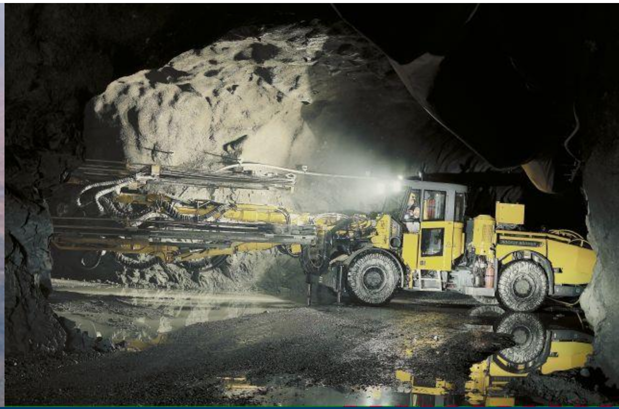
  

Common RTOS-related bugs

How avoid and detect



Dr. Johan Kraft, CEO/CTO/founder, Percepio AB
johan.kraft@percepio.com

percepio
SENSING SOFTWARE

Real-Time Operating Systems

- A base software platform for your firmware
- Provides multithreading
  - Tasks – Separate threads of execution
  - Supporting services - Semaphores, Queues, Timers, etc.
- An RTOS is fast, compact and deterministic
  - Common also on (32-bit) MCUs
- Many exists, some more common
  - FreeRTOS, µC/OS, ThreadX, VxWorks…

RTOS multi-tasking

"Superloop" design

```
while(1){
  if (condition1){
    Func1();
  }

  if (condition2){
    Func2();
  }

  if (condition3){
    LowPowerMode();
  }else{
    Sleep(10)
  }
}
```

RTOS system

```
/* Task 1 */
while(1){
  DelayUntil(Time + 10);
  Func1();
}
```

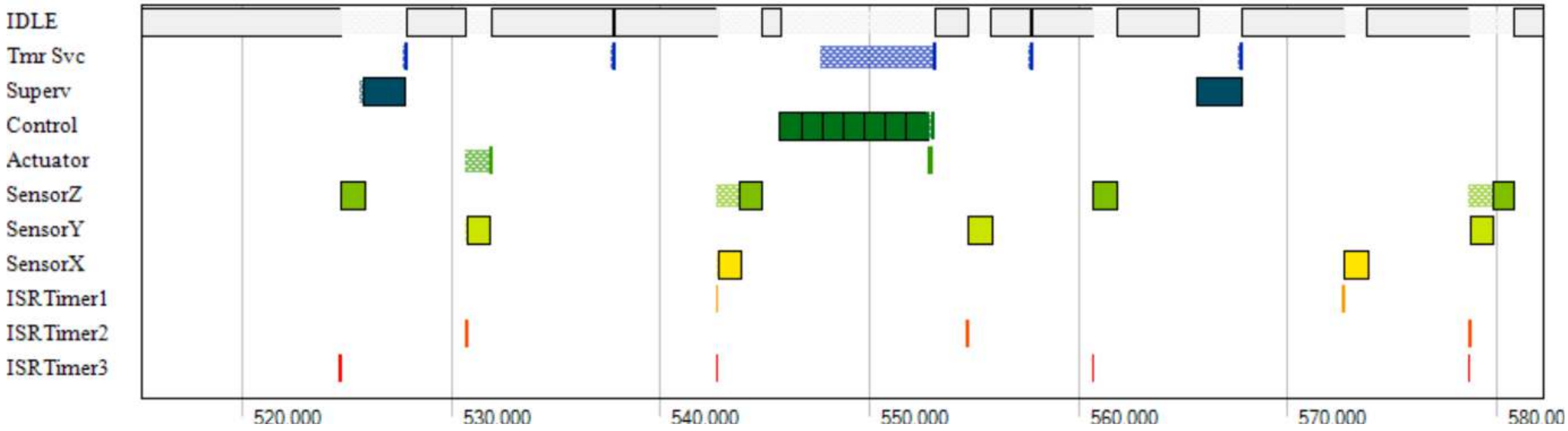
```
/* Task 2 */
while(1){
  WaitForEvent(B);
  Func2();
}
```

```
/* Idle task */
while(1){
  LowPowerMode();
}
```

Each task has:

- Separate execution context (stack and registers)
- Fixed scheduling priority (relative urgency)
- Scheduling status (ready/waiting)

Runtime view: RTOS multi-tasking



(Example from Percepio Tracealyzer)

Most RTOS use fixed priority, pre-emptive scheduling:

- Always selects task with highest priority, that is ready to execute
- May use ”round-robin” (alternate between tasks) if same priority
- The RTOS can pre-empt a running task at any point, to let a higher priority task start.

Signaling a task using a semaphore

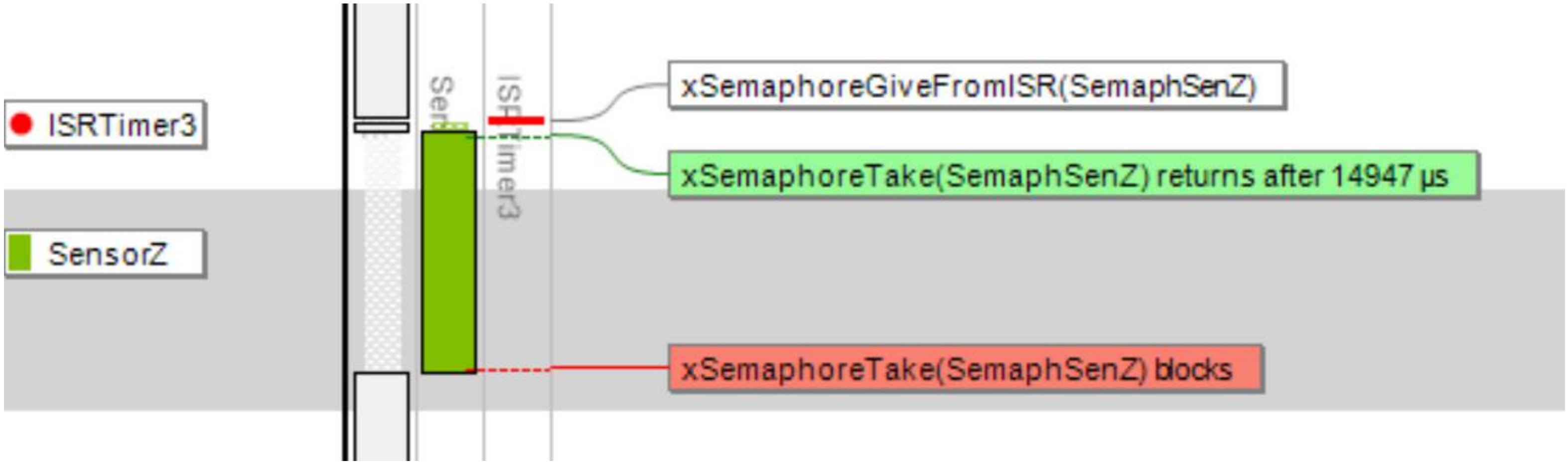
```
/* Task A */
...
xSemaphoreGive(SemB);
...
```

```
/* ISR X */
...
xSemaphoreGiveFromISR(SemB);
...
```

Semaphore ”SemB”

```
/* Task B */

while(1){
  /* Waits here, until signaled */
  xSemaphoreTake(SemB, FOREVER);
  DoMyThing();
}
```

Runtime view: semaphore



(Example from Percepio Tracealyzer)

percepio SENSING SOFTWARE

Passing data using message queues

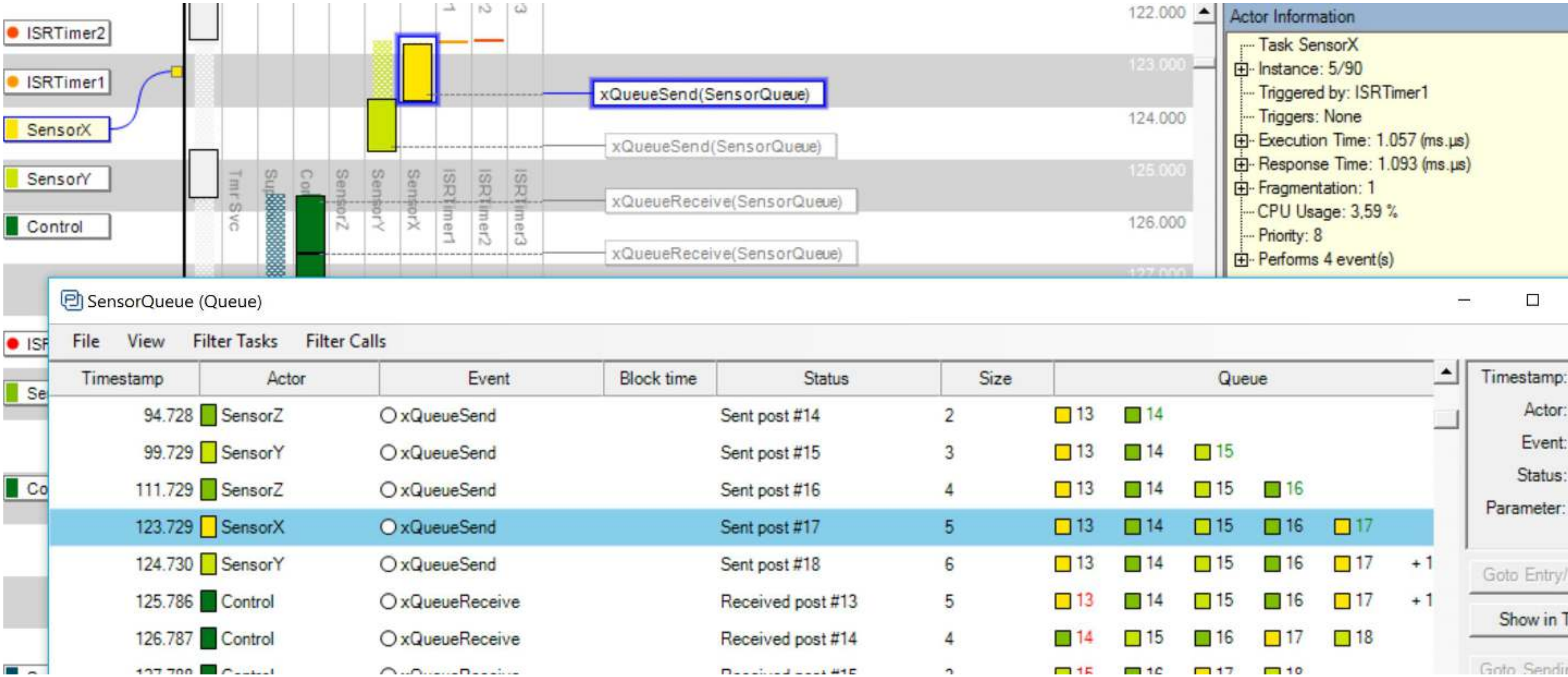
```
/* Task A */
...
msg.cmd = CMD1;
msg.param1 = x;
msg.param2 = y;
xQueueSend(qB);
...
```

```
/* ISR X */
...
msg.cmd = CMD2;
msg.param1 = a;
xQueueSendFromISR(qB);
...
```

Message queue ”qB”

