Powered by
Qt Group

TECHNICAL ARTICLE

INSIGHTS

The Bug That Vanishes When You Look

Debugging RTOS Timing on Arm Cortex-M Without Stopping the MCU

By Rafael Taubinger
Industry Insight Lead &
Technical Advisor, IAR

Introduction

Your firmware passes every bench test, then misses a deadline on the test rig once every few hours. Connect a debugger and the fault disappears. This article explains why timing bugs hide from breakpoints, how the trace hardware built into most Cortex-M devices reveals what is happening in real time, and how RTOS activity can be visualized on a timeline to expose the root cause.

Every embedded team has met this bug. The tests are green, the code review found nothing, and the system runs for hours. Then, with no crash, no fault handler and no log entry, something happens late. By the time anyone notices, the evidence is gone.

The instinct is to look closer: set a breakpoint, add a printf, step through the suspect function. With timing bugs, that is the wrong move. Halting the CPU, or even slowing it with a log message, changes the interaction between tasks and interrupts you are trying to catch, and the bug goes quiet.

You rarely need new hardware to break this cycle. Most Cortex-M devices already contain trace blocks that timestamp what the software is doing while it runs at full speed. This article follows one real case, an industrial monitoring node that answered its network master too late, and shows how a few ITM writes and one kernel hook turned an invisible scheduler into a timeline that exposed the cause.

Contents

1. Why Do Timing Bugs Disappear When You Set a Breakpoint? 4

2. Why Are RTOS Timing Bugs so Hard to Debug? 4

3. How Do ITM and SWO Let You Debug without Stopping the MCU? 5

4. What is Timeline Debugging, and How Do You Read a Trace? 6

5. How Do You Put an RTOS Scheduler On a Timeline? 7

6. Can You Debug a System That is Already Misbehaving? 8

7. How Do You Find the Root Cause of a Missed Deadline? 8

8. SWO or ETM: When Do You Need Instruction Trace? 10

9. How Do You Stop a Timing Bug From Coming Back? 10

10. Does It Matter How the Code Was Written? 11

11. Can AI help with Debugging, Not Only with Writing Code? 12

12. Conclusion 13

13. References 14

1. Why Do Timing Bugs Disappear When You Set a Breakpoint?

Halting the CPU changes the timing that caused the bug. You freeze the exact interaction between tasks and interrupts that you were trying to catch.

"The bug disappeared every time the debugger stopped the MCU"

Recently, a real customer's industrial monitoring node exposed the problem. Three tasks acquired sensor data, processed measurements and answered requests from a network master. Under heavy load, roughly once every few hours, a response arrived after the master's timeout. There was no crash, no fault handler and no log entry, just a late reply.

A breakpoint in the communication task showed a correct state, and the next few hundred responses were on time. Logging was no better. A 40-character printf over a UART at 115200 baud takes about 3.5 ms, so on a system with millisecond deadlines it is not an observation but a new timing profile. Engineers call this a **Heisenbug**: the bug that vanishes when you try to look at it.

The code may be entirely correct. Every function does what it was written to do. The failure lives in the order and timing of events across the whole system, so the question must change from "What is my code doing?" to "What is my system doing, and when?"

That question matters even more as AI generates a growing share of firmware. A model can produce correct-looking code in seconds, but deadlines are validated in execution, not in source files. This bug was found by measurement, not code review, and the next one will be too.

Modern Cortex-M devices already contain hardware support for observing software execution without disrupting normal operation. The challenge is rarely collecting the data. The challenge is turning that data into actionable insight before the next intermittent failure occurs.

2. Why Are RTOS Timing Bugs so Hard to Debug?

An RTOS like Eclipse ThreadX and FreeRTOS hides scheduling from the application. Each task behaves as though it owns the processor, while the kernel switches context, services interrupts and arbitrates shared resources. A conventional debugger shows only the code running right now. It cannot tell you why a task was suspended, which interrupt triggered a switch, or who holds the semaphore everyone is waiting for.

