

07 — Linux Applications (TraxProbe on Linux)

TraxProbe & Traxcope User Manual · Version 2.0.0 · Updated 2026-10-07

This chapter is the full guide to tracing a Linux user-space application with TraxProbe and viewing it in Traxcope. It covers x86_64 PCs, embedded Linux boards, and AMD/Xilinx PetaLinux on Zynq-7000, Zynq UltraScale+ (ZynqMP) and Versal.

TraxProbe runs **inside the traced process**, not in the kernel. It reports what the process does: its threads, locks, logs and variables. It asks the kernel, through `perf_event_open`, for what a process cannot see about itself: which task ran on each CPU, and which interrupts fired.

Short versions of this material are in Chapter 02 — Linux and VxWorks targets and Chapter 13 — Linux targets. This chapter goes further: build commands, cross-compiling, PetaLinux images, LD_PRELOAD, permissions, several processes, and troubleshooting.

To install Traxcope itself on a Linux PC, see Chapter 01 — Install Traxcope › Ubuntu. Traxcope needs nothing Linux-specific to receive a Linux trace: it connects over TCP.

1. What you get

In Traxcope	Comes from	Needs
A lane per thread of your program, by name	<code>trax_linux_thread_register()</code> , or the thread's own name	Nothing
Mutex, semaphore, condition-variable and read-write-lock events, with hold and wait times	Wrappers in front of the libc calls (link time or LD_PRELOAD)	The wrap list (§4.4) or the preload library (§5)
Logs, variables, streams, markers, state machines, device triggers	Your instrumented code — the same macros as on an MCU (Chapters 10–17)	A linked-in build (§4)
CPU lanes: which thread ran on which CPU	The scheduler collector (kernel context-switch records)	Kernel permission (§6)
Other processes and kernel threads on the machine	The same collector, system-wide	<code>perf_event_paranoid <= 0</code> or <code>CAP_PERFMON</code> (§6)
Interrupt (ISR) lanes, named from <code>/proc/interrupts</code>	The kernel's <code>irq_handler_entry</code> / <code>irq_handler_exit</code> tracepoints	<code>perf_event_paranoid = -1</code> or <code>CAP_PERFMON</code> , plus readable tracefs (§6)

Every timestamp is `CLOCK_MONOTONIC` in nanoseconds. Application frames and kernel records share this one clock, so nothing is correlated after the fact.

What a Linux recording does not contain:

Not available	Why
Heap and stack figures	The heap belongs to the C library; a thread's stack has no high-water mark to read. The Memory View shows no value rather than a guess
Ready-to-run latency (how long a woken thread waited for a CPU)	The Linux port does not send task-ready events in 2.0
Events from inside kernel drivers	TraxProbe is a user-space library. A driver shows up only through its interrupts, its kernel threads, and the application threads that wait on it
Lock calls that bypass the wrappers	A call that is inlined, statically linked, or made on a <code>futex</code> directly is not traced. The recording does not invent it

2. Requirements

Item	Requirement
Architecture	x86_64, ARM64 (aarch64: ZynqMP, Versal), 32-bit ARM hard-float (arm-linux-gnueabi: Zynq-7000). The SDK is C99 with pthreads
C library	glibc
Compiler	GCC (or a compatible compiler) with GNU ld, gold or a recent lld — the linker fragment uses INSERT
Kernel, for CPU lanes	CONFIG_PERF_EVENTS; context-switch records need Linux 4.3 or newer
Kernel, for interrupt lanes	CONFIG_FTRACE and CONFIG_TRACEPOINTS as well. PetaLinux kernels ship with all three
Permissions	None for your own threads, locks and data. CPU, other-process and interrupt lanes need the settings in §6
Network	The target must be reachable from the Traxcope PC on one TCP port per traced process (default 4242)
Traxcope	2.0.0 or newer, on Windows or Linux. Process lanes need a 2.0 host together with a 2.0 SDK

3. Choose an integration

	Linked in (§4)	LD_PRELOAD (§5)
Rebuild the application	Yes	No
Threads and locks	Yes	Yes
CPU, process and interrupt lanes	Yes	Yes
Logs, variables, streams, markers, state machines	Yes	No — the program contains no TraxProbe calls
Named locks	Yes (trax_linux_object_name())	No — locks show under their address
Locks inside other shared libraries	No — --wrap binds only what you link	Yes — every call through the dynamic linker
Statically linked programs	Yes	No — the dynamic linker never loads the library

Use the linked-in build for your own code. Use LD_PRELOAD for a program you cannot rebuild: a vendor daemon, a binary from another team, or a quick first look.

4. Build TraxProbe into your application

4.1 Copy the SDK

Copy the TraxProbe SDK from your Traxcope install into your source tree (paths in Chapter 02 — Where the SDK comes from). This chapter assumes:

```

my_app/
  App/trax_config.h      your configuration (§4.2)
  main.c
  Third_Party/TraxProbe/ the SDK, unchanged

```

There is no Linux template in config/templates/. Use the file in §4.2.

