Getting started with XMC5000 MCU on ModusToolbox™ software

About this document

Scope and purpose

This application note introduces the XMC5000 MCU from the XMC™ microcontroller family, covering its architecture and development tools. It demonstrates how to create your first project using the Eclipse IDE for ModusToolbox™ software and guides you to additional online resources to accelerate your learning about the XMC5000 MCU.

***Note**: This application note covers only the XMC5000 MCU family.*

Intended audience

This document is intended for users who are new to the XMC5000 MCU family and ModusToolbox™ software.

Associated part family

All XMC5000 devices.

Software version

ModusToolbox™ software 3.8 or later.

Table of contents

About this document . . . 1
Table of contents . . . 2
1 Introduction . . . 4
1.1 Architecture and product lines . . . 4
1.2 XMC5000 MCU features . . . 5
2 XMC5000 MCU family resources . . . 7
3 XMC5000 MCU development kits . . . 8
4 Development ecosystem . . . 9
4.1 Firmware/application development . . . 9
4.1.1 Installing the ModusToolbox™ tools package . . . 9
4.1.2 Choosing an IDE . . . 9
4.1.3 ModusToolbox™ help . . . 10
5 Getting started with XMC5000 MCU design . . . 11
5.1 Prerequisites . . . 11
5.1.1 Hardware . . . 11
5.1.2 Software . . . 11
5.2 XMC5000 multi-core project structure . . . 11
5.3 Application development . . . 12
5.4 About the design . . . 13
5.5 Create a new application . . . 13
5.5.1 Eclipse IDE for ModusToolbox™ . . . 13
5.5.1.1 View and modify the design . . . 18
5.5.1.1.1 Open Device Configurator . . . 20
5.5.1.1.2 Add retarget-io middleware . . . 21
5.5.1.2 Write firmware . . . 22
5.5.1.3 Build the application . . . 29
5.5.1.4 Program the device . . . 30
5.5.1.5 Test your design . . . 32
5.5.1.6 Debugging the application using KitProg3/MiniProg4 . . . 34
5.5.1.7 Multi-core project . . . 36
5.5.2 Visual Studio Code for ModusToolbox™ . . . 45
5.5.3 IAR Embedded Workbench for ModusToolbox™ . . . 45
5.5.4 Arm® Keil® µVision® for ModusToolbox™ . . . 45
6 Summary . . . 46
References . . . 47
7 Glossary . . . 48
Revision history . . . 49

Table of contents

Trademarks 50

Disclaimer 51

1 Introduction

The XMC5000 MCU family integrates the following features on a single chip:

- Up to 160 MHz 32-bit Arm® Cortex®-M4F CPU with the following:
  - Single-cycle multiply
  - Single precision floating-point unit (FPU)
  - Memory protection unit (MPU)
- Up to 100-MHz 32-bit Arm® Cortex® M0+ CPU with the following:
  - Single-cycle multiply
  - Memory protection unit (MPU)
- Programmable analog and digital peripherals
- Up to 2112 KB of code flash with an additional, up to 128 KB of work flash, and an internal SRAM of up to 256 KB
- ModusToolbox™ development environment with installable SDKs and libraries, industry-standard Arm® tools, and RTOS support

The ModusToolbox™ software environment supports XMC5000 MCU family application development with a set of tools for configuring the device, setting up the peripherals, and complementing your projects with world-class middleware.

This application note introduces you to the capabilities of the XMC5000 MCU family, gives an overview of the development ecosystem, and gets you started with a simple 'Hello World' application wherein you learn to use the XMC5000 MCU family. The document also explains how to create the application from an empty starter application, but the completed design is available as a code example for ModusToolbox™ on GitHub.

1.1 Architecture and product lines

Figure 1 shows a detailed block diagram of the MCU.

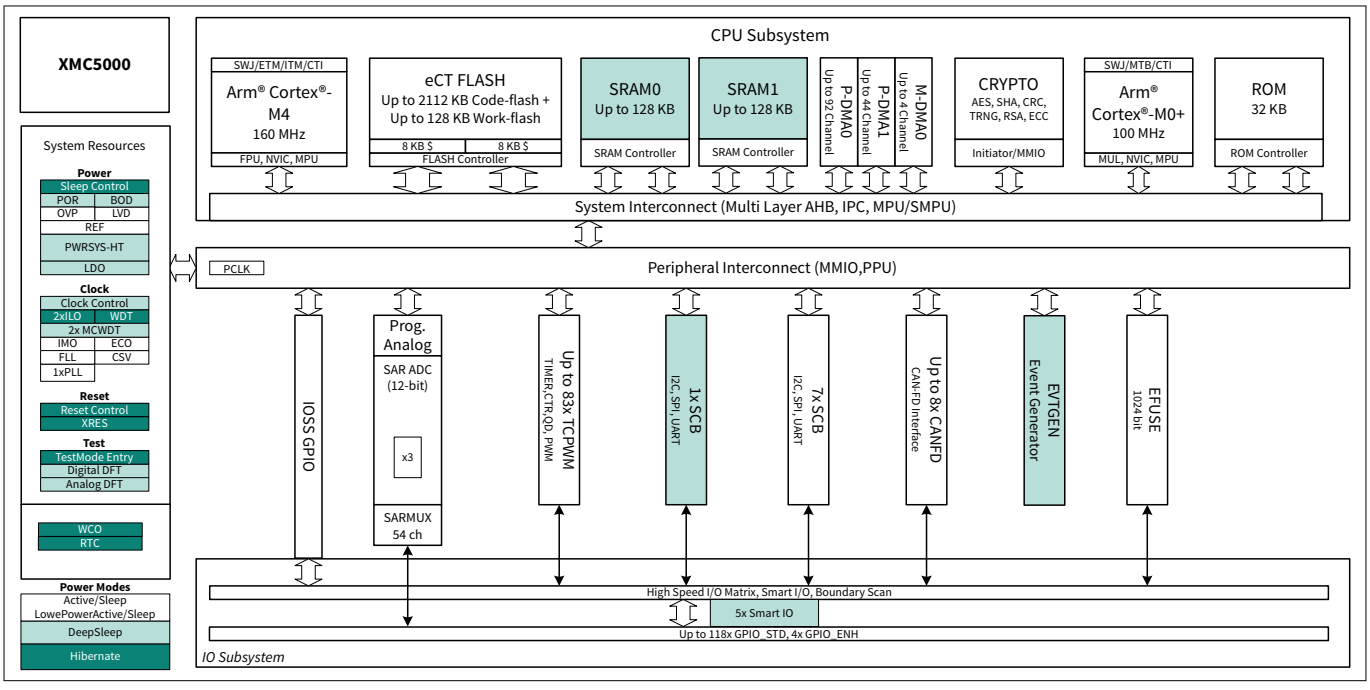


Figure 1 XMC5000 architecture diagram

Table 1 provides an overview of the product line.

Table 1 XMC5000 MCU family product line

Device series	Details
XMC5100 series	Two-core architecture: Up to 80 MHz Arm® Cortex®-M4F and up to 80 MHz Cortex®-M0+ 576 KB code flash, 64 KB work flash, 64 KB SRAM Packages: 64/100 LQFP
XMC5200 series	Two-core architecture: Up to 160 MHz Arm® Cortex®-M4F and up to 100 MHz Cortex®-M0+ 1088 KB code flash, 96 KB work flash, 128 KB SRAM Packages: 64/100/144 LQFP
XMC5300 series	Two-core architecture: Up to 160 MHz Arm® Cortex®-M4F and up to 100 MHz Cortex®-M0+ 2112 KB code flash, 128 KB work flash, 256 KB SRAM Packages: 100/144 LQFP

***Note**: For more details, see the device datasheets.*

1.2 XMC5000 MCU features

The following is a list of key features of the XMC5000 MCU. For more information, see the device datasheets and reference manuals.

- CPU subsystem
  - 32-bit dual-core CPU subsystem
    - 160 MHz (max.) 32-bit Arm® Cortex®-M4F CPU with single-cycle multiply, single precision floating-point, and memory protection units
    - 100 MHz (max.) 32-bit Arm® Cortex® M0+ CPU with single-cycle multiply and memory protection unit
  - Inter-processor communication in hardware
  - Two types of DMA controllers: one to support peripheral-to-memory (and vice versa) and one for memory-to-memory data transfers over the AHB bus
  - Up to 2112 KB of code flash along with up to 128 KB of work flash
    - Flash programming on JTAG/SWD interface
    - Read-while-Write (RWW) allows updating the code flash and work flash while executing from it
    - Single- and dual-bank modes (specifically for firmware over-the-air (FOTA) update)
  - An internal SRAM of up to 256 KB
  - Crypto engine to support Enhanced Secure Hardware Extension (eSHE) and Hardware Secure Module (HSM)

    The crypto engine and software support the following functions:
    - RSA-2048, RSA-3072, RSA-4096, ECC-256, ECC-384, SHA-2, SHA-3, AES-128/-192/-256, and 3DES
    - True random number generator (TRNG) and pseudo random number generator (PRNG)

    - Hash function
    - Galois/Counter mode (GCM)
  - Hardware error correction (SECDED ECC) on all safety-critical memories (SRAM and flash)
- Communication
  - High-speed CAN FD communication supporting up to 8 Mbps data rate
  - Serial interface to support various serial communications (UART/SPI/I2C)
- Miscellaneous
  - Low-power 2.7 V to 5.5 V operation, with two robust brownout detect (BOD) and overvoltage detect (OVD) options
  - Programmable GPIOs, and smart I/O to perform Boolean operations on signals going to and from I/O pins
  - Deep Sleep and Hibernate power modes for low-power solution
  - High-performance 12-bit, 1 Msps analog-to-digital converter (ADC)
  - Hardware watchdog function
  - Real-time clock with auto-calibration
  - Timing and pulse-width modulation with support for timer, capture, quadrature, pulse-width modulation (PWM outputs), PWM with dead-time (PWM_DT), pseudo-random PWM (PWM_PR), and shift register (SR) modes; some PWM channels also support stepper motor control
  - Event generator to support cyclic wakeup from Deep Sleep mode and peripheral trigger in active power mode
  - Debugging via JTAG controller (IEEE-1149.1-2001 compliant interface), and Arm® SWD port
  - Supports Arm® Embedded Trace Macrocell (ETM)
    - Data trace using SWD
    - Instruction and data trace using JTAG

2 XMC5000 MCU family resources

The XMC5000 family 32-bit Arm® Cortex® microcontroller webpage contains a wealth of data that will assist you in selecting the right XMC5000 device and quickly and effectively integrate it into your design. The following is an abbreviated list of resources for the XMC5000 MCU family.

- Overview**:** XMC5000 MCU family webpage
- Product selectors**:** XMC5000 series MCU
- Datasheet**:** Describes and provides electrical specifications for each device family
- Application notes and code examples cover a broad range of topics, from basic to advanced
- Reference manuals**:** Provides detailed descriptions of the architecture and registers in each device family
- XMC5000 MCU programming specification**:** Provides the information necessary to program the nonvolatile memory of XMC5000 MCU family devices
- **Development tools:** KIT_XMC52_EVK
- **Technical support:** XMC™ MCU community forum, XMC™ knowledge base articles

3 XMC5000 MCU development kits

- Development kits
  - XMC5000 Evaluation Kit (KIT_XMC52_EVK)

For the complete list of kits for the XMC5000 MCU family along with the shield modules, see the microcontroller kits page.

4 Development ecosystem

4.1 Firmware/application development

Infineon provides the ModusToolbox™ software for firmware/application development on the XMC5000 MCU.

ModusToolbox™ software is a modern, extensible development ecosystem supporting a wide range of Infineon microcontroller devices, including PSOC™ Arm® Cortex® Microcontrollers, TRAVEO™ T2G Arm® Cortex® microcontroller, XMC™ industrial microcontrollers, AIROC™ Wi-Fi devices, AIROC™ Bluetooth® devices, and USB-C Power Delivery microcontrollers. This software includes configuration tools, low-level drivers, middleware libraries, and other packages that enable you to create MCU and wireless applications. All tools run on Windows, macOS, and Linux. ModusToolbox™ includes an Eclipse IDE, which provides an integrated flow with all the ModusToolbox™ tools. Other IDEs such as Visual Studio Code, IAR Embedded Workbench, and Arm® MDK (µVision) are also supported.

ModusToolbox™ software supports stand-alone device and middleware configurators. Use the configurators to set the configuration of different blocks in the device and generate code that can be used in firmware development.

Libraries and enablement software are available at the GitHub site.

ModusToolbox™ tools and resources can also be used on the command-line. See the "Build system" section in the ModusToolbox™ tools package user guide for detailed documentation.

4.1.1 Installing the ModusToolbox™ tools package

See the ModusToolbox™ tools package installation guide for details.

4.1.2 Choosing an IDE

ModusToolbox™ software, the latest-generation toolset, is supported across Windows, Linux, and macOS platforms. ModusToolbox™ software supports third-party IDEs, including the Eclipse IDE, Visual Studio Code, Arm® MDK (µVision), and IAR Embedded Workbench. The tools package includes an implementation for all the supported IDEs. The tools support all XMC5000 MCUs. The associated BSP and library configurators also work on all three host operating systems.

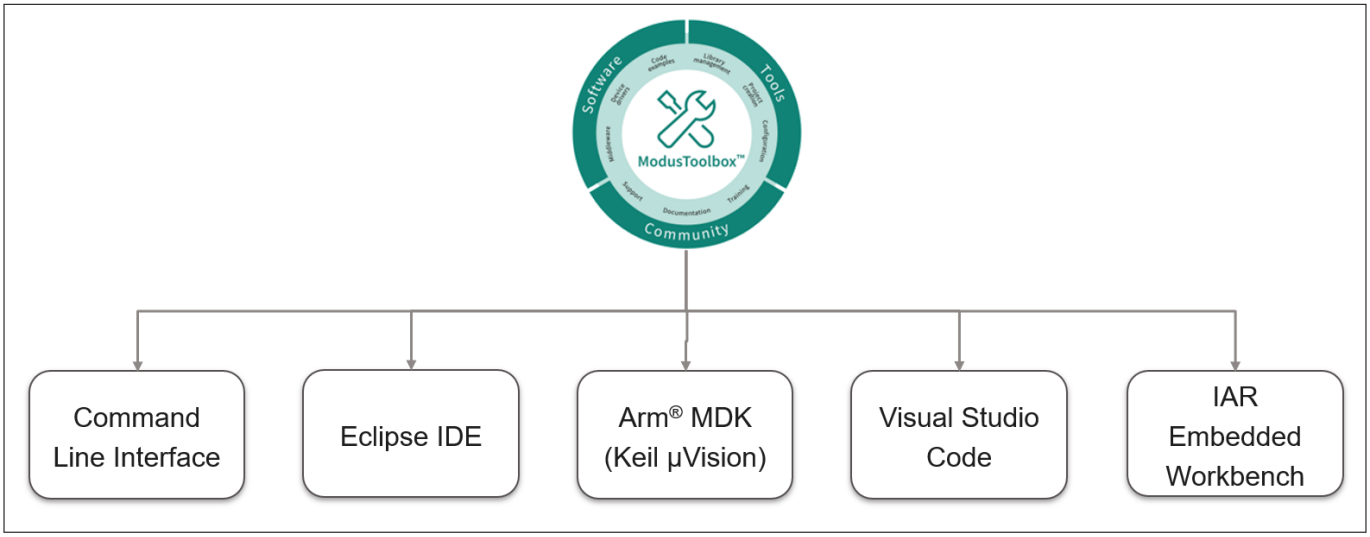


Figure 2 ModusToolbox™ environment

4.1.3 ModusToolbox™ help

The ModusToolbox™ ecosystem provides documentation and training. Launch the Eclipse IDE for ModusToolbox™ software and navigate to the following Help menu items:

Choose **Help** > **ModusToolbox™ General Documentation:**

- **ModusToolbox™ Documentation Index:** Provides brief descriptions and links to various types of documentation included as part of the ModusToolbox™ software
- **ModusToolbox™ Installation Guide:** Provides instructions for installing the ModusToolbox™ software
- **ModusToolbox™ User Guide:** This guide primarily covers the ModusToolbox™ aspects of building, programming, and debugging applications. Additionally, it covers various aspects of the tools installed along with the IDE
- **ModusToolbox™ Training Material:** Links to the training material are available at **https://github.com/Infineon/training-modustoolbox**
- **Release Notes:** Describes the features and known limitations for the ModusToolbox™ software, provided as part of the ModusToolbox™ tools package included with the installer

For documentation on Eclipse IDE for ModusToolbox™, choose **Help** > **Eclipse IDE for ModusToolbox™ documentation**.

