02 — TraxProbe Integration Guide

> **TraxProbe & Traxcope User Manual** · Version 1.0.0 · Updated 2026-08-25
 · https://embedya.com

This is the firmware-side reference. It covers project setup, every configuration decision (mandatory vs optional), transports, metadata storage modes, RTOS vs bare-metal integration, and tuning.

If you just want the fastest possible bring-up, do Chapter 01 first and come back here for the details.

---

1. Library layout

```
TraxProbe/
  inc/              Public headers — include trax.h and you have everything
  config/           Default sub-configurations (NEVER edit; override in
trax_config.h)
  src/              Core engine
  os/               RTOS ports (FreeRTOS, BareMetal) + common tables
  hw_port/          Hardware ports (ARM_Cortex_M, Zynq, Xtensa, Generic) —
header-only
  transports/       Built-in RTT transport + transport dispatcher
  linker/           trax_meta.ld — metadata placement fragment (GNU ld)
                    esp/trax_meta.lf — same placements as an ESP-IDF ldgen
fragment
```

Application code only ever includes `<trax.h>`. It pulls in the full API: logs, vars, streams, ISR, markers, state machines, diagnostics, and (when configured) the RTOS port.

Project import

The firmware project needs **exactly three** settings. Nothing else.

What	Where
TraxProbe include directory	`TraxProbe/inc/` — all library headers; `config/` and `hw_port/` resolve internally via relative includes, selected by the macros in your `trax_config.h` (`TRAX_CFG_HW_PORT`, `TRAX_CFG_RTOS_TYPE`, …)

What	Where
Metadata linker script	`TraxProbe/linker/trax_meta.ld` as an *additional* linker script (ESP-IDF: `linker/esp/trax_meta.lf` — see §4)
Your `trax_config.h` folder	Any folder on the include path (this manual uses `App/Cfg/`)

Do not add extra include paths, and do not cherry-pick `src/`, `transports/`, or `os/` trees. Hardware port, transport, and RTOS port are compile-time selections from `trax_config.h`. All hardware ports are header-only.

2. Configuration reference

All configuration lives in **one user file**: `trax_config.h`, in any folder that is on the include path. The library includes it from `trax_config_default.h`, then layers modular defaults on top (`config/trax_config_log.h`, `_buffer.h`, `_hw_port.h`, `_rtos.h`, `_meta.h`, `_diag.h`). Anything you do not define falls back to a documented default; anything mandatory that is missing produces a clear `#error` at compile time.

2.1 Mandatory settings

Setting	Purpose	Example
`TRAX_CFG_HW_PORT`	Selects the hardware port	`TRAX_HW_PORT_ARM_CORTEX_M`
`TRAX_CFG_TIMER_FREQ_HZ`	Frequency of the timestamp timer (sent to the host so it can decode time)	`64000000U`
`TRAX_CFG_TIMESTAMP_TIMER_VAL`	C expression reading the hardware counter used as fine time — **TICK_TIMER mode only**. Any running counter works: SysTick, DWT, a peripheral timer, etc.	`(SysTick->VAL)`
`TRAX_CFG_TICK_COUNTER_PERIOD`	Counts of that counter per tick — **TICK_TIMER mode only**	`64000U` (1 ms @ 64 MHz)

> **FREERUN mode:** the hardware port supplies its own free-running counter automatically — `TRAX_CFG_TIMESTAMP_TIMER_VAL` and `TRAX_CFG_TICK_COUNTER_PERIOD` are not required. Only `TRAX_CFG_HW_PORT` and `TRAX_CFG_TIMER_FREQ_HZ` remain mandatory. You must still call `trax_timestamp_tick()` periodically — at least once before the counter's upper bits wrap (see §2.2).

2.2 Timestamping — the one decision that matters most

Every frame carries a 32-bit timestamp split as `[tick:TICK_BITS][fine:32-TICK_BITS]` (default 8/24). Two modes:

TICK_TIMER (default) — software tick counter + a hardware timer register as fine time. Works on every MCU, typically using SysTick:

```
#define TRAX_CFG_TIMESTAMP_MODE       TRAX_TIMESTAMP_TICK_TIMER  /* default */
#define TRAX_CFG_TICK_BITS            8
#define TRAX_CFG_TIMESTAMP_TIMER_VAL  (SysTick->VAL)
#define TRAX_CFG_TIMESTAMP_TIMER_DIR  TRAX_TIMER_DIR_DOWN        /* SysTick
counts down */
#define TRAX_CFG_TICK_COUNTER_PERIOD  64000U                     /* SysTick-
>LOAD + 1 */
#define TRAX_CFG_TIMER_FREQ_HZ        64000000U
```

You **must** call `trax_timestamp_tick()` from the tick interrupt:

- Bare metal: from your `SysTick_Handler`.
- FreeRTOS: automatic — the SysTick trace hook calls it for you.

FREERUN — the hardware port provides a 32-bit free-running counter automatically (e.g. `DWT->CYCCNT` on ARM, `mcycle` on RISC-V), wired up via `TRAX_HW_PORT_FREERUN_COUNTER` in the port header. No user-defined timer register is needed; only `TRAX_CFG_TIMER_FREQ_HZ` is required:

```
#define TRAX_CFG_TIMESTAMP_MODE   TRAX_TIMESTAMP_FREERUN
#define TRAX_CFG_TICK_BITS        8          /* tick period auto-derived:
2^24 cycles */
#define TRAX_CFG_TIMER_FREQ_HZ    168000000U
```

Wrap tracking is still required: call `trax_timestamp_tick()` from a periodic ISR (or `trax_timestamp_poll()` from the main loop) at least once before the upper `TICK_BITS` of the counter wrap. At 168 MHz with 8 tick bits the wrap period is 2^{24} cycles ≈ 99 ms — any tick-rate ISR is fast enough.

Non-integer timer frequencies are supported via `TRAX_CFG_TIMER_FREQ_DIV` (actual frequency = `FREQ_HZ / FREQ_DIV`).

2.3 Common optional settings

These are the knobs integrators actually set. Anything omitted here (diagnostics, memory-monitor thresholds, fault-record sizing, RTT buffer channels, …) lives in Chapter 99.

Setting	Default	Why you would chang
`TRAX_ENABLE`	`1`	Set `0` (or compile with `DTRAX_ENABLE=0`) to str and metadata from a pr every macro becomes a function an inline stub
`TRAX_CFG_TRANSPORT`	`TRAX_TRANSPORT_RTT`	Set `TRAX_TRANSPORT_CU` UART/TCP/USB/SPI (s
`TRAX_CFG_RTOS_TYPE`	`TRAX_RTOS_NONE`	Set `TRAX_RTOS_FREERTO` tracing + the TraxCtrl dr When an RTOS is selec also mandatory.
`TRAX_CFG_OUT_BUFFER_SIZE32`	`2048` words (8 KB)	Bigger ring absorbs lon traffic apps use 4096–1 §7)
`TRAX_CFG_IN_BUFFER_SIZE32`	`16` words	Host-to-target comman needs changing
`TRAX_CFG_META_STORAGE`	`TRAX_META_IN_FLASH`	Set `TRAX_META_ELF_ONL` metadata (see §4)
`TRAX_LOG_COMPILE_LEVEL`	`TRAX_LOG_LEVEL_TRACE`	Compile-time log strippi this level cost zero byte **No `TRAX_CFG_` prefix** – `TRAX_CFG_LOG_COMPILE`
`TRAX_FILE_LOG_LEVEL`	= global	Per-file override; define `#include "trax.h"` in rule: no `TRAX_CFG_` pre
`TRAX_CFG_STREAM_CNT`	`0`	Number of variable stre instantiates. Must be a **literal** (not an enum) an every stream TID index
`TRAX_CFG_SM_CNT`	`0`	Number of `TRAX_SM_DE` machines. Sizes the cu table snapshotted into s

Setting	Default	Why you would chang
		`STREAM_GAP` (late join + Must cover every SM TI 16)
`TRAX_CFG_PROJECT_NAME`	`"Unknown"`	Shown in Traxcope's fir
`TRAX_CFG_BUILD_VERSION`	`""`	Human-readable firmwa in Traxcope
`TRAX_CFG_BUILD_ID`	`__DATE__ "T" __TIME__`	Unique build fingerprint pipeline id, ...)
`TRAX_CFG_SYNC_ROLE`	`TRAX_SYNC_ROLE_NONE`	Multi-probe sync maste 18)
`TRAX_CFG_META_NAME_LEN` / `_TAG_LEN` / `_FORMAT_LEN` / `_UNIT_LEN` / `_DESC_LEN`	32 / 32 / 64 / 8 / 32	Metadata string field siz save flash, grow for lon stay multiples of 4)
`TRAX_CFG_CTRL_TASK_PRIORITY` / `_STACK_SIZE` / `_PERIOD_MS`	`1` / `256` words / `10` ms	TraxCtrl drain-task budg Raise priority only wher idle slack and you acce competing with real wor
`TRAX_CFG_CORE_COUNT`	`1`	SMP targets running on multiple cores (1–63)
`TRAX_CFG_BASEPRI`	unset (PRIMASK)	Cortex-M3/M4/M7/M33- TraxProbe critical sectic all IRQs to a BASEPRI urgent than this value a — and must not call an (same contract as FreeI Typical: `configMAX_SYSCALL_IN`
`TRAX_CFG_MAX_RTOS_TASKS` / `_OBJECTS`	`16` / `16`	FreeRTOS task-table ar table sizes. Raise if you tasks / queues / semapl than the default or Trax extras
`TRAX_CFG_RTOS_TASK_NAME_MAX`	`16`	Stored task-name lengt NUL). Set to `configMAX` so names are not trunca
`TRAX_CFG_ISR_YIELD_TO_SCHEDULER`	`1`	FreeRTOS: when a tick with a yield pending (`traceISR_EXIT_TO_SC` `portYIELD_FROM_ISR`), `ISR_EXIT` so Traxcope Scheduler → new task.

Setting	Default	Why you would chang
		`ISR_EXIT` and shows th Cortex-M resume of the
`TRAX_CFG_OWN_FREERTOS_TICK_HOOK`	`1`	`1` = TraxProbe owns `vApplicationTickHook` `configUSE_TICK_HOOK`, `trax_app_tick_hook()` USE_TICK_HOOK uncl the generated stub. `0` = hook and must call `trax_freertos_on_tic`
`TRAX_CFG_FREERTOS_TICK_ISR_PRIORITY`	lowest NVIC prio	Priority shown on the au ISR lane. Override only programs SysTick to so than the FreeRTOS low
`TRAX_CFG_FAULT_ENABLE`	`0`	Opt-in crash capture & p across resets (Chapter
`TRAX_CFG_DIAG_REPORT_PERIOD_MS`	`1000`	Period of on-wire diagn shown in Traxcope's pro

The complete table including diagnostics, memory-monitor, metadata field lengths, RTT buffers, and fault knobs is in Chapter 99.

2.4 Trace ID (TID) headers

Every traceable item needs a 16-bit TID from its feature range. The recommended pattern is one small header per feature, using enum auto-increment anchored at the range's user-start constant — duplicates become impossible:

```
/* trax_var_tid.h */
enum trax_var_tid_t {
    VAR_TID_QUEUE_DEPTH = TRAX_TID_RANGE_VAR_USER_START,   /* 0x2000 */
    VAR_TID_GRID_R,                                         /* 0x2001 */
    VAR_TID_GRID_S,                                         /* 0x2002 */
    /* append new TIDs at the end only */
};
```

Range	Feature	User enums start at
`0x0100–0x1FFF`	Logs	`TRAX_TID_RANGE_LOG_USER_START` (0x0110 — 0x0100-0x010F is library-reserved)

Range	Feature	User enums start at
`0x2000–0x2FFF`	Variables	`TRAX_TID_RANGE_VAR_USER_START`
`0x3000–0x3FFF`	Variable streams	`TRAX_TID_RANGE_STREAM_USER_START`
`0x4000–0x4FFF`	ISRs	`TRAX_TID_RANGE_ISR_USER_START` (0x4010 — 0x4000-0x400F is port-reserved)
`0x5000–0x5FFF`	Markers	`TRAX_TID_RANGE_MARKER_USER_START`
`0x6000–0x6FFF`	Kernel events	library-owned, never user-assigned
`0x7000–0x70FF`	State machines	`TRAX_TID_RANGE_SM_USER_START`

Every runtime macro statically asserts that its TID lies in the correct range, so a wrong TID is a compile error, not a silent decode failure. Stream and state-machine TID *indices* (offset from the range start) must also stay inside `TRAX_CFG_STREAM_CNT` and `TRAX_CFG_SM_CNT` — defining a stream or SM without raising the matching count is a compile error.

3. Transports

3.1 Built-in RTT (default)

```
/* trax_config.h — nothing to do; RTT is the default. Explicitly: */
#define TRAX_CFG_TRANSPORT  TRAX_TRANSPORT_RTT
```

The built-in backend provides the transport functions — no user code required. On the Traxcope side, select the RTT transport and your J-Link.

3.2 Custom transport (UART, TCP, USB, SPI, ...)

```
/* trax_config.h */
#define TRAX_CFG_TRANSPORT  TRAX_TRANSPORT_CUSTOM
```

There is no struct to fill and no registration call. Selecting `TRAX_TRANSPORT_CUSTOM` compiles the RTT backend out, and your application **implements four `trax_transport_*()` functions in the transport `.c`** — the linker binds them by name. If any is missing, the build fails with `undefined reference to 'trax_transport_...'`, so a misconfigured transport can never fail silently. Do not bind these from `trax_config.h`.

```c
#include <trax_transport.h>

/* Called once by trax_init(). REQUIRED. */
int trax_transport_init(void)
{
    /* Arm the transport BINDING, not the hardware itself — clocks, GPIO
     * and baud rate are typically already set up by your bsp_init().
     * Here you only enable what the trace link needs, e.g.:
     *   - UART: enable the RX path / RX interrupt or DMA
     *   - TCP:  open the listening socket
     * If there is nothing to arm, just return 0. */

    /* <enable RX path of your link> */

    return 0;                   /* non-zero aborts trax_init() */
}

/* Up-channel: send trace data to the host. REQUIRED. All-or-nothing. */
size_t trax_transport_write(const void *p_src, size_t size)
{
    /* Return exactly one of:
     *   size — every byte committed to the wire / DMA / TX ring
     *   0    — nothing accepted (TX temporarily full). Soft backpressure:
     *          the core keeps the batch in the application ring and
     *          retries it on the next trax_process() pass. No data lost.
     *
     *   if (uart_tx_free_bytes(...) < size)
     *       return 0;
     *   uart_tx_enqueue(..., p_src, size);
     *   return size;
     *
     * NEVER commit a prefix and return a partial count — that is a torn
     * frame and stops the stream with TRAX_STOP_REASON_TRANSPORT_FAIL.
     * A blocking driver that always waits until the whole batch fits is
     * allowed, but it stalls trax_process() while it waits.
     *
     * If you opt in to TRAX_CFG_TRANSPORT_TX_FREE (below), the core sizes
     * each batch to your reported free space first, so this almost never
     * has to return 0. */

    /* <send size bytes from p_src, or reject the whole call> */

    return size;
}

/* Down-channel: read host commands. REQUIRED. Non-blocking. */
```

```c
size_t trax_transport_read(void *p_dst, size_t size)
{
    /* Copy up to `size` bytes that arrived from the host into p_dst and
     * return immediately with the number actually copied (0 is valid —
     * do NOT wait for data).
     *
     * The core drains by calling this until it returns LESS than `size`,
     * so a short read is the "nothing left" signal — there is no separate
     * bytes-available hook. It therefore runs on every trax_process()
     * pass, idle ones included: keep it cheap, and put any per-poll
     * housekeeping the link needs (a non-blocking accept() on a socket
     * backend, say) right here. */

    /* <copy available RX bytes into p_dst> */

    return 0;                    /* bytes actually read */
}

/* Session restart: drop stale TX. REQUIRED. Empty body if nothing to discard.
*/
void trax_transport_clear_tx(void)
{
    /* Empty any software TX ring you own. Leave RX alone. Bytes already
     * handed to a blocking UART cannot be recalled — the host treats
     * those as pre-session noise, and this body stays empty. */
}
```

Function	Direction	Required?	Role
`trax_transport_init`	—	yes	Arm the binding; `return 0` if nothing to do
`trax_transport_write`	up	yes	All-or-nothing send: `size` or `0` (retry). Partial = fatal
`trax_transport_read`	down	yes	Non-blocking host-command read; short read = drained
`trax_transport_clear_tx`	—	yes	Drop stale TX on session restart (empty body if none)

Required functions live in `transport.c`. Optional extras stay in `trax_config.h` by naming one of your own helpers. None of the knobs below is a `trax_transport_*()` function, so leaving them out is not a link error — each falls back to the behaviour a transport had before the knob existed:

Opt-in	Signature	Default when omitted
`TRAX_CFG_TRANSPORT_TX_FREE`	`size_t(void)`	`SIZE_MAX` — free space unknown. **Recommended:** the core caps each drain batch at your reported free space BEFORE collecting frames, so `write()` never has to reject one and `trax_process()` stays cheap under backpressure. Without it the core falls back to try-and-retry, which also makes `TRAX_CFG_TX_BATCH_MAX_BYTES` load-bearing — keep that cap below your TX buffer's real capacity or a max-size batch can never fit
`TRAX_CFG_TRANSPORT_BUF_SIZE`	constant	`0` — capacity unknown. Total TX-buffer size in bytes, diagnostics only: Traxcope can then show "used / capacity" instead of a bare byte count. RTT self-reports
`TRAX_CFG_TRANSPORT_BYTES_USED`	`size_t(void)`	Derived as `BUF_SIZE - TX_FREE`, or `0` when either is unknown. Occupancy and free space are two views of one buffer, so define this only when that subtraction does not match your TX path (a link that reports occupancy but not free space, or a ring whose reserved slots make it off by a byte and you want the gauge exact)

```
/* trax_config.h — all optional, CUSTOM only */
#define TRAX_CFG_TRANSPORT_TX_FREE     my_uart_tx_free      /* size_t
my_uart_tx_free(void);     */
#define TRAX_CFG_TRANSPORT_BUF_SIZE    4096U
#define TRAX_CFG_TRANSPORT_BYTES_USED  my_uart_tx_pending   /* size_t
my_uart_tx_pending(void);  */
```

Contract highlights:

- **All four functions are required** and belong in the transport `.c`, including the often-trivial `init()` and `clear_tx()`. Leaving any undefined is a link error, never a silent no-op and never a config-macro default. `init()` stays mandatory because it is the only hook where

the RX path gets armed: defaulting it would silently kill the down-channel instead of failing the build. `clear_tx()` is the same: session restart always calls it, so write an empty body when there is nothing to discard.

- `write()` returning `0` is retryable backpressure, not a fatal error. A value other than `size` or `0` tears the stream down with `TRAX_STOP_REASON_TRANSPORT_FAIL`.
- A non-zero `init()` return aborts `trax_init()`, which returns the same value — without a transport binding there is no channel to stream on, so the diag, fault and OS layers are not brought up on top of it.
- The functions run in the TraxCtrl task (RTOS) or in the caller of `trax_process()` (bare metal) — they must never block indefinitely.
- Switching back to RTT needs no source changes: the RTT backend uses distinct `trax_transport_rtt_*()` names, so your custom transport file can stay in the build without causing duplicate-symbol errors. Guard any ISR handlers in that file — those names are not remapped.

An optional copy-paste skeleton lives in `transports/Custom/`.

On the Traxcope side, a custom UART transport pairs with the **UART** transport type (set the matching port/baud); a custom TCP transport pairs with the **TCP** type.

4. Metadata storage: ELF-only vs in-flash

Static metadata (log format strings, variable names/units/colors, marker deadlines, SM state tables, ...) is generated at compile time by the `TRAX_*_DEFINE` macros and placed in dedicated `.trax_*` linker sections. Where those sections end up is your choice:

	`TRAX_META_ELF_ONLY`	`TRAX_META_IN_FLASH` **(default)**
MCU flash cost	**Zero** — sections are placed at a synthetic address outside flash, so they exist in the `.elf` but are skipped by `objcopy`/the flasher	Metadata occupies flash
Traxcope needs the ELF file?	**Yes** — set the ELF path in probe config	Optional — metadata is also serialized on the wire at session start
Session start traffic	Smaller	Larger (static tables transmitted once)
Best for	Production firmware over UART; flash-constrained parts	Field debugging without access to build artifacts

```
/* trax_config.h */
#define TRAX_CFG_META_STORAGE   TRAX_META_ELF_ONLY     /* or TRAX_META_IN_FLASH
*/
```

Switching modes requires **no other source changes** — the `TRAX_*_DEFINE` macros automatically emit into `.trax_log` vs `.trax_log_elf` style section variants, and `linker/trax_meta.ld` defines both output placements. Traxcope's ELF reader understands both variants.

What is *never* affected by this choice: dynamic metadata (task names, queue names registered at runtime) is always sent over the wire at session start, and trace frames themselves are always runtime wire data.

Linker / ELF requirements

- Add `linker/trax_meta.ld` to your linker invocation in **both** modes. It is an *additional* script — your board's main linker script stays as it is.
- `trax_meta.ld` maps the flash-resident sections into a memory region named `FLASH`. Most vendor scripts (e.g. STM32CubeIDE) already define it; if yours names the flash region differently (`ROM`, `m_text`, ...), add one line after its `MEMORY{}` block: `REGION_ALIAS("FLASH", ROM);`. A build-time assert inside the fragment catches the misconfiguration with a clear error message.
- Keep the `.elf` of every build you ship if you use ELF-only mode — without it, Traxcope can show raw TIDs but not names or formatted log text.
- The ELF must match the running firmware. Traxcope shows the firmware's build ID (from `TRAX_CFG_BUILD_ID`) in the probe panel, which makes mismatches easy to spot.

ESP-IDF targets (ESP32 family). ESP-IDF generates its final linker script itself (via `ldgen`) and rejects sections that no script places, so the generic `.ld` fragment cannot be used there. Use the provided `linker/esp/trax_meta.lf` instead — an ldgen *linker fragment* that makes the same placements (metadata into flash rodata, same `__trax_*_start/_end` boundary symbols), so nothing changes on the C side. Register it in the component that builds the TraxProbe sources (path relative to the component directory):

```
idf_component_register(
    ...
    LDFRAGMENTS "linker/esp/trax_meta.lf"
)
```

Both metadata modes work exactly as described above; in ELF-only mode the fragment is inert and can stay registered.

5. RTOS integration (FreeRTOS)

5.1 Configuration

```
/* trax_config.h */
#define TRAX_CFG_RTOS_TYPE          TRAX_RTOS_FREERTOS
#define TRAX_CFG_FREERTOS_VERSION   TRAX_FREERTOS_VERSION(11, 2, 0)  /*
mandatory with an RTOS */
```

`TRAX_CFG_FREERTOS_VERSION` is required whenever `TRAX_CFG_RTOS_TYPE` is not `TRAX_RTOS_NONE`. Read the triple from `tskKERNEL_VERSION_NUMBER` in your kernel's `task.h` (supported range V10.2.0 … V11.2.x). Omitting it, or setting a version below V10.2.0, is a compile error.

```
/* FreeRTOSConfig.h — at the bottom, AFTER all config defines */
#define configUSE_TRACE_FACILITY  1     /* REQUIRED — enables traceXXX hooks
*/

#if (configUSE_TRACE_FACILITY == 1)
#include "trax.h"                       /* wires every kernel hook to a trace
frame */
#endif
```

That is all. Including `trax.h` from `FreeRTOSConfig.h` pulls in the FreeRTOS port header, which maps every kernel trace hook (`traceTASK_SWITCHED_IN`, `traceQUEUE_SEND`, `traceMALLOC`, ...) to a TraxProbe frame. **No application code is needed** to see tasks, context switches, queue/semaphore/mutex traffic, heap activity, and the kernel tick in Traxcope. See Chapter 13 for what gets traced and how it renders.

Recommended extras in `FreeRTOSConfig.h`:

Define	Why
`INCLUDE_uxTaskGetStackHighWaterMark 1`	Enables per-task stack headroom in Traxcope's Memory View
`INCLUDE_xTaskGetCurrentTaskHandle 1`	Used by the trace hooks
`configCHECK_FOR_STACK_OVERFLOW 2`	The library's weak `vApplicationStackOverflowHook` emits a stack-overflow event before asserting
`configQUEUE_REGISTRY_SIZE 8`	Registered queue names appear in Traxcope instead of raw handles

FreeRTOS-specific knobs in `trax_config.h` (defaults are fine for most boards):