4.2 trax_config.h

```
#ifndef APP_TRAX_CONFIG_H_
#define APP_TRAX_CONFIG_H_

#include "trax_tid.h"

/* Platform: Linux user space, built-in TCP server */
#define TRAX_CFG_HW_PORT          TRAX_HW_PORT_LINUX
#define TRAX_CFG_RTOS_TYPE        TRAX_RTOS_LINUX
#define TRAX_CFG_TRANSPORT        TRAX_TRANSPORT_TCP
#define TRAX_CFG_TCP_PORT         4242
#define TRAX_CFG_PROJECT_NAME     "my_app"

/* Clock: CLOCK_MONOTONIC in nanoseconds (both lines are required) */
#define TRAX_CFG_TIMESTAMP_MODE    TRAX_TIMESTAMP_FREERUN
#define TRAX_CFG_TIMER_FREQ_HZ    1000000000U

/* Buffers: one ring per traced thread, TraxProbe's own two included */
#define TRAX_CFG_RING_COUNT        12U
#define TRAX_CFG_OUT_BUFFER_SIZE32 49152U /* 192 KB = 16 KB per ring */

/* Task table: every thread the scheduler collector sees (255 at most) */
#define TRAX_CFG_MAX_RTOS_TASKS    250
#define TRAX_CFG_MAX_RTOS_OBJECTS  64

/* Your TIDs */
enum { TID_LOG_APP = TRAX_TID_RANGE_LOG_USER_START };
enum { TID_VAR_LOOP_COUNT = TRAX_TID_RANGE_VAR_USER_START };

#endif
```

What the Linux-specific lines mean:

Setting	Linux default	Notes
TRAX_CFG_HW_PORT / TRAX_CFG_RTOS_TYPE	—	TRAX_HW_PORT_LINUX and TRAX_RTOS_LINUX, always together
TRAX_CFG_TRANSPORT	—	TRAX_TRANSPORT_TCP: the SDK's TCP server (Chapter 02 §3.3)
TRAX_CFG_TIMESTAMP_MODE	—	Must be TRAX_TIMESTAMP_FREERUN; any other mode stops the build with #error
TRAX_CFG_TIMER_FREQ_HZ	—	1000000000U. The port reads CLOCK_MONOTONIC in ns; the low 32 bits wrap every 4.295 s and Traxcope rebuilds the full time
TRAX_CFG_RING_COUNT	8	Threads traced at the same time , not CPU cores. Range 1–16. TraxProbe's own threads tTraxCtrl and tTraxSched take two. A ring is reused after its thread exits
TRAX_CFG_OUT_BUFFER_SIZE32	2048 (8 KB)	In 32-bit words, split evenly between the rings. The default is too small for Linux; 16 KB per ring is the SDK's recommendation
TRAX_CFG_MAX_RTOS_TASKS	16	Entries for every thread Traxcope names, including other programs' threads when the collector is system-wide. Keep it above TRAX_CFG_LINUX_SCHED_MAX_FOREIGN (200). Ceiling 255
TRAX_CFG_META_STORAGE	TRAX_META_IN_FL ASH	Keep the default on Linux: the metadata is sent at session start and Traxcope needs no ELF

All other options (logs, variables, markers, buffers) work as in Chapter 02 §2. The Linux-only options are listed in §11.

4.3 Initialize and register threads

```
#include <pthread.h>
#include <unistd.h>
#include "trax.h"
#include "trax_rtos_port.h"

TRAX_VAR_DEFINE(TID_VAR_LOOP_COUNT, "Loop Count", TRAX_DATA_TYPE_UINT32,
                0, 1000, "", TRAX_COLOR_GREEN, TRAX_PLOT_LINE);

static pthread_mutex_t s_lock = PTHREAD_MUTEX_INITIALIZER;

static void *worker(void *arg)
{
    (void)arg;
    trax_linux_thread_register("worker"); /* names this thread's lane */
    for (uint32_t n = 0; n < 500U; n++) {
        pthread_mutex_lock(&s_lock); /* traced through the wrapper */
        TRAX_VAR_SET(TID_VAR_LOOP_COUNT, n);
        pthread_mutex_unlock(&s_lock);
        usleep(10000);
    }
    trax_linux_thread_unregister(); /* closes the lane */
    return NULL;
}

int main(void)
{
    if (trax_init() != 0) { /* starts tTraxCtrl and tTraxSched */
        return 1;          /* most likely: the TCP port is taken */
    }
    trax_linux_thread_register("main");
    trax_linux_object_name(&s_lock, "loopLock");
    TRAX_LOG_INFO(TID_LOG_APP, "app", "started, pid %d", (int)getpid());

    pthread_t t;
    pthread_create(&t, NULL, worker, NULL);
    pthread_join(t, NULL);
    return 0;
}
```

Rules:

1. Call `trax_init()` early in `main()`, **before you create threads**. It opens the TCP port and starts two threads: `tTraxCtrl` (drains the rings every `TRAX_CFG_CTRL_TASK_PERIOD_MS`, default 10 ms; answers Traxcope) and `tTraxSched` (the scheduler collector). There is no `trax_task()` to start yourself.
2. `trax_init()` returns non-zero when the transport cannot start — in practice, when another process holds the TCP port.
3. Call `trax_linux_thread_register("name")` at the top of each thread you care about, and `trax_linux_thread_unregister()` before it returns.
 - `NULL` as the name uses the name set with `pthread_setname_np()`.
 - A thread that never registers is still traced. It appears under its kernel thread id.
 - For threads other than `main`, the name is also given to the kernel (cut to 15 characters), so `top`, `ps -L` and `perf` show the same name. The `main` thread is never renamed, because its name is the process name.
4. Give locks names with `trax_linux_object_name(&obj, "name")`. Without a name, a lock shows under its address. You may call it before or after the object is first used, from any thread.
5. There is no shutdown call. When the process exits, the TCP connection closes.

