

## PSoC<sup>®</sup> 3, PSoC 4, and PSoC 5LP UART Bootloader

**Authors:** Anu M D, Siddalinga Reddy

**Associated Project:** Yes

**Associated Part Family:** CY8C3xxx, CY8C42xx, CY8C5xxx

**Software Version:** PSoC<sup>®</sup> Creator™ 3.1 CP1 and higher

**Related Application Notes:** [click here](#)

To get the latest version of this application note, or the associated project file, please visit <http://www.cypress.com/go/AN68272>.

AN68272 describes a UART-based bootloader for PSoC<sup>®</sup> 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 and a C#-based bootloader application.

### Contents

Introduction .....	1
Terms and Definitions .....	2
Using a Bootloader .....	2
Bootloader Function Flow .....	2
Techniques to Enter Bootloader .....	3
Projects .....	4
UART Bootloader .....	4
PSoC 3 and PSoC 5LP Bootloadables .....	7
PSoC 4 Bootloadables .....	9
Bootloading Using a PC Host .....	11
Bootloading Using an Embedded Host .....	12
Testing the Projects .....	15
Kit Configuration .....	15
Bootloading PSoC 3 .....	15
Bootloading PSoC 4 .....	15
Summary .....	16
Related Application Notes .....	16
Related Projects .....	16
Appendix A – Memory .....	17
Appendix B – Project Files .....	20
Appendix C – Host / Target Communications .....	21
Appendix D – Host Core APIs .....	24
Appendix E – Bootloader and Device Reset .....	25
Appendix F – Miscellaneous Topics .....	28
Appendix G – C# Bootloader Host Application .....	31
Worldwide Sales and Design Support .....	36

### 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 the ARM<sup>®</sup> 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<sup>2</sup>C, UART, or SPI. A bootloader communicates with a host to get new application code or data, and writes it into the device's flash memory.

In this application note you will learn:

- How to 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
  - How to create a PC bootloader application

This application note assumes that you are familiar with PSoC and the PSoC Creator 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](#), or [AN77759 - Getting Started with PSoC 5LP](#) respectively. If you are new to PSoC Creator, see the [PSoC Creator home page](#).

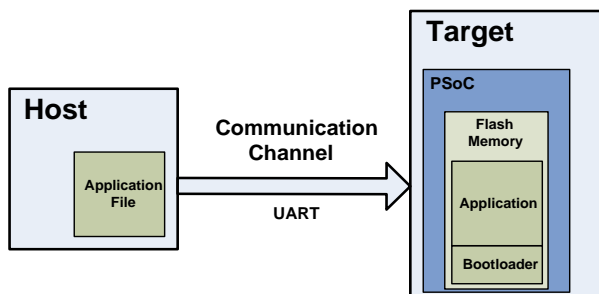
This application note also assumes that you are familiar with bootloader concepts. If you are new to these concepts, see [AN73854 - PSoC 3 and PSoC 5LP Introduction to Bootloaders](#). For a complete list of other application notes on bootloading, see [Related Application Notes](#).

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.

## Terms and Definitions

[Figure 1](#) illustrates the main elements in a bootloader system. It shows that the product's embedded firmware must be able to use the communication port for two different purposes – normal operation and updating flash. That portion of the embedded firmware that knows how to update the flash is called the **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 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 different function 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 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.

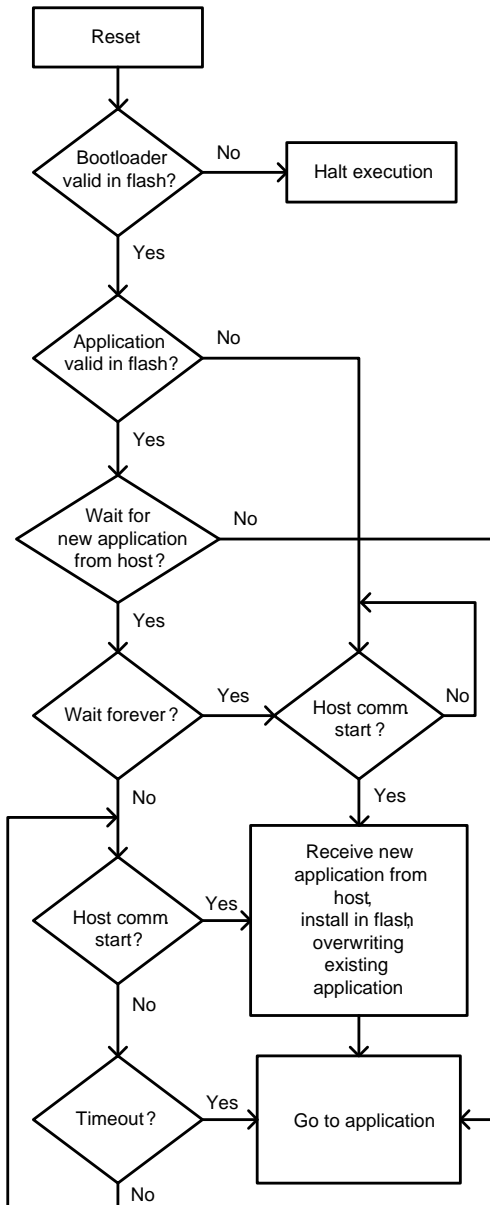
## Bootloader Function Flow

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 exists to start bootloading, and that 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) 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?

### 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.

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 and PSoC 5LP Introduction to Bootloaders](#).

## Projects

Now, let us look at specific examples of how bootloaders are implemented in PSoC and PSoC Creator.

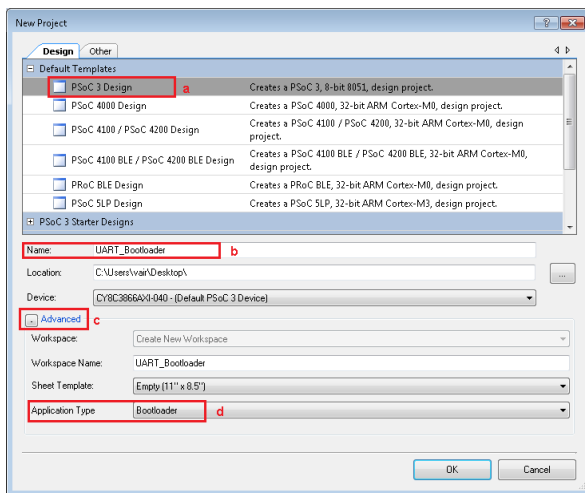
This section shows you the steps to create PSoC Creator bootloader, bootloadable and embedded bootloader host projects. The projects are designed to be used with the [CY8CKIT-030](#), [CY8CKIT-042](#), and [CY8CKIT-050](#) kits; they can be easily adapted for other kits such as the [CY8CKIT-001](#). The projects also require PSoC Creator version 3.1 SP1 or higher.

### UART Bootloader

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

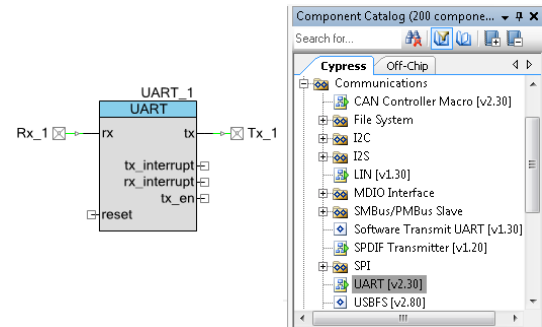
1. Create a new PSoC Creator project, as [Figure 3](#) shows.
  - a) Select the design template to be either 'PSoC 3 Design', or 'PSoC 5LP Design', or one of the PSoC 4 designs.
  - b) Name the project as UART\_Bootloader.
  - c) Click on the '+' button next to Advanced to expand the configuration options. Create a new workspace for the project.
  - d) Select 'Bootloader' as the application type.

Figure 3. Creating UART\_Bootloader Project



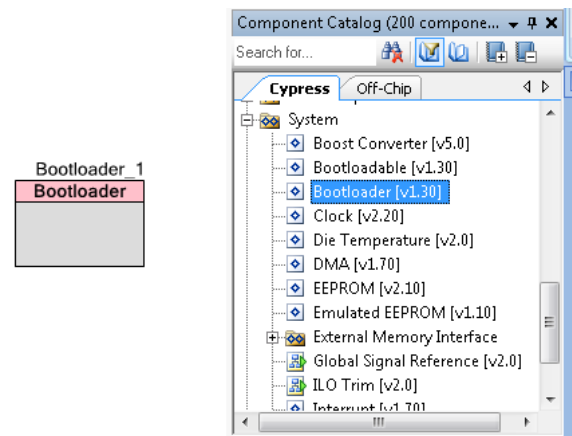
2. Add a UART Component to the top design schematic as [Figure 4](#) shows.

Figure 4. UART Component



3. Add a Bootloader Component to the top design schematic as [Figure 5](#) shows.

Figure 5. Bootloader Component



4. To blink an LED, add a PWM (TCPWM in the case of 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. Now, with an LED and resistor added as annotation Components, the top design of the project for PSoc 3 and PSoc 5LP looks similar to [Figure 6](#) and the top design for PSoc 4 looks similar to [Figure 7](#).

Figure 6. Top Design of UART\_Bootloader Project for PSoc 3 and PSoc 5LP

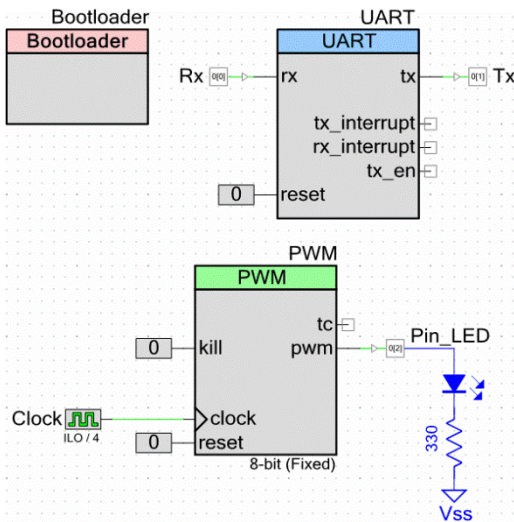
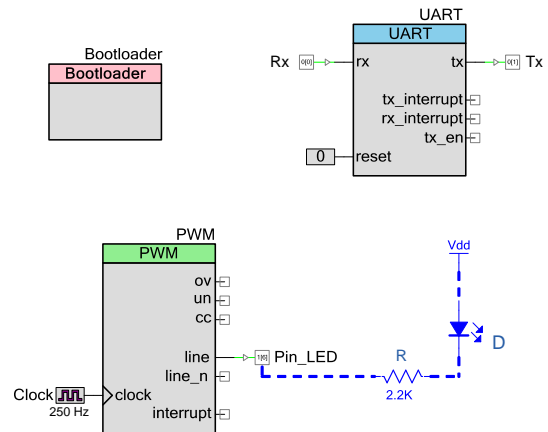


Figure 7. Top Design of UART Bootloader Project for PSoc 4

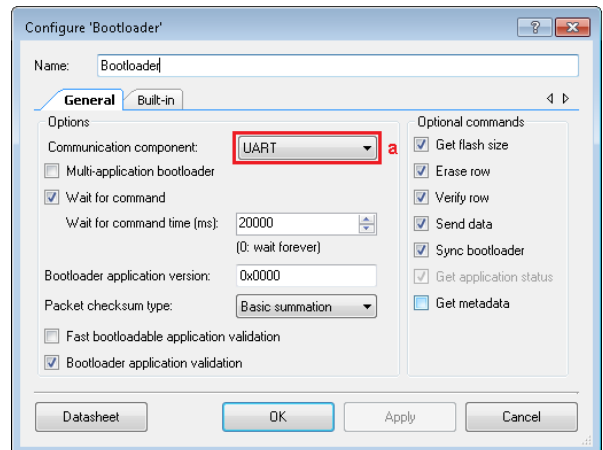


In this example there is no need to reset the UART so the reset terminal is connected to a Logic Low '0' Component.

7. To configure the Bootloader, double-click on the Component.

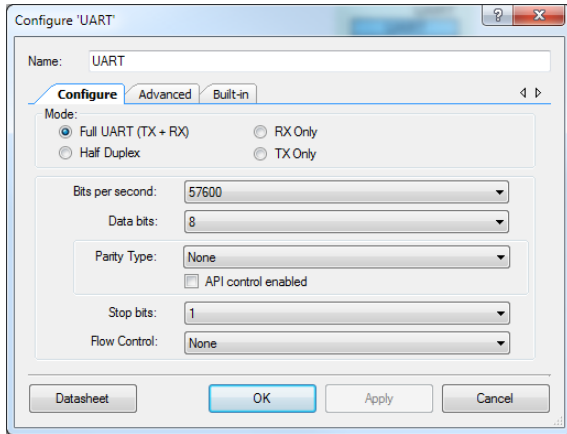
- a) Select UART as the Communication component, as [Figure 8](#) shows. Leave the other parameters at their default settings. For more information on these configuration parameters, refer to the [Bootloader Component datasheet](#).

Figure 8. Bootloader Configuration



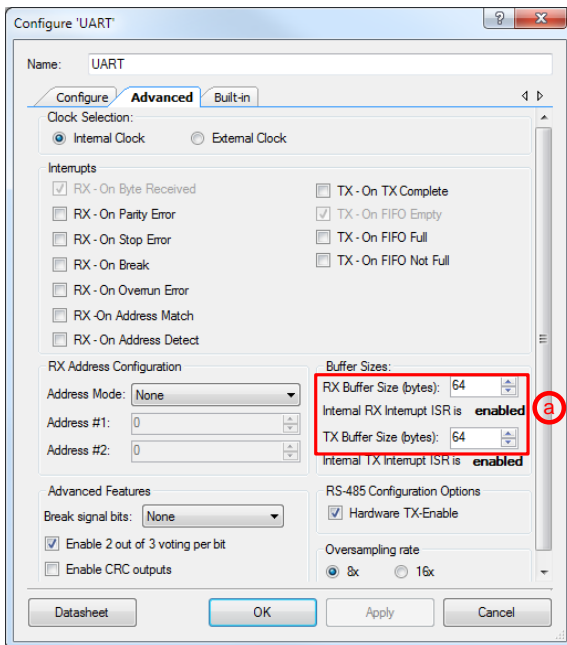
- To configure the UART Component, double-click on it. By default, it is in Full UART mode with a data rate of 57,600 bps. Leave all parameters at their default settings. [Figure 9](#) shows the basic configuration tab of the UART Component.

Figure 9. Basic UART Configuration



- Click on the Advanced tab of the UART configuration window.
  - 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 10](#).

Figure 10. Advanced UART Configuration



- 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.

- 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.

- For the Pin\_LED Component, leave the parameters at their default settings.

- 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. Assign the Pins as [Figure 11](#) and [Figure 12](#) show.

Figure 11. PSoc 3 and PSoc 5LP Pin Assignment for UART\_Bootloader Project

Name /	Port	Pin	Lock
Pin_LED	P6[2]	91	<input checked="" type="checkbox"/>
Rx	P0[0] OpAmp2:vout	71	<input checked="" type="checkbox"/>
Tx	P0[1] OpAmp0:vout	72	<input checked="" type="checkbox"/>

Figure 12. PSoc 4 Pin Assignment for UART Bootloader Project

Name /	Port	Pin	Lock
Pin_LED	PI[6] OA0:vplus_alt	43	<input checked="" type="checkbox"/>
Rx	P0[0] COMPl:inp, SCB0:spi_ssel[1]	24	<input checked="" type="checkbox"/>
Tx	P0[1] COMPl:inn, SCB0:spi_ssel[2]	25	<input checked="" type="checkbox"/>

- Review the *main.c* file – the `CyBtldr_Start()` function is added automatically when you create a bootloader project. This API function does the entire bootloading operation. It does not return – it ends with a software device reset. As such, any code that is placed after this API call is never executed.

Add two API functions 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();

    CyBtldr_Start();

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

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

15. Build the project and program it into a PSoC 3 device on **CY8CKIT-030**. If your target device is a PSoC 4, change the device before building the project, and program it to a **CY8CKIT-042**. If your target device is a PSoC 5LP, change the device before building the project, and program it to a **CY8CKIT-050**.

For **CY8CKIT-030** and **CY8CKIT-050**, Connect Tx and Rx on the connector P5 to the port pins P0[1] and P0[0] on the connector P4 using wires. For **CY8CKIT-042**, connect J8\_9 to P0[1] (Tx) and J8\_10 to P0[0] (Rx).

**Note** When building a UART-based project for PSoC 4, depending on the UART Component used and the selected baud rate, you may get a warning:

```
warning: Clock Warning: (UART_2_IntClock's accuracy range '461.538 kHz ±2.000%, (452.308 kHz - 470.769 kHz)' is not within the specified tolerance range '460.800 kHz ±2.000%, (451.584 kHz - 470.016 kHz)').
```

This is due to the base accuracy of the PSoC 4 IMO, which is ±2%. Your project may require a higher-accuracy external clock. For more information, see the [UART Component datasheet](#) or the PSoC Creator help article "Working with Clocks".

You have now created a simple UART-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 UART Component datasheets](#), and [AN73854 - PSoC 3 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 E](#) for details.

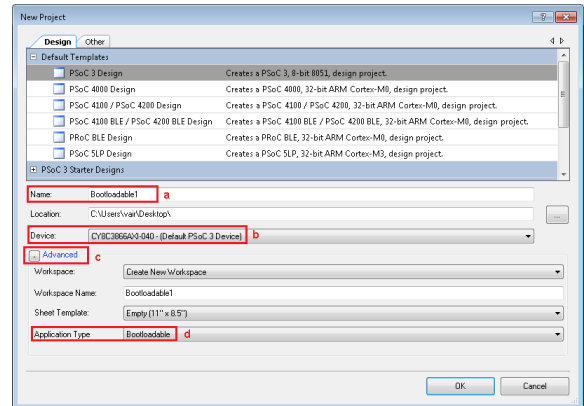
Let us now look at how to create bootloadable applications that can be used with this bootloader.

### PSoC 3 and PSoC 5LP Bootloadables

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

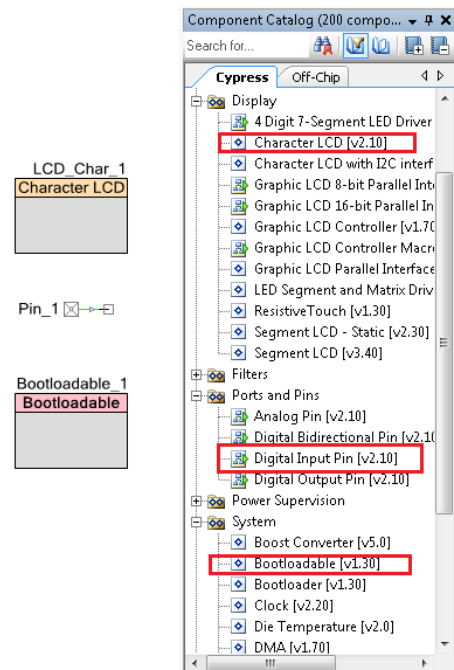
1. Create a new PSoC Creator project, as [Figure 13](#) shows.
  - a) Name the project as Bootloadable1.
  - b) The devices for this project and the [UART\\_Bootloader](#) project must be the same.
  - c) Click on the '+' button next to Advanced to expand the configuration options.
  - d) Select 'Bootloadable' as the application type.

Figure 13. Creating Bootloadable1 Project



2. For this project, we need the Bootloadable, Digital Input Pin, and LCD Components. Add these Components to your top design schematic, as [Figure 14](#) shows.

Figure 14. Bootloadable1 Project Components



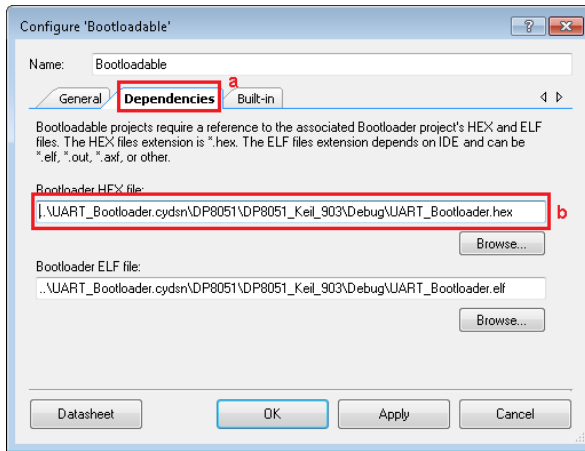
3. Rename the Components according to [Table 2](#).

Table 2. 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 do this, go to the dependencies tab of the Bootloadable Component configuration window as Figure 15 shows.
  - b) Select the *UART\_Bootloader.hex* file, as Figure 15 shows. For more information on Bootloader Component configuration, see the [Bootloader Component datasheet](#).

Figure 15. Bootloadable Component Configuration



You may find the *UART\_Bootloader.hex* file in Bootloader project's Debug or Release folder:

When PSoC 3 is the Bootloader,

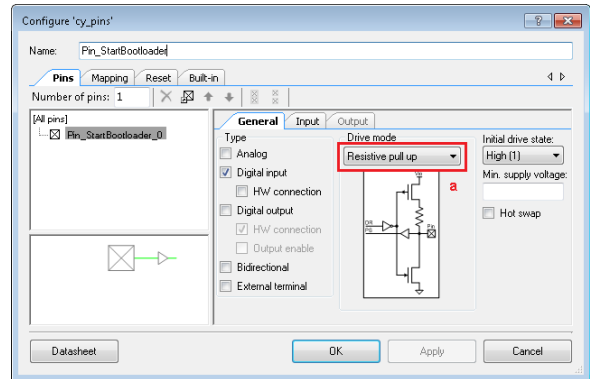
..\UART\_Bootloader\UART\_Bootloader.cydsn\DP8051\DP8051\_Keil\_951\Debug\UART\_Bootloader.hex

When PSoC 5LP is the Bootloader,

..\UART\_Bootloader\UART\_Bootloader.cydsn\CortexM3\ARM\_GCC\_484\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 16 shows.

Figure 16. Digital Input Pin Configuration



6. Now, with the addition of the annotation Components for the button and the input pin, the top design is complete; it should be similar to Figure 17.

Figure 17. Top Design of the 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 18 shows.

Figure 18. Pin Assignment 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 hard wired to P2[6:0], and SW2 is hard wired 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.

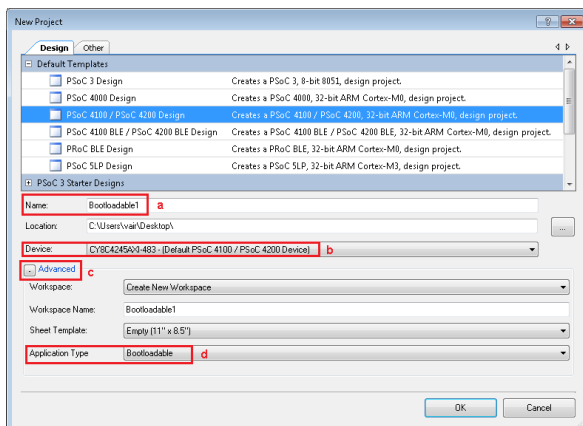
To bootload this project using a PC Host, see [Bootloading using a PC Host](#).

### PSoC 4 Bootloadables

We shall now create two bootloadable projects for PSoC 4 that can be used with the CY8CKIT-042. They are very similar – one blinks the green LED and the other blinks the blue LED on the CY8CKIT-042. This section describes the steps for creating these bootloadable projects.

1. Create a new PSoC Creator project as [Figure 19](#) shows.
  - a) Name the project as `Bootloadable1`.
  - b) The devices for this project and the [UART\\_Bootloader](#) project must be the same.
  - c) Click on the '+' button next to the Advanced tab to expand the configuration options.
  - d) Select 'Bootloadable' as the application type.

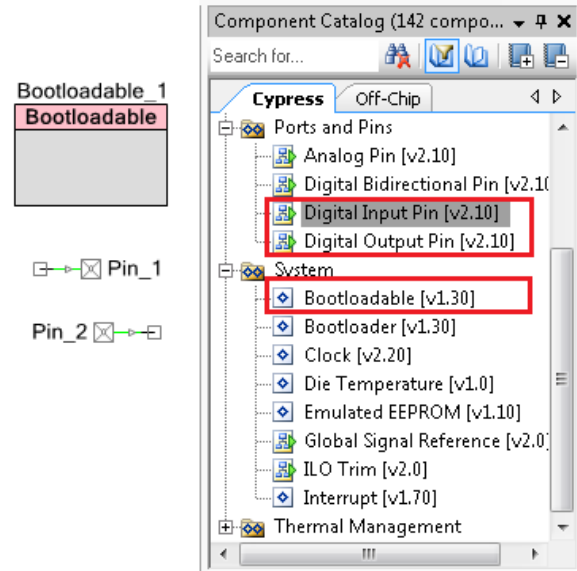
Figure 19. Creating Bootloadable1 Project for PSoC 4



2. For this project, we need the Bootloadable, Digital Input Pin and the Digital Output Pin Components. Add

these Components to your top design schematic, as [Figure 20](#) shows.

Figure 20. Bootloadable1 Project Components



3. Rename the Components according to [Table 3](#).

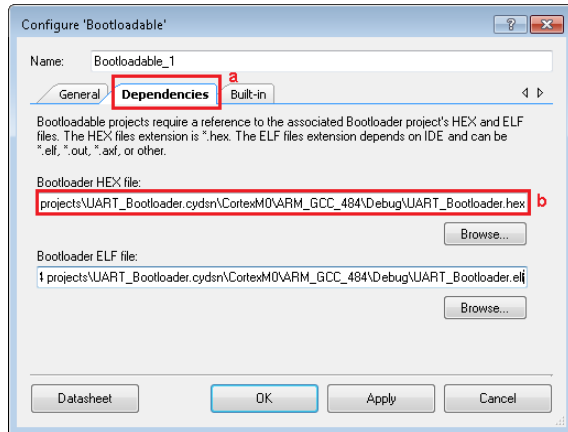
Table 3. Bootloadable1 Component Names

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 do this, go to the dependencies tab of the Bootloadable component configuration window as [Figure 21](#) shows.
  - b) Select the `UART_Bootloader.hex` file as shown. For more information on the Bootloader Component configuration, see the [Bootloader Component datasheet](#).

Figure 21. Bootloadable Component Configuration

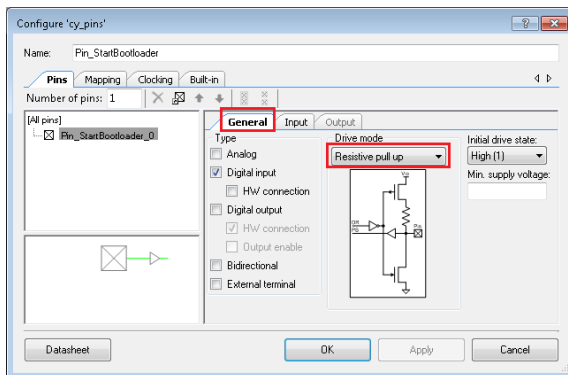


You may find the *UART\_Bootloader.hex* file in the Bootloader project's Debug or Release folder:

..\*UART\_Bootloader\UART\_Bootloader.cydsn\CortexM0\ARM\_GCC\_484\Debug\UART\_Bootloader.hex*

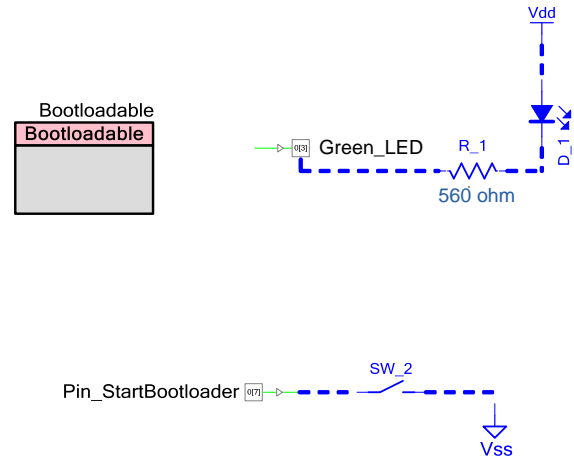
- The digital input pin (Pin\_StartBootloader) is used to switch from the application back to the bootloader. When the DVK button is pressed, it shorts to ground, so configure the drive mode of the pin to be Resistive Pull Up, as Figure 22 shows.

Figure 22. Digital Input Pin Configuration



- Now, with the addition of annotation Components for the button and the input pin, the top design is complete; it should be similar to Figure 23.

Figure 23. Top Design of the Bootloadable1 Project



- Assign the Pin Components to physical pins. In the Workspace Explorer window, double-click the *Bootloadable1.cydwr* file and assign the pins as Figure 24 shows.

Figure 24. Pin Assignments of Bootloadable1 Project

Name	Port	Pin	Lock
Green_LED	P0[2] SCB0:SPI:SS3	26	<input checked="" type="checkbox"/>
Pin_StartBootloader	P0[7] SCB1:SPI:SS0	31	<input checked="" type="checkbox"/>

On the CY8CKIT-042 kit board, the Green LED is hard-wired to P0[2] and SW2 is hard-wired to P0[7].

- 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.

- Build the project. When a bootloadable project is built, PSoC Creator generates a `.cyacd` file. This is the file that is bootloaded onto the target. For more information on this file and its contents, see [Appendix B](#).

**Note** When building a UART-based project for PSoC 4, depending on the UART Component used and the selected baud rate, you may get a warning:

```
warning: Clock Warning: (UART_2_IntClock's accuracy range '461.538 kHz ±2.000%, (452.308 kHz - 470.769 kHz)' is not within the specified tolerance range '460.800 kHz ±2.000%, (451.584 kHz - 470.016 kHz)').
```

This is due to the base accuracy of the PSoC 4 IMO, which is  $\pm 2\%$ . Your project may require a higher-accuracy external clock. For more information, see the [UART Component datasheet](#) or the PSoC Creator help article “Working with Clocks.”

- To create the other bootloadable project that blinks the Blue LED, repeat the previous steps in this section. Name the project as Bootloadable2. The only difference between the two projects is that the Digital Output Pin Blue\_LED is connected to P0[3]. On the CY8CKIT-042 kit board the Blue LED is hard wired to the pin P0[3].

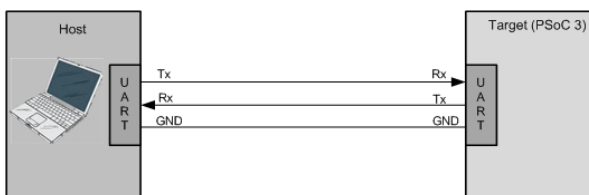
**Note** For PSoC Creator versions before 3.0, if the bootloader is updated, you must also rebuild all bootloadable projects that depend on that bootloader project. Use the “Clean and Build” option.

Now, let’s bootstrap this project into a target PSoC 4 using the UART Bootloader Host application (PC Host).

### Bootloading Using a PC Host

A bootloader host executable is provided with this application note for bootloading an application from a PC host, as [Figure 25](#) shows. `UARTBootloaderHost.exe` can be found in the `Bootloader_Host_GUI_exe` folder inside the `AN68272.zip` file. Review and follow the instructions in the `Prerequisites.txt` file to install the software required to run this tool. The dll to be used with the executable for 64-bit and 32-bit Windows platforms are given in respective folders inside the `Bootloader_Host_GUI_exe` folder. Note that this executable does not support multi-application bootloading. Use the Bootloader Host provided with PSoC Creator for that purpose. This tool can be accessed through Tools > Bootloader Host.

Figure 25. Bootloading Using a PC Host



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

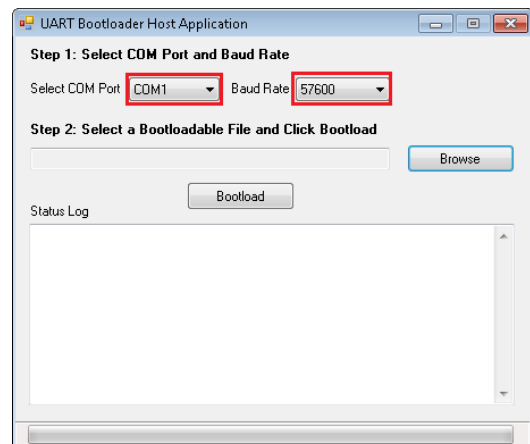
- To bootstrap PSoC 3 or PSoC 5LP, connect an RS-232 serial cable to port (P7) of the [CY8CKIT-030](#) / [CY8CKIT-050](#).

To bootstrap PSoC 4, an RS-232 serial cable is not required, as the PSoC 5LP on the [CY8CKIT-042](#) has a USB-UART bridge. Therefore, you just need to provide an external connection between the UART port pins of PSoC 4 and the onboard PSoC 5LP on the [CY8CKIT-042](#). Connect P0[0] to Pin 10 on the header J8 and P0[1] to Pin 9 on the header J8.

- Open the Bootloader Host Application (`UARTBootloaderHost.exe`). Select the appropriate COM port and baud rate, as [Figure 26](#) shows. The baud rate selected must be the same as configured in the UART Component in the bootloader project ([Figure 9](#) on page 6).

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

Figure 26. Bootloader Host Application



- 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,

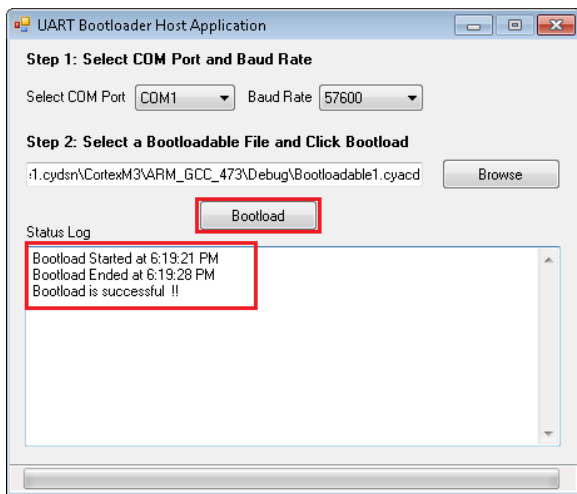
```
... \Bootloadable1.cydsn\CortexM3\ARM_GCC_484\Debug\Bootloadable1.cyacd
```

When PSoC 4 is the Bootloader,

```
...\\Bootloadable1.cydsn\\CortexM0\\ARM_GCC_484D
ebug\\Bootloadable1.cyacd
```

4. Browse the `.cyacd` file and click the “Bootload” button to begin bootloading. The bootloading status will be displayed in the Log section as Figure 27 shows.

Figure 27. Downloading Bootloadable Project



5. On a successful bootload operation, in the case of PSoC 3 or PSoC 5LP the message "Hello" is displayed on the CY8CKIT-030 / CY8CKIT-050 LCD. In the case of PSoC 4, the Green LED on the CY8CKIT-042 starts to blink.
6. To bootload again, press SW2 on the DVK. This makes the PSoC device enter the Bootloader. Now, repeat the steps 3 to 5 to bootload again.

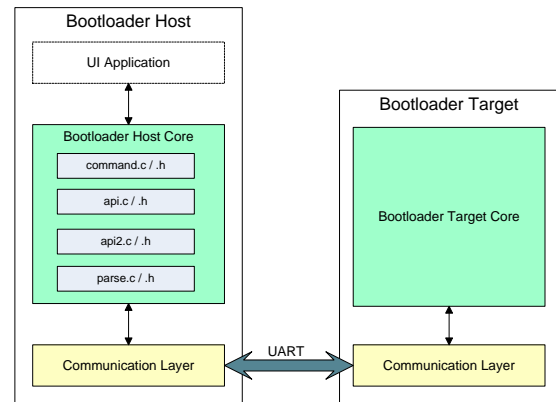
## Bootloading Using an Embedded Host

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.

### Bootloader Host Program

Figure 28 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 28. 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. They contain 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.

### Bootloader System APIs

All APIs for the Target Core and Communication Layer are automatically generated by PSoC Creator, when you build a bootloader project.

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

```
<install folder> \ PSoC Creator \ 3.1 \ 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`.

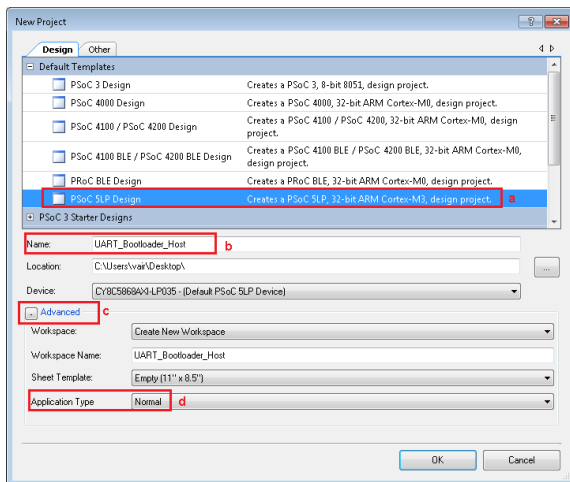
[Appendix F](#) shows you how to use these APIs to create your own host using C#.

### 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.

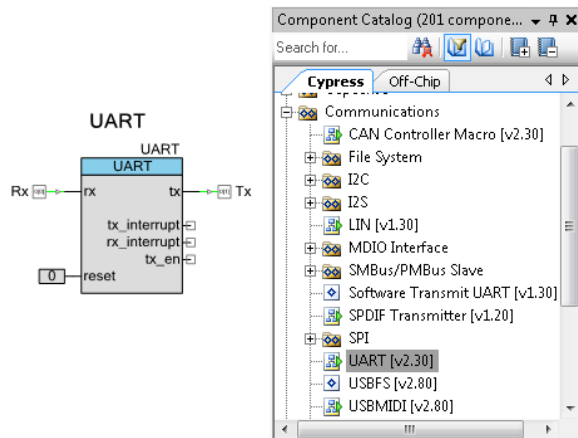
1. Create a new PSoC Creator project, as [Figure 29](#) shows.
  - a) Select the design template to be 'PSoC 5LP Design'.
  - b) Name the project as UART\_Bootloader\_Host.
  - c) Click on the '+' button next to Advanced to expand the configuration options and create a new workspace for the project.
  - d) Make sure that the Application Type is Normal.

Figure 29. Create a UART\_Bootloader\_Host Project



2. Add a UART Component to the top design schematic, as [Figure 30](#) shows. Also, add Digital Input Pin and Character LCD Components to the top design.

Figure 30. UART Component



3. Rename the Components according to [Table 4](#).

Table 4. 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

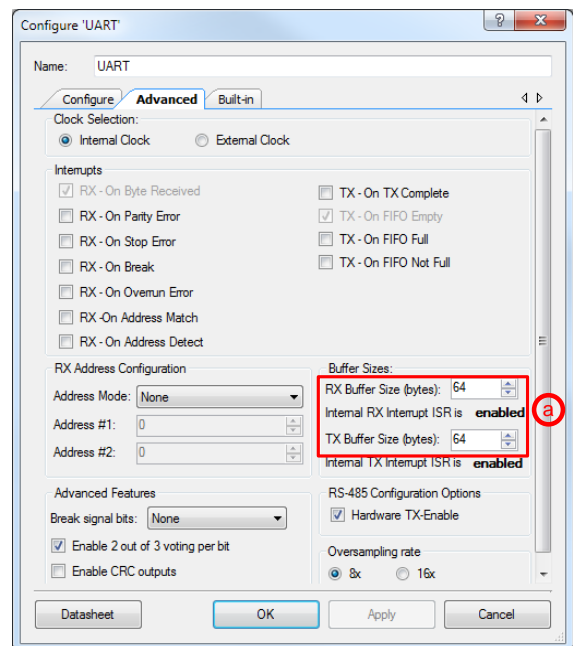
The next step is to configure these Components.

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.

- 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 31](#) on page 13.

Figure 31. UART Advanced Configuration

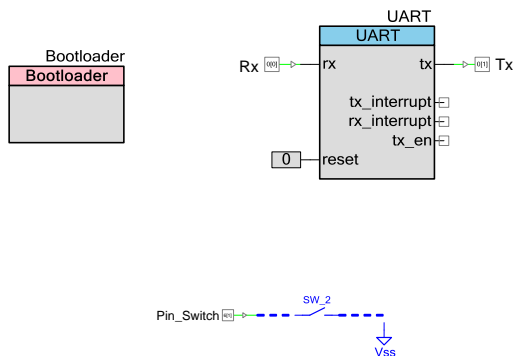


5. The Digital Input Pin Pin\_Switch is used to initiate the bootloading operation in the host. When the kit button

is pressed, it shorts to ground, so we need to configure this pin to have a resistive pull-up.

- Now, with the addition of annotation Components to the button, the top design of this project should be similar to [Figure 32](#).

Figure 32. Top Design of UART\_Bootloader\_Host project



- 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 33](#) shows.

Figure 33. UART\_Bootloader\_Host Project Pin Assignments

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

On the CY8CKIT-030 and CY8CKIT -050 kit boards, the LCD pins are hard wired to P2[6:0], and SW2 is hard wired to P6[1]. Wire the kit board to connect the designated port pins (P4) to TX and RX (P5).

- 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 on page 12).

The *main()* function has a variable called 'toggle'. It alternates between '0' and '1' on every button press.

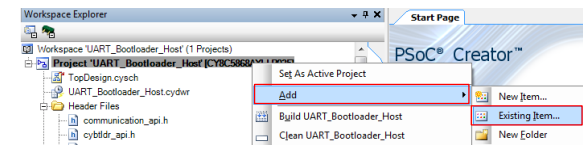
This makes the host select alternate bootloadable files.

- 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:

```
<install folder> \PSoC Creator \ 3.1 \ PSoC Creator \ cybootloaderutils
```

To include these files, go to the Workspace Explorer window, right-click on the project name, and select Add > Existing Item, as [Figure 34](#) 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 34. Adding API files



- 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 may 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.

- 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 *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.

- Build the project and program it into the PSoC 5LP on the [CY8CKIT-050](#) kit.

## Testing the Projects

**Note** 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.

### Kit Configuration

To test the projects, configure the kits as follows:

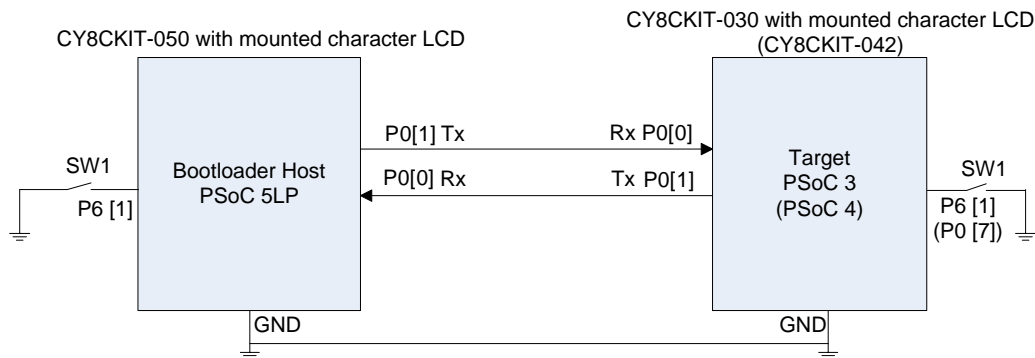
For **CY8CKIT-030**:

1. Program the PSoC 3 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 **CY8CKIT-042**:

1. Program the PSoC 4 with the UART\_Bootloader project (PSoC 4 project).

Figure 35. Host / Target Connections



### Bootloading PSoC 3

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 *Bootloadable1.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. The LED starts blinking.

2. Set the jumper J9 to 5 V.

For **CY8CKIT-050**:

1. Program the 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:

1. P0 [0] of CY8CKIT-030 (CY8CKIT-042) to P0 [1] of CY8CKIT-050
2. 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 35](#).

**Note** In [Figure 35](#), the target can also be a PSoC 5LP (CY8CKIT-050), in which case, the Pin connections are same as that for CY8CKIT-030.

- On 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.

### 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 on the CY8CKIT-050.
- On the first button press (P6 [1]) on the CY8CKIT-050, the *Bootloadable1.cyacd* file is bootloaded to the target PSoC 4. On successful completion, the Green LED on the CY8CKIT-042 starts to blink.

- For subsequent bootloading operations, press the button (P0 [7]) on the target CY8CKIT-042. This makes the PSoC 4 enter the bootloader and be ready to bootload a new application. The Red LED on the CY8CKIT-042 starts to blink.
- On 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 CY8CKIT-042 starts to blink.

## Summary

This application note has 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 [Appendix](#) sections and the [PSoC 3](#), [PSoC 4](#) 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](#)
- [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.

## Related Projects

The projects attached to this application note are organized as shown in [Table 5](#).

Table 5. 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 another PSoC 3 or PSoC 4 or PSoC 5LP with UART as the communication channel.
UART_Bootloader	Bootloader project that has UART as the communication channel. This 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 CY8CKIT-042.
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 CY8CKIT-042.

## 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 36](#) 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 has a maximum flash of 64 KB, so it has only one array and the only valid array ID is 0. The PSoC 4100 and 4200 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 36](#) 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's the first 128-byte boundary). This region of the flash includes:

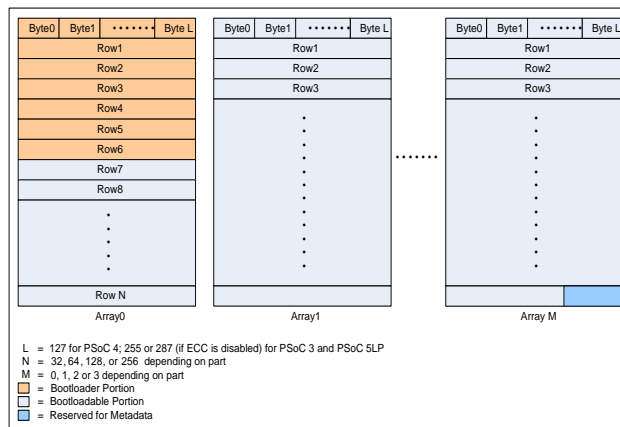
- Vector table for the bootloadable project (PSoC 4 and PSoC 5LP only)
- The bootloadable project code and data

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 (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, see [Metadata Layout in Flash](#).

Figure 36. Physical Organization of Flash Memory in PSoC

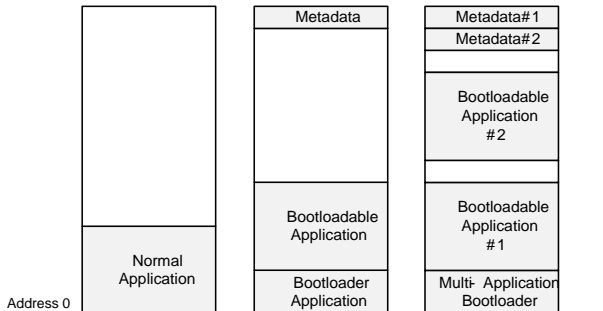


## Memory Usage in PSoC

There are two types of bootloader project types: standard bootloader and multi-application bootloader. 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 37 shows the flash memory usage for each type of PSoC Creator project.

Figure 37. 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 37 shows. Various parameters, depending upon the device used, are stored in this block, as Table 6 shows. For the multi-application bootloader, there are two sets of metadata.

Table 6. Metadata Layout

Address	PSoC 3	PSoC 4 / PSoC 5LP
0x00	App Checksum	App Checksum
0x01	Reserved	Application Address
0x02		
0x03	Application Address	Application Address
0x04		
0x05	NA	Last Bootloader Row
0x06	NA	
0x07	Last Bootloader Row	
0x08		Application Length
0x09		
0x0A		
0x0B	Application Length	NA
0x0C		NA
0x0D		NA
0x0E		NA

Address	PSoC 3	PSoC 4 / PSoC 5LP
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	Bootloadable Application Custom ID	Bootloadable Application Custom ID
0x17		
0x18	NA	NA
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 7 shows. Each row of flash can be configured to have a different protection level, which can be set using PSoC Creator (the Flash Security tab of the .cydwr file).

Table 7. Levels of Flash Protection

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

**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 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**

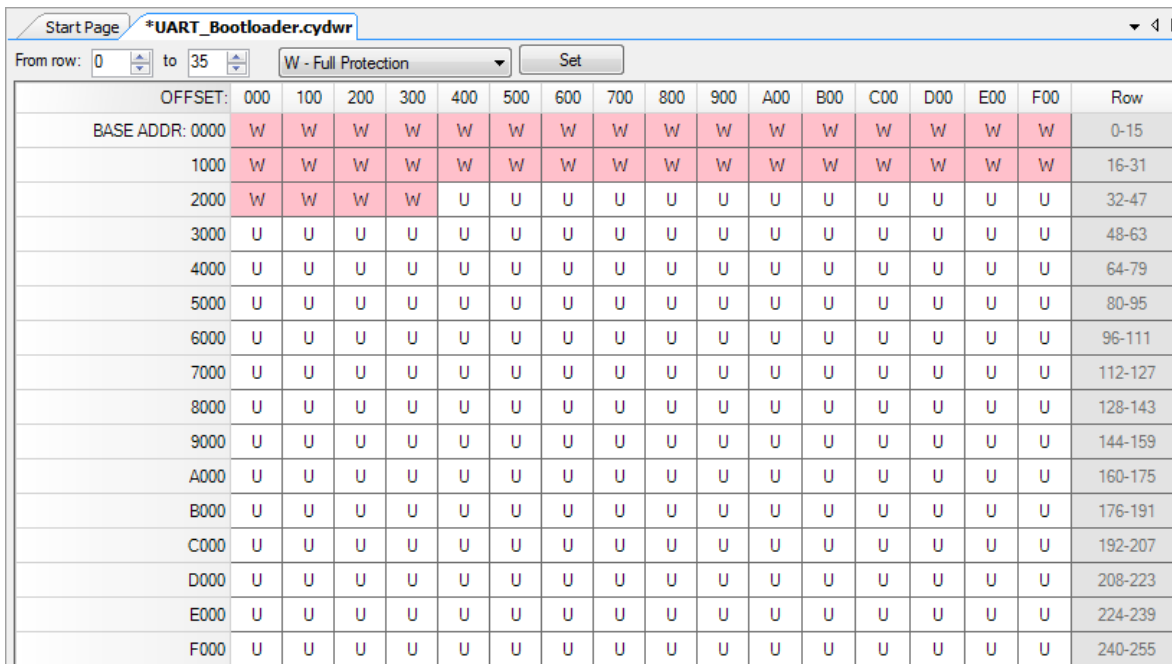
When the bootloader project is built, the PSoC Creator Output window shows the amount of flash used. For

example, if the flash occupied by the 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), i.e., flash locations 0x0000 to 0x2300. Set the flash protection level as Full protection for these rows (under the Flash Security tab of the .cydwr file in PSoC Creator). The protection level for the remaining rows can be Unprotected (the default) or Field upgrade, as Figure 38 shows. For more information on how to use the Flash Protection dialog, see the PSoC Creator help article "Flash Security Editor."

Figure 38. Flash Protection in PSoC Creator



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

**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, 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]
```

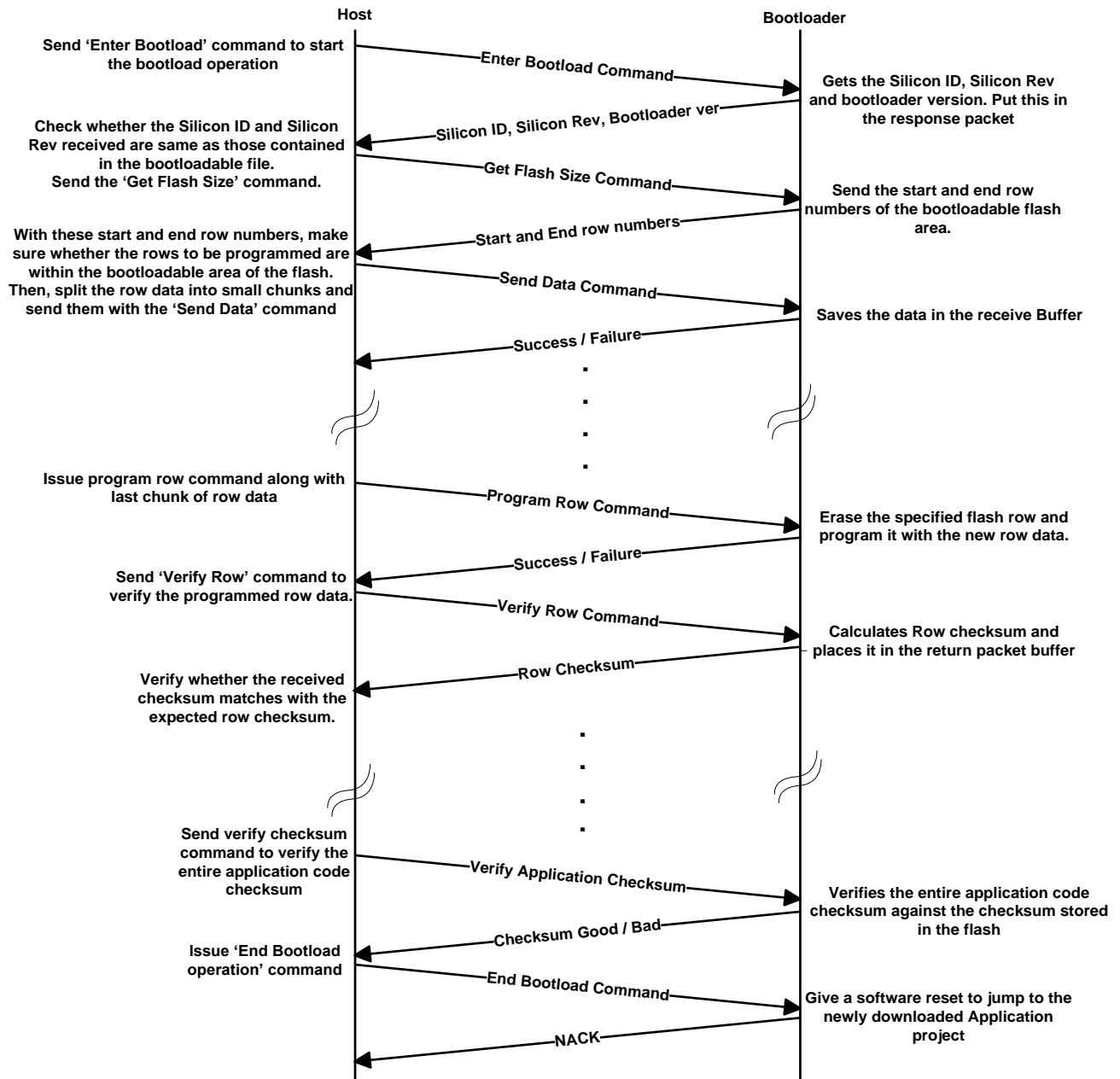
The checksum type in the header indicates the type of checksum used in the packets sent between the bootloader and the bootloader host during the bootloading operation. If this byte is 0, the checksum is a basic summation. If it is 1, the checksum is CRC-16.

## Appendix C – Host / Target Communications

### Communication Flow

In [Bootloader Function Flow](#), we looked at the operation of a bootloader in PSoC, and UