Performing ETM and ITM trace on PSOC™ 6 MCU

About this document

Scope and purpose

This application note introduces the trace features of PSOC™ 6 MCU and the supporting software tools, helping you get started with performing instruction (ETM) and instrumentation (ITM) tracing on PSOC™ 6 MCUs. A sample project is configured using ModusToolbox™ and exported to third-party tools like IAR Embedded Workbench and Keil µVision to perform trace. The application note also guides you to explore more features of trace and other resources available online to accelerate your learning.

If you are new to PSOC™ 6 MCU and ModusToolbox™ software environment, see AN228571 - Getting started with PSOC™ 6 MCU on ModusToolbox™ software.

Intended audience

This application note is intended for advanced engineers who want to use the trace capabilities of the PSOC™ 6 MCU.

Table of contents

About this document . . . 1
Table of contents . . . 2
1 Introduction . . . 4
1.1 Trace in MCU designs . . . 4
1.2 The importance of trace . . . 4
1.3 Overview of subsequent chapters . . . 4
2 General and PSOC™ 6 MCU Arm® trace architecture . . . 5
2.1 General trace architecture . . . 5
2.1.1 Trace source . . . 5
2.1.1.1 ETM . . . 5
2.1.1.2 ITM . . . 5
2.1.1.3 MTB . . . 6
2.1.2 Trace sink . . . 6
2.1.2.1 ETB . . . 6
2.1.2.2 ETF . . . 6
2.1.2.3 TPIU . . . 6
2.1.2.4 SWO . . . 7
2.1.3 Trace link . . . 7
2.1.3.1 Funnel . . . 7
2.1.3.2 Replicator . . . 7
2.1.3.3 CTI . . . 7
2.2 Trace output . . . 7
2.2.1 On-chip capture . . . 7
2.2.2 Off-chip capture . . . 8
2.3 Trace infrastructure examples . . . 8
2.3.1 Single-core, off-chip ETM trace example . . . 8
2.3.2 Multi-core, off-chip ETM trace example . . . 8
2.3.3 Multi-core, off-chip ETM CTI trace example . . . 9
2.4 PSOC™ 6 MCU trace infrastructure . . . 9
3 Hardware and software requirements . . . 11
3.1 Hardware requirements . . . 11
3.1.1 Trace probes (external debugger) . . . 11
3.1.2 Development board . . . 11
3.2 Software requirements . . . 12
4 Performing trace on PSOC™ 6 MCU . . . 13
4.1 Creating or importing the project using ModusToolbox™ . . . 13
4.2 Performing trace on IAR Embedded Workbench (EW) . . . 14
4.2.1 Import the ModusToolbox™ project into IAR EW . . . 14

Table of contents

4.2.2 Configure the debugger script and debugger . . . 15
4.2.3 Perform ETM trace . . . 16
4.2.4 Perform ITM trace (printf-style debugging) . . . 18
4.2.4.1 Perform ITM trace using I-jet . . . 18
4.2.4.2 Perform ITM trace using J-Link/J-Trace . . . 22
4.3 Performing trace on Keil µVision . . . 24
4.3.1 Import the ModusToolbox™ project into Keil µVision . . . 24
4.3.2 Configure the debugger script and debugger . . . 25
4.3.2.1 Configure the debugger script with ULINKpro Cortex® Debugger . . . 25
4.3.2.2 Configure the debugger script and debugger with J-Link/J-Trace . . . 28
4.3.3 Perform ETM trace . . . 31
4.3.4 Perform ITM trace (printf-style debugging) . . . 32

5 Summary . . . 35

References . . . 36

Revision history . . . 37

Disclaimer . . . 38

1 Introduction

1.1 Trace in MCU designs

According to Arm®'s definition, "trace" refers to the process of capturing data that illustrates how components in a design operate, execute, and perform. In simpler terms, trace helps capture and visualize the operations occurring inside the MCU in a non-intrusive way.

There are several types of trace, each typically requiring a separate trace generation component in the MCU. These can be broadly classified into four types:

- Instruction trace
- Data trace
- Instrumentation trace
- System trace

While this application note focuses specifically on instruction and instrumentation trace, the approach described can be applied to data and system trace as well.

Instruction trace: Generates information about the execution of instructions within a core or processor. For example, the Embedded Trace Macrocell (ETM) is a source of instruction trace data.

Instrumentation trace: Collects data from a variety of sources, including hardware components such as the Data Watchpoint and Trace unit (DWT) and software components like 32-bit stimulus registers. The data collected can be output in a printf-style format for debugging purposes. The Instrumentation Trace Macrocell (ITM) is used for capturing this instrumentation trace data.

1.2 The importance of trace

Designing a complex embedded system efficiently relies directly on the ability to precisely debug issues that arise during development. One major benefit of the trace technique is its non-intrusive nature, which sets it apart from other debugging techniques. This means that trace can be used to fetch all the instructions running in the core without affecting the actual application flow. Moreover, trace enables the analysis of other system-level information and data without halting the processor or adding extra lines of code.

In addition to providing cycle count information and timestamps, trace data can also be utilized to profile code and measure performance at the function level.

For a comprehensive understanding of the importance of trace, See the Non-intrusive debugging with ETM trace article from IAR Systems.

1.3 Overview of subsequent chapters

The subsequent chapters in this application note provide an overview of the Arm® trace architecture and its implementation in PSOC™ 6 MCUs. The hardware and software requirements for performing trace on PSOC™ 6 MCUs are outlined. Later sections describe how to import an existing PSOC™ 6 MCU project in ModusToolbox™ and enable trace from ModusToolbox™. Additionally, the steps for exporting a ModusToolbox™ project, editing the debugger scripts, and performing trace on third-party tools like IAR Embedded Workbench and Keil µVision are detailed.

2 General and PSOC™ 6 MCU Arm® trace architecture

2.1 General trace architecture

Trace components can be classified into three main types:

- **Trace source**: A component that generates trace data, such as ETM, ITM
- **Trace sink**: A component that stores or outputs the trace data, including Embedded Trace Buffer (ETB), Embedded Trace FIFO (ETF), Trace Port Interface Unit (TPIU), Serial Wire Output (SWO)
- **Trace link**: A component that links trace or non-trace components together, such as Funnel, Replicator and Cross Trigger Interface (CTI)

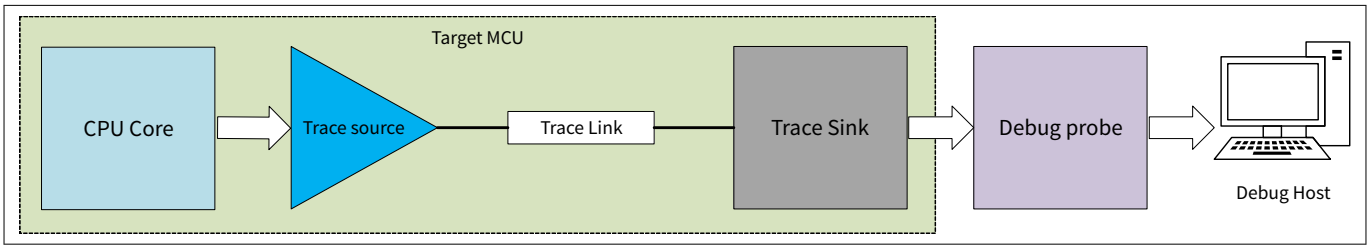


Figure 1 Simplified general trace architecture

2.1.1 Trace source

Although there are many trace source components, the scope of this application note is limited to the ETM and ITM components.

2.1.1.1 ETM

The Embedded Trace Macrocell (ETM) component allows for instruction and data trace. During chip design, the ETM block can be configured to omit data trace if it is not required.

At a high level, the ETM does not generate a trace packet for every instruction the CPU executes. Instead, it outputs information about the instruction flow (jump or no jump) and sometimes outputs the full destination address (when there is a branching instruction). The debug host typically has a copy of the application image, and using the data from the ETM component, the complete program execution can be reconstructed. Since CPU speeds are usually higher, the ETM block needs to compress the instruction execution data and packetize it before sending it to a trace sink.

Between the ETM block and a trace sink, a FIFO buffer is usually provided to allow sufficient time for the trace sink to process and route the trace data.

The ETM data also includes timestamps, which can be used to determine the time consumed by code or functions. Function profiling is very useful in optimizing the regions of your code that consume a lot of CPU bandwidth.

2.1.1.2 ITM

The Instrumentation Trace Macrocell (ITM) component is an application-driven trace source. The data for ITM comes from several sources.

- **Software source** – The application writes directly to the ITM stimulus registers to generate packets. For example, it can send out raw ADC data, which can be plotted and visualized at the debug host
- **Hardware source** – The DWT block provides data watchpoints, data trace, debug events, and profiling counters. The data from these debug systems is routed through the ITM
- **Timestamps** – A counter in the ITM provides a timestamp for each trace packet

2.1.1.3 MTB

The Micro Trace Buffer (MTB) component allows for basic instruction trace. During a trace operation, the debugger configures the MTB to allocate a small portion of the SRAM as a trace buffer for storing trace information. The SRAM can be a dedicated or shared one (system SRAM used by the CPU). When a branching instruction occurs or if the program flow changes due to interrupts, the MTB stores the source and destination PC (program counter) information. The MTB operates mostly in circular buffer mode: when the allocated memory is full, the oldest branch information is overwritten by new branch information.

2.1.2 Trace sink

Similar to the trace source, there are many trace sink components. However, the scope of this document is limited to the following components: ETB, ETF, TPIU, and SWO.

2.1.2.1 ETB

The Embedded Trace Buffer (ETB) is a dedicated SRAM that stores generated trace data on-chip for later retrieval and analysis. The SRAM acts as a circular buffer, wrapping when the buffer size limit is reached. It works by replacing the oldest trace data with the newest data.

2.1.2.2 ETF

The Embedded Trace FIFO (ETF) contains a dedicated SRAM that can be used as either a circular buffer, a hardware FIFO, or a software FIFO. In circular buffer mode, the ETF has the same functionality as the ETB. In hardware FIFO mode, the ETF smooths out fluctuations in the trace data. In software FIFO mode, on-chip software uses the ETF to read out the data over the debug Advanced Microcontroller Bus Architecture (AMBA) peripheral bus interface.

2.1.2.3 TPIU

The Trace Port Interface Unit (TPIU) routes the trace data to external pins, where a debugger is connected to capture the trace data. The TPIU also adds source identification information into the trace stream, enabling trace to be re-associated with its trace source.

Usually, four data pins and one clock pin are associated with the TPIU component, although this is a design-time configuration that can be changed as needed. If the TPIU block has a connection to four data pins, it is not necessary to use all four pins to output the trace data. In this case, the TPIU can be configured to operate in 1-bit, 2-bit, or 4-bit mode.

2.1.2.4 SWO

The trace data from the source is directly passed to an external debugger using a single-wire output called Serial Wire Output (SWO). Due to the required trace bandwidth, the single-pin SWO is not suitable for outputting the ETM trace data; it is mainly used to pass the ITM data.

2.1.3 Trace link

2.1.3.1 Funnel

The funnel merges multiple AMBA Trace Bus (ATB) streams, each carrying data from trace sources, into a single ATB stream. This single ATB stream can then be routed to a trace sink, replicator, or another funnel.

2.1.3.2 Replicator

The replicator splits the AMBA ATB stream coming out of the funnel into multiple ATB streams. This allows the trace data to be routed to multiple sinks.

2.1.3.3 CTI

The Cross Trigger Interface (CTI) generates and routes triggers between different trace components. For example, the ETB sends a trigger to the ETM block to halt the CPU when the buffer is full.

2.2 Trace output

As discussed earlier, trace data is bandwidth-intensive and therefore needs compression or encoding before being converted to packets. As a result, the trace data is not directly in a human-readable format. When captured by a debugger, trace data is decompressed, decoded, and processed to convert it to a human-readable format. Sometimes, when transmitting raw data through the ITM, trace packets can directly contain raw data without encoding.

A debugger can capture trace data in two ways:

- On-chip capture
- Off-chip capture

2.2.1 On-chip capture

In this scenario, the ETB stores the trace data. At specific points during debugging, the external debugger extracts the on-chip trace data using a 2-pin serial wire debug (SWD) interface.

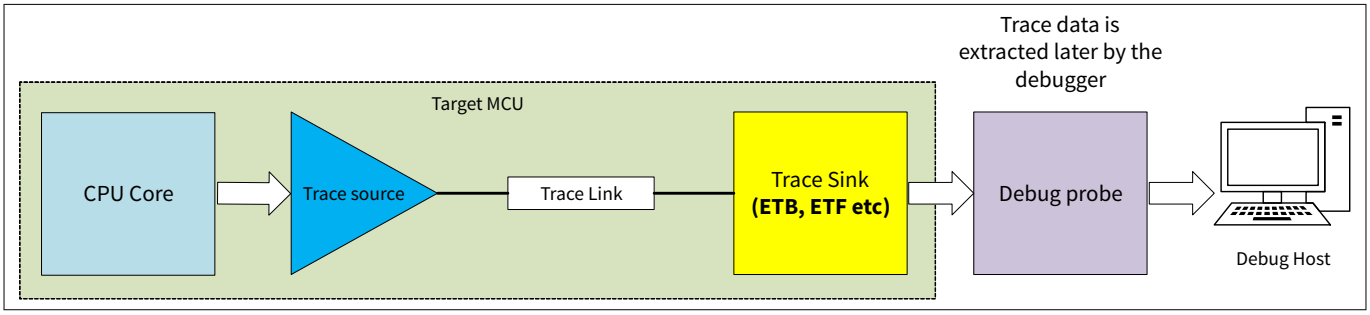


Figure 2 Trace architecture for on-chip capture

2.2.2 Off-chip capture

In this scenario, the TPIU and SWO pins route the trace data to an external debugger in real-time. The debugger then processes the data and displays it in a human-readable format. The primary focus of this application note is to understand this mode of trace capturing.

