



TraceRecorder Integration User Guide

Version 1.1.0

Published September 1, 2025

Table of Contents

About this guide	ii
Conventions	iii
Feedback	iii
Chapter 1: Distribution Overview	3
Updating the Distribution	4
Tracealyzer Information	4
Chapter 2: Functional Overview	5
High-Level TraceRecorder Integration Overview	5
The Trace Buffer	6
Chapter 3: Installation and Use	7
Recording ISRs	8
Configuring the Trace Buffer	9
Recording Custom User Events	10
Exporting and Viewing the Trace	10
Chapter 4: STM32F769I-DISCO IAR Example	11
Building the Example with TraceRecorder	11
Recording the SysTick ISR	16
Recording a Customer User Event	17
Adding a Debugger Macro for Exporting the Trace	18
Exporting and Viewing the Trace	19



About this guide

The purpose of this guide is to provide embedded developers with the information necessary to successfully integrate Percepio's TraceRecorder library with the PX5 Real-Time Operating System (RTOS). This guide is organized into the following chapters:

Chapter 1: Distribution Overview. This chapter provides an overview of the distribution, including where to find the documentation and source code.







Chapter 2: Functional Overview. This chapter describes how TraceRecorder is integrated with the PX5 RTOS.

Chapter 3: Installation and Use. This chapter describes how to integrate TraceRecorder with the PX5 RTOS.

Chapter 4: STM32F769I-DISCO IAR Example. This chapter goes through the complete integration process using a project included in the STM32F769I-DISCO IAR PX5 RTOS evaluation workspace.

Conventions

Various conventions are used in this guide. C source code function prototypes and examples are written in `courier` font. Supplemental documentation, API, and parameter names are italicized when discussed. There are also several symbols that are used to highlight important features or topics, as follows:

Symbol	Meaning
	This is a general information symbol.
	The caution symbol indicates that the user should be aware of important usage scenarios associated with a specific topic or API.
	The danger symbol indicates that the user should be aware of the serious consequences of certain scenarios associated with a specific topic or API.
	This symbol indicates the case where the associated API scenario does not result in preemption.
	This symbol indicates the case where the associated API scenario results in preemption.
	This symbol indicates the case where the associated API scenario results in suspension.

Feedback

All feedback is greatly appreciated. Please send e-mail feedback to support@px5rtos.com with "PX5 RTOS TraceRecorder Integration User Guide Feedback" in the subject line.

Chapter 1: Distribution Overview

This chapter describes the PX5 RTOS TraceRecorder integration distribution. The distribution is in a GitHub repository (located here: https://github.com/PX5-RTOS/px5_trace_recorder) and contains the following folders:

```
PX5-RTOS_TraceRecorderSource
    config
    definition_file
    documentation
    include
    streamports
```

The *PX5-RTOS_TraceRecorderSource* folder contains the C source files of the TraceRecorder library and the following folders.

The *config* folder contains files that allow the user to configure the TraceRecorder library.

The *definition_file* folder contains the XML file Tracealyzer uses to interpret the raw trace data.

The *documentation* folder contains this document.

The *include* folder contains the C header files of the TraceRecorder library.

The *streamports* folder contains code for different types of streaming supported by TraceRecorder; currently for the PX5 RTOS, only RingBuffer is supported.

Updating the Distribution

The PX5-RTOS_TraceRecorderSource repository is a fork of the TraceRecorder library and continuously pulls changes from it to ensure the latest updates are included. As such, the user only needs to pull changes from the PX5-RTOS_TraceRecorderSource repository to ensure they have the most up-to-date code.

Tracealyzer Information

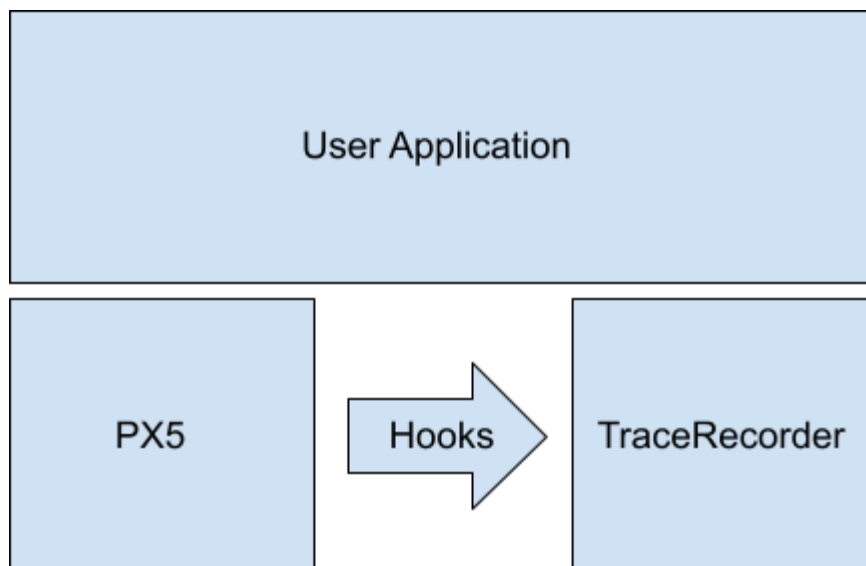
Percepio's Tracealyzer is the application that displays the trace file generated from the TraceRecorder library. For more information about Tracealyzer – including a download link – use the following link:
<https://percepio.com/tracealyzer/>

Chapter 2: Functional Overview

Tracealyzer provides system-level visualization to applications running on the PX5 RTOS. This visualization covers all system events, including API calls, context-switches, thread suspensions, thread resumptions, and Interrupt Service Routines (ISRs). It allows developers to capture long software traces, spanning hours or even days, for example for burn-in testing or profiling, or when looking for rare errors or system crashes.

High-Level TraceRecorder Integration Overview

At the highest level, the TraceRecorder integration package looks like the following:



The PX5 RTOS has hooks (implemented via macros) that are configured to call the TraceRecorder library to record various RTOS events, including API calls, context-switches, etc. These hooks are implemented in *PX5/include/trcKernelPort.h*.

The user application can also record its own custom events if desired – this is explained more in [Recording Custom User Events](#).

The Trace Buffer

TraceRecorder records events into an internal ring buffer. This buffer is part of a larger data structure that contains all the data TraceRecorder uses to function. This data structure is a statically allocated global variable defined in **PX5-RTOS_TraceRecorderSource/trcStreamingRecorder.c**:

```
static TraceRecorderData_t xRecorderData TRC_CFG_RECORDER_DATA_ATTRIBUTE;
```

Note the `TRC_CFG_RECORDER_DATA_ATTRIBUTE`: this is a configurable symbol that the user can define to place the data structure in a specific memory RAM section. This symbol is defined in **PX5-RTOS_TraceRecorderSource/config/trcConfig.h**.

Chapter 3: Installation and Use

Integrating TraceRecorder with the PX5 RTOS is easy. Simply follow these steps:

- (1) Clone the `PX5-RTOS_TraceRecorderSource` repository from GitHub (located here: https://github.com/PX5-RTOS/PX5-RTOS_TraceRecorderSource).
- (2) Include all C source files from the following folders into your PX5 RTOS project:
 1. `PX5-RTOS_TraceRecorderSource`
 2. `PX5-RTOS_TraceRecorderSource/streamports/RingBuffer`
- (3) Add the following directories to your compiler's include path:
 1. `PX5-RTOS_TraceRecorderSource/config`
 2. `PX5-RTOS_TraceRecorderSource/include`
 3. `PX5-RTOS_TraceRecorderSource/streamports/RingBuffer/config`
 4. `PX5-RTOS_TraceRecorderSource/streamports/RingBuffer/include`
- (4) In `PX5-RTOS_TraceRecorderSource/config/trcConfig.h` set `TRC_CFG_HARDWARE_PORT` to the architecture used. For some hardware ports an `#include` of the processor's header file needs to be done by replacing the line `#error "Trace Recorder: Please include your processor's header file here and remove this line."` If the error line is removed and your project compiles the header file isn't needed. All available hardware ports can be found at the bottom of `PX5-RTOS_TraceRecorderSource/include/trcDefines.h`.
- (5) In `PX5-RTOS_TraceRecorderSource/config/trcKernelPortConfig.h` set `TRC_CFG_CPU_CLOCK_HZ` to the frequency used by the timer used for time stamping in the hardware port.

