PSoC 6 MCU Device Firmware Update Software Development Kit Guide

About this document

Scope and purpose

This guide provides comprehensive information on how to use the Device Firmware Update (DFU) Software Development Kit (SDK) to develop DFU systems for Cypress PSoC 6 MCU products. A detailed description of the SDK application programming interface (API), and build instructions for multiple bootloader code examples, are included.

Table of contents

About this document **1**
Table of contents **1**
1 Introduction **3**
2 What Is Device Firmware Update? **4**
2.1 Terms and Definitions 5
2.2 Using a DFU Module 6
2.3 Basic DFU Function Flow 6
2.4 Other Use Cases 6
3 DFU SDK Description **7**
3.1 DFU SDK Files 7
3.1.1 User Callback Functions 9
3.1.2 Linker Scripts 9
3.2 DFU Ecosystem 9
4 How to Use the SDK **11**
4.1 Determine the Applications in Your System 11
4.2 Locate Applications in Memory 12
4.3 Design the Applications 13
4.3.1 ModusToolbox Instructions 13
4.3.2 PSoC Creator Instructions 16
4.4 Build and Program the Applications 22
5 PSoC 6 MCU DFU Code Examples **23**
5.1 How to Build the Code Examples 24
5.2 PSoC 6 MCU Basic DFUs 25
5.2.1 ModusToolbox Instructions 25
5.2.2 PSoC Creator Instructions 28
5.3 PSoC 6 MCU BLE Bootloaders 34
5.3.1 BLE Bootloader with Upgradeable Stack 40
5.4 PSoC 6 MCU Dual-Application Bootloader 41
5.5 PSoC 6 MCU Encrypted Bootloader 44
6 Related Documents **47**
Appendix A. DFU Host Tool **49**
Appendix B. Host Command/Response Protocol **50**

B.1 Command/Response Packet Structure 50
B.2 Commands 50
B.2.1 Enter DFU 51
B.2.2 Sync DFU 52
B.2.3 Exit DFU 52
B.2.4 Send Data 52
B.2.5 Send Data Without Response 53
B.2.6 Program Data 53
B.2.7 Verify Data 54
B.2.8 Erase Data 54
B.2.9 Verify Application 55
B.2.10 Set Application Metadata 55
B.2.11 Get Metadata 56
B.2.12 Set EIVector 56
Appendix C. *.cyacd2* File Format 58
Appendix D. Application Metadata 59
Appendix E. Metadata Structure 61
Appendix F. Post-Build File Listings 62
F.1 ModusToolbox Post-Build Bash File 62
F.2 PSoC Creator Post-Build Batch File 63
Revision history 65

1 Introduction

This guide gives an overview of device firmware update (DFU, also called "bootloader") fundamentals, followed by a detailed description of the Cypress DFU Software Development Kit (SDK) and how to use it with PSoC 6 MCU.

Note: *The term “bootloader” has become overloaded in the industry and is frequently confused with the device startup (“boot-up”) process. Therefore, this application note, and the associated code examples are transitioning to “device firmware update”, or DFU, to reduce ambiguity.*

Multiple development tools are covered in this guide, including Cypress’ ModusToolbox™ integrated development environment (IDE) and Cypress' PSoC Creator IDE. ModusToolbox IDE is a free IDE for Windows®, macOS®, and Linux®. It provides a single, coherent, and familiar design experience for Internet of Things (IoT) designers, combining PSoC 6 MCU and the industry’s most deployed WiFi and Bluetooth technologies. For more information on ModusToolbox IDE, click **here.**

PSoC Creator is a free Windows-based IDE that enables concurrent hardware and firmware design of systems based on multiple Cypress PSoC MCU families. For more information on PSoC Creator, click **here**.

If you are new to DFU / bootloaders in general, basic concepts and design principles are explained in the next section,**What is Device Firmware Update?**

If you are familiar with DFU concepts and want to see how they are implemented in PSoC 6 MCU, see the sections **DFU SDK Description, How to Use the SDK**, and **PSoC 6 MCU DFU Code Examples.** For a list of the DFU code examples, see **Related Documents**. For a complete list of PSoC 6 MCU code examples, click **here**.

Note: *At the time of publication, UART-, I²C-, SPI-, and BLE-based DFU systems are supported. Other communication channels will be added in future SDK updates*

2 What Is Device Firmware Update?

Device firmware update (DFU) is a common part of MCU system design. DFU makes updating a product's firmware in the field possible. In a typical product, firmware is stored in an MCU's flash memory. The MCU is mounted on a printed circuit board (PCB) and embedded in a product, as **Figure 1** shows.

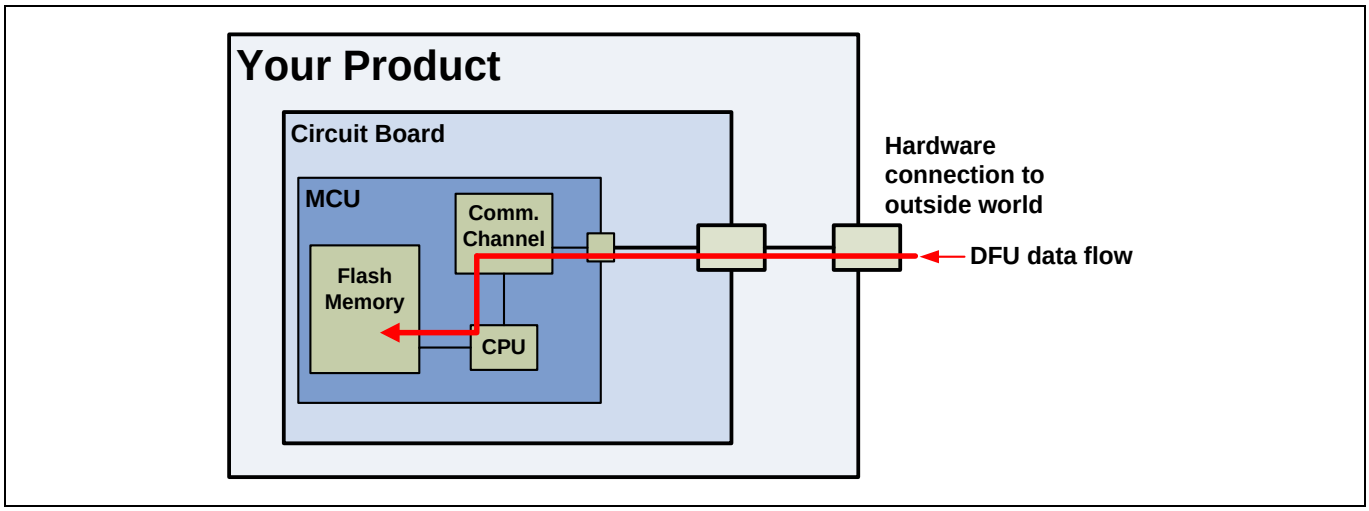

Figure 1 DFU Data Flow Block Diagram

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 better way to update firmware in the field is to use an existing connection between the product and the outside world. The connection may be a standard communication port such as I^2C, USB, or UART, a wireless channel such as BLE, or a custom protocol.

2.1 Terms and Definitions

Figure 1 implies that the product's embedded firmware must be able to use the communication port for two different purposes: normal operation and updating flash. The portion of the embedded firmware that knows how to update the flash is called a **DFU module, or bootloader**, as **Figure 2** shows.

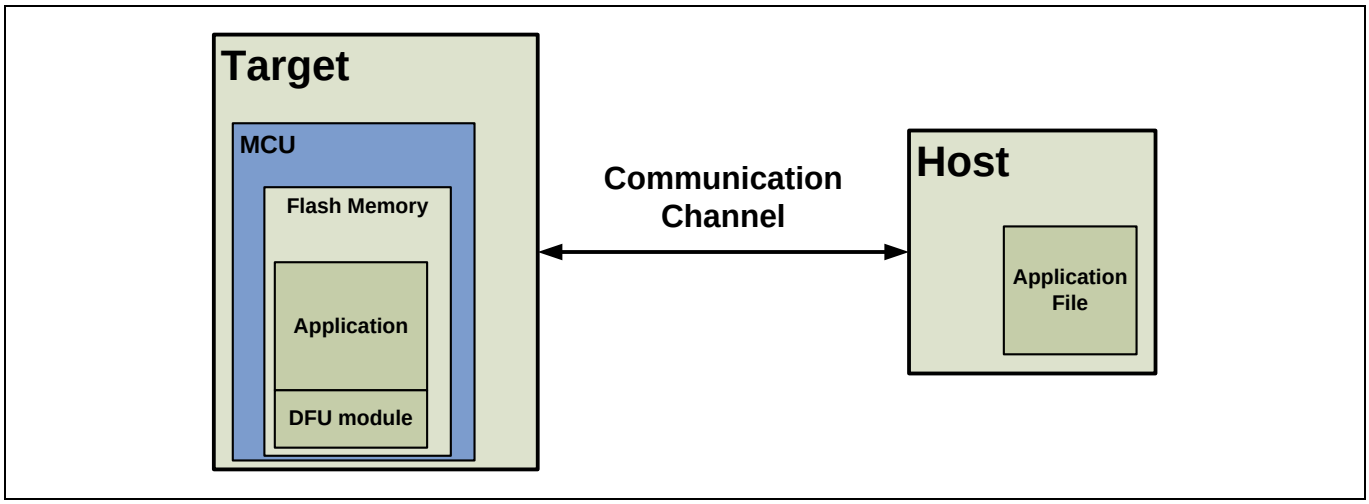

Figure 2 Generic DFU System

Typically, the system that provides the data to update the flash is called the **host**, and the system being updated is called the **target**. The host can be an external computer or another MCU on the same PCB as the target.

The act of transferring data from the host to the target flash is called **DFU**, or a **DFU** operation (In previous documentation, Cypress referred to the DFU as the bootloader. However, this term is frequently confused with the boot-up process. Therefore, in this application note and associated code examples, we refer to this as DFU to reduce ambiguity.) The data that is placed in flash is called the **application** or **firmware image**.

2.2 Using a DFU Module

The DFU module and the application typically share a communication port. The first step in DFU is to put the product in a mode where the DFU module, and not the application, is executing. This can be done in response to an event such as a button press or a received command. The application detects such an event and responds by transferring control to the DFU module.

After the DFU module is running, the host can send a “start DFU” command over the communication channel. If the DFU module sends an “OK” response, a DFU operation can begin.

2.3 Basic DFU Function Flow

During DFU, the host reads the file for the new application, parses it into flash program commands, and sends those commands to the DFU module. After the entire file is received and installed in target flash, the DFU module can pass control to the new application.

A DFU module typically executes first after device reset.[1]. It can then perform the following actions:

- Check the new application’s validity before transferring control to that application
- Manage the timing to start host communication
- Do the DFU operation
- Pass control to the new application

2.4 Other Use Cases

Other more complex DFU use cases exist. A common use case is for an application to be running and have some event alert the application that an update is available.

While the application continues to execute its normal tasks, it also downloads the new application into a temporary location. After the new application is checked for validity, it is copied into the correct location in flash, and control is transferred to it. **Figure 3** is a flow diagram that shows how this works.

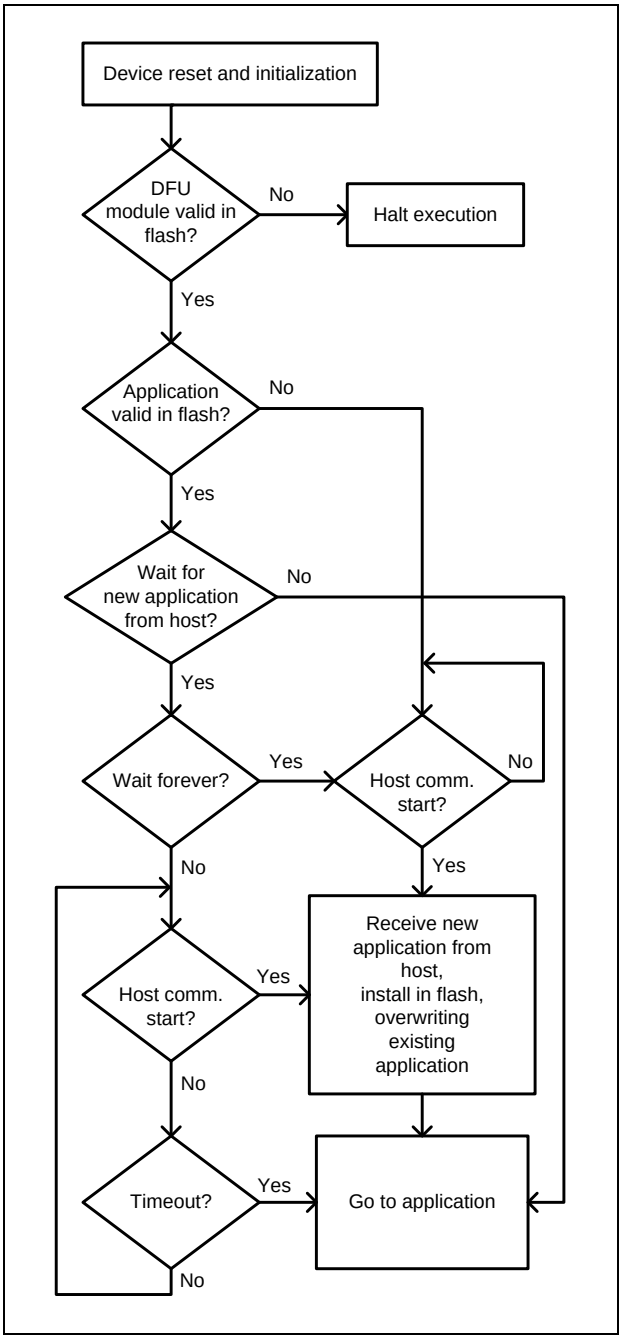

Figure 3 DFU Function Flow

1 In a typical MCU, several events may cause a device reset: for example, a device power up or a voltage level on a reset pin. In addition, firmware can trigger a reset by writing to a device register. This is known as a “software reset”, or SRES. Using the SRES feature, an application can reset the device, for example in response to an event such as a button press. This effectively transfers control to the DFU module because the DFU module executes first at device reset.

3 DFU SDK Description

The Cypress DFU SDK is an API consisting of a set of callable functions and other elements that enable rapid DFU system development. The SDK is shipped as a part of the Cypress PSoC 6 SDK. The SDK is typically found in *C:\ ... \ ModusToolbox_1.0 \ libraries \ psoc6sw-1.0 \ components \ psoc6mw \ dfu*. The SDK may be used by other IDEs as well as ModusToolbox IDE.

The SDK consists of the following:

- Source code (typically *.h* and *.c* files as well as linker scripts and *.mk* files) that implements the SDK API
- Documentation; i.e., this guide.
- Code examples

Note: *You may modify the DFU SDK source code for custom purposes, for example to modify or add commands to the host interface protocol (see **Appendix B: Host Command/Response Protocol)**.*

3.1 DFU SDK Files

Table 1 lists the files in the SDK.

Note: *PSoC 6 MCU has two Arm® CPUs: a Cortex®-M4 (CM4) and a Cortex-M0+ (CM0+). Linker script files are provided for each CPU, for multiple IDEs. For more information, see **AN215656**, PSoC 6 MCU Dual-CPU System Design.*

Table 1 DFU SDK Files

File	Description	
cy_dfu.h, .c	The DFU SDK files	
cy_dfu_bwc_macro.h	Enables backward compatibility with the Cypress legacy Bootloader SDK	
dfu_user.h	Contains user-editable `#define` statements that control the operation and the enabled features in the SDK	
dfu_user.c	Contains user functions required by the SDK: Five functions that control communications with the DFU host; these are also called “transport functions” Two functions – `ReadData ()` and `WriteData ()` – that control access to internal or external memory The functions provided are defaults and may be modified for your application	
transport_uart.h, .c *transport_... .h, .c*	Contains DFU transport functions for the host communication channel being used; these functions are typically called by the transport functions in dfu_user.c	
dfu_cm4.ld, *dfu_cm0p.ld*	Custom GCC linker scripts	In each application, these linker script files replace the default linker script files; these files locate the code and data sections for each of the CPUs as well as the DFU module and other regions.
dfu_cm4.scat, *dfu_cm0p.scat,* *dfu_mdk_common.h,* *dfu_mdk_symbols.c*	Custom MDK linker scripts; the common files exist to create necessary symbol definitions.	

File	Description	
dfu_cm4.icf, dfu_cm0p.icf	Custom IAR linker scripts	

The files are organized in the DFU folder as **Figure 4** shows:

```
\---dfu
    |   cy_dfu.h
    |   cy_dfu.c
    |   cy_dfu_bwc_macro.h
    \---config
    |       dfu_user.h
    |       dfu_user.c
    |       transport_uart.h
    |       transport_uart.c
    |       transport_... .h  transport files for other
    |       transport_... .c  communication channels
    \---linker_scripts
        \---GCC
        |       dfu_cm0p.ld
        |       dfu_cm4.ld
        \---MDK
        |       dfu_cm0p.scat
        |       dfu_cm4.scat
        |       dfu_mdk_common.h
        |       dfu_mdk_symbols.c
        \---IAR
                dfu_cm0p.icf
                dfu_cm4.icf
```

Figure 4 DFU SDK File Organization

3.1.1 User Callback Functions

The DFU SDK API functions call back to several user functions. This allows you to customize the following DFU operations:

- Host communication, also called transport functions or communication interface functions
- Reading and writing device's internal flash as well as other nonvolatile memory (NVM), for example external flash

Table 2 lists the required user callback functions. The DFU code examples show typical code for these functions; see **DFU Code Examples** and **Related Documents**.

Table 2 User Callback Functions

Function	Description
For Host Communication. Examples are in *dfu_user.c* and *transport_xxx.h* and *.c* files	
`Cy_DFU_TransportStart()`	Opens and initializes the communication channel
`Cy_DFU_TransportStop()`	Closes the communication channel
`Cy_DFU_TransportReset()`	Re-initializes the channel, typically to bring it back to a known state
`Cy_DFU_TransportRead()`	Receives a packet from the host; see **Host Packet Structure**
`Cy_DFU_TransportWrite()`	Sends a packet to the host; see **Host Packet Structure**
For Non-Volatile Memory (NVM) Access. Examples are in *dfu_user.c*	
`Cy_DFU_ReadData()`	Reads data from the device flash or other NVM
`Cy_DFU_WriteData()`	Writes data to the device flash or other NVM

3.1.2 Linker Scripts

A DFU system is a system with multiple applications (at least two) in NVM. You decide where in NVM each application resides. The DFU SDK includes template linker script files (see **Table 1** and **Figure 4**) that you can adapt to your application. The DFU code examples include example linker script files.

3.2 DFU Ecosystem

Other elements of the DFU system are not included in the SDK. They are available in ModusToolbox IDE, elsewhere in the PSoC 6 SDK, in the PSoC Creator IDE, or in the peripheral driver library (PDL), and can be used by other IDEs:

- CyMCUElfTool: *cymcuelftool.exe* in the folder *... \ ModusToolbox_1.0 \ tools \ cymcuelftool-1.0 \ bin*. This program is called as part of building an application, as **Figure 5** shows. (The diagram is similar for PSoC Creator and other IDEs.)
- This program combines core and application image files into output files. It is a command line program; its option syntax is documented in the Help command (`-h / --help`) output. A user guide is available in the folder *... \ cymcuelftool-1.0 \ doc*.
- Other drivers – for communication, flash, etc. – are in the folder *... \ ModusToolbox_1.0 \ libraries \ psoc6sw-1.0 \ components \ psoc6pdl \ drivers*. They are called as needed by the DFU SDK API functions.
- Device Firmware Update Host Tool: *dfuh-tool.exe* in the folder *... \ ModusToolbox_1.0 \ tools \ dfuh-tool-1.0*. This is an example of the "host" shown in **Figure 2**. For more information, see **Appendix A, Device Firmware Update Host Tool.**

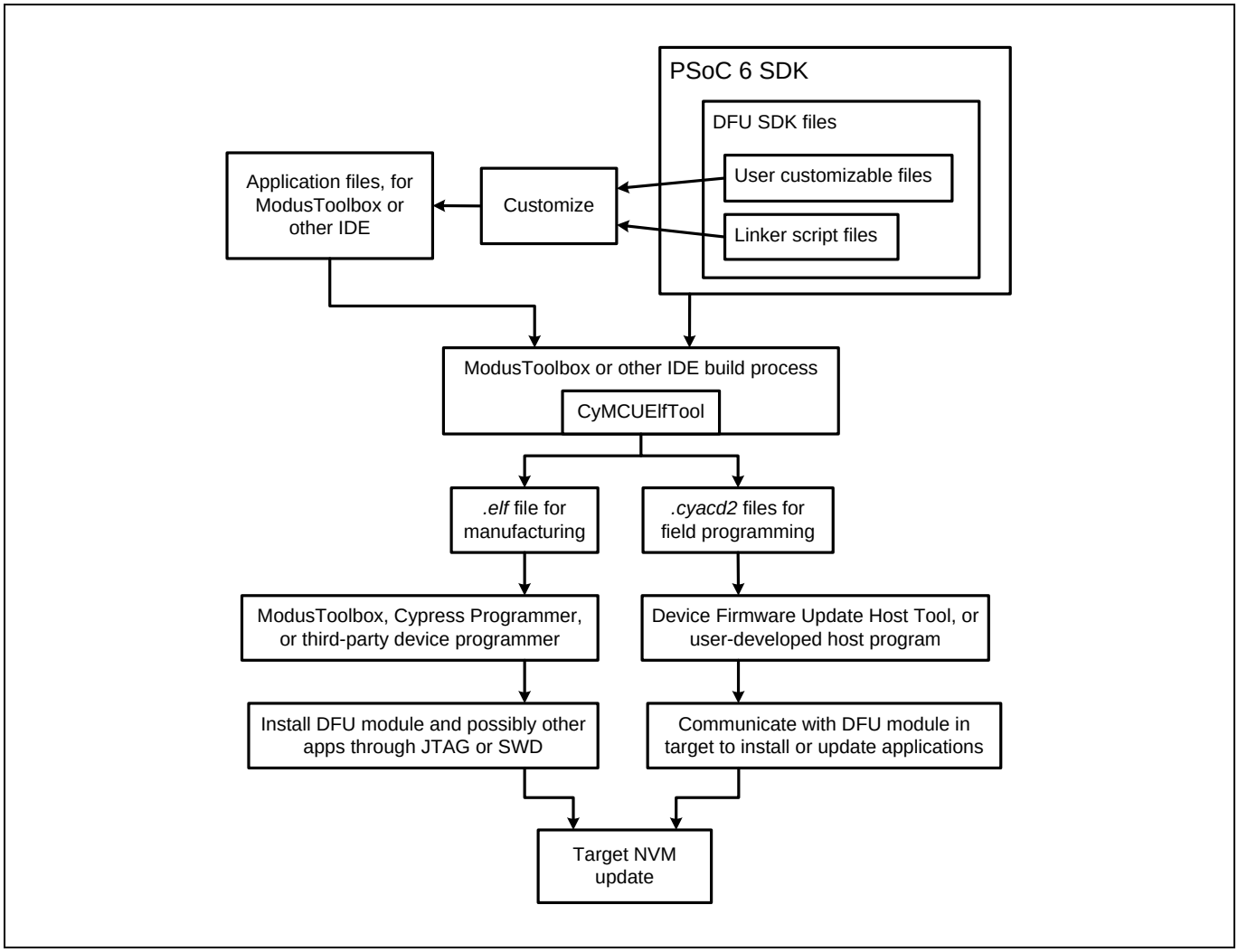

Figure 5 ModusToolbox Application Build and DFU Process Diagram

4 How to Use the SDK

This section describes the general process for using the DFU SDK. At a high level, the steps are Comparison of Devices

- **Determine the Applications in Your System**
- **Locate Applications in Memory**
- **Design the Applications**
- **Build and Program the Applications**

For detailed instructions on how to build the DFU SDK code examples, go to **DFU Code Examples**.