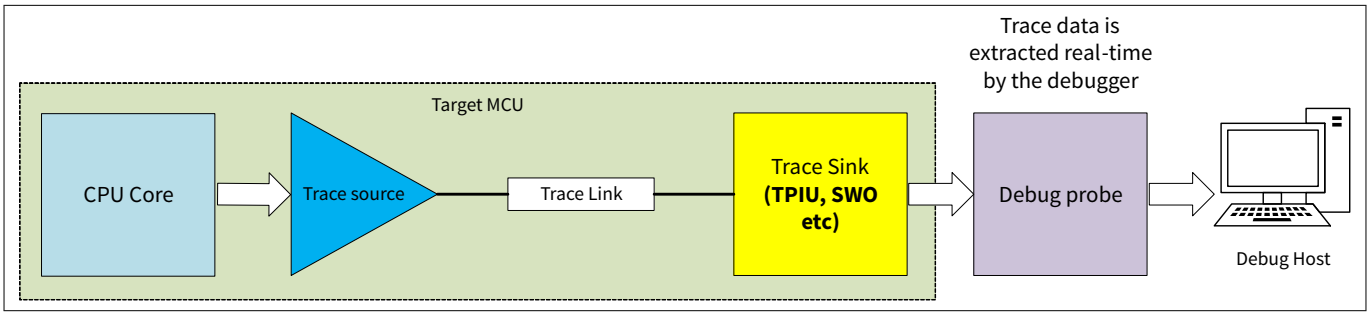


Figure 3 Trace architecture for off-chip capture

2.3 Trace infrastructure examples

Trace components are highly design-configurable; the choice of components to use is also left to the MCU vendors. This section shows a few trace infrastructure examples that can be implemented during design.

2.3.1 Single-core, off-chip ETM trace example

The ETM generates both instruction and data traces for the core. The ETF buffers the data before sending it to the TPIU component, which then forwards the data to the external debugger in real time using the data and clock TPIU pins.

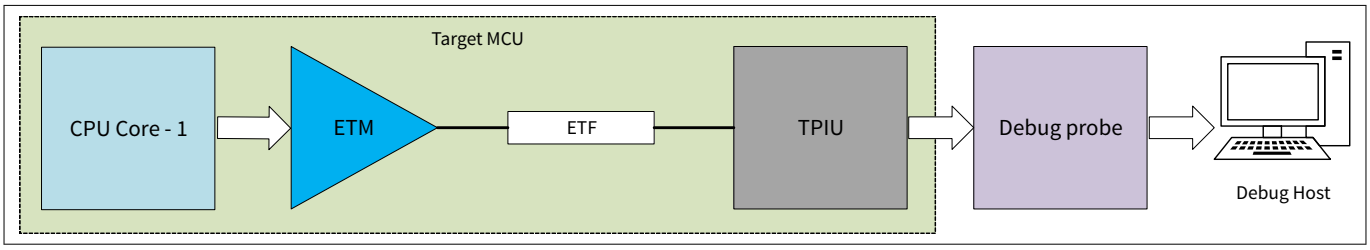


Figure 4 Trace architecture for single-core, off-chip ETM trace

2.3.2 Multi-core, off-chip ETM trace example

The funnel merges ETM data from multiple cores into one ATB. The data is then routed through the ETF, TPIU, and finally to the debugger.

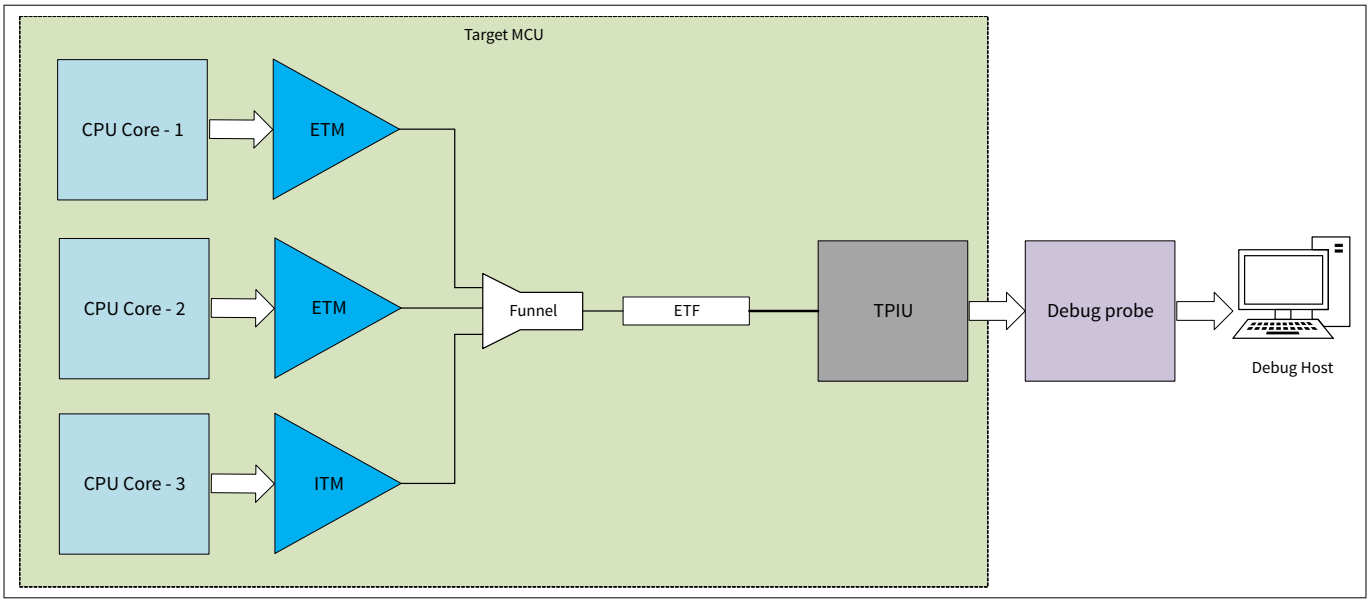


Figure 5 Trace architecture for multi-core, off-chip ETM trace

2.3.3 Multi-core, off-chip ETM CTI trace example

This example is similar to the previous multiple-core example, except that now triggers are routed between different trace components. For instance, the ETF sends a halt request to the cores when the FIFO memory starts to overflow. This halt request halts the cores, allowing the debugger to collect the trace data. Once the FIFO has free space, the ETF sends a resume request to start the cores again.

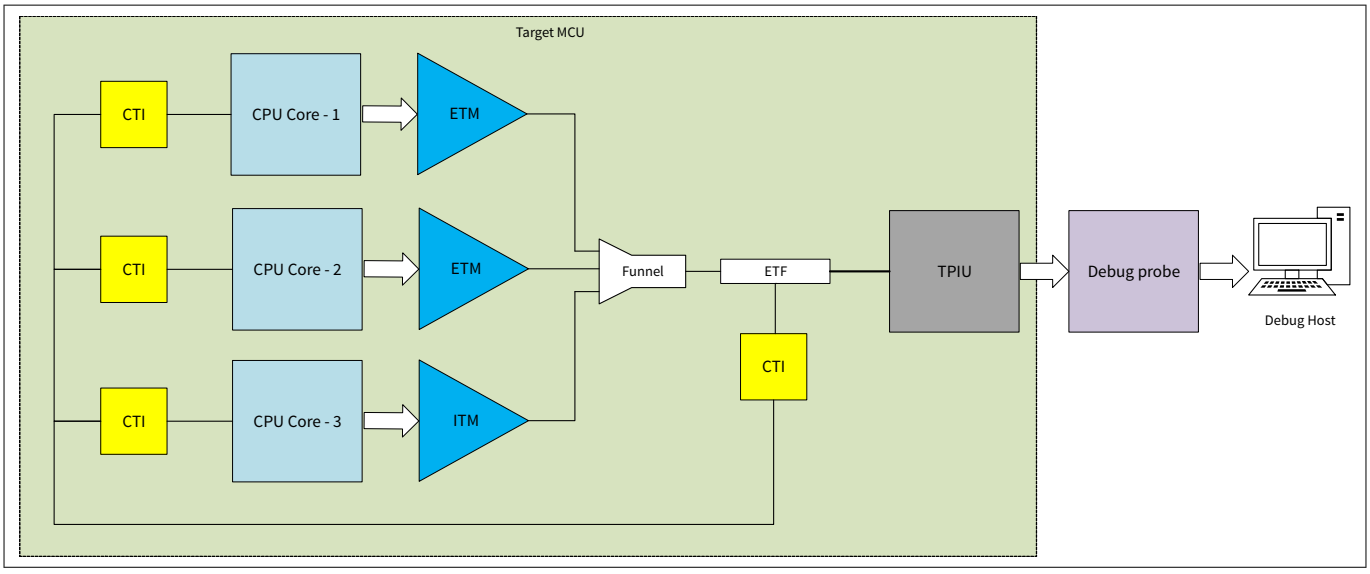


Figure 6 Trace architecture for multi-core, off-chip ETM trace with CTI

2.4 PSOC™ 6 MCU trace infrastructure

The PSOC™ 6 MCU is a dual-CPU microcontroller featuring an Arm® CM4 and a CM0+ core.

The CM4 core supports 4-bit ETM and ITM as trace sources. It features a TPIU port (four data pins and one clock pin) and an SWO pin for outputting trace data to an external debugger. Note that the SWO pin is multiplexed with the JTAG interface, and both cannot be used simultaneously. The TPIU pins can be routed to multiple I/O ports on the PSOC™ 6 MCU; see the device datasheet for more details.

2 General and PSOC™ 6 MCU Arm® trace architecture

The CM0+ CPU supports MTB with 4-KB dedicated SRAM. This application note does not cover MTB tracing.

The PSOC™ 6 MCU also includes an Embedded Cross Trigger (also referred to as CTI) for synchronized debugging and tracing of both CPUs.

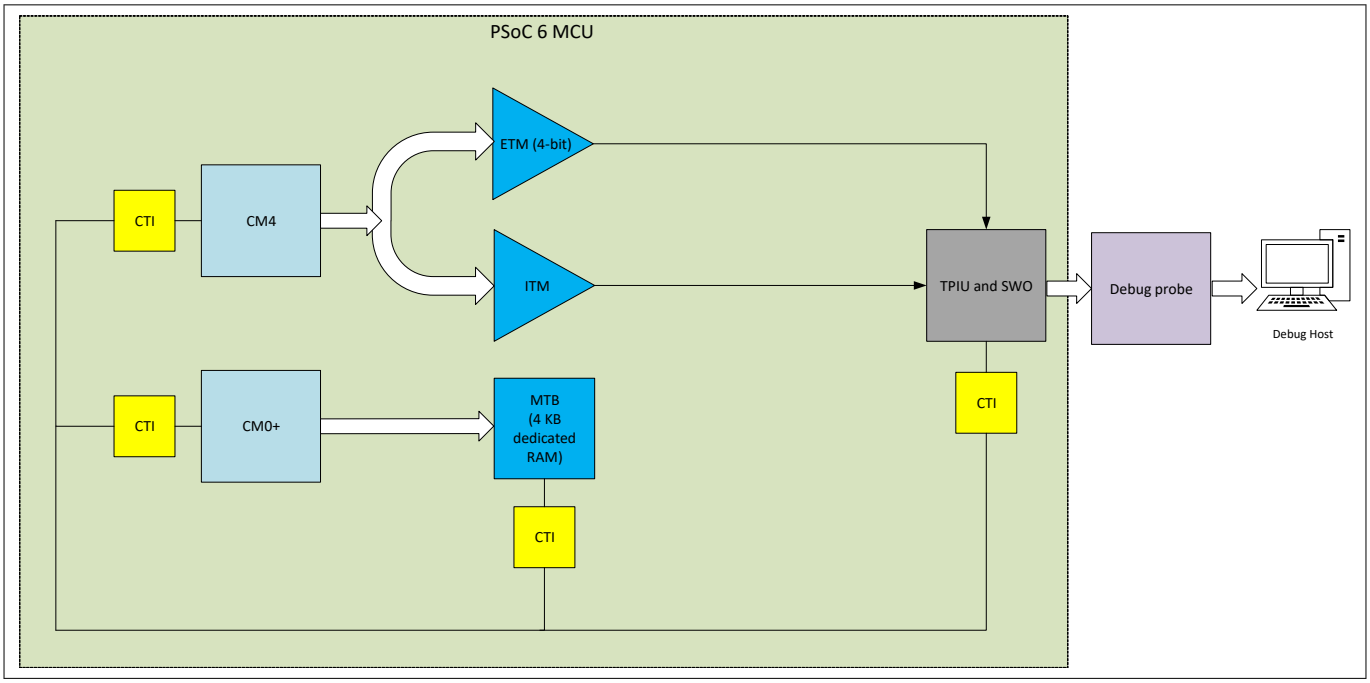


Figure 7 PSOC™ 6 MCU trace architecture

3 Hardware and software requirements

This section lists the hardware and software requirements for performing hands-on trace on PSOC™ 6 MCUs.

3.1 Hardware requirements

3.1.1 Trace probes (external debugger)

This application note uses three trace probes:

- **I-jet Trace for Arm® Cortex® -M**: This probe is used with IAR Embedded Workbench software tools. For more information, see I-jet Trace for Arm® Cortex® -M
- **ULINKpro debug and trace unit**: This probe is used with Keil µVision software tools. For more information, see ULINKpro
- **J-Link/J-Trace**: This probe is used with IAR Embedded Workbench software tools and Keil µVision software tools. see, J-Link/J-Trace

3.1.2 Development board

This application note uses the CY8CEVAL-062S2 evaluation kit. Typically, on evaluation kits, the trace pins are multiplexed with other peripherals due to limited I/Os. To locate the trace pins, open the CY8CEVAL-062S2 schematics file and search for the term "trace". You will find a box labeled "Trace multiplexed pins", as shown in Figure 8.

