PSoC® 3 and PSoC 5LP I^{2}C Bootloader

Authors: Anu M D, Tushar Rastogi

Associated Project: Yes
Associated Part Family: All PSoC® 3 and PSoC 5LP parts
Software Version: PSoC Creator™ 4.0 and higher
Related Application Notes: For a complete list of the application notes, click here

AN60317 describes an I^{2}C-based bootloader for PSoC® 3 and PSoC 5LP. In this application note you will learn how to use PSoC Creator™ to quickly and easily build an I^{2}C-based bootloader project, and bootloadable projects. It also shows how to build an I^{2}C-based embedded bootloader host program.

Contents

1 Introduction

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. At the factory, initial programming of firmware into a product is typically done through the MCU's Joint Test Action Group (JTAG) or Serial Wire Debugger (SWD) interface. However, these interfaces are usually not accessible in the field.

This is where bootloading comes in. Bootloading is a process that allows you to upgrade your system firmware over a standard communication interface such as USB, I^{2}C, UART or SPI. A bootloader communicates with a host to get new application code or data, and writes it into the device's flash memory.

In this application note you will learn:

- How to add an I2C bootloader to a PSoC Creator™ project for PSoC® 3 or PSoC 5LP
- How to use the Bootloader Host Tool
- The basic building blocks and functionality of a bootloader host system

- How to create an embedded I2C bootloader host using PSoC 5LP

This application note assumes that you are familiar with PSoC and the PSoC Creator IDE. If you are new to PSoC 3 or PSoC 5LP, refer to AN54181, Getting Started with PSoC 3 or AN77759, Getting Started with PSoC 5LP. If you are new to PSoC Creator, see the PSoC Creator home page.

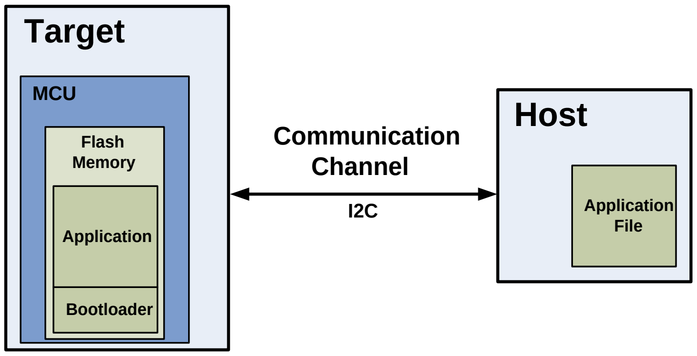
This application note also assumes that you are familiar with bootloader concepts. If you are new to these concepts, see AN73854, PSoC 3 and PSoC 5LP Introduction to Bootloaders. For a complete list of other application notes on bootloading, click here.

Finally, this application note assumes that you are familiar with the I^{2}C protocol and the PSoC Creator I^{2}C Master and Slave Components. If you are new to these Components, see the PSoC Creator I2C Component datasheet. You can also get the datasheet by right-clicking on an I2C Component in PSoC Creator.

1.1 Terms and Definitions

Figure 1 illustrates the main elements in a bootloader system. It shows 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**. The other terms in Figure 1 are defined below.

Figure 1. Bootloading System Diagram



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 PC 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 **bootloadable**.

Another common term for bootloading is **in-system programming (ISP)**. Cypress has a product with a similar name called In-System Serial Programmer (ISSP) and an operation called Host-Sourced Serial Programming (HSSP). For more information, see AN73054, PSoC Programming Using an External Microcontroller (HSSP).

1.2 Using a Bootloader

A bootloader communication port is typically shared between the bootloader and the actual application. The first step to use a bootloader is to manipulate the product so that the bootloader, and not the application, is executing.

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.

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 sent, the bootloader can pass control to the new application.

1.3 Bootloader Function Flow

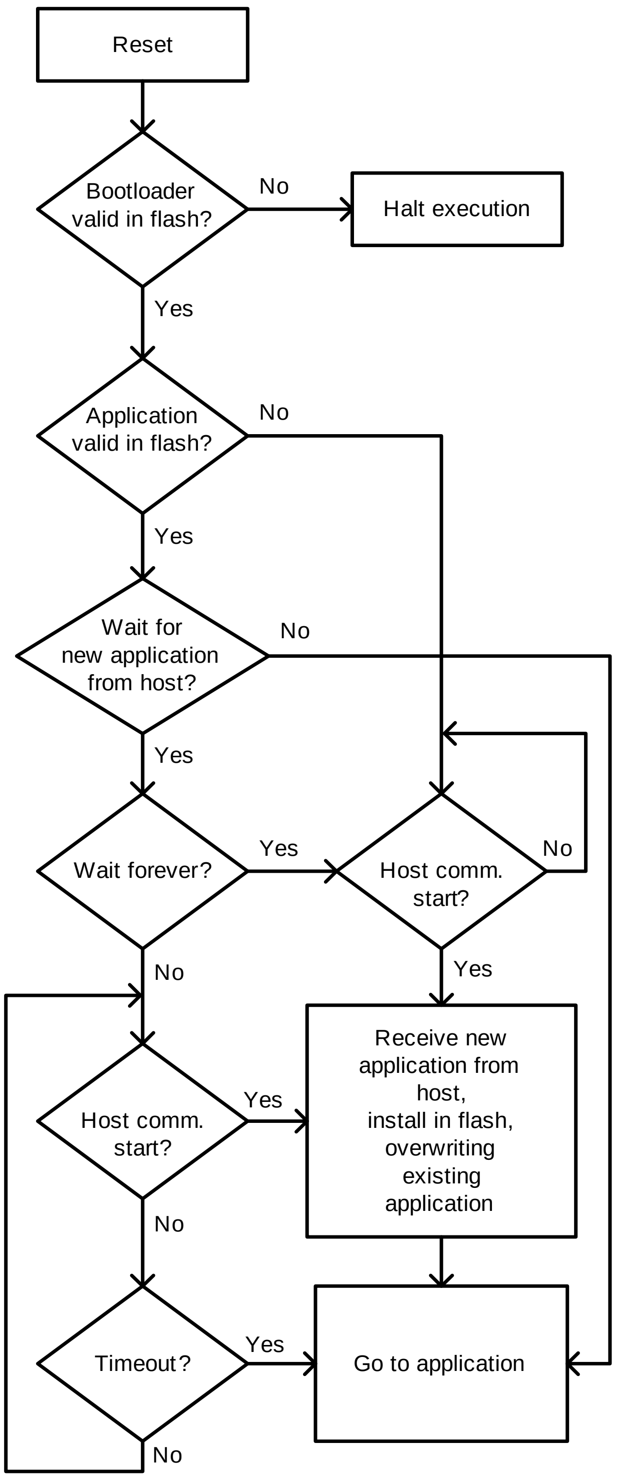
Typically when the device resets, the bootloader is the first function to execute. It then performs the following actions:

- Checks the application's validity before letting it run
- Manages the timing to start host communication
- Does the bootload / flash update operation

- And finally, passes control to the application

Figure 2 on page 3 is a flow diagram that shows how this works.

Figure 2. Bootload Process Flowchart


Reset
Bootloader valid in flash?
No
Halt execution
Yes
Application valid in flash?
No
Yes
Wait for new application from host?
No
Yes
Wait forever?
Yes
Host comm. start?
No
Yes
No
Receive new application from host, install in flash, overwriting existing application
Host comm. start?
Yes
No
Timeout?
Yes
Go to application
No

1.4 Techniques to Enter Bootloader

As mentioned previously, the bootloader is the first function to run at reset. As Figure 2 shows, the bootloader code waits for the host for a short period of time before passing control to the application. This may cause the host to miss an opportunity to start the bootload operation. However, another way exists to start bootloading, and that is to pass control from the application or bootloadable back to the bootloader.

1.4.1 Bootloadable API

The Bootloadable Component in PSoC Creator has an API (Application Programming Interface) to start the bootloader: Bootloadable_Load(). This allows the host to start a bootload operation at any time.

The problem with this method is that you must depend on the application code to perform an application upgrade. What happens if the application has a defect that prevents transfer of control to the bootloader?

1.4.2 Customize Bootloader

Instead, it may be better to have the bootloader wait an infinite amount of time for the host. To do that, we can customize the bootloader project to check for some user input before calling Bootloader_Start() and running through its normal routine.

2 Projects

Now that you have seen how bootloaders are implemented in PSoC and PSoC Creator, let us look at some specific examples of how this is done.

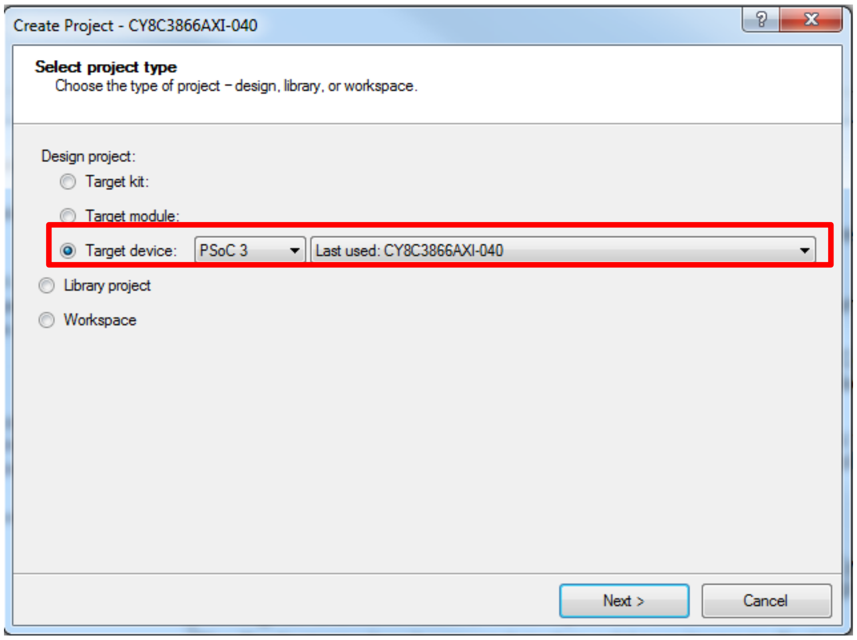
This section shows you the steps to create PSoC Creator bootloader, bootloadable and embedded bootloader host projects. The projects are designed to be used with the CY8CKIT-030 and CY8CKIT-050 kits. They require PSoC Creator 3.0 SP1 or higher.

2.1 I^{2}C Bootloader

In this section, we create and build an I^{2}C based bootloader project.

1. Create a new PSoC Creator project.
   a. Select the target device as PSoC 3 or PSoC 5LP as Figure 3 shows.
   b. Select an empty schematic as shown in Figure 4.
   c. Name the workspace and the project as I2C_Bootloader as Figure 5 shows.

Figure 3. Selecting Device for Bootloader Project

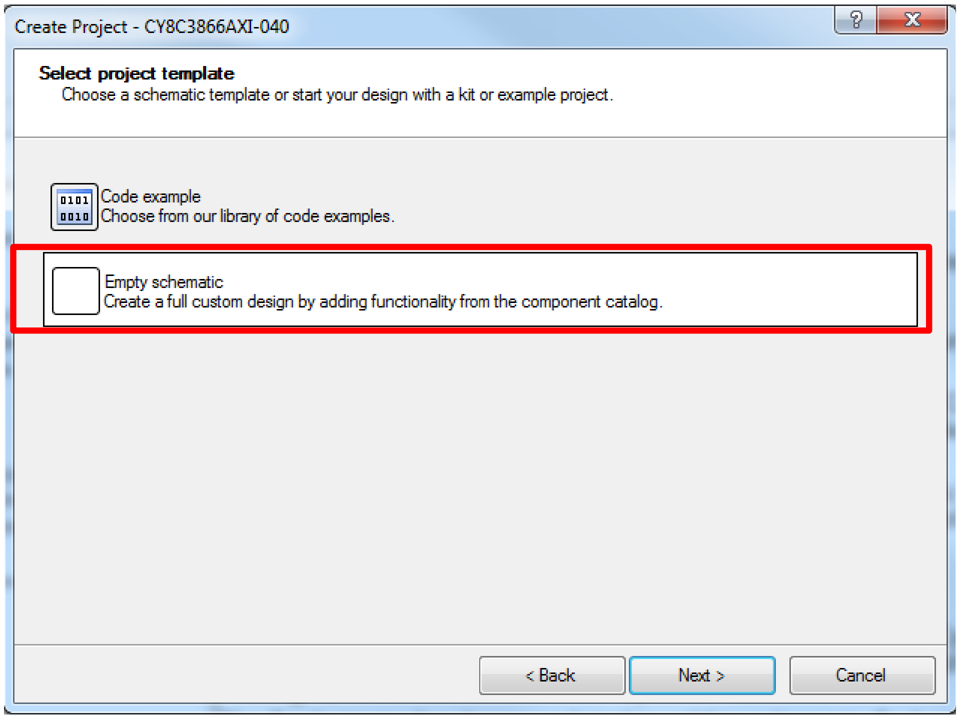


Create Project - CY8C3866AXI-040

Select project type
Choose the type of project – design, library, or workspace.