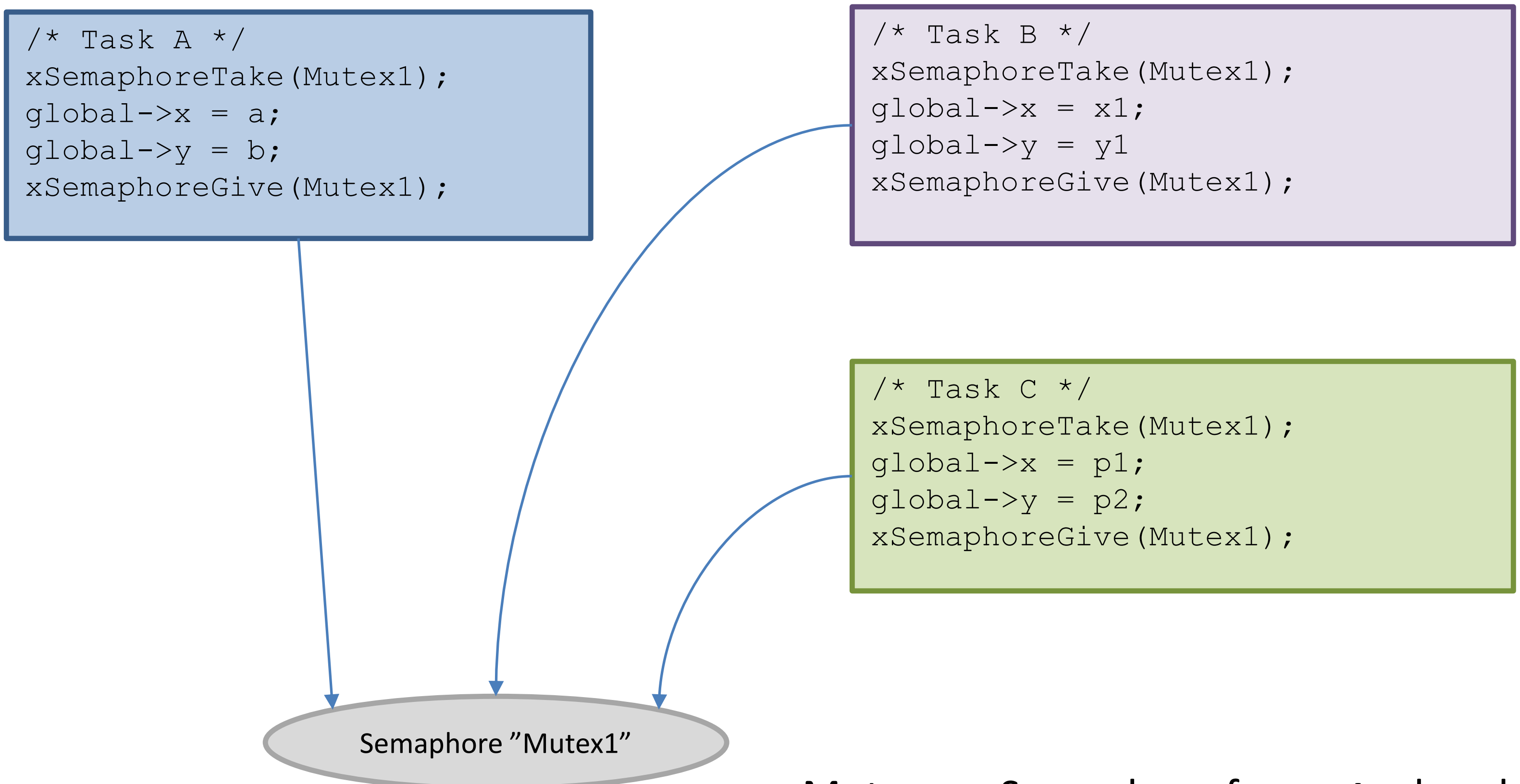
```
/* Task B */

while(1){
  /* Waits until message or timeout */
  sts = xQueueReceive(qB, &msg, 10);
  if (sts == TIMEOUT){
     HandleTimeout();
  }else switch(msg.cmd){
     case CMD1: HandleEvent1(); break;
     case CMD2: HandleEvent2(); break;
  }
}
```

Runtime view: message queues



(Example from Percepio Tracealyzer)

Sharing resources using a mutex



Mutex – a Semaphore for **mut**ual **ex**clusion
(try to avoid, but sometimes necessary!)

Sharing resources using a dedicated task

```
/* Task A */
...
msg.cmd = SEND;
msg.data = x;
msg.len = sizeof(x);
xQueueSend(TX_Queue);
...
```

```
/* Task B */
...
msg.cmd = SEND;
msg.data = y;
msg.len = sizeof(y);
xQueueSend(TX_Queue);
...
```

”TX_Queue”

```
/* Task TX_Task */

while(1){
   /* Waits until message or timeout */
   sts = xQueueReceive(TX_Queue, &msg, FOREVER);
   ...
   switch(msg.cmd){
      case SEND: tx_write(msg.data, msg.len);
                 break;
      ...
   }
}
```

RTOS Benefits:
Easier to design complex applications

- Easier to handle multiple interfaces (TCP/IP, USB, HMI…)
  - One task for each purpose…
- Easier to pass data between ISRs and application
  - Safely! (home-cooked solutions may not be)
  - Reduce ISR processing time – let a task do the work
- Easier to maintain and extend
  - Tasks allow for modular design
  - Easy to add new tasks, independent of period or trigger

RTOS Benefits: More efficient design

- Avoid wasting cycles on inefficient polling
  - Tasks sleep individually, wakes up on the right RTOS event.
- Save energy using Low Power Modes
  - Use the Idle Task to enter LPM, using e.g. “wfi” instruction.
  - Tickless Idle – disable the RTOS tick interrupt.
- More responsive system – shorter interrupt latency
  - Minimize ISR time by delegating jobs from ISRs to tasks.
  - Activate the task from the ISR, using a semaphore
    - Task starts immediately, thanks to pre-emptive scheduling

RTOS Overhead

- Code (ROM)
  - Typically 5-10 KB
- Data (RAM)
  - 200-300 bytes for common kernel data
  - ~128 byte per task stack + ~50 bytes for task control block
- Processor time
  - Task-switches take 100-200 clock cycles (a few thousand times/sec)
  - Periodic OS tick – very small impact in itself
- Interrupt latency
  - May increase due to critical sections in RTOS kernel
  - Time-critical ISRs can be allowed to pre-empt the RTOS kernel, if they don't use any RTOS services.

RTOS Challenges - Learning curve

- An RTOS introduces a new abstraction level – tasks
  - You are no longer in direct control over the code execution!
- You need to design how the tasks interact and share data
  - When to use a semaphore, mutex, message queue, etc.
- You need to decide suitable task priorities
  - Relative urgency – not always obvious
- You need to understand
  - The general principles
  - Best practices and common pitfalls
  - The API and configuration of your RTOS

RTOS Challanges - Test & Debug

- The system behavior is not apparent from the source code
  - Timing and RTOS scheduling is not visible!
- Task-switches are often asynchronous to the program flow
  - Strikes at different locations, depending on “random” variations in input timing and execution times
  - There can be a enormous number of possible execution scenarios, with different timing and execution order
- Why do I need to worry about this?
  - Bugs may depend on timing, very difficult to find and reproduce!
  - Risk for “nightmare bugs” that only appear under special conditions
  - Most debug tools provide little support for multi-tasking issues

Symptoms of RTOS-related bugs

- Tasks works fine in isolation but not as a full system
- Slow performance
- System locks up, or sometimes stops responding
- System appears brittle – minor changes results in weird errors
- Random variations in output timing
- Sometimes corrupted data, or wrong output
- Random crashes/hard-faults

Problem: Stack overflow

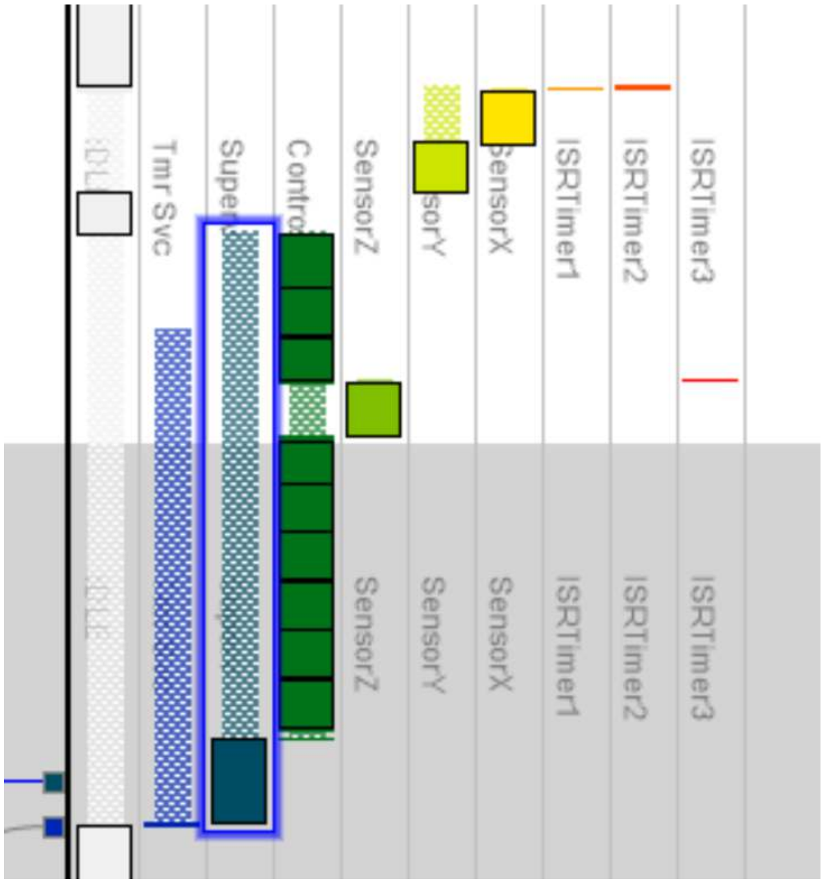
- Symptom: Strange behavior, hard faults (crashes)
- Problem: Each task has a separate stack, if not large enough the stack may accidentally overwrite other data…

How avoid stack overflow

- Check the “high watermark” of the stack usage for each task after extensive testing, make sure there is some safety margin
- Make sure to enable stack overflow detection in your RTOS
- Some IDEs can calculate the worst case stack usage
- Don’t use recursion! :-)

Problem: Task starvation (slow response)

- Symptom: One or several tasks runs slow, or not at all
- Problem: Higher priority tasks use too much processor time, not enough remaining for the lower priority tasks.