- (6) In *px5_user_config.h*, add `#include "trcRecorder.h"`.
- (7) Define *PX5_THREAD_ENTER_EXIT_NOTIFY_ENABLE* in your assembler's preprocessor settings.
- (8) In main after *platform_setup* is called (or any time after the timestamp source is setup) and before *px5_pthread_start* is called, add a call to *xTraceEnable* with *TRC_START* as the argument. For example:

```
int main()
{
    /* Call the platform setup function.  */
    platform_setup();

    /* Enable and start tracing.  */
    xTraceEnable(TRC_START);

    /* Start the PX5 RTOS.  */
    px5_pthread_start(...);

    . . .
}
```

At this point the program is configured for tracing and RTOS events will be recorded into an internal TraceRecorder buffer.

Recording ISRs

To record ISRs in the trace, the following steps must be taken for each ISR:

- (1) Declare an ISR trace handle:

```
TraceISRHandle_t SysTickTraceHandle;
```

- (2) Register the ISR with TraceRecorder; this should be done after *xTraceEnable* but before *px5_pthread_start*:

```
int main()
{
    . . .

    /* Enable and start tracing.  */
    xTraceEnable(TRC_START);

    /* Register the SysTick ISR.  */
    xTraceISRRegister("SysTick", 0, &SysTickTraceHandle);
}
```

```

        /* Start the PX5 RTOS. */
        px5_pthread_start(...);

        . . .
    }

```

The first parameter to *xTraceISRRegister* is the name of the ISR and the second parameter is its priority; these will be displayed in Tracealyzer.

- (3) Add a call to *xTraceISRBegin* to the beginning of the ISR, passing the previously registered trace handle:

```

void SysTick_Handler(void)
{

    /* Record the beginning of SysTick ISR. */
    xTraceISRBegin(SysTickTraceHandle);

    . . .
}

```

- (4) Add a call to *xTraceISREnd* to the end of the ISR, passing *PX5_TRACE_CONTEXT_SWITCH_PENDING_CHECK*:

```

void SysTick_Handler(void)
{

    . . .

    /* Record end of SysTick ISR. */
    xTraceISREnd(PX5_TRACE_CONTEXT_SWITCH_PENDING_CHECK);
}

```

Each ISR should now be included in the trace and be viewable in Tracealyzer.

Configuring the Trace Buffer

There are a couple compile-time configuration options for the ring buffer.

These options are in *PX5-*

RTOS_TraceRecorderSource/streamports/RingBuffer/config/trcStreamPortConfig.h. The following describes each option in detail:

Build Option	Meaning
<i>TRC_CFG_STREAM_PORT_BUFFER_SIZE</i>	Defines the size of the ring buffer that events are recorded into.

TRC_CFG_STREAM_PORT_RINGBUFFER_MODE

Defines what happens when the buffer is full. Can either be set to *TRC_CFG_STREAM_PORT_RINGBUFFER_MODE_OVERWRITE_WHEN_FULL* for continuous recording, or *TRC_CFG_STREAM_PORT_RINGBUFFER_MODE_STOP_WHEN_FULL* for stopping when the buffer is full.

Recording Custom User Events

Custom user events can be recorded via TraceRecorder's *trcPrint* API, which is documented in *PX5-RTOS_TraceRecorderSource/include/trcPrint.h* (included in this distribution). The two primary APIs are *xTracePrintF* and its optimized versions *xTracePrintF0*, *xTracePrintF1*, *xTracePrintF2*, *xTracePrintF3*, and *xTracePrintF4*. An example usage of these APIs is demonstrated in [the example chapter](#).

Exporting and Viewing the Trace

The TraceRecorder library records events into a RAM ring buffer. In order to view the events in Tracealyzer, this ring buffer must be exported to a .hex or .bin file. How this is done depends on the IDE. The Tracealyzer User Manual provides examples for various IDEs under the "Making snapshots" section – please refer to it.

