AN84401

PSoC® 3 and PSoC 5LP SPI Bootloader

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

AN84401 describes an SPI-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 SPI-based bootloader and bootloadable projects. It also shows how to build an SPI-based embedded bootloader host program.

Contents

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^2C, 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:

- The basic building blocks and functionality of a bootloader host system
- How to add an SPI bootloader to a PSoC Creator project for PSoC 3 or PSoC 5LP

This application note also shows you how to create your own embedded SPI bootloader host, which is based on a 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.

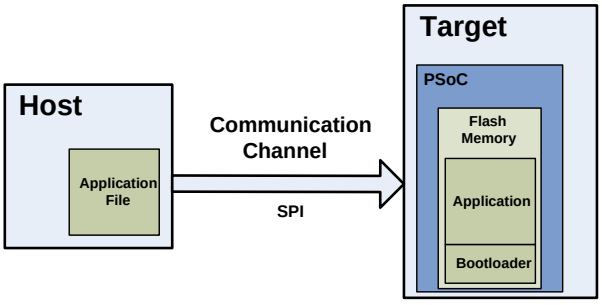
This application note also assumes that you are familiar with bootloader concepts. If you are new to these concepts, see AN73854, PSoC 3 and PSoC 5LP: Introduction to Bootloaders. For a complete list of other application notes on bootloading, see Related Application Notes.

Finally, this application note assumes that you are familiar with the SPI protocol and the PSoC Creator SPI Master and Slave Components. If you are new to these Components, see the PSoC Creator Component datasheets SPI Master and SPI Slave. You can also get these datasheets by right-clicking on the SPI Components in PSoC Creator.

Terms and Definitions

Figure 1 illustrates the main elements in a bootloadable 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. The portion of the embedded firmware that knows how to update 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 (such as PSoC 5LP) 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).

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 *.cyacd 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.

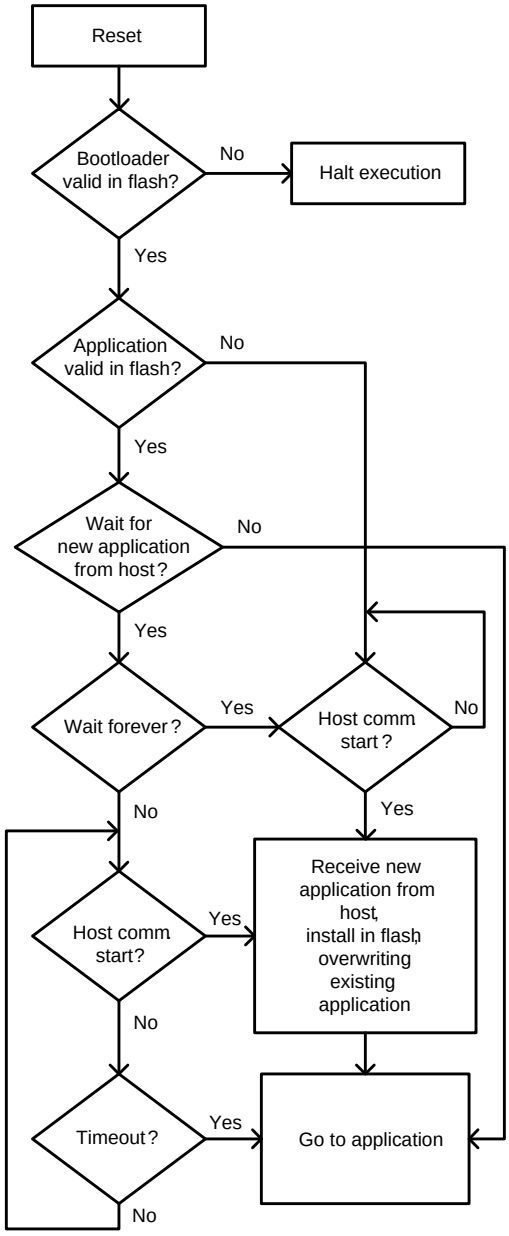
Bootloader Function Flow

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 shows typical bootloader functions.

Figure 2: Bootloader Function Flow



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 to start bootloading is to pass control from the application or bootloadable back to the bootloader.

Bootloadable API

The Bootloadable Component in PSoC Creator has an application programming interface (API) 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?

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.

Projects

Now, let us look at some specific examples of how PSoC bootloaders are developed.

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.

SPI Bootloader

In this section, we create and build an SPI-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 SPI_Bootloader as Figure 5 shows.

Figure 3: Selecting Device for Bootloader Project

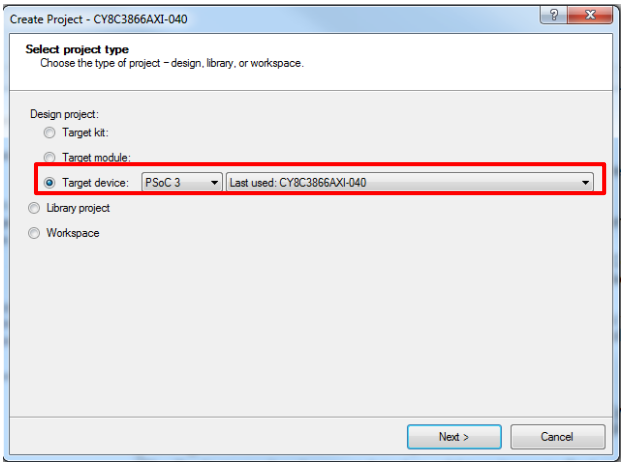


Figure 4: Selecting Empty Schematic for SPI_Bootloader Project

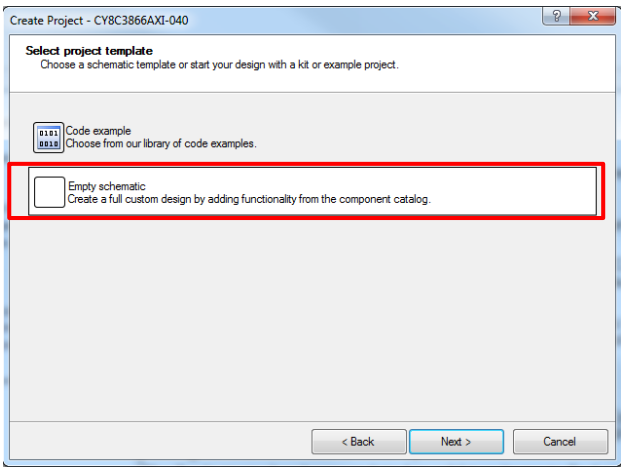
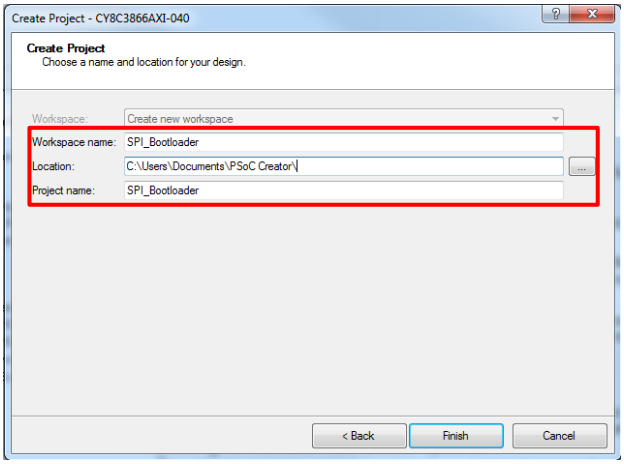
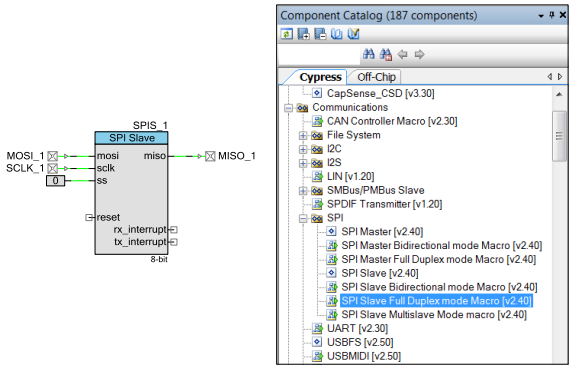


