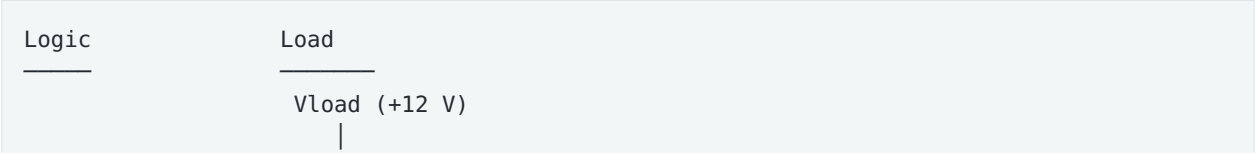
```
gpioset gpiochip4 22=1   # relay on
gpioset gpiochip4 22=0   # relay off
```

Mechanical relay characteristics:

- **Switching time**: ~10 ms on, ~5 ms off.
- **Contact rating**: typically 10 A AC at 250 V (per relay datasheet).
- **Life**: ~10^5 switches at rated load. 10^7 at much-reduced load.
- **Noise**: audible click. Some applications (libraries, hospitals) prefer SSRs to avoid noise.

114.3 MOSFET for DC loads, fast, silent, infinite life

For DC loads (LEDs, fans, small heaters, solenoid valves), use a logic-level N-FET (IRLZ44N, AOD508, IRF3205) instead of a relay:



```
                     Load
                      │
GPIO ─── 100 Ω ──┤ Gate
                 ├ Drain
                 ┤ Source
                      │
                     GND
                      │
                 10 kΩ pull-down on gate
                 (so the gate doesn't float when GPIO is tri-stated during boot)
```

PWM the gate to dim a load (LEDs, motor speed):

```
# 25 kHz PWM (above audio range) at 40 % duty
echo 40000 > /sys/class/pwm/pwmchip0/pwm0/period
echo 16000 > /sys/class/pwm/pwmchip0/pwm0/duty_cycle
```

For inductive DC loads (DC motor, solenoid): add a flyback diode (Schottky for fast loads) across the load.

114.4 Solid-state relays (SSRs) for AC loads, the production choice

For mains AC loads (heaters, lamps, pumps, fans on AC):

```
Logic side                    Load side
──────────                    ──────────
GPIO 3.3 V ─── 330 Ω ─── ┌──┴── Opto ──┐── Triac/SCR ─── L (Live)
                         │             │
                         │             │       ┌── snubber R+C
GND ─────────────────────└─────────────┘       │   across triac
                                               │
                          Load (heater, lamp, etc.)   N (Neutral)
```

A typical 25 A SSR module (Fotek SSR-25DA, Crydom A2425) handles 240 VAC × 25 A. Key features:

- **Optical isolation** between logic and load (typically 4 kV).
- **Zero-cross switching**: triac fires only at the AC zero-crossing. Eliminates inrush and harmonics. Mandatory for resistive (heater) loads. Rough for motor loads (the motor may lag).
- **No moving parts**: silent, fast, infinite life if not abused.
- **Always-on leakage**: even when “off,” a few mA leaks through. Don’t rely on the SSR to make a load *electrically dead* for service work. Use a contactor or pull the plug.

Linux side: just GPIO toggle, same as mechanical relay. The SSR's internal opto + zero-cross logic handles the rest.

114.5 AC safety, non-negotiable rules

> **Examples and production:** Do not burn fuses, enroll production keys, or sign release images while using these examples. Use throwaway keys and back up the unsigned image plus the key directory before testing irreversible security flows.

Live AC kills. Working with mains:

1. **Isolation 4 kV minimum** between logic and mains side. Module SSRs deliver this. Bench-built circuits often don't.
2. **Fuses on the AC side.** A shorted load (motor stalls, heater coil melts) without a fuse will burn wiring or weld SSR contacts.
3. **Proper wire gauge.** 16 AWG minimum for 10 A circuits. 14 AWG for 15 A. Solid copper, properly crimped to terminals.
4. **Insulated enclosure** with no exposed mains-side conductors when assembled. Use commercial enclosures with strain reliefs.
5. **Earth bonding.** Any metal enclosure must be earthed. A pre-failure short-to-chassis trips the earth-leakage breaker instead of electrocuting the user.
6. **GFCI / RCD upstream.** Your distribution panel should have residual-current protection. Don't rely on it as primary safety, but it saves lives on partial failures.
7. **No live work.** Never wire or troubleshoot a powered-up circuit. Pull the plug, even for "quick checks."
8. **Voltmeter before touching.** After unplugging, verify no voltage. Capacitors hold charge.
9. **Certification for products.** UL (US), CE (EU), CCC (China) require safety testing of any mains-load product before sale.

This chapter cannot replace a proper electrical-safety class. Get a qualified electrician to review your design if you are not one.

114.6 Worked example, 4-channel home-automation relay board

Hardware:

- i.MX6ULL on a DIN-rail enclosure

- 4× Songle SRD-05VDC-SL-C relays driving 240 VAC outputs through screw terminals
- 4× 2N2222 BJT + 1N4007 flyback for coil drive
- 4× LEDs showing each channel state
- Input: PIR motion sensor on a GPIO (Ch 67-style)

Software:

- A Python daemon listening on MQTT `home/relay/N/set`
- States published on `home/relay/N/state`
- Home Assistant subscribes via MQTT integration → 4 toggle switches in the dashboard

```
import paho.mqtt.client as mqtt
import gpiod

chip = gpiod.Chip('gpiochip0')
relays = [chip.get_line(p) for p in [20, 21, 22, 23]]
for r in relays:
    r.request(consumer='relay', type=gpiod.LINE_REQ_DIR_OUT, default_vals=[0])

state = [0, 0, 0, 0]

def on_message(c, userdata, msg):
    parts = msg.topic.split('/')
    if len(parts) >= 4 and parts[3] == 'set':
        n = int(parts[2])
        val = 1 if msg.payload.decode().lower() in ('on', '1', 'true') else 0
        relays[n].set_value(val)
        state[n] = val
        c.publish(f'home/relay/{n}/state', 'ON' if val else 'OFF', retain=True)

c = mqtt.Client()
c.on_connect = lambda c, u, f, rc: c.subscribe('home/relay/+/set')
c.on_message = on_message
c.connect('localhost')
c.loop_forever()
```

About 40 lines. Install as a systemd unit. Open Home Assistant on a phone, tap “Living Room Lamp”, and the relay clicks the lamp on via MQTT.

114.7 Key Notes

- Buzzer, relay, MOSFET and SSR interfaces differ in load behavior and protection requirements. A GPIO can request a state without safely supplying the load.

- Reset and startup levels should leave actuators in a defined safe state. Software initialization occurs after a period in which pins may have another function or be high impedance.
- Inductive energy, inrush, dissipation and leakage constrain switching-device selection. An off command does not necessarily provide electrical isolation.
- Mains switching requires a qualified electrical design and enclosure. The examples are not instructions to build or experiment with exposed mains circuitry.

114.8 Pitfalls

- **No flyback diode on relay coil.** Repeated back-EMF spikes will eventually damage the BJT and SoC.
- **GPIO direct-driving a relay coil.** Coil draws 30 mA at 5 V (= 150 mW). GPIO typically tolerates 20 mA max. Burn-out symptom: GPIO works once, then never again.
- **Cheap SSR with rated current.** Fotek SSR-40DA rated “40 A”, actually good for ~25 A continuous and only with a real heatsink. De-rate aggressively.
- **Zero-cross SSR with inductive load.** Motors lag. Zero-cross switching at voltage-zero is at current-peak for inductive load → contact stress. For inductive loads use random-fire SSR.
- **Buzzer at the right pitch for piezo resonance.** Most piezos have a sharp resonant peak around 2.7 kHz. Driving off-resonance gets you 10 dB less SPL. Find the peak with a sweep.

> **MCU bridge:** Think of SPL like the tiny early startup code that runs from internal SRAM before DDR is usable. **SPL:** Secondary Program Loader, a tiny first U-Boot stage that fits in OCRAM and initializes DDR.

- **PWM frequency in audio range.** A motor PWM at 1 kHz whines audibly. Bump to 20 kHz+.
- **No GPIO pull-down on MOSFET gate.** During boot, the GPIO is high-impedance for a few seconds. The load floats and may spuriously partially-on. 10 kΩ pull-down ensures off-on-boot.
- **AC neutral switching.** Always switch the LIVE wire, never the NEUTRAL. Switching neutral leaves the load energized when “off” → shock hazard.
- **SSR on the AC neutral.** Same problem. SSR must be on live.

- **No fuses.** A failed driver shorts the load. Without a fuse, the wiring becomes the fuse.
- **Inadequate creepage / clearance.** PCB tracks carrying mains must be ≥4 mm apart with no solder bridges. Use a real PCB house with mains-safety design rules.
- **Treating an off SSR as electrically isolated.** SSR leakage is 1–5 mA. Enough to make an LED glow faintly, or, for service, to give a small shock. Pull the plug for service.

114.9 Going deeper

- **Songle SRD relay datasheet**: most common 5 V coil mechanical relay.
- **Crydom and Omron SSR catalogs**: quality vs Fotek.
- **IRLZ44N / AOD508 datasheets**: logic-level MOSFETs.
- **IPC-2221**: PCB design standards for creepage and clearance for mains.
- **UL 508A**: industrial control panel safety standard.
- **`pwm-fan` driver in mainline**: PWM-controlled fan with tach feedback, a related actuator pattern.
- **Home Assistant MQTT integration docs**: for the dashboard side.
- **Ch 48**: PWM kernel framework.
- **Ch 51A**: watchdog (essential for safety-critical relay control).

> Next chapter: **Chapter 115: Dual FEC + hosted Ethernet**, networking with both i.MX6ULL FECs plus W5500 / ENC28J60 SPI Ethernet.

Chapter 115: Dual FEC + hosted Ethernet

What: putting **multiple Ethernet interfaces** on an i.MX6ULL. The SoC has two on-chip **FEC (Fast Ethernet Controller)** instances at 100 Mbps. We bring both up simultaneously as `eth0` and `eth1` with separate PHYs. For a third (or fourth) interface, we add **WIZnet W5500** (SPI Ethernet with hardware TCP/IP) or **Microchip ENC28J60** (older, slower, mainline-supported) as `spi-ethernet` chips. We then build typical multi-NIC scenarios: a **router** (eth0 WAN, eth1 LAN), a **bridge** (eth0+eth1 transparent L2), and an **isolated subnet** for industrial bus traffic (eth1 talks only to PLCs, eth0 to corporate network).

Why: any “edge gateway” product has 2+ Ethernet ports, one to the internet, one to a local industrial network, because mixing the two on a single physical port creates security and reliability problems. The i.MX6ULL is unusual in having two on-chip FECs (most SoCs in this class have one). This is a feature you should exploit. When you need a third interface (e.g., a separate management LAN, or a Modbus-TCP island), SPI Ethernet chips are the only way without an external switch, and they’re easy on Linux thanks to mainline drivers.

Focus: Dual-MAC on one SoC means two PHYs. Each needs its own pinmux, reference clock, and IRQ. The kernel netdev model already isolates the two MACs, they look like two NICs. The hard part is the bring-up: pinmux conflicts (FEC1 shares many pins with FEC2 + UART), separate PHY addresses on MDIO, and per-PHY interrupt routing. The W5500 implements TCP/IP in hardware. You talk to it at the socket layer over SPI, not as a netdev. This does not fit the Linux netdev model. Mainline-friendly choices are ENC28J60 (slow, netdev-presenting) and TI’s KSZ8851 / Davicom DM9051 (10/100, netdev, faster). We cover all three patterns. **MCU bridge:** Think of an IRQ like an EXTI/NVIC interrupt path, except Linux splits the hard interrupt from deferred work and must share lines across drivers. **MAC:** Media Access Control in networking and radio chapters. It is the layer that owns framing and medium access. **PHY:** physical-layer block or chip that converts digital MAC signals to electrical or radio signals. **IRQ:** interrupt request, the signal path that tells the CPU or interrupt controller that hardware needs service.

Tooling. This chapter uses `iproute2`, `iperf3`, `tcpdump`, `ethtool`, optional `bridge-utils`.

- **Ubuntu-base (target):** `apt install iproute2 iperf3 tcpdump ethtool bridge-utils`
- **Buildroot:** `BR2_PACKAGE_IPROUTE2=y BR2_PACKAGE_IPERF3=y BR2_PACKAGE_TCPDUMP=y BR2_PACKAGE_ETHTOOL=y BR2_PACKAGE_BRIDGE_UTILS=y` **Buildroot:** a configuration-driven build system that produces a complete root filesystem and related images.

> - Full per-tool reference: Userspace tooling appendix. **MCU bridge:** Think of the rootfs as the firmware image's file-backed runtime environment. On an MCU you link everything into flash. On Linux, programs and config live in this mounted tree. **rootfs:** root filesystem, the directory tree mounted at / that contains /bin, /etc, /dev, and libraries.

115.1 i.MX6ULL FEC overview

The i.MX6ULL has **2× FEC** (FEC1 and FEC2 in the reference manual), each:

- 10/100 Mbps MAC
- RMII PHY interface (4 wires for data + 1 clock vs 8+ for MII. Cheaper, fewer pins)
- IEEE 1588 PTP timestamp support
- Separate DMA channel

> **MCU bridge:** Think of DMA like the MCU DMA controller you used for UART or SPI, but with cache coherency, scatter-gather descriptors, and kernel ownership rules added. **DMA:** Direct Memory Access. Hardware moves data to or from memory without the CPU copying each byte.

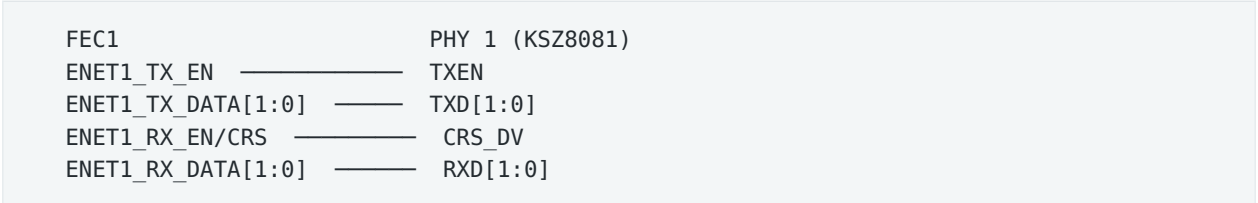
- Separate IRQ

Each FEC needs:

- An external PHY chip (KSZ8081, LAN8720A, RTL8201F are common).
- A 50 MHz clock to the RMII interface (either from the PHY's own crystal or from the SoC).
- Pin-muxed RGMII/RMII signals.
- MDIO (a 2-wire management bus shared between both PHYs on most designs).

115.2 Wiring two PHYs

The reference Point Atom MINI board has only FEC1 wired (single Ethernet). To bring up FEC2 you must either choose a board with both routed (some i.MX6ULL devkits have it), or wire FEC2 to a second PHY breakout. Pinout (RMII):



```
   ENET1_RX_ER  ──────────────  RXER
   ENET1_TX_CLK (RMII REF)  ──  REF_CLK   (50 MHz; can be SoC-out or PHY-out)
   ENET1_MDIO  ──────────────  MDIO
   ENET1_MDC   ──────────────  MDC

   FEC2                       PHY 2 (KSZ8081, different MDIO address)
   ENET2_*   ────────────────  same as above, separate pins
```

Each PHY has an MDIO address set by strap pins (typically 0, 1, 2, ...). Put FEC1's PHY at address 0 and FEC2's PHY at address 1. Both PHYs can share the same MDIO bus. The FEC reads the address that matches its `phy-handle` in the DT.

DT for both FECs (excerpt from `arch/arm/boot/dts/nxp/imx/imx6ull-myboard.dts`):

```
&fec1 {
    pinctrl-names = "default";
    pinctrl-0 = <&pinctrl_enet1>;
    phy-mode = "rmii";
    phy-handle = <&ethphy0>;
    status = "okay";

    mdio {
        #address-cells = <1>;
        #size-cells = <0>;

        ethphy0: ethernet-phy@0 {
            reg = <0>;
            micrel,led-mode = <1>;
            clocks = <&clks IMX6UL_CLK_ENET_REF>;
            clock-names = "rmii-ref";
        };

        ethphy1: ethernet-phy@1 {
            reg = <1>;
            micrel,led-mode = <1>;
            clocks = <&clks IMX6UL_CLK_ENET2_REF>;
            clock-names = "rmii-ref";
        };
    };
};

&fec2 {
    pinctrl-names = "default";
    pinctrl-0 = <&pinctrl_enet2>;
    phy-mode = "rmii";
    phy-handle = <&ethphy1>;
    status = "okay";
};
```

The MDIO node is under FEC1 (one MDIO instance), with both PHYs as children. FEC2 references `ethphy1` via `phy-handle`.

After boot:

```
ip link
# 2: eth0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 ... ether AA:BB:CC:00:00:01
# 3: eth1: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 ... ether AA:BB:CC:00:00:02
```

`ethtool eth0` and `ethtool eth1` should show “Link detected: yes” when cables are plugged.

115.3 Router pattern, WAN on eth0, LAN on eth1

```
# Bring up
ip addr add 192.168.1.1/24 dev eth1
ip link set eth1 up

dhclient eth0          # WAN gets its IP from upstream

# Enable IP forwarding
sysctl -w net.ipv4.ip_forward=1

# NAT (masquerade LAN behind WAN)
iptables -t nat -A POSTROUTING -o eth0 -j MASQUERADE
iptables -A FORWARD -i eth1 -o eth0 -j ACCEPT
iptables -A FORWARD -i eth0 -o eth1 -m state --state RELATED,ESTABLISHED -j ACCEPT

# DHCP server for LAN
apt install dnsmasq
cat >> /etc/dnsmasq.conf <<EOF
interface=eth1
dhcp-range=192.168.1.50,192.168.1.150,12h
EOF
systemctl restart dnsmasq
```

That is the full router setup. Add `nftables` for modern filtering, or stick with `iptables` for familiarity.

115.4 Bridge pattern, transparent L2 between eth0 + eth1

For when you want the i.MX6ULL to be invisible to upstream traffic but still observe / log it:

```
ip link add br0 type bridge
ip link set eth0 master br0
ip link set eth1 master br0
ip link set br0 up
ip addr add 192.168.1.5/24 dev br0    # optional, for management
```

Traffic flows transparently between eth0 and eth1. The i.MX6ULL can sniff via `tcpdump -i br0`, optionally filter / mirror via `tc` qdiscs. Useful for inline industrial-traffic monitoring.

115.5 Isolated subnet pattern, eth1 for industrial bus

For security + reliability: eth0 is the corporate network (with internet). Eth1 is a dedicated subnet to PLCs / sensors. No bridge, no routing, packets do not cross.

```
ip addr add 192.168.2.1/24 dev eth1     # private industrial subnet
ip link set eth1 up
# No forwarding; eth1 traffic stays on eth1

# Your application reads modbus from 192.168.2.x devices via eth1
# Your application reports to cloud via eth0
```

Compromise of corporate side doesn't expose PLCs. Compromise of PLC subnet doesn't pivot to cloud. This is the industrial-IoT best practice and a major reason for needing dual-NIC.

115.6 Adding a third NIC via SPI, the W5500 (hardware TCP/IP)

> **Driver choice:** Use the in-tree, maintained driver first. Use out-of-tree, spidev, or custom-driver paths only after you accept the kernel-version maintenance cost and document who owns updates.

W5500 is unusual: it's not a netdev. It's a **hardware TCP/IP stack** with 8 sockets, accessed over SPI. You don't talk Ethernet frames to it, you `OPEN`, `CONNECT`, `SEND`, `RECV` at the TCP/UDP level.

This makes W5500 awkward to use on Linux. The mainline kernel does not have a W5500 driver in `drivers/net/`. There are out-of-tree drivers that wrap W5500 sockets as Linux `AF_INET` sockets, but they're niche.

The practical Linux approach: use W5500's sockets directly from user-space via SPI, or choose a different chip (DM9051, ENC28J60) that presents as a netdev.

115.7 Adding a third NIC via SPI, ENC28J60 / DM9051 (mainline netdev)

For a third netdev, use ENC28J60 (slow but ubiquitous) or DM9051 (faster, both mainline). DM9051 is 10/100. ENC28J60 is 10 Mbps only.

DT for DM9051:

```
&ecspi3 {
    cs-gpios = <&gpio4 26 GPIO_ACTIVE_LOW>;
    status = "okay";
```

```
    eth: ethernet@0 {
        compatible = "davicom,dm9051";
        reg = <0>;
        spi-max-frequency = <20000000>;
        interrupts-extended = <&gpio4 27 IRQ_TYPE_LEVEL_LOW>;
        reset-gpios = <&gpio4 28 GPIO_ACTIVE_LOW>;
        local-mac-address = [DE AD BE EF 00 03];
    };
};
```

After `modprobe dm9051`:

```
ip link
# 4: eth2: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 ... ether DE:AD:BE:EF:00:03
```

You now have eth0 (FEC1), eth1 (FEC2), eth2 (DM9051 SPI). Three independent Ethernet interfaces.

Throughput limits:

- FEC1/FEC2: ~95 Mbps each (line rate at 100 Mbps).
- DM9051 SPI at 20 MHz: ~8 Mbps (SPI overhead dominates).
- ENC28J60 SPI at 20 MHz: ~3 Mbps.

DM9051 is adequate for a management interface or a low-rate Modbus-TCP island. It is not suitable for primary high-bandwidth traffic.

115.8 How the FEC driver works

`drivers/net/ethernet/freescale/fec_main.c`, the mainline FEC driver, ~3500 lines. Key points:

- One netdev per `&fecN` DT node.
- Setup at probe: enable clocks, reset MAC, attach PHY via `of_phy_connect`, allocate BD ring buffers.
- TX: `fec_enet_start_xmit` → fills a BD, kicks the DMA, returns.
- RX: NAPI poll. The driver's interrupt schedules NAPI. NAPI loops over the RX ring, calls `napi_gro_receive` per packet.
- PHY: `linkstate` change events from the PHY (via MDIO polling) update netif_carrier_on/off, triggering `ifconfig` to show link state.

Walk of `fec_enet_start_xmit` (paraphrased):

```
static netdev_tx_t fec_enet_start_xmit(struct sk_buff *skb, struct net_device *ndev) {
    struct fec_enet_private *fep = netdev_priv(ndev);
```

```
    struct bufdesc *bd = fec_enet_get_nextbd(fep->bd_tx);
    /* Map skb data to a DMA-coherent buffer (or use the skb's own DMA-mapped page) */
    dma_addr_t addr = dma_map_single(..., skb->data, skb->len, DMA_TO_DEVICE);
    bd->cbd_bufaddr = addr;
    bd->cbd_datlen = skb->len;
    bd->cbd_sc = BD_ENET_TX_READY | BD_ENET_TX_LAST | BD_ENET_TX_INTR | BD_ENET_TX_TC;
    /* Write 1 to FEC TX descriptor active register to kick TX */
    writel(0, fep->hwp + FEC_TX_DESC_ACTIVE);
    return NETDEV_TX_OK;
}
```

NAPI receive:

```
static int fec_enet_rx_napi(struct napi_struct *napi, int budget) {
    int pkts_received = 0;
    while (pkts_received < budget) {
        struct bufdesc *bd = next_rx_bd(fep);
        if (bd->cbd_sc & BD_ENET_RX_EMPTY) break;   /* no more packets */
        struct sk_buff *skb = napi_alloc_skb(napi, bd->cbd_datlen);
        memcpy(skb->data, dma_data, bd->cbd_datlen);
        skb->protocol = eth_type_trans(skb, ndev);
        napi_gro_receive(napi, skb);
        bd->cbd_sc |= BD_ENET_RX_EMPTY;             /* return BD to RX ring */
        pkts_received++;
    }
    if (pkts_received < budget) napi_complete_done(napi, pkts_received);
    return pkts_received;
}
```

This is canonical NAPI: process up to `budget` packets per call. If fewer, complete and re-enable IRQ. Higher latency, much higher throughput than per-packet interrupts.

115.9 Key Notes

- Multiple network interfaces provide distinct links, while routing, bridging and segmentation define how traffic crosses between them.
- Each MAC/PHY path needs consistent wiring, management addressing and identity. Device enumeration does not establish a safe WAN/LAN assignment.
- A bridge forwards at layer 2, while a router applies layer-3 policy. Interface count alone does not create isolation.
- An SPI Ethernet chip’s integration depends on its driver and operating mode. Hardware socket offload and Linux netdev operation are different interfaces, so support must be established for the selected stack.

115.10 Pitfalls

- **MDIO address conflict.** Both PHYs configured to address 0 → only one responds. The other looks like “not present.” Verify PHY strap pins.
- **Pinmux conflict.** FEC2 pins overlap UART2 + I²C on some i.MX6ULL packages. Check `pinctrl-imx6ull.h` carefully. Some functions are mutually exclusive.
- **RMII REF_CLK direction.** Either the SoC supplies REF_CLK to the PHY, or the PHY’s crystal does, and the PHY supplies it back to the SoC. Wrong direction = no link. Configurable in DT via `clocks` property + PHY strap pins.
- **EMC noise.** Two RMII interfaces close together radiate similarly. Common-mode chokes on each Ethernet jack help.
- **No explicit MAC address.** Kernel generates from chip UID. Random reboots can yield different MACs if UID isn’t read correctly. Set `local-mac-address` in DT for production.
- **Setting MAC after `ip link up`.** Some PHYs latch the MAC at link-up. Changes after may not take effect. Set before `up`.
- **DM9051 IRQ level vs edge.** DM9051 datasheet specifies active-low level-triggered IRQ. Misconfigured edge-triggered loses interrupts and the interface hangs.
- **SPI Ethernet under load.** Even DM9051 caps at ~10 Mbps. If you try to use it as a primary interface, performance suffers. Use it as a management or low-bandwidth interface only.
- **W5500 expectation mismatch.** W5500 is not a netdev. If you ordered “SPI Ethernet” thinking it would integrate with `ip link`, you’ll be surprised. Order DM9051 or ENC28J60 for true netdev.
- **Bridge + ip address on members.** Once an interface is enslaved to a bridge, only the bridge gets an IP. The slave interfaces don’t participate in L3.

115.11 Going deeper

- **`drivers/net/ethernet/freescale/fec_main.c`**: read it. Canonical mainline NIC driver.
- **`drivers/net/ethernet/davicom/dm9051.c`**: SPI Ethernet driver.
- **`Documentation/networking/`** kernel docs, NAPI, bridging, VLAN, bonding.
- **`iproute2` manuals**: `ip link`, `ip addr`, `ip route`.
- **`nftables` wiki**: modern firewall/NAT.
- **NXP IMX6ULL Reference Manual ch. 22 (ENET)**: register-level FEC details.

- **KSZ8081 datasheet**: most-used PHY, strap-pin docs are critical.
- **IEEE 1588 / PTP**: for clock distribution over Ethernet (extends Ch 107’s GPS-time use case).
- **Ch 52**: the original FEC driver chapter.
- **Ch 91**: for WiFi as an alternative network interface.

> Next chapter: **Chapter 116: PMICs and the regulator framework**, power-management ICs (PCA9450, PF8200).

Chapter 116: PMICs and regulator framework

What: Power Management ICs, single-chip power solutions that replace the half-dozen discrete LDOs and buck converters around an SoC. We cover **NXP PCA9450** (the i.MX-recommended PMIC for i.MX8M. Also used on some i.MX6 designs), **NXP PF8200** (industrial), **Rohm BD71850MWV** (compact, integrated for i.MX6/8 cores). On the i.MX6ULL we walk the I²C register map of a typical PMIC, configure voltage rails for SoC + DDR + I/O via the kernel **regulator framework**, integrate with **DVFS** (Ch 51B), and measure the power savings from PMIC-coordinated voltage scaling vs always-on discrete LDOs. **MCU bridge:** Think of a PMIC like a programmable power-tree supervisor: it replaces discrete enables and LDO assumptions with sequenced rails the kernel can model. **DDR:** external DRAM that must be configured and trained before most software can run from it. **PMIC:** Power Management IC, a chip that sequences and regulates the board's voltage rails.

Why: every i.MX6ULL design has 4–6 voltage rails: 3.3 V (I/O), 1.35 V (DDR3), 1.275 V (SoC core), 2.5 V (analog), 1.8 V (some I/O), 5 V (USB). Discrete LDOs work but: (a) bring-up sequencing is tricky (DDR before SoC core, etc.), (b) no central control for sleep, (c) BOM = 6+ chips. A PMIC consolidates these into 1 chip with I²C control, programmable voltages, ramp-rate control, sequencing, and per-rail enable for runtime power management. For products that ship in volume or need real sleep/wake, a PMIC isn't optional, it's the only practical path.

Focus: The regulator framework treats every rail as a "supply". Drivers declare their consumer-supply relationship in the DT. The kernel computes the power-on order from the dependency graph, and waits for each rail to stabilise before letting consumers probe. The PMIC driver translates "set supply VDDARM to 1.275 V" into the right I²C writes. DVFS uses this: when cpufreq drops to 396 MHz, it calls `regulator_set_voltage(VDDARM, 1.150 V)` first, saving ~30 % of core power. If you get the boot-sequence wrong, for example the kernel starts the FEC before its PHY's 1.8 V rail is stable, you will see PHY probe failures that look random. **PHY:** physical-layer block or chip that converts digital MAC signals to electrical or radio signals.

116.1 Discrete vs PMIC

	Discrete LDOs/bucks	PMIC
Chip count	5–10	1

	Discrete LDOs/bucks	PMIC
PCB area	~600 mm²	~200 mm²
BOM cost (qty 1k)	$4–6	$3–5
Sequencing	external supervisor IC	built-in, programmable
Per-rail enable	additional GPIOs	I²C
DVFS support	software-toggling LDO enable (slow, noisy)	I²C voltage change (fast, glitch-free)
Sleep modes	external GPIOs + LDOs	per-rail sleep states + global low-power mode
Fault detection	per-LDO PG pin	unified PMIC FAULT line

PMICs are better in every dimension once a board has more than three rails or needs runtime power management. Discrete LDOs remain common only for the cheapest designs, or for battery rails where the PMIC's ~50 µA quiescent is too high.

116.2 Anatomy of a typical PMIC (PCA9450)

The PCA9450 (i.MX8M-recommended. Technically over-specified for i.MX6ULL but illustrative):

- 4× Buck DC-DC (BUCK1: 1.0–1.65 V @ 3.5 A for VDDARM, BUCK2: same for SOC, BUCK3: 1.0–1.65 V @ 1.5 A, BUCK4: 0.6–2.1875 V)
- 1× Buck for DDR (BUCK5: 1.1 V / 1.35 V)
- 6× LDO (LDO1: 1.6/3.3 V @ 100 mA, LDO2: 1.5–3.3 V @ 250 mA, LDO3–5: similar)
- I²C control + interrupt
- Programmable power-up sequence
- Sleep state with selectable rails on/off
- Fault detection (over-current, over-temp, under-voltage)

I²C register map (~80 registers):

Reg	Name	Purpose
0x01	DEV_ID	chip ID (0x10 for PCA9450A)
0x14	BUCK1OUT	output voltage code
0x15	BUCK1CTRL	enable, soft-start, ramp rate
0x18	BUCK2OUT	
0x20	BUCK5OUT	DDR voltage
0x21	BUCK5CTRL	
0x30	LDO1CTRL	
0x40	PWRON_DELAY	sequencing
0x70	INT_LATCH	interrupt status

Voltage encoding: per-buck typically `Vout = 0.6 + N × 0.025 V` for codes 0..63, so 1.275 V = code 27. The driver's translation table maps mV ↔ code.

116.3 The Linux regulator framework

`drivers/regulator/`, central framework. Every rail is a `struct regulator_dev`. Consumers `regulator_get()` by supply name in DT, then `regulator_enable()` / `regulator_disable()` / `regulator_set_voltage()`.

```
Hardware                Driver                Consumer
────────                ──────                ────────
PCA9450 BUCK1 ──────► pca9450-regulator.c ──► CPU clock driver (DVFS)
                                               "vddarm-supply = <&buck1>;"
PCA9450 BUCK5 ──────► pca9450-regulator.c ──► DDR3 driver
                                               "ddr-supply = <&buck5>;"
PCA9450 LDO1  ──────► pca9450-regulator.c ──► Ethernet PHY
                                               "vio-supply = <&ldo1>;"
```

DT:

```
&i2c1 {
    pca9450: pmic@25 {
        compatible = "nxp,pca9450a";
        reg = <0x25>;
        interrupt-parent = <&gpio4>;
        interrupts = <22 IRQ_TYPE_LEVEL_LOW>;

        regulators {
            buck1: BUCK1 {
                regulator-name = "VDD_ARM";
                regulator-min-microvolt = <600000>;
                regulator-max-microvolt = <1650000>;
                regulator-boot-on;
                regulator-always-on;
                regulator-ramp-delay = <3125>;
            };
            buck5: BUCK5 {
                regulator-name = "VDD_DRAM";
                regulator-min-microvolt = <1100000>;
                regulator-max-microvolt = <1350000>;
                regulator-boot-on;
                regulator-always-on;
            };
            ldo1: LDO1 {
                regulator-name = "VDD_SNVS";
                regulator-min-microvolt = <1600000>;
                regulator-max-microvolt = <3300000>;
                regulator-always-on;
            };
            /* ... etc ... */
        };
    };
};

&cpu0 {
    vddarm-supply = <&buck1>;
};

&fec1 {
    phy-supply = <&ldo1>;
};
```

After boot:

```
ls /sys/class/regulator/
# regulator.0  regulator.1  ...

cat /sys/class/regulator/regulator.0/name
# VDD_ARM

cat /sys/class/regulator/regulator.0/microvolts
# 1275000
```

Or `regulator_summary` (debugfs):

```
mount -t debugfs none /sys/kernel/debug
cat /sys/kernel/debug/regulator/regulator_summary
# regulator                       use open bypass voltage current     min     max
# regulator-dummy                    0   0      0     0mV     0mA     0mV     0mV
# VDD_SNVS                           2   2      0  3300mV     0mA  1600mV  3300mV
# VDD_ARM                            1   1      0  1275mV     0mA   600mV  1650mV
```

The regulator summary makes the power tree visible: you can see exactly which consumer keeps each rail enabled.

116.4 Power-up sequencing, the most subtle bring-up trap

i.MX6ULL has a required power-up sequence:

1. VDD_SNVS (always-on RTC domain), must be first
2. VDD_HIGH_IN (analog supply), within 10 ms of SNVS
3. VDD_ARM_IN, VDD_SOC_IN, together, within 100 ms
4. NVCC_DRAM (1.35 V for DDR3), before any DDR access
5. Per-bank I/O supplies (NVCC_GPIO_*), before any GPIO use

> **MCU bridge:** Think of Linux GPIO like the same pin set/reset block you used on STM32, but accessed through a kernel subsystem that owns numbering, direction, interrupts, and user-space exposure. **GPIO:** General-Purpose Input/Output, a pin controlled as a digital input, output, or interrupt source.

1. POR_B released (reset deassert), last

If you violate this, behavior ranges from “doesn’t boot” to “boots but crashes intermittently” to “silicon damage.” The PMIC’s programmable sequencer enforces this in hardware, set PWRON_DELAY registers per rail and the PMIC powers them in the right order at the right intervals.

At runtime, `regulator_enable()` walks the supply dependency graph and powers parent supplies first. Mark each rail with `regulator-boot-on` if the kernel inherits an already-on rail. Mark with `regulator-always-on` if it must never disable.

116.5 DVFS, coordinated voltage and frequency scaling

When cpufreq drops the CPU clock (Ch 51B), it should also drop the core voltage. The `cpufreq-dt` driver does this via the regulator framework:

```
&cpu0 {
    cpu-supply = <&buck1>;
    operating-points-v2 = <&cpu_opp_table>;
};

cpu_opp_table: opp-table {
    compatible = "operating-points-v2";

    opp-396000000 {
        opp-hz = /bits/ 64 <396000000>;
        opp-microvolt = <1150000>;
        opp-supported-hw = <0xc>, <0x7>;
    };
    opp-528000000 {
        opp-hz = /bits/ 64 <528000000>;
        opp-microvolt = <1200000>;
    };
    opp-696000000 {
        opp-hz = /bits/ 64 <696000000>;
        opp-microvolt = <1275000>;
    };
};
```

The OPP table tells cpufreq: “at 396 MHz, 1.15 V is sufficient.” On transition:

```
1. cpufreq decides to drop to 396 MHz
2. regulator_set_voltage(buck1, 1150000) - PMIC ramps down via I²C
3. After ramp + settling, write new clock divider
4. CPU now runs at 396 MHz / 1.15 V
```

Power saving from 1.275 V to 1.150 V:

- Static (leakage) power ∝ V. ~10 % reduction.
- Dynamic power ∝ V² × f. ~28 % reduction at same frequency, more if f drops too.

Combined with f drop from 696 → 396 MHz: ~60 % total power saving for “idle background” load. This is why DVFS exists.

116.6 Sleep state coordination

In suspend-to-RAM:

- Most rails disable.
- DDR rail stays on (self-refresh).
- VDD_SNVS always on (so RTC + wake circuits work).

The PMIC’s SLEEP mode does this in one command from Linux suspend callback:

```
/* Inside i.MX6ULL suspend driver */
if (pmic->sleep_mode) {
```

```
    /* PMIC enters its preconfigured sleep state */
    pca9450_set_state(pmic, PCA9450_STATE_SLEEP);
    /* Now SoC enters WFI; only SNVS and DDR are powered */
}
```

On wake: PMIC's wake pin (typically tied to an i.MX EXTRBOOT or PMIC's WAKE_IN) brings the PMIC back to active state, which restores all rails to their pre-sleep values, in the correct sequence. Linux resumes.

Without a PMIC, each rail needs its own GPIO with its own timing. The suspend driver grows complex. With a PMIC, suspend is one I²C transaction.

116.7 From scratch, minimal PMIC interaction over I²C

If you're prototyping with a discrete-LDO board, but want to demonstrate the principles, here's how you'd talk to a hypothetical PMIC's BUCK1 register directly:

```
/* pmic_test.c */
#include <linux/i2c-dev.h>
#include <fcntl.h>
#include <stdio.h>
#include <sys/ioctl.h>
#include <unistd.h>

#define PMIC_ADDR 0x25
#define REG_DEV_ID 0x00
#define REG_BUCK1_VSET 0x14

static int i2c_write(int fd, uint8_t reg, uint8_t val) {
    uint8_t buf[2] = { reg, val };
    return write(fd, buf, 2) == 2 ? 0 : -1;
}

static int i2c_read(int fd, uint8_t reg, uint8_t *val) {
    if (write(fd, &reg, 1) != 1) return -1;
    return read(fd, val, 1) == 1 ? 0 : -1;
}

int main(void) {
    int fd = open("/dev/i2c-0", O_RDWR);
    ioctl(fd, I2C_SLAVE, PMIC_ADDR);

    uint8_t id;
    i2c_read(fd, REG_DEV_ID, &id);
    printf("PMIC ID: 0x%02X\n", id);

    /* Read current BUCK1 setting */
    uint8_t v;
    i2c_read(fd, REG_BUCK1_VSET, &v);
    printf("BUCK1 code: 0x%02X (Vout = %.3f V)\n", v, 0.6 + v * 0.025);

    /* Lower BUCK1 to 1.150 V (code 22 = 0.6 + 22*0.025) */
    i2c_write(fd, REG_BUCK1_VSET, 22);
    sleep(1);
```

```
    i2c_read(fd, REG_BUCK1_VSET, &v);
    printf("After: BUCK1 code: 0x%02X (Vout = %.3f V)\n", v, 0.6 + v * 0.025);

    /* WARNING: directly writing PMIC registers bypasses the regulator framework.
     * Don't do this when consumers depend on the rail (you may starve VDDARM).
     * Use only for bring-up exploration. */
    return 0;
}
```

This is for understanding. In real Linux code you go through the regulator framework via the consumer driver, never directly poke PMIC registers from an unrelated process.

116.8 Key Notes

- Regulator providers describe rails and consumers request the supplies they need. This dependency graph should match the physical power tree.
- Voltage and frequency transitions need safe ordering and settling time. Operating points must come from qualified silicon and board requirements, not trial undervolting.
- Cold-start sequencing, runtime power changes and suspend state have different constraints. Kernel-managed supplies do not eliminate pre-kernel power dependencies.
- PMIC access changes live hardware state. Raw register access, shorted rails and guessed settings are not valid substitutes for a documented power design.

116.9 Pitfalls

- **Direct PMIC register pokes bypass safety.** Lowering VDDARM below the OPP minimum mid-execution = SoC crashes immediately. Always use the regulator framework.
- **OPP table wrong voltage.** Setting 396 MHz at 1.0 V (below spec) may “work” most of the time but crashes randomly. Match the SoC datasheet exactly.
- **Sleep-mode rails not configured.** PMIC enters sleep. User-relevant rail (e.g., backlight) stays on, wakes everything via leakage. Configure SLEEP_REG for each rail.
- **No `regulator-boot-on` on rails the kernel finds already on.** Kernel assumes off, tries to enable, may sequence wrongly. Mark every pre-up rail.
- **`regulator-always-on` everywhere.** Defeats DVFS and sleep. Use only for rails that genuinely cannot disable (DDR, RTC, etc.).

- **I²C bus contention.** PMIC, EEPROM, RTC, sensors all on i2c0. PMIC ops during heavy traffic delay → DVFS transitions slow → cpufreq stalls. Put PMIC on a dedicated I²C bus if you have one spare.
- **PMIC requires VDDA before I²C.** Some PMICs (PCA9450) won't ACK I²C until their internal LDOs are up. The kernel I²C probe must retry. The driver handles this. Ad-hoc test scripts may not.
- **OTP-programmed PMIC defaults.** Some PMICs ship with vendor-specific OTP values. If you got a "datasheet default" example, your part may differ. Read all registers at boot, verify against your design.
- **Ramp rate too fast.** Fast voltage steps cause overshoot. PMIC `regulator-ramp-delay` (µV/µs) controls slew. Default may be too aggressive for sensitive consumers.
- **WAKE_IN polarity wrong.** Active-low vs active-high, wrong polarity = PMIC ignores wake. Spec mismatch.
- **DDR rail timing.** DDR3 needs ≤1 ms from VREF to VDDQ stable. PMIC sequence must enforce this. Otherwise DDR initializes corruptly.

116.10 Going deeper

- **NXP PCA9450 Datasheet** + **AN12575 application note** (the design guide for i.MX8 power).
- **NXP PF8200 Datasheet**: industrial 14-rail PMIC.
- **Rohm BD71850MWV Datasheet**: compact i.MX-focused PMIC.
- **`drivers/regulator/`**: kernel framework + chip drivers.
- **`Documentation/devicetree/bindings/regulator/`**: DT binding docs.
- **`Documentation/power/regulator/`**: kernel regulator framework guide.
- **i.MX6ULL Reference Manual ch. 11 (Power Supply Strategy)**: the canonical power-up sequence.
- **Ch 51B**: runtime PM + DVFS chapter. The consumer side.
- **Ch 75**: INA226 + current monitoring. How you measure rail consumption.

> Next chapter: **Chapter 117: External RTC**, the last chapter of Part VII.

Chapter 117: External RTC

> **regmap:** a kernel helper that wraps register reads and writes over I2C, SPI, or MMIO. **MCU bridge:** Think of regmap like a typed wrapper around your read_reg() and write_reg() helpers, with caching, locking, and bus differences handled centrally.

What: external battery-backed real-time clocks. **Maxim DS3231** (TCXO, ±2 ppm, the highest-accuracy popular choice), **NXP PCF8563** (cheap, common, ±20 ppm), **Microchip MCP79410** (with built-in EEPROM, plus a unique ID). On the i.MX6ULL we wire I²C, walk the kernel `rtc-i2c` family of drivers (`rtc-ds1307` covers DS3231. `rtc-pcf8563` for PCF8563. `rtc-mcp7941x` for Microchip), use `hwclock` to sync between system clock and RTC, configure **alarm interrupts** for wake-from-suspend, and integrate with **Ch 51B's runtime PM** so the i.MX6ULL can sleep for hours and wake exactly on a scheduled RTC alarm.

Why: the i.MX6ULL has an *internal* RTC in the SNVS (Secure Non-Volatile Storage) domain, it survives reboots but loses time without a backup battery on the VDD_SNVS rail. Many board designs skip the SNVS battery to save 50 cents → the SoC's RTC is useless. The external RTC fix: a $0.50 chip plus a $0.30 coin cell on the I²C bus. The device knows the correct time on every cold boot, runs scheduled alarms even when Linux is off, and stays calibrated for years. For products that schedule actions ("daily sensor upload at 06:00") or need accurate timestamps in logs across power outages, an external RTC isn't optional.

Focus: The RTC chip is a 32.768 kHz oscillator plus counters, accessed over I²C. The kernel `rtc-*` driver exposes it as `/dev/rtcN`. `hwclock` syncs between the hardware clock and the system clock. `chrony` (or `systemd-timesyncd`) keeps the system clock disciplined from NTP or PPS, and writes the corrected time back to the RTC. Three clock domains coexist (hardware RTC, system clock, NTP source). Their interactions are what's tricky. Alarm interrupts let the RTC wake the SoC from suspend. The alarm pin must be wired to a GPIO that the kernel can use as a wake source. This wiring requirement is the most-skipped detail in RTC bring-up. **GPIO:** General-Purpose Input/Output, a pin controlled as a digital input, output, or interrupt source.

Tooling. This chapter uses `hwclock` (in `util-linux`), `chrony` for time discipline, optional `i2c-tools`.

- **Ubuntu-base (target):** `apt install util-linux chrony i2c-tools`
- **Buildroot:** `BR2_PACKAGE_UTIL_LINUX_HWCLOCK=y BR2_PACKAGE_CHRONY=y BR2_PACKAGE_I2C_TOOLS=y` **Buildroot:** a configuration-driven build system that produces a complete root filesystem and related images.

> - Full per-tool reference: Userspace tooling appendix. **MCU bridge:** Think of the rootfs as the firmware image's file-backed runtime environment. On an MCU you link everything into flash. On Linux, programs and config live in this mounted tree. **rootfs:** root filesystem, the directory tree mounted at / that contains /bin, /etc, /dev, and libraries.

117.1 Chip comparison

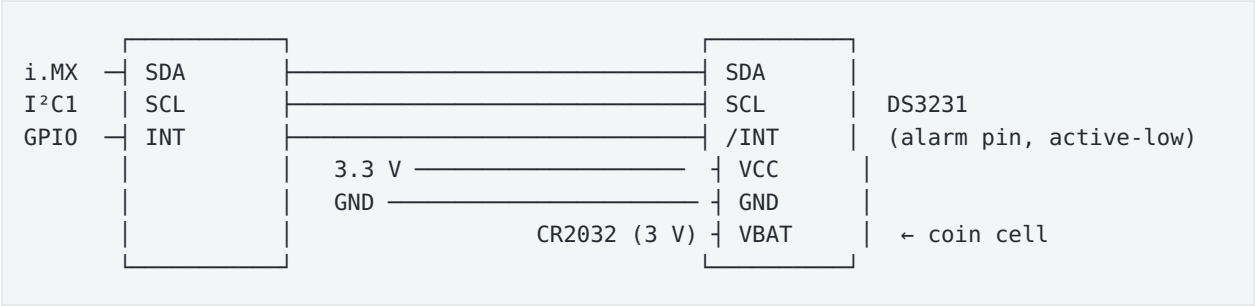
	DS3231	PCF8563	MCP79410
Crystal	integrated TCXO	external 32.768 kHz	external 32.768 kHz
Accuracy	±2 ppm (1 min/year)	±20 ppm (10 min/year)	±20 ppm
Temperature compensation	yes, internal	no	no
Battery life	6–10 years on CR2032	6–10 years	6–10 years
Alarm interrupts	2	1	2
EEPROM	no	no	64 B EEPROM + 128 B SRAM
Unique ID	no	no	yes (6 bytes)
Power supply	2.3–5.5 V	1.8–5.5 V	1.8–5.5 V
Cost	$1.50 (chip) / $2.50 (module with battery)	$0.30	$0.80
Kernel driver	`rtc-ds1307` (covers DS3231 too)	`rtc-pcf8563`	`rtc-mcp7941x`

Pick guide:

- **DS3231**: accuracy matters (logs, scheduling, fleet sync). Willing to pay $1.50.

- **PCF8563**: BOM critical. ±10 min/year drift acceptable. You have NTP to correct.
- **MCP79410**: when you also want a tiny EEPROM (board serial, calibration data) without adding a separate chip.

117.2 Wiring DS3231



The INT pin is an open-drain output. Pull-up to 3.3 V. Wire to a GPIO that maps to a wake source (the i.MX6ULL EXTRBOOT or ONOFF wake pins are ideal. Otherwise any GPIO with `wake-up-source` capability).

117.3 Device tree

```
&i2c1 {
    pinctrl-names = "default";
    pinctrl-0 = <&pinctrl_i2c1>;
    clock-frequency = <100000>;
    status = "okay";

    rtc@68 {
        compatible = "maxim,ds3231";       /* uses rtc-ds1307 driver */
        reg = <0x68>;
        interrupt-parent = <&gpio4>;
        interrupts = <23 IRQ_TYPE_EDGE_FALLING>;
        interrupts-extended = <&gpio4 23 IRQ_TYPE_EDGE_FALLING>;
        wakeup-source;
    };
};
```

After boot:

```
dmesg | grep rtc
# rtc-ds1307 1-0068: registered as rtc0
# rtc-ds1307 1-0068: setting system clock to 2026-05-31 12:34:56 UTC (...)
```

`/dev/rtc0` (and `/dev/rtc` symlink) appears. The kernel automatically reads the RTC at boot, sets the system clock to it.

If the i.MX6ULL has its SNVS RTC enabled, you'll have *two* RTCs (`rtc0` = SNVS, `rtc1` = DS3231). Decide which is primary (DS3231 if your SNVS lacks battery). Override:

```
# Force hwclock to use the DS3231
hwclock --rtc=/dev/rtc1 --systohc
```

117.4 hwclock, system clock ↔ RTC sync

```
hwclock --show                    # read hw clock
# 2026-05-31 12:34:56.123456+00:00

hwclock --systohc                 # write system → hw
hwclock --hctosys                 # write hw → system

date -s "2026-05-31 15:00:00"
hwclock --systohc                 # save new time to RTC battery
```

At boot, systemd runs `hwclock --hctosys` automatically (or kernel does it via the rtc-* driver's `set_system_time_from_rtc` if `CONFIG_RTC_HCTOSYS=y`).

Time zone: hwclock can store the RTC in UTC or local time. UTC is strongly recommended, local time breaks across DST transitions. `/etc/adjtime` records which is in use.

117.5 Alarm interrupts, waking from suspend

> **Privilege boundary:** $ means normal user. # or sudo means root and can change host or target state. After a privileged command, verify the expected device, service, or file appears before continuing. Roll back by undoing the config change or stopping the service you just enabled.

DS3231 has two alarms. PCF8563 has one. Set an alarm:

```
# Wake in 30 seconds via rtcwake
rtcwake -m mem -s 30
# (system suspends; in 30 s, RTC alarm fires → SoC wakes via INT pin → kernel resumes)
```

`rtcwake` is the universal tool. Under the hood:

1. `ioctl(rtc_fd, RTC_WKALM_SET, &alarm)` to set the alarm time + enable.
2. `echo mem > /sys/power/state` to suspend.
3. RTC's INT pin pulled low at the alarm time.
4. INT pin wired to a wake-capable GPIO → ARM core wakes from WFI.
5. Kernel resumes. Alarm is cleared.

From C:

```
#include <linux/rtc.h>
#include <fcntl.h>
#include <sys/ioctl.h>

int fd = open("/dev/rtc0", O_RDWR);

struct rtc_wkalrm alarm = {0};
alarm.enabled = 1;
/* Set to current time + 5 minutes */
ioctl(fd, RTC_RD_TIME, &alarm.time);
alarm.time.tm_min += 5; if (alarm.time.tm_min >= 60) alarm.time.tm_min -= 60,
alarm.time.tm_hour++;
ioctl(fd, RTC_WKALM_SET, &alarm);

/* Now suspend the system */
system("echo mem > /sys/power/state");
/* Execution resumes here after wake */
```

For a battery-powered sensor: sleep 10 minutes. Wake. Read sensors. Transmit. Sleep again. With DVFS + suspend-to-RAM + RTC alarm + power-managed peripherals, the i.MX6ULL idles at < 5 mA between samples.

117.6 How the rtc-ds1307 driver actually works

`drivers/rtc/rtc-ds1307.c`, covers DS1307, DS1337, DS1338, DS1339, DS3231, DS3232, …, a family of register-compatible chips.

Key functions:

```
static int ds3231_get_time(struct device *dev, struct rtc_time *t) {
    struct ds1307 *ds = dev_get_drvdata(dev);
    u8 regs[7];
    regmap_bulk_read(ds->regmap, 0x00, regs, 7);   /* read sec..year */
    t->tm_sec  = bcd2bin(regs[0] & 0x7F);
    t->tm_min  = bcd2bin(regs[1] & 0x7F);
    t->tm_hour = bcd2bin(regs[2] & 0x3F);          /* 24h mode */
    t->tm_wday = bcd2bin(regs[3] & 0x07) - 1;
    t->tm_mday = bcd2bin(regs[4] & 0x3F);
    t->tm_mon  = bcd2bin(regs[5] & 0x1F) - 1;
    t->tm_year = bcd2bin(regs[6]) + 100;           /* RTC stores year-2000 */
    return 0;
}
```

The RTC chips store time in **BCD** (Binary-Coded Decimal, each nibble is a decimal digit). The driver does the bcd↔bin conversion. The 7-byte read covers all date/time registers in one I²C transaction.

For the alarm:

```
static int ds3231_set_alarm(struct device *dev, struct rtc_wkalrm *t) {
    /* Write alarm-time registers (similar BCD encoding) */
    /* Set ALARM_INTERRUPT_ENABLE bit */
```

```
    /* Clear OSCILLATOR_STOP_FLAG if set (otherwise alarm may miss) */
    return 0;
}

static irqreturn_t ds3231_irq(int irq, void *dev_id) {
    /* Read STATUS_REG; if alarm flag set, clear it + call rtc_update_irq */
    rtc_update_irq(rtc, 1, RTC_AF | RTC_IRQF);
    return IRQ_HANDLED;
}
```

`rtc_update_irq` notifies user-space (via select/poll on `/dev/rtcN`) that the alarm fired.

117.7 Time discipline, RTC + NTP + chrony together

Your system has three clock sources:

1. **External RTC** (hardware, battery-backed, ±2 to ±20 ppm)
2. **System clock** (software, derived from CPU timer or LPC timer)
3. **NTP / PPS** (network or GPS, Ch 107)

chrony's job: discipline the system clock from the best available source, and write back to the RTC.

```
# /etc/chrony/chrony.conf
pool 2.pool.ntp.org iburst                # NTP source

# RTC: use as backup when network unavailable
rtcfile /var/lib/chrony/chrony.rtc
rtcsync                                    # write system→RTC periodically

# If you have a GPS PPS (Ch 107)
refclock SHM 0 refid GPS poll 4 noselect
refclock PPS /dev/pps0 refid PPS lock GPS prefer trust
```

`rtcsync` makes chrony sync the system→RTC every 11 minutes. On reboot, the RTC has accurate time even if NTP isn't immediately available.

117.8 From scratch, minimal DS3231 reader in C

For when you want to debug RTC behavior without the kernel driver:

```
/* ds3231_min.c */
#include <fcntl.h>
#include <linux/i2c-dev.h>
#include <stdio.h>
#include <sys/ioctl.h>
#include <unistd.h>

#define DS3231_ADDR 0x68
```

```
static int i2c_fd;

static uint8_t bcd2bin(uint8_t v) { return (v >> 4) * 10 + (v & 0x0F); }
static uint8_t bin2bcd(uint8_t v) { return ((v / 10) << 4) | (v % 10); }

int main(void) {
    i2c_fd = open("/dev/i2c-0", O_RDWR);
    ioctl(i2c_fd, I2C_SLAVE, DS3231_ADDR);

    /* Read 7-byte time block (sec, min, hour, dow, day, mon, year-2000) */
    uint8_t reg = 0;
    write(i2c_fd, &reg, 1);
    uint8_t buf[7];
    read(i2c_fd, buf, 7);

    int sec  = bcd2bin(buf[0] & 0x7F);
    int min  = bcd2bin(buf[1] & 0x7F);
    int hour = bcd2bin(buf[2] & 0x3F);
    int day  = bcd2bin(buf[4] & 0x3F);
    int mon  = bcd2bin(buf[5] & 0x1F);
    int year = bcd2bin(buf[6]) + 2000;
    printf("DS3231: %04d-%02d-%02d %02d:%02d:%02d\n",
           year, mon, day, hour, min, sec);

    /* Read temperature (DS3231 has a built-in temp sensor for TCXO compensation) */
    reg = 0x11;
    write(i2c_fd, &reg, 1);
    uint8_t t[2];
    read(i2c_fd, t, 2);
    int temp_int = (int8_t)t[0];
    int temp_frac = (t[1] >> 6) * 25;
    printf("Temp: %d.%02d °C\n", temp_int, temp_frac);

    return 0;
}
```

The DS3231’s built-in thermometer is a useful bonus. It drives the TCXO temperature compensation internally and can also be read as a ±3 °C ambient sensor. Use it for “is the device too hot” alerts.

117.9 Key Notes

- The RTC retains time independently of the system clock, while NTP or another reference disciplines system time. Synchronization direction and validity need one clear policy.
- Battery backup preserves a clock only within the actual supply arrangement. A fitted socket does not establish a healthy backup source or valid time.
- An alarm needs a supported wake path from the RTC to the SoC. Programming an alarm does not prove that the board can resume from the selected sleep state.

- RTC numbering, UTC/local-time policy and oscillator status matter when several clocks exist. A successful read should not hide that time may have been lost or never initialized.

117.10 Pitfalls

- **No backup battery.** Battery socket present but empty. Time resets on every power-cycle. Always populate the battery (or use a supercap with charging circuit).
- **Battery dead, no monitoring.** CR2032 lasts 6–10 years but does die. After 8 years, the RTC silently loses time on next outage. Monitor the OSF (Oscillator Stop Flag) bit. Alert when set.
- **VBAT < VCC on power-up.** Some RTCs (DS3231) need VCC ≥ VBAT before they start counting. If VCC ramps slowly, RTC may not start. Add a power-good supervisor or check OSF after every boot.
- **OSF not cleared.** DS3231 latches OSF after VBAT loss. If you don't clear it, the alarm interrupts are inhibited. Clear in CONTROL_STATUS register at boot.
- **I²C bus pull-up missing or too weak.** DS3231 expects 4.7 kΩ to 10 kΩ pull-ups. Weaker = slow edges = errors at 400 kHz.
- **Multiple chips at 0x68.** The MPU-6050 IMU also defaults to address 0x68, bus conflict. The DS3231 address is fixed. Strap the MPU-6050's AD0 pin to move it to 0x69.
- **BCD vs binary confusion.** Direct register reads return BCD. Treating as binary gives nonsense (0x13 read as 19 instead of 13). Convert with bcd2bin.
- **Year-2100 problem.** Some RTCs store year as 0..99. The "century" bit handles 2000–2099 only. Post-2100 these RTCs roll over to 2000.
- **INT pin not wake-capable.** GPIO wired but not configurable as a wake source → suspend works but never wakes. Verify with `dmesg | grep wakeup`.
- **Alarm time wrong fields.** DS3231 alarm requires specific masks (DY/DT bit for day-of-week vs day-of-month). Wrong mask = alarm fires constantly or never.
- **MCP79410 EEPROM at separate I²C address.** Same chip, but the EEPROM section is at 0x57, not 0x6F (the RTC's address). Easy to miss.
- **Temperature sensor not used for TCXO comp without proper init.** DS3231 has automatic TCXO compensation but require Convert Temp bit to be set periodically (or run once and rely on automatic 64-second updates).

117.11 Going deeper

> **Examples and production:** Do not burn fuses, enroll production keys, or sign release images while using these examples. Use throwaway keys and back up the unsigned image plus the key directory before testing irreversible security flows.

- **Maxim DS3231 Datasheet**: TCXO theory, alarm registers, OSF semantics.
- **NXP PCF8563 Datasheet**: sibling cheap RTC.
- **Microchip MCP79410 Datasheet**: RTC + EEPROM + Unique ID.
- **`drivers/rtc/rtc-ds1307.c`**: covers DS3231 too. Readable.
- **`drivers/rtc/`**: see other RTC drivers. Same pattern.
- **`Documentation/rtc.txt`**: RTC subsystem overview.
- **`hwclock(8)`, `rtcwake(8)`, `chrony.conf(5)`**: the user-space toolchain.
- **NTPv4 + chrony architecture**: for the multi-source clock discipline math.
- **Ch 51B**: runtime PM + suspend. The consumer side of “wake on RTC alarm.”
- **Ch 107**: GPS+PPS for the sub-µs precision case where an RTC alone isn’t enough.

> **End of Part VII, Device Cookbook (Ch 64–117, 54 chapters).** These chapters compare representative devices, bus interfaces, driver responsibilities and integration pitfalls. They are independent references, not a required build sequence. The Key Notes collect the design decisions that still matter when the implementation changes.

> Next: **Part VIII, Debug, production, advanced** examines observability, reproducible builds, secure boot and updates. The question shifts from how one interface works to how a system can be diagnosed, maintained and recovered. **MCU bridge:** Think of JTAG like SWD debugging on Cortex-M: halt, read registers, set breakpoints. The Cortex-A path adds MMU state, privilege modes, and more complex reset behavior. **JTAG:** the hardware debug scan chain used to halt, inspect, and single-step CPUs.

Chapter 118: JTAG, OpenOCD, GDB at every layer

What: the **hardware-level debug stack**: **JTAG adapters** (FT2232H-based generic, J-Link, SEGGER pro), **OpenOCD** as the software bridge between adapter and target, and **GDB** as the user interface. We wire JTAG to the i.MX6ULL's debug header, write an OpenOCD config, halt the CPU at the very first reset vector instruction, single-step through U-Boot, attach to the running kernel with full `vmlinux` symbol resolution, and inspect a kernel module's variables interactively. **MCU bridge:** Think of U-Boot like a much larger boot stub plus debug monitor: it initializes hardware, loads the next image, and gives you commands before Linux starts. **JTAG:** the hardware debug scan chain used to halt, inspect, and single-step CPUs. **U-Boot:** the bootloader that initializes enough hardware to load and start the Linux kernel. **OpenOCD:** the host program that talks to a JTAG adapter and exposes a GDB server.

Why: when the LED doesn't blink and `printk` isn't an option (the kernel hasn't started yet), JTAG is the only way in. The same is true when U-Boot hangs in DCD execution, or the kernel oopses before serial init. JTAG lets you read every register, dump every memory region, set hardware breakpoints, and step a single instruction at a time, at any layer (bare-metal, bootloader, kernel, user-space). It's the difference between guessing the boot fails somewhere in CCM init, and seeing that XTAL_24M is at 0 mV because the crystal isn't running. **CCM:** Clock Controller Module. It selects clock sources, dividers, and gates for the SoC. **DCD:** Device Configuration Data: ROM-executed register writes that prepare clocks and DDR before your code runs.

Focus: OpenOCD is the bridge: it speaks USB to the JTAG adapter and exposes a GDB-server on TCP. GDB connects, fetches symbols from your ELF, and gives you the familiar `b`, `n`, `s`, `p` commands. The tricky parts are: getting the adapter's USB-IDs right for OpenOCD, choosing the right *target* config (Cortex-A7 vs Cortex-M differ massively), handling Cortex-A's complications (multi-CPU, secure-vs-non-secure world, MMU on/off), and giving GDB the right symbol files at the right times (one ELF for bare-metal, another for U-Boot, another for kernel + per-module symbols). **ELF:** Executable and Linkable Format, the standard Linux object and executable file format.

Tooling. Host: `openocd`, `gdb-multiarch` (or `arm-none-linux-gnueabihf-gdb` from your cross-toolchain). **Target:** nothing, gdb attaches via OpenOCD over JTAG, no on-target software needed. Ubuntu-host install: `apt install openocd gdb-multiarch`. Full per-tool reference: Userspace tooling appendix. **MCU bridge:** Think of the rootfs as the firmware image's file-backed runtime environment. On an MCU you link

> everything into flash. On Linux, programs and config live in this mounted tree.
> **rootfs:** root filesystem, the directory tree mounted at / that contains /bin, /etc, /dev, and libraries.

118.1 JTAG basics, the 4-wire debug bus

JTAG (IEEE 1149.1) is a serial scan chain. Four mandatory wires + 1 optional + 1 hardware reset:

```
TDI  — Test Data In   (host → target shift)
TDO  — Test Data Out  (target → host shift)
TCK  — Test Clock     (host drives)
TMS  — Test Mode Sel  (host steers FSM)
TRST — Test Reset     (optional; if absent, soft-reset via TMS sequence)
SRST — System Reset   (optional; resets the SoC itself)
```

The target has a TAP (Test Access Port) state machine. TMS steers it through states (Run-Test-Idle, Shift-IR, Shift-DR, etc.). The host sends bits via TDI, sees them shifted back via TDO. Higher-level commands (read register, read memory, halt CPU, single-step) are encoded as sequences of these primitives.

ARM Cortex-A debug uses the **CoreSight** extension on top of JTAG (or SWD on Cortex-M. Cortex-A uses JTAG). Within CoreSight, **DAP** (Debug Access Port), **ETM** (trace), **TPIU** (trace output) live.

> **CoreSight:** ARM’s debug and trace block family behind JTAG debugging on Cortex-A.

118.2 Adapter comparison

	FT2232H breakout	SEGGER J-Link EDU	SEGGER J-Link Pro	NXP MCU-Link Pro
Cost	$15–25	$60 (non-commercial)	$400	$50
Speed	up to 30 MHz JTAG clock	up to 50 MHz	50+ MHz	30 MHz
OpenOCD support	excellent	yes	yes	partial

	FT2232H breakout	SEGGER J-Link EDU	SEGGER J-Link Pro	NXP MCU-Link Pro
Proprietary tools	n/a	J-Link GDBServer (closed)	yes	proprietary
SWD support	yes	yes	yes	yes
Voltage flexibility	1.8, 5 V	1.2, 5 V	1.2, 5 V	1.8, 5 V
Use case	hobbyist + most pro work	dev	production volume	NXP-specific

Pick guide:

- **FT2232H breakout** (e.g., Olimex ARM-USB-OCD-H, FTDI's eval board), the workhorse for OpenOCD. Cheap, works with everything.
- **J-Link EDU**: when you want SEGGER's polished GDBServer + Ozone GUI, non-commercial use.
- **J-Link Pro**: production / commercial.

118.3 Wiring JTAG to the Point Atom

The Point Atom MINI exposes the JTAG header on a 2.54 mm 10-pin connector (or 20-pin Cortex JTAG. Check your specific board revision). Pinout (10-pin Cortex):

```
1  VTREF       — target voltage reference (3.3 V on i.MX6ULL)
2  TMS / SWDIO
3  GND
4  TCK / SWCLK
5  GND
6  TDO / SWO
7  KEY         — present-pin (low if connected)
8  TDI         — JTAG only
9  GND
10 RESET       — SRST
```

Wire to the adapter's matching connector (often it's the same Cortex pinout).

Important: VTREF must be 3.3 V. The adapter senses this and adjusts its IO level. Misconfigured VTREF (5 V into a 3.3 V SoC) = blown JTAG pins on the SoC. Permanent.

118.4 OpenOCD config

> **Driver choice:** Use the in-tree, maintained driver first. Use out-of-tree, spidev, or custom-driver paths only after you accept the kernel-version maintenance cost and document who owns updates.

OpenOCD needs three things: interface config (adapter), target config (CPU), and board config (memory map, reset behavior).

`/etc/openocd/imx6ull-pa.cfg`:

```
# Adapter — FTDI-based interface (e.g. Olimex ARM-USB-OCD-H)
source [find interface/ftdi/olimex-arm-usb-ocd-h.cfg]
adapter speed 1000      ;# Start at 1 MHz; ramp up to 6 MHz after init

# i.MX6ULL is single-core Cortex-A7
source [find target/imx6ull.cfg]

# Reset config: SRST = system reset; TRST optional
reset_config srst_only srst_push_pull

# Board-specific: where SDRAM lives, the OCRAM, etc.
set OCRAM_BASE 0x00900000
set OCRAM_SIZE 0x00020000      ;# 128 KB
set DRAM_BASE  0x80000000
set DRAM_SIZE  0x20000000      ;# 512 MB

# Init script — runs at openocd startup
init
reset halt
```

The `target/imx6ull.cfg` (shipped with OpenOCD) declares the CoreSight DAP, the A7 core, the TAP-IDCODE expected from the chip. If your SoC variant isn't in stock OpenOCD, you may need a custom file.

Run:

```
openocd -f /etc/openocd/imx6ull-pa.cfg
# Open On-Chip Debugger 0.12.0+...
# Info : Listening on port 6666 for tcl connections
# Info : Listening on port 4444 for telnet connections
# Info : auto-selecting first available session transport "jtag"
# Info : clock speed 1000 kHz
# Info : JTAG tap: imx6ull.cpu tap/device found: 0x5ba00477 (mfg: 0x23b ARM, part:
0xba00, ver: 0x5)
# Info : Listening on port 3333 for gdb connections
# Info : target halted in ARM state due to debug-request
```

OpenOCD listens on:

- TCP 3333, GDB protocol

- TCP 4444, telnet (for direct OpenOCD commands)
- TCP 6666, TCL

You can `telnet localhost 4444` and issue raw commands:

```
> reset halt
> mdw 0x00900000 4
0x00900000: 12345678 9abcdef0 deadbeef cafebabe
> reg
> mww 0x20a4000 0x00000001     ;# write a register
> resume
```

118.5 GDB connected to OpenOCD, bare-metal

For Ch 9's LED-in-assembly bare-metal example:

```
arm-none-eabi-gdb led.elf
(gdb) target remote :3333
Remote debugging using :3333
0x00900000 in ?? ()
(gdb) load                          # write the ELF to target memory
Loading section .text, size 0x40 lma 0x900000
Start address 0x900000
(gdb) break _start
Breakpoint 1 at 0x900000: file led.S, line 10.
(gdb) continue
Breakpoint 1, _start () at led.S:10
10        ldr r0, =GPIO1_BASE
(gdb) step
(gdb) info registers
r0  0x209c000  34602880
r1  ...
(gdb) x/16w 0x209c000             # examine GPIO1 register block
```

Single-step a literal assembly program. Watch registers change. See exactly which instruction flips the GPIO bit.

> **MCU bridge:** Think of Linux GPIO like the same pin set/reset block you used on STM32, but accessed through a kernel subsystem that owns numbering, direction, interrupts, and user-space exposure. **GPIO:** General-Purpose Input/Output, a pin controlled as a digital input, output, or interrupt source.

This is how you learn the bare-metal layer: step every instruction and match it to the reference manual.

118.6 GDB on U-Boot

```
arm-none-linux-gnueabihf-gdb u-boot
(gdb) target remote :3333
(gdb) symbol-file u-boot          # already loaded
(gdb) add-symbol-file spl/u-boot-spl 0x00908000   # SPL lives in OCRAM
(gdb) break board_init_f
(gdb) continue
Breakpoint 1, board_init_f (boot_flags=0) at common/board_f.c:880
880        gd = (gd_t *)((CONFIG_SYS_INIT_SP_ADDR) & ~7) - 16;
(gdb) next
881        ...
```

You're now stepping through U-Boot's C code, single-line at a time, in the same way as gdb on a user-space binary. Set breakpoints in `board_init_r`, in `cmd_bootm`, anywhere.

U-Boot is built without `-O2` if you `make DEBUG=1`. With default `-O2` some variables are optimized out, debug builds give cleaner GDB experience.

118.7 GDB on the running kernel, `vmlinux` + KASLR

For kernel debug, you need:

1. `CONFIG_DEBUG_KERNEL=y`, `CONFIG_DEBUG_INFO=y`, `CONFIG_GDB_SCRIPTS=y` in kconfig.
2. `vmlinux` (the unstripped ELF in the kernel build dir).
3. KASLR (Kernel Address Space Layout Randomization) disabled, or the offset known.

```
# Build kernel with debug info
make ARCH=arm CROSS_COMPILE=arm-none-linux-gnueabihf- imx_v7_defconfig
echo CONFIG_DEBUG_INFO=y >> .config
echo CONFIG_GDB_SCRIPTS=y >> .config
make ARCH=arm zImage modules

# Connect
arm-none-linux-gnueabihf-gdb vmlinux
(gdb) target remote :3333
(gdb) lx-symbols                  # the kernel GDB scripts: load module symbols too
(gdb) break start_kernel
(gdb) continue
... U-Boot runs, kernel handoff happens, breakpoint hits ...
Breakpoint 1, start_kernel () at init/main.c:925
925     {
(gdb) bt
#0  start_kernel () at init/main.c:925
#1  0x80101000 in stext () at arch/arm/kernel/head.S:115
```

The `lx-symbols` GDB command (part of the kernel's `scripts/gdb/` Python helpers) iterates `/proc/modules`-style data inside the kernel to find loaded modules and load each `.ko`'s symbols at the correct offset. Now `break my_driver_probe` works on a dynamically loaded driver.

