Getting started with XMC7000 MCU on ModusToolbox™ software

About this document

Scope and purpose

This application note helps you explore the XMC7000 MCU architecture and development tools and shows you how to create your first project using the Eclipse IDE for ModusToolbox™ software. This application note also guides you to more resources available online to accelerate your learning about XMC7000 MCU.

Intended audience

This document is intended for the users who are new to XMC7000 MCU and ModusToolbox™ software.

Associated part family

All **XMC7000 MCU** devices

Software version

ModusToolbox™ software 3.0 or above

More code examples? We heard you.

To access an ever-growing list of XMC7000 code examples using ModusToolbox™, please visit the **GitHub** site.

Table of contents

Table of contents

1 Introduction

The XMC7000 device is a microcontroller targeted at industrial applications. The XMC7000 integrates the following features on a single chip:

- Up to two 350-MHz 32-bit Arm® Cortex®-M7 CPUs each with
  - Single-cycle multiply
  - Single-/double-precision floating point unit (FPU)
  - 16-KB data cache, 16-KB instruction cache
  - Memory protection unit (MPU)
  - 16-KB instruction and 16-KB data tightly-coupled memory (TCM)
- 100-MHz 32-bit Arm® Cortex® M0+ CPU with single-cycle multiply and MPU
- Programmable analog and digital peripherals
- Up to 8384 KB of code flash with an additional, up to 256 KB of work flash and an internal SRAM of up to 1024 KB
- XMC7000 MCU is suitable for a variety of power-sensitive applications such as:
  - Wireless charging
  - Lighting
  - Power supply servers
  - Robots and drones
  - MHA
  - Industrial drives
  - PLC
  - I/O modules
  - Electric two-wheelers

The **ModusToolbox™ software environment** supports XMC7000 MCU application development with a set of tools for configuring the device, setting up peripherals, and complementing your projects with world-class middleware. See the **Infineon** GitHub repos for BSPs for all kits, libraries for popular functionality like DFU and emWin, and a comprehensive array of **example applications** to get you started.

Figure 1 illustrates an application-level block diagram for a real-world use case using XMC7000 MCU.

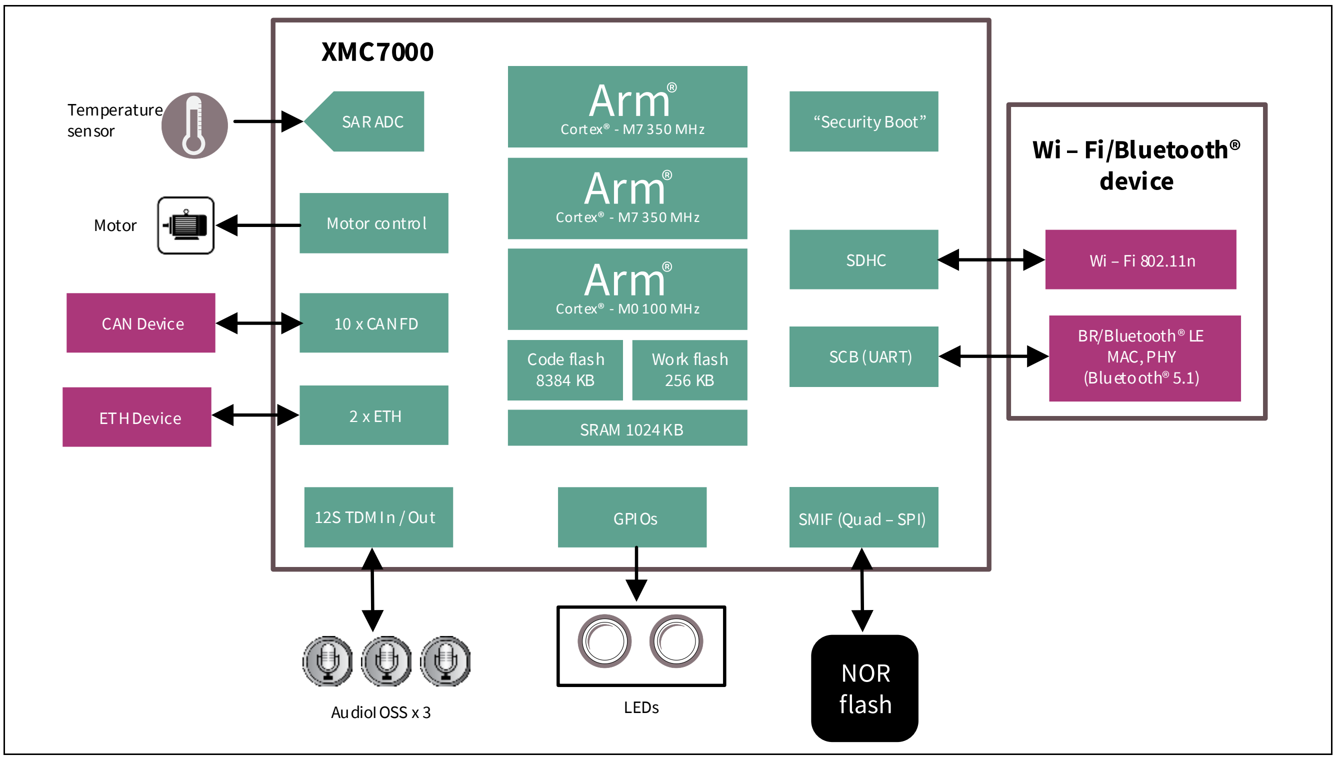


Figure 1 **Application-level block diagram using XMC7000 MCU**

XMC7000 MCU is a highly capable and flexible solution. For example, the real-world use case in **Figure 1** takes advantage of these features:

- A buck converter for ultra-low-power operation
- An analog front end (AFE) within the device to condition and measure sensor outputs such as ambient light sensor
- Serial Communication Blocks (SCBs) to interface with multiple digital sensors such as motion sensors
- Programmable digital logic (smart I/O) and peripherals (Timer Counter PWM or TCPWM) to drive the motor and LEDs respectively
- Up to 10 CAN FD channels with increased data rate (up to 8 Mbps) supports all the requirement of CAN FD specification V1.0 for non-ISO CAN FD
- Up to two 10/100/1000 Mbps Ethernet MAC interfaces conforming to IEEE-802.3az supports MII/RMII/RGMII/AVB/PTP PHY interfaces
- SDIO interface to a Wi-Fi/Bluetooth® device to provide IoT cloud connectivity
- Product security features managed by CM0+ CPU and application features executed by CM7 CPUs

There are two product lines in XMC7000. **Table 1** provides overview of different product lines:

Table 1 XMC7000 MCU product lines

Device series	Details
XMC7100	Triple-core architecture: 250-MHz Arm® Cortex®-M7 and 100-MHz Cortex®-M0+ 4-MB flash, 768-KB RAM Packages: 100/144/176 TEQFP, 272 BGA
XMC7200	Triple-core architecture: 350-MHz Arm® Cortex®-M7 and 100-MHz Cortex®-M0+ 8-MB flash, 1-MB RAM Packages: 176 TEQFP, 272 BGA

Note: Not all the features available in all the devices in a product line. See the ***device datasheets*** *for more details.*

This application note introduces you to the capabilities of the XMC7000 MCU, gives an overview of the development ecosystem, and gets you started with a simple 'Hello World' application wherein you learn to use the XMC7000 MCU. We will show you how to create the application from an empty starter application, but the completed design is available as a **code example for ModusToolbox™ on GitHub**.

For hardware design considerations, see **Hardware design guide for the XMC7000 Family**.

2 Development ecosystem

2.1 XMC7000 resources

A wealth of data available **here** helps you to select the right XMC™ device and quickly and effectively integrate it into your design. For a comprehensive list of XMC7000 MCU resources, see **How to design with XMC7000 MCU**. The following is an abbreviated list of resources for XMC7000 MCU.

- **Overview: XMC7000 MCU webpage**
- **Product selectors: XMC7000 MCU**
- **Datasheets** describe and provide electrical specifications for each device family.
- **Application notes** and **code examples** cover a broad range of topics, from basic to advanced level. You can also browse our collection of code examples. See **Code examples**.
- **Technical reference manuals (TRMs)** provide detailed descriptions of the architecture and registers in each device family.
- **XMC7000 MCU programming specification** provides the information necessary to program the nonvolatile memory of XMC7000 MCU devices.
- **Development tools:** Many low-cost **kits** are available for evaluation, design, and development of different applications using XMC7000 MCUs.
- **Technical Support: XMC7000 community forum**, **knowledge base articles**

2.2 Firmware/application development

There are one development platforms that you can use for application development with XMC7000 MCU:

- **ModusToolbox™:** ModusToolbox™ software includes configuration tools, low-level drivers, middleware libraries, and operating system support, as well as other packages that enable you to create MCU and wireless applications. It also includes an Eclipse IDE, which provides an integrated flow with all the ModusToolbox™ tools.

  ModusToolbox™ supports stand-alone device and middleware configurators that are fully integrated into the Eclipse IDE. Use the configurators to set the configuration of different blocks in the device and generate code that can be used in firmware development. ModusToolbox™ supports all XMC7000 MCU devices. It is recommended that you use ModusToolbox™ for all application development for XMC7000 MCUs. See the **ModusToolbox™ user guide** for more information.

  Libraries and enablement software are available at the **GitHub** site.

  ModusToolbox™ tools and resources can also be used in the command line. See the "Using command-line" section in the **ModusToolbox™ user guide** for detailed documentation.

2.2.1 Choosing an IDE

ModusToolbox™ software, the latest-generation toolset, includes the Eclipse IDE and therefore is supported across Windows, Linux, and macOS platforms. Eclipse IDE for ModusToolbox™ is integrated with quick launchers for tools and design configurators in the Quick Panel. ModusToolbox™ also supports third-party IDEs, including Visual Studio Code, Arm® MDK (µVision), and IAR Embedded Workbench. The tool supports all XMC7000 MCU devices. The associated hardware and middleware configurators also work on all three host operating systems.

Use ModusToolbox™ to take advantage of the power and extensibility of an Eclipse-based IDE. ModusToolbox™ is supported on Windows, Linux, and macOS. It is recommended to use ModusToolbox™.

2.2.2 ModusToolbox™ software

ModusToolbox™ is a set of tools and software that enables an immersive development experience for creating converged MCU and wireless systems and enables you to integrate our devices into your existing development methodology. To achieve this goal, ModusToolbox™ leverages popular third-party ecosystems such as FreeRTOS and Arm® Mbed, and adds specific features for Wi-Fi, Bluetooth®, CAPSENSE™, and security.

Eclipse IDE for ModusToolbox™ is a multi-platform development environment that supports application configuration and development.

Figure 2 shows a high-level view of the tools/resources included in the ModusToolbox™ software. For a more in-depth overview of the ModusToolbox™ software, see **ModusToolbox™ user guide**.

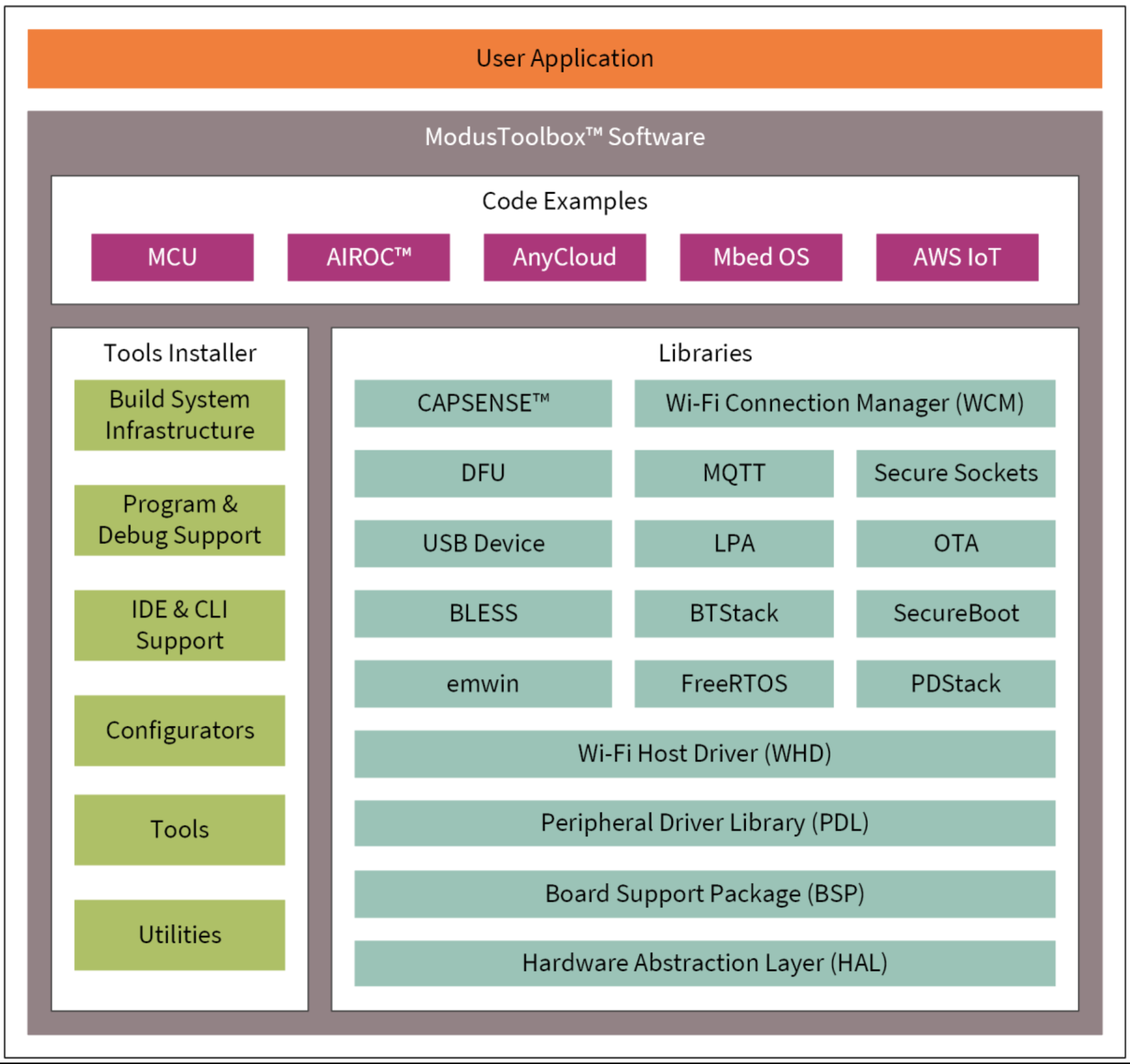


Figure 2 ModusToolbox™ software

ModusToolbox™ installer includes the design configurators and tools, and the build system infrastructure.

