VITA 46.11 Tier-1 IPMC Software Specification

IPMIRustFRU: Rust IPMC firmware for the NUCLEO-F401RE, x86 emulator and bench tools

Revision 11-21-2026

Quick Start

1. Host tests and the emulator, from the repository root (Rust 1.99.1 from `rust-toolchain.toml`, GNU host): set `$env:IPMC_TIER23 = "0"` in PowerShell for the public Tier-1 build, then `cargo test --workspace`, followed by `cargo run -p ipmc-emu -- replay local_only/vectors/sbc_ab.vec --b-fitted 1 --b-enabled 1 --policy alternate --quiet` (the capture file is local only; on a fresh clone the capture tests skip with a message).
2. Firmware build and flash (NUCLEO-F401RE on USB): `powershell -ExecutionPolicy Bypass -File crates/ipmc-nucleo-f401re/gen.ps1 flash`. `flash` builds the bootloader (sectors 0-1) and the runtime (from sector 3) and copies the board image `ipmc-board.bin` to the drive; `flash -Swd` programs the patched runtime ELF over SWD instead (section 12.1; the two paths treat the FRU store differently, section 9.1). Select the actual ST-LINK virtual COM port at 115200 8N1 (COM4 on the current bench). `gen.ps1 console -ComPort COM4 "sys status"` sends a command and prints the reply; `-ComPort` defaults to COM3 for older bench setups.
3. Bench check with an independent IPMI implementation (Aardvark on IPMB-A, PB8/PB9): `tools\python-ipmi\.venv\Scripts\ipmitool.exe -I aardvark -t 0x82 raw 0x06 0x01` answers `00 01 80 00 01 02 09 00 00 00 4b 49 00 00 00 00`; `python tools/board_test_report.py --com-port COM4` runs every bench check and writes `local_only/Documentation/board_test_report.md`.
4. Capture log and application port: `DEBUG DUMP` on the console prints the IPMB capture log (section 10.11); `python -I tools\capture\capture2pcap.py dump.txt -o dump` turns a saved dump into a file Wireshark opens. A USB-serial adapter on PB6/PB7 (COM5 on the bench) reaches the application port (section 7.6): `tools\python-ipmi\.venv\Scripts\python.exe tools\app-port\app_cli.py --port COM5 INFO STATUS`.
5. Firmware update over the console CLI or application UART is included in the public Tier-1 release (section 16.5). For the onboard ST-LINK console, run the COM4 CLI procedure there. For APP, `gen.ps1 build` writes the patched runtime `ipmc-nucleo-f401re.hex`; `python tools\fwupdate\fwupdate.py --port COM5 --enter app --enter-port COM5 update ipmc-nucleo-f401re.hex` asks the running firmware for the bootloader (`BOOTLOADER CONFIRM` on the application port), erases the runtime sectors, loads the file record by record, compares the SHA-256 the bootloader reports with the file and starts the new runtime.

Table of Contents

1. Overview
2. System Context
3. Requirements
4. Architecture
5. Build-Time Configuration
6. Module Descriptions
7. Interfaces
8. Clocking, Reset and Start-Up
9. Memory Map and Data Layouts
10. Data Flow and Protocol
11. Software API
12. Build, Integration, and Configuration
13. Verification

14. Example Sequences
15. Bootloader
16. Firmware Update and Image Integrity
17. Constraints and Limitations

The section 10 diagrams are Mermaid state and sequence diagrams and one WaveDrom timing diagram; the block drawings are the seven sheets of the `ipmc-software` drawing set (`Documentation/diagrams/src/ipmc-software.js`, rendered to `Documentation/diagrams/out/png/`): sheet 1 crate structure (Figure 4-1), sheet 2 request path (Figure 10-1), sheet 3 sensor data, FRU state and persistence (Figure 10-2), sheet 4 build, test and bench setup (Figure 13-1), sheet 5 flash map and images (Figure 9-1), sheet 6 firmware update data path (Figure 16-1), sheet 7 payload processor integration (Figure 7-1). The Google Slides reference copies wait for the Drive folder (`Documentation/diagrams/drive-config.json` holds `folder_id` TBD); until it is set, the source and the rendered sheets are the drawing set.

1. Overview

IPMIRustFRU is an Intelligent Platform Management Controller (IPMC) for a VPX plug-in module, written in Rust to ANSI/VITA 46.11-2022 Tier 1. It runs on an STM32F401RE (NUCLEO-F401RE board), answers a Chassis Manager over the redundant Intelligent Platform Management Bus (IPMB-A and IPMB-B, I2C at 100 kHz; 400 kHz FAST-MODE at Tier-3), serves the FRU 0 inventory and the Device SDR repository, reports the seven mandatory VITA sensors and keeps Write FRU Data in on-chip flash across a reset. The same IPMI code builds for the PC as an emulator that replays captured Chassis Manager traffic or serves a live Chassis Manager over TCP. This document explains how the code works, crate by crate and module by module, for a software engineer who maintains or extends it.

The primary function is a Tier-1 IPMC: the 13 Tier-1 mandatory commands plus Broadcast Get Device ID, the 7 mandatory sensors, FRU 0 with Common Header, Board Info and Product Info areas, the FRU 0 Operational State machine (M0, M1, M4, M6; M5 at Tier-3), dual-bus link rules (alternation, resend on the other bus, Set IPMB State), a startup self test and NVMRO write protection.

The firmware runs on the STM32F401RE with the bit-bang IPMB master (default) or the hardware I2C master. The x86 emulator offers capture replay (`replay`), real-time TCP serving (`serve`) and an SDR dump (`sdr`).

The firmware has two serial ports, both interrupt driven. The console (USART2, the ST-LINK virtual COM port) carries the command line, including the DEBUG group: the IPMB capture log records every whole frame received and sent and every trace line, and the console dumps it as CSV or as trace rows for a spreadsheet or Wireshark (section 10.11). The application port (APP_UART, USART1 on PB6 and PB7) lets a payload CPU on the same board ask for status, run IPMI requests locally and write the per-unit FRU fields (serial numbers, part numbers, asset tag, manufacturing date); its protocol is the `ipmc-app` crate (section 7.6).

The firmware is two images in one flash. The bootloader in sectors 0 and 1 starts first after every reset, checks the runtime firmware against the length and SHA-256 digest of its image header, and starts it, or stays and serves a command session on the console or the application port, where the runtime can be erased and loaded again as Intel HEX records (sections 15 and 16). The runtime firmware in sectors 3 to 7 is the IPMC described in the other sections; sector 2 holds the FRU flash store.

The public build is the Tier-1 IPMC described here. A commercial release build adds a private Tier-2/3 extension through the hook interface of section 4 (Build modes and the Tier-2/3 extension). The public build answers every command the Tier-1 table does not own C1h; section 7.2 lists the commands VITA 46.11 requires at Tier 2 and Tier 3 and the commands the commercial build answers; its tests are in `Documentation/ipmc_test_coverage.md` (section "Tier-2/3 extension").

The public build leaves out the Tier-2 and Tier-3 functions (events, payload power control by the Chassis Manager, FRU activation policy; they belong to the commercial release build), optional Tier-1 commands (Get Mandatory Sensor Numbers, FRU Control Capabilities, Get IPMB Link Info, Get FRU Hash, Get Device GUID, Set SEL Time: all answer C1h), subsidiary FRUs, and IPMI bridging. The built FRU image has no Internal Use area; `verify` still checks that area when a written image contains one. Its MultiRecord area holds one record, the VITA 46.11 Alternate Manufacturing Date/Time record.

The code is the authority for every statement here. Where the VITA 46.11-2022 text and the captured live Chassis Manager disagree, the code follows the specification.

2. System Context

- **Target device:** STM32F401RET6 (Cortex-M4F, 512 KB flash, 96 KB RAM) on the NUCLEO-F401RE, 84 MHz system clock from the 8 MHz ST-LINK MCO clock in HSE bypass.
- **Operating environment:** bare metal, `no_std`, one cooperative main loop plus interrupt handlers that move bytes, stamp times and drive the LED (the I2C slaves, the USART1 and USART2 rings, SysTick). No RTOS, no heap.
- **Host build:** `x86_64-pc-windows-gnu` (the same sources build for `x86_64-unknown-linux-gnu`); the emulator is a `std` binary.
- **Dependencies:** `cortex-m 0.7.10`, `cortex-m-rt 0.7.7`, `stm32f4 0.16.0` (PAC, feature `stm32f401`, firmware crate only), `heapless 0.8.0` (SPSC queues), `embedded-hal 1.0.0` (bus traits of the example external sensor drivers). Every version is an exact `=` requirement.
- **Peers on the IPMB:** a Chassis Manager at 8-bit address 20h (`IPMB_CM_ADDR8`; the captured live Chassis Manager uses 10h). The IPMC answers at (40h + GA[4:0]) x 2: slot 1 is 82h.
- **Bench:** an Aardvark I2C host adapter on IPMB-A with its pull-ups on, driven by python-ipmi as the Chassis Manager; the ST-LINK USB port carries the console (COM4 on the current bench) and the drag-and-drop flash drive (`NOD_F401RE`); a USB-serial adapter (COM5) on PB6 and PB7 carries the application port (`tools/app-port/app_cli.py`).
- **Normative documents:** ANSI/VITA 46.11-2022 (local, licensed, not in the repository), IPMI v2.0 rev 1.1, IPMI Platform Management FRU Information Storage Definition v1.0 rev 1.3, RM0368 (STM32F401 reference manual), DS10086 (datasheet), UM10204 (I2C-bus specification).

Term	Meaning
IPMB	Intelligent Platform Management Bus: I2C with IPMI framing and two checksums
rsSA / rqSA	responder / requester slave address, 8-bit form (7-bit address shifted left, R/W bit 0)
NetFn	network function; even for a request, odd (request + 1) for its response
VSO	VITA Standards Organization; its Group Extension defining-body byte is 03h
SDR	Sensor Data Record; the Device SDR repository lives in the IPMC
FRU	Field Replaceable Unit; FRU 0 is the module that holds the IPMC
GA, GAP	geographic address straps GA[4:0] and their odd-parity strap
NVMRO	non-volatile memory read-only strap; asserted, the IPMC writes no non-volatile memory
T1..T9, C1	VITA 46.11 Table 4.1.1-1 timing and retry parameters
APP_UART	the application UART (USART1) between the IPMC and the payload CPU

3. Requirements

Functional requirements

ID	Requirement	Rule	Where
FR-001	Answer the Tier-1 mandatory commands of Tables 10.1.1-1 and 10.1.2-1; every other command answers C1h	Rules 4.2-2, 4.1.1-16	`ipmc-core/src/commands.rs`
FR-002	Answer Broadcast Get Device ID (general-call 00h before the frame) at the own address only	IPMI 20.9	`link.rs` `on_frame`
FR-003	FRU Device ID other than 0 answers C9h (Get FRU Address Info ignores the ID when a physical address key is given)	Rules 4.2-3, 4.1.1-17	`cmd/mod.rs` `check_fru_device_id`
FR-004	Power-up bus state: IPMB-A enabled, IPMB-B isolated until Set IPMB State; with `IPMB_COMPLIANCE` `NonStrict` (default) a request that arrives on IPMB-B before the first Set IPMB State that sets IPMB-B is answered on IPMB-B and the bus state does not change, with `Strict` IPMB-B drops its frames	Rules 4.1.1-4, 4.1.8-1, 4.1.8-2, 9.3-10 (deviation: owner decision 10-27-2026)	`link.rs` `BusPair::power_up`, `Link::b_window_open`
FR-005	Accept requests on both buses; alternate responses (`Strict`; `NonStrict` answers on the request's bus)	Rules 4.1.1-5, 4.1.1-6	`link.rs` `choose_tx_bus`
FR-006	Resend a failed transmission on the other bus, at most `IPMB_TX_RETRIES` times	Rule 4.1.1-7	`link.rs` `on_tx_result`
FR-007	Refuse a Set IPMB State that leaves no bus enabled (CCh)	Rule 4.1.8-3	`link.rs` `plan_ipmb_state`
FR-008	Bus state changes only by Set IPMB State or a persistent local fault	Rule 9.3-10	`link.rs`, `ipmc-runtime` `link.rs`
FR-009	A retried request (same requester, sequence, command and data) is answered from a cache and runs no handler	Obs 4.1.1-6 (a retry keeps its sequence number)	`dupcache.rs`
FR-010	Report sensors 0..6 with the layouts of Tables 10.2.1-1 to 10.2.7-1, and sensor 7 (Payload Mode, Table 10.2.8-1) at Tier-3	Table 10.2-1	`sensor.rs`, `dispatch.rs` `read_sensor`

ID	Requirement	Rule	Where
FR-011	FRU 0 holds Common Header, Board Info and Product Info areas; Chassis Info offset 00h	Rules 8.1-2, 8.1-3	`ipmc-fru/src/records.rs`
FR-012	FRU data readable whatever the payload power	Rule 8-1	`cmd/storage.rs`
FR-013	Write FRU Data while NVMRO is asserted answers 80h and changes nothing; an application port `FRU SET` answers `ERROR NVMRO` and changes nothing; no flash write while NVMRO is asserted	Rules 8-3, 8-5, Rec 4.1.10-1	`cmd/storage.rs`, `ipmc-runtime app_port.rs`, `persist.rs`, `drivers/flash.rs`, `bsp/flash_regs.rs`
FR-014	FRU 0 state machine M0 to M1 on the initial checks, M1 to M4 on valid payload power, M4 to M6 on power loss, M6 to M1 on a complete shutdown; at Tier-3 also M4 to M5 on a deactivation request, M5 to M4 when it is cleared and M5 to M6 on power loss	Section 4.1.5, Rules 4.1.5.4.1-2, 4.1.5.4.2-3, 4.1.5.4.3-2, 4.1.5.4.4-2, 4.1.5.4.4-5, 4.1.5.4.5-1	`fru_state.rs`
FR-015	A payload power-on request in M6 is refused, and an extension can refuse it through the `power_on_allowed` hook; one check serves the boot power-on, the handler hand-off and the console	Rules 4.1.5.4.5-2, 4.1.5.4.2-1	`fru_state.rs` `check_power_on`, `ext.rs` `check_power_on`
FR-016	An invalid hardware address (even parity or GA = 0) isolates both buses and keeps the FRU in M0	Rules 4.1.5.4.1-4, 4.1.5.4.1-5	`addr.rs`, `link.rs`
FR-017	Startup self test of the FRU image and the SDR chain; 55h 00h or 57h with the failure bits; a failure keeps the FRU in M0; rerun after each accepted Write FRU Data	IPMI Table 20-5, Rule 4.1.5.4.1-9, Rec 8-2	`selftest.rs`
FR-018	Write FRU Data and an application port `SET` survive a reset (`IPMC_FRU_PERSIST true`); the FRU store is flash sector 2 alone	owner decisions 10-23-2026 and 10-25-2026	`persist.rs`, `ipmc-runtime fru_log.rs`, `memory.x`, `link.x`
FR-019	No IPMC pull-ups on the IPMB lines (`IPMB_PULL_UPS false`; section 17)	Rule 9.2.3-2	`drivers/ipmb_pins.rs`, `i2c_hw.rs`
FR-020	A (netFn, cmd) the Tier-1 table does not own goes to the extension hook; the Tier-1 build has none and answers C1h	Rules 4.2-2, 4.1.1-16	`ext.rs`, `dispatch.rs`

ID	Requirement	Rule	Where
FR-021	A Tier-2/3 build sends requests one at a time: a response wait of `IPMB_REQ_TIMEOUT_MS` (T3), up to `IPMB_REQ_RETRIES` retries with the same sequence number, a failed transmission retried on the other bus after T6 (60 ms minimum), T4 between new requests, T9 before a sequence number is reused; a Tier-1 build sends none	Rules 4.1.1-9, 4.1.1-10, 7-4, Table 4.1.1-1	`requester.rs`, `link.rs`
FR-022	A Cold Reset request restarts the processor once its response has left the egress queue, the application port output has left and an accepted FRU write is saved, or after `IPMC_RESET_WAIT_MS`	IPMI v2.0 20.2	`reset_gate.rs`, `ipmc-runtime` `link.rs`
FR-023	The optional speed byte of Set IPMB State sets each bus to 100 kHz (00b) or, at Tier-3 on a fitted bus, 400 kHz (01b); every other value, a reserved bit and 400 kHz below Tier-3 answer CCh with no change; every bus starts and restarts at 100 kHz	Rules 4.1.1-4.3, 4.1.1-4.4, 4.1.1-14	`cmd/vita.rs`, `link.rs` `plan_ipmb_speed`
FR-024	A pending Warm Reset (set by an extension handler) restarts the link state and the requester without a processor reset, once its response has left and an accepted FRU write is saved, or after `IPMC_RESET_WAIT_MS`	IPMI v2.0 20.3	`reset_gate.rs`, `ipmc-runtime` `link.rs`, `link.rs` `restarted`
FR-025	A pending payload reset (set by an extension handler) holds PAYLOAD_RST_OUT_N (PA9, open drain) low for `IPMC_PAYLOAD_RESET_PULSE_MS` without blocking the main loop; the FRU state does not change	Rule 4.1.2-9	`reset_pulse.rs`, `sensing.rs`
FR-026	Every whole IPMB frame the IPMC receives (before the parser) or sends (with its outcome) and every trace line goes into a RAM ring of `IPMC_LOG_SIZE` bytes, oldest overwritten first; the console dumps it as CSV or as trace rows	owner, Phase 11 (PLAN, 10-25-2026)	`ipmc-log`, `ipmc-runtime` `log_cmds.rs`, `link.rs`
FR-027	A payload CPU on APP_UART gets, per request line, zero or more `KEY: value` data lines and exactly one final line (`OK` or `ERROR <CODE>: <sentence>`), with the optional tag on the final line, in text a person reads and a parser splits; a `FRU SET` of a unit field rewrites that field in place, moves no area and goes to the flash through the Write FRU Data save path	owner, Phase 11 (PLAN, 10-25-2026)	`ipmc-app`, `ipmc-runtime` `app_port.rs`, `ipmc-fru/src/unit.rs`

ID	Requirement	Rule	Where
FR-028	The bootloader checks the runtime image header (magic, format, size, length, vector table address), the vector table and the SHA-256 digest at every reset, and starts the runtime only when the check passes and no pin or request keeps the bootloader; a failed check keeps the bootloader and its command session	owner, Phase 12 (PLAN decisions 31 to 37)	`ipmc-boot/src/header.rs`, `decision.rs`, `src/boot/main.rs`, `ipmc-boot/src/run.rs`
FR-029	The runtime is erased and loaded again over the console or the application port as Intel HEX records, one record in flight, each record range-checked, erased-checked and read back; sectors 0 and 1 are refused twice and sector 3 is erased first	owner, Phase 12	`ipmc-boot/src/load.rs`, `geometry.rs`, `session.rs`, `bsp/flash_regs.rs`, `tools/fwupdate/fwupdate.py`

Performance requirements

Item	Target	Notes
Response time	at most 227 ms (T5)	request STOP to response STOP; `RESPONSE_DEADLINE_MS`, late ones counted
One master write	at most T2 + T1 = 80 ms	retries included; `WRITE_BUDGET_NS`; one write per pass
Clock stretch	at most 3 ms (T8)	`IPMB_STRETCH_TIMEOUT_US 3000`
Bus-free wait	at least 60 ms (T2)	`IPMB_BUS_FREE_TIMEOUT_MS 60`; ends at the write budget
One IPMB frame	at most 20 ms (T1)	`ipmb_common.rs` `T1_FRAME_NS`
IPMB clock	100 kHz maximum; 400 kHz maximum in FAST-MODE (Tier-3)	half bit 5000 ns; measured SCL period 12.0 us (83 kHz). FAST-MODE: bit-bang 6795 ns (about 147 kHz), hardware master 2981 ns (about 335 kHz)
Runtime image check	164 ms for the 133,120-byte Tier-1 image at 84 MHz (11-13-2026)	`header::check` SHA-256, once per reset; `Image check:` line of the bootloader trace; about 1.2 us per byte
Flash sector erase	typical 250 ms, at most 500 ms (16 KB sector, x32 programming), core stalled; 198 ms measured on the bench	DS10086 Table 45; IPMB slaves off the bus meanwhile; one erase per 60 saves, and only when the bus is quiet

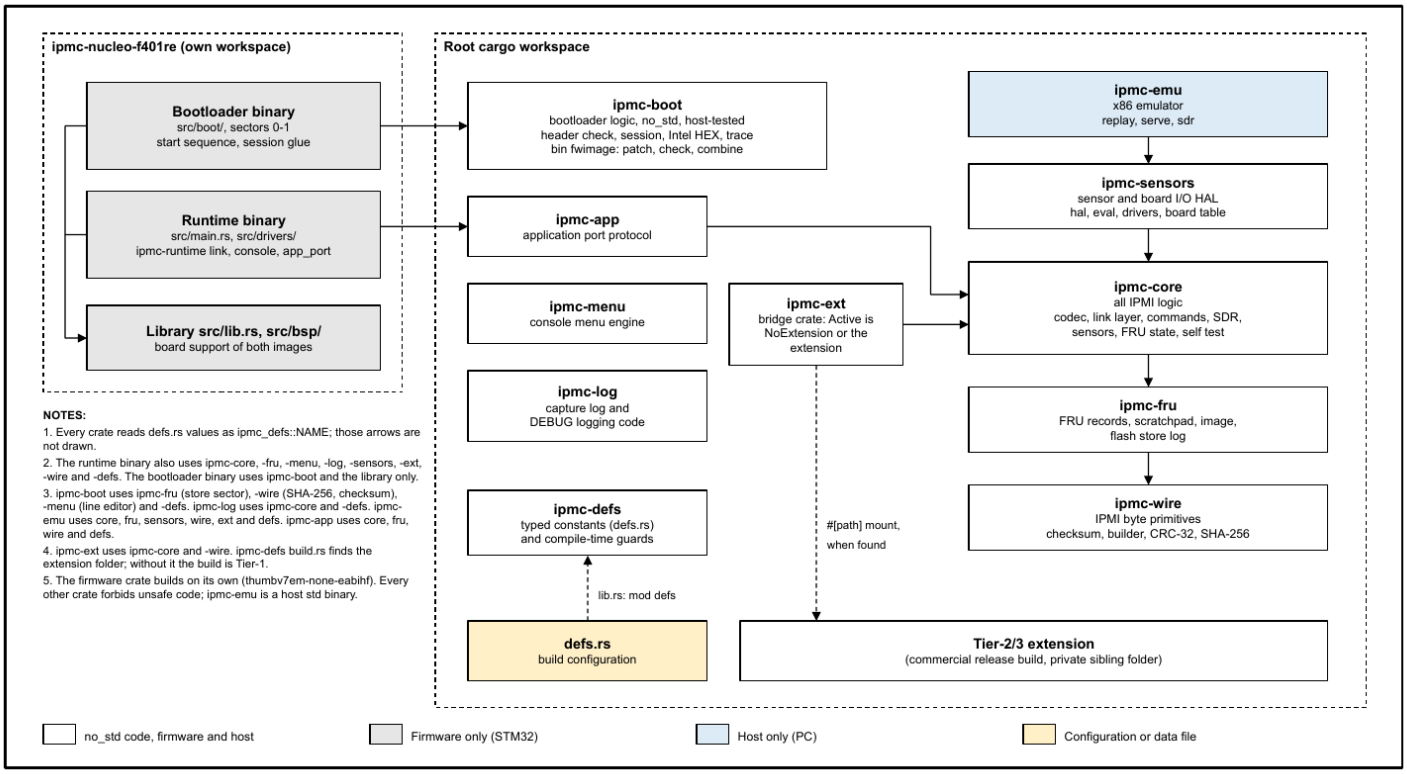
Compliance requirements

ANSI/VITA 46.11-2022 Tier 1, Layer 1 IPMC (VSO revision 22h); IPMI v2.0 rev 1.1 message formats, completion codes (Table 5-2) and the Group Extension defining-body rule (Table 5-1, section 5.3.1). The spec-only suite in `tools/conformance/` (section 13) checks the IPMC against those texts.

4. Architecture

Crate structure

Figure 4-1. Crate structure and dependencies (ipmc-software, sheet 1)



Source: `Documentation/diagrams/src/ipmc-software.js`, sheet 1.

The code is a Cargo workspace of eleven crates plus the firmware crate, which is its own workspace (it builds only for `thumbv7em-none-eabihf`). Dependencies point one way: the binaries at the top (the emulator, the two firmware binaries and the `fwimage` tool), the IPMI and bootloader logic in the middle, the configuration reader at the bottom. The bridge crate `ipmc-ext` sits between the IPMI logic and the binaries; it is the only place the private Tier-2/3 extension enters the build.

Crate	Kind	Holds
`ipmc-defs`	`no_std` library	the build configuration as Rust (`src/defs.rs`, and the extension's `defs_tier23.rs` when the extension is built in): one typed `pub const` per define, no parser, compile-time guards on every value, the IPMB timing arithmetic (`i2c_timing.rs`) and the one detection of the Tier-2/3 extension (`build.rs`, `tier23_detect.rs`, `dep_cfg.rs`)
`ipmc-wire`	`no_std` library	IPMI byte primitives: zero checksum, const record builder, type/length ASCII field, CRC-32, SHA-256 (`Sha256`), the boot request check word, hex-byte parser

Crate	Kind	Holds
`ipmc-fru`	`no_std` library	every FRU 0 record (image build, decoder, `verify`), the FRU field editor (`unit.rs`) and the manufacturing date (`date.rs`), the RAM scratchpad (`FruStore`), the flash store (`persist`) and, behind the `host` feature, the in-RAM flash `RamFlash` of the tests and the emulator
`ipmc-core`	`no_std` library, `forbid(unsafe_code)`, no allocator	every IPMI decision: frame codec, address decode, link layer with the IPMB requester, duplicate cache, queues, byte ring, shared names (`names.rs`), command table, dispatcher, extension hook interface, handlers, SDR repository, Tier rules, event enables, sensor encoders, FRU state machine, self test, reset gate, trace hook
`ipmc-log`	`no_std` library, `forbid(unsafe_code)`, no allocator	the capture log and the logging logic of the DEBUG menu: record ring (`record.rs`), CSV and trace printers (`print.rs`), the paced dump (`dump.rs`), the trace tee (`tee.rs`) and the log switches with their status lines (`switches.rs`); depends on `ipmc-core` and `ipmc-defs`
`ipmc-boot`	`no_std` library with a host binary, `forbid(unsafe_code)`, no allocator	every bootloader decision with no F401 code: the image header and its one check, the SHA-256 digest use, the start decision and the countdown, the boot request words, the region policy of erase and `LOAD`, the command table and the session state machine, the Intel HEX parser, the trace text, the generic boot sequence `run.rs` (`BootBoard`, `BootParts`, `boot`: the start-up order, the countdown, the stay, the session loop over `ipmc-hal` handles, host-tested over a test board) and the `fwimage` host tool (section 15); depends on `ipmc-defs`, `ipmc-wire`, `ipmc-fru`, `ipmc-hal` and `ipmc-menu`
`ipmc-hal`	`no_std` library, `forbid(unsafe_code)`, no allocator	the boundary between the IPMC code and one board, plain data and the interrupt-side frame state: the IPMB master errors and counters (`ipmb.rs`), the received frame, the consumer trait `FrameSource` and `FrameAssembler` (`frame.rs`: the frame state of one I2C target, host-tested, events returned and counted by the board), the serial counters and the `SerialPort` trait (`serial.rs`), `ResetCause`, `WatchdogError` and the `SystemControl`, `Watchdog`, `Straps` and `StatusLed` traits with `ClockSummary` (`system.rs`), `Clock`, `CycleCounter` and `delay_us` (`time.rs`), `KeptWords` (also implemented by `[u32; N]`) with the checked record `store_checked` and `take_checked` (`kept.rs`), the `IpmbBus` trait and `BringUpFailure` (`ipmb.rs`), `Board` and `Platform` (`board.rs`), the kept fault record (`fault.rs`), and the host test doubles of the traits behind the `testing` feature (`testing.rs`: `FakePort`, `TestClock`, `TestCycles`, `TestStraps`, `TestLed`, shared by the `ipmc-boot` and `ipmc-runtime` tests); depends on `ipmc-defs`, `ipmc-core`, `ipmc-fru`, `ipmc-sensors`, `ipmc-wire` and `heapless`; the F401 crate re-exports its types at their old paths

Crate	Kind	Holds
`ipmc-runtime`	`no_std` library, `forbid(unsafe_code)`, no allocator	the IPMC services that run on any board, generic over the `ipmc-hal` traits and host-tested: the console text layer (`text.rs`: `ConsoleText` for every `SerialPort`, `ConsoleTrace`, `MenuOut`, `DumpOut`), the console (`console.rs`: `Console<B, L>` and `Context<B, L>`, the command tables as associated consts that `Console::new` checks at build time) with its command groups (`data_cmds.rs`, `debug_cmds.rs`, `log_cmds.rs`, `sensing_cmds.rs`, `ipmb_cmds.rs`), the application port (`app_port.rs`: `AppPort<B>`, `LinkHost`), the FRU log owner `FruLog<F: FlashLog>` (`fru_log.rs`), the sensor service `Sensing<A, Io>` (`sensing.rs`), the request sequence kept across a software reset (`kept.rs`), the IPMB link service (`link.rs`: `IpmbLink<B>`, `start_buses`, `LinkArgs`, `Restart` and `restart`) and the start-up order with the main loop (`run.rs`: `StartInfo`, `BoardParts`, `SystemParts`, `start`, `Runtime::pass`, `run`) and, behind the feature `sim` (std, never in a firmware image), the simulated board `sim.rs` that the unit tests and the emulator run on; depends on `ipmc-app`, `ipmc-boot`, `ipmc-core`, `ipmc-defs`, `ipmc-ext`, `ipmc-fru`, `ipmc-hal`, `ipmc-log`, `ipmc-menu`, `ipmc-sensors` and `ipmc-wire`
`ipmc-ext`	`no_std` library, `forbid(unsafe_code)`	the bridge: `ipmc_ext::Active` is `NoExtension` in the Tier-1 build and the extension's state when `ipmc-defs` found it; the firmware and the emulator hold one `Active`
`ipmc-sensors`	`no_std` library	sensor and board-I/O HAL: four traits, hardware-free evaluation, drivers (simulation, example I2C/SPI parts; the STM32F401 ADC and GPIO drivers are in the firmware crate), board table, sampling monitor, payload reset pulse timing, GPIO pin ownership
`ipmc-menu`	`no_std` library, no dependencies besides `ipmc-defs`	the console menu engine (MenuBuilder rules, tiered groups)
`ipmc-app`	`no_std` library, `forbid(unsafe_code)`, no allocator; a reference crate for a customer application (PLAN decision 27)	the application port protocol: line assembler, command table `COMMANDS`, handlers, and the `AppHost` trait through which the firmware gives the protocol its view of the board (section 6.8)
`ipmc-emu`	host `std` binary and library	the x86 emulator: a board that runs `ipmc-runtime` on the simulated board `Sim` (`Emulator`), the vector reader, capture replay (also the Event Receiver), real-time `serve`, SDR dump; depends on `ipmc-runtime` with the feature `sim` and on `ipmc-hal`

Crate	Kind	Holds
`ipmc-nucleo-f401re`	one crate, `no_std`, its own cargo workspace: a library and two binaries	the library (`src/lib.rs`, `src/bsp/`) is the board support both images use: clock, UART driver, LED, SysTick, DWT cycle counter, strap pins, NVMRO and the BOOTLOADER pin, reset cause, FLASH register layer, boot request words, fault handlers, `static_slot`, the `out!` console macro; the runtime binary (`src/main.rs`, `src/drivers/`, sectors 3-7) holds start-up, the hardware only the runtime uses (IPMB masters, I2C slaves, IPMB and board pins, the FRU store driver), the console, the application port transport and the composition of the crates above; the bootloader binary `ipmc-boot` (`src/boot/`, sectors 0-1) is the hardware glue of `ipmc_boot::run::boot`: the `BootBoard` implementation, the start sequence and jump, flash access

Layering rules

- `ipmc-core` decides; the firmware and the emulator move values. A handler never touches hardware; a driver never decides an IPMI response.
- No IPMI code runs in an interrupt. The I2C interrupt posts whole frames to a per-bus single-producer single-consumer queue; the main loop runs the link layer and the handlers.
- Two images, one library: the runtime and the bootloader link the same `src/bsp/` code and never run together. The bootloader owns the processor from reset until it jumps; `src/boot/start.rs` returns every peripheral it used to its reset state before the jump, and the runtime start-up then runs unchanged (section 15.6). `bsp/flash_regs.rs` refuses sectors 0 and 1 for both.
- One owner per hardware resource: `ports` owns GPIOA..GPIOC and hands out pin claims; `drivers/i2c_hw.rs` owns I2C1 and I2C3; the bit-bang master owns the IPMB pins only while it transmits; `bsp/flash_regs.rs` owns the FLASH registers after clock start-up (`drivers/flash.rs` is the store policy over it); `uart.rs` owns USART1, USART2 and their pins; `led.rs` owns PA5; `ga.rs` owns the strap pins (GA, GAP, NVMRO and the BOOTLOADER pin); `ipmc-runtime/src/link.rs` owns the Link, the dispatch Context and the capture log (the one `IpmbLink<F401>` sits in a static of `main.rs`); `ipmc-runtime/src/app_port.rs` moves bytes between the APP_UART rings and `ipmc-app` and owns nothing else.
- The application protocol never touches hardware: `ipmc-app` asks the board through the `AppHost` trait, and the per-unit FRU fields reach the flash through the one path a Chassis Manager's Write FRU Data uses (the RAM image, the dirty mark, the save step).
- The dispatcher asks the extension hook only after the Tier-1 table has no row for a command, so a Tier-1 command cannot be replaced. Nothing specific to a Tier-2/3 command, event or hook is written in this repository: board support, processor functions (reset, flash, timers, the I2C speed) and the link layer, the IPMB requester included, live here; the Tier-2/3 commands, hooks and sensor events live in the extension.
- One implementation of each function, judged by behaviour (`tools/dupcheck.py` is a gate): one checksum, one frame parser and encoder, one GA decode, one sensor-read function, one record builder, one retry loop shared by both master methods.
- One file for every tunable value: `crates/ipmc-defs/src/defs.rs` (section 5). Values fixed by a specification or datasheet stay in the module that owns them, with the rule cited beside each.

Key design choices

- The link layer (`ipmc-core/src/link.rs`) is a pure state machine: frames go in with the STOP time, responses come out with the bus to use, results come back. The same object runs in the firmware and the emulator, so the dual-bus rules are tested on the PC against 309 captured exchanges.
- The bit-bang master (`drivers/i2c_bitbang.rs`) drives the slave's own pins. The I2C peripheral stays a slave; for a transmission it is switched off, the same pins are driven open-drain as GPIO, then the peripheral is switched back on. This follows the earlier PIC32 controller and avoids the STM32 I2C master state machine; the hardware master stays available behind `IPMB_MASTER_MODE`.
- The response bus is chosen at first transmission (`Link::next_tx`), not at queue time, so under `Strict` alternation follows the frames really handed to a driver, resends included.
- The duplicate cache (`dupcache.rs`) is keyed by the exact request, so a retry runs no handler and a retried Write FRU Data or Set IPMB State takes effect once.
- Set IPMB State uses a hand-off (`PendingIpmbState`). The handler cannot reach the Link; it validates against a read-only view and leaves the change for `Link::poll`, which applies it before the response is queued.
- The FRU flash store (`ipmc-fru/src/persist.rs`) is an append-only log in one sector (flash sector 2, 16 KB, 31 records of 528 bytes) with a commit word programmed last and a CRC-32 per record; the highest sequence number wins at boot, and a power cut at any word of an append leaves the previous record loadable. The save that finds the sector full erases it and writes the newest image first (owner decision 10-25-2026, no second copy): a power cut between that erase and that write loses the stored image, and the next start-up loads the built image.
- The capture log (`ipmc-log/src/record.rs`) is one byte ring of whole records. Every trace sink the firmware hands to the core is a `LoggedTrace`, which records the line and prints it on the console unless the IPMI console trace is off (section 10.11).
- Both UART ports run one driver (`bsp/uart.rs`) with a receive and a transmit ring each; a write never waits for the wire, and a line that does not fit the transmit ring is dropped and counted (section 6.6).
- Configuration is checked at compile time (`ipmc-defs`). A malformed or out-of-range define fails the build with its line number or the guard that names it.
- The requester (`requester.rs`) sits in the link layer. The request sequence number, response matching and the resend timing sit beside the response path of `Link`, so one `next_tx` hands out both and responses keep priority over requests. The extension decides what to send and when; a Tier-1 build never sends a request (Rule 7-4).
- A processor reset uses a hand-off (`reset_gate.rs`). A handler asks for a Cold Reset through `Context::pending_reset`; the owner of the processor performs it when the gate says nothing is lost (section 8.2).
- Processor and payload services use a hand-off (`dispatch.rs`). A handler never switches payload power, pulses the payload reset output or restarts the link; it leaves a request in the `Context` (`pending_power`, `pending_payload_reset`, `pending_reset`) and the owner of that resource performs it (section 6.3, section 8.2).
- The tier rules live in one place (`tier.rs`). Get Device ID, the Device SDR records, the event enables, the FRU state machine, the sensor count and the IPMB speed derive their Tier-dependent behaviour from six rules (section 9.6).

Build modes and the Tier-2/3 extension

One source tree builds two kinds of image. The public build is the Tier-1 IPMC of this document. The commercial release build adds the Tier-2/3 extension, which lives in a private sibling folder named `IPMIRustFRU_Tier23`; this repository holds only the hook interface, the bridge and the detection. The public Tier-1 build answers every Tier-2/3 command C1h (section 7.2).

Build	Selected by	`IPMC_TIER`	`ipmc_ext::Active`
Public Tier-1	`defs.rs`: `IPMC_TIER23`, `IPMC_TIER2`, `IPMC_TIER3` all `false`	1	`NoExtension`
Commercial release, Tier-2	`IPMC_TIER23` and `IPMC_TIER2` `true`, `IPMC_TIER3` `false`	2	the extension
Commercial release, Tier-3 (the default)	all three `true`	3	the extension
Evaluation (section 8.4)	`IPMC_EVAL=1` with the extension, release profile	as the commercial build	the extension

The three switches sit at the top of `defs.rs`. `lib.rs` derives `IPMC_TIER` from them (3 with `IPMC_TIER3`, 2 with `IPMC_TIER2` alone, 1 without the extension) and stops the build when `IPMC_TIER3` is set without `IPMC_TIER2` or `IPMC_TIER2` differs from `IPMC_TIER23`. The IPMB bus settings are independent of the Tier: a Tier-2 or Tier-3 build runs on IPMB-A, on both buses, or with IPMB-B not fitted, as `IPMB_BUS_B_ENABLE` and `IPMB_COMPLIANCE` say.

`crates/ipmc-defs/build.rs` is the one detection point; its rules are pure functions in `tier23_detect.rs`, tested by `tests/tier23.rs`. A cfg must be known before the crate compiles, so `build.rs` reads the one line `pub const IPMC_TIER23: bool = ...;` of `defs.rs` (`parse_defs`; a missing, repeated or malformed line fails the build).

Setting	Result
`IPMC_TIER23` `false`	off, whatever exists: the public Tier-1 build
`IPMC_TIER23` `true`	on; a missing marker file (the extension's `defs_tier23.rs` or its library source) fails the build, names it and says how to make the Tier-1 build
environment `IPMC_TIER23=0` or `1`	overrides `defs.rs` for one build, with a cargo warning; the tools that build Tier-1 and Tier-2/3 in one run (`tools/test_coverage.py`, `tools/captured_sdr.py`, the gates) use it; any other value fails the build

- Defines: off uses `defs.rs` and `IPMC_TIER` = 1; on, `lib.rs` mounts the extension's `defs_tier23.rs` (a `#[path]` module under `cfg(tier23)`) and re-exports it beside `defs.rs`; a name in both files is ambiguous where it is used and fails the build. The guards of the extension's defines are mounted from the extension folder (`defs_guards.rs`).
- `ipmc-defs` publishes the decision. It declares `links = "ipmc_defs"` and prints `cargo:tier23=1` or `0` (and `cargo:eval=1` or `0`); cargo hands these to each direct dependent as `DEP_IPMC_DEFS_TIER23` and `DEP_IPMC_DEFS_EVAL`. `build.rs` also sets `cfg(tier23)` for `ipmc-defs` itself and exports `ipmc_defs::TIER23_BUILT`.

- The dependents share one build-script step. `ipmc-ext` and the firmware crate include `crates/ipmc-defs/dep_cfg.rs` and call `cfg_from_defs(key, cfg, path)`, which turns the published value into `cfg(tier23)` or `cfg(ipmc_eval)`. They read no file and no other variable, so they cannot disagree with `ipmc-defs`.
- With `cfg(tier23)`, `ipmc-ext` mounts the extension's sources through a relative `#[path]` and `Active` is the extension's state type; otherwise `Active` is `NoExtension`; `IPMC_TIER` is 1 without the extension by its definition.
- `build.rs` watches `src/defs.rs`, so a change of the switch takes effect at the next build. `gen.ps1 build` sets no Tier variable.
- `IPMC_EVAL` is unset or 0 (off) or 1 (on); `IPMC_EVAL=1` without the extension, or outside the release profile, fails the build. On, it prints a build warning with the length and sets `cfg(ipmc_eval)`.

The hook interface is `ipmc_core::ext::Extension`, five methods with a default for the last four.

Method	Called	Purpose
`command(ctx, req, out)`	by the dispatcher, after the Tier-1 table has no row for (netFn, cmd) and the LUN and VSO checks passed	answers a command the extension owns with its name and a `Handler`-style result; `None` gives C1h. For NetFn 2Ch the request data and `out` exclude the VSO byte, as for a `Handler`
`poll(ctx, requester, now_ms, trace)`	once per `Link::poll`, after the queued requests are answered and `ctx.link` is refreshed	the extension sends its requests through the `Requester`
`on_sample(ctx, transition, now_ms, trace)`	once per sensor sample, after `Context::update_sensors` stored the snapshot and stepped the FRU state machine	`transition` is the FRU state change that sample made, if any
`warm_reset()`	once, after the owner applied a Warm Reset (`Context::warm_reset` ran)	the extension re-initialises its volatile state (a request in flight, its event timers) and keeps what a Warm Reset keeps (section 8.2)
`power_on_allowed(ctx)`	by `ext::check_power_on`, after the FRU state machine's own check passed	whether payload power may come on now; the default allows it, so the Tier-1 build is unchanged, and a Tier-3 extension answers false while the FRU is activation-locked (Rule 4.1.5.4.2-1)

`NoExtension` answers nothing and takes the defaults of the hooks. `ext::feed_sensors(ctx, ext, now_ms, snapshot, trace)` is the one sensor-sample entry of the firmware and the emulator: it calls `update_sensors`, then `on_sample` with the transition, and returns the transition. `ext::check_power_on(ctx, ext)` is the one combined power-on check: `FruMachine::check_power_on` first (`PowerOnRefused::DeactivationInProgress` in M6), then `power_on_allowed` (`PowerOnRefused::ActivationLocked`); the first refusal is returned. The boot power-on, the `pending_power` On hand-off and the console `POWER ON` all call it.

5. Build-Time Configuration

`crates/ipmc-defs/src/defs.rs` is the single build configuration. It is plain Rust with no parse or generation step: one `pub const NAME: T = value;` item per tunable, each with a `///` comment that carries the rule beside it, in the order of the sections below. The file holds no `use`, no function, no macro and no check; a person changes a build by editing a value (and, for a choice, the variant of a small enum). Every crate reads a value as `ipmc_defs::NAME`; `lib.rs` re-exports the module (`pub use defs::*`).

Each define has the type its consumers need:

Kind	Type	Examples
switch or enable	`bool`	`IPMB_BUS_B_ENABLE`, `IPMC_FRU_PERSIST`, `IPMC_SUPPORT_SENSOR`
size, depth, count, length used as an index	`usize`	`IPMB_MSG_MAX`, `IPMC_LOG_SIZE`, `IPMC_FRU_SIZE`, the `IPMC_FRU_*_MAX` rooms
wire or register value of a fixed width	`u8`, `u16`	`IPMB_CM_ADDR8`, `IPMB_GA_FORCED`, `IPMC_DEVICE_ID`, `IPMC_PRODUCT_ID`, `IPMC_EMU_PORT`
time, rate, frequency, other number	`u32`	`IPMB_I2C_HZ`, `IPMC_SENSOR_PERIOD_MS`, `IPMC_MANUFACTURER_ID`
signed number	`i32`	the `IPMC_TEMP_*_C` thresholds and `IPMC_SIM_TEMP_C`
text	`&str`	`IPMC_FW_NAME`, the FRU strings, the SDR ID strings; six FRU strings have `_T1`, `_T2`, `_T3` variants that `IPMC_TIER` selects
choice between alternatives	enum	`MasterMode`, `IpmbCompliance`, `StrapSource`, `Source`, `TraceLevel`

A define that names an earlier one is an expression of it (`pub const B: usize = A;`). Because the types state each define's kind, a wrong type or a value outside its type fails to compile at the define or at its consumer, and the build needs no `[i32]` tag or `{0, 1}` guard. `lib.rs` holds more than 100 compile-time guards (`const _: () = assert!(..)`, written out and from the FRU-string, SDR-name and rail guard macros) that tie values to their rules (for example `IPMB_STRETCH_TIMEOUT_US` at most 3000, the T8 limit; `IPMC_FRU_SIZE` a multiple of 8; every FRU string at most 63 bytes); a failed guard stops the build and names the define. The Python bench tools read the values they share with the IPMC (`IPMC_EMU_PORT`, `IPMC_APP_UART_BAUD`, `IPMC_APP_LINE_MAX`, `IPMC_CONSOLE_BAUD`) through `tools/defs.py`, the one reader of the file outside the Rust build.

Prefixes: `IPMB_` for bus values, `IPMC_` for controller and application values. The tables below group the defines; the file itself carries the rule beside each item.

Bench and test switches

The block at the top of `defs.rs` holds the switches to check before a system build (PLAN 3.7 decision 24).