118.8 Inspecting kernel state with `lx-*` commands

The kernel's GDB scripts (`scripts/gdb/linux/`):

```
(gdb) lx-dmesg                     # the printk ring buffer, formatted
(gdb) lx-ps                         # task list
(gdb) lx-lsmod                      # loaded modules
(gdb) lx-cmdline                    # boot command line
(gdb) lx-version                    # uname-equivalent
(gdb) lx-iomem                      # like /proc/iomem
(gdb) p $lx_current()->comm         # the running task's name
(gdb) p init_task                   # init process struct
```

These walk the kernel's in-memory data structures the same way `procfs` does, but live and without needing the kernel itself functional (debug a wedged kernel).

118.9 Hardware breakpoints + watchpoints

Cortex-A7 has 6 hardware breakpoints and 4 watchpoints (counts may vary). GDB uses them transparently:

```
(gdb) hbreak my_function              # hardware breakpoint
(gdb) watch *(int *)0x80100000        # break on write to this addr
(gdb) rwatch global_var                # break on read
```

Hardware breakpoints are needed for read-only memory (you can't replace the instruction with a breakpoint trap if the memory is in flash/ROM). Software breakpoints (default `break`) replace the instruction with the ARM `BKPT` (A32: `0xE1200070`. Thumb-2: `0xBE00`). The CPU traps to the prefetch-abort vector. The kernel/debugger sees a debug-exception, and gdb regains control.

118.10 Reset behavior, the trickiest part

Often the target is in an unknown state from a previous crash. You want to halt at the first instruction after reset.

OpenOCD's `reset halt` does:

1. Assert SRST.
2. Configure the CPU's debug-halt-on-reset bit (DBGEN + HRR in DSCR).
3. Deassert SRST.
4. CPU starts, immediately halts at PC = reset vector (= Boot ROM entry).

To debug ROM execution: from `reset halt` state, set a breakpoint on the first DCD-runtime address, `resume`, watch ROM execute DCD then jump to your image.

`reset init` runs OpenOCD's TCL "init" hooks first (typically configure clocks/DDR via OpenOCD before loading your binary), useful for debugging code that requires DDR but you don't want to depend on the Boot ROM doing it.

> **DDR:** external DRAM that must be configured and trained before most software can run from it.

118.11 Key Notes

> **Example safety:** Do not burn fuses, enroll production keys, or sign release images while studying these examples. Use throwaway keys and back up the unsigned image plus the key directory before testing irreversible security flows.

> **Privilege boundary:** $ means normal user. # or sudo means root and can change host or target state. After a privileged command, verify the expected device, service, or file appears before continuing. Roll back by undoing the config change or stopping the service you just enabled.

- JTAG provides a hardware access path when the target cannot run a software debug agent. The adapter, OpenOCD configuration, and GDB symbol files have separate responsibilities.
- Reset control and memory initialization determine which addresses are usable. Halting a core does not make uninitialized DDR safe to load or inspect.
- Symbols describe a particular image at particular runtime addresses. U-Boot relocation, kernel address randomization, and module loading each change that relationship.
- Hardware breakpoints and watchpoints are finite resources. Read-only code and changing memory state place different demands on them.
- Halting or writing target state can disturb interrupts, watchdogs, and peripherals. A debugger observation is not evidence of normal real-time behavior.
- Debug accessibility is a production security and support tradeoff. JTAG-disable fuses are irreversible provisioning decisions, not reading prerequisites.

118.12 Pitfalls

- **VTREF wrong.** Adapter at 5 V into 3.3 V SoC fries the JTAG pins. Always confirm VTREF before connecting.

- **JTAG disabled by fuse.** Some boards ship with the SJC_DISABLE eFuse blown for security. JTAG silently doesn't connect. Verify with a fresh chip.
- **SRST not wired.** Without SRST, OpenOCD's `reset` is a soft-reset (TMS sequence). Some states (CPU in WFI) can't be reset this way. Wire SRST for production debug.
- **TCK too fast for unhardened wiring.** Long flying-wire JTAG (>10 cm) needs slower TCK (≤ 1 MHz). Start at 1 MHz. Ramp up only if reliable.
- **Multi-core target.** If your SoC is multi-core (i.MX6ULL is single-core. Most newer chips are multi), OpenOCD config must declare each core separately.
- **Cortex-A MMU on, virtual vs physical addresses.** After MMU is enabled, `mdw 0x80100000` queries virtual address. Sometimes you want physical. Use `mdw phys 0x80100000` in OpenOCD.
- **GDB `monitor` commands.** You can run OpenOCD commands from GDB via `monitor`: `monitor reset halt`. Useful when you don't want to switch windows.
- **Symbols missing for kernel module.** `lx-symbols` works only on debug-built kernels. Check `CONFIG_GDB_SCRIPTS=y`.
- **KASLR.** Kernel address randomization shifts vmlinux symbols at each boot. Disable for debug (`nokaslr` boot arg) or use `add-symbol-file` with the runtime offset.
- **OpenOCD config drift.** Each OpenOCD version slightly different config syntax. Tested-known-good configs in your repo save hours.
- **Adapter unplugged mid-session.** OpenOCD doesn't recover. Restart it. GDB needs `target remote :3333` again.
- **HW breakpoint exhaustion.** Cortex-A7 has 6 HW breakpoints. If you set 7, GDB silently uses software for the 7th, which fails on ROM/flash.

118.13 Going deeper

- **OpenOCD User Guide** (https://openocd.org/doc/html/), the canonical reference.
- **ARM Cortex-A Series Programmer's Guide, ch. On Debug**: CoreSight, DAP, ETM in depth.
- **`scripts/gdb/`** in Linux source, the kernel's GDB scripts.
- **`Documentation/dev-tools/gdb-kernel-debugging.rst`** in Linux source.
- **NXP IMX6ULL Reference Manual, ch. 56 (System JTAG Controller)**: fuse and security configuration.
- **SEGGER J-Link User Guide**: for the J-Link path.
- **Ch 119**: kernel debugging without JTAG (`ftrace`, `kgdb`, eBPF).

- **Ch 120**: user-space debugging with gdbserver + perf + strace.
- **Ch 125A**: VSCode + gdbserver for the IDE-driven debug workflow.

> Next chapter: **Chapter 119: Kernel debugging without JTAG**, ftrace, eBPF, KGDB.

Chapter 119: Kernel debugging without JTAG

> **ELF:** Executable and Linkable Format, the standard Linux object and executable file format. **Device Tree:** a data file that describes board hardware to the Linux kernel instead of hard-coding it in C. **MCU bridge:** Think of Device Tree like a board-level hardware description table that replaces hard-coded #define LED_PORT GPIOA decisions. Unlike an MCU header, the kernel parses it at boot and matches it to drivers.

> **What:** the **software-only kernel debugging toolkit** that works on a deployed device with no hardware debug access. **printk**'s deeper toolbox (`pr_debug`, `dynamic_debug`, ring-buffer levels), **ftrace** (function tracer + `function_graph` + tracepoint events), **trace-cmd** + **KernelShark** (record + GUI), **bpftrace** and **bcc** (eBPF for live kernel introspection), **kgdb** over serial (when you do want a debugger but only have UART), and the **oops decoder** workflow (`addr2line`, `scripts/decode_stacktrace.sh`).
>
> **Why:** JTAG is for bench work. This chapter covers what you can run on a deployed device with no debug header. You can't ship a fleet with a JTAG cable attached. You can ship a fleet with ftrace enabled. If a customer's device hangs once every three days, you need to know what the kernel was doing in the second before the freeze. Ftrace's persistent buffer plus the oops decoder answers that. EBPF lets you attach a probe to `tcp_retransmit_skb` on a production server and count retransmits per remote address, without recompiling the kernel. **JTAG:** the hardware debug scan chain used to halt, inspect, and single-step CPUs.
>
> **Focus:** match the tool to the symptom. Too much output in `dmesg`: use `dynamic_debug` to filter. "It worked once, now hangs": ftrace `function_graph` on the suspect subsystem. "What system calls is this app making?": a bpftrace one-liner. "Kernel oops on customer device": save dmesg and run decode_stacktrace.sh against the matching vmlinux. "I want to breakpoint and step a remote production kernel": kgdb over serial (rare, but sometimes the right call).
>
> **Tooling. Target:** `trace-cmd` (for ftrace), optional `bpfcc-tools` / `bpftrace` (eBPF, better on aarch64 / newer kernels). **Host:** `kernelshark` to visualise ftrace dumps. `crash(8)` for vmcore analysis. Ubuntu install: `apt install trace-cmd kernelshark bpfcc-tools bpftrace`. Buildroot: `BR2_PACKAGE_TRACE_CMD=y`, `BR2_PACKAGE_BCC=y`, `BR2_PACKAGE_BPFTRACE=y`. Full reference: Userspace tooling appendix. **MCU bridge:** Think of the rootfs as the firmware image's file-backed runtime environment. On an MCU you link everything into flash. On Linux, programs and config live in this

> mounted tree. **rootfs:** root filesystem, the directory tree mounted at / that contains /bin, /etc, /dev, and libraries. **Buildroot:** a configuration-driven build system that produces a complete root filesystem and related images.

119.1 printk

The kernel's `printk` is your first line of debug. Levels:

```
pr_emerg("system unusable\n");           /* KERN_EMERG, 0 */
pr_alert("action immediately\n");        /* KERN_ALERT, 1 */
pr_crit("critical conditions\n");        /* KERN_CRIT, 2 */
pr_err("error conditions\n");             /* KERN_ERR, 3 */
pr_warn("warning conditions\n");          /* KERN_WARNING, 4 */
pr_notice("normal but significant\n");    /* KERN_NOTICE, 5 */
pr_info("informational\n");               /* KERN_INFO, 6 */
pr_debug("debug-level\n");                /* KERN_DEBUG, 7 */
```

`dmesg -w` (follow) shows the live ring buffer. `dmesg -l err,warn` filters. The ring buffer is bounded (default ~128 KB. Configurable via `CONFIG_LOG_BUF_SHIFT`).

Console log level (which prints to console vs only-ring-buffer):

```
cat /proc/sys/kernel/printk
# 4   4   1   7   <-- current, default, min, default-console
echo 7 > /proc/sys/kernel/printk   # show DEBUG and lower on console
```

Or boot with `loglevel=7` cmdline.

`pr_debug` is worth knowing about: by default it compiles to nothing (zero-cost when off). Enable per-file via `dyndbg`:

```
# Enable all pr_debug in net/wireless/
echo 'file net/wireless/*.c +p' > /sys/kernel/debug/dynamic_debug/control

# Enable a single function
echo 'func nl80211_get_wiphy +p' > /sys/kernel/debug/dynamic_debug/control

# At boot via cmdline:
dyndbg="file drivers/net/ethernet/freescale/fec_main.c +p"
```

This enables existing debug prints in mainline drivers without rebuilding.

119.2 ftrace

`ftrace` lives in `/sys/kernel/tracing/`. It's a function call tracer that records every kernel function call (and optionally entry/exit pairs) with nanosecond timestamps into a ring buffer.

```
cd /sys/kernel/tracing
echo function > current_tracer        # trace every function call
echo 1 > tracing_on

# ... run your test workload ...

echo 0 > tracing_on
cat trace | head -50
# # tracer: function
# # entries-in-buffer/entries-written: 9994/9994 ...
# my_app-1024  [000] d... 12.345: __vfs_read <-vfs_read
# my_app-1024  [000] d... 12.346: ext4_file_read <-__vfs_read
# my_app-1024  [000] d... 12.347: ext4_buffered_read <-ext4_file_read
# ...
```

This is *every kernel function called by any process* during the trace window. The buffer fills fast (~MB/sec). Use filters:

```
echo ext4_* > set_ftrace_filter         # trace only ext4_ functions
echo my_app > set_ftrace_pid            # only trace one process
echo > set_ftrace_filter                # clear
```

function_graph, the call tree visualization

```
echo function_graph > current_tracer
echo ext4_file_read > set_graph_function
echo 1 > tracing_on
# ... workload ...
echo 0 > tracing_on
cat trace
#  3)               |  ext4_file_read() {
#  3)               |    generic_file_read_iter() {
#  3)               |      filemap_read() {
#  3)   1.234 us    |        ext4_buffered_read();
#  3)   5.678 us    |      }
#  3) ! 23.456 us   |    }
#  3) + 45.012 us   |  }
```

Indentation shows call depth. Duration per call (`us`). Markers (`!` = >100 µs, `+` = >10 µs) draw attention to slow paths. Useful for performance investigation.

Events, predefined tracepoints

The kernel ships hundreds of tracepoints (`/sys/kernel/tracing/events/`):

```
ls events/
# block  cpu_id  irq  kvm  net  sched  syscalls  tcp  workqueue  ...

# Enable sched_switch events
echo 1 > events/sched/sched_switch/enable
echo 1 > tracing_on
sleep 1
```

```
echo 0 > tracing_on
cat trace
# bash-1024 [000] d..3. 12.345: sched_switch: prev_comm=bash prev_pid=1024 ...
next_comm=kworker/0:1
# kworker/0:1-15 [000] d..3. 12.346: sched_switch: ...
```

Combined: “which processes ran in the last second + what kernel functions did they call” → use both `function_graph` + `sched/sched_switch` events.

trace-cmd + KernelShark

For larger traces and a GUI:

```
apt install trace-cmd kernelshark

# Record while a workload runs
trace-cmd record -e sched_switch -e block -p function_graph -g ext4_file_read myworkload
# (creates trace.dat)

# Open in GUI
kernelshark trace.dat
```

KernelShark gives a timeline-per-CPU view with function-graph trees and event flags overlaid. Useful for `why did this 1-second operation take 10 seconds`.

119.3 eBPF

eBPF lets you attach safe (verified) C-like programs to thousands of kernel hook points. `bpftrace` is the high-level DSL. `bcc` (Python+C) is the lower-level library.

```
apt install bpftrace

# Count TCP retransmits per remote IP
bpftrace -e 'kprobe:tcp_retransmit_skb { @retx[ntop(((struct sock*)arg0)-
>__sk_common.skc_daddr)] = count(); }'
# ^C
# @retx[192.168.1.5]: 12
# @retx[8.8.8.8]: 3

# Histogram of read sizes
bpftrace -e 'kprobe:vfs_read { @reads = hist(arg2); }' -c 'dd if=/dev/zero of=/tmp/x
bs=4096 count=1000'

# Trace every execve
bpftrace -e 'tracepoint:syscalls:sys_enter_execve { printf("%s %s\n", comm, str(args-
>filename)); }'
```

eBPF programs are production-safe. The in-kernel verifier rejects infinite loops, bad memory access, and anything that would crash the kernel. You can run them on a live customer device.

For embedded, i.MX6ULL is technically a Cortex-A7 (32-bit) and eBPF support on 32-bit ARM is limited. Better tooling on aarch64. The principle is the same. Consider arm64 SoCs for newer designs where eBPF is the primary debug tool.

119.4 kgdb, GDB over serial

When you do want full GDB on a deployed device but have no JTAG:

> **GDB:** the debugger. In cross-debugging it runs on the host while controlling code on the target.

```
# Build kernel with CONFIG_KGDB=y, CONFIG_KGDB_SERIAL_CONSOLE=y
# Boot with: kgdboc=ttymxc0,115200 kgdbwait

# Kernel pauses early in boot waiting for debugger
# On host:
arm-none-linux-gnueabihf-gdb vmlinux
(gdb) target remote /dev/ttyUSB0
(gdb) ... full GDB experience ...
(gdb) continue
```

Limitations:

- The console is taken. You can't `dmesg` from a serial terminal while kgdb owns it.
- A scheduled-out task can't be inspected (only the currently-running one + scheduled queues).
- Performance overhead, every breakpoint is a serial round-trip.

Most useful for: a kernel that hangs early-boot (you set `kgdbwait`). A deployed device with a specific reproducible bug. A CI test runner that can attach gdb on test failure.

119.5 Kernel oops

When the kernel hits an unhandled fault, it prints an "oops":

```
Unable to handle kernel NULL pointer dereference at virtual address 00000018
pgd = 80c54000
[00000018] *pgd=00000000
Internal error: Oops: 5 [#1] PREEMPT ARM
Modules linked in: my_driver(O)
CPU: 0 PID: 1234 Comm: my_app Not tainted 6.1.0-myimg #1
Hardware name: Freescale i.MX6 ULL (Device Tree)
PC is at my_driver_probe+0x24/0x100 [my_driver]
LR is at __platform_driver_probe+0x20/0x50
pc : [<7f000024>]    lr : [<8050a0b0>]    psr: 60000113
...
[<7f000024>] (my_driver_probe [my_driver]) from [<8050a0b0>]
```

```
(__platform_driver_probe+0x20/0x50)
[<8050a0b0>] (__platform_driver_probe) from [<8050a1a0>] ...
```

The stack trace addresses are virtual (kernel-VA mapped). To decode:

```
# Auto-decode via the kernel's helper
dmesg | scripts/decode_stacktrace.sh vmlinux /path/to/modules > oops.decoded

# Manual: addr2line on the kernel ELF
arm-none-linux-gnueabihf-addr2line -e vmlinux -f 0x8050a0b0
# __platform_driver_probe
# drivers/base/platform.c:583
```

For a module, the offset within the module file:

```
arm-none-linux-gnueabihf-addr2line -e my_driver.ko -f 0x24
# my_driver_probe
# /path/to/my_driver.c:42
```

That points to the exact source line. Run `git blame` on it to see which patch introduced the regression.

For the oops to be useful, you must have:

- `CONFIG_DEBUG_INFO=y` when building.
- The *exact* same kernel + modules ELFs that were running on the failing device.

Tip: always keep the build artifacts (`vmlinux`, `.ko` files with debug info) for every shipped build. Without them, oops decoding is impossible.

119.6 kdump, full crash dump

For really deep autopsies, `kdump` captures the entire kernel memory image after a crash:

```
Crashed kernel → kexec → small "capture kernel" boots → saves /proc/vmcore to disk
Reboot with normal kernel; analyze vmcore with crash(8) on host
```

```
crash vmlinux vmcore
crash> bt                     # backtrace at time of crash
crash> ps                     # all tasks at time of crash
crash> mod                    # loaded modules
crash> rd 0x80c00000 32       # read kernel memory
crash> log                    # dmesg
```

`crash` is RH’s tool. Takes some learning, but for oopses you can’t reproduce, it’s the right tool.

Embedded systems often lack the disk space for vmcore (200+ MB). Skip kdump and rely on ftrace + oops decoder.

119.7 Key Notes

> **Privilege boundary:** $ means normal user. # or sudo means root and can change host or target state. After a privileged command, verify the expected device, service, or file appears before continuing. Roll back by undoing the config change or stopping the service you just enabled.

- Software observability depends on code paths that still execute. Logs and trace buffers cannot replace hardware debug access when the required kernel infrastructure is unavailable.
- `dynamic_debug` selects existing messages, ftrace records execution and events, and kgdb stops execution. These interfaces answer different questions and impose different disturbance.
- Filters and bounded buffers determine which trace history survives. Missing or overwritten events limit conclusions about ordering and latency.
- An oops belongs to the exact kernel and module build that produced it. Archived `vmlinux`, module symbols, and load-address context connect the failure to source.
- kgdb shares the serial transport with other users. Console ownership and target pauses are part of the debug design.
- eBPF availability depends on the architecture and kernel configuration. kdump additionally consumes reserved RAM and dump storage, which can be restrictive on a small target.

119.8 Pitfalls

- **dmesg buffer wraps.** Default 128 KB. Verbose drivers eat it in seconds. Bump to 1 MB with `CONFIG_LOG_BUF_SHIFT=20`.
- **printk during fast path.** A printk in an IRQ context with `loglevel >= 4` blocks for 1+ ms (UART transmission). Don’t `pr_info` in hot paths.
- **ftrace overhead.** Function tracer adds ~50 ns per traced kernel call. Realistic kernel function rates under load on a Cortex-A7 are 1–10 M/s, so unfiltered tracing typically costs 5–10 % CPU. Always use `set_ftrace_filter` to scope.
- **ftrace buffer fills in seconds.** Default 1 KB per CPU. Bump to `echo 8192 > buffer_size_kb` for usable durations.

- **lost trace events.** When the buffer fills, oldest events drop. Check `cat /sys/kernel/tracing/per_cpu/cpu0/stats` for lost_events.
- **dynamic_debug requires CONFIG_DYNAMIC_DEBUG=y.** Most distros have it. Verify.
- **kgdb console conflict.** Once kgdb attaches, the same UART can't be used for dmesg. Use a second serial port or USB-serial.
- **bpftrace BTF requirement.** Modern bpftrace expects BTF info in vmlinux. Older kernels without `CONFIG_DEBUG_INFO_BTF=y` can't be probed by BTF-typed programs.
- **eBPF on 32-bit ARM.** Limited support. Many newer features are arm64-only. I.MX6ULL is 32-bit. Use ftrace/trace-cmd instead.
- **vmlinux without DEBUG_INFO.** Oops decoding fails silently, addresses can't be mapped to symbols. Build with `CONFIG_DEBUG_INFO=y`.
- **module addresses changing.** Each `insmod` chooses a different load address. Can't reuse oops decoder output across loads. Capture both oops and `/proc/modules` simultaneously.

119.9 Going deeper

- **`Documentation/trace/ftrace.rst`**: the canonical ftrace guide.
- **`Documentation/trace/events.rst`**: tracepoints + events.
- **`Documentation/dev-tools/kgdb.rst`**: kgdb docs.
- **Brendan Gregg's `bpftrace/bcc` tutorials**: http://brendangregg.com.
- **`scripts/decode_stacktrace.sh`** in kernel source.
- **Greg Kroah-Hartman's "Linux Kernel Driver" tutorials**: debugging chapters.
- **Steven Rostedt's ftrace papers**: the LWN series.
- **`crash(8)` man page + Red Hat docs**: for vmcore analysis.
- **Ch 118**: JTAG when none of the above suffice.
- **Ch 120**: user-space side.

> Next chapter: **Chapter 120: User-space debugging**, gdbserver, strace, perf, coredumpctl.

Chapter 120: User-space debugging

> **What:** the toolkit for debugging your **user-space applications** on the i.MX6ULL target from a host workstation. **gdbserver** + **gdb-multiarch** for breakpoint-and-step debugging across the network. **strace** for "what syscalls is this program making". **ltrace** for shared-library calls. **perf** for sampling profilers + hardware-counter-based analysis + flamegraphs. **core dumps** with `coredumpctl` for post-mortem analysis of crashed processes. **GDB:** the debugger. In cross-debugging it runs on the host while controlling code on the target.
>
> **Why:** Kernel debugging (Ch 118, 119) is less common in day-to-day work. Most of the time you're debugging applications. The pattern: target runs `gdbserver`. Host runs `gdb-multiarch` with the unstripped binary. You set breakpoints by source line, inspect variables, step through code, exactly as if developing locally. `strace` reveals "the open() is returning EACCES" before you've even opened gdb. `perf` answers "why is my video pipeline using 80 % CPU" with a flamegraph. Once these are set up, embedded app debug feels much like desktop debug.
>
> **Focus: gdbserver is the network agent (no debugger UI. Just exposes the process's debug API over TCP). Gdb-multiarch on the host knows ARM and connects. The unstripped ELF + sysroot give it symbols and headers**. For performance: `perf` is the universal sampling tool. Understand the difference between sampling (CPU%-style overview, low overhead) and tracing (every event, high overhead). For crashed programs: configure `coredumpctl` to save dumps to a known location, retrieve from the target, analyze on the host. **ELF:** Executable and Linkable Format, the standard Linux object and executable file format.
>
> **Tooling. Target:** `gdbserver`, `strace`, `ltrace`, `perf` (from `linux-tools`), optional `valgrind`. **Host:** `gdb-multiarch` (or your cross gdb), Brendan Gregg's `FlameGraph` scripts (`git clone https://github.com/brendangregg/FlameGraph`). Ubuntu install (target): `apt install gdbserver strace ltrace linux-tools-generic valgrind`. Buildroot: `BR2_PACKAGE_GDB=y` + `BR2_PACKAGE_GDB_SERVER=y`, `BR2_PACKAGE_STRACE=y`, `BR2_PACKAGE_LTRACE=y`, `BR2_PACKAGE_LINUX_TOOLS_PERF=y`. Full reference: Userspace tooling appendix. **MCU bridge:** Think of the rootfs as the firmware image's file-backed runtime environment. On an MCU you link everything into flash. On Linux, programs and config live in this mounted tree. **rootfs:** root filesystem, the directory tree mounted at / that contains /bin, /etc, /dev, and libraries. **Buildroot:** a configuration-driven build system that produces a complete root filesystem and related images.

120.1 Target side, install gdbserver

On a Buildroot/Yocto rootfs:

> **Yocto:** a metadata-driven build system for producing custom Linux distributions.

```
# Buildroot menuconfig:
#   Target packages → Debugging, profiling and benchmark → gdb → gdbserver

# Or via Debian/Ubuntu on the target
apt install gdbserver
```

`gdbserver` is small (~100 KB statically linked). No debug-info needed on the target.

120.2 Host side, gdb-multiarch + sysroot

```
apt install gdb-multiarch
```

`gdb-multiarch` is gdb with all architecture support compiled in (ARM, MIPS, RISC-V, ...). For a dedicated cross-debug:

```
apt install gdb-multiarch
# or:
arm-none-linux-gnueabihf-gdb        # part of the cross-toolchain
```

Tell GDB about your **sysroot**, the target's filesystem layout, so GDB can resolve symbols in shared libraries:

```
(gdb) set sysroot /path/to/target/rootfs
(gdb) set solib-search-path /path/to/target/rootfs/lib:/path/to/target/rootfs/usr/lib
```

120.3 Remote debug, gdbserver + gdb-multiarch

```
# On target:
gdbserver :2345 /path/to/myapp arg1 arg2
# Process myapp created; pid = 1234
# Listening on port 2345

# On host:
arm-none-linux-gnueabihf-gdb /path/to/build/myapp     # unstripped binary
(gdb) set sysroot /path/to/target/sysroot
(gdb) target remote 192.168.1.100:2345
(gdb) b main
(gdb) continue
Breakpoint 1, main (argc=3, argv=0x...) at myapp.c:42
42        printf("Hello, world!\n");
(gdb) n
(gdb) p argv[1]
$1 = 0x7fffd344 "arg1"
```

This is the bread-and-butter cross-debug workflow. You write code on your Linux host, cross-compile, copy to target (NFS or scp), gdbserver on target, attach from host. Loop time: 10 seconds.

> **NFS:** Network File System, which lets the target mount a host directory over Ethernet during development.

Important, `--multi` mode for re-launching:

```
gdbserver --multi :2345
# (gdb) target extended-remote 192.168.1.100:2345
# (gdb) set remote exec-file /path/to/myapp
# (gdb) run arg1 arg2
```

`--multi` keeps gdbserver alive across program runs, no need to restart on every test. Use during heavy iteration.

120.4 Attaching to a running process

```
gdbserver :2345 --attach <pid>
```

From host:

```
(gdb) target remote 192.168.1.100:2345
(gdb) bt           # see where in code the process is
(gdb) detach        # release without killing
```

Especially useful for hung daemons: attach, `bt`, `print global_state`, identify the deadlock, detach.

120.5 strace, trace syscalls

```
strace ./myapp
execve("./myapp", ["./myapp"], 0x7ffe... /* 14 vars */) = 0
brk(NULL)                               = 0x55c0000
arch_prctl(ARCH_SET_FS, 0x7fff...)   = 0
openat(AT_FDCWD, "/etc/myapp.conf", O_RDONLY) = 3
read(3, "key=value\n", 4096)         = 10
close(3)                                = 0
write(1, "Hello, world!\n", 14)      = 14
exit_group(0)                           = ?
+++ exited with 0 +++
```

Every syscall, its arguments, its return value. Useful for:

- “Why is `open()` failing?” → `strace -e openat ./myapp` shows the failed path + errno.

- “What files does this access?” → `strace -e file ./myapp`.
- “Slow startup?” → `strace -c ./myapp` shows time per syscall.
- “Hanging?” → `strace -p <pid>` shows what syscall it’s blocked in.

Options:

- `-f` follow forks (track child processes).
- `-e trace=open,read,write` filter to syscalls of interest.
- `-o file` save to file (don’t drown the terminal).
- `-T` show time per call.
- `-t` add timestamp.

For embedded, gdb-multiarch is a host tool. Strace runs on the target.

120.6 ltrace, same for library calls

`ltrace` shows shared-library function calls (libc, libpthread, libssl, your-libfoo). Less popular than strace but complementary:

```
ltrace ./myapp
__libc_start_main(0x401170, 1, 0x7fff..., 0x401200 ...
printf("Hello, %s!\n", "world")              = 14
malloc(64)                                    = 0x55c0080
free(0x55c0080)                               = <void>
+++ exited (status 0) +++
```

Use when “is this calling the right OpenSSL function” matters.

120.7 perf, sampling profiler and counters

`perf` is the Linux performance toolkit. Three main modes:

perf top, htop for CPU functions

```
perf top -p <pid>
# Real-time view of which functions are using CPU
#  35.4 %  myapp        compute_hash
#  12.1 %  myapp        memcpy
#   8.2 %  libc.so.6    malloc
```

Continuously updated, like `htop`, but functions instead of processes. Press `?` for help. Arrow keys to navigate. `Enter` to drill into a function’s assembly.

perf record + perf report, sampling profile

```
perf record -F 99 -g ./myapp           # sample at 99 Hz with call graphs
perf report
# Children      Self  Command  Shared Object  Symbol
# +   35.4%    35.4%  myapp    myapp          compute_hash
# +   12.1%    12.1%  myapp    myapp          memcpy
# ...
```

99 Hz sampling, not 100 Hz, avoids harmonics with kernel timers (which run at 100/250/1000 Hz). At 99 Hz you get about one sample per 10 ms with 0.1–0.5 % CPU overhead. Good for figuring out what an app is actually doing.

Flamegraphs

```
git clone https://github.com/brendangregg/FlameGraph
perf record -F 99 -g ./myapp
perf script | ./FlameGraph/stackcollapse-perf.pl | ./FlameGraph/flamegraph.pl > out.svg
# Open out.svg in a browser; interactive flame graph
```

A useful CPU-profile visualization. The x-axis is sample count (roughly time spent). The y-axis is the call stack. Click any block to zoom. Type to search.

Hardware counters

```
perf stat ./myapp
#       12345.67 msec task-clock                #    0.999 CPUs utilized
#                100      context-switches
#         1,234,567      cache-misses             #   12.34 % of all cache refs
#       567,890,123      instructions             #    1.23  insn per cycle
#       456,789,012      cycles
```

Counters tell you why something is slow: high cache-miss rate → memory bound. Low instructions-per-cycle → branch misprediction or stall.

For embedded:

- `perf` compiles for ARM cleanly.
- Hardware counters on i.MX6ULL Cortex-A7 are limited to a handful. High-end profiling is easier on Cortex-A53/A72.

120.8 Core dumps, post-mortem

When an app crashes:

```
# Enable on target
ulimit -c unlimited
echo /var/log/core/core.%e.%p > /proc/sys/kernel/core_pattern
```

```
./crashy_app
# Segmentation fault (core dumped)

ls /var/log/core/
# core.crashy_app.1234
```

Or with systemd-coredump:

```
# /etc/systemd/coredump.conf
[Coredump]
Storage=external
Compress=yes
ProcessSizeMax=2G
ExternalSizeMax=2G

coredumpctl list                    # see all recent cores
coredumpctl dump 1234 > /tmp/core   # extract one
```

Analyze on the host:

```
arm-none-linux-gnueabihf-gdb crashy_app /tmp/core
(gdb) bt
#0  0x000115a4 in crash_function () at crashy.c:42
#1  0x00011620 in main () at crashy.c:10
(gdb) p some_variable
(gdb) f 1
(gdb) info locals
```

You get the dying process's stack + register + memory state, debuggable as if it was alive.

120.9 Real-world workflow, debugging a hung app

Symptom: customer reports `myapp` "freezes" after ~1 hour.

```
# 1. Find the PID
ps aux | grep myapp
# user  1234  ...

# 2. Is it blocked on a syscall?
cat /proc/1234/wchan
# poll_schedule_timeout              # yep, in poll()
strace -p 1234
# poll([{fd=3, events=POLLIN}], 1, -1)    # blocked waiting for fd 3 forever
ls -la /proc/1234/fd/3
# lrwx... -> socket:[12345678]

# 3. What socket? netstat shows
ss -anp | grep 12345678
# tcp ... 192.168.1.100:5555  ESTAB  pid=1234,fd=3

# 4. Other end stopped responding? confirm via tcpdump
```

```
tcpdump -ni eth0 host 192.168.1.100

# 5. Get a backtrace to see WHERE in code it's waiting
gdbserver :2345 --attach 1234
# (on host)
(gdb) target remote ...
(gdb) bt
#0  0x... in poll () from /lib/libc.so.6
#1  0x... in wait_for_response () at myapp.c:127
#2  0x... in main_loop () at myapp.c:200
```

Now you know: line 127 calls poll() on a socket that's hung. Fix: add a timeout, handle disconnect, reconnect.

120.10 Key Notes

- `gdbserver` controls the target process while host GDB owns symbols and source lookup. The host executable and target libraries must describe the deployed build.
- The sysroot supplies the target's library context on the host. A wrong sysroot can make a plausible backtrace refer to the wrong implementation.
- `strace` exposes the process-to-kernel boundary, while `ltrace` exposes supported library calls. A syscall wait and an application deadlock are different diagnoses.
- Sampling identifies where execution time accumulates without recording every event. Counters and call graphs require workload and unwind context, not universal thresholds.
- A core dump preserves crash-time process state. Storage limits, permissions, and artifact retention determine whether that state remains usable.
- In the hung-socket case, an unbounded `poll()` wait is an application failure boundary. Timeouts and reconnect policy belong to the application, not the debugger.

120.11 Pitfalls

- **gdbserver and gdb-multiarch ABI mismatch.** Cross-compiler ARM ABI must match target's libc ABI (gnueabihf vs gnueabi). Different ABI = unable to set breakpoints in shared libraries.
- **Sysroot pointing to wrong path.** When `sysroot` points at the wrong path, gdb loads the host's `libc.so.6` from `/lib`. Symbols then mismatch. Always `set sysroot` before `target remote`.
- **Stripped binaries.** No symbols, no source-level debug. Build with `-g`. Copy unstripped to host. Ship stripped to target.

- **PIE binaries with ASLR.** Address space randomization makes addresses different each run. GDB handles it. Manual address arithmetic doesn't. Disable ASLR for repeatable debug: `setarch -R ./myapp`.
- **strace heavy slowdown.** Tracing a high-syscall-rate process can 10× slow it. Use `-e trace=read,write` to filter.
- **strace doesn't show shared library calls.** Use ltrace or gdb for that.
- **perf record -g with no CFI.** Without `-fno-omit-frame-pointer` in the build, perf can't unwind stacks. Compile with both `-g` and `-fno-omit-frame-pointer`.
- **perf hardware events unavailable.** Some VMs / containers / 32-bit ARM can't access perf counters. Falls back to software events only.
- **core dump truncated.** Default `ulimit -c` is often 0 (disabled). Set `ulimit -c unlimited`. Also kernel.core_pattern must allow writing somewhere.
- **systemd-coredump compresses cores.** `coredumpctl dump` decompresses. `xz -d` if you pulled directly.
- **gdb attach permission denied.** Need CAP_SYS_PTRACE (root) on target, or run as the same user as the target process. /proc/sys/kernel/yama/ptrace_scope = 0 to allow ptrace of any process. 1 is "only descendants" (default on many distros).
- **Detaching gdbserver leaves the process running.** Use `(gdb) detach` then exit. `(gdb) quit` without detaching kills the process, usually NOT what you want.

120.12 Going deeper

- **GDB manual** (https://sourceware.org/gdb/onlinedocs/), chapters on remote debugging.
- **`strace(1)`, `ltrace(1)`**: man pages.
- **Brendan Gregg's `perf` tutorial**: http://brendangregg.com/perf.html.
- **`Documentation/admin-guide/perf-security.rst`**: for kernel.perf_event_paranoid.
- **`Flame Graph` repo + paper**: http://brendangregg.com/flamegraphs.html.
- **`coredumpctl(1)`, `systemd-coredump.conf(5)`**.
- **Valgrind** (memcheck, callgrind), slower but more detailed. Runs on the target if you have enough RAM.
- **AddressSanitizer (ASan), UndefinedBehaviorSanitizer (UBSan)**: compile-time instrumentation. Catches memory bugs at runtime.
- **Ch 125A**: VSCode + gdbserver workflow for IDE users.

> Next chapter: **Chapter 120A: Mainline patch submission workflow** (inserted v1.2).

Chapter 120A: Mainline patch submission workflow

> **Driver choice:** Use the in-tree, maintained driver first. Use out-of-tree, spidev, or custom-driver paths only after you accept the kernel-version maintenance cost and document who owns updates.

> **What:** the **end-to-end workflow** for submitting a patch to the Linux kernel: `git format-patch`, `scripts/checkpatch.pl`, `scripts/get_maintainer.pl`, `git send-email`, the `b4` tool for series management, response etiquette (`Reviewed-by`, `Acked-by`, v2/v3 iteration), and the **Lore** archive for finding similar prior work. Worked on a real candidate patch, e.g., a YAML binding addition for a sensor used in your Cookbook chapter, or a one-line bug fix in the FEC driver.
>
> **Why:** If you write a driver that's useful, it can go upstream. Upstream-merged code gets security backports and API migrations for free. An out-of-tree fork is yours to maintain. The kernel community has strict and partly unwritten rules. A wrong commit-message format, an untested patch, a hostile reply to review, or a top-posted email is enough to get a patch silently dropped, no matter how good the code is.
>
> **Focus: the workflow is git-format-patch → checkpatch → get_maintainer → send-email → respond to review → v2 → repeat**. The cultural part is harder: be concise. One fix per patch. Explain *why* not just *what*. Never ignore review feedback (even if you disagree, respond). CC the right people but no spammy CC. Subject lines are `[PATCH] subsystem/file: short summary`. Lore.kernel.org is the public archive of every mailing-list discussion since ~1998. Always search it before sending. A "novel" fix may have been tried and rejected three times already, and the rejection threads tell you why.

120A.1 Decide what you're submitting

Categories, in order of acceptance ease:

Patch type	Likelihood of acceptance	Reviewer count
Trivial typo / comment fix	high	low

Patch type	Likelihood of acceptance	Reviewer count
Bug fix with clear reproducer	high	medium
YAML binding for an existing chip	high	medium (DT maintainers)
New driver for unsupported chip	medium	high (subsystem maintainer + experts)
New subsystem	very low	huge (the whole community)
Performance optimization	medium	high (must have benchmarks)
Cleanup / refactor	medium	medium (must justify churn)
Feature already done another way	very low	(rejected, see Lore)

For your first patch: **trivial typo, bug fix with reproducer, or YAML binding** is the sweet spot.

120A.2 Prerequisites, clone the right tree

For most subsystems, patch against either:

- **mainline** (`linux/torvalds.git`), for very recent code.
- The **subsystem maintainer's tree**, listed in `MAINTAINERS`. E.g., for NXP i.MX changes: Shawn Guo's `linux/arm/imx`. For networking: `netdev`. For staging: `staging-next`.

```
git clone https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git
cd linux
git checkout -b my-patch
```

120A.3 Make the change

> **Examples and production:** Do not burn fuses, enroll production keys, or sign release images while using these examples. Use throwaway keys and back up the unsigned image plus the key directory before testing irreversible security flows.

Same tools as any kernel patch, Edit, build, test:

```
make ARCH=arm CROSS_COMPILE=arm-none-linux-gnueabihf- imx_v7_defconfig
# Edit
make ARCH=arm CROSS_COMPILE=arm-none-linux-gnueabihf- zImage
# Test on hardware

git add -p
git commit
```

Commit message structure:

```
subsystem/file: short summary (max 70 chars, imperative mood)

Explanation of WHY this change is needed. Refer to the problem,
not the patch. Cover:
- What was broken / missing
- Why prior approaches don't work (if any)
- What user-visible effect this has
- Reference any tests / kernel versions affected

Add a Fixes: tag if this fixes a known regression:
Fixes: 0123456789abcdef ("original subject line")

Signed-off-by: Your Name <your@email.com>
```

The `Signed-off-by` is the **Developer Certificate of Origin**, you certify you have the right to submit this code under the kernel's license. `git commit -s` adds it automatically. Without it, the patch is rejected.

120A.4 checkpatch.pl, the style validator

```
git format-patch -1
# 0001-subsystem-fix-something.patch

./scripts/checkpatch.pl --strict 0001-*.patch
# total: 0 errors, 0 warnings, 0 checks, 25 lines checked
# 0001-... has no obvious style problems and is ready for submission.
```

checkpatch enforces:

- Line length ≤ 100 chars

- Tabs for indentation, not spaces
- C89 declarations (declare-at-top-of-block, modern kernels relaxed)
- No trailing whitespace
- `if (cond)` not `if( cond )`
- `goto out_unlock;` style for cleanup
- Many other things

`--strict` catches more (e.g., long lines that aren't strictly forbidden but discouraged). For your first patch: fix everything checkpatch reports, even warnings. Maintainers run checkpatch themselves. Clean output is professional.

120A.5 get_maintainer.pl, who to send to

```
./scripts/get_maintainer.pl 0001-*.patch
# Shawn Guo <shawnguo@kernel.org> (maintainer:ARM/FREESCALE IMX...)
# Sascha Hauer <s.hauer@pengutronix.de> (maintainer:ARM/FREESCALE IMX...)
# Pengutronix Kernel Team <kernel@pengutronix.de> (reviewer:ARM/FREESCALE IMX...)
# Fabio Estevam <festevam@gmail.com> (reviewer:ARM/FREESCALE IMX...)
# NXP Linux Team <linux-imx@nxp.com> (reviewer:ARM/FREESCALE IMX...)
# linux-arm-kernel@lists.infradead.org (moderated list:ARM/FREESCALE IMX...)
# linux-kernel@vger.kernel.org (open list)
```

The output is your To and CC list. Don't add extra CCs beyond what `get_maintainer.pl` returns. CC'ing every kernel address you can find gets you ignored or filtered.

120A.6 Send via git send-email

`git send-email` is the only patch-submission tool kernel maintainers accept. **Not** GitHub PRs, not email attachments, not pastebin links. Plain text email with the patch in the body.

```
# One-time config
git config --global sendemail.smtpserver smtp.gmail.com
git config --global sendemail.smtpserverport 587
git config --global sendemail.smtpencryption tls
git config --global sendemail.smtpuser your@gmail.com

# For Gmail, generate an App Password (with 2FA enabled) and use that

# Send
git send-email --to "Shawn Guo <shawnguo@kernel.org>" \
               --cc "Sascha Hauer <s.hauer@pengutronix.de>" \
               --cc linux-arm-kernel@lists.infradead.org \
               --cc linux-kernel@vger.kernel.org \
               0001-*.patch
```

Or to-from-file:

```
git send-email --to-cmd='./scripts/get_maintainer.pl --no-rolestats' --cc-cmd ... 0001-*.patch
```

`git send-email` is finicky to set up the first time (SMTP auth, App Passwords for Gmail/Outlook). Allow an hour for first-time setup.

120A.7 Multi-patch series + cover letter

For more than one patch:

```
git format-patch -3 --cover-letter
# 0000-cover-letter.patch
# 0001-foo-fix-bar.patch
# 0002-foo-add-baz.patch
# 0003-foo-update-bindings.patch
```

Edit `0000-cover-letter.patch` (the `*** SUBJECT HERE ***` and `*** BLURB HERE ***` placeholders):

```
Subject: [PATCH 0/3] foo: bug fix + new feature

This series fixes the X bug in foo (patch 1) by approach Y, then adds
support for the new Z chip (patches 2-3). Tested on hardware A and B.

Series structure:
  Patch 1: foo: fix X bug (the actual bug fix)
  Patch 2: foo: add support for Z chip (the driver code)
  Patch 3: dt-bindings: foo: add Z binding (the DT binding)

Patch 1 stands alone; patches 2-3 require patch 1 first.

Andy Doe (3):
  foo: fix X bug
  foo: add support for Z chip
  dt-bindings: foo: add Z binding

 ...
```

Cover letter sets the context maintainers need to triage. Send all patches together:

```
git send-email --to ... 0000-cover-letter.patch 0001-*.patch 0002-*.patch 0003-*.patch
# OR send the whole folder:
git send-email --to ... 00*.patch
```

120A.8 v2 / v3, revising based on feedback

When a maintainer responds with feedback, address each point. If you disagree, say so politely and explain why.

```
# Make changes
git commit --amend  # OR rebase + rework

git format-patch -1 -v 2 --in-reply-to=<msg-id-from-original-v1>
# 0001-subsystem-fix-something.patch (header now says "[PATCH v2]")

# Add a changelog at the bottom of the cover letter:
#   ---
#   Changes in v2:
#     - addressed Foo's feedback about Bar
#     - removed redundant baz check
git send-email --to ... 0001-v2-*.patch
```

The `---` separator in the commit body means “this text goes under the patch as a note, not part of the commit message.” Put your changelog there.

120A.9 Reviewed-by, Acked-by, Tested-by, Reported-by

In reply to your patch, others may say “Reviewed-by: …”, that means they read it carefully and approve. Include their tag in your v2:

```
Reviewed-by: Reviewer Name <reviewer@example.com>
Signed-off-by: Your Name <you@example.com>
```

Tags:

- **Signed-off-by**: required. Certifies origin.
- **Reviewed-by**: reviewer carefully read and approved.
- **Acked-by**: subsystem maintainer or expert agrees (lower bar than Reviewed-by).
- **Tested-by**: someone other than you tested it on hardware.
- **Reported-by**: someone reported the bug this fixes.
- **Suggested-by**: someone suggested this approach.
- **Co-developed-by**: pairs with Signed-off-by for joint authorship.

Tags accumulate over revisions. Carry forward all relevant ones.

120A.10 Replying to reviews, etiquette

```
On Mon, Mar 5, 2026 at 10:00 AM, Foo Maintainer wrote:
> > +    if (val < 0)
> > +        return -EINVAL;
> > +    val *= 2;
>
> Why is this multiplication needed? Comment seems lacking.

Apologies for the missing comment. The hardware register interprets
the value in 50% steps; multiplying by 2 converts user-supplied
percent to register units. I'll add a comment in v2.

Thanks for the review.
```

Rules:

- **Reply inline, below the quoted text** (not top-posted).
- **Trim quoted context**: don’t quote the whole patch back.
- **Plain text email**: no HTML, no signatures with images.
- **Address every comment**: either “fixed in v2,” “I disagree because …,” or “this is out of scope for this patch.”
- **Don’t take feedback personally**: Code review is the kernel’s quality bar. Criticism is professional, not personal.

If you go silent for more than two weeks after a review, your patch will be dropped from the maintainer’s queue. Stay engaged.

120A.11 Lore, search before you send

Lore.kernel.org is the public archive of every kernel mailing-list post since ~1998.

Before sending, search for:

- The file you’re touching (`drivers/net/ethernet/freescale/fec_main.c`)
- The function name
- The bug symptom

```
https://lore.kernel.org/search?q=fec_main.c+phy+probe
```

You may discover: someone already proposed your fix and it was rejected for a reason. Or there’s a parallel discussion you should join. Or the maintainer is mid-rework that supersedes your patch. Discovering this *before* sending saves embarrassment.

`b4` is a tool that simplifies series management:

```
pip install b4
b4 am https://lore.kernel.org/all/20260301...     # download a patch series locally
b4 prep -n my-series                                # start a new series for tracking
b4 send                                             # send via git send-email under the
hood
b4 trailers                                         # collect Reviewed-by/Tested-by from
replies
```

`b4` is increasingly standard in newer kernel work. Use it once you've sent your first patch the manual way.

120A.12 A worked example, adding a YAML binding for an existing chip

Suppose you wrote Ch 99's nRF24L01 user-space driver and want to upstream the *DT binding* (the actual kernel driver is out-of-tree but the binding is reusable):

```
# 1. Find the right path
ls Documentation/devicetree/bindings/net/wireless/
# atmel,*.yaml  marvell,*.yaml  ti,*.yaml  ...
# No nordic,nrf24*.yaml — opportunity.

# 2. Write the YAML
cat > Documentation/devicetree/bindings/net/wireless/nordic,nrf24l01p.yaml <<'EOF'
# SPDX-License-Identifier: (GPL-2.0-only OR BSD-2-Clause)
%YAML 1.2
---
$id: http://devicetree.org/schemas/net/wireless/nordic,nrf24l01p.yaml#
$schema: http://devicetree.org/meta-schemas/core.yaml#

title: Nordic nRF24L01+ 2.4 GHz radio

maintainers:
  - Your Name <you@example.com>

allOf:
  - $ref: spi-peripheral-props.yaml#

properties:
  compatible:
    enum:
      - nordic,nrf24l01p

  reg:
    maxItems: 1

  spi-max-frequency:
    maximum: 10000000

  ce-gpios:
    maxItems: 1
    description: Chip Enable pin (radio TX/RX activation)

  interrupts:
```

```
    maxItems: 1

required:
  - compatible
  - reg
  - ce-gpios
  - interrupts

unevaluatedProperties: false

examples:
  - |
    #include <dt-bindings/gpio/gpio.h>
    #include <dt-bindings/interrupt-controller/irq.h>

    spi {
        #address-cells = <1>;
        #size-cells = <0>;

        radio@0 {
            compatible = "nordic,nrf24l01p";
            reg = <0>;
            spi-max-frequency = <8000000>;
            ce-gpios = <&gpio4 27 GPIO_ACTIVE_HIGH>;
            interrupts-extended = <&gpio4 28 IRQ_TYPE_EDGE_FALLING>;
        };
    };
EOF

# 3. Validate the YAML
make ARCH=arm dt_binding_check DT_SCHEMA_FILES=Documentation/devicetree/bindings/net/
wireless/nordic,nrf24l01p.yaml

# 4. Commit
git add Documentation/devicetree/bindings/net/wireless/nordic,nrf24l01p.yaml
git commit -s -m "dt-bindings: net: wireless: add nordic,nrf24l01p

The nRF24L01+ is a 2.4 GHz GFSK radio commonly used in low-cost
mesh and remote-control products. This adds a DT binding for
SPI-connected modules, used by community out-of-tree drivers as
well as user-space spidev-based stacks.

Signed-off-by: Your Name <you@example.com>"

# 5. Generate patch
git format-patch -1

# 6. Check
./scripts/checkpatch.pl --strict 0001-*.patch

# 7. Find maintainers
./scripts/get_maintainer.pl 0001-*.patch
# (DT bindings → Rob Herring + devicetree@vger.kernel.org)

# 8. Send
git send-email --to "Rob Herring <robh@kernel.org>" \
               --cc "Krzysztof Kozlowski <krzk+dt@kernel.org>" \
               --cc devicetree@vger.kernel.org \
```

```
                --cc linux-kernel@vger.kernel.org \
                0001-*.patch
```

DT bindings are one of the easiest categories to get merged, they’re additive (don’t break anything) and self-contained, which makes them a good first-patch target.

120A.13 Key Notes

- A patch carries both a code change and its rationale. One logical change per patch makes the affected interface and regression risk visible.
- The target tree and subsystem maintainers define integration ownership. `MAINTAINERS`, prior discussions, and touched paths supply that context.
- `checkpatch` examines style, while binding checks examine schema contracts. Neither demonstrates device behavior or substitutes for accurately described test evidence.
- `Signed-off-by` records origin certification. Review, acknowledgement, and test trailers record different contributions and cannot be invented or treated as interchangeable.
- A series needs an explicit dependency order. Its cover letter explains the whole change, while individual commits remain understandable on their own.
- Revision changelogs and inline replies preserve interface decisions. A proposed binding is a review candidate, not a promise of acceptance or driver support.

120A.14 Pitfalls

- **HTML email.** Gmail’s web UI sends HTML by default. Mailing lists silently drop HTML patches. Use `git send-email` or plain-text mode.
- **Tabs vs spaces.** Some MUAs (Outlook, Thunderbird) helpfully convert tabs to spaces, mangling the patch. `git send-email` avoids this.
- **Reply-all to mailing lists.** When replying, hit Reply-all to keep the list CC’d. Off-list replies lose the discussion context.
- **Top-posting.** “I disagree, here’s why” *above* the quoted message reverses reading order. Reply inline.
- **Spam-quoting.** Quoting the entire patch + entire prior reply in your response → 500-line emails. Trim aggressively.
- **Patch in attachment.** Mailing-list filters drop attachments. Inline only.
- **Subject line lacks subsystem prefix.** “Fix bug” gets ignored. “drivers/net/foo: fix NULL deref” gets read.

- **Missing Signed-off-by.** Patch is silently dropped.
- **Tested-on: my Pi.** Maintainers don't care unless reproducible by them. State: kernel version, defconfig, test method.
- **Multi-changes in one patch.** "Fix bug + cleanup + add feature" → maintainer asks you to split. Always one logical change per patch.
- **Sending and disappearing.** Patch sent, maintainer asks question, you don't respond for a month → patch is dropped. Stay engaged.
- **Ignoring DCO.** Every patch must be `Signed-off-by` your real legal name. Pseudonyms or company aliases without legal-real-name are usually rejected.
- **Wrong tree.** Submitting a USB patch against the staging tree → maintainer redirects you to `usb` tree. Get_maintainer.pl tells you the right tree. Respect it.

120A.15 Going deeper

- **`Documentation/process/submitting-patches.rst`**: the canonical reference.
- **`Documentation/process/coding-style.rst`**: kernel C style.
- **`Documentation/process/email-clients.rst`**: how to configure every common MUA for kernel work.
- **`scripts/checkpatch.pl`, `scripts/get_maintainer.pl`**: read them. They're shell-helpers.
- **`b4` documentation**: https://b4.docs.kernel.org/.
- **Lore archive**: https://lore.kernel.org/.
- **LWN.net**: the periodical of kernel development. Read weekly to absorb culture.
- **Kernel Mentees / Outreachy**: community programs for first-time contributors.
- **Greg Kroah-Hartman's "How to send patches to the Linux kernel"**: old but still essential.
- **Ch 121**: capstone custom-board port (your most-likely upstreamable contribution).

> Next chapter: **Chapter 121: Capstone: custom board port**.

Chapter 121: Custom board port as a system design

> **NFS:** Network File System, which lets the target mount a host directory over Ethernet during development.

> **What:** a board port as an integration problem. A custom PCB or MINI variant changes assumptions shared by **U-Boot configuration**, **kernel Device Tree and drivers**, **rootfs contents**, and **image generation**. This chapter follows those dependencies and shows where each layer owns a decision. **MCU bridge:** Think of U-Boot like a much larger boot stub plus debug monitor: it initializes hardware, loads the next image, and gives you commands before Linux starts. **U-Boot:** the bootloader that initializes enough hardware to load and start the Linux kernel.
>
> **Why:** Each Cookbook chapter covered one piece. A board port is where those pieces have to agree: pin multiplexing (Ch 5), DDR initialization (Ch 14), U-Boot policy (Ch 22), kernel DT (Ch 27), and the peripherals actually fitted. Understanding that agreement makes generated code and configuration easier to assess. A successful build alone does not establish a working board. **DDR:** external DRAM that must be configured and trained before most software can run from it.
>
> **Focus: dependencies determine the bring-up order**. A console makes early failures observable, DDR supports larger images, and storage or networking supplies later boot inputs. Peripheral integration comes after those foundations. Recovery must remain independent of the candidate being evaluated. The order below is a design example, not a required reader project. **MCU bridge:** Think of JTAG like SWD debugging on Cortex-M: halt, read registers, set breakpoints. The Cortex-A path adds MMU state, privilege modes, and more complex reset behavior. **JTAG:** the hardware debug scan chain used to halt, inspect, and single-step CPUs.

121.1 Scope, what to port

The port spans several contracts:

Boundary	Design responsibility	Evidence needed separately
Early hardware	Power, reset, pin routing and DDR assumptions	Measurements and memory qualification on the fitted board

Boundary	Design responsibility	Evidence needed separately
Boot inputs	Storage layout, image selection and recovery	Actual loader behavior and artifact integrity
Peripheral integration	Supplies, clocks, buses, driver binding and user interface	End-to-end behavior, including failure and removal
Build and release	Pinned inputs, configuration and artifact provenance	Repeatable output and a qualified release candidate

An existing-driver chip such as an AT24 EEPROM or DS3231 RTC keeps the software contract relatively small. Camera, graphics and DMA-heavy additions widen the dependency set. That difference affects integration risk, not which assignment a reader must complete.

> **DMA:** Direct Memory Access. Hardware moves data to or from memory without the CPU copying each byte.

121.2 Hardware variants, what counts as “custom”

Three cases change different parts of the contract:

Case A, Software policy on unchanged MINI hardware

The physical board can stay unchanged while product policy differs:

- Different model string in U-Boot (board name in `bdinfo`)
- Different default hostname in Linux
- Custom default `bootargs`

These changes affect identity and boot policy, not the power tree or DDR wiring. Adding an external device moves the design into the next case.

Case B, A MINI variant with changed connections

Possible additions or changes include:

- Replace the WiFi module (if any) with a different one
- Add an external RTC chip + battery

- Add an SPI Ethernet (DM9051 from Ch 115)
- Mod the LCD interface

Changed wiring introduces new supply, pin-mux and Device Tree dependencies. Each addition needs its actual module specification and connection review. This is a design comparison, not a soldering instruction.

Case C, A full custom PCB

A custom board can change the early boot assumptions themselves:

- Different RAM (DDR3 256 MB vs the MINI's 512 MB? Different timing).
- Different PMIC.

> **MCU bridge:** Think of a PMIC like a programmable power-tree supervisor: it replaces discrete enables and LDO assumptions with sequenced rails the kernel can model. **PMIC:** Power Management IC, a chip that sequences and regulates the board's voltage rails.

- Different boot media (eMMC, QSPI flash instead of SD).
- Different I/O complement (your specific application, sensors, motors, displays).
- Different connectors / form factor.

The port must now account for hardware design and qualification as well as software integration. A similar SoC does not justify reusing another board's DDR, power or reset values. Detailed PCB design is outside this chapter's scope.

121.3 Bring-up as a dependency model

Power and reset must be valid before a ROM observation is meaningful. The ROM then needs the correct boot source and image format. Early software needs a usable stack, console and memory arrangement before later storage, network or peripheral behavior can be interpreted.

Those dependencies do not create a universal pass/fail list. Rail limits come from the actual components and board specification. DDR evidence depends on geometry, timings, operating conditions and the chosen test coverage, not a fixed one-hour run. Destructive memory tests must not overwrite the running image or its data.

Console output also depends on the selected image: the MINI ROM+DCD path in Chapters 19 and 22 has no SPL banner. Similarly, a PHY response, driver probe or bus acknowledgement establishes only part of an interface. The application path and recovery behavior remain separate.

The useful design model is a chain of prerequisites with an observation at each boundary. A symptom narrows the search, but rarely identifies one failed layer by itself.

121.4 U-Boot’s board contract

The bootloader does not discover every detail of a new PCB. Its configuration, board code and hardware description must agree with the fitted design. Chapters 19 and 22 distinguish the vendor MINI reference from the modern MINI candidate. An EVK configuration is a comparison, not a substitute for either.

121.4.1 Reference selection

A reference is useful when its assumptions are visible: source revision, CPU, memory geometry, power/reset arrangement, boot media and supported peripherals. A similar processor name is only the beginning of that comparison. Keep inherited features disabled until their connections and supporting drivers are understood.

121.4.2 Board integration

Build-system entries select which board implementation exists in the resulting image. Board hooks then own operations that cannot be expressed by generic drivers alone. The important design question is which component owns each initialization step, not how many files were copied.

121.4.3 Hardware description

The U-Boot hardware description must match the services needed before Linux: console, boot storage and any qualified policy inputs. Linux can need additional devices later. The two descriptions have different consumers, but cannot disagree about the physical wiring.

121.4.4 DDR handoff

Changed RAM changes the memory contract: geometry, clock, timing, pad settings and calibration belong to the fitted part and board. Chapter 14 explains those relationships. A tool’s generated values still need a documented input set and qualification.

The MINI ROM+DCD paths in Chapters 19 and 22 prepare DDR before U-Boot executes. An SPL-based platform assigns that job differently. These are alternative ownership models, not interchangeable places to paste a calibration table.

121.4.5 Candidate acceptance and recovery

Compilation establishes a software milestone. Image inspection establishes format and placement. A UART banner establishes observed execution. None represents a fixed percentage of completed bring-up.

Persistent installation is a separate acceptance decision. It needs an identified medium, a verified image layout, safe power conditions and a board-matched recovery path. Building a candidate must not automatically authorize writing it to a disk.

121.5 Kernel and boot-input agreement

Linux receives a kernel image, a hardware description and boot arguments. The root filesystem supplies the first process and its dependencies. A board port succeeds only when those independently produced inputs agree.

121.5.1 Describe the fitted hardware

A MINI variant starts from a description whose routing matches that variant. Additions and omissions follow the schematic, not the peripherals on an EVK. Pin control, clocks, supplies, reset lines and interrupt wiring form one connected device path.

121.5.2 Own the board artifact

Including a DTS in the build produces an artifact; it does not ensure that the bootloader selects that artifact. Model and compatible strings identify a design, while the release configuration owns which DTB is delivered and loaded.

121.5.3 Keep runtime dependencies matched

The kernel configuration decides which drivers exist and whether they are built in or loaded as modules. Rootfs assembly must include matching modules, firmware and userspace dependencies. A stale DTB or module tree can invalidate an otherwise successful kernel build.

121.5.4 Define the handoff

Load addresses must respect the selected image format and memory layout. Console selection must match the supported UART, and the root argument must identify the intended filesystem. Diagnostic output narrows the failure search; silence alone does not identify a bad DTB, DDR setting or console.

121.6 Custom peripheral, adding an external RTC

An attached DS3231 illustrates a small integration boundary. It is an optional device, not a stock MINI feature. Its supply, logic levels, pull-ups, backup circuit and chosen bus need a connection design before software description.

121.6.1 DT addition

Inside a qualified I2C controller node, a minimal child fragment can express the device’s identity and bus address:

```
rtc@68 {
    compatible = "maxim,ds3231";
    reg = <0x68>;
};
```

This is not a complete board DTS. It omits the parent controller, pin control and board-specific resources. The Linux v6.6 RTC binding defines the node’s properties. An interrupt or wakeup property needs actual wiring and a supported wake path; neither should be copied from another board.

121.6.2 Driver selection

The compatible string connects the node to a supported driver. The kernel configuration determines whether that implementation is present. Device Tree does not supply a missing driver or implement the RTC protocol.

121.6.3 Observation boundaries

Driver binding, readable time, valid backup state, alarm behavior and system-clock policy are different observations. Registration as an RTC device does not qualify all of them. Device numbering also does not identify which RTC the product should trust at startup.

121.7 Reproducible artifacts, separate deployment

A build entry point is useful when it exposes its inputs and produces reviewable artifacts. Combining source checkout, compilation and destructive disk writes in one script hides an important approval boundary.

Layer	Inputs it owns	Output consumed elsewhere
Bootloader build	Pinned source, board configuration, toolchain and image format	Loader candidate
Kernel build	Pinned source, kernel configuration and board description	Kernel, DTB and matching modules
Rootfs build	Package selection, libraries, firmware and service policy	Root tree and image
Image assembly	Partition layout, placement rules and artifact identities	Candidate media image
Deployment	Identified physical medium and recovery policy	Installed candidate

Reproducibility needs controlled sources, tools, configuration and other build inputs. A script name is not that evidence. The two project-local Arm toolchains and explicit environment selection remain the policy from Chapter 3.

Image assembly can produce a file for inspection without touching a device. Deployment is intentionally separate: a correct build can still be written to the wrong medium, at the wrong offset, or before recovery is available.

121.8 Failure boundaries and recovery

No usable early output

Power/reset, boot selection, image format, memory initialization and console configuration are possible boundaries. The ROM’s DCD is a register-write structure, not a producer of a UART transcript. Preserve observations rather than calling every garbled character a DDR diagnosis.

U-Boot runs, but Linux does not reach userspace

A prompt narrows the search without qualifying all RAM. The next boundary includes kernel format, load ranges, DTB, handoff state and console selection. Later failures add rootfs access, PID 1 and its runtime dependencies.

Linux runs, but a peripheral path fails

A peripheral depends on routing, power, clocks, reset, controller support and driver binding. For MINI Ethernet, retain the actual carrier/PHY qualification limits from Part III rather than treating EVK networking as board evidence.

Description and implementation do not meet

A missing node, disabled node, unmatched compatible or absent driver can prevent binding. A driver may also defer while waiting for a supply or clock provider. A log entry is evidence to interpret, not a universal diagnosis.

A candidate cannot start

Recovery must not depend on the failing candidate executing correctly. Chapter 8 distinguishes ROM USB access from later U-Boot services. The recovery image and media layout must match the board; a stock EVK card is not automatically a MINI fallback.

121.9 Key Notes

- A board port connects fitted hardware to several software descriptions. Power and reset behavior, early DDR setup, U-Boot configuration, kernel DT, and rootfs contents have distinct owners.
- The dependency chain starts before Linux. Unusable rails, reset, or DDR can prevent any kernel diagnostic, while a serial banner establishes only partial progress.
- Device Tree describes wiring and resources, while drivers implement behavior. A matching `compatible` string alone does not establish correct clocks, supplies, interrupts, or pin mux.
- The external RTC case separates board description, driver selection, and userspace access. Its presence and wiring are additions, not stock MINI assumptions.
- Reproducible image construction needs pinned inputs and matching modules and libraries. Writing an image to a particular medium is a separate, destructive deployment operation.

- Recovery depends on a board-specific known-good image and accessible boot path. Compilation and illustrative output do not qualify a custom board or the MINI.

121.10 Design history

A useful bring-up record connects a decision to its assumptions, observed evidence and remaining uncertainty. It explains why a clock, memory setting or recovery policy was accepted, not merely which command was typed.

That history lets a later engineer distinguish a deliberate constraint from an inherited default. A compiler upgrade, changed component or new carrier revision can then reopen the relevant decision without repeating every earlier investigation.

121.11 Pitfalls

- **Copying a reference without its assumptions.** Configuration and DTS reuse need a hardware and source-version comparison.
- **Treating one observation as full qualification.** A banner, probe message or successful build establishes only a particular milestone.
- **Hiding deployment inside the build.** Artifact production and permission to write a physical medium are separate responsibilities.
- **Reusing DDR values after hardware changes.** Timing, geometry and calibration need evidence for the fitted design and operating conditions.
- **Shipping mismatched artifacts.** The kernel, DTB, modules, firmware and rootfs must belong to a consistent release.
- **Depending on the candidate for recovery.** Keep a documented board-specific route that remains usable when normal boot fails.

121.12 Going deeper

- Chapter 14: DDR timing, initialization ownership and qualification limits.
- Chapters 19 and 22: vendor reference versus modern MINI candidate.
- Chapters 25-27: kernel artifacts, boot inputs and Device Tree.
- Chapters 31-35: init, libraries and rootfs construction.
- Chapter 121A: build evidence and controlled hardware integration in CI.
- Chapter 122A: preserving hardware contracts during BSP migration.
- Chapter 120A: upstream maintenance and review.

> Next chapter: **Chapter 121A: CI/CD for embedded Linux**.

Chapter 121A: CI/CD for embedded Linux

What: continuous integration for embedded Linux. The pieces:

- Build U-Boot, kernel, and rootfs on every commit. **MCU bridge:** Think of the rootfs as the firmware image's file-backed runtime environment. On an MCU you link everything into flash. On Linux, programs and config live in this mounted tree. **MCU bridge:** Think of U-Boot like a much larger boot stub plus debug monitor: it initializes hardware, loads the next image, and gives you commands before Linux starts. **rootfs:** root filesystem, the directory tree mounted at / that contains /bin, /etc, /dev, and libraries. **U-Boot:** the bootloader that initializes enough hardware to load and start the Linux kernel.
- Run smoke tests on **real hardware** in a board farm via a self-hosted CI runner with USB-OTG flashing.
- Use a **Labgrid**-style test harness.
- Signal pass/fail back to the PR.

We use **GitHub Actions** (or **GitLab CI**) with a self-hosted runner that has a USB connection to a Point Atom MINI. On every push, the runner does the Ch 121 build and flashes via `uuu`. It then watches serial for the `=>` prompt, runs a sysfs check, captures the log, and marks the PR pass or fail. **sysfs:** a kernel-generated filesystem under /sys that exposes devices, drivers, and attributes.

Why: any embedded product shipping updates from more than one developer benefits from CI. The risk is real: someone merges a DT change that breaks boot, nobody notices until a customer tries to update, and you spend days firefighting. With CI plus real-hardware smoke tests on every PR, that bug surfaces in ten minutes. The setup cost is small (a dev board, a Linux host, and a few hours) and pays back quickly.

Focus: a normal cloud CI runner has no USB connection to your board. To run hardware tests, self-host a runner on a Linux box that physically owns the board. GitHub Actions / GitLab CI register the self-hosted runner. The runner does cross-builds, then drives the board via uuu plus a serial terminal scripted in Python. A smoke test is small (boot, get prompt, run a few checks, capture the log) but catches most regressions. Scale from one board to a farm of 10 via Labgrid (RPC framework for board control).

121A.1 What “CI” means for embedded

Traditional cloud CI (Travis, Circle, GitHub Actions hosted runners):

- Runs in a VM with no hardware access.
- Can cross-compile your kernel + check it builds.
- Can run unit tests on x86 (QEMU is option).
- **Cannot** verify the binary works on real silicon.

For embedded, building cleanly isn’t enough. The real value of CI is **catching regressions on actual hardware**, a DT change that compiles but breaks boot, an MMC driver edit that boots but corrupts the rootfs, a regulator change that boots but burns more power.

The architecture:

```
   PR opened
        │
        ▼
   GitHub Actions (cloud runner): cross-build U-Boot + kernel
        │
        ▼
   Artifact upload (kernel.zImage, dtb, u-boot.imx, rootfs.tar)
        │
        ▼
   Self-hosted runner (your Linux box with USB to a board):
        │ - download artifacts
        │ - flash board via uuu
        │ - power-cycle board (via USB-power-switch GPIO)
        │ - capture serial; wait for "=>"
        │ - in U-Boot, load kernel + boot
        │ - wait for shell prompt
        │ - run sysfs check (e.g., ls /sys/class/net/eth0 should exist)
        │ - capture full serial log
        │ - report pass/fail
        ▼
   PR status updated
```

121A.2 GitHub Actions, the cross-build half

`.github/workflows/build.yml`:

```
name: build
on:
  push:
    branches: [main]
  pull_request:
    branches: [main]

jobs:
```

```
  cross-build:
    runs-on: ubuntu-22.04
    steps:
      - uses: actions/checkout@v4
        with:
          submodules: recursive

      - name: Install cross toolchain
        run: |
          sudo apt-get update
          sudo apt-get install -y bison flex bc libssl-dev \
            device-tree-compiler u-boot-tools curl xz-utils
          test -n "$ARM_GNU_TOOLCHAIN_URL"
          curl -L "$ARM_GNU_TOOLCHAIN_URL" -o /tmp/arm-gnu-toolchain.tar.xz
          mkdir -p "$RUNNER_TEMP/arm-gnu-toolchain"
          tar -xf /tmp/arm-gnu-toolchain.tar.xz -C "$RUNNER_TEMP/arm-gnu-toolchain"
          TOOLCHAIN_DIR=$(find "$RUNNER_TEMP/arm-gnu-toolchain" -maxdepth 1 -type d \
            -name 'arm-gnu-toolchain-*-x86_64-arm-none-linux-gnueabihf' | sort | tail -1)
          ln -sfn "$TOOLCHAIN_DIR" "$RUNNER_TEMP/arm-gnu-toolchain/current"
          echo "$RUNNER_TEMP/arm-gnu-toolchain/current/bin" >> "$GITHUB_PATH"

      - name: Build U-Boot
        run: |
          cd u-boot
          make ARCH=arm CROSS_COMPILE=arm-none-linux-gnueabihf- myboard_defconfig
          make ARCH=arm CROSS_COMPILE=arm-none-linux-gnueabihf- -j$(nproc)

      - name: Build kernel
        run: |
          cd linux
          make ARCH=arm CROSS_COMPILE=arm-none-linux-gnueabihf- imx_v7_defconfig
          make ARCH=arm CROSS_COMPILE=arm-none-linux-gnueabihf- -j$(nproc) zImage dtbs

      - name: Upload artifacts
        uses: actions/upload-artifact@v4
        with:
          name: board-images
          path: |
            u-boot/u-boot-dtb.imx
            linux/arch/arm/boot/zImage
            linux/arch/arm/boot/dts/nxp/imx/imx6ull-myboard.dtb
```

Now every PR gets a clean build. Failure = compile error caught.