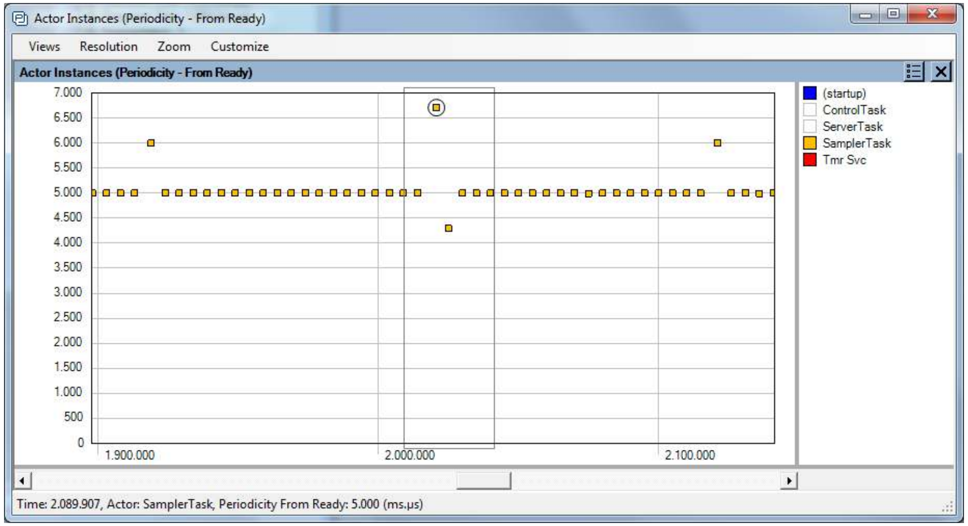


How avoid task starvation

- Avoid polling/busy wait and make sure to put tasks to sleep after completion (delay, wait for semaphore…), so other tasks of lower priority can execute.
- Use higher priorities <u>only</u> for tasks with predictable execution pattern and shorter execution times
- Tasks triggered by external events and/or longer execution times should have <u>lower</u> priorities
- Divide longer jobs into multiple task, with appropriate priority
- Rate monotonic schedulability analysis?

Problem: Task jitter

- Symptom: Disturbances in the timing of periodic tasks
- Problem: The execution of a task is sometimes delayed, by higher priority tasks or by ISRs.

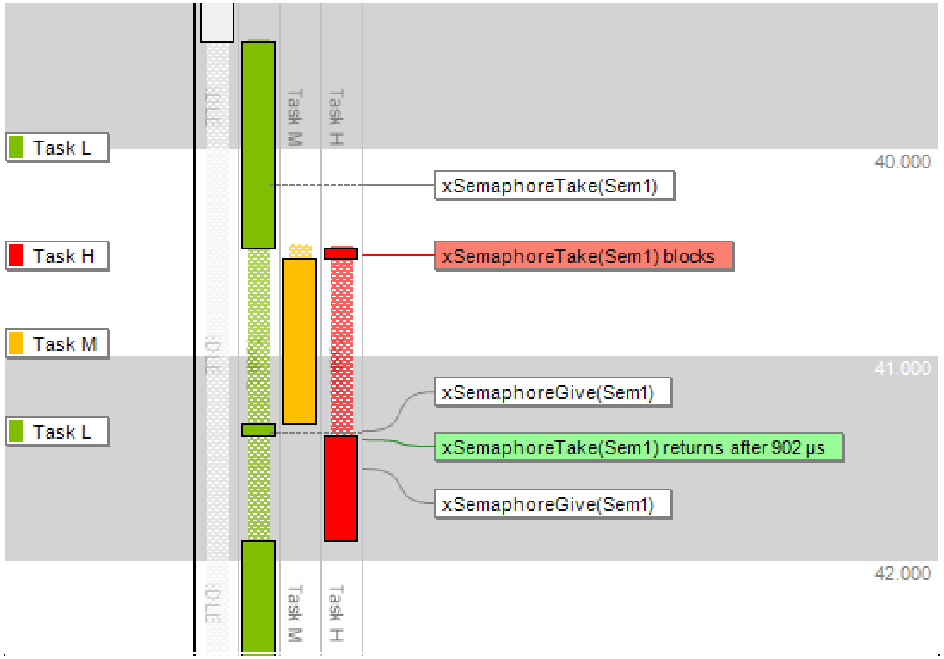


How avoid task jitter

- Make sure to use preemptive scheduling
- RTOS tick rate should be a lot higher than the shortest task period
- Don’t disable interrupts to protect critical sections
  - Disables the RTOS!
  - Use mutexes, or let a dedicated task manage the resource.
- If disturbance is from higher priority tasks
  - Change priorities?
  - Add an offset to the execution, so they don’t overlap?
  - Reduce their execution time?
- If disturbance is from ISRs
  - Reduce their execution time, e.g., delegating processing to tasks.
  - Put time-critical code in high-priority ISR, driven by periodic timer.

Problem: Priority Inversion

- Symptom: High priority task is delayed by lower priority tasks
- Problem: Mutex held by lower priority task, gets preempted and delayed by mid priority task.
- Can also occur with queues and other blocking objects



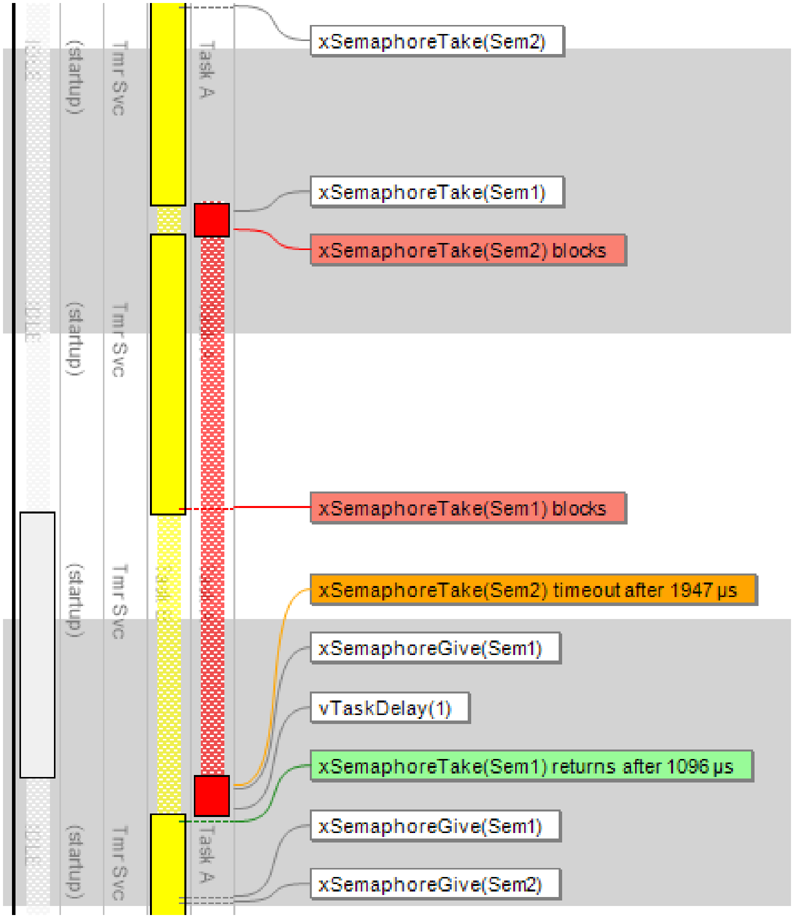
How avoid priority Inversion

- Avoid sharing resources between tasks (e.g., using mutexes)
  - Have a dedicated task that manage each resource
- If sharing is required, use Mutexes with "Priority Inheritance"
  - If a high-priority task H is waiting for a resource, held by a lower-priority task L, the RTOS temporarily raises the priority of task L to avoid pre-emption by irrelevant middle-priority tasks.
- Generally, use a single blocking point per task (to get input)
  - Avoid mutexes…
  - Avoid other blocking, e.g., when writing to a full message queues
    - Set timeout 0, check return value and handle any error

Problem: Deadlock

- Symptom: Multiple tasks suddenly stop to execute
- Problem: Circular wait on blocking kernel calls

- “Solved” by timeout here, but this can hide the problem!



How avoid deadlock

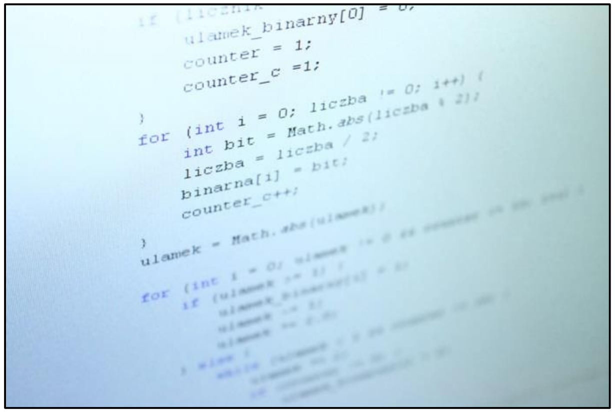
- Avoid critical sections…
- Especially avoid multiple nested critical sections, using two or several mutexes at the same time!
- But if required, make sure that:
  - All tasks locks and unlocks the mutexes in the same order, and
  - The unlocking should be inverted to the locking order.

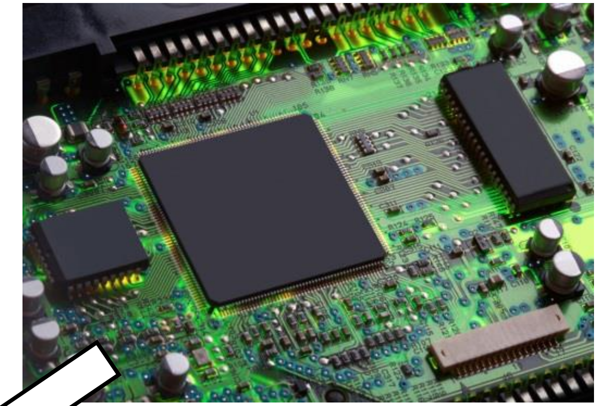
```
Task 1               Task 2
Lock MutexA          Lock MutexA
Lock MutexB          Lock MutexB
…                    …
Unlock MutexB        Unlock MutexB
Unlock MutexA        Unlock MutexA
```

How detect RTOS bugs

- Diagnostic features in your IDE
  - Stack calculation features
  - RTOS-aware debugger (inspect object states)
- Diagnostic features in your RTOS
  - Return value from API calls
  - CPU usage statistics (per task)
  - Stack diagnostics – high watermark and overflow detection
- But to see a timeline, you need tracing!

