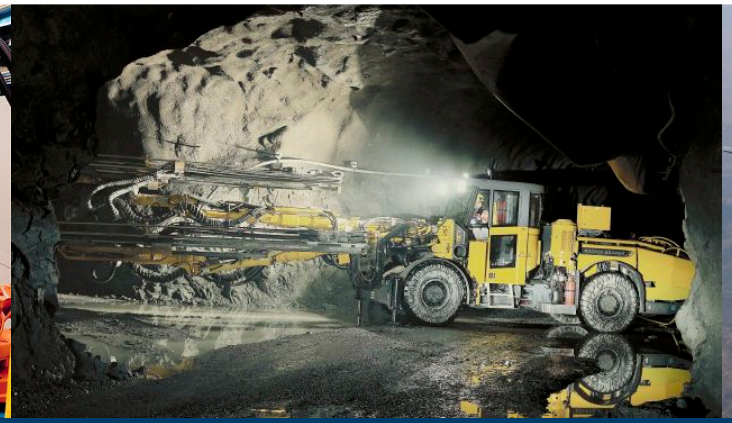


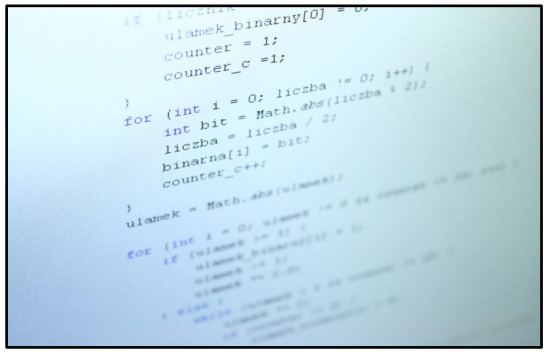
Trace Visualization for Efficient Debugging of Embedded Systems

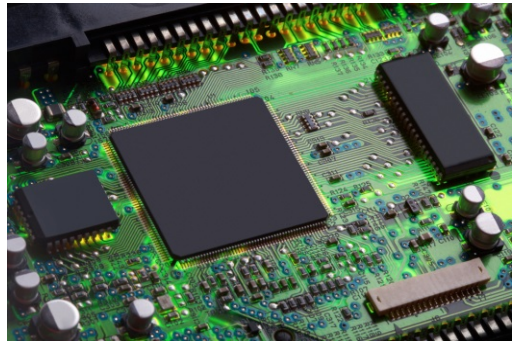
Percepio AB

percepio®

Source code is **not** the full picture





?

Emergent behaviors arise when modules are integrated into systems.

Runtime effects, not visible in the source code…

What is really going on in runtime?

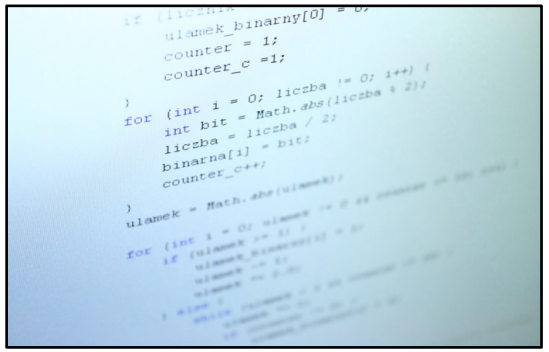
Correctness

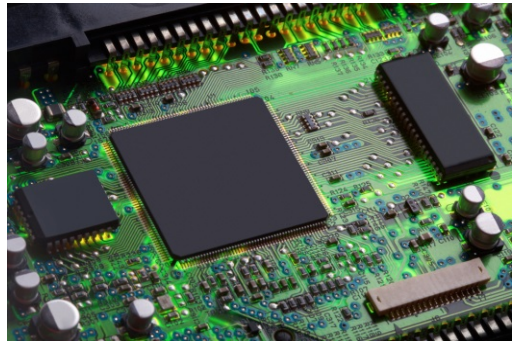
- Timeouts?
- Deadlocks?
- Enough hardware resources?
- …

Performance, e.g.,

- Responsiveness?
- Throughput?
- Efficient use of hardware resources?
- …

Source code is **not** the full picture





?

Emergent behaviors arise when modules are integrated into systems.

Runtime effects, not visible in the source code…

What is really going on in runtime?

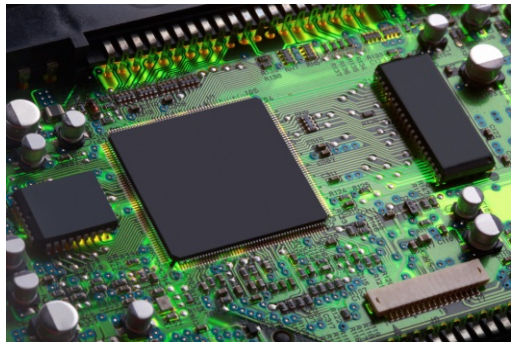
Correctness

- Timeouts?
- Deadlocks?
- Enough hardware resources?
- …

Performance, e.g.,

- Responsiveness?
- Throughput?
- Efficient use of hardware resources?
- …

We need to Monitor in Runtime



```
184140        000100fc -> write r31
184141  100a8   IncrementCounterByl  add          %r0,
184141        0000002e -> write r0
184142  100ac   IncrementCounterByl+0x04  extb
184142        0000002e -> write r0
184143  100b0   IncrementCounterByl+0x08  j_s
184144  100fc   main+0x30       stb          %r0,[%rl]
184144             2e -> write mem [0x011000]
184145  10100   main+0x34       ldb          %r0,[%r2]
184145             5a <- read  mem [0x011001] -> writ
184146        0000005a -> write r0
184147  10104   main+0x38       bl_s         IncrementC
184147        00010106 -> write r31
184148  100b4   IncrementCounterBy2  add          %r0,
184148        0000005c -> write r0
184149  100b8   IncrementCounterBy2+0x04  extb
184149        0000005c -> write r0
184150  100bc   IncrementCounterBy2+0x08  j_s
```

```
9.330]  xQueueReceive(CtrlDataQueue, 100) retu
0.253]  OS Ticks: 8109
1.253]  OS Ticks: 8110
1.270]  Context switch on CPU 0 to Pos_ADC_ISR
1.281]  xQueueSendFromISR(CtrlDataQueue)
1.290]  Context switch on CPU 0 to Control
1.868]  xQueueSend(MotorQueue)
1.878]  Actor Ready: Motor
1.889]  Context switch on CPU 0 to Motor
1.900]  xQueueReceive(MotorQueue, 10) returns
1.934]  xQueueReceive(MotorQueue, 10) blocks
1.954]  Context switch on CPU 0 to Control
1.965]  xQueueReceive(CtrlCmdQueue, 0) timeout
1.977]  xQueueReceive(CtrlDataQueue, 100)
1.990]  xQueueReceive(CtrlCmdQueue, 0) timeout
```

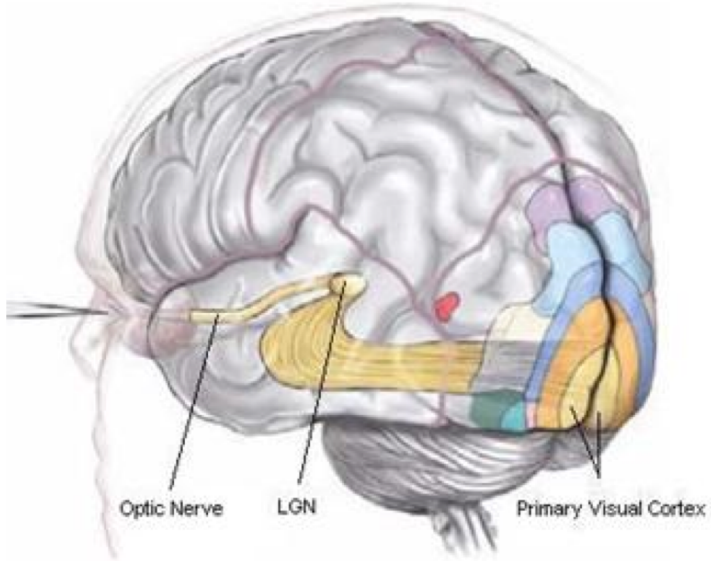
```
vc] Starting key provisioning...
vc] Write root certificate...
vc] Write device private key...
Svc] Write device certificate...
Svc] Key provisioning done...
Svc] Starting WiFi...
r Svc] WiFi module initialized.
S-MAIN] WiFi connected to AP AndroidAP.
WS-MAIN] Attempt to Get IP.
WS-MAIN] IP Address acquired 192.168.0.51
WS-LED] [Shadow 0] MQTT: Creation of dedicated MQ
WS-LED] Sending command to MQTT task.
QTT] Received message 10000 from queue.
QTT] Looked up a7sw0r7rvpirn.iot.us-east-1.amazona
MQTT] MQTT Connect was accepted. Connection estab
MQTT] Notifying task.
AWS-LED] Command sent to MQTT task passed.
```