121A.3 The self-hosted runner, the hardware half

On your Linux box that physically owns a Point Atom MINI:

```
# 1. Install runner
mkdir actions-runner && cd actions-runner
curl -O -L https://github.com/actions/runner/releases/download/v2.310.0/actions-runner-
linux-x64-2.310.0.tar.gz
tar xzf actions-runner-linux-x64-2.310.0.tar.gz

# 2. Register (token from GitHub → repo settings → Actions → Runners)
./config.sh --url https://github.com/myuser/myrepo --token AABBCCDDEEFF
```

```
# 3. Run (or systemd-install)
./run.sh
# Connected to GitHub
# 2026-05-31 12:00:00Z: Listening for Jobs
```

Now add a hardware-test job:

```
  hardware-smoke:
    needs: cross-build
    runs-on: [self-hosted, imx6ull]
    steps:
      - name: Download artifacts
        uses: actions/download-artifact@v4
        with:
          name: board-images

      - name: Flash via uuu
        run: |
          # Power off board (via USB power-control GPIO on the runner)
          gpioset gpiochip0 18=0
          sleep 2

          # Set boot mode pins to USB-SDP (also via GPIO)
          gpioset gpiochip0 19=1

          # Power on
          gpioset gpiochip0 18=1
          sleep 1

          # uuu sees the board enter SDP; loads U-Boot to OCRAM, runs it
          uuu -b emmc u-boot-dtb.imx

          # Now uuu pushes kernel + rootfs
          uuu -b emmc_all u-boot-dtb.imx zImage imx6ull-myboard.dtb rootfs.tar

      - name: Smoke test
        run: |
          python3 smoke_test.py /dev/ttyUSB0 --timeout 60

      - name: Upload serial log
        if: always()
        uses: actions/upload-artifact@v4
        with:
          name: serial-log
          path: serial.log
```

The runner is labelled `imx6ull`. Only jobs targeting that label run on this hardware. You can have multiple labelled runners (one per board type).

121A.4 The smoke test script

```
#!/usr/bin/env python3
"""Smoke test: power-cycle target, watch serial for boot completion + checks."""
import serial, time, sys, argparse
```

```
def expect(port, pattern, timeout):
    deadline = time.time() + timeout
    buf = ''
    while time.time() < deadline:
        chunk = port.read(port.in_waiting or 1).decode('utf-8', errors='replace')
        buf += chunk
        sys.stdout.write(chunk)
        sys.stdout.flush()
        if pattern in buf:
            return buf
    raise TimeoutError(f"Did not see {pattern!r} within {timeout}s")

def send(port, line):
    port.write((line + '\r\n').encode())
    port.flush()

def main():
    ap = argparse.ArgumentParser()
    ap.add_argument('device')
    ap.add_argument('--timeout', type=int, default=60)
    args = ap.parse_args()

    p = serial.Serial(args.device, 115200, timeout=1)

    # Watch for U-Boot prompt
    expect(p, '=>', args.timeout)
    print("\n[OK] U-Boot prompt detected")

    # Boot kernel
    send(p, 'boot')

    # Watch for login prompt (BusyBox-typical)
    expect(p, 'login:', 60)
    print("\n[OK] Kernel booted to login")

    # Auto-login
    send(p, 'root')
    expect(p, '# ', 10)
    print("\n[OK] Logged in")

    # Sanity checks
    send(p, 'ls /sys/class/net/eth0')
    expect(p, '# ', 5)
    send(p, 'echo $?')
    rc = expect(p, '# ', 5)
    assert '0' in rc.split('\n')[-2], "eth0 not present"
    print("[OK] eth0 sysfs present")

    send(p, 'cat /proc/meminfo | head -3')
    expect(p, '# ', 5)

    send(p, 'i2cdetect -y 0 2>&1')
    expect(p, '# ', 10)

    # Power off cleanly
    send(p, 'poweroff')
    print("[OK] All checks passed")
```

```
if __name__ == '__main__':
    main()
```

Run on the self-hosted runner. Outputs all of the serial conversation. Exit 0 on success, non-zero on any failure → PR gets a red X.

121A.5 Labgrid, for board farms

When you have 5+ boards or want richer control (toggle pins, video capture, multimeter readings), **Labgrid** (Pengutronix) is the answer. It's a Python framework + RPC server that owns the hardware.

```
pip install labgrid

# Config: /etc/labgrid/places.yaml
places:
  imx6ull-1:
    drivers:
      - DigitalOutputPowerDriver:
          name: power
          gpio: gpio18-out
      - SerialDriver:
          name: serial
          port: /dev/ttyUSB0
      - SDMuxDriver:
          name: sdmux
          gpio: gpio20-out

# In tests
client = labgrid.Client()
target = client.acquire('imx6ull-1')
target.power.on()
target.sdmux.switch_to_target()
target.serial.expect('=>')
```

Multiple test runners can share the same board farm. Labgrid handles locking. For larger teams, this is the production-grade setup.

121A.6 Test artifact storage

A board boot log is ~50 KB. Build outputs are ~150 MB. You generate 10–50 builds/day. Plan storage:

- **GitHub Actions artifact retention**: 90 days default. Configurable.
- **Self-hosted artifact server** (e.g., minio S3-compatible): infinite retention.
- **A binary cache** (e.g., `sccache` for builds): speeds up re-builds.

For long-term: stash every shipped build's `vmlinux` and `.ko` files in S3, keyed by git-sha. When a customer reports an oops, you can decode it (Ch 119).

121A.7 Caching for fast cross-builds

Cross-builds are slow (~10–30 min). Speed up:

- **ccache**: caches compiler output keyed by source hash. 80 % cache hit = 5× speed-up.

```
- uses: actions/cache@v3
  with:
    path: ~/.ccache
    key: ccache-${{ matrix.target }}-${{ github.sha }}
    restore-keys: ccache-${{ matrix.target }}-
```

- **Module dependencies cache**: kernel `*.cmd` files + `.o` files. `make` is incremental if these survive.
- **Toolchain cache**: download the cross compiler once. Cache the result.

121A.8 Trigger patterns

Don’t run the full pipeline on every commit:

```
on:
  push:
    branches: [main]                # always test main
    paths-ignore:
      - '**/*.md'                   # skip docs-only changes
  pull_request:
    branches: [main]
    paths:
      - '!**/*.md'
  schedule:
    - cron: '0 6 * * *'             # daily build (catches dependency drift)
  workflow_dispatch:                 # manual trigger via UI
```

Tag pushes can trigger the **release** workflow (build + sign + upload a downloadable image to your customer portal).

121A.9 Notifications

When the board farm is offline or builds fail repeatedly:

- **Slack webhook**: `${{ secrets.SLACK_WEBHOOK }}` posts to a channel on failure.
- **Email**: GitHub does this by default for failed Actions.
- **Pager**: PagerDuty integration for production-blocking failures.

```
- name: Notify on failure
  if: failure()
```

```
  run: |
    curl -X POST $SLACK_WEBHOOK_URL \
      -H 'Content-Type: application/json' \
      -d "{\"text\":\":x: Build failed: ${{ github.event.pull_request.title }} ($
{{ github.run_id }})\"}"
```

121A.10 Key Notes

- Cloud runners construct cross-built artifacts and check source. A hardware runner owns the physical board and provides a different kind of evidence.
- The build-to-test handoff must identify matching bootloader, kernel, DTB, and rootfs artifacts. A successful build does not establish that those artifacts boot together.
- Board control requires exclusive ownership of power, boot selection, storage, and serial access. Labgrid or equivalent locking prevents concurrent jobs from changing the same target.
- A serial prompt or sysfs entry is a bounded smoke-test observation, not full functional, power, or fault-recovery qualification.
- Self-hosted runners are a trust boundary because submitted code can reach attached hardware and host resources. Untrusted contributions cannot inherit unrestricted board or signing access.
- Caches shorten builds but do not replace pinned inputs. Retained logs and exact debug artifacts connect a failed test or field report to its build.

121A.11 Pitfalls

- **Self-hosted runner security.** A runner with checkout permissions can run arbitrary PR code. Don’t allow forks to trigger your hardware runner without manual approval. `pull_request_target` runs the workflow with write permissions on the target repo, which a forked PR can abuse.
- **USB instability.** Long USB cables drop intermittently. Uuu fails randomly. Use short, shielded cables. Powered hubs.
- **uuu version drift.** New SoCs need new uuu versions. Pin to a known-good.
- **Serial port conflicts.** Two tests grabbing /dev/ttyUSB0 simultaneously = chaos. Labgrid handles locking. Ad-hoc scripts need flock.
- **Board “stuck on” between tests.** If a previous test crashed the board, it may not respond to flash. Always hard-power-cycle at test start.

- **Time-of-day cron.** `cron: '0 6 * * *'` is UTC. Off-by-time-zone embarrassment. Comment your timezone assumption.
- **Caches stale.** ccache occasionally returns wrong objects when toolchain changes. Invalidate cache on toolchain bump.
- **Test pollution.** Test #1 leaves the board with the wrong network config. Test #2 fails for unrelated reasons. Always flash a fresh image.
- **Storage exhausted.** Build artifacts add up. GitHub limits to 500 MB per repo. Purge old artifacts.
- **Workflow YAML errors.** Subtle indentation or quoting bugs in YAML cause "workflow failed before starting" with no useful error. Validate with `actionlint`.

121A.12 Going deeper

- **GitHub Actions documentation**: https://docs.github.com/actions.
- **GitLab CI docs**: similar concepts, different syntax.
- **Labgrid**: https://labgrid.org/.
- **uuu (Universal Update Utility)**: NXP's MFGTOOL successor. Https://github.com/nxp-imx/mfgtools.
- **`usbrelay`**: controls cheap USB relay boards for power-cycling.
- **`actionlint`**: GitHub Actions YAML validator.
- **Pengutronix's "Labgrid" talks on YouTube**: board-farm architecture in production.
- **LAVA (Linaro Automated Validation Architecture)**: the Linaro-style board farm. Complex but powerful.
- **Ch 121**: the build script that CI calls.
- **Ch 125**: for over-the-air updates that CI builds and signs.

> Next chapter: **Chapter 122: Build your own cross-toolchain**, bootstrapping gcc + glibc/musl from sources.

Chapter 122: Build your own cross-toolchain

> **ELF:** Executable and Linkable Format, the standard Linux object and executable file format.

> **What:** building a complete **cross-compiling toolchain**, binutils + gcc + glibc (or musl/uClibc-ng) + gdb, from upstream sources, using **crosstool-NG** (the canonical tool) and, as a one-time example, by hand. We resolve the bootstrap puzzle: “gcc needs libc to compile programs, libc needs gcc to compile itself, gcc needs binutils, …”. The output: an `arm-none-linux-gnueabihf-*` toolchain in `/opt/x-tools/`. We compare it size-for-size and behavior-for-behavior against a pre-built Linaro / Bootlin / Yocto SDK toolchain. **Yocto:** a metadata-driven build system for producing custom Linux distributions.
>
> **Why:** for most users, the Arm GNU Toolchain installed in Chapter 3 is fine. Build your own when one of these matters:
>
> 1. **Pinning**, the apt version updates with Ubuntu. Your build might silently change behavior. Your own toolchain is reproducible across teams and time.
> 2. **Custom libc / configuration**, you need `glibc` 2.34 specifically. Or you want musl for size. Or you want a hardened gcc with stack protector defaults.
> 3. **Understanding**, every “weird linker error” makes sense once you’ve built the linker. It also teaches what every flag and stage actually does.
>
> **Focus:** a multi-stage build solves the chicken-and-egg problem. Stage 1 gcc has no libc and can only compile freestanding code. It builds the kernel headers, then glibc. Stage 2 gcc is then built against the new glibc and has a full C++/pthread runtime. Each stage knows where to find the previous. The directories, prefix, and `--with-sysroot` flags must agree. Get any of these wrong and the linker can’t find libc, or gcc looks in `/usr/lib` instead of the cross sysroot. Crosstool-NG hides this complexity in menuconfig + sequenced builds.

122.1 What’s in a toolchain

A cross-toolchain has 4 parts:

Component	What it does	Source
binutils	assembler (`as`), linker (`ld`), object tools (`objcopy`, `objdump`, `nm`, `readelf`, `strip`, `ar`)	https://ftp.gnu.org/gnu/binutils
gcc	the C/C++ compiler driver + frontends + middle-end + backend	https://ftp.gnu.org/gnu/gcc
libc	C runtime library + dynamic linker (glibc, musl, uClibc-ng, newlib)	https://ftp.gnu.org/gnu/glibc, https://musl.libc.org
gdb	the debugger (host-side; gdbserver runs on target)	https://ftp.gnu.org/gnu/gdb

Plus optional:

- **gcc support libraries**: libgcc, libstdc++, libgfortran, libgomp, libitm, libsanitizer.
- **kernel headers**: the userspace-visible kernel API headers (`asm/`, `linux/`, `mtd/`), needed for glibc to compile syscall wrappers.
- **linker scripts, sysroot layout, multilib variants**.

The full prefix-namespace convention:

```
/opt/x-tools/arm-unknown-linux-gnueabihf/
├── bin/
│   ├── arm-unknown-linux-gnueabihf-gcc      ← compiler driver
│   ├── arm-unknown-linux-gnueabihf-ld
│   ├── arm-unknown-linux-gnueabihf-objdump
│   └── ...
├── arm-unknown-linux-gnueabihf/             ← "target" subtree
│   ├── lib/        crt*.o, libc.so, libpthread.so
│   ├── sys-root/
│   │   ├── usr/include/    headers
│   │   ├── usr/lib/         libraries
│   │   └── lib/             dynamic linker
│   └── ...
├── lib/
│   └── gcc/arm-unknown-linux-gnueabihf/<gcc-ver>/
│       ├── libgcc.a
│       ├── include/         (gcc's own headers — stddef.h, stdarg.h)
│       └── ...
└── share/
```

The “triple” `arm-unknown-linux-gnueabihf` encodes: arch (arm), vendor (unknown, generic), OS (linux), ABI (gnueabihf = GNU C library + ARM EABI + hard-float).

122.2 The bootstrap problem

You want to build gcc. Gcc needs glibc to compile programs. Glibc needs gcc to compile itself. Circular.

Resolution, **multi-stage build**:

```
   1. binutils (assembler + linker)                    ┐
                                                       │ Self-contained
   2. Kernel headers (NOT compiled — just installed    │ — no libc needed
       from linux source)                              ┘

   3. gcc "stage 1" (a.k.a. bootstrap gcc):
      - configured with --disable-shared, --disable-libssp, etc.
      - links nothing, just compiles
      - has libgcc.a (the gcc support library, freestanding)
      - this gcc can compile glibc's source but cannot link an app

   4. glibc:
      - configured to use stage-1 gcc + binutils + kernel headers
      - builds libc.so, libpthread.so, crt0.o, etc.

   5. gcc "stage 2" (the final gcc):
      - configured with --enable-shared, --enable-threads, etc.
      - linked against the freshly-built glibc
      - now full C++ runtime, full pthreads, full sanitizers
```

Each stage produces artifacts the next stage consumes. Crosstool-NG sequences this automatically.

122.3 crosstool-NG, the standard tool

```
# Get crosstool-NG
git clone https://github.com/crosstool-ng/crosstool-ng.git
cd crosstool-ng
./bootstrap                  # autoconf/automake; one-time
./configure --enable-local   # build in current dir
make
PATH=$PWD:$PATH              # use local ct-ng

# Configuration
mkdir ../crosstool-build && cd ../crosstool-build
ct-ng menuconfig
```

The menuconfig is structured similarly to kernel menuconfig:

```
   Target options
     → Target Architecture: arm
```

```
      → Endianness: Little endian
      → Bitness: 32 bit
      → Architecture level: armv7-a
      → CPU: cortex-a7
      → Tune: cortex-a7
      → Floating point: hardware (FPU)
      → FPU: vfpv3-d16

    Toolchain options
      → Tuple's vendor string: yourvendor
      → Tuple's alias: arm-yourvendor-linux-gnueabihf

    Operating System
      → Target OS: linux
      → Linux kernel version: 6.6.x (the headers version)

    C compiler
      → gcc version: 13.2.0
      → C++ + Fortran on/off
      → Enable Link Time Optimization

    C library
      → glibc 2.38   (or musl, or uClibc-ng)

    Binutils
      → binutils version: 2.41

    Debug facilities
      → gdb: enabled (for cross-debug)
      → ltrace, strace: enabled
```

Save the config. Build:

```
ct-ng build
# Downloads sources (binutils, gcc, glibc, kernel, gdb)
# Builds binutils
# Installs kernel headers
# Builds gcc stage 1
# Builds glibc
# Builds gcc stage 2
# Builds gdb cross
# Installs to ~/x-tools/arm-yourvendor-linux-gnueabihf/
# Takes 30–90 minutes on a fast machine; longer on a Pi
```

After:

```
export PATH=~/x-tools/arm-yourvendor-linux-gnueabihf/bin:$PATH
arm-yourvendor-linux-gnueabihf-gcc --version
# gcc (crosstool-NG 1.27.0) 13.2.0
arm-yourvendor-linux-gnueabihf-gcc -o hello hello.c
file hello
# hello: ELF 32-bit LSB pie executable, ARM, EABI5 version 1 (SYSV), dynamically linked,
interpreter /lib/ld-linux-armhf.so.3, ...
```

Your own toolchain, pinned to specific versions, reproducible.

122.4 Manual mini-build, for the masochist (and the educated)

For one-time understanding, build it by hand. Compressed (a full tutorial is 30+ pages. Here's the skeleton):

```
TARGET=arm-mine-linux-gnueabihf
PREFIX=/opt/x-tools/$TARGET
SYSROOT=$PREFIX/$TARGET/sysroot
JOBS=$(nproc)

mkdir -p $SYSROOT
export PATH=$PREFIX/bin:$PATH

# 1. binutils
tar xf binutils-2.41.tar.xz
cd binutils-2.41
mkdir build && cd build
../configure --prefix=$PREFIX --target=$TARGET --with-sysroot=$SYSROOT \
             --disable-nls --disable-werror
make -j$JOBS && make install
cd ../..

# 2. Kernel headers (no compilation)
tar xf linux-6.6.tar.xz
cd linux-6.6
make ARCH=arm INSTALL_HDR_PATH=$SYSROOT/usr headers_install
cd ..

# 3. gcc stage 1 (compiler only, no libc yet)
tar xf gcc-13.2.0.tar.xz
cd gcc-13.2.0
./contrib/download_prerequisites           # downloads mpfr, gmp, mpc
mkdir build && cd build
../configure --prefix=$PREFIX --target=$TARGET --with-sysroot=$SYSROOT \
             --disable-multilib --disable-shared --disable-threads \
             --disable-libatomic --disable-libgomp --disable-libquadmath \
             --disable-libssp --disable-libvtv --disable-libstdcxx \
             --enable-languages=c
make -j$JOBS all-gcc all-target-libgcc
make install-gcc install-target-libgcc
cd ../..

# 4. glibc
tar xf glibc-2.38.tar.xz
cd glibc-2.38
mkdir build && cd build
../configure --prefix=/usr --host=$TARGET --target=$TARGET \
             --with-headers=$SYSROOT/usr/include \
             --disable-multilib --disable-werror libc_cv_forced_unwind=yes
make -j$JOBS
make install DESTDIR=$SYSROOT
cd ../..

# 5. gcc stage 2 (full C/C++ with glibc)
cd gcc-13.2.0/build
rm -rf *
```

```
../configure --prefix=$PREFIX --target=$TARGET --with-sysroot=$SYSROOT \
             --disable-multilib --enable-shared --enable-threads=posix \
             --enable-languages=c,c++
make -j$JOBS
make install
cd ../..

# Verify
$TARGET-gcc -o hello hello.c
file hello
```

Each `configure` line is a tour of GNU autoconf flags. The `--with-sysroot=$SYSROOT` is the key glue: gcc looks here for `usr/include/` (headers) and `usr/lib/` (libraries). Get the path wrong and gcc happily uses your host's headers, your binaries get built but link against the wrong libc.

122.5 glibc vs musl, the C library choice

	glibc	musl	uClibc-ng	newlib
Size (static helloworld)	800 KB	8 KB	30 KB	30 KB
Compatibility	extreme	high (mostly POSIX)	medium	bare-metal/ embedded
Performance	best (NPTL threads, vectorized memcpy, …)	OK	OK	n/a
License	LGPL	MIT	LGPL	BSD
dlopen support	yes	yes	yes	partial
Locale + i18n	full	limited	limited	minimal
Use case	desktop, server, full systems	static linking, containers, small Linux	older embedded	bare-metal RTOS

Pick guide:

- **glibc**: default for any system that has > 64 MB RAM and runs a real distribution.

- **musl**: when you statically link (Alpine-style images, single-binary apps) or want < 50 MB rootfs.

> **MCU bridge:** Think of the rootfs as the firmware image's file-backed runtime environment. On an MCU you link everything into flash. On Linux, programs and config live in this mounted tree. **rootfs:** root filesystem, the directory tree mounted at / that contains /bin, /etc, /dev, and libraries.

- **uClibc-ng**: legacy embedded. Mostly being replaced by musl.
- **newlib**: bare-metal Cortex-M. Not for Linux.

The toolchain is built around the libc choice. You can't easily swap later. Pick at the start.

122.6 ABI and the "hf" suffix

`arm-none-linux-gnueabihf` decoded:

- **arm**: 32-bit ARM
- **linux**: Linux kernel target
- **gnueabi**: GNU C library + ARM EABI calling convention
- **hf**: hardware float, uses FPU registers for argument passing in floating-point calls

Vs `arm-linux-gnueabi` (soft-float, passes float args in integer registers. Works on any ARM but slower):

- **hf binaries cannot run on soft-float systems** and vice versa.
- The dynamic linker name differs: `/lib/ld-linux-armhf.so.3` vs `/lib/ld-linux.so.3`.
- The i.MX6ULL Cortex-A7 has VFPv3 (hardware float). Use **hf**.

`gcc -mfloat-abi=hard` is required at compile time. The toolchain has it as the default.

122.7 Multilib, multiple variants in one toolchain

A single gcc can produce binaries for multiple ABIs (e.g., both soft-float and hard-float, both armv7-a and armv5te). Enable with `--enable-multilib`. Costs: longer build time, more disk space.

For most embedded projects: pick one ABI. Disable multilib for clarity. Crosstool-NG's `--disable-multilib` is the default.

122.8 Sysroot vs prefix vs target

These three concepts confuse first-time toolchain builders:

- **prefix**: where the toolchain binaries (gcc, ld, etc.) install: `/opt/x-tools/<triple>/`
- **target**: the system the toolchain produces code for: `arm-none-linux-gnueabihf`
- **sysroot**: the target's logical "/", where gcc/ld look for target headers and libraries. Often `<prefix>/<target>/sysroot/`

When compiling for the target:

```
gcc                 reads .c source
  → gcc preprocessor   includes <stdio.h> from sysroot/usr/include/
  → cc1               compiles to assembly
  → as                assembles to .o
  → ld                links against sysroot/usr/lib/libc.so
```

A "sysroot" is essentially the target filesystem layout, available on the host for cross-build. Buildroot, Yocto, and even crosstool-NG produce one.

> **Buildroot:** a configuration-driven build system that produces a complete root filesystem and related images.

For application development, you often want the sysroot to also contain libraries from your rootfs (libcurl, libssl, ...). Buildroot's "Per-package directories" + SDK export gives this.

122.9 Comparing your toolchain to pre-built ones

```
# Yours
$TARGET-gcc --version

# Linaro
wget https://snapshots.linaro.org/.../gcc-linaro-...-linux-gnueabihf.tar.xz

# Bootlin
wget https://toolchains.bootlin.com/.../arm-buildroot-linux-gnueabihf.tar.bz2

# Yocto SDK (after building an image; produces a self-extracting installer)
./poky-glibc-x86_64-core-image-...-toolchain-....sh
```

Compare:

- **Binary sizes** (size of "hello world" with each)
- **Default options** (`gcc -dumpspecs` shows the spec file)

- **glibc version**: `strings $SYSROOT/lib/libc.so.6 | grep version`
- **Compatibility**: does code built with yours run on the target built with theirs?

In practice they're all similar (all build from the same upstream). Differences:

- Yocto/Buildroot SDKs are tied to a specific rootfs. Yours is generic.
- Linaro toolchains often include experimental optimizations.
- Your own toolchain has your chosen versions + flags.

122.10 Key Notes

- A toolchain combines code-generation tools, target runtime components, and a debugger. Its target ABI and libraries must agree with the deployed rootfs.
- Bootstrapping resolves a dependency cycle in stages: binutils and kernel headers precede a limited compiler, then libc and the final compiler runtime.
- The prefix locates host tools, the target tuple names the output architecture and ABI, and the sysroot supplies target headers and libraries. These paths serve different purposes.
- Hard-float calling conventions affect binary interfaces, not just available instructions. Startup objects, libraries, and the dynamic loader must share the ABI.
- Changing libc or enabling multilib changes compatibility and maintenance scope. A smaller sample binary alone does not establish application compatibility.
- The custom-build sketches explain construction rather than assign a replacement installation. The book's selected Arm prebuilt toolchains remain the reference environment.

122.11 Pitfalls

> **Driver choice:** Use the in-tree, maintained driver first. Use out-of-tree, spidev, or custom-driver paths only after you accept the kernel-version maintenance cost and document who owns updates.

- **Build out of order.** Build glibc before stage 1 gcc → fails. Always sequence binutils → headers → gcc1 → glibc → gcc2.
- **Wrong sysroot path.** gcc looks in `/usr/include` instead of `$SYSROOT/usr/include`. Uses host headers. ABI mismatch on target. Verify `gcc -print-sysroot`.

- **Missing kernel headers.** glibc build complains about missing `linux/...` headers. Install them first via `make headers_install`.
- **Host-tool versions too old.** Building gcc 13 needs autoconf 2.69+, makeinfo, modern bison/flex. Use Ubuntu 22.04+.
- **gcc bootstrap requires C++ compiler.** Stage 1 gcc compiles with the host gcc. Host must support C++14+.
- **Out-of-tree build required.** Most GNU sources don't support in-source builds. Always `mkdir build && cd build && ../configure ...`.
- **Static glibc.** Linking statically against glibc bloats binaries. Static linking glibc *also* breaks dlopen and getaddrinfo. Use musl for static.
- **Multiple toolchains in PATH.** Two `arm-none-linux-gnueabihf-gcc` binaries → wrong one picked. Always explicit full path or curate PATH.
- **PIE vs PIC defaults.** Modern gcc defaults to `-fPIE -pie`. Older code expecting non-PIE may break (especially if your kernel doesn't support PIE binaries, which is rare).
- **Locale ate the build.** glibc builds depend on `LC_ALL=C`. If your locale is set weirdly, configure scripts misbehave.
- **`-march=armv7-a` vs `-mcpu=cortex-a7` confusion.** Both work. Cortex-a7 is more specific (enables errata). Pick one and stay consistent.
- **Cross-gdb missing python.** crosstool-NG's gdb may lack Python support → no `lx-symbols`. Enable `--with-python` for gdb.

122.12 Going deeper

- **`crosstool-NG documentation`**: `docs/`. Well-written.
- **Bootlin's "Toolchains for Embedded Linux"** training material, gold-standard.
- **`Linux From Scratch` (LFS)**: manual-build instructions. Not embedded but the bootstrap chapter is the canonical reference.
- **GNU binutils, gcc, glibc manuals**: long but the truth.
- **musl libc documentation**: much shorter than glibc, good intro.
- **Yocto's `meta-toolchain` recipe**: for understanding how Yocto produces SDKs.
- **Bootlin's pre-built toolchains** (https://toolchains.bootlin.com), to compare against your own.
- **GCC's "internals" docs**: for deep understanding of stages, passes, register allocation.

- **Ch 6**: the original toolchain chapter (consumer-side).
- **Ch 123**: Yocto vs Buildroot. Both use cross-toolchains.

> Next chapter: **Chapter 122A: BSP → mainline migration playbook**. **BSP:** Board Support Package: vendor patches, configs, bootloader files, and scripts needed to boot one board.

Chapter 122A: BSP → mainline migration playbook

> **PHY:** physical-layer block or chip that converts digital MAC signals to electrical or radio signals. **PWM:** Pulse-Width Modulation, a timer output whose duty cycle controls average power or encodes timing. **MCU bridge:** Think of Linux PWM like an MCU timer output channel, except the driver exposes period, duty cycle, polarity, and enable state through a subsystem. **IIO:** Industrial I/O, Linux's subsystem for sensors, ADCs, DACs, and buffered sampled data. **ASoC:** ALSA System-on-Chip, the embedded audio layer that connects CPU audio ports, codecs, and board wiring.

> **What:** the **systematic procedure** for taking an inherited vendor BSP (NXP `linux-imx 4.1.15`, ST's `stm32mp1 4.19`, TI's `ti-linux-5.10`, ...) and moving the product to a **mainline** Linux kernel that's supportable for the product's lifetime. We cover the patch inventory + classification, the subsystem-by-subsystem migration order (the safest path through the dependency graph), the test-coverage strategy, the parallel-tree maintenance during the migration, the upstreaming of recoverable bits, and the "do not migrate" decision criterion.
>
> **Why:** you join a project. The existing kernel is Linux 4.1.15 from NXP's 2017 BSP. The product ships for 8 more years. Security CVEs accumulate weekly. Mainline is at 6.6+. And the BSP is *frozen*, no upstream updates because the vendor moved on. This is a project-defining decision. This chapter is the playbook for getting it right.
>
> **Focus: classify every vendor patch into mainline-merged (delete), mainline-equivalent (replace), still-needed (forward-port), or vendor-only (decide individually). Migrate subsystem-by-subsystem in dependency order. Maintain BOTH trees in parallel during the transition. CI on both**. The hardest part isn't technical. `git rebase` handles most of the code work. The hardest part is cultural, convincing management that six months of kernel work, with no new features, pays back over the product's lifetime.

122A.1 Why migration is hard

A typical 2017-era NXP BSP:

- 4.1.15 mainline + **~7,000 NXP patches**
- ~1,500 of which are now in mainline (rebased, refactored, or rewritten)
- ~2,000 are obsolete (fix bugs in code mainline rewrote)

- ~2,000 are still-needed (drivers for chips mainline doesn't support)
- ~1,500 are NXP-internal-only (vendor APIs, downstream UI hacks, broken HW workarounds)

Plus there's a toolchain problem. Gcc 6.x won't compile a mainline 6.6 kernel cleanly. The kernel's hard minimum is gcc 5.1 (see `Documentation/process/changes.rst`), but newer features need gcc 11+. Most distros ship gcc 12+ for embedded cross-builds.

- Old U-Boot 2017.04 doesn't speak modern FIT.

> **MCU bridge:** Think of U-Boot like a much larger boot stub plus debug monitor: it initializes hardware, loads the next image, and gives you commands before Linux starts. **FIT:** Flattened Image Tree, U-Boot's container format for kernels, DTBs, initramfs images, hashes, and signatures. **U-Boot:** the bootloader that initializes enough hardware to load and start the Linux kernel.

- Old Buildroot/Yocto recipes pinned to old library versions.

> **Yocto:** a metadata-driven build system for producing custom Linux distributions. **Buildroot:** a configuration-driven build system that produces a complete root filesystem and related images.

- Old GStreamer 1.10 doesn't speak to mainline V4L2 drivers, which now use newer APIs.

Staying on 4.1.15 means accumulated CVEs go unfixed:

- CVE-2017-1000405 (the "Huge Dirty COW" / Dirty COW THP race): patched in 4.1.51, missed in your 4.1.15.
- CVE-2018-3639 (Spectre v4): no fix backported to your fork.
- CVE-2021-4034 (Polkit pwnkit): irrelevant to kernel but kernel-related stack.
- 2024+ CVEs: countless.

Migrating to mainline 6.6+ LTS buys:

- 6 years of security maintenance from kernel.org.
- Modern features (eBPF, io_uring, PREEMPT_RT, USB4, …).

> **PREEMPT_RT:** the Linux real-time patch set that makes more kernel paths preemptible and reduces latency.

- Predictable LTS cycle.
- Upstream-able fixes.

For products with > 2 years of remaining ship life, **migrate**. For products in last-12-months, **monitor CVEs and backport critical fixes manually**.

122A.2 Inventory phase, classify every patch

```
# In the BSP kernel
git log --oneline v4.1.15..HEAD > patches.txt
wc -l patches.txt
# 7234 patches
```

Categorize each. A spreadsheet (or simple script):

```
patch_id | subject | files_touched | category
0123abcd | Add support for SXX sensor | drivers/iio/light/sxx.c | NEW DRIVER (still
needed if SXX is on product)
0124abcd | Fix race in FEC driver | drivers/net/ethernet/freescale/fec_main.c | BUG FIX
(check if in mainline)
0125abcd | Add NXP-specific GStreamer plugin support | gst/ | VENDOR API (decide)
0126abcd | Backport: net/tcp fix from 4.4 | net/tcp.c | OBSOLETE (mainline already has
it)
0127abcd | Add IMX support for HDMI bridge | drivers/gpu/drm/bridge/ | NEW DRIVER
(probably in mainline by now)
```

For each “still-needed” or “new driver,” check mainline:

```
cd /tmp/linux-mainline
git log --grep "SXX" --oneline
git log --diff-filter=A --all -- drivers/iio/light/sxx.c
```

If mainline has it already (someone committed the same fix after 2017), delete the BSP patch and use the mainline version.

If not: this is your **upstreaming opportunity**, extract the patch, clean it, submit to mainline (Ch 120A) so the next person doesn’t have to.

After inventory:

```
2000 mainline-merged: delete
2000 obsolete: delete
1500 vendor-only: triage (keep/delete/upstream)
1500 still-needed: forward-port (write into 6.6 mainline)
```

Now you have ~3000 patches to *port forward*, not 7000.

122A.3 Pin the BSP, set up the migration "stable" branch

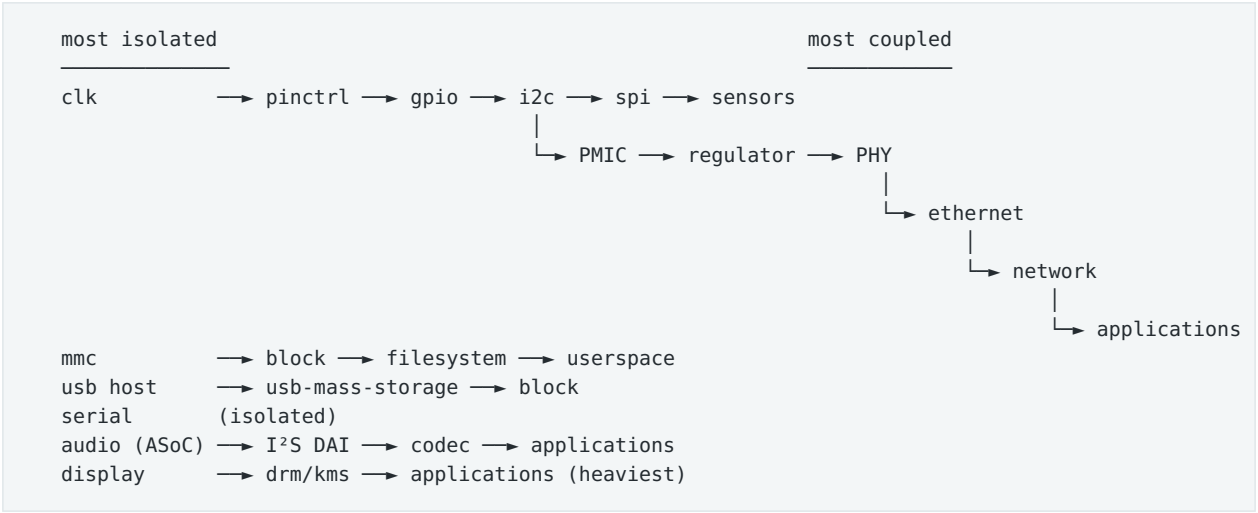
Before migration, freeze BSP development:

- Branch: `bsp-frozen-2026q2` from current BSP HEAD.
- Only critical security backports and customer-blocking bugs land here.
- All new feature work goes on `mainline-6.6-port` branch.

This buys you 6+ months to migrate while still shipping fixes to existing customers.

122A.4 Order of attack, subsystem dependency graph

Migrate subsystems in dependency order, **most-isolated first**. The graph:



This graph expresses dependencies, not a universal six-month schedule. Clock, pin-control and supply providers must exist before their consumers can work. Storage supplies later filesystem access, and a working controller is only one part of a complete peripheral path.

Migration order follows the fitted hardware and the product's required boot, application and recovery paths. Optional media devices can wait when they are outside those paths. Application packages also need compatible libraries and interfaces. Risk comes from the actual BSP divergence and missing evidence, not a fixed number of weeks assigned to each subsystem.

122A.5 Per-subsystem migration

For each subsystem:

1. **Identify divergence**: `git diff bsp-frozen..mainline-6.6 -- drivers/foo/`. What's different?
2. **Determine which patches still apply**: try `git cherry-pick` each. Many trivially conflict. Mainline has refactored the file.
3. **For each conflict, decide**: (a) BSP patch is dead, code mainline replaced it. Drop. (b) BSP patch is still needed, port to mainline's new structure. (c) Mainline is missing functionality, extract to a separate patch, upstream.
4. **Test on hardware**: Each subsystem migrated must pass its tests before moving on.

Tools that help:

- **`git blame`** on mainline file to understand the new structure.
- **`scripts/get_maintainer.pl`** to find a mainline maintainer who knows the subsystem (ask them if your BSP patch's intent is now solved differently).
- **`Documentation/devicetree/bindings/`** to see how the DT contract changed.

122A.6 The "pinned driver" problem

> **Driver choice:** Use the in-tree, maintained driver first. Use out-of-tree, spidev, or custom-driver paths only after you accept the kernel-version maintenance cost and document who owns updates.

Some vendor drivers are pinned to BSP-API versions and rewriting them is hard. Examples:

- Vivante GPU's userspace blob expects vendor's GLES driver, which talks to vendor's kernel DRM driver. Mainline has `etnaviv` (reverse-engineered), but vendor app may not support etnaviv's slightly different EGL config.
- VPU (Video Processing Unit) for hardware H.264 decode: vendor proprietary blob. Mainline `coda` driver covers some chips.
- ISP (Image Signal Processor): vendor blob. Little mainline support.

For each "pinned" component, options:

- **Replace with mainline equivalent + accept feature reduction** (etnaviv for Vivante).
- **Run vendor driver as out-of-tree module** + accept the maintenance cost.

- **Switch to a different SoC** in your next product revision where mainline has full support.

The first option is best long-term. The second is a stopgap. The third is the strategic decision.

122A.7 Maintaining two trees in parallel

During the 6-month migration:

- `bsp-frozen-2026q2` ships to all current customers.
- `mainline-6.6-port` is what engineers work on.
- Backport critical bug fixes from BSP → mainline-port (small patches, easy).
- *Don't* port mainline-port features back to BSP (one-way migration).

CI on both:

- Cross-build both kernels on every commit.
- Run hardware smoke tests on both (need two boards or a board with a runtime-switchable boot image).

Plan the cutover. Pick a date. After that date, new customer shipments use the mainline-ported kernel. Existing customers can choose to update or stay on BSP.

122A.8 When NOT to migrate

If any of these is true: **don't migrate. Manage CVEs manually on the BSP**:

- Product is end-of-life within 12 months.
- Critical hardware silicon has no mainline driver (and you can't write one in 6 months).
- Vendor SDK / app stack only works with BSP version (and rewriting the app is out of scope).
- Customer regulatory certification requires the specific kernel version + binary build (re-cert costs more than CVE backports).

In these cases: have someone watch CVE feeds (oss-security, kernel.org). Backport critical fixes manually. Document the deviation from upstream for compliance.

122A.9 Worked example, NXP iMX6ULL 4.1.15 → 6.6 mainline

This is the canonical migration for this book's target SoC.

```
# 1. Set up workspaces
mkdir migrate-imx6ull && cd migrate-imx6ull
git clone https://github.com/nxp-imx/linux-imx.git nxp-bsp
cd nxp-bsp && git checkout imx_4.1.15_2.0.0_ga && cd ..
git clone https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git mainline
cd mainline && git checkout v6.6 && cd ..

# 2. Get the BSP patch list
cd nxp-bsp
git log --oneline v4.1.15..HEAD | wc -l       # ~ 7000 patches
git format-patch v4.1.15..HEAD -o ../bsp-patches/
cd ..

# 3. Quick wins — mainline already supports i.MX6ULL well as of 6.6
# - clk, pinctrl, gpio, i2c, spi, mmc, usb host: all in mainline
# - FEC: mainline driver, full feature parity
# - audio (SAI + WM8960): mainline driver
# - eLCDIF: mainline DRM driver (imx-lcdif)
# - CSI (camera): partial mainline support (imx-csi)
# - PWM, RTC (SNVS), watchdog (WDOG): all mainline

# 4. What probably needs porting from BSP
# - Specific board DTS additions (your customer's variants)
# - Out-of-tree drivers (e.g., RFM69, custom IIO sensors)
# - User-space driver agents (some IPC mechanism)

# 5. Test
cd mainline
make ARCH=arm CROSS_COMPILE=arm-none-linux-gnueabihf- imx_v7_defconfig
make ARCH=arm CROSS_COMPILE=arm-none-linux-gnueabihf- -j$(nproc) zImage dtbs
# Boot from existing rootfs first; later update rootfs to a libc ABI matched to gcc 13

# 6. Iterate per subsystem
# Verify Ethernet works → check.
# Verify USB host enumerates a USB stick → check.
# Verify the specific sensor your product has → port the driver if needed.

# 7. Cut over
# Update CI to test mainline as primary.
# Issue product firmware update.
# Deprecate the BSP branch.
```

For i.MX6ULL specifically: mainline support is *excellent* as of 6.6, most BSP work was upstreamed during 2018–2023. Many migrations end up with ~50 forward-ported patches (board DTS + a couple of out-of-tree drivers). Expect months of work, not years.

122A.10 Key Notes

- Migration preserves required product behavior across changing interfaces, rather than replaying every vendor patch onto a newer tree.
- Inventory separates already-merged fixes, obsolete changes, forward-port candidates, and vendor-only contracts. A patch’s purpose matters more than whether it cherry-picks cleanly.
- Clock, pin-control, power, bus, and application dependencies constrain migration order. A peripheral failure can originate in a provider below its driver.
- Pinned vendor APIs and binary components can make a mainline replacement incomplete. Feature loss, out-of-tree maintenance, and future hardware changes are distinct tradeoffs.
- Parallel BSP and migration branches preserve a shipping baseline but require ownership of security fixes, integration evidence, and eventual cutover.
- Kernel, DT, rootfs, and application contracts all contribute to migration risk. Mainline compilation alone does not establish compatibility with the fitted MINI or existing product.

122A.11 Pitfalls

- **Underestimating scope.** “It’s just a kernel upgrade.” A 7000-patch migration is **6 person-months minimum**. Plan accordingly.
- **Migrating to non-LTS mainline.** Mainline rolls every 9 weeks. Pick an LTS (currently 6.6, supported until 2028). Don’t pick the bleeding-edge.
- **Skipping subsystem isolation.** “Let me bring up everything at once” → debug becomes impossible. Phase-by-phase.
- **Forgetting userspace compatibility.** Mainline 6.6 expects glibc 2.35+. Your BSP rootfs has glibc 2.24. Rebuild rootfs too.

> **MCU bridge:** Think of the rootfs as the firmware image’s file-backed runtime environment. On an MCU you link everything into flash. On Linux, programs and config live in this mounted tree. **rootfs:** root filesystem, the directory tree mounted at / that contains /bin, /etc, /dev, and libraries.

- **DT bindings drift.** A binding that “worked in 4.1.15” may have been refactored in mainline. Update DT to match.
- **Out-of-tree drivers break.** Every kernel API change risks your custom driver. Keep custom drivers minimal. Upstream them when possible.

- **Toolchain ABI mismatch.** gcc 13 produces slightly different code than gcc 6. Some assumptions in old assembly break.

> **ABI:** Application Binary Interface: the calling convention, register use, binary format, and library contract that let separately built code run together.

- **Customer regression.** Sometimes a behavior the customer relies on was a BSP-only "feature" not in mainline. Document. Communicate. Sometimes you have to accept regression.
- **Management commitment fades.** "It's 4 months in, no new features yet, can we go back?" Plan for this. Have weekly visible progress milestones. Show CVE-burndown charts.
- **Going halfway.** Migrating half the subsystems = the worst of both worlds (now you have two trees forever). Commit to the cutover.
- **Not migrating despite need.** "Easier to backport CVEs." After 100+ CVEs, the backport effort exceeds the migration effort. Track this.

122A.12 Going deeper

- **NXP's Yocto BSP release notes**: show what got merged upstream by version.
- **Kernel LTS announcements** (lore.kernel.org → stable@vger), for current LTS lifetimes.
- **CVE database (NIST NVD)**: search by `linux_kernel`.
- **Bootlin's "BSP support" presentations**: practical migration case studies.
- **`scripts/git_diff_filter.sh`** in kernel, for filtering relevant diff.
- **Greg KH's stable-tree maintenance talks**: for why and how stable trees exist.
- **`grsecurity` and `KSPP`**: for understanding the security argument for mainline.
- **Ch 30A**: the original kernel-lifecycle decision framework.
- **Ch 120A**: for upstreaming what's worth saving.
- **Ch 123 / Ch 123A**: for the Yocto / Buildroot recipe migration that goes alongside.

> Next chapter: **Chapter 123: Yocto vs Buildroot, an honest comparison**.

Chapter 123: Yocto vs Buildroot, an honest comparison

What: picking a build system shapes your product's CI, release, and maintenance flow for years. **Buildroot** (the make-driven, "tightly-curated tree of packages" approach) vs **Yocto/OpenEmbedded** (the metadata-driven, "recipes + layers" approach). We walk through the mental model of each, build the same image with both side-by-side, compare reproducibility, build times, SDK production, multi-machine support, layer composition, and BSP integration. Then a verdict on when each wins, when each is a poor fit, and when neither is right. **BSP:** Board Support Package: vendor patches, configs, bootloader files, and scripts needed to boot one board. **Yocto:** a metadata-driven build system for producing custom Linux distributions. **Buildroot:** a configuration-driven build system that produces a complete root filesystem and related images.

Why: most production embedded Linux teams use Buildroot or Yocto. Some use both. The choice affects:

- hiring (Yocto skills are scarcer and more expensive than Buildroot),
- CI infrastructure (Yocto builds are slower and need more storage),
- onboarding a new engineer (Buildroot is friendlier),
- maintaining a fleet of variants (Yocto wins),
- debugging a build problem (Buildroot wins).

Choosing badly costs months. Choosing well saves them.

Focus: the mental model is different, Buildroot is "menuconfig builds a complete image". Yocto is "metadata recipes are combined to produce many possible images." Buildroot scales by adding packages. Yocto scales by adding layers + machines + distros. For a single product with 1–3 variants, Buildroot wins. For a vendor BSP that serves dozens of customer products from one codebase, Yocto wins. Most teams overestimate their multi-variant complexity and end up with Yocto sledgehammers cracking Buildroot walnuts.

123.1 Mental model side-by-side

	Buildroot	**Yocto / OpenEmbedded**
Configuration unit	a single `defconfig`	a stack of *layers*; a *machine* + *distro* + *image* recipe
Build engine	GNU Make	BitBake (Python-driven, task-based)
Package definitions	`package/foo/foo.mk` + `Config.in`	`meta-*/recipes-*/foo/foo_1.2.3.bb`
Per-package config	menuconfig globals	`bbappend` files per layer
Parallel builds	yes (make -j)	yes (BitBake tasks across CPUs + machines)
Build cache	none (rebuild from scratch is the norm)	sstate-cache (artifacts keyed by input hash)
Time-to-first-image	30–60 min	1–4 hours (initial); 5–30 min (incremental)
Output	a single `output/images/` tree	per-image `tmp/deploy/images/<machine>/...`
SDK production	`make sdk`	`bitbake -c populate_sdk core-image-minimal`
Reproducibility	quasi (depends on host)	strong (with `BB_HASHSERVE`, ~bit-for-bit)
Number of packages	~3000	~10,000+
Documentation	clear, single PDF	sprawling, spread across many sources

	Buildroot	Yocto / OpenEmbedded
Learning curve	gentle (few weeks)	steep (months)

123.2 Buildroot in 5 minutes

```
git clone https://git.buildroot.net/buildroot
cd buildroot
make qemu_arm_versatile_defconfig             # or your custom defconfig
make menuconfig
# Target options → Target Architecture → ARM
# Toolchain → external Linaro toolchain
# Target packages → Hardware handling → i2c-tools (check)
# Filesystem images → ext2/3/4 root filesystem (check)
# Bootloaders → U-Boot
# Kernel → use the Linux 6.6
make -j$(nproc)
# 30 min later:
ls output/images/
# bzImage  rootfs.ext4  rootfs.tar  u-boot.bin
```

That’s everything in one tree: cross-toolchain (built or downloaded), U-Boot, kernel, rootfs with the packages you picked. One command, predictable output. To save your config:

> **MCU bridge:** Think of the rootfs as the firmware image’s file-backed runtime environment. On an MCU you link everything into flash. On Linux, programs and config live in this mounted tree. **MCU bridge:** Think of U-Boot like a much larger boot stub plus debug monitor: it initializes hardware, loads the next image, and gives you commands before Linux starts. **rootfs:** root filesystem, the directory tree mounted at / that contains /bin, /etc, /dev, and libraries. **U-Boot:** the bootloader that initializes enough hardware to load and start the Linux kernel.

```
make savedefconfig           # writes to configs/myboard_defconfig
```

Commit `configs/myboard_defconfig` to git. Teammates `make myboard_defconfig && make` to get identical output.

Adding a custom package:

```
# package/myapp/Config.in
config BR2_PACKAGE_MYAPP
    bool "myapp"
    help
      My internal app.
```

```
# package/myapp/myapp.mk
MYAPP_VERSION = 1.0
MYAPP_SITE = $(BR2_EXTERNAL)/src/myapp
MYAPP_SITE_METHOD = local
MYAPP_DEPENDENCIES = libcurl

define MYAPP_BUILD_CMDS
    $(MAKE) CC="$(TARGET_CC)" -C $(@D)
endef

define MYAPP_INSTALL_TARGET_CMDS
    $(INSTALL) -m 0755 $(@D)/myapp $(TARGET_DIR)/usr/bin/
endef

$(eval $(generic-package))
```

50 lines. The `generic-package` framework provides defaults for everything you don't override. Add `source "package/myapp/Config.in"` somewhere in the `Config.in` tree. Rebuild. `myapp` is now in the rootfs.

123.3 Yocto in 30 minutes

```
git clone -b kirkstone https://git.yoctoproject.org/poky
cd poky
. oe-init-build-env
# Working in build/

# Pick a machine
# bblayers.conf: BBLAYERS += "/path/to/meta-imx /path/to/meta-mybsp"
# local.conf: MACHINE = "imx6ull-myboard"
#             DISTRO = "poky"

bitbake core-image-minimal
# Takes 1–4 hours first build (downloads + compiles ~1000 packages)
ls tmp/deploy/images/imx6ull-myboard/
# core-image-minimal-imx6ull-myboard.tar.bz2
# zImage  imx6ull-myboard.dtb  u-boot.imx  ...
```

The pieces:

- **`poky`** = the reference distro + tooling.
- **`meta-imx`** = NXP's BSP layer (or `meta-freescale` for older).
- **`meta-mybsp`** = your layer with board DTS + machine config.
- **`local.conf`** = build-specific knobs.

A layer's structure:

```
meta-mybsp/
├── conf/
```

```
│   ├── layer.conf
│   └── machine/imx6ull-myboard.conf
├── recipes-bsp/
│   └── u-boot/u-boot-mybsp_2026.04.bb
├── recipes-kernel/
│   └── linux/linux-mybsp_6.6.bb
├── recipes-core/
│   └── images/myapp-image.bb
└── recipes-myapp/
    └── myapp/myapp_1.0.bb
```

Each `.bb` recipe is a Python+shell file describing how to fetch, configure, build, install, and package one component.

`myapp_1.0.bb`:

```
DESCRIPTION = "My internal app"
LICENSE = "MIT"
LIC_FILES_CHKSUM = "file://LICENSE;md5=abc..."

SRC_URI = "git://github.com/myorg/myapp.git;protocol=https;branch=main"
SRCREV = "abc123def456"
S = "${WORKDIR}/git"

DEPENDS = "curl"

do_compile() {
    oe_runmake CC="${CC}" CFLAGS="${CFLAGS}" LDFLAGS="${LDFLAGS}"
}

do_install() {
    install -d ${D}${bindir}
    install -m 0755 ${S}/myapp ${D}${bindir}/
}
```

The `${...}` are BitBake variables filled in by the layer config + base classes (`inherit autotools`, `inherit cmake`, etc., provide build-system-specific defaults).

123.4 When Buildroot wins

- **One product, few variants.** A single defconfig per variant. Everything in one tree.
- **Build-from-source is the goal.** No interest in vendor SDKs or sharing recipes across products.
- **Small team.** 1–5 engineers can master Buildroot in a week.
- **Predictable build time matters.** Buildroot builds are fast (30 min) and deterministic.
- **Debugging build failures.** Stepping through a `make` is easier than a BitBake task graph.

- **Small rootfs.** Buildroot's curated packages are lean. The default core-image is ~30 MB. Yocto's `core-image-minimal` is closer to **~10 MB** (depending on init/packagegroups). Only `core-image-base` or `core-image-full-cmdline` climbs into the 60 MB+ range.

Example: a hobbyist product, a one-off consumer gadget, a small-fleet IoT device. Buildroot for these is right.

123.5 When Yocto wins

- **Multi-customer BSP.** One vendor's layer (`meta-imx`) shared across 50+ downstream products.
- **Many machines, one codebase.** `MACHINE = imx6ull-foo` vs `MACHINE = imx8mq-bar` from the same layer.
- **Vendor support contracts.** NXP, TI, ST ship Yocto BSPs as their officially-supported integration.
- **Reproducibility (binary-identical builds).** sstate-cache + `BB_HASHSERVE` give the strongest reproducibility in the industry.
- **Compliance / license tracking.** Yocto generates SBOMs (Software Bills of Materials), license manifests, source archives for GPL compliance.
- **Distro-style packaging.** Build .deb or .rpm for OTA delivery.

Example: a vendor-grade industrial gateway, a fleet of medical devices, anything needing rigorous compliance + 10-year maintenance.

123.6 When neither is right

- **Bare-metal RTOS**: both these tools assume a full Linux userspace.
- **Single-binary container**: use Alpine + Docker. No rootfs builder needed.
- **Pre-built distribution**: `Ubuntu Core` or `Debian arm64` may give you everything in 1 day. Skip the build tooling.
- **Microcontroller**: neither applies. Use Zephyr or vendor SDKs.

123.7 Side-by-side example, same recipe in both

A package `foo` that builds with autotools:

Buildroot

`package/foo/Config.in`:

```
config BR2_PACKAGE_FOO
    bool "foo"
    select BR2_PACKAGE_LIBBAR
```

`package/foo/foo.mk`:

```
FOO_VERSION = 1.0
FOO_SITE = https://example.com
FOO_SOURCE = foo-$(FOO_VERSION).tar.gz
FOO_DEPENDENCIES = libbar
FOO_LICENSE = MIT

$(eval $(autotools-package))
```

~10 lines. Done.

Yocto

`meta-mybsp/recipes-myapp/foo/foo_1.0.bb`:

```
DESCRIPTION = "foo utility"
HOMEPAGE = "https://example.com"
LICENSE = "MIT"
LIC_FILES_CHKSUM = "file://COPYING;md5=abc..."

SRC_URI = "https://example.com/foo-${PV}.tar.gz"
SRC_URI[md5sum] = "abc..."
SRC_URI[sha256sum] = "abc..."

DEPENDS = "libbar"

inherit autotools
```

~10 lines too! Yocto's class system (`inherit autotools`) handles configure/build/install identically to Buildroot's `autotools-package`. For simple recipes, similar effort.

For complex builds, Yocto's task graph (do_fetch, do_unpack, do_patch, do_configure, do_compile, do_install, do_package, ...) gives finer control but more places to get lost.

123.8 Reproducible builds

Buildroot is mostly reproducible: pin Buildroot version + pin defconfig + pin host toolchain → likely identical output, but not guaranteed.

Yocto with `BB_HASHSERVE` is strongly reproducible, same inputs always produce bit-for-bit identical outputs. Important for security-audited products and regulatory compliance.

Both can be made fully reproducible with care. Yocto requires less effort.

123.9 SDK production

For application developers (who don't want to build the whole kernel + rootfs), produce an SDK:

Buildroot

```
make sdk
# output/images/sdk.tar.gz — extract, source environment-setup, cross-compile your app
```

Yocto

```
bitbake -c populate_sdk core-image-minimal
# tmp/deploy/sdk/poky-glibc-x86_64-core-image-minimal-...-toolchain-....sh
# Run the installer; source environment-setup-...
```

Both produce a directory with `arm-none-linux-gnueabihf-gcc` and a sysroot. Yocto's SDK is slightly larger (more headers, more docs). Buildroot's is leaner.

123.10 CI integration

Both work in CI. Yocto needs more storage (sstate-cache can be 10+ GB) and longer initial build. Buildroot starts faster but has no cache. Every clean build is 30 min.

For CI:

- **Buildroot**: `make defconfig && make` in a Docker container per build.
- **Yocto**: keep an sstate-cache mounted across builds (NFS, S3). First build is slow, subsequent are fast.

> **NFS:** Network File System, which lets the target mount a host directory over Ethernet during development.

123.11 Key Notes

- Buildroot organizes an image around configuration and package definitions. Yocto composes recipes, layers, machines, distribution policy, and image contents.
- The useful comparison is the product's variation and maintenance model, not hello-world recipe length. Simple packages can be compact in either system.
- Build and runtime dependencies must both reach the final image. Successful compilation does not establish that deployed libraries and services are present.

- An SDK exports a toolchain and target sysroot for applications. Its compatibility belongs to the rootfs it represents, not merely its architecture name.
- Input pinning, host assumptions, and artifact comparison underpin reproducibility. Caches reduce work but do not by themselves prove byte-identical output.
- Layer reuse and variant composition offer flexibility at the cost of more metadata interactions. CI storage, licensing records, and source retention are part of the decision.

123.12 Pitfalls

- **Yocto initial build crushing your laptop.** Needs 50+ GB disk, 4+ GB RAM, 4+ cores. A Raspberry Pi can't do it. Use a beefy build server.
- **Buildroot reproducibility myths.** Same defconfig + different host = different output (host gcc version, host libc influences some packages). Pin everything you can.
- **Yocto layer recipes too generic.** Easy to write recipes that work on x86 but break on ARM. `bitbake -c devshell` to debug per-package builds.
- **sstate-cache poisoning.** A bad sstate entry causes cryptic failures. `bitbake -c cleansstate <recipe>` to evict.
- **Layer-priority confusion.** Two layers ship the same recipe. The higher-priority one wins. Set explicit priorities in `layer.conf`.
- **`bbappend` for wrong version.** `foo_%.bbappend` matches any version. `foo_1.2.%.bbappend` matches 1.2.x. Use the right pattern.
- **Buildroot `make clean` blasts everything.** Then full 30-min rebuild. Use `make foo-rebuild` for incremental.
- **License compliance forgotten.** GPL requires source distribution. Both tools can produce source tarballs. Configure and store them.
- **DISTRO != MACHINE.** Yocto's `DISTRO` (e.g., `poky`, `oe-core`, `mistral`) and `MACHINE` (e.g., `imx6ull-myboard`) are orthogonal. Mixing them up = surprising configurations.
- **Building on macOS / Windows.** Both Buildroot and Yocto only support Linux build hosts well. Use a Linux VM or WSL2.
- **Choosing Yocto for a 1-person project.** You'll spend more time on Yocto than on your product. Use Buildroot.

123.13 Going deeper

- **Buildroot manual**: https://buildroot.org/downloads/manual/manual.html. Concise, covers everything.
- **Yocto Project Documentation**: https://docs.yoctoproject.org/. Sprawling but authoritative.
- **OpenEmbedded Layer Index**: https://layers.openembedded.org/. Search for existing recipes/layers.
- **`bblayers.conf` + `local.conf`**: your Yocto build's primary controls.
- **Bootlin's Buildroot training material**: free PDFs.
- **Konsulko Group's Yocto training**: paid but excellent.
- **`bitbake-cookerdaemon-log`**: for debugging build orchestration.
- **`oe-pkgdata-util` + `bitbake-getvar`**: for introspecting recipes.
- **Ch 123A**: when you've decided Yocto + need to write production-quality layers.
- **Ch 35**: original Buildroot chapter.

> Next chapter: **Chapter 123A: Yocto layer development in depth**.

Chapter 123A: Yocto layer development in depth

What: the **Yocto layer design** that production vendors use. We build a 3-layer stack: **`meta-mybsp`** (board + BSP, kernel + U-Boot + DT for `imx6ull-myboard`), **`meta-mybsp-mini`** (board variant, same SoC, smaller display), **`meta-mybsp-myapp`** (application layer, your in-house Qt app, MQTT daemon, OTA config). Plus a separate **distro layer** (`meta-mybsp-distro`) that pins package versions + DISTRO_FEATURES. We walk every meaningful concept: layer priorities, bbappend patterns, machine config, `IMAGE_FEATURES`/`DISTRO_FEATURES`, `wic` for image partitioning, `RAUC`/SWUpdate integration, and the `SRC_URI` cache so reproducible builds work offline. **MCU bridge:** Think of U-Boot like a much larger boot stub plus debug monitor: it initializes hardware, loads the next image, and gives you commands before Linux starts. **BSP:** Board Support Package: vendor patches, configs, bootloader files, and scripts needed to boot one board. **RAUC:** an embedded update framework for signed A/B image installation and rollback. **Yocto:** a metadata-driven build system for producing custom Linux distributions. **U-Boot:** the bootloader that initializes enough hardware to load and start the Linux kernel.

Why: Ch 123 chose Yocto. Now the meat of the work is *writing layers properly*. A bad layer organization makes every change a hunt across the metadata tree. A good one isolates your product from upstream, keeps variants simple, and survives upstream layer updates. Done well, Yocto becomes a tool you'd recommend. Done badly, it becomes the build system everyone hates.

Focus: layers stack like CSS, later layers override earlier ones via priority. The roles are:

- **bbappend**, extend an upstream recipe.
- **a new `.bb`**, your own packages.
- **machine config**, "this hardware needs these kernel modules."
- **distro config**, "this product line needs OpenSSL 3 + systemd."
- **image recipes**, "this final shipped image contains these packages."

Get these separations right and a 5-machine, 3-distro, 10-app matrix becomes manageable. Mix them up (machine-specific stuff in distro config) and you build a tar pit.

123A.1 Layer anatomy

A canonical Yocto layer:

```
meta-mybsp/
├── conf/
│   ├── layer.conf                ← who I am, my priority, my namespace
│   ├── machine/
│   │   └── imx6ull-myboard.conf   ← MACHINE definitions
│   └── distro/
│       └── mybsp-distro.conf      ← optional DISTRO definition
├── recipes-bsp/
│   ├── u-boot/u-boot-mybsp_2026.04.bb
│   └── u-boot/u-boot-mybsp_2026.04.bbappend
├── recipes-kernel/
│   └── linux/linux-mybsp_6.6.bb
├── recipes-core/
│   ├── images/myapp-image.bb
│   └── images/myapp-debug-image.bb  ← variant: includes gdb, strace
├── recipes-myapp/
│   └── myapp/myapp_1.0.bb
├── classes/
│   └── mybsp-helper.bbclass        ← shared helpers
├── files/                          ← static files for recipes to install
└── README.md
```

`conf/layer.conf` is the heart:

```
# Layer name and priority
BBFILE_COLLECTIONS += "mybsp"
BBFILE_PATTERN_mybsp = "^${LAYERDIR}/"
BBFILE_PRIORITY_mybsp = "10"

# Recipes search paths
BBPATH .= ":${LAYERDIR}"
BBFILES += "${LAYERDIR}/recipes-*/*/*.bb \
            ${LAYERDIR}/recipes-*/*/*.bbappend"

# Layer-version compat
LAYERVERSION_mybsp = "1"
LAYERSERIES_COMPAT_mybsp = "kirkstone langdale mickledore nanbield scarthgap"
LAYERDEPENDS_mybsp = "core meta-freescale"
```

Priority `10` is higher than `meta-freescale`'s 8, your overrides win.

123A.2 Machine config

`conf/machine/imx6ull-myboard.conf`:

```
#@TYPE: Machine
#@NAME: i.MX6ULL MyBoard
#@DESCRIPTION: 528 MHz, 512 MB DDR3, FEC × 2, 1 USB
```

```
require conf/machine/include/imx6ull.inc

MACHINEOVERRIDES =. "mx6ull:"

KERNEL_DEVICETREE = "imx6ull-myboard.dtb"
UBOOT_CONFIG ??= "myboard"
UBOOT_CONFIG[myboard] = "imx6ull_myboard_defconfig,sdcard"

# Kernel modules to always install
MACHINE_EXTRA_RRECOMMENDS += "kernel-module-fec kernel-module-mwifiex-sdio"

# wic image layout
IMAGE_FSTYPES += "wic.bz2 wic.bmap"
WKS_FILE = "imx6ull-myboard.wks"

# Specific UART for serial console
SERIAL_CONSOLES = "115200;ttymxc0"
```

`require` includes shared SoC config from upstream `meta-freescale`'s `imx6ull.inc`. Your machine adds board-specific overrides. The `KERNEL_DEVICETREE` selects the DT. `UBOOT_CONFIG` selects the U-Boot defconfig.

123A.3 Image recipes

`recipes-core/images/myapp-image.bb`:

```
SUMMARY = "MyApp production image"
LICENSE = "MIT"

inherit core-image

IMAGE_FEATURES += "ssh-server-dropbear"

IMAGE_INSTALL += " \
    myapp \
    rauc \
    chrony \
    mosquitto-clients \
    libssl3 \
    "

# Strip debug info for production
IMAGE_INSTALL:remove = "kernel-module-*-dbg"
INHIBIT_PACKAGE_STRIP = "0"
INHIBIT_PACKAGE_DEBUG_SPLIT = "0"

IMAGE_ROOTFS_SIZE = "204800"   # 200 MB
IMAGE_OVERHEAD_FACTOR = "1.1"
```

`bitbake myapp-image` builds the rootfs with these packages, applies any IMAGE_FEATURES (ssh-server-dropbear adds dropbear ssh), produces a `.wic.bz2` for SD/eMMC flashing.

> **MCU bridge:** Think of the rootfs as the firmware image's file-backed runtime environment. On an MCU you link everything into flash. On Linux, programs and config live in this mounted tree. **rootfs:** root filesystem, the directory tree mounted at / that contains /bin, /etc, /dev, and libraries.

Variant for debug:

```
# recipes-core/images/myapp-debug-image.bb
require myapp-image.bb

SUMMARY = "MyApp debug image (with gdb, strace, perf)"
IMAGE_INSTALL += "gdb gdbserver strace ltrace perf valgrind dropbear"
IMAGE_FEATURES += "debug-tweaks"
```

`bitbake myapp-debug-image` builds the same base image + debug tools. One-line difference. Fully separate output artifact.

123A.4 Custom recipe, your app

`recipes-myapp/myapp/myapp_1.0.bb`:

```
SUMMARY = "MyApp embedded controller"
LICENSE = "Proprietary"
LIC_FILES_CHKSUM = "file://LICENSE;md5=abc..."

SRC_URI = "git://gitlab.internal/myorg/myapp.git;protocol=ssh;branch=main"
SRCREV = "abc123def4567890abc123def4567890abc123de"   # use the full 40-char SHA1
PV = "1.0+git${SRCREV}"                               # ${SRCPV} was removed in Scarthgap
S = "${WORKDIR}/git"

DEPENDS = "libcurl mosquitto qtbase"
RDEPENDS:${PN} = "chrony mosquitto-clients"

inherit cmake systemd

SYSTEMD_SERVICE:${PN} = "myapp.service"

EXTRA_OECMAKE = "-DBUILD_TESTS=OFF"

do_install:append() {
    install -d ${D}${systemd_unitdir}/system
    install -m 0644 ${WORKDIR}/myapp.service ${D}${systemd_unitdir}/system/
    install -d ${D}${sysconfdir}/myapp
    install -m 0644 ${WORKDIR}/myapp.conf ${D}${sysconfdir}/myapp/
}

FILES:${PN} += "${systemd_unitdir}/system/myapp.service ${sysconfdir}/myapp/"
```

What's happening:

- `inherit cmake` adds `do_configure/compile/install` for CMake.

- `inherit systemd` enables auto-registration of the systemd service.
- `DEPENDS` = build-time deps. `RDEPENDS:${PN}` = runtime deps.
- `do_install:append` adds extra install steps (the systemd unit + config file).
- `FILES:${PN}` declares which paths go in the main package (vs `-dev`, `-dbg`).

Now `bitbake myapp` builds your app. `bitbake myapp-image` includes it in the rootfs.

123A.5 bbappend, extending upstream recipes

You need to enable an extra kernel config option. Don't fork the kernel recipe. Bbappend it:

`recipes-kernel/linux/linux-imx_%.bbappend`:

```
FILESEXTRAPATHS:prepend := "${THISDIR}/${PN}:"

SRC_URI += "file://0001-myboard-add-RFM69-driver.patch \
            file://myboard.cfg"
```

Where `myboard.cfg` is a kernel config fragment:

```
CONFIG_SPI_RFM69=y
CONFIG_IIO_RFM69=y
CONFIG_WIREGUARD=y
```

BitBake applies the patch and the config fragment automatically. You haven't forked `meta-imx`'s kernel recipe. You've augmented it. The change survives upstream layer updates.

123A.6 Distro vs machine vs image, separation of concerns

Distinguishing what goes where:

Concept	What it controls	Example
DISTRO	The product *family*'s policy	"we use systemd, OpenSSL 3, glibc, en_US locale"
MACHINE	The hardware variant	"i.MX6ULL Cortex-A7 at 528 MHz with FEC × 2, no GPU"

Concept	What it controls	Example
IMAGE	The shipped artifact	“myapp + ssh + chrony, no debug tools, 200 MB rootfs”

If you mix these (for example, putting a machine-specific kernel module into the distro config) you create surprises:

- A second machine’s image gets that module unnecessarily.
- Debugging “why does machine X have this kernel module” leads you on a wild goose chase.

Always: machine-specific stuff in `machine/*.conf`. Distro-wide policy in `distro/*.conf`. Image content in `images/*.bb`.

123A.7 Layer priorities for variant management

Build a stack:

```
meta-poky                 (priority 5)   ← reference distro
meta-openembedded/meta-oe (priority 6)
meta-freescale            (priority 7)   ← i.MX SoC support
meta-freescale-3rdparty   (priority 4)   ← board-specific overrides
meta-mybsp                (priority 10)  ← our BSP layer
meta-mybsp-mini           (priority 12)  ← variant for the MINI board
meta-mybsp-myapp          (priority 15)  ← app-specific overrides
meta-mybsp-distro         (priority 20)  ← product-family-wide policy
```

When two layers ship `recipes-bsp/u-boot/u-boot-imx_%.bbappend`, the higher-priority layer’s takes precedence. Use this to override BSP-defaults from your distro layer.

123A.8 wic, disk image layouts

`wic` (Wic Image Creator) generates partition layouts. Spec file:

`recipes-bsp/u-boot/imx6ull-myboard.wks`:

```
# Bootloader at offset 1k (i.MX SD/eMMC raw region)
bootloader --append="console=ttymxc0,115200 root=/dev/mmcblk0p2 rw rootwait"
part u-boot --source rawcopy --sourceparams="file=u-boot-dtb.imx" --ondisk mmcblk0 --no-
table --align 1
part /boot --source bootimg-partition --ondisk mmcblk0 --fstype=vfat --label boot --
active --align 1024 --size 16
part / --source rootfs --ondisk mmcblk0 --fstype=ext4 --label rootfs --align 1024
part swap --ondisk mmcblk0 --size 64 --label swap --fstype=swap
```

After `bitbake myapp-image`:

```
ls tmp/deploy/images/imx6ull-myboard/
# myapp-image-imx6ull-myboard.wic.bz2
bzcat myapp-image-imx6ull-myboard.wic.bz2 | sudo dd of=/dev/sdX bs=4M
```

Done. The wic image has the right partition table, U-Boot in the raw offset, kernel+DTB in /boot, rootfs in /, swap.

For OTA-ready layouts: add A/B partitions for RAUC.

123A.9 RAUC integration

```
# In myapp-image.bb
IMAGE_INSTALL += "rauc"

# /etc/rauc/system.conf in the rootfs (via a custom recipe)
[system]
compatible=mybsp-imx6ull
bootloader=uboot
mountpoint=/mnt/rauc/bundle

[slot.rootfs.0]
device=/dev/mmcblk0p2
type=ext4
bootname=A

[slot.rootfs.1]
device=/dev/mmcblk0p3
type=ext4
bootname=B
```

And in wic file: two A/B rootfs partitions instead of one.

After deployment:

```
rauc install bundle.raucb
reboot
# System boots into the new slot; on success, marks it "good"
```

Ch 125 covers this in depth.

123A.10 SRC_URI cache, offline reproducibility

For builds you can run without internet:

```
# Pre-fetch everything
bitbake -c fetchall myapp-image
# Or:
bitbake --runall=fetch myapp-image
```

```
# Now downloads/ contains all source tarballs + git clones
# Archive it for offline build:
tar cf yocto-downloads.tar downloads/

# On the offline builder
tar xf yocto-downloads.tar
# In local.conf:
BB_GENERATE_MIRROR_TARBALLS = "1"
BB_NO_NETWORK = "1"
```

For reproducible-at-time-X builds: archive downloads + sstate-cache + pinned layer revisions.

123A.11 bbappend recipes for production

Common pattern: a vendor recipe almost-does-what-you-want. You bbappend to tweak.

```
# recipes-connectivity/openssh/openssh_%.bbappend
FILESEXTRAPATHS:prepend := "${THISDIR}/${PN}:"

# Disable root login, enable our hardened sshd_config
SRC_URI += "file://sshd_config"

do_install:append() {
    install -m 0600 ${WORKDIR}/sshd_config ${D}${sysconfdir}/ssh/sshd_config
}
```

Place `sshd_config` in `recipes-connectivity/openssh/openssh/sshd_config`. Yocto finds it via `FILESEXTRAPATHS`.