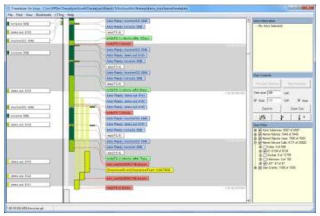
RTOS-aware tracing

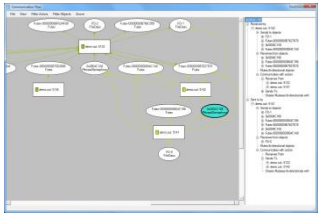
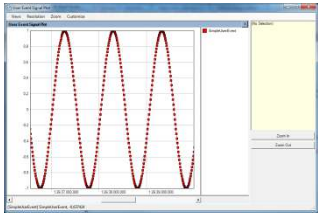




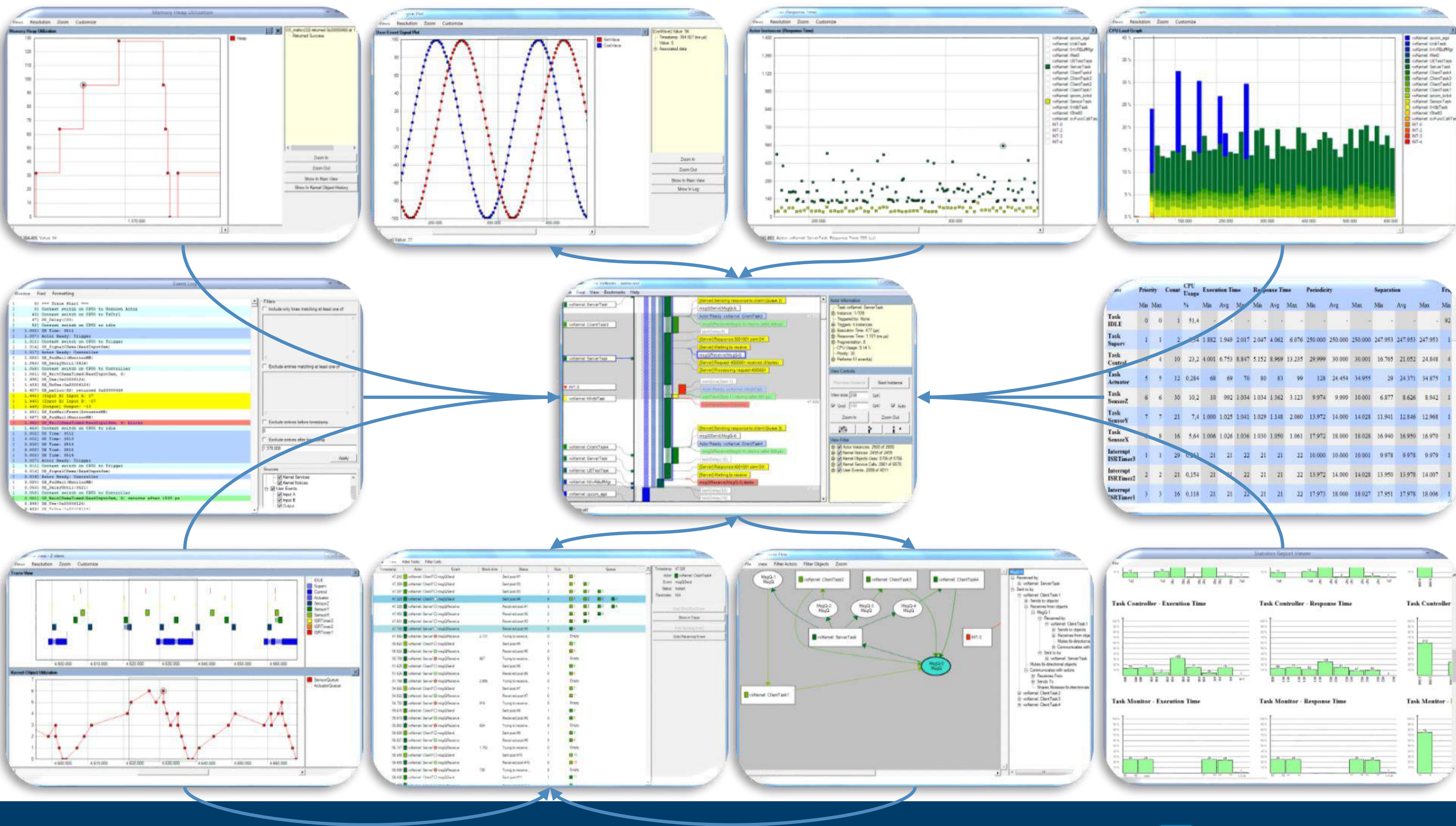
Software-defined Tracing

Trace recorder library – stores RTOS events like task switches and API calls. Also allows for application logging

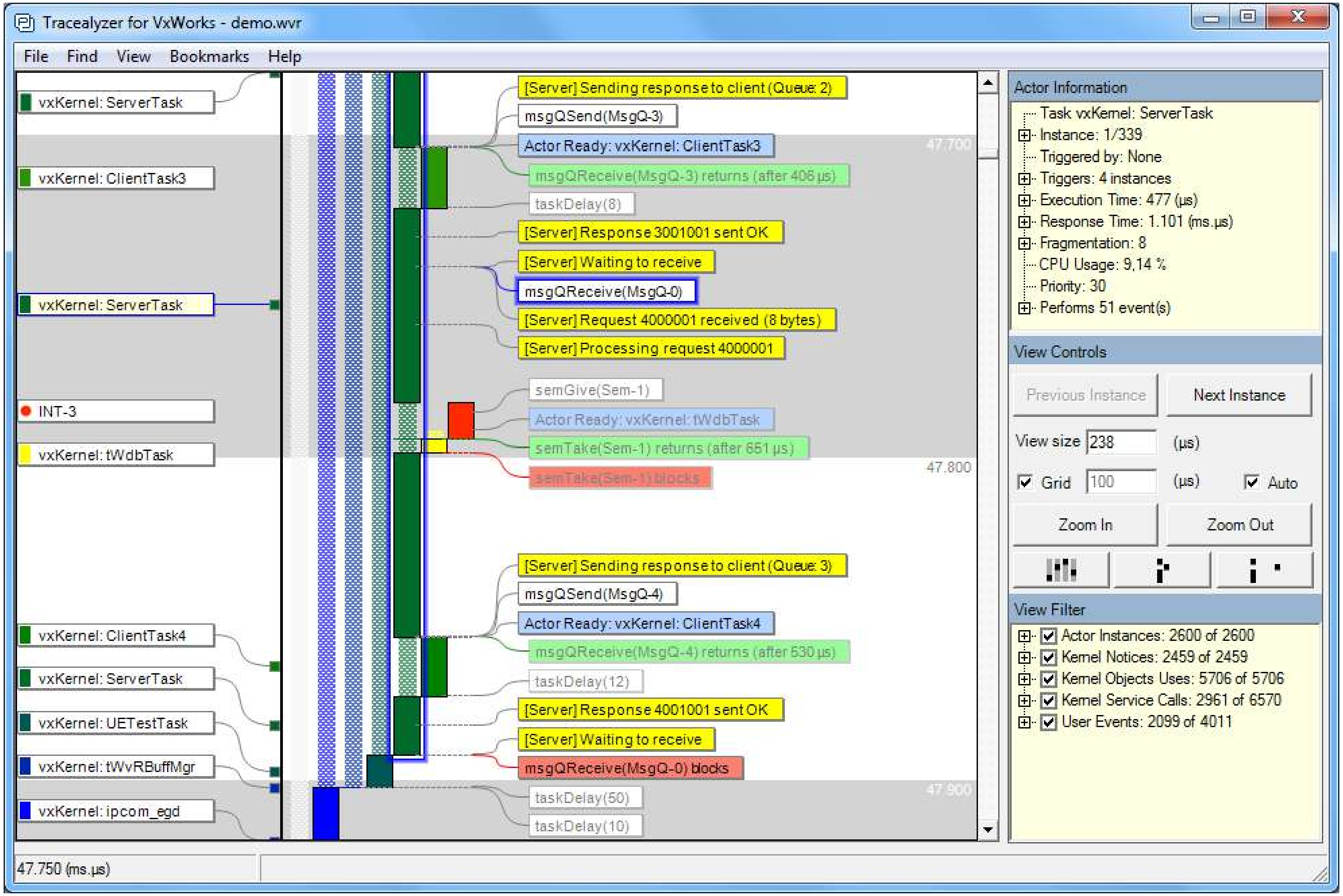


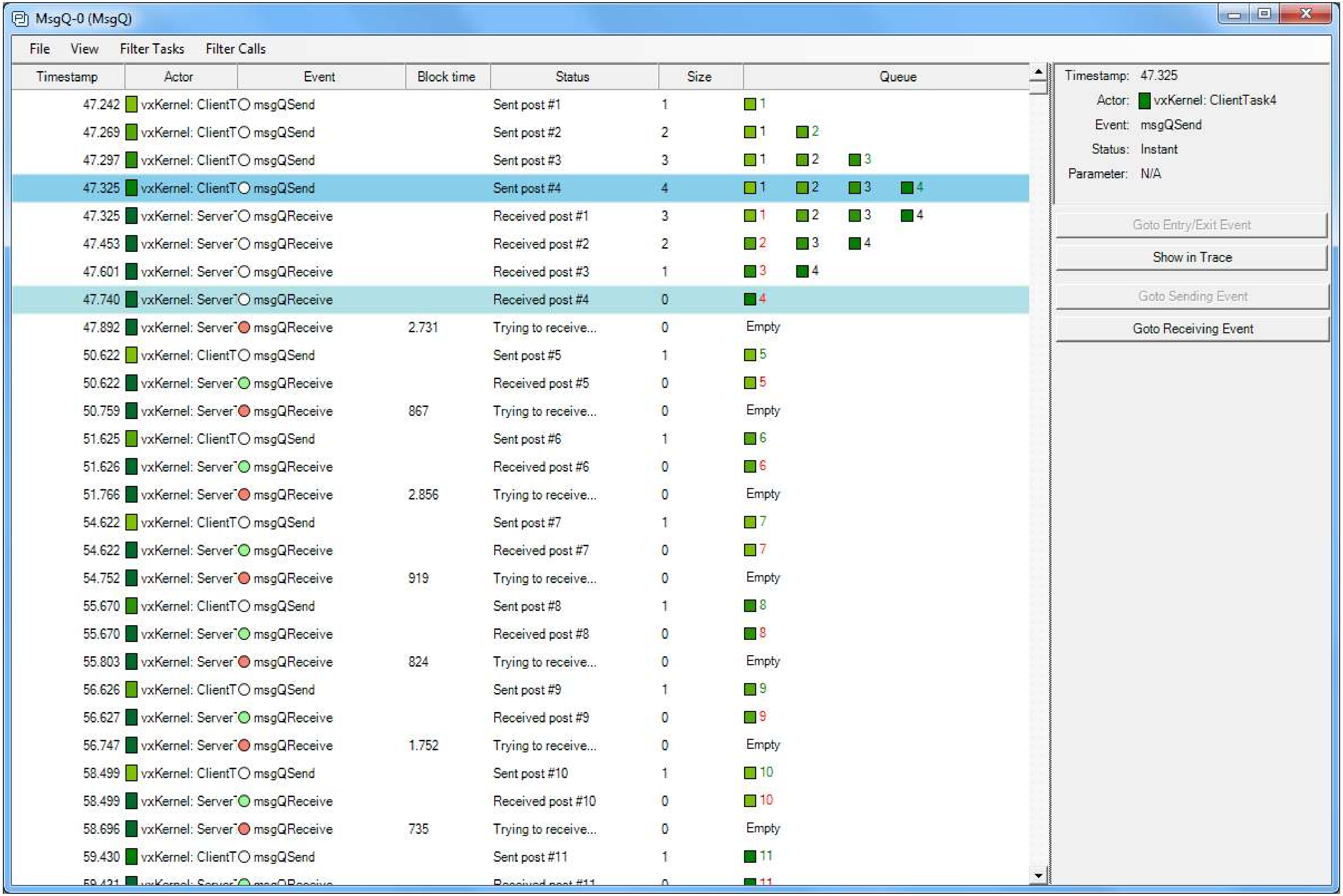
RTOS Trace Visualization



Tracealyzer - Main View



- Task scheduling
- Preemptions
- Interrupts
- RTOS API calls
- Blocking
- Resumes
- Timeouts
- RTOS Tick
- User Events



MsgQ-0 (MsgQ)