- Do not call TraxProbe from a signal handler. A frame started in a handler that interrupted another frame on the same thread is dropped and counted.

Threads and their priorities: SCHED_FIFO / SCHED_RR priorities (1–99) are reported as they are; a SCHED_OTHER thread is reported as 0. A larger number is more urgent, and Traxcope orders the thread lanes by it.

4.4 Build with GCC

The SDK sources to compile:

Folder	Files
src/, debug/	All *.c
os/common/, os/Linux/	All *.c
hw_port/Linux/src/	All *.c
transports/TCP/	trax_transport_tcp.c

Flags you must pass:

Flag	Why
-D_GNU_SOURCE	Required; the port stops the build with #error without it
-pthread	Threads
-ftls-model=initial-exec	Each thread's ring becomes an offset from the thread pointer instead of a function call on every frame
-Wl,-T,<sdk>/linker/linux/trax_linux.ld	Collects TraxProbe's metadata sections. It is added with INSERT, so your toolchain's own linker script stays in place. Do not use linker/trax_probe.ld on Linux
The full --wrap list below	Routes the pthread and semaphore calls through TraxProbe's wrappers
-Wl,--build-id	Lets Traxcope check that the program matches the ELF you give it. Most distribution toolchains pass it already

A complete build:

```

TRAXPROBE=Third_Party/TraxProbe
TRAX_WRAP="pthread_mutex_lock pthread_mutex_trylock pthread_mutex_timedlock \
pthread_mutex_unlock pthread_mutex_destroy \
sem_wait sem_trywait sem_timedwait sem_post sem_destroy \
pthread_cond_wait pthread_cond_timedwait \
pthread_cond_signal pthread_cond_broadcast pthread_cond_destroy \
__sem_timedwait64 __pthread_cond_timedwait64 __pthread_mutex_timedlock64 \
pthread_rwlock_rdlock pthread_rwlock_tryrdlock \
pthread_rwlock_wrlock pthread_rwlock_trywrlock \
pthread_rwlock_timedrdlock pthread_rwlock_timedwrlock \
pthread_rwlock_unlock pthread_rwlock_destroy \
__pthread_rwlock_timedrdlock64 __pthread_rwlock_timedwrlock64"

gcc -O2 -g -pthread -D_GNU_SOURCE -ftls-model=initial-exec \
-IApp -I$TRAXPROBE/inc -I$TRAXPROBE/config \
$TRAXPROBE/src/*.c $TRAXPROBE/debug/*.c $TRAXPROBE/os/common/*.c \
$TRAXPROBE/os/Linux/*.c $TRAXPROBE/hw_port/Linux/src/*.c \
$TRAXPROBE/transports/TCP/trax_transport_tcp.c \
main.c \
-Wl,--build-id -Wl,-T,$TRAXPROBE/linker/linux/trax_linux.ld \
$(for s in $TRAX_WRAP; do printf -- '-Wl,--wrap=%s ' "$s"; done) \
-o my_app -lm -lrt

```

- IApp is the folder that holds your trax_config.h. TraxProbe includes it by name.
- Pass the whole wrap list, every time.** os/Linux/trax_linux_sync.c refers to the real function behind every wrapper; a partial list fails to link with undefined reference to '__real_...'.

- The three `__*64` names (and the two `rwlock` ones) are what a 32-bit glibc with 64-bit `time_t` calls the timed waits. On Zynq-7000 (32-bit) they are required: without them every timed wait is silently untraced. On 64-bit targets they cost nothing. Keep them everywhere.
- Only calls in the code you link are wrapped. A lock taken inside another shared library is not traced in this mode.

