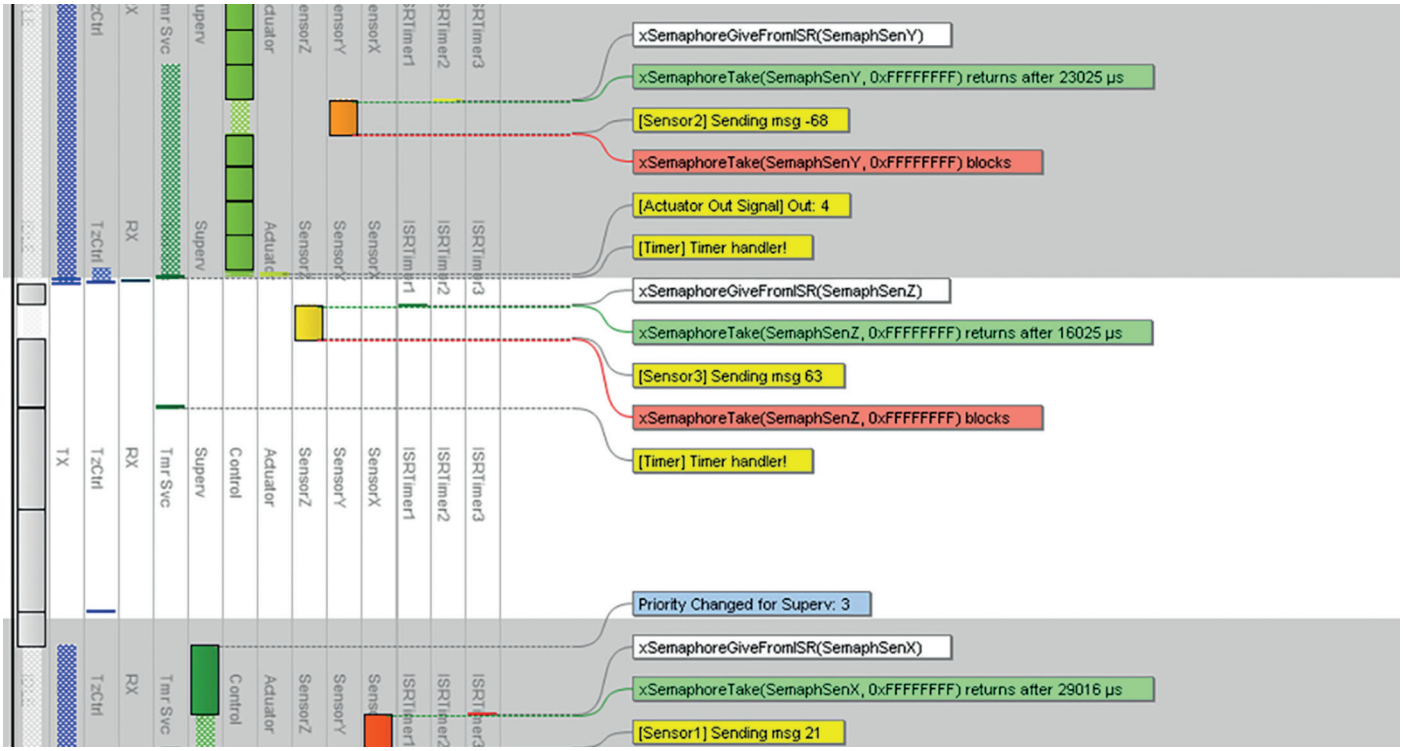


RTOS 101

Understand your real-time applications
with the help of Percepio Tracealyzer

percepio®

RTOS 101

Tasks, Priorities and Analysis

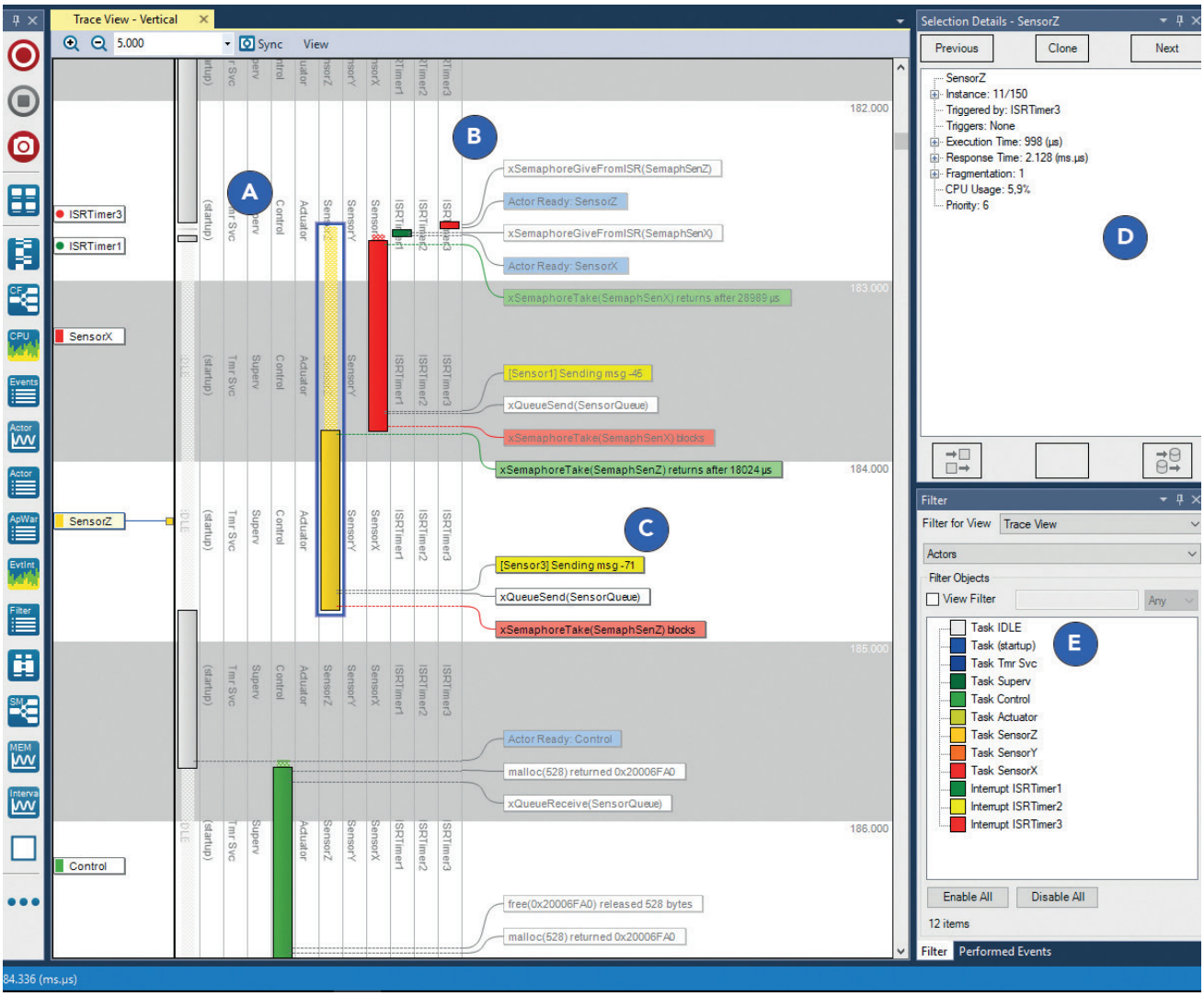


Figure 1: Tracealyzer showing RTOS task scheduling and calls to RTOS services.

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 preemptive 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 RTOSes use fixed-priority scheduling (FPS), where the developers assign each task a static priority level to indicate its relative urgency. The RTOS scheduler always chooses the task with highest priority from the tasks current-

