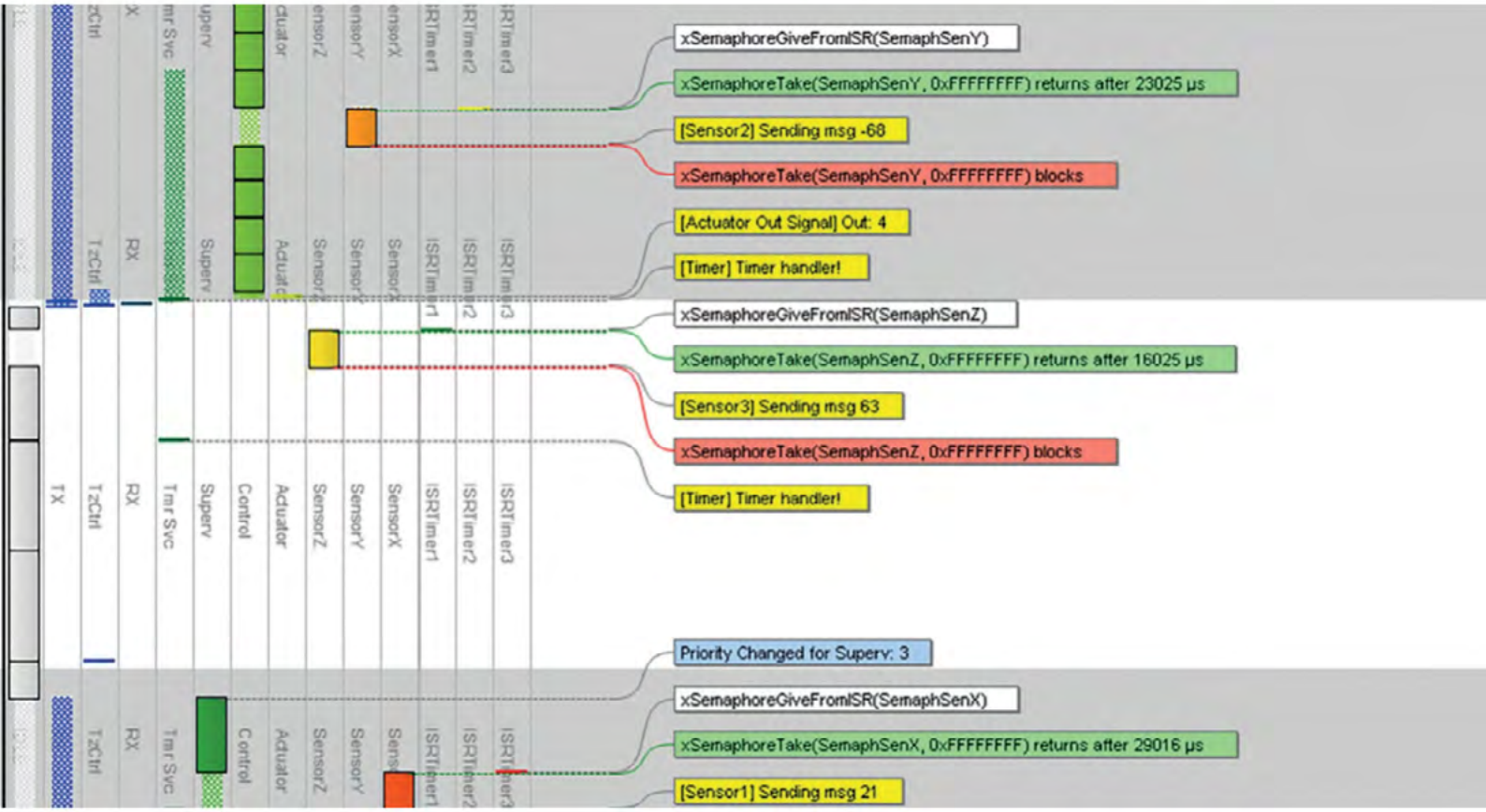
HighIntegritySystems

WITTENSTEIN



Understand your Real Time Application

With the help of Tracealyzer

WITTENSTEIN high integrity systems
Americas: +1 408 625 4712
ROTW: +44 1275 395 600

Email: sales@high**integrity**systems.com
Web: www.high**integrity**systems.com

HighIntegritySystems

WITTENSTEIN

Tracealyzer for SAFE**RTOS**

Using Tracealyzer in Your Application

Tracealyzer gives an unprecedented level of insight into the runtime world of SAFE**RTOS**-based embedded software. Solve complex problems in a fraction of the time that would otherwise be needed, develop more robust designs to prevent future problems and find new ways to enhance your software's performance. This improved understanding allows you to increase the overall software quality and reduce the time spent troubleshooting.

When working with a Real-Time Operating System (RTOS), you have probably noticed that a traditional debugger is often inadequate for many types of problems. Your debugger will show you the current system state when stopping on a break-point, but how did the system reach this state? What does the task timing look like, how much of the CPU time was used, and by what tasks? And what is actually going on in the run-time system?

