Dual-image DFU bootloader functionality for EZ-PD™ PMG1-S3 MCU

About this document

Scope and purpose

This application note builds upon the EZ-PD™ PMG1-S3 Device Firmware Upgrade (DFU) code example, introducing the dual image bootloader architecture and describing how the EZ-PD™ PMG1-S3 MCU selects the image to run as the primary application during boot-up.

The benefits of implementing a bootloader and partitioning the internal flash into two separate spaces for firmware updates include increased reliability and improved fault recovery. If the new firmware has issues (such as boot failures, crashes, or power interruptions), the bootloader can automatically roll back to the old version, preventing the device from becoming bricked. This significantly enhances the security of remote updates, particularly in unattended or physically inaccessible applications. The device can receive and write the new firmware to the inactive partition in the background while still running the old firmware, without the need to immediately restart or halt services. The bootloader can integrate encryption, signature verification, or rollback prevention mechanisms to ensure that only legitimate firmware can be accepted.

This document applies to applications that require the USB-C PD controller functionality integrated into the MCU. It enables firmware updates for the MCU itself via USB-CDC, UART, or I2C. Partitioning the memory into two separate spaces allows for a more flexible design of the system's firmware update mechanism. This allows general-purpose MCUs to support more flexible designs while meeting charging and power-supply requirements.

Intended audience

This document is intended for application engineers who develop dual-image bootloaders for the EZ-PD™ PMG1-S3 MCU.

Key points

- Explains a dual-image DFU bootloader for EZ-PD™ PMG1-S3 MCU that selects the active firmware during startup
- Describes flash partitioning, metadata, and boot-selection logic for managing primary and secondary application images
- Demonstrates ModusToolbox™ linker scripts, configuration, and code changes that are required to build bootloader and app images
- Provides programming and verification workflows using USB-CDC, UART, and DFU Host Tool to validate image updates
- Design benefits including rollback support, fault recovery, and background firmware delivery

About this document

Product family

Infineon's EZ-PD™ PMG1 (Power Delivery Microcontroller Gen 1) is a portfolio of high-voltage microcontrollers (MCUs) with USB Power Delivery (PD). The EZ-PD™ PMG1 MCUs have a market-proven USB PD stack, ensuring spec compliance and interoperability.

Target applications

- Consumer electronics
- Battery backup unit (BBU)

Table of contents

About this document **1**
Table of contents **3**
1 Introduction **4**
2 DFU bootloader overview **5**
2.1 EZ-PD™ PMG1-S3 MCU DFU operation upgrade 5
2.2 EZ-PD™ PMG1-S3 MCU DFU function flow 6
2.3 EZ-PD™ PMG1-S3 MCU flash memory layout 8
3 Single image DFU in ModusToolbox™ **9**
4 Programming and debugging single image DFU bootloader **14**
4.1 Target board CY7113 EZ-PD™ PMG1-S3 MCU Prototyping Kit 14
4.2 Create an application DFU_LED_Blinky project 14
4.3 DFU update verification 17
4.3.1 Programming DFU_Bootloader application 18
4.3.2 DFU Host Tool 20
4.3.3 Update the main application using the DFU Host Tool 21
5 Dual image bootloader use cases **22**
6 EZ-PD™ PMG1-S3 MCU flash structure **23**
7 Dual image bootloader features **24**
8 Modify DFU bootloader in ModusToolbox™ for dual image **26**
8.1 Modify the sample code of the DFU bootloader 26
8.1.1 EZ-PD™ PMG1-S3 MCU internal flash size allocation 27
8.1.2 Configure the EZ-PD™ PMG1-S3 MCU 29
8.1.3 Modify the DFU_Bootloader code 35
9 Modify DFU_Blinky_LED in ModusToolbox™ for the application **43**
10 Programming and verifying the entire operating mechanism **52**
11 Related resources **57**
References **58**
Revision history **59**
Disclaimer **60**

1 Introduction

The EZ-PD™ PMG1 MCU Device Firmware Update (DFU) bootloader is a software component that initializes the memory hardware required for flash read and write operations, configures protection units, and runs the main application during system startup. It plays a crucial role in the boot-up sequence of an MCU-based system.

The bootloader is designed with simplicity as its core principle, occupying minimal resources, with the primary focus on enabling extremely fast boot times. It often resides in a small portion of non-volatile memory.

This document uses the Device Firmware Update (DFU) feature for advanced applications, allowing users to store two different applications in the MCU's internal flash memory simultaneously via DFU or to use them as backup boot firmware images. The firmware development described in this document is based on ModusToolbox™ 3.8 and Eclipse 2026.1.

Key functions of the bootloader include:

- **EZ-PD™ PMG1-S3 MCU hardware initialization:** Configures basic MCU peripherals, such as system clocks, communication interfaces, and I/O interfaces
- **System boot source selection:** Determines which image to load as the main application, such as Image 1 or Image 2 in internal flash memory
- **EZ-PD™ PMG1-S3 MCU firmware update support:** Allows EZ-PD™ PMG1-S3 MCU to update its firmware in the field, typically through a secure and reliable mechanism, via a communication interface, such as I2C, UART, or USB-CDC

To conclude, a bootloader is a crucial component that ensures a seamless and secure startup of MCU-based systems, facilitating firmware updates and preserving system integrity. There are two types of bootloaders that EZ-PD™ PMG1-S3 MCU supports, namely the Device Firmware Update (DFU) bootloader and the MCUboot self-based bootloader.

EZ-PD™ PMG1-S3 MCU DFU bootloader

The Device Firmware Update (DFU) is a process that transfers firmware to the device. The target device (e.g., MCU) and the host machine (e.g., Computer) establish a connection to download the firmware. This connection can be wired (UART, I2C, USB). The target device receives the firmware from the host, validates it, and executes it. For EZ-PD™ PMG1-S3 MCU, Infineon offers DFU middleware (MW), which provides an over-the-wire update solution.

DFU is a common part of an MCU system. ModusToolbox™ enables in-field updates of a product's firmware. In a typical product, firmware is stored in an MCU's flash memory and can be used to build a small-footprint bootloader for EZ-PD™ PMG1-S3 MCUs.

DFU supports the development of bootloaders with or without a transport interface. For example, you can build a DFU-based bootloader with transport capabilities, such as UART and USB. In another use case, you can build a DFU-based bootloader without any transport interface. This application note also describes the DFU usage supported by the EZ-PD™ PMG1-S3 MCU.

EZ-PD™ PMG1-S3 MCU bootloader implements the following key functionalities:

- Firmware image validation
- Firmware image upgrade
- MCU reset recovery/power fail-safe mechanism
- Dual-image support

2 DFU bootloader overview

At the factory, initial programming of firmware into a product is typically done at PCB assembly time, using the MCU's JTAG or SWD interface. However, these interfaces are not usually available in the field and therefore are generally not used for firmware updates. A more efficient way to update firmware in the field is to use an existing connection between the product and the outside world. The connection can be a standard communication port, such as I2C, USB, or UART; a wireless channel, such as Bluetooth® Low Energy; or a custom protocol. The DFU bootloader is a sample project provided by ModusToolbox™ that can be used directly. It is very simple internally and includes the Hardware Abstraction Layer (HAL) software layer for the EZ-PD™ PMG1-S3 MCU. When combined with ModusToolbox™ Device Configurator, it allows you to easily add the required custom features.

2.1 EZ-PD™ PMG1-S3 MCU DFU operation upgrade

The **DFU Host Tool** (typically running on the host PC) sends the application (\<APPNAME>.cyacd2) to the device. The bootloader receives the application image in chunks and overwrites the existing application image (see Figure 1). If a DFU session is interrupted, the application firmware will not be in a usable state. However, the device can still run the bootloader and perform the necessary actions to download and install the application in subsequent DFU sessions.

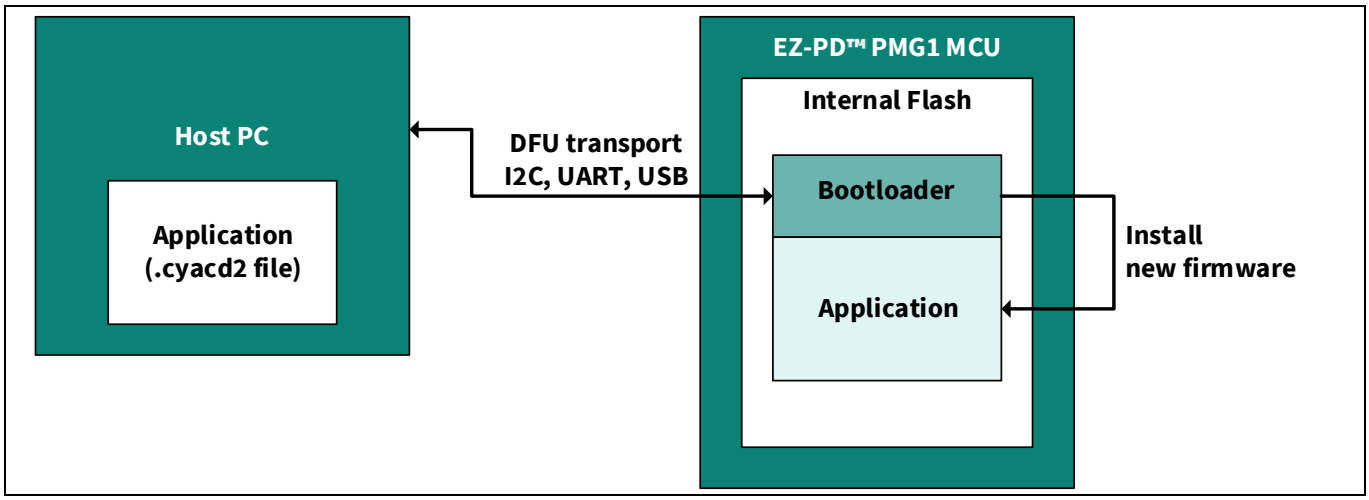


Figure 1 EZ-PD™ PMG1-S3 MCU DFU update operation using overwrite

The bootloader supports I2C, UART, and USB-CDC interfaces for communicating with the DFU Host Tool. The bsps directory includes all the necessary configurations to select the supported interfaces. See Table 1 for the default configuration details. These default configurations can be changed according to the use case. However, you must ensure that the configuration of the DFU Host Tool matches the bootloader's configuration.

Table 1 DFU transport configurations

	Default	Description
DFU transport: I2C		
Mode	Slave	The device acts as a slave
Address	16	7-bit slave device address
DFU transport: UART		
Com mode	Standard	Standard UART operation
Data width	8 bits	8 bits of data in a UART Frame

	Default	Description
Parity	None	Parity bit is not used
Stop bits	1	One stop bit used
Baud rate	115200 bps	The UART baud rate is 115200 bps
DFU transport: USB-CDC		
Endpoints mask	255	–
Endpoint 1 transfer type	Interrupt	Interrupt
Endpoint 2 transfer type	Bulk	To download and verify the binary image
Endpoints buffer management	Manual CPU	USB supports communication using Auto/Manual DMA
Endpoints access type	8-bit	–

2.2 EZ-PD™ PMG1-S3 MCU DFU function flow

While the application continues executing its normal tasks, it downloads the new application to a temporary location. After the new application is verified for validity, it is copied to the correct location in flash, and control is transferred to it. Figure 2 shows the process flow diagram.

2 DFU bootloader overview

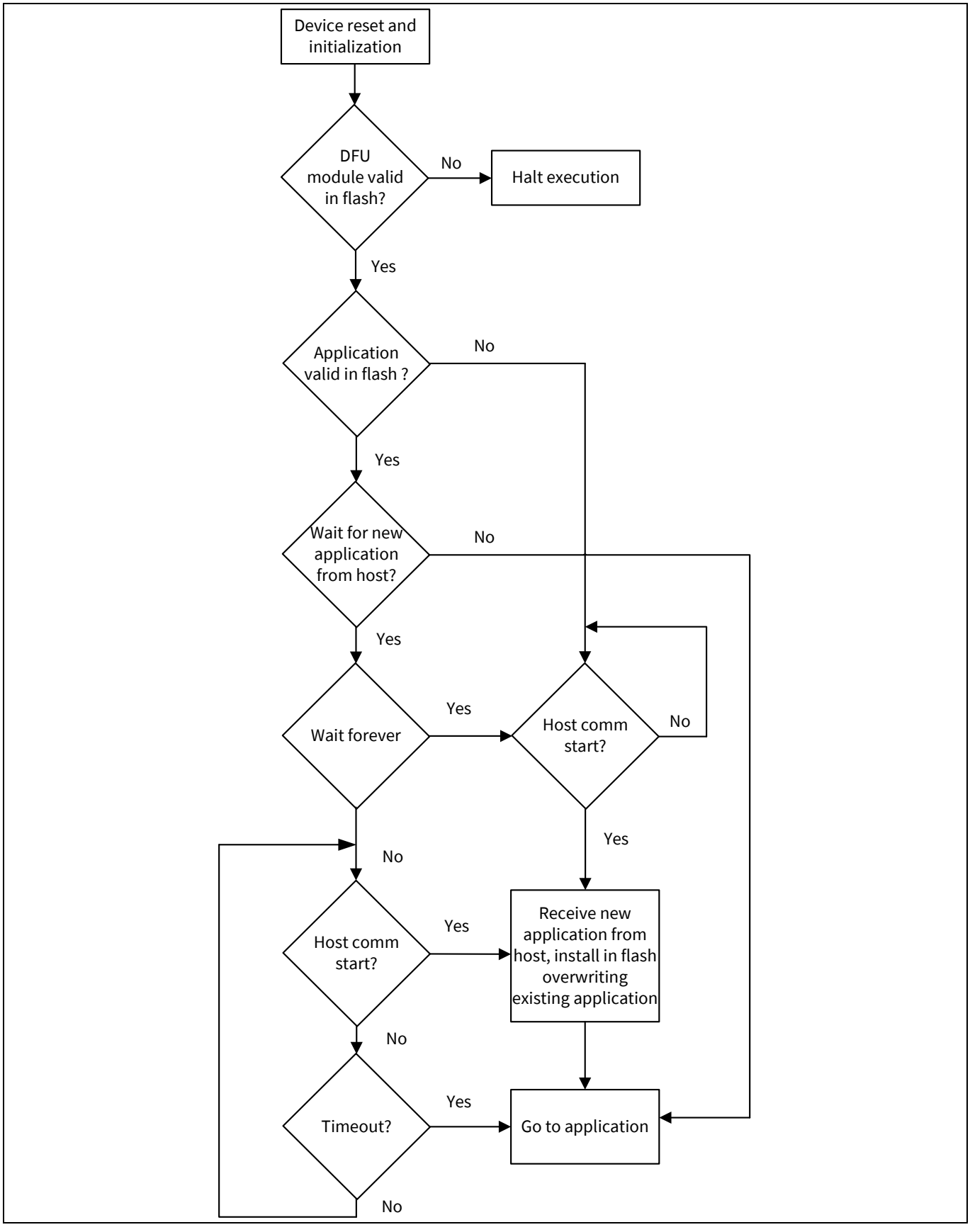


Figure 2 DFU function flow

2.3 EZ-PD™ PMG1-S3 MCU flash memory layout

The first 24 KB of flash is reserved for the bootloader. The size allocated to the application firmware can vary depending on the amount of flash memory available on the target device. The size of this empty/reserved region depends on the available flash size of the selected target device. See the respective device datasheets for details. The last 1 KB region of the flash is reserved for bootloader metadata (see Figure 3).

To change the memory layout or usage, update the respective target linker script files. The linker scripts can also be modified to define dedicated memory regions for each application. Locate the corresponding dfu_cm0p_app0.ld file in the DFU project path created in the *..\<root>\mtw\DFU_Bootloader\templates\TARGET_PMG1CY7113\COMPONENT_CM0P\TOOLCHAIN_GCC_ARM*\dfu_cm0p_app0.ld. The linker scripts can also be modified to define dedicated memory regions for each application.

The RAM is shared by the bootloader (mtb-example-pmg1-dfu-bootloader) and the blinky (mtb-example-pmg1-dfu-blinky-led) applications, with a common area used by both projects.

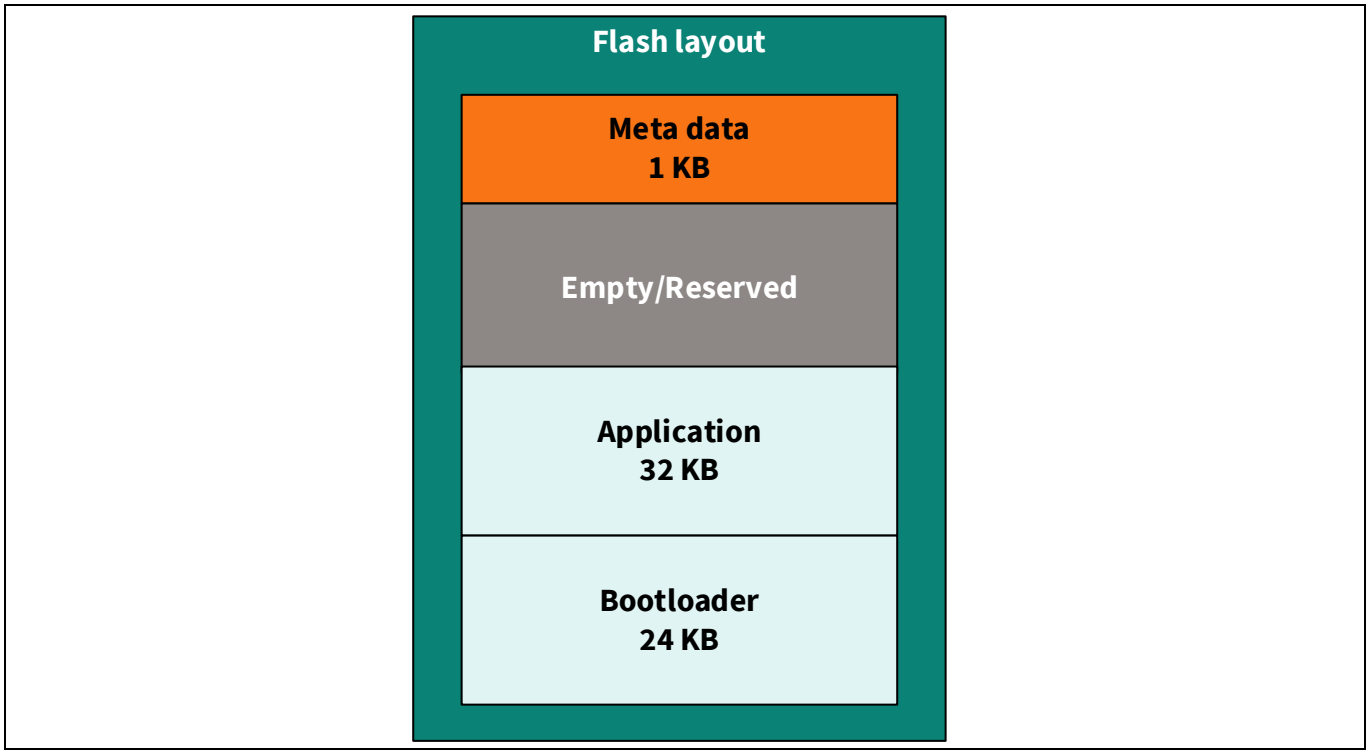


Figure 3 Memory layout

3 Single image DFU in ModusToolbox™

This application note is based on ModusToolbox™ 3.8 and Eclipse IDE for ModusToolbox™ version 2026.1.

This section demonstrates how to create a new DFU application for the EZ-PD™ PMG1-S3 device by using the included sample project. This bootloader runs on the CM0+ core. For more information on the design aspect, see the mtb-example-pmg1-dfu-bootloader README file.

1. Install ModusToolbox™ v3.0 or above
2. Create a new **ModusToolbox™ Application** as shown in Figure 4

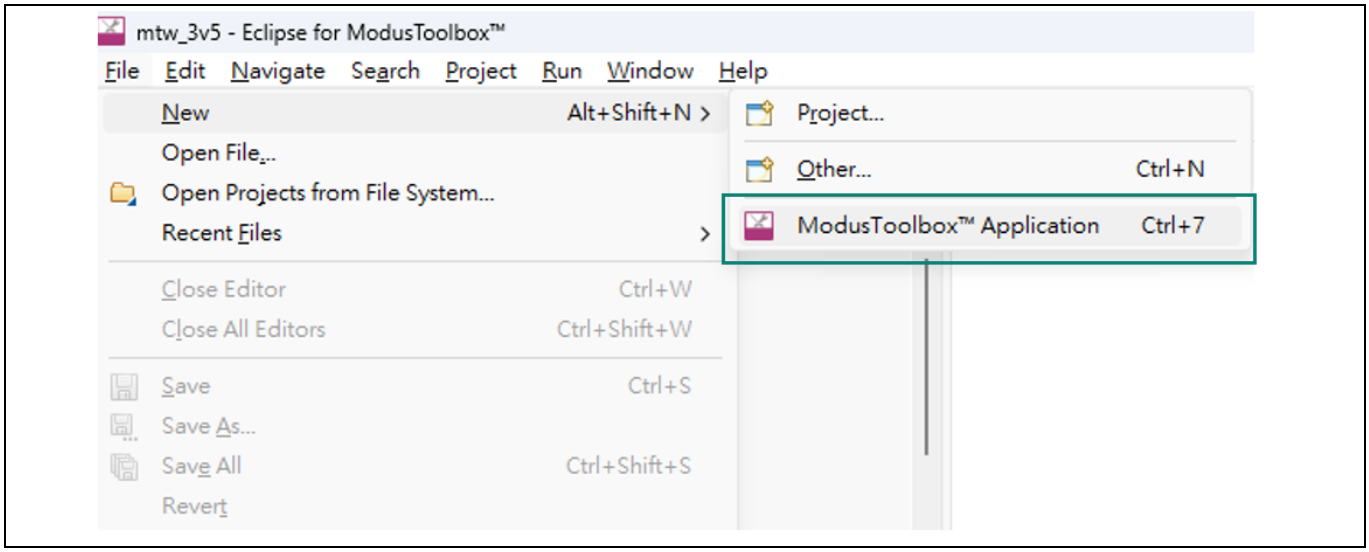


Figure 4 Create DFU ModusToolbox™ Application

3. Select the **PMG1-CY7113** Prototyping Kit as shown in Figure 5 and click **Next >**

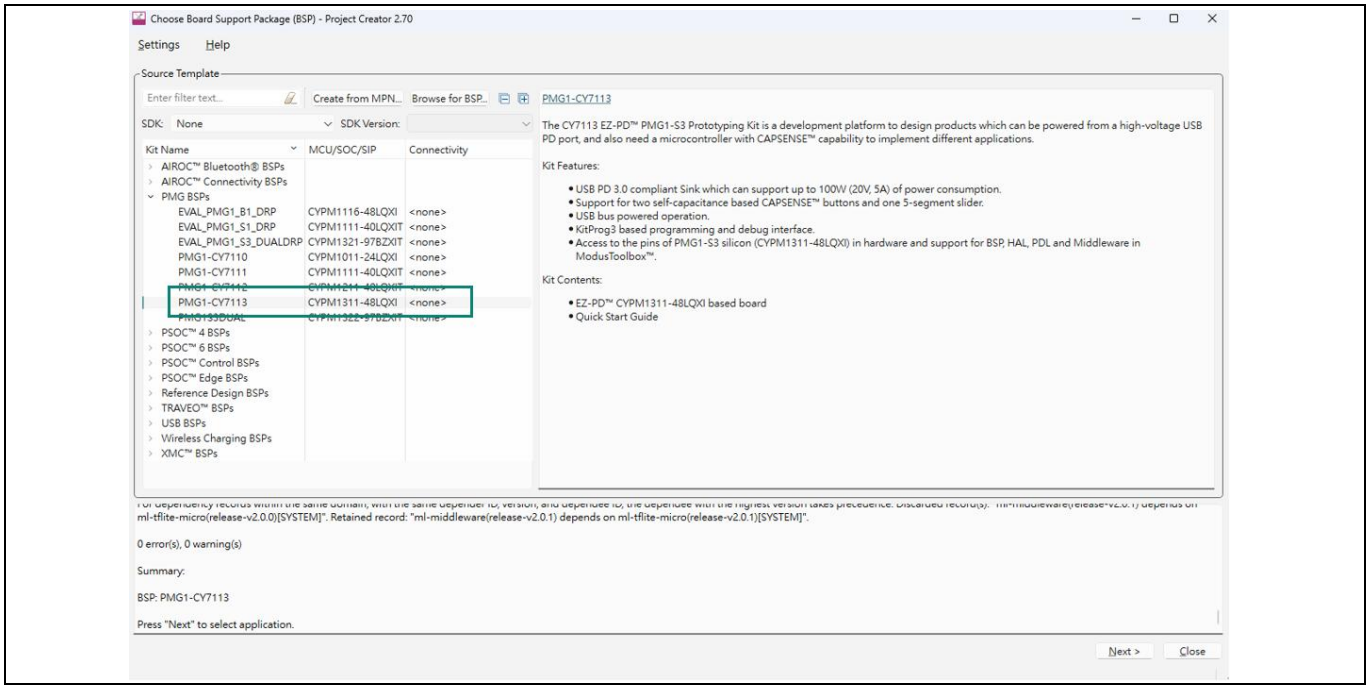


Figure 5 Choose the target kit

4. Select **DFU_Bootloader** as shown in Figure 6 and click **Next >**

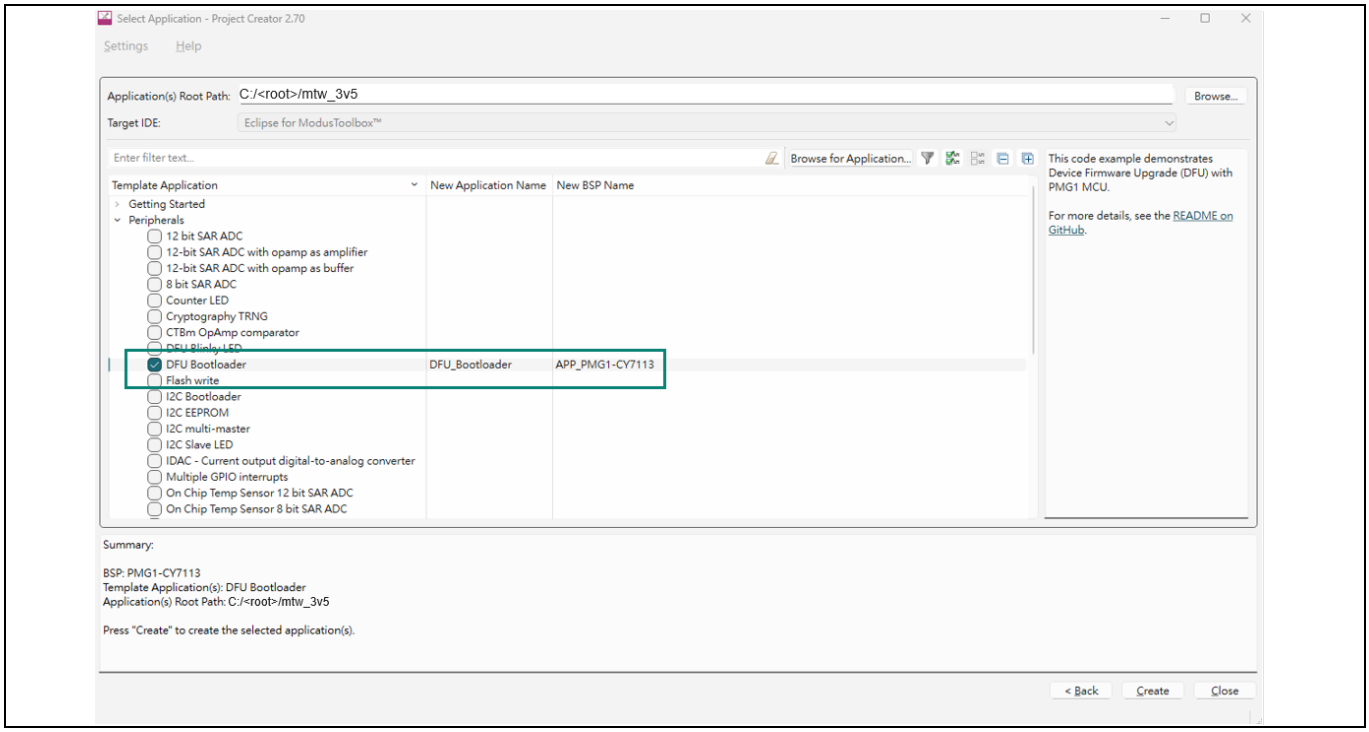


Figure 6 Choose the DFU Bootloader application

Note: After selecting the DFU_Bootloader application, it takes a while to load all the required libraries.