The build system infrastructure includes the new project creation wizard that can be run independent of the Eclipse IDE, the make infrastructure, and other tools.

All the ModusToolbox™ development flows depend on the provided low-level resources. These include:

- Board Support Packages (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 APIs to initialize the board and provide access to board level peripherals. It includes low-level resources such as Peripheral Driver Library (PDL) for XMC7000 MCU and has macros for board peripherals. It uses the HAL to configure the board. Custom BSPs can be created to enable support for end-application boards. Refer to the “Board Support Packages” section in **ModusToolbox™ user guide** for more information.
- **Hardware Abstraction Layer (HAL)** – HAL provides a high-level interface to configure and use hardware blocks on MCUs. It is a generic interface that can be used across multiple product families. The focus on ease-of-use and portability means that the HAL does not expose all the low-level peripheral functionality. The HAL wraps the lower-level drivers (like XMC7000 PDL) and provides a high-level interface to the MCU. The interface is abstracted to work on any MCU. This helps you write application firmware independent of the target MCU.

  The HAL can be combined with platform-specific libraries (such as XMC7000 PDL) within a single application. You can leverage the HAL's simpler and more generic interface for most of an application, even if one portion requires finer-grained control.
- XMC7000 **Peripheral Driver Library (PDL)** – The PDL integrates device header files, startup code, and peripheral drivers into a single package. The PDL supports the XMC7000 MCU device family. The drivers abstract the hardware functions into a set of easy-to-use APIs. These are fully documented in the PDL API Reference.

  The PDL reduces the need to understand register usage and bit structures, thus easing software development for the extensive set of peripherals in the XMC7000 MCU series. You configure the driver for your application, and then use API calls to initialize and use the peripheral.
- Extensive middleware libraries that provides specific capabilities to an application. All the middleware is delivered as libraries and via GitHub repositories.

2.2.3 XMC7000 software resources

The XMC7000 software includes driver and middleware configurators to get you started developing firmware with XMC7000 MCU. It contains configurators, drivers, libraries, middleware, as well as various utilities, makefiles, and scripts. It also includes relevant drivers, middleware, and examples for use with industrial applications. You may use any or all tools in any environment you prefer.

2.2.3.1 Configurators

ModusToolbox™ software provides graphical applications called configurators that make it easier to configure a hardware block. For example, instead of having to search through all the documentation to configure a serial communication block as a UART with a desired configuration, open the appropriate configurator and set the baud rate, parity, and stop bits. Upon saving the hardware configuration, the tool generates the "C" code to initialize the hardware with the desired configuration.

Configurators are independent of each other, but they can be used together to provide flexible configuration options. They can be used stand alone, in conjunction with other tools, or within a complete IDE. Configurators are used for:

- Setting options and generating code to configure drivers
- Setting up connections such as pins and clocks for a peripheral
- Setting options and generating code to configure middleware

For XMC7000 MCU applications, the available configurators include:

- **Device Configurator:** Set up the system (platform) functions, as well as the basic peripherals (For example, UART, Timer, and PWM).
- **QSPI Configurator:** Configure external memory and generate the required code.
- **Smart I/O Configurator:** Configure the smart I/O.

Each of the above configurators create their own files (e.g.: *design.cyqspi* for QSPI). The configurator files (*design.modus* or *design.cyqspi*) are usually provided with the BSP. When an application is created based on a BSP, the files are copied into the application. You can also create custom device configurator files for an application and override the ones provided by the BSP.

2.2.3.2 Library management for XMC7000 MCU

With the release of ModusToolbox™ v3.0, applications can optionally share board support packages (BSPs) and libraries. If needed, different applications can use different versions of the same BSP/library. The file types associated with libraries using this flow have a *.mtb* extension.

Going further, Section 4 of this document describes creating a new application using this Flow.

For more information on MTB Flow, see the **Library Manager user guide** located at *<install_dir> /ModusToolbox/tools_<version>/library-manager/docs/library-manager.pdf.*

2.2.3.3 Software development for XMC7000 MCU

Significant source code and tools are provided to enable software development for XMC7000 MCUs. You use tools to specify how you want to configure the hardware, generate code for that purpose which you use in your firmware, and include various middleware libraries for additional functionality, like FreeRTOS. This source code makes it easier to develop the firmware for supported devices. It helps you quickly customize and build firmware without the need to understand the register set.

In the ModusToolbox™ environment, you use configurators to configure either the device, or a middleware library, like QSPI functionality.

The XMC7000 Peripheral Driver Library code is delivered as the **mtb-pdl-cat1** library. Middleware is delivered as separate libraries for each feature/function.

Whether you use Eclipse IDE, a third-party IDE, or the command line, firmware developers who wish to work at the register level should refer to the driver source code from the PDL. The PDL includes all the device-specific header files and startup code you need for your project. It also serves as a reference for each driver. Because the PDL is provided as source code, you can see how it accesses the hardware at the register level.

Some devices do not support particular peripherals. The PDL is a superset of all the drivers for any supported device. This superset design means:

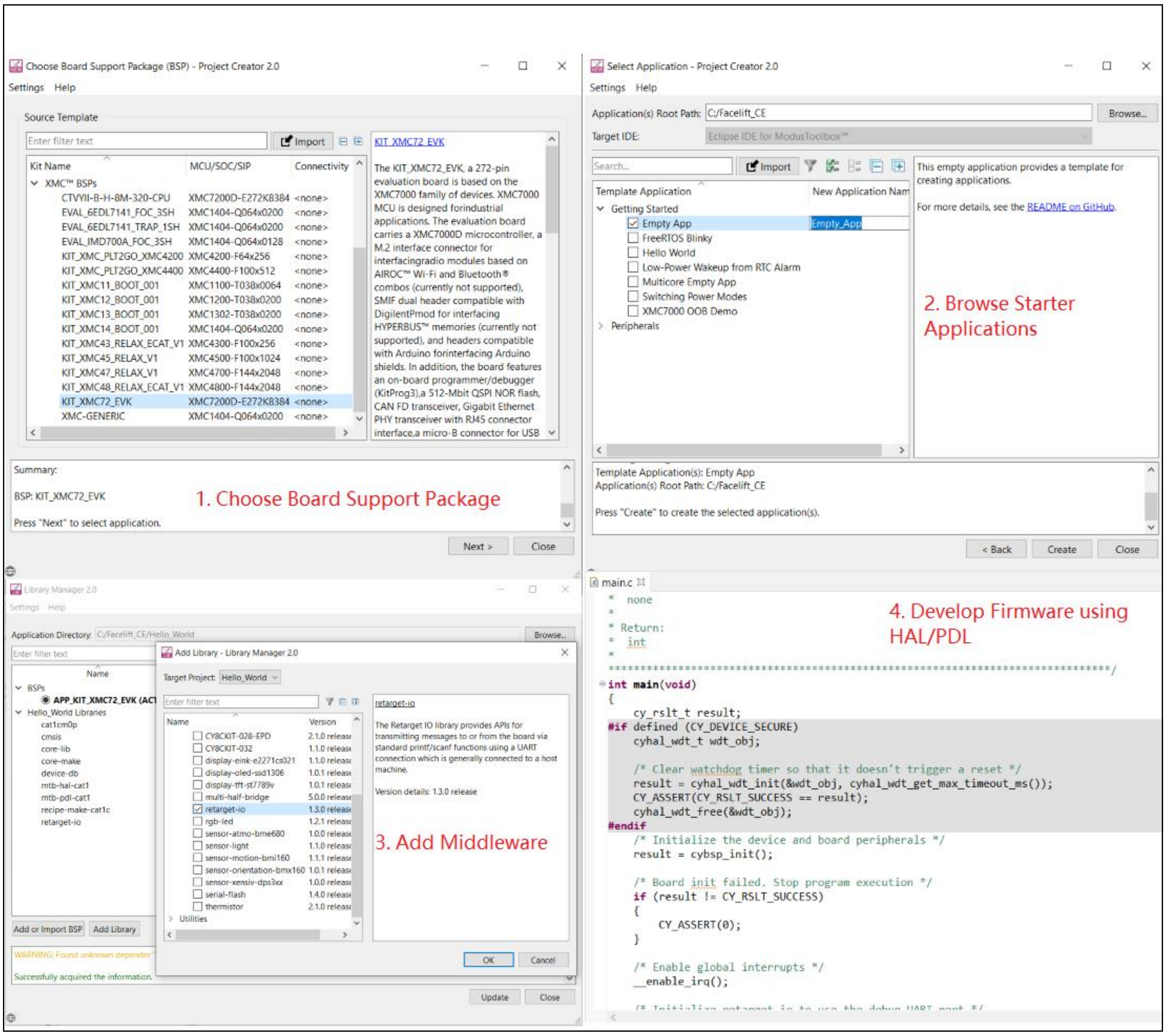
- All API elements needed to initialize, configure, and use a peripheral are available.
- The PDL is useful across various XMC7000 MCU devices, regardless of available peripherals.
- The PDL includes error checking to ensure that the targeted peripheral is present on the selected device.

This enables the code to maintain compatibility across members of the XMC7000 MCU device family as long as the peripherals are available. A device header file specifies the peripherals that are available for a device. If you write code that attempts to use an unsupported peripheral, you will get an error at compile time. Before writing code to use a peripheral, consult the datasheet for the particular device to confirm support for that peripheral.

As **Figure 3** shows, with the Eclipse IDE for ModusToolbox™ software, you can:

1. Choose a board support package (BSP).

2. Create a new application based on a list of starter applications, filtered by kit.
3. Add middleware.
4. Develop your application firmware using the HAL or PDL for XMC7000 MCU.


Choose Board Support Package (BSP) - Project Creator 2.0
Settings
Help
Source Template
Enter filter text
Import
Next >
Close
1. Choose Board Support Package
Select Application - Project Creator 2.0
Browse...
Search...
< Back
Create
2. Browse Starter Applications
Library Manager 2.0
Add Library - Library Manager 2.0
Add or Import BSP
Add Library
OK
Cancel
Update
3. Add Middleware
main.c
4. Develop Firmware using HAL/PDL

Kit Name	MCU/SOC/SIP	Connectivity
XMC™ BSPs		
CTVYII-B-H-8M-320-CPU	XMC7200D-E272K8384	<none>
EVAL_6EDL7141_FOC_3SH	XMC1404-Q064x0200	<none>
EVAL_6EDL7141_TRAP_1SH	XMC1404-Q064x0200	<none>
EVAL_IMD700A_FOC_3SH	XMC1404-Q064x0128	<none>
KIT_XMC_PLT2GO_XMC4200	XMC4200-F64x256	<none>
KIT_XMC_PLT2GO_XMC4400	XMC4400-F100x512	<none>
KIT_XMC11_BOOT_001	XMC1100-T038x0064	<none>
KIT_XMC12_BOOT_001	XMC1200-T038x0200	<none>
KIT_XMC13_BOOT_001	XMC1302-T038x0200	<none>
KIT_XMC14_BOOT_001	XMC1404-Q064x0200	<none>
KIT_XMC43_RELAX_ECAT_V1	XMC4300-F100x256	<none>
KIT_XMC45_RELAX_V1	XMC4500-F100x1024	<none>
KIT_XMC47_RELAX_V1	XMC4700-F144x2048	<none>
KIT_XMC48_RELAX_ECAT_V1	XMC4800-F144x2048	<none>
KIT_XMC72_EVK	XMC7200D-E272K8384	<none>
XMC-GENERIC	XMC1404-Q064x0200	<none>

KIT_XMC72_EVK

The KIT_XMC72_EVK, a 272-pin evaluation board is based on the XMC7000 family of devices. XMC7000 MCU is designed forindustrial applications. The evaluation board carries a XMC7000D microcontroller, a M.2 interface connector for interfacingradio modules based on AIROC™ Wi-Fi and Bluetooth® combos (currently not supported), SMIF dual header compatible with DigilentPmod for interfacing HYPERBUS™ memories (currently not supported), and headers compatible with Arduino forinterfacing Arduino shields. In addition, the board features an on-board programmer/debugger (KitProg3),a 512-Mbit QSPI NOR flash, CAN FD transceiver, Gigabit Ethernet PHY transceiver with RJ45 connector interface,a micro-B connector for USB

Summary:

BSP: KIT_XMC72_EVK

Press "Next" to select application.

