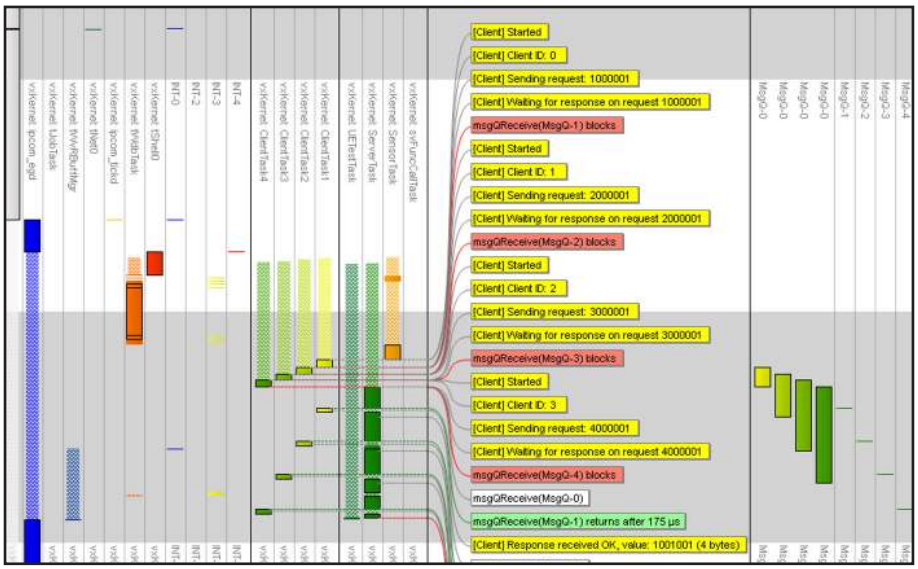
White Paper

Stop Guessing

See Inside RTOS Firmware with Visual Tracing

The debugging of RTOS-based systems can be dramatically simplified—reducing debugging time from days or weeks to hours—with better insight into their real-time execution. This requires RTOS-level software tracing, where good visualization is key to make sense of the data.



percepio®

Stop Guessing

See Inside RTOS Firmware With Visual Tracing

Dr. Johan Kraft, CEO and founder, Percepio AB

A real-time operating system (RTOS) is a fast, deterministic operating system that is typically small enough to be suitable for use in microcontrollers (MCUs), making RTOSes ideal for use in embedded and IoT applications. Developers are moving towards RTOSes because they can help reduce code complexity, guarantee hard timing deadlines, and facilitate reuse of software modules. The structure imposed by using an RTOS increases the application's maintainability and makes adding features easier. This is all good for management because it can increase development efficiency, decrease time to market, and improve product reliability and customer satisfaction.

However, all these advantages do not come without complexities. Subtle coding choices can result in elusive errors or performance issues in the final product, that are not apparent in the source code. The system seems to operate as intended in the lab, but there can be countless execution scenarios that are impossible to fully cover by testing or code reviews. In the worst case, the system passes testing but crashes during customer use. To ensure reliable operation, all parts of the application code need to follow best practices in RTOS-based design. This requires good insight into the system's real-time behavior.

Trace visualization can be thought of as a slow-motion video of the application's internals.

When such problems occur, debugging can be a nightmare since the circumstances that caused the problem are often not known in detail and thereby difficult to reproduce. Developers often find themselves guessing their way through this, trying one thing after another to get the application to run properly. But to be sure the problem is solved, developers need to understand the specific sequence of software events that caused the problem, including the interactions between the application and RTOS. Traditional debugging tools can't offer this capability.

RTOS trace visualization, which can be thought of as a slow-motion video of the application's internals, is a good way to be confident that an RTOS software runs as designed—and is the fastest way to detect and correct bugs.

Challenges in RTOS-based design

The main job of an RTOS is to provide multitasking, which allows for separation of software functionality into multiple "parallel" programs, known as tasks. An RTOS creates the illusion of parallel execution by rapidly switching the execution between the tasks. Unlike general-purpose operating systems, an RTOS gives the developer full control over the multitasking and therefore enables deterministic real-time behavior.