The bootloader application consists of the *DFU_Bootloader* project. Figure 7 shows the folder structure of the bootloader application.

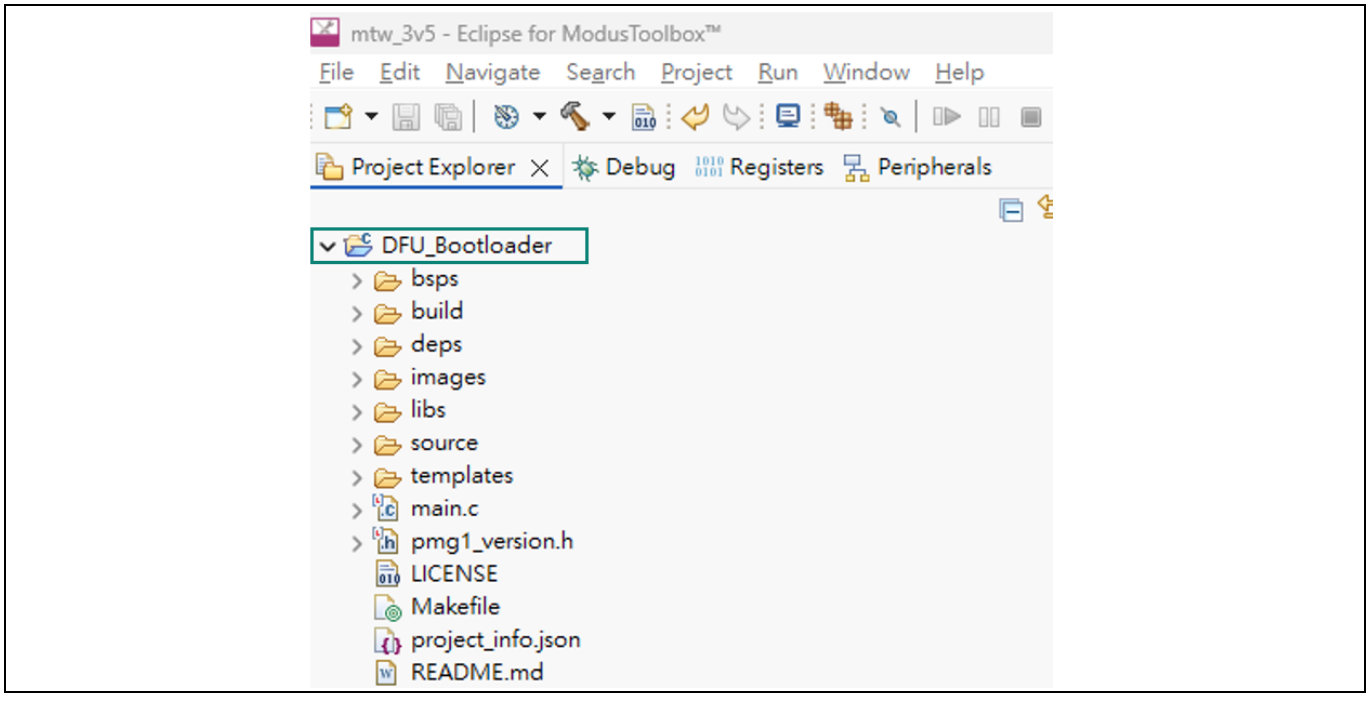


Figure 7 Bootloader directory structure

The bootloader project in Figure 7 is the boot application that does verification and validation of the blinky project. However, the core bootloader library, also known as DFU, is in the *\mtb_shared* directory of the workspace as shown in Figure 8

The DFU library performs most of the functions typically associated with a bootloader. In particular, the missing piece is the final step of jumping to the main image. Instead, the last step is implemented by the boot application. Bootloader functionality is separated in this way to enable unit testing of the bootloader. A library can be unit tested, but an application cannot. Therefore, functionality is delegated to the DFU library whenever possible.

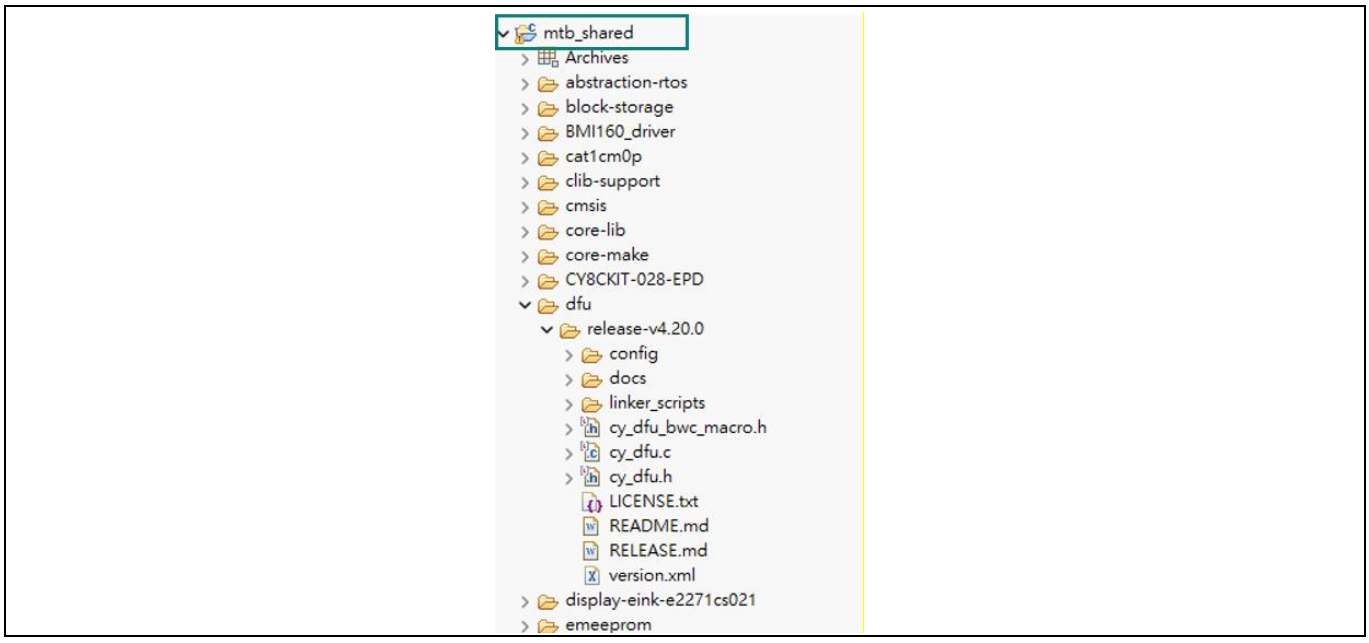


Figure 8 DFU Bootloader core firmware

5. Open the *Makefile*, search for **TRANSPORT_OPT?=USB_CDC**, and confirm that it is set to **USB_CDC**. The DFU bootloader supports UART, I2C, and USB-CDC. In this application, USB-CDC is used as the interface for updating and transmitting data as shown in Figure 9

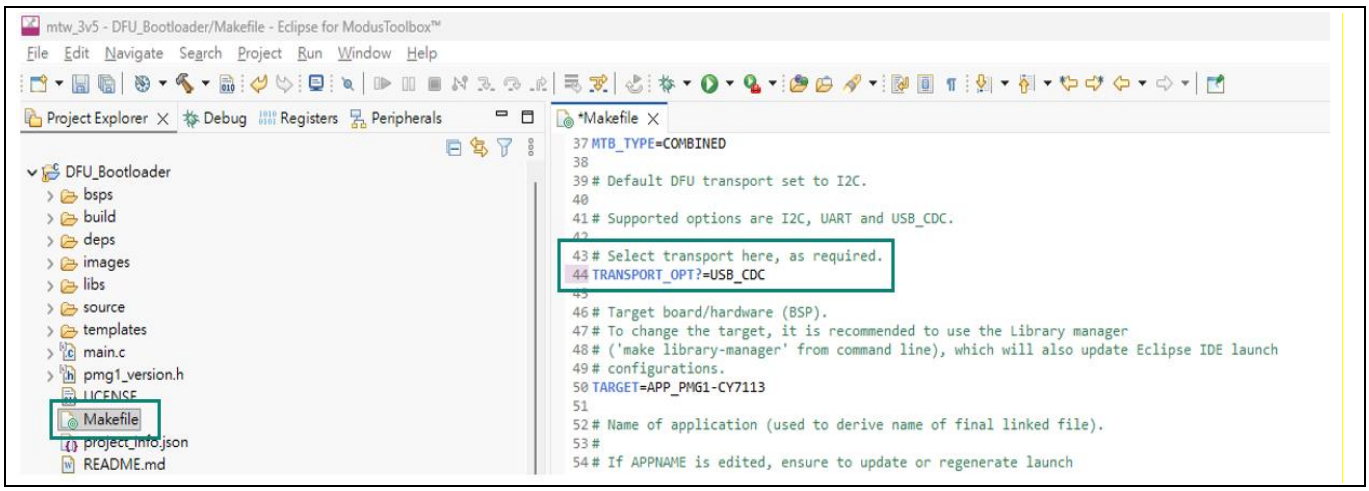


Figure 9 Configure Makefile

6. Right-click on the **DFU_Bootloader** application and select **Build Project** as shown in Figure 10 to build the application

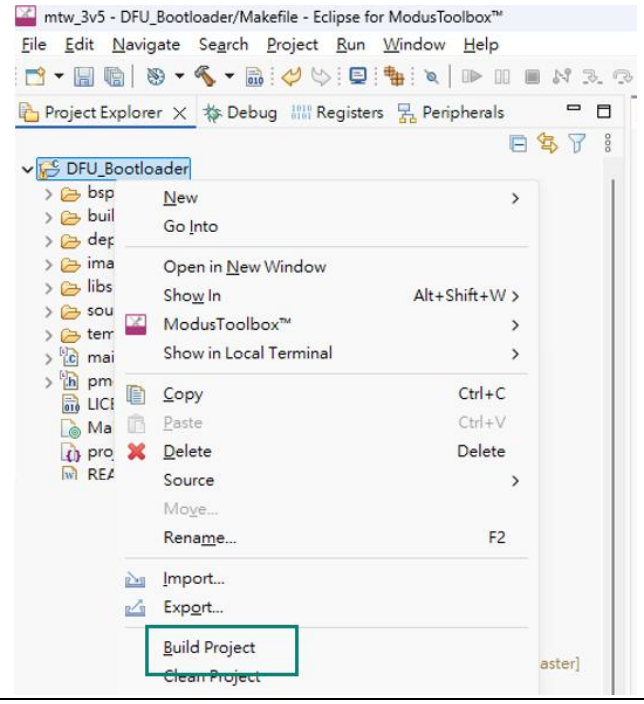


Figure 10 Building application

The build process builds the **DFU_bootloader** project, as shown in Figure 11.

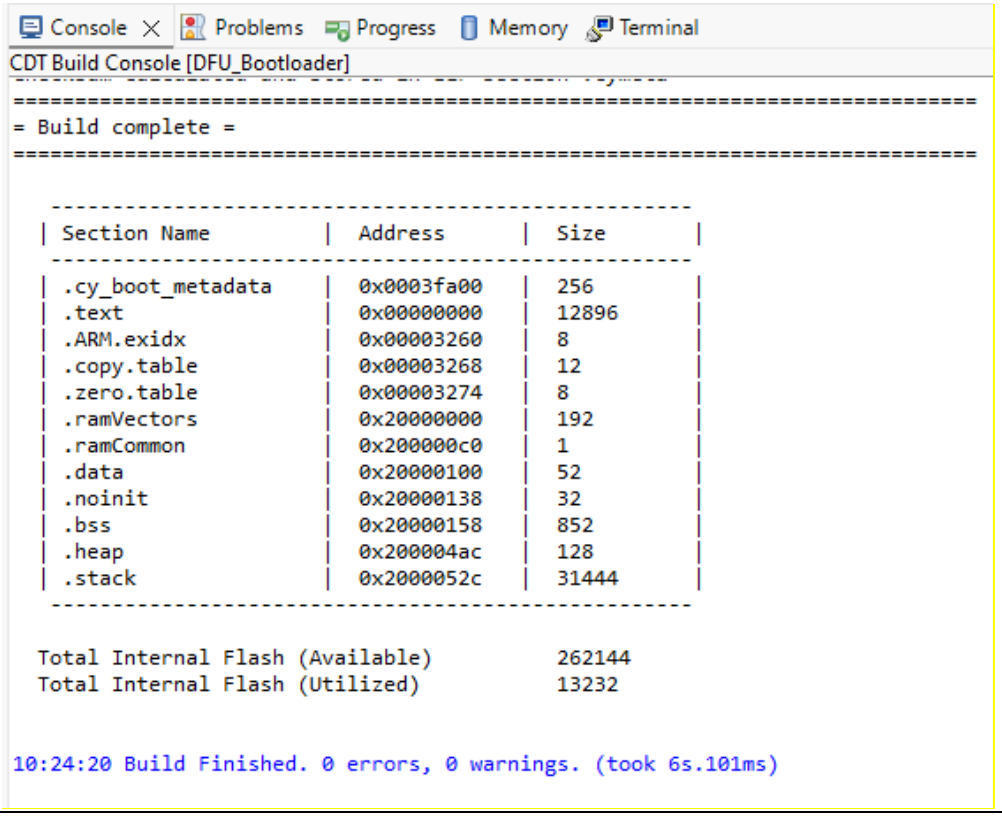


Figure 11 DFU Bootloader successful build

7. In the default path ..\<*root*>*DFU_Bootloader**build**USB_CDC**APP_PMG1-CY7113**Debug*, verify whether a hex file is generated. Verify that the flashing program file generated by *DFU_Bootloader* has a file extension of *.hex*, while the file generated for DFU updating the internal main app program of the MCU has a file extension of *cyacd2* as shown in Figure 12

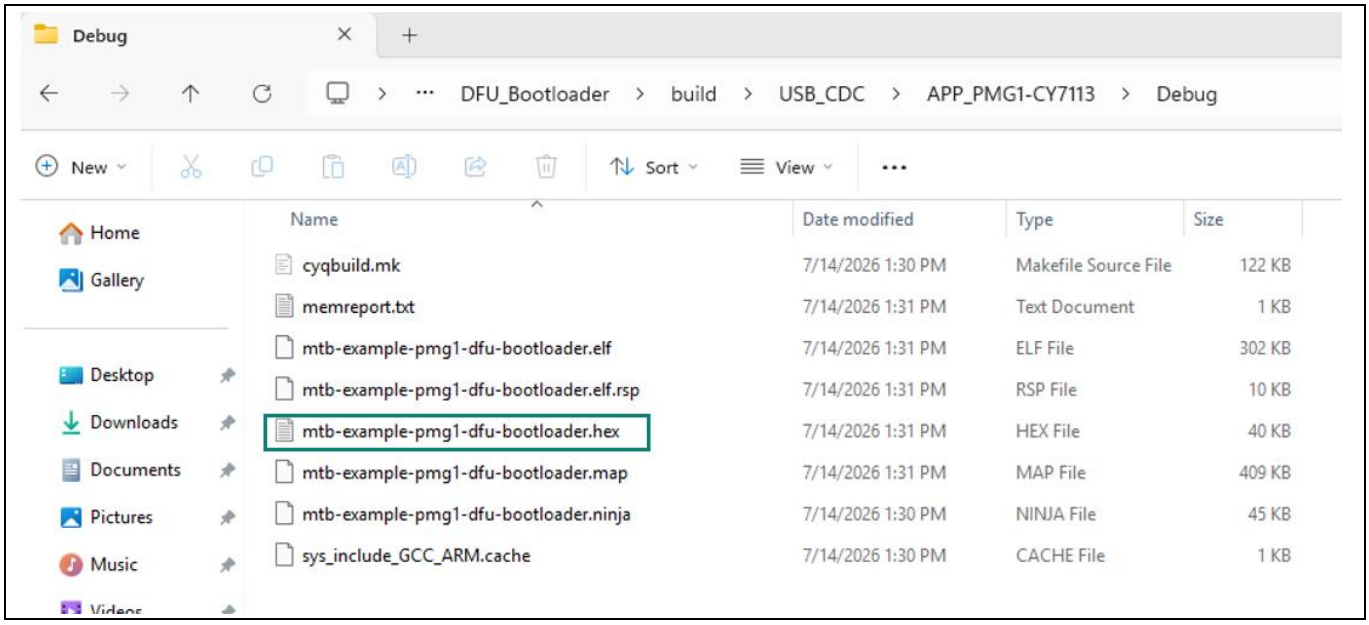


Figure 12 Hex file generated after building the project

4 Programming and debugging single image DFU bootloader

Completing this verification requires not only the bootloader project but also an application project to verify that, after the bootloader successfully updates the firmware, the system can smoothly enter and execute the application program.

4.1 Target board CY7113 EZ-PD™ PMG1-S3 MCU Prototyping Kit

Verification is performed using the CY7113 EZ-PD™ PMG1-S3 MCU Prototyping Kit. The CY7113 EZ-PD™ PMG1-S3 MCU Prototyping Kit (Figure 13) is a development platform for designing products that can be powered from a high-voltage USB PD port while requiring a microcontroller with CAPSENSE™ capability for various applications. This platform supports the USB PD 3.1 protocol with up to 140 W (28 V, 5 A) of power delivery. It also includes two self-capacitance-based CAPSENSE™ buttons and one 5-segment slider.

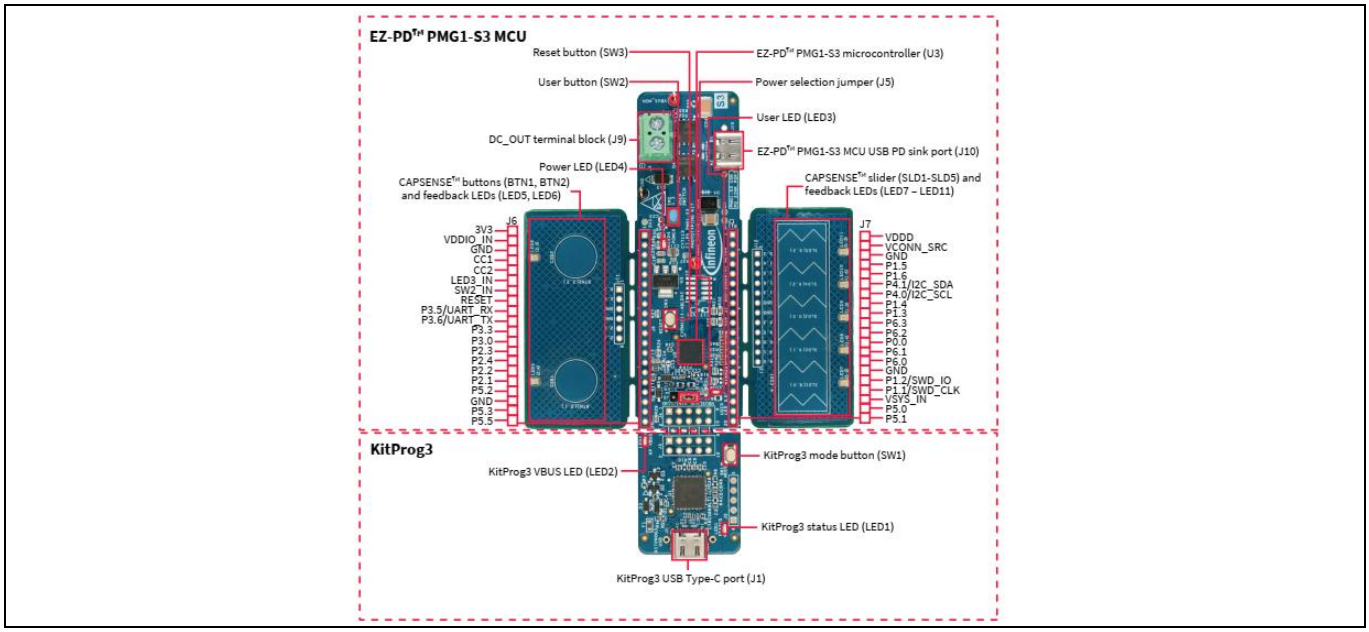


Figure 13 CY7113 EZ-PD™ PMG1-S3 MCU Prototyping Kit

4.2 Create an application DFU_LED_Blinky project

This section explains how to create an application that updates through the DFU bootloader to blink an LED on the evaluation board. See mtb-example-pmg1-dfu-blinky-led README file.

1. Create a new **ModusToolbox™ Application** as shown in Figure 14

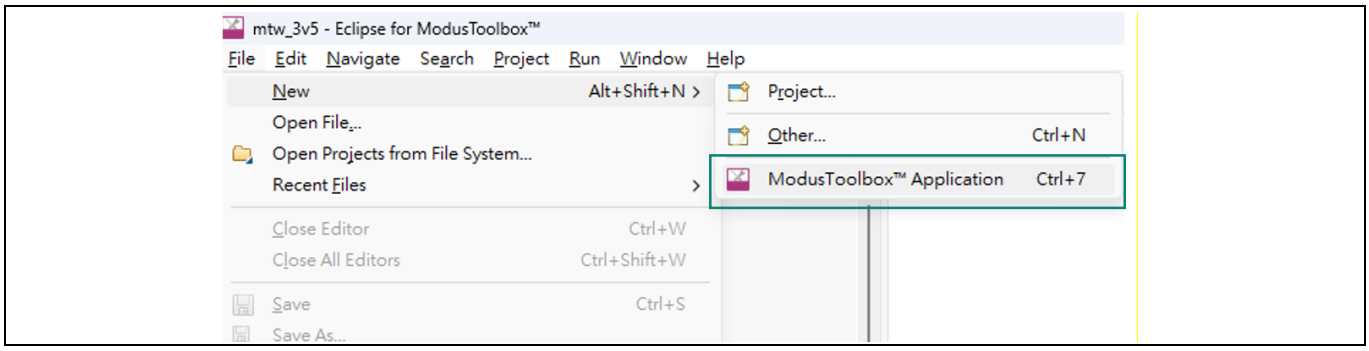


Figure 14 Create DFU ModusToolbox™ Application

2. Select the **PMG1-CY7113** Prototyping Kit as shown in Figure 15, and click **Next >**

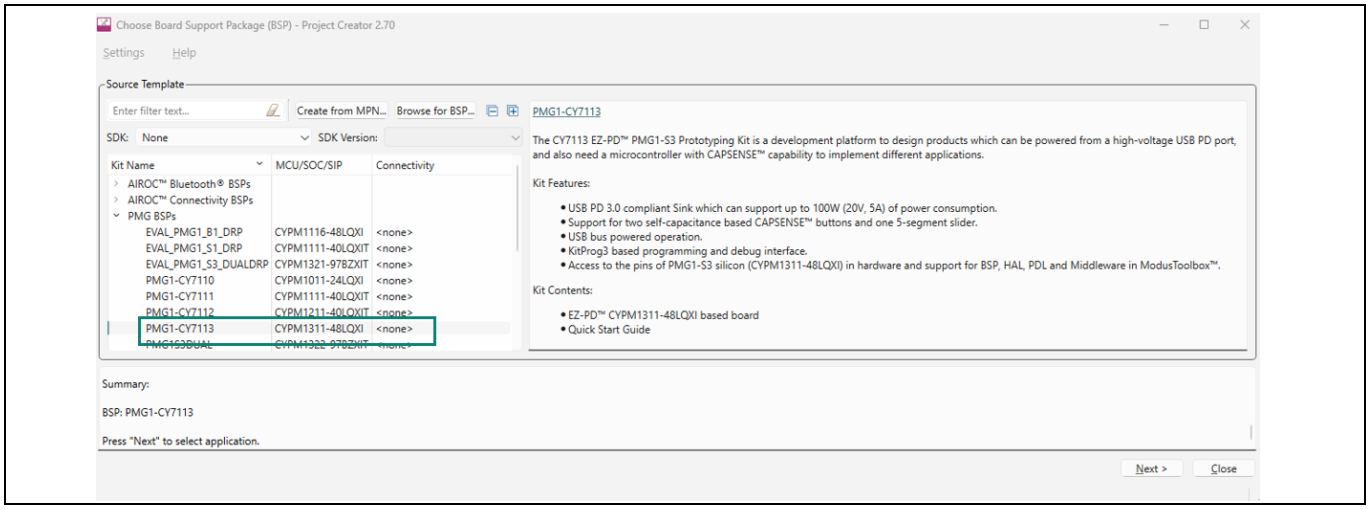


Figure 15 Choose the target kit

3. Select **DFU_Blinky_LED** application as shown in Figure 16 and click **Next >**

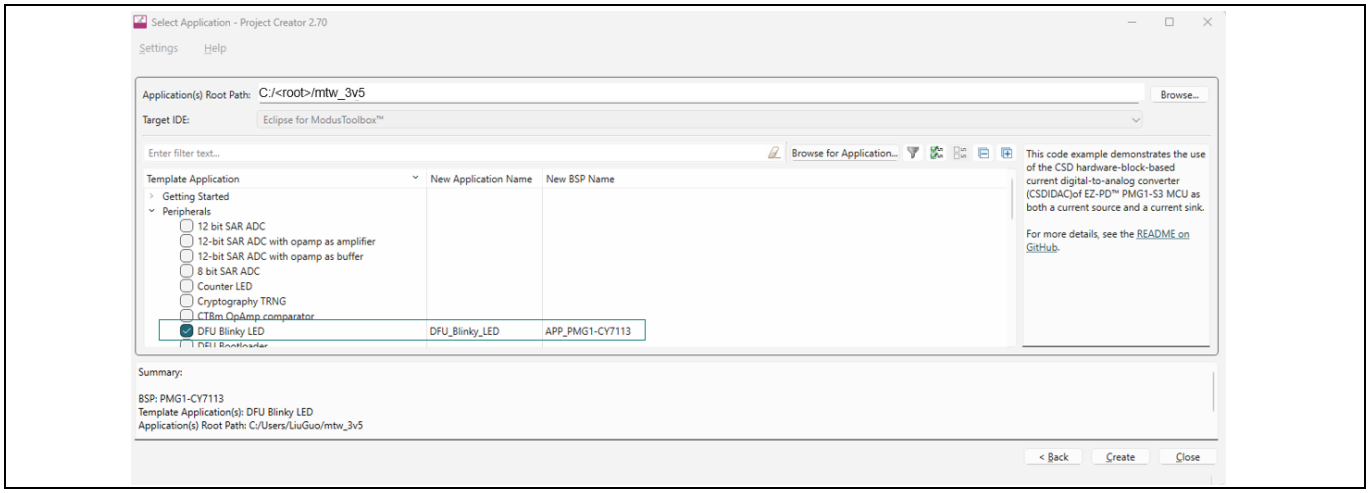


Figure 16 Select DFU_Blinky_LED application

The blinky LED application consists of the DFU_Blinky_LED project. Figure 17 shows the folder structure of the DFU_Blinky_LED application