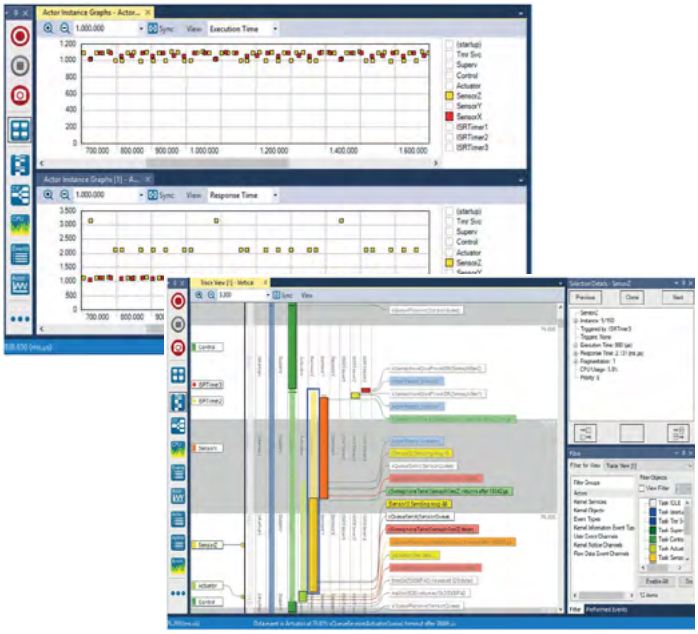
Tracealyzer:

Records the runtime behavior of SAFE**RTOS**, logs selected events from your application and explains the data through more than 30 graphical views, that are interconnected and easy to navigate.

Tracealyzer shows you the "big picture", with high-level views that complement the detailed debugger view. The 20+ innovative views are designed to be intuitive to read – by clicking in one view you can open or focus another view on the clicked location.

Tracealyzer Is best experienced live, so contact one of our sales representatives here and request your 30 day SAFE**RTOS** evaluation package with Tracealyzer.

Tracealyzer relies on a trace recorder library for SAFE**RTOS**. The recorder library is delivered in C source code. Customers who purchase SAFE**RTOS** and Tracealyzer are supplied two versions of the SAFE**RTOS** code. One version which has been instrumented with the trace recorder library for use in development and another version of SAFE**RTOS** without the trace library for production.



30 Day Evaluation License

Evaluate Tracealyzer with a free 30 day evaluation license.

Please contact one of our representatives for pricing information or to request your free 30 day evaluation package.

Contact Our Sales Representatives Directly

For Europe & Asia Pacific

Salomea Paprotny
+44 1275 395 600
Salomea.Paprotny@wittenstein.co.uk
www.highintegritysystems.com/contact

For Americas

Dave Vandenberg
+1 408 625 4712
D.Vandenberg@highintegritysystems.com
www.highintegritysystems.com/contact

