es:saar

The Appropriate Tool for Behavioral Insight

Selecting the Right Introspection Tool to Debug the Runtime Behavior of Embedded Systems

Abstract:

As embedded systems have grown more complex, so has the need to understand how they behave in the real world. Fortunately, available tools have kept up with this demand, and they continue to improve. However, as these tools have matured, the core issue has shifted from limited insight to fragmented approaches to gain it. Over time, several tool families have emerged to address specific analysis problems within particular market segments under specific constraints. There is no single scenario that defines embedded development. Accordingly, no single tool can cover all scenarios. Each segment now relies on familiar toolchains, which strengthens fragmentation.

The best way to select the most appropriate tool from the available options is to start with the specific context of the situation. Context always matters more than raw power. Next, find tools that align with the necessary insights and associated trade-offs. Ultimately, it's not just about tool capability. It's about fit. This guide is designed to help readers find the right fit.

The Appropriate Tool for Behavioral Insight

Selecting the Right Introspection Tool to Debug the Runtime Behavior of an Embedded System

All engineering teams working in the field of embedded systems face the same challenge. At some point, they must reconcile design intent with real-world behavior. Whether they are bound by automotive compliance, able to move quickly without these constraints, or need to calibrate a sensor or tune a sophisticated controller, their approach remains essentially the same. They must observe, validate, and adjust runtime system behavior.

However, "system behavior" is a broad term, so it's important to consider what it *actually* means. As engineers, what are we attempting to observe?

Sometimes, it's about observing a system's health. Is the system stable and responsive? Does it remain within acceptable limits? Other times, we're interested in execution and event flow. Which functions ran, when, and in what order? Sometimes, it comes down to state changes in variable values. The problem is that no single tool can provide insight into all these areas without significant trade-offs. Most tools excel in one area but compromise in others.

This leads us to the topic of this article. We will explore the types of insights and trade-offs that developers of embedded systems face when selecting tools to interface with their runtime behavior. Before delving into these categories, however, it is important to understand the context in which these tools were developed. We will briefly review their history and the reasons for their significant differences across industries. In summary, this article aims to answer a single question:

How should one choose the right tool to interface with the runtime behavior of an embedded system?

Part 1: Tool Evolution and Segment Realities

1.1 Evolution of Introspection Tools

Before 1990: The Early Days of Inaccessibility

In the early days of embedded systems, the 8- and 16-bit microcontrollers of the 70s and 80s developers faced extreme constraints: no runtime hooks, no memory to spare, and no structured visibility. Debugging meant toggling LEDs or sending out serial logs. Oscilloscopes could validate pin-level timing but gave no insight into what software was doing. EPROM-based workflows made iteration painfully slow, and in-circuit emulators (ICEs), though powerful relative to other tools of the time, were expensive and inaccessible to many.

1990s: Internal Introspection

The shift from EPROMs to flash memory significantly shortened iteration cycles. In the mid-1990s, the use of ICEs changed with the adoption of on-chip debugging via interfaces such as JTAG. These interfaces provided direct access to the processor's state, eliminating the need for specialized emulation hardware. This lowered costs and reduced setup complexity. Although integration remained fragmented, debugging became more accessible, supported by an increasing number of vendor tools and early integrated development environments (IDEs). This marked the start of insights into the internal behavior of an embedded system. At the same time, tools for embedded systems were tailored to segment-specific requirements. Hardware-in-the-loop (HiL) systems began to be adopted, and the CAN calibration protocol (CCP) was introduced by ASAM in 1994. For the first time, developers could measure and calibrate ECU parameters during execution.

2000s: The Emergence of Tracing

From 2000 to 2010, embedded systems evolved from basic task-switching software into real-time capable platforms. The widespread adoption of real-time operating systems (RTOS) created new demands for debugging. Engineers needed time-correlated insight into task switches, interrupts, and variable behavior, something that traditional breakpoint-based debugging could not provide. In response, tool and silicon vendors developed new solutions. For example, ARM introduced Serial Wire Debug (SWD) as an improvement over JTAG and its CoreSight infrastructure with the Cortex-M3. This enabled engineers to use instruction tracing features, such as the Embedded Trace Macrocell and the Instrumentation Trace Macrocell, as well as data streaming via the Serial Wire Viewer (SWV). Meanwhile, other vendors, including Infineon, NXP (formerly Philips), and Renesas, promoted their own debugging and tracing strategies. Tool vendors such as Lauterbach, PE Micro, and IAR Systems integrated these

architectures to provide tracing and live introspection, though capabilities varied depending on the silicon. For example: SEGGER's J-Link, introduced in the early 2000s, expanded SWV support by 2008. This enabled trace capabilities on Cortex-M devices for a broader market.

