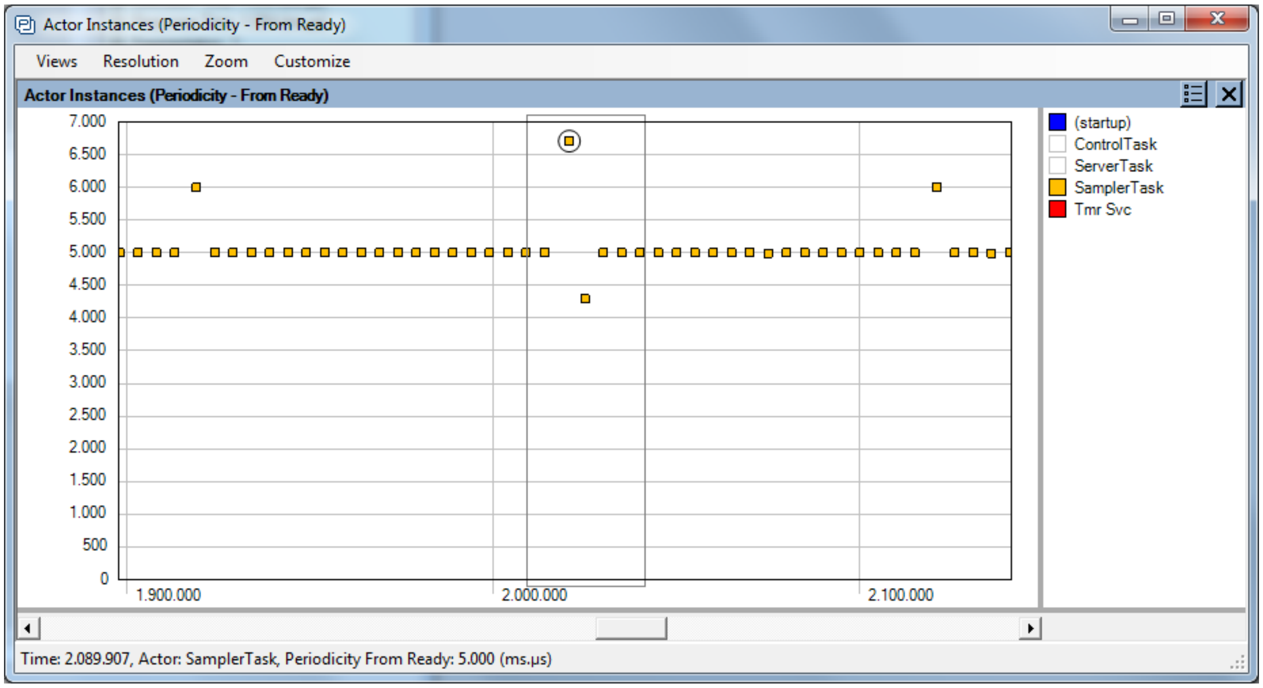
Customer Case: Unexpected Timing

We collect examples of how Tracealyzer has been of 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 periodic task not running as expected, here called SamplerTask. This was the highest priority task in the system and should execute every 5 ms to read a sensor. But they found significant timing variations in the data sampling. Why?

