20 — Fault Analysis & Post-Mortem

> **TraxProbe & Traxcope User Manual** · Version 1.0.0 · Updated 2026-07-28
 · https://embedya.com

A device that hard-faults in the field usually leaves you nothing but a reset. With **TraxFault** enabled, the fault handler captures the CPU registers, the ARM fault-status registers, and the running RTOS task into a RAM region that *survives the reset*; on the next boot the record is shipped to Traxcope, which decodes the fault bit-by-bit, tells you the faulting address, and symbolizes PC/LR into `function() at file:line`. Every boot also reports *why* the device reset (power-on, watchdog, software, brown-out, ...).

Runtime cost: **zero**. TraxFault runs only inside the fault handler (the system is already dead) and once at boot / session start. No hooks or conditionals are added to any `TRAX_LOG` / `TRAX_VAR` / ISR emit path.

How it works

```
── crash ──────────────────► fault handler captures registers + CFSR + task
                            into .trax_noinit (magic + CRC), requests reset
── reboot ─────────────────► trax_init() validates the record, caches it,
                            frees the persistent slot, latches reset reason
── Traxcope connects ──────► reset reason emitted every session start;
                            the recovered crash record emitted once per boot —
                            crash notification raised immediately, dump
                            auto-archived to the Faults section
── Fault View ─────────────► decoded CFSR/HFSR, faulting address, register
                            dump, PC/LR symbolized against your ELF
```

The record is sealed with a magic word and a CRC-32 written last, so a power loss mid-capture can only ever produce an *invalid* record — never a wrong one.

On the device

TraxFault is **opt-in** (it defines the Cortex-M fault handlers, which would collide with the empty non-weak ones CubeMX generates — see step 3).

1. Enable the module

```
/* trax_config.h */
#define TRAX_CFG_FAULT_ENABLE   1
```

2. Add the noinit section to your linker script

The record lives in a section that must be `NOLOAD` and outside the `.bss` zero-fill range. Copy the block from `linker/trax_noinit.ld` into your GCC script's `SECTIONS { }`, after `.bss`:

```
.trax_noinit (NOLOAD) :
{
  . = ALIGN(4);
  KEEP(*(.trax_noinit))
  . = ALIGN(4);
} > RAM
```

Pick a RAM region that is powered in your normal run mode and not wiped by a bootloader. Access latency is irrelevant (the section is only touched at fault time and boot time), so on multi-bank parts any always-on SRAM/DTCM works. If your project already has a `.noinit` section, point TraxFault at it instead: `#define TRAX_CFG_FAULT_SECTION ".noinit"`.

3. Clear the way for the fault handlers

With `TRAX_CFG_FAULT_HANDLERS` at its default of `1`, TraxProbe defines `HardFault_Handler` — and on Mainline cores (M3/M4/M7/M33...) also `MemManage_Handler`, `BusFault_Handler`, `UsageFault_Handler` — as capture shims. The vector table picks them up by name.

- **STM32CubeMX users:** delete the generated empty handlers from `stm32xxxx_it.c` (they are non-weak and would collide).
- **Keeping your own handlers:** set `TRAX_CFG_FAULT_HANDLERS 0` and call `trax_fault_capture()` from your own naked shim (the required asm is documented in `trax_fault.h`). The shim switches to a dedicated fault stack, so even a stack-overflow crash is captured cleanly.

4. (Recommended) Report the reset reason

Override the weak hook with your device's reset-status register so every boot reports a normalized reason plus the raw register value:

```
enum trax_fault_reset_reason_t trax_fault_port_reset_reason(uint32_t *p_raw)
{
    uint32_t csr = RCC->CSR;
    *p_raw = csr;
```

```
    RCC->CSR |= RCC_CSR_RMVF;                                /* clear for next
boot */
    if (csr & RCC_CSR_IWDGRSTF) return TRAX_RESET_REASON_IWDG;
    if (csr & RCC_CSR_SFTRSTF)  return TRAX_RESET_REASON_SOFTWARE;
    if (csr & RCC_CSR_PWRRSTF)  return TRAX_RESET_REASON_POWER_ON;
    if (csr & RCC_CSR_PINRSTF)  return TRAX_RESET_REASON_PIN;
    return TRAX_RESET_REASON_UNKNOWN;
}
```

The default (no override) reports `UNKNOWN` — everything else still works.

That's it. No application code changes: capture, recovery, and emission are automatic. If you want to react to a crash in firmware (e.g. enter a safe mode), `trax_fault_get_last()` returns the recovered record (or `NULL`) and `trax_fault_get_reset_reason()` the latched reset reason.

What gets captured

Field group	Content	Availability
Core registers	R0–R12, SP, LR, PC, xPSR, EXC_RETURN, MSP, PSP	all Cortex-M
Fault kind	HardFault / MemManage / BusFault / UsageFault (from ICSR.VECTACTIVE — correct even when a fault escalates to HardFault)	all Cortex-M
Fault-status registers	CFSR, HFSR, DFSR, MMFAR, BFAR, AFSR, SHCSR	ARMv7-M+ only (M3/M4/M7/M33...)
RTOS context	current task name (16 chars), task priority, tick count, scheduler state (bare metal / not started / running / suspended)	task fields: FreeRTOS with scheduler started; scheduler state: all
Stack snapshot	`TRAX_CFG_FAULT_STACK_SNAPSHOT` bytes (default 512, 0 disables) from the faulting SP upward — exception frame first, caller frames above. Clamped to the end of stack RAM (auto-detected; override with `TRAX_CFG_FAULT_RAM_END`)	all Cortex-M
Timing	TRAX timestamp + session id of the crashed run	all

The snapshot costs `TRAX_CFG_FAULT_STACK_SNAPSHOT` bytes in the noinit section plus the same again for the boot-time cache; it is copied only inside the fault handler, so the runtime cost stays zero.

On **ARMv6-M** (Cortex-M0/M0+) there is no CFSR/BFAR in silicon — the record carries the register snapshot and PC/LR, and Traxcope states the limitation explicitly instead of showing zeros. Validity flags in the record mark which groups are present, so the display always degrades gracefully.

Flight recorder — the last moments before the crash

Knowing *where* the firmware died is often not enough — you want to see *how it got there*. With the flight recorder enabled (it is, by default, whenever TraxFault is on), the fault handler also preserves the newest trace frames still sitting in the trace ring buffer — the logs, ISR entries, markers, task switches and variable updates that were emitted just before the crash but never made it to the host. They are copied into a CRC-sealed noinit slot (`TRAX_CFG_FAULT_TRACE_SNAPSHOT` bytes, default 1024, 0 disables) and replayed to Traxcope on the first session start after the reboot, right behind the crash record.

The replay is paced by `trax_process()` — one small chunk per call, and only when the ring has comfortable headroom — so the pre-crash history can never starve or overflow the *new* session. It is emitted once, then invalidated.

The history is most complete exactly when you need it most: if the transport was congested or streaming had stopped (e.g. a ring overflow preceding the crash), the undrained tail is large and the recorder captures all of it up to the configured cap.

On RTOS builds the fault handler additionally dumps its live **task and object tables** into a second CRC-sealed noinit block (the dynamic-metadata snapshot). Budget it deliberately: `TRAX_CFG_FAULT_DYNMETA_TASKS` × 40 B + `TRAX_CFG_FAULT_DYNMETA_OBJECTS` × 32 B (defaults 16/16 → 1168 B including the header). The caps are honest about their limits — the wire frames carry both the live count *and* the stored count, so Traxcope flags any truncation and names the option to raise. Match the caps to your `TRAX_CFG_MAX_RTOS_TASKS` / `_OBJECTS` to never truncate; shrink them to save noinit RAM on small parts. Set both to 0 to disable (bare-metal builds force 0 automatically).

