percepio™
SENSING SOFTWARE

Stop Guessing – Trace Visualization for RTOS Firmware Debugging

Dr. Johan Kraft, CEO and founder, Percepio AB

Some decades ago, the embedded industry shifted focus from assembly to C programming. Faster processors and better compilers allowed for raising the level of abstraction in order to improve development productivity and quality.

We are now in the middle of a new major shift in firmware development technology. The increasing use of real-time operating systems (RTOS) represents the third generation of embedded software development. By using an RTOS, you introduce a new abstraction level that enables more complex applications, but not without complications.

An RTOS is a fast, deterministic operating system for use in embedded and IoT applications. These operating systems are often very small and so are suitable for use in microcontrollers (MCUs). The main job of an RTOS is to provide multithreading, thereby allowing for separation of software functionality into multiple "parallel" programs, which are known as "tasks." An RTOS creates the illusion of parallel execution by rapidly switching the executing task. Unlike general-purpose operating systems, an RTOS gives the developer full control over the multithreading and therefore enables deterministic real-time behavior.

There are many benefits of using an RTOS, but it is not a solution in itself and it comes with new challenges. Many developers have already dived into RTOS-based design, but perhaps without fully realizing the implications on debugging and validation.

The RTOS trend

Real-time operating systems have been around since the early 1980s, but -- for a number of reasons -- they are becoming more and more common. Embedded applications are becoming increasingly complex, connected, and event-driven, using advanced peripherals and middleware stacks that all need to be managed in parallel. Using an RTOS simplifies this greatly. The alternative to using an RTOS is to implement some kind of execution control yourself, using custom states and logic. For more complex applications, this tends to get complicated, error-prone, and problematic to maintain. By using an RTOS, you delegate the execution control to a highly optimized RTOS kernel that has been thoroughly tested and proven in use.

RTOSes are no longer a hype, as they were 10-15 years ago, but are used broadly in all sorts of embedded applications. It's not only commercial RTOS vendors who are advocating RTOS-based design; many MCU vendors now include RTOSes in their software development kits, in order to speed up firmware development for their customers.

The RTOS trend is also clearly visible in the Embedded Market Survey, probably the most established and trusted study of the industry. The most frequent answer for "greatest technology challenge" is now "RTOS." This concern has grown significantly from 12% in 2013, to 17 % in 2014, up to 26% in the 2015 survey. A related trend is that the use of in-house solutions, or bare-metal design (no RTOS),

is decreasing in favor of leading RTOSes. The RTOS market has historically been very fragmented, but now seems to be slowly consolidating as developers gravitate towards the leading players.

Although an RTOS is no "silver bullet" that solves all problems, and is perhaps not suitable for certain systems, the increasing use of RTOSes is without question a major trend today and one that's likely to continue.

Challenges of using an RTOS

So what makes an RTOS so special that it can be called the third-generation in firmware design? An RTOS is a very special software component, since it takes control over the program execution and brings a new level of abstraction in the form of tasks. When using an RTOS, the control-flow of your 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.

This is a double-edged sword. It 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.

As was previously mentioned, an RTOS is not a solution in itself, and there are many pitfalls for the unwary. 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 like task priorities (relative urgency) that can be far from obvious. Even if you have written all your code 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.