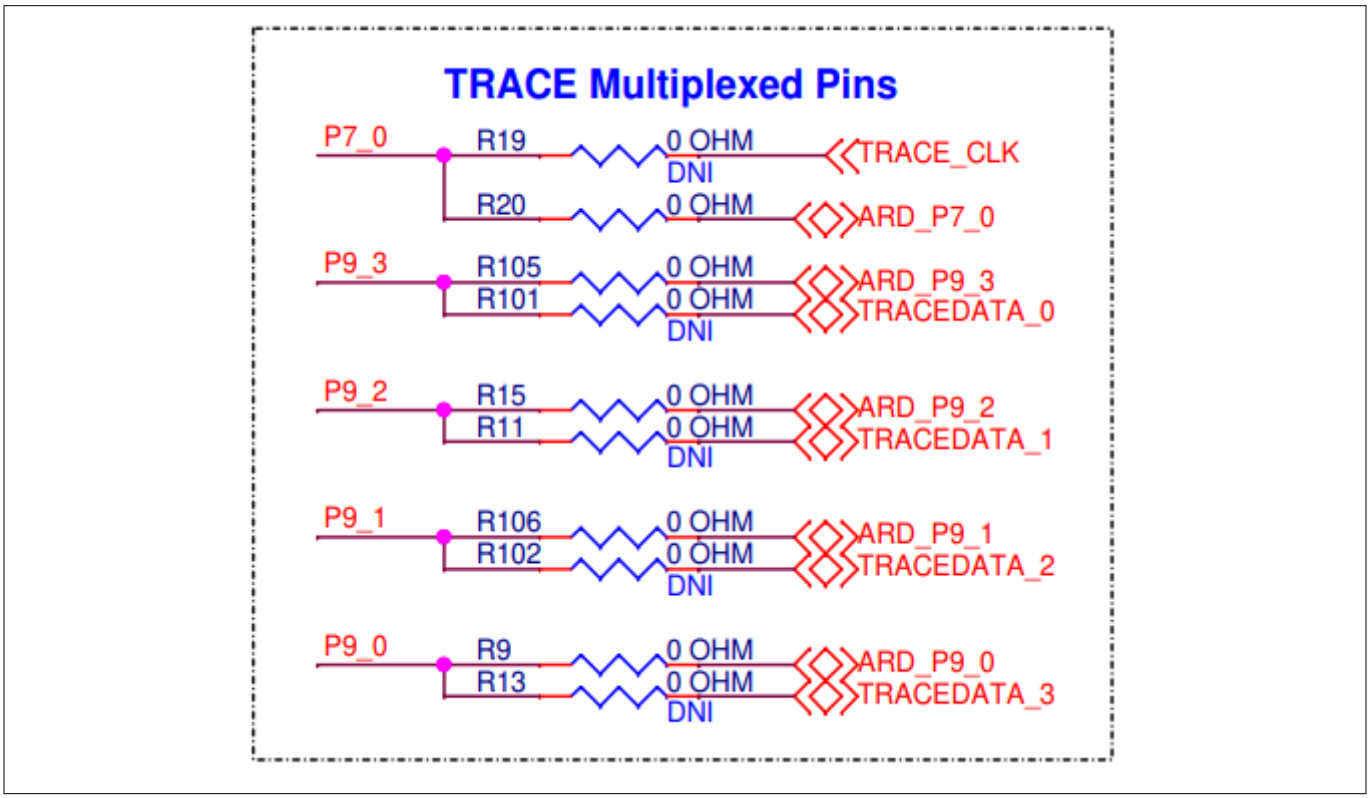


Figure 8 CY8CEVAL-062S2 kit trace pin schematics

In this kit, the trace pins are multiplexed to the Arduino port pins. To expose the trace pins to the J22 ETM header, do the following:

1. Unload the resistances R20, R105, R15, R106, and R9

2. Load 0 Ω resistances to R19, R101, R11, R102, and R13
3. Solder the ETM header (20-pin FRC male connector) in place

***Notes**:*

1. *The CY8CKIT-062-BLE and CY8CKIT-062-WIFI-BT evaluation kits do not require any hardware modifications, as the trace pins are already brought out onto the 20-pin debug header. However, for any other evaluation kit or custom board, perform the necessary hardware modifications to ensure that the trace pins are properly brought out*
2. *PSOC™ 63 Bluetooth® LE devices are not supported for ETM tracing using ULINKpro. However, use other supported trace units, such as I-jet Trace and J-Trace PRO, to perform ETM tracing on PSOC™ 63 Bluetooth® LE devices*

3.2 Software requirements

This application note uses the following software tools:

- ModusToolbox™ software: ModusToolbox™ v3.0 is required to create or import a project for PSOC™ 6 MCUs. This tool is also used to export the project to third-party tools
- IAR Embedded Workbench for Arm®: IAR EW (tested with v8.42.2) and I-jet trace can be used to perform tracing on PSOC™ 6 MCUs
- Keil µ Vision: µVision (tested with v5.36) and ULINKpro can be used to perform tracing on PSOC™ 6 MCUs

4 Performing trace on PSOC™ 6 MCU

4.1 Creating or importing the project using ModusToolbox™

Since ETM and ITM are supported only on the CM4 core of the PSOC™ 6 MCU, the application note will use a single-core project for trace demonstration. The following steps explain how to import the PSOC™ 6 MCU Empty App code example into Eclipse IDE for ModusToolbox™ and configure the project to reserve resources for tracing:

1. Launch the Project Creator GUI tool and select a workspace
2. Once the Project Creator wizard opens, expand PSOC™ 6 BSPs and select **CY8CEVAL-062S2**. Click **Next**
3. In the new window, expand **Getting Started** and select **Empty App**. Click **Create**
4. Once the application is created, open **modus-shell** and go to the project directory. Execute the **make config** command
5. Once the Device Configurator opens, select the **System** tab. Under Resources, select **Debug**
6. Edit the debug parameters as shown in Figure 9 to reserve resources for performing trace. This step is required only to reserve the resources, so that they are not accidently assigned to some other peripheral by the HAL hardware manager. Note that the configuration of the same resources is actually done by the third-party debugger scripts.

   When selecting the trace pins and clock divider, ensure to pick the resources that are defined in the BSP and have aliases starting with CYBSP_. The pins defined in the BSP will match the pins as seen in the Development board section. If you wish to choose some other resources, ensure that the third-party debugger scripts also configure the same resources.

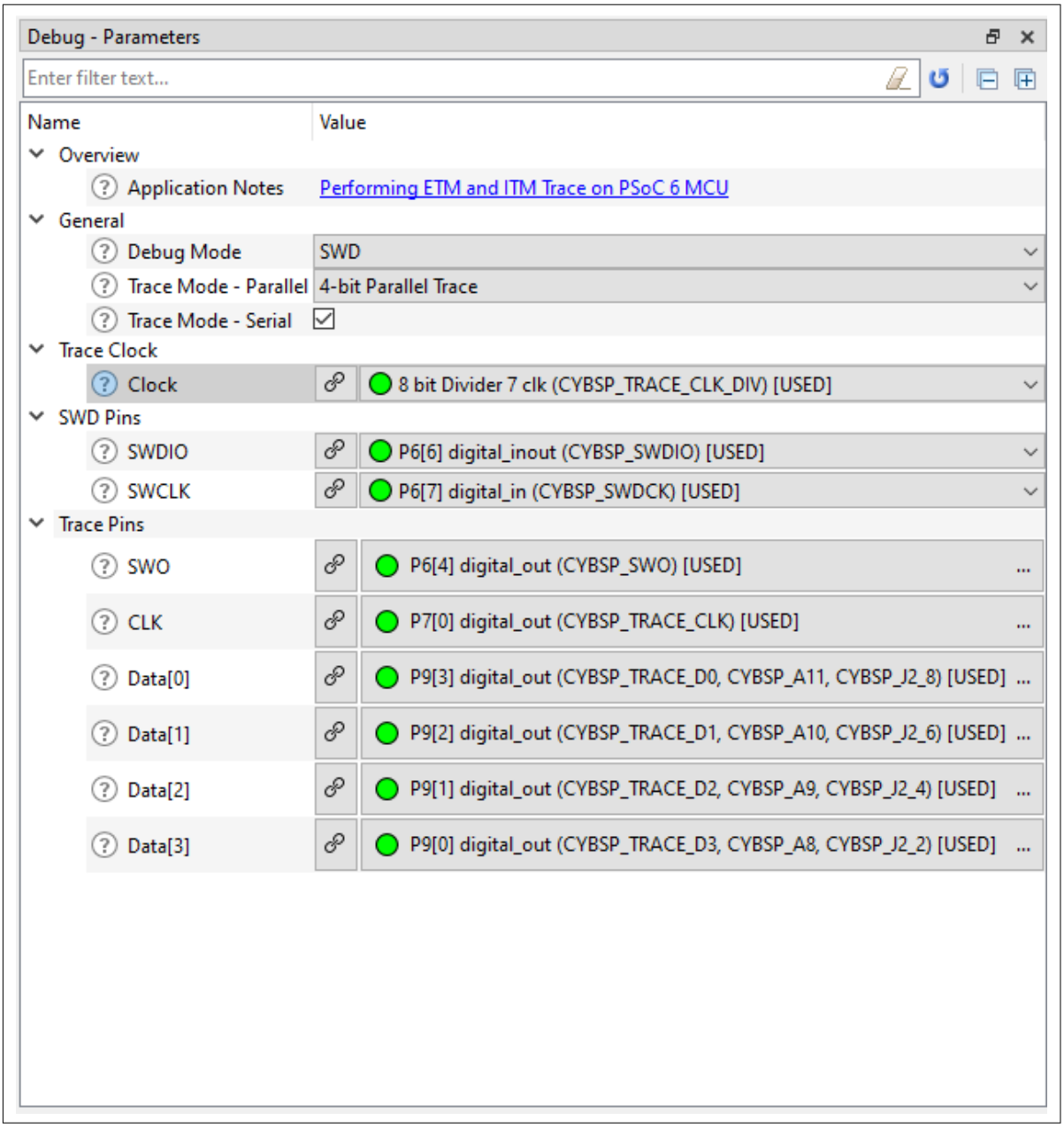


Figure 9 CY8CEVAL-062S2 kit trace pin schematics

7. Click **File > Save**. Close the Device Configurator

For detailed steps on creating a new project in Eclipse IDE for ModusToolbox™, see AN228571: Getting started with PSOC™ 6 MCU on ModusToolbox™ software.

4.2 Performing trace on IAR Embedded Workbench (EW)

4.2.1 Import the ModusToolbox™ project into IAR EW

To export the project from ModusToolbox™ to IAR EW, do the following:

1. Open **modus-shell** and navigate to the **Empty App** application directory
2. Run the following command:

```
make ewarm8 TOOLCHAIN=IAR
```

To import the project into IAR EW, do the following:

1. Start IAR Embedded Workbench
2. On the main menu, select **Project > Create New Project > Empty project** and click **OK**
3. Browse to the **Empty App** application directory created in the Creating or importing the project using ModusToolbox™ section , enter a desired application name, and click **Save**
4. After the application is created, select **File > Save Workspace**. Then, enter a desired workspace name
5. Select **Project > Add Project Connection** and click **OK**
6. On the **Select IAR Project Connection File** dialog, select the `.ipcf` file and click **Open**

   The project will be created in the workspace window
7. Right-click the project and select **Make** to build it

4.2.2 Configure the debugger script and debugger

The debugger configures the TPIU port and necessary clocks before starting the trace. This configuration is done through functions listed in a debugger script. The debugger scripts for PSOC™ 6 MCUs are located in the IAR EW installation directory: `<IAR Installation path> \Embedded Workbench 8.4\arm\config\debugger\Infineon\PSoC6`. The debugger scripts have the `.dmac` extension.

The debugger scripts have default values for the TPIU port and clock divider selection. However, the default value for the TPIU port is usually not applicable for all development kits. Therefore, the script file must be edited to choose the correct TPIU port.

The CY8CEVAL-062S2 evaluation kit uses the CY8C62xA PSOC™ 6 MCU device. To configure the TPIU port, open the `CY8C6xxA_CM4.dmac` debugger script from the IAR EW installation directory (`<IAR Installation path> \Embedded Workbench 8.4\arm\config\debugger\Infineon\PSoC6`). In the file, search for the `_TRACE_path` parameter.

The `_TRACE_path` parameter in the script determines the GPIO port to configure before starting trace. The parameter and its comment are shown in Figure 10.

```
//  The parameter could be any combination of TraceDx routes:
//  0x0000 - All Trace Data pins routed to port 7
//  0x1111 - All Trace Data pins routed to port 9
//  0x2222 - All Trace Data pins routed to port 10
//  0x0001 - TraceD3..TraceD1 pins routed to port 7 and TraceD0
//           routed to port 9. Configuration for CY8CKIT-062-BLE board. (default)
//
__param _TRACE_path = 0x0001;
```

Figure 10 IAR debugger script snapshot for PSOC™ 6 MCU

As seen in the Development board section, the trace data pins are routed to **Port 9** of the kit. Therefore, the default value of **0x0001** for the `_TRACE_path` parameter is not correct in this case and must be modified.

To override the parameter values in the debugger script without editing the file directly, do the following:

1. In the IAR EW, go to **Project > Options > Debugger > Extra Options** and select **Use command line options**
2. In the text box, enter the following command:

```
macro_param _TRACE_path=0x1111
```

Note that this command is space-sensitive; extra spaces should not be added around the equals sign.

You can modify any other parameter in the script in a similar way if required. For the flow followed in this application note, modifying the trace path is sufficient.

To select the debugger, in the IAR EW, go to **Project > Options > Debugger > Setup** and select **Driver(I-jet or J-Link/J-Trace)** from the **Driver** drop-down menu.

4.2.3 Perform ETM trace

In the `main.c` file, modify the `main` function as shown in Code Listing 1. This modification will toggle the user LED on the kit every second.

Code Listing 1

```
int main(void)
{
  cy_rslt_t result;
  /* Initialize the device and board peripherals */
  result = cybsp_init() ;
  if (result != CY_RSLT_SUCCESS)
  {
    CY_ASSERT(0);
  }
  __enable_irq();
  /* Initialize the User LED */
  result = cyhal_gpio_init(CYBSP_USER_LED, CYHAL_GPIO_DIR_OUTPUT,
                           CYHAL_GPIO_DRIVE_STRONG, CYBSP_LED_STATE_OFF);

  /* GPIO init failed. Stop program execution */
  if (result != CY_RSLT_SUCCESS)
  {
    CY_ASSERT(0);
  }
  for (;;)
  {
    cyhal_system_delay_ms(1000);

    /* Invert the USER LED state */
    cyhal_gpio_toggle(CYBSP_USER_LED);
  }
}
```

To start the ETM trace, follow these steps:

1. Click the **Download and Debug** icon on the IAR EW toolbar to program the image and start the debug perspective

2. To open ETM Trace Window, do the following:
   a. For I-jet, select **I-jet > ETM trace**. The ETM trace window appears
   b. For J-Link/J-Trace, select **J-Link/J-Trace> ETM trace**. The ETM trace window appears
3. Right-click the ETM trace window and select **Enable**
4. Click the **Go** icon on the toolbar to start the execution and collection of ETM trace data

You should see the user LED toggling every second. The debug log window is shown in Figure 11.