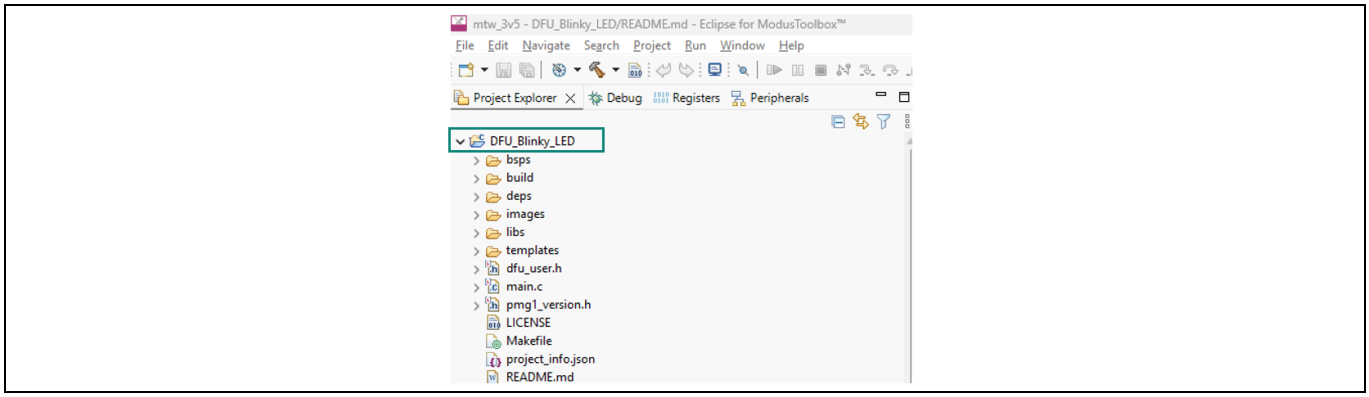


Figure 17 DFU_Blinky_LED directory structure

4. Right-click on the **DFU_Blinky_LED** application and select **Build Project** to build the application as shown in Figure 18

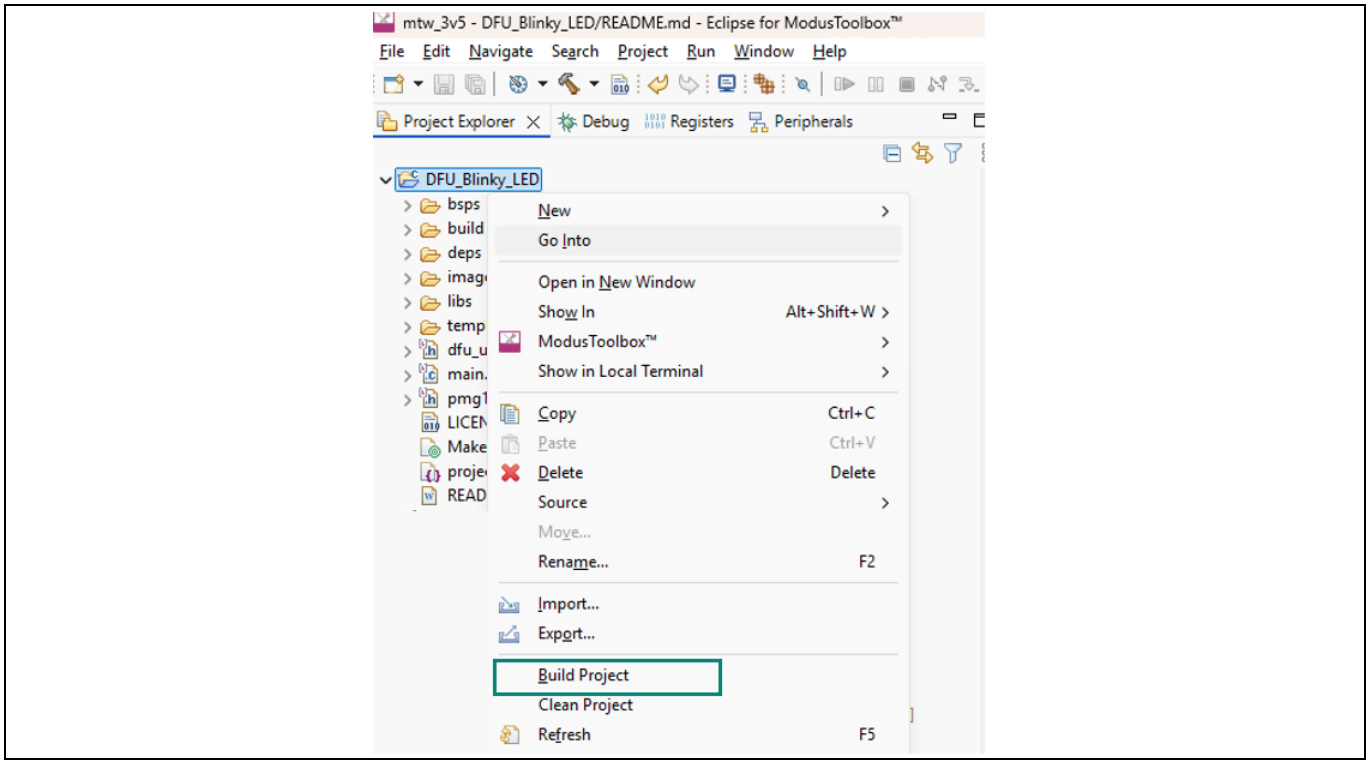


Figure 18 Building DFU_Blinky_LED application

The build process builds the **DFU_Blinky_LED** project, as shown in Figure 19.

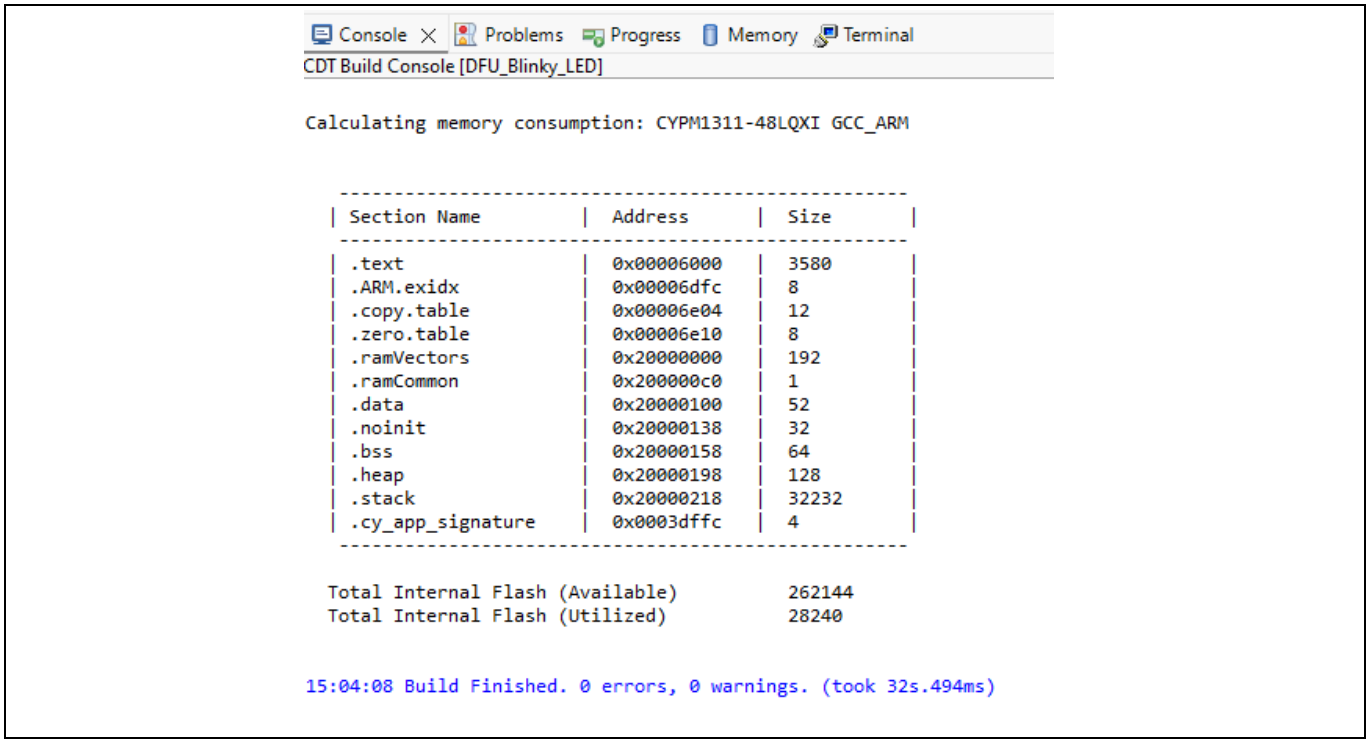


Figure 19 DFU_Blinky_LED successful build

5. See the generated *mtb-example-pmg1-dfu-blinky-led_crc.cyacd2* file in the *..\<root>\mtw\DFU_Blinky_LED\build\APP_PMG1-CY7113\Debug* folder. This file will be used later for DFU to update the main program firmware as shown in Figure 20

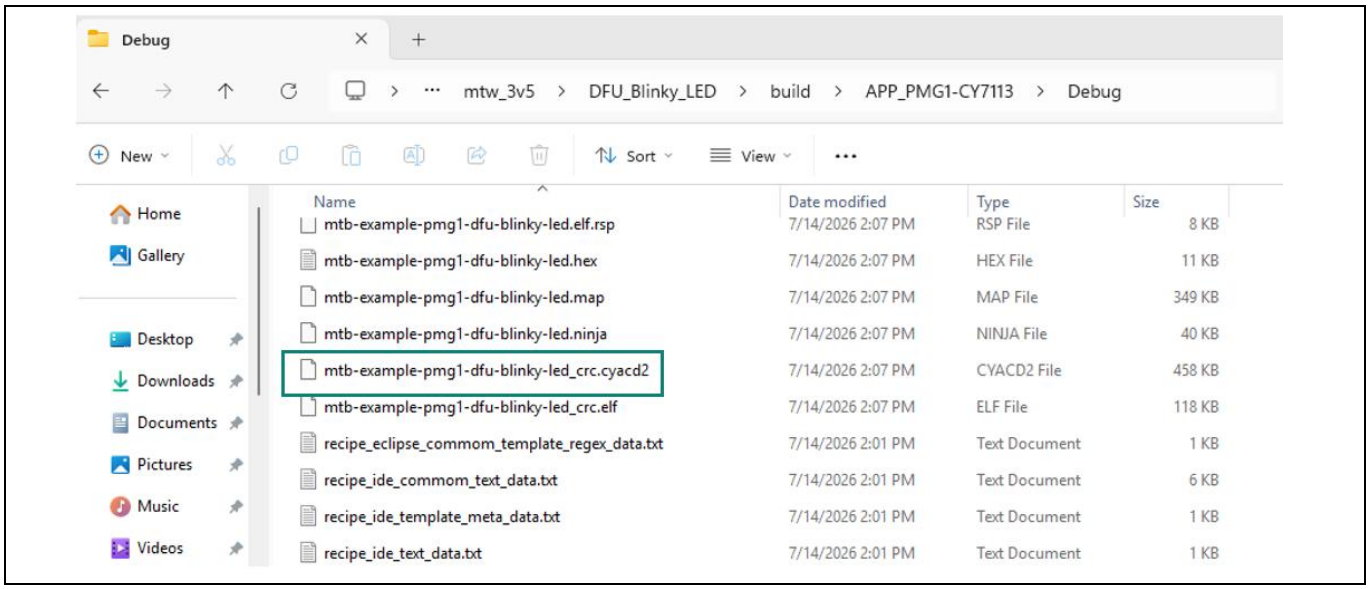


Figure 20 CYACD2 file generated after building the project

4.3 DFU update verification

Before performing the DFU update on the main application, you must first flash the DFU firmware into the EZ-PD™ PMG1-S3 MCU, as this is necessary for the boot process. The jumpers (J5) connect to pins 2 and 3 to supply a 5 V voltage from the USB connector at J1 as shown in Figure 21.

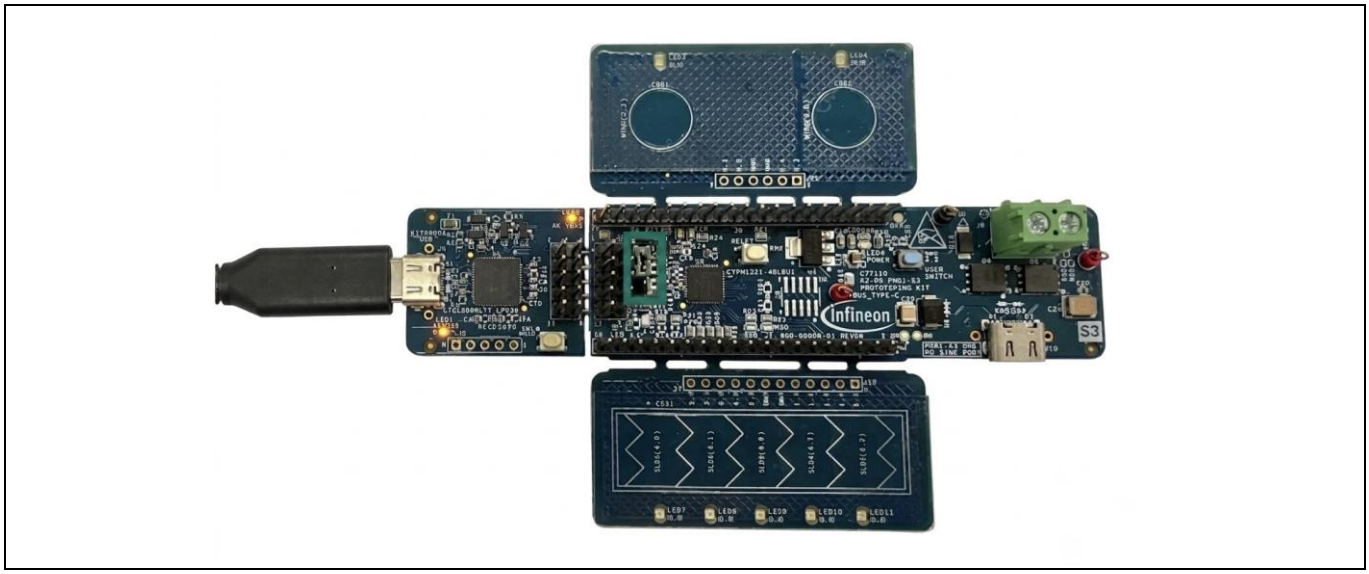


Figure 21 CY7113 EZ-PD™ PMG1-S3 MCU Prototyping Kit set up

4.3.1 Programming DFU_Bootloader application

Use the ModusToolbox™ Programming tool to program the previously built *mtb-example-pmg11-dfu-bootloader.hex* into the target EZ-PD™ PMG1-S3 MCU. The kit includes a built-in programmer that can be used for this programming process, as shown in Figure 22.

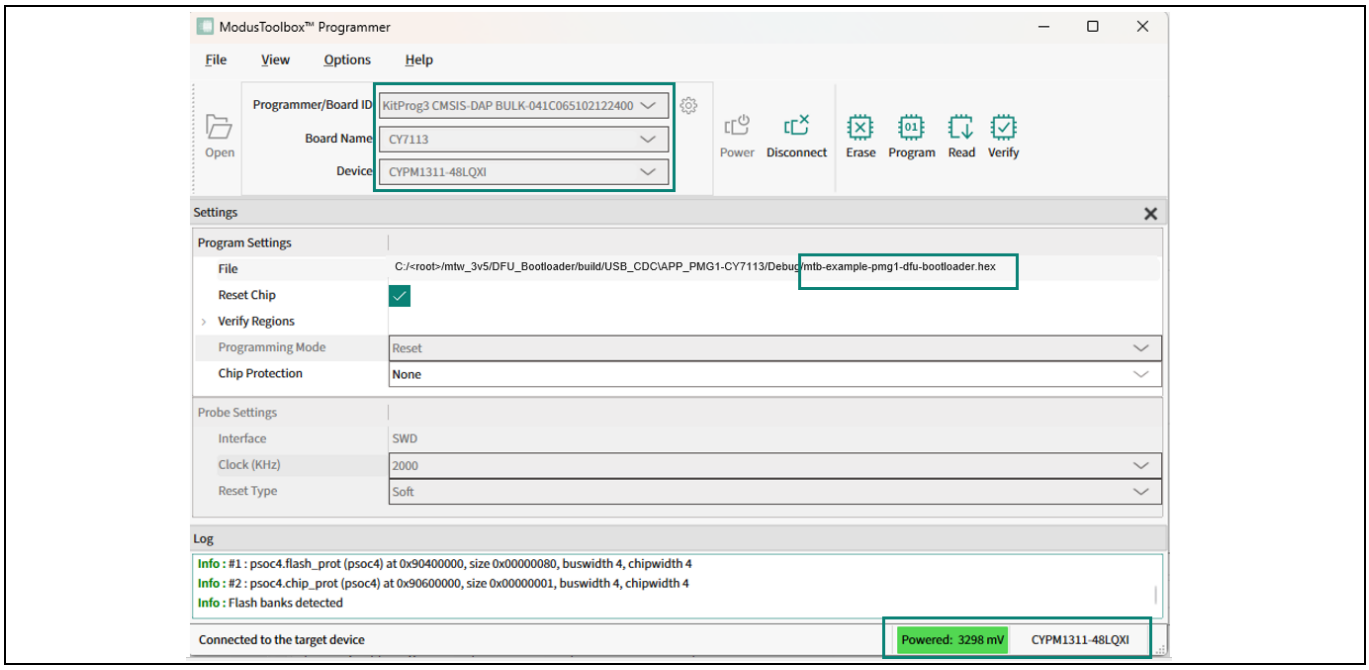


Figure 22 mtb-programmer

After selecting the target kit and the HEX file, click **Program**. The writing process begins and will be complete when the programming progress reaches 100%, as shown in Figure 23.

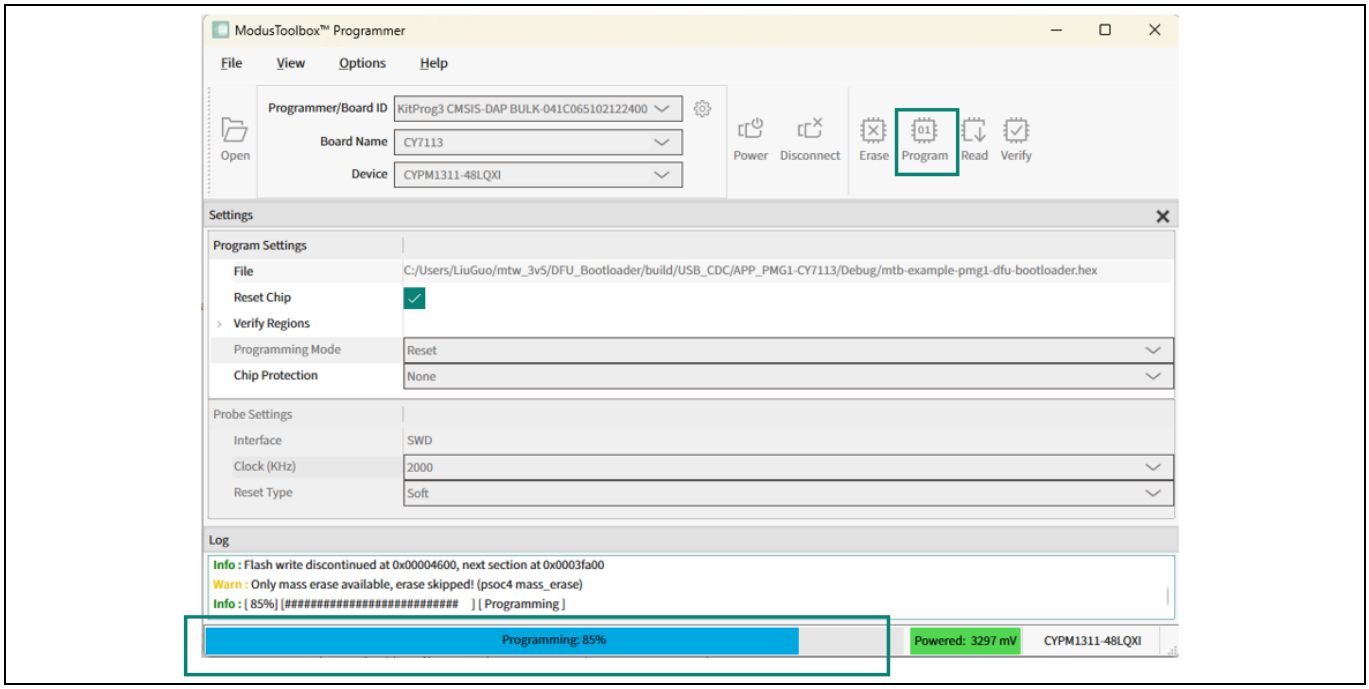


Figure 23 Hex file programming process

As shown in Figure 24, after programming is complete, the tool displays **Device programmed successfully**.

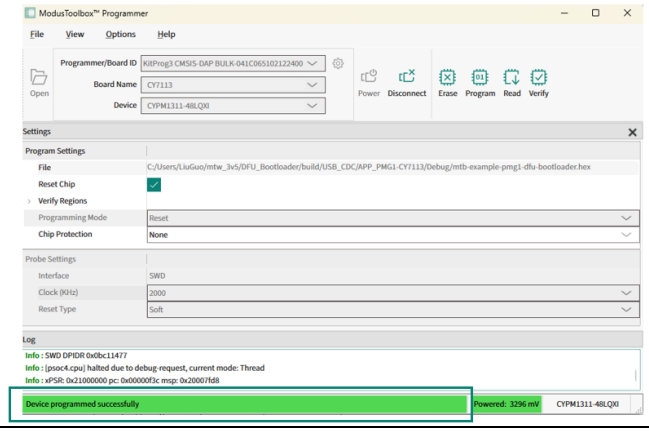


Figure 24 Device programmed successfully

Connect another USB cable to J10 on the kit and open **Device Manager** to check if a **USB-CDC** device COM port is displayed. Do not remove the original USB cable from J1, as it will be used for power as shown in Figure 25 and Figure 26.

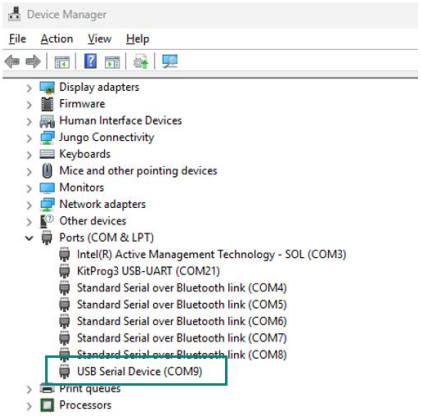


Figure 25 Verify the USB-CDC device in Device Manager

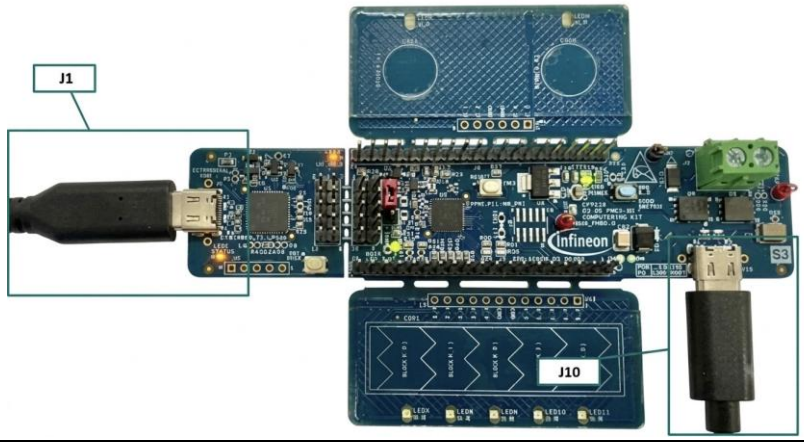


Figure 26 J1 provides power, while J10 is the USB-CDC device

4.3.2 DFU Host Tool

The Device Firmware Update (DFU) Host tool is a stand-alone program included with the ModusToolbox™ software. This tool is used to communicate with EZ-PD™ PMG1, PSOC™ 6, PSOC™ 4, PSOC™ Edge, PSOC™ Control, and XMC™ 7000 MCUs that have already been programmed with an application that includes the DFU capability.

The DFU Host Tool is installed along with ModusToolbox™. You can launch the DFU by selecting "Device Firmware Update Host Tool" from the Tools list in the Quick Start menu, as shown in Figure 27 and Figure 28.