File | View | Filter Tasks | Filter Calls

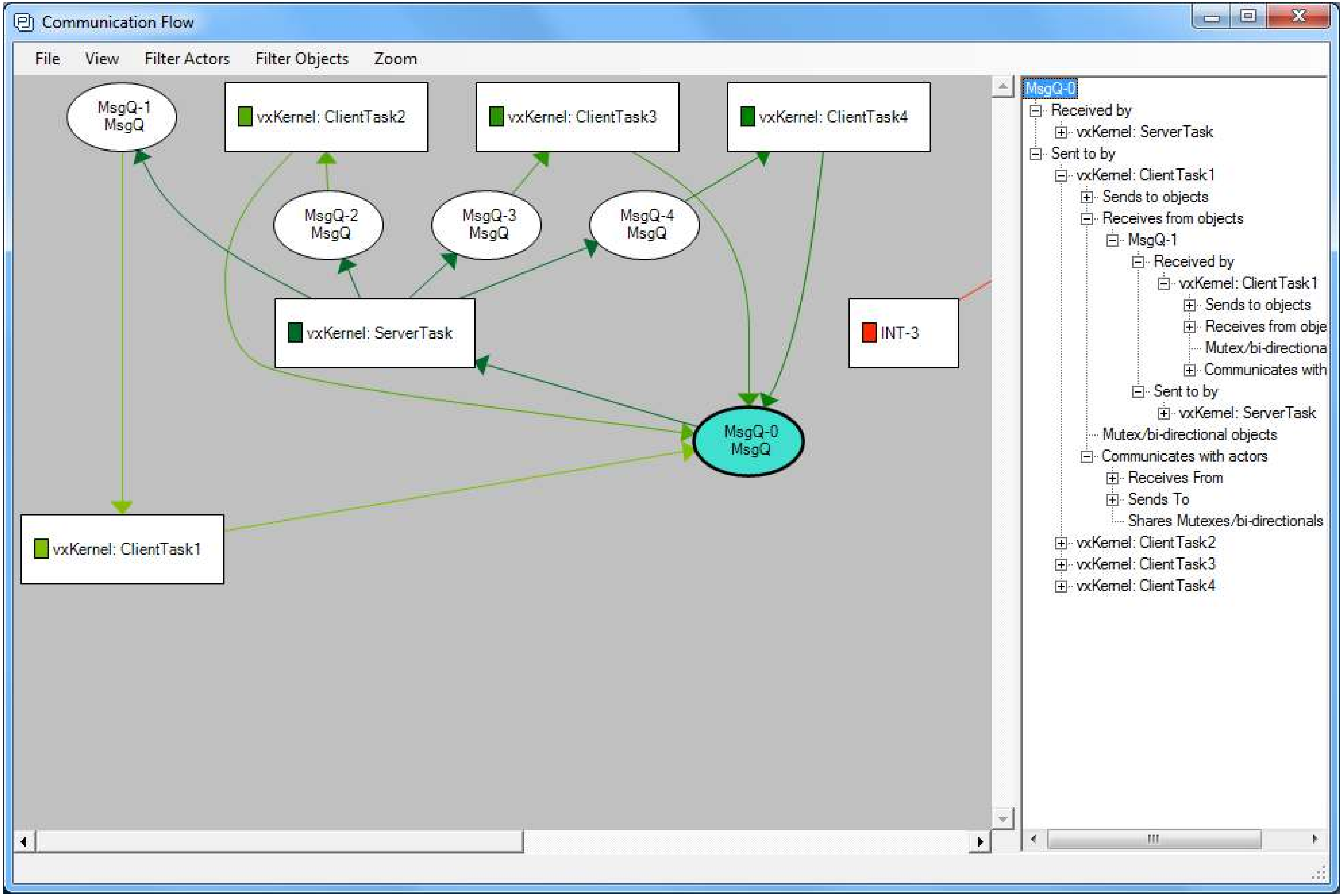
Timestamp	Actor	Event	Block time	Status	Size	Queue
47.242	vxKernel: ClientT	msgQSend		Sent post #1	1	1
47.269	vxKernel: ClientT	msgQSend		Sent post #2	2	1 2
47.297	vxKernel: ClientT	msgQSend		Sent post #3	3	1 2 3
47.325	vxKernel: ClientT	msgQSend		Sent post #4	4	1 2 3 4
47.325	vxKernel: Server	msgQReceive		Received post #1	3	1 2 3 4
47.453	vxKernel: Server	msgQReceive		Received post #2	2	2 3 4
47.601	vxKernel: Server	msgQReceive		Received post #3	1	3 4
47.740	vxKernel: Server	msgQReceive		Received post #4	0	4
47.892	vxKernel: Server	msgQReceive	2.731	Trying to receive...	0	Empty
50.622	vxKernel: ClientT	msgQSend		Sent post #5	1	5
50.622	vxKernel: Server	msgQReceive		Received post #5	0	5
50.759	vxKernel: Server	msgQReceive	867	Trying to receive...	0	Empty
51.625	vxKernel: ClientT	msgQSend		Sent post #6	1	6
51.626	vxKernel: Server	msgQReceive		Received post #6	0	6
51.766	vxKernel: Server	msgQReceive	2.856	Trying to receive...	0	Empty
54.622	vxKernel: ClientT	msgQSend		Sent post #7	1	7
54.622	vxKernel: Server	msgQReceive		Received post #7	0	7
54.752	vxKernel: Server	msgQReceive	919	Trying to receive...	0	Empty
55.670	vxKernel: ClientT	msgQSend		Sent post #8	1	8
55.670	vxKernel: Server	msgQReceive		Received post #8	0	8
55.803	vxKernel: Server	msgQReceive	824	Trying to receive...	0	Empty
56.626	vxKernel: ClientT	msgQSend		Sent post #9	1	9
56.627	vxKernel: Server	msgQReceive		Received post #9	0	9
56.747	vxKernel: Server	msgQReceive	1.752	Trying to receive...	0	Empty
58.499	vxKernel: ClientT	msgQSend		Sent post #10	1	10
58.499	vxKernel: Server	msgQReceive		Received post #10	0	10
58.696	vxKernel: Server	msgQReceive	735	Trying to receive...	0	Empty
59.430	vxKernel: ClientT	msgQSend		Sent post #11	1	11
59.431	vxKernel: Server	msgQReceive		Received post #11	0	11

Timestamp: 47.325
Actor: vxKernel: ClientTask4
Event: msgQSend
Status: Instant
Parameter: N/A

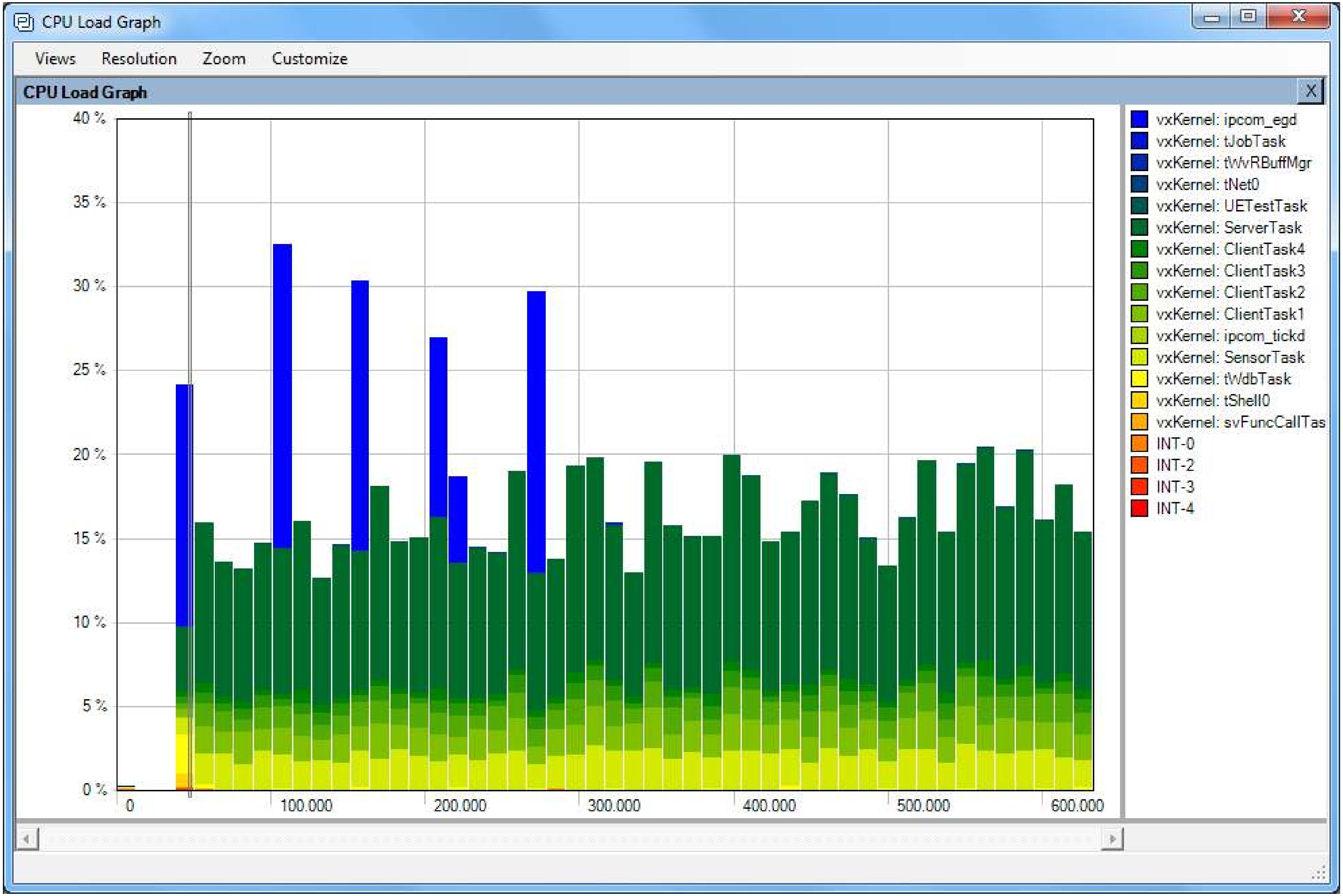
Goto Entry/Exit Event
Show in Trace
Goto Sending Event
Goto Receiving Event