123A.12 Key Notes

- `MACHINE` describes hardware, `DISTRO` expresses shared policy, and image recipes select shipped contents. Separating these responsibilities limits variant leakage.
- Recipes own component build and packaging behavior. `bbappend` files extend matching recipes, so their version patterns and effective overrides are maintenance contracts.
- `DEPENDS` supplies build inputs, while `RDEPENDS` supplies deployed dependencies. A successful recipe build does not establish that a service can run in the image.
- Pinned `SRCREV` values, layer revisions, and retained downloads identify inputs. Offline access and cache hits do not independently prove reproducibility.
- Wic describes storage layout in an artifact. Copying that artifact to a medium is destructive and requires separate identification of the intended device.

- A/B layout and RAUC slots must agree with bootloader selection and success reporting. Installing the update package alone does not provide rollback.

123A.13 Pitfalls

- **Recipe in wrong layer.** Recipe for an upstream package living in your BSP layer = drifts from upstream. Use bbappend instead.
- **Hardcoded paths.** `/home/dev/yocto/...` in a recipe = breaks on every other dev's machine. Always use `${WORKDIR}` and similar variables.
- **bbappend without `FILESEXTRAPATHS:prepend`.** BitBake can't find your patches. Add the prepend.
- **Layer priority too high.** Overrides everything. Surprising consequences. Use minimum priority needed.
- **`SRCREV = "${AUTOREV}"`.** Builds use latest HEAD. Non-reproducible. Pin to specific commit hashes.
- **Caching across layer changes.** sstate-cache may keep using old object. `bitbake -c cleansstate <recipe>` after edits.
- **`do_install` install with wrong owner.** Resulting rootfs has files owned by build user, breaking installers. Use `install -m 0755 ...` not `cp`.
- **DEPENDS vs RDEPENDS confusion.** DEPENDS at build time. RDEPENDS at run time. Forgetting RDEPENDS means missing libs at runtime.
- **Recipe order: do_compile before do_configure.** Yocto's task order is fixed. If you `addtask` you must specify ordering.
- **machine.conf assumes a kernel feature not enabled.** `KERNEL_DEVICETREE = "x.dtb"` but the dtb isn't in your kernel's Makefile. Bitbake fails late.
- **PR (Package Revision).** Bumping `PR = "r1"` on a recipe lets package managers (OPKG, etc.) know to upgrade. Easy to forget on bugfix recipes.
- **License license license.** Every recipe needs a `LICENSE` and `LIC_FILES_CHKSUM`. Proprietary stuff: use `LICENSE = "Proprietary"` + `LICENSE_FLAGS` for legal compliance.

123A.14 Going deeper

> **Examples and production:** Do not burn fuses, enroll production keys, or sign release images while using these examples. Use throwaway keys and back up the unsigned image plus the key directory before testing irreversible security flows.

- **Yocto Mega-Manual**: https://docs.yoctoproject.org/singleindex.html.
- **Yocto Reference Manual**: variables, classes, tasks.
- **`bitbake-layers`**: `create-layer`, `add-layer`, `show-layers`.
- **`devtool`**: for iterative recipe development (`devtool modify`, `devtool finish`).
- **`oe-pkgdata-util`**: query the package database.
- **`bitbake -g <image>. cat task-depends.dot`**: visualize task dependencies.
- **`pyrex`**: containerized Yocto builds.
- **Konsulko + Pengutronix Yocto consulting reports**: for production-grade patterns.
- **`meta-virtualization`**: for containers in Yocto.
- **`meta-security`**: for hardened distro layers.
- **Ch 123**: the comparison chapter that led you here.
- **Ch 125**: RAUC / OTA for Yocto-built images.

> Next chapter: **Chapter 124: Secure boot (HAB) and OP-TEE**. **HAB:** High Assurance Boot, NXP’s ROM-enforced secure boot mechanism on i.MX SoCs.

Chapter 124: Secure boot (HAB) and OP-TEE

> **Examples and production:** Do not burn fuses, enroll production keys, or sign release images while using these examples. Use throwaway keys and back up the unsigned image plus the key directory before testing irreversible security flows.

> **What: NXP HAB (High Assurance Boot)**, the SoC-enforced chain-of-trust that ensures only signed bootloaders/kernels run on production i.MX devices. Plus **TrustZone** and **OP-TEE**, the ARM-architectural Secure World and the most-used open-source TEE (Trusted Execution Environment). We walk: the cryptographic chain ROM → SRK fuses → CSF → signed U-Boot → signed kernel → dm-verity rootfs. NXP's **CST (Code Signing Tool)** for producing CSF files. The *key ceremony* (how to generate, store, and rotate signing keys). TrustZone primer (monitor mode, SMC calls, world switch). OP-TEE basics (Trusted Application lifecycle, REE↔TEE communication, TA development). **MCU bridge:** Think of the rootfs as the firmware image's file-backed runtime environment. On an MCU you link everything into flash. On Linux, programs and config live in this mounted tree. **MCU bridge:** Think of U-Boot like a much larger boot stub plus debug monitor: it initializes hardware, loads the next image, and gives you commands before Linux starts. **rootfs:** root filesystem, the directory tree mounted at / that contains /bin, /etc, /dev, and libraries. **U-Boot:** the bootloader that initializes enough hardware to load and start the Linux kernel.
>
> **Why:** verified boot is needed for any product handling user data, payment credentials, certificate-based identity, or DRM. Without it, an attacker with physical access can replace U-Boot, boot a custom kernel that bypasses authentication, or extract storage encryption keys. With HAB plus dm-verity, the device resists most physical-access attacks. Silicon-level attacks (decap, side-channel, fault injection) remain possible but require expensive equipment. OP-TEE adds a runtime-isolated execution domain, Secure World keys, crypto operations, attestation primitives are inaccessible even to a fully-compromised Linux kernel.
>
> **Focus:** the chain works like this:
>
> 1. The ROM checks U-Boot's signature against the SRK hash in eFuses.
> 2. Verified U-Boot checks the kernel and DT signature.
> 3. The verified kernel mounts a dm-verity'd rootfs whose hash matches.
> 4. User apps can use OP-TEE to access secrets that no part of Linux can read.
>
> Break any link and all later links lose meaning.

> Key management is the part that bites: if you lose the private key you brick the fleet. If you expose it you hand attackers full control. This chapter is short on the easy bits and long on the parts you'll regret skipping.
>
> **Tooling. Host:** `openssl` (preinstalled), NXP's **CST** (Code Signing Tool, downloaded from NXP after registration, non-redistributable). **Target:** OP-TEE client (`tee-supplicant`, `libteec`), build from `OP-TEE/optee_os` + `OP-TEE/optee_client`, or use Buildroot's `BR2_PACKAGE_OPTEE_CLIENT=y`. Full reference: Userspace tooling appendix. **Buildroot:** a configuration-driven build system that produces a complete root filesystem and related images.

124.1 The threat model

What HAB + dm-verity defends against:

- **Boot-time tampering**: attacker replaces U-Boot on SD with a modified copy. ROM checks the signature. Fails. Refuses to boot.
- **OS modification**: attacker mounts the SD on their PC and edits `/etc/passwd`. Dm-verity detects the hash mismatch. Mount fails. Linux refuses to boot.
- **Kernel module injection**: attacker installs a rootkit `.ko`. Linux's `module.sig_enforce` (paired with a hash in initramfs) rejects unsigned modules.
- **Runtime malware**: attacker exploits a Linux vulnerability. Runs code in Normal World. OP-TEE's secure-world keys remain inaccessible.

What it *doesn't* defend against:

- **Silicon-level attacks**: chip decapping, side-channel power analysis, fault injection. Requires expensive equipment and is detected by audit.
- **JTAG**: must be disabled by fuse in production.

> **MCU bridge:** Think of JTAG like SWD debugging on Cortex-M: halt, read registers, set breakpoints. The Cortex-A path adds MMU state, privilege modes, and more complex reset behavior. **MCU bridge:** Think of the MMU as a hardware address translator in front of every load/store. Cortex-M usually runs physical addresses directly. Linux relies on virtual addresses and page permissions. **JTAG:** the hardware debug scan chain used to halt, inspect, and single-step CPUs.

- **Pre-fuse-blown attacker**: someone with the device before it leaves the factory. Manufacturing security is a separate concern.
- **TEE compromise**: OP-TEE itself has bugs. Keep it updated.

124.2 i.MX HAB, the chain of trust

```
Power on
     │
     ▼
ROM verifies: U-Boot signature matches SRK in fuses?
     │ yes
     ▼
U-Boot runs
     │
     ▼
U-Boot verifies: FIT image (kernel + DT + initramfs) signed?
     │ yes
     ▼
Kernel runs
     │
     ▼
Kernel mounts rootfs with dm-verity: root hash matches kernel cmdline?
     │ yes
     ▼
userspace
     │
     ▼
Kernel module load: each .ko signed with key in keyring?
     │ yes
     ▼
Daemons run
```

Each step refuses to proceed if signature fails. Fail-open is impossible, the SoC's mask ROM hardcodes this behavior.

The crypto

- **SRK (Super Root Key)**: a 4096-bit RSA public key. The hash of the SRK is **blown into eFuses** during manufacturing.
- **CSF (Command Sequence File)**: a binary blob that points to the SRK, the actual signing key (CSK), the signature, and the regions of the image to verify.
- **CST (Code Signing Tool)**: NXP's tool that generates the CSF from your image + your keys.

For a single i.MX6ULL part: you have one set of SRK keys. You sign all your firmware with the corresponding CSK. The part trusts your firmware and nothing else.

124.3 Key ceremony, the most under-emphasized topic

Before any HAB, you generate **the keys**. This is a ritual, not a checkbox:

1. **Air-gapped machine**: never connected to the internet. A laptop in a safe.
2. **Generate root CA**: `openssl genrsa -aes256 -out ca.key 4096`. Password protect.

3. **Generate SRKs (× 4)**: NXP CST supports up to 4 SRKs (you can revoke individual ones). Generate all 4 on the air-gapped machine.
4. **Hash for fuse blow**: compute SHA-256 of SRK public key concatenation. This 256-bit hash goes into 8 OTP fuses.
5. **Backup**: SRK private keys go on (multiple) hardware tokens (Nitrokey HSM, YubiHSM). Store in different physical locations.
6. **Signing**: production signing happens on the air-gapped machine. Signed binaries go out via USB stick.
7. **Rotation plan**: if SRK 1 is compromised, you can blow another fuse to revoke it. SRK 2 becomes active. **If all 4 are compromised, you must scrap the fleet**, the SoC will never trust new keys.

For prototyping: simpler procedure (keys on your laptop, no HSM). For production: take the ceremony seriously. If you lose the keys, you can't ship firmware updates. Whatever was last signed is what the fleet runs forever.

124.4 Producing a signed U-Boot

```
# 1. Generate keys (CST has a key-gen script)
cd cst-3.4.0/keys
./hab4_pki_tree.sh
# Answers:
#   key length: 4096
#   keys are signed: yes
#   serial number: random
#   key passphrase: <set one>

# This produces:
#   SRK_1_2_3_4_table.bin    (the SRK table U-Boot embeds)
#   SRK_1_2_3_4_fuse.bin     (the hash to blow into fuses)
#   crts/SRK1_sha256_4096_65537_v3_ca_crt.pem  (the SRK certs)
#   keys/SRK1_sha256_4096_65537_v3_ca_key.pem  (the SRK private keys)
#   crts/CSF1_1_sha256_4096_65537_v3_usr_crt.pem
#   keys/CSF1_1_sha256_4096_65537_v3_usr_key.pem

# 2. Build U-Boot with HAB support
cd u-boot
make CONFIG_SECURE_BOOT=y CONFIG_FIT_SIGNATURE=y CONFIG_HAB=y myboard_defconfig
make CROSS_COMPILE=arm-none-linux-gnueabihf- -j$(nproc)

# 3. Generate CSF (Command Sequence File)
# Use the helper script that comes with U-Boot or NXP's CST
cat > csf.txt <<EOF
[Header]
    Version = 4.3
    Hash Algorithm = sha256
    Engine Configuration = 0
    Certificate Format = X509
    Signature Format = CMS
```

```
[Install SRK]
    File = "crts/SRK_1_2_3_4_table.bin"
    Source index = 0

[Install CSFK]
    File = "crts/CSF1_1_sha256_4096_65537_v3_usr_crt.pem"

[Authenticate CSF]

[Install Key]
    Verification index = 0
    Target index = 2
    File = "crts/IMG1_1_sha256_4096_65537_v3_usr_crt.pem"

[Authenticate Data]
    Verification index = 2
    Blocks = 0x877FF400 0x00000000 0x00065F60 "u-boot-ivt.imx"
EOF

cst -i csf.txt -o csf.bin

# 4. Append CSF to U-Boot
cat u-boot.imx csf.bin > u-boot-signed.imx

# 5. Flash + verify
dd if=u-boot-signed.imx of=/dev/sdX bs=1k seek=1 conv=fsync
# Boot; in U-Boot:
hab_status
# No HAB Events Found!     <-- success
```

The hard part is *verifying* the CSF before fuse-blow. Once you blow the SRK fuse, an unsigned U-Boot will brick the part forever (well, recoverable via SDP if HAB hasn't been closed). Test with **HAB open** mode first (the SoC checks signatures and logs failures but still boots) until you're confident.

124.5 Closing HAB, the irreversible step

Once you're sure:

```
=> fuse prog 0 6 0x2                # set HAB_TYPE = closed
=> reset
```

After reset: ROM refuses to boot any unsigned image. **This is irreversible**. The part will only run firmware signed by your SRK forever.

Production flow:

- All fuses blown at factory programming station.
- Devices that ship are HAB-closed.
- Devices for development have HAB open (separate part SKUs or marked units).

124.6 Signed FIT for kernel + DTB

After U-Boot is signed, sign the kernel-DTB-initramfs FIT image so U-Boot only boots a trusted kernel:

> **FIT:** Flattened Image Tree, U-Boot's container format for kernels, DTBs, initramfs images, hashes, and signatures.

```
# .its file describing the FIT
cat > kernel.its <<EOF
/dts-v1/;
/ {
    description = "Kernel + DTB + ramdisk";
    #address-cells = <1>;
    images {
        kernel {
            description = "Linux kernel";
            data = /incbin/("zImage");
            type = "kernel"; arch = "arm"; os = "linux"; compression = "none";
            load = <0x80800000>; entry = <0x80800000>;
            hash-1 { algo = "sha256"; };
        };
        fdt {
            description = "Flattened DT";
            data = /incbin/("imx6ull-myboard.dtb");
            type = "flat_dt"; arch = "arm"; compression = "none";
            hash-1 { algo = "sha256"; };
        };
        ramdisk {
            description = "Initramfs";
            data = /incbin/("initramfs.cpio.gz");
            type = "ramdisk"; arch = "arm"; os = "linux";
            compression = "gzip";
            load = <0>; entry = <0>;
            hash-1 { algo = "sha256"; };
        };
    };
    configurations {
        default = "conf-1";
        conf-1 {
            description = "Boot";
            kernel = "kernel";
            fdt = "fdt";
            ramdisk = "ramdisk";
            hash-1 { algo = "sha256"; };
            signature-1 {
                algo = "sha256,rsa4096";
                key-name-hint = "dev";
                sign-images = "kernel", "fdt", "ramdisk";
            };
        };
    };
};
EOF
```

```
# Sign
mkimage -f kernel.its -K imx6ull-myboard.dtb -k keys/ -r kernel.itb

# The U-Boot DTB now contains the public key as a /signature/key-dev node
```

U-Boot's `bootm` verifies signatures before transferring control. Boot fails noisily if mismatched.

124.7 dm-verity for the rootfs

dm-verity = a kernel feature that hashes each block of a read-only block device into a Merkle tree. The root hash is verified at mount. Any disk modification is detected on the read of the affected block.

```
# Build verity tree
veritysetup format /dev/loop0 /dev/loop1
# Outputs: Root hash: abcdef01234... (this hash is the "trusted" anchor)

# At kernel cmdline:
# root=/dev/dm-0 dm-mod.create="root,,,ro, 0 7864320 verity 1 /dev/mmcblk0p2 /dev/
mmcblk0p3 4096 4096 983040 1 sha256 abcdef01... abcdef01..."

# Or via dracut/initramfs that calls veritysetup
```

The root hash goes in the kernel cmdline (signed via FIT signature, so an attacker can't change it). The rootfs is read-only. Logs go to overlayfs in tmpfs (lost on reboot, by design) or to a separate data partition.

124.8 TrustZone + OP-TEE

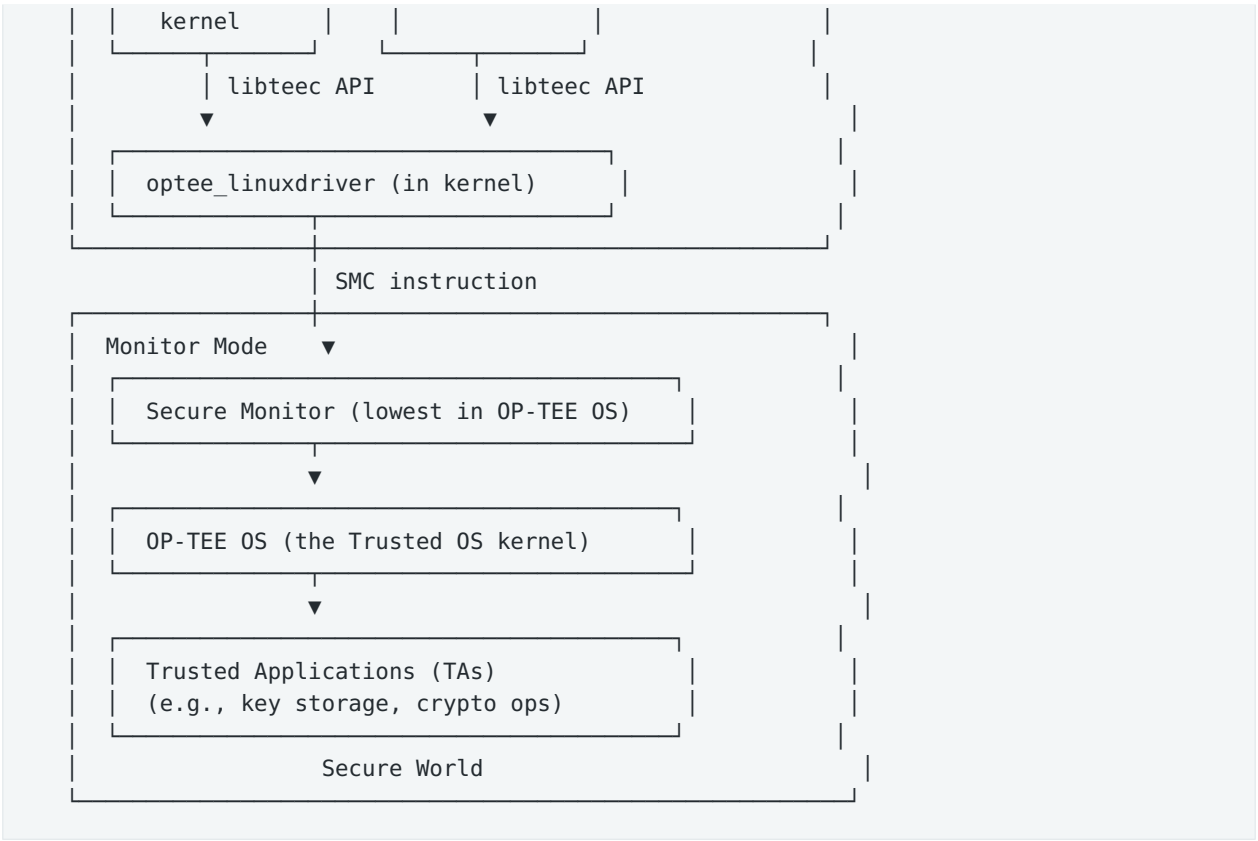
ARMv7-A's TrustZone divides the CPU into two "worlds":

- **Normal World** (NW): regular Linux
- **Secure World** (SW): runs a Trusted OS (OP-TEE)

Each has its own MMU, exception vectors, peripheral access rules (configurable per peripheral). Switching is via **SMC** (Secure Monitor Call) instruction → traps to Monitor Mode → switches world.

> **MMU:** Memory Management Unit, hardware that translates virtual addresses to physical addresses and enforces permissions.





The kernel can request services from OP-TEE via the optee driver. User-space talks via `libteec`. Each request is an SMC → kernel context switch → OP-TEE serves request → SMC return.

124.9 Bringing up OP-TEE on i.MX6ULL

```
# Get OP-TEE
git clone https://github.com/OP-TEE/optee_os.git
cd optee_os

make PLATFORM=imx-mx6ull CROSS_COMPILE=arm-none-linux-gnueabihf- \
     CFG_TZDRAM_START=0x9e000000 CFG_TZDRAM_SIZE=0x02000000

# Produces tee.bin (the Trusted OS image)
```

In FIT image:

```
loadables {
    tee {
        description = "OP-TEE";
        data = /incbin/("tee.bin");
        type = "tee";
        arch = "arm";
        load = <0x9e000000>; entry = <0x9e000000>;
        compression = "none";
    };
};
```

```
configurations {
    conf-1 {
        kernel = "kernel"; fdt = "fdt";
        loadables = "tee";
    };
};
```

U-Boot loads TEE. Kernel cmdline `tee=on` enables it. On boot the kernel finds it via DT (`firmware/optee` node).

```
dmesg | grep tee
# optee: probing for conduit method.
# optee: revision 3.20 (abcdef01)
# optee: dynamic shared memory is enabled
# optee: initialized driver
```

124.10 A simple Trusted Application

OP-TEE TAs are ELFs that run in Secure World. Hello-world TA:

```
// ta/hello_ta.c
#include <tee_internal_api.h>

#define TA_HELLO_UUID { 0x12345678, 0x1234, 0x5678, { 0x9a, 0xbc, ... } }

TEE_Result TA_CreateEntryPoint(void) { return TEE_SUCCESS; }
void TA_DestroyEntryPoint(void) {}
TEE_Result TA_OpenSessionEntryPoint(uint32_t param_types, TEE_Param params[4],
                                    void **sess_ctx) {
    return TEE_SUCCESS;
}
void TA_CloseSessionEntryPoint(void *sess_ctx) {}

TEE_Result TA_InvokeCommandEntryPoint(void *sess_ctx, uint32_t cmd_id,
                                      uint32_t param_types, TEE_Param params[4]) {
    if (cmd_id == 0) {
        DMSG("hello from secure world");
        return TEE_SUCCESS;
    }
    return TEE_ERROR_BAD_PARAMETERS;
}
```

Normal-world client:

```
// host/main.c
#include <tee_client_api.h>

int main(void) {
    TEEC_Context ctx;
    TEEC_Session sess;
    TEEC_UUID uuid = TA_HELLO_UUID;
    uint32_t err_origin;
```

```
    TEEC_InitializeContext(NULL, &ctx);
    TEEC_OpenSession(&ctx, &sess, &uuid, TEEC_LOGIN_PUBLIC, NULL, NULL, &err_origin);
    TEEC_InvokeCommand(&sess, 0, NULL, &err_origin);
    TEEC_CloseSession(&sess);
    TEEC_FinalizeContext(&ctx);
    return 0;
}
```

Run. OP-TEE kernel side passes the request. The TA in Secure World prints “hello from secure world” to OP-TEE’s serial log (which the kernel may or may not relay).

Real TAs include:

- **Key storage**: the secret stays in Secure World. Only operations like sign/verify are exposed.
- **Secure storage of credentials**: passwords, tokens, certificates that survive a kernel compromise.
- **Attestation**: proving to a server that the firmware running here is what the server expects.

124.11 Key Notes

- HAB anchors bootloader authentication in ROM and a provisioned key hash. FIT verification, rootfs integrity, and module policy extend trust only when their inputs are protected.
- Open-mode observations and closed-mode enforcement are different states. Fuse closure and key revocation are irreversible provisioning decisions, not reading requirements.
- Key custody, backups, authorized signing, revocation, and recovery determine whether future updates remain possible. Signed images are only one part of that lifecycle.
- dm-verity detects changes using a trusted root hash and block hashes. Writable logs and persistent data need storage separate from the verified read-only rootfs.
- TrustZone separates Normal World from Secure World. The monitor, OP-TEE, Linux client driver, and TA interfaces define which requests and buffers cross the boundary.
- A Trusted Application narrows access to secrets but has its own validation and update obligations. Secure boot alone does not prevent rollback or guarantee runtime correctness.

124.12 Pitfalls

- **Lost SRK keys.** You can't sign new firmware. Whatever was last signed is what the fleet runs from then on. Use HSMs. Back up. Document.
- **Closing HAB on an untested binary.** Brick.
- **Forgot to revoke an old SRK.** Compromised CSK can sign malicious firmware. Periodically rotate.
- **Keys committed to git.** Even private repos leak. Audit your repos. `git-secrets` to scan.
- **CSF aligned wrong.** "Blocks = 0x... 0x... 0x..." with wrong offsets → CSF validates wrong region. Sig matches but actual binary is different. Use NXP's tools. Don't hand-edit.
- **U-Boot env on writable storage with secure boot.** If env can be modified, attacker can set bootargs. Either: sign env, or move to read-only env, or strip non-essential commands from U-Boot.
- **Boot from USB-SDP with HAB closed.** USB-SDP still works because Boot ROM accepts a signed binary over USB. Attacker with USB access can replace your firmware → make sure UART/USB ports are physically inaccessible in field.
- **dm-verity hash on cmdline becomes leaked.** Not a security failure per se (the hash is public), but reveals the rootfs version. Use signed cmdline.
- **OP-TEE bug = full Secure World compromise.** Keep OP-TEE updated. Subscribe to OP-TEE security advisories.
- **TA crashes due to TEE_TIMEOUT_INFINITE bugs.** Some operations block forever. Reset OP-TEE only by reboot.
- **No anti-rollback.** Attacker downgrades to an old, vulnerable firmware. Use FIT's `compatible` versioning + monotonic version counter checked by U-Boot.

124.13 Going deeper

- **NXP AN4581: Secure Boot on i.MX 6 Series**: the canonical reference.
- **NXP CST 3.4+ documentation** + sample CSFs.
- **U-Boot's `doc/imx/habv4/`**: practical signing setup.
- **OP-TEE documentation**: https://optee.readthedocs.io.
- **`Documentation/admin-guide/dm-verity.rst`** in kernel.
- **`mkimage -k <keydir> -r <its>`**: for FIT signing.

- `/lib/modules/$(uname -r)/build/scripts/sign-file`: for kernel-module signing.
- **GlobalPlatform TEE specifications**: for cross-vendor TEE compatibility.
- **NIST SP 800-193: Platform Firmware Resilience**: government-grade requirements.
- **Ch 125**: OTA needs secure boot to be meaningful.
- **Ch 7**: the original Boot ROM chapter.

> Next chapter: **Chapter 125: Field updates (RAUC, SWUpdate, Mender)**. **RAUC:** an embedded update framework for signed A/B image installation and rollback.

Chapter 125: Field updates

Privilege boundary: $ means normal user. # or sudo means root and can change host or target state. After a privileged command, verify the expected device, service, or file appears before continuing. Roll back by undoing the config change or stopping the service you just enabled.

Examples and production: Do not burn fuses, enroll production keys, or sign release images while using these examples. Use throwaway keys and back up the unsigned image plus the key directory before testing irreversible security flows.

What: over-the-air (OTA) firmware update systems for shipped embedded Linux products. Three options compared: **RAUC** (Robust Auto-Update Client, German rail-grade, simple, well-fit Yocto), **SWUpdate** (Toradex-originated, flexible, complex), **Mender** (commercial-style with hosted backend. Also self-host). Each implements **A/B partitioning** + atomic update + rollback. We design a complete OTA flow: build → sign → host → device pulls + verifies + installs → reboots into new partition → marks “good” if boot completes → fall-back to other partition if not. **RAUC:** an embedded update framework for signed A/B image installation and rollback. **Yocto:** a metadata-driven build system for producing custom Linux distributions.

Why: most shipping products need a way to deliver updates. Bug fixes, security patches, new features, all delivered after the box leaves your hand. The risk: a botched update bricks 10,000 devices. The mitigation: atomic A/B updates + rollback on boot failure. Get the OTA architecture right and you can ship updates weekly. Get it wrong and one bad update can disable thousands of devices in the field.

Focus: A/B is the standard pattern:

- two rootfs partitions, **MCU bridge:** Think of the rootfs as the firmware image’s file-backed runtime environment. On an MCU you link everything into flash. On Linux, programs and config live in this mounted tree. **rootfs:** root filesystem, the directory tree mounted at / that contains /bin, /etc, /dev, and libraries.
- the running kernel mounts one,
- an update writes to the other,
- the bootloader switches to the new one on next reboot,
- if the new one fails to reach a “we’re good” marker within a deadline, the bootloader reverts to the old one.

> The complexity is everywhere else. The open questions are:
>
> - Who hosts the update bundle?
> - How is it signed (Ch 124's keys)?
> - How does the device know an update is available?
> - What if the user's WiFi drops mid-download?
> - How do you stage rollouts (10 % → 50 % → 100 %)?
>
> These systems address all of them.
>
> **Tooling.** This chapter uses `rauc` *or* `swupdate` *or* `mender-client` (pick one), plus `casync` for delta-update chunking.
>
> - **Ubuntu-base (target):** `apt install rauc casync  # or: swupdate, or mender-client`
> - **Buildroot:** `BR2_PACKAGE_RAUC=y BR2_PACKAGE_CASYNC=y  # or BR2_PACKAGE_SWUPDATE=y / BR2_PACKAGE_MENDER=y` **Buildroot:** a configuration-driven build system that produces a complete root filesystem and related images.
> - Full per-tool reference: Userspace tooling appendix.

125.1 Comparison

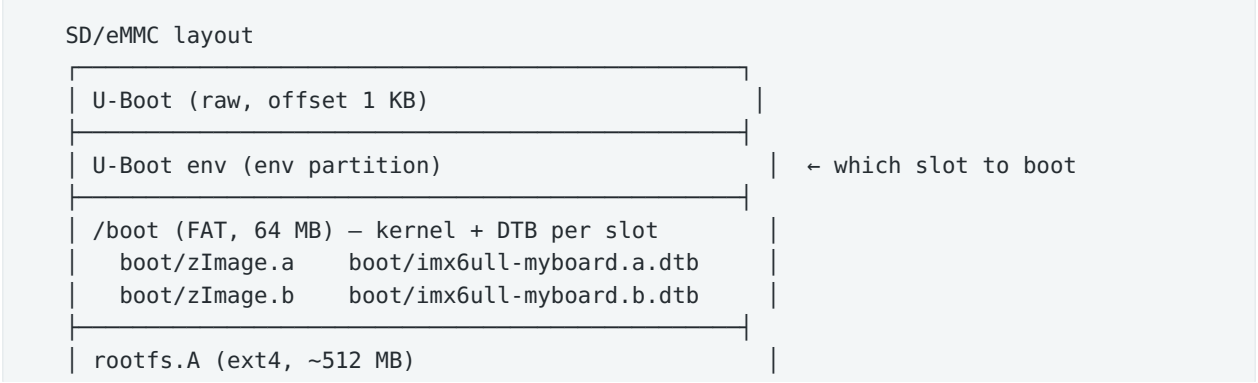
	RAUC	SWUpdate	Mender
Origin	Pengutronix (Germany)	Toradex (Switzerland)	Northern.tech (Norway)
License	LGPL	GPL	Apache 2.0 (client); commercial backend optional
Bundle format	"RAUC bundle" (squashfs+manifest+signature)	CPIO archive with metadata	Mender artifact (.mender)
Signing	OpenSSL/PKCS#11 + X.509 cert chain	OpenSSL CMS	Mender's own

	RAUC	SWUpdate	Mender
Update transports	local, web pull, HTTP streaming	local, HTTP, local-update	local, mender server (cloud or self-hosted)
A/B + rollback	yes	yes	yes
Delta updates	optional (casync)	yes (bsdiff style)	yes (binary delta)
Backend / cloud	none (you build your own)	none	Mender Hosted (paid) or self-host
Yocto integration	excellent (meta-rauc)	excellent (meta-swupdate)	excellent (meta-mender)
Buildroot integration	yes	yes	yes
Use case	EU industrial; vendor-rolled-back-end	Toradex ecosystem; complex updates	“I want a hosted SaaS” or “I want it just to work”

Pick guide:

- **RAUC**: control freaks. You write your own backend. LGPL OK.
- **SWUpdate**: complex update scenarios (e.g., updating only FPGAs, only a partition’s worth of files).
- **Mender**: fastest time-to-deployment. Willing to pay for hosted backend. Or self-host the open-source server.

125.2 A/B partition design



```
    ├──────────────────────────────────────────────────┤
    │ rootfs.B (ext4, ~512 MB)                         │
    ├──────────────────────────────────────────────────┤
    │ /data (ext4, remaining) — persistent user data   │  ← survives updates
    └──────────────────────────────────────────────────┘
```

The bootloader knows: “current slot = A or B”. Read from env var. Update flow:

1. Currently running A. Download new image. Write to B.
2. Set env var: “next boot = B. Previous = A”.
3. Reboot.
4. U-Boot reads env. Boots B’s kernel.
5. Once Linux reaches a known-good state, a specific service reports active, it signals boot success via a U-Boot environment variable.
6. If step 5 doesn’t happen within N seconds, U-Boot reverts to A on next power-cycle (watchdog auto-reset).

125.3 RAUC end-to-end

Device side

`/etc/rauc/system.conf`:

```
[system]
compatible=mybsp-imx6ull-myboard
bootloader=uboot
mountpoint=/run/rauc

[keyring]
path=/etc/rauc/rauc-bundle.cert.pem

[slot.rootfs.0]
device=/dev/mmcblk0p2
type=ext4
bootname=A

[slot.rootfs.1]
device=/dev/mmcblk0p3
type=ext4
bootname=B
```

U-Boot env (set at first boot):

```
BOOT_ORDER="A B"
BOOT_A_LEFT=3
BOOT_B_LEFT=3
```

`BOOT_X_LEFT` counts down with each failed boot attempt. Reaches 0 → fall back to the other slot.

U-Boot script:

```
=> setenv bootcmd 'rauc_boot'
=> setenv rauc_boot 'if test "${BOOT_ORDER%%* *}" = "A"; then \
                       run boot_a; \
                     else \
                       run boot_b; \
                     fi'
=> setenv boot_a 'load mmc 0:1 ${kernel_addr_r} zImage.a; \
                  load mmc 0:1 ${fdt_addr_r} imx6ull-myboard.a.dtb; \
                  setenv bootargs "console=ttymxc0,115200 root=/dev/mmcblk0p2
rauc.slot=A"; \
                  bootz ${kernel_addr_r} - ${fdt_addr_r}'
=> setenv boot_b '... mmcblk0p3 rauc.slot=B'
```

After Linux boots to a known-good state, user-space calls:

```
rauc status mark-good
```

This sets `BOOT_X_LEFT=3` again (full credit) and is the “we’re alive” signal.

Building a bundle

```
# meta-mybsp/recipes-core/bundles/myapp-bundle.bb
inherit bundle
RAUC_BUNDLE_COMPATIBLE = "mybsp-imx6ull-myboard"
RAUC_BUNDLE_VERSION = "${PV}"
RAUC_BUNDLE_DESCRIPTION = "MyApp ${PV} OTA bundle"

RAUC_BUNDLE_SLOTS = "rootfs"
RAUC_SLOT_rootfs = "myapp-image"
RAUC_SLOT_rootfs[fstype] = "ext4"

RAUC_KEY_FILE ?= "${TOPDIR}/keys/rauc-bundle.key.pem"
RAUC_CERT_FILE ?= "${TOPDIR}/keys/rauc-bundle.cert.pem"
```

`bitbake myapp-bundle` produces `myapp-bundle-1.0.raucb`, a signed bundle.

Device-side install

```
# Local install (via USB stick or scp)
rauc install /mnt/usb/myapp-bundle-1.0.raucb

# Streaming via HTTPS
rauc install https://updates.example.com/imx6ull/myapp-bundle-1.0.raucb
```

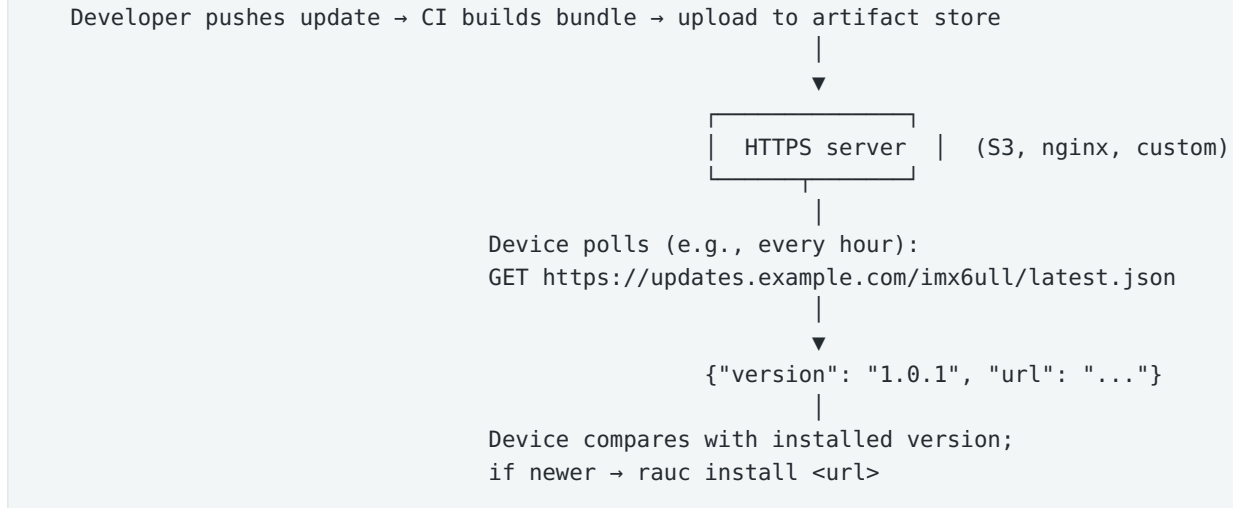
RAUC verifies the signature against the keyring, writes to the inactive slot, updates bootloader env, reboots.

After reboot to new slot, if all OK:

```
rauc status mark-good
```

If something fails (e.g., your watchdog daemon never says “OK”), U-Boot reverts on next reboot.

125.4 Network architecture



You build the “device polling” daemon yourself. RAUC provides the install primitive but no scheduler / staging logic.

For staged rollouts: serve different `latest.json` to different device IDs (canary 10 %, then 50 %, then 100 %).

For privacy, each device authenticates with its own client TLS certificate, provisioned at manufacturing time. The server checks per-device permissions.

125.5 Delta updates with casync

A 1 GB rootfs is too big to push over LTE every week. **casync** (Lennart Poettering’s tool) chunks images. Only changed chunks transfer:

```
# Build a casync index
casync make rootfs.caidx /path/to/rootfs/

# Each rootfs.caidx is a 1 MB index pointing to chunks in a chunk store
# Device downloads only the chunks it doesn't already have
```

RAUC integrates with casync via `RAUC_CASYNC_BUNDLE = "1"`. Typical delta: 5–50 MB instead of 500 MB. Massive savings on cellular fleet.

125.6 SWUpdate’s strengths

SWUpdate’s `sw-description` file is more flexible than RAUC’s bundle format:

```
software = {
    version = "1.0.1";
    images: ({
        filename = "rootfs.ext4";
        device = "/dev/mmcblk0p3";    # or @slot.B
        sha256 = "abc...";
    });
    scripts: ({
        filename = "preinstall.sh";
        type = "shellscript";
    });
}
```

You can update:

- A rootfs partition.
- A single file in the rootfs (`/usr/bin/myapp`).
- An FPGA bitstream.
- A kernel module.
- Run pre/post-install scripts.

For products with non-uniform update needs (e.g., the rootfs rarely changes but the FPGA config updates monthly), SWUpdate wins.

125.7 Mender’s strengths

Mender ships a **complete server stack** (frontend, REST API, artifact storage). Self-host or use Mender’s hosted SaaS.

```
# Build a mender artifact in Yocto
inherit mender-full
INHERIT += "mender-full"
MENDER_DEVICE_TYPE = "imx6ull-myboard"
```

Result: a `core-image-base-imx6ull-myboard.mender` file.

Push to Mender server:

```
mender-cli artifacts upload --description "Bug fix release" core-image-base.mender
mender-cli deployments create --artifact-name 1.0.1 --device-group production
```

The server orchestrates: which devices get it, how many at a time, monitor success rate, auto-pause on failure threshold.

For teams that don’t want to build their own backend, Mender is the fastest path.

125.8 Boot-success detection, the actually-hard part

Defining “boot succeeded” matters. Options:

- **Service-based**: a critical systemd service started OK (e.g., `myapp.service`).
- **Connectivity-based**: device reaches a heartbeat URL within 60 s.
- **Application-based**: the app processes ≥1 request without crashing.
- **Manual**: user presses a “looks fine” button (development only).

Each has a trade-off:

- Service-based misses bugs that crash the service later (5 minutes after boot).
- Connectivity-based misses bugs that prevent network setup.
- Application-based is best but requires app-side instrumentation.

Pattern in production: combine 2–3. “Service started + heartbeat seen + ≥1 user request processed in 10 min” → mark good.

If conditions not met: watchdog fires. Reboot. U-Boot reverts. You’re back on the previous version. Alert sent.

125.9 Key Notes

- An A/B update preserves the running slot while constructing a candidate in the inactive slot. Completion must precede changing persistent boot selection.
- The update client verifies and installs artifacts, the bootloader selects a slot and tracks attempts, and application health policy decides when to mark it good.
- A watchdog supplies a return path when software cannot report failure. Rollback also requires durable attempt state and a usable previous slot, not just two partitions.
- Persistent data belongs outside replaceable system images. Data-format changes can still make an old slot unusable unless migration policy supports rollback.
- Authentication, hardware compatibility, anti-rollback policy, and staged delivery address different risks. A correctly signed bundle can contain a broken application.
- Delta transfer reduces traffic without changing atomicity or recovery. Backend scheduling and fleet monitoring remain separate from the installation primitive.

125.10 Pitfalls

- **No watchdog → no rollback enforcement.** If your “mark good” check doesn’t run, U-Boot still boots the new slot forever. Watchdog (Ch 51A) is mandatory for safe OTA.
- **Power loss mid-update.** If you write to the inactive partition + lose power mid-write, the inactive partition is corrupt. Next boot uses the *active* (= old) slot. Old still works → no harm. But: if you flip the bootloader pointer *before* writing finishes, you brick. Write-then-flip ordering matters.
- **Bundle signed with wrong key.** Device’s keyring doesn’t include this key. Install rejected. Verify keys deployed to fleet match signing keys.
- **Key compromise.** Same as Ch 124: stolen signing key = attacker can ship arbitrary firmware. Use HSMs. Rotate.
- **Anti-rollback missing.** Attacker can install old (vulnerable) versions. Add version-number check in update logic.
- **Persistent data clobbered.** `/data` lost because update wrote new image and didn’t preserve it. Keep `/data` in a separate partition.
- **Network failure mid-download.** RAUC’s HTTPS streaming handles resume. Bundle downloads can survive disconnect. Test it.
- **Bundle larger than rootfs partition.** Won’t fit. Provision rootfs partitions 1.5× expected size to allow growth.
- **Bundle version not incremented.** Device says “I already have 1.0.0, nothing to do.” Always increment.
- **Rolling out untested update.** Stage: dev → 10 % canary → 50 % → 100 %, with auto-pause on >5 % failure rate.
- **Cellular fleet downloading 500 MB each.** Use delta updates (casync). Else your data bill is shocking.
- **Updates breaking customer’s customizations.** A user-modified `/etc/foo` is overwritten by the update. Document. Warn customers. Provide a /etc/local-overrides mechanism.

125.11 Going deeper

- **RAUC documentation**: https://rauc.readthedocs.io/.
- **SWUpdate documentation**: https://sbabic.github.io/swupdate/.
- **Mender documentation**: https://docs.mender.io/.
- **`meta-rauc` + `meta-rauc-community`** layers, for Yocto integration.

- **`casync`**: for delta updates.
- **U-Boot's `env` command + `setenv saveenv`**: the persistent state at the heart of slot selection.
- **Pengutronix blog posts on RAUC**: many production case studies.
- **`mender-cli` + `mender-cli api`**: for CI integration.
- **NIST SP 800-189: Resilient Interdomain Traffic Exchange**: for fleet-OTA threat models.
- **Ch 51A**: watchdog (mandatory companion).
- **Ch 121A**: CI for building bundles.
- **Ch 124**: secure boot for ensuring updates are authentic.

> Next chapter: **Chapter 125A: VSCode + gdbserver remote-debug workflow**.

Chapter 125A: VSCode + gdbserver remote-debug workflow

> **Buildroot:** a configuration-driven build system that produces a complete root filesystem and related images.

> **What:** the **IDE-driven cross-debug workflow** for engineers who prefer Visual Studio Code over the gdb command line. **`gdbserver`** on the target. **`gdb-multiarch`** on the host. VSCode's C/C++ extension and `launch.json` tying them together. `.vscode/c_cpp_properties.json` resolving headers from the cross-sysroot so "Go to Definition" works on both your app *and* kernel sources. Plus a short note on Source Insight, an old commercial editor that's still the fastest tool for read-only kernel-source navigation. **GDB:** the debugger. In cross-debugging it runs on the host while controlling code on the target.
>
> **Why:** many readers come from microcontroller backgrounds where the IDE *is* the debugger (Keil, IAR, STM32CubeIDE). Forcing them to learn gdb's tui mode just to set a breakpoint is unnecessary. VSCode plus the right config gives the same click-to-set-breakpoint experience cross-debugging a remote ARM target, while leaving the underlying gdb fully scriptable for when you do want the command line. Setting up `launch.json` once pays back every debug session after.
>
> **Focus:** VSCode's debug UI is a wrapper around gdb. `launch.json` configures it. You set: which gdb binary, which binary to debug, where gdbserver listens, and where the source tree lives. The non-obvious part is `c_cpp_properties.json`. It must point IntelliSense at the target's sysroot headers, not the host's. Otherwise "Go to Definition" finds your laptop's `stdio.h`, not the cross-compiled one. With both files right, IDE-style debug and accurate Go-to-Definition make embedded debug feel close to desktop.

125A.1 Target side, install gdbserver

```
# Yocto / Buildroot: enable gdbserver in image
# OR Debian/Ubuntu on the target:
apt install gdbserver

# Static-linked is preferable so it works even with weird libc situations
```

`gdbserver` is small, about 100 KB statically linked, and needs no debug info on the target.

125A.2 Host side, VSCode + extensions

```
# Install VSCode (Microsoft .deb or your distro's package)
# Install extensions:
#   C/C++ (ms-vscode.cpptools)
#   C/C++ Themes (optional)
#   Native Debug (webfreak.debug) — alternative; sometimes friendlier for cross
```

And the cross-debugger:

```
apt install gdb-multiarch
```

125A.3 The two config files

VSCode reads two project-local files from `.vscode/`:

- **`launch.json`**: debug-time configuration (which gdb, which binary, where to connect).
- **`c_cpp_properties.json`**: IntelliSense / Go-to-Definition (which headers, which compiler).

`.vscode/launch.json`

```
{
    "version": "0.2.0",
    "configurations": [
        {
            "name": "Cross-debug myapp on imx6ull-pa",
            "type": "cppdbg",
            "request": "launch",
            "program": "${workspaceFolder}/build/myapp",
            "miDebuggerPath": "/usr/bin/gdb-multiarch",
            "miDebuggerServerAddress": "192.168.1.100:2345",
            "cwd": "${workspaceFolder}",
            "args": [],
            "stopAtEntry": true,
            "setupCommands": [
                {
                    "description": "Use sysroot",
                    "text": "set sysroot /home/dev/yocto/build/tmp/work/imx6ull-myboard-
poky-linux-gnueabi/myapp/1.0-r0/recipe-sysroot",
                    "ignoreFailures": false
                },
                {
                    "description": "Enable pretty-printing",
                    "text": "-enable-pretty-printing",
                    "ignoreFailures": true
                }
            ],
            "preLaunchTask": "Start gdbserver"
        }
```

```
    ]
}
```

Key fields:

- **`program`**: the unstripped ELF on the host. Symbols come from here.

> **ELF:** Executable and Linkable Format, the standard Linux object and executable file format.

- **`miDebuggerPath`**: cross-gdb. Must understand ARM.
- **`miDebuggerServerAddress`**: where gdbserver listens on the target.
- **`setupCommands`**: extra gdb commands at start (`set sysroot`, `set solib-search-path`, ...).
- **`preLaunchTask`** (optional): a VSCode task that starts gdbserver on the target. See below.

`.vscode/tasks.json`, pre-launch gdbserver

```
{
    "version": "2.0.0",
    "tasks": [
        {
            "label": "Start gdbserver",
            "type": "shell",
            "command": "ssh root@192.168.1.100 'gdbserver --multi :2345 &'",
            "isBackground": true,
            "problemMatcher": []
        }
    ]
}
```

VSCode runs this task before launching debug. Gdbserver is up. The debug session connects immediately.

`.vscode/c_cpp_properties.json`

```
{
    "configurations": [
        {
            "name": "i.MX6ULL Cross-compile",
            "includePath": [
                "${workspaceFolder}/**",
                "/home/dev/yocto/build/tmp/work/imx6ull-myboard-poky-linux-gnueabi/myapp/
1.0-r0/recipe-sysroot/usr/include",
                "/home/dev/yocto/build/tmp/work/imx6ull-myboard-poky-linux-gnueabi/myapp/
1.0-r0/recipe-sysroot/usr/include/c++/12"
            ],
            "defines": [],
```

```
            "compilerPath": "/home/dev/x-tools/arm-none-linux-gnueabihf/bin/arm-none-
linux-gnueabihf-gcc",
            "cStandard": "c17",
            "cppStandard": "c++17",
            "intelliSenseMode": "linux-gcc-arm"
        }
    ],
    "version": 4
}
```

With this, hover-over-symbol and “Go to Definition” find the target’s headers, not your host’s. Without this, IntelliSense reports errors that aren’t real: it thinks you’re compiling for x86, so ARM-only macros are undefined and code inside `#ifdef __arm__` looks dead.

125A.4 Debugging workflow

1. **Build** on host: `cd build && cmake --build .` (or your build system).
2. **Copy binary** to target: `scp build/myapp root@target:/usr/bin/` (or NFS rootfs auto-syncs).

> **MCU bridge:** Think of the rootfs as the firmware image’s file-backed runtime environment. On an MCU you link everything into flash. On Linux, programs and config live in this mounted tree. **NFS:** Network File System, which lets the target mount a host directory over Ethernet during development. **rootfs:** root filesystem, the directory tree mounted at / that contains /bin, /etc, /dev, and libraries.

1. **F5 in VSCode**: pre-launch task starts gdbserver. Gdb connects. Binary loads. Stops at entry (or your `main()`).
2. **Set breakpoints** by clicking in the gutter of source files.
3. **Continue/step/inspect** via the debug toolbar.
4. **Watch / Locals / Call Stack** panels show variables and stack.
5. **Debug Console** for arbitrary gdb commands (`-exec print foo`).

When done: VSCode disconnects. Gdbserver stops (or stays running if `--multi`).

125A.5 Multi-target, connect to multiple boards

For a fleet:

```
"configurations": [
    { "name": "Debug on board A", "miDebuggerServerAddress": "192.168.1.100:2345", ... },
    { "name": "Debug on board B", "miDebuggerServerAddress": "192.168.1.101:2345", ... },
```

```
    { "name": "Debug on board C", "miDebuggerServerAddress": "192.168.1.102:2345", ... }
]
```

Pick the right config from VSCode's debug dropdown.

125A.6 Debugging kernel modules (advanced)

Set a breakpoint in your Chapter 36/41 LED driver's `probe()`:

```
{
    "name": "Debug LED kernel module",
    "type": "cppdbg",
    "request": "launch",
    "program": "${workspaceFolder}/path/to/linux/vmlinux",
    "miDebuggerPath": "/usr/bin/gdb-multiarch",
    "miDebuggerServerAddress": "192.168.1.100:1234",
    "setupCommands": [
        { "text": "set arch arm" },
        { "text": "set sysroot /home/dev/sysroot" },
        { "text": "add-auto-load-safe-path /home/dev/yocto/build/tmp/work/.../linux/
scripts/gdb/" },
        { "text": "source /home/dev/yocto/build/tmp/work/.../linux/scripts/gdb/vmlinux-
gdb.py" }
    ]
}
```

The kernel-side gdb stub is **KGDB over UART** (Ch 119) on port 1234. After connect:

```
> -exec lx-symbols   # load module symbol files
> break led_probe    # the breakpoint
> continue
# Now insmod the module on the target — breakpoint hits in VSCode.
```

125A.7 Source Insight, the read-only navigator

Source Insight is a 25-year-old commercial editor (Windows. Runs in Wine on Linux). For navigating *huge* code bases (Linux source is ~30 M lines) it's still fastest:

- 5 s to index Linux kernel.
- Hover any symbol → instant definition + cross-references.
- Call graph generation.

It's not a debugger. Not an editor for serious projects. But for "I'm reading the kernel source and want to navigate quickly," nothing beats it. License: $239 one-time.

VSCode's IntelliSense on the kernel source works but is slower (~5 minutes to index). For active editing: VSCode. For pure reading: Source Insight.

125A.8 Key Notes

- VSCode presents debugger state but does not implement target debugging. Host GDB and target `gdbserver` retain their command-line responsibilities.
- `launch.json` identifies the executable, debugger, endpoint, and setup commands. Those references must describe one consistent deployed build.
- `c_cpp_properties.json` controls editor parsing and source navigation. Correct IntelliSense does not establish that runtime symbols or libraries match.
- The host ELF and target sysroot supply debug context. Source-path substitution solves moved trees, not executable or library version mismatches.
- Pre-launch automation owns connection readiness and cleanup. A background SSH command that never becomes ready can block the IDE without exposing a target defect.
- Kernel-module debugging needs a kernel transport and relocated module symbols. An application `gdbserver` connection cannot substitute for that interface.

125A.9 Pitfalls

- **Wrong gdb binary.** `/usr/bin/gdb` (host) won't debug ARM. Needs `gdb-multiarch` or the cross-toolchain gdb.
- **Sysroot path stale.** Yocto recreates `recipe-sysroot` paths each build. The hardcoded path in `launch.json` drifts. Use `${workspaceFolder}` + relative paths or update on rebuild.

> **Yocto:** a metadata-driven build system for producing custom Linux distributions.

- **gdbserver not actually started.** preLaunchTask SSH command fails silently. Debug session times out. Verify with manual SSH first.
- **IntelliSense crashes parsing kernel source.** 30M lines exceeds VSCode's default RAM. Increase: `"C_Cpp.intelliSenseEngine": "default"`, plus 8+ GB RAM.
- **Source paths don't match.** Binary built in `/build/...` but you opened the project in `/src/...`. GDB can't find source files. Add `substitute-path` to setupCommands.
- **Stripped binary on target.** gdbserver opens it but no symbols. Always use unstripped on host side. Target-side `strip` is fine.
- **Breakpoints don't hit.** Often: target's libc not matching host's compile-time libc. Symbols mismatch. Libc functions step into stripped code. Set sysroot.

- **Slow over remote SSH.** gdbserver+ssh tunnel adds ~50 ms per breakpoint. Use direct TCP (open the port in firewall) if you control the network.
- **“Cannot find ld-linux-armhf.so.3”.** sysroot wrong. Verify it points to `${YOCTO_BUILD}/.../recipe-sysroot/`, not the host’s `/`.
- **VSCode’s debug console is one-liner.** Multi-line gdb commands (define, document) need to go in a separate script file loaded via `source` from setupCommands.
- **Pre-launch task hangs.** If `gdbserver --multi :2345 &` doesn’t background properly, VSCode waits forever. Use `isBackground: true` and a `problemMatcher` that VSCode treats as “this is OK to keep running.”

125A.10 Going deeper

- **VSCode C/C++ extension documentation**: https://code.visualstudio.com/docs/cpp/cpp-debug.
- **`launch.json` reference**: all options.
- **GDB manual chapters on MI (Machine Interface)**: what VSCode uses to talk to gdb.
- **`gdb-dashboard`**: terminal alternative if you decide to leave VSCode.
- **`gef` and `pwndbg`**: gdb plugins for security research, with VSCode-like features in terminal.
- **CLion** + remote toolchain, JetBrains commercial alternative. Even smoother UX, paid.
- **Ch 118**: JTAG with gdb (same gdb, different remote target).

> **MCU bridge:** Think of JTAG like SWD debugging on Cortex-M: halt, read registers, set breakpoints. The Cortex-A path adds MMU state, privilege modes, and more complex reset behavior. **JTAG:** the hardware debug scan chain used to halt, inspect, and single-step CPUs.

- **Ch 120**: gdbserver + cli gdb (same workflow, manual setup).
- **Ch 119**: KGDB for kernel-side debug.

> Next chapter: **Chapter 126: Closing: what to read next**. This closes the main i.MX6ULL bring-up path; Part IX continues with optional virtualization studies.

Chapter 126: Closing: what to read next

The main path has followed an embedded system from its boot contract to its update policy. The important result is a model of the responsibilities in that system, not a claim that reading it qualifies a board or turns every example into production code. This chapter points to references for refining that model.

> **MCU bridge:** Think of the rootfs as the firmware image's file-backed runtime environment. On an MCU you link everything into flash. On Linux, programs and config live in this mounted tree. **MCU bridge:** Think of U-Boot like a much larger boot stub plus debug monitor: it initializes hardware, loads the next image, and gives you commands before Linux starts. **rootfs:** root filesystem, the directory tree mounted at / that contains /bin, /etc, /dev, and libraries. **U-Boot:** the bootloader that initializes enough hardware to load and start the Linux kernel.

126.1 The next-step canon

Five books and sites that pick up where this book ends:

1. Linux Device Drivers, 3rd Ed. (LDD3), Corbet, Rubini, Kroah-Hartman

The canonical kernel-driver book. Written for kernel 2.6 (so the specific APIs are dated), but the *concepts* are timeless, char devices, blocks, networking drivers, USB drivers, PCI, sleeping primitives, kernel memory management. The mental model it gives you is correct. Just check current APIs in the kernel source when implementing.

Free online: https://lwn.net/Kernel/LDD3/

2. Bootlin's "Embedded Linux" training material

Bootlin (formerly Free Electrons) is a French embedded-Linux training + consulting company. Their training slides + labs are free, comprehensive, current, and license-permissive. Topics: kernel internals, Yocto, Buildroot, audio, video, real-time, security.

> **Yocto:** a metadata-driven build system for producing custom Linux distributions. **Buildroot:** a configuration-driven build system that produces a complete root filesystem and related images.

https://bootlin.com/training/

Probably the best free online resource for further study.

3. `kernelnewbies.org`

Wiki + mailing list for new kernel contributors. Hosts:

- A “first patch” tutorial.
- Release notes for every kernel version since 2.6, read these to track what’s changing.
- “KernelHacking” tutorials.

http://kernelnewbies.org

The most welcoming community for first-time kernel contributors. Mailing-list lurkers welcome.

4. LWN.net

The professional-grade kernel-development periodical. A paid subscription ($10/month) is worth it if you work with Linux full-time. Coverage:

- Weekly summaries of LKML threads.
- Deep dives on new kernel features (eBPF, io_uring, sched_ext).
- Conference reports (Linux Plumbers, Kernel Summit, Open Source Summit).
- Subsystem maintainer profiles.

Reading LWN weekly is one of the most reliable ways to absorb kernel-development culture and stay current.

https://lwn.net/

5. `Documentation/process/` in the Linux source tree

The kernel ships its own onboarding documentation. Read in order:

- `1.Intro.rst`: “the kernel project is huge”
- `2.Process.rst`: “how the release cycle works”
- `3.Early-stage.rst`: “what to do before sending a patch”
- `4.Coding.rst`: “the code style and norms”
- `5.Posting.rst`: “how to send patches (Ch 120A’s bible)”
- `6.Followthrough.rst`: “how to respond to reviews”

- `7.AdvancedTopics.rst`: “everything else”

200 pages total. Every new contributor should read it once.

126.2 Communities and conferences

- **Linux Plumbers Conference** (annual. ~Sep–Oct), the developer’s conference. Streamed talks free.
- **Embedded Linux Conference** (Linux Foundation. Spring + fall), embedded-specific.
- **FOSDEM** (Brussels, Feb), free, sprawling, every open-source project under one roof.
- **Open Source Summit** (Linux Foundation, multiple cities), broader open-source.
- **DebConf** (annual), Debian-specific. Many embedded developers attend.

Local: search for “Linux meetup” or “embedded systems meetup” in your city. Bring questions. The community is welcoming.

126.3 Mailing lists

Subscribe (digest mode if volume is too high):

- **linux-kernel@vger.kernel.org**: the firehose. ~500 mails/day. Lurk.
- **linux-arm-kernel@lists.infradead.org**: ARM-specific. ~50/day. Relevant to you.
- **linux-imx@nxp.com**: NXP-imx-specific (moderated). ~5/day. Very relevant.
- **devicetree@vger.kernel.org**: DT bindings discussions.
- **stable@vger.kernel.org**: stable-kernel patch announcements.
- **kernelnewbies@kernelnewbies.org**: beginner Q&A.

Lurk for a month before posting. Read the patches you can understand. Ignore the rest. Slowly your understanding accretes.

126.4 Three further specializations

Depending on your interests:

Path A, Kernel hacker

Goal: become a subsystem maintainer.

- Pick a subsystem (e.g., IIO sensors, GPIO, regulators).

> **MCU bridge:** Think of Linux GPIO like the same pin set/reset block you used on STM32, but accessed through a kernel subsystem that owns numbering, direction, interrupts, and user-space exposure. **IIO:** Industrial I/O, Linux's subsystem for sensors, ADCs, DACs, and buffered sampled data. **GPIO:** General-Purpose Input/Output, a pin controlled as a digital input, output, or interrupt source.

- Read every patch on that subsystem's mailing list for 6 months.
- Submit small fixes. Build reputation.
- Eventually: maintain a driver, then a subsystem.

Path B, Product engineer

Goal: ship great embedded products.

- Master one BSP family deeply (i.MX, STM32MP, TI Sitara, RPi).

> **BSP:** Board Support Package: vendor patches, configs, bootloader files, and scripts needed to boot one board.

- Build the CI/release/OTA infrastructure (Ch 121A, 125).
- Develop soft skills: talking to product managers, defending engineering trade-offs.
- Eventually: tech lead on a product team.

Path C, Embedded security

Goal: build trustworthy embedded systems.

- Deepen on TrustZone + OP-TEE.
- Study secure boot in depth across multiple SoCs.
- Learn cryptography (Boneh's online course is excellent).
- Eventually: security architect. Review designs across products.

126.5 Skills outside Linux you'll still need

- **C and C++**: kernel is C. Many embedded apps are C++.
- **Python**: for tooling, build scripts, test automation.
- **Rust**: increasingly relevant. Learn it for new kernel contributions and for high-reliability user-space.

- **Git**: beyond `add/commit/push`. Understand rebase, bisect, blame, format-patch, send-email.
- **Make + CMake + Meson**: build system fluency saves hours.
- **Networking**: TCP/IP at the implementation level, not just usage. RFC 793 / 1122.
- **Shell scripting**: `bash`, `awk`, `sed`. Indispensable.
- **Schematic / PCB reading**: you may never design boards, but you'll read thousands.
- **Public speaking and writing**: explaining technical decisions to non-engineers is a force multiplier.

126.6 Key Notes

- Every layer needs an owner and a contract: what it receives, what it changes, what it promises, and how failure is reported.
- Device Tree describes hardware relationships, while drivers and frameworks implement behavior. A matching node is not proof that electrical or timing assumptions are correct.
- Update design must preserve a recoverable state through interruption and incompatible changes. A/B is one useful mechanism, not the only possible strategy.
- Source versions and board provenance are part of an implementation's meaning. Vendor support and upstream support should be assessed against actual hardware coverage and maintenance needs.
- Secure boot, provisioning, runtime authority and updates protect different boundaries. None alone establishes a secure product.
- Implementation tools can generate code, but design judgment still has to account for dependencies, failure modes and evidence from the real system.

126.7 Acknowledgements

(Reserved for the book author's final acknowledgements: the people who made the journey possible, the colleagues who reviewed early drafts, the readers who reported errors, the maintainers whose code makes this possible at all.)

126.8 Errata + feedback

(Reserved: URL of the book's errata page. An email or GitHub issues tracker for reader-submitted corrections.)

> **End of the main i.MX6ULL path.**

An embedded system is easier to reason about when a failure can be traced to a particular responsibility. That is the thread joining the bootloader, kernel, rootfs and application chapters. The details will change, but questions about ownership, interfaces and recovery will remain useful.

The optional next path, **Part IX: Applied virtualization and mixed-criticality systems**, asks when those responsibilities should be separated more strongly. It uses QEMU models, Xen, the i.MX6ULL’s limits, Jailhouse and STM32MP1’s Linux/RTOS split.

Chapter 127: Why embedded products use hypervisors

> **What:** decide whether an embedded Linux product needs a hypervisor, an RTOS companion core, TrustZone, containers, PREEMPT_RT, or just one well-designed Linux system.
>
> **Why:** HYP mode is a CPU feature, not a product requirement. A hypervisor is useful only when it creates a boundary the product actually needs.
>
> **Focus:** name the failure you are trying to contain. Then choose the smallest boundary that contains it. **MCU bridge:** Think of a hypervisor like a bootloader that never leaves. It stays below operating systems, owns the machine boundary, and decides which OS sees which memory, interrupts, and devices. **Hypervisor:** software running at the CPU virtualization privilege level. It creates and controls one or more operating-system guests. **Guest:** an operating system running under a hypervisor.

127.1 The wrong way to start

The wrong question:

```
Can this Cortex-A7 run Xen or Jailhouse?
```

That question is not useless, but it is too early.

The right first question:

```
What must survive when something else fails?
```

If the answer is “nothing special, reboot the product”, you may not need a hypervisor.

If the answer is “the fieldbus must continue while the UI restarts”, now there is a boundary to design.

If the answer is “private keys must remain protected even if Linux is compromised”, that may be TrustZone, not Xen.

If the answer is “the motor loop needs 20 us jitter”, that may be an MCU core, not a Linux guest.

The product failure decides the architecture.

127.2 What a hypervisor is not

A hypervisor is not:

- a faster Linux,
- a real-time patch,
- a security sticker,
- a replacement for watchdog design,
- a substitute for Device Tree discipline,
- a way to make one CPU behave like two physical CPUs,
- a shortcut around understanding boot and memory maps.

On a small SoC, a hypervisor usually makes the system harder before it makes anything better.

That is fine when the boundary is worth it. It is waste when the boundary is imaginary.

127.3 The failure-containment question

The following cases illustrate how a failure model can suggest a boundary. They are design candidates, not guarantees of containment on every platform:

Failure	Must keep running	Allowed response	Boundary needed
UI process crash	field control	restart UI process	process supervisor
web stack compromised	private keys	deny key access	TrustZone or separate security chip
display driver crashes kernel	safety monitor	keep monitor alive	hypervisor or MCU core
fieldbus task blocks	cloud upload	restart fieldbus task	process or service boundary
old vendor BSP stuck on 4.1	new app stack	run both temporarily	Xen-style VM boundary

Failure	Must keep running	Allowed response	Boundary needed
motor loop jitter too high	motor control	move loop off Linux	MCU/RTOS or dedicated partition

This table often kills a bad hypervisor idea early, which is a win.

127.4 Boundaries available to us

There are several ways to separate work.

Process boundary

One Linux kernel. Multiple processes.

```
Linux kernel
  app_ui
  app_network
  app_fieldbus
```

Use this when:

- one kernel can own all devices,
- a process crash is enough isolation,
- systemd/runit/supervision can restart services,
- security requirements fit Linux users, groups, seccomp, and MAC.

This is the default for many products.

Container boundary

One Linux kernel. Multiple packaged user spaces.

```
Linux kernel
  container: UI
  container: gateway service
  container: update agent
```

Use this when:

- deployment packaging matters,
- services need separate filesystem views,
- the kernel is trusted,

- device ownership remains centralized.

Avoid this when:

- you need a different kernel,
- a kernel crash must be contained,
- a hostile driver is in scope.

Containers are not virtual machines. They share the host kernel.

PREEMPT_RT boundary

One Linux kernel with better scheduling latency.

```
Linux + PREEMPT_RT
  realtime thread
  normal services
```

Use this when:

- Linux must respond faster,
- device drivers can be made RT-friendly,
- the timing budget is milliseconds or low hundreds of microseconds,
- security isolation is not the primary problem.

Avoid this when:

- you need hard MCU-style timing,
- the control loop is small enough for an M-core,
- a Linux kernel fault must not stop control.

PREEMPT_RT improves latency. It does not isolate faults.

TrustZone boundary

One CPU complex, two worlds:

```
Secure World: OP-TEE, keys, secure services
Normal World: Linux
```

Use this when:

- keys must survive Linux compromise,
- secure boot must chain into runtime secrets,

- a small trusted service is enough.

Avoid this when:

- you want two normal operating systems,
- you want a rich RTOS beside Linux,
- the codebase in Secure World would become large.

TrustZone is for trusted services. It is not a general two-Linux mechanism.

remoteproc/OpenAMP boundary

Linux on an A-core. RTOS or firmware on an M-core.

```
Cortex-A7: Linux
Cortex-M4: Zephyr or FreeRTOS
communication: RPMsg/OpenAMP
```

Use this when:

- the SoC has an MCU companion core,
- control logic fits there,
- message passing is acceptable,
- Linux should supervise firmware load and recovery.

This is the natural STM32MP157 pattern.

Xen boundary

Hypervisor first, then virtual machines:

```
Xen
  Dom0 Linux: control domain
  DomU Linux: guest
```

Use this when:

- you need multiple Linux systems,
- a legacy BSP must be contained,
- guest lifecycle matters,
- domains may be restarted independently,
- paravirtual console, block, and network devices help.

Avoid this when:

- one Linux is enough,
- static device partitioning is all you need,
- RAM is too small,
- boot complexity is not justified.

Xen is the right tool for “two Linux systems” in this part.

Jailhouse boundary

Linux boots first, then gives hardware away:

```
Linux root cell
Jailhouse
  inmate cell: CPU + RAM + device
```

Use this when:

- you have spare CPU cores,
- cells can be statically partitioned,
- the isolated workload is small,
- you want a bare-metal or RTOS inmate.

Avoid this when:

- the SoC has one CPU,
- guests need dynamic resource sharing,
- you want rich virtual devices,
- the device ownership table is messy.

Jailhouse is not a scheduler. It is a partitioner.

127.5 Why i.MX6ULL is still worth using

The i.MX6ULL is not a perfect virtualization board:

- one Cortex-A7 core,
- small RAM budget,
- no Cortex-M companion,

- no spare A-core for Jailhouse-style partitioning.

But it is still useful:

- ARMv7-A mode structure is visible,
- HYP mode is real on this Cortex-A7,
- U-Boot handoff is real,
- Device Tree is real,
- Xen Dom0 and DomU concepts can be demonstrated,
- the board’s limitations teach honest architecture.

We do not use i.MX6ULL to pretend a small single-core SoC is a cloud host. We use it because the whole book already built this board from reset vector upward. Seeing Xen on the same board completes the mental map.

127.6 Why QEMU comes first

QEMU gives us a clean failure surface.

With QEMU:

- reset is instant,
- boot logs are easy to capture,
- memory size is configurable,
- CPU count is configurable,
- no SD card is corrupted,
- no board is wedged,
- GDB can attach before the first instruction we care about.

That is why the path is:

```
QEMU plain Linux
QEMU U-Boot
QEMU Xen
i.MX6ULL Xen
QEMU Jailhouse ARM64
STM32MP1 Linux + RTOS
```

QEMU teaches the pattern. Real hardware teaches the cost.

127.7 Why STM32MP1 appears in an i.MX6ULL book

The book’s main board is i.MX6ULL, but the question “Linux + RTOS” deserves an honest answer.

On i.MX6ULL:

```
one Cortex-A7
Linux and RTOS must share the same CPU if both run there
```

On STM32MP157:

```
dual Cortex-A7 + Cortex-M4
Linux can run on A7
RTOS can run on M4
```

That is a better production split for many products.

So Part IX says:

- Xen on i.MX6ULL is a real HYP-mode experiment.
- Jailhouse on i.MX6ULL is not a good first target.
- Jailhouse in QEMU ARM64 is a good partitioning study.
- STM32MP1 Linux + M4 RTOS is the realistic Linux+RTOS product pattern.

That distinction matters. It prevents the reader from forcing one board to teach every lesson.

127.8 Case study: field gateway

Product:

- Ethernet or LTE uplink,
- RS-485 Modbus fieldbus,
- local web UI,
- OTA updates,
- must keep polling field devices even if UI crashes.

Bad first answer

```
Run everything in one process.
```

One crash kills everything.

Better simple answer

```
One Linux
  systemd service: fieldbusd
  systemd service: webui
  systemd service: updater
```

If `webui` crashes, `fieldbusd` continues.

This may be enough.

Container answer

```
One Linux
  container: web UI
  host service: fieldbusd
  host service: updater
```

Use this if packaging and update separation are important.

Hypervisor answer

```
Xen
  Dom0: fieldbus and watchdog
  DomU: web UI and cloud stack
```

Use this only if the web/cloud stack is risky enough that a kernel or driver fault must be contained.

