

Visualizing SAFERTOS® Applications

Enhancing Safety and Reliability with Tracealyzer

Issue 1.1 - 16, February 2026

Written in partnership with Percepio

Copyright date as document date.



CONTENTS

| | |
|--|----|
| Contents..... | 2 |
| List Of Figures..... | 3 |
| List of Notation..... | 3 |
| | |
| CHAPTER 1 Introduction..... | 4 |
| | |
| CHAPTER 2 RTOS Fundamentals..... | 5 |
| 2.1 Tasks, Priorities, and Analysis..... | 5 |
| 2.2 Priority Decides Scheduling..... | 5 |
| | |
| CHAPTER 3 Visualization with Tracealyzer..... | 6 |
| 3.1 Task Scheduling in Tracealyzer..... | 6 |
| 3.2 Actors, Instances and Fragments..... | 6 |
| 3.3 Revealing History..... | 7 |
| | |
| CHAPTER 4 Performance Analysis | 8 |
| 4.1 Response Time and Execution Time..... | 8 |
| | |
| CHAPTER 5 Advanced Tracealyzer Features..... | 9 |
| 5.1 State Machines and Intervals..... | 9 |
| 5.2 Overview With Communication Flow..... | 10 |
| | |
| CHAPTER 6 Practical Example - Watchdog Reset..... | 11 |
| 6.1 Watchdog Reset..... | 11 |
| 6.2 Why The Blocking?..... | 12 |
| 6.3 Switching Priorities Solved The Problem..... | 13 |
| | |
| CHAPTER 7 Conclusion..... | 14 |
| | |
| Contact Information..... | 15 |



List of Figures

| | |
|--|----|
| Figure 1 Main Trace View..... | 6 |
| Figure 2 Execution and Response Time..... | 7 |
| Figure 3 Runnables..... | 9 |
| Figure 4 Communication Flow..... | 10 |
| Figure 5 Watchdog Margin Varies..... | 11 |
| Figure 6 Watchdog Blocking Send..... | 12 |
| Figure 7 Watchdog Queue Full..... | 12 |
| Figure 8 Watchdog Margin vs. CPU Load..... | 13 |
| Figure 9 Watchdog Stable System..... | 13 |

List of Notation

| | |
|------|-----------------------------------|
| API | Application Programming Interface |
| CPU | Central Processing Unit |
| IFE | Finish Events |
| FPS | Fixed-priority scheduling |
| HMI | Human-Machine Interface |
| IFE | Instance Finish Events |
| I/O | Input/Output |
| MCU | Microcontroller Unit |
| RTOS | Real Time Operating System |
| UI | User Interface |



CHAPTER 1 Introduction

The use of a Real-Time Operating System (RTOS) has become increasingly common in embedded software development, offering multitasking, improved reliability, and maintainability in complex systems. While RTOS designs enable efficient handling of multiple concurrent tasks such as control, communication, and human-machine interfaces, they also introduce challenges in task scheduling, performance analysis, and debugging. This white paper explores how Percepio Tracealyzer provides developers with powerful visualization and analysis capabilities to address these challenges, helping to optimize system performance and reliability in RTOS-based designs.



CHAPTER 2 RTOS Fundamentals

2.1 Tasks, Priorities, and Analysis

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 and tasks, which execute seemingly in parallel and independent of each other. An RTOS provides multitasking, in a reliable and maintainable manner, which makes it easier to design applications with multiple concurrent functions such as control, communication and Human-Machine Interface (HMI).

The overhead of an RTOS is negligible on modern 32- or 64-bit processors and is often more than compensated for by more efficient designs enabled by multitasking.

2.2 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 RTOS use fixed-priority scheduling (FPS), where the developers assign each task a static priority level to indicate their relative urgency. The RTOS scheduler then chooses the task with the highest priority among the tasks that are ready to execute. This is a 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 that consume a lot of processor time may prevent lower priority tasks from executing.