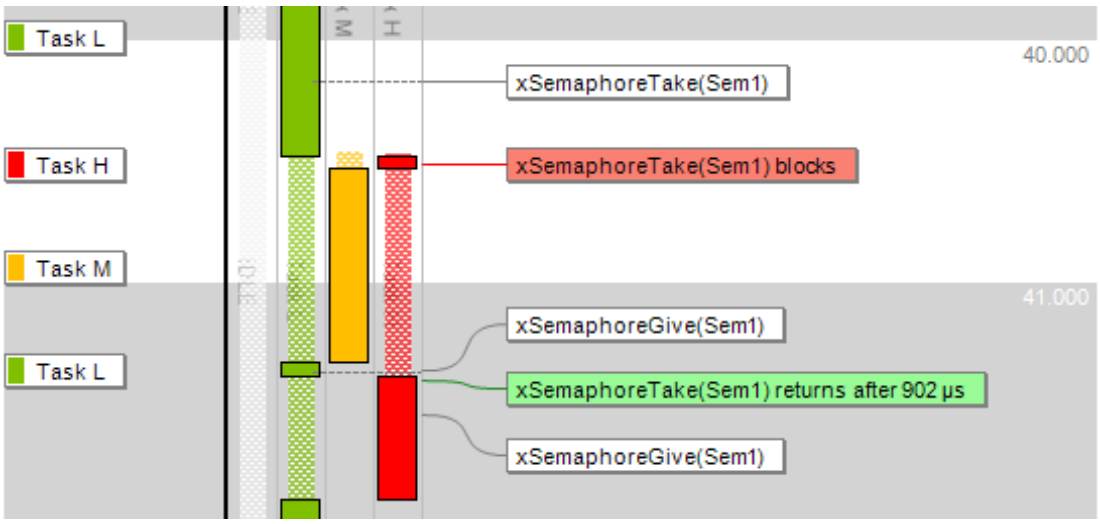
The fundamental problem that makes RTOS-based design difficult is that RTOS tasks are not isolated entities. There is at least one kind of dependency between the tasks -- their shared processor time. With fixed-priority pre-emptive scheduling, higher priority tasks can wake up and take over the execution at almost any point, thereby delaying the execution of lower priority tasks until all of the higher priority tasks have completed.

Other kinds of shared resources (such as global data or hardware peripherals) also results in dependencies between tasks, as the necessary synchronization may block the tasks from executing when desired. This may cause unpredictable delays if not designed correctly, independent of task priorities.

An example of this kind of problem can found in NASA's Pathfinder mission, where they landed a rover on Mars. 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."

Pathfinder with the Sojourner rover during preparation (Source: NASA)

Priority Inversion may occur when a high-priority task (Task H in the below illustration) tries to access a shared resource such as a communications interface, currently in use by a lower priority task (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 (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 system resets, lost data and nearly a mission failure.



Priority Inversion as shown in Percepio Tracealyzer

Task dependencies like scheduling and shared resources are affected by timing; e.g., execution times and input timing. Such timing properties are not visible in the source code and tend to vary depending on system state and situation. This makes it nearly impossible to predict the real-time behavior of an

RTOS-based system from the source code alone. Depending on many factors, tasks may execute slower than intended, have random unexpected delays, or never get to execute at all. And even if the system seems to operate as intended in the lab, there can be countless other execution scenarios with more or less significant differences in timing, some of which might cause problems. In the worst case, the system passes testing but crashes at random occasions for your customers. Such problems can easily be missed during system-level testing. Also, they can be a nightmare to reproduce for analysis unless you have access to detailed diagnostic information associated with the problem.

Debugging RTOS-based systems

It is quite natural to want to debug on the same abstraction level as you develop. When the embedded industry moved from assembly to C programming, debugging tools quickly followed by providing source-level debugging, thereby making the C code perspective the normal debugging view. However, debugging tools have not evolved to any significant extent in response to the RTOS trend. Some debuggers have been updated with "RTOS awareness" features that allow you to inspect the state of RTOS objects -- like tasks and semaphores -- while debugging. But these are incremental improvements of the "second generation" source-level debugger, with strict focus on the source code and run/halt/single-step debugging. Debugging an RTOS-based system using only a traditional source-level debugger is equivalent to using an assembly-level debugger when programming in C.

To fully understand the runtime behavior of an RTOS-based system, you need the ability to observe the real-time behavior at the RTOS level; i.e., a tracing tool with RTOS awareness. This works as a complement to traditional debugging tools, providing a timeline at the RTOS level that greatly facilitates debugging, validation, and performance optimization. While a traditional debugger is much like a microscope for inspecting the detailed execution within a task, tracing is more like a slow-motion video of the real-time execution.

There are two types of tracing, with slightly different purposes: Hardware-based and software-based. Hardware-based tracing is generated by the processor itself and provides a detailed execution trace at the source code or assembly level, but with little or no RTOS awareness. Moreover, recording the high-speed data stream requires advanced tracing hardware and suitable trace support both in the processor and on the board. This kind of tracing tends to produce vast amounts of data at a low abstraction level, which can be difficult to comprehend and is limited to the tracing functionality implemented in the processor. Hardware-based tracing is mainly used for coverage analysis and debugging of particularly nasty problems, where a low-level instruction trace is required.