Figure 5 Creating SPI_Bootloader Project



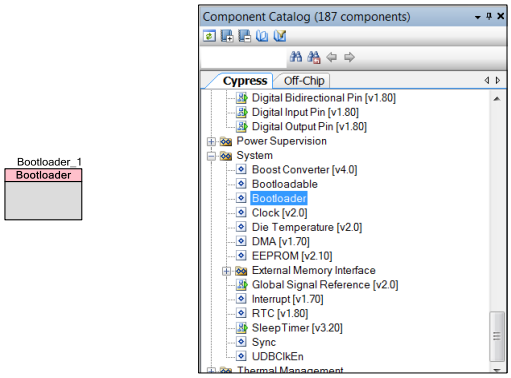
2. Add an SPI Slave Full Duplex Mode Macro to the top design schematic (you can use an SPI Slave Component instead of the macro; the macro automatically includes pins with the Component), as Figure 6 shows.

Figure 6: SPI Slave Full Duplex Mode Macro Component



3. Add a Bootloader Component to the top design schematic, as Figure 7 shows.

Figure 7: Bootloader Component



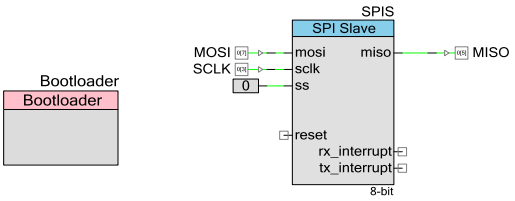
4. Rename the Components and pins as Table 1 shows.

Table 1: Bootloader Project Component Names

Component	Name
Bootloader_1	Bootloader
SPIS_1	SPIS
MOSI_1	MOSI
MISO_1	MISO
SCLK_1	SCLK

5. Now, the top design of the project looks similar to Figure 8.

Figure 8: Top Design of SPI_Bootloader Project

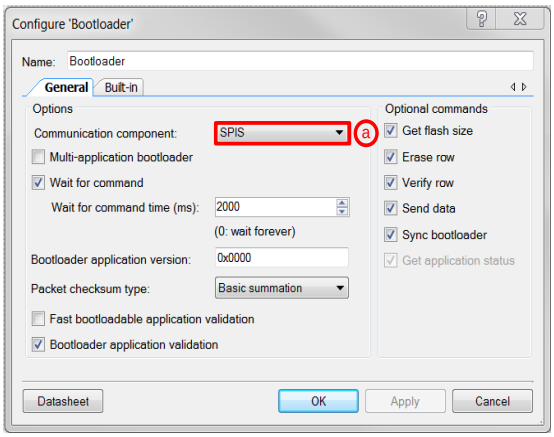


Since there is only one slave with which the host communicates, the Slave Select (SS) line of the SPI Slave is connected to logic low.

The next step is to configure these Components.

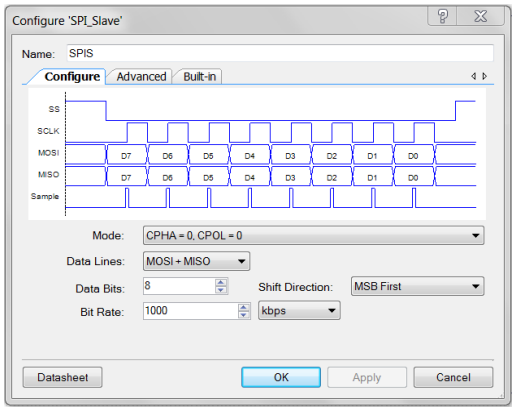
6. To configure the Bootloader, double-click on the Component.
   a) Select SPIS as the Communication component, as Figure 9 shows. Leave the other parameters at their default settings. For more information on these configuration parameters, refer to the Bootloader Component datasheet.

Figure 9: Bootloader Configuration



7. To configure the SPI Slave Component, double-click on it. By default it is in mode 0 with a data rate of 1000 Kbps. Keep these default settings; you can perform bootloading in any SPI mode and at any supported data rate. Figure 10 shows the basic configuration tab of the SPI Slave Component.

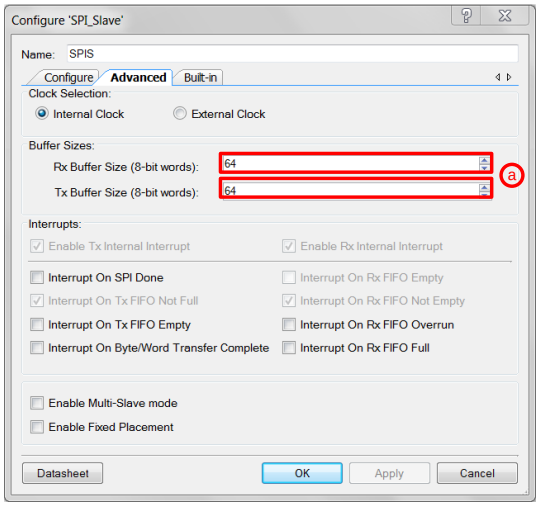
Figure 10: Basic SPI Slave Configuration



8. Click the **Advanced** tab of the SPIS configuration window.

   a) Set both the receive (Rx) and transmit (Tx) buffer sizes to 64, to avoid communication overflow (the host packet size is as much as 64 bytes). Leave the other parameters at their default settings. See Figure 11.

Figure 11: Advanced SPI Slave Configuration



9. Assign the Pin Components to physical pins. In the Workspace Explorer window, double-click the *SPI_Bootloader.cydwr* file, and click the **Pins** tab. Assign the Pins as Figure 12 shows.

Figure 12: Pin Assignment for SPI_Bootloader Project

	Name	Port	Pin	Lock
	MISO	P0[5]	77	☑
	MOSI	P0[7]	79	☑
	SCLK	P0[3]	74	☑

10. Review the *main.c* file – the CyBtldr_Start() function is added automatically when you create a bootloader project. Starting with PSoC Creator 3.2 and the Bootloader Component v1.40, a call to Bootloader_Start() should 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, any code that is placed after this API call is never executed.
11. Build the project and program it into a PSoC 3 device on CY8CKIT-030. If your target device is a PSoC 5LP, change the device before building the project, and program it to a CY8CKIT-050.

