IoT Online Conference

O percepio®

www.iotonlineconference.com

Using Visual Trace Diagnostics to Uncover Hidden Issues

Johan Kraft





Introduction - Runtime Monitoring



Visual Trace Diagnostics, Examples





THE SPEAKER

Dr. Johan Kraft

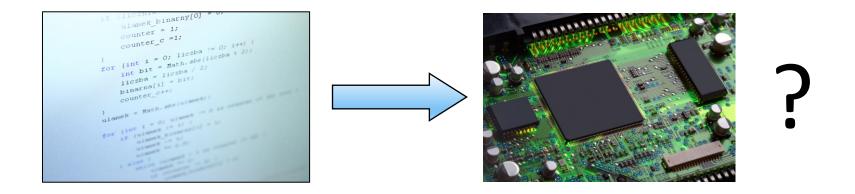


\bigcirc CEO, CTO and founder, Percepio AB

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

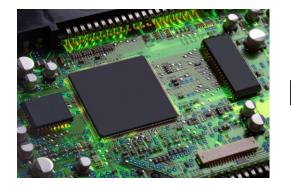


The runtime behavior also depend on dynamic effects

Such as variations in software timing and interference between tasks

Not visible in the source code, only in runtime!

RUNTIME MONITORING



rme TULO	5.515] CONCEXC SWICEN ON CRO D CO CONCIDI	<pre>ivc] Starting key provisioning</pre>
184140 000100fc -> write r31	9.330] xQueueReceive(CtrlDataQueue, 100) retu	vc] Write root certificate
184141 100a8 IncrementCounterByl add %r0,	0.253] OS Ticks: 8109	<pre>wc] Write device private key</pre>
184141 0000002e -> write r0 184142 100ac IncrementCounterBy1+0x04 extb	1.253] OS Ticks: 8110	Svc] Write device certificate
184142 0000002e -> write r0	1.270] Context switch on CPU 0 to Pos_ADC_ISR	Svc] Key provisioning done
184143 100b0 IncrementCounterBy1+0x08 j_s	1.281] xQueueSendFromISR(CtrlDataQueue)	Svc] Starting WiFi
184144 100fc main+0x30 stb %r0,[%r1]	1.290] Context switch on CPU 0 to Control	r Svc] WiFi module initialized.
184144 2e -> write mem [0x011000] 184145 10100 main+0x34 1db %r0,[%r2]	1.868] xQueueSend(MotorQueue)	IS-MAIN] WiFi connected to AP AndroidAP.
184145 5a <- read mem [0x011001] -> writ	1.878] Actor Ready: Motor	WS-MAIN] Attempt to Get IP.
184146 0000005a -> write r0	1.889] Context switch on CPU 0 to Motor	WS-MAIN] IP Address acquired 192.168.0.51
184147 10104 main+0x38 bl_s IncrementC	1.900] xQueueReceive(MotorQueue, 10) returns	WS-LED] [Shadow 0] MQTT: Creation of dedicated MQ1 WS-LED] Sending command to MQTT task.
184147 00010106 -> write r31 184148 100b4 IncrementCounterBy2 add %r0,	1.934) xQueueReceive(MotorQueue, 10) blocks	QTT] Received message 10000 from queue.
184148 0000005c -> write r0	1.954] Context switch on CPU 0 to Control	QTT] Looked up a7sw0r7rvpirn.iot.us-east-1.amazona
184149 100b8 IncrementCounterBy2+0x04 extb	1.965) xQueueReceive(CtrlCmdQueue, 0) timeout	MQTT] MQTT Connect was accepted. Connection establ
184149 0000005c -> write r0	1.977] xQueueReceive(CtrlDataQueue, 100)	MQTT] Notifying task.
184150 100bc IncrementCounterBy2+0x08 j_s 184151 10106 main±0v3a eth 2r0 [2r2]	1.990) xQueueReceive(CtrlCmdQueue, 0) timeout	AWS-LED] Command sent to MQTT task passed.