4.5 CMake

The same sources and flags in CMake (3.13 or newer):

```
set(TRAXPROBE ${CMAKE_CURRENT_SOURCE_DIR}/Third_Party/TraxProbe)

file(GLOB TRAXPROBE_SRC
    ${TRAXPROBE}/src/*.c ${TRAXPROBE}/debug/*.c
    ${TRAXPROBE}/os/common/*.c ${TRAXPROBE}/os/Linux/*.c
    ${TRAXPROBE}/hw_port/Linux/src/*.c)
list(APPEND TRAXPROBE_SRC ${TRAXPROBE}/transports/TCP/trax_transport_tcp.c)

set(TRAX_WRAP
    pthread_mutex_lock pthread_mutex_trylock pthread_mutex_timedlock
    pthread_mutex_unlock pthread_mutex_destroy
    sem_wait sem_trywait sem_timedwait sem_post sem_destroy
    pthread_cond_wait pthread_cond_timedwait
    pthread_cond_signal pthread_cond_broadcast pthread_cond_destroy
    __sem_timedwait64 __pthread_cond_timedwait64 __pthread_mutex_timedlock64
    pthread_rwlock_rdlock pthread_rwlock_tryrdlock
    pthread_rwlock_wrlock pthread_rwlock_trywrlock
    pthread_rwlock_timedrdlock pthread_rwlock_timedwrlock
    pthread_rwlock_unlock pthread_rwlock_destroy
    __pthread_rwlock_timedrdlock64 __pthread_rwlock_timedwrlock64)
list(TRANSFORM TRAX_WRAP PREPEND "-Wl,--wrap=")

find_package(Threads REQUIRED)

add_executable(my_app main.c ${TRAXPROBE_SRC})
target_include_directories(my_app PRIVATE App ${TRAXPROBE}/inc ${TRAXPROBE}/config)
target_compile_definitions(my_app PRIVATE _GNU_SOURCE)
target_compile_options(my_app PRIVATE -ftls-model=initial-exec)
target_link_options(my_app PRIVATE
    -Wl,--build-id "-Wl,-T,${TRAXPROBE}/linker/linux/trax_linux.ld" ${TRAX_WRAP})
target_link_libraries(my_app PRIVATE Threads::Threads m rt)
```

4.6 Cross-compile for ARM (Zynq, ZynqMP, Versal)

Nothing in the SDK needs special help to cross-compile. Use the same command with your cross compiler:

Target	Compiler (Debian/Ubuntu packages)
ZynqMP, Versal (ARM64)	aarch64-linux-gnu-gcc
Zynq-7000 (32-bit ARM)	arm-linux-gnueabi-gcc

For a PetaLinux or Yocto image, build against the image's own SDK so the binary matches the board's glibc:

1. In your PetaLinux project, build the SDK: `petalinux-build --sdk`. Install it with the `sdk.sh` it produces.
2. In each shell: `source <sdk-install-dir>/environment-setup-*`. This sets `$CC` with the right `--sysroot` and CPU flags.
3. Run the build from §4.4 with `$CC` (unquoted) in place of `gcc`.
4. Copy the binary to the board (`scp my_app root@<board-ip>:/usr/bin/`).

A binary built with a distribution cross compiler and run on an older board glibc may fail to start. In that case build against the image's SDK, or link statically (`-static`; the wrap list still applies).

4.7 PetaLinux / Yocto: put it in the image

Image settings for full scheduler and interrupt lanes (details in §6). Install a sysctl file:

```
# recipes-apps/my-app/files/99-trax.conf -> /etc/sysctl.d/99-trax.conf
kernel.perf_event_paranoid = -1
```

A recipe, in outline, for an application with its own Makefile that uses \$(CC), \$(CFLAGS) and \$(LDFLAGS) and the sources and flags of §4.4:

```
SUMMARY = "my_app with TraxProbe"
LICENSE = "CLOSED"
SRC_URI = "file://app/ file://traxprobe/ file://99-trax.conf"
S = "${WORKDIR}/app"

EXTRA_OEMAKE = "CC='${CC}' TRAXPROBE='${WORKDIR}/traxprobe'"

do_install() {
    install -D -m 0755 ${B}/my_app ${D}${bindir}/my_app
    install -D -m 0644 ${WORKDIR}/99-trax.conf ${D}${sysconfdir}/sysctl.d/99-trax.conf
}

# Instead of the sysctl file, to give only this program the right:
# pkg_postinst_ontarget:${PN}() { setcap cap_perfmon+ep ${bindir}/my_app; }
```

- Add the recipe to your meta-user layer and to the image.
- Yocto checks that your Makefile passes \$(LDFLAGS) to the link. If it does not, the build stops with a QA error.
- The image installs a **stripped** binary. If you give Traxcope an ELF (optional, §8.1), use the unstripped one from the recipe's build directory, not the copy from the board.

4.8 Production builds

TRAX_ENABLE 0 removes TraxProbe, as on an MCU (Chapter 02 §8). On Linux, two more things in 2.0:

1. Remove the --wrap flags from the link. With TraxProbe disabled the wrappers do not exist, and the link fails with undefined reference to '__wrap_...'.
2. Guard the Linux port calls (trax_linux_thread_register(), trax_linux_thread_unregister(), trax_linux_object_name()) with #if TRAX_ENABLE. They have no disabled stubs in 2.0, so they fail to link otherwise.

A build that keeps TraxProbe in but has no host connected costs little: with no client on the port, the transport discards data instead of filling the rings.

5. Trace a program you cannot rebuild (LD_PRELOAD)

The same tracing, bound by the dynamic linker instead of the static one. The program is not rebuilt, relinked or changed.

5.1 Build the preload library

The Traxcope installer does not ship a prebuilt library; build it once per target architecture from the SDK.