Kernel Object History: shows all events on a specific kernel object

percepio SENSING SOFTWARE

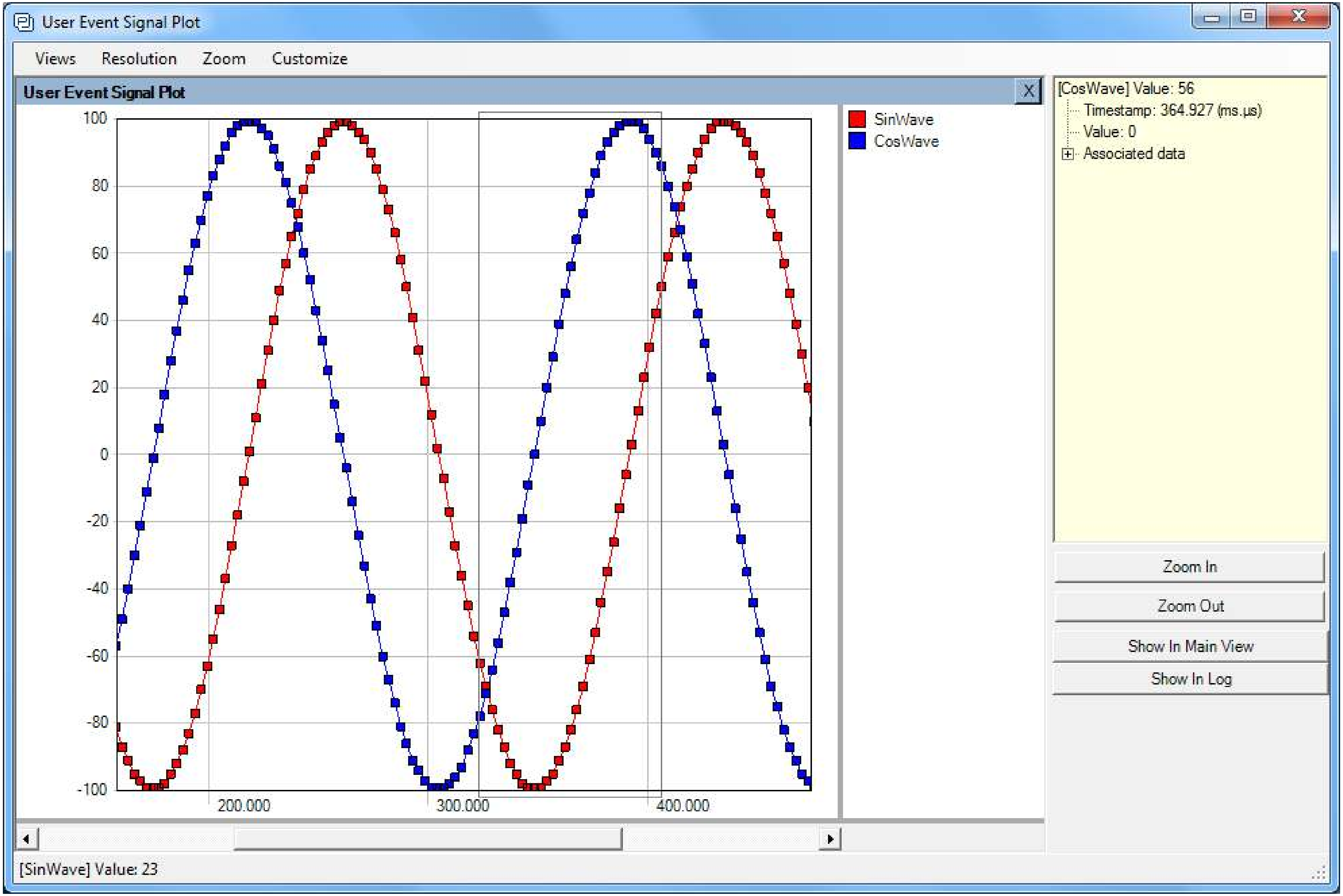


Communication Flow: dependencies through kernel objects



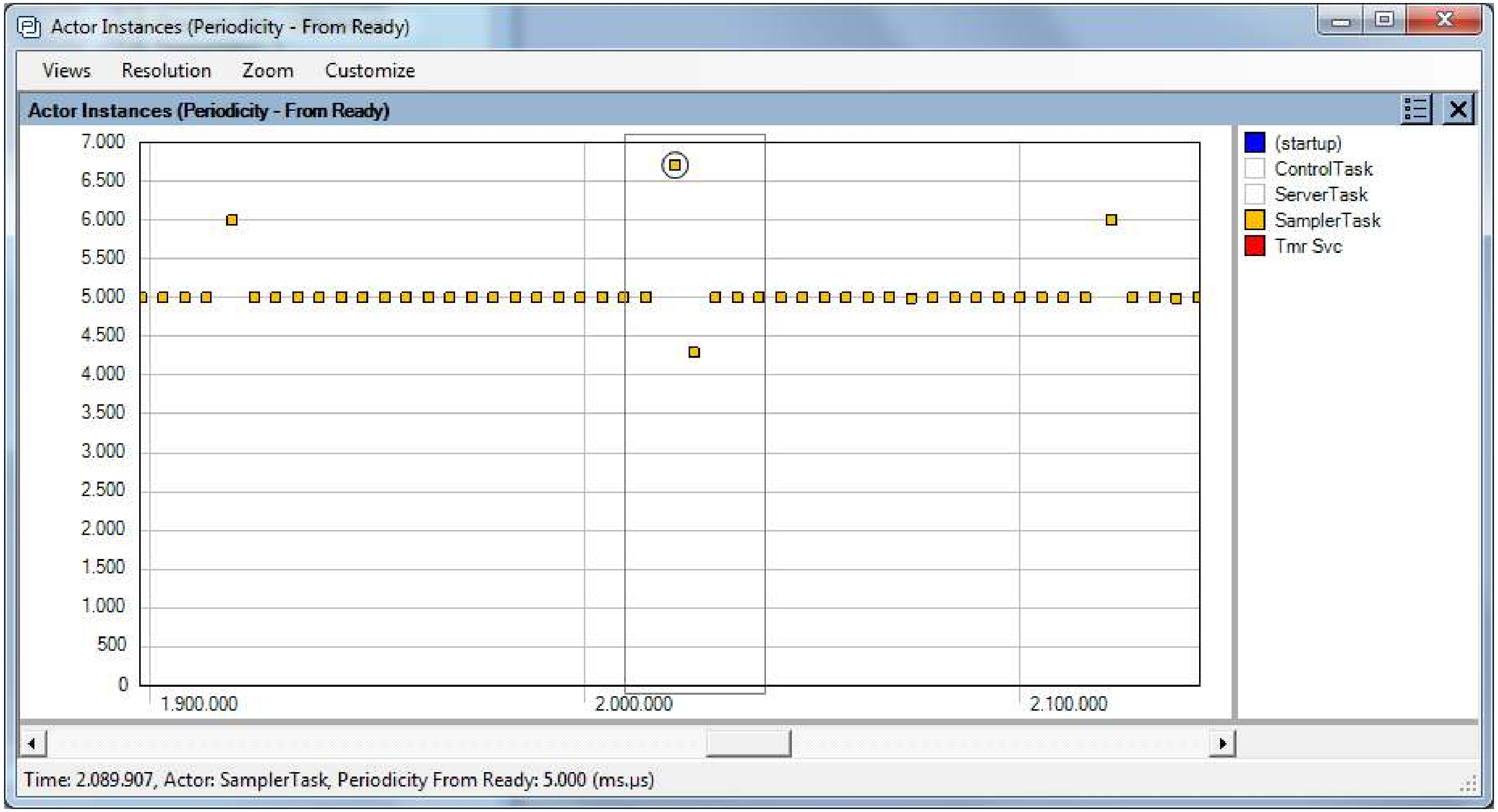
CPU Load Graph: Use of processor cycles, per task and ISR

percepio SENSING SOFTWARE



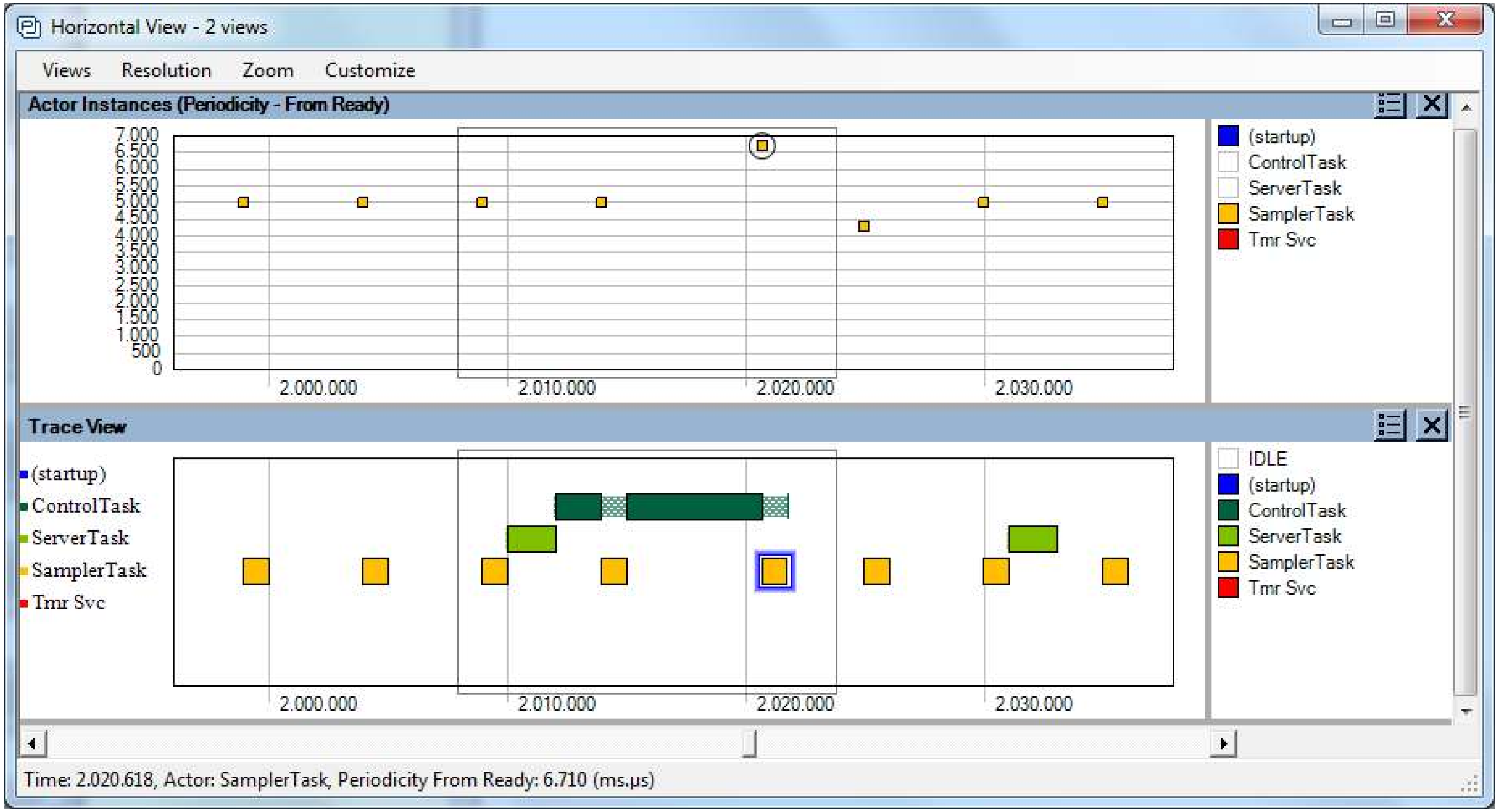
User Event Signal Plot: Based on "User Event" data

Example 1: Detecting and analyzing Task Jitter



Task should execute every 5 ms, but random variations of 1-2 ms!

Compare with the Task Trace...