ly 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.

Analyzing task priorities and runtime behavior of RTOS-based applications requires recording and visualization 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 analyze.

time), but also by higher priority tasks and interrupts that preempt the task, as illustrated in Figure 1. So if the response time is too long, optimizing the code of the problematic task might be a waste of time, unless you know what actually causes the long response time.

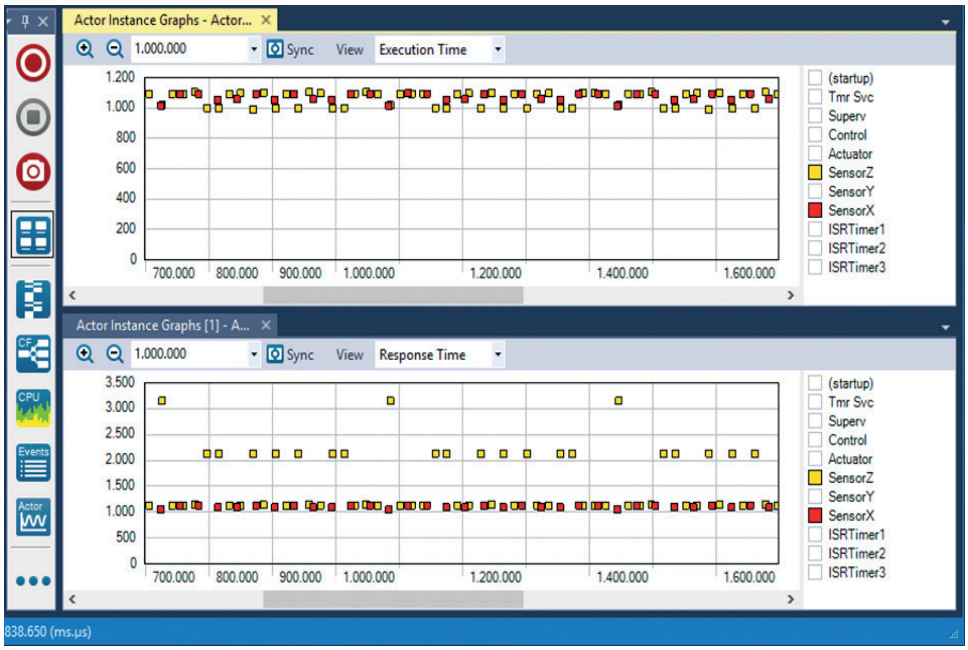


Figure 2: Execution time and response time for each execution of two tasks.

