IoT
Online
Conference

www.iotonlineconference.com

Using Visual Trace Diagnostics to Uncover Hidden Issues

Johan Kraft

AGENDA

1. Introduction - Runtime Monitoring
2. Visual Trace Diagnostics, Examples
3. Hands-on Demo

THE SPEAKER

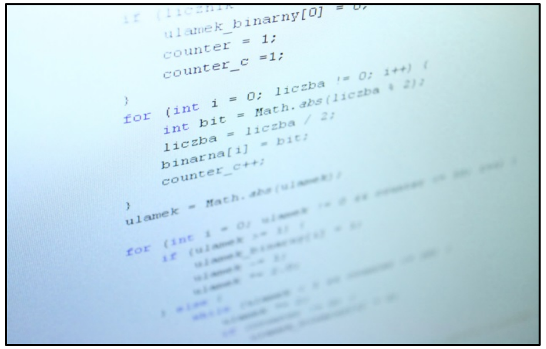
Dr. Johan Kraft

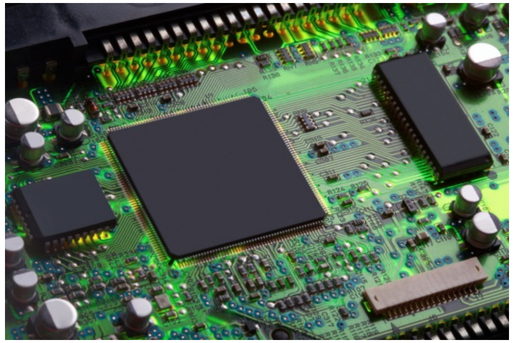
CEO, CTO and founder, Percepio AB

Focus: embedded software tracing and visualization for simplified development

Original developer of Percepio’s first product for visual trace diagnostics, Tracealyzer, and the founder of the company. Background in applied academic research in collaboration with industry, focused on embedded software timing analysis, and embedded software development at ABB Robotics. PhD in computer science.

SOURCE CODE IS NOT THE FULL PICTURE





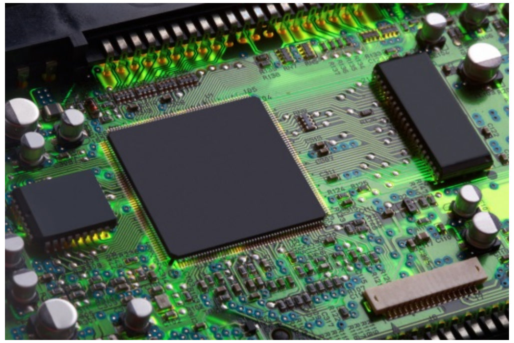
?

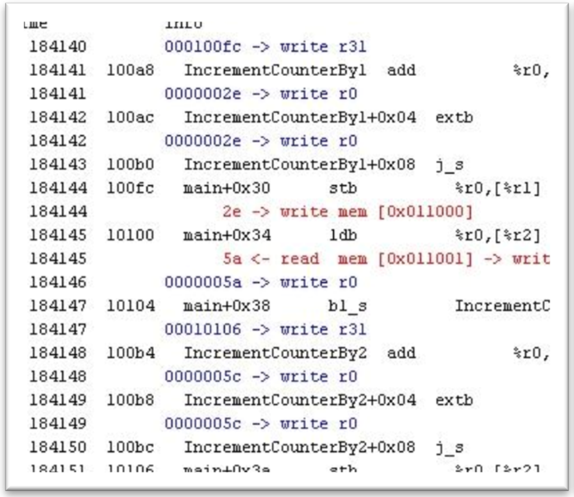
The runtime behavior also depend on dynamic effects

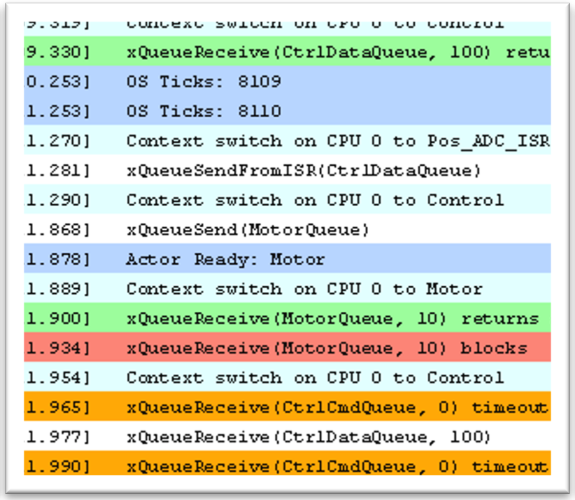
Such as variations in software timing and interference between tasks

Not visible in the source code, only in runtime!