- **User Guide:** Provides descriptions about creating applications as well as building, programming, and debugging them using the Eclipse IDE
- **Eclipse IDE Survival Guide:** This is a link to a forum with answers for questions about how to get common tasks done

5 Getting started with XMC5000 MCU design

This section provides the following:

- Demonstrates how to build a simple design based on the XMC5000 MCU family and program it on to the development kit
- Makes it easy to learn XMC5000 MCU design techniques and how to use the ModusToolbox™ software with different IDEs

5.1 Prerequisites

Before you get started with the application development instructions, ensure that you have the appropriate development kit for your XMC5000 MCU product line and have installed the required software. Ensure that you have an active Internet connection to access the GitHub repositories during project creation.

5.1.1 Hardware

The design is developed for XMC5000 Family Evaluation Kit (KIT_XMC52_EVK). However, you can build the application for other development kits also. For more details, see the Application development section.

5.1.2 Software

Install ModusToolbox™ software 3.8 or later.

After installing the software, see the ModusToolbox™ tools package user guide to get an overview of the software.

5.2 XMC5000 multi-core project structure

Figure 3 shows the new folder structure for an example multi-core application.

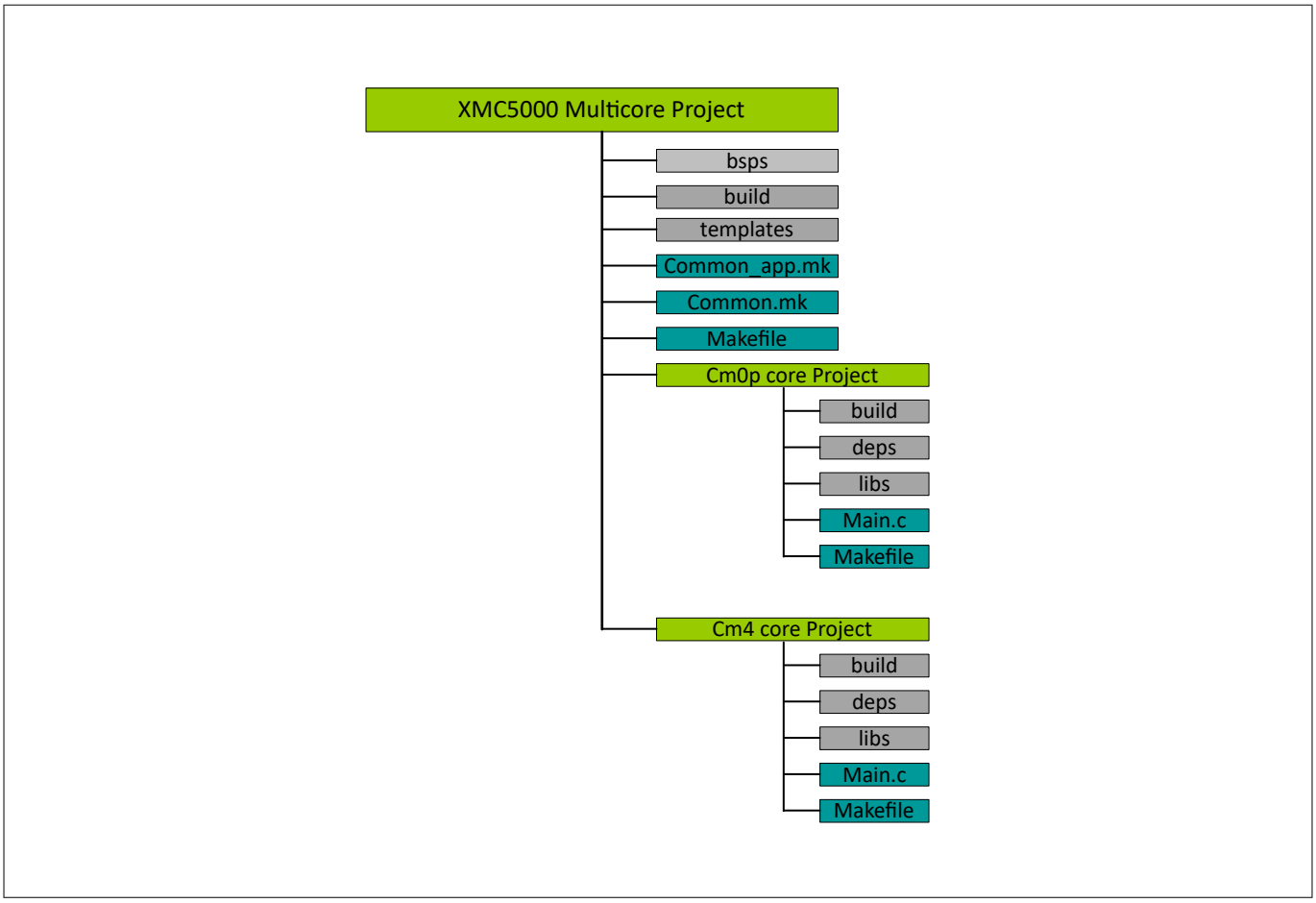


Figure 3 XMC5000 multi-core project structure

As shown in the Figure 3, XMC5000 multi-core project contains `cm0p` and `cm4` projects.

For multi-core applications, there are multiple projects, but only one project per core. The applications have app-owned BSPs that will be common to all projects inside a multi-core application.

The project execution starts with executing the code in CM0+ core. In a single core project, a pre-compiled program of CM0+ is used, which launches CM4 core and enters into to Sleep. In a multi-core project, the application code of all two cores is visible and you can see how code is running in two cores.

5.3 Application development

These instructions are grouped into several sections. Each section is dedicated to a phase of the application development workflow. The key sections are:

1. Create a new application
2. View and modify the design
3. Write firmware
4. Build the application
5. Program the device
6. Test your design

This design is developed for the XMC5000 Evaluation Kit. You can use other supported kits to test this example by selecting the appropriate kit while creating the application.

5.4 About the design

This design uses the XMC5000 MCU family to execute two tasks: UART communication and LED control.

After device reset, this code example uses the UART to print a “Hello World” message to the serial port stream, and starts blinking the user LED on the kit. When you press the **Enter** key on the serial console, the blinking is paused or resumed.

5.5 Create a new application

This section guides you in creating a new application. It uses the **Empty App** starter application and manually adds the functionality from the **Hello World** starter application.

As mentioned in the Choosing an IDE section, ModusToolbox™ software supports the following third-party IDEs:

1. Eclipse IDE for ModusToolbox™
2. Visual Studio Code for ModusToolbox™
3. IAR Embedded Workbench for ModusToolbox™
4. Arm® Keil® µVision® for ModusToolbox™

The following sections provide details on how to create a new application on different IDEs.

5.5.1 Eclipse IDE for ModusToolbox™

If you are familiar with developing applications with ModusToolbox™ software, you can use the **Hello World** starter application directly. It is a complete design, with all the firmware written for the supported kits. You can walk through the instructions and observe how the steps are implemented in the code example.

If you start from scratch and follow all the instructions in this application note, you can use the **Hello World** code example as a reference while following the instructions.

Launch the Dashboard 3.8 application to get started.

***Note**: Ensure that the Dashboard 3.8 application is connected to the Internet to clone the starter application onto your machine.*

The Dashboard 3.8 application helps you get started using the various tools with easy access to documentation and training material, and provides a simple path for creating applications and creating and editing BSPs.

1. Open the Dashboard 3.8 application. Do one of the following:
   - **Windows:** Navigate to `[ModusToolbox™ installation path]/tools_3.8/dashboard/dashboard.exe` or you can also select the "ModusToolbox™ Dashboard 3.8" item from the Windows Start menu
   - **Linux:** Navigate to `[ModusToolbox™ installation path]/tools_3.8/dashboard` and run the executable
   - **macOS:** Run the "dashboard" app
2. On the Dashboard 3.8 window, in the right pane, in the **Target IDE** drop down list, select **Eclipse IDE for ModusToolbox™**, and click **Launch Eclipse IDE for ModusToolbox™**

5 Getting started with XMC5000 MCU design

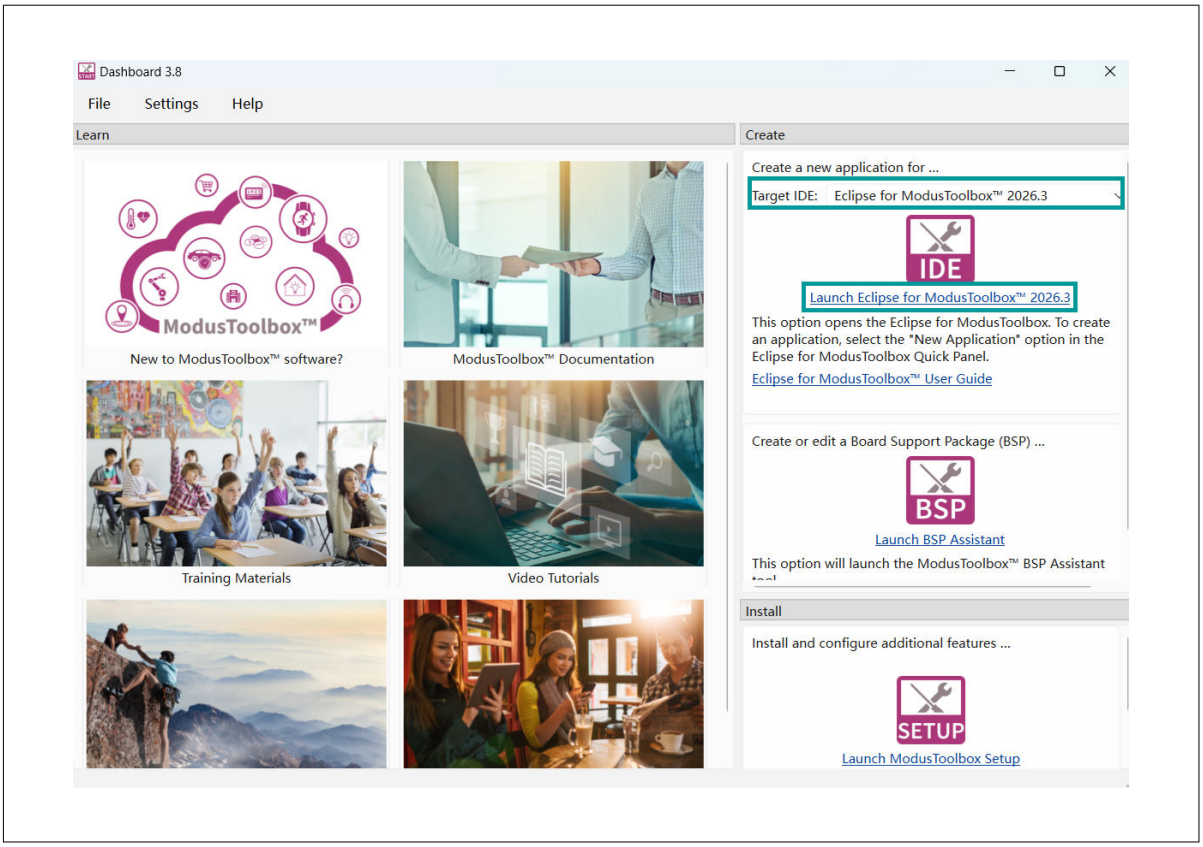


Figure 4 Dashboard 3.8 application

3. Select a new workspace

At launch, Eclipse IDE for ModusToolbox™ displays a dialog to choose a directory for use as the workspace directory. The workspace directory is used to store workspace preferences and development artifacts. You can choose an existing empty directory by clicking the **Browse** button, as shown in the following figure. Alternatively, you can type in a directory name to be used as the workspace directory along with the complete path, and the IDE will create the directory.

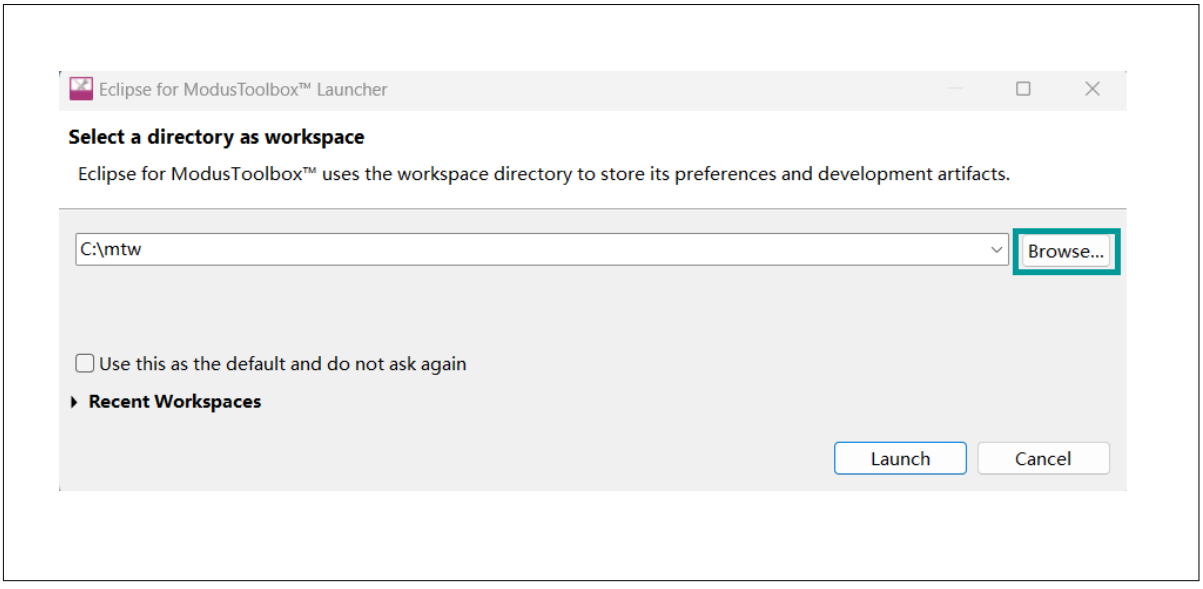


Figure 5 Select a directory as the workspace

4. Create a new ModusToolbox™ application

a. Click **New Application** in the Start group of the Quick Panel

b. Alternatively, you can choose **File** > **New** > **ModusToolbox™ Application**, as Figure 6 shows. This displays the Eclipse IDE for ModusToolbox™ Application window

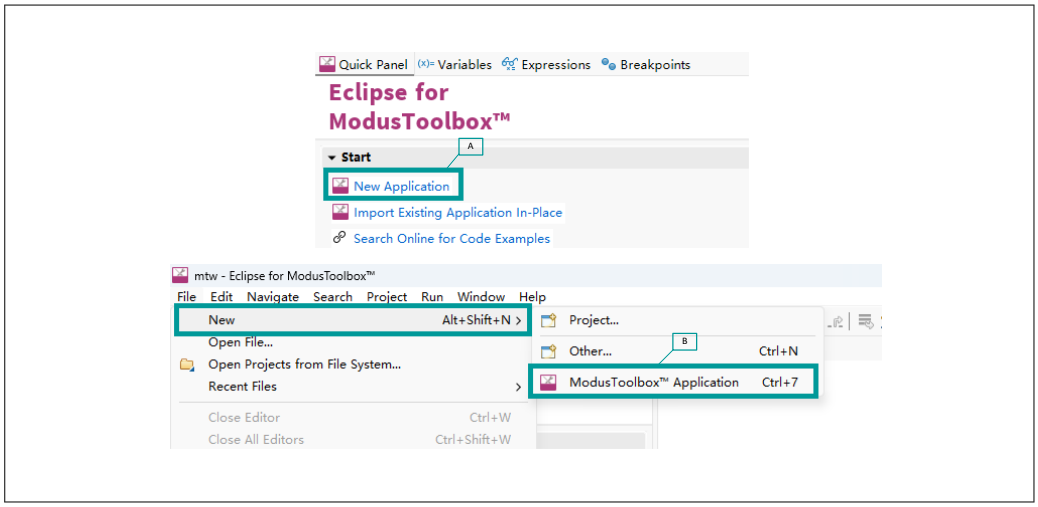


Figure 6 Create a New ModusToolbox™ Application

5. Select a target XMC5000 Evaluation Kit

ModusToolbox™ speeds up the development process by providing BSPs that set various workspace/ project options for the specified development kit in the new application dialog.