Application(s) Root Path: C:/Facelift_CE

Target IDE: Eclipse IDE for ModusToolbox™

Template Application	New Application Nam
Getting Started	
✓ Empty App	Empty_App
FreeRTOS Blinky	
Hello World	
Low-Power Wakeup from RTC Alarm	
Multicore Empty App	
Switching Power Modes	
XMC7000 OOB Demo	
Peripherals	

This empty application provides a template for creating applications.

For more details, see the README on GitHub.

Template Application(s): Empty App
Application(s) Root Path: C:/Facelift_CE

Press "Create" to create the selected application(s).

Application Directory: C:/Facelift_CE/Hello_World

Name
- BSPs
  - APP_KIT_XMC72_EVK (ACT
- Hello_World Libraries
  - cat1cm0p
  - cmsis
  - core-lib
  - core-make
  - device-db
  - mtb-hal-cat1
  - mtb-pdl-cat1
  - recipe-make-cat1c
  - retarget-io

WARNING: Found unknown depender

Successfully acquired the information.

Target Project: Hello_World

Name	Version
CY8CKIT-028-EPD	2.1.0 release
CY8CKIT-032	1.1.0 release
display-eink-e2271cs021	1.1.0 release
display-oled-ssd1306	1.0.1 release
display-tft-st7789v	1.0.1 release
multi-half-bridge	5.0.0 release
✓ retarget-io	1.3.0 release
rgb-led	1.2.1 release
sensor-atmo-bme680	1.0.0 release
sensor-light	1.1.0 release
sensor-motion-bmi160	1.1.1 release
sensor-orientation-bmx160	1.0.1 release
sensor-xensiv-dps3xx	1.0.0 release
serial-flash	1.4.0 release
thermistor	2.1.0 release
Utilities	

retarget-io

The Retarget IO library provides APIs for transmitting messages to or from the board via standard printf/scanf functions using a UART connection which is generally connected to a host machine.

Version details: 1.3.0 release

```
 *  none
 *
 * Return:
 *  int
 *
 *******************************************************************************/
int main(void)
{
    cy_rslt_t result;
#if defined (CY_DEVICE_SECURE)
    cyhal_wdt_t wdt_obj;

    /* Clear watchdog timer so that it doesn't trigger a reset */
    result = cyhal_wdt_init(&wdt_obj, cyhal_wdt_get_max_timeout_ms());
    CY_ASSERT(CY_RSLT_SUCCESS == result);
    cyhal_wdt_free(&wdt_obj);
#endif
    /* 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();

    /* Initialize retarget-io to use the debug UART port */
```

Figure 3 Eclipse IDE for ModusToolbox™ resources and middleware

2.3 Support for other IDEs

You can develop firmware for XMC7000 MCUs using your favorite IDE such as **IAR Embedded Workbench or Visual Studio Code**.

ModusToolbox™ configurators are standalone tools that can be used to set up and configure XMC7000 MCU resources and other middleware components without using the Eclipse IDE. The device configurator and middleware configurators use the *design.x* files within the application workspace. You can then point to the generated source code and continue developing firmware in your IDE.

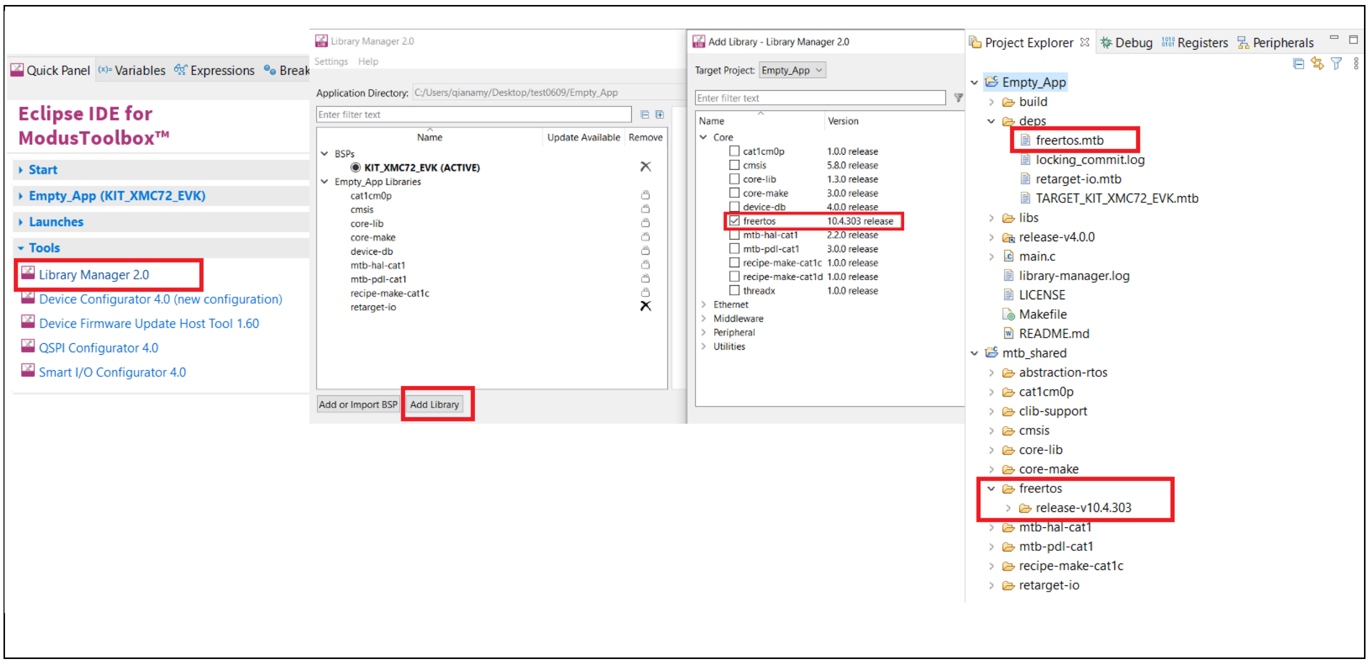
If there is a change in the device configuration, edit the *design.x* files using the configurators and regenerate the code for the target IDE. It is recommended that you generate resource configurations using the configuration tools provided with ModusToolbox™ software.

See **AN225588 – Using ModusToolbox™ software with a third-party IDE** for details.

2.4 FreeRTOS support with ModusToolbox™

Adding native FreeRTOS support to a ModusToolbox™ application project is like adding any library or middleware. You can import the FreeRTOS middleware into your application by using the Library Manager. Select the application project and click the **Library Manager** link in the **Quick Panel**. Select **freertos** from the **Add Library** > **Core** dialog, as **Figure 4** shows.

The *.mtb* file pointing to the FreeRTOS middleware is added to the application project. The middleware content is also downloaded and placed inside the corresponding folder called **freertos**. To continue working with FreeRTOS, follow the steps in the Quick Start section of **FreeRTOS documentation**.



Name	Version
Core	
☐ cat1cm0p	1.0.0 release
☐ cmsis	5.8.0 release
☐ core-lib	1.3.0 release
☐ core-make	3.0.0 release
☐ device-db	4.0.0 release
✓ freertos	10.4.303 release
☐ mtb-hal-cat1	2.2.0 release
☐ mtb-pdl-cat1	3.0.0 release
☐ recipe-make-cat1c	1.0.0 release
☐ recipe-make-cat1d	1.0.0 release
☐ threadx	1.0.0 release

Figure 4 Import FreeRTOS middleware in ModusToolbox™ application

2.5 Programming/debugging

All XMC7000 kits have a KitProg3 onboard programmer/debugger. It supports Cortex® Microcontroller Software Interface Standard - Debug Access Port (CMSIS-DAP). See the **KitProg3 user guide** for details.

The Eclipse IDE requires KitProg3 and uses the **OpenOCD** protocol for debugging XMC7000 MCU applications. It also supports GDB debugging using industry standard probes like the **Segger J-Link**.

ModusToolbox™ includes the **fw-loader** command-line tool to update and switch the KitProg firmware from KitProg2 to KitProg3. Refer to the **XMC7000 Programming/Debugging - KitProg Firmware Loader** section in the Eclipse IDE for ModusToolbox™ user guide for more details.

For more information on debugging firmware on XMC™ devices with ModusToolbox™, refer to the “Program and Debug” section in the Eclipse IDE for ModusToolbox™ user guide.

2.6 XMC7000 MCU development kits

Table 2 Development kits

Product line	**Development kits**
Performance	**XMC 7200 Evaluation Kit (KIT_XMC72_EVK)**

For the complete list of kits for the XMC7000 MCU along with the shield modules, see the **Microcontroller (MCUs) kits** page.

3 Device features

XMC7000 MCU product lines have extensive feature sets as shown in **Figure 5**. The following is a list of its major features. For more information, see the device **datasheet**, the **technical reference manual (TRM)**, and the section on **References**.

- **CPU Subsystem**
  - One or two 250-MHz/350-MHz Arm® Cortex®-M7 and 100-MHz Arm® Cortex®-M0+
  - Inter-processor communication supported in hardware
  - Three DMA controllers
- **Integrated memories**
  - Up to 8384 KB of code flash with an additional 256 KB work flash
  - Up to 1024 KB of SRAM selectable retention granularity
- **Cryptography engine**
  - Supports Enhanced Secure Hardware Extension (eSHE) and Hardware Security Module (HSM)
  - Secure boot and authentication
  - AES: 128-bit blocks, 128-/192-/256-bit keys
  - 3DES: 64-bit blocks, 64-bit key
  - Vector unit supporting asymmetric key cryptography such as Rivest-Shamir-Adleman (RSA) and Elliptic Curve (ECC)
  - SHA-1/2/3: SHA-512, SHA-256, SHA-160 with variable length input data
  - CRC: supports CCITT CRC16 and IEEE-802.3 CRC32
  - True random number generator (TRNG) and pseudo random number generator (PRNG)
  - Galois/Counter Mode (GCM)
- **Safety for application**
  - Memory Protection Unit (MPU)
  - Shared Memory Protection Unit (SMPU)
  - Peripheral Protection Unit (PPU)
  - Watchdog Timer (WDT)
  - Multi-counter Watchdog Timer (MCWDT)
  - Low-voltage Detector (LVD)
  - Brown-out Detection (BOD)
  - Overvoltage Detection (OVD)
  - Clock Supervisor (CSV)
  - Hardware error correction (SECDED ECC) on all safety-critical memories (SRAM, flash, TCM)
- **Low-power 2.7-V to 5.5-V operation**
  - Low-power Active, Sleep, Low-power Sleep, DeepSleep, and Hibernate modes for fine-grained power management
  - Configurable options for robust BOD
- **Wakeup**
  - Up to two pins to wake from Hibernate mode
  - Up to 220 GPIO pins to wake from Sleep modes
  - Event Generator, SCB, Watchdog Timer, RTC alarms to wake from DeepSleep modes

- **Clocks**
  - Internal Main Oscillator (IMO)
  - Internal Low-Speed Oscillator (ILO)
  - External Crystal Oscillator (ECO)
  - Watch Crystal Oscillator (WCO)
  - Phase-Locked Loop (PLL)
  - Frequency-Locked Loop (FLL)
- **Communication interfaces**
  - Up to 10 CAN FD channels
  - Up to 11 runtime-reconfigurable SCB (serial communication block) channels, each configurable as I2C, SPI, or UART
  - Up to two 10/100/1000 Mbps Ethernet MAC interfaces conforming to IEEE-802.3az
- **External memory interface**
  - One SPI (Single, Dual, Quad, or Octal) or HYPERBUS™ interface
  - On-the-fly encryption and decryption
  - Execute-In-Place (XIP) from external memory
- **SDHC interface**
  - One Secure Digital High Capacity (SDHC) interface supporting embedded MultiMediaCard (eMMC), Secure Digital (SD), or SDIO (Secure Digital Input Output)
  - Data rates up to SD High-Speed 50 MHz, or eMMC 52 MHz DDR
- **Audio interface**
  - Three Inter-IC Sound (I2S) Interfaces (based on the NXP I2S bus specification) for connecting digital audio devices
  - I2S, left justified, or Time Division Multiplexed (TDM) audio formats
  - Independent transmit or receive operation, each in master or slave mode
- **Timers**
  - Up to 102 blocks of 16-bit and 16 blocks of 32-bit Timer/Counter Pulse-Width Modulator (TCPWM)
  - Up to 16 Event Generation (EVTGEN) timers supporting cyclic wakeup from DeepSleep
- **Real time clock (RTC)**
  - Year/Month/Date, Day-of-week, Hour:Minute:Second fields
  - 12- and 24-hour formats
  - Automatic leap-year correction
- **I/O**
  - Clock supervisor (CSV)
  - Three I/O types: GPIO_STD/ GPIO_ENH/ HSIO_STD
- **Smart I/O**
  - Up to five smart I/O blocks, which can perform Boolean operations on signals going to and from I/Os
  - Up to 36 I/Os (GPIO_STD) supported
- **I/O subsystem**
  - Up to 220 GPIOs with programmable drive modes, drive strength, slew rates
  - Two ports with smart I/O that can implement Boolean operations