RUNTIME MONITORING

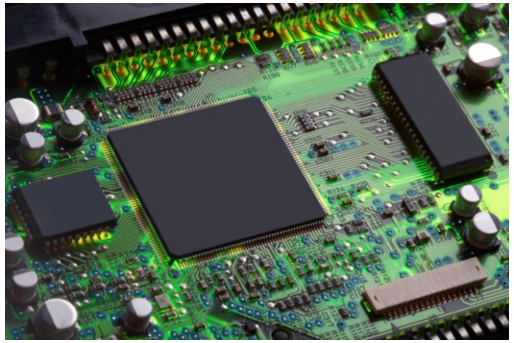


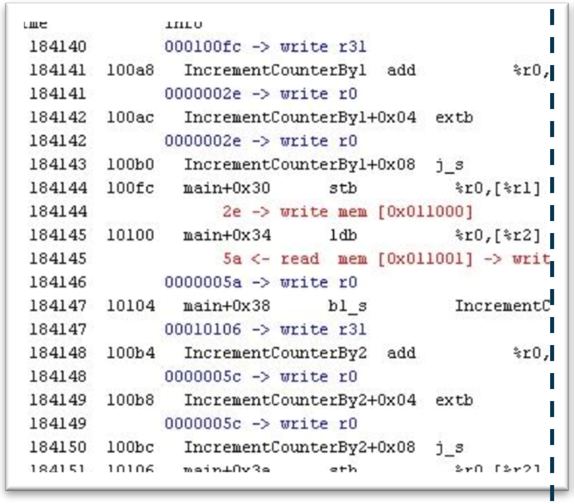


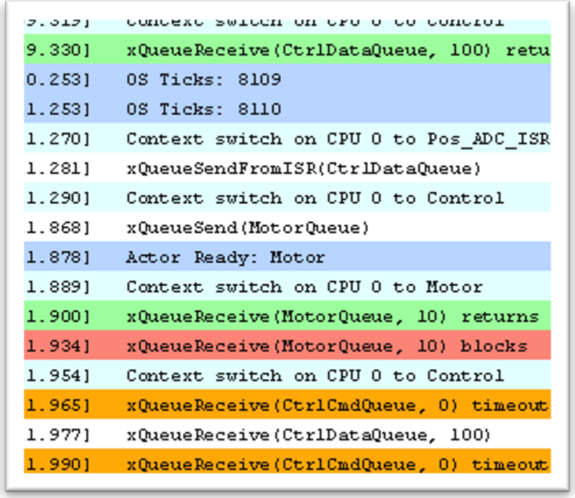


	Instruction Trace	Software Event Trace	Application Logging
Producer	Processor core	Software	Software
Abstraction Level	Low	Medium	High
Overhead	None	Minor	Depends on method
Special HW needed	Yes	No	No

RUNTIME MONITORING

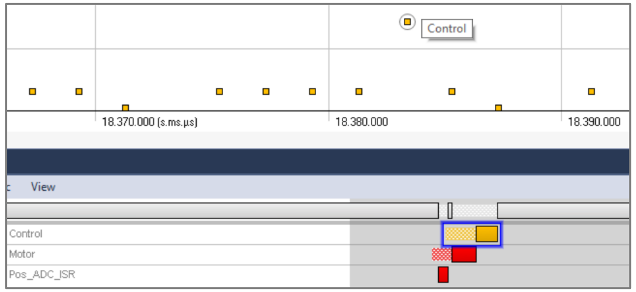




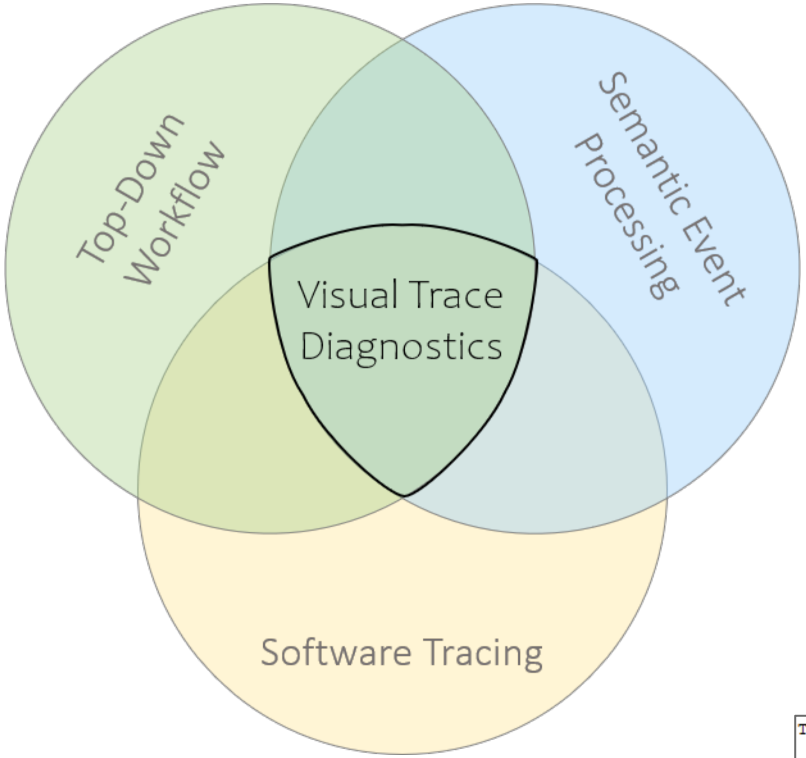


	Instruction Trace	Software Event Trace	Application Logging
Producer	Processor core	Software	Software
Abstraction Level	Low	Medium	High
Overhead	None	Minor	Depends on method
Special HW needed	Yes	No	No