Likely decision

Start with one Linux and separate services. Move to Xen only if field evidence or security requirements justify a kernel boundary.

127.9 Case study: industrial HMI with safety monitor

Product:

- touchscreen UI,
- display stack,
- data logging,
- safety output,
- watchdog,

- remote update.

Display stacks are large. UI code changes often. Safety logic should be small and boring.

One-Linux design

```
Linux
  UI process
  safety-monitor process
  watchdog daemon
```

This is easy but weak. A kernel/display-driver failure can still stop safety logic.

TrustZone design

```
Secure World: small safety or key service
Normal World: Linux UI
```

This is suitable only if the secure part is tiny. Do not put a large control application into Secure World.

MCU companion design

```
A-core Linux: UI, display, logging
M-core RTOS: safety monitor, watchdog policy, safety output
```

This is often best if the SoC has an M-core.

Xen design

```
Xen
  Dom0: safety supervisor, watchdog, selected hardware
  DomU: UI Linux
```

This can work when the safety workload needs an A-core or when no M-core exists.

Likely decision

On STM32MP157, use the M4 for the safety monitor if it fits. On i.MX6ULL, consider Xen only if a separate safety microcontroller is not available and the risk justifies the complexity.

127.10 Case study: legacy BSP containment

Product:

- old vendor camera stack works only on Linux 4.1,

- product application wants Linux 6.x,
- hardware cannot change this generation,
- camera feature must ship.

This is one of the clearest Xen-style cases.

Possible architecture:

```
Xen
  Dom0 Linux 6.x: networking, update, storage, product app
  DomU Linux 4.1: camera vendor stack
shared channel: frame transfer or network
```

This design is not pretty, but it may isolate the old BSP while the rest of the product moves forward.

Questions:

- Who owns the camera hardware?
- Does old Linux need direct device access?
- Can frames cross domains safely?
- How is the old guest updated?
- Can the product tolerate the RAM cost?
- What is the exit plan for removing the old guest?

Hypervisors are often used as migration scaffolding. Scaffolding should not become architecture forever.

127.11 Case study: motor control

Product:

- user interface,
- Ethernet,
- motor-control loop,
- hard fault input,
- PWM outputs,
- current feedback.

If the control loop needs tight timing, do not start with Xen.

Likely order:

1. Dedicated motor-control MCU.
2. SoC M-core with RTOS.
3. Bare-metal or RTOS partition on a spare A-core.
4. PREEMPT_RT only if measured latency fits.
5. Xen only if the problem is OS isolation, not microsecond control.

Reason:

```
Hypervisor isolation != hard real-time control
```

The hypervisor can isolate worlds. It does not make Linux a motor-control MCU.

127.12 Device ownership decides architecture

Before writing code, make a device table:

Device	Owner	Shared?	DMA?	Reset owner	Notes
UART console	Dom0 or Xen	no	no	Dom0	debugging lifeline
Ethernet	Dom0	virtual net to guest	yes	Dom0	do not pass through first
eMMC/SD	Dom0	virtual block to guest	yes	Dom0	rootfs and updates
Watchdog	Dom0 or safety core	no	no	safety policy	must be explicit
GPIO bank	one owner	avoid	no	owner	banks contain mixed pins
I2C controller	one owner	proxy	maybe	owner	bus has many devices

Device	Owner	Shared?	DMA?	Reset owner	Notes
Display	UI domain	no	yes	UI owner	large driver stack

If the table is confusing, the product will be confusing.

127.13 Watchdog policy

Ask this early:

```
Who pets the watchdog?
```

Possible answers:

- Linux init system,
- Dom0 watchdog daemon,
- hypervisor-aware watchdog service,
- M-core safety firmware,
- external supervisor MCU.

Bad answer:

```
Everyone pets it.
```

If every world can pet the watchdog, one broken world can hide another broken world.

A useful policy says:

- what “healthy” means,
- who reports health,
- who aggregates health,
- who pets the hardware watchdog,
- what happens when DomU dies,
- what happens when Dom0 dies,
- what gets logged before reset.

127.14 Update policy

Virtualization changes updates.

One Linux product:

```
A/B rootfs update
rollback if boot fails
```

Xen product:

```
Xen binary
Dom0 kernel
Dom0 rootfs
DomU kernel
DomU rootfs
guest configs
shared protocol version
rollback rules
```

Linux + M4 product:

```
Linux image
M4 firmware
RPMsg protocol version
resource table
rollback compatibility
```

Every extra world is another artifact to version, sign, test, and roll back.

This is one reason “no hypervisor” is often the professional answer.

127.15 Security boundary sanity check

A boundary is only useful if the attack path cannot simply walk around it.

Ask:

- Can this domain program DMA into another domain’s RAM?
- Can this domain reconfigure clocks or resets for another domain’s device?
- Can this domain change pinmux?
- Can this domain access the update partition for another domain?
- Can this domain pet the global watchdog?
- Can this domain read secrets?
- Can this domain write bootloader environment?

If yes, write down why that is acceptable or fix the design.

127.16 Choosing between Xen and Jailhouse

Use Xen when the product wants virtual machines:

- multiple Linux systems,
- Dom0 control,
- guest lifecycle,
- paravirtual console/block/network,
- migration away from old BSPs,
- guest restart as a normal operation.

Use Jailhouse when the product wants static partitions:

- one Linux root cell,
- one or more isolated inmate cells,
- dedicated CPUs,
- dedicated memory,
- dedicated devices,
- no rich scheduling or virtual device model.

Quick rule:

```
If you want "two Linux systems", think Xen.
If you want "Linux plus a dedicated bare-metal/RTOS island", think Jailhouse or M-core.
```

On i.MX6ULL, use Xen for the real-board experiment. Use QEMU ARM64 for the first Jailhouse experiment.

127.17 The Part IX path

The chapters are ordered to avoid magic:

Chapter	Result
128	QEMU starts, GDB attaches, DTB is visible

Chapter	Result
129	Tiny ARM Linux boots directly in QEMU
130	U-Boot boots that Linux in QEMU
131	HYP, stage-2 MMU, virtual IRQs explained
132	Xen boots Dom0 in QEMU
133	DomU Linux starts in QEMU
134	Xen boots Dom0 on i.MX6ULL
135	DomU Linux starts on i.MX6ULL
136	Device, memory, DMA ownership is analyzed
137	Jailhouse starts an inmate in QEMU ARM64
138	Bare-metal or Zephyr inmate runs
139	STM32MP1 Linux + M4 RTOS pattern is built

The sequence is deliberate. We do not start by throwing Xen at a real board and hoping the serial log is kind.

127.18 Key Notes

- The choice begins with a failure to contain and the workload that must survive it. A process, container, guest, partition or companion MCU provides a different boundary.
- CPU count, memory, timing and device dependencies limit feasible designs. A hypervisor does not create extra physical cores or independent power domains.
- CPU memory protection and device DMA protection are separate mechanisms. Assigning MMIO alone is not a complete isolation argument.

- Watchdog and reset policy must identify whose health matters and who can recover the system. Independent watchdog petting can hide a failed workload.
- Separate execution worlds create separate update and compatibility responsibilities. Recovery must account for their shared interfaces and management domain.
- The smallest adequate boundary is usually easier to maintain than a stronger boundary with unexplained dependencies.

127.19 Design cases

Case A: field gateway

A gateway with an uplink, fieldbus, web UI and OTA needs polling to recover when its UI fails. Separate Linux services may contain that process failure. Containers can help package them independently, but still share the kernel. Xen becomes relevant when the required boundary includes a failing kernel or a legacy stack.

Case B: industrial HMI with safety monitor

Display and logging failures should not silently disable a safety function. A companion core or external controller can separate timing and recovery responsibilities, but its peripherals, power and communication links still need analysis. Neither an MCU nor a hypervisor makes the whole product safety-qualified by its presence.

Case C: legacy BSP containment

A legacy camera stack and a newer application stack may justify separate kernels. Xen is a candidate when hardware support, memory and device ownership permit it. Containment can reduce migration coupling, while preserving the maintenance and security obligations of the old stack.

127.20 What makes the decision defensible

An architecture explanation connects each failure to the boundary intended to contain it, the resources that boundary owns, and the recovery that remains available. It also identifies shared dependencies: management software, DMA, clocks, reset, power and persistent state.

A rejected alternative is useful when its tradeoff is explicit. A design is not justified merely by naming Xen, an RTOS or isolation. Measurements and fault evidence are separate from that explanation, and must eventually test the actual requirement.

127.21 Pitfalls

- **Starting with the tool.** Start with the failure boundary.
- **Mistaking scheduling for isolation.** PREEMPT_RT improves latency. It does not isolate a malicious driver.
- **Mistaking containers for virtual machines.** Containers share the host kernel.
- **Ignoring DMA.** CPU page tables do not automatically stop a device from writing memory.
- **Ignoring watchdog policy.** A watchdog with no owner is just a timer.
- **Ignoring updates.** Two worlds mean two version histories and two rollback plans.
- **Overusing TrustZone.** Secure World should stay small.
- **Overusing the M-core.** The M4 is useful, but it is not a tiny Linux machine.
- **Overusing Xen.** Xen is powerful, but one Linux is better when one Linux is enough.
- **Overusing Jailhouse.** Jailhouse needs clean static partitioning and spare CPUs.

127.22 Going deeper

- QEMU Arm `virt` machine documentation.
- Xen Project ARM documentation.
- Jailhouse README and QEMU ARM64 demo notes.
- Linux remoteproc documentation.
- Linux RPMsg documentation.
- OpenAMP documentation.
- STM32MP1 reference manuals.
- Zephyr STM32MP157 board documentation.

Chapter 128: QEMU as a virtual hardware model

> **What:** install QEMU, create 32-bit and 64-bit Arm virtual machines, inspect QEMU's generated Device Tree, and attach GDB before any kernel exists.
>
> **Why:** hypervisor bring-up fails before Linux can help you. QEMU gives us a board we can reset in one second, script exactly, and inspect from the outside.
>
> **Focus:** a QEMU command line is a hardware description. Treat it like a schematic: CPU, RAM, UART, interrupt controller, timer, storage, and firmware are all explicit choices. **QEMU:** a machine emulator. `qemu-system-arm` creates a complete Arm machine, not just one process. **virt machine:** QEMU's generic virtual Arm board. It is designed for virtual machines, not for matching one physical board. **GDB remote stub:** a tiny debug server built into QEMU that lets GDB halt and inspect the emulated CPU.

128.1 What QEMU is, and what it is not

QEMU can emulate several kinds of things:

Program	Meaning
`qemu-system-arm`	emulates a complete 32-bit Arm machine
`qemu-system-aarch64`	emulates a complete 64-bit Arm machine, and can also run some 32-bit Arm machines
`qemu-arm`	runs one Arm Linux user-space program on the host

This chapter uses **system emulation**:

```
host Linux
  qemu-system-arm
    virtual CPU
    virtual RAM
    virtual UART
    virtual GIC
    virtual timer
    generated DTB
```

It does not emulate the i.MX6ULL. The QEMU `virt` board gives us a simpler Arm hardware model for examining Linux and hypervisor boot contracts. Its successful boot cannot establish compatibility with the MINI’s controllers or wiring.

What QEMU `virt` gives us:

- configurable RAM,
- configurable CPU count,
- GIC interrupt controller,
- ARM architectural timer,
- PL011 UART,
- virtio devices,
- generated Device Tree,
- a platform supported by Linux, U-Boot, Xen, and Jailhouse examples.

What it does **not** give us:

- i.MX6ULL IOMUXC,
- MMDC DDR training,
- FEC Ethernet,
- SNVS,
- OCOTP fuses,
- HAB,
- the Point Atom board schematic.

Use QEMU to learn the contract. Use i.MX6ULL to learn the board.

128.2 Install tools

On Ubuntu or Debian:

```
$ sudo apt update
$ sudo apt install qemu-system-arm qemu-system-aarch64 gdb-multiarch \
    device-tree-compiler u-boot-tools file
```

Create the workspace:

```
$ mkdir -p ~/imx6ull/virt-lab/{boot,logs,notes}
$ cd ~/imx6ull/virt-lab
```

Record tool versions:

```
$ qemu-system-arm --version | tee logs/qemu-system-arm-version.txt
$ qemu-system-aarch64 --version | tee logs/qemu-system-aarch64-version.txt
$ gdb-multiarch --version | head -1 | tee logs/gdb-version.txt
$ dtc --version | tee logs/dtc-version.txt
```

hypervisor examples are version-sensitive. Keep the versions with the logs.

128.3 Start a 32-bit Arm board with no kernel

Run:

```
$ qemu-system-arm \
    -M virt \
    -cpu cortex-a15 \
    -m 256M \
    -nographic
```

Expected behavior: not much. There is no kernel, no firmware, and no bootloader. QEMU created a machine and then had nothing useful to execute.

Quit:

```
Ctrl-a x
```

In `-nographic` mode, `Ctrl-a` is QEMU's escape prefix. `x` exits.

Now write down what each option means:

Option	Meaning
`-M virt`	create the generic Arm virtual board
`-cpu cortex-a15`	use a 32-bit ARMv7-A CPU model
`-m 256M`	give the guest 256 MiB RAM
`-nographic`	route guest serial and QEMU monitor through the terminal

The machine did not boot, but this was not a failure. It proved QEMU is installed and the virtual board can be created.

128.4 Start a 64-bit Arm board

The later Jailhouse example needs spare CPUs. ARM64 QEMU supplies a multicore model:

```
$ qemu-system-aarch64 \
    -M virt \
    -cpu cortex-a53 \
    -smp 4 \
    -m 1024M \
    -nographic
```

Quit with `Ctrl-a x`.

New option:

Option	Meaning
`-smp 4`	create four virtual CPUs

The i.MX6ULL cannot give us four A-cores. QEMU can. That is why Jailhouse appears first on QEMU ARM64 instead of on i.MX6ULL.

128.5 Ask QEMU what machines it knows

List supported 32-bit Arm machines:

```
$ qemu-system-arm -machine help | less
```

Search for `virt`:

```
$ qemu-system-arm -machine help | grep virt
```

List CPU models:

```
$ qemu-system-arm -cpu help | less
```

This is the QEMU equivalent of opening a board catalog. We choose `virt` because it is generic and maintained for virtual machine use.

128.6 Dump the generated Device Tree

QEMU can generate the DTB for the virtual board and write it to a file:

```
$ cd ~/imx6ull/virt-lab
$ qemu-system-arm \
    -M virt,dumpdtb=boot/qemu-virt-arm.dtb \
    -cpu cortex-a15 \
    -m 256M \
    -nographic
```

If QEMU sits after dumping, quit with `Ctrl-a x`.

Convert to DTS:

```
$ dtc -I dtb -O dts -o boot/qemu-virt-arm.dts boot/qemu-virt-arm.dtb
```

Inspect:

```
$ less boot/qemu-virt-arm.dts
```

Find the root compatible:

```
$ grep -n "compatible" boot/qemu-virt-arm.dts | head
```

Find memory:

```
$ grep -n "memory@" boot/qemu-virt-arm.dts
```

Find CPUs:

```
$ grep -n "cpu@" boot/qemu-virt-arm.dts
```

Find serial:

```
$ grep -n -A6 -B2 "pl011" boot/qemu-virt-arm.dts
```

Find interrupt controller:

```
$ grep -n -A10 -B2 "interrupt-controller" boot/qemu-virt-arm.dts | head -40
```

Find timer:

```
$ grep -n -A10 -B2 "timer" boot/qemu-virt-arm.dts | head -40
```

This DTB is the “schematic” Linux will read in Chapter 129.

128.7 Prove RAM size changes the DTB

Dump a second DTB with 512 MiB:

```
$ qemu-system-arm \
    -M virt,dumpdtb=boot/qemu-virt-arm-512m.dtb \
    -cpu cortex-a15 \
    -m 512M \
    -nographic
```

Convert:

```
$ dtc -I dtb -O dts -o boot/qemu-virt-arm-512m.dts boot/qemu-virt-arm-512m.dtb
```

Compare memory nodes:

```
$ grep -n -A4 "memory@" boot/qemu-virt-arm.dts
$ grep -n -A4 "memory@" boot/qemu-virt-arm-512m.dts
```

The hardware description changed because the virtual hardware changed. This is exactly what happens on real boards too, just with a DTS file maintained by humans instead of generated by QEMU.

128.8 Halt the virtual CPU and attach GDB

Start QEMU halted:

```
$ qemu-system-arm \
    -M virt \
    -cpu cortex-a15 \
    -m 256M \
    -nographic \
    -S -s
```

Options:

Option	Meaning
`-S`	create the machine but do not start the CPU
`-s`	open GDB server on TCP port 1234

In another terminal:

```
$ gdb-multiarch
(gdb) set architecture arm
(gdb) target remote :1234
(gdb) info registers
(gdb) x/8i $pc
```

Save the transcript:

```
(gdb) set logging file logs/gdb-qemu-no-kernel.txt
(gdb) set logging enabled on
(gdb) info registers
(gdb) x/8i $pc
(gdb) set logging enabled off
```

Continue:

```
(gdb) c
```

Stop QEMU with `Ctrl-a x`.

There is still no kernel. That is the point. We have proven that GDB can attach before any operating system exists.

128.9 Use the QEMU monitor

Start QEMU with a monitor on stdio:

```
$ qemu-system-arm \
    -M virt \
    -cpu cortex-a15 \
    -m 256M \
    -nographic \
    -monitor stdio
```

Try:

```
(qemu) info version
(qemu) info registers
(qemu) info mtree
(qemu) quit
```

`info mtree` is useful later. It shows QEMU’s memory map from the emulator side.

Save a copy:

```
(qemu) log none
```

for most examples we use the serial console, not the monitor, but knowing the monitor exists is useful when a guest is silent.

128.10 The virtual board memory map

Use:

```
(qemu) info mtree
```

Look for:

- RAM,
- UART MMIO,
- GIC MMIO,
- virtio MMIO,
- flash or firmware regions if present.

This is the QEMU-side equivalent of a reference manual memory map. In Chapter 129, Linux will show its view through `/proc/iomem`. Comparing the two is a good sanity check.

128.11 What to save

After this chapter, your workspace should contain:

```
boot/
  qemu-virt-arm.dtb
  qemu-virt-arm.dts
  qemu-virt-arm-512m.dtb
  qemu-virt-arm-512m.dts
logs/
  qemu-system-arm-version.txt
  qemu-system-aarch64-version.txt
  gdb-version.txt
  dtc-version.txt
  gdb-qemu-no-kernel.txt
notes/
  qemu-command-line.md
```

Create `notes/qemu-command-line.md`:

```
# QEMU command line notes

<span id="bit-arm-lab-machine"></span>
## 32-bit ARM model

Command:
```

```
...

Options:

- `-M virt`:
- `-cpu cortex-a15`:
- `-m 256M`:
- `-nographic`:

## Hardware visible in DTB

- RAM:
- UART:
- interrupt controller:
- timer:
- CPU count:
```

If you cannot fill that note, do not go to Chapter 129 yet.

128.12 Key Notes

- A QEMU launch describes a virtual machine: CPU, memory, interrupt controller, devices and firmware are explicit inputs. Its generated DTB describes that selected model.
- System emulation and user-mode emulation have different boundaries. `qemu-system-arm` models a machine, while `qemu-arm` executes an ARM user-space program using host services.
- The `virt` machine is not an i.MX6ULL replica. It exposes boot and virtualization contracts without qualifying MINI pinmux, DDR, clocks or electrical behavior.
- Halting the virtual CPU and providing a GDB connection are independent controls. They make startup inspectable without making the model identical to physical timing.
- Machine parameters, artifact versions and logs are reproducibility evidence. Changing memory or devices changes the system being described.

128.13 Pitfalls

- **Expecting i.MX6ULL devices.** `virt` will not have IOMUXC, MMDC, FEC, SNVS, or OCOTP.
- **Using user-mode QEMU by accident.** `qemu-arm` is not a machine.
- **Forgetting `Ctrl-a x`.** Closing the terminal is a clumsy reset button.
- **No GDB architecture selected.** Use `set architecture arm` before inspecting a 32-bit target.

- **Treating generated DTB as permanent.** It describes the QEMU command line you used. Change the command line, regenerate it.
- **Skipping notes.** Later Xen failures often reduce to “I forgot what machine I built.”

128.14 Going deeper

- QEMU Arm system emulator documentation.
- QEMU `virt` machine documentation.
- QEMU monitor documentation.
- GDB remote debugging documentation.
- U-Boot QEMU ARM documentation.

Chapter 129: Build and boot tiny Linux in QEMU

> **What:** build a 32-bit ARM Linux kernel, build a static BusyBox initramfs, generate a QEMU Device Tree, and boot to a shell in QEMU.
>
> **Why:** before Xen, Dom0, DomU, or Jailhouse enter the story, the reader must understand a plain ARM Linux boot with no hidden board support package. Kernel, DTB, initramfs, command line, console. Those are the pieces.
>
> **Focus:** kernel, Device Tree and initramfs are separate boot inputs with different owners. QEMU makes their interfaces easier to inspect. A failure in one input can produce symptoms in another layer, so the boot contract matters more than the command sequence. **QEMU:** a machine emulator. Here it creates a fake ARM board with RAM, CPU, UART, timer, interrupt controller, and virtio devices. **DTB:** Device Tree Blob, the binary hardware description passed to the Linux kernel. **initramfs:** a cpio archive the kernel unpacks into RAM and runs as the first root filesystem.

129.1 What we are about to build

The boot path is:

```
qemu-system-arm
  -> creates an ARMv7-A "virt" board
  -> loads zImage into guest RAM
  -> loads qemu-virt-arm.dtb into guest RAM
  -> loads initramfs.cpio.gz into guest RAM
  -> starts the kernel
Linux
  -> reads the DTB
  -> initializes the PL011 UART
  -> unpacks the initramfs
  -> runs /init
/init
  -> mounts /proc, /sys, /dev
  -> gives us a BusyBox shell
```

No U-Boot. No Xen. No Dom0. No virtual disks.

We remove those layers on purpose. If this chapter fails, the failure is in one of four inputs:

- kernel image,
- DTB,

- initramfs,
- command line.

That is the mental model we want before Chapter 130 adds U-Boot and Chapter 132 adds Xen.

129.2 Host packages

On Ubuntu or Debian:

```
$ sudo apt update
$ sudo apt install build-essential git bc bison flex libssl-dev \
    gcc-arm-linux-gnueabihf qemu-system-arm device-tree-compiler \
    cpio gzip xz-utils rsync file wget
```

Check the tools:

```
$ arm-linux-gnueabihf-gcc --version | head -1
$ qemu-system-arm --version | head -1
$ dtc --version
```

This chapter uses Debian/Ubuntu's `arm-linux-gnueabihf-` toolchain name. If you prefer the book's project-local Arm GNU toolchain, set:

```
$ export CROSS_COMPILE=arm-none-linux-gnueabihf-
```

Otherwise:

```
$ export CROSS_COMPILE=arm-linux-gnueabihf-
```

Everything below uses `$CROSS_COMPILE`.

129.3 Workspace

```
$ mkdir -p ~/imx6ull/virt-lab/{src,build,rootfs,boot,logs}
$ cd ~/imx6ull/virt-lab
```

By the end, the important files will be:

```
boot/
  zImage
  qemu-virt-arm.dtb
  qemu-virt-arm.dts
  initramfs.cpio.gz
logs/
  qemu-linux-direct.log
```

```
  qemu-linux-wrong-console.log
  qemu-linux-no-initramfs.log
  qemu-linux-no-init.log
```

129.4 Build BusyBox first

We need a user space. The smallest useful one is a static BusyBox.

Download:

```
$ cd ~/imx6ull/virt-lab/src
$ wget https://busybox.net/downloads/busybox-1.36.1.tar.bz2
$ tar xf busybox-1.36.1.tar.bz2
$ cd busybox-1.36.1
```

Configure:

```
$ make ARCH=arm CROSS_COMPILE=$CROSS_COMPILE defconfig
```

Enable static linking:

```
$ scripts/config -e STATIC
$ make ARCH=arm CROSS_COMPILE=$CROSS_COMPILE olddefconfig
```

Build:

```
$ make ARCH=arm CROSS_COMPILE=$CROSS_COMPILE -j$(nproc)
```

Inspect the result:

```
$ file busybox
busybox: ELF 32-bit LSB executable, ARM, EABI5, statically linked, ...

$ ls -lh busybox
```

The important words are **ARM** and **statically linked**. If `file` says x86-64, you built for the host by accident. If it says dynamically linked, the initramfs will need shared libraries, which we are deliberately avoiding.

129.5 Install BusyBox into the rootfs tree

Use BusyBox’s installer. Do not create every applet symlink by hand.

```
$ rm -rf ~/imx6ull/virt-lab/rootfs/*
$ make ARCH=arm CROSS_COMPILE=$CROSS_COMPILE \
    CONFIG_PREFIX=~/imx6ull/virt-lab/rootfs install
```

Check:

```
$ cd ~/imx6ull/virt-lab/rootfs
$ ls
bin  linuxrc  sbin  usr

$ file bin/busybox
bin/busybox: ELF 32-bit LSB executable, ARM, EABI5, statically linked, ...

$ ls -l bin/sh
```

BusyBox should have installed `bin/sh` as a symlink to `busybox`.

Now add the directories Linux user space expects:

```
$ mkdir -p proc sys dev tmp run mnt root etc
$ chmod 1777 tmp
```

129.6 Write `/init`

The kernel's first user-space process will be `/init` because we will pass `rdinit=/init`.

Create it:

```
$ cat > init <<'EOF'
#!/bin/sh

mount -t proc none /proc
mount -t sysfs none /sys
mount -t devtmpfs none /dev

echo
echo "================================="
echo " QEMU tiny Linux is alive"
echo "================================="
echo

echo "[init] kernel:"
uname -a

echo
echo "[init] command line:"
cat /proc/cmdline

echo
echo "[init] CPU:"
grep -E 'Processor|model name|Hardware|Features' /proc/cpuinfo || true

echo
echo "[init] memory:"
head -5 /proc/meminfo

echo
echo "[init] device-tree compatible:"
```

```
tr '\0' '\n' < /sys/firmware/devicetree/base/compatible 2>/dev/null || true

echo
echo "[init] starting shell"
exec /bin/sh
EOF

$ chmod +x init
```

This script does three important mounts:

- `/proc`: kernel process and system information,
- `/sys`: device model and sysfs,
- `/dev`: device nodes from `devtmpfs`.

Without `/dev`, a real system quickly becomes annoying. Without `/proc` and `/sys`, you are blind.

129.7 Pack the initramfs

From inside the rootfs directory:

```
$ cd ~/imx6ull/virt-lab/rootfs
$ find . -print0 | cpio --null -ov --format=newc | gzip -9 > ../boot/initramfs.cpio.gz
```

Inspect the archive without unpacking it:

```
$ cd ~/imx6ull/virt-lab
$ ls -lh boot/initramfs.cpio.gz
$ gzip -dc boot/initramfs.cpio.gz | cpio -it | head -30
```

You must see:

```
.
bin
bin/busybox
bin/sh
init
proc
sys
dev
```

If `/init` is missing, the kernel cannot run it. If `bin/sh` is missing, `/init` will fail at the final `exec`.

129.8 Build Linux for QEMU ARM `virt`

Get Linux:

```
$ cd ~/imx6ull/virt-lab/src
$ git clone https://git.kernel.org/pub/scm/linux/kernel/git/stable/linux.git
$ cd linux
$ git checkout v6.6
```

Configure:

```
$ make ARCH=arm O=~/imx6ull/virt-lab/build/linux-qemu multi_v7_defconfig
```

Enable the pieces this boot needs:

```
$ cd ~/imx6ull/virt-lab/src/linux
$ scripts/config --file ~/imx6ull/virt-lab/build/linux-qemu/.config \
    -e DEVTMPFS \
    -e DEVTMPFS_MOUNT \
    -e BLK_DEV_INITRD \
    -e SERIAL_AMBA_PL011 \
    -e SERIAL_AMBA_PL011_CONSOLE \
    -e VIRTIO \
    -e VIRTIO_MMIO

$ make ARCH=arm O=~/imx6ull/virt-lab/build/linux-qemu olddefconfig
```

Build:

```
$ make ARCH=arm O=~/imx6ull/virt-lab/build/linux-qemu \
    CROSS_COMPILE=$CROSS_COMPILE -j$(nproc) zImage
```

Copy:

```
$ cp ~/imx6ull/virt-lab/build/linux-qemu/arch/arm/boot/zImage \
    ~/imx6ull/virt-lab/boot/zImage
```

Inspect:

```
$ file ~/imx6ull/virt-lab/boot/zImage
$ ls -lh ~/imx6ull/virt-lab/boot/zImage
```

`zImage` is a compressed ARM kernel image. It is not an ELF file you can run on the host.

129.9 Get QEMU’s Device Tree

The `virt` machine generates its own hardware description. Dump it:

```
$ cd ~/imx6ull/virt-lab
$ qemu-system-arm \
    -M virt,dumpdtb=boot/qemu-virt-arm.dtb \
    -cpu cortex-a15 \
```

```
    -m 256M \
    -nographic
```

Depending on QEMU version, it may exit immediately after dumping the DTB or sit with no useful boot. If it sits, quit:

```
Ctrl-a x
```

Convert the DTB to readable DTS:

```
$ dtc -I dtb -O dts -o boot/qemu-virt-arm.dts boot/qemu-virt-arm.dtb
```

Now inspect the hardware Linux will see:

```
$ grep -n "compatible" boot/qemu-virt-arm.dts | head
$ grep -n "pl011" boot/qemu-virt-arm.dts
$ grep -n "memory@" boot/qemu-virt-arm.dts
$ grep -n "interrupt-controller" boot/qemu-virt-arm.dts | head
```

You should find a PL011 UART. That is why the Linux console will be `ttyAMA0`.

129.10 Boot Linux directly

Run:

```
$ cd ~/imx6ull/virt-lab
$ qemu-system-arm \
    -M virt \
    -cpu cortex-a15 \
    -m 256M \
    -nographic \
    -kernel boot/zImage \
    -dtb boot/qemu-virt-arm.dtb \
    -initrd boot/initramfs.cpio.gz \
    -append "console=ttyAMA0 root=/dev/ram0 rdinit=/init"
```

Expected output near the end:

```
Run /init as init process

================================
 QEMU tiny Linux is alive
================================

[init] kernel:
Linux (none) ...

[init] command line:
console=ttyAMA0 root=/dev/ram0 rdinit=/init
```

```
[init] starting shell
/ #
```

Save a clean log:

```
$ qemu-system-arm \
    -M virt \
    -cpu cortex-a15 \
    -m 256M \
    -nographic \
    -kernel boot/zImage \
    -dtb boot/qemu-virt-arm.dtb \
    -initrd boot/initramfs.cpio.gz \
    -append "console=ttyAMA0 root=/dev/ram0 rdinit=/init" \
    2>&1 | tee logs/qemu-linux-direct.log
```

Inside the guest:

```
/ # cat /proc/iomem
/ # ls /sys/firmware/devicetree/base
/ # tr '\0' '\n' < /sys/firmware/devicetree/base/compatible
/ # mount
/ # poweroff -f
```

If `poweroff -f` does not exit QEMU, use:

```
Ctrl-a x
```

129.11 Every QEMU option explained

Option	Meaning
`-M virt`	create QEMU’s generic ARM virtual board
`-cpu cortex-a15`	emulate an ARMv7-A CPU suitable for this example
`-m 256M`	give the guest 256 MiB RAM
`-nographic`	connect guest serial to the terminal
`-kernel boot/zImage`	load the Linux kernel directly

Option	Meaning
`-dtb boot/qemu-virt-arm.dtb`	pass this Device Tree to the kernel
`-initrd boot/initramfs.cpio.gz`	load this cpio archive as initramfs
`-append "..."`	pass the kernel command line

The command line is the virtual hardware bench. Change one piece at a time.

129.12 What `rdinit=/init` did

The kernel unpacks the initramfs into a RAM filesystem and looks for a first program.

With:

```
rdinit=/init
```

the kernel runs `/init` from the initramfs.

Without `rdinit=`, the kernel tries the usual paths:

```
/sbin/init
/etc/init
/bin/init
/bin/sh
```

Our initramfs has `/init`, so we tell the kernel exactly what to run.

129.13 Failure cases in the boot contract

Mismatch	Consequence	Boundary
Console does not name the configured UART	Useful output can disappear despite continued execution	Observation path
No usable root filesystem or initramfs	The kernel cannot start initial user space	Kernel-to-user-space handoff

Mismatch	Consequence	Boundary
`/init` lacks executable permission or a usable interpreter	Finding the file does not make it executable	Filesystem and exec contract
`/init` is built for another architecture	Execution can fail with a format error	Target ABI
DTB does not describe the modeled machine	Device discovery or startup can fail	Hardware description

QEMU can generate a DTB in some direct-boot paths, so omitting an explicit DTB argument is not by itself a universal failure. What matters is whether the kernel receives a description consistent with the machine. These are diagnostic relationships, not required fault-injection tasks.

129.14 Clean rebuild script

Create `run-direct.sh`:

```
#!/bin/sh
set -eu

qemu-system-arm \
    -M virt \
    -cpu cortex-a15 \
    -m 256M \
    -nographic \
    -kernel boot/zImage \
    -dtb boot/qemu-virt-arm.dtb \
    -initrd boot/initramfs.cpio.gz \
    -append "console=ttyAMA0 root=/dev/ram0 rdinit=/init"
```

Make it executable:

```
$ chmod +x run-direct.sh
$ ./run-direct.sh
```

This script is the input to Chapter 130. U-Boot will replace QEMU’s direct loader, but the artifacts stay the same.

129.15 Key Notes

- The kernel, Device Tree and initramfs are separate boot inputs. They provide executable kernel code, a hardware description and the initial user-space tree respectively.
- PID 1 is a runtime role with specific failure consequences, not just an arbitrary shell. The `/init` path must contain an executable compatible with the target and its available interpreter or libraries.
- Console routing determines where evidence is visible. An invisible console and an unbooted kernel are different failure cases.
- A static BusyBox reduces initramfs dependencies at a footprint cost. Dynamic linking instead requires a consistent loader and library set.
- QEMU demonstrates this boot contract, not the physical MINI’s hardware support.

129.16 Pitfalls

- **Building BusyBox for the host.** Check with `file bin/busybox`.
- **Dynamic BusyBox.** Static is simpler for initramfs.
- **Missing `/init` executable bit.** The kernel cannot execute it.
- **Wrong `console=`.** QEMU `virt` uses PL011, which appears as `ttyAMA0`.
- **Confusing `qemu-system-arm` with `qemu-arm`.** `qemu-system-arm` emulates a whole machine. `qemu-arm` runs one ARM user-space program.
- **Assuming QEMU `virt` equals i.MX6ULL.** It does not. It teaches the boot contract.
- **Changing kernel, DTB, and initramfs together.** Change one input at a time.

129.17 Going deeper

- QEMU Arm `virt` machine documentation.
- Linux ARM boot protocol.
- Linux initramfs documentation.
- BusyBox build documentation.
- Chapter 130: U-Boot in QEMU.

Chapter 130: U-Boot in QEMU

> **What:** build U-Boot for QEMU ARM, create a FAT boot disk, manually boot the Chapter 129 Linux system, then turn the commands into a U-Boot script.
>
> **Why:** real boards do not usually jump straight from reset into Linux. Xen also depends on firmware to load images, edit Device Tree, choose addresses, and enter the next stage correctly.
>
> **Focus:** U-Boot owns the handoff: loading bytes, preparing metadata, choosing boot arguments and entering the next stage. Xen adds another consumer of that contract. Understanding the addresses and entry conditions explains the transition without treating a command sequence as proof. **U-Boot:** the bootloader used throughout the book. It initializes enough hardware to load and start the next image. **boot script:** a text file wrapped by `mkimage` so U-Boot can execute it with `source`.

130.1 What changes from Chapter 129

Chapter 129:

```
QEMU -> Linux kernel -> initramfs -> shell
```

This chapter:

```
QEMU -> U-Boot -> Linux kernel -> initramfs -> shell
```

The Linux kernel, DTB, and initramfs are the same artifacts. The loader changes.

That is the lesson. Boot artifacts are not tied to one loader. A loader’s job is to place them in RAM and describe them correctly.

130.2 Prerequisites

You should already have:

```
~/imx6ull/virt-lab/boot/zImage
~/imx6ull/virt-lab/boot/qemu-virt-arm.dtb
~/imx6ull/virt-lab/boot/initramfs.cpio.gz
```

Check:

```
$ cd ~/imx6ull/virt-lab
$ ls -lh boot/zImage boot/qemu-virt-arm.dtb boot/initramfs.cpio.gz
```

Also use the same cross-compiler setting:

```
$ export CROSS_COMPILE=arm-linux-gnueabihf-
```

or your project-local:

```
$ export CROSS_COMPILE=arm-none-linux-gnueabihf-
```

130.3 Install disk-image tools

```
$ sudo apt install mtools dosfstools
```

We use `mtools` because it can copy files into a FAT image without mounting it as root.

130.4 Build U-Boot for QEMU ARM

Get U-Boot:

```
$ cd ~/imx6ull/virt-lab/src
$ git clone https://source.denx.de/u-boot/u-boot.git
$ cd u-boot
```

Configure:

```
$ make qemu_arm_defconfig
```

Build:

```
$ make CROSS_COMPILE=$CROSS_COMPILE -j$(nproc)
```

Inspect:

```
$ file u-boot
$ file u-boot.bin
$ ls -lh u-boot u-boot.bin
```

`u-boot.bin` is the raw firmware image QEMU will load with `-bios`.

Copy it:

```
$ cp u-boot.bin ~/imx6ull/virt-lab/boot/u-boot-qemu-arm.bin
```

130.5 Create a FAT boot disk

Create a 64 MiB image:

```
$ cd ~/imx6ull/virt-lab
$ dd if=/dev/zero of=boot/qemu-fat.img bs=1M count=64
$ mkfs.vfat boot/qemu-fat.img
```

Create `/boot` inside it:

```
$ mmd -i boot/qemu-fat.img ::/boot
```

Copy artifacts:

```
$ mcopy -i boot/qemu-fat.img boot/zImage ::/boot/zImage
$ mcopy -i boot/qemu-fat.img boot/qemu-virt-arm.dtb ::/boot/qemu-virt-arm.dtb
$ mcopy -i boot/qemu-fat.img boot/initramfs.cpio.gz ::/boot/initramfs.cpio.gz
```

List the image:

```
$ mdir -i boot/qemu-fat.img ::/boot
```

If the files are not visible here, U-Boot will not see them either.

130.6 Start U-Boot

Run:

```
$ qemu-system-arm \
    -M virt \
    -cpu cortex-a15 \
    -m 256M \
    -nographic \
    -bios boot/u-boot-qemu-arm.bin \
    -drive if=none,file=boot/qemu-fat.img,format=raw,id=hd0 \
    -device virtio-blk-device,drive=hd0
```

Stop autoboot when prompted. You should reach:

```
=>
```

If you miss the countdown, quit with `Ctrl-a x` and restart.

130.7 Inspect U-Boot’s view of the machine

At the U-Boot prompt:

```
=> version
=> bdinfo
=> printenv
```

Now scan virtio:

```
=> virtio scan
=> virtio info
```

List the FAT filesystem:

```
=> fatls virtio 0:0 /
=> fatls virtio 0:0 /boot
```

Expected:

```
zImage
qemu-virt-arm.dtb
initramfs.cpio.gz
```

If this fails, debug storage before booting Linux.

130.8 Choose load addresses

We need three non-overlapping regions:

```
0x40200000  zImage
0x43000000  DTB
0x44000000  initramfs
```

Why:

- QEMU `virt` RAM starts at `0x40000000`.
- `0x40200000` leaves room below the kernel.
- DTB and initramfs are far enough away not to collide.
- The addresses are easy to recognize in logs.

Set them:

```
=> setenv kernel_addr_r 0x40200000
=> setenv fdt_addr_r 0x43000000
=> setenv ramdisk_addr_r 0x44000000
```

130.9 Load Linux artifacts by hand

Load kernel:

```
=> fatload virtio 0:0 ${kernel_addr_r} /boot/zImage
```

Load DTB:

```
=> fatload virtio 0:0 ${fdt_addr_r} /boot/qemu-virt-arm.dtb
```

Load initramfs and save its size:

```
=> fatload virtio 0:0 ${ramdisk_addr_r} /boot/initramfs.cpio.gz
=> setenv ramdisk_size ${filesize}
=> echo ${ramdisk_size}
```

Inspect memory:

```
=> iminfo ${kernel_addr_r}
```

`iminfo` may not understand `zImage` on all builds. That is fine. The important evidence is the `fatload` byte count.

130.10 Inspect and edit the DTB

Tell U-Boot where the DTB is:

```
=> fdt addr ${fdt_addr_r}
=> fdt header
=> fdt print /chosen
```

If `/chosen` does not exist, create it:

```
=> fdt mknode / chosen
```

Usually QEMU's DTB already has `/chosen`.

Set bootargs:

```
=> setenv bootargs "console=ttyAMA0 root=/dev/ram0 rdinit=/init"
=> fdt set /chosen bootargs "${bootargs}"
=> fdt print /chosen
```

This makes the handoff visible. U-Boot will also pass bootargs to `bootz`, but printing `/chosen` now trains the habit needed for Xen.

130.11 Boot manually

Run:

```
=> bootz ${kernel_addr_r} ${ramdisk_addr_r}:${ramdisk_size} ${fdt_addr_r}
```

Expected:

```
QEMU tiny Linux is alive
/ #
```

Inside Linux:

```
/ # cat /proc/cmdline
/ # tr '\0' '\n' < /sys/firmware/devicetree/base/compatible
/ # poweroff -f
```

If `poweroff -f` does not exit QEMU, use `Ctrl-a x`.

130.12 Save a manual boot log

Run QEMU again and tee the host terminal:

```
$ qemu-system-arm \
    -M virt \
    -cpu cortex-a15 \
    -m 256M \
    -nographic \
    -bios boot/u-boot-qemu-arm.bin \
    -drive if=none,file=boot/qemu-fat.img,format=raw,id=hd0 \
    -device virtio-blk-device,drive=hd0 \
    2>&1 | tee logs/qemu-uboot-manual.log
```

Repeat the manual commands. This log is your proof before scripting.

130.13 Write a U-Boot script

Create `boot-qemu-linux.cmd` on the host:

```
virtio scan

setenv kernel_addr_r 0x40200000
setenv fdt_addr_r 0x43000000
setenv ramdisk_addr_r 0x44000000

fatload virtio 0:0 ${kernel_addr_r} /boot/zImage
fatload virtio 0:0 ${fdt_addr_r} /boot/qemu-virt-arm.dtb
fatload virtio 0:0 ${ramdisk_addr_r} /boot/initramfs.cpio.gz
setenv ramdisk_size ${filesize}
```

```
setenv bootargs "console=ttyAMA0 root=/dev/ram0 rdinit=/init"

fdt addr ${fdt_addr_r}
fdt resize 4096
fdt set /chosen bootargs "${bootargs}"
fdt print /chosen

bootz ${kernel_addr_r} ${ramdisk_addr_r}:${ramdisk_size} ${fdt_addr_r}
```

Wrap it:

```
$ mkimage -A arm -T script -C none -n "QEMU Linux boot" \
    -d boot-qemu-linux.cmd boot-qemu-linux.scr
```

Copy to the FAT image:

```
$ mcopy -o -i boot/qemu-fat.img boot-qemu-linux.scr ::/boot/boot-qemu-linux.scr
$ mdir -i boot/qemu-fat.img ::/boot
```

130.14 Run the script

Start QEMU again. At U-Boot:

```
=> virtio scan
=> fatload virtio 0:0 ${scriptaddr} /boot/boot-qemu-linux.scr
=> source ${scriptaddr}
```

Expected result: same Linux shell as manual boot.

Save:

```
$ qemu-system-arm ... 2>&1 | tee logs/qemu-uboot-script.log
```

130.15 Make autoboot use the script

For QEMU examples, we can type `source` manually. For real boards, we eventually want a repeatable boot command.

At U-Boot:

```
=> setenv bootcmd 'virtio scan; fatload virtio 0:0 ${scriptaddr} /boot/boot-qemu-
linux.scr; source ${scriptaddr}'
=> boot
```

Do not save the environment yet. QEMU’s environment persistence depends on the build and storage setup. For this chapter, proving `bootcmd` works is enough.

130.16 Failure cases in the handoff

An incorrect initramfs extent can prevent archive unpacking or initial user-space execution. Overlapping load ranges can corrupt a previously loaded artifact before the kernel receives control. Wrong console arguments can instead hide a kernel that is still running.

Storage discovery is another boundary: a script cannot load from a virtio device that its selected U-Boot configuration has not discovered. These failures explain why load policy includes sizes, addresses, bus state and arguments, not merely filenames.

130.17 Why this matters for Xen

Xen on ARM uses the same skills with more files:

```
load Xen
load Dom0 kernel
load Dom0 initramfs
load DTB
edit /chosen for Xen bootargs
add /chosen module nodes
jump to Xen
```

If this chapter feels mechanical, good. Chapter 134 is the same handoff pattern with a hypervisor instead of a plain kernel.

130.18 Key Notes

- U-Boot owns artifact loading and the next-stage handoff. The kernel owns interpreting its DTB, arguments and initial filesystem after that handoff.
- Load addresses are a memory-layout design. Kernel placement, decompression, DTB and initramfs must remain compatible with available RAM and each other.
- File sizes and module extents are metadata with consequences. A correctly named file loaded to the wrong range is still an invalid handoff.
- Boot scripts encode policy and failure behavior. Automation should preserve the same checks and stop conditions as the mechanism it describes.
- The QEMU virtio storage path is an emulated interface, not evidence that a MINI USDHC or physical card works.

130.19 Pitfalls

- **Wrong disk interface.** With virtio disks, run `virtio scan`.

- **Overlapping load addresses.** Kernel, DTB, and initramfs must not collide.
- **Forgetting ramdisk size.** `bootz` needs `addr:size`.
- **Trusting default bootargs.** Print `/chosen`.
- **Scripting before manual boot works.** Manual first, script second.
- **Saving a bad environment.** Do not persist experiments until the command is proven.
- **Forgetting logs.** The serial log is the only witness once a boot goes wrong.

130.20 Going deeper

- U-Boot QEMU ARM documentation.
- U-Boot `bootz`, `source`, `mkimage`, `fdt`, and `virtio` command documentation.
- QEMU block-device documentation.
- Chapter 134 for the same pattern booting Xen on i.MX6ULL.

Chapter 131: HYP mode, stage-2 MMU, and virtual interrupts

> **What:** understand the CPU and memory mechanisms that let Xen and Jailhouse run operating systems under another layer of control.
>
> **Why:** without HYP mode, stage-2 translation, and virtual interrupts, the later chapters become cargo-cult boot commands. This chapter gives every later log line a hook.
>
> **Focus:** a guest kernel is not a process. It is a kernel running at PL1, with a hypervisor at PL2 deciding what that kernel is allowed to see. **HYP mode:** the ARMv7-A hypervisor processor mode, privilege level PL2. **Stage-2 MMU:** the hypervisor-controlled translation from guest physical addresses to real physical addresses. **Virtual interrupt:** an interrupt presented to a guest by the hypervisor, whether or not it is currently a physical interrupt line.

131.1 Start from the non-virtualized machine

Normal ARMv7-A Linux on our board uses:

```
User process      PL0   USR mode
Linux kernel      PL1   SVC/IRQ/ABT/UND modes
```

When a user program calls `write()`:

```
user code
  -> svc instruction
  -> CPU enters SVC mode
  -> Linux syscall handler runs
  -> return to user
```

When UART interrupt fires:

```
UART asserts IRQ
  -> GIC signals CPU
  -> CPU enters IRQ mode
  -> Linux IRQ handler runs
  -> return to interrupted context
```

Linux owns the privileged machine.

131.2 Add the hypervisor layer

With virtualization:

```
Guest user process       PL0
Guest kernel             PL1
Hypervisor               PL2 / HYP
```

The guest kernel still runs privileged code. It still has page tables. It still handles syscalls. It still thinks it owns a machine.

But the hypervisor controls the machine behind that machine.

The hypervisor can:

- trap selected guest operations,
- control guest physical memory,
- inject virtual interrupts,
- expose or hide devices,
- pause or destroy a guest,
- switch between guests.

This is the key sentence:

```
The guest owns a virtual machine. The hypervisor owns the real machine.
```

131.3 `svc`, `hvc`, and `smc`

Three instructions look similar because all are deliberate exceptions.

Instruction	Goes to	Typical meaning
`svc`	PL1 kernel	user asks kernel
`hvc`	PL2 hypervisor	guest asks hypervisor
`smc`	Monitor/Secure World	Normal World asks Secure World

Examples:

```
printf()
  -> write()
  -> svc
  -> Linux syscall

Xen hypercall
  -> hvc
  -> Xen

OP-TEE call
  -> smc
  -> secure monitor / secure world
```

Do not blur these:

- HYP mode is for virtualization.
- Monitor mode is for TrustZone world switching.

They both feel “below Linux”, but they solve different problems.

131.4 What can trap to HYP

The hypervisor can configure traps for sensitive operations. Exact controls depend on the ARM virtualization extension registers, but conceptually traps include:

- selected CP15/system register accesses,
- guest WFI/WFE behavior,
- guest interrupt-controller accesses,
- guest timer behavior,
- stage-2 translation faults,
- hypercalls through `hvc`.

Trap flow:

```
guest kernel instruction
  -> CPU decides this action must trap
  -> CPU enters HYP mode
  -> hypervisor handler runs
  -> hypervisor emulates/allows/denies
  -> guest resumes or is killed
```

The guest may never know the trap happened.

131.5 Stage 1 without virtualization

Chapter 17 built a simple stage-1 mapping:

```
virtual address -> physical address
```

Linux uses a more complex version:

```
user VA 0x40001000 -> PA 0x81234000
kernel VA 0xc0200000 -> PA 0x80200000
```

The stage-1 page tables answer:

```
What physical address does this virtual address mean?
Can this access read?
Can this access write?
Can this access execute?
Is it cacheable?
Is it device memory?
```

In normal Linux, the answer’s physical address is real physical memory.

131.6 Stage 1 inside a guest

Inside a guest, stage 1 still exists:

```
guest virtual -> guest physical
```

The guest kernel builds those page tables. From the guest’s point of view, nothing strange happened.

Example:

```
DomU process VA     0x40001000
DomU stage 1 maps   0x40001000 -> 0x80200000
```

The guest calls `0x80200000` physical.

But it is only **guest physical**.

131.7 Stage 2

The hypervisor adds:

```
guest physical -> real physical
```

Full path:

```
guest virtual -> guest physical -> real physical
```

Example:

```
DomU process VA       0x40001000
guest stage 1         0x40001000 -> 0x80200000
hypervisor stage 2    0x80200000 -> 0x8a200000
real DRAM access      0x8a200000
```

The guest sees:

```
0x80200000
```

The bus sees:

```
0x8a200000
```

That is the memory isolation boundary.

131.8 Stage-2 fault

If the guest touches a guest physical address that stage 2 does not map:

```
guest load/store/fetch
  -> stage 1 succeeds
  -> stage 2 fails
  -> CPU traps to HYP
  -> hypervisor handles stage-2 fault
```

The hypervisor can:

- map a page lazily,
- emulate a device,
- inject an abort into the guest,
- kill the guest.

This is why a bad guest memory access does not have to corrupt Dom0.

131.9 Stage 2 and MMIO

MMIO is just address space from the CPU’s point of view.

If a guest reads:

```
0x0209c000
```

the hypervisor chooses:

- map it to the real GPIO register,
- leave it unmapped and fault,
- map it to an emulated device,
- map it only for Dom0.

On a small embedded SoC, passing real MMIO to a guest is serious. The register may control clocks, resets, interrupts, pins, or DMA.

131.10 Stage 2 does not solve DMA

This warning deserves its own section.

Stage 2 protects CPU translations.

DMA is different:

```
device bus master -> RAM
```

A DMA-capable device may write memory without the guest CPU executing a load or store. If the SoC has no IOMMU or firewall for that device, a passed-through device may be able to write outside the guest’s assigned memory.

So:

```
CPU isolation: stage 2
DMA isolation: IOMMU/firewall/platform design
```

Do not claim safe passthrough until both are handled.

131.11 Interrupts before virtualization

Normal path:

```
device raises interrupt
  -> GIC distributor
  -> GIC CPU interface
  -> CPU enters IRQ mode
  -> Linux handler
```

Linux expects to:

- configure interrupt priority,
- enable/disable IRQs,
- acknowledge interrupts,
- send end-of-interrupt,
- mask/unmask devices,
- map IRQ numbers to drivers.

131.12 Interrupts with a hypervisor

With a hypervisor:

```
physical interrupt
  -> hypervisor or assigned guest policy
  -> maybe virtual interrupt injection
  -> guest IRQ handler
```

The hypervisor decides:

- which physical interrupt belongs to Dom0,
- which belongs to a DomU,
- which is virtual,
- which is hidden,
- which should wake a paused guest,
- which should never be exposed.

The guest still sees something that looks like an interrupt controller. It must, or Linux will not boot.

131.13 Virtual GIC

ARM systems use the GIC. A guest kernel expects a GIC-like interface.

The hypervisor can provide a **virtual GIC**:

```
guest writes virtual GIC register
  -> trap or virtual interface
  -> hypervisor updates virtual interrupt state
```

This is why Xen logs mention GIC details and why the DTB must describe interrupt controllers correctly.

If GIC setup is wrong, failure is often early and quiet.

131.14 Timers before virtualization

Every OS needs a timer for:

- scheduler ticks,
- timeouts,
- sleeps,
- TCP retransmits,
- RCU,
- soft lockup detection,
- watchdog userspace.

On ARMv7-A, the architected timer is a central piece of Linux boot.

131.15 Virtual timers

A guest needs a timer it can believe.

The hypervisor must ensure:

- the guest can read time,
- timer interrupts arrive,
- one guest cannot break another guest’s time,
- time continues sensibly when guests are scheduled.

Timer bugs look weird:

- boot stalls,
- sleeps never wake,
- RCU stalls,
- scheduler warnings,
- networking timeouts.

Always read timer lines in Xen and Linux logs.

131.16 Device models

Three common models:

Model	What guest sees	Example
Emulated	fake device implemented by hypervisor	legacy PC devices in desktop VMs
Paravirtual	hypervisor-aware virtual device	Xen console, Xen block, Xen net
Passthrough	real hardware device	assigning a UART or controller

Embedded Xen often starts with:

- Dom0 owns physical devices,
- guests use paravirtual console/block/net,
- passthrough is delayed until the ownership table is clean.

That is the safe order.

131.17 Dom0 and DomU

Xen names its domains:

```
Dom0: privileged control domain
DomU: unprivileged guest domain
```

Dom0 is Linux, but not normal bare-metal Linux.

Dom0 can:

- run the Xen toolstack,
- create DomU guests,
- provide virtual block/network backends,
- own real devices,
- mediate guest lifecycle.

DomU can:

- run Linux,
- use assigned memory,
- use virtual devices,
- crash without necessarily killing Dom0.

131.18 Jailhouse root cell and inmate

Jailhouse uses different language:

```
root cell: Linux after Jailhouse is enabled
inmate cell: isolated partition
```

Xen boots first. Jailhouse is enabled after Linux boots.

Compare:

```
Xen:
  firmware -> Xen -> Dom0 -> DomU

Jailhouse:
  firmware -> Linux -> enable Jailhouse -> inmate
```

Both use virtualization hardware. Their product models are different.

131.19 What to look for in logs

When reading logs, mark these:

```
Who printed first:
CPU model:
CPU count:
virtualization extension visible:
GIC version:
timer:
memory map:
Xen command line:
Dom0 command line:
Dom0 start:
DomU start:
console:
```

The most useful skill is seeing the layer boundary in a log.

131.20 Example annotation: plain Linux

Plain QEMU Linux from Chapter 129:

```
Booting Linux on physical CPU 0x0
Linux version ...
Machine model: linux,dummy-virt
Kernel command line: console=ttyAMA0 root=/dev/ram0 rdinit=/init
GIC: ...
clocksource: arch_sys_counter
Run /init as init process
```

Interpretation:

- Linux printed first.
- No Xen before it.
- Linux owns the machine.
- Console is physical PL011 as `ttyAMA0`.
- `/init` came from initramfs.

131.21 Example annotation: Xen + Dom0

Xen boot:

```
Xen ...
Command line: console=dtuart ...
Booting on ARM ...
GICv2 initialization
Bringing up CPU0
Loading Dom0 kernel
```

Dom0 Linux later:

```
Linux version ...
Kernel command line: console=hvc0 ...
Xen: initializing event channel driver
Run /init as init process
```

Interpretation:

- Xen printed first.
- Linux is Dom0.
- Dom0 console is `hvc0`, not the physical UART.
- Xen controls the machine boundary.

131.22 Translation ownership

Access	Stage 1 result	Stage 2 result	Stage 1 owner	Stage 2 owner
Native Linux user load	Physical address	No second stage	Linux	None
Dom0 user load	Guest physical address	Machine physical address	Dom0 Linux	Xen
DomU user load	Guest physical address	Machine physical address	DomU Linux	Xen
Bare-metal inmate load	Depends on inmate setup	Assigned physical mapping	Inmate or no stage 1	Jailhouse

The guest’s idea of a physical address is not necessarily a machine address. A second-stage fault reports a CPU access that violates the hypervisor’s mapping. It does not prove that a bus-mastering device has the same restriction.

131.23 Key Notes

- Stage 1 belongs to the guest operating system when it uses translation. Stage 2 gives the hypervisor a separate CPU memory boundary.
- HYP mode controls virtualization in Normal World. It is not the Secure World boundary managed through Monitor mode.
- Virtual interrupts and timers are part of a usable guest platform. Memory mapping alone is not enough to run another kernel correctly.
- Dom0 is privileged within Xen’s architecture, but remains a domain. Jailhouse instead changes ownership of an already-running Linux system’s resources.
- Startup logs can distinguish firmware, hypervisor and guest stages. They establish observed progress, not DMA isolation, timing guarantees or electrical qualification.

131.24 Pitfalls

- **Thinking guest physical is real physical.** It is real only inside the guest.

- **Thinking HYP is Secure World.** HYP virtualizes Normal World. Monitor mode switches worlds.
- **Ignoring stage-2 faults.** They are the hypervisor memory boundary doing its job.
- **Ignoring interrupts.** A guest with broken virtual interrupts is not a usable OS.
- **Ignoring timer messages.** Time bugs look like random boot hangs.
- **Assuming stage 2 protects DMA.** It protects CPU translations, not every bus master.
- **Treating Dom0 as ordinary Linux.** Dom0 is privileged, but still a Xen domain.
- **Confusing Xen and Jailhouse.** Xen creates and manages domains. Jailhouse partitions hardware after Linux boots.

131.25 Going deeper

- ARMv7-A Virtualization Extensions documentation.
- Cortex-A7 MPCore Technical Reference Manual.
- Xen ARM boot and Device Tree documentation.
- Xen event channels and grant tables.
- Jailhouse documentation.
- Chapter 17 for stage-1 MMU foundations.
- Chapter 136 for DMA and device ownership.

Chapter 132: Xen in QEMU

> **What:** build Xen for 32-bit Arm, build a Dom0-capable Linux kernel, create a tiny Dom0 initramfs, build a Xen-aware DTB, and boot `QEMU -> Xen -> Dom0`.
>
> **Why:** Xen on the real i.MX6ULL adds board-specific failure modes. QEMU lets us prove the Xen boot contract first.
>
> **Focus:** Xen boots first. Linux is no longer the first privileged software. Dom0 is Linux, but Linux running as a Xen domain. **Xen:** a type-1 hypervisor. On ARM it runs before Linux and creates domains. **Dom0:** the first privileged Xen domain, usually Linux, responsible for control and device backends. **DomU:** an unprivileged guest domain created later by Dom0.

132.1 Boot path

The target boot path:

```
qemu-system-arm
  -> loads Xen as the kernel image
  -> loads Dom0 kernel into RAM
  -> loads Dom0 initramfs into RAM
  -> passes a DTB describing those modules
Xen
  -> initializes CPU, GIC, timer, memory
  -> creates Domain 0
Dom0 Linux
  -> boots with console=hvc0
  -> unpacks initramfs
  -> runs /init
```

This is the first time Linux is not the owner of the machine.

132.2 Workspace

```
$ mkdir -p ~/imx6ull/xen-qemu/{src,build,rootfs,boot,logs,notes}
$ cd ~/imx6ull/xen-qemu
```

Final artifacts:

```
boot/
  xen
  zImage-dom0
  dom0-initramfs.cpio.gz
  qemu-virt-arm-xen.dtb
  qemu-virt-arm-xen.dts
```

```
logs/
  xen-qemu-dom0.log
notes/
  xen-qemu-addresses.md
```

132.3 Install host packages

```
$ sudo apt update
$ sudo apt install build-essential git bc bison flex libssl-dev \
    gcc-arm-linux-gnueabihf qemu-system-arm device-tree-compiler \
    cpio gzip file
```

Set cross compiler:

```
$ export CROSS_COMPILE=arm-linux-gnueabihf-
$ export ARCH=arm
```

If you use the book's project-local toolchain, set `CROSS_COMPILE` to that prefix instead.

132.4 Choose addresses first

QEMU `virt` 32-bit RAM starts at `0x40000000`.

Use:

```
0x44000000  Dom0 zImage
0x48000000  Dom0 initramfs
```

In this chapter, QEMU loads Xen with `-kernel boot/xen` and passes the DTB with `-dtb`. We do not manually choose Xen's own load address here. We do manually choose the Dom0 module addresses because Xen finds those modules through `/chosen/module@.../reg`.

Create notes:

```
$ cat > notes/xen-qemu-addresses.md <<'EOF'
# Xen QEMU addresses

QEMU RAM base: 0x40000000

| Artifact | Address | Why |
|----------|---------|-----|
| Dom0 zImage | 0x44000000 | module 1 |
| Dom0 initramfs | 0x48000000 | module 2 |

Xen itself is loaded by QEMU through -kernel.
The DTB is passed by QEMU through -dtb.
EOF
```

The DTB module nodes must match the Dom0 addresses exactly.

132.5 Build Xen

Get Xen:

```
$ cd ~/imx6ull/xen-qemu/src
$ git clone https://github.com/xen-project/xen.git
$ cd xen
$ git checkout RELEASE-4.21.1
```

If that exact tag is unavailable in the future, use a current supported release and record it.

Build only the hypervisor:

```
$ make XEN_TARGET_ARCH=arm32 CROSS_COMPILE=$CROSS_COMPILE dist-xen
```

Copy:

```
$ cp xen/xen ~/imx6ull/xen-qemu/boot/xen
$ file ~/imx6ull/xen-qemu/boot/xen
$ ls -lh ~/imx6ull/xen-qemu/boot/xen
```

The file must be an Arm Xen image, not x86.

132.6 Build Dom0 Linux

Get Linux:

```
$ cd ~/imx6ull/xen-qemu/src
$ git clone https://git.kernel.org/pub/scm/linux/kernel/git/stable/linux.git
$ cd linux
$ git checkout v6.6
```

Configure:

```
$ make ARCH=arm O=~/imx6ull/xen-qemu/build/linux-dom0 multi_v7_defconfig
```

Enable Dom0-relevant options:

```
$ scripts/config --file ~/imx6ull/xen-qemu/build/linux-dom0/.config \
    -e XEN \
    -e XEN_DOM0 \
    -e HVC_XEN \
    -e XEN_DEV_EVTCHN \
    -e XENFS \
    -e XEN_SYS_HYPERVISOR \
    -e BLK_DEV_INITRD \
    -e DEVTMPFS \
    -e DEVTMPFS_MOUNT
```

```
$ make ARCH=arm O=~/imx6ull/xen-qemu/build/linux-dom0 olddefconfig
```

Build:

```
$ make ARCH=arm O=~/imx6ull/xen-qemu/build/linux-dom0 \
    CROSS_COMPILE=$CROSS_COMPILE -j$(nproc) zImage
```

Copy:

```
$ cp ~/imx6ull/xen-qemu/build/linux-dom0/arch/arm/boot/zImage \
    ~/imx6ull/xen-qemu/boot/zImage-dom0
$ ls -lh ~/imx6ull/xen-qemu/boot/zImage-dom0
```

132.7 Build Dom0 initramfs

Use a tiny BusyBox rootfs. You can reuse the static BusyBox from Chapter 129.

```
$ cd ~/imx6ull/xen-qemu
$ rm -rf rootfs/*
$ mkdir -p rootfs/{bin,sbin,proc,sys,dev,tmp,run,etc,root}
$ cp ~/imx6ull/virt-lab/rootfs/bin/busybox rootfs/bin/busybox
$ chmod +x rootfs/bin/busybox
$ cd rootfs/bin
$ for app in $(./busybox --list); do ln -sf busybox $app; done
$ cd ..
```

Create `/init`:

```
$ cat > init <<'EOF'
#!/bin/sh
mount -t proc none /proc
mount -t sysfs none /sys
mount -t devtmpfs none /dev

echo
echo "====================="
echo " QEMU Dom0 is alive"
echo "====================="
echo

echo "[dom0] kernel:"
uname -a

echo
echo "[dom0] cmdline:"
cat /proc/cmdline

echo
echo "[dom0] xen evidence:"
ls /sys/hypervisor 2>/dev/null || true
dmesg | grep -i xen | head -20 || true
```

```
exec /bin/sh
EOF
$ chmod +x init
```

Pack:

```
$ find . -print0 | cpio --null -ov --format=newc | gzip -9 > ../boot/dom0-
initramfs.cpio.gz
```

Check:

```
$ cd ~/imx6ull/xen-qemu
$ ls -lh boot/dom0-initramfs.cpio.gz
$ gzip -dc boot/dom0-initramfs.cpio.gz | cpio -it | grep '^init$'
```

132.8 Get a QEMU DTB

Dump:

```
$ cd ~/imx6ull/xen-qemu
$ qemu-system-arm \
    -M virt,virtualization=on,dumpdtb=boot/qemu-virt-arm-base.dtb \
    -cpu cortex-a15 \
    -m 512M \
    -nographic
```

Quit with `Ctrl-a x` if it stays open.

Convert:

```
$ dtc -I dtb -O dts -o boot/qemu-virt-arm-base.dts boot/qemu-virt-arm-base.dtb
```

The `virtualization=on` part matters. Xen needs virtualization support exposed by QEMU.

132.9 Add Xen `/chosen` nodes

Copy the base DTS:

```
$ cp boot/qemu-virt-arm-base.dts boot/qemu-virt-arm-xen.dts
```

Open `boot/qemu-virt-arm-xen.dts` and find `/chosen`. Add or replace the contents so it has this shape:

```
chosen {
    xen,xen-bootargs = "console=dtuart dtuart=serial0 dom0_mem=256M dom0_max_vcpus=1";
    xen,dom0-bootargs = "console=hvc0 earlycon=xenboot root=/dev/ram0 rdinit=/init";
```

```
    module@44000000 {
        compatible = "multiboot,kernel", "multiboot,module";
        reg = <0x44000000 0x0>;
    };

    module@48000000 {
        compatible = "multiboot,ramdisk", "multiboot,module";
        reg = <0x48000000 0x0>;
    };
};
```

Now replace the two `0x0` sizes with real file sizes.

Before doing that, check how many cells this DTB expects for addresses and sizes:

```
$ grep -n "#address-cells\\|#size-cells" boot/qemu-virt-arm-xen.dts | head
```

For the 32-bit QEMU `virt` DTB used in this chapter, the simple two-cell form is the common shape:

```
reg = <address size>;
```

If your dumped DTB uses two address cells or two size cells at the relevant parent node, write the `reg` property with the matching number of cells instead:

```
reg = <0x0 0x44000000 0x0 0x006a1234>;
```

Do not guess. `dtc` can compile a syntactically valid DTB that still describes the module address in the wrong cell format.

Get sizes in hex:

```
$ printf '0x%x\n' $(stat -c%s boot/zImage-dom0)
$ printf '0x%x\n' $(stat -c%s boot/dom0-initramfs.cpio.gz)
```

If the DTB uses the simple two-cell form and Dom0 kernel size is `0x6a1234`, the first `reg` becomes:

```
reg = <0x44000000 0x6a1234>;
```

If initramfs size is `0x91234`, the second becomes:

```
reg = <0x48000000 0x91234>;
```

Compile:

```
$ dtc -I dts -O dtb -o boot/qemu-virt-arm-xen.dtb boot/qemu-virt-arm-xen.dts
```

Inspect:

```
$ dtc -I dtb -O dts boot/qemu-virt-arm-xen.dtb | less
```

Search for:

- `xen,xen-bootargs,`
- `xen,dom0-bootargs,`
- `module@44000000,`
- `module@48000000.`

Also re-check the final `reg` properties. They must match:

```
Dom0 kernel load address and size
Dom0 initramfs load address and size
the address-cell and size-cell format used by this DTB
```

132.10 Boot Xen

Run:

```
$ cd ~/imx6ull/xen-qemu
$ qemu-system-arm \
    -M virt,virtualization=on \
    -cpu cortex-a15 \
    -m 512M \
    -nographic \
    -kernel boot/xen \
    -dtb boot/qemu-virt-arm-xen.dtb \
    -device loader,file=boot/zImage-dom0,addr=0x44000000 \
    -device loader,file=boot/dom0-initramfs.cpio.gz,addr=0x48000000
```

Expected sequence:

```
Xen ...
Booting on ARM ...
Loading Dom0 kernel ...
Linux version ...
QEMU Dom0 is alive
/ #
```

Save a log:

```
$ qemu-system-arm \
    -M virt,virtualization=on \
    -cpu cortex-a15 \
    -m 512M \
    -nographic \
    -kernel boot/xen \
    -dtb boot/qemu-virt-arm-xen.dtb \
    -device loader,file=boot/zImage-dom0,addr=0x44000000 \
    -device loader,file=boot/dom0-initramfs.cpio.gz,addr=0x48000000 \
    2>&1 | tee logs/xen-qemu-dom0.log
```

132.11 Verify from Dom0

Inside Dom0:

```
/ # cat /proc/cmdline
/ # ls /sys/hypervisor
/ # dmesg | grep -i xen
/ # mount
/ # poweroff -f
```

The command line should contain:

```
console=hvc0 earlycon=xenboot root=/dev/ram0 rdinit=/init
```

The console is `hvc0` because Dom0 is using Xen's paravirtual console, not the physical PL011 UART directly.

132.12 Annotate the log

Create:

```
$ cp logs/xen-qemu-dom0.log logs/xen-qemu-dom0.annotated.log
```

Add comments manually around:

```
Xen first line:
Xen command line:
CPU/GIC/timer:
Dom0 creation:
Linux first line:
Dom0 command line:
/init output:
```

This annotation becomes input for Chapter 131's log-reading example.

132.13 Failure cases in Xen startup

Xen depends on exposed virtualization hardware and a consistent description of its modules. A machine without the necessary extensions can fail before Dom0 starts. A wrong module address points Xen at the wrong bytes, while a wrong module extent can leave Dom0 without a usable initramfs.

Console mismatch is different: Xen can print while Dom0 output is missing. The two stages have separate console ownership, so the last visible message does not by itself identify a failed guest.

132.14 Key Notes

- Xen takes control before Dom0 Linux. Dom0 is a privileged domain with platform responsibilities, not native Linux outside the hypervisor.
- Module addresses and lengths in the Xen-aware DTB must agree with the actual loaded artifacts. Separate Xen and Dom0 command lines control separate stages.
- Virtualization exposure, interrupt delivery and timers are part of the machine contract. A guest cannot create hardware capabilities that the virtual machine did not expose.
- Xen’s console and Dom0’s console have different owners. Their logs provide different evidence of how far startup progressed.
- A Dom0 shell establishes a minimal boot path, not DomU lifecycle support, safe device passthrough or real-board qualification.

132.15 Pitfalls

- **No `virtualization=on`.** Xen needs virtualization support exposed.
- **DTB module address mismatch.** QEMU loader address and DTB `reg` must agree.
- **Wrong module size.** Xen does not know your file size unless you tell it.
- **Dom0 console wrong.** Start with `console=hvc0 earlycon=xenboot`.
- **Too large Dom0.** Keep the initramfs small.
- **Skipping annotation.** If you cannot mark the handoff in the log, you do not understand the boot yet.

132.16 Going deeper

- Xen ARM booting documentation.

- Xen ARM Device Tree boot module documentation.
- QEMU loader device documentation.
- Linux Xen guest configuration options.
- Chapter 133 for starting the first DomU.

Chapter 133: First DomU Linux

> **What:** add the Xen toolstack to Dom0, build a tiny DomU kernel and initramfs, write an `xl` domain config, and start a second Linux kernel under Xen.
>
> **Why:** Xen is not interesting until a guest exists. This chapter produces the first real "two Linux systems are alive" result.
>
> **Focus:** Dom0 controls the platform. DomU receives only the memory, vCPU, console, and devices Xen gives it. **xl:** Xen's command-line tool for creating, listing, controlling, and destroying domains. **xenstore:** Xen's shared configuration/state database used by domains and tools. **event channel:** Xen's lightweight notification mechanism between domains and Xen.

133.1 Boot path

We extend Chapter 132:

```
QEMU -> Xen -> Dom0 Linux -> xl create -> DomU Linux
```

The new moving parts are:

- a Dom0 rootfs with Xen tools,
- a DomU kernel,
- a DomU initramfs,
- an `xl` config file.

133.2 Why the Chapter 132 Dom0 is too small

Chapter 132 used a tiny BusyBox initramfs. It was perfect for proving:

```
Xen -> Dom0 -> shell
```

But `xl` needs more user space:

- libraries,
- xenstore daemon,
- xenfs,
- event channel device,

- control scripts or minimal replacements.