Software-based tracing means that snippets of trace code are added to the target software for the purpose of recording important events in the RTOS and also -- optionally -- in the application code. You typically don't need to insert the RTOS trace code yourself, as this is often provided by the RTOS or tool vendor. Many RTOS kernels already contains “trace macros” at strategic locations. These are disabled by default, but can be redefined to store any available data when executed. Software-based tracing can therefore be used on any processor, store any kind of software event and include any relevant data. The downside is the overhead of the tracing code, but RTOS-level tracing on a modern 32-bit MCU requires only a few percent of the processor time, since relatively little data needs to be

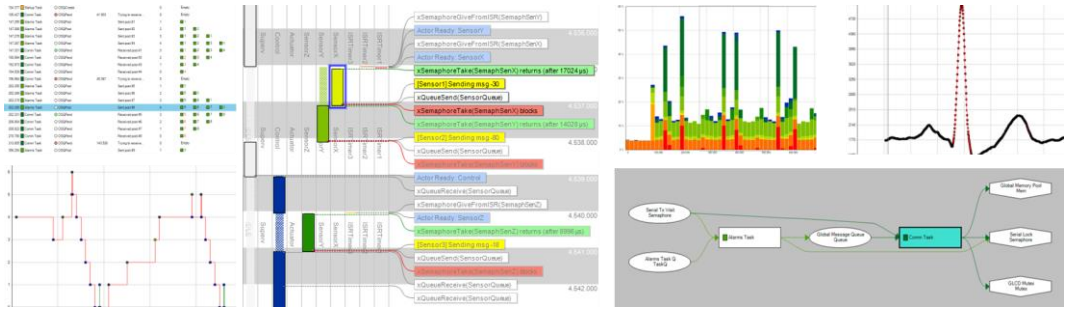
recorded and the traced events are not very frequent. This makes it possible to stream the data continuously using common debug interfaces, or other interfaces such as USB or TCP/IP. The low data rate also allows for tracing to a RAM buffer. Just a few kilobytes can be sufficient for getting a decent trace of the most recent events. This way, tracing can be used outside the lab as well; e.g., during field testing or in deployed operation.

Visualization is crucial for comprehending traces. Many embedded systems have more or less cyclic behaviors, so most of the trace data will be irrelevant repetitions of "normal" behaviors. The interesting parts are usually the anomalies, but they can be hard to find if you don't know exactly what to look for. The human brain is, however, phenomenal at recognizing visual patterns and anomalies, assuming that the data is visualized properly.

While several tools can display an RTOS trace as a classic Gantt chart, this visualization is not suitable for displaying other events in the same view; e.g., API calls or user logging. However, a vertical execution trace can display such events using text labels, pointing into the graphical execution trace. Moreover, showing just an execution trace is a quite limited perspective, since a lot more information can be derived from an RTOS trace. For instance, the interactions of tasks and ISRs can be presented as a dependency graph, and it is also possible to show trace views focusing on other relevant RTOS objects, such as semaphores, queues and mutexes.

Percepio Tracealyzer

Percepio is the leading specialist in RTOS trace visualization and offer the Tracealyzer tools, providing unprecedented insight into the runtime world. With Tracealyzer you can improve your development speed and software quality significantly, as you get better means for debugging, optimization and for avoiding potential problems at an early stage.