Common RTOS Failure Modes

- Priority inversion
- Starvation
- Deadlock
- Interrupt storms
- Missed deadlines
- Stack overflow

Typical failures share this property: priority inversion, starvation, deadlock, interrupt bursts that push a task past its deadline, and stack overflow. On the monitoring node any of them could explain a late reply, and guessing which would take days. Answering the question needs a timeline of events across the whole system.

3. How Do Instrumentation Trace Macrocell (ITM) and Serial Wire Output (SWO) Let You Debug without Stopping the MCU?

Most Cortex-M3, M4, M7 and M33 devices include CoreSight trace hardware. DWT timestamps events and counts cycles, ITM records software events, SWO transports trace data to the probe and, on some devices, ETM adds instruction-level trace (Figure 1).

CoreSight at a Glance

DWT › Measure timing
ITM › Generate events
SWO › Stream trace data
ETM › Record instructions

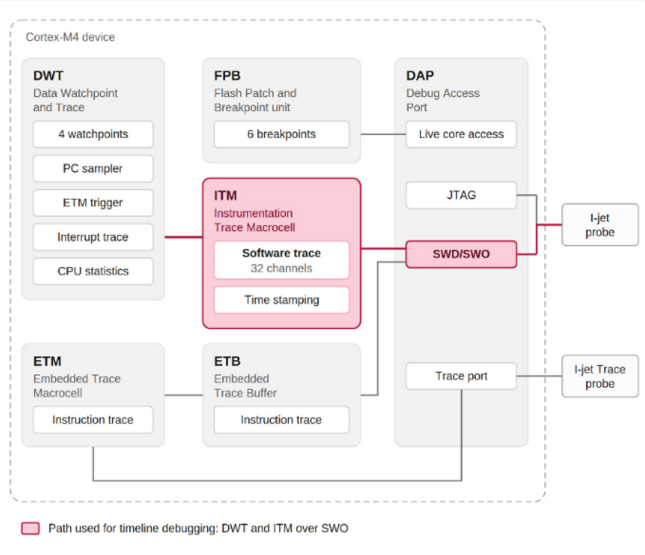

Figure 1: Arm CoreSight [1] blocks on a Cortex-M4. The highlighted path, DWT and ITM through SWO to the I-jet probe, is the one used in this article.

Marking an interrupt takes two writes:

```
void ADC_IRQHandler(void)

{

    ITM->PORT[1].u8 = 0x01;      /* enter */

    HAL_ADC_IRQHandler(&hadc1);

    ITM->PORT[1].u8 = 0x02;      /* exit  */

}
```

The hardware adds the timestamp and the CPU moves on after a few cycles, where a UART printf costs milliseconds. The DWT reports interrupt entry and exit by itself, and live watch and the memory window can update while the target runs.

For the monitoring node this meant no rewiring, only a probe on the debug connector that was already there.

4. What Is Timeline Debugging, and How Do You Read a Trace?

A timeline view plots software events against time, much like an oscilloscope plots voltage. Interrupts, thread switches, variable values, ITM events and, with a suitable probe, current share one time axis (Figure 2).