Define	Default	Meaning
`IPMC_EVENTS_AT_BOOT`	`false`	read only by a board when it fills `StartInfo::events_at_boot` (the F401 main; the emulator uses `EmuConfig::events_at_start`, true), no effect in a Tier-1 build: false (bench) starts the event generator off and the console prints `Events: off from start-up`; true (system build) starts it on. `DEBUG EVENTS ON` or `OFF` changes it at run time; the behaviour is in `../IPMIRustFRU_Tier23/Documentation/` (private)
`IPMC_LOG_AT_BOOT`	`true`	the capture log records from start-up; `DEBUG LOG ON` or `OFF` changes it at run time
`IPMC_I2C_TRACE_AT_BOOT`	`false`	one console line per IPMB frame in and out from start-up; `DEBUG TRACE I2C ON` or `OFF` changes it at run time; a row that does not fit the console TX ring whole is dropped and counted (`tx_lines_dropped`), and the frame stays in the capture log

IPMB link and drivers

Define	Default	Meaning
`IPMB_MASTER_MODE`	`MasterMode::BitBang`	master method of both buses; `MasterMode::Hardware` selects the hardware I2C master
`IPMB_BUS_A_ENABLE`	`true`	IPMB-A built in; must be true (Rule 4.1.1-1)
`IPMB_BUS_B_ENABLE`	`true`	IPMB-B built in (Get VSO Capabilities reports two interfaces); `false` leaves its pins parked and reports one
`IPMB_PULL_UPS`	`true`	the processor's internal pull-ups (about 40 kΩ) on the four IPMB pins in every pin mode, so an unwired bus reads idle-high; a bench deviation from Rule 9.2.3-2 (section 17); `false` for a module in a chassis
`IPMB_I2C_HZ`	100000	IPMB clock, STANDARD-MODE
`IPMB_I2C_FAST_HZ`	400000	IPMB FAST-MODE clock (Rule 4.1.1-4.4; Tier-3, entered with the Set IPMB State speed byte); fixed at 400000, the hardware master derives its CCR and TRISE from it (RM0368 18.6.8, 18.6.9)
`IPMB_MSG_MAX`	32	whole frame, rsSA to the second checksum

Define	Default	Meaning
`IPMB_COMPLIANCE`	`IpmbCompliance::NonStrict`	`Strict`: VITA 46.11 as written; IPMB-B ignores every frame until a Set IPMB State enables it (Rules 4.1.1-4, 4.1.8-1) and responses alternate between IPMB-A and IPMB-B when both are enabled (Rule 4.1.1-6). `NonStrict`: until the first Set IPMB State that sets IPMB-B a request that arrives on IPMB-B is answered on IPMB-B (the IPMB-B state and the IPMB Link sensor still read isolated), and every response goes back on the bus its request arrived on. In both, the IPMC's own requests and events alternate buses and a failed transmission is resent on the other bus (Rule 4.1.1-7). `LinkConfig::from_defs` derives `TxPolicy` (`Alternate` / `Arrival`) and `IpmbBPowerUp` (`Strict` / `NonStrict`) from it; the emulator options `--policy` and `--b-power-up` override them one by one. The conformance case `LINK-ALTERNATION` fails under `NonStrict` (section 10.2)
`IPMB_CM_ADDR8`	20h	Chassis Manager address; only guarded, no code reads it (responses go to each request's rqSA)
`IPMB_DUP_CACHE`	64	duplicate-cache entries (8..=64)
`IPMB_DUP_WINDOW_MS`	3000	retry window: covers 11 attempts of 250 ms, below the 3.846 s sequence reuse of the captured Chassis Manager
`IPMB_TX_RETRIES`	3	resends of one response on the other bus (3..=10, C1)
`IPMB_REQ_TIMEOUT_MS`	250	wait for the response to a request this IPMC sent, before a retry (60..=250; T3max, Rule 4.1.1-9; T6 spaces the retries)
`IPMB_REQ_RETRIES`	10	retries of a request after its first transmission (3..=10; C1max, Rule 4.1.1-10): 11 transmissions x 250 ms = 2.75 s, inside the T9 expiry of 5 s
`IPMB_REQ_GAP_MS`	5	least time from one new request to the next (at least 5; T4, Table 4.1.1-1)

Define	Default	Meaning
`IPMC_RESET_WAIT_MS`	3000	longest wait before a pending Cold Reset runs anyway (1..=5000): the response leaving the egress queue, one flash sector erase (at most 500 ms) and the FRU write fit inside it
`IPMC_PAYLOAD_RESET_PULSE_MS`	100	time PAYLOAD_RST_OUT_N (PA9) is held low by a payload reset pulse (1..=1000; Rule 4.1.2-9)
`IPMB_INGRESS_DEPTH`	8	received requests held for `Link::poll` (1..=32)
`IPMB_TXQ_DEPTH`	4	response frames in the egress queue
`IPMB_RX_QUEUE_DEPTH`	4	frames one bus interrupt can post before the main loop drains them (2..=16)
`IPMB_BITBANG_HALF_NS`	5000	bit-bang half bit
`IPMB_BITBANG_FAST_LOW_NS`	1400	bit-bang SCL low time in FAST-MODE (1300..=100000; the I2C fast-mode tLOW minimum is 1300 ns)
`IPMB_BITBANG_FAST_HIGH_NS`	1000	bit-bang SCL high time in FAST-MODE (600..=100000; the I2C fast-mode tHIGH minimum is 600 ns)
`IPMB_STRETCH_TIMEOUT_US`	3000	longest stretched SCL (T8)
`IPMB_BUS_FREE_TIMEOUT_MS`	60	T2 share of the master write budget (T2 + T1); the free-bus wait ends at that budget
`IPMB_ARB_RETRIES`	3	same-bus retries after arbitration loss (0..=10)
`IPMB_FAULT_STREAK`	8	consecutive bus-fault errors before a bus is marked faulted (2..=64)
`IPMB_FAULT_CLEAR_MS`	1000	both lines idle-high this long before a fault clears
`IPMC_IPMI_TRACE`	`true`	one trace line per dispatched request, and the link and event lines (failed transmissions, resends, requests sent). false removes the link and dispatcher lines at compile time, so they are absent from the capture log as well as from the console; the status lines stay (FRU log saves, bus faults, bus speed changes, resets, and the FRU state change line)

Define	Default	Meaning
`IPMC_LOG_SIZE`	8192	capture log ring, bytes (1024..=16384, a power of two; the stack reservation check of `link.x` fails a build whose statics leave too little RAM); the usable size is one byte less, because the ring keeps a slot empty; each `DEBUG DUMP` row walks the log from its start, so a larger log makes the dump slower per row
`IPMC_STACK_RESERVE_BYTES`	35840	runtime stack reservation, bytes (multiple of 8, below the RAM size): RAM that must stay free above the statics; `link.x` fails the link when the statics leave less
`IPMC_BOOT_STACK_RESERVE_BYTES`	8192	bootloader stack reservation, bytes (multiple of 8, below the RAM size); `boot.x` fails the link when the statics leave less
`IPMC_LOG_TEXT_MAX`	160	longest trace line the log keeps, bytes (32..=200); a longer line is cut and ends in `...`

Straps, identity and FRU

Define	Default	Meaning
`IPMB_GA_SOURCE`	`StrapSource::Forced`	GA from this file (bare Nucleo, no straps wired) or `StrapSource::FromPins`
`IPMB_GA_FORCED`	01h	slot 1, hardware address 41h, IPMB 82h; GAP generated for odd parity
`IPMB_NVMRO_SOURCE`, `IPMB_NVMRO_FORCED`	forced, 0	NVMRO deasserted (writes allowed)
`IPMC_DEVICE_ID`, `IPMC_DEVICE_REVISION`, `IPMC_DEVICE_PROVIDES_SDRS`	01h, 0, 1	Get Device ID byte 2, byte 3 bits 3:0, byte 3 bit 7
`IPMC_IPMI_VERSION`	02h	IPMI 2.0 (least significant digit in bits 7:4)
`IPMC_SUPPORT_*`	sensor 1, FRU inventory 1, others 0	Get Device ID byte 7
`IPMC_MANUFACTURER_ID`	000000h	unspecified until an IANA enterprise number is registered
`IPMC_PRODUCT_ID`	494Bh	product ID

Define	Default	Meaning
`IPMC_FRU_BOARD_*`, `IPMC_FRU_PRODUCT_*`, `IPMC_FRU_FILE_ID` (the board file ID), `IPMC_FRU_PRODUCT_FILE_ID` (empty), `IPMC_FRU_ASSET_TAG`	strings	FRU board and product area fields
`IPMC_FRU_MFG_DATE`	`2026-10-23 00:00`	manufacturing date, `YYYY-MM-DD HH:MM` (UTC), 1996-01-01 00:01 to 9999-12-31 23:59; the build checks it with the date parser of `date.rs`. It is the default of the VITA 34h record (section 6.2)
`IPMC_FRU_SIZE`	512	FRU inventory area size: the built image with every field's reserved room and the 14-byte date record (section 9.2); the flash record is 528 bytes, 31 per sector
`IPMC_FRU_CONSOLE_WRITE`	`true`	true = the console `FRU` menu sets every field (`SET`) and restores the built image (`RESTORE`); false = the menu is read only (`SHOW`, `DUMP`, `STORE`, `FIELDS`, `GET`) and `SET` and `RESTORE` do not exist (not in `HELP`, a typed `SET` is an unknown command)
`IPMC_FRU_PERSIST`	`true`	Write FRU Data and the application port survive reset (the flash store, sector 2 alone, 16 KB); false keeps the FRU in RAM and a reset restores the built image
`IPMC_FRU_BOARD_MFR_MAX`, `IPMC_FRU_BOARD_PRODUCT_MAX`, `IPMC_FRU_BOARD_SERIAL_MAX`, `IPMC_FRU_BOARD_PART_MAX`, `IPMC_FRU_BOARD_FILE_ID_MAX`, `IPMC_FRU_PRODUCT_MFR_MAX`, `IPMC_FRU_PRODUCT_NAME_MAX`, `IPMC_FRU_PRODUCT_PART_MAX`, `IPMC_FRU_PRODUCT_VERSION_MAX`, `IPMC_FRU_PRODUCT_SERIAL_MAX`, `IPMC_FRU_ASSET_TAG_MAX`, `IPMC_FRU_PRODUCT_FILE_ID_MAX`	24, 32, 24, 24, 16, 24, 32, 24, 16, 24, 24, 16	room the built areas reserve for each text field, bytes, whatever the default string's length (the serial numbers 17..=63, so that a MAC address text fits; the others 2..=63; each default must fit). An update up to this length rewrites its field in place and no area moves. The product version default must equal the crate's major.minor; a value written later is not checked against it
`IPMC_FRU_PERSIST_DELAY_MS`	1000	quiet time after the last write before a save
`IPMC_FRU_SCAN_CHUNK`	64	bytes per read when the log checks a range for erased flash (a multiple of 4, 16..=272; the buffer lives on the stack)

Define	Default	Meaning
`IPMC_FRU_ERASE_QUIET_MS`	500	run-time erase waits for an empty egress queue and this long without a received frame
`IPMC_FRU_PAUSE_WAIT_MS`	40	longest wait for an IPMB slave to go idle before an erase
`IPMC_URGENT_LINK_WAIT_MS`	250	longest wait of the urgent save before a Cold Reset for the link to hold no accepted request and no untransmitted response (228..=1000): longer than the 227 ms response deadline, so an accepted request is answered first. `IPMC_RESET_WAIT_MS` must exceed it plus `IPMC_FRU_PAUSE_WAIT_MS` plus 500 ms, the DS10086 Table 45 sector erase maximum (16 KB sector)
`IPMC_PAYLOAD_MODES`	3	payload operational modes this payload supports, bit n = mode n (modes 0..=14, bit 0 set, bit 15 clear): 3 is modes 0 and 1. The Payload Mode sensor SDR record (sensor 7, Tier-3) carries it as its reading and assertion masks
`IPMC_SDR_*_NAME`	strings	SDR ID strings, 2..=16 bytes (locator at most 6); `IPMC_SDR_PAYLOAD_MODE_NAME` names sensor 7 (Tier-3 only); `IPMC_SDR_DIE_TEMP_NAME` names the die temperature analog sensor, the rails use `IPMC_RAILn_NAME`
`IPMC_SDR_ANALOG_SENSORS`, `IPMC_SDR_ANALOG_FIRST`, `IPMC_SDR_RAIL_SPAN_PCT`, `IPMC_RAIL_COUNT`, `IPMC_TEMP_NOMINAL_C`	`true`, 8, 170, 5, 25	publish the rails and the die temperature as analog sensors with Full SDR records (numbers from 08h, after the mandatory sensors; section 9.5); the board rail table length (a guard in `board_sensors.rs` checks it); the nominal die temperature of the temperature sensor

Console, clock, LED, sensors and emulator

Define	Default	Meaning
`IPMC_CONSOLE_BAUD`	115200	console UART, 8N1 (`gen.ps1` reads the same line)

Define	Default	Meaning
`IPMC_CONSOLE_BYTES_PER_POLL`	16	console bytes per main-loop pass; at most one command line per pass
`IPMC_UART_RX_BUF`	256	console receive ring, bytes; a power of two, one slot stays empty, and the usable bytes (size - 1) cover `IPMC_MENU_LINE_MAX`
`IPMC_UART_TX_BUF`	4096	console transmit ring, bytes, a power of two; 4096 bytes are 356 ms at 115200 baud
`IPMC_APP_UART_ENABLE`	`true`	the application port APP_UART (USART1) runs; false leaves USART1 off and parks its pins
`IPMC_APP_UART_BAUD`	115200	application port baud rate, 8N1
`IPMC_APP_UART_FLOW`	`false`	true = RTS/CTS hardware handshake on PA12/PA11; false = TX and RX only, and PA11/PA12 stay unclaimed
`IPMC_APP_UART_RX_BUF`, `IPMC_APP_UART_TX_BUF`	256, 1024	application port rings, bytes, powers of two; the receive ring holds at least one line (`IPMC_APP_LINE_MAX` + 2), the transmit ring (512 at least) takes the output in pieces: the output buffer drains into it across main-loop passes (`drain`) and no guard ties the ring to `ANSWER_MAX`; the ring holds `IPMC_APP_UART_TX_BUF` - 1 bytes at once
`IPMC_APP_LINE_MAX`	128	longest application port request line, bytes (16..=`IPMC_APP_UART_RX_BUF` - 3)
`IPMC_APP_BYTES_PER_POLL`	32	most application port bytes one main-loop pass takes
`IPMC_APP_PROMPT`	`APP>`	application port prompt, printed after the banner and after every answer and in front of the echo of every request; 1..=16 printable ASCII bytes, the last a space (`tools/app-port/app_cli.py` reads it through `tools/defs.py`)
`IPMC_BOOT_PROMPT`	`BOOT>`	bootloader command-mode prompt (Phase 12); same rules as `IPMC_APP_PROMPT`
`IPMC_BOOT_COUNTDOWN_BENCH`, `IPMC_BOOT_COUNTDOWN_FIELD`	3, 0	seconds a valid runtime image waits for `WAKEUP` on CLI or APP before it starts (0..=30); `IPMC_BOOT_COUNTDOWN` (in `ipmc-defs/src/lib.rs`) selects the field value in a `gen.ps1 build -Field` image (`cfg(ipmc_field)`); each second delays the first IPMB answer after a reset by one second; 0 compiles no countdown (section 15.3)

Define	Default	Meaning
`IPMC_FWU_PIN_SETTLE_US`, `IPMC_FWU_LED_WAIT_CHZ`, `IPMC_FWU_LOAD_IDLE_S`	100, 200, 30	bootloader: wait after configuring the BOOTLOADER pin before its one read (10..=10000 us); LD2 blink rate while it waits for `WAKEUP` or a command (centi-Hz); seconds without a line before `LOAD` ends with `ERR TIMEOUT` (1..=600)
`IPMC_BOOT_COMPARE_CHUNK`, `IPMC_BOOT_READ_CHUNK`	64, 256	bootloader stack buffers: bytes read back to verify a programmed record (1..=255) and bytes read per step of the SHA-256 image check (1..=1024)
`IPMC_CONSOLE_UART_NAME`, `IPMC_APP_UART_NAME`	`CLI`, `APP`	the names the firmware and the bootloader print for USART2 and USART1 (`SYS STATUS` lines `CLI UART:` and `APP UART:`, the `APP UART not started` line); 1..=8 printable ASCII bytes, no space; a PC's port number is never printed
`IPMC_APP_FRU_WRITE`	`true`	the application port may `FRU SET` the unit fields and run the IPMI Write FRU Data request; false refuses both with `ERROR WRITE_DISABLED` (the table `WRITE_REFUSED` in `commands.rs` lists the refused NetFn and command pairs), while `FRU GET` and the other `IPMI` requests (FRU Control, Set IPMB State, Set Event Receiver) still run, as from the Chassis Manager
`IPMC_MENU_LINE_MAX`, `IPMC_MENU_TOKENS_MAX`	128, 40	console line and token limits
`IPMC_FRU_DUMP_BYTES_PER_LINE`	16	bytes on one line of `FRU DUMP` (1..=16)
`IPMC_FW_NAME`, `IPMC_CONSOLE_PROMPT`	`ipmc-nucleo-f401re`, `CLI`	firmware name of the banner, `SYS VERSION`, the application port banner and its `INFO` answer; root prompt text (1..=16 printable ASCII bytes, no space; the menu appends the group path and `> `)
`IPMC_HSE_HZ`, `IPMC_SYSCLK_HZ`	8000000, 84000000	clock input and system clock (PLL values checked at compile time)
`IPMC_LED_*`, `IPMC_FAULT_BLINK_CHZ`	1 Hz heartbeat, 10 Hz activity, 5 Hz fault	LD2 patterns (centihertz)
`IPMC_WATCHDOG_ENABLE`	`true`	the independent watchdog runs; the main loop feeds it once per pass, and a hang, a panic or a processor fault resets the processor within `IPMC_WATCHDOG_MS`

Define	Default	Meaning
`IPMC_WATCHDOG_MS`	4000	watchdog period in ms at the nominal 32 kHz LSI (1000..=16000); the F401 guard requires the shortest real period (LSI at 47 kHz) to exceed twice the 500 ms FRU sector erase
`IPMC_ADC_INTERNAL_SOURCE`	`Source::Hw`	VDDA and die temperature from ADC1
`IPMC_ADC_EXTERNAL_SOURCE`	`Source::Sim`	rails A0..A3 simulated (unwired on the bench)
`IPMC_FAULT_SOURCE`, `IPMC_POWER_SOURCE`	`IPMC_SRC_HW`	fault inputs and PWR_EN on the pins
`IPMC_POWER_AT_BOOT`	`true`	request payload power at start-up (a board fills `StartInfo::power_at_boot` from it; the emulator uses `EmuConfig::power_on_at_start`)
`IPMC_PGOOD_TIMEOUT_MS`	1000	longest wait for PGOOD after PWR_EN
`IPMC_SENSOR_PERIOD_MS`	1000	sensor sample period
`IPMC_ADC_AVERAGE`	8	conversions averaged per ADC read
`IPMC_SENSOR_TRACE`	`TraceLevel::Events`	sensor console lines at start-up
`IPMC_RAIL0..4_*`, `IPMC_RAIL_NONCRIT_PCT`, `IPMC_RAIL_CRIT_PCT`	5 rails, 5 %, 10 %	rail names, nominal mV, dividers, windows; rail 4 is `3V3_AUX` (3300 mV, divider 2:1, A3 on PB0)
`IPMC_TEMP_*_C`	LNC 0, LC -10, UNC 70, UC 85, UNR 100	temperature thresholds
`IPMC_SIM_*`	25 C, every input pin high	simulator start values: PGOOD asserted, FAULT_N, OVERTEMP_N and PAYLOAD_RESET_N deasserted
`IPMC_EMU_PORT`, `IPMC_EMU_SERVE_POLL_MS`, `IPMC_EMU_TX_DONE_DELAY_MS`	5470, 1, 1	emulator TCP port, step period, delay of the Event Receiver's answer in the replay
`IPMC_EMU_VECTOR_LINE_MAX`, `IPMC_EMU_REPORT_LINES_MAX`, `IPMC_EMU_ERROR_ECHO_MAX`	512, 8, 80	emulator line and report limits

The four request and reset defines in the IPMB table are read by the link layer, the reset gate and the requester in every build; a Tier-1 build compiles them and sends no request. The FAST-MODE and payload defines are read in every build as well; the timing in `ipmc-defs` `i2c_timing.rs` is used only after a Set IPMB State with a 400 kHz speed byte (Tier-3), and the reset pulse and payload mode values only by the Tier-3 handlers and the Payload Mode sensor record.

A build with the extension also mounts the extension's own defines file (`defs_tier23.rs`) through the mechanism of section 4 (Build modes and the Tier-2/3 extension). Its defines, among them the Tier the build reports, the evaluation length and the

event log size, are listed in section 7.2 (Switches). The public `defs.rs` names no Tier-2/3 feature; a name defined in both files fails the build.

Pin assignments are not in `defs.rs`: they live in the owning board-support module (`drivers/ipmb_pins.rs`, `bsp/ga.rs`, `drivers/board_pins.rs`, `bsp/uart.rs`) and in `local_only/Documentation/nucleo_f401re_pinout.md`. Python bench-tool settings live in `tools/chassis-manager/cm_settings.py`; only values the IPMC and a tool must agree on (such as `IPMC_EMU_PORT`) are read by the tools from `defs.rs` through `tools/defs.py`.

6. Module Descriptions

Each module file opens with a comment that states its purpose, its owner rules and the specification tables it implements; the paragraphs below summarise them and say how the modules connect.

6.1 `ipmc-defs`, `ipmc-wire`

`crates/ipmc-defs/` (`build.rs`, `src/defs.rs`, `src/lib.rs`, `src/i2c_timing.rs`)

`src/defs.rs` is the configuration (section 5); `src/lib.rs` re-exports it, mounts the extension's `defs_tier23.rs` and `defs_guards.rs` under `cfg(tier23)`, and holds the compile-time guards. `build.rs` writes no file and reads one define, `IPMC_TIER23`. It is the one detection point of the Tier-2/3 extension (section 4, Build modes and the Tier-2/3 extension): it reads that line of `defs.rs`, the `IPMC_TIER23` override and `IPMC_EVAL`, checks the extension's marker files (`defs_tier23.rs` and `crates/ipmc-tier23/src/lib.rs`) and publishes the decision. `tier23_detect.rs` holds the pure rules (`parse_defs`, `parse_override`, `decide`, `parse_eval`, `check_eval`); `dep_cfg.rs` is the build-script step `ipmc-ext` and the firmware crate share.

`i2c_timing.rs` is the IPMB timing arithmetic, hardware-free and host-tested, shared by the bit-bang master, the hardware master and the bus-free waits. `cycles_for(ns, core_hz)` converts a time to core cycles, rounding up. `BitPhases::standard` and `BitPhases::fast` give the bit-bang SCL low and high phases in cycles (100 kHz: `IPMB_BITBANG_HALF_NS` each; 400 kHz: `IPMB_BITBANG_FAST_LOW_NS` and `IPMB_BITBANG_FAST_HIGH_NS`) with the slack above the minimum high time (4.0 us standard mode, 0.6 us fast mode, I2C-bus specification Table 5). `hw_ccr(pclk1, fast)` is the hardware master's CCR divisor (standard mode ceil(PCLK1 / (2 x 100000)); fast mode with F/S = 1 and DUTY = 0 ceil(PCLK1 / (3 x 400000)), RM0368 18.6.8) and `hw_trise(freq_mhz, fast)` its TRISE value (FREQ + 1 for 1000 ns; FREQ x 300 ns + 1 for 300 ns, RM0368 18.6.9). At PCLK1 = 42 MHz the divisors are 210 and 35 and TRISE is 43 and 13.

`crates/ipmc-wire/src/lib.rs`

The byte primitives shared by frames, FRU and SDR records and the tools: `zero_checksum` (the two's-complement checksum of IPMB frames and FRU areas, IPMI 13.8 and FRU spec sections 8, 11, 12), `Builder<N>` (a `const` byte-record builder: the FRU image is built at compile time; the SDR records are built by `SdrRepository::new` with the resolved address when `Context::new` runs), `Builder::push_type_length_ascii` (the IPMI type/length field, 8-bit ASCII), `crc32` (IEEE 802.3, used by the FRU flash record), `Sha256` (`new`, `update`, `finish`: the digest of the runtime image, section 16.2), `check_word` (the check word of the boot request, section 15.4) and `hex_byte` and `hex_word` (the hex-token parsers of the emulator, the bootloader and the tools).

6.2 `ipmc-fru`: FRU records, unit fields, scratchpad and flash store

`crates/ipmc-fru/src/records.rs`

Builds the 512-byte FRU 0 image at compile time from the `IPMC_FRU_*` strings: the 8-byte Common Header (version 1, Internal Use 00h, Chassis Info 00h, Board Info at offset 01h = byte 8, Product Info at offset 12h = byte 144 (90h), MultiRecord at offset 28h = byte 320 (140h), checksum C4h), the Board Info Area (136 bytes: language, the standard manufacturing date 000000h, manufacturer, product name, serial, part number, FRU file ID, C1h end marker, room for the fields, padding, checksum), the Product Info Area (176 bytes: language, manufacturer, name, part or model number, version, serial, asset tag, an empty file ID, C1h, room, padding, checksum) and the MultiRecord area with the one 14-byte Alternate Manufacturing

Date/Time record, 334 bytes in all (`IMAGE_LEN`), then 00h padding to `IPMC_FRU_SIZE`. The room is a run of 00h bytes after the C1h marker that FRU Information Storage Definition section 10 allows up to the area length: each text field reserves its `IPMC_FRU_*_MAX` bytes whatever the default string's length, so the pad at the end of an area is the sum over its fields of `MAX` less the default's length (58 bytes in the board area, 104 in the product area by default). A string the area cannot hold or an image larger than the area fails the build. The decoder (`common_header`, `area`, `Area::fields`, `multirecords`) and `verify` check a possibly rewritten image: header version and checksum, area versions, lengths, zero checksums, end-of-fields markers and the MultiRecord chain, and that the common header, the board area, the product area, the MultiRecord chain and the first byte of an Internal Use Area share no byte (`FruError::Overlap`): an image whose header offsets put one area on another fails `verify`, so a local field edit of it answers `Corrupt` (`FRU_IMAGE`) and changes nothing, instead of one field aliasing another.

The manufacturing date is the VITA 46.11 Alternate Manufacturing Date/Time record (Table 10.3.10-1, VITA record 34h), 14 bytes in the MultiRecord area:

Offset	Size	Content
0	1	record type ID C0h (OEM)
1	1	82h: end-of-list bit 7 set, bits 6:4 zero, record format version 2h
2	1	09h, the record length after the 5-byte header
3	1	record checksum: zero checksum of bytes 5..13
4	1	header checksum: zero checksum of bytes 0..3
5	3	`AC 81 00`, the VITA manufacturer ID 0081ACh, least significant byte first
8	1	34h, the VITA record ID
9	1	00h, the record format version
10	4	minutes since 1996-01-01 00:00 UTC, least significant byte first; 00000000h = unspecified

The standard 3-byte Mfg Date/Time of the Board Info Area (FRU spec Table 11-1) holds minutes since 1996-01-01 00:00 and ends at FFFFFFh, 2027-11-24 20:15; a Chassis Manager decodes only that field, so it cannot be widened. This IPMC therefore keeps the date only in the 32-bit field of the VITA 34h record, which covers 1996-01-01 00:01 to 9999-12-31 23:59, and leaves the standard field 000000h (unspecified) in the built image and after every write. A Chassis Manager that reads only the standard field sees "unspecified"; one that reads the VITA record sees the date. A read follows VITA 46.11 section 10.3.10: when the standard field is nonzero (a Chassis Manager wrote it) that field is the date in use and the record is not used; only while the standard field is 000000h is the date the record's. Minute 0 means "unspecified" in both places, so no date maps to it. The record uses the one checksum of `ipmc-wire` and the one const record builder (`Builder`).

`crates/ipmc-fru/src/date.rs`

The date text `YYYY-MM-DD HH:MM` (UTC) and the minutes count: `days_from_civil` and `civil_from_days` (Gregorian leap-year rule, so 2100-02-29 does not exist), `text_of`, `parse_text` and `DEFAULT_MINUTES`, all `const fn`, so the build parses `IPMC_FRU_MFG_DATE` with the same code the console and the application port use and fails on a date outside the range (1996-01-01 00:01 to 9999-12-31 23:59; 1996-01-01 00:00 is minute 0, "unspecified", and is refused). `DateRange` prints the accepted range in the bad-date answer of both ports.

`crates/ipmc-fru/src/unit.rs`

The FRU field editor (section 7.6 gives the protocol that calls it). `FruField` names the thirteen fields in the order `ALL` lists them: `BOARD_MFR`, `BOARD_PRODUCT`, `BOARD_SERIAL`, `BOARD_PART`, `BOARD_FILE_ID`, `MFG_DATE`, `PRODUCT_MFR`, `PRODUCT_NAME`, `PRODUCT_PART`, `PRODUCT_VERSION`, `PRODUCT_SERIAL`, `ASSET_TAG`, `PRODUCT_FILE_ID`; `max_len` is the room `IPMC_FRU_*_MAX` reserves (16 characters for the date text). `FruStore::set_unit_field` decodes the area, writes its fields back in the same order with the one field replaced, then C1h, 00h up to the area's old length and the zero checksum, so the area length never changes and no other area moves. The rewritten image is built beside the old one, passes `verify`, and only then replaces it. `MFG_DATE` changes the four minutes bytes of the VITA 34h record in place, rewrites its two checksums (also when they were wrong before: a record with bad checksums accepts the write and is sound afterwards) and sets the standard board field to 000000h (and the board checksum), so after a write the record is the date in use. `MFG_DATE` with an empty value clears the date: the record minutes are 00000000h, the standard field 000000h and the value reads `unspecified`. A value read (`MfgDate::effective`) is the standard field when it is nonzero, else the record's minutes when its checksums are good and they are not 0, else `unspecified`. An image without a MultiRecord area answers `AreaMissing` to a write, one without the record `NoRoom`; the area never moves. The date is the text `YYYY-MM-DD HH:MM` (UTC), 1996-01-01 00:01 to 9999-12-31 23:59. A refusal leaves the image as it was: `FieldError` has `TooLong`, `BadLength` (a text of one byte is reserved by the type/length rule), `NotAscii`, `NoRoom` (an image a Chassis Manager wrote may have no spare room), `Nvmro` (the firmware checks the pin, the store cannot see it), `BadDate`, `AreaMissing` and `Corrupt`. `write_unit_field` prints a field as stored; `mfg_date` returns both stored forms for `FRU SHOW`.

`crates/ipmc-fru/src/store.rs`

`FruStore` is the RAM scratchpad behind Read FRU Data and Write FRU Data: `read(offset, count)`, `contains`, `write(offset, data)` (sets a dirty mark), `take_dirty`, `bytes` and `image`. It holds the built image (`new`) or an image loaded from flash (`from_image`). No other crate holds FRU bytes. It also holds the Tier-3 non-volatile settings byte (bit 0 Default-Activation-Locked, bit 1 Commanded-Deactivation-Ignored, Rule 4.1.2-10.2): `nv_settings`, `load_nv_settings` (from the flash log, not dirty) and `set_nv_settings` (masked with `NV_SETTINGS_MASK`; a change sets the dirty mark, so the flash log saves it as it saves a Write FRU Data). The byte is 00h at power-up and stays 00h in a Tier-1 and Tier-2 build.

`crates/ipmc-fru/src/persist.rs` (tests in `persist/tests.rs`)

The FRU store is one flash sector (sector 2, 16 KB, `STORE_START` and `STORE_LEN`) behind the `FlashLog` trait (`sector_len`, `read`, `program`, `erase`, `millis`). `Persist::boot` scans the sector and loads the valid record with the highest sequence number; it erases nothing when it finds one, because the sector holds the only copy. A sector that is not blank and holds no valid record (damage only, nothing lost) is erased at boot unless NVMRO is asserted. `Persist::poll` is the save policy (dirty mark, quiet time, NVMRO, erase gate); `Persist::save` appends one record of the image and the settings byte, and when no slot is free it erases the sector and writes the record as the first one; `Persist::restore` writes the built image as a new record at once (console `FRU RESTORE`; the settings byte stays as it is). `store_region_ok` checks the linker's store region against the sector the log erases (section 9.1). Section 9.4 gives the record format and section 10.8 the save flow. The tests cut power at every word of a record and check the previous record still wins; the erase of a full sector is the one operation that can lose the image.

6.3 `ipmc-core`: IPMI logic

`crates/ipmc-core/src/ipmb.rs` - frame codec. `parse` checks length (7..=32 bytes), the header checksum, the address bytes (bit 0 clear), the response minimum and the data checksum, and returns a `Frame` (header, completion code for a response, data borrowed from the input). It rejects and never clamps; each failure is an `IpmbError` variant. `encode_request`, `encode_response` and `response_header_for` build frames with both checksums. A message is a response when its NetFn is odd.

`crates/ipmc-core/src/addr.rs` - geographic address. `resolve(raw_pins)` applies `IPMB_GA_SOURCE` and `IPMB_NVMRO_SOURCE` to the raw strap word (bit n = level of PC(n)): GA[4:0] are logic 1 when grounded, the hardware address is 40h + GA, the IPMB address twice that, parity must be odd over GA0..GA4 and GAP, and GA 0 is invalid. A forced GA gets a generated GAP. `AddrInfo::usable` is the one decision the link layer and the FRU initial checks read. `nvmro_asserted(raw)` is the one NVMRO rule, shared by start-up and run-time sampling.

`crates/ipmc-core/src/link.rs` - the dual-bus link layer (section 10.2). `Link` holds the bus states, per-bus counters, the ingress FIFO, a drop log, the duplicate cache, the egress queue and the bus of the latest transmission. Its main-loop API is `on_frame`, `poll`, `next_tx`, `on_tx_result` (section 11.1). A driver reports each transmission as a `TxOutcome`: `Sent`, or `Failed(TxFault)` with the cause the master saw (address NACK, data NACK, arbitration lost, bus busy, stretch timeout, line stuck, unspecified). A failure prints one line that names the destination, the sequence number and the cause, for example `IPMB-A response to 20h seq 01h not delivered, address 20h not acknowledged: no device answers at 20h; resend 1 of 3 on IPMB-B`, and a response given up on says `dropped after N resends`; a request the requester sends prints `sent`, `retry n of 10` or `not delivered` lines of the same form. `IPMC_IPMI_TRACE` false removes these lines at compile time. `plan_ipmb_state` is the one rule of Set IPMB State. `BusPair` power-up: IPMB-A Local Control, IPMB-B isolated, both isolated when the address is unusable. `Link` also owns the `Requester`: `on_frame` offers each valid response to it, `next_tx` hands out a response first and a due request only when the egress queue is empty, `on_tx_result` books a request transmission, and `poll` steps the requester timing, traces a request that ended and calls `Extension::poll` once. `LinkConfig::requests_allowed` is `IPMC_TIER >= 2`; a Tier-1 image does not reach the requester code. `with_first_request_seq`, `next_request_seq` and `recent_request_seqs` carry the request sequence number and the sequence numbers sent inside T9 across a processor reset. `Link::restarted(addr, now_ms)` is the one rebuild of a link for a Cold or Warm Reset: the same configuration and address, the power-up bus state, empty queues and caches, and the request sequence state of the old link carried over. Each `BusState` also holds a `BusSpeed` (`Standard` 100 kHz or `Fast` 400 kHz, power-up `Standard`, Rule 4.1.1-4.3); `plan_ipmb_speed` is the one rule of the speed byte (FAST-MODE on a bus that is not fitted is refused with `BusNotFitted`; STANDARD-MODE is always accepted) and `Link::set_ipmb_speed` applies it. `plan_ipmb_state` and `plan_ipmb_speed` share one loop, `plan_per_bus`. The driver owner reads `bus_state(bus).speed` and applies it between transfers (section 10.7).

The IPMB-B power-up window (`IPMB_COMPLIANCE`, `IpmbBPowerUp`, owner decision 10-27-2026): VITA 46.11 Rule 4.1.1-4 starts an IPMC that supports IPMB-B with IPMB-A enabled and IPMB-B disabled, Rule 4.1.8-1 lets it send only on IPMB-A after power-up or a cold reset, and Rule 4.1.8-2 lets a requester use only IPMB-A for a target until it knows that the target supports IPMB-B. `IpmbBPowerUp::Strict` (`IpmbCompliance::Strict`) is the rule as written: `on_frame` drops every frame on IPMB-B (`bus_unavailable`) until a Set IPMB State gives IPMB-B Local Control. `IpmbBPowerUp::NonStrict` (`IpmbCompliance::NonStrict`, the default) arms a window at power-up and after every Warm or Cold Reset when IPMB-B is fitted and the hardware address is usable (`Link::b_power_up_window`). While `Link::b_window_open` is true (the window is armed, IPMB-B is still isolated and has no local fault) a request that arrives on IPMB-B is accepted and carries a window mark (`Pending::window`, then `EgressFrame::window`, and the duplicate-cache entry stored for it) to its response. The request, or a response matched by the requester, is counted in `BusCounters::rx_power_up_window`; a frame that fails to parse is not. `next_tx` sends a marked response on IPMB-B whatever the `TxPolicy` says while IPMB-B is still in the window or has become enabled (`window_response_on_b`), so the Set IPMB State that ends the window, and any request accepted before it, is answered on the bus it arrived on; after an explicit Override or a local fault of IPMB-B a marked response goes by the policy, on IPMB-A. A retry of a marked request that arrives on IPMB-B is answered from the cache on IPMB-B in the same way. A failed transmission on IPMB-B is resent on IPMB-A. The bus state does not change: IPMB-B stays isolated, the IPMB Link sensor reports it so (Rule 9.3-10), and the IPMC sends nothing else on IPMB-B (no requests, no events, no responses to IPMB-A requests). The first Set IPMB State that sets IPMB-B (any value except FFh, no change) ends the window for good, so an explicit Override is honoured strictly; a Set IPMB State that only changes IPMB-A, and one that the plan refuses, leave the window open. A recorded local fault of IPMB-B closes it for good as well: clearing the fault does not reopen it, and only the next Warm or Cold Reset arms it again. The window departs from Rules 4.1.1-4 and 4.1.8-1 for those responses only. Under `NonStrict` the `TxPolicy` is `Arrival` as well: every response goes back on the bus of its request (a departure from Rule

4.1.1-6 for responses only); `next_request_tx`, which chooses the bus of the IPMC's own requests and events, keeps alternating, and the resend on the other bus after a failed transmission (Rule 4.1.1-7) is the same in both settings. The conformance case `LINK-ALTERNATION` (Rule 4.1.1-6) fails under `NonStrict` by design.

`crates/ipmc-core/src/requester.rs` - the IPMB requester, the one request this IPMC sends at a time. A Tier-1 IPMC sends none (Rule 7-4); a Tier-2/3 IPMC is an Event Generator (Rule 7-3) and its extension calls `submit(&Request, now_ms)`, which builds the frame (rqSA own address, rqLUN 0, next 6-bit rqSeq) and returns the sequence number. The module holds the request state only; it never touches a bus. The `Link` hands the frame to a driver and reports the outcome and the responses it receives. `Phase` runs Idle, Due, Sending, Backoff, Waiting, Cancelling and Done.

Rule	Value	Where
response wait, T3	`IPMB_REQ_TIMEOUT_MS` (250 ms) from the end of a successful transmission	Rule 4.1.1-9
retries	`IPMB_REQ_RETRIES` (10): 11 transmissions, the same sequence number each time	Rule 4.1.1-10, Observation 4.1.1-6
failed transmission, T6	`T6_MIN_MS` (60 ms) after a failed transmission the retry goes out on the other bus, when it is enabled and a transmission is left; otherwise the wait for a response starts and the same bus is tried at the next timeout	Table 4.1.1-1, Rule 4.1.1-7
retry bus	the bus other than the one of the request's own previous transmission, the other enabled bus when that one is disabled	Rule 4.1.1-6
new requests, T4	`IPMB_REQ_GAP_MS` (5 ms) from the end of the previous request's first transmission	Table 4.1.1-1
sequence reuse, T9	`SEQ_EXPIRY_MS` (5 s) after a sequence number's latest transmission; a blocked next sequence number is skipped to the next free one, in order, wrapping	Table 4.1.1-1
priority	a queued response always goes out before a request	link layer

`submit` refuses with `NotAllowed` (Tier-1 build), `Busy` (a request or an untaken completion exists), `TooSoon` (T4), `SeqNotExpired` (all 64 sequence numbers were sent less than T9 ago) or `TooLong` (the frame does not fit or a header field is out of range), and a refused call changes nothing. A response matches when destination, source, NetFn + 1, command, sequence number and both LUNs equal the request's. The request ends with an `Answer` (completion code and data) or `NoResponse` after the last transmission; `take_completion` returns it once. `cancel` drops the request and any untaken completion, lets a frame already on the wire finish its one transmission and keeps the sequence number used for T9; the extension calls it when the request lost its purpose. `Requester::new(allowed, first_seq)` starts the sequence count at `first_seq` and `with_recent_seqs` books the sequence numbers the previous run sent inside T9, so a Cold Reset continues the count and no number is reused before its T9 (section 8.2).

`crates/ipmc-core/src/reset_gate.rs` - `ResetGate`, the one decision when a pending processor reset runs. `poll(pending, now_ms, egress_empty, fru_clean, trace)` returns true when the egress queue is empty (the response left or was dropped) and the FRU store holds no unsaved change, or when `IPMC_RESET_WAIT_MS` has passed since the request was first seen; the second case prints one trace line naming what was still open. With no reset pending it forgets the first-seen time.

`crates/ipmc-core/src/tier.rs` - the Tier rules, each a `const fn` of a tier argument so a test checks all three Tiers while the build runs at one: `event_generator` (Tier 2 and 3, Rule 7-3), `sel_device` (Tier 3, Rule 7-8), `per_state_event_enables` (Tier 3), `activation_control` (Tier 3: the IPMC controls payload power and uses M5, Rules 4.1.2-10.1 and 4.1.5.4.3-2), `payload_mode_sensor` (Tier 3: sensor 7, Rule 6.1-17) and `fast_mode` (Tier 3: 400 kHz, Rule 4.1.1-4.4). The constants `EVENT_GENERATOR`, `SEL_DEVICE`, `PER_STATE_EVENT_ENABLES`, `ACTIVATION_CONTROL`, `PAYLOAD_MODE_SENSOR` and `FAST_MODE` apply them to `IPMC_TIER`.

`crates/ipmc-core/src/event_enable.rs` - `EventEnables`, one `SensorEvents` (events on or off, assertion mask, deassertion mask) per sensor of the Tier, 0..=6 and sensor 7 at Tier-3 (`sdr::sensor_count`). The power-up values come from the SDR event masks (`sdr::event_masks`), so the records and the enables cannot disagree: all off with zero masks at Tier 1, all on with the record masks at Tier 2 and 3. Get Sensor Reading reads status bit 7 from it; the Tier-2/3 code writes it.

`crates/ipmc-core/src/ext.rs` - the `Extension` trait (the hooks `command`, `poll`, `on_sample`, `warm_reset` and `power_on_allowed`), `NoExtension`, `feed_sensors` and `check_power_on` (section 4, Build modes and the Tier-2/3 extension).

`crates/ipmc-core/src/dupcache.rs` - `DupCache` keeps the response frame of every answered request for `IPMB_DUP_WINDOW_MS` under a `Key` of requester address, sequence number, NetFn, command, requester LUN, responder LUN and the exact request data. When every entry is live the oldest is evicted and counted. Time is a wrapping millisecond counter.

`crates/ipmc-core/src/queue.rs` - `Fifo<T, N>`, a fixed-capacity FIFO that rejects and counts on full and never overwrites, and `EgressQueue`, the response frames with their arrival bus, arrival time, assigned bus and resend count. The ingress FIFO and the drop log use `Fifo` too. `Bus` (A or B) with `other()` and `index()` (A is 0, B is 1, the index of the per-bus arrays) lives here.

`crates/ipmc-core/src/clock.rs` - `at_or_after(now_ms, at_ms)` and `earlier(a_ms, b_ms)`, the one wrap-safe comparison of the 32-bit millisecond clock (a difference below 2^31 ms counts as "after"); the emulator's serve loop, the payload reset pulse, the Tier-2 event timers and the firmware's receive-queue merge all use it.

`crates/ipmc-core/src/commands.rs` - the command table `COMMANDS`, one row per command: `cmd(netfn, cmd, name, spec, handler)`. Adding a command is adding one row (section 12.3); `find` is a linear scan over 13 rows. `tools/sync_ipmi_commands.py` keeps `Documentation/ipmi_commands.md` in step with this table, and a test fails while they disagree.

`crates/ipmc-core/src/dispatch.rs` - `Context` (section 11.2), the completion codes (`cc`), and the dispatcher. `run_request` answers C2h for a destination LUN other than 0 before any lookup; for NetFn 2Ch it echoes the request's defining-body byte as response data byte 0 on every path, answers C7h with no data for an empty request and C1h for a defining body other than 03h, and hands the handler the data after the VSO byte. `build_response_with` checks the output buffer before any handler runs; `dispatch_with` reserves an egress slot first, so a response that cannot be queued never causes a handler side effect. `run_command` looks the command up in the table; a (netFn, cmd) with no row goes to `Extension::command`, and only a miss there answers C1h. `Context::take_hand_offs(ext)` returns `HandOffs` (the payload power request, the payload reset request and the power-on check they meet) and clears them; the firmware main loop and the emulator both consume it. `Context` carries the Tier-2/3 hand-offs and state, each written by one owner (section 11.2): `pending_reset` (`Some(ResetRequest::Cold)` or `Some(ResetRequest::Warm)`, set by a handler and applied by the owner of the processor, section 8.2; `Context::warm_reset` clears the Context part of a Warm Reset), `pending_power` (a `PowerRequest`, On or Off, for the owner of the payload power), `pending_payload_reset` (a pulse request for the payload reset output), `deactivation_requested` (a Tier-3 standing request that `update_sensors` feeds to the FRU state machine, M4 to M5, only where `tier::activation_control`), `payload_mode` (the state of sensor 7, power-up mode 0 complete;

`PayloadMode` in `sensor.rs`), `event_enables`, and the settings byte accessors `nv_settings` and `set_nv_settings` over `FruStore`. `trace_request` writes one line per request, `FN(06h) CMD(01h) Get Device ID -> CC 00h, 15 bytes`, when `IPMC_IPMI_TRACE` is 1 and returns at once when it is 0. `Context::read_sensor` is the one function that maps a sensor number to its encoder, used by Get Sensor Reading and the console.

`crates/ipmc-core/src/cmd/` - handlers, one file per NetFn: `app.rs` (Get Device ID, Get Self Test Results), `se.rs` (Get Device SDR Info, Reserve Device SDR Repository, Get Device SDR, Get Sensor Reading), `storage.rs` (Get FRU Inventory Area Info, Read FRU Data, Write FRU Data), `vita.rs` (Get VSO Capabilities, Set IPMB State, Get Device Locator Record ID, Get FRU Address Info). Get VSO Capabilities builds the IPMB capabilities byte from the Tier and the fitted buses (`ipmb_capabilities`: bits 5:4 the maximum frequency, 00b 100 kHz, 01b 400 kHz at Tier-3; bits 1:0 the interface count minus one). Set IPMB State parses the optional speed byte per Tier (`parse_speed`) and checks it with `plan_ipmb_speed`, as section 10.7 describes. `mod.rs` holds `require_no_data` and `check_fru_device_id`. A handler writes only its data bytes into the slice it is given and returns `Ok(len)` or `Err(completion code)`.

`crates/ipmc-core/src/identity.rs` - `DEVICE_IDENTITY`, assembled from `ipmc_defs` values; the firmware revision comes from the crate version (0.1), and a guard ties `IPMC_FRU_PRODUCT_VERSION` to the same major.minor. No literal identity value lives here. `additional_support(tier)` builds Get Device ID byte 7 from the `IPMC_SUPPORT_*` defines and the Tier rules (section 9.6).

`crates/ipmc-core/src/sdr.rs` - `SdrRepository`: eight records (nine at Tier-3) built once from the `IPMC_SDR_*_NAME` strings and the IPMB address, record IDs 0..=7 (0..=8 at Tier-3, section 9.5), the reservation counter (`reserve`, `reservation_valid`), `resolve` (0000h first, FFFFh last), `next_id`, `describe` and `locator_record_id` (7; 8 at Tier-3). `mandatory_count(tier)` is 7, or 8 with the Payload Mode sensor (sensor 7) at Tier-3; the tables are sized for the most (`SENSOR_COUNT_MAX`, the mandatory sensors and the analog ones; `RECORD_COUNT_MAX`) and the records after the locator stay unused below Tier-3. `SdrRepository::with_analog` inserts the Full Sensor Records of the analog sensors between the compact records and the locator (the locator keeps the last record and takes the new sensor count as its ID); `Context::new` has none and `Context::with_analog` takes them. The hook is `analog::AnalogSensors` (`count`, `record`, `read`), the pattern of `ext::Extension`: `ipmc-core` decides nothing about a rail, and `ipmc-sensors` `analog.rs` implements it (section 6.4). `Context::read_sensor_at` answers the analog numbers (from `analog::ANALOG_FIRST`) through `read`, and `EventEnables::power_up_with_analog` gives them events off and masks 0000h. At Tier-3 the FRU State record adds M5 and the IPMB Link record adds the 400 kHz states (section 9.6). The file header lists every place the records differ from the captured live IPMC and why. Sensor initialization, sensor capabilities, the event masks and the locator's global initialization and device capabilities follow the Tier (section 9.6).

