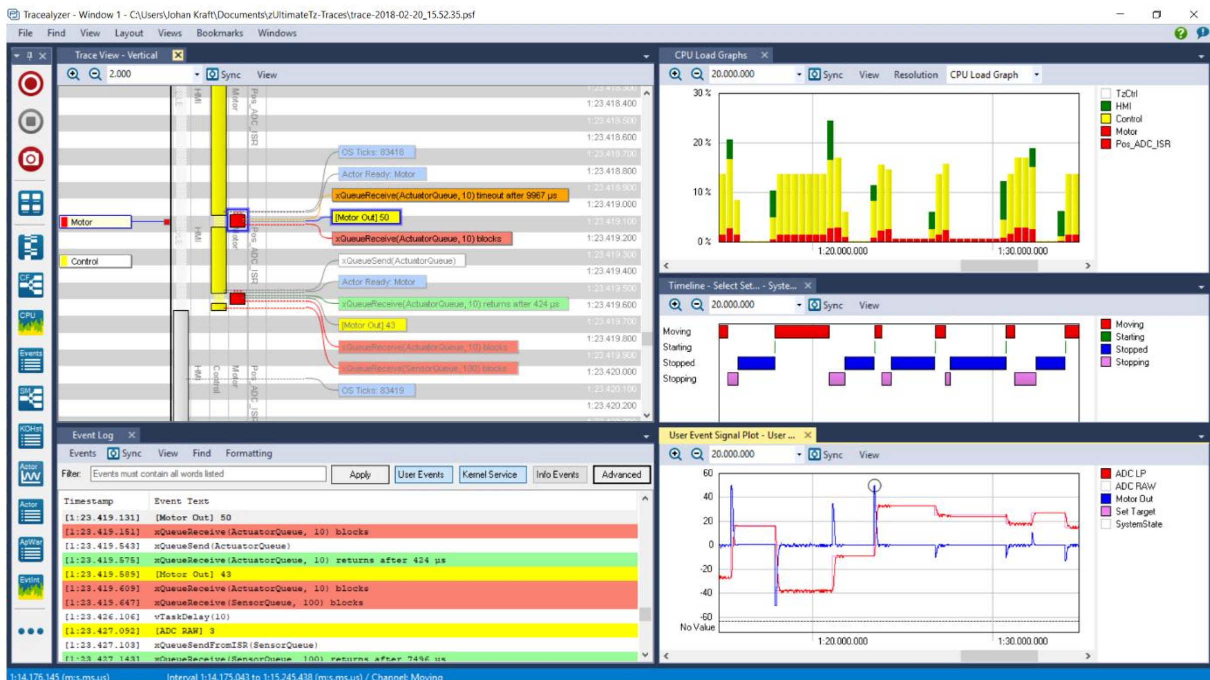
percepio
SENSING SOFTWARE

Percepio® TraceRecorder Integration Guide

Version 1.1.2 – June 15, 2023.



Percepio® TraceRecorder Integration Guide. Copyright © Percepio AB, 2023
https://percepio.com

1. Introduction

The TraceRecorder library is the target-side component of Percepio® Tracealyzer®. This is a C library intended for RTOS and bare-metal applications that has been refined since 2009. This guide reflects Tracealyzer v4.8, although most is accurate also for v4.6 and v4.7.

TraceRecorder is provided under the Apache 2.0 license. TraceRecorder is available at:

https://github.com/percepio/TraceRecorderSource

TraceRecorder comes with support for various RTOSes, such as FreeRTOS, ThreadX and Zephyr. These existing integrations are found in the /kernelports folder.

This guide describes how to integrate the TraceRecorder library with other real-time operating systems and similar software platforms, i.e. for cases where existing support is not available in the /kernelports folder.

This is intended for integration by developers/maintainers of real-time operating systems (RTOS) and middleware such as communication stacks. It may also be used with other software platforms, and for bare-metal applications without an RTOS. Support for 64-bit systems was added in Tracealyzer v4.8.

Several Tracealyzer features can be used on any system, without specific kernel tracing support in TraceRecoder. This is described in the “BareMetal” section of the Tracealyzer User Manual (see “Creating and Loading Traces” -> “Percepio TraceRecorder”). The “BareMetal” solution is also recommended as starting point for developing new RTOS integrations as described in this document.

However, several Tracealyzer features are designed for typical RTOS features, such as task scheduling, queues, mutexes and semaphores. We call these “kernel events”, although such instrumentation is possible also in the application code. The difference compared to generic “user events” is that kernel events are tagged with a “semantic operation” that makes Tracealyzer understand the meaning of the event, not only the name and arguments of a API call but also what the function actually did. For example, “Task X attempted to lock mutex Y but was blocked and after Z µs there was a timeout.” By integrating TraceRecorder as described this guide you can add your own “kernel events” and take full advantage of Tracealyzer on a broad range of software platforms.

This guide assumes you have a basic understanding of how the TraceRecorder library is used in our existing integrations with e.g. FreeRTOS, ThreadX and Zephyr. For instance, how to initialize and start the trace recording and common configuration options. Make sure to read the section “Creating and Loading Traces” in the Tracealyzer User Manual and see the getting started guides at https://percepio.com/tracealyzer/gettingstarted/.

Percepio® TraceRecorder Integration Guide. Copyright © Percepio AB, 2023
https://percepio.com

1.1 Assumptions

- The tracing is done using software instrumentation that calls the TraceRecorder library.
- If needed, the target RTOS can be modified to call trace instrumentation functions.
- There is a communication interface from the target system to the host computer, and with sufficient throughput if continuous trace streaming is desired. For example, a network connection or debug probe supported by Tracealyzer (e.g. SEGGER J-Link).

1.2 Terminology

- Actor: An execution context, such as a task, thread or interrupt handler.
- Actor Instance: A single execution occurrence of an actor, e.g. one iteration of a task loop.
- Object: A unique target-side software object that is referenced in the trace data, such as an actor, message queue, mutex, semaphore, etc. It has a location in memory and a lifetime.
- Object Handle: An integer value that uniquely identifies an Object.
- Task: A target-side thread, typically running application code.

Percepio® TraceRecorder Integration Guide. Copyright © Percepio AB, 2023
https://percepio.com

1.3 TraceRecorder

TraceRecorder is a C library that is included in the target system and responsible for timestamping, encoding, buffering and transmission of software event data, intended for transfer to the host PC and visualization in Tracealyzer.

The TraceRecorder library is typically called from instrumentation/hooks in an RTOS kernel or other instrumented API, or by adding explicit logging calls in the application code.

An event consists of a timestamp, an event code, and optionally one or several event parameters. The parameters depend on the event code. The meaning of the event codes is defined in host-side XML files, described later in this guide.

The TraceRecorder library can be used in two ways. Either for continuous streaming to host, or for snapshot tracing. The latter means that the recorder writes the data to a RAM buffer from where "snapshots" can be saved/uploaded at any point. In this approach, the trace length is limited by the size of the RAM buffer.

The recorder consists of a generic part that is the same for all kernel integrations, and various “port” modules that provide specific integrations:

- Kernel ports: define the instrumentation code that is added in the software platform to call the TraceRecorder. Percepio provides a number of predefined kernel ports, including a “bare metal” kernel port with can be extended as described in this guide.
- Hardware ports: provides processor-specific definitions, e.g. for timestamping.
- Stream ports: provide definitions for trace output to host and for receiving commands from host (start/stop commands)

While the TraceRecorder library is provided as open source software (Apache 2.0 license), changing the generic parts of the code may break compatibility with the Tracealyzer application. If you have ideas for improvements, please contact support@percepio.com.

Note that the TraceRecorder library includes two separate implementations, for historical reasons called "Snapshot mode" and "Streaming mode". These names are no longer accurate, as the newer “streaming mode” also support snapshots (using the RingBuffer stream port). The modes are more accurately two different implementations and data formats. The “streaming mode” is the default setting. The older "snapshot mode" is deprecated and not covered by this porting guide.

The steps required for porting the TraceRecorder to a new RTOS kernel are as follows:

- Processor-specific definitions
- Enabling basic kernel tracing
- Tracing API calls
- Defining a stream port
- Host-side XML files

These aspects are covered by the following sections.

percepio
SENSING SOFTWARE

2. Processor-specific definitions

Tracealyzer is not very dependent on the hardware since it relies on software instrumentation rather than hardware-specific trace functions, but there are still a few hardware dependencies. These are defined as "hardware ports" in trcHardwarePort.h. Each hardware port contains a set of definitions intended for a specific processor architecture/family.

To use an existing hardware port, set TRC_CFG_HARDWARE_PORT in *trcConfig.h* to the appropriate hardware port setting.

In case a suitable hardware port is not already available in *trcHardwarePort.h*, the user needs to define a custom hardware port. This is a matter of defining a few macros, mostly about timestamping. A high-level description is provided at https://percepio.com/devblog-adapting-tracealyzer-nios-ii/, and this section provides more details.