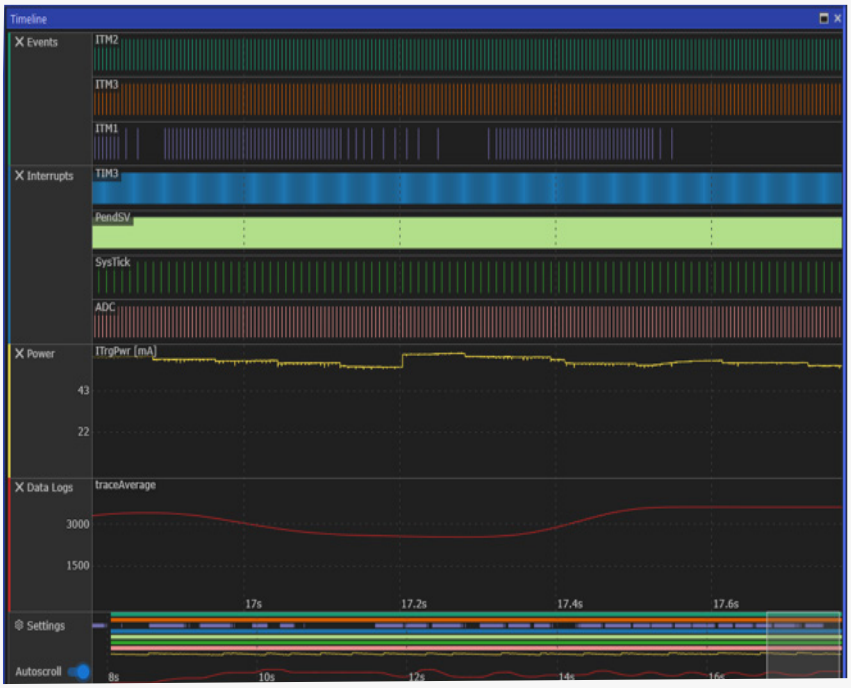

Figure 2: The IAR Timeline stacks ITM events, interrupts, current and a logged variable on one time axis. Captured with the cross-platform IDE for IAR Embedded Workbench for Arm[2].

Timeline Fact

The IAR Timeline combines RTOS activity, interrupts, software events and application data into a single time-correlated view, making timing problems easier to identify.

The interrupt log needs no instrumentation at all (Figure 3). On Cortex-M the PendSV exception performs the context switch in ThreadX, FreeRTOS and most other kernels, so the number of PendSV entries is effectively a task-switch counter.

PendSV = Context Switch

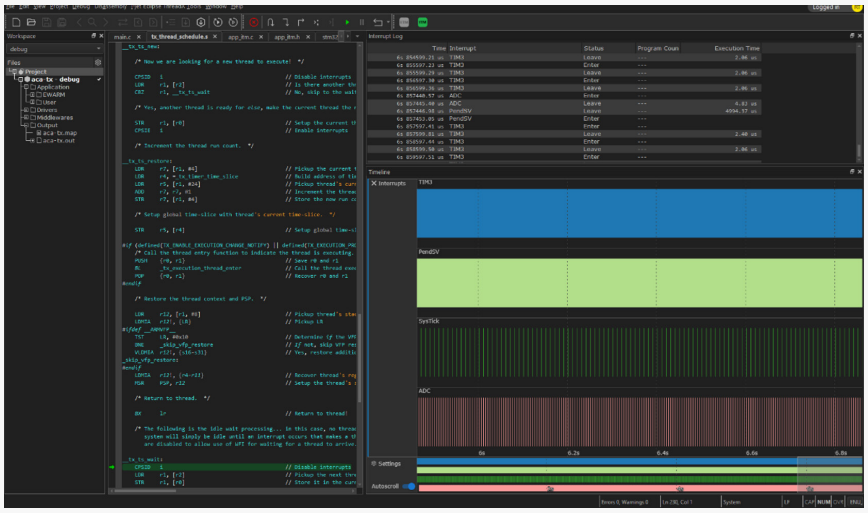

Figure 3: The Interrupt Log lists every entry and exit with its execution time. The Timeline draws the same events as lanes.

5. How Do You Put an RTOS Scheduler On a Timeline?

Every RTOS has a routine that switches tasks. Emit one ITM event from it, carrying a small integer that identifies the task, and each task appears as its own level on the timeline, like a logic analyser trace.