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

Percepio offers the RTOS-aware tool Tracealyzer, with over 30 interactive views, to help developers visualize, comprehend, and analyze how the system behaves during runtime. Tracealyzer provides a sophisticated visualization that makes it easier to make sense of what is happening. 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.

Instances are quite central in Tracealyzer since they are used both in the trace visualization and for providing timing statistics. 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). Instances can be further divided into fragments, which is what happens when execution is preempted by a higher priority task or an interrupt handler. Clicking on a fragment in the main trace view selects the entire instance, as shown in Figure 1.

You might wonder how the stream of task scheduling events is 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. Sometimes Tracealyzer's heuristics might not do the right thing for you, however, and then you have the option to let your code generate explicit Instance Finish Events (IFE) at suitable points during execution. This way you can control how to group events into instances, and thereby also control the interpretation and display of timing statistics.

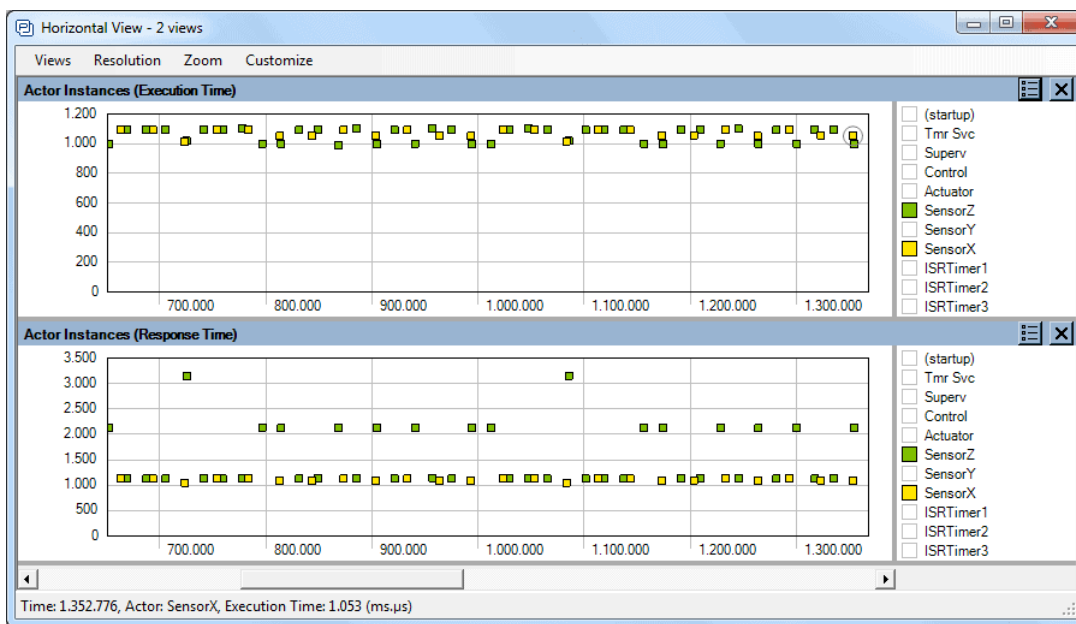


Figure 2 Execution and Response Time

3.3 Revealing History

Clicking on a semaphore, queue or mutex event in the main trace view opens the Kernel Object History view for the selected object, as illustrated above, showing a separate timeline with all operations on and states of this specific object. You can double-click on an event in this view to locate it in the main trace view.

For queue objects, you also see the number of messages in the buffer at any point, and you can track messages from send to receive or vice versa. For mutex objects you see the name of the task that currently owns the mutex.

Tracealyzer also offers a separate Event Log view that shows all events in the trace and allows you to filter based on object, event type, and many other attributes.



CHAPTER 4 Performance Analysis

4.1 Response Time and Execution Time

Performance analysis is an important aspect of embedded software development. There are several well-known ways to do this, but again RTOS-based systems present some unique problems that you should be aware of. Let's look at response time, the time from when a task is activated until it is completed, as an example.