2.1 Hardware ports

Tracealyzer needs accurate timestamps on the events, preferably on microsecond level or better. The default resolution in the Tracealyzer views is 1 µs, but internally Tracealyzer uses the native timestamps stored by the TraceRecorder library. They can be of any resolution. However, if using a very low resolution (e.g. 1 ms) some events may seem to occur at the same timestamp.

The timestamping is done internally in the recorder's event functions, based on the TRC_HWTC macros. To make a new hardware port you need to define these macros. This is documented extensively in *trcHardwarePort.h*, which also provides plenty of examples for other processor families.

See also *3.1 Protecting critical sections*, as you may need to update this as well.

2.1.1. Timestamps

The timestamps are read using the TRC_HWTC_COUNT macro and stored in the events as 32-bit integers. The TRC_HWTC_COUNT macro should ideally sample a free-running counter in the processor, such as clock cycle counter (if the clock frequency is fixed). An alternative time source is the internal counter of interrupt timers (e.g. for the OS tick interrupt). The TRC_HWTC_TYPE definition tells Tracealyzer what kind of time source that is used, so timestamps can be interpreted correctly.

Note that wrap-arounds (0xFFFFFFFF → 0x00000000) are handled automatically by Tracealyzer, as long as there is at least one event in between each wrap-around.

For Tracealyzer to translate these timestamps into microseconds, milliseconds, seconds, minutes, etc., you also need to define TRC_HWTC_FREQ_HZ. This should be the clock rate of the TRC_HWTC_COUNT counter in Hz.

Full documentation about these macros is available in *trcHardwarePort.h*.

2.1.2 Meaning of interrupt priority levels

The *TraceRecorder* allows for tracing interrupt handlers by calling xTraceISRBegin() and xTraceISREnd(). You first need to store a name and the IRQ priority level of the traced interrupt handler using xTraceISRRegister().

The meaning of IRQ priority levels however differs between different processor families. For some processors, IRQ priority level 0 indicates the most urgent priority level, and for other processors it means the opposite.

To ensure that Tracealyzer understands your IRQ priorities correctly, make sure the TRC_IRQ_PRIORITY_ORDER definition is set correctly in *trcHardwarePort.h*. This should be 0 if lower IRQ priority values imply highest (most urgent) priority, otherwise 1.

3. Platform support

This section describes how to integrate the TraceRecorder library with an RTOS kernel or similar software platform. This requires protecting critical sections in the TraceRecorder, registering objects and events, and providing matching XML files to the Tracealyzer application.

3.1 Protecting critical sections

The recorder contains several critical sections where the recorder must not be preempted by other recorder calls on the same core, for example by an OS tick event, a task-switch or a traced interrupt handler.

The critical sections inside the recorder library are surrounded by macros, intended to provide protection against nested recorder calls:

- TRACE_ALLOC_CRITICAL_SECTION: Allows for allocating a local variable.
- TRACE_ENTER_CRITICAL_SECTION: Protect the following critical section.
- TRACE_EXIT_CRITICAL_SECTION: Restore state after the critical section.

An example of how the recorder uses these macros is shown below.

```
void TraceRecorderFunction(...)
{
   TRACE_ALLOC_CRITICAL_SECTION();  /* Allocate a local variable for the interrupt mask*/
   TRACE_ENTER_CRITICAL_SECTION();  /* Store the interrupt mask and disable interrupt */
   /* The critical section */
   TRACE_EXIT_CRITICAL_SECTION();   /* Reapply the previous interrupt mask */
}
```

Defining these macros is a fundamental part of porting the TraceRecorder library. These macros are normally defined in trcKernelPort.h, as they might rely on RTOS-specific functions.

Note: This section is only applicable if you are using an RTOS or similar with pre-emptive scheduling, or are doing tracing in interrupt handlers. If nested recorder calls cannot occur, the TRACE_..._CRITICAL_SECTION definitions can be empty.

Note that the recorder functions might be called both from task-level and interrupt-level so the critical section macros must work in both cases. This is not always the case for the default critical sections provided in your RTOS. On some RTOSes you need to use different functions depending on the context (Task/ISR). In that case, you need a condition inside the ENTER and EXIT macros. An example is shown below.

```
#define TRACE_ENTER_CRITICAL_SECTION() critical_section_begin()

void critical_section_begin(void)
{
    if (in_task_context()) /* Processor-specific condition, must be provided */
    {
        /* Protect critical section, on task level */
    }
    else
    {
        /* Protect critical section during startup (or ISR, if nested ISRs are allowed) */
    }
}
```

Do not attempt to use a mutex or semaphore for protecting these critical sections, as they may generate additional trace events causing recursive TraceRecorder calls.

Percepio® TraceRecorder Integration Guide. Copyright © Percepio AB, 2023
https://percepio.com

percepio
SENSING SOFTWARE

3.1.1. Method A: Disabling interrupts

To avoid nested recorder calls, the safest (and easiest) approach is to disable interrupts during the recorder's critical sections. Disabling interrupts is generally not advised in real-time systems, but the critical sections of the recorder library are typically very short. Moreover, depending on the processor used, it is not required to disable all interrupts, as described later. However, by disabling ALL interrupts in the critical sections you can safely generate trace events from any part of your code, including interrupt handlers. An example of such a solution for Arm Cortex-M devices is shown below (using Arm's CMSIS library):

```
#define TRACE_ALLOC_CRITICAL_SECTION() uint32_t TRACE_ALLOC_CRITICAL_SECTION_NAME;
#define TRACE_ENTER_CRITICAL_SECTION() {TRACE_ALLOC_CRITICAL_SECTION_NAME = __get_PRIMASK(); __set_PRIMASK(1);}
#define TRACE_EXIT_CRITICAL_SECTION() {__set_PRIMASK(TRACE_ALLOC_CRITICAL_SECTION_NAME);}
```

The ENTER macro should always disable all interrupts where the TraceRecorder library is used, either due to direct calls or indirectly by calls to traced functions. It should also save the previous interrupt mask. The EXIT macro should not enable interrupts unconditionally, but instead restore the interrupt mask that was saved in the ENTER macro. Otherwise, you might enable interrupts too early.

To avoid that timing-critical interrupts are delayed by the recorder's critical sections, depending on the processor used you may choose to only disable lower priority interrupts while leaving higher priority (and time critical) interrupts enabled. On Arm Cortex-M3/M4/M7 this is achieved using the BASEPRI register. If using this approach, the higher-priority (i.e., not disabled) interrupt handlers are not allowed to call the TraceRecorder library.

3.1.2 Method B: Disabling the kernel scheduler

Assuming TraceRecorder is only used in the scheduler and in tasks on a single core, the critical sections only need to disable the task switching.

This can be achieved by calling a kernel function that temporarily disables the scheduler, which is available in many RTOS kernels. This typically leaves the OS tick interrupt enabled, but instead sets a software flag in the RTOS kernel that prevents context-switching from taking place. If using this approach, you need to ensure that no other trace event is generated by the OS tick interrupt when the scheduler is disabled (e.g., "task ready" or "OS tick" events).

Note: if using this approach, it is not safe to call the recorder library from interrupt handler other than those managed by the kernel's scheduler. Kernel interrupts may remain active since assumed to not trigger any traced kernel operations when the kernel scheduler is disabled.

3.2 Adding trace instrumentation

Trace instrumentation is the code that calls the TraceRecorder library on relevant events, such as task-switches and API calls. This feeds information into the TraceRecorder library, that takes care of the rest.

Tracealyzer is very flexible and does not specify what events that should be traced or what information to include for each event, with a few exceptions presented later. Generally, it is up to developer that is porting the recorder library to select what information to include.

The provided information needs to be specified in an XML file for the Tracealyzer application, so it can make sense of the data. It is important that the XML file matches the provided data in every case. The XML file format is presented later in this document.

We generally recommend instrumenting the following types of events, in priority order:

- Priority 1: core kernel scheduling events, especially:
    - Task-switches (when changing the running task)

  - Actor ready (when tasks/threads become ready to execute)
- Priority 2: RTOS API calls, especially:
  - Communication and synchronization (semaphores, mutexes, etc.)
  - Operations affecting the kernel scheduling (priority changes, delays, etc.)
  - Dynamic memory allocation, if relevant (malloc/free, memory block pools, etc.)
- Priority 3: Middleware API calls (e.g. networking and I/O stacks), especially:
  - Initialization (create, open, etc.)
  - Data output (write, put, send, etc.)
  - Data input (read, get, receive, etc.)

3.2.1 Object Handles

Most kernel events are operations on a kernel object, such as activating a thread or locking a mutex. Such events should include an Object Handle that uniquely identifies the object.

