Please note that Cypress is an Infineon Technologies Company.
The document following this cover page is marked as “Cypress” document as this is the company that originally developed the product. Please note that Infineon will continue to offer the product to new and existing customers as part of the Infineon product portfolio.

Continuity of document content
The fact that Infineon offers the following product as part of the Infineon product portfolio does not lead to any changes to this document. Future revisions will occur when appropriate, and any changes will be set out on the document history page.

Continuity of ordering part numbers
Infineon continues to support existing part numbers. Please continue to use the ordering part numbers listed in the datasheet for ordering.

AN68272

PSoC® 3, PSoC 4, and PSoC 5LP UART Bootloader

Authors: Anu M D, Siddalinga Reddy

Associated Project: Yes

Associated Part Family: CY8C3xxx, CY8C42xx, CY8C4Axx, CY8C40xxS, CY8C41xxS, CY8C41xxPS, CY8C5xxx

Software Version: PSoC Creator™ 4.2

Related Application Notes: For a complete list, Related Application Notes

More code examples? We heard you.

To access an ever-growing list of hundreds of PSoC code examples, please visit our code examples web page. You can also explore the PSoC 4 video library here.

AN68272 describes a UART-based bootloader for PSoC® 3, PSoC 4, and PSoC 5LP. In this application note, you will learn how to use PSoC Creator™ to quickly and easily build a UART-based bootloader project and bootloadable projects. It also shows how to build a UART-based embedded bootloader host program.

Contents

1 Introduction

Bootloaders are a common part of an MCU system design. A bootloader makes it possible for a product's firmware to be updated in the field. At the factory, the firmware is initially programmed into a product typically through the MCU's Joint Test Action Group (JTAG) or the Arm® serial wire debug (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:

- How to create a UART bootloader using PSoC Creator
- Bootloader host topics:

- How to use the Bootloader Host tool
- The basic building blocks and functionality of a bootloader host system
- How to create an embedded UART bootloader host using PSoC 5LP

This application note assumes that you are familiar with PSoC and the PSoC Creator Integrated Design Environment (IDE). If you are new to PSoC 3, PSoC 4 or PSoC 5LP refer to AN54181 – Getting Started with PSoC 3, AN79953 – Getting Started with PSoC 4, AN77759 – Getting Started with PSoC 5LP respectively. If you are new to PSoC Creator, see the PSoC Creator home page.

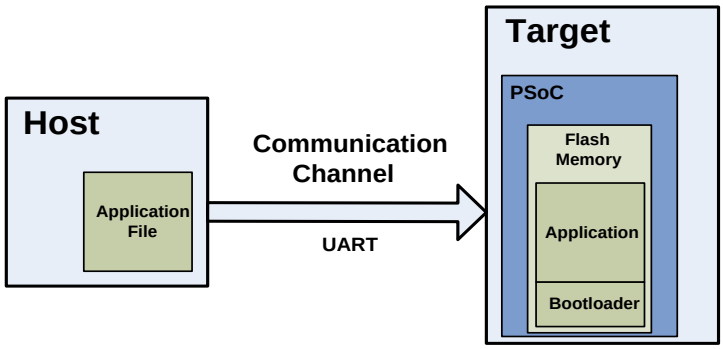
This application note also assumes that you are familiar with bootloader concepts. If you are new to these concepts, see AN73854 – PSoC 3, PSoC 4, and PSoC 5LP Introduction to Bootloaders. For a complete list of other application notes on bootloading, see Related Application Notes.

Finally, this application note assumes that you are familiar with the UART protocol and the PSoC Creator UART Component. If you are new to the UART Component, see the PSoC Creator UART Component datasheet. You can also get the datasheet by right-clicking on the UART 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. The portion of the embedded firmware that knows how to update the flash is called the “bootloader.” The other terms in Figure 1 are defined in the following paragraphs.

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 (PC host) or another MCU (embedded host such as PSoC 5LP device) 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 a “bootload” for short. The firmware that is placed in flash is called the “application” or the “bootloadable.”

Another common term for bootloading is “in-system programming (ISP).” Cypress has a product with a similar name but a different function called in-system serial programming (ISSP)” and an operation called “host-sourced serial programming (HSSP).” For more information, see AN73054 – PSoC 3 and PSoC 5LP 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 target 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.

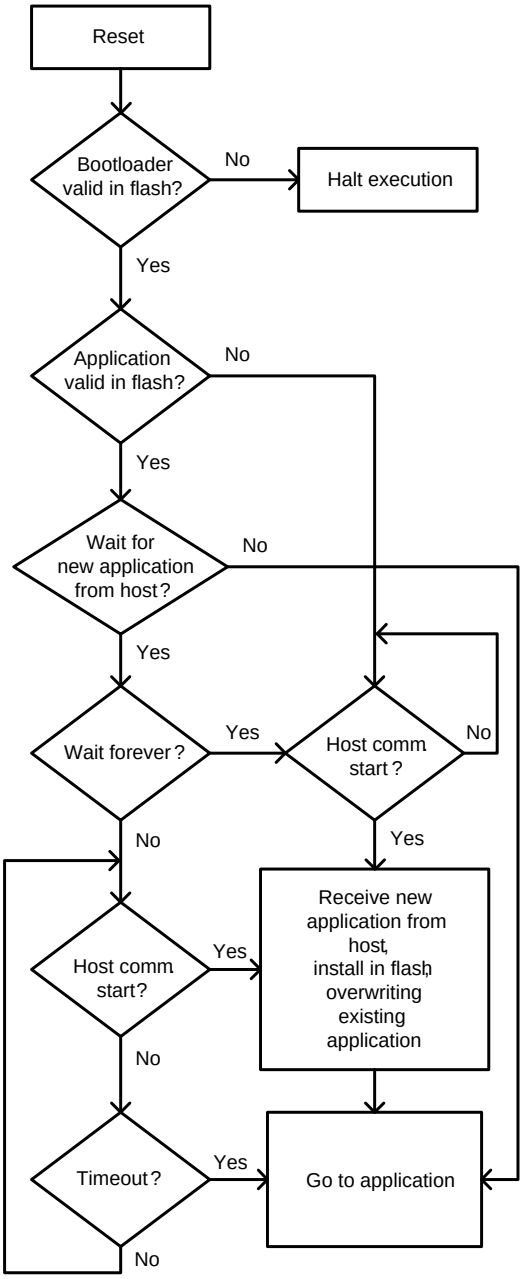
1.3 Bootloader Function Flow

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
- Passes control to the application

Figure 2 shows the typical bootloader functions.

Figure 2. Bootloader Function Flow



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 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 application programming interface (API) function 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, you can customize the bootloader project to check for some user input before calling `Bootloader_Start()` and running through its normal routine.

For example, the bootloader may monitor the UART and wait forever for a user command before calling `Bootloader_Start()`. For more information, refer to AN73854 – PSoC 3, PSoC 4, and PSoC 5LP Introduction to Bootloaders.

2 Projects

This section shows you the steps to create the following PSoC Creator projects:

- UART bootloader
- Bootloadable
- Embedded bootloader host

The projects are designed to be used with Cypress development kits. Kit selection is based on the target device; refer to Kit Selection for details. You may be required to change the pins connection based on the kit. Review the specific kit documentation for connections. The projects can be easily adapted for other custom boards.

2.1 UART Bootloader

In this section, you create and build a UART-based bootloader project. One feature of this project is that while bootloading is taking place, the kit's LED blinks.

1. Create a new PSoC Creator project, and name it "UART_Bootloader." Select the target device and create a new workspace for the project.

   Note: For PSoC Creator 3.1 and lower version, the application type should be specified while creating the project. To do so, click on the "+" button next to the **Advanced** tab to expand the configuration options. Select **Bootloader** as the application type.

2. Add a UART Component to the TopDesign schematic, as Figure 3 and Figure 4 show.

Figure 3. UART Component for PSoC 3 and PSoC 5LP

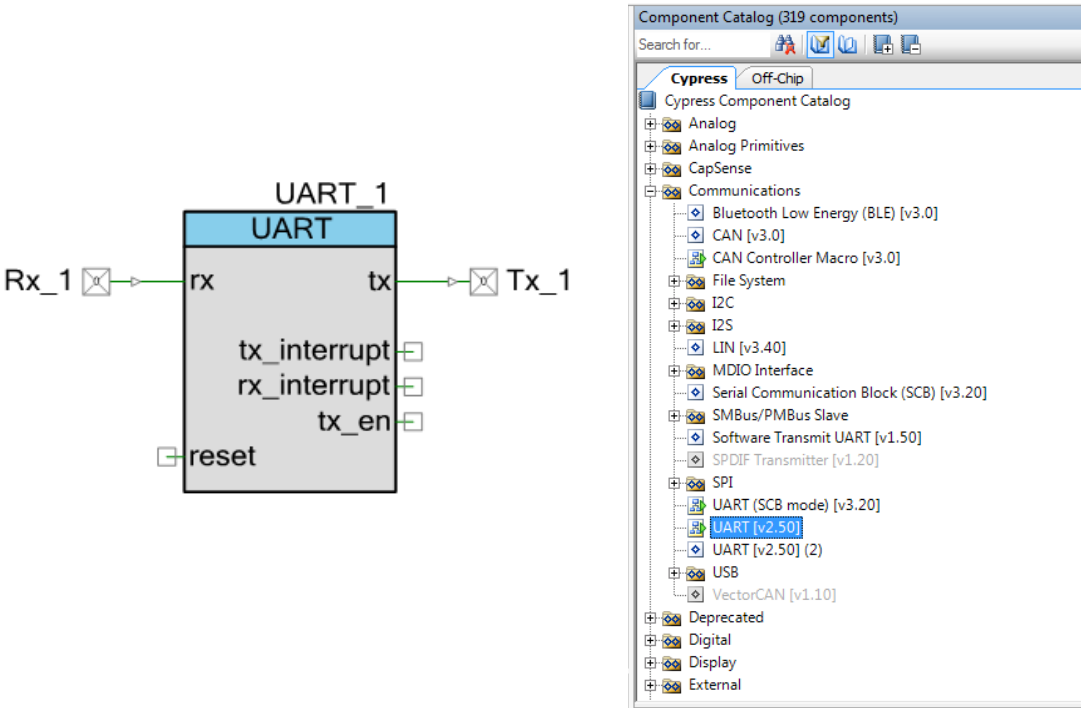
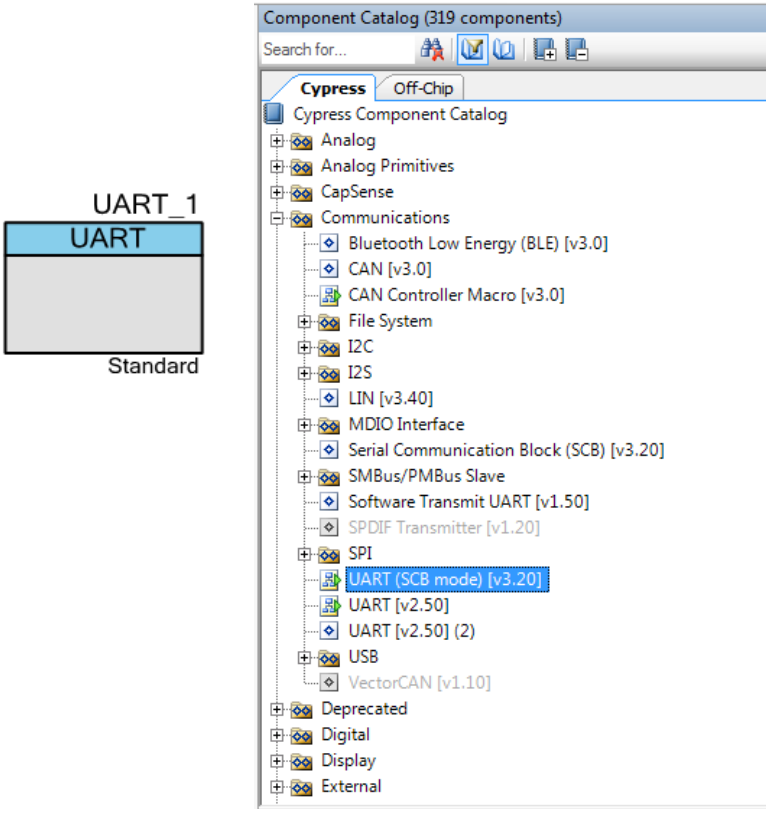
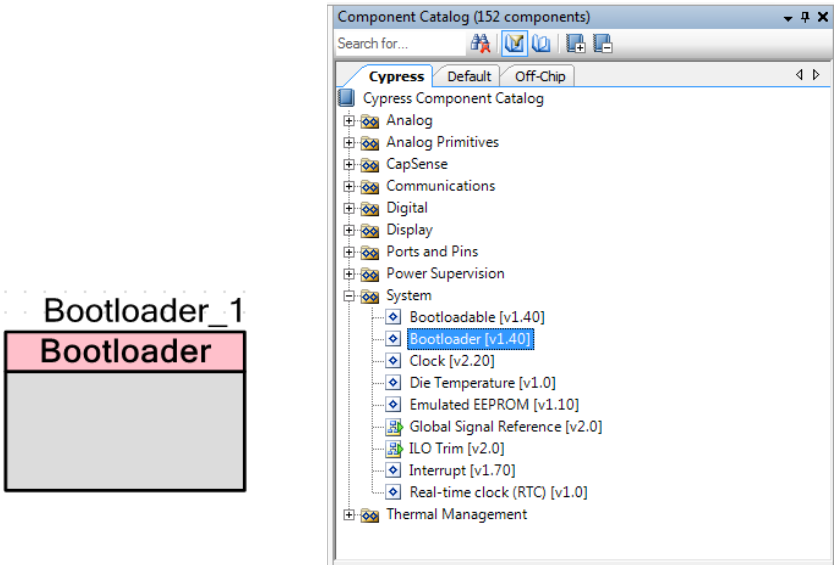


Figure 4. UART Component for PSoC 4



3. Add a Bootloader Component to the TopDesign schematic, as Figure 5 shows.

Figure 5. Bootloader Component



4. To blink an LED, add a PWM (TCPWM for PSoC 4), a Clock, and a Digital Output Pin Component to the schematic.
5. Rename the Components and pins, as Table 1 shows.