There are many ways to measure response time, e.g., by toggling an I/O pin and measuring with a logic analyzer, or by adding code to log the number of clock cycles from start to finish. Either way, you will be able to see how much time it takes for this particular task to perform one cycle – but you may not be able to figure out why the value varies. Did that loop take twice as much time as the other due to something in your code? Because of a full buffer that needed to be flushed? Or was it because another task preempted the measured task? Optimizing your code before you know the answer to that question may turn out to be a waste of time.

Another important performance measurement is execution time, the actual processor time used by a particular piece of code over the measured interval. You might measure this using a solution that samples the program counter and provides a high-level overview of where processor time is spent. Several common IDEs support this, and most ARM-based MCUs even provide hardware support for it. What you get is, however, an average measurement which can be wildly inaccurate for less frequent functions or tasks. Moreover, you will not catch sporadic cases of unusually long executions that might cause problems such as timeouts.

Using Tracealyzer, you can look at the runtime world from many different perspectives, including plots of task execution and response times like in Figure 2. In the graphs, we can see that execution times are steady for both tasks but sometimes the response time of “SensorZ” is much higher. Clicking one of the outlying data points opens the corresponding interval in the main trace view where you should be able to see the cause, or at least find some clues to follow. All views in Tracealyzer are interconnected in similar ways.

CHAPTER 5 Advanced Tracealyzer Features

5.1 State Machines and Intervals

Tracealyzer allows you to define custom data sets, called Intervals and State Machines, based on events in the trace. This allows you to analyze application-specific properties, i.e., the things that really matter for your system.

An Interval is used to measure or highlight the time between two specific events. By adding an Interval definition to a trace, you create a data set containing all intervals (pairs of events) matching the definition. For example, if a trace contains the events "On" and "Off", you can define an interval starting at "On" and ending at "Off" to see all "On" -> "Off" sequences, independent of other events in between.

A State Machine is used to visualize any kind of state information logged in the trace. When defining a state machine, you specify events corresponding to state changes, which creates a group of interval data sets, one for each observed state, in the trace.

Intervals and states can be displayed in several Tracealyzer views. You can also generate statistics reports for them, much like you would do for actors.

Historically, intervals and state machines had to be defined in the application UI, or in external configuration files, but the latest version of Tracealyzer allows you to define them directly in application code using the TraceRecorder API. This makes custom tracing more efficient and easier to use. Views like state graphs and interval plots are available directly when such traces are loaded in Tracealyzer.

Recent versions of Tracealyzer also support Runnables, a common concept in automotive software. A runnable is a software component that runs within an actor, typically implemented as a single C function. In Tracealyzer, runnables are implemented as a special case of intervals (Figure 3) where the interval set, and channels are bound to an actor. And like intervals, runnables for a particular trace can be defined either in XML configuration files or in application code.

Runnables allows for detailed profiling of any C/C++ code where timing and performance are important and can be useful in any kind of embedded software. Having said that, the runnables support in Tracealyzer has been developed with the automotive use case in mind and has the following limitations:

- Each runnable must be bound to one specific actor.
- Runnables cannot be nested. An actor can only have one runnable active. This means that runnables can only be used for top level activities within an actor, not for complete nested function tracing.