On processors using physical memory addressing (e.g. microcontrollers) the memory address of the object can typically be used as handle. The object address is registered as the Object Handle when the object is created/initialized by calling xTraceObjectRegister or xTraceObjectRegisterWithoutHandle, as explained in later sections.

Example:

```
void MutexInitalize(mutex_t* mutex, char* name) {
		...
		(void)xTraceObectRegisterWithoutHandle(PSF_EVENT_MUTEX_CREATE, mutex, name, 0);
		...
}

void MutexLock(mutex_t* mutex) {
		...
		(void)xTraceEventCreate1(PSF_EVENT_MUTEX_LOCK, mutex); // Address used as handle
		...
}
```

If kernel objects can be created and deleted in runtime (dynamic allocation) the Object Handle should be unregistered at the delete event using xTraceObjectUnregisterWithoutHandle.

On systems using virtual memory and multiple processes with separate address spaces, it is not advised to use virtual addresses as Object Handles. This could lead to incorrect results if two different objects in different processes would get the same virtual addresses and Object Handles by chance. This is at least the case as of Tracealyzer v4.8, but this might be resolved in future versions.

Another alternative is to assign an explicit identifier for each created object that can be provided as Object Handle in TraceRecorder calls. Such identifiers should be a non-zero integer. This can be stored as a field within the object data structure for easy access in the instrumentation code.

Example:

```
void MutexInitalize(mutex_t* mutex, char* name) {
	mutex->handle = GetNextAvailableObjectHandle();
	(void)xTraceObjectRegisterWithoutHandle(PSF_EVENT_MUTEX_CREATE, mutex->handle, name, 0);
}

void MutexLock(mutex_t* mutex) {
	(void)xTraceEventCreate1(PSF_EVENT_MUTEX_LOCK, mutex->handle); // Explicit handle
}
```

3.2.2 The TraceRecorder API

The following TraceRecorder functions are important for tracing kernel events:

```
traceResult xTraceObjectRegisterWithoutHandle(uint32_t uiEventCode, void* pvObject, const char* szName, TraceUnsignedBaseType_t uxState)
```

This function is used to register an object in the trace, such as tasks, queues, semaphores etc. This results in "create" events, where the event code is provided by the parameter uiEventCode. This event code must be defined in the XML file. The parameter pvObject specifies the object handle and uxState the object state (see below).

```
traceResult xTraceObjectUnregisterWithoutHandle(uint32_t uiEventCode, void* pvObject, TraceUnsignedBaseType_t uxState)
```

Unregisters an object from the trace when it is deleted/deallocated.

```
traceResult xTraceObjectSetNameWithoutHandle(void* pvObject, const char* szName)
```

Sets an object's name for display in Tracealyzer.

```
traceResult xTraceObjectSetStateWithoutHandle(void* pvObject, TraceUnsignedBaseType_t  uxState)
```

This function is used to set the current *state attribute* of an object. The meaning depends on the object type.

- *Tasks*: the scheduling priority
- *Queues*: the current number of items in the queue
- *Mutexes*: the object handle of the current owner (or 0 if the mutex is not locked)
- *Semaphores*: the *semaphore* counter

```
traceResult xTraceTaskSwitch(void* pvTask, TraceUnsignedBaseType_t uxPriority)
```

This function registers a context switch, where the kernel changes the executing actor. The parameters refer to the actor that begins to execute, where pvTask is the task ID (i.e. the PCB address) and uxPriority is the current scheduling priority.

```
traceResult xTraceTaskReady(void* pvTask)
```

This function registers an Actor Ready event, meaning that Actor pvTask is now ready to execute.

```
traceResult xTraceEventCreate0(uint32_t uiEventCode)
```

This function creates an event without any additional arguments.

```
traceResult xTraceEventCreate1(uint32_t uiEventCode, TraceUnsignedBaseType_t uxParam1)
```

This function creates an event with one argument.

```
traceResult xTraceEventCreate2(uint32_t uiEventCode, TraceUnsignedBaseType_t uxParam1, TraceUnsignedBaseType_t uxParam2)
```

This function creates an event with two arguments.

```
traceResult xTraceEventCreate3(uint32_t uiEventCode, TraceUnsignedBaseType_t uxParam1, TraceUnsignedBaseType_t uxParam2, TraceUnsignedBaseType_t uxParam3)
```

This function creates an event with three arguments.

```
traceResult xTraceEventCreate4(uint32_t uiEventCode, TraceUnsignedBaseType_t uxParam1, TraceUnsignedBaseType_t uxParam2, TraceUnsignedBaseType_t uxParam3, TraceUnsignedBaseType_t uxParam4)
```

This function creates an event with four arguments.

```
traceResult xTraceEventCreate5(uint32_t uiEventCode, TraceUnsignedBaseType_t uxParam1, TraceUnsignedBaseType_t uxParam2, TraceUnsignedBaseType_t uxParam3, TraceUnsignedBaseType_t uxParam4, TraceUnsignedBaseType_t uxParam5)
```

This function creates an event with five arguments.

```
traceResult xTraceEventCreate6(uint32_t uiEventCode, TraceUnsignedBaseType_t uxParam1, TraceUnsignedBaseType_t uxParam2, TraceUnsignedBaseType_t uxParam3, TraceUnsignedBaseType_t uxParam4, TraceUnsignedBaseType_t uxParam5, TraceUnsignedBaseType_t uxParam6)
```

This function creates an event with six arguments.

Note that the data types ending with ...BaseType_t depends on definitions in trcTypes.h. They are by default mapped to 32-bit integer data types (int32_t and uint32_t), but can be overridden by defining TRC_BASE_TYPE and TRC_UNSIGNED_BASE_TYPE, e.g. -DTRC_BASE_TYPE=int64_t.

TraceRecorder handles all parameters as unsigned. To store signed integers, cast them to unsigned when calling TraceRecorder and then specify a signed data type in the XML configuration file introduced in Section 3.3. This is specified in the *Param* element using the *Type* attribute. See Appendix B for details.

Percepio® TraceRecorder Integration Guide. Copyright © Percepio AB, 2023
https://percepio.com

3.2.3 Trace Hooks and Events

The integration between the RTOS kernel code and the recorder library can is typically done by adding trace hooks, that call the TraceRecorder when enabled but are inactive when tracing is disabled. This can be done in several ways, but the easiest and most efficient is probably by using preprocessor function macros, e.g. #define TP(args). Using such “trace macros”, the tracing code can be completely disabled by switching between different definitions of the trace macros. A set of empty trace macro definitions is used by default, not producing any code. The user can then toggle #define USE_TRACEALYZER (or similar) to enable or disable the tracing.

```
#if !(USE_TRACEALYZER)
 #define tpTASK_SWITCH(taskId, taskPriority)
#else
 #define tpTASK_SWITCH(taskId, taskPriority) xTraceTaskSwitch(taskId, taskPriority)
#endif
```

In the above examples, “taskId” can be the address of the task control block, or some other numeric identifier of the object that serves as the Object Handle. Finally, the current task priority is provided. This results in a task-switch event.

Events are identified by their *event code*, an integer value between 0–4095. For standard events like *task switch*, an *event code* has already been defined in *trcKernelPort.h*. When adding additional events you need to add a unique *event code* for each event.

Custom events are created using xTraceEventCreate0..6(), where the 0..6 reflects the number of parameters to the event after the event code parameter.

```
void tp_myevent(uint16_t x, uint16_t y) {

		(void)xTraceEventCreate2(MYEVENTCODE, x, y);

		return;
}
```

The trace macros are defined in *trcKernelPort.h*, but you should not include this header file directly in your code. Include the main header file instead, *trcRecorder.h*.

A simple way to make the trace macros available in the RTOS kernel code is to include *trcRecorder.h* in a central RTOS header file, or a kernel configuration header file.

We provide a minimal version of *trcKernelPort.h* that can be used as starting point (the “bare metal” kernel port). The Zephyr kernel port is quite easy to follow and can be used as reference.

3.2.4 Basic Kernel Tracing – Minimal Example

In order to have tasks and other kernel objects appear in the trace they must be properly registered using xTraceObjectRegisterWithoutHandle().

We create a trace point macro that will be referenced in the kernel where task creation is performed.

```
#define tpTASK_CREATE(task, name, priority) \
              xTraceObjectRegisterWithoutHandle(PSF_EVENT_TASK_CREATE, task, name, priority)
```

We then add `tpTASK_CREATE(pxNewTask, szName, priority)` in the code where new tasks are created.

The below example shows a minimal trace with just two actors, “IDLE” and “MyTask”.

```
#define tpTASK_SWITCH(task, priority) xTraceTaskSwitch(task, task->priority)

Timestamp =   0: tpTASK_SWITCH(MyTask);
Timestamp = 100: tpTASK_SWITCH(IDLE);
Timestamp = 500: tpTASK_SWITCH(MyTask);
Timestamp = 600: tpTASK_SWITCH(IDLE);
```

Note that the first and second parameters normally would be provided by reading the “current task” pointer (e.g., ptrCurrentTask and ptrCurrentTask->priority). The MyTask and IDLE symbols are only used to illustrate the data.