Table 1. Bootloader Project Component Names

Component	Name
Bootloader_1	Bootloader
UART_1	UART
Rx_1	Rx
Tx_1	Tx
Clock_1	Clock
Pin_1	Pin_LED
PWM_1 / TCPWM_1	PWM

6. With an LED and resistor added as annotation Components, the TopDesign of the project for PSoC 3 and PSoC 5LP looks similar to Figure 6, and the TopDesign for PSoC 4 looks similar to Figure 7.

Figure 6. TopDesign of UART_Bootloader Project for PSoC 3 and PSoC 5LP

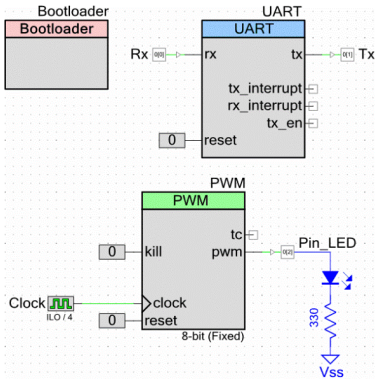
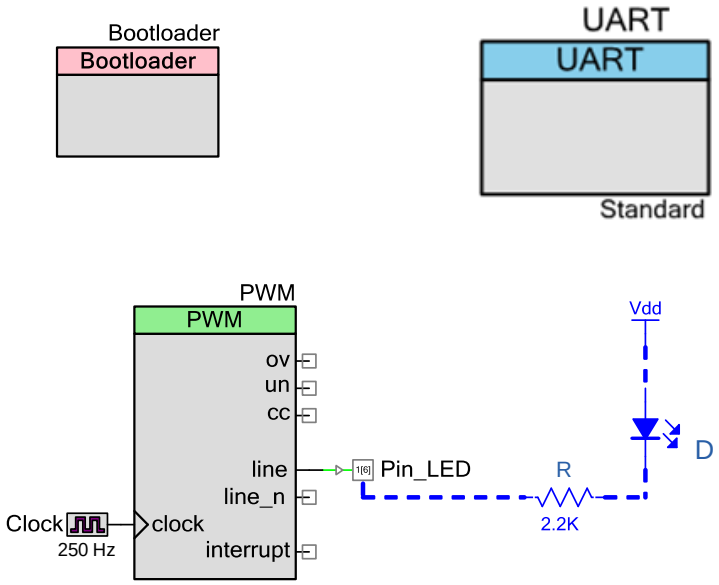


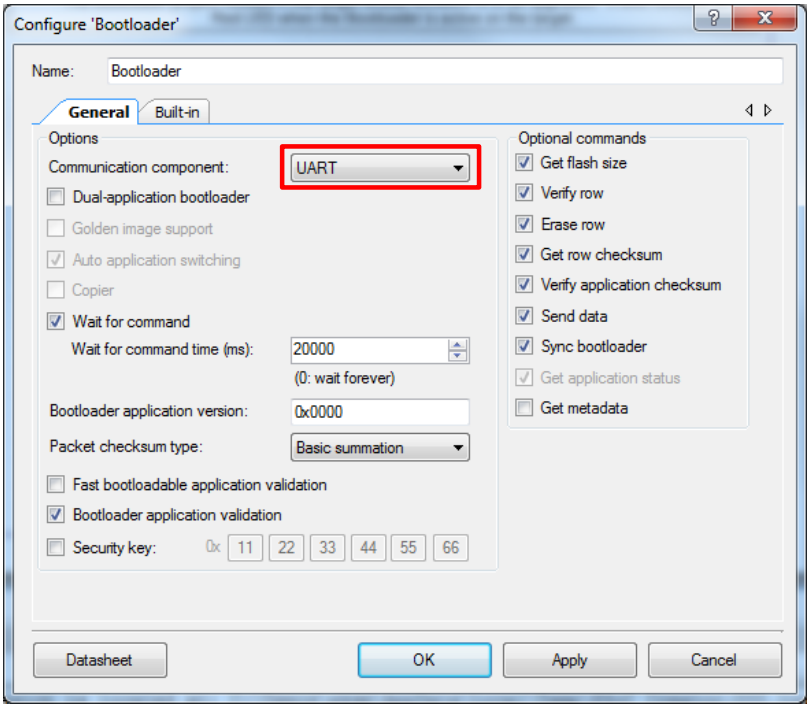
Figure 7. TopDesign of UART Bootloader Project for PSoC 4



In this example, there is no need to reset the UART so the reset terminal is connected to a Logic Low '0' Component.

7. To configure the Bootloader, double-click on the Component.
8. Select **UART** as the **Communication component**, as Figure 8 shows. Leave the other parameters at their default settings. For more information on these configuration parameters, refer to the Bootloader Component datasheet.

Figure 8. Bootloader Configuration



9. To configure the UART Component, double-click on it. By default, it is in Full UART mode with a data rate of 57,600 bps. Leave all parameters at their default settings. Figure 9 and Figure 10 shows the basic configuration tab of the UART Component.

Figure 9. Basic UART Configuration for PSoC 3 and PSoC 5LP

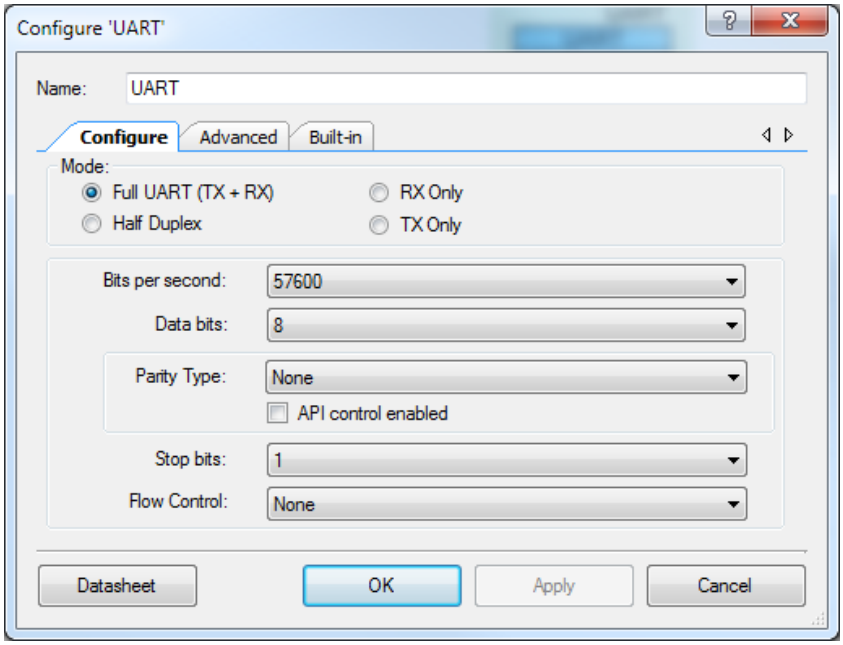
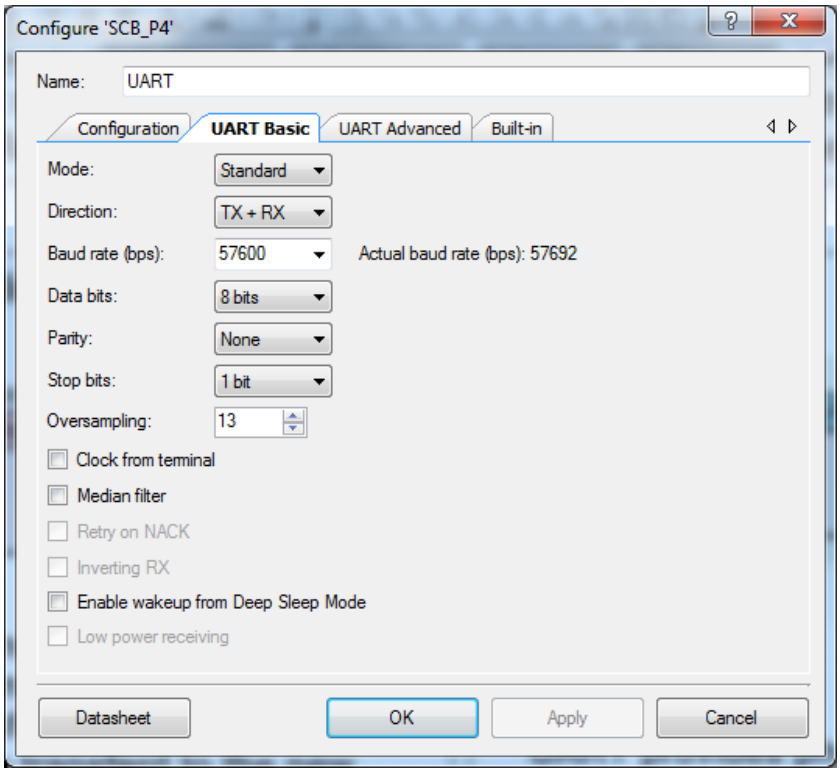


Figure 10. Basic UART Configuration for PSoC 4



10. Click on the **Advanced** tab of the UART configuration window.
11. 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 and Figure 12.

Figure 11. Advanced UART Configuration for PSoC 3 and PSoC 5LP

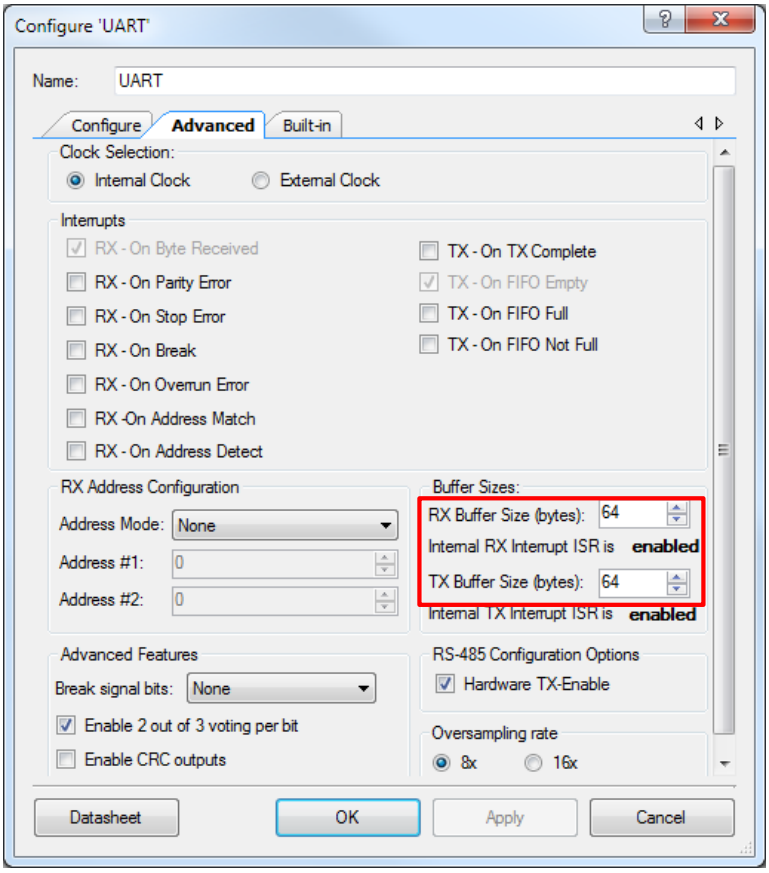
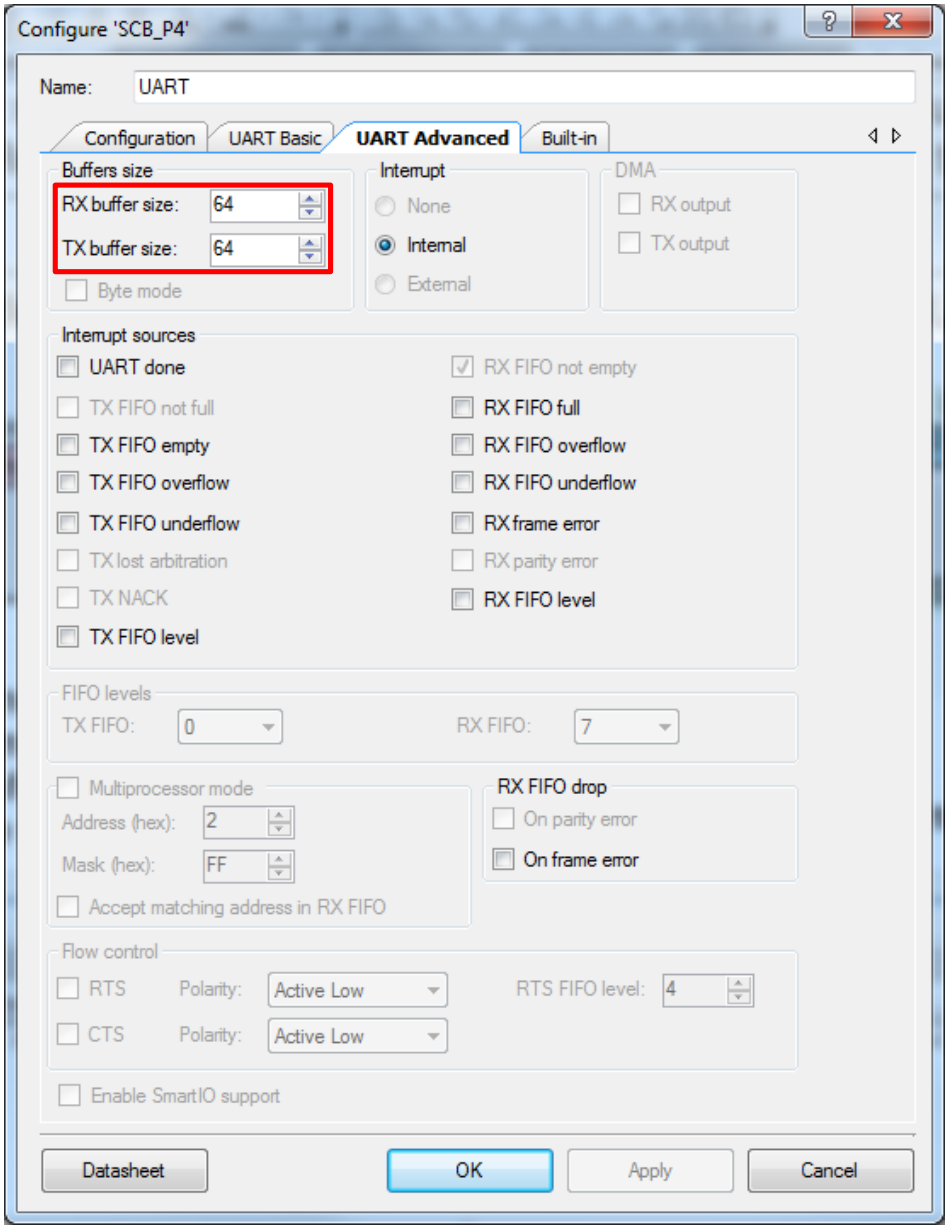