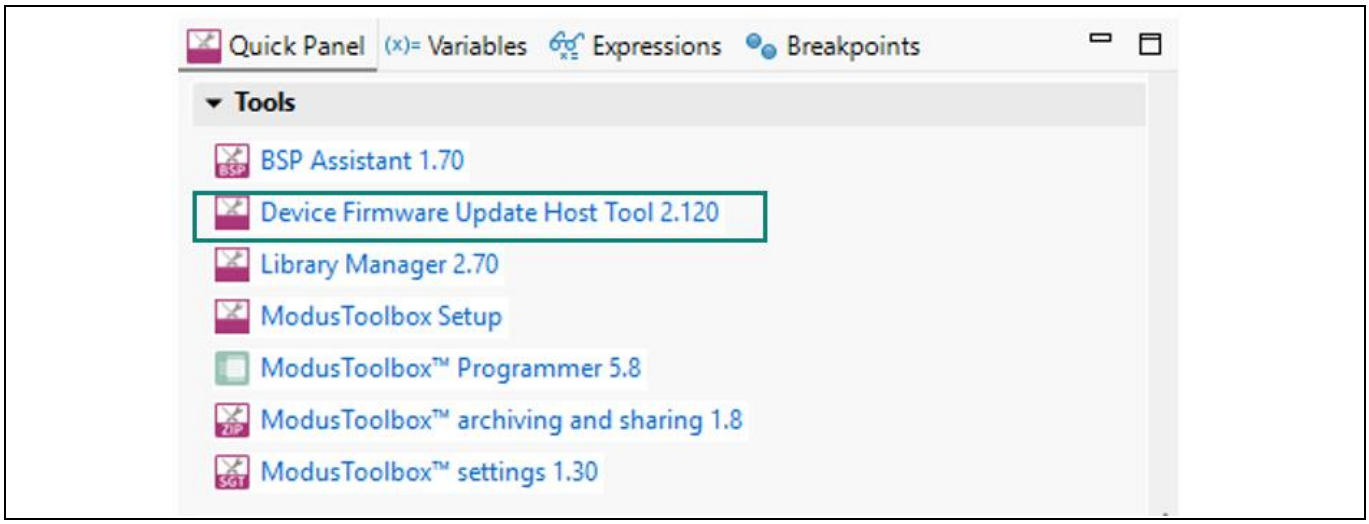


Figure 27 Device Firmware Update Host Tool

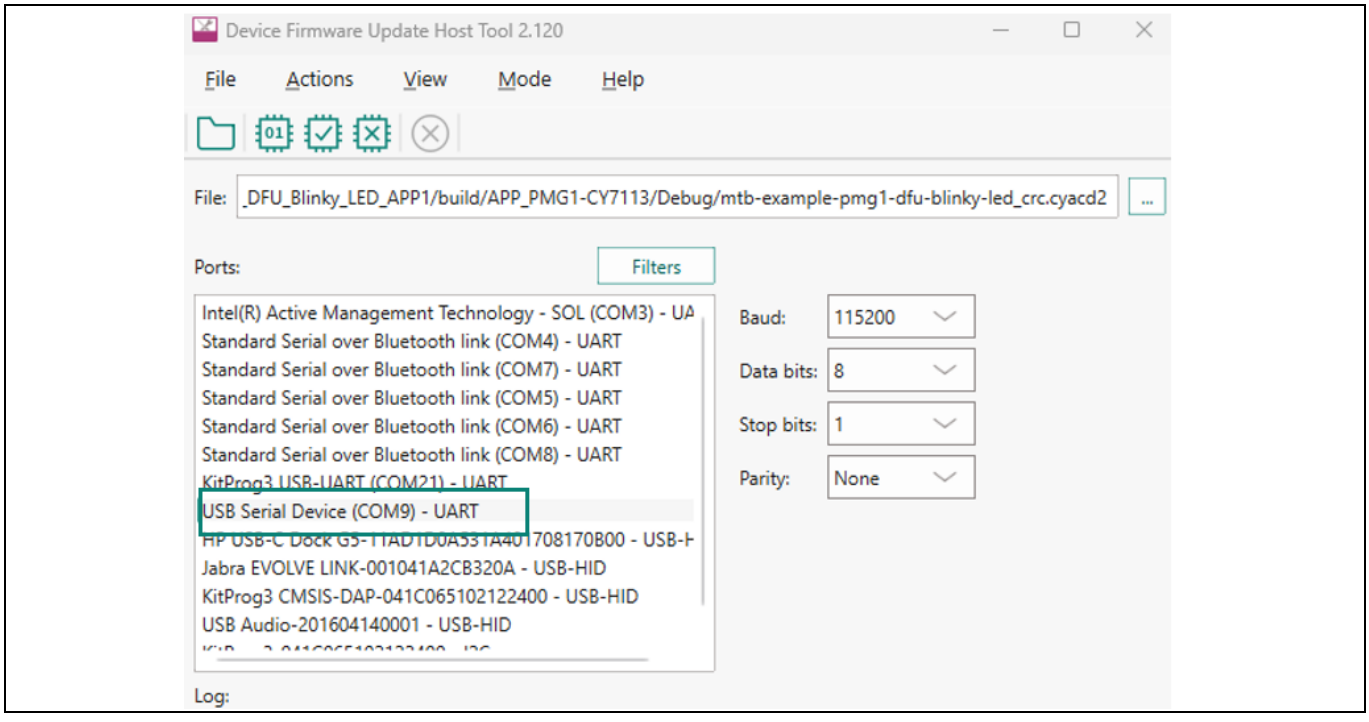


Figure 28 Device Firmware Update Host Tool displays USB-CDC device

4.3.3 Update the main application using the DFU Host Tool

If only the DFU_Bootloader firmware is present on the **PMG1-CY7113 Prototyping Kit**, the **LED3** on the kit will blink at a frequency of 1 Hz, as shown in Figure 29. When the DFU writes the APP program into the EZ-PD™ PMG1-S3 MCU, the LED's blinking frequency increases to 10 Hz.

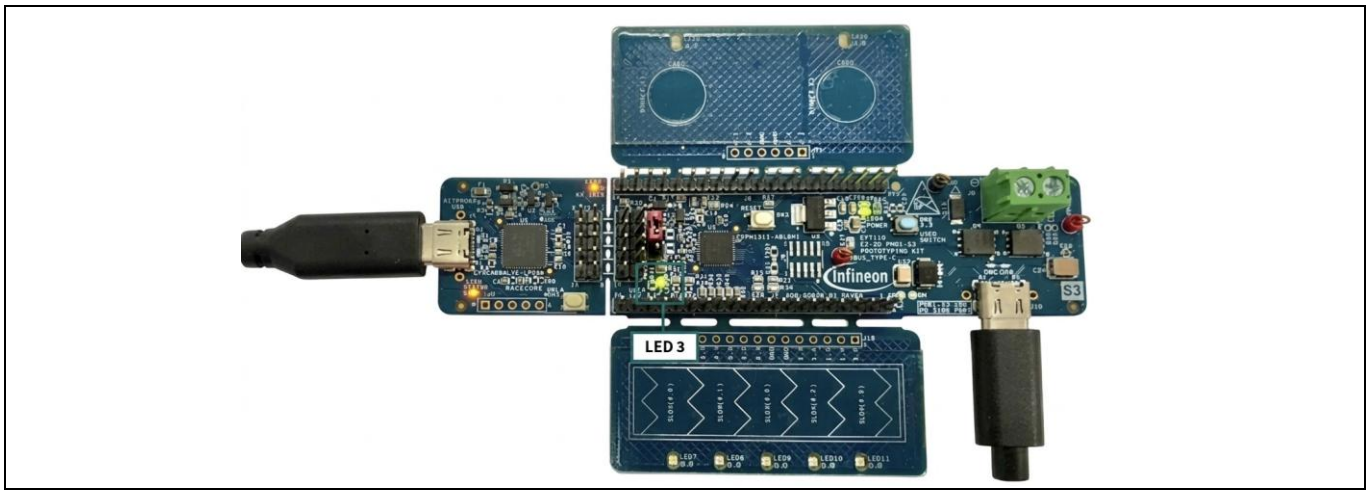


Figure 29 EZ-PD™ PMG1-CY7113 Prototyping Kit LED3 blinking status indicator

Select the previously built *DFU_Blinky_LED CYACD2* file, *mtb-example-pmg1-dfu-blinky-led_crc.cyacd2*, and click on **Program**. Wait until the progress reaches 100% and the program inside the EZ-PD™ PMG1-S3 MCU will transition from the bootloader to the blink LED program, causing the blinking frequency of LED3 to increase to 10 Hz, as shown in Figure 30.

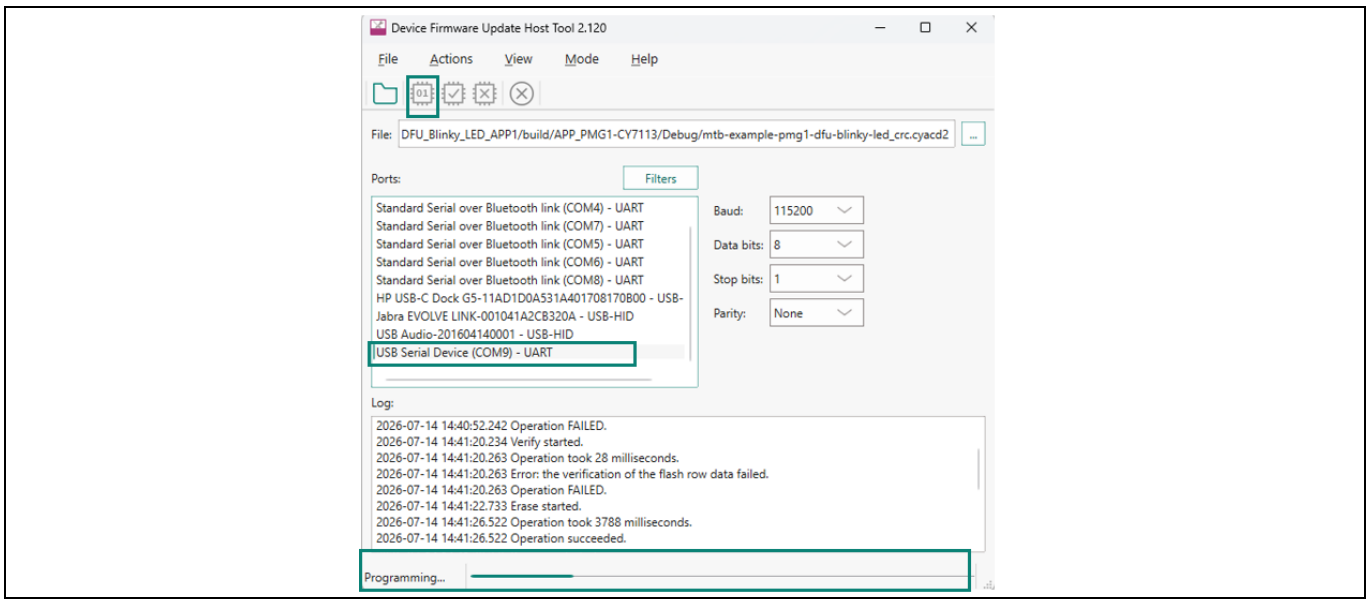


Figure 30 DFU Host Tool update process

5 Dual image bootloader use cases

- In some battery backup units (BBUs) or other systems, the firmware is designed to ensure proper MCU operation by dividing the internal flash memory into multiple segments. In the event of a failure during the update process, it can revert to the originally functioning application execution block
- Firmware update

  The bootloader divides the flash memory into two sections for each application: a primary slot and a secondary slot. The initial version of the application is stored in the primary slot. When an update is available, the device's firmware update application receives the upgrade image over a wired (Device Firmware Upgrade, or DFU) or wireless (Over-the-Air, or OTA) communication interface and saves it to the secondary slot. The bootloader always starts from the primary slot, but if an update is requested, it copies the new version from the secondary slot to the primary slot. The system can use the bootloader to check during the boot process which version is the latest between the primary and secondary slots and select the appropriate one to boot. Generally, after a firmware update, the version number can be written to an external EEPROM if the system is designed to support this, or a small portion of the internal flash memory of the MCU can be reserved to store system information.

6 EZ-PD™ PMG1-S3 MCU flash structure

These devices feature an Arm® Cortex®-M0/M0+ CPU and a USB-C PD controller, along with analog and digital peripherals. EZ-PD™ PMG1 is targeted for any embedded system that provides/consumes power to/from a high-voltage USB-C PD port and leverages the microcontroller to provide additional control capability. Figure 31 shows the EZ-PD™ PMG1 MCU family segmentation. The EZ-PD™ PMG1-S3 MCU has two models: Single port PD and dual port PD functionality, both of which feature a 256 KB flash memory size.

- **Flash memory size:** The EZ-PD™ PMG1-S3 MCU has a 256 KB (2x 128 KB) flash module
- **SRAM memory size:** 32 KB of SRAM, which is retained during Deep Sleep is provided
- **ROM size:** 96 KB of supervisory ROM that contains boot and configuration routines is provided. In addition to the flash erase and program routines provided for the EZ-PD™ PMG1-S3 MCU, the SROM also contains flash checksum routines

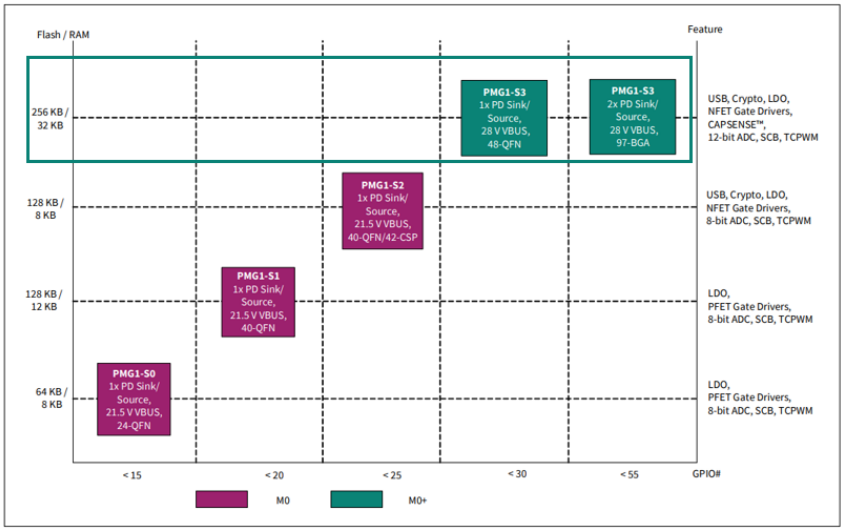


Figure 31 EZ-PD™ PMG1 MCU family segmentation

The EZ-PD™ PMG1-S3 MCU memory map is detailed in the following tables. For additional information, see the EZ-PD™ PMG1-S3 MCU registers reference manual. The EZ-PD™ PMG1-S3 MCU features a fixed address map, enabling access to memory and peripherals through simple memory access instructions. The address space is divided into the regions shown in Table 2.

Table 2 EZ-PD™ PMG1-S3 MCU address map

Address range	Use
0x00000000-0x0003FFFF	256 KB flash
0x0FFFF000-0x0FFFF3FF	1 KB supervisory flash
0x10000000-0x10001FFF	8 KB Boot SROM (no direct read/execute access)
0x10002000-0x10017FFF	88 KB User SROM (read/execute access allowed)
0x20000000-0x20007FFF	32 KB SRAM

7 Dual image bootloader features

Dual-image operation involves updating and verifying more than one image on a device. Consider a bootloader with dual-image support. The operation is as follows:

1. The DFU bootloader first checks which image version is newer between the primary and secondary images. It selects either the primary or secondary image to boot based on the selection criteria
2. During firmware updates, only the primary or secondary space is updated. Only the specified image space needs to be erased. After the update, the version can be recorded for later reference as a selection criterion during activation
3. After the update is complete, the MCU can decide when to perform the reboot action
4. When developing the application code, configure the flash address to specify whether the image is for the primary or secondary slot. Therefore, there is no need to specifically designate the location during the firmware update in the DFU Host Tool
5. In the boot selection, the internal flash space can be partitioned to record the firmware versions stored in primary and secondary. When the bootloader needs to enter the application, it reads the version number and boots into the latest version; alternatively, it can use a switch to control the boot selection

7 Dual image bootloader features

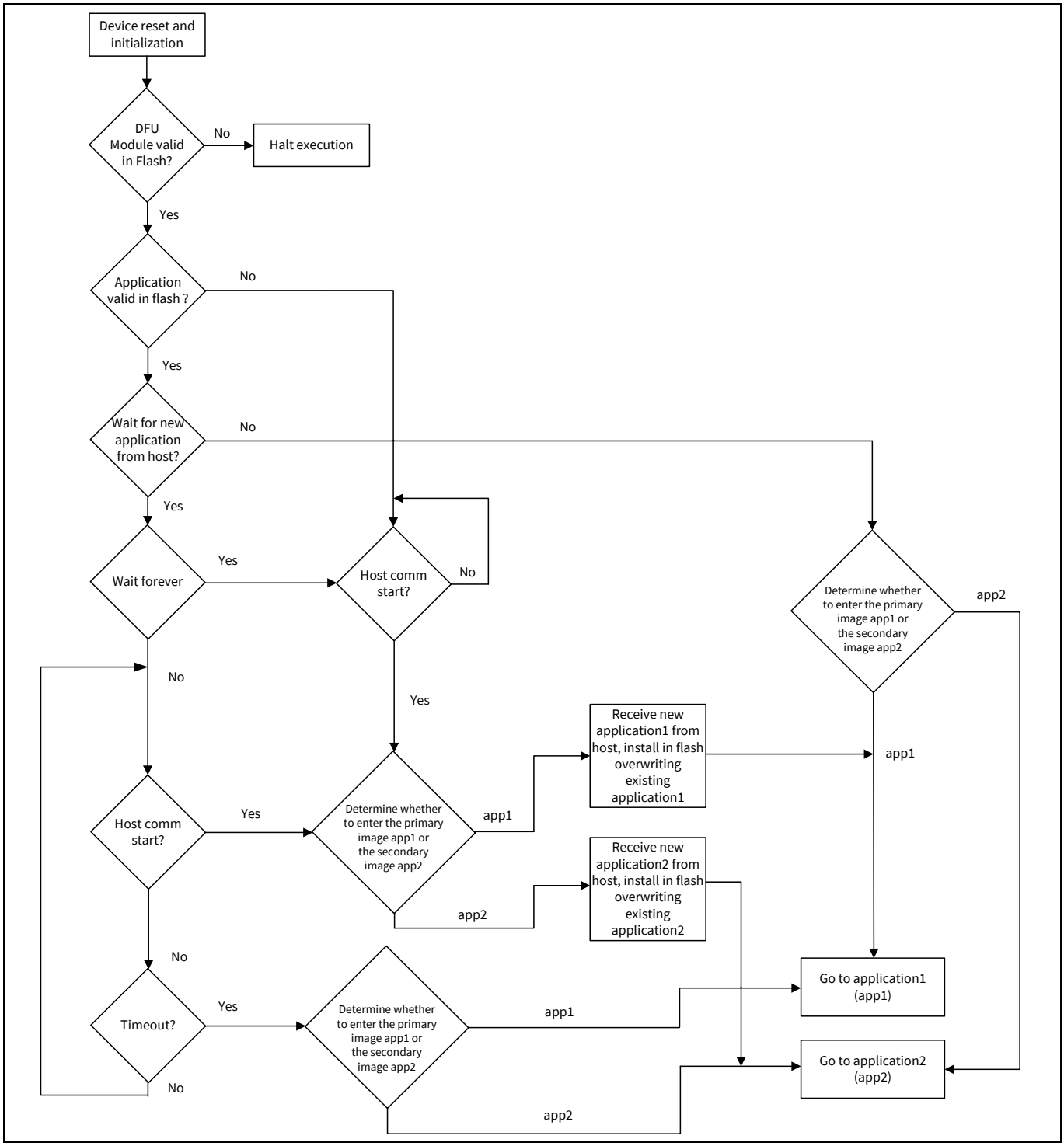


Figure 32 Dual Image boot and update flow

8 Modify DFU bootloader in ModusToolbox™ for dual image

The EZ-PD™ PMG1-S3 MCU has 256 KB of internal flash, which will be divided into three blocks.

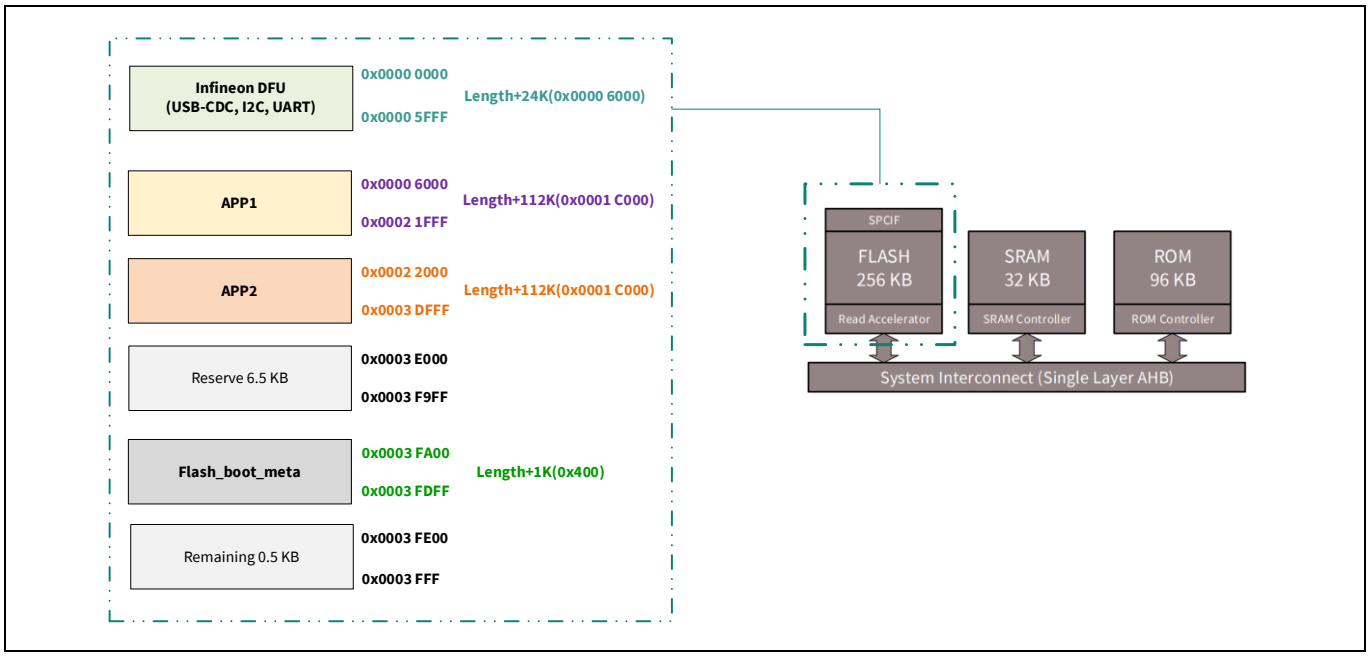


Figure 33 EZ-PD™ PMG1-S3 MCU memory configuration

8.1 Modify the sample code of the DFU bootloader

Modify the **DFU_Bootloader** example within ModusToolbox™ and use the **DFU_Blinky_LED** example to generate separate APP1 and APP2 image files, which will be updated separately using the DFU Host Tool to achieve individual firmware updates, as shown in Figure 34 and Figure 35.

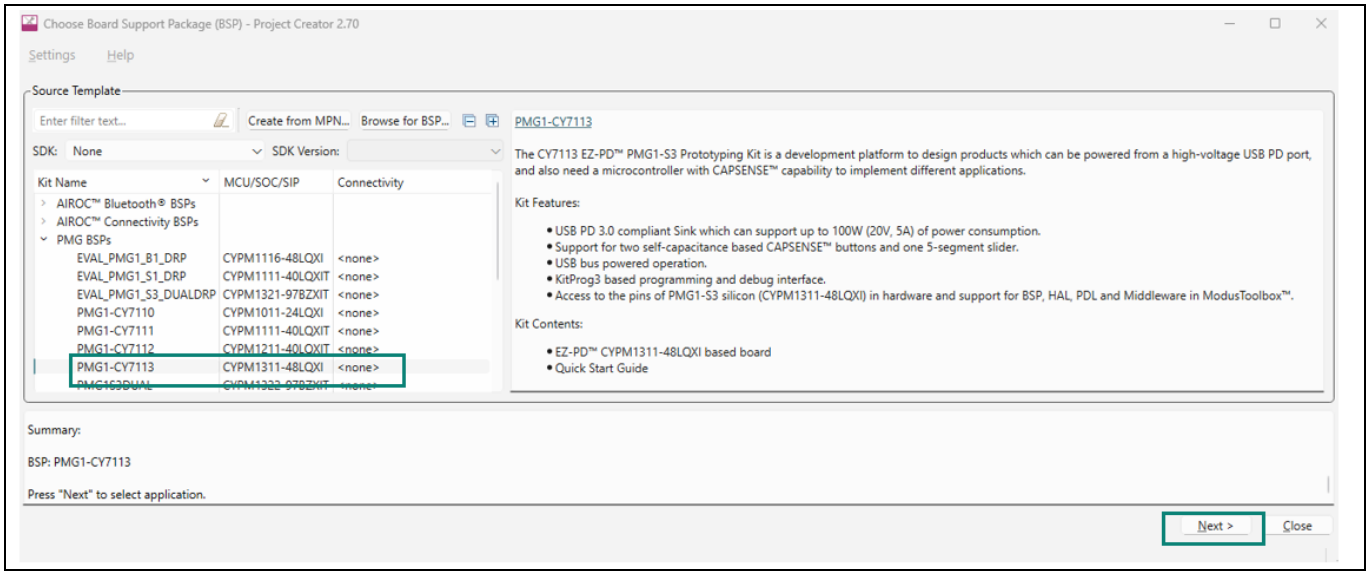


Figure 34 Choose the target kit PMG1-CY7113

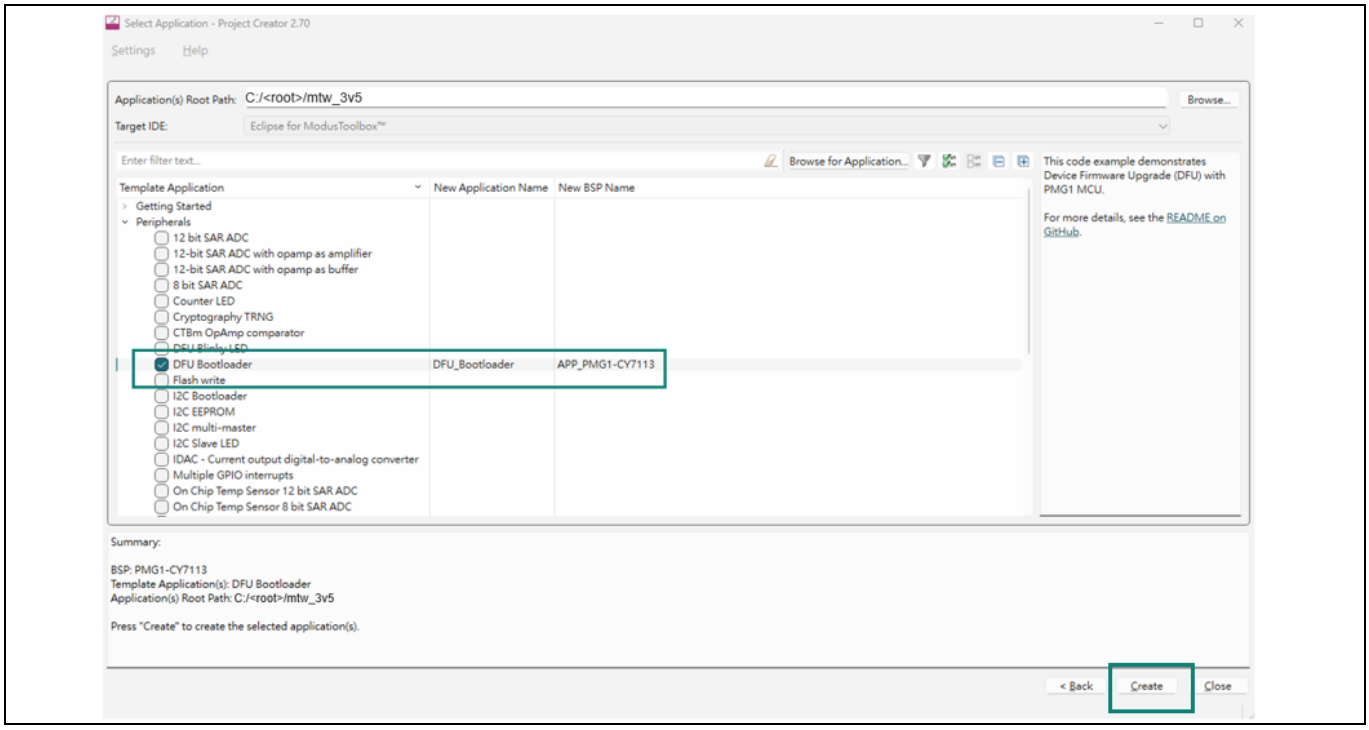


Figure 35 Choose the DFU bootloader application

8.1.1 EZ-PD™ PMG1-S3 MCU internal flash size allocation

The *dfu_cm0p_app0.ld* file is located in **bsps** > **TARGET_APP_PMG1CY7113** > **COMPONENT_CM0P** > **TOOLCHAIN_GCC_ARM** for internal flash memory in the Project Explorer as shown in Figure 36.