	Instruction Trace	**Event Trace**	**Application Logging**
Producer	Processor core	Software (API or Kernel)	Software (application)
Abstraction Level	Low	Medium	High
Overhead	None	Some	More
System Requirements	High	Low	Low
Flexibility	Low	High	High

Making Traces Useful

- Raw traces are difficult to make sense of
  - Vast amounts of boring, repeating *patterns*
  - It's usually the *anomalies* that are interesting
- Could we use a powerful neural network to find them?
- Yes, we already have this - the Human Brain!
  - Extremely good at pattern recognition
  - But only for data in visual form…

```
[29.681.787]   free(0x0000C3C0) released 160 bytes
[29.682.107]   free(0x0000C370) released 80 bytes
[29.682.357]   free(0x0000C2C8) released 168 bytes
[29.685.357]   free(0x0000C258) released 112 bytes
[29.690.679]   malloc(112) returned 0x0000C258
[29.702.250]   malloc(216) returned 0x0000C2C8
[29.702.524]   malloc(80) returned 0x0000C3A0
[29.702.820]   malloc(160) returned 0x0000C3F0
[29.715.853]   free(0x0000C3F0) released 160 bytes
[29.716.172]   free(0x0000C3A0) released 80 bytes
[29.716.423]   free(0x0000C2C8) released 216 bytes
[29.718.456]   free(0x0000C258) released 112 bytes
[29.824.579]   malloc(112) returned 0x0000C258
[29.828.374]   free(0x0000C258) released 112 bytes
[29.830.290]   malloc(112) returned 0x0000C258
[29.841.798]   malloc(216) returned 0x0000C2C8
[29.842.155]   malloc(80) returned 0x0000C3A0
[29.842.437]   malloc(160) returned 0x0000C3F0
[29.855.445]   free(0x0000C3F0) released 160 bytes
[29.855.695]   free(0x0000C3A0) released 80 bytes
[29.856.002]   free(0x0000C2C8) released 216 bytes
[29.858.294]   free(0x0000C258) released 112 bytes
[29.964.516]   malloc(112) returned 0x0000C258
[29.968.319]   free(0x0000C258) released 112 bytes
[29.971.221]   malloc(112) returned 0x0000C258
[29.988.896]   malloc(328) returned 0x0000C2C8
```



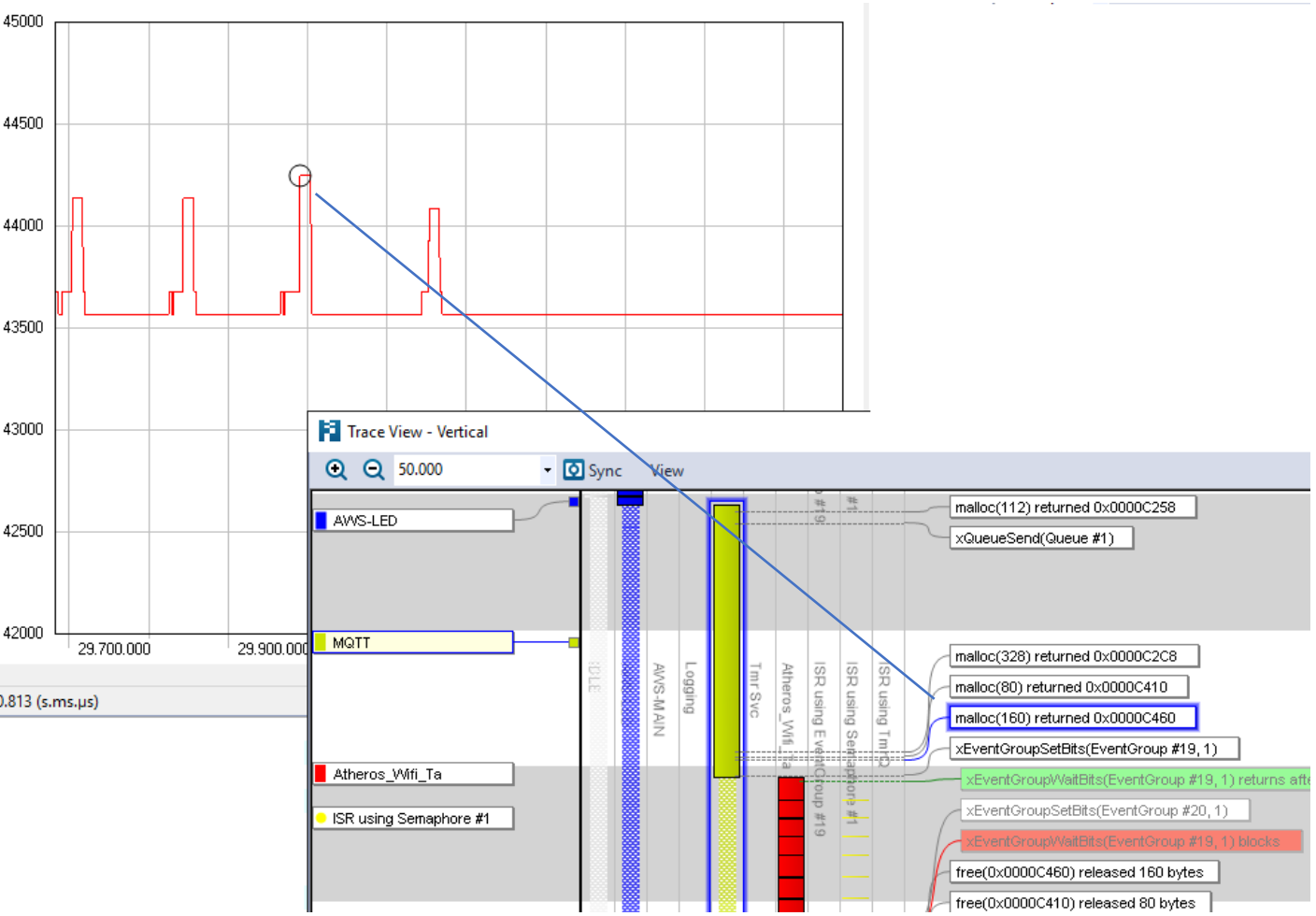
Trace Visualization

Trace Visualization

What is desired on high level?

- Abstract the data into meaningful overviews.