You have now created a simple SPI-based bootloader. It can communicate with a host and download and install into flash a new application, or bootloadable project. The bootloader can be expanded and customized in a number of ways. See the Bootloader and SPI Component datasheets, and AN73854 for details.

Note The bootloader occupies a portion of the PSoC flash, reducing the amount of flash available for the application. See Appendix E for details.

Let us now look at how to create bootloadable applications that can be used with this bootloader.

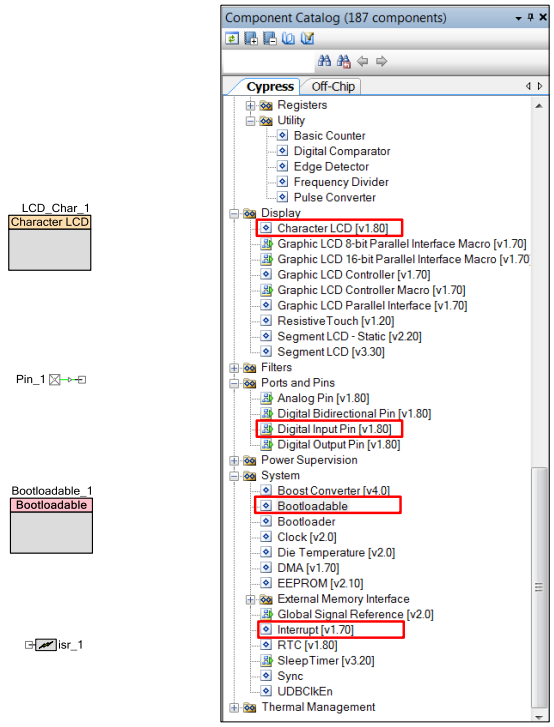
Bootloadables

We shall now create two bootloadable projects. They are very similar – one displays "Hello" on a character LCD and the other displays "Bye". This section describes the steps for creating these bootloadable projects.

1. Create a new PSoC Creator project as described in step 1 of SPI Bootloader.
   a) Name the project as Bootloadable_1.
   b) The devices for this project and the SPI_Bootloader project must be the same.

   For this project we need the Bootloadable, Digital Input Pin, Interrupt, and LCD Components. Add these Components to your top design schematic, as Figure 13 shows.

   Figure 13: Required Components for the Bootloadable_1 Project



2. Rename the Components according to Table 2.

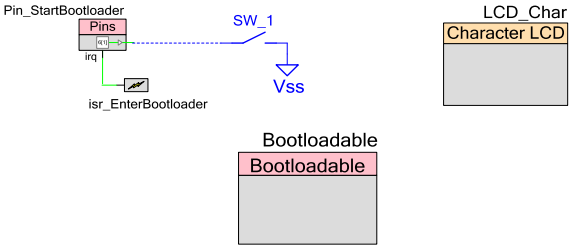
   Table 2: Bootloadable Project Component Names

Component	Name
Bootloadable_1	Bootloadable
Pin_1	Pin_StartBootloader
isr_1	isr_EnterBootloader
LCD_Char_1	LCD_Char

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

   With the addition of an annotation Component for the button, the top design is complete; it should be similar to Figure 14.

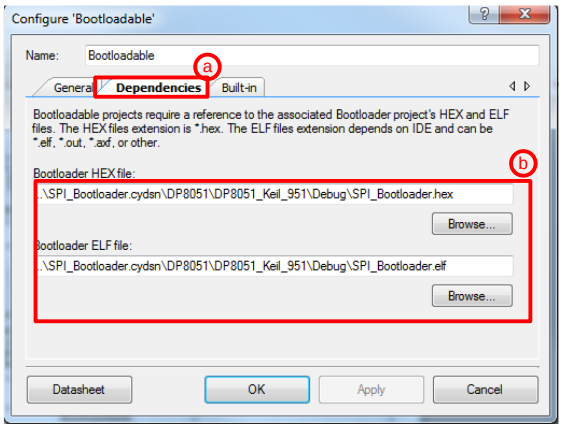
   Figure 14: Top Design of the Bootloadable_1 Project



   The next step is to configure these Components.

4. To configure the Bootloadable Component, double-click on it.
   a) A bootloadable project is always linked to the *.hex* file of a bootloader project. To do this, go to the dependencies tab of the Bootloadable component configuration window as Figure 15 shows.
   b) Link the bootloadable to the *SPI_Bootloader.hex* file, as Figure 15 shows. For more information on Bootloader Component configuration, refer to the Bootloader Component datasheet.

   Figure 15: Bootloadable Component Configuration



   You may find the *SPI_Bootloader.hex* file in the Bootloader project's Debug / Release folder:

   When PSoC 3 is the Bootloader,

   ..\SPI_Bootloader\SPI_Bootloader.cydsn\DP8051\DP8051_Keil_903\Debug\SPI_Bootloader.hex

   When PSoC 5LP is the Bootloader,

   ..\SPI_Bootloader\SPI_Bootloader.cydsn\CortexM3\ARM_GCC_441\Debug\SPI_Bootloader.hex

5. 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.
   a) 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 16 shows.
   b) Also configure it to generate an interrupt on its falling edge, as Figure 17 shows – the interrupt is generated when the button is pressed (and not when it is released).

Figure 16: Digital Input Pin Configuration

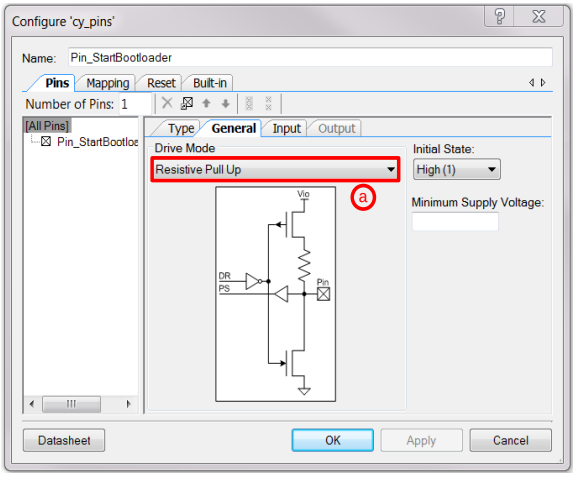
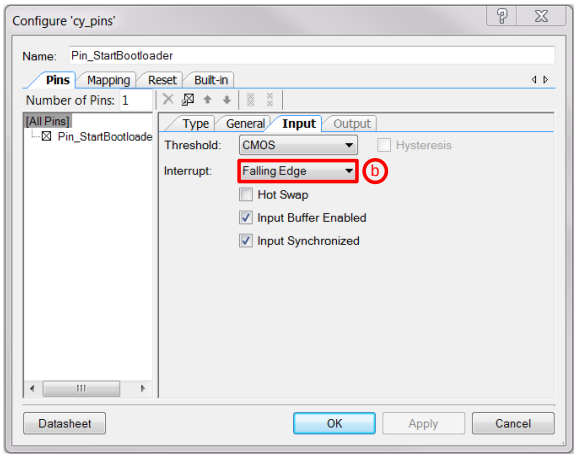


Figure 17: Digital Input Pin Configuration



6. Assign the Pin Components to physical pins. In the Workspace Explorer window, double-click the *Bootloadable_1.cydwr* file and assign the pins as Figure 18 shows.