a. In the **Choose Board Support Package (BSP)** dialog, choose the **Kit Name** that you have. The steps that are followed use **KIT_XMC52_EVK**, as shown in Figure 7

b. Click **Next**

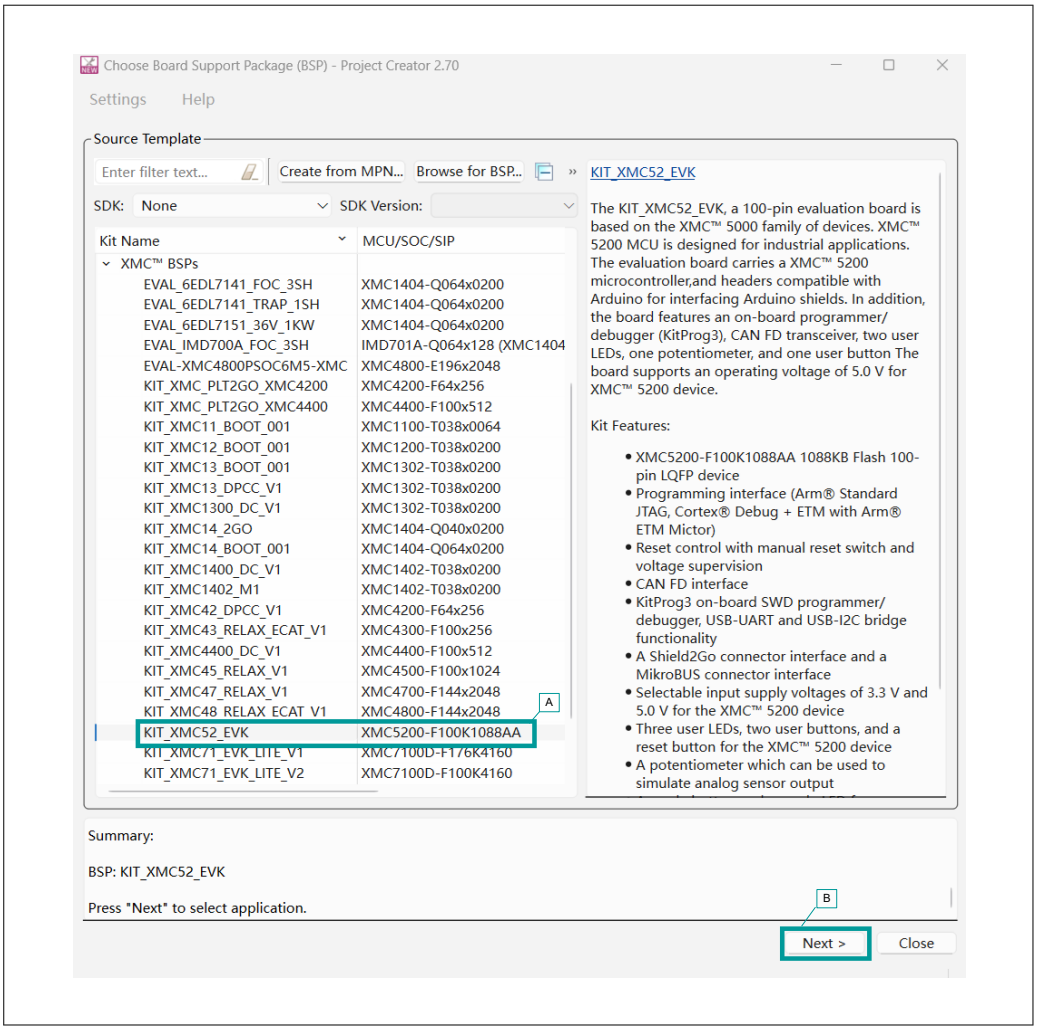


Figure 7 Choose target hardware

c. In the **Select Application** dialog, select the **Empty App** starter application, as shown in the following figure

d. In the **Name** field, type in a name for the application, such as **Hello_World**. You can choose to leave the default name if you prefer

***Note**: Try to use a short name without spaces in between.*

e. Click **Create** to create the application, as shown in the following figure and wait for the Project Creator to automatically close once the project is successfully created

5 Getting started with XMC5000 MCU design

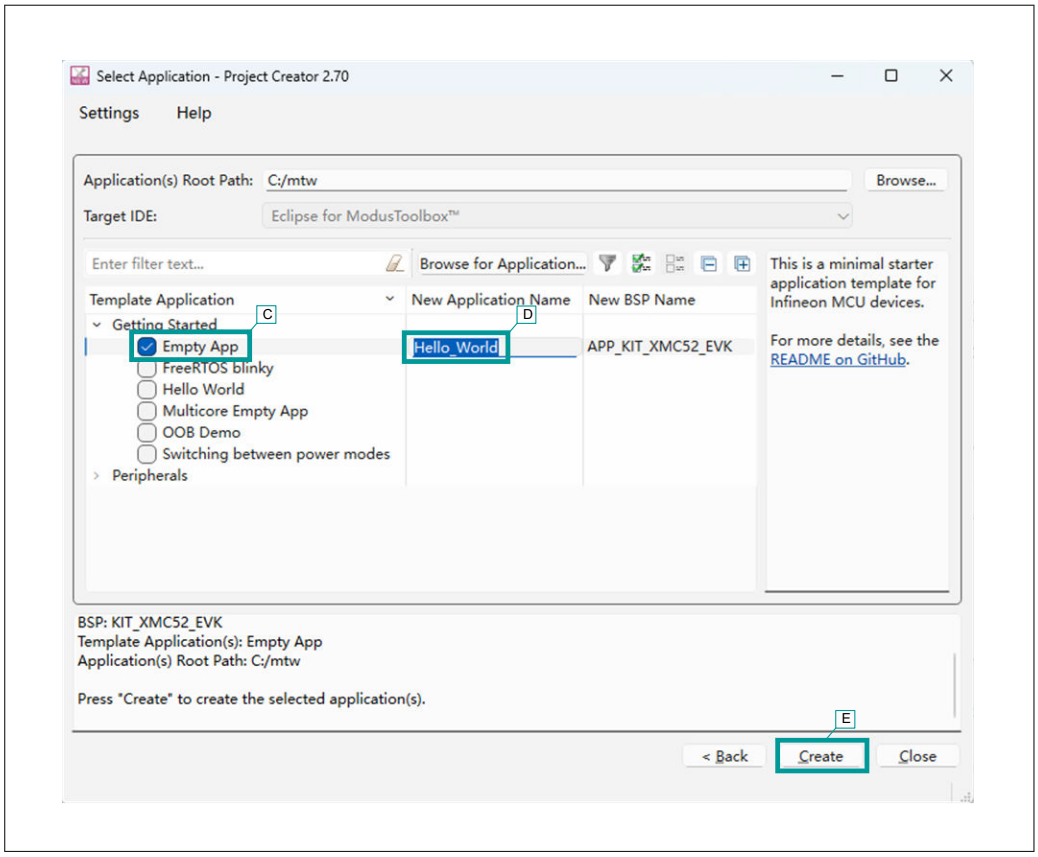


Figure 8 Choose starter application

You have successfully created a new ModusToolbox™ application for XMC5000 MCU.

If you are using custom hardware based on the XMC5000 MCU or a different XMC5000 MCU part number, see the "Creating your Own BSP" section in the ModusToolbox™ user guide.

***Note**: To create a multi-core project, select **Multicore Empty App** from **Select Application** dialog as shown in Figure 9. See the ModusToolbox™ applications for more details.*

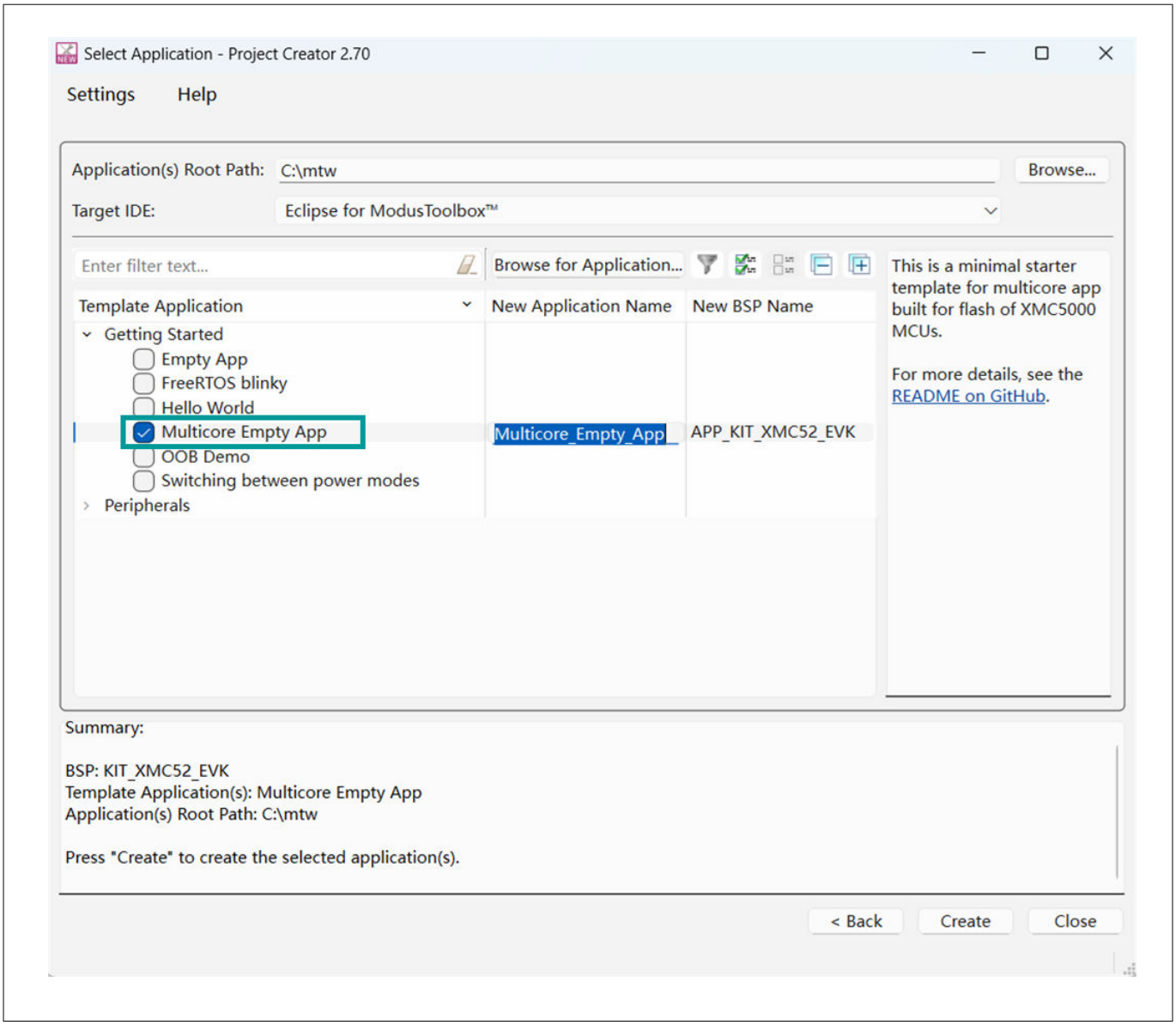


Figure 9 Choose multi-core starter application

5.5.1.1 View and modify the design

Figure 10 shows the ModusToolbox™ Project Explorer interface displaying the structure of the application project.

The XMC5000 MCU family consists of two cores: one CM0+ core and one M4F core. This application note shows an example code for firmware development using ModusToolbox™ software.

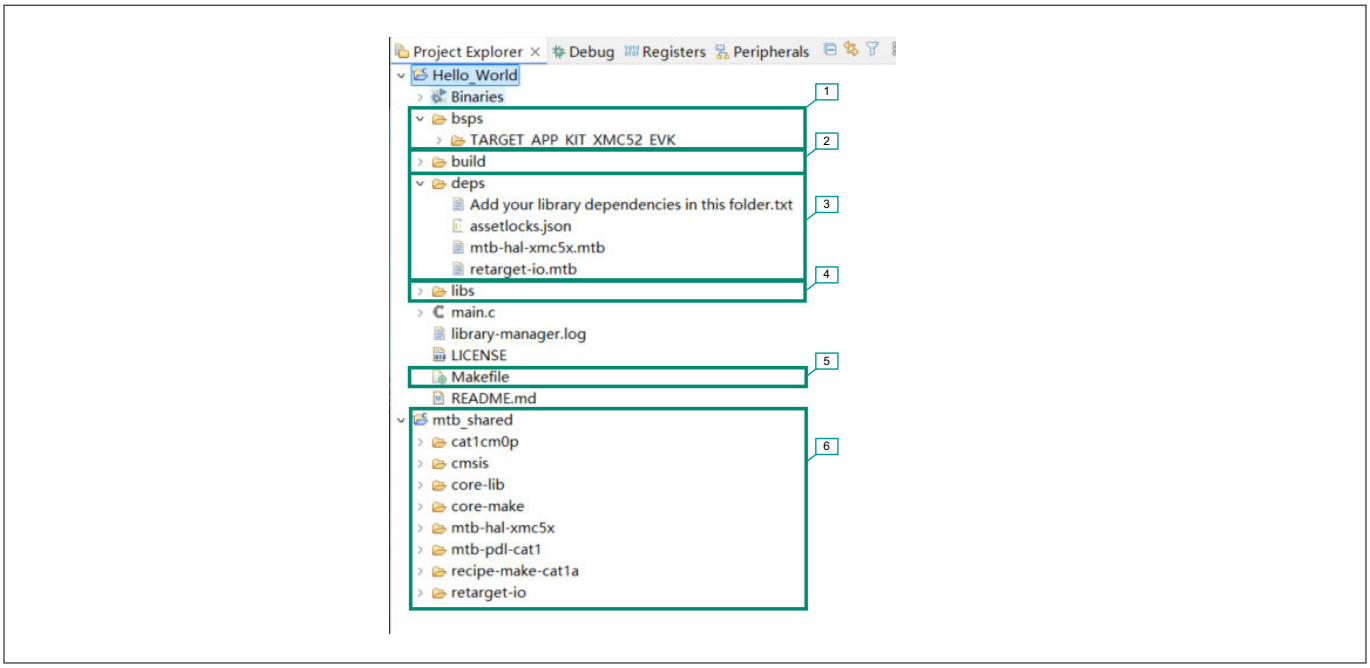


Figure 10 Project Explorer view

A project folder consists of various subfolders, each denoting a specific aspect of the project.

1. The files provided by the BSP are available in the *bsps* folder and are listed under *TARGET_<bsp name>* subfolders. All the input files for the device and peripheral configurators are in the *config* folder inside the BSP. The *GeneratedSource* folder in the BSP contains the files that are generated by the configurators and are prefixed with *cycfg_*. These files contain the design configuration as defined by the BSP. From ModusToolbox™ 3.x or later, you can directly customize the configurator files of the BSP for your application rather than overriding the default design configurator files with custom design configurator files because BSPs are completely owned by the application. The *bsps* folder also contains the linker scripts and the start-up code for the XMC5000 MCU used on the board
2. The *build* folder contains all the artifacts resulting from a build of the project. The output files are organized by target BSPs
3. The *deps* folder contains *.mtb* files, which provide the locations from which ModusToolbox™ pulls the libraries that are directly referenced by the application. These files typically contain the GitHub location of a library. The *.mtb* files also contain a git commit hash or tag that tells which version of the library is to be fetched and a path as to where the library should be stored locally. For example, here, `retarget-io.mtb` points to `mtb://retarget-io#latest-v1.X#$$ASSET_REPO$$/retarget-io/latest-v1.X`. The variable `$$ASSET_REPO$$` points to the root of the shared location which defaults to `mtb_shared`. If the library must be local to the application instead of shared, use `$$LOCAL$$` instead of `$$ASSET_REPO$$`
4. The *libs* folder also contains *.mtb* files. In this case, they point to libraries that are included indirectly as a dependency of a BSP or another library. For each indirect dependency, the Library Manager places a *.mtb* file in this folder. These files have been populated based on the targets available in the *deps* folder

   For example, using the **KIT_XMC52_EVK** BSP populates the *libs* folder with the following *.mtb* files: cat1cm0p, cmsis.mtb, core-lib.mtb, core-make.mtb, deivce-db.mtb, mtb-hal-xmc5x.mtb, mtb-pdl-cat1.mtb, receipe-make-cat1a, retarget-io.mtb

   The *libs* folder contains the *mtb.mk* file, which stores the relative paths of all the libraries required by the application. The build system uses this file to find all the libraries required by the application. Everything in the *libs* folder is generated by the Library Manager so you should not manually edit anything in that folder
5. An application contains a Makefile which is at the application's root folder. This file contains the set of directives that the make tool uses to compile and link the application project. There can be more than

one project in an application. In that case, there is a Makefile at the application level and one inside each project

6. By default, when creating a new application or adding a library to an existing application and specifying it as shared, all libraries are placed in an *mtb_shared* directory adjacent to the application directories

   The *mtb_shared* folder is shared between different applications within a workspace. Different applications may use different versions of shared libraries if necessary

5.5.1.1.1 Open Device Configurator

BSP configurator files are in the `bsps/TARGET_<BSP-name>/config` folder. Click **<Application-name>** from **Project Explorer** and then click **Device Configurator** link in the **Quick Panel** to open the `design.modus` file in the **Device Configurator** as shown in the following figure. You can also open other configuration files in their respective configurators or click the corresponding links in the **Quick Panel**.