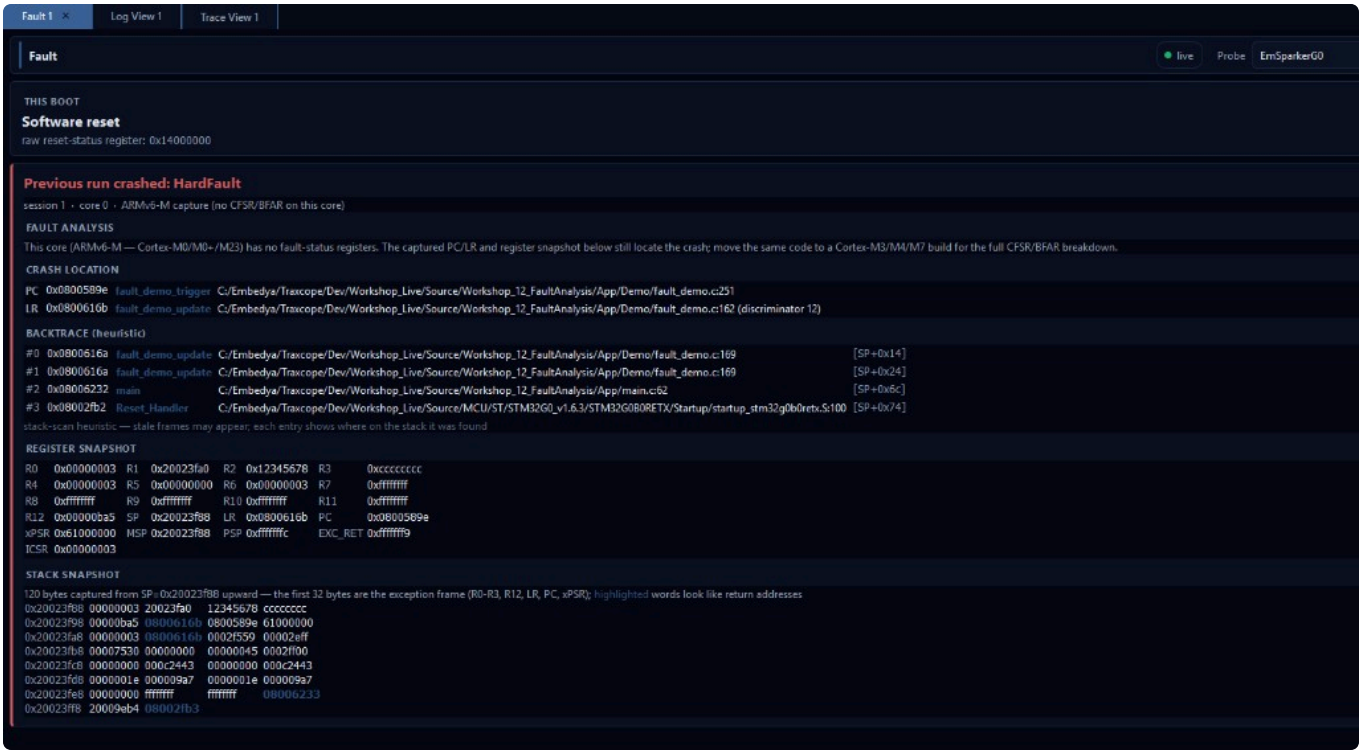
The whole crash-dump footprint is compile-time accountable: `TRAX_FAULT_NOINIT_TOTAL` (in `trax_config_fault.h`, cross-checked against the real struct sizes by static asserts) sums the fault record, the flight buffer and the dynmeta block — 2884 bytes with all defaults. If your `.trax_noinit` section maps to a **fixed-size** region (a `MEMORY` carve-out, a bootloader-shared block, battery-backed SRAM), set `TRAX_CFG_FAULT_NOINIT_MAX` to that region's size and the build fails the moment a config change makes the dump outgrow it — instead of the linker silently placing half a snapshot. With the default auto-sized section fragment the linker grows the section as needed, but setting the budget still documents intent.

In Traxcope

You don't have to be looking for a crash to learn about one. The moment a fault record arrives at session start, Traxcope raises a **crash notification** — a non-blocking toast in the corner of the window ("Previous run crashed: BusFault in task `motor_ctrl`") with an **Investigate** button that jumps straight to the Fault View, plus a red counter badge on the **Faults** icon in the activity bar. The live stream is never interrupted: the toast can be dismissed and the badge stays until you open the Faults section.

Crash handling runs app-level, not view-level: archiving to disk, the notification, and the Crash History source are all created even when no Fault View is open.

For the full post-mortem, open the **Fault View** via Toolbox → **Fault**.



Screenshot: Fault View after a HardFault on a Cortex-M0+ (STM32G0) — reset-reason card, symbolized crash location (PC on the faulting line, LR on the caller), heuristic backtrace down to `Reset_Handler`, register snapshot, and the stack hexdump with return-address candidates highlighted. On ARMv7-M+ cores the FAULT ANALYSIS section additionally decodes CFSR/HFSR bit by bit with the faulting address.

The view has four areas:

- **THIS BOOT** — the reset-reason card, updated on every session start: normalized reason (power-on, pin, software, IWDG, WWDG, brown-out, low-power, lockup) plus the raw reset-status register value. Watchdog and lockup resets are painted red — those are the "nobody pressed reset" cases worth investigating.