An RTOS takes control over program execution and brings a new level of abstraction in the form of these tasks. When using an RTOS, the control-flow of the program is no longer apparent from the source code, since the RTOS decides which task to execute at any given moment. This is a fundamental change, similar to the shift from assembly to C programming, as it allows for higher productivity using higher abstraction, but also means less control over the fine details.

While an RTOS can reduce the complexity of the application source code, it does not reduce the inherent complexity of the application itself.

This double-edged sword can make it easier to design complex applications, but these applications may subsequently turn out to be difficult to validate and debug. While an RTOS can reduce the complexity of the application source code, it does not reduce the inherent complexity of the application itself. A set of seemingly simple RTOS tasks can result in surprisingly complex runtime behavior when executing together as a system.

The developer needs to determine how the tasks are to interact and share data using the RTOS services. Moreover, the developer needs to decide important RTOS parameters such as task priorities (relative urgency) that can be far from obvious. Even if all your code is written according to best practices in RTOS-based design, there might be other parts of the system—in-house or third-party components—that run in the same RTOS environment but that may not follow the same principles.

The fundamental problem that makes RTOS-based design difficult is that RTOS tasks are not isolated entities but have dependencies that may delay or stall the task execution in unexpected ways. This may reduce performance, make the system unresponsive or unstable, or even cause intermittent data loss.

There is at least one kind of dependency between the tasks: they share the processor time. Higher-priority tasks can wake up and take over the execution at almost any point, until all active higher-priority tasks have completed. Moreover, tasks often use shared software resources (e.g., global data and peripheral interface drivers) that require blocking synchronization calls to prevent access conflicts. Such task dependencies may depend on many factors, including variations in input values, input timing, and task execution times.

Such issues are not visible in the code and often not detectable in unit tests, but show up in the integrated product, either during full system testing or in customer use. This makes them difficult to reproduce for debugging, unless the developer knows the exact sequence of software events that led up to the problem.

Finding Bugs in RTOS-based Systems

When the embedded industry moved from assembly to C programming, debugging tools quickly followed with source-level debugging, which made the C code perspective the normal debugging view. Unfortunately, the tools generally haven't evolved beyond this level. Some of them have been enhanced with features that allow developers to inspect the state of RTOS objects such as tasks and semaphores, but that is not enough. An RTOS debugging tool must understand the concept of time, be able to correlate events, and allow developers to observe the real-time behavior of an application.

This calls for RTOS tracing, which means that software events are recorded in runtime, in the RTOS kernel and optionally also in the application code, intended for host-side analysis.

Good visualization is key to understanding RTOS traces. Many embedded systems exhibit a cyclic behavior, which means that a trace mostly consists of repetitions of the "normal" pattern. The interesting part is usually the anomalies, but they can be very difficult to spot in a raw data stream. A graphical presentation makes any anomalies stand out.

An RTOS debugging tool must understand the concept of time, be able to correlate events, and allow developers to observe the real-time behavior of an application.

Moreover, a debugging tool that understands RTOS events and data structures can extract much more information from a trace than just the basic execution flow. For instance, it is possible to construct a dependency graph showing the interaction between tasks, interrupt service routines, and RTOS objects such as semaphores and message queues.

Seeing Is Understanding

The primary job for a software tracing tool is to capture events in the target system, from scheduling and RTOS calls to timer ticks and application-specific log messages. But a quick look at a typical event log (below) makes it clear although this might be useful, textual presentation does not scale to the large amounts of data resulting from software tracing. This example only represents about 2 milliseconds of execution. An RTOS trace spanning over a few minutes can contain millions of events.