Tasks, Priorities and Analysis

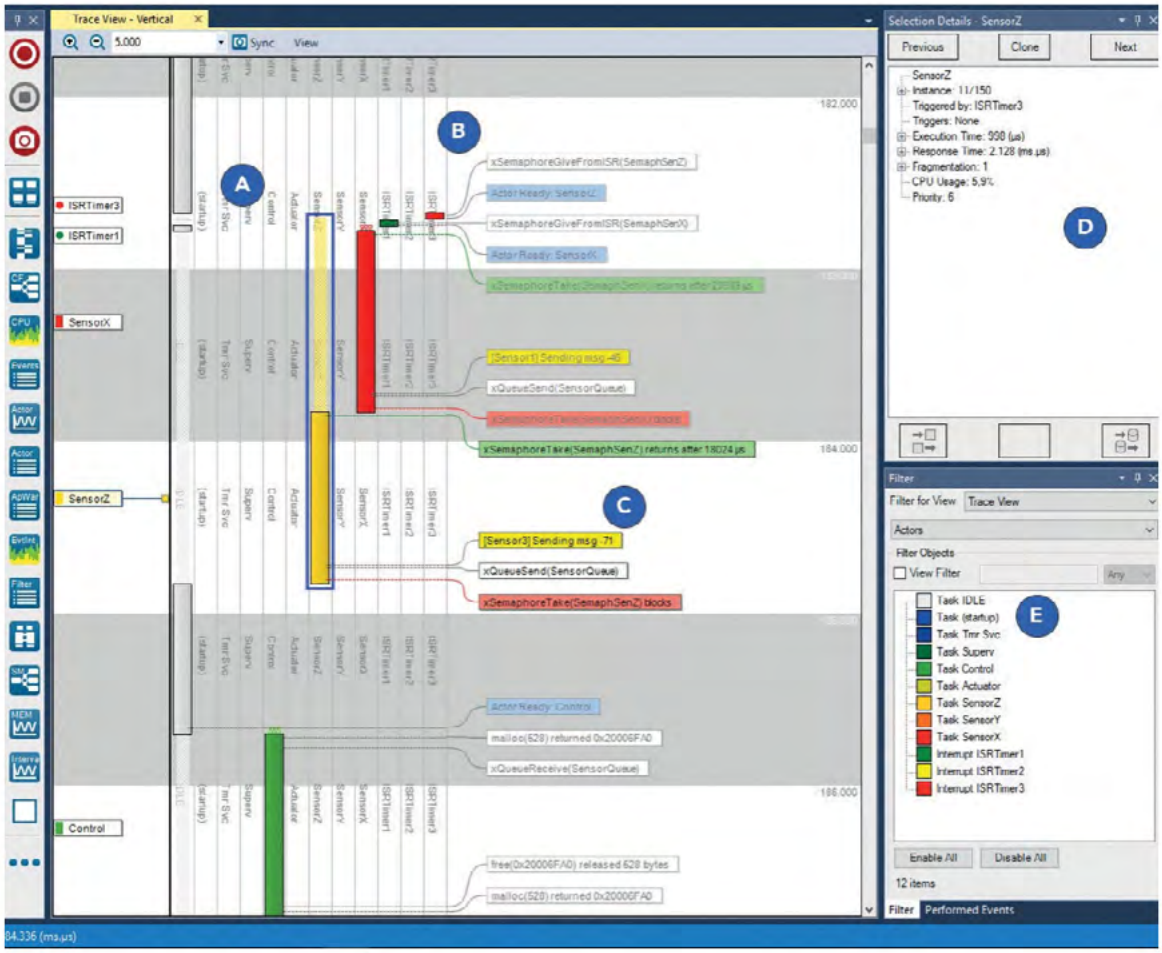


Figure 1: Tracealyzer showing RTOS task scheduling and calls to RTOS services.

The use of a Real-Time Operating System (RTOS) is increasingly common in embedded software designs, as an RTOS makes it easy to divide your code into smaller blocks, tasks, which execute seemingly in parallel and independent of each other. An RTOS provides multi-tasking, in a reliable and maintainable manner, which makes it easier to design applications with multiple concurrent functions such as control, communication and HMI.

The overhead of an RTOS is negligible on modern 32-bit processors and is often more than compensated for by more efficient designs enabled by multi-tasking.

Priority decides scheduling

An RTOS typically implements pre-emptive multi-tasking using a periodic interrupt routine (the “tick” interrupt) that switches the running task when required. The decision of what task to execute is known as task scheduling and most RTOS use fixed-priority scheduling (FPS), where the developers assign each task a static priority level to indicate their relative urgency. The RTOS scheduler always chooses the task with highest priority from the tasks currently ready to execute. This is a quite simple and elegant solution that allows the RTOS scheduler to be very small, highly optimized and thoroughly validated.

It is however important to assign suitable task priorities, otherwise the system performance will suffer or the system might even become unresponsive. This is because high priority tasks may prevent lower priority tasks from executing if they consume too much processor time.

Tasks, Priorities and Analysis

Analysing task priorities and runtime behaviour of RTOS based applications requires recording and visualisation of the task scheduling. For this purpose Percepio offers the Tracealyzer tools with over 25 interactive views that make the recorded traces easier to comprehend and analyse.

Task scheduling in Tracealyzer

Figure 1 shows the main view of Tracealyzer, a vertical timeline focused on the execution of tasks and interrupt handlers (A) annotated with text labels showing events (B) including RTOS API calls and custom “user events” (C). The “Selection Details” panel (D) shows properties of the highlighted task and the “View Filter” (E) allows for filtering of the display. Double-clicking on task fragments or event labels opens other related views showing related points in the trace, e.g., a chronological list of all executions of a selected task.

The response time of a task, i.e., the time from activation until completion, is affected not just by the actual processor time used by the task itself (execution time), but also by higher priority tasks and interrupts that pre-empt the task, as illustrated in Figure 1. So if the response time is too long, optimising the code of the problematic task might be a waste of time, unless you know what actually causes the long response time.

Execution time versus response time

With Tracealyzer you get many perspectives of the runtime world, including plots of task execution times and response times like in Figure 2 below. We can see that execution times are pretty steady for both tasks, but sometimes the response time of “SensorZ” is much higher. By clicking on such a data point, you open the corresponding interval in the main trace view (Figure 1) and see the cause. All views in Tracealyzer are interconnected in similar ways.

