AN213924

PSoC 6 MCU Bootloader Software Development Kit (SDK) Guide

Author: Mark Ainsworth
Associated Part Family: All PSoC® 6 MCU Parts
Associated Code Example: CE213903
Related Application Notes: see Related Documents

More code examples? We heard you.

To access an ever-growing list of hundreds of PSoC code examples, please visit our code examples web page. You can also explore the PSoC video library here.

This guide provides comprehensive information on how to use the Bootloader Software Development Kit (SDK) to develop bootload systems for Cypress PSoC 6 MCU products. Detailed descriptions of the application programming interface (API) and code examples are included.

Contents

1 Introduction

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

A number of development tools are covered, including PSoC Creator™. PSoC Creator is a free Windows-based integrated design environment (IDE) that enables concurrent hardware and firmware design of systems based on Cypress PSoC MCUs. For more information on PSoC Creator, click here.

If you are new to bootloaders in general, basic concepts and design principles are explained in the next section, What is a Bootloader?

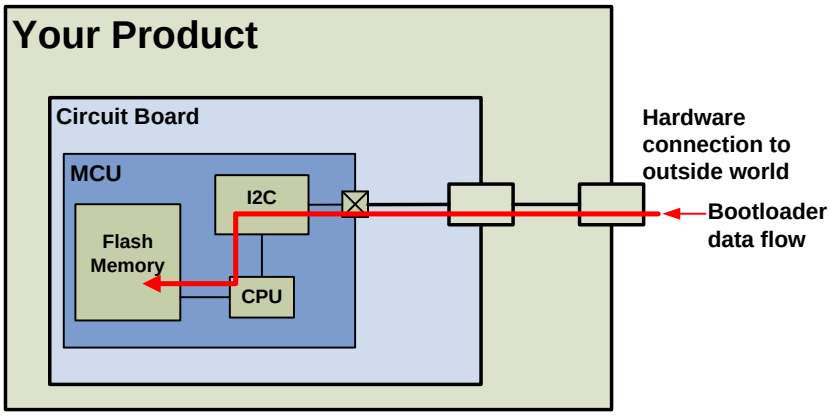
If you are familiar with bootloaders and want to see how they are implemented in PSoC 6 MCU, see the sections Bootloader SDK Description, How to Use the SDK, and Bootloader Code Examples. For a list of the bootloader code examples, see Related Documents. Click here for a complete list of PSoC 6 MCU code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

Note: At this time, only the UART and I^2C bootloaders are supported.

2 What Is a Bootloader?

Bootloaders are a common part of MCU system design. A bootloader makes it possible for a product's firmware to be updated in the field. 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. Bootloader Data Flow Block Diagram



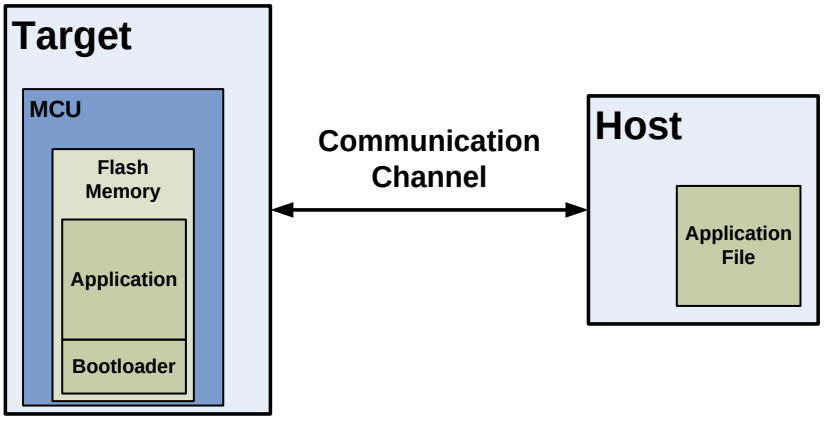
At the factory, initial programming of firmware into a product is typically done at PCB assembly time, using the MCU's Joint Test Action Group (JTAG) or Serial Wire Debugger (SWD) interface. However, these interfaces are not usually available in the field, and therefore are generally not used for firmware 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 Bluetooth low-energy (BLE), or it may be 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. That portion of the embedded firmware that knows how to update the flash is called a **bootloader**, as Figure 2 shows.

Figure 2. Bootloader 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 **bootloading**, or a **bootload** operation, or just **bootload** for short. The data that is placed in flash is called the **application** or **firmware image**.

2.2 Using a Bootloader

The bootloader and the application typically share a communication port. The first step in using a bootloader is to manipulate the product so that the bootloader, and not the application, is executing. This can be done in response to an event such as pressing a button on the product, or by sending a command to the product. The application detects such an event and responds by transferring control to the bootloader.

Once the bootloader is running, the host can send a “start bootload” command over the communication channel. If the bootloader sends an “OK” response, bootloading can begin.

2.3 Basic Bootloader Function Flow

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

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

- Check the application’s validity before transferring control to that application
- Manage the timing to start host communication
- Do the bootload / flash update operation
- Pass control to the application

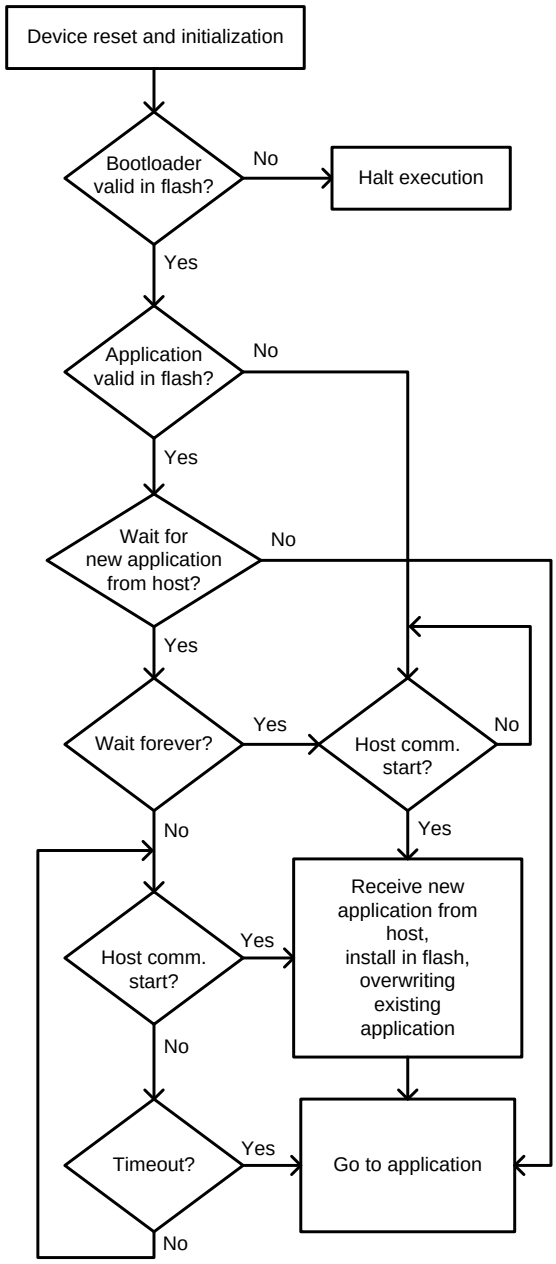
Figure 3 is a flow diagram that shows how this works.

2.4 Other Use Cases

Other more complex bootloading 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. Once the new application is verified, it is copied into the correct location in flash, and control is transferred to it.

Figure 3. Bootloader Function Flow



[1] In a typical MCU, a number of events may cause a device reset, for example 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 bootloader, because the bootloader executes first at device reset.

3 Bootloader SDK Description

The Cypress Bootloader SDK is an API consisting of a set of callable functions and other elements that enable rapid bootloader development. The SDK is shipped as a part of the Cypress Peripheral Driver Library (PDL) 3.0.1. The PDL is included as part of PSoC Creator 4.2; it is typically found in *C:\ Program Files (x86) \ Cypress \ PDL*. The SDK may be used by other IDEs as well as PSoC Creator.

The SDK consists of the following:

- Source code, typically *.h* and *.c* files that implement the SDK API
- Documentation, i.e., this Guide. See also the Bootloader SDK API Reference Manual in the PDL doc folder.
- Code examples

Note: You may modify the Bootloader 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 Bootloader SDK Files

The SDK contains the files listed in Table 1.

Note: PSoC 6 MCU has two CPU cores, ARM® Cortex®-M4 (CM4) and ARM Cortex-M0+ (CM0+). Linker script files are provided for each core, for a number of IDEs. For more information, see AN215656, PSoC 6 MCU Dual-Core CPU system Design.