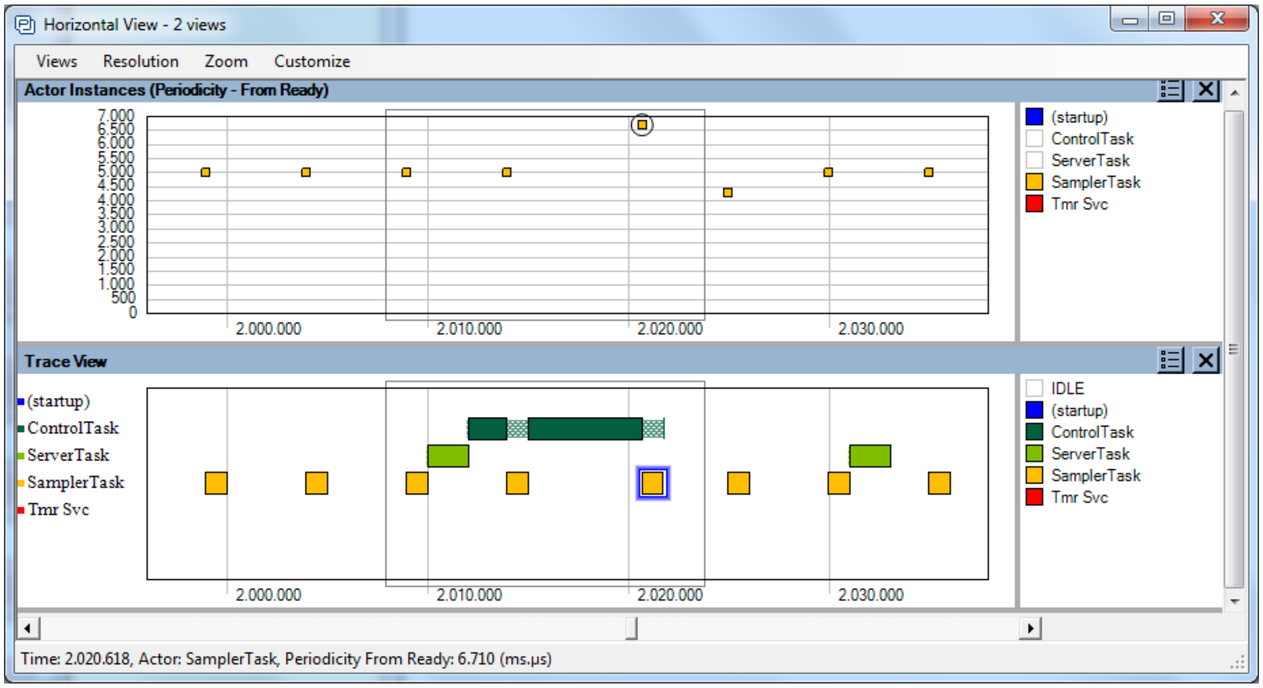
With Tracealyzer, they soon found the Actor Instance Graph, showing runtime metrics of individual task executions, as shown below. There are several such graphs, one for each supported metric, where each data point represents one execution of the task. The X-axis shows its start time of the instance and the Y-axis shows the specific metric, in this case "Periodicity – from Ready", i.e., the time in between task activations.



Time: 2.089.907, Actor: SamplerTask, Periodicity From Ready: 5.000 (ms.µs)

As you can see in the above screenshot, the SamplerTask is not executing strictly every 5 ms as intended. Usually it does, but there are a cases with unusually long intervals in between activations, over 6.5 ms.