Figure 18: Pin assignment of Bootloadable_1 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 file (isr_EnterBootloader.c). 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. The declaration is shown below.

```
#include "Pin_StartBootloader.h"
extern uint8 bootload_flag;
```

8. A completed Bootloadable_1 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. When a bootloadable project is built, PSoC Creator generates a *.cyacd* file. This is the file that is bootloaded on to the target. For more information on this file and its contents, see Appendix B.

10. To create the other bootloadable project that displays "Bye", repeat steps 1 to 9 in this section. Name the project Bootlodable_2. The only difference between the two projects is that the code in *main.c* displays "Bye" instead of "Hello".

11. Build the second project. This generates the *Bootloadable_2.cyacd* file.

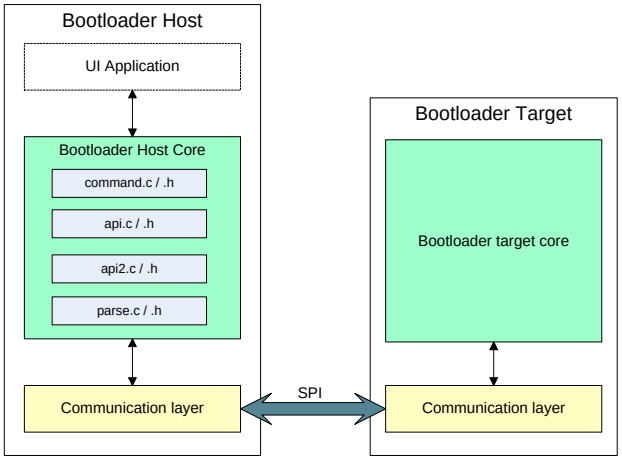
SPI Bootloader Host

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.

Bootloader Host Program

Figure 19 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 19. 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 (SPI) specific APIs to perform this function. This layer is responsible for sending and receiving protocol packets between the host and the target.

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*.

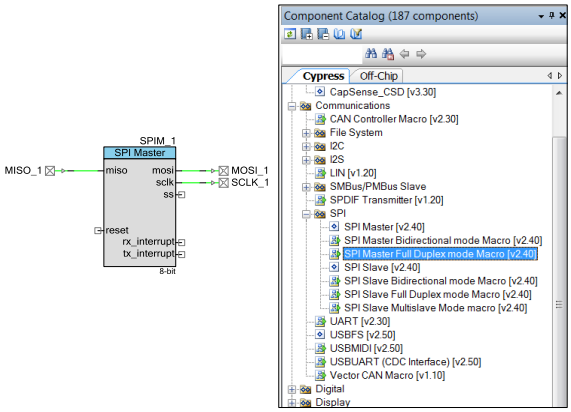
Steps to Create a SPI Bootloader Host Project

This section shows you how to create an embedded SPI 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 SPI Bootloader.
   a) Select the device to be PSoC 5LP
   b) Select an empty design schematic
   c) Name the project as SPI_Bootloader_Host

2. Since the bootloader project has a SPI Slave, the host project must have a SPI Master. So, add a SPI Master Full Duplex Mode Macro Component to the top design schematic as Figure 20 shows. Also, add Digital Input Pin, Interrupt, and Character LCD Components to the top design.

   Figure 20: SPI Master Full Duplex Mode Macro Component



3. Rename the Components according to Table 3.

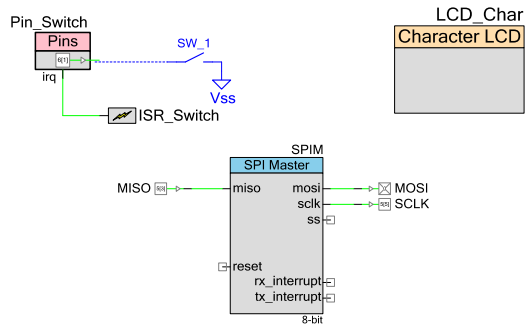
   Table 3: Component List for SPI Bootloader Host Project

Component	Name
SPIM_1	SPIM
MOSI_1	MOSI
MISO_1	MISO
SCLK_1	SCLK
Pin_1	Pin_Switch
isr_1	ISR_Switch
LCD_Char_1	LCD_Char

4. Connect the ISR_Switch Component to the interrupt output (irq) of this pin.

   With the addition of an annotation Component for the button, the top design of this project should be similar to Figure 21.

   Figure 21: Top Design of the SPI_Bootloader_Host Project



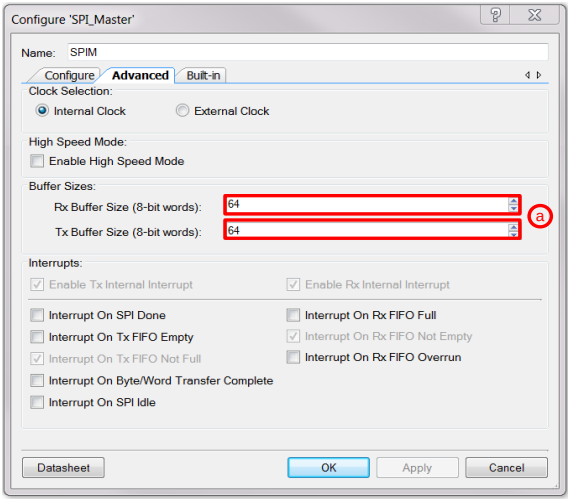
   The next step is to configure these Components.

5. To configure the SPI Master Component, double-click on it. By default, the mode of operation is mode 0 and the data rate is 1000 Kbps. Keep these default settings.

   Note The project can run with any SPI mode and at any supported data rate, but the SPI Slave in the bootloader project and the SPI Master in this project must have the same mode of operation and data rate.

   a) In the **Advanced** tab of the Component configuration window, set the transmit (Tx) and receive (Rx) buffer sizes to 64, to avoid any communication overflow (the host packet is as much as 64 bytes). This is illustrated in Figure 22.

   Figure 22: SPI Master Component Advanced Configuration



6. 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.

7. Assign the input and output pins. In the Workspace Explorer window, double-click the file *SPI_Bootloader_Host.cydwr* and assign the pins as Figure 23 shows.

   Figure 23: SPI_Bootloader_Host Project Pin Assignments

Name	Port	Pin
\LCD_Char:LCDPort[6:0]\	P2[6:0]	2,1,99…95
MISO	P5[3]	19
MOSI	P5[1]	17
Pin_Switch	P6[1]	90
SCLK	P5[5]	32

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 external variable in *ISR_Switch.c*.

9. Add firmware to this project. The SPI_Bootloader_Host 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 19 on page 9).

   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.