Design project:
- Target kit:
- Target module:
- Target device: PSoC 3 | Last used: CY8C3866AXI-040 (selected)
- Library project
- Workspace

Figure 4. Selecting empty schematic for I2C_Bootloader Project



Create Project - CY8C3866AXI-040

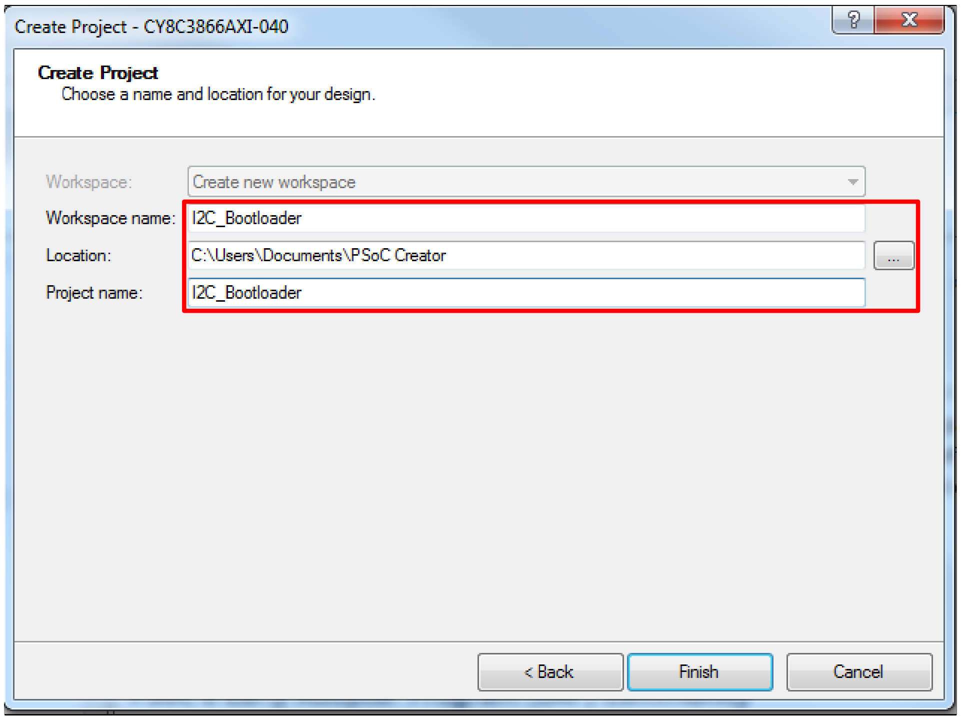
Select project template

Choose a schematic template or start your design with a kit or example project.

Code example
Choose from our library of code examples.

Empty schematic
Create a full custom design by adding functionality from the component catalog.

Figure 5. Creating I2C_Bootloader Project



Create Project - CY8C3866AXI-040

Create Project

Choose a name and location for your design.

Workspace: Create new workspace
Workspace name: I2C_Bootloader
Location: C:\Users\Documents\PSoC Creator
Project name: I2C_Bootloader

2. Add I2C Slave Macro and Bootloader Components to the top design schematic. Rename the Components and Pins as Table 1 shows.

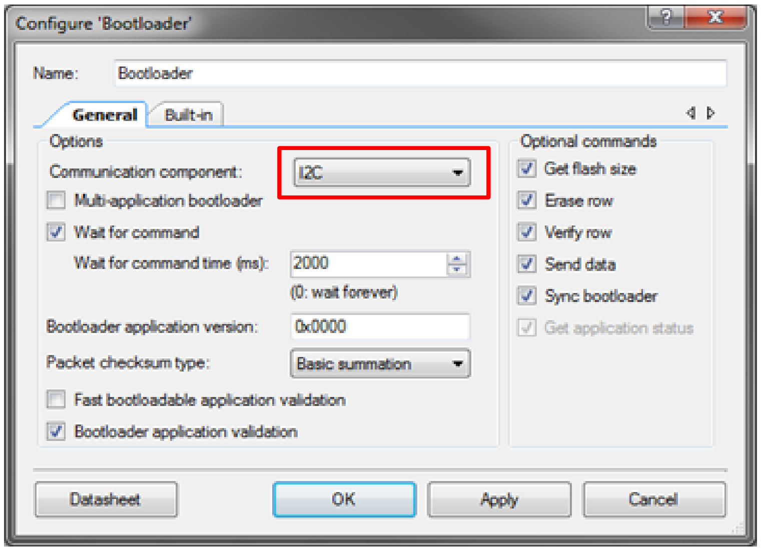
Table 1. Bootloader Project Component Names

Component	Name
Bootloader_1	Bootloader
I2C_1	I2C
SDA_1	Pin_SDAT
SCL_1	Pin_SCLK

The next step is to configure these Components.

3. To configure the Bootloader, double-click on the Component. Select I2C as the Communication component, as Figure 6 shows. Leave the other parameters at their default settings. For more information on these configuration parameters, please refer to the Bootloader Component datasheet.

Figure 6. Bootloader Configuration



Configure 'Bootloader'

Name: Bootloader

General | Built-in

Options

Communication component: I2C

☐ Multi-application bootloader

☑ Wait for command

Wait for command time (ms): 2000

(0: wait forever)

Bootloader application version: 0x0000

Packet checksum type: Basic summation

☐ Fast bootloadable application validation

☑ Bootloader application validation

Optional commands

☑ Get flash size

☑ Erase row

☑ Verify row

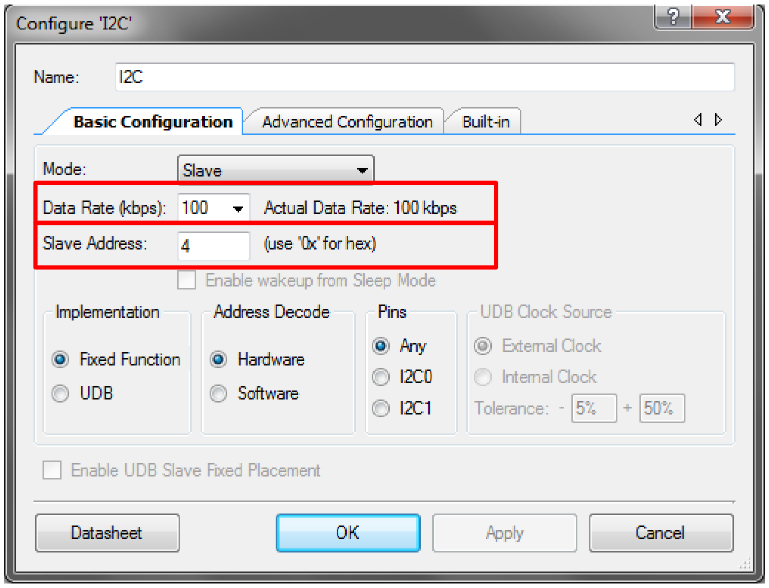
☑ Send data

☑ Sync bootloader

☑ Get application status

4. To configure the I2C Slave Component, double-click on it. By default, the data rate is set to 100 kbps and the slave address is set to 4. If you want to use a different address you can enter it in the slave address box. Leave the other parameters at their default settings. Figure 7 shows the basic configuration tab of the I^{2}C Slave Component.

Figure 7. Basic I^{2}C Slave Configuration


Configure 'I2C'

Name: I2C

Basic Configuration | Advanced Configuration | Built-in

Mode: Slave

Data Rate (kbps): 100 Actual Data Rate: 100 kbps

Slave Address: 4 (use '0x' for hex)

Enable wakeup from Sleep Mode

Implementation: (●) Fixed Function ( ) UDB

Address Decode: (●) Hardware ( ) Software

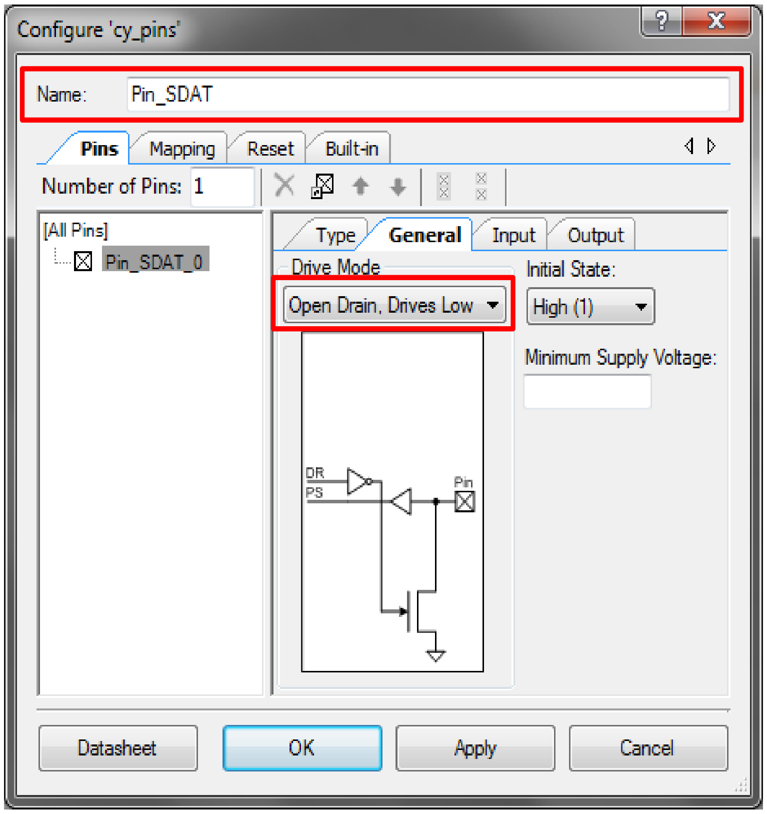
Pins: (●) Any ( ) I2C0 ( ) I2C1

UDB Clock Source: (●) External Clock ( ) Internal Clock; Tolerance: - 5% + 50%

Enable UDB Slave Fixed Placement

5. Double-click on the I^{2}C pins. Set the **Drive Mode** of the pins to **Open Drain, Drives Low**, as Figure 8 shows. You must also provide external resistive pull-ups of 1.8 kΩ on the SDA and SCL lines.

Figure 8. I2C Pin Configuration


Configure 'cy_pins'

Name: Pin_SDAT

Pins | Mapping | Reset | Built-in

Number of Pins: 1

[All Pins]
Pin_SDAT_0

Type | General | Input | Output

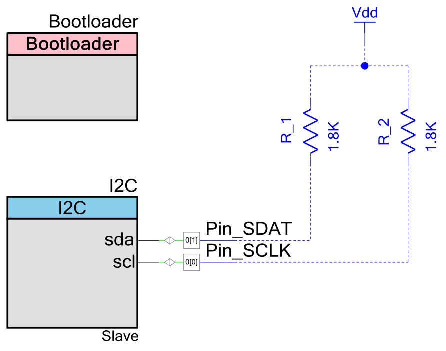
Drive Mode: Open Drain, Drives Low

Initial State: High (1)

Minimum Supply Voltage:

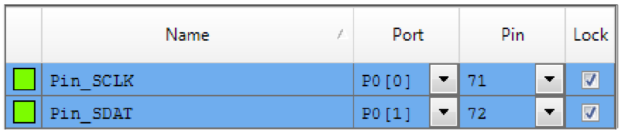
6. Now, with the pull-up resistors added as annotation Components, the top design of the project looks similar to Figure 9.

Figure 9. Top Design of I²C_Bootloader Project



7. Assign the Pin Components to physical pins. In the Workspace Explorer window, double-click the *I2C_Bootloader.cydwr* file, and click on the Pins tab. Assign the Pins as Figure 10 shows.

Figure 10: I²C Pin Assignments



	Name	Port	Pin	Lock
🟩	Pin_SCLK	P0[0]	71	✓
🟩	Pin_SDAT	P0[1]	72	✓

8. Review the *main.c* file – the CyBtldr_Start() function is added automatically when you create a bootloader project. Note that starting with PSoC Creator 3.2 and Bootloader component v1.40, a call to Bootloader_Start() has to be manually placed in the *main.c* file. This API function does the entire bootloading operation. It does not return – it ends with a software device reset. So, code after this API call is never executed.

9. Build the project and program it into a PSoC 3 device on CY8CKIT-030 or a PSoC 5LP on CY8CKIT-050.

2.2 Bootloadables

We shall now create two bootloadable projects. The first project displays "Bootloaded" on a character LCD. The second project demonstrates how to enter the bootloader project from a bootloadable project.

2.2.1 Bootloadable Project – Example 1

This section describes the steps to create the first bootloadable project:

1. Create a new PSoC Creator project as described in step 1 of I2C Bootloader section.
   a) Name the project as Bootloadable_1.
   b) The devices for this project and the I2C Bootloader project must be the same.
2. For this project we need the Bootloadable and Character LCD Components. Add these Components to your top design schematic. Name the Components according to Table 2.

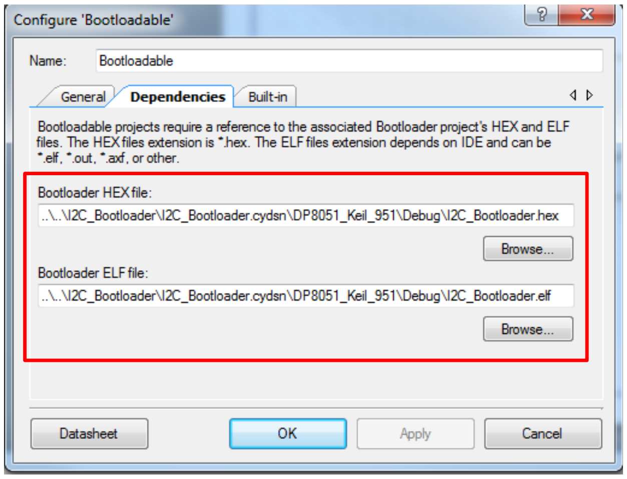
Table 2. Bootloadable1 Project Component Names

Component	Name
Bootloadable_1	Bootloadable
LCD_Char_1	LCD_Char

3. To configure the Bootloadable Component, double-click on it.

   A bootloadable project is always linked to the *.hex* file of a bootloader project. To do this, go to the dependencies tab and then link the bootloadable to the *I2C_Bootloader.hex* file, as Figure 11 shows. For more information on Bootloader Component configuration, refer to the Bootloader Component datasheet.

Figure 11. Bootloadable Component Configuration



Bootloadable projects require a reference to the associated Bootloader project's HEX and ELF files. The HEX files extension is *.hex. The ELF files extension depends on IDE and can be *.elf, *.out, *.axf, or other.

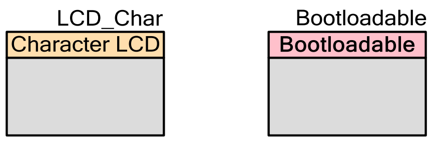
Bootloader HEX file: ..\..\I2C_Bootloader\I2C_Bootloader.cydsn\DP8051_Keil_951\Debug\I2C_Bootloader.hex

Bootloader ELF file: ..\..\I2C_Bootloader\I2C_Bootloader.cydsn\DP8051_Keil_951\Debug\I2C_Bootloader.elf

You may find the *I2C_Bootloader.hex* file in bootloader project's Debug / Release folder:
When PSoC 3 is the Bootloader,
..\I2C_Bootloader\I2C_Bootloader.cydsn\DP8051\ DP8051_keil_903\Debug\I2C_Bootloader.hex
When PSoC 5LP is the Bootloader,
..\I2C_Bootloader\I2C_Bootloader.cydsn\CortexM3\ ARM_GCC_441\Debug\I2C_Bootloader.hex

4. Now, the top design is complete; it should be similar to Figure 12.

Figure 12. Top Design of Bootloadable1 Project

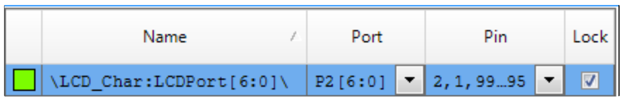


5. Add the following code to *main.c* to display the message "Bootloaded" on the LCD:

```
/* Initialize the LCD and display a
   message */
LCD_Char_Start();
LCD_Char_PrintString("Bootloaded");
```

6. Assign the Pin Components to physical pins. In the Workspace Explorer window, double-click the *Bootloadable1.cydwr* file and assign the pins as Figure 13 shows.