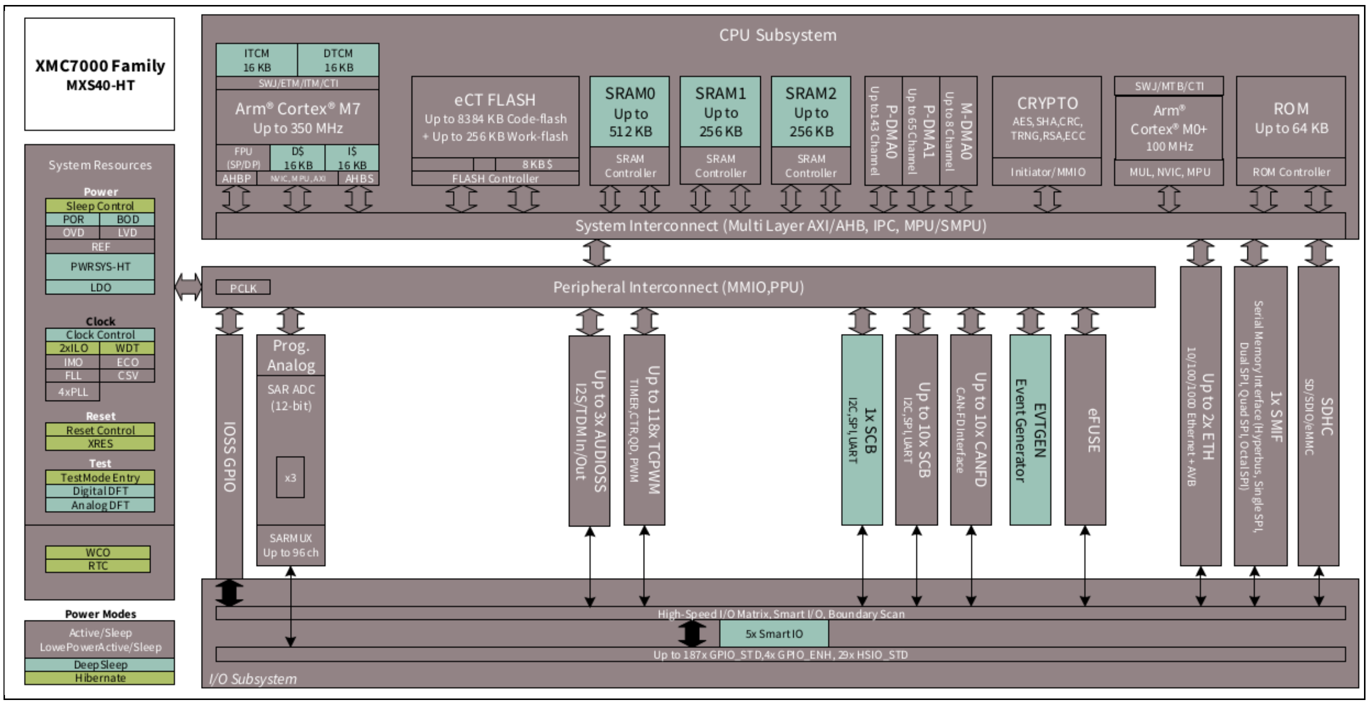


Figure 5 **XMC7000 MCU block diagram**

- **Programmable analog**
  - Three SAR A/D converters with up to 99 external channels (96 I/Os + 3 I/Os for motor control)
  - Each ADC supports 12-bit resolution and sampling rates of up to 1 Msps
  - Each ADC also supports six internal analog inputs
  - Each ADC supports addressing of external multiplexers
  - Each ADC has a sequencer supporting autonomous scanning of configured channels
  - Synchronized sampling of all ADCs for motor-sense applications

4 My first XMC7000 MCU design using Eclipse IDE for ModusToolbox™ software

This section does the following:

- Demonstrate how to build a simple XMC7000 MCU-based design and program it on to the development kit
- Make it easy to learn XMC7000 MCU design techniques and how to use the Eclipse IDE for ModusToolbox™ software

4.1 Prerequisites

Before you get started, make sure that you have the appropriate development kit for your XMC7000 MCU product line, and have installed the required software. You also need internet access to the GitHub repositories during project creation.

4.1.1 Hardware

- The design is developed for **XMC7200 Evaluation Kit**. However, you can build the projects for other development kits. See the **Using these instructions** section.

4.1.2 Software

- **ModusToolbox™** 3.0 or above

After installing the software, refer to the **ModusToolbox™ user guide** to get an overview of the software.

4.2 Using these instructions

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

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

This design is developed for the **XMC7200 Evaluation Kit**. You can use other supported kits to test this example by selecting the appropriate kit while creating the application.

4.3 About the design

This design uses the CM7 CPU of the XMC7000 MCU to execute two tasks: UART communication and LED control.

After device reset, the CM7 CPU 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.

4.4 Part 1: Create a new application

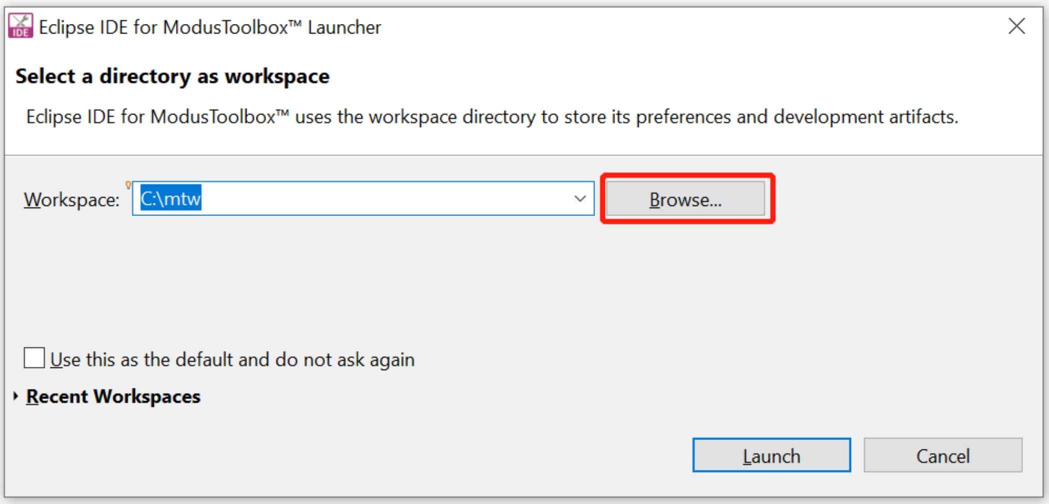
This section takes you on a step-by-step guided tour of the new application process. It uses the '**Empty App**' starter application and guides you through the design development stages, and programming.

If you are familiar with developing projects with ModusToolbox™, 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 use the code example as a reference while following the instructions.

Launch Eclipse IDE for ModusToolbox™ to get started. Please note that Eclipse IDE for ModusToolbox™ software needs access to the internet to successfully clone the starter application onto your machine.

1. **Select a new workspace.**
   At launch, Eclipse IDE for ModusToolbox™ presents 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 **Figure 6** shows. Alternatively, you can type in a directory name to be used as the workspace directory along with the complete path, and Eclipse IDE will create the directory for you.



Eclipse IDE for ModusToolbox™ Launcher

Select a directory as workspace

Eclipse IDE for ModusToolbox™ uses the workspace directory to store its preferences and development artifacts.

Workspace: C:\mtw

Use this as the default and do not ask again

Figure 6 **Select a directory as the workspace**

2. **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 7** shows.
   The Eclipse IDE for ModusToolbox™ Application window appears.

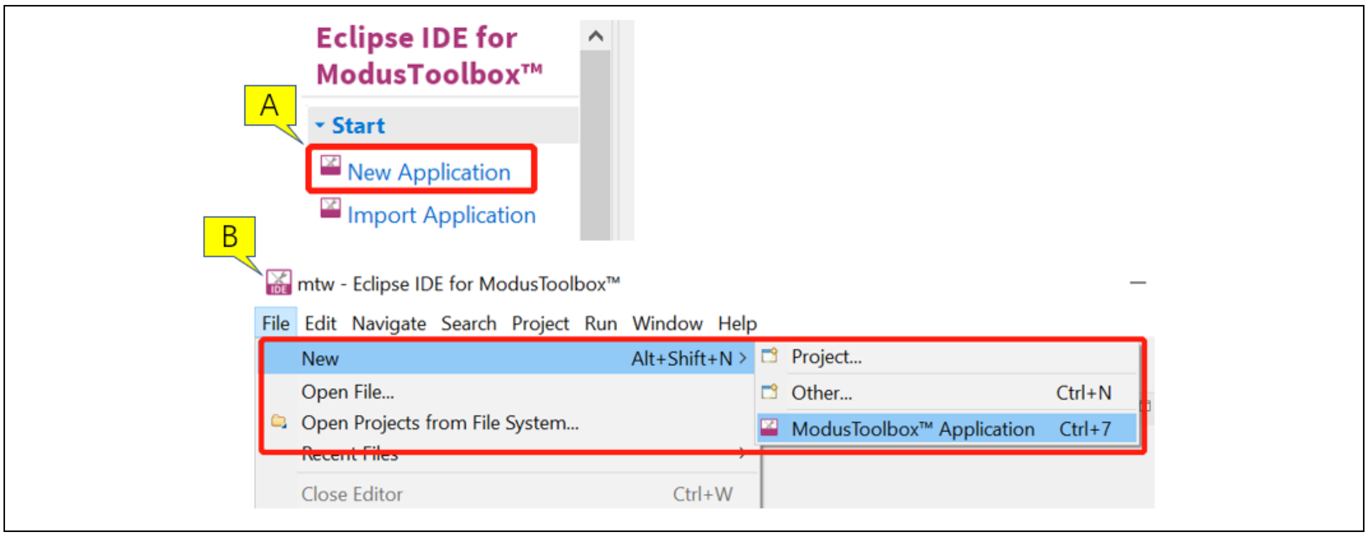
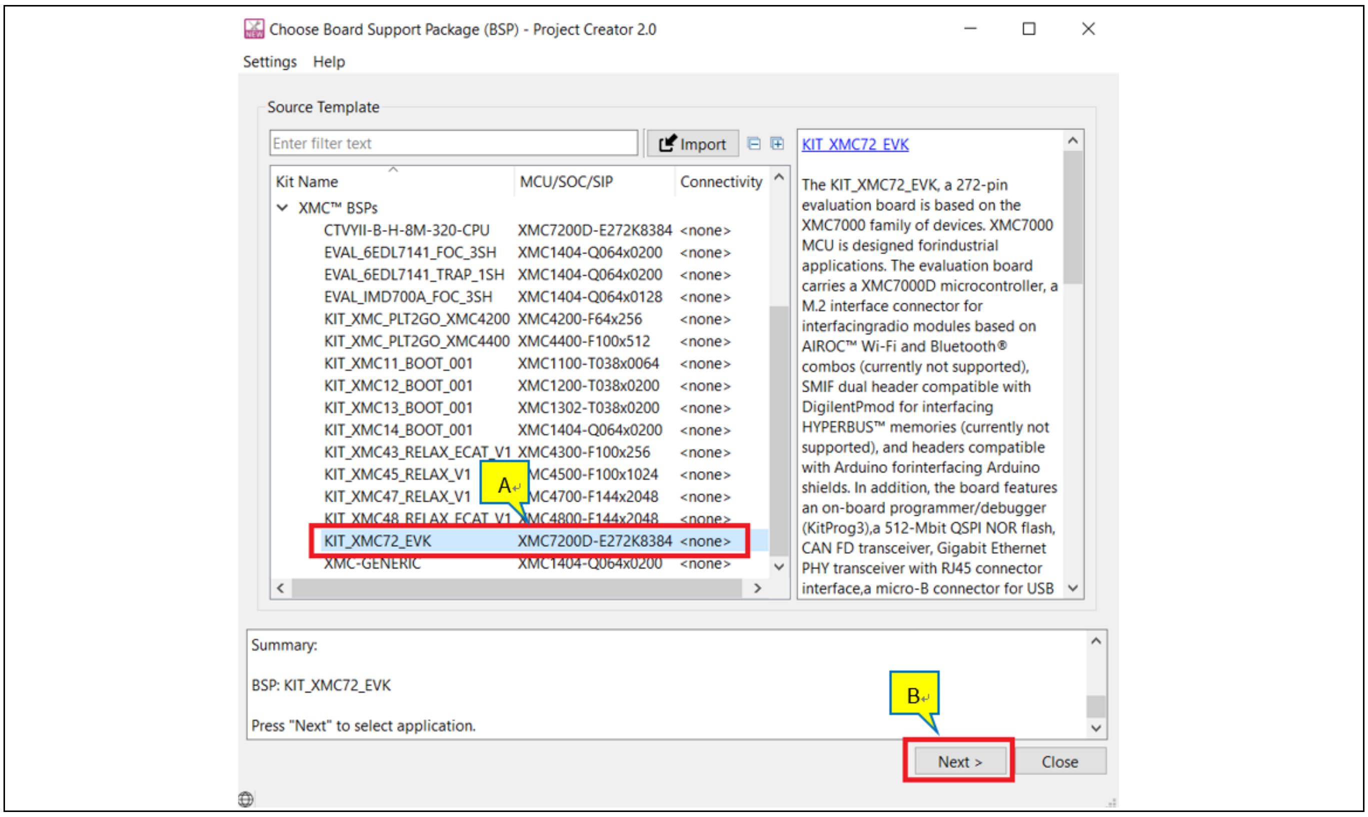

Eclipse IDE for ModusToolbox™
A
Start
New Application
Import Application
B
mtw - Eclipse IDE for ModusToolbox™
File Edit Navigate Search Project Run Window Help
New Alt+Shift+N
Open File...
Open Projects from File System...
Recent Files
Close Editor Ctrl+W
Project...
Other... Ctrl+N
ModusToolbox™ Application Ctrl+7

Figure 7 **Create a New ModusToolbox™ Application**

3. **Select a target XMC7200 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 follow use **KIT_XMC72_EVK**. See **Figure 8** for help with this step.
   b) Click **Next**.