1. Create Preload/trax_config.h:

```
#ifndef PRELOAD_TRAX_CONFIG_H_
#define PRELOAD_TRAX_CONFIG_H_

#include "trax_tid.h"

#define TRAX_CFG_HW_PORT          TRAX_HW_PORT_LINUX
#define TRAX_CFG_RTOS_TYPE        TRAX_RTOS_LINUX
#define TRAX_CFG_TRANSPORT        TRAX_TRANSPORT_TCP
#define TRAX_CFG_META_STORAGE     TRAX_META_IN_FLASH
#define TRAX_CFG_PROJECT_NAME     "Linux (preload)"

/* The wrappers take the libc names and find the real calls with dlsym. */
#define TRAX_CFG_LINUX_PRELOAD    1

/* The port is read from TRAX_PORT when the library loads. */
extern uint16_t trax_linux_tcp_port;
#define TRAX_CFG_TCP_PORT         trax_linux_tcp_port

#define TRAX_CFG_CORE_COUNT        1      /* fallback; read at run time */

#define TRAX_CFG_RING_COUNT        16U
#define TRAX_CFG_OUT_BUFFER_SIZE32 32768U /* 128 KB, 8 KB per ring */
#define TRAX_CFG_MAX_RTOS_TASKS    64
#define TRAX_CFG_MAX_RTOS_OBJECTS  200

#define TRAX_CFG_TIMESTAMP_MODE    TRAX_TIMESTAMP_FREERUN
#define TRAX_CFG_TIMER_FREQ_HZ     100000000U

#endif
```

TRAX_CFG_MAX_RTOS_TASKS 64 suits a look at the program itself. For system-wide lanes on a busy machine, raise it to 250 as in §4.2.

2. Build the shared object (no --wrap here):

```
TRAXPROBE=Third_Party/TraxProbe
gcc -O2 -g -pthread -D_GNU_SOURCE -ftls-model=initial-exec -fPIC -shared \
  -IPreload -I$TRAXPROBE/inc -I$TRAXPROBE/config \
  $TRAXPROBE/src/*.c $TRAXPROBE/debug/*.c $TRAXPROBE/os/common/*.c \
  $TRAXPROBE/os/Linux/*.c $TRAXPROBE/hw_port/Linux/src/*.c \
  $TRAXPROBE/transport/TCP/trax_transport_tcp.c \
  -Wl,-T,$TRAXPROBE/linker/linux/trax_linux.ld \
  -o libtraxprobe_preload.so -ldl
```

For a board, use the cross compiler or \$CC from §4.6.

5.2 Run a program under it

```
TRAX_PORT=4242 TRAX_PROGRAM=my_app \
LD_PRELOAD=/usr/lib/libtraxprobe_preload.so /usr/bin/my_app
```

Environment variable	Meaning	Default
TRAX_PORT	TCP port to listen on	4242
TRAX_PROGRAM	Trace only the process with this name; every other process that loads the library stays untouched	any
TRAX_ENABLE	0 loads the library and does nothing	1

Always set TRAX_PROGRAM when anything stands in front of the program — a shell script, timeout, stdbuf, make, start-stop-daemon. LD_PRELOAD is inherited, so the wrapper process loads the library too and takes the port first. Your program then runs untraced and prints on stderr:

```
[TraxProbe] my_app: could not listen on TCP 4242; running untraced. Another process may hold the port
-- LD_PRELOAD is inherited, so set TRAX_PROGRAM to the program you mean.
```

TRAX_PROGRAM is compared with the process's short name (program_invocation_short_name), the file name it was started as.

5.3 What it shows, and the limits

- Every thread from its first instruction, under the name the program gives it. A thread named later with pthread_setname_np() is renamed in Traxcope.
- Mutexes, semaphores, condition variables and read-write locks, under their addresses. That includes calls made inside other shared libraries.
- CPU, process and interrupt lanes, exactly as in a linked-in build (permissions in §6).
- The probe's firmware info shows the project name Linux (preload).
- No logs, variables, streams, markers or state machines.
- Not traced: statically linked programs, calls the compiler inlined, and futexes the program uses directly.
- Do not combine LD_PRELOAD with file capabilities (setcap) on the same program. The kernel runs such a program in secure-execution mode, and the dynamic linker then ignores an LD_PRELOAD path. Use the sysctl (§6) or run as root instead.

6. Scheduler and interrupt lanes: permissions

The scheduler collector (tTraxSched) starts with trax_init() (TRAX_CFG_LINUX_SCHED, default 1). It asks the kernel for context-switch records on every CPU. If the kernel refuses, it falls back to the process's own threads. If that is refused too, it writes a warning into the recording.

Kernel setting	What Traxcope draws
kernel.perf_event_paranoid <= 2 (the upstream default)	Your process's own threads switching on and off the CPUs. Other processes do not appear, and nothing shows what ran in the gaps
kernel.perf_event_paranoid <= 0, or CAP_PERFMON on the program, or running as root	Every task on the machine: your threads, other processes, kernel threads
The row above and perf_event_paranoid = -1 or CAP_PERFMON or root, and readable tracefs	The above plus interrupt (ISR) lanes
Neither	Empty CPU lanes, and a WARNING log scheduler not visible: perf denied (errno N)

Notes:

- Only the "neither" case writes a log. In the own-threads fallback, the recording simply has no other processes. If you expect them and do not see them, check the setting.
- In the own-threads fallback, only threads created **after** trax_init() are followed. Call it before you start threads.
- Interrupt lanes are all-or-nothing: if the tracepoints cannot be opened on every CPU, none are shown.