4.1 Determine the Applications in Your System

The first step is to decide how many applications are to be in your design, and the purpose of each. Which applications will have DFU capability? **Figure 6** shows some typical memory maps for multiple applications:

- Map **A** shows a basic case, where Application #0 downloads and installs Application #1.
- Map **B** shows a more complex case. There are three applications. All applications have DFU modules – they can download and install any other application. (In practice, Application #0 is usually not updated.) A general data section is located outside all applications.
- Map **C** shows the same case as map **B** except that one of the applications is located in external nonvolatile memory (NVM).
- In each of the maps, the DFU system uses a small amount of flash (typically one row) to store application metadata. For information, see **Appendix D, Application Metadata.**

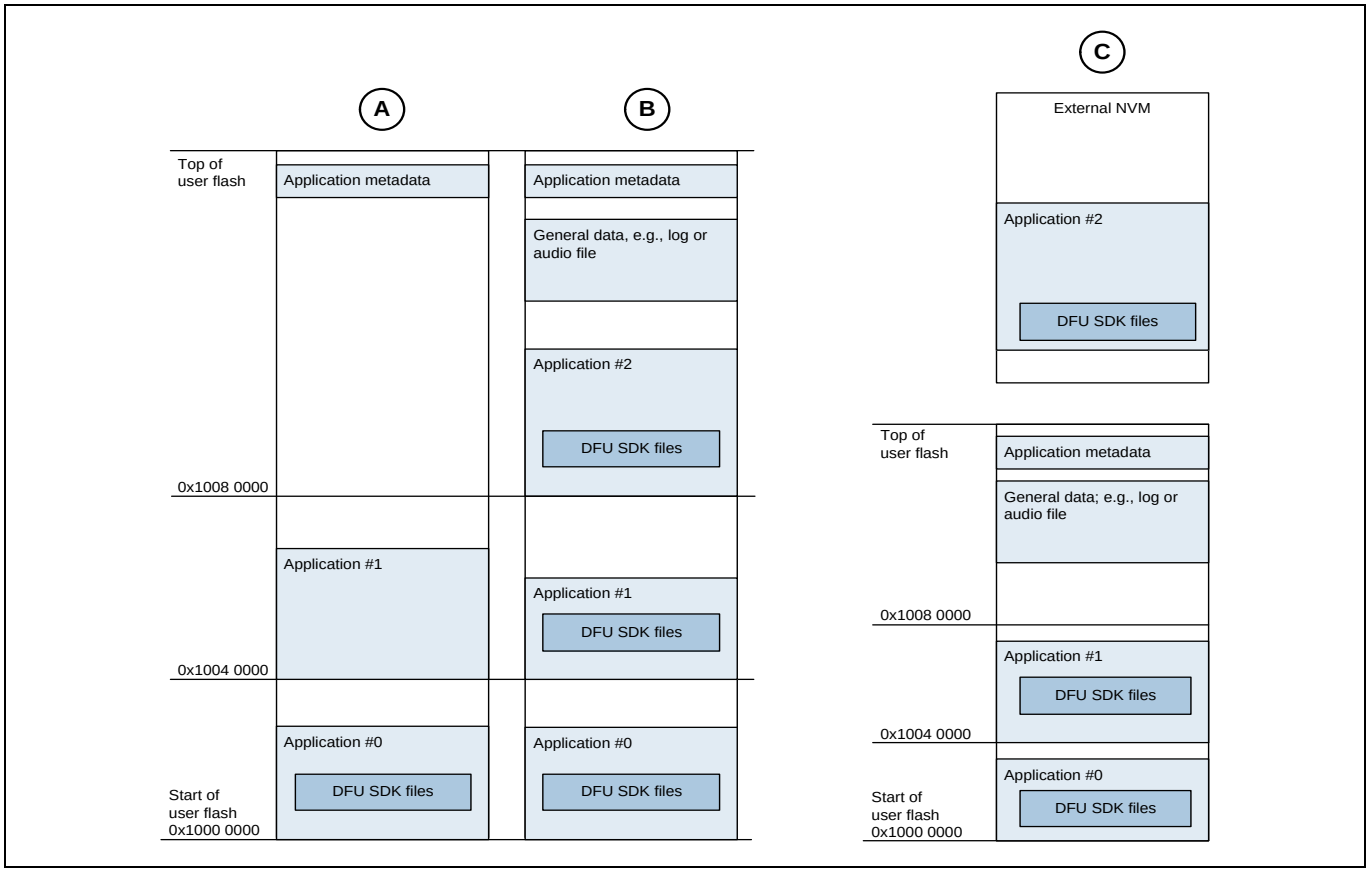

Figure 6 Example DFU Memory Maps

4.2 Locate Applications in Memory

The next step is to decide where each application is to be located in flash. There are several factors to consider:

- A maximum of 63 applications are supported, depending on the metadata size. For more information on DFU SDK metadata, see **Appendix D, Application Metadata.**
- An application may have code and data for one or more CPUs, for example CM4 and CM0+ in a PSoC 6 MCU device. For more information, see the application note **AN215656**, *PSoC 6 MCU Dual-CPU System Design*.
- The first application, usually called "App0", must be located at the beginning of user flash, which in PSoC 6 MCU is at address 0x1000 0000, as **Figure 6** shows. This is the first application to execute after device reset.[2]

Note: *Another module, such as a Secure Image module in a secure system, may occupy address 0x1000 0000, and App0 may be located elsewhere in flash. For more information, see **AN221111**, Creating a Secure System.*

- Any application may incorporate the DFU SDK, and thus be able to download and install another application, and/or transfer control to another application. In most cases, App0 should incorporate the DFU SDK.
- When writing flash in non-blocking mode, PSoC 6 MCU has a read-while-write (RWW) capability across 512-KB regions. That is, the DFU code may execute in one flash region while updating flash in another region. For more information, see the PDL flash driver documentation and the device **Technical Reference Manual (TRM).**
- Applications may reside in external memory as well as the device's internal flash, as map **C** in **Figure 6** shows. External memory is in a 128-MB sub-region from 0x1800 0000 to 0x1FFF FFFF.

Note: *When executing out of external memory, the external memory device must be mapped into the PSoC 6 MCU using the QSPI block in XIP mode. For more information, see the **device Technical Reference Manual (TRM).***

- PSoC 6 MCU applications include validation bytes, which are saved in NVM, as part of the application. This enables an application to be validated before transferring control to it.

Note: *Depending on the security environment within which the applications exist, an application can be structured for simple validation (with checksum or CRC data) as well as authorization (with encrypted signature data). The DFU system can either directly implement validation or call a function in the Flash Boot module to check authorization. For more information, see **AN221111**, Creating a Secure System.*

- NVM data that may be updated during the normal workflow, for example an events log, should not be part of an application, because it makes the application difficult to validate. Instead, designate a region of NVM that is outside any application's space and use it for data storage. **Figure 6** shows two examples in maps **B** and **C**.
- Reserve one row of flash for application metadata, as **Figure 6** shows. (In rare cases more than one row may be required.) Application metadata is managed by the DFU SDK and is used by applications to transfer control to another application. Application metadata space should be outside any application space.

[2] The CM0+ CPU executes SROM and other system-level code first at device reset, then transfers control to the application residing at 0x1000 0000. For more information, see the Technical Reference Manual (TRM).

4.3 Design the Applications

4.3.1 ModusToolbox Instructions

*Note: This section contains only DFU-specific instructions. For detailed step-by-step instructions for creating a ModusToolbox application, see ModusToolbox Help; **AN221774**, Getting Started with PSoC 6 MCU; or **DFU Code Examples** in this document.*

The next step is to design each of the applications identified in the **previous steps**. Each application is a single ModusToolbox application, independent from any other application. With ModusToolbox IDE, you can have all the applications in one workspace, or in separate workspaces, as well as in separate locations on your computer. Before getting started with PSoC 6 MCU, developing a plan for workspaces and applications for your overall system development needs is recommended.

Note: A ModusToolbox application for PSoC 6 MCU (dual-CPU) is composed of five projects:

- <project name>_config: the device configuration file *design. modus*, and device configuration source files
- <project name>_mainapp: source files for the Cortex-M4 (CM4) CPU
- <project name>_mainapp_cm0p: source files for the Cortex-M0+ (CM0+) CPU
- <project name>_mainapp_psoc6pdl: driver files for the Cortex-M4 (CM4) CPU
- <project name>_mainapp_cm0p_psoc6pdl: driver files for the Cortex-M0+ (CM0+) CPU

Do the following for each ModusToolbox application that is to do DFU operations and/or be an installable application:

- If the application is to do DFU operations, enable a communication peripheral in the *design.modus* file. That peripheral implements the communication channel to the DFU host.
- The DFU SDK includes support for many communication channel types, including UART, SPI, I²C, and BLE. Code examples are available for each channel type; see **DFU Code Examples** and **Related Documents**.
- You can also create a custom communication channel. The driver must implement the transport functions described in **User Callback Functions**.
- Incorporate the DFU SDK into the project, as **Figure 7** shows. Right-click the _mainapp project in the Project Explorer window and click **ModusToolbox Middleware Selector**.

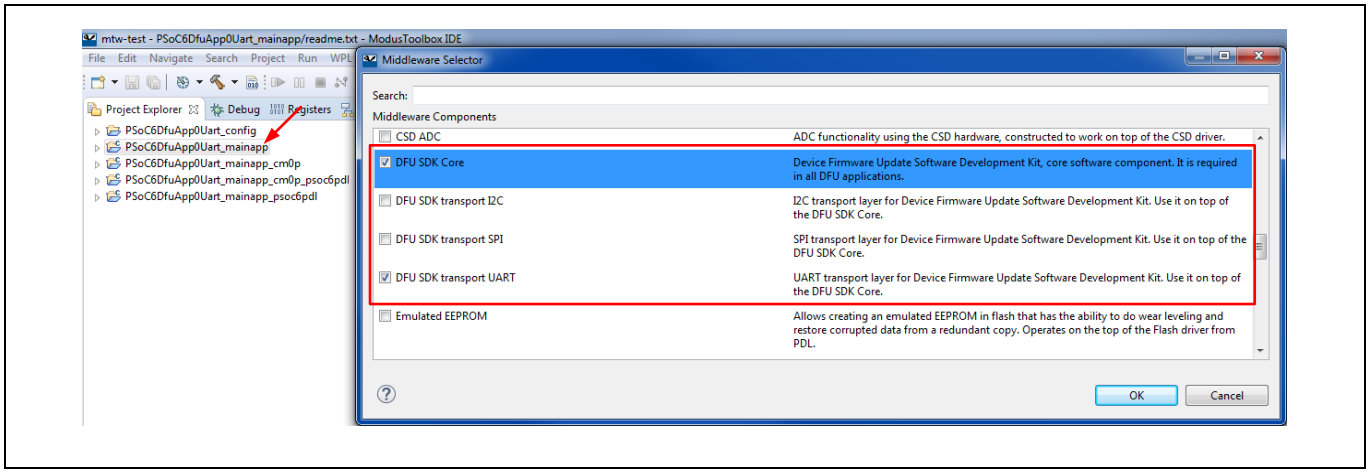

Figure 7 Incorporate DFU SDK into a ModusToolbox Project

If the application will do DFU operations, select **DFU SDK Core** and a **DFU SDK transport xxx** box.

Note: *In most cases, only one transport box is selected. However, if you want to have a custom communication channel, you can leave all transport boxes unselected, or even select multiple boxes.*

If the application is a downloadable application and will transfer control to another application, select **DFU SDK Core**. The code to transfer control to another application is included in the DFU SDK core files.

Click **OK** when done. Required source *.c*, *.h*, and linker script (e.g., *.ld*) files are automatically added to the project.

Repeat this step for the _mainapp_cm0p project; but select only the **DFU SDK core** box.

- Installed linker script files are by default set up for Application #0 (App0). For other applications, edit the files by changing the application number. The following example shows edits for downloadable application App1, for the GCC linker, in *dfu_cm0p.ld* and *dfu_cm4.ld*:

```
/*
* DFU SDK specific: aliases regions, so the rest of the code does not use
* application-specific memory region names
*/
REGION_ALIAS("flash_core0", flash_app1_core0);
REGION_ALIAS("flash",       flash_app1_core1);
REGION_ALIAS("ram",           ram_app1_core1);

/* DFU SDK specific: sets an app Id */
__cy_app_id = 1;
```

- Change the project properties to use the DFU linker script files instead of the default linker script files. Select the *dfu_cm4.ld* file for _mainapp, as **Figure 8** shows. Include the relative path to the file. Select the *dfu_cm0p.ld* file for _mainapp_cm0p (not shown in the figure).

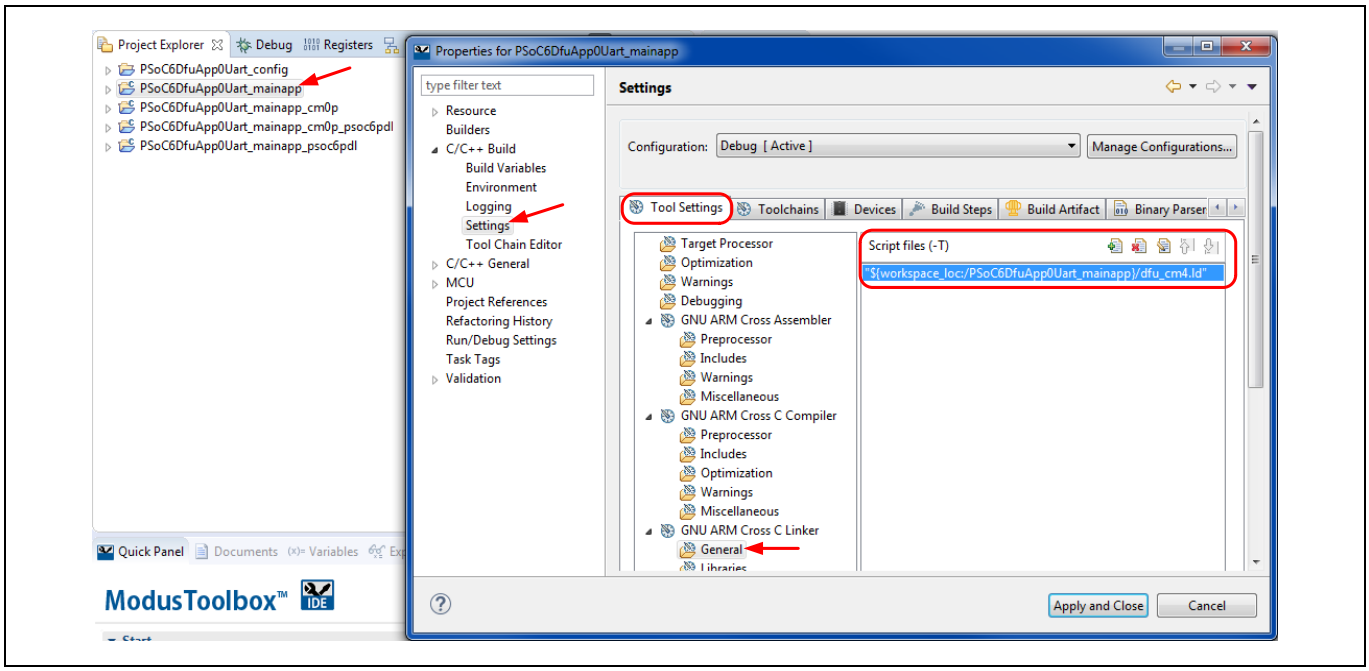

Figure 8 ModusToolbox Project Properties Setting for Custom Linker Script

- For downloadable application projects, add a post-build command to call the Cypress utility program *cymcuelftool.exe*, as **Figure 9** shows. **cymcuelftool** is included with your ModusToolbox installation. It generates a **.cyacd2* file, which is downloaded by the DFU host (see **Figure 5**). The post-build command is done only for the CM4 binary. cymcuelftool combines the CM4 (__mainapp) and CM0+ (mainapp_cm0p) projects to create the output *.cyacd2* file.

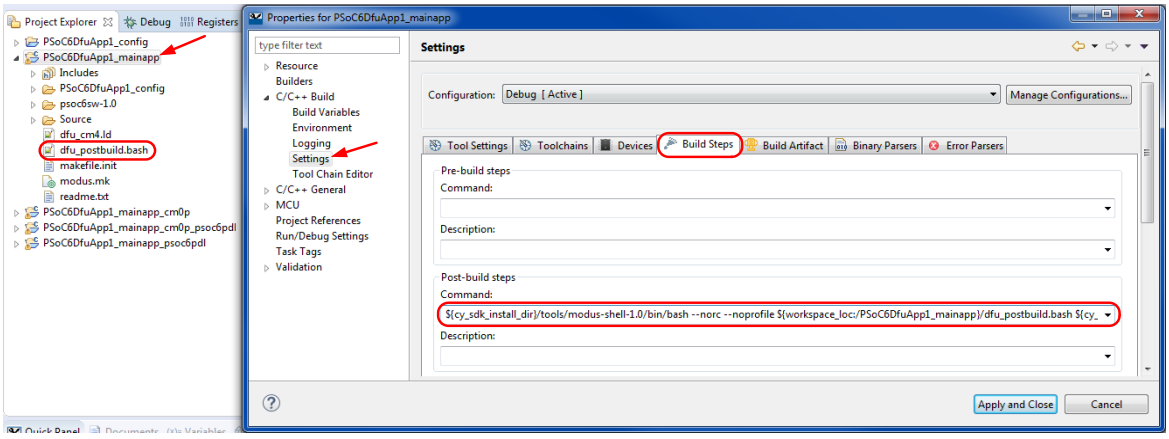

Figure 9 ModusToolbox Post-Build Command

The command typically invokes a bash file such as *dfu_postbuild.bash* in code example **CE213903**, *PSoC 6 MCU Basic DFU*. For convenience, copy and paste the file into your _mainapp project.

Then change the _mainapp project properties to add command text similar to the following, as **Figure 9** shows.

```
${cy_sdk_install_dir}/tools/modus-shell-1.0/bin/bash --norc --noprofile
${workspace_loc:/PSoC6DfuApp1_mainapp}/dfu_postbuild.bash
${cy_sdk_install_dir}/tools/cymcuelftool-1.0/bin/cymcuelftool
${workspace_loc:/PSoC6DfuApp1_mainapp_cm0p}/${config_name:PSoC6DfuApp1_mainap
p_cm0p}/PSoC6DfuApp1_mainapp_cm0p.elf
${workspace_loc:/PSoC6DfuApp1_mainapp}/${config_name:PSoC6DfuApp1_mainapp}/PS
oC6DfuApp1_mainapp.elf ARM_CM4
```

- Add code as needed to the *main.c* and other source files in both the CM4 (__mainapp) and CM0+ (mainapp_cm0p) projects. Specific requirements are:
  - Add a `#include "cy_dfu.h"` statement to all source files as needed to access the DFU SDK API.
  - For downloadable applications, in *main.c* in the __mainapp_cm0p project, change the `Cy_SysEnableCM4() function call to:` `Cy_SysEnableCM4((uint32_t)(&__cy_app_core1_start_addr));`
  - For downloadable applications, in *main.c* in the __mainapp project, add the following global statement. Adjust the array size for the selected signature type.

```
/* This section holds signature data for application verification. */
CY_SECTION(".cy_app_signature") __USED static const uint32_t
cy_bootload_appSignature[1];
```

In your overall code in both main files, consider the following:

- Which DFU function(s) to call:
  - `Cy_DFU_Complete()`: blocks while doing the entire DFU operation. Call this function if there is no other task to do during DFU.