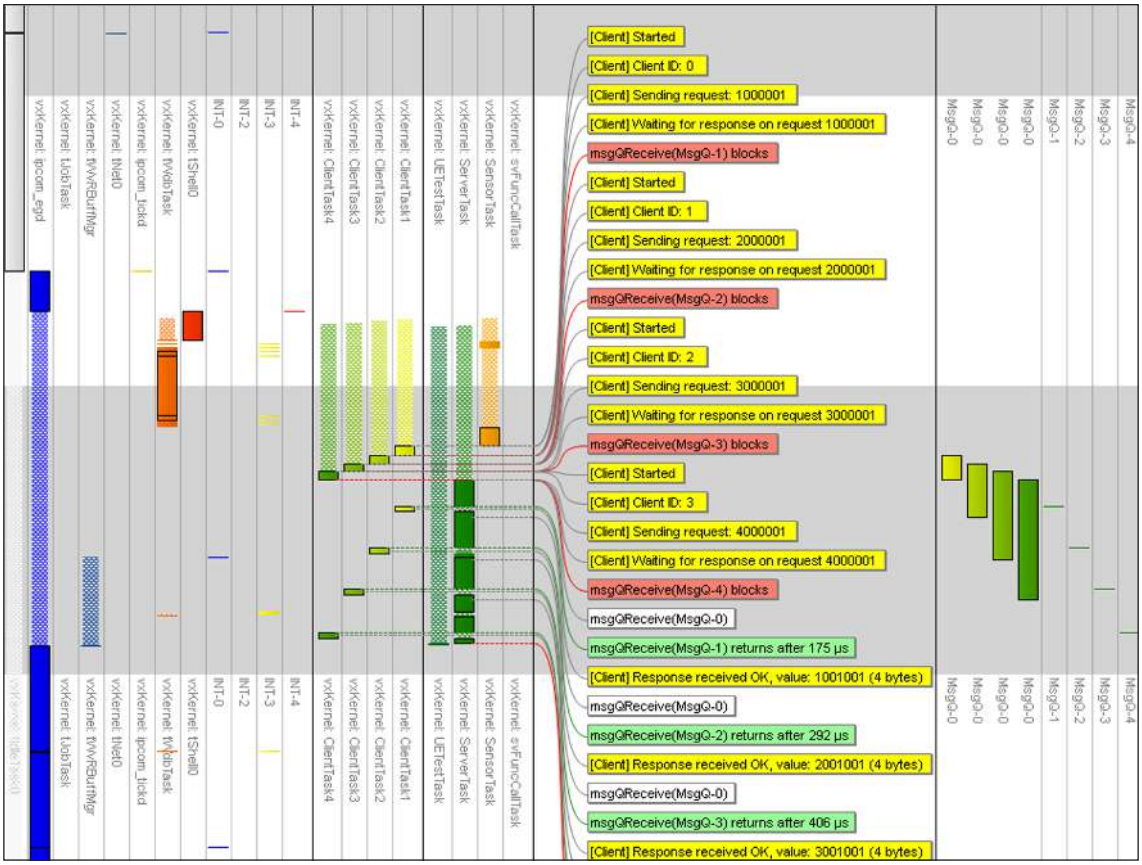
Timestamp	Actor	Event Text
1:01.569.286	Control	Context switch on CPU 0 to Control
1:01.569.297	Control	xQueueReceive(CtrlDataQueue, 100) returns after 1970 µs
1:01.569.310	Control	xQueueReceive(CtrlCmdQueue, 0) timeout/fail
1:01.569.327	Control	xQueueReceive(CtrlDataQueue, 100) blocks
1:01.569.347	IDLE	Context switch on CPU 0 to IDLE
1:01.570.231	IDLE	OS Ticks: 61569
1:01.571.231	IDLE	OS Ticks: 61570
1:01.571.239	IDLE	Actor Ready: Motor
1:01.571.257	Pos_ADC_ISR	Context switch on CPU 0 to Pos_ADC_ISR
1:01.571.268	Pos_ADC_ISR	xQueueSendFromISR(CtrlDataQueue)
1:01.571.277	Pos_ADC_ISR	Actor Ready: Control
1:01.571.286	IDLE	Context switch on CPU 0 to IDLE
1:01.571.296	Motor	Context switch on CPU 0 to Motor
1:01.571.317	Motor	xQueueReceive(MotorQueue, 10) timeout after 9422 µs
1:01.571.326	Motor	[Motor PWM] 0
1:01.571.346	Motor	xQueueReceive(MotorQueue, 10) blocks
1:01.571.366	Control	Context switch on CPU 0 to Control
1:01.571.376	Control	xQueueReceive(CtrlDataQueue, 100) returns after 2049 µs
1:01.571.389	Control	xQueueReceive(CtrlCmdQueue, 0) timeout/fail
1:01.571.407	Control	xQueueReceive(CtrlDataQueue, 100) blocks
1:01.571.427	IDLE	Context switch on CPU 0 to IDLE

An event log can be informative but it does not scale to large amounts of trace data.