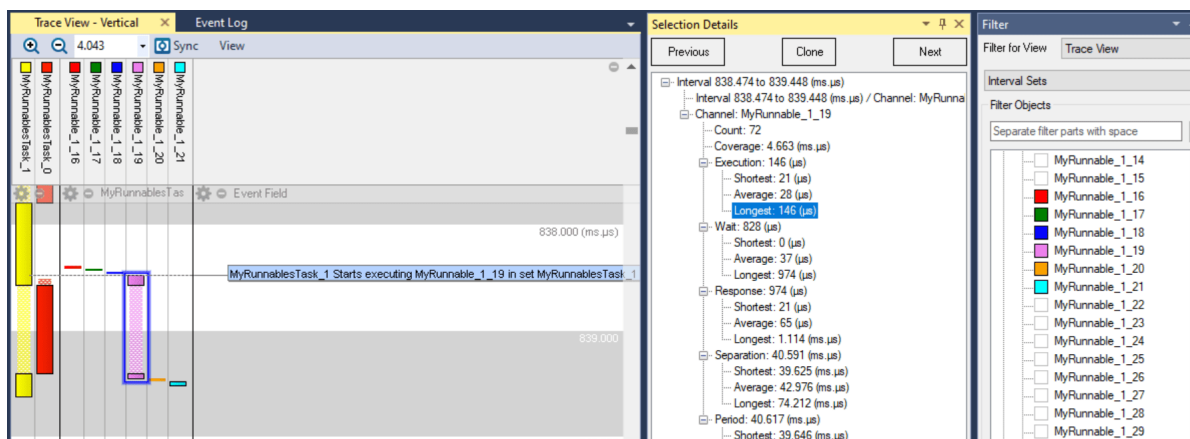


Figure 3 Runnables

5.2 Overview With Communication Flow

The main trace view with all its details remains the flagship of Tracealyzer but it's worth mentioning that Tracealyzer also offers a helicopter-like view of an RTOS application that can be very handy for spotting a certain class of bugs. It is called the Communication Flow diagram and shows the message flow between core RTOS objects—mutexes, semaphores, queues et cetera—in your application. See Figure 4.

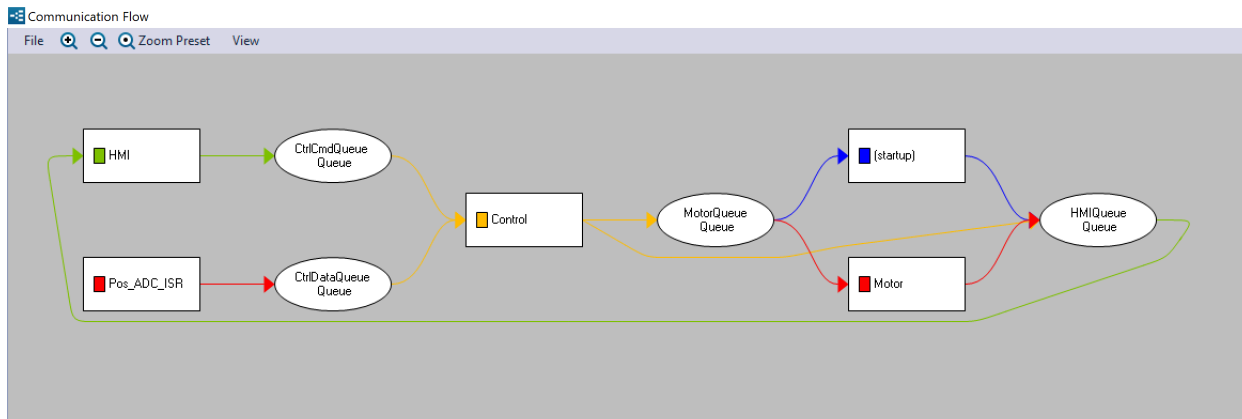


Figure 4 Communication Flow



CHAPTER 6 Practical Example – Watchdog Reset

6.1 Watchdog Reset

Finally, let's review an example of how Tracealyzer has been of use to one of our customers.

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 the watchdog timer had expired. This timer was supposed to be reset in a high priority task that executed periodically.

The ability to insert custom User Events comes in handy in this case. They are similar to a classic printf() call, and events here have been added when the watchdog timer was reset and when it expired. User events also support data arguments, so each event includes the timer value (just before resetting it) to see the watchdog "margin", i.e., remaining time. The result can be seen in Figure 5, in the yellow text labels.