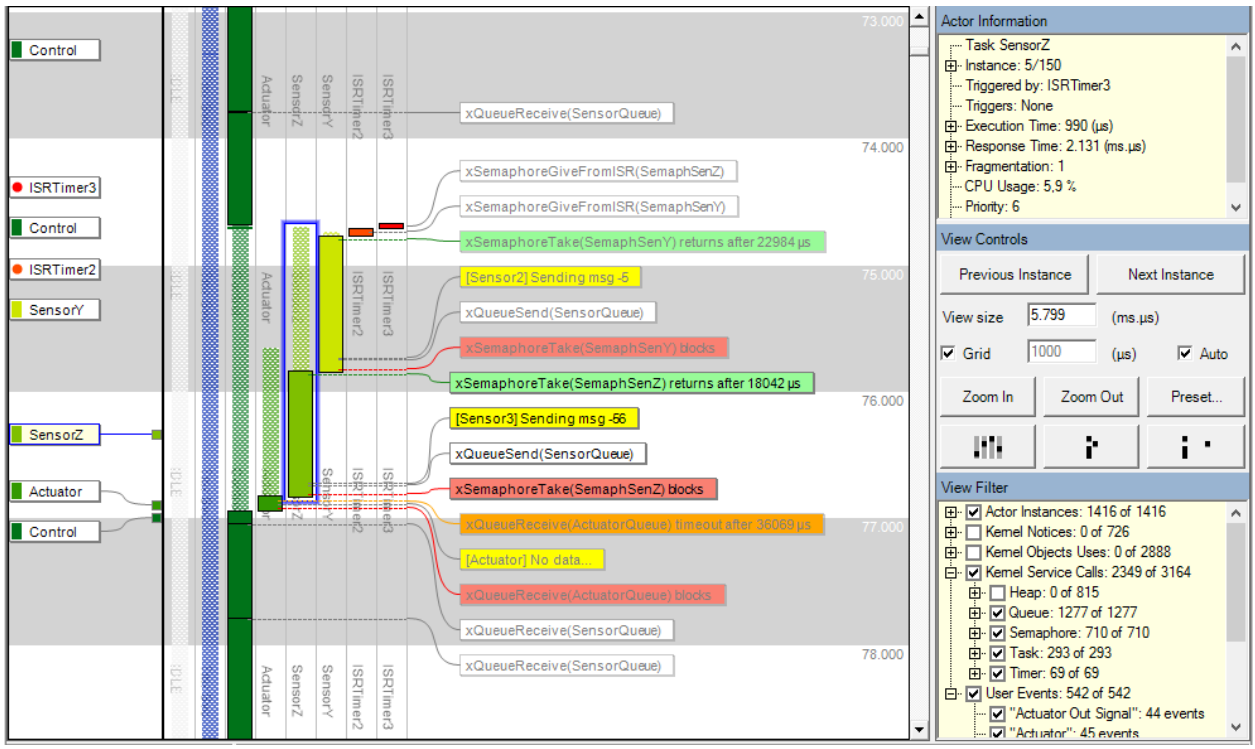
The visualization tool includes more than 25 interactive views that reveal many aspects of the real-time behavior, leveraging both RTOS events, memory allocation events and custom User Events in the application code. Visualizations range from detailed graphical traces and text logs, to custom data plots, task statistics, and dependency graphs. All views are interconnected in clever ways, allowing you to drill down from anomalies in high level views to the specific events, and easily switch perspectives while maintaining focus on the issue at hand.

Percepio AB
Köpmangatan 1A
72215 Västerås
SWEDEN
www.percepio.com

Tracealyzer is a stand-alone tool, that can be used run in parallel with an existing debugger. It does not require any particular hardware and works on essentially any processor, assuming a supported RTOS is used. Tracing is possible either via continuous trace streaming, or using a RAM buffer. The former allows for tracing your system over long durations, while the latter allows traces to be collected without debugger connection, during field testing or deployed operation.

Some customers even have Tracealyzer enabled by default in production code and collect traces remotely. This gives their developers detailed diagnostics of problems in field operation, that otherwise could be nearly impossible to reproduce.