Finding a bug in a large text log is like looking for a needle in a haystack, without knowing what the needle looks like. To get to the next level—to understand what is intended behavior and what is not—the developer needs suitable tools for data visualization.

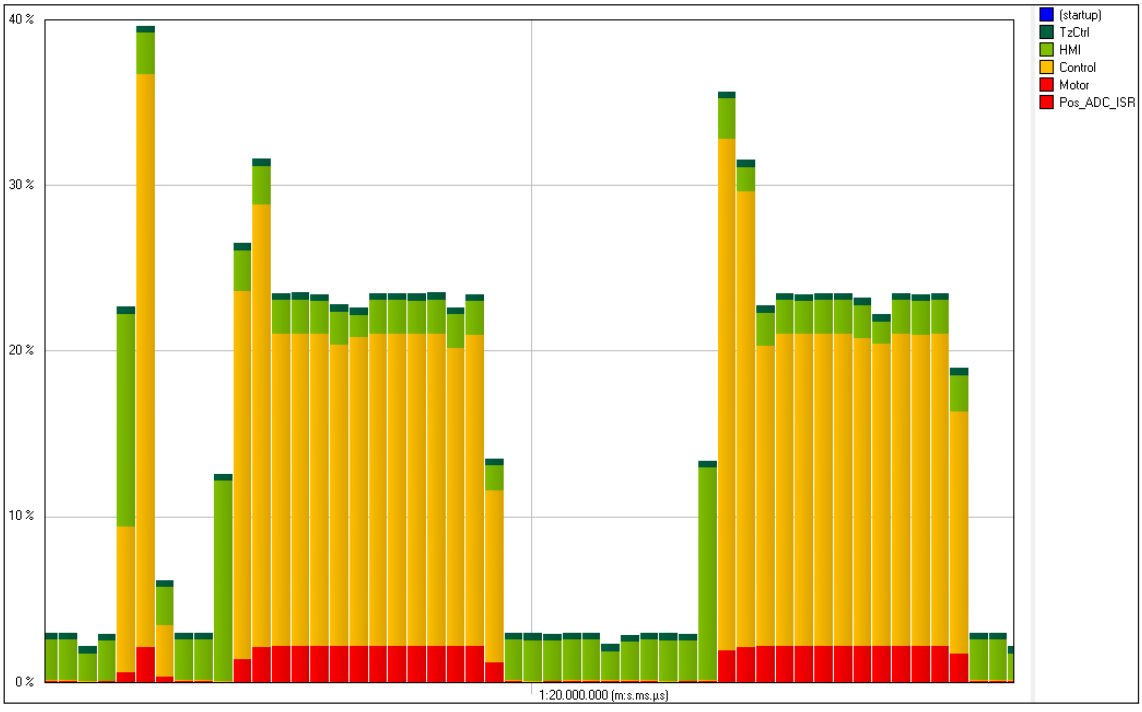
Finding a bug in a large text log is like looking for a needle in a haystack, without knowing what the needle looks like.

A better way to present large amounts of RTOS trace data is to use a graphical Gantt-style **trace view**, which allows for displaying the trace data on an interactive timeline. This allows the developer to zoom out and overview a vast amount of trace data, identify abnormal patterns, and zoom in to see the details. A graphical trace view may include not only the RTOS task execution but may also include API calls, application log messages, and other events, as shown in the example on the next page.



The trace view in Percepio Tracealyzer, showing a vertical timeline of the trace data.

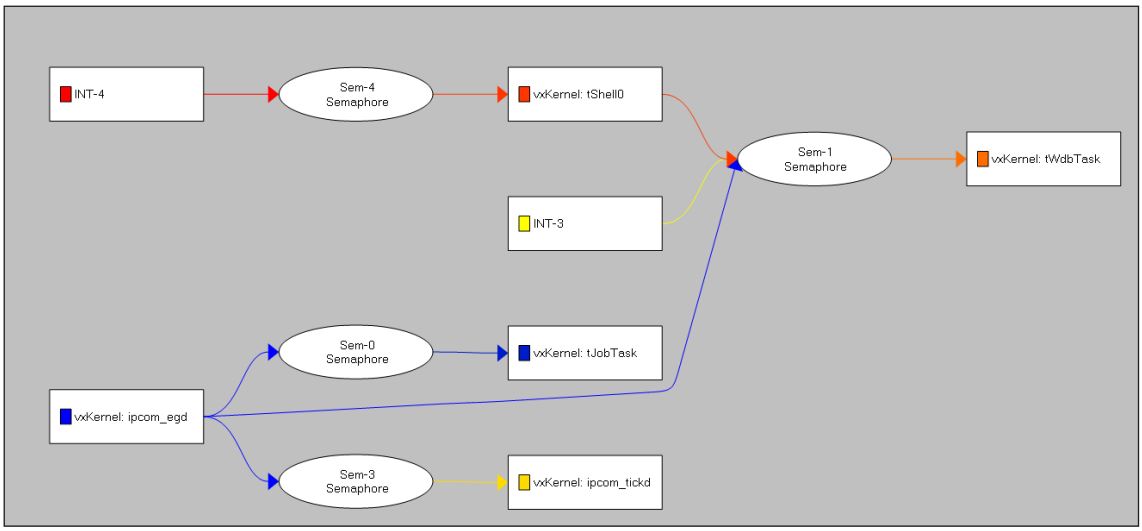
One common issue for embedded system developers is that target systems tend to be constrained in terms of both CPU power and memory. That is why a diagram like this **CPU load graph** (below) can be helpful: it displays the amount of processor time used by each task and interrupt service routine. Armed with this information, a developer can quickly see any hot spots where the load approaches 100 percent over longer periods (where tasks are likely to be delayed) as well as the amount of remaining CPU time available for adding more features without upgrading the hardware.



A CPU load graph from Percepio Tracealyzer shows processor usage per task over time, clearly showing where a specific task is consuming much higher CPU resources than others.

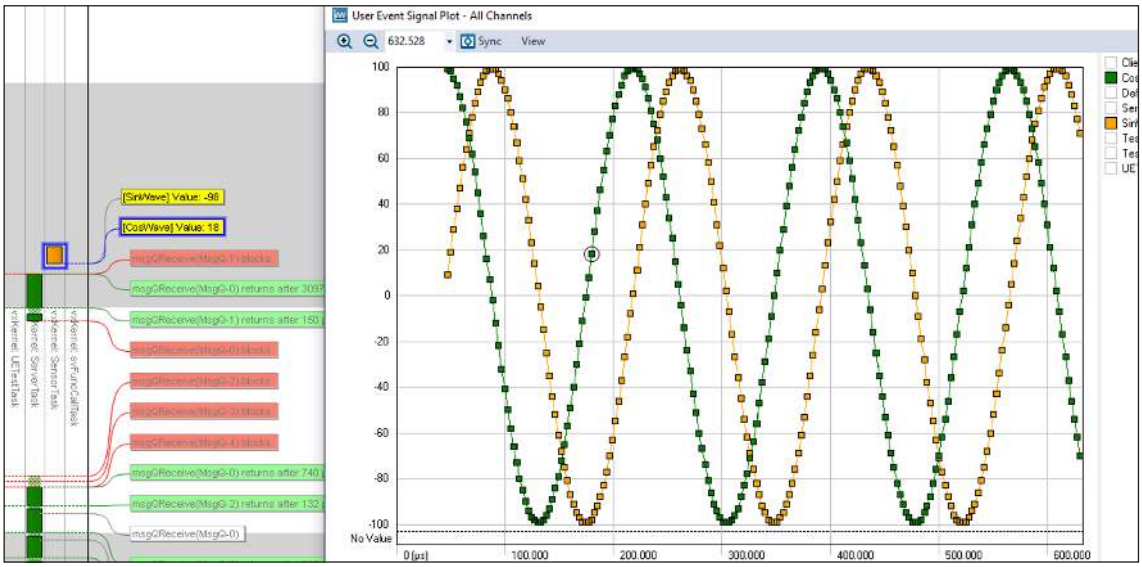
To connect back to the debugging aspect of tracing, a runaway task in an application that consumes more CPU than intended will be clearly visible in a CPU load graph. Tasks using inefficient "busy waiting" will also stand out. This wastes CPU time that otherwise would be available to other tasks and is a common violation of best practices in RTOS-based design.

By tracing API calls, you actually capture dependencies between tasks that a tool can visualize in a **dependency graph**, like the one below. Such a visual summary of the application design provides confidence that the application code works as intended and can also reveal bugs related to incorrect or missing API calls, i.e., issues that may cause stability problems.



This communication flow or dependency graph is a good starting point for many debugging sessions as it shows a bird's-eye view of the application architecture, and then allows the developer to drill down to any object in the graph by double-clicking it.

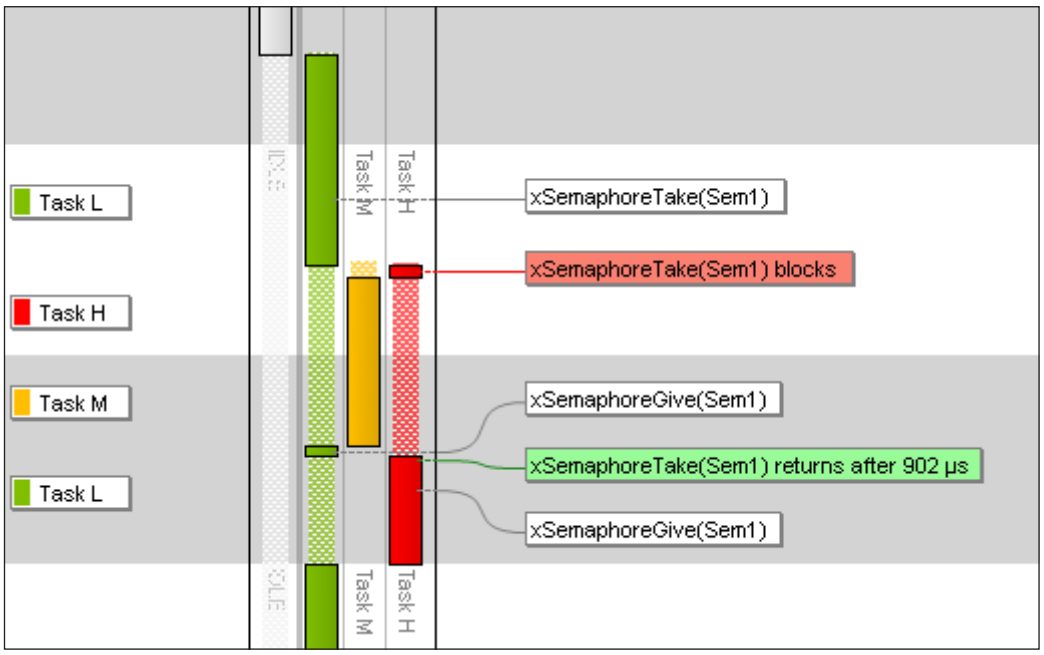
A good tracing tool should also allow the developer to log custom **application-specific data** in the trace stream. These events can be used for almost anything, but one common usage is to log important variable values and state transitions from the application code.



Percepio Tracealyzer supports application data logging and plotting over time.

Example: Priority Inversion on Mars
A highly visible and expensive example of RTOS debugging challenges was illustrated during NASA's Pathfinder mission that landed a rover on Mars in 1997. During the mission, the spacecraft experienced total system resets causing lost data. After much trouble, NASA found the cause to be a classic RTOS problem known as priority inversion.