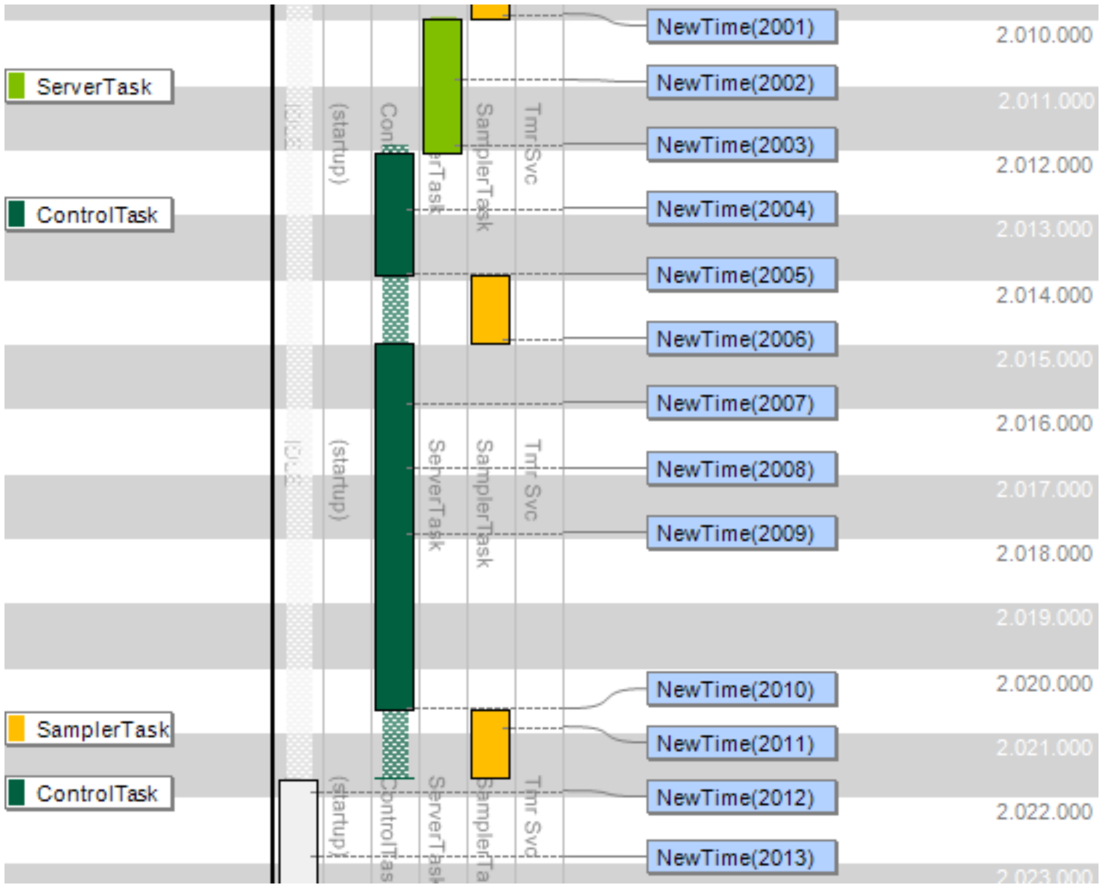
So what is going on at these instances? If we add a horizontal trace view, synchronized and with the other tasks enabled, we see that the ControlTask (dark green) is running in this interval. Is this causing the problem? But SamplerTask (orange) has higher priority, so its should preempt ControlTask. So what is going on here?



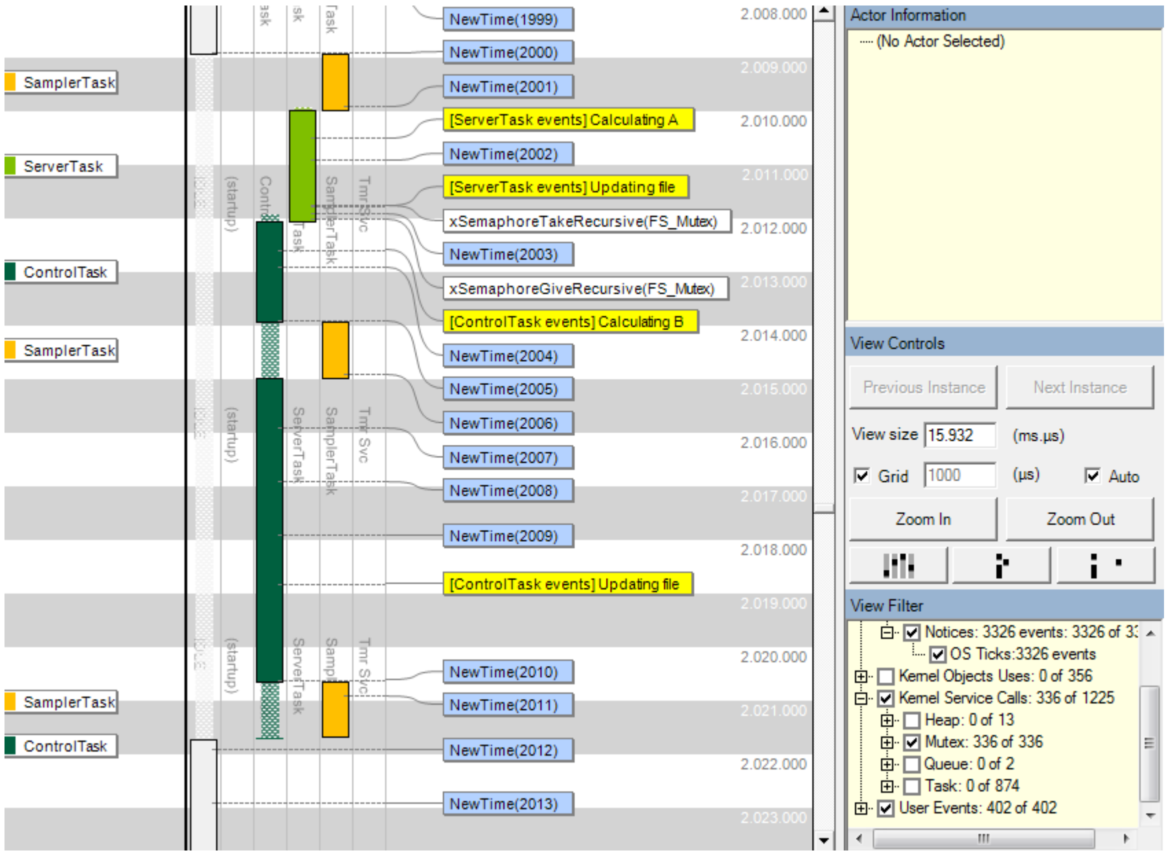
Time: 2.020.618, Actor: SamplerTask, Periodicity From Ready: 6.710 (ms.µs)

