

## Using Percepio TraceRecorder with “BareMetal” systems

Tracealyzer is based on instrumentation-based tracing that works with essentially any processor using the open-source Percepio TraceRecorder library. This offers predefined integrations for FreeRTOS, SafeRTOS, Zephyr and Azure RTOS ThreadX, as well as a Bare Metal integration option.

The **Bare Metal** variant offers an RTOS-independent solution without predefined kernel tracing, allowing for application-level tracing in any C/C++ software by custom instrumentation.

The Bare Metal option is also a good starting point for more advanced custom instrumentation using the Tracealyzer SDK. This provides documentation and code examples for custom integrations of the TraceRecorder library and allows for leveraging the full functionality of Tracealyzer, including the powerful RTOS awareness.

### What can be traced?

- **Interrupt service routines (ISRs)** can be traced by adding calls to `xTraceISRBegin()` and `xTraceISREnd()` in your interrupt handlers. See the TraceRecorder ISR functions in [trcISR.h](#) for further details. These are included in the general public API, accessed by including **trcRecorder.h**. Note: If ISR tracing is not used, the CPU time spent in ISRs will instead be attributed to the currently running task.
- You may log custom **User Events** using the functions in the [trcPrint API](#). These are included in the general public API, accessed by including **trcRecorder.h**. User events can be used as debug messages to provide more information in Tracealyzer about what your code is doing. User Events may also be used for data logging and can be plotted in the User Event Signal Plot.
- Tracealyzer v4.7 and later also supports **Runnable Tracing**. Runnables is an automotive concept corresponding to top-level functions or "sub-tasks". However, runnable tracing is applicable and useful in most systems. With runnable tracing you can see the execution times of specific code sections and also visualize these "runnables" in the trace view to better understand the system execution. See [trcRunnable.h](#) for more details.
- **Custom intervals** and **state variables** can also be traced, either using general User Events but also using more efficient APIs in [trcInterval.h](#) and [trcStateMachine.h](#).

Learn more about Tracealyzer User Events, State Machines and Intervals in these blog posts:

- <https://percepio.com/understanding-your-application-with-user-events/>
- <https://percepio.com/visualizing-state-machines/>
- <https://percepio.com/digging-deeper-into-state-machines-visualization/>

Note that each traced event typically adds a few microseconds to the execution time, so the execution times will be somewhat higher than without tracing enabled. The ISR tracing support is mainly intended for debugging purposes, not for accurate profiling of ISR execution times in the microsecond range.

### Integrating the TraceRecorder

To integrate the TraceRecorder using the BareMetal option, follow these steps:

1. Decide which [streamport](#) to use, i.e. where the TraceRecorder should store or transmit the data. Predefined streamports are found in the **/streamports** folder. When used with Percepio Detect or DevAlert, select **RingBuffer**. This is also recommended as the starting point for general use, since easy to get started with. This streamport writes the trace data to a circular RAM buffer from which "snapshots" can be saved.
2. Copy the follow TraceRecorder code into your project:

```
TraceRecorder/*.c
TraceRecorder/kernelports/BareMetal/*.c
TraceRecorder/streamports/<StreamPort>/*.c
```

3. Add the following header files to your project. You can put all header files in the same directory if you prefer. Make sure to **update the compiler's "include paths"** to ensure all header files are found.

```
TraceRecorder/config/*.h
TraceRecorder/include/*.h
TraceRecorder/kernelports/BareMetal/config/*.h
TraceRecorder/kernelports/BareMetal/include/*.h
TraceRecorder/streamports/<StreamPort>/config/*.h
TraceRecorder/streamports/<StreamPort>/include/*.h
```

4. In **/config/trcConfig.h**, set TRC\_CFG\_HARDWARE\_PORT to a suitable *hardware port* for your device (see /include/trcDefines.h and /include/trcHardwarePort.h).
5. In **/config/trcConfig.h**, you also may need to include the processors header file. This is needed for the Arm Cortex-M port to allow TraceRecorder to access the CMSIS-Core definitions, but not all hardware ports need this. Remove the line `#error "Trace Recorder: Please include your processor's header file here and remove this line."` and try building the project. If don't you get any errors, you don't need the include. Otherwise include the main header file for your processor here, e.g. `#include "stm32f4xx.h"` .
6. Call `xTraceEnable(TRC_START)` in your `main()` function to initialize and start the recorder. This should be done after the initial hardware setup, but must be called before any other TraceRecorder functions. Optionally, `xTraceInitialize()` can be used to initialize the trace system without starting the tracing. Then call `xTraceEnable()` at a later point to start the tracing. Other start options for `xTraceEnable` are described in **trcRecorder.h**.

For technical support, please contact [support@percepio.com](mailto:support@percepio.com).