2010s: From Tooling Constraints to Fragmentation

From 2010 to 2020, the demand for real-time performance increased, and multicore processors became standard even in mid-tier embedded applications. This pushed silicon and tool vendors further. ARM's CoreSight introduced the Trace Memory Controller (TMC) and the Embedded Trace Router (ETR), which enabled trace collection and streaming through high-bandwidth interfaces. Tools such as the SEGGER J-Trace Pro made live ETM streaming possible, enabling developers to collect instruction-level trace data in real time on the host machine. Outside of ARM, Infineon's MCDS + AGBT, Renesas's advanced debug interface, and NXP's Nexus trace offered comparable capabilities and increasingly specialized in automotive, industrial, and safety-oriented applications.

While this symbiosis between silicon and tool vendors developed further, a novel approach matured during this time: software tracing platforms. These platforms became a viable alternative for many applications. Tools such as Percepio's Tracealyzer offer visual timelines, interruption heat maps, CPU load graphs, and state transition models. With these tools, developers could observe their systems' behavior for race conditions and measure jitter, preemption patterns, or latency bottlenecks.

After 2010: The Shift to Custom Toolchains

By the end of the 2010s, the landscape of embedded tools had fractured. There was a tool for every use case, segment, and budget. However, it was rare for one tool to work across boundaries. The challenge was no longer tool availability. The challenge was stitching together insights across layers of timing, control flow, and variable state.

Nevertheless, a complete toolchain exists. In regulated environments such as automotive and aerospace, teams sometimes build complete dynamic testing stacks by combining hardware-in-the-loop (HiL) platforms, trace tools, and calibration protocols such as XCP. These setups can deliver near-total observability of timing, control flow, and variable access. However, this level of visibility comes at a cost: dedicated infrastructure, custom integration, and maintenance that few teams outside of these regulated environments can afford. For everyone else, selecting the right tools for insight is a trade-off. This trend has only accelerated since 2020. Today's embedded systems are real-time, distributed, field-deployed, and often decoupled from traditional debugging infrastructure. The old problem was acquiring sufficient tools. The new problem is choosing the right tool.

Key Takeaways

- **Before 1990:** Embedded debugging was constrained by hardware limitations. Engineers relied on indirect methods like LEDs and oscilloscopes. System internals were essentially invisible.
- **1990 - 2000:** On-chip debugging (via JTAG) and flash memory shortened iteration times and marked the start of internal system introspection. Tool support grew, and real-time calibration protocols emerged.
- **2000-2010:** The rise of RTOS-driven systems created demand for time-correlated debugging. Trace infrastructure (e.g., SWD, SWV, ETM) enabled real-time visibility. Tools integrated these features to varying degrees.
- **After 2010:** The challenge shifted from tool availability to tool selection and integration. Highly specialized toolchains enable full observability in regulated industries, but most teams must balance cost, complexity, and insight.

1.2 Segments Realities for Introspection Tools

Not all engineering teams face the same constraints, risks, or realities. Having looked at how embedded debugging has evolved over time, the next step is to understand how it varies across different market segments. Depending on the industry, debugging and tuning embedded systems can mean vastly different things. At one end of the spectrum are the “four horsemen of compliance” in electronics: Automotive, Medical, Aerospace, and Military. At the other end of the spectrum are consumer and industrial embedded systems, as well as R&D endeavors.

Tools in Compliance Constrained Segments

In the automotive industry, the ISO 26262 standard influences nearly every decision, from the use of XCP-based calibration tools such as CANape to the integration of ECUs into full-scale hardware-in-the-loop environments from companies like dSPACE and National Instruments (NI). The cost is enormous, but so are the safety requirements. The result is a rigid, high-assurance pipeline with certified validation platforms. Automotive debugging is shaped not by what’s technically possible, but by what’s certifiable and required. In aerospace, this concept is taken even further. It's not just about whether a debugger can provide insight to tune a controller, but also whether it can perform consistently under audit. This leaves little room for dynamic, exploratory debugging. Medical devices face similar constraints, albeit with a different flavor. Regulatory bodies require compliance and design traceability from requirements to test results. Debug ports are usually locked down during production or field testing. Logging is then non-intrusive, scrubbed of patient data, and subject to audit. The barrier to insight here is legal, not technical. Engineers may want full runtime visibility, but unless it's validated, encrypted, and documented, it's out of the question. In military applications, all compliance demands converge, including extreme performance requirements, strict safety and mission-critical reliability standards, and uncompromising information security measures. Tooling must support highly deterministic systems under real-time constraints, often on ruggedized or custom hardware, while meeting stringent assurance standards such as DO-178 for defense aviation and MIL-STD guidelines. Debugging and introspection tools must be examined for telemetry risks and operated within tightly controlled, often classified, environments.