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

Table 1. Bootloader SDK Files

File	Description
cy_bootload.h, .c	The bootloader software development kit (SDK) files.
bootload_user.h	Contains user-editable `#define` statements that control the operation and enabled features in the SDK.
bootload_user.c	Contains user functions required by the SDK: • Five functions that control communications with the bootloader host. These are also called transport functions. • Two functions – `ReadData()` and `WriteData()` – that control access to internal or external memory
transport_uart.h, .c *transport_... .h, .c*	Contains bootloader transport functions for the host communications Component being used. These functions are typically called by the transport functions in *bootload_user.c*.
bootload_common.ld	GCC linker script. It is user-editable—it controls the memory layout and the locations in memory for each application, and the code and data for each CPU core in each application. This file is included in the custom GCC linker scripts described next. This file is common to all applications.
bootload_cm4.ld, *bootload_cm0p.ld*	Custom GCC linker scripts. In each application, these files replace the auto-generated linker script files. These files locate the code and data sections for each of the CPU cores as well as the bootloader and other regions. These files include the memory layout described in *bootload_common.ld*.
bootload_mdk_common.h *bootload_mdk_symbols.c*	Similar in function to the GCC and IAR common linker scripts, for MDK. The MDK linker does not support includes in *.scat* files, so these files exist to create the necessary defines. These files are user-editable—they control the memory layout and the locations in memory for each application, and the code and data for each CPU core in each application. These files are common to all applications.
bootload_cm4.scat, *bootload_cm0p.scat*	Custom MDK linker scripts. In each application, these files replace the auto-generated linker script files. These files locate the code and data sections for each of the CPU cores as well as the bootloader and other regions.
bootload_common.icf	IAR linker script. It is user-editable—it controls the memory layout and the locations in memory for each application, and the code and data for each CPU core in each application. This file is included in the custom IAR linker scripts described below. This file is common to all applications.

File	Description
bootload_cm4.icf, bootload_cm0p.icf	Custom IAR linker scripts. In each application, these files replace the auto-generated linker script files. These files locate the code and data sections for each of the CPU cores as well as the bootloader and other regions. These files include the memory layout described in *bootload_common.icf*.
post_build_core1.bat	Batch file to create the downloadable application image for App1.

The files are organized in the PDL bootloader folder as Figure 4 shows:

Figure 4. Bootloader SDK File Organization

```
\---bootloader
    |   cy_bootload.h
    |   cy_bootload.c
    |   bootload_user.h
    |   bootload_user.c
    |   transport_uart.h
    |   transport_uart.c
    |   transport_... .h  template files for other
    |   transport_... .c  communication channels
    \---gcc_linker_scripts
    |       bootload_common.ld
    |       bootload_cm0p.ld
    |       bootload_cm4.ld
    \---mdk_linker_scripts
    |       bootload_mdk_common.h
    |       bootload_mdk_symbols.c
    |       bootload_cm0p.scat
    |       bootload_cm4.scat
    \---iar_linker_scripts
            bootload_common.icf
            bootload_cm0p.icf
            bootload_cm4.icf
```

3.1.1 User Callback Functions

The Bootloader SDK API functions call back to a number of user functions. This allows you to customize these important bootloading operations:

- Host communications, also called transport functions or communication interface functions.
- Reading and writing device internal flash as well as other non-volatile memory (NVM), for example external flash.

The required user callback functions are listed in Table 2. The bootloader code examples show typical code for these functions; see Bootloader Code Examples and Related Documents.

Table 2. User Callback Functions

Function	Description
For Host Communication. Examples are in *transport_xxx.h* and *.c* files	
Cy_Bootload_TransportStart()	Opens and initializes the communication channel
Cy_Bootload_TransportStop()	Closes the communication channel
Cy_Bootload_TransportReset()	Re-initializes the channel, typically to bring it back to a known state
Cy_Bootload_TransportRead()	Receives a packet from the host
Cy_Bootload_TransportWrite()	Sends a packet to the host
For Non-Volatile Memory (NVM) Access. Examples are in *bootload_user.c*.	
Cy_Bootload_ReadData()	Reads data from device flash or other NVM
Cy_Bootload_WriteData()	Writes data to device flash or other NVM

3.1.2 Linker Scripts

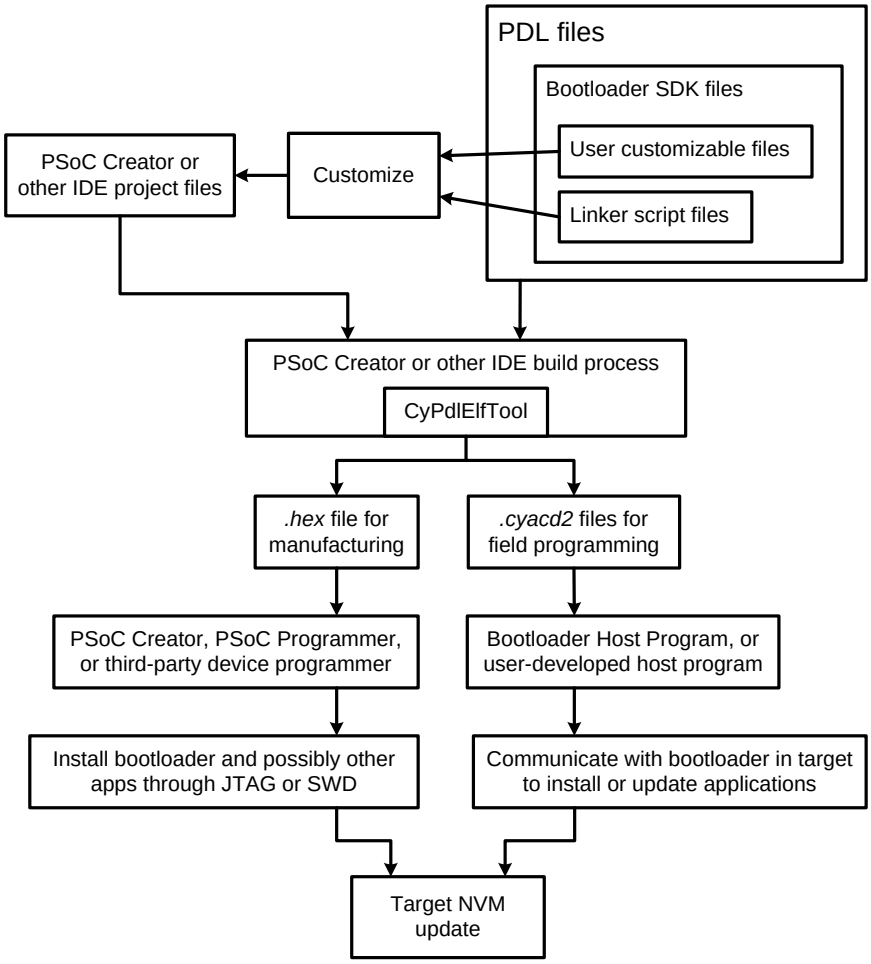
A system with a bootloader is by definition a system with multiple applications (at least two) in NVM. It is up to the user to decide where in NVM each application resides. The Bootloader SDK includes template linker script files (see Table 1 on page 4 and Figure 4 on page 5) that you can adapt to your application. The bootloader code examples include example linker script files.

3.2 Bootloading Ecosystem

Other elements of the bootloading system are not included in the SDK. They are available in PSoC Creator or the PDL, and can be used by other IDEs:

- Bootloader Host Program (BHP). This is an example of the "host" shown in Figure 2 on page 2. Find it in the PSoC Creator menu item **Tools** > **Bootloader Host...**. For more information, see Appendix A, Bootloader Host Program.
- CyPDLElfTool is a utility that is provided as part of the PDL. It is called as part of project build, for PSoC Creator or other IDEs, as Figure 5 shows. Its main purpose is to combine 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.
- Other drivers – for communication, flash, security, etc. – are in the PDL drivers folder. They are called as needed by the Bootloader SDK API functions.

Figure 5. Project Build and Bootload Process Diagram



4 How to Use the SDK

This section describes the general process for using the Bootloader SDK. At a high level, the steps are:

1. Determine the Applications in Your System
2. Locate Applications in Memory
3. Design the Application
4. Build and Program the Application

For detailed instructions on how to build the Bootloader SDK code examples, go to How to Build the Bootloader Code Examples.