Kit Name	MCU/SOC/SIP	Connectivity
XMC™ BSPs		
CTVYII-B-H-8M-320-CPU	XMC7200D-E272K8384	<none>
EVAL_6EDL7141_FOC_3SH	XMC1404-Q064x0200	<none>
EVAL_6EDL7141_TRAP_1SH	XMC1404-Q064x0200	<none>
EVAL_IMD700A_FOC_3SH	XMC1404-Q064x0128	<none>
KIT_XMC_PLT2GO_XMC4200	XMC4200-F64x256	<none>
KIT_XMC_PLT2GO_XMC4400	XMC4400-F100x512	<none>
KIT_XMC11_BOOT_001	XMC1100-T038x0064	<none>
KIT_XMC12_BOOT_001	XMC1200-T038x0200	<none>
KIT_XMC13_BOOT_001	XMC1302-T038x0200	<none>
KIT_XMC14_BOOT_001	XMC1404-Q064x0200	<none>
KIT_XMC43_RELAX_ECAT_V1	XMC4300-F100x256	<none>
KIT_XMC45_RELAX_V1	XMC4500-F100x1024	<none>
KIT_XMC47_RELAX_V1	XMC4700-F144x2048	<none>
KIT_XMC48_RELAX_ECAT_V1	XMC4800-F144x2048	<none>
KIT_XMC72_EVK	XMC7200D-E272K8384	<none>
XMC-GENERIC	XMC1404-Q064x0200	<none>

KIT_XMC72_EVK

The KIT_XMC72_EVK, a 272-pin evaluation board is based on the XMC7000 family of devices. XMC7000 MCU is designed forindustrial applications. The evaluation board carries a XMC7000D microcontroller, a M.2 interface connector for interfacingradio modules based on AIROC™ Wi-Fi and Bluetooth® combos (currently not supported), SMIF dual header compatible with DigilentPmod for interfacing HYPERBUS™ memories (currently not supported), and headers compatible with Arduino forinterfacing Arduino shields. In addition, the board features an on-board programmer/debugger (KitProg3),a 512-Mbit QSPI NOR flash, CAN FD transceiver, Gigabit Ethernet PHY transceiver with RJ45 connector interface,a micro-B connector for USB

Summary:

BSP: KIT_XMC72_EVK

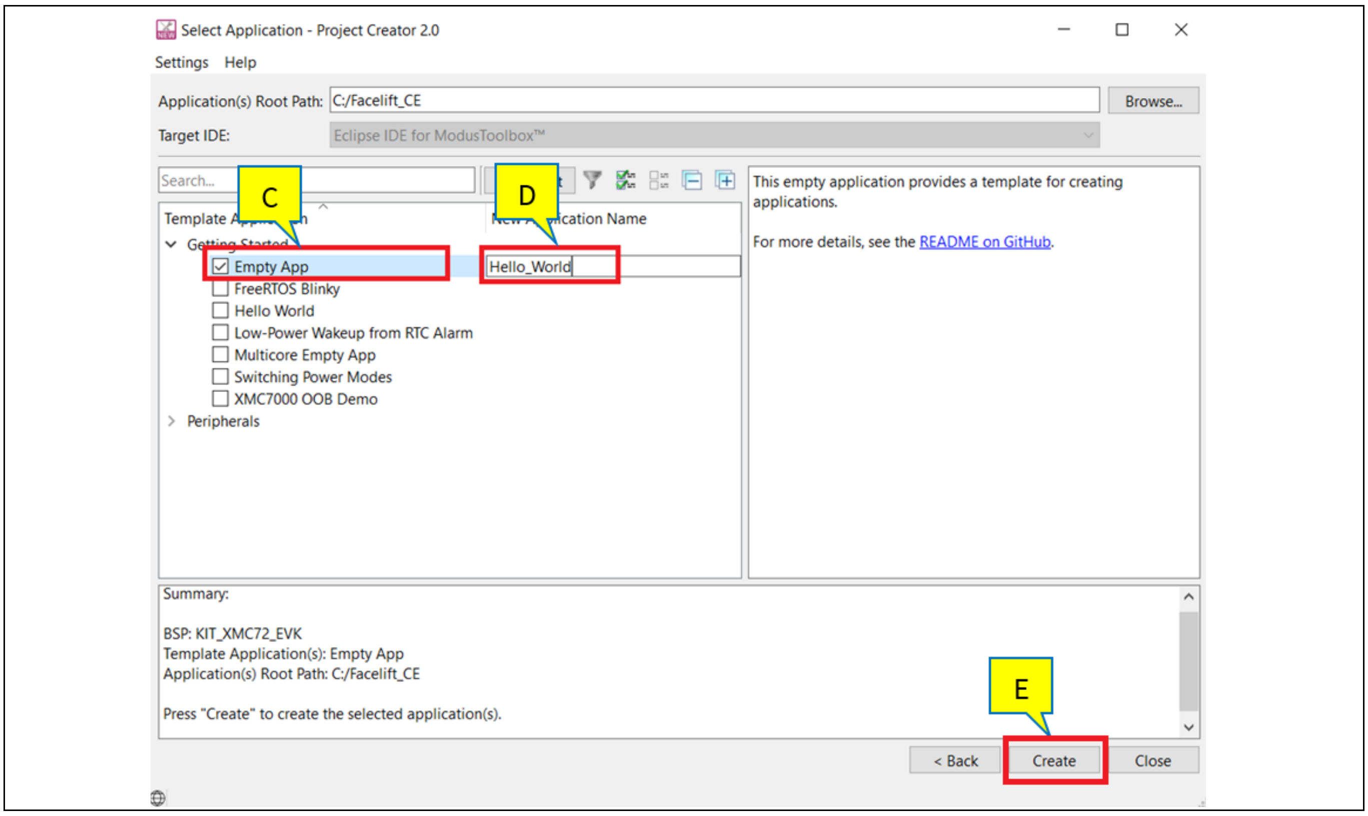
Press "Next" to select application.

Figure 8 **Choose target hardware**

   c) In the **Starter Application** dialog, select **Empty App** starter application, as **Figure 9** shows.

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.

e) Click **Create** to create the application, as **Figure 9** shows, wait for the Project Creator to automatically close once the project is successfully created.


Select Application - Project Creator 2.0
Settings Help
Application(s) Root Path: C:/Facelift_CE
Browse...
Target IDE: Eclipse IDE for ModusToolbox™
Search...
C
D
Template Application
New Application Name
Getting Started
Empty App
Hello_World
FreeRTOS Blinky
Hello World
Low-Power Wakeup from RTC Alarm
Multicore Empty App
Switching Power Modes
XMC7000 OOB Demo
Peripherals
This empty application provides a template for creating applications.
For more details, see the README on GitHub.
E
< Back
Create
Close

Summary:

BSP: KIT_XMC72_EVK
Template Application(s): Empty App
Application(s) Root Path: C:/Facelift_CE

Press "Create" to create the selected application(s).

Figure 9 **Choose starter application**

You have successfully created a new ModusToolbox™ application for a XMC7000 MCU.

The BSP uses XMC7200D-E272K8384 as the default device that is mounted on the **XMC7200 Evaluation Kit**.

If you are using custom hardware based on XMC7000 MCU, or a different XMC7000 MCU part number, see the **Creating your Own BSP** section in the **ModusToolbox™ user guide**. The guide is also available under *ide_3.0>docs* folder of the ModusToolbox™ installation directory.

4.5 Part 2: View and modify the design

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

XMC7000 MCU consists of three cores: one CM0+ core and two CM7 cores. This application note shows the firmware development using the CM7 core with ModusToolbox™ software.

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

a) An application project contains a Makefile which is typically at the root folder. It has instructions on how to recreate the project. This file also 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, and each dependent project usually resides within its own folder within the application folder and contains its own Makefile.

b) The *bsps* folder contains all the configuration files generated by the device and peripheral configurators are included in the *GeneratedSource* folder of the BSP and are prefixed with *cycfg_*. These files contain the design configuration as defined by the BSP. You can view and modify the design configuration by clicking the Device Configurator link in the Quick Panel. However, note that if you upgrade the BSP library to a newer version, the manual edits done to the *design.x* files are lost. You can also create custom Device Configurator files for an application and override the ones provided by the BSP. See the "Modifying the BSP configuration for a single application" section in the **ModusToolbox™ user guide** for more details.

The BSP folder also contains the linker scripts and the startup code for the XMC7000 MCU device used on the board.

c) The *build* folder contains all the artifacts resulting from the make build of the project. The output files are organized by target BSPs.

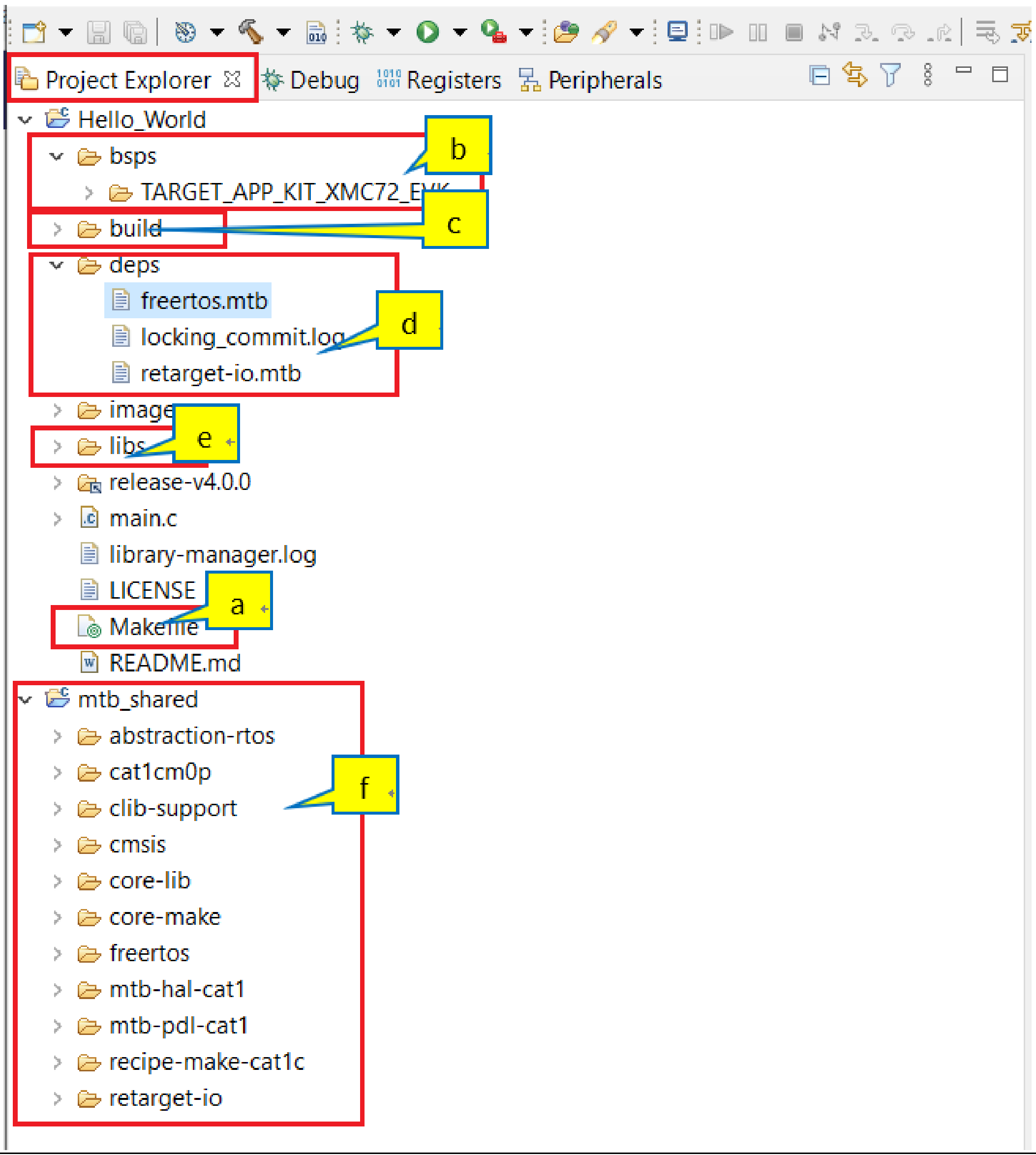


Figure 10 Project Explorer view

d) The *deps* folder contains *.mtb* files, which provide the location from which ModusToolbox™ software pulls the BSP/library that is directly referenced by the application. These files typically contain the GitHub location of the entire library. The *.mtb* files also contains 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.