	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



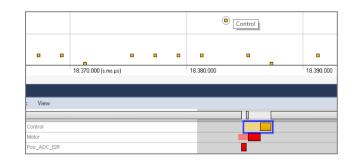
		18
		18
		18
		18
		18
		18
		> 18
	\neg	18
3		18
		18
		18
		18
		18

I			TULO		ше
		write r31	000100fc ->		184140
%r0,	add	CounterByl	Increment	100a8	184141
10		write r0	0000002e ->		184141
extb	+0x04	CounterBy14	Increment	100ac	184142
		write r0	0000002e ->		184142
j_s	+0x08	CounterBy1+	Increment	100b0	184143
%r0,[%r1]		stb	main+0x30	100fc	184144
000]	[0x01	write mem	2e ->		184144
%r0,[%r2]		1db	main+0x34	10100	184145
001] -> writ	[0x01	read mem	5a <-		184145
0000		write rO	0000005a ->		184146
IncrementC		bl_s	main+0x38	10104	184147
		write r31	00010106 ->		184147
%r0,	add	CounterBy2	Increment	100b4	184148
		write r0	0000005c ->		184148
extb	+0x04	CounterBy2+	Increment	100b8	184149
~		write r0	0000005c ->		184149
j_s	+0x08	CounterBy2+	Increment	100bc	184150
2rn [2r2]		eth	main±0v3a	10106	184151

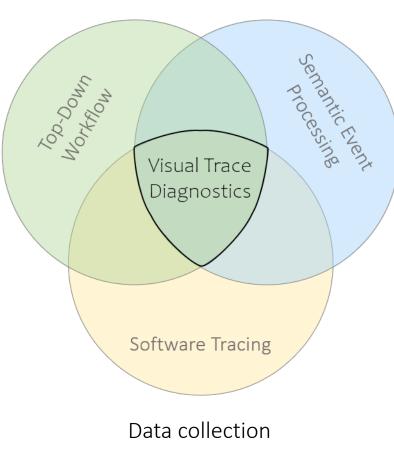
Т	2.3171	CONCEXE SWICEN ON CFO O CO CONCIDI	<pre>ivc] Starting key provisioning</pre>
1	9.330]	xQueueReceive(CtrlDataQueue, 100) retu	vc] Write root certificate
4	0.253]	OS Ticks: 8109	vc] Write device private key
Ч	1.253]	OS Ticks: 8110	Svc] Write device certificate
Ч	1.270]	Context switch on CPU 0 to Pos_ADC_ISR	Svc] Key provisioning done
	1.281]	xQueueSendFromISR(CtrlDataQueue)	Svc] Starting WiFi
	1.290]	Context switch on CPU 0 to Control	r Svc] WiFi module initialized.
	1.868]	xQueueSend(MotorQueue)	IS-MAIN] WiFi connected to AP AndroidAP.
it	1.878]	Actor Ready: Motor	WS-MAIN] Attempt to Get IP.
	1.889]	Context switch on CPU 0 to Motor	WS-MAIN] IP Address acquired 192.168.0.51
.C	1.900]	xQueueReceive(MotorQueue, 10) returns	WS-LED] [Shadow 0] MQTT: Creation of dedicated MQ1 WS-LED] Sending command to MQTT task.
9	1.934]	xQueueReceive(MotorQueue, 10) blocks	QTT] Received message 10000 from queue.
"I	1.954]	Context switch on CPU 0 to Control	QTT] Looked up a7sw0r7rvpirn.iot.us-east-1.amazona
÷Ц	1,965]	xQueueReceive(CtrlCmdQueue, 0) timeout	MQTT] MQTT Connect was accepted. Connection establ
il	1.977]	xQueueReceive(CtrlDataQueue, 100)	MQTT] Notifying task.
.il	1,9901	xOueueReceive(CtrlCmdOueue, 0) timeout	AWS-LED] Command sent to MQTT task passed.

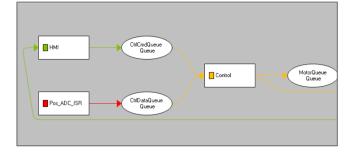
	Instruction Trace	Software Event Trace	Application Logging
Producer	Processor core	Software	Software
Abstraction Level	Low	Medium	High
Overhead	None	Minor	Depends on method
Special HW needed	Yes	No	No