Figure 13. Pin Assignment of Bootloadable1 Project



	Name	Port	Pin	Lock
🟩	\LCD_Char:LCDPort[6:0]\	P2[6:0]	2,1,99..95	✓

7. Build the project. When a bootloadable project is built, PSoC Creator generates a *.cyacd* file. This is the file that is bootloaded onto the target. For more information on this file and its contents, see Appendix B.

Note: For PSoC Creator versions before 3.0, if the bootloader is updated, you must also rebuild all bootloadable projects that depend on that bootloader project. Use the “Clean and Build” option.

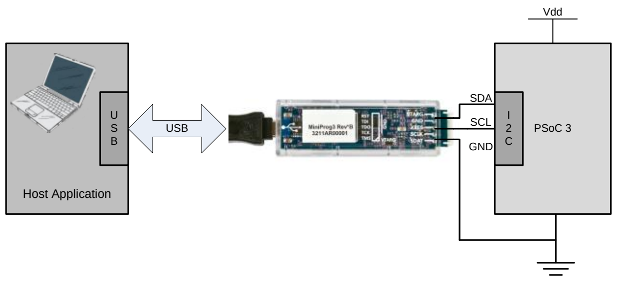
Note: Flash protection settings for a bootloadable project are ignored; the associated bootloader project’s flash protection settings take precedence.

Now, let us bootload this project into PSoC 3 using PSoC Creator.

2.2.2 Bootloading Using a PC Host

A bootloader host executable is provided with PSoC Creator for bootloading an application from a PC host. The CY8CKIT-002 MiniProg3 is used as a USB-I^{2}C bridge to interface the PC to the bootloader, as Figure 14 shows.

Figure 14. Bootloading Using a PC Host

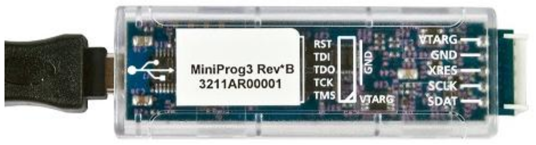


Note: MiniProg3 has internal pull-up resistors on the SDA and SCL lines. Therefore we need not use external pull-ups with MiniProg3.

Follow the steps given below to bootload an application using the bootloader host program. As noted previously, you must program the bootloader project to the PSoC device before starting a bootload operation.

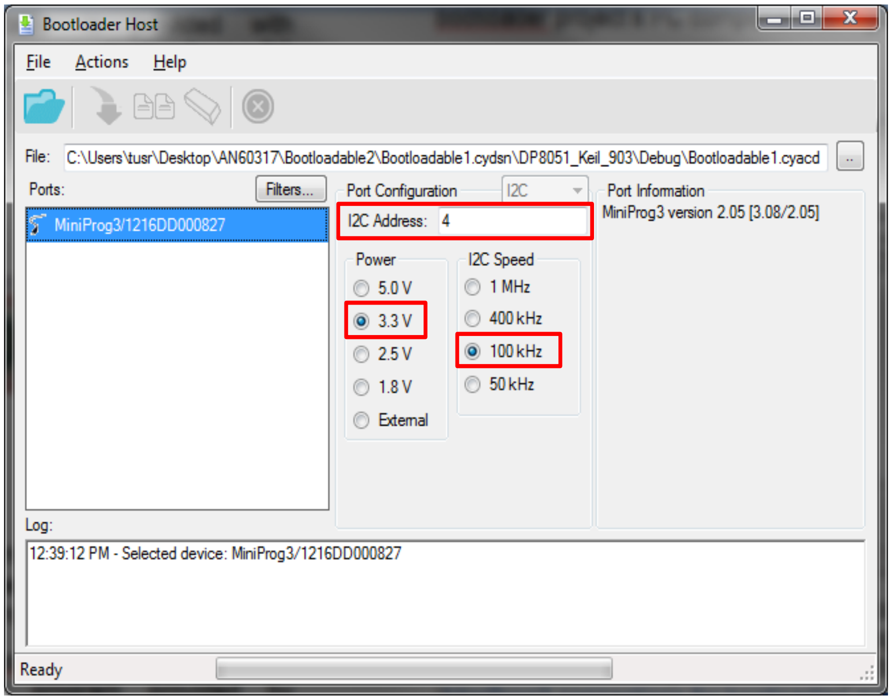
1. To start, connect the PC to the MiniProg3 through the USB cable. Connect the MiniProg3 to the I^{2}C pins (In the example, SDAT is connected to P0[1], SCLK is connected to P0[0], and GND is connected to DVK ground), as Figure 10 and Figure 14 show. Figure 15 shows the pin connections of the Miniprog3, when used as a USB-I^{2}C bridge.

Figure 15. MiniProg3 – I2C Connections



2. Open the Bootloader Host tool by navigating to Tools > Bootloader Host in PSoC Creator.
3. Make sure that the bootloader host application's I^{2}C configuration, shown in Figure 16, is the same as the bootloader project's I2C Component configuration (Figure 7).

Figure 16. Bootloader Host Application



File: C:\Users\tusr\Desktop\AN60317\Bootloadable2\Bootloadable1.cydsn\DP8051_Keil_903\Debug\Bootloadable1.cyacd

Ports: MiniProg3/1216DD000827

Port Configuration: I2C

I2C Address: 4

Power: ( ) 5.0 V [✓ 3.3 V] ( ) 2.5 V ( ) 1.8 V ( ) External

I2C Speed: ( ) 1 MHz ( ) 400 kHz [✓ 100 kHz] ( ) 50 kHz

Port Information: MiniProg3 version 2.05 [3.08/2.05]

Log:

```
12:39:12 PM - Selected device: MiniProg3/1216DD000827
```

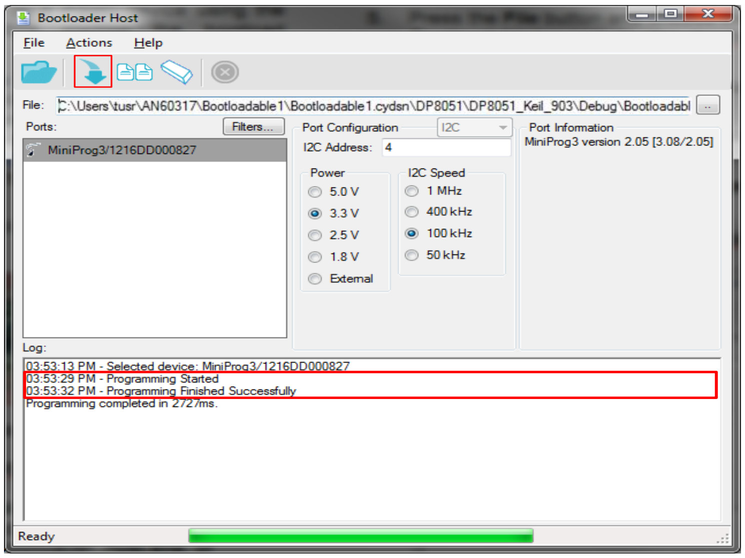
The Power setting determines whether to power the Miniprog3 from the USB host or externally, i.e. from the target PCB. For details, see the knowledge base article MiniProg3 Connections for Bootloading Over I^{2}C.

4. Press the **File** button and choose the bootloadable file *Bootloadable1.cyacd* in the bootloadable project's Debug/Release folder:

   When PSoC 3 is the Bootloader,
   ..\Bootloadable1.cydsn\DP8051\DP8051_Keil_903\ Debug\Bootloadable1.cyacd
   When PSoC 5LP is the Bootloader,
   ..\Bootloadable1.cydsn\CortexM3\ARM_GCC_441\ Debug \Bootloadable1.cyacd

5. To bootload the device, click the **Program** button. You should get a screen similar to Figure 17.

Figure 17. Downloading Bootloadable Project



File: C:\Users\tusr\AN60317\Bootloadable1\Bootloadable1.cydsn\DP8051\DP8051_Keil_903\Debug\Bootloadabl

Ports: MiniProg3/1216DD000827

Port Configuration: I2C

I2C Address: 4

Power: ( ) 5.0 V [✓ 3.3 V] ( ) 2.5 V ( ) 1.8 V ( ) External

I2C Speed: ( ) 1 MHz ( ) 400 kHz [✓ 100 kHz] ( ) 50 kHz

Port Information: MiniProg3 version 2.05 [3.08/2.05]

Log:

```
03:53:13 PM - Selected device: MiniProg3/1216DD000827
03:53:29 PM - Programming Started
03:53:32 PM - Programming Finished Successfully
Programming completed in 2727ms.
```

6. After the bootloadable project is downloaded successfully, a software reset occurs, and the device starts executing the new application. The message "Bootloaded" appears on the LCD.

To bootload another application, reset the device (press and release the kit reset button), and then quickly hit the program button of the bootloader host. For details on bootloader wait time, see **Wait For Command Time** in the Bootloader Datasheet.

2.2.3 Bootloadable Project – Example 2

If you want to bootload a new project (application upgrade) when the current bootloadable project is running, the bootloadable project can invoke the bootloader by calling the function Bootloadable_Load().

In this example the Bootloadable_Load() function is called when a button is pressed. This section describes the steps for creating this bootloadable project:

1. Create a new PSoC Creator project of application type Bootloadable similar to Example 1. Name the project as Bootloadable2. The PSoC devices for this project and the I2C_Bootloader project must be the same.
2. For this project we need the Bootloadable, Pin, Interrupt, and Character LCD Components. Add these Components to your top design schematic, and name them according to Table 3.

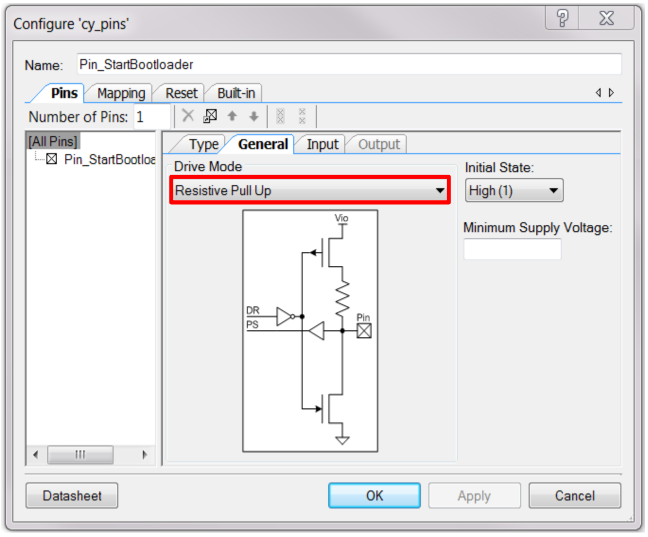
Table 3. Bootloadable Component Names

Component	Name
Bootloadable_1	Bootloadable
Pin_1	Pin_StartBootloader
isr_1	isr_EnterBootloader
LCD_Char_1	LCD_Char

3. Link the Bootloadable Component to the bootloader project as Figure 11 shows.
4. The digital input pin Pin_StartBootloader is used to switch from the application back to the bootloader. When the button connected to this pin is pressed, the application enters the bootloader by calling the API function Bootloadable_Load(). The bootloader waits indefinitely for the host to start the bootload operation.

   When the DVK button is pressed, it shorts to ground, so configure the drive mode of the Pin to be Resistive Pull Up, as Figure 18 shows. Also configure it to generate an interrupt on its falling edge, as Figure 19 shows – the interrupt is generated when the button is pressed (and not when it is released).

Figure 18. Digital Input Pin Configuration


Configure 'cy_pins'

Name: Pin_StartBootloader

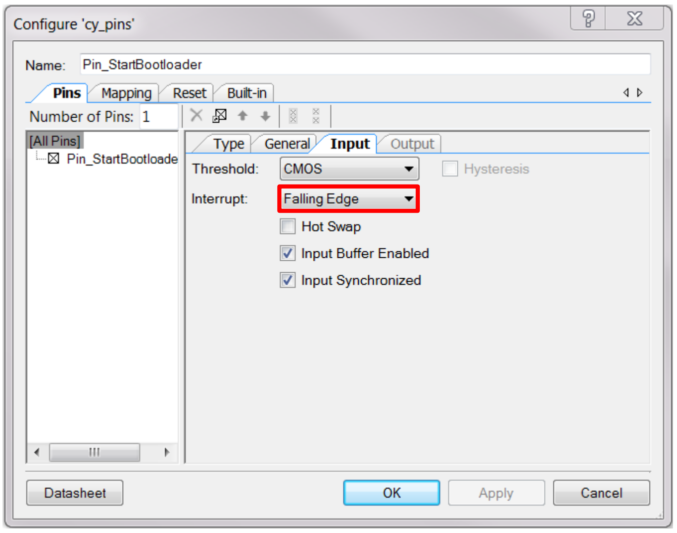
Number of Pins: 1

Drive Mode: Resistive Pull Up

Initial State: High (1)

Minimum Supply Voltage:

Figure 19. Digital Input Pin Configuration



Configure 'cy_pins'

Name: Pin_StartBootloader

Number of Pins: 1

Threshold: CMOS

Interrupt: Falling Edge

Hot Swap: unchecked

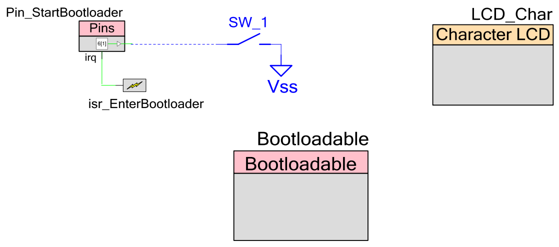
Input Buffer Enabled: checked

Input Synchronized: checked

5. Connect the ISR Component isr_EnterBootloader to the interrupt terminal (irq) of the Pin.

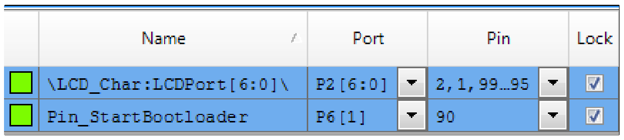
   Now, with the addition of an annotation Component for the button the top design is complete; it should be similar to Figure 20.