`crates/ipmc-core/src/sensor.rs` - the sensor numbers and types of Table 10.2-1, the `SensorSnapshot` that `ipmc-sensors` fills, and one encoder per reading layout: `encode_summary` (FRU Health, FRU Voltage), `encode_temperature`, `encode_link`, `encode_fru_state`, `encode_payload_test_results` (unavailable: no payload test runs) and `encode_payload_test_status` (not in progress), and at Tier-3 `encode_payload_mode` (sensor 7, four bytes) over `PayloadMode` (current mode, previous mode, `TransitionStatus`; modes 0..=14). `encode_link` sets the 400 kHz bits 4 and 5 of the link state from `LinkBus::fast`. `encode_threshold_reading` writes the three data bytes of an analog sensor (value, status, comparison bits). `apply_event_enable` is the one place that writes status bit 7. `LocalStatus` is the 3-bit IPMB local status of Table 10.2.2-1.

`crates/ipmc-core/src/fru_state.rs` - `FruMachine`, pure: `step(now_ms, inputs)` makes at most one transition and returns it; `check_power_on` refuses in M6. The states are M0, M1, M4, M5 and M6; M5 (FRU Deactivation Request) is reachable only at Tier-3 (`State::reachable_at`), through `Inputs::deactivation_requested`, which is false below Tier-3. `PowerOnRefused` names the two refusals (`DeactivationInProgress`, `ActivationLocked`). Section 10.4 has the diagram.

`crates/ipmc-core/src/selftest.rs` - `run(fru, sdr)` : `ipmc_fru::verify` on the image and a walk of the SDR chain (each record exists, carries the ID used to reach it, has a matching length byte, the walk ends at FFFFh after `record_count` records). Result 55h 00h, or 57h with bit 5 (FRU), bit 6 (SDR chain) or bit 3 (SDR empty).

`crates/ipmc-core/src/trace.rs` - the one `Trace` trait (`line(format_args!(..))`), implemented by the firmware console, the emulator's stdout sink, `NoTrace` and `RecordingTrace` (tests).

`crates/ipmc-core/src/ring.rs` - `ByteRing<N>`, the one byte FIFO of the project: N a power of two, N - 1 bytes usable (one slot stays empty), no allocator, every index masked and every slice access checked. The capture log stores its records in it and the firmware's UART driver keeps its receive and transmit queues in it.

`crates/ipmc-core/src/names.rs` - the names and bounded text the link layer and the printers share. `TxFault` (why a transmission failed) and `FaultMeaning` ; `Names` and `CoreNames` give a command its name (the Tier-1 table, then the extension, then Platform Event, the one request the IPMC sends); `NetFnName` and `CommandName` ; `format_cut` and `Bounded` format a line into a stack buffer and cut it with `...` ; `HexBytes` .

`crates/ipmc-log` - the capture log and all the logging logic of the DEBUG menu (section 10.11), so a person looks in one place. It depends on `ipmc-core` (`names.rs` , `ring.rs` , `ipmb.rs`); the core never depends on it.

File	Holds
`src/record.rs`	the record layout, `CaptureLog<N>` (a `ByteRing` of whole records), `DefaultLog` (the size of `IPMC_LOG_SIZE`), `Dir` , `FrameStatus` (`Received` , `Injected` , `Sent` or `Failed(TxFault)`), `Stats` , `Record` , `Kind` ; `frame` adds a whole IPMB frame, `text` a trace line cut at `IPMC_LOG_TEXT_MAX` ; a record that does not fit drops the oldest whole records first
`src/print.rs`	the printers `CsvHeader` , `CsvRow` and `TraceRow` (`Display` types, the same ones the live I2C trace uses)
`src/dump.rs`	`DumpCursor` (walks the records held when a dump starts and counts a record the ring overwrote meanwhile), `DumpFormat` (CSV or TRACE), `DumpHeader` (the first line), `DumpSink` (where a dump prints) and `DumpJob` (a few rows per main-loop pass while the sink has room for `ROW_MAX`)
`src/tee.rs`	`CaptureTrace` , the `Trace` that records every line and forwards it to the console when asked
`src/switches.rs`	`LogState` (the capture log, the IPMI console trace switch, the I2C trace switch), `TraceWhat` , and the `Display` types `LogLine` and `SwitchLines` for the status lines

6.4 `ipmc-sensors` : sensors and board I/O

`crates/ipmc-sensors/src/analog.rs` publishes each rail and the die temperature as an IPMI analog sensor: the Full Sensor Record (conversion, thresholds, units), the reading and the comparison bits, computed from the neutral rail table (name, nominal, windows; the channel and divider of each rail are the board crate's `RAIL_WIRING`) and the defs.rs rail and temperature defines beside the measurement definitions (section 9.5 lists the record). `BoardAnalog` implements the `ipmc_core::analog::AnalogSensors` hook.

`crates/ipmc-sensors/src/board_sensors.rs` is the place to start: it lists every value the board measures and how each Tier-1 reading is built. The board supplies one `RailWiring` (channel and divider) per rail as `RAIL_WIRING` in `drivers/board_pins.rs` ; `BoardParts::rail_wiring` carries it to the `Monitor` , which hands it to `build_snapshot` , and a compile-time check matches its length to the rail table. FRU Voltage is good when every rail of the board table is inside its

non-critical window; FRU Temperature carries the threshold flags of the die temperature; FRU Health is bad on FAULT_N, OVERTEMP_N, PWR_EN without PGOOD, a critical rail or a temperature at or above UC.

Layer	Files	Holds
Interfaces	`hal/analog.rs`, `hal/digital.rs`, `hal/power.rs`	`AnalogSource`, `DigitalInputs`, `PowerSwitch`, `ResetOutput` (the payload reset output), the shared `SourceError`
Evaluation	`eval/rail.rs`, `eval/temperature.rs`, `eval/faults.rs`, `eval/power_fsm.rs`, `eval/led.rs`	rail windows, calibration arithmetic (VREFIN_CAL, TS_CAL1, TS_CAL2), input assertion bitmap, PWR_EN/PGOOD state machine, LED level as a pure function of time
Drivers	`drivers/sim/`, `drivers/i2c/` (INA226, PCA9555), `drivers/spi/` (MCP3208)	trait implementations; the external parts are host-tested examples with no bus assigned
Routing	`routing.rs`	per source switch, the on-chip driver or the simulator
Sampling	`monitor.rs`, `trace.rs`, `reset_pulse.rs`	one sample: inputs, power state machine, PWR_EN drive, snapshot, change lines; `Monitor::apply_power_request` carries out a handler's `PowerRequest`; `PayloadResetPulse` times one payload reset pulse
Pin ownership	`pin_claims.rs` (table), firmware `bsp/ports.rs` (registers)	one owner per pin; a second claim fails naming both owners

`Monitor::apply_power_request(request, power_on_check, now_ms, io, trace)` is the one path from `Context::pending_power` to the PWR_EN drive, for the firmware main loop and the emulator. On asks the result of `ext::check_power_on` first (M6 or an activation lock): a refusal powers nothing and prints one line; Off is never refused.

`PayloadResetPulse` (`reset_pulse.rs`, hardware-free) times one pulse on the payload reset output (Rule 4.1.2-9). The owner calls `pulse_payload_reset(out, now_ms, ms)` when a handler set `Context::pending_payload_reset`, which asserts the output through `ResetOutput::set_reset`, and `service(out, now_ms)` on every main-loop pass, which releases it when the time is up (wrapping compare). Nothing waits, so no handler and no loop pass is held for the pulse; a second request restarts the time; a failed release is tried again on the next call. The STM32F401 driver (firmware `drivers/gpio.rs`) claims PA9 with the other board I/O as an open-drain output, released (high impedance) before the pin becomes an output, so the payload sees no pulse at start-up.

The crate forbids `unsafe`; the register access of the ADC, GPIO and pin ports is in the firmware crate (`bsp/ports.rs`, `drivers/adc.rs`, `drivers/gpio.rs`).

6.5 `ipmc-menu`: console engine

Function or type	What it does
`LineEditor::push_byte`	one received byte in; an `EditEvent` out (`Echo`, `Backspace`, `Line`, `LineTooLong`, or nothing)

Function or type	What it does
`Menu::process_line`	matches a line against the group tables, runs the handler, prints help or the usage text; returns an `Outcome`
`Menu::write_prompt`, `current_group`	the prompt with the group path
`Menu::validate_tables`, `syntax::validate_syntax`, `validate_table`	`const fn` table checks, run in compile-time assertions
`Args` (`len`, `get`, `text_at`, `u32_at`, `i32_at`, `on_off_at`), `optional`, `on_off_text`	the parameters of a command as typed values, with an `ArgError` for a bad one
`report_line_error`, `MenuError`	the text of a line that could not run
`eq_ignore_case`	the one case-insensitive keyword compare; the bootloader command table uses it too

`crates/ipmc-menu/src/` (`menu.rs`, `line.rs`, `args.rs`, `syntax.rs`; user documentation in `MenuDocumentation.md`). `LineEditor::push_byte` turns received bytes into echo, erase, line-complete and line-too-long events; `Menu::process_line` matches a line against the group tables (keywords case-insensitive and exact, `[REQUIRED]` and `{OPTIONAL}` parameters, optional last, and as the last token a `{NAME...}` that takes every remaining word, as in `DEBUG IPMI [NETFN] [CMD] {DATA...}`), prints help in fixed columns for `HELP` and `?`, and calls the handler with the caller's context and an `Args` view. A line holds at most `IPMC_MENU_LINE_MAX` (128) bytes and `IPMC_MENU_TOKENS_MAX` (40) tokens. `Menu::validate_tables` is a `const fn`; `ipmc-runtime/src/console.rs` calls it in a compile-time assertion (the associated const `Console::TABLES_VALID`, evaluated by `Console::new`), so a bad syntax string in the firmware tables fails the build.

6.6 `ipmc-nucleo-f401re`: firmware

File	Role
`src/main.rs`	the hardware start-up (section 8.3 steps 1 to 8: reset cause, clock, pins, UARTs, sensor hardware, straps, DWT counter, the IPMB master), the `StartInfo`, `BoardParts` and `SystemParts` literals and the call of `ipmc_runtime::run`, which runs the rest of the start-up and the main loop; `home`, the one `IpmbLink<F401>` in a static
`src/lib.rs`	the library root: `pub mod bsp`; the library holds only what both images use and no `core::fmt` is reachable from its always-linked code: the fault handlers write fixed text and the number writers of `ipmc_boot::text` through `ipmc_boot::text::SerialText` over a `bsp::uart::Port`
`src/board.rs`	`F401`, the `ipmc_hal::board::Board` and `Platform` of the NUCLEO-F401RE: names the zero-sized handles of `bsp/hal_impl.rs` and the runtime drivers for each role

File	Role
`src/drivers/ipmb_bus.rs`	`F401Ipmb`, the `ipmc_hal::ipmb::IpmbBus` of the two buses: owns the selected master, the clock and interrupt controllers the I2C slave start needs and the I2C peripherals until a bus starts; `bring_up` is the bus recovery then `i2c_hw::init` (a failure is `BringUpFailure::Init` with the `I2cError` name), `write`, `set_speed`, `lines_high` and `clear_bus_fault` go to the master, `suspend`, `suspend_faulted`, `resume` and `slave_stats` to `i2c_hw`
`src/sensing.rs`	`hardware_needs` (which on-chip hardware the source switches ask for) and the compile-time guard that refuses the I2C and SPI sources; the sensor service, the SENSORS, POWER and SIM commands and the Monitor are in `ipmc-runtime` and `ipmc-sensors`
`src/bsp/clock.rs`	HSE bypass (HSI fallback) to 84 MHz, PLL M 8 / N 336 / P 4 / Q 7, compile-time check of the PLL against `IPMC_SYSCLK_HZ`
`src/drivers/i2c_hw.rs`	I2C1 and I2C3: slave interrupts (EV and ER; the frame state is `ipmc_hal::frame::FrameAssembler`, the driver counts its events), the per-bus frame queues, `suspend` / `resume` for the bit-bang master, `set_speed` (CCR, TRISE and F/S for 100 or 400 kHz while the slave is idle), `HwMaster`, `clear_stuck_busy`, driver counters
`src/drivers/i2c_bitbang.rs`	`BitbangMaster`: START, bits, ACK, STOP on open-drain GPIO with DWT timing at the phases of the bus speed (`BitPhases`), clock-stretch read-back, arbitration check, bus recovery (9 clocks and a STOP, always at 100 kHz)
`src/drivers/ipmb_master.rs`	the `IpmbMaster` trait (with `speed` and `set_speed`), `MasterError`, counters, `SelectedMaster` chosen at compile time by `IPMB_MASTER_MODE`
`src/drivers/ipmb_common.rs`	the cycle-counter use (`now` and `start_cycle_counter` come from `bsp/cycles.rs`), `Deadline`, `wait_for`, `wait_bus_free` (tBUF, slave idle, stuck-BUSY signature), `write_retrying` (arbitration retries inside one T2 + T1 budget)
`src/drivers/ipmb_pins.rs`	claims PB8, PB9, PA8, PB4 and parks them as inputs; `BusPins` owns their three modes (`idle`, `drive`, `alternate`), each with the pull `IPMB_PULL_UPS` selects
`src/drivers/flash.rs`	the FRU store driver (`Flash`, the `FlashLog` implementation): sector 2 only, the range `map`, at start-up compares the linker symbols with the sector (`store_region_ok`) and refuses to start the store unless the region is exactly sector 2, the bootloader ends before it and the runtime starts after it; built over `FlashRegs`
`src/bsp/flash_regs.rs`	`FlashRegs`, the FLASH peripheral's owner: `erase_sector` (sectors 2 to 7), `program_words` (PSIZE x32), `program_bytes` (PSIZE x8), `read`; refuses sectors 0 and 1 and every address below `0x0800_8000` (`Err(Range)`) so no caller can reach the bootloader sectors; refuses erase and program while NVMRO is asserted; bounded BSY polls per sector size
`src/bsp/cycles.rs`	the DWT cycle counter: `now`, `start_cycle_counter`

File	Role
`src/bsp/boot_request.rs`	the four boot request words at `__boot_request` (the start of NOINIT), `read`, `write` and `clear` over `kept_words` (their meaning is `ipmc_boot::request`)
`src/bsp/kept_words.rs`	the one raw access to the NOINIT words that survive a reset: `read`, `write` and `clear`; the checked record of three data words and a check word (`ipmc_wire::check_word`) is `ipmc_hal::kept::store_checked` and `take_checked`; the callers own the words and the magic
`src/bsp/hal_impl.rs`	the `ipmc-hal` traits over the F401 modules, owning no hardware: `SysTickClock`, `DwtCycles { hz }`, `Uart<PORT>` (`CONSOLE`, `APP`) and `Port` as `SerialPort`, `GaStraps`, `Ld2`, the `KeptWords` handles `BootRequestWords`, `KeptSeqWords` and `FaultWords`, `CortexSystem` (`SystemControl`); `Watchdog` implements its trait in `bsp/watchdog.rs`
`flash_map.x`, `memory.x`, `link.x`, `boot.x`	the one flash and RAM map (BOOT, STORE, APP, NOINIT, RAM), the runtime region file and stack reservation, the runtime section placement and image asserts, and the bootloader script (section 9.1)
`src/app_header.rs`	the runtime image header static (`.app_header`, 64 bytes at `0x0800_C200`) and `read`, the 64 bytes read back from flash once at start for `StartInfo` (the runtime prints the `Image:` banner text from them)
`src/boot/`	the bootloader binary `ipmc-boot` (sectors 0-1): `main.rs` (reset cause, clock, pins, LED, console UART, then `ipmc_boot::run::boot`), `board.rs` (`F401Boot`, the `BootBoard` implementation), `start.rs` (the start sequence and the one `asm!` jump), `flash_image.rs` (`ImageSource` and `FlashOps` over `FlashRegs`); section 15.7 lists the functions
`src/bsp/ga.rs`	strap pins PC0..PC6 with pull-ups, raw level read, live NVMRO sample; PC10 BOOTLOADER (input, pull-down, owner `boot_pin`, printed in the banner and `SYS STATUS`)
`src/bsp/systick.rs`	1 ms tick, `millis`, `seconds`, `catch_up` (adds the time the core stalled on a flash erase, measured with the DWT counter)
`src/bsp/uart.rs`	the one UART driver of both ports (`Port::Console` on USART2, PA2/PA3; `Port::App` on USART1, PB6/PB7 and optional PA11/PA12), interrupt driven in both directions; described below the table
`src/bsp/led.rs`	LD2 on PA5: heartbeat, 10 Hz activity window after each IPMB frame, manual modes, fault blink
`src/drivers/board_pins.rs`	ADC channels and GPIO pins of the sensor signals, the payload reset output (PA9) and the PAYLOAD_RESET_N input (PA6)

File	Role
`src/bsp/reset_cause.rs`	reset-cause decode; `system_reset` (AIRCR SYSRESETREQ, so the next banner reads "software reset"); the four `.uninit` words `KEPT`, untouched by start-up code, that carry the next request sequence number and the mask of the sequence numbers sent inside T9 across a software reset (`ipmc_runtime::kept::keep_sequence` and `kept_sequence` store and read them through `KeptSeqWords`, with a check word, read back only after a software reset)
`src/bsp/fault.rs`, `static_slot.rs`	fail-stop handlers (HardFault, unhandled interrupt, panic) that record, print one line and park; the one `Mutex<RefCell<Option<T>>>` access pattern
`src/bsp/fault_record.rs`	the four kept fault words (kind, value, aux, check) at `__fault_record`, the last 16 bytes of NOINIT: `write`, and `take` (once at start-up)
`src/bsp/watchdog.rs`	the IWDG registers and the DBGMCU freeze bit `DBG_IWDG_STOP`: `setting` (prescaler and reload for a period, const), `Watchdog::start` (sets `DBG_IWDG_STOP` just before the KR start key), `feed`, `period_ms` (the `ipmc_hal::system::Watchdog` trait)
`src/drivers/eval_timer.rs`, `src/eval.rs`	evaluation image only (`cfg(ipmc_eval)`): the TIM2 run-time counter and the evaluation period check (section 8.4)

Files of `ipmc-runtime` (`crates/ipmc-runtime/src/`; all `no_std`, no `unsafe`, generic over `B: Board`):

File	Role
`run.rs`	`StartInfo` (reset cause, kept fault, clock summary, application port fault, the 64 image header bytes, the firmware version), `BoardParts<B>` and `SystemParts<P>` (the started services a board hands in), and `run`, the start-up order after the hardware and the main loop (section 8.3)
`console.rs`	`Console<B, L>` and `Context<B, L>` (`L: LinkHome<B>`: the link by `&'static mut` on the firmware, by `Box` in the emulator); the command tables as associated consts and the build-time checks `TABLES_VALID` and `HELP_FITS` (the longest HELP listing fits the transmit ring of `ipmc-hal`); the `SYS` and `LED` handlers; the banner; `poll` (returns the `Restart` that `SYS RESET` asked for, without a prompt), `service_app`, `service_sensors`, `service_link`, `restart`
`data_cmds.rs`	`FRU SHOW`, `FRU DUMP`, `FRU STORE`, `FRU FIELDS`, `FRU GET`, `FRU SET`, `FRU RESTORE`, `SENSORS SDR`
`debug_cmds.rs`	the `DEBUG` group (section 10.11) except the logging commands: `STATUS` (prints the `ipmc-log` status lines), `EVENTS`, `IPMI`, `INJECT`, `CACHE`, `BOOTLOADER`, `WATCHDOG`; `events_off_at_boot`
`log_cmds.rs`	the logging half of the `DEBUG` group (section 10.11): `LOG`, `DUMP`, `CLEAR`, `TRACE`, each a thin wrapper that parses its words, calls `ipmc-log` and prints its `Display` lines; the paced dump writes through `DumpOut` over the console port
`sensing_cmds.rs`	the `SENSORS`, `POWER` and `SIM` commands over `Sensing`

File	Role
`ipmb_cmds.rs`	the `IPMB STATS` and `IPMB STATE` commands: read-only views of `IpmbLink` (the link counters, the driver and master counters through `IpmbLink::ipmb()`, the fault streaks through `fault_state`)
`app_port.rs`	the transport of the application port (section 7.6): `AppPort::poll` takes bytes from the application port, feeds the `ipmc-app` line editor, queues the echo, the answer and the prompt in one output buffer and drains it into the transmit ring without waiting; `LinkHost`, the `AppHost` implementation over the link layer
`link.rs`, `fru_log.rs`, `sensing.rs`, `kept.rs`, `text.rs`	the IPMB link service, the FRU log owner, the sensor service, the kept request sequence and the console text layer (the main functions are listed below)

Main functions of the runtime binary and the library:

Module	Main functions	What they do
`main.rs`	`main`, `home`	the hardware start-up of section 8.3, then `ipmc_runtime::run`; `home` keeps the one `IpmbLink<F401>` in a static (`#[inline(never)]`, so the 17 KB temporary of `singleton!` lives in its own frame, which is gone when it returns)
`drivers/ipmb_bus.rs`	`F401Ipmb::new`, the `IpmbBus` methods	the master, the I2C slaves and their peripherals behind the neutral bus trait
`ipmc-runtime` `link.rs`	`start_buses`, `IpmbLink::new`, `with_config` (a link configuration other than the defs.rs one, for the emulator's board variants), `service`, `kept_state`, `feed_sensors`, `power_on_check`, `take_hand_offs`, `service_fru_log`, `set_unit_field`, `fru_restore`, `request_bootloader`, `local_request`, `local`, `inject`, `log_text`, `stop_for_eval`, `restart`	start the buses and build the Link and Context; one IPMB step per main-loop pass, which returns a `Restart` once a Cold Reset runs; hand the sensor snapshot to the core; the payload power check and the power and reset hand-offs; the FRU save step with the erase pause; the one unit-field write path; the boot request and the processor restart (`restart`, the one function that flushes both ports, keeps the sequence, writes the boot request words and resets); the local IPMI request and frame injection of the DEBUG group; the capture log entry points
`sensing.rs`	`hardware_needs`	which hardware the source switches need
`ipmc-runtime` `fru_log.rs`	`FruLog::boot`, `parts`, `status`	load the newest valid FRU record before the I2C slaves start; own the flash and `Persist`

Module	Main functions	What they do
`ipmc-runtime` `sensing.rs`	`Sensing::new`, `start`, `service`, `sample_now`, `serve_hand_offs`, `take_fresh`, `payload_power_on`	build the routers over the backends and the simulators; sample on the period; apply power requests and the payload reset pulse; each call that prints takes the console and the time of the pass
`ipmc-runtime` `text.rs`, `kept.rs`	`ConsoleText::print`, `line_or_drop`, `row_or_drop`, `ConsoleTrace`, `MenuOut`, `DumpOut`; `keep_sequence`, `kept_sequence`	formatted console output over any `SerialPort`; the request sequence state across a software reset
`ipmc-runtime` `run.rs`	`start`, `Runtime::pass`, `Runtime::restart`, `run`	the start-up order after the hardware (`start`), one pass of the main loop (`pass`; a restart ends it and is returned) and the endless loop that is `start` and `pass` (`run`), the same on every board; a host test calls `start` and `pass` itself
`ipmc-runtime` `sim.rs` (feature `sim`)	`Sim` (`Board` and `Platform`), `SimIpmb`, `SimRx`, `IpmbState`, `SimPins`, `SimSystem`, `SimWatchdog`, `SimExt`, `parts`, `Handles`; `SimSerial`, `SimClock`, `SimCycles`, `SimStraps`, `SimLed` from `ipmc-hal` `testing`	the simulated board: settable clock and cycle counter, serial ports that record and take bytes, per-bus frame queues with an outbox of written frames (bus, bytes, outcome), a write-outcome policy (per bus, over both buses, and scripted outcomes), scripted line levels and suspend results, the RAM flash as the FRU store, the simulated analog inputs and board I/O, the extension held boxed so a test can replace it (`SimExt`); a reset is an unwinding `Reset` that runs no panic hook
`ipmc-runtime` `console.rs`	`Console::new`, `poll`, `service_app`, `service_sensors`, `service_link`, `restart`, `service_fru_log`, `banner`, `prompt`	the command tables, the `SYS` and `LED` handlers, the banner, and the per-pass service calls of the main loop; `poll`, `service_app` and `service_link` are `#[inline(never)]`, so the locals of a step (the link's buffers, the line parser, the application port's answer) live in the step's own frame and not in the frame of `run`
`ipmc-runtime` `debug_cmds.rs`, `log_cmds.rs`, `data_cmds.rs`, `sensing_cmds.rs`, `ipmb_cmds.rs`	`cmd_debug_*`, `cmd_fru_*`, `cmd_sensors_*`, `cmd_power_*`, `cmd_sim_*`, `cmd_ipmb_*`, `events_off_at_boot`	the DEBUG group (including `cmd_debug_bootloader`), the logging commands over `ipmc-log`, the FRU and SDR listings, the SENSORS, POWER and SIM groups, the IPMB views

Module	Main functions	What they do
`ipmc-runtime` `app_port.rs`	`AppPort::poll`, `LinkHost`	move bytes between the application port rings and `ipmc-app`; the `AppHost` implementation over the link layer
`app_header.rs`	`read`	the runtime image header static and the 64 header bytes read back from flash for the `Image:` banner text
`drivers/i2c_hw.rs`	`init`, `suspend`, `resume`, `set_speed`, `slave_idle`, `clear_stuck_busy`, `stats`	the I2C slave interrupts, the per-bus frame queues, the pause for the bit-bang master and for an erase
`drivers/i2c_bitbang.rs`, `drivers/ipmb_master.rs`, `drivers/ipmb_common.rs`	`BitbangMaster::new`, `IpmbMaster::write`, `write_retrying`, `wait_bus_free`, `wait_for`, `Deadline`	one master write on open-drain GPIO or the I2C peripheral, with the bus-free wait and the arbitration retries inside the T2 + T1 budget
`drivers/flash.rs`	`Flash::new` (the `FlashLog` implementation)	the FRU store driver over `FlashRegs`, sector 2 only
`bsp/flash_regs.rs`	`FlashRegs::erase_sector`, `program_words`, `program_bytes`, `read`; `read_flash`	the one owner of the FLASH registers; refuses sectors 0 and 1 and addresses below `0x0800_8000`, and refuses erase and program while NVMRO is asserted
`bsp/uart.rs`	`init`, `stop`, `getc`, `write`, `write_all_wait`, `flush`, `discard_rx`, `count_line_dropped`, `counters`	the one UART driver of both ports
`bsp/clock.rs`, `bsp/systick.rs`, `bsp/cycles.rs`, `bsp/led.rs`	`clock::init`, `restore_reset_state`; `systick::millis`, `catch_up`; `cycles::now`; `led::activity`, `set_blink_chz`, `set_fault`	the clock tree, the millisecond time, the DWT counter, LD2
`bsp/reset_cause.rs`, `bsp/boot_request.rs`, `bsp/ga.rs`	`read`, `take`, `system_reset`; `boot_request::read`, `write`, `clear`; `ga::init`, `read_raw`, `nvmro_asserted`, `bootloader_pin_high`	the reset cause and the words kept across a software reset, the boot request words, the strap, NVMRO and BOOTLOADER pins

The UART driver (`bsp/uart.rs`) runs one code path for both ports; a `Port` value selects the peripheral, the pins, the ring sizes and the baud rate. Each port has a receive and a transmit `ByteRing`. One interrupt body (`service`) moves a received byte into the receive ring and sends the next byte of the transmit ring; the main loop reads with `getc` and writes with `write`, which copies as much as the ring takes and never waits. `ConsoleText::print` (`ipmc-runtime`, the `out!` macro) writes a whole console message through `write_all_wait` and waits only while the ring is full; `line_or_drop` writes a trace line and its CR LF whole when the ring has room for all of it, and otherwise drops it and counts it (`tx_lines_dropped`), so a trace line never stalls the IPMB; `row_or_drop` does the same for the longer per-frame I2C trace row. `write` copies at most 64 bytes

per critical section and loops. `flush` waits, bounded, until the last byte has left; it runs before a reset (the console and the application port) and before the fault park. `discard_rx` empties the receive ring of a port in one critical section and starts a reader stopped by flow control again. Every wait polls the same `service` body, so it also ends where the interrupt cannot run (a HardFault). Receive overflow, overrun, transmit timeout and dropped lines are counted per port and printed by `SYS STATUS`. The interrupts run at NVIC priority 2 << 4, below the I2C slaves (1 << 4). The interrupt body runs inside the critical section of the port state, so an I2C interrupt that arrives meanwhile waits for it, a few microseconds. With `IPMC_APP_UART_FLOW` true the driver enables RTS/CTS in the peripheral (CTSE and RTSE, RM0368 19.3.14) and leaves a byte in the data register when the receive ring is full, so RTS stays deasserted until `getc` makes room. RTS deasserts only when the data register holds an unread byte, so a sender with a byte in flight can still overrun by one byte; the overrun counter counts it and the application port answers the next line end `ERROR LINE_DAMAGED` after discarding the receive ring (`discard_rx`); PA11 reads "not clear to send" when it is open.

6.7 `ipmc-emu`: x86 emulator

An `Emulator` is a board that runs the firmware logic: `ipmc_runtime::run::start` and `Runtime::pass` (the code the firmware runs, section 6.6) on the simulated board `ipmc_runtime::sim::Sim`. The console and the application port are text buffers, the IPMB buses are frame queues with a record of every frame written, the FRU flash is an `ipmc_fru::RamFlash`, the sensors read the simulators, and a processor reset is an unwinding `Reset`. The emulator holds no IPMI logic and no orchestration of its own: the console, the link service (drain of the two queues, `Link::poll`, one write per pass, the bus-fault bookkeeping, the FRU save with the erase pause, the reset gate), the sensor service and the hand-offs are `ipmc-runtime`'s.

Main functions: `Emulator::new`, `new_with_flash`, `rx`, `poll`, `take_sent`, `console_line`, `flash`, `with_extension`, `set_fail_every`, `set_bus_down` (the board, driven by the caller); `replay::replay` and `ReplayReport` (capture replay and its classes); `serve::Board::handle` and `step`, `serve::run` (the real-time board behind the TCP line protocol); `vectors` (the vector reader); `main` (`replay`, `serve`, `sdr`).

File	Role
`src/emulator.rs`	`Emulator`: the started `Runtime<Sim, Box<IpmbLink<Sim>>>` and the handles of its `Sim`; `rx` puts a frame in the queue of a bus, `poll` runs the main loop, `take_sent` hands over the frames the master wrote
`src/vectors.rs`	the one reader of the vector format (`local_only/vectors/sbc_ab.vec`: time, bus tag, hex bytes)
`src/replay.rs`	feeds each captured request at its capture time, takes the frames the board wrote, compares each response with the captured one: Match, SameCC, CCDiff, NotImpl, Unans, NoCap, NoBus
`src/serve.rs`	`Board`, the emulator in real time behind the TCP line protocol (section 7.4); one client at a time; a new client or `#reset` starts a fresh board
`src/trace.rs`, `src/main.rs`	stdout trace sink and hex formatter; command line `replay`, `serve`, `sdr`, `help`

The caller supplies what the hardware supplies. `rx(bus, bytes, now_ms)` is the slave driver's frame at a STOP: it joins the queue of the bus with `now_ms` as its STOP time (`Err(BusUnavailable)` when no target runs on the bus, as IPMB-B in a build without it; `Err(IngressFull)` when the queue holds as many frames as the board's). `poll(now_ms)` sets the board's clock and runs the main loop: `follow_payload_supply` (PGOOD follows PWR_EN while `IPMC_SIM_PGOOD_HIGH` is 1; with 0

PGOOD stays low, so the PGOOD timeout path stays reachable), then one `Runtime::pass` (watchdog, console, application port, sensor sample and hand-offs, `IpmbLink::service`), at least twice and then until a pass writes no frame, so a response and the requests the IPMC raises with it leave in the same call. The console text of every pass goes to the caller's `Trace` line by line. The board's time never runs back (an earlier `now_ms` is raised to the latest one given).

A frame the master writes goes to the outbox of `SimIpmb` with the outcome the simulated bus gave it: `set_fail_every(n)` fails every nth write over both buses and `set_bus_down` fails every write on one bus, both with an address NACK, so `IpmbLink` books the failure with `on_tx_result` and the link resends on the other bus (Rule 4.1.1-7). `take_sent` returns the outbox as `Tx { bus, bytes, ok }`. A write happens in the pass that queued the response, at the pass clock; the replay reports a transmission done `IPMC_EMU_TX_DONE_DELAY_MS` after the request only for the Event Receiver's answer to the IPMC's own request.

The configuration choices of a board (`EmuConfig`) reach the runtime through the `home` closure of `start`, which builds the link as the firmware's `home` does: `IpmbLink::with_config` with the configuration's `LinkConfig` (IPMB-B fitted, response policy, power-up behaviour), IPMB-B brought up when fitted, IPMB-B enabled when configured, the extension from the factory (`Emulator::with_extension`; `SimExt` holds it boxed). `power_on_at_start` and `events_at_start` go into the `StartInfo` of the board parts (`power_at_boot`, `events_at_boot`), the start data the runtime reads. The straps read `EmuConfig::strap`.

A Cold Reset is the board's restart: `IpmbLink::service` returns `Restart`, `Emulator::poll` calls `Runtime::restart` (both ports flushed, `keep_sequence` into the `[u32; 4]` kept words, the boot request, then `SimSystem::system_reset`, which unwinds with `Reset`), catches the unwind and starts a new run with `start` over the flash of the run that ended and the kept words. Everything is new: link, context, extension, sensors and payload power (`StartInfo::power_at_boot` requests power again, the stored default activation lock keeps the payload off), the FRU store comes from the flash (`FruLog::boot`; an accepted write not saved by then is lost, as on the board), and the board's clock counts from the call that reset it, as the tick counts from a processor reset, so the kept request sequence state counts as sent at 0 ms of the new run (`Link::with_first_request_seq`). A Warm Reset is `IpmbLink::warm_reset`, section 8.2: the board's own, so IPMB-B returns to its power-up state whatever the configuration enabled at start. The flash is the `RamFlash` under the log of the firmware (`Persist` through `FruLog`), the same save rule: an accepted Write FRU Data or a settings change is saved `IPMC_FRU_PERSIST_DELAY_MS` after the last one, at once while a reset is pending. `Emulator::flash` returns it (completing a pending save first, as a clean power cycle does) and `Emulator::new_with_flash` starts a board from it, so a power cycle of the emulator keeps the flash log.

`console_line` types a line into the console (a person at the terminal). The firmware passes `IPMC_EVENTS_AT_BOOT` and `IPMC_POWER_AT_BOOT` in `StartInfo` (`events_at_boot`, `power_at_boot`); with `events_at_boot` false the runtime sends itself the Set Event Receiver FFh that turns the events off and prints `Events: off from start-up`. The emulator and the conformance runs start with events on (PLAN decision 24): `EmuConfig::events_at_start` (true by default) sets `events_at_boot`, so the runtime sends no Set Event Receiver and prints no events line, and the extension never sees the off-and-on cycle that would resend the link and payload mode events. `replay` also plays the Event Receiver role of the captured Chassis Manager: a request the IPMC sends to 20h is answered one step later, 00h for a Platform Event and C1h for anything else, and counted in the summary line (`ipmc_requests`, `events`). Without the extension no request is sent and the counts are 0.

6.8 `ipmc-app`: application port protocol

`crates/ipmc-app/src/` (`lib.rs`, `commands.rs`, `engine.rs`, `host.rs`, `line.rs`, `tests.rs`; the module comment of `lib.rs` is the protocol description and the recipe for a new command)

The crate is `no_std` and `forbid(unsafe_code)`; it never touches hardware, and it does not depend on `ipmc-menu`. It is a reference crate for a customer application (PLAN decision 27) and may repeat code of the other crates; section 7.6 (Connecting a payload processor) is the part for an engineer who wires a payload processor to it. A byte travels `UART byte -> serve -> LineEditor::push -> respond -> engine::run -> COMMANDS row -> handler -> AppHost`:

File	Holds
`line.rs`	`LineEditor<N>` collects bytes into lines: it echoes printable bytes, erases with BS SP BS on backspace (08h, 7Fh), ends a line at CR, LF or CR LF (one line end), reports an empty line, a line longer than `N`, a byte that is not printable ASCII and lost bytes (`damage`) once, at the line end, and drops the bytes; `AnswerBuf<N>`, the fixed buffer the echo, the answer and the prompt are formatted into
`engine.rs`	the command engine, in the style of the MenuBuilder menu: `Row` (syntax string, one sentence of help, handler), `Syntax::parse` and `check_table` (the table rules), `find` (case-insensitive whole keywords, the longest keyword sequence wins, the match is a value, not a return code), `Args` (the parameters of a row), `Fail` (code and sentence of an `ERROR` line), `write_help` (the aligned table), `run` (the tag, the row, the handler, the final line; a final line or error line that does not fit the buffer rolls the answer back to its start and answers `ANSWER_TOO_LONG`), `Quoted` (a request quoted in a sentence: at most `QUOTE_MAX` = 32 bytes, then `...`), `MAX_PARAMS` (the largest parameter count of any row of `COMMANDS`, computed at compile time), `final_line` (the parsing rule of the payload CPU)
`commands.rs`	`COMMANDS`, one `Row` per command; the handlers; `banner`; `respond` (answers what the editor returned, including the `LINE_TOO_LONG`, `BAD_CHAR` and `LINE_DAMAGED` answers for a line that cannot run, and ends every answer with the prompt); `ANSWER_MAX` (the answer buffer size, computed from the table and the field lengths, see section 7.6); `serve` (one received byte: its echo, and at a line end the answer)
`host.rs`	the `AppHost` trait, the one door to the board: `version`, `status`, `fru_state`, `unit_field`, `set_unit_field`, `ipmi`, `command_name`, `save_state`, `request_bootloader`; and the plain types it passes (`Status`, `SaveState`, `AppError`, `IpmiData`)
`tests.rs`	`REQUEST_TABLE`, one row per request line with the answer the test host must produce; `HELP_TABLE`, the exact help text; the engine, editor and session tests; the test host (`FakeHost`) is an `AppHost` over a plain `FruStore`

The runtime implements `AppHost` as `LinkHost` in `ipmc-runtime` `app_port.rs`, over the IPMB link layer: the dispatch `Context` (addresses, FRU image, FRU state, latest sensor snapshot), the flash log status, the live NVMRO pin, the capture log's command names and the SysTick time. A handler returns without waiting for hardware and never runs in an interrupt. Section 7.6 gives the lines the port accepts and answers.

6.9 `ipmc-boot`: bootloader

The bootloader crate and its binary are section 15 (modules and main functions in section 15.1, the firmware glue in section 15.7); the image header, the digest and the update procedure are section 16.

6.10 Tools

Path	Role
`tools/trace2vec.py`	merges the two logic-analyser I2C exports (IPMB-A and IPMB-B, one time base) into `local_only/vectors/sbc_ab.vec` with bus tags, redacted
`tools/sync_ipmi_commands.py`	rewrites the implemented column of `Documentation/ipmi_commands.md` from `COMMANDS`; `--check` for the gate
`tools/captured_sdr.py`	decodes the 40 captured SDRs next to ours into `local_only/Documentation/captured_sdr.md` (local only)
`tools/dupcheck.py`	scores every pair of Rust function bodies with identifiers normalised; fails on a production pair at or above 0.85 or a test pair at or above 0.92
`tools/capture/`	`capture2pcap.py` turns a saved `DEBUG DUMP` into a pcap file (LINKTYPE 260) and a CSV file (section 10.11); `test_capture2pcap.py`
`tools/fwupdate/`	`fwupdate.py`, the host update tool of the bootloader (section 16.5): wake, pre-check, erase, stop-and-wait `LOAD`, `VERIFY`, `RUN`; `fwu_settings.py`; `test_fwupdate.py` with a fake bootloader
`tools/app-port/`	`app_cli.py`, the bench client of the application port (section 7.6): tagged requests (the tags start from the clock) from the command line, an interactive prompt or a raw terminal (`--raw`) on COM5, exit status 0 when every answer is `OK`; `app_settings.py` (port, answer timeout, tag form, the prompt and the line limit read from `defs.rs`); `test_app_cli.py` with a fake transport
`tools/python-ipmi/`	venv with python-ipmi and pyaardvark; `README.md` is the Tier-1 command-line reference
`tools/chassis-manager/`	VITA 46.11 Chassis Manager on python-ipmi: `cm_link.py` (requester rules: T3 250 ms, retries with the same sequence, alternation, per-target IPMB-B), `cm_commands.py`, `chassis_manager.py`, `crosscheck.py` (compares a run with the captured Chassis Manager, every difference tagged), `ipmc_emu_interface.py` (TCP transport), `cm_settings.py` (tool settings), `cm_events.py` (the Chassis Manager as Event Receiver: answers each Platform Event request and records its decoded fields), `test_cm_link.py`
`tools/chassis-manager/` options for events	`--cm-tier 1|2` (2: Event Receiver, answers a Platform Event with 00h; 1: not an Event Receiver, answers C1h), `--withhold-events N` (the first N Platform Event frames get no answer, retries count, so a run shows the IPMC retrying with the same sequence number), `--set-event-receiver` (sends Set Event Receiver to each IPMC found after discovery; a Tier-1 IPMC answers C1h, which is not an error), `--cold-reset-at-s T` (sends Cold Reset once to each IPMC found, T seconds into the polling)
`tools/conformance/`	spec-only Tier-1 suite: `TRANSPORT.md` (its only interface), `adapter.py` (maps it onto the Aardvark or the emulator), `run_conformance.py`, eight `tests_*.py` files, `REQUIREMENTS.md`

Path	Role
`tools/board_test_report.py`	runs the command line, Chassis Manager, conformance, link-health and FRU persistence checks against the board and writes `local_only/Documentation/board_test_report.md`

7. Interfaces

7.1 IPMB frame format

All addresses are 8-bit (the 7-bit I2C address shifted left, bit 0 clear); byte 0 is the destination as it appears in the I2C address phase. The I2C slave interrupt prepends the matched address byte (own address, or 00h for a general call) to the bytes it receives, so the link layer always sees a whole frame.

Byte	Request	Response
0	rsSA	rqSA
1	NetFn << 2 | rsLUN	(NetFn + 1) << 2 | rqLUN
2	chk1: bytes 0..2 sum to 0 mod 256	chk1
3	rqSA	rsSA
4	rqSeq << 2 | rqLUN	rqSeq << 2 | rsLUN
5	command	command
6..	request data	completion code, then response data
last	chk2: bytes 3..n-1 sum to 0 mod 256	chk2

A request is 7 to 32 bytes, a response 8 to 32 (`IPMB_MSG_MAX`). For NetFn 2Ch/2Dh the first data byte is the defining body, 03h for VSO; "byte 3" of a VITA response table is response data byte 0 after the VSO byte.

7.2 Command set

The commands the public Tier-1 build implements are the rows of `COMMANDS` in `crates/ipmc-core/src/commands.rs`, the 13 Tier-1 mandatory commands and the broadcast form of Get Device ID (`Documentation/ipmi_commands.md`, section 1, is generated from the table):

NetFn	Cmd	Command	Request data	Response data (after CC)	Notes
06h	01h	Get Device ID	none	15 bytes from `DEVICE_IDENTITY`	also the broadcast form
06h	04h	Get Self Test Results	none	55h 00h, or 57h and failure bits	
04h	20h	Get Device SDR Info	none or operation	count, flags 01h	07h sensors, or 08h records with bit 0 set; 08h sensors and 09h records at Tier-3

NetFn	Cmd	Command	Request data	Response data (after CC)	Notes
04h	21h	Get Device SDR	reservation, record ID, offset, count	next ID, bytes	C5h for a non-zero offset without the current reservation; CBh no record; C9h range; CAh too large
04h	22h	Reserve Device SDR Repository	none	reservation ID	each call returns a new non-zero ID
04h	2Dh	Get Sensor Reading	sensor number	reading, status, state	CBh for a number above 6 (above 7 at Tier-3, section 9.3)
0Ah	10h	Get FRU Inventory Area Info	FRU ID	00h 02h 00h	512 bytes, byte access
0Ah	11h	Read FRU Data	FRU ID, offset, count	count, bytes	C9h count 0 or range; CAh too large for one frame
0Ah	12h	Write FRU Data	FRU ID, offset, bytes	count written	80h while NVMRO is asserted; reruns the self test; saved to flash after the quiet time
2Ch	00h	Get VSO Capabilities	VSO	03h, 00h, IPMB capabilities, 00h, 22h, 00h, 00h	IPMB capabilities 00h (one interface) or 01h (two), 100 kHz; 10h or 11h at Tier-3, 400 kHz (section 9.6)
2Ch	09h	Set IPMB State	VSO, A state, B state, optional speed	03h	CCh for an unsupported link ID, speed, unfitted bus or both buses isolated (section 10.7)
2Ch	0Dh	Get Device Locator Record ID	VSO, FRU ID	03h 07h 00h	record 0008h (03h 08h 00h) at Tier-3
2Ch	40h	Get FRU Address Info	VSO, then none, FRU ID, or FRU ID, key type, site, site type	03h, HW address, IPMB address, FFh, 00h, site, 00h	CCh key type other than 03h; D3h another site

Common completion codes: C1h unknown command (and every Tier-1 not-applicable command), C2h a destination LUN other than 0, C7h a wrong request length, C9h a FRU ID other than 0 (except the physical-address form of Get FRU Address Info, which ignores the ID). Reservation IDs, record IDs and FRU offsets are two bytes, LS byte first; a Get Device SDR count

of FFh reads the rest of the record; a Set IPMB State bus byte is 00h (isolate), 01h (Local Control) or FFh (no change). The optional speed byte holds the IPMB-A speed in bits 1:0 and the IPMB-B speed in bits 3:2: 00b is 100 kHz and 01b is 400 kHz at Tier-3 on a fitted bus; every other value, a non-zero bits 7:4 and 400 kHz below Tier-3 answer CCh with no change (section 10.7).

Commands VITA 46.11 requires above Tier 1

The public build implements the Tier-1 commands above and answers every other command C1h (Rules 4.2-2 and 4.1.1-16). The table lists the commands whose Tier-2 or Tier-3 column of VITA 46.11 Tables 10.1.1-1 and 10.1.2-1 is mandatory (M, or O/M with a condition) and whose Tier-1 column is not. It is taken from `Documentation/ipmi_commands.md`, which holds the full tables, their page numbers and the footnotes behind each `[n]` marker; a Tier-1 build answers each of these commands C1h.