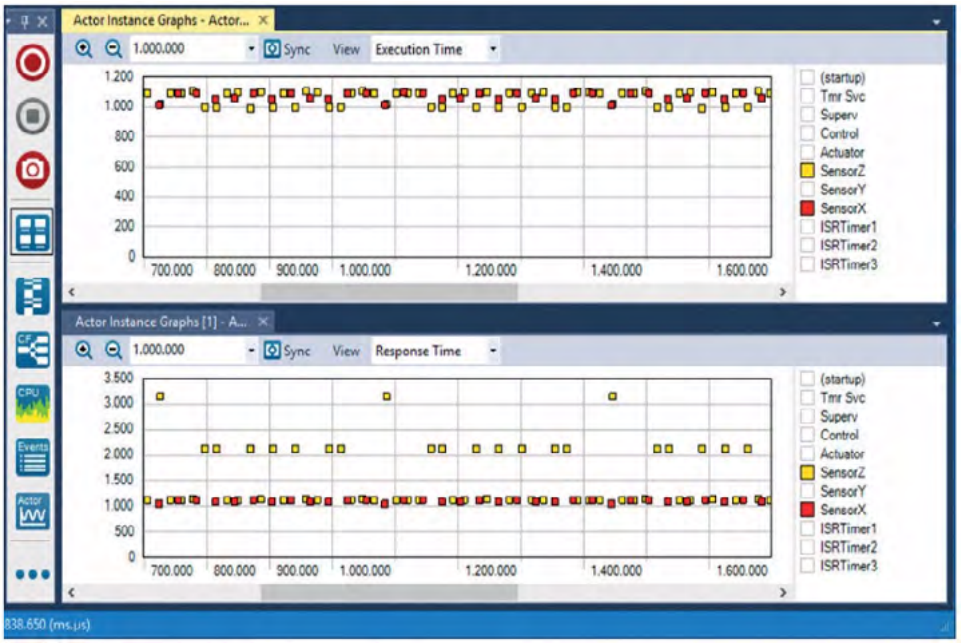


Figure 2: Execution time and response time for each execution of two tasks.

WITTENSTEIN high integrity systems
Americas: +1 408 625 4712
ROTW: +44 1275 395 600

Email: sales@high**integrity**systems.com
Web: www.high**integrity**systems.com

Semaphores and Queues

An RTOS makes it easy to divide your code into smaller blocks, tasks, which execute seemingly in parallel and independent of each other.

Having fully independent tasks is rarely possible in practice. In many cases, tasks need to be activated on a particular event, e.g., from an interrupt service routine or from another task requesting a service. In such cases, tasks often need to receive related input, i.e., parameters. Moreover, tasks often need to share hardware resources such as communication interfaces which can only be used by one task at a time, i.e. mutual exclusion, a type of synchronisation.

Some common synchronization objects:

Semaphore:
A signal between tasks/interrupts that does not carry any additional data. The meaning of the signal is implied by the semaphore object, so you need one semaphore for each purpose. The most common type of semaphore is a binary semaphore, that triggers activation of a task. The typical design pattern is that a task contains a main loop with an RTOS call to “take” the semaphore. If the semaphore is not yet signalled, the RTOS blocks the task from executing further until some task or interrupt routine “gives” the semaphore, i.e., signals it.

Mutex:
A binary semaphore for mutual exclusion between tasks, to protect a critical section. Internally it works much the same way as a binary semaphore, but it is used in a different way. It is “taken” before the critical section and “given” right after, i.e., in the same task. A mutex typically stores the current “owner” task and may boost its scheduling priority to avoid a problem called “priority inversion”.

Counting Semaphore:
A semaphore that contains a counter with an upper bound. This allows for keeping track of limited shared resources. Whenever a resource is to be allocated, an attempt to “take” the semaphore is made and the counter is incremented if below the specified upper bound, otherwise the attempted allocation blocks the task (possibly with a timeout) or fails directly, depending on the parameters to the RTOS semaphore service. When the resource is to be released, a “give” operation is made which decrements the counter.

Queue:
A FIFO buffer that allows for passing arbitrary messages to tasks. Typically, each queue has just one specific receiver task and one or several sender tasks.

Queues are often used as input for server-style tasks that provide multiple services/commands. A common design pattern in that case is to have common data structure for such messages consisting of a command code and parameters, and use a switch statement in the receiver task to handle the different message codes. If using a union structure for the parameters, or even just a void pointer, the parameters can be defined separately for each command code.

Most RTOS provide many types of mechanisms for safe communication and synchronisation in between tasks and between interrupt routines and tasks. Task H (as illustrated in Figure 3) blocks until the mutex is available, and is often not a problem in itself since a mutex is typically only held for brief durations during a critical section.

WITTENSTEIN high integrity systems
Americas: +1 408 625 4712
ROTW: +44 1275 395 600

Email: sales@high**integrity**systems.com
Web: www.high**integrity**systems.com

Semaphores and Queues

Inheritance:
However, as illustrated in Figure 3, the blocking may become a lot longer if an unrelated medium-priority task ("Task M") comes in and pre-empts Task L, thereby delaying the release of the mutex that Task H is waiting for. This phenomenon is called Priority Inversion.

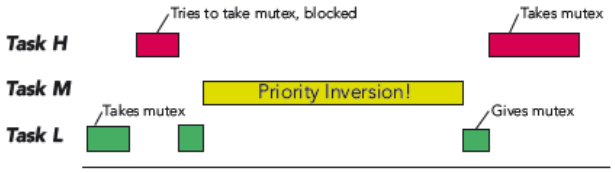


Figure 3. In principle, a high-priority task ('H' above) should never be blocked by lower-priority tasks. In practice, certain design decisions can result in this happening anyway, a condition known as Priority Inversion.

Priority inversion:
Priority Inversion is what caused NASA problems on the Mars Pathfinder mission. This means that a higher priority task is accidentally delayed by a lower priority task, which normally is not possible in an RTOS using Fixed Priority Scheduling. This may however occur, e.g., if the high-priority task ("Task H") needs to take a mutex that is currently held by a lower priority task ("Task L").