The result can be seen in *Figure 1*.

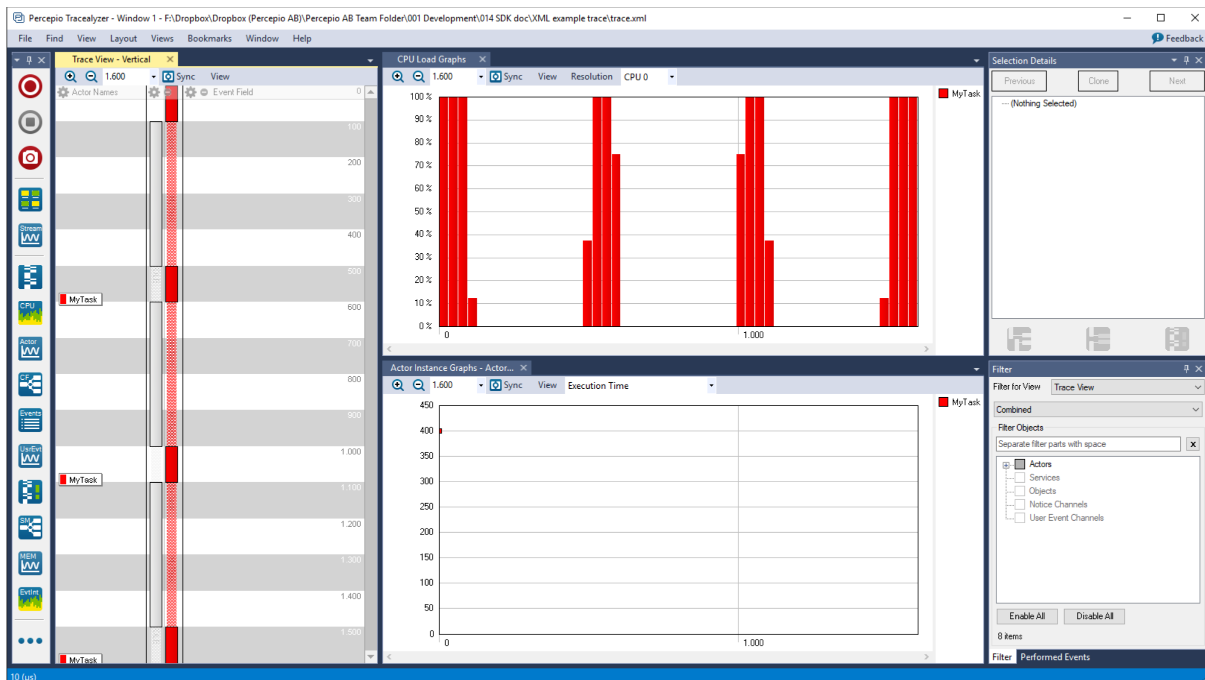


Figure 1. Tracealyzer showing the minimal trace example.

The solid rectangles (“fragments”) show where MyTask is executing. However, note the cross-hatched regions that connect the fragments. This indicates that the actor is still active but not executing. This is normally used when a task is preempted by another task but is still ready to execute.

When an actor is inactive, e.g. waiting for the next input or the next activation time, it is recommended to end instance. This will split the execution into multiple “actor instances” (jobs), where each instance becomes a separate data point in the Tracealyzer statistics. This is done automatically by Tracealyzer if you provide “actor ready” events, which should be generated when tasks become ready to execute (active) from a previous dormant (inactive) state. An extended example is shown below, additions in green.

```
#define tpTASK_READY(task) xTraceTaskReady(task)

T =   0: tpTASK_READY(MyTask);
T =   0: tpTASK_SWITCH(MyTask);
T = 100: tpTASK_SWITCH(IDLE);
T = 480: tpTASK_READY(MyTask);
T = 500: tpTASK_SWITCH(MyTask);
T = 600: tpTASK_SWITCH(IDLE);
```

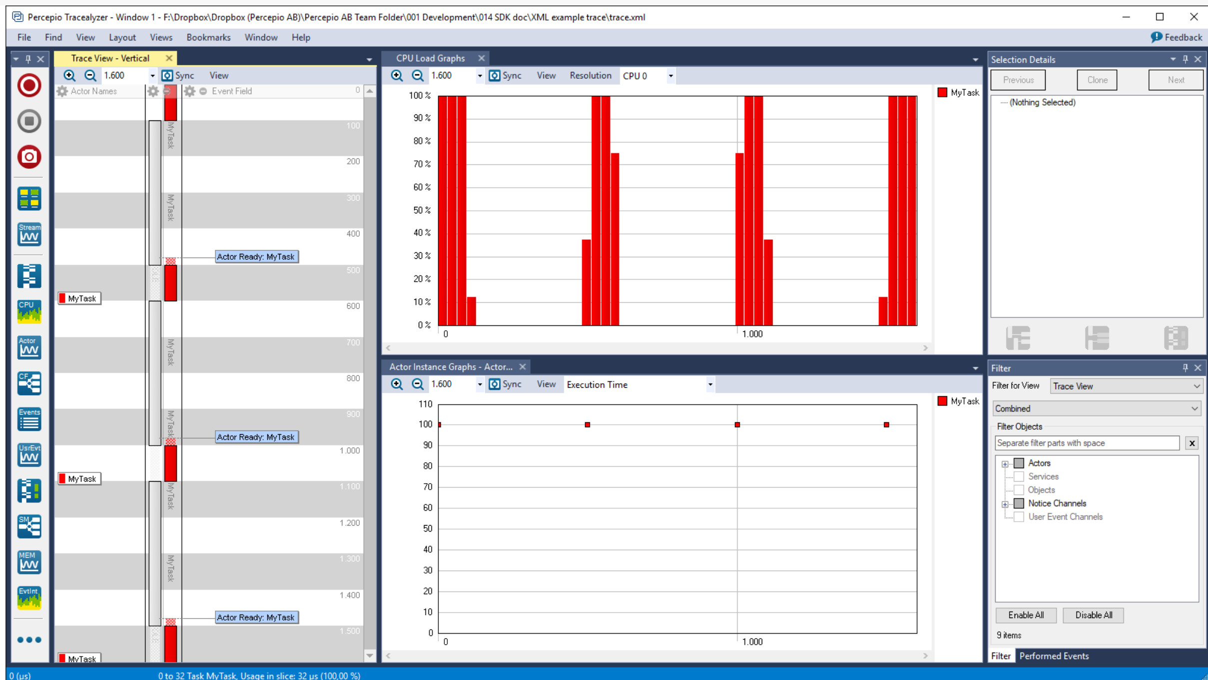


Figure 2. Tracealyzer showing context-switch and actor ready events.

Note the difference compared to *Figure 1*. The cross-hatched regions are now limited to a brief period just before the actor begins to execute (from “Ready” to “Switch”) and the execution of MyTask has been divided into multiple actor instances. This way, the timing metrics shown in Tracealyzer will reflect the actor instances (jobs) correctly.

To summarize this section, the instrumentation needed for a basic task-switch trace is:

- Register Actors (tasks) on creation by calling xTraceObjectRegisterWithoutHandle();
- Actor ready event: When an Actor becomes ready to execute;
- Task-switch event: When an Actor is switched in and starts executing.

3.3 Host-side XML file for basic kernel tracing

After having implemented the basic kernel support described in section 3, you should be able to record a trace in the proper format. However, for Tracealyzer to understand the trace data it needs an XML file on the host side. This section presents the XML structure.

The configuration name is defined in *trcKernelPort.h* by *TRC_PLATFORM_CFG*. The binary format has a version number, indicated by *TRC_PLATFORM_CFG_MAJOR*, *TRC_PLATFORM_CFG_MINOR* and *TRC_PLATFORM_CFG_PATCH*.

Example:

```
#define TRC_PLATFORM_CFG "MY_KRNL"
#define TRC_PLATFORM_CFG_MAJOR 1
#define TRC_PLATFORM_CFG_MINOR 2
#define TRC_PLATFORM_CFG_PATCH 3
```

Given the above attributes, Tracealyzer will assume the XML file is called “`MY_KRNL-v1.2.3.xml`”.

When Tracealyzer opens the trace file, it will find this information and search a list of directories for the matching XML file.

As of Tracealyzer v4.8 you can edit the directories to search for XML files in the Tracealyzer settings (File -> Settings -> Definition File Paths).

In Tracealyzer v4.7, the following directories are searched:

- The same directory as the trace file.
- My Documents/Tracealyzer/cfg (if using Windows)
- /Tracealyzer/cfg (if using Linux)
- Any path provided in the TZ_CFG_PATH environment variable. Multiple paths should be separated by semicolons.