  - `Cy_DFU_Init()` followed by a series of calls to `Cy_DFU_Continue()`: `Cy_DFU_Continue()` blocks while receiving, processing, and responding to one command packet from the host. Call these functions if other tasks must be done during DFU.
- Whether each application shall pass control to another application. Add calls to DFU SDK API functions `Cy_DFU_ValidateApp()` and `Cy_DFU_ExecuteApp()` as needed.

For more information, see the PDL DFU SDK documentation.

Note: *You can copy and paste code from the code examples listed in **DFU Code Examples** and **Related Documents**.*

4.3.2 PSoC Creator Instructions

Note: *This section contains only bootloader-specific (in ModusToolbox IDE, this has been changed to DFU operation) instructions. For detailed step-by-step instructions for creating a PSoC Creator project, see PSoC Creator Help; **AN210781**, Getting Started with PSoC 6 MCU with Bluetooth Low Energy (BLE) Connectivity; or **How to Build the Code Examples** in this document.*

The next step is to design each of the applications identified in the **previous steps**. Each application is a single PSoC Creator project, independent from any other project. With PSoC Creator, you can have all projects in one workspace (*.cywrk* file), or in separate workspaces as well as in separate locations on your computer. Before getting started with PSoC 6 MCU, developing a plan for workspaces and projects for your overall system development needs is recommended.

Do the following for each PSoC Creator project that is to be a bootloader or an installable application:

- If the application is to do bootloading, place a communication Component on the project schematic; i.e., the *TopDesign.cysch* file. This Component implements the communication channel to the bootloader host. Connect the Component terminals to the appropriate physical pins.
- The Bootloader SDK includes support for many of the communication Components in the PSoC Creator Component Catalog, including UART, SPI, I^2C, and BLE. Code examples are available for each communication Component; see **Bootloader Code Examples** and **Related Documents**.
- For compatibility with the default transport files in the SDK, name the Component as **UART**, **I2C**, **SPI**, or **BLE**.
- You can also create a custom communication channel. The driver must implement the transport functions described in **User Callback Functions**.
- Incorporate the Bootloader SDK into the project, as **Figure 10** shows. Right-click the project in the Workspace Explorer window and select **Build Settings...**. In the **Build Settings** dialog, select **Peripheral Driver Library**.

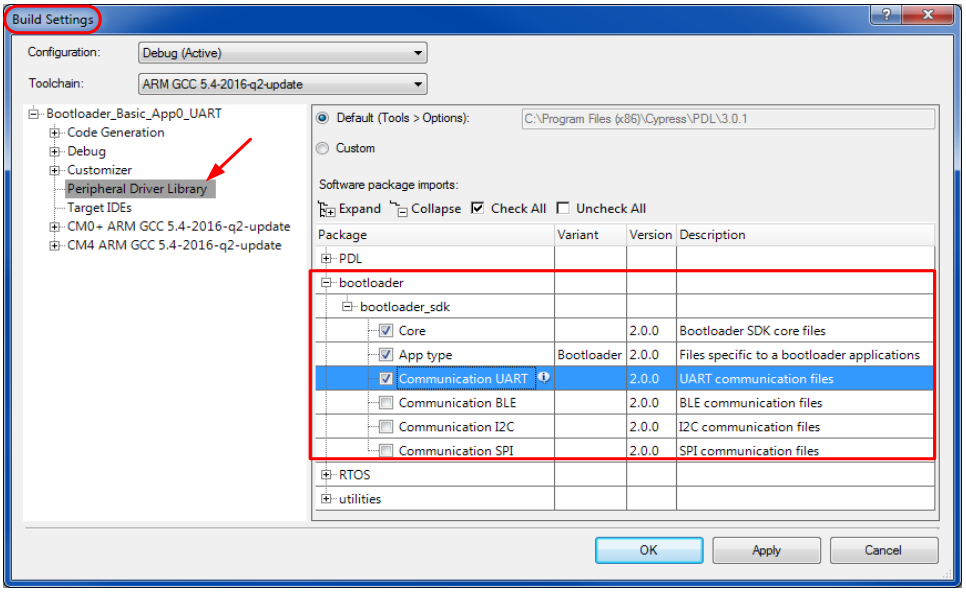

Figure 10 Incorporate Bootloader SDK into a PSoC Creator Project

If the application will do bootload operations, select **Core**, **App type bootloader,** and a **Communication** box.

In most cases, only one communication box is selected. However, if you want to have a custom communication channel, you can leave all communication boxes unselected, or even select multiple boxes.

If the application is a downloadable application and will transfer control to another application, select **Core**. The code to transfer control to another application is included in the SDK core files.

Click **OK** when done. Required source *.c*, *.h*, and linker script (e.g., *.ld*) files are automatically added to the project.

- Generate the project files. Click the project in the Workspace Explorer window and select **Build** > **Generate Application**. The files are added to the project, as **Figure 11** shows.
- Note that PSoC 6 MCU has two Arm CPUs: CM4 and CM0+. Each CPU has its own folder, plus Generic folders and a shared files folder. Linker script files to locate the code for each CPU, for multiple IDEs, are automatically copied and added to the project. For more information, see **AN215656**, *PSoC 6 MCU Dual-CPU System Design*.
- **Figure 11** shows bootloader files added to the CM0p folder. Similar files are also added to the CM4 folder but are not shown.
- Edit the files *bootload_user.h* and *bootload_user.c*. Review the **user callback functions** in *bootload_user.c*, and edit them for any customization that is needed, for example, to write and read external memory. Usually the functions can be left unchanged.

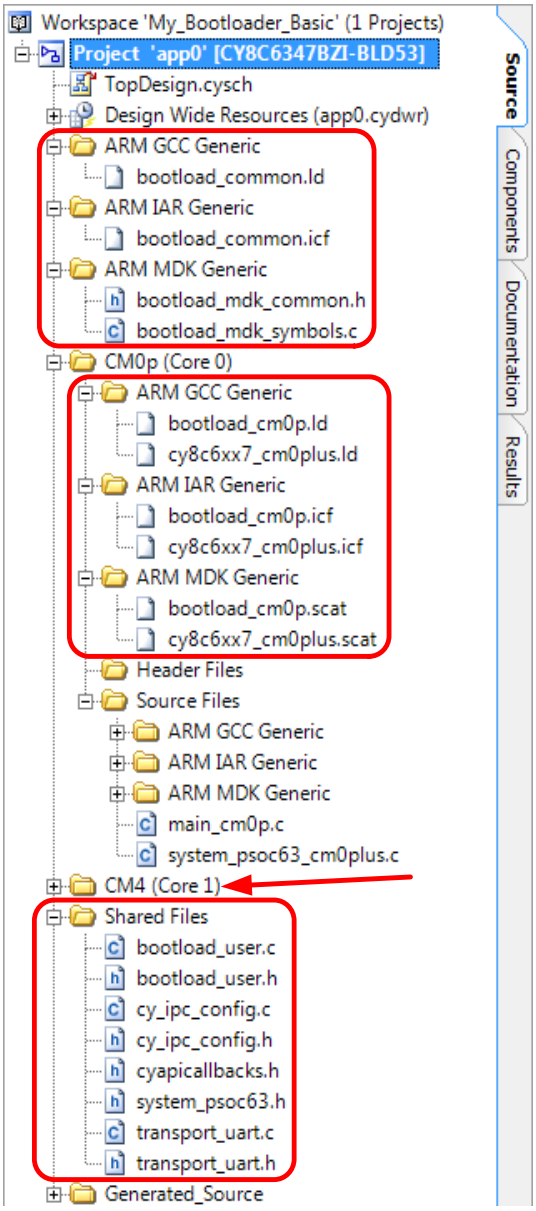

Figure 11 . Add SDK Files to a PSoC Creator Project

Note: *The* bootload_user.c *file that is added contains the default code for UART. For other communication channels such as I²C and SPI, edit* bootload_user.c *as follows:*

- Change:
  ```
  #include "transport_uart.h"
  ```
  to:
  ```
  #include "transport_i2c.h"
  ```
  or:
  ```
  #include "transport_spi.h"
  ```
  or:
  ```
  #include "transport_ble.h"
  ```
- Change five instances of "`UART_Uart`" to "`I2C_I2c`", "`SPI_Spi`", or "`CyBLE`".

- Depending on the compiler you are using, edit the appropriate common linker script file, to encode the decisions made in **Locate Applications in Memory**. For an example, see **Section 5.3 Step 9**.
- Installed linker script files are by default set up for application #0 (app0). For other applications, edit the files by changing the application number. The following example shows edits for app1, for the GCC linker, in *bootload_cm0p.ld* and *bootload_cm4.ld*:

```
  /*
  * Bootloader SDK-specific: aliases regions, so the rest of the code does
  not
  * use application-specific memory region names
  */
  REGION_ALIAS("flash_core0", flash_app1_core0);
  REGION_ALIAS("flash",       flash_app1_core1);
  REGION_ALIAS("ram",           ram_app1_core1);

  /* Bootloader SDK-specific: sets App ID */
  __cy_app_id = 1;
```

The Keil µVision MDK linker is more complex. The following example shows edits for app1 in *bootload_cm0p.scat* and *bootload_cm4.scat*:

```
; Flash
#define FLASH_START             CY_APP1_CORE0_FLASH_ADDR
#define FLASH_SIZE              CY_APP1_CORE0_FLASH_LENGTH

; Emulated EEPROM Flash area
#define EM_EEPROM_START         CY_APP1_CORE0_EM_EEPROM_ADDR
#define EM_EEPROM_SIZE          CY_APP1_CORE0_EM_EEPROM_LENGTH

; External memory
#define XIP_START               CY_APP1_CORE0_SMIF_ADDR
#define XIP_SIZE                CY_APP1_CORE0_SMIF_LENGTH

; RAM
#define RAM_START               CY_APP1_CORE0_RAM_ADDR
#define RAM_SIZE                CY_APP1_CORE0_RAM_LENGTH
```

And edits for App1 in *bootload_mdk_symbols.c*:

```
__cy_app_core1_start_addr   EQU __cpp(CY_APP1_CORE1_FLASH_ADDR)

/* Application number (ID) */
__cy_app_id                 EQU 1

/* CyMCUElfTool uses these to generate an application signature */
__cy_app_verify_start     EQU __cpp(CY_APP1_CORE0_FLASH_ADDR)
__cy_app_verify_length    EQU __cpp(CY_APP1_CORE0_FLASH_LENGTH +
                                    CY_APP1_CORE1_FLASH_LENGTH -
                                    __CY_BOOT_SIGNATURE_SIZE)
```

- Change the Project Build settings to use the bootloader linker script files instead of the default linker script files. Select the *bootload_cm0p.ld* file for the CM0+ CPU, as **Figure 12** shows. Include the relative path to the file. Select the *bootload_cm4.ld* file for the CM4 CPU (not shown in the figure).

Note: *Linker script files are provided for GCC, Keil µVision MDK, and IAR Embedded Workbench linkers. The example shown is for GCC. Use the appropriate linker script for your IDE.*

Note: This operation should be done for both Debug and Release configurations.

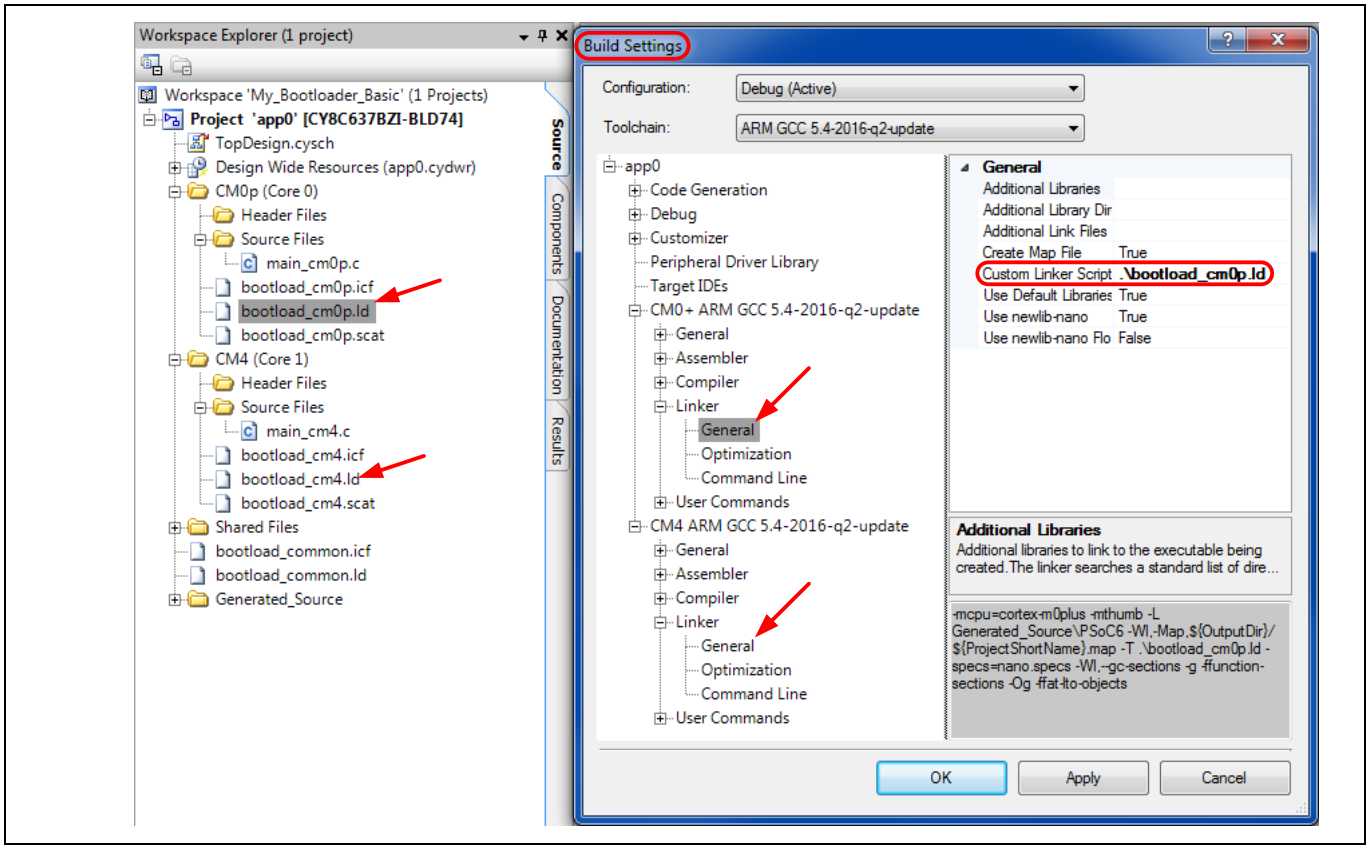

Figure 12 PSoC Creator Project Build Settings for Custom Linker Scripts

- For downloadable application projects, add a post-build batch file to call the Cypress utility program *cymcuelftool.exe*. **cymcuelftool** is included with your PSoC Creator installation. It generates a **.cyacd2* file, which is downloaded by the bootloader host (see **Figure 5**). The batch file is applied only to the CM4 binary.

Note:

1. *See a Bootloader SDK code example such as **CE213903**, PSoC 6 MCU Basic Bootloaders, for an example of a batch file. It is usually found in folder* CE213903 \ Bootloader_Basic_App1.cydsn*. The batch file contents are also available in **Appendix E**. For convenience, copy and paste the file into your PSoC Creator project folder* app1.cydsn*.*
2. *Then in PSoC Creator, in the Workspace Explorer window, add the batch file to the project's Shared Files folder, and add that file as a post-build command to the Cortex-M4 build, as **Figure 13** shows.*
3. *This operation must be done for both Debug and Release configurations.*

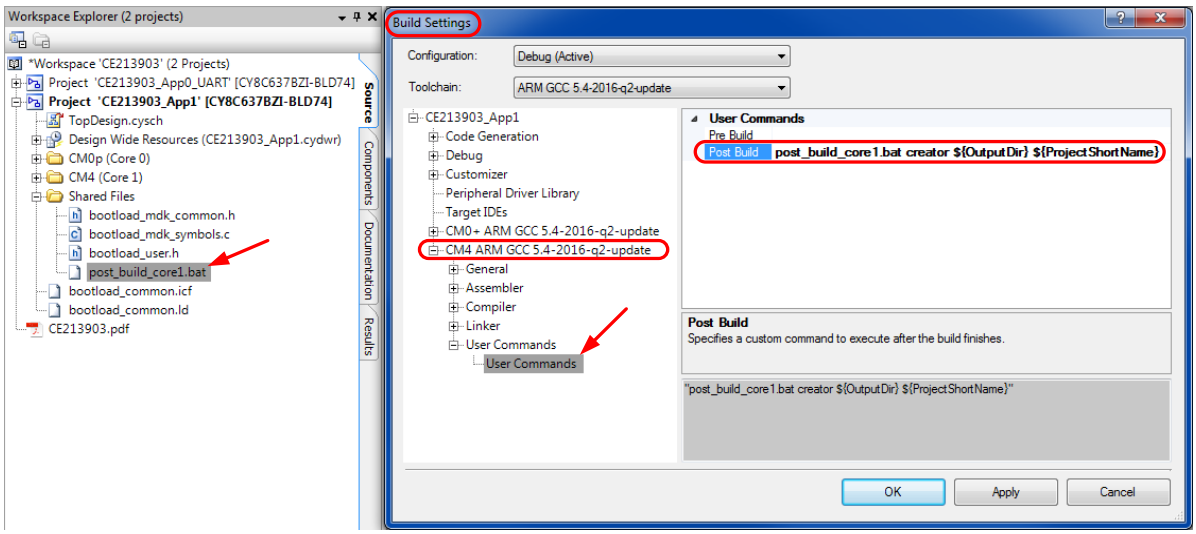

Figure 13 PSoC Creator Post-Build Command

Add code as needed to the two *main_cmx.c* and other source files. For downloadbale application projects, add to the *main_cm4.c* file a global array to store the application signature. For example, for checksum, the simplest and most common signature type, add the following:

```
/* This section holds signature data for application verification.
   For checksum verification, set the number of elements in the array to 1,
     and
   in bootload_common.ld set __cy_boot_signature_size = 4.
*/

     CY_SECTION(".cy_app_signature") __USED static const uint32_t
                                     cy_bootload_appSignature[1];
```

In your overall code in both main files, consider the following:

- Which bootloader function(s) to call:
  - `Cy_Bootload_DoBootload()`: blocks while doing the entire bootload operation. Call this function if there is no other task to do while bootloading.
  - `Cy_Bootload_Init()` followed by a series of calls to `Cy_Bootload_Continue()`: `Cy_Bootload_Continue()` blocks while receiving, processing, and responding to one command packet from the host. Call these functions if other tasks must be done while bootloading.
- Whether each application shall pass control to another application. Add calls to Bootloader SDK API functions `ValidateApplication()` and `ExecuteApplication()` as needed.

For more information, see the PDL Bootloader documentation.

Note: *You can copy and paste code from the code examples listed in **Bootloader Code Examples** and **Related Documents**.*

4.4 Build and Program the Applications

Build all the applications. Several output files are created, as **Table 3** shows for ModusToolbox IDE. PSoC Creator output files are similar. (See also **Figure 5**). All output files are in the Debug or Release folder of their respective projects.

Table 3 DFU Output Files for ModusToolbox IDE (Example from CE213903, PSoC 6 Basic DFU)

File Name	Description
App0 with DFU module, for ModusToolbox IDE	
PSoC6DfuApp0xxx_mainapp_cm0p.elf	Linker output, CM0+ project
PSoC6DfuApp0xxx_mainapp_cm0p_signed.elf	cymcuelftool output after the signing process, CM0+ project
PSoC6DfuApp0xxx_mainapp.elf	Linker output, CM4 project
PSoC6DfuApp0xxx_mainapp_signed.elf	cymcuelftool output after the signing process, CM4 project
PSoC6DfuApp0xxx_mainapp_final.elf	cymcuelftool final output, combining CM4 and CM0+ outputs. This is the file that is programmed to the target device.
App1, the downloadable application, for ModusToolbox IDE	
PSoC6DfuApp1_mainapp_cm0p.elf	Linker output, CM0+ project
PSoC6DfuApp1_mainapp_cm0p_signed.elf	cymcuelftool output after the signing process, CM0+ project
PSoC6DfuApp1_mainapp.elf	Linker output, CM4 project
PSoC6DfuApp1_mainapp_signed.elf	cymcuelftool output after the signing process, CM4 project
PSoC6DfuApp1_mainapp_final.elf	cymcuelftool final output, combining CM4 and CM0+ outputs.
PSoC6DfuApp1_mainapp_dfu.elf	cymcuelftool final output, signed with CRC bytes. See *dfu_postbuild.bash*.
PSoC6DfuApp1_mainapp_dfu.cyacd2	cymcuelftool downloadable output. See *dfu_postbuild.bash*. This is the input to the Device Firmware Update Host Tool.