For example, the *TARGET_KIT_XMC72_EVK.mtb* file points to **http://git-ore.aus.cypress.com/repo-staging/TARGET_KIT_XMC72_EVK#latest-v1.X#$$ASSET_REPO$$/TARGET_KIT_XMC72_EVK/latest-**

v1.X. The *latest-v1.X* tag in the link denotes the specific release of the BSP. The variable $$ASSET_REPO$$ points to the root of the shared location. If the library has to be local to the application instead of shared, use $$LOCAL$$ instead of $$ASSET_REPO$$.

Similarly, *retarget-io.mtb* points to **https://github.com/infineon/retarget-io#latest-v1.X#$$ASSET_REPO$$/retarget-io/latest-v1.X**.

e) The *libs* folder also contains *.mtb* files, these point to libraries that are included indirectly as a dependency of a BSP or another library. For each indirect dependency, the Library Manager places an *.mtb* file in this folder. These files have been populated based on the targets available in *deps* folder.

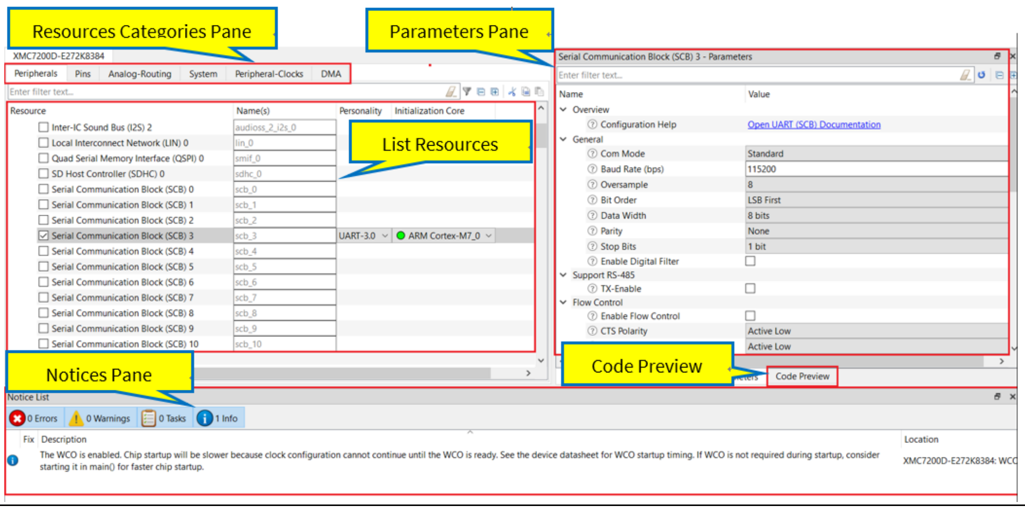
For example, using the BSP lib file, *TARGET_KIT_XMC72_EVK.mtb* populates the *libs* folder with the following *.mtb* files: ***core-lib.mtb***, ***core-make.mtb***, ***mtb-hal-cat1.mtb***, ***mtb-pdl-cat1.mtb***, ***recipe-make-cat1c.mtb***.

The *libs* folder contains the *mtb.mk* file, which stores the relative paths of the all the libraries required by the application. The build system uses this file to find all the libraries required by the application.

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

The *mtb_shared* folder is shared between different applications that use the same versions of the BSP/library.

Of interest are the configuration files that are in the *COMPONENT_BSP_x* folder. Click on the **Device Configurator** link in the **Quick Panel**. **Figure 11** shows the resulting window called the **Device Configurator** window. You can also double-click open the other *design.x* files to open them in their respective configurators or click the corresponding links in the **Quick Panel**.



Resource	Name(s)	Personality	Initialization Core
☐ Inter-IC Sound Bus (I2S) 2	audioss_2_i2s_0		
☐ Local Interconnect Network (LIN) 0	lin_0		
☐ Quad Serial Memory Interface (QSPI) 0	smif_0		
☐ SD Host Controller (SDHC) 0	sdhc_0		
☐ Serial Communication Block (SCB) 0	scb_0		
☐ Serial Communication Block (SCB) 1	scb_1		
☐ Serial Communication Block (SCB) 2	scb_2		
✓ Serial Communication Block (SCB) 3	scb_3	UART-3.0	● ARM Cortex-M7_0
☐ Serial Communication Block (SCB) 4	scb_4		
☐ Serial Communication Block (SCB) 5	scb_5		
☐ Serial Communication Block (SCB) 6	scb_6		
☐ Serial Communication Block (SCB) 7	scb_7		
☐ Serial Communication Block (SCB) 8	scb_8		
☐ Serial Communication Block (SCB) 9	scb_9		
☐ Serial Communication Block (SCB) 10	scb_10		

Serial Communication Block (SCB) 3 - Parameters

Name	Value
Overview	
Configuration Help	Open UART (SCB) Documentation
General	
Com Mode	Standard
Baud Rate (bps)	115200
Oversample	8
Bit Order	LSB First
Data Width	8 bits
Parity	None
Stop Bits	1 bit
Enable Digital Filter	☐
Support RS-485	
TX-Enable	☐
Flow Control	
Enable Flow Control	☐
CTS Polarity	Active Low
	Active Low

Notice List

0 Errors | 0 Warnings | 0 Tasks | 1 Info

Fix	Description	Location
	The WCO is enabled. Chip startup will be slower because clock configuration cannot continue until the WCO is ready. See the device datasheet for WCO startup timing. If WCO is not required during startup, consider starting it in main() for faster chip startup.	XMC7200D-E272K8384: WCO

Figure 11 design.modus overview

The **Device Configurator** window provides a **Resources Categories** pane. 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 a **EZI2C**, **I2C**, **SPI**, or **UART** personalities. The **Alias** 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. Any errors, warnings, and information messages arising out of the configuration are displayed in the **Notices** pane.

The application project contains relevant files that help you create an application for the CM7 CPU (*main.c*).

At this point in the development process, we are ready to add the required middleware to the design. The only middleware required for the Hello World application is the **retarget-io** library.

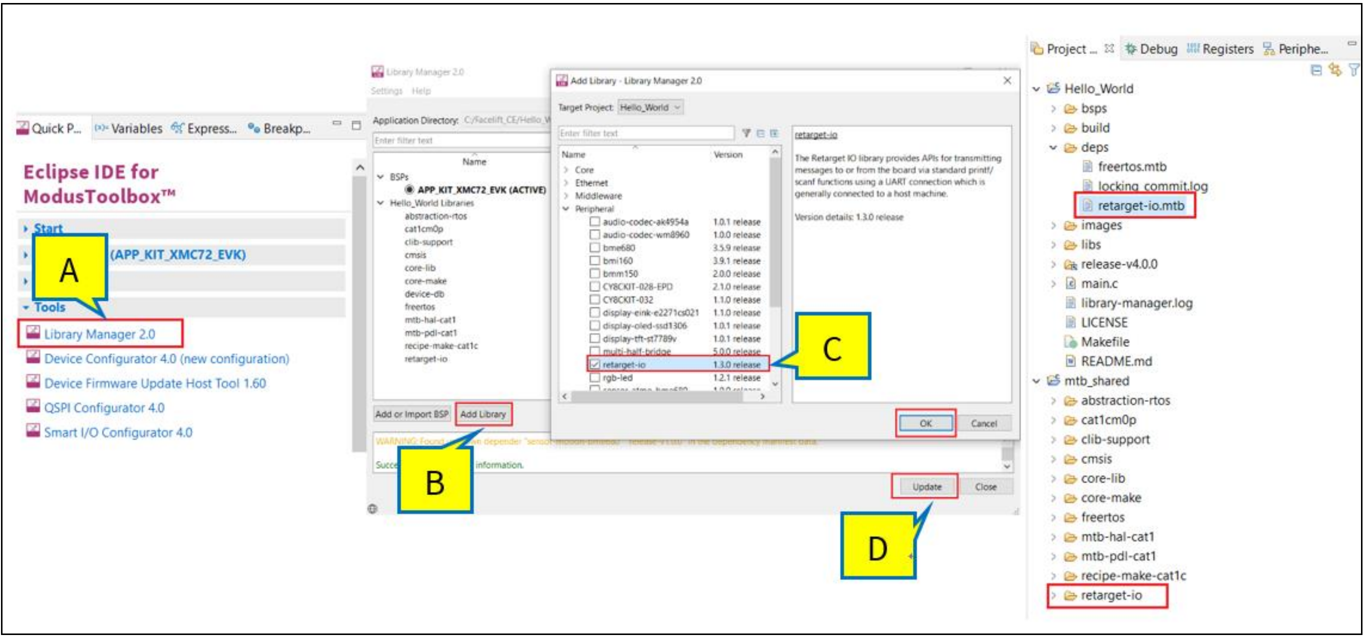


Figure 12 Add the retarget-io middleware

1. **Add retarget-io middleware.**

   In this step, you will 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 *main.c* code.

   a) In the **Quick Panel**, click on the **Library Manager** link.

   b) In the subsequent dialog, select the Add **Library** tab.

   c) Under **Peripherals**, select and enable **retarget-io**, and click **OK**.

   d) Click **Update.**

   The files necessary to use the **retarget-io** middleware are added in the *mtb_shared/retarget_io* folder; the *.mtb* file is also added to the *deps* folder, as **Figure 12** shows.

2. **Configuration of UART, timer peripherals, pins and system clocks**

   The configuration of the debug UART peripheral, timer peripheral, pins and system clocks can be done directly in the code using the function APIs provided by BSP and HAL. See **Part 3: Write firmware**.

4.6 Part 3: 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 XMC7000 starter application, you can copy the respective source code to the *main.c* 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

We now examine the code in the *main.c* file of the application. **Figure 13** shows the firmware flowchart.

After reset, resource initialization for this example is performed by the CM7 CPU. It configures the system clocks, pins, clock to peripheral connections, and other platform resources.

After reset, 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 the USB-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 CM7 CPU toggles the LED state on the kit.

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

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

`cybsp_init()` - This BSP function sets up the HAL hardware manager and initializes all the system resources of the device including but not limited to the system clocks and power regulators.

`cy_retarget_io_init()` - This function from the **retarget-io** middleware uses the aliases set up for debug UART pins to configure the debug UART with a standard baud rate of 115200 and redirects the input/output stream to the debug UART.

`cyhal_gpio_init()` - This function from the GPIO HAL initializes the physical pin to drive the LED. The LED used is derived from the BSP definition.

`timer_init()` - This function wraps a set of timer HAL function calls to instantiate and configure a timer. It also sets up a callback for the timer interrupt.

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

Code Listing 1

```
#include "cyhal.h"
#include "cybsp.h"
#include "cy_retarget_io.h"


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

/* LED blink timer clock value in Hz  */
#define LED_BLINK_TIMER_CLOCK_HZ          (10000)

/* LED blink timer period value */
#define LED_BLINK_TIMER_PERIOD            (9999)


/*******************************************************************************
* Function Prototypes
*******************************************************************************/
void timer_init(void);
static void isr_timer(void *callback_arg, cyhal_timer_event_t event);


/*******************************************************************************
* Global Variables
*******************************************************************************/
bool timer_interrupt_flag = false;
bool led_blink_active_flag = true;
```

Code Listing 1

```
/* Variable for storing character read from terminal */
uint8_t uart_read_value;

/* Timer object used for blinking the LED */
cyhal_timer_t led_blink_timer;


/*******************************************************************************
* 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. 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();

    /* Initialize retarget-io to use the debug UART port */
    result = cy_retarget_io_init(CYBSP_DEBUG_UART_TX, CYBSP_DEBUG_UART_RX,
                                 CY_RETARGET_IO_BAUDRATE);

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

    /* 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);
    }
```

Code Listing 1

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

    printf("****************** "
           "HAL: 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 */
        if (cyhal_uart_getc(&cy_retarget_io_uart_obj, &uart_read_value, 1)
             == CY_RSLT_SUCCESS)
        {
            if (uart_read_value == '\r')
            {
                /* Pause LED blinking by stopping the timer */
                if (led_blink_active_flag)
                {
                    cyhal_timer_stop(&led_blink_timer);

                    printf("LED blinking paused \r\n");
                }
                else /* Resume LED blinking by starting the timer */
                {
                    cyhal_timer_start(&led_blink_timer);

                    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 */
            cyhal_gpio_toggle(CYBSP_USER_LED);
```

Code Listing 1

```
        }
    }
}


/*******************************************************************************
* 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
*
*******************************************************************************/
 void timer_init(void)
 {
    cy_rslt_t result;

    const cyhal_timer_cfg_t led_blink_timer_cfg =
    {
        .compare_value = 0,                 /* Timer compare value, not used */
        .period = LED_BLINK_TIMER_PERIOD,   /* Defines the timer period */
        .direction = CYHAL_TIMER_DIR_UP,    /* Timer counts up */
        .is_compare = false,                /* Don't use compare mode */
        .is_continuous = true,              /* Run timer indefinitely */
        .value = 0                          /* Initial value of counter */
    };

    /* 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 = cyhal_timer_init(&led_blink_timer, NC, NULL);

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

    /* Configure timer period and operation mode such as count direction,
       duration */
    cyhal_timer_configure(&led_blink_timer, &led_blink_timer_cfg);