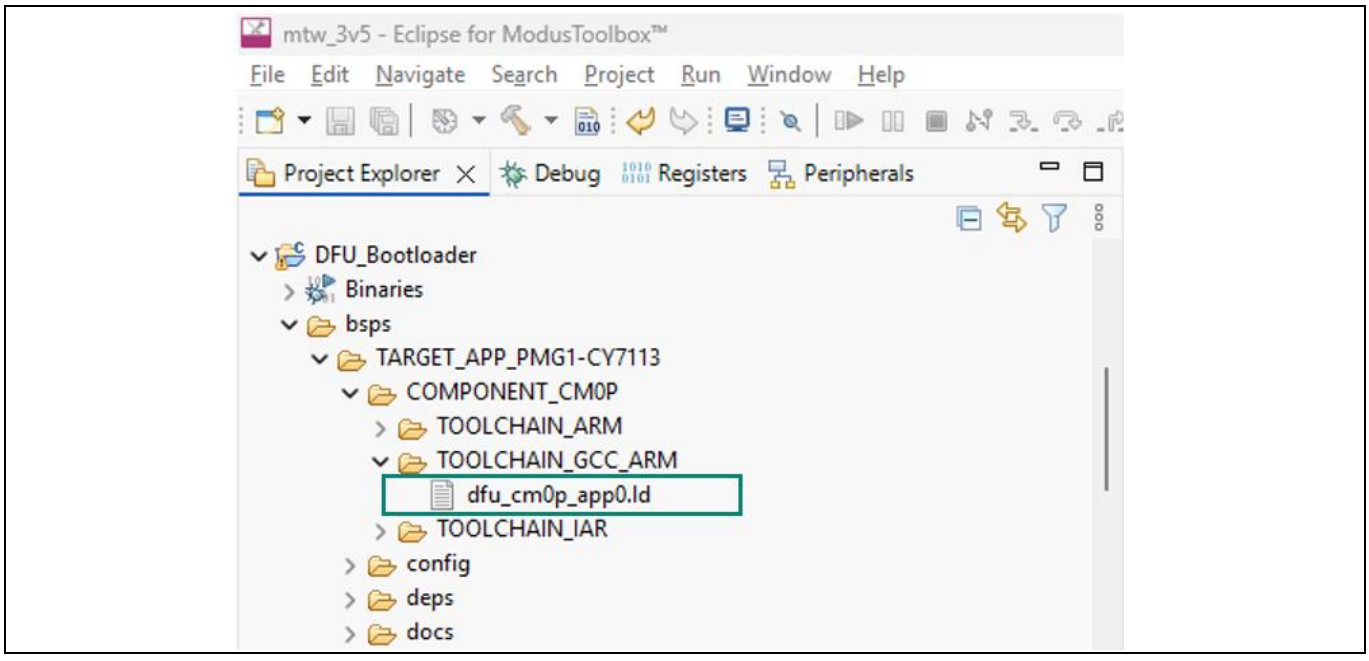


Figure 36 ModusToolbox™ Project Explorer

Divide the internal flash memory into four sections for use by the DFU bootloader, APP1, APP2, and metadata. Flash memory is non-volatile, meaning the data written to it persists. Edit the flash and RAM memory region

sizes in **dfu_cm0p_app0.ld**. The RAM section does not need to be divided separately; the individual names are used for easier management when configuring the build project later, as shown in Figure 37.

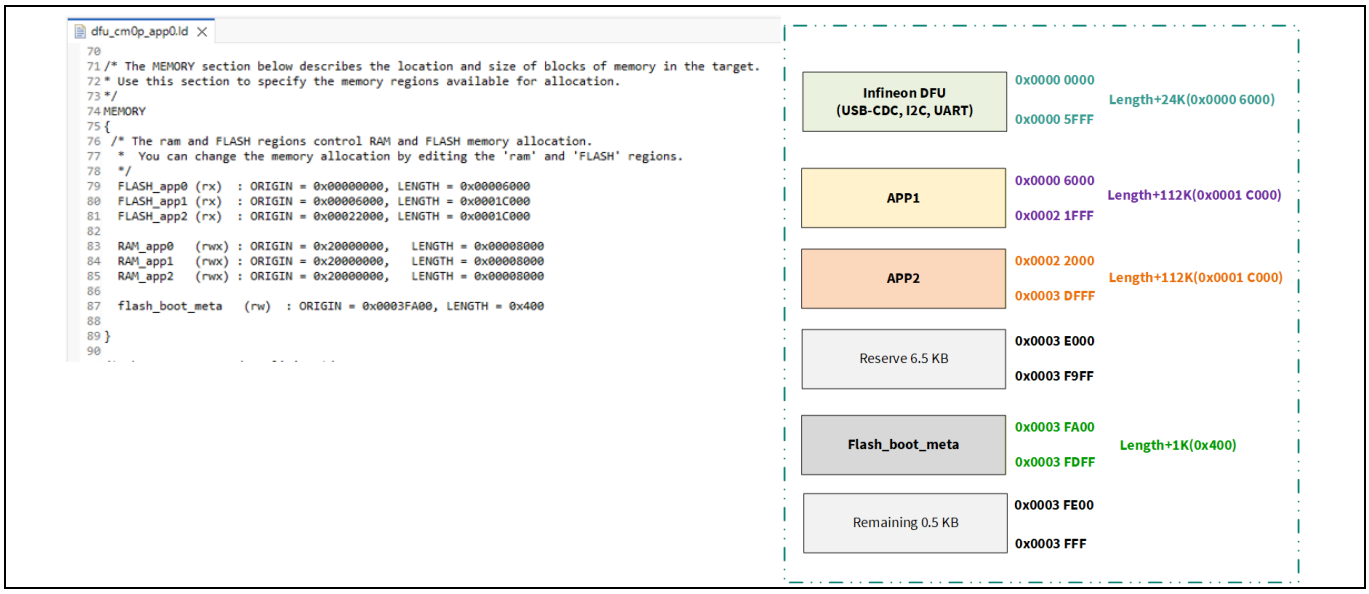


Figure 37 dfu_cm0p_app0.ld Flash and RAM allocation

Set the starting address of the application and the length of the flash address used. Assign app0 to the bootloader, APP1 to Application 1, and APP2 to Application 2, specifying them individually, as shown in Code Listing 1.

Code Listing 1 dfu_cm0p_app0.ld by application to set the metadata

```
/* Used by the DFU SDK application to set the metadata */
__cy_app0_verify_start  = ORIGIN(FLASH_app0);
__cy_app0_verify_length = LENGTH(FLASH_app0) - __cy_boot_signature_size;
__cy_app1_verify_start  = ORIGIN(FLASH_app1);
__cy_app1_verify_length = LENGTH(FLASH_app1) - __cy_boot_signature_size;
__cy_app2_verify_start  = ORIGIN(FLASH_app2);
__cy_app2_verify_length = LENGTH(FLASH_app2) - __cy_boot_signature_size;
```

This setting defines the range for the current DFU_Bootloader project. The configuration of **__cy_app_id** specifies **0** for the app0 bootloader, 1 for APP1, and 2 for APP2. These configurations must be consistent across the project's application settings, as shown in Code Listing 3.

Code Listing 2 dfu_cm0p_app0.ld aliases regions

```
/*
* DFU SDK specific: aliases regions, so the rest of code does not use
* application specific memory region names
*/
REGION_ALIAS("FLASH", FLASH_app0);
REGION_ALIAS("RAM",     RAM_app0);
```

```
/* DFU SDK specific: sets an app Id */
__cy_app_id = 0;
```

8.1.2 Configure the EZ-PD™ PMG1-S3 MCU

The pins used in this application for the CY7113 EZ-PD™ PMG1-S3 MCU Prototyping Kit are referenced in Figure 38 and Table 3. The DFU bootloader supports update interfaces including I2C, USB-CDC, and UART, with communication to the host PC primarily using USB-CDC.

Table 3 EZ-PD™ PMG1-S3 MCU Prototyping Kit pin assignment

Pins	Function
P1[1]	CYBSP_SWDCLK
P1[2]	CYBSP_SWDIO
P1[5]	CYBSP_USER_SELECT
P3[3]	CYBSP_USER_BTN
P3[5]	CYBSP_UART_RX
P3[6]	CYBSP_UART_TX
P4[0]	CYBSP_I2C_SCL
P4[1]	CYBSP_I2C_SDA
P5[5]	CYBSP_USER_LED
P8[0]	USBDP
P8[1]	USBDM

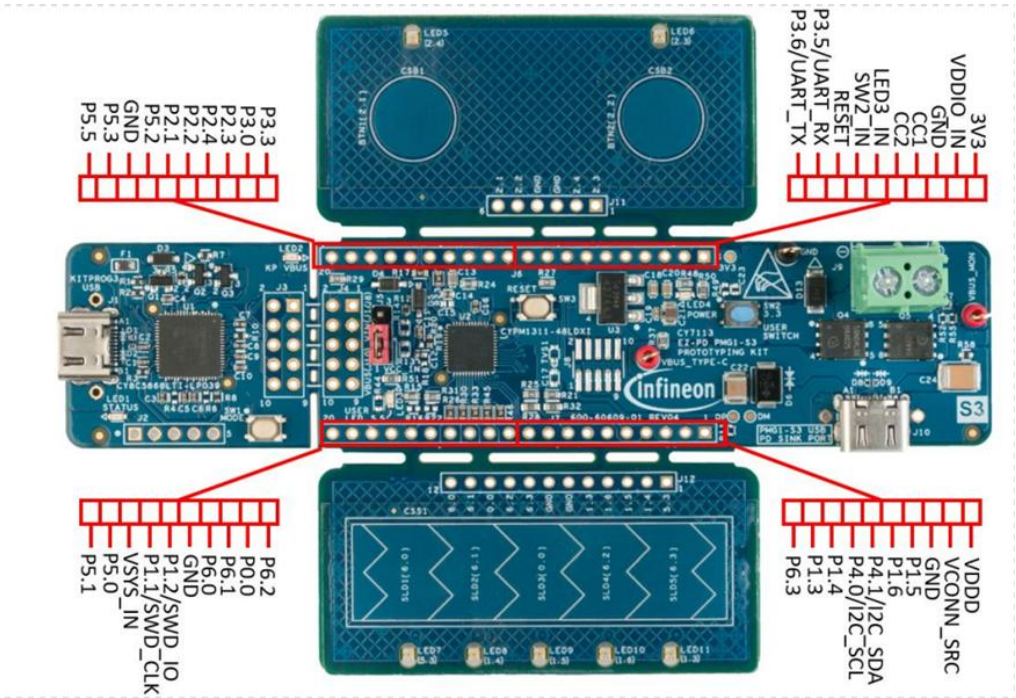


Figure 38 CY7113 EZ-PD™ PMG1-S3 MCU Prototyping Kit pin definitions

- In **Device Configurator** > **Peripherals** > **Communication** > **SCB Block** > **Personality** > **I2C**: Configure I2C to support image updates via I2C. The I2C update interface can be provided to connect other processors with the EZ-PD™ PMG1-S3 MCU for updates, as shown in Figure 39. The I2C pin definitions are shown in Figure 40

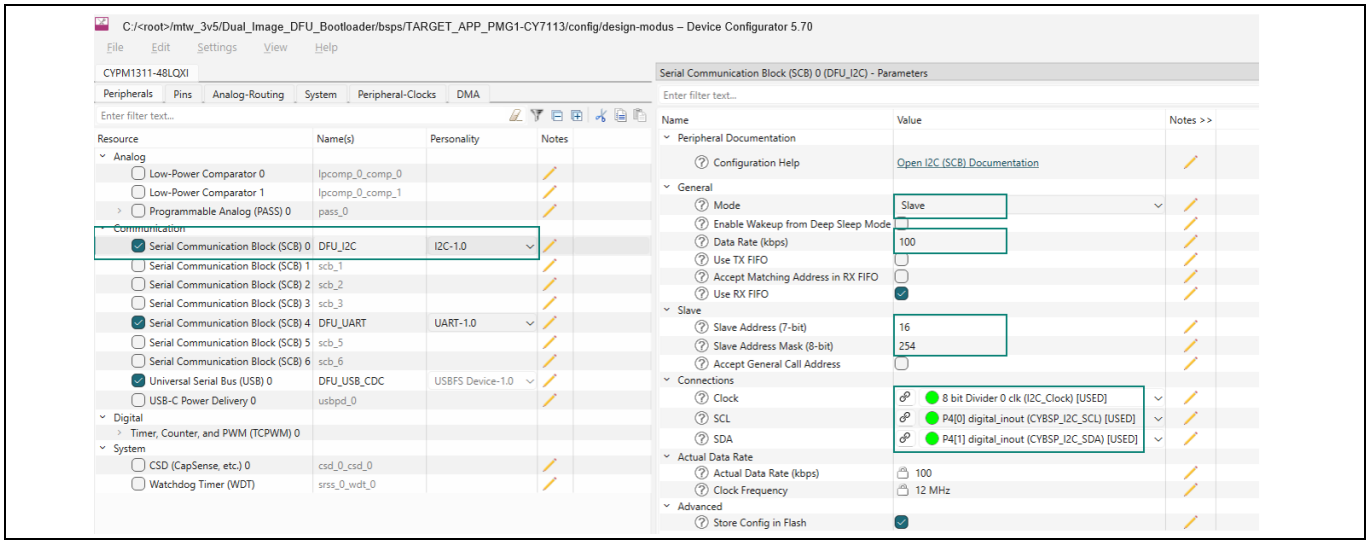


Figure 39 I2C configuration



Figure 40 Pin definitions for I2C

- In **Device Configurator** > **Peripherals** > **UART**: Configuring the UART allows other SoCs or processors to update the EZ-PD™ PMG1-S3 MCU, and it can also be used for updates from the host PC, as shown in Figure 41. The UART pin definitions are shown in Figure 42

8 Modify DFU bootloader in ModusToolbox™ for dual image

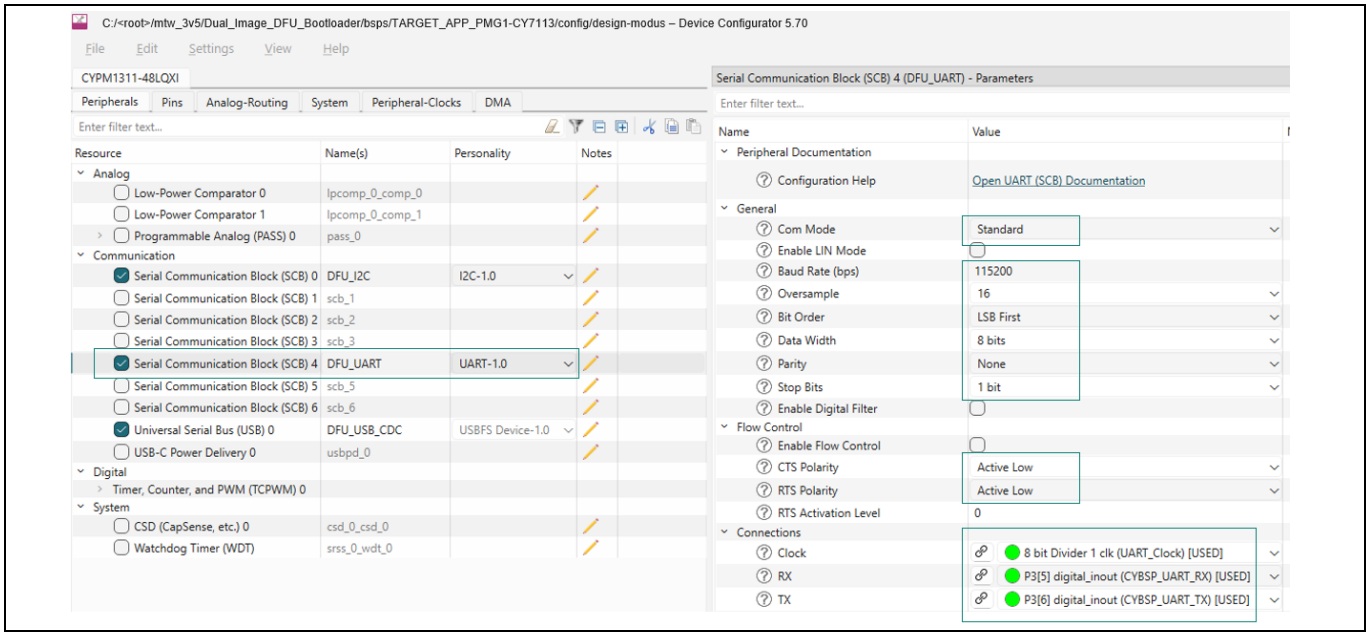


Figure 41 UART configuration

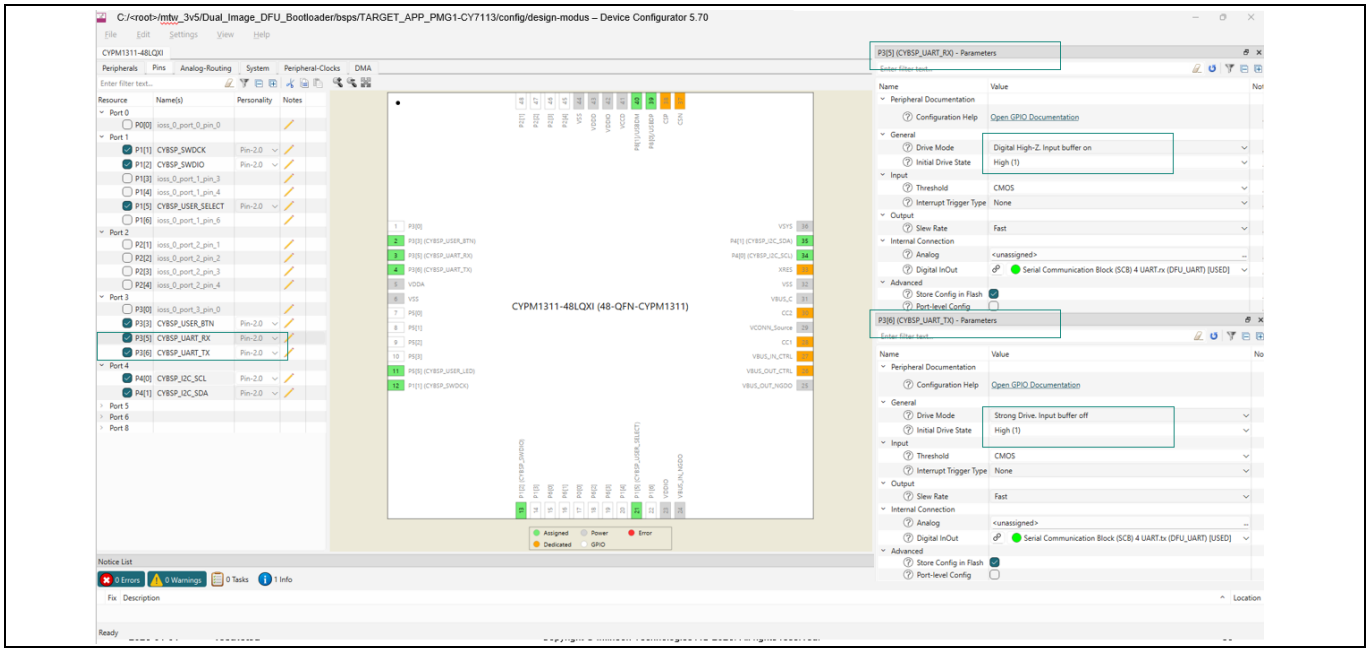


Figure 42 Pin definitions for UART

- In **Device Configurator** > **Peripherals** >**USB-CDC**: The USB-CDC here utilizes the USB FS functionality within the MCU. After powering on the MCU, a USB-CDC device can be enumerated to work with the DFU Host Tool for image updates, as shown in Figure 43. The USB-CDC pin definitions are shown in Figure 44

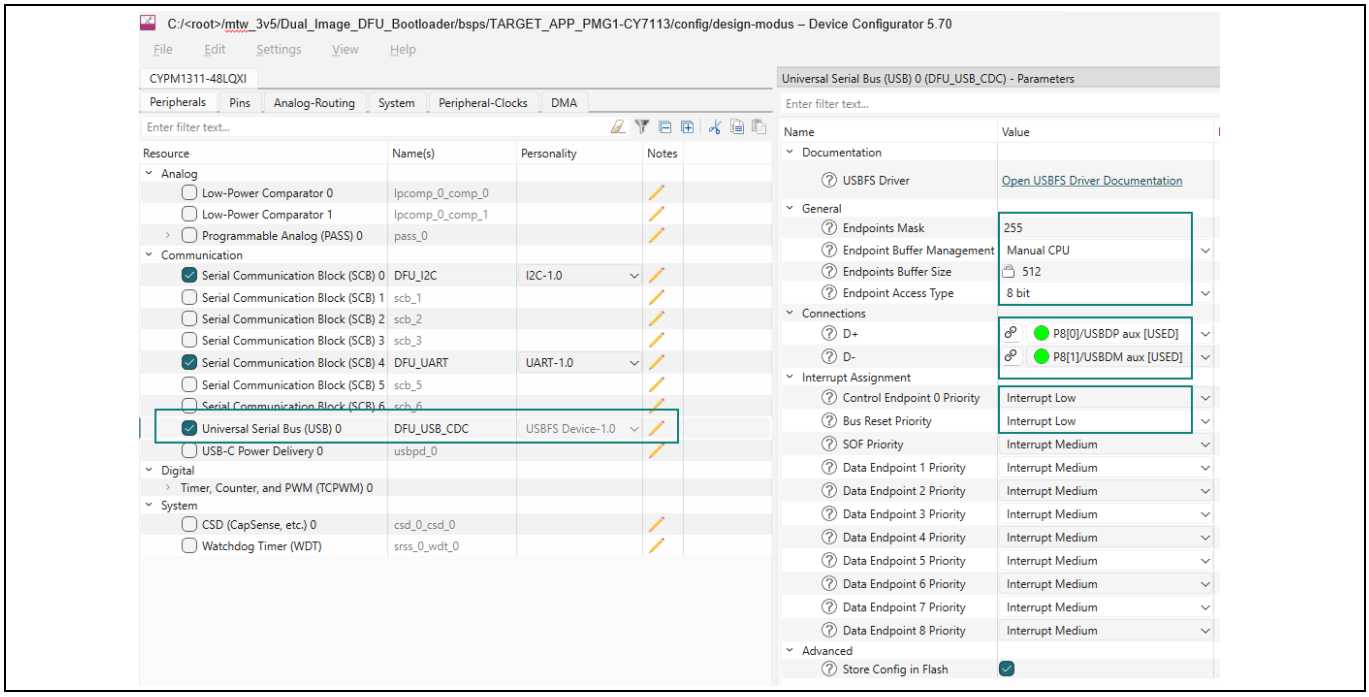


Figure 43 USB-CDC configuration

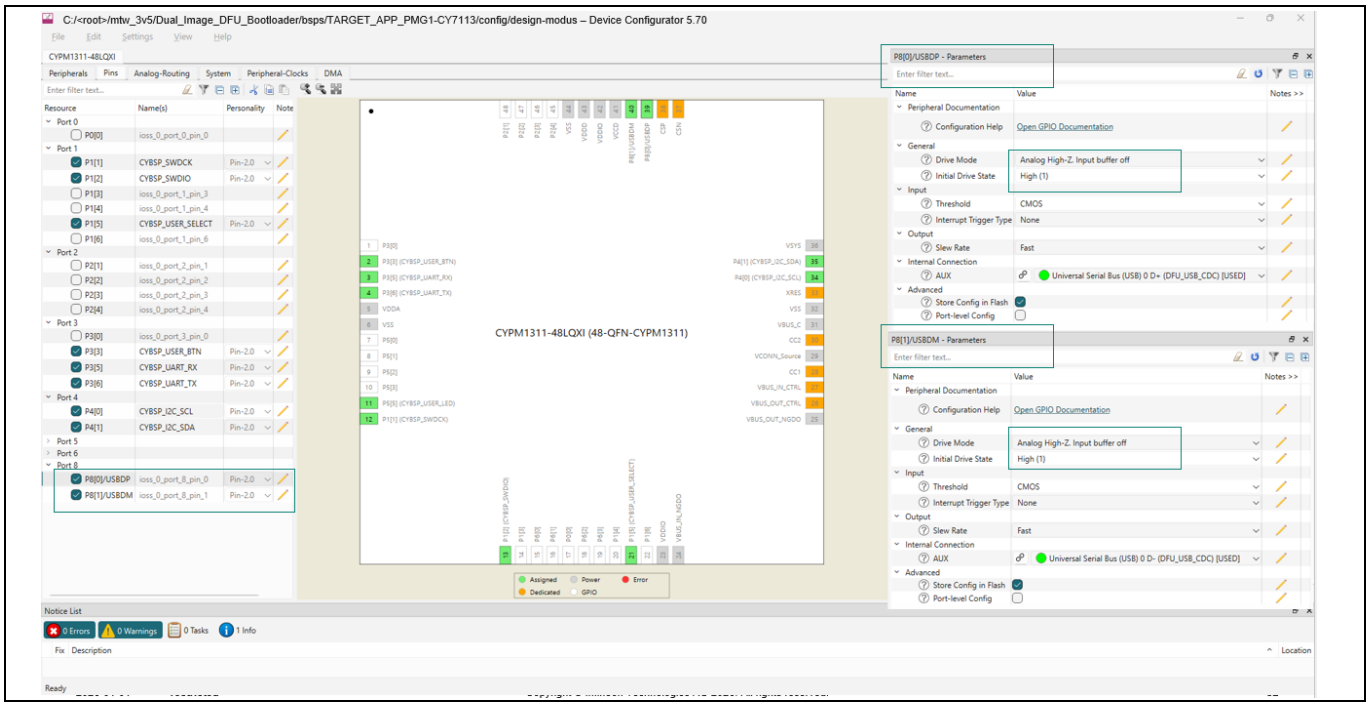


Figure 44 Pin definitions for USB-CDC

- In **Device Configurator** > **Peripherals** >**TIMER**: Configuring the timer functionality allows for adjusting the blinking frequency of the status indicator LED, as shown in Figure 45

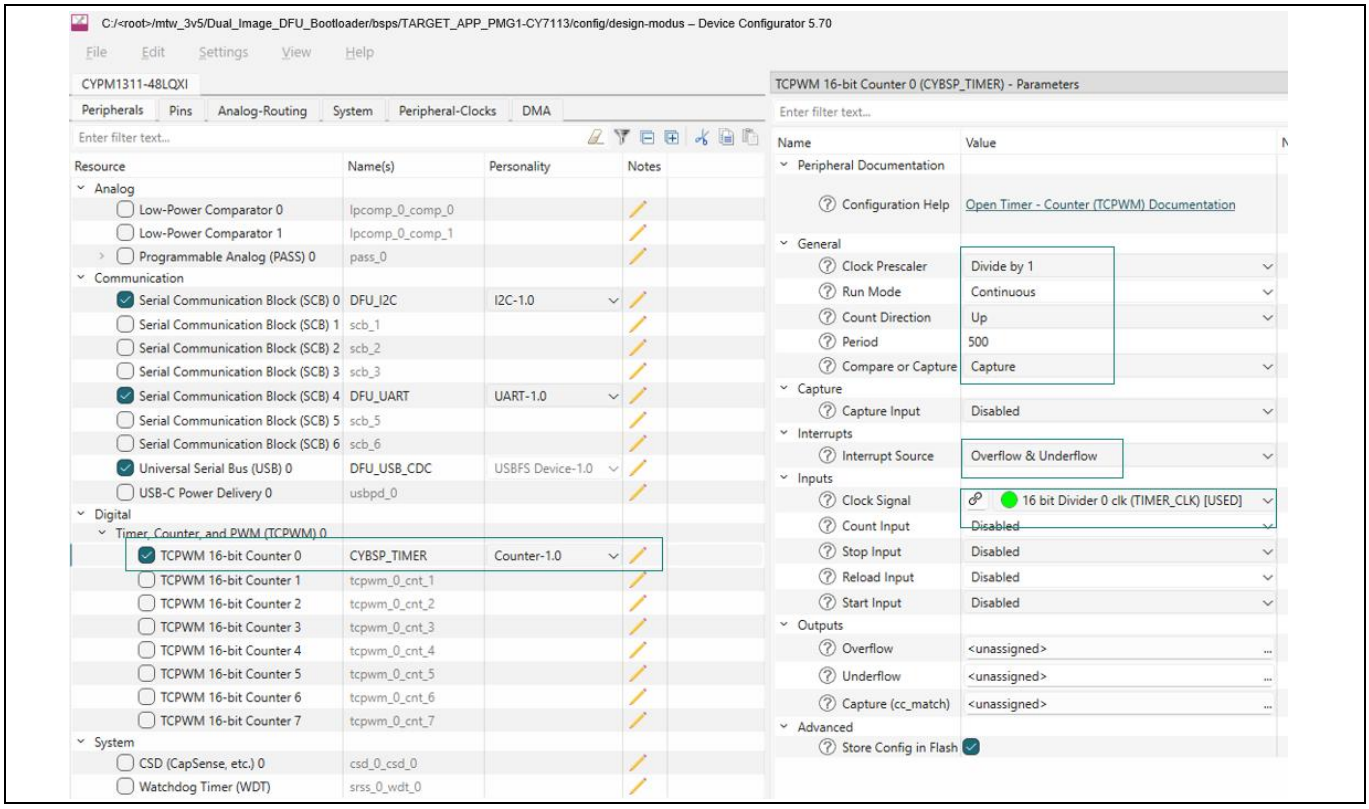


Figure 45 TIMER Configuration Device Configurator PIN Define-SWD

- In **Device Configurator** > **Pin define** >**SWD**: The SWDCK/SWDIO pins are used to enable the mini programmer to program the MCU and they only need to be configured as shown in Figure 46

