Zephyr™ Project
Developer Summit
June 8-10, 2021 • @ZephyrIoT

Using Visual Trace Diagnostics on Zephyr Applications

JOHAN KRAFT, PERCEPIO AB

Speaker

Johan Kraft, PhD CS

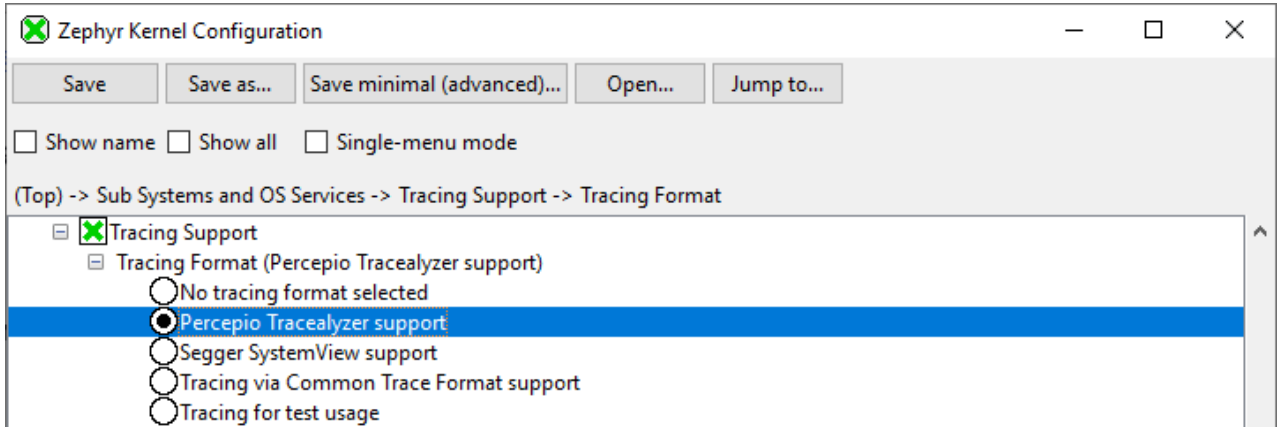
CEO and CTO, Percepio AB, Sweden

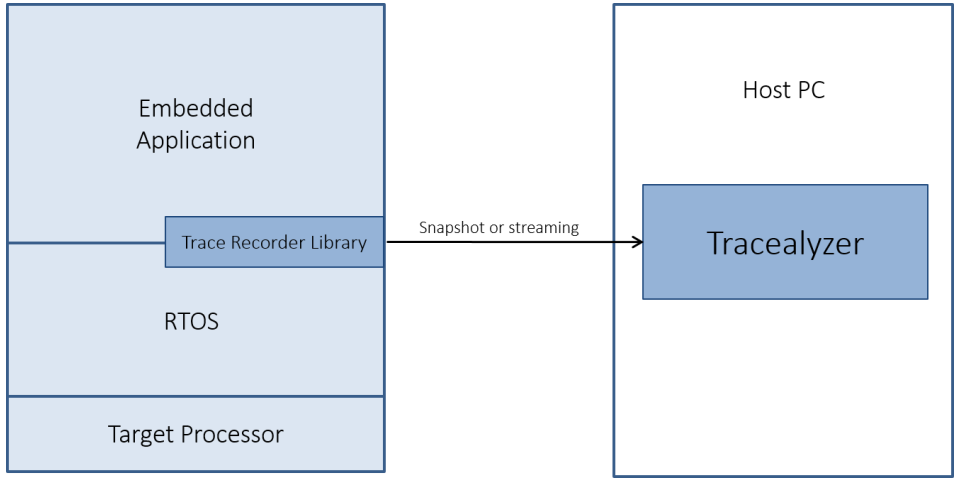
- Focus: visual trace diagnostics for simplified embedded software development
- Original developer of Percepio Tracealyzer (2009-)
- 20 years experience in RTOS tracing
- Two granted patents
- PhD in computer science, basically about practical methods for embedded software timing analysis, in collaboration with industry (2010)
- Early career at ABB Robotics

New Tracing Support in Zephyr 2.6

Zephyr™ Project
Developer Summit

Under “Sub Systems and OS Services” -> “Tracing Support” -> “Tracing Format”



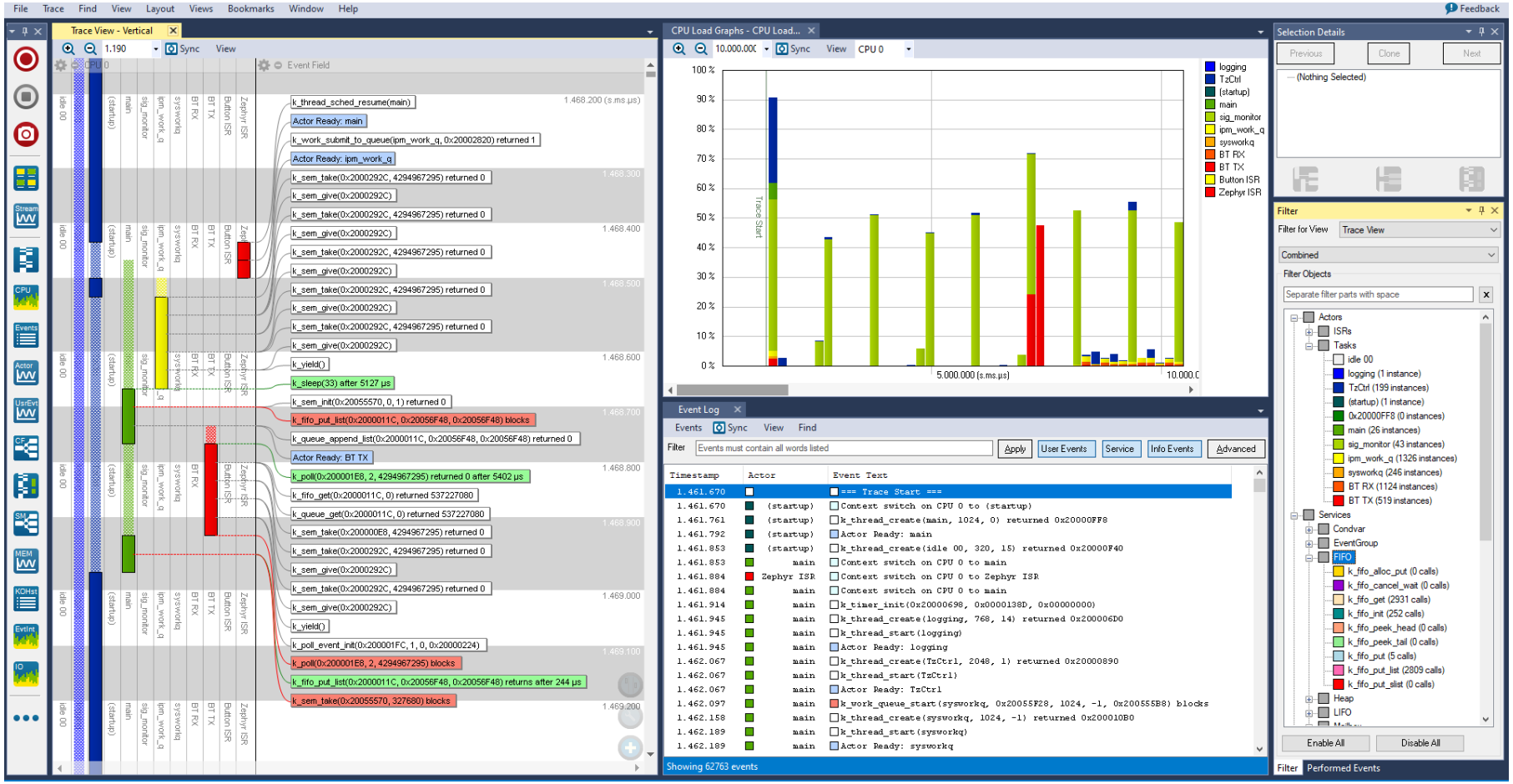


Includes Percepio’s trace recorder*, now adapted for Zephyr and provided under Apache 2.0.
Found under modules/debug/TraceRecorder

Provides data for Visual Trace Diagnostics in Percepio Tracealyzer, mainly targeting application developers.