Program the *App0..._mainapp_final.elf* file into the device, using either ModusToolbox IDE or PSoC Creator. For more information on device programming, see the Help menus in these tools.

At a later time, you can use the DFU module in Application #0, along with a host program, to download and install application. cyacd*2* files into the device. See **Appendix A. DFU Host Tool**.

5 PSoC 6 MCU DFU Code Examples

There are several code examples associated with this application note. They demonstrate the different ways that **DFU operations** can be done.

A complete list of code examples and other documents, with download links on **www.cypress.com**, is available in **Related Documents**.

Table 4 shows an overview-level list of the code examples:

Table 4 List of PSoC 6 MCU DFU Code Examples

CE #	Title	Description
Supports ModusToolbox IDE and PSoC Creator		
CE213903	PSoC 6 MCU DFU Basic	A set of examples that demonstrate basic DFU operations; see **Error! Reference source not found.**, memory map A: Downloading an application from a host, using various communication channels: UART, I2C, and SPI Installing the downloaded application into user flash Validating an application, and then transferring control to that application
Supports PSoC Creator only		
CE216767	PSoC 6 MCU with Bluetooth Low Energy (BLE) Connectivity Bootloader	Same as CE213903, but uses BLE as the communication channel
CE220959	PSoC 6 MCU BLE Bootloader with External Memory	Same as CE216767, but saves the application temporarily in external memory, then copies it to its final destination in user flash
CE220960	PSoC 6 MCU BLE Bootloader with Upgradeable Stack	Same as CE216767; in addition the BLE stack can be upgraded
CE221984	PSoC 6 MCU Dual-Application Bootloader	Same as CE213903, but demonstrates bootloading two applications, with a factory default (“golden image”) mode; the communication channel is I2C
CE222802	Bootloader with Encryption and Signing	Same as CE213903, but the application is encrypted and signed for validation; the bootloader decrypts the application and validates its signature; the communication channel is UART
The following code examples are planned or are in development:		
CE2xxxxx	USB HID DFU	Same as CE213903, but uses USB HID as the communication channel

CE #	Title	Description
Supports ModusToolbox IDE and PSoC Creator		
CE2xxxxx	USB Mass Storage DFU	Same as CE213903, but uses USB Mass Storage as the communication channel
CE2xxxxx	DFU with Multiple Applications	Same as CE221984; each application can update another application; see **Figure 6**, memory map B
CE2xxxxx	PSoC 6 MCU DFU with External Memory Source	Same as CE213903, but the DFU source is external memory
CE2xxxxx	Embedded Host Program	Similar to that described in **AN60317**, PSoC 3 and PSoC 5LP I2C Bootloader

Most of the code examples consist of multiple separate applications, called “App0” and “App1”. In some cases, additional applications, called “App2”, “App3”, and so on, are included. Each application is a separate application in ModusToolbox IDE or a separate project in PSoC Creator.

Generally, all applications are in the same ModusToolbox or PSoC Creator workspace. As noted in **Design the Applications,** projects can exist in separate workspaces as well as in separate locations on your computer.

Usually App0 does the DFU operation; it downloads and installs the other applications. One notable exception is **CE220960**, BLE Bootloader with Upgradeable Stack – in that code example App1 downloads and installs App2; App0 just transfers control to one of the other applications.

The basic code example, **CE213903**, has multiple '_App0_ ... ' projects; each project has a different communication channel. Advanced communication channels such as BLE and USB are demonstrated in other code examples.

The code examples support the **CY8CKIT-062-BLE PSoC 6 BLE Pioneer Kit.** Applications typically blink a kit LED at different rates or in different colors, making it easy to tell which application is currently running. Porting a code example to another kit or to your system is a straightforward task.

In general, the applications are designed such that you can transfer control from one application to another, by holding a kit button down for 0.5 second. The BLE bootloader code examples use the Immediate Alert Service (IAS) to transfer control between applications.

5.1 How to Build the Code Examples

The following are step-by-step instructions showing how to build each DFU code example listed in**Table 4**, with optional adaptations for your application. For more information, see **How to Use the SDK**.

Note: *When building the code examples, in many cases it is easiest to copy portions of the existing code example into your project. The copied portions can then be modified for your application. The portions to be copied are listed in each set of instructions.*

The instructions are based on Cypress's ModusToolbox and PSoC Creator IDEs; they can be adapted for other IDEs.

5.2 PSoC 6 MCU Basic DFUs

This section shows how to build the basic DFUs in code example **CE213903**. There are five general steps:

- Create App0 and the workspace.
- Configure App0 as a DFU.
- Add App1.
- Configure App1 as an installable application.
- Build and test the DFU and the application.

These steps apply to both ModusToolbox IDE and PSoC Creator:

5.2.1 ModusToolbox Instructions

1. Create App0 and the workspace.

Start ModusToolbox IDE, and select or create the workspace for your applications, as **Figure 14** shows:

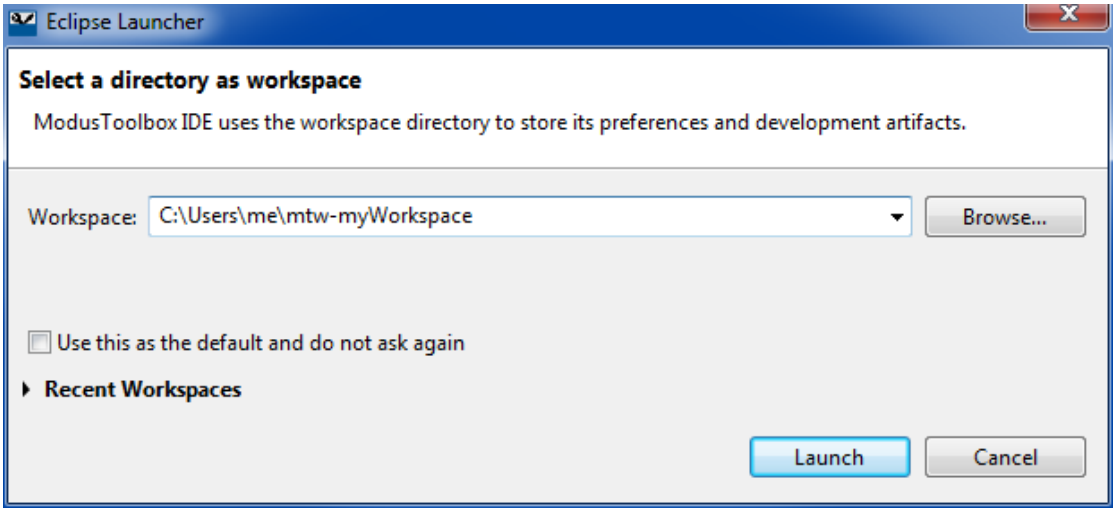

Figure 14 Define a ModusToolbox Workspace

Create a new application. The easiest way to do this is to click **New Application** in the **Quick Panel** window. In the next dialog, click **Dev/Eval Kit**, and select **CY8CKIT-062-BLE** for the **CY8CKIT-062-BLE PSoC 6 BLE Pioneer Kit**. Click **Next**.

In the next dialog, **Starter Application**, select **EmptyPSoC6App**, and change the **Name**. CE213903 uses "PSoC6DfuApp0Uart", "PSoC6DfuApp0I2c", and "PSoC6DfuApp0Spi"; choose a name that indicates that this is the application that does the DFU. You can also indicate the host communication channel. Click **Next**. In the Summary dialog, confirm your selections and click **Finish**.

2. Configure App0 as a DFU.

To add DFU capability to an application, you must:

- Add a host communication channel to your design.
- Add other peripherals as needed
- Add DFU code

- **Add a communication channel.** Open *design. modus* in project <name>_config. Select a channel type, and configure it to work with the CY8CKIT-062-BLE kit:
  - For UART: Add a UART to SCB 5, with the following changes from the default: Oversample = 12, Clock = any convenient clock, RX = P5[0], TX = P5[1]. You can optionally assign names to the UART and the UART pins.
  - For I²C: Add an I2C to SCB 3, with the following changes from the default: Data Rate (kbps) = 400, Clock = any convenient clock, SCL = P6[0], SDA = P6[1]. You can optionally assign names to the I2C and the I2C pins.
  - For SPI: Add a SPI to SCB 6, with the following changes from the default: Clock = any convenient clock, SCLK = P12[2], MOSI = P12[0], MISO = P12[1], SS1 = P12[4]. You can optionally assign names to the SPI and the SPI pins.
- **Add other peripherals as needed.** For compatibility with CE213903, configure pins for kit LED and button support:
  - Configure pin P0[3] as follows: name = KIT_RGB_R, Drive Mode = Strong Drive Input Buffer off. Confirm that the Initial Drive State = High (1).
  - Configure pin P0[4] as follows: name = KIT_BTN1, Drive Mode = Resistive Pull-Up Input buffer on. Confirm that the Initial Drive State = High (1).
- **Add DFU code.** Right-click project <name>_mainapp and select **ModusToolbox Middleware Selector**. Select Middleware **DFU SDK Core**, and **DFU SDK transport UART** or one of the other transport options.

  Do the same for <name>_mainapp_cm0p and select only **DFU SDK core**. Required source *.c*, *.h*, and linker script (e.g., *.ld*) files are automatically added to both projects.

  For I²C and SPI channels, edit the file *dfu_user.c*, in the <name>_mainapp project *Source* folder, as follows:

  - Change `#include "transport_uart.h"` to:
    `#include "transport_i2c.h"` or
    `#include "transport_spi.h"`.
  - Change five instances of "`UART_Uart`" to "`I2C_I2c`" or "`SPI_Spi`".

  Integrate DFU code into the *main.c* files in <name>_mainapp and <name>_mainapp_cm0p. The easiest way to do this is to copy and paste the code from the code example **CE213903**, and then modify the code for your application.

  Right-click project <name>_mainapp and select **Properties**. In C/C++ **Build** > **Settings** > **Tool Settings** > **GNU ARM Cross C Linker** > **General**, change the file in the **Script files** entry to *dfu_cm4.ld*, to link using the custom script in the project. Do the same for project <name>_mainapp_cm0p, for *dfu_cm0p.ld*.
- **Build the project.** The easiest way to do this is to select <name>_mainapp, then in the Quick Panel click **Build <name> Application**.

3. Add App1.

As noted previously, with ModusToolbox IDE you can add applications to the same workspace as the DFU application, or they can be in separate workspaces, folders, or both. These instructions show how to add App1 to the same workspace as the DFU App0.

Note: *You can create any number of installable applications that work with the same DFU system. Before getting started with developing applications, developing a plan for DFU and applications for your overall system development needs is recommended. See **Determine the Applications in Your System**.*

Create a new application in the same manner as described previously. Select the same kit as was done for App0. Change the name; CE213903 uses "PSoC6DfuApp1". Choose a name that indicates that this is the downloadable application.

4. **Configure App1 as an installable application.**

- **Add other peripherals as needed.** Edit *design. modus* for this application. For compatibility with CE213903, configure pins for kit LED and button support:

  Configure pin P0[3] as follows: name = KIT_RGB_R, Drive Mode = Strong Drive Input Buffer off. Confirm that the Initial Drive State = High (1).

  Configure pin P0[4] as follows: name = KIT_BTN1, Drive Mode = Resistive Pull-Up Input buffer on. Confirm that the Initial Drive State = High (1).

- **Add DFU code.** Right-click project <name>_mainapp and select **ModusToolbox Middleware Selector**. Select Middleware **DFU SDK Core**. Do the same for <name>_mainapp_cm0p. Required source *.c*, *.h*, and linker script (e.g., *.ld*) files are automatically added to both projects. Adding DFU SDK Core to the application is done only to enable transfer of control to App0.

  Edit the files *dfu_cm4.ld* in project <name>_mainapp and *dfu_cm0p.ld* in project <name>_mainapp_cm0p as follows. This changes the application # for App1, and properly locates flash and RAM memory for App1:

```
/*
* DFU SDK specific: aliases regions, so the rest of code does not use
* application specific memory region names
*/
REGION_ALIAS("flash_core0", flash_app1_core0);
REGION_ALIAS("flash",       flash_app1_core1);
REGION_ALIAS("ram",           ram_app1_core1);

/* DFU SDK specific: sets an app Id */
__cy_app_id = 1;
```

Note: *It is important to not change anything else in these. ld files. With the exception of the above edits, they must be the same as the App0. ld files.*

Integrate DFU code into the *main.c* files in <name>_mainapp and <name>_mainapp_cm0p. The easiest way to do this is to copy and paste the code from the code example CE213903, and then modify the code for your application.

Copy the file *dfu_postbuild.bash* from the code example into the project <name>_mainapp. See **Appendix E**.

Right-click project <name>_mainapp_cm0p and select **Properties**. In **C/C++ Build** > **Settings** > **Tool Settings** > **GNU ARM Cross C Linker** > **General**, change the file in the **Script files** entry to *dfu_cm0p.ld*, to link using the custom script in the project.

Do the same for project <name>_mainapp, for *dfu_cm4. ld.* Then, in the same window select **Build Steps**, and replace the **Post-build steps Command** with the following:

```
${cy_sdk_install_dir}/tools/modus-shell-1.0/bin/bash --norc --noprofile
${workspace_loc:/PSoC6DfuApp1_mainapp}/dfu_postbuild.bash
${cy_sdk_install_dir}/tools/cymcuelftool-1.0/bin/cymcuelftool
${workspace_loc:/PSoC6DfuApp1_mainapp_cm0p}/${config_name:
PSoC6DfuApp1_mainapp_cm0p}/PSoC6DfuApp1_mainapp_cm0p.elf
```

```
${workspace_loc:/PSoC6DfuApp1_mainapp}/${config_name:PSoC6DfuApp1_mainapp}/
PSoC6DfuApp1_mainapp.elf ARM_CM4
```

to create the output. cyacd2 file, which is used to download the application for DFU. See **Appendix C**.

- **Build the project.** The easiest way to do this is to select <name>_mainapp, then in the Quick Panel click **Build <name> Application**.

5. Test the DFU and application.

Program App0 into a **CY8CKIT-062-BLE PSoC 6 BLE Pioneer Kit**. Then use the Device Firmware Update Host Tool (DHT) to download App1 to the kit; see tool usage instructions in **Appendix A**.

You can test the DFU and application using the instructions in the **CE213903** document, Operation section.

5.2.2 PSoC Creator Instructions

1. Create App0 and the workspace.

You can create a PSoC Creator project and its containing workspace at the same time. In PSoC Creator, select **File** > **New** > **Project...**. The Create Project window with the Select project type panel appears, as **Figure 15** shows.

- Click Target device.
- In the first drop-down menu, select PSoC 6.
- In the next drop-down menu, select CY8C6347BZI-BLD53. This is the PSoC 6 MCU device that is installed in the **CY8CKIT-062-BLE PSoC 6 BLE Pioneer Kit.**
- Click Next.

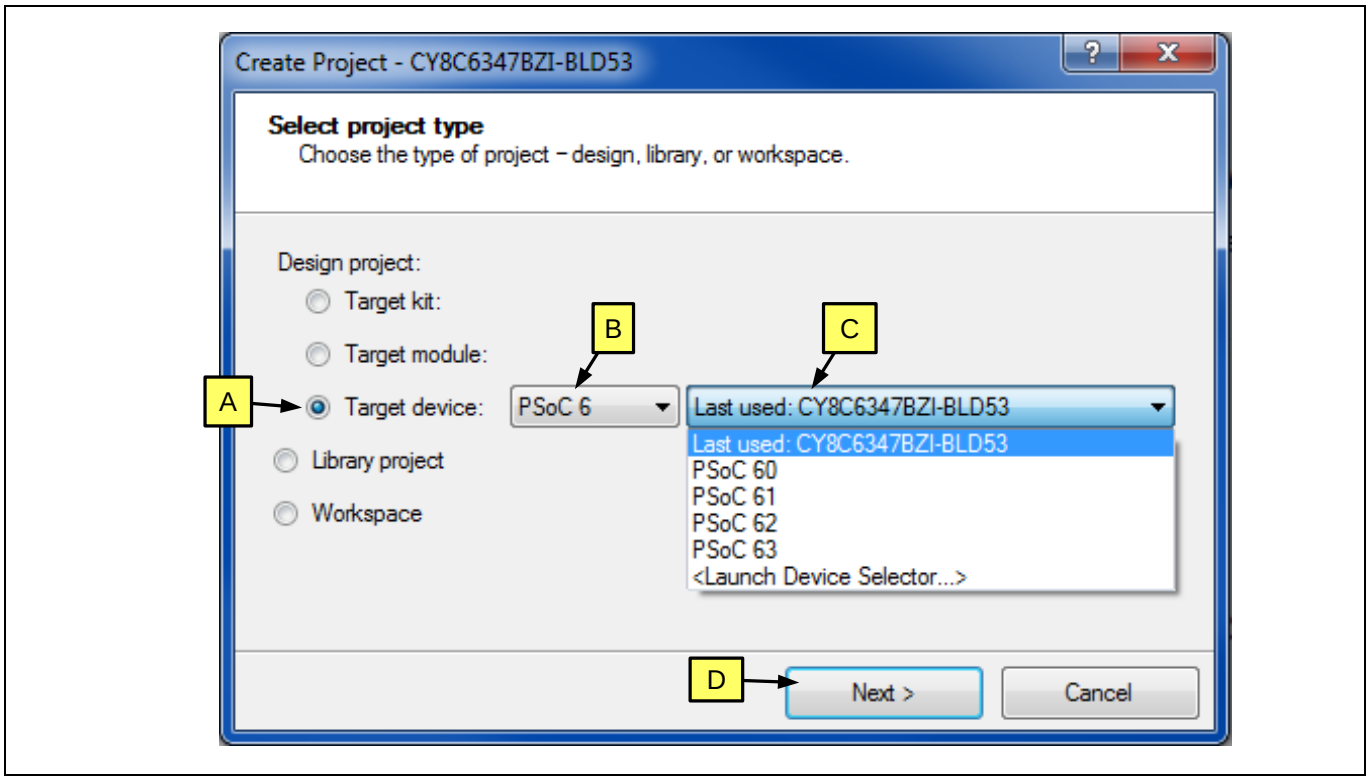

Figure 15 PSoC Creator Select the Target Device

Click **Next** on the next two panels, **select project template** and **Set target IDE(s)**. The Create Project panel appears, as **Figure 16** shows.

Enter the **Workspace name** and select its **Location**. Set **Project name** to “app0”. Click **Finish**.

A new folder is created in the indicated location; the folder name is the same as the workspace name. A folder *app0.cydsn* is created within the workspace folder; the. cydsn folder contains all project files.

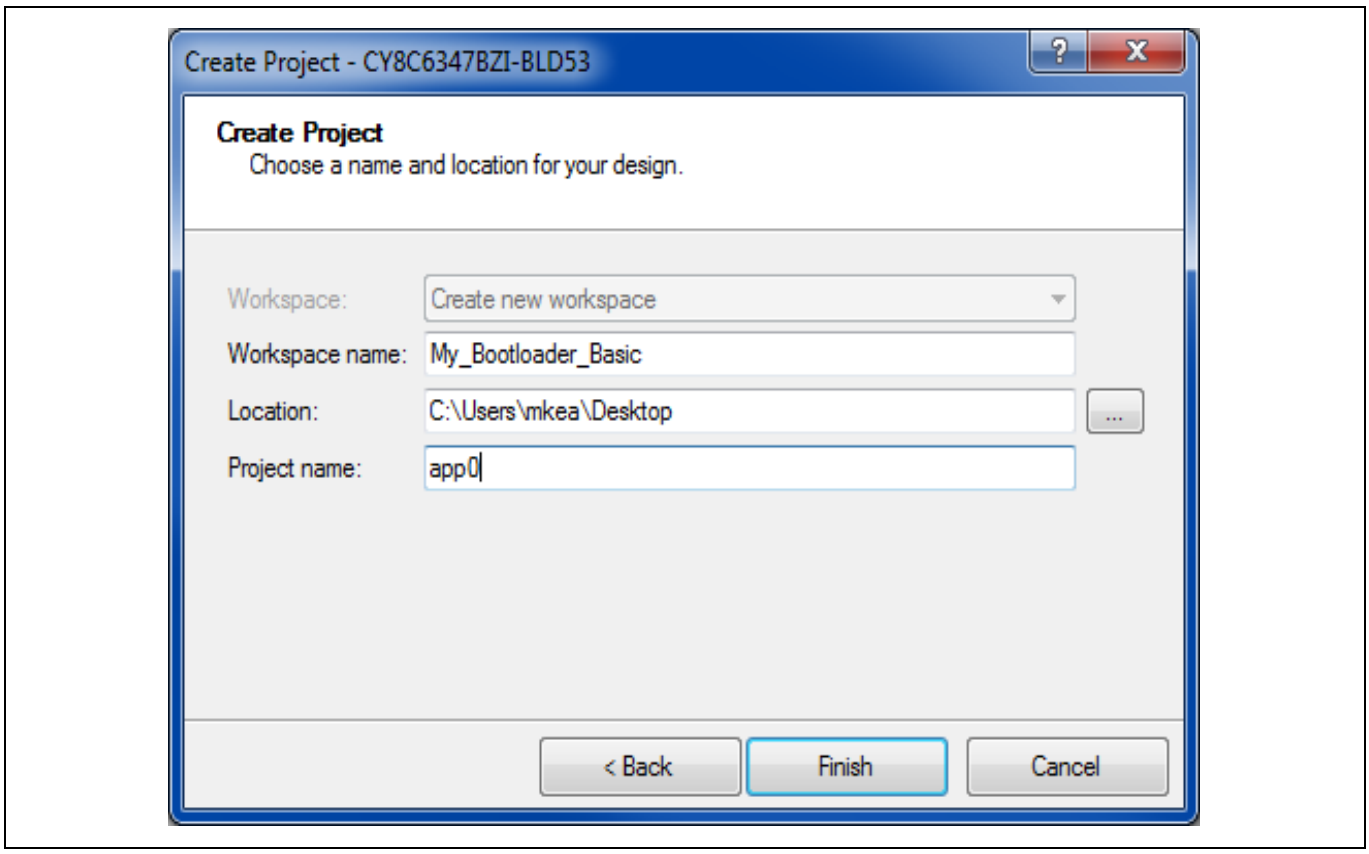

Figure 16 Create the PSoC Creator Project

The new workspace and project files are shown in the Workspace Explorer window; see **Figure 11**.