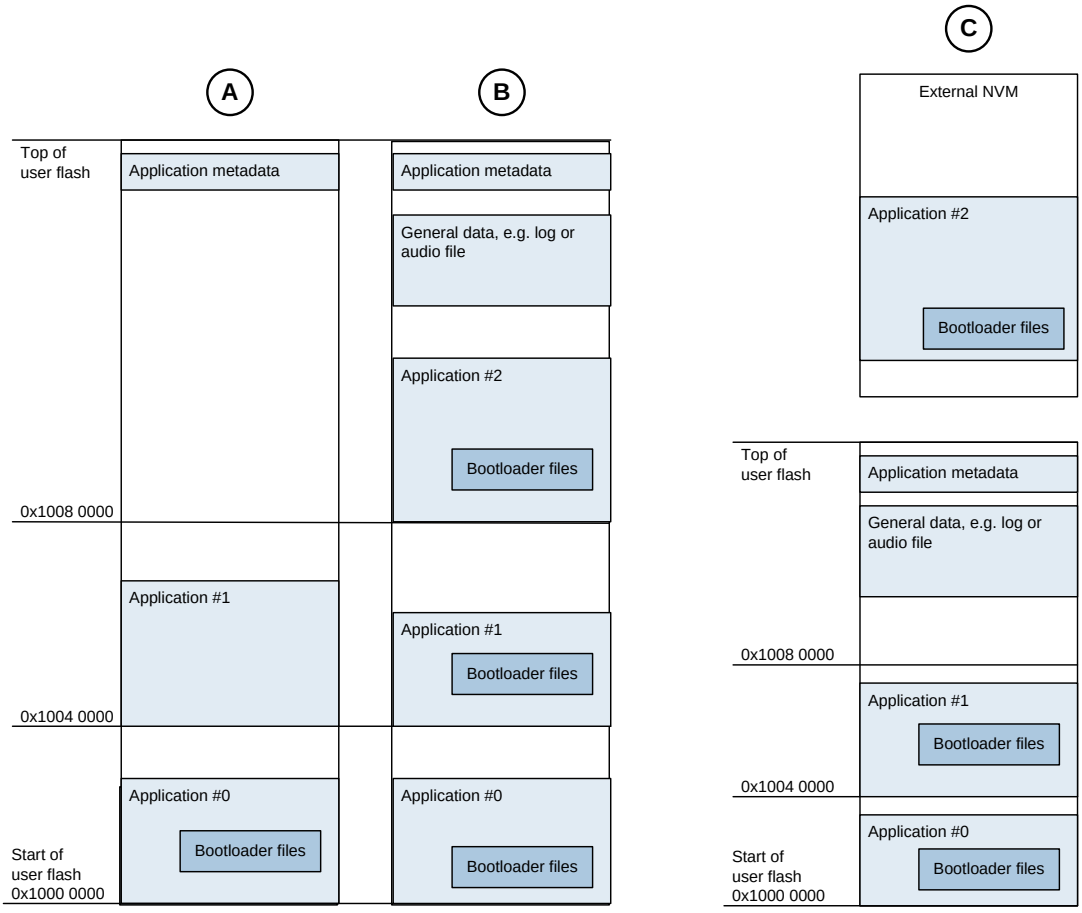
4.1 Determine the Applications in Your System

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

- Map **A** shows a basic case, where application #0 bootloads application #1.
- Map **B** shows a more complex case. There are three applications. All applications can bootload any other application. (In practice, application #0 is usually not updated.) A data file is located outside all of the applications.
- Map **C** shows the same case as map **B** except that one of the applications is located in external non-volatile memory (NVM).

In each of the maps, the bootloader system uses a small amount of flash (usually one row) to store application metadata. For information, see Appendix E, Application Metadata.

Figure 6. Example Memory Maps for Bootloading



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 flash row size.
- An application may have code for one or more CPU cores, for example the ARM Cortex-M4 and Cortex-M0+ in a PSoC 6 MCU device. For more information, see application note AN215656, PSoC 6 MCU Dual-Core CPU system Design.
- The first application, usually called “App0”, must be located at the beginning of user flash, which in PSoC 6 is at address 0x1000 0000, as Figure 6 on page 7 shows. This is the first application to execute after device reset.[1]
- Any application may incorporate the Bootloader SDK, and thus have bootloader capability; that is, the ability to download and install any other application. In most cases, App0 should have bootloader capability.
- PSoC 6 flash has a read-while-write (RWW) capability, across 256-KB regions. That is, bootloader code may execute in one flash region while updating flash in another region. As an example, Figure 6 shows applications in different 256-KB regions.
- Applications may reside in external memory as well as device internal flash, as map **C** in Figure 6 shows. External memory is located in a 128-MB sub-region from 0x1800 0000 to 0x1FFF FFFF.

  Note: Internal flash and external memory are collectively called non-volatile memory, or NVM.
- PSoC 6 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: NVM data that may be updated during normal workflow, for example an events log, or image or audio data, 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 a few rows of flash (only one is needed in most cases) for application metadata, as Figure 6 shows. Application metadata is managed by the Bootloader SDK, and is used by applications to transfer control to another application. Application metadata space should be outside any application space.

4.3 Design the Application

Note: This section contains only bootloader-specific 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 Bootloader Code Examples in this document.

Next, design each of the applications identified in the previous steps. Each application is a single PSoC Creator (or other IDE) project. With PSoC Creator, you can have all of the applications in one workspace (*.cywrk* file), or in separate workspaces as well as in separate locations on your computer. Before getting started with PSoC, it is a good idea to develop a workspaces / projects plan for your overall system development needs.

Note: All projects must have the same PSoC device.

Do the following for each of your PSoC Creator projects that incorporates the Bootloader SDK:

1. 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 most of the communication Components in the PSoC Creator Component Catalog; including UART, SPI, I^2C, BLE, and USB. Code examples are available for each communication Component; see Bootloader Code Examples and Related Documents.

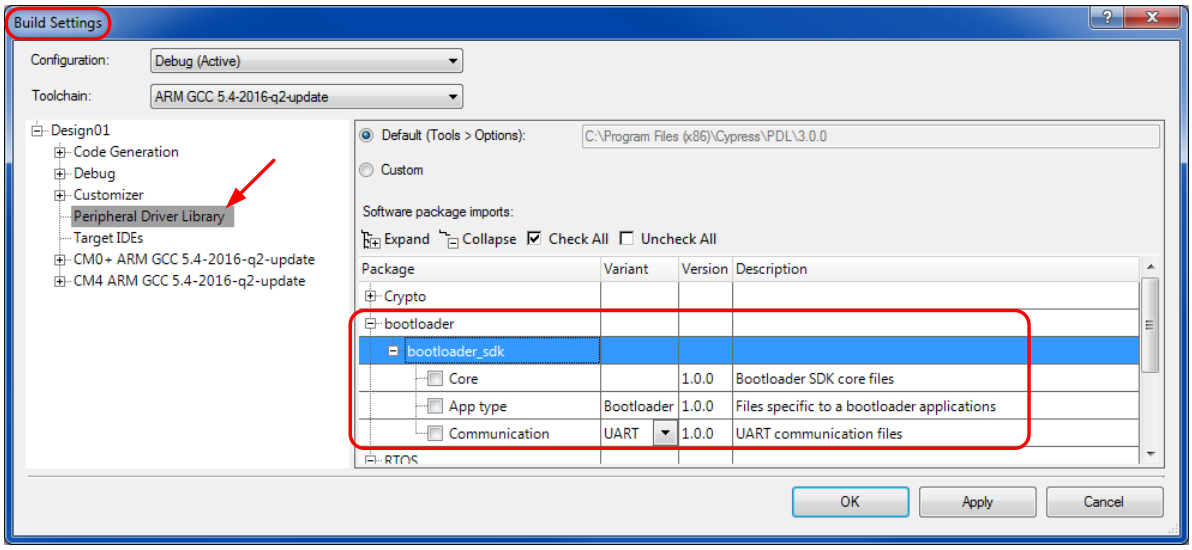
   Note: For compatibility with the auto-generated transport files, name the Component as **UART**.

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

You can also create a custom communication channel; it must implement the transport functions described in User Callback Functions.

2. Incorporate the Bootloader SDK into the project, as Figure 7 shows. Right-click the project in the Workspace Explorer window, and select **Build Settings...**. In the **Build Settings** dialog select **Peripheral Driver Library**.

Figure 7. Incorporate Bootloader SDK into a PSoC Creator Project



Check **Core** if the project is either a bootloader or a downloadable application. A downloadable application may need to transfer control to another application, and the code to do so is included in the Bootloader SDK core files.

If the project is a bootloader, also check **App type** and **Communication**, and select the host communication channel type.

Note: For I^2C bootloader, do not check the **Communication** box.

Click **OK** when done.