*Hosted at https://github.com/percepio/

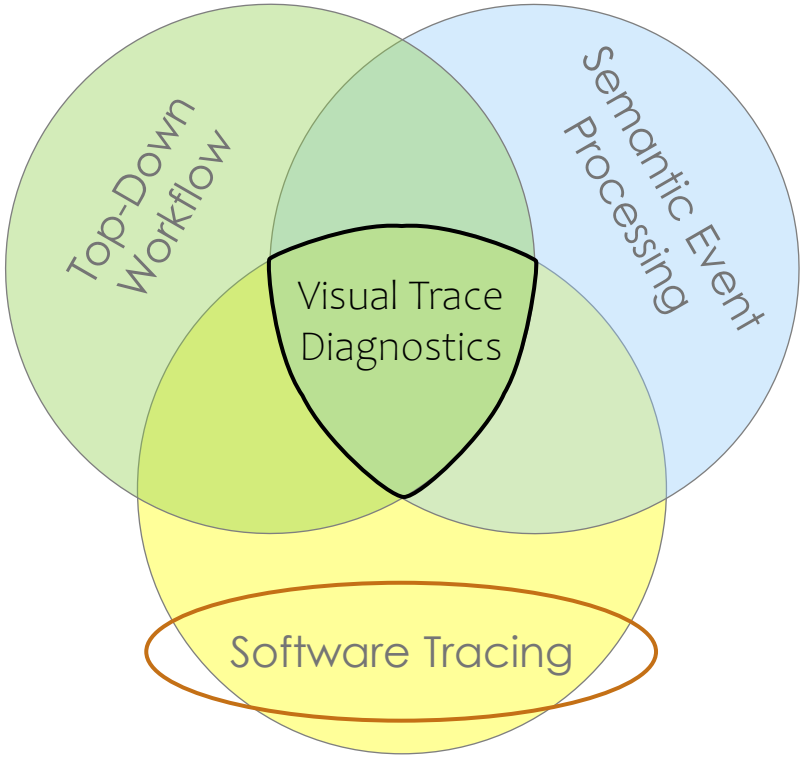
Percepio Tracealyzer



- Live trace streaming
- Flexible data channel
  - STLINK
  - Segger J-Link
  - TCP/IP, USB CDC, …
  - Customizable
    - See /streamports folder
- Developed since 2009
- Commercial product
- Free for academic use

Provides visual trace diagnostics for several RTOS and Linux (using LTTng). Supports Windows and Linux as host.

What is Visual Trace Diagnostics



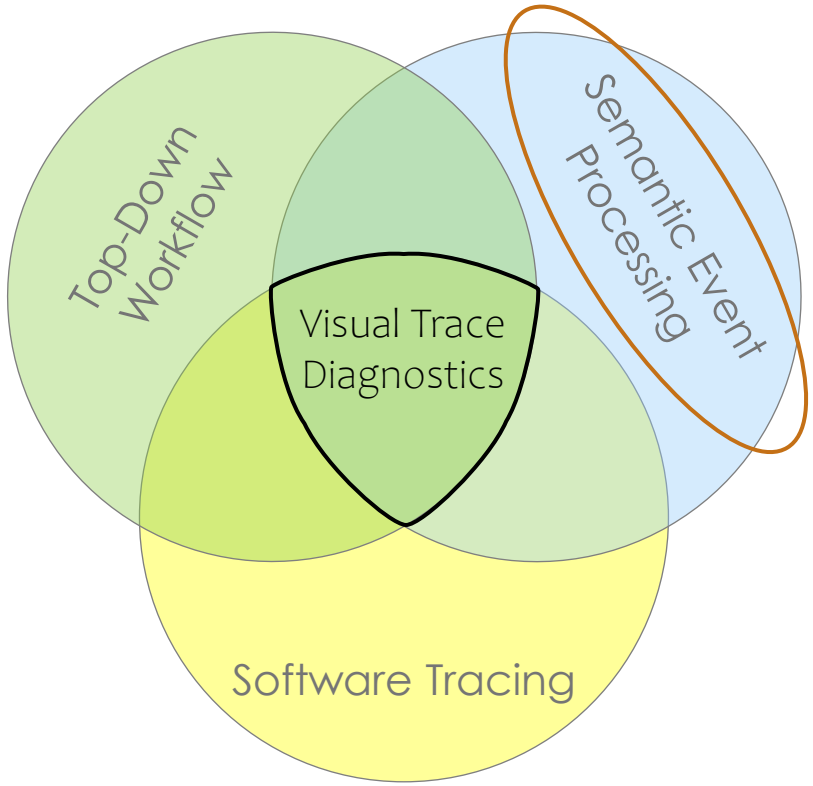
Input: a software trace, i.e. a list of events:

- Timestamp (typically µs or better)
- Event code (e.g. 42 = “context switch”)
- Event-specific arguments (e.g. thread = “MyThread”)

```
[  1.461.670] === Trace Start ===
[  1.461.670] Context switch on CPU 0 to (startup)
[  1.461.761] k_thread_create(main, 1024, 0) returned 0x20000FF8
[  1.461.792] Actor Ready: main
[  1.461.853] k_thread_create(idle 00, 320, 15) returned 0x20000F40
[  1.461.853] Context switch on CPU 0 to main
[  1.461.884] Context switch on CPU 0 to Zephyr ISR
[  1.461.884] Context switch on CPU 0 to main
[  1.461.914] k_timer_init(0x20000698, 0x0000138D, 0x00000000)
[  1.461.945] k_thread_create(logging, 768, 14) returned 0x200006D0
[  1.461.945] k_thread_start(logging)
[  1.461.945] Actor Ready: logging
[  1.462.067] k_thread_create(TzCtrl, 2048, 1) returned 0x20000890
[  1.462.067] k_thread_start(TzCtrl)
[  1.462.067] Actor Ready: TzCtrl
[  1.462.097] k_work_queue_start(sysworkq, 0x20055F28, 1024, -1, 0x200555B8) blocks
[  1.462.158] k_thread_create(sysworkq, 1024, -1) returned 0x200010B0
[  1.462.189] k_thread_start(sysworkq)
[  1.462.189] Actor Ready: sysworkq
[  1.462.219] Context switch on CPU 0 to sysworkq
[  1.462.219] Context switch on CPU 0 to main
[  1.462.250] k_work_queue_start(sysworkq, 0x20055F28, 1024, -1, 0x200555B8) returns after 153 µs
[  1.462.250] k_work_queue_start(ipm_work_q, 0x200538A0, 2048, -1, 0x00000000) blocks
[  1.462.341] k_thread_create(ipm_work_q, 2048, -1) returned 0x20000788
[  1.462.341] k_thread_start(ipm_work_q)
```

- Basic logs are useful for verifying specific things
  - What happens at X?
- But challenging to use for debugging purposes
  - Lots of data - difficult to overview
  - You might not know what to search for…

What is Visual Trace Diagnostics

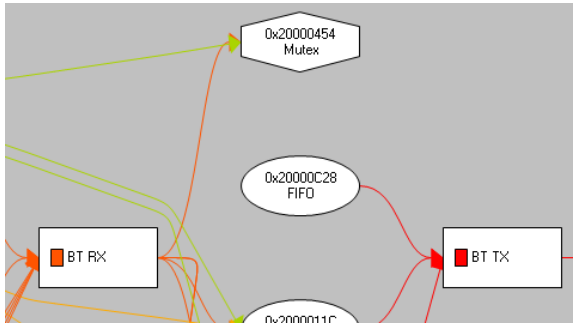



```
<KernelServiceGroup name="Mutex">
    <KernelService name="k_mutex_init"    operation="Initialize">
        <Parameter name="handle" format="{0}"/>
        <ReturnValue name="return" format=" returned {0}"/>
    </KernelService>
    <KernelService name="k_mutex_unlock"  operation="ReleaseMutex">
        <Parameter name="handle" format="{0}"/>
        <ReturnValue name="return" format=" returned {0}"/>
    </KernelService>
    <KernelService name="k_mutex_lock"    operation="LockMutex"
        <Parameter name="handle" format="{0}"/>
        <Parameter name="timeout" format="{0}"/>
        <ReturnValue name="return" format=" returned {0}"/>
    </KernelService>
</KernelServiceGroup>
```

Object History - 0x20000454 (Mutex)

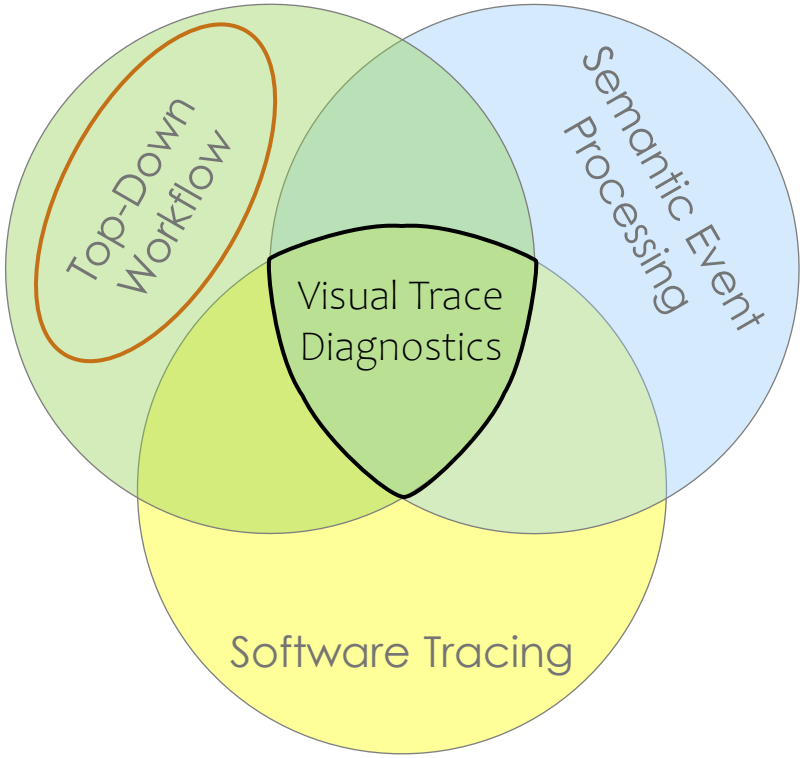
Sync | View | Filter Tasks | Filter Calls