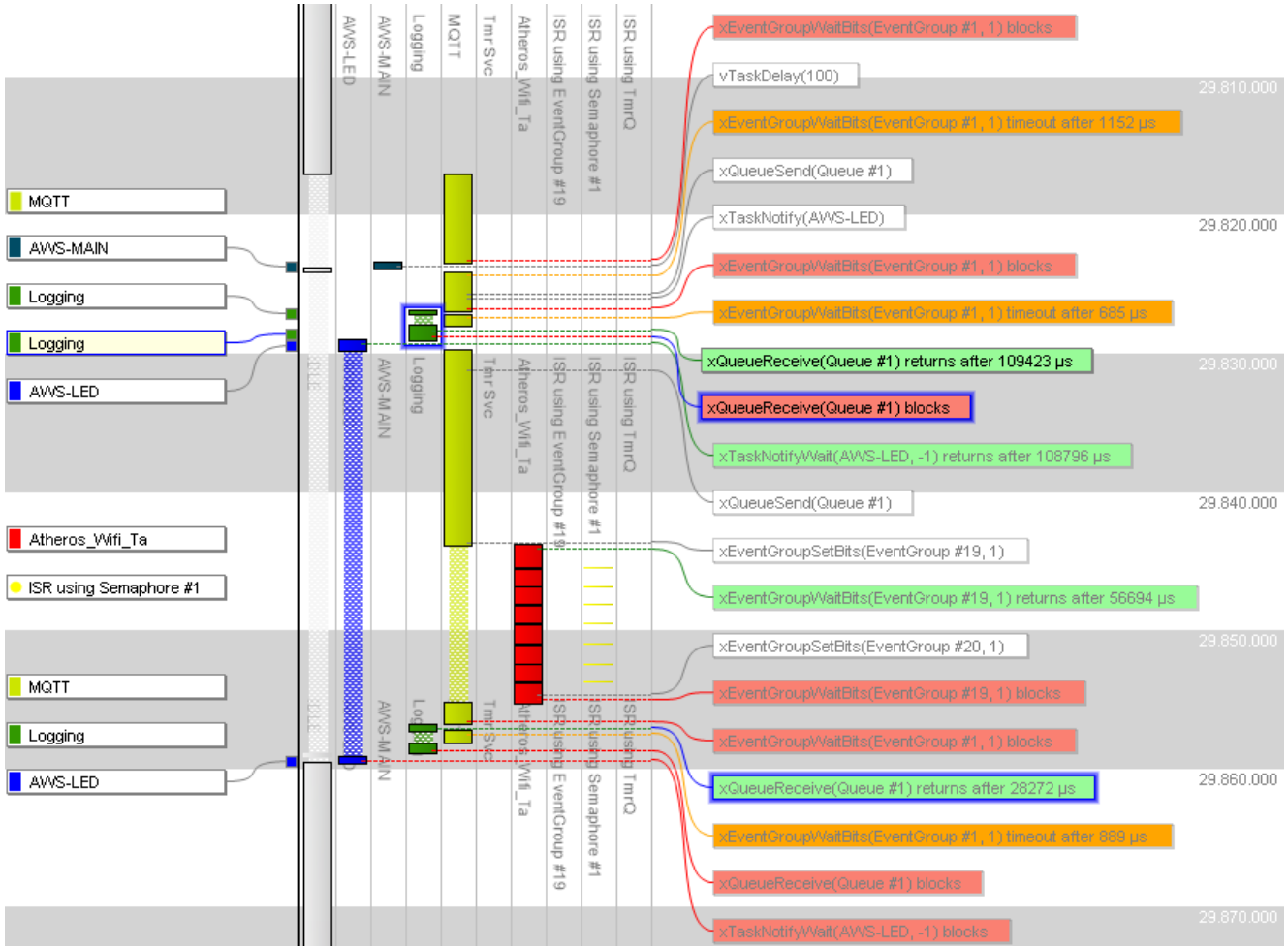
Trace Visualization



What is desired on high level?

- Abstract the data into meaningful overviews.
- Connect related views, allow drill-down.

Trace Visualization



What is desired on high level?

- Abstract the data into meaningful overviews.
- Connect related views, allow drill-down.
- Reveal dependencies between events.

Trace Visualization

What is desired on high level?

- Abstract the data into meaningful overviews.
- Connect related views, allow drill-down.
- Reveal dependencies between events.

Requires specialized visualization that actually *understands* the meaning of the trace data.

This way, far better understanding is possible.

Trace Visualization

What can be studied? Some examples:

- Multi-threading and timing
  - Context switches, internal kernel events
  - Execution time, response time, periodicity…
- API calls (OS, Middleware stacks, Drivers…)
  - Call sequences and timing
  - Parameters and return values
  - Blocking and timeouts
  - Object dependencies
- Application logging
  - Debug messages, variable values…
- Time between important events
- State changes over time

Benefits of Trace Visualization

Debugging
Find the cause of issues

Validation
System working as intended?

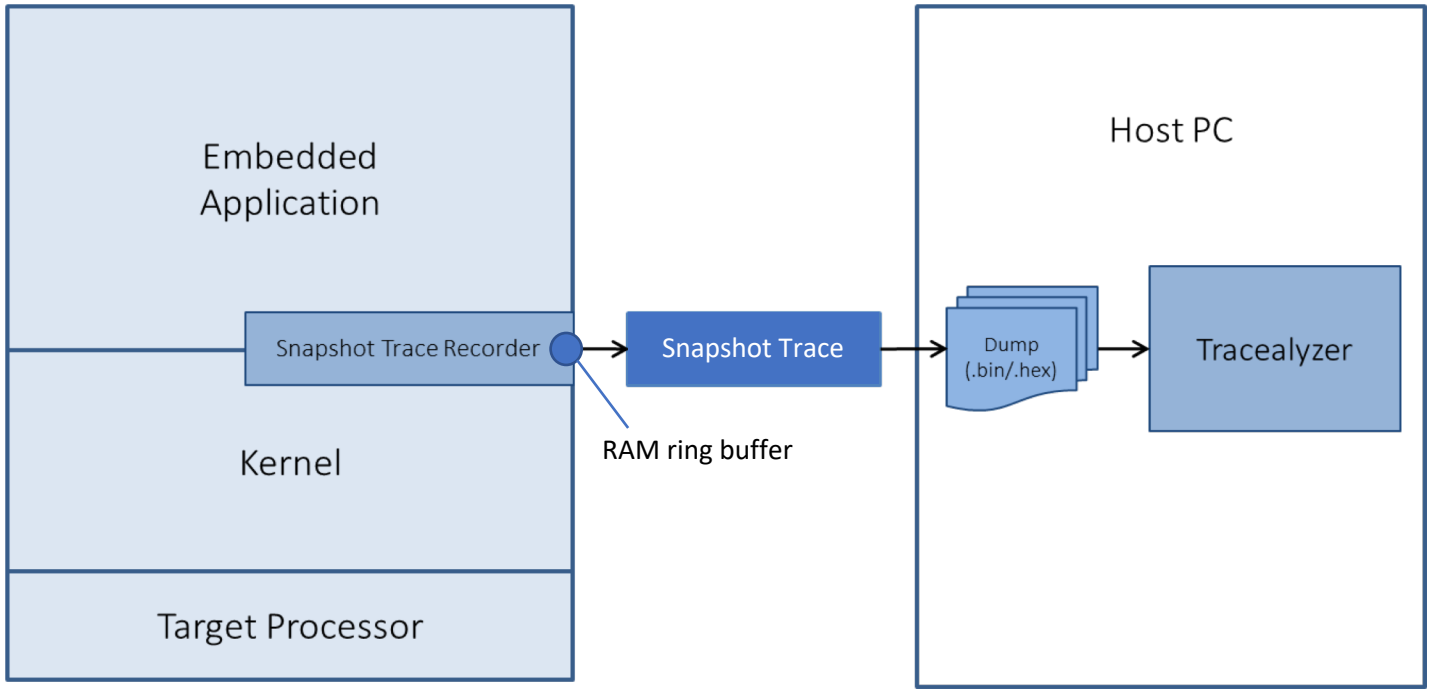
Optimization
Get more performance

Training
Explain how an OS or API works

Documentation
Visualize designs or issues

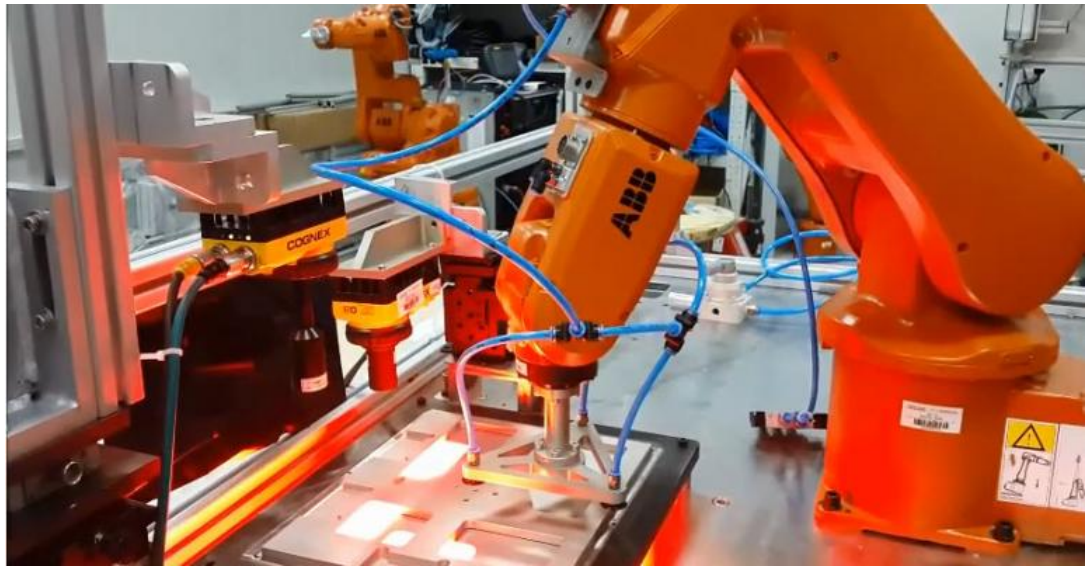
Methods for Event Tracing – Snapshot Trace

- Last events kept in RAM buffer
  - Typically a ring buffer
- Save a snapshot when desired
  - When halted on a breakpoint
  - Automatically on runtime errors
- Can be very memory efficient, allowing for use on MCUs.
  - Percepio's snapshot trace format uses only 4-8 byte per event
  - 5-10 KB trace buffer often sufficient



Snapshot Trace in Deployed Systems

- Example: ABB Robotics
- Snapshot tracing active at all times, also during customer operation.
- In case of errors, a trace is submitted to the developers for analysis.
- Benefits
  - Learn about every serious error directly, the very first time it occurs.
  - Get detailed information that pinpoints the problem, reducing the need for site visits and lab guesswork.