Check the current setting:

```
cat /proc/sys/kernel/perf_event_paranoid
```

Change it until the next reboot:

```
sudo sysctl -w kernel.perf_event_paranoid=-1
```

Change it permanently: put kernel.perf_event_paranoid = -1 in /etc/sysctl.d/99-trax.conf (§4.7 for an image).

Or grant the capability to one program only, leaving the system policy alone:

```
sudo setcap cap_perfmon+ep /usr/bin/my_app
```

CAP_PERFMON needs Linux 5.8 or newer; older kernels need CAP_SYS_ADMIN. Copying or rebuilding the binary drops the capability; run setcap again after every install.

Interrupt lanes also need tracefs. TraxProbe reads the tracepoint ids and record layout from /sys/kernel/tracing (or /sys/kernel/debug/tracing), which is root-only by default. Either run the program as root (common on embedded images), or relax the mount:

```
mount -o remount,mode=755 /sys/kernel/tracing
chmod -R a+rX /sys/kernel/tracing/events/irq
```

What to expect from interrupts:

- Interrupts are named from /proc/interrupts at start-up (arch_timer, uart-pl011, xhci_hcd, ...).
- On an ARM GIC (Zynq, ZynqMP, Versal) almost every interrupt, including the timer, goes through the traced path. Expect over a thousand interrupts per second on a quad-core ARM64.
- On x86 the local timer and inter-processor interrupts bypass it, so a PC shows only device interrupts — tens per second.
- Up to TRAX_CFG_LINUX_SCHED_MAX_IRQS (48) interrupts get lanes; those that have already fired at start-up are numbered first.

To turn the collector off and trace only your own process: #define TRAX_CFG_LINUX_SCHED 0.

7. Several processes

Each traced process is its own TraxProbe instance with its own TCP server. Traxcope sees each one as a separate probe.

1. Give each process its own port: TRAX_CFG_TCP_PORT in a linked-in build, TRAX_PORT under LD_PRELOAD.
2. Let **only one** of them run the system-wide scheduler collector. Build the others with #define TRAX_CFG_LINUX_SCHED 0. Two system-wide collectors record every CPU event twice. (The preload library has no switch for this at run time; build a second copy with TRAX_CFG_LINUX_SCHED 0 if you need one.)
3. In Traxcope, add one TCP probe per port (§8.1) and connect each.

All processes on one machine stamp with the same CLOCK_MONOTONIC, so no sync wire (Chapter 18) is needed. Lining several Linux probes up on one time axis has not been verified for 2.0; compare times across probes with care.

The Community Edition connects **one probe at a time**, so it traces one process at a time. The one process with system-wide lanes still shows every other process's threads in its CPU and process lanes.

8. Connect Traxcope

8.1 Create a TCP probe

1. Start the traced program on the target. It listens on all network interfaces on its TCP port.
2. In Traxcope, open the **Probes** panel and click **Add** (see Chapter 01 — Step 5).
3. **General** tab: give the probe a name. Leave **ELF File** empty.
 - A linked-in build with the default TRAX_META_IN_FLASH sends all names, units and formats at session start.
 - If you do set **ELF File** (for the build-identity check), choose the **unstripped** executable from your build directory. Linux executables have no file extension: pick **All Files (*)** in the file dialog.
 - For an LD_PRELOAD target, leave it empty.
4. **Transport** tab:
 - **Type**: TCP
 - **IP Address** *: the target's IPv4 address (for example 192.168.1.10). Host names are not accepted.

- **Port *:** 4242, or the port you configured. The field starts at 8080; change it.
- **Timeout:** 5 sec is fine.

5. Click **Test**, then save the probe.

Network notes:

- **Firewall:** open the port on the target (for example `sudo ufw allow 4242/tcp` on Ubuntu).
- **WSL2:** a program running in WSL2 on the same Windows PC is reached at 127.0.0.1.
- **One host at a time.** A second Traxcope cannot attach to the same process.
- **No authentication.** Anyone who can reach the port can read the trace. Use it on a trusted network only.

8.2 Connect, record, stream

1. Click **Connect**. The firmware info card shows the project name (TRAX_CFG_PROJECT_NAME).
2. Click **• REC** if you want to keep the whole session (Standard and Professional).
3. Click **Start Stream**.

Start the recording **before** you start streaming, or as soon as possible. A system-wide Linux trace produces thousands of events per second (about 10,000 per second on a 4-core PC running a small demo program). Without a recording, Traxcope keeps only the newest events in memory, and the Trace View marks older stretches with a hatched band: *events lost — not recording, only the newest stay in memory*.

If the program restarts, click **Disconnect**, **Connect**, then **Start Stream** again. A host that connects to a running program always starts a fresh session; nothing from before the connection is sent.

Details of the probe card: Chapter 03 §2.2.