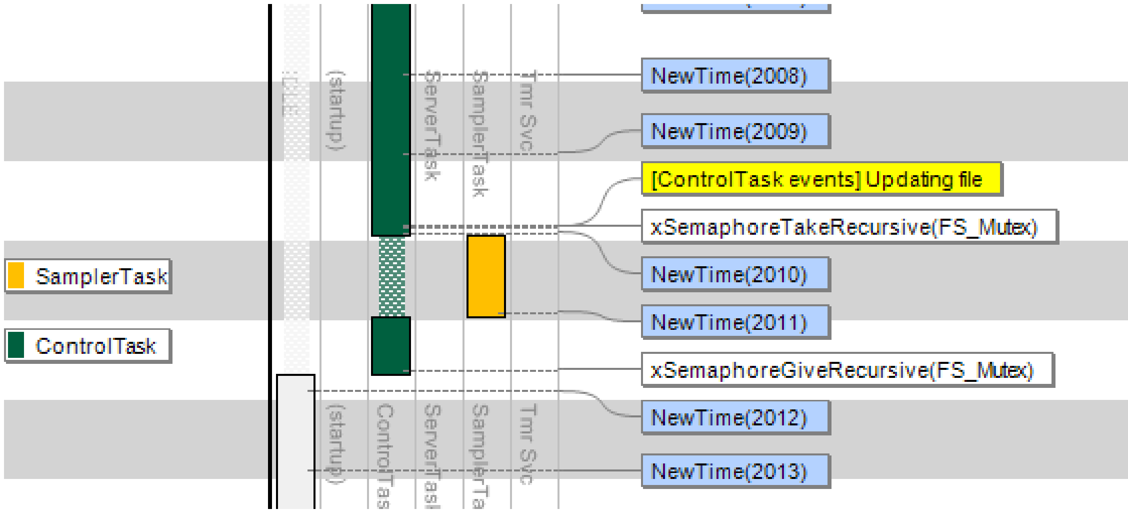


Something delays the activation of SamplerTask, probably ControlTask!

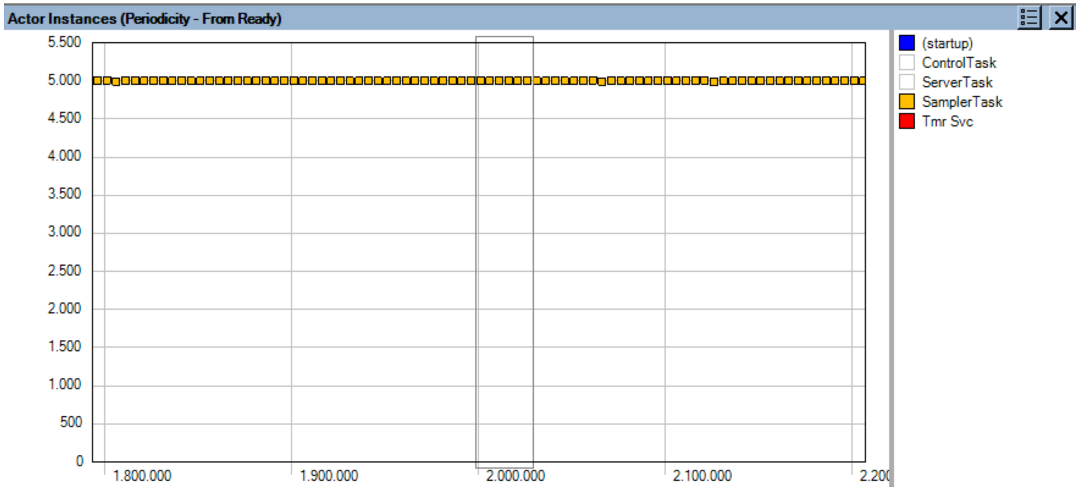
Why delayed activation?

ControlTask seems to disable interrupts!

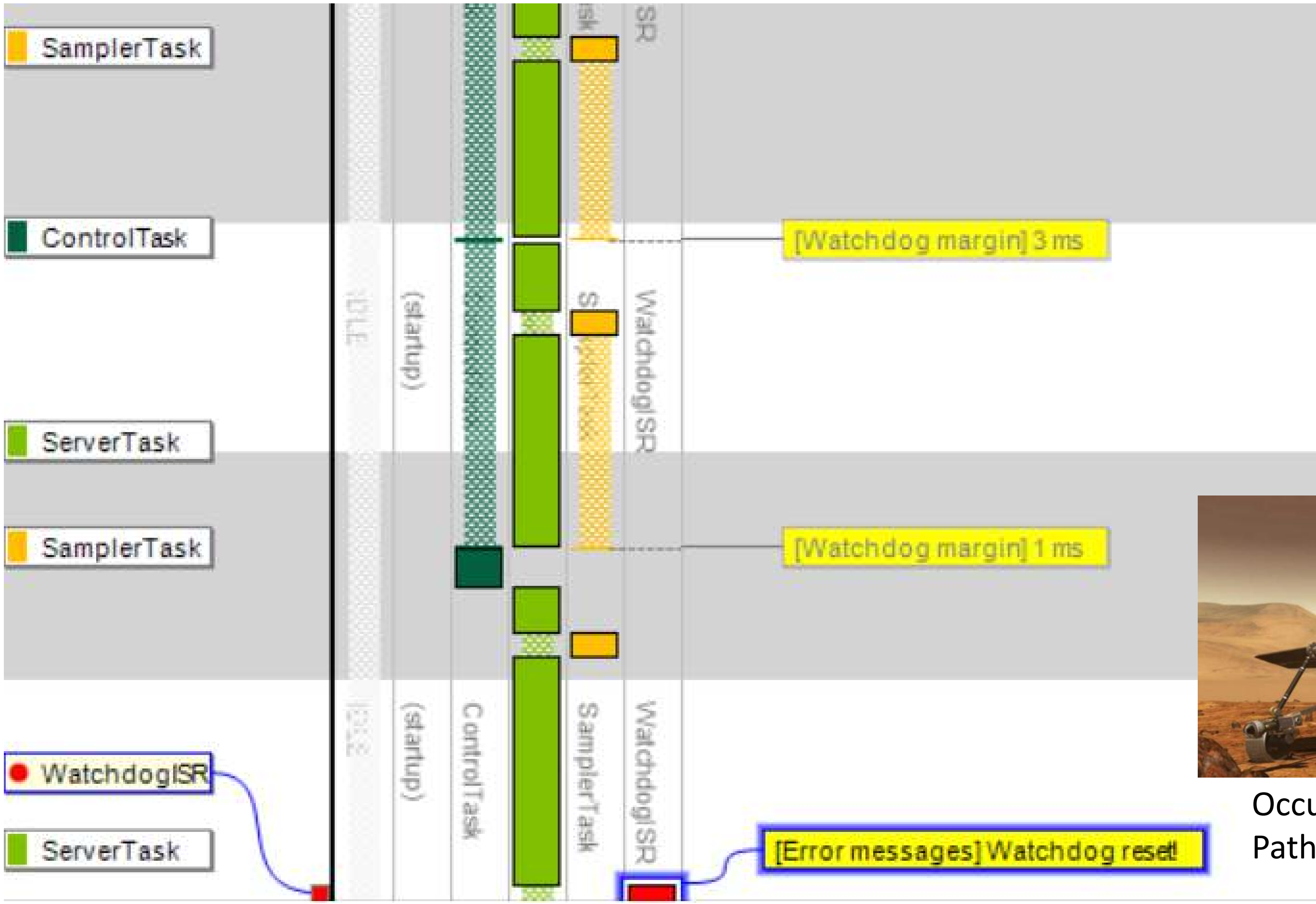
When using a Mutex instead of disabling interrupts...