    /* Set the frequency of timer's clock source */
    cyhal_timer_set_frequency(&led_blink_timer, LED_BLINK_TIMER_CLOCK_HZ);

    /* Assign the ISR to execute on timer interrupt */
    cyhal_timer_register_callback(&led_blink_timer, isr_timer, NULL);

    /* Set the event on which timer interrupt occurs and enable it */
    cyhal_timer_enable_event(&led_blink_timer, CYHAL_TIMER_IRQ_TERMINAL_COUNT,
                              7, true);

    /* Start the timer with the configured settings */
    cyhal_timer_start(&led_blink_timer);
```

Code Listing 1

```
  }


/*******************************************************************************
* 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
*
*******************************************************************************/
static void isr_timer(void *callback_arg, cyhal_timer_event_t event)
{
    (void) callback_arg;
    (void) event;

    /* Set the interrupt flag and process it from the main while(1) loop */
    timer_interrupt_flag = true;
}
```

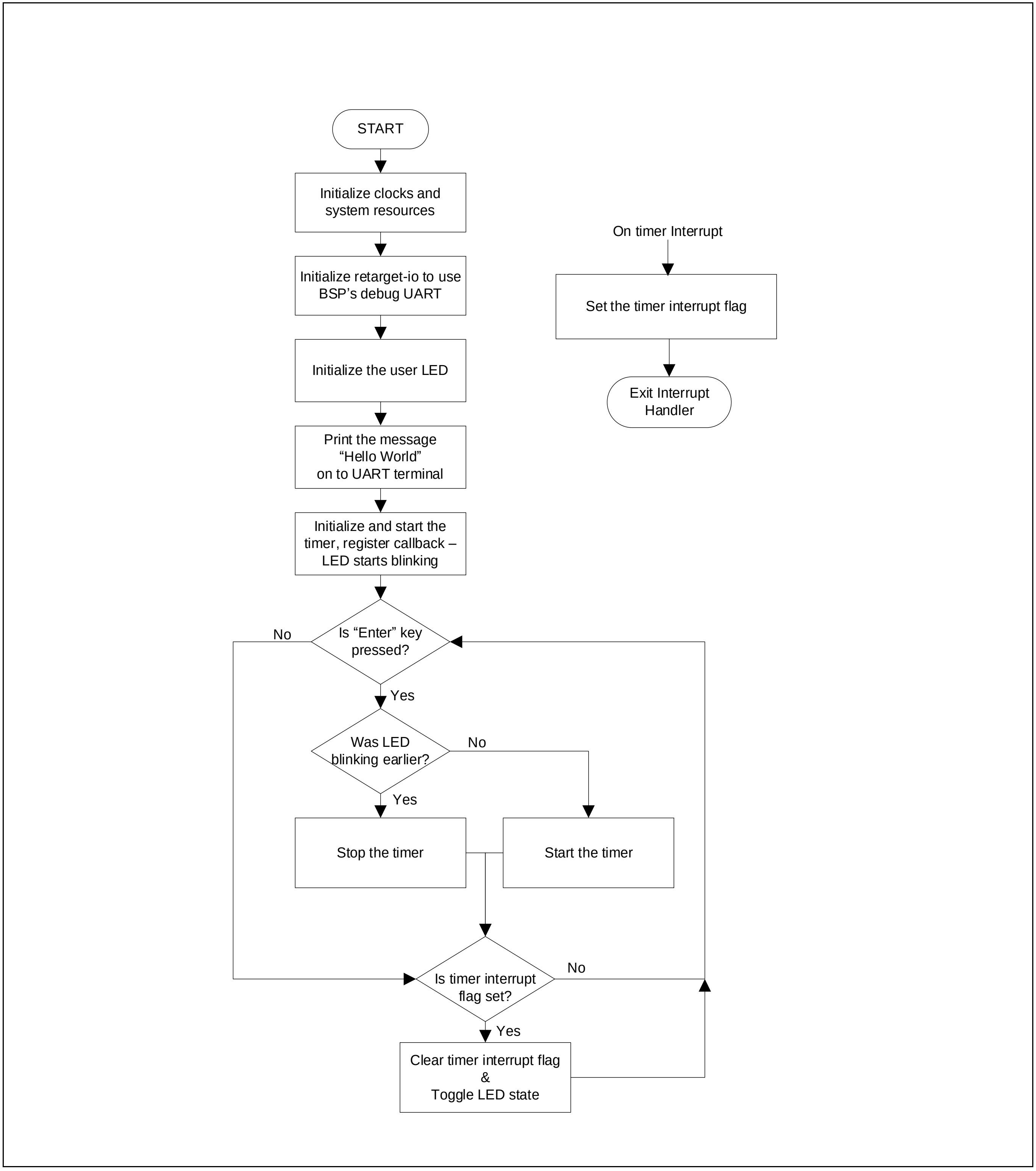

START
Initialize clocks and system resources
Initialize retarget-io to use BSP's debug UART
Initialize the user LED
Print the message "Hello World" on to UART terminal
Initialize and start the timer, register callback – LED starts blinking
Is "Enter" key pressed?
No
Yes
Was LED blinking earlier?
No
Yes
Stop the timer
Start the timer
Is timer interrupt flag set?
No
Yes
Clear timer interrupt flag & Toggle LED state
On timer Interrupt
Set the timer interrupt flag
Exit Interrupt Handler

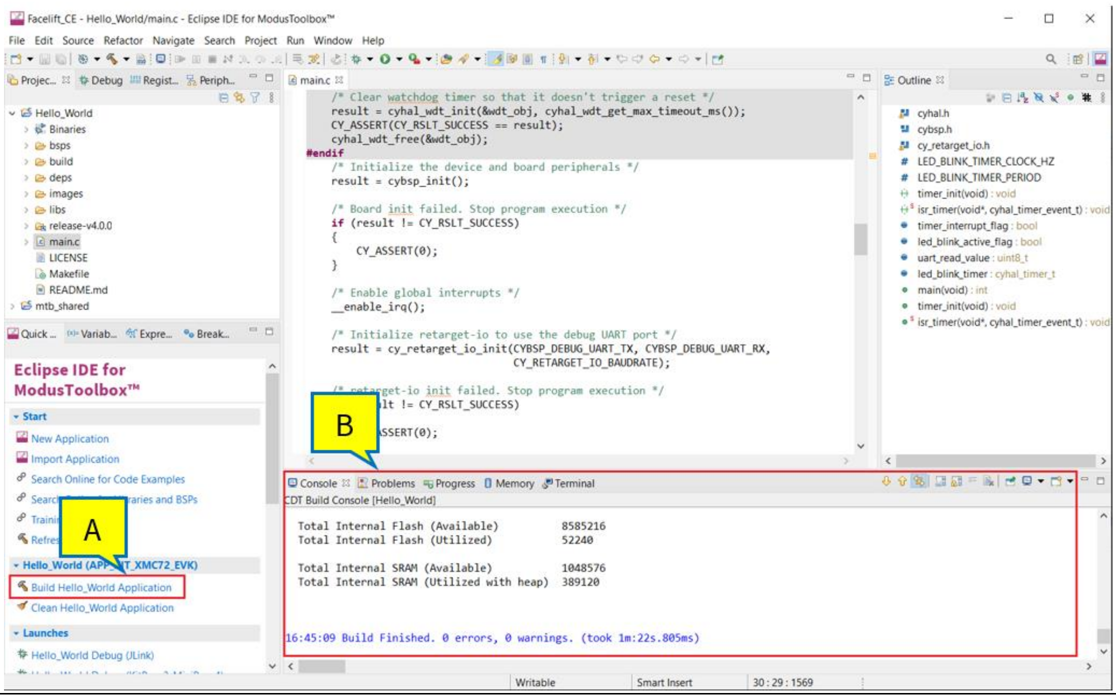
Figure 13 Firmware flowchart

This completes the summary of how the firmware works in the code example. Feel free to explore the source files for a deeper understanding.

4.7 Part 4: Build the application

This section shows how to build the application.

1. **Build the application**.
   a) Select the application project in the Project Explorer window and click on the **Build <name> Application** shortcut under the **<name>** group in the Quick Panel. It selects the **Debug** build configuration and compiles/links all projects that constitute the application.
   b) The **Console** view lists the results of the build operation, as **Figure 14** shows.


Facelift_CE - Hello_World/main.c - Eclipse IDE for ModusToolbox™
File Edit Source Refactor Navigate Search Project Run Window Help
Hello_World
Binaries
bsps
build
deps
images
libs
release-v4.0.0
main.c
LICENSE
Makefile
README.md
mtb_shared
Eclipse IDE for ModusToolbox™
Start
New Application
Import Application
Search Online for Code Examples
Hello_World (APP_KIT_XMC72_EVK)
Build Hello_World Application
Clean Hello_World Application
Launches
Hello_World Debug (JLink)
A
B
Outline
cyhal.h
cybsp.h
cy_retarget_io.h
LED_BLINK_TIMER_CLOCK_HZ
LED_BLINK_TIMER_PERIOD
Console
Problems
Progress
Memory
Terminal
Writable
Smart Insert
30 : 29 : 1569

```
    /* Clear watchdog timer so that it doesn't trigger a reset */
    result = cyhal_wdt_init(&wdt_obj, cyhal_wdt_get_max_timeout_ms());
    CY_ASSERT(CY_RSLT_SUCCESS == result);
    cyhal_wdt_free(&wdt_obj);
#endif
    /* 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();

    /* Initialize retarget-io to use the debug UART port */
    result = cy_retarget_io_init(CYBSP_DEBUG_UART_TX, CYBSP_DEBUG_UART_RX,
                                 CY_RETARGET_IO_BAUDRATE);

    /* retarget-io init failed. Stop program execution */
         lt != CY_RSLT_SUCCESS)

         SSERT(0);
```

CDT Build Console [Hello_World]

```
 Total Internal Flash (Available)          8585216
 Total Internal Flash (Utilized)           52240

 Total Internal SRAM (Available)           1048576
 Total Internal SRAM (Utilized with heap)  389120


16:45:09 Build Finished. 0 errors, 0 warnings. (took 1m:22s.805ms)
```

Figure 14 Build the application

If you encounter errors, revisit prior steps to ensure that you accomplished all the required tasks.

Note: You can also use the command line interface (CLI) to build the application. See the "Using the command-line" section in the ***ModusToolbox™ user guide****. This document is located in the /ide_<version>/docs/ folder in the ModusToolbox™ installation.*

4.8 Part 5: Program the device

This section shows how to program the XMC7000 MCU device.

ModusToolbox™ software uses the OpenOCD protocol to program and debug applications on XMC7000 MCU devices. For ModusToolbox™ software to identify the device on the kit, the kit must be running KitProg3. See **Programming/debugging** for details.

If you are using a development kit with a built-in programmer, connect the board to your computer using the USB cable.

If you are developing on your own hardware, you may need a hardware programmer/debugger; for example, a **CY8CKIT-005 MiniProg4**.

1. **Program the application**.
   a) Connect to the board and perform the following step.
   b) Select the application project and click on the **<application name> Program (KitProg3_MiniProg4)** shortcut under the **Launches** group in the Quick Panel, as **Figure 15** 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.

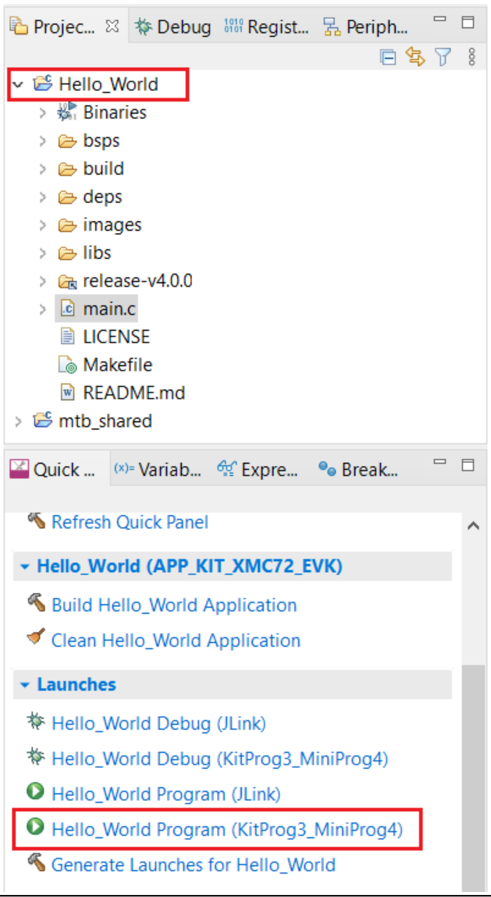
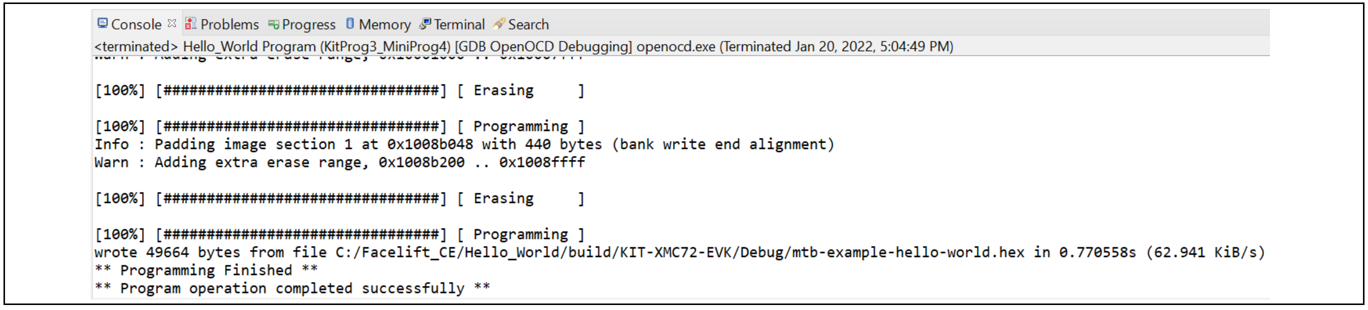

Projec...
Debug
Regist...
Periph...
Hello_World
Binaries
bsps
build
deps
images
libs
release-v4.0.0
main.c
LICENSE
Makefile
README.md
mtb_shared
Quick ...
Variab...
Expre...
Break...
Refresh Quick Panel
Hello_World (APP_KIT_XMC72_EVK)
Build Hello_World Application
Clean Hello_World Application
Launches
Hello_World Debug (JLink)
Hello_World Debug (KitProg3_MiniProg4)
Hello_World Program (JLink)
Hello_World Program (KitProg3_MiniProg4)
Generate Launches for Hello_World

Figure 15 Programming an application to a device

The **Console** view lists the results of the programming operation, as **Figure 16** shows.