Setting	Default	Why you would change it
`TRAX_CFG_ISR_YIELD_TO_SCHEDULER`	`1`	`1` (default) — a tick/user ISR that returns with a yield pending does **not** emit `ISR_EXIT`. Traxcope already closes the open ISR when `SWITCH_OUT` arrives, so the timeline is ISR → Scheduler → new task and you save one 16-byte kernel frame per yielding tick. `0` — emit `ISR_EXIT` and show the literal Cortex-M resume of the preempted task (typically Idle, 2–5 µs) before PendSV
`TRAX_CFG_OWN_FREERTOS_TICK_HOOK`	`1`	TraxProbe provides a strong `vApplicationTickHook` that calls `trax_freertos_on_tick()` then the weak `trax_app_tick_hook()`. CubeMX: leave **USE_TICK_HOOK unchecked** and delete any generated stub in `app_freertos.c`, or the link fails with a multiple-definition error. Put application tick work in `trax_app_tick_hook()`, not in `vApplicationTickHook`. Set `0` only if you must keep your own hook — then call `trax_freertos_on_tick()` from it
`TRAX_CFG_MAX_RTOS_TASKS`	`16`	Task-table capacity. Raise if `xTaskCreate` count can exceed 16
`TRAX_CFG_MAX_RTOS_OBJECTS`	`16`	Queue / semaphore / mutex / timer table capacity
`TRAX_CFG_RTOS_TASK_NAME_MAX`	`16`	Copied task-name length. Match `configMAX_TASK_NAME_LEN`
`TRAX_CFG_FREERTOS_TICK_ISR_PRIORITY`	lowest NVIC	SysTick lane priority in the Trace View
`TRAX_CFG_CTRL_TASK_PRIORITY` / `_STACK_SIZE` / `_PERIOD_MS`	`1` / `256` / `10`	TraxCtrl drain-task budget — see the observability contract in §5.2

5.2 Lifecycle under FreeRTOS

Under FreeRTOS, SysTick (and therefore valid timestamps) only exists after `vTaskStartScheduler()`. The internal **TraxCtrl** task that drains the ring and handles host commands is itself a FreeRTOS task, so it cannot run until the scheduler is up. That is why `trax_wait_stream_started()` must **not** be called from `main()`.

Do the wait — and any other post-scheduler init — at the top of an application task you were going to create anyway. CubeMX already creates `defaultTask`; reuse it. Do not allocate a one-shot boot task and `vTaskDelete(NULL)`: that spends a TCB plus stack only to throw them away, requires `INCLUDE_vTaskDelete`, and on `heap_1` the memory never comes back.

```
int main(void)
{
    bsp_init();      /* hardware up first — trax_init() calls
trax_transport_init() */
    trax_init();     /* creates TraxCtrl; it runs after the scheduler */

    xTaskCreate(app_task, "app", APP_STACK, NULL, APP_PRIO, NULL);
    vTaskStartScheduler();                  /* never returns */
}

static void app_task(void *arg)
{
    (void)arg;

    trax_wait_stream_started();   /* optional: block until Traxcope connects
*/

    /* other tasks, queues, streams, ISRs — CREATE events get valid timestamps
*/
    app_objects_init();

    for (;;) {
        /* real work — keep this stack */
    }
}
```

`trax_wait_stream_started()` is optional (demos use it so startup CREATE / stream-start frames are captured live). Production firmware usually starts without waiting. Creating the other tasks from `main()` before the scheduler is valid FreeRTOS; those CREATE events land at timestamp 0 and Traxcope still learns names from ELF / session metadata.

TraxCtrl (default: priority 1, 256-word stack, 10 ms period — tunable via `TRAX_CFG_CTRL_TASK_*`) drains the ring buffer and processes host commands; you never call `trax_process()` yourself.

5.3 Bare-metal lifecycle

- Call `trax_process()` periodically from the main loop (it is non-blocking).
- Call `trax_timestamp_tick()` from your SysTick (TICK_TIMER mode).
- Optionally bracket each main-loop iteration with `TRAX_MAIN_LOOP_BEGIN()` / `TRAX_MAIN_LOOP_END()` to get execution slices in the Trace View (Chapter 14).

6. Runtime control & diagnostics

API	Purpose
`trax_init()`	Initialize everything; recording starts immediately
`trax_process()`	Bare-metal drain + command processing (no-op to call extra)
`trax_start_stream()` / `trax_stop_stream()`	Programmatically enable/disable transmission (normally driven by Traxcope)
`trax_wait_stream_started()`	Block until the host requests streaming
`trax_set_buffer_alloc_fail_callback(cb)`	Immediate notification on ring-buffer overflow (ISR context — keep it short)

Diagnostics (`trax_diag.h`) — the firmware self-monitors throughput, ring-buffer usage, overflows, and corruption:

```
/* 1. Subscribe to specific events */
trax_diag_subscribe(TRAX_DIAG_EVT_OVERFLOW | TRAX_DIAG_EVT_WRITE_FAIL, my_cb);

/* 2. Threshold alerts (fires once per crossing) */
trax_diag_set_threshold(TRAX_DIAG_METRIC_RING_BUF_USED,
                        TRAX_CFG_OUT_BUFFER_SIZE32 * 4 * 75 / 100);

/* 3. Poll a consistent snapshot at any time */
struct trax_diag_stats_t s;
trax_diag_get_stats(&s);
```

A diagnostics snapshot is also emitted on the wire every second (`TRAX_CFG_DIAG_REPORT_PERIOD_MS`) — Traxcope shows it live in the probe configuration panel (throughput, buffer usage, drop counters). Data-loss events additionally produce ERROR-level log lines tagged `diag` so they are visible in the Log View.