Most RTOS provide mutexes with "Priority Inheritance" (or other similar protocols) which raises the scheduling priority of the owner task if another, higher priority tasks becomes blocked by the mutex, which avoids interference from medium-priority tasks. Priority Inversion can also occur with queues and other similar primitives.

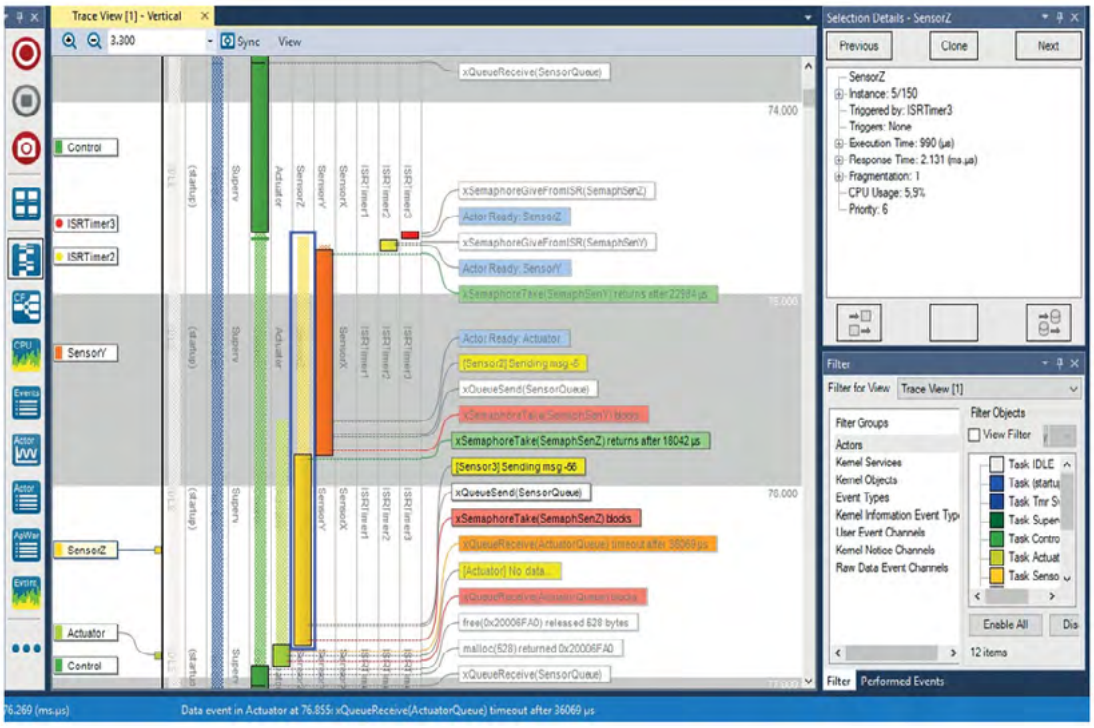


Figure 4. Tracealyzer displays most RTOS calls, including operations on semaphores, mutexes and queues, in the main timeline.

WITTENSTEIN high integrity systems
Americas: +1 408 625 4712
ROTW: +44 1275 395 600

Email: sales@highintegritysystems.com
Web: www.highintegritysystems.com

Semaphores and Queues

Tracealyzer allows you to see most RTOS calls made by the application, including operations on queues, semaphores and mutexes, in the vertical timeline of the main trace view, in parallel with the task scheduling, interrupts, and logged application events – see Figure 4 on the previous page.

Revealing history:
By clicking on any semaphore, queue or mutex event in the main trace view, you open up the Kernel Object History view for the selected object, as illustrated in figure 5, showing a separate timeline with all operations and states of this specific object. You can double-click in this view to find the corresponding event in the main trace view.

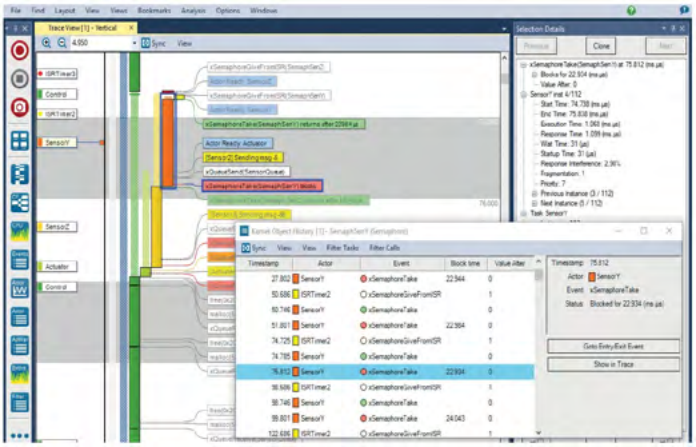


Figure 5. Double-clicking an event from the main trace view brings up the History view for the corresponding object.

For queue objects, you also get a visual display of the number of messages in the buffer at any point, and you can even track messages from send to receive or vice versa. For mutex objects you see the name of the current owning task.