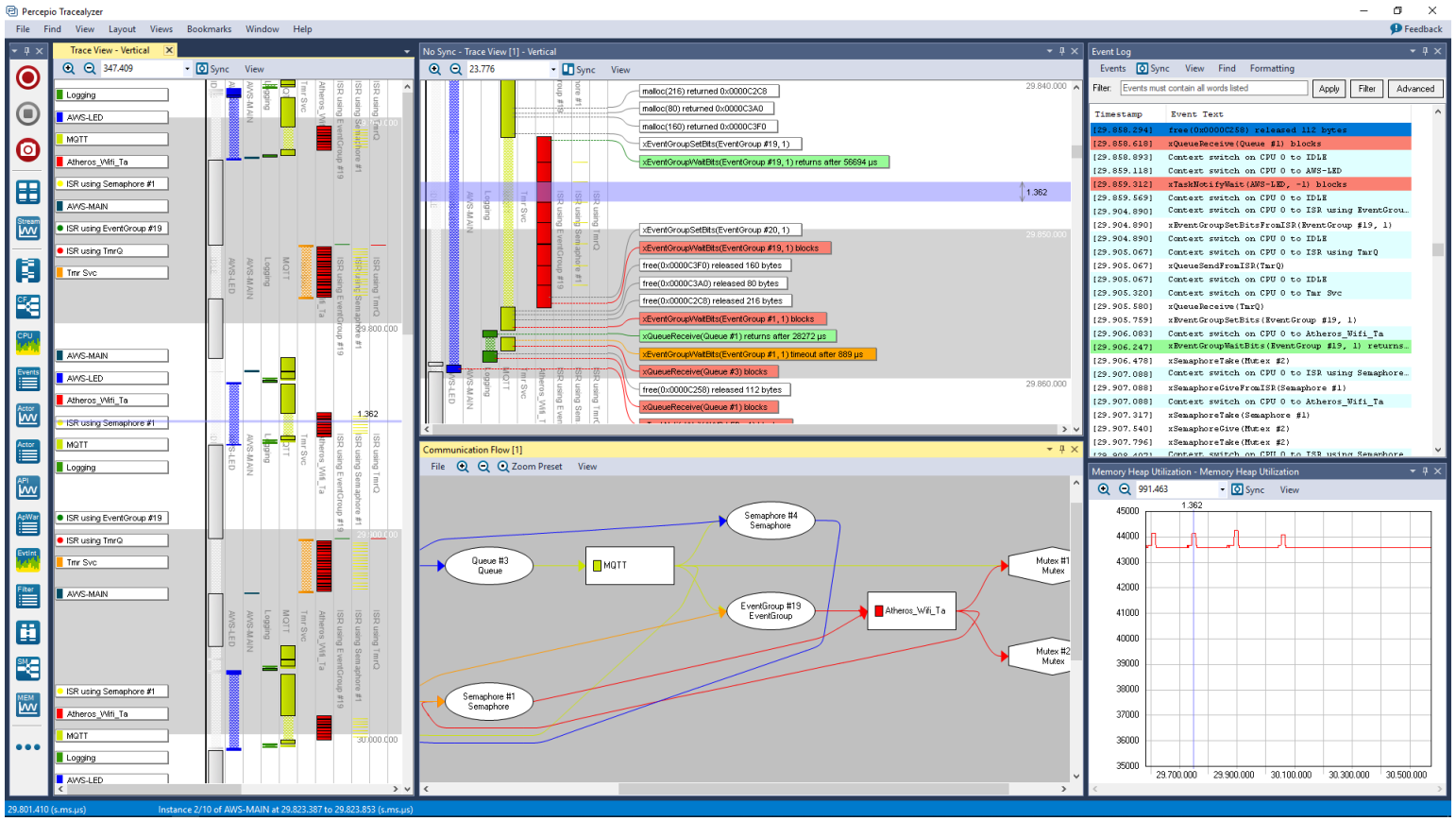


"ABB Robotics is using the first generation Tracealyzer in all of the IRC5 robot controllers shipped since 2005. The tool has proven its value many times in all corners of the world."

Roger Kulläng, Global System Architect, ABB Robotics

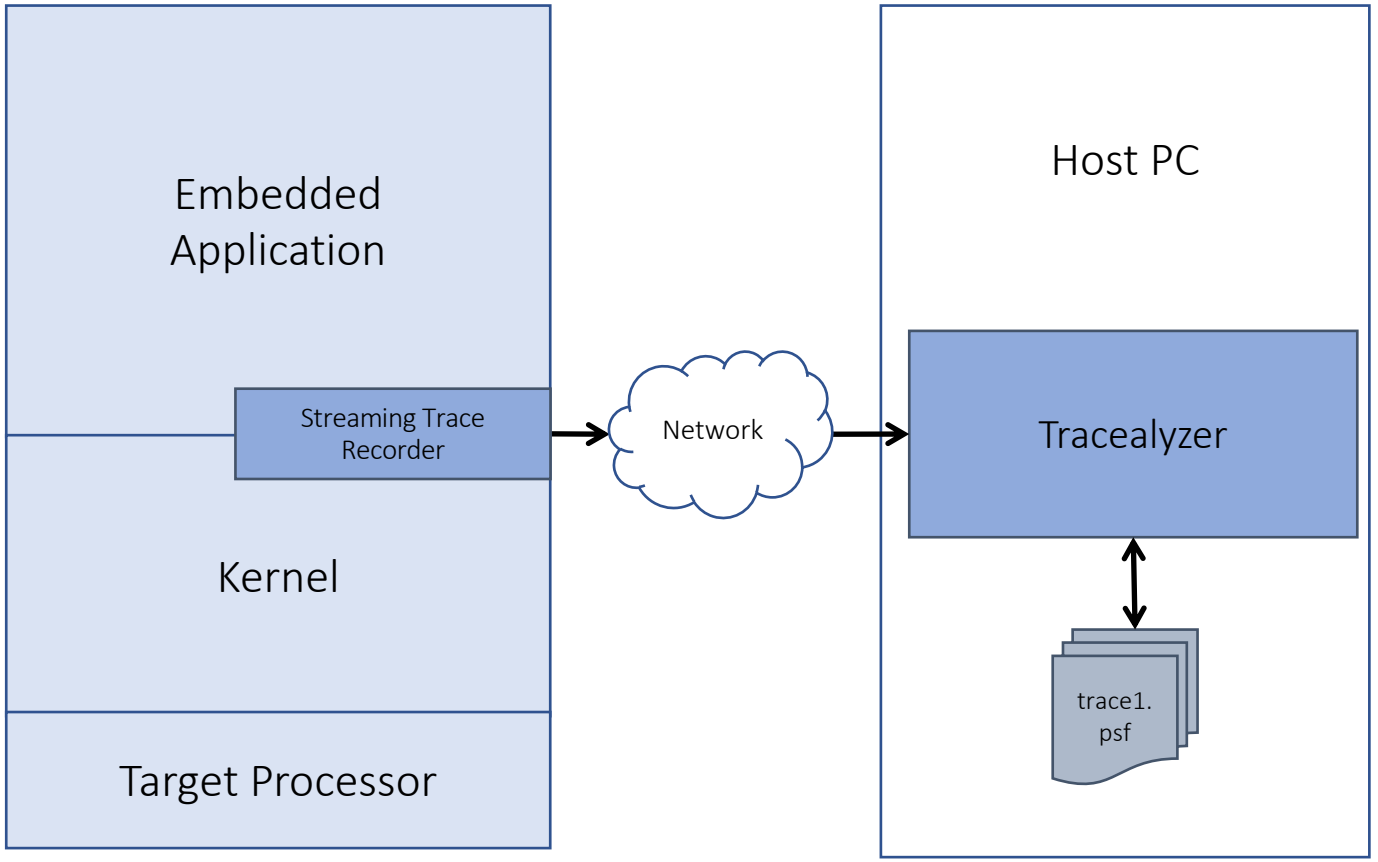
Snapshot Tracing on IoT Devices

- Perfect for IoT systems, where secure connectivity is already in place.
- In case of errors, upload a trace for analysis.
- Compact and valuable information
  - This is just **5 KB** of trace data
  - 350 ms trace, during a busy period, with many details.