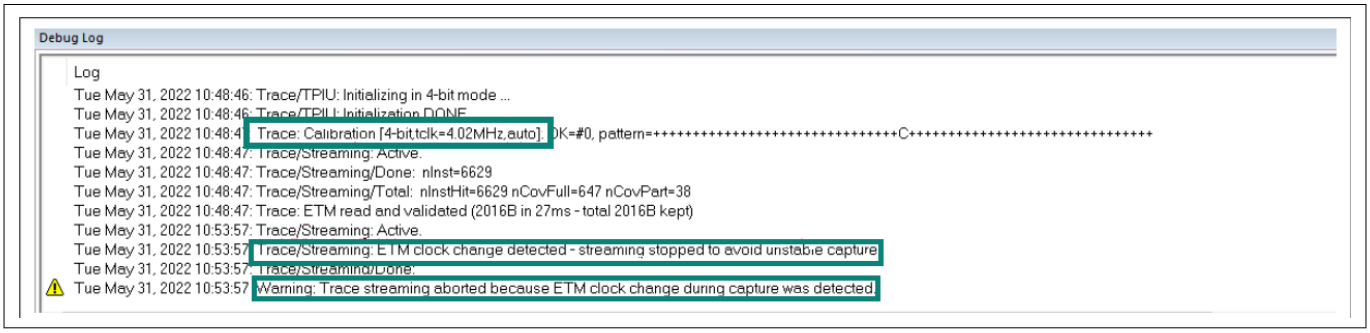


Figure 11 IAR debug log for ETM trace abortion warning

The trace calibration detects a clock of 4 MHz before entering the `main` function. This is because the debug scripts configure only the clock divider and not the FLLs/PLLs that generate the high-frequency peripheral clock from the internal main oscillator (IMO – 8 MHz). Therefore, when the code just starts executing from the beginning of the `main` function, the FLLs/PLLs are disabled, and the IMO is routed directly to the high-frequency clock. The debugger, by default, applies a **divide-by-2** value to the clock, resulting in 4 MHz.

Once the code executes the `cybsp_init()` function, the FLLs/PLLs are enabled, and the high-frequency peripheral clock now have a value of 100 MHz. The debugger now sees a different clock and stops tracing. The logs highlighted in the previous **Debug log** window report the same.

The ideal way to trace is to set a breakpoint right after the `cybsp_init()` function. Once the breakpoint is hit, tracing stops. Now, if you restart the trace, the debugger recalibrates to the new frequency and starts capturing data correctly. See Figure 12.

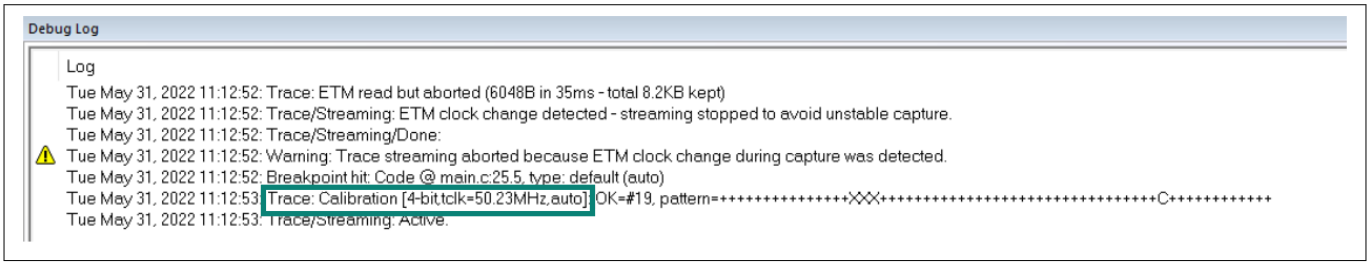


Figure 12 IAR debug log for ETM trace recalibration

If you pause the trace, the trace data collected will be populated in the **ETM trace** window as shown in Figure 13.

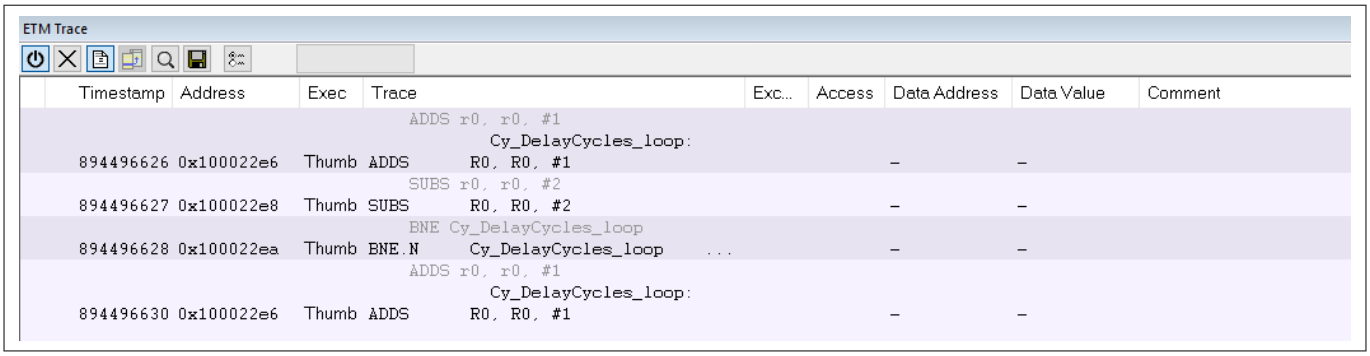


Figure 13 IAR ETM trace window showing ETM data

4.2.4 Perform ITM trace (printf-style debugging)

4.2.4.1 Perform ITM trace using I-jet

1. In IAR EW, go to **Project > Options > I-jet > Trace** and select **Serial (SWO)** under the **Mode** drop-down list. Uncheck the **Allow ETB** checkbox
2. Next, go to **Project > Options > General Options > Library Configuration**. Select **Semihosted**, **Via SWO** and select the **Use CMSIS** checkbox

In the `main.c` file, modify the `main` function and header list to include the `printf` statement as shown in Code Listing 2.

4 Performing trace on PSOC™ 6 MCU

Code Listing 2

```
#include "cy_pdl.h"
#include "cyhal.h"
#include "cybsp.h"
#include "stdio.h"

int main(void)
{
  cy_rslt_t result;

  /* Initialize the device and board peripherals */
  result = cybsp_init() ;
  if (result != CY_RSLT_SUCCESS)
  {
    CY_ASSERT(0);
  }

  __enable_irq();

  /* Initialize the User LED */
  result = cyhal_gpio_init(CYBSP_USER_LED, CYHAL_GPIO_DIR_OUTPUT,
                           CYHAL_GPIO_DRIVE_STRONG, CYBSP_LED_STATE_OFF);

  /* GPIO init failed. Stop program execution */
  if (result != CY_RSLT_SUCCESS)
  {
    CY_ASSERT(0);
  }

  int i = 0;

  for (;;)
  {
    cyhal_system_delay_ms(1000);

    /* Invert the USER LED state */
    cyhal_gpio_toggle(CYBSP_USER_LED);

    printf("Hello empty project - %d\n", i++);
  }
}
```

To start the trace, do the following:

1. Click the **Download and Debug** icon on the IAR EW toolbar to program the image and start the debug perspective
2. Go to **I-jet > SWO Configuration** and configure the ITM Stimulus Ports as shown in Figure 14

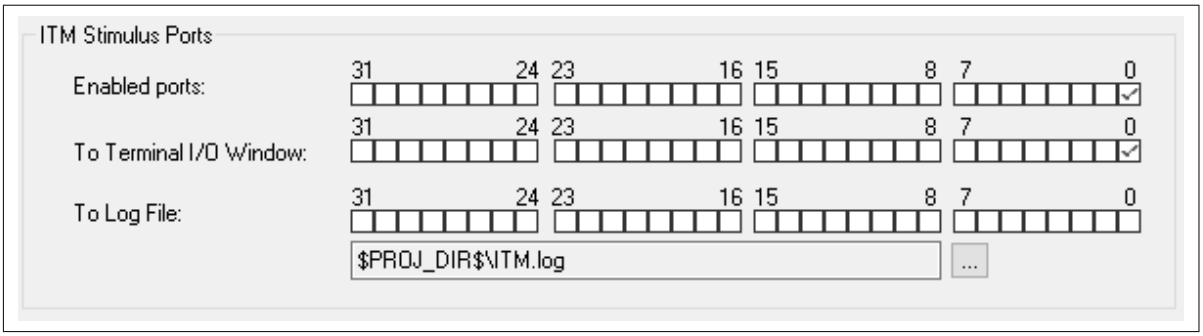


Figure 14 IAR SWO configuration for ITM

3. Go to the **View** tab and select **Terminal I/O**. A Terminal I/O window will open, where you will see print messages
4. Click the **Go** icon on the toolbar to start execution and collection of ITM data

You should see the user LED toggling every second. The Terminal I/O window displays logs as shown in Figure 15.

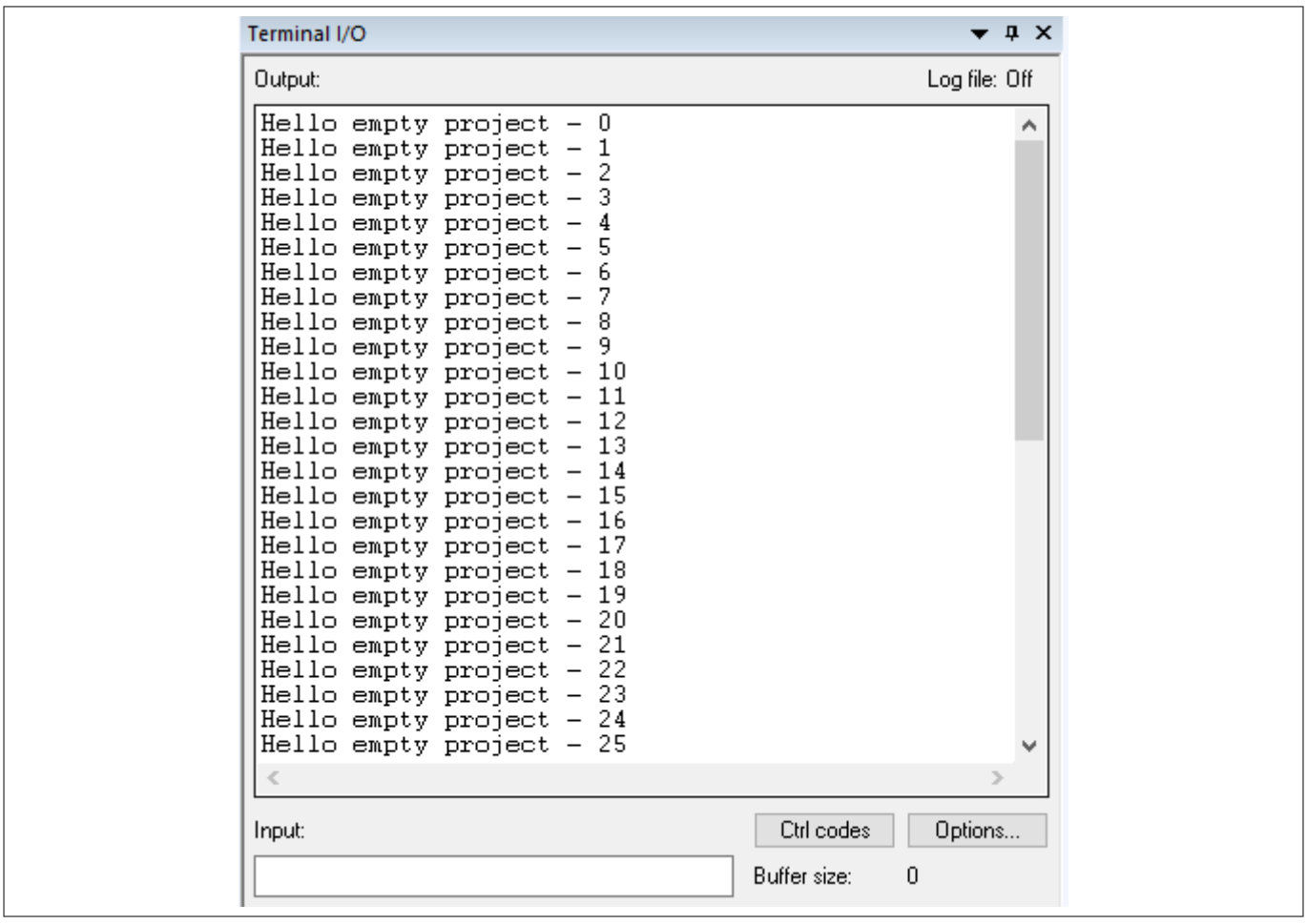


Figure 15 IAR terminal I/O window showing printf output

ITM_SendChar() for Trace

When a project incorporates the retarget-io library, the traditional debugging method utilizing the `printf()` function may be ineffective. To debug in this scenario, it is recommended to utilize the `ITM_SendChar()` function, which is specifically designed for projects that have already implemented UART monitoring through the retarget-io library. This approach enables effective debugging and logging capabilities in such scenarios.

4 Performing trace on PSOC™ 6 MCU

In the `main.c` file, modify the `main` function and header list to include the `ITM` statement as shown in Code Listing 3.

Code Listing 3

```
#include "cy_pdl.h"
#include "cyhal.h"
#include "cybsp.h"
#include "stdio.h"

int main(void)
{
  cy_rslt_t result;

  /* Initialize the device and board peripherals */
  result = cybsp_init() ;
  if (result != CY_RSLT_SUCCESS)
  {
    CY_ASSERT(0);
  }

  __enable_irq();

  /* Initialize the User LED */
  result = cyhal_gpio_init(CYBSP_USER_LED, CYHAL_GPIO_DIR_OUTPUT,
                           CYHAL_GPIO_DRIVE_STRONG, CYBSP_LED_STATE_OFF);

  /* GPIO init failed. Stop program execution */
  if (result != CY_RSLT_SUCCESS)
  {
    CY_ASSERT(0);
  }

  int i = 0;

  for (;;)
  {
    cyhal_system_delay_ms(1000);

    /* Invert the USER LED state */
    cyhal_gpio_toggle(CYBSP_USER_LED);

     ITM_SendChar('H');
  }
}
```

1. Click the **Download and Debug** icon on the IAR EW toolbar to program the image and start the debug perspective
2. Go to **I-jet > SWO Configuration** and configure it as shown in Figure 16