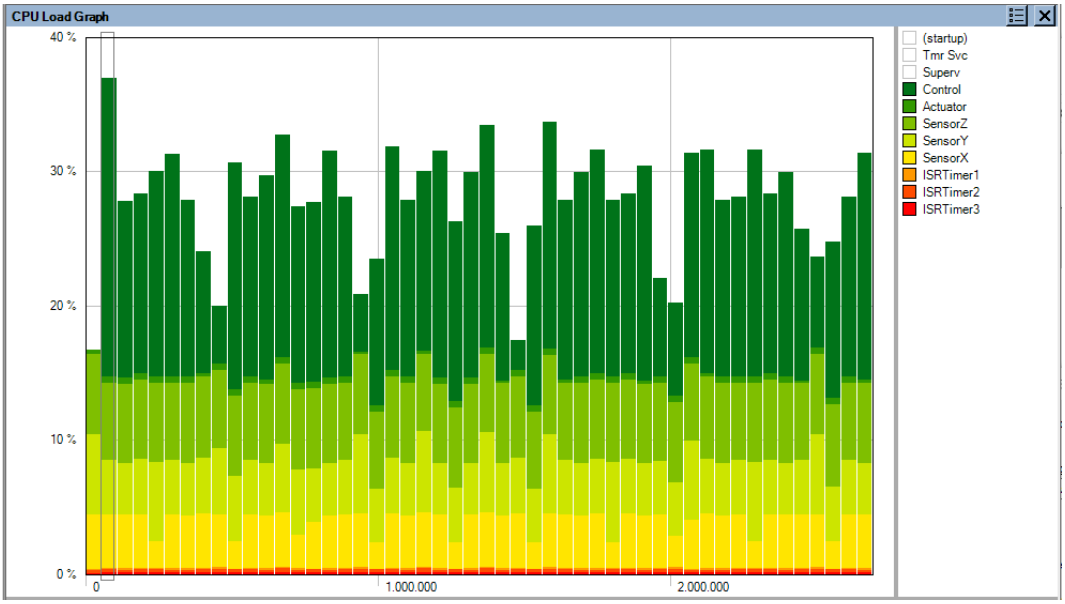
The **main trace view** visualizes the all data on a vertical time line, focusing on the execution of tasks, interrupt handlers and events, such as RTOS API calls and your own custom User Events. The event labels are color coded depending on the type and status of the operation. Each task and interrupt handler are assigned a unique color based on its priority, used consistently in all views. Double-clicking on a task, ISR or event opens another view, showing related events.



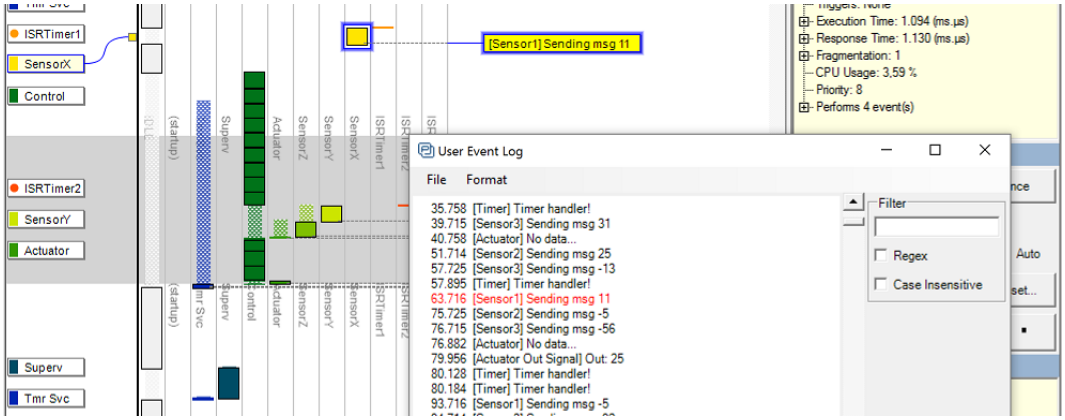
The **CPU Load Graph** (below) displays the amount of processor time used by each task and interrupt, on a horizontal timeline. This gives an overview of the trace, and can be used to focus the main trace view, simply by double clicking in the graph.

Percepio AB
Köpmangatan 1A
72215 Västerås
SWEDEN

www.percepio.com



User Events means explicit logging added into the application code, similar to the classic "printf" call but the processing is much faster, only some microsecond on most 32-bit MCUs. User Events are shown with yellow labels in the main view, and can be used to provide further detail about the application context, for debugging or performance analysis.