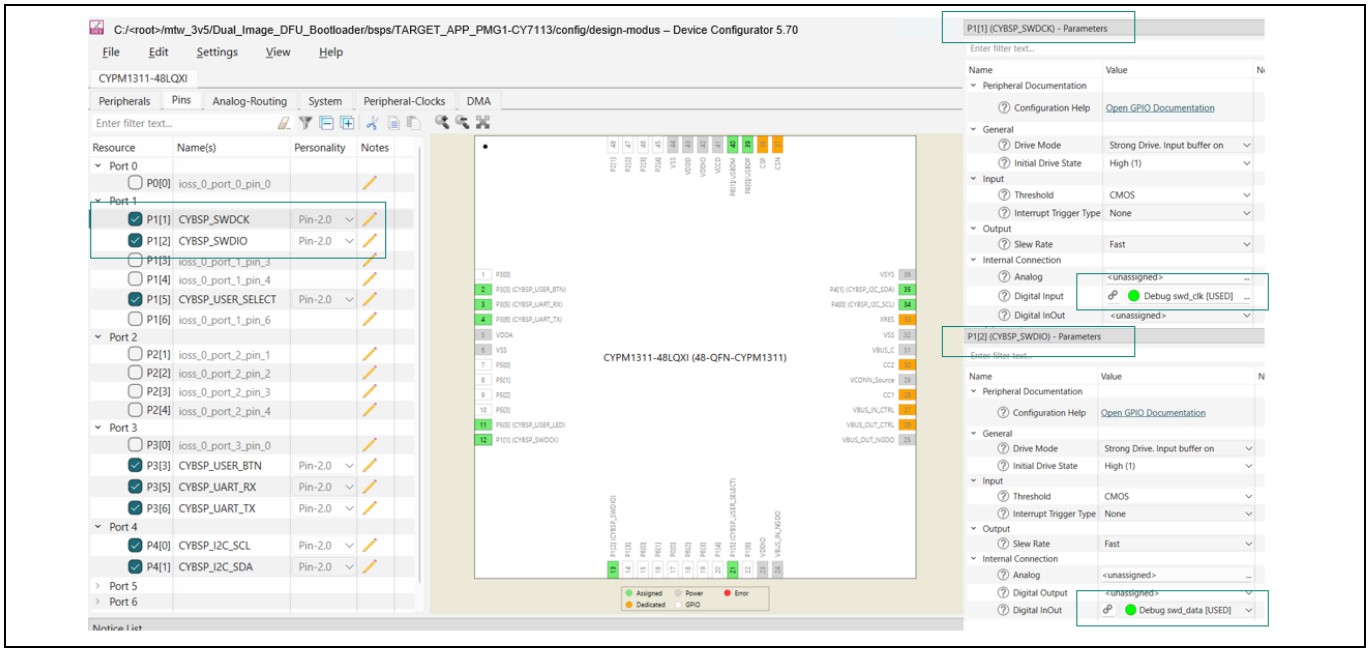


Figure 46 Pin define SWD interface

- In **Device Configurator** > **Pin define** > **GPIO**: Three functional GPIO pins will be configured and pin assignments will be provided as shown in Figure 47, Figure 48, and Figure 49

CYBSP_USER_SELECT: Used to select whether to update the primary image or secondary image and whether the DFU bootloader boots from the primary image or secondary image during startup

- **CYBSP_USER_EN:** This pin is configured as an input and has interrupt triggering functionality
- **CYBSP_USER_LED:** This pin is set as an output to control the blinking status of the LED



Figure 47 GPIO CYBSP_USER_SELECT configuration

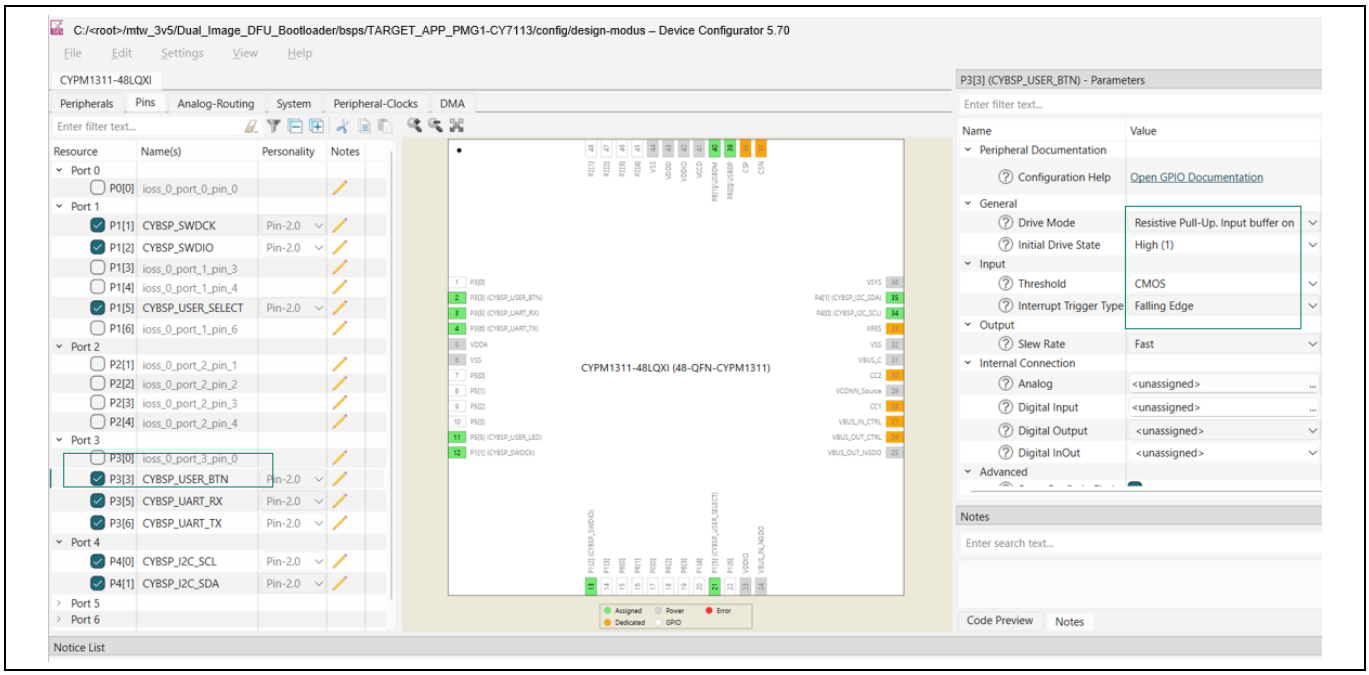


Figure 48 GPIO CYBSP_USER_EN configuration



Figure 49 GPIO CYBSP_USER_LED configuration

8.1.3 Modify the DFU_Bootloader code

Throughout the entire project code, certain sections need to be modified to enable the DFU_Bootloader to switch between the primary and secondary image files.

1. Modify *dfu_user.h*/*dfu_user.c* file: Modify **CY_DFU_MAX_APPS** from 2U to 3U in the *dfu_user.h* file. The purpose is for the DFU bootloader to recognize the firmware stored in the three separate areas allocated in flash memory, as shown in Code Listing 3

Code Listing 3 Modify CY_DFU_MAX_APP Macro at dfu_user.h

```
/**
* The number of applications in the metadata,
* for 512 bytes in a flash row - 63 is the maximum possible value,
* because 4 bytes are reserved for the entire metadata CRC.
*
* The smallest metadata size if CY_DFU_MAX_APPS * 8 (bytes per one app) + 4
(bytes for CRC-32C)
*/
#define CY_DFU_MAX_APPS     (3U)   /*modify from (2U) to (3U)*/
```

2. Add a macro to define flash memory address and length to *dfu_user.h* and modify metadata function at *dfu_user.c*, as shown in Code Listing 4 and Code Listing 5

Code Listing 4 Add a macro to define flash memory address and length in dfu_user.h

```
/* group_dfu_macro_config */
```

```
/**DFU Bootloader Flash start and length**/
 extern uint8_t __cy_app0_verify_start;
 extern uint8_t __cy_app0_verify_length;
 /**Primary Image App1 Flash start and length**/
 extern uint8_t __cy_app1_verify_start;
 extern uint8_t __cy_app1_verify_length;
 /**Secondary Image App2 Flash start and length**/
 extern uint8_t __cy_app2_verify_start;
 extern uint8_t __cy_app2_verify_length;
 /**app boot signature**/
 extern uint8_t __cy_boot_signature_size;

/**DFU Bootloader Flash start and length**/
#define CY_DFU_APP0_VERIFY_START ((uint32_t)&__cy_app0_verify_start )
#define CY_DFU_APP0_VERIFY_LENGTH((uint32_t)&__cy_app0_verify_length )
/**Primary Image App1 Flash start and length**/
#define CY_DFU_APP1_VERIFY_START ((uint32_t)&__cy_app1_verify_start )
#define CY_DFU_APP1_VERIFY_LENGTH((uint32_t)&__cy_app1_verify_length )
/**Secondary Image App2 Flash start and length**/
#define CY_DFU_APP2_VERIFY_START ((uint32_t)&__cy_app2_verify_start )
#define CY_DFU_APP2_VERIFY_LENGTH((uint32_t)&__cy_app2_verify_length )
/**app boot signature**/
#define CY_DFU_SIGNATURE_SIZE    ((uint32_t)&__cy_boot_signature_size)
```

Code Listing 5 Modify cy_dfu_metadata in dfu_user.c

```
 /*
* The DFU SDK metadata's initial value is placed here
* Note: The number of elements is equal to the number of apps multiplied by
2.
* This is because of the two fields per app plus one element for the CRC-32C
field.
*/
CY_SECTION(".cy_boot_metadata") __USED
static const uint32_t cy_dfu_metadata[CY_FLASH_SIZEOF_ROW / sizeof(uint32_t)]
=
{
    CY_DFU_APP0_VERIFY_START, CY_DFU_APP0_VERIFY_LENGTH,
    /* The App0 base address and length */
```

```
    CY_DFU_APP1_VERIFY_START, CY_DFU_APP1_VERIFY_LENGTH,
    /* The App1 base address and length */

    CY_DFU_APP2_VERIFY_START, CY_DFU_APP2_VERIFY_LENGTH,
    /* The App2 base address and length */

    0U /* The rest does not matter     */
};
```

3. Modify the function to jump to a specified boot flash memory address, as shown in Code Listing 6

Code Listing 6 Modify the API cy_en_dfu_status_t parameter in main.c

```
/*******************************************************************************
*****
* Function Name: ManageBootableApp
********************************************************************************
*****
* Performs validation of application, clears the existing reset reason if
* a valid image is found.
*
* Parameters:
*  dfu_params     input DFU parameters.
* Returns:
* CY_DFU_SUCCESS if operation is successful and an error code in case of
failure.
********************************************************************************
****/
static cy_en_dfu_status_t manage_bootable_app(cy_stc_dfu_params_t
*dfu_params, uint32_t AppAddress)
{
    cy_en_dfu_status_t status = CY_DFU_SUCCESS;
    if(!dfu_params){
     status = CY_DFU_ERROR_UNKNOWN;
    }
 /* Satisfy the compiler. */
 (void)dfu_params;
 if (status == CY_DFU_SUCCESS)
 {
 /* Validate the APP in SECONDARY slot. */
 status = Cy_DFU_ValidateApp(AppAddress, NULL);
```

```
}

  if (status == CY_DFU_SUCCESS)
  {
    /*Clear reset reason because Cy_DFU_ExecuteApp() performs a
    software reset.
    Without clearing, two reset reasons would be present.*/
    do
    {
      Cy_SysLib_ClearResetReason();
     }while(Cy_SysLib_GetResetReason() != 0 );
  }
  return status;
}
```

4. Modify the main code to add UART output for debug logs and implement the conditional check for internal flash address jumps, as shown in Code Listing 7 and Code Listing 8

Code Listing 7 Add the header file and macro USER_APP_ID2

```
#include <stdio.h>
/*******************************************************************************
* Macros
*******************************************************************************/
/* Application ID */
#define USER_APP_ID (1u)

/* Application ID 2 */
#define USER_APP_ID_2 (2u)
```

Code Listing 8 Add buffer for message and JumpApp_Address variable

```
/*add parameter for uart debug message*/
cy_stc_scb_uart_context_t DFU_UART_context;
char text[100];

/*to keep bootup flash address*/
uint32_t JumpAPP_Address=0;
```

5. During the main initialization, add a GPIO reading condition to determine whether to launch the application via APP1 or APP2, as shown in Code Listing 9

Code Listing 9 Initialize the UART and add the GPIO

```
/*Enter the code in main.c for the initialization phase*/
  cy_en_scb_uart_status_t initstatus; /*for uart initial*/
  initstatus=Cy_SCB_UART_Init(DFU_UART_HW, &DFU_UART_config,
              &DFU_UART_context);
  if(initstatus!=CY_RSLT_SUCCESS)
  {
        CY_ASSERT(CY_ASSERT_FAILED);
   }
Cy_SCB_UART_Enable(DFU_UART_HW);

Cy_SCB_UART_PutString(DFU_UART_HW,"|==============================|\r\n");
Cy_SCB_UART_PutString(DFU_UART_HW,"|INFINEON PMG1S3 DFU Bootloader|\r\n");
Cy_SCB_UART_PutString(DFU_UART_HW,"|==============================|\r\n");

/*Very important, the CYBSP_USER_SELECT_PIN is an input function pin that
determines whether the
*DFU will update the firmware, and whether the boot process will use the
  primary image or the
*secondary image.
 CYBSP_USER_SELECT_PIN Read High: Update Image app1/ Run  app1 ;
 Low: Update Image app2/Run app2.
*/
if(Cy_GPIO_Read(CYBSP_USER_SELECT_PORT,CYBSP_USER_SELECT_PIN)){
    JumpAPP_Address=USER_APP_ID;
}else{
    JumpAPP_Address=USER_APP_ID_2;
}
/* In the case of non-software reset check if there is a valid app image.
* If these is - switch to it.*/
if( (Cy_SysLib_GetResetReason() != CY_SYSLIB_RESET_SOFT) ){
/* Boot to application image, if valid. */
    status = manage_bootable_app(&dfu_params,JumpAPP_Address);
    if (status == CY_DFU_SUCCESS){
        /* Never returns from here. */
        Cy_DFU_ExecuteApp(JumpAPP_Address);
    }
/* Continue DFU execution (below), if user application not valid !. */
```

```
}
```

6. Ensure that the code is modified so that the API changes from **manage_bootable_app(&dfu_params)** to **manage_bootable_app(&dfu_params, JumpAPP_Address)**, as indicated in the highlighted sections that require modification, as shown in Code Listing 10 and Code Listing 11

Code Listing 10 Modify API manage_bootable_app at main.c for loop

```
for (;;)
{
     status = Cy_DFU_Continue(&state, &dfu_params);
     ++count;
     if (state == CY_DFU_STATE_FINISHED)
     {
  /* Finished loading the application image.
   * Validate DFU application, if it is valid then switch to it.
   */
  /*Use the modified API and pass in the Image parameter.*/

   status = manage_bootable_app(&dfu_params,JumpAPP_Address);
   if (status == CY_DFU_SUCCESS)
   {
     Cy_DFU_TransportStop();
     Cy_DFU_ExecuteApp(JumpAPP_Address);
}
     else if (status == CY_DFU_ERROR_VERIFY){
```

Code Listing 11 Timeout jump to application app

```
if((count >= counter_timeout_seconds(DFU_IDLE_TIMEOUT_MS,
   DFU_SESSION_TIMEOUT_MS) ) && (state == CY_DFU_STATE_NONE))
{
    status = manage_bootable_app(&dfu_params,JumpAPP_Address);
   if (status == CY_DFU_SUCCESS){
       Cy_DFU_TransportStop();
       Cy_DFU_ExecuteApp(JumpAPP_Address);
   }

/* In case, no valid user application, lets start fresh all over.
* This is just for demonstration.
```

```
* Final application can change it to either assert, reboot, enter low power
mode etc,
* based on usecase requirements.
*/
   count = 0 ;
}

/* Is user_button event detected ?. */
if(is_user_event_detected == true)
{
    is_user_event_detected = false;
    /* Boot to application image, if valid. */
    status = manage_bootable_app(&dfu_params,JumpAPP_Address);
    if (status == CY_DFU_SUCCESS)
    {
     /* Never returns from here. */
        Cy_DFU_ExecuteApp(JumpAPP_Address);
    }
}
```

7. Build the modified **Dual_Image_DFU_Bootloader** project, as shown in Figure 50

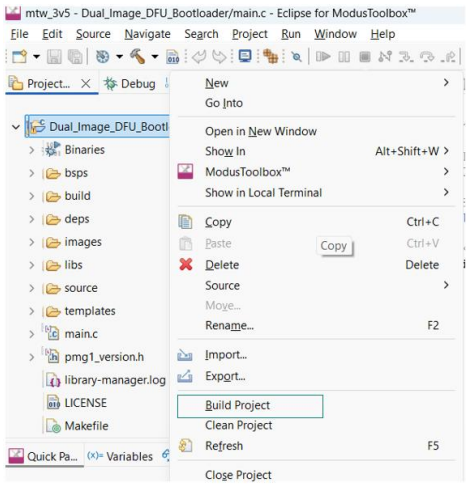


Figure 50 Building Modify Dual_Image_DFU_Bootloader

8. Build the **Dual Image DFU_Bootloader** project, as shown in Figure 51

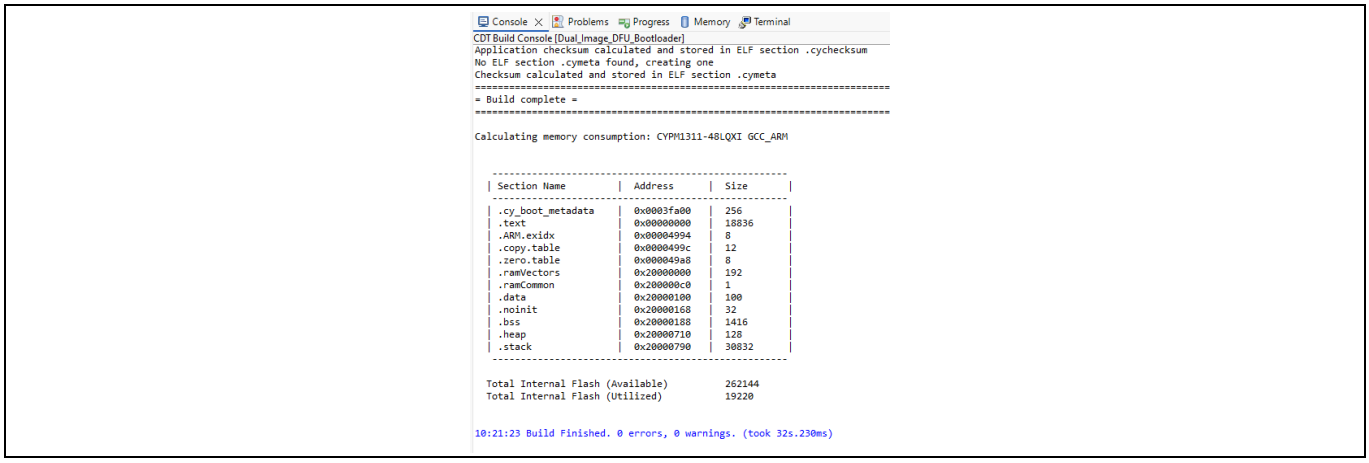


Figure 51 Dual Image DFU_Bootloader successful build

In the default path *..\<root>\workspace\Dual_Image_DFU_Bootloader\build\USB_CDC\APP_PMG1-CY7113\Debug*, verify that a hex file has been generated. Verify that the flashing program file generated by DFU_Bootloader has a file extension of **hex**, while the file generated for DFU updating the internal main app program of the MCU has a file extension of **cyacd2**, as shown in Figure 52.

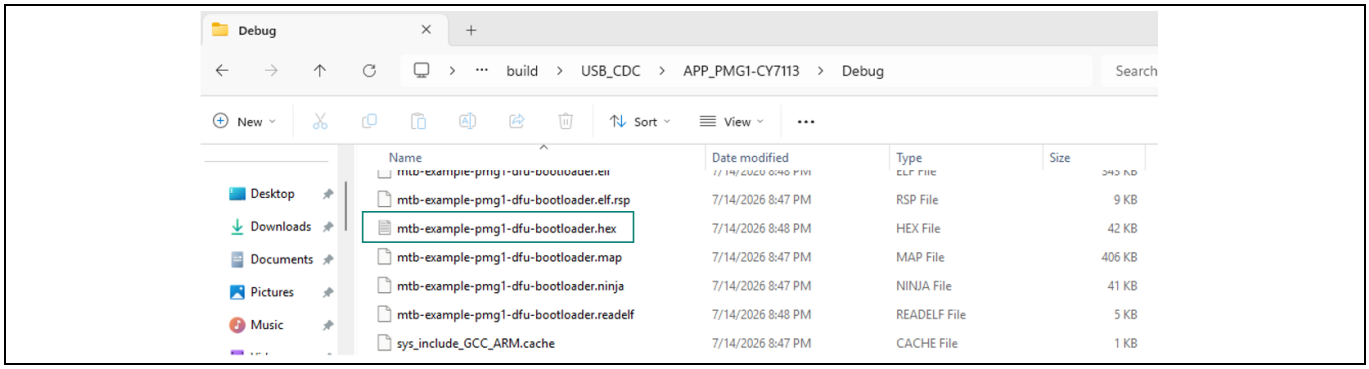


Figure 52 Dual Image DFU_Bootloader hex file generated after building the project

9 Modify DFU_Blinky_LED in ModusToolbox™ for the application

The LED blink sample used here will be sourced from the basic example **DFU_Blinky_LED** within ModusToolbox™. Follow these steps for project creation process:

1. As shown in Figure 53 and Figure 54, applications will be created and renamed as Dual_DFU_Blinky_LED_APP1 and Dual_DFU_Blinky_LED_APP2

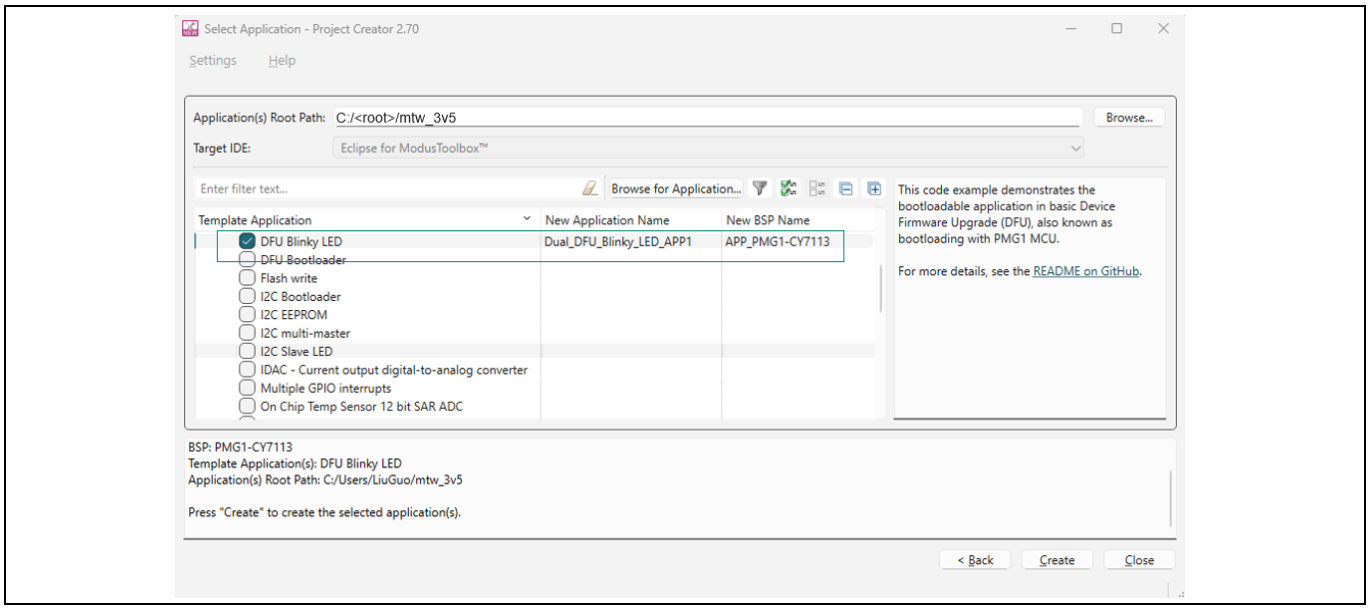


Figure 53 Create Dual_DFU_Blinky_APP1 project

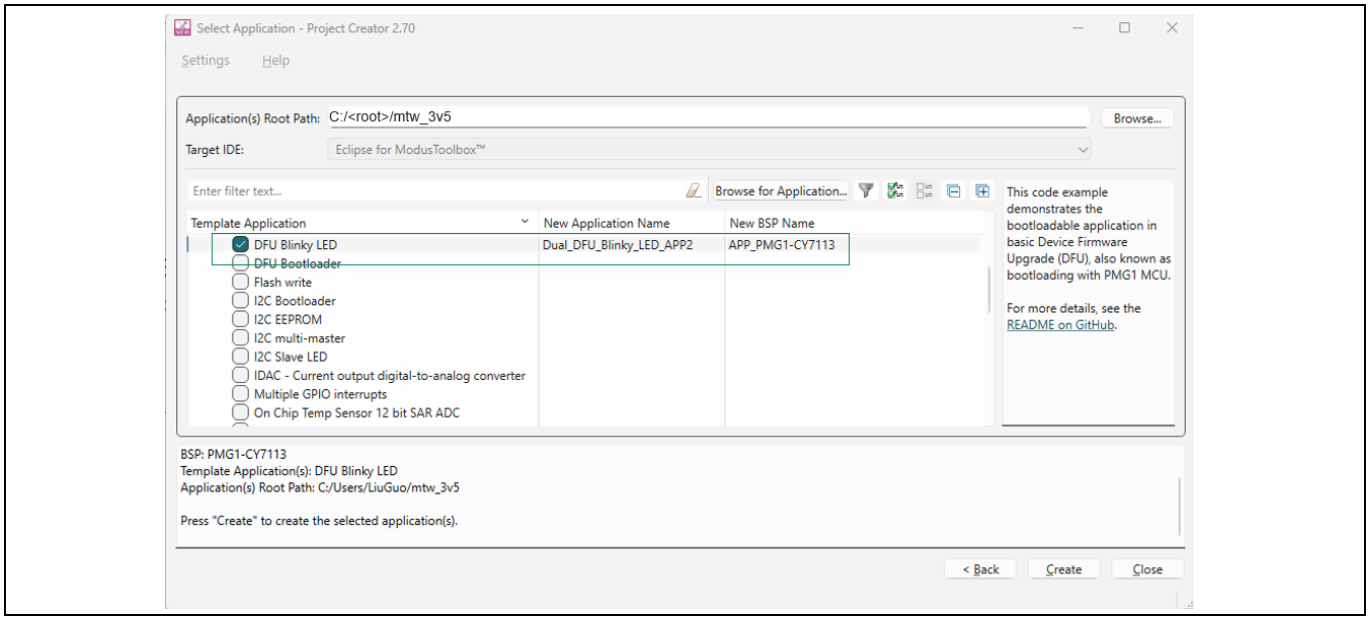


Figure 54 Create Dual_DFU_Blinky_APP2 project

2. Configure the application (APP1/APP2) Flash memory usage. Config target flash split configuration file **dfu_cm0p_app1.ld**, as shown in Code Listing 12 and Figure 57. In **app1**, the location of Flash must specify the configuration for **FLASH_app1** with REGION_ALIAS("FLASH", **FLASH_app1**)