Tools in Less Constrained Segments

At the same time, there are hardly any formal restrictions on tools for consumer and industrial embedded systems, unlike the other segments we discussed. The main requirements are cost-effectiveness and speed, including fast feedback, iteration, and shipping. If a tool is inexpensive and gets the job done, it's preferred. However, that freedom comes with a risk: a lack of structure and predictability. Debugging can be quick when it works but it can also become a critical bottleneck when it doesn't. Without fitting tools or a clear process, teams often get stuck in test-iterate-postprocess loops that don’t scale for complex projects.

Tools in Unconstrained Segments

Finally, we come to the opposite end of the compliance spectrum: university and corporate R&D. Here, debugging is not about compliance constraints but about possibility. There are essentially no regulations to adhere to, only prototypes to develop and insights to pursue. Depending on their budget and focus, teams might set up a complete dSPACE setup or put together a Python trace pipeline. Sometimes, they don't adopt existing tools, but rather invent the ones that the rest of the industry will use five years later. This is also the space from which es:saar came. Our tool, es:scope was developed on test benches, in labs, and under pressure to determine why a system was misbehaving.

Key Takeaways

- **Compliance-constrained segments prioritize certification over flexibility.** In sectors like automotive, aerospace, medical, and military, tools must support traceability, repeatability, and auditability. Introspection can be limited by legal, regulatory, or security requirements, not technical feasibility.
- **Less constrained segments value speed and pragmatism:** In consumer and industrial systems, the focus is on fast iteration and low cost. Engineers often rely on lightweight tools, but this can introduce fragility and scaling issues as systems mature and become more complex.
- **Unconstrained environments foster experimentation and invention:** In R&D and academic contexts, the absence of compliance requirements allows teams to build or adapt tools freely in pursuit of insightfully driving innovation across the broader industry.
- **There is no “best” tool, only the right one for your constraints:** Effective debugging is about aligning the tool’s capabilities with what you need to observe, what you're allowed to observe, and what you must prove. That balance is different in every segment.

1.3 From Context to Selection

Toolchain Fragmentation Over Industry Segments

This brief overview has demonstrated that, over time, tools have been developed to address specific problems within particular market segments under specific constraints. Consequently, no single tool can cover all scenarios because no single scenario defines embedded development. As embedded systems have grown in complexity, so has the effort required to understand their real-world behavior. Fortunately, the tools have kept up, and they continue to do so. However, as the tools matured, the core issue shifted from limited insight to fragmented approaches to gain insight. Now, each segment relies on familiar toolchains, which strengthens diversification.

Selecting the Most Appropriate Tool

In theory, a lack of insight should not be due to inadequate tools nowadays, but rather to an inappropriate choice of tools. In practice, however, the lack of tools can exacerbate the problem. As we have seen, many embedded developers, particularly those working with industrial and consumer systems, must use whatever tools are at their disposal. The best way to select the most appropriate tool among the available options is to start with individual constraints. Context always matters more than raw power.

Navigating Between Insight and Trade-Off

After understanding the individual context, the next step is to find tools that align with the necessary insights and associated trade-offs. In this article, we define three core trade-offs: intrusion, effort, and depth of insight. In the next chapter, we will examine the categories of available tools, the insights they offer, and how to evaluate them based on these trade-offs. Our goal is not to define the "best" tools but to develop a practical strategy for selecting the most appropriate ones given real-world constraints. Ultimately, it's not just about tool capability. It's about fit.

Part 2: The Tool Map

This chapter outlines the types of insights that embedded tools provide and explains how to evaluate the relevant trade-offs. Most tools offer in-depth analysis of one aspect of system behavior, such as timing, status, control flow, or internal state. However, they rarely provide insights that extend beyond this focus without significant trade-offs. Each tool serves a purpose, but when misunderstood or misapplied, even the most powerful ones can create barriers or mislead developers about system behavior.