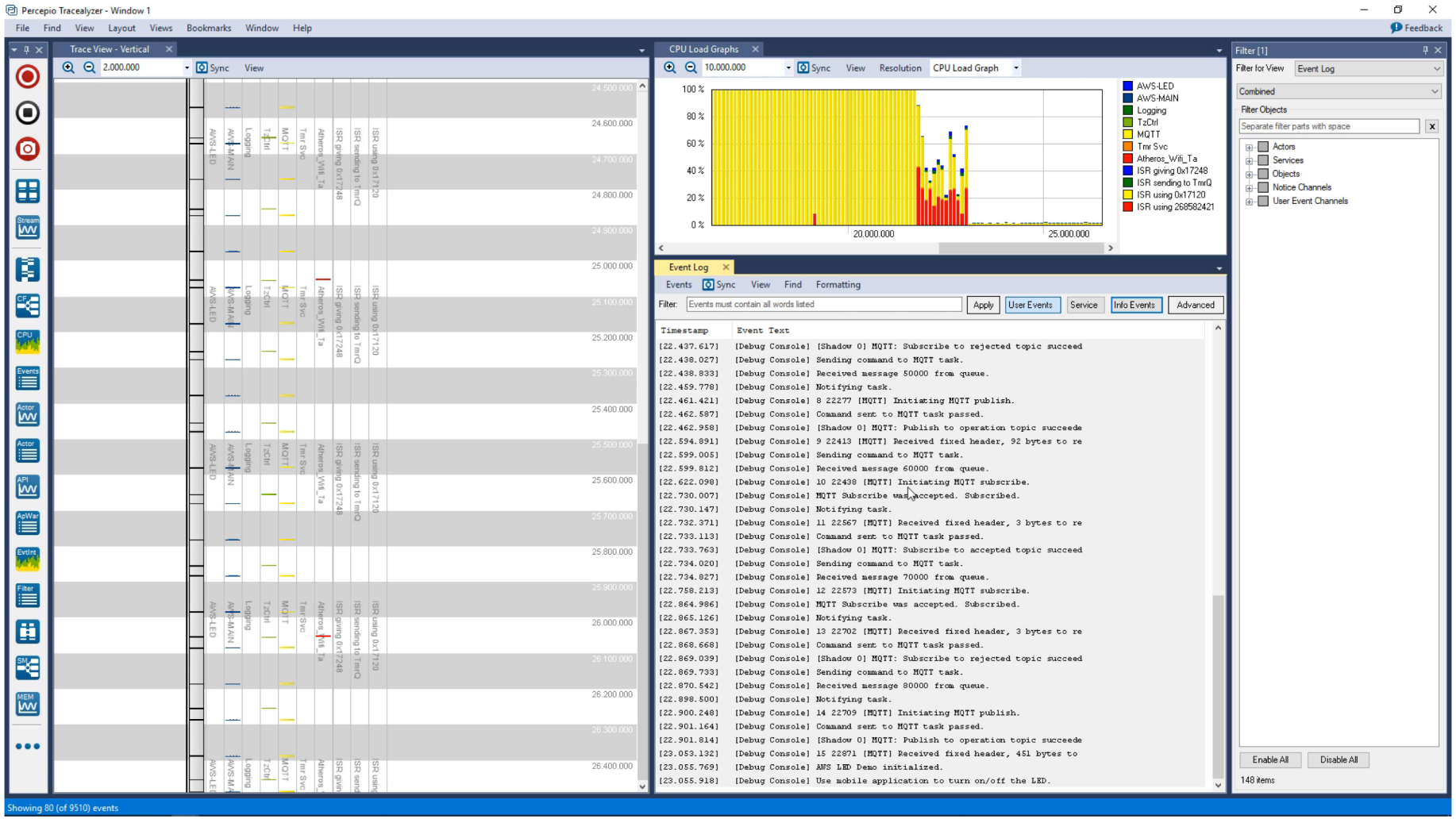


Event Tracing Method 2: Streaming Trace

- Continuous transfer
  - Allows unlimited trace length
  - We have tested up to 70 hours
- Can use any interface with decent throughput
  - Debug/trace ports, TCP/IP, USB or even UARTs.
- Can be combined with live visualization!