3.3.1 Basic XML configuration

The first step when Tracealyzer parses the trace data is to translate the event codes to known event types. This mapping is provided by the configuration XML file. Below is a minimal configuration, sufficient for the previous example in *Figure 2*.

```xml
<?xml version="1.0" encoding="utf-8"?>
<PlatformExtension>
  <EventCodes>
    <EventGroup name="Core">
      <Event code="0x01" type="TraceStart" />
      <Event code="0x02" type="TimestampConfig" />
      <Event code="0x03" type="ObjectName" />
      <Event code="0x06" type="TaskReady" />
      <Event code="0x07" type="SwitchToTask"/>
    </EventGroup>
    <EventGroup name="Task">
      <Event code="0x30" type="ObjectState">
        <Param index="0" type="Handle" class="Task"/>
        <Param index="1" type="Int32" useAs="StateAfter"/>
        <FollowWith key="TaskCreate" />
      </Event>
      <Event key="TaskCreate" service="Task/xTaskCreate" type="KernelServiceReturn" status="StatusOK">
        <Param index="0" type="Handle" class="Task" useAs="Arg"/>
        <Param index="1" type="UInt32" useAs="Arg"/>
      </Event>
    </EventGroup>
  </EventCodes>
  <TargetPlatform>
    <KernelServiceGroups>
      <KernelServiceGroup name="Task">
        <KernelService name="xTaskCreate" operation="Initialize">
          <Parameter name="task" format="Task {0}" />
          <Parameter name="priority"/>
        </KernelService>
      </KernelServiceGroup>
    </KernelServiceGroups>
    <ObjectClasses>
      <ObjectClass name="Task" type="Actor"/>
      <ObjectClass name="ISR" type="Actor"/>
    </ObjectClasses>
    <TaskPriorityDirection>HigherNumberIsMoreImportant</TaskPriorityDirection>
  </TargetPlatform>
</PlatformExtension>
```

The meaning of the XML elements and attributes definitions in the above example are outlined below. A more comprehensive explanation of some details follow after this section.

- `EventGroup`: A set of related event codes
  - `Event`: A specific event code
    - `code`: The event code provided by the instrumentation to the TraceRecorder
    - `key`: (optional) A unique identifier that can be referenced by *FollowWith*
    - `type`: The event type. For a full list of possible values see *Appendix B*.
      - `TraceStart`: Generated automatically when the recorder library begins recording events
      - `TimestampConfig`: Generated automatically when the recording begins
      - `ObjectName`: Event type that sets the name of an object
      - `ObjectState`: Updates the object data attribute (a.k.a. object state, such as task priority)
      - `TaskReady`: When tasks become ready to execute from a dormant state.
      - `SwitchToTask`: When the executing task is changed
      - `KernelServiceReturn`: Kernel API function call

- `status`: The status of the event. For a full list of possible values see *Appendix B*.
  - `StatusOK`: Success
- `Param`: A parameter. Note that the target decides how many parameters are sent. The host just references them. Unreferenced parameters are simply ignored.
  - `index`: Zero-based index of the parameter.
  - `type`: For a full list of possible values see *Appendix B*.
    - `Handle`: An object handle, i.e., the ID of an object
    - `UInt32`: A 32-bit unsigned integer
  - `class`: The *ObjectClass* being referenced by the *Handle* parameter type
  - `useAs`: What to do with the parameter value. For a full list of possible values see *Appendix B*.
    - `Arg`: Will be shown in the Trace Views
    - `StateAfter`: Changes the state of the object (referenced by the *Handle*) to this value
- `FollowWith`: (optional) Process additional definitions, in the element indicated by *key*.

3.3.2 Definition of event codes

The event codes are typically defined in *trcKernelPort.h* using #define statements. They must match the XML file definitions. The allowed range is 0–4095.

An example is shown below, the core events from the FreeRTOS version of *trcKernelPort.h*:

```
    #define PSF_EVENT_TRACE_START                              0x01  /* TraceStart */
    #define PSF_EVENT_TS_CONFIG                                0x02  /* TimestampConfig */
    #define PSF_EVENT_OBJ_NAME                                 0x03  /* ObjectName */
    #define PSF_EVENT_TASK_READY                               0x30  /* TaskReady */
    #define PSF_EVENT_TASK_ACTIVATE                            0x37  /* SwitchToTask */
```

3.3.3 Object data

The above example only contains one custom event, from an example kernel function called *xTaskCreate* used for creating new tasks. On “create” events it is typically desired to do two things in Tracealyzer: (1) create a new object including the *Object Data* attribute, and (2) create a “kernel call event” that makes the xTaskCreate-call visible in the trace as an event label.

The Object Data is a generic attribute found in all Tracealyzer objects, that is used for different purposes depending on the object type.

For Actor objects such as tasks, the Object Data should contain the scheduling priority. In the example, this is set using the ObjectState event type. This creates a new Actor object and sets the Object Data to the provided task priority. This is the relevant code from the example:

```
<Event code="0x30" type="ObjectState">
  <Param index="0" type="Handle" class="Task"/>
  <Param index="1" type="Int32" useAs="StateAfter"/>
  <FollowWith key="TaskCreate" />
</Event>
```

The first parameter has the type “Handle”, i.e. the Object Handle/ID of the object. The second parameter is the scheduling priority that we map to “StateAfter”, meaning this value applies from this event until the next update. Note that “Object State” an older name for “Object Data”, which is still reflected in the XML format.

The FollowWith element is used to add a second action on this event, as explained below.

In some cases, you may want to use the recorder function xTraceObjectSetStateWithoutHandle() to store the object data explicitly. This is stored in a table within the TraceRecorder library, which is included when a trace session is started. This way, Tracealyzer is informed about the initial state of all objects, even if they were created before the tracing started.

3.3.4 FollowWith

In the example, you may note the FollowWith element on event code 0x30.

```
<EventGroup name="Task">
  <Event code="0x30" type="ObjectState">
    <Param index="0" type="Handle" class="Task"/>
    <Param index="1" type="Int32" useAs="StateAfter"/>
    <FollowWith key="TaskCreate" />
  </Event>
  <Event key="TaskCreate" service="Task/xTaskCreate" type="KernelServiceReturn" status="StatusOK">
    <Param index="0" type="Handle" class="Task" useAs="Arg"/>
    <Param index="1" type="UInt32" useAs="Arg"/>
  </Event>
</EventGroup>
```

FollowWith is a mechanism for Tracealyzer to perform multiple actions on a single event. Internally it allows an event to be handled by multiple event handlers. For example, a single event could perform a service call, like WakeOtherTask(), then switch to a that task, then perhaps even do something else.

In this example, it is used to first update the Object Data and then create a kernel service call entry that is displayed in the trace. This mechanism can also be used for similar purposes on delete/close API calls, i.e., (1) display the event and then (2) apply the delete/close action inside Tracealyzer.

Events referenced by FollowWith don't have to be in the same event group.

3.4 Kernel services

To show traced API calls as event labels, they need to be mapped to a *KernelService*.

```
<EventGroup name="Task">
  <Event code="0x30" type="ObjectState">
    <Param index="0" type="Handle" class="Task"/>
    <Param index="1" type="Int32" useAs="StateAfter"/>
    <FollowWith key="TaskCreate" />
  </Event>
  <Event key="TaskCreate" service="Task/xTaskCreate" type="KernelServiceReturn" status="StatusOK">
    <Param index="0" type="Handle" class="Task" useAs="Arg"/>
    <Param index="1" type="UInt32" useAs="Arg"/>
  </Event>
</EventGroup>
```

In the above example, we provide such a *KernelService* definition for event code 0x30. This will display a "xTaskCreate" event in Tracealyzer as a "kernel service" call.

The *service* attribute is used to link this definition to additional information in the *TargetPlatform* structure, specifying higher level information such as the meaning of the event and how to present it.

The type *KernelServiceReturn* is used for events at the return of kernel services. It may optionally be combined with *KernelServiceEntry* events to capture both the entry and return of potentially blocking events. This way, Tracealyzer becomes aware of the blocking time and displays it in various ways, for example as red/green labels in the trace view and in the *Service Call Block Time* view.

For potentially blocking kernel functions, every *KernelServiceEntry* event must have a matching *KernelServiceReturn*.

For non-blocking kernel functions, only a *KernelServiceReturn* event is needed.

3.4.1 TargetPlatform

After the event types have been understood by Tracealyzer, additional semantic information about the events is often needed to fully leverage all functionality in Tracealyzer. By mapping the events to "operations" understood by Tracealyzer, and the Object Classes to predefined Tracealyzer types, Tracealyzer will automatically include these events and objects in all relevant views. This is provided in the *TargetPlatform* section of the XML file. The TargetPlatform definitions also allow for custom formatting of the event labels.