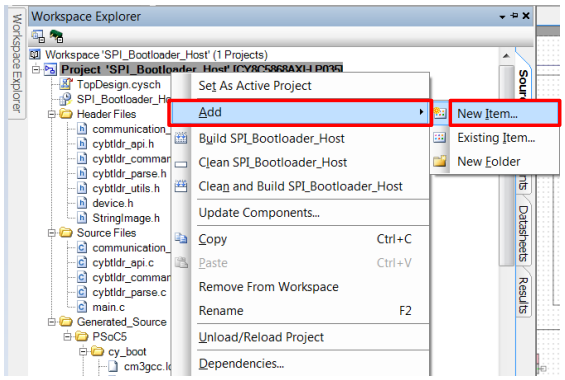
10. As explained previously, a bootloader host core is built upon four API files. These files do all of the host bootloading operations. We must include these files in

our project. Find these API files at the following location:

PSoC Creator \ 4.0 \ 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 24 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 24: 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 SPI_Bootloader_Host project associated with this application note (follow the previous step in adding 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 'StringImage1' and 'StringImage2'. 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 SPI_Bootloader_Host project associated with this application note.

13. Build the project and program it into a PSoC 5LP on CY8CKIT-050.

Testing the Projects

Note The *main.c* file of the SPI_Bootloader_Host project 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'.

Kit Configuration

To test the projects, configure the kits as follows:

For CY8CKIT-030:

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

For CY8CKIT-050:

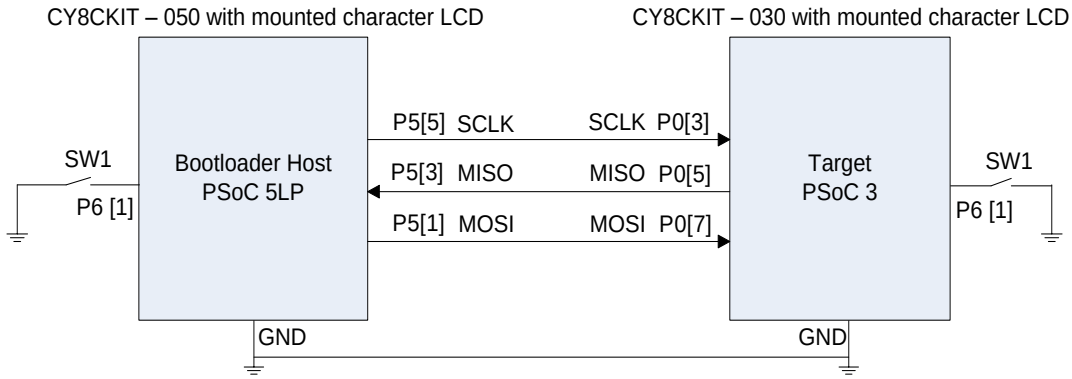
1. Program the PSoC 5LP with the SPI_Bootloader_Host project.
2. Set jumpers J10 and J11 to 5 V.
3. Connect the character LCD to Port 2 [6:0].

Make the following connections between the two DVKs:

1. P0 [7] of CY8CKIT-030 to P5 [1] of CY8CKIT-050
2. P0 [5] of CY8CKIT-030 to P5 [3] of CY8CKIT-050
3. P0 [3] of CY8CKIT-030 to P5 [5] of CY8CKIT-050
4. Short together the ground pins of the kits.

The connections are illustrated in Figure 25.

Figure 25: Host / Target Connections



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_1.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 the next button press on CY8CKIT-050, the *Bootloadable_2.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.

Summary

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

Related Application Notes

You can refer to the following application notes for better understanding of the bootloaders and flash programming.

- AN73854 – PSoC 3 and PSoC 5LP Introduction to Bootloaders
- AN60317 – PSoC 3 and PSoC 5LP I2C Bootloader
- AN73503 – USB HID Bootloader for PSoC 3 and PSoC 5LP
- AN68272 – PSoC 3 and PSoC 5LP Customizing 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.

Related Projects

The projects attached to this application note are organized as shown in Table 4.

Table 4: Projects associated with this application note

Design Project Name	Description
SPI_Bootloader_Host	Embedded bootloader host project with SPI as the communication channel.
SPI_Bootloader	Bootloader project that has SPI as the communication channel.
Bootloadable_1	Bootloadable project that displays "Hello" on the Character LCD of the target device.
Bootloadable_2	Bootloadable project that displays "Bye" on the Character LCD of the target device.

Appendix A – Memory

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 26 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 error correction code (ECC) 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 26 also shows that the first few rows of flash are occupied by the bootloader. This space includes:

- 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:

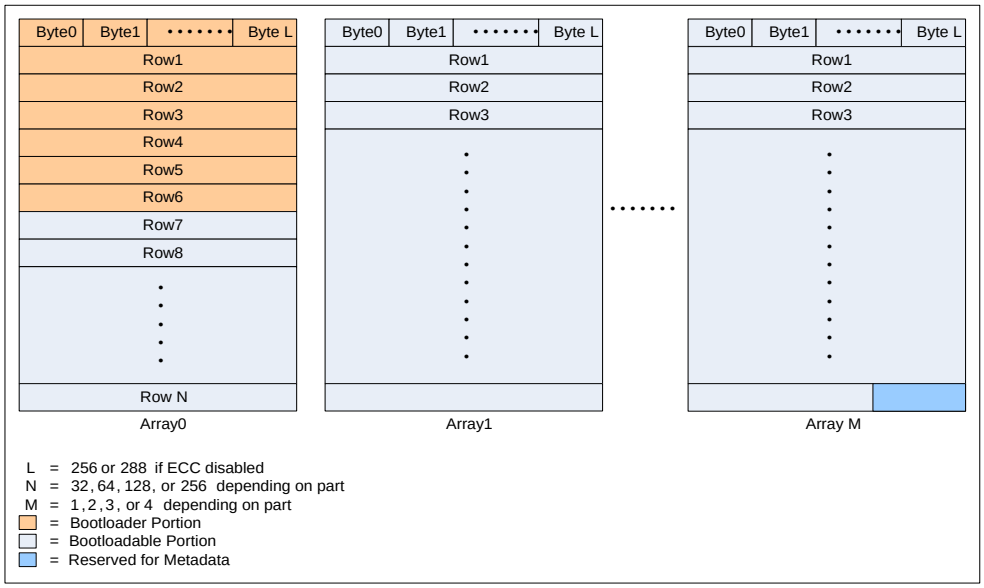
- Vector table for the bootloadable project (PSoC 5LP only)
- The bootloadable project code and data

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 (4 bytes).

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

Figure 26: Physical Organization of Flash Memory in PSoC



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

Figure 27: Flash Memory Usage

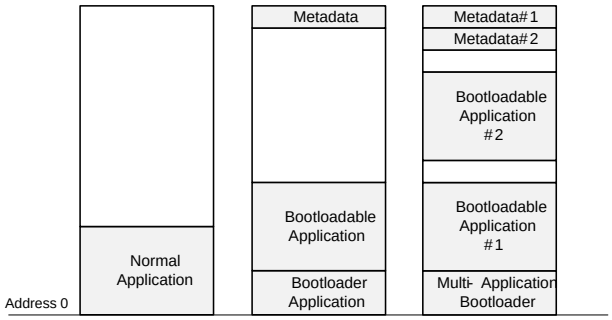


Table 5. Metadata Layout

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.