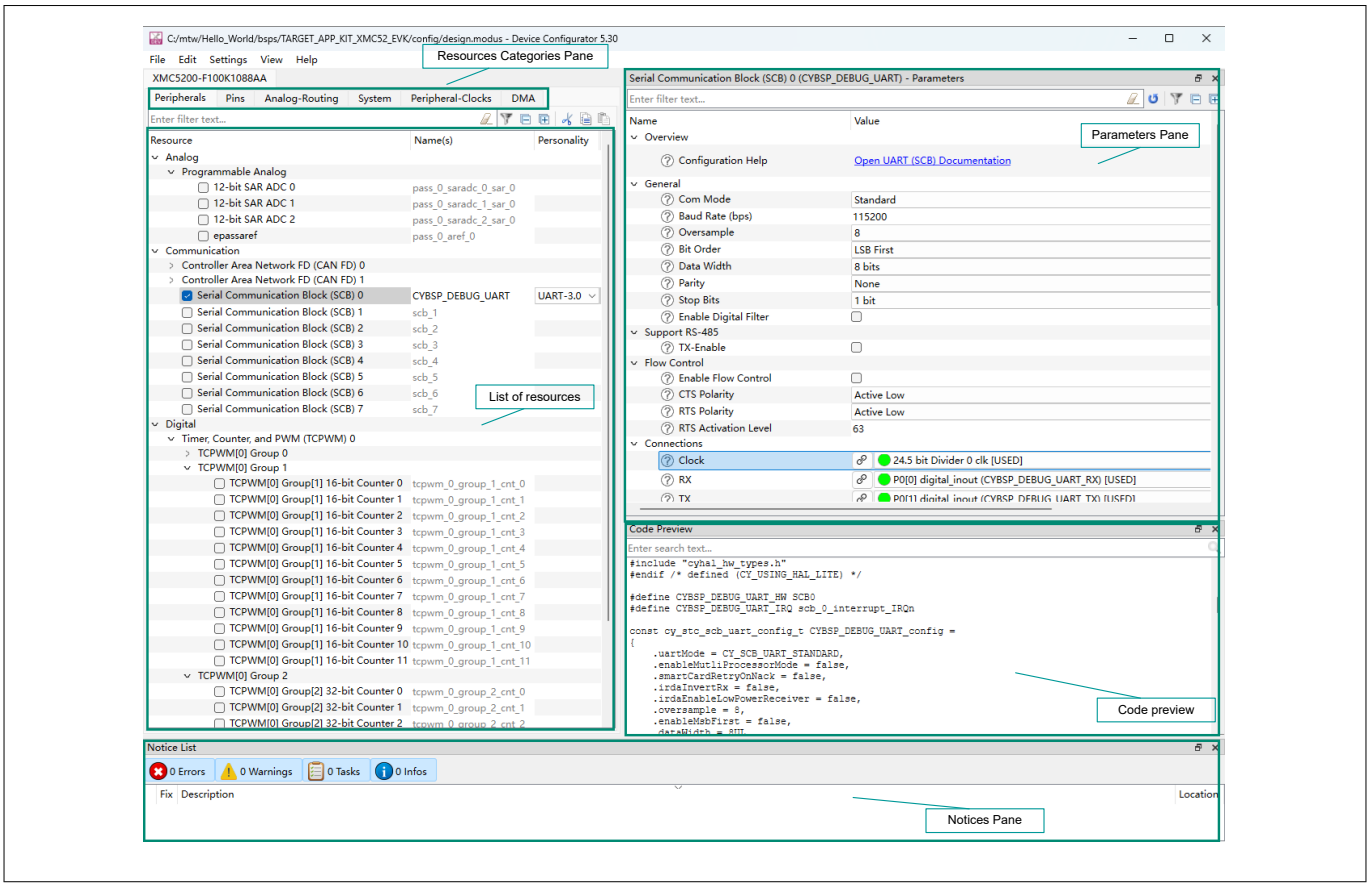


Figure 11 Device Configurator - UART configuration

The **Device Configurator** provides a set of **Resources Categories** tabs. Here, you can choose between different resources available in the device such as peripherals, pins, and clocks from the **List of Resources**.

You can choose how a resource behaves by choosing a **Personality** for the resource. For example, a **Serial Communication Block (SCB)** resource can have **EZI2C**, **I2C**, **SPI**, or **UART** personalities. The **Name(s)** is your name for the resource, which is used in firmware development. One or more aliases can be specified by using a comma to separate them (with no spaces).

The **Parameters** pane is where you enter the configuration parameters for each enabled resource and the selected personality. The **Code Preview** pane shows the configuration code generated per the configuration parameters selected. This code is populated in the `cycfg_` files in the `GeneratedSource` folder. The **Parameters** pane and **Code Preview** pane may be displayed as tabs instead of separate windows but the contents will be the same.

Any errors, warnings, and information messages arising out of the configuration are displayed in the **Notices** pane. Configuring the peripheral is required for both PDL- and HAL-based implementations to work.

Figure 11 also shows that SCB 0 is enabled and configured. As SCB 0 is used for communicating with the user through the debug UART terminal, replicate the same configuration in your setup for the "Hello World" application.

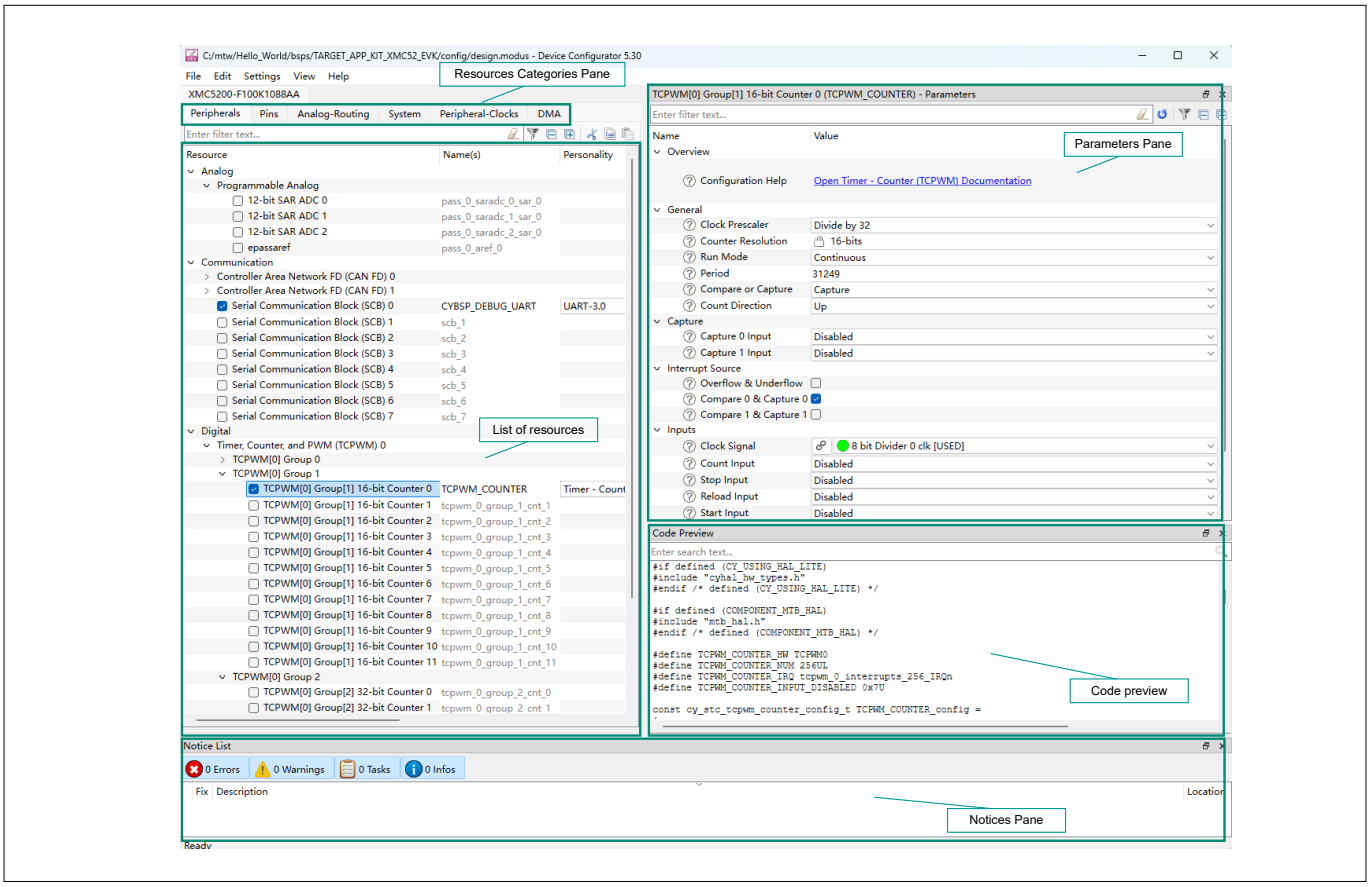


Figure 12 Device Configurator - Timer configuration

Figure 12 shows that the timer TCPWM[0] Group[1] 16-bit Counter 0 is configured in Timer - Counter mode.

This configuration is used in the code to generate interrupts for LED toggling. Make the same configuration in your setup. To make the LED toggle every second, the input clock frequency is set to 1 MHz. Ensure that this configuration is also made in your setup.

In the "Hello World" application, you are using a GPIO connected to the LED on the EVK. To use the GPIO, enable pin P19.0 in the **Pins** tab in the Resource Categories pane. Also, ensure that the Drive Mode is set to **'Strong Drive. Input buffer off'**.

At this point in the development process, the required middleware is ready to be added to the design. The middleware required for the "Hello World" application is the retarget-io library and mtb-hal-xmc5x library.

5.5.1.1.2 Add retarget-io middleware

This section explains how to add the retarget-io middleware to redirect standard input and output streams to the UART configured by the BSP. The initialization of the middleware will be done in the `main.c` file.

1. In the **Quick Panel**, click the **Library Manager** link
2. In the subsequent dialog, click **Add Libraries**
3. Under **Peripherals**, select and enable retarget-io and **mtb-hal-xmc5x**
4. Click **OK** and then **Update**

The files necessary to use the retarget-io and **mtb-hal-xmc5x** middleware are added in the `mtb_shared` > `retarget_io` folder, and the `.mtb` file is added to the `deps` folder, as shown in the following figure.

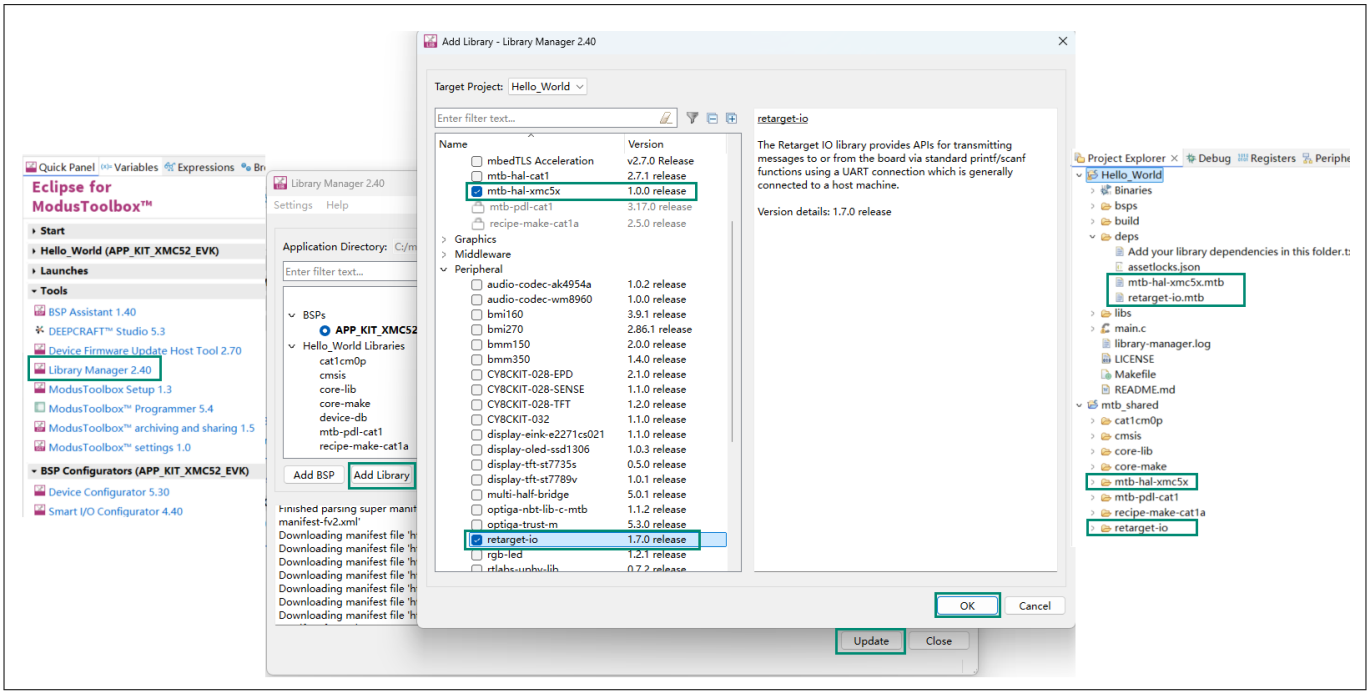


Figure 13 Add the retarget-io and mtb-hal-xmc5x middleware

5.5.1.2 Write firmware

At this point in the development process, you have created an application with the assistance of an application template and modified it to add the retarget-io middleware. In this part, you write the firmware that implements the design functionality.

If you are working from scratch using the empty XMC5000 MCU family starter application, you can copy the respective source code to the `main.c` file of the application project from the code snippet provided in this section. If you are using the "Hello World" code example, all the required files are already in the application.

Firmware flow

Examine the code in the `main.c` file of the application. Figure 14 shows the firmware flowchart.

After reset, resource initialization for this example is performed by the CM4 CPU. It configures the system clocks, pins, clock to peripheral connections, and other platform resources.

Then, the clocks and system resources are initialized by the BSP initialization function. The retarget-io middleware is configured to use the debug UART, and the user LED is initialized. The debug UART prints a “Hello World!” message on the terminal emulator, the onboard KitProg3 acts as the USB-to-UART bridge to create the virtual COM port. A timer object is configured to generate an interrupt every 1000 milliseconds. At each timer interrupt, the CPU toggles the LED state on the kit.

The firmware is designed to accept the 'Enter' key as an input, and on every press of the 'Enter' key, the firmware starts or stops the blinking of the LED.

Note that this application code uses BSP/HAL/middleware functions to execute the intended functionality.

- `cybsp_init()`: This BSP function initializes all the system resources of the device, including but not limited to the system clocks and power regulators
- `Cy_SCB_UART_Init()`: This function initializes the debug UART
- `mtb_hal_uart_setup()` and `cy_retarget_io_init()`: These functions set up the HAL UART and redirect the input/output stream to the debug UART

- `mtb_hal_uart_get()`: The while loop calls this function to detect the pressing of the 'Enter Key', which start or stop the LED toggling
- `timer_init()`: This function wraps a set of timer function calls to instantiate and configure a hardware timer. It also sets up a callback for the timer interrupt
- `isr_timer()`: This is the timer ISR getting executed in every 1000 milliseconds. This function sets a flag for toggling the LED

The flag set by the timer ISR is checked in the main loop, and the LED is toggled based on it.