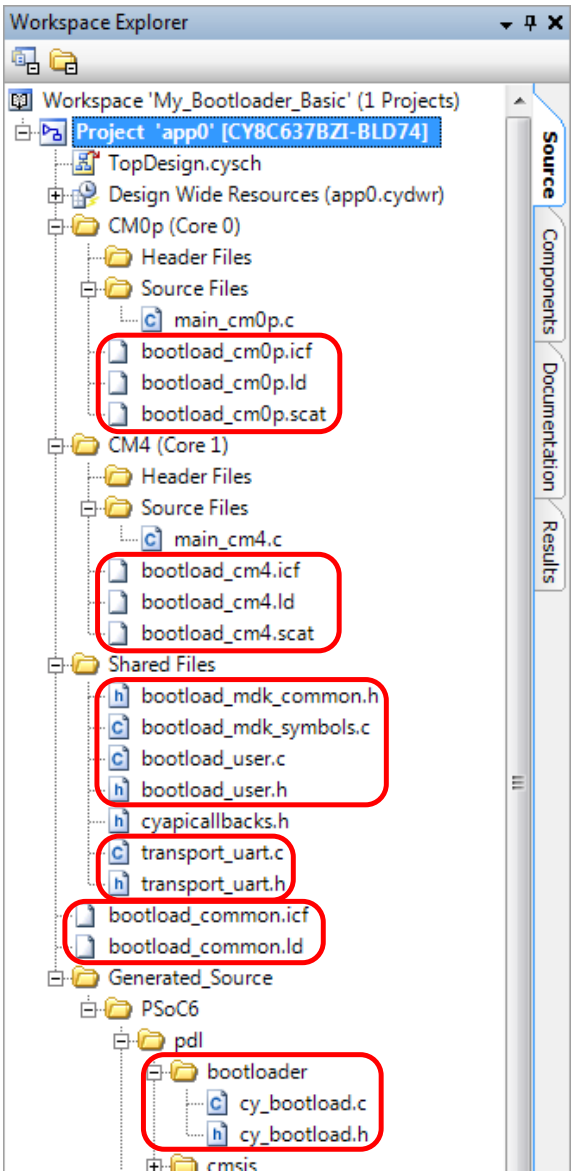
3. Generate the project files. Click the project in the Workspace Explorer window, and select menu item **Build** > **Generate Application**. The files listed in Table 1 on page 4 are created and added to the project, as Figure 8 shows.

   Note: PSoC 6 MCU has two CPU cores, ARM Cortex-M4 and Cortex-M0+. Each core has its own folder, plus there is a shared files folder. Linker script files to locate the code for each core, for a number of IDEs, are automatically created and added to the project. For more information, see AN215656, PSoC 6 MCU Dual-Core CPU system Design.

4. 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. In many cases the functions can be left unchanged.

5. Depending on which compiler you are using, edit the appropriate *common* linker script file, to encode the decisions made in Locate Applications in Memory.

Figure 8. Add Bootloader SDK Files to a Project



6. The 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 in *bootload_cm0p.ld* and *bootload_cm4.ld*:

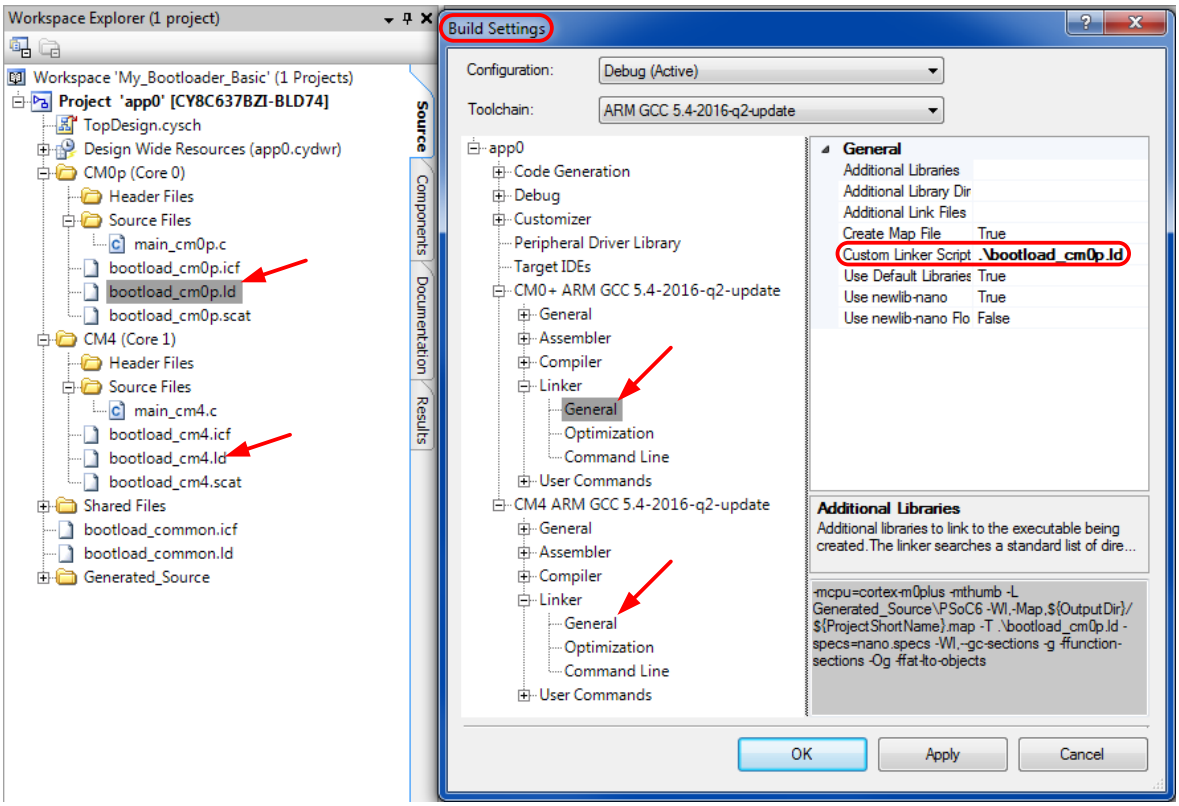
```
/*
* Bootloader SDK specific: aliases regions, so the rest of code does not use
* application specific memory region names
*/
REGION_ALIAS("flash_core0", flash_app1_core0);
REGION_ALIAS("flash",       flash_app1_core1);
REGION_ALIAS("ram",           ram_app1_core1);

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

7. Change the Project Build settings to use the bootloader linker script files instead of the default linker script files. Select the file *psoc6ble_cm0p.ld* for the Cortex-M0+ core, as Figure 9 shows. Include the relative path to the file. Select the file *psoc6ble_cm4.ld* for the Cortex-M4 core (not shown in Figure 9).

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

   Note: This operation must be done for both Debug and Release configuration.

Figure 9. Project Build Settings for Custom Linker Scripts



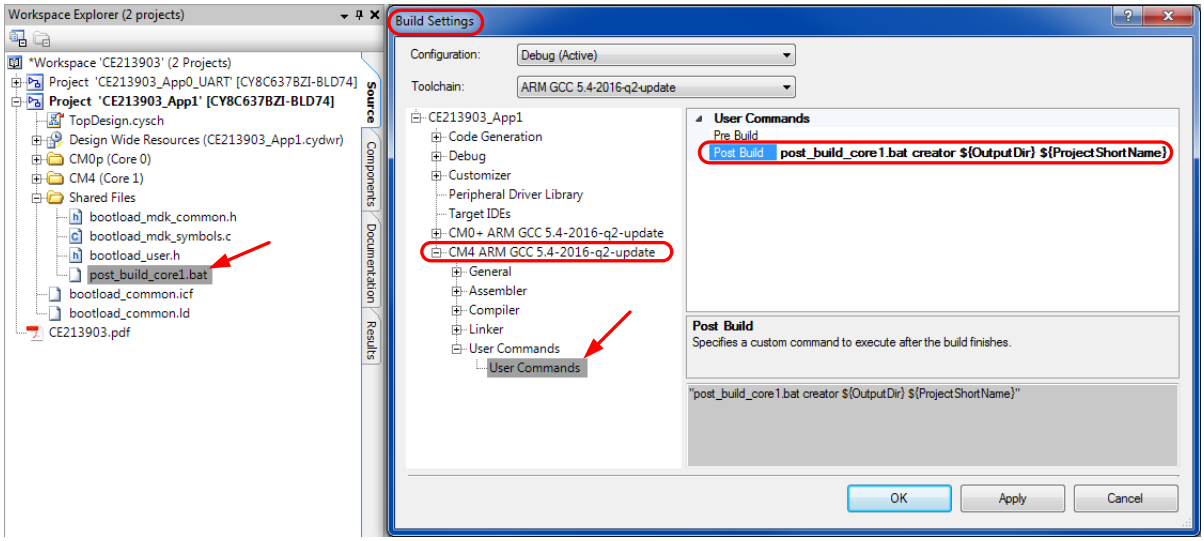
8. For updateable application projects, add a post-build batch file to call the Cypress utility program *cypdlelftool.exe*. cypdlelftool is included with your PSoC Creator installation. It generates a **.cyacd2* file, which is downloaded by the bootloader host (see Figure 5 on page 6). The batch file is applied only to the Cortex-M4 binary.

   Note: 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\CE213903_App1.cydsn*, as Figure 10 shows. The batch file contents are also available in Appendix E. For convenience, you can copy and paste the file into your project folder *app1.cydsn*, and add it to the project *Shared Files* folder.

   Then add a post-build command to the Cortex-M4 build, as Figure 10 shows.

   Note: This operation must be done for both Debug and Release configuration.

Figure 10 Post-Build Command



9. Add code as needed to the two *main_xxx.c* and other source files. In your overall design, consider when each application shall pass control to another application. Add calls to Bootloader SDK API functions `ValidateApplication()` and `ExecuteApplication()` as needed.

   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 Application

Build all of the applications. Several output files are created (see also Figure 5 on page 6):

- A *.hex* file for application #0 (App0). Use the *.hex* file for factory programming.

  Note: Using cypdlelftool, it is possible to create a *.hex* file that incorporates multiple applications, for example App0 and App1, for factory programming.

- A *.cyacd2* file for each installable application (see Step 8 in the previous section). Use these files for field updates, that is, for bootloading.

Program the *.hex* file into the device, using either PSoC Creator or PSoC Programmer. For more information, see the Help menus in these tools.

At a later time, you can use the bootloader in application #0, as well as a bootloader host program, to download and install application *.cyacd2* files into the device. See Appendix A, Bootloader Host Program.

5 Bootloader Code Examples

There are several code examples associated with this application note. They demonstrate the different ways that bootloading 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 3 shows an overview-level list of the code examples:

Table 3. List of Bootloader Code Examples

CE #	Title	Description
CE213903	PSoC 6 MCU Basic Bootloaders	A set of examples that demonstrate a number of basic bootloading operations; see Figure 6 on page 7, memory map A: • Downloading an application from a host, using various PSoC Creator Components for host communication: UART, I²C, and SPI. At this time, only UART and I²C are supported. • Installing the downloaded application into flash or external memory • Validating an application, and then transferring control to that application
The following code examples are planned or in development:		
CE216767	PSoC 6 MCU BLE Bootloader	Same as above, but uses the Bluetooth low-energy (BLE) system as the communication channel
CE2xxxxx	PSoC 6 MCU BLE Bootloader with Upgradeable Stack	Same as above, plus the BLE stack can be upgraded
CE2xxxxx	USB HID Bootloader	Same as CE231903, but uses USB HID as the communication channel
CE2xxxxx	USB Mass Storage Bootloader	Same as CE231903, but uses USB Mass Storage as the communication channel
CE2xxxxx	Bootloader with Multiple Applications	Bootloads multiple applications; see Figure 6, memory map B. Each application can bootload another application.
CE2xxxxx	Golden Image Bootloader	Demonstrates the golden image bootloading application
CE2xxxxx	Bootloader with Encryption	Same as CE213903, but the application is encrypted
CE2xxxxx	PSoC 6 MCU BLE Bootloader with External Memory Source	Same as CE216767, but the bootload source is external memory
CE2xxxxx	Embedded Host Program	Similar to that described in AN60317, PSoC 3/4/5LP 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 project in PSoC Creator.

Generally, all PSoC Creator applications in a code example are in the same workspace. As noted in Design the Application, projects can exist in separate workspaces as well as in separate locations on your computer.

Usually App0 does the bootloading; it downloads and installs 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.

Note: At this time, only UART and I²C are supported.

The code examples support the CY8CKIT-062-BLE PSoC 6 BLE Pioneer Kit. Applications typically blink a kit LED at different rates, making it easy to tell which application is currently running.

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.

5.1 How to Build the Bootloader Code Examples

Following are step-by-step instructions showing how to build and operate each of the bootloader code examples listed in Table 3 on page 13. For more information, see How to Use the SDK.

The instructions are based on PSoC Creator; they can be adapted for other IDEs.

5.1.1 PSoC 6 MCU Basic Bootloader Code Examples

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

1. Create App0 and the workspace.
2. Configure App0 as a bootloader.
3. Add App1.
4. Configure App1 as an installable application.
5. Build the Bootloader and Application.

1. Create App0 and the workspace.

You can create a PSoC Creator project and its containing workspace at the same time. This step shows you how.

Open PSoC Creator. Select menu item **File** > **New** > **Project...**. The **Create Project**, **Select project type** window appears, as Figure 11 shows.

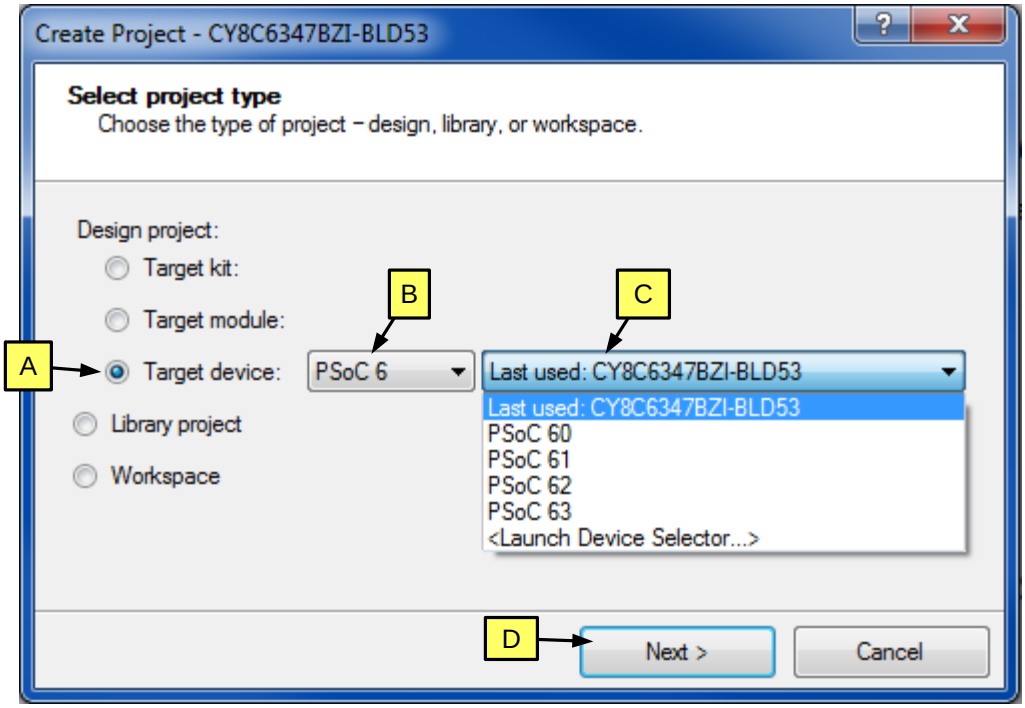
A. Click Target device.

B. In the first drop-down menu, select **PSoC 6**.

C. 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.

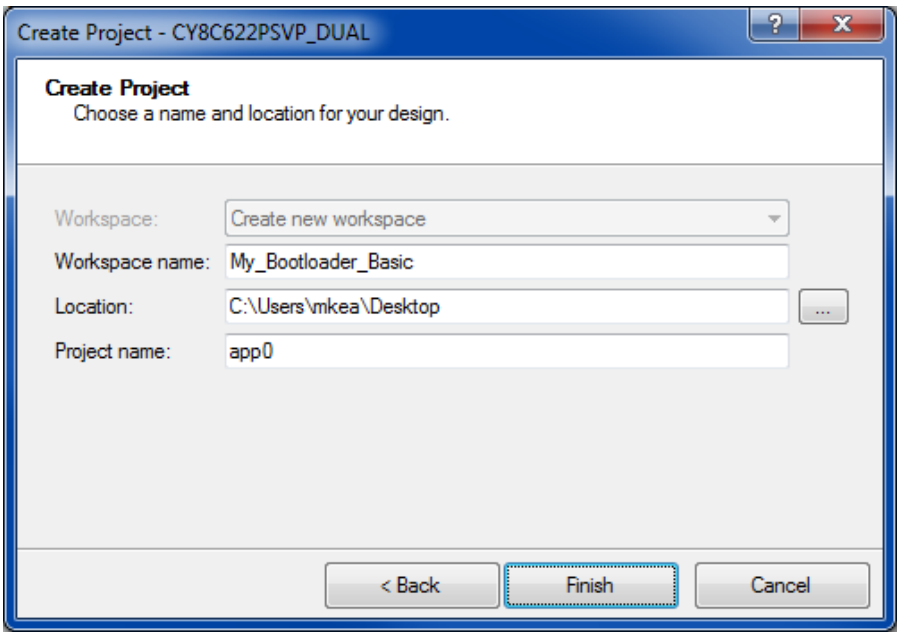
D. Click **Next**.

Figure 11. 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 12 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 of the project files.

Figure 12. Create the Project



The new workspace and project files are shown in the Workspace Explorer window—see Figure 8 on page 10.

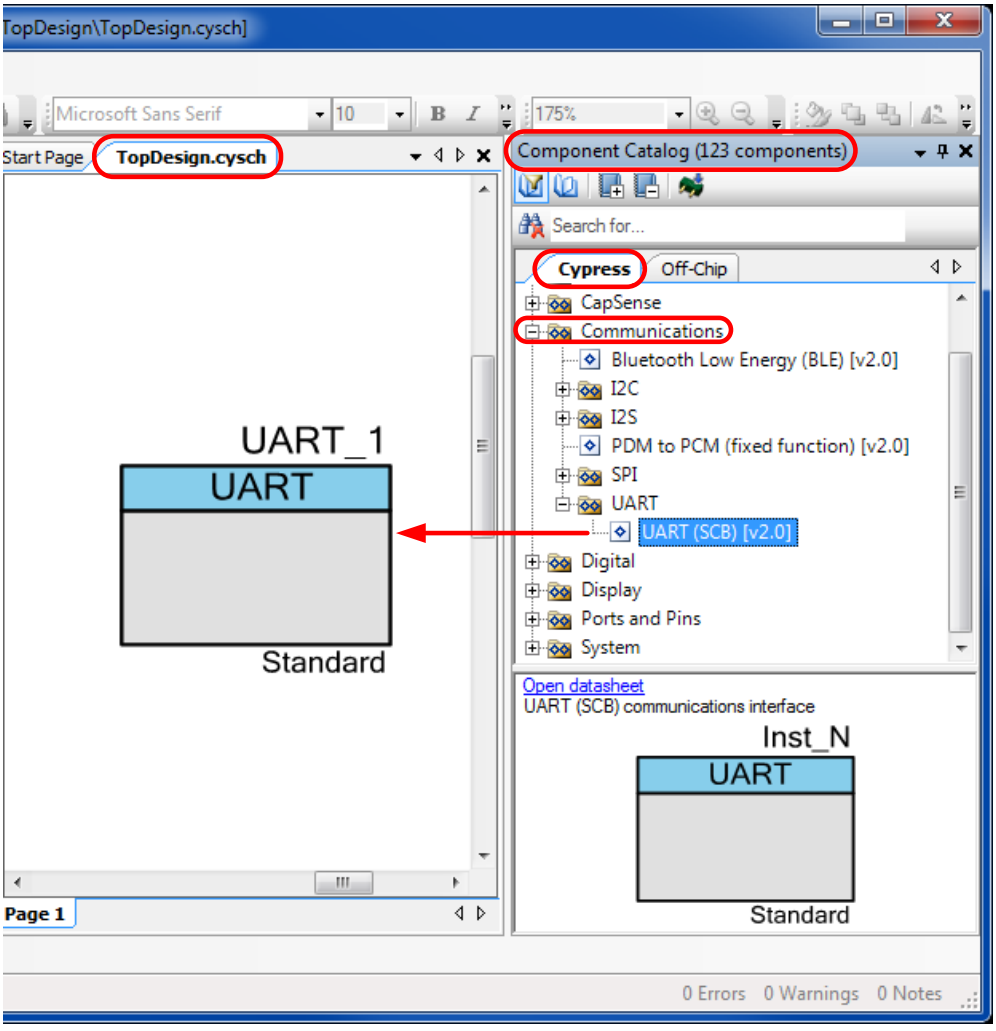
2. Configure App0 as a bootloader.

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.
- Add Bootloader SDK and template files to the project

2a. 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 13 shows. Drag the Component onto the schematic.

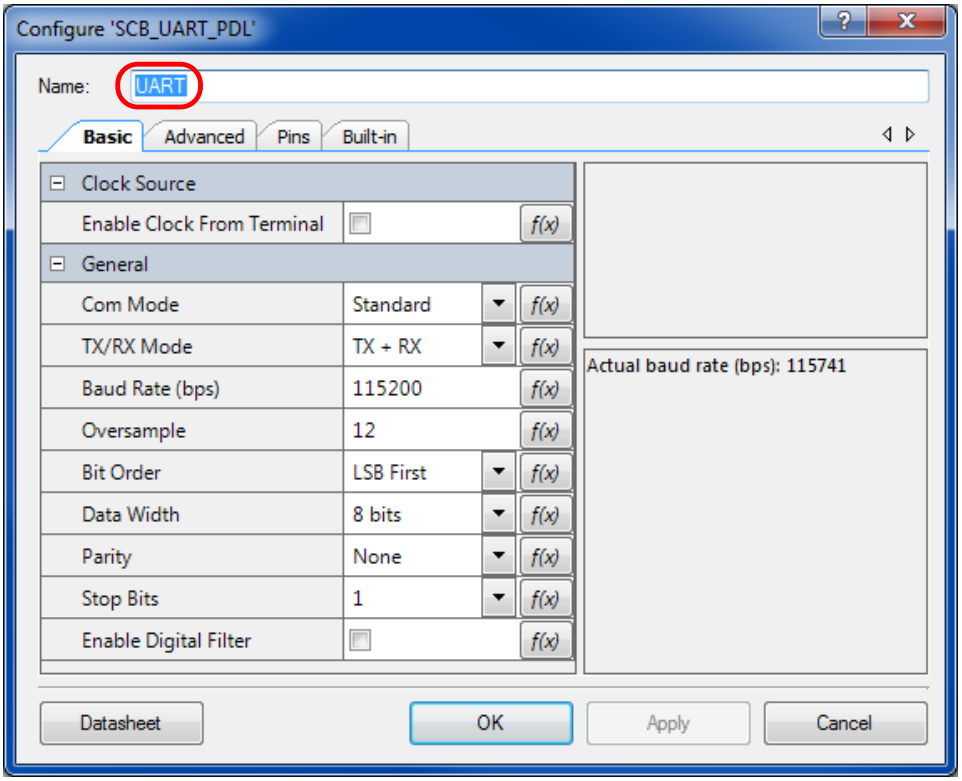
Figure 13. Add a Communication Component to a Bootloader Project



Double-click the Component on the schematic to configure its parameters, such as baud rate, number of bits, etc., as Figure 14 shows. Figure 14 shows the default parameter settings except that the **Name** is changed from UART_1 to UART. For I2C Component, change the **Name** from "I2C_1" to "I2C." This is recommended to work with the default Bootloader SDK files that are added to your project later.

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

Figure 14. Configure the Communication Component



2b. Add other Components. 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 code example CE213903, and then modify the schematic for your application.

In the Design Wide Resources window, Pins tab, connect the UART or I2C and other Component pins to the appropriate physical pins. Use CE213903 and the Guide for your kit as guidance.

2c. Add bootloader files to the project. First, incorporate the Bootloader SDK into the project—see Figure 7 on page 9. Then select **Build** > **Generate Application**. When done, the project should look like Figure 8 on page 10, with the addition of some startup and other non-bootloader files.

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 code example CE213903, and then modify the code for your application.

For I^2C bootloader, copy the files *transport_i2c.h* and *transport_i2c.c* from code example CE213903 to your project, and add them to the Shared Files folder. Then edit the file *bootload_user.c* as follows:

- Change `#include "transport_i2c.h"` to `#include "transport_i2c.h"`.
- Change five instances of "`UART_Uart`" to "`I2C_I2c`".

2d. Build the Project. Change the project build settings to use the template linker script files – see Figure 9 on page 11. Then select **Build** > **Build app0**.

2e. Test the Project. Before continuing, you can test your new app0 by programming it into a CY8CKIT-062-BLE PSoC 6 BLE Pioneer Kit. Then build the CE213903_App1 project in code example CE213903. Finally, use the Bootloader Host Program to bootload CE213903_App1 to the kit—see instructions in Appendix A.

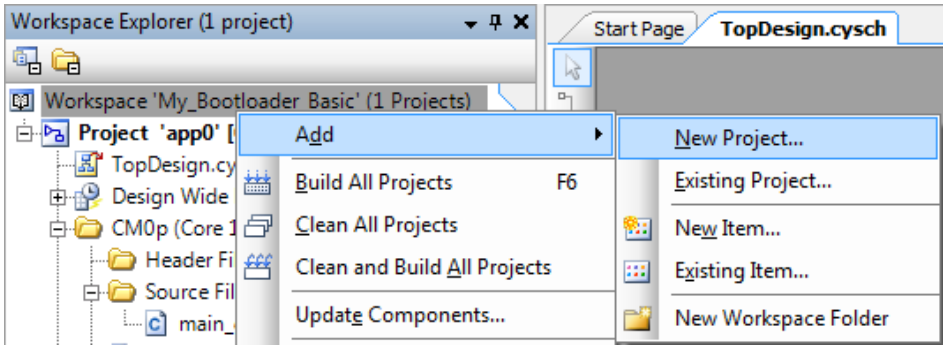
3. Add App1.

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 and/or folders. 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 developing applications, it is a good idea to develop a workspaces / projects plan for your overall system development needs.

In the PSoC Creator Workspace Explorer window, right-click the **Workspace**, then select **Add** > **New Project...**, as Figure 15 shows:

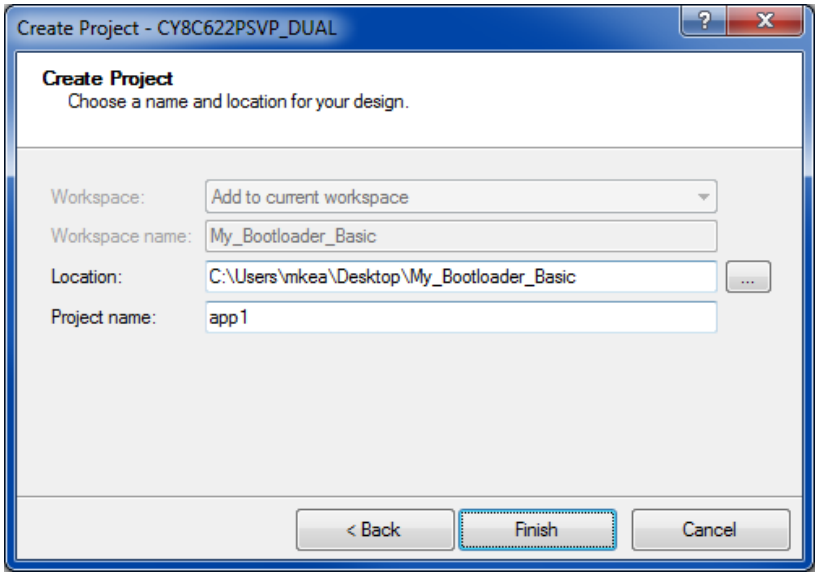
Figure 15. Add a New Project to a PSoC Creator Workspace



The Create Project, Select project type window appears, see Figure 11 on page 14. Make sure that the device selected is the same as for App0. Press **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 16 shows. The default Location is the workspace folder. Click **Finish**. A new folder *app1.cydsn* is created within the workspace folder; the *.cydsn* folder contains all of the project files.

Figure 16. Create a Second Project



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

   Note: App1 must have the same target device as App0. It should also have the same clock configuration (see *.cydwr* file, **Clocks** tab) and be built with the same toolchain (GCC or MDK) as App0, or application transfer may fail.

   4a. 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 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 as guidance.

   4b. Add Bootloader SDK and template files **to the project.** Incorporate the Bootloader SDK into the project—see Figure 7 on page 9. Check 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 8 on page 10, 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 9 on page 11.

   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 10 on page 12.

   4c. 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 code example CE213903, and then modify the code for your application.

5. **Build the Bootloader and Application.**

   After all projects are created, build them as described in Build and Program the Applications. Click the menu item **Build**, and select the appropriate build option.

   The code example applications are now ready to be installed in a kit, either at the factory or bootloaded in the field.

6 Related Documents

Application Notes	
AN210781 – Getting Started with PSoC 6 MCU with Bluetooth Low Energy (BLE) Connectivity	Introduces the PSoC 6 MCU with Bluetooth Low Energy (BLE) Connectivity
Code Examples	
CE213903 – PSoC 6 MCU Basic Bootloaders	Describes UART and I2C Bootloaders for PSoC 6 MCU
Device Documentation	
PSoC 6 MCU Datasheets	PSoC 6 Technical Reference Manuals
Development Kit (DVK) Documentation	
PSoC 6 MCU Kits	

Appendix A Bootloader Host Program (BHP)

The Bootloader Host Program (BHP) is a standalone graphical tool provided with PSoC Creator. This tool is used to communicate with a PSoC device that has a bootloader installed.

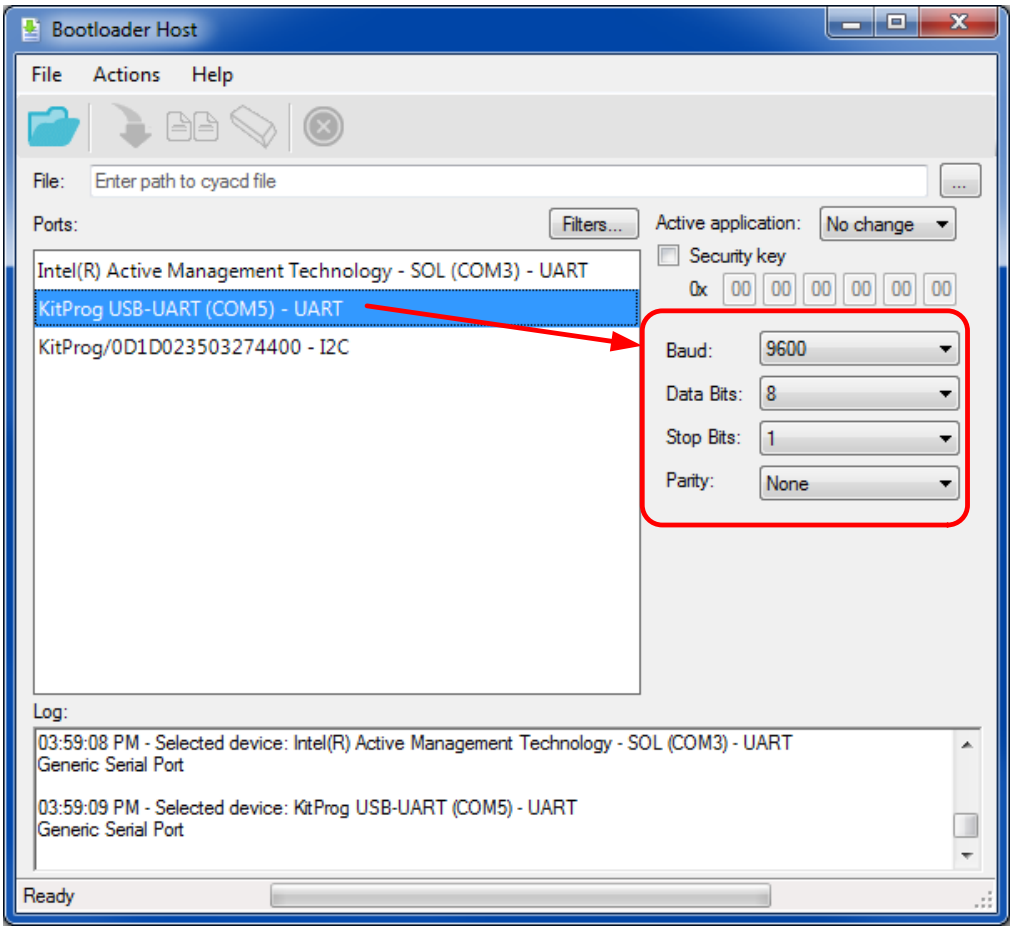
Note: You cannot use the BHP to install a bootloader into a device. Instead, you must program a bootloader into flash through the device SWD/JTAG port, using other tools such as PSoC Creator or PSoC Programmer. After a bootloader is installed, you can use the BHP to install a bootloadable application.