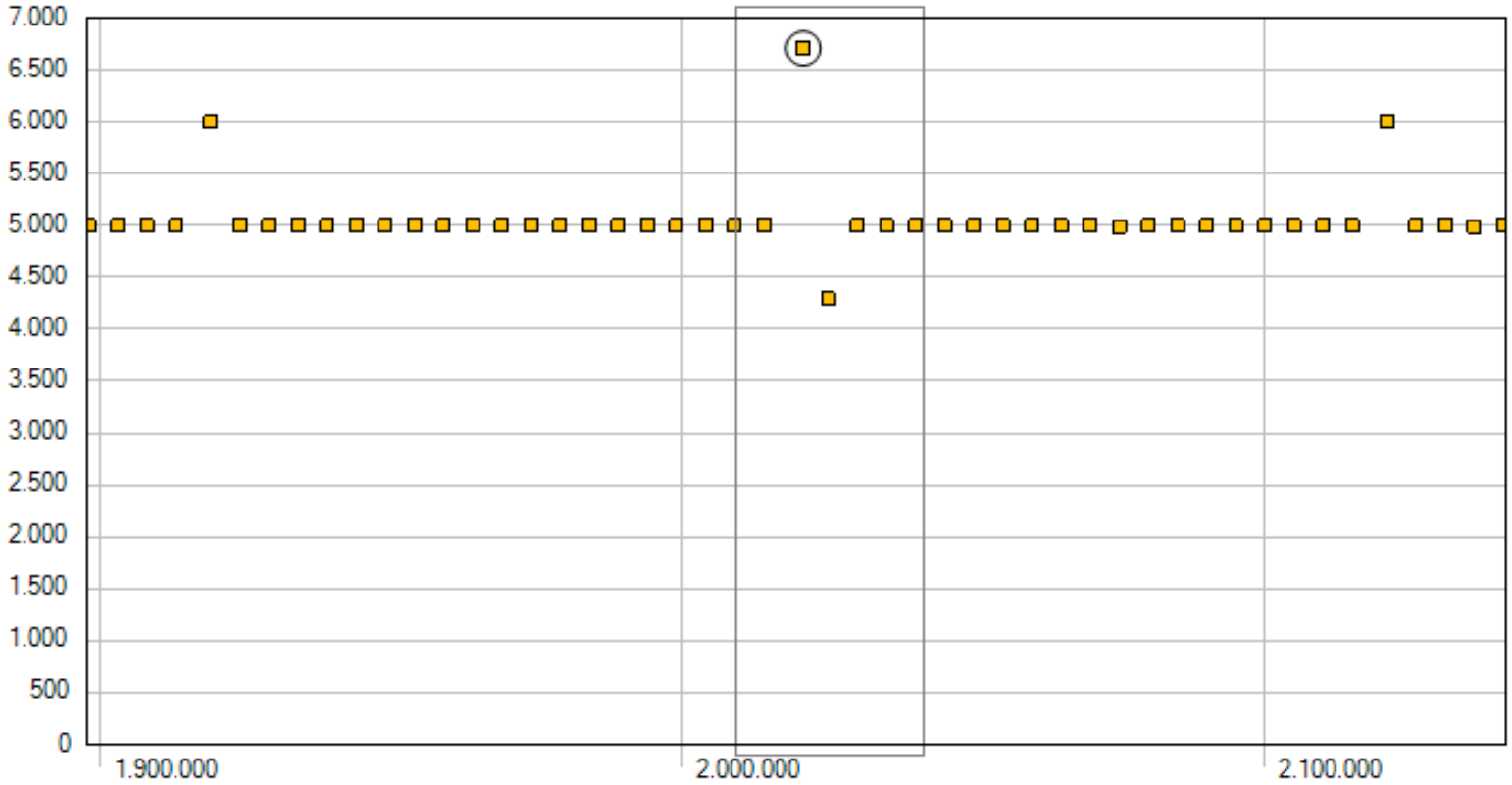


Example: Live Streaming



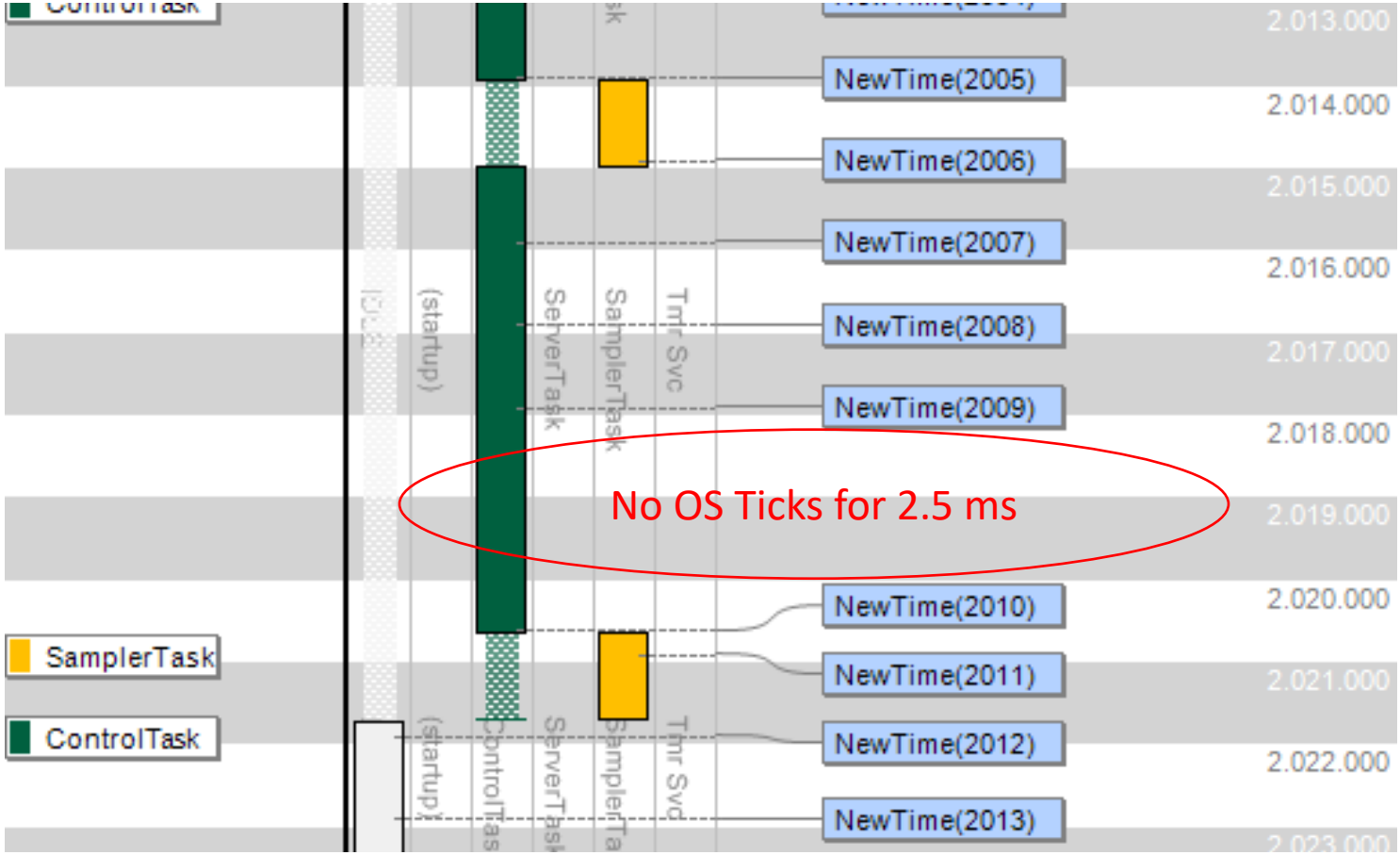
Customer Case 1 – Random Timing Variations

Periodic task should execute every 5 ms…



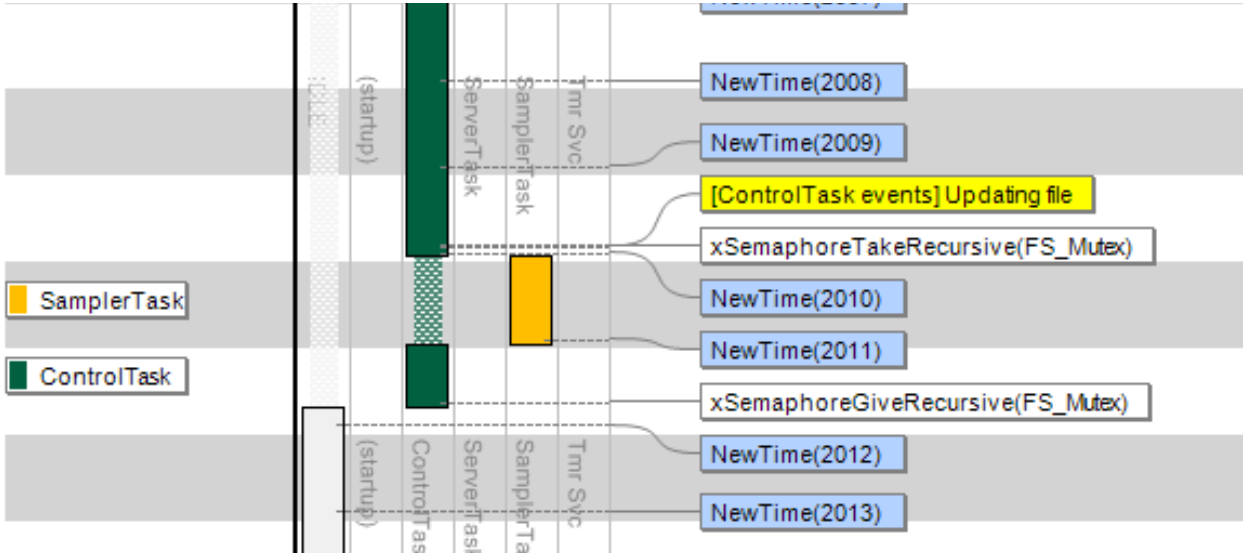
Y-axis shows time between activations. There are random variations between 4-7 ms

When looking closer at these locations...



ControlTask seems to disable interrupts, thereby disabling the RTOS scheduler!

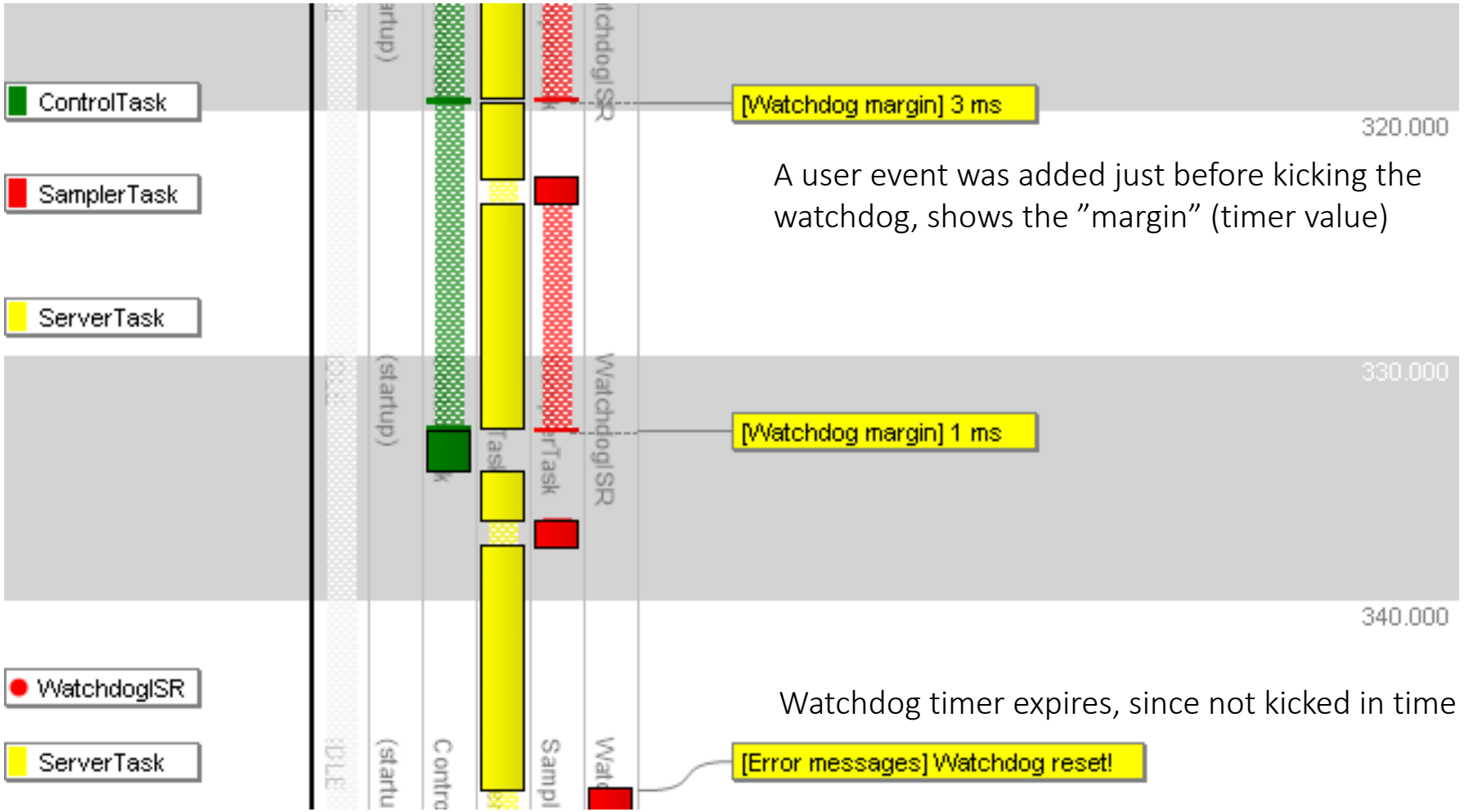
Don’t disable interrupts, use a Mutex instead!