Figure 12. Advanced UART Configuration for PSoC 4



12. To configure the PWM Component, double-click on it. Set the **Period** to **255**, and the **Compare** to **127**. Leave the other parameters at their default settings.
13. To configure the Clock Component, double-click on it. Set the **Frequency** to **ILO / 4**, or ~250 Hz. Leave the other parameters at their default settings.
14. For the Pin_LED Component, leave the parameters at their default settings.
15. Assign the Pin Components to physical pins. In the **Workspace Explorer** window, double-click the *UART*_Bootloader.*cydwr* file, and click on the **Pins** tab. Table 2 shows the pin assignments for different kits. Pin assignments depend on the kit; refer to the kit user guide for information.

Table 2. Pin Assignments for UART Bootloader Projects for Kits

Pin Name	CY8CKIT-030	CY8CKIT-050	CY8CKIT-042	CY8CKIT-041-40xx	CY8CKIT-041-41xx	CY8CKIT-149
\UART:rx\	P0[0]	P0[0]	P4[0]	P0[4]	P0[4]	P7[0]
\UART:rx\	P0[1]	P0[1]	P4[1]	P0[5]	P0[5]	P7[1]
Pin_LED	P6[2]	P6[2]	P1[6]	P3[4]	P3[4]	P3[4]

16. Add the `Bootloader_Start()` function to the *main.c* file. This API function does the entire bootloading operation. It does not return—it ends with a software device reset. Therefore, any code that is placed after this API call is not executed.

    Add the `PWM_Start()` function to initialize the PWM in `main()`, as Code 1 shows. For more information on this Component API, see the PWM Component datasheet.

Code 1. PWM Initialization in Bootloader

```
void main()
{
  /* Initialize PWM */
  PWM_Start();

  Bootloader_Start();

  /* Uncomment this line to enable
     global interrupts. */
  /* CyGlobalIntEnable; */

  for(;;)
  {
    /* Place your code here. */
  }
}
```

17. Build the project and program it into a specific kit based on your target device selection. Refer to Kit Selection to get kit-related information based on your device selection.

You have now created a simple UART-based bootloader. It can communicate with a host and download the bootloadable project. The bootloader can be expanded and customized in a number of ways; see the Bootloader and UART Component datasheets and AN73854 – PSoC 3, PSoC 4 and PSoC 5LP Introduction to Bootloaders for details.

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

Now you will see how to create bootloadable applications that can be used with this bootloader.

2.2 PSoC 3 and PSoC 5LP Bootloadables

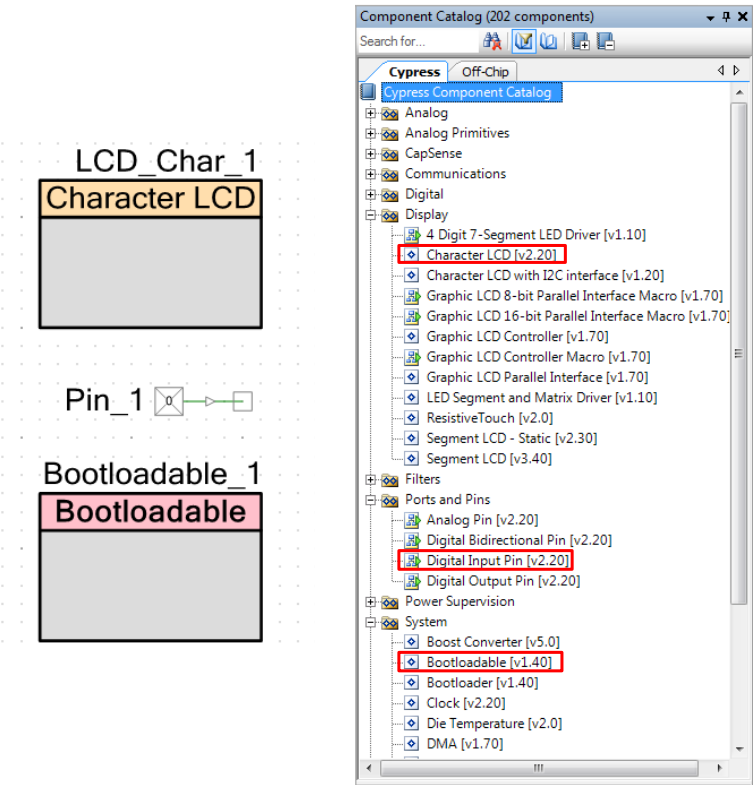
This section involves creating two bootloadable projects. They are very similar—one displays "Hello" on the kit's character LCD and the other displays "Bye." The section describes the steps to create these bootloadable projects.

1. Create a new PSoC Creator project, and name the project "Bootloadable1." The devices for this project and the UART_Bootloader project must be the same.

   Note: For PSoC Creator 3.1 and lower versions, the application type should be specified while creating the project. To do so, click on the "+" button next to the **Advanced** tab to expand the configuration options. Select **Bootloadable** as the application type.

2. For this project, you need the Bootloadable, Digital Input Pin, and LCD Components. Add these Components to your TopDesign schematic, as Figure 13 shows.

Figure 13. Bootloadable1 Project Components



3. Rename the Components according to Table 3.

Table 3. Bootloadable Project Component Names

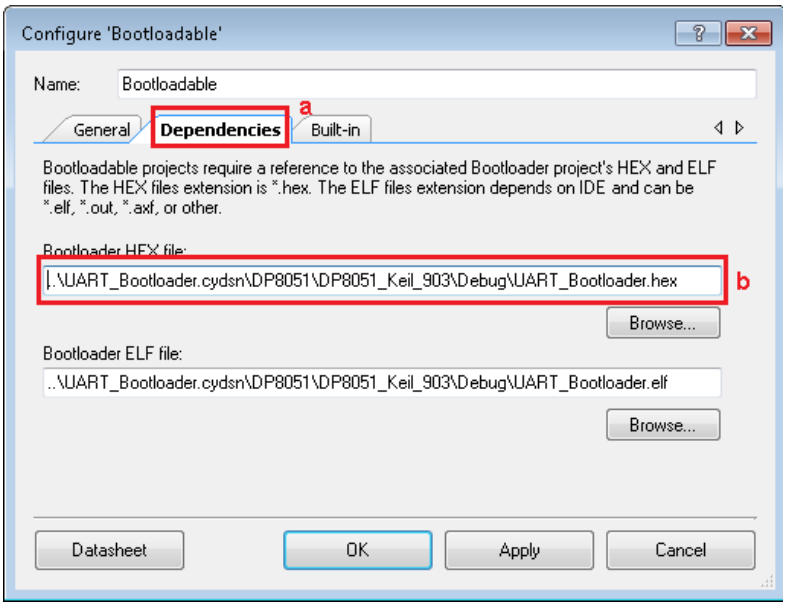
Component	Name
Bootloadable_1	Bootloadable
Pin_1	Pin_StartBootloader
LCD_Char_1	LCD_Char

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 configure the component, go to the **Dependencies** tab of the Bootloadable Component configuration window, as Figure 14 shows.

B. Select the *UART_Bootloader.hex* file, as Figure 14 shows. For more information on Bootloader Component configuration, see the Bootloader Component datasheet.

Figure 14. Bootloadable Component Configuration



You can find the *UART_Bootloader.hex* file in the bootloader project's Debug or Release folder:

When PSoC 3 is the bootloader,

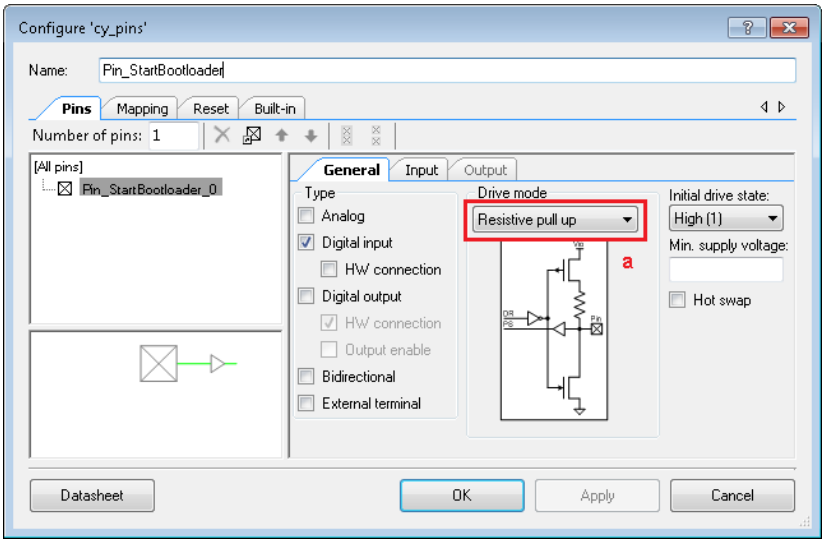
..\UART_Bootloader\UART_Bootloader.cydsn\DP8051\DP8051_Keil_951\Debug\UART_Bootloader.hex

When PSoC 5LP is the bootloader,

..\UART_Bootloader\UART_Bootloader.cydsn\CortexM3\ARM_GCC_493\Debug\UART_Bootloader.hex

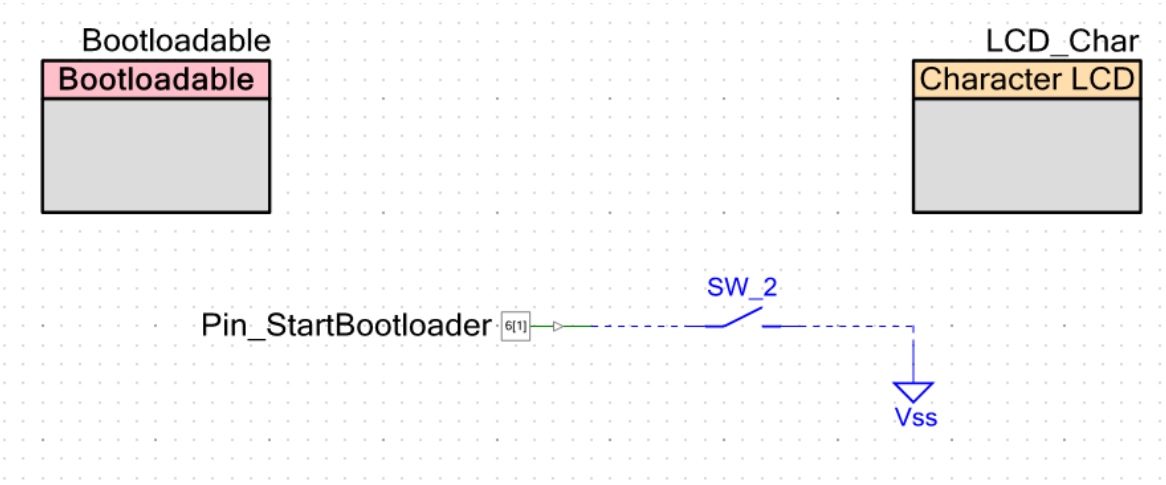
5. The digital input pin Pin_StartBootloader is used to switch from the application back to the bootloader. When the DVK button is pressed, it shorts to ground, so configure the drive mode of the Pin to be **Resistive pull up**, as Figure 15 shows.

Figure 15. Digital Input Pin Configuration



6. With the addition of the annotation Components for the button and the input pin, the TopDesign is complete; it should be similar to Figure 16.

Figure 16. TopDesign of Bootloadable1 Project for PSoC 3 and PSoC 5LP



7. Assign the Pin Components to physical pins. In the **Workspace Explorer** window, double-click the *Bootloadable1.cydwr* file and assign the pins as Figure 17 shows.

Figure 17. Pin Assignments of Bootloadable1 Project

Name	Port
\LCD_Char:LCDPort[6:0]\	P2[6:0]
Pin_StartBootloader	P6[1]

On the CY8CKIT-030 and CY8CKIT-050 kit boards, the LCD pins are hardwired to P2[6:0], and SW2 is hardwired to P6[1].

8. A completed Bootloadable1 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 status of the Pin_StartBootloader. When this pin shorts to ground, the API function `Bootloadable_Load()` is called to invoke the bootloader. The bootloader waits indefinitely for the host to start the bootload operation.

9. 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.
10. To create the other bootloadable project that displays "Bye," repeat the previous steps in this section. Name the project “Bootloadable2.” The only difference between the two projects is that the code in *main.c* displays "Bye" instead of "Hello."

    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.