Code Listing 12 dfu_cm0p_app1.ld by application to set the metadata

```
/* The MEMORY section below describes the location and size of blocks of
memory in the target.
* Use this section to specify the memory regions available for allocation.
*/
MEMORY
{
/* The ram and FLASH regions control RAM and FLASH memory allocation.
*  You can change the memory allocation by editing the 'ram' and 'FLASH'
regions.
*/
FLASH_app0 (rx)  : ORIGIN = 0x00000000, LENGTH = 0x00006000
FLASH_app1 (rx)  : ORIGIN = 0x00006000, LENGTH = 0x0001C000
FLASH_app2 (rx)  : ORIGIN = 0x00022000, LENGTH = 0x0001C000

RAM_app0   (rwx) : ORIGIN = 0x20000000,   LENGTH = 0x00008000
RAM_app1   (rwx) : ORIGIN = 0x20000000,   LENGTH = 0x00008000
RAM_app2   (rwx) : ORIGIN = 0x20000000,   LENGTH = 0x00008000

flash_boot_meta   (rw)  : ORIGIN = 0x0003FA00, LENGTH = 0x400

}

/* The DFU SDK metadata limits */
__cy_boot_metadata_addr = ORIGIN(flash_boot_meta);
__cy_boot_metadata_length = __cy_memory_0_row_size;

/* The Product ID, used by CyMCUElfTool to generate a updating file */
__cy_product_id = 0x01020304;

/* The checksum type used by CyMCUElfTool to generate a updating file */
__cy_checksum_type = 0x00;

/* Used by the DFU SDK application to set the metadata */
__cy_app0_verify_start  = ORIGIN(FLASH_app0);
__cy_app0_verify_length = LENGTH(FLASH_app0) - __cy_boot_signature_size;
__cy_app1_verify_start  = ORIGIN(FLASH_app1);
__cy_app1_verify_length = LENGTH(FLASH_app1) - __cy_boot_signature_size;
__cy_app2_verify_start  = ORIGIN(FLASH_app2);
```

```
__cy_app2_verify_length = LENGTH(FLASH_app2) - __cy_boot_signature_size;
/*
* The size of the application signature.
* E.g. 4 for CRC-32,
*      32 for SHA256,
*     256 for RSA 2048.
*/
__cy_boot_signature_size = 4;

/*
* DFU SDK specific: aliases regions, so the rest of the code does not use
* application-specific memory region names
*/
REGION_ALIAS("FLASH", FLASH_app1);
REGION_ALIAS("RAM",     RAM_app1);
```

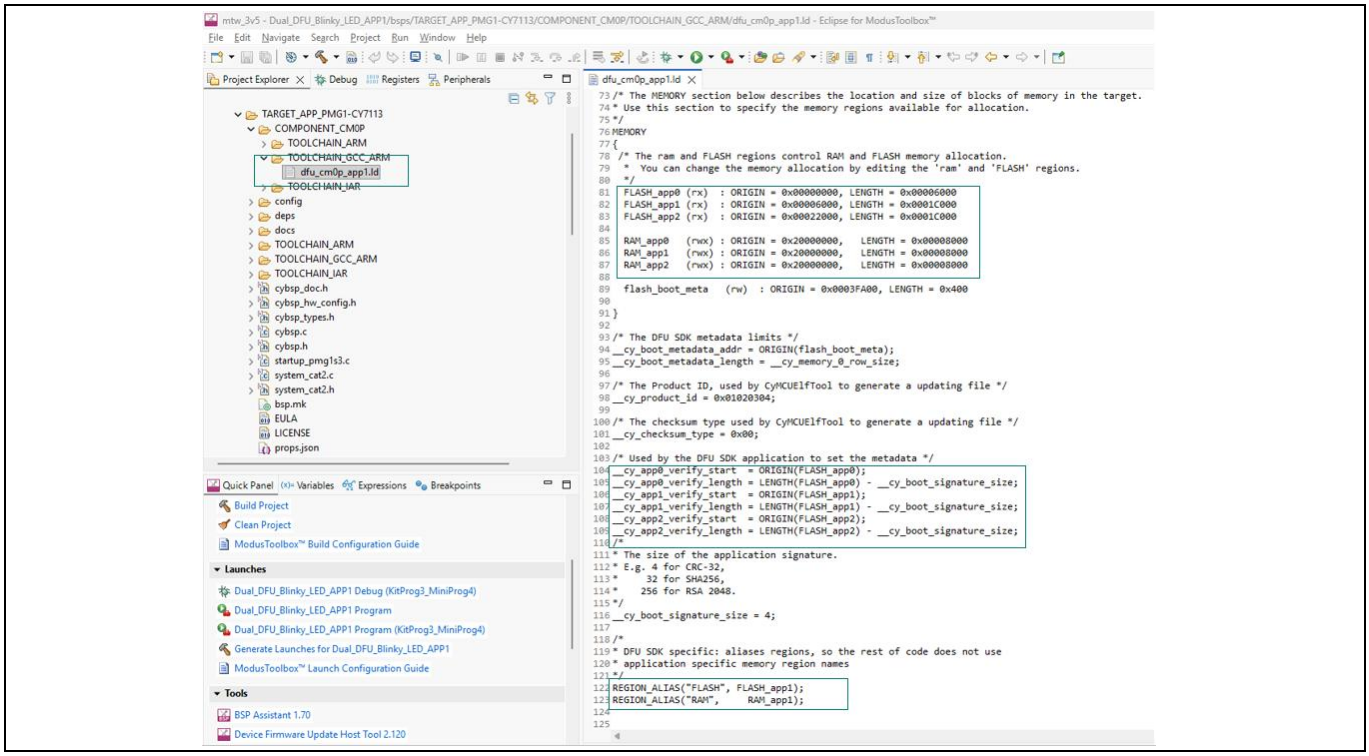


Figure 55 Application 1 dfu_cm0p_app1.ld configuration

3. Config target flash split configuration file **dfu_cm0p_app2.ld**, as shown in Code Listing 13 and Figure 56. In **app2**, the location of Flash must specify the configuration for **FLASH_app2** with REGION_ALIAS("FLASH", **FLASH_app2**)

Code Listing 13 dfu_cm0p_app2.ld by application to set the metadata

```
/* The MEMORY section below describes the location and size of blocks of
memory in the target.
* Use this section to specify the memory regions available for allocation.
*/
MEMORY
{
 /* The ram and FLASH regions control RAM and FLASH memory allocation.
 *  You can change the memory allocation by editing the 'ram' and 'FLASH'
regions.
*/
  FLASH_app0 (rx)   : ORIGIN = 0x00000000, LENGTH = 0x00006000
  FLASH_app1 (rx)   : ORIGIN = 0x00006000, LENGTH = 0x0001C000
  FLASH_app2 (rx)   : ORIGIN = 0x00022000, LENGTH = 0x0001C000

  RAM_app0    (rwx) : ORIGIN = 0x20000000,    LENGTH = 0x00008000
  RAM_app1    (rwx) : ORIGIN = 0x20000000,    LENGTH = 0x00008000
  RAM_app2    (rwx) : ORIGIN = 0x20000000,    LENGTH = 0x00008000

  flash_boot_meta    (rw)   : ORIGIN = 0x0003FA00, LENGTH = 0x40
}
/* The DFU SDK metadata limits */
__cy_boot_metadata_addr = ORIGIN(flash_boot_meta);
__cy_boot_metadata_length = __cy_memory_0_row_size;

/* The Product ID, used by CyMCUElfTool to generate a updating file */
__cy_product_id = 0x01020304;

/* The checksum type used by CyMCUElfTool to generate a updating file */
__cy_checksum_type = 0x00;

/* Used by the DFU SDK application to set the metadata */
__cy_app0_verify_start  = ORIGIN(FLASH_app0);
__cy_app0_verify_length = LENGTH(FLASH_app0) - __cy_boot_signature_size;
__cy_app1_verify_start  = ORIGIN(FLASH_app1);
__cy_app1_verify_length = LENGTH(FLASH_app1) - __cy_boot_signature_size;
__cy_app2_verify_start  = ORIGIN(FLASH_app2);
__cy_app2_verify_length = LENGTH(FLASH_app2) - __cy_boot_signature_size;
/*
```

```
* The size of the application signature.
* E.g. 4 for CRC-32,
*      32 for SHA256,
*     256 for RSA 2048.
*/
__cy_boot_signature_size = 4;

/* DFU SDK specific: aliases regions, so the rest of code does not use
* application specific memory region names*/
REGION_ALIAS("FLASH", FLASH_app2);
REGION_ALIAS("RAM",     RAM_app2);
```

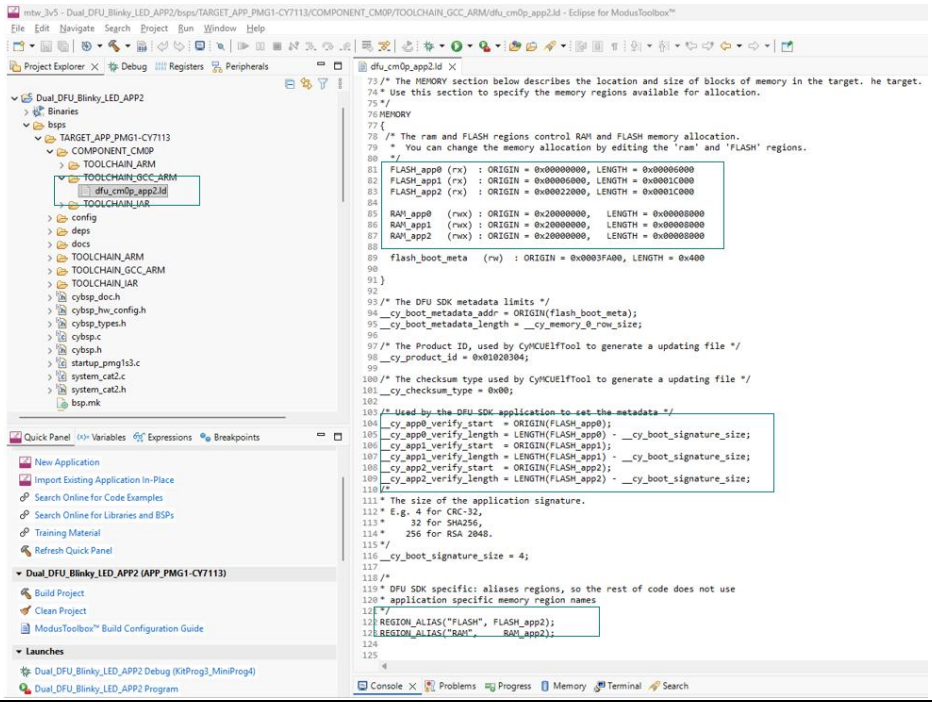


Figure 56 Application 2 dfu_cm0p_app2.ld configuration

4. Modify the Makefile for **dfu_cm0p_app1.ld** and **dfu_cm0p_app2. ld**. Allow the compiler to find the corresponding ID file configuration, as shown in Code Listing 14 and Code Listing 15

Code Listing 14 Modify the APP1 makefile to use the path to the linker script

```
# Path to the linker script to use (if empty, use the default linker
script).

LINKER_SCRIPT=$(MTB_TOOLS__TARGET_DIR)/COMPONENT_$(MTB_RECIPE__CORE)/
TOOLCHAIN_$(TOOLCHAIN)/dfu_cm0p_app1.$(MTB_RECIPE__SUFFIX_LS)
```

Code Listing 15 Modify the APP2 makefile to use the path to the linker script

```
# Path to the linker script to use (if empty, use the default linker
script).

LINKER_SCRIPT=$(MTB_TOOLS__TARGET_DIR)/COMPONENT_$(MTB_RECIPE__CORE)/
TOOLCHAIN_$(TOOLCHAIN)/dfu_cm0p_app2.$(MTB_RECIPE__SUFFIX_LS)
```

5. The Device Configurator includes the UART Interface and the approach for both APP1 and APP2 is the same, as shown in Figure 57 and Figure 58

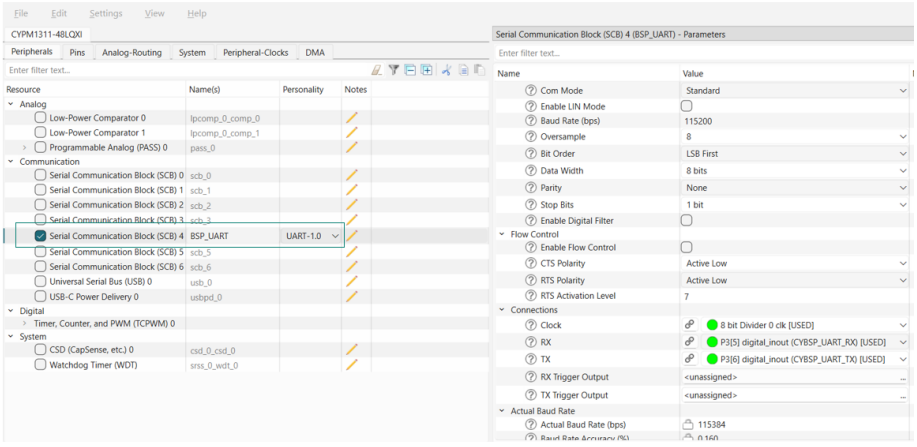


Figure 57 Application UART interface configuration

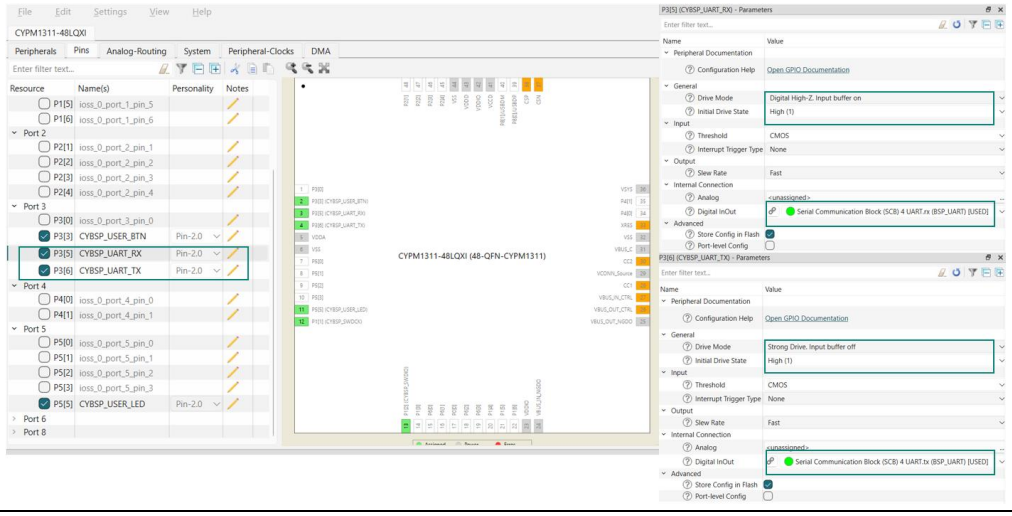


Figure 58 Application UART pin assignment and configuration

6. Modify the APP1 and APP2 main code and add a UART debug message. During testing and verification, you can check in the terminal which image code the system is currently executing.

Code Listing 16 Modify the application main code

```
cy_stc_scb_uart_context_t UART_context;
char text[100];
```

```
/*Define LED toggle wait time*/
#define LED_TOGGLE_INTERVAL_MS      (100u)
int main(void)
{
    cy_rslt_t result;
    /* Initialize the device and board peripherals */
    result = cybsp_init() ;
    if (result != CY_RSLT_SUCCESS)
    {
        CY_ASSERT(0);
    }

    cy_en_scb_uart_status_t initstatus;//for uart initial
    initstatus=Cy_SCB_UART_Init(BSP_UART_HW, &BSP_UART_config,
               &UART_context);
    if(initstatus!=CY_RSLT_SUCCESS)
    {
      //CY_ASSERT(CY_ASSERT_FAILED);
    }
    Cy_SCB_UART_Enable(BSP_UART_HW);

    /* Configure interrupt */
    Cy_SysInt_Init(&user_button_intr_config, &user_button_event_handler);

    /* Clear any pending interrupt and enable interrupt. */
    NVIC_ClearPendingIRQ(user_button_intr_config.intrSrc);
    NVIC_EnableIRQ(user_button_intr_config.intrSrc);

    /* Enable global interrupts */
    __enable_irq();

    //add message log App1/App2
    Cy_SCB_UART_PutString(BSP_UART_HW,"|-INFINEON PMG1S3 boot App1-|\r\n");

    for (;;)
    {
        /* Invert the USER LED state */
        Cy_GPIO_Inv(CYBSP_USER_LED_PORT, CYBSP_USER_LED_PIN);
        /* Delay between LED toggles app1/app2 different*/
        Cy_SysLib_Delay(LED_TOGGLE_INTERVAL_MS);
```

```
        /* Switch to bootloader, if user button is pressed. */
        if(is_user_event_detected == true)
        {
            is_user_event_detected = false;
            Cy_DFU_ExecuteApp(BOOTLOADER_ID);
        }
    }
}
```

7. Build each project and generate the required cyacd2 firmware image, as shown in Figure 59 and Figure 60

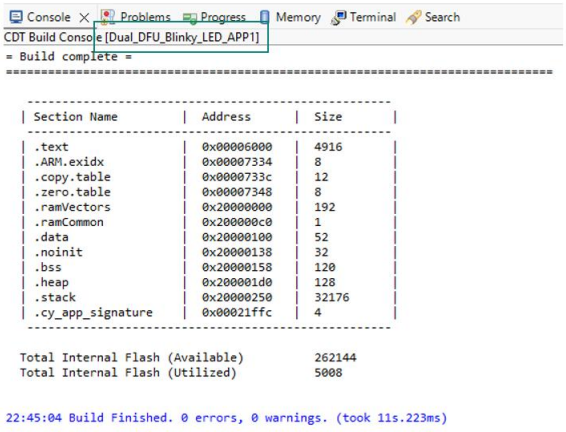


Figure 59 Application 1 build result

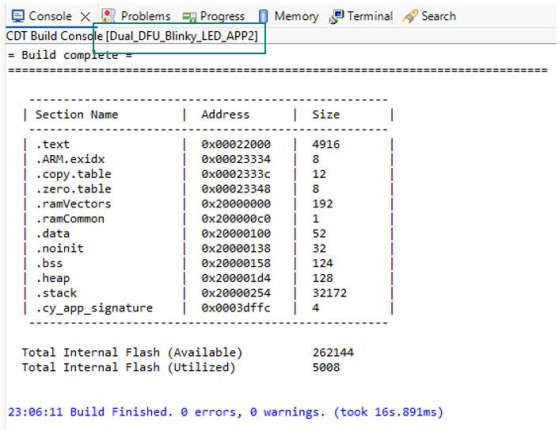


Figure 60 Application 2 build result

8. After the building is complete, you can find the generated *mtb-example-pmg1-dfu-blinky-led_crc.cyacd2* file in the default path ..\ *username**mtw**Dual_DFU_Blinky_LED_APP1**build**APP_PMG1-CY7113**Debug* and path ..\ *username**mtw**Dual_DFU_Blinky_LED_APP2**build**APP_PMG1-CY7113**Debug* folders. This file will be used later for DFU to update the main program firmware, as shown in Figure 61 and Figure 62

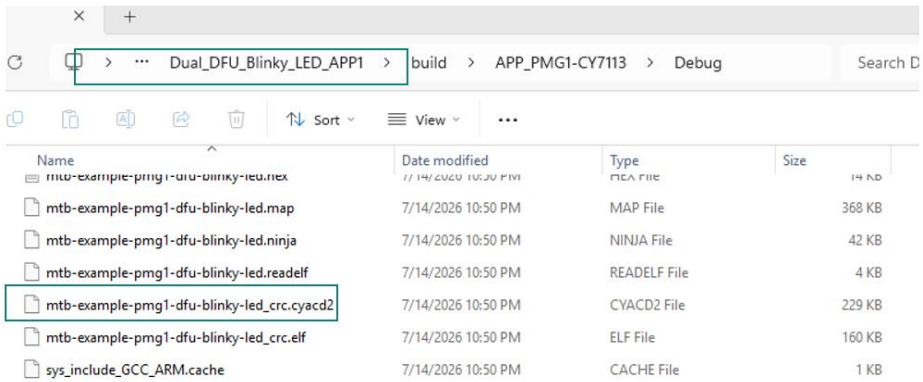


Figure 61 CYACD2 file generated after building the App1 project

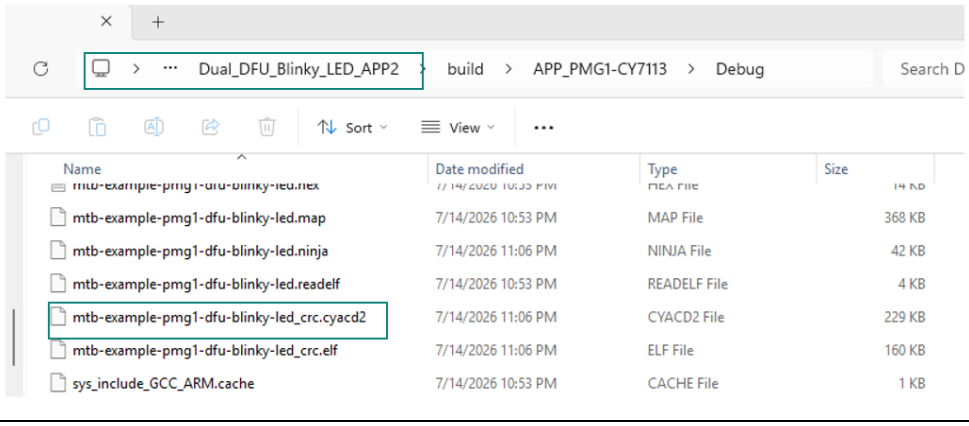


Figure 62 CYACD2 file generated after building the App2 project

10 Programming and verifying the entire operating mechanism

In this application, the firmware updated via DFU is updated into the PMG1-S3 space by the DFU Host Tool. There is no need to specifically move the flash memory; if a selection mechanism is added before the DFU Bootloader starts up, it can choose to prioritize either the primary image or the secondary image as the boot process for this session. Perform verification according to the following steps.

1. Program the DFU bootloader HEX file through the **mtb-programmer**. The jumpers (J5) connect to PIN 2 and PIN 3 to supply a 5 V voltage from the USB connector at J1, as shown in Figure 63. The UART jump connections on the CY7113 EZ-PD™ PMG1-S3 MCU Prototyping Kit and the prototyping kit itself include a mini-programmer for burning functionality along with a virtual COM port. Here, connect the MCU's UART functionality to the prototyping kit's COM port with jumpers, as shown in Figure 64

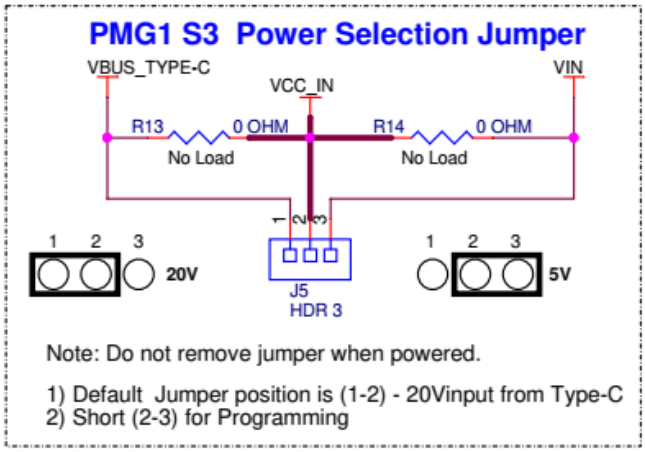


Figure 63 EZ-PD™ PMG1-S3 MCU power selection jumper

- **Blue color cable:** Connect PIN 10 of J4 to PIN 10 of J6, [P3.6] UART_TX
- **Green color cable:** Connect PIN 8 of J4 to PIN 9 of J6, [P3.5] UART_RX

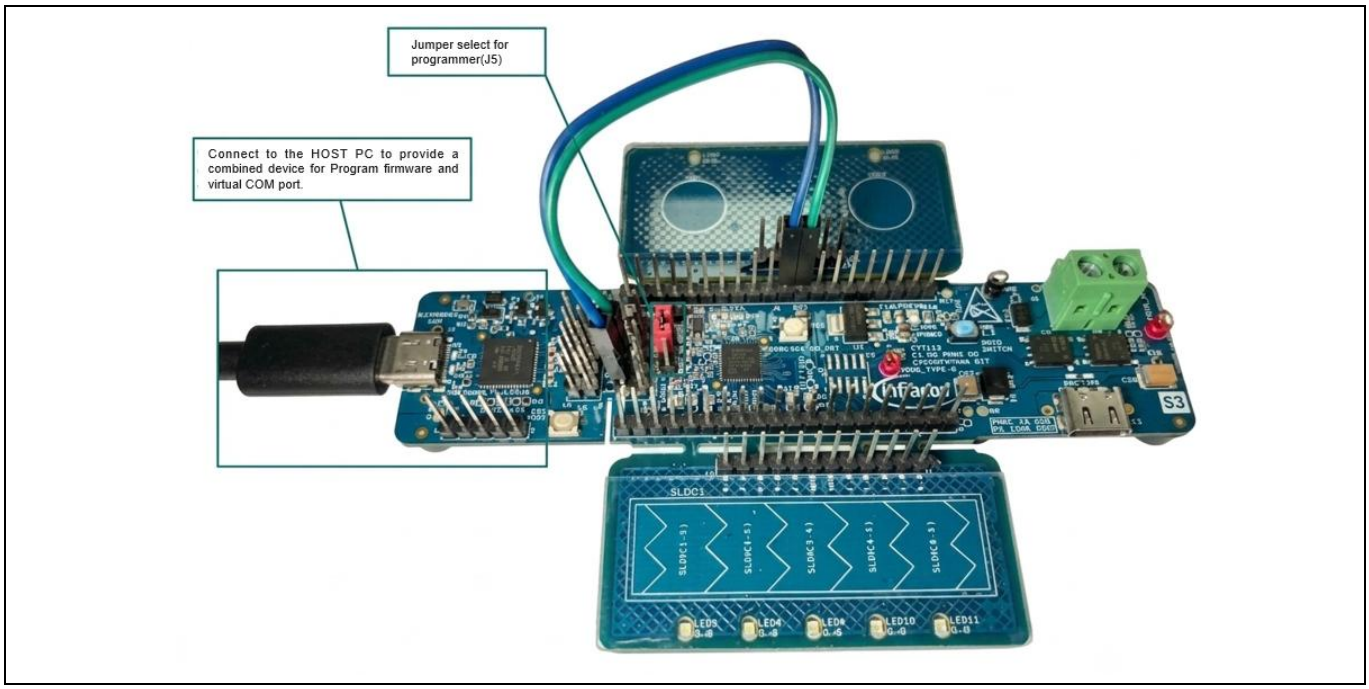


Figure 64 Jumper select for programmer (J5) and cable connect

2. Program the dual image DFU bootloader into the EZ-PD™ PMG1-S3 MCU using the **mtb-programmer**, as shown in Figure 65, and the firmware programmer, as shown in Figure 66