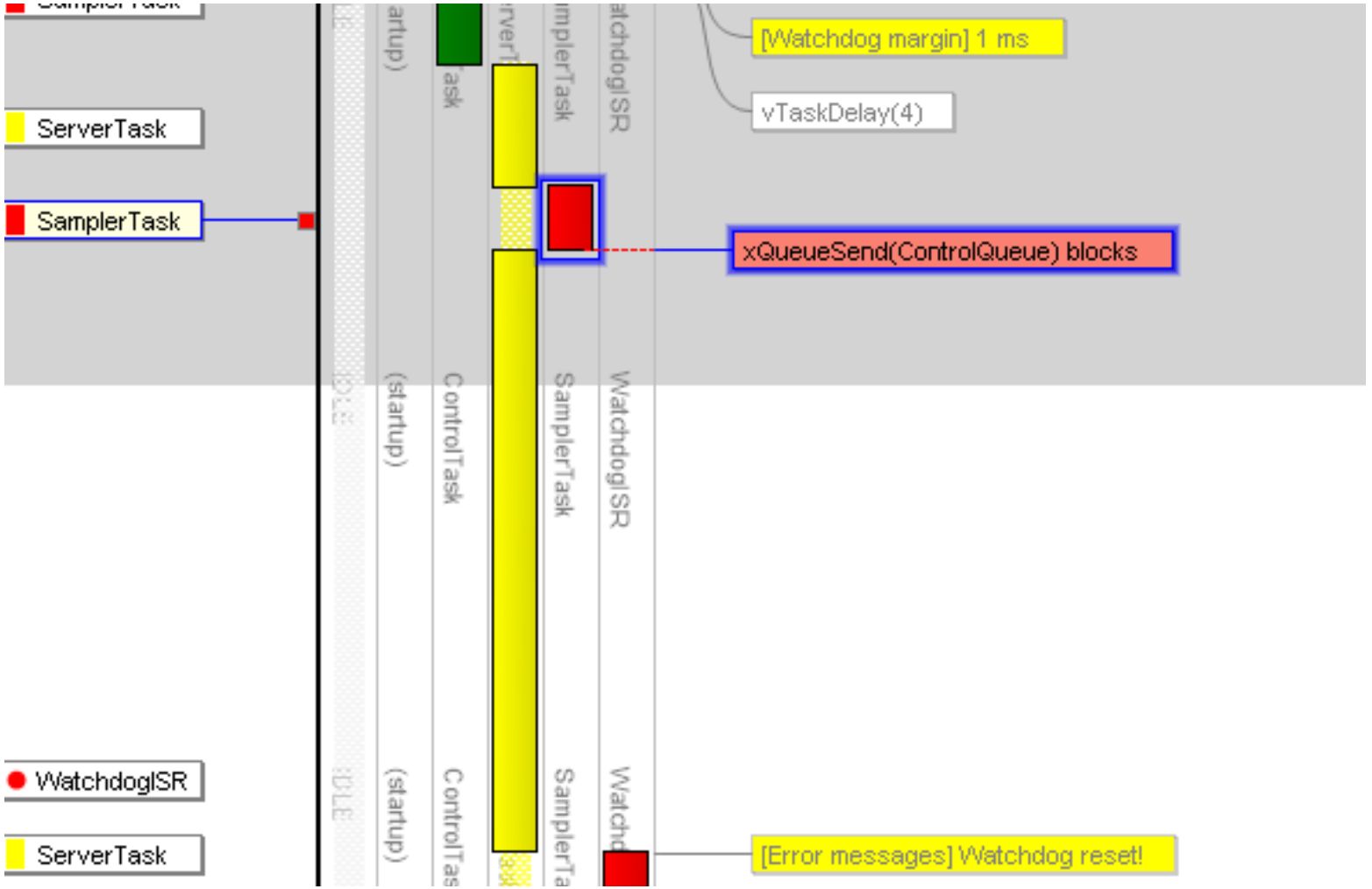
Now perfect 5 ms periodicity!

Customer Case 2 – The Watchdog Reset

Sometimes a Watchdog Reset occurs, why?



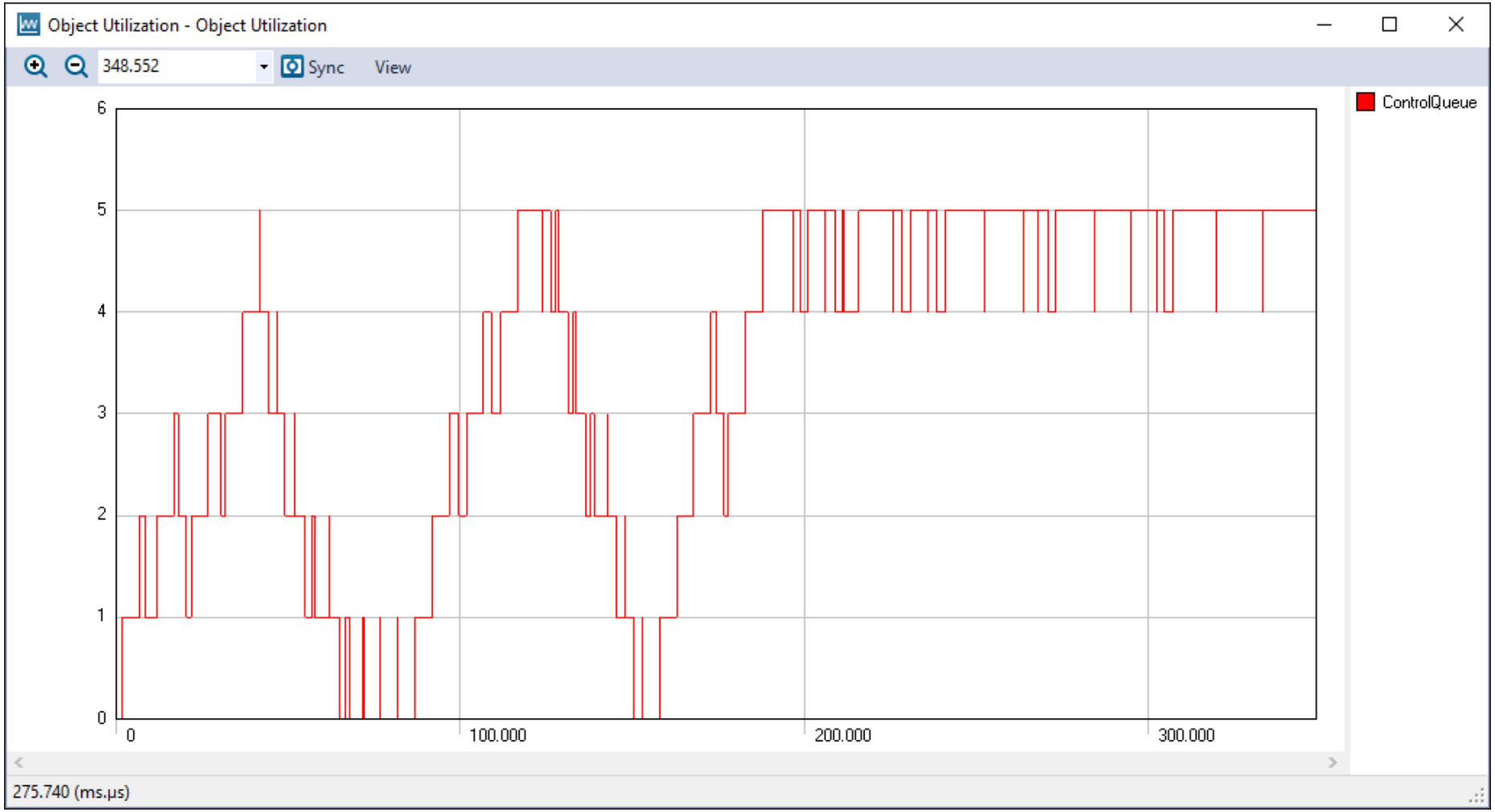
Why doesn’t SamplerTask kick the Watchdog?