Timestamp	Actor	Event	Block time	Status
9.108.978	BT RX	k_mutex_lock		Direct lock
9.109.009	BT RX	k_mutex_unlock		Released
9.263.214	sig_monitor	k_mutex_lock		Direct lock
9.302.338	BT RX	k_mutex_lock	63.538	Waiting for lock
9.365.875	sig_monitor	k_mutex_unlock		Released
9.365.875	BT RX	k_mutex_lock		Locked after waiting

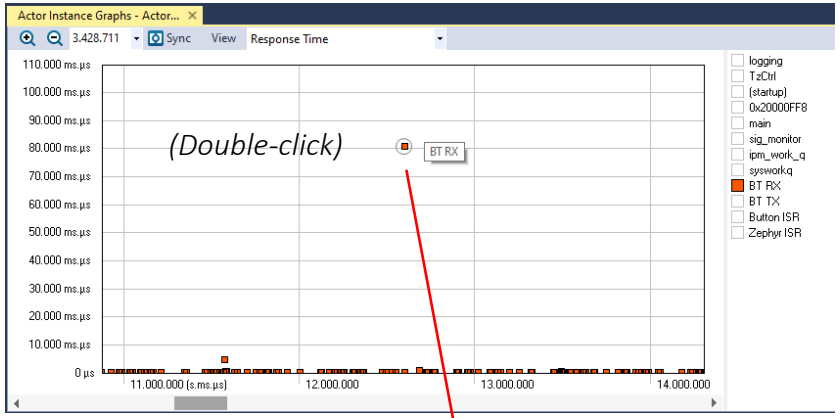


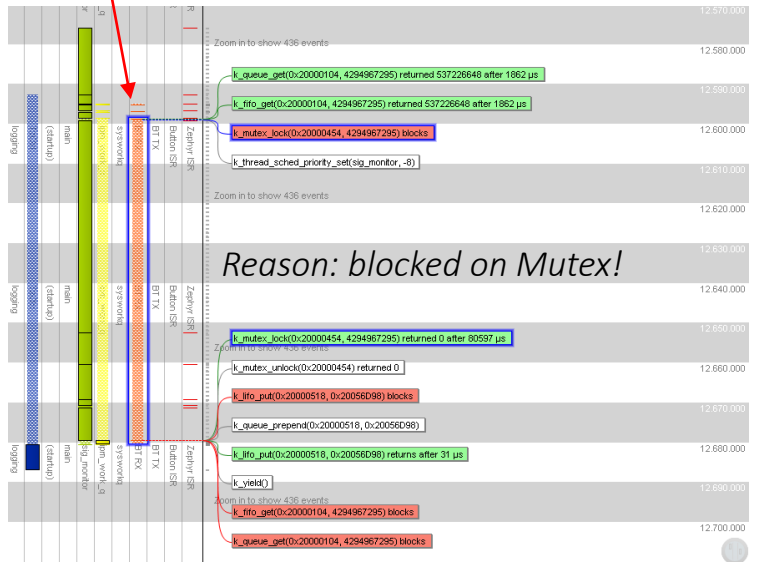
- Events processed into an object graph, using a semantic model of the events
  - e.g. Mutex events identify a Mutex object, with an “Owner” (a thread) modified by “Lock” and “Unlock” operations.
- The result can be visualized from many perspectives and at different abstraction levels

What is Visual Trace Diagnostics



Response time plot for thread BT RX





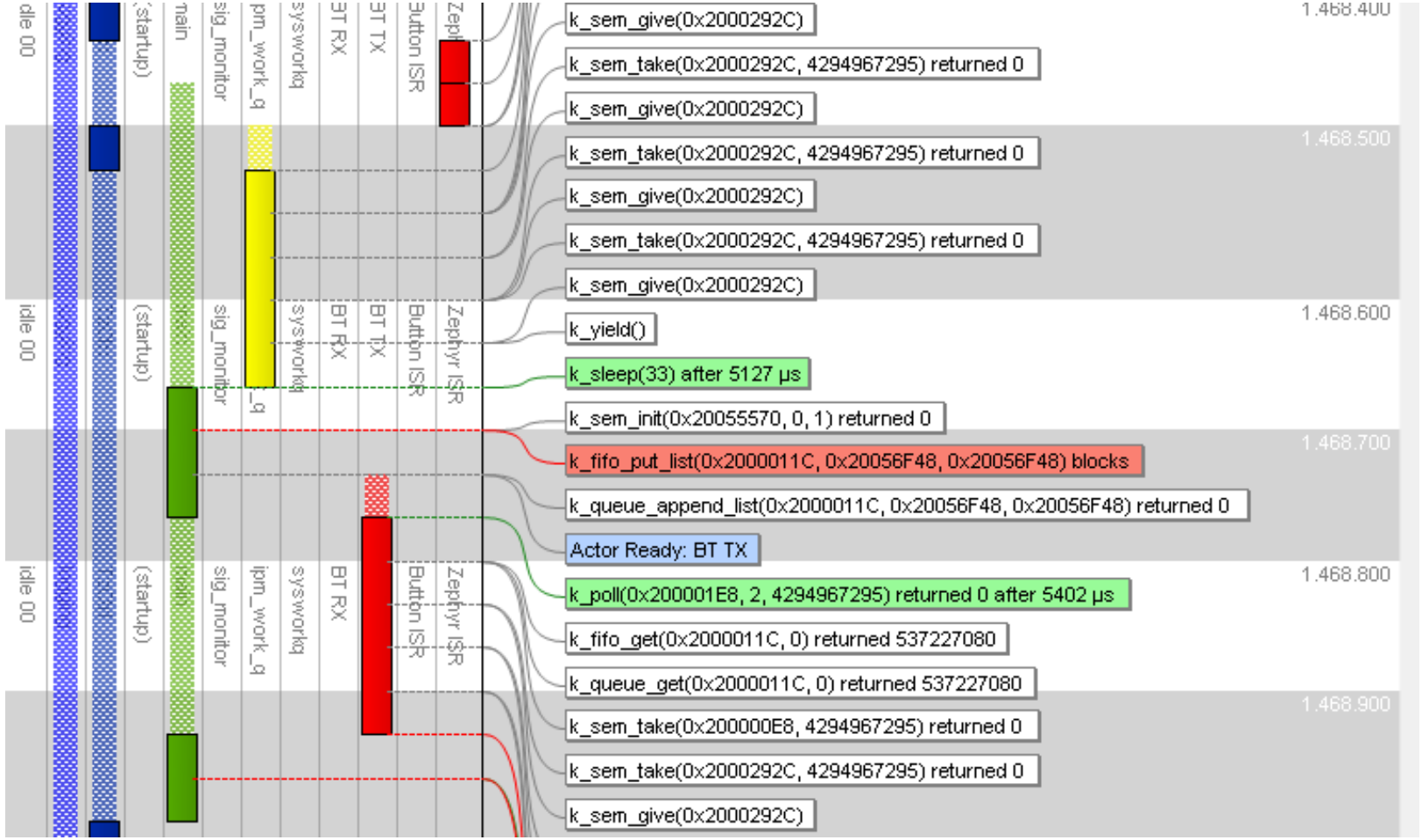
Multiple connected views allows a top-down workflow, suitable for debugging

- Spot anomalies visually in the overviews
- Drill down using detailed views to analyze the cause

Use Cases for Visual Trace Diagnostics

- Debugging of multithreaded RTOS applications
  - Especially for issues related to multithreading, resource usage or software timing
  - When the target system can't be halted for debugging
  - In general - whenever you want a timeline of the software
- Analyze application design
  - Analyze functional behavior over time (data plots, state diagrams) together with system-level execution
  - Find design violations wrt. best practices - can improve efficiency, testability and reliability
    - For instance, bad scheduling patterns, busy waiting or chaotic behavior due to timing variations.
- Measure software timing, resource utilization and performance
  - And see the *reason* behind the numbers. Find bottlenecks and optimize!
    - Improve application performance and user experience
    - Allow for using a more cost-effective processor
  - Major improvements are sometimes possible with small changes, assuming sufficient insight…