8.3 Trace View

The Trace View works as described in Chapter 13 — In Traxcope. On Linux the lanes are, from top to bottom:

Lanes	Contents
CPU lanes	One per CPU (the count is read from the target at run time). Each shows which thread ran
ISR lanes	One per interrupt, named from <code>/proc/interrupts</code>
Your process	Header lane <code>pid · name</code> , bold and expanded , with a lane per thread under it. Thread lanes are ordered by priority; the lane head shows it (P80)
Other processes	One header lane each, <code>pid · name</code> , collapsed , sorted by name. A collapsed header shows the combined activity of its threads
Single-thread programs	A plain lane, no header
Programs that ran several times and ended	One row, <code>name — N runs</code> (for example <code>sleep — 809 runs</code>)
Kernel threads	One collapsed group for all kernel threads (kworker, ksoftirqd, ...)

- Click a process header's name to expand or collapse it.
- Right-click → **Hide Lanes Without Activity** removes interrupt and thread lanes that never fired.
- TraxProbe's own threads `tTraxCtrl` and `tTraxSched` appear inside your process. Their CPU time is the probe's cost.
- With `TRAX_CFG_LINUX_SCHED_PER_THREAD 0` the target folds other programs' threads into one lane per program before sending; use it only for an older host that cannot group.

Lock events are drawn on the thread lanes and listed in the Event View:

Event	Meaning
MTX_TAKE_BLOCK, then MUTEX_TAKE	The mutex was busy; the thread waited, then got it
MUTEX_TAKE alone	Uncontended take
MUTEX_TAKE_RECURSIVE / MUTEX_GIVE_RECURSIVE	A recursive mutex taken again / released one level
MUTEX_TAKE_FAILED	trylock found it busy, or timedlock expired
MUTEX_GIVE	Release, recorded just before the unlock
SEM_TAKE / SEM_GIVE	sem_wait / sem_post, with the count after the operation
COND_WAIT_BLOCK ... COND_WAIT_END (or COND_WAIT_TIMEOUT)	A condition-variable wait. The mutex is released before and retaken after, as separate events, so the wait is not counted as hold time. The end row shows "Waited N µs"
COND_SIGNAL / COND_BROADCAST	Recorded even if nobody waits — a signal with no wait ending after it is a lost wakeup
RW_RDLOCK, RW_WRLOCK, RW_LOCK_BLOCK, RW_LOCK_FAILED, RW_UNLOCK	Read-write locks; each reader's hold is timed on its own

8.4 Other views

View	On Linux
Log View, Scope, Stream views, Marker Stats, State Machine view, Trigger Studio	Same as on an MCU. Markers pair start and stop by thread, so a marker started on one CPU and stopped on another is measured correctly
Event View	All events above, plus thread create/delete
Memory View	No heap or stack values (§1)

9. Sizing and cost

Rings. TRAX_CFG_RING_COUNT is the number of threads that can write at the same time, plus two for TraxProbe. A thread that finds every ring taken writes into a sink: its frames are dropped, counted, and reported to Traxcope as a session gap. Raise the ring count (16 at most) or accept it.

Buffer. System-wide scheduler tracing writes 16-byte frames for every context switch and interrupt. Rates measured on the same demo program:

	Interrupts/s	Context switches/s
ARM64, GIC, 4 cores (QEMU)	1,373	1,014
x86_64, APIC, 8 cores	49	17,598

16 KB per ring holds a scheduling delay's worth comfortably. When the collector's ring is three-quarters full it pauses and leaves the rest in the kernel's buffers, so a small ring costs latency, not data. A *Session gap* event in Traxcope means a ring did overflow: raise TRAX_CFG_OUT_BUFFER_SIZE32.

Cost. Measured on an idle 8-core x86 PC with system-wide tracing streaming:

	Probe idle	Streaming
Traced process CPU	6.0 % of one core	7.6 % of one core
Whole machine busy	4.4 %	5.0 %
Recording rate	—	about 100 KB/s idle; ten times that on a busy machine

Cost of one log line (median per call, WSL2, 4 CPUs): TRAX_LOG_INFO 0.21 μ s; fprintf 0.74 μ s; fprintf + fflush 2.35 μ s; syslog 11.8 μ s. TraxProbe does not format text on the target and takes no lock or system call on the frame path.

Expect a larger share on a board with fewer cores, and a larger recording on ARM because of the interrupt rate. Measure on your target.

Lowering the load:

- Turn the collector off (TRAX_CFG_LINUX_SCHED 0) when the question is only about your own process.
- Leave perf_event_paranoid at 2: you get only your own threads' switches, far fewer events.
- Hide unneeded lanes in Traxcope; this does not reduce traffic, only clutter.