From our initial example:

```
<TargetPlatform>
```

Percepio® TraceRecorder Integration Guide. Copyright © Percepio AB, 2023
https://percepio.com

```
  <KernelServiceGroups>
    <KernelServiceGroup name="Task">
      <KernelService name="xTaskCreate" operation="Initialize">
        <Parameter name="task" format="Task {0}" />
        <Parameter name="priority"/>
      </KernelService>
    </KernelServiceGroup>
  </KernelServiceGroups>
  <ObjectClasses>
    <ObjectClass name="Task" type="Actor"/>
    <ObjectClass name="ISR" type="Actor"/>
  </ObjectClasses>
  <TaskPriorityDirection>HigherNumberIsMoreImportant</TaskPriorityDirection>
</TargetPlatform>
```

The meaning of these definitions are:

- KernelServiceGroups: Container for all *KernelServiceGroup* nodes
  - KernelServiceGroup: A set of related *Kernel Services*
    - KernelService: A specific *Kernel Service*
      - name: The *Kernel Service* name
      - operation: What the kernel service does. For a full list of possible *ObjectClass* values, see *Appendix A*.
        - Initialize: The kernel service is creating and initializing an object (an actor or other object)
      - Parameter: A parameter to the *Kernel Service.* More details are found in *Appendix A*.
        - name: The parameter name (normally not displayed)
        - format: How the parameter should be displayed
- ObjectClasses: Provides a list of *ObjectClass* definitions.
  - ObjectClass: Provides a platform-specific type name for a class of objects (e.g. “OSMutex_t”) and maps this to a predefined ObjectClass type understood by Tracealyzer (e.g. Mutex). For a full list of supported *ObjectClass* types, see *Appendix A*.
- TaskPriorityDirection: indicates the meaning of task priority values. Possible values are:
  - *HigherNumberIsMoreImportant*
  - *LowerNumberIsMoreImportant*

4. Tracing Kernel Service/API calls

Tracealyzer provides a lot of functionality related to RTOS service calls and Middleware API calls, such as operations on message queues, semaphores, mutexes, sockets, dynamic memory allocation etc. Such events are traced in the same way in the section 3, i.e., when the function is called, you insert a call to one of the API functions with a unique event code and relevant arguments. The difference is in the host-side XML files.

In the following example we will extend the basic example from earlier with tracing of two RTOS service calls, MUTEX_Lock and MUTEX_Release, each generating a single event. These events have a single argument, the Object Handle (address) of the affected Mutex object.

The result is seen in *Figure 3*, where we can now see events like `MUTEX_Lock(MUTEX1)` and also the dependency between MUTEX1 and MyTask in the Communication Flow graph. This dependency is understood automatically by Tracealyzer thanks to the XML configuration.

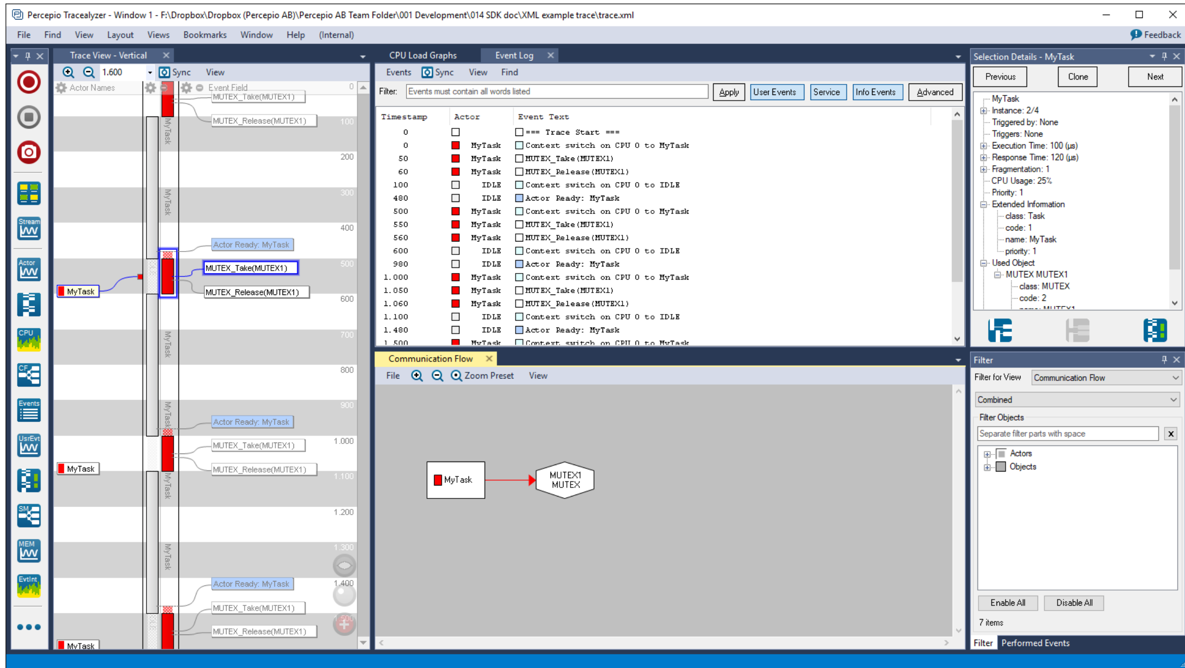


Figure 3. Tracealyzer showing context-switch, actor ready events and Mutex events.

To trace these events using the TraceRecorder API, we define the following trace hooks that are called from suitable locations within the mutex functions.

```
#define tpMUTEX_CREATE(mutex, name) \
              xTraceObjectRegisterWithoutHandle(PSF_EVENT_MUTEX_CREATE, mutex, name, 0);

#define tpMUTEX_LOCK(mutex) tp_mutex_lock(mutex)

void tp_mutex_lock(void* mutex) {
              (void)xTraceEventCreate1(PSF_EVENT_MUTEX_LOCK, mutex);
}

#define tpMUTEX_RELEASE(mutex) tp_mutex_release(mutex)

void tp_mutex_release(void* mutex) {
              (void)xTraceEventCreate1(PSF_EVENT_MUTEX_RELEASE, mutex);
}
```

The corresponding event tracing calls are illustrated below (additions in green).

```
T =   0: tpTASK_READY(MyTask);
T =   0: tpTASK_SWITCH(MyTask, MyTask->priority);
T =  50: tpMUTEX_LOCK(MyMutex);
T =  60: tpMUTEX_RELEASE(MyMutex);
T = 100: tpTASK_SWITCH(IDLE, IDLE->priority);
T = 480: tpTASK_READY(MyTask);
T = 500: tpTASK_SWITCH(MyTask, MyTask->priority);
T = 550: tpMUTEX_LOCK(MyMutex);
T = 560: tpMUTEX_RELEASE(MyMutex);
T = 600: tpTASK_SWITCH(IDLE, IDLE->priority);
```

4.1 Updated XML configuration

To show the added events properly in Tracealyzer, we need to update the XML file accordingly. The additions are shown in green.

```
<?xml version="1.0" encoding="utf-8"?>
< PlatformExtension>
  < EventCodes>
    <EventGroup name="Core">
      <Event code="0x01" type="TraceStart" />
      <Event code="0x02" type="TimestampConfig" />
      <Event code="0x03" type="ObjectName" />
      <Event code="0x06" type="TaskReady" />
      <Event code="0x07" type="SwitchToTask"/>
    </EventGroup>
    <EventGroup name="Task">
```

```
    <Event code="0x30" type="ObjectState">
      <Param index="0" type="Handle" class="Task"/>
      <Param index="1" type="Int32" useAs="StateAfter"/>
      <FollowWith key="TaskCreate" />
    </Event>
    <Event key="TaskCreate" service="Task/xTaskCreate" type="KernelServiceReturn" status="StatusOK">
      <Param index="0" type="Handle" class="Task" useAs="Arg"/>
      <Param index="1" type="UInt32" useAs="Arg"/>
    </Event>
  </EventGroup>
  <EventGroup name="MUTEX">
    <Event code="0x40" service="MUTEX/MUTEX_Create" type="KernelServiceReturn">
      <Param index="0" type="Handle" class="MUTEX" useAs="Arg"/>
    </Event>
    <Event code="0x42" service="MUTEX/MUTEX_Lock" type="KernelServiceReturn">
      <Param index="0" type="Handle" class="MUTEX" useAs="Arg"/>
    </Event>
    <Event code="0x45" service="MUTEX/MUTEX_Release" type="KernelServiceReturn">
      <Param index="0" type="Handle" class="MUTEX" useAs="Arg"/>
    </Event>
  </EventGroup>
 </EventCodes>
 <TargetPlatform>
  <KernelServiceGroup name="Task">
    <KernelService name="xTaskCreate" operation="Initialize">
      <Parameter name="task" format="Task {0}" />
      <Parameter name="priority"/>
    </KernelService>
  </KernelServiceGroup>
  <KernelServiceGroup name="MUTEX">
    <KernelService name="MUTEX_Create" operation="Initialize"/>
    <KernelService name="MUTEX_Lock" operation="LockMutex"/>
    <KernelService name="MUTEX_Release" operation="ReleaseMutex"/>
  </KernelServiceGroup>
  <ObjectClasses>
    <ObjectClass name="Task" type="Actor"/>
    <ObjectClass name="ISR" type="Actor"/>
    <ObjectClass name="MUTEX" type="Mutex"/>
  </ObjectClasses>
  <TaskPriorityDirection>HigherNumberIsMoreImportant</TaskPriorityDirection>
 </TargetPlatform>
</ PlatformExtension>
```

We create a new *EventGroup* for MUTEX in the *TargetPlatform* section, where we add three events corresponding to creating, locking and releasing mutex objects, using KernelService elements.