Perhaps the ControlTask is disabling interrupts for a few milliseconds and thereby also the kernel tick interrupt? We can find out in the main trace view, which allows us to inspect the tick events. Let’s have a look…

By double-clicking on the data point, the corresponding interval is shown in the main trace view. This view provides a vertical time-line, with events presented as horizontal text labels. The OS tick events should occur every 1 ms in this system, but there is a gap of about 2.5 ms, starting at 2.018.000.



So it seems plausible that ControlTask is disabling interrupts, probably to protect access to a shared resource, like a file system. But SamplerTask is not using that resource, so is it really necessary to block it by disabling interrupts? Let's check how the other task, ServerTask, is handling this.



```
NewTime(1999)
NewTime(2000)
NewTime(2001)
[ServerTask events] Calculating A
NewTime(2002)
[ServerTask events] Updating file
xSemaphoreTakeRecursive(FS_Mutex)
NewTime(2003)
xSemaphoreGiveRecursive(FS_Mutex)
[ControlTask events] Calculating B
NewTime(2004)
NewTime(2005)
NewTime(2006)
NewTime(2007)
NewTime(2008)
NewTime(2009)
[ControlTask events] Updating file
NewTime(2010)
NewTime(2011)
NewTime(2012)
NewTime(2013)
```

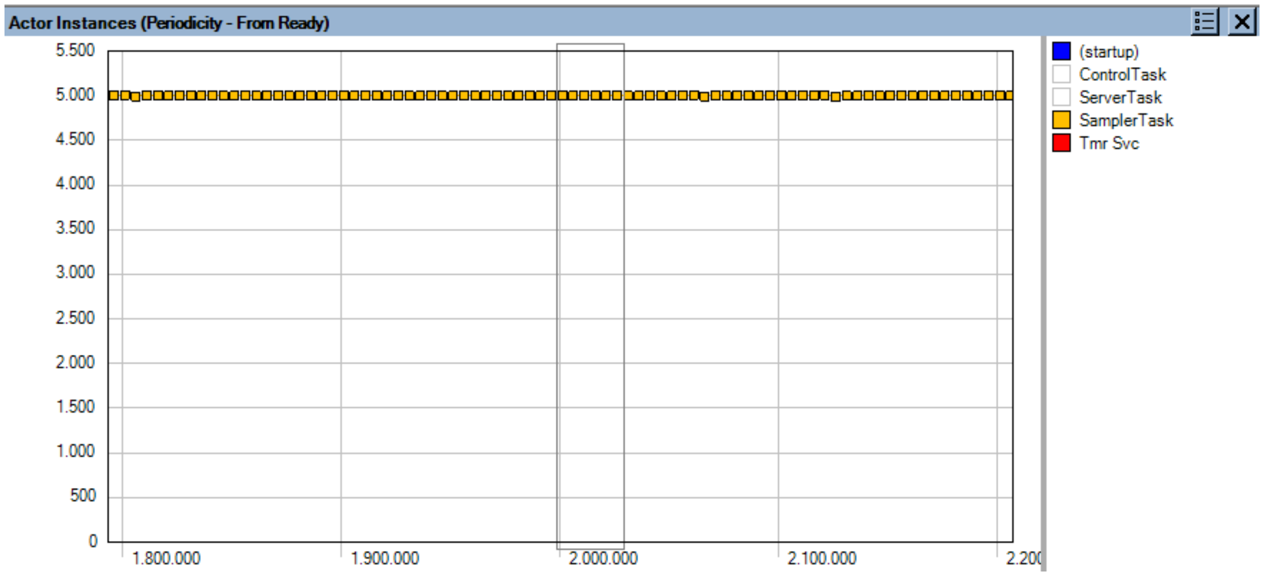
```
2.008.000
2.009.000
2.010.000
2.011.000
2.012.000
2.013.000
2.014.000
2.015.000
2.016.000
2.017.000
2.018.000
2.019.000
2.020.000
2.021.000
2.022.000
2.023.000
```

(No Actor Selected)

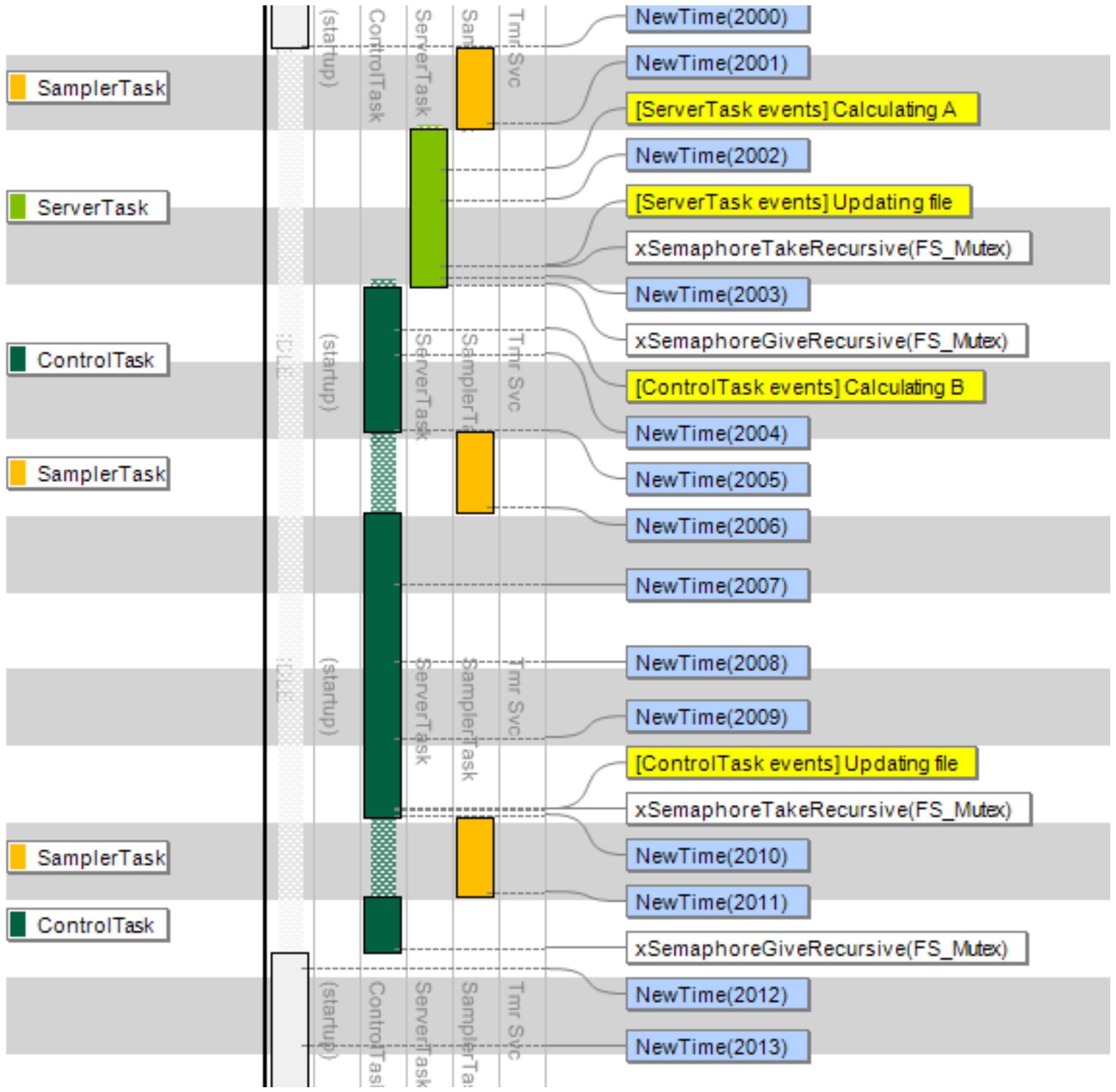
View size 15.932 (ms.µs)
☑ Grid 1000 (µs) ☑ Auto

```
☑ Notices: 3326 events: 3326 of 33
   ☑ OS Ticks:3326 events
☐ Kernel Objects Uses: 0 of 356
☑ Kernel Service Calls: 336 of 1225
   ☐ Heap: 0 of 13
   ☑ Mutex: 336 of 336
   ☐ Queue: 0 of 2
   ☐ Task: 0 of 874
☑ User Events: 402 of 402
```