Figure 20. Top Design of the Bootloadable2 Project



6. Assign the Pin Components to physical pins. In the Workspace Explorer window, double-click the *Bootloadable2.cydwr* file and assign the pins as Figure 21 shows.

Figure 21. Pin Assignment of Bootloadable2 Project



	Name	Port	Pin	Lock
🟩	\LCD_Char:LCDPort[6:0]\	P2[6:0]	2,1,99..95	✓
🟩	Pin_StartBootloader	P6[1]	90	✓

7. Build the project; this generates the ISR Component API files. Then add code to the interrupt service routine to set the variable 'bootload_flag'. The code is shown below.

```
CY_ISR(isr_EnterBootloader_Interrupt)
{
  /* Place your Interrupt code here. */
  /* `#START
     isr_EnterBootloader_Interrupt` */
  bootload_flag = 1u;
  Pin_StartBootloader_ClearInterrupt();
  /* `#END` */
}
```

Note: 'bootload_flag' is defined in *main.c*, and therefore must be declared as an extern variable in the *isr_EnterBootloader.c* file.

8. A completed Bootloadable2 project is associated with this application note. Insert the code listing from the *main.c* file of this associated project to the *main.c* file of your project.

   The main() function continuously checks the 'bootload_flag' variable. If the variable is set, main() displays the message "Waiting to BL" and calls the Bootloadable_Load() API. This API invokes the Bootloader.
9. Build the project again.
10. Download the bootloadable project *Bootloadable2.cyacd* using the steps described in the section Bootloading Using a PC Host. You can see the message "Bye" and "SW1- P6.1 to BL" displayed on the screen.
11. Press **SW1** (Already connected to P6[1] on the DVK) to enter the bootloader. The message "Waiting to BL" is displayed on the LCD on program execution.
12. Start the bootloader host program and select a different bootloadable file such as *Bootloadable1.cyacd* and press the **Program** button.
13. After the new application is successfully bootloaded, the message on the screen changes to "Bootloaded", as the Bootloadable1 project is supposed to do.

Note: To bootload again you must reset the device and quickly download the new *.cyacd* file. This is because the Bootloadable1 application does not have the Bootloadable_Load() function call to invoke the bootloader, and hence the bootloader can only be invoked on reset.

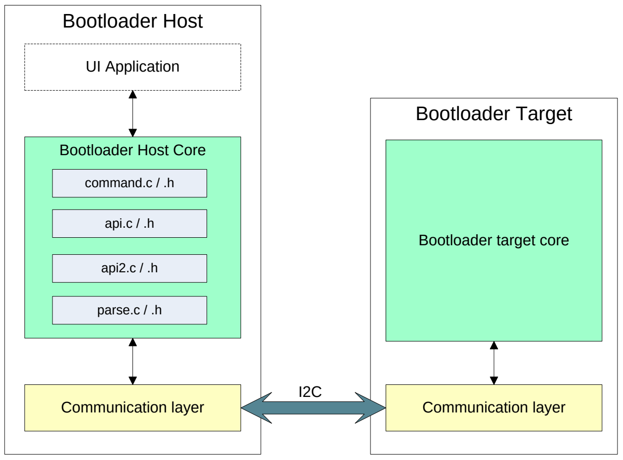
2.3 I²C Bootloader Host

In addition to studying the example projects, it is useful to understand the general structure of a bootloader host program. This can help you to build your own bootloader host system.

2.3.1 Bootloader Host Program

Figure 22 illustrates a protocol level diagram of a bootloader system. The bootloader host and target each have two blocks – the core and the communication layer.

Figure 22. Protocol Level Diagram of Bootloading



The **Core** performs all bootloading operations. The host core sends command packets and flash data to the target. Based on the response from the target, it decides whether to continue bootloading.

The target core decodes the commands from the host, executes them by calling flash routines such as erase row, program row, and verify row, and forms response packets.

The **Communication layer** on both the host and the target provides physical layer support to the bootloading protocol. They contain communication protocol (I²C) specific APIs to perform this function. This layer is responsible for sending and receiving protocol packets between the host and the target.

2.3.2 Bootloader System APIs

All APIs for the target side core and Communication layer are automatically generated by PSoC Creator, when you build a bootloader project.

The host side APIs for the Core are also provided by PSoC Creator, and can be found at:

PSoC Creator \ 3.0 \ PSoC Creator \ cybootloaderutils

For more information on these API files, see Appendix D.

The only code that you need to write is the host side API functions for the communication layer, which are in a file pair *communication_api.c / .h*. There are four functions – OpenConnection(), CloseConnection(), ReadData() and WriteData(). They are pointed to by function pointers within the 'CyBtldr_CommunicationsData' structure, defined in *cybtldr_api.h*.

In this project, we shall use Bootloadable Example Project 2 to generate the *.cyacd* files that we need to invoke the bootloader on a switch press. Generate a *.cyacd* file for this project which writes "Bye" on the LCD, and rename it to *Bootloadable_Bye.cyacd*. Similarly generate a *Bootloadable_Hello.cyacd* by modifying the *main.c* code to write "Hello" on the LCD. These two files are provided with the application note for quick reference.

2.3.3 Steps to Create an I^{2}C Bootloader Host Project

This section shows you how to create an embedded I^{2}C bootloader host project using PSoC 5LP, which can bootload a PSoC 3 device. With this project, the host can bootload two different bootloadable files (*.cyacd* files) on alternate switch presses.

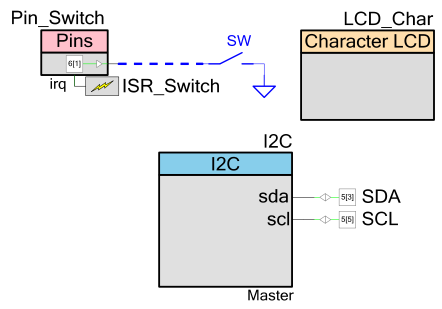
1. Create a new PSoC Creator project as described in Step 1 of I2C Bootloader section.
    a. Select the device to be PSoC 5LP.
    b. Select an empty design schematic.
    c. Name the project as I2C_Bootloader_Host.
2. Since the bootloader project has an I2C Slave, the host project must have an I2C Master. So, add an I2C Master (Fixed Function) Component to the top design schematic. Also, add Pin, Interrupt and Character LCD Components to the top design. Name the Components according to Table 4.

Table 4. Component List for I2C Bootloader Host Project

Component	Name
I2C_1	I2C
SDA_1	SDA
SCL_1	SCL
Pin_1	Pin_Switch
isr_1	ISR_Switch
LCD_Char_1	LCD_Char

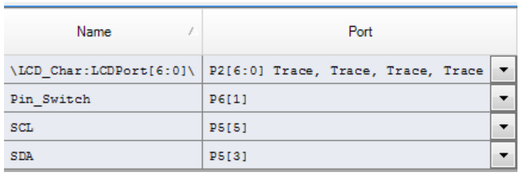
3. To configure the I2C Master Component, double-click on it. By default, the data rate is 100 Kbps. Verify that the slave address in the dialog box is the same as the address in the I2C Slave Component (refer to I2C Bootloader).
4. The digital input pin Pin_Switch is used to initiate the bootloading operation in the host. When the DVK button is pressed, it shorts to ground, so we need to configure this pin to have a resistive pull-up and generate an interrupt on its falling edge.
5. Connect the ISR_Switch Component to the interrupt output (irq) of this pin.
6. Now, with the addition of an annotation Component for the button the top design of this project should be similar to Figure 23.

Figure 23. Top Design of the I2C_Bootloader_Host_PSoC5LP Project



7. Assign the input and output pins. In the Workspace Explorer window, double-click the file *I2C_Bootloader_Host_PSoC5LP.cydwr* and assign the pins as Figure 24 shows.

Figure 24. Pin Assignment for the I2C_Bootloader_Host_PSoC5LP Project



Name	Port
\LCD_Char:LCDPort[6:0]\	P2[6:0] Trace, Trace, Trace, Trace
Pin_Switch	P6[1]
SCL	P5[5]
SDA	P5[3]

8. Build the project to generate the ISR Component API files. Add code to the interrupt service routine to set the variable 'switch_flag'. The code is shown below.

```
CY_ISR(ISR_Switch_Interrupt)
{
  /* Place your Interrupt code here. */
  /* `#START ISR_Switch_Interrupt` */
  switch_flag = 1;
  Pin_Switch_ClearInterrupt() ;
  /* `#END` */
}
```

Note: 'switch_flag' is defined in *main.c*, and therefore must be declared as an extern variable in *ISR_Switch.c*.

9. Add firmware to this project. The I2C_Bootloader_Host_PSoC5LP project is attached to this application note. Insert the code listing from the *main.c* file of this associated project to the *main.c* file of your project.

   The main() function in *main.c* continuously checks the 'switch_flag' variable. When it is set, bootloading is initiated. The file *main.c* has a function called BootloadStringImage(). This function bootloads the *.cyacd* file using the Bootloader Host API files (host core; see Figure 22).

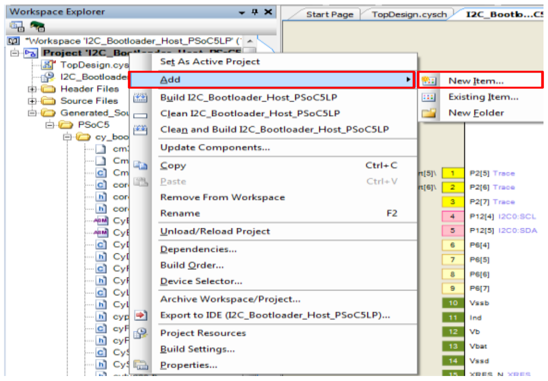
   The main() function has another variable called 'toggle'. It alternates between '0' and '1' on every button press. This makes the host select alternate bootloadable files.

10. As explained previously, a bootloader host core is built upon four API files. These files do all of the host bootloading operations. We must include these files in our project. Find these API files at the following location:

    PSoC Creator \3.0 \ PSoC Creator \ cybootloaderutils

    To include these files, go to the Workspace Explorer window, right-click on the project name, and select Add > New Item, as Figure 25 shows. Add the following files: *cybtldr_api.c / .h*, *cybtldr_command.c / .h*, *cybtldr_parse.c / .h*, and *cybtldr_utils.h*. Update these files by copying from the project attached to this application note.

Figure 25. Adding API Files



11. In addition to the bootloading API files, a host also requires communication layer support. This support is provided by adding the *communication_api.c / .h* files. You may include the contents of these files from the I2C_Bootloader_Host_PSoC5LP project associated with this application note (follow the previous step to add these files to the project). Update these files by copying from the project attached to this application note.
12. Now, include the bootloadable files in the host system. When a bootloadable file is built, a *.cyacd* file is generated; the file is similar to a *.hex* output file. For more information on the .cyacd file, see Appendix B.

    Copy the contents of this file in the form of an array of strings such that each line is an element of the array. Since we have two bootloadable files, we must define two such arrays, named 'StringImage_1' and 'StringImage_2'. For each array, define a macro to store the number of lines in that array. Define these arrays in a separate file named *StringImage.h* (this file must be added to the project before defining the strings).

    Refer to the *StringImage.h* file in the I2C_Bootloader_Host_PSoC5LP project associated with this application note.
13. Build the project and program it into a PSoC 5LP on CY8CKIT - 050.

3 Testing the Projects

Note: In the project I2C_Bootloader_Host_PSoC5LP *main.c* has a macro called TARGET_DEVICE. This macro is used to choose the target device between PSoC 3 and PSoC 5LP. By default, it is defined as 'PSoC_3' (another macro in the same file). If you are using a PSoC 5LP as your target device, change the definition of this macro to 'PSoC_5LP'.

3.1 Kit configuration

To test the projects, configure the kits as follows:

For CY8CKIT–030:

1. Program the PSoC 3 with the I2C_Bootloader project
2. Set jumpers J10 and J11 to 5 V
3. Connect the character LCD to Port 2 [6:0]

For CY8CKIT–050:

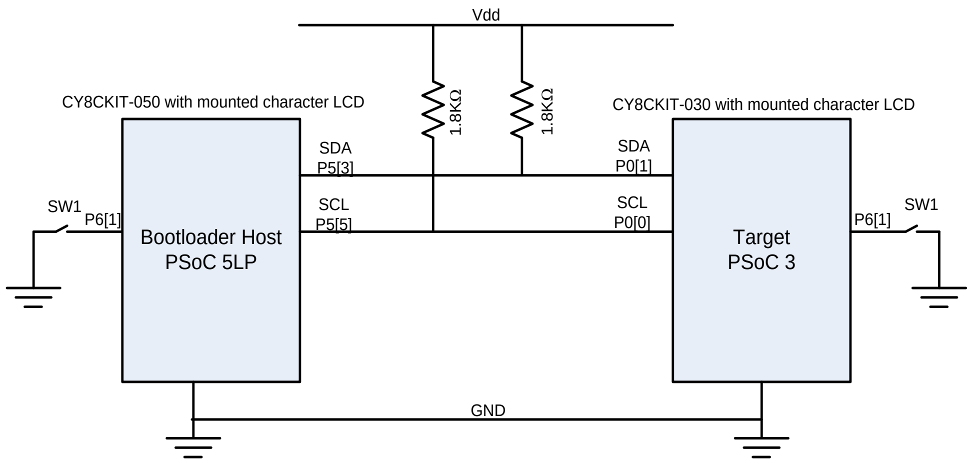
1. Program the PSoC 5LP with the I2C_Bootloader_Host_PSoC5LP project
2. Set jumpers J10 and J11 to 5 V
3. Connect the character LCD to Port 2 [6:0]

Make the following connections between the two DVKs:

1. P0 [0] of CY8CKIT – 030 to P5 [5] of CY8CKIT–050
2. P0 [1] of CY8CKIT – 030 to P5 [3] of CY8CKIT–050
3. Short together the ground pins of the kits.

The connections are illustrated in Figure 26.

Figure 26. Host / Target Connections



3.2 Verifying the Results

After the DVKs are configured, you can test the example projects, as follows:

- On the first button press (P6 [1]) on the CY8CKIT-050, the *Bootloadable_Hello.cyacd* file is bootloaded to the target PSoC 3. On successful completion the message "Bootloaded - Hello" is displayed on the CY8CKIT-050 LCD and the message "Hello" is displayed on the CY8CKIT-030 LCD.
- For subsequent bootloading operations, press the button (P6 [1]) on the CY8CKIT-030. This makes the PSoC 3 enter the bootloader and be ready to bootload a new application.
- On next button press on CY8CKIT-050, the *Bootloadable_Bye.cyacd* file is bootloaded to the target PSoC 3. On successful bootloading the message "Bootloaded - Bye" is displayed on the CY8CKIT-050 LCD and the message "Bye" is displayed on the CY8CKIT-030 LCD.

If you wish to port the Bootloader Host project from PSoC 5LP to PSoC 3, refer to the project I2C_Bootloader_Host_PSoC3 attached to this application note.

Note: PSoC 3 has less flash memory and limited stack space compared to the PSoC 5LP. Due to flash size limitations only smaller *.cyacd* files can be stored in PSoC 3. And due to limited stack space, the linker overlays function variables and arguments in memory. When you use a function pointer, the linker cannot correctly create a call tree for your program. To overcome this compiler issue, some modification is required in the Bootloader Host APIs to make it work with PSoC 3. For this reason, all references in the 'CommunicationData' structure, which contains function pointers, are replaced by direct function calls in the project I2C_Bootloader_Host_PSoC3.

4 Summary

This application note has explained how to bootload PSoC 3 and PSoC 5LP using I^{2}C as the communication interface. It also introduced the basic building blocks of a bootloader host, and showed how to build an embedded I^{2}C bootloader host.

Bootloaders are a standard method for doing field upgrades. With PSoC Creator doing the entire configuration for you, it is easy to make a bootloader for PSoC.

For more advanced information, see the Appendix sections and the PSoC 3 and PSoC 5LP Technical Reference Manuals.

5 Related Application Notes

To better understand bootloaders and flash programming, Refer to the following application notes:

- AN73854 – PSoC 3 and PSoC 5LP Introduction to Bootloaders
- AN73503 – USB HID Bootloader for PSoC 3 and PSoC 5LP

- AN68272 – PSoC 3 and PSoC 5LP Customizing the Bootloader Communication Channel
- AN73054 – PSoC 3 and PSoC 5LP Programming Using an External Microcontroller (HSSP)
- AN61290 – PSoC 3 and PSoC 5LP Hardware Design Considerations
- AN54181 – Getting Started with PSoC 3
- AN77759 – Getting Started with PSoC 5LP

To learn more about the many other features and capabilities of PSoC, click here for a complete list of application notes.

6 Related Projects

The projects attached to this application note are organized as Table 5 shows.

Table 5. Projects Attached to This Application Note

Design Project Name	Description
I2C_Bootloader	This project demonstrates how to create an I^{2}C bootloader project for PSoC 3 and PSoC 5LP, using PSoC Creator
Bootloadable1	This project demonstrates how to create bootloadable projects for PSoC 3 and PSoC 5LP using PSoC Creator. This application prints “Bootloaded” on the LCD.
Bootloadable2	This Project demonstrates how to create bootloadable projects and how to enter the bootloader from a bootloadable project.
I2C_Bootloader_Host_PSoC5LP	This is a sample bootloader host program demonstrating a PSoC 5LP bootloading a PSoC 3.
I2C_Bootloader_Host_PSoC3	This is a similar sample bootloader host program demonstrating a PSoC 3 device bootloading another PSoC 3 device by modifying some of the bootloader APIs.

About the Authors

Name	Anu M D
Title:	Sr. Applications Engineer
Background:	Anu is an applications engineer in Cypress Semiconductor Programmable Systems Division focused on PSoC Applications.
Name:	Tushar Rastogi
Title:	Applications Engineer
Background:	Tushar is an applications engineer in Cypress Semiconductor Programmable Systems Division focused on PSoC Applications.

Appendix A. Memory

A.1 Flash Memory Details

Flash memory provides storage for firmware, bulk data, ECC data, device configuration data, factory configuration data and user defined flash protection data. Figure 27 shows the physical organization of flash memory in PSoC 3 and PSoC 5LP.

PSoC flash is divided into blocks called arrays. Arrays are uniquely identified by array IDs. Each array has 256 rows of flash memory. Each row has 256 data bytes plus, if enabled, 32 ECC (error correction code) bytes. You can use the 32 ECC bytes to store configuration data instead of error correction data. So, an array can have 64 KB or 72 KB for instruction and data storage.

The number of flash arrays depends on the device and the part. PSoC 3 has a maximum flash of 64 KB, so it has only one array and the only valid array ID is 0. PSoC 5LP has a maximum of 256 KB of flash, or 4 flash arrays, with valid array IDs 0 to 3.

Flash memory is programmed one row at a time. It can be erased in 64 row sectors or the entire flash can be erased at once. Rows are identified by a unique combination of the array ID and the row number.

Figure 27 also shows that the first X rows of flash are occupied by the bootloader. X is set such that there is enough space for:

- The vector table for the bootloader, starting at address 0 (PSoC 5LP only)
- The bootloader project configuration bytes
- The bootloader project code and data
- The checksum for the bootloader portion of the flash

For PSoC 5LP, the vector table contains the initial stack pointer (SP) value for the bootloader project, and the address of the start of the bootloader project code. It also contains vectors for the exceptions and interrupts to be used by the bootloader. In PSoC 3, the interrupt vectors are not in flash. They are supplied by the interrupt controller.

The bootloadable project occupies the flash starting at the first 256-byte boundary after the bootloader. This region of the flash includes:

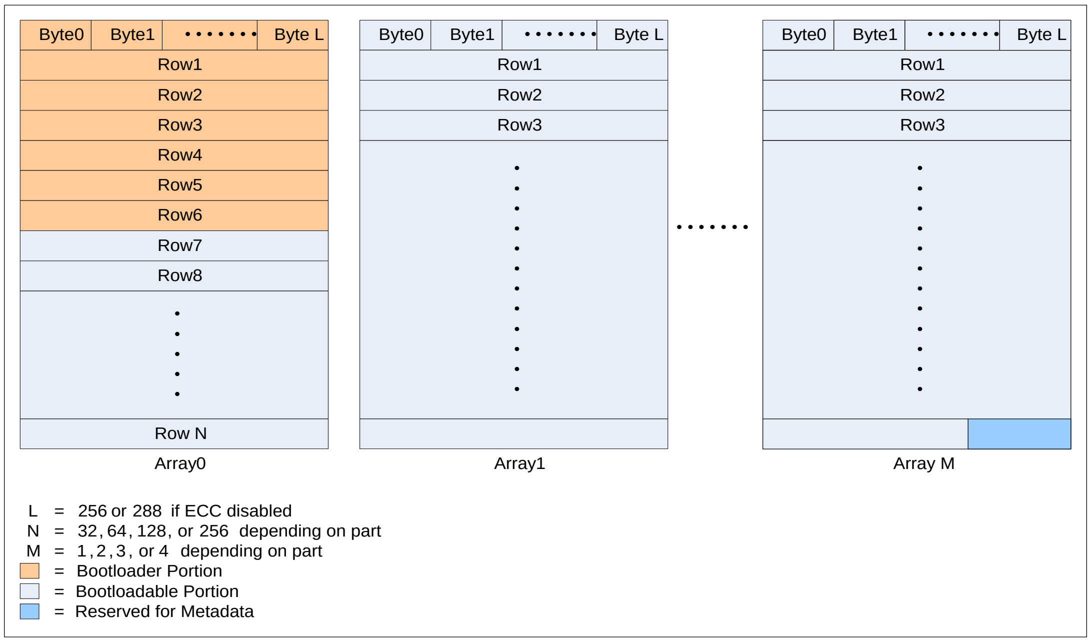
- Vector table for the bootloadable project (PSoC 5LP only)
- The bootloadable project code and data

The highest 64-byte block of flash is used as a common area for both projects. Parameters saved in this block include:

- The entry in flash of the bootloadable project (4-byte address)
- The amount of flash occupied by the bootloadable project (number of flash rows)
- The checksum for the bootloadable portion of flash (one byte)
- The size in bytes of the bootloadable portion of flash (Four bytes).

For more information on the location of metadata in the flash memory, refer Metadata Layout in Flash.

Figure 27. Physical Organization of Flash Memory in PSoC

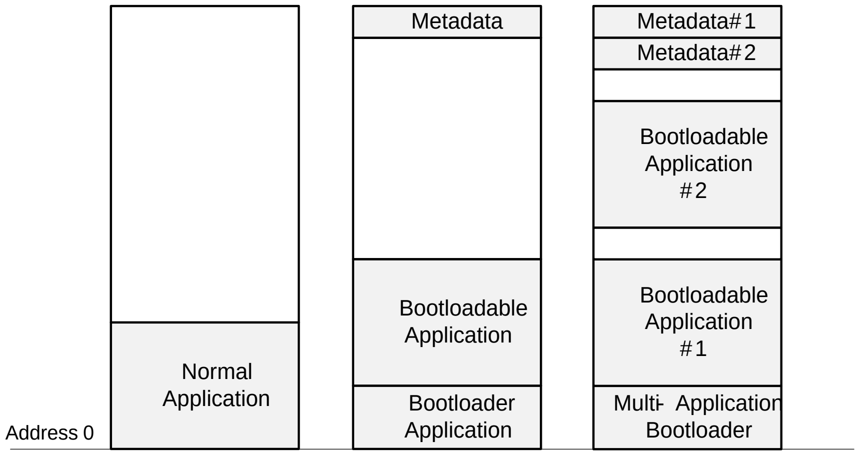


A.2 Memory Usage in PSoC

There are two types of bootloader project types, standard bootloader and dual-application bootloader. Dual-application bootloaders are also called multi-application bootloaders. They are useful for designs that require a guarantee that there is always a valid application that can be run. But this guarantee comes with a limitation that each application has only one half of the flash available.

Figure 28 shows the flash memory usage for each type of PSoC Creator project.

Figure 28. Flash Memory Usage



A.3 Metadata Layout in Flash

The metadata section is the highest 64-byte block of flash, and is used as a common area for both bootloader and bootloadable projects, as Figure 28 shows. Various parameters, depending upon the device used, are stored in this block, as Table 6 shows. For the multi-application bootloader, there are two sets of metadata. The metadata of application 1 occupies the highest 64-byte block of flash and the metadata of application 2 occupies 64 bytes from the subsequent flash row.

Table 6. Metadata Layout

Address	PSoC 3	PSoC 5LP
0x00	App Checksum	App Checksum
0x01	Reserved	Application Address
0x02		
0x03	Application Address	
0x04		
0x05	Reserved	Last Bootloader Row
0x06		
0x07	Last Bootloader Row	Reserved
0x08		
0x09	Reserved	Application Length
0x0A		
0x0B	Bootloadable Application Length	
0x0C		
0x0D	Reserved	Reserved
0x0E		
0x0F		
0x10	Active Bootloadable Application	Active Bootloadable Application
0x11	Bootloadable Application Verification Status	Bootloadable Application Verification Status
0x12	Bootloader Application Version	Bootloader Application Version
0x13		
0x14	Bootloadable Application ID	Bootloadable Application ID
0x15		
0x16	Bootloadable Application Version	Bootloadable Application Version
0x17		
0x18	Bootloadable Application Custom ID	Bootloadable Application Custom ID
0x19		
0x1A		
0x1B		
0x20-0x3F	NA	NA

Note: For the multi-application bootloader, Last Bootloader Row for metadata (image 2) signifies the last row of bootloadable 1in the flash section and not the bootloader row.

A.4 Flash Protection

If the bootloader code is invalid, it makes the product unusable. So it is important to protect the bootloader portion of the flash from accidental overwrites.

All PSoC 3 and PSoC 5LP devices include a flexible flash protection system. This feature is designed to prevent duplication and reverse engineering of proprietary code. But it can also be used to protect against inadvertent writes to the bootloader portion of flash.

Four protection levels are provided for flash memory, as Table 7 shows. Each row of flash can be configured to have a different protection level, which can be set using PSoC Creator (the Flash Security tab of the *.cydwr* file).

Table 7. Levels of Flash Protection

Protection level	Allowed	Not allowed
Unprotected	External read and write; Internal read and write	-
Factory upgrade	External write; Internal read and write	External read
Field upgrade	Internal read and write	External read and write
Full protection	Internal read	External read and write; Internal write

Once the bootloader portion of the flash is configured to have a protection level of Full protection, it cannot be changed in the field. The only way to alter the protection level or to change the bootloader code is to completely erase the flash and reprogram it using the JTAG / SWD interface.

An example for protecting bootloader flash follows:

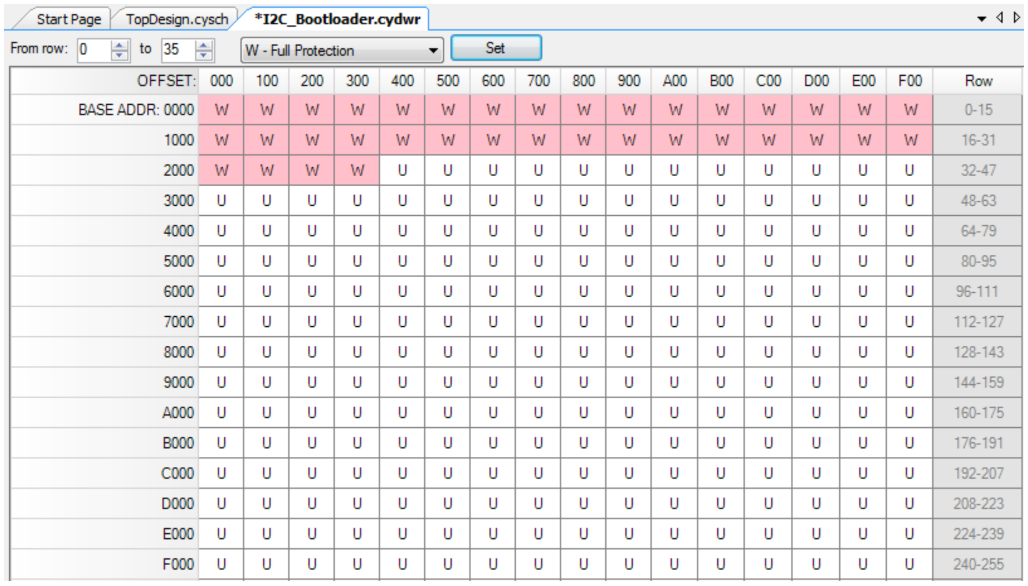
A.5 Example for Flash Protection

When the bootloader project is built, the PSoC Creator Output window shows the amount of flash used. For example, if the flash occupied by the I2C_Bootloader project is 8886 bytes, then the output is (for PSoC 3 with 64 KB flash):

```
Flash used: 8886 of 65536 bytes (13.6 %).
```

The bootloader thus occupies 35 rows of flash (8886 / 256), i.e., flash locations 0x0000 to 0x2300. Set the flash protection level as Full protection for these rows (under the Flash Security tab of the *.cydwr* file in PSoC Creator). The protection level for the remaining rows can be Unprotected (the default) or Field upgrade, as Figure 29 shows.

Figure 29: Flash Protection in PSoC Creator



From row: 0 to 35 W - Full Protection

OFFSET:	000	100	200	300	400	500	600	700	800	900	A00	B00	C00	D00	E00	F00	Row
BASE ADDR: 0000	W	W	W	W	W	W	W	W	W	W	W	W	W	W	W	W	0-15
1000	W	W	W	W	W	W	W	W	W	W	W	W	W	W	W	W	16-31
2000	W	W	W	W	U	U	U	U	U	U	U	U	U	U	U	U	32-47
3000	U	U	U	U	U	U	U	U	U	U	U	U	U	U	U	U	48-63
4000	U	U	U	U	U	U	U	U	U	U	U	U	U	U	U	U	64-79
5000	U	U	U	U	U	U	U	U	U	U	U	U	U	U	U	U	80-95
6000	U	U	U	U	U	U	U	U	U	U	U	U	U	U	U	U	96-111
7000	U	U	U	U	U	U	U	U	U	U	U	U	U	U	U	U	112-127
8000	U	U	U	U	U	U	U	U	U	U	U	U	U	U	U	U	128-143
9000	U	U	U	U	U	U	U	U	U	U	U	U	U	U	U	U	144-159
A000	U	U	U	U	U	U	U	U	U	U	U	U	U	U	U	U	160-175
B000	U	U	U	U	U	U	U	U	U	U	U	U	U	U	U	U	176-191
C000	U	U	U	U	U	U	U	U	U	U	U	U	U	U	U	U	192-207
D000	U	U	U	U	U	U	U	U	U	U	U	U	U	U	U	U	208-223
E000	U	U	U	U	U	U	U	U	U	U	U	U	U	U	U	U	224-239
F000	U	U	U	U	U	U	U	U	U	U	U	U	U	U	U	U	240-255

A.6 Nonvolatile Latch (NVL) Settings

NVLs can be configured in a bootloader project or any other normal PSoC Creator project, but not in the bootloadable projects. This is because the NVL settings are always loaded on device bootup. Upon device bootup, the bootloader project executes first followed by the bootloadable code. Therefore, a bootloadable project's NVL settings are those of the bootloader project with which it is associated.

Some of the PSoC Creator Design wide resource (*.cydwr*) settings are programmed using user NVLs. You will get a warning or error message if some of the *.cydwr* settings for bootloadable project are different from bootloader project's settings.

Appendix B. Project Files

B.1 Bootloadable Output Files

When any PSoC Creator project is built, an output file of type *.hex* is generated. This is the file that is downloaded to the PSoC while programming using the JTAG / SWD interface.

For a bootloadable project, this file is a combined *.hex* file of both the bootloadable and the related bootloader project. This file is typically used to download both projects via JTAG / SWD in a production environment.

B.2 *.cyacd File Format

When a bootloadable project is built, an additional file of type *.cyacd* (application code and data) is also generated. This file contains a header followed by lines of flash data. Excluding the header, each line in the file represents an entire row of flash data. The data is stored as ASCII data in big endian format. Hence, while bootloading, the contents of this file must be parsed (converted from ASCII to hex). Parsing is not required for programming a file of type *.hex*.

The header of this file has the following format:

```
[4 bytes Silicon ID] [1 byte Silicon rev]
[1 byte checksum type]
```

The flash lines have the following format:

```
[1 byte array ID] [2 bytes row number]
[2 bytes data length] [N bytes of data]
[1 byte checksum]
```

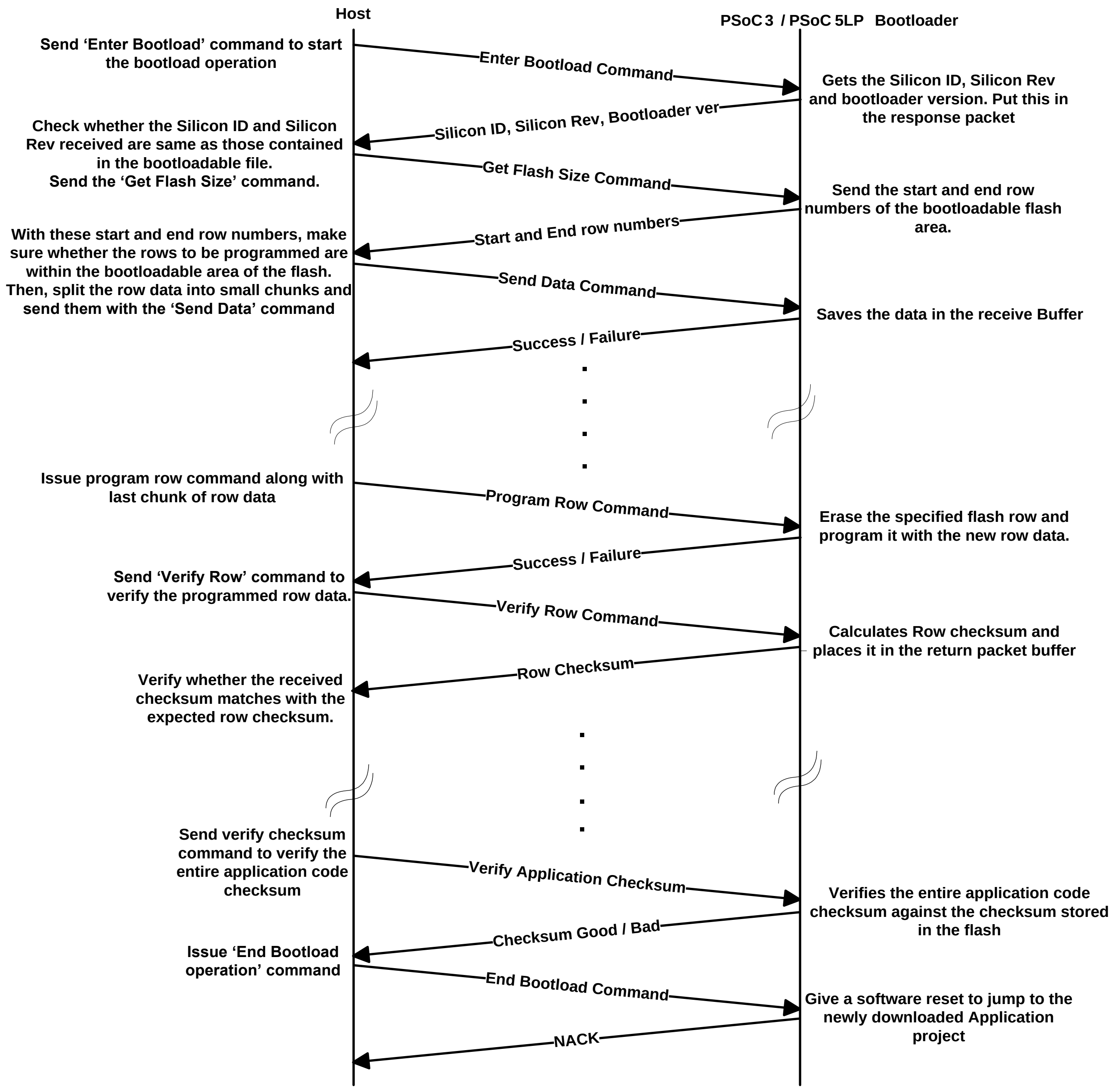
The checksum type in the header indicates the type of checksum used in the packets sent between the bootloader and the bootloader host during the bootloading operation. If this byte is 0, the checksum is a basic summation. If it is 1, the checksum is CRC-16.

Appendix C. Host / Target Communications

C.1 Communication Flow

In Bootloader Function Flow, we looked at the operation of a bootloader in PSoC 3 and PSoC 5LP, and SPI Bootloader Host introduced the building blocks of a bootloader host. With this background, Figure 30 explains the flow of communication between the host and the target during a bootloading operation. This gives the order in which commands are issued to the target and responses are received. See Command and Status / Error Codes for a complete list of bootload commands, their codes and their expected responses.

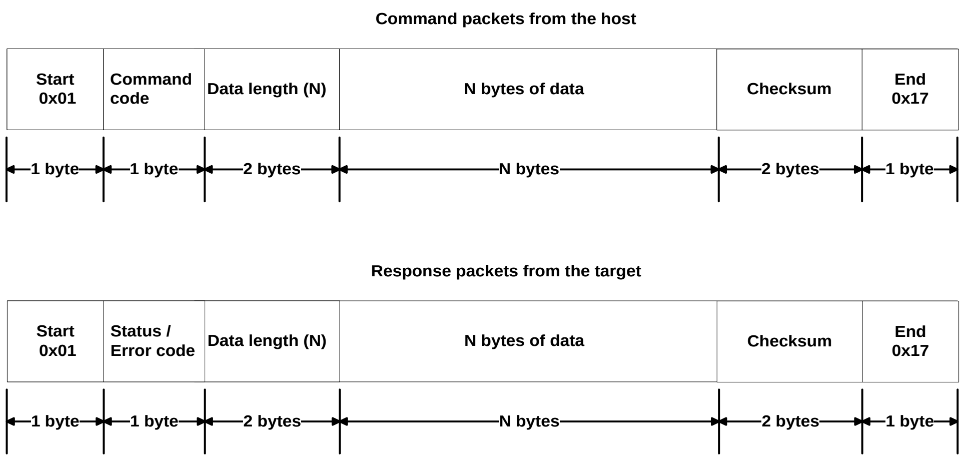
Figure 30. Communication Flow During Bootloading


Host
PSoC 3 / PSoC 5LP Bootloader
Send 'Enter Bootload' command to start the bootload operation
Enter Bootload Command
Gets the Silicon ID, Silicon Rev and bootloader version. Put this in the response packet
Silicon ID, Silicon Rev, Bootloader ver
Check whether the Silicon ID and Silicon Rev received are same as those contained in the bootloadable file.
Send the 'Get Flash Size' command.
Get Flash Size Command
Send the start and end row numbers of the bootloadable flash area.
Start and End row numbers
With these start and end row numbers, make sure whether the rows to be programmed are within the bootloadable area of the flash.
Then, split the row data into small chunks and send them with the 'Send Data' command
Send Data Command
Saves the data in the receive Buffer
Success / Failure
Issue program row command along with last chunk of row data
Program Row Command
Erase the specified flash row and program it with the new row data.
Success / Failure
Send 'Verify Row' command to verify the programmed row data.
Verify Row Command
Calculates Row checksum and places it in the return packet buffer
Row Checksum
Verify whether the received checksum matches with the expected row checksum.
Send verify checksum command to verify the entire application code checksum
Verify Application Checksum
Verifies the entire application code checksum against the checksum stored in the flash
Checksum Good / Bad
Issue 'End Bootload operation' command
End Bootload Command
Give a software reset to jump to the newly downloaded Application project
NACK

C.2 Protocol Packet Format

The bootloading operation involves exchange of command and response packets between the host and the target. These packets have specific formats, as Figure 31 shows.

Figure 31. Bootloading Packet Format



Each packet includes checksum bytes. The checksum is calculated for N + 4 bytes in a packet from Start byte to the N bytes of data and excludes the End byte. The checksum can be a basic summation (2's complement) or CRC-16 depending on the bootloader project setting. When sending multi byte data such as DataLength and Checksum, least significant byte is sent first.

The bootloader responds to each command from the host with a response packet. The format of the response packet is similar to the command packet except that there will be status/error code instead of command code. The important commands and data bytes and the bootloader response packet data are given in Table 8 on page 27.

C.3 I²C Packet Information for Bootloader Host

The I²C packet is a lower level of communication between the two devices. The packet format for reading and writing PSoC from host is given in Figure 32 and Figure 33.