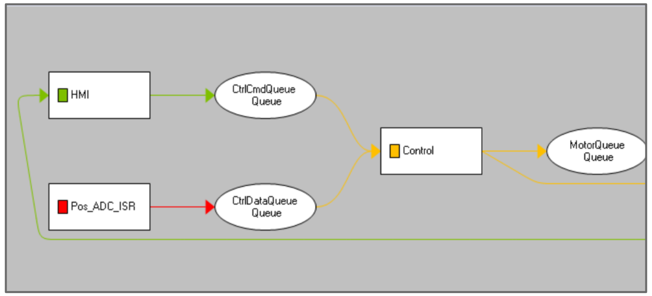
VISUAL TRACE DIAGNOSTICS



Visualization allowing for drill-down from overviews to details



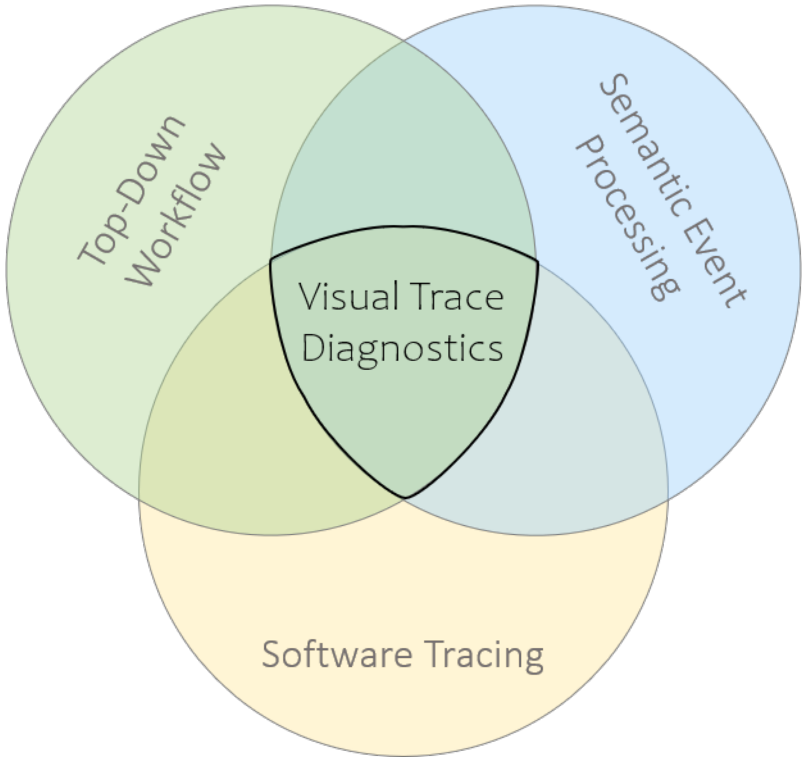
Data collection



Data processed into a meaningful model connecting related events and objects

Timestamp	Actor	Event Text
1.305	Control	xQueueReceive(CtrlDataQueue, 100) blocks
1.325	HMI	Context switch on CPU 0 to HMI
1.333	HMI	vTaskDelayUntil(500)
1.349	TzCtrl	Context switch on CPU 0 to TzCtrl
1.426	TzCtrl	Unused Stack for TzCtrl: 81
1.433	TzCtrl	vTaskDelay(20)
1.448	IDLE	Context switch on CPU 0 to IDLE

VISUAL TRACE DIAGNOSTICS



Use cases:

- Debugging at System Level
- Finding Software Design Flaws
- Verifying Timing and Performance

USE CASE 1: DEBUGGING AT SYSTEM LEVEL

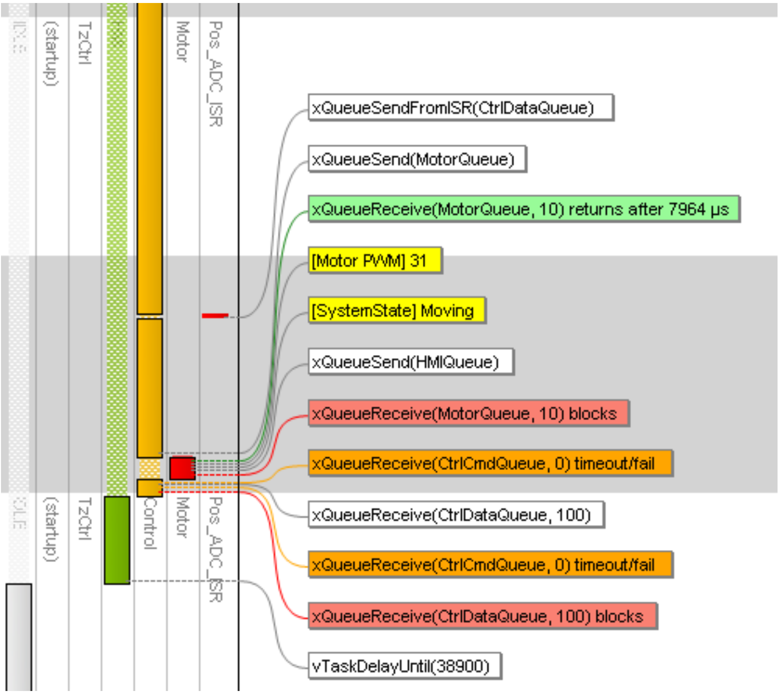
When you need a timeline of your software at runtime