Eclipse ThreadX makes this surprisingly easy. With execution profiling enabled, the scheduler invokes instrumentation hooks on every thread switch. In this example, those hooks send thread IDs to the ITM trace stream:

```
#include <arm_itm.h>

__root void _tx_execution_thread_enter(void)

{

    TX_THREAD *t = _tx_thread_current_ptr;

   ITM_EVENT8(ITM_THREADID_CHANNEL, (UCHAR)t->tx_thread_ux_seq);

}

__root void _tx_execution_thread_exit(void)

{

   ITM_EVENT8(ITM_THREADID_CHANNEL, 0U);   /* 0 means idle */

}
```

Ready ≠ Running

A timeline shows when a thread stopped. An RTOS-aware debugger tells you why. The Thread List reads the kernel's own data structures and shows each thread's state, priority and peak stack usage (Figure 4). On the monitoring node it settled one question at once: whenever a reply was late, the communication task was never blocked. It was Ready, but never running.

Trace data alone identifies activity. RTOS awareness [3] provides the context needed to understand scheduler decisions. Viewing both together eliminated the need to manually reconstruct task interactions from logs.

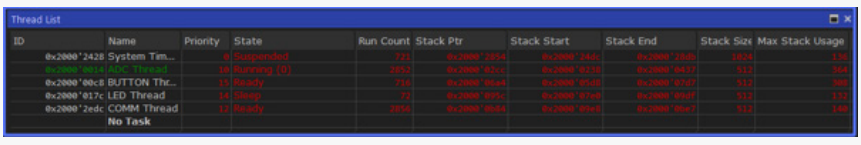

Figure 4: While the ADC thread runs its processing burst, the communication thread is Ready but not running.

6. Can You Debug a System That is Already Misbehaving?

Yes. The debugger can attach to an application that is already running, without a reset or a download. Use the Attach to Running Target toolbar button (Figure 5) or Debug > Attach to Running Target. Batch runs use the --attach_to_running_target option of cspybat.

Response deadline:
5 ms allowed
14.1 ms observed

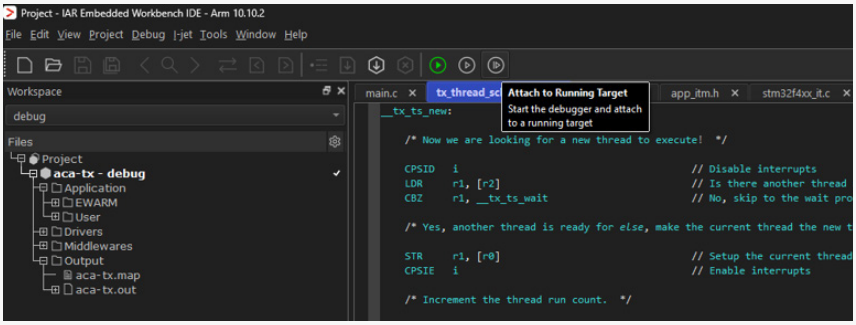

Figure 5: Attach to Running Target connects to a running application without a reset or download

This changes the order of work. The usual download, reset and run destroys the state you want to study, while a rig that fails every few hours may already have run for two. Attaching keeps that situation alive: connect, record, and the application never notices.

Breakpoints defined in the project are set during attachment, or disabled if they cannot be set without stopping the target. Not every driver and device combination supports the option, and C-SPY tells you when it does not.

7. How Do You Find the Root Cause of a Missed Deadline?

The next time the node misbehaved, nobody pressed reset. The debugger attached to the running board and the Timeline recorded while the fault happened. The ISR markers, the thread-switch hooks and an ITM event carrying each response time were already in the build.

The figures in this article come from a reproduction of that behavior under Eclipse ThreadX on an STM32F429 board with an I-jet probe. An ADC interrupt fires every 5 ms. When the averaged sample moves, the acquisition thread runs a processing burst, which we padded to 3.9 ms of CPU time and force on every sample with a Live Watch flag.

A communication thread must answer a request, arriving every 20 ms, within 5 ms. The response needs 2.5 ms of CPU time. Priorities, highest first: acquisition, communication, then the LED and button threads. In the Events lanes, ITM1 marks each ADC interrupt and ITM2 carries the measured response time in microseconds.