Figure 32. Communication Packet Sent from Host to Bootloader Through I²C Interface (Write Packet)

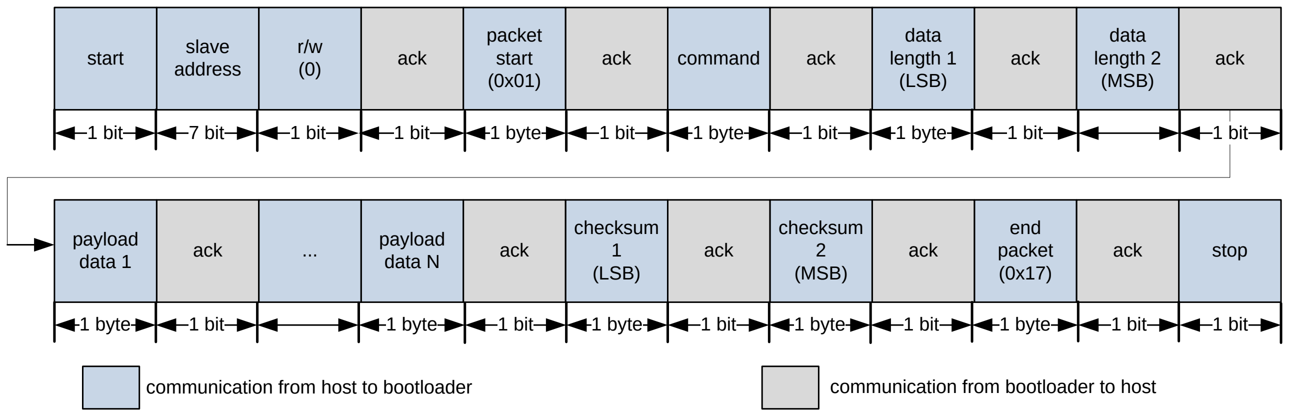
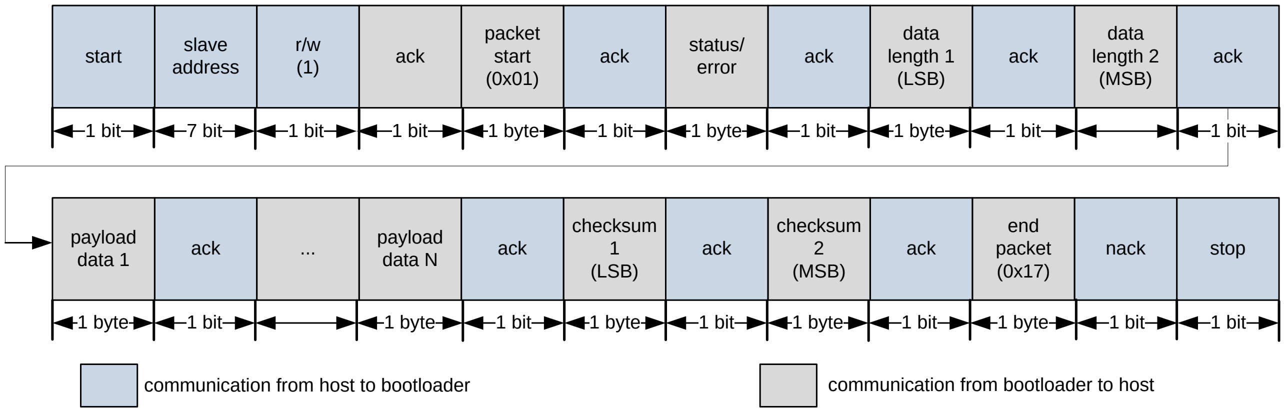


Figure 33. Communication Packet Received From Bootloader Through I^{2}C Interface (Read Packet)



C.4 Command and Status / Error Codes

As the previous section explains, the command and response packet structures are similar. The only difference is that the second byte contains a command code or a status / error code.

Table 8 provides a list of commands and their expected responses.

Table 9 on page 28 provides a list of status and error codes.

Table 8. Bootloading Commands

Command Byte	Command	Data Byte in the Command Packet	Expected Response Data Bytes
0x31	Verify checksum	N/A	1 byte: Non zero or '0'. If it is a non-zero byte, then the application checksum matches and it is a valid application. If it is a zero byte, then the checksum is bad and the application is invalid.
0x32	Get flash size	Flash array ID, 1 byte	First row number of the bootloadable flash, 2 bytes; Last row number of the bootloadable flash, 2 bytes. The bootloader responds to this command with the first full row after the bootloader application (first row of the bootloadable application) and last flash row in the selected flash array.
0x33	Get application status (valid only for dual application bootloader)	Application number, 1 byte	Valid application status, 1 byte; Active application status, 1 byte. Checks whether the specified application is valid and it is active. The application is valid and active if the return byte is zero.
0x34	Erase row	Flash array ID, 1 byte; Flash row number, 2 bytes	N/A. Erases the contents of the specified flash row.
0x35	Sync bootloader	N/A	N/A. Resets the bootloader to a clean state. Any data which was buffered in will be thrown out. This command is needed only if the bootloader and the host go out of sync with each other.
0x36	Set active application (valid only for dual application bootloader)	Application number, 1 byte	N/A. Sets the specified application as active.
0x37	Send data	N bytes of data to be sent	N/A. The received data bytes will be buffered by the bootloader in anticipation of the Program row command.

Command Byte	Command	Data Byte in the Command Packet	Expected Response Data Bytes

0x38	Enter bootloader	N/A	Silicon ID, 4 bytes;
			Silicon Rev, 1 byte;
			Bootloader version, 3 bytes;
			All the commands are ignored until this command is received.

0x39	Program row	Flash array ID, 1 byte;
		Flash row number, 2 bytes;
		N bytes of data to be sent
			N/A.
			After sending multiple bytes of data to the bootloader using Send data command, the last chunk of data is sent along with this command.

0x3A	Verify row	Flash array ID, 1 byte; Flash row number, 2 bytes	Row checksum, 1 byte. This is calculated as the 2's complement of sum of all flash row data. Returns the checksum of the specified row.
0x3B	Exit bootloader	N/A	N/A. This command is not acknowledged.

Table 9. Bootloading Status / Error Codes

Status / error codes	Label	Description
0x00	CYRET_SUCCES	The command was successfully received and executed.
0x01	CYRET_ERR_FILE	File is not accessible.
0x02	CYRET_ERR_EOF	Reached the end of the file.
0x03	CYRET_ERR_LENGTH	The number of data bytes received is not in the expected range.
0x04	CYRET_ERR_DATA	The data is not of the proper form.
0x05	CYRET_ERR_CMD	The command is not recognized.
0x06	CYRET_ERR_DEVICE	The expected device does not match the detected device.
0x07	CYRET_ERR_VERSION	The bootloader version detected is not supported.
0x08	CYRET_ERR_CHECKSUM	The checksum does not match the expected value.
0x09	CYRET_ERR_ARRAY	The flash array is not valid.
0x0A	CYRET_ERR_ROW	The flash row is not valid.
0x0B	CYRET_BTLDR	The bootloader is not ready to process data.
0x0C	CYRET_ERR_APP	The application is not valid and cannot be set as active (valid only for dual application bootloader).
0x0D	CYRET_ERR_ACTIVE	The application is currently marked as active (valid only for dual application bootloader).
0x0F	CYRET_ERR_UNK	An unknown error occurred.
0xFF	CYRET_ABORT	The operation was aborted.

Appendix D. Host Core APIs

cybtldr_api2.c / .h

This is a higher level API that handles the entire bootload operation. It has functions to open and close files. It invokes the functions of the *cybtldr_api.c / .h* API for the bootload operations. This API can be used when building a GUI based bootloader host.

cybtldr_parse.c / .h

This module handles the parsing of the *.cyacd* file that contains the bootloadable image to send to the device. It also has functions for setting up access to the file, reading the header, reading the row data, and closing the file.

cybtldr_api.c / .h

This is a row-level API file for sending a single row of data at a time to the bootloader target. It has functions for setting up the bootload operation, erasing a row, programming a row, verifying a row and ending the bootload operation. Table 10 describes in detail the functions of this API file.

Table 10. Functions of *cybtldr_api.c /.h*

Function	Description
CyBtldr_StartBootloadOperation	Enables the communication interface and sends an Enter Bootloader command to the target. From the response packet received, verifies the silicon ID, silicon revision of the target device, and bootloader version.
CyBtldr_ProgramRow	First validates a row, i.e., sends a Get Flash Size command to the target for a particular array ID of the target flash. In response to this, the target returns the start and end row numbers of the bootloadable flash portion in that array. The host reads this response and checks whether the specified row is in the bootloadable area of the flash. If row validation is a success, the host breaks the row data into smaller pieces and sends them to the target using Send Data commands. Along with the last portion of row data, sends a Program Row command to the target.
CyBtldr_VerifyRow	This function also first validates a row for a particular array ID and row number. If row validation is successful, sends a Verify Row command for the validated flash row. In response to this command, the target returns the checksum of the row. The returned checksum is verified against the expected checksum value.
CyBtldr_EraseRow	This function also first validates a row for a particular array ID and row number. If row validation is successful, sends an Erase Row command for the validated flash row.
CyBtldr_EndBootloadOperation	Sends an Exit Bootload command and disables the communication interface.

cybtldr_command.c / .h

This API handles the construction of command packets to the target and parsing the response packets received from the target. The cybtldr_api.c / .h invokes the functions of this API. For example, to send an Enter Bootload command, CyBtldr_StartBootloadOperation() calls the CyBtldr_CreateEnterBootloadCmd() function of this API. It also has a function for calculating the checksum of the command packets before sending to the target.

Appendix E. Bootloader and Device Reset

As noted elsewhere in this application note, transferring control from the bootloader to the bootloadable, or vice versa, is always done through a device reset. This may be a consideration if your system must continue to perform mission-critical functions while changing from one program to the other. This section details why reset must be used, as well as its implications for device performance in your application.

E.1 Why is Device Reset Needed?

To understand why device reset is needed, it is important to note that the bootloader and bootloadable projects in your system are each completely self-contained PSoC Creator projects. Each project has its own device configuration settings. Thus, when you change from one project to the other, you can completely redefine the hardware functions of the PSoC device.

To implement complex custom functions, device configuration can involve the setting of thousands of PSoC registers. This is especially true for PSoC's digital and analog routing features. When you configure the registers and routing, you must make sure that, in addition to setting the bits for the new configuration, you reset the bits for the old configuration. Otherwise, the new configuration may not work, and may even damage the device.

So when changing between bootloader and bootloadable projects, we do a device software reset (SRES). This causes all PSoC registers to be reset to their default states. Configuration for the new project can then begin. Note that by assuming that all PSoC registers are initialized to their device reset default states, we can reduce both configuration time and flash memory usage.

E.2 Effect on Device I/O Pins

As described in application notes AN61290, PSoC® 3, PSoC 5LP Hardware Design Considerations, and AN60616, PSoC Startup Process, during the reset and startup process the PSoC I/O pins are in three distinct drive modes, as Table 11 shows.

Table 11. PSoC I/O Pin Drive Modes During Device Reset

Startup Event	I/O Pin Drive Mode	Duration (Typical)		Comment
		Slow IMO (12 MHz)	Fast IMO (48 MHz)	
Device reset (SRES) active Device reset removed	HI-Z Analog	40 µs		While reset is active, the I/Os are held in the HI-Z Analog mode.
Nonvolatile Latches (NVLs) copied to I/O ports Code starts executing	NVL setting: HI-Z Analog, Pull-up, or Pull-down	~12 ms	~4 ms	Duration depends on code execution speed and configuration complexity.
I/O ports and pins are configured	PSoC Creator project configuration	n/a		8 possible drive modes. See device datasheet for details.
Code reaches main()	Code may change I/O pin function	n/a		

For details on NVL usage in PSoC, see a device datasheet. In your PSoC Creator project, the NVL settings are established in two places:

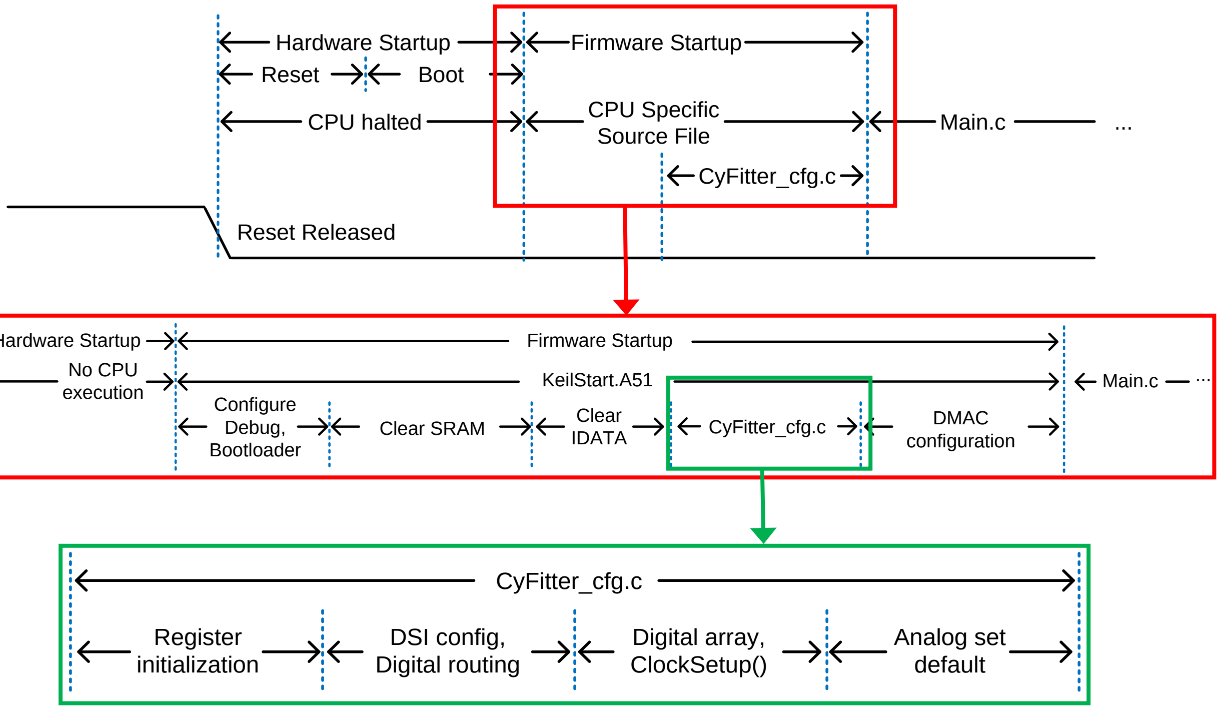
- The **Reset** tab for I/O ports, the individual Pin Component configurations
- The **System** tab for all other NVLs, the design-wide resources (DWR) window

The NVLs are updated when the device is programmed with your project. Note that a bootloadable project cannot set NVLs; its DWR settings must match those in the associated bootloader project.

Final I/O drive modes are set by individual Pin Component configurations.

Figure 34 shows the timing diagrams for device startup and configuration. The example in the middle diagram is for PSoC 3; similar processes exist for PSoC 4 and PSoC 5LP. For more information, see AN60616, PSoC Startup Process.

Figure 34. Device Startup Process Diagrams



E.3 Effect on Other Functions

At device reset, UDB registers are reset, so all UDB-based Components cease to exist and their functions are stopped. The same is also true for analog Components based on the configurable SC/CT blocks in PSoC 3 and PSoC 5LP.

All fixed peripherals – digital and analog – are reset to their idle states. This includes the DMA, DFB, timers (TCPWM), I²C, USB, CAN, ADCs, DACs, comparators, and opamps. All clocks are stopped except the IMO.

All digital and analog routing control registers are reset. This causes all digital and analog switches to be opened, breaking all connections within the device. This includes all connections to the I/Os except the NVLs.

All hardware-based functions are restored after configuration (see Figure 34). All firmware functions are restored when the project's main() function starts executing.