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 6 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 6: 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

When the bootloader portion of the flash is configured to have the Full protection level, 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:

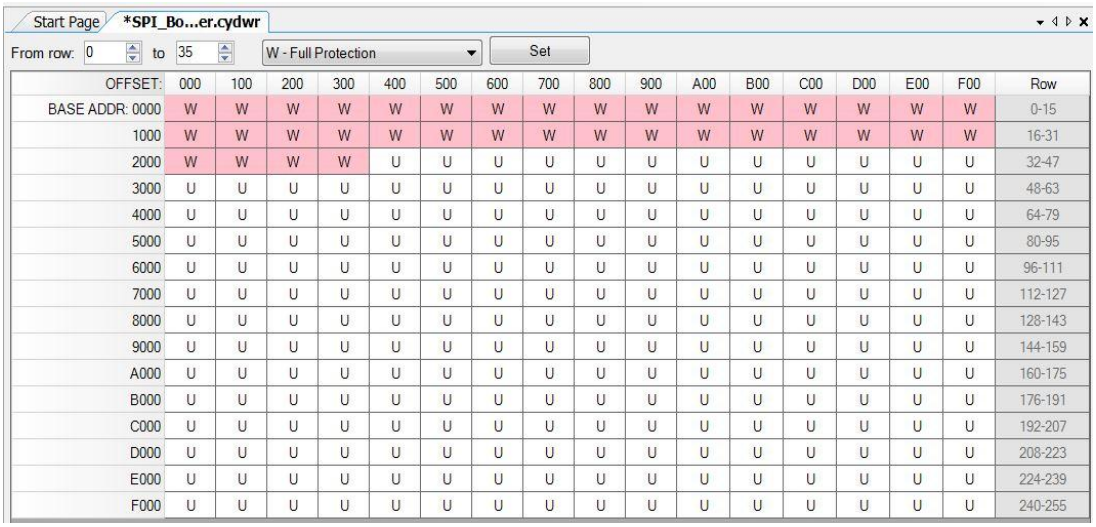
Example for Flash Protection

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 SPI_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), that is, 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 28 shows.

Figure 28: Flash Protection in PSoC Creator



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

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.