Insight Categories

In this text, we focus on insights into the runtime behavior of embedded systems. In the introduction, we asked what we wanted to observe in system behavior, since this is a broad term. Sometimes, behavioral insight is simply about the system's health: Is it functioning properly? Is the CPU load acceptable? Has memory usage spiked? Other times, we care about the execution flow, where we ask which functions are running, whether the control logic is executing in the right order, whether a branch was taken, and whether a task switch occurred. Then there is the event flow layer, which considers how long operations take, when events occur relative to each other, and whether the real-time behavior still meets system constraints. Finally, we may be interested in the state flow of variables over time: How does a pulse width modulation (PWM) impact the current ripple? Does a sensor signal have creepage? Is an internal flag toggling at the right time? These questions divide behavioral insights into four layers.

1. **System Health**
   Broad operating metrics like CPU load, free heap, error counters to observe performance degradation, memory leaks, long-term uptime issues. Typical tools are monitoring agents, dashboards, diagnostic logs. These will be excluded from further discussion as this is usually only an aggregate of the other system behavior insights.
2. **Execution Flow (Logical Behavior)**
   Observe which functions run in which order to understand logic flow, tracking regressions, uncovering concurrency issues.
3. **Event Flow (Timing Behavior)**
   Observe the duration, and overlap, task timing, ISR jitter, preemption patterns to detect race conditions, missed deadlines and responsiveness breakdowns.
4. **State Tracking (Variable Behavior)**
   Focuses on internal data during runtime to control parameters, flags, counters, etc. and to support tuning, loop debugging, and real-world validation.

Trade-Off Categories

Engineers sometimes overvalue tool familiarity and undervalue trade-offs. However, understanding tradeoffs and hidden costs is just as important as understanding technical capabilities. To illustrate the trade-offs, consider tools such as hardware tracers, which provide precise, non-intrusive insights. However, they require significant setup effort and expertise. Conversely, low-effort options, such as variable watchers or printf logging, are easy to integrate but typically offer shallow insight, missing timing issues, masking concurrency problems, and distorting control flow.

The three key trade-off categories are intrusion, effort and insight depth:

1. **Intrusion:** Impact and change on system behavior.
2. **Effort**: Time spent setting up and maintaining tools, switching contexts, and iterating the process.
3. **Depth:** The extent to which a tool reveals the behavior of a system through the specific insights it provides.

Insight depth is a qualitative measure used here to evaluate how effectively a tool reveals system behavior through the insights it provides. High depth offers broad, direct visibility into internal behavior, whereas low depth results in narrow, indirect, or proxy-based views. For example, logic analyzers deliver high timing accuracy at system boundaries, but they provide limited insight into internal software behavior.

The Tool Categories

Category	Examples	Strengths	Best For	Limitations	Not Ideal For
Hardware Instruments	Oscilloscopes, Logic Analyzers	High signal accuracy, accurate voltage and timing at pin-level	Debugging electrical interfaces (SPI, I²C, UART)	No access to software internals; physical probing required	Debugging logic flow or software variables
Hardware Tracers	SEGGER J-Trace, Lauterbach TRACE32	Precise, non-intrusive execution tracing; nanosecond timing	Deep control-flow and timing analysis	Expensive; requires silicon support and dedicated probes	Low-cost projects; dynamic tuning of variables
Software Tracers	SEGGER SystemView, Percepio Tracealyzer	RTOS-level task tracing; event sequencing; visual timeline views	Scheduling issues, preemption bugs	Limited variable inspection; adds overhead; interface bandwidth constrained	High-speed signal analysis or multi-variable debugging
HiL Platforms	dSPACE, NI VeriStand	Full system simulation and validation with environment models and I/O	Certification workflows, integration testing	High cost, long setup, limited internal visibility without extra tooling	Quick iteration, low-level software insight
Calibration Protocols	Vector CANape, ETAS INCA (XCP/CCP)	Real-time calibration in automotive ECUs; works with industry standards	Standard ECU parameter tuning	Complex setup; limited speed and flexibility; intrusive in some configurations	Bare-metal systems, dynamic observability
IDE Debuggers	STM32CubeIDE, Keil uVision, GDB	Step-through execution, breakpoint logic, memory inspection	Start-up debugging, logic validation	Halts execution; breaks real-time behavior; intrusive	Debugging race conditions or timing bugs
Variable Watch Tools	STM32CubeMonitor, Infineon MicroInspector	Live view of scalar variables; fast setup; no external hardware needed	Parameter tuning, basic diagnostics	Limited signal types; vendor-locked; low bandwidth for fast-changing data	Complex data correlation, high-speed feedback loops
Custom Logging	UART prints, GPIO pulse tagging, CSV + Python	Universally accessible, hardware-agnostic, low cost	Early bring-up, prototype insight	Manual sync; lacks structure; hard to maintain; not scalable	Timing-critical systems or structured debug workflows
es:scope® Platform	es:scope, es:prot (middleware)	Real-time variable introspection; scope-like views; interface-agnostic	Control tuning, signal dynamics, feedback loops	Needs firmware integration; not a task tracer; not for historical stack analysis	RTOS flow tracing, post-mortem failure reconstruction