Tracealyzer also provides an overview of the interactions between tasks and interrupts via kernel objects such as queues, semaphores and mutexes. This gives a high-level illustration of the runtime architecture based on the trace, and you can even generate this for specified intervals in the trace. An example is shown below, figure 6. Rectangles indicates tasks and interrupts, while ellipses indicate queues or semaphores. Mutexes are shown as hexagons. Since sometimes binary semaphores are used as mutexes, the classification of Mutexes are made based on their usage pattern, so semaphore objects may also be displayed with hexagons if they are used like a mutex, i.e., taken and given by the same task.

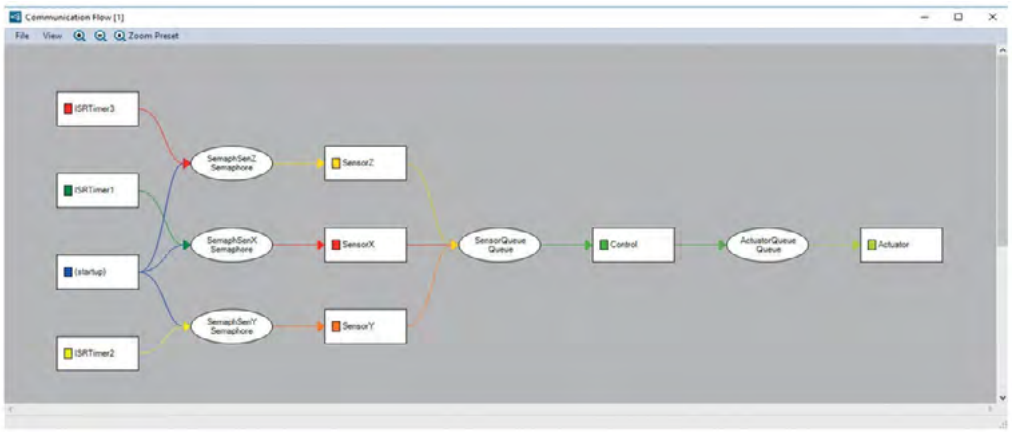


Figure 6. The Communications Flow view can be a good place to start your debugging from, as it shows how messages are passed around within the application.

Performance Analysis with Tracealyzer

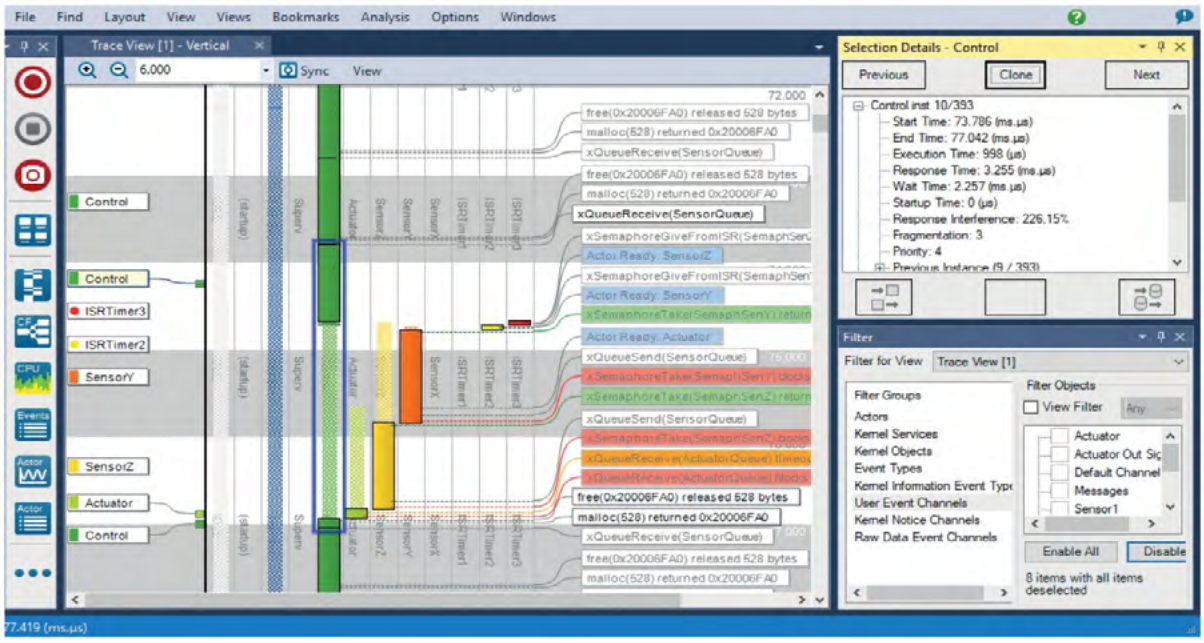


Figure 7. Actors, Instances and Fragments in Tracealyzer.

When developing firmware using a Real-Time Operating System (RTOS), how do you measure the software performance? One important aspect of performance analysis is response time, the time from point A to point B in the code, e.g., from when a task is activated until it is completed. This can be measured in many ways, e.g., by toggling an I/O pin and measuring with a logic analyzer, or by adding some extra code that measures the number of clock cycles between the two points. But a basic measurement like this only measures the total amount of processor time between these points, without any information about contributing factors, such as interrupts routines or other tasks that interfere due to pre-emptive scheduling (see Tasks, Priorities and Analysis, page 2).