VISUAL TRACE DIAGNOSTICS



Visualization allowing for drill-down from overviews to details

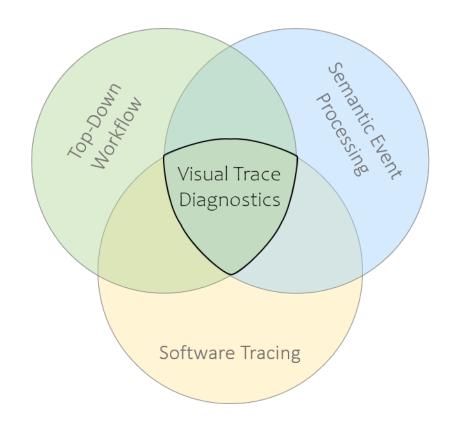




Data processed into a meaningful model connecting related events and objects

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

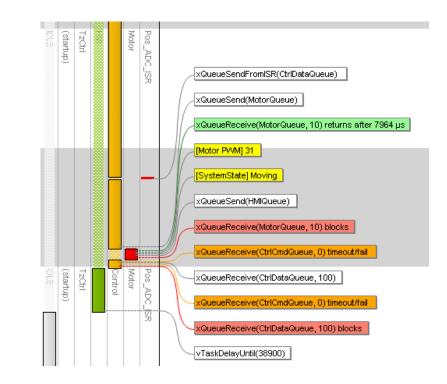
When you need a timeline of your software at runtime

Especially important when

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

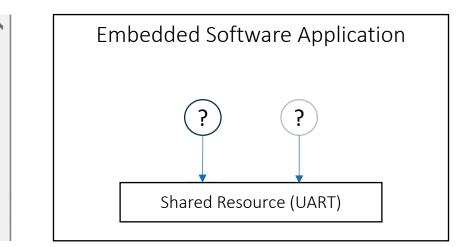
Superior to classic "printf debugging"

- 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.

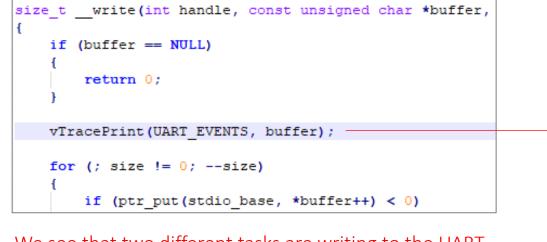
Error detected in full-system testing: Occasional data corruption

Two types of data written, "LLL..." and "HHH..."

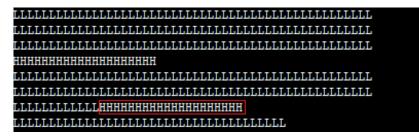
Sometimes they are mixed up and corrupt the data stream

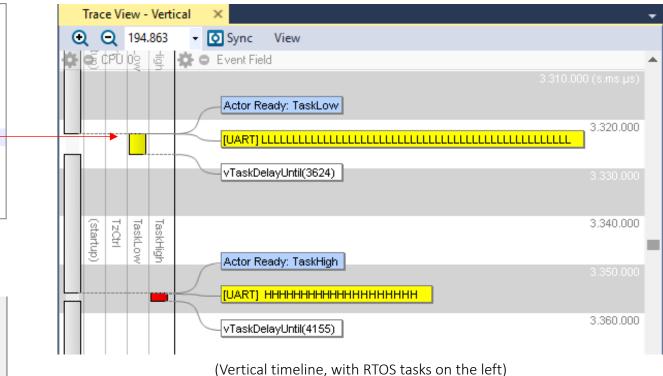
EXAMPLE 1: RECORD A TRACE

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



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





EXAMPLE 1: USE TOP-DOWN ANALYSIS

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

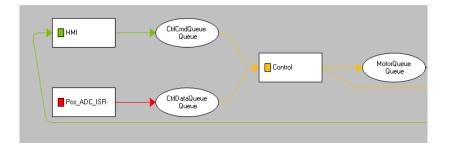
Visual overview of task response times (10 s) Trace view at selected location (15 ms) Actor Instance Graphs - Actor... × Trace View - Vertical -Sync View 👛 o deu o 🥸 🗅 Eivent Field 📕 (startup) 6.500 ms.µs TzCtrl Actor Ready: TaskLow TaskLow TaskLow 6.000 ms.us TaskHigh 5.500 ms.us Actor Ready: TaskHigh 5.000 ms.us (UART) ННННННННННННННННН 8.156.000 4.500 ms.µs _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ Red task preempts yellow task while writing data! 4.000 ms.µs Each data point is one full execution of a task (a job). 3.500 ms.µs vTaskDelavUntil(8955) The Y axis shows e.g. task response time. 3.000 ms.µs Double-click to navigate the Trace view here. 8.158.000 2.500 ms.us 2.000 ms.us 1.500 ms.us 1.000 ms.us (startup) 500 µs vTaskDelavLIntil(8456) 0 us 10.000.000 5.000.000 (s.ms.us) татататататан.

