How to visualize response times in FreeRTOS

Efficient development of FreeRTOS-based firmware requires understanding of the timing and interactions between tasks, interrupts and the kernel.

Copyright ©2022 Percepio AB. All Rights Reserved.

How to visualize response times in FreeRTOS

Efficient development of FreeRTOS-based firmware requires understanding of the timing and interactions between tasks, interrupts and the kernel.

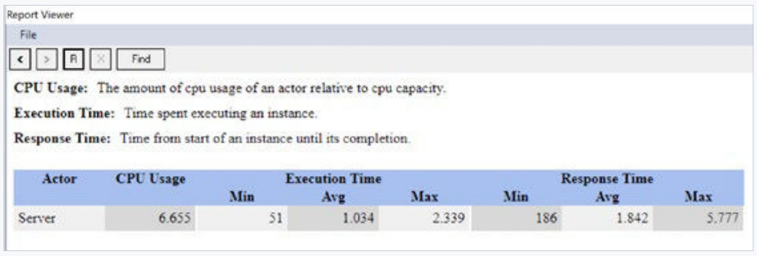
Percepio Tracealyzer is the premier solution for analysis and visualization of FreeRTOS-based embedded software. It provides more than 30 graphical interconnected views of different aspects of the software's real-time behavior.

We collect and reproduce examples of how customers have applied Tracealyzer to real-world issues. In this example, the customer developed a networked system running a TCP/IP stack, a flash file system, and an RTOS running on an ARM Cortex-M4 microcontroller. The system contained several RTOS tasks, including a server-style task that responds to network requests, and a log file spooler task. The response time on network requests had been an issue in the past and when testing the latest build, the situation had deteriorated. Now they really needed to figure this out!

They started by comparing the source code between the two versions, but they could not find any obvious cause for the longer response time. There were many small changes, seemingly due to refactoring, but no new functions were added. Therefore, they decided to use Tracealyzer to compare the runtime behaviors of the old and new versions.

Traces were recorded from both versions under similar conditions. They began the comparison in the Statistics Report (Figure 1A and Figure 1B), which includes high-level timing statistics such as CPU usage, number of executions, scheduling priorities and response times.

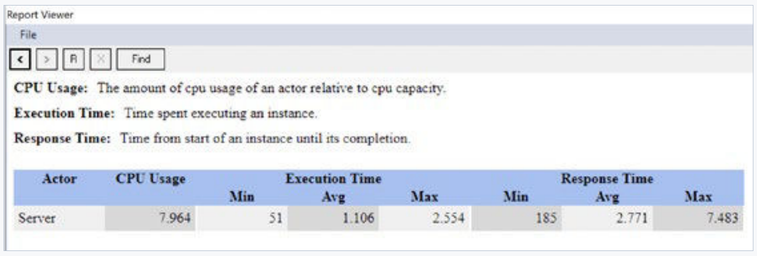
Percepio Tracealyzer is the premier solution for analysis and visualization of FreeRTOS-based embedded software.



CPU Usage: The amount of cpu usage of an actor relative to cpu capacity.
Execution Time: Time spent executing an instance.
Response Time: Time from start of an instance until its completion.

Actor	CPU Usage	Execution Time			Response Time		
		Min	Avg	Max	Min	Avg	Max
Server	6.655	51	1.034	2.339	186	1.842	5.777

*fig.*1A



CPU Usage: The amount of cpu usage of an actor relative to cpu capacity.
Execution Time: Time spent executing an instance.
Response Time: Time from start of an instance until its completion.

Actor	CPU Usage	Execution Time			Response Time		
		Min	Avg	Max	Min	Avg	Max
Server	7.964	51	1.106	2.554	185	2.771	7.483

*fig.*1B

As expected, the Statistics Report revealed that response times (total time from reception of a message until it is ready to receive the next message) for the Server task were about 50 percent higher in the new version. However, the execution times (i.e., the time spent executing) were similar: only about 7 percent longer in the new version. This led to the conclusion that the main reason for longer response time must be other tasks that interfered. But which tasks?

To determine which tasks interfered with the Server task, we clicked on the extreme values in the Statistics Report. This focused the main trace view on the corresponding locations so we could see the details, as illustrated below. And by opening parallel instances of Tracealyzer, one for each trace, we could easily compare them and spot the differences.

Since the Server task performed several services, two User Events were added (a User Event is basically a custom ***printf*** statement) to mark where the specific requests were received and answered, labeled ***ServerLog*** in Figure 2A and 2B. The zoom levels are identical, so we could clearly see the longer response time in the new version. We also saw that the Logger task preempted the Server task 11 times, compared to only 6 times in the earlier version.

Moreover, we see that the Logger task ran on a higher priority than the Server task, otherwise logging calls would not have preempted the Server task.

So there seemed to be new logging calls added in the new version, causing the Logger task to interfere more with the Server task. To see what was logged, we added another User Event in the Logger task to show all log messages in the trace view. Doing so informed us that other tasks besides Server generated log messages, for instance the

COMMUNICATION FLOW GRAPH

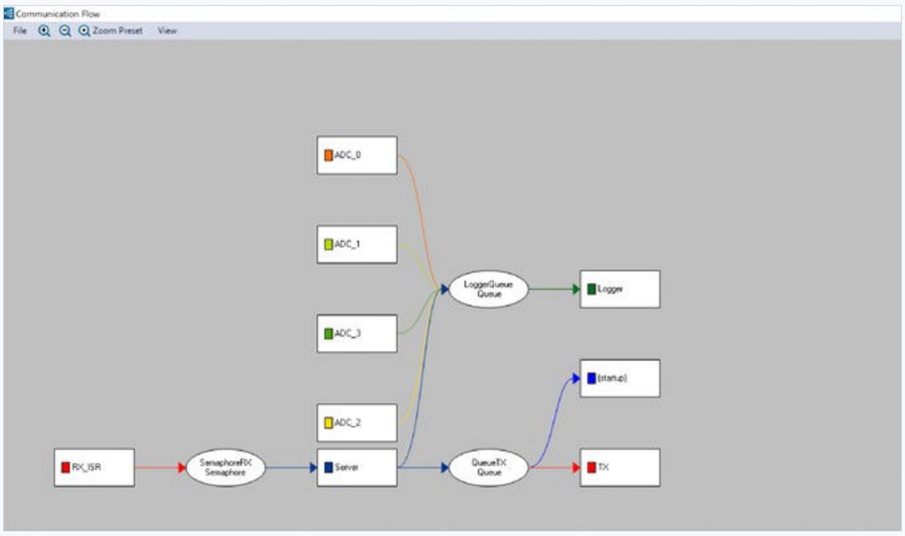


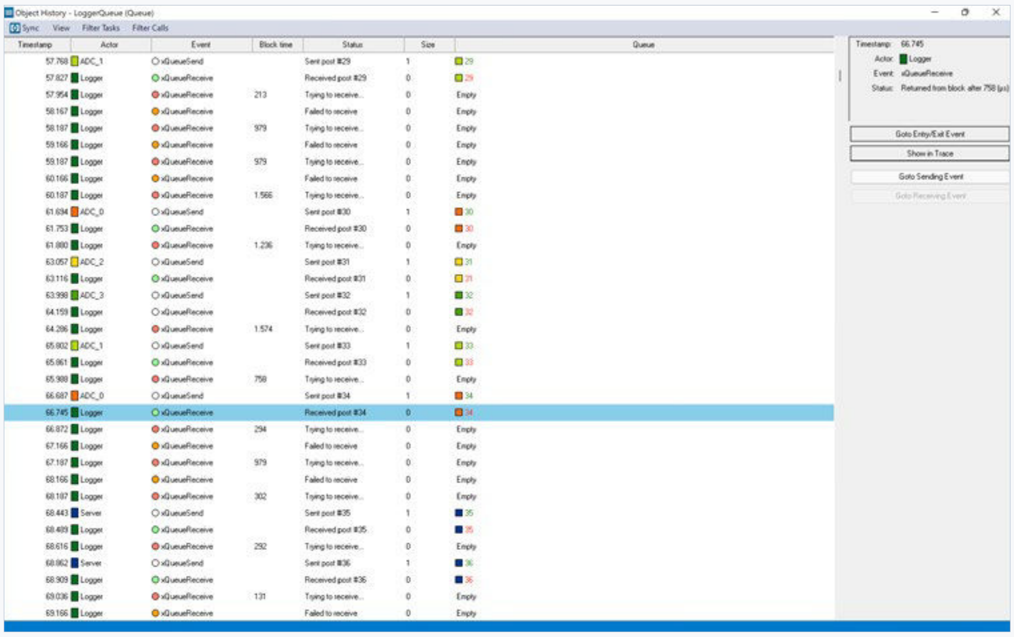
*fig.*4

ADC_0 task. To see all tasks sending messages to the Logger task, we used Tracealyzer's Communication Flow graph, illustrated in Figure 4. The Communication Flow graph shows a summary of all operations on message queues, semaphores and other kernel objects, performed by tasks and interrupts in the trace. This visualizes the high-level application design as well as runtime dependencies in the recorded situation.