For this chapter, use a small Buildroot Dom0. That is still buildable, but less fragile than hand-copying every toolstack dependency.

133.3 Buildroot Dom0 shape

Create a Buildroot config for ARMv7 with:

```
Target architecture: ARM little endian
Target ABI: EABIhf
Target CPU: cortex-A15 for QEMU
Init system: BusyBox or systemd, BusyBox is smaller
Root filesystem: cpio gzip
```

Packages:

```
Xen tools
dtc, optional
busybox shell utilities
```

Buildroot symbol names vary. Search:

```
$ make menuconfig
```

Then use `/` search for:

```
xen
```

Enable the Xen toolstack package if available in your Buildroot release.

The output should include a rootfs cpio archive and `/usr/sbin/xl` or `/usr/bin/xl` inside it.

133.4 Dom0 kernel config

Dom0 Linux needs:

```
CONFIG_XEN=y
CONFIG_XEN_DOM0=y
CONFIG_HVC_XEN=y
CONFIG_XEN_DEV_EVTCHN=y
CONFIG_XENFS=y
CONFIG_XEN_SYS_HYPERVISOR=y
CONFIG_XEN_BACKEND=y
CONFIG_XEN_BLKDEV_BACKEND=y
CONFIG_XEN_NETDEV_BACKEND=y
```

For the first DomU, block/net backends are optional if the guest uses only initramfs and console. Enable them anyway so Chapter 135 has a smoother path.

133.5 Dom0 startup script

Dom0 must mount xenfs and start xenstore if your rootfs does not do it automatically.

Minimal init sequence:

```
mount -t proc none /proc
mount -t sysfs none /sys
mount -t devtmpfs none /dev
mkdir -p /proc/xen
mount -t xenfs none /proc/xen

# Depending on packaging:
xenstored --pid-file /run/xenstored.pid &
sleep 1
```

Some distributions start xenstored through init scripts. Buildroot packaging may provide a service. The test is not “did a service file exist”, it is:

```
# xl info
```

133.6 Dom0 checks before DomU

Boot Xen + Dom0 as in Chapter 132, but with the richer Dom0 initramfs.

Inside Dom0:

```
# which xl
# mount | grep xen
# ls /proc/xen
# ls /dev/xen 2>/dev/null || true
# xl info
# xl list
```

Expected `xl list`:

```
Name                                        ID   Mem VCPUs      State   Time(s)
Domain-0                                     0   ...     1     r-----     ...
```

If `xl info` fails, stop. DomU cannot work until Dom0 tooling works.

133.7 Build DomU Linux

Use a separate kernel build directory:

```
$ cd ~/imx6ull/xen-qemu/src/linux
$ make ARCH=arm O=~/imx6ull/xen-qemu/build/linux-domu multi_v7_defconfig
```

Enable:

```
$ scripts/config --file ~/imx6ull/xen-qemu/build/linux-domu/.config \
    -e XEN \
    -e HVC_XEN \
    -e XEN_DEV_EVTCHN \
    -e XENFS \
    -e BLK_DEV_INITRD \
    -e DEVTMPFS \
    -e DEVTMPFS_MOUNT

$ make ARCH=arm O=~/imx6ull/xen-qemu/build/linux-domu olddefconfig
```

Build:

```
$ make ARCH=arm O=~/imx6ull/xen-qemu/build/linux-domu \
    CROSS_COMPILE=$CROSS_COMPILE -j$(nproc) zImage
```

Copy into Dom0 rootfs or a location Dom0 can read:

```
/boot/domu-zImage
```

133.8 Build DomU initramfs

DomU `/init`:

```
#!/bin/sh
mount -t proc none /proc
mount -t sysfs none /sys
mount -t devtmpfs none /dev

echo
echo "================"
echo " DomU is alive"
echo "================"
echo

uname -a
cat /proc/cmdline
head -5 /proc/meminfo

exec /bin/sh
```

Pack:

```
$ find . -print0 | cpio --null -ov --format=newc | gzip -9 > domu-initramfs.cpio.gz
```

Copy into Dom0:

```
/boot/domu-initramfs.cpio.gz
```

133.9 Domain config

Create `/etc/xen/domu-tiny.cfg` inside Dom0:

```
name = "domu-tiny"
kernel = "/boot/domu-zImage"
ramdisk = "/boot/domu-initramfs.cpio.gz"
memory = 64
vcpus = 1
extra = "console=hvc0 rdinit=/init"
on_crash = "preserve"
```

Line by line:

Line	Meaning
`name`	domain name shown by `xl list`
`kernel`	path inside Dom0 rootfs
`ramdisk`	path inside Dom0 rootfs
`memory`	MiB assigned to DomU
`vcpus`	virtual CPUs assigned to DomU
`extra`	DomU kernel command line
`on_crash`	keep crashed domain for inspection

Start with 64 MiB. It is enough for an initramfs shell and small enough to keep memory pressure visible.

133.10 Start DomU

Inside Dom0:

```
# xl create -c /etc/xen/domu-tiny.cfg
```

Expected:

```
DomU is alive
/ #
```

Detach:

```
Ctrl-]
```

List:

```
# xl list
```

Attach again:

```
# xl console domu-tiny
```

133.11 Prove there are two Linux kernels

In Dom0:

```
# uname -a
# cat /proc/cmdline
# xl list
```

In DomU:

```
# uname -a
# cat /proc/cmdline
# head -5 /proc/meminfo
```

The two command lines should differ:

- Dom0 has Xen Dom0 bootargs.
- DomU has `console=hvc0 rdinit=/init`.

The memory sizes should differ too.

133.12 Shut down and restart

Try graceful shutdown:

```
# xl shutdown domu-tiny
# xl list
```

If the tiny DomU lacks enough user-space support for graceful shutdown:

```
# xl destroy domu-tiny
```

Start it again:

```
# xl create -c /etc/xen/domu-tiny.cfg
```

Lifecycle matters. A guest that can boot once but not restart is not understood yet.

133.13 Prove failure isolation

Inside DomU:

```
# kill -SEGV 1
```

If the guest remains alive, try:

```
# echo c > /proc/sysrq-trigger
```

This requires sysrq support and may not be enabled. The goal is simple: break DomU.

Back in Dom0:

```
# xl list
# dmesg | tail -40
```

Dom0 should survive. Xen should survive. QEMU should keep running.

This is the first concrete hypervisor payoff.

133.14 Save logs

Capture:

```
# xl info > /root/xl-info.txt
# xl list > /root/xl-list-with-domu.txt
# dmesg > /root/dom0-dmesg-after-domu.txt
```

On the host, save the whole QEMU serial run:

```
$ qemu-system-arm ... 2>&1 | tee logs/xen-qemu-domu.log
```

133.15 Failure cases in guest creation

Missing kernel files or an impossible memory allocation can stop domain creation. A malformed initramfs can fail later, after the guest kernel has begun executing. A wrong console can conceal that progress without being the cause of a stopped kernel.

These cases belong to different owners: toolstack artifact access, hypervisor resource allocation and guest runtime setup. Treating them all as a failed `xl` command discards useful diagnostic information.

133.16 Key Notes

- A DomU is another kernel with assigned memory, virtual CPUs and interfaces. It is not an ordinary process in Dom0's PID namespace.
- Dom0's toolstack and service state own domain management. A minimal proof-of-boot initramfs does not necessarily contain those management facilities.
- Kernel and rootfs paths in a domain configuration are resolved in Dom0, not on the development host.
- Guest restart, normal shutdown and forced destruction have different persistence consequences. Lifecycle policy should distinguish them.
- Containing one guest failure is evidence for that failure path. It does not establish containment of Dom0, shared hardware or every DMA-capable device.

133.17 Pitfalls

- **Using the tiny Chapter 132 Dom0.** It probably lacks `xl`.
- **No xenfs.** Mount `/proc/xen`.
- **xenstored not running.** `xl info` will fail.
- **Wrong DomU console.** Start with `console=hvc0`.
- **Too much DomU memory.** Use 64 MiB first.
- **Confusing host files and Dom0 files.** `kernel = "/boot/domu-zImage"` is a path inside Dom0.
- **Skipping restart.** Boot once, destroy, boot again.

133.18 Going deeper

- Xen `xl` documentation.
- Xen `xl.cfg` documentation.
- Xen event channels.
- Xen xenstore documentation.
- Chapter 135 for DomU on real i.MX6ULL.

Chapter 134: Xen on i.MX6ULL

> **What:** boot Xen on the real i.MX6ULL board, then let Xen boot a small Dom0 Linux.
>
> **Why:** QEMU helps explain the boot structure. Applying it to a physical board adds separate questions about U-Boot handoff mode, Device Tree, console ownership and RAM limits. The example below describes the intended contract, not a captured or qualified MINI boot.
>
> **Focus:** Xen is now the first privileged software after U-Boot. Linux is no longer bare metal.

134.1 What success looks like

The intended boot sequence makes the successive owners visible:

```
U-Boot prompt
  -> loads Xen
  -> loads Dom0 Linux zImage
  -> loads Dom0 initramfs
  -> loads and edits the board DTB
  -> jumps to Xen
Xen prints first
Xen creates Domain 0
Dom0 Linux boots
Dom0 reaches a shell
```

DomU, networking and passthrough depend on an earlier handoff working correctly. The smaller boot contract is:

```
U-Boot -> Xen -> Dom0 shell
```

Chapter 135 adds the DomU lifecycle to this model. Reading either chapter does not require executing this sequence.

134.2 Reality check for i.MX6ULL

The i.MX6ULL is a good learning target for Xen mechanics, but not a comfortable virtualization product target.

Property	i.MX6ULL consequence
One Cortex-A7 core	Xen can time-slice domains, but there is no physical parallelism.
512 MiB RAM on our board	Dom0 must be small. DomU must be tiny.
No spare A-core	Jailhouse-style static CPU partitioning is not useful here.
Real board DTB	Xen and Dom0 must agree on UART, GIC, timer, memory, and devices.
DMA-capable devices	Device assignment must be conservative without an IOMMU story.

The point is not to make the i.MX6ULL a server. The point is to see HYP mode, stage-2 translation, Dom0, and real boot handoff on the same board used in the rest of the book.

134.3 Prerequisites

You should already have:

- Chapter 24’s TFTP/NFS workflow or a working FAT boot partition.
- Chapter 26’s mainline Linux boot on the i.MX6ULL.
- Chapter 29’s initramfs skills.
- Chapter 132’s Xen-in-QEMU result.
- A serial console that captures the full U-Boot and Linux log.

Use serial logging. Hypervisor bring-up without a saved log is self-inflicted fog.

Host tools:

```
$ sudo apt install build-essential git bison flex bc libssl-dev \
    device-tree-compiler u-boot-tools gcc-arm-linux-gnueabihf
```

If your book workspace already uses a project-local toolchain, keep using it:

```
$ export CROSS_COMPILE=$HOME/imx6ull/toolchains/arm-gnu-toolchain-*/bin/arm-none-linux-
gnueabihf-
$ export ARCH=arm
```

In command examples below, adjust `CROSS_COMPILE` to match your actual toolchain prefix.

134.4 Artifact layout

Use one workspace directory for this chapter:

```
$ mkdir -p ~/imx6ull/xen-lab/{src,out,boot,logs}
$ cd ~/imx6ull/xen-lab
```

The final boot directory will contain:

```
boot/
  xen
  zImage-dom0
  dom0-initramfs.cpio.gz
  imx6ull-mini-xen.dtb
  boot-xen.scr
```

The names are intentionally boring. During failure analysis, boring filenames are kindness.

134.5 The boot contract

On 32-bit ARM, Xen follows the Linux `zImage` boot protocol closely enough that U-Boot can jump to it like a kernel image. The important differences:

- Xen must be entered in **HYP mode**.
- Xen requires **Device Tree**.
- U-Boot passes the DTB address in `r2`.
- Dom0 kernel and ramdisk are not passed as the normal Linux initramfs argument. They are described as **multiboot modules** in `/chosen`.

So U-Boot does not boot Dom0 directly. It boots Xen and hands Xen a DTB that says:

```
/chosen
  xen,xen-bootargs = "..."
  xen,dom0-bootargs = "..."
  module@...  -> Dom0 kernel
  module@...  -> Dom0 initramfs
```

Xen reads those nodes, creates Domain 0, and starts the Dom0 kernel.

134.6 Pick memory addresses

Our board DRAM starts at `0x80000000`. Use conservative, separated load addresses:

```
0x80800000  Xen
0x83000000  DTB
0x84000000  Dom0 zImage
0x88000000  Dom0 initramfs
```

Why these numbers:

- they are inside DRAM,
- they avoid the low DRAM region where boot code often works,
- they leave room between artifacts,
- they are easy to recognize in logs.

If your Dom0 initramfs is large, move it higher or make it smaller. Do not guess. Check sizes:

```
$ ls -lh boot/
```

134.7 Build Xen

Use a current supported Xen release. At the time this chapter was drafted, the Xen 4.21 series is current, and Xen 4.20 is also a supported series. If your package manager or release tarball has a newer supported stable release, use it and record the exact version in your configuration notes.

```
$ cd ~/imx6ull/xen-lab/src
$ git clone https://github.com/xen-project/xen.git
$ cd xen
$ git checkout RELEASE-4.21.1
```

Build only the hypervisor first:

```
$ make XEN_TARGET_ARCH=arm32 CROSS_COMPILE=arm-none-linux-gnueabihf- dist-xen
```

Expected output artifact:

```
$ ls -lh xen/xen
```

Copy it:

```
$ cp xen/xen ~/imx6ull/xen-lab/boot/xen
```

If your Xen tree produces a differently named ARM boot image, inspect the `xen/` directory:

```
$ find xen -maxdepth 1 -type f -printf '%f\n' | sort
```

The artifact must be the ARM Xen image, not the host tools and not an x86 binary.

134.8 Build a Dom0-capable Linux kernel

Start from the same Linux tree used in Part IV. Create a separate build directory so this experiment does not disturb your normal kernel:

```
$ cd ~/imx6ull/linux
$ make ARCH=arm O=../linux-dom0 imx_v6_v7_defconfig
```

Enable Xen support:

```
$ make ARCH=arm O=../linux-dom0 menuconfig
```

Minimum options to check:

```
CONFIG_XEN=y
CONFIG_XEN_DOM0=y
CONFIG_HVC_XEN=y
CONFIG_XEN_DEV_EVTCHN=y
CONFIG_XENFS=y
CONFIG_XEN_SYS_HYPERVISOR=y
CONFIG_XEN_BACKEND=y
CONFIG_XEN_BLKDEV_BACKEND=y
CONFIG_XEN_NETDEV_BACKEND=y
```

You do not need every Xen feature for the first boot, but Dom0 should have enough infrastructure for Chapter 135.

Build:

```
$ make ARCH=arm O=../linux-dom0 CROSS_COMPILE=arm-none-linux-gnueabihf- -j$(nproc) zImage
dtbs
```

Copy the kernel and the normal board DTB:

```
$ cp ../linux-dom0/arch/arm/boot/zImage ~/imx6ull/xen-lab/boot/zImage-dom0
$ cp ../linux-dom0/arch/arm/boot/dts/nxp/imx/imx6ull-*.dtb ~/imx6ull/xen-lab/boot/
```

Rename the exact DTB you use:

```
$ cd ~/imx6ull/xen-lab/boot
$ cp imx6ull-your-board.dtb imx6ull-mini-xen.dtb
```

Replace `imx6ull-your-board.dtb` with the DTB that already boots native Linux on your board.

134.9 Build a tiny Dom0 initramfs

Dom0 should be small. Reuse the BusyBox initramfs pattern from Chapter 29.

Minimum `/init`:

```
#!/bin/sh
mount -t proc none /proc
mount -t sysfs none /sys
mount -t devtmpfs none /dev
echo
echo "Dom0 is alive"
uname -a
cat /proc/cmdline
exec sh
```

Build the archive:

```
$ cd ~/imx6ull/xen-lab/dom0-root
$ find . -print0 | cpio --null -ov --format=newc | gzip -9 > ../boot/dom0-
initramfs.cpio.gz
```

Keep it tiny:

```
$ ls -lh ../boot/dom0-initramfs.cpio.gz
```

If it is tens of megabytes, stop and shrink it. The first Dom0 needs a shell, not a distribution.

134.10 Prepare the DTB

We will let U-Boot edit the DTB at boot time. That keeps the original board DTB reusable for native Linux.

The DTB must contain:

- the normal board description,
- `/chosen/xen,xen-bootargs`,

- `/chosen/xen,dom0-bootargs`,
- a Dom0 kernel module node,
- a Dom0 ramdisk module node.

The module nodes must include:

- `compatible`,
- `reg`,
- optional `bootargs` if you choose per-module bootargs.

The `reg` property must use the address-cell and size-cell format expected by the DTB. Many 32-bit ARM board DTBs use:

```
#address-cells = <1>;
#size-cells = <1>;
```

For those, a module looks like:

```
reg = <0x84000000 0x006a1234>;
```

Some generated or 64-bit-oriented trees use more cells. In that case the same module may need:

```
reg = <0x0 0x84000000 0x0 0x006a1234>;
```

Check your actual DTB before writing the U-Boot script:

```
$ dtc -I dtb -O dts imx6ull-mini-xen.dtb | grep -n "#address-cells\\|#size-cells" | head
```

This is a common source of “Xen cannot find Dom0” failures.

For the first boot, keep Xen arguments quiet but visible:

```
console=dtuart dtuart=serial0 dom0_mem=192M dom0_max_vcpus=1
```

Dom0 arguments:

```
console=hvc0 earlycon=xenboot root=/dev/ram0 rdinit=/init
```

`hvc0` is the Xen paravirtual console. If Dom0 prints nothing after Xen starts it, console selection is the first thing to suspect.

134.11 U-Boot boot script

Create `boot-xen.cmd`:

```
setenv xen_addr_r 0x80800000
setenv fdt_addr_r 0x83000000
setenv dom0_kernel_addr_r 0x84000000
setenv dom0_ramdisk_addr_r 0x88000000

setenv xen_file xen
setenv dom0_kernel_file zImage-dom0
setenv dom0_ramdisk_file dom0-initramfs.cpio.gz
setenv fdt_file imx6ull-mini-xen.dtb

fatload mmc 0:1 ${xen_addr_r} ${xen_file}
fatload mmc 0:1 ${dom0_kernel_addr_r} ${dom0_kernel_file}
setenv dom0_kernel_size ${filesize}
fatload mmc 0:1 ${dom0_ramdisk_addr_r} ${dom0_ramdisk_file}
setenv dom0_ramdisk_size ${filesize}
fatload mmc 0:1 ${fdt_addr_r} ${fdt_file}

fdt addr ${fdt_addr_r}
fdt resize 8192

fdt set /chosen xen,xen-bootargs "console=dtuart dtuart=serial0 dom0_mem=192M
dom0_max_vcpus=1"
fdt set /chosen xen,dom0-bootargs "console=hvc0 earlycon=xenboot root=/dev/ram0 rdinit=/
init"

fdt mknode /chosen module@84000000
fdt set /chosen/module@84000000 compatible "multiboot,kernel" "multiboot,module"
fdt set /chosen/module@84000000 reg <0x84000000 ${dom0_kernel_size}>

fdt mknode /chosen module@88000000
fdt set /chosen/module@88000000 compatible "multiboot,ramdisk" "multiboot,module"
fdt set /chosen/module@88000000 reg <0x88000000 ${dom0_ramdisk_size}>

fdt print /chosen
bootz ${xen_addr_r} - ${fdt_addr_r}
```

This script assumes the board DTB uses one address cell and one size cell for these `/chosen` module nodes. That is the normal shape for many 32-bit ARM board trees. If your DTB uses a different cell width, change both `fdt set ... reg` lines before booting.

Build the script:

```
$ mkimage -A arm -T script -C none -n "boot Xen on i.MX6ULL" \
    -d boot-xen.cmd boot-xen.scr
```

Copy `boot/` files to the FAT boot partition.

> **Why `fdt print /chosen` before boot?** Because it shows the exact handoff Xen will see. If Xen cannot find Dom0, the answer is often visible there.

134.12 First boot

At the U-Boot prompt:

```
=> load mmc 0:1 ${scriptaddr} boot-xen.scr
=> source ${scriptaddr}
```

Capture the full serial log:

```
$ picocom -b 115200 /dev/ttyUSB0 | tee ~/imx6ull/xen-lab/logs/xen-dom0-first-boot.log
```

Expected stages:

1. U-Boot loads four files.
2. `fdt print /chosen` shows Xen and Dom0 bootargs plus two module nodes.
3. Xen prints before Linux.
4. Xen reports CPU, GIC, timer, and memory.
5. Xen says it is loading or booting Domain 0.
6. Linux Dom0 prints with `hvc0` console.
7. `/init` prints `Dom0 is alive`.

If stage 3 never appears, U-Boot did not successfully enter Xen.

If stage 3 appears but stage 5 does not, Xen is alive but cannot create Dom0.

If stage 5 appears but Dom0 prints nothing, debug Dom0 console and kernel config.

134.13 Verification commands in Dom0

Inside the Dom0 shell:

```
# uname -a
# cat /proc/cmdline
# ls /proc/xen
# ls /sys/hypervisor
# dmesg | grep -i xen
```

If `/proc/xen` or `/sys/hypervisor` is missing, Dom0 either lacks Xen support or did not boot as a Xen domain.

For this chapter, `xl` is optional. The toolstack is useful in Chapter 135, but the first proof is lower-level: Dom0 knows it is running under Xen.

134.14 Troubleshooting

U-Boot rejects the Xen image

Symptoms:

```
Bad Linux ARM zImage magic
```

or no jump at all.

Check:

- you copied the ARM Xen image, not a host binary,
- `file boot/xen` on the host,
- Xen build target was `XEN_TARGET_ARCH=arm32`,
- the image was not corrupted by text-mode copy.

If your U-Boot cannot boot the raw Xen artifact with `bootz`, use a FIT image in a later pass. Do not mix FIT debugging into the first raw-module experiment.

Xen prints, then stops before Dom0

Most likely causes:

- missing `/chosen/module@...` nodes,
- bad `compatible` strings,
- bad `reg` address or size,
- Dom0 kernel address overlaps Xen or DTB,
- Dom0 memory too large.

Use:

```
=> fdt print /chosen
```

before `bootz`. The DTB handoff is evidence.

Dom0 starts but no shell appears

Most likely causes:

- wrong Dom0 `console=`,
- missing `CONFIG_HVC_XEN`,
- initramfs missing `/init`,
- `/init` not executable,
- wrong `rdinit=`.

Try Dom0 bootargs:

```
console=hvc0 earlycon=xenboot root=/dev/ram0 rdinit=/bin/sh
```

If `/bin/sh` works, your `/init` script is the bug.

Native Linux used UART but Dom0 uses `hvc0`

That is expected. Native Linux owns the physical UART. Dom0 under Xen normally speaks through the Xen console. Xen owns the early physical UART path.

Xen reports not in HYP mode

Xen requires HYP mode on ARM. Check U-Boot configuration and handoff path. This is not a Linux config problem. It is a firmware/CPU-mode problem.

134.15 Key Notes

- The physical boot path adds U-Boot and MINI-specific hardware dependencies before Xen can start. QEMU success cannot supply evidence for those dependencies.
- Xen, Dom0 and bootloader artifacts must agree on addresses, DTB contents, entry conditions and source versions.
- The i.MX6ULL has one Cortex-A7 core and a finite RAM budget. Virtual CPUs share that core rather than create physical parallelism.
- A successful Xen/Dom0 boot would establish a boot milestone, not safe peripheral assignment or deterministic guest timing.
- The supplied examples remain a design study unless supported by actual logs from the identified board revision and configuration.

134.16 Pitfalls

- **Treating Dom0 like native Linux.** Dom0 is privileged, but it is still a Xen domain.
- **Changing five things at once.** First boot uses serial console and initramfs only.
- **Using a large distro Dom0.** Save that for later. Boot the smallest possible rootfs first.
- **Forgetting exact versions.** Xen, Linux, and U-Boot version drift matters.
- **Assuming device assignment is safe.** Booting Dom0 says nothing about safe DMA isolation.
- **Skipping the DTB print.** If the `/chosen` nodes are wrong, Xen cannot guess your intent.

134.17 Going deeper

- Xen ARM booting documentation: `docs/misc/arm/booting.txt`.
- Xen ARM Device Tree boot modules: `docs/misc/arm/device-tree/booting.txt`.
- Xen command-line documentation.
- U-Boot `bootz`, `fdt`, and FIT image documentation.
- Chapter 135: DomU Linux on i.MX6ULL.
- Chapter 136: devices, memory, and DMA boundaries.

Chapter 135: DomU Linux on i.MX6ULL

> **What:** start a tiny DomU Linux guest on the real i.MX6ULL board after Xen and Dom0 are working.
>
> **Why:** QEMU proved the mechanics. The i.MX6ULL proves the real constraints: one CPU, small RAM, real storage, real serial logs, and no room for vague boot assumptions.
>
> **Focus:** success is isolation, not performance. A tiny DomU that can die without killing Dom0 is the goal. **DomU:** an unprivileged Xen guest domain. **Dom0:** the privileged Xen control domain that creates and manages DomU guests.

135.1 Starting point

Do not begin here until Chapter 134 works:

```
U-Boot -> Xen -> Dom0 Linux -> shell
```

Minimum Dom0 proof:

```
# uname -a
# cat /proc/cmdline
# ls /sys/hypervisor
# dmesg | grep -i xen
```

If Dom0 is unstable, adding DomU only hides the bug.

135.2 What changes from QEMU

QEMU gave us comfort:

- easy reset,
- plenty of fake RAM,
- configurable CPU count,
- no SD-card wear,
- easy file injection.

i.MX6ULL gives us reality:

- one Cortex-A7 core,

- 512 MiB RAM on the reference board,
- real U-Boot environment,
- real SD/eMMC,
- real serial console,
- real DTB,
- real memory pressure.

So the first DomU must be modest:

```
memory = 32 or 64 MiB
vcpus = 1
initramfs root
Xen console only
no networking
no block device
no passthrough
```

135.3 Upgrade Dom0 from “proof shell” to “control domain”

Chapter 134’s smallest Dom0 may not include `xl`. For DomU, Dom0 needs:

- Xen toolstack,
- xenstore,
- xenfs,
- event channel support,
- enough filesystem to hold DomU kernel/initramfs/config.

Check:

```
# which xl
# xl info
# xl list
```

If `xl` is missing, rebuild Dom0 rootfs with Xen tools, or use an NFS rootfs while developing.

135.4 Dom0 kernel options

Dom0 should have:

```
CONFIG_XEN=y
CONFIG_XEN_DOM0=y
CONFIG_HVC_XEN=y
CONFIG_XEN_DEV_EVTCHN=y
CONFIG_XENFS=y
CONFIG_XEN_SYS_HYPERVISOR=y
CONFIG_XEN_BACKEND=y
CONFIG_XEN_BLKDEV_BACKEND=y
CONFIG_XEN_NETDEV_BACKEND=y
```

For this chapter, block and net backend are not strictly needed, but enabling them avoids rebuilding immediately in the next experiment.

135.5 Prepare transfer path

You need a way to place DomU files into Dom0:

```
/boot/domu-zImage
/boot/domu-initramfs.cpio.gz
/etc/xen/imx6ull-domu.cfg
```

Good options:

- NFS mount from host,
- TFTP fetch into Dom0,
- copy to SD card boot partition,
- include files in Dom0 rootfs image.

For iteration, NFS is easiest:

```
# mkdir -p /mnt/host
# mount -t nfs -o nolock 192.168.7.1:/srv/nfs/imx6ull /mnt/host
# mkdir -p /boot /etc/xen
# cp /mnt/host/domu-zImage /boot/
# cp /mnt/host/domu-initramfs.cpio.gz /boot/
# cp /mnt/host/imx6ull-domu.cfg /etc/xen/
```

If NFS is not working yet, use the SD card. Do not debug NFS and DomU at the same time.

135.6 Build DomU kernel

Use the same Linux source as Chapter 133 or the Part IV kernel tree.

Configure a DomU build:

```
$ make ARCH=arm O=../linux-domu-imx6ull imx_v6_v7_defconfig
```

Enable:

```
$ scripts/config --file ../linux-domu-imx6ull/.config \
    -e XEN \
    -e HVC_XEN \
    -e XEN_DEV_EVTCHN \
    -e XENFS \
    -e BLK_DEV_INITRD \
    -e DEVTMPFS \
    -e DEVTMPFS_MOUNT

$ make ARCH=arm O=../linux-domu-imx6ull olddefconfig
```

Build:

```
$ make ARCH=arm O=../linux-domu-imx6ull \
    CROSS_COMPILE=arm-none-linux-gnueabihf- -j$(nproc) zImage
```

Copy:

```
$ cp ../linux-domu-imx6ull/arch/arm/boot/zImage /srv/nfs/imx6ull/domu-zImage
```

135.7 Build DomU initramfs

Use static BusyBox. DomU `/init`:

```
#!/bin/sh
mount -t proc none /proc
mount -t sysfs none /sys
mount -t devtmpfs none /dev

echo
echo "======================"
echo " i.MX6ULL DomU alive"
echo "======================"
echo

uname -a
cat /proc/cmdline
head -5 /proc/meminfo

echo
echo "Try killing me from Dom0 with: xl destroy imx6ull-domu"

exec /bin/sh
```

Pack:

```
$ find rootfs-domu -print0 | cpio --null -ov --format=newc | gzip -9 \
    > /srv/nfs/imx6ull/domu-initramfs.cpio.gz
```

Keep it small:

```
$ ls -lh /srv/nfs/imx6ull/domu-initramfs.cpio.gz
```

If it is large, stop and shrink it. This board has no RAM to waste.

135.8 Domain config

Create `imx6ull-domu.cfg`:

```
name = "imx6ull-domu"
kernel = "/boot/domu-zImage"
ramdisk = "/boot/domu-initramfs.cpio.gz"
memory = 64
vcpus = 1
extra = "console=hvc0 rdinit=/init"
on_crash = "preserve"
```

If 64 MiB is too much for your Dom0 memory budget, try 32 MiB. If 32 MiB is too small for your kernel/initramfs, reduce the initramfs before raising memory.

135.9 Start DomU

In Dom0:

```
# xl create -c /etc/xen/imx6ull-domu.cfg
```

Expected:

```
i.MX6ULL DomU alive
/ #
```

Detach:

```
Ctrl-]
```

List:

```
# xl list
```

Expected shape:

```
Name                                        ID   Mem VCPUs      State   Time(s)
Domain-0                                     0   ...     1     r-----     ...
imx6ull-domu                                 1    64     1     r-----     ...
```

135.10 Prove there are two Linux systems

In Dom0:

```
# uname -a
# cat /proc/cmdline
# xl list
```

In DomU:

```
# uname -a
# cat /proc/cmdline
# head -5 /proc/meminfo
```

Record both. The kernels may be built from the same source, but they are not the same running system.

135.11 Restart DomU

From Dom0:

```
# xl shutdown imx6ull-domu
# xl list
```

If graceful shutdown is not supported by the tiny guest:

```
# xl destroy imx6ull-domu
```

Start again:

```
# xl create -c /etc/xen/imx6ull-domu.cfg
```

The restart is important. A domain that boots only once is not an understood domain.

135.12 Prove failure isolation

Inside DomU:

```
# kill -SEGV 1
```

or from Dom0:

```
# xl destroy imx6ull-domu
```

Now prove Dom0 survived:

```
# date
# xl list
# dmesg | tail -40
```

The board must not reset. Dom0 must remain responsive.

This is the core result of the chapter.

135.13 Memory pressure check

In Dom0:

```
# free -m
# xl info | grep -i memory
```

Start DomU and run again:

```
# xl create /etc/xen/imx6ull-domu.cfg
# free -m
# xl list
```

Write down:

```
Dom0 memory before:
Dom0 memory after:
DomU assigned:
Free memory remaining:
```

This keeps the experiment honest. The i.MX6ULL is small.

135.14 What this proves

This chapter proves:

- Xen can create a second Linux domain on real i.MX6ULL hardware.
- Dom0 and DomU have separate assigned memory.
- DomU has a separate kernel command line.
- DomU can be destroyed without resetting the board.

- The serial log can show real HYP-mode virtualization, not only QEMU.

135.15 What this does not prove

It does not prove:

- good performance,
- hard real-time behavior,
- safe device passthrough,
- production-ready update flow,
- that single-core i.MX6ULL is a good multi-guest product platform.

One Cortex-A7 means one physical execution engine. Xen can schedule domains. It cannot create another core.

135.16 Failure cases on the physical platform

A guest allocation that exceeds the available whole-system budget can fail or starve Dom0. A missing initramfs or unusable `/init` breaks the guest's runtime contract. Selecting the physical UART as its console does not grant DomU ownership of that device.

A failure in guest user space may be contained while Dom0 remains operational, but that outcome must be observed for the actual setup. It does not establish recovery from a shared-device or control-domain failure.

135.17 Key Notes

- Dom0 and DomU have separate kernel and memory views while sharing the i.MX6ULL's one physical CPU. Scheduling capacity and failure containment are different objectives.
- Guest memory consumes a whole-platform budget alongside Dom0, Xen and supporting allocations. A small nominal guest size is not a complete memory account.
- Virtual console and managed guest interfaces avoid assuming that a physical UART or peripheral belongs to DomU.
- Destroying a guest can contain a workload failure, but differs from an orderly shutdown that protects persistent data.
- Guest survival tests do not establish DMA isolation, real-time guarantees or recovery from a failing control domain.

135.18 Pitfalls

- **Using a giant Dom0.** The board has limited RAM.
- **Using a giant DomU.** Start with 32 or 64 MiB.
- **Expecting physical UART in DomU.** Use `hvc0`.
- **Debugging DomU before `xl info` works.** Toolstack first.
- **Forgetting this is one physical CPU.** Do not expect parallel throughput.
- **Treating destroy as production shutdown.** It is fine for initial example, not for a product.
- **Skipping memory accounting.** Small boards punish optimism.

135.19 Going deeper

- Xen `xl` toolstack documentation.
- Xen ARM DomU documentation.
- Xen console documentation.
- Chapter 136 for device assignment and DMA boundaries.

Chapter 136: Devices, memory, and DMA boundaries

> **What:** decide which domain owns which hardware, and understand why DMA makes embedded virtualization dangerous if treated casually.
>
> **Why:** booting two kernels is only the easy half. The hard half is proving that one domain cannot corrupt another through devices, clocks, resets, interrupts, or DMA.
>
> **Focus:** CPU memory isolation is not the same as platform isolation. Stage 2 protects CPU translations. Devices need their own ownership story. **DMA:** Direct Memory Access, where a device reads or writes RAM without the CPU copying each byte. **Passthrough:** assigning a real hardware device to a guest. **Backend/frontend:** a split driver model where Dom0 owns real hardware and guests use virtual devices.

136.1 The ownership rule

Every real device needs one owner.

Owner means control of:

- MMIO registers,
- IRQ line,
- clock gate,
- reset line,
- pinctrl state,
- regulator or power switch,
- DMA channels,
- error recovery,
- suspend/resume policy,
- debug responsibility.

If two domains both think they own one controller, the design is already broken.

136.2 Start with the safest design

The safest first Xen design on i.MX6ULL:

```
Xen
  Dom0 Linux
    owns all real devices
    owns storage
    owns network
    owns UART
    owns watchdog
  DomU Linux
    uses Xen console
    uses initramfs
    no real devices
```

This is boring, and boring is good. It proves domain lifecycle before hardware sharing.

136.3 Device ownership table

Create `notes/device-ownership-imx6ull-xen.md`:

```
# Device ownership: i.MX6ULL Xen

| Device | MMIO | IRQ | DMA | Owner | Shared? | Notes |
|--------|------|-----|-----|-------|---------|-------|
| UART1 console |      |     | no  | Xen/Dom0 | no | early debug |
| FEC Ethernet |      |     | yes | Dom0 | virtual net later | DMA risk |
| USDHC SD/eMMC |     |     | yes | Dom0 | virtual block later | rootfs |
| GPIO1 |             |     | no  | Dom0 | avoid | mixed pins |
| I2C1 |              |     | maybe | Dom0 | proxy if needed | shared bus |
| watchdog |          |     | no | Dom0/safety | no | reset policy |
```

Fill MMIO and IRQ from the board DTB:

```
$ dtc -I dtb -O dts -o imx6ull-board.dts imx6ull-board.dtb
$ grep -n -A20 "uart" imx6ull-board.dts
$ grep -n -A20 "ethernet" imx6ull-board.dts
$ grep -n -A20 "usdhc" imx6ull-board.dts
```

Do not guess. Device Tree is the evidence.

136.4 Why GPIO banks are tricky

A GPIO controller owns many pins.

Example:

```
GPIO1_IO03  LED
GPIO1_IO09  button
GPIO1_IO16  reset line for peripheral
GPIO1_IO28  unrelated board strap
```

Passing the whole GPIO1 controller to DomU gives DomU access to every pin in that bank, not just “the LED.”

Safer pattern:

- Dom0 owns GPIO controller,
- Dom0 exposes a narrow service,
- DomU asks through a controlled channel.

For initial examples, do not pass GPIO controllers.

136.5 Why I2C and SPI buses are tricky

An I2C controller is a bus master. Many devices may sit behind it:

```
i2c1
  pmic
  rtc
  sensor
  touch controller
```

If DomU owns the controller, DomU can talk to everything on that bus.

SPI has the same issue with chip selects. Passing the controller may expose devices you did not intend to expose.

Safer pattern:

- Dom0 owns the bus,
- Dom0 runs the real driver,
- DomU receives data through virtual channel or shared memory.

136.6 CPU MMU vs DMA

CPU access:

```
guest virtual -> guest physical -> real physical
```

Stage 2 can block that.

DMA access:

```
device -> bus -> RAM
```

The CPU may not be involved.

If a guest owns a DMA-capable device, that device may be able to write outside the guest’s memory unless the platform has an IOMMU or bus firewall configured.

So before passthrough, ask:

```
Can this device DMA?
Can its DMA addresses be restricted?
Who programs the DMA descriptors?
Can the guest point DMA at Dom0 memory?
Does this SoC have an IOMMU for this device?
```

If you cannot answer, do not pass it through.

136.7 Device models ranked by risk

Model	Risk	Example
Dom0 owns real device, DomU has no access	lowest	initial examples
Dom0 owns real device, DomU uses paravirtual frontend	low	virtual network/block
Dom0 owns device, exports narrow custom service	medium	sensor proxy
DomU owns simple non-DMA MMIO device	medium	simple UART
DomU owns DMA-capable device	high	Ethernet, SD/MMC, USB
Multiple domains touch one controller	unacceptable first design	shared GPIO/I2C controller

Start at the top. Move down only when evidence forces you.

136.8 Watchdog ownership

Never leave watchdog ownership implicit.

Bad design:

```
Dom0 pets watchdog
DomU pets watchdog
maybe a script pets watchdog too
```

Good design:

```
Dom0 owns hardware watchdog.
DomU reports heartbeat to Dom0.
Dom0 pets watchdog only if Dom0 and required DomUs are healthy.
If DomU dies, Dom0 restarts DomU.
If Dom0 dies, watchdog resets board.
```

Even better for some products:

```
external MCU or M-core owns safety watchdog
Linux reports health
```

136.9 Reset ownership

Devices often share reset lines.

Example:

```
GPIO reset line controls radio module
same module also uses UART and power regulator
```

If DomU owns the UART but Dom0 owns reset, failure recovery requires cooperation.

Write it down:

```
Who can reset the device?
Who can power-cycle it?
Who reloads firmware?
Who handles stuck IRQ?
Who logs failures?
```

136.10 Clocks and pinctrl

Linux drivers assume they can request clocks, resets, pinctrl states, and regulators.

If DomU gets a device but not its supporting resources, the driver may fail or hang.

If DomU gets the supporting resources too broadly, it may affect Dom0 devices.

This is why passthrough on embedded SoCs is not just “map MMIO and IRQ.”

136.11 Shared communication patterns

Safer communication options:

- Xen console,
- Xen virtual network,
- Xen virtual block,
- event channels,
- grant tables,
- shared memory ring buffer,
- RPMsg/OpenAMP on heterogeneous SoCs.

For each shared channel, define:

```
Who allocates memory:
Who writes:
Who reads:
How notification works:
What happens if one side resets:
How protocol versions are checked:
```

136.12 Example: safe first i.MX6ULL design

```
Dom0:
  UART console
  SD/eMMC
  Ethernet
  GPIO/I2C/SPI
  watchdog
  update system

DomU:
  no passthrough
  initramfs or virtual block
  hvc console
  virtual network later
```

Product use:

- DomU runs risky app logic,
- Dom0 supervises,
- Dom0 can restart DomU,
- hardware remains under one kernel.

This is the first design to try.

136.13 Example: risky design

```
DomU owns Ethernet MAC directly.
```

Questions:

- Does FEC DMA?
- Can DMA be restricted?
- Who owns PHY reset GPIO?
- Who owns MDIO bus?
- Who owns clocks?
- Can DomU wedge the network hardware?
- Can Dom0 recover it?

If the answer is “not sure”, this is not a first design.

136.14 Device descriptions and their limits

A DTB can expose MMIO ranges, interrupts and references to clocks, pinctrl and reset resources. Those facts help reveal the dependency graph, but the DTB is a software description, not independent evidence of fitted wiring or DMA containment.

A device assignment must consider the whole usable resource set. Passing one controller while another domain changes its shared clock, GPIO bank or reset can violate both domains’ assumptions.

136.15 Key Notes

- Every hardware resource needs one control owner, including its configuration, reset and recovery path. Sharing requires a defined mediation interface.
- A GPIO pin may sit in a bank whose registers control many pins. The assignable hardware granularity can be larger than the application wants.
- I2C and SPI peripherals share a controller and bus state. Giving devices separate names does not create independent bus ownership.
- CPU stage-2 translation does not automatically confine DMA. A device’s bus-mastering reach needs its own isolation argument.

- Watchdog health, update compatibility and reset reasons are system policies. They cannot be delegated to every guest without a coordination rule.

136.16 Why rejecting passthrough can be the right design

A guest’s need for a device does not justify giving it the complete controller. Assignment can be unsuitable when DMA cannot be confined, supporting resources are shared, or no recovery owner can reset the device independently.

A mediated interface may keep physical ownership in Dom0 while exposing a narrower service. That adds protocol and availability dependencies, but makes the authority being granted explicit. Rejection is a design outcome, not a missing implementation.

136.17 Pitfalls

- **“It boots” mistaken for “it is isolated.”** Boot success says little about DMA safety.
- **Mapping MMIO without clocks/resets.** Drivers need supporting resources.
- **Passing GPIO banks casually.** A bank is many pins.
- **Passing DMA devices without an IOMMU story.** Dangerous.
- **Letting everyone pet the watchdog.** That hides failures.
- **No reset owner.** Stuck hardware needs a recovery path.
- **No update owner.** Device firmware and guest software must be versioned.
- **No logs.** A product that cannot explain reset reason cannot be debugged in the field.

136.18 Going deeper

- Linux DMA API documentation.
- Xen grant tables and event channels.
- Linux Device Tree bindings for the device you want to pass through.
- i.MX6ULL Reference Manual bus, DMA, clock, reset, and IOMUX chapters.
- STM32MP1 remoteproc/RPMsg/OpenAMP documentation.

Chapter 137: Jailhouse in QEMU ARM64

> **What:** boot a QEMU ARM64 Linux system, enable Jailhouse, and start a small inmate cell.
>
> **Why:** Jailhouse is useful only when you understand what it removes from Linux and gives to another cell.
>
> **Focus:** no magic VM language. We will track CPUs, RAM, MMIO, interrupts, and console ownership.

137.1 Why this chapter is not on i.MX6ULL

The i.MX6ULL has one Cortex-A7 core.

That matters.

Jailhouse is a partitioning hypervisor. Its cleanest teaching example is:

```
CPU 0..N: Linux root cell
CPU M: inmate cell
some RAM: Linux root cell
some RAM: inmate cell
some devices: Linux root cell
some devices: inmate cell
```

On a one-core chip, there is no spare CPU to hand to the inmate while Linux retains a core. The ownership model remains useful to study, but this multicore example is not a deployment recipe for the i.MX6ULL.

QEMU ARM64 can give us a fake machine with many CPUs. The official Jailhouse QEMU ARM64 configuration uses a virtual ARM machine with 1 GiB of RAM and multiple virtual CPUs. That is a much better initial example.

This is the honest path:

```
first: QEMU ARM64, so the model is visible
later: real board only when the hardware shape fits
```

The goal is not to pretend QEMU is your product. The goal is to learn the Jailhouse workflow without fighting board bring-up at the same time.

137.2 Jailhouse in one picture

Xen boots before Linux:

```
firmware -> Xen -> Dom0 Linux -> DomU guests
```

Jailhouse starts after Linux:

```
firmware -> Linux -> jailhouse.ko -> Jailhouse active -> inmate cells
```

Before Jailhouse is enabled, Linux owns the whole machine.

After Jailhouse is enabled, Linux becomes the **root cell**. Jailhouse then blocks Linux from touching resources that belong to other cells.

That is the core idea:

```
Linux does not become stronger.
Linux gives resources away.
Jailhouse enforces the new ownership table.
```

Jailhouse is not trying to emulate a PC for each guest. It is trying to partition real hardware.

137.3 Vocabulary we need

Root cell

The Linux system that booted first. It loads `jailhouse.ko`, enables Jailhouse, creates inmate cells, and remains responsible for management.

Inmate cell

The isolated workload. It can be bare-metal code, an RTOS, or sometimes a small Linux guest.

Cell config

A compiled hardware ownership table. It describes CPUs, memory regions, interrupt controllers, PCI or MMIO devices, and console settings.

Root-cell config

The system-wide config used when enabling Jailhouse. On QEMU ARM64 this is commonly built from `configs/arm64/qemu-arm64.c`.

Inmate config

The config for one non-root cell. On the QEMU ARM64 demo path this is commonly built from `configs/arm64/qemu-arm64-inmate-demo.c`.

Inmate binary

The code loaded into the inmate cell. For the first demo this is usually a small test binary such as `gic-demo.bin`.

137.4 What the QEMU ARM64 demo proves

The first Jailhouse demo should prove only five things:

1. Linux can boot on the QEMU ARM64 virtual machine.
2. Linux can load the Jailhouse kernel module.
3. Jailhouse can be enabled with the QEMU ARM64 root-cell config.
4. A non-root inmate cell can be created, loaded, and started.
5. Linux root cell stays alive while the inmate runs.

Do not add Zephyr yet.

Do not add a second Linux yet.

Do not edit the cell config yet.

First make the known-good demo work. Then change one thing at a time.

137.5 Workspace

Use a separate workspace because this example will create images, logs, configs, and notes:

```
$ mkdir -p ~/imx6ull/jailhouse-lab/{src,build,logs,notes}
$ cd ~/imx6ull/jailhouse-lab
```

Record host versions:

```
$ uname -a | tee logs/host-uname.txt
$ lsb_release -a 2>/dev/null | tee logs/host-release.txt
```

If your distribution does not have `lsb_release`, use:

```
$ cat /etc/os-release | tee logs/host-release.txt
```

137.6 Install host tools

Install the tools used by the demo and by later inspection:

```
$ sudo apt update
$ sudo apt install qemu-system-aarch64 git make gcc-aarch64-linux-gnu \
    device-tree-compiler flex bison libssl-dev bc cpio rsync file \
    docker.io
```

Check the important tools:

```
$ qemu-system-aarch64 --version | tee logs/qemu-version.txt
$ aarch64-linux-gnu-gcc --version | head -n 1 | tee logs/aarch64-gcc-version.txt
$ dtc --version | tee logs/dtc-version.txt
```

The upstream Jailhouse README says the ARM64 QEMU demo needs QEMU 3.0 or newer. Modern Ubuntu and Debian releases are normally far beyond that.

The `jailhouse-images` reference flow uses `kas-container`, so Docker must also work for your normal user or through `sudo`. Check:

```
$ docker --version
$ docker run --rm hello-world
```

If Docker permission fails, fix that before starting the image build. Do not debug Jailhouse while Docker itself is broken.

137.7 Get the source

Clone Jailhouse:

```
$ cd ~/imx6ull/jailhouse-lab/src
$ git clone https://github.com/siemens/jailhouse.git
$ cd jailhouse
$ git rev-parse HEAD | tee ../../logs/jailhouse-commit.txt
```

Now find the QEMU ARM64 configs:

```
$ ls configs/arm64 | tee ../../logs/jailhouse-arm64-configs.txt
$ ls inmates/demos/arm64 | tee ../../logs/jailhouse-arm64-inmates.txt
```

Expected names may include:

```
qemu-arm64.c
qemu-arm64-inmate-demo.c
gic-demo.c
```

Do not worry if the exact list changes across releases. The concepts do not change:

```
root-cell config
inmate-cell config
inmate binary
```

137.8 Use the reference image path first

Jailhouse has a lot of moving parts:

```
QEMU command line
Linux kernel image
root filesystem image
Jailhouse kernel module
Jailhouse user tool
root-cell config
inmate config
inmate binary
```

For the first successful run, use the Jailhouse reference image project:

```
$ cd ~/imx6ull/jailhouse-lab/src
$ git clone https://github.com/siemens/jailhouse-images.git
$ cd jailhouse-images
$ git rev-parse HEAD | tee ../../logs/jailhouse-images-commit.txt
```

The reference image project is built around `kas-container`. It provides a menu for virtual targets:

```
$ ./kas-container menu
```

Select the QEMU ARM64 Jailhouse target from the menu. After the image is generated, start it with:

```
$ ./start-qemu.sh arm64
```

This is intentionally the shortest path. The first time, your job is to get a working root cell and inmate cell, not to debug every package in the image build.

Expect this build to download and compile a full reference image stack. It is much heavier than building the tiny BusyBox initramfs in Chapter 129.

If the menu names have changed in your release, look at the repository README and `conf/multiconfig`:

```
$ ls conf/multiconfig
$ grep -R "arm64" -n conf scripts start-qemu.sh
```

Write down the exact target name you used:

```
Jailhouse image target:
```

That line belongs in your configuration notes.

137.9 The manual QEMU shape

The reference image scripts hide details. Before running Jailhouse commands, understand the QEMU machine they are creating.

The upstream ARM64 demo has this shape:

```
$ qemu-system-aarch64 \
    -cpu cortex-a57 \
    -smp 16 \
    -m 1G \
    -machine virt,gic-version=3,virtualization=on,its=off \
    -nographic \
    -netdev user,id=net \
    -device virtio-net-device,netdev=net \
    -drive file=LinuxInstallation.img,format=raw,id=disk,if=none \
    -device virtio-blk-device,drive=disk \
    -kernel Image \
    -append "root=/dev/vda1 mem=768M"
```

Do not copy this blindly yet. Use it as a map.

Important pieces:

Option	Meaning
`-cpu cortex-a57`	Run an ARMv8-A CPU model.
`-smp 16`	Create sixteen virtual CPUs.
`-m 1G`	Give the machine 1 GiB RAM.
`virtualization=on`	Expose virtualization support to the guest.
`gic-version=3`	Use a GICv3 interrupt controller.
`its=off`	Match the demo platform expectation.

Option	Meaning
`mem=768M`	Leave the top RAM area unused by Linux.

That last line is critical.

Linux sees only 768 MiB:

```
0x40000000 .. lower RAM used by Linux
```

The rest can be used by Jailhouse and inmate cells:

```
top RAM reserved for Jailhouse and inmates
```

If Linux uses the same memory as an inmate, isolation is already broken. The system may crash before you learn anything useful.

137.10 Inspect the root-cell config

Open the QEMU ARM64 root-cell config:

```
$ cd ~/imx6ull/jailhouse-lab/src/jailhouse
$ less configs/arm64/qemu-arm64.c
```

Find the comment near the top:

```
NOTE: Add "mem=768M" to the kernel command line.
```

That comment is not decoration. It is the contract between Linux boot arguments and the Jailhouse memory map.

Now search for the hypervisor memory region:

```
$ grep -n "hypervisor_memory" -A5 configs/arm64/qemu-arm64.c
```

You should see a physical start address and a size.

Write it in your notes:

```
Jailhouse hypervisor memory:
```

Now search for CPUs:

```
$ grep -n "cpus" -A4 configs/arm64/qemu-arm64.c
```

On this demo config, the CPU bitmap represents the CPUs available to the root cell config. A bitmap of `0xffff` means sixteen CPUs are described.

This is why the demo QEMU command uses:

```
-smp 16
```

The QEMU command and the cell config must describe the same machine.

137.11 Inspect the inmate config

Now open the inmate config:

```
$ less configs/arm64/qemu-arm64-inmate-demo.c
```

Look for:

```
.name
.cpus
.mem_regions
.irqchips
.console
```

The inmate config answers four practical questions:

```
Which CPU can the inmate run on?
Where is the inmate RAM?
Which interrupts can reach the inmate?
How does the inmate print?
```

This is the file you will annotate later.

137.12 Boot the reference image

Start the reference image:

```
$ cd ~/imx6ull/jailhouse-lab/src/jailhouse-images
$ ./start-qemu.sh arm64 | tee ../../logs/qemu-arm64-boot.txt
```

Login using the credentials documented by the image project.

Inside the QEMU guest, collect basic evidence:

```
# uname -a
# cat /proc/cmdline
# nproc
# free -m
```

Expected ideas:

```
many CPUs visible before Jailhouse is enabled
kernel command line contains mem=768M
memory is less than the full 1 GiB QEMU machine
```

The exact numbers depend on the image, but the shape should match.

137.13 Find Jailhouse inside the guest

Inside the QEMU guest:

```
# which jailhouse
# find /lib/modules -name "jailhouse.ko*"
# find /usr -name "*.cell" | grep jailhouse
# find /usr -name "gic-demo.bin" -o -name "*demo*.bin"
```

Common locations include:

```
/usr/share/jailhouse/cells/
/usr/libexec/jailhouse/
/lib/modules/.../extra/
```

Do not memorize paths. Learn to find the artifacts.

Record them:

```
jailhouse tool:
jailhouse module:
root-cell config:
inmate-cell config:
inmate binary:
```

137.14 Enable Jailhouse

Load the module:

```
# modprobe jailhouse
```

If that fails, check:

```
# dmesg | tail -n 80
# uname -r
# find /lib/modules/$(uname -r) -name "jailhouse.ko*"
```

The module must match the running kernel.

Now enable Jailhouse with the QEMU ARM64 root-cell config:

```
# jailhouse enable /path/to/qemu-arm64.cell
```

Replace the path with the real path you found in the image.

Collect evidence:

```
# dmesg | tail -n 120
# jailhouse cell list
```

Expected concept:

```
Jailhouse enabled
root cell listed
Linux still responsive
```

If the guest freezes here, do not continue. Debug the enable step first.

137.15 What changed after enable

Before enable:

```
Linux kernel controls the whole virtual machine.
```

After enable:

```
Jailhouse controls the partition boundaries.
Linux is still running, but only as the root cell.
```

This distinction matters for debugging.

If Linux tries to use memory that no longer belongs to it, Jailhouse can block that access.

If an inmate tries to touch a device that does not belong to it, Jailhouse can block that access.

If the cell config is wrong, the wrong thing gets blocked or the wrong thing is allowed.

So the cell config is not a helper file. It is the security boundary.

137.16 Create the inmate cell

Create the demo inmate:

```
# jailhouse cell create /path/to/qemu-arm64-inmate-demo.cell
```

List cells:

```
# jailhouse cell list
```

Expected concept:

```
root cell exists
inmate cell exists
inmate not running yet
```

Creating a cell builds the partition. It does not mean code is running inside it.

137.17 Load and start the inmate

Load the demo binary:

```
# jailhouse cell load inmate-demo /path/to/gic-demo.bin
```

Start it:

```
# jailhouse cell start inmate-demo
```

Check the cell list:

```
# jailhouse cell list
```

Expected concept:

```
inmate-demo running
root Linux still responsive
```

Depending on the demo image, output may appear on the serial console, the Jailhouse console, or the QEMU terminal. Check:

```
# jailhouse console
# dmesg | tail -n 120
```

The exact demo output is less important than the state transition:

```
created -> loaded -> running
```

137.18 Stop and disable

Stop the inmate if the demo supports shutdown:

```
# jailhouse cell shutdown inmate-demo
```

Destroy the cell:

```
# jailhouse cell destroy inmate-demo
```

Disable Jailhouse:

```
# jailhouse disable
```

If a running cell prevents disable, stop or destroy the cell and run disable again.

Final evidence:

```
# jailhouse cell list
# dmesg | tail -n 120
```

The clean development loop is:

```
enable root-cell config
create inmate
load inmate binary
start inmate
stop inmate
destroy inmate
disable Jailhouse
```

137.19 Build the mental map

Draw this in your notes:

```
QEMU virt machine

RAM: 1 GiB total
  lower area: Linux root cell
  top area: Jailhouse plus inmates

CPUs: 16 virtual CPUs
  root cell: Linux management side
  inmate: CPU subset from inmate config
```

```
Interrupt controller:
  GICv3

Console:
  root Linux console
  Jailhouse debug console
  inmate console path
```

Then fill in real addresses from the configs.

This is the moment where Jailhouse stops being a command sequence and becomes an ownership table.

137.20 Failure case: overlapping memory ownership

The reference setup reserves memory outside Linux’s usable range for Jailhouse and its cells. If Linux’s memory limit and cell configuration disagree, Linux can use pages another owner expects to control. Enabling may fail, or the inconsistent setup may become unstable.

The important boundary is exclusive ownership of physical memory, not the particular text of one launch argument.

137.21 Failure case: a mismatched CPU layout

A root-cell configuration describes a particular CPU topology and ownership mask. A virtual machine exposing fewer or different CPUs cannot satisfy that description. Enable or later cell creation can fail.

A configuration file is hardware-specific policy. Its format being valid does not establish that its resources exist on the running machine.

137.22 Failure case: an incompatible executable

An inmate image is linked for its assigned memory and entry conditions. A Linux user-space executable instead expects a loader, ABI and process environment. Loading its bytes into cell memory does not supply those services.

Cell loading is not `execve()`. An image may be rejected or fail at runtime depending on what the loader checks and what the binary assumes.

137.23 Annotate the root-cell config

Create a note file:

```
$ cp src/jailhouse/configs/arm64/qemu-arm64.c notes/qemu-arm64-annotated.c
```

Add comments beside:

```
hypervisor_memory
debug_console
platform_info.arm.gic_version
platform_info.arm.gicd_base
platform_info.arm.gicr_base
root_cell.name
cpus
mem_regions
irqchips
pci_devices
```

Your comments should answer:

```
what is this resource?
who owns it?
why is it here?
which QEMU option must match it?
```

This is more valuable than memorizing commands.

137.24 Annotate the inmate config

Copy the inmate config:

```
$ cp src/jailhouse/configs/arm64/qemu-arm64-inmate-demo.c notes/qemu-arm64-inmate-
annotated.c
```

Add comments beside:

```
cell name
CPU bitmap
memory regions
communication region
interrupt chip
console
```

For each memory region, classify it:

```
RAM
MMIO
communication
loadable area
read only
read write
executable
```

The flags tell you what the inmate is allowed to do with that region.

137.25 Why this is useful in real products

Jailhouse is useful when your product has a clear static split.

Good shape:

```
Linux UI and network stack on CPU 0..2
control workload on CPU 3
dedicated device for control workload
shared memory protocol between the two sides
```

Weak shape:

```
one CPU
many shared devices
unclear ownership
need to start and stop guests dynamically
need rich virtual hardware
```

For the weak shape, Xen, Linux processes, containers, PREEMPT_RT, or a separate MCU may be better.

The professional skill is not “use a hypervisor”. The professional skill is choosing the smallest boundary that solves the product problem.

137.26 Key Notes

- Jailhouse partitions resources after Linux has booted. It does not provide Xen’s general domain-management model.
- Root-cell and inmate descriptions must match actual CPUs, physical memory, MMIO and interrupts. Reservation and ownership must agree with Linux.
- Resources removed from the root cell must no longer be used there. Two drivers cannot independently own the same controller because their processes have different names.
- An inmate needs an executable and runtime contract consistent with its cell. A Linux application binary is not interchangeable with it.
- The multicore QEMU ARM64 study does not make Jailhouse’s spare-core partitioning available on the single-core i.MX6ULL.
- Successful cell startup demonstrates a particular ownership transition, not comprehensive platform or safety isolation.

137.27 Troubleshooting

`modprobe jailhouse` fails

The module may not match the running kernel.

Check:

```
# uname -r
# find /lib/modules/$(uname -r) -name "jailhouse.ko*"
# dmesg | tail -n 80
```

`jailhouse enable` freezes the guest

The root-cell config likely does not match the QEMU machine or Linux memory reservation.

Check:

```
QEMU RAM size
QEMU CPU count
QEMU GIC version
kernel command line
root-cell config
```

`cell create` fails

The inmate config may reference CPUs, memory, or IRQs not available to that cell.

Check the inmate config and the root-cell config together.

The inmate starts but prints nothing

The inmate may still be running. The console may be somewhere else.

Check:

```
# jailhouse cell list
# jailhouse console
# dmesg | tail -n 120
```

`jailhouse disable` fails

An inmate may still be running.

Stop or destroy non-root cells first:

```
# jailhouse cell list
# jailhouse cell shutdown inmate-demo
# jailhouse cell destroy inmate-demo
# jailhouse disable
```

137.28 What comes next

The reference example illustrates this ownership transition:

```
Linux root cell
Jailhouse module
root-cell config
inmate-cell config
inmate binary
cell lifecycle commands
```

Chapter 138 removes one more layer of magic. Instead of running the stock demo inmate, we will build and reason about a tiny non-Linux inmate workload.

Chapter 138: Bare-metal or Zephyr inmate cell

> **What:** build a tiny non-Linux workload and run it as a Jailhouse inmate cell.
>
> **Why:** the most useful Jailhouse product pattern is often Linux plus one small deterministic workload.
>
> **Focus:** start with bare metal, then move to Zephyr only after the cell contract is clear.

138.1 Why not start with Zephyr

Zephyr is not the hard part.

The hard part is the contract:

```
where does the inmate start?
where is its RAM?
which CPU runs it?
which MMIO regions can it touch?
which interrupt lines can reach it?
how does it talk to Linux?
```

If you start with Zephyr and it fails, you have too many possible causes:

```
wrong Jailhouse cell config
wrong link address
wrong Zephyr board target
wrong devicetree
wrong console driver
wrong interrupt setup
wrong shared memory setup
```

So we start with a tiny bare-metal inmate. It is not a product. It is a flashlight.

Once the flashlight works, Zephyr becomes much less mysterious.

138.2 The inmate is not a Linux program

A normal Linux program starts because the kernel does a lot of work:

```
loads ELF segments
sets up virtual memory
maps shared libraries
```

```
builds a stack
passes argc and argv
jumps to user mode
```

A Jailhouse inmate does not get that environment.

For a simple bare-metal inmate, you provide:

```
entry point
link address
stack
exception vector choice if needed
MMIO access code
loop or shutdown path
```

That is why this command is wrong:

```
# jailhouse cell load inmate-demo /bin/ls
```

`/bin/ls` expects Linux. An inmate binary expects the cell environment.

138.3 The three addresses

Keep these three addresses separate:

Address	Meaning
Cell memory physical address	The RAM range assigned in the inmate cell config.
Inmate link address	The address the compiler and linker assume at build time.
Load address	The address where `jailhouse cell load` places the binary.

For the simplest bare-metal inmate, make all three match:

```
cell memory starts at 0x7c000000
binary is linked at 0x7c000000
binary is loaded at 0x7c000000
```

If the cell config gives the inmate RAM at one address but the binary is linked for another, the first branch or global variable access may go into nonsense.

138.4 Workspace

Continue from the previous chapter:

```
$ mkdir -p ~/imx6ull/jailhouse-lab/inmates/baremetal
$ cd ~/imx6ull/jailhouse-lab/inmates/baremetal
```

We will create:

```
start.S
main.c
inmate.ld
Makefile
README-notes.txt
```

This chapter shows the shape. You will fill in the exact base addresses from your QEMU ARM64 inmate config.

138.5 Read the demo inmate config first

Open the demo inmate config:

```
$ cd ~/imx6ull/jailhouse-lab/src/jailhouse
$ less configs/arm64/qemu-arm64-inmate-demo.c
```

Find the RAM region assigned to the inmate.

Write it down:

```
inmate RAM physical start:
inmate RAM size:
inmate load address:
inmate CPU:
inmate console type:
inmate console MMIO base:
```

Do not write code until those lines are filled.

138.6 The smallest useful startup code

Create `start.S`:

```
.section .text.start
.global _start

_start:
    ldr x0, =_stack_top
    mov sp, x0
```

```
    bl inmate_main

1:
    wfi
    b 1b
```

What this does:

```
set stack pointer
call C code
wait forever when C returns
```

There is no C runtime here. No `main(argc, argv)`. No libc. No exit syscall.

138.7 The linker script

Create `inmate.ld`.

Replace `0x7c000000` with the inmate RAM start from your cell config:

```
ENTRY(_start)

MEMORY
{
    RAM (rwx) : ORIGIN = 0x7c000000, LENGTH = 16M
}

SECTIONS
{
    . = ORIGIN(RAM);

    .text : {
        *(.text.start)
        *(.text*)
    } > RAM

    .rodata : {
        *(.rodata*)
    } > RAM

    .data : {
        *(.data*)
    } > RAM

    .bss : {
        __bss_start = .;
        *(.bss*)
        *(COMMON)
        __bss_end = .;
    } > RAM

    . = ALIGN(16);
    . = ORIGIN(RAM) + LENGTH(RAM);
```

```
    _stack_top = .;
}
```

This script says:

```
put code at the start of inmate RAM
put read only data after code
put initialized data after that
put zero data after that
place stack at the top of inmate RAM
```

For a real inmate, you may reserve a fixed stack size and detect overflow. For the initial example, this is enough.

138.8 A C file that does not need libc

This first version uses a PL011 UART because it is easy to understand. Treat it as a template, not as a guaranteed QEMU ARM64 Jailhouse demo binary.

Create `main.c`:

```
#include <stdint.h>

#define UART_BASE 0x09000000UL
#define UART_DR   (*(volatile uint32_t *)(UART_BASE + 0x00))
#define UART_FR   (*(volatile uint32_t *)(UART_BASE + 0x18))
#define UART_FR_TXFF (1U << 5)

static void uart_putc(char c)
{
    while (UART_FR & UART_FR_TXFF) {
    }

    UART_DR = (uint32_t)c;
}

static void uart_puts(const char *s)
{
    while (*s != '\0') {
        if (*s == '\n') {
            uart_putc('\r');
        }
        uart_putc(*s);
        s++;
    }
}

void inmate_main(void)
{
    uart_puts("hello from bare-metal Jailhouse inmate\n");

    for (;;) {
        __asm__ volatile("wfi");
```

```
    }
}
```

This assumes a PL011 UART at `0x09000000`, which is the QEMU `virt` UART base used by many ARM examples.

But do not trust the example. Check the inmate config:

```
If the UART is not assigned to the inmate, this code is wrong.
If Linux still owns that UART, this code is wrong.
If the console uses a virtual Jailhouse console instead, this code is not the right
output path.
```

One device has one owner. That rule is the whole game.

The stock Jailhouse `gic-demo.bin` is a better first proof because its source and cell config are already matched by the Jailhouse project. Your own PL011 inmate is the next experiment after you have identified which console or MMIO region the inmate really owns.

138.9 Build without libc

Create `Makefile`:

```
CROSS_COMPILE ?= aarch64-linux-gnu-
CC := $(CROSS_COMPILE)gcc
OBJCOPY := $(CROSS_COMPILE)objcopy

CFLAGS := -Wall -Wextra -ffreestanding -fno-stack-protector -fno-pic \
          -mgeneral-regs-only -O2
LDFLAGS := -nostdlib -Wl,-T,inmate.ld -Wl,-Map,inmate.map

all: inmate.bin

inmate.elf: start.S main.c inmate.ld
    $(CC) $(CFLAGS) $(LDFLAGS) start.S main.c -o $@

inmate.bin: inmate.elf
    $(OBJCOPY) -O binary $< $@

clean:
    rm -f inmate.elf inmate.bin inmate.map
```

Build:

```
$ make
$ file inmate.elf inmate.bin
$ aarch64-linux-gnu-objdump -h inmate.elf
```

Check the map:

```
$ grep -E "_start|_stack_top|\\.text" inmate.map
```

Expected idea:

```
_start is at the inmate link address
stack top is inside the inmate RAM region
```

138.10 Load your inmate

Boot the QEMU ARM64 Jailhouse image from Chapter 137.

Copy your binary into the guest. Use the easiest path available:

```
$ scp inmate.bin root@QEMU_IP:/root/
```

If networking is not available, use a shared folder, rebuild the image, or copy through the QEMU disk image. The transport is not the lesson.

Inside the QEMU guest, first prove the stock demo still works:

```
# modprobe jailhouse
# jailhouse enable /path/to/qemu-arm64.cell
# jailhouse cell create /path/to/qemu-arm64-inmate-demo.cell
# jailhouse cell load inmate-demo /path/to/gic-demo.bin
# jailhouse cell start inmate-demo
```

Watch the console or the output path used by the demo image.

Expected result:

```
the stock inmate produces timer or interrupt output
Linux root cell stays alive
```

Only after that proof should you shut down the cell and load your own binary:

```
# jailhouse cell shutdown inmate-demo
# jailhouse cell load inmate-demo /root/inmate.bin
# jailhouse cell start inmate-demo
```

Expected result if the PL011 ownership assumption is correct:

```
hello from bare-metal Jailhouse inmate
```

If you do not see output, first check whether the inmate is running:

```
# jailhouse cell list
```

A silent inmate is not automatically a dead inmate.

138.11 If the UART path is wrong

Many first attempts fail because the console assumption is wrong.

There are three common output paths:

```
real or emulated UART MMIO
Jailhouse virtual debug console
shared memory buffer
```

If your config does not assign the PL011 UART to the inmate, do not keep poking `0x09000000`.

Instead, use the demo binary and config as the reference:

```
# jailhouse cell load inmate-demo /path/to/gic-demo.bin
# jailhouse cell start inmate-demo
```

If the stock demo prints but your binary does not, compare:

```
entry address
link address
console path
binary format
CPU mode expectation
```

If neither prints, debug the cell config and QEMU console setup.

138.12 Add a shared-memory heartbeat

Printing is useful, but products communicate through protocols.

Add one shared page to the inmate config only after the print path works.

A simple memory layout:

```
offset 0x00: magic
offset 0x04: version
offset 0x08: root_to_inmate_counter
offset 0x0c: inmate_to_root_counter
offset 0x10: status
```

Define a struct:

```
#include <stdint.h>

#define SHMEM_BASE 0x7faf0000UL
```

```
struct shared_page {
    volatile uint32_t magic;
    volatile uint32_t version;
    volatile uint32_t root_to_inmate_counter;
    volatile uint32_t inmate_to_root_counter;
    volatile uint32_t status;
};

static struct shared_page *const shmem = (void *)SHMEM_BASE;
```

Then update a counter:

```
shmem->magic = 0x4a48494dU;
shmem->version = 1;

for (;;) {
    shmem->inmate_to_root_counter++;
    __asm__ volatile("wfi");
}
```

The root cell can inspect that memory only if the root cell also has a mapping for it through a driver or test tool.

Direct `/dev/mem` access bypasses the ownership and lifecycle interfaces a product protocol needs. A readable address is not evidence of a safe shared-memory design.

138.13 Shared memory rules

Shared memory is where many mixed-criticality designs get messy.

Use rules:

```
fixed layout
explicit version
single writer for each field
bounded message size
clear reset state
no pointers across the boundary
no unbounded strings
no shared ownership of hardware registers
```

Bad protocol:

```
Linux writes any field whenever it wants
inmate writes any field whenever it wants
both sides share pointers
both sides assume the other side is alive
```

Good protocol:

```
Linux owns command fields
inmate owns status fields
both sides check version
both sides can detect stale data
watchdog policy is written down
```

This is more important than the hypervisor.

138.14 Reload loop

During development you need a repeatable loop:

```
# jailhouse cell shutdown inmate-demo
# jailhouse cell destroy inmate-demo
# jailhouse cell create /path/to/qemu-arm64-inmate-demo.cell
# jailhouse cell load inmate-demo /root/inmate.bin
# jailhouse cell start inmate-demo
```

If `shutdown` is not implemented, `destroy` may be the only practical development path.

Record what your inmate supports:

```
clean shutdown: yes or no
reload without disabling Jailhouse: yes or no
requires full QEMU restart: yes or no
```

Production systems need a defined stop and recovery path. An illustrative startup sequence does not establish that interrupts, buffers and device state are safe to reuse after stopping the inmate.

138.15 Failure case: inconsistent addresses

The linker, cell memory map and loader must describe the same runtime location. When the binary assumes a different origin, its references or entry conditions can target unassigned memory and fail.

A raw binary does not carry a relocation solution merely because its bytes were accepted by the loader.

138.16 Failure case: unassigned MMIO

An inmate can only rely on devices whose assigned MMIO and supporting resources match its configuration. An incorrect UART base may cause an access fault, no output or a wait that never completes.

Silence is ambiguous: it does not distinguish a missing device, a protection violation and a wrong console path by itself.

138.17 Failure case: invalid runtime state

A C call needs a valid stack in writable assigned memory. If the stack pointer lies outside that region, the first spill or call can fail before any useful diagnostic output.

Small bare-metal code still has runtime dependencies. Removing libc does not remove stack, alignment or initialized-state requirements.