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

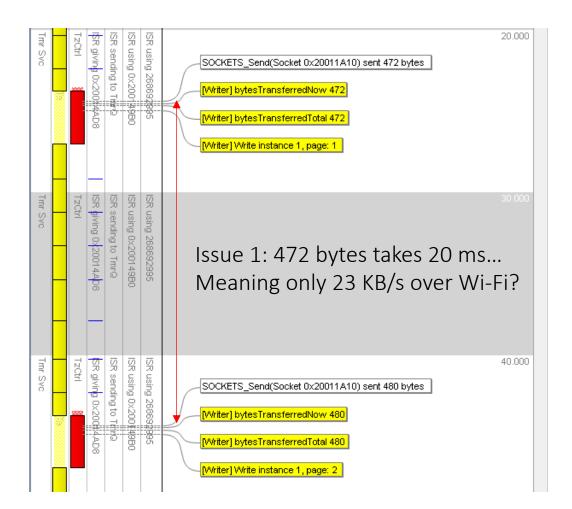
Solution: Add mutex calls for mutual exclusion, or move the writes to a single task.

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)

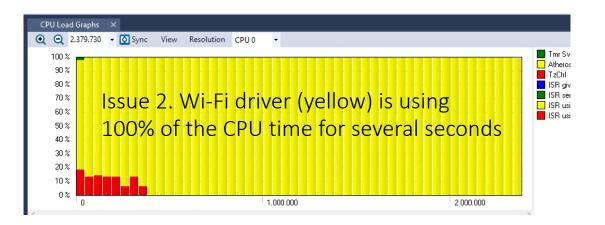


EXAMPLE 2: SOFTWARE DESIGN FLAWS

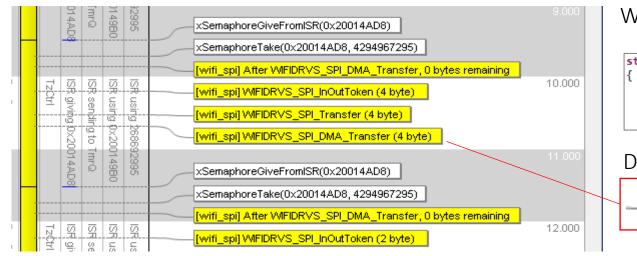


When evaluating an IoT demo from an MCU vendor, **two issues** where noticed immediately in the views...





EXAMPLE 2.1: SLOW WI-FI



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...

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?

[wifi_spi] WFIDRVS_SPI_DMA_Transfer (4 byte)

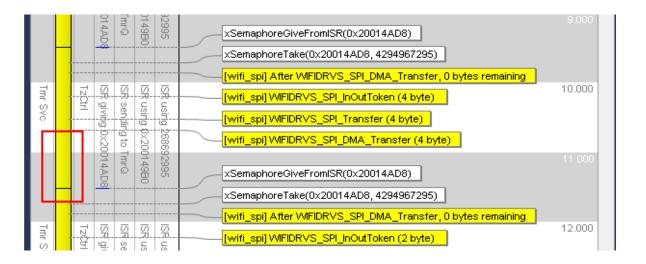


EXAMPLE 2.2: HIGH CPU LOAD

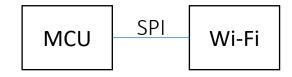
The trace view shows that WiFi driver task <u>doesn't</u> <u>suspend</u> while waiting for the SPI transfer to finish.

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:







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

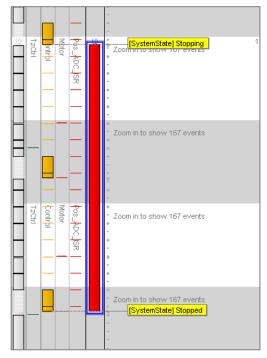
Keep Track of Software Timing and Resource Usage Metrics

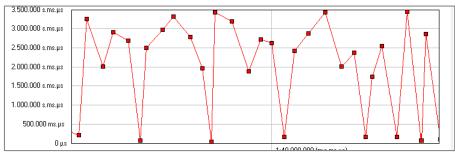
Software Timing

- 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

© IoTOnlineConference.com All rights reserved



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)



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