Task scheduling in Tracealyzer

Figure 1 (left) 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

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 above. 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.

RTOS 101

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, as described in the previous article in this series.

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 synchronization.

Thread-safe is tricky

Inexperienced developers may try to use global variables for such purposes, but implementing thread-safe communication is tricky and a home-cooked solution may fail if a task-switch strikes at a critical point.

For instance, consider this situation:

```
1: while (COM1_busy); // if busy,
   wait until free
2: COM1_busy = 1;
3: SendBytes(COM1, data);
4: COM1_busy = 0;
```

The initial loop (line 1) may seem to give exclusive access to the COM1 interface (line 3), but if using an RTOS this is often not a safe solution. It probably works most of the time, perhaps often enough to pass all testing, but if an interrupt strikes after the initial loop on line 1 but before the assignment at line 2 and this results in a task-switch, a second task could get into the critical section before the first task is finished.

Implementing a thread-safe critical section requires either disabling interrupts or using special instructions for atomic "test-and-set". Considering this, it is typically easier (and safer!) to use the RTOS services intended for this purpose.

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 signaled, 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", discussed below.

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 a 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.

pose. Most RTOSes provide many types of mechanisms for safe communication and synchronization in between tasks and between interrupt routines and tasks.

held by a lower priority task ("Task L"). This blocks Task H until the mutex is available, and is often not a problem in itself since a mutex is typically only held for brief periods during a critical section.

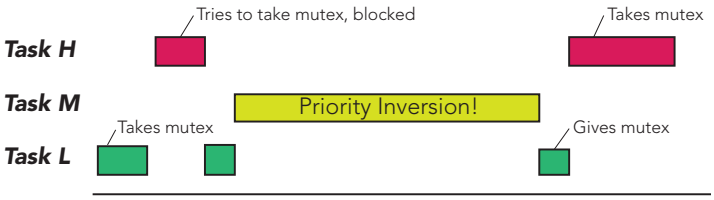


Figure 1. In principle, a high-priority task ('H' above) should never be blocked by lower-priority tasks. In practice, certain design decisions can result in this happening anyway, a condition known as Priority Inversion.

Inheritance

However, as illustrated in Figure 1 (left), the blocking may become a lot longer if an unrelated medium-priority task ("Task M") comes in and preempts Task L, thereby delaying the release of the mutex that Task H is waiting for. This phenomenon is called 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 RTOSes 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

Most RTOSes provide mutexes with "Priority Inheritance" (or similar) which raises the scheduling priority of the owner task if a 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, as described in *Customer Case: The mysterious watchdog reset* (see page 10).