One breakpoint helped, used the other way round: on the line that counts a missed deadline. It stops the CPU only after the late response is recorded, so the Timeline already holds what led to it. The capture showed the sequence (Figure 6a):

1. A request arrives just as an ADC interrupt fires. The acquisition thread runs its 3.9 ms burst while the communication thread sits Ready.

2. In the 1.1 ms gap, communication gets the CPU, but the next ADC interrupt, 5 ms after the first, starts another burst halfway through its response.

3. This repeats. The response completes after the third burst, 14.1 ms after the request, against a 5 ms deadline.

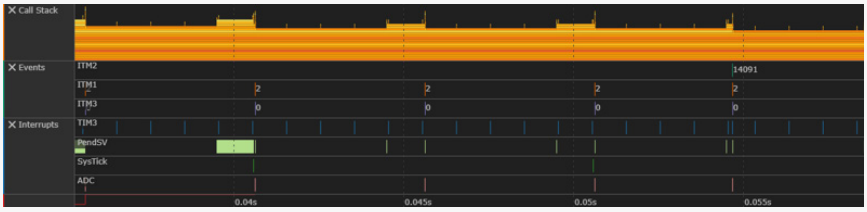

Figure 6a: With the processing burst on, a response needing 2.5 ms of CPU time takes 14.1 ms. ITM2 shows it in microseconds. The event and interrupt logs also provide precise timing.

CoreSight provided the data. The IAR Timeline and logs provided the explanation.

Nothing here is wrong in isolation. The defect is the interaction, visible only with all events on one time axis. Zoomed in (Figure 6b), the ADC interrupt lasts under 5 µs, yet the CPU then stays busy for 3.9 ms.

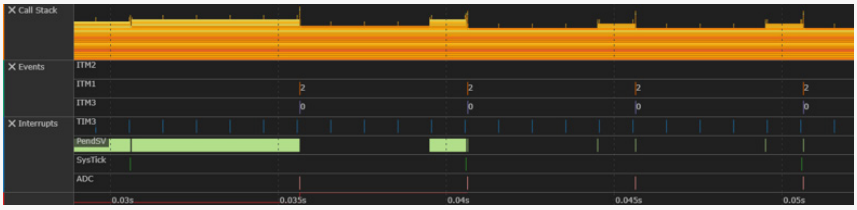

Figure 6b: One 5 ms ADC period. The PendSV lane is green only while the CPU is idle.

The fix was a design change. The acquisition thread now only averages and posts the new value to a queue. A worker thread, below the communication thread and even below the LED and button threads, does the processing (Figure 6c).

With the processing burst still enabled, late responses fell from 348 to none, and the worst case from 14.1 ms to 2.5 ms (Figure 8). The CPU is still busy. What changed is who gets it first. A breakpoint could never have shown this, and a log would have taken hours to reconstruct.

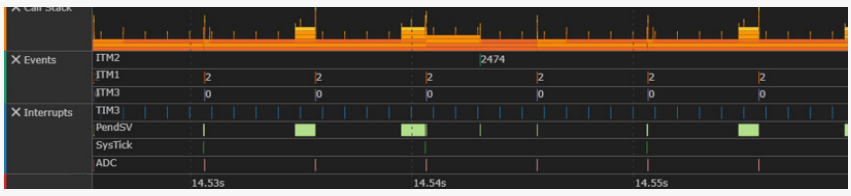

Figure 6c: After the fix, the same request is answered 2.5 ms later.

Before
14.1 ms response time
348 missed deadlines

After
2.5 ms response time
0 missed deadlines

8. SWO or ETM: When Do You Need Instruction Trace?

For timing problems like this, SWO is usually enough. Add ETM instruction trace when you must know exactly which instructions ran, for example after a hard fault or a corrupted pointer: it lets you step backwards from the fault through the last thousands to millions of instructions.

It needs a trace probe such as I-jet Trace and a device with a trace port or on-chip buffer. With ITM events travelling alongside, the Call Stack and interrupt lanes share one time axis (Figure 7).