Especially important when

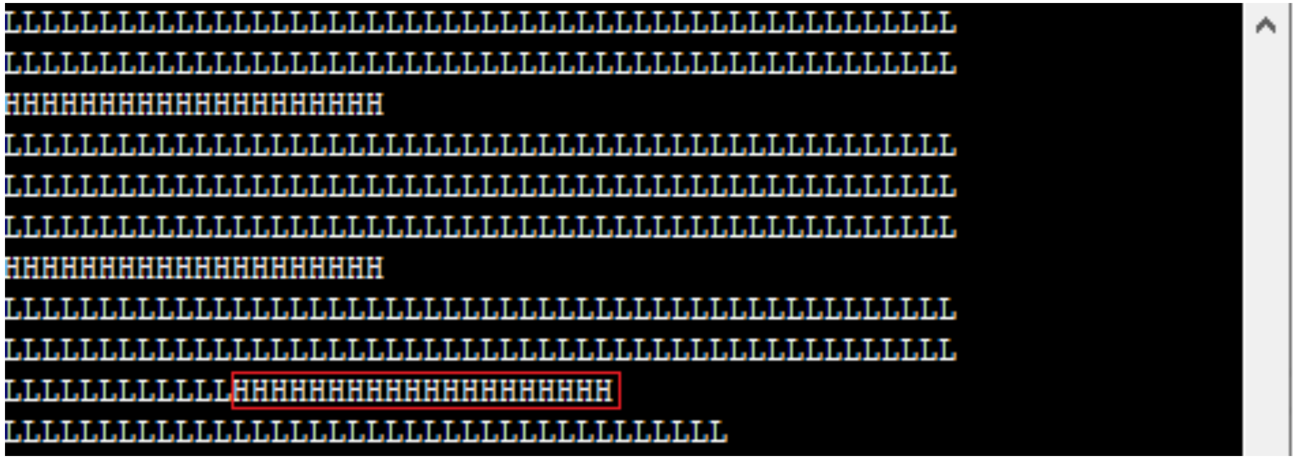
- The location of the bug is not obvious
- The issue is difficult to reproduce
- Not possible to use interactive (halting) debug

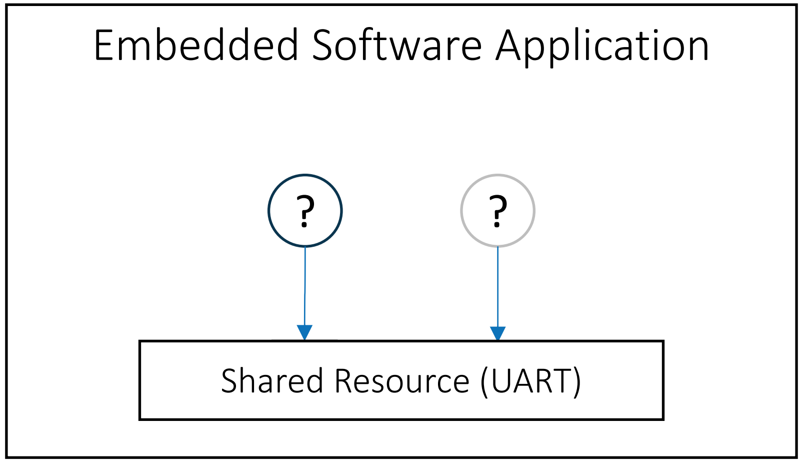
Superior to classic “printf debugging”

- Very low intrusiveness, typically 100x faster than printf logging
- More information, e.g., kernel, resource usage, …
- Easy and powerful visual analysis, that scales to large data sets



EXAMPLE 1: SYSTEM-LEVEL DEBUGGING





The system is writing data to a shared resource, in this case a UART serial port.

Error detected in full-system testing: Occasional data corruption

Two types of data written, “LLL…” and “HHH…”

Sometimes they are mixed up and corrupt the data stream

EXAMPLE 1: RECORD A TRACE

RTOS tasks and API calls are recorded automatically. Extra logging added in the UART driver.

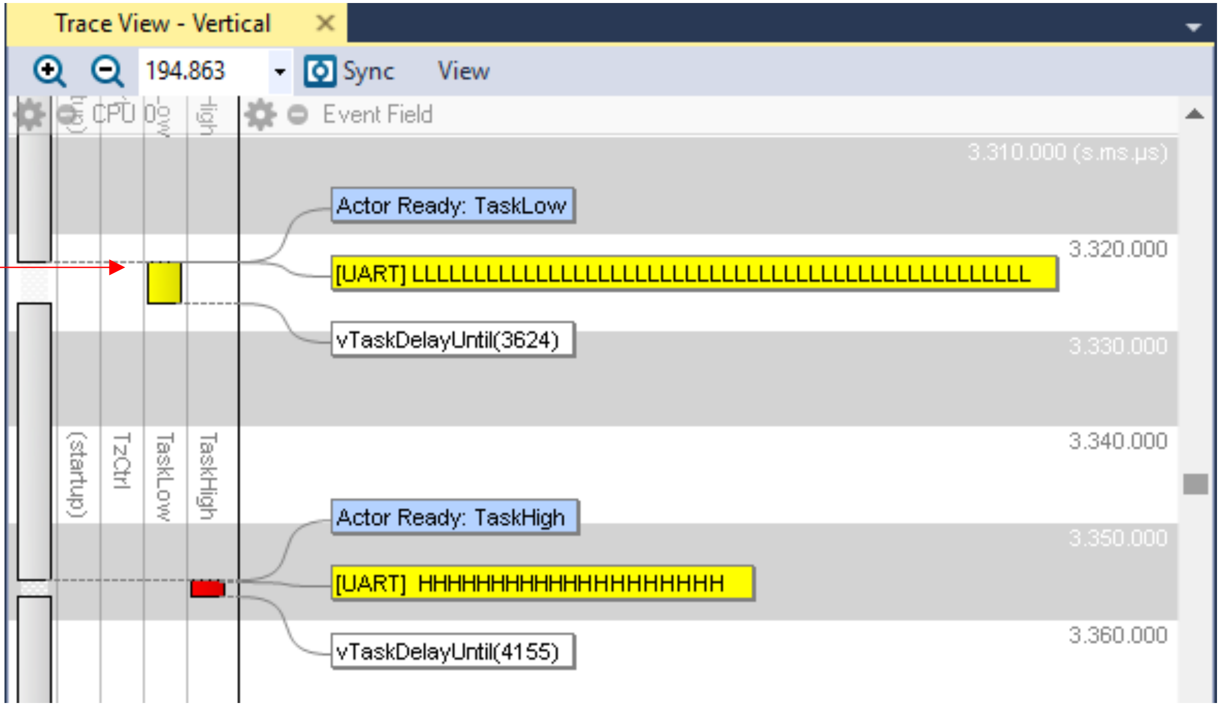
```c
size_t __write(int handle, const unsigned char *buffer,
{
    if (buffer == NULL)
    {
        return 0;
    }

    vTracePrint(UART_EVENTS, buffer);

    for (; size != 0; --size)
    {
        if (ptr_put(stdio_base, *buffer++) < 0)
```