Part 3: The Categories in Detail

Now that we have mapped space of insight, trade-off and tool categories, the next chapter will link these by analyzing each tool category for the insight and tradeoff. We start at the edge of the system with hardware measurement tools and progress from hardware to software-based tools. This is not based on a scientific method or evaluation, but rather on developer experience which risks sacrificing measurable truth for usefulness.

Hardware Instruments

Oscilloscopes and logic analyzers from Tektronix, Keysight, and Rohde & Schwarz offer unmatched signal-level precision. These devices are the gold standard for verifying input/output (I/O) timing and decoding hardware protocols. However, they are ineffective beyond the system boundary. They cannot be used to understand execution flow; they can only be used to understand the signals that reach the pins, which can provide indirect insight into state and events. This can be useful for exposed signals. Apart from that, they are often too removed to provide behavioral insight.

Metric	Score	Justification
Event Flow	+2	These tools are best-in-class for signal-level timing accuracy. You get exact edge timing, pulse width, jitter, and delays with nanosecond resolution.
Execution Flow	–2	They can't access or interpret software logic, branching, or task switching. You only infer behavior indirectly from physical I/O.
State Flow	–2	They provide no access to internal variables or memory. Unless variables are exposed through pins, you're blind to them.
Intrusion	+2	Totally non-intrusive. They observe signals passively without touching system execution.
Setup Effort	0	Moderate. Requires physical wiring and knowledge of signal mapping for signals. If state variables are supposed to be exposed this has to be set up.
Depth	-1	Offers high timing insight at system boundaries but lacks visibility into internal behavior. The insight depth is low.

Hardware Tracing Tools

Tracing tools such as SEGGER J-Trace and Lauterbach TRACE32 use on-chip tracing architectures, like ARM CoreSight and Infineon MCDS, to capture execution paths at the instruction level, task switches, and precise timing, all with minimal intrusion during runtime. They are irreplaceable for diagnosing race conditions, validating execution paths, and analyzing deeply embedded real-time systems. They offer the perfect deal on the surface: fast, low-intrusion access to internal state, memory, and trace buffers. However, in the real world, the right interface may be missing, disabled in production, or limited by the silicon vendor. Trace bandwidth may collapse under high-frequency execution. In-depth analysis of controller variables may require extensive post-processing.

Metric	Score	Justification
Event Flow	+2	These tools capture timestamped execution traces with cycle-level accuracy. Perfect for identifying jitter, latency, race conditions.
Execution Flow	+2	Full visibility into execution flow: function calls, context switches, and branching even across interrupts and threads.
State Flow	0	Can capture variable changes if trace instrumentation is added, but this often requires extensive post-processing. Not suited for high bandwidth streaming or live tuning
Intrusion	1	Nearly zero runtime impact, but trace bandwidth can limit visibility during high activity.
Setup Effort	- 1	Setup is complex: you need the right silicon support, external probes, and trace clocks/pins. Often painful to configure across toolchains.
Depth	+2	These tools are integrated directly into the hardware, providing the deepest possible insight into runtime behavior. They capture execution, timing, and system state in exceptional detail.

Software Tracing Tools

When it comes to tracing, tools such as SEGGER SystemView and Percepio Tracealyzer provide more user-friendly solutions. They integrate with RTOS events and instrument application code. They also stream execution traces via SWO or UART. These tools provide insight into task scheduling, event timing, and flow behavior. They also have lower setup costs and broader accessibility.