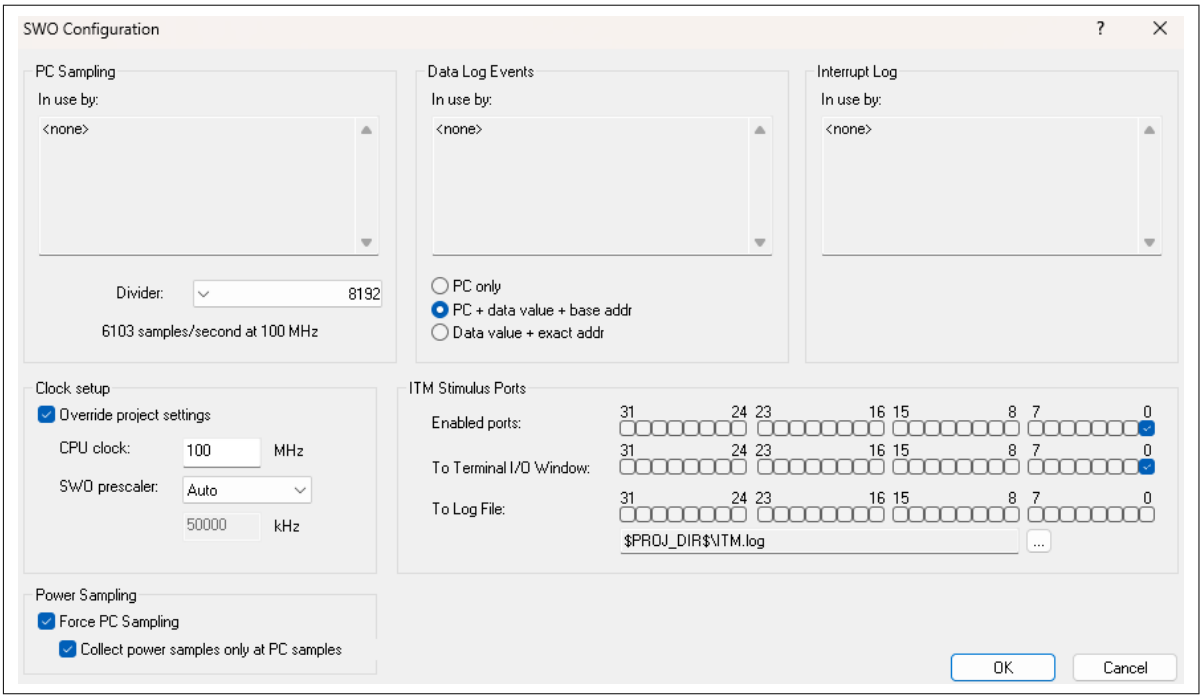


Figure 16 SWO Configuration

3. Go to **View** tab and select **Terminal I/O**. A Terminal I/O window will open, where you will see print messages
4. Click the **Go** icon on the toolbar to start the execution and collection of **ITM data**
5. You should see the user LED toggling every second. The Terminal I/O window displays logs as shown in Figure 17

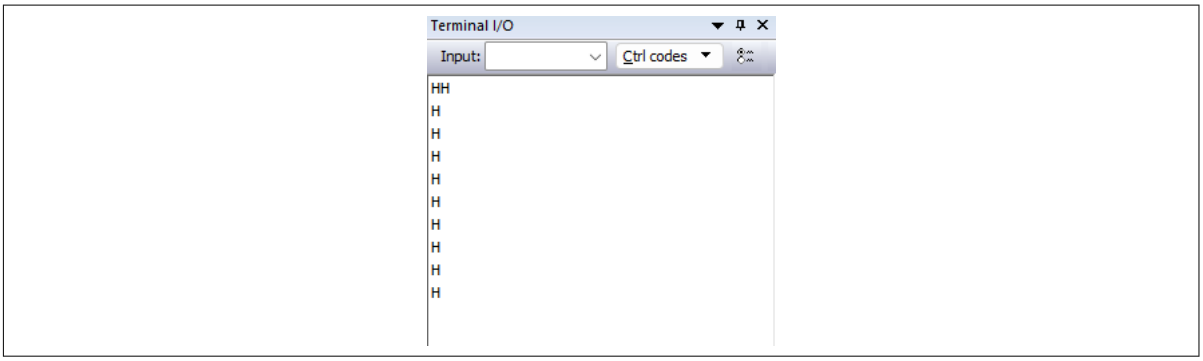


Figure 17 IAR terminal I/O window showing printf output

4.2.4.2 Perform ITM trace using J-Link/J-Trace

1. In IAR EW, go to **Project > Options > I-jet > Trace** and select **Serial (SWO)** under the mode drop-down list. Uncheck the **Allow ETB** checkbox
2. Next, go to **Project > Options > General Options > Library Configuration**. Select **Semihosted**, **Via SWO** and select the **Use CMSIS** checkbox
3. In the `main.c` file, modify the main function and header list as per Code Listing 2 in the Perform ITM trace (printf-style debugging) section
4. Click the **Download and Debug** icon on the IAR EW toolbar to program the image and start the debug perspective
5. Go to **J-Link > SWO Configuration** and configure the ITM Stimulus Ports as shown in Figure 18

4 Performing trace on PSOC™ 6 MCU

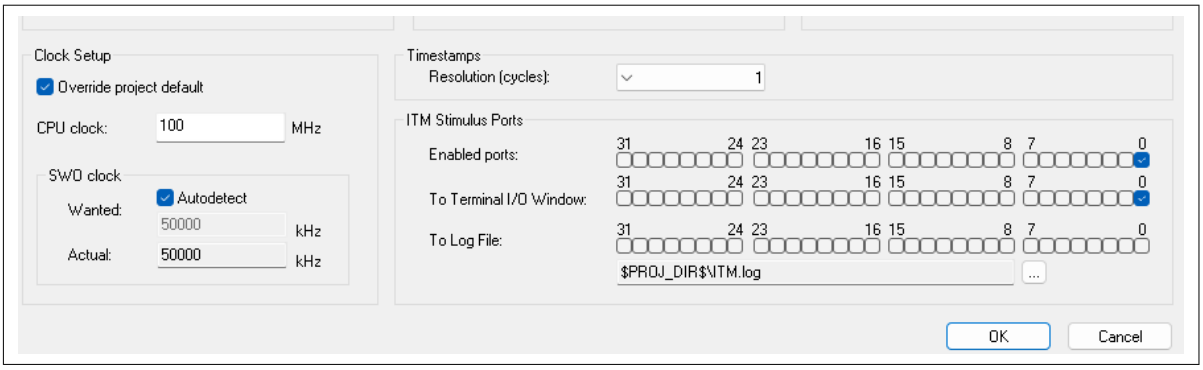


Figure 18 IAR SWO configuration for ITM

6. Go to the **View** tab and select **Terminal I/O**. A Terminal I/O window will open, where you will see print messages
7. Click the **Go** icon on the toolbar to start execution and collection of ITM data

   You should see the user LED toggling every second. The Terminal I/O window displays logs as shown in Figure 19

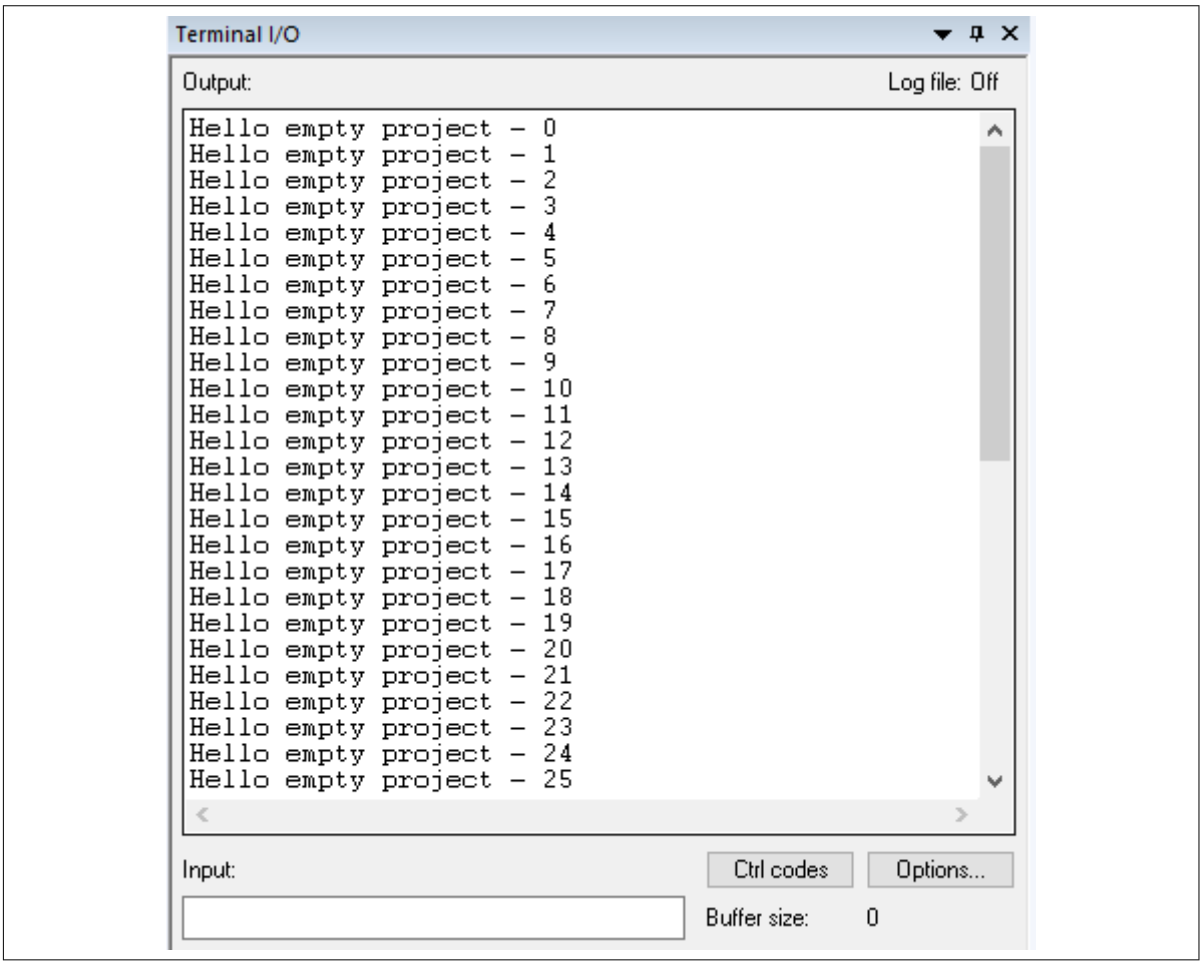


Figure 19 IAR terminal I/O window showing printf output

You can also call the `scanf` function to read from the Terminal I/O window.

ITM_SendChar() for Trace

In the `main.c` file, modify the code as per Code Listing 3 in the Perform ITM trace using I-jet section.

a. Click the **Download and Debug** icon on the IAR EW toolbar to program the image and start the debug perspective

b. Go to **J-Link> SWO Configuration** and configure it as shown in Figure 20

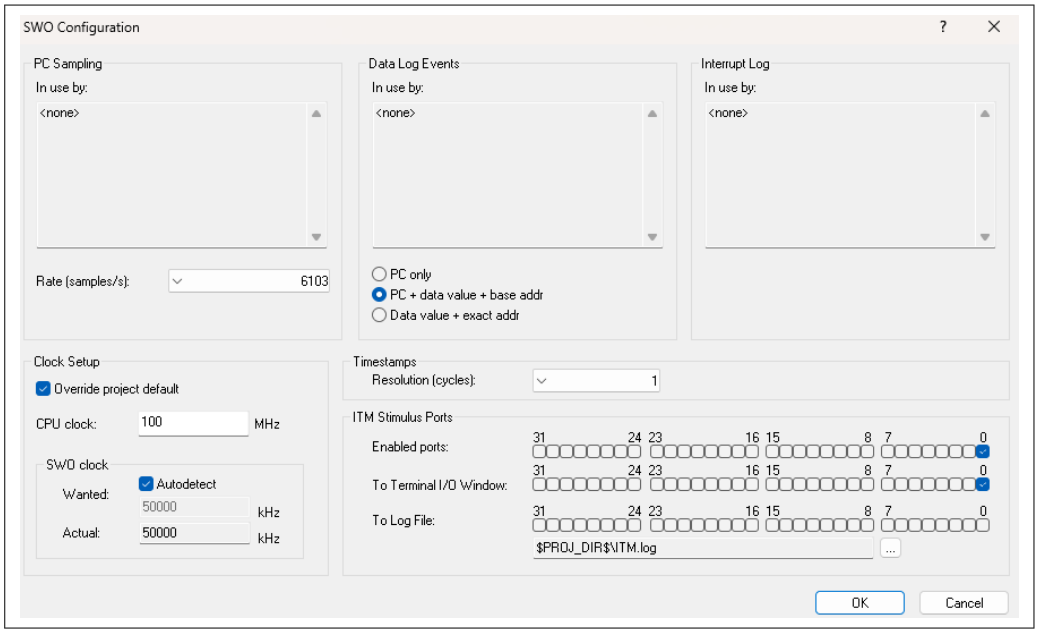


Figure 20 SWO Configuration

c. Go to **View** tab and select **Terminal I/O**. A Terminal I/O window will open, where you will see print messages

d. Click the **Go** icon on the toolbar to start the execution and collection of ITM data

e. You should see the user LED toggling every second. The Terminal I/O window displays logs as shown in Figure 21

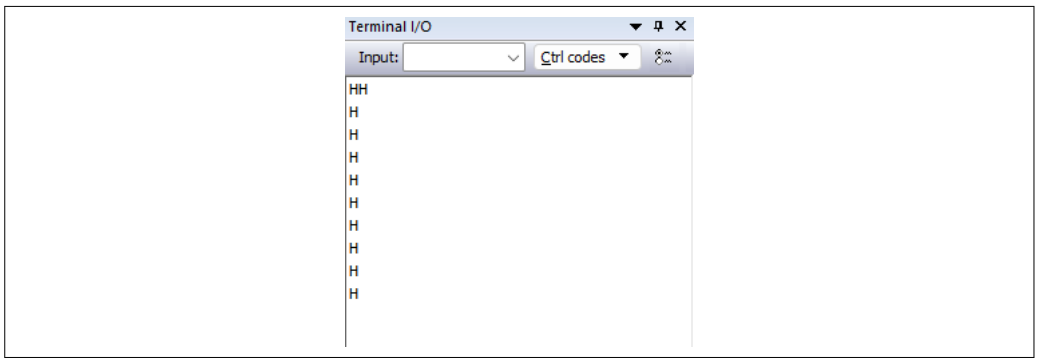


Figure 21 IAR terminal I/O window showing printf output

4.3 Performing trace on Keil µVision

4.3.1 Import the ModusToolbox™ project into Keil µVision

To export the project from ModusToolbox™ to Keil µVision, perform the following steps:

1. Open **modus-shell** and navigate to the `Empty App` application directory created in the Creating or importing the project using ModusToolbox™ section
2. Run the following command:

```
make uvision5 TOOLCHAIN=ARM
```

To import the project into µVision, do the following:

1. Open the `Empty App` directory in File Explorer
2. Double-click the **`*.cprj`** file, which launches the Keil µVision IDE. The first time you do this, a Pack Installer dialog box will appear
3. Follow the on-screen instructions to install the **`Infineon::PSoC6_DFP pack`** (v1.3 or higher). This pack can also be installed separately by launching the Pack Installer tool in µVision. The project will be created in the project window
4. Right-click the project and select **Options for Target <application-name>**
5. Select the **Device** tab. Depending on the kit you are using, select the relevant PSOC™ 6 MCU device under Infineon's list.