NetFn/Cmd	Command	Tier 1	Tier 2	Tier 3
06h/02h	Cold Reset	O[2]	M[2]	M
06h/03h	Warm Reset	O	O	M
06h/2Eh	Set BMC Global Enables[6]	O	O	O/M[7]
06h/2Fh	Get BMC Global Enables	O	O	M
06h/34h	Send Message	O	O	O/M[7]
06h/52h	Master Write-Read	O[2]	O[2]	O/M[8]
04h/00h	Set Event Receiver	NA	M	M
04h/01h	Get Event Receiver	NA	M	M
04h/02h	Platform Event (a.k.a "Event Message")	NA	M	M
04h/28h	Set Sensor Event Enable	NA	O	M
04h/29h	Get Sensor Event Enable	NA	O[2]	M
0Ah/40h	Get SEL Info	O	O	M
0Ah/42h	Reserve SEL	O[2]	O[2]	M
0Ah/43h	Get SEL Entry	O	O	M
0Ah/47h	Clear SEL	O	O	M
0Ah/48h	Get SEL Time	O	O	M
0Ah/49h	Set SEL Time	O	O	M
2Ch/04h	FRU Control	O	O	O/M[6]
2Ch/0Ah	Set FRU State Policy Bits	O/M[2]	O/M[2]	M
2Ch/0Bh	Get FRU State Policy Bits	O/M[2]	O/M[2]	M

NetFn/Cmd	Command	Tier 1	Tier 2	Tier 3
2Ch/0Ch	Set FRU Activation	O	O	M
2Ch/1Eh	FRU Control Capabilities	O/M[3]	M	M
2Ch/44h	Get Mandatory Sensor Numbers	O	O/M[4]	M
2Ch/45h	Get FRU Hash	O	O	M
2Ch/46h	Get Payload Mode Capabilities	O	O	M
2Ch/47h	Set Payload Mode	O	O	M
2Ch/4Bh	Get Power Supply Capabilities	O	O	O/M[5]
2Ch/4Ch	Get Power Supply Status	O	O	O/M[5]
2Ch/4Dh	Set Power Supply Status	O	O	O/M[5]

The commercial release build adds the Tier-2/3 extension (private source, `../IPMIRustFRU_Tier23`). Its commands, the events it sends and its switches follow; its test cases and results are in `Documentation/ipmc_test_coverage.md` (section "Tier-2/3 extension"). The commands use the public mechanisms of this repository: the extension hook (section 4, Build modes and the Tier-2/3 extension), the requester (section 6.3), the reset hand-off (section 8.2) and the Tier rules (section 9.6).

Tier-2 commands

NetFn / Cmd	Command	Tier	What it does	Switches
06h / 02h	Cold Reset	2	Request without data (C7h otherwise). Sets `Context::pending_reset` to Cold; the owner of the processor restarts it after the response has left.	`IPMC_TIER`
04h / 00h	Set Event Receiver	2	Two bytes: receiver address (8-bit form, FFh disables) and LUN in bits 1:0. C7h for another length, CCh for an odd address other than FFh or a reserved bit. Starts the generator over with the new receiver (`EventGen::set_receiver`); at Tier-3 with an address other than FFh, `Tier3::resend_active` sends the active Tier-3 events again.	`IPMC_TIER`, `IPMC_EVENT_QUEUE_DEPTH`, `IPMC_FRU_EVENT_RESEND_MS`
04h / 01h	Get Event Receiver	2	Request without data. Response: receiver address (FFh = disabled) and LUN.	`IPMC_TIER`

NetFn / Cmd	Command	Tier	What it does	Switches
2Ch / 1Eh	FRU Control Capabilities	2	Request: FRU Device ID (C7h unless one byte, C9h above the Managed FRUs). Response: `fru_control_capabilities_mask(IPMC_TIER)`: 00h at Tier-2, `tier3::FRU_CONTROL_CAPABILITIES` (11h: Set FRU Activation, cold reset) at Tier-3.	`IPMC_TIER`

Tier-3 commands

A Tier-3 build answers the Tier-2 rows as well.

Reset, enables and the FRU commands

NetFn / Cmd	Command	Tier	What it does	Switches
06h / 03h	Warm Reset	3	Request without data. Sets `Context::pending_reset` to Warm unless a Cold Reset is already pending. The payload state, the FRU state and the Tier-3 state are kept; `Extension::warm_reset` drops only the event in flight.	`IPMC_TIER`
06h / 2Fh	Get BMC Global Enables	3	Request without data. Response: `BMC_GLOBAL_ENABLES` (08h, System Event Logging on).	`IPMC_TIER`
2Ch / 04h	FRU Control	3	Request: FRU Device ID, option. Option 00h (cold reset) in M4 or M5 sets `Context::pending_payload_reset`, a pulse on the payload reset output; other states D5h; other options CCh; C7h unless two bytes.	`IPMC_TIER`
2Ch / 44h	Get Mandatory Sensor Numbers	3	Request: FRU Device ID. D3h in M0. Response: the eight bytes of `MANDATORY_SENSORS`: 00h, 02h, 03h, 04h, 05h, 06h, FFh, 07h.	`IPMC_TIER`
2Ch / 45h	Get FRU Hash	3	Request: FRU Device ID. Response: the CRC-32 (`ipmc_wire::crc32`) of the whole FRU 0 image, least significant byte first, computed for each request.	`IPMC_TIER`
2Ch / 46h	Get Payload Mode Capabilities	3	Request: FRU Device ID. Response: `IPMC_PAYLOAD_MODES` as two bytes, least significant first.	`IPMC_PAYLOAD_MODES`

NetFn / Cmd	Command	Tier	What it does	Switches
2Ch / 47h	Set Payload Mode	3	Request: FRU Device ID, target mode (bits 7:4 reserved), 0 to 8 OEM bytes that are ignored. CCh for a reserved bit or a mode not in `IPMC_PAYLOAD_MODES`; D5h while a transition is in progress. Otherwise the current mode becomes the previous one, the transition completes at once and the Payload Mode event is queued.	`IPMC_PAYLOAD_MODES`, `IPMC_EVENT_QUEUE_DEPTH`

Activation and policy bits

NetFn / Cmd	Command	Tier	What it does	Switches
2Ch / 0Ah	Set FRU State Policy Bits	3	Request: FRU Device ID, mask, values (bits 3:0 are DAL, CDI, DL, AL; bits 7:4 reserved, CCh). DAL and CDI are non-volatile (bits 0 and 1 of `Context::nv_settings`); AL and DL are volatile (`Locks`). Nothing changes on an error: D5h for a DAL or CDI change while NVMRO is asserted, D5h for clearing DL in M4 with CDI set. Follow-on actions: DL cleared in M4 requests a deactivation; DL or CDI set in M5 withdraws it; AL cleared in M1 powers the payload on; AL set in M1 with the payload powering on switches it off. The FRU State event that follows carries cause 3h.	`IPMC_FRU_PERSIST`, `IPMB_NVMRO_SOURCE`, `IPMB_NVMRO_FORCED`
2Ch / 0Bh	Get FRU State Policy Bits	3	Request: FRU Device ID. Response: `Locks::policy_byte` (DAL bit 3, CDI bit 2, DL bit 1, AL bit 0). AL reads as DAL until the first M0 to M1.	`IPMC_FRU_PERSIST`
2Ch / 0Ch	Set FRU Activation	3	Request: FRU Device ID, action (00h deactivate, 01h activate, CCh for others). The result depends on the FRU state; the table below gives it. The FRU State event that follows carries cause 1h.	`IPMC_FRU_PERSIST`

Set FRU Activation by state (the code of `set_fru_activation`):

State	Activate	Deactivate
M0	D5h	D5h
M1	AL cleared, payload power on	00h, nothing
M4	00h, nothing	CDI set: D5h, nothing; else AL set, payload power off

State	Activate	Deactivate
M5	DL set, deactivation request withdrawn	CDI clear: AL set, payload power off; CDI set: DL set, request withdrawn, then D5h
M6	D5h	00h, nothing

The power-on gate `Tier3::power_on_allowed` is the Tier-1 power-on hook: payload power comes on only while AL is clear. `Tier3::poll` drops a deactivation request in any state but M4 and M5.

Sensor event enables

NetFn / Cmd	Command	Tier	What it does	Switches
04h / 28h	Set Sensor Event Enable	3	Request: sensor number, byte 2, optional assertion and deassertion masks (two bytes each, missing bytes read 0); C7h unless 2 to 6 bytes; CBh for a sensor without an SDR record; CCh for action 11b. Byte 2 bit 7 switches the sensor's events on or off, bit 6 is accepted and ignored, bits 5:4 select none, enable or disable for the masks. A bit the sensor's SDR record does not offer is masked off without an error.	`IPMC_TIER`
04h / 29h	Get Sensor Event Enable	3	Request: sensor number (C7h unless one byte, CBh for no record; the analog sensors 08h.. count as records and read events off, masks 0000h; Set accepts them and changes nothing). Response: byte 2 (bit 7 events enabled, bit 6 scanning, always 1), assertion mask, deassertion mask.	`IPMC_TIER`

System Event Log

The log holds `IPMC_SEL_ENTRIES` records of 16 bytes in RAM (64 by default). It does not survive a Cold Reset or a power cycle; a Warm Reset keeps it. A record ID is the position plus one. Every event the generator makes is added once as a type 02h system event record, stamped with the SEL clock and the IPMC's own address; a full log takes no new record and sets the overflow flag. The clock counts seconds of run time plus the offset Set SEL Time chose.

NetFn / Cmd	Command	Tier	What it does	Switches
0Ah / 40h	Get SEL Info	3	Request without data. Response: version 51h, entry count, free bytes (free records times 16), latest add and erase timestamps (FFFFFFFFh before the first), operation support (bit 1 Reserve SEL, bit 7 overflow).	`IPMC_SEL_ENTRIES`
0Ah / 42h	Reserve SEL	3	Request without data. Response: a new non-zero reservation ID; the previous one is cancelled.	`IPMC_SEL_ENTRIES`

NetFn / Cmd	Command	Tier	What it does	Switches
0Ah / 43h	Get SEL Entry	3	Request: reservation ID, record ID (0000h first, FFFFh last), offset, byte count (FFh to the end). C5h for a wrong reservation or a non-zero offset without one, CBh for no such record, C9h for an offset of 16 or more. Response: the next record ID (FFFFh after the last) and the record bytes.	`IPMC_SEL_ENTRIES`
0Ah / 47h	Clear SEL	3	Request: reservation ID, "CLR", action (AAh erase, 00h status). C5h for a wrong reservation, CCh for a wrong signature or action. The erase completes at once; both actions answer progress 1h, and reading the completion ends the reservation.	`IPMC_SEL_ENTRIES`
0Ah / 48h	Get SEL Time	3	Request without data. Response: the SEL clock, four bytes, least significant first.	none
0Ah / 49h	Set SEL Time	3	Request: four bytes, least significant first. The clock reads that value now and counts on from it.	none

A reservation is cancelled by the next Reserve SEL and by a record added to the log.

Events the extension sends

The extension also sends Platform Event messages (04h/02h) to the Event Receiver through the Tier-1 requester. `EventGen` makes them; `Tier3` moves each made event into the SEL. A request completed with C1h stops generation and clears the queue; C0h keeps the event for a resend after `IPMC_FRU_EVENT_RESEND_MS`.

Event	Tier	Trigger
FRU State	2	A FRU state change; cause 6h for a payload power failure, else the cause a command named (1h Set FRU Activation, 3h policy-bit automation), else 0h. A newer event replaces an unacknowledged one, which is resent every `IPMC_FRU_EVENT_RESEND_MS`.
System IPMB Link, offsets 0 to 3	2	A change of the link sensor state.
FRU Health, FRU Voltage	2	A change of the sensor state.
FRU Temperature	2	The upper non-critical, critical and non-recoverable offsets (3 to 5).
System IPMB Link, offsets 4 and 5	3	IPMB-A or IPMB-B enters or leaves 400 kHz (FAST-MODE).
Payload Mode	3	Set Payload Mode accepted; event data 2 holds the transition status and the previous mode.

Set Event Receiver queues the active states again (`RESEND_STATES` : 6 at Tier-2, 9 at Tier-3); `IPMC_EVENT_QUEUE_DEPTH` must be at least that (a compile-time guard). Each event is checked against `Context::event_enables` before it is made (`EventGen::enabled_for`).

Switches

The Tier-2/3 values live in the private configuration file `defs_tier23.rs` (guards in `defs_guards.rs`); the rows marked `defs.rs` are public defines (`crates/ipmc-defs/src/defs.rs`).

Switch	File	Default	Effect on the commands
`IPMC_TIER23` , `IPMC_TIER2` , `IPMC_TIER3`	`defs.rs`	true, true, true	Build the extension and select its Tier; `ipmc_defs::IPMC_TIER` (3, or 2 with `IPMC_TIER3` false) selects the rows answered and the Tier reported in Get VSO Capabilities.
`IPMC_SEL_ENTRIES`	`defs_tier23.rs`	64	SEL size, 1 to 1024 records.
`IPMC_FRU_EVENT_RESEND_MS`	`defs_tier23.rs`	3000	Resend period of an unacknowledged FRU State event, 100 to 60000.
`IPMC_EVENT_QUEUE_DEPTH`	`defs_tier23.rs`	12	Queued sensor events, 1 to 32; at least 6 at Tier-2 and 9 at Tier-3.
`IPMC_EVAL_MINUTES`	`defs_tier23.rs`	30	Evaluation build only (`gen.ps1 build -Eval`): minutes of run time after which the IPMB stops, 1 to 7158.
`IPMC_PAYLOAD_MODES`	`defs.rs`	3	Modes Get Payload Mode Capabilities answers and Set Payload Mode accepts (modes 0 and 1).
`IPMC_FRU_PERSIST`	`defs.rs`	true	A Tier-3 build needs it true: DAL and CDI are stored with the FRU image.
`IPMB_NVMRO_SOURCE` , `IPMB_NVMRO_FORCED`	`defs.rs`	forced, false	NVMRO asserted makes Set FRU State Policy Bits answer D5h for a DAL or CDI change.
`IPMB_BUS_B_ENABLE`	`defs.rs`	true	Enables IPMB-B, which the FAST-MODE events of offset 5 need.

7.3 Console

The selected ST-LINK COM port (COM4 on the current bench), 115200 8N1 (`IPMC_CONSOLE_BAUD`), prompt `CLI>` (`CLI/<group>>` inside a group; `IPMC_CONSOLE_PROMPT`). `HELP` or `?` lists the groups; `<GROUP> HELP` lists a group; a command is typed in full from the top level (`SENSORS SHOW`) or after entering its group, and `UP` or `..` leaves the group. A line holds at most 128 bytes and 40 tokens. A syntax token in `[ ]` is required, in `{ }` optional, and a last `{NAME...}` takes every remaining word; `crates/ipmc-menu/MenuDocumentation.md` has the command reference with the help text of every command.

The command tree is the set of tables in `ipmc-runtime/src/console.rs` : the two top-level aliases and eight groups. The table below is generated from those tables (the FRU rows `SET` and `RESTORE` exist with `IPMC_FRU_CONSOLE_WRITE` true).

Group	Syntax	Help text
(top level)	`STATUS`	Alias of SYS STATUS
(top level)	`RESET`	Alias of SYS RESET
`SYS`	`STATUS`	Uptime, clocks, geographic address, LED, last reset cause
	`RESET`	System reset
	`VERSION`	Firmware and core library versions
`LED`	`ON`	LD2 on
	`OFF`	LD2 off
	`BLINK [HZ]`	LD2 blink at the given rate in Hz; the limits print with LED STATUS
	`AUTO`	LD2 back to the heartbeat / IPMB activity pattern
	`STATUS`	LD2 mode, rate and the accepted blink rate limits
`IPMB`	`ADDR`	Geographic address, IPMB address, parity and NVMRO with their sources
	`STATS`	Per-bus receive, error, transmit, resend and replay counters
	`STATE`	Per-bus fitted, mode and enable state, the transmit policy and the link sensor bytes
`SENSORS`	`SHOW`	Rails, die temperature, inputs, power state, FRU state, Tier-1 readings and their data bytes
	`SDR`	The Device SDR records: sensor number, type, event/reading type, ID string
	`TRACE {LEVEL}`	Sensor trace level OFF, EVENTS (changes) or ALL (changes and samples)
`POWER`	`ON`	Assert PWR_EN and wait for PGOOD
	`OFF`	Release PWR_EN and clear a latched fault
	`CLEAR`	Clear a latched power fault
	`STATUS`	Power state, PWR_EN, latched fault
`SIM`	`RAIL {N} {MV}`	Simulated rail N (0 = VDDA) in millivolts; no argument lists them
	`TEMP {DC}`	Simulated die temperature in tenths of a degree C
	`FAULT {NAME} {STATE}`	Assert (ON) or clear (OFF) a simulated input; no argument lists them
	`SHOW`	Source of every group and the simulated values
`FRU`	`SHOW`	FRU image decoded: common header, board and product areas, checksums

Group	Syntax	Help text
	`DUMP {OFFSET} {COUNT}`	Hex dump of the FRU image; default offset 0 and the rest of the area
	`STORE`	Flash log state: records, newest sequence, free bytes, pending write, next erase
	`FIELDS`	Every FRU field with its maximum length and its current value
	`GET [FIELD]`	Current value of one FRU field; FRU FIELDS lists the names
	`SET [FIELD] {VALUE...}`	Write a FRU field (no value clears a text field or the date; MFG_DATE takes YYYY-MM-DD HH:MM); saved to flash after the save delay; refused while NVMRO is asserted
	`RESTORE`	Save the built FRU image as a new log record and put it back; refused while NVMRO is asserted
`DEBUG`	`STATUS`	Debug switches, capture log use, event receiver, duplicate cache
	`LOG {STATE}`	Capture log ON or OFF; no argument shows its state
	`DUMP {FORMAT}`	Print the capture log: CSV (default) or TRACE; any key stops it
	`CLEAR`	Empty the capture log
	`TRACE {WHAT} {STATE}`	Console trace: IPMI (request, link and event lines) or I2C (one line per frame), ON or OFF
	`EVENTS {STATE}`	Event messages ON (receiver 20h) or OFF (receiver FFh)
	`IPMI [NETFN] [CMD] {DATA...}`	Run one IPMI request locally, as from 20h; nothing goes on the bus
	`INJECT [BUS] {BYTES...}`	Hand a whole IPMB frame to the link as if received on bus A or B
	`CACHE {ACTION}`	Duplicate cache: no argument shows the entries, CLEAR empties it
	`BOOTLOADER {CONFIRM}`	Restart into the bootloader; CONFIRM needed while payload power is on

`FRU FIELDS` lists the thirteen fields with the longest value of each and the value now stored, one line each (`BOARD_SERIAL  max 24  IK-T1-000001`); `FRU GET [FIELD]` prints one. `FRU SET [FIELD] {VALUE...}` writes a field through `IpmbLink::set_unit_field` , the path the application port uses (the live NVMRO check, the store's dirty mark, the delayed flash save; there is no second writer). The value is the words after the field name joined by single spaces; no value clears a text field and clears the date (`OK: MFG_DATE unspecified; ...`); `MFG_DATE` takes `YYYY-MM-DD HH:MM` . The answer is `OK: <FIELD> <value as stored>; saved to flash after the save delay (FRU STORE)` , or `Error: <FIELD> <FieldError name>` with the maximum for `TooLong` and the date range for `BadDate` . With `IPMC_FRU_CONSOLE_WRITE` false the `SET` and `RESTORE` rows are not in the table (the table is chosen at compile time). `FRU SHOW` prints the common header, both areas, a board line that says which stored form is in use (`standard field 000000h (unspecified); the date is in the VITA 34h record` , or, when a Chassis Manager wrote the standard field, `standard field 302010h = <date> UTC, in use; the VITA 34h record is not used while the standard field is nonzero` ; `FRU FIELDS` , `FRU GET MFG_DATE` and the application

port read the same date, VITA 46.11 section 10.3.10) and a MultiRecord section with the record decoded (`Alternate Mfg Date/Time (VITA 34h): 2026-10-23 00:00 UTC`) and its checksum states; a missing record or a bad checksum shows as unspecified, not as a failure. `FRU RESTORE` answers `busy` and erases nothing while the link holds a request or a response (section 10.8).

Console output waits while the console transmit ring is full, and the IPMB service waits with it, so every console command prints at most what the ring holds, `IPMC_UART_TX_BUF` - 1 bytes (4095 by default), whatever the FRU image, a define or the bus put in front of it. `ipmc-runtime` `data_cmds.rs` works the worst row of each listing out in a comment beside its constant and a `const` assertion names the define when the ring is too small:

Command	Bound	Default
`FRU SHOW`	the fixed lines (`SHOW_FIXED` 640), at most `SHOW_FIELDS_MAX` = 12 field rows per area (85 bytes each) and at most `SHOW_RECORDS_MAX` MultiRecord rows (205 bytes each) behind 256 bytes of slack; what is left is one line each (`... <n> more fields (FRU DUMP shows the bytes)` , `... <n> more MultiRecord records (FRU DUMP shows the bytes)`)	(4095 - 640 - 2 x 12 x 85 - 256) / 205 = 5 records
`FRU DUMP`	`DUMP_BYTES_MAX` bytes in whole lines (55 bytes per 16-byte line); a longer range ends in `... <n> more bytes (FRU DUMP <offset> <count> shows them)`	(4095 - 256) / 55 = 69 lines = 1104 bytes
`FRU FIELDS`	13 rows of at most 90 bytes (`FIELDS_ROW`), checked at compile time	1170 bytes
`SENSORS SDR`	the header line and at most `RECORD_COUNT_MAX` (9) records of at most 128 bytes (`SDR_ROW`), checked at compile time	1248 bytes
`DEBUG CACHE`	at most `CACHE_ROWS_MAX` rows of 128 bytes (`CACHE_ROW`), then `... <n> more entries` ; the cache holds up to 64 entries that the bus filled	(4095 - 256) / 128 = 29 rows
`HELP` of the root and of every group	`help_worst` in `ipmc-runtime` `console.rs` adds the rows of the longest listing, checked at compile time (`Console::HELP_FITS`)	the longest listing
`DEBUG DUMP`	not bound: a paced job (`DumpJob`) that prints rows while the ring has room for the longest one	

The other commands print a fixed set of lines of fewer than 128 bytes each. `FRU SHOW` shows the Alternate Mfg Date/Time record (VITA 34h) as the date, as `unspecified` for minute 0, and, for a count above 9999-12-31 23:59 (a record a Chassis Manager wrote), as `out of range (<hex>h minutes), unspecified` : a reader takes such a record as no date, and the console does not invent one (`ipmc_fru::date::checked_text_of` returns `None` for it).

The DEBUG group switches and reads the capture log, runs a request locally and injects a frame (section 10.11); `DEBUG BOOTLOADER {CONFIRM}` restarts into the bootloader (section 16.4); `DEBUG WATCHDOG` prints the `Watchdog:` line, and `HANG CONFIRM` or `PANIC CONFIRM` runs the reset test (a watchdog that is not running refuses both; without `CONFIRM` the command prints what it does). `DEBUG DUMP` is a job: it prints a header line, then rows on the following main-loop passes while the console transmit ring has room for the longest row, then `# end of capture` ; any received byte except a line feed stops it with `# dump stopped` , and a new command line waits until it ends.

The banner after reset prints the firmware and core versions, the `Image:` line (the version, the length and the first 16 digits of the SHA-256 that the image header in flash holds, section 9.1; `SYS VERSION` prints the same two lines), build profile, reset

cause, clocks, geographic address with its source, IPMB address, the application port line (`App port: USART1 115200 8N1, RTS/CTS off`, or `App port:  off`), the `BOOTLOADER pin:` line and the self-test result (`Self test:  55h 00h (passed)`). With `IPMC_EVENTS_AT_BOOT` false in a build that sends events, the line `Events: off from start-up` follows. `SYS STATUS` adds the failure counters of both UARTs (receive overflow, transmit timeout, overrun, trace lines dropped, free transmit bytes).

7.4 Emulator TCP line protocol (`ipmc-emu serve`)

ASCII, one message per line, at most `IPMC_EMU_VECTOR_LINE_MAX` characters, on 127.0.0.1 port `IPMC_EMU_PORT` (5470).

Direction	Line	Meaning
client to emulator	`A <hex bytes>` / `B <hex bytes>`	a frame written on that bus; first byte the 8-bit destination (00h for a broadcast)
client to emulator	`#bus A down` / `#bus A up` (and B)	break or repair a bus: frames are lost and the IPMC's transmissions on it fail
client to emulator	`#reset`	power-cycle the board (fresh board, both buses up, clock restarts; the flash log keeps its content, so a saved FRU write and the policy bits survive)
client to emulator	`#quit`	end the session
emulator to client	`A <hex>` / `B <hex>`	a frame the IPMC transmits
emulator to client	`#ready <hex>`	at connect and after `#reset`: the IPMC's own address
emulator to client	`# <text>`	a trace line (with `--trace`)
emulator to client	`#error <line>: <reason>`	a rejected client line; the session continues

7.5 Pins

Signal	Pin	Header	Owner	Direction
IPMB-A SCL / SDA	PB8 / PB9 (I2C1, AF4)	D15 / D14	`ipmb_pins.rs`, `i2c_hw.rs`	open-drain; internal pull-up with `IPMB_PULL_UPS`
IPMB-B SCL / SDA	PA8 (I2C3, AF4) / PB4 (I2C3, AF9)	D7 / D5	`ipmb_pins.rs`, `i2c_hw.rs`	open-drain; internal pull-up with `IPMB_PULL_UPS`; parked when not fitted
GA0..GA4, GAP, NVMRO	PC0..PC4, PC5, PC6	morpho	`ga.rs`	input, pull-up

Signal	Pin	Header	Owner	Direction
Console TX / RX	PA2 / PA3 (USART2, AF7)	ST-LINK VCP	`uart.rs`	
APP_UART TX / RX	PB6 / PB7 (USART1, AF7)	CN5-3 (D10) / CN7-21	`uart.rs` (claim "app_uart")	TX output; RX input, pull-up
APP_UART CTS / RTS	PA11 / PA12 (USART1, AF7)	CN10-14 / CN10-12	`uart.rs`	claimed only with `IPMC_APP_UART_FLOW` true; CTS input with pull-up (an open CTS reads "not clear to send"), RTS output
BOOTLOADER	PC10	CN7-1	`ga.rs` (claim "boot_pin")	input, pull-down; logic 1 at reset keeps the bootloader (Phase 12); read for the banner and `SYS STATUS`
LD2	PA5	D13	`led.rs`	output, high = on
Rails 1..4	PA0, PA1, PA4, PB0 (ADC1 ch 0, 1, 4, 8)	A0..A3	`adc.rs` (claim "adc")	analog
PWR_EN	PB5	D4	`gpio.rs` (claim "board_io")	output, active high
PAYLOAD_RST_OUT_N	PA9	D8	`gpio.rs` (claim "board_io")	output, open drain, active low, released (high impedance) at start-up; held low `IPMC_PAYLOAD_RESET_PULSE_MS` by a payload reset pulse requested by an extension handler
PGOOD	PB10	D6	`gpio.rs`	input, active high, pull-up
FAULT_N	PA10	D2	`gpio.rs`	input, active low, pull-up
OVERTEMP_N	PC7	D9	`gpio.rs`	input, active low, pull-up
PAYLOAD_RESET_N	PA6	D12	`gpio.rs`	input, active low, pull-up

`local_only/Documentation/nucleo_f401re_pinout.md` is the full pin table for people; a pin change edits the owning module and that table together.

7.6 Application port protocol (`ipmc-app`)

The application port connects the payload CPU on the same board to the IPMC over APP_UART (USART1, PB6 TX, PB7 RX, `IPMC_APP_UART_BAUD` 115200, 8N1; RTS/CTS on PA12/PA11 with `IPMC_APP_UART_FLOW` true). There is one mode: the output

is text a person reads at a terminal and a payload CPU splits with three rules. `PROTOCOL` is 2 (`PROTOCOL_VERSION`); a change that breaks an existing request or answer raises it, a new command does not.

Terminal behaviour

- After reset the IPMC prints the banner, one line, and the prompt `APP>`: `IPMC application port, <firmware name> v<version>, Tier <n>, protocol <n>. Type HELP or ?.` The banner is not an answer, so it has no final line. The prompt comes again after every answer.
- The IPMC echoes every printable byte it receives. Backspace (08h or 7Fh) erases the last character and is answered BS SP BS; with nothing typed it does nothing.
- A line ends at CR, LF or CR LF (one line end: no empty second line; two alike, CR CR, are two). It is at most `IPMC_APP_LINE_MAX` (128) bytes. An empty line prints only the prompt again. A line that is too long ends in `ERROR LINE_TOO_LONG: ...`; a byte other than printable ASCII (20h..7Eh) or a tab, in `ERROR BAD_CHAR: ...`; the bytes past the limit are not echoed.
- Commands and keywords are not case sensitive.

The answer

Every request line gets zero or more data lines and exactly one final line:

- A data line is `KEY: value`: KEY in capitals and underscores, one space after the colon, the value as the rest of the line (empty after the colon and space when there is none, as in `DATA:`). The rows of `HELP` are the one exception: the aligned table, each row starting with two spaces.
- The final line is `OK`, or `ERROR <CODE>: <sentence>`, CODE in capitals and underscores and the sentence written for a person (`ERROR TOO_LONG: BOARD_SERIAL is at most 24 characters`).
- A request may start with a tag, `#` and 1 to 8 letters or digits. The final line carries it after the `OK` or `ERROR`: `OK #a7`, `ERROR #a7 TOO_LONG: ...`. Data lines carry no tag. A bad tag and a line that could not run (too long, bad byte, lost bytes) are answered without a tag.
- A handler that fails after it wrote data lines has them taken back: an `ERROR` answer never mixes with data lines.
- An error sentence that quotes the request (`UNKNOWN`, an unknown field, a bad hex byte, a `HELP` filter that matches nothing) quotes at most `QUOTE_MAX` = 32 bytes of it and ends the quote in `...` when it cuts, so the final line stays within `FINAL_LINE_MAX` (175 bytes) whatever the length of the request.

Parsing rule for a payload CPU

A payload sends the next request after the final line of the previous answer and matches an answer to its request by the tag. The port takes no new byte while an answer drains, so bytes sent early wait in the 256-byte receive ring (`IPMC_APP_UART_RX_BUF`); the limit is that ring filling while a long answer drains, after which bytes are lost (counted, and the next line end is answered `LINE_DAMAGED`).

Send the request with CR LF. Skip every line that starts with `APP>` (the prompt, and the prompt joined with the echo of the request). Collect the lines that look like `KEY: value`. The answer ends at the first line that is `OK`, starts with `OK` or starts with `ERROR`; on `OK` the data lines are complete, on `ERROR` they are to be ignored. `ipmc_app::final_line` is this rule in code and the bench client `tools/app-port/app_cli.py` follows it. The client also keeps in step with the prompt: before its first request, and after a request that timed out or ended in a partial line, it sends one CR and reads until the prompt arrives (bounded by the answer timeout); a line is the echo of its request only when it starts with the prompt followed by exactly the request text, and without tags a final line is trusted only after that echo, so the echo of a request such as `OK` is not taken for its answer.

Commands

(`COMMANDS`; `HELP` prints this table from the code, aligned):

Syntax	Help text	Answer (data lines, then `OK`)
`HELP {COMMAND...}`	List the commands, or describe one	the aligned table of every command, or of the rows that start with the words given
`? {COMMAND...}`	Same as HELP	the same: `? <words>` is `HELP <words>`
`INFO`	Firmware name, version, Tier and protocol version	`FIRMWARE`, `VERSION`, `TIER`, `PROTOCOL`
`STATUS`	IPMB address, GA, FRU state, payload power, health, NVMRO, uptime	`IPMB_ADDRESS: 82h`, `GA`, `FRU_STATE: M4 FRU Active`, `PAYLOAD_POWER: on`, `HEALTH: ok`, `NVMRO: deasserted`, `UPTIME: 0d 00:17:48`
`FRU STATE`	Operational state of FRU 0 (M0, M1, M4, M5 or M6)	`FRU_STATE: M4 FRU Active`
`FRU FIELDS`	Every FRU field: current value and maximum length	one line per field, `<FIELD>: <value> (max <n>)`; an empty text field is `<FIELD>:  (max <n>)` (two blanks between the colon and the parenthesis); `MFG_DATE` is `YYYY-MM-DD HH:MM` (UTC)
`FRU GET [FIELD]`	One FRU field	`<FIELD>: <value>`
`FRU SET [FIELD] {VALUE...}`	Write a FRU field; no value clears it; MFG_DATE as YYYY-MM-DD HH:MM	`<FIELD>: <value as stored>`; `MFG_DATE: unspecified` after a clear
`FRU SAVE`	Whether the FRU image is in flash (a change saves itself after the save delay)	`SAVE: done`, or `SAVE: pending` while a change waits (not an error)
`IPMI [NETFN] [CMD] {DATA...}`	Run one IPMI request locally (hex bytes); nothing goes on the IPMB	`CC: 00h`, `DATA: 01 80 00 01 02 09 00 00 00 4B 49 00 00 00 00` (empty after the colon-space when there is none), `COMMAND: App Get Device ID` (no line when the build has no name for it)
`BOOTLOADER {CONFIRM}`	Restart into the bootloader; CONFIRM needed while payload power is on	`BOOTLOADER: restarting`; the restart follows the answer (section 8.3); the port is silent until the bootloader starts the runtime again. While payload power is on, the line without `CONFIRM` answers `ERROR CONFIRM` and asks for no restart

Syntax strings: leading keywords in capitals, then `[NAME]` for a required parameter, `{NAME}` for an optional one and `{NAME...}` for the rest of the line as one value (last only). The longest keyword sequence the line starts with wins; then the argument count is checked and a mismatch is `ERROR ARGS: usage: <syntax>`. An unknown command is `ERROR UNKNOWN: unknown command '<word>'; type HELP or ?`.

`STATUS` words: `PAYLOAD_POWER` is `on`, `off` or `fault` (`off` also covers a power-on that still waits for PGOOD); `HEALTH` is `ok` or `fault` (`fault` covers a bad reading and a missing one, before the first sample); `NVMRO` is `asserted` or `deasserted`. `IPMI` takes every number as a hex byte (`06`, `0x06` and `6` are the same) and runs the request through the dispatcher as if the Chassis Manager had sent it, the same function as `DEBUG IPMI`; a completion code other than 00h is still an `OK` answer, because the request ran; `COMMAND` is the name the capture log prints.

The FRU fields (`FRU FIELDS`; `max` is the room `IPMC_FRU_*_MAX` reserves; the console uses the same table):

Field	FRU area	Longest value
`BOARD_MFR`	Board Info, manufacturer	24
`BOARD_PRODUCT`	Board Info, product name	32
`BOARD_SERIAL`	Board Info, serial number	24
`BOARD_PART`	Board Info, part number	24
`BOARD_FILE_ID`	Board Info, FRU file ID	16
`MFG_DATE`	the VITA 34h record, manufacturing date and time, written as `YYYY-MM-DD HH:MM` (UTC, 1996-01-01 00:01 to 9999-12-31 23:59); reads the standard board field when it is nonzero (a Chassis Manager wrote it), else the record, else `unspecified` (no record, a bad checksum or minute 0)	16
`PRODUCT_MFR`	Product Info, manufacturer	24
`PRODUCT_NAME`	Product Info, product name	32
`PRODUCT_PART`	Product Info, part or model number	24
`PRODUCT_VERSION`	Product Info, version	16
`PRODUCT_SERIAL`	Product Info, serial number	24
`ASSET_TAG`	Product Info, asset tag	24
`PRODUCT_FILE_ID`	Product Info, FRU file ID	16

`FRU SET` writes the RAM image through `IpmbLink::set_unit_field`, the editor of section 6.2, then reruns the self test; the dirty mark and the quiet time of `IPMC_FRU_PERSIST_DELAY_MS` carry the change to the flash, as for a Write FRU Data from the Chassis Manager, so a payload CPU that wrote fields repeats `FRU SAVE` until it reads `SAVE: done` before it removes power. The text of a field is the rest of the line with the blanks at both ends removed, so blanks inside the value stay. `FRU SET` with no value clears a text field, and `FRU SET MFG_DATE` with no value clears the date (`MFG_DATE: unspecified`).

Error answers

The codes; each has its own sentence:

Code	Cause
`UNKNOWN`	no command of that name (`ERROR UNKNOWN: unknown command 'X'; type HELP or ?`, the name quoted at most 32 bytes); also `HELP X` for a name that is no command
`ARGS`	missing or extra arguments (`usage: <syntax>`), an unknown field, a bad tag, a bad hex byte, more data than one frame holds, a tag with no command
`LINE_TOO_LONG`	the line is longer than `IPMC_APP_LINE_MAX`; it was dropped
`BAD_CHAR`	the line holds a byte that is not printable ASCII; it was dropped
`TOO_LONG`	the value is longer than the field holds (`BOARD_SERIAL is at most 24 characters`)
`BAD_LENGTH`, `NOT_ASCII`	a text of one byte (a type/length field holds 0 or at least 2 bytes); a value that is not printable ASCII
`NO_ROOM`	the rewritten fields do not fit the area (an image a Chassis Manager wrote may have no spare room)
`NVMRO`	the NVMRO pin is asserted; nothing is written
`BAD_DATE`	the date text is malformed or out of range (`use YYYY-MM-DD HH:MM (UTC), 1996-01-01 00:01 to 9999-12-31 23:59`)
`FRU_IMAGE`	the area is missing or damaged, the image's areas overlap, or the rewritten image would fail `verify`
`WRITE_DISABLED`	`IPMC_APP_FRU_WRITE` is false and the request is `FRU SET` or an `IPMI` Write FRU Data; nothing runs
`CONFIRM`	`BOOTLOADER` without `CONFIRM` while payload power is on (PWR_EN asserted; the reset releases it): `payload power is on and drops at the reset; send BOOTLOADER CONFIRM`; no restart is asked
`LINE_DAMAGED`	`<n> bytes lost or damaged on the port; the line was dropped`: the UART lost bytes (overrun or receive ring full) or received a byte with a framing, noise or parity error; the firmware discarded every byte waiting in the receive ring (and the byte it had just taken, when it found the loss with one in hand), and the first line end that arrives after the discard is answered with this error, without a tag, and does not run. Lines after it run normally. A line that was complete inside the discarded bytes gets no answer: the payload CPU's answer timeout covers it, and it asks again
`IPMI_FAILED`	the request did not run: the request or its response does not fit one IPMB frame, or the dispatcher gave no response (a NetFn that is odd or above 3Eh)
`LOG_OFF`	`FRU SAVE` with the flash log off (`IPMC_FRU_PERSIST` false or the store did not start)
`ANSWER_TOO_LONG`	`the answer does not fit <n> bytes`: the answer did not fit the answer buffer (`ANSWER_MAX`, 1280 bytes by default); the engine goes back to the start of the answer and writes this error line, with the tag, in place of the data lines. A buffer too small even for that line holds nothing of the answer, never half a line

Output and time

The firmware formats the echo, the answer and the prompt into one buffer of `ANSWER_MAX` bytes and copies it to the transmit ring as the ring has room (`ipmc-runtime` `app_port.rs` `drain`, called on every pass: the answer drains into the ring across passes, so the ring (`IPMC_APP_UART_TX_BUF` - 1 usable bytes) need not hold it whole and no guard ties the two); the main loop never waits for the UART. `ANSWER_MAX` is computed at compile time (`commands.rs`): the echo of one pass (`IPMC_APP_BYTES_PER_POLL` x 3 = 96 bytes), plus the larger of the longest data answer (`HELP` or `FRU FIELDS` at 63-byte values: 1141 bytes, with `OK #<tag>` and the prompt: 1256 bytes in all) and the longest `ERROR` final line (`FINAL_LINE_MAX` 175 bytes) with the prompt, rounded up to a multiple of 64: 1280 bytes with the default table. While output is pending the port takes no new byte, so one request is answered at a time, in order. A pass takes at most `IPMC_APP_BYTES_PER_POLL` bytes and ends after one answer, so the IPMB service waits at most for the formatting of one answer (estimated from the code at up to about 1 ms for `HELP` or `FRU FIELDS`; `FRU SET` and `IPMI` run their whole work in that pass; an estimate, to be measured on the board); the 1280 bytes need 111 ms on the wire at 115200 baud, spent in the UART interrupt. Every request line and the final line of every answer go into the capture log as `app request <line>` and `app answer <line>` (section 10.11). The port looks at its loss counter before the first byte of a pass and after every byte it takes, before the byte is served. A loss found then discards the byte in hand as well and ends the pass: the byte can be the line end of the damaged line, or of a line before it, and it is not trusted. The editor stays damaged until a line end that arrives after the discard, so the suffix of a damaged line (`FO` CR after a lost `IN`) answers `LINE_DAMAGED` and never runs as a request of its own, and a request that lost a byte never runs. A Cold Reset asked for on this port waits for its own answer: the output left in the buffer or the transmit ring keeps the reset gate from going (section 8.2).

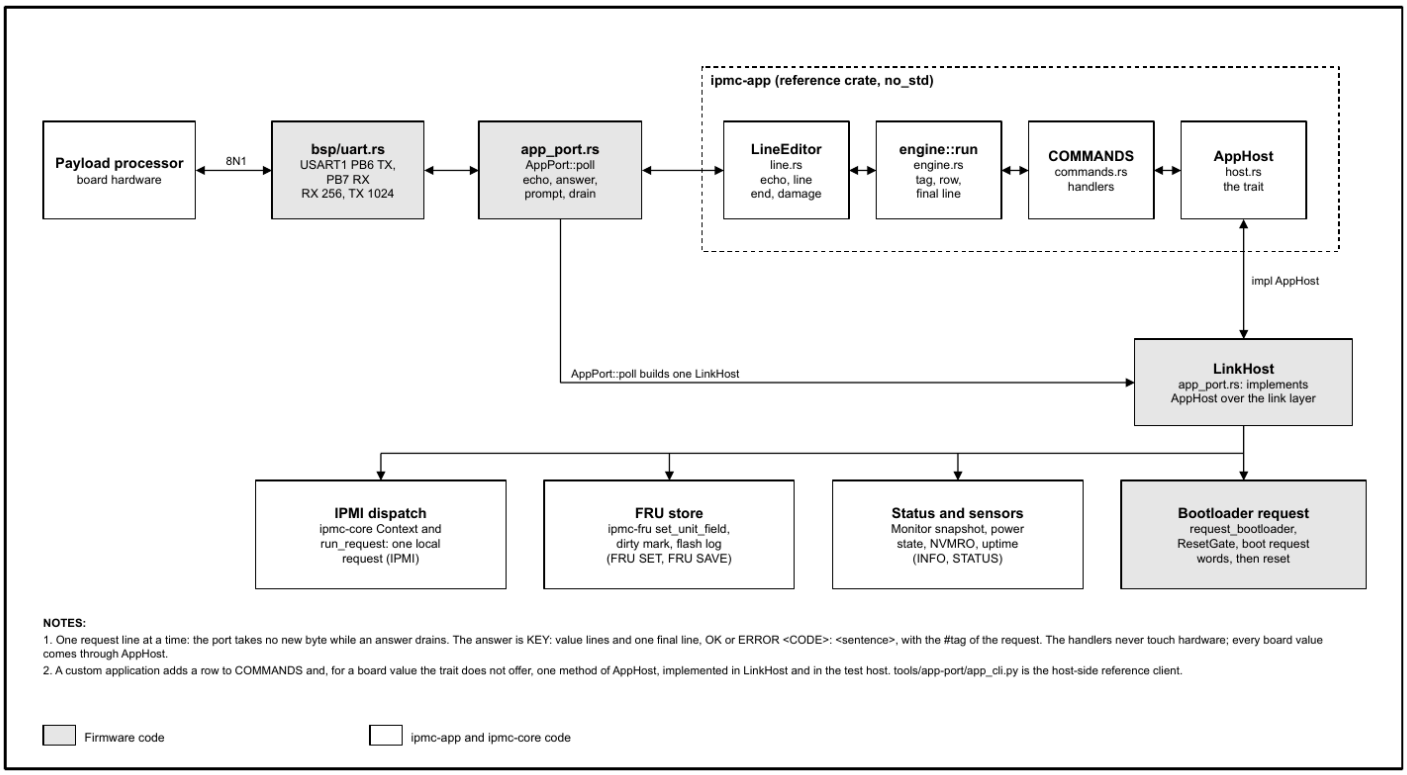
A session

The text is the output of the unit test `a_session_prints_as_the_documents_show_it` for the Tier-1 build and the test host's FRU image; typed text follows `APP>` :

```
IPMC application port, ipmc-nucleo-f401re v0.1.0, Tier 1, protocol 2. Type HELP or ?.
APP> HELP
  HELP {COMMAND...}              - List the commands, or describe one
  ? {COMMAND...}                 - Same as HELP
  INFO                           - Firmware name, version, Tier and protocol version
  STATUS                         - IPMB address, GA, FRU state, payload power, health, NVMRO, uptime
  FRU STATE                      - Operational state of FRU 0 (M0, M1, M4, M5 or M6)
  FRU FIELDS                     - Every FRU field: current value and maximum length
  FRU GET [FIELD]                - One FRU field
  FRU SET [FIELD] {VALUE...}     - Write a FRU field; no value clears it; MFG_DATE as YYYY-MM-DD HH:MM
  FRU SAVE                       - Whether the FRU image is in flash (a change saves itself after the s
  IPMI [NETFN] [CMD] {DATA...}   - Run one IPMI request locally (hex bytes); nothing goes on the IPMB
  BOOTLOADER {CONFIRM}           - Restart into the bootloader; CONFIRM needed while payload power is o
OK
APP> STATUS
IPMB_ADDRESS: 82h
GA: 1
FRU_STATE: M4 FRU Active
PAYLOAD_POWER: on
HEALTH: ok
NVMRO: deasserted
UPTIME: 0d 00:17:48
OK
APP> FRU FIELDS
BOARD_MFR: IronKernel (max 24)
BOARD_PRODUCT: VPX Tier-1 IPMC (max 32)
BOARD_SERIAL: IK-T1-000001 (max 24)
BOARD_PART: IK-IPMC-T1-NUC (max 24)
BOARD_FILE_ID: ipmc-fru-v1 (max 16)
MFG_DATE: 2026-10-23 00:00 (max 16)
PRODUCT_MFR: IronKernel (max 24)
PRODUCT_NAME: Tier-1 IPMC Reference (max 32)
PRODUCT_PART: IK-IPMC-T1 (max 24)
PRODUCT_VERSION: 0.1 (max 16)
PRODUCT_SERIAL: IK-T1-000001 (max 24)
ASSET_TAG:  (max 24)
PRODUCT_FILE_ID:  (max 16)
OK
APP> FRU SET BOARD_SERIAL 123456789012345678901234
ERROR TOO_LONG: BOARD_SERIAL is at most 24 characters
APP> IPMI 06 01
CC: 00h
DATA: 01 80 00 01 02 09 00 00 00 4B 49 00 00 00 00
COMMAND: App Get Device ID
OK
APP> FRU SAVE
SAVE: done
OK
APP>
```

Connecting a payload processor

Figure 7-1. Payload processor integration through the application port (ipmc-software, sheet 7)



Source: `Documentation/diagrams/src/ipmc-software.js`, sheet 7.

`ipmc-app` is a reference crate for a customer application (PLAN decision 27): it may repeat code of the other crates, and a customer application replaces or extends it. The protocol crate never touches hardware; the firmware gives it the board through the `AppHost` trait.

Wiring and settings (3.3 V logic; the pins are claimed by `bsp/uart.rs` under the owner `app_uart`):

Item	Value
IPMC TX, payload RX	PB6 (USART1, AF7), header CN5-3 (D10)
IPMC RX, payload TX	PB7 (USART1, AF7), header CN7-21; input with pull-up
Ground	common ground with the payload processor
Optional RTS/CTS	`IPMC_APP_UART_FLOW` true: IPMC CTS input on PA11 (CN10-14, with pull-up; open reads "not clear to send") from the payload's RTS, IPMC RTS output on PA12 (CN10-12) to the payload's CTS; false leaves PA11 and PA12 unclaimed
Format	`IPMC_APP_UART_BAUD` 115200, 8 data bits, no parity, 1 stop bit
Port enable	`IPMC_APP_UART_ENABLE` true; false leaves USART1 off and parks its pins
Buffers	receive ring `IPMC_APP_UART_RX_BUF` 256 bytes, transmit ring `IPMC_APP_UART_TX_BUF` 1024 bytes, longest request line `IPMC_APP_LINE_MAX` 128 bytes, `IPMC_APP_BYTES_PER_POLL` 32 bytes per main-loop pass

Item	Value
Text	prompt `IPMC_APP_PROMPT` (`APP> `), name `IPMC_APP_UART_NAME` (`APP`)
Write access	`IPMC_APP_FRU_WRITE` true; false answers `ERROR WRITE_DISABLED` to `FRU SET` and to an `IPMI` Write FRU Data

Line protocol in one place (the rules are above): the payload sends one request line ending in CR, LF or CR LF; a request may start with a tag, `#` and 1 to 8 letters or digits. The IPMC answers zero or more data lines `KEY: value` and exactly one final line, `OK` or `ERROR <CODE>: <sentence>`, with the tag after the `OK` or `ERROR` word. The payload skips the lines that start with the prompt, collects the data lines, and ends the answer at the first line that is `OK`, starts with `OK ` or starts with `ERROR `. The codes the payload handles are in the error table above, among them `LINE_DAMAGED` (a lost or damaged byte: send the request again), `CONFIRM` (`BOOTLOADER` while payload power is on) and `NVMRO`.

Worked sessions. Typed text follows `APP> `; the payload sends the tag, the IPMC answers it on the final line.

Reading the state, then writing the unit fields and waiting until they are in flash (the payload repeats `FRU SAVE` until it reads `SAVE: done`):

```
APP> #a1 STATUS
IPMB_ADDRESS: 82h
GA: 1
FRU_STATE: M4 FRU Active
PAYLOAD_POWER: on
HEALTH: ok
NVMRO: deasserted
UPTIME: 0d 00:17:48
OK #a1
APP> #a2 FRU GET BOARD_SERIAL
BOARD_SERIAL: IK-T1-000001
OK #a2
APP> #a3 FRU SET BOARD_SERIAL 00:11:22:33:44:55
BOARD_SERIAL: 00:11:22:33:44:55
OK #a3
APP> #a4 FRU SET MFG_DATE 2026-10-25 08:30
MFG_DATE: 2026-10-25 08:30
OK #a4
APP> #a5 FRU SAVE
SAVE: pending
OK #a5
APP> #a6 FRU SAVE
SAVE: done
OK #a6
```

Refusals carry the tag and no data lines:

```
APP> #a7 FRU SET BOARD_SERIAL 123456789012345678901234
ERROR #a7 TOO_LONG: BOARD_SERIAL is at most 24 characters
APP> #a8 BOGUS
ERROR #a8 UNKNOWN: unknown command 'BOGUS'; type HELP or ?
```

An IPMI request run locally (the dispatcher answers as if the Chassis Manager had sent it; nothing goes on the IPMB):

```
APP> IPMI 06 01
CC: 00h
DATA: 01 80 00 01 02 09 00 00 00 4B 49 00 00 00 00
COMMAND: App Get Device ID
OK
```

Entering the bootloader to update the runtime (section 16.4). With payload power on the plain command only warns; after `CONFIRM` the IPMC restarts and the port stays silent until the bootloader or the runtime prints again:

```
APP> BOOTLOADER
ERROR CONFIRM: payload power is on and drops at the reset; send BOOTLOADER CONFIRM
APP> BOOTLOADER CONFIRM
BOOTLOADER: restarting
OK
```

The seam for a custom application is the `AppHost` trait (`crates/ipmc-app/src/host.rs`). A host implements these methods; `LinkHost` in `ipmc-runtime` `app_port.rs` is the implementation and `FakeHost` in `tests.rs` the test one:

Method	Gives the protocol
`version`	the firmware name and version text of `INFO`
`status`	`Status`: IPMB address, GA, power state, health, NVMRO, uptime, whether PWR_EN is asserted
`fru_state`	the FRU 0 Operational State
`unit_field`, `set_unit_field`	one FRU unit field read, or written through the one Write FRU Data save path
`ipmi`	one local IPMI request: completion code and data (`IpmiData`), or `AppError`
`command_name`	the name of a (NetFn, command) pair for the `COMMAND:` line
`save_state`	`SaveState`: `Saved`, `Pending` or `LogOff`
`request_bootloader`	asks for a restart into the bootloader

A custom command is one handler and one row of `COMMANDS` in `ipmc-app/src/commands.rs`, and, for a board value the trait does not offer, one new `AppHost` method; section 12.4 lists the steps and the tests that guard a new row. `tools/app-port/app_cli.py` is the reference host-side client: it follows the parsing rule above, tags every request, keeps in step with the prompt, and exits 0 when every answer is `OK` (`python tools\app-port\app_cli.py --port COM5 INFO STATUS`; options in `tools/app-port/README.md`).

8. Clocking, Reset and Start-Up

8.1 Clocks

Clock	Source	Frequency	Used by	Notes
HSE	ST-LINK MCO into OSC_IN, bypass	8 MHz	PLL input	HSI 16 MHz (M = 16) when HSE never becomes ready
SYSCLK, HCLK	PLL: 8 MHz / 8 x 336 / 4	84 MHz	core, DWT cycle counter, SysTick	`IPMC_SYSCLK_HZ`, compile-time check
PCLK1	HCLK / 2	42 MHz	I2C1, I2C3 (CR2.FREQ 42), USART2	APB1 maximum
PCLK2	HCLK	84 MHz	ADC1 (ADCCLK = PCLK2 / 4 = 21 MHz), USART1 (the application port)	
PLLQ	VCO / 7	48 MHz	unused	
SysTick	HCLK	1 kHz tick	`millis`, LED pattern	`catch_up` after a flash erase
DWT CYCCNT	HCLK	84 MHz	bit-bang timing, deadlines, erase stall time	started before the FRU log boot

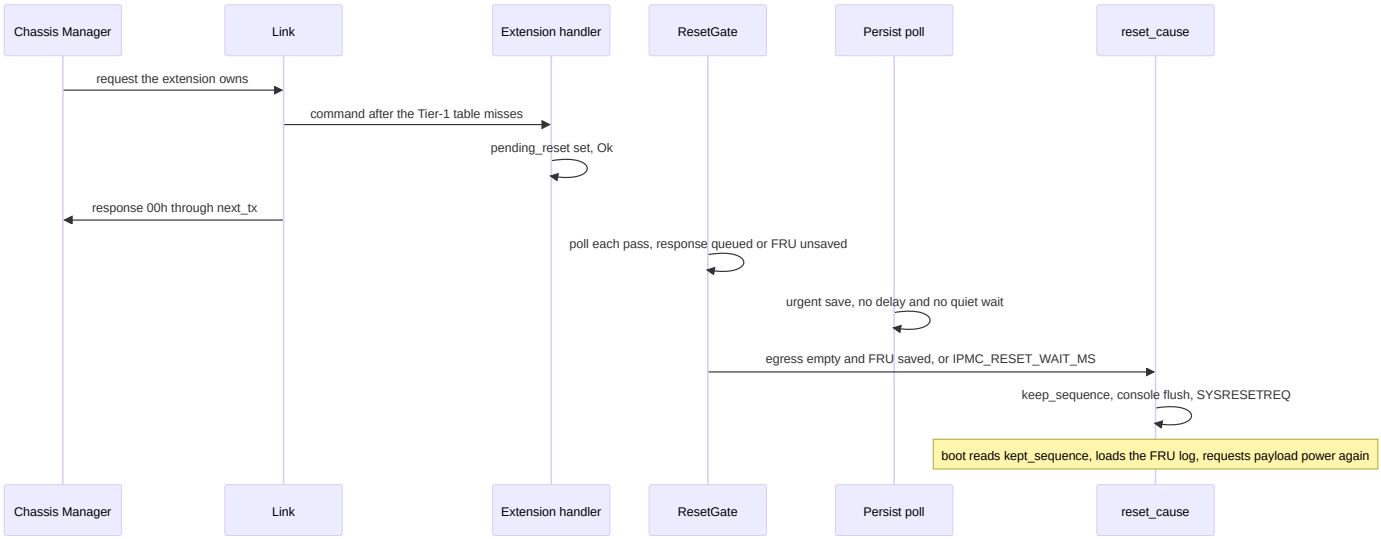
8.2 Reset

The firmware has one reset domain: the MCU reset (power-on, pin, software from `SYS RESET`, from a pending Cold Reset or from a bootloader request, or the independent watchdog). A pending Warm Reset and a pending payload reset are not processor resets; they use the same reset gate. The commands that set these hand-offs belong to the commercial build (section 7.2: Cold Reset, Warm Reset, FRU Control). `bsp/reset_cause.rs` reads RCC_CSR once at start-up, reports the most specific cause in the banner and clears the flags. A HardFault, an unhandled interrupt or a panic (the handler of `bsp/fault.rs`, both images) stores its kind, value and check word in the kept fault record (`bsp/fault_record.rs`, NOINIT, readable with a debugger), prints one line when the UART is up and parks in `wfi`. The independent watchdog (`bsp/watchdog.rs`, PLAN decision 42) is the only reset after that park: the runtime starts it just before the main loop with `IPMC_WATCHDOG_MS` and feeds it once per loop pass, nowhere else, so a parked, hung or panicked runtime resets within the period. The next start prints `Last reset:  independent watchdog` and, from the record, `Last fault:` (`HARDFAULT pc=0x...`, `UNHANDLED IRQ n` or `PANIC line n column m`). The bootloader starts no watchdog, and before the runtime starts it the processor stays parked. The watchdog freezes while a debugger halts the core (`DBG_IWDG_STOP`). The period guard in `watchdog.rs` requires the shortest real period to exceed twice the 500 ms FRU sector erase; `DEBUG WATCHDOG HANG CONFIRM` and `DEBUG WATCHDOG PANIC CONFIRM` test the reset. RAM state starts from its Rust initialisers; the state kept across a reset is the FRU log in flash and, after a software reset, the request sequence words described below.

Cold Reset hand-off

A handler never resets the processor. An extension handler sets `Context::pending_reset` to `Some(ResetRequest::Cold)` and queues its response as any response; a bootloader request (`IpmbLink::request_bootloader` , section 16.4) enters the same path. `IpmbLink::service` returns the reset as a value, `Restart { boot }` , and the owner of the processor, the firmware main loop or the emulator, performs it.

1. Each pass `IpmbLink::service_reset` (inside `service`) calls `ResetGate::poll(pending, now_ms, egress_empty, fru_clean, trace)` . The gate returns true when the egress queue is empty (the response left the board or was dropped), the application port has no output left (its buffer and TX ring are empty, so a reset asked for on that port finishes its own answer first) and the FRU store holds no unsaved change, or when `IPMC_RESET_WAIT_MS` has passed since it first saw the request; in the second case one trace line says what was still open.
2. While a reset is pending the FRU save runs at once (`service_fru_log`): no persist delay and no wait for a quiet bus, because the IPMB is about to go away. The erase waits, up to 250 ms (longer than the T5 limit of 227 ms, `IPMC_URGENT_LINK_WAIT_MS`), for the link to hold no accepted request and no untransmitted response, and runs anyway after that (one console line says so); the erase itself is bounded (at most 500 ms, covered by `IPMC_RESET_WAIT_MS`). A failed attempt returns to the normal delay, so a failing flash does not stall every pass until the wait runs out.
3. When the gate says go, one console line goes out and `service` returns the `Restart` at once (the pass ends there); the board's main loop hands it to `ipmc_runtime::link::restart` , which flushes the console and the application port (each bounded by its flush bound; the application port output normally left already, because the gate waited for it; with CTS held off the gate's `IPMC_RESET_WAIT_MS` runs out and the reset goes ahead). `ipmc_runtime::kept::keep_sequence` stores the next request sequence number and the mask of the sequence numbers sent inside T9 (`Link::recent_request_seqs`) in four `.uninit` words, which start-up code does not touch, the last a check word over the other three. `reset_cause::system_reset` then requests SYSRESETREQ, so the banner after the restart reads "software reset". `ipmc_runtime::link::restart` owns this step (flush both ports, keep the sequence, write the boot request words when a bootloader request rode on the reset, `SystemControl::system_reset`) and `SYS RESET` calls it as well (through `console::restart_board`): a console reset keeps the request sequence like a Cold Reset and writes no boot request.
4. At the next start-up `ipmc_runtime::kept::kept_sequence` returns the stored values after a software reset, and zero after any other reset or when the check word does not match. `IpmbLink::new` (through `LinkArgs::kept`) hands them to `Link::with_first_request_seq` , which books the recent numbers as sent at start-up, so a request after the restart cannot repeat a sequence number the Event Receiver saw inside T9.
5. Everything else restarts as at power-up: link, context, sensors and the extension's state (IPMB-B is isolated again until Set IPMB State; with `IPMB_COMPLIANCE` `NonStrict` , IPMB-B fitted and a usable address the IPMB-B power-up window is armed again), and the FRU image loads from the flash log.



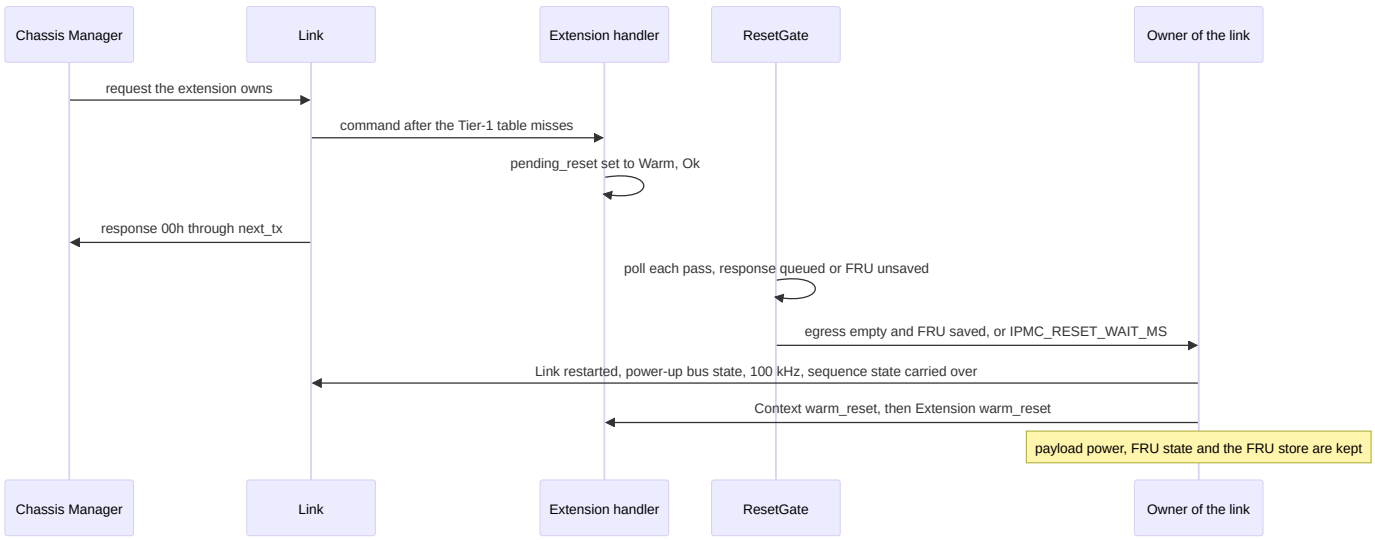
Limitation (owner decision 21, 10-24-2026): on the NUCLEO-F401RE a processor reset is a processor reset, which releases PWR_EN (PB5) with every GPIO, so the payload loses power for the reset and boot time. Recommendation 4.1.9-3, that a reset should not change the payload state, is not met on this board, which has no latch to hold the pin; a target board holds PWR_EN externally. The emulator models the board.

Warm Reset hand-off

A pending Warm Reset restarts the link state without a processor reset, so PWR_EN keeps its level and the payload keeps its power (the problem of decision 21 does not arise). An extension handler sets `Context::pending_reset` to `Some(ResetRequest::Warm)` and queues its response; a pending Cold Reset wins over a Warm Reset request. The same `ResetGate` decides when it runs (the response has left and an accepted FRU write is saved, or `IPMC_RESET_WAIT_MS`), and the FRU save runs at once while it is pending. The owner (`IpmbLink::warm_reset`, in the firmware and in the emulator) calls `Link::warm_reset`, which does, in this order:

1. builds a new Link with `Link::restarted`: the same configuration and address, the power-up bus state (IPMB-A Local Control, IPMB-B isolated until the next Set IPMB State (with `NonStrict`, IPMB-B fitted and a usable address the IPMB-B power-up window is armed again), both buses back at 100 kHz, Rule 4.1.1-4), empty queues and duplicate cache, and the request sequence state carried over so no sequence number is reused inside T9 (the requester is rebuilt with it);
2. runs the owner's `reapply(old, new)`: the firmware keeps a start-up fault of a bus, so a bus that failed at start-up stays disabled;
3. calls `Context::warm_reset`, which installs the new link view and clears every `pending_*` hand-off, the Warm Reset request included;
4. calls `Extension::warm_reset`, which forgets the event in flight, and gives the owner's `ResetGate` a fresh state;
5. the owner prints `Warm Reset: link state re-initialised`; `apply_bus_speeds` returns the drivers to 100 kHz on the next pass.

Kept: payload power and PWR_EN, the FRU state, the FRU store and its settings byte, the SDR repository, the self test result, the sensors and the state of the extension that its `warm_reset` hook keeps. Not kept: the IPMB state (IPMB-B returns to isolated and both buses to 100 kHz, Rules 4.1.9-2 and 4.1.1-4.3, shown as Link events) and the duplicate cache, which starts empty after a Warm or a Cold Reset, so a Chassis Manager retry of a request answered just before the reset runs its handler again.



Payload reset pulse

A payload reset (Rule 4.1.2-9) resets the payload, not the IPMC, and leaves the FRU state unchanged. An extension handler sets `Context::pending_payload_reset`; each main-loop pass `Sensing::serve_hand_offs` takes it and `PayloadResetPulse::pulse_payload_reset` drives PAYLOAD_RST_OUT_N (PA9, open drain, active low) low for `IPMC_PAYLOAD_RESET_PULSE_MS`, and the same call releases it when the time is up. The pulse is not blocking: IPMB traffic continues during it. A build whose power group is simulated has no output and prints that. The emulator runs the same service on the simulated pins and prints the same console line.

8.3 Start-up order (`main.rs` and `ipmc_runtime::run`)

The runtime, linked at `0x0800_C000`, does not start at a reset by itself: the bootloader in sectors 0-1 starts first, checks the runtime image and starts it with the sequence of section 15.6 (the countdown, the session and the entry paths are sections 15 and 16). The steps below then run as from a reset; the runtime clears and reports the reset cause itself.

1. Reset cause (first, before anything can reset again); the kept fault record (`fault_record::take`); after a software reset the kept request sequence numbers are read by `kept_sequence` in step 10.
2. Clock to 84 MHz; a clock fault is reported later in the banner.
3. `ports::init` takes GPIOA..GPIOC; LED claim; SysTick start; USART2 console (a UART failure blinks LD2 at 5 Hz and parks; a pin-claim conflict prints both owners and parks); then, when `IPMC_APP_UART_ENABLE` is 1, APP_UART on USART1 (clocked from PCLK2). A failure of the application port other than a pin-claim conflict does not stop the board: the console prints `APP UART not started: <reason>` after the banner. The evaluation image starts its TIM2 run-time counter after the UARTs (section 8.4).
4. IPMB pins parked as inputs, with the internal pull-up when `IPMB_PULL_UPS` is true.
5. Sensor hardware for the groups whose source is hardware (ADC1, board GPIO).
6. Strap pins (`ga::init`).
7. DWT cycle counter, then `F401Ipmb::new`: the selected master takes the pins and the DWT counter (the I2C slaves do not start yet). `main` fills in `BoardParts` (`StartInfo`: reset cause, kept fault, clock summary, the application port fault, the 64 image header bytes read from flash, the firmware version; the started handles; the FRU flash driver; the master) and `SystemParts` (reset, watchdog, the kept words) and calls `ipmc_runtime::run`, which does the steps 8 to 11 the same way on every board.
8. The FRU log boot (`fru_log::FruLog::boot`): checks the store region against the linker symbols, loads the newest valid record of sector 2, and erases the sector only when it holds damage and no valid record and NVMRO is deasserted. It runs before the I2C slaves exist, because an erase stalls the core (typical 250 ms, at most 500 ms).
9. `addr::resolve` on the raw straps; a master that did not start (step 7) prints `FATAL: IPMB master: ...` and parks; then `ipmc_runtime::link::start_buses`: when the address is usable, each fitted bus gets `IpmbBus::bring_up` (bus recovery when SDA is held low, then the I2C slave). A bus that fails stays disabled with its local status set, and the fault tracker restarts it once the lines read idle-high.
10. The kept request sequence state (`kept_sequence`, step 1) and `home` (`IpmbLink::new`): the static Link (first request sequence number from the kept state) and dispatch Context (self test runs; a stored image replaces the built one and the self test runs on it).
11. Banner (the application port line and the `BOOTLOADER pin:` line included), the `APP UART not started` line when it applies, the events setting (`IPMC_EVENTS_AT_BOOT` false makes the IPMC send itself Set Event Receiver FFh and print `Events: off from start-up`; a build without event messages answers C1h and prints nothing), sensor start (payload power requested when `IPMC_POWER_AT_BOOT` is 1 and the power-on check, `IpmbLink::power_on_check`, allows it; a refusal prints `Power at boot refused: ...`), prompt, watchdog start (`Watchdog::start`, when `IPMC_WATCHDOG_ENABLE`; a failure prints `Watchdog not started: <reason>` and the board runs without it), main loop. Each pass runs `watchdog.feed` first, then `console.poll` (one console line, or a step of a running `DEBUG DUMP`; it returns the `Restart` that `SYS RESET` asked for, which `Console::restart` performs without a prompt),

`console.service_app` (at most one application request line), `console.service_sensors`, `console.service_link` (a `Restart` of a Cold Reset or a restart into the bootloader goes the same way) and `wfi`.

8.4 Evaluation build switch

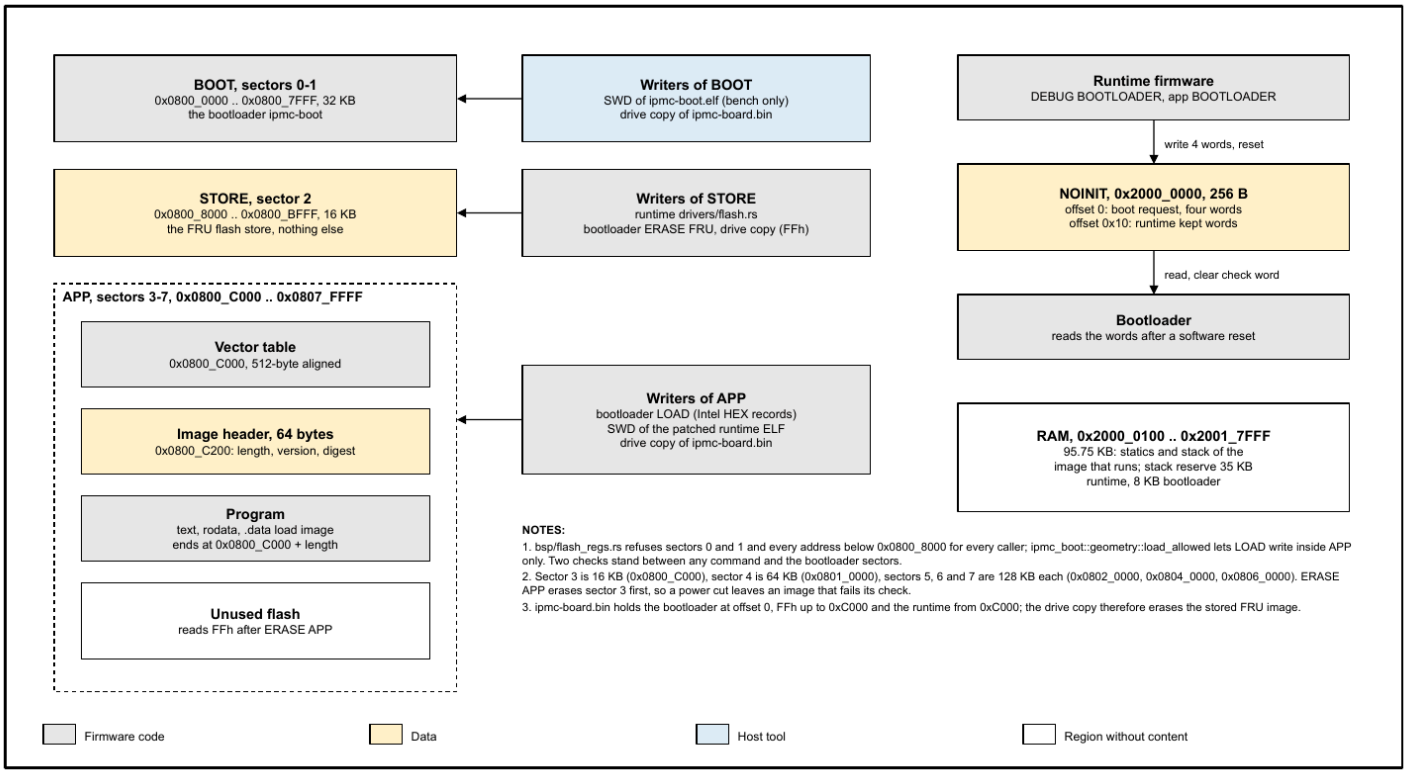
The evaluation image is a time-limited build of the commercial release build (owner decision 20); the switch and its mechanism are public code; section 7.2 lists the commands of the commercial build.

- `gen.ps1 build -Eval` or `gen.ps1 flash -Eval` sets `IPMC_EVAL=1` for the runtime and writes `ipmc-board-eval.bin` (and `ipmc-nucleo-f401re-eval.hex`, `.elf`); without `-Eval` the script passes `IPMC_EVAL=0`, so a build made after an evaluation build carries none of it, and the bootloader is always built with `IPMC_EVAL=0`. `-Eval` needs the extension folder.
- `ipmc-defs` accepts `IPMC_EVAL=1` only with the extension built in and the release profile, and fails the build otherwise. It sets `cfg(ipmc_eval)`, which the firmware crate takes through `dep_cfg.rs`. Every evaluation line sits behind that cfg (`eval.rs`, `drivers/eval_timer.rs`, the timer start and the period check in `main.rs`, `IpmbLink::stop_for_eval`), so no other image, a Tier-1 or normal release or debug build, contains any evaluation code or a flag that could stop the IPMB.
- `drivers/eval_timer.rs` runs TIM2 as a free-running 32-bit up-counter at 10 kHz (0.1 ms per tick, 7158 minutes of range), clocked from PCLK1 (twice PCLK1 when the APB1 prescaler is not 1, RM0368 section 6.2). It starts after the console and before the banner; a start failure stops the image, because it could not end the evaluation. SysTick is not used. The banner of the evaluation image carries the line `EVALUATION BUILD: the IPMB stops after N minutes`.
- The period is `IPMC_EVAL_MINUTES` of run time (a define of the extension's `defs_tier23.rs`, section 4). A power cycle starts a new period.
- At expiry the main loop calls `stop_for_eval` once and prints one console line. It switches both I2C slaves off through the driver's suspend (a healthy bus waits up to the pause time for idle, a faulted one is switched off at once), drops what is queued and never calls `IpmbLink::service` again, so the whole IPMB, responses and requests, stops. The console and the sensor sampling continue, and the FRU save step still runs: a Write FRU Data accepted before the stop is saved, and the save never touches the stopped slaves.

9. Memory Map and Data Layouts

9.1 Flash and RAM (`flash_map.x`, `link.x`, `boot.x`, `drivers/flash.rs`, `flash_regs.rs`)

Figure 9-1. Flash map, writers of each region and the boot request (ipmc-software, sheet 5)



Source: `Documentation/diagrams/src/ipmc-software.js`, sheet 5.

The STM32F401RE has 512 KB of flash in eight sectors (RM0368 Table 5). `flash_map.x` is the one place of every address; the runtime script (`link.x`, through `memory.x`) and the bootloader script (`boot.x`) both include it, and `ipmc_boot::geometry` holds the same numbers as Rust constants (the test `crates/ipmc-boot/tests/flash_map.rs` parses `flash_map.x` and fails on a difference):

Sector	Base	Size	Region	Contents	Written by
0, 1	`0x0800_0000`	2 x 16 KB	`BOOT`	the bootloader `ipmc-boot`: its vector table, code and constants (32 KB)	SWD or the drive copy, bench only

Sector	Base	Size	Region	Contents	Written by
2	`0x0800_8000`	16 KB	`STORE`	the FRU flash store, nothing else	the runtime store driver (sector 2 only); the bootloader erases it on `ERASE FRU` only
3	`0x0800_C000`	16 KB	`APP`	the runtime: vector table at `0x0800_C000`, image header at `0x0800_C200`, then `.text`, `.rodata`, the `.data` load image	the bootloader (`LOAD`), SWD, the drive copy
4	`0x0801_0000`	64 KB	`APP`	the runtime	as above
5, 6, 7	`0x0802_0000`	3 x 128 KB	`APP`	the runtime	as above

Region	Base	Size	Contents
`NOINIT`	`0x2000_0000`	256 B	offset 0: the four words of the boot request (`__boot_request`, `bsp/boot_request.rs`); offset 0x10: the runtime `.uninit` (`KEPT`); offset 0xF0: the four kept fault words (`__fault_record`, `bsp/fault_record.rs`); no start-up code of either image clears it
`RAM`	`0x2000_0100`	95.75 KB	statics and stack of whichever image runs; `_stack_reserve` (35 KB for the runtime, `memory.x`; 8 KB for the bootloader, `boot.x`) must stay free above the statics
System memory	`0x1FFF_7A2A` .. `0x1FFF_7A2F`	6 bytes	factory VREFIN_CAL, TS_CAL1, TS_CAL2 (read by `adc.rs`)

Image header. The runtime links a 64-byte header at `0x0800_C200` (`src/app_header.rs`, the `.app_header` section) with its length and SHA-256 digest; section 16.1 gives the layout and sections 16.2 and 16.3 the digest and the check.

`link.x` is the script of cortex-m-rt 0.7.7 with these differences: the whole image sits in `APP` (`.vector_table` at `ORIGIN(APP)`, the 64-byte `.app_header` at `__app_header`, `.text` after it), `.uninit` sits in `NOINIT` at offset 0x10 and ends at or below `__fault_record` (offset 0xF0), `BOOT` and `STORE` receive no output section, and asserts fail the build when the vector table or the header is not at its address, the header is not 64 bytes, the vector table reaches into the header, the program starts inside the header, a section lies below `APP`, the image does not end inside `APP`, or `.uninit` does not fit `NOINIT`. `boot.x` is the same script for the bootloader: everything in `BOOT`, the RAM sections in `RAM`, nothing in `NOINIT`, and an assert that the image ends inside the 32 KB `BOOT` region (sectors 0-1). The store is the one sector the FRU log erases and programs, so an erase never touches the bootloader or the program. `flash_map.x` exports `__boot_end`, `__app_vectors`, `__app_header`, `__boot_request`, `__fru_store_start` and `__fru_store_end`, and `link.x` exports `__program_start`; `Flash::new` (firmware `drivers/flash.rs`) hands the store region, `__boot_end` and `__app_vectors` to `ipmc_fru::persist::store_region_ok`, which refuses to start the store unless the region is exactly sector 2 (`STORE_START`, `STORE_LEN`), the bootloader ends before it and the runtime starts after it. A layout that drifts apart can therefore never erase code.

`gen.ps1 build` makes the images (section 12.1): `ipmc-boot.bin` (the bootloader, at most 32 768 bytes, a warning above 24 576), the runtime ELF with its header patched, `ipmc-nucleo-f401re.hex` (the file for `LOAD`) and the board image `ipmc-board.bin`: the bootloader at offset 0, FFh up to `0xC000` (the rest of `BOOT` and all of the store sector), the runtime from there. The objcopy calls use `--gap-fill 0xFF`, so the gaps read FFh (erased), not 00h. No runtime-only `.bin` is written: a drive copy programs a `.bin` from `0x0800_0000` and it would land on the bootloader. The ways to program the board treat the store differently:

Way	What it writes	The FRU store
Drive copy of `ipmc-board.bin` to `NOD_F401RE` (`gen.ps1 flash`, `flash -Drive`)	everything: the bootloader, the store as FFh, the runtime	erased (FFh), so the stored FRU image and the unit fields go; the next start-up loads the built image
SWD program of the patched runtime ELF (`gen.ps1 flash -Swd`)	the sections of the ELF only: sectors 3 onward	sector 2 is not touched and keeps its records; the bootloader stays
SWD program of both ELFs (`gen.ps1 flash -Swd -Bootloader`, bench only)	sectors 0-1 and sectors 3 onward	sector 2 is not touched

A reflash may erase the whole flash, unit fields included, and the firmware has no guard against it (owner decision 10-25-2026). A board that holds an image of the earlier layout (the runtime vector table in sector 0 or at `0x0800_8000`) takes one drive copy of `ipmc-board.bin`.

`link.x` asserts that the RAM from the end of the statics (`.data`, `.bss`; `.uninit` is in `NOINIT`) to the top of RAM is at least `_stack_reserve`, 35 KB (`IPMC_STACK_RESERVE_BYTES` of `defs.rs`, which `build.rs` writes to `stack_runtime.x`; the bootloader's `IPMC_BOOT_STACK_RESERVE_BYTES`, 8 KB, goes to `stack_boot.x`; `memory.x` and `boot.x` include their fragment); the error names `IPMC_LOG_SIZE` and the four UART buffer defines. The 35 KB come from the release ELF of 10-26-2026 (`arm-none-eabi-objdump -d`: the frame of each function from its push list and `sub sp`, summed along the longest call chain, indirect calls followed by hand): start-up 27,408 bytes (`main` 9,864 + `IpmbLink::take` 16,616 + `Link::with_first_request_seq` 856 + `memcpy` 72), run time 13,584 bytes (`main` 9,864 + the menu 752 + the deepest console handler, `FRU SET` over `IpmbLink::set_unit_field`, 2,968; the application port paths are shorter), and 544 bytes of interrupt nesting (SysTick 120, I2C event 72, USART 40 and three 104-byte exception frames) on top of either path. The largest, 27,952 bytes, plus a margin of about 25 percent for calls through trait objects, rounds up to 35 KB. The frames of `main` and `take` do not shrink with `IPMC_LOG_SIZE` (the log lives in the static), so the check is on the statics. A larger frame (a new local in `main` or `take`, a larger FRU image) is measured again and `_stack_reserve` changes with it. Since the link service moved to `ipmc-runtime`, `take` is the board function `home` of `main.rs` (`IpmbLink::new` over `singleton!`, `#[inline(never)]`); on the ELF of 10-27-2026 its frame is 17,384 bytes and that of `main` 9,896 bytes (the other figures above are the 10-26-2026 measurement). With the console, the commands and the main loop in `ipmc_runtime::run`, the frames of the ELF built with `gen.ps1 build` are: `main` 920 bytes (the hardware start-up and the `BoardParts` literal), `run` 3,640 (the console and the start-up values for as long as the firmware runs), `home` 17,368, and the loop steps `Console::poll` 1,080, `service_app` 648 and `service_link` 6,416 (each `#[inline(never)]`, so their locals live in their own frames while they run and not in the frame of `run`); `FruLog::boot` is `#[inline(never)]` too (1,640). The start-up chain is `main` 920 + `run` 3,640 + `home` 17,368 + the memory copies below it 72 = 22,000 bytes, the run-time chain through the link step about 11,300 and through the deepest console handler about 9,300, so the largest is the start-up with the interrupt nesting, about 22,600 bytes against the reserve of 35 KB.

The statics of the firmware are the `IpmbLink` (the duplicate cache of 64 entries of 76 bytes, the capture log of `IPMC_LOG_SIZE` bytes and the frame queues) and the four UART rings (`IPMC_UART_RX_BUF`, `IPMC_UART_TX_BUF`, `IPMC_APP_UART_RX_BUF`, `IPMC_APP_UART_TX_BUF`), which live in `.bss` and not in the flash image.

9.2 FRU 0 image (512 bytes)

The six text fields that name the Tier (board product name, serial and part number; product name, part number and serial) carry the Tier of the build: `defs.rs` holds a `_T1`, `_T2` and `_T3` literal of each ("VPX Tier-1 IPMC", "VPX Tier-2 IPMC", "VPX Tier-3 IPMC", serials "IK-T1-000001" and so on), and `ipmc-defs` `lib.rs` defines the name every consumer reads with the selector `by_tier`, the variant of `IPMC_TIER`: the public build is Tier 1, the build with the extension Tier 3 (Tier 2 with `IPMC_TIER3` false). The string and room guards check all eighteen literals, so a bad variant fails every build; the reserved room comes from the selected default. `tools/defs.py` returns the Tier-1 variant unless a Tier is passed. The lengths are equal across the Tiers, so the layout below holds for each.

Offset	Size	Content
00h	8	Common Header: `01 00 00 01 12 28 00 C4` (version 1; Internal Use, Chassis Info 00h; Board at 1 x 8, Product at 12h x 8, MultiRecord at 28h x 8, checksum)
08h	136	Board Info Area (length byte 11h): version, length, language, the standard manufacturing date 000000h (3 bytes), manufacturer, product name, serial, part number, FRU file ID, C1h, room for the five text fields (58 bytes of 00h), padding, checksum
90h	176	Product Info Area (length byte 16h): version, length, language, manufacturer, name, part/model, version, serial, asset tag, FRU file ID (empty), C1h, room for the seven text fields (104 bytes of 00h), padding, checksum
140h	14	MultiRecord area: the VITA 46.11 Alternate Manufacturing Date/Time record (section 6.2): `C0 82 09 <data sum> <header sum> AC 81 00 34 00 20 42 F7 00` for the default date
14Eh	178	00h padding to 200h (`IMAGE_LEN` is 334)

The area sizes are those of the default strings and `IPMC_FRU_*_MAX` values; an area is always a multiple of 8 bytes, and a longer string or a larger `_MAX` grows it (the build fails when the image no longer fits `IPMC_FRU_SIZE`). The room after C1h is the `_MAX` value of each text field less the length of its default string, summed over the area, so an update of a field up to its `_MAX` rewrites the field in place and the area keeps its length (section 6.2).

9.3 Get Sensor Reading data (bytes after the completion code)

Sensor	Type	Byte 2	Byte 3	Byte 4
0 FRU State	F0h	00h	40h	Current State Mask, bit n for Mn (M0 01h, M1 02h, M4 10h, M6 40h; M5 20h at Tier-3)
1 IPMB Link	F1h	B local control bit 7, B status 6:4, A local control bit 3, A status 2:0	40h	link state bit 3 both, 2 only B, 1 only A, 0 none; at Tier-3 bit 4 IPMB-A and bit 5 IPMB-B run at 400 kHz
2 FRU Health	F2h	00h	40h	01h functioning, 02h not functioning

Sensor	Type	Byte 2	Byte 3	Byte 4
3 FRU Voltage	02h	00h	40h	01h in range, 02h out of range
4 FRU Temperature	F3h	00h	40h	LNC bit 0, LC 1, UNC 3, UC 4, UNR 5
5 Payload Test Results	F4h	00h	60h (unavailable)	00h
6 Payload Test Status	F5h	00h	40h	01h not in progress
08h..0Dh analog sensors	02h, 01h	converted count (section 9.5)	40h; 60h while the source is unavailable	comparison bits of IPMI v2.0 Table 35-15: LNC bit 0, LC 1, UNC 3, UC 4, UNR 5 (the temperature only), bits 7:6 read 1 (so a nominal reading ends in C0h)
7 Payload Mode (Tier-3)	F7h	NVMRO bit 6, transition status bits 5:4 (00b in progress, 01b complete, 10b failed), previous mode bits 3:0; gDiscrete1 bit 7 reads 0	40h	current-mode mask, LS byte (bit n is mode n); a fifth byte holds the MS byte (modes 8..14 in bits 6:0), bit 7 reads 1

Status byte (response byte 3) 40h is "scanning enabled"; bit 5 (20h) marks a reading unavailable; bit 7 is the sensor's event enable (`Context::event_enables`, `sensor::apply_event_enable`): 0 in the Tier-1 build, so the table shows 40h, and 1 from power-up in a Tier-2/3 build (the byte reads C0h) until an enable changes. Sensors 2..4 read `00 60 00` before the first sample and while their source cannot be read. On the bench with only IPMB-A enabled sensor 1 reads `08 40 02`. Sensor 7 exists only at Tier-3 (Get Sensor Reading answers CBh for it below Tier-3 and for any number above 7); it is always available, and in mode 0 with the transition complete it reads `10 C0 01 80` (status C0h with the event enable).

9.4 FRU flash record (`persist.rs`)

Offset	Size	Field
0	4	commit word 55464931h, programmed last
4	4	sequence number, from 1, never 0 or FFFFFFFFh
8	2	image length (= `IPMC_FRU_SIZE`)
10	1	record format version (1)
11	1	Tier-3 non-volatile settings: bit 0 Default-Activation-Locked, bit 1 Commanded-Deactivation-Ignored (Rule 4.1.2-10.2), other bits 0
12	512	FRU image
524	4	CRC-32 over bytes 4..524

A record is 528 bytes (132 words), so the 16 KB store sector holds 31 records (16,368 bytes; 16 bytes at the end stay unused). All fields are little-endian; the length, version and settings fields share one word, every other field is whole words. Erased flash reads FFh. The save that finds no free slot erases the sector and writes the record as the first one (section 10.8).

The settings byte is byte 11, the high byte of what the 16-bit version field of an earlier format held (0001h). A record written before the byte existed therefore reads as settings 00h, and a record with settings 00h is byte-identical to one of that format: a Tier-1 and a Tier-2 build never write a non-zero value. A stored byte is masked with 03h when loaded and a saved byte before it is written. A build from before the byte rejects a record whose settings are not 00h and loads the newest older record or the built image without a report, so a Tier-3 image that saved a policy bit is not downgraded in the field without a FRU check. The byte rides in the record with the image, so one dirty mark and one save rule cover a Write FRU Data and a policy-bit change; `FRU RESTORE` keeps the byte as it is.

A record from an older image size (another `IPMC_FRU_SIZE`, so another length field and record length) is not valid for this build: every slot reads invalid and the built image loads. `Persist::boot` finds no valid record in a sector that is not blank and, with NVMRO deasserted, erases the sector at once, so the old content is dropped at that boot (`BootReport::erased`, the console boot line); a flash over SWD, which keeps the store sector, loses it the same way at the first boot. With NVMRO asserted the boot erases nothing (VITA 46.11 Rec 4.1.10-1): the old records stay in the sector, the built image runs, and the first save after NVMRO clears erases them. A flash by drive copy erases the sector before the boot.

9.5 Device SDR repository

The current default includes six analog sensors: **13 sensors and 14 records at Tier-1/2**, with the locator at **000Dh**; Tier-3 has 14 sensors, 15 records and locator **000Eh**. The first table below shows the mandatory-only layout before analog records are inserted ahead of the locator.

Record ID	Type	Content
0000h..0006h	02h compact sensor	sensors 0..6 in number order: entity A0h (VPX module) instance 60h, scanning on, events off at Tier 1, reading masks per Table 10.2-1, ID strings from `IPMC_SDR_*_NAME`; initialization, capabilities and event masks by Tier (section 9.6)
0007h (Tier-3 only)	02h compact sensor	sensor 7, Payload Mode: sensor type F7h, event/reading type 6Fh, initialization 03h, capabilities 40h, reading and assertion masks `IPMC_PAYLOAD_MODES`, deassertion mask 0000h, ID string `IPMC_SDR_PAYLOAD_MODE_NAME`
0007h (0008h at Tier-3)	12h MC device locator	owner = own IPMB address, global initialisation 01b and device capabilities 0Bh at Tier 1 (section 9.6 for Tier 2 and 3), entity A0h instance 60h, ID string "IPMC"

With `IPMC_SDR_ANALOG_SENSORS` (the firmware and the emulator pass `ipmc-sensors` `BOARD_ANALOG`) the analog sensors follow the compact records:

Record ID	Type	Content
0007h..000Bh (0008h..000Ch at Tier-3)	01h Full sensor (IPMI v2.0 Table 43-1)	sensors 08h..0Ch, one per rail of the board table (VDDA, 12V payload, 5V, 3V3 payload, 3V3_AUX), ID strings `IPMC_RAILn_NAME`, sensor type 02h (voltage), unit Volts, unsigned
000Ch (000Dh at Tier-3)	01h Full sensor	sensor 0Dh, the die temperature, ID string `IPMC_SDR_DIE_TEMP_NAME`, sensor type 01h, unit degrees C, 2's complement

Every analog record has the entity and owner of the compact records, sensor initialization 01h (scanning on, no events), capabilities 4Fh (auto re-arm, no hysteresis, threshold access fixed and unreadable, event message control none), event/reading type 01h (threshold), assertion and deassertion events none, linear conversion with B 0, hysteresis 0, and the reading mask bits 12..14 of the first two mask words for the comparisons Get Sensor Reading returns (LNC, LC for the rails and the temperature; UNC, UC for the rails; UNC, UC and UNR for the temperature). A rail's conversion is `y = M x count x 10^-2` volts with M = ceil(nominal mV x `IPMC_SDR_RAIL_SPAN_PCT` / 100 / 2550) (170 %): 8 for 12 V (80 mV per count), 4 for 5 V, 3 for the 3.3 V rails; the nominal sits near 60 % of the 8-bit range (12 V: 150, 96h). Its thresholds are the `IPMC_RAIL_NONCRIT_PCT` and `IPMC_RAIL_CRIT_PCT` windows rounded to a count (UNC 9Eh, UC A5h, LNC 8Fh, LC 87h for 12 V), the normal maximum and minimum the non-critical window, the upper non-recoverable threshold FFh and the lower one 00h (not defined: the sensor maximum and minimum, so no limit lies inside the range). The temperature has M 1, R exponent 0, nominal `IPMC_TEMP_NOMINAL_C`, thresholds `IPMC_TEMP_UNR_C`, `_UC_C`, `_UNC_C`, `_LC_C`, `_LNC_C` (lower non-recoverable 80h, the sensor minimum, -128 C), normal maximum and minimum the UNC and LNC. The reading comes from the sensor snapshot in millivolts and tenths of a degree, so the bench build shows the simulated rails at nominal and the die temperature as measured. The comparison bits (`comparison_flags`) compare the returned reading byte with the record's own threshold bytes, inclusively (at or above an upper one, at or below a lower one; unsigned for a rail, two's complement for the temperature), and byte 4 bits 7:6 read 1 (reserved, Table 35-15). FRU Voltage and FRU Temperature (sensors 3 and 4) keep their millivolt and tenth-of-a-degree evaluation, in which a rail exactly on a window edge is in range, so they and these bits can differ within half a count of an edge (12V payload at 12600 mV reads 9Eh, the UNC byte, while FRU Voltage reads in range).

Get Device SDR Info reports 7 sensors (or 8 records with operation bit 0), and 8 sensors (9 records) at Tier-3, plus 6 sensors and 6 records with the analog sensors (13 sensors and 14 records at Tier 1; 14 and 15 at Tier-3); Get Device Locator Record

ID returns the locator's record ID, 0007h (0008h at Tier-3) without the analog sensors and 000Dh (000Eh at Tier-3) with them. The locator is always the last record.

9.6 Tier-dependent bytes

The Tier rules of `tier.rs` set these bytes; the build reports the Tier of `IPMC_TIER` (1 in the public build, section 4). A Tier-1 IPMC generates no events (Rule 7-4); a Tier-2 and a Tier-3 IPMC are Event Generators (Rule 7-3); a Tier-3 IPMC is also a SEL Device (Rule 7-8) with per-state event enables. A Tier-3 IPMC also controls payload power and activation (Rule 4.1.2-10.1, FRU State M5), has the Payload Mode sensor (sensor 7, Rule 6.1-17) and supports FAST-MODE, 400 kHz (Rule 4.1.1-4.4).