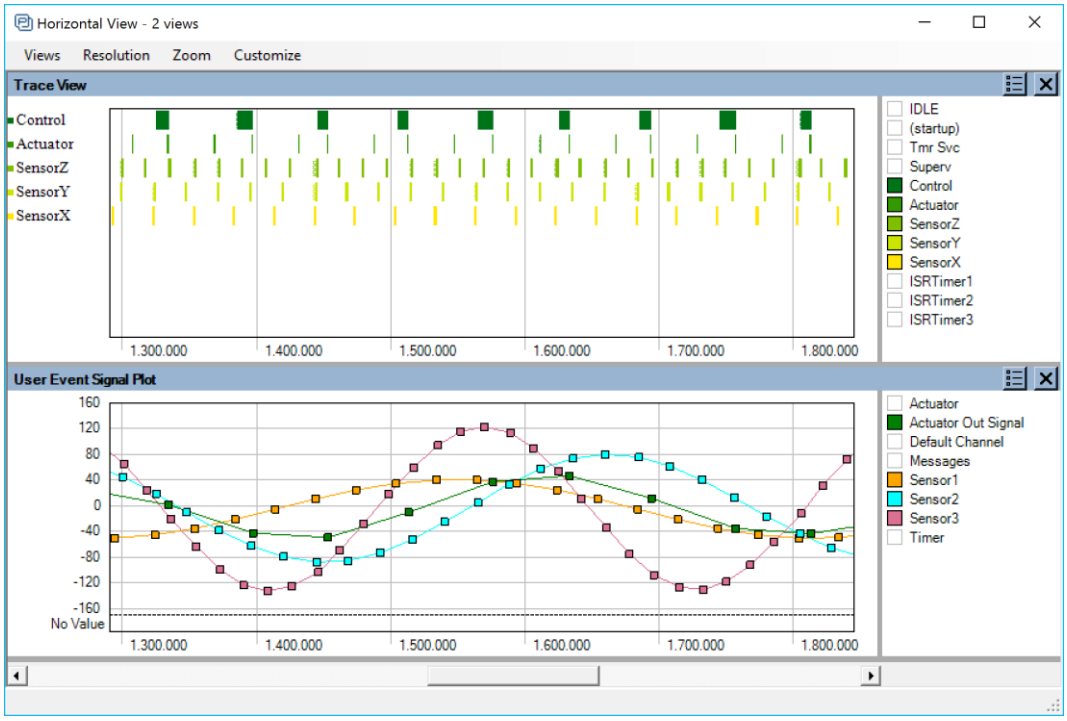


User Events are also listed separately in the **User Event Log**, that gives better overview. Like in most other views, each list entry is linked to the corresponding event in the main Trace View; double-clicking focuses the main view on the corresponding event.

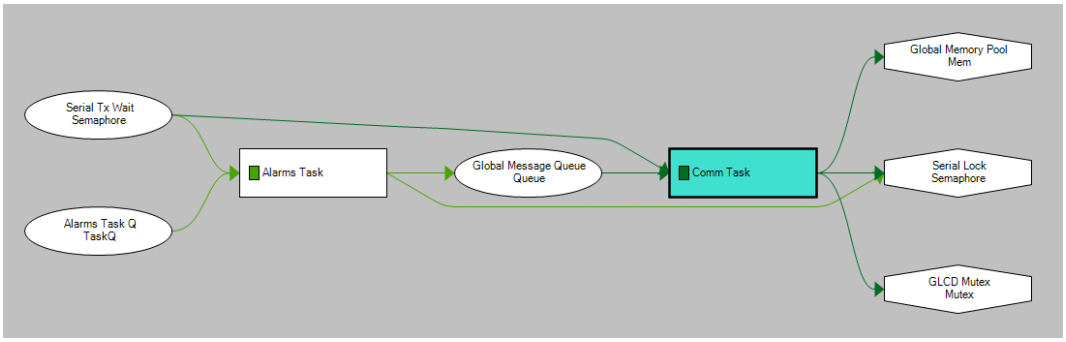
Percepio AB
Köpmangatan 1A
72215 Västerås
SWEDEN

www.percepio.com

The **User Event Signal Plot** allows you to plot data from User Events over a horizontal timeline. This can be used to visualize state variables or signals, e.g., in a control loop. In case you spot any anomaly, just double-click on the data point to focus the main trace view on this point. This way, you see the context of the data point and can find important clues to help solve the problem.



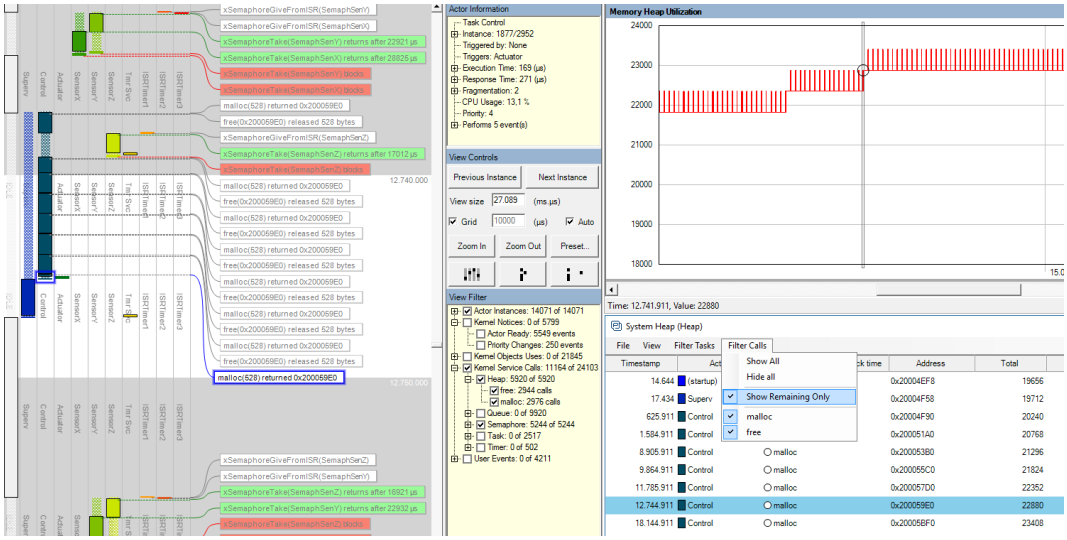
All horizontal views can be combined on a single timeline, i.e. with synchronized selections, zooming and scrolling, like in the above example. Many other horizontal views are available.



The **Communication Flow** graph is an RTOS dependency graph, showing what tasks that communicate and the kernel objects used. This provides a high-level view of the software design, based on a specific trace - a whole trace, or for a specific interval only. Any unexpected or unsound dependencies become obvious in this view. Like in the main trace view, double-clicking on a kernel object node opens a list related events, like all operations on the selected message queue.

Percepio AB
Köpmangatan 1A
72215 Västerås
SWEDEN
www.percepio.com

Dynamic memory usage can also be analyzed. Even though generally considered a bad practice within embedded systems, dynamic allocation is sometimes necessary. And in that case, you should make sure it works the way you intended. Tracealyzer allows you to analyze the allocations and even provides a filter that makes it easier to spot memory leads.



Tracealyzer is available for leading RTOSes and well as for Linux systems. It supports most 32-bit processors out-of-the box, including ARM, and it is easy to port for other architectures. See our website https://percepio.com for further information, or contact support@percepio.com.

Conclusion

The RTOS trend is quite apparent in the embedded industry and for good reasons. Due to increasingly complex and connected applications, more and more developers rely on an RTOS. RTOSes can be regarded as the third generation of firmware development, since multithreading brings a higher level of abstraction and less detailed control over the execution. This has significant pitfalls that call for better debugging support at the RTOS level. Common debugging tools have not, however, evolved significantly in response to the RTOS trend.

The debugging of RTOS-based systems can be simplified with better insight into their real-time execution. This requires RTOS-level tracing, where visualization is crucial to make sense of the data. Several tools can display an RTOS trace as a horizontal Gantt chart, but this is not ideal. More sophisticated visualization is both possible and available, optimized for RTOS traces, that makes it easier to understand the runtime system, spot important issues, and verify the solutions.

Percepio Tracealyzer is the leading solution for this purpose, enabling better insight, higher quality and faster development. Download Tracealyzer today and see the runtime world!