   For the CY8CEVAL-062S2 kit, select **Infineon > PSOC™ 62 > CY8C62xA > CY8C624ABZI-S2D44 > CY8C624ABZI-S2D44:Cortex-M0p**. Click **OK**
6. Repeat the previous step and select **Infineon > PSOC™ 62 > CY8C62xA > CY8C624ABZI-S2D44 > CY8C624ABZI-S2D44:Cortex-M4**. Click **OK**
7. At this point, you should see a folder named "`DebugConfig`" created in the `Empty App` project directory. The `DBGCONF` files in this folder contain the TPIU port selection information
8. Open the project options again
9. Select the **C/C++ (AC6)** tab and do the following, then click **OK**:
   - Set Language C to c99
   - Set Warnings to AC5-like Warnings
   - Set Optimizations to -Os balanced
10. Right-click the project and click **Build Target**

4.3.2 Configure the debugger script and debugger

The debugger configures the TPIU port and necessary clocks before starting the trace. The `PSoC6_DFP` pack has the necessary files to perform this configuration. As explained earlier, the `DBGCONF` files in the `DebugConfig` folder, located under the project directory, contain the TPIU port selection information. The default value for the TPIU port is usually not applicable for all development kits; therefore, the `DBGCONF file` must be edited to choose the correct TPIU port.

As seen in the Development board section, the trace data pins are routed to Port 9 of the kit.

4.3.2.1 Configure the debugger script with ULINKpro Cortex® Debugger

To override the default values in the `DBGCONF` file, do the following:

1. In µVision, select **File > Open**
2. Navigate to the `DebugConfig` folder in the `Empty App` project directory, select the `CM4 DBGCONF` file, and click **Open**
3. In the editor that opens with the file content, select **Configuration Wizard** at the bottom of the editor

4 Performing trace on PSOC™ 6 MCU

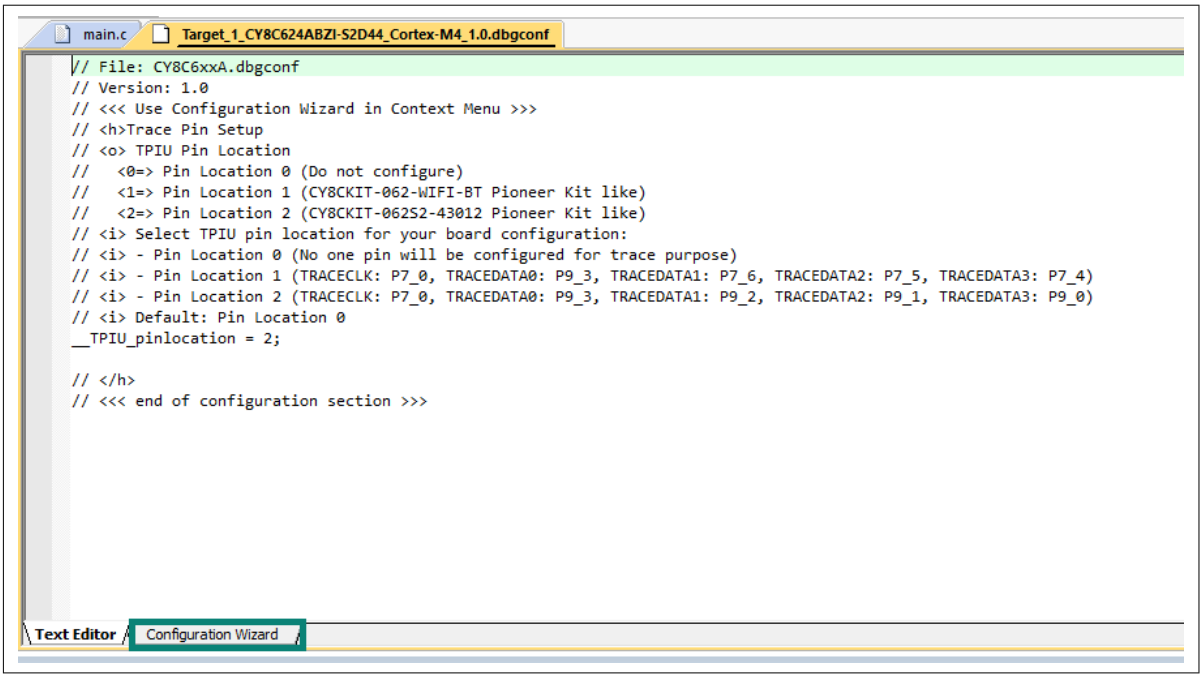


Figure 22 Opening the DBGCONF file in Configuration Wizard

4. Select **Trace Pin Setup > TPIU Pin Location > Pin Location 2** from the drop-down list. Details about the drop-down options are provided at the end of the editor. Once the selection is complete, close the file.

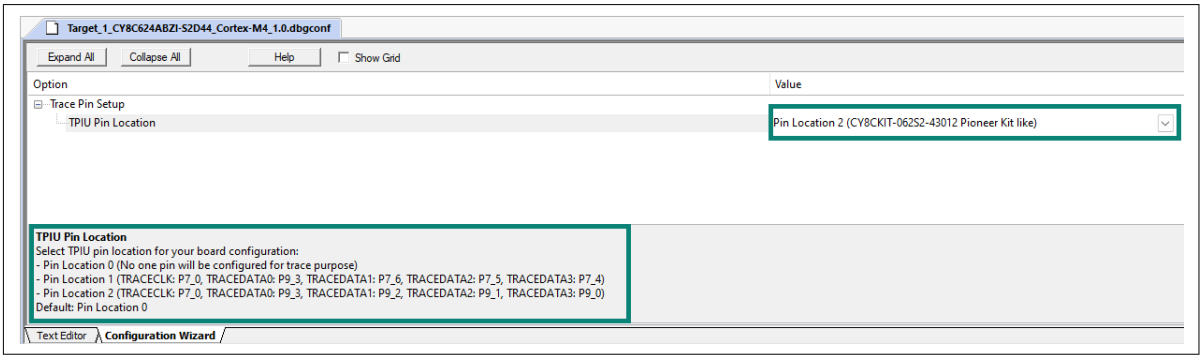


Figure 23 Editing the DBGCONF file in Configuration Wizard

To select and configure the debugger in µVision, do the following:

1. Open the project options
2. Select the **Debug** tab and choose the **ULINKpro Cortex® Debugger** from the drop-down menu
3. Click the **Settings** button next to it

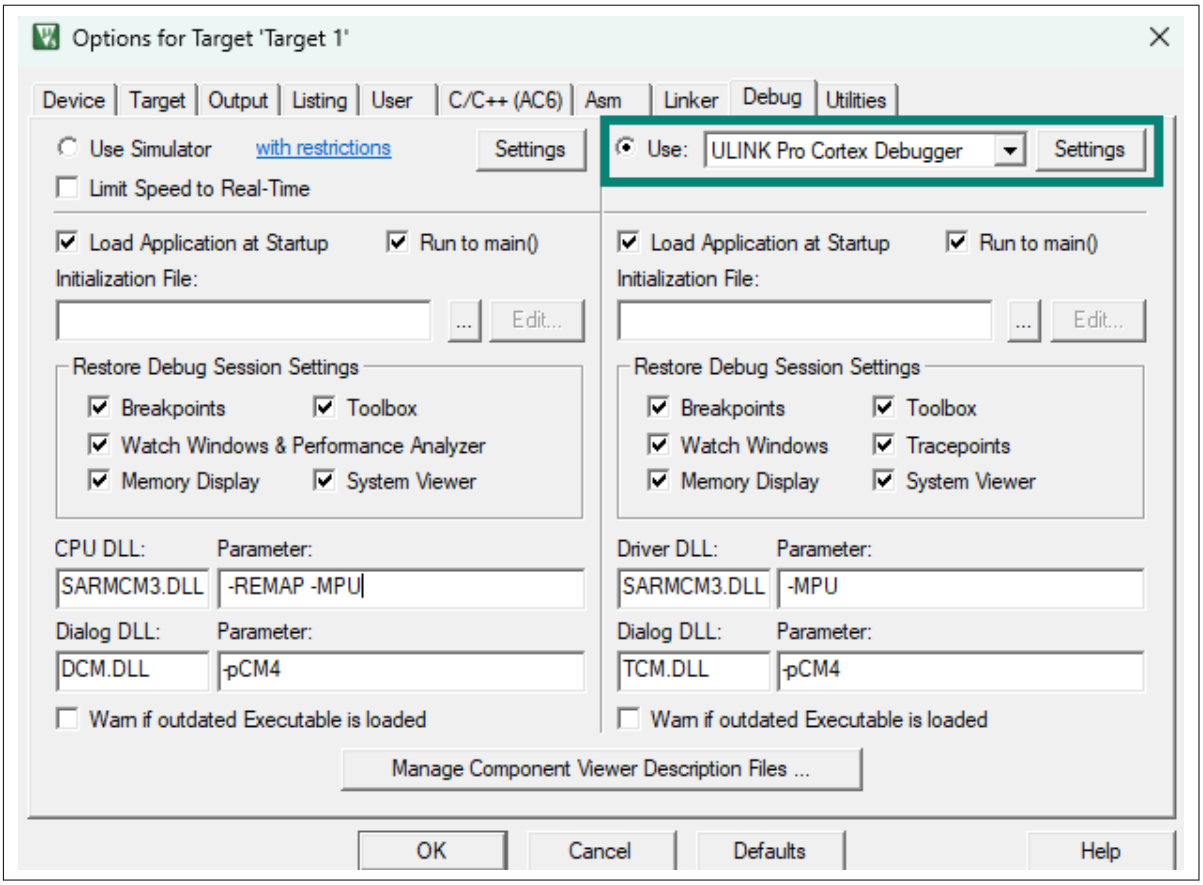


Figure 24 Configurations in Debug tab

4. In the new window, configure the settings of the **Debug** tab to match Figure 25

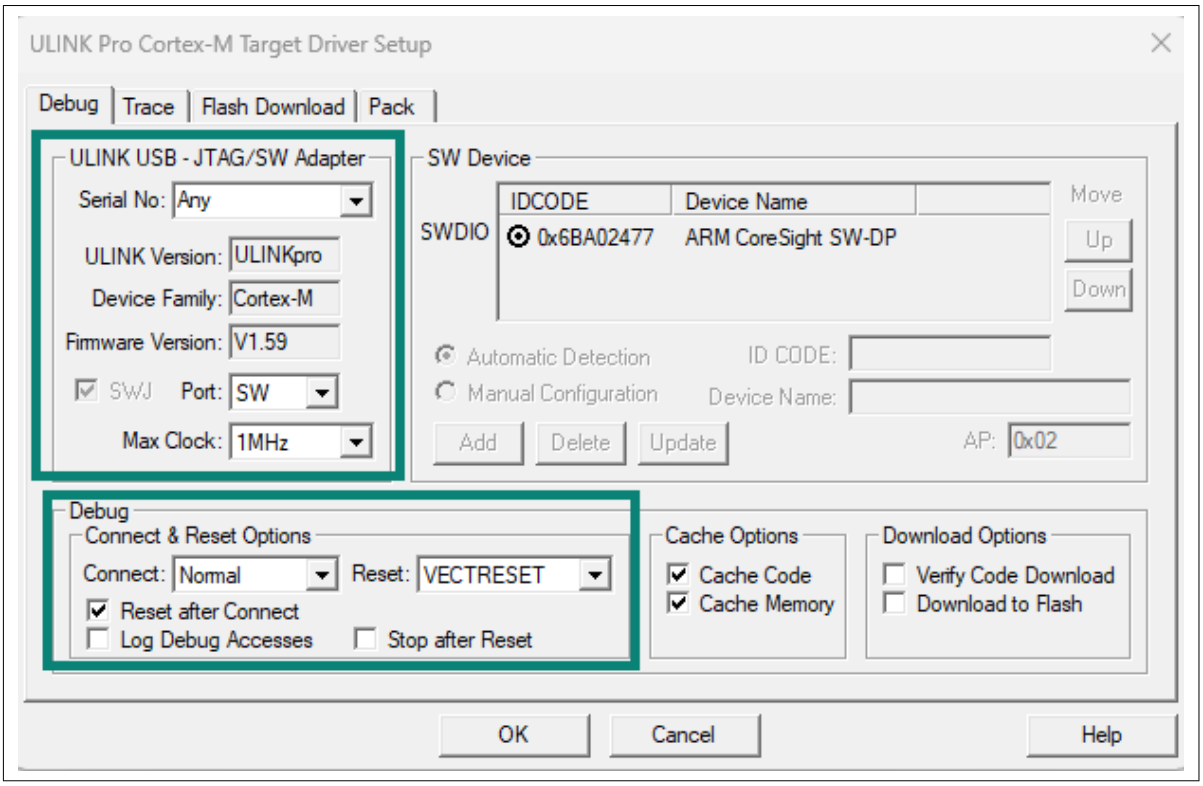


Figure 25 Configurations in Debug > Debug tab

5. Select the **Trace** tab and configure the settings to match Figure 26

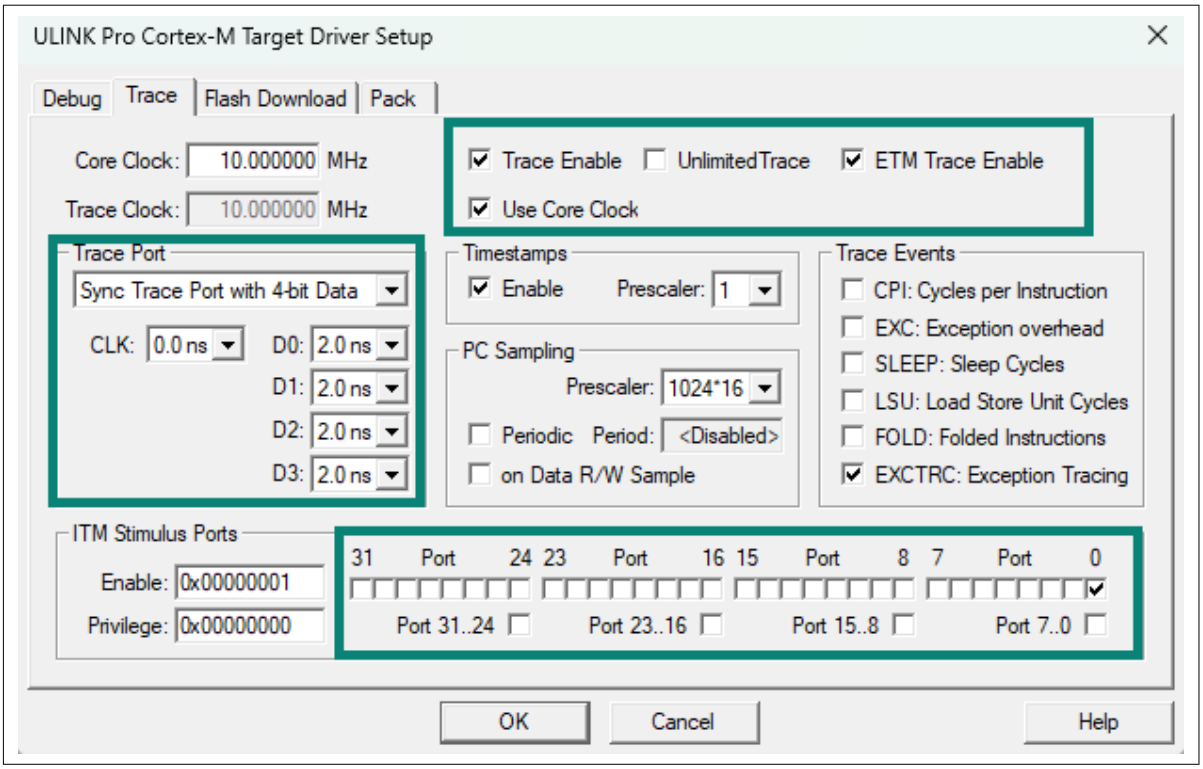


Figure 26 Configurations in Debug > Trace tab

6. Select the **Flash Download** tab and select the checkboxes for **Program**, **Verify**, **Reset**, and **Run**
7. Select the **Pack** tab and **Enable checkbox** option, and then click **OK**