Use SWO first
Most timing problems can be solved without instruction trace.

Add ETM only when you must reconstruct every executed instruction.

Figure 7: With ETM and ITM together, the Call Stack lane shows what runs between interrupts: the ADC thread blocks on an empty queue and the scheduler's wait loop runs.

9. How Do You Stop a Timing Bug From Coming Back?

Turn the scenario that exposed it into an automated test. The fix was a change of task structure and priority, exactly what gets undone by accident months later, when someone "optimizes" the worker away.

The firmware can measure its own worst case with the DWT (Data Watchpoint and Trace) cycle counter, almost for free:

```
CoreDebug->DEMCR |= CoreDebug_DEMCR_TRCENA_Msk;   /* once */

DWT->CTRL        |= DWT_CTRL_CYCCNTENA_Msk;

t_request = DWT->CYCCNT;              /* request received */

uint32_t dt = DWT->CYCCNT - t_request;    /* response sent */

if (dt > worst_response) worst_response = dt;
```

A test build replays the heavy-load scenario and fails if the worst case, or the count of late responses, exceeds the budget (Figure 8). Running the same build configuration locally and in CI helps ensure that timing regressions are caught early. With the IAR Build Tools available as container images, the same environment can be used from a developer's desktop to automated test pipelines on Linux and Windows.

Worst-case latency:
Before: 1,184,693 cycles
After: 207,940 cycles
82% reduction in response latency

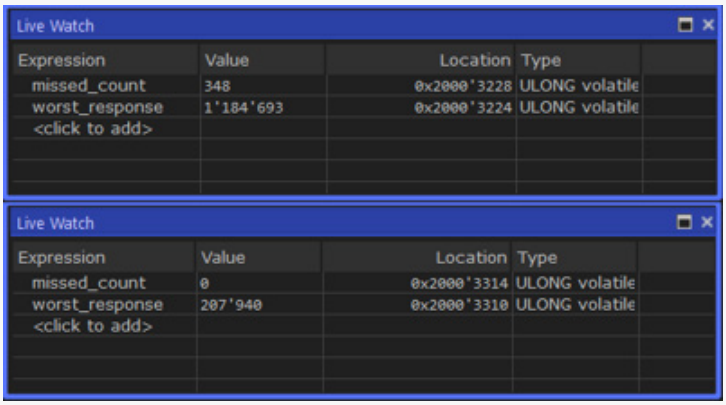

Figure 8: Live Watch before and after the fix, with the processing burst enabled. Before: 348 late responses and a worst case of 1,184,693 cycles, 14.1 ms at 84 MHz. After: none late and 207,940 cycles, 2.48 ms.

10. Does It Matter How the Code Was Written?

No. Developers can still write every line themselves, in the cross-platform IDE on Windows or Linux, in Visual Studio Code, Eclipse or whatever they prefer. They can also use AI-assisted tools and code generators. What matters is that, whichever route a team takes, it can build, test, verify and trust the result.

The monitoring node shows why. Every function was correct, and the failure lived in the interaction. Generated, suggested or pasted code is a draft until a developer has reviewed, built, tested, measured and approved it, and the test from the previous section is how a team does the measuring. Tools can speed up each step but cannot be accountable, and in functional safety work under IEC 61508 or ISO 26262 the evidence needs an accountable owner.

11. Can AI Help with Debugging, Not Only with Writing Code?

Yes. The open-source MCP connector for the IAR C-SPY Debugger [4] lets AI assistants access debugger and trace data. Instead of reasoning only about source code, an assistant can help analyse what actually happened on the target.

One detail matters: the target should remain running whenever possible. An assistant that repeatedly halts the CPU can make a timing bug disappear just as effectively as a breakpoint.

AI can help gather and analyse evidence, but the developer remains responsible for interpreting the results and approving the fix.

"For timing bugs, the target should remain running whenever possible. AI agents can help analyse what happened, but repeatedly halting the CPU can make the problem disappear."