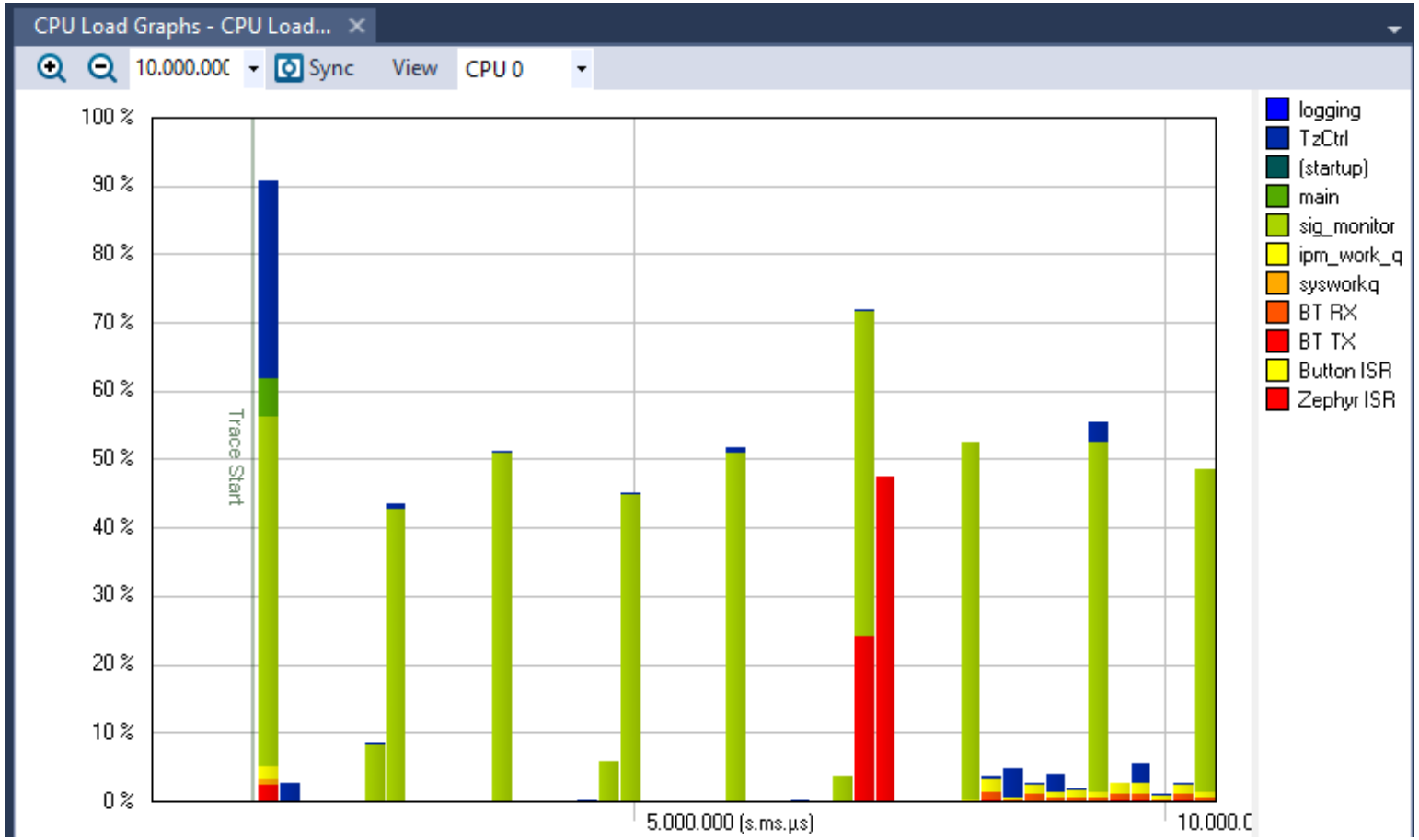
What you can see in Tracealyzer



From “automatic” kernel tracing

- Scheduling
- API calls (“kernel services”)
  - FIFO/LIFO
  - Semaphore
  - Mutex
  - Pipe
  - Queue
  - Heap
  - …
- Blocking and timeouts

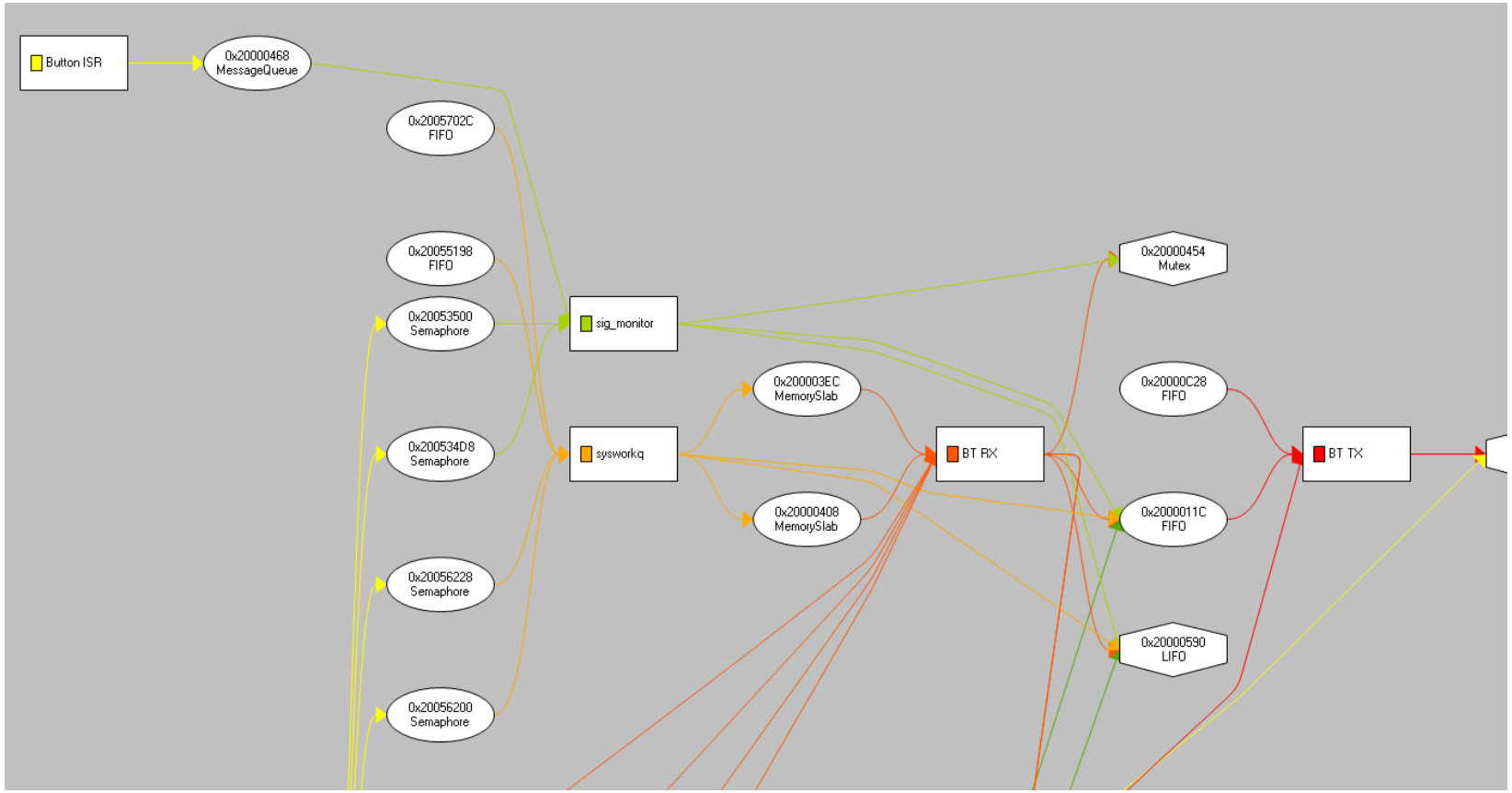
What you can see in Tracealyzer



From “automatic” kernel tracing

- CPU load
- Communication Flow
- Object History
- Execution Statistics
- Various Overviews