We see that two different tasks are writing to the UART.
Are they interfering? How do we find the error?

```
LLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLL
LLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLL
LLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLL
HHHHHHHHHHHHHHHHHHHH
LLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLL
LLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLL
LLLLLLLLLLLLHHHHHHHHHHHHHHHHHHHH
LLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLL
```

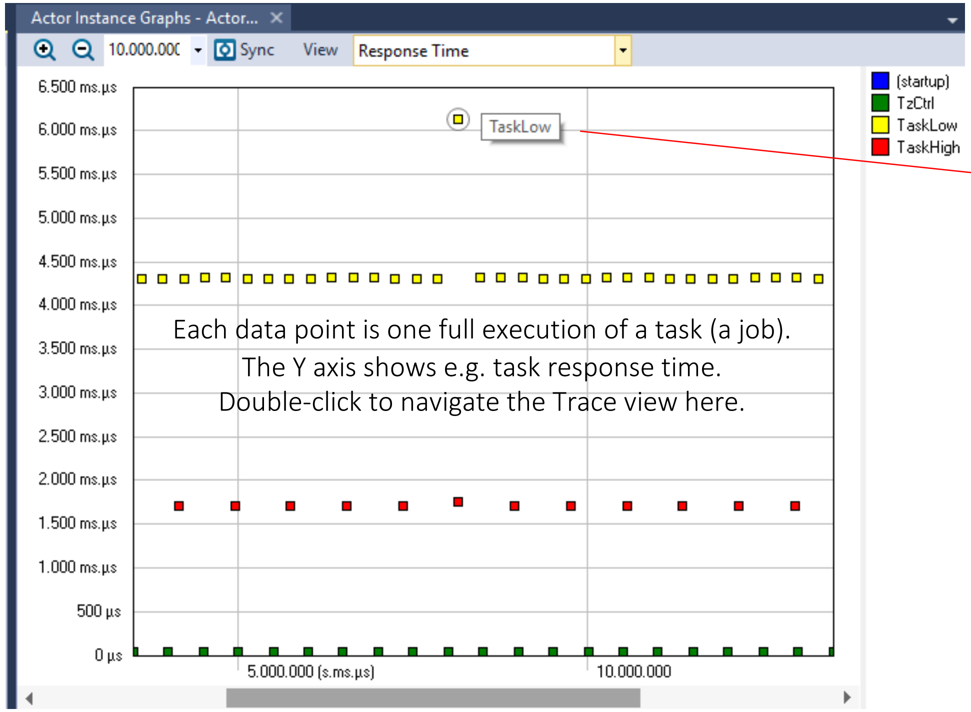


(Vertical timeline, with RTOS tasks on the left)

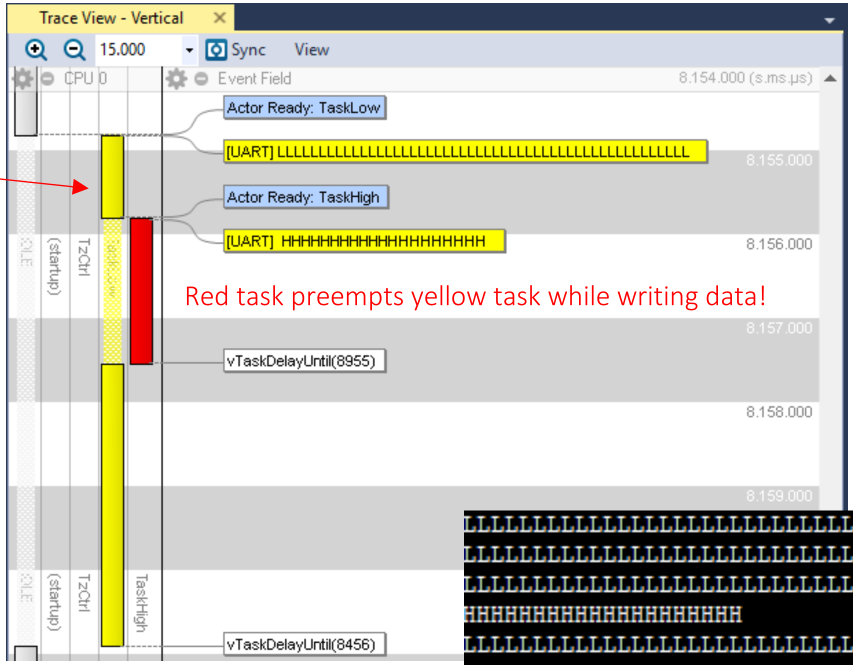
EXAMPLE 1: USE TOP-DOWN ANALYSIS

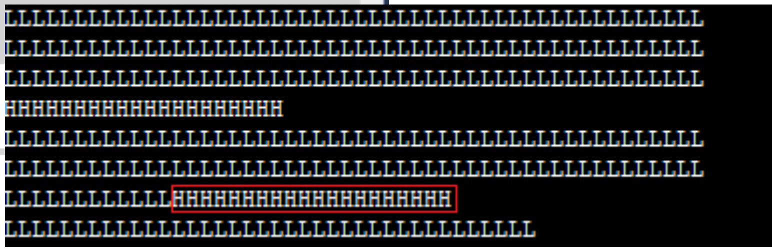
Spot issues in visual overviews, drill down to see the details…