Finally, we add a new *ObjectClass* element for MUTEX objects. This is mapped to the Tracealyzer object class type *Mutex*, which allows Tracealyzer to understand what kind of object it is and account for that in the presentation (e.g., in the Object History view). The list of object class types can be found in Appendix A.

The name “MUTEX” is intended to be the native type name. This is displayed in Tracealyzer but does not affect the analysis. This allow Tracealyzer to display more recognizable type names, matching the platform’s terminology.

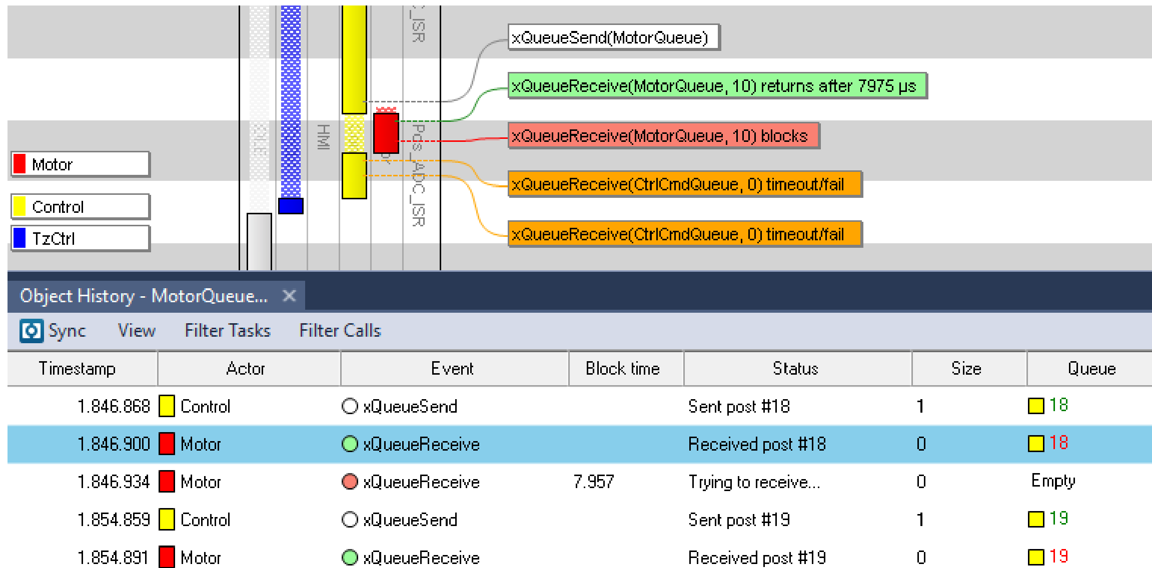
You may have multiple *ObjectClass* elements mapped to the same object class type, like shown below.

```
<ObjectClass name="MutexType1" type="Mutex"/>
<ObjectClass name="MutexType2" type="Mutex"/>
```

4.2. Status awareness, blocking and timeouts

Tracealyzer can understand the status of *KernelService* events (API calls), such as success, failure, blocking and timeout, and use this to derive the state of objects over time, like the number of messages in a queue.

For example, in Figure 4 the event xQueueSend(MotorQueue) is shown as a white label, meaning immediate success (no blocking). And since MotorQueue is known to be of object class type *Queue*, and the service xQueueSend is tagged with operation *Enqueue*, Tracealyzer understands that an entry has been added to MotorQueue. Tracealyzer can even figure out which of the following xQueueReceive() operations that received this particular message, and vice versa.



Timestamp	Actor	Event	Block time	Status	Size	Queue
1.846.868	Control	xQueueSend		Sent post #18	1	18
1.846.900	Motor	xQueueReceive		Received post #18	0	18
1.846.934	Motor	xQueueReceive	7.957	Trying to receive...	0	Empty
1.854.859	Control	xQueueSend		Sent post #19	1	19
1.854.891	Motor	xQueueReceive		Received post #19	0	19

Figure 4. Status awareness for API call status and related color-coding of events.

The status outcome of API calls is visualized by color coding of the event labels – white for successful non-blocking calls, red for blocking calls, green for return after blocking, and orange on timeouts and immediate errors (e.g. bad parameters).

Let's begin with how status is provided for non-blocking API calls. Let's assume xQueueReceive() is a non-blocking call. We then have two cases, Success (message received) or Timeout/Error (no message), each with a separate event code. We define these as two separate events, both linked to the same service (Queue/xQueueReceive) but with different status attributes.

```
<Event code="0x52" service="Queue/xQueueReceive" type="KernelServiceReturn" status="StatusOK">
        <Param index="0" type="Handle" class="Queue" useAs="Arg"/>
        <Param index="1" type="TimeoutInOsTicks" useAs="Arg"/>
        <Param index="2" type="Int32" useAs="StateAfter"/>
</Event>

<Event code="0x53" service="Queue/xQueueReceive" type="KernelServiceReturn" status="StatusTimeout">
        <Param index="0" type="Handle" class="Queue" useAs="Arg"/>
        <Param index="1" type="TimeoutInOsTicks" useAs="Arg"/>
        <Param index="2" type="Int32" useAs="StateAfter"/>
</Event>
```

But xQueueReceive() is actually a blocking function. To show blocking with red and green labels, we need two events for each call. First a *KernelServiceEntry* event when the function begins and then a *KernelServiceReturn* when the function returns. The *KernelServiceEntry* events should always have status set to *StatusBlocks*. So we need to add the following:

```
<Event code="0x54" service="Queue/xQueueReceive" type="KernelServiceEntry" status="StatusBlocks">
        <Param index="0" type="Handle" class="Queue" useAs="Arg"/>
        <Param index="1" type="TimeoutInOsTicks" useAs="Arg"/>
        <Param index="2" type="Int32" useAs="StateAfter"/>
</Event>
```

Note that *StatusBlocks* only means that the operation *may* block, not that every such event indicates blocking. When a specific StatusBlocks operation does not block, Tracealyzer hides the *KernelServiceEntry* event and only show the *KernelServiceReturn* events to reduce the number of labels shown. If the call was successful (*StatusOK*), the label is shown in white instead of green, since there was no blocking to return from. This optimization can be observed for the xQueueSend() call in Figure 4.

5. Tracing Interrupt Service Routines (ISRs)

The TraceRecorder allows for tracing interrupt handlers by calling xTraceISRBegin() and xTraceISREnd(), i.e. when the handler begins and end. This is already RTOS independent, so no porting is required here.

This is intended for selective tracing of relevant application-specific external interrupt handlers. It is not suitable for highly frequent and/or timing-critical ISRs. While the TraceRecorder is highly optimized and fast, a small ISR might be just a single line of code and might be executed thousands of times per second. Adding software-based tracing to such ISRs will add a relatively high overhead. For this reason, we don't recommend tracing the RTOS kernel interrupts (such as the OS Tick) using these functions.

You may also store a name and the IRQ priority level of the traced interrupt handler using xTraceISRRegister(). These are documented in the Tracealyzer User Manual and in *trcRecorder.h*.

6. Stream ports

A stream port defines how the TraceRecorder library should communicate with the host-side Tracealyzer application. Its interface consists of two preprocessor #defines that point to custom functions for sending trace data and receiving commands from Tracealyzer (start/stop). A stream port can use any channel, including debug interfaces and functional I/O like TCP or UDP sockets. See the /streamports folder for examples.

The RingBuffer stream port is provided to allow for snapshots of the most recent events. In this setup, the trace data is kept in a target-side ring buffer until collected. This buffer only keeps the most recent data, as older data is overwritten when the buffer becomes full. Snapshot traces are useful in field testing and deployment where a connection for continuous streaming might not be available. This setup can be used within Percepio DevAlert to provide trace snapshots on reported anomalies.

For streaming to host it is possible to use a unidirectional channel, although a bidirectional channel is better. In the latter case, it is possible to start and stop the tracing interactively from the Tracealyzer application. With only a unidirectional channel, the tracing needs to be started from the target system by calling vTraceEnable(TRC_START). Note that the Tracealyzer host application must be in listening mode at this point, i.e. press "Start" in Tracealyzer before the target system is started.