2. Configure App0 as a Bootloader.

To add bootloader capability to an application, you must:

- Add a communication Component to the project schematic. Use this Component to communicate with your bootloader host.
- Add other Components to the project schematic as needed for your application.
- Add Bootloader SDK and template files to the project.

- **Add a communication Component.** Open or double-click the project schematic (file *TopDesign.cysch* in the Workspace Explorer window).

  Open the Component Catalog, and navigate to your desired communication Component, for example UART, as **Figure 17** shows. Drag the Component onto the schematic.

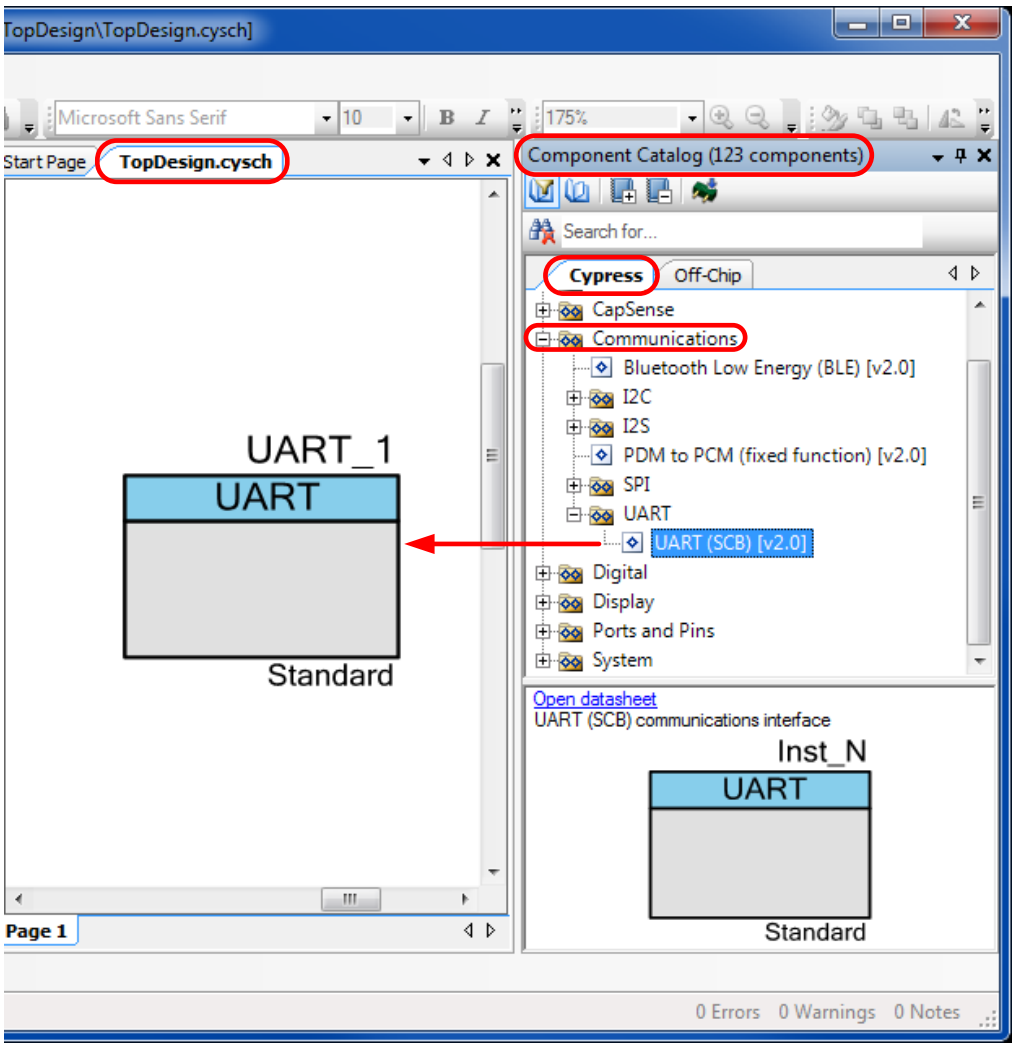

Figure 17 Add a PSoC Creator Communication Component to a Bootloader Project

Double-click the Component on the schematic to configure its parameters, such as baud rate, number of bits, etc., as **Figure 18** shows.

Figure 18 shows the changed parameter settings for this code example; note that **Name** is changed from UART_1 to UART, I2C_1 to I2C, or SPI_1 to SPI. This is recommended to work with the default Bootloader SDK files that are copied to your project later.

When done configuring the Component, click **OK**.

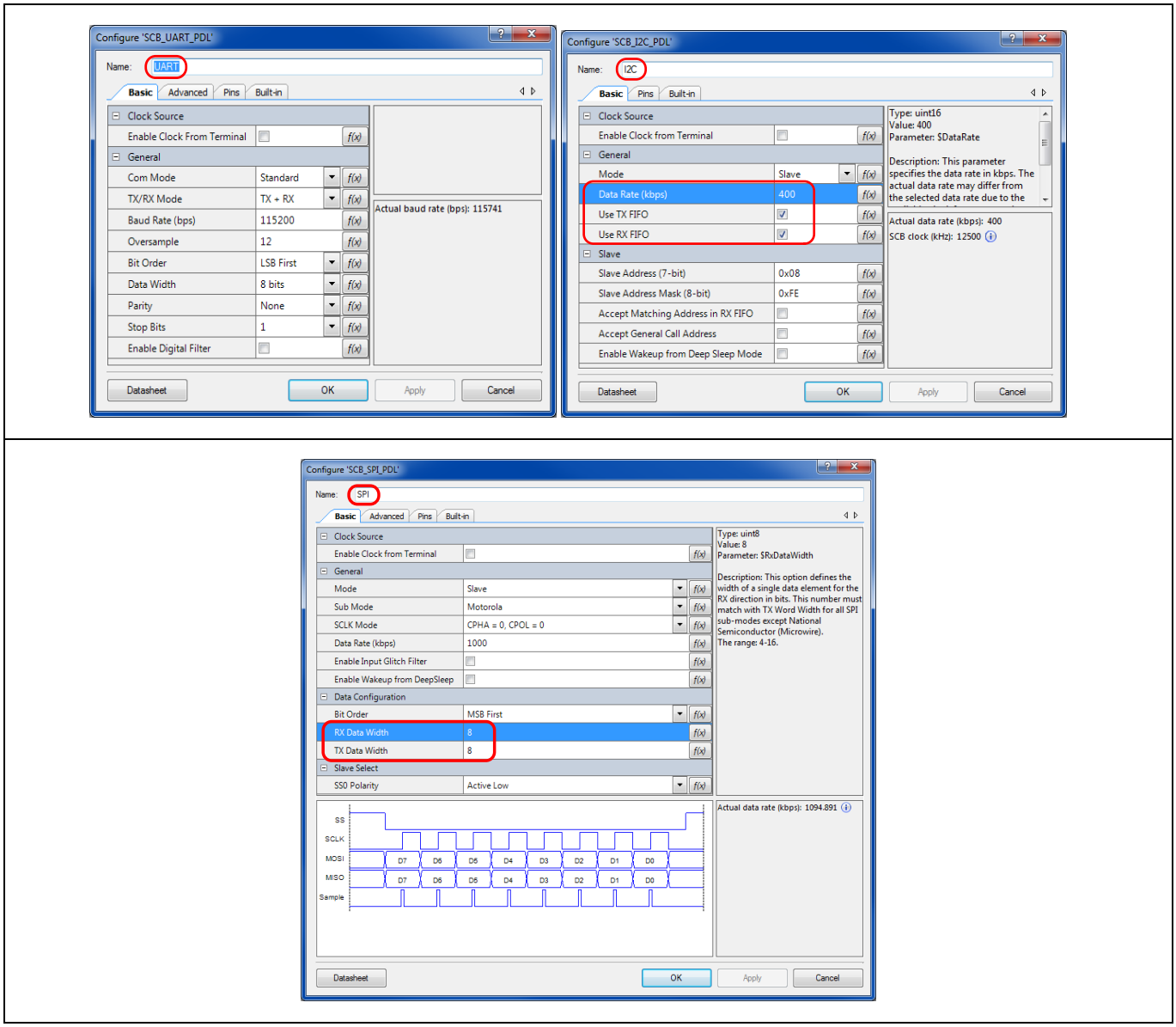

Figure 18 Configure the PSoC Creator Communication Component: UART, I2C, or SPI

- **Add other Components.** Add other PSoC Creator Components such as LED and button pins to the project schematic. The easiest way to do this is to copy and paste portions of the project schematic from the code example **CE213903**, and then modify the schematic as needed for your application.

  In the Design Wide Resources window, **Pins** tab, connect the UART, I2C, SPI, and other Component pins to the appropriate physical pins. Use **CE213903** and the guide for your kit for instruction.

- **Add Bootloader code to the project.** First, incorporate the Bootloader SDK into the project – see **Figure 10**.

  Then select **Build** > **Generate Application**. When done, the project should look like **Figure 11**, with the addition of some startup and other non-bootloader files.

For I²C and SPI bootloaders, edit the file *bootload_user.c*, in the project *Shared Files* folder, as follows:

- Change `#include "transport_uart.h"` to:
  `#include "transport_i2c.h"` or
  `#include "transport_spi.h"`.
- Change five instances of "`UART_Uart`" to "`I2C_I2c`" or "`SPI_Spi`".

Add bootloader code to the *main_cm0p.c* and *main_cm4.c* files. The easiest way to do this is to copy and paste the code from the code example **CE213903**, and then modify the code for your application.

- **Build the project.** Change the project build settings to use the template linker script files; see **Figure 12**. Then select **Build** > **Build app0**.

3. Add App1.

As noted previously, with PSoC Creator, you can add other projects – applications – to the same workspace as the bootloader application, or they can be in separate workspaces, folders, or both. These instructions show how to add App1 to the same workspace as the bootloader App0.

Note: *You can create any number of installable applications that work with the same bootloader. Before getting started with developing applications, developing a plan for bootloader and applications for your overall system development needs is recommended. See **Determine the Applications in Your System**.*

In the PSoC Creator Workspace Explorer window, right-click the **Workspace**, then select **Add** > **New Project...**, as **Figure 19** shows:

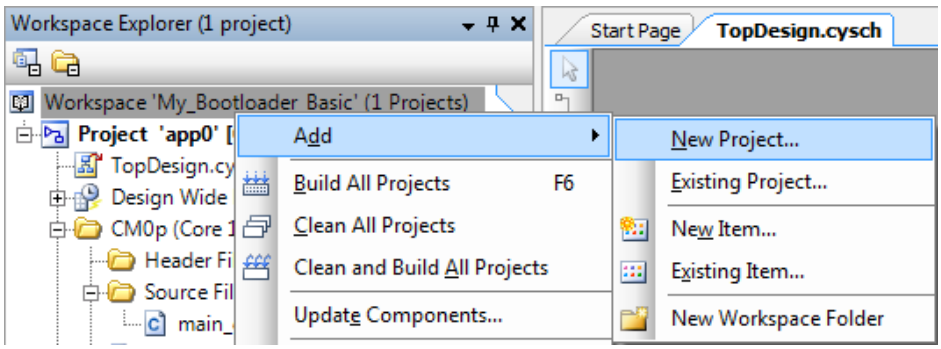

Figure 19 Add a New Project to a PSoC Creator Workspace

The Create Project window with the Select project type panel appears; see **Figure 15**. Make sure that the device selected is the same as for App0. Click **Next**.

Click **Next** on the next two panels: **Select project template** and **Set target IDE(s)**. The Create Project panel appears.

Set **Project name** to “app1”, as **Figure 20** shows. The default Location is the workspace folder. Click **Finish**.

A new *app1.cydsn* folder is created within the workspace folder; the *.cydsn* folder contains all project files.

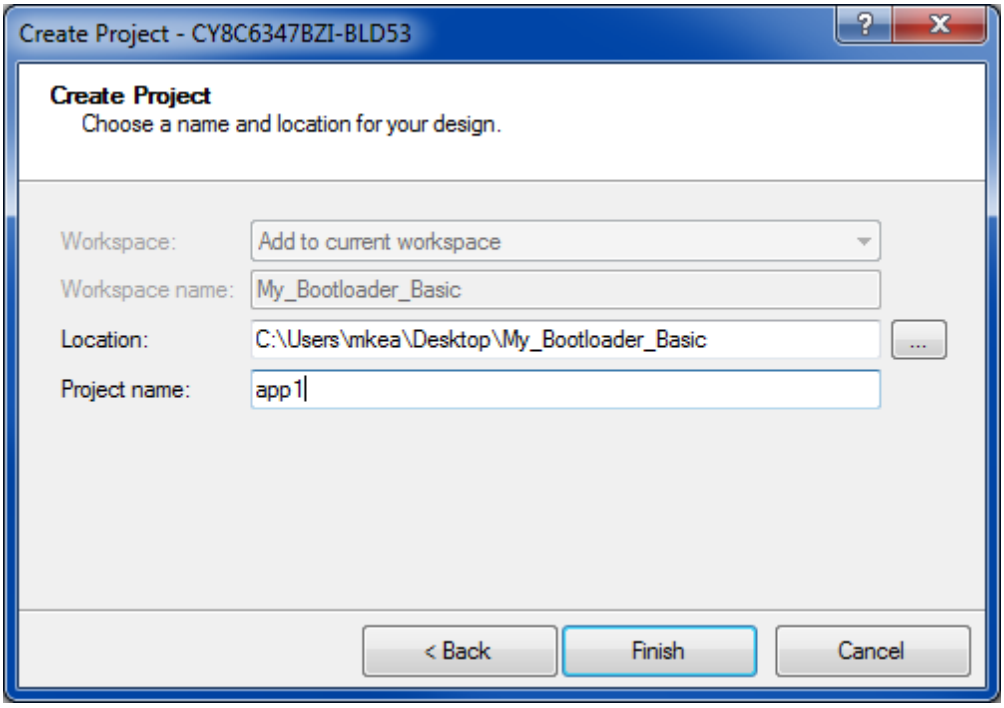

Figure 20 Create a Second PSoC Creator Project

4. Configure App1 as an installable application.

Note: App1 should be built with the same toolchain (GCC or MDK) as App0, or application transfer may fail.

- **Add other Components such as LED and button pins to the project schematic.** The easiest way to do this is to copy and paste portions of the project schematic from the code example **CE213903**, and then modify the schematic for your application.

  In the Design Wide Resources window, **Pins** tab, connect the Component pins to the appropriate physical pins. Use CE213903 and the guide for your kit for instruction.

- **Add Bootloader SDK and template files to the project.** Incorporate the Bootloader SDK into the project; see **Figure 10.** Select only the **Core** box, because App1 is a downloadable application and does not have bootloader capabilities.

  Then, select **Build** > **Generate Application**. When done, the project should look like **Figure 11**, with the addition of some startup and other non-bootloader files. Change the project build settings to use the template linker script files; see **Figure 8**.

  Add a post-build batch file to the *Shared Files* folder, as described in **Section 4.3 Step 8**. Then add a post-build command to the Cortex-M4 build – see **Figure 13**.

- **Edit files.** Edit the linker script files as described in **Section 4.3 Step 6**.

  Add bootloader code to the *main_cm0p.c* and *main_cm4.c* files. The easiest way to do this is to copy and paste the code from the code example **CE213903**, and then modify the code for your application.

Make sure that an array has been added to the *main_cm4.c* file for the application signature, as described in **Section 4.3.2 Step 8**.

- **Build the project.** After all build setting changes and file edits are complete, select **Build** > **Build app1**.

5. **Test the bootloader and applications**.

Program app0 into a **CY8CKIT-062-BLE PSoC 6 BLE Pioneer Kit**. Then use the Bootloader Host Program (BHP) to bootload app1 to the kit; see BHP usage instructions in **Appendix A.**

You can test the bootloader and applications using the instructions in the **CE213903** document, Operation section.

5.3 PSoC 6 MCU BLE Bootloaders

This section shows how to build three different BLE bootloaders in code examples:

- **CE216767**, PSoC 6 MCU with Bluetooth Low Energy (BLE) Connectivity Bootloader
- **CE220959**, PSoC 6 MCU BLE Bootloader with External Memory.
- **CE220960**, PSoC 6 MCU BLE Bootloader with Upgradeable Stack (see **this subsection**).

These code examples are similar to the **basic bootloaders code example**, with some additional features:

- CE216767 downloads a new application directly into flash, as **Figure 21** shows.
- CE220959 downloads a new application into temporary storage in an external NVM (kit IC U4, Cypress 512-Mbit serial NOR flash), and then copies it to its final destination in the PSoC 6 MCU device flash, as **Figure 22** shows.
- The bootloader (App0) and the application (App1) each have their own separate copy of the BLE stack code, as **Figure 21** and **Figure 22** show. Each copy occupies more than 128 KB of flash.

  For a bootloader with a single shared BLE stack, see **BLE Bootloader with Upgradeable Stack**.

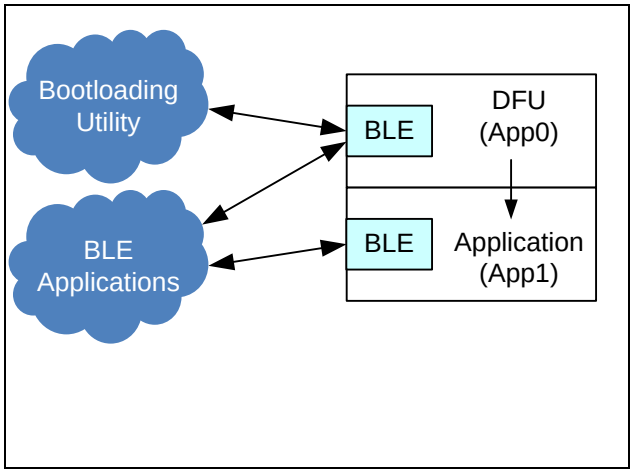

Figure 21 Data and Control Flow for CE216767

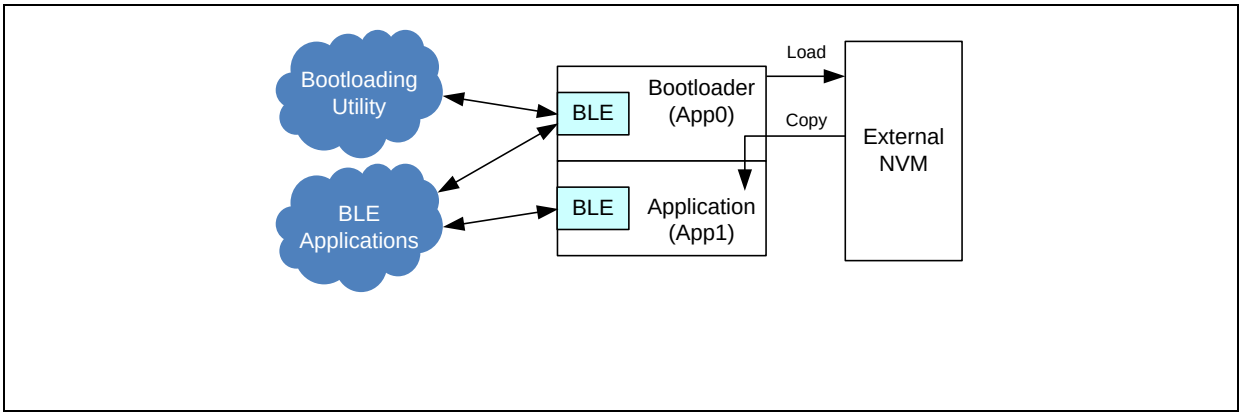

Figure 22 Data and Control Flow for CE220959

- The BLE stack code can be executed by either CPU or by both CPUs; this is a selectable option in the BLE Component configuration dialog. The code examples demonstrate this feature; in App0 the CM4 CPU executes the BLE stack, in App1 the CM0+ CPU executes the BLE stack.
- App0 and App1 enable different services in their BLE Component configurations:
  - App0: Bootloader and Immediate Alert Service (IAS). The IAS service is implemented when the application is not bootloading.
  - App1: Human Interface Device (HID) and Immediate Alert Service (IAS)

Both applications use the IAS to transfer control from one application to the other.

- The code examples make extensive use of **CY8CKIT-062-BLE PSoC 6 BLE Pioneer Kit** resources. All three RGB LEDs, the user button SW2, the BLE subsystem, and the USB-UART bridge are all used by both applications (App0 and App1).