- **FAULT ANALYSIS** — shown when a crash record arrived: a headline ("Previous run crashed: BusFault in task `motor_ctrl`") and a context line (session, core, RTOS tick, task priority — including a red flag when the fault hit inside a *suspended-scheduler* critical section), then the CFSR/HFSR decoded **bit by bit** in plain language with likely-cause hints — e.g. `PRECISERR` with the exact faulting address from BFAR, `INVSTATE` ("branch to an address without the Thumb bit — often a corrupted function pointer or a return address smashed by a buffer overflow"), `STKERR` ("stacking failed — stack overflow is the usual suspect"), `NOCP` ("FPU instruction with the FPU disabled"), `FORCED` escalation, and so on.
- **CRASH LOCATION** — PC and LR resolved to function names. Traxcope parses the symbol table (`.symtab`) of the ELF configured for the probe **in-process** — no toolchain or other software needs to be installed on the analysis machine, and it works with ELFs from any compiler (GCC, Clang/armclang, IAR, Keil). If a GNU/LLVM `addr2line` happens to be available (`PATH` , STM32CubeIDE, or an Arm GNU Toolchain install), Traxcope additionally upgrades each entry with `file:line` detail from DWARF.
- **BACKTRACE (heuristic)** — the stack snapshot is scanned for words that look like Thumb return addresses into the ELF's `.text` ; each hit is symbolized and listed with the stack offset it was found at. This is a conservative stack scan, not a real unwind: stale return addresses from already-returned functions can appear, so treat entries as candidates — the `[SP+offset]` column tells you how deep each one sat.
- **REGISTER SNAPSHOT** — the full dump: R0–R12, SP, LR, PC, xPSR, MSP, PSP, EXC_RETURN, and (on ARMv7-M+) CFSR/HFSR/DFSR/MMFAR/BFAR/SHCSR/ICSR.
- **STACK SNAPSHOT** — a hexdump of the captured stack, addressed from the faulting SP. The first 32 bytes are the hardware-stacked exception frame; recognized return-address candidates are highlighted so you can eyeball the call chain.
- **TASKS AT CRASH** — the crashed run's RTOS task and kernel-object tables, dumped by the fault handler (RTOS builds). Every task with its name, handle, priority, stack size and a latched **STACK OVERFLOWED** flag; every queue/semaphore/mutex with its type and dimensions. If the live table held more entries than the noinit caps could store, the card shows a truncation warning naming the exact config option to raise (`TRAX_CFG_FAULT_DYNMETA_TASKS` / `_OBJECTS`) — you always *know* when the snapshot is partial, and by how much.
- **PRE-CRASH HISTORY (flight recorder)** — the decoded event/log timeline recovered from the crashed run's final moments, newest first, each row stamped with its distance to the crash (`T−1.42 ms`). Logs are rendered through their format strings, ISRs/markers/variables show their metadata names — the same detail you would have seen live. Use it to answer the *how*: which ISR ran last, what the state machine did, what a variable was drifting toward when the fault hit.

Opening the crash history in the full views

As soon as the flight-recorder replay finishes arriving, Traxcope **automatically** rebuilds the recovered frames into a complete read-only data source named *"Crash History S<n>"* — registered exactly like a loaded recording, no click required. (The card's **Open in Trace / Event / Scope views** button does the same thing manually and flips to "Opened as …" once the source exists.) Select it as the source in any view:

- **Trace View** — task, ISR and main-loop swimlanes for the final milliseconds, with normal zoom and cursors. Whatever was still executing at the crash instant is closed at the last captured timestamp, so the lane activity runs right up to the moment of death.
- **Event View** — the same rows as the card, but filterable, searchable, and with the executing-context column. RTOS task events (switch in/out, ready, delay, priority changes, stack overflow) resolve to task names.
- **Log View** — the pre-crash logs with levels and tags.
- **Scope View** — variable samples from the crashed run, both async updates (`TRAX_VAR_SET`) and high-rate stream channels, plottable against the crash time.

Two reconstructions deserve a note on *how* they work, because the crashed session's runtime state did not survive the reset:

- **Task lanes.** Kernel frames carry raw task handles (TCB addresses). On RTOS builds the fault handler also dumps its **live task and object tables** into noinit RAM (the *dynamic-metadata snapshot*, `TRAX_CFG_FAULT_DYNMETA_TASKS` / `_OBJECTS` entries, default 16 each), and Traxcope receives them right after the crash record. That table is authoritative: it has the exact names, priorities and stack sizes at the moment of the crash — *including tasks created dynamically mid-run that no longer exist after the reboot*. Handle resolution runs in priority order: the crashed run's own table first, then the current session's map (startup task handles repeat across boots), and finally a raw-address placeholder lane so no event is ever dropped.
- **Stream samples.** Stream timing normally anchors on the session's stream-start control frame, which is gone. But every captured data frame carries its own sequence counter *and* its own wire timestamp, and the sample period is static metadata — so the anchor is re-derived from the first captured frame. Sample spacing is exact; the absolute anchor can be offset by at most one frame's transmit latency.