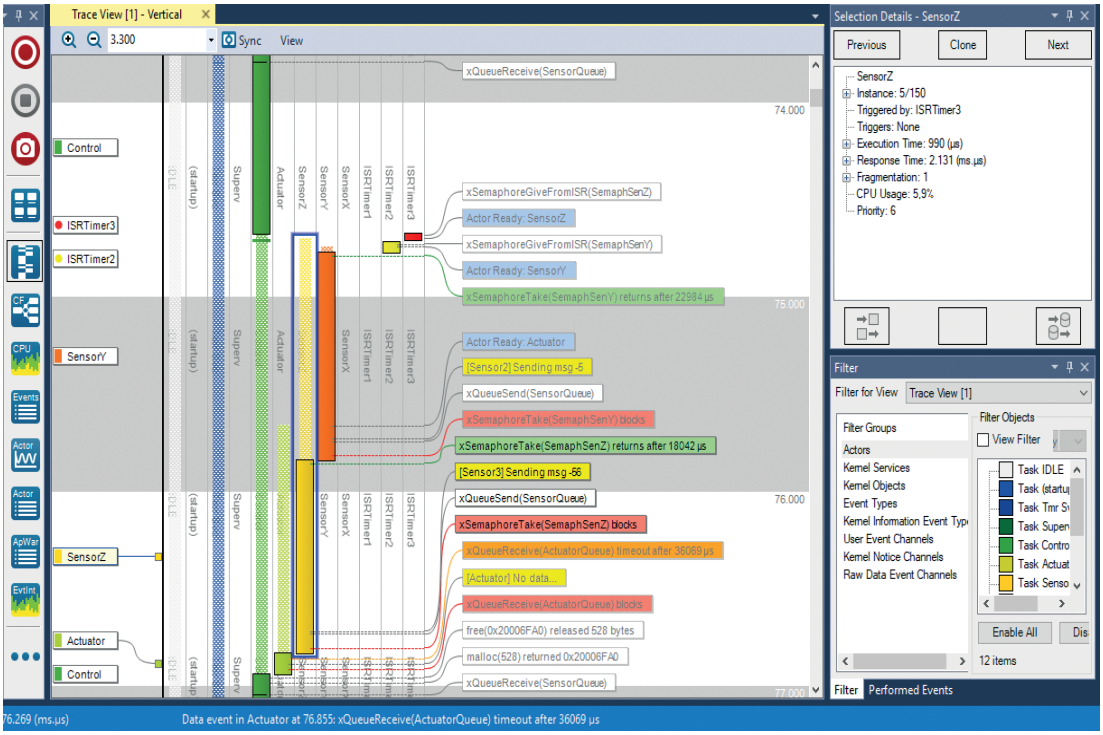


Figure 2. Tracealyzer displays most RTOS calls, including operations on semaphores, mutexes and queues, in the main timeline.

Percepio 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 2 on the previous page.

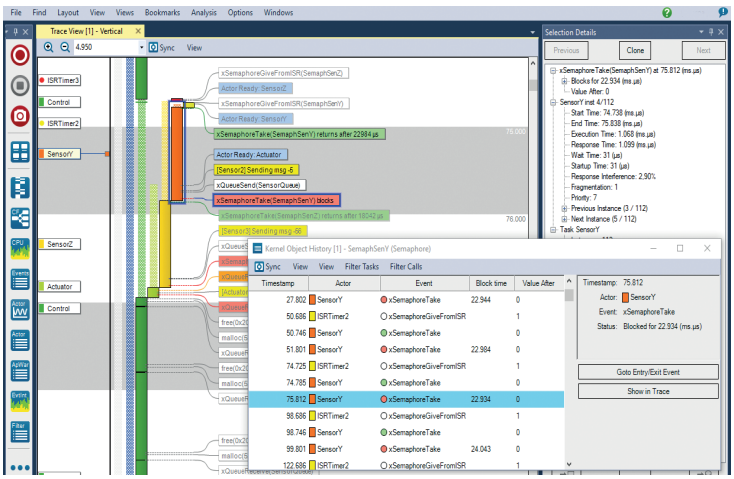


Figure 3. Double-clicking an event from the main trace view brings up the History view for the corresponding object.

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 above, 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.

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. Rectangles indicate tasks and interrupts, while ellipses indicate queues or semaphores. Mutexes are shown as hexagons. Since sometimes binary semaphores are used as mutexes, the Mutexes are classified 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.

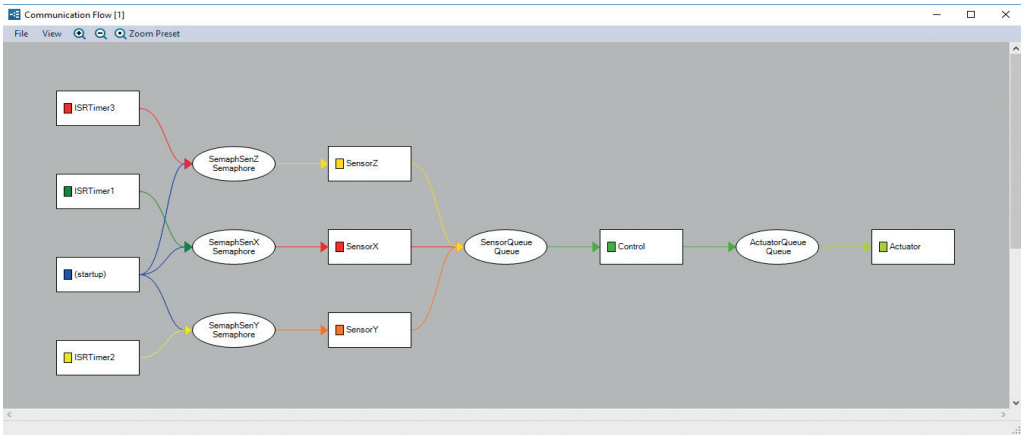


Figure 4. The Communications Flow view can be a good place to start your debugging from, as it shows how messages are passed around within the application.

Remember: implementing thread-safe communication is tricky.