In this case, the Communication Flow revealed that five tasks sent logging messages. By double-clicking the ***LoggerQueue*** node in the graph, we opened the Kernel Object History view to see all operations on this message queue (Figure 5). As expected, we saw that Logger task received messages frequently, one message at a time, and was blocked after each message, as indicated by the red light in the Event column.

But was this really a good design? It is probably not necessary to write log messages to file one-by-one. If we could increase the scheduling priority of the Server task above that of Logger, then Server would not be preempted as frequently and would thereby be able to respond

OBJECT HISTORY VIEW, SHOWING OPERATIONS ON A PARTICULAR MESSAGE QUEUE.



Timestamp	Actor	Event	Block time	Status	Size	Queue
57.768	🟩 ADC_1	○ xQueueSend		Sent post #29	1	🟩 29
57.827	🟩 Logger	● xQueueReceive		Received post #29	0	🟩 29
57.954	🟩 Logger	● xQueueReceive	213	Trying to receive...	0	Empty
58.167	🟩 Logger	● xQueueReceive		Failed to receive	0	Empty
58.187	🟩 Logger	● xQueueReceive	979	Trying to receive...	0	Empty
59.166	🟩 Logger	● xQueueReceive		Failed to receive	0	Empty
59.187	🟩 Logger	● xQueueReceive	979	Trying to receive...	0	Empty
60.166	🟩 Logger	● xQueueReceive		Failed to receive	0	Empty
60.187	🟩 Logger	● xQueueReceive	1.566	Trying to receive...	0	Empty
61.694	🟧 ADC_0	○ xQueueSend		Sent post #30	1	🟧 30
61.753	🟩 Logger	● xQueueReceive		Received post #30	0	🟧 30
61.880	🟩 Logger	● xQueueReceive	1.236	Trying to receive...	0	Empty
63.057	🟨 ADC_2	○ xQueueSend		Sent post #31	1	🟨 31
63.116	🟩 Logger	● xQueueReceive		Received post #31	0	🟨 31
63.998	🟩 ADC_3	○ xQueueSend		Sent post #32	1	🟩 32
64.159	🟩 Logger	○ xQueueReceive		Received post #32	0	🟩 32
64.286	🟩 Logger	● xQueueReceive	1.574	Trying to receive...	0	Empty
65.802	🟩 ADC_1	○ xQueueSend		Sent post #33	1	🟩 33
65.861	🟩 Logger	● xQueueReceive		Received post #33	0	🟩 33
65.988	🟩 Logger	● xQueueReceive	758	Trying to receive...	0	Empty
66.687	🟧 ADC_0	○ xQueueSend		Sent post #34	1	🟧 34
66.745	🟩 Logger	● xQueueReceive		Received post #34	0	🟧 34
66.872	🟩 Logger	● xQueueReceive	294	Trying to receive...	0	Empty
67.166	🟩 Logger	● xQueueReceive		Failed to receive	0	Empty
67.187	🟩 Logger	● xQueueReceive	979	Trying to receive...	0	Empty
68.166	🟩 Logger	● xQueueReceive		Failed to receive	0	Empty
68.187	🟩 Logger	● xQueueReceive	302	Trying to receive...	0	Empty
68.443	🟦 Server	○ xQueueSend		Sent post #35	1	🟦 35
68.489	🟩 Logger	● xQueueReceive		Received post #35	0	🟦 35
68.616	🟩 Logger	● xQueueReceive	292	Trying to receive...	0	Empty
68.862	🟦 Server	○ xQueueSend		Sent post #36	1	🟦 36
68.909	🟩 Logger	● xQueueReceive		Received post #36	0	🟦 36
69.036	🟩 Logger	● xQueueReceive	131	Trying to receive...	0	Empty
69.166	🟩 Logger	● xQueueReceive		Failed to receive	0	Empty

Timestamp:	66.745
Actor:	🟩 Logger
Event:	xQueueReceive
Status:	Returned from block after 758 (µs)