To bootload this project using a PC host, see Bootloading Using a PC Host.

2.3 PSoC 4 Bootloadables

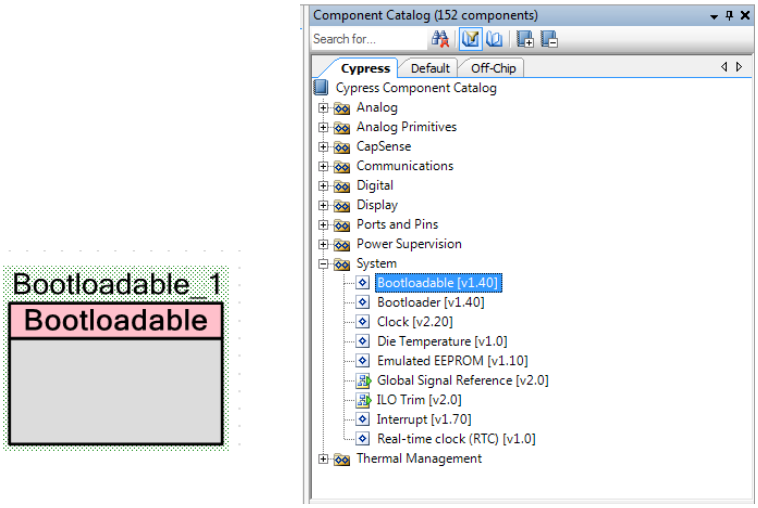
In this section, you will create two bootloadable projects for PSoC 4. These projects are designed for the CY8CKIT-042 kit, but they can be easily adapted to other kits that use other PSoC 4 devices. For kit-related information, refer to Kit Selection and the kit user manual. These projects are very similar–one blinks the green LED and the other blinks the blue LED on the kit. This section describes the steps for creating these bootloadable projects.

1. Create a new PSoC Creator project, and name the project “Bootloadable1.” The devices for this project and the UART_Bootloader project must be the same.

   Note: For PSoC Creator 3.1, the application type should be specified while creating the project. To do so, click on the “+” button next to the **Advanced** tab to expand the configuration options. Select **Bootloadable** as the application type.

2. For this project, you need the Bootloadable, Digital Input Pin and Digital Output Pin Components. Add these Components to your TopDesign schematic, as Figure 18 shows.

Figure 18. Bootloadable1 Project Components



3. Rename the Components according to Table 4.

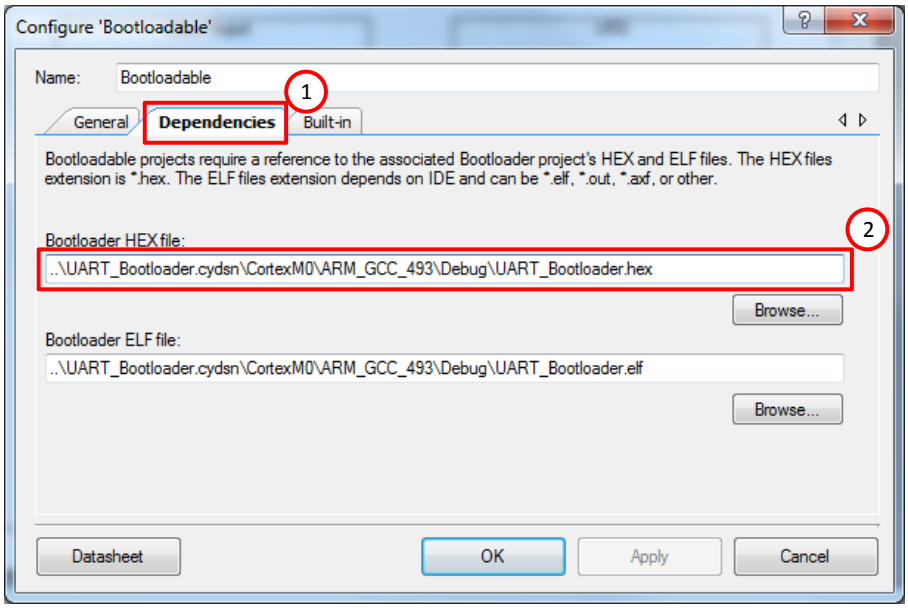
Table 4. Bootloadable1 Component Names

Component	Name
Bootloadable_1	Bootloadable
Pin_1	Green_LED
Pin_2	Pin_StartBootloader

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 configure the component, go to the **Dependencies** tab of the Bootloadable Component configuration window, as Figure 19 shows.
   B. Select the *UART_Bootloader.hex* file as shown. For more information on the Bootloader Component configuration, see the Bootloader Component datasheet.

Figure 19. Bootloadable Component Configuration

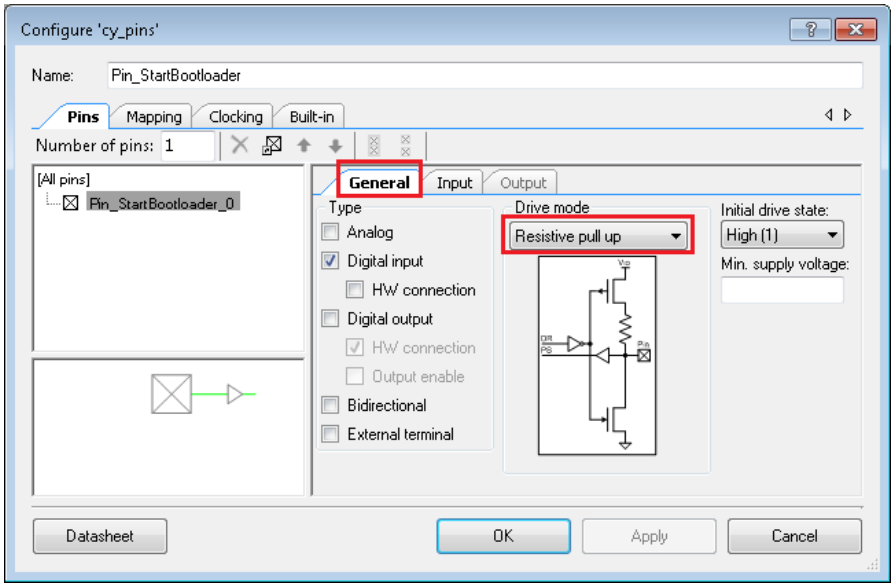


You can find the *UART_Bootloader.hex* file in the bootloader project's Debug or Release folder:
For PSoC 42xx,
..\UART_Bootloader\UART_Bootloader.cydsn\Cortex M0\ARM_GCC_493\Debug\UART_Bootloader.hex

For PSoC CY8C4Axx, CY8C40xxS, CY8C41xxS and CY8C41xxPS,
..\UART_Bootloader\UART_Bootloader.cydsn\Cortex M0p\ARM_GCC_493\Debug\UART_Bootloader.hex

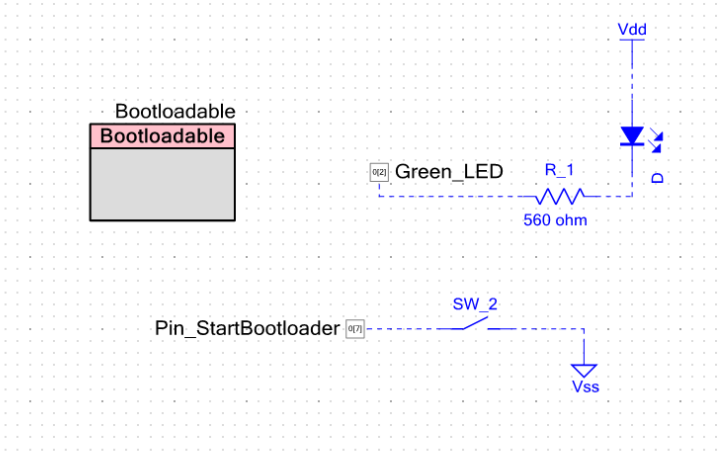
5. The digital input pin (Pin_StartBootloader) is used to switch from the application back to the bootloader. When the DVK button is pressed, it shorts to ground, so configure the drive mode of the pin to be **Resistive pull up**, as Figure 20 shows.

Figure 20. Digital Input Pin Configuration



6. With the addition of annotation Components for the button and the input pin, the TopDesign is complete; it should be similar to Figure 21.

Figure 21. TopDesign of Bootloadable1 Project



7. Assign the Pin Components to physical pins. In the **Workspace Explorer** window, double-click the *Bootloadable1.cydwr* file and assign the pins. Table 5 shows the pin assignment for bootloable1 project for different kits.

Table 5. Pin Assignments for Bootloadable1 for Kits

Pin Name	CY8CKIT-042	CY8CKIT-041-40xx	CY8CKIT-041-41xx	CY8CKIT-149
Green_LED	P0[2]	P2[6]	P2[6]	P5[2]
Pin_StartBootloader	P0[7]	P0[7]	P0[7]	P3[7]

On the CY8CKIT-042 kit board, the green LED is hardwired to P0[2], and SW2 is hardwired to P0[7].

8. A completed bootloadable project for PSoC 4 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 status of the Pin_StartBootloader. When this pin shorts to ground, the API function `Bootloadable_Load()` is called to invoke the bootloader. The bootloader waits indefinitely for the host to start the bootload operation.

9. 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.

10. To create the other bootloadable project that blinks the blue LED, repeat the previous steps in this section. Name the project “Bootloadable2.” The only difference between the two projects is that the Digital Output Pin Blue_LED connection will be different.

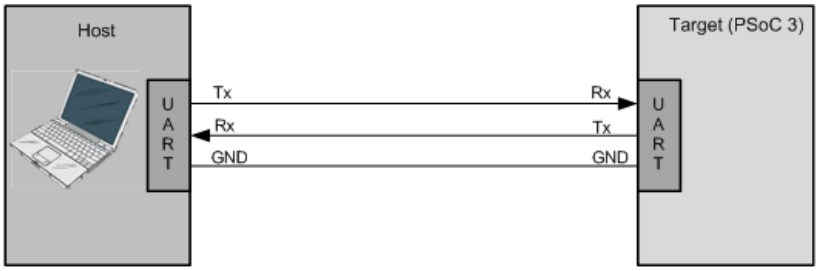
    Note: For PSoC Creator versions before 3.0, if the bootloader is updated, you must also rebuild all bootloadable projects that depend on that bootloader project. Use the “Clean and Build” option.

    You will now bootload this project into a target PSoC 4 using the UART Bootloader Host application (PC host).

2.4 Bootloading Using a PC Host

PSoC Creator provides a Bootloader Host tool for bootloading an application from a PC host, as Figure 22 shows. This tool can be accessed through **Tools** > **Bootloader Host**.

Figure 22. Bootloading Using a PC Host



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

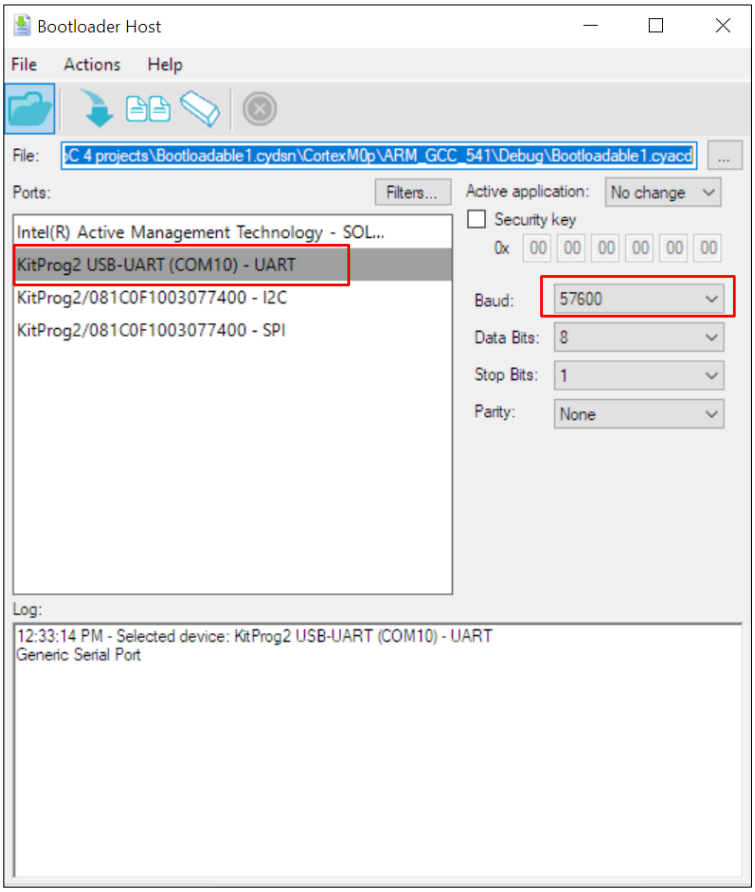
1. To bootload PSoC 3 or PSoC 5LP, connect an RS-232 serial cable to port (P7) of CY8CKIT-030/CY8CKIT-050.

   To bootload PSoC 4, an RS-232 serial cable is not required, as the PSoC 5LP on the PSoC 4 kits has a USB–UART bridge. Therefore, you only need to provide an external connection between the UART port pins of PSoC 4 and the onboard PSoC 5LP on the kit. For example, in CY8CKIT-042 connect P0[0] to pin 10 on the header J8 and P0[1] to pin 9 on the header J8. Refer to the kit guide for details if you have a different PSoC 4 kit.

2. Open the Bootloader Host application from PSoC Creator (**Tools** > **Bootloader Host**). Select the appropriate COM port and baud rate, as Figure 23 shows. The baud rate selected must be the same as that configured in the UART Component in the bootloader project (Figure 9 on page 8).

The COM ports in your computer are listed in the Ports (COM and LPT) category in the Device Manager. If you are using a USB-to-UART Bridge, they appear in this category after enumeration. The COM port number is displayed in brackets after the COM port name.