Visual overview of task response times (10 s)



Trace view at selected location (15 ms)

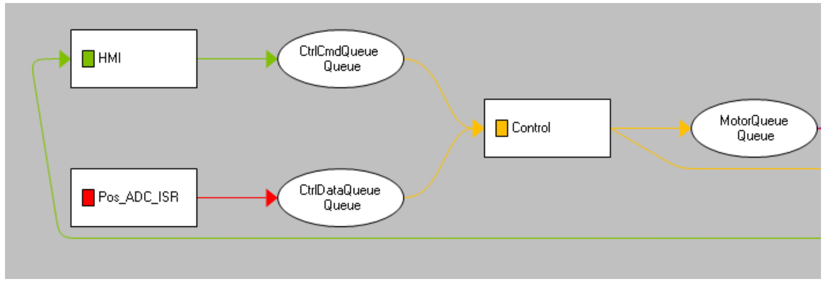




Conclusion: Access conflict due to undesired task preemption, caused by missing synchronization between tasks.

Solution: Add mutex calls for mutual exclusion, or move the writes to a single task.

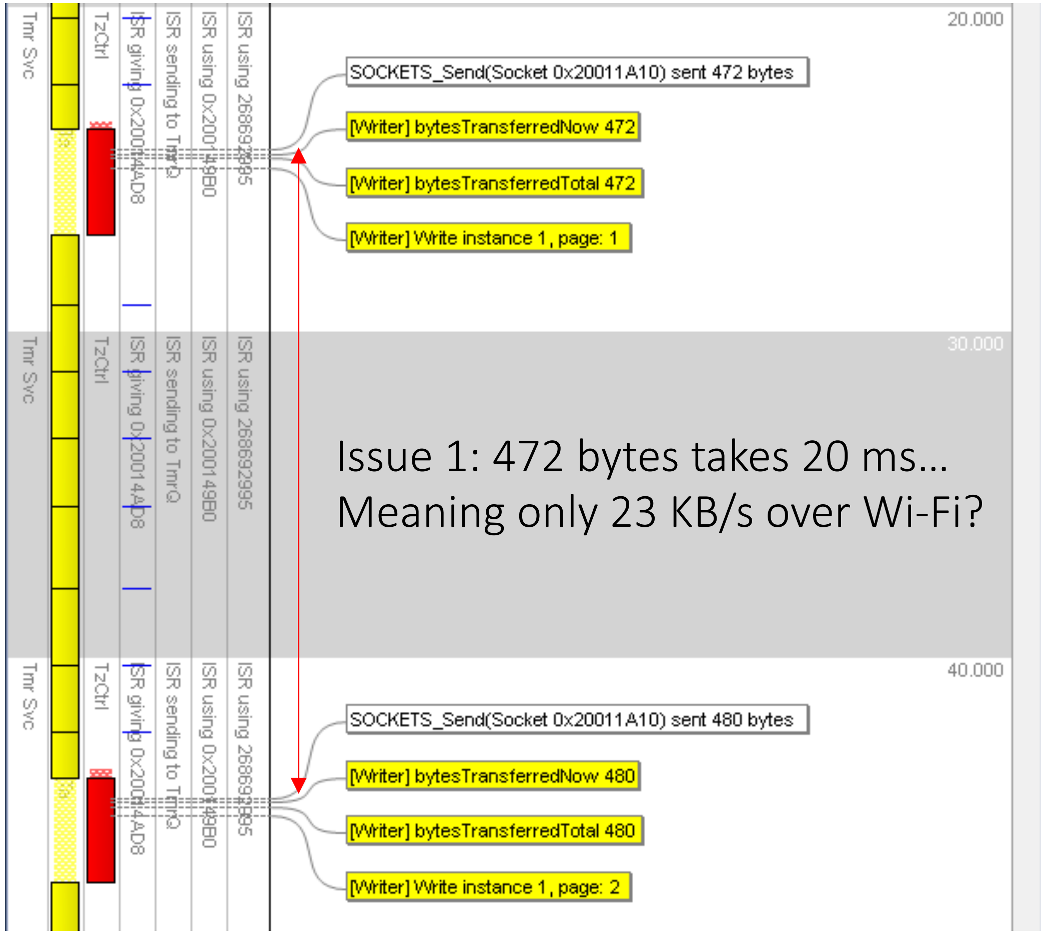
USE CASE 2: FINDING SOFTWARE DESIGN FLAWS

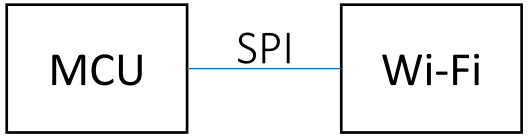


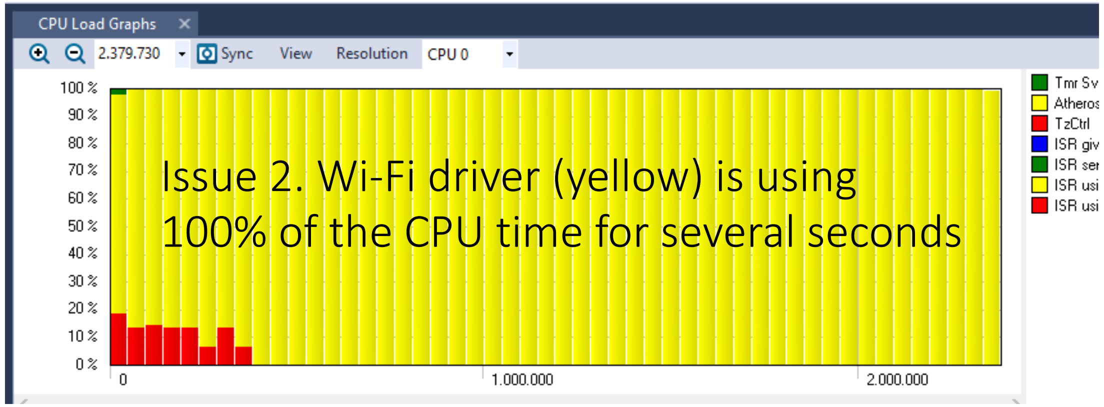
- RTOS software design can be quite challenging
  - Important to follow Best Practices
- Software design flaws – Deviations from Best Practices
  - Reducing performance and responsiveness (e.g. busy waiting)