RTOS 101

Performance Analysis with Tracealyzer

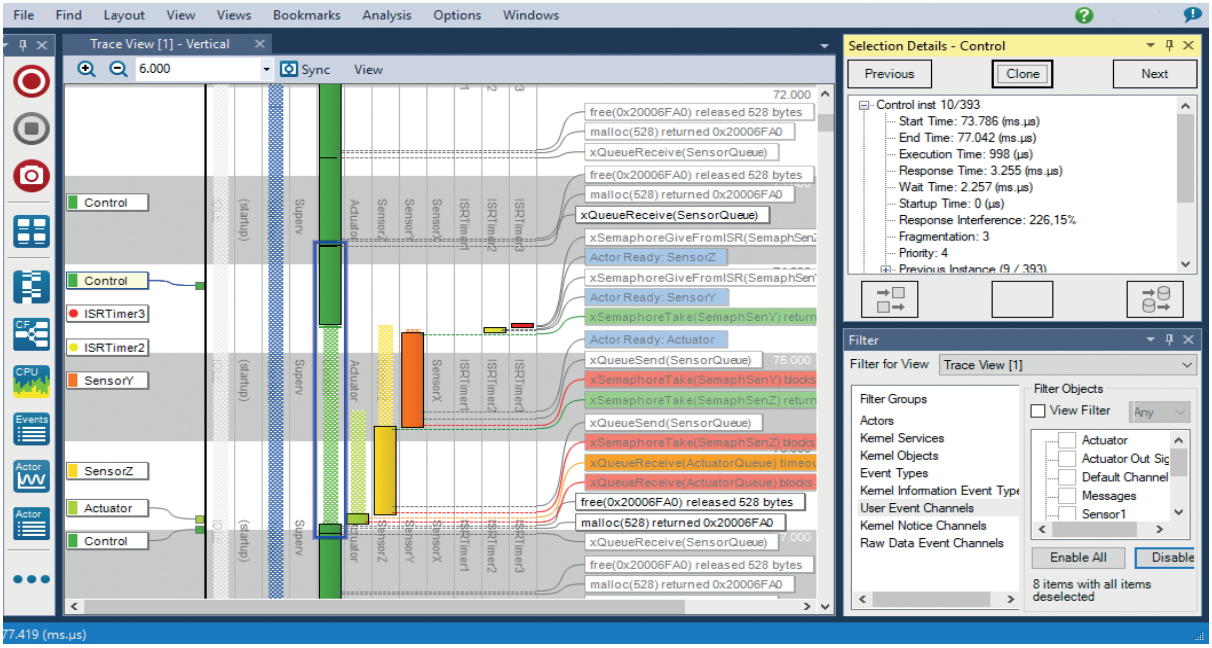


Figure 1. Actors, Instances and Fragments in Tracealyzer.

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 preemptive 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 a solution 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 supports 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.

Tracealyzer is available for several leading operating systems and provides a sophisticated visualization that makes it easier to comprehend the traces.

The main view of Tracealyzer (Figure 1, 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 exe-

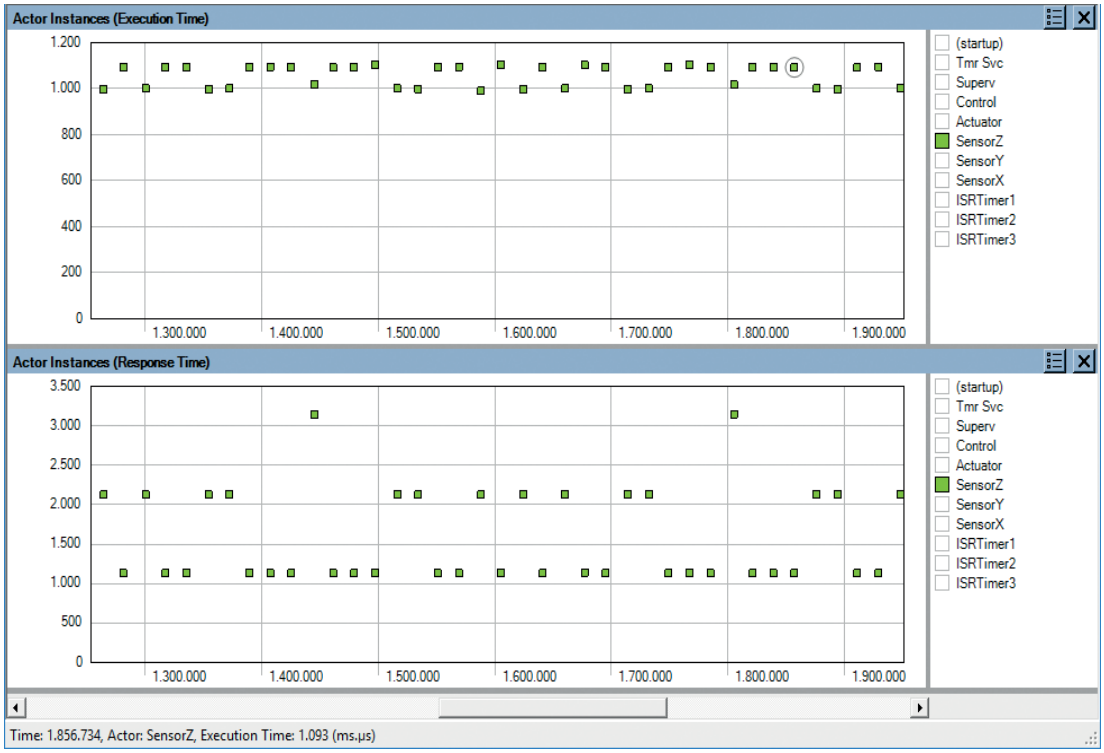


Figure 2. Plot showing variations in execution time (above) and response time (below), over time.

cution. 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.