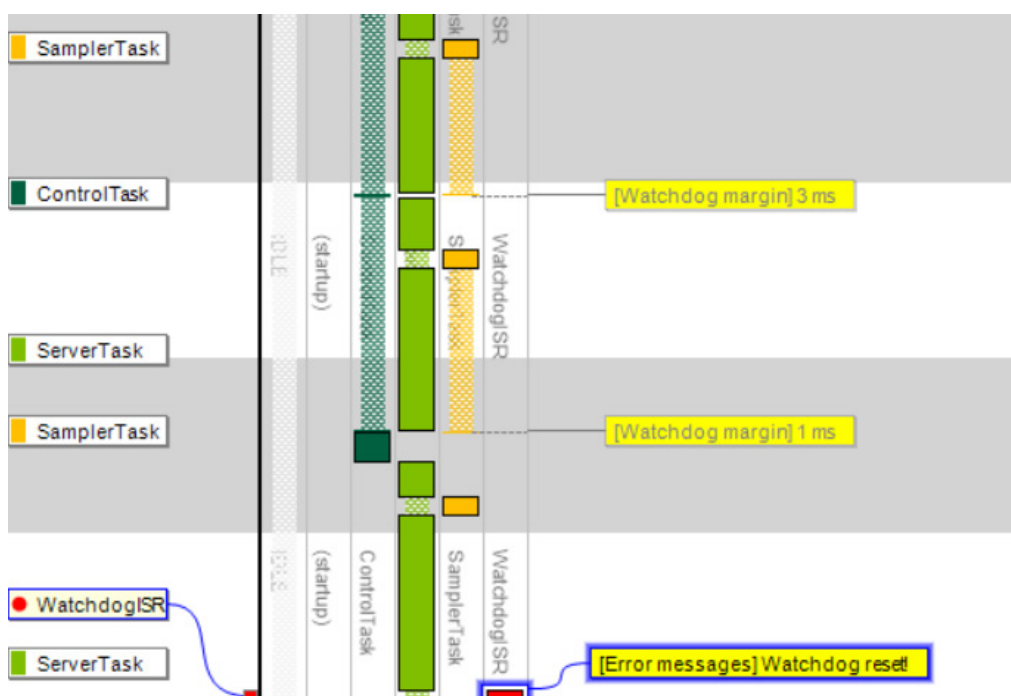


Figure 5 Watchdog Margin Varies

We can see that the SamplerTask is running, but it fails to clear the watchdog timer in the last execution of the task, which causes the system to reset ("Watchdog reset!") after a while. So why didn't SamplerTask reset the watchdog timer?

Let's look at Kernel Service calls (Figure 6) to see what the sampling task was doing. The last event of SamplerTask is a call to xQueueSend, a kernel function that adds a message to a queue. Note that the label is red, meaning that the xQueueSend call blocked, which caused a context switch to ServerTask before the watchdog timer could be reset.