However, they also have limitations. Runtime overhead, limited data throughput, and the need for manual instrumentation reduce precision. Although the required setup effort is reduced, it shifts to software setup. These tools help track control flow but often cannot display variable state changes quickly enough to debug unstable loops or sensor anomalies.

Metric	Score	Justification
Event Flow	+1	Provides reasonably accurate timestamps for RTOS events and user-instrumented markers. Limited by interface bandwidth and timestamp resolution (e.g. SWO/UART).
Execution Flow	+1	Captures task switches, interrupt entry/exit, and application-level events. Good for understanding RTOS behavior and logic sequencing.
State Flow	0	Can show variable changes if manually instrumented but not designed for fast or continuous streaming. Adds code overhead, lacks tuning-grade feedback.
Intrusion	0	Moderate. Adds runtime overhead through instrumentation and streaming, especially over UART/SWO. May affect some timing-sensitive systems.
Setup Effort	0	Easier than hardware tracers. Mostly software integration with vendor libraries but still requires careful tracepoint planning.
Depth	+1	Offers high contextual clarity at the RTOS level, but limited resolution for very fast or deeply nested code paths.

HiL Platforms

Hardware-in-the-loop (HIL) systems, such as those offered by dSPACE and NI, simulate the entire physical environment around the embedded system being tested. These systems are essential for validating complex systems under realistic conditions. They are also fundamental to certification workflows, especially in safety-critical domains.

These platforms have evolved into active observability environments that can capture signal traces, bus communication, and plant behavior with synchronized precision. With the right tools, engineers can inject faults, automate test scenarios, adjust parameters, and monitor system variables in real time. However, HIL visibility primarily focuses on external interactions and test responses rather than deep internal software execution. Accessing internal behavior, such as function execution or low-level timing, requires additional instrumentation and setup. Debug interfaces, calibration protocols (e.g., XCP), and internal model hooks are used for this purpose. This often involves time-consuming customization. Additionally, HiL systems are usually closed toolchains, so flexibility is limited when dealing with edge cases, such as pulse width modulation (PWM) details, asynchronous events, or custom triggers. HiL platforms excel at showing how the system responds to a test scenario. However, understanding why the system behaved in a certain way often requires deeper tracing or introspective tools, especially when relying on standard configurations.

Metric	Score	Justification
Event Flow	+1	Able to measure timing at the I/O, bus, and plant simulation levels. Supports synchronized signal injection and capture. Not cycle-exact at the firmware level but excellent for system-level timing validation.
Execution Flow	0	Limited. While some internal software state can be observed through model instrumentation or exposed interfaces (e.g. XCP), HiL is not built for tracing internal execution logic like task switches or instruction flow.
State Flow	+1	Good capabilities via integration with calibration protocols (XCP, CCP) or mapped I/O variables. Good for test automation and system validation scenarios.
Intrusion	+1	Low to moderate. HiL testing is typically non-intrusive at the signal level but depends on how internal variables are exposed and instrumented.
Setup Effort	-2	Very high. Requires plant models, hardware setup, simulation validation, and integration with DUT.
Depth	-1	Provides strong system-level correlation between inputs and outputs under real-world conditions but lacks visibility into internal software behavior unless specifically instrumented. Insight is broad but indirect in default configurations.

XCP/CCP Calibration Tools

Tools like Vector CANape and ETAS INCA are excellent at their intended purpose: live calibration and variable measurement via XCP or CCP. This is essential in automotive ECUs. However, these tools have limited functionality, providing little insight into control flow or real-time task interaction. Additionally, setup and infrastructure can be cumbersome, rendering them less suitable for early-stage or non-automotive development.

Metric	Score	Justification
Event Flow	-1	Very limited. Not designed to capture precise timing, task durations, or execution flow. Sampling intervals depend on bus bandwidth and configuration.
Execution Flow	-2	None. These tools provide no view into function calls, logic flow, or scheduling behavior.
State Flow	+1	Excellent. Designed for high-precision, real-time variable access. Supports calibration, logging, and measurement of mapped internal variables through XCP/CCP.
Intrusion	+1	Low. Works via dedicated measurement protocols (XCP/CCP), designed to be minimally intrusive on the running system.
Setup Effort	-1	Moderate to high. Requires proper integration into the build system (A2L files), variable mapping, and configuration.
Depth	+1	Offers high-resolution access to internal variable states in calibrated ECUs, making it ideal for tuning and validation. However, it lacks visibility into control flow and timing behavior, and insight is limited to predefined, scalar-accessible data.