4.3.2.2 Configure the debugger script and debugger with J-Link/J-Trace

To override the default values in the `DBGCONF` file, do the following:

1. In µVision, select **File > Open**
2. Navigate to the `DebugConfig` folder in the `Empty App` project directory, select the `CM4 DBGCONF` file, and click **Open**
3. In the editor that opens with the file content, select **Configuration Wizard** at the bottom of the editor

4 Performing trace on PSOC™ 6 MCU

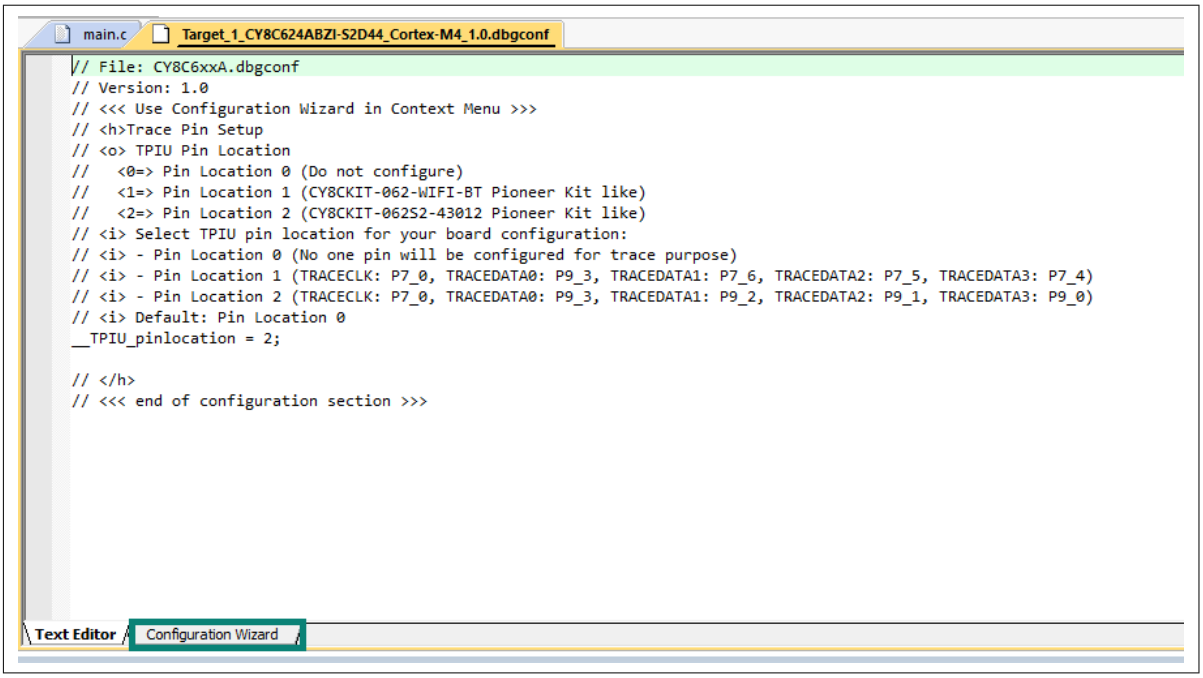


Figure 27 Opening the DBGCONF file in Configuration Wizard

4. Select **Trace Pin Setup > TPIU Pin Location > Pin Location 2** from the drop-down list. Details about the drop-down options are provided at the end of the editor. Once the selection is complete, close the file

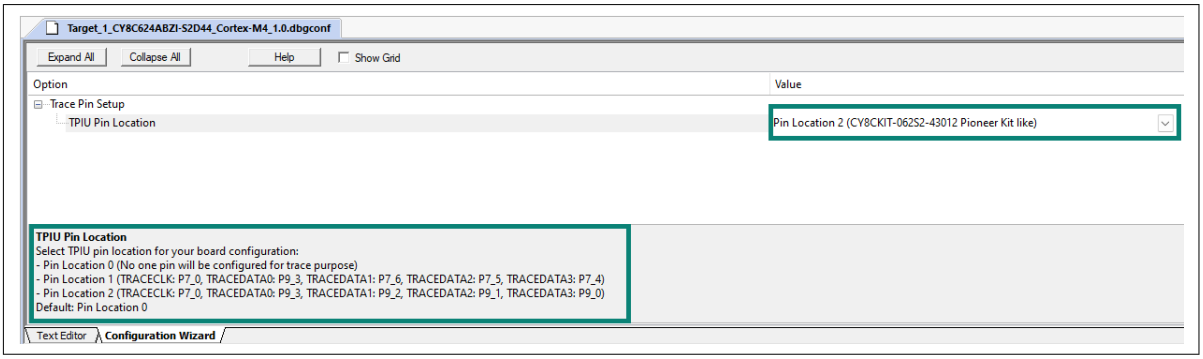


Figure 28 Editing the DBGCONF file in Configuration Wizard

To select and configure the debugger in µVision, do the following:

1. Open the project options
2. Select the **Debug** tab and choose the **J-LINK/J-TRACE Cortex® Debugger** from the drop-down menu
3. Click the **Settings** button next to it

4 Performing trace on PSOC™ 6 MCU

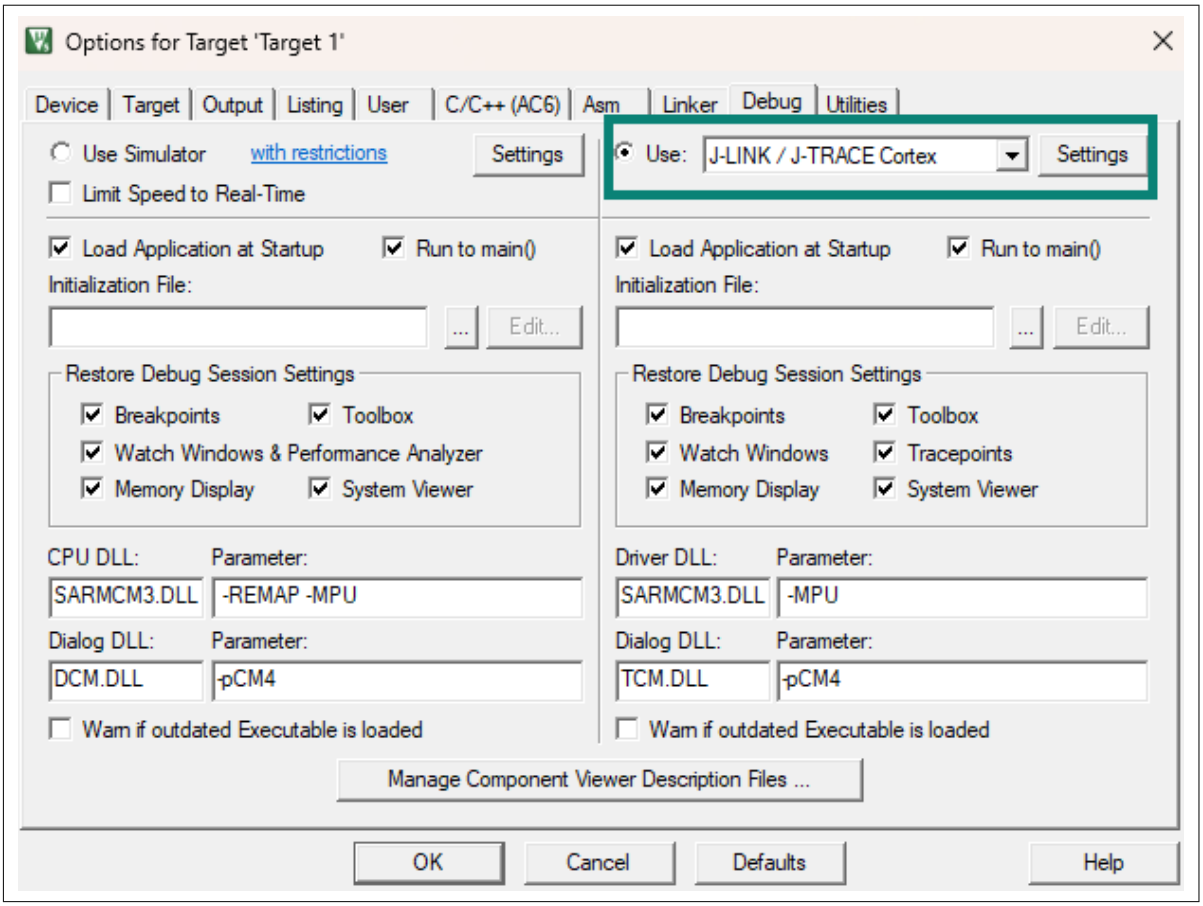


Figure 29 Configurations in Debug tab

4. In the new window, configure the settings of the **Debug** tab to match

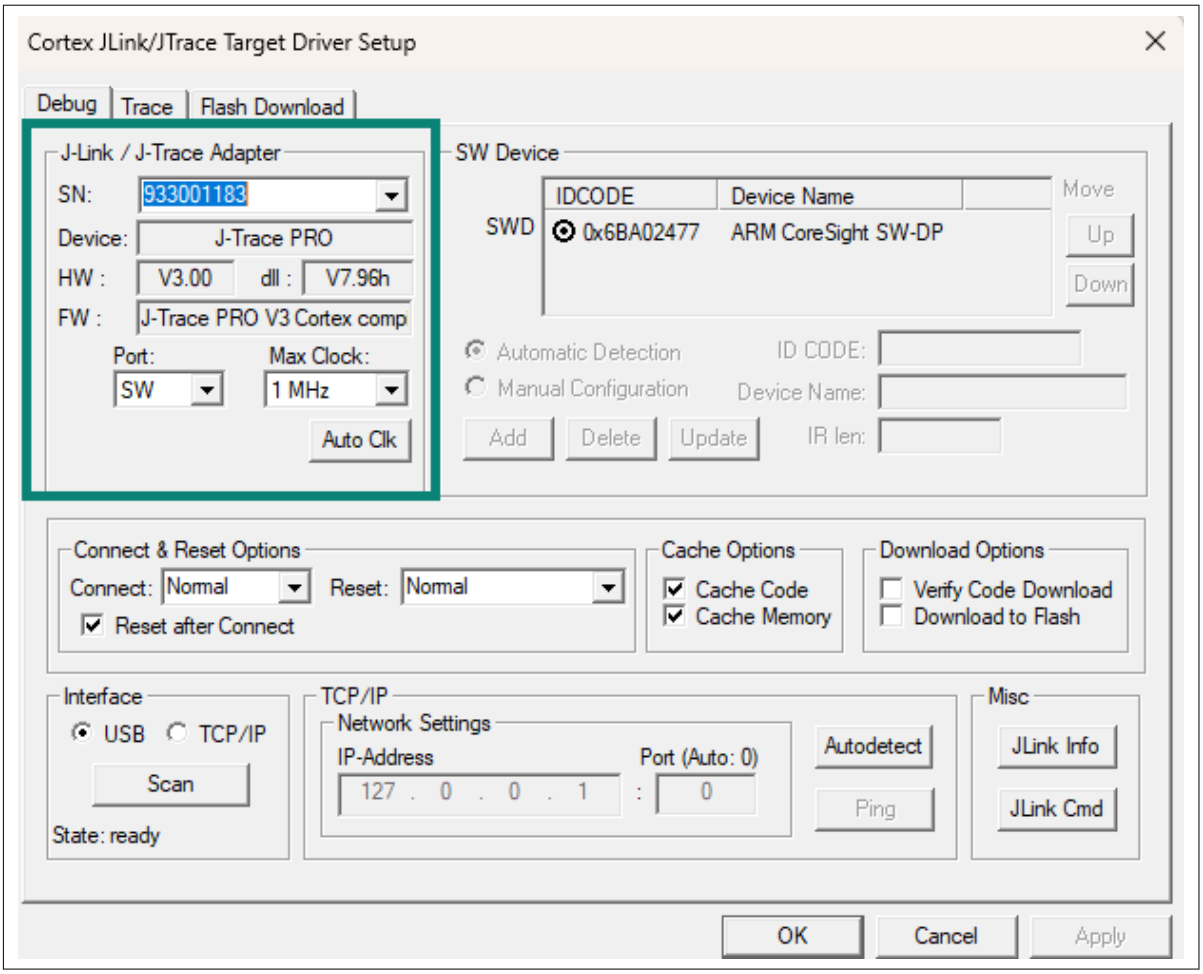


Figure 30 Configurations in Debug > Debug tab

5. Select the **Flash Download** tab and select the checkboxes for **Program**, **Verify**, **Reset** and **Run**
6. Select the **Pack** tab and **Enable checkbox** option, and then click **OK**

4.3.3 Perform ETM trace

In the `main.c` file, modify the code as per Code Listing 1 described in the Perform ETM trace section. This modification will toggle the user LED on the kit every second.