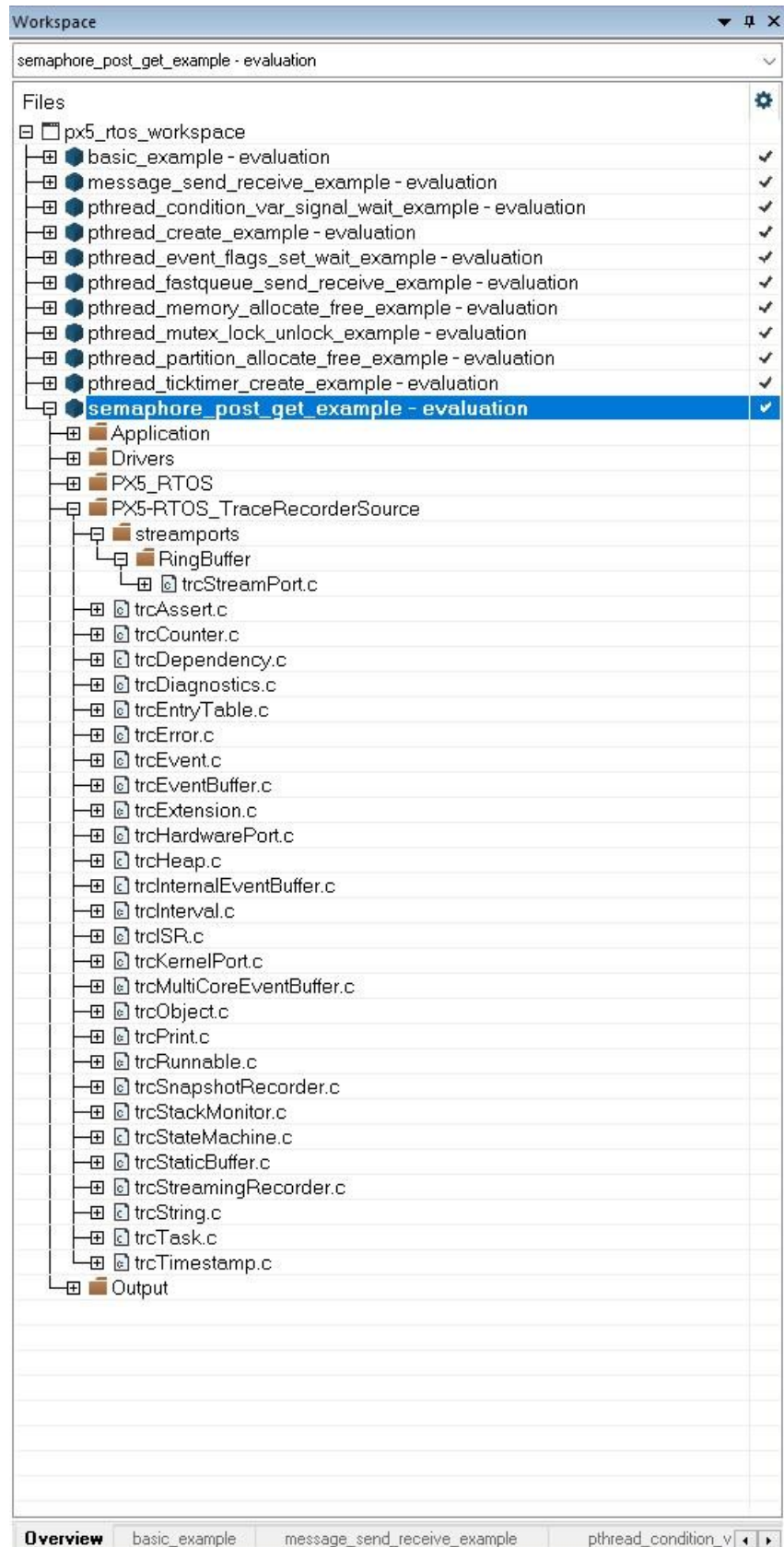
Before opening the trace file (the exported .hex or .bin file) in Tracealyzer, one must first add the definitions file for the PX5 RTOS to Tracealyzer's search path - this file tells Tracealyzer how to interpret the trace file. To do this, open Tracealyzer and go to "File -> Settings -> Project Settings -> Definition File Paths" and add the *PX5-RTOS_TraceRecorderSource/definition_file* directory. Upon doing so, you should be able to open the trace file in Tracealyzer.

Chapter 4: STM32F769I-DISCO IAR Example

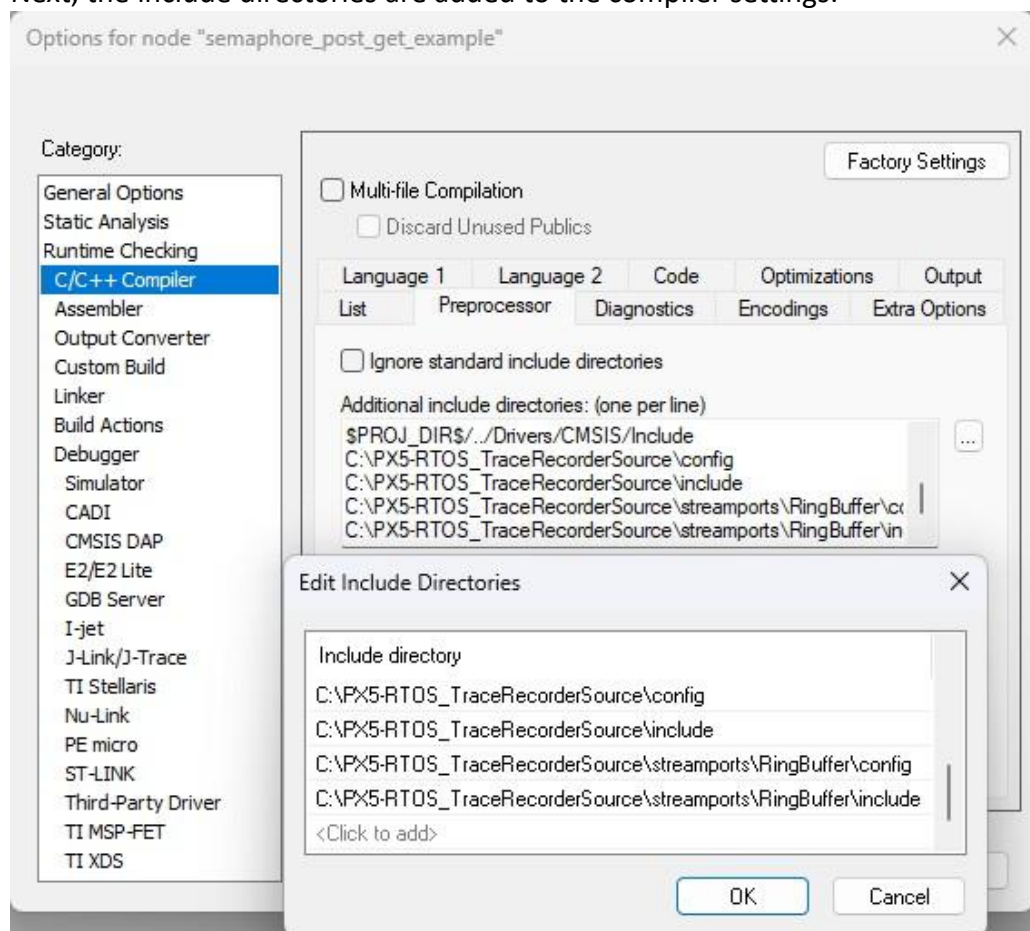
This chapter provides an example demonstrating the integration process starting from the *semaphore_post_get_example* in the PX5 RTOS IAR evaluation workspace for the STM32F769I-DISCO. At the end, a trace is exported and viewed in Tracealyzer. While this example is for a specific board and IDE, the process will be similar to other hardware and tools.

Building the Example with TraceRecorder

After cloning the PX5-RTOS_TraceRecorderSource repository from GitHub, the C source files are then included in the project:



Next, the include directories are added to the compiler settings:



Next, *TRC_CFG_HARDWARE_PORT* in **PX5-RTOS_TraceRecorderSource/config/trcConfig.h** is set to *TRC_HARDWARE_PORT_ARM_Cortex_M*:

```

/**
 * @def TRC_CFG_HARDWARE_PORT
 * @brief Specify what hardware port to use (i.e., the "timestamping driver").
 *
 * All ARM Cortex-M MCUs are supported by "TRC_HARDWARE_PORT_ARM_Cortex_M".
 * This port uses the DWT cycle counter for Cortex-M3/M4/M7 devices, which is
 * available on most such devices. In case your device don't have DWT support,
 * you will get an error message opening the trace. In that case, you may
 * force the recorder to use SysTick timestamping instead, using this define:
 *
 * #define TRC_CFG_ARM_CM_USE_SYSTICK
 *
 * For ARM Cortex-M0/M0+ devices, SysTick mode is used automatically.
 *
 * See trcHardwarePort.h for available ports and information on how to
 * define your own port, if not already present.
 */
#define TRC_CFG_HARDWARE_PORT TRC_HARDWARE_PORT_ARM_Cortex_M

```

Next, the line *#error "Trace Recorder: Please include your processor's header file here and remove this line"* in **PX5-RTOS_TraceRecorderSource/config/trcConfig.h** is removed and the processor's header file is included:

```

/*****
 * Include of processor header file
 *
 * Here you may need to include the header file for your processor. This is
 * required at least for the ARM Cortex-M port, that uses the ARM CMSIS API.
 * Try that in case of build problems. Otherwise, remove the #error line below.
 *****/
//#error "Trace Recorder: Please include your processor's header file here and remove this line."
#include "stm32f7xx.h"

```

Next, **TRC_CFG_CPU_CLOCK_HZ** in **PX5-RTOS_TraceRecorderSource/config/trcKernelPortConfig.h** is set to the speed of the processor (216 MHz):

```

/**
 * @brief Board CPU clock frequency in Hz. Must be changed from 0.
 */
#define TRC_CFG_CPU_CLOCK_HZ 216000000

```

Next, **trcRecorder.h** is included in **px5_user_config.h**:


```

#ifndef PX5_USER_CONFIG_HEADER
#define PX5_USER_CONFIG_HEADER

/* This file delivered blank and reserved for the application to define
   configuration of PX5. */

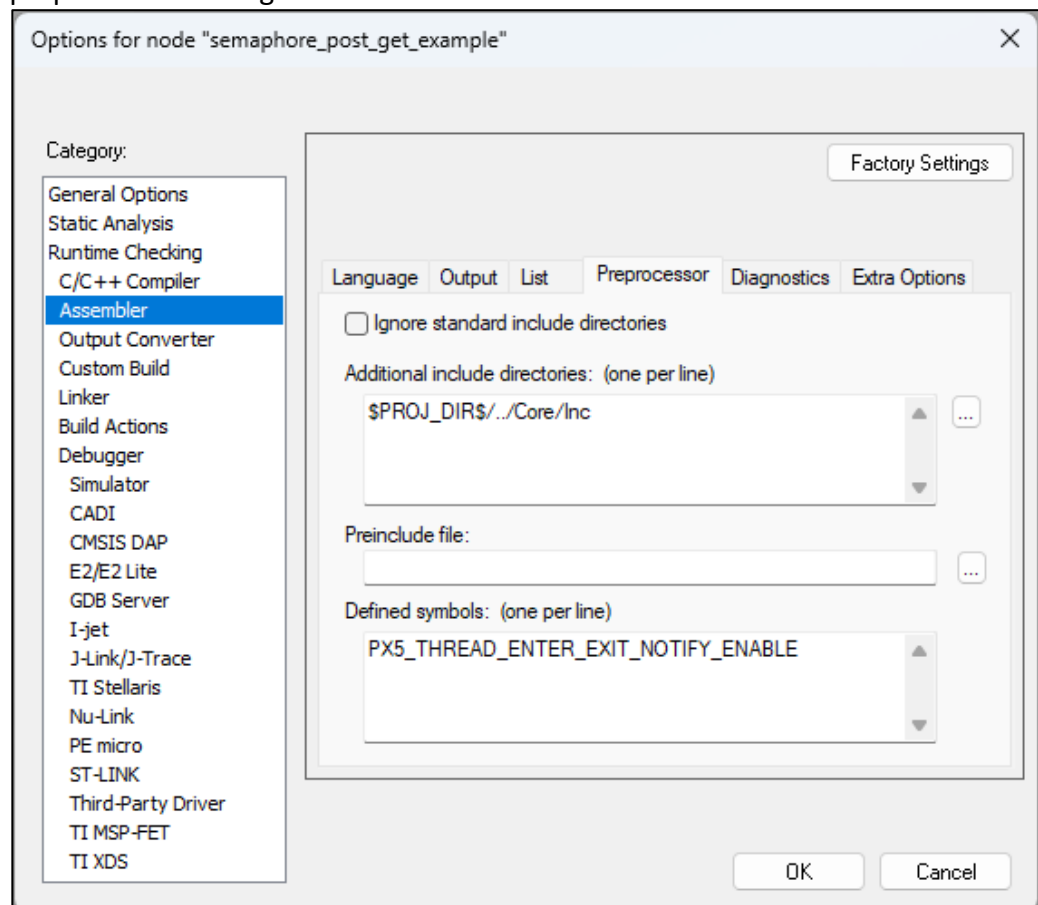
/* #define PX5_LEVEL_3_ERROR_PROCESSING    while(1) { } */

#include "trcRecorder.h"

#endif /* PX5_USER_CONFIG_HEADER */

```

Next, `PX5_THREAD_ENTER_EXIT_NOTIFY_ENABLE` is defined in the assembler's preprocessor settings:



Finally, `xTraceEnable` is called after `platform_setup` but before `px5_pthread_start`:

```

int main()
{

    /* Call the platform setup function. */
    platform_setup();

    /* Enable and start tracing. */
    xTraceEnable(TRC_START);

    /* Start the PX5 RTOS. Note that the value 0xBD93A508 is merely repres
        enhances the PX5 Pointer/Data Verification (PDV) when enabled. */
    px5_pthread_start(0xBD93A508, memory_area, sizeof(memory_area));
}

```

At this point, the example is configured for tracing and RTOS events will be recorded in TraceRecorder's internal buffer.

Recording the SysTick ISR

To record the SysTick ISR, first an ISR trace handle is declared as a global variable:

```

/* Copyright (C) PX5 - all rights reserved. */

#include <stdio.h>
#include "pthread.h"
#include "sched.h"
#include "semaphore.h"

/* The PX5 semaphore post-get example shows how easy it is to use semaphores &
relationship. In this example, a consumer child thread is created and waits
The main thread acts as the producer, feeding the consumer child thread the
semaphore post API calls. Note that for better clarity, API return values &
PX5 recommends that applications check all API return values in production

/* Define the trace handle for the SysTick ISR. */
TraceISRHandle_t SysTickTraceHandle;

/* Define thread handles for the consumer child thread in this example. */
pthread_t      consumer_child_thread;

```

Next, the SysTick ISR is registered in *main* after *XTraceEnable* but before *px5_pthread_start*:

```

int main()
{

    /* Call the platform setup function. */
    platform_setup();

    /* Enable and start tracing. */
    xTraceEnable(TRC_START);

    /* Register the SysTick ISR. */
    xTraceISRRegister("SysTick", 0, &SysTickTraceHandle);

    /* Start the PX5 RTOS. Note that the value 0xBD93A508 is merely repres
        enhances the PX5 Pointer/Data Verification (PDV) when enabled. */
    px5_pthread_start(0xBD93A508, memory_area, sizeof(memory_area));
}

```

Lastly, calls to *xTraceISRBegin* and *xTraceISREnd* are added to *SysTick_Handler*:

```

/**
 * @brief This function handles System tick timer.
 */
void px5_timer_interrupt_process(void);

extern TraceISRHandle_t SysTickTraceHandle;

void SysTick_Handler(void)
{

    /* Record the beginning of SysTick ISR. */
    xTraceISRBegin(SysTickTraceHandle);

    /* Call PX5 RTOS timer interrupt processing. */
    px5_timer_interrupt_process();

    /* Record end of SysTick ISR. */
    xTraceISREnd(PX5_TRACE_CONTEXT_SWITCH_PENDING_CHECK);

}

```

The SysTick ISR should now be recorded in the trace.

Recording a Customer User Event

To demonstrate using the *trcPrint* API for recording custom user events, we will record the value of *consumer_child_thread_counter* using the optimized version of *xTracePrintf*:

```

/* Define the consumer child thread (same priority as the "main" thread). */
void *      consumer_child_thread_entry(void *arguments)
{
    TraceStringHandle_t counter_channel;
    TraceStringHandle_t counter_format_string;

    /* Register the channel and format strings. */
    xTraceStringRegister("CounterChannel", &counter_channel);
    xTraceStringRegister("child_thread_counter: %d", &counter_format_string);

    /* Loop indefinitely. */
    while (1)
    {
        /* Increment the consumer child thread counter. */
        consumer_child_thread_counter++;

        /* Record the value of priority_child_thread_counter. */
        xTracePrintF1(counter_channel, counter_format_string, consumer_child_thread_counter);

        /* Wait on the semaphore set from the main thread. */
        sem_wait(&semaphore);

        /* In a real system, processing (or consumption) would happen here - the availability
    }

    return(NULL);
}

```

The value of *consumer_child_thread_counter* will now be recorded in the trace.

Adding a Debugger Macro for Exporting the Trace

The IAR debugger can be configured to save the trace buffer to a host-side file using a debugger macro. Follow these steps:

- (1) Create a macro file in the project directory (e.g. "save_trace_buffer.mac") with the following contents:

```

__var start;
__var end;

save_trace_buffer()
{
    start = __smmessage "Memory:0x",RecorderDataPtr:%x;
    end = __smmessage "Memory:0x",(RecorderDataPtr + 1):%x;
    __memorySave(start,end,"intel-extended", "$PROJ_DIR$\\trace.hex", 0);

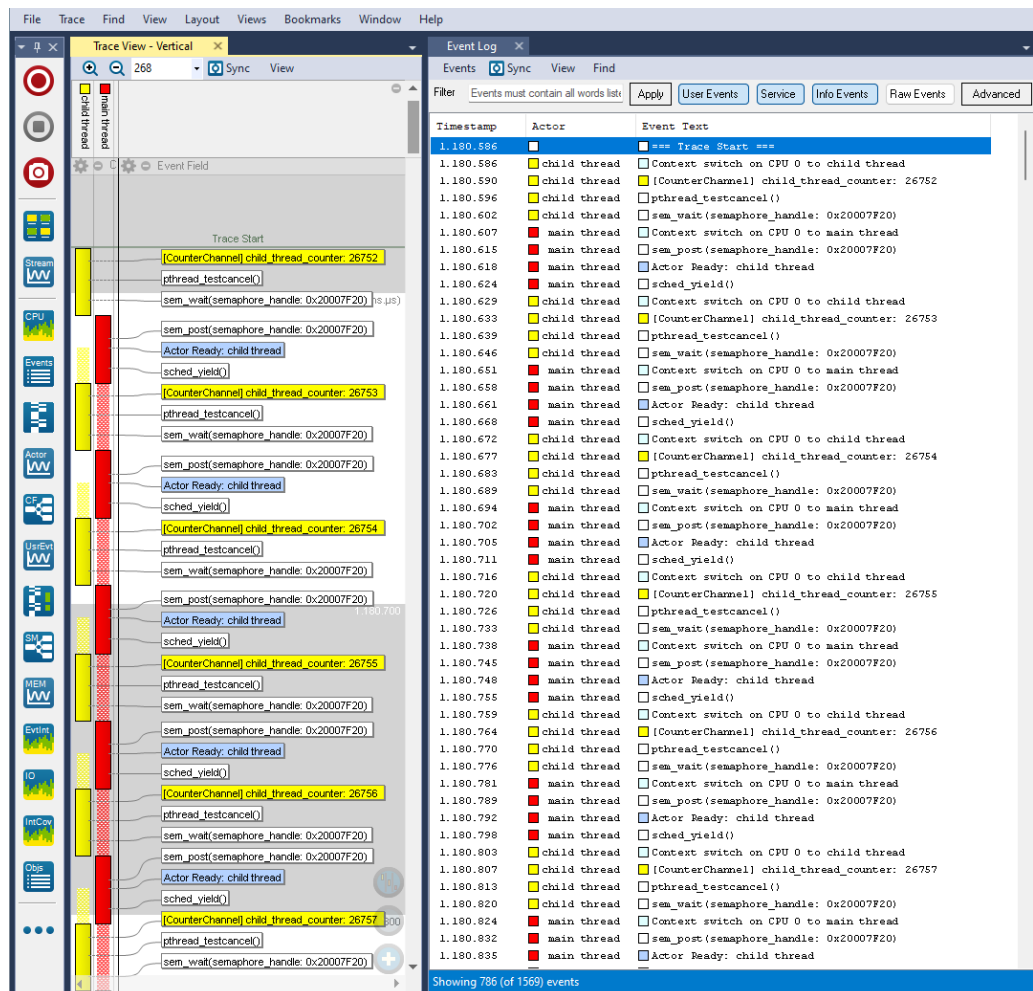
    return 0;
}

```

- (2) Add the macro file under *Options -> Debugger -> Use Macro File(s)*.
- (3) When in a debug session, open *View -> Macros -> Debugger Macros* and check that "save_trace_buffer" is included.
- (4) In the Debugger Macros view, right-click on the "save_trace_buffer" macro and select "Add to Quicklaunch window". Now you can save the trace by double-clicking the blue "refresh" icon in the Quicklaunch window.

Exporting and Viewing the Trace

Run the program for a few seconds and pause it. The trace buffer should be full now. Invoke the "save_trace_buffer" macro to export the trace buffer. A file named "trace.hex" should now be in your project directory. Open it with Tracealyzer and you should see the following:





Enhance • Simplify • Unite

11440 West Bernardo Court • Suite 300
San Diego, CA 92127, USA

Phone: +1 (858) 753-1715
Website: px5rtos.com

© PX5 • All Rights Reserved