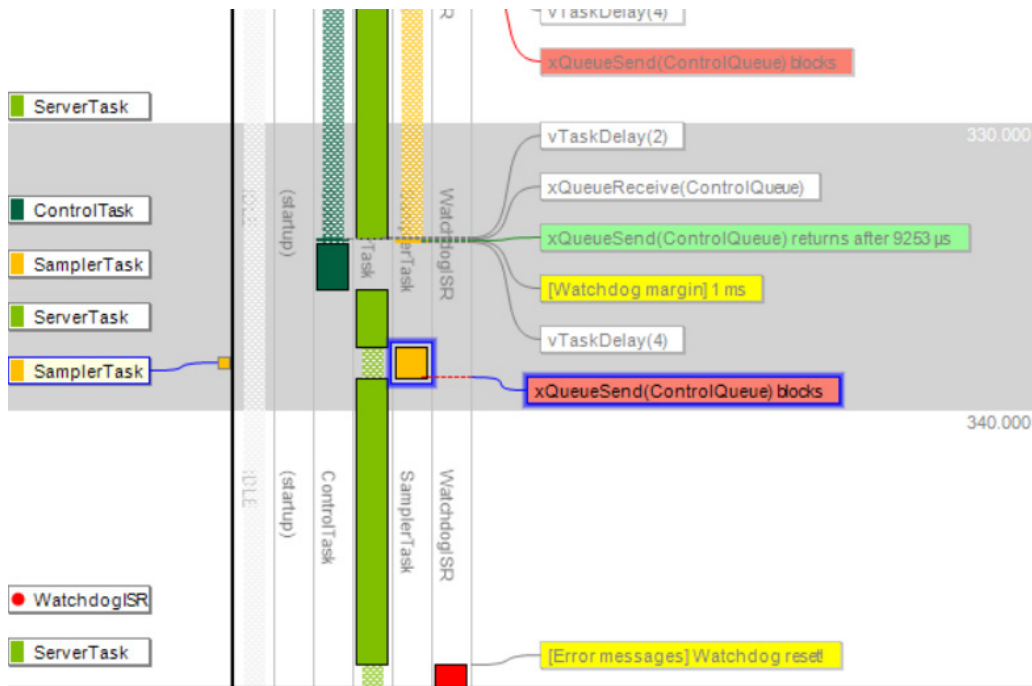


Figure 6 Watchdog Blocking Send

6.2 Why The Blocking?

So why did xQueueSend block? By double-clicking on this event label, we open the Object History View, showing all operations on this queue, ControlQueue, as illustrated in Figure 7. The rightmost column shows the buffered messages. We can see that ControlQueue already contains five messages and is probably full, hence the blocking. But the ControlTask is supposed to read the queue and make room, why hasn't this worked as expected?

| 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 7 Watchdog Queue Full

To investigate this, we need to look at how the watchdog margin varies over time. We have this information in the user event logging, and we can open a User Event Signal Plot to plot it over time. Adding a CPU Load Graph on the same timeline, we can see how task execution affects the watchdog margin, as shown below (Figure 8).

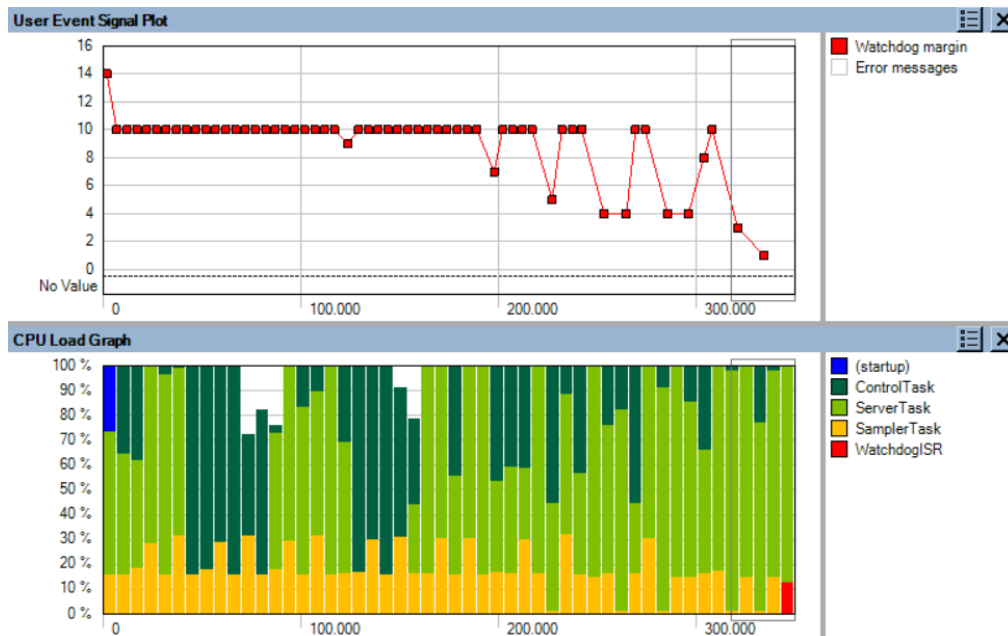


Figure 8 Watchdog Margin vs. CPU Load

It is obvious from the CPU Load Graph that 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, ControlTask gets less and less CPU time. This is an intrinsic effect of Fixed Priority Scheduling, which is used in most RTOSes. Most likely, this could cause the full message queue, as ControlTask might receive too little processor time to be able to read messages fast enough. A solution could be to change the scheduling priorities, so that ControlTask runs at a higher priority than ServerTask. Let's try that and see what happens.