Priority inversion may occur when a high-priority task (red Task H in the illustration below) tries to access a shared resource such as a communications interface, currently in use by a lower priority task (green Task L). Normally, Task H would become blocked for a brief duration until Task L returns the shared resource. A priority inversion occurs if a medium-priority task (yellow Task M) happens to pre-empt Task L at this point, delaying the high-priority task as illustrated below. In the NASA Pathfinder case, this caused repeated timeout errors leading to system resets, data loss and nearly a mission failure.



A priority inversion as shown in Percepio Tracealyzer.

Tracing allows developers to detect and prevent these types of issues. Tracing entails recording software behavior during runtime, allowing for later analysis of collected trace data. Tracing is most often a development bench activity, but tracing can also be enabled for production use, continuously active to record behaviors and catch errors post-deployment. Production tracing can be an effective technique for detecting rarely manifested errors that are difficult to reproduce in a debugger. These can include situations where the system responds more slowly than expected, gives incorrect or suboptimal output, freezes up, or crashes.

Hardware vs. Software Tracing
Tracing can be performed either in hardware (in the processor) or in software. Hardware-based tracing generates a detailed instruction-level execution history, while software-based tracing focuses on selected software events, typically in the operating system and important application-level interfaces. Hardware-generated trace provides details regarding control-flow and does not impact the execution of the traced system, but it does require special equipment and a trace-enabled hardware platform.

Software-generated trace does not require any special hardware and can even be deployed in shipped products similar to a black-box flight recorder used in aviation. Moreover, software trace allows for storing any application data at these events, including local variables and function parameters, while hardware trace is often limited to only the control-flow and possibly global data accesses, assuming a high-speed trace port. Software tracing does induce some CPU overhead, but this is typically not noticeable (a few percent).

Software tracing relies on target-system RAM for temporary buffering of the trace data, but the RAM buffers are usually configurable to allow for balancing RAM usage vs. buffer sizes.

Tracing is especially important for systems that integrate a real-time operating system. A central feature of RTOSes is multitasking—the ability to run multiple programs (tasks) on a single processor core by rapidly switching among execution contexts. Multitasking, however, makes software behavior more complex, and affords the developer less control over run-time behavior as execution is pre-empted by the RTOS.

Hardware Trace	Software Trace

• Generated by CPU features
• Exact instruction sequence
• Non-intrusive
• Often only control-flow trace
  • General data trace requires special chips and boards due to high data rates
• For lab use only
• Examples:
  • Lauterbach TRACE32®
  • iSystem Bluebox
	• Generated by software
	• Higher abstraction level
	  • RTOS task execution
	  • API calls
	  • Application-specific logging
	• Uses target CPU and RAM
	• Any software event and data
	• Advanced analysis possible
	• No extra hardware needed
	  • For lab use
	  • As crash recorder in field use
	• Examples:
	  • Wind River System Viewer
	  • Percepio Tracealyzer

"The many system views of Tracealyzer from Percepio make it easy to quickly find solutions that we have not seen using System Viewer. The visualization has several advantages over the System Viewer and makes it easier to understand system behavior."
— Johan Fredriksson, Software Architect, SAAB AB

Stop Guessing

The debugging of RTOS-based systems can be dramatically simplified—reducing debugging time from days or weeks to hours—with better insight into their real-time execution. This requires RTOS-level software tracing, where good visualization is key to make sense of the data. While several tools can provide basic RTOS tracing, more sophisticated visualization makes it far easier to understand the trace, spot important issues, and verify the solutions.

Percepio Tracealyzer is the best tool for software tracing, enabling sharper insight, higher quality, and faster development. It is available for leading RTOSes and Linux systems and supports most 32-bit and 64-bit processors out-of-the-box. The Tracealyzer application runs on Windows and Linux hosts.

A free, fully functional evaluation of Tracealyzer can be downloaded from https://percepio.com/. The installer includes a pre-recorded demo trace so you can quickly start exploring all 30+ interconnected views.

About Percepio

Percepio is the developer of a highly visual runtime diagnostics tool for embedded and Linux-based software, Tracealyzer, and of the award-winning DevAlert, a cloud-based service for error reporting in deployed IoT devices with Tracealyzer diagnostics. Percepio collaborates with several leading vendors of operating systems for embedded software and is a member of the Amazon Web Services Partner Network (Advanced Technology Partner).