138.18 Move to Zephyr

Now Zephyr is reasonable.

Zephyr gives you:

```
scheduler
drivers
devicetree
logging
timers
IPC libraries
build system
```

But Zephyr still must obey the same Jailhouse contract:

```
link address matches cell RAM
drivers match assigned devices
interrupts match assigned IRQs
shared memory matches both sides
console path is real
```

The cell config does not become optional because Zephyr is present.

138.19 Install Zephyr tools

Use a Python virtual environment:

```
$ sudo apt install python3-venv python3-pip
$ mkdir -p ~/imx6ull/zephyr-lab
$ cd ~/imx6ull/zephyr-lab
$ python3 -m venv .venv
$ . .venv/bin/activate
$ pip install west
```

Fetch Zephyr:

```
$ west init zephyrproject
$ cd zephyrproject
$ west update
$ west zephyr-export
$ pip install -r zephyr/scripts/requirements.txt
```

Record versions:

```
$ west --version | tee ~/imx6ull/jailhouse-lab/logs/west-version.txt
$ git -C zephyr rev-parse HEAD | tee ~/imx6ull/jailhouse-lab/logs/zephyr-commit.txt
```

138.20 Build a normal Zephyr QEMU sample first

Before Jailhouse, prove Zephyr builds:

```
$ cd ~/imx6ull/zephyr-lab/zephyrproject/zephyr
$ west boards | grep -i qemu
$ west build -b qemu_cortex_a53/qemu_cortex_a53 samples/hello_world
```

If your Zephyr release uses the older short board name, use:

```
$ west build -b qemu_cortex_a53 samples/hello_world
```

Use the exact target printed by `west boards`. The important part is that this is the ARM64 QEMU Cortex-A53 board, not a Cortex-M board.

Run the normal Zephyr QEMU target if supported by your setup:

```
$ west build -t run
```

This test is not Jailhouse. It proves the Zephyr toolchain and board support before you mix in cells.

138.21 What must change for Zephyr as an inmate

A normal Zephyr QEMU board assumes it owns the machine.

A Zephyr inmate owns only a partition.

You must align:

```
Zephyr link address
Zephyr RAM size
Zephyr device tree
Zephyr console
Zephyr interrupt controller view
Jailhouse inmate cell config
```

Start from the smallest sample:

```
samples/hello_world
```

Avoid networking, storage, and complex drivers until hello-world prints from the inmate cell.

138.22 Zephyr resource-table path

For Linux plus a remote processor, Zephyr has OpenAMP resource-table samples. That is especially relevant on STM32MP1 in Chapter 139.

For Jailhouse, the resource table is not the root mechanism. Jailhouse uses cell configs.

So separate the two paths:

```
Jailhouse inmate on A-core:
  Jailhouse cell config is the hardware ownership contract.

STM32MP1 M4 firmware:
  remoteproc resource table and RPMsg describe communication resources.
```

This distinction prevents a lot of confusion.

138.23 Key Notes

- Cell configuration, link address, entry point and stack form one runtime contract. They must agree before a C function can execute.
- A private console and shared-memory heartbeat are different evidence paths. Neither should assume ownership of hardware still used by root Linux.
- Shared memory needs a protocol for ownership, ordering, validity and peer restart. A visible counter alone does not specify reliable communication.
- Loading, stopping and reloading a cell must account for stale state and supporting resources, not merely replace instruction bytes.
- Moving to Zephyr adds board definitions, drivers and scheduling assumptions. A normal QEMU Zephyr build does not establish compatibility as a Jailhouse inmate.

138.24 The Zephyr adaptation boundary

Zephyr’s configured RAM base and size, console device, CPU assumptions and shared-memory description must match the cell’s assigned resources. Its startup and interrupt model also have to work under that entry contract.

A resource table used by a remoteproc integration is not automatically a Jailhouse cell description. Those mechanisms have different loaders and ownership rules, so copying metadata between them does not establish an adaptation.

138.25 Product lesson

The useful product pattern is not:

```
we ran an RTOS under a hypervisor
```

The useful product pattern is:

```
Linux owns complex services
the inmate owns one bounded real-time responsibility
the two sides communicate through a small protocol
the hardware ownership table is reviewed like safety code
```

That is why the bare-metal inmate matters. It shows the boundary without hiding it inside an RTOS.

138.26 Troubleshooting

The inmate does not print

Check:

```
is the cell running?
is the console assigned?
does the UART base match the assigned MMIO?
does the inmate use the right output path?
```

The inmate crashes immediately

Check:

```
link address
load address
stack address
binary format
cell RAM flags
```

The root cell freezes

Check:

```
memory overlap
device ownership overlap
```

```
wrong interrupt assignment
bad cell config
```

Zephyr builds but does not run as inmate

Return to the bare-metal inmate.

Then compare:

```
entry point
linker map
devicetree memory
console device
interrupt controller setup
```

138.27 What comes next

Jailhouse taught us static partitioning.

Chapter 139 studies a different and very practical production pattern: Linux on Cortex-A7 plus RTOS firmware on a Cortex-M4 companion core. That is not a hypervisor lab, but for STM32MP1 it is often the better architecture.

Chapter 139: STM32MP1 Linux plus RTOS, the production pattern

> **What:** build the mental and practical path for Linux on STM32MP1 Cortex-A7 plus firmware on the Cortex-M4.
>
> **Why:** many real Linux plus RTOS products do not need a hypervisor. They need the SoC’s companion microcontroller core used properly.
>
> **Focus:** remoteproc, RPMsg, OpenAMP, firmware ownership, update policy, and when Jailhouse still makes sense.

139.1 Why this chapter belongs in the hypervisor part

This chapter is not a Jailhouse chapter.

That is the point.

A good engineer does not force every isolation problem into a hypervisor. On STM32MP1, the hardware already gives you a natural split:

```
Cortex-A7 cores: Linux
Cortex-M4 core: RTOS or bare-metal firmware
```

That is often a better product boundary than running another A-core guest.

The hypervisor lesson is:

```
choose the boundary that fits the hardware
```

For i.MX6ULL:

```
one Cortex-A7
no companion M-core
```

For STM32MP157:

```
two Cortex-A7 cores
one Cortex-M4 core
```

If your real-time workload fits on the M4, use the M4 first. Save Jailhouse for problems that really need A-core partitioning.

139.2 The production shape

The common STM32MP1 product shape:

```
Linux on Cortex-A7:
  UI
  networking
  storage
  update agent
  logging
  supervision

RTOS or bare metal on Cortex-M4:
  fast control loop
  simple safety monitor
  deterministic peripheral handling
  wakeup task
  small protocol endpoint

communication:
  RPMsg over shared memory
  OpenAMP conventions
  mailbox or interrupt notifications

management:
  Linux remoteproc loads and starts M4 firmware
```

Linux and M4 are not two Linux processes. They are two processors in one SoC.

139.3 What remoteproc does

`remoteproc` is a Linux kernel framework for controlling another processor in the same system.

On a supported SoC, Linux can:

```
load firmware
start the remote processor
stop the remote processor
report crashes
create RPMsg virtio devices when the firmware announces them
```

The key idea:

```
Linux is the supervisor.
M4 firmware is firmware.
```

The M4 firmware is not a Linux process. It does not use Linux syscalls. It does not share the Linux scheduler.

139.4 What RPMsg does

`RPMsg` is a message bus for communication with remote processors.

The common implementation uses:

```
shared memory buffers
virtio vrings
mailbox or interrupt notifications
named channels
endpoints
```

The mental model:

```
Linux app
  -> Linux rpmsg driver
  -> RPMsg channel
  -> shared memory vring
  -> M4 OpenAMP endpoint
```

This is message passing. It is not two CPUs casually writing each other’s global variables.

139.5 What OpenAMP does

OpenAMP is the ecosystem around this kind of asymmetric multiprocessing.

It gives firmware-side pieces for:

```
RPMsg
virtio
resource tables
shared memory setup
remote processor communication
```

In practice, Zephyr can use OpenAMP samples to talk to Linux RPMsg drivers.

Important distinction:

```
Jailhouse cell config describes A-core partitioning.
remoteproc resource table describes remote processor resources.
```

Do not mix those concepts.

139.6 First board assumptions

This chapter assumes an STM32MP157 Discovery style board such as:

```
STM32MP157C-DK2
STM32MP157F-DK2
similar STM32MP15 board with Cortex-M4 enabled
```

Board names vary by BSP and Zephyr release. You must check your exact board.

On the Linux side:

```
# cat /proc/device-tree/model
# uname -a
# ls /sys/class/remoteproc
```

On the Zephyr side:

```
$ west boards | grep -i stm32mp
```

Write down:

```
board model:
Linux BSP:
kernel version:
Zephyr board target:
remoteproc node:
```

139.7 Linux kernel features to check

Boot Linux on the STM32MP1 board.

Check remoteproc:

```
# ls /sys/class/remoteproc
```

Expected shape:

```
remoteproc0
```

Check kernel messages:

```
# dmesg | grep -i remoteproc
# dmesg | grep -i rpmsg
```

Check config if available:

```
# zcat /proc/config.gz | grep -E "REMOTE_PROC|RPMSG|STM32"
```

Useful symbols may include:

```
CONFIG_REMOTEPROC
CONFIG_STM32_RPROC
CONFIG_RPMSG
CONFIG_RPMSG_CHAR
CONFIG_RPMSG_TTY
```

Names vary by kernel version and BSP. The important evidence is:

```
Linux exposes a remoteproc device for the M4
Linux can create RPMsg devices when firmware supports them
```

139.8 Device Tree features to check

Remoteproc is SoC specific. If the Device Tree does not describe the M4 and shared memory correctly, Linux cannot manage it.

Inspect the running tree:

```
# find /proc/device-tree -iname "*m4*" -o -iname "*rproc*" -o -iname "*rpmsg*"
```

Also save the boot DTB if your boot flow makes it available.

Things to look for:

```
M4 remoteproc node
reserved memory for firmware or vrings
mailbox nodes
shared memory regions
RPMsg related nodes
status = "okay"
```

If `/sys/class/remoteproc` is empty, debug Device Tree and kernel config before debugging Zephyr.

139.9 Build the simplest M4 firmware first

Start with a firmware that does one visible thing:

```
prints on the M4 console
or toggles an LED owned by M4
or creates an RPMsg hello channel
```

Do not start with a control loop.

Do not start with a product protocol.

First prove:

```
Linux can load firmware
Linux can start M4
M4 actually runs
Linux can stop M4
Linux can start M4 again
```

139.10 Zephyr setup

Use a separate workspace:

```
$ sudo apt install python3-venv python3-pip
$ mkdir -p ~/stm32mp1-m4-lab
$ cd ~/stm32mp1-m4-lab
$ python3 -m venv .venv
$ . .venv/bin/activate
$ pip install west
$ west init zephyrproject
$ cd zephyrproject
$ west update
$ west zephyr-export
$ pip install -r zephyr/scripts/requirements.txt
```

Find STM32MP1 boards:

```
$ cd zephyr
$ west boards | grep -i stm32mp
```

A board name may look like:

```
stm32mp157c_dk2/stm32mp157cxx
```

Older Zephyr releases may accept the shorter `stm32mp157c_dk2` name. Use the board target from your Zephyr checkout, not from memory.

139.11 Build a basic Zephyr sample

Build hello world:

```
$ west build -b stm32mp157c_dk2/stm32mp157cxx samples/hello_world
```

If your Zephyr release uses the older short board name, replace the target with `stm32mp157c_dk2`. If your board revision differs, use the target shown by `west boards`.

Find the output:

```
$ ls build/zephyr
$ file build/zephyr/zephyr.elf
```

For remoteproc, the ELF is often the useful artifact because it contains loadable sections and metadata. Some BSP flows use a different packaged firmware file. Use the format required by your Linux remoteproc driver and board documentation.

Record:

```
Zephyr board:
Zephyr sample:
firmware file:
firmware size:
```

139.12 Install firmware on the board

Copy the firmware to the board:

```
$ scp build/zephyr/zephyr.elf root@BOARD_IP:/lib/firmware/stm32mp1-m4.elf
```

On the board:

```
# ls -l /lib/firmware/stm32mp1-m4.elf
# file /lib/firmware/stm32mp1-m4.elf
```

The firmware path matters because remoteproc asks the Linux firmware loader for a file name.

139.13 Start M4 with remoteproc

On the board:

```
# ls /sys/class/remoteproc
# cat /sys/class/remoteproc/remoteproc0/state
# echo stm32mp1-m4.elf > /sys/class/remoteproc/remoteproc0/firmware
# echo start > /sys/class/remoteproc/remoteproc0/state
# cat /sys/class/remoteproc/remoteproc0/state
```

Expected result:

```
running
```

Check logs:

```
# dmesg | tail -n 120
```

Stop:

```
# echo stop > /sys/class/remoteproc/remoteproc0/state
# cat /sys/class/remoteproc/remoteproc0/state
```

Expected result:

```
offline
```

Start again:

```
# echo start > /sys/class/remoteproc/remoteproc0/state
```

This start-stop-start loop is your first production-relevant test.

139.14 If start fails

Do not guess. Classify the failure.

Firmware file not found

Check:

```
# ls -l /lib/firmware
# dmesg | tail -n 80
```

Bad firmware format

Check:

```
# file /lib/firmware/stm32mp1-m4.elf
# readelf -h /lib/firmware/stm32mp1-m4.elf
```

Wrong load address

Check the Zephyr linker map and the STM32MP1 memory layout.

Remoteproc node missing

Check kernel config and Device Tree.

M4 starts but no output

Check which UART, LED, or RPMsg endpoint the firmware uses. The M4 may be running with no visible output path.

139.15 Move from hello world to RPMsg

Once start and stop work, build an RPMsg/OpenAMP sample.

Zephyr has an OpenAMP resource-table sample:

```
samples/subsys/ipc/openamp_rsc_table
```

Build shape:

```
$ west build -b stm32mp157c_dk2/stm32mp157cxx samples/subsys/ipc/openamp_rsc_table
```

If the sample requires an overlay for your board, add it only after reading the sample README and your board documentation.

The firmware must describe the RPMsg resources in a way Linux remoteproc understands. That is what the resource table is for.

139.16 Linux RPMsg checks

After starting RPMsg-capable firmware:

```
# dmesg | grep -i rpmsg
# ls /sys/bus/rpmsg/devices
# ls /dev/rpmsg* 2>/dev/null
# ls /dev/ttyRPMSG* 2>/dev/null
```

Depending on the Linux BSP, user-space access may appear as:

```
/dev/rpmsg_ctrl*
/dev/rpmsg*
/dev/ttyRPMSG*
kernel sample driver only
```

The exact device node is less important than the evidence:

```
the M4 firmware announced an RPMsg service
Linux created a matching RPMsg device or driver binding
messages can cross the boundary
```

139.17 RPMsg echo test

If your BSP exposes a TTY-style RPMsg device:

```
# echo hello > /dev/ttyRPMSG0
# cat /dev/ttyRPMSG0
```

If your BSP exposes rpmsg char devices, use the matching test tool from the BSP or kernel samples.

Record:

```
RPMsg service name:
Linux device node:
message sent:
message received:
dmesg evidence:
```

Do not hide the mechanism behind “it works”. A future update will break this unless you know which service name and driver binding made it work.

139.18 The shared memory picture

RPMsg is built on shared memory, but you should think in messages.

The physical picture:

```
DDR or SRAM region visible to both processors
two vrings
shared buffers
mailbox notification
```

The software picture:

```
Linux endpoint sends message
M4 endpoint receives message
M4 endpoint replies
Linux endpoint receives reply
```

This prevents a bad habit:

```
two processors sharing arbitrary structs with no ownership rules
```

For a product, write a protocol:

```
message type
version
sequence number
payload length
checksum if needed
timeout behavior
reset behavior
```

139.19 Who owns the peripherals

STM32MP1 can route many peripherals to the A7 side or the M4 side.

Before firmware uses a peripheral, decide:

```
Linux owns it
or M4 owns it
or it is shared through a proper driver and protocol
```

Do not let Linux and M4 both program the same hardware block casually.

Make a table:

Peripheral	Owner	Reason	Linux node	M4 driver
UART for Linux console	Linux	debug access	enabled	disabled
Timer for control loop	M4	deterministic timing	disabled	enabled
Ethernet	Linux	network stack	enabled	disabled
Safety GPIO	M4	fast reaction	disabled	enabled
RPMsg shared memory	both	communication	reserved	OpenAMP

Device ownership is the embedded version of process isolation.

139.20 Watchdog policy

A mixed Linux plus RTOS product needs a watchdog policy.

The policy must define supervision and reset authority, including these cases:

```
which side kicks the hardware watchdog?
can Linux restart M4 firmware?
can M4 reset Linux?
what happens if RPMsg stops?
what happens if M4 firmware crashes?
what happens if Linux update installs incompatible firmware?
```

Common safe pattern:

```
M4 owns fast safety reaction
Linux owns update and logging
Linux supervises M4 heartbeat
```

```
M4 supervises critical output state
hardware watchdog resets the whole system only for unrecoverable faults
```

The exact answer depends on the product. The important part is to write it down.

139.21 Firmware versioning

Linux and M4 firmware are updated together or deliberately versioned apart.

Every RPMsg protocol should have:

```
protocol version
firmware version
minimum Linux driver version
feature flags
unknown message handling
```

First message after boot:

```
M4 -> Linux: hello, firmware version, protocol version, feature flags
Linux -> M4: accepted or rejected
```

If Linux and M4 disagree, fail clearly.

Do not let a field update produce a silent protocol mismatch.

139.22 Failure case: unavailable firmware

Remoteproc asks the Linux firmware loader for the selected image. A missing or inaccessible file can stop the load/start path before the M4 executes anything.

A firmware path is one dependency of startup, not proof that the image is compatible with the remote processor.

139.23 Failure case: incompatible target assumptions

Firmware built for another target can assume different memory, startup or peripherals. The result may be a load error, crash, silence or missing communication.

The board target is part of the firmware contract with the loader and hardware. A successful compiler invocation does not establish that contract.

139.24 Failure case: no matching RPMsg endpoint

The M4 can be running while its announced RPMsg service has no matching Linux driver or access path. Processor state and communication readiness are separate milestones.

Service naming, payload format and restart behavior belong to the protocol shared by both sides.

139.25 When Jailhouse is useful on STM32MP1

Jailhouse can still make sense on STM32MP1 when:

```
the workload needs A7 performance
the M4 is too small
you have a dual-A7 variant
static CPU and device partitioning is acceptable
the device ownership map is simple
you can dedicate one A7 core to the isolated workload
```

Example:

```
A7 core 0: Linux UI and networking
A7 core 1: isolated control Linux or RTOS cell
M4: low-power monitor or safety fallback
```

But do not choose this first.

First ask:

```
can the M4 do the real-time job?
can Linux plus PREEMPT_RT handle it?
can a normal Linux process boundary handle it?
```

If yes, Jailhouse may be unnecessary complexity.

139.26 Xen versus Jailhouse versus M4

Choose **M4 remoteproc** when:

```
the workload fits on Cortex-M4
you need MCU-style timing
message passing is enough
Linux can supervise firmware
```

Choose **Jailhouse** when:

```
you need static A-core partitioning
you can dedicate CPU and devices
```

```
guest lifecycle is simple
low overhead matters
```

Choose **Xen** when:

```
you need multiple Linux guests
you need richer guest lifecycle
you need Dom0 style management
you accept more virtualization complexity
```

Choose **plain Linux** when:

```
process isolation is enough
containers are enough
PREEMPT_RT is enough
the hardware ownership problem is simple
```

This decision is more important than tool preference.

139.27 Key Notes

- Remoteproc manages a remote processor’s firmware and lifecycle, RPMsg carries messages, and OpenAMP supplies software for those mechanisms. They are related but not interchangeable responsibilities.
- The STM32MP1 A7/M4 split supplies a physical companion core absent from the i.MX6ULL. It is an architectural comparison, not a MINI configuration.
- Firmware layout, reserved memory and peripheral ownership must agree between Linux and the M4. Shared hardware still needs explicit arbitration.
- A running M4 is not evidence of a working RPMsg channel. Naming, buffers and protocol compatibility must also agree.
- Watchdog and update policy should cover both processors and their shared interface. One healthy side must not hide the other’s loss of useful progress.
- A companion core can improve timing separation without automatically establishing security or safety isolation of the complete platform.

139.28 Troubleshooting

No `/sys/class/remoteproc`

Check:

```
kernel config
Device Tree
```

```
BSP support
remoteproc driver loaded
M4 node status
```

Firmware loads but M4 does nothing visible

Check:

```
firmware entry address
M4 clock and reset
console ownership
LED or GPIO ownership
Zephyr board target
```

RPMsg devices do not appear

Check:

```
resource table
shared memory reservation
mailbox setup
service name
Linux RPMsg driver
kernel log
```

Stop fails

Check whether the firmware cooperates with shutdown. Some remote processors or firmware images may not stop cleanly without board-specific support.

Everything works once but not after restart

Check reset state:

```
shared memory cleared
vrings reinitialized
peripherals reset
firmware does not assume power-on defaults
Linux driver handles remove and probe
```

139.29 Product checklist

Before calling a Linux plus M4 design production-ready, fill this out:

```
M4 firmware owner:
Linux supervisor owner:
firmware storage location:
firmware update mechanism:
rollback mechanism:
protocol version field:
heartbeat interval:
```

```
Linux action on missed heartbeat:
M4 action on Linux silence:
watchdog owner:
shared memory region:
peripheral ownership table:
crash log path:
field diagnostic command:
factory recovery path:
```

Blank lines are architecture debt.

139.30 Closing Part IX

Part IX started with hypervisor mode because ARM virtualization is real and useful.

But the deeper lesson is broader:

```
isolation is a design tool
```

Sometimes the right boundary is a hypervisor.

Sometimes it is an MMU process boundary.

Sometimes it is a Linux driver.

Sometimes it is an M4 companion core.

The grown-up embedded Linux skill is to match the boundary to the product, then make the boundary visible enough that another engineer can debug it at 2 a.m.

Embedded Linux on i.MX6ULL — From First Boot to First Driver

Architecture, Interfaces, and Design Decisions

Target board: Point Atom (正点原子) MINI, with revision-specific schematics: i.MX6ULL Cortex-A7 and a documented 512 MiB DDR3L core-board example. Check the fitted chip's speed grade before selecting clocks. Other boards require their own power, DDR, storage, pin, and peripheral configuration; a pin change alone is not a port. **Target reader:** Embedded engineer fluent in MCU / bare-metal / RTOS, new to Linux. **Host environment:** Native x86_64 Ubuntu 22.04 LTS is the Chapter 3 baseline. A VM needs working USB passthrough and a reachable board Ethernet connection. **Linux kernel target:** v6.6 LTS. Driver-API signatures (`class_create`, `i2c_driver.probe`, `i2c_driver.remove`) and DTS paths (`arch/arm/boot/dts/nxp/imx/`) in this book follow v6.6 conventions. **Philosophy:** Understand what each layer owns, the contracts between layers, and the consequences of a design choice. Code and commands make those mechanisms visible, not a sequence of assignments. Mainline sources, vendor documentation, and board schematics are reference material. Part VIII later covers toolchain construction and Yocto.

Scope

9 Parts, 178 chapters (131 numbered + 47 supplementary, "letter-suffix" convention). The numbered chapters provide the conceptual reading path. Supplementary chapters (letter-suffixed: `18A`, `18B`, `35A`, `52A`, etc.) explore particular design problems and can be read when relevant. Each chapter ends with Key Notes rather than an assigned lab. Examples are not claims of qualified hardware behavior. Parts IV–IX still need full technical and hardware review.

What the book covers

- **Part I — Foundations.** Host setup, the ARMv7-A architecture as it differs from Cortex-M, the i.MX6ULL SoC, the GNU toolchain, the Boot ROM's `IVT` / `DCD` / `BootData` contract, and a hardware bring-up checklist.
- **Part II — Bare-metal i.MX6ULL.** The mechanisms beneath a bootloader: reset entry, C runtime and linker layout, the ROM image contract, UART, clock and DDR initialization, exceptions, timers, MMU and caches. Peripheral examples connect register-level behavior to ownership, timing and data flow.
- **Part III — U-Boot, deeply.** A MINI-first vendor baseline and a separate mainline study tree show how startup, board support and boot policy fit together. GPIO, I2C, display, networking, board identity, rollback, recovery and power checks are design

studies. `bootcmd`, `bootargs`, FIT and development transports expose the contracts between ROM, bootloader and kernel. MINI hardware and optional modules remain distinct.

- **Part IV — The Kernel.** Build mainline Linux for i.MX6ULL, boot it from U-Boot, debug failed boots, describe a custom board with Device Tree, validate DT bindings, trace `start_kernel()` to PID 1, build an initramfs, turn it into a recovery system, manage product configs, and choose a kernel lifecycle.
- **Part V — Root filesystem & user space.** A `busybox`-based hand-built rootfs; `/proc` `/sys` `devtmpfs`; init systems; libc and dynamic linking; Buildroot; bootable SD/eMMC and SPI NOR images; product board directories; Ubuntu-base as a fully-featured alternative; services, networking, time, logging, security, read-only root + overlayfs, and containers on embedded.
- **Part VI — Driver development.** ~33 chapters covering every common subsystem from char devices and platform drivers through I²C/SPI/PWM/RTC/IIO/regmap/DMA/Net/Sound/DRM/USB, with deeper treatment of CAN, multi-touch, block devices, WIFI, cellular modems, HDMI bridges, kernel timers, async notification, watchdog, power management, PREEMPT_RT real-time, MTD/UBI, V4L2/GStreamer, and a Rust-for-Linux sidebar. **One canonical example per subsystem** — depth on real chips lives in Part VII.
- **Part VII — Device cookbook.** 54 chapters compare device choices through electrical interfaces, Linux subsystem support, data flow, timing and failure limits. Schematics, Device Tree and code examples illustrate those choices. Storage, sensors, converters, displays, cameras, audio, wireless links, industrial buses and actuators can be read as independent references.
- **Part VIII — Debug, production, advanced.** Observability across layers, kernel and userspace debugging, board-port design contracts, build inputs, Yocto layer development, secure boot, field updates, upstream maintenance, CI evidence and BSP migration.
- **Part IX — Applied virtualization and mixed-criticality systems.** QEMU as a virtual hardware model, ARM HYP mode, Xen guest boundaries, device ownership, DMA and isolation limits, Jailhouse partitioning, and STM32MP1's Linux/RTOS A7/M4 split. Virtual examples explain architecture, not MINI electrical or timing qualification.

How to read this book

Start with Chapter 1. Follow the problem, then the mechanism and its design consequences. A register listing or command can be read as evidence of an interface without reproducing it. Chapters use **Key Notes** to collect their important architectural decisions and limits, alongside pitfalls and primary references. They do not require a repeated section template or completed exercises.

PART I — FOUNDATIONS

> *You are an MCU engineer. You know what a vector table is, what a linker script is, what `volatile` is for. This part exists to give you names for the things Linux adds on top.*

Chapter 1 — Preface and how to use this book

- Who this book is for (the MCU/bare-metal engineer)
- What we mean by “raw” and where integrated build systems fit later
- Conceptual reading routes, optional references, and the limits of examples
- How chapters depend on each other (dependency graph)
- Conventions: prompt symbols, register notation, file paths
- **Key Notes: Layers have owners.** ROM, bootloader, kernel and application make different decisions. A failure becomes easier to reason about when the responsible layer is identified.

Chapter 2 — What “Embedded Linux” actually is

- The four layers: Boot ROM → Bootloader → Kernel → User space
- Kernel space vs user space; the syscall boundary
- Process, thread, file descriptor — vocabulary first
- How this differs from your RTOS: virtual memory, demand paging, ELF loading
- Why Linux is “big” (and where its size actually lives)
- **Focus:** the *user/kernel split* is the single most important concept in the whole book
- **Key Notes:** Boot is a sequence of ownership transfers: the ROM selects and loads an image, the bootloader prepares Linux’s inputs, and the kernel starts user space. Ordinary application requests do not return through the bootloader.

Chapter 3 — Host environment setup

- Choosing a host OS (Ubuntu 22.04 LTS); why native Linux beats WSL/VM for this work
- Essential build/inspection packages first; TFTP/NFS services are a separate later-preparation track
- Installing two project-local Arm GNU toolchains: `arm-none-linux-gnueabihf-` for Linux and `arm-none-eabi-` for bare metal

- A short, explicitly sourced environment script; no global compiler installation or shell-startup changes
- Manual TFTP/NFS configuration with scoped access and undo instructions; explicit privileged serial/USB access
- Project-local NXP `uuu` (Universal Update Utility), pinned to a named release
- A reproducible workspace layout for the rest of the book
- **Key Notes:** Compiler installation and compiler selection are separate state. `. ~/imx6ull/scripts/env.sh` changes the current Bash shell and its future children, not another already-open terminal. Running the file in a child shell cannot change its parent.

Chapter 4 — ARMv7-A and the Cortex-A7, for the MCU engineer

- ARMv7-A vs ARMv7-M: what Cortex-A adds (MMU, privilege levels, generic timer, NEON, multicore option)
- Exception model: USR / SYS / SVC / IRQ / FIQ / ABT / UND (and how this maps to Linux's user/kernel split)
- Banked registers, the program status register, mode switching
- Generic-timer architecture versus the actual timer enabled by the selected Linux configuration
- Cache hierarchy: L1 I/D, integrated L2 inside the Cortex-A7 MPCore (128 KB on i.MX6ULL — no external PL310), inner/outer shareable, MESI
- MMU concepts: virtual address, page table walk, TLB, ASIDs, domains
- NEON / VFP overview
- **Focus:** MMU + privilege modes on this Cortex-A7 Linux path, with worked exception/register examples
- **Key Notes:** A Cortex-A7 exception selects banked SP/LR and saves status, but does not push a Cortex-M-style register frame. Shared registers, usable mode stacks, and return-state preservation remain software responsibilities.

Chapter 5 — A tour of the i.MX6ULL SoC

- Block diagram and what each block does
- Memory map: OCRAM, ROM, DDR aperture, peripheral regions
- Clock tree at 30,000 ft: oscillators → PLLs → CCM → root clocks → peripheral gates
- Power domains and the PMU

- IOMUX: the universal-multiplexer pattern (and how `iomuxc.h` is generated)
- Boot fuses (eFuses) and BOOT_MODE pins
- The supplied 4,127-page reference manual: a verified chapter catalog and worked UART1 lookup
- **Key Notes:** A usable peripheral path combines a controller address, a live clock, the selected pad function, electrical settings, and board routing. Correctness at one point does not establish the rest of that path.

Chapter 6 — The toolchain

- `gcc` is *not* one tool: cpp, cc1, as, collect2, ld
- `binutils`: `as`, `ld`, `objcopy`, `objdump`, `nm`, `readelf`, `strip`, `ar`, `addr2line`
- The C library: glibc vs musl vs uClibc-ng vs newlib; why our first bare-metal builds use no C library
- ABI: EABI vs hard-float vs soft-float; `arm-none-linux-gnueabihf` decoded
- ELF format: program headers vs section headers; what the loader actually reads
- Linker scripts: `MEMORY`, `SECTIONS`, `VMA` vs `LMA`, `ENTRY`, `KEEP`
- Make basics that matter: implicit rules, pattern rules, automatic variables, `.PHONY`, recursive vs non-recursive
- **Key Notes:** The GCC driver coordinates preprocessing, compilation, assembly, and linking. A valid object file does not establish that symbol resolution or final placement succeeded, and an old output file can survive a failed rebuild.

Chapter 7 — The Boot ROM, IVT, DCD, and BootData

- What the Boot ROM does on power-on (in order, with addresses)
- Reading `BOOT_MODE[1:0]` and `BOOT_CFG` pins
- Boot device options: SD/eMMC, NAND, SPI-NOR, USB-SDP, EIM-NOR
- The **IVT (Image Vector Table)**: layout, every field decoded
- The **DCD (Device Configuration Data)**: a tiny scripting language the ROM executes to bring up DDR and clocks *before your code runs*
- The **BootData** structure: load address and image length
- USB-SDP (Serial Download Protocol): command IDs and the later `uuu` transfer path
- HAB (High Assurance Boot) introduction (deep-dived in Ch. 124)

- **Focus:** distinguish image placement, linked addresses, DCD initialization, and the alternative SPL-first path
- **Key Notes:** The Boot ROM is a loader with a fixed contract. The IVT names the entry point and associated structures, BootData describes the loaded extent, and DCD supplies initialization before application code can run.

Chapter 8 — Hardware bring-up checklist

- Unboxing the Point Atom board: physical inspection, jumpers, SD slot, OTG cable
- Establish the exact input rating, polarity, selector/jumper settings, and USB power interactions from the matching supplier guide before applying power
- Unpowered short screening is not proof of healthy rails; powered measurements need documented test points and limits
- UART1 wiring (TXD/RXD/GND), correct voltage levels, 115200 8N1
- Identify the removable card and record its identity; re-identify it before every later write
- Optional: JTAG header pinout, OpenOCD interface adapter (FT2232H, J-Link)
- Revision-specific USB boot selector and ROM SDP enumeration; transfer/execution are later milestones
- **Key Notes:** Bring-up evidence belongs to a particular board revision, fitted core and power arrangement. A matching product name is not enough to transfer another board’s connector, voltage or selector assumptions.

PART II — BARE-METAL i.MX6ULL

> *Before Linux can manage a board, something must establish memory, clocks and a way to report errors. These chapters show those responsibilities directly, in OCRAM and then DDR, so they remain recognizable inside a bootloader.*

Chapter 9 — First LED, pure assembly

- Reset vector and where the Boot ROM jumps to
- Setting up the SVC-mode stack in OCRAM
- Enabling the GPIO clock via CCM_CCGR registers
- IOMUX configuration for the LED pin (Point Atom schematic pin)

- Direct GPIO data register write
- Infinite loop (no return, no exit)
- Building with `as`, linking with `ld`, image with `objcopy -O binary`
- **Key Notes:** The ROM owns loading and the initial handoff. The assembly payload owns the GPIO configuration after entry, with no C runtime, scheduler or DDR dependency.

Chapter 10 — C + startup.S + linker script

- Why pure C cannot run yet: who sets the stack pointer, who clears `.bss`, who copies `.data`?
- `startup.S`: stack init, `.bss` zero, `.data` copy from LMA to VMA, branch to `main`
- A complete linker script: `MEMORY` block, `SECTIONS`, `_etext` / `_sdata` / `_edata` / `_sbss` / `_ebss` symbols
- The `.init`, `.text`, `.rodata`, `.data`, `.bss` regions and why each exists
- `__attribute__((section(...)))` and when to reach for it
- Makefile that compiles `.S` and `.c` together, links, makes raw binary
- **Key Notes:** The linker describes placement and exports boundaries. Startup establishes the aligned stack and object contents before calling C. Neither the compiler nor the IVT supplies that runtime preparation.

Chapter 11 — Hand-building a Boot ROM-acceptable image

- Recap of Ch. 7: what the Boot ROM expects to find at offset 0x400 of the boot media
- Writing a `.cfg` file for `mkimage -T imximage` — *but first*, do it manually so we know what the tool generates
- Walking the IVT byte-by-byte in a hex editor
- Writing a DCD script that initializes SDRAM controller for DDR3 (we'll use NXP's tested values for now; we derive them ourselves in Ch. 14)
- Padding and offsets for the SD card layout: where the IVT lives, where the image lives
- File, media and RAM offsets are separate parts of the ROM image contract
- **Pitfall:** off-by-one in the IVT `self` pointer is the #1 reason boards "do nothing"

- **Key Notes:** File offsets, media offsets and RAM addresses are different coordinates. In this primary SD layout, writing the already-padded file at media byte zero places the IVT at `0x400`. An extra seek would shift it out of place.

Chapter 12 — UART driver and `printf`

- UART1 register map: USR1, URXD, UTXD, UCR1–4, UFCR, UBIR, UBMR
- Computing the baud divisor from the module's input clock
- Polling-mode TX/RX
- Implementing a minimal `_putchar()` so we can hook up `printf`
- A 200-line `printf` clone (or use a tiny third-party one like `mpaland/printf`) — we explicitly avoid pulling in newlib
- **Key Notes:** UART1 transport and number formatting are separate interfaces. The integrated USB-TTL bridge carries console bytes, while USB-OTG carries ROM downloader commands.

Chapter 13 — CCM clock tree bring-up

- The clock tree, drawn end-to-end for i.MX6ULL: 24 MHz XTAL → ARM PLL / System PLL → root clocks → peripheral gates
- CCM, CCM_ANALOG, and PMU register groups
- Auditing selected clock paths without changing live operating points
- AHB and IPG bus clocks
- The CCGR gating registers: each peripheral has 2 bits, what they mean
- Side topic: how to compute power for a given clock configuration
- **Key Notes:** A frequency belongs to a complete selected path: reference, PLL or bypass, PFD where used, muxes and dividers. CCGR gates control availability, not rate.

Chapter 14 — DDR3 initialization with MMDC

- DDR3 fundamentals refresher: banks, ranks, rows, columns, CL/tRCD/tRP/tRAS
- Reading your specific DDR3 chip's datasheet (Point Atom schematic → part number)
- The MMDC (Multi-Mode DDR Controller) register groups: MDCTL, MDPDC, MDOTC, MDCFG0/1/2, MDMISC, MAARCR, MAPSR, MPPDCMPR1/2, MPWLDECTRL0/1, MPDGCTRL0/1, MPRDDLCTL, MPWRDLCTL, MPMUR0…

- The DDR3 initialization sequence per JEDEC: precharge all → MR2 → MR3 → MR1 → MR0 → ZQCAL
- Calibration: write leveling, DQS gating, read/write delay calibration
- Using NXP’s DDR Stress Tool to verify your settings before trusting them
- Translating verified register values into a DCD script
- **Focus:** matching initialization assumptions to fitted memory and its qualification limits
- **Key Notes:** DDR availability depends on a matched clock, geometry, electrical setup, timing/command sequence, refresh and calibration. A valid address or a copied delay table establishes none of that configuration by itself.

Chapter 15 — Exceptions and the GIC

- The ARMv7-A exception vector table (8 entries) and its software responsibilities
- High vs low vectors (`VBAR` register)
- The GIC v2 (Generic Interrupt Controller): Distributor + CPU Interface, register maps
- Configuring an SPI (Shared Peripheral Interrupt) and an SGI (Software-Generated Interrupt)
- Top-half ISR pattern in bare-metal: save context, ack the GIC, call C handler, return from exception
- **Key Notes:** Cortex-A7 banks IRQ SP/LR and saves CPSR, but software owns shared-register preservation, stack alignment and exception return. A C handler alone is not an exception-entry interface.

Chapter 16 — Timers (EPIT and GPT)

- EPIT (Enhanced Periodic Interrupt Timer): a 1 ms tick
- GPT (General Purpose Timer): free-running counter for delay/timestamp
- Building `udelay()` and `mdelay()` primitives
- Using GPT as a profiling tool: cycle-count any function
- **Key Notes:** GPT’s free-running count represents elapsed ticks. EPIT requests periodic service. Their clocksource/clockevent roles separate time measurement from scheduling, without prescribing Linux’s eventual timer selection.

Chapter 17 — MMU and caches

- Why we want the MMU on, even in “bare-metal”: cache control

- Translation regimes: TTBR0/TTBR1, 32-bit short-descriptor format
- Building a first-level page table that maps SoC peripherals as Device memory and DRAM as Normal Cacheable
- Domains and access permissions
- Enabling the I-cache and D-cache (`SCTLR` bits)
- Cache maintenance: invalidate, clean, clean+invalidate by VA, set/way
- Measuring the performance difference with the GPT
- **Focus:** you now possess *every* primitive Linux needs from firmware. Carry this fact through Part III.
- **Key Notes:** Identity mapping preserves pointer values but changes access checks and memory behavior. Device/XN register mappings and Normal RAM mappings are policy choices, not consequences of `volatile`.

Chapter 18 — Optional bare-metal peripherals

- I²C (I2C1) to an EEPROM on the board
- SPI to an external flash
- LCD via eLCDIF — bring up the framebuffer manually, draw a pattern
- Why we stop here and move to U-Boot
- **Key Notes:** The controller’s presence does not establish an attached device. EEPROM address width, flash command set and panel timing belong to the actual populated part and routed board, not a generic i.MX6ULL peripheral name.

Chapter 18A — Supplementary: Project organization the “STM32-style”

Once a bare-metal program crosses ~500 lines, the single-file layout we used through Ch 14 stops scaling. This chapter refactors our work-so-far into proper header/source separation, register-definition macros in a single `imx6ull.h`, BSP folder layout (`bsp/clk/`, `bsp/gpio/`, `bsp/uart/`, …), and a top-level `Makefile` that builds the BSP and links it against `main.c`. Also discusses the NXP SDK’s `MCIMX6Y2.h` struct-based register approach as an alternative — when to adopt it, when to stay hand-rolled.

- **Focus:** the moment you have ≥2 peripherals, organization saves more time than it costs
- **Key Notes:** SoC offsets, board wiring and driver behavior describe different ownership domains. A shared hardware header prevents duplicate definitions without making every board’s routes interchangeable.

Chapter 18B — Supplementary: Button input and beep (active buzzer)

A polled GPIO input example separates electrical input behavior from software debounce and logical key events. The fitted active buzzer is an on/off output. A passive piezo requiring PWM is a different device, not an interchangeable implementation.

- **Key Notes:** A pad level and an application event are different interfaces. PSR reports the input now. The history filter reports one qualified press or release rather than repeating while a level is held.

Chapter 18C — Supplementary: Bare-metal RTC (SNVS)

The SNVS (Secure Non-Volatile Storage) is the only always-on domain on the chip; its 32-bit second counter survives main-power-off. We initialize it, set the time, sleep the rest of the SoC, wake it, read it back. Then a small sidebar on how this maps to Linux's `struct rtc_class_ops` in Chapter 48.

- **Key Notes:** The LP counter's 47 bits represent 32.768 kHz ticks, including 15 fractional bits. Whole seconds still need a software-chosen epoch and calendar conversion. Zero is not a valid wall-clock setting by itself.

PART III — U-BOOT, DEEPLY

> *The register-level work from Part II gives us a way to read U-Boot's source. We will trace its framework, distinguish a build from board qualification, and test each boot decision before trusting it.*

Chapter 19 — U-Boot from source, first boot

- Pin Point Atom's public tutorial source for the documented 512 MiB DDR / 8 GB eMMC MINI core
- Inspect its board files, make the initial configuration visible, and use Chapter 3's existing portable compiler
- Build the MINI vendor target and inspect its no-SPL ROM+DCD image
- Separate file offsets, SD placement, runtime addresses and the reserved environment range
- Inspect commands without sweeping live memory or saving experimental policy
- **Key Notes:** The pinned vendor target describes the documented 512 MiB eMMC core. Shared ALPHA/MINI software does not make every carrier connection or PHY revision interchangeable.

Chapter 20 — U-Boot SPL: the missing link

- Compare the MINI’s ROM+DCD route with ROM+SPL and identify who initializes DDR
- Introduce a separate pinned upstream v2026.04 study tree without replacing the vendor MINI image
- Build a separate PICO i.MX6UL SPL reference for source study, not MINI flashing
- Trace OCRAM startup, DDR-resident BSS, size limits and the board’s handoff path
- Distinguish the SPL wrapper, full payload format and raw MMC load sector
- **Key Notes:** DCD and SPL solve the same early-memory dependency with different owners: ROM interprets a table, while SPL executes board-specific initialization code.

Chapter 21 — U-Boot internals

- Follow ARM entry, current initcall machinery, `board_init_f`, relocation and `board_init_r`
- Explain early stack/global data, selected relocation fixes and the BSS overlay
- Compile and register a complete `book_hello` command
- Separate compiled defaults, RAM environment changes and persistent backend selection
- Distinguish driver binding from probing and U-Boot’s control DT from the kernel DT
- **Key Notes:** The ARM full-U-Boot path divides work between C planning and assembly handoff. `board_init_f` records reservations in GD, then `crt0.S` changes stack/GD and relocates execution.

Chapter 22 — Porting U-Boot to the MINI

- Build a concrete `mx6ull_pa_mini` modern migration from documented MINI/core wiring
- Connect board Kconfig, defconfig, control DTS, header and ROM DCD
- Supply a MINI control tree without the EVK carrier include, and verify the ordered vendor DCD
- Integrate UART1, TF, eight-bit eMMC reset and single ENET2 with a datasheet-matched PHY candidate
- **Key Notes:** A board port connects build selection, control DT, image configuration and board hooks. A matching SoC target alone does not describe the MINI carrier or populated core.

Chapter 23 — `bootcmd`, `bootargs`, FIT images

- Gate storage selection, root identification, every required load and argument construction
- Explain `bootm`, `bootz`, raw initramfs sizes and separate image buffers
- Trace `/chosen/bootargs`, kernel console/root prerequisites and root versus init failure
- Package a complete script and a kernel+DTB hash-only FIT
- Separate hashes, trusted required signatures and the boot-chain policy
- **Key Notes:** `bootcmd` is U-Boot execution policy, while `bootargs` becomes a candidate Linux command line through the OS FDT. Helpers and kernel configuration can alter what `/proc/cmdline` ultimately reports.

Chapter 24 — Workflows: TFTP, NFS, USB-OTG

- Stage matched artifacts as real files under the established TFTP server root
- Keep U-Boot’s network loader separate from Linux’s Ethernet/IP/NFS-root path
- Explain built-in root-network drivers, export permissions and the observed command line
- Separate userspace and kernel rebuild/restart loops
- **Key Notes:** TFTP loads boot payloads into RAM, NFS backs Linux’s root filesystem, and ROM USB download enters an earlier recovery stage. They solve different failures and have different prerequisites.

Optional deep dive

Chapter 24A — Supplementary: Building an i.MX6ULL teaching port from scratch

- Reuse generic ARMv7 startup and U-Boot frameworks under an independent teaching architecture
- Show complete UART1, clock, GPT1 and USDHC2 PIO driver files and manual integration edits
- Trace bounded polling, baud/clock calculations, reset behavior and card error recovery
- Include the full pinned vendor DDR sequence for the documented 512 MiB eMMC core

- Explain IVT/Boot Data fields, file/media/RAM offsets and evidence checkpoints
- **Key Notes:** The port joins several contracts: ROM image layout, early ARM runtime, U-Boot relocation, Device Tree discovery and peripheral framework callbacks. A correct interface at one stage does not establish the next.

Applied U-Boot scenarios

Chapter 24B - Supplementary: U-Boot board policy with GPIO and I2C

- Add late policy to the existing port without duplicating its hook
- Use MINI’s stock KEY0 as the first policy input; keep TMP102 as an optional I2C1 add-on
- Claim the GPIO1_IO18 descriptor, handle logical polarity and debounce, then release ownership
- Keep failure closed and dispatch exactly one normal or recovery path
- **Key Notes:** `board_late_init()` runs after environment import but before normal dispatch. A blocked-first hook prevents stale saved commands from opening the gate before required inputs are valid.

Chapter 24C - Supplementary: Ethernet fallback boot in U-Boot

- Use MINI’s single ENET2 route and distinguish older LAN8720 from V2.2 SR8201F support
- Review reset/clock ownership, driver matching and uniquely allocated MAC addresses
- Apply explicit static network state and test DHCP separately
- Gate fresh kernel/DTB loads and arguments on both local and TFTP paths
- Keep fallback before kernel handoff and distinguish a local rootfs from network payloads
- **Key Notes:** Local-first fallback responds to load failures or a boot command returning before handoff. It cannot diagnose a later Linux failure or recover a bootloader that never starts.

Chapter 24D - Supplementary: Board identity and variant selection in U-Boot

- Define a bounded versioned 64-byte record for an optional external EEPROM, absent from stock MINI

- Validate complete fields, MAC/serial values and supported hardware combinations
- Publish readiness only after checked selection and identity assignments
- Keep unknown identity closed rather than inventing a universally safe DTB
- **Key Notes:** The 64-byte record is a proposed optional-EEPROM interface, not a stock MINI factory format. Fixed hardware can select its matching OS tree explicitly without invented identity data.

Chapter 24E — Supplementary: Multi-variant FIT images and DT overlays at runtime

- Package fixed-core no-display, ID4342 and ID7016 configurations with matching OS DTBs
- Stage the FIT separately from its kernel destination and name the selected configuration
- Use a complete harmless host overlay fixture and reload both inputs after apply failure
- Distinguish the working OS tree from U-Boot’s control tree
- **Key Notes:** FIT configurations share payloads through named references. One bundle simplifies release bookkeeping but does not reduce the set of hardware combinations that need qualification or make updates atomic.

Chapter 24F - Supplementary: Watchdog, bootcount, and rollback in U-Boot

- Require a separately qualified persistent backend and layout beyond the RAM-only scaffold
- Count candidate attempts and roll back to the recorded good slot
- Gate complete slot loads, root PARTUUID and grouped metadata changes
- Let Linux verify the actual running root, kernel, slot and health before acceptance
- Explain ignored save failures, metadata durability and watchdog-handoff limits
- **Key Notes:** Bootcount measures attempts, not diagnosed kernel failures. With an initial zero and limit three, counts one through three use the candidate and count four selects the alternate command.

Chapter 24G - Supplementary: Factory and recovery modes in U-Boot

- Distinguish ROM SDP, working-U-Boot services and Linux recovery dependencies
- Distinguish MINI USB_TTL, USB_OTG1 service and USB_OTG2 host connectors

- Review modern controller support and authorized device identity
- Expose only a reviewed disposable target through UMS or DFU
- Require fresh write authorization and preserve exclusive host/target storage ownership
- **Key Notes:** ROM SDP, U-Boot USB service and Linux recovery require progressively more working software. A U-Boot gadget cannot repair an image that prevents U-Boot from starting.

Chapter 24H - Supplementary: PMIC and power policy in U-Boot

- Trace MINI’s discrete regulators, reset supervision and SNVS backup supply
- Separate status/enable requests from coherent measured rail/energy evidence
- Keep an optional monitor’s refusal adapter out of stock MINI boot until its hardware is implemented
- Prioritize power veto over normal, alternate and writable recovery paths
- **Key Notes:** The documented MINI/core uses discrete regulators with no fitted I2C PMIC or rail monitor. `PMIC_ON_REQ` is a sequencing net, and the CR1220 backs SNVS rather than powering updates.

Chapter 24I - Supplementary: U-Boot display and boot screen

- Review the actual video/BMP symbols, LCDIF driver parsing and board header requirement
- Use MINI J1 RGB888 wiring and preserve panel-ID/boot-strap isolation
- Qualify the added panel’s timing, power and GPIO1_IO08 backlight separately
- Keep the bounded BMP file buffer distinct from scanout framebuffer and other RAM users
- Make optional splash failure unable to bypass recovery, rollback or power policy
- **Key Notes:** BMP source data, framebuffer pixels and RGB wires are separate formats. The MXS 24/18-bit output paths use 32-bit framebuffer storage, independent of a 24-bit BMP decoder.

Chapter 24J - Supplementary: SPI LCD and I2C OLED displays in U-Boot

- Distinguish RGB scanout, SPI display RAM and I2C OLED page data
- Do not infer a U-Boot driver from a Linux compatible string

- Implement bounded SPI transport with explicit D/C, CS, byte order and cleanup
- Use a qualified SSD1306 profile, zero offset length and correct command/data control bytes
- **Key Notes:** The stock MINI has neither serial display, and v2026.04 lacks the named panel drivers used in these examples. A compatible string or working bus driver does not supply reset, initialization or rendering.

PART IV — THE KERNEL

> *The kernel is large but knowable. Its boot interfaces, hardware descriptions and configuration form separate parts of the design.*

Chapter 25 — Building mainline Linux for i.MX6ULL

- Clone the stable tree at the book's pinned tag, `v6.6`
- `make ARCH=arm imx_v6_v7_defconfig`
- The build artifacts: `vmlinux`, `Image`, `zImage`, `arch/arm/boot/dts/nxp/imx/*.dtb`
- `vmlinux` vs `zImage`: who decompresses, when, where
- Modules: `make modules && make modules_install INSTALL_MOD_PATH=...`
- **Key Notes:** A kernel build is defined by its source revision, resolved `.config` and toolchain. An output directory separates generated state from source, but does not make different configurations interchangeable.

Chapter 26 — Booting the kernel from U-Boot

- Loading via TFTP into RAM with `&&`, so failed downloads stop the boot
- `bootz 0x82000000 - 0x83000000`
- The first 30 lines of kernel boot log — every line decoded
- "Uncompressing Linux... done, booting the kernel." — *where* in the kernel source this happens (`arch/arm/boot/compressed/head.S`)
- **Key Notes:** U-Boot owns image loading and the CPU handoff state. Linux then owns its memory mappings, interrupt handling and driver initialization.

Chapter 26A — Supplementary: Kernel boot failure playbook

- Keep one known-good `bootz` command and one known-good `bootargs`

- Classify failures: no output, early output then silence, VFS panic, init panic, or driver probe failure
- Use `earlycon`, `loglevel=8`, `ignore_loglevel`, and `initcall_debug`
- Debug wrong DTB, wrong console, bad rootfs, missing init, and NFS root hangs
- **Key Notes:** A boot failure has a stage: image handoff, console availability, root mounting, init execution or device probing. Similar visible symptoms can originate in different stages.

Chapter 27 — Device Tree: the contract between firmware and kernel

- What problem DT solves (no more board-files, no more #ifdefs)
- DTS syntax: nodes, properties, phandles, labels, references
- `.dtsi` vs `.dts`, includes, overlays
- `compatible` strings — the single field that determines which driver binds
- `reg`, `interrupts`, `clocks`, `pinctrl-0`, `status` — the universal properties
- Walking `imx6ull.dtsi` → `imx6ull-14x14-evk.dts` end-to-end
- `dtc` and the `dtbs_check` flow against YAML bindings
- Writing your first overlay to add a new I²C device
- **Focus:** for an MCU engineer this is the largest mental shift. Spend extra time here.
- **Key Notes:** The SoC `.dtsi` describes controllers and shared resources. The board `.dts` describes connections, supplies and enabled devices, so a shared SoC does not make two boards' DTBs interchangeable.

Chapter 27A — Supplementary: DT bindings YAML + `dt_binding_check`

A 2018+ mainline-hygiene requirement. Since kernel v4.18 every new device-tree binding must ship a YAML schema and pass `make dt_binding_check`. Without it, your patch will not be accepted upstream. Without it, your binding can drift silently between board variants and you will not know.

- Why JSON-Schema for DT bindings (vs the old `.txt` files)
- A binding for a custom node, written from scratch, validated
- `make dt_binding_check` and `make dtbs_check` — what each does
- Common errors and how to read them

- **Key Notes:** A binding is the contract between hardware description and driver interpretation. YAML expresses that contract using JSON Schema constraints rather than replacing it with implementation details.

Chapter 27B — Supplementary: Device Tree for a custom i.MX6ULL board

- Start from `imx6ull.dtsi` and write the board `.dts`
- Add memory, `/chosen`, aliases, UART console, regulators, SD/eMMC, Ethernet, I2C, SPI, LEDs, and keys
- Update the kernel DTS Makefile so the board DTB is built
- Bring up one peripheral at a time and check each driver from the boot log
- Use `dtbs_check` to catch binding mistakes before hardware testing
- **Key Notes:** A board DTS owns external topology, not the SoC register description or driver implementation. Included SoC nodes provide the base that board-specific references refine.

Chapter 28 — Kernel startup, traced

- `start_kernel()` — read it function-by-function
- `setup_arch()`, `setup_machine_fdt()`, memblock, paging_init, mm_init
- `rest_init()` → `kernel_init` thread → `run_init_process("/sbin/init")`
- `kthreadd`, the idle thread, init's pid 1
- Where `printk` ring buffer lives; how early-boot printk works before serial drivers exist
- **Key Notes:** Architecture entry establishes CPU recognition and the initial address mappings before ordinary kernel C services exist. Failures there may precede a usable console.

Chapter 29 — Initramfs from scratch

- What an initramfs *is* (cpio archive, not a filesystem image)
- Building a one-binary initramfs: a single statically-linked program that prints "hello" and `reboot()`s
- Then a BusyBox initramfs with a real shell
- Embedding the initramfs in the kernel image vs loading it separately
- The handoff: kernel mounts initramfs as `/`, runs `/init`

- **Key Notes:** An initramfs supplies files before the normal storage-backed root is available. The kernel unpacks a cpio archive, then early user space owns the next boot or recovery decision.

Chapter 29A — Supplementary: Initramfs as a recovery system

- Turn the Chapter 29 BusyBox initramfs into a small recovery environment
- Add a recovery `/init` with shell, read-only rootfs mount, log collection, and reset/update placeholders
- Boot a raw `initramfs.cpio.gz` with `bootz kernel initrd_addr:initrd_size dtb`
- Add safe update rules: checksum, target confirmation, unmounted partition, and factory-data protection
- Connect recovery selection to U-Boot bootmode, button, bootcount, or failed normal boot
- **Key Notes:** Recovery independence depends on the whole boot path: firmware must still load a working kernel, board DTB and archive. Running from RAM avoids reliance on the main rootfs, not every storage or hardware dependency.

Chapter 30 — Kernel configuration deep-dive

- `make menuconfig` — but reading the `.config`, not clicking blindly
- The big knobs: `CONFIG_PREEMPT*`, `CONFIG_HZ`, `CONFIG_TICK_ONESHOT`, `CONFIG_NO_HZ`, `CONFIG_HIGH_RES_TIMERS`
- Tracing, debug, lockdep options
- Module vs built-in: when does it matter?
- Generating your own custom `defconfig` and saving it under `arch/arm/configs/`
- **Key Notes:** Kconfig resolves requested options against dependencies and defaults. A hand-edited symbol does not necessarily survive resolution or become compiled code.

Chapter 30A — Supplementary: Kernel lifecycle — mainline, stable, LTS, vendor BSPs

The decision framework most readers never see laid out explicitly. Six release tracks, each with different stability/feature/security guarantees:

- **Mainline** (Linus’s tree, ~weekly rc, 9-week cycle)
- **Stable** (Greg KH’s tree, fixes only, ~1 month lifetime per minor)
- **Long-Term Support (LTS)** (selected mainline releases get fixes for 2 or 6 years)

- **Vendor BSP** (NXP, ST, TI; usually a frozen mainline + thousands of patches)
- **Yocto/Buildroot-curated** (a vendor BSP plus a layer or two of patches, retargeted)
- **Distribution kernels** (Debian, Ubuntu, Fedora; not for embedded targets in general)
- When each is right: dev, shipping a product, shipping into critical/long-life environments
- The **migration cost** of a vendor BSP — pinned forever to that minor version's API
- How to read a kernel release announcement and decide what it means for you
- **Why old kernel forks are a trap**: a 2017-era kernel has missed eight years of security fixes
- **Key Notes:** A release tag fixes source identity, while a maintained branch supplies an ongoing stream of changes. A teaching baseline and a supported shipping baseline serve different purposes.

Chapter 30B — Supplementary: Product kernel configs

- Split kernel configuration into base, development, production, and recovery fragments
- Use `scripts/kconfig/merge_config.sh`, separate `O=` build directories, and `olddefconfig`
- Decide which drivers must be built in and which can be modules
- Use `scripts/diffconfig` and `savedefconfig` before release
- Set `CONFIG_LOCALVERSION` so the board reports the exact kernel flavor
- **Key Notes:** Development, production and recovery kernels share hardware needs but serve different operational roles. A diagnostic interface useful in development is not automatically a production requirement.

PART V — ROOT FILESYSTEM & USER SPACE

Chapter 31 — A root filesystem, by hand

- The FHS (Filesystem Hierarchy Standard) cheat sheet
- Building BusyBox from source, statically linked
- Creating `/bin`, `/sbin`, `/etc`, `/dev`, `/proc`, `/sys`, `/tmp`, `/var`, `/root`, `/lib`
- A minimal `/etc/inittab`, `/etc/init.d/rcS`, `/etc/fstab`, `/etc/passwd`, `/etc/group`, `/etc/profile`

- Exporting via NFS, mounting from kernel cmdline `root=/dev/nfs`
- **Key Notes:** A rootfs supplies executables, libraries and configuration. Directories such as `/proc`, `/sys` and `/dev` are mount points whose runtime contents are not produced by copying ordinary files.

Chapter 32 — /proc, /sys, devtmpfs

- `/proc` — the original process FS, now also a kernel-info FS
- Useful files: `/proc/cpuinfo`, `/proc/meminfo`, `/proc/interrupts`, `/proc/iomem`, `/proc/devices`, `/proc/<pid>/maps`, `/proc/<pid>/status`
- `/sys` — the modern device model surface
- `/sys/class/gpio/`, `/sys/bus/i2c/devices/`, `/sys/devices/platform/`
- `devtmpfs` vs static `/dev` vs `mdev` vs `udev`
- **Key Notes:** procfs exposes process and system state, sysfs exposes the device model, and devtmpfs supplies device nodes. Their common file-shaped interface does not give them the same purpose.

Chapter 33 — Init systems

- BusyBox `init`: minimal, inittab-based
- `sysvinit` / OpenRC: the classical world
- `systemd`: services, sockets, targets, journals — a brief sober tour
- Why an embedded target may not want any of these
- **Key Notes:** PID 1 has system-lifetime responsibilities, including child reaping and shutdown coordination. Running an application directly as init transfers those responsibilities to the application rather than removing them.

Chapter 34 — libc, dynamic linking, and the loader

- glibc vs musl vs uClibc-ng (size, license, compatibility)
- ELF dynamic linking: PLT, GOT, `LD_LIBRARY_PATH`, `RPATH`
- `ldd`, `readelf -d`, `LD_DEBUG=files`
- `/lib/ld-linux-armhf.so.3` — what it actually does
- Static linking, when to use it on embedded

- **Key Notes:** A dynamically linked ELF declares an interpreter path and needed libraries. The target must provide a compatible loader and dependency set before application startup can succeed.

Chapter 35 — Buildroot, *after* you can do it by hand

- Why Buildroot exists; what it really automates
- `make menuconfig` for Buildroot
- Reading `output/build/` and `output/target/` — recognizing every step you already did in Ch. 31
- Adding a custom package
- Comparing the generated rootfs against your hand-built one
- **Key Notes:** Buildroot owns the dependency-ordered assembly of selected packages into images. Its configuration describes the intended system rather than a sequence of edits performed on a live target.

Chapter 35D — Supplementary: Bootable SD and eMMC image layout

- Split storage into boot, rootfs, and data partitions
- Use filesystem labels instead of guessed `/dev/mmcblkXpY` names
- Manually partition and format an SD card once, so the image builder is understandable
- Add `/data` to `/etc/fstab`
- Automate the layout with `genimage`
- **Key Notes:** Boot files, the root filesystem and application data have different owners and update lifetimes. Separate partitions make those boundaries visible but do not by themselves provide rollback.

Chapter 35I — Supplementary: Boot from SPI NOR flash

- Explain when the Boot ROM can really boot from SPI/QSPI NOR
- Compare block devices with MTD devices
- Design a small SPI NOR partition map for U-Boot, env, kernel, DTB, rootfs, and data
- Add Device Tree fixed partitions
- Build a SquashFS rootfs and flash it with U-Boot `sf update`

- **Key Notes:** SPI NOR offers eraseblock-based storage through MTD, not an eMMC-style block device. Filesystem and update choices must account for erase geometry and limited capacity.

Chapter 35E — Supplementary: Buildroot product board directory

- Build a product-style `board/<vendor>/<board>/` directory
- Use `rootfs-overlay`, `post-build.sh`, `post-image.sh`, and `genimage.cfg`
- Add kernel and BusyBox config fragments
- Save and rebuild from a product defconfig
- Decide what to commit and what to leave as generated output
- **Key Notes:** A product board directory is the reviewable source of image policy. Generated output is a build result, not the place to preserve product configuration.

Chapter 35A — Supplementary: Ubuntu-base rootfs as a peer to BusyBox/Buildroot

For projects where binary size is not the constraint but **familiarity is** (engineers used to `apt install` on their dev machines), an Ubuntu-base rootfs gives you a fully-fledged Debian-family userland on the target — `apt`, `bash`, full `coreutils`, glibc. We unpack Ubuntu-base, `chroot` into it under `qemu-user-static` to install packages on the host, then NFS-mount it from the target.

- When to choose this vs BusyBox (size, cold-start) or Buildroot (reproducibility)
- The `chroot` + `qemu-user-static` trick for installing target packages from the host
- DHCP and `apt` on the target
- **Key Notes:** Ubuntu-base trades a small, tightly controlled image for familiar packages and a larger maintenance surface. The choice depends on product storage, memory and update policy, not on whether a shell can start.

Chapter 35F — Supplementary: Your application as a service

- Write an app that behaves well under an init system
- Add a BusyBox init script with start, stop, and restart
- Add a systemd unit for Ubuntu-base or systemd Buildroot images
- Decide where config, logs, runtime files, and data belong
- Add simple health checks

- **Key Notes:** A service has a lifecycle contract: dependencies, readiness, shutdown, failure handling and ownership of its files. Installing an executable establishes only one part of that contract.

Chapter 35G — Supplementary: Networking, time, and logging

- Bring up Ethernet manually and with DHCP
- Debug gateway vs DNS failures
- Configure hostname and `/etc/hosts`
- Start NTP and explain why wrong time breaks TLS and package managers
- Configure BusyBox syslog and persistent logs under `/data`
- **Key Notes:** Link state, IP configuration, routing and DNS are separate layers. Success at one layer narrows a diagnosis but does not establish that the next layer works.

Chapter 35B — Supplementary: Read-only rootfs + overlayfs (the industrial pattern)

An immutable base separates the shipped system from writable runtime state. Tmpfs and persistent overlays have different lifetimes, durability requirements and recovery policies. A read-only mount does not make the complete product immune to power loss.

- Mounting `ext4` with `ro`; what fails (`/etc/resolv.conf`, `/var/run/utmp`, `/tmp`)
- `overlayfs` mount syntax: `lowerdir`, `upperdir`, `workdir`
- `/etc/fstab` for an RO root + writable overlays
- The data-partition split: `/data/` for app data, `/var/log/` for logs
- Power-loss boundaries, application commit policy and storage qualification
- Factory reset as a data-lifecycle decision, not live-overlay deletion
- **Key Notes:** An immutable rootfs separates the shipped system from changing data. Read-only mounting reduces exposure to interrupted writes to that filesystem, but does not make the storage device or the entire product immune to power loss.

Chapter 35H — Supplementary: Rootfs security basics

- Separate development convenience from release policy
- Lock root login and use SSH keys
- Run the product app as a non-root user

- Review file permissions, setuid files, secrets, and listening services
- Use read-only rootfs and `/data` mount options as part of defense
- **Key Notes:** Security begins with authority: which account owns a service, which files it can change, and which interfaces it exposes. Running everything as root erases useful boundaries.

Chapter 35C — Supplementary: Container runtimes on embedded (Podman + OCI)

Some products use containers to isolate the application from the base system, so the same base rootfs can host many app updates. We bring up Podman on the i.MX6ULL, run a tiny Alpine container, talk to a host GPIO from inside it, and manage the container with Quadlet.

- Why containers on embedded: app/OS split for OTA, sandboxing, reproducibility
- Podman vs Docker on small devices
- Kernel namespaces, cgroups, and overlayfs requirements
- Bind-mounting `/sys/class/leds` into a container to talk to hardware
- OCI image format; how to build one without Docker on the host
- **Key Notes:** Containers package applications separately from the base system, but share its kernel. Application rollback and kernel or bootloader rollback are different update boundaries.

Appendix — Userspace tooling reference

- One table per tool family (networking, audio, BT, cellular, sensors, debug, …) with both **Ubuntu-base** `apt install` and **Buildroot** `BR2_PACKAGE_*` lines
- “Install everything” cheat sheets (Ubuntu-base one-liner, Buildroot Kconfig fragment)
- Per-Part subsets so a reader doing only Part VI/VII/VIII can install just what they need
- Pitfalls (libgpiod v1↔v2 CLI, missing WiFi firmware, mesh-vs-classic BlueZ, etc.)

PART VI — DRIVER DEVELOPMENT

> *A driver connects a hardware contract to a Linux subsystem. These chapters follow that boundary through discovery, ownership, execution context, data flow and removal. Listings make the contract inspectable; Key Notes collect its design consequences.*

Chapter 36 — Your first kernel module

- The Loadable Kernel Module (LKM) build system: `obj-m`, Kbuild, `KDIR`
- `module_init`, `module_exit`, `MODULE_LICENSE`, `MODULE_AUTHOR`
- `printk` log levels, dmesg
- `insmod`, `rmmod`, `lsmod`, `modinfo`, `depmod`, `modprobe`
- Cross-compiling against the kernel tree you built in Ch. 25
- **Key Notes:** A loadable module joins the kernel's address space. Its callbacks have kernel privileges and no process-level fault isolation.

Chapter 37 — Character device drivers

- The classical chardev pattern
- `register_chrdev_region` / `alloc_chrdev_region`
- `struct cdev` and `cdev_add`
- `struct file_operations`: `open`, `release`, `read`, `write`, `llseek`, `unlocked_ioctl`
- `copy_to_user` / `copy_from_user` — and *why* you can't just memcpy
- A 200-line driver exposing a software FIFO over `/dev/myfifo`
- **Key Notes:** The device number selects a `cdev`, and the VFS dispatches operations through its `file_operations`. The pathname alone does not define device behavior.

Chapter 38 — Auto device nodes (class + device)

- `class_create`, `device_create` and what udev/mdev do with `/sys/class/.../uevent`
- The uevent protocol, hotplug
- **Key Notes:** `cdev` registration provides operation dispatch, while class/device registration describes the instance in sysfs. Publishing a node before its backing interface is ready creates an access race.

Chapter 39 — The platform driver + device tree binding

- Why platform drivers replace the old "register a chardev manually" pattern
- `struct platform_driver`, `of_match_table`, `probe`, `remove`
- The bind dance: DT node + driver `compatible` → `probe()` called
- `platform_get_resource`, `devm_ioremap`, `devm_request_irq`

- Adding your driver’s DT binding under `Documentation/devicetree/bindings/`
- **Focus:** once this clicks, every kernel subsystem looks the same
- **Key Notes:** Device Tree describes a hardware instance, the driver supplies behavior, and the platform bus matches the two. One driver can bind to several devices with separate private state.

Chapter 40 — The misc framework (shortcut chardev)

- When `miscdevice` is enough
- A `/dev/hwrng`-style driver in 80 lines
- **Key Notes:** Misc devices share major 10 and obtain an individual minor. The framework reduces registration plumbing without changing the `file_operations` contract.

Chapter 41 — Concurrency

- The four “lock” families: atomic_t, spinlock, mutex, rwlock + sequence locks
- When to use each (interrupt context vs process context)
- Per-CPU variables
- RCU at an introductory level
- Preemption rules: `preempt_disable`, `local_irq_save`
- Lockdep and how to read its splats
- **Key Notes:** Concurrency depends on access paths, not only CPU count. Syscalls, interrupts, timers, and deferred work can interleave on a single-core system.

Chapter 42 — Sleeping, waiting, polling

- `wait_queue_head_t`, `wait_event_interruptible`, `wake_up`
- Implementing blocking `read()`
- `poll_wait`, `EPOLL` from the driver side
- `O_NONBLOCK` semantics
- **Key Notes:** A wait queue records interest in a condition. The producer owns state changes and wakeups, while the reader remains responsible for checking whether its condition is satisfied.

Chapter 43 — Interrupts (top half, bottom half, threaded)

- `request_irq`, `IRQF_*` flags, shared interrupts
- Top half: keep it short, ack the device, schedule deferred work
- Bottom halves: softirq, tasklet (deprecated), workqueue, threaded IRQ
- DT `interrupts` property and the `interrupt-parent` chain
- **Key Notes:** IRQ domains translate controller-specific interrupt descriptions into the virtual IRQ used by a driver. A DT hardware number is not a portable `request_irq` argument.

Chapter 44 — The GPIO subsystem

- The old `gpio_request`/`gpio_set_value` interface vs the new `gpiod_*` interface
- pinctrl bindings revisited
- Userspace: `/sys/class/gpio` (deprecated) vs `gpiochipN` + `libgpiod`
- **Key Notes:** Pinctrl owns mux and pad configuration, while GPIO consumers own direction and logical value through descriptors. Successful descriptor acquisition does not establish correct board wiring.

Chapter 45 — Input subsystem (button driver)

- `input_dev`, `input_event`, `EV_KEY`, `EV_ABS`, `input_register_device`
- `/dev/input/eventN` and `evtest`
- **Key Notes:** An input driver translates device-specific observations into declared type/code/value events. Evdev and higher-level user-space consumers own delivery and interpretation.

Chapter 46 — I²C drivers

- Master controller (i.MX6ULL has 4 × I²C) vs client driver split
- The i.MX I²C controller (I2C1..4) register-level overview
- Writing a *client* driver for an EEPROM or sensor (e.g., AP3216 ambient light on the Point Atom board)
- DT binding for I²C devices
- Using `i2c-tools` (`i2cdetect`, `i2cdump`, `i2cset`, `i2cget`) to validate

- **Key Notes:** The adapter owns controller operation, the client identifies one bus address, and the chip driver owns device semantics. A DT client description is not proof of a responding chip.

Chapter 47 — SPI drivers

- SPI subsystem, master + slave
- ECSPI controller on i.MX6ULL
- Writing a driver for an SPI flash or ADC
- `spidev` for quick userspace prototyping
- **Key Notes:** The SPI controller owns bus scheduling and signaling, while each `spi_device` carries chip-select and configuration constraints. The chip driver owns command framing and interpretation.