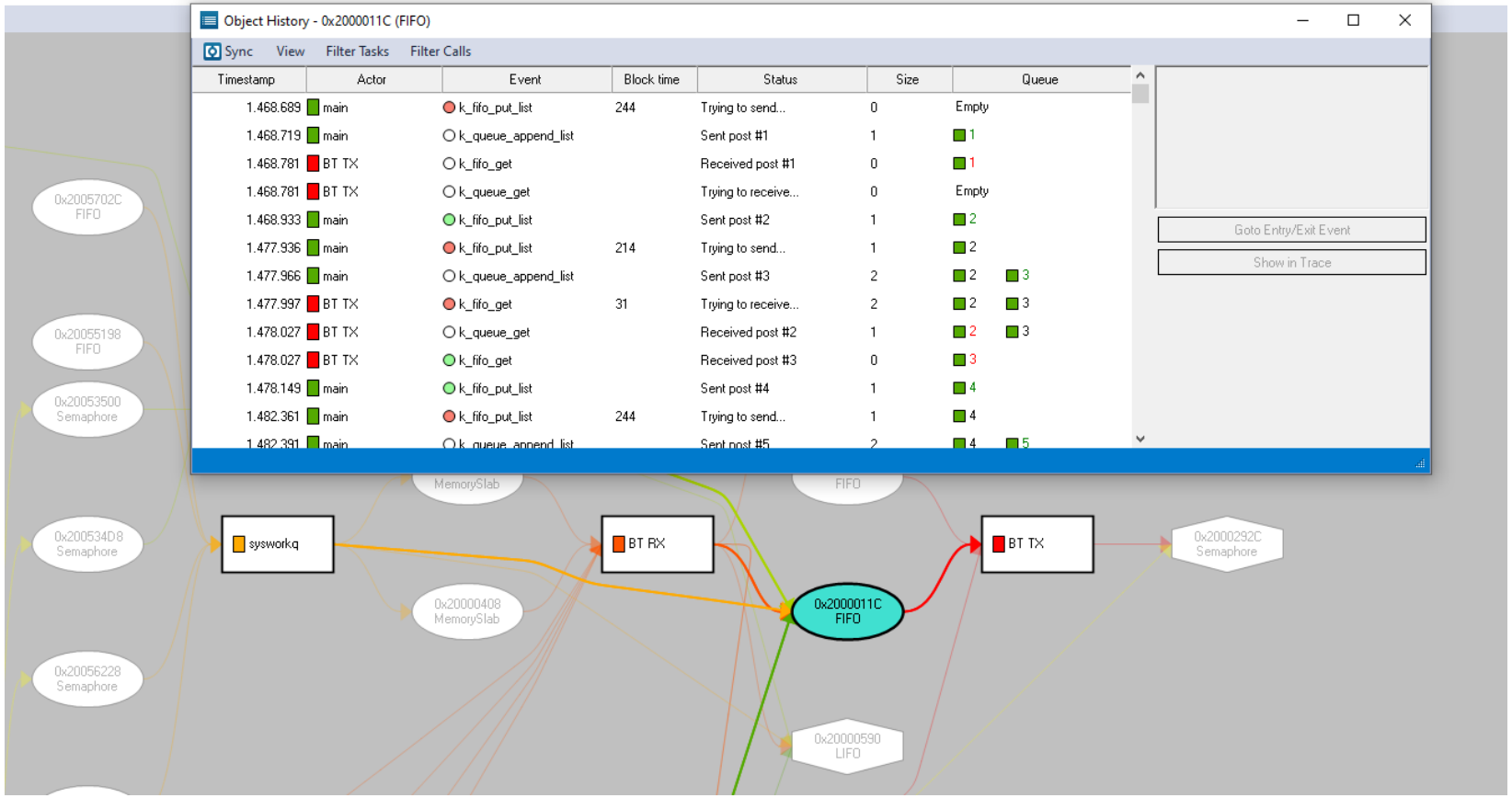
What you can see in Tracealyzer



From “automatic” kernel tracing

- CPU load
- Communication Flow
- Object History
- Execution Statistics
- Various Overviews

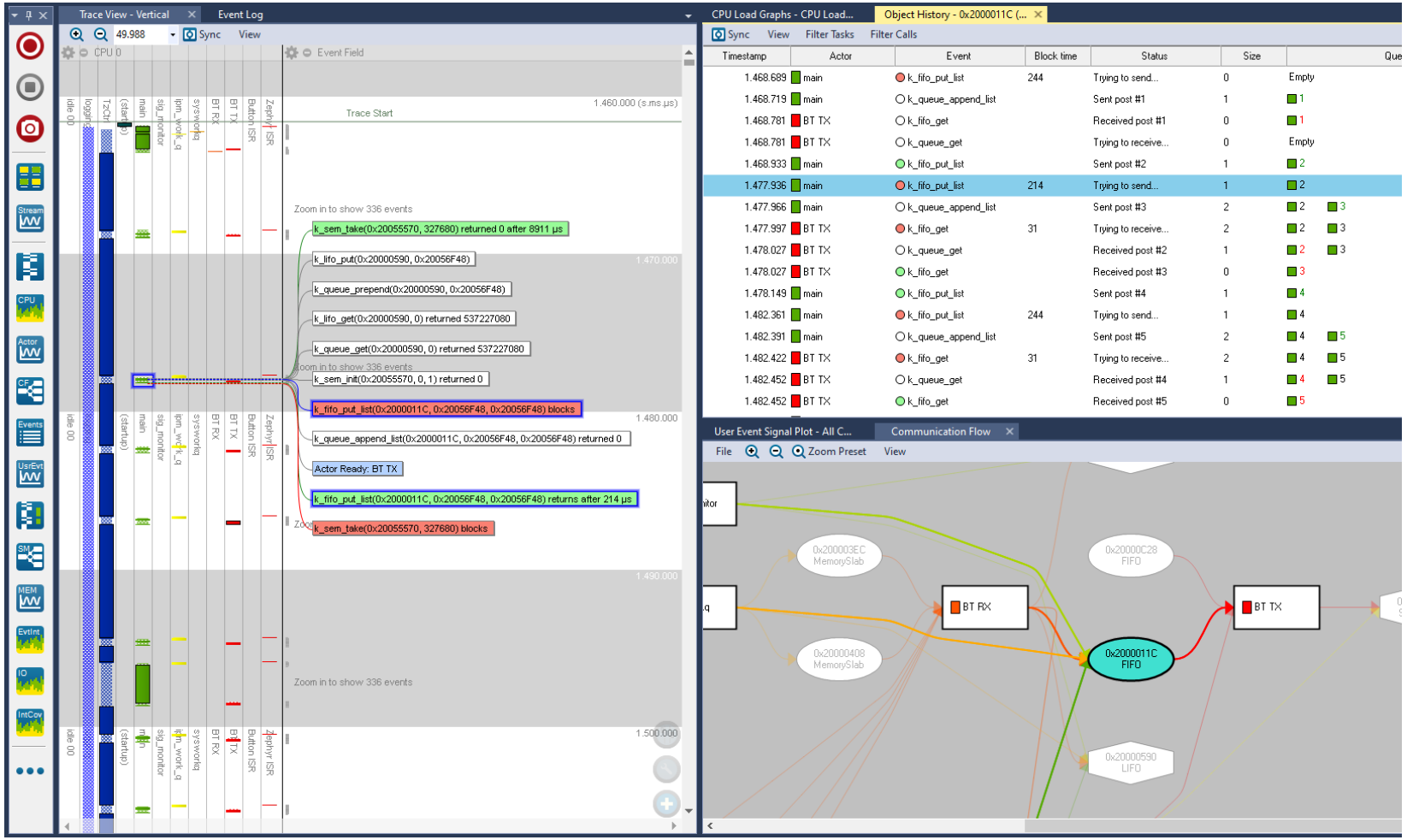
What you can see in Tracealyzer



From “automatic” kernel tracing

- CPU load
- Communication Flow
- Object History
- Execution Statistics
- Various Overviews

What you can see in Tracealyzer



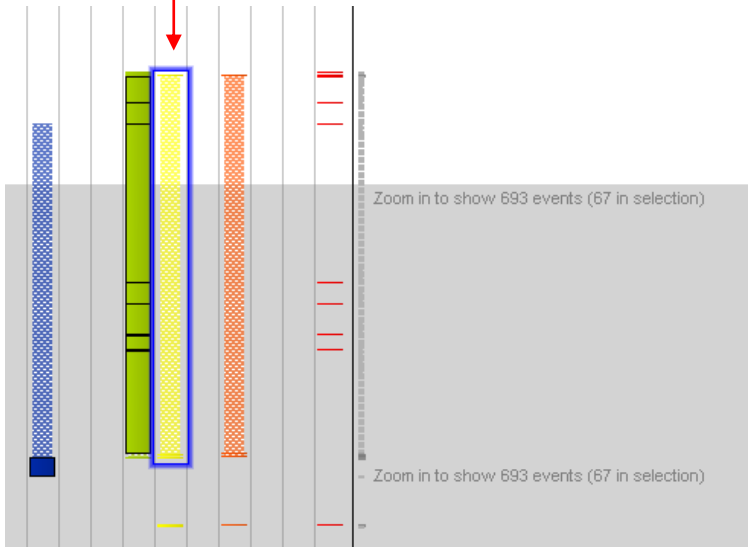
From “automatic” kernel tracing

- CPU load
- Communication Flow
- Object History (linked to trace)
- Execution Statistics
- Various Overviews

What you can see in Tracealyzer

Actor	Count	CPU Usage	Execution Time			Response Time			Periodicity (Ready)			Period
			Min	Avg	Max	Min	Avg	Max	Min	Avg	Max	Min
idle 00	1	87.307	38.132.263	38.132.263	38.132.263	43.603.546	43.603.546	43.603.546	N/A	N/A	N/A	N/A
logging	1	0.000	92	92	92	72.388	72.388	72.388	N/A	N/A	N/A	N/A
TzCtrl	199	1.254	92	2.747	57.709	122	20.294	111.115	200.165	220.428	311.127	200.165
(startup)	1	0.000	183	183	183	183	183	183	N/A	N/A	N/A	N/A
main	26	0.025	153	397	2.655	305	641	2.899	671	3.143	9.338	671
sig_monitor	43	9.377	92	95.245	104.065	153	96.649	109.161	702	1.025.116	1.109.161	702
ipm_work_q	1326	0.596	0	183	916	31	885	103.210	458	32.959	6.720.459	458
sysworkq	246	0.358	0	610	2.167	31	702	2.258	702	178.162	14.085.541	702
BT RX	1124	0.410	31	153	519	31	763	102.081	458	38.879	6.803.894	458
BT TX	519	0.196	61	153	214	61	244	641	671	84.290	6.720.459	671
Button ISR	2	0.000	0	0	0	0	0	0	7.650.574	7.650.574	7.650.574	7.650.574
Zephyr ISR	1662	0.476	0	122	143.799	0	122	143.799	31	26.276	302.460	31