IDE Debuggers

For many engineers, the debugger integrated into their IDE, such as GDB, Keil, or STM32CubeIDE, is their go-to tool. These debuggers are ideal for identifying specific types of bugs, including stack overflows, memory corruption, and logic errors. However, they are fundamentally intrusive. Breakpoints halt execution. Stepping distorts timing. In real-time or concurrent systems, this approach often masks the bug you're trying to expose. Breakpoints are useful, especially in the early stages of development, but they can be misused in systems where timing is the problem.

Metric	Score	Justification
Event Flow	-2	Very poor. Halting the system destroys timing context. Cannot observe real-time execution or concurrency.
Execution Flow	0	Moderate. Stepping through code gives insight into logic paths, but not under real-time conditions. Limited to static exploration.
State Flow	+1	Good for local variable inspection, stack content, and memory access. But not real-time.
Intrusion	-2	High. Requires halting or stepping the CPU. Distorts or breaks real-time behavior.
Setup Effort	+2	Very low. Integrated in most IDEs, typically works out-of-the-box with minimal configuration.
Depth	+1	It offers deep symbolic access to all system internals, including registers, memory, and variables, at any point in time. However, since insight is only available during halted execution, there is a lack of continuity and context for observing real-world runtime behavior.

Custom Logging and Scripting

Printf logging, GPIO edge tagging, and CSV dumps with Python post-processing work well in resource-constrained environments and for early prototyping. However, they do not scale well for embedded development. Logs get out of sync. Timing becomes distorted. Analysis becomes manual, slow, and error-prone. While these methods solve immediate problems, they introduce hidden costs, such as tribal knowledge, fragility, and an increased maintenance burden.

Metric	Score	Justification
Event Flow	-1	Low. GPIO tagging can offer basic event timing, but UART/printf distorts execution and lacks precision. Alignment issues common.
Execution Flow	-1	Limited. Printfs or GPIO toggles can signal when a code block runs, but offer no structured or high-resolution flow tracking.
State Flow	0	Moderate. Can output variables manually; flexible, but labor-intensive and not scalable.
Intrusion	-1	Medium to high. Printfs and logging distort timing, consume CPU cycles, and may interfere with behavior.
Setup Effort	-2	High. Requires custom code, script maintenance, sync work, and post-processing. Error-prone and manual.
Depth	0	Offers basic visibility through custom signals or logs, but lacks structured, continuous, or scalable insight. Depth depends entirely on manual effort and design-time foresight, making it fragile and inconsistent.

Variable Watch Tools

In response to the need for real-time tuning, many silicon vendors now offer variable watch tools, such as STM32CubeMonitor, MicroInspector from Infineon, and Real-Time Chart from Renesas. These tools connect via debug interfaces to extract scalar variable values in near real time. They're ideal for parameter tuning and early validation. However, they are limited to pre-selected variables, are often vendor-locked, and are constrained by bandwidth. Additionally, they don't capture timing, task context, or control flow.

Metric	Score	Justification
Event Flow	-1	Minimal. No access to execution timing, scheduling, or delays. Not suitable for debugging timing-sensitive behavior.
Execution Flow	-1	None. Offers no awareness of task switches, execution paths, or logic flow.
State Flow	+1	High for selected variables. Fast feedback, continuous updates of scalar variables. Limited to pre-defined, low-bandwidth channels.
Intrusion	0	Low to moderate. Generally safe for runtime use, but performance impact depends on polling frequency and interface.
Setup Effort	+1	Low. Provided by chip vendors, integration is usually simple and well-documented.
Depth	+1	Focused. Excellent for parameter tuning and watching control signals, but blind to system behavior outside selected variables.

es:scope platform

The ES:Scope platform is a software-based testing and measurement solution for embedded systems. It includes lightweight C middleware for the target and a desktop application that offers runtime visibility and interaction with internal variables via standard interfaces, such as UART, USB, and Ethernet.

The platform enables virtual probing and live tuning without the need for external hardware. Its strength lies in variable-level access and runtime interaction. Although it is limited in control-flow insight, it offers a pragmatic, lightweight alternative for many visibility and calibration tasks.