```
<terminated> Hello_World Program (KitProg3_MiniProg4) [GDB OpenOCD Debugging] openocd.exe (Terminated Jan 20, 2022, 5:04:49 PM)

[100%] [#################################] [ Erasing     ]

[100%] [#################################] [ Programming ]
Info : Padding image section 1 at 0x1008b048 with 440 bytes (bank write end alignment)
Warn : Adding extra erase range, 0x1008b200 .. 0x1008ffff

[100%] [#################################] [ Erasing     ]

[100%] [#################################] [ Programming ]
wrote 49664 bytes from file C:/Facelift_CE/Hello_World/build/KIT-XMC72-EVK/Debug/mtb-example-hello-world.hex in 0.770558s (62.941 KiB/s)
** Programming Finished **
** Program operation completed successfully **
```

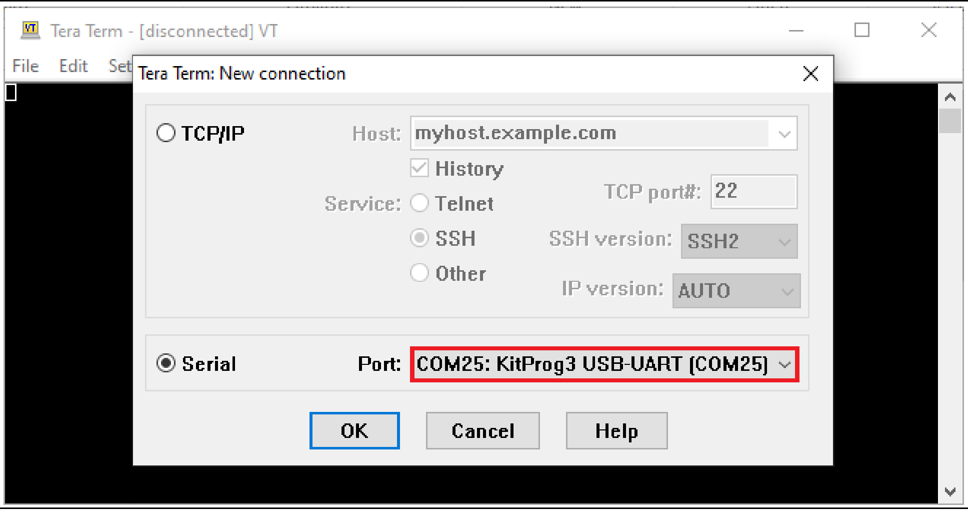
Figure 16 Console – programming results

4.9 Part 6: Test your design

This section describes how to test your design.

Follow the steps below to observe the output of your design. This note uses Tera Term as the UART terminal emulator to view the results. 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 17** shows. Note that your COM port number may be different.

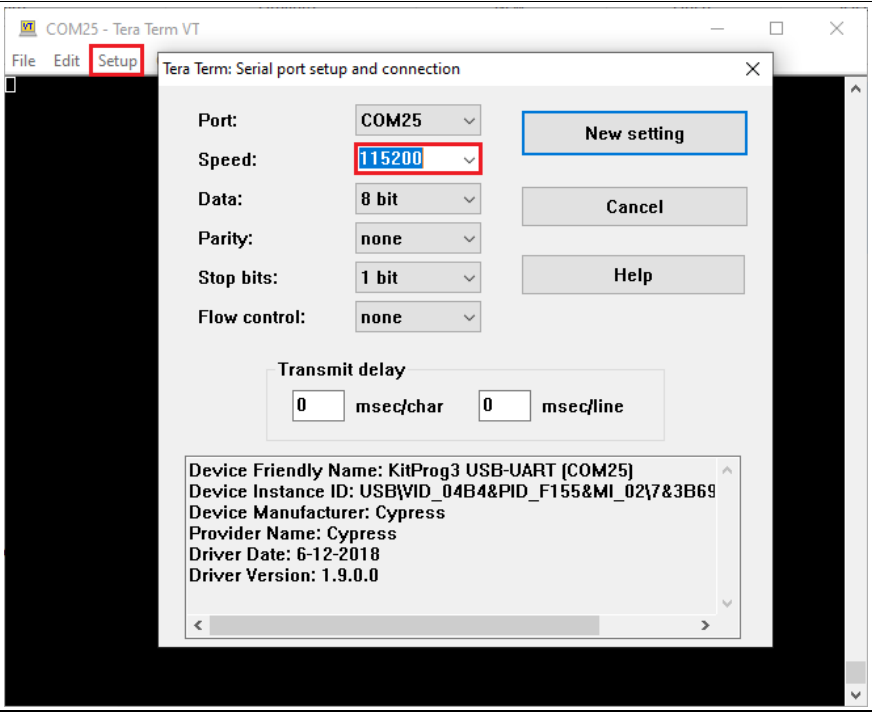


Serial [✓]   Port: COM25: KitProg3 USB-UART (COM25)

Figure 17 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 18** shows.



Port: COM25
Speed: 115200
Data: 8 bit
Parity: none
Stop bits: 1 bit
Flow control: none

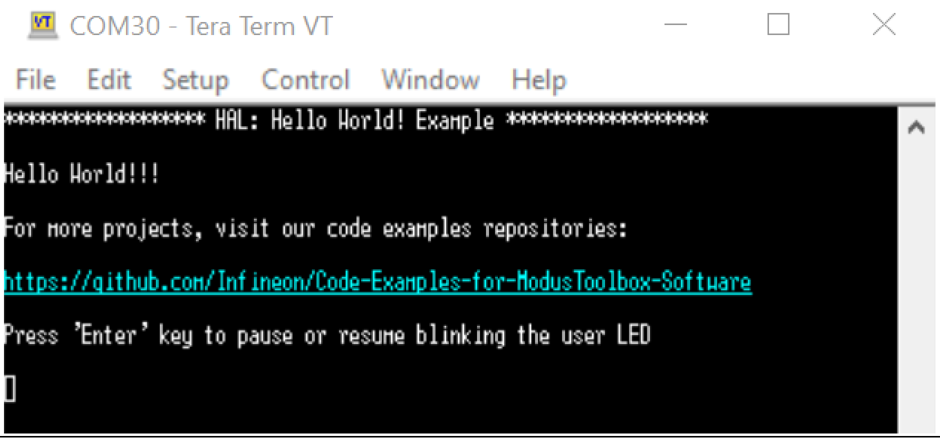
Transmit delay
0 msec/char 0 msec/line

```
Device Friendly Name: KitProg3 USB-UART (COM25)
Device Instance ID: USB\VID_04B4&PID_F155&MI_02\7&3B69
Device Manufacturer: Cypress
Provider Name: Cypress
Driver Date: 6-12-2018
Driver Version: 1.9.0.0
```

Figure 18 Configuring the baud rate in Tera Term

3. **Reset the device.**

   Press the reset switch (**SW1**) on the kit. A message appears on the terminal as **Figure 19** shows. The user LED on the kit will start blinking.

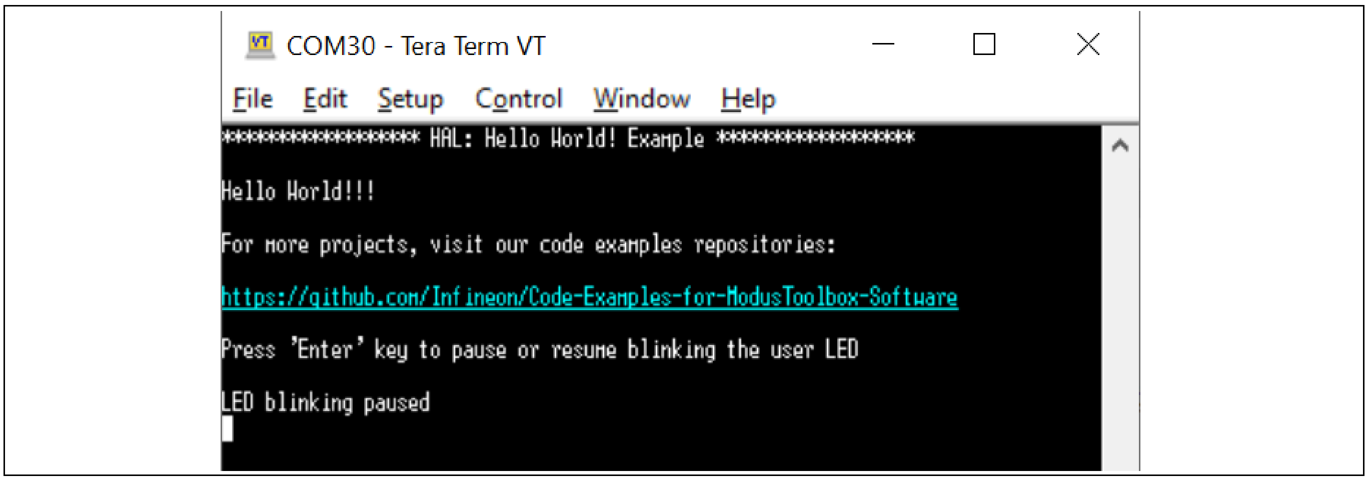


```
****************** HAL: Hello World! Example ******************

Hello World!!!

For more projects, visit our code examples repositories:

https://github.com/Infineon/Code-Examples-for-ModusToolbox-Software

Press 'Enter' key to pause or resume blinking the user LED
```

Figure 19 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 **Figure 20** shows.



```
****************** HAL: Hello World! Example ******************

Hello World!!!

For more projects, visit our code examples repositories:

https://github.com/Infineon/Code-Examples-for-ModusToolbox-Software

Press 'Enter' key to pause or resume blinking the user LED

LED blinking paused
```

Figure 20 Printed UART message

5 Summary

This application note explored the XMC7000 MCU device architecture and the associated development tools. XMC7000 MCU is a truly programmable embedded system-on-chip with configurable analog and digital peripheral functions, memory, and a triple-CPU system on a single chip. The integrated features and low-power modes make XMC7000 MCU an ideal choice for industrial drives, robotics, transportation, and other related applications.

References

For a complete and updated list of XMC7000 MCU code examples, please visit our **GitHub**. For more XMC7000 MCU-related documents, please visit our **XMC7000 MCU** product web page.

Table 3 lists the system-level and general application notes that are recommended for the next steps in learning about XMC7000 MCU and ModusToolbox™.

Table 3 General and system-level application notes

Document	Document name
AN234224	Hardware design guide for the XMC7000 family
AN225588	Using ModusToolbox™ software with a third-party IDE
AN234021	Low power mode procedure in XMC7000 family

Table 4 lists the application notes (AN) for specific peripherals and applications.

Table 4 Documents related to XMC7000 MCU features

Document	Document name
System resources, CPU, and interrupts	
AN234226	How to use interrupt in the XMC7000 family
AN234225	How to use DMA in the XMC7000 family
Peripherals	
AN230938	How to use SCB in XMC7000 family
AN234380	How to use ethernet controller in XMC7000 family
AN234227	Using the SMIF in the XMC700 family
AN234197	How to use trigger multiplexer in XMC700 family
AN234002	How to use CAN FD in XMC7000 family
AN234119	Timer, Counter and PWM (TCPWM) usage in XMC7000 family

Glossary

This section lists the most commonly used terms that you might encounter 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.
- **Hardware Abstraction Layer (HAL)**: The HAL wraps the lower level drivers (like **MTB-PDL-CAT1**) and provides a high-level interface to the MCU. The interface is abstracted to work on any MCU.
- **KitProg**: The KitProg is an onboard programmer/debugger with USB-I2C and USB-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 XMC7000 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 version	Date of release	Description of changes
**	2022-01-25	Initial release.
*A	2022-06-17	Updated **Figure 3**, **Figure 4**, **Figure 8**, **Figure 9**, **Figure 10**, **Figure 11**, **Figure 12**, **Figure 14**, and **Figure 15**.

Edition 2022-06-17

Published by

Infineon Technologies AG

81726 Munich, Germany

Do you have a question about this document?

Go to: www.infineon.com/support

Document reference

002-34334 Rev. *A