Figure 23. Bootloader Host Application



3. Choose the appropriate bootloadable file in the bootloadable project's Debug/Release folder:

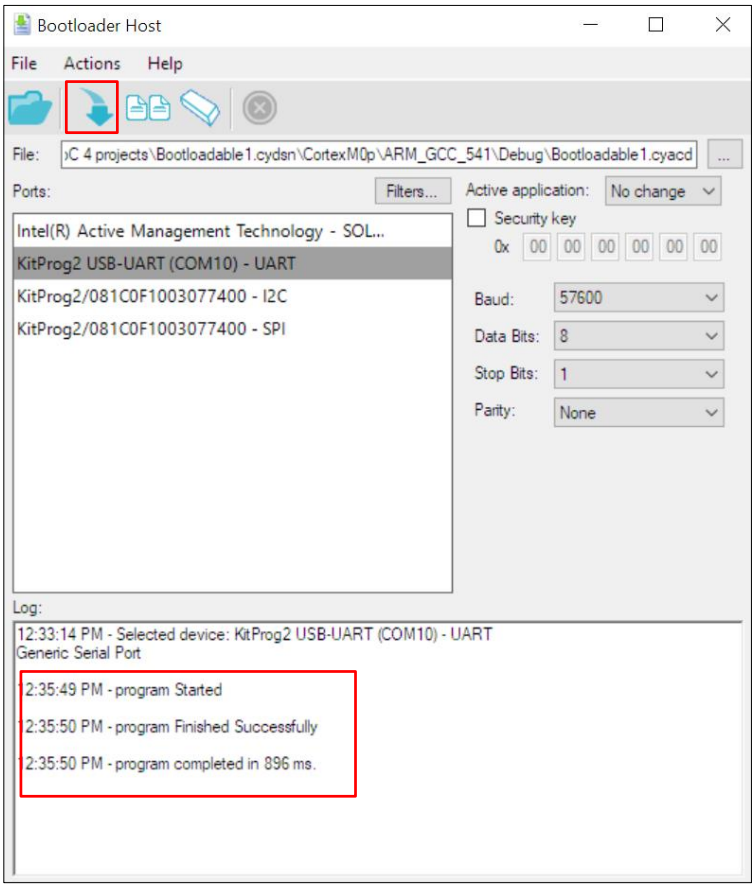
   When PSoC 3 is the bootloader,
   ...\Bootloadable1.cydsn\DP8051\DP8051_Keil_951\Debug\Bootloadable1.cyacd

   When PSoC 5LP is the bootloader,
   ...\Bootloadable1.cydsn\CortexM3\ARM_GCC_493\Debug\Bootloadable1.cyacd

   When PSoC 4000S/4100S/4100 S Plus or 4100PS is the bootloader,
   ...\Bootloadable1.cydsn\CortexM0p\ARM_GCC_493\Debug\Bootloadable1.cyacd

   When PSoC 4200 is the bootloader,
   ...\Bootloadable1.cydsn\CortexM0\ARM_GCC_493\Debug\Bootloadable1.cyacd
4. Browse the *.cyacd* file and click the **Program** button to begin bootloading. The bootloading status will be displayed in the **Log** section, as Figure 24 shows.

Figure 24. Downloading Bootloadable Project



5. After a successful bootload operation, for PSoC 3 or PSoC 5LP the message "Hello" is displayed on the CY8CKIT-030/CY8CKIT-050 LCD. For PSoC 4, the green LED on the kit starts to blink.

   Note: For KitProg 2.16 needs a device reset after the successful bootload operation to pass the control to bootloaded application. This is a known issue and it will be resolved in higher version of KitProg.

6. To bootload again, press SW2 on the DVK. This makes the PSoC device enter the bootloader. Now repeat steps 3 to 5 to bootload again.

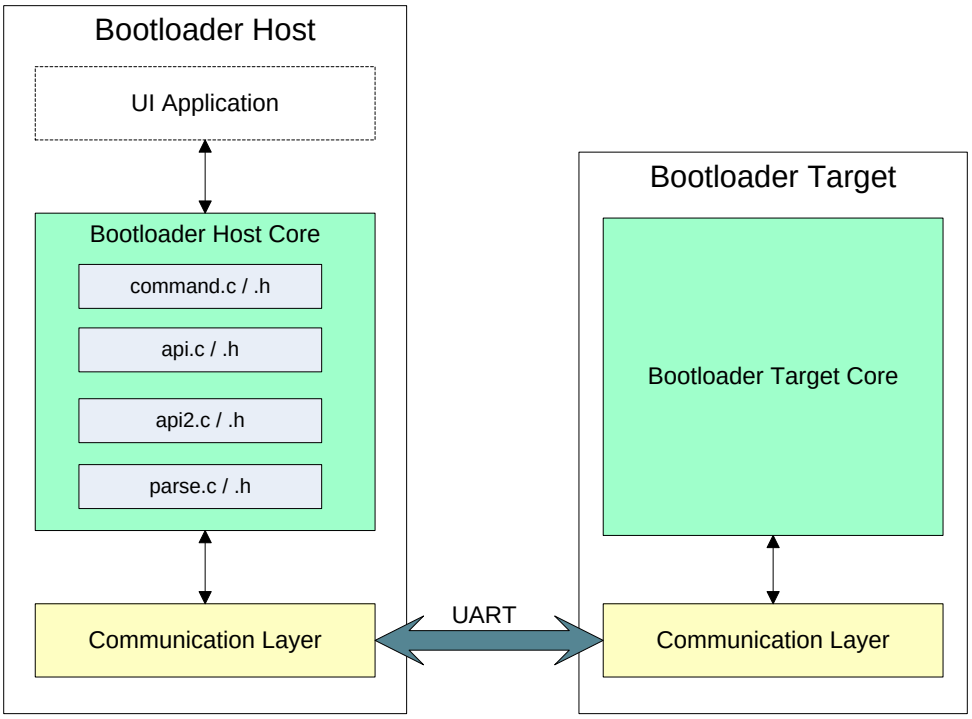
2.5 Bootloading Using an Embedded Host

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.

2.5.1 Bootloader Host Program

Figure 25 illustrates a protocol-level diagram of a bootloader system. The bootloader host and target each have two blocks: a core and a communication layer.

Figure 25. Protocol-Level Diagram of Bootloading



- The bootloader host core performs all bootloading operations—it sends command packets and flash data to the target. Based on the response from the target, it decides whether to continue bootloading.
- The bootloader 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. It contains communication protocol (UART)-–specific APIs to perform this function. This layer is responsible for sending and receiving protocol packets between the host and the target.

2.5.2 Steps to Create a UART Bootloader Host Project

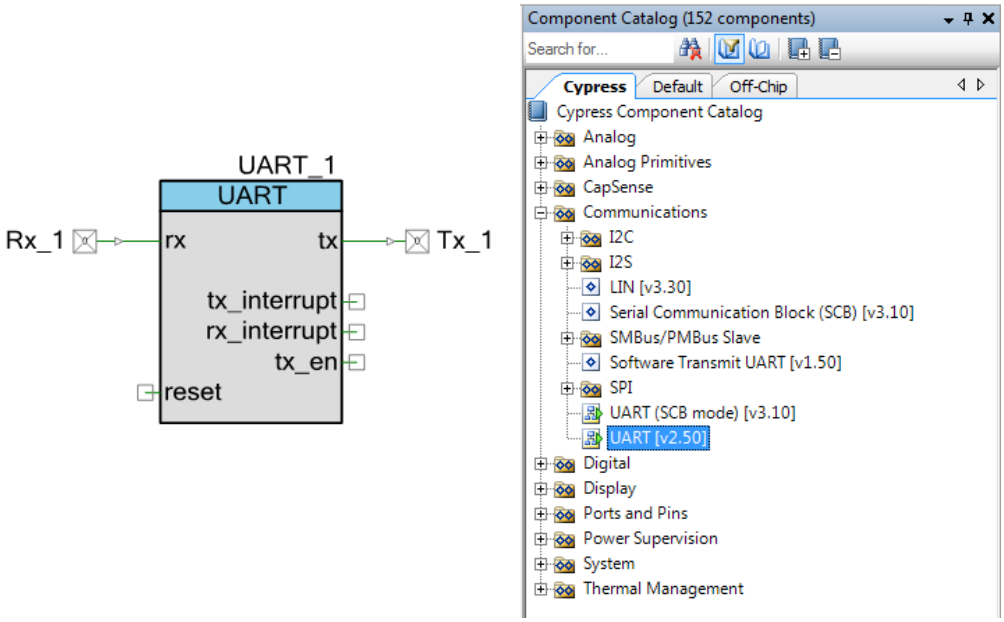
This section shows you how to create an embedded UART bootloader host project using PSoC 5LP, which can bootload another PSoC device. With this project, the host can bootload two different bootloadable files (*.cyacd* files) on alternate switch presses.

1. Create a new PSoC Creator project, select the PSoC 5LP as a target device, name the project "UART_Bootloader_Host," and create a new workspace for that project.

   Note: For PSoC Creator 3.1, the application type should be specified while creating project. To do son, click the "+" button next to the **Advanced** tab to expand the configuration options. Select **Normal** as the application type.

2. Add a UART Component to the TopDesign schematic, as Figure 26 shows. Also, add Digital Input Pin and Character LCD Components to the TopDesign.

Figure 26. UART Component



3. Rename the Components according to Table 6.

Table 6. Component List for UART Bootloader Host Project

Component	Name
UART_1	UART
Rx_1	Rx
Tx_1	Tx
Pin_1	Pin_Switch
LCD_Char_1	LCD_Char

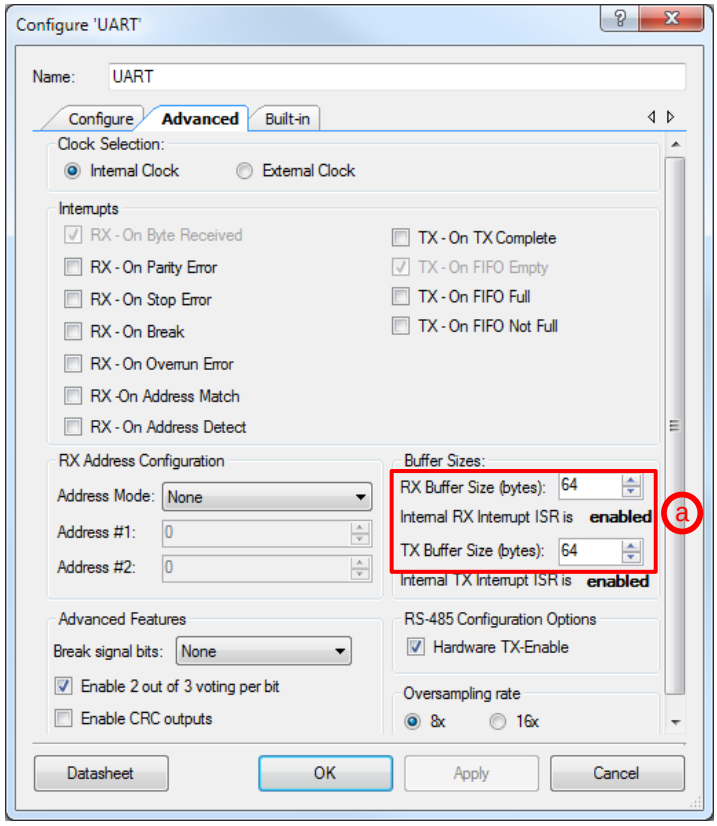
The next step is to configure these Components.

4. To configure the UART Component, double-click on it. By default, it is in Full UART mode with a data rate of 57,600 bps. Leave all parameters at their default settings.

   Note: The project can run at any supported data rate, but the data rate must be the same as that of the bootloader project.

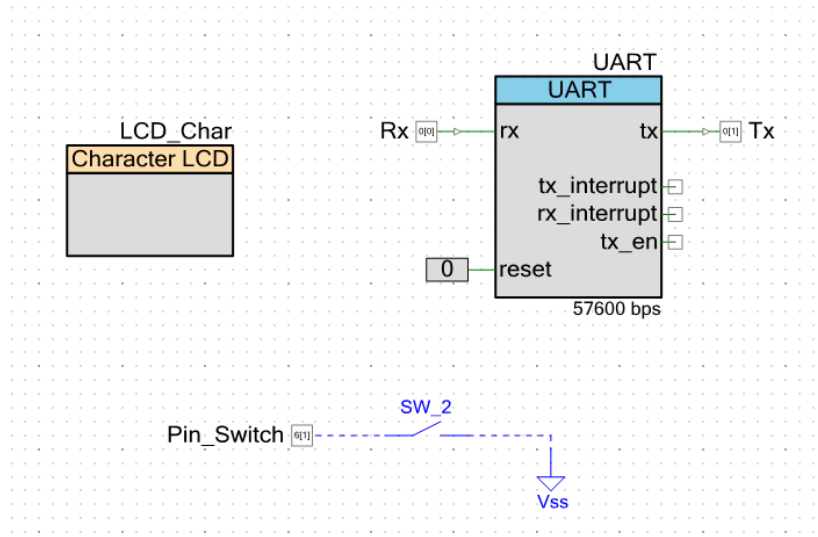
5. In the **Advanced** tab of the Component configuration window, set the transmit (Tx) and receive (Rx) buffer sizes to 64 to avoid any communication overflow (the host packet is as much as 64 bytes), as illustrated in Figure 27.

Figure 27. UART Advanced Configuration



6. The Digital Input Pin Pin_Switch is used to initiate the bootloading operation in the host. When the kit button is pressed, it shorts to ground, so you need to configure this pin to have a resistive pull-up.
7. With the addition of annotation Components to the button, the TopDesign of this project should be similar to Figure 28.

Figure 28. TopDesign of UART_Bootloader_Host project



8. Assign the input and output pins. In the **Workspace Explorer** window, double-click the file *UART_Bootloader_Host.cydwr* and assign the pins as Figure 29 shows.