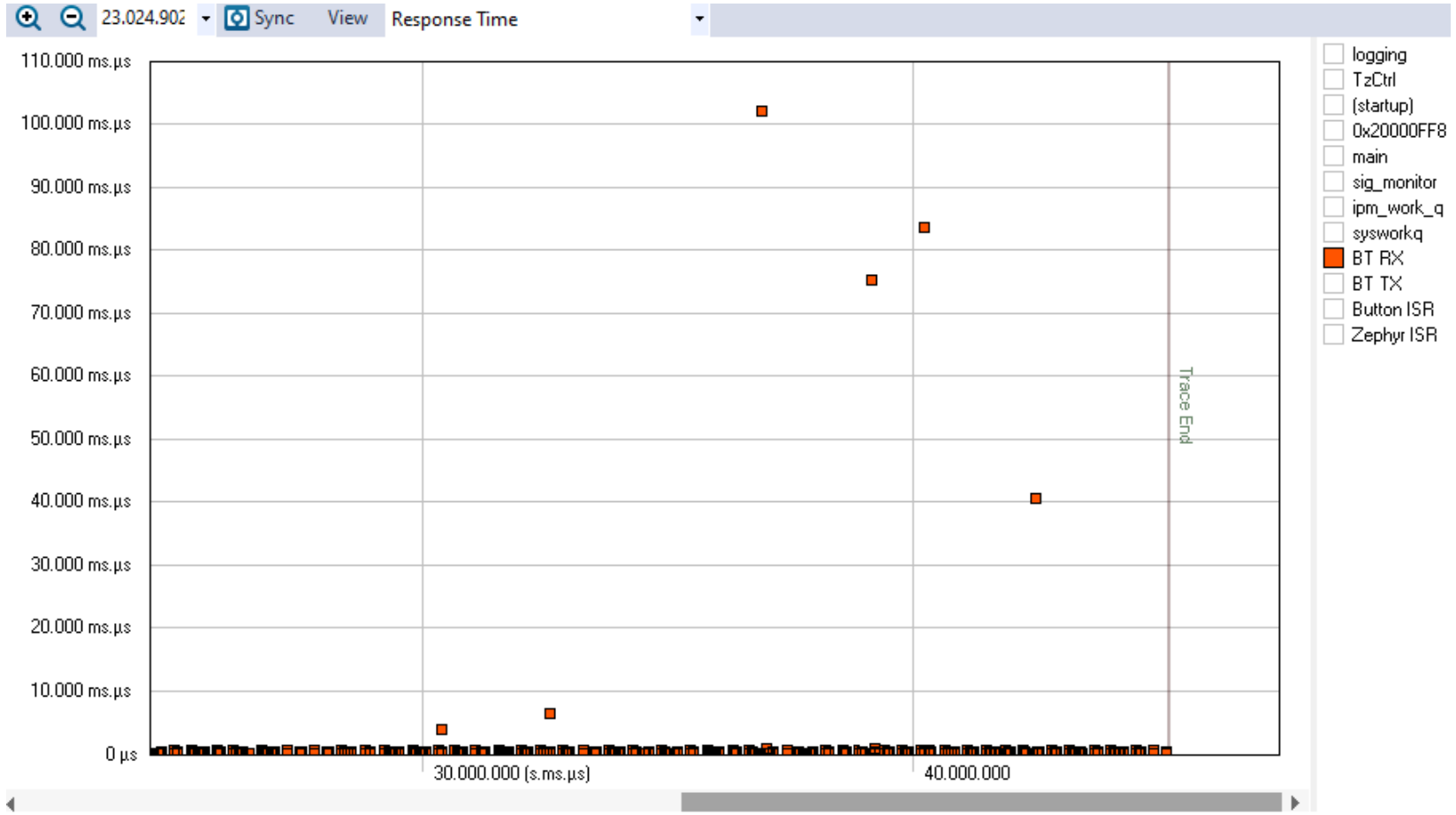
(click)



From “automatic” kernel tracing

- CPU load
- Communication Flow
- Object History
- Execution Statistics
- Various Overviews

What you can see in Tracealyzer



From “automatic” kernel tracing

- CPU load
- Communication Flow
- Object History
- Execution Statistics
- Various Overviews

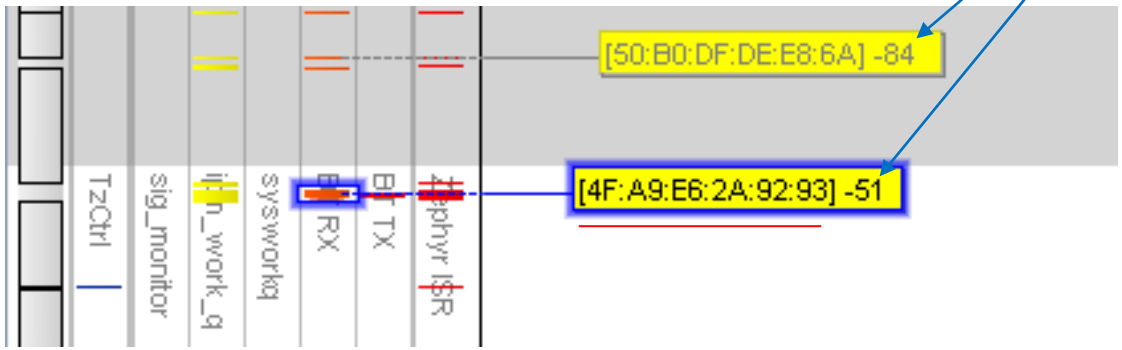
What you can see in Tracealyzer


```
if (dev_id < 0) {
    dev_id = tz_ble_device_store(addr, rssi);
}

/* Log BLE signal strength as a user event */
vTracePrintF(tz_ble_device_list[dev_id].ts_handle, "%d", rssi);
```

User event channel name (= BLE device)

Value (= signal strength)



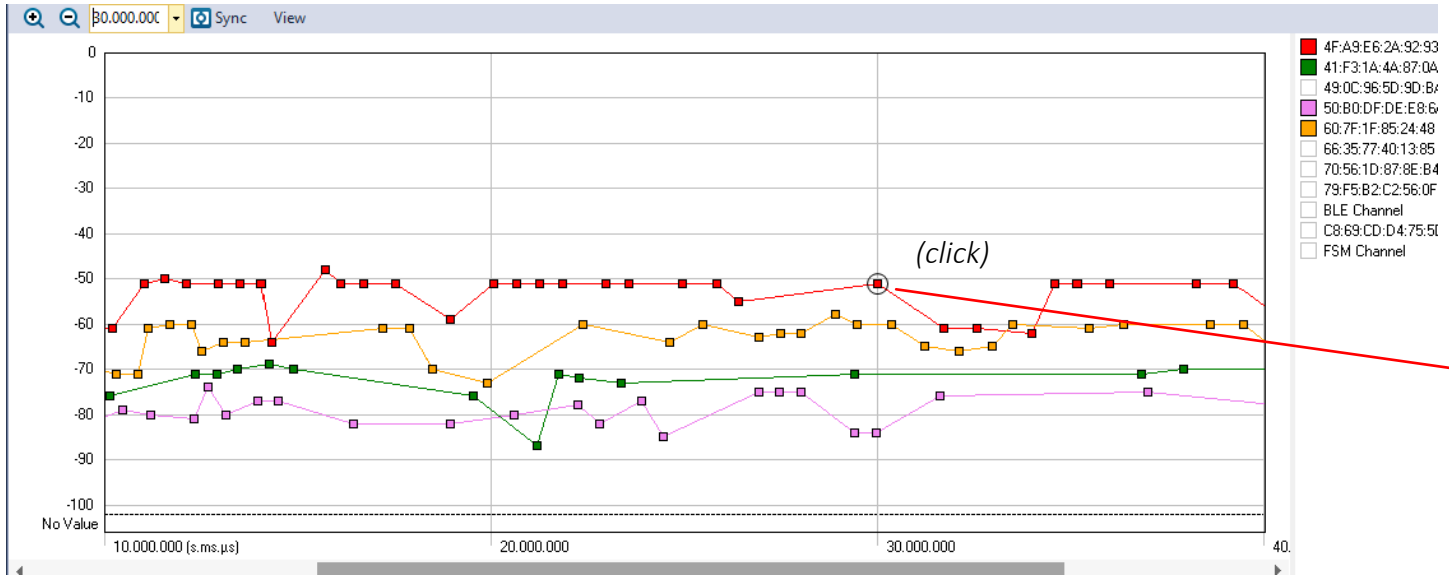
From explicit “user events”