Item	Tier 1	Tier 2	Tier 3
Get VSO Capabilities, IPMC identifier byte bits 1:0 (`IPMC_TIER` - 1)	00b	01b	10b
Get Device ID byte 7, additional device support	09h	29h	2Dh
compact record, sensor initialization	01h	03h	03h
compact record, sensor capabilities	43h	42h	40h
compact record, assertion and deassertion event masks	0000h	per sensor, below	per sensor, below
locator record, global initialization (bits 1:0)	01b	00b	00b
locator record, device capabilities	0Bh	2Bh	2Fh
Get Sensor Reading status bit 7 at power-up (all sensors)	0	1	1
event generator, SEL device, per-state event enables	no, no, no	yes, no, no	yes, yes, yes
activation control (M5), Payload Mode sensor, FAST-MODE	no, no, no	no, no, no	yes, yes, yes
sensors, SDR records with the default analog sensors (Get Device SDR Info)	13, 14	13, 14	14, 15
locator record ID with the default analog sensors (Get Device Locator Record ID)	000Dh	000Dh	000Eh
Get VSO Capabilities, IPMB capabilities byte bits 5:4 (maximum frequency)	00b, 100 kHz	00b, 100 kHz	01b, 400 kHz
sensor 0 reading mask (FRU State)	0053h	0053h	0073h
sensor 1 reading mask (IPMB Link)	000Fh	000Fh	003Fh
Set IPMB State speed byte accepts 01b (400 kHz)	no, CCh	no, CCh	yes, on a fitted bus

Byte 7 of Get Device ID carries the sensor and FRU inventory bits (09h) at every Tier, the Event Generator bit (bit 5) from Tier 2 and the SEL Device bit (bit 2) at Tier 3. The locator's device capabilities byte is that value plus the Device SDR bit (02h).

Sensor capabilities carry auto re-arm (bit 6) and the event message control in bits 1:0: 11b none (Tier 1), 10b global disable only (Tier 2), 00b per state (Tier 3).

Event masks at Tier 2 and 3, assertion and deassertion (`sdr.rs`, LS byte first in the record). Tier-3 adds M5 to sensor 0, the 400 kHz states (offsets 4 and 5, asserted and deasserted) to sensor 1, and sensor 7 with one event per supported mode (`IPMC_PAYLOAD_MODES`, here modes 0 and 1):

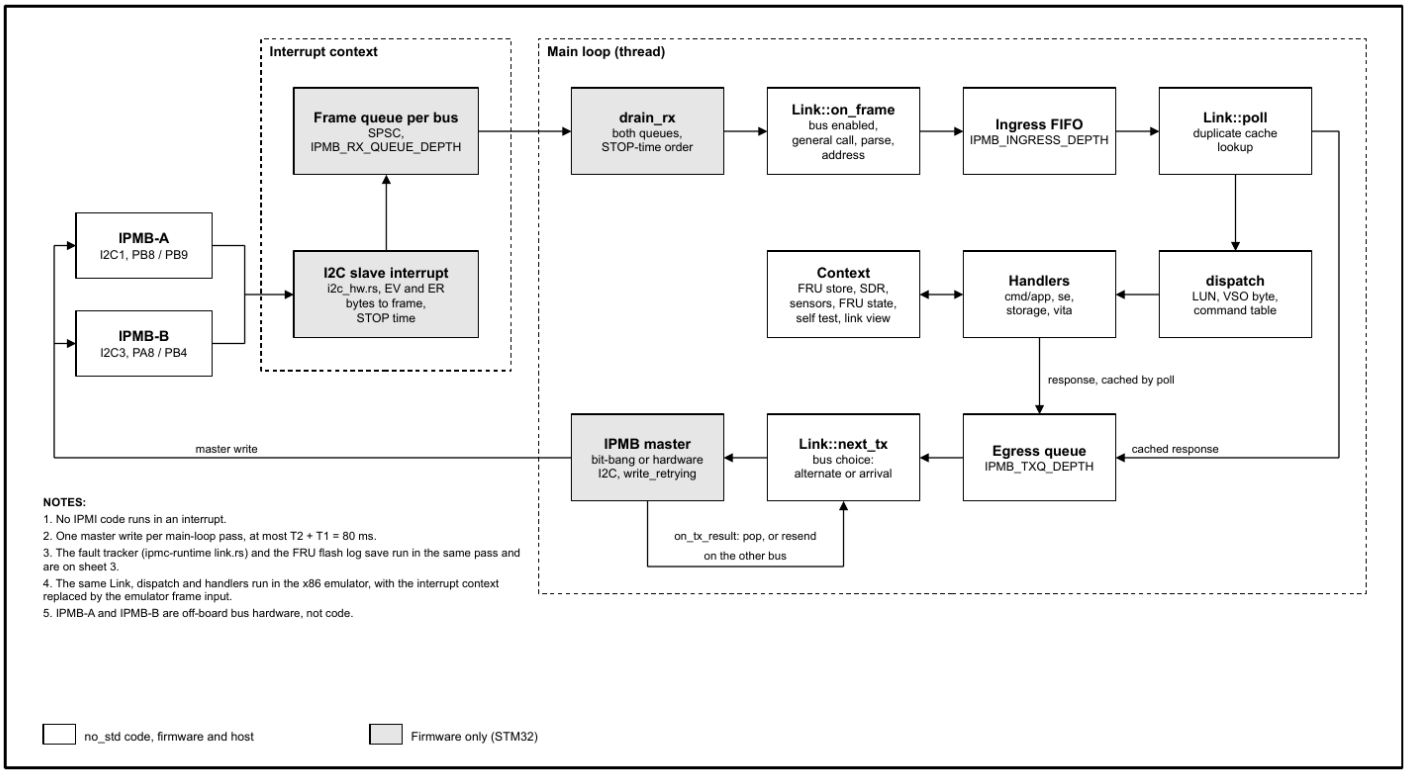
Sensor	Tier 2 assertion	Tier 2 deassertion	Tier 3 assertion	Tier 3 deassertion
0 FRU State	0053h	0000h	0073h	0000h
1 IPMB Link	000Fh	0000h	003Fh	0030h
2 FRU Health	0003h	0000h	0003h	0000h
3 FRU Voltage	0003h	0000h	0003h	0000h
4 FRU Temperature	0038h	0038h	0038h	0038h
5 Payload Test Results, 6 Payload Test Status	0000h	0000h	0000h	0000h
7 Payload Mode	none	none	0003h	0000h

The power-up event enables (`EventEnables::power_up`) take the same masks, so the records and the enables cannot disagree.

10. Data Flow and Protocol

10.1 Request path in the firmware

Figure 10-1. Firmware request path, receive to transmit (ipmc-software, sheet 2)



Source: `Documentation/diagrams/src/ipmc-software.js`, sheet 2.

- On receive, the I2C EV interrupt (`i2c_hw.rs` `on_event`) clears ADDR, starts a frame with the matched address byte, appends each data byte, and at STOP posts the frame with the SysTick time to that bus's SPSC queue. A frame longer than 33 bytes, an address-only write or a full queue is dropped and counted. The ER interrupt counts and clears error flags and drops the partial frame, except for ARLO alone (the hardware master lost arbitration), which is counted and keeps the reception, because the winner's frame may be addressed to this slave. A read of the slave answers FFh bytes.
- In the main loop, `IpmbLink::service` samples NVMRO, drains both queues in STOP-time order into `Link::on_frame` (at most 2 x `IPMB_RX_QUEUE_DEPTH` frames per pass; each frame goes into the capture log before the parser sees it), runs `Link::poll`, books slave errors on the fault streak, transmits at most one response, clears faulted buses whose lines have been idle-high for `IPMB_FAULT_CLEAR_MS`, and runs the FRU log save step.
- On transmit, `Link::next_tx` gives the oldest response and its bus; the master writes it (`write_retrying`: free-bus wait, then the transfer, arbitration retries on the same bus inside one T2 + T1 budget); the frame and its outcome go into the capture log; `Link::on_tx_result` pops the frame on success or moves it to the other bus for a resend.

10.2 Link layer (`ipmc-core/src/link.rs`)

`on_frame(bus, frame, now_ms)`:

1. A bus that is disabled or not fitted drops the frame (`BusUnavailable`), except IPMB-B while the power-up window is open (`Link::b_window_open`, `IPMB_COMPLIANCE` `NonStrict`): the frame is accepted, a request carries the window mark to its response, and the request or matched response is counted in `rx_power_up_window`.
2. A leading 00h (general call) is stripped and the request marked broadcast.
3. `ipmb::parse`; a parse error is counted per kind; a response is offered to the requester and kept when it answers the outstanding request (`rx_answers`), else dropped (a Tier-1 IPMC sends no requests, so it drops every response).
4. rsSA must be the own address; a broadcast must also be Get Device ID (`NotForUs` otherwise).
5. The request joins the ingress FIFO with its bus and arrival time; a full FIFO drops the new frame. Every drop is logged for `poll` to trace; the counters keep the totals.

`poll`: traces the drops, then takes requests oldest first while the egress queue has room. A request whose `Key` is in the duplicate cache within the window gets the cached response queued again and runs no handler (`replays` counter). Any other request is dispatched against a private staging queue; a pending Set IPMB State change is applied; the response is cached and queued with the arrival bus and time. After the requests, `poll` steps the requester timing (a retransmission falls due, or the request ends with no response), traces a request that ended, refreshes the link view in the context and calls `Extension::poll` once.

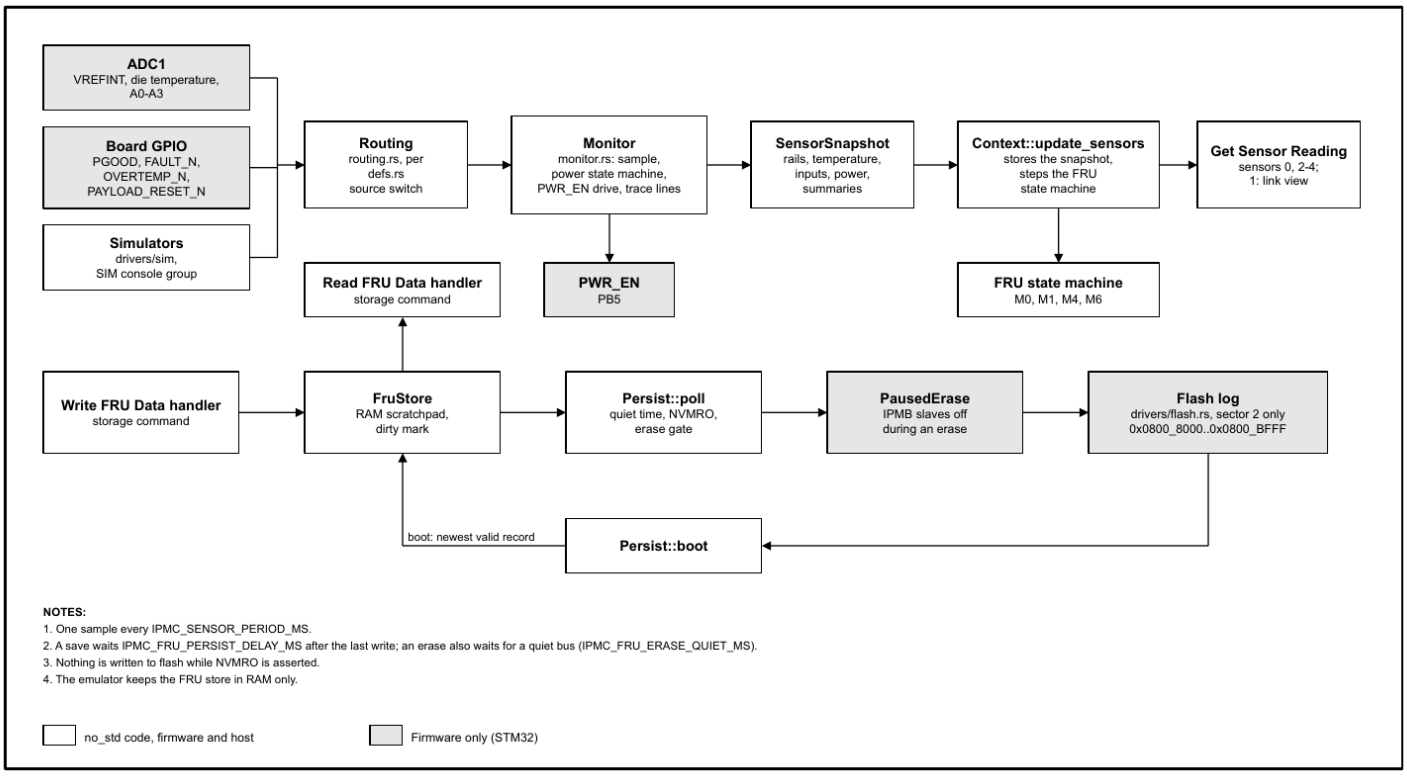
`next_tx`: a response handed out for the first time gets its bus now. ALTERNATE prefers the bus other than the latest transmission (resends count); ARRIVAL prefers the request's bus; when the preferred bus is not enabled the other enabled bus is used. A response whose assigned bus has since been disabled moves to the other bus. None when neither bus is enabled. The exception is a response to a request accepted on IPMB-B inside the power-up window: it goes out on IPMB-B whatever the policy says while IPMB-B is still in the window or has become enabled, and by the policy after an explicit Override or a local fault of IPMB-B. Only the head of the queue is handed out, so with IPMB-A faulted an unsent response to an IPMB-A request at the head holds back the window responses behind it (the existing queue order). A queued response always goes first; with none queued, a request that is due goes out on the bus other than the one of its own previous transmission (the other enabled bus when that one is disabled), and the same frame is handed out until its outcome is reported.

`on_tx_result(bus, ok, now_ms, trace)`: success pops the frame and counts a late response when the transmission completed more than 227 ms after the request's STOP. Failure counts `tx_fail` and the streak; when the other bus is enabled and the frame has fewer than `IPMB_TX_RETRIES` resends it moves there, otherwise it is dropped (`tx_dropped`) and the requester's retry gets the cached response. A request frame is booked by the requester instead: success starts its T3 wait, a failure waits at least T6 and retransmits on the other bus when it is enabled (`tx_requests` counts every request transmission).

Bus state per bus: `fitted`, `mode` (Isolated or Local Control, the Set IPMB State value) and `status` (local fault, `LocalStatus`). A bus is enabled when it is fitted, in Local Control and without a local fault. The IPMB-B power-up window (`Link::b_window_open`, `IPMB_COMPLIANCE`) is not part of the bus state: it lets `on_frame` accept requests on IPMB-B and `next_tx` answer them on IPMB-B while IPMB-B is still isolated, and it ends with the first Set IPMB State that sets IPMB-B or the first recorded fault of IPMB-B.

10.3 Sensor data, FRU state and FRU persistence

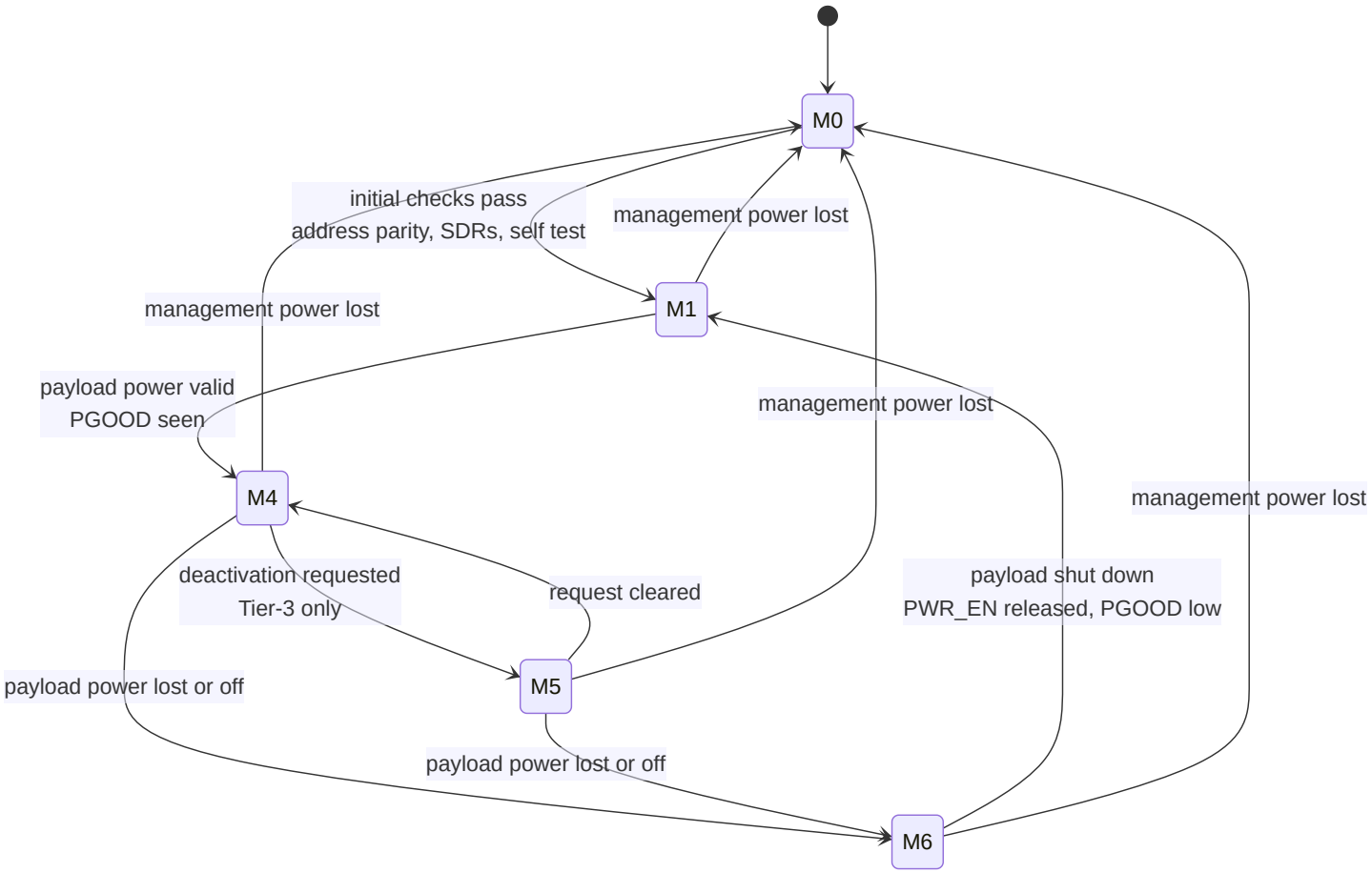
Figure 10-2. Sensor data, FRU state and FRU persistence (ipmc-software, sheet 3)



Source: `Documentation/diagrams/src/ipmc-software.js`, sheet 3.

Once per `IPMC_SENSOR_PERIOD_MS` the `Monitor` reads the inputs, steps the power state machine, drives PWR_EN, builds a `SensorSnapshot` and prints the change lines. `IpmbLink::feed_sensors` hands the snapshot to `ext::feed_sensors`, which calls `Context::update_sensors` and then the extension's `on_sample` hook with the FRU state transition that sample made. `update_sensors` stores the snapshot for Get Sensor Reading and steps the FRU state machine with the initial checks, "payload power valid" (power state On, so PGOOD was seen), "payload shut down" (PWR_EN released and PGOOD read deasserted) and, at Tier-3 only, the standing deactivation request (`Context::deactivation_requested`).

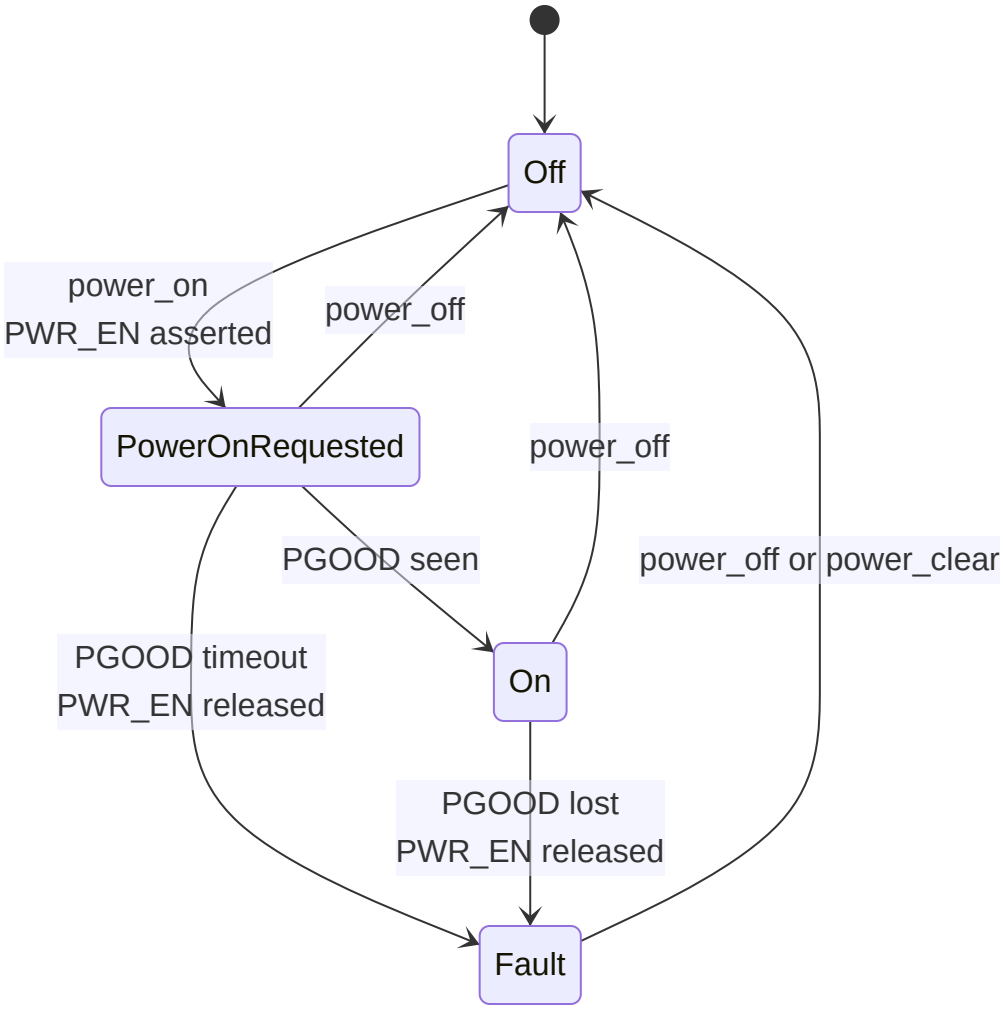
10.4 FRU 0 Operational State machine (`fru_state.rs`)



One transition per `step`. The state is reported only when Get Sensor Reading polls sensor 0 (a Tier-1 IPMC sends no events; the extension's `on_sample` hook sees each transition). The order inside `step` is: management power lost (to M0), then the transition of the current state; from M4 and M5, lost payload power (to M6) is tested before the deactivation request. Power that returns during M6 does not cancel the deactivation. M5 (FRU Deactivation Request) is reachable only at Tier-3, where the IPMC controls payload power (`tier::activation_control`): `Inputs::deactivation_requested` is the Tier-3 handlers' standing request (Rules 4.1.5.4.3-2 and -6) and is false below Tier-3, so M4 never leaves for M5 there. M7 is declared by the Chassis Manager, never by the IPMC for its own FRU, so it is not reachable. A self-test failure after M0 does not leave M1, M4 or M5. Management power is valid by construction on this board (the IPMC runs from it).

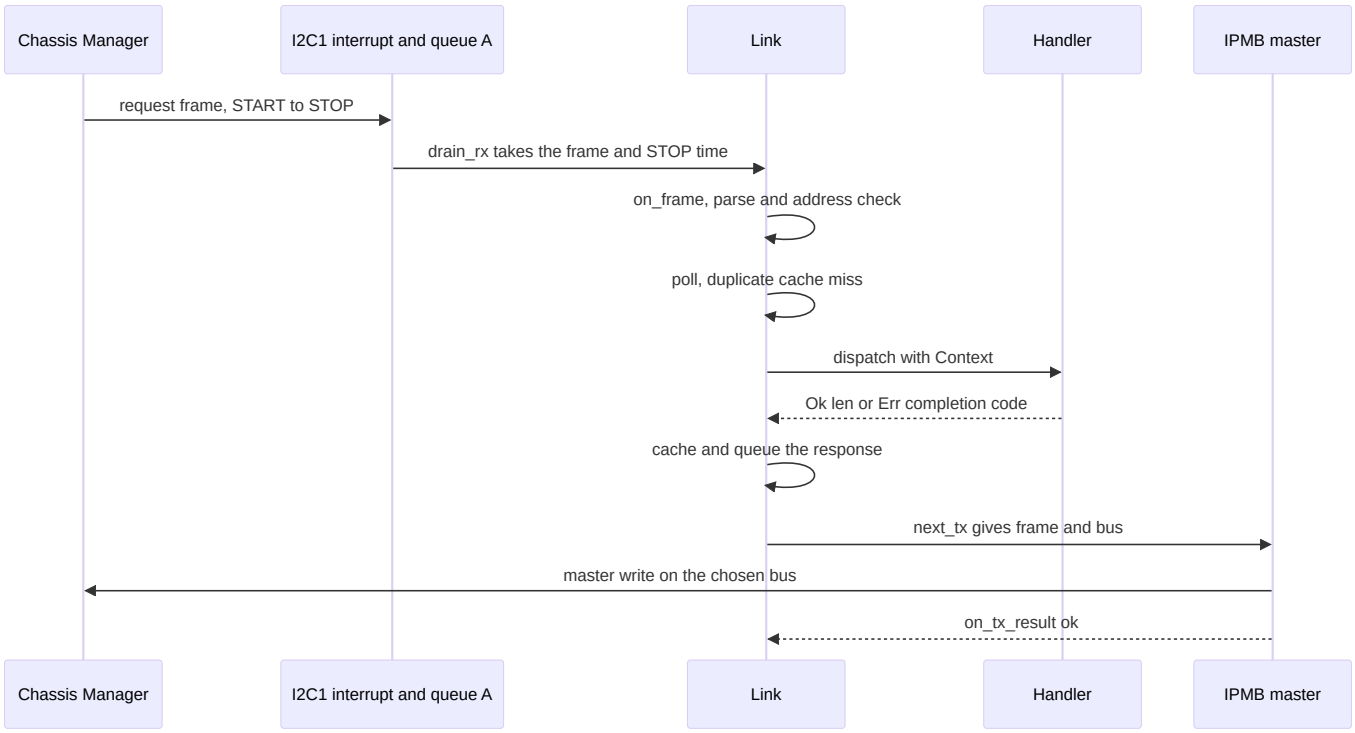
A payload power-on request is refused in M6 (`PowerOnRefused::DeactivationInProgress`) and, when an extension says so, while the FRU is Activation-Locked (`PowerOnRefused::ActivationLocked`, from `Extension::power_on_allowed`, Rule 4.1.5.4.2-1). `ext::check_power_on` is the one check for the boot power-on, the `pending_power` On hand-off (`Monitor::apply_power_request`) and the console `POWER ON`; the first refusal is returned and printed with its `name()`.

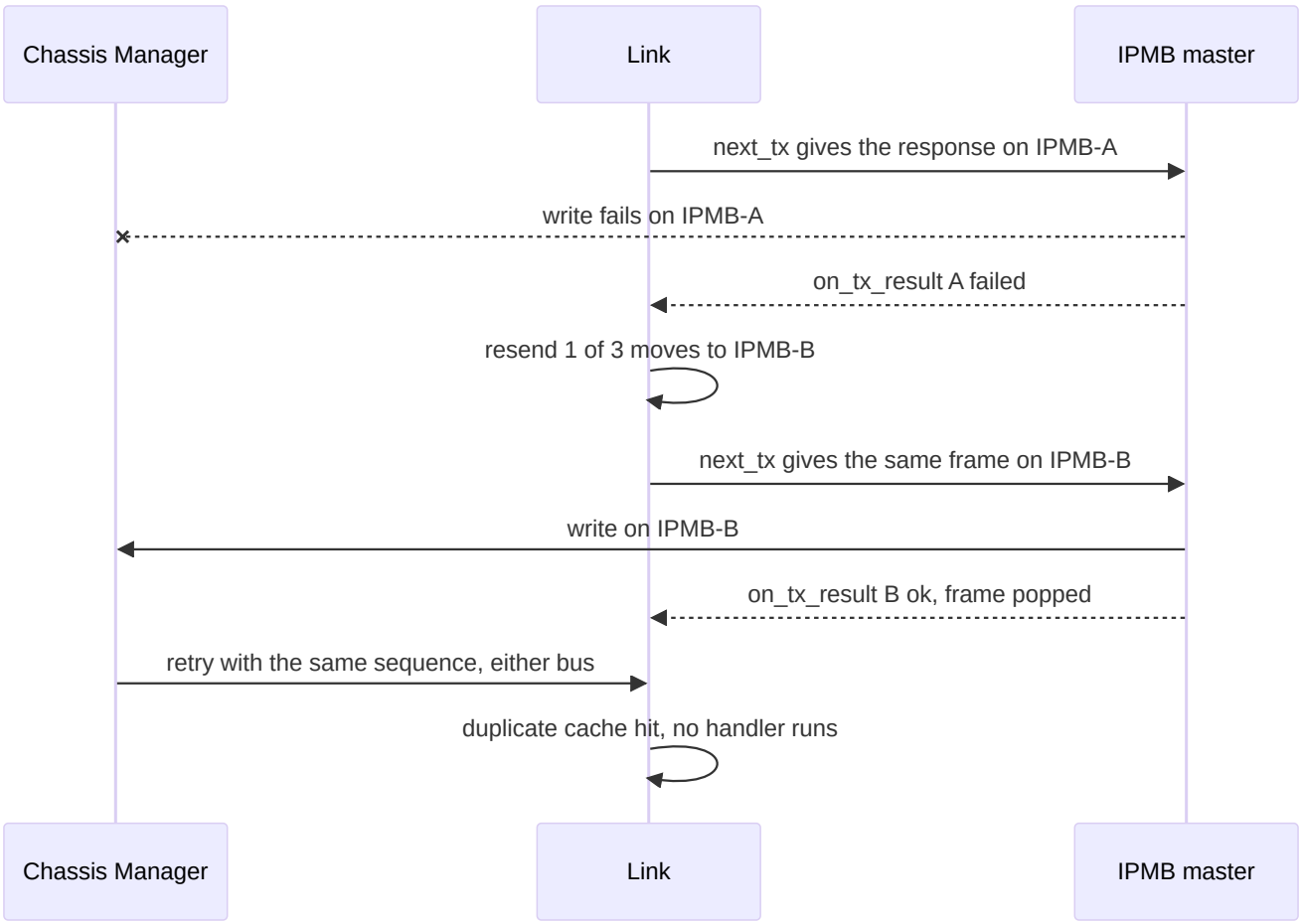
10.5 Payload power state machine (`ipmc-sensors/src/eval/power_fsm.rs`)



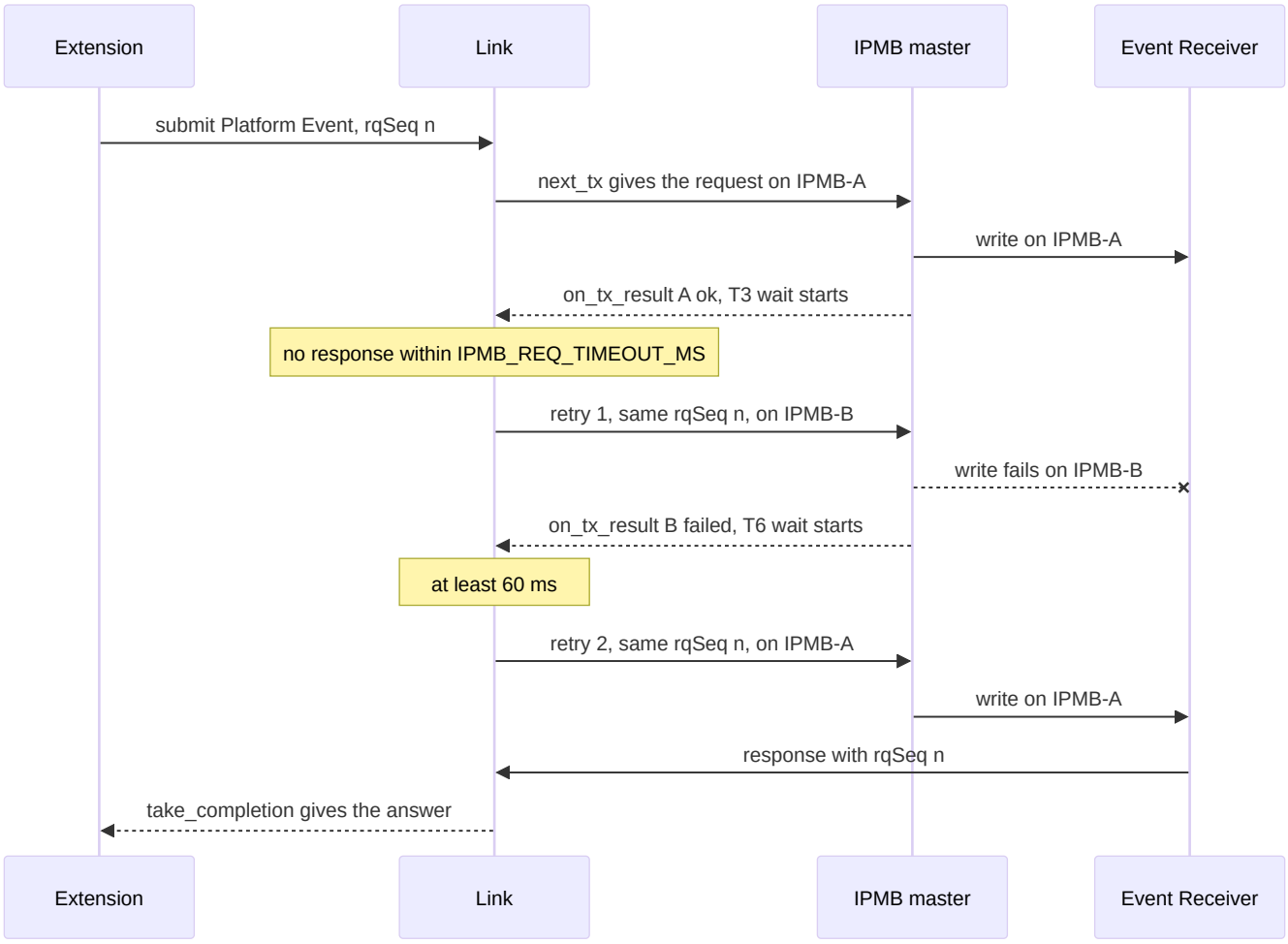
A fault stays latched until `POWER OFF` or `POWER CLEAR`, and `power_on` is refused while it is latched. A failed PWR_EN write latches `Fault` (EnableWrite) from any state. PGOOD seen in the same step as the timeout wins. Console `POWER ON` and a `pending_power` On request check `ext::check_power_on` first (section 10.4). The start-up request (`IPMC_POWER_AT_BOOT`) runs in M0, where the FRU machine's check always passes; the extension's activation lock can still keep the payload off at boot.

10.6 Request, response and resend sequences

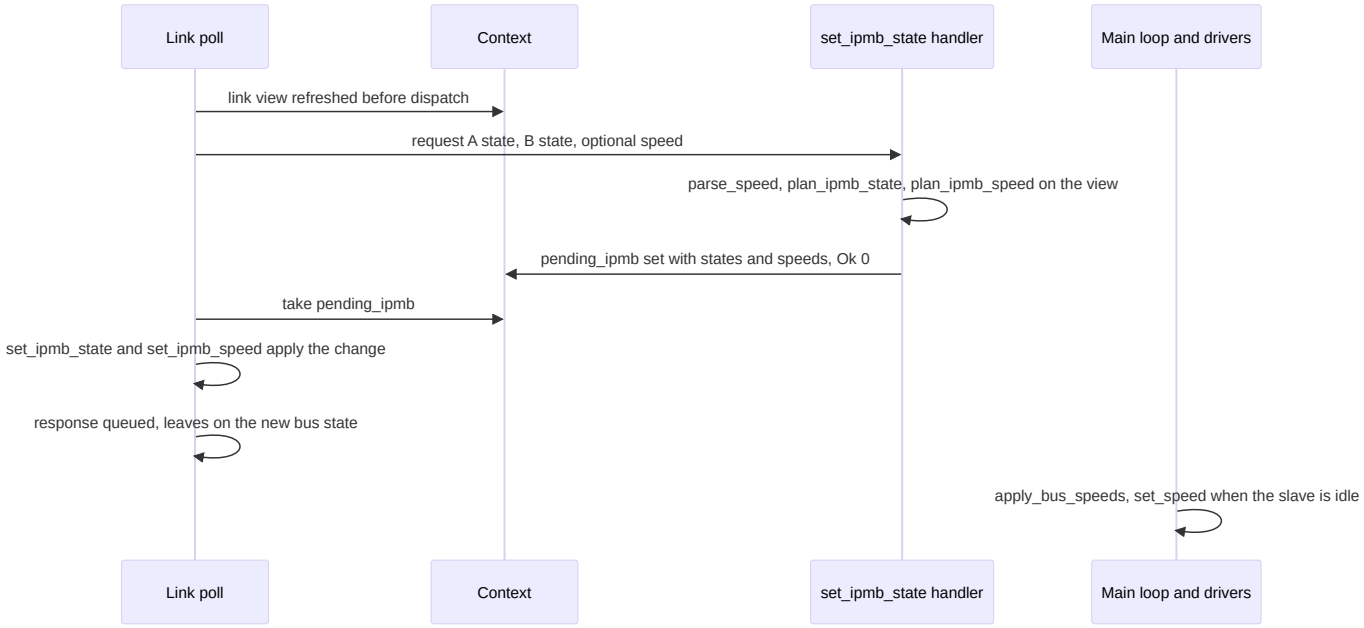




A request this IPMC sends (a build with the extension) keeps one sequence number through its retries. The first transmission goes out on IPMB-A and is acknowledged, but no response arrives within `IPMB_REQ_TIMEOUT_MS`; the retry goes on IPMB-B, the bus other than the previous transmission's, and fails on the wire; after at least T6 (60 ms) the next retry goes on IPMB-A and the response arrives.



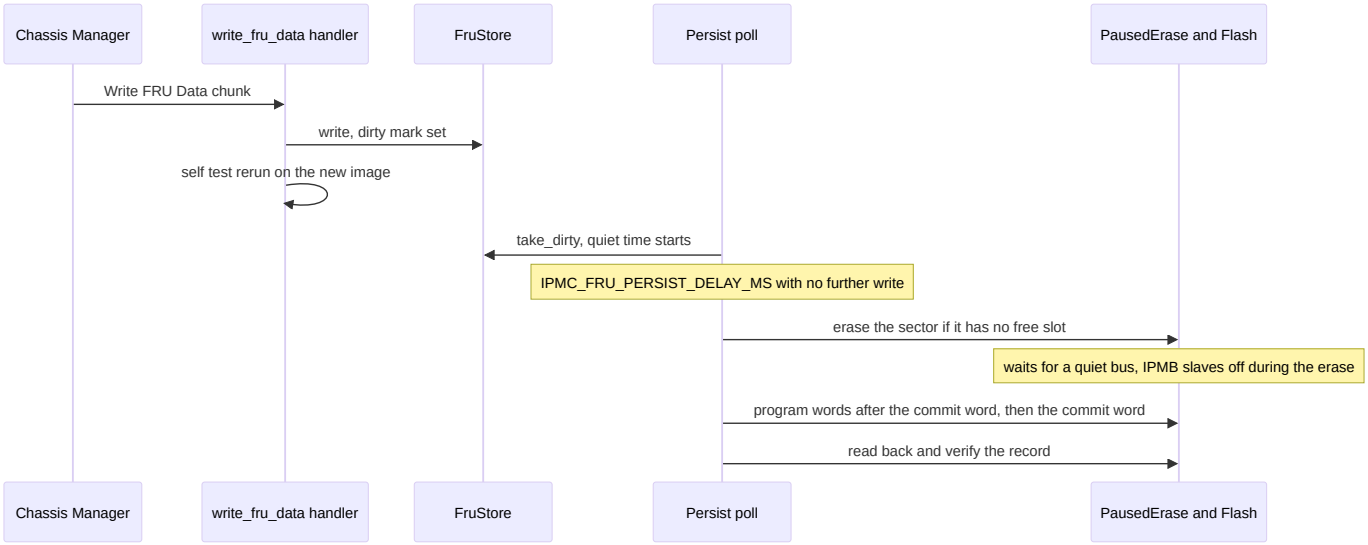
10.7 Set IPMB State hand-off



The speed byte is optional. Its bits 1:0 are the IPMB-A speed, bits 3:2 the IPMB-B speed and bits 7:4 are reserved: 00b is STANDARD-MODE, 100 kHz; 01b is FAST-MODE, 400 kHz, accepted only at Tier-3 (`tier::fast_mode`); 10b, 11b, a non-zero reserved nibble and 01b below Tier-3 answer CCh (Rule 4.1.1-14). `plan_ipmb_speed` also refuses 400 kHz for a bus that is not fitted (CCh); 100 kHz is always accepted. A request with an absent speed byte changes no speed. A refusal changes nothing, states included. `Link::poll` applies the state, then the speed, before the response is queued; the response leaves on the new state, still at the old speed.

The firmware applies the speed between transfers. `IpmbLink::apply_bus_speeds` runs at the end of `service`, compares each running bus's `bus_state(bus).speed` with the speed applied, and calls `IpmbMaster::set_speed`. That calls `i2c_hw::set_speed`, which suspends the slave only when it is idle (a busy or stopped slave returns false and the next pass asks again), loads CCR, TRISE and F/S for the speed and resumes the slave; the master write uses the new phases from the next frame on, and one console line `IPMB-A: running at 400 kHz` marks the change. Every restart of the link starts at STANDARD-MODE (Rule 4.1.1-4.3): at power-up, after a Cold Reset and after a Warm Reset (`Link::restarted`), and the drivers follow on the next pass. The System IPMB Link sensor and its FAST-MODE events follow the link state as soon as Set IPMB State is applied; the driver changes the clock once its slave is idle, so for a pass or two the sensor can show 400 kHz before the master runs at it. Bus recovery at start-up runs at 100 kHz.

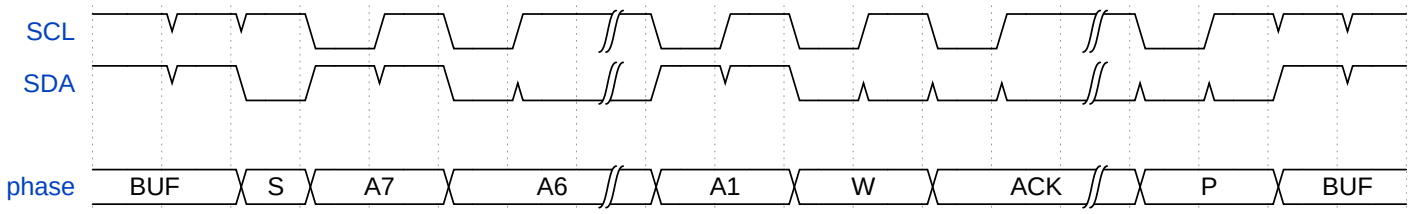
10.8 FRU save to the flash log



The store is one sector with 31 slots, so the erase comes once per 31 saves: the save that finds no free slot erases the sector and writes the newest image as the first record. That erase starts only when the egress queue is empty and no frame arrived for `IPMC_FRU_ERASE_QUIET_MS`; until then the image stays safe in RAM. `FRU RESTORE` erases at once when its record needs a sector, without that quiet gate, but answers `busy` and erases nothing while the link holds a request or a response. `PausedErase` suspends each running slave (a healthy one waits up to `IPMC_FRU_PAUSE_WAIT_MS` for idle; a faulted one is switched off at once), erases, resumes the slaves and reports the time the board was silent (a status line on the console and in the capture log), measured with the DWT counter. After the slaves are off the driver checks both receive queues and the link (`Link::request_in_flight`: an accepted request not yet answered, or a response not yet transmitted, on either bus); a frame or a request in that state defers the erase (`FlashError::Deferred`): the slaves resume, nothing is printed, the write stays pending with the failure counter unchanged, and the next pass retries; `systick::catch_up` adds the stalled time to the clock. On the bench the sector erase paused the IPMB for 198 ms (DS10086 Table 45, x32 programming: typical 250 ms, maximum 500 ms); a request that arrives in that time is missed and the Chassis Manager retries it. A power cut between the erase and the end of the first record's programming loses the stored image (section 17). A failed save keeps the write pending and retries once per delay. A per-unit field written through the application port takes the same path: `set_unit_field` marks the store dirty and `service` saves it after the quiet time.

10.9 Bit-bang master timing (`drivers/i2c_bitbang.rs`)

Bit-bang write: START, address 82h, ACK, STOP (one slot = 5 us half bit)



One slot is `IPMB_BITBANG_HALF_NS` (5000 ns). The bus is free when both lines have read high for tBUF (4.7 us) and the slave is idle. START: SDA falls while SCL is high (tHD;STA timed from the SDA write). Each bit: SDA set while SCL is low, at least tSU;DAT (250 ns) before SCL is released; SCL released and read back until high (clock stretching, up to `IPMB_STRETCH_TIMEOUT_US`); SDA sampled; after every released 1 bit a low SDA means another master holds the bus (`ArbitrationLost`). ACK: SDA released for the ninth clock and sampled low (the slave acknowledges). STOP: SDA low, SCL released, SDA released. Lines are only ever released or driven low (open-drain output: latch 1 releases the line, latch 0 drives it low); a pin is never driven high. The SCL period measured with the DWT counter is 12.0 us (83 kHz), below the 100 kHz limit.

At 400 kHz (FAST-MODE, Tier-3) the two phases differ: SCL low lasts `IPMB_BITBANG_FAST_LOW_NS` (1400 ns, at least the I2C fast-mode tLOW of 1300 ns) and SCL high `IPMB_BITBANG_FAST_HIGH_NS` (1000 ns, at least tHIGH of 600 ns). The cycle counts come from `ipmc-defs` `BitPhases`; the low phase also times the bus-free wait after a STOP (at least 1.3 us in fast mode) and the high phase the START setup and hold and the STOP setup. The slack above the minimum high time absorbs a slow rise, and a stretched clock extends the phase so SCL stays high for the minimum after it was seen high. The SCL period measured with the DWT counter is 6795 ns (about 147 kHz): the two phase times plus the loop overhead of the pin operations, with SCL low and high both above the fast-mode minimums. The hardware master runs CCR 35 (F/S = 1, DUTY = 0) and TRISE 13 at PCLK1 = 42 MHz; its measured SCL period is 2981 ns (about 335 kHz).

Peripheral hand-off: `i2c_hw::suspend` re-checks the slave is idle inside a critical section, masks the bus's EV and ER interrupts and clears PE, which releases both lines; the pins go to open-drain GPIO; after the STOP `i2c_hw::resume` restores the alternate function and re-arms the slave exactly as `init` does. While PE is 0 the slave cannot hear its bus; a frame another master sends at that moment is lost and recovered by the requester's retry.