The BHP supports communicating with a number of Cypress MCU devices, via I^2C, SPI, UART, or USB, as Figure 17 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 I^2C and SPI, a special communication port is needed, such as a KitProg module. The port configuration fields change depending on the selected port.

Note: At this time, only the UART and I^2C bootloaders are supported.

Note: BHP does not support Bluetooth Low-Energy (BLE). Use Cypress's CySmart™ product instead, see CySmart User Guide section "Updating Peripheral Device Firmware".

Figure 17. Bootloader Host Program Graphical User Interface (GUI)



Using the Bootloader Host application, you can:

- Download and install a bootloadable application to a device
- Verify a bootloadable application that is already installed on a device

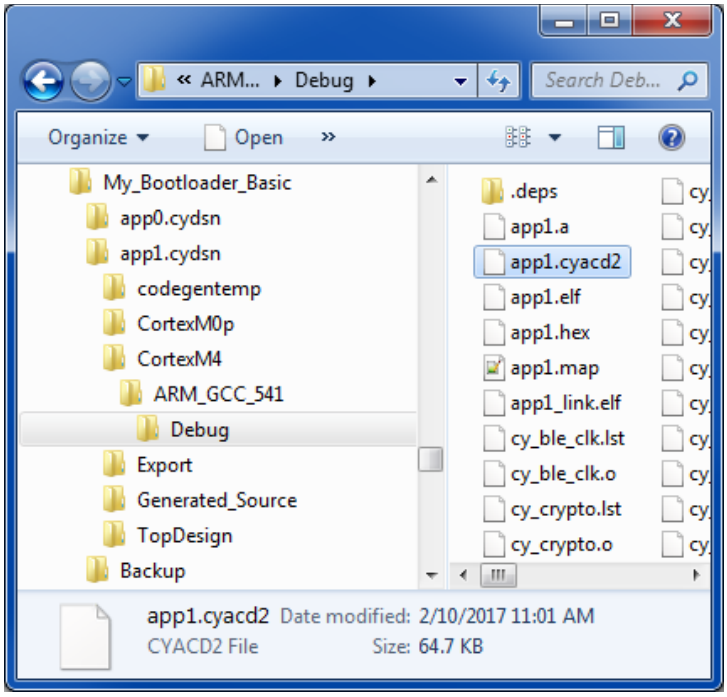
- Erase a bootloadable application from a device

Note: The source code for the BHP is provided in *<PSoC Creator install* folder>*cybootloaderutils*. It contains routines to do file parsing and command construction. You can add a GUI and a communication protocol to build your own host program.

To use the BHP, do the following:

1. Select the communication port that matches the communication Component in your bootloader. For more information, see Figure 13 on page 16.
2. Click the **File** icon, or menu item **File** > **Open**, and navigate to your application's *.cyacd2* file, as Figure 18 shows. This file is created when a bootloadable application project is built by PSoC Creator; for more information, see Build and Program the Applications.

Figure 18. Select the .cyacd or .cyacd2 File



3. Make sure that the bootloader is running in your target device. Click the BHP **Program** icon, or menu item **Actions** > **Program**. The selected application is downloaded and installed in the target device. Depending on your system-level design (see Determine the Applications in Your System), control is transferred from the bootloader to the downloaded application.

Appendix B Host Command / Response Protocol

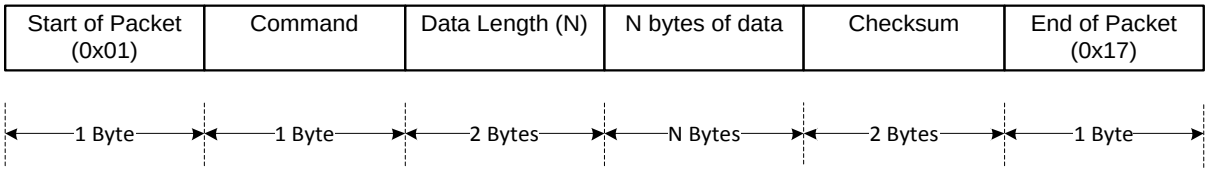
The bootloader communicates with a host using a simple command-response protocol, regardless of communication channel used. The bootloader receives commands from the communication channel, and responds to each command by sending one or more bytes to the communication channel. See Figure 2 on page 2.

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

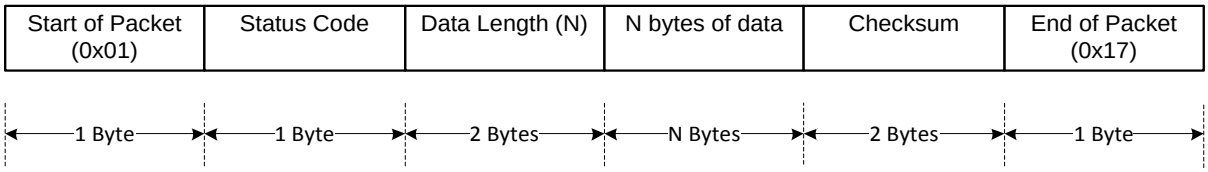
Communication packets sent from the host to the bootloader have the structure shown in Figure 19:

Figure 19. Bootloader Command Packet Structure



Response packets sent from the Bootloader to the host have the structure shown in Figure 20:

Figure 20. Bootloader Response Packet Structure



All multi-byte fields are LSB first.

B.2 Commands

Table 4 shows a list of all commands supported by the Bootloader SDK. All commands except Exit Bootloader are ignored until the Enter Bootloader command is received.

Table 4. Bootloader Commands List

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

There is no specific requirement for command execution time.

Table 5 shows a list of all status and error codes supported by the Bootloader SDK.

Table 5. Bootloader Status and Error Codes List

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

B.2.1 Enter Bootloader

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

B.2.2 Sync Bootloader

Resets the bootloader to a known state, making it ready to accept a new command. Any data that was buffered is discarded. This command is only needed if the bootloader 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 Bootloader

Exits from the bootloader. Ends the bootload 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 bootloader. 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 bootloader. This reduces the bootload 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 device internal flash or page of external non-volatile memory (NVM). May follow a series of Send Data 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 device internal flash or page of SMIF. May follow a series of Send Data commands.

This command is optional; its presence depends on a user configuration.

- 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.

- 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 Structure.

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

- 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.

- 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 bootloader to decrypt data before writing it to flash.

This command is optional; its presence depends on a user configuration.

- 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

.cyacd2 file data is in the form of ASCII hex numbers. 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 three types:

- Header
- Encryption Initial Vector
- Data

A header row has the structure shown in Figure 21:

Figure 21. *.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 bootloader packet (see Appendix B, Command / Response Packet Structure). 0 = checksum, 1 = CRC.
- App ID: See Figure 6 on page 7. This also controls which portion of the application metadata is updated for this application.

An encryption initial vector row has the structure shown in Figure 22:

Figure 22. *.cyacd2* Encryption Initial Vector Row Structure

Header	Encryption Initial Vector
5 characters: "`@EIV:`"	0, 8, or 16 bytes

A data row has the structure shown in Figure 23:

Figure 23. *.cyacd2* Data Row Structure

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

Appendix D Application Metadata Structure

The Bootloader SDK uses a designated region of NVM to store information about the applications—see Figure 6 on page 7. 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

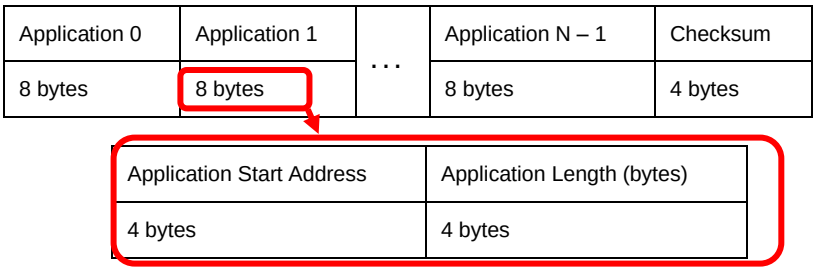
As noted in Figure 6, metadata typically occupies one flash row or NVM page. (In devices with small amounts of flash, multiple rows or pages may be used.) Figure 6 also shows that metadata is located outside of any application.

There are two *.elf* file (see Figure 5 on page 6) symbols that control the metadata location and size:

- `__cy_boot_metadata_addr`—defines the start address of metadata in flash or NVM.
- `__cy_boot_metadata_length`—defines the length of metadata in bytes. It must be a multiple of the flash row or NVM page length.

Application metadata has eight bytes of data per application, followed by four bytes for checksum, as Figure 24 shows:

Figure 24. Metadata Structure



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