E.4 Example: Fan Control

Let us examine how a bootloader and its associated device reset can be integrated into a typical application such as fan control. PSoC Creator provides a Fan Controller Component, which encapsulates all necessary hardware blocks including PWMs, tachometer input capture timer, control registers, status registers, and a DMA channel or interrupt. For more information, see the Fan Controller Application page.

The fan control application is in a bootloadable project. Optionally, the bootloader may be customized to keep the fan running while bootloading.

The fan can also be kept running while the device is reset, during the transfer between the bootloader to the bootloadable, as Table 12 shows.

Table 12. PSoC I/O Pin Drive Modes During Device Reset for Fan Controller

I/O Pin Drive Mode	Comment
HI-Z Analog	Optionally add external pull-up or pull-down resistor to the PWM pin, for 100% duty cycle. This may not be needed because the fan may keep spinning due to inertia.
NVL setting: HI-Z Analog, Pull-up, or Pull-down	Optionally set the PWM Pin Component reset value to Pull-up or Pull-down, for 100% duty cycle. This may not be needed because the fan may keep spinning due to inertia.
PSoC Creator project configuration	Set the PWM Pin Component drive mode and initial state, for 100% duty cycle. The PWM Component becomes active but does not run.
Main() starts executing	When PWM_Start() is called, the PWM starts driving the PWM pin at the Component's default duty cycle. Firmware can read the tachometer data and start actively controlling the duty cycle.

Appendix F. Miscellaneous Topics

Bootloader versus HSSP

The bootloader allows your system firmware to be upgraded over a communication interface. But for a complete flash upgrade, including the bootloader flash area, you must use the JTAG / SWD programmer (Host Sourced Serial Programming). The in-system serial programming (ISSP) specifications to create HSSP are given in AN62391 (PSoC 3) and AN64359 (PSoC 5LP).

What happens if power fails during the bootload operation?

If power fails during the bootload operation, then at the next reset the checksum of the bootloadable project does not match the expected value (the bootloadable project's checksum stored in the last row of flash) and the bootloadable project is considered to be invalid. Program execution remains in the bootloader until a successful bootload happens. The bootloader host must send a start bootload command to re-start the bootload operations.

Why do we need a reset to jump between the bootloader and the bootloadable projects?

PSoC 3 and PSoC 5LP are enormously configurable devices. The bootloader allows you to change on-chip hardware resources as well as firmware. Due to its highly configurable architecture, hardware reconfiguration (placement, routing, functional) is possible only from a reset state. Therefore the bootloader requires a reset to jump between the bootloader and bootloadable projects. See Bootloader and Device Reset for more information.

Multi-Application Bootloader

Multi-Application Bootloader (MABL) is used to put two bootloadable applications in flash simultaneously. The two applications can be the same to ensure that there is always a valid application in the device's flash. Or, the two applications can be different so that they can be switched using bootloader commands. This functionality comes with the obvious limitation that each application has one half of the available flash memory. Figure 28 on page 21 shows the project placement in flash memory for a MABL.

MABL can be implemented by following these steps which are different from standard bootloader application:

1. Check the Dual application bootloader checkbox in Bootloader configuration window
2. Add two bootloadable projects to the workspace, say, Project_A and Project_B. For each project, add a dependency to the MABL project. Two *.cyacd* files are generated for each project:
   - *Project_A_1.cyacd* and *Project_A_2.cyacd*
   - *Project_B_1.cyacd* and *Project_B_2.cyacd*
3. The *.cyacd* file with suffix 1 always occupies the first half of flash and *.cyacd* file with suffix 2 occupies the second half. Because of this only certain combinations of *.cyacd* file can be used. These combinations are:
   - *Project_A_1.cyacd* and *Project_A_2.cyacd*
   - *Project_B_1.cyacd* and *Project_B_2.cyacd*
   - *Project_A_1.cyacd* and *Project_B_2.cyacd*
   - *Project_B_1.cyacd* and *Project_A_2.cyacd*
4. Program the device with the multi-application bootloader project and bootload the applications (*.cyacd* files) sequentially, in one of the above combinations, using the Bootloader Host application.
5. To switch between applications, follow these steps:
   - Open Bridge Control Panel by navigating Start > All Programs > Cypress > Bridge Control Panel
   - Connect the PC host to the device through a CY3240-I2USB SB-I²C Bridge or a MiniProg3.
   - Send command 0x38 to enter the bootloader:
     ```
     w 04 01 38 00 00 C7 FF 17
     r 04  x  x  x  x  x x x
     ```
   - To switch from application_1 to application_2, send the set_active_application command (0x36):
     ```
     w 04 01 36 01 00 01 C7 FF 17
     r 04  x  x   x  x  x  x  x
     ```
   - To switch from application_2 to application_1, send the set_active_application command (0x36):

```
w 04 01 36 01 00 00 C8 FF 17
r  04   x   x    x   x   x   x   x
```

- Send the exit_bootloader command (0x3B) to launch the application:

```
w 04 01 3B 00 00 C4 FF 17
r  04   x   x   x   x   x  x  x
```

These commands are explained in Table 13, using the example of the set_sctive_application command (application_1 to application_2):

Table 13. Command Bytes

Bytes	1	1	1	2	N	2	1
Value	04	01	36	00 01	01	C7FF	17
Description	slave address	packet start byte	command	no. of bytes to follow (LSB first)	data bytes	checksum	packet end byte

Memory Requirement for Bootloader

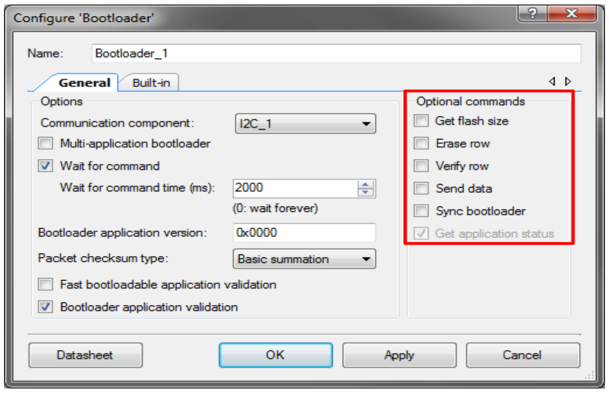
A typical I²C bootloader project with all the optional commands included occupies approximately 7 KB of PSoC 3 flash with Keil 8051 compiler optimization level 5.

It occupies approximately 5.4 KB of PSoC 5LP flash with GCC compiler optimization set to "size". You can find the memory used by the bootloader project in the output window, when you build the project. RAM memory used by the bootloader project can be reused by the bootloadable project.

The memory usage of a bootloader project can be reduced a small amount by removing the optional commands supported by the Bootloader Component, as Figure 35 shows.

Set the Device Configuration Mode to Compressed in the *.cydwr* > System tab, as Figure 36 shows, to minimize flash memory usage. Device Configuration Mode to DMA if startup time is more important than code size.

Figure 35. Unchecking Optional Commands in Bootloader Component

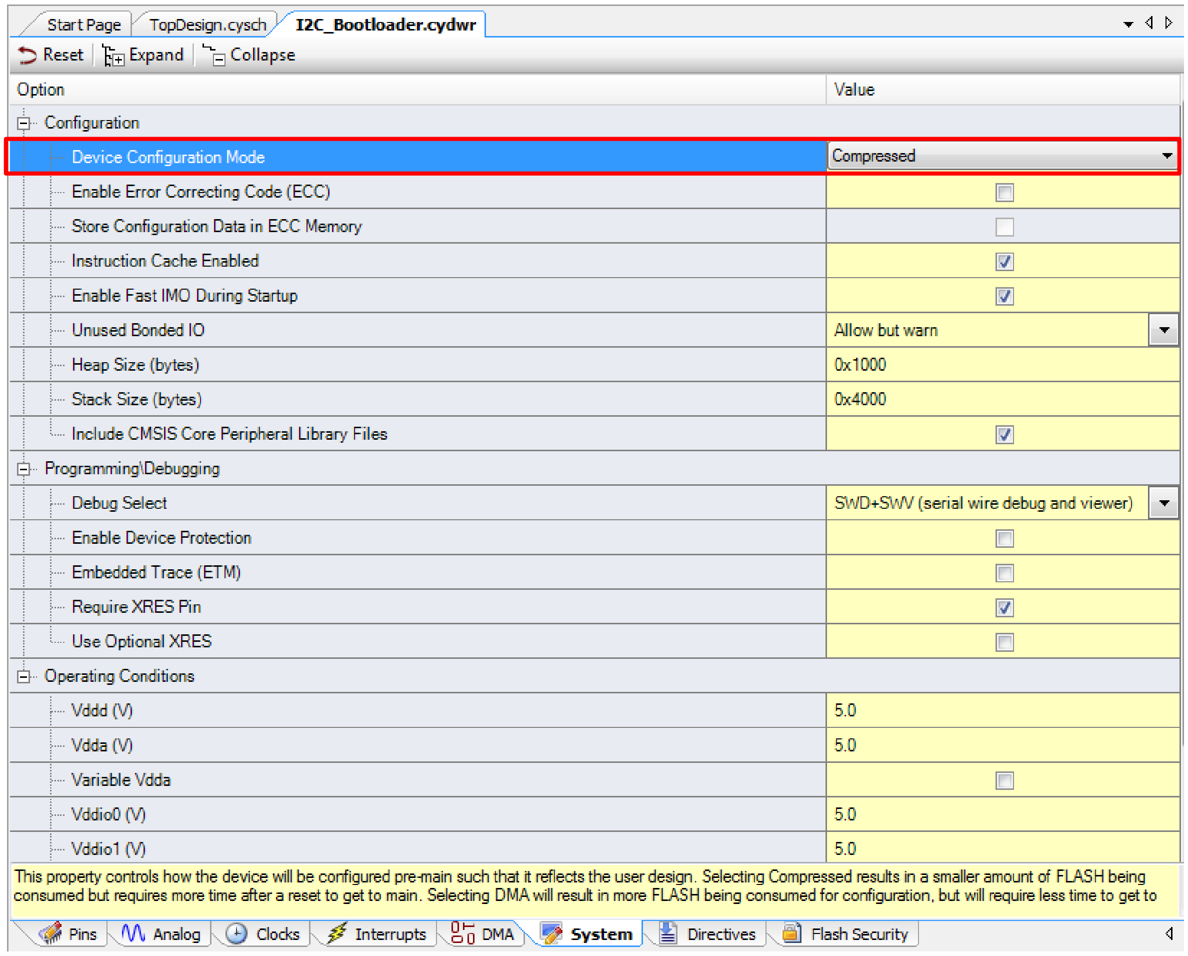


Name: Bootloader_1

Communication component: I2C_1
Multi-application bootloader [ ]
Wait for command [✓]
Wait for command time (ms): 2000
(0: wait forever)
Bootloader application version: 0x0000
Packet checksum type: Basic summation
Fast bootloadable application validation [ ]
Bootloader application validation [✓]

Optional commands
Get flash size [ ]
Erase row [ ]
Verify row [ ]
Send data [ ]
Sync bootloader [ ]
Get application status [✓]

Figure 36. Device Configuration Mode



Start Page | TopDesign.cysch | I2C_Bootloader.cydwr

Reset | Expand | Collapse

Option	Value
Configuration	
Device Configuration Mode	Compressed
Enable Error Correcting Code (ECC)	☐
Store Configuration Data in ECC Memory	☐
Instruction Cache Enabled	☑
Enable Fast IMO During Startup	☑
Unused Bonded IO	Allow but warn
Heap Size (bytes)	0x1000
Stack Size (bytes)	0x4000
Include CMSIS Core Peripheral Library Files	☑
Programming\Debugging	
Debug Select	SWD+SWV (serial wire debug and viewer)
Enable Device Protection	☐
Embedded Trace (ETM)	☐
Require XRES Pin	☑
Use Optional XRES	☐
Operating Conditions	
Vddd (V)	5.0
Vdda (V)	5.0
Variable Vdda	☐
Vddio0 (V)	5.0
Vddio1 (V)	5.0

This property controls how the device will be configured pre-main such that it reflects the user design. Selecting Compressed results in a smaller amount of FLASH being consumed but requires more time after a reset to get to main. Selecting DMA will result in more FLASH being consumed for configuration, but will require less time to get to

Document History

Document Title: AN60317 - PSoC® 3 and PSoC 5LP I^{2}C Bootloader

Document Number: 001-60317

Revision	ECN	Orig. of Change	Submission Date	Description of Change
**	2896214	ANMD	03/19/2010	New application note.
*A	3011572	ANMD	08/19/2010	Added level 5 Optimization in Memory Organization section.
*B	3119712	ANMD	12/23/2010	Added section Converting a Normal Application Project to Bootloadable Project
*C	3308212	ANMD	07/27/2011	Added details of bootloader communication packets and how to build your own bootloader host.
*D	3452308	ANMD	12/01/2011	Updated Bootloader in PSoC Creator. Updated Steps to Create an I2C Bootloader Project. Updated Bootloadable Project – Example 1. Updated Bootloading Using PC Host. Updated Bootloadable Project – Example 2. Updated Bootloadable Project Output Files. Updated PSoC 3 or PSoC 5 Bootloader Protocol Packets. Updated Example Project. Updated in new template.
*E	3569438	ANMD	04/02/2012	Added Contact information as per the template. Updated Table 1 and Table 2. Minor text edits.
*F	3672940	NIDH/ANMD	07/13/2012	Updated application note and example project for PSoC Creator 2.1.
*G	3819268	ANCY	11/22/2012	Updated for PSoC 5LP
*H	3940871	TUSR	03/07/2013	Added appendix on Multi Application Bootloader, Metadata Layout, I2C Packet format Revised the bootloader host project for PSoC 5LP Added new project I2C_Bootloader_Host_PSoC3 Updated and revised content for clarity and uniformity with other bootloader application notes
*I	3958453	MKEA	04/08/2013	Corrected errors in hyperlinks. Corrected a kit reference.
*J	4339544	RNJT	04/10/2014	Updated for PSoC Creator 3.0 SP1
*K	4435010	MKEA	07/17/2014	Added Bootloader and Device Reset
*L	4605702	RNJT	12/23/2014	Added a note that bootloader flash protection settings take precedence and bootloadable settings are ignored.
*M	5774557	VKVK	06/15/2017	Updated template Updated application note and example project for PSoC Creator 4.0

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

All other trademarks or registered trademarks referenced herein are the property of their respective owners.

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