Chapter 48 — PWM and RTC

- PWM framework, `pwm_chip`, sysfs `/sys/class/pwm/`
- RTC framework: `rtc_class_ops`, `hwclock`, NTP integration
- **Key Notes:** A PWM provider implements hardware timing, while a consumer owns the signal’s purpose. Backlight, fan, and beeper frameworks add role-specific policy above raw period and duty cycle.

Chapter 49 — IIO (Industrial I/O) for ADC and sensors

- Why IIO replaced ad-hoc sensor drivers
- Channels, triggers, buffers
- A driver for the on-chip ADC1 of the i.MX6ULL
- Reading from `/sys/bus/iio/devices/iio:device0/in_voltage0_raw`
- **Key Notes:** Channels describe measurable quantities and their attributes. The sensor driver supplies acquisition and conversion semantics, while IIO supplies a common user-space interface.

Chapter 50 — regmap

- The pattern: every device driver was duplicating “read/modify/write a register” code
- `regmap_init_mmio`, `regmap_init_i2c`, `regmap_init_spi`
- `regmap_update_bits`, cache types, debugfs integration
- Refactor an earlier chapter’s driver to use regmap

- **Key Notes:** Regmap separates register semantics from bus encoding. Address width, value width, endianness, and command flags must describe the actual chip protocol.

Chapter 51 — DMA

- The DMA-API: `dma_alloc_coherent` vs streaming `dma_map_single`
- Cache coherency: why the kernel cares about direction and ownership
- i.MX6ULL SDMA controller overview
- A DMA-driven UART example (or audio DMA preview)
- **Key Notes:** The DMA provider owns the controller, while a consumer owns transfer intent, buffers, and completion handling. A channel request must match the peripheral’s routing and capabilities.

Chapter 51A — Supplementary: Watchdog driver and brown-out resilience

No product ships without one. The i.MX6ULL `WDOG1/WDOG2` modules generate a system reset if not “kicked” within a programmable window. We write the kernel driver, plumb the user-space daemon (`systemd-watchdog` or `busybox watchdog`), and design the application-level kick policy.

- The kernel `watchdog` framework, `struct watchdog_device`, `wdog_ops`
- `/dev/watchdog` and the `ioctl` interface
- Window timing: too-fast kicks are as bad as too-slow
- Boot-time vs runtime: when does the WDOG arm?
- Pre-timeout warnings for graceful shutdown
- **Key Notes:** A hardware watchdog provides a reset path that is less dependent on kernel scheduling than a software timer. Its actual clock, power, and reset dependencies bound its independence.

Chapter 51B — Supplementary: Power management — runtime PM, suspend/resume, DVFS

Battery-powered or thermal-constrained products demand real PM. The Linux PM core supports four orthogonal mechanisms: runtime PM (per-device idle), system suspend (whole-board sleep), DVFS (frequency/voltage scaling), and CPU idle (low-power C-states).

- The PM-runtime callbacks: `runtime_suspend`, `runtime_resume`, `runtime_idle`

- `pm_runtime_get_sync` / `pm_runtime_put` discipline in drivers
- System sleep states: `freeze`, `standby`, `mem`, `disk` — what's implemented on i.MX6ULL
- DVFS: `cpufreq` governors (ondemand, schedutil, conservative)
- The CPU idle subsystem (`cpuidle`); WFI as the C1 state
- `tickless` (`CONFIG_NO_HZ_IDLE`) for power
- **Key Notes:** Runtime PM follows a device's activity, DVFS changes operating points under load, and system sleep coordinates the whole platform. These are distinct policies with different dependency and wakeup requirements.

Chapter 52 — Network driver (FEC + KSZ8081 PHY)

- The netdev model: `struct net_device`, `ndo_*` ops, NAPI
- The FEC (Fast Ethernet Controller) on i.MX6ULL
- The PHY subsystem and MDIO
- DT bindings for ethernet
- Bringing the interface up, running `ping`, `iperf`, `tcpdump`
- **Key Notes:** Ethernet divides responsibility between the MAC, PHY and Linux network stack. MDIO manages PHY state, while RMII carries the frame data and clock relationship.

Chapter 52A — Supplementary: PREEMPT_RT — real-time Linux as a full chapter

For motion control, audio, robotics, industrial protocols, mainline Linux's PREEMPT_RT patchset turns the kernel into a deterministic real-time kernel without giving up the Linux API. Since v6.12 most of PREEMPT_RT has been merged into mainline; the rest is in flight.

- What "real-time" means precisely (worst-case latency, not "fast")
- The PREEMPT_RT design: threaded IRQs, sleeping spinlocks, rt_mutex with priority inheritance, ftrace-anchored measurement
- Building a PREEMPT_RT kernel on i.MX6ULL
- `cyclictest` — the standard latency benchmark
- Affinity, isolation (`isolcpus=`), and the `SCHED_FIFO`/`SCHED_DEADLINE` schedulers
- Tuning: disable CPU-idle deeper states, pin IRQs, lock memory with `mlockall`
- What still cannot meet hard-RT requirements on a Cortex-A7 single-core

- **Key Notes:** Real-time design concerns deadlines and worst-case interference, not average throughput. PREEMPT_RT reduces sources of kernel scheduling latency but does not establish a universal deadline guarantee.

Chapter 53 — Sound (ALSA / ASoC)

- The “Sound Open Architecture” stack
- ASoC: machine driver + codec driver + platform driver
- SAI controller on i.MX6ULL + WM8960 codec (typical Point Atom config)
- `aplay`, `arecord`, `alsamixer`
- **Key Notes:** ASoC separates the CPU audio interface, codec and board-specific connection. The machine description joins reusable drivers rather than replacing their responsibilities.

Chapter 54 — LCD framebuffer and DRM

- The eLCDIF controller
- Legacy fbdev (`/dev/fb0`) vs modern DRM/KMS (`/dev/dri/card0`)
- Writing pixels from user space; testing with `fbset`, `modetest`
- Touchscreen integration (GT9147 typical Point Atom)
- **Key Notes:** DRM/KMS separates framebuffer storage, scanout, mode selection and output connections. Legacy fbdev exposure does not describe the whole display pipeline.

Chapter 54A — Supplementary: MTD / UBI / UBIFS for raw NAND

If your product flashes raw NAND (not eMMC), you need the MTD subsystem under the kernel and UBI/UBIFS on top for wear-leveling and bad-block management.

- MTD model: `struct mtd_info`, `mtd_read/write/erase`
- The NAND subsystem (`nand_chip`) on top of MTD
- Bad-block management (BBT, OOB layout)
- UBI: volumes, wear-leveling, attach/detach
- UBIFS: a journaling FS on UBI
- `mtdinfo`, `ubinfo`, `ubiformat`, `ubinize`

- **Key Notes:** MTD exposes raw flash geometry and operations, UBI manages logical eraseblocks and wear, and UBIFS provides a filesystem above UBI. These layers solve different problems.

Chapter 54B — Supplementary: V4L2 + GStreamer for camera input (CSI)

The i.MX6ULL CSI is paired with an OV5640 (typical) or OV2640 sensor. V4L2 (Video4Linux 2) is the kernel framework; GStreamer is the user-space pipeline glue.

- V4L2 model: `struct v4l2_device`, `struct video_device`, `v4l2_ioctl_ops`
- The i.MX6ULL `imx-pxp` and `imx-csi` drivers
- A user-space `v4l2-ctl` capture: `v4l2-ctl --stream-mmap --stream-to=...`
- GStreamer pipeline: `v4l2src ! videoconvert ! kmssink` → fullscreen camera
- The bayer/YUV/RGB conversion path
- Encoding to JPEG/H.264 on i.MX6ULL (none in HW; software encoder via `libjpeg`/`x264`)
- **Key Notes:** The capture pipeline joins sensor, interface and memory buffers through V4L2 and, where applicable, the media graph. GStreamer consumes that exposed stream in user space.

Chapter 55 — USB gadget

- USB host vs device modes; OTG role detection
- `libcomposite` and ConfigFS-based gadgets
- A “USB serial” and a “USB mass storage” gadget walkthrough
- **Key Notes:** A USB gadget is the device side of a connection, not a USB host. The controller role, connector wiring and VBUS handling must support that role.

Chapter 55A — Supplementary: Kernel timers and high-resolution timers (hrtimers)

Three timer APIs in the kernel: legacy `timer_list` (jiffies-based, ms resolution), `delayed_work` (workqueue-driven), and `hrtimer` (sub-µs, the modern default for new code). When to use which; how each is implemented under the hood; how the tick subsystem multiplexes many timers on one hardware event.

- `mod_timer`, `del_timer_sync`, the `timer_list.function` callback
- `hrtimer_init`, `hrtimer_start`, `HRTIMER_MODE_REL` vs `_ABS`, `_PINNED`
- `timer_setup` (the modern, type-safe API)

- **Key Notes:** Kernel timers schedule callbacks, not private threads. Their execution context determines which operations are legal, while workqueues provide a place for work that may sleep.

Chapter 55B — Supplementary: Asynchronous notification (SIGIO / fasync)

The fourth I/O model after blocking/non-blocking/poll: the driver delivers a `SIGIO` to a registered user-space process when data is ready, so the process is "informed" without ever calling `read()`. `fasync_helper`, `kill_fasync`, the `F_SETOWN`/`F_SETFL O_ASYNC` user-space dance.

- When SIGIO matters: legacy code, signal-driven simple loops, low-latency without busy-poll
- Why epoll/io_uring have mostly replaced it for new code, and why drivers still ship it
- **Key Notes:** SIGIO is a notification that device state may need attention, not the data itself. The application's queue or read interface remains the source of events.

Chapter 55C — Supplementary: CAN bus and FlexCAN

Industrial automation runs on CAN. The i.MX6ULL has two **FlexCAN** controllers. The Linux SocketCAN subsystem treats CAN like a network interface: `ip link set can0 up type can bitrate 500000`. Then `cansend`, `candump`, `can-utils`.

- The FlexCAN controller register map and message-buffer model
- SocketCAN: `PF_CAN`, `SOCK_RAW`, `struct can_frame`
- `iproute2` CAN extensions; `can-utils`
- ISO-TP, CAN-FD — what i.MX6ULL supports and what it doesn't
- **Key Notes:** SocketCAN exposes CAN through Linux networking while FlexCAN owns controller behavior. The external transceiver and bus wiring remain essential physical dependencies.

Chapter 55D — Supplementary: Block device drivers

Char devices are sequential; block devices are random-access in fixed-size sectors. The block-I/O layer multiplexes requests from many filesystems onto one device with elevator/scheduling policies.

- `struct block_device`, `struct gendisk`, `struct block_device_operations`
- The `request_queue`, `make_request_fn`, modern `blk_mq` multi-queue
- A 4 MB RAM-disk block driver in < 200 lines

- How `/dev/mmcblk0` and `/dev/sda` look under the hood
- **Key Notes:** The block layer describes sector-oriented I/O through bios, requests and queues. A block driver must honor offsets, lengths and completion semantics rather than act like an unstructured byte stream.

Chapter 55E — Supplementary: WIFI — wpa_supplicant, USB and SDIO dongles

Connecting an i.MX6ULL to Wi-Fi in 2026 is rarely an in-house driver job — you pick a chip with mainline support and stand up `wpa_supplicant`. We walk an RTL8188EUS USB Wi-Fi dongle (cfg80211) and an RTL8189FS SDIO module (vendor staging tree) end-to-end.

- Kernel: `CONFIG_CFG80211`, `CONFIG_WIRELESS_EXT`, `CONFIG_RTL8XXXU`
- Firmware blobs in `/lib/firmware/`
- `wpa_supplicant.conf`, WPA2-PSK, hidden SSIDs, EAP-PEAP
- `wpa_cli` and the `D-Bus` interface
- Diagnosing “no SSID found”: antenna, country code, channel-12/13, regulatory domain
- **Key Notes:** Wi-Fi spans bus discovery, chip firmware, kernel wireless interfaces, authentication and IP networking. Failure at one layer should not be diagnosed as a failure of the whole stack.

Chapter 55F — Supplementary: Cellular modems (PPP, ECM/NCM, GNSS)

Quectel EC20-style modems present themselves as USB composite devices: PPP serial, RNDIS/ECM data, NMEA GNSS. We cover all three modes, plus `quectel-CM` and `qmi_wwan` for the QMI-based path.

- USB option driver, vendor/product IDs, the `option_ids[]` table in `drivers/usb/serial/option.c`
- PPP path: `pppd` chat script, APN, MTU
- ECM/NCM: a virtual Ethernet over USB; `usb0`
- GNSS: `/dev/ttyUSB1` NMEA stream → `gpsd` → `gpsmon`
- **Key Notes:** Cellular integration separates modem control, the data interface and host network configuration. QMI, MBIM, Ethernet-like modes and PPP imply different drivers and management paths.

Chapter 55G — Supplementary: Multi-touch — MT-A, MT-B protocols, GT911

The Linux input subsystem evolved two multi-touch protocols: **Type A** (per-frame, packed) and **Type B** (per-slot, persistent). Most modern panels use Type B. We bring up a Goodix **GT911** capacitive touch controller (5-point), the typical pairing for the Point Atom MINI's optional LCD.

- I²C wiring + interrupt + reset of GT911
- `input_mt_init_slots`, `input_mt_slot`, `input_report_abs(ABS_MT_POSITION_X/Y)`
- `evtest` and how to read a multi-touch event stream
- `tslib` user-space library for legacy single-touch panels
- **Key Notes:** MT-B reports contacts through slots and tracking IDs. Finger lifetime is different from the number of coordinate samples received.

Chapter 55H — Supplementary: RGB-to-HDMI via sii902x

The i.MX6ULL has no native HDMI. An external parallel-RGB-to-HDMI bridge illustrates the DRM bridge model, board wiring and mode constraints shared by controller, bridge and sink.

- The bridge chip's I²C control path vs. its parallel-RGB data path
- `drm/bridge` model in the kernel; `drm_bridge_add`
- EDID reading; mode selection from a list
- **Key Notes:** A DRM bridge transforms one display interface into another between the controller and connector. The i.MX6ULL has no native HDMI, so the external bridge and its wiring are part of the design.

Chapter 55I — Supplementary: Rust-for-Linux — first kernel module

Since Linux 6.1, Rust is a supported language for kernel modules. As of 2026 the support is still gated to a small set of subsystems but is growing fast. We write our Chapter 36 hello-LKM in Rust, as a sidebar / supplemental experiment — not because we recommend Rust for production embedded yet, but because the reader will see it land in their next kernel update.

- Why Rust in the kernel (memory safety, type-state)
- `rust/` directory tour; how the build picks up Rust source
- `kernel::prelude::*`, `module!{}` macro
- Compare side-by-side with the C version from Chapter 36

- What’s stable, what’s nightly, what’s blocked
- **Key Notes:** Rust changes how memory ownership and some concurrency rules are expressed, but a driver still needs a correct hardware and kernel-interface design.

PART VII — DEVICE COOKBOOK

> *Part VI taught you the kernel’s driver frameworks with one good example each. This Part is the reference: for each common device class, two or three real chips compared side-by-side, with schematics, DT bindings, driver code or kernel-driver enablement, user-space access, and the pitfalls specific to each chip.*
>
> *The device chapters are independent design references. Compare the physical measurement or output, its electrical and timing requirements, Linux support and the application interface. Schematics and listings explain those relationships without requiring a separate project for every chip.*

Group A — Storage devices

Chapter 64 — QSPI NOR flash

- **Chips:** Winbond W25Q128 / W25Q64; Macronix MX25L256; Micron MT25Q. Trade-offs (size, speed, erase granularity, security features).
- The i.MX6ULL QSPI controller and its limits
- DT bindings for `spi-nor`; `m25p80`; the `jedec,spi-nor` compatible
- MTD partitions for u-boot env, kernel, dtb, rootfs slot A/B
- Erasing and writing from user-space: `mtd-utils` (`flash_erase`, `nandwrite`, `flashcp`)
- XIP from QSPI — when it makes sense, when it doesn’t
- **Pitfalls:** read-only mode after lockdown; quad-mode vs single-mode performance trap
- **Key Notes:** NOR programming, erase and read are different operations with different alignment and timing rules. Write-enable and bounded busy polling belong to the device state machine, not just to a convenient API.

Chapter 65 — I²C / SPI EEPROM

- **Chips:** Microchip AT24C02/32/256/512 (I²C); AT25-series (SPI); 25LCxx (SPI).

- The `at24` kernel driver and what its DT properties really do (`pagesize`, `read-only`, `wp-gpios`)
- The `nvmem` framework: where EEPROM cells become typed kernel-side accessors
- Writing the MAC address from EEPROM into the FEC at probe time
- Wear: page-write counting, the “spread your writes” pattern
- **Pitfalls:** I²C addressing scheme variants; the 256-byte boundary “rollover” bug
- **Key Notes:** EEPROM is useful for small persistent records such as identity and calibration. Its byte-oriented interface still has page-program, completion and endurance constraints.

Chapter 66 — SD card and eMMC deep dive

- The MMC subsystem; SDIO is the same machinery (Ch 91 will reuse it)
- Speed modes: DS, HS, HS200, HS400 — what your eMMC and your i.MX SDHC support
- CMD23 vs CMD12 multi-block read terminators
- eMMC partition table: boot partitions, RPMB, GP areas
- Wear-levelling: what eMMC firmware does, what UBI does, what neither does
- TRIM/discard for eMMC vs SD
- **Key Notes:** The MMC core manages card protocols, the host driver manages the controller, and the block layer exposes storage requests. A failure can occur at any of these boundaries.

Group B — Environmental sensors

Chapter 67 — Temperature / humidity / pressure

- **Chips:** Bosch BME280 (T/H/P I²C+SPI); Sensirion SHT3x (T/H I²C, lab-grade); ASAir AHT20 (T/H I²C, cheap).
- IIO drivers for each; how user-space reads via `/sys/bus/iio/devices/`
- Calibration registers in BME280; why the raw reading isn’t the temperature
- Polling vs trigger-driven sampling
- Time-averaging in user-space: simple moving average vs exponential
- **Pitfalls:** self-heating in BME280 (the device’s own current warms its sensor); the “after power-on wait 2 ms” SHT3x trap

- **Key Notes:** Raw sensor counts become useful measurements only through the correct scale or compensation path. BME280 calibration coefficients belong to the individual chip.

Chapter 68 — Light & color sensors

- **Chips:** Rohm BH1750 (ambient lux, I²C); AMS TSL2561 (broad-band + IR-channel, I²C); Vishay VEML7700 (low-power lux).
- IIO drivers, `in_illuminance_input` interpretation
- The “logarithmic eye” — what lux means and when it lies
- **Bonus:** AMS TCS34725 (RGB color sensor) and Avago APDS-9960 (RGB + gesture); driver and DT included
- **Pitfalls:** TSL2561’s IR-saturation regime; lens / window choice on enclosure
- **Key Notes:** Light sensors do not all observe the same spectral response. Lux conversion, color channels and infrared rejection are device-specific design choices.

Chapter 69 — Air quality, gas, particulate matter

- **Chips:** Sensirion SCD30 (CO_2 NDIR, I²C); AMS CCS811 (eCO_2/TVOC, I²C); Plantower PMS5003 (PM1/2.5/10, UART); Nova SDS011 (PM2.5/10, UART); MQ-x analog series + ADC.
- Why “eCO_2” is *not* CO_2 (CCS811 estimates from VOCs and resistance drift)
- NDIR true-CO_2 measurement (SCD30); cost vs accuracy
- Laser-scattering PM sensors; airflow control via fan PWM
- The 24-hour warm-up problem with CCS811
- **Pitfalls:** MQ-x cross-sensitivity (most “MQ-2 = LPG” claims are wrong); PM-sensor fan failure detection
- **Key Notes:** NDIR CO2, metal-oxide gas response and optical particle sensing measure different quantities. An inferred eCO2 value must not be presented as a direct CO2 measurement.

Group C — Motion sensors

Chapter 70 — I²C IMUs

- **Chips:** InvenSense MPU6050 (6-axis, classic); MPU9250 (9-axis with mag); InvenSense ICM-20948 (newer 9-axis).
- The IIO trigger/buffer mechanism for 6-/9-axis IMUs at 1000 Hz

- DMP firmware blob in MPU6050 — what it gives you (quaternions), what it costs (closed-source, less control)
- The magnetometer-as-slave-I²C-master quirk in MPU9250
- Tilt-compensated heading; sensor fusion via Madgwick filter in user-space
- **Pitfalls:** stale I²C addresses (0x68/0x69); gyro bias drift after temp changes
- **Key Notes:** An IIO trigger and buffer decouple capture from user-space reads. Channel layout, scale and timestamps form the sample interface consumed by later processing.

Chapter 71 — SPI IMUs (high-rate / low-noise)

- **Chips:** ST LSM6DSO (6-axis); InvenSense ICM-42688 (high-rate, low-noise); Analog Devices ADXL345 (3-axis accel only).
- Why SPI for IMUs above ~400 Hz
- ICM-42688’s FIFO and watermark-IRQ — sampling thousands of points per second efficiently
- ODR / range / filter settings
- **Pitfalls:** SPI mode 3 vs mode 0 confusion; ODR vs filter-cutoff mismatch
- **Key Notes:** A FIFO watermark batches samples to reduce interrupt and bus overhead. Larger batches also increase delivery latency and the amount of data at risk during a stall.

Group D — Position & distance

Chapter 72 — Distance & proximity

- **Chips:** STMicro VL53L0X (ToF laser, I²C); VL53L1X (longer range); HC-SR04 (ultrasonic GPIO); Sharp GP2Y0A (IR analog).
- The VL53L0X firmware-loading dance
- ToF lighting/glass surface failure modes
- HC-SR04 vs GPIO timing — why this is hard on Linux without a high-precision timer or PRU
- **Pitfalls:** VL53L0X bus collisions (I²C address conflicts); HC-SR04 +5 V level-shifting
- **Key Notes:** Distance sensors use different physical mechanisms, so target shape, ambient conditions and range affect them differently. Software cannot make their limitations interchangeable.

Chapter 73 — Magnetometer / compass

- **Chips:** Honeywell HMC5883L (I²C, common but EOL); QST QMC5883L (cheap clone); Memsic MMC5983MA (newer, lower noise).
- Hard-iron and soft-iron calibration; why your compass points 23° wrong from the start
- IIO `magn_x/y/z` output and the `scale` attribute
- **Pitfalls:** mounting the magnetometer near a buck regulator destroys readings
- **Key Notes:** A magnetometer measures a field vector, not an already-correct compass heading. Scale, axis orientation and calibration belong in the interpretation path.

Chapter 74 — Hall-effect & rotary position sensors

- **Chips:** AMS AS5048A (magnetic rotary, SPI/I²C); Allegro A1324 (linear Hall analog); Infineon TLE5012 (high-rate angular sensor).
- Off-axis vs on-axis sensor placement
- The IIO `angl` channel and absolute-position rotary encoders
- High-rate angular sampling for motor commutation
- **Pitfalls:** magnet-distance sensitivity (1 mm changes everything); diametric vs axial magnets
- **Key Notes:** Rotary magnetic sensing depends on magnet type, alignment and field strength as well as electronics. Mechanical geometry is part of the measurement design.

Group E — Power & current

Chapter 75 — Current and power monitoring

- **Chips:** TI INA219 (low-side, I²C); INA226 (improved, calibration); INA3221 (3-channel); Allegro ACS712 (Hall analog).
- Shunt selection: heat, accuracy, layout
- INA226 calibration register — why “current_lsb = max_current / 32768”
- 3-rail simultaneous monitoring via INA3221
- **Pitfalls:** shared-ground violations; the “ACS712 zero point isn’t 2.5 V” calibration

- **Key Notes:** The shunt, analog input range and calibration setting define the current measurement. Digital resolution alone does not determine useful accuracy.

Chapter 76 — Battery fuel gauge + charger

- **Chips:** Maxim MAX17048 (1-cell fuel gauge, I²C); TI BQ27441-G1; TP4056 (cheap charger); MCP73833; TI BQ24074 (path-managed charger).
- The battery-empty problem and SoC (state-of-charge) estimation
- Charger thermal regulation and JEITA temperature profiles
- The `power_supply_class` kernel framework; `/sys/class/power_supply/`
- **Pitfalls:** TP4056 mass-production cells without temperature monitoring; SoC drift over months
- **Key Notes:** Fuel estimation, charging, cell protection and power-path management are separate responsibilities. A Linux battery percentage is not a charging-safety mechanism.

Group F — Specialty sensors

Chapter 77 — 1-Wire sensors

- **Chips:** Maxim DS18B20 (digital thermometer, 1-Wire); DHT11 / DHT22 (single-wire T/H, *not* true 1-Wire); DS2480B (1-Wire master).
- The kernel `w1` subsystem and its slave drivers
- Why DHT11/22 is a timing-critical mess on a non-RT Linux (and why dedicated MCU helper is sometimes the answer)
- Parasitic-power mode for DS18B20
- **Pitfalls:** the “USB-Serial DS9097 emulator does the parity tricks for you” pattern
- **Key Notes:** Maxim 1-Wire and the DHT22’s single-signal protocol are not the same interface. Similar wiring does not imply compatible timing or drivers.

Chapter 78 — MEMS microphones (I²S)

- **Chips:** InvenSense INMP441 (I²S MEMS); ICS-43434 (lower noise); SPH0645 (similar).
- ASoC DAI for I²S microphones; sampling via `arecord`
- Stereo via two mono mics; beamforming basics
- **Pitfalls:** WS line ringing on long cables; PDM vs I²S confusion

- **Key Notes:** An I2S microphone supplies digital samples through a clocked audio interface. ASoC joins that interface to the CPU DAI and DMA path without inventing a register-controlled codec.

Chapter 79 — Health sensors (HR / SpO2)

- **Chips:** Maxim MAX30100 (basic); MAX30102 (improved, smaller); MAX30105 (with particle-sensing channel).
- PPG (photoplethysmography) raw signal, IR vs red channel
- Heart-rate extraction in user-space; SpO2 calibration constants
- **Pitfalls:** motion artifacts (this is why a Fitbit's HR is noisy when you walk); SpO2 accuracy claims
- **Key Notes:** The sensor supplies optical samples, while signal processing estimates pulse-related quantities. Raw red and infrared counts are not heart rate or oxygen saturation.

Group G — Analog conversion & clock generation

Chapter 80 — External ADCs

- **Chips:** TI ADS1115 (16-bit I²C); ADS1256 (24-bit SPI low-noise); Microchip MCP3008 (10-bit 8-ch SPI cheap); Analog Devices AD7606 (16-bit 8-ch parallel sampling, true simultaneous).
- IIO drivers; sample rate vs ENOB trade-offs
- AD7606's true-simultaneous sampling for power-quality / vibration
- Ratiometric measurements (load cell, RTD)
- **Pitfalls:** input filter loading; reference-voltage stability
- **Key Notes:** Resolution, conversion rate, input range and simultaneous sampling are independent selection criteria. A larger nominal bit count does not ensure more useful information.

Chapter 81 — External DACs + clock generators

- **Chips:** Microchip MCP4725 (12-bit I²C DAC); Analog Devices AD5663 (16-bit SPI DAC); SiLabs Si5351 (programmable 3-output clock generator); CDCE9xx series.
- IIO `out_voltage*` for DACs
- Si5351 — generating arbitrary clocks for SDR or for chip evaluation
- The kernel `clk` framework integration with external clock chips

- **Pitfalls:** PLL closed-loop stability; Si5351's "frequency = output * 2^N / M" config nightmare
- **Key Notes:** A DAC is an analog-output provider, while a clock generator is a clock provider. They share a control bus but expose different contracts to Linux consumers.

Group H — Displays

Chapter 82 — RGB parallel LCD on LCDIF

- **Panels:** ATK4384 (4.3" 480×272); ATK7016 (7" 1024×600); ATK10261 (10.1" 1280×800).
- The DPI / RGB-parallel timing model: pixel clock, hsync, vsync, porches
- `panel-simple` and how to add a custom panel
- DRM/KMS basics for our case; the `imx-lcdif` driver
- Backlight via PWM + `pwm_bl`
- **Pitfalls:** pixel-clock too fast → tearing; reversed RGB swap on the connector
- **Key Notes:** A parallel RGB panel consumes a continuously timed pixel stream from LCDIF. Active pixels, blanking intervals and polarities must agree with its electrical specification.

Chapter 83 — SPI LCD

- **Panels:** ST7789 (240×320, 16-bit/18-bit color); ILI9341 (320×240); ILI9488 (320×480).
- mainline `panel-mipi-dbi` driver + DRM tiny architecture
- fbtft legacy and why we don't use it for new designs
- Tearing-effect output and synchronized updates
- **Pitfalls:** SPI clock ceiling vs panel refresh; backlight-PWM bleed
- **Key Notes:** An SPI display controller stores pixels and refreshes its glass independently. The host updates controller memory rather than generating every raster timing edge.

Chapter 84 — QSPI LCD

- **Panels:** GC9D01, ST77916, Sitronix SF055A; round LCDs popular for smart watches
- Quad-SPI mode in the i.MX6ULL QSPI controller
- The high-bandwidth bytes-per-frame budget

- mipi-dbi-spi quad-mode driver
- **Pitfalls:** quad mode timing margins; data-order swap
- **Key Notes:** Quad data lanes can reduce pixel-transfer time, but command phases, overhead and refresh synchronization still affect usable frame rate.

Chapter 85 — OLED & e-paper

- **Panels:** SSD1306, SH1106 (small OLED I²C/SPI); SSD1322 (large yellow OLED); SSD1680 (2.9” e-paper); IT8951 (large e-paper controller).
- `ssd1307fb` legacy framebuffer driver; `repaper` (e-paper DRM driver)
- e-paper refresh modes: full vs partial; ghosting
- **Pitfalls:** OLED burn-in; e-paper partial-refresh ghost mitigation
- **Key Notes:** OLED and e-paper have different refresh architectures. OLED updates controller memory for an active display, while e-paper applies a panel-specific waveform to change a retained image.

Chapter 86 — Touch input ICs

- **Chips:** Vishay TTP223 (single-key cap, GPIO output); NXP MPR121 (12-key cap, I²C); XPT2046 (4-wire resistive touch, SPI).
- Adding capacitive touch buttons to a Linux input device via `gpio-keys`
- MPR121 IIR baseline tracking; setting touch/release thresholds
- 4-wire resistive touch + ADS7846/XPT2046 kernel driver; calibration via `xinput_calibrator`
- Compare with GT911 multi-touch (covered in Ch 55G — separate chip class)
- **Pitfalls:** XPT2046’s interrupt-pin pulldown; resistive panels’ temperature drift
- **Key Notes:** A capacitive button, multi-electrode touch controller and resistive panel expose different signals. The input subsystem can unify events without erasing their physical differences.

Group I — Cameras

Chapter 87 — Parallel CSI cameras

- **Sensors:** OmniVision OV7725 (0.3 MP); OV5640 (5 MP, parallel mode); OV2640 (2 MP); GalaxyCore GC2145 (2 MP).
- The i.MX6ULL parallel CSI (no MIPI-CSI on this SoC)

- V4L2 sensor sub-device model
- Sensor-side: PLL, clocks, exposure, gain, white balance, autofocus (OV5640)
- IPU/CSI capture pipeline
- **Pitfalls:** the 5 MP “still mode” of OV5640 vs streaming; FOV vs lens choice
- **Key Notes:** A parallel camera has a control path over I2C and a pixel path into CSI. A sensor probe proves only the first of those boundaries.

Chapter 88 — USB UVC cameras

- The `uvcvideo` driver; what works mainstream, what doesn’t
- Bandwidth budgeting on USB 2.0 (480 Mbps theoretical; ~30 MB/s practical) — implications for resolution × frame rate
- MJPEG vs YUYV vs H.264 modes
- **Pitfalls:** bandwidth contention with USB WiFi or USB Ethernet; non-UVC “drivers needed” cameras
- **Key Notes:** UVC describes a class-based control and streaming interface. Reusing a class driver reduces integration work but does not remove device quirks or mode constraints.

Group J — Audio devices

Chapter 89 — I²S audio codecs

- **Chips:** Cirrus WM8960 (used on many i.MX boards); SGTL5000 (NXP); Everest ES8388 (cheap, ESP32 favourite); TI TLV320AIC3104; Realtek ALC5640.
- ASoC machine + codec + cpu-DAI triplet (referenced in Ch 53)
- Differences: built-in DAC quality, PGA capability, mic-preamp
- Headphone-jack detection
- **Pitfalls:** master/slave clock-mode mismatch with i.MX SSI; sample-rate clock-derivation
- **Key Notes:** A codec driver joins register control, digital audio format and analog routing. The board’s machine description connects it to the CPU audio interface.

Chapter 90 — Digital class-D amplifiers

- **Chips:** TI TAS5805M (I²C-controlled DSP amp); Maxim MAX98357A (pure I²S, no I²C); TI PCM5102A (audio DAC + headphone).

- The “amp without a codec” pattern: send raw I²S, amp does the rest
- TAS5805M’s DSP coefficients (EQ, compression, dynamic-range control)
- The Bluetooth-speaker product pattern: A2DP source → I²S → MAX98357A
- **Pitfalls:** ground-loop hum in class-D speakers; TAS5805M’s I²C dance during startup
- **Key Notes:** A digital amplifier can accept I2S without offering a register interface. Linux still needs a component and DAI description to connect the audio path.

Group K — WiFi modules

Chapter 91 — SDIO WiFi

- **Modules:** AP6212 (Broadcom-based, used on Point Atom boards); Realtek RTL8189FTV; Marvell SD8801.
- The `brcmfmac` driver for Broadcom; `rtl8xxxu` and out-of-tree `rtl8189es`
- Firmware loading and per-board NVRAM file
- SDIO-WiFi clock and pin signal-integrity issues
- **Pitfalls:** AP6212 + Realtek firmware blob version mismatch; NVRAM regulatory-domain
- **Key Notes:** SDIO discovery is a transport milestone before Wi-Fi firmware, calibration and authentication can work. Each stage provides different diagnostic evidence.

Chapter 92 — USB WiFi

- **Modules:** Realtek RTL8188EUS (cheap dongles); MediaTek MT7601; Ralink RT5370.
- `rtl8xxxu` (mainline) vs out-of-tree `rtl8188eus` (better)
- Driver kernel-version compatibility hell
- Bandwidth contention vs USB Camera
- **Pitfalls:** out-of-tree DKMS pain; ralink stale signal
- **Key Notes:** USB removes much board-specific bus wiring but does not remove firmware, driver or radio-policy dependencies.

Chapter 93 — Hosted WiFi via ESP32 / ESP8266

- **Modules:** ESP32 (esp-hosted firmware); ESP8266 (AT-command firmware).
- The `esp-hosted` driver: SPI/UART transport, Linux sees a normal `wlan0`

- AT-command mode: ESP as a TCP/IP offload engine
- When this beats SDIO: legacy SoCs without SDIO, MCU+Linux co-existence
- **Pitfalls:** AT-command flow control; esp-hosted firmware version pinning
- **Key Notes:** Hosted networking and AT-command networking put the IP-stack boundary in different places. That choice determines whether Linux sees a normal network interface or an application-specific socket service.

Chapter 94 — WiFi+BT combo modules

- **Modules:** AP6212 (BCM43438 — combined); Realtek RTL8723BS (SDIO combo).
- The shared-antenna problem and PTA (packet-traffic arbitration)
- Bringing up both Wi-Fi and BT simultaneously on a shared module
- BT-over-UART + Wi-Fi-over-SDIO on the same chip
- **Pitfalls:** BT/Wi-Fi co-channel coexistence (2.4 GHz crowded)
- **Key Notes:** A combo module contains distinct Wi-Fi and Bluetooth paths, often using SDIO and UART respectively. Success of one interface does not prove the other works.

Group L — Bluetooth & mesh

Chapter 95 — HCI BLE over UART/USB

- **Modules:** Nordic nRF52 with Zephyr HCI firmware; Broadcom BCM4343 BT; CSR8510 USB.
- The Bluetooth HCI specification; how Linux’s `bluez` talks HCI over UART or USB
- `hciattach`, `btmgmt`, `bluetoothctl`
- BLE central vs peripheral; GATT services
- **Pitfalls:** UART flow-control; HCI command/event timing
- **Key Notes:** HCI separates the radio controller from the Bluetooth host stack. BlueZ exposes application services above that boundary rather than requiring applications to implement link-layer behavior.

Chapter 96 — AT-command BLE modules

- **Modules:** HM-10 (CC2540 with AT-command firmware); HC-08; JDY-08.
- The “easy” path: BLE without a kernel stack — UART AT commands

- Limitations: max throughput, no real GATT control, vendor-specific quirks
- When this is the right choice (legacy, simplicity, BOM cost)
- **Pitfalls:** firmware variant differences (HM-10 cloned five ways)
- **Key Notes:** An AT-BLE module owns most of the Bluetooth stack and exposes a serial-facing service. Linux sees its command/data protocol, not necessarily an HCI controller managed by BlueZ.

Chapter 97 — BLE Mesh

- BLE Mesh protocol layers; provisioning, addressing, models
- `bluez-mesh` daemon and the `meshctl` tool
- Sample mesh networks: 20-node room lighting controller
- **Pitfalls:** provisioning friction; mesh-traffic flood prevention
- **Key Notes:** Mesh models define application behavior, elements contain models, and publication and subscription connect them through addresses. Network membership alone does not configure useful application behavior.

Group M — Long-range & specialty wireless

Chapter 98 — LoRa

- **Chips:** Semtech SX1278 (legacy, 433/868/915 MHz); SX1262 (current); LLCC68 (cheap SX1262 alternative); EByte E22 modules.
- The kernel `sx127x` driver and user-space LoRa stacks
- LoRaWAN vs LoRa-P2P; ChirpStack vs The Things Network
- Spreading factor, bandwidth, coding rate trade-offs
- **Pitfalls:** frequency-band regulation per region; antenna SWR
- **Key Notes:** LoRa is a modulation and packet-radio mechanism, while LoRaWAN adds network roles, joining, security and regional behavior. A point-to-point radio is not a multichannel gateway.

Chapter 99 — Sub-GHz proprietary

- **Chips:** Nordic nRF24L01 (2.4 GHz proprietary, 250 kbps – 2 Mbps); TI CC1101 (sub-GHz multi-band); CC1200.
- When nRF24L01 beats BLE (raw throughput, latency, multi-PRX one-PTX hub)
- CC1101 for sub-GHz remote controls and “smart” home devices

- **Pitfalls:** crystal-frequency mismatch tank-circuit issues; ack-timing in nRF24
- **Key Notes:** These radios implement different state machines and packet services. The nRF24L01+ is a 2.4 GHz device, while CC1101 represents the sub-GHz path.

Chapter 100 — ZigBee / Thread / 802.15.4

- **Chips:** TI CC2530 (legacy ZigBee, Z-Stack); Nordic nRF52840 (modern, OpenThread); Silicon Labs EFR32MG.
- ZigBee / Thread / 802.15.4 mesh tech compared
- The CC2530 as ZNP (ZigBee Network Processor) → Linux runs zigbee2mqtt
- nRF52840 OpenThread Border Router for the Matter ecosystem
- **Pitfalls:** Thread certification and credential setup; ZigBee profile compatibility
- **Key Notes:** IEEE 802.15.4 is a radio/MAC foundation, not a synonym for Zigbee, Thread or Matter. Shared radio hardware does not make those networks interchangeable.

Chapter 101 — UWB ranging

- **Chips:** Qorvo (Decawave) DWM1000; DWM3000 (newer, AirTag-compatible UWB IC); NXP NCJ29D5.
- Time-of-flight ranging principle; centimeter-accuracy indoor positioning
- Two-way ranging vs TDoA
- iPhone / AirTag UWB ecosystem and FiRa standard
- **Pitfalls:** antenna delay calibration; multipath in real environments
- **Key Notes:** UWB ranging depends on radio-captured transmit and receive times, not Linux’s time to service an interrupt. Host scheduling still limits how quickly exchanges can be prepared.

Group N — Cellular

Chapter 102 — USB 4G LTE modems

- **Modules:** Quectel EC20, EC25; SimCom SIM7600; Telit LM940.
- QMI vs MBIM vs RNDIS modes (how the modem appears to Linux)
- ModemManager + nm-connection
- Multi-band, LTE Cat-4 vs Cat-6 vs CA

- **Pitfalls:** USB power draw of LTE modems (>2A on TX); APN-vs-SIM mismatch
- **Key Notes:** A cellular modem exposes separate control, data and sometimes positioning interfaces. Their Linux device nodes represent different functions of the same system.

Chapter 103 — UART AT-command modems

- **Modules:** SIMCom A7670C, SIM7600 (UART variants); Air724UG; ML302.
- The PPP+chat approach for legacy modems
- AT-command discovery and capabilities probing
- The 2G/3G legacy chains
- **Pitfalls:** chat script timing; carrier-specific AT command quirks
- **Key Notes:** PPP divides responsibility between modem transport, user-space negotiation and the kernel network interface. A UART stream becomes a link only after that negotiation succeeds.

Chapter 104 — NB-IoT / Cat-M1

- **Modules:** Quectel BC95, BC26; SimCom SIM7080G.
- Low-power cellular IoT — when it makes sense vs LoRa vs WiFi
- AT command flows for ~30 mA RX, sub-mA PSM/eDRX sleep
- MQTT and CoAP profiles for NB-IoT
- **Pitfalls:** carrier band availability (NB-IoT not universal); PSM wake-up latency
- **Key Notes:** NB-IoT and LTE-M offer different coverage, latency and mobility tradeoffs. The selected module and operator must support the intended service.

Group O — Identification

Chapter 105 — RFID / NFC

- **Chips:** NXP MFRC522 (SPI 13.56 MHz, ubiquitous Arduino-clone); PN532 (I²C/SPI/UART, NFC-A/B/F); NCV6 series (HF-only).
- The MFRC522 register-level dance
- libnfc + neard for high-level NFC
- Mifare Classic vs DESFire — security tier comparison
- **Pitfalls:** Mifare Classic broken-by-default cryptanalysis; NFC reading-range limits

- **Key Notes:** Reader transport, tag protocol and application authorization are separate layers. Reading a tag UID identifies a presented value, not proof of an authorized person.

Chapter 106 — Fingerprint sensors

- **Modules:** Grow R503 (capacitive, UART); FPM10A (optical, UART); GT-521F (capacitive); AS608.
- UART command protocols; template enrollment and matching
- libfprint for direct USB fingerprint scanners
- Privacy implications: template storage and revocation
- **Pitfalls:** sensor warm-up on first scan; template-format incompatibility across vendors
- **Key Notes:** The module owns image capture and matching, while the host owns enrollment authority, user identity and the action permitted after a result.

Group P — Positioning

Chapter 107 — GPS / GNSS + PPS time synchronization

- **Modules:** u-blox NEO-6M / 8M / 9M (multi-GNSS); ATGM336H (cheap Chinese alternative); SiRFStar V; Quectel L86.
- NMEA-0183 parsing; UBX binary protocol for u-blox
- gpsd + chrony for sub-microsecond PPS time sync
- A stratum-1 NTP server from a Pi-class device + GPS
- **Pitfalls:** cold/warm start times; multipath in urban canyons; PPS pulse polarity
- **Key Notes:** GNSS messages identify time and position, while PPS provides a timing edge. Combining them requires associating that edge with the correct second and validity state.

Group Q — Industrial buses

Chapter 108 — RS-485 + Modbus RTU

- **Transceivers:** Maxim MAX485, SP3485; Analog Devices ADM2483 (isolated); MAX13487 (auto-direction).
- RS-485 physical layer: differential pair, biasing, termination, fail-safe

- DE/RE direction control via GPIO or auto-direction transceivers
- Linux RS-485 mode in the UART driver (`SER_RS485_ENABLED`)
- Modbus RTU framing and libmodbus
- **Pitfalls:** termination resistor placement; ground reference between segments
- **Key Notes:** RS-485 defines an electrical link, while Modbus RTU defines framing and operations above it. Good electrical signaling does not establish correct register semantics.

Chapter 109 — LIN bus

- **Transceivers:** NXP TJA1020, TJA1027; Microchip MCP2003B.
- LIN as a “UART with break and checksum” sub-CAN bus
- Master/slave roles; PID, response, classic checksum vs enhanced
- Linux’s lack of native LIN — userspace SLIP-style or custom driver
- **Pitfalls:** LIN frame timing tolerance; cable-length-vs-baud trade-offs
- **Key Notes:** LIN uses one schedule owner to issue headers and coordinate responses. Adding another sender without that role agreement creates collisions.

Chapter 110 — CAN deep dive

- (Extension of Ch 55C) **Transceivers:** TJA1051 (5 V), TJA1463 (CAN-FD), MCP2562; SPI-CAN bridge MCP2515 (when SoC has no FlexCAN free).
- CAN-FD vs classic CAN; bit-stuffing and frame-rate trade-offs
- ISO-TP for ISO-15765 diagnostics
- SocketCAN advanced: BCM (broadcast manager), CAN-J1939
- **Pitfalls:** termination 60 Ω vs 120 Ω; differential signal-integrity at 1 Mbit/s
- **Key Notes:** CAN controller capability, transceiver capability and network protocol are separate choices. A CAN-FD-capable transceiver cannot turn the i.MX6ULL’s classic FlexCAN controller into a CAN-FD controller.

Group R — Motors & encoders

Chapter 111 — Quadrature encoders & rotary

- **Devices:** incremental optical encoders; magnetic encoders; the `rotary_encoder` driver and `gpio-keys` for low-res.

- Hardware quadrature decode on i.MX6ULL's eXtended quadrature decoder (XBAR + ENC)
- A 2-pin GPIO quadrature decode in software (and why it doesn't work at high speed)
- IIO `angl` channel
- **Pitfalls:** Z-index pulse handling; debouncing mechanical encoders
- **Key Notes:** Quadrature encoders report relative transitions, while absolute encoders report a position within their defined range. Homing and accumulated turns are different state problems.

Chapter 112 — Stepper & DC motor drivers

- **Drivers:** TI DRV8825 (stepper, micro-stepping); Allegro A4988; Trinamic TMC2209 (silent stepper with UART config); BTS7960 (43 A H-bridge DC); DRV8302 (BLDC).
- Step-Dir control via PWM/GPIO with a kernel `pwm` driver
- TMC2209 UART config of microstep, RMS current, stallGuard
- BLDC commutation: trapezoidal vs sinusoidal vs FOC
- **Pitfalls:** missed steps from EMI; back-EMF damage to drivers
- **Key Notes:** Stepper pulse generation, brushed-motor PWM and BLDC commutation have different timing contracts. Linux can supervise a system without owning every fast control loop.

Group S — Indicators & smart LEDs

Chapter 113 — WS2812 / SK6812 / APA102 addressable LEDs

- **Chips:** WS2812B (timing-strict 1-wire); SK6812 (similar, RGBW variant); APA102 (SPI-based, no timing issues).
- Three implementation approaches:
  1. **GPIO bit-bang** with PREEMPT_RT — only viable for short strips
  2. **PWM + DMA** — repurpose PWM for serial bitstream
  3. **SPI + DMA** — encode WS2812 timing in SPI bytes; cheap and robust
- APA102's SPI native protocol; per-pixel global brightness
- **Pitfalls:** 5 V level shifting from 3.3 V GPIOs; long-strip data-line integrity

- **Key Notes:** Pulse-width-coded LEDs need a continuous waveform with the selected part’s timing. Encoding into a hardware-driven stream moves that responsibility away from scheduled bit toggles.

Chapter 114 — Beepers, relays, SSRs

- Passive beepers via PWM; active beepers via GPIO
- Mechanical relays vs MOSFETs vs SSRs — when to use which
- AC SSR safety: opto-isolation, zero-cross switching, snubber circuits
- **Pitfalls:** relay-coil flyback; SSR sub-cycle leakage
- **Key Notes:** Buzzer, relay, MOSFET and SSR interfaces differ in load behavior and protection requirements. A GPIO can request a state without safely supplying the load.

Group T — Network & system power

Chapter 115 — Dual FEC + hosted Ethernet

- Both i.MX6ULL FEC1 + FEC2 simultaneously: pin-mux, PHY-per-MAC, sep subnets
- IP forwarding, bridging, and bonding between the two
- **Chips:** WIZnet W5500 (SPI Ethernet with hardware TCP/IP); Microchip ENC28J60 (cheaper but slower).
- Adding a 3rd Ethernet via SPI when you need 3+ network interfaces
- **Pitfalls:** clock skew between PHYs; W5500 socket exhaustion
- **Key Notes:** Multiple network interfaces provide distinct links, while routing, bridging and segmentation define how traffic crosses between them.

Chapter 116 — PMICs and regulator framework

- **Chips:** NXP PCA9450 (i.MX-recommended PMIC); PF8200; Rohm BD71850MWV.
- The `regulator` framework: consumer/provider model
- Voltage sequencing for SoC + DDR + I/O rails
- Dynamic voltage scaling integration with DVFS (Ch 51B)
- **Pitfalls:** boot-sequence races (kernel boots before PMIC’s voltage settles); thermal throttling
- **Key Notes:** Regulator providers describe rails and consumers request the supplies they need. This dependency graph should match the physical power tree.

Chapter 117 — External RTC

- **Chips:** Maxim DS3231 (TCXO, ±2 ppm); NXP PCF8563 (common cheap); Microchip MCP79410 (with EEPROM).
- Why crystal RTCs drift (and why TCXO is worth the money for fleets)
- Battery-backed time keeping; the “RTC dead but device works” failure mode
- `hwclock`, `rtc-i2c` family of drivers
- Alarm interrupts for wake-from-suspend (Ch 51B integration)
- **Pitfalls:** the i.MX6ULL internal RTC is part of SNVS and loses time without backup battery; DS3231 alarm-pin polarity
- **Key Notes:** The RTC retains time independently of the system clock, while NTP or another reference disciplines system time. Synchronization direction and validity need one clear policy.

PART VIII — DEBUG, PRODUCTION, ADVANCED

Chapter 118 — JTAG, OpenOCD, GDB at every layer

- Connecting a JTAG adapter (FT2232H / J-Link) to the Point Atom JTAG header
- OpenOCD config for i.MX6ULL
- GDB scripts for bare-metal, U-Boot (`gdb-multiarch u-boot`), and the kernel (`vmlinux` symbols)
- Hardware breakpoints, watchpoints
- **Key Notes:** JTAG provides a hardware access path when the target cannot run a software debug agent. The adapter, OpenOCD configuration, and GDB symbol files have separate responsibilities.

Chapter 119 — Kernel debugging without JTAG

- `printk` and log levels
- `pr_debug` and `CONFIG_DYNAMIC_DEBUG`
- ftrace: `function`, `function_graph`, events
- trace-cmd and KernelShark
- `bpftrace` and `bcc` tools (introductory)
- kgdb over UART

- Decoding an oops: `addr2line`, `scripts/decode_stacktrace.sh`, `CONFIG_DEBUG_INFO`
- **Key Notes:** Software observability depends on code paths that still execute. Logs and trace buffers cannot replace hardware debug access when the required kernel infrastructure is unavailable.

Chapter 120 — User-space debugging

- `gdbserver` on target, `gdb-multiarch` on host
- `strace`, `ltrace`
- `perf` (sampling, counters, flamegraphs)
- Core dumps and `coredumpctl`
- **Key Notes:** `gdbserver` controls the target process while host GDB owns symbols and source lookup. The host executable and target libraries must describe the deployed build.

Chapter 120A — Supplementary: Mainline patch submission workflow

If you write a driver in this book and it’s good, it can go upstream. We walk the entire process end-to-end on a real candidate patch (e.g., a tweak to the FEC driver, or a YAML binding for a new sensor).

- `git format-patch` and the one-patch-per-fix discipline
- `scripts/checkpatch.pl --strict`; the warnings that matter, the ones that don’t
- `scripts/get_maintainer.pl` to find the right list and the right reviewer
- `git send-email` setup (the only patch-submission tool the kernel community accepts)
- Subject-line conventions: `[PATCH] drivers/net/ethernet/freescale/fec: ...`
- Cover letters; v1 / v2 / v3 etiquette; `Reviewed-by`, `Acked-by`, `Tested-by`, `Reported-by`, `Suggested-by`, `Co-developed-by`
- Replying to review feedback — what to do, what not to do
- The Lore archive, `b4` for series management
- **Key Notes:** A patch carries both a code change and its rationale. One logical change per patch makes the affected interface and regression risk visible.

Chapter 121 — Custom board port as a system design

- Compare unchanged MINI hardware, attached-device variants and custom PCBs

- Follow ownership across early hardware, U-Boot, kernel description and rootfs
- Separate reproducible artifact production from physical deployment
- Explain failure boundaries, recovery and the evidence behind design decisions
- **Key Notes:** A board port connects fitted hardware to several software descriptions. Power and reset behavior, early DDR setup, U-Boot configuration, kernel DT, and rootfs contents have distinct owners.

Chapter 121A — Supplementary: CI/CD for embedded Linux

Modern teams build U-Boot, kernel, rootfs, and run a smoke test on the actual hardware on every commit. We set this up using GitHub Actions (or GitLab CI), a self-hosted runner with USB-OTG to a board, and a small Labgrid-style harness.

- Cross-builds in CI: caching, deterministic builds, `bitbake-no-network` patterns
- Image artifact storage (size budget: ~150 MB per build × 20 commits/day)
- A self-hosted runner with a board on a USB hub
- The minimal smoke test: boot, wait for `=>`, run a 5-second sysfs check, capture serial log
- Pass/fail signaling back to the PR
- Notifications when the board farm is offline
- **Key Notes:** Cloud runners construct cross-built artifacts and check source. A hardware runner owns the physical board and provides a different kind of evidence.

Chapter 122 — Build your own cross-toolchain

- Bootstrap problem: gcc needs libc, libc needs gcc, gcc needs binutils
- crosstool-NG step-by-step: kernel headers → binutils → gcc stage 1 → glibc/musl → gcc stage 2
- Comparing your toolchain against a pre-built Arm GNU one (size, behavior, sysroot)
- **Key Notes:** A toolchain combines code-generation tools, target runtime components, and a debugger. Its target ABI and libraries must agree with the deployed rootfs.

Chapter 122A — Supplementary: BSP → mainline migration playbook

You inherited a Linux 4.1.15 vendor BSP from a previous project or a customer. The product needs to ship updates for the next eight years. You must move to a supportable mainline kernel. Here is the playbook.

- Inventory: list every patch the vendor applied; classify (vendor-feature / vendor-bugfix / mainline-merged / dead-code)
- The pinned-driver problem and how to break each pin
- Per-subsystem upstreaming order (most-self-contained first: clk → pinctrl → gpio → i2c → … → display/network last)
- Maintaining two trees during the migration (vendor “shipping” + mainline “next”)
- Bisection across kernel versions when an old hack now breaks
- When to **not** migrate (truly captive silicon with no mainline future)
- A concrete worked example: NXP i.MX6ULL 4.1.15 → 6.x mainline
- **Key Notes:** Migration preserves required product behavior across changing interfaces, rather than replaying every vendor patch onto a newer tree.

Chapter 123 — Yocto vs Buildroot, an honest comparison

- The mental model: package metadata vs configuration system
- Layers, recipes, BBLAYERS
- When Buildroot is better; when Yocto is better; when *neither* is right
- A single recipe written for Yocto and for Buildroot, side by side
- **Key Notes:** Buildroot organizes an image around configuration and package definitions. Yocto composes recipes, layers, machines, distribution policy, and image contents.

Chapter 123A — Supplementary: Yocto layer development in depth

For shipping at scale, Yocto is industry standard, and the meat of using it is *writing layers*. We build a vendor layer (`meta-mybsp`), a board layer (`meta-mybsp-mini`), and an application layer (`meta-mybsp-myapp`), with the right bbappend pattern between them.

- Layer anatomy: `conf/`, `recipes-*/`, `classes/`, `wic/`
- `bbappend` and the layer-priority dance
- Writing a `.bb` for our LED-driver kernel module (Chapter 41)
- Writing a `.bb` for an in-house Qt app

- `wic` for image layouts (partition tables, RAUC slots)
- Distro layers (`meta-mybsp-distro`) vs board layers
- `IMAGE_FEATURES`, `EXTRA_IMAGE_FEATURES`, `DISTRO_FEATURES`
- The `SRC_URI` cache that makes the build reproducible without internet
- **Key Notes:** `MACHINE` describes hardware, `DISTRO` expresses shared policy, and image recipes select shipped contents. Separating these responsibilities limits variant leakage.

Chapter 124 — Secure boot (HAB) and OP-TEE

- The chain of trust: ROM → SRK fuses → CSF → signed U-Boot → signed kernel → dm-verity rootfs
- HAB CST (Code Signing Tool), `csf` files, key ceremony
- TrustZone primer, `monitor` mode, SMC calls
- OP-TEE basics: TA (Trusted Application) lifecycle
- **Key Notes:** HAB anchors bootloader authentication in ROM and a provisioned key hash. FIT verification, rootfs integrity, and module policy extend trust only when their inputs are protected.

Chapter 125 — Field updates

- A/B partition scheme with U-Boot
- RAUC, SWUpdate, Mender — comparison
- Atomic updates, rollback, fail-safe boot
- **Key Notes:** An A/B update preserves the running slot while constructing a candidate in the inactive slot. Completion must precede changing persistent boot selection.

Chapter 125A — Supplementary: VSCode + gdbserver remote-debug workflow

Many readers came here from VSCode and would rather debug there than learn `tui` mode. We set up `gdbserver` on the target, `gdb-multiarch` on the host, the VSCode `launch.json` that joins them, and the `.vscode/c_cpp_properties.json` that resolves headers from the cross-sysroot — so `Go to Definition` works on kernel sources too.

- Source Insight as a faster alternative for kernel-source navigation only (read-only, but very fast)
- The minimum target setup: a statically-linked `gdbserver` binary in `/usr/bin/`

- Single-stepping a kernel module loaded with `insmod`
- **Key Notes:** VSCode presents debugger state but does not implement target debugging. Host GDB and target `gdbserver` retain their command-line responsibilities.

Chapter 126 — Closing: what to read next

- LDD3 (still relevant where it isn’t outdated)
- Bootlin training material
- `kernelnewbies.org`
- LWN — the single most useful periodical for a Linux engineer
- The mailing list etiquette guide; how to send your first patch
- **Key Notes:** Every layer needs an owner and a contract: what it receives, what it changes, what it promises, and how failure is reported.

PART IX — APPLIED VIRTUALIZATION AND MIXED-CRITICALITY SYSTEMS

> *This part is optional and advanced. It exists for readers who want real hypervisor experiments, not only HYP-mode vocabulary. The path starts in QEMU, moves to Xen on the i.MX6ULL, then uses Jailhouse and STM32MP1 where the hardware model is a better fit.*

Chapter 127 — Why embedded products use hypervisors

- The real product problems: crash isolation, security boundaries, mixed criticality, legacy BSP containment, and device ownership
- Containers vs PREEMPT_RT vs TrustZone vs remoteproc/OpenAMP vs Xen vs Jailhouse
- Why a hypervisor is sometimes the clean answer and sometimes architectural overkill
- Match the required failure boundary to the ownership model before selecting tools
- **Key Notes:** The choice begins with a failure to contain and the workload that must survive it. A process, container, guest, partition or companion MCU provides a different boundary.

Chapter 128 — QEMU as a virtual hardware model

- QEMU as a repeatable ARM hardware model, not an i.MX6ULL clone

- ARMv7-A `virt`, ARM64 `virt`, and when `vexpress` is useful
- Serial console, memory size, CPU model, DTB, kernel, initrd, and command line as explicit hardware choices
- GDB with `-S -s` before Linux prints anything
- **Key Notes:** A QEMU launch describes a virtual machine: CPU, memory, interrupt controller, devices and firmware are explicit inputs. Its generated DTB describes that selected model.

Chapter 129 — Build and boot tiny Linux in QEMU

- Build a tiny ARM Linux kernel and BusyBox/Buildroot initramfs
- Direct boot: `QEMU → Linux → initramfs → /init`
- DTB, `console=`, `rdinit=`, and the first shell
- Failure-driven learning: wrong console, missing init, wrong DTB
- **Key Notes:** The kernel, Device Tree and initramfs are separate boot inputs. They provide executable kernel code, a hardware description and the initial user-space tree respectively.

Chapter 130 — U-Boot in QEMU

- Put U-Boot in front of the QEMU Linux boot
- Manual `bootz` / `booti`: load kernel, DTB, initramfs into non-overlapping RAM
- `bootargs`, scripted boot, and FIT handoff
- Why hypervisors depend on firmware doing the next-stage handoff correctly
- **Key Notes:** U-Boot owns artifact loading and the next-stage handoff. The kernel owns interpreting its DTB, arguments and initial filesystem after that handoff.

Chapter 131 — HYP mode, stage-2 MMU, and virtual interrupts

- PL0, PL1, PL2, HYP mode, `svc`, and `hvc`
- HYP mode vs TrustZone Monitor mode
- Stage-1 translation: virtual → physical
- Stage-2 translation: guest physical → real physical
- Virtual timer, virtual GIC, and why guests need believable interrupts
- **Key Notes:** Stage 1 belongs to the guest operating system when it uses translation. Stage 2 gives the hypervisor a separate CPU memory boundary.

Chapter 132 — Xen in QEMU

- Boot path: `QEMU → Xen → Dom0 Linux`
- Xen binary, Dom0 kernel, initramfs/rootfs, DTB, and console
- `xl info`, memory allocation, and where control moves from Xen to Dom0
- Keeping the first Xen boot small and serial-only
- **Key Notes:** Xen takes control before Dom0 Linux. Dom0 is a privileged domain with platform responsibilities, not native Linux outside the hypervisor.

Chapter 133 — First DomU Linux

- Boot path: `Xen → Dom0 → xl create → DomU`
- Domain config explained line by line: name, kernel, ramdisk, memory, vCPUs, console, command line
- Tiny BusyBox/Buildroot guest first, not a full distribution
- Failure isolation: DomU can die while Dom0 survives
- **Key Notes:** A DomU is another kernel with assigned memory, virtual CPUs and interfaces. It is not an ordinary process in Dom0’s PID namespace.

Chapter 134 — Xen on i.MX6ULL

- Moving from QEMU to the real board
- i.MX6ULL reality: one Cortex-A7, limited RAM, real DTB, real UART, real U-Boot handoff
- Build Xen for ARM32 and boot Dom0 Linux on the Point Atom board
- Confirm that the board demonstrates HYP-mode mechanics but is not a rich virtualization platform
- **Key Notes:** The physical boot path adds U-Boot and MINI-specific hardware dependencies before Xen can start. QEMU success cannot supply evidence for those dependencies.

Chapter 135 — DomU Linux on i.MX6ULL

- A tiny DomU on the real board: one vCPU, small RAM, initramfs, console only
- `xl list`, DomU console, `uname -a` in both worlds
- What this proves: real guest isolation

- What it does not prove: performance, device pass-through safety, or hard real-time behavior
- **Key Notes:** Dom0 and DomU have separate kernel and memory views while sharing the i.MX6ULL's one physical CPU. Scheduling capacity and failure containment are different objectives.

Chapter 136 — Devices, memory, and DMA boundaries

- Device ownership tables: owner, shared/not shared, DMA, reset, clock, pinctrl, debug
- Why the CPU MMU does not automatically protect against DMA
- UART, Ethernet, GPIO bank, SPI/I2C, timers, and interrupt ownership
- Xen grant tables, event channels, virtual network, shared memory
- **Key Notes:** Every hardware resource needs one control owner, including its configuration, reset and recovery path. Sharing requires a defined mediation interface.

Chapter 137 — Jailhouse in QEMU ARM64

- Why Jailhouse is taught on QEMU ARM64 first: multiple CPUs and a known demo platform
- Boot path: `QEMU ARM64 → Linux root cell → jailhouse.ko → system cell → inmate cell`
- Root cell vs inmate cell, CPU assignment, memory regions, interrupts, and no scheduler
- Why i.MX6ULL is a poor Jailhouse target even though Cortex-A7 can implement virtualization extensions
- **Key Notes:** Jailhouse partitions resources after Linux has booted. It does not provide Xen's general domain-management model.

Chapter 138 — Bare-metal or Zephyr inmate cell

- Start with a minimal bare-metal inmate before adding Zephyr
- Link address, assigned RAM, console ownership, timer, and stop/reload path
- Zephyr as the RTOS-flavored inmate once the bare-metal path is understood
- Shared memory and interrupt discipline
- **Key Notes:** Cell configuration, link address, entry point and stack form one runtime contract. They must agree before a C function can execute.

Chapter 139 — STM32MP1 Linux plus RTOS, the production pattern

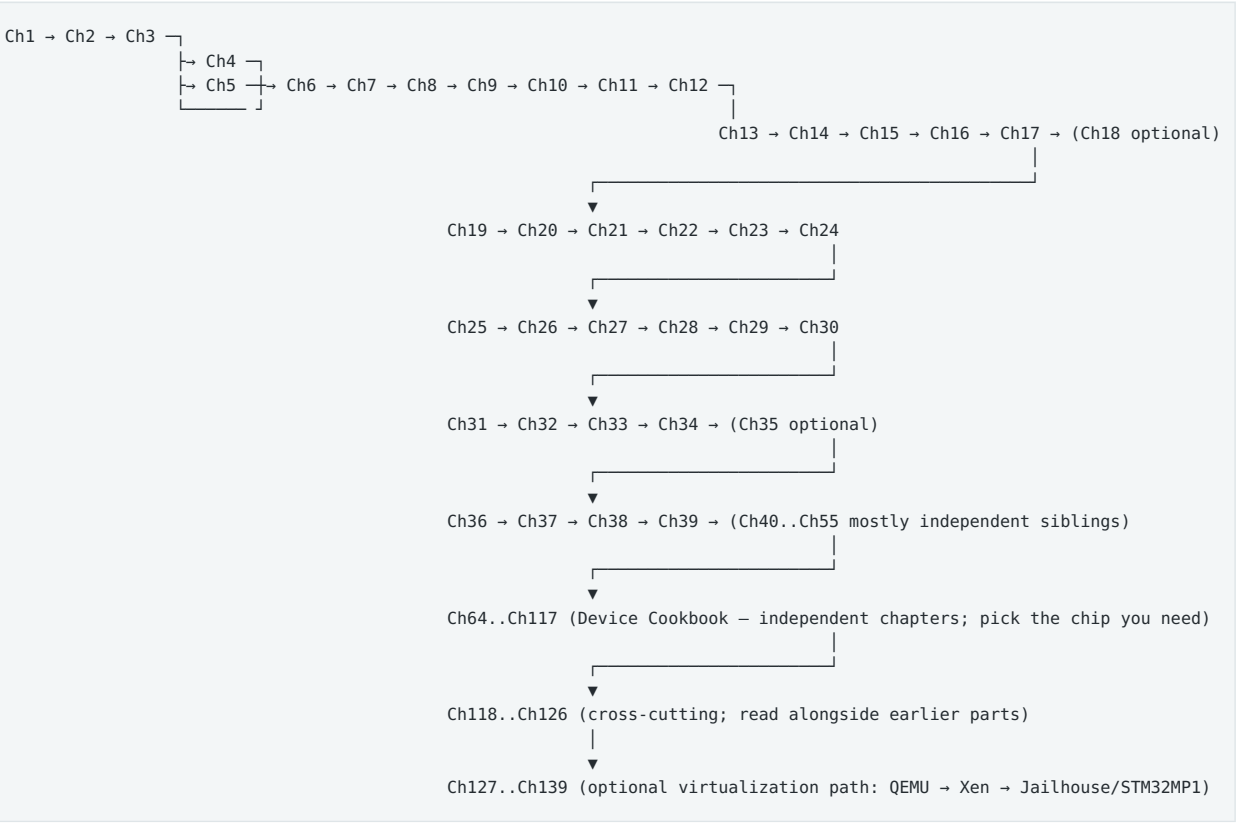
- Why STM32MP157 is a better Linux+RTOS teaching target than i.MX6ULL: dual A7 plus M4
- Main production path: Linux on Cortex-A7, Zephyr/FreeRTOS on Cortex-M4, RPMsg/OpenAMP between them
- Optional Jailhouse path for A7-to-A7 partitioning when the M4 is not enough
- Decision checklist: Xen vs Jailhouse vs remoteproc/OpenAMP vs no hypervisor
- **Key Notes:** Remoteproc manages a remote processor’s firmware and lifecycle, RPMsg carries messages, and OpenAMP supplies software for those mechanisms. They are related but not interchangeable responsibilities.

Chapter coverage

Part	Numbered	Supplementary	Total
I — Foundations	8	0	8
II — Bare-metal i.MX6ULL	10	3	13
III — U-Boot	6	10	16
IV — Kernel	6	6	12
V — Rootfs & user space	5	9	14
VI — Driver development	20	14	34
VII — Device cookbook	54	0	54
VIII — Debug & advanced	9	5	14

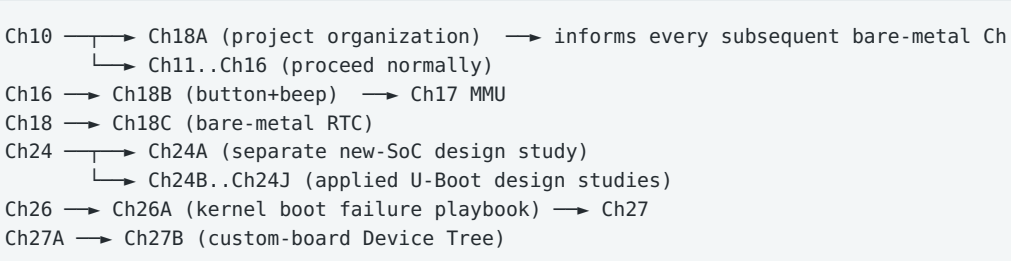
Part	Numbered	Supplementary	Total
IX — Applied virtualization	13	0	13
Total	**131**	**47**	**178**

Dependency graph (so the reader can prune)



The arrows describe conceptual dependencies, not required builds. A kernel or driver reader can revisit the bare-metal mechanism when an address, clock or interrupt assumption needs explanation. The device chapters and advanced design studies are references, not a sequence of assignments.

Where the supplementary chapters fit



```
Ch29 ──► Ch29A (initramfs recovery system)
Ch30A ──► Ch30B (product kernel configs)
Ch35 ──► Ch35D (bootable SD/eMMC layout) ──► Ch35I (boot from SPI NOR)
Ch35I ──► Ch35E (Buildroot board directory)
Ch35E ──► Ch35A (Ubuntu-base, optional alternative) ──► Ch35F (app as service)
Ch35F ──► Ch35G (network/time/logging) ──► Ch35B (read-only rootfs)
Ch35B ──► Ch35H (rootfs security basics) ──► Ch35C (containers)
Ch43 ──► any of Ch55A..Ch55H (siblings, independent)
Ch44..Ch55I ──► any chapter of Part VII Device Cookbook (Ch 64..Ch 117) — pick the chip class you need
Ch120 ──► Ch125A (VSCode workflow; can be read after any driver chapter)
Ch126 ──► Ch127..Ch139 (optional virtualization and mixed-criticality path)
```

Project decisions

1. **Target board.** Point Atom MINI with a revision-specific 512 MiB DDR3L core-board example is the primary reference. Clock limits come from the actual chip marking and electrical conditions, not the board name. Parts IV–VIII teach concepts applicable to other Linux-capable ARM SoCs, but their board configuration and drivers still need porting.

2. **Language.** English only. No bilingual edition planned.

3. **Source baselines.** Linux **v6.6** API conventions; the supplied MINI vendor U-Boot **2016.03** tree for the board baseline and **v2026.04** for the separate mainline study; two Arm GNU **13.2.Rel1** toolchains. Later Buildroot and Yocto chapters name their own source versions. Do not silently mix board support, generated artifacts or API signatures from different trees.

4. **Code-listing license.** MIT. Reuse does not establish that a snippet is complete, safe for a particular board, or compatible with another source version.

5. **Tooling.** Markdown sources + **Sphinx + Furo + MyST-Parser**, hosted on GitHub Pages and auto-rebuilt on push. The reading path begins at Chapter 1. Publishing instructions live in the repository's `PUBLISH.md`.

6. **Examples. Inline in the chapters.** There is no companion `code/` repository. Listings illustrate mechanisms and interface contracts rather than assigned projects. Architecture review and physical qualification are separate responsibilities, even when implementation is generated by a tool.

Status

The full manuscript is present in this repository. Its reading model is architecture-first: chapter-specific Key Notes replace assigned labs, and code illustrates mechanisms and interfaces. Removing assignments does not turn an implementation example into a qualified system.

Part	Chapters	Current state
Part I — Foundations	1–8	Drafted
Part II — Bare-metal i.MX6ULL	9–18 + 18A–18C	Drafted
Part III — U-Boot, deeply	19–24 + 24A–24J	Drafted
Part IV — The Kernel	25–30 + 26A, 27A, 27B, 29A, 30A, 30B	Drafted
Part V — Root filesystem & user space	31–35 + 35A–35I	Drafted
Part VI — Driver development	36–55 + 51A, 51B, 52A, 54A, 54B, 55A–55I	Drafted
Part VII — Device cookbook	64–117	Drafted
Part VIII — Debug, production, advanced	118–126 + 120A, 121A, 122A, 123A, 125A	Drafted
Part IX — Applied virtualization and mixed-criticality systems	127–139	Drafted, virtualization and mixed-core design studies

Parts I-III have received technical/editorial review and source/build checks recorded in the project reports. Physical MINI qualification remains unperformed. Later parts still need their own detailed technical review. Commands, register tables and production/security flows remain version- and board-dependent examples, not deployment instructions validated by this editorial change.

Start with Chapter 1 for the reading order; the full table of contents gives the full scope.