If the firmware ever stops streaming on its own, it tells the host why: a `STREAM_STOP` frame carries a reason code (`FRAME_CORRUPT` , `TRANSPORT_FAIL` , `HEAP_EXHAUSTED` , ...) which Traxcope renders as a banner instead of leaving you guessing at the silence.

7. Buffer sizing & bandwidth budgeting

The output ring buffer is `TRAX_CFG_OUT_BUFFER_SIZE32 × 4` bytes of RAM. Rules of thumb:

- A typical frame is 12–20 bytes (3–5 words): header + timestamp + TID + 0–2 params.
- The ring only needs to absorb the **burst** between two drain passes (TraxCtrl period or `trax_process()` interval) — size it for peak burst, not average rate.
- When the ring fills up, streaming pauses and resumes with an explicit gap (see §7.1 below). No frame is ever partially delivered — overflow never corrupts data, it only loses it.
- High-traffic configurations (RTOS trace + several streams) typically use 4096–16384 words (16–64 KB).

7.1 Overflow behavior: pause, drain, resume

A full ring is a pause, never a dead session ("accordion" mode — always on, not configurable). When producers outrun the transport (bandwidth leakage), frame production is gated, the application ring drains completely, then the firmware emits a `STREAM_GAP` resync frame and resumes streaming. The result is coherent stream blocks separated by explicit gaps — never a corrupted or dead session. The gap frame carries everything Traxcope needs to stay coherent: gap start/end time anchors, the timer-overflow counter (so silent timer wraps during the pause don't skew the timebase), the dropped-frame count, and a fresh snapshot of the dynamic state (RTOS task/object tables, stream runtime state). Traxcope renders the episode as an explicit, quantified data gap — an amber `STREAM_GAP` entry in the Event View, gap counters in the probe diagnostics card, NaN regions in variable-stream plots — not an error, and everything captured before and after the gap remains valid.

Resume is the same on every transport: as soon as the application ring is empty. A still-full TX buffer is handled by the transport write contract (backpressure), not by a second pause — waiting for the UART/RTT ring to drain while producers stay gated idles the wire and invents extra gaps. Keep `trax_process()` pumping (do not `delay()` between passes) so the application ring can feed the transport at wire rate.

What a gap costs: LOG, trace and kernel-event frames produced during the pause are lost (the count is reported). What it doesn't cost: variable streams keep their sample clock — the sequence index keeps advancing while gated, so post-gap samples land at the correct time and the hole plots as NaN. There is no hot-path cost either: the successful-allocation path is untouched; the gap machinery lives entirely in the cold overflow handler and `trax_process()` .

Bandwidth example: four variable streams (~49 kB/s) plus full FreeRTOS kernel trace, ISR events and logs fit a 921600-baud UART (~92 kB/s) with headroom, using a 64 KB ring.

To reduce traffic: lower stream rates, raise `TRAX_LOG_COMPILE_LEVEL`, or disable subsystems you are not currently inspecting — every feature is independent.

8. Production builds

One switch removes everything:

```
#define TRAX_ENABLE 0          /* in trax_config.h, or -DTRAX_ENABLE=0 */
```

When disabled, all API macros (`TRAX_LOG_*`, `TRAX_VAR_*`, `TRAX_ISR_*`, `TRAX_MARKER_*`, `TRAX_SM_*`, stream macros, FreeRTOS hooks) expand to no-ops, all functions become inline stubs, and every TraxProbe translation unit compiles to nothing — zero flash, zero RAM, zero cycles. Application code does not need a single `#ifdef`.

Intermediate options:

- `TRAX_LOG_COMPILE_LEVEL` — keep the probe but strip verbose logs (no `TRAX_CFG_` prefix).
- `TRAX_FILE_LOG_LEVEL` — per-file log level override (define before including `trax.h` in a .c file).
- `TRAX_CFG_META_STORAGE TRAX_META_ELF_ONLY` — keep full functionality but pay zero flash for metadata.