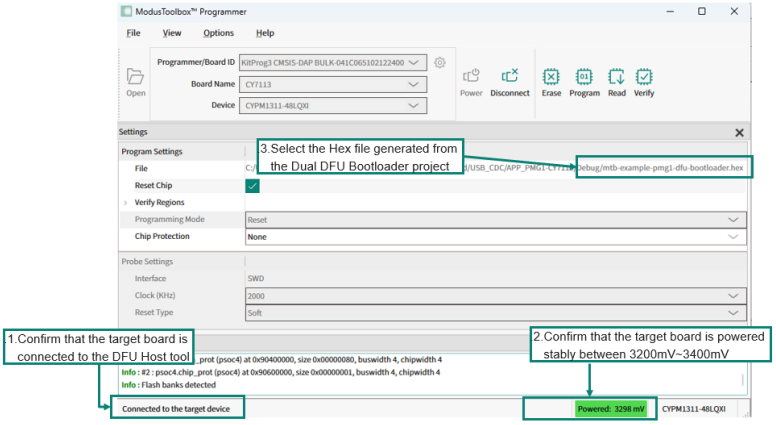


Figure 65 Connection status of the target board and the mtb-programmer

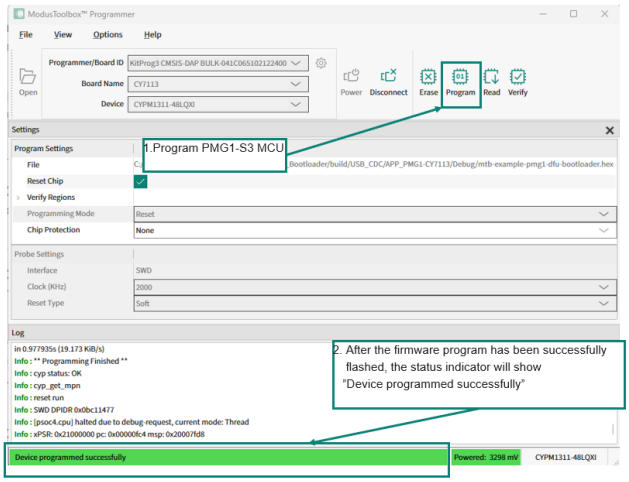


Figure 66 Finish program dual DFU bootloader

3. Adjust the position of J5 so that the power source is connected through the USB-CDC connector at J10 to supply power to the EZ-PD™ PMG1-S3 MCU. The original virtual COM port will still be available and can be used for subsequent verification of the message log, as shown in Figure 67

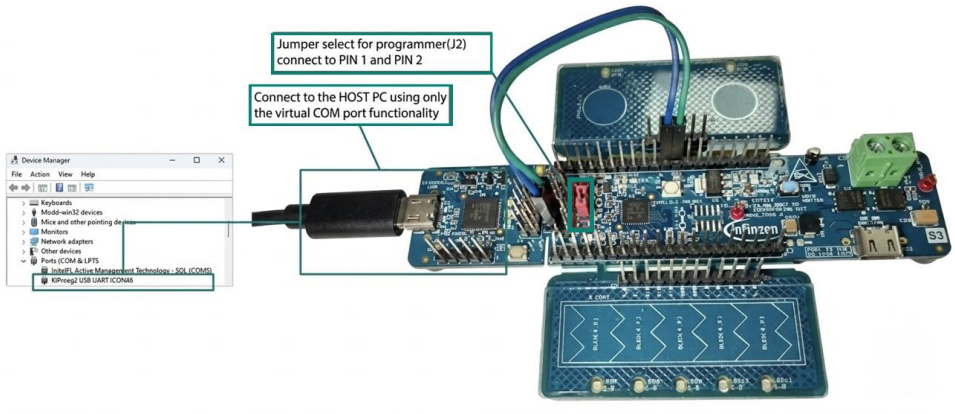


Figure 67 Jumper select for programmer (J5) connect to PIN 1 and PIN 2

4. After connecting J10 to the USB cable, the LED 4 power indicator will glow continuously, and the flashing of LED 3 indicates that the MCU's DFU bootloader is functioning normally. In KitProg3 USB-UART's COM port, you can see the printed messages through the terminal software. Additionally, after J10 is connected to the USB cable, a USB-CDC device will be generated, which is configured in the EZ-PD™ PMG1-S3 MCU. This USB-CDC interface will be used for updating the firmware image in the following steps, as shown in Figure 68

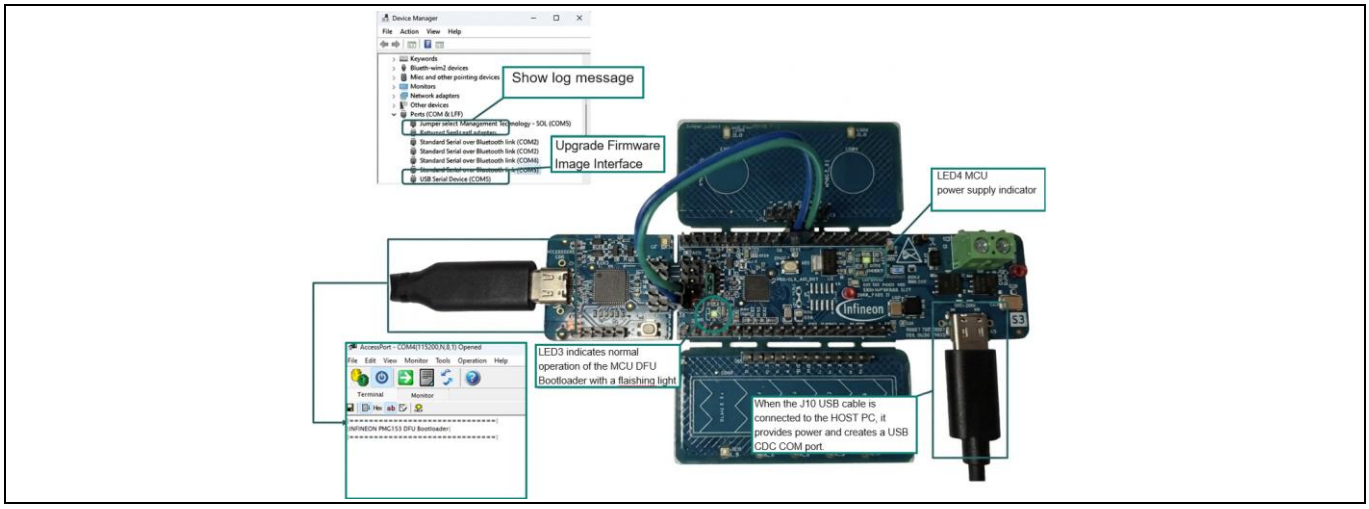


Figure 68 J10 connect USB cable

5. This application document uses the **CYBSP_USER_SELECT_PIN** as a criterion for determining whether to update the primary image or secondary image. When the **CYBSP_USER_SELECT_PIN** is at a HIGH logic level, it allows updating or normal operation of APP1 with the primary image. When the **CYBSP_USER_SELECT_PIN** is at a LOW logic level, it allows updating or normal operation of APP2 with the secondary image

Table 4 CYBSP_USER_SELECT_PIN read state

Logic level	Update image	Running image
HIGH	Primary image APP1	Primary image APP1
LOW	Secondary image APP2	Secondary image APP2

6. First, store the primary image address in the APP1 program. The yellow color cable **CYBSP_USER_SELECT_PIN** (J7, PIN4 [P1.5]) will be connected to a 3.3 V pin with a pull-up, as shown in Figure 69

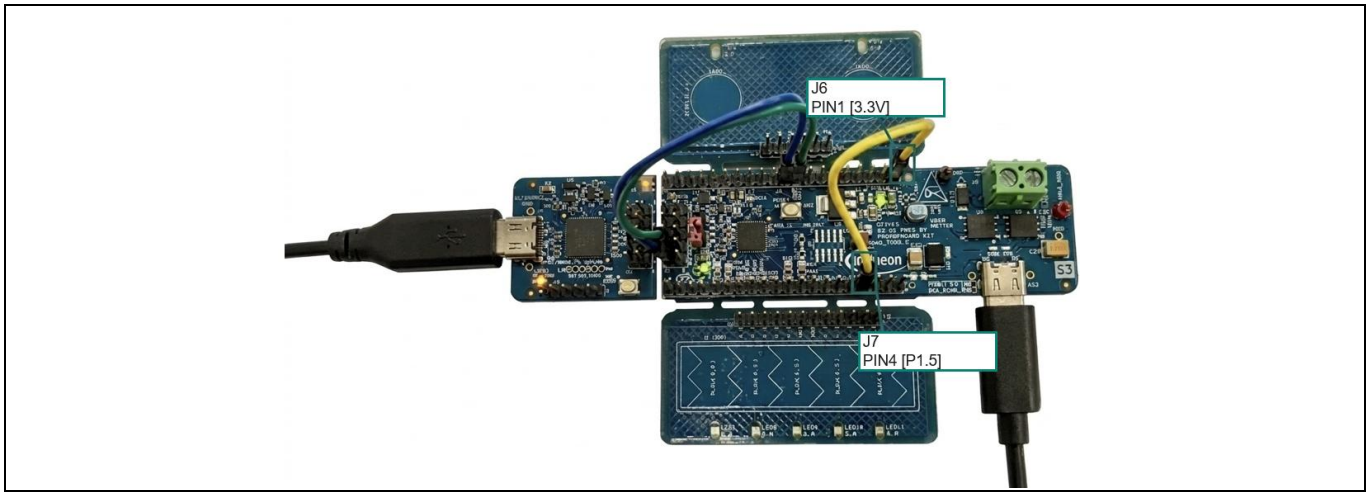


Figure 69 CYBSP_USER_SELECT_PIN pull HIGH

7. Use the DFU Host Tool and EZ-PD™ PMG1-S3 MCU USB-CDC interface to upload the APP1 firmware to the primary image address, as shown in Figure 70

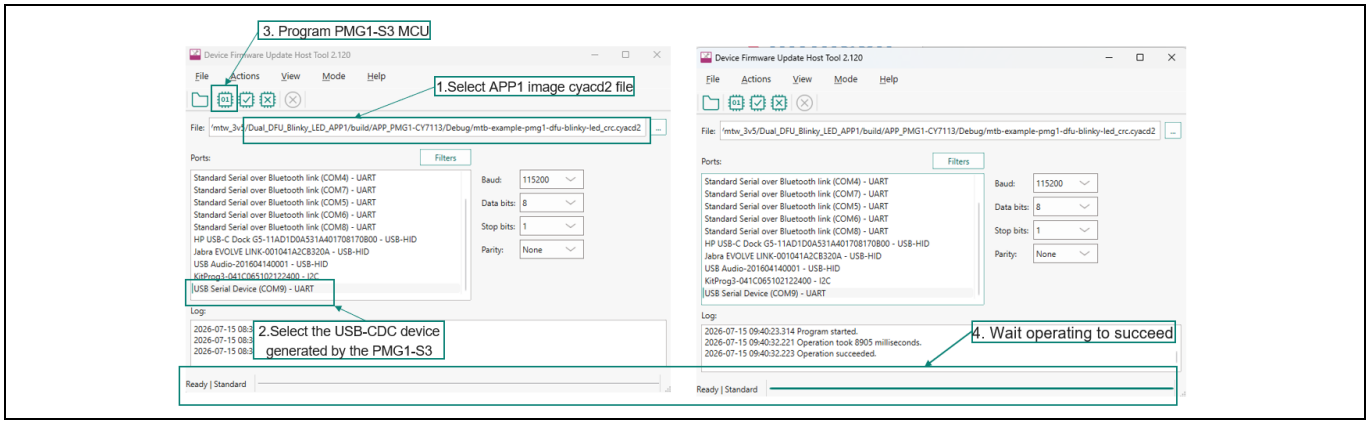


Figure 70 Program APP1 image file

8. Second, store the secondary image address in the APP2 program. The yellow color cable **CYBSP_USER_SELECT_PIN** (J7, PIN4 [P1.5]) will be connected to a GND pin with a pull-down, as shown in Figure 71

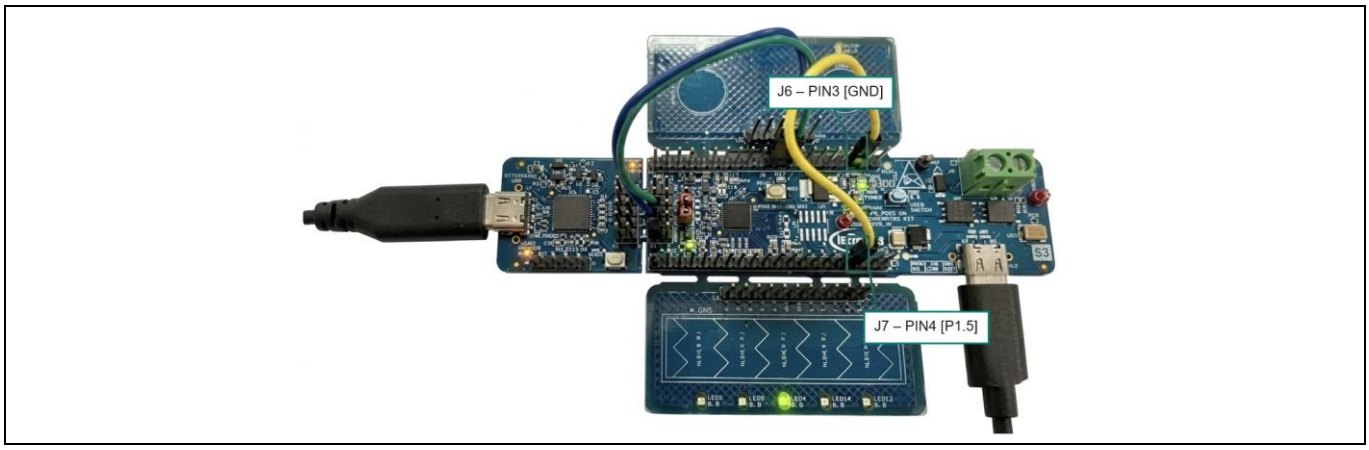


Figure 71 CYBSP_USER_SELECT_PIN pull LOW

9. Use the DFU Host Tool and EZ-PD™ PMG1-S3 USB-CDC interface to upload the APP2 firmware to the secondary image address, as shown in Figure 72

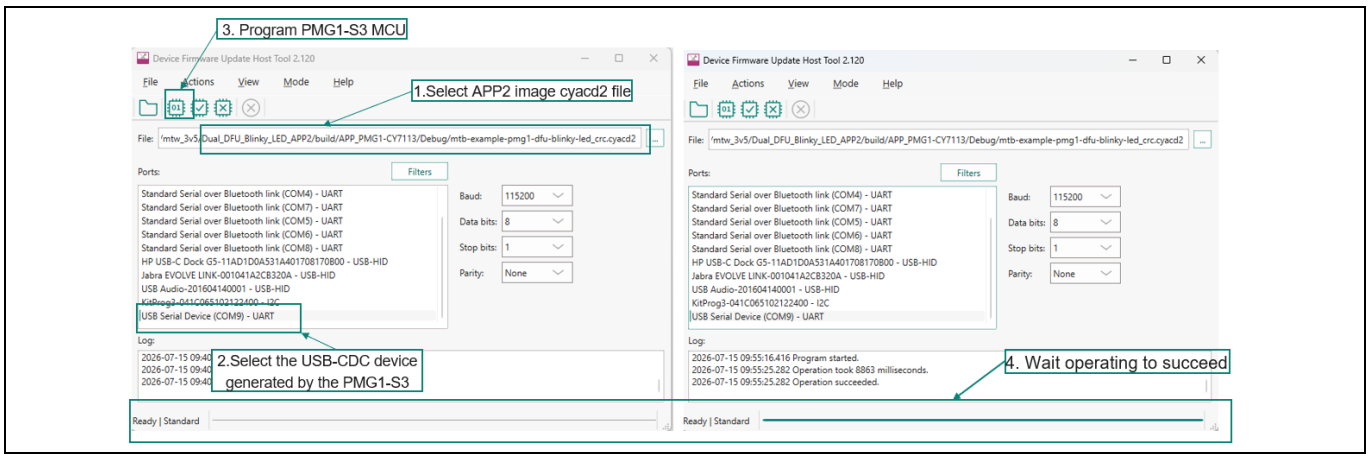


Figure 72 Program APP2 image file

10. The boot process enters the APP1 test. The yellow color cable **CYBSP_USER_SELECT_PIN** (J7, PIN4 [P1.5]) will be connected to a 3.3 V pin with a pull-up. You can observe the log and the blinking frequency of **LED 3**. The blinking frequency in APP1 is defined in the code to have an interval of 100 ms, as shown in Figure 73

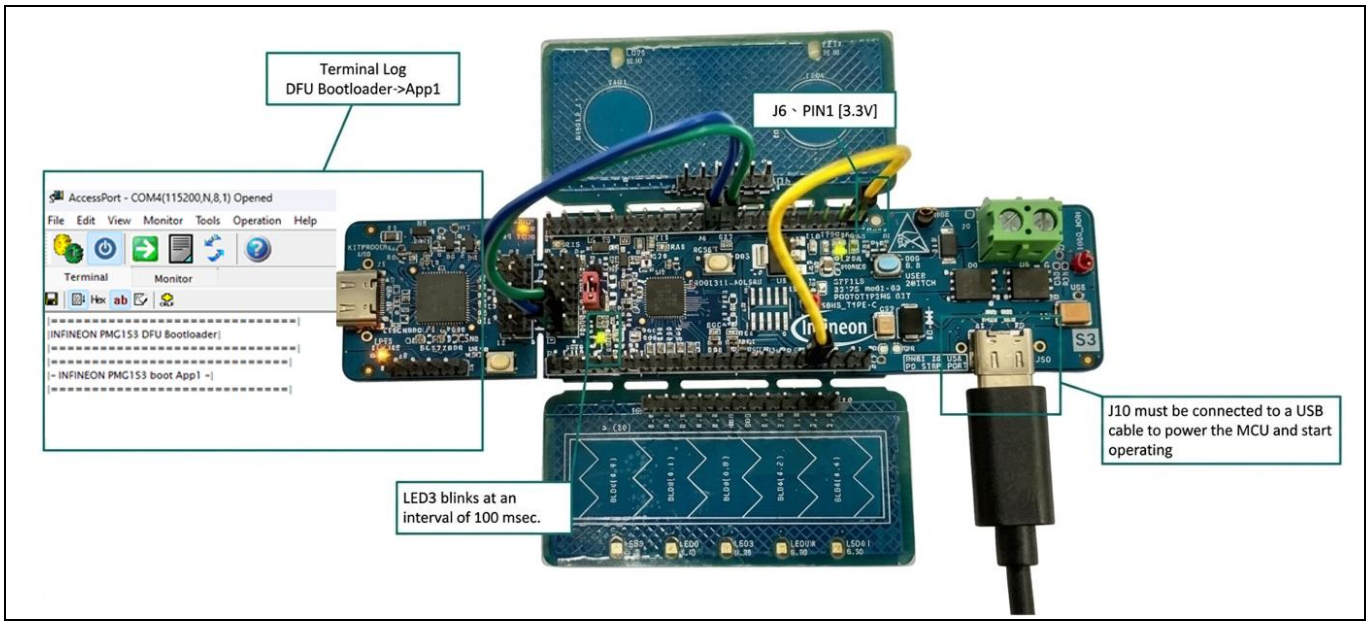


Figure 73 EZ-PD™ PMG1-S3 bootloader initiates the APP1 program

11. The boot process enters the APP2 test. The yellow cable **CYBSP_USER_SELECT_PIN** (J7, PIN4 [P1.5]) will be connected to a GND pin with a pull-down. You can observe the log and the blinking frequency of **LED 3**. The blinking frequency in APP2 is defined in the code to have an interval of 200 msec, as shown in Figure 74

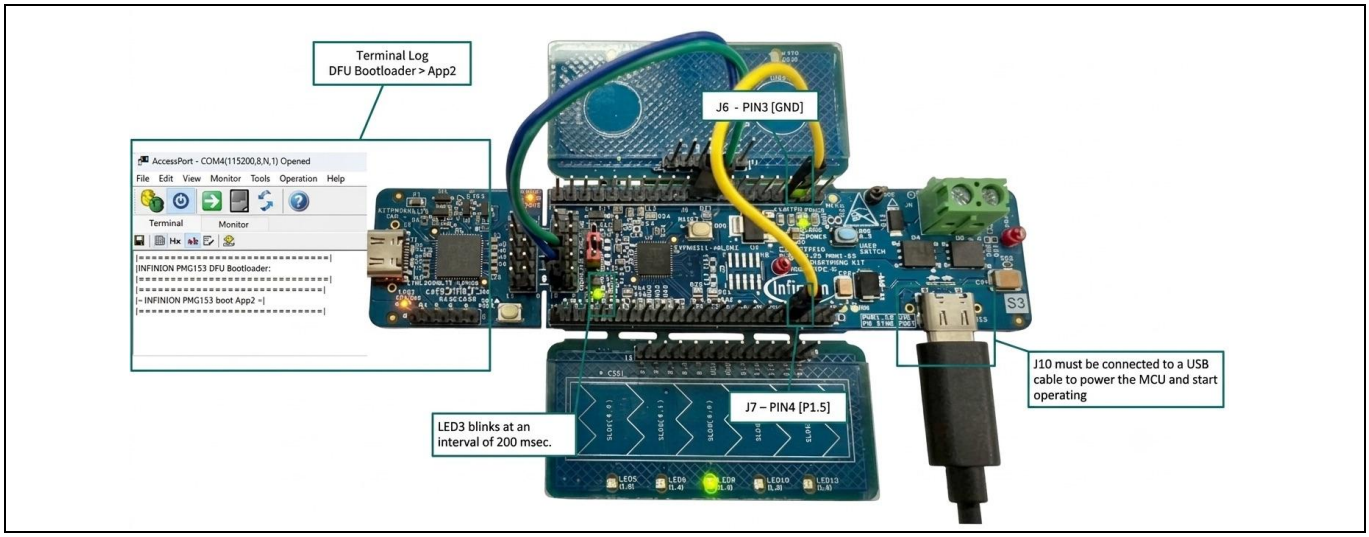


Figure 74 EZ-PD™ PMG1-S3 MCU bootloader initiates the APP2 program

11 Related resources

- Webpage: EZ-PD™ PMG1-S3 high-voltage MCU with USB-C and PD
- Datasheet: CYPM13xx, EZ-PD™ PMG1-S3 Power Delivery MCU Gen1
- Infineon Developer Community: EZ-PD™ USB Type-C forum

References

[1] Infineon Technologies AG: *EZ-PD™ PMG1 MCU prototyping kits guide*; Available online

[2] Infineon Technologies AG: *ModusToolbox™ Programmer GUI user guide*; Available online

[3] Infineon Technologies AG: *ModusToolbox™ DFU Host Tool user guide*; Available online

[4] Infineon Technologies AG: *Eclipse for ModusToolbox™ user guide*; Available online

[5] Infineon Technologies AG: *EZ-PD™ PMG1-S3 MCU architecture reference manual*; Available online

[6] Infineon Technologies AG: *Getting started with EZ-PD™ PMG1 MCU on ModusToolbox™ software*; Available online

[7] Infineon Technologies AG: *Device Firmware Update (DFU) middleware (MW) for ModusToolbox™*; Available online

[8] Infineon Technologies AG: *PSOC™ 6 MCU Device Firmware Update (DFU) software development kit guide*; Available online

[9] Infineon Technologies AG: *EZ-PD™ PMG1 MCU: Basic device firmware upgrade (DFU) bootloader*; Available online

[10] Infineon Technologies AG: *EZ-PD™ PMG1 MCU: DFU Blinky LED application*; Available online

Revision history

Document revision	Date	Description of changes
**	2026-08-07	Initial release

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

Edition: 2026-08-07

Published by
Infineon Technologies AG
Am Campeon 1-15
85579 Neubiberg, Germany

© 2026 Infineon Technologies AG.
All rights reserved

Do you have a question about this document?
Email: erratum@infineon.com

Document reference:
002-43359 Rev. **

Important Notice
Products which may also include samples and may be comprised of hardware or software or both ("Product(s)") are sold or provided and delivered by Infineon Technologies AG and its affiliates ("Infineon") subject to the terms and conditions of the frame supply contract or other written agreement(s) executed by a customer and Infineon or, in the absence of the foregoing, the applicable Sales Conditions of Infineon. General terms and conditions of a customer or deviations from applicable Sales Conditions of Infineon shall only be binding for Infineon if and to the extent Infineon has given its express written consent.
For the avoidance of doubt, Infineon disclaims all warranties of non-infringement of third-party rights and implied warranties such as warranties of fitness for a specific use/purpose or merchantability.
Infineon shall not be responsible for any information with respect to samples, the application or customer's specific use of any Product or for any examples or typical values given in this document.
If this document is marked as 'Preliminary', 'Target', 'Draft' or 'Proposal', Infineon reserves the right to change all information given in this document at any time without notice. Subject to the development and release of a Product for series supply by Infineon, the technical specifications of the Product are set forth in the relevant datasheet provided by Infineon without such marking.
The data contained in this document is exclusively intended for technically qualified and skilled customer representatives. It is the responsibility of the customer to evaluate the suitability of the Product for the intended application and the customer's specific use and to verify all relevant technical data contained in this document in the intended application and the customer's specific use. The customer is responsible for properly designing, programming, and testing the functionality and safety of the intended application, as well as complying with any legal requirements related to its use.
Unless otherwise explicitly approved by Infineon, Products may not be used in any application where a failure of the Products or any consequences of the use thereof can reasonably be expected to result in personal injury. However, the foregoing shall not prevent the customer from using any Product in such fields of use that Infineon has explicitly designed and sold it for, provided that the overall responsibility for the application lies with the customer.
Infineon expressly reserves the right to use its content for commercial text and data mining (TDM) according to applicable laws, e.g. Section 44b of the German Copyright Act (UrhG).
If the Product includes security features:
Because no computing device can be absolutely secure, and despite security measures implemented in the Product, Infineon does not guarantee that the Product will be free from intrusion, data theft or loss, or other breaches ("Security Breaches"), and Infineon shall have no liability arising out of any Security Breaches.
If this document includes or references software:
The software is owned by Infineon under the intellectual property laws and treaties of the United States, Germany, and other countries worldwide. All rights reserved.
Therefore, you may use the software only as provided in the software license agreement accompanying the software.
If no software license agreement applies, Infineon hereby grants you a personal, non-exclusive, non-transferable license (without the right to sublicense) under its intellectual property rights in the software (a) for software provided in source code form, to modify and reproduce the software solely for use with Infineon hardware products, only internally within your organization, and (b) to distribute the software in binary code form externally to end users, solely for use on Infineon hardware products. Any other use, reproduction, modification, translation, or compilation of the software is prohibited.
For further information on the Product, technology, delivery terms and conditions, and prices, please contact your nearest Infineon office or visit https://www.infineon.com.