Now perfect **5 ms periodicity!**

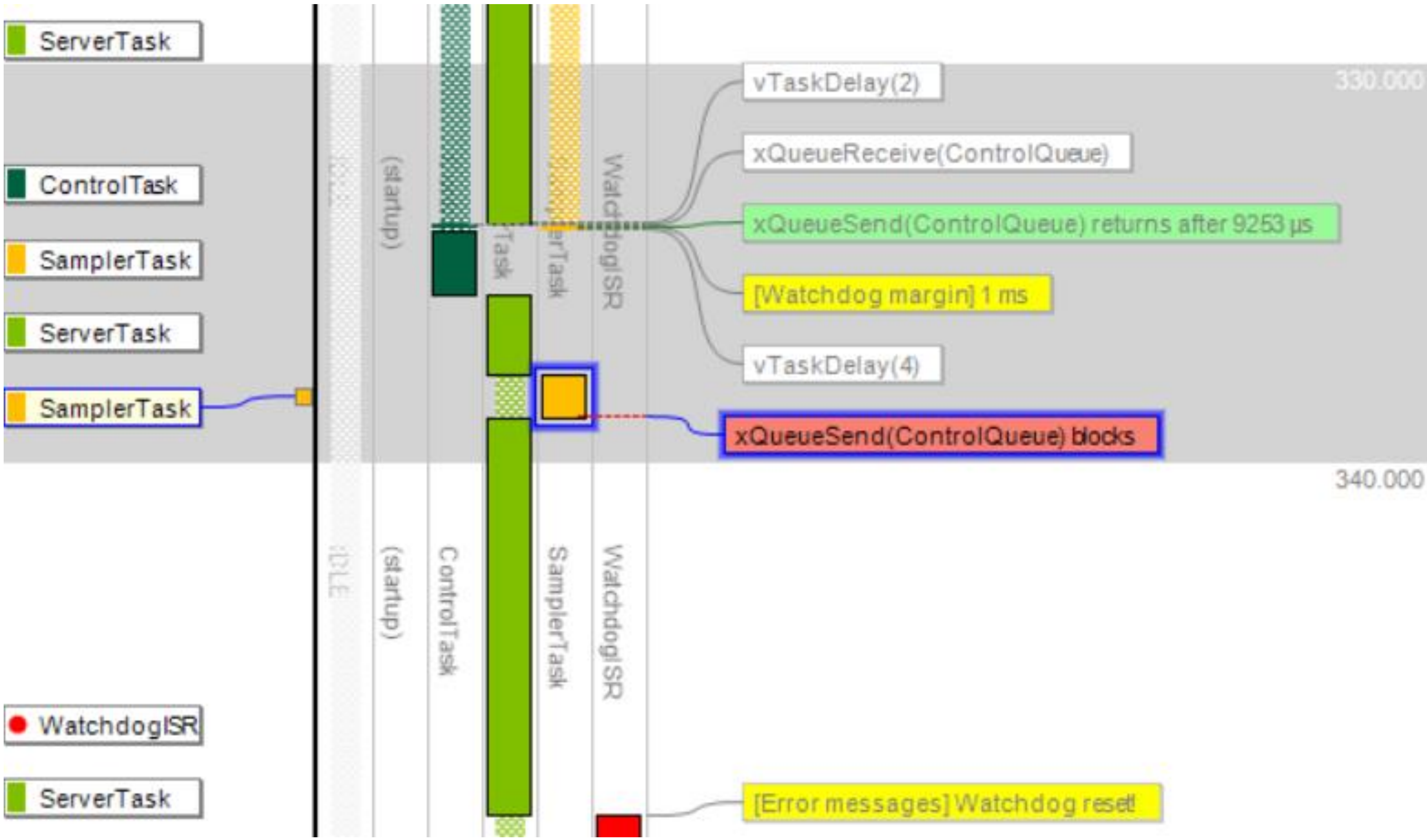


Example 2: Priority Inversion



Occurred in NASA's Pathfinder mission

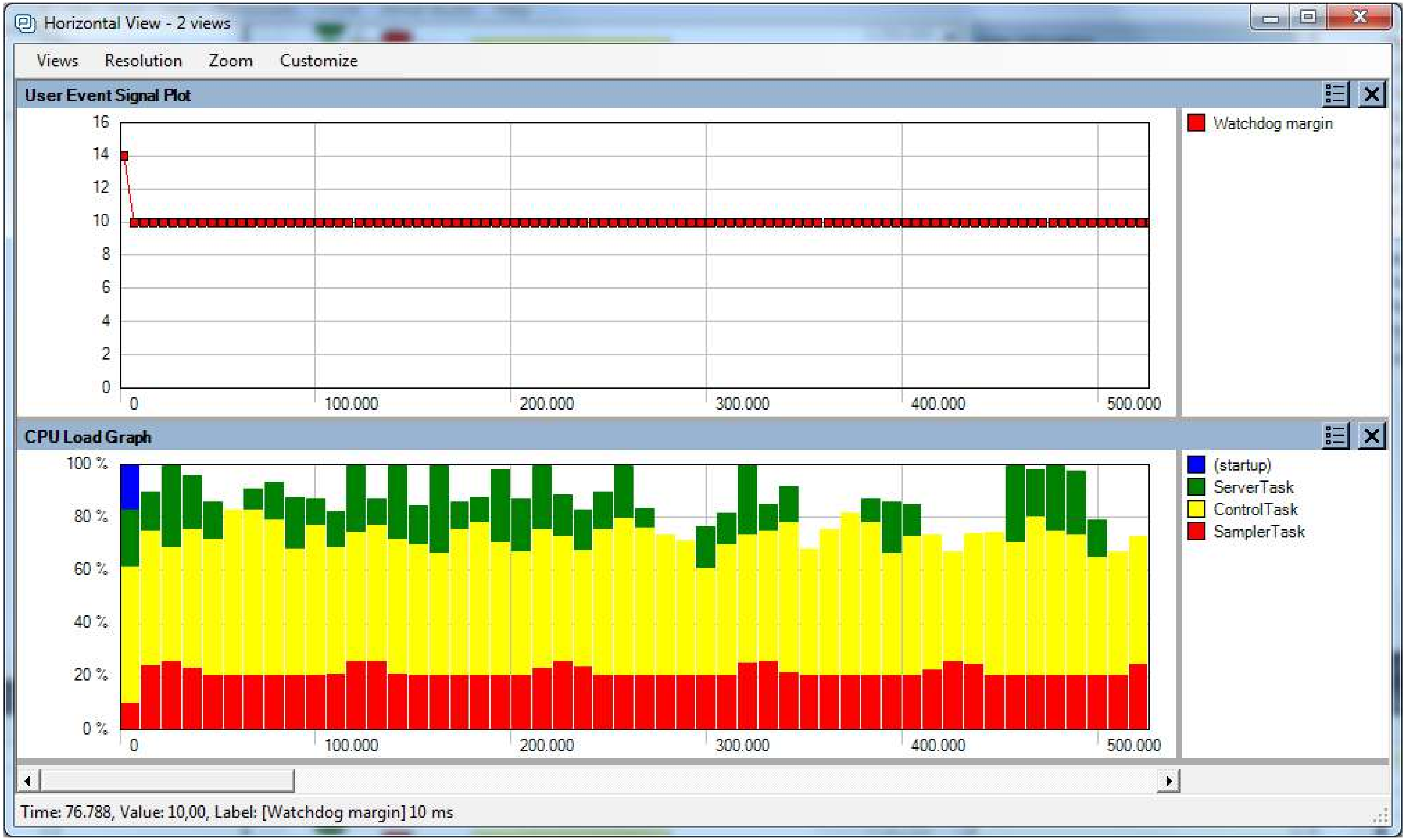
SamplerTask blocked, stops kicking the Watchdog



Blocked on Send means full queue... ControlTask is not reading the queue fast enough?

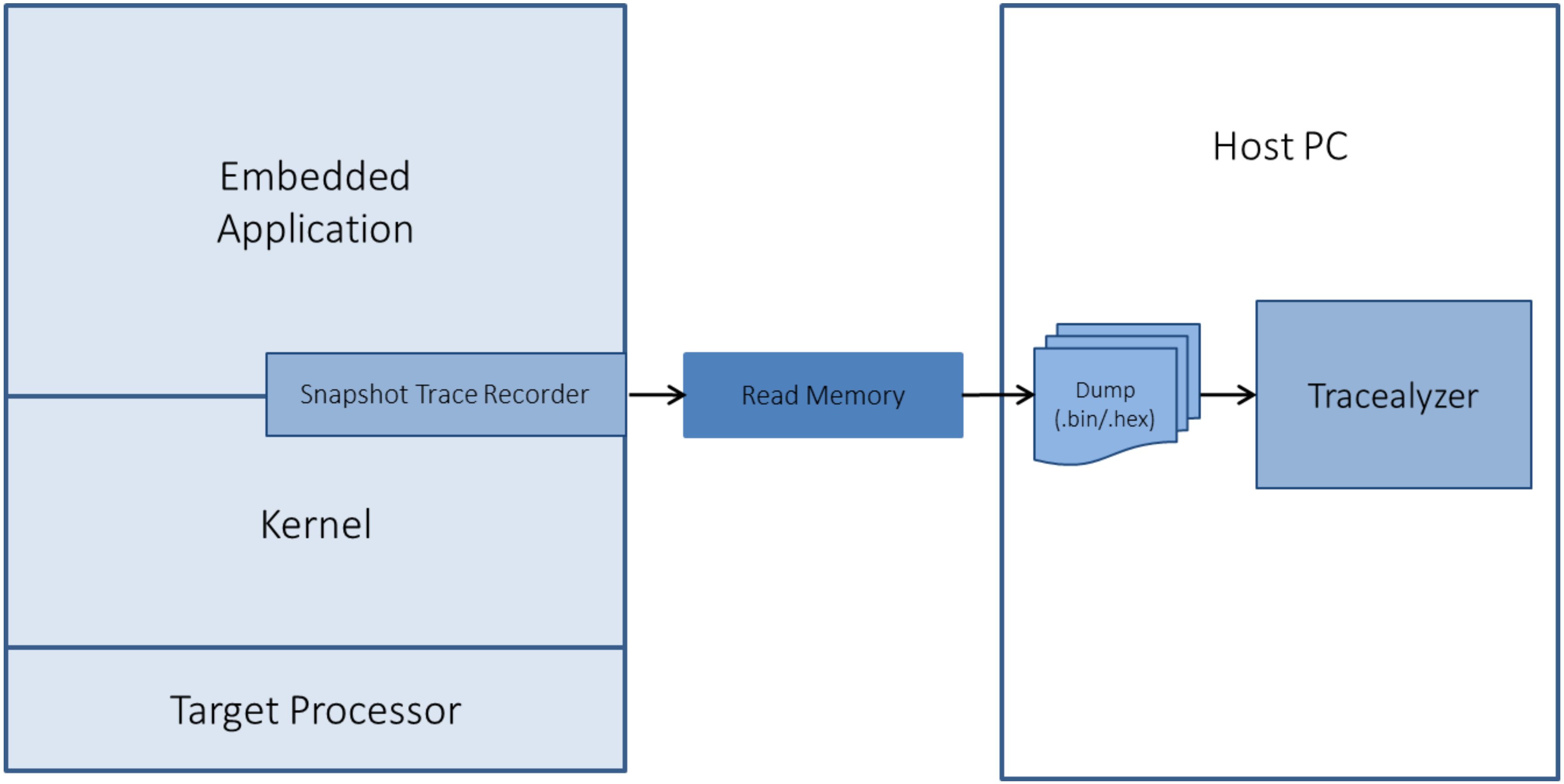
ServerTask has higher priority than ControlTask, does not leave enough cycles when high load!

With swapped task priorities – problem solved!



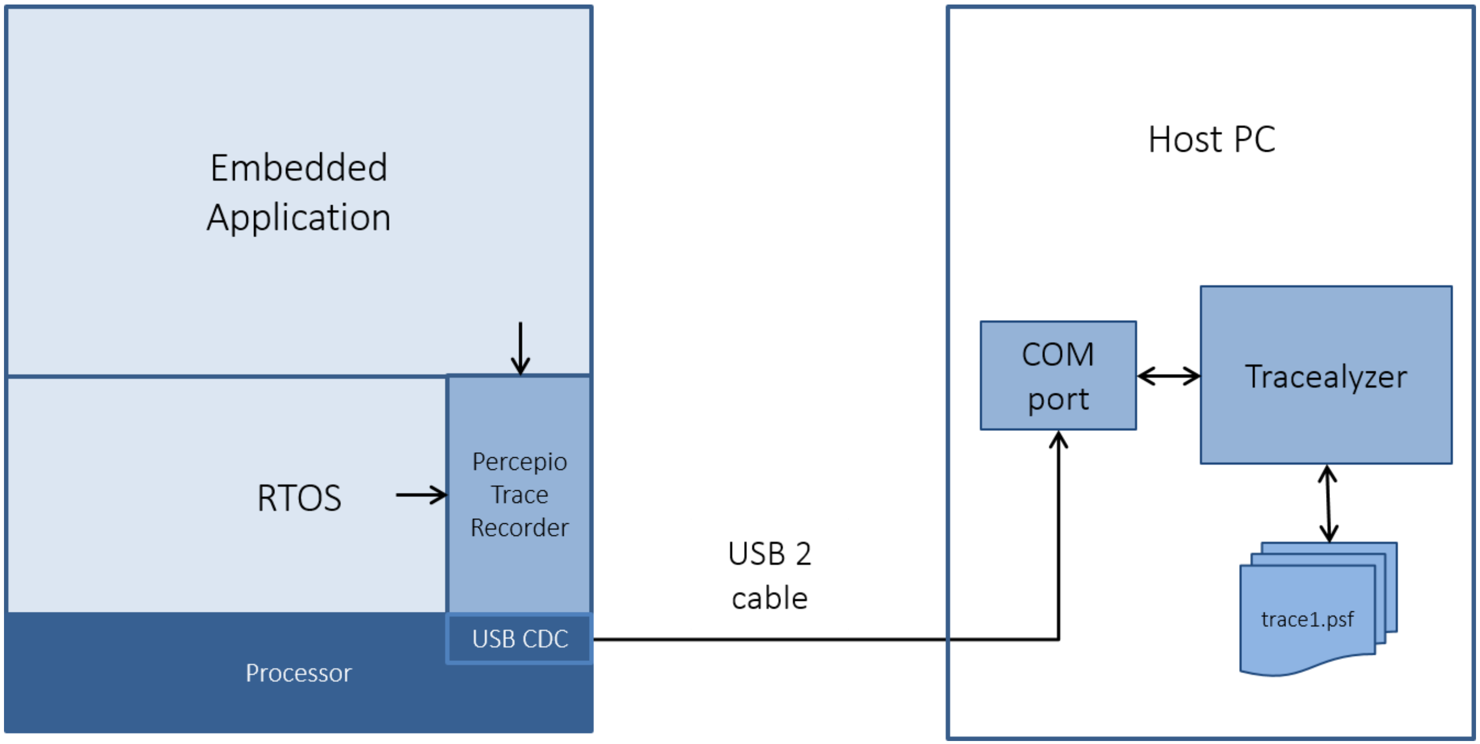
Getting the Data

Snapshot trace



Works for any processor and debugger, can be deployed in field.
Gives a short trace only, limited by RAM buffer size.

Streaming trace



Unlimited trace duration, small runtime footprint

Several interfaces can be used (USB, UART, TCP/IP, debug probe…)

Questions?