  - System testing less effective, higher risk of missed bugs
- Examples:
  - Unsuitable task priorities
  - Large timing variations, perhaps even in high priority tasks
  - Lots of task dependencies (e.g. mutex synchronization)

EXAMPLE 2: SOFTWARE DESIGN FLAWS

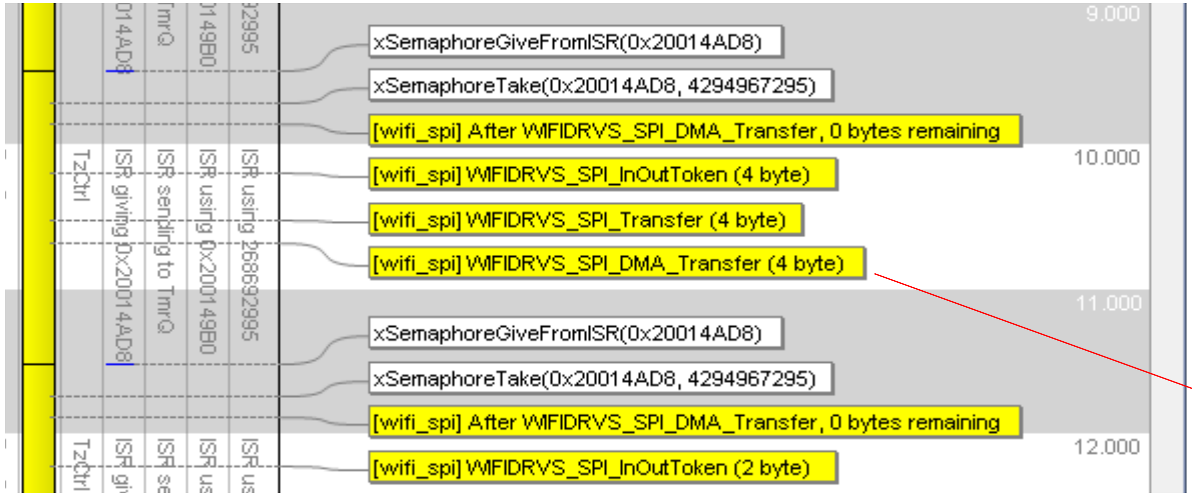
When evaluating an IoT demo from an MCU vendor, **two issues** where noticed immediately in the views...







EXAMPLE 2.1: SLOW WI-FI



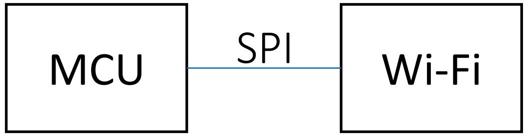
We added logging in the Wi-Fi driver…

```
static A_STATUS WIFIDRVS_SPI_DMA_Transfer(spi_transfer_t *transfer)
{
    assert(NULL != transfer);

    vTracePrintF(uec, "WIFIDRVS_SPI_DMA_Transfer (%d byte)", transfer->dataSize);
```

DMA is used for transferring 4 bytes?