10. Troubleshooting

Symptom	Cause	Fix
Connect fails, connection refused	Program not running, wrong IP or port, or a firewall	Check <code>ss -ltnp \ grep 4242</code> on the target; open the port; check the probe's Port (it starts at 8080)
<code>trax_init()</code> returns non-zero	The TCP port is taken, often by an earlier instance still running	Stop the other process, or use another <code>TRAX_CFG_TCP_PORT</code>
Under <code>LD_PRELOAD</code> : "could not listen on TCP ...; running untraced"	A wrapper process inherited <code>LD_PRELOAD</code> and took the port	Set <code>TRAX_PROGRAM</code> to your program's name
Under <code>LD_PRELOAD</code> : nothing happens, no listener	The program is statically linked, or has file capabilities (secure-execution mode ignores the path)	Use a linked-in build; use the <code>sysctl</code> instead of <code>setcap</code>
CPU lanes empty; WARNING scheduler not visible: perf denied	<code>perf_event_paranoid</code> too high and no <code>CAP_PERFMON</code>	§6
CPU lanes show only your threads; no other processes	The collector fell back to your own process (<code>perf_event_paranoid</code> is 1 or 2)	Set it to 0 or below, or grant <code>CAP_PERFMON</code>
CPU lanes full, no ISR lanes	<code>tracefs</code> not readable, or <code>paranoid</code> above -1 without <code>CAP_PERFMON</code>	§6, <i>Interrupt lanes also need tracefs</i>
Few ISR lanes on a PC	Normal on x86: timer and IPIs bypass the traced path	—
Lock events never appear	The calls are not linked through the wrappers	Pass the full wrap list (§4.4), or use <code>LD_PRELOAD</code>
Timed waits missing on Zynq-7000 (32-bit)	The <code>__*64</code> names are missing from the wrap list	Use the full list in §4.4
Link error undefined reference to <code>'__real_...'</code>	Partial wrap list	Pass the full list
Link error undefined reference to <code>'__wrap_...'</code> or <code>trax_linux_thread_register</code> with <code>TRAX_ENABLE 0</code>	Disabled build still linked with <code>--wrap</code> , or Linux calls not guarded	§4.8
Threads shown as <code>tid N</code>	The thread never registered and had no name	Call <code>trax_linux_thread_register()</code> or <code>pthread_setname_np()</code>
Lanes named by number although threads have names	More threads than <code>TRAX_CFG_MAX_RTOS_TASKS</code>	Raise it (255 at most)
<i>Session gap</i> events, gaps in the timeline	A ring overflowed, or more threads than rings	Raise <code>TRAX_CFG_OUT_BUFFER_SIZE32</code> or <code>TRAX_CFG_RING_COUNT</code>
WARNING records up to <code>N ms</code> late; holding back	The collector was kept off the CPU and events reach Traxcope late	Harmless: no events are lost, they only reach Traxcope later. If it repeats, the target has too little CPU headroom
Hatched <i>events lost</i> — <i>not recording</i> band	Live view without a recording keeps only the newest events	Press REC before Start Stream
ELF error on a PetaLinux binary	The ELF is the stripped copy installed on the board	Give the unstripped ELF from the build directory, or leave ELF File empty
Every CPU event appears twice	Two traced processes both run the system-wide collector	§7: <code>TRAX_CFG_LINUX_SCHED 0</code> in all but one
Build stops: compile with <code>-D_GNU_SOURCE</code>	Flag missing	Add <code>-D_GNU_SOURCE</code>
Build stops: set <code>TRAX_CFG_TIMESTAMP_MODE</code> to <code>TRAX_TIMESTAMP_FREERUN</code>	Default tick-timer mode	Add the two clock lines from §4.2

11. Linux configuration reference

Options for `trax_config.h`, in addition to those in §4.2 and Chapter 02.

Option	Default	Meaning
TRAX_CFG_TCP_PORT	4242	Listen port
TRAX_CFG_TCP_SEND_TIMEOUT_MS	2000	Drop the host if it accepts no data for this long
TRAX_CFG_TCP_SNDBUF	262144	Socket send buffer in bytes; 0 = OS default
TRAX_CFG_CTRL_TASK_PERIOD_MS	10	How often tTraxCtrl drains the rings
TRAX_CFG_LINUX_SCHED	1	Scheduler and interrupt collector on/off
TRAX_CFG_LINUX_SCHED_PER_THREAD	1	1: every thread of every program is sent separately, with its process; Traxcope groups them. 0: other programs' threads are folded into one lane per program on the target
TRAX_CFG_LINUX_SCHED_PAGES	8	Kernel buffer pages per CPU channel (power of two)
TRAX_CFG_LINUX_SCHED_PERIOD_MS	10	How often the collector reads the kernel buffers
TRAX_CFG_LINUX_SCHED_MAX_CPUS	64	CPUs the collector listens to
TRAX_CFG_LINUX_SCHED_MAX_FOREIGN	200	Other threads alive at once whose process is remembered; keep it below TRAX_CFG_MAX_RTOS_TASKS
TRAX_CFG_LINUX_SCHED_MAX_IRQS	48	Interrupts that get a lane
TRAX_CFG_LINUX_SYNC_OBJECTS	256	Locks whose announcement is remembered; past it, locks are announced again on use
TRAX_CFG_LINUX_PRELOAD	0	1 only in the LD_PRELOAD library build (§5.1)

12. Licensing

Linux tracing works with every Traxcope license. The usual tier limits apply (Chapter 04 §2). In the Community Edition:

- Live streaming for 3 minutes per rolling hour.
- **One connected probe at a time**, so one traced process at a time (§7).
- No recording, so a long system-wide session keeps only its newest events in memory (§8.2).

Standard and Professional have none of these limits.