10.10 Error handling

Error	Detection	Reporting	Recovery
Bad frame (length, checksum, address bit)	`ipmb::parse` in `on_frame`	`RxErrors` per kind, trace line	dropped; the requester retries

Error	Detection	Reporting	Recovery
Frame on a disabled bus, not for us, response	`on_frame`	`bus_unavailable`, `not_for_us`, `rx_response`	dropped; a request on IPMB-B inside the power-up window is accepted and counted in `rx_power_up_window` once queued
Ingress or egress full	`Fifo::push`	`ingress_full`, `egress_dropped`, `dispatch_errors`	dropped; requester retries
Transmit failure (NACK, busy, stretch, stuck line)	`MasterError` from the master	`tx_fail`, per-kind master counters, trace line	resend on the other bus up to `IPMB_TX_RETRIES`
Arbitration loss	released SDA reads low, or ARLO	`arb_retries`	same-bus retry up to `IPMB_ARB_RETRIES` within the write budget
Persistent bus fault	`IPMB_FAULT_STREAK` consecutive bus-fault errors (stretch timeout, SCL/SDA stuck, bus busy, slave BERR or TIMEOUT)	console `IPMB-A fault: <name>`, IPMB Link sensor local status	bus disabled (Rule 9.3-10); cleared after both lines read high for `IPMB_FAULT_CLEAR_MS`, with the BUSY workaround and a slave re-arm
Stuck BUSY flag	both lines high longer than T7 (3 ms) with BUSY set and no frame	`busy-errata-resets` counter	`clear_stuck_busy` (I2C peripheral reset sequence)
Bus fails at start-up	`bring_up` (recovery or slave init)	console line, local status	bus disabled; restarted after the lines read idle-high
Request without a response (a build with the extension)	`Requester` T3 timeout after each transmission	trace line `request seq .. no response after N transmissions`	retried up to `IPMB_REQ_RETRIES` times with the same sequence number; then the completion is no response and the extension decides
Payload power-on refused	`ext::check_power_on`: the FRU is in M6, or the extension reports the FRU Activation-Locked	console line `power on request refused: ...`, `Power at boot refused: ...`, or the refusal text of console `POWER ON`	no power change; allowed again once the FRU is in M1 or activated
Pending reset blocked	`ResetGate`: response still queued or FRU write unsaved	trace line after `IPMC_RESET_WAIT_MS`	the reset runs anyway at that time

Error	Detection	Reporting	Recovery
Corrupt FRU image	`selftest::run`	Get Self Test Results 57h with bit 5	FRU stays in M0 when found at start-up; Write FRU Data or `FRU RESTORE` repairs it
Flash program, erase or verify failure	`FlashError`	console `FRU log: save failed: <name>`, `FRU STORE`	write stays pending, retried once per delay
Runtime image fails its check at a reset	`header::check` in the bootloader	`Runtime image:` and `Image check:` lines, `Decision: staying in the bootloader: <reason>`	the bootloader waits for `WAKEUP`; an update or `ERASE FRU` and `RUN` recovers (section 16)
PGOOD timeout or loss	`power_fsm`	console event line, `POWER STATUS`; FRU Health is bad only while PWR_EN is asserted without PGOOD, so it returns to good once the fault releases PWR_EN	latched until `POWER OFF` or `POWER CLEAR`
Pin-claim conflict, UART failure	`ports::claim`, `uart::init`	console line naming both owners, or 5 Hz LED	fail-stop (park); an application port failure other than a pin conflict prints `APP UART not started` and the board runs without it
HardFault, unhandled interrupt, panic, main-loop hang	`bsp/fault.rs` handlers, `bsp/watchdog.rs`	console line (`HARDFAULT pc=`, `UNHANDLED IRQ`, `PANIC <file>:<line>`), then after the reset `Last reset: independent watchdog` and `Last fault:`; `SYS STATUS Watchdog:`	the handler records and parks; the watchdog resets the processor within `IPMC_WATCHDOG_MS` (the bootloader has none)
UART receive ring full, overrun, transmit timeout	`uart.rs` `service`, `wait_for_room`	per-port counters in `SYS STATUS`	the byte is dropped (receive) or the bounded wait ends (transmit)
UART byte with a framing, noise or parity error	`uart.rs` `service` (SR FE, NF, PE)	`line errors` in `SYS STATUS`, `Port errors:` in the bootloader `INFO`	the byte is queued as received; the line it belongs to fails its own check (an unknown command, a record checksum)

Error	Detection	Reporting	Recovery
Trace line without room in the console transmit ring	`ConsoleText::line_or_drop`	`tx lines dropped` in `SYS STATUS`	the line is dropped whole and never stalls the IPMB; the capture log still holds it
Application answer longer than the transmit ring has room for	`ipmc-runtime` `app_port.rs` `drain`	none (the answer is only late)	the output stays in its buffer and leaves as the ring has room; no new request is taken meanwhile, and the main loop never waits
Capture log full	`CaptureLog::add`	`overwritten` in `DEBUG STATUS` and in the dump header	the oldest whole records are dropped

10.11 Capture log (`ipmc-log`, `ipmc-runtime` `link.rs`, `log_cmds.rs`, `debug_cmds.rs`)

The capture log is the field-service record of the IPMB. A person enables it on a target system, reproduces a fault and mails the dump back; the dump reads the same on the console, in a spreadsheet and, through `tools/capture/capture2pcap.py`, in Wireshark. The log lives in RAM (one `ByteRing` of `IPMC_LOG_SIZE` bytes, 8191 usable by default) and is empty after a reset.

What goes in:

- every whole IPMB frame taken from a driver queue, before the parser (so a frame the parser rejects is there too), with status `rx`;
- every frame handed to the master, with its outcome: `ok`, or the cause of the failure;
- every frame handed to the link by `DEBUG INJECT`, with status `inject`;
- the trace lines of the link, the dispatcher and the requester, the status lines (FRU log saves, bus faults and speed changes, resets), the lines of `DEBUG IPMI` (`console request ...`, `console response ...`) and the application port lines (`app request ...`, `app answer ...`).

`IPMC_IPMI_TRACE` false removes the link and dispatcher trace lines at compile time, so they are absent from the capture log as well as from the console; the frames, the status lines and the console and application lines stay. `DEBUG TRACE IPMI OFF` stops only the console copy: the log keeps the lines. `DEBUG TRACE I2C ON` (or `IPMC_I2C_TRACE_AT_BOOT` true) prints one console line per frame, the same row `DEBUG DUMP TRACE` prints, even while the log is off.

Record layout (little-endian): a 6-byte header, then the payload.

Byte	Content
0	flags: bit 0 kind (0 frame, 1 text), bit 1 bus (0 A, 1 B), bit 2 direction (0 in, 1 out), bits 6:3 status code
1	payload length: a frame at most `IPMB_MSG_MAX` (32) bytes, a text line at most `IPMC_LOG_TEXT_MAX` (160)
2..5	`time_ms`, u32, the SysTick time
6..	the payload: the IPMB frame as it is on the wire, or the line as ASCII

The status code is 0 received, 1 injected, 2 sent, then 3 + n for a failed transmission with the cause n: address NACK, data NACK, arbitration lost, bus busy, stretch timeout, line stuck, unspecified. The record index is not stored: the i-th held record is `total - records + i`. A record that does not fit drops the oldest whole records first and counts them. A 23-byte frame takes 29 bytes of the ring.

The DEBUG group (help text as in `ipmc-runtime` `console.rs`):

Command	Help text
`DEBUG STATUS`	Debug switches, capture log use, event receiver, duplicate cache
`DEBUG LOG {STATE}`	Capture log ON or OFF; no argument shows its state

Command	Help text
`DEBUG DUMP {FORMAT}`	Print the capture log: CSV (default) or TRACE; any key stops it
`DEBUG CLEAR`	Empty the capture log
`DEBUG TRACE {WHAT} {STATE}`	Console trace: IPMI (request, link and event lines) or I2C (one line per frame), ON or OFF
`DEBUG EVENTS {STATE}`	Event messages ON (receiver 20h) or OFF (receiver FFh)
`DEBUG IPMI [NETFN] [CMD] {DATA...}`	Run one IPMI request locally, as from 20h; nothing goes on the bus
`DEBUG INJECT [BUS] {BYTES...}`	Hand a whole IPMB frame to the link as if received on bus A or B
`DEBUG CACHE {ACTION}`	Duplicate cache: no argument shows the entries, CLEAR empties it
`DEBUG BOOTLOADER {CONFIRM}`	Restart into the bootloader; CONFIRM needed while payload power is on
`DEBUG WATCHDOG {TEST} {CONFIRM}`	Watchdog state; HANG or PANIC with CONFIRM tests the reset

A local request (`DEBUG IPMI` , the application port `IPMI` , the start-up events-off request) reads the live NVMRO pin first, as the main loop does, so it decides like a bus request.

`DEBUG IPMI` and `DEBUG INJECT` take hex bytes (`06` , `0x06` and `6` are the same). `DEBUG IPMI` runs the request through the dispatcher (`IpmbLink::local_request`) and prints `-> CC 00h, 15 bytes: 01 80 ... (App Get Device ID)` . `DEBUG INJECT` hands the frame to the link as received on bus A or B; the link answers a request it accepts on the bus like any other.

`DEBUG DUMP` prints one header line, the rows and an end line. The header line is

```
# IPMC capture: ipmc-nucleo-f401re v0.1.0, ipmc-core v0.1.0, IPMB address 82h, Tier 1, uptime 123456 ms
```

and the end line is `# end of capture` . A dump runs as a job: a few rows per main-loop pass, while the console transmit ring has room for the longest row, so the main loop never waits for the wire. It prints the records held when it started; a record the ring overwrote meanwhile is skipped and counted in `# <n> records overwritten during the dump` .

The CSV format has one header row and one row per record, with the columns

Column	Content
`rec`	the record index since the last clear
`time_ms`	the SysTick time of the record
`type`	`FRAME` or `TEXT`
`bus`	`A` or `B` ; empty for text

Column	Content
`dir`	`IN` or `OUT`; empty for text
`status`	`rx`, `inject`, `ok`, `addr-nack`, `data-nack`, `arb-lost`, `bus-busy`, `stretch`, `line-stuck` or `failed`; empty for text
`len`	the frame length in bytes; empty for text
`summary`	quoted; for a frame `<src>h-><dest>h <NetFn> <Req or Rsp> <command> seq <n>h` and `CC <cc>h` for a response, or `unparsed: <error>`; for text the line
`bytes`	the frame bytes in hex, upper case, one space apart; empty for text

```
rec,time_ms,type,bus,dir,status,len,summary,bytes
12,123456,FRAME,A,IN,rx,7,"20h->82h App Req Get Device ID seq 01h",82 18 66 20 04 01 DB
13,123459,TEXT,,,,,"FN(06h) CMD(01h) Get Device ID -> CC 00h, 15 bytes",
14,123461,FRAME,A,OUT,ok,23,"82h->20h App Rsp Get Device ID seq 01h CC 00h",20 1C C4 82 04 01 00 01 80
```

`DEBUG DUMP TRACE` prints the same records as Wireshark-style text rows: `#<index>`, the time in seconds with milliseconds, `IPMB-A` or `IPMB-B` and the direction, `<src>h -> <dest>h`, the NetFn name with `Req` or `Rsp` in a 14-character column, the command name in a 22-character column, the tail (`seq <n>h`, `CC <cc>h`, and the status word of an injected or failed frame), and after `|` the frame bytes. A text record is `#<index>`, the time, `TEXT` and the line.

```
#12   123.456  IPMB-A  IN   20h -> 82h  App Req       Get Device ID          seq 01h          | 82 18 6
#13   123.459  TEXT    FN(06h) CMD(01h) Get Device ID -> CC 00h, 15 bytes
#14   123.461  IPMB-A  OUT  82h -> 20h  App Rsp       Get Device ID          seq 01h CC 00h   | 20 1C C
```

A command name comes from the Tier-1 table, then from the extension, then from Platform Event, the one request the IPMC sends; an unknown command prints as `CMD 5Ah`, a NetFn without a name as `NetFn 3Ah`.

The Wireshark path: save the terminal log of a CSV dump and run `python -I tools/capture/capture2pcap.py dump.txt -o dump [--csv]`. The tool reads only the lines between `# IPMC capture:` and `# end of capture` (a log may hold several dumps and other terminal text) and skips a `TRACE` block. It writes `dump.pcap`, a little-endian pcap with link type 260 (`LINKTYPE_IPMI_HPM_2`): one packet per record, stamped with the record's `time_ms`. Each packet holds one block of the Wireshark IPMI trace format: byte 0 the block type (0 frame, 2 text line) in bits 7:4 and the channel (0) in bits 3:0, bytes 1..4 the seconds and bytes 5..6 the milliseconds, byte 7 the data type 01h (IPMB 1.0), bytes 8..9 the flags (bit 6 IPMB-B, bit 7 direction to the controller, an `IN` frame), byte 10 the length and then the data, the whole IPMB frame from rsSA or the ASCII line. A failed `OUT` frame is the frame followed by a text packet `tx failed: <status>`. With `--csv` the rows are also copied to `dump.csv`.

11. Software API

11.1 Link layer (`ipmc_core::link::Link`)

API	Inputs	Output	Notes
`Link::new(config, &addr)`	`LinkConfig { b_fitted, tx_policy, b_power_up, requests_allowed }`, `AddrInfo`	`Link`	`const`; power-up bus state; unusable address isolates both buses; `with_first_request_seq` sets the first rqSeq, `next_request_seq` reads it; `restarted(&addr, now_ms)` builds the Link of a Cold or Warm Reset; `warm_reset(ctx, ext, gate, now_ms, reapply)` is the one order of a Warm Reset; `LinkConfig::bench(tx_policy, requests_allowed)` is the configuration of the tests
`on_frame(bus, frame, now_ms)`	bus, whole frame (own address or 00h first), STOP time	`Result<(), FrameDrop>`	main loop only; copies at most `IPMB_MSG_MAX` bytes; never traces
`poll(ctx, ext, now_ms, trace)`	dispatch `Context`, `&mut dyn Extension`, time, trace sink	requests taken	runs handlers (the extension for a command the table lacks); stops when the egress queue is full; steps the requester and calls `ext.poll`
`next_tx()`		`Option<(Bus, &[u8])>`	response first, else a due request; frame stays handed out until `on_tx_result`
`on_tx_result(bus, ok, now_ms, trace)`	outcome of the frame handed out	`Result<(), LinkError>`	`NoTxInFlight`, `TxBusMismatch` change nothing
`set_ipmb_state(a, b)`	`Option<BusMode>` per bus	`Result<(), LinkError>`	through `plan_ipmb_state`, all or nothing
`set_ipmb_speed(a, b)`	`Option<BusSpeed>` per bus	`Result<(), LinkError>`	through `plan_ipmb_speed`, all or nothing; `BusNotFitted` for 400 kHz on a bus that is not built in
`set_local_status(bus, status)`	`LocalStatus`		the firmware's persistent-fault decision

API	Inputs	Output	Notes
readers			`view`, `bus_state`, `counters`, `config`, `own_address`, `evictions`, `late_responses`, `dispatch_errors`, `egress_empty`, `egress_dropped`, `sensor_data`

11.2 Dispatch (`ipmc_core::dispatch`)

```
pub type Handler = fn(&mut Context, &Frame, &mut [u8]) -> Result<usize, u8>;
```

`Ok(n)`: n response data bytes written after the completion code. `Err(code)`: that completion code with no data. For NetFn 2Ch the frame data and the output slice both exclude the VSO byte.

`Context` field	Owner (only writer)	Read by
`addr: AddrInfo`	`addr::resolve` at start-up; `nvmro_asserted` by the NVMRO sample	Write FRU Data, Get FRU Address Info, initial checks
`sensors: SensorSnapshot`	`update_sensors`	Get Sensor Reading 2..4
`fru: FruStore`	Write FRU Data; `set_fru` at boot and `FRU RESTORE`	Read FRU Data, self test, `Persist::poll`
`sdr: SdrRepository`	Reserve Device SDR Repository (reservation)	Get Device SDR, Get Device SDR Info, locator
`self_test: SelfTest`	`Context::new`, `set_fru`, Write FRU Data	Get Self Test Results, initial checks
`fru_state: FruMachine`	`update_sensors`	Get Sensor Reading 0
`link: LinkView`	`Link::poll` before every dispatch	Get VSO Capabilities, Set IPMB State, Get Sensor Reading 1
`pending_ipmb: Option<PendingIpmbState>`	Set IPMB State handler	`Link::poll` (applies and clears)
`event_enables: EventEnables`	`Context::new` (power-up values by Tier); an extension handler	`read_sensor` (status bit 7), the event generator

`Context` field	Owner (only writer)	Read by
`pending_reset: Option<ResetRequest>`	an extension handler (Cold or Warm Reset)	`ResetGate` owner: `IpmbLink::service_reset`, which the firmware main loop and the emulator run (cleared by the reset; `Context::warm_reset` clears it for a Warm Reset)
`pending_power: Option<PowerRequest>`	an extension handler (payload power)	the owner of the payload power, through `Monitor::apply_power_request` (On asks `ext::check_power_on`)
`pending_payload_reset: bool`	an extension handler (payload reset)	the owner of PAYLOAD_RST_OUT_N: `PayloadResetPulse` (section 8.2)
`deactivation_requested: bool`	an extension handler (activation)	`update_sensors`, at Tier-3 only (M4 to M5)
`payload_mode: PayloadMode`	an extension handler (payload mode)	`read_sensor` (sensor 7)
settings byte, in `fru` (`nv_settings`, `set_nv_settings`)	an extension handler (policy bits), the boot load (`set_fru`)	the extension, `Persist::poll` (saved with the image)

Other entry points: `Context::new(addr)`, `set_fru(store)`, `initial_checks()`, `update_sensors(now_ms, &snapshot, trace) -> Option<Transition>`, `read_sensor(number, &buses, out) -> Result<usize, SensorReadError>` (`read_sensor_at(tier, ..)` for a given Tier; `out` holds `sensor::READING_LEN_MAX` = 4 bytes), `warm_reset(link_view)`, `dispatch(ctx, req, arrival, arrival_ms, queue, trace)` (the Tier-1 table and `NoExtension`), `dispatch_with(table, ext, ctx, ...)` and `build_response_with` (the one frame builder behind `dispatch_with` and the tests).

11.3 Hardware-facing traits

Trait	Crate	Methods	Implemented by
`FlashLog`	`ipmc-fru::persist`	`sector_len`, `read`, `program` (whole words), `erase`, `millis`	`bsp::flash::Flash`, `PausedErase`, the test flash in `persist/tests.rs`
`IpmbMaster`	firmware `bsp::ipmb_master`	`state`, `state_mut`, `write_once`; provided `write`, `pins`, `recover_if_stuck`, `lines_high`, `clear_bus_fault`, `stats`, `speed`, `set_speed`	`BitbangMaster`, `HwMaster`
`AnalogSource`	`ipmc-sensors::hal`	`vdda_mv`, `die_temperature_dc`, `rail_mv`	ADC1, simulator, INA226, MCP3208
`DigitalInputs`	`ipmc-sensors::hal`	`level_high`	board GPIO, simulator, PCA9555

Trait	Crate	Methods	Implemented by
`PowerSwitch`	`ipmc-sensors::hal`	`set_enable`, `is_enabled`	board GPIO, simulator, PCA9555
`ResetOutput`	`ipmc-sensors::hal`	`set_reset(asserted)`	board GPIO (PA9, open drain), the test pin of `reset_pulse.rs`
`Trace`	`ipmc-core::trace`	`line(fmt::Arguments)`	firmware console, emulator stdout, `NoTrace`, `RecordingTrace`
`Extension`	`ipmc-core::ext`	`command`, `poll`, `on_sample`, `warm_reset`, `power_on_allowed`	`NoExtension`; the Tier-2/3 extension through `ipmc_ext::Active`
`MenuOutput`	`ipmc-menu`	`write_str`	firmware UART output
`AppHost`	`ipmc-app::host`	`version`, `status`, `fru_state`, `unit_field`, `set_unit_field`, `ipmi`, `command_name`, `save_state`, `request_bootloader`	`LinkHost` in the firmware `app_port.rs`, `FakeHost` in the tests
`ImageSource`	`ipmc-boot::header`	`read(addr, out)`	`FlashImage` (flash), `SliceImage` (a buffer, `fwimage` and the tests)
`FlashOps`	`ipmc-boot::load`	`erase_sector`, `program` (and `read` through `ImageSource`)	`FlashImage` in `src/boot/flash_image.rs`, the RAM model in the tests
`BootHw`	`ipmc-boot::session`	`now_ms`, `activity`, `port_counts` (and `FlashOps`)	`Hw<B>` in `ipmc-boot/src/run.rs` (over the board's flash, clock, LED and ports), the test bench
`TextOut`	`ipmc-boot::text`	`put(text)`	`SerialText` (a `SerialPort`), `Both`, the string collectors of the tests and `fwimage`
`Names`	`ipmc-core::names`	`command(netfn, cmd)`	`CoreNames` (Tier-1 table, extension, sent requests)

11.4 Concurrency model

- Single core, one thread (the main loop) plus interrupts. Interrupt handlers (I2C EV/ER at NVIC priority 1 << 4, above USART1 and USART2 at 2 << 4, and SysTick) move bytes, stamp times and bump counters; the SysTick handler also evaluates the LED pattern and writes PA5. The I2C handlers run above the UART so a slow handler does not stretch SCL past T8.
- Hand-off from interrupt to thread is lock-free: one `heapless::spsc` queue per bus, producer in the interrupt, consumer in `ipmc-runtime` `link.rs`. Driver statics use `Mutex<RefCell<Option<T>>>` through `static_slot.rs`, every access inside a short critical section.

- Counters shared between interrupt and thread are single-word atomics with Relaxed ordering.
- `Link`, `Context`, handlers, the console and the trace sink never run in an interrupt. Nothing in `ipmc-core` blocks; the blocking calls are the master write (bounded by T2 + T1), a console message while the transmit ring is full (86.8 us per byte, bounded per byte; otherwise the message goes into the ring and the interrupt sends it), a sensor sample (about 2 ms of polled ADC conversions, 23.4 us each), a flash word program (up to 100 us per word) and a flash erase (at most 500 ms, IPMB slaves off).
- A flash erase or program stalls every flash access, interrupt handlers included, because code runs from the same bank.

12. Build, Integration, and Configuration

12.1 Toolchain

Rust 1.99.1 with the GNU host toolchain, fixed by `rust-toolchain.toml` at the repository root (channel, the components rustfmt, clippy and llvm-tools, and the targets `thumbv7em-none-eabihf` and `x86_64-pc-windows-gnu`; rustup reads it in both workspaces and installs the toolchain on first use); `gen.ps1` uses `arm-none-eabi-objcopy` for the images and `arm-none-eabi-size` for the section sizes. cargo is not on PATH in every shell (Bash: `export PATH="$HOME/.cargo/bin:$PATH"`). The firmware release profile uses `opt-level = "s"`, `lto = true`, `codegen-units = 1` and keeps debug information; the bootloader uses the profile `boot` (`inherits = "release"`, `opt-level = "z"`, `panic = "abort"`).

```
# host crates: format, lint, test
cargo fmt --all -- --check
cargo clippy --workspace --all-targets -- -D warnings
cargo test --workspace
# firmware (its own workspace)
powershell -NoProfile -ExecutionPolicy Bypass -File crates/ipmc-nucleo-f401re/gen.ps1 build
powershell -NoProfile -ExecutionPolicy Bypass -File crates/ipmc-nucleo-f401re/gen.ps1 flash
# lint every firmware binary (the runtime and the bootloader)
(cd crates/ipmc-nucleo-f401re && cargo clippy --release -- -D warnings && cargo clippy --profile boot -
powershell -NoProfile -ExecutionPolicy Bypass -File crates/ipmc-nucleo-f401re/gen.ps1 console "ipmb sta
# emulator
cargo run -p ipmc-emu -- replay local_only/vectors/sbc_ab.vec --b-fitted 1 --b-enabled 1 --policy alter
cargo run -p ipmc-emu -- serve --b-fitted 1
cargo run -p ipmc-emu -- sdr
```

The two build modes (section 4) use the same commands. `IPMC_TIER23`, `IPMC_TIER2` and `IPMC_TIER3` at the top of `defs.rs` select the build: all three `true` (the default) is the commercial Tier-3 build and needs the extension folder next to the repository, all three `false` is the public Tier-1 build. `gen.ps1 build|flash -Eval` makes the evaluation image (section 8.4). The numbers of the capture replay (section 13.2) belong to the Tier-1 build.

`gen.ps1 build` makes, in `crates/ipmc-nucleo-f401re/target/thumbv7em-none-eabihf/release`:

1. the bootloader: `cargo build --profile boot --bin ipmc-boot` (always `IPMC_EVAL=0`), then `ipmc-boot.bin` (`objcopy -O binary --gap-fill 0xFF`) and `fwimage size`, which fails above 32 768 bytes and warns above 24 576;
2. the runtime: `cargo build --release --bin ipmc-nucleo-f401re`, then `ipmc-nucleo-f401re.app.img` (the bytes from `0x0800_C000`);
3. `fwimage patch` writes the image length and the SHA-256 digest into the header (`app_header.bin`), `objcopy --update-section .app_header=...` gives the patched ELF `ipmc-nucleo-f401re.elf`, the image is made from it again and `fwimage check` runs the bootloader's own image check on it; a mismatch stops the build;
4. `ipmc-nucleo-f401re.hex` (`objcopy -O ihex` of the patched ELF) is the file for `LOAD`;
5. `fwimage combine` writes `ipmc-board.bin`.

Every runtime file carries `-eval` after `gen.ps1 build -Eval`, and every file carries `-field` after `-Field` (`IPMC_FIELD=1` for both cargo calls: the boot countdown is `IPMC_BOOT_COUNTDOWN_FIELD`; the two combine as `-eval-field`). `gen.ps1`

`flash` and `flash -Drive` copy `ipmc-board.bin` to the `NOD_F401RE` drive and print `Flash OK` only after the copy, the drive check and the `FAIL.TXT` check pass. `flash -Swd` programs the patched runtime ELF with OpenOCD (`openocd -f interface/stlink.cfg -f target/stm32f4x.cfg -c "program <ELF> verify" -c reset -c exit`); `-Swd -Bootloader` programs the bootloader ELF as well. `-Swd` never falls back to the drive copy, `-Drive` never uses OpenOCD (both together are refused), and a SWD program that starts and fails is an error. SWD needs the WinUSB driver on the ST-LINK debug interface. The paths leave the FRU store in different states (section 9.1). `gen.ps1 console <command>` reads a reply until the console has been quiet for 600 ms, up to 60 s and 131072 bytes, so a full `DEBUG DUMP` (about 25 KB, 2.3 s at 115200 baud) comes back whole. For `SYS RESET`, the helper extends its quiet-read timeout using `IPMC_BOOT_COUNTDOWN_BENCH` plus one second, so the boot countdown does not truncate the captured runtime banner. `-ComPort` selects the actual serial port (COM4 on this bench, default COM3 for compatibility).

12.1.1 Public demo package and engineering files

`demos/nucleo-f401re/package.ps1` builds and checks Tier-1 explicitly with `IPMC_TIER23=0`, even when the private extension is present. It exports the combined `ipmc-board-tier1.bin`, UART-loadable `ipmc-nucleo-f401re-tier1.hex`, a SHA-256 list and a build manifest to the demo's `images/` directory. The manifest identifies the source revision, dirty-worktree state, compiler, configuration hash and Tier. The packaging command does not program a board. Initial drive programming erases the saved FRU store; UART APP reload preserves it.

The root `README.md` links all public PDF documentation, the demo firmware, scripts, license choices and per-product/SKU pricing. The detailed board, portability, bootloader planning and bench documents are retained under ignored `local_only/Documentation`; public users do not need those files to build or reload the Tier-1 demo. Edit `local_only/vita_commmands.md` to maintain the full command catalog, then run `Documentation/build_brief.ps1` to regenerate the public command appendix and brief PDF. The generator does not overwrite the local catalog. Questions and licensing: support@ironkernel.com.

12.2 Changing a tunable

Edit the value in `crates/ipmc-defs/src/defs.rs` and rebuild. A new define is a `///` comment and one `pub const NAME: T = value;` item there, with the type its consumers need, and, when it has a range rule, a guard in `ipmc-defs/src/lib.rs`. A value fixed by a specification is not a define: it is a named constant in the owning module with its rule beside it.

12.3 Adding an IPMI command

1. Write the handler in `ipmc-core/src/cmd/<netfn>.rs` with the `Handler` signature; validate the request length first (C7h), write only data bytes, return `Err(code)` for refusals.
2. Add one row to `COMMANDS` in `commands.rs`.
3. Run `python -I tools/sync_ipmi_commands.py` so `Documentation/ipmi_commands.md` matches (`cargo test` fails until it does).
4. Add a table-driven test in `ipmc-core/tests/tier1_handlers.rs`; check the replay totals still read match=44 same_cc=61 cc_diff=115 not_implemented=89 (Tier-1 build; the commercial build has its own numbers in the private documents), or state why they change.

12.4 Adding an application port command

1. Write the handler in `ipmc-app/src/commands.rs` with the `Handler` signature (arguments, host, output): it writes its data lines (`KEY: value`) with `data` and returns `Ok`, or returns `Err(Fail::new("CODE", format_args!("sentence")))`; the engine writes the final `OK` or `ERROR <CODE>: <sentence>` line and takes back the data lines of a failed handler.

2. Add one row to `COMMANDS`: the syntax string (keywords in capitals, then `[NAME]` required, `{NAME}` optional, `{NAME...}` the rest of the line, last), one sentence of help and the handler. `HELP` lists the row at once, aligned; the test `the_command_table_is_well_formed` rejects a malformed syntax, a required parameter after an optional one, a rest parameter that is not last and two rows with the same keywords.
3. A value the host does not offer yet is a new method of `AppHost` (`host.rs`), implemented in `LinkHost` (`ipmc-runtime` `app_port.rs`) and in `FakeHost` (`tests.rs`); the compiler lists the places that miss it.
4. Add one row to `REQUEST_TABLE` in `tests.rs` (the request line and the lines the fake host must answer), add the help row to `HELP_TABLE`, and run `cargo test -p ipmc-app`.
5. Run `cargo clippy --workspace --all-targets -- -D warnings` and `python -I tools/dupcheck.py` (a handler that repeats another handler's code fails it), and add the row to section 7.6.

12.5 Adding a sensor part

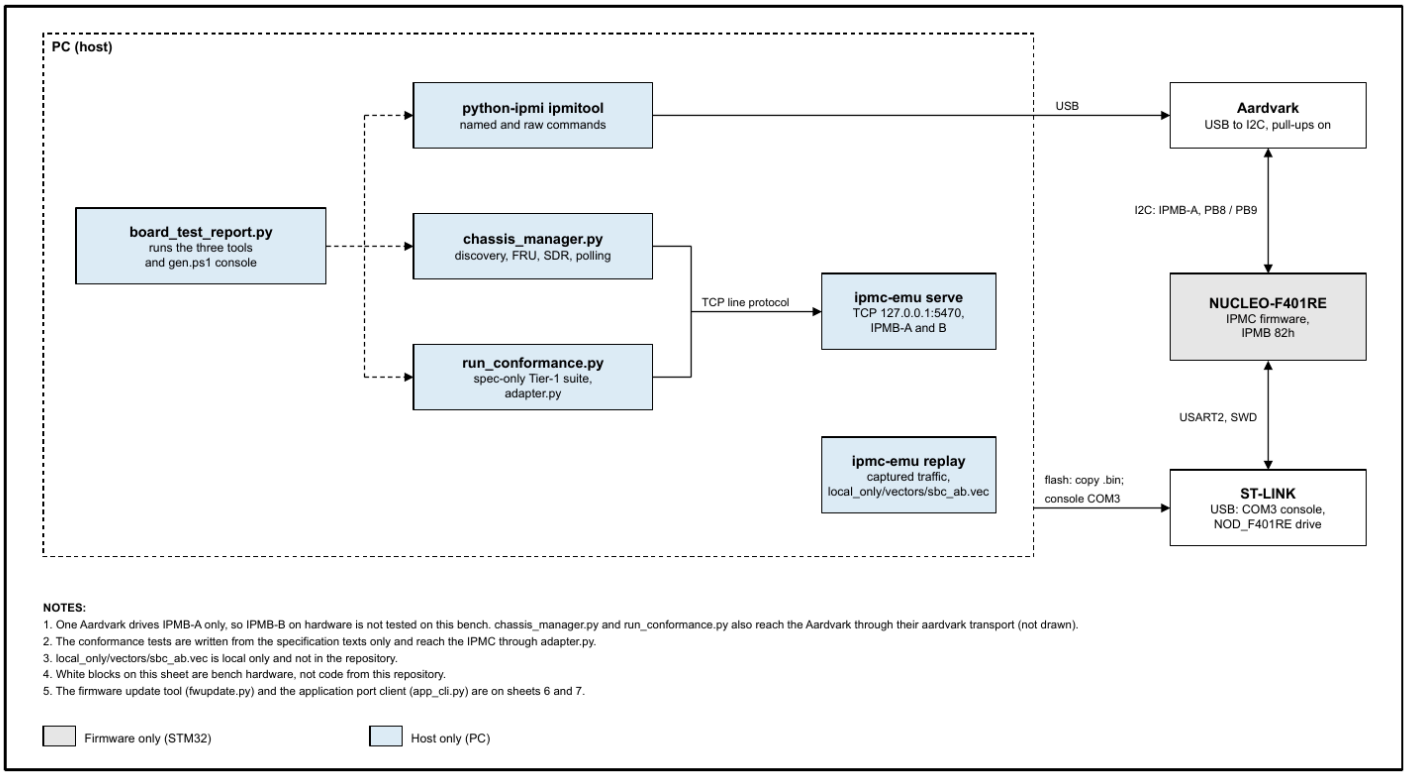
Write `drivers/<bus>/<part>.rs` implementing the matching `hal` trait against the embedded-hal 1.0 bus trait, test it against the crate's mock bus, hold it in the router in `routing.rs`, and select it in `defs.rs` with `Source::I2c` or `Source::Spi`. The firmware build refuses those values until a bus that is not an IPMB bus is assigned (`sensing.rs` guard).

12.6 Gates before handing work back

`cargo fmt`, `cargo clippy -D warnings` and `cargo test` on the workspace; firmware `gen.ps1 build`, `cargo clippy --release -- -D warnings` and `cargo fmt -- --check` inside `crates/ipmc-nucleo-f401re`; `sync_ipmi_commands.py --check`; `captured_sdr.py --check`; `dupcheck.py`; the name check against the local banned-name list; and for link or handler changes the replay totals above. With the extension folder present, a tenth gate repeats `cargo fmt`, `cargo clippy` and `cargo test` with `IPMC_TIER23=0` (the public Tier-1 build), runs the replay totals with `IPMC_TIER23=0` (they belong to the Tier-1 build), checks the extension's library source with `rustfmt --check --edition 2021`, adds the extension folder as a second path to the name check, and lints the evaluation code with `IPMC_EVAL=1` in the firmware clippy gate; `dupcheck.py` scans both repositories by itself.

13. Verification

Figure 13-1. Build, test and bench setup (ipmc-software, sheet 4)



Source: `Documentation/diagrams/src/ipmc-software.js`, sheet 4.

13.1 Host and tool tests (11-13-2026)

`Documentation/ipmc_test_coverage.md` lists every test case with its result; this section names what each crate covers.

The current run uses rustc 1.99.1 (58cf8f9c3, 2026-10-13), the GNU Windows host and the repository lockfiles. Tier-1 has **800 Rust cases, all passed**; the extension build has **853 Rust cases, all passed**. The executed Python tool tests report **220 passed and 0 failed**. Both capture replay baselines pass. Hardware and serial-reload results are separate from these host results.

Crate	Covers
`ipmc-core`	codec, address decode, link layer (bus speed, restarted link, the failure lines of a transmission), requester timing, reset gate, Tier rules, FRU state machine including M5, Payload Mode sensor, speed byte per Tier, event enables, extension hook and the power-on check, duplicate cache, queues, byte ring, names, dispatcher, handlers (`tests/tier1_handlers.rs`), Tier-1 link rules, SDR walk, command document in step (`tests/commands_doc.rs`)

Crate	Covers
`ipmc-app`	line assembler (CR LF, long line, bad byte), every request of the protocol against the test host (`REQUEST_TABLE`), tags, every error code, `WRITE_DISABLED`, the `HELP` listing
`ipmc-log`	capture log (records, wrap, drop order, CSV and trace rows, dump cursor and job, header line, cut lines, trace tee, switch lines)
`ipmc-sensors`	the analog sensors (byte-exact Full records, conversions, thresholds, readings through a real `Context`), evaluation, power state machine, routing, monitor and the power-request path, payload reset pulse timing, pin claims, LED level, example drivers against mock buses
`ipmc-emu`	vector reader, captured frames decode and re-encode byte-identically, replay runs including the Event Receiver role, Cold and Warm Reset rebuild, the flash copy across a power cycle, serve protocol over TCP, command line
`ipmc-fru`	image build, decoder, `verify`, store, the unit field editor (every field, every refusal, a MAC address in a serial), the single-sector flash log including a power cut at every word, the erase of a full sector and the settings byte
`ipmc-menu`	line editor, matching, help, table validation
`ipmc-defs`	the Tier-2/3 detection rules (`tests/tier23.rs`) and the IPMB timing arithmetic (`i2c_timing.rs`); the guards of `lib.rs` run at compile time. `tools/test_defs.py` tests the Python reader of `defs.rs`
`ipmc-ext`	the default `Active` answers nothing for an unowned command; the bridge and `ipmc-defs` agree on the extension
`ipmc-boot`	image header (placeholder bytes, patch, every `check` failure, digest range), the decision table row by row, the countdown window, boot request encode and decode, region policy (`load_allowed` at every boundary, erase sectors), Intel HEX parser, `LOAD` (full image, repeated and overlapping records, unaligned head and tail, flash faults), session commands and answers, trace text, the whole boot sequence over a test board (`run/tests.rs`: trace bytes, countdown digits, `WAKEUP` choice on each port, stay, boot request read and clear, `RUN`, `RESET`, lost-byte watch), `fwimage`, and `tests/flash_map.rs`, which parses `flash_map.x` and fails on a difference from `geometry.rs`
`ipmc-wire`	checksum, builder, CRC-32, SHA-256 (FIPS 180-4 vectors, every length 0 to 300, split feeds), hex parser

The per-crate counts are in the `cargo test --workspace` output. The Chassis Manager request policy and the Event Receiver have 100 Python unit tests (`tools/chassis-manager/test_cm_link.py`); `tools/fwupdate/test_fwupdate.py` runs the update tool against a fake bootloader; `tools/capture/test_capture2pcap.py` checks the pcap blocks (with tshark, when `C:\Program Files\Wireshark\tshark.exe` exists) and `tools/app-port/test_app_cli.py` checks the bench client against a fake transport. Tests are table-driven; `tools/dupcheck.py` fails near-identical test pairs as well as production pairs.

13.2 Capture replay

`ipmc-emu replay` feeds the 309 captured requests of a live Chassis Manager and SBC IPMC (IPMB-A and IPMB-B on one time base). With IPMB-B fitted and enabled and the alternate policy the result is match 44, same CC 61, CC differ 115, not implemented 89, every request answered exactly once. The differences are designed: the capture's IPMC is an earlier-

revision controller that answers commands the 2022 specification makes not applicable to Tier 1 (C1h here), and the Device ID, FRU and SDR contents are this IPMC's own.

13.3 Executed requirement checks and Chassis Manager tests

Current validation, 11-13-2026:

Executed tests	PASS	FAIL
Tier-1 strict emulator requirement checks	103	0
Tier-1 physical IPMB-A requirement checks	98	0
Board report: environment	7	0
Board report: command-line commands and error probes	27	0
Board report: Chassis Manager	11	0
Board report: requirement checks and totals check	99	0
Board report: link and driver health	8	0
Board report: FRU persistence and restoration	8	0
Board report total executed checks	160	0

The board report includes its requirement checks; these counts overlap and must not be added as independent hardware tests. This section lists executed checks. The tested physical setup is NUCLEO-F401RE with one Aardvark on IPMB-A and the onboard ST-LINK console on COM4. The strict emulator uses both simulated buses, `--policy alternate --b-power-up strict`; the capture cross-check reports zero unexpected differences. Host replay baselines are in section 13.2.

```
$env:IPMC_TIER23 = '0'
cargo run -p ipmc-emu -- serve --port 49183 --b-fitted 1 `
  --policy alternate --b-power-up strict
# In another terminal:
tools/python-ipmi/.venv/Scripts/python.exe tools/conformance/run_conformance.py `
  --transport emu --port 49183 --report tools/conformance/conformance_emu.md
```

The public PASS/FAIL results are generated from executed checks. Full raw reports remain local for engineering traceability. The default NonStrict bench policy uses arrival-bus responses; the strict dual-bus requirement checks use the explicit alternate/strict configuration above.

13.4 Serial reload and persistence acceptance

The public Tier-1 image was programmed through ST-LINK, exercised over IPMB-A, then reloaded through the console CLI in 38.1 seconds. The bootloader verified the application SHA-256 and returned `OK RUN`. Subsequent boot confirmed Tier-1 identity, self-test `55h 00h` and the expected image length/digest. Decoded FRU Board/Product fields remained unchanged across reload. The physical board report was then rerun, including write/reset/persistence/restoration checks, with 160 PASS and 0 FAIL. See section 16.5 for the exact procedure and section 16.7 for measured values.

The public validation record is `demos/nucleo-f401re/validation.md`; the root README links firmware downloads, operator instructions, license/pricing and the public PDFs.

14. Example Sequences

14.1 Bring-up with a stored FRU image

1. Reset; the bootloader starts the runtime (section 8.3); `reset_cause::take` reads RCC_CSR; the clock reaches 84 MHz.
2. Pins claimed, console up, application port up, IPMB pins parked, strap pins read: GA forced to 01h, IPMB 82h.
3. `FruLog::boot`: the store region is checked against sector 2 and the sector is scanned; the valid record with the highest sequence number is loaded, and nothing is erased.
4. I2C1 slave armed at 7-bit address 41h (OAR1 = 4082h), general call on; I2C3 stays off.
5. `Context::new` builds the image and SDRs, `set_fru` installs the stored image and the self test runs on it: banner `Self test: 55h 00h (passed)`.
6. First sensor sample: M0 to M1 (initial checks pass); one step makes one transition, so the next sample, one period later, takes M1 to M4 once PGOOD is seen.

14.2 Get Sensor Reading of sensor 1

1. Request on IPMB-A: `82 10 6E 20 14 2D 01 9E` (NetFn 04h, sequence 05h, sensor 1).
2. The EV interrupt posts the frame; `on_frame` accepts it; `poll` misses the cache.
3. `run_request`: LUN 0, NetFn 04h; `commands::find` gives `get_sensor_reading`.
4. `Context::read_sensor(1, &link.buses, out)` encodes the link view: `08 40 02`.
5. Response `20 14 CC 82 14 2D 00 08 40 02 F3` queued, `next_tx` picks IPMB-A (the only enabled bus), the bit-bang master writes it, `on_tx_result(A, true)` pops it.

14.3 Write FRU Data, reset, read back

1. `raw 0x0a 0x12 0x00 0x35 0x00 0x32` writes one serial byte (response `00 01`); the self test reads 57h until step 2.
2. A second write corrects the board area checksum; the self test returns to 55h 00h.
3. After `IPMC_FRU_PERSIST_DELAY_MS` without another write the save appends a record (`FRU log: saved record N at offset M`).
4. `SYS RESET`; boot loads the new record; Read FRU Data returns the written bytes.
5. `FRU RESTORE` saves the built image as a new record; after a reset the original serial returns.

14.4 Failover in the emulator

1. `ipmc-emu serve --b-fitted 1`; the Chassis Manager enables IPMB-B with Set IPMB State.
2. The client sends `#bus A down`; the next response handed out on IPMB-A fails (`on_tx_result(A, false)`), moves to IPMB-B and is delivered there.
3. Requests the Chassis Manager sends on IPMB-A are lost; it retries on IPMB-B with the same sequence number; a retry of an answered request is replayed from the cache.

14.5 Capture a failure and open it in Wireshark

1. The log records from start-up (`IPMC_LOG_AT_BOOT` true); `DEBUG LOG ON` starts it otherwise, and `DEBUG CLEAR` empties it before a run.
2. Reproduce the problem.
3. Start a terminal log and type `DEBUG DUMP`; the console prints the header line, the CSV rows and `# end of capture`. A key press stops a dump early.
4. `python -I tools/capture/capture2pcap.py dump.txt -o dump --csv` writes `dump.pcap` and `dump.csv`.
5. Open `dump.pcap` in Wireshark (link type 260). A frame that did not leave the board shows as the frame followed by a text packet `tx failed: addr-nack`, naming the cause.

14.6 Write the unit fields at the first boot

1. The payload CPU sends `INFO` and reads `PROTOCOL: 2` and the final `OK`, then `FRU FIELDS` for the longest value of each field.
2. It sends `FRU SET BOARD_SERIAL 00:11:22:33:44:55`, `FRU SET PRODUCT_SERIAL ...`, `FRU SET BOARD_PART ...`, `FRU SET PRODUCT_PART ...`, `FRU SET ASSET_TAG ...` and `FRU SET MFG_DATE 2026-10-25 14:30` (or any other of the thirteen fields); each answer's data line echoes the value as stored, and the final line is `OK`.
3. After `IPMC_FRU_PERSIST_DELAY_MS` without another write, the FRU log saves the image (`FRU log: saved record N at offset M`).
4. The payload CPU repeats `FRU SAVE` until it reads `SAVE: done` and may then remove power.
5. A Chassis Manager's Read FRU Data returns the new values and the self test still reads 55h 00h.

15. Bootloader

The bootloader is the program in flash sectors 0 and 1. It runs first after every reset, checks the runtime firmware and either starts it, counts down for a `WAKEUP`, or stays and serves a command session on one of the two UARTs (PLAN decisions 31 and 33 to 37). Every decision is the host-tested `ipmc-boot` crate (`crates/ipmc-boot`, `no_std`, `forbid(unsafe_code)`); the binary `ipmc-boot` (`crates/ipmc-nucleo-f401re/src/boot/`) is the glue that reads the pins, moves bytes between the UART rings and the session, programs the flash and performs the jump. The current bootloader BIN is 23,360 bytes (`.text` 23,320 B and `.data` 40 B), measured 11-13-2026, of the 32,768 bytes of sectors 0 and 1; `local_only/Documentation/ipmc_boot_plan.md` is the design of record and gives the open questions and their answers. Section 16 covers the image header, the digest and the update procedure.