5 Getting started with XMC5000 MCU design

Copy the following code snippet to the `main.c` file of your application project.

```
/*******************************************************************************
* Header Files
*******************************************************************************/
#include "cy_pdl.h"
#include "cybsp.h"
#include "cy_retarget_io.h"
#include "mtb_hal.h"

/*******************************************************************************
* Macros
*******************************************************************************/

/*******************************************************************************
* Global Variables
*******************************************************************************/
/* Interrupt configuration */
const cy_stc_sysint_t IRQ_CFG =
{
    .intrSrc = ((NvicMux3_IRQn << CY_SYSINT_INTRSRC_MUXIRQ_SHIFT) | TCPWM_TIMER_IRQ),
    .intrPriority = 7UL
};

bool timer_interrupt_flag = false;
bool led_blink_active_flag = true;

/* Variable for storing character read from terminal */
uint8_t uart_read_value = 0UL;

/* For the Retarget -IO (Debug UART) usage */
static cy_stc_scb_uart_context_t    UART_context;           /** UART context */
static mtb_hal_uart_t               UART_hal_obj;           /** Debug UART HAL object  */

/*******************************************************************************
* Function Prototypes
*******************************************************************************/
void timer_init(void);
void isr_timer(void);

/*******************************************************************************
* Function Name: main
********************************************************************************
* Summary:
* This is the main function. It sets up a timer to trigger a periodic interrupt.
* The main while loop checks for the status of a flag set by the interrupt and
* toggles an LED at 1Hz to create an LED blinky. Will be achieving the 1Hz Blink
* rate based on the The LED_BLINK_TIMER_CLOCK_HZ and LED_BLINK_TIMER_PERIOD
* Macros,i.e. (LED_BLINK_TIMER_PERIOD + 1) / LED_BLINK_TIMER_CLOCK_HZ = X ,Here,
* X denotes the desired blink rate. The while loop also checks whether the
* 'Enter' key was pressed and stops/restarts LED blinking.
*
* Parameters:
```

```
*  none
*
* Return:
*  int
*
*******************************************************************************/
int main(void)
{
    cy_rslt_t result;

    /* Initialize the device and board peripherals */
    result = cybsp_init();

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

    /* Enable global interrupts */
    __enable_irq();

    /* Debug UART init */
    result = (cy_rslt_t)Cy_SCB_UART_Init(UART_HW, &UART_config, &UART_context);

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

    Cy_SCB_UART_Enable(UART_HW);

    /* Setup the HAL UART */
    result = mtb_hal_uart_setup(&UART_hal_obj, &UART_hal_config, &UART_context, NULL);

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

    result = cy_retarget_io_init(&UART_hal_obj);

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

    /* \x1b[2J\x1b[;H - ANSI ESC sequence for clear screen */
    printf("\x1b[2J\x1b[;H");
```

```
        printf("****************** "
               "PDL: Hello World! Example "
               "****************** \r\n\n");

        printf("Hello World!!!\r\n\n");
        printf("For more projects, "
               "visit our code examples repositories:\r\n\n");

        printf("https://github.com/Infineon/"
               "Code-Examples-for-ModusToolbox-Software\r\n\n");

        /* Initialize timer to toggle the LED */
        timer_init();

        printf("Press 'Enter' key to pause or "
               "resume blinking the user LED \r\n\r\n");

        for (;;)
        {
            /* Check if 'Enter' key was pressed */
            uart_read_value = Cy_SCB_UART_Get(UART_HW);
            if (uart_read_value == '\r')
            {
                /* Pause LED blinking by stopping the timer */
                if (led_blink_active_flag)
                {
                    Cy_TCPWM_Counter_Disable(TCPWM_TIMER_HW, TCPWM_TIMER_NUM);

                    printf("LED blinking paused \r\n");
                }
                else /* Resume LED blinking by starting the timer */
                {
                    Cy_TCPWM_Counter_Enable(TCPWM_TIMER_HW, TCPWM_TIMER_NUM);
                    Cy_TCPWM_TriggerStart_Single(TCPWM_TIMER_HW, TCPWM_TIMER_NUM);

                    printf("LED blinking resumed\r\n");
                }

                /* Move cursor to previous line */
                printf("\x1b[1F");

                led_blink_active_flag ^= 1;
            }

            /* Check if timer elapsed (interrupt fired) and toggle the LED */
            if (timer_interrupt_flag)
            {
                /* Clear the flag */
                timer_interrupt_flag = false;

                /* Invert the USER LED state */
                Cy_GPIO_Inv(CYBSP_USER_LED_PORT, CYBSP_USER_LED_PIN);
            }
```

```
    }
}


/*******************************************************************************
* Function Name: timer_init
********************************************************************************
* Summary:
* This function creates and configures a Timer object. The timer ticks
* continuously and produces a periodic interrupt on every terminal count
* event. The period is defined by the 'period' and 'compare_value' of the
* timer configuration structure 'led_blink_timer_cfg'. Without any changes,
* this application is designed to produce an interrupt every 1 second.
*
* Parameters:
*  none
*
* Return :
*  void
*
*******************************************************************************/
 void timer_init(void)
 {
    cy_rslt_t result;

    /* Initialize the timer object. Does not use input pin ('pin' is NC) and
     * does not use a pre-configured clock source ('clk' is NULL). */
    result = Cy_TCPWM_Counter_Init(TCPWM_TIMER_HW, TCPWM_TIMER_NUM, &TCPWM_TIMER_config);

    if(result != CY_TCPWM_SUCCESS)
    {
        CY_ASSERT(0);
    }

    /* Interrupt settings */
    Cy_SysInt_Init(&IRQ_CFG, &isr_timer);
    NVIC_ClearPendingIRQ(NvicMux3_IRQn);
    NVIC_EnableIRQ((IRQn_Type) NvicMux3_IRQn);

    Cy_TCPWM_Counter_Enable(TCPWM_TIMER_HW, TCPWM_TIMER_NUM);

    /* Start the timer with the configured settings */
    Cy_TCPWM_TriggerStart_Single(TCPWM_TIMER_HW, TCPWM_TIMER_NUM);
 }


/*******************************************************************************
* Function Name: isr_timer
********************************************************************************
* Summary:
* This is the interrupt handler function for the timer interrupt.
*
* Parameters:
```

```
*    callback_arg    Arguments passed to the interrupt callback
*    event            Timer/counter interrupt triggers
*
* Return:
*  void
*******************************************************************************/
void isr_timer(void)
{
    /* Get interrupt source */
    uint32_t intrMask = Cy_TCPWM_GetInterruptStatusMasked(TCPWM_TIMER_HW, TCPWM_TIMER_NUM);

    /* Clear interrupt source */
    Cy_TCPWM_ClearInterrupt(TCPWM_TIMER_HW, TCPWM_TIMER_NUM, intrMask);

    /* Set interrupt flag */
    timer_interrupt_flag = true;
}

/* [] END OF FILE */
```

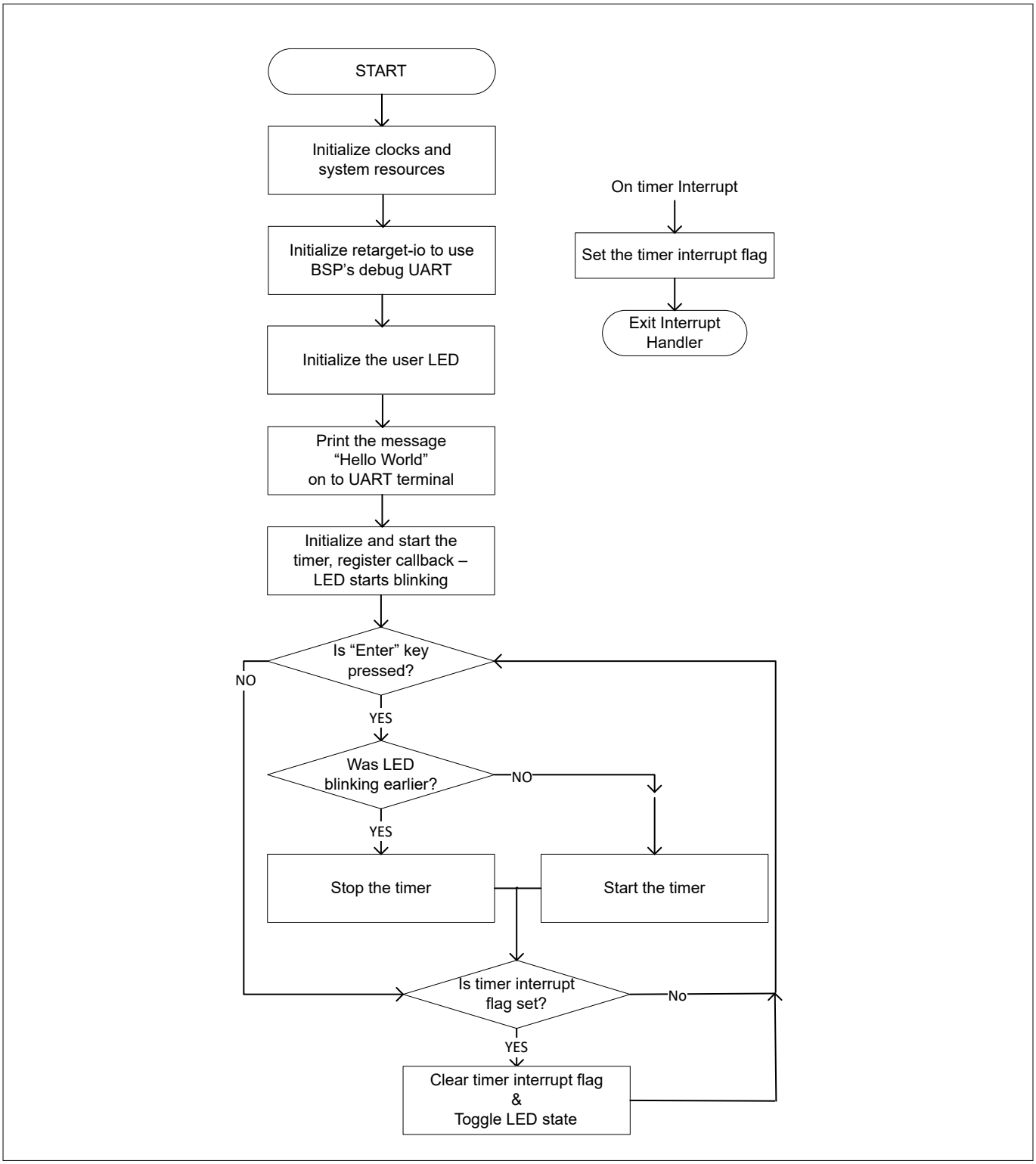


Figure 14 Firmware flowchart

This completes the summary of how the firmware works in the code example. Explore the source files for a deeper understanding.

5.5.1.3 Build the application

This section shows how to build the application.

1. Select the application project in the **Project Explorer** view
2. Click the **Build Project** shortcut under the **Hello_World** group in the **Quick Panel**

   It selects the build configuration from the Makefile and compiles/links all projects that constitute the application. By default, Debug configurations are selected. The **Console view** lists the results of the build operation, as Figure 15 shows

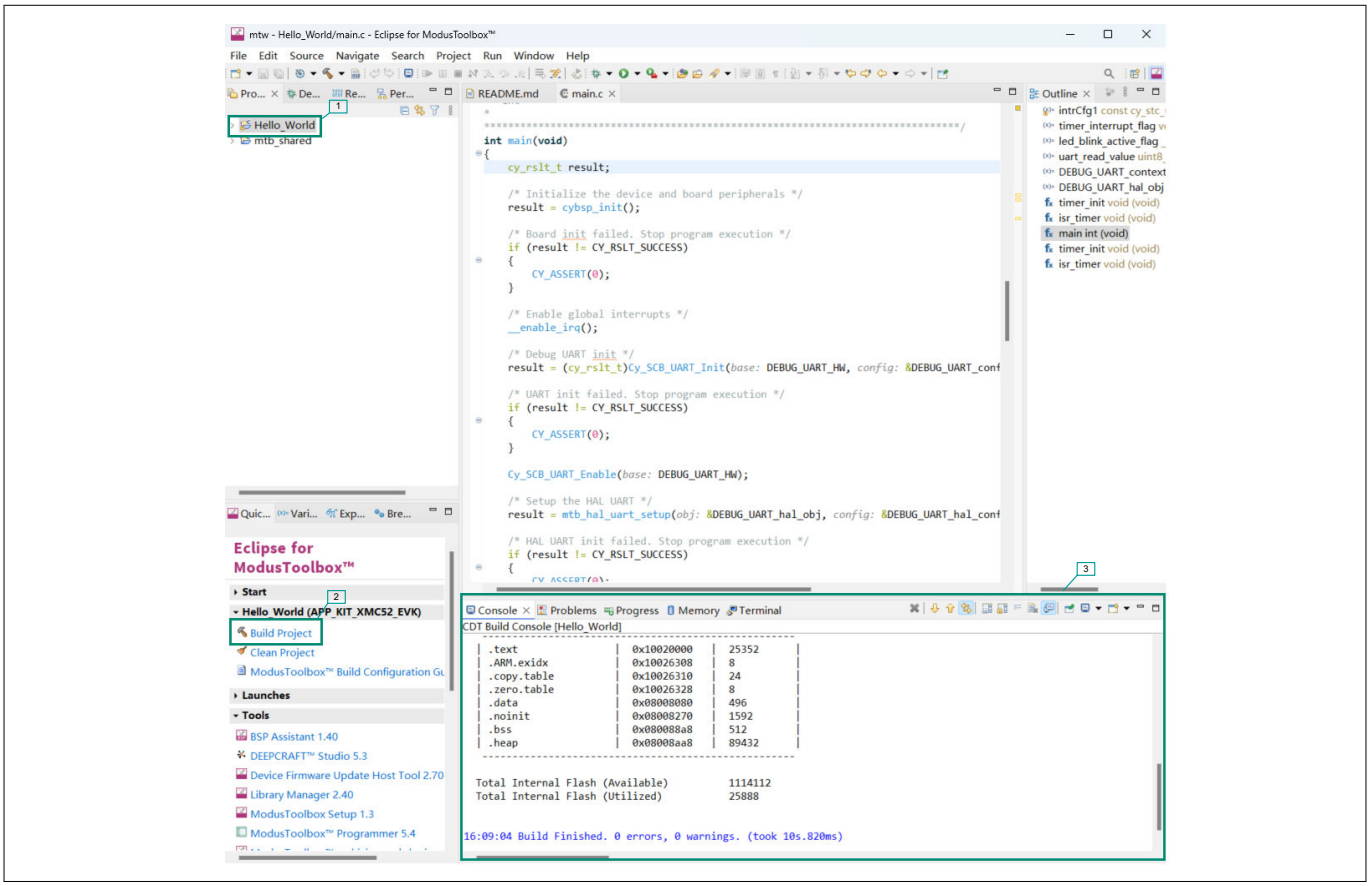


Figure 15 Build the application

If you encounter errors, revisit the earlier steps to ensure that you completed all the required tasks.

***Note**: You can also use the command-line interface (CLI) to build the application. See the **Build system** section in the ModusToolbox™ tools package user guide. This document is located in the `/docs_<version>/` folder in the ModusToolbox™ installation.*

5.5.1.4 Program the device

This section shows how to program the XMC5000 MCU.

ModusToolbox™ software uses the OpenOCD protocol to program and debug applications on XMC5000 MCU on the evaluation kit.

As the evaluation kit is with a built-in programmer, connect the board to your computer using the provided USB cable.

5 Getting started with XMC5000 MCU design

If you are developing on your own hardware, you may need a hardware programmer/debugger, for example, J-Link, or ULinkpro, or MiniProg.

1. Program the application:
   a. Connect to the board
   b. Select the application project and click the **Hello_World Program (KitProg3_MiniProg4)** shortcut under the **Launches** group in the Quick Panel, as Figure 16 shows. The IDE will select and run the appropriate run configuration. Note that this step will also perform a build if any files have been modified since the last build

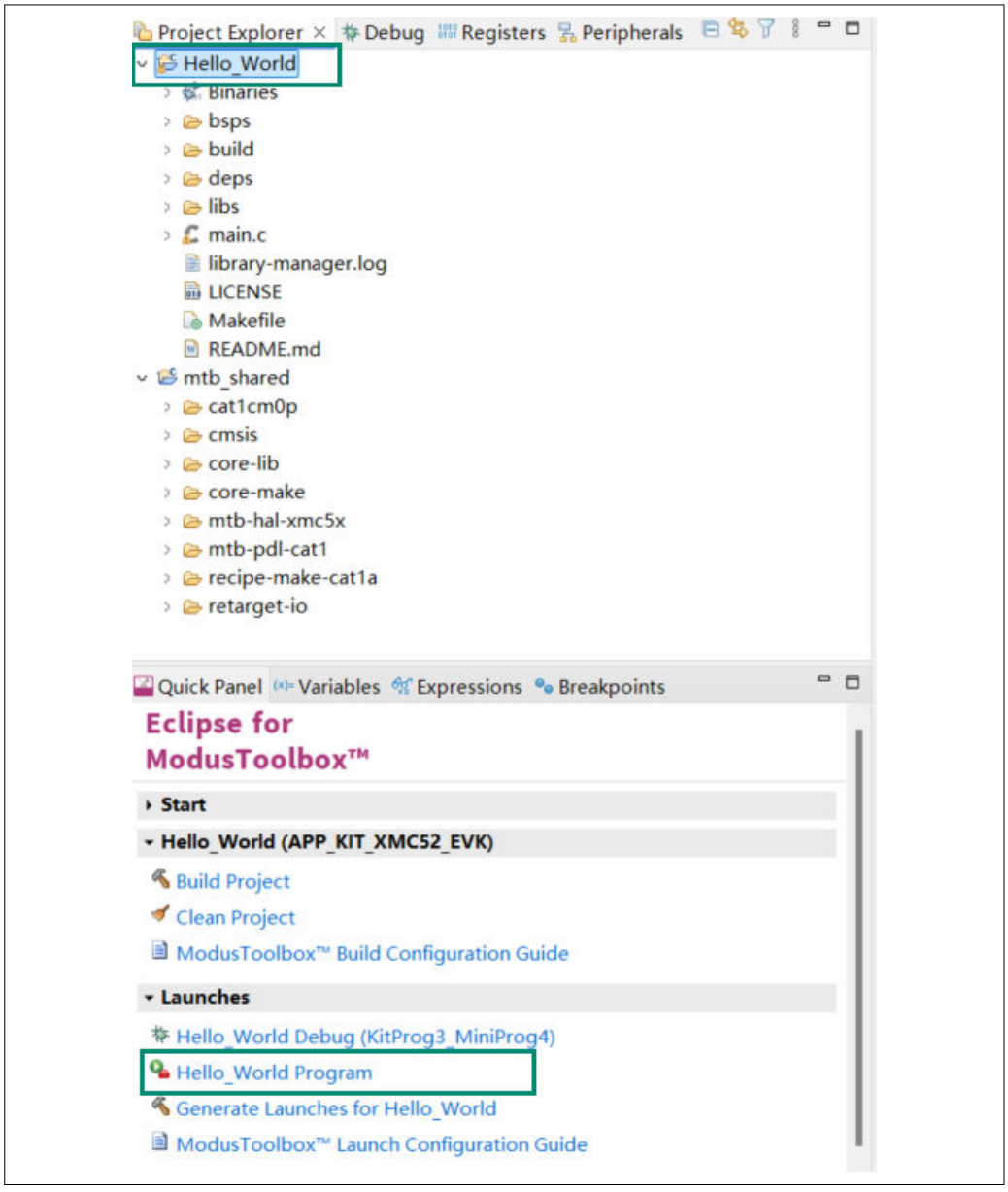


Figure 16 Programming an application to a device

The Console view lists the results of the programming operation, as Figure 17 shows.