Another important performance aspect of performance analysis is execution time, the actual processor time used by a particular piece of code. You might use solutions that samples the program counter and provides a high level overview of those using the most processor time. This is supported by several common IDEs and most ARM-based MCUs provide hardware support for this purpose. This is however an average measurement of the typical distribution and is inaccurate for less frequent functions or tasks. Moreover, this does not reveal sporadic cases of unusually long executions that might cause problems such as timeouts.

Tracing with RTOS knowledge:
To get an exact picture of the RTOS behavior you need a solution for RTOS- aware tracing. Tools for this purpose have been around for many years, but only for certain operating systems and each tool typically only support a particular operating system. They typically display a horizontal Gantt chart showing task execution over time. This is however not ideal for RTOS traces as it is hard to show other events in parallel, such as RTOS API calls.

The main view of Tracealyzer (Figure 7, above) uses a vertical timeline, that allows for showing not only RTOS scheduling and interrupts, but also other events such as RTOS calls or custom User Events, using horizontal text labels. These labels “float” and spread out evenly to avoid overlaps. The rectangles in the scheduling trace correspond to intervals of uninterrupted execution. These are called “fragments” in Tracealyzer. The term “Actor” is used to denote all execution contexts in the traced system, such as tasks and interrupt handlers. The task scheduling can be rendered in different ways, or “View Modes”, with associated buttons found under the Zoom buttons. In this mode, the fragments are ordered in multiple columns, one for each Actor.

Performance Analysis with Tracealyzer

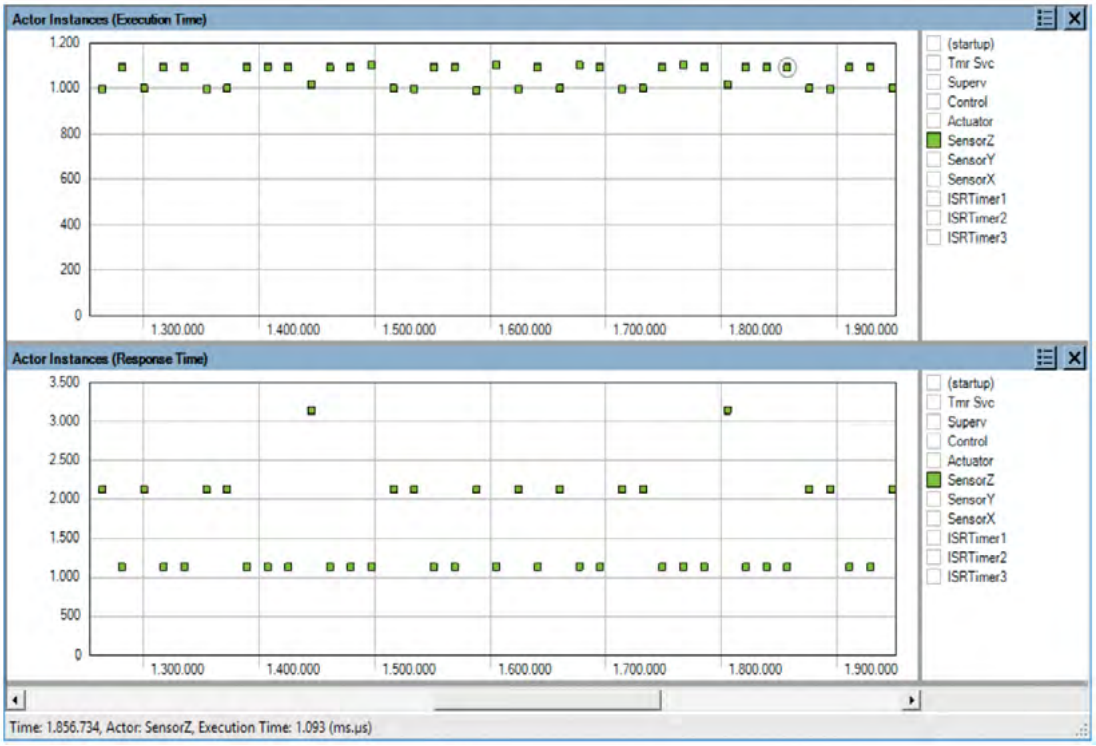


Figure 8. Plot showing variations in execution time (above) and response time (below), over time.

One actor, many instances

Tracealyzer has a concept of "instances" not found in other RTOS tracing tools, meaning a particular execution of an Actor, i.e., from when a "job" is triggered until it is finished. The instance concept is quite central in Tracealyzer, since instances are used both in the trace visualisation and for providing timing statistics. When clicking on the actor fragment in the Tracealyzer main view, the entire Actor Instance is highlighted with a blue rectangle as depicted in Figure 7.