Ideally, this stream port interface should be independent of the traced functions, meaning that the transfer of trace data should not result in additional traced events. For instance, if using a TCP/IP stack as stream port, this might involve semaphores, message queues or mutexes, i.e., operations that typically are traced. This will increase the amount of data that is generated and in worst case saturate the stream port interface.

7. Additional notes

Typically, we don't record any event when a task is switched out, i.e. in the context-switch from the previous task to the kernel, but only when the new task is switched in. This reduces the number of task-switch events and also focuses the visualization on the application tasks instead of the kernel behavior.

However, this means that the kernel interrupt execution (e.g. task-switch time) is not visible, but instead attributed to the previously executing actor. This time is however negligible in most cases, although the effect can be relatively large on short tasks.

Appendix A - TargetPlatform XML reference

Object class types

Object class type	Description
Actor	An actor object, i.e. an execution context such as a task, thread or interrupt handler.
Queue	An IPC queue object that can hold multiple messages. Generally, one or more actors will write/post to the queue and one actor will read/receive the messages.
Semaphore	A semaphore type object. It generally supports increase and decrease operations. These are sometimes used as mutexes, but more often used for signaling between actors.
Mutex	Some form of lockable object. This is typically used to control access to some shared resource.
Heap	Represents some form of memory that can be allocated in runtime. This class type can also be used to describe different memory or block pools.
IOChannel	This is used to describe some form of data stream object, such as a file handle or TCP socket. Note that for a socket, this does not describe the connection but rather the individual sockets.
Other	Some other type of object. Tracealyzer will not understand this object, but it can be referenced by services and listed in history views.

KernelService Attributes

Attribute	Description
name	The name of the service (the traced function).
operation	The type of operation this service performs. See below for possible values.
parameters	None, Object, ObjectAndNumericParameter, NumericParameterOnly or Variable. Default is Object. This is used to format the displayed string with respect to the parameters.

Kernel Service: “operation” attribute

Operation	Class types	Description
LockMutex	Mutex	Represents an attempt to lock/acquire the object.
ReleaseMutex	Mutex	Represents an unlock/release of the object.
Enqueue	Queue	Posts a message to the end of the queue.
EnqueueFirst	Queue	Posts a message to the front of the queue.
Dequeue	Queue	Receives and removes a message from the front of the queue.
Clear	Queue	Removes all messages from the queue
IncreaseSemaphore	Semaphore	Increases the value of the semaphore. Typically releases the semaphore and anyone waiting for it.
DecreaseSemaphore	Semaphore	Decreases the value of the semaphore. Typically waits for the semaphore.
MaximizeSemaphore	Semaphore	Sets the semaphore to its maximum value. This can also be used for events etc. to signal ON.
MinimizeSemaphore	Semaphore	Sets the semaphore to its minimum value (i.e. -2^31). This can be used for event objects to signal OFF as it will effectively prevent IncreaseOperations from releasing it.

Percepio® TraceRecorder Integration Guide. Copyright © Percepio AB, 2023
https://percepio.com

Operation	Class types	Description
MinimizeSemaphoreNoLock	Semaphore	Sets the semaphore to 0.
SetSemaphore	Semaphore	Sets the semaphore value explicitly.
OtherRead	Other	Some form of read operation. This will allow Tracealyzer to track data flow in the communication flow graph, without really understanding what the operation or object is really about.
OtherWrite	Other	Like above except it's some form of unknown write operation.
Initialize	All	Should be used when objects are created in runtime, at least if objects can be deleted.
Deinitialize	All	Destroys the object. Should be used if the object is deleted in runtime.
ChangePriority	Actor, ActorGroup	Changes the priority of an actor.
StartInstance	Actor	This service will cause the referenced actor to wake up, right away or delayed. This could for example be some SetTimeout() or StartTimer() service.
AllocateMemory	Heap	Some form of memory allocation. More often than not this is malloc(), but it can also be used for other things like block pools or similar.
FreeMemory	Heap	Some form of memory release/free. Releases a previously acquired memory block.
Delay	Actor / none	A delay operation, such as Sleep(). It will typically only have a numerical value representing the time to be inactive in which case it will implicitly be assumed to reference the caller.
ReAllocateMemory	Heap	Resize an existing memory allocation. This is typically the realloc() C function.
ReadButKeep	Queue	Receives a message from a queue, but does not remove it.
Remove	Queue	Removes a message from a queue.
Peek	Queue	Checks if there's a message in a queue, and if so returns it without removing it.
WaitButDoNotRead	Queue	Waits for a message to arrive, but without reading or removing it.
WaitForMultipleObjects	Queue, Semaphore, Mutex, Other	Waits for multiple objects. Note that this can represent both WaitForAny or WaitForAll scenarios.

Appendix B – EventCodes XML reference

Event: "type" attribute

- TraceStart – generated when calling xTraceEnable.
- TimestampConfig – generated when calling xTraceEnable.
- ObjectName – sets the display name of an object.
- ObjectState – sets the object data attribute of an objects (e.g., task scheduling priority).
- IsrProperties – sets the display name and priority level of ISRs.
- TaskReady – should be provided when an actor is activated (becomes ready to execute)
- OsTimestamp – can be generated on OS Tick events (optional).
- SwitchToTask – generated on context-switches where a task is started (application thread)
- SwitchToInterrupt – generated by xTraceISRBegin, i.e. when a traced ISRs has started
- KernelServiceEntry – generated on entry to a kernel service call (only needed on potentially blocking calls)
- KernelServiceReturn - generated when returning from a kernel service call
- KernelServiceCallFromIsr – on kernel service calls that are only intended for use in ISRs
- UserEvent – a generic user-defined event, e.g. from calling xTracePrintF
- LowPowerBegin – generated just before the system goes into tickless idle (processor sleep with tick interrupts disabled)
- LowPowerEnd – generated when waking up after a tickles idle period
- PriorityChange – generated when an actor (task) changes scheduling priority
- InstanceEnd – optional event to explicitly mark the current actor instance as finished
- ForgetObject – unregisters an Object Handle. Typically added as a "FollowsWith" from Delete/Close KernelService events.
- Notice – a target-defined "Notice" event with a message
- InstanceNotice – like Notice but connected to the current actor instance

Event: "status" attribute

There are three types of *KernelService* events, for which valid status attributes are:

	StatusOK	StatusTimeout	StatusError	StatusBlocking
`KernelServiceReturn`	Allowed	Allowed	Allowed	N/A
`KernelServiceEntry`	N/A	N/A	N/A	Required
`KernelServiceCallFromIsr`	Allowed	Allowed	Allowed	N/A

KernelServiceEntry events are typically only generated when a kernel service calls blocks (resulting in a context switch). On non-blocking calls it is sufficient with a KernelServiceReturn event.

Param: "index" attribute

`index` is used to tell `Tracealyzer` which parameter slot to put the parameter data in.

Param: "type" attribute

- Handle – An object handle (identifier)
- TimeoutInOsTicks – The timeout parameter in blocking systems calls with an explicit timeout duration, expressed in "OS Ticks".
- TimestampInOsTicks – A timestamp expressed in "OS Ticks".
- Int32 – Signed integer
- UInt32 – Unsigned integer
- String – A null-terminated string stored in the event. This typically needs to be the last parameter.
- StringHandle – A handle to a string, e.g., provided by `xTraceStringRegister()`. The string itself is not stored in the event.
- IPv4Address – An IPv4 address, e.g. from tracing certain function calls in a TCP/IP stack.

Param: "useAs" attribute

- Arg – For KernelService events, the parameter is an argument to the kernel service.
- Return – For KernelService events, the parameter is the return value from the kernel service.
- StateAfter – For KernelService events, the parameter is the "state" of the referenced object after the operation. For instance, if the event is a "send to queue", the UseAs="StateAfter" parameter should contain the number of buffered messages after the "send" has completed.
- Channel – For User Events, the parameter is the User Event Channel. This is typically applied on parameters of type "stringhandle".
- Message – For parameters of type "string", e.g. in User Events, this means that the string should be used as the displayed event string.
- PrintfArgStart – For User Events, this indicates the first entry in the list of data arguments.
- Actor – The parameter is a reference to an Actor. Used in PriorityChange events, see e.g. FreeRTOS-v1.2.0.xml.
- Priority – The parameter is an actor priority. Used in PriorityChange events, see e.g. FreeRTOS-v1.2.0.xml.
- TimestampInOsTicks – The parameter is a timestamp. Used in OsTimestamp events, see e.g. FreeRTOS-v1.2.0.xml.
- ReturnStatus – Allows for setting the service return status (StatusOK, StatusError, StatusTimeout) conditionally, based on a parameter value. See e.g. aws_wifi-v1.0.0.xml. Can be combined with "Return" by adding a separate parameter element with the same index.