6.3 Switching Priorities Solved The Problem

The next screenshot (Figure 9) shows the result. After switching task scheduling priorities, the system shows a much more stable behavior. The CPU load of SamplerTask (red here) is quite steady around 20%, indicating a stable periodic behaviour, and the watchdog margin is a perfect line, always at 10 ms. It does not expire anymore – problem solved!

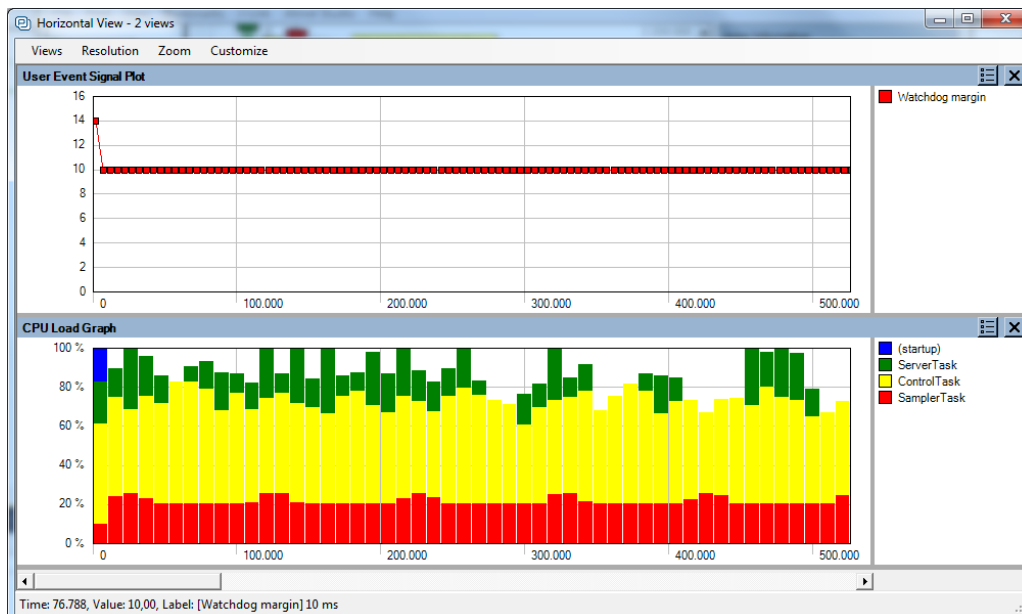


Figure 9 Watchdog Stable System



CHAPTER 12 Conclusion

Effective use of an RTOS requires not only careful design of task priorities and scheduling but also robust tools to analyze and validate runtime behavior. Percepio Tracealyzer enables engineers to gain deep insight into task execution, performance bottlenecks, and system interactions, making it easier to detect and resolve complex issues. By improving visibility into runtime behaviour, Tracealyzer helps developers of safety-critical systems, such as those built on SAFERTOS®, to ensure reliable, predictable, and high-performing embedded applications.



WITTENSTEIN

CONTACT INFORMATION

User feedback is essential to the continued maintenance and development of SAFERTOS®. Please provide all software and documentation comments and suggestions to the most convenient contact point listed below.

Contact WITTENSTEIN high integrity systems

Address: WITTENSTEIN high integrity systems
3 Rivergate, 1st Floor
Temple Quay
Bristol, BS1 6EW
England

Phone: +44 (0)1275 395 600
Email: support@HighIntegritySystems.com

Website www.HighIntegritySystems.com

Contact Percepio

Address: Percepio AB
Varmvalsvägen 13
72130 Västerås
Sweden

Contact: <https://percepio.com/contact-us/>

Website <https://percepio.com/>

All Trademarks acknowledged.

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



WITTENSTEIN

email: sales@highintegritysystems.com
web: www.highintegritysystems.com