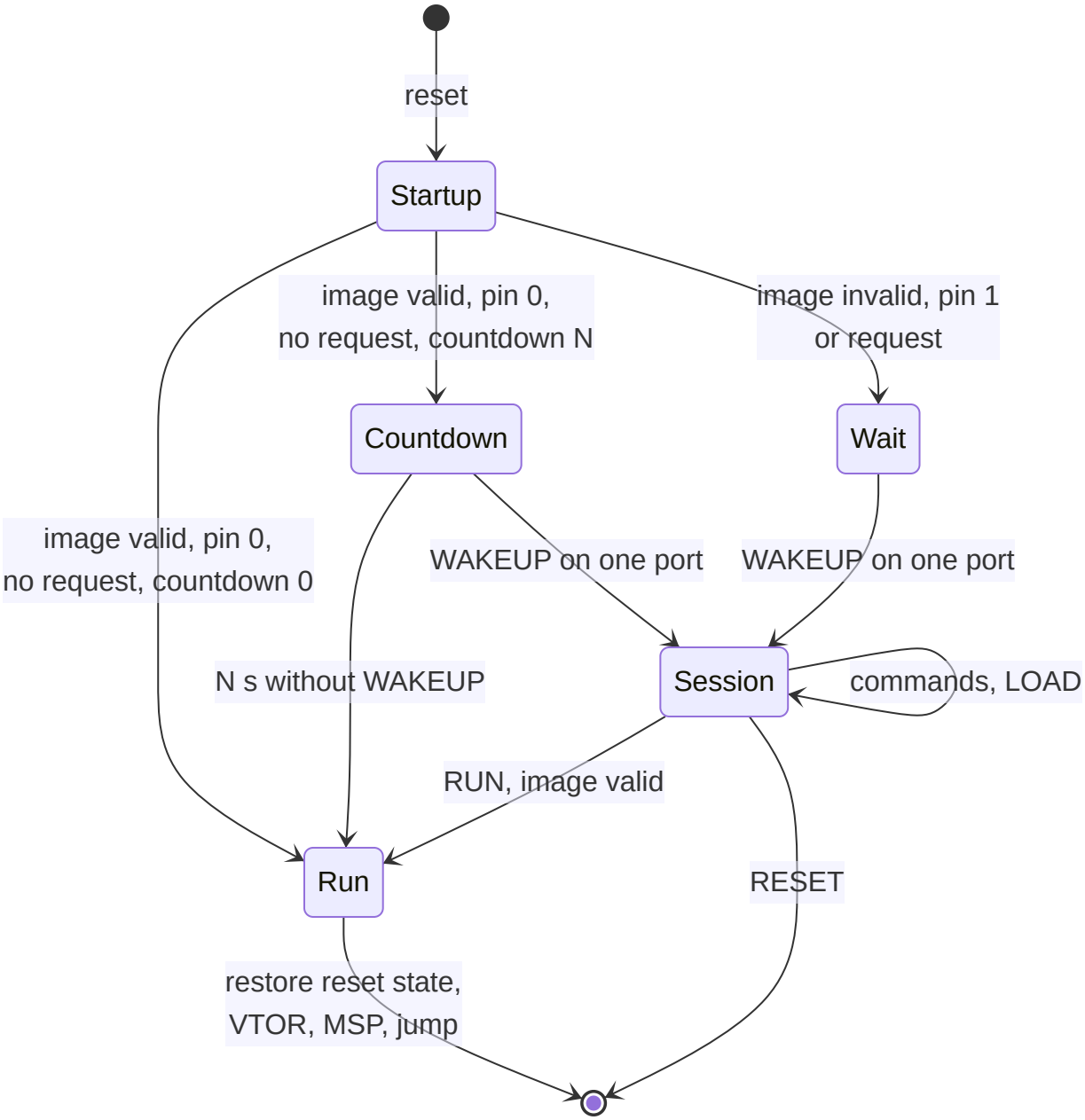
15.1 Modules and main functions of `ipmc-boot`

File	Main functions and types	What they do
`src/geometry.rs`	`SECTORS`, `Sector`, `Target`, `Target::sectors`, `load_allowed`	the eight sectors of RM0368 Table 5, the BOOT, STORE and APP address constants, the sectors an erase target names in erase order (FRU is sector 2; APP is sectors 3 to 7, sector 3 first), and the one range rule of `LOAD`: every byte inside APP, overflow-safe, length 0 refused; compile-time assertions tie the table and the STORE sector to `ipmc-fru`
`src/header.rs`	`placeholder`, `decode`, `check`, `digest`, `patch`, `version_bytes`, `ImageSource`, `SliceImage`, `Report`, `ImageError`, `HeaderField`, `PatchError`	the 64-byte runtime image header: the const bytes the runtime links, the decoder, the one image check (header, vector table, SHA-256), the digest over the hashed range, and the build-time patch that writes length and digest (section 16.1)
`src/request.rs`	`encode`, `decode`, `Interface`, `MAGIC`, `CHECK_MAGIC`	the four boot request words that cross a software reset (section 15.4)
`src/decision.rs`	`decide`, `Decision`, `Reason`, `Countdown`, `Countdown::step`, `Tick`, `PortId`	the start decision as a pure function of the pin, the request and the image check, and the countdown window as a state machine over elapsed milliseconds
`src/trace.rs`	`banner`, `reset_cause`, `pin`, `request`, `runtime_image`, `image_check`, `image_check_full`, `stored_digest`, `decision`, `countdown`, `starting`, `bootloader`, `build`, `port`, `port_errors`, `erased`, `clock_fault`, `digest_or_none`	one writer per line of the start-up trace and of `INFO`; the trace and `INFO` call the same writers, so each text exists once

File	Main functions and types	What they do
`src/text.rs`	`TextOut`, `put_dec`, `put_hex`, `put_hex_bytes`, `put_addr`, `put_version`, `CRLF`	numbers as text without `core::fmt`, so the image fits sectors 0 and 1
`src/hex.rs`	`parse`, `Record`, `DATA_MAX`	the Intel HEX record parser: form, checksum, type and count, 64 KB wrap; types 00, 01, 04 and 05
`src/error.rs`	`BootError`, `BootError::code`, `put_err`, `put_err_head`, `put_err_word`	every `ERR` code of the session in one enum with the one answer-line writer (section 15.5)
`src/load.rs`	`FlashOps`, `erase`, `Loader`, `Loader::record`, `LoadStep`	the flash operations the session needs (`erase_sector`, `program`), the erase of a target's sectors and the `LOAD` state machine: range check, erased check, program, read back and compare
`src/commands.rs`	`COMMANDS`, `Cmd`, `Row`, `find`, `words`, `put_help`	the flat command table, the case-insensitive lookup and the help table
`src/run.rs`	`BootBoard`, `BootParts`, `boot`; private `sequence`, `countdown`, `Editors::wakeup`, `serve`, `session_pass`, `Hw<B>` (`BootHw`)	the generic boot sequence of section 15.2 over the `ipmc-hal` handles of a board: trace, strap pins and settle wait, boot request read and clear, timed image check, decision, countdown or stay, the session loop with its lost-byte watch; generics only, the `dyn` types are `TextOut` and `BootHw`
`src/session.rs`	`Session`, `Session::open`, `line`, `line_too_long`, `line_damaged`, `poll`, `echo`, `BootHw`, `StartRecord`, `Action`, `is_wakeup`	the session state machine: one complete line in, the answer text out, `Run` or `Reset` returned to the caller after it flushed the port
`src/bin/fwimage.rs`	`patch`, `check`, `combine`, `size`	the host tool of the build (section 16.2)

15.2 Start-up sequence and decision

Source: `ipmc_boot::run` (`sequence`), `src/boot/main.rs`, `ipmc_boot::decision`, `ipmc_boot::trace`.



The order of `main`:

1. `reset_cause::read` decodes RCC_CSR without clearing it, so the runtime reads and reports the cause itself; `clock::init` brings the clock to 84 MHz (a clock fault is reported on a `Clock:` line); `ports::init`, the LED and the console UART (USART2, `IPMC_CONSOLE_BAUD`) start. A pin claim conflict prints both owners and parks, a console that does not start blinks LD2 and parks.
2. The first trace lines: banner (`ipmc-boot v<x.y.z> (sectors 0-1)`, with `, field` in a field build) and `Reset cause:`.
3. The DWT cycle counter starts (it times the image check), `ga::init` claims the strap pins and the BOOTLOADER pin (PC10, input with pull-down), the code waits `IPMC_FWU_PIN_SETTLE_US` (100 us) and reads the pin once (`BOOTLOADER pin: 0` or `1 (forced)`).
4. The boot request words at `__boot_request` are read and the check word is cleared; the request counts only after a software reset (`Request:`).
5. `header::check` runs on the flash, timed with the cycle counter (`Runtime image:`, `Image check:` with the first 16 digest digits, `match` or `MISMATCH`, and the time).

6. `decide` returns the decision (`Decision:`).
7. Start: the `Starting the runtime firmware at 0x0800_C000` line and `start_runtime` (section 15.6). Countdown: section 15.3. Stay: SysTick and the LD2 wait blink start, APP_UART starts when `IPMC_APP_UART_ENABLE` is true, and both ports wait for `WAKEUP` without a time limit.

The decision table (`decide`, tested row by row):

BOOTLOADER pin	Request	Image check	Decision	Reason text
0	none	valid, countdown 0	start the runtime at once; no UART listens	
0	none	valid, countdown N	countdown	
1	any	any	stay	`BOOTLOADER pin`
0	yes	any	stay	`requested`
0	none	missing	stay	`no image`
0	none	bad magic, bad header or bad vector table	stay	`bad magic`, `bad header`, `bad vector table`
0	none	digest mismatch	stay	`digest mismatch`

The pin row comes before the request row, so the trace names the pin when both apply. A stay never counts down. The start-up trace goes to the console only; a valid image with no countdown leaves APP_UART silent, so a payload processor on that port sees no text at a reset. The lines, for a valid image and a normal power-on:

```
ipmc-boot v0.1.0 (sectors 0-1)
Reset cause:    power-on
BOOTLOADER pin: 0
Request:        none
Runtime image:  header at 0x0800_C200: found, v0.1.0, 138524 bytes, flags 00h
Image check:    SHA-256 1f3a09c27b54e6d8 match, 171 ms
Decision:       starting the runtime
Starting the runtime firmware at 0x0800_C000
```

`Runtime image:` reads `missing`, `bad magic`, `bad header (<field>)` (`format`, `size`, `vector address` or `length`) or `bad vector table` when the check stops before the digest, and `Image check:` then reads `not run`. A stay prints `Decision:  staying in the bootloader: <reason>, waiting for WAKEUP on CLI or APP` (`CLI` alone with `IPMC_APP_UART_ENABLE` false). The UART names come from `IPMC_CONSOLE_UART_NAME` and `IPMC_APP_UART_NAME`; a PC port number is never printed.

15.3 Boot countdown

`IPMC_BOOT_COUNTDOWN` is 3 s in the bench image (`IPMC_BOOT_COUNTDOWN_BENCH`) and 0 in a field image (`gen.ps1 build -Field`, `IPMC_BOOT_COUNTDOWN_FIELD`; `cfg(ipmc_field)`, a build guard limits it to 0 to 30). With 0 the valid image starts at once. With N above 0 and a valid image, no pin and no request (`ipmc_boot::run::countdown`, `Countdown::step`):

1. SysTick and APP_UART start; both ports print `Send WAKEUP on CLI or APP within N s to stay in the bootloader`, CR LF, then the digit N.
2. Each following second adds a space and the next lower digit on the same line (`3 2 1`) on both ports; the window is N seconds from the first digit.
3. Both ports listen with one `LineEditor` each, without echo; only a complete line whose one token is `WAKEUP` in any letter case counts, and every other line gets no answer. The first such line (the console is read first within a pass) ends the countdown with CR LF on both ports, answers `OK` on its port, opens the session there and stops the other UART; the console prints `Decision: staying in the bootloader: WAKEUP in countdown`.
4. After N seconds without `WAKEUP` the digits end with CR LF and the start path runs. A `WAKEUP` whose line end arrives after the window is not answered.

15.4 Boot request

The runtime asks for the bootloader with four words at `__boot_request`, the start of the NOINIT RAM region (`bsp/boot_request.rs` : `read` , `write` , `clear` ; the meaning is `ipmc_boot::request`):

Word	Content
0	`MAGIC` B007_5EC7h
1	the requesting interface: 1 console (`DEBUG BOOTLOADER`), 2 application port (`BOOTLOADER`)
2	reserved, 0
3	check word: words 0 to 2 XOR `CHECK_MAGIC` 5A17_B007h (`ipmc_wire::check_word`)

The runtime writes the words just before the processor restarts (`IpmbLink::request_bootloader` asks for a Cold Reset on the one reset path, section 8.2). The bootloader reads them, clears the check word at once, and accepts them only when the reset was a software reset and the check word matches (`request::decode`), so a stale RAM pattern after a power-on is refused. The trace names the interface: `bootloader requested by the runtime firmware (console)` or `(app port)` .

15.5 Session

The session is the `Session` state machine over the trait `BootHw` (the flash through `FlashOps` , `now_ms` , `activity` , `port_counts`). `ipmc_boot::run` (`serve` , `session_pass`) moves bytes between the UART rings, the `LineEditor` of `ipmc-menu` and the session, and performs the `Run` and `Reset` the session returns. A line whose one token is `WAKEUP` chooses its port (section 15.3); the other USART is stopped (`uart::stop` : interrupt masked, RCC reset, clock off, pins back to inputs without a pull) and stays silent until the next reset. The session has no idle timeout; `RUN` , `RESET` and a power cycle leave it.

In command mode the typed characters are echoed (a backspace as BS SP BS, the line end as CR LF), the answer follows, then the prompt `IPMC_BOOT_PROMPT` (`BOOT>`). Every command ends with one status line that starts with `OK` or `ERR` . Keywords are case-insensitive and exact; extra or wrong words give `ERR ARGS <command>` , an unknown word `ERR UNKNOWN <word>` .

Command	Answer
`WAKEUP`	`OK`
`HELP` , `?`	the command table, `OK`

Command	Answer
`INFO`	the start-up record of this reset (banner, reset cause, pin, request, runtime image, `Image check:` with all 64 digest digits and the time of a fresh check, `Stored digest:`, decision), then `Bootloader:`, `Build:` (`bench` or `field`), `Port:` (name, baud, `8N1 RTS/CTS on` or `off`), `Port errors:` (overrun, line, rx overflow, tx timeout) and `Erased:` (`none`, `FRU`, `APP`, `FRU APP`), `OK`
`ERASE APP`, `ERASE FRU`	`Erase sectors 3-7 (runtime firmware)? Type CONFIRM` or `Erase sector 2 (FRU flash store)? Type CONFIRM`, then `OK PENDING`
`CONFIRM`	runs the pending erase: `OK ERASED <APP or FRU> <ms> ms`, `ERR FLASH <fault>` or `ERR NVMRO`; with nothing pending `ERR NOTHING_PENDING`
`LOAD`	`OK LOAD APP: send records` and record mode; without a completed `ERASE APP` in this session `ERR NOT_ERASED erase APP first`
`VERIFY`	`OK VERIFY valid SHA-256 <64 hex> <ms> ms`, or `ERR IMAGE <reason> computed <64 hex or none> stored <64 hex or none>`
`RUN`	image check; valid: `OK RUN`, flush, the start sequence; else `ERR IMAGE <reason>`
`RESET`	`OK RESET`, flush, SYSRESETREQ

After `OK PENDING` the next line must be `CONFIRM`; any other line, an empty one included, gives `ERR CANCELLED` and is not run. `ERASE APP` erases sector 3 first (it holds the vector table and the header) and then sectors 4 to 7, so a power cut leaves an image that fails its check. An erase or a program stalls the core; the host sends nothing until the answer. A failed erase clears the "erased" mark of its target.

Record mode (`Loader::record`): nothing is echoed and no prompt is printed between records. Each line that starts with `:` is one Intel HEX record and gets `OK` or an `ERR` line, and the loader stays in record mode after an error, so the host resends. A data record is checked in this order: form, checksum, type and count, 64 KB wrap, the range (`load_allowed`: every byte inside APP), the erased check (every target byte reads FFh), the program, and a read back that must equal the record. An accepted record retriggers the LED activity blink; record 04 sets the upper address bits, 05 is accepted and ignored. Record 01 runs the image check and answers `OK END <n> records <b> bytes, image valid` or `ERR IMAGE <reason>`; the prompt returns. A line that does not start with `:` ends record mode with `ERR ABORTED` and is not run; an empty line is ignored; a line that is too long gives `ERR LINE_TOO_LONG`; a line in which the port lost or damaged a byte (UART overrun, RX ring overflow or line error, counted since the previous byte) is dropped with `ERR LINE_DAMAGED bytes lost or damaged on the port; the line was dropped` and cancels a pending erase (record mode stays); no line for `IPMC_FWU_LOAD_IDLE_S` (30) seconds gives `ERR TIMEOUT`.

The error codes (`BootError`, one enum for every answer):

Code	Cause
`FORMAT`	a record that is not `:` and an even number of hex digits of at least 10, or a count that does not match its type
`CHECKSUM`	the record's zero checksum fails
`TYPE`	a record type other than 00, 01, 04, 05

Code	Cause
`WRAP`	a data record that crosses a 64 KB segment
`RANGE`	a data record outside APP (sectors 3 to 7)
`NOT_ERASED`	a target byte is not FFh (a repeated or overlapping record), or `erase APP first` for `LOAD` without an erase
`VERIFY`	the programmed bytes do not read back as sent
`FLASH <fault>`	the flash controller reported a fault; the fault name follows
`NVMRO`	NVMRO is asserted: the register layer refuses every erase and program
`LINE_TOO_LONG`, `LINE_DAMAGED`	a line over the editor limit; a line with lost or damaged bytes
`ABORTED`, `TIMEOUT`	record mode left by a line that does not start with `:`; no line for `IPMC_FWU_LOAD_IDLE_S`
`CANCELLED`, `NOTHING_PENDING`	a line other than `CONFIRM` after `OK PENDING`; `CONFIRM` with no erase armed
`IMAGE <reason>`	the image check failed: `missing`, `bad magic`, `bad header`, `bad vector table`, `digest mismatch`
`UNKNOWN`, `ARGS`	a word that is no command; a known command with other words than its row

While the session runs the IPMC holds neither IPMB bus (no service, no answer from 82h) and does not drive PWR_EN or PAYLOAD_RST_OUT_N; LD2 blinks at `IPMC_FWU_LED_WAIT_CHZ` in command mode and shows the activity pattern while records load.

15.6 Start sequence into the runtime

`src/boot/start.rs` `start_runtime` is the last code the bootloader runs, for every start (after the checks, after the countdown, after `RUN`). It leaves the processor as a reset leaves it for every part the bootloader touched, so the runtime start-up of section 8.3 runs unchanged:

1. flush the console, and APP_UART when it is up;
2. mask interrupts and stop SysTick;
3. pulse a reset through RCC on USART1, USART2, PWR and GPIOA to GPIOC, put their clock enables back to the reset values, then clear every NVIC enable and pending bit;
4. `clock::restore_reset_state` returns the clock tree to the reset state (a fault there does not stop the start; the runtime reports a PLL it finds running as its clock fault);
5. stop the DWT cycle counter;
6. write VTOR with `__app_vectors` (0x0800_C000);
7. load MSP from word 0 of the table, clear CONTROL, unmask interrupts and branch to word 1, the reset vector (the one `asm!` of the binary).

The image check has accepted the table before: word 0 is an 8-byte aligned address in RAM above NOINIT and word 1 an odd address inside the image.

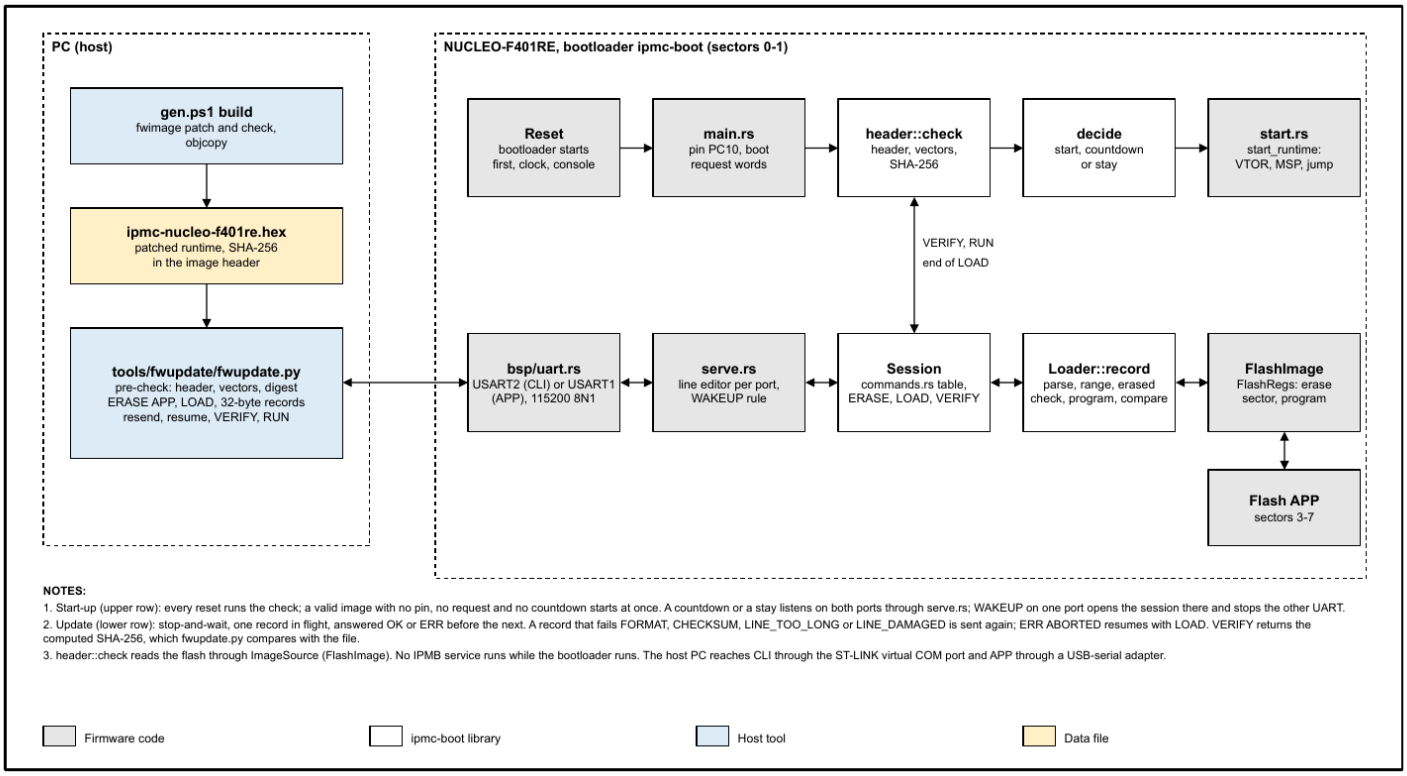
15.7 Firmware crate glue (`src/boot/`)

File	Main functions	What they do
`main.rs`	`main`	the reset cause, the clock, the pin owner, the LED and the console UART, then the `BootParts` and `F401Boot` handed to `ipmc_boot::run::boot`, which runs the start-up order of section 15.2 and the dispatch to start, countdown or session
`board.rs`	`F401Boot` (`BootBoard`)	`init_straps` (cycle counter, strap and BOOTLOADER pins), `start_ports` (SysTick, APP_UART), `stop_port`, `start_runtime`, `system_reset`, `wait_for_interrupt`
`start.rs`	`start_runtime`	the start sequence of section 15.6 (the flush of both ports is done by `ipmc_boot::run` before it)
`flash_image.rs`	`FlashImage` (`ImageSource::read`, `FlashOps::erase_sector`, `program`)	the flash over `FlashRegs`: an unaligned head and tail by bytes, the aligned middle by words in batches of 64

The bootloader uses these parts of the library `src/bsp/`: `clock` (`init`, `restore_reset_state`), `reset_cause` (`read`, `system_reset`), `uart` (the one driver of both ports), `led`, `systick`, `ga` (strap pins and the BOOTLOADER pin), `boot_request`, `cycles` and `flash_regs`. It never touches the IPMB pins, PWR_EN, PAYLOAD_RST_OUT_N, the ADC, I2C1, I2C3 or the FRU store.

16. Firmware Update and Image Integrity

Figure 16-1. Firmware update data path and start-up image check (ipmc-software, sheet 6)



Source: `Documentation/diagrams/src/ipmc-software.js`, sheet 6.

The runtime firmware in sectors 3 to 7 carries a header with its length and SHA-256 digest. The bootloader checks it at every reset and before every start, and a host tool replaces the runtime over a UART with a stop-and-wait Intel HEX load. Nothing in the update path can write sectors 0 and 1.

16.1 Image header

At `0x0800_C200`, 64 bytes, little-endian, one fixed address for every build (`ipmc_boot::header`). The runtime vector table is 101 words and ARMv7-M aligns it to 512 bytes, so it occupies `0x0800_C000..0x0800_C193` and the header starts at the first address after the aligned table.

Offset	Size	Field	Value and check
0x00	4	magic	0x434D_5049, the bytes `IPMC` in memory order; FFFF_FFFFh is an erased flash (`missing`), any other value is `bad magic`
0x04	2	header format	1

Offset	Size	Field	Value and check
0x06	2	header size	64
0x08	4	image length	bytes from `0x0800_C000`; a multiple of 4; at least 0x240 (vector table area and header) and at most 475 136 (`APP_LEN_MAX`); FFFF_FFFFh in the linked placeholder
0x0C	4	firmware version	major, minor, patch, 0 of the runtime crate version (`version_bytes` of `CARGO_PKG_VERSION`)
0x10	4	build flags	bit 0 Tier-2/3 extension built in (`FLAG_TIER23`), bit 1 evaluation build (`FLAG_EVAL`); other bits 0
0x14	4	vector table address	0x0800_C000
0x18	8	reserved	0
0x20	32	SHA-256 digest	of the hashed range of section 16.2; FFh in the linked placeholder

The runtime links the header as one `#[used]` static in the section `.app_header` (`src/app_header.rs`), built by the const function `header::placeholder(version, flags)`; `link.x` asserts its address and size. The offsets, the magic and the size exist once, in `ipmc_boot::header`, and the runtime static, the `fwimage` tool and the bootloader all use them. The runtime reads the header back for the `Image:` banner line (`app_header::read` gives the bytes, `ipmc-runtime` `console.rs` prints version, length, first 16 digest digits).

16.2 Digest and build pipeline

The digest is SHA-256 (FIPS 180-4), computed in software by `ipmc_wire::Sha256` (`new`, `update`, `finish`; the STM32F401 has no hash peripheral): a rolled 64-round loop over a 16-word circular message schedule, about 1.2 us per byte at 84 MHz. It proves the bytes in sectors 3 to 7 are the bytes the build hashed; it is not a signature and does not prove who built them.

Hashed range: `APP_START` to `APP_START + length` without the 32 digest bytes, that is `0x0800_C000..0x0800_C21F` and then `0x0800_C240..(0x0800_C000 + length)`. Every other header field is inside the hash, so a changed length, version or flag is a mismatch. The bootloader hashes the length the header states, not the whole region, so the check time follows the image size; the image is hashed with `--gap-fill 0xFF`, and a `LOAD` never writes a gap, so the gaps read FFh in flash as in the hash.

`fwimage` (`crates/ipmc-boot/src/bin/fwimage.rs`, exit 0 ok, 1 refused with one reason line, 2 usage):

Subcommand	Does
`fwimage patch <app.img> <header.bin>`	pads the image with FFh to a multiple of 4 bytes, checks the placeholder at offset 0x200 and the vector table words, writes the length and then the digest into the image file in place, and writes the 64 patched header bytes to `header.bin`
`fwimage check <app.img>`	runs `header::check`, the bootloader's own image check, on an image

Subcommand	Does
`fwimage combine <boot.bin> <app.img> <board.bin>`	writes the board image: the bootloader at offset 0, FFh up to `0xC000`, the runtime from `0xC000`
`fwimage size <boot.bin>`	fails above 32 768 bytes and warns above 24 576

`gen.ps1 build` (section 12.1) runs them in this order: build the bootloader and `fwimage size`; build the runtime and cut `ipmc-nucleo-f401re.app.img` (the bytes from `0x0800_C000`); `fwimage patch`; `objcopy --update-section .app_header=app_header.bin` gives the patched ELF; the image is cut again from the patched ELF and `fwimage check` runs on it, and a mismatch stops the build; `objcopy -O ihex` of the patched ELF gives `ipmc-nucleo-f401re.hex`, the file for `LOAD`; `fwimage combine` writes `ipmc-board.bin`. The images, their writers and what each erases are in section 9.1.

16.3 The image check and when it runs

`header::check` is the one function, run by the bootloader at every reset (step 5 of section 15.2), by `VERIFY`, `RUN`, `INFO` and the end of `LOAD` (record 01), and by `fwimage check` at build time. The steps, in order, stop at the first failure:

1. Magic: erased flash gives `missing`, another value `bad magic`.
2. Format 1, size 64, vector table address 0x0800_C000, length in range and a multiple of 4; else `bad header` with the field named.
3. Vector table: word 0 (initial stack pointer) 8-byte aligned and inside `0x2000_0100..=0x2001_8000` (RAM above NOINIT); word 1 (reset vector) odd (Thumb) and, with bit 0 cleared, after the header and inside the image; else `bad vector table`.
4. SHA-256 of the hashed range equal to the stored digest; else `digest mismatch`.

A failed check at a reset keeps the bootloader and waits for `WAKEUP`; a failed `RUN` answers `ERR IMAGE <reason>` and starts nothing.

16.4 Entry paths

Path	How	Result
Console command	`DEBUG BOOTLOADER {CONFIRM}`	with payload power off, or with `CONFIRM` (any letter case): `Restarting into the bootloader` (and, with power on, `Payload power drops at the reset`), then the request and the reset path. With power on and no `CONFIRM`: `Payload power is on and drops at the reset into the bootloader` and `Send DEBUG BOOTLOADER CONFIRM to restart`, no restart
Application port	`BOOTLOADER {CONFIRM}`	`BOOTLOADER: restarting` then `OK`; with power on and no `CONFIRM`: `ERROR CONFIRM: payload power is on and drops at the reset; send BOOTLOADER CONFIRM`, no restart

Path	How	Result
Countdown	`WAKEUP` on CLI or APP inside `IPMC_BOOT_COUNTDOWN` seconds	session on that port (bench image: 3 s; field image: no countdown)
BOOTLOADER pin	PC10 at logic 1 at reset	the bootloader stays whatever the image check says; the field entry for a runtime that is valid but does not work
Invalid image	the check fails at a reset	the bootloader stays and waits for `WAKEUP`

The two commands keep no state: the confirmation is the argument of the second command. "Payload power is on" is the PWR_EN test (`Status::payload_power` , the console's `Sensing::payload_power_on`); the reset releases PWR_EN on the NUCLEO-F401RE, so the plain command only warns. Both commands call `IpmbLink::request_bootloader(interface)` , which asks for a Cold Reset on the one reset path (`service_reset`): the answer has left the port, an accepted FRU write is saved, or `IPMC_RESET_WAIT_MS` has passed. Just before the processor restarts the four request words are written (`boot_request::write(request::encode(interface))`) and the bootloader accepts them after the software reset (section 15.4).

16.5 Update procedure

16.5.1 Public Tier-1: reload through the console CLI

This feature, the bootloader and `tools/fwupdate/` are included in the public Tier-1 software. No private Tier-2/3 source or commercial extension is required to use the CLI update mechanism. The applicable license still governs Product deployment.

1. Obtain `demos/nucleo-f401re/images/ipmc-nucleo-f401re-tier1.hex` and verify its file hash against `SHA256SUMS.txt` , or build it with the demo packaging script. This is the application image, linked from `0x0800C000` ; the combined board BIN is for initial ST-LINK drive programming, not UART loading.
2. Connect the NUCLEO ST-LINK USB connector and find its virtual COM port. The current bench uses COM4; COM numbers are assigned by Windows. The console is 115200 baud, 8 data bits, no parity, one stop bit. Close other serial applications first.
3. Install Python 3 and `pyserial` (`python -m pip install pyserial`). From the repository root, replacing COM4 with the port found in step 2, run:

```
python tools/fwupdate/fwupdate.py --port COM4 --uart cli `
  --enter cli --enter-port COM4 update `
  demos/nucleo-f401re/images/ipmc-nucleo-f401re-tier1.hex
```

4. The tool sends `DEBUG BOOTLOADER` to the running CLI and requests confirmation if restarting interrupts payload power. The manual confirmed form is `DEBUG BOOTLOADER CONFIRM` ; `--confirm` on the host tool intentionally supplies confirmation automatically. This resets the application and can drop payload power.
5. The tool waits for `BOOT>` , checks the HEX/header/digest on the PC, sends `ERASE APP` and `CONFIRM` , streams the Intel HEX records after `LOAD` , runs `VERIFY` , compares the returned SHA-256 with the image header, and sends `RUN` . A successful run exits 0. Ordinary APP update preserves bootloader sectors 0-1 and FRU-store sector 2; the application uses sectors 3-7.

6. Reopen the CLI or run the helper below. Check the Tier-1 identity, self-test, FRU checksums and IPMB-A response after the reload:

```
powershell -NoProfile -ExecutionPolicy Bypass `
  -File crates/ipmc-nucleo-f401re/gen.ps1 console -ComPort COM4 `
  "sys status" "fru show" "ipmb stats"
```

For manual loading, the bootloader accepts `WAKEUP`, `INFO`, `ERASE APP`, `CONFIRM`, `LOAD`, Intel HEX records, `VERIFY` and `RUN`. Follow the exact ordering and prompt handling below; the host tool is preferred for full images. Use `--no-run` when deliberately remaining in the bootloader after verification. The application UART is also supported with `--uart app --enter app` and a separate 3.3 V UART connection on PB6/PB7; it is not required for console CLI loading.

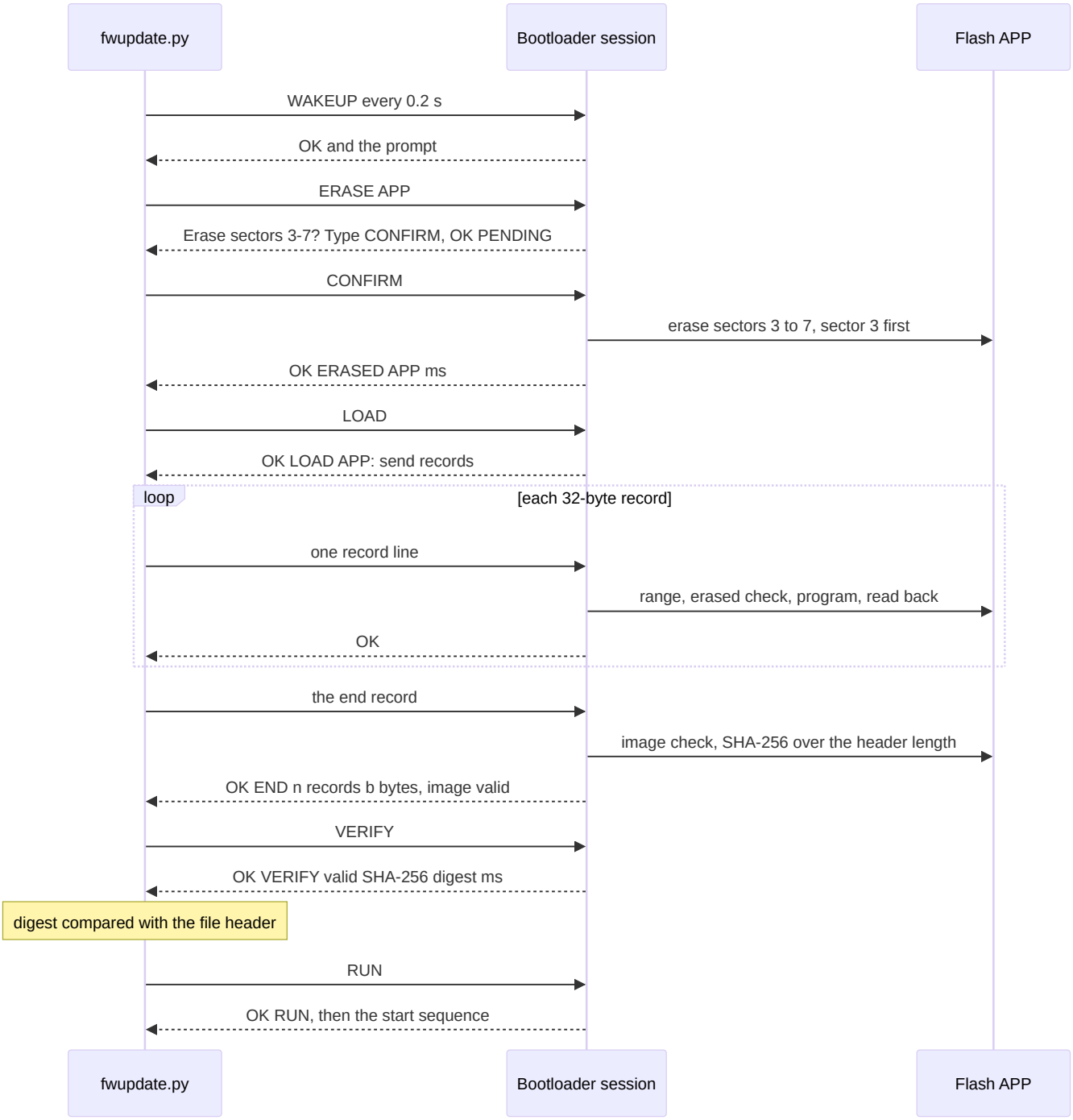
If the port cannot be opened, close its other user and recheck the COM number. If the running firmware cannot enter the bootloader, fit the documented boot-entry jumper PC10 to 3.3 V and use the field procedure in `tools/fwupdate/README.md`, or recover using the combined board image through ST-LINK. A failed or interrupted APP load is not a reason to erase the FRU store or overwrite the bootloader. SHA-256 validates integrity; this mechanism does not authenticate a publisher or verify a firmware signature. Preserve the supplied console log and exit status when recording an update test.

16.5.2 Host tool and wire sequence

`tools/fwupdate/fwupdate.py` (settings in `fwu_settings.py`, README in `tools/fwupdate/`; `--port` is required, `--uart cli` or `app` names the port and picks the baud rate):

1. Wake: the tool sends `WAKEUP` every 0.2 s (60 s limit; `--field` has none) until the `BOOT>` prompt arrives. With `--enter app|cli --enter-port COMy` it first opens the runtime port and sends the entry command (`--confirm` appends `CONFIRM`; without it the tool asks on stdin while payload power is on).
2. Pre-check on the PC, refused before anything is sent: every byte inside `0x0800_C000..0x0808_0000`; header magic, format, size, length, vector address; vector words 0 and 1; the SHA-256 recomputed with `hashlib` equal to the stored digest (`--no-precheck` skips it).
3. `ERASE APP` and `CONFIRM` (up to 20 s for the answer; about 3.5 s measured), then `LOAD`.
4. Records, stop-and-wait: the image is re-chunked into aligned 32-byte records with an 04 record at each 64 KB boundary, an all-FFh chunk is skipped, and each record is answered before the next goes out (`--as-is` sends the file's records). `ERR FORMAT`, `ERR CHECKSUM`, `ERR LINE_TOO_LONG` and `ERR LINE_DAMAGED` resend the record, up to 3 times; any other `ERR` stops the run and names the record and the answer. `ERR ABORTED` (a damaged leading `:` ended record mode; the erase stays in force and every earlier record is programmed) resumes: the tool sends `LOAD`, the last 04 record sent, then the refused record, up to 3 resumes per update.
5. The end record answers `OK END <n> records <b> bytes, image valid`; `VERIFY` answers `OK VERIFY valid SHA-256 <hex>`, and the tool compares that digest with the digest in the file's header (a difference is a refusal; with `--as-is` or a file without a readable header the tool says it did not compare); then `RUN` (not with `--no-run` or `--field`).

Exit codes: 0 success, 1 a refusal or an `ERR` answer, 2 usage, 3 a timeout. A port that cannot be opened is a refusal (`refused: <port>: <message>`). The field procedure fits the jumper PC10 to 3.3 V, runs `fwupdate.py --port COM5 --field update app.hex`, powers the board, and removes the jumper after the update.



16.6 Update recovery and protected regions

- No bootloader command names sector 0 or 1: the erase targets are sector 2 and sectors 3 to 7, and `load_allowed` refuses any record below `0x0800_C000`.
- A second, independent refusal sits in the register layer: `bsp/flash_regs.rs` (`FlashRegs`) refuses to erase sectors 0 and 1 and to program any address below `0x0800_8000` (`Err(Range)`), for the runtime's store driver and for the bootloader alike; the runtime has no command that erases or programs anything but sector 2.
- `ERASE APP` erases sector 3 first, which holds the vector table and the header, so a power cut at any point of an update leaves an image that fails the check; the board then stays in the bootloader at the next reset and takes the update again.

- The digest covers the stated length, so a truncated or partly written image is a `digest mismatch` or a `bad header`, never a start.
- NVMRO asserted refuses every erase and program (`ERR NVMRO`).
- Protection is software only: no sector is write-protected in the option bytes, and neither image reconfigures PA13/PA14 (SWD) or sets readout protection. A faulty bootloader image flashed on the bench is recovered over SWD or by a drive copy of a good `ipmc-board.bin`.

16.7 Measured numbers

Current Tier-1 console reload, 11-13-2026, NUCLEO-F401RE, COM4 at 115200 baud:

Quantity	Current measured value
Application image / combined board image	133,120 / 182,272 bytes
Bootloader BIN	23,360 bytes
Host CLI update duration	38.1 s, exit 0
Bootloader end-of-load response	`OK END 4157 records 133000 bytes, image valid`
Verification	`OK VERIFY valid SHA-256 182777d6d3055c3c37b6fa7b1bbd510cee6f933709bd36c8b4069a4f66347503 164 ms`
Restart	`OK RUN`; subsequent boot confirms Tier-1 FRU identity and self-test `55h 00h`
Runtime sections	text 132,804 bytes; data 184 bytes; BSS 23,164 bytes
Bootloader sections	text 23,320 bytes; data 40 bytes; BSS 6,176 bytes

The full application file includes its header and padding; the loader's programmed-byte count excludes records skipped or interpreted as addressing/termination metadata. See the demo image manifest for file hashes and build provenance. The following older measurements remain historical evidence for their stated image and date, not the current package's size or test result.

Bench measurements of 10-26-2026 (release build, 115200 baud, 84 MHz; `local_only/Documentation/ipmc_boot_plan.md` sections 13.2 and 13.3):

Quantity	Value
Runtime image	139 456 bytes, 4 354 records (32-byte records; an all-FFh chunk is not sent)
Full update over APP (USART1)	about 75 s, no record resent
Full update over CLI (USART2)	about 40 s
`ERASE APP` (sectors 3 to 7)	3.5 s to 3.6 s

Quantity	Value
SHA-256 of the image at start-up	172 ms for 139 456 bytes (171 ms for the earlier 138 524-byte image, the `Image check:` trace line)
`SYS RESET` to the runtime banner	3.18 s in the bench build (3 s countdown); 187 ms in the field build
Bootloader image	22 336 bytes of 32 768

17. Constraints and Limitations

- **Tier 1 in the public build.** Optional and Tier-2/3 commands answer C1h; there is no event generation, no FRU activation control by the Chassis Manager and no subsidiary FRU. The commercial release build adds Tier-2 and Tier-3 (section 7.2); its open bench checks are in `Documentation/ipmc_test_coverage.md`.
- **FAST-MODE is proven on IPMB-A only.** The one-Aardvark bench has IPMB-A; FAST-MODE on both buses waits for the second Aardvark.
- **FAST-MODE rates.** The bit-bang master reaches about 147 kHz in FAST-MODE and the hardware master about 335 kHz; both keep the fast-mode minimum low and high times and stay below 400 kHz.
- **A processor reset drops payload power on this board.** The reset releases PWR_EN with every GPIO (section 8.2, owner decision 21), so a Cold Reset, a bootloader entry and a firmware update drop the payload's power while PWR_EN is asserted; a Warm Reset and a payload reset pulse do not. The entry commands warn first and need `CONFIRM`. A target board holds PWR_EN externally.
- **One outstanding request.** The requester sends one request at a time; a second `submit` returns `Busy` until the first completes or is cancelled.
- **One master transmission per main-loop pass.** A response burst drains one frame per pass; the egress queue holds `IPMB_TXQ_DEPTH` (4) responses, and further requests wait in the ingress FIFO.
- **Bit-bang deafness.** While the bit-bang master transmits, the slave peripheral is off; a frame another master sends then is lost and recovered by its retry.
- **Flash erase stall.** An erase stops the core for up to 500 ms (typical 250 ms; 198 ms measured); the IPMB slaves are off the bus for that time (the Chassis Manager sees timeouts and retries). Erases are rare: the store sector holds 31 records, the erase comes once per 31 saves, and only when the bus has been quiet for `IPMC_FRU_ERASE_QUIET_MS`.
- **One-sector FRU store.** The store has no second copy (owner decision 10-25-2026). A power cut between the erase of a full sector and the end of the first record's programming loses the stored image, and the next start-up loads the built image, unit fields included.
- **Reflash.** A drive copy of `ipmc-board.bin` writes FFh over the store sector and erases the stored FRU image; an SWD program of the runtime ELF keeps it (section 9.1). The firmware has no guard against either. A UART update (section 16) erases sectors 3 to 7 only and keeps the FRU store.
- **Digest, no signature.** The bootloader proves the image bytes match the digest in their header; it does not prove who built them (owner decision, PLAN decision 33). A signature check does not fit the 32 KB of sectors 0 and 1 with the session.
- **The bootloader is changed on the bench only.** A new bootloader needs SWD or a drive copy; the UART `LOAD` refuses sectors 0 to 2. While the bootloader runs the IPMC answers nothing on either IPMB and does not drive PWR_EN.
- **No hardware watchdog.** A flash busy flag that never clears, or any other hang, has no automatic recovery; the bootloader starts none either, so a runtime watchdog would not run into its session.
- **Console output.** Output goes into a 4096-byte ring that the interrupt sends (356 ms at 115200 baud). A message waits only while the ring is full (86.8 us per byte); a trace line that does not fit is dropped whole and counted, so tracing cannot stall the IPMB. A long listing (`DEBUG DUMP`) runs as a paced job and is stopped by a key press.
- **Capture log.** The log lives in RAM and is empty after a reset or power cycle; it holds the newest `IPMC_LOG_SIZE` - 1 bytes of records (a 23-byte frame takes 29 bytes). A dump of a full log is about 25 KB.
- **Application port.** The IPMC answers one request line per main-loop pass and drops an answer the transmit ring cannot take whole. The port has no emulator model: it runs on the board only (COM5 on the bench). The six unit fields are the only FRU content it writes; `IPMI` runs requests locally and puts nothing on the IPMB. The bench runs without RTS/CTS.

- **Straps forced on the bench.** GA and NVMRO come from `defs.rs`; the pin paths are built but not exercised end to end, and the NVMRO electrical polarity is not yet confirmed against VITA 46.0.
- **Manufacturer ID 000000h** (unspecified) until an IANA enterprise number is registered.
- **IPMB-B** is built in and enabled. On the board its start-up and idle state are checked; traffic on IPMB-B, both buses together and failover are tested in the emulator until a device drives IPMB-B.
- **IPMC pull-ups on the IPMB (bench deviation).** `IPMB_PULL_UPS true` turns on the internal pull-ups (about 40 kΩ) of PB8, PB9, PA8 and PB4 so an unwired bus reads idle-high; Rule 9.2.3-2 forbids IPMC pull-ups on the System IPMB. On a 2.49 kΩ backplane they lower each line to about 2.34 kΩ and add about 80 µA of sink current. They are too weak to be a bus's only pull-up; a device on the bench still brings its own. A module for a chassis builds with `false`.
- **Duplicate window 3000 ms**, not the VITA 5 s sequence expiry: a Chassis Manager that reuses a sequence number with identical data inside 3 s gets the cached response.
- **Emulator flash is RAM.** Flash persistence is tested in `ipmc-fru` against `RamFlash` and on the board. The emulator runs the same `Persist` over a `RamFlash` and keeps it across a reset and a `#reset` power cycle.
- **No custom registers.** The firmware uses the STM32F401 peripherals through the `stm32f4` PAC; each driver names the RM0368 section of every register field it writes (I2C chapter 18, flash section 3.5, RCC chapter 6, ADC chapter 11, GPIO chapter 8).

Contact IronKernel

IronKernel provides board design review, bring-up and bare-metal firmware development for VPX management controllers. We can help port the Rust IPMC, integrate the IPMB and adapt FRU records, sensor definitions and payload controls.

To discuss firmware integration, a reference design or custom development, send your target board, project goal, required VITA 46.11 tier and Chassis Manager interface requirements.

IronKernel.com | support@ironkernel.com