```
#define BOOTLOAD_MAX_APPLICATIONS (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 bootloader commands Program Data and Verify Data. The default algorithm is CRC-32C.

Appendix E Post-Build Batch File Listing

Following is a listing of the post-build batch file for App1, from CE213903.

```
@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_PDL_ELF_TOOL=%PDL_PATH%"\tools\win\elf\cypdlelftool.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_PDL_ELF_TOOL% -P %OUTPUT_DIR%\%PRJ_NAME%%ELF_EXT% --output %OUTPUT_DIR%\%PRJ_NAME%.cyacd2
```

Document History

Document Title: AN213924 - PSoC 6 MCU Bootloader Software Development Kit (SDK) Guide

Document Number: 002-13924

Revision	ECN	Orig. of Change	Submission Date	Description of Change
**	5653720	MKEA	03/08/2017	New application note
*A	5810045	MKEA	07/10/2017	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	5866384	MKEA	09/11/2017	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.

Worldwide Sales and Design Support

Cypress maintains a worldwide network of offices, solution centers, manufacturer's representatives, and distributors. To find the office closest to you, visit us at Cypress Locations.

Products

ARM® Cortex® Microcontrollers	cypress.com/arm
Automotive	cypress.com/automotive
Clocks & Buffers	cypress.com/clocks
Interface	cypress.com/interface
Internet of Things	cypress.com/iot
Memory	cypress.com/memory
Microcontrollers	cypress.com/mcu
PSoC	cypress.com/psoc
Power Management ICs	cypress.com/pmic
Touch Sensing	cypress.com/touch
USB Controllers	cypress.com/usb
Wireless Connectivity	cypress.com/wireless

PSoC® Solutions

PSoC 1 | PSoC 3 | PSoC 4 | PSoC 5LP | PSoC 6

Cypress Developer Community

Forums | WICED IOT Forums | Projects | Videos | Blogs | Training | Components

Technical Support

cypress.com/support

Cypress Semiconductor
198 Champion Court
San Jose, CA 95134-1709