*.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]
[2bytes data length] [N bytes of data]
[1 byte checksum]
```

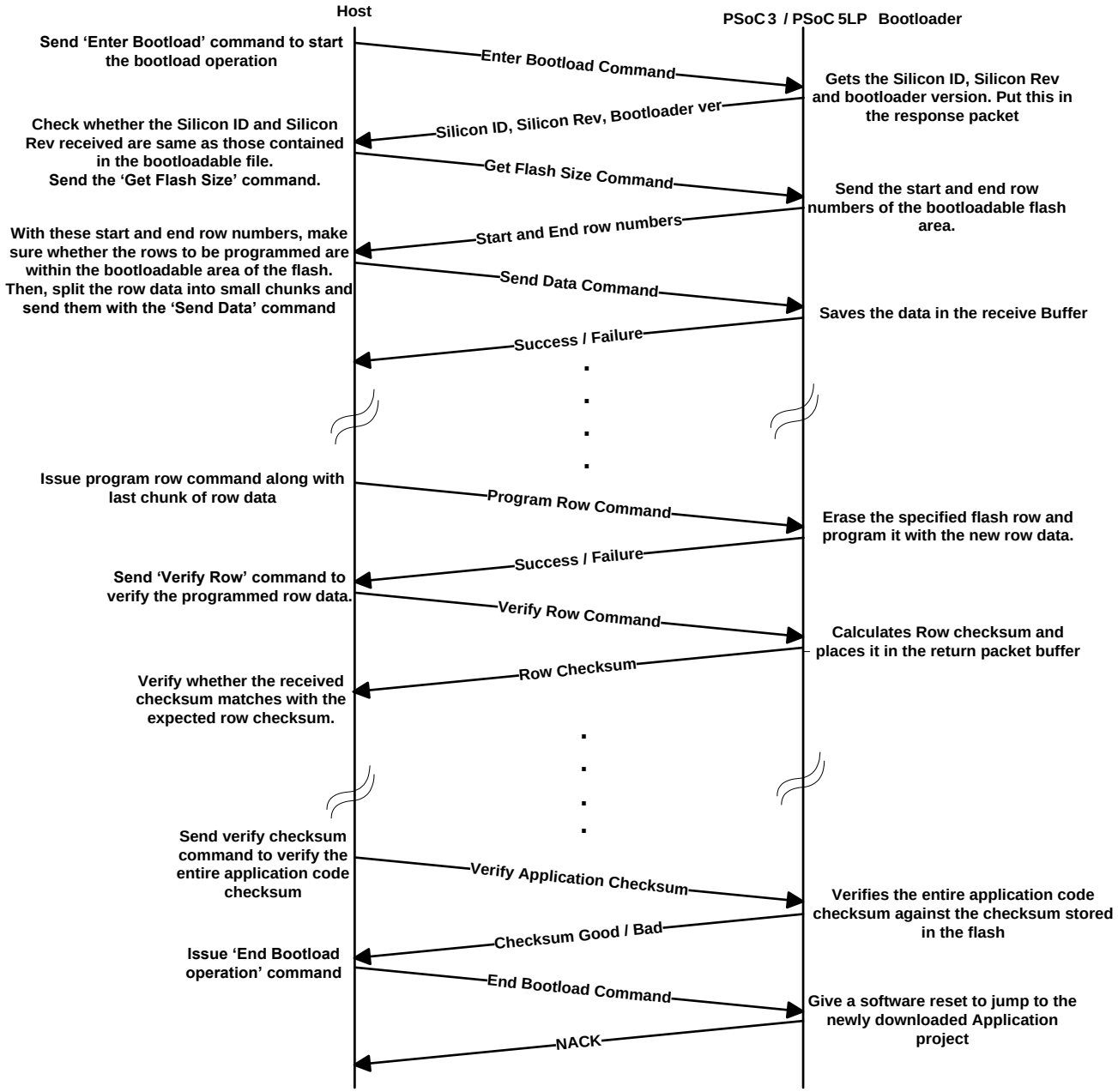
The checksum type in the header indicates the type of checksum used in the packets sent between the bootloader and the bootloader host during the bootloading operation. If this byte is 0, the checksum is a basic summation. If it is 1, the checksum is CRC-16.

Appendix C – Host / Target Communications

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 29 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.

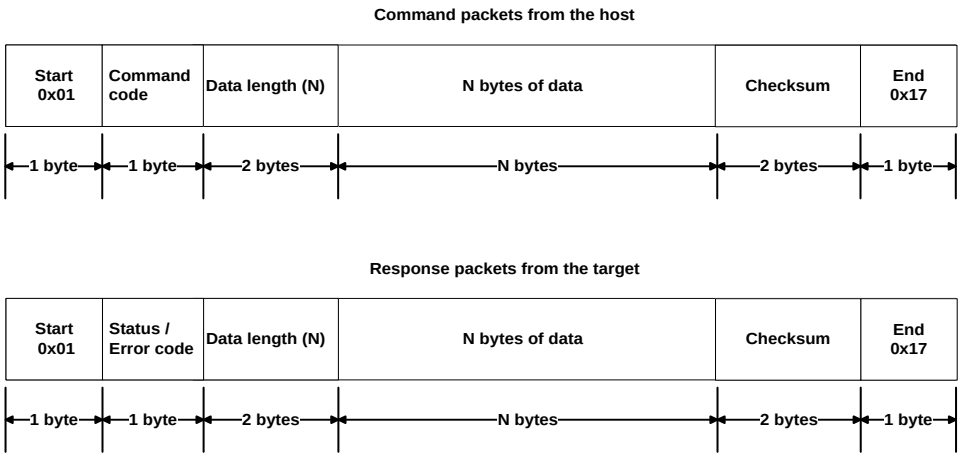
Figure 29: Communication Flow During Bootloading



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 30 shows.

Figure 30: 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 7.

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 7 provides a list of commands and their expected responses. Table 8 provides a list of status and error codes.

Table 7: 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. These numbers are for requested array ID.
0x33	Get application status (valid only for dual application bootloader)	Application number, 1 byte	Valid application number, 1 byte; Active application number, 1 byte. Checks whether the specified application is valid and it is active.
0x34	Erase row	Flash array ID, 1 byte; Flash row number, 2 bytes	N/A. Erases the contents of the specified flash row.

Command byte	Command	Data byte in the command packet	Expected response data bytes
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.
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. Returns the checksum of the specified row.
0x3B	Exit bootloader	N/A	N/A. This command is not acknowledged.

Table 8: 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).

Status / error codes	Label	Description
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 9 describes in detail the functions of this API file.

Table 9: 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, that is, 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.

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, you must 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, you can reduce both configuration time and flash memory usage.

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 10 shows.

Table 10. 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		Eight possible drive modes. See the 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:

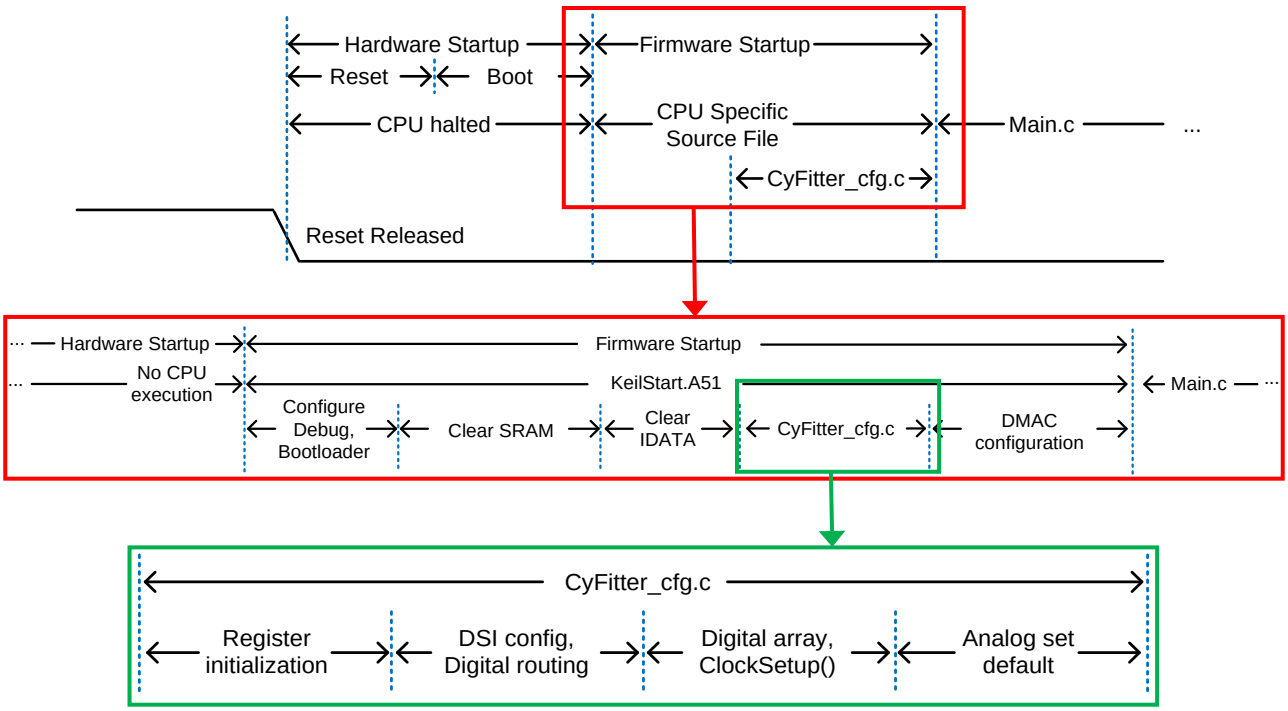
- The **Reset** tab for I/O ports, the individual Pin Component configurations
- The **System** tab for all other NVLs, the design-wide resources (DWR) window

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 31 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 31. Device Startup Process Diagrams



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), I^2C, 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 31). All firmware functions are restored when the project's main() function starts executing.

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 11 shows.

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

I/O Pin Drive Mode	Comment
HI-Z Analog	Optionally, add an external pull-up or pull-down resistor to the PWM pin, for 100 percent 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 percent 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 percent 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 programming specifications to create HSSP are given in PSoC 3 Programming Specifications and PSoC 5LP Programming Specifications. Refer to the application note AN73054 for an example of HSSP programming.

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.

Debugging Bootloadable Projects

In the PSoC Creator bootloader system, the bootloader project executes first (at device reset) and then the bootloadable project. The jump from the bootloader to the bootloadable project is done through a software controlled device reset. This resets the debugger interface, which means that the bootloadable project cannot be run in debugger mode.

To debug a bootloadable project, convert it to Application Type Normal, debug it, and then convert it back to Bootloadable after debugging is done.

Another option is to program the Bootloadable project *.hex* file on to the device and then use **Attach** to run the target option for debugging, while the bootloadable project is running. In this case, you can debug the bootloadable project only from the point where debugger is attached to the device.

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 27 on page 15 shows the placement of two applications in flash memory.

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

1. Set the application type as Multi-App Bootloader while starting the new project. Select the Multi-App 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. Thus, only certain combinations of *.cyacd* files 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.
5. To switch between applications, send the 'Set Active Application' command to the bootloader. You can create this command using the API function CyBtldr_CreateSetActiveAppCmd(). Before sending the Set Active App command, send the 'Enter Bootloader' command and after sending all the commands, send the 'Exit Bootloader command'. For more information on these APIs, refer the *CyBtldr_Command.c / .h* files.