To build the code examples, do the following. For more information on these steps, refer to **PSoC 6 MCU Basic Bootloaders** or **How to Use the SDK**.

1. Create App0 and the workspace, as **Figure 23** shows.

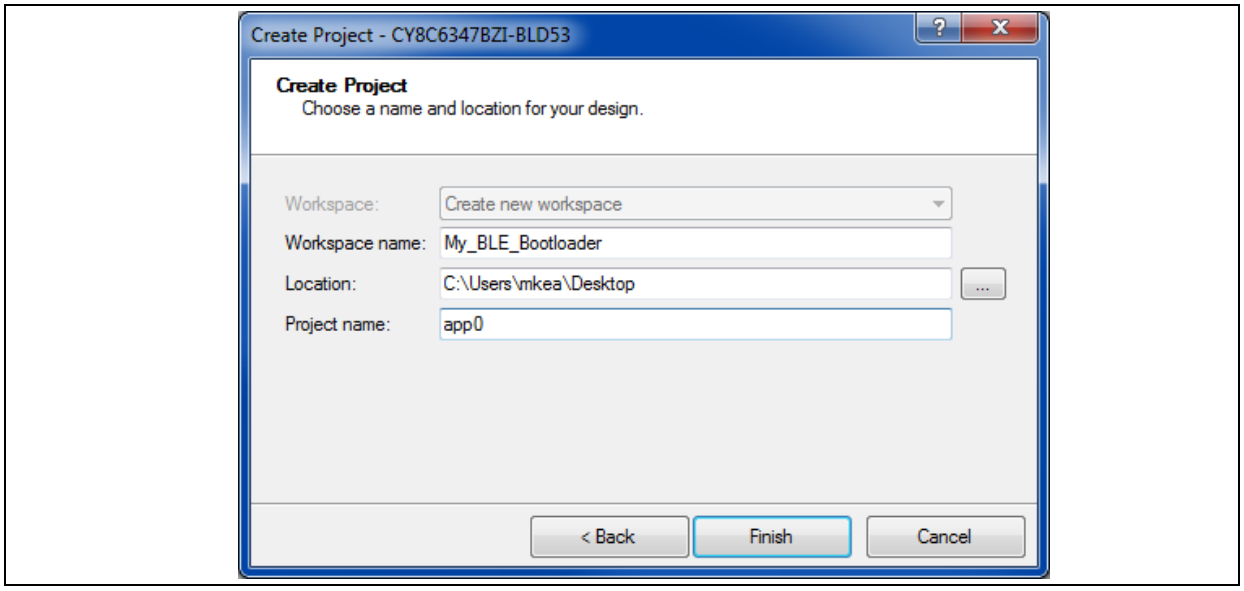

Figure 23 PSoC Creator Create BLE Bootloader App0

2. Add a BLE Component to the App0 Top Design schematic and configure the Component according to the BLE Component Configuration section in either code example. Specifically, make the following changes from the default configuration:

- (optional, but recommended to work with default Bootloader SDK files) Change the Component name to 'BLE'.
- **General** tab (see **Figure 24**): CPU core: **Single core (Complete Component on CM4)**

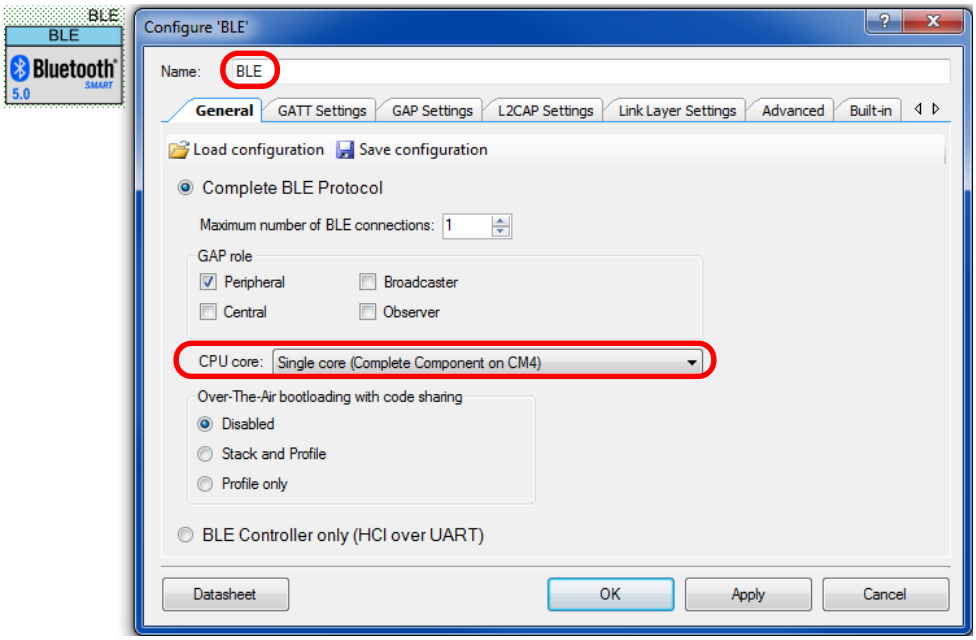

Figure 24 PSoC Creator BLE Component, General Tab Configuration

- GATT Settings tab (see **Figure 25**):
  - Generic Access, Peripheral Preferred Connection Parameters:
    1. Minimum Connection Interval: **0x000C**
    2. Maximum Connection Interval: **0x000C**
    3. Connection Supervision Timeout Multiplier: **0x00C8**

    These intervals are selected to minimize the bootloading time. Adjust as needed for your application.
  - Right-click the Server node in the GATT tree, and add the Bootloader service. Optionally, add the Immediate Alert service; the code examples use this service to transfer control between applications. There is no need to edit either service's characteristics.
  - Attribute MTU size (bytes): 512

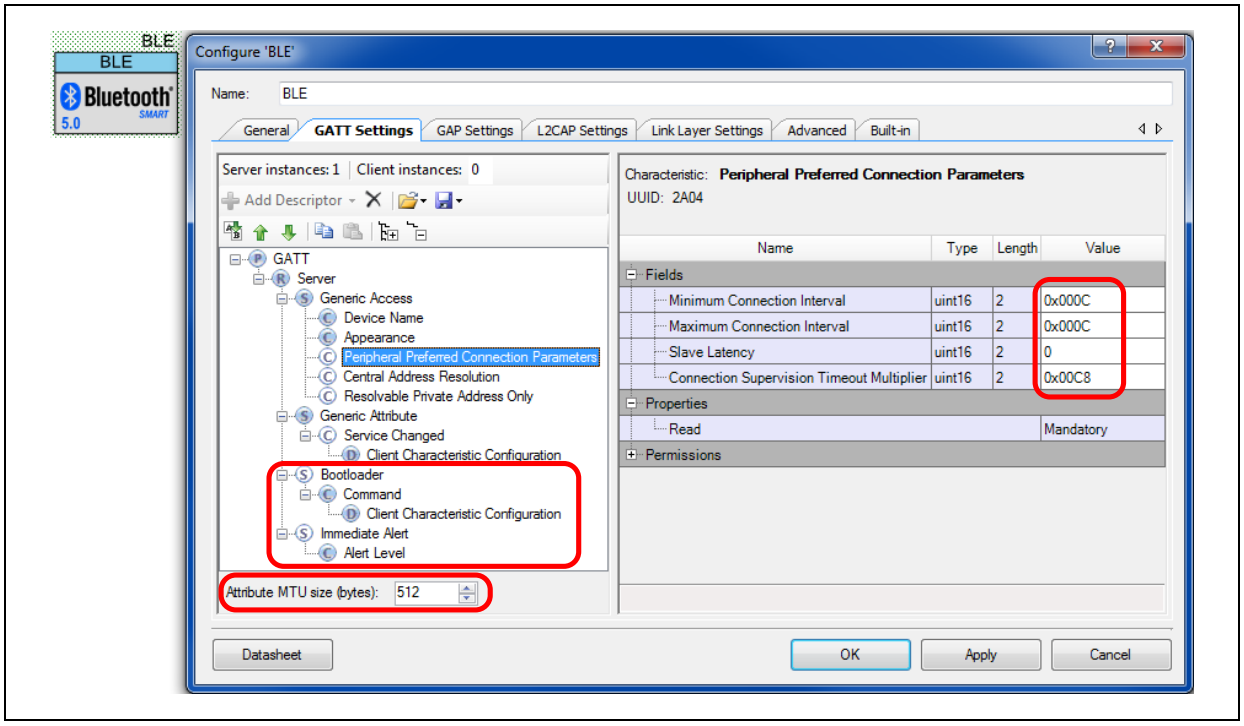

Figure 25 PSoC Creator BLE Component, GATT Settings Tab Configuration

- GAP Settings tab:
  - General, Device Name: The code examples use “BLE Bootloader”; use different text as needed for your application.
  - Peripheral Configuration 0, Advertisement packet: **Local Name** checked and set to **Complete**.
  - Security configuration 0 (see **Figure 26**), Security level: **Unauthenticated pairing with encryption**.
  - Security configuration 0, I/O capabilities: **No Input No Output**.
  - Security configuration 0, Bonding requirement: **No Bonding**.

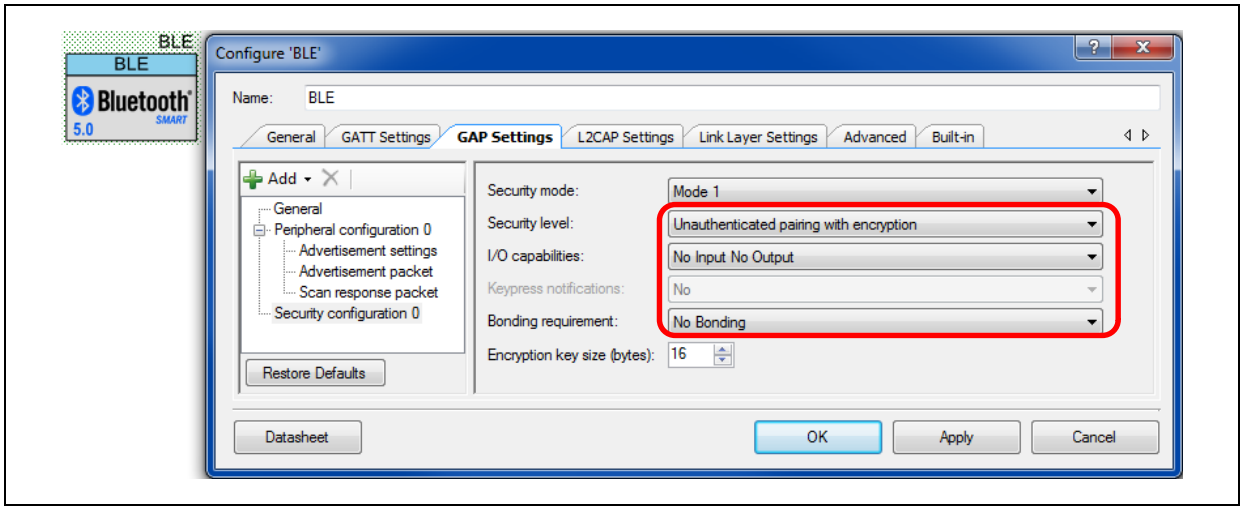

Figure 26 PSoC Creator BLE Component, GAP Settings Tab Configuration

- Link Layer Settings tab:
  - Link layer max TX and RX payload size (bytes): 251, for faster bootloading

3. For **CE220959**, add an SMIF Component to the App0 schematic, and configure the Component according to the SMIF Component Configuration section. Specifically, make the following changes from the default configuration:

- (optional, but recommended to work with CE220959 files) Name: SMIF
- Select **SMIF Datalines [2:3]**. The external memory IC has a quad-SPI (four data-line) interface with the PSoC 6 MCU device. The **SMIF Datalines [0:1]** box is selected as a default.
- Unselect **Generate code from cy_smif.cysmif file**. Files *cy_smif_memconfig.h/.c* and *smif_mem.h/.c* are already provided with the code example.

4. As needed, copy other Components – i.e., LEDs and button – from either the **CE216767** or **CE220959** top design schematic to your top design schematic.
5. In the Design Wide Resources window:

- **Pins** tab: Connect the Component pins to the appropriate physical pins. Note that the BLE Component does not use any pins.
- **Clocks** tab: Enable the WCO, and source Clk_LF and Clk_Bak from WCO. This required for operating the BLE Component in certain modes; see the BLE Component datasheet for details.
  - For **CE220959**, enable Clk_HF2 and set its frequency to 50 MHz. The easiest way to do this is to select the divide by 2 option in the Clk_HF2 section. The frequency limit is required for correct operation of the external memory interface.

6. Configure App0 as a bootloader in the project **Build Settings** – see **Figure 10**. Select **BLE** for the communication channel.
7. Select **Build** > **Generate Application**. This creates the bootloader linker files, e.g., *bootload_cm4.ld*.
8. Go back to the project **Build Settings** and set the **Custom Linker Script** for CM0+ and CM4 to the respective bootloader linker script files – see **Figure 12**.
9. Edit the flash and RAM memory region sizes in *bootload_common. ld*, as follows:

```
MEMORY
{
    flash_app0_core0   (rx)   : ORIGIN = 0x10000000, LENGTH = 0x10000
    flash_app0_core1   (rx)   : ORIGIN = 0x10010000, LENGTH = 0x30000
    flash_app1_core0   (rx)   : ORIGIN = 0x10040000, LENGTH = 0x32000
    flash_app1_core1   (rx)   : ORIGIN = 0x10072000, LENGTH = 0x02000
. . .
    ram_app0_core0     (rwx)  : ORIGIN = 0x08000100, LENGTH = 0x1F00
    ram_app0_core1     (rwx)  : ORIGIN = 0x08002000, LENGTH = 0x8000
. . .
    ram_app1_core0     (rwx)  : ORIGIN = 0x08000100, LENGTH = 0x1FF00
    ram_app1_core1     (rwx)  : ORIGIN = 0x08020000, LENGTH = 0x20000
. . .
```

The default *bootload_common. ld* allocates an equal amount of flash and RAM to each CPU in each application. The allocations must be adjusted for these code examples because:

- The BLE stack size is large
- App1 size should be minimized to reduce bootloading time

Note that the same changes are done for App1 in the code examples. Similar memory layouts exist for the MDK and IAR linker scripts.

10. Copy and paste files *debug.h/.c*, *ias.h/.c*, and *bootload_user.h/.c* from your **CE216767** or **CE220959** folder to your *app0.cydsn* folder. Overwrite the existing default *bootload_user.h/.c* files. In addition, for CE220959, copy and paste *cy_smif_memconfig.h/.c* and *smif_mem.h/.c* files.
    - The debug and ias (Immediate Alert Service) files are unique to the code examples and may be optional for your application.
    - The bootload_user files have already been edited to call the BLE transport functions instead of the default UART transport functions.
    - For CE220959, the SMIF files define the external memory configuration and provide functions to access external memory.
11. In the PSoC Creator Workspace Explorer window, include or move the files from the previous step to the project CM4 *Header Files* and *Source Files* folders, as **Figure 27** shows.
12. If these files are in the *Shared Files* folder, they may be included in the CM0+ build and compile errors may result.
13. Copy CM0+ and CM4 main code from **CE216767** or **CE220959** to your main files. Update the main and other source files as needed for your application.
14. Build app0, and program the kit with app0.
15. You can test your app0 by installing app1 from CE216767 or CE220959. Follow the instructions in the code example document, Operation section.
16. If you build your own app1, remember to edit the following files:
    - Three linker script files as described in **Step 9** and in **Section 4.3.2 Step 6**.
    - The batch file as described in **Section 4.3.2 Step 8**.

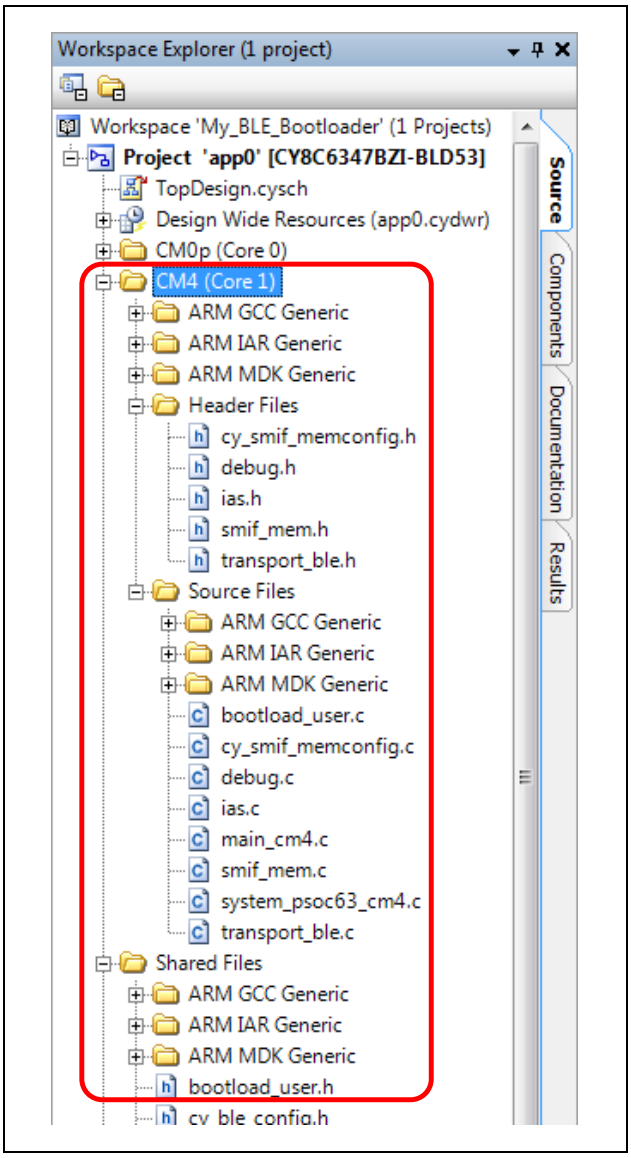

Figure 27 Source Files Included in PSoC Creator CM4 Folders

5.3.1 BLE Bootloader with Upgradeable Stack

This code example is different from **the other BLE bootloader code examples** in that it has three applications, as **Figure 28** shows.

In this example App0 does not do bootloading; it is solely a launcher application. It also copies a stack update from a temporary location to the proper location and starts either the stack or the user application. App0 cannot be updated by OTA bootloading.

App1 is the stack application. It contains the bootloader and the BLE stack. The bootloader can download an update to the user application or to the stack. Updating the stack requires placing the update in a temporary location and switching to the launcher for copying, because the stack cannot overwrite itself.

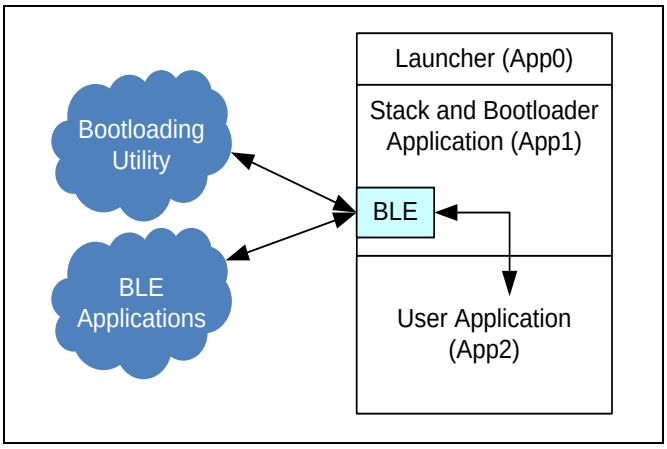

Figure 28 Data and Control Flow for CE220960

App2 is the user application. It has a PSoC Creator BLE Component, but it contains only profiles without the supporting stack code. Required stack code and variables are shared from the stack application, considerably reducing the size of the user application.

The steps to build this code example are similar to those for the other BLE code examples; the main differences are that you create three projects, and the BLE configurations are different:

- Create App0 and the workspace, as **Figure 23** shows.
- As needed, copy Components – i.e., LEDs and button – from the **CE220960** App0 top design schematic to your top design schematic.
- In the Design Wide Resources window, **Pins** tab, connect the Component pins to the appropriate physical pins. Use **CE220960** and the guide for your kit for instructions.
- Configure App0 as a bootloader in the project **Build Settings** – see **Figure 10**. You must do this to enable the copy application and transfer control functions – select only the **Core** and **App type** boxes. Do not select any of the communication channel boxes.
- Select Build > Generate Application. This creates a number of files, including the bootloader linker files, e.g., *bootload_cm4. ld.*
- Go back to the project **Build Settings** and set the **Custom Linker Script** for CM0+ and CM4 to the respective bootloader linker script files; see **Figure 12**. Also, add to the **CM4 User Commands** a call to the *post_build_core1.bat* file; see **Figure 13**.
- Update the files, and copy the new files, listed in **Table 5** from the corresponding files in **CE220960**. Add code as needed for your application.

Table 5 CE220960 PSoC Creator App0 Files Modified from the Bootloader SDK Default

File	Description of Modification from the Default
bootload_common.ld, bootload_cm0p.ld, bootload_cm4.ld *bootload_mdk_common.h, bootload_mdk_symbols.c* *bootload_cm0p.scat, bootload_cm4.scat*	Modified memory allocations for the three applications
bootload_user.h, bootload_user.c	Modified number of applications in metadata
main_cm0p.c, main_cm4.c	Addition of code to do the code example functions
New Files; copy from the code example, and add to the *Shared Files* folder	
post_build_core1.bat	Copies output to a build location for App1 and App2

- Add a new project App1 to the workspace. Add Components and files to the project, and configure it, in the same manner as App0 in the **other BLE code examples**; except in the BLE configuration dialog, select **Stack and Profile** for **Over-The-Air bootloading with code sharing**.
- Use CE220960 App1 for guidance. Note that in **CE220960** App1 has two post-build batch files, one for each core.
- Test using the instructions in CE220960.
- Add a new project App2 to the workspace. Add Components and files to App2, and configure it, in the same manner as App0 in the **other BLE code examples**, except in the BLE configuration dialog, select **Profile Only** for **Over-The-Air bootloading with code sharing**. Use CE220960 App2 for guidance.

5.4 PSoC 6 MCU Dual-Application Bootloader

This section shows how to build the BLE bootloader in the code example **CE221984**. This code example is like the **basic bootloaders code example**, except that there are two downloadable applications instead of one. There are three applications; App0 is the bootloader and App1 and App2 are the downloadable applications.

The bootloader transfers control to one of the applications in either a basic mode or a factory default ("golden image") mode. In factory default mode, the bootloader (App0) does not overwrite an installed and valid App1; an attempt to do so results in an error message. In basic mode, either application can be overwritten.

In this code example, the bootloader uses an I^2C communication channel. Changing to a different communication channel is simple; for more information, refer to one of the **previous** sections.

To build this code example, do the following steps. For more information on these steps, refer to **PSoC 6 MCU Basic Bootloaders** or **How to Use the SDK**.

1. Create App0 and the workspace, as **Figure 29** shows.

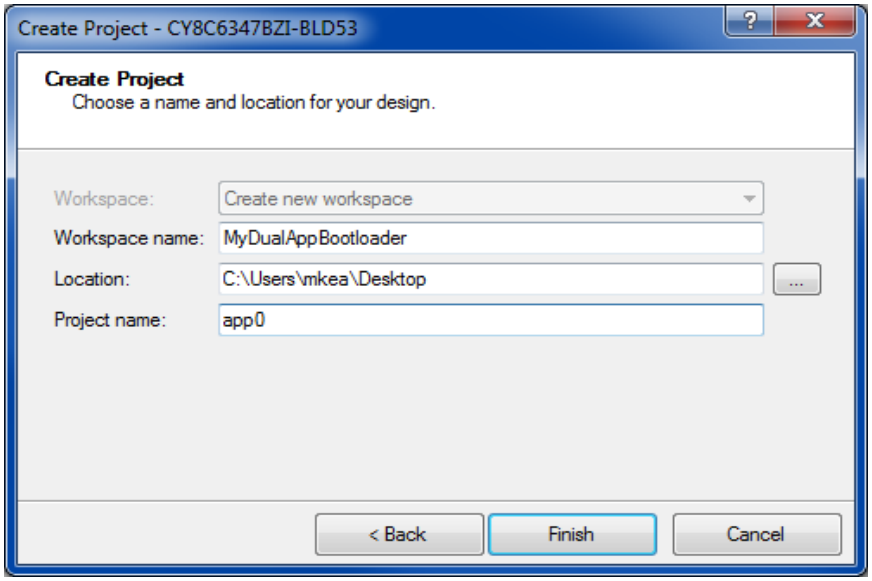

Figure 29 PSoC Creator Create Dual-Application Bootloader App0

2. Configure App0 as a bootloader in the project Build Settings; see **Figure 10**. Select I2C for the communication channel.
3. Select Build > Generate Application. This creates the bootloader linker files, e.g., *bootload_cm4.ld*.
4. Go back to the project Build Settings and set the Custom Linker Script for CM0+ and CM4 to the respective bootloader linker script files – see **Figure 12**.
5. Add an I2C Component to the App0 schematic and configure the Component according to **Figure 18**. At this time, you can also add the Pin Components used by **CE221984**; the easiest way to do this is to copy and paste the Components from the code example schematic.
6. In the Design-Wide Resources window, Pins tab, connect the Component pins to the appropriate physical pins.
7. Update the files listed in **Table 6** with the content of the corresponding files in **CE221984**. Add code as needed for your application.

Table 6 CE221984 App0 Files Modified from the Bootloader SDK Default

File	Description of Modification from the Default
bootload_common. ld *bootload_mdk_common.h*	Addition of memory regions for App2
bootload_user.h	Set the `CY_BOOTLOAD_GOLDEN_IMAGE_IDS` macro to identify App1 as the factory default image. To change the code example mode, change the `CY_BOOTLOAD_OPT_GOLDEN_IMAGE` macro to a nonzero value.
bootload_user.c	Changed transport functions to I²C from UART.

File	Description of Modification from the Default
	Added code to `Cy_Bootload_WriteData ()` to do the factory default ("golden image") check.
main_cm0p.c *main_cm4.c*	Addition of code to do the code example functions

8. Build the app0 project, and program it into the target kit. Test the application download (bootload) process. You can use the test steps listed in the Operation section of the **CE221984** document. Try changing the `CY_BOOTLOAD_OPT_GOLDEN_IMAGE` macro to a nonzero value and note the different behavior of the bootloader when you download App1.

   You can test the bootloader using the app1 and app2 projects already in the code example, or create your own applications, as the following step shows:

9. Add an app1 project and an app2 project. You can configure both projects as installable applications. The detailed steps required are presented in **Section 5.2.2 Step 3** and **Step 4**. Significant tasks are:

- Update the project build settings:
  - Peripheral Driver Library > Bootloader SDK. Select only the **Core** box. (**Figure 10**)
  - Linker custom scripts. **(Figure 12**)
  - Post-build batch file (**Figure 13**)
- Add the *post_build_core1.bat* file to the project, in the *Shared Files* folder. See **Appendix E** for typical batch file content.
- Update the project schematic and design-wide resources files with the content of the corresponding files in **CE221984**. Modify the design as needed for your application.
- Update the files listed in **Table 7** with the content of the corresponding files in CE221984. Add code as needed for your application.

Table 7 CE221984 PSoC Creator App1 and App2 Files Modified from the Bootloader SDK Default

File	Description of Modification from the Default
bootload_common. ld *bootload_mdk_common.h*	Addition of memory regions for App2
bootload_cmp0.ld *bootload_cm4.ld* *bootload_cmp0.scat* *bootload_cm4.scat*	Specify the memory regions and App ID as being for app1 or app2
bootload_user.h	Set the `CY_BOOTLOAD_GOLDEN_IMAGE_IDS` macro to identify App1 as the factory default image; to change the code example mode, change the `CY_BOOTLOAD_OPT_GOLDEN_IMAGE` macro to a nonzero value
main_cm0p.c *main_cm4.c*	Addition of code to do the code example functions

5.5 PSoC 6 MCU Encrypted Bootloader

This section shows how to build a bootloading system with security features, as demonstrated in the code example **CE222802**. This code example is similar to the **basic bootloaders code example**, with additional features – signing and encryption – needed for secure bootloading. App0 is the bootloader; the downloadable App1 can be signed, encrypted, or both.

In this code example, the bootloader uses a UART communication channel. Changing to a different communication channel is simple; for more information, refer to one of the **previous** sections.

To build this code example, do the following steps. For more information on these steps, refer to **PSoC 6 MCU Basic Bootloaders** or **How to Use the SDK**.

Note: Signing and encryption require keys. Files containing default keys are provided in the code example, but for your applications you will want to create new keys. The code example supports creating new keys, but to do so you must first install OpenSSL and Python in your computer. See the code example document for details.

1. Create App0 and the workspace, as **Figure 30** shows.

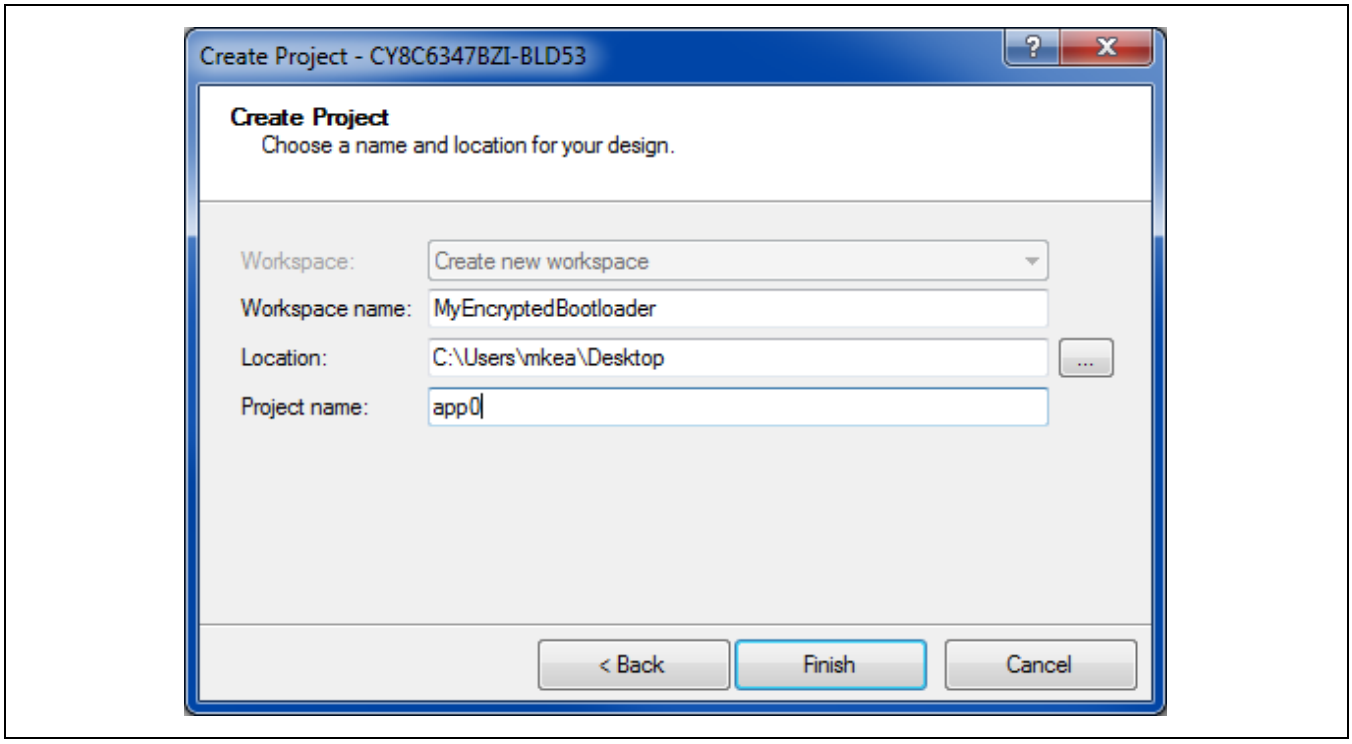

Figure 30 PSoC Creator Create Encrypted Bootloader App0

2. Configure App0 as a bootloader in the project Build Settings; see **Figure 10**. Select UART for the communication channel.
3. Also in the Build Settings, select Peripheral Driver Library > PDL > crypto, and under Variant select Extra library to enable support for AES decryption.

4. Select Build > Generate Application. This creates the bootloader linker files, e.g., *bootload_cm4.ld*.
5. Go back to the project Build Settings and set the Custom Linker Script for CM0+ and CM4 to the respective bootloader linker script files; see **Figure 12.** Also, add to the CM4 User Commands a call to the post_build_core1.bat file; see **Figure 13**.
6. Add a UART Component to the App0 schematic and configure the Component according to **Figure 18**. At this time, you can also add the Pin Components used by **CE222802**; the easiest way to do this is to copy and paste the Components from the code example schematic.
7. In the Design-Wide Resources window, Pins tab, connect the Component pins to the appropriate physical pins. Use **CE222802** and the guide for your kit for instruction.
8. Update the files, and copy the new files, listed in**Table 8** from the corresponding files in **CE222802**. Add code as needed for your application.

Table 8 CE222802 App0 Files Modified from the Bootloader SDK Default

File	Description of Modification from the Default
bootload_common.ld bootload_mdk_common.h	Different signature size
bootload_user.h	Set #defines to enable the crypto block and secure verification
bootload_user.c	Addition of key variables and decrypt function
main_cm0p.c main_cm4.c	Addition of code to do the code example functions
New Files; copy from the code example, and add to the Shared Files folder	
cy_si_config.h cy_si_keystorage.h, .c	Key storage
post_build_core1.bat	Signs and encrypts the output file

9. Copy from **CE222802** the project folder CE222802_Keys.cylib into your workspace folder. This folder contains key files that are used by post_build_core1.bat. You can replace these keys with your own keys; see the code example document for instructions.
10. The folder also contains a PSoC Creator library project file CE222802_Keys.cylib. You can optionally add this library project to your workspace.
11. Build the app0 project, and program it into the target kit. Test the application download (bootload) process. You can use the test steps listed in the Operation section of the code example document.

Note: *The bootloader project is built in a format such that it is validated by a Flash Boot module in PSoC 6 MCU supervisory flash (Sflash), as part of device initialization. If you reprogram the device to replace app0 with another project, validation will fail and the device will not boot. If you want to reprogram the device with other projects, you must first disable validation of app0 and then reprogram it into the device – see the instructions in the code example document.*

You can test the bootloader using the app1 project already in the code example, or create your own application, as the following step shows:

12. Add an app1 project. You can configure is as an installable application. The detailed steps required are presented **in Section 5.2.2 Step 3** and **Step 4**. Significant tasks are:

- Update the project build settings:
  - Peripheral Driver Library > Bootloader SDK. Select only the **Core** box. (**Figure 10**)
  - Linker custom scripts. (**Figure 12**)
  - Post-build batch file (**Figure 13**)
- Update the project schematic and design-wide resources files with the content of the corresponding files in **CE222802.** Modify the design as needed for your application.
- Update the files, and copy the new files, listed in **Table 9** from the corresponding files in CE222802. Add code as needed for your application.

Table 9 CE221984 App1 Files Modified from the Bootloader SDK Default

File	Description of Modification from the Default
bootload_common.ld *bootload_mdk_common.h*	Specify memory regions
bootload_cmp0.ld *bootload_cm4.ld* *bootload_cmp0.scat* *bootload_cm4.scat*	Specify memory regions and App ID
bootload_user.h	Set #defines to enable the crypto block and secure verification
main_cm0p.c *main_cm4.c*	Addition of code to do the code example functions
New Files; copy from the code example, and add to the *Shared Files* folder	
cy_si_config.h	Key storage
post_build_core1.bat	Signs and encrypts the output file

6 Related Documents

For a comprehensive list of PSoC 6 MCU resources, see **KBA223067** in the Cypress community.

Application Notes	
AN221774 – Getting Started with PSoC 6 MCU	Describes PSoC 6 MCU devices and how to build your first ModusToolbox or PSoC Creator project
AN210781 – Getting Started with PSoC 6 MCU with Bluetooth Low Energy (BLE) Connectivity	Describes PSoC 6 MCU with BLE Connectivity devices and how to build your first PSoC Creator project
AN215656 – PSoC 6 MCU: Dual-CPU System Design	Describes the dual-CPU architecture in PSoC 6 MCU, and shows how to build a simple dual-CPU design
AN219434 – Importing PSoC Creator Code into an IDE for a PSoC 6 MCU Project	Describes how to import the code generated by PSoC Creator into your preferred IDE
Code Examples	
CE213903 – PSoC 6 MCU Basic DFU	Describes UART, I^2C, and SPI DFU for PSoC 6 MCU
CE216767 – PSoC 6 MCU with Bluetooth Low Energy (BLE) Connectivity Bootloader	Describes a BLE bootloader for PSoC 6 MCU
CE220959 – PSoC 6 MCU BLE Bootloader with External Memory	Similar to the BLE bootloader; the downloaded application is temporarily saved in external memory and then copied to its final destination
CE220960 – PSoC 6 MCU BLE Bootloader with Upgradeable Stack	Similar to the BLE bootloader; the BLE stack can be updated in addition to the application
CE221984 – PSoC 6 MCU Dual-Application I2C Bootloader	Similar to the basic I^2C bootloader; manages two downloaded applications instead of one
CE222802 – PSoC 6 MCU Encrypted Bootloader	Similar to the basic UART bootloader; the application is digitally signed and encrypted
PSoC Creator Component Datasheets	
UART	Supports UART-based communications
I2C	Supports I^2C-based communications
SPI	Supports SPI-based communications
BLE	Supports BLE communications
Device Documentation	
PSoC 6 MCU: PSoC 63 with BLE Datasheet	**PSoC 6 MCU: PSoC 63 with BLE Architecture Technical Reference Manual**
Development Kit Documentation	
CY8CKIT-062-BLE	PSoC 6 BLE Pioneer Kit
CY8CKIT-062-WiFi-BT	PSoC 6 WiFi-BT Pioneer Kit

CY8CPROTO-063-BLE	PSoC 6 BLE Prototyping Kit
CY8CPROTO-062-4343W	PSoC 6 Wi-Fi Prototyping Kit
Tool Documentation	
ModusToolbox	ModusToolbox IDE simplifies development for IoT designers. It delivers easy-to-use tools and a familiar microcontroller (MCU) integrated development environment (IDE) for Windows, macOS, and Linux.
PSoC Creator	PSoC Creator enables concurrent hardware and firmware editing, compiling and debugging of PSoC devices. Applications are created using schematic capture and over 150 pre-verified, production-ready peripheral Components. Look in the downloads tab for Quick Start and User Guides.
Peripheral Driver Library (PDL)	Installed by PSoC Creator 4.2. Look in the <PDL install folder>/doc for the User Guide and the API Reference

Appendix A. DFU Host Tool

The DFU Host Tool (DHT) is a standalone graphical tool provided with ModusToolbox IDE. (There is also a Bootloader Host Program (BHP) provided with PSoC Creator. It operates much the same as DHT.) This tool is used to communicate with a PSoC device that has DFU installed. Using this tool, you can:

- Download and install an application to a device
- Verify an application that is already installed on a device
- Erase an application from a device

Note: *You cannot use DHT to install a DFU application into a device. Instead, you must program it through the device SWD/JTAG port, using other tools such as ModusToolbox IDE or Cypress Programmer. After a DFU application is installed, you can use DHT to install a downloadable application.*

DHT supports communicating with Cypress MCU devices via UART, I²C, SPI, or USB, as **Error! Reference s ource not found.**shows. You can see all devices available for connection. For UART or USB, communication can be done directly from your computer by connecting an appropriate cable. For I²C and SPI, a special communication port is needed, such as a KitProg module. The port configuration fields change depending on the selected port.

Note:

13.*At this time, only the UART, I²C, and SPI-based DFU are supported for PSoC 6 MCU.*

4. *DHT does not support Bluetooth Low-Energy (BLE). Use Cypress' **CySmart** product instead; see the CySmart User Guide section "Updating Peripheral Device Firmware".*

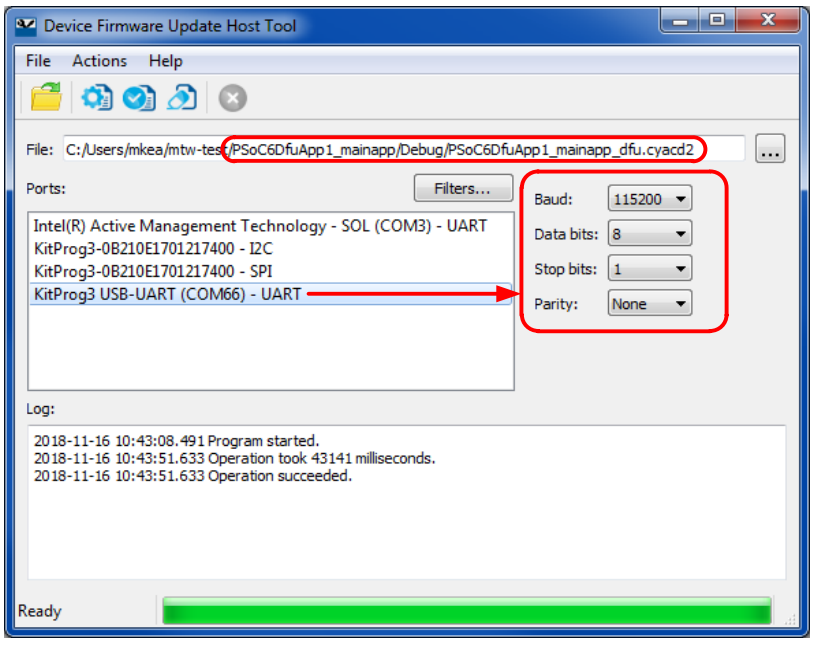

Figure 31 DFU Host Program Graphical User Interface (GUI)

Appendix B. Host Command/Response Protocol

The DFU module communicates with a host using a simple command-response protocol, regardless of the communication channel used. The DFU module receives commands from the communication channel and responds to each command by sending one or more bytes to the communication channel. See **Figure 2**.

The commands and responses are in the form of a byte stream, packetized in a manner that ensures the integrity of the data being transmitted. A packet validity check method is included and consists of a 2's complement 16-bit checksum.

B.1 Command/Response Packet Structure

Communication packets sent from the host to the DFU module have the structure shown in **Figure 32**:

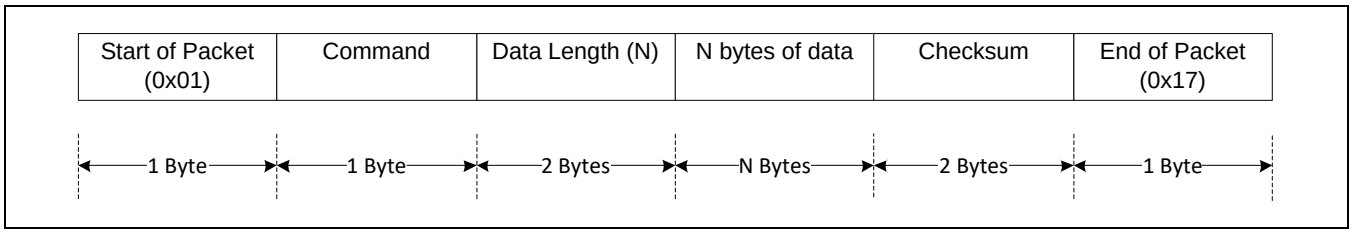

Figure 32 DFU Command Packet Structure

Response packets sent from the DFU module to the host have the structure shown in **Figure 33**:

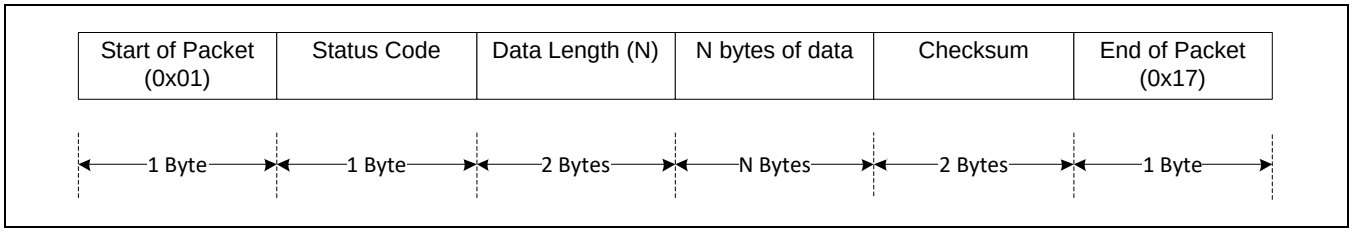

Figure 33 DFU Response Packet Structure

All multi-byte fields are LSB first.

B.2 Commands

Table 10 shows a list of all commands supported by the DFU SDK. All commands except Exit DFU are ignored until the Enter DFU command is received.

Table 10 DFU Commands List

DFU Commands		
Enter/Exit	**DFU Operation**	**Miscellaneous**
Enter DFU	**Send Data**	**Verify Application**

DFU Commands		
Enter/Exit	**DFU Operation**	**Miscellaneous**
Sync DFU	**Send Data Without Response**	**Set Application Metadata**
Exit DFU	**Program Data**	**Get Metadata**
	Verify Data	**Set EIVector**
	Erase Data	

There is no specific requirement for command execution time.

Table 11 shows a list of all status and error codes supported by the DFU SDK.

Table 11 DFU Status and Error Codes List

Status/Error Code	Value	Description
CY_DFU _SUCCESS	0x00	The command was successfully received and executed
CY_ DFU _ERROR_VERIFY	0x02	Verification of non-volatile memory (NVM) after writing failed
CY_ DFU _ ERROR_LENGTH	0x03	The amount of data sent is greater than expected
CY_ DFU _ ERROR_DATA	0x04	Packet data is not of the proper form
CY_ DFU _ ERROR_CMD	0x05	The command is not recognized
CY_ DFU _ ERROR_CHECKSUM	0x08	Packet checksum or CRC does not match the expected value
CY_ DFU _ ERROR_ROW	0x0A	The flash row number is not valid
CY_ DFU _ ERROR_ROW_ACCESS	0x0B	The flash row number cannot be accessed, for example due to MPU protection
CY_ DFU _ERROR_UNKNOWN	0x0F	An unknown error occurred

B.2.1 Enter DFU

Begins a DFU operation. All other commands except Exit DFU are ignored until this command is received. Responds with device information and DFU SDK version.

- Input
  - Command Byte: 0x38
  - Data Bytes:
    - 4 bytes (optional): product ID. If these bytes are included, and they are not 00 00 00 00, they are compared to device product ID data.
- Output
  - Status/Error Codes:
    - Success
    - Error Command

    - Error Data, used for product ID mismatch
    - Error Length
    - Error Checksum
  - Data Bytes:
    - 4 bytes: Device JTAG ID
    - 1 byte: Device revision
    - 3 bytes: DFU SDK version

B.2.2 Sync DFU

Resets the DFU to a known state, making it ready to accept a new command. Any data that was buffered is discarded. This command is needed only if the DFU module and the host get out of sync with each other.

- Input
  - Command Byte: 0x35
  - Data Bytes: N/A
- Output: N/A – this command is not acknowledged

B.2.3 Exit DFU

Exits from the DFU. Ends the DFU operation.

- Input
  - Command Byte: 0x3B
  - Data Bytes: N/A
- Output: N/A – This command is not acknowledged

B.2.4 Send Data

Transfers a block of data to the DFU module. This data is buffered in anticipation of a Program Data or Verify Data command. If a sequence of multiple send data commands are sent, the data is appended to the previous block. This command is used to break up large data transfers into smaller pieces, to prevent channel starvation in some communication protocols.

- Input
  - Command Byte: 0x37
  - Data Bytes:
    - n bytes: Data to write or verify
- Output
  - Status/Error Codes:
    - Success
    - Error Command
    - Error Data
    - Error Length

    - Error Checksum
  - Data Bytes: N/A

B.2.5 Send Data Without Response

Same as the Send Data command, except that no response is generated by the DFU module. This reduces DFU time for some applications.

- Input
  - Command Byte: 0x47
  - Data Bytes:
    - n bytes: Data to write or verify
- Output: N/A

B.2.6 Program Data

Writes data to one row of the device internal flash or page of external nonvolatile memory (NVM). May follow a series of **Send Data** or **Send Data Without Response** commands.

- Input
  - Command Byte: 0x49
  - Data Bytes:
    - 4 bytes: Address. Must be within the correct memory address space, and appropriately aligned. For internal flash, it must be aligned to a flash row boundary. For external memory, it must conform to external memory alignment requirements.
    - 4 bytes: CRC-32C of the entire data to be written. The data is verified both before and after programming.
    - n bytes: Data to write into the flash row or external NVM page.
- Output
  - Status/Error Codes:
    - Success
    - Error Command
    - Error Data
    - Error Length
    - Error Checksum
    - Error Flash Row
    - Error Flash Row Access
  - Data Bytes: N/A

B.2.7 Verify Data

Compares data to one row of the device internal flash or page of SMIF. May follow a series of **Send Data** or **Send Data Without Response** commands.

This command is optional; its presence depends on a user configuration macro in *dfu_user.h*.

- Input
  - Command Byte: 0x4A
  - Data Bytes:
    - 4 bytes: Address. Must be within the correct memory address space, and appropriately aligned. For internal flash, it must be aligned to a flash row boundary. For external memory, it must conform to external memory alignment requirements.
    - 4 bytes: CRC-32C of the entire data to be verified.
    - n bytes: Data to compare with the flash row or SMIF page.
- Output
  - Status/Error Codes:
    - Success
    - Error Verify
    - Error Command
    - Error Data
    - Error Length
    - Error Checksum
    - Error Flash Row
    - Error Flash Row Access
  - Data Bytes: N/A
- Implementation details
  - The command returns the “Success” status code if all data bytes match the bytes starting at the specified flash address, otherwise “Error Verify”.

B.2.8 Erase Data

Erases the contents of the specified internal flash row or SMIF page.

This command is optional; its presence depends on a user configuration macro in *dfu_user.h*.

- Input
  - Command Byte: 0x44
  - Data Bytes:
    - 4 bytes: Address. Must be within the correct memory address space, and appropriately aligned. For internal flash, it must be aligned to a flash row boundary. For external memory, it must conform to external memory alignment requirements.

- Output
  - Status/Error Codes:
    - Success
    - Error Command
    - Error Data
    - Error Length
    - Error Checksum
    - Error Flash Row
    - Error Flash Row Access
  - Data Bytes: N/A

B.2.9 Verify Application

Reports whether the checksum for the application in flash or external NVM is valid.

- Input
  - Command Byte: 0x31
  - Data Bytes:
    - 1 byte: Application number of the application to be verified. May range from 0 to the number of applications minus one.
- Output
  - Status/Error Codes:
    - Success
    - Error Command
    - Error Data
    - Error Length
    - Error Checksum
    - Error Flash Row Access
  - Data Bytes:
    - 1 byte: 1/0 for application is valid or not valid

B.2.10 Set Application Metadata

This command is used to set a given application's metadata. See **Appendix D, Application Metadata.**

Note: This command does not update the metadata if the user configures the DFU SDK to keep the metadata unchanged.

- Input
  - Command Byte: 0x4C
  - Data Bytes:
    - 1 byte: Application #

    - 8 bytes: metadata field format per Appendix D
- Output
  - Status/Error Codes:
    - Success
    - Error Command
    - Error Length
    - Error Data
    - Error Checksum
    - Error Flash Row Access
  - Data Bytes: N/A

B.2.11 Get Metadata

Reports selected metadata bytes.

This command is optional; its presence depends on a user configuration macro in *dfu_user.h*.

- Input
  - Command Byte: 0x3C
  - Data Bytes:
    - 2 bytes: from offset within row; 0 – 511
    - 2 bytes: to offset within row; 0 – 511 (inclusive)
- Output
  - Status/Error Codes:
    - Success
    - Error Command
    - Error Length
    - Error Data
    - Error Checksum
    - Error Flash Row Access
  - Data Bytes:
    - N bytes – per from and to offset bytes (inclusive)

B.2.12 Set EIVector

Sets an encryption initialization vector (EIV). This enables the DFU module to decrypt data before writing it to flash.

This command is optional; its presence depends on a user configuration macro in *dfu_user.h*.

- Input
  - Command Byte: 0x4D
  - Data Bytes:

    - n bytes: the vector; 0, 8, or 16 bytes, little-endian raw data
- Output
  - Status/Error Codes:
    - Success
    - Error Command
    - Error Length
    - Error Data
    - Error Checksum
  - Data Bytes: N/A

Appendix C. *.cyacd2* File Format

The *.cyacd2* file contains downloadable application data. It is created by CyMCUElfTool, and used by host programs such as Cypress' **DFU Host Program** and **CySmart** to send applications to the target DFU module, as **Figure 5** shows. The file data is in the form of ASCII hex numbers, similar to Intel hex format. Each byte of data is represented by two characters. For example, a byte 0x1E is represented by the characters 0x31 (ASCII '1') followed 0x45 (ASCII 'E').

All multi-byte fields are little-endian.

The file consists of a series of lines, or rows. Each row is terminated with ASCII CR, LF characters. A row is one of the following types:

- Encryption initial vector: An encryption initial vector row is of the format `@EIV:<bytes>`. The data in <bytes> is used by the host program in the **SetEIV command** to the DFU module.
- Application verification information: An application verification information row is of the format:
- `@APPINFO:[__cy_app_verify_start],[__cy_app_verify_length]`.
- The start and length data are used by the host program in the **Set Application Metadata command** to the DFU module.
- Header: A header row has the structure shown in **Table 12**.

Table 12 *cyacd2* Header Row Structure

File Version	Silicon ID	Silicon Revision	Checksum Type	App ID	Product ID
1 byte	4 bytes	1 byte	1 byte	1 byte	4 Bytes

  - File Version: Numbered starting at 1.
  - Silicon ID, Silicon Revision, Product ID: Used to prevent the application from being downloaded to the wrong device.
  - Checksum Type: The method used to verify a DFU packet (see **Appendix B, Command / Response Packet Structure**). 0 = checksum, 1 = CRC.
  - App ID: See **Figure 6**. This also controls which portion of the application metadata is updated for this application.
- Data: A data row has the structure shown in **Table 13**.

Table 13 *cyacd2* Data Row Structure

Header	Address	Data
1 character: ":"	4 bytes	N bytes

The value of N equals the total amount of data to be sent with a series of **Send Data** or **Send Data Without Response** commands followed by a **Program Data** or **Verify Data** command.

The value of N typically, but not necessarily, equals the length of an NVM row. For example, if downoading into RAM, then N may be an arbitrary value.

Appendix D. Application Metadata

The DFU SDK uses a designated region of NVM (or RAM in some cases) to store information about the applications – see **Figure 6**. Metadata information is generally used for the following purposes:

- Validate an application
- Transfer control from one application to another
- Copy an application image from a temporary location to its designated location

As noted in **Figure 6**, metadata typically occupies one flash row or NVM page. (In devices with small amounts of flash, multiple rows or pages may be used.) **Figure 6** also shows that metadata is located outside of any application.

Table 14 contains symbols that are used to define the location, size and usage of the DFU metadata. The symbols are defined in the SDK linker script files and C source files.

Note: *All examples shown are for the GCC compiler and linker. Similar statements exist in source and linker script files for the MDK and IAR compilers and linkers.*

Table 14 Metadata-Related Symbols

Symbol	Defined In	Purpose
`flash_boot_meta`	*dfu_cm0p.ld, dfu_cm4.ld*	Defines the physical memory region that contains the metadata
Example Usage: `flash_boot_meta (rw) : ORIGIN = 0x100FFA00, LENGTH = 0x400` Defines a region in the last 1 KB of the 1-MB PSoC 6 MCU user flash (which starts at 0x1000 0000).		
`__cy_boot_metadata_addr` `__cy_boot_metadata_length`	*dfu_cm0p.ld, dfu_cm4.ld*	These symbols define a compiler-independent memory address range that is used to store the metadata. These symbols are used in the DFU SDK code and may be used in user code.

Example Usage: /* Bootloader SDK metadata limits */

```
/* Note that __cy_memory_0_row_size equals the row length in
bytes of
   PSoC 6 user flash. */
__cy_boot_metadata_addr = ORIGIN (flash_boot_meta);
__cy_boot_metadata_length = __cy_memory_0_row_size;
```

Symbol	Defined In	Purpose
.cy_boot_metadata	*dfu_cm0p.ld, dfu_cm4.ld*	The DFU metadata is stored in this section. At build time, CyMCUElfTool calculates the checksum of this section and places it in the last four bytes of the section. When a checksum is not needed, rename the section to any other name.

Example Usage:
```
cy_boot_metadata :
{
    KEEP(*(.cy_boot_metadata))
} > flash_boot_meta
```

CY_DFU_MAX_APPS	*dfu_user.h*	Allows the user to control the maximum number of applications supported in the DFU metadata

Example Usage:
```
/* The smallest metadata size is CY_DFU_MAX_APPS * 8 bytes
per app +
   an optional 4 bytes for metadata checksum */
#define CY_DFU_MAX_APPS  (2u)
```

CY_DFU_METADATA_WRITABLE	*dfu_user.h*	The DFU does not necessarily write metadata – it can be done by the application or some other user code. Or metadata may be set using a compile-time constant within an application. An application can have metadata that is smaller than an NVM row. In all these cases, set this macro to 0 to prevent the DFU from writing metadata. However, note that the DFU requires that a metadata region exist and be properly initialized. This may be done by the DFU itself, or by an application.

Example Usage:
```
/* A non-zero value allows writing metadata with the
SetAppMetadata command. */
#define CY_DFU_METADATA_WRITABLE  (1)
```

Appendix E. Metadata Structure

Application metadata has eight bytes of data per application, followed by four bytes for checksum, as **Table 15** shows:

Table 15 Metadata Structure

Application 0	Application 1	...	Application N – 1	Checksum
8 bytes	8 bytes		8 bytes	4 bytes

Application Start Address	Application Length (bytes)
4 bytes	4 bytes

You can set the number of applications N in the *dfu_user.h* file:

```
#define CY_DFU_MAX_APPS (N)
```

The default value of N is 2.

Each application start address must be aligned to a flash row or NVM page boundary. The application length must be a multiple of the flash row or NVM page length.

The Checksum is calculated with the same algorithm that is used in the DFU commands **Program Data** and **Verify Data**. The default algorithm is CRC-32C.

Appendix F. Post-Build File Listings

The following are listings of the post-build bash and batch files for App1, from **CE213903**. The files are similar if not the same in the other code examples. The same bash/batch can be used with multiple downloadable applications, for example app2 in the **dual-application bootloader.**

F.1 ModusToolbox Post-Build Bash File

```
#!/bin/bash

################################################################################

# This script is designed to post process a PSoC 6 application. It performs
sign and merge.
#
# usage:
#   dfu_postbuild.bash <MCUELFTOOL_LOC> <CM0P_LOC> <CM4_LOC> <MCU_CORE>
#
################################################################################

MCUELFTOOL_LOC=$1
CM0P_LOC=$2
CM4_LOC=$3
MCU_CORE=$4

echo Script: cymcuelftool_postbuild
echo 1: MCUELFTOOL_LOC : $MCUELFTOOL_LOC
echo 2: CM0P_LOC       : $CM0P_LOC
echo 3: CM4_LOC        : $CM4_LOC
echo 4: MCU_CORE       : $MCU_CORE
echo

filenameNoExt_cm0p="${CM0P_LOC%.*}"
filenameNoExt_cm4="${CM4_LOC%.*}"

if [ "$MCU_CORE" == "ARM_CM4" ]; then
$MCUELFTOOL_LOC --sign $CM4_LOC --output $filenameNoExt_cm4"_signed.elf"
$MCUELFTOOL_LOC --merge $filenameNoExt_cm4"_signed.elf"
$filenameNoExt_cm0p"_signed.elf" --output $filenameNoExt_cm4"_final.elf"

#DFU
$MCUELFTOOL_LOC --sign $filenameNoExt_cm4"_final.elf" CRC --output
$filenameNoExt_cm4"_dfu.elf"
```

```
$MCUELFTOOL_LOC -P $filenameNoExt_cm4"_dfu.elf" --output
$filenameNoExt_cm4"_dfu.cyacd2"

else
$MCUELFTOOL_LOC --sign $CM0P_LOC --output $filenameNoExt_cm0p"_signed.elf"
  if [ -e $filenameNoExt_cm4"_signed.elf" ]; then
  $MCUELFTOOL_LOC --merge $filenameNoExt_cm4"_signed.elf"
$filenameNoExt_cm0p"_signed.elf" --output $filenameNoExt_cm4"_final.elf"
  fi
fi
```

F.2 PSoC Creator Post-Build Batch File

```
@rem Usage:
@rem Call post_build_core1.bat <tool> <output_dir> <project_short_name>
@rem E.g. in PSoC Creator 4.2:
@rem      post_build_core1.bat creator ${OutputDir} ${ProjectShortName}

@echo ----------------------------------------
@echo   Post-build commands for Cortex-M4 core
@echo ----------------------------------------

@rem Set proper path to your PDL 3.x and above installation
@set PDL_PATH="C:\Program Files (x86)\Cypress\PDL\3.0.1"

@set CY_MCU_ELF_TOOL=%PDL_PATH%"\tools\win\elf\cymcuelftool.exe"

@set IDE=%1

@if "%IDE%" == "creator" (
    @set OUTPUT_DIR=%2
    @set PRJ_NAME=%3
    @set ELF_EXT=.elf
)

@if "%IDE%" == "uvision" (
    @set OUTPUT_DIR=%2
    @set PRJ_NAME=%3
    @set ELF_EXT=.axf
)
```

```
@if "%IDE%" == "iar" (
    @set OUTPUT_DIR=%2
    @set PRJ_NAME=%3
    @set ELF_EXT=.out
)

@if "%IDE%" == "eclipse" (
    @set OUTPUT_DIR=%2
    @set PRJ_NAME=%3
    @set ELF_EXT=
)

%CY_MCU_ELF_TOOL% -S %OUTPUT_DIR%\%PRJ_NAME%%ELF_EXT% CRC
%CY_MCU_ELF_TOOL% -P %OUTPUT_DIR%\%PRJ_NAME%%ELF_EXT% --output
%OUTPUT_DIR%\%PRJ_NAME%.cyacd2
```

Revision history

Document version	Date of release	Description of changes
**	2017-03-08	New Application Note.
*A	2017-07-10	Updated for release of PSoC Creator 4.1 and PDL 3.0.0. Removed an error code from section B.2.12. Added Appendix E. Miscellaneous edits throughout.
*B	2017-08-25	Updated for release of PSoC Creator 4.2 and PDL 3.0.1. Added support for I^2C to basic bootloaders instructions. Other edits. Ported to new application note document template. Confidential tag removed.
*C	2018-01-04	Updated for release of PSoC Creator 4.2 ES100. Added support for SPI to basic bootloaders instructions. Added support for BLE bootloader and BLE bootloader with external memory. Added **Table 14** to Appendix D. Other edits. Ported to new application note document template.
*D	2018-03-27	Added support for code examples CE220959 and CE221984. Added figures 17 and 18. Miscellaneous minor updates and edits throughout the document. Ported to new application note document template.
*E	2018-02-25	Added support for CE220960 and CE222802. Added instructions for application signature data, in sections **4.3** and **0**.
*F	2018-12-08	Added support for ModusToolbox IDE and DFU SDK. Term “bootload” changed to “DFU”.
*G	2021-03-30	Migrated to Infineon template.

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

Edition 2021-03-31

Published by

Infineon Technologies AG

81726 Munich, Germany

Do you have a question about this document?

Go to www.cypress.com/support

Document reference

002-13924 Rev. *G

IMPORTANT NOTICE

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

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

For further information on the product, technology, delivery terms and conditions and prices please contact your nearest Infineon Technologies office (**www.infineon.com**).

WARNINGS

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

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