- Custom application events
- Data plots
- Intervals
- State machines

What you can see in Tracealyzer

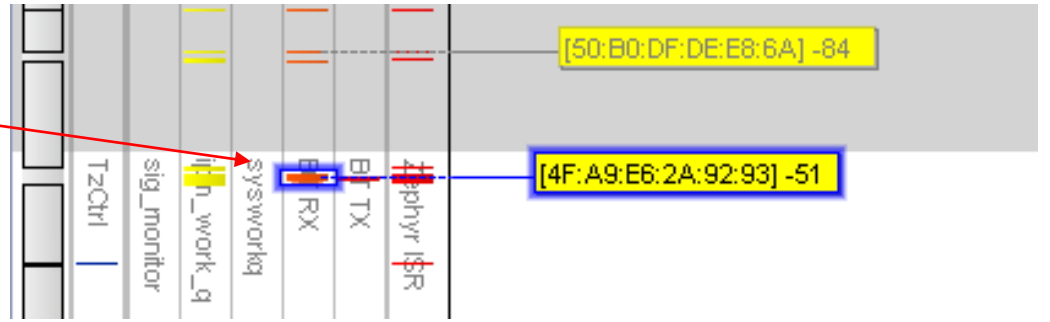

```
if (dev_id < 0) {
    dev_id = tz_ble_device_store(addr, rssi);
}

/* Log BLE signal strength as a user event */
vTracePrintF(tz_ble_device_list[dev_id].ts_handle, "%d", rssi);
```



From explicit “user events”

- Custom application events
- Data plots
- Intervals
- State machines

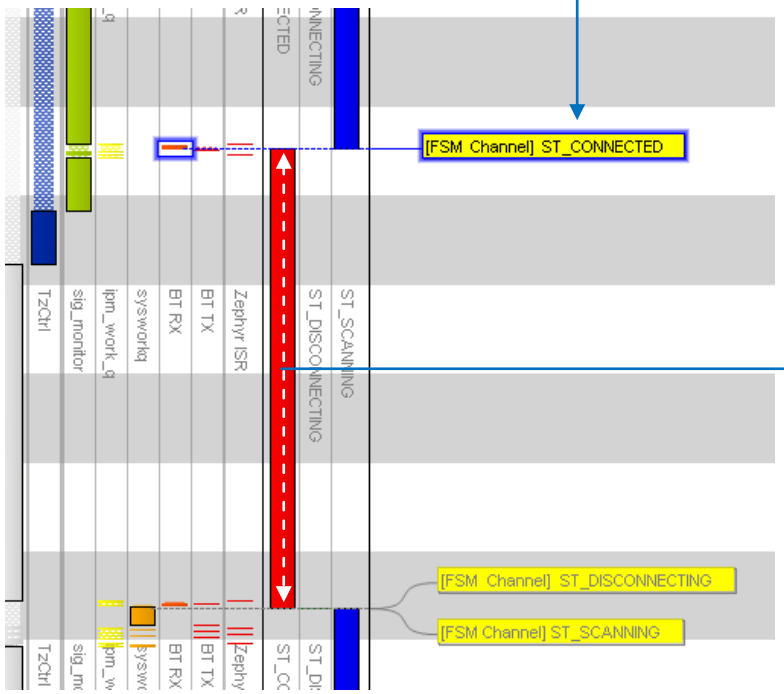


What you can see in Tracealyzer


```
vTracePrint(tz_channel_fsm, "ST_SCANNING");


/* This demo doesn't require active scan */
err = bt_le_scan_start(&scan_param, device_found);
if (err) {
```

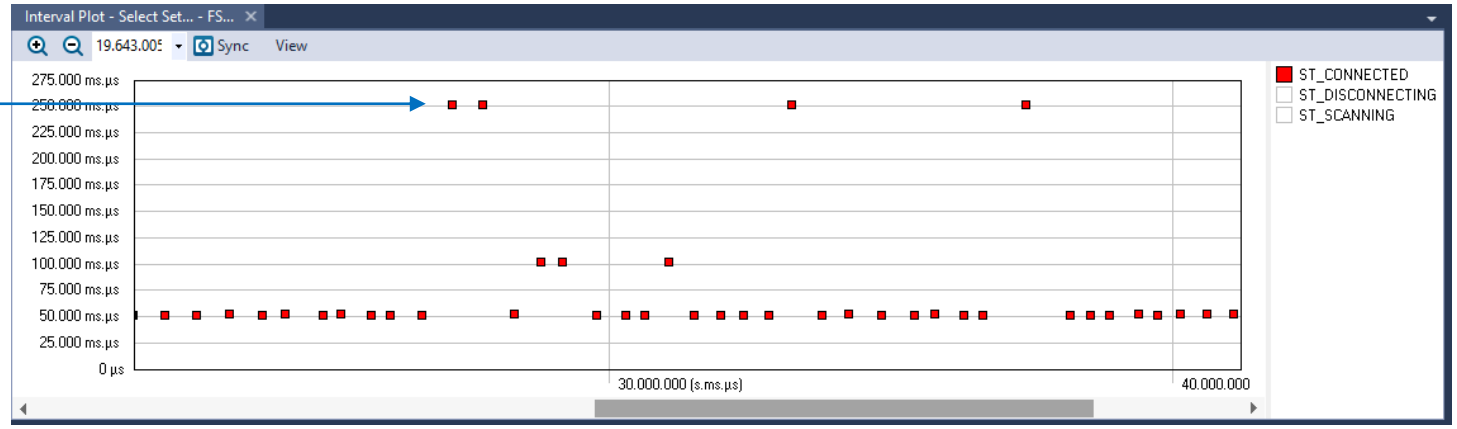
States as intervals



From explicit “user events”

- Custom application events
- Data plots
- Intervals
- State machines

Interval plot – The Y-axis shows the duration of each interval / state.

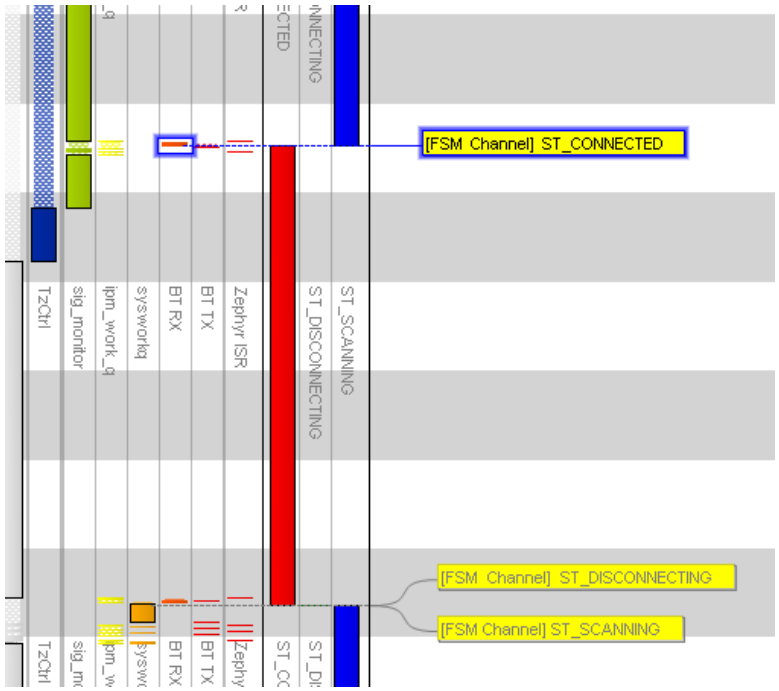


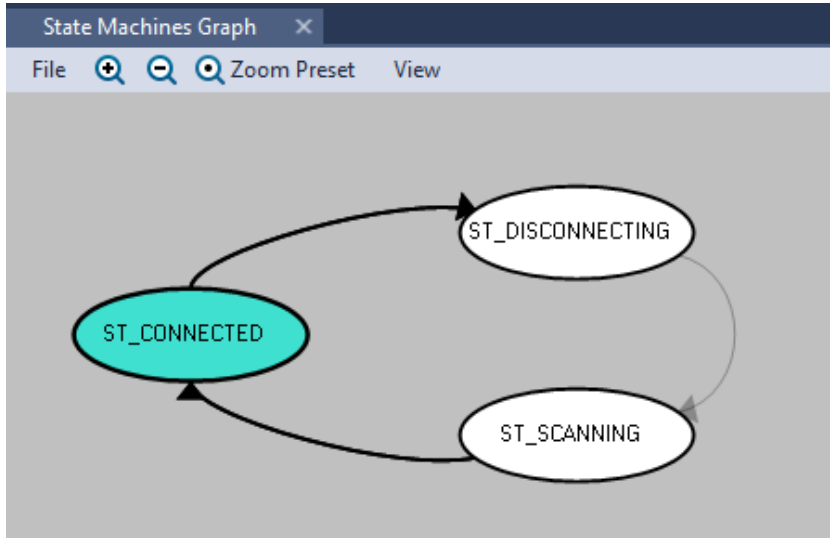
What you can see in Tracealyzer


```
vTracePrint(tz_channel_fsm, "ST_SCANNING");


/* This demo doesn't require active scan */
err = bt_le_scan_start(&scan_param, device_found);
if (err) {
```