Figure 29. UART_Bootloader_Host Project Pin Assignments

Name	Port
\LCD_Char:LCDPort[6:0]\	P2[6:0]
Pin_Switch	P6[1]
Rx	P0[0]
Tx	P0[1]

On the CY8CKIT-050 kit boards, the LCD pins are hardwired to P2[6:0], and SW2 is hardwired to P6[1]. Wire the kit board to connect the designated port pins (P4) to TX and RX (P5).

9. Add firmware to this project. The UART_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 status of Pin_Switch. When it is grounded, bootloading is initiated. The file *main.c* has a function called `BootloadStringImage()`, which is defined in *device.h*. This function bootloads the *.cyacd* file using the Bootloader Host API files (host core; see Figure 25 on page 21).

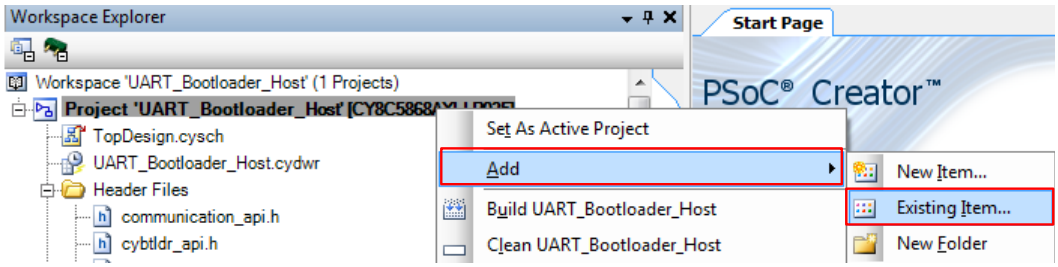
   The `main()` function has a variable called “toggle.” It alternates between '0' and '1' on every button press. This makes the host select alternate bootloadable files.

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

    <install folder> \ PSoC Creator \ 3.3 \ PSoC Creator \ cybootloaderutils

    To include these files, go to the **Workspace Explorer** window, right-click on the project name, and choose **Add** > **Existing Item**, as Figure 30 shows. Add the following files provided by PSoC Creator: *cybtldr_api.c /.h*, *cybtldr_command.c /.h*, *cybtldr_parse.c /.h*, and *cybtldr_utils.h*.

Figure 30. Adding API Files



11. In addition to the bootloading API files, the host also requires communication layer support. This support is provided by adding the *communication_api.c /.h* files. You can include the contents of these files from the UART_Bootloader_Host 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.

    A. 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 you have two bootloadable files, you 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).

    B. Refer to the *StringImage.h* file in the UART_Bootloader_Host project associated with this application note.

    Alternatively, you can use the Windows C# application provided along with this application note to generate the *StringImage.h* file.

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

3 Testing the Projects

The *main.c* file of the UART_Bootloader_Host project has a macro called “TARGET_DEVICE.” This macro is used to choose the target device among PSoC 3, PSoC 4, and PSoC 5LP. By default, it is defined as “PSoC_3” (another macro in the same file). If you are using PSoC 4 or PSoC 5LP as your target device, change the definition of this macro to “PSoC_4” or “PSoC_5LP” respectively.

3.1 Kit Configuration

To test the projects, configure the kits as follows:

For CY8CKIT-030:

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

For PSoC 4-based kits:

1. Program PSoC 4 with the UART_Bootloader project.
2. Set the jumper to 5 V.

For CY8CKIT-050:

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

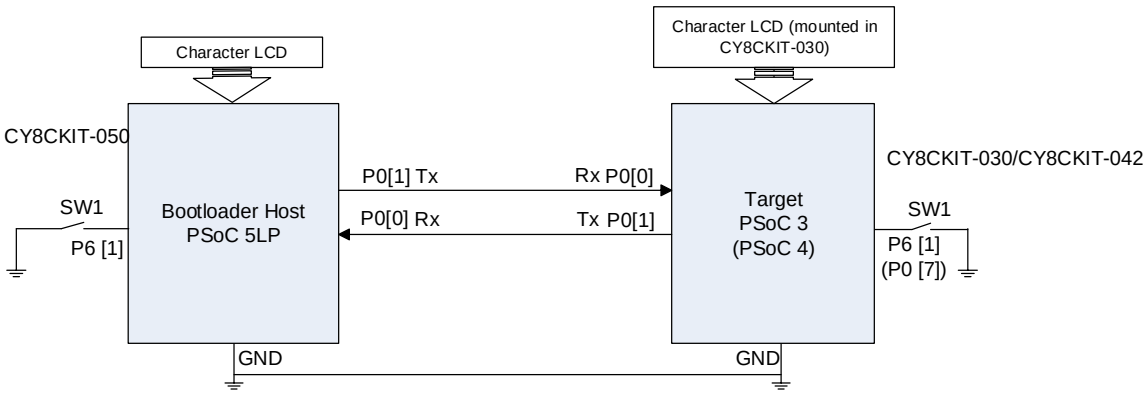
Make the following connections between the two DVKs. This example considers the CY8CKIT-030 and CY8CKIT-042 kits to be the target.

1. Connect P0 [0] of CY8CKIT-030 (CY8CKIT-042) to P0 [1] of CY8CKIT-050.
2. Connect P0 [1] of CY8CKIT-030 (CY8CKIT-042) to P0 [0] of CY8CKIT-050.
3. Short together the ground pins of the kits.

The connections are illustrated in Figure 31. The connection may change based on the UART pins selected in the bootloader project.

Note: In Figure 31, the target can also be a PSoC 5LP device (CY8CKIT-050), in which case, the pin connections are same as that for CY8CKIT-030.

Figure 31. Host/Target Connections



3.2 Bootloading PSoC 3

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

- On the first button press (P6 [1]) on CY8CKIT-050, the *Bootloadable1.cyacd* file is bootloaded to the target PSoC 3 device. 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 CY8CKIT-030. This makes the PSoC 3 device enter the bootloader and be ready to bootload a new application. The LED starts blinking.
- On the next button press on CY8CKIT-050, the *Bootloadable_2.cyacd* file is bootloaded to the target PSoC 3 device. 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.

3.3 Bootloading PSoC 4

- In the *main.c* file of the UART_Bootloader_Host project, change the TARGET_DEVICE macro to PSoC_4. Rebuild the project and program it into the PSoC 5LP device on the CY8CKIT-050.
- After the first button press (P6 [1]) on CY8CKIT-050, the *Bootloadable1.cyacd* file is bootloaded to the target PSoC 4. On successful completion, the green LED on the PSoC 4 kit starts to blink.
- For subsequent bootloading operations, press the button Pin_Startbootloader. This makes the PSoC 4 enter the bootloader and be ready to bootload a new application. The red LED on the PSoC 4 kit starts to blink.
- After the next button press on CY8CKIT-050, the *Bootloadable2.cyacd* file is bootloaded to the target PSoC 4. On successful completion, the blue LED on the PSoC 4 kit starts to blink.

4 Summary

This application note explained how to bootload PSoC 3, PSoC 4, and PSoC 5LP using UART as the communication interface. It also introduced the basic building blocks of a bootloader host and showed how to build an embedded UART 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 Memory sections and the PSoC 3, PSoC 4, and PSoC 5LP Technical Reference Manuals.

5 Related Application Notes

Refer to the following application notes to learn more about bootloaders and flash programming.

- AN73854 – PSoC 3, PSoC 4, and PSoC 5LP Introduction to Bootloaders
- AN60317 – PSoC 3 and PSoC 5LP I2C Bootloader
- AN73503 – USB HID Bootloader for PSoC 3 and PSoC 5LP
- AN84401 – PSoC 3 and PSoC 5LP SPI Bootloader
- AN86526 – PSoC 4 I2C Bootloader
- 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
- AN79953 – Getting Started with PSoC 4
- 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 shown in Table 7.

Table 7. Projects Associated with This Application Note

Design Project Name	Description
UART_Bootloader_Host	This is an embedded bootloader host project demonstrating a PSoC 5LP bootloading a PSoC 3, PSoC 4, or PSoC 5LP with UART as the communication channel.
UART_Bootloader	This bootloader project has UART as the communication channel. The bootloader blinks an LED.
Bootloadable1	For PSoC 3/PSoC 5LP, this project displays “Hello” on the character LCD of the target device. For PSoC 4, this project blinks the green LED on the target kit.
Bootloadable2	For PSoC 3/PSoC 5LP, this project displays “Bye” on the character LCD of the target device. For PSoC 4, this project blinks the blue LED on the target kit.

About the Authors

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

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 32 shows the physical organization of the flash memory in PSoC 3, PSoC 4, and PSoC 5LP.

PSoC flash is divided into blocks called arrays. Arrays are uniquely identified by array IDs. In PSoC 3 and PSoC 5LP, 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.

In PSoC 4, each array has 128 or 256 rows of flash memory. Each row has 128 data bytes. So, an array can have 16 KB or 32 KB for instruction and data storage.

The number of flash arrays depends on the device and the part. PSoC 3 and PSoC 4100S have a maximum flash of 64 KB, so they have only one array, and the only valid array ID is 0. PSoC 4100S Plus devices have maximum flash of 128 KB, and the only valid array ID is 0. PSoC 4100/4200, PSoC 4100PS, and PSoC 4000S devices have a maximum flash of 32 KB, so they have 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 32 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 4 and 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 4 and 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 (for PSoC 4, it is the first 128-byte boundary). This region of the flash includes:

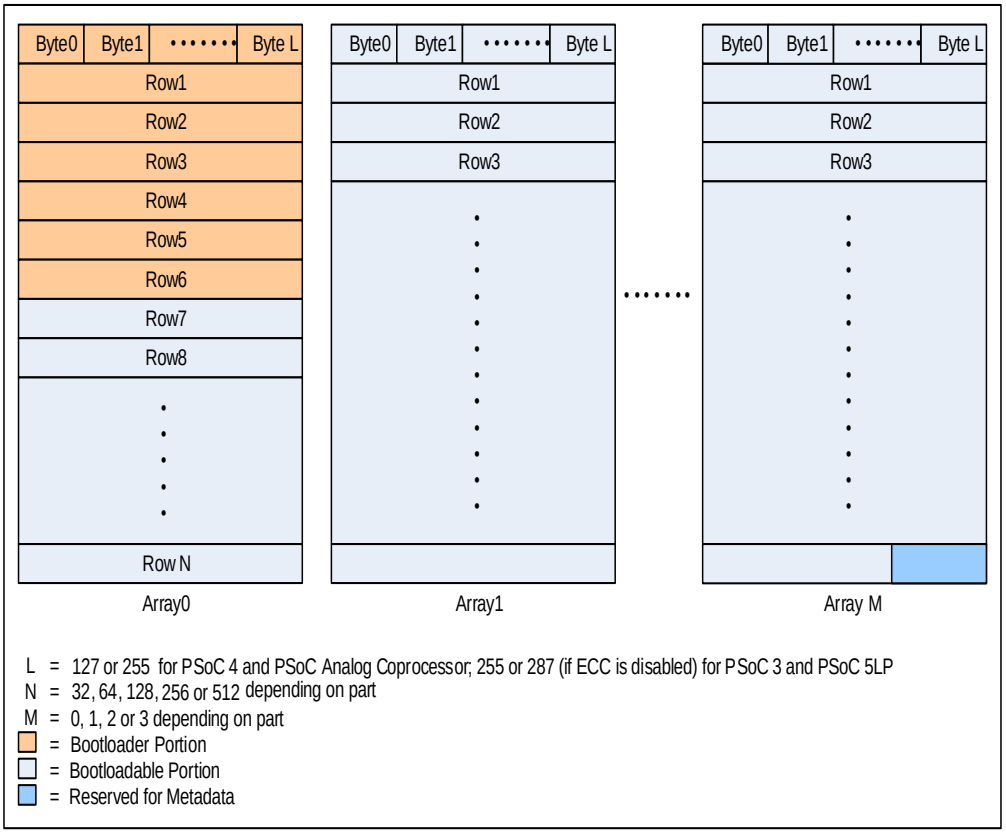
  - Vector table for the bootloadable project (PSoC 4 and PSoC 5LP only)
  - The bootloadable project code and data

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 (1 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, see Metadata Layout in Flash.

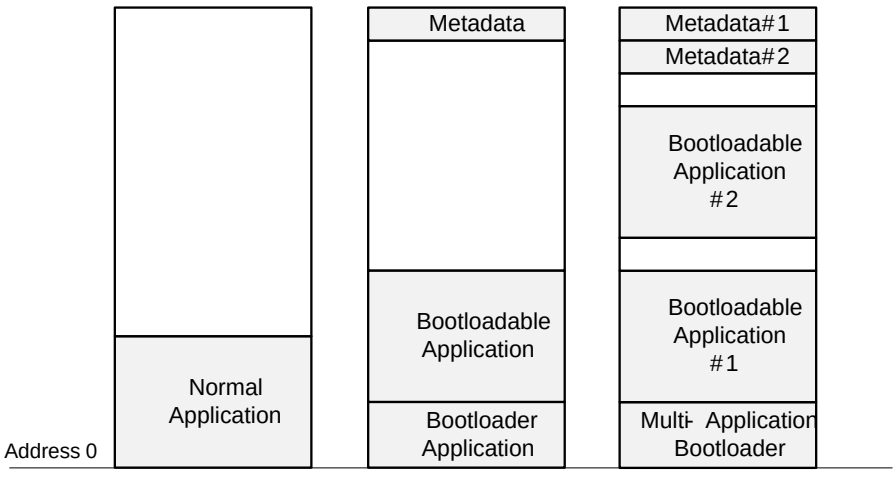
Figure 32. Physical Organization of Flash Memory in PSoC



Memory Usage in PSoC

There are two types of bootloader projects: standard and multi-application. The multi-application bootloader is 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 33 shows the flash memory usage for each type of PSoC Creator project.