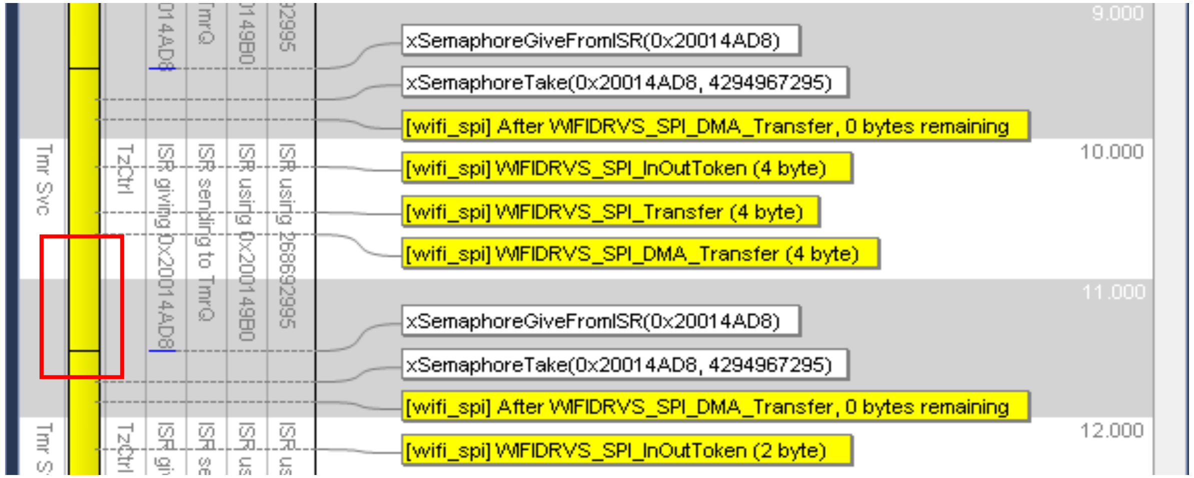
Conclusion 1: DMA is used for *every* SPI message, even for small handshaking messages of 2-4 bytes. Very inefficient, as the DMA transfers take 1-2 ms to initiate…



Solution: Don’t using DMA for small amounts of data…

EXAMPLE 2.2: HIGH CPU LOAD

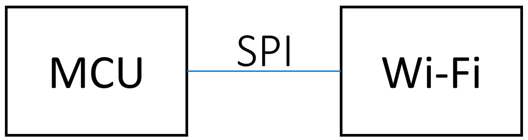
The trace view shows that WiFi driver task doesn't suspend while waiting for the SPI transfer to finish.



Conclusion 2: Most likely a "busy waiting" loop here. Bad practice when using an RTOS, since preventing other tasks from executing. Probably unintentional.

A likely culprit was quickly found in the driver code:

```
#if !NON_BLOCKING_TX
        /*Wait till packet is sent to target*/
        if ((block_result = BLOCK(pCxt, ath_sock_context[index], TRANSMIT_BLOCK_TIMEOUT, TX_DIRECTION)) != A_OK)
        {
            result = block_result;
```



Solution: Ensure the RTOS task is suspended while waiting, e.g. by enabling the NON_BLOCKING_TX option.

USE CASE 3: VERIFYING TIMING AND PERFORMANCE

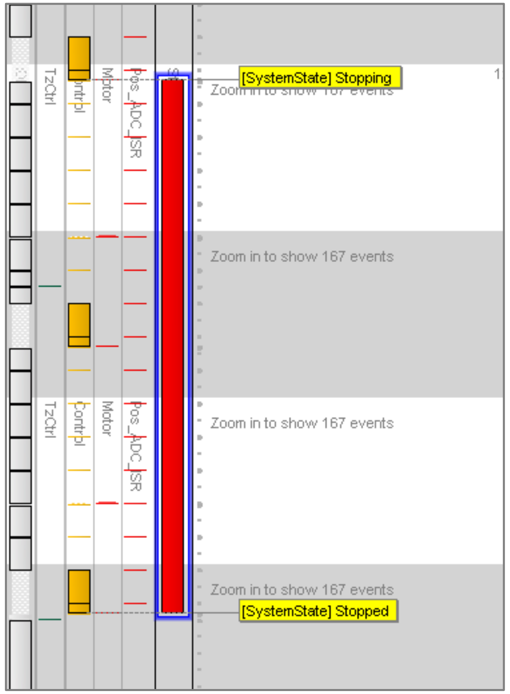
Keep Track of Software Timing and Resource Usage Metrics

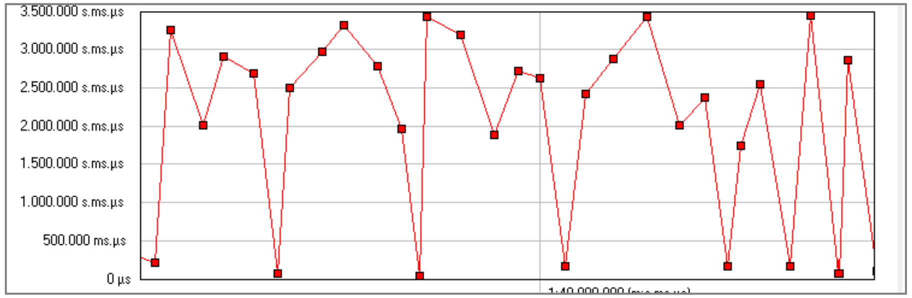
Software Timing

- Per Task – Execution time, response time, periodicity, …
- Custom Intervals – Time between user-defined events

Resource Usage

- CPU usage (percentage per task)
- Memory usage (malloc/free, stack usage per task)
- I/O usage (e.g. bytes per second)





HANDS-ON DEMO

OVERHEAD?

For Percepio TraceRecorder library, used for e.g. FreeRTOS:

RAM usage: Typically 5-10 KB for trace buffer (configurable)

Flash usage: 10-20 KB needed for trace recorder library

CPU usage:

- Microseconds per event
- Depends on application (event rate, amount of logging)
- Some penalty, but typically not noticeable (a few percent)

SUMMARY

Visual Trace Diagnostics - Top-Down Workflow from Overviews to Details

- System Level Debugging
- Finding Software Design Flaws
- Verifying Timing and Performance

Tracing performed by a software library

- Kernel activity (scheduling, API calls)
- Add user events for additional information
- No special hardware needed

About 100x faster than printf logging, so the overhead is typically not a problem

THANK YOU

IoT
Online
Conference

www.iotonlineconference.com

IoT
Online
Conference

www.iotonlineconference.com