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 visualization 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 1.

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 2 above).

The performance metrics can also be tabulated, as in the Actor Statistics view (Figure 3, below). For instance, we can see in the table that both average and maximum response times for the task Controller are slightly higher than the execution times; the difference is due to Controller sometimes being interrupted by other tasks, or by an interrupt service routine. A look at the fragmentation numbers back this up, as almost all instances of Controller have three fragments – i.e., they were interrupted twice.

It's all connected

All views in Tracealyzer are interconnected, so by clicking on the plotted

Actor	Count	CPU Usage	Execution Time			Response Time			Fragmentation		
			Min	Avg	Max	Min	Avg	Max	Min	Avg	Max
Supervisor	5	4.059	38	10.743	20.960	38	15.984	33.704	1	3	5
Tmr Svc	1	0.034	446	446	446	446	446	446	1	1	1
Controller	66	25.400	56	5.093	10.604	56	5.551	11.063	1	2.939	3

Figure 3. A tabulation of execution times and response times (in µs) for the selected tasks. The min and max values are clickable and links to the corresponding actor instance in the timeline.

data points or the tabulated values, you find the corresponding locations in the main trace view and can see the detailed RTOS behavior behind the statistics.

Great, but how is the stream of task scheduling events grouped into task instances? 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 "QueueReceive" 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 RTOS calls that normally should be counted as IFEs, such as Delay calls and QueueReceive calls. This requires no extra configuration and is usually correct. However, for cases where these implicit rules are unsuitable, you may generate explicit events (IFEs) that mark the instance as finished by calling a certain function in our recorder library. An example of this is shown in Figure 4 (above), 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.

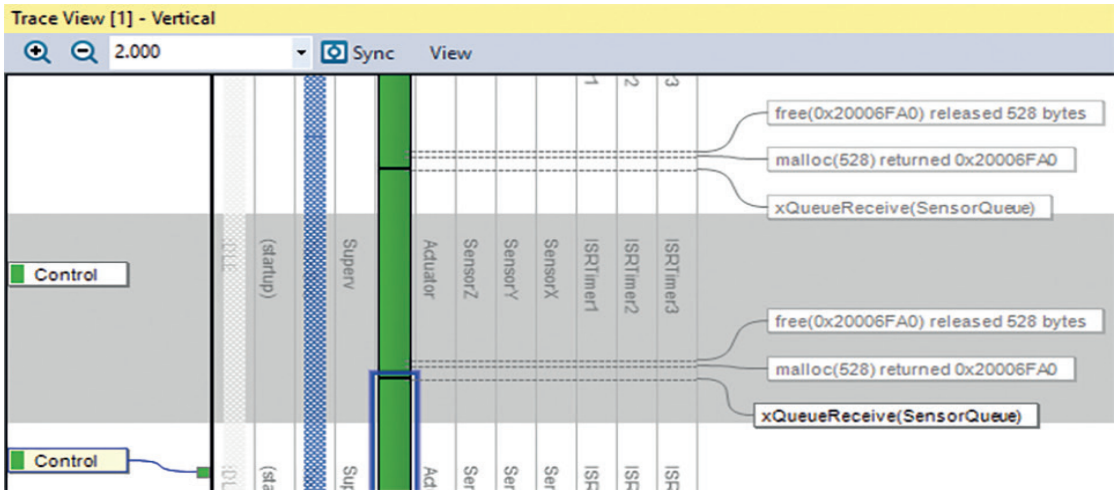


Figure 4. Instance Finish Events (IFE) allows you to define your own custom intervals.