Timestamps are in the *crashed run's* timebase — absolute values are arbitrary, distances to the crash are what matter. The source stays open across reconnects until you open a newer crash from the same probe. (Kernel *object* events — queue/semaphore/mutex operations — are not yet materialized in the full views.)

The record is emitted **once per boot** at the first session start after the crash and stays in the view for the whole session, so you can connect long after the device rebooted and still get the post-mortem.

The crash dump archive — every crash saved, analyzable forever

A crash dump used to live only in the current session: the next crash, the next reconnect, or closing Traxcope discarded it. Not anymore — every crash is archived automatically, and the archive has its own home: the **Faults** section in the activity bar (next to Probes and Recordings).

- **Auto-save.** Every complete crash any live probe delivers is written to a `.traxcrash` file in the application data folder (a few KB each) — no Fault View needs to be open. The file is re-saved as the pieces arrive — fault record, then the crashed run's task tables, then the flight replay — so even a partial delivery is preserved. The file also embeds the session metadata needed to interpret it later: the schema snapshot, swimlane entities, handle maps, decode parameters, and the ELF path for symbolization.
- **The Faults list.** Each archived dump appears as a card — fault kind, crashed task, probe, session, capture date. Click a card to open it: the dump materializes as a full source and the Fault View is raised on it, so you read the crash card, backtrace and task tables **with no device attached**; the same source appears in the Trace/Event/Scope/Log pickers for the pre-crash history. The card's context menu offers **Open**, **Export…**, and **Delete** — recording is automatic, keeping or discarding is your decision.
- **Import / Export.** `.traxcrash` files are portable. Export a dump to hand to a colleague; import a file a field technician sent you (it is copied into the archive so it stays in the list) and analyze it as if the device were on your desk.
- The Fault View's bottom **CRASH DUMP ARCHIVE** card offers the same archive inline: pick a dump, **Open**, **Import…**, or **Export current…** for the selected probe's crash.

Because the archive is written the moment the data arrives, "I crashed it twice and lost the first dump" can't happen — both are on disk, each under its own probe + session + date name.

Reading a post-mortem — a 60-second routine

1. **Reset card first.** IWDG reset with *no* crash record means the firmware hung without faulting (a fault record + software reset means TraxFault caught it).
2. **Headline + decode.** The CFSR decode usually names the mechanism directly: precise bus fault at address X, null-pointer write, undefined instruction, stack overflow during exception stacking.
3. **Crash location.** PC is the faulting instruction; LR tells you who called it. An implausible PC with a sensible LR usually means a wild jump — check the LR site for the corrupted function pointer.
4. **Registers.** With the `file:line` in hand, the register values map onto the local variables of that function — often enough to identify the bad pointer without a debugger.

Troubleshooting

Symptom	Likely cause
Linker: multiple definition of `HardFault_Handler`	CubeMX's empty handler still in `stm32xxxx_it.c` — delete it, or set `TRAX_CFG_FAULT_HANDLERS 0` and call `trax_fault_capture()` from your own shim
Crash happened but no record after reboot	`.trax_noinit` section missing from the linker script (record landed in a zero-filled orphan), RAM region wiped by a bootloader, or power was fully cycled (RAM contents lost)
Reset reason always UNKNOWN	`trax_fault_port_reset_reason()` not overridden for your MCU
Reset reason wrong / stale	Something else reads the reset-status register without clearing it, or clears it before TraxFault runs — reset-status flags accumulate until cleared
No CFSR/HFSR decode shown	ARMv6-M target (Cortex-M0/M0+): those registers don't exist; the view says so and shows the register snapshot instead
PC/LR not symbolized	No ELF configured for the probe, or the ELF is fully stripped (no `.symtab`) — keep symbols in release ELFs; they cost nothing on the target
Function names shown but no `file:line`	Normal without a toolchain installed — names come from the ELF symbol table; `file:line` needs `addr2line` (checked on `PATH`, then STM32CubeIDE / Arm GNU Toolchain locations)
PC symbolizes to nonsense	ELF is stale — rebuild and re-flash so the ELF matches the firmware that crashed
Record for a *previous* crash reappears	It doesn't — records are emitted once and the persistent slot is invalidated at boot. If you see one, the device crashed again