Figure 33. Flash Memory Usage



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 33 shows. Various parameters, depending on the device used, are stored in this block, as Table 8 shows. For the multi-application bootloader, there are two sets of metadata.

Table 8. Metadata Layout

Address	PSoC 3	PSoC 4 / PSoC 5LP
0x00	App Checksum	App Checksum
0x01	Reserved	Application Address
0x02		
0x03	Application Address	
0x04		
0x05	NA	Last Bootloader Row
0x06	NA	
0x07	Last Bootloader Row	
0x08		
0x09		Application Length
0x0A		
0x0B	Application Length	
0x0C		
0x0D		NA
0x0E		NA
0x0F	NA	NA
0x10	Application Active	Application Active
0x11	Application Verified	Application Verified
0x12	Bootloader Application Version	Bootloader Application Version
0x13	Bootloadable Application ID	Bootloadable Application ID
0x14		
0x15	Bootloadable Application Version	Bootloadable Application Version
0x16		
0x17	Bootloadable Application Custom ID	Bootloadable Application Custom ID
0x18		
0x19-0x3F	NA	NA

Note: For the multi-application bootloader, Last Bootloader Row for metadata (image 2) signifies the last row of bootloadable 1 in 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.

PSoC 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 9 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 9. 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

Note: PSoC 4 has only two levels of flash protection: Unprotected and full protection.

After 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 the bootloader code is to completely erase the flash and reprogram it using the JTAG/SWD interface.

An example for protecting bootloader flash follows.

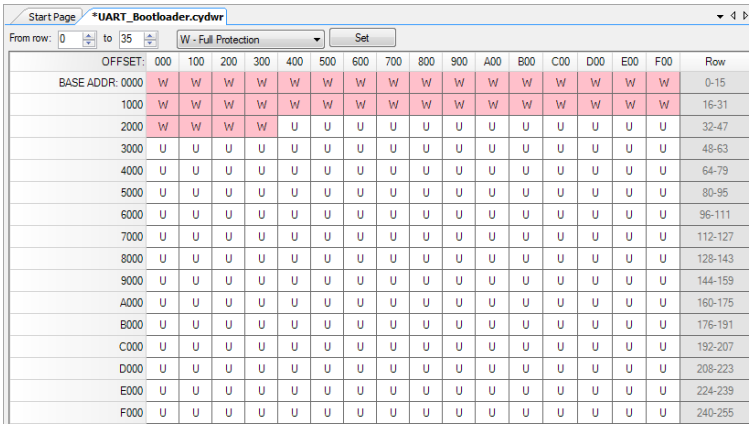
Example of Flash Protection

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 UART_Bootloader project is 9250 bytes, then the output is (for PSoC 3 with 64 KB flash):

```
Flash used: 9250 of 65536 bytes (14.11 %).
```

The bootloader thus occupies 37 rows of flash (9250/256), that is, flash locations 0x0000 to 0x2300. Set the flash protection level as full protection for these rows (in 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 34 shows. For more information on how to use the flash protection dialog, see the PSoC Creator help article “Flash Security Editor.”

Figure 34. Flash Protection in PSoC Creator



Nonvolatile Latch Settings

Nonvolatile latch (NVL) settings 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 the bootloadable project are different from the 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 device while programming using the JTAG/SWD interface.

For a bootloadable project, the *.hex* 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. Therefore, 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]
```

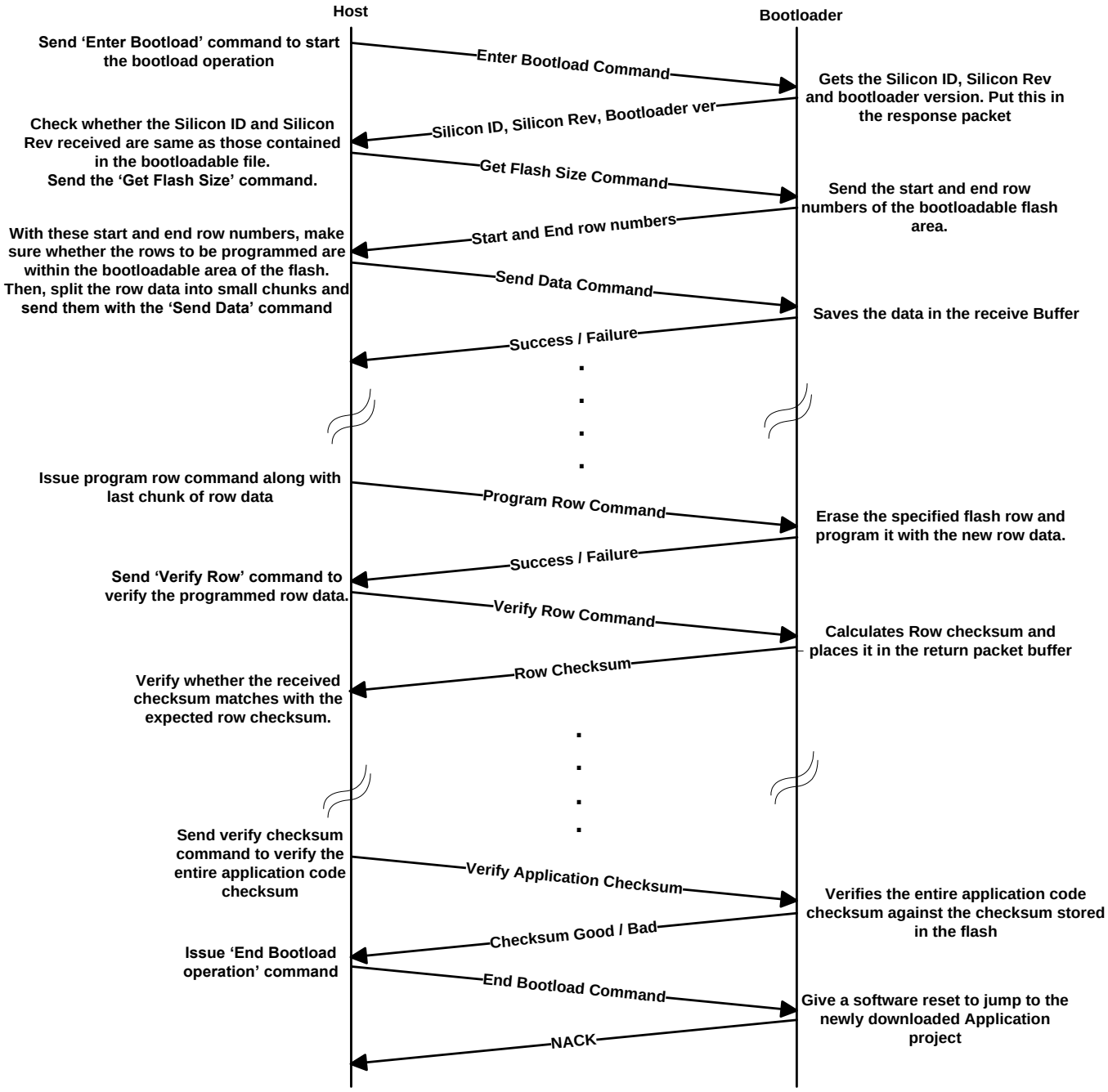
The checksum type in the header indicates the type of checksum used in the packets sent between the bootloader and the bootloader host during the bootloading operation. If this byte is 0, the checksum is a basic summation. If it is 1, the checksum is CRC-16.

Appendix C. Host/Target Communications

Communication Flow

The Bootloader Function Flow section looked at the operation of a bootloader in PSoC, and Bootloading using embedded host introduced the building blocks of a bootloader host. With this background, Figure 35 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.

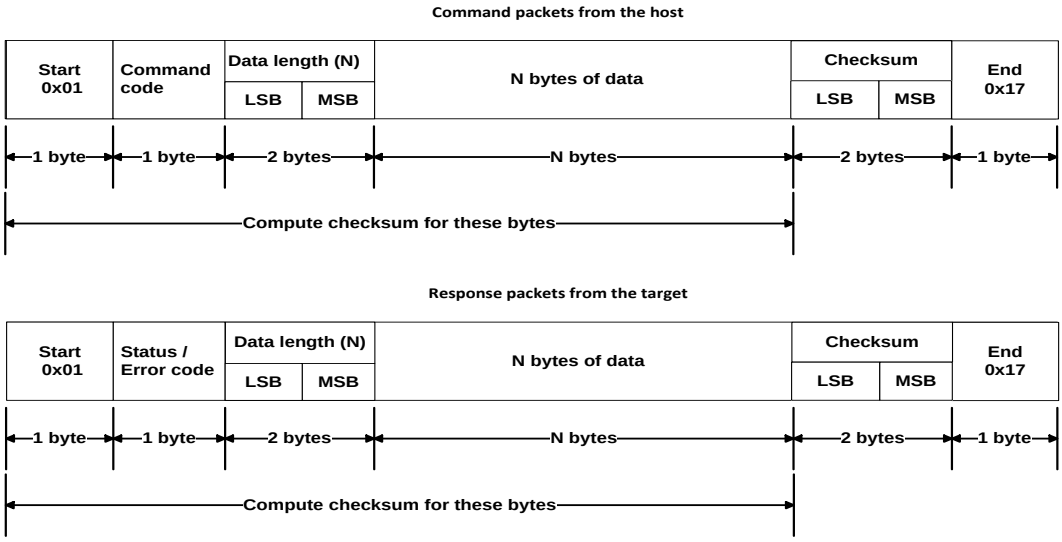
Figure 35. Communication Flow During Bootloading



Protocol Packet Format

The bootloading operation involves the exchange of command and response packets between the host and the target. These packets have specific formats, as Figure 36 shows.

Figure 36. Bootloading Packet Format



Each packet includes checksum bytes. The checksum can be a basic summation (2's complement) or CRC-16, depending on the bootloader project setting. When sending multibyte data such as DataLength and Checksum, the 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 a status/error code instead of the command code. The important commands and data bytes and the bootloader response packet data are given in Table 10.

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 10 provides a list of commands and their expected responses. Table 11 provides a list of status and error codes.

Table 10. 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 the requested array ID.
0x33	Get application status (valid only for multi-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 that 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 multi-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 the 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 11. Bootloading Status / Error Codes – Possible Responses to Commands

Status/ Error Code	Label	Description
0x00	CYRET_SUCCESS	The command was successfully received and executed.
0x02	BOOTLOADER_ERR_VERIFY	The verification of flash failed.
0x03	BOOTLOADER_ERR_LENGTH	The amount of data available is outside the expected range.
0x04	BOOTLOADER_ERR_DATA	The data is not in the proper form.
0x05	BOOTLOADER_ERR_CMD	The command is not recognized.
0x06	BOOTLOADER_ERR_DEVICE	The expected device does not match the detected device.
0x07	BOOTLOADER_ERR_VERSION	The bootloader version detected is not supported.
0x08	BOOTLOADER_ERR_CHECKSUM	The checksum does not match the expected value.
0x09	BOOTLOADER_ERR_ARRAY	The flash array ID is not valid.
0x0A	BOOTLOADER_ERR_ROW	The flash row number is not valid.
0x0C	BOOTLOADER_ERR_APP	The application is not valid and cannot be set as active.
0x0D	BOOTLOADER_ERR_ACTIVE	The application is currently marked as active.
0x0F	BOOTLOADER_ERR_UNK	An unknown error occurred.

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 low-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 12 describes in detail the functions of this API file.

Table 12. Functions of cybtldr_api.c /.h

Function	Description
CyBtldr_StartBootloadOperation	▪ This function enables the communication interface and sends an enter bootloader command to the target. ▪ From the response packet received, it verifies the silicon ID, silicon revision of the target device, and bootloader version.
CyBtldr_ProgramRow	▪ This function 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, 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, the function sends a program row command to the target.
CyBtldr_VerifyRow	▪ This function first validates a row for a particular array ID and row number. ▪ If row validation is successful, it sends a verify row command for the validated flash row. In response, the target returns the checksum of the row. ▪ The returned checksum is verified against the expected checksum value.
CyBtldr_EraseRow	▪ This function first validates a row for a particular array ID and row number. ▪ If row validation is successful, it sends an erase row command for the validated flash row.
CyBtldr_EndBootloadOperation	This function 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 API 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 previously, 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 Device Reset Is Needed

To understand why a 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, the device configuration can involve the setting of thousands of PSoC registers. This is especially true for the PSoC 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 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 13 shows.

Table 13. 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.
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	NA		Eight possible drive modes. See device datasheet for details.
Code reaches main()	Code may change I/O pin function.	NA		

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

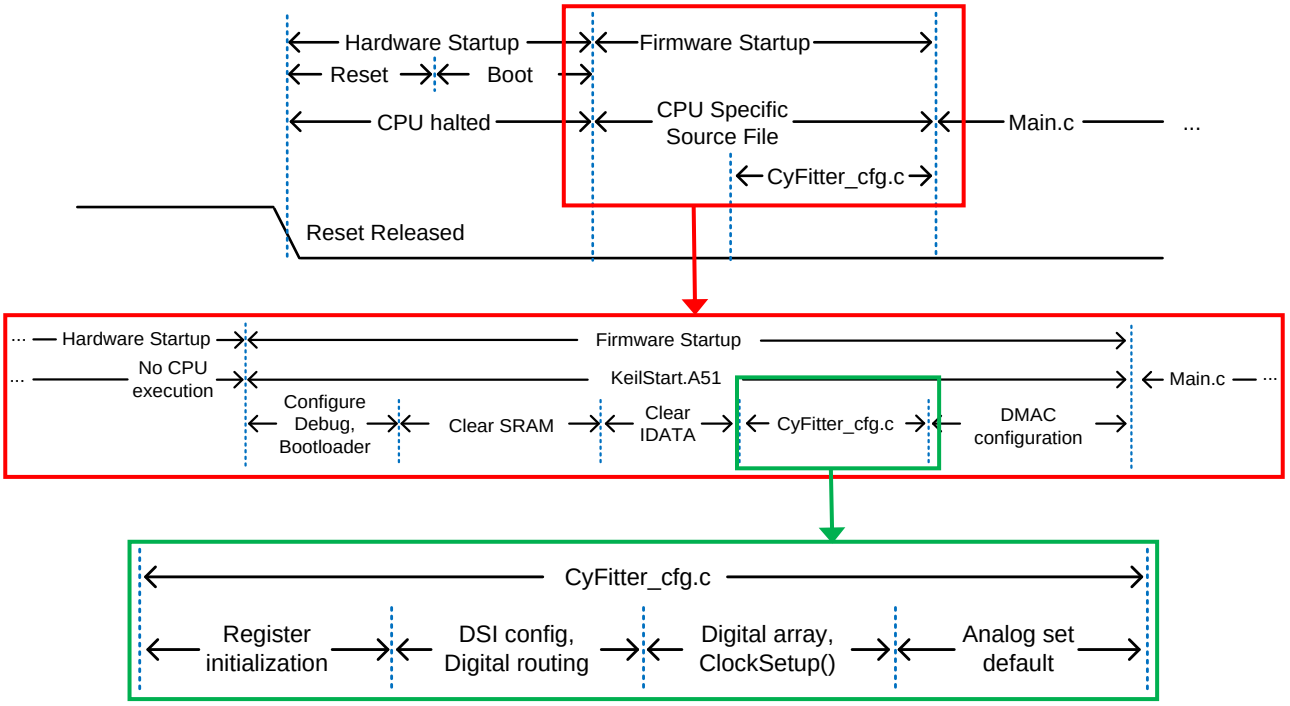
- The **Reset** tab for I/O ports, the individual Pin Component configurations
- The **System** tab for all other NVLs, the Design-Wide Resources (DWR) window

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 37 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 37. Device Startup Process Diagrams



Effect on Other Functions

At device reset, universal digital block (UDB) registers are reset, so all UDB-based Components cease to exist and their functions are stopped. The same holds true for analog Components based on the configurable SC/CT blocks and universal analog block (UAB)–based components.

All fixed peripherals, both digital and analog, are reset to their idle states. This includes the DMA, DFB, timers (TCPWM), I^{2}C, USB, CAN, ADCs, DACs, comparators, and opamps. All clocks are stopped except the internal main oscillator (IMO).

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

All hardware-based functions are restored after configuration (see Figure 37). All firmware functions are restored when the project's `main()` function starts executing.

Example: Fan Control

This section examines 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 the 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 can 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 14 shows.

Table 14. 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 a 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 a 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 a 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. The 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 (HSSP). The ISSP specifications to create HSSP are given in AN62391 (PSoC 3).

What Happens When Power Fails During the Bootload Operation

If power fails during the bootload operation, 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 restart the bootload operations.

Why You Need a Reset to Jump Between the Bootloader and Bootloadable Projects

PSoC is an enormously configurable device. 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 Appendix E – Bootloader and Device Reset.

Converting a Normal Application Project to a Bootloadable Project

If you have already created a standard (normal) project and want to convert it to a bootloadable project, add a Bootloadable Component to the TopDesign and add the bootloader project's *.hex* file as a dependency as Figure 14 on page 13 shows.

If a project is created as a normal project and then later changed to a bootloader project by adding a Bootloader Component to the TopDesign, you need to insert the `Bootloader_Start()` function call in *main.c* for the bootloader project to work as expected.

Note: For PSoC Creator 3.1, if you want to convert a standard normal) project to a bootloadable/bootloader project, change the application type of the project to bootloadable/bootloader. To do so, right-click on the project, choose **Build** > **Code Generation** > **General,** and change the application type in addition to adding the Bootloadable/Bootloader Component to the TopDesign.

Debugging Bootloadable Projects

In the PSoC Creator bootloader system, the bootloader project executes first (at device reset) followed by 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 the application type to normal, debug it, and then convert it back to bootloadable after debugging is done.

Another option is to program the bootloadable project *.hex* file into the device and then use the ”Attach to running 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

A 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 the bootloader commands. This functionality comes with the obvious limitation that each application has one-half of the available flash memory. Figure 33 on page 29 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. Create a new MABL bootloader project. Select the Multi-App bootloader checkbox in the Bootloader configuration window.

   Note: For PSoC Creator 3.1, set the application type as Multi-App Bootloader.
2. Add two bootloadable projects to the workspace, for example, Project_A and Project_B. For each project, add a dependency to the MABL project. Two *.cyacd* files are generated for each project: one for the lower part of flash and one for the upper part of flash:

- *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 the *.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 to the *CyBtldr_Command.c* / *.h* files.

Memory Requirement for Bootloader

A typical UART 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 4.6 KB of PSoC 4 flash with the GCC compiler optimization set to “size.” And it occupies approximately 5.4 KB of PSoC 5LP flash with the 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. The 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 38 shows.

Set the **Device Configuration Mode** to **Compressed** in the *.cydwr* **System** tab, as Figure 39 shows, to minimize flash memory usage. Set **Device Configuration Mode** to **DMA** if startup time is more important than code size.

Figure 38. Unchecking Optional Commands in Bootloader Component

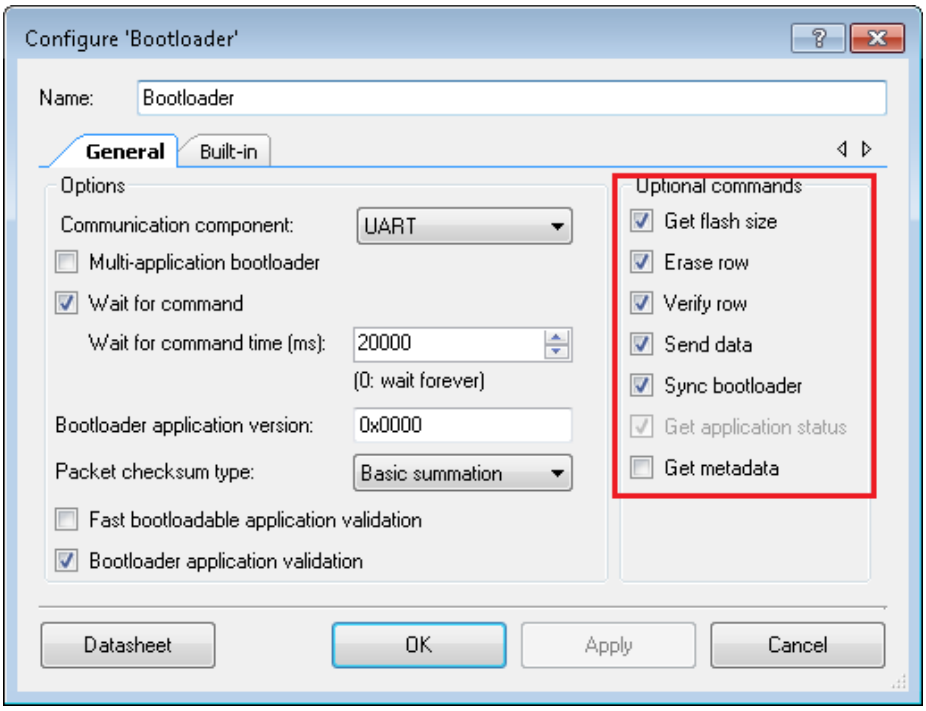
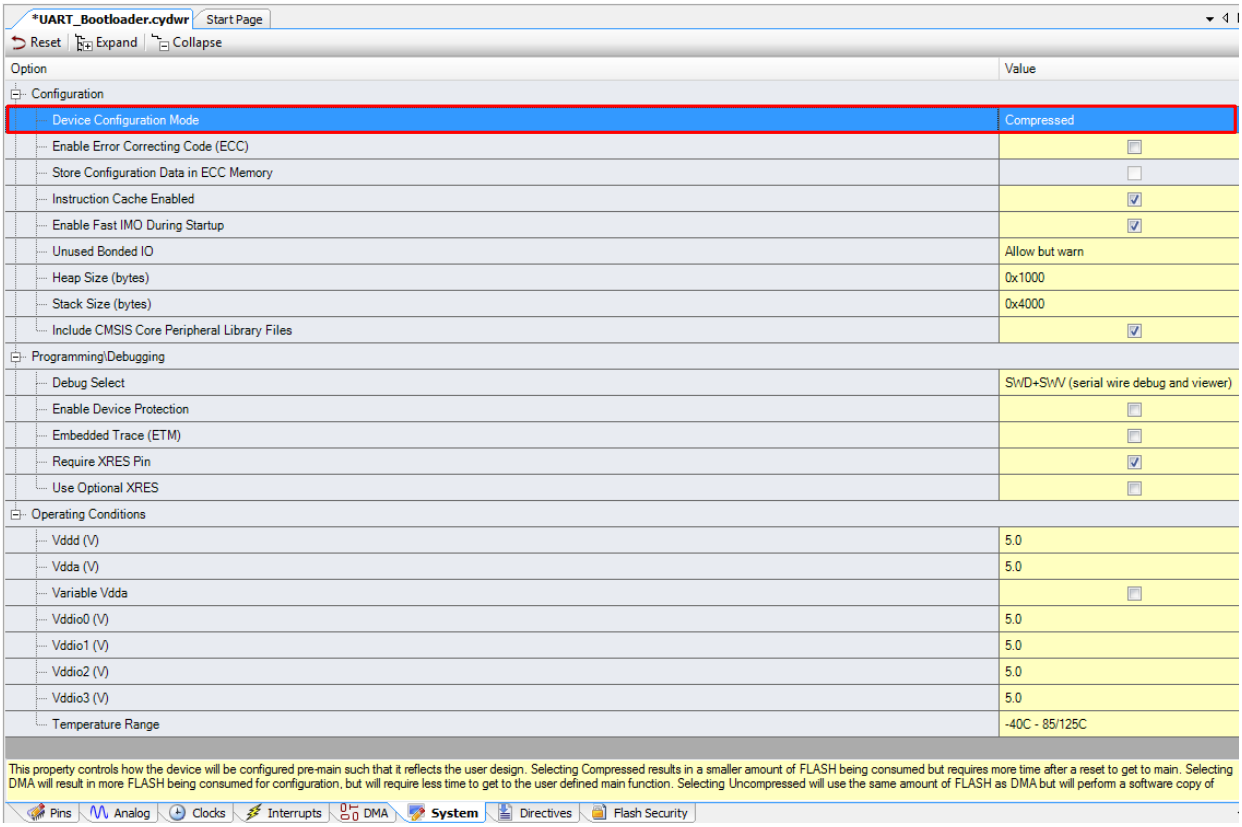


Figure 39. Device Configuration Mode



Appendix G. Kit Selection

Table 15. Kit Selection Based on Target Device

Target Device	Kit Name	User Guide
CY8C42xx	CY8CKIT-042 PSoC 4 Pioneer Kit	CY8CKIT-042
CY8C41xx-PS	CY8CKIT-147 PSoC 4100PS Prototyping Kit	CY8CKIT-147
CY8C40xx-S	CY8CKIT-041-40xx 4 S-Series Pioneer Kit	CY8CKIT-041
CY8C41xx-S	CY8CKIT-041-41xx 4 S-Series Pioneer Kit	CY8CKIT-041
CY8C41xx-S	CY8CKIT-149 PSoC 4100S Plus Prototyping Kit	CY8CKIT-149
CY8C3xxx	CY8CKIT-030 PSoC 3 Development Kit	CY8CKIT-030
CY8C5xxx	CY8CKIT-050 PSoC 5 Development Kit	CY8CKIT-050

Document History

Document Title: AN68272 - PSoC® 3, PSoC 4, and PSoC 5LP UART Bootloader

Document Number: 001-68272

Revision	ECN	Orig. of Change	Submission Date	Description of Change
**	3208805	ANMD	03/27/2011	New application note.
*A	3471679	ANMD	12/22/2011	Revised AN with UART Bootloader Example Updated template
*B	3623001	ANMD	05/22/2012	Updated figures 9 and 10, and tables 2 and 3. Minor text edits. Added Appendix C. Updated template and project files.
*C	3671377	ANMD	07/11/2012	Updated the Application Note project and document to PSoC Creator 2.1.
*D	3811899	PHAL	11/26/2012	Updated for PSoC 5LP
*E	3895950	PHAL	03/19/2013	Updated project files Sunset review
*F	4078736	SRYP	07/31/2013	Updated to align with SPI and I2C bootloader application notes. Added a UART bootloader host project. Added support for PSoC 4.
*G	4339454	RNJT	04/10/2014	Updated for PSoC Creator 3.0 SP1
*H	4435010	MKEA	07/17/2014	Added Appendix E – Bootloader and Device Reset
*I	4678368	VAIR	03/17/2015	Updated for PSoC Creator 3.1 CP1 Updated the screenshots of PC bootloader application Sunset update
*J	5152725	AKSM	03/03/2016	Updated for PSoC Analog Coprocessor, PSoC 4000S/4100S Updated images for PSoC Creator 3.3 SP2 Added Kit Selection
*K	5767773	AESATP12	06/14/2017	Updated logo and copyright.
*L	5959591	TAVA	11/08/2017	Updated for PSoC 4100S Plus devices Updated for PSoC Creator version from 3.3 to 4.2
*M	6085168	DIMA	03/12/2018	Updated for PSoC 4100PS devices Added *readme.txt* to instruct using the bootloader host GUI source code for 64-bit systems Updated template
*N	6175130	NIDH	05/15/2018	Removed obsolete Bootloader Host tool from the AN – added PSoC Creator based tool information instead. Updated projects to PSoC Creator 4.2. Minor edits through the document.
*O	6190743	NIDH	05/30/2018	Updated abstract

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 MCU

Cypress Developer Community

Community | Projects | Videos | Blogs | Training | Components

Technical Support

cypress.com/support

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