Customer Case

Mysterious Watchdog Reset

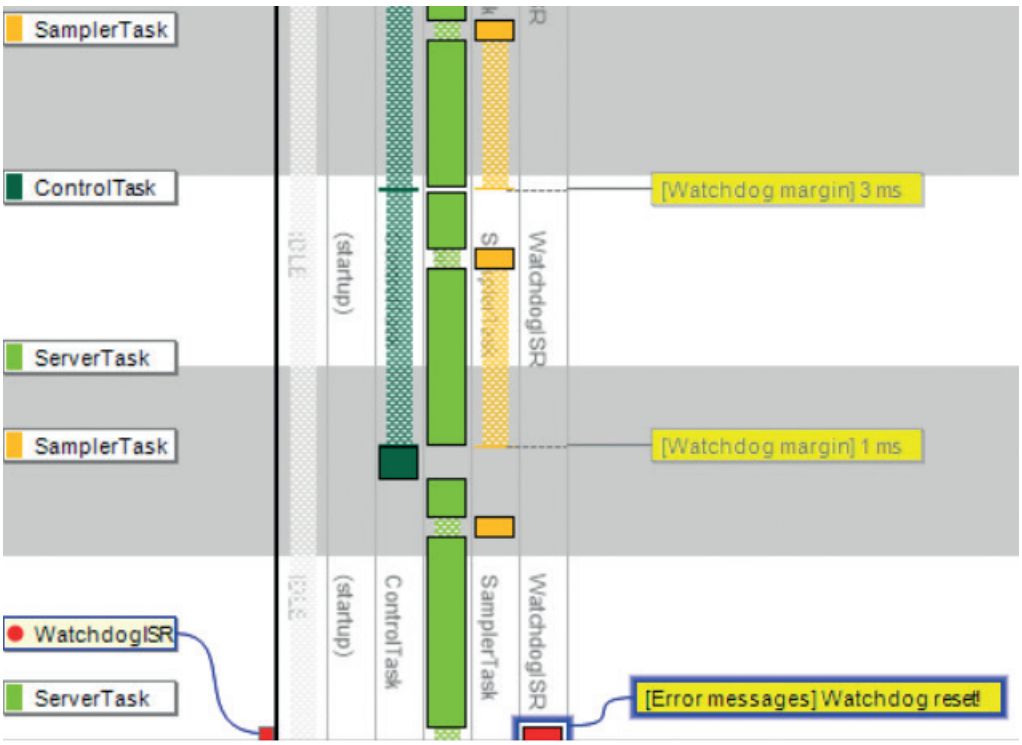


Figure 1. In Tracealyzer, a User Event (the yellow labels) are used to log and display user data, somewhat similar to classic printf() statements.

We collect examples of how Tracealyzer has been useful to our customers and have recreated similar issues to illustrate the benefits of our Tracealyzer tools for embedded software developers.

In this case, a customer had an issue with a randomly occurring reset. By placing a breakpoint in the reset exception handler, they figured out that it was the watchdog timer that had expired. The watchdog timer was supposed to be reset in a high priority task that executed periodically.

The ability to insert custom User Events comes handy in this case. They are similar to a classic "printf()" call and events have here been added when the watchdog timer was reset and when it expired. User events also support data arguments, and this has been used to log the timer value (just before resetting it) to see the watchdog "margin", i.e., remaining time. The result can be seen above, in the yellow text labels.

We can see that the SamplerTask is running, but it does not clear the watchdog timer in the last execution of the task, which resets the system after a while ("Watchdog reset!"). So why didn't SamplerTask reset the watchdog timer? Let's display Kernel Service calls to see what the task was doing.

The last event of SamplerTask is a call to xQueueSend, a kernel function that puts a message in a message queue. Note that the label is red, meaning that the xQueueSend call blocked the task, which caused a context-switch to ServerTask before the watchdog timer had been reset, which caused it to expire and reset the system.

Why the blocking?

So why was xQueueSend blocking the task? By double-clicking on this event label, we open the Object History View, showing all operations on this particular queue, "ControlQueue", as illustrated on the next page.

The rightmost column shows the buffered messages. We can see that the message queue already contains five messages and probably is full, hence the blocking. But the ControlTask is supposed to read the queue and make room, why hasn't this worked as expected?

To investigate this, we look at how the watchdog margin varies over time. We have this information in the user event

Figure 2. SamplerTask blocks on a send-to-queue operation.

Timestamp	Actor	Event	Block time	Status	Size	Queue
142.255	ControlTask	xQueueReceive		Received post #27	1	27 28
143.805	SamplerTask	xQueueSend		Sent post #29	2	28 29
144.684	ControlTask	xQueueReceive		Received post #28	1	28 29
147.391	ControlTask	xQueueReceive		Received post #29	0	29
148.805	SamplerTask	xQueueSend		Sent post #30	1	30
149.913	ControlTask	xQueueReceive		Received post #30	0	30
152.141	ControlTask	xQueueReceive	1.772	Trying to receive...	0	Empty
153.805	SamplerTask	xQueueSend		Sent post #31	1	31
153.913	ControlTask	xQueueReceive		Received post #31	0	31
156.295	ControlTask	xQueueReceive	19.307	Trying to receive...	0	Empty
158.815	SamplerTask	xQueueSend		Sent post #32	1	32
163.805	SamplerTask	xQueueSend		Sent post #33	2	32 33
168.805	SamplerTask	xQueueSend		Sent post #34	3	32 33 34
173.805	SamplerTask	xQueueSend		Sent post #35	4	32 33 34 35
175.602	ControlTask	xQueueReceive		Received post #32	3	32 33 34 35
177.664	ControlTask	xQueueReceive		Received post #33	2	33 34 35
178.805	SamplerTask	xQueueSend		Sent post #36	3	34 35 36
183.805	SamplerTask	xQueueSend		Sent post #37	4	34 35 36 37
188.805	SamplerTask	xQueueSend		Sent post #38	5	34 35 36 37 38
193.812	SamplerTask	xQueueSend		Trying to send...	5	34 35 36 37 38

Figure 3. The Object History view can tell us a lot about an RTOS object. In this case, a message queue, one of the things you can see is the number of messages in the queue.

logging, and by using the User Event Signal Plot we can plot the watchdog margin over time. Adding a CPU Load Graph on the same timeline, we can see how task execution affects the watchdog margin, as shown below (left).