Metric	Score	Justification
Event Flow	0	Indirect. While not designed for timing trace or scheduling analysis, sampling rates may be sufficient for timing correlation between signals depending on update rate and signal behavior.
Execution Flow	-1	Not the focus. No native function-level tracing or task execution tracking. Not intended to replace ETM/SWV/RTOS-aware tools.
State Flow	+2	Strong. High-speed streaming of scalar variable data with live interaction and tuning. Virtual probes offer flexible low-intrusive access and interface-agnostic offer flexible transmission.
Intrusion	0	Low. Instrumentation is lightweight and interface-agnostic; uses standard transport layers. Does not block or halt execution. Impact depends on bandwidth configuration and sampling strategy.
Setup Effort	+1	Moderate to low. Requires integration of es:prot middleware into the target, but no external probes or vendor-specific tooling needed. Desktop setup is straightforward once middleware is running.
Depth	+1	Provides strong, real-time visibility into internal variables with live tuning and virtual probes.

Key Takeaways

Most teams have the tools they need to capture certain aspects of system behavior. However, very few tools offer insight across boundaries, such as those between timing and logic, function flow and variable state, and system behavior and internal causality. Each category exists for a reason. However, none exist without tradeoffs. What matters isn't just what a tool does, but also how it does it and the cost in terms of procurement, setup, and usage. Moreover, the real cost isn't just effort; it's the delay in understanding and time to insight.

When debugging needs to happen in real time across layers and often in the field, that cost becomes a bottleneck. Effort becomes unpredictable, and getting stuck in adjustment iterations becomes the norm. That's why understanding this landscape matters. It's not about replacing tools that work, but rather knowing where they work, where they don't, and what's missing in between. The following table summarizes this landscape.

	Insight			Tradeoff		
Category	**Timing Insight**	**Control Flow Insight**	**Variable State Insight**	**Intrusion**	**Setup and usage effort**	**Depth**
Hardware Instruments	2	-2	-2	2	0	-1
Software Tracers	1	1	0	0	0	1
Hardware Tracers	2	2	0	1	-2	2
HiL Platforms	1	0	1	1	-2	-1
Calibration Protocols	-1	-1	1	1	-1	1
IDE-Debuggers	-2	0	1	-2	2	1
Variable Watch Tools	-1	-1	1	0	1	1
es:scope®	0	-1	2	0	1	1
Custom Logging	-1	-1	0	-1	-2	0

Part 4: Summary

When selecting a tool for observing embedded system behavior, the challenge is rarely technical capability alone. Instead, it's about **finding the right fit**—given constraints, requirements, and trade-offs. The following three-step process can help guide effective tool selection:

---

1. **Context**

Start by understanding your unique situation:

- **What do you need to observe?** Timing, logic, state?
- **What are you allowed to observe?** Consider compliance, security, or legal limitations.
- **What must you prove?** Auditable traceability, functional safety, or runtime correctness?
- **What tools are available to you?** Account for your budget, platform support, and existing infrastructure.

---

2. **Insight**

Identify the kind of insight your task requires:

- **Timing Insight**: Do you need to know *when* things happen, with meaningful time resolution?
- **Control Flow Insight**: Do you need to trace *what* code runs and *in what order*?
- **Variable State Insight**: Do you need to observe *how internal data* changes at runtime?

---

3. **Tradeoff**

Every tool comes with compromises. Evaluate them:

- **Intrusion**: Will the tool alter system behavior or add significant runtime overhead?
- **Effort**: How much setup, integration, and iteration does the tool require?
- **Insight Depth**: Will the insight be direct and detailed or limited to indirect, proxy-level observations?

Ultimately, choosing the right tool is a strategic decision, not just a technical one. The more your needs align with the tool's capabilities, the faster and more confidently you can bridge the gap between design and behavior. In embedded systems, insight defines what is observable. Only what is observable can be controlled.

Disclaimer

As of this writing, I, Joshua Summa, am the CEO of es:saar, the company behind the es:scope platform discussed in this article. While every effort has been made to present an objective and balanced view of embedded tooling, it is important to disclose this affiliation explicitly. The evaluations and comparisons in this document reflect practical experience and technical analysis, but readers should be aware of this potential source of bias.

contact@essaar.de

Phone: +49 17681456107

E-mail: joshua.summa@essaar.de

Web: https://essaar.de

Campus A 1.1, Starterzentrum 1

D-66123 Saarbrücken

Sitz der Gesellschaft: Saarbrücken

Registergericht: Amtsgericht Saarbrücken, HRB 108623

Geschäftsführer: Joshua Summa