*fig.*5

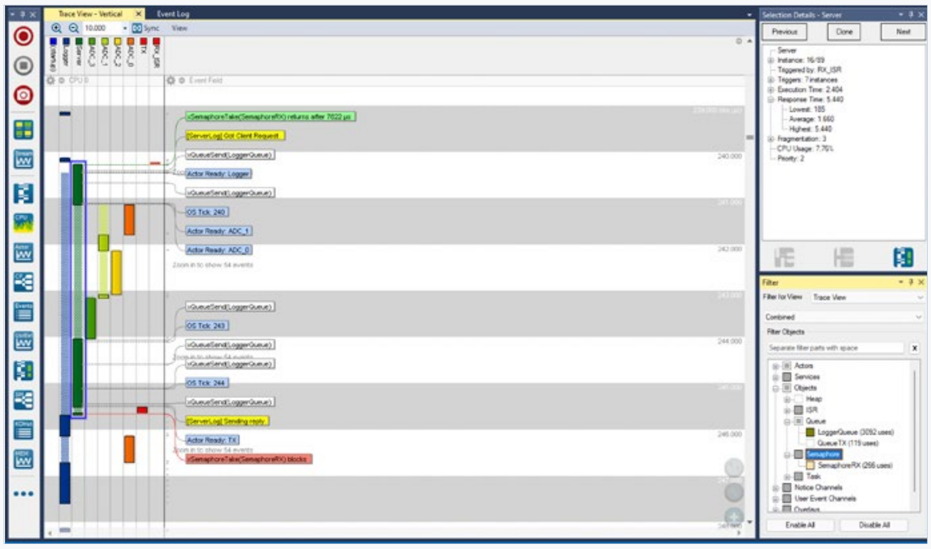
faster. The log messages could then be buffered in ***LoggerQueue*** until Server and other high priority tasks have completed. Only then would Logger resume and process all buffered messages in a batch.

We tried that. Figure 6 shows the result. The highest response time for the Server task was now just 5.4 ms, which was even faster than in the earlier version (5.7 ms) despite more logging. This was possible because the Logger task processed all pending messages in a batch after Server was finished, instead of preempting Server for each log message.

We could also see event labels for the message queue operations, and as expected there were several ***xQueueSend*** calls in sequence, without blocking or task preemptions. There were still some preemptions, caused by the A/D converter tasks, but this no longer caused extra activations of the Logger task.

Problem solved!

HIGHEST RESPONSE TIME OF SERVER TASK AFTER CHANGING THE PRIORITIES.



```
xSemaphoreTake(SemaphoreRX) returns after 7022 µs
[ServerLog] Got Client Request
xQueueSend(LoggerQueue)
Actor Ready: Logger
xQueueSend(LoggerQueue)
OS Tick: 240
Actor Ready: ADC_1
Actor Ready: ADC_0
Zoom in to show 54 events
xQueueSend(LoggerQueue)
OS Tick: 243
xQueueSend(LoggerQueue)
xQueueSend(LoggerQueue)
OS Tick: 244
xQueueSend(LoggerQueue)
[ServerLog] Sending reply
Actor Ready: TX
xSemaphoreTake(SemaphoreRX) blocks
```

```
Server
Instance: 16/89
Triggered by: RX_ISR
Triggers: 7 instances
Execution Time: 2.404
Response Time: 5.440
    Lowest: 185
    Average: 1.660
    Highest: 5.440
Fragmentation: 3
CPU Usage: 7.76%
Priority: 2
```

fig. 6

How does it work?

Tracealyzer uses flexible software-defined tracing and works on any processor. To record a trace, you only need to include Percepio's recorder library in your build, configure it and start the tracing.

The performance overhead is only a few microseconds per event, and you can stream the trace continuously to the host computer via a debug probe, TCP/IP, ITM or other channels. The trace can also be kept in a target-side RAM buffer and uploaded on demand.

**To learn more and get started,
please refer to the following on-line resources:**

PRODUCT PAGE | USER GUIDE | GETTING STARTED

If you have any questions, please contact support@percepio.com or your local distributor.

ABOUT THE AUTHOR//

Dr. Johan Kraft is CEO and founder of Percepio AB. Dr. Kraft is the original developer of Percepio Tracealyzer, a tool for visual trace diagnostics that provides insight into runtime systems to accelerate embedded software development. His applied academic research, in collaboration with industry, focused on embedded software timing analysis. Prior to founding Percepio in 2009, he worked in embedded software development at ABB Robotics. Dr. Kraft holds a PhD in computer science.

Learn more about why Tracealyzer is the premier solution for analysis and visualization of FreeRTOS-based embedded software.

Visit percepio.com/tracealyzer.

About Percepio

Percepio is the leading provider of visual trace diagnostics for embedded and IoT software systems in development and in the field.

Percepio Tracealyzer combines software tracing with powerful visualizations, allowing users to visually spot and analyze issues in software recordings during development and testing.

Percepio DevAlert is a cloud service for monitoring deployed IoT devices, combining automatic, real-time error reporting with visual trace diagnostics powered by Tracealyzer. Complimentary evaluation licenses are available for both products.

For more information, visit Percepio.com.

percepio

Copyright 2022 Percepio AB. All rights reserved. | info@percepio.com