States as intervals



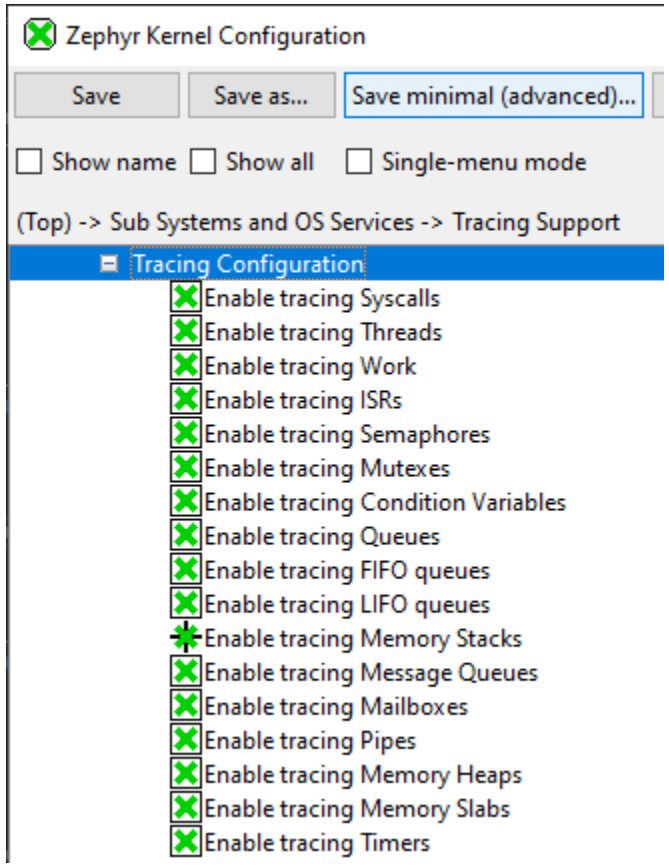


From explicit “user events”

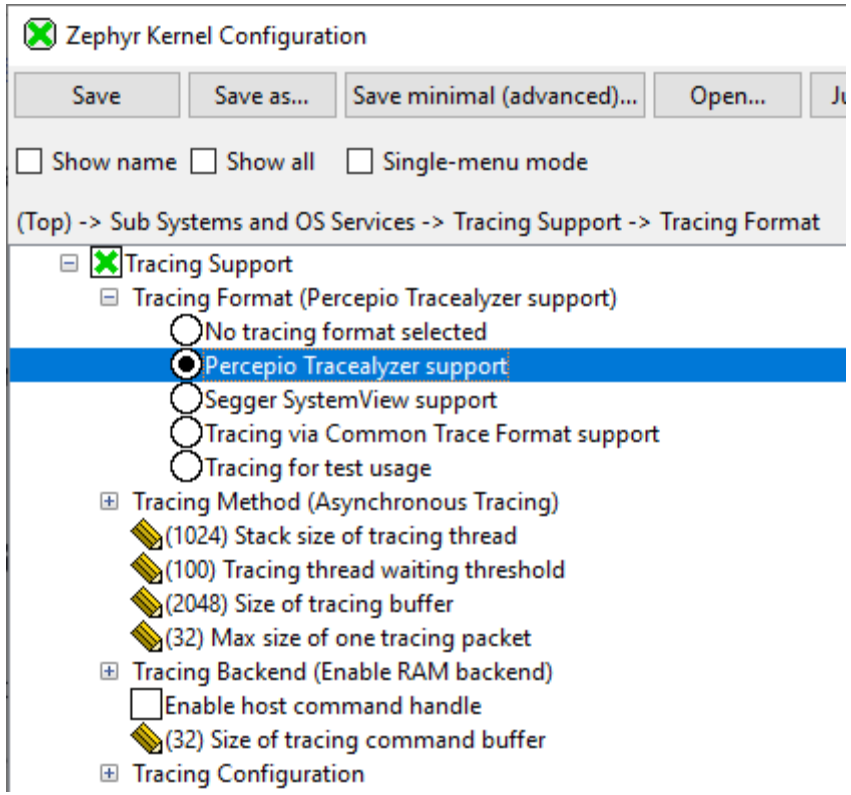
- Custom application events
- Data plots
- Intervals
- State machines

How Enable Tracealyzer support

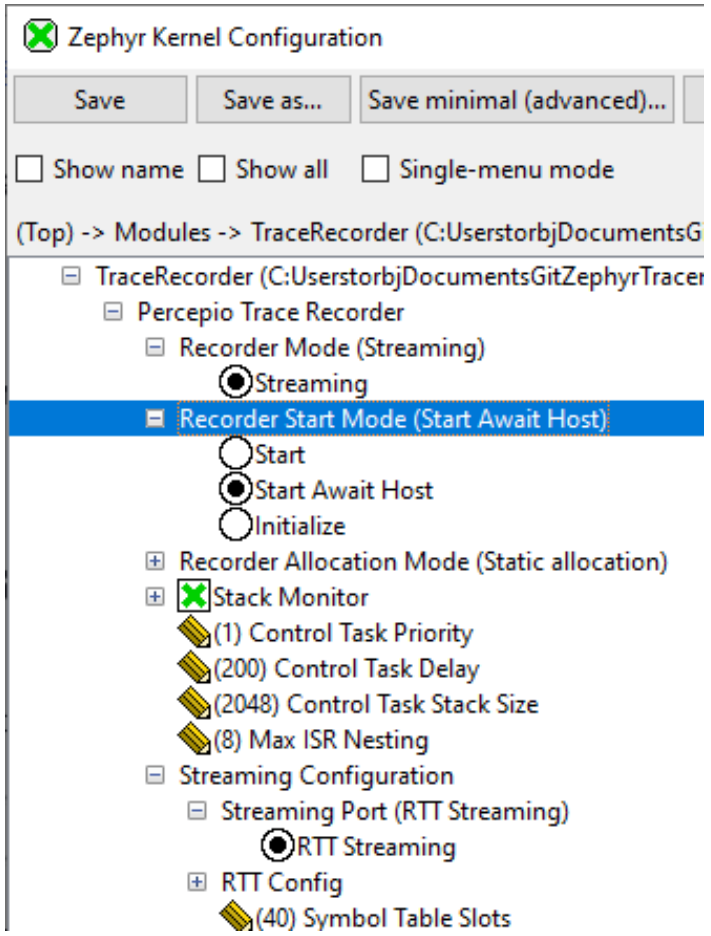
Zephyr™ Project
Developer Summit



Enable tracing (with filters)

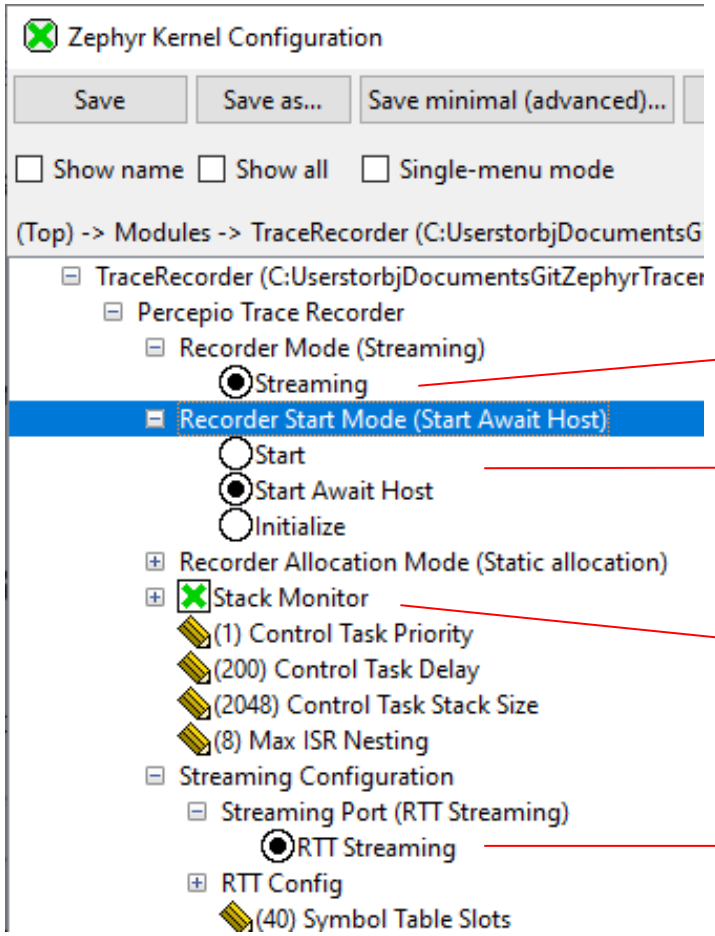


Select Percepio Tracealyzer support



Configure (optional)

How Enable Tracealyzer support



Currently only streaming mode supported.
Snapshot mode is coming (ring-buffer in target RAM)

Start: Start tracing directly on startup
Start Await Host: Blocks until Start command from Tracealyzer
Initialize: Tracing begins on Start command from Tracealyzer

Periodic stack usage checking per thread (remaining margin)
Runs as a periodic task,

The selected “stream port” (how to output the data)
Additional stream ports are found in the /streamports folder.

Learning More

- https://percepio.com
- support@percepio.com

- Using Tracealyzer on Zephyr requires:
  - Zephyr v2.6
  - Tracealyzer v4.5 – provides Beta support for Zephyr (released on June 15)
  - A way to stream the data to host, e.g. a Segger J-Link.

Thanks for joining!

- Questions?


Zephyr™ Project
Developer Summit
June 8-10, 2021 ▪ @ZephyrIoT