In the lower right corner, you find the View Filter, where we have now enabled “Mutex” type kernel calls, and also User Events – generated by logging calls added to the application code. Here we can see that the ServerTask (bright green) is using a Mutex for protecting access to the file system, but ControlTask makes no such calls, even though the logging indicates a similar operation in both tasks (“Updating file”). So this can probably be solved by changing ControlTask to use the mutex (FS_Mutex) instead of disabling interrupts. So let’s see what happens if we change this. In the screenshot below you see that SamplerTask is now running periodically every 5 ms, as intended.



When looking at the detailed trace (below), we verify that ControlTask is now using FS_Mutex and the OS ticks are running periodically, even in the critical section, and lets SamplerTask preempt as intended.



```
NewTime(2000)
NewTime(2001)
[ServerTask events] Calculating A
NewTime(2002)
[ServerTask events] Updating file
xSemaphoreTakeRecursive(FS_Mutex)
NewTime(2003)
xSemaphoreGiveRecursive(FS_Mutex)
[ControlTask events] Calculating B
NewTime(2004)
NewTime(2005)
NewTime(2006)
NewTime(2007)
NewTime(2008)
NewTime(2009)
[ControlTask events] Updating file
xSemaphoreTakeRecursive(FS_Mutex)
NewTime(2010)
NewTime(2011)
xSemaphoreGiveRecursive(FS_Mutex)
NewTime(2012)
NewTime(2013)
```

Tracealyzer provides over 20 interactive views of the runtime world, connected in clever ways. It is available for several leading real-time operating systems as well as for Linux. Learn more at our website www.percepio.com.

Stay tuned for the next Customer Case – the mysterious random watchdog reset.

Precepio Tracealyzer available from Phaedrus Systems
www.phaedsys.com
info@phaedsys.com