Percepio AB
Köpmangatan 1A
72215 Västerås
SWEDEN

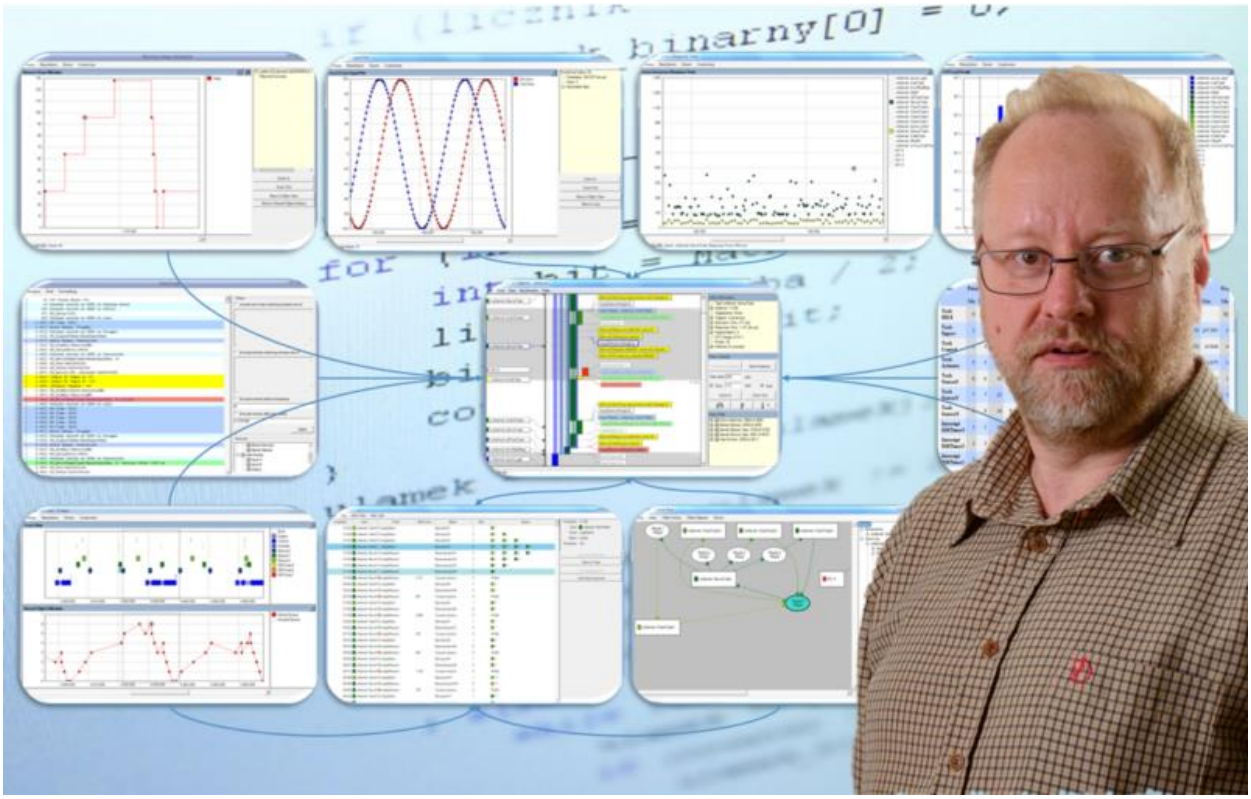
www.percepio.com

Appendix A: Getting Started with Percepio Tracealyzer

Download the Tracealyzer version matching your RTOS at https://percepio.com. The installer also includes a pre-recorded demo trace that allows you to explore the visualizations directly. And for a proper test drive, you may register for a free evaluation license.

If your host computer is running something other than Windows, download the .tgz archive, extract the files and run the executable using Mono (http://www.mono-project.com/). Mono is an open source .NET framework, sponsored by Microsoft. Tracealyzer v3.x officially supports *Linux* by using Mono. Official Mac support is planned for v4.0.

To make your own recordings, follow the instructions in the User Manual for your specific Tracealyzer version. Learn more about Tracealyzer at https://percepio.com/gettingstarted



Want some help getting started? We offer a free video conference session with Niclas Lindblom, Sr. FAE at Percepio AB. This is not a sales presentation, but personal "hands-on" technical session for up to 45 minutes, using your own hardware and tools.

Contact support@percepio.com to book a time!

Percepio AB
Köpmangatan 1A
72215 Västerås
SWEDEN
www.percepio.com