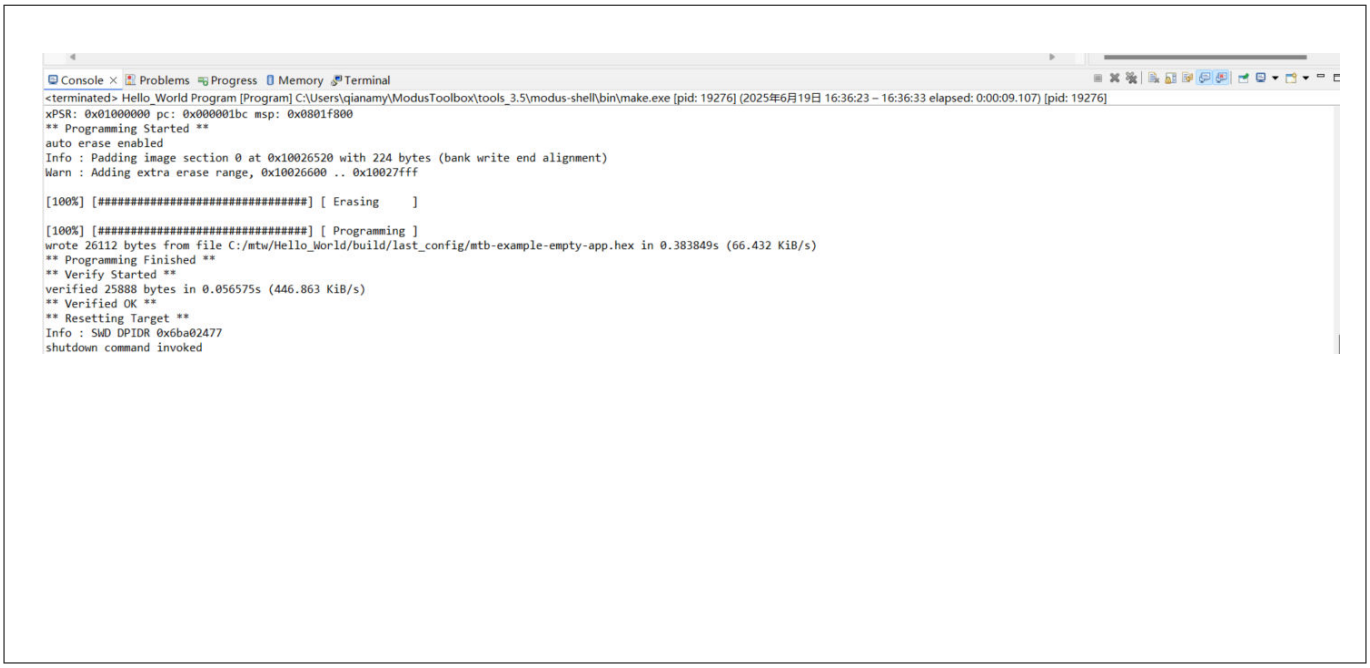


Figure 17 Console – programming results

5.5.1.5 Test your design

This section describes how to test your design.

Follow these steps to observe the output of your design. This document uses Tera Term as the UART terminal emulator to view the results, but you can use any terminal of your choice to view the output.

1. **Select the serial port**

   Launch Tera Term and select the USB-UART COM port as Figure 18 shows. Note that your COM port number may be different.

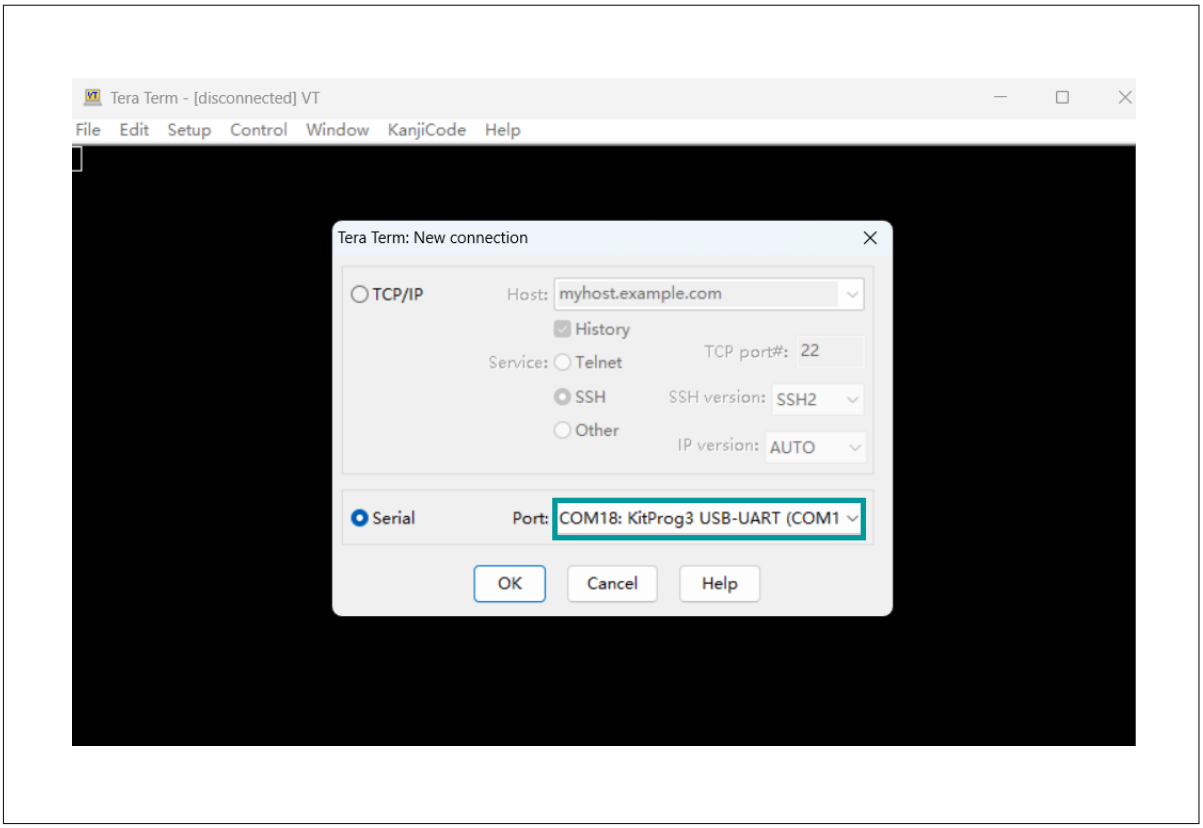


Figure 18 Selecting the KitProg3 COM port in Tera Term

2. Set the baud rate

Set the baud rate to 115200 under **Setup** > **Serial port** as Figure 19 shows.

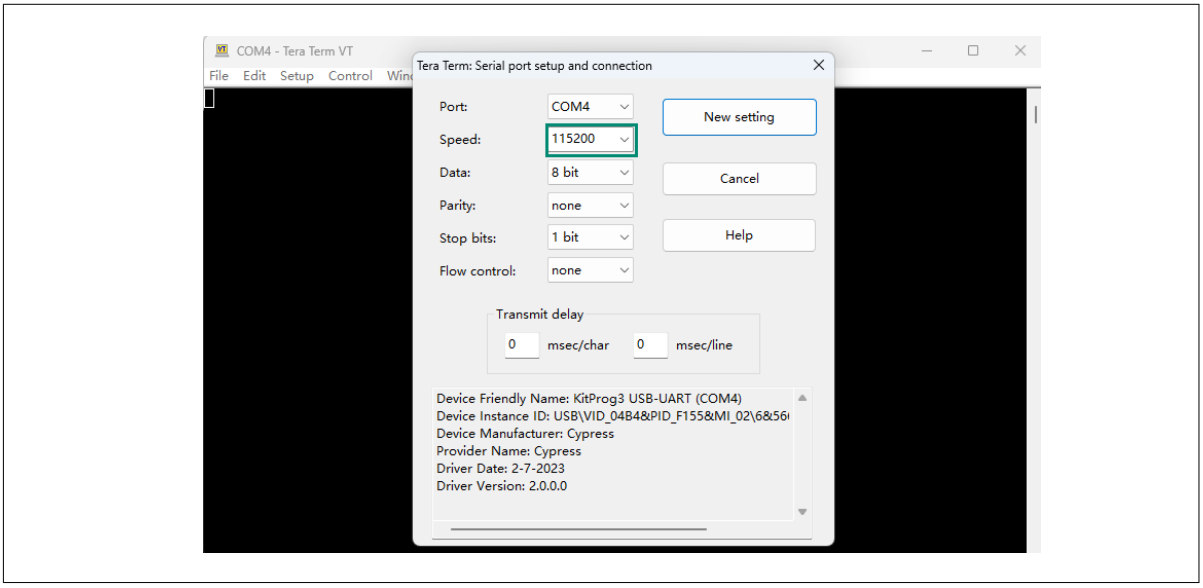


Figure 19 Configuring the baud rate in Tera Term

3. Reset the device

Press the reset switch (**SW1**) on the kit. The message shown in Figure 20 appears on the terminal. The user LED on the kit will start blinking.

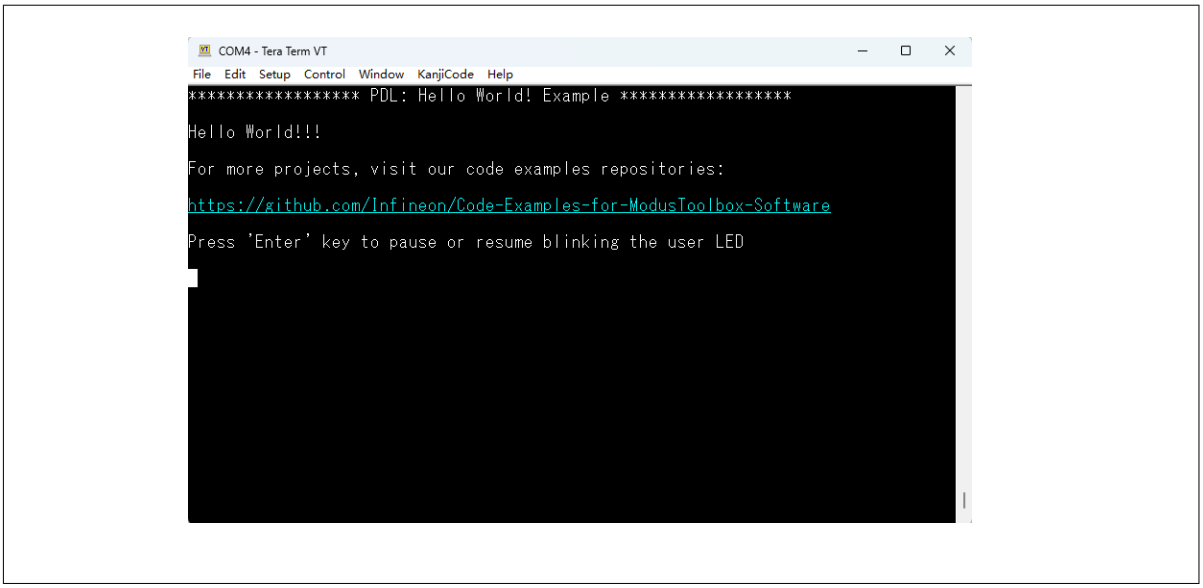


Figure 20 Printed UART message

4. Pause/resume LED blinking functionality

Press the **Enter** key to pause/resume blinking the LED. When the LED blinking is paused, a corresponding message will be displayed on the terminal as shown in Figure 21.

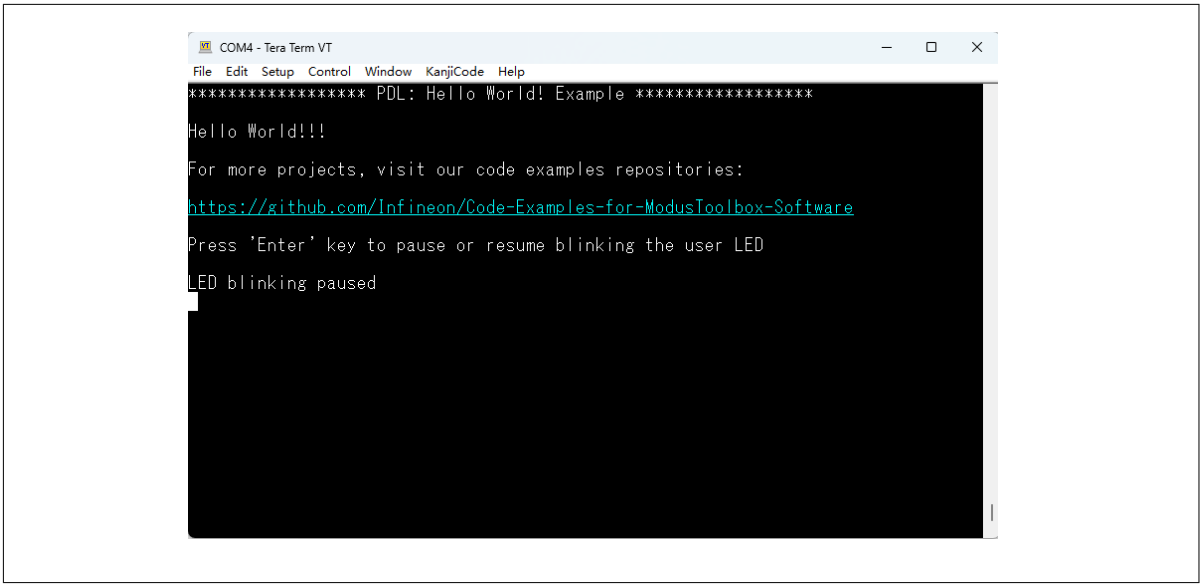


Figure 21 Printed UART message

5.5.1.6 Debugging the application using KitProg3/MiniProg4

XMC5000 kits come with either the KitProg3 or J-Link onboard programmer/debugger. See the KitProg3 user guide for details of KitProg3 or see the J-Link user guide for the details of J-Link.

The Eclipse IDE contains several launch configurations that control various settings for programming the devices and launching the debugger. Depending on the kit and the type of applications you are using, there are various launch configurations available. One such configuration is the KitProg3/MiniProg4 launch configuration. See the "PSOC™ MCU programming/debugging" section in the Eclipse IDE for ModusToolbox™ user guide for more details on the launch configurations.

5 Getting started with XMC5000 MCU design

When an application is created, the tool generates the launch configurations for KitProg3_MiniProg4 or J-link under **Launches** in the **Quick Panel**. For the XMC5000 Evaluation Kit, it will generate launch configurations for KitProg3, as shown in the following figure.