Clue

Supposed to kick the watchdog, but is blocked on a message queue send.

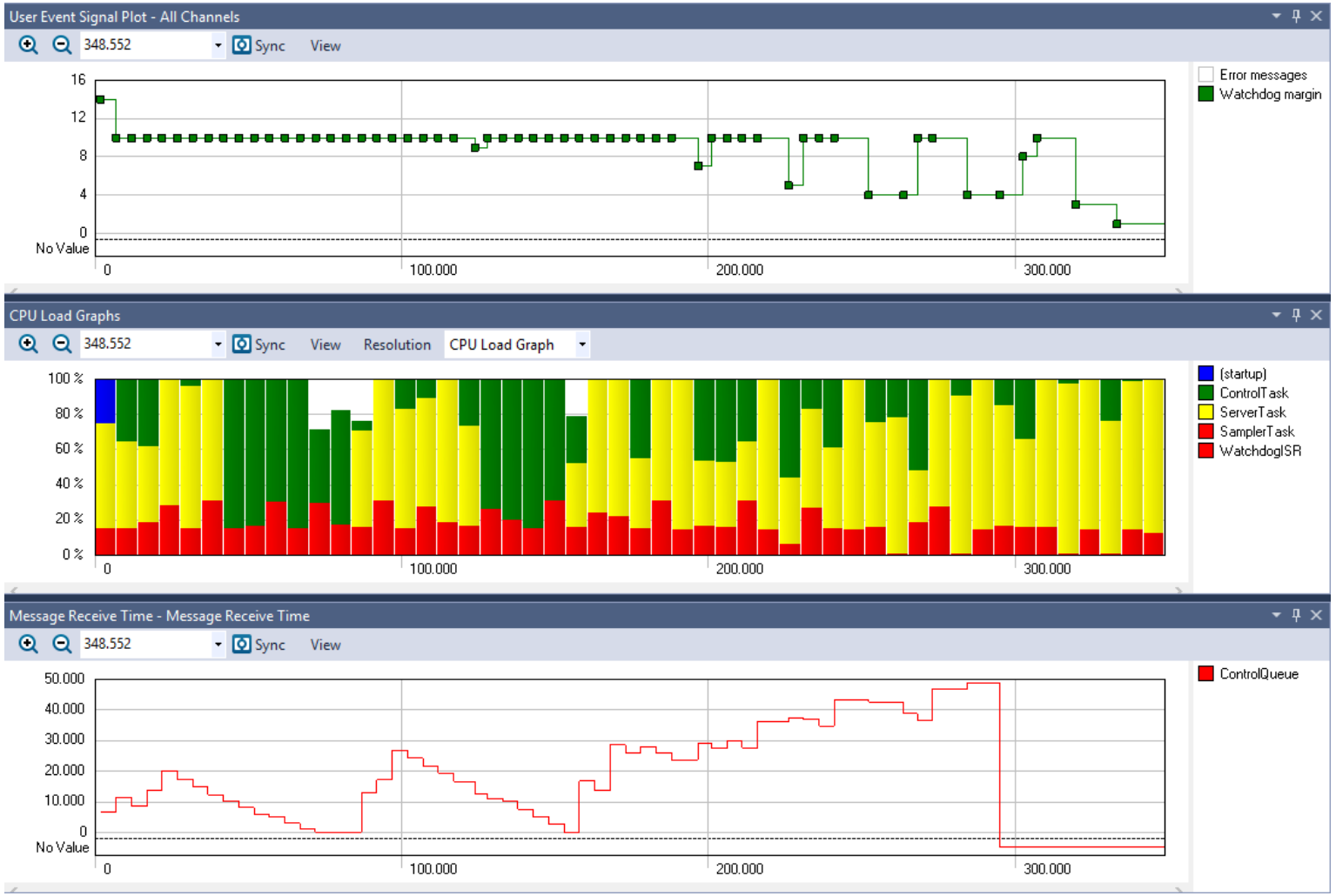
Why is ControlQueue blocking?



Clue

The queue seems to only allow 5 messages, so blocks because it gets full!

So ControlQueue gets full… Why?



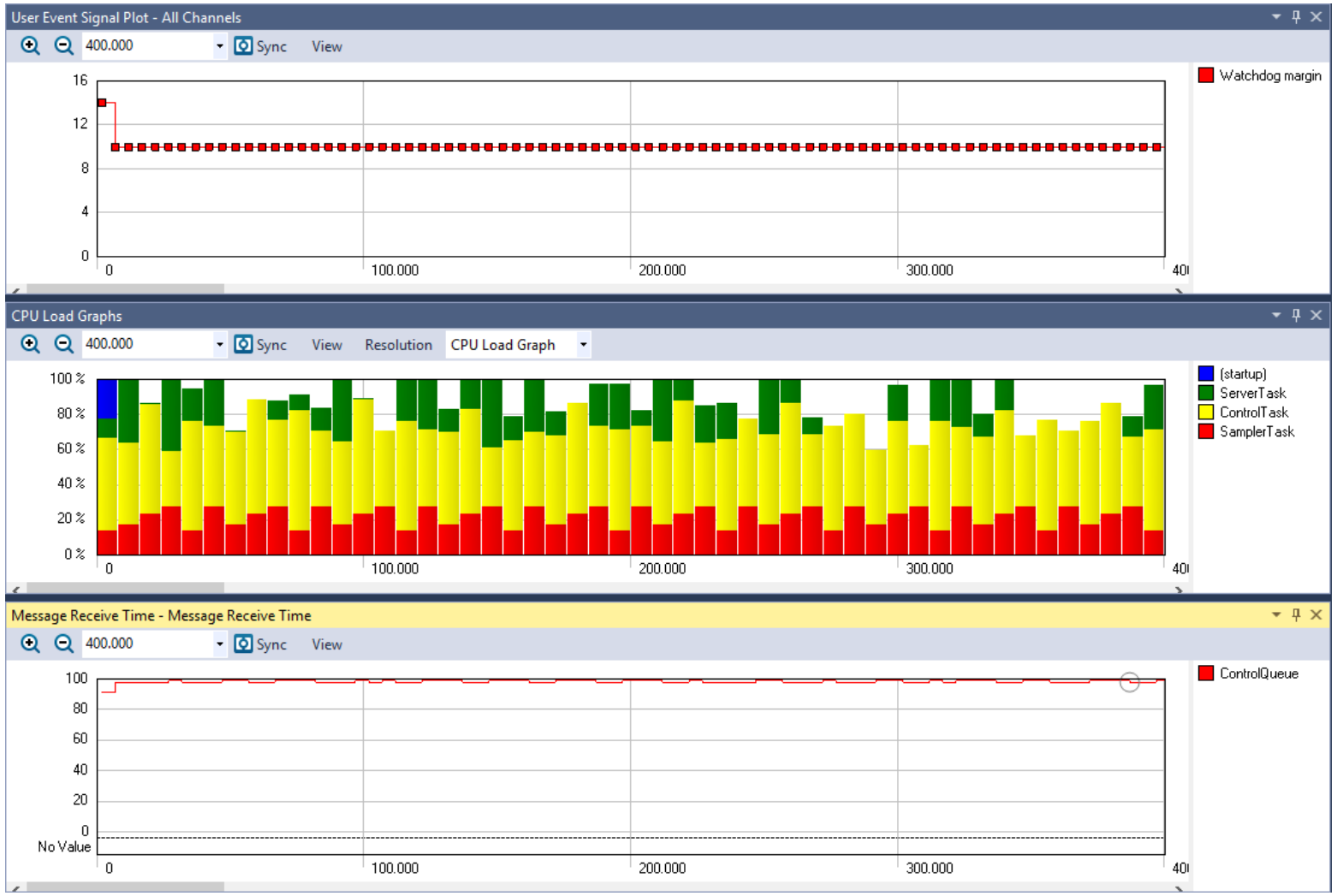
Symptom
Watchdog ”margin” decreases…

Clue 1
ControlTask (green) gets less CPU time. This since ServerTask (yellow) uses more CPU time and has higher scheduling priority.

Clue 2
Messages are buffered in ControlQueue for longer time (send -> receive).

Conclusion
ControlTask can’t keep up reading the messages due to ServerTask’s higher priority.

Swapped Task Priorities – Problem Solved!



Summary – Customer Case 2

Thank you! Questions?