Moreover, performance metrics such as execution time and response time are calculated for each instance and can be visualized as detailed plots showing the variations over time (Figure 8, above) and as histograms showing the distributions. The latter is shown in Figure 9 (right) where we can see that the highest response time of "Control Task" is 3255 µs in this trace, while the highest execution time is just 1087 µs, meaning that most of the response time is due to interference from other tasks or interrupts.

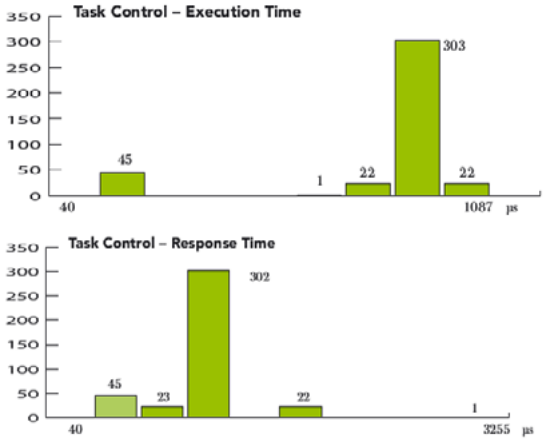


Figure 9. Distribution of execution times and response times (in µs) for the task instances.

WITTENSTEIN high integrity systems
Americas: +1 408 625 4712
ROTW: +44 1275 395 600

Email: sales@highintegritysystems.com
Web: www.highintegritysystems.com

Performance Analysis with Tracealyzer

It's all connected

All views in Tracealyzer are interconnected, so by clicking on the plotted data points or the histogram bars, you find the corresponding locations in the main trace view and can see the detailed RTOS behaviour behind the statistics.

Great, but how is the stream of task scheduling events grouped into task in- stances? This is fairly obvious for cyclic RTOS tasks, where an instance corresponds to an iteration of the main loop, delimited by a blocking RTOS call, e.g., a "QueueRecieve" or a "DelayUntil" somewhere in the loop. But a task might perform multiple such calls, so how does Tracealyzer know where to end the current instance and begin a new instance?

For this purpose, Tracealyzer has a concept of "instance finish events" (IFEs) that are defined in two ways. Users don't need to bother about this in most cases, as there is a set of standard rules that specify what RTOS calls that normally should be counted as IFEs, such as Delay calls and QueueRecieve calls. This requires no extra configuration and is usually correct. However, for cases these implicit rules are unsuitable, you may generate explicit events (IFEs) that marks the instance as finished, this by calling a certain function in our recorder library.

An example of this is shown in Figure 10 (below), where the dark green control task is divided into multiple instances despite no task-switches occurring at these points. This way you can manually decide how to group events into instances, and thereby control the interpretation of the timing statistics.

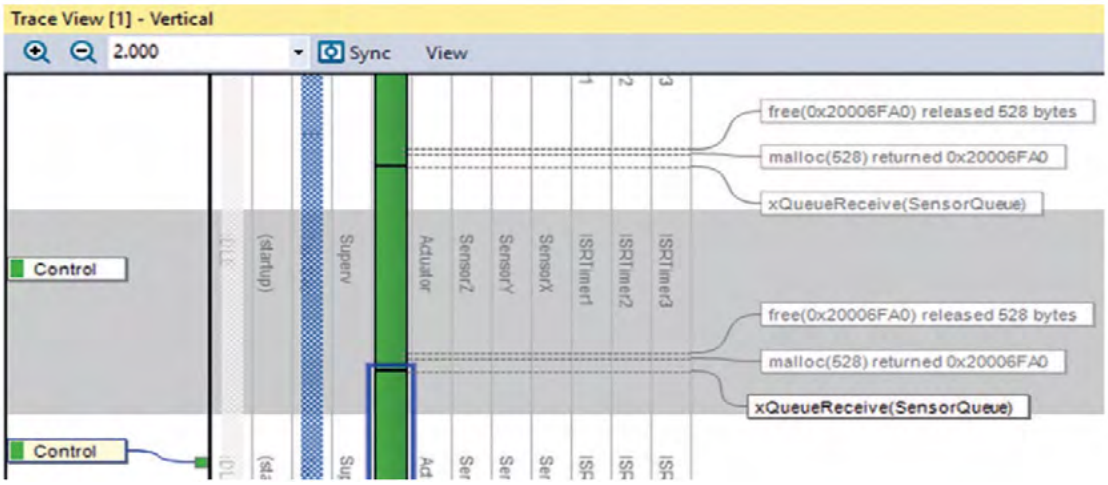


Figure 10. Instance Finish Events (IFE) allows you to define your own custom intervals.

Download Information

WITTENSTEIN high integritty systems (WHIS) supports the use of Percepio's Tracealyzer with SAFE**RTOS**®. You can download free demos of SAFE**RTOS** from our website: **www.highintegritysystems.com**

For more information don't hesitate to contact the WHIS Sales Representatives directly:

For Europe & Asia Pacific
Salomea Paprotny
+44 1275 395 600
Salomea.Paprotny@wittenstein.co.uk
www.highintegritysystems.com/contact

For Americas
Dave Vandenberg
+1 408 625 4712
D.Vandenberg@highintegritysystems.com
www.highintegritysystems.com/contact