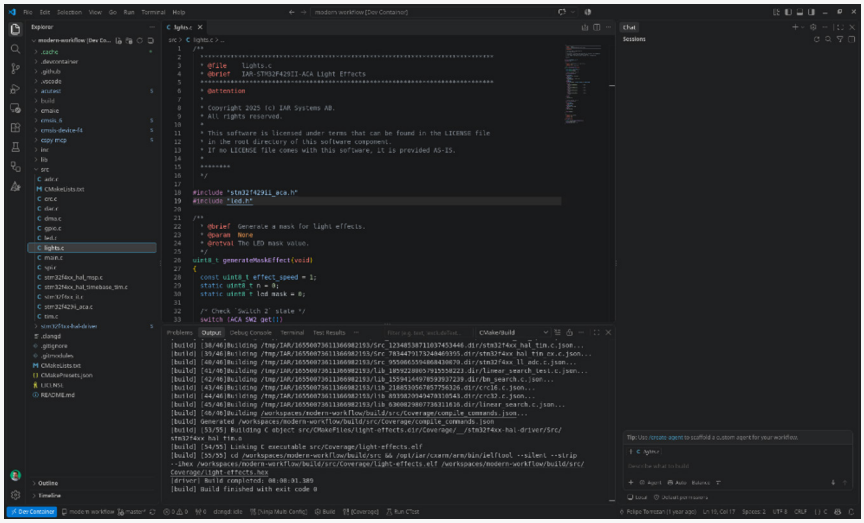

Figure 9. Microsoft Copilot connected to the IAR C-SPY Debugger through MCP in Visual Studio Code, using debugger and trace data to analyse real target execution.

To try this AI agentic workflow (Figure 9) without installing a toolchain locally, start with the Modern Workflow example project [5] at Github, a Visual Studio Code Dev Container with IAR Build Tools, CMake presets, unit tests and CI.

12. Conclusion

The hardest embedded bugs are rarely syntax errors. They are timing problems that exist only in the interaction between correct components, so you need a tool that shows time. The trace hardware is already in most Cortex-M devices you use. A few ITM writes and one kernel hook turn an invisible scheduler into a picture you can read, and the same capture becomes a test that keeps the bug fixed.

The code may come from a keyboard or from a model. The proof comes from the timeline and the test.

In modern embedded systems, knowing when something happened is often as important as knowing what happened.

13. References

[1] Arm CoreSight architecture overview: https://support.arm.com/documentation/102551/0100/Understanding-CoreSight

[2] IAR Platform and Embedded Workbench for Arm: https://www.qt.io/development/iar-embedded-tools

[3] IAR C-SPY and RTOS awareness documentation: https://docs.iar.com/ewarm/10.1x/en/c-spy-debugging/preface/contents.html

[4] IAR C-SPY MCP connector: https://github.com/iarsystems/cspy-mcp

[5] IAR modern workflow example, with CMake, and a CI pipeline: https://github.com/iarsystems/modern-workflow

About the Author

Rafael Taubinger is Industry Insight Lead & Technical Advisor at IAR by Qt Group in Sweden, with more than 20 years of experience in the embedded systems industry. He specializes in embedded software development, debugging and trace technologies, modern development workflows, long-term maintenance strategies, and regulatory topics such as the Cyber Resilience Act (CRA). Rafael has delivered technical training and workshops throughout North America, Latin America, and Europe.

About Qt Group

Qt Group (Nasdaq Helsinki: QTCOM) is a global software company, trusted by industry leaders and over 1.5 million developers worldwide to create applications and smart devices that users love. We help our customers increase productivity through the entire product development journey: from UI design to software development, optimizing embedded systems, and quality management. Our customers are in more than 70 different industries in over 180 countries. Qt Group employs some 1100 people, and its net sales in 2025 were 216.3 MEUR. To learn more, visit **www.qt.io.**

Qt Group