To perform an ETM trace, do the following:

1. Click the **Build** icon on the µVision toolbar to rebuild the code
2. Click the **Start or Stop Debug Session** icon on the µVision toolbar to program the image and start the debug perspective

   To open the ETM trace window, follow these steps:

   a. **For ULINKpro**: Select **View > Trace > Trace Data** to open an ETM trace window
   b. **For J-Link/J-Trace**:
      1. Select the **Trace** tab and configure the settings to match

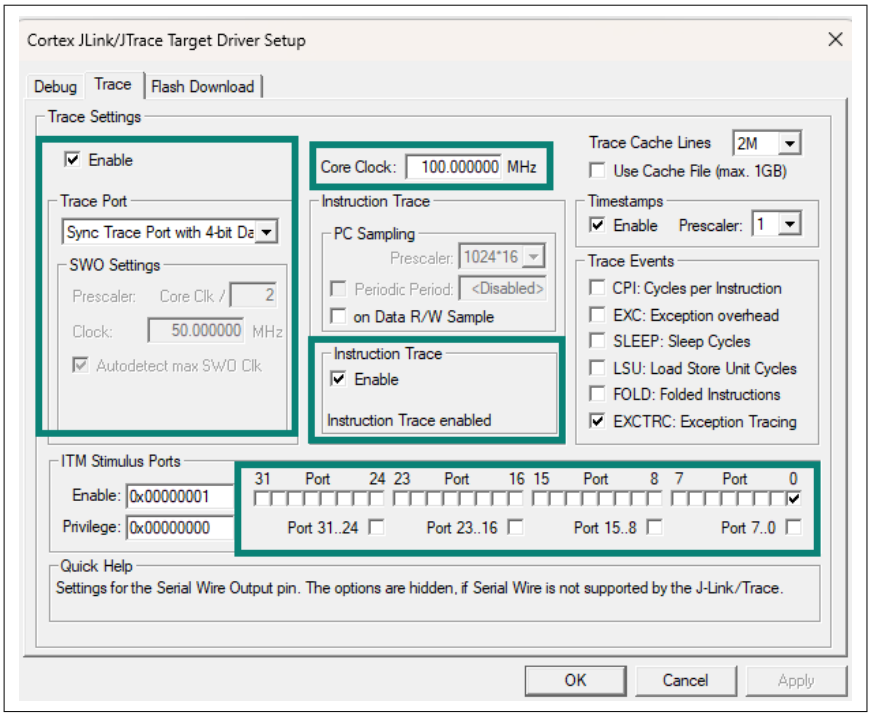
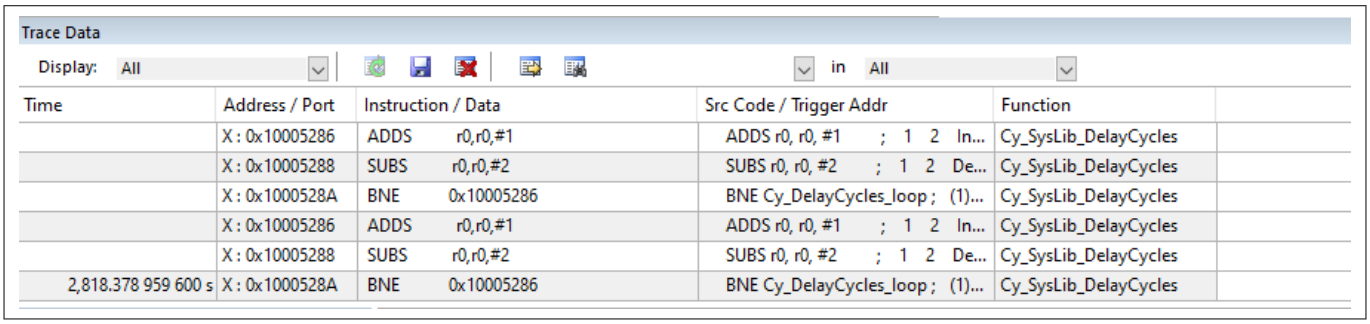


Figure 31 Configurations in Debug > Trace tab

2. Select **View > Trace > Instruction Trace** to open an ETM trace window

3. Click the **Run** icon on the toolbar to start the execution and collection of ETM trace data

You should see the user LED toggling every second. When you pause the trace, the collected trace data will populate in the ETM trace data window, as shown in Figure 32.



Time	Address / Port	Instruction / Data	Src Code / Trigger Addr	Function
	X : 0x10005286	ADDS r0,r0,#1	ADDS r0, r0, #1 ; 1 2 In...	Cy_SysLib_DelayCycles
	X : 0x10005288	SUBS r0,r0,#2	SUBS r0, r0, #2 ; 1 2 De...	Cy_SysLib_DelayCycles
	X : 0x1000528A	BNE 0x10005286	BNE Cy_DelayCycles_loop ; (1)...	Cy_SysLib_DelayCycles
	X : 0x10005286	ADDS r0,r0,#1	ADDS r0, r0, #1 ; 1 2 In...	Cy_SysLib_DelayCycles
	X : 0x10005288	SUBS r0,r0,#2	SUBS r0, r0, #2 ; 1 2 De...	Cy_SysLib_DelayCycles
2,818.378 959 600 s	X : 0x1000528A	BNE 0x10005286	BNE Cy_DelayCycles_loop ; (1)...	Cy_SysLib_DelayCycles

Figure 32 µVision trace data windows showing the ETM data

The ULINKpro debugger automatically calibrates to the new trace clock if the trace clock changes during code execution.

4.3.4 Perform ITM trace (printf-style debugging)

To configure for printf-style debugging, do the following:

1. In µVision, go to **Project** > **Manage** > **Run-Time Environment**
2. Once the **Manage Run-Time Environment** window opens, select **Compiler** > **IO**, and then select the checkboxes for **STDIN** and **STDOUT**. In the drop-down list for STDIN and STDOUT, select **ITM**
3. In the `main.c` file, modify the `main` function and header list as described in the Perform ITM trace (printf-style debugging) section

To start the trace, follow these steps:

1. To perform an ITM trace, follow the steps mentioned below:
   a. For ULINKpro Cortex® Debugger follow the steps described in the Configure the debugger script with ULINKpro Cortex® Debugger section
   b. For J-Link/J-Trace, follow the steps described in the Configure the debugger script and debugger with J-Link/J-Trace section and configure the settings as shown in Figure 33

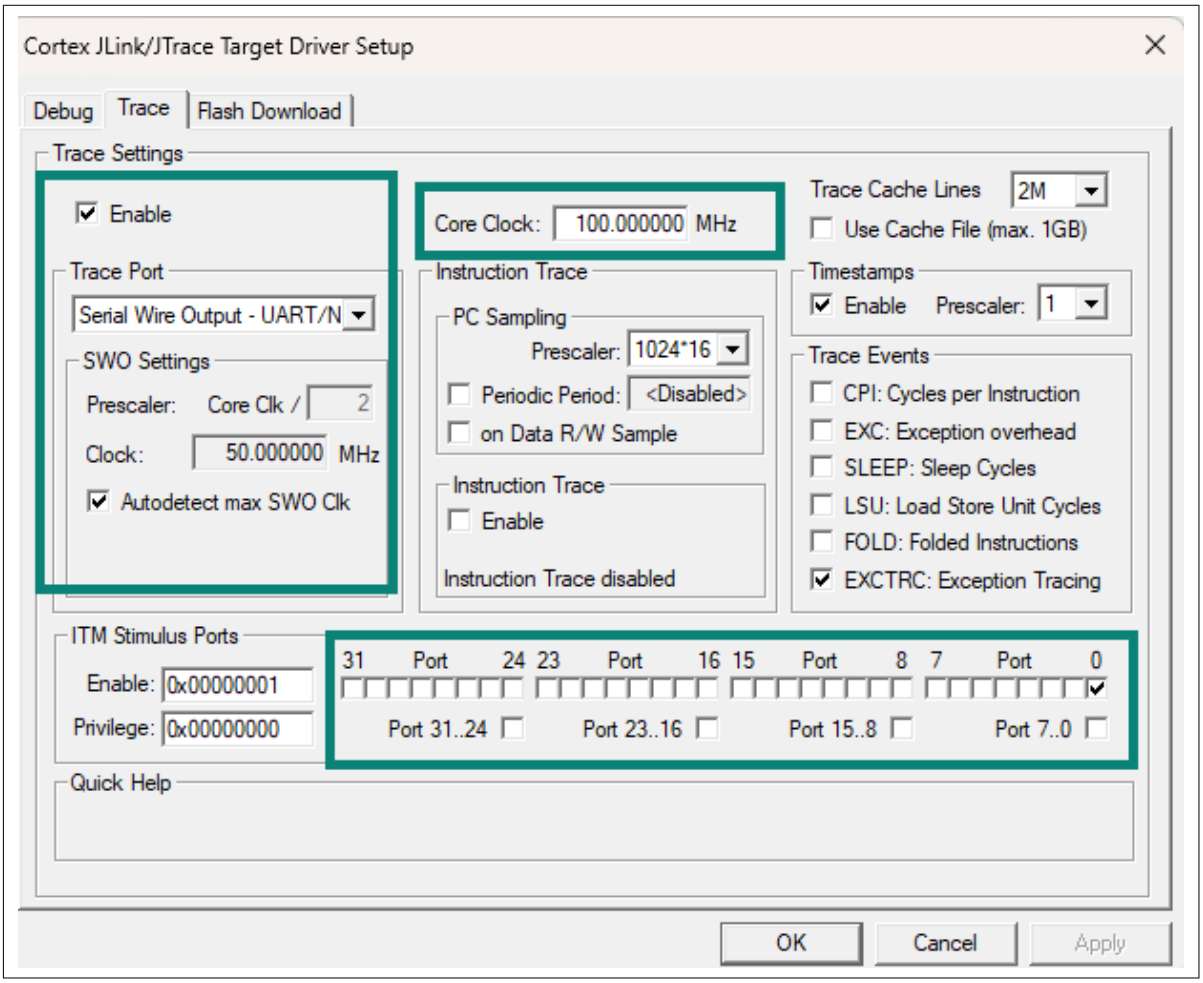


Figure 33 Configurations in Debug > Trace tab

2. Click the **Build** icon on the µVision toolbar to rebuild the code
3. Click the **Start or Stop Debug Session** icon on the µVision toolbar to program the image and start the debug perspective
4. Select **View > Serial Windows > Debug (printf) Viewer**
5. Click the **Run** icon on the toolbar to start the execution and collection of ITM data

You should see the user LED toggling every second. The Debug Viewer displays logs as shown in Figure 34.

4 Performing trace on PSOC™ 6 MCU

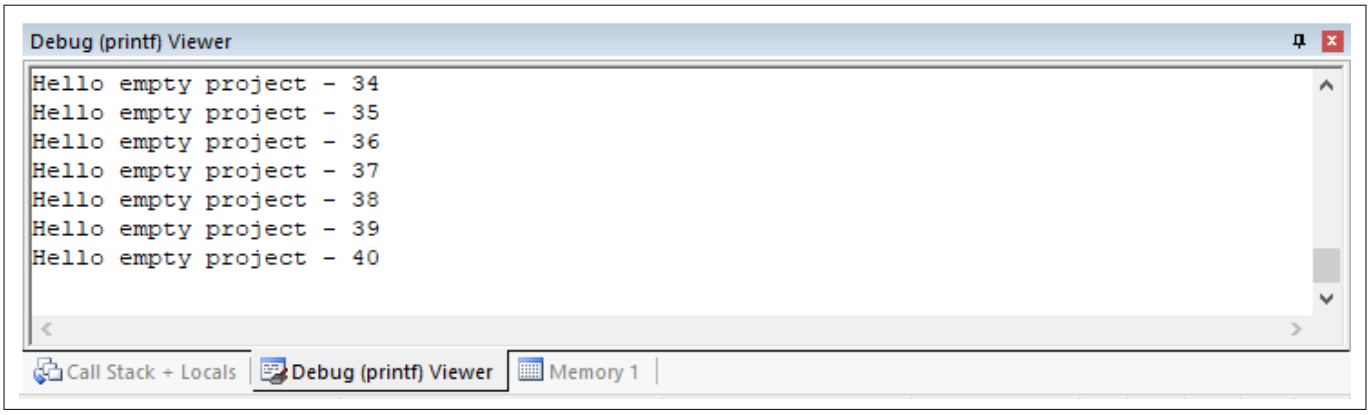


Figure 34 µVision Debug Viewer showing printf outputs

5 Summary

This application note has discussed the PSOC™ 6 MCU trace architecture and its associated development tools. It has also covered various general aspects of Arm® trace architecture. With this background, you should now be able to understand and perform trace on PSOC™ 6 MCUs to its full potential.

Although this application note has focused on demonstrating the ETM and ITM trace, third-party tools generally offer a wide range of additional trace features, including function profiling, data watchpoints, interrupt logging, and code coverage. Refer to the respective tool documentation to understand all the features they offer and learn how to use them effectively for complex embedded application debugging and development.

References

[1] AN228571: Getting started with PSOC™ 6 MCU on ModusToolbox™ software
[2] ModusToolbox™ tools package user guide (v3.0)
[3] Learn the architecture: Understanding trace by Arm® Developer
[4] IAR Embedded Workbench IDE user guide
[5] Keil µVision user guide

Revision history

Document version	Date of release	Description of changes
**	2022-09-22	Initial release.
*A	2022-12-09	Section 3.1.2 - Added a note and fixed errors.
*B	2023-08-17	Updated link references.
*C	2024-09-30	Updated sections: Creating or importing the project using ModusToolbox™ and the following subsections. Performing trace on IAR Embedded Workbench (EW) and the following subsections. Performing trace on Keil µVision and the following subsections.

Trademarks

All referenced product or service names and trademarks are the property of their respective owners.

Edition 2024-09-30

Published by

Infineon Technologies AG

81726 Munich, Germany

Do you have a question about any aspect of this document?

Email: erratum@infineon.com

Document reference
IFX-sad1663315768779

Important notice

The information contained in this application note is given as a hint for the implementation of the product only and shall in no event be regarded as a description or warranty of a certain functionality, condition or quality of the product. Before implementation of the product, the recipient of this application note must verify any function and other technical information given herein in the real application. Infineon Technologies hereby disclaims any and all warranties and liabilities of any kind (including without limitation warranties of non-infringement of intellectual property rights of any third party) with respect to any and all information given in this application note.

The data contained in this document is exclusively intended for technically trained staff. It is the responsibility of customer's technical departments to evaluate the suitability of the product for the intended application and the completeness of the product information given in this document with respect to such application.

Warnings

Due to technical requirements products may contain dangerous substances. For information on the types in question please contact your nearest Infineon Technologies office.

Except as otherwise explicitly approved by Infineon Technologies in a written document signed by authorized representatives of Infineon Technologies, Infineon Technologies' products may not be used in any applications where a failure of the product or any consequences of the use thereof can reasonably be expected to result in personal injury.