In the CPU Load Graph, we see that the ServerTask is executing a lot in the second half of the trace, and this seems to impact the watchdog margin. ServerTask (bright green) has higher priority than ControlTask (dark green), so when it is executing a lot in the end of the trace, we see that ControlTask is getting less CPU time. This is an intrinsic effect of Fixed Priority Scheduling, which is used by most RTOSes. Most likely, this could cause the full message queue, since ControlTask might not be able to read messages fast enough when the higher priority ServerTask is using most of the CPU time. This is an example of a Priority Inversion problem, as the SamplerTask is blocked by an unrelated task of lower priority. A solution could be to change the scheduling priorities, so that ControlTask gets higher priority than ServerTask. Let's try that and see how it would look.

Switching priorities solves problem

The right screenshot below shows the result of switching the task scheduling priorities between ServerTask and ControlTask. The system now shows a much more stable behavior. The CPU load of SamplerTask (red here) is quite steady around 20%, indicating a stable periodic behavior, and the watchdog margin is a perfect line, always at 10 ms. It does not expire anymore – problem solved!

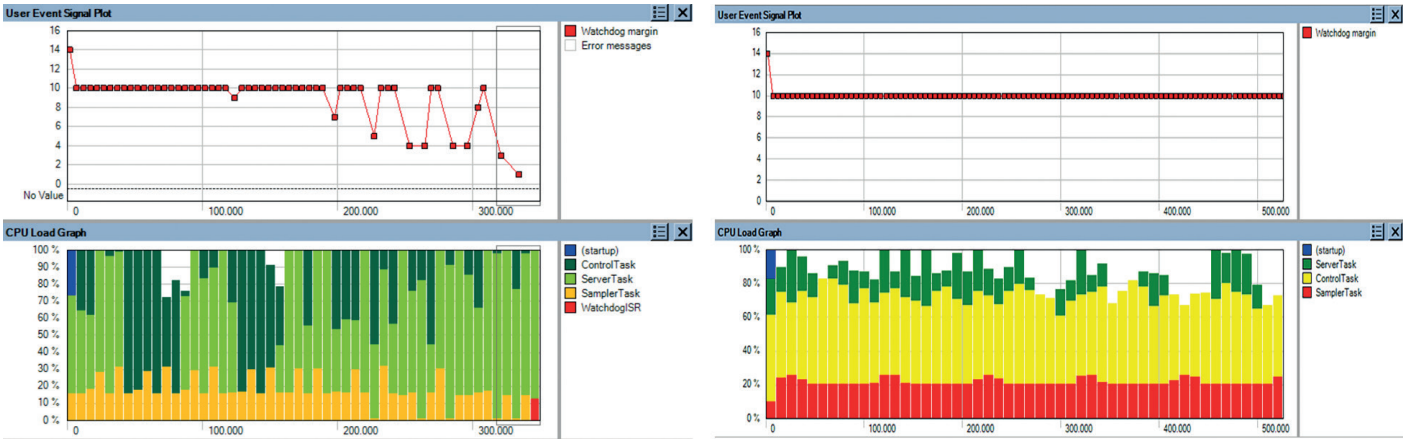
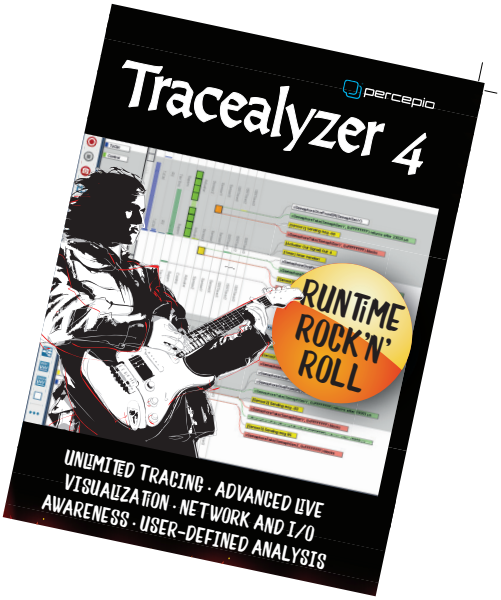


Figure 4. In the original system configuration (left) ControlTask could not empty the message queue fast enough, finally causing the watchdog timer to expire. After adjusting priorities everything works as intended (right).



Download Tracealyzer

https://percepio.com/download
Time-limited evaluation license included

Percepio AB, Västerås, Sweden
https://percepio.com

percepio®

Percepio March 2020