Memory Requirement for Bootloader

A typical SPI 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 32 shows.

Set the Device Configuration Mode to 'Compressed' in *.cydwr* > System tab, as Figure 33 shows, to minimize flash memory usage. Set Device Configuration Mode to DMA if startup time is more important than code size.

Figure 32: Unchecking Optional Commands in Bootloader Component

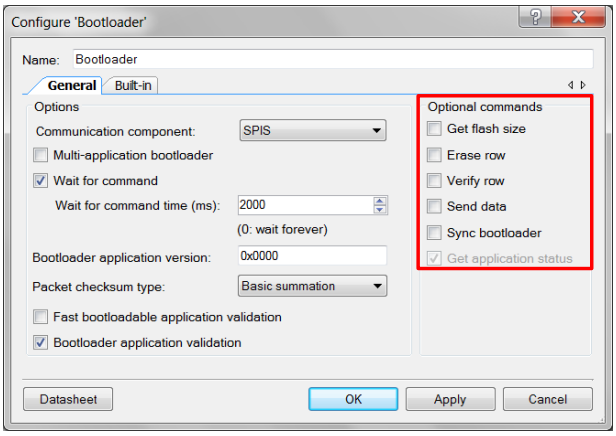
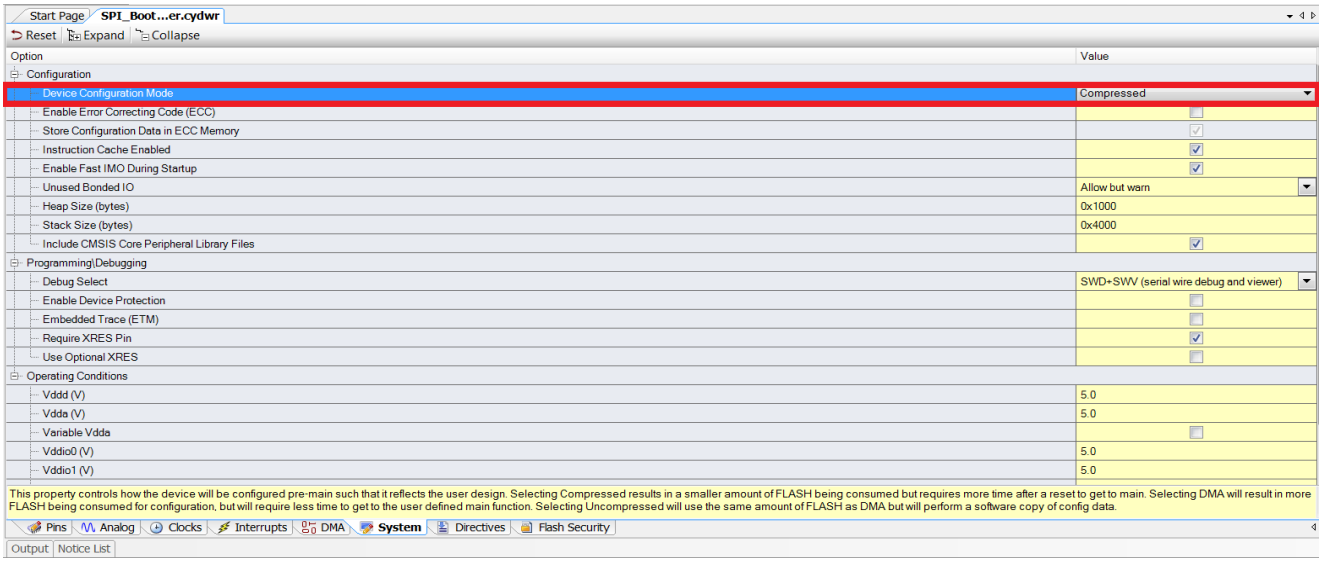


Figure 33: Device Configuration Mode



Document History

Document Title: AN84401 - PSoC® 3 and PSoC 5LP SPI Bootloader

Document Number: 001-84401

Revision	ECN	Orig. of Change	Submission Date	Description of Change
**	3947770	PHAL	03/28/2013	New document.
*A	4356458	RNJT	04/22/2014	Updated projects for PSoC Creator 3.0 SP1.
*B	4435010	MKEA	07/17/2014	Added Appendix E – Bootloader and Device Reset
*C	5688060	AESATMP8	04/19/2017	Updated logo and copyright.
*D	5724110	VKVK	06/16/2017	Updated the project and documentation 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

© Cypress Semiconductor Corporation, 2013-2017. This document is the property of Cypress Semiconductor Corporation and its subsidiaries, including Spansion LLC (“Cypress”). This document, including any software or firmware included or referenced in this document (“Software”), is owned by Cypress under the intellectual property laws and treaties of the United States and other countries worldwide. Cypress reserves all rights under such laws and treaties and does not, except as specifically stated in this paragraph, grant any license under its patents, copyrights, trademarks, or other intellectual property rights. If the Software is not accompanied by a license agreement and you do not otherwise have a written agreement with Cypress governing the use of the Software, then Cypress hereby grants you a personal, non-exclusive, nontransferable license (without the right to sublicense) (1) under its copyright rights in the Software (a) for Software provided in source code form, to modify and reproduce the Software solely for use with Cypress hardware products, only internally within your organization, and (b) to distribute the Software in binary code form externally to end users (either directly or indirectly through resellers and distributors), solely for use on Cypress hardware product units, and (2) under those claims of Cypress’s patents that are infringed by the Software (as provided by Cypress, unmodified) to make, use, distribute, and import the Software solely for use with Cypress hardware products. Any other use, reproduction, modification, translation, or compilation of the Software is prohibited.

TO THE EXTENT PERMITTED BY APPLICABLE LAW, CYPRESS MAKES NO WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, WITH REGARD TO THIS DOCUMENT OR ANY SOFTWARE OR ACCOMPANYING HARDWARE, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. To the extent permitted by applicable law, Cypress reserves the right to make changes to this document without further notice. Cypress does not assume any liability arising out of the application or use of any product or circuit described in this document. Any information provided in this document, including any sample design information or programming code, is provided only for reference purposes. It is the responsibility of the user of this document to properly design, program, and test the functionality and safety of any application made of this information and any resulting product. Cypress products are not designed, intended, or authorized for use as critical components in systems designed or intended for the operation of weapons, weapons systems, nuclear installations, life-support devices or systems, other medical devices or systems (including resuscitation equipment and surgical implants), pollution control or hazardous substances management, or other uses where the failure of the device or system could cause personal injury, death, or property damage (“Unintended Uses”). A critical component is any component of a device or system whose failure to perform can be reasonably expected to cause the failure of the device or system, or to affect its safety or effectiveness. Cypress is not liable, in whole or in part, and you shall and hereby do release Cypress from any claim, damage, or other liability arising from or related to all Unintended Uses of Cypress products. You shall indemnify and hold Cypress harmless from and against all claims, costs, damages, and other liabilities, including claims for personal injury or death, arising from or related to any Unintended Uses of Cypress products.

Cypress, the Cypress logo, Spansion, the Spansion logo, and combinations thereof, WICED, PSoC, CapSense, EZ-USB, F-RAM, and Traveo are trademarks or registered trademarks of Cypress in the United States and other countries. For a more complete list of Cypress trademarks, visit cypress.com. Other names and brands may be claimed as property of their respective owners.