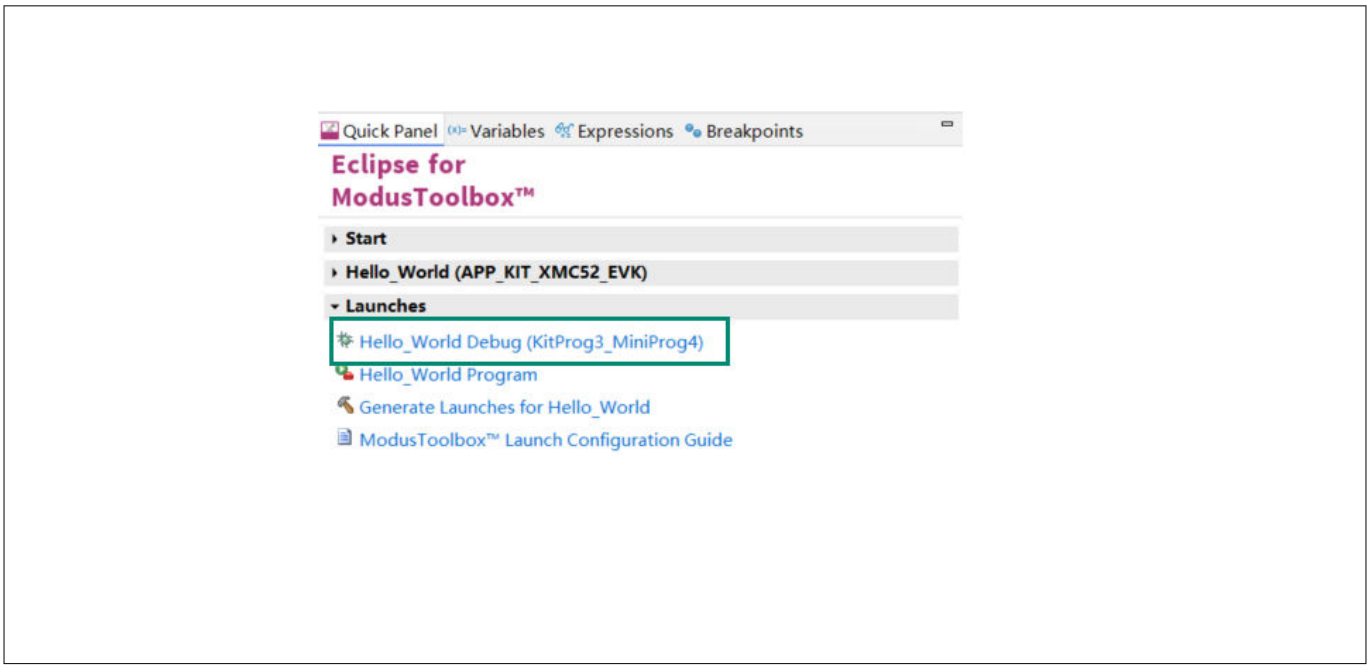


Figure 22 KitProg3/MiniProg4 launch configuration

Connect the device to the host machine and click the **Hello_World Debug (KitProg3_MiniProg4)** launch to start debugging, as shown in Figure 22. Once the debugging starts, the execution halts at the `main()` function, and you can start debugging from the start of `main()` as shown in the following figure.

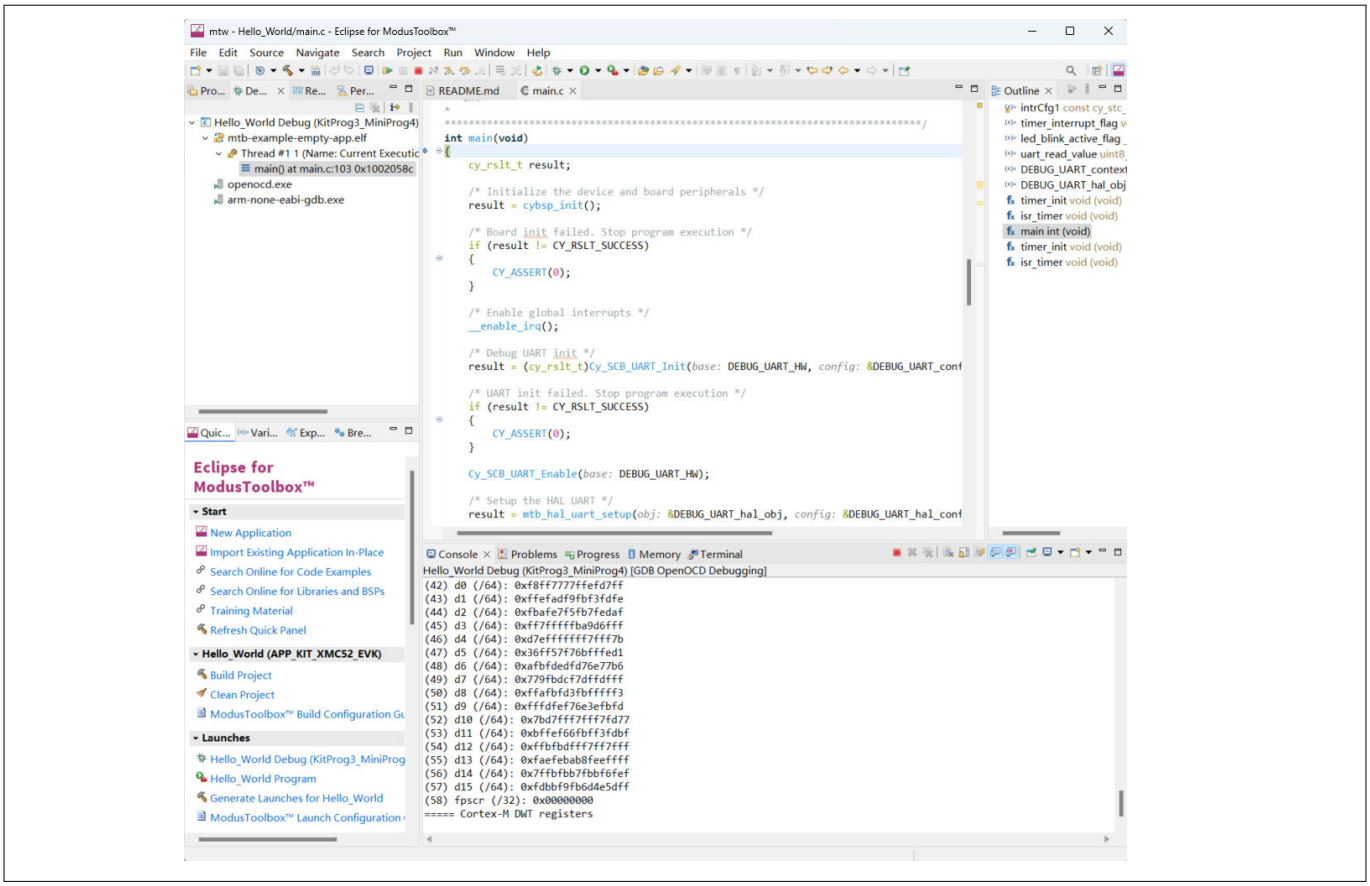


Figure 23 Debug main()

5.5.1.7 Multi-core project

1. Create a multi-core project by selecting **Multicore Empty App** during project creation. See the Create a new application section for more details. Rename the project as required

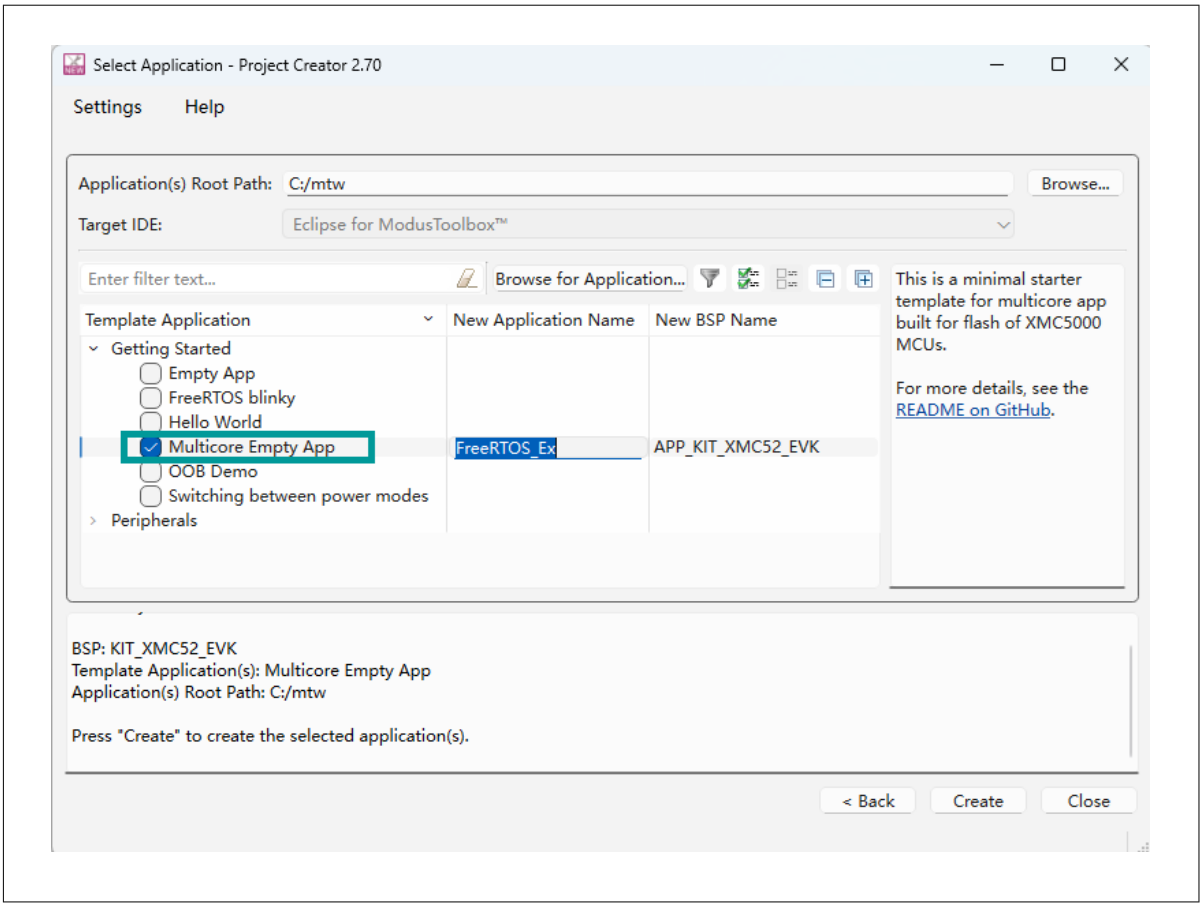


Figure 24 Create Multicore Empty App

2. Open **Library Manager**, select **proj_cm0p Libraries**, click **Add Library**, select **freertos**, click **OK**, and then **Update** to add the library to the project as shown in Figure 25

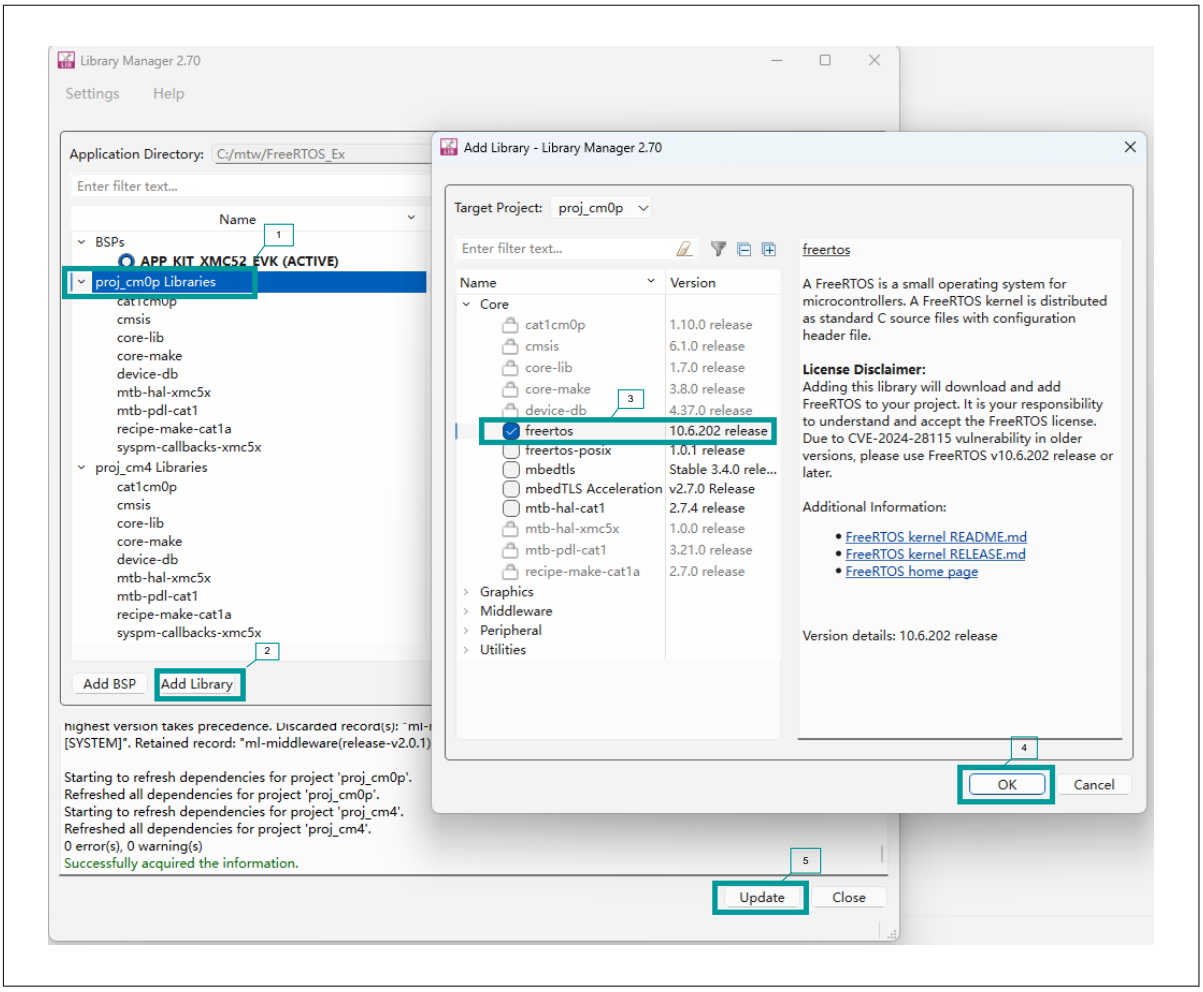


Figure 25 Adding FreeRTOS library to cm0p project

3. Select **proj_cm4 Libraries**, click **Add Library**, add **freertos**, click **OK**, and then **Update** to add the library to the project as shown in Figure 26

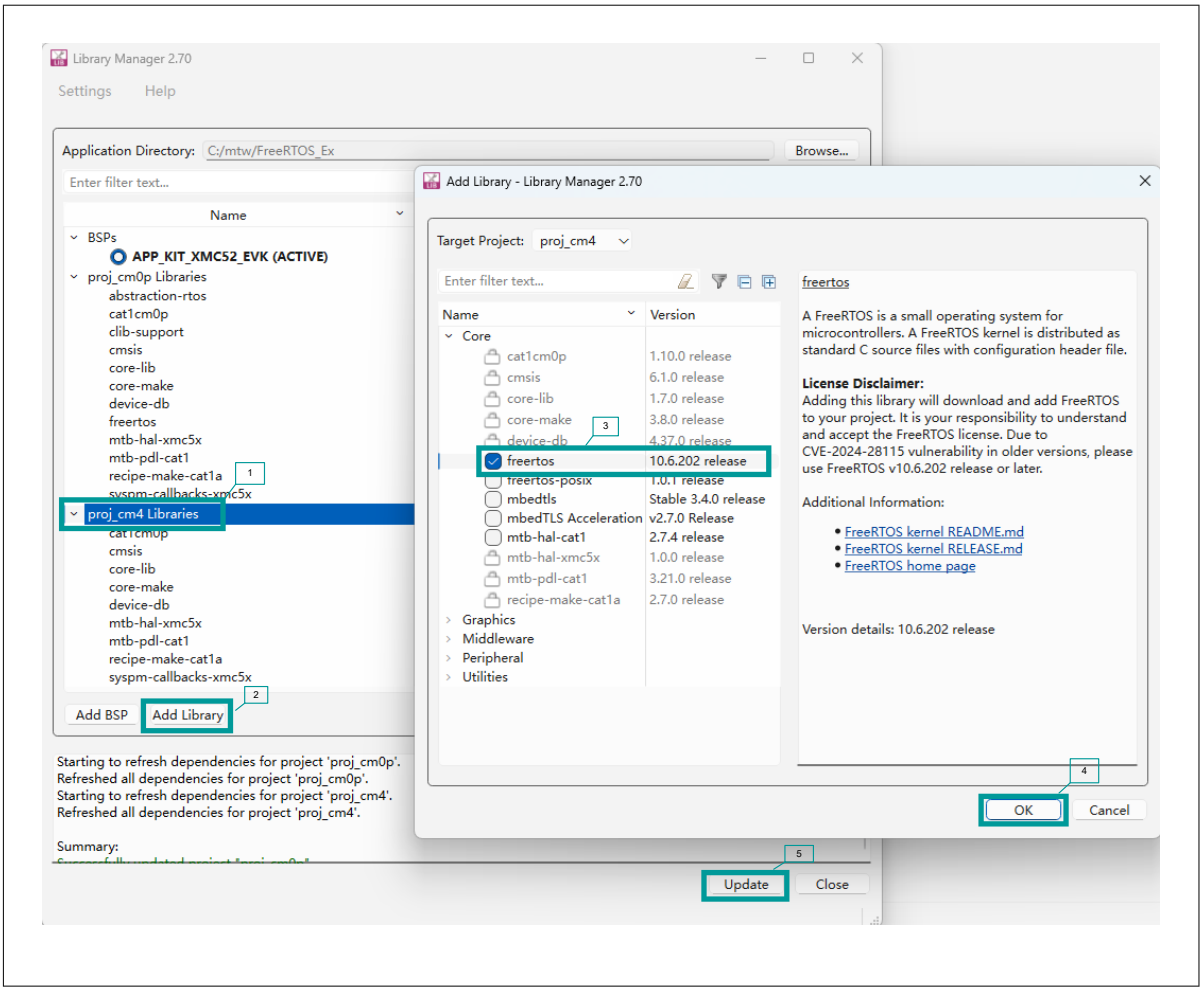


Figure 26 Adding FreeRTOS library to cm4 project

4. Add the following line to the Makefile of two projects:

```
COMPONENTS+=FREERTOS
```

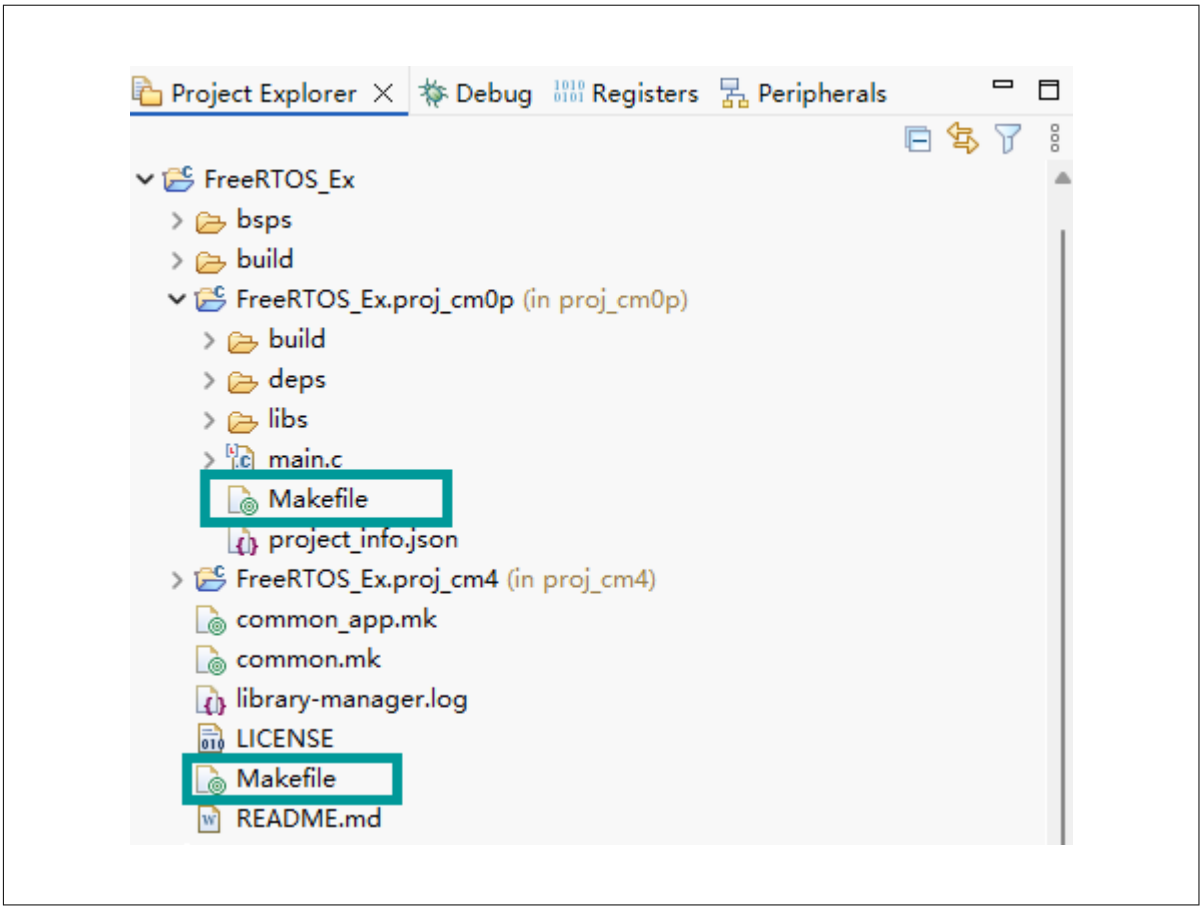


Figure 27 Makefile of cm0p and cm4 projects

5. Copy the `FreeRTOSConfig.h` from https://github.com/Infineon/freertos/blob/master/Source/portable/COMPONENT_CM0/FreeRTOSConfig.h to `proj_cm0p` project
6. Copy the `FreeRTOSConfig.h` from https://github.com/Infineon/freertos/blob/master/Source/portable/COMPONENT_CM0/FreeRTOSConfig.h to `proj_cm4` project

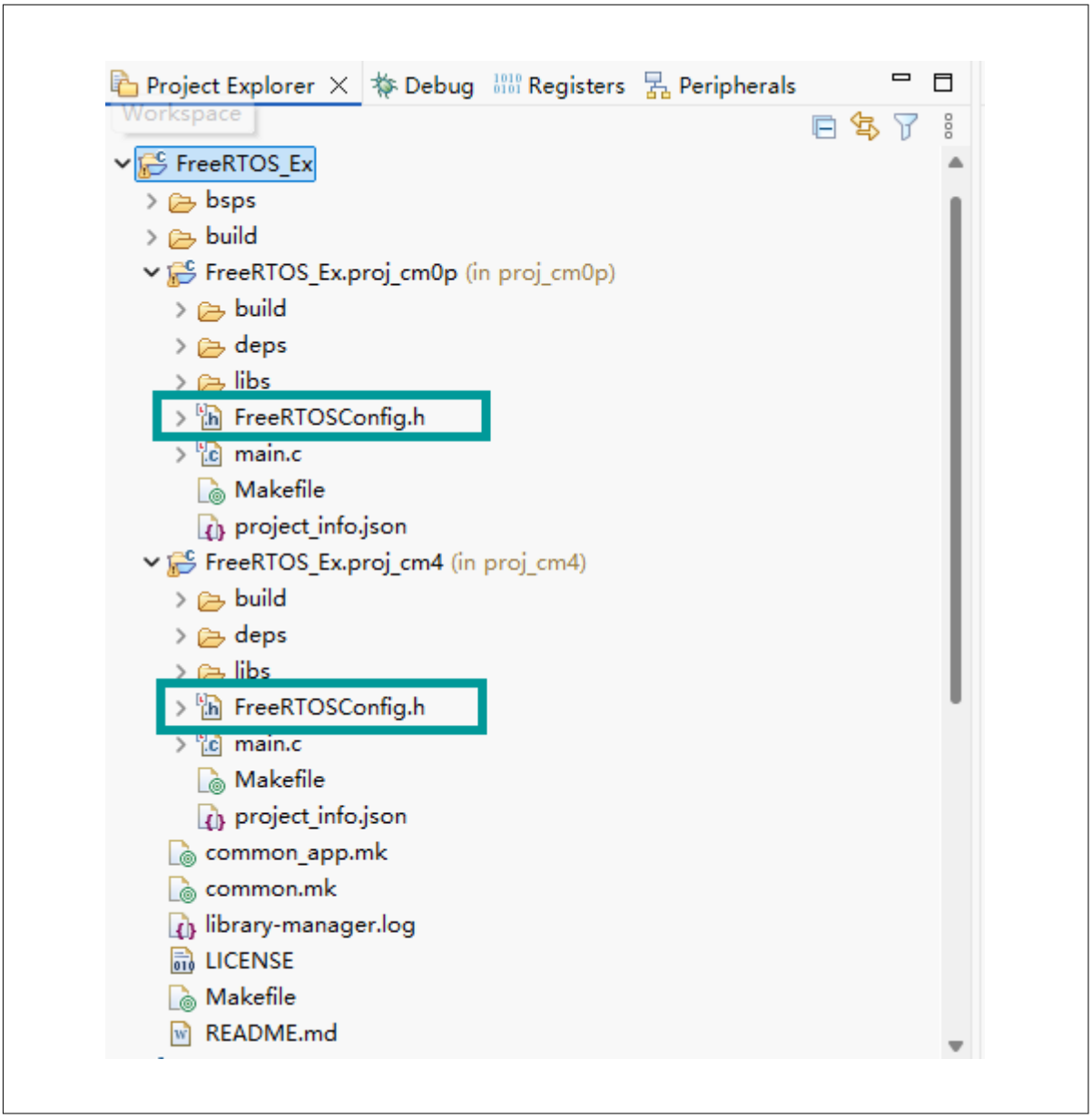


Figure 28 Project structure after adding FreeRTOSConfig.h

7. Open the copied `FreeRTOSConfig.h` file and remove the #warning. This is a template line. Ensure to do so in all two `FreeRTOSConfig.h` files of the cm0p and cm4 projects
8. The FreeRTOS library requires MCWDT to periodically trigger the tasks. MCWDT0 and MCWDT1 are assigned to be used by the CM0P and CM4 cores respectively

9. Replace the `main.c` file of the `proj_cm0p` project with the following code snippet:

```
/*******************************************************************************
* Header Files
*******************************************************************************/
#include "cy_pdl.h"
#include "cybsp.h"
#include "FreeRTOS.h"
#include "task.h"

/*******************************************************************************
* Function Prototypes
*******************************************************************************/
/*******************************************************************************
* Function Name: vBlinkTask
********************************************************************************
* Summary:
* This is a FreeRTOS task that is used to blink an LED.
*
* Parameters:
* void
*
* Return:
* int
*
*******************************************************************************/
void vBlinkTask(void *pvParameters)
{
    (void)pvParameters;
    for (;;) {
        Cy_GPIO_Inv(CYBSP_USER_LED1_PORT, CYBSP_USER_LED1_PIN );
        vTaskDelay(pdMS_TO_TICKS(500));
    }
}

/*******************************************************************************
* Function Name: MCWDT_ISR
********************************************************************************
* Summary:
* This is MCWDT Interrupt Service Function.
*
* Parameters:
* void
*
* Return:
* int
*
*******************************************************************************/
void MCWDT_ISR(void)
{
    Cy_MCWDT_ClearInterrupt(MCWDT0, CY_MCWDT_CTR0);

    if (xTaskIncrementTick() != pdFALSE) {
```

```
        portYIELD_FROM_ISR(pdTRUE);
    }
}
/*******************************************************************************
* Function Name: main
********************************************************************************
* Summary:
* This function creates a FreeRTOS task.
*
* Parameters:
*  void
*
* Return:
*  int
*
*******************************************************************************/
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 CM4. CY_CORTEX_M4_APPL_ADDR must be updated if CM4 memory layout is
changed. */
    Cy_SysEnableCM4(CY_CORTEX_M4_APPL_ADDR);

    /* Enable global interrupts */
    __enable_irq();

    /*Create a FreeRTOS task*/
    xTaskCreate(vBlinkTask, "Blinky", configMINIMAL_STACK_SIZE * 2, NULL,
tskIDLE_PRIORITY + 1, NULL);

     /* Start the FreeRTOS scheduler */
    vTaskStartScheduler();

    for (;;)
    {

    }
}

/* [] END OF FILE */
```

10. Replace the `main.c` file of the `proj_cm4` project with following code snippet:

```
/*******************************************************************************
* Header Files
*******************************************************************************/
#include "cy_pdl.h"
#include "cybsp.h"
#include "FreeRTOS.h"
#include "task.h"

/*******************************************************************************
* Function Prototypes
*******************************************************************************/
/*******************************************************************************
* Function Name: vTaskCM4
********************************************************************************
* Summary:
* This is a FreeRTOS task that is used to blink an LED.
*
* Parameters:
* void
*
* Return:
* int
*
*******************************************************************************/
void vTaskCM4(void *pvParameters)
{
    (void)pvParameters;
    for (;;) {
        Cy_GPIO_Inv(CYBSP_USER_LED2_PORT, CYBSP_USER_LED2_PIN );
        vTaskDelay(pdMS_TO_TICKS(1000));
    }
}

/*******************************************************************************
* Function Name: main
********************************************************************************
* Summary:
* This is the main function for CM4 CPU.
*
* Parameters:
*  void
*
* Return:
*  int
*
*******************************************************************************/
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 global interrupts */
    __enable_irq();

    /*Create FreeRTOS tasks for the CM4*/
    xTaskCreate(vTaskCM4, "CM4_Task", configMINIMAL_STACK_SIZE * 3, NULL,
tskIDLE_PRIORITY + 2, NULL);

    /* Start the FreeRTOS scheduler */
    vTaskStartScheduler();

    for (;;)
    {
    }
}

/* [] END OF FILE */
```

11. Build the project and program the KIT_XMC52_EVK. See the Program the device section for more details on programming the board
12. Observe that two LEDs on the XMC52 EVK starts blinking. The FreeRTOS component has created one task in each core. The LEDs are being handled within these tasks. All two tasks are getting triggered successfully

5.5.2 Visual Studio Code for ModusToolbox™

See the Visual Studio Code for ModusToolbox™ user guide for creating a new application on VS Code.

5.5.3 IAR Embedded Workbench for ModusToolbox™

See the IAR Embedded Workbench for ModusToolbox™ user guide for creating a new application on IAR Embedded Workbench.

5.5.4 Arm® Keil® µVision® for ModusToolbox™

See the Arm® Keil® µVision® for ModusToolbox™ user guide for creating a new application on Keil® µVision®.

6 Summary

This application note explores the XMC5000 MCU family device architecture and the associated development tools. The XMC5000 MCU family is a truly-programmable embedded system-on-chip with configurable analog and digital peripheral functions, memory, and a dual-CPU system on a single chip.

References

Webpage

[1] Infineon Technologies AG: *XMC5000 webpage*; Available online

Kit guide

[2] Infineon Technologies AG: *KIT_XMC52_EVK Evaluation Kit user guide*; Available online

Code examples

[3] Infineon Technologies AG: *GitHub*; Available online

ModusToolbox™

[4] Infineon Technologies AG: *ModusToolbox™ software*; Available online
[5] Infineon Technologies AG: *ModusToolbox™ tools package installation guide*; Available online
[6] Infineon Technologies AG: *ModusToolbox™ tools package release notes*; Available online
[7] Infineon Technologies AG: *ModusToolbox™ tools package quick start guide*; Available online
[8] Infineon Technologies AG: *ModusToolbox™ user guide* Available online
[9] Infineon Technologies AG: *Eclipse IDE for ModusToolbox™ user guide* Available online
[10] Infineon Technologies AG: *Visual Studio Code for ModusToolbox™ user guide*; Available online
[11] Infineon Technologies AG: *Arm® Keil® µVision® for ModusToolbox™ user guide*; Available online
[12] Infineon Technologies AG: *IAR Embedded Workbench for ModusToolbox™ user guide*; Available online

7 Glossary

This section lists the most commonly used terms that you might see while working with XMC™ family of devices.

- Board support package (BSP): A BSP is the layer of firmware containing board-specific drivers and other functions. The board support package is a set of libraries that provide firmware APIs to initialize the board and provide access to board level peripherals
- **KitProg**: The KitProg is an onboard programmer/debugger with USB-I2C and USB-to-UART bridge functionality. The KitProg is integrated onto most XMC™ development kits
- MiniProg3/MiniProg4: Programming hardware for development that is used to program XMC™ devices on your custom board or XMC™ development kits that do not support a built-in programmer
- **Personality**: A personality expresses the configurability of a resource for a functionality. For example, the SCB resource can be configured to be an UART, SPI, or I2C personalities
- Middleware: Middleware is a set of firmware modules that provide specific capabilities to an application. Some middleware may provide network protocols (e.g. MQTT), and some may provide high-level software interfaces to device features (e.g. USB, audio)
- ModusToolbox™: An Eclipse-based embedded design platform for embedded systems designers that provides a single, coherent, and familiar design experience, combining the industry's most deployed Wi-Fi and Bluetooth® technologies, and the lowest power, most flexible MCUs with best-in-class sensing
- Peripheral Driver Library: The peripheral driver library (PDL) simplifies software development for the XMC5000 MCU architecture. The PDL reduces the need to understand register usage and bit structures, thus easing software development for the extensive set of peripherals available

Revision history

Document revision	Date	Description of changes
**	2025-09-01	Initial release
*A	2026-04-30	• Edits throughout the document • Added XMC5000 multi-core project structure and Multi-core project sections • Updated to ModusToolbox™ 3.8

Trademarks

The Bluetooth® word mark and logos are registered trademarks owned by Bluetooth SIG, Inc., and any use of such marks by Infineon is under license.

PSOC™, formerly known as PSoC™, is a trademark of Infineon Technologies. Any references to PSoC™ in this document or others shall be deemed to refer to PSOC™.

Trademarks

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

Edition 2026-04-30

Published by

Infineon Technologies AG

81726 Munich, Germany

© 2026 Infineon Technologies AG

All Rights Reserved.

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

Email: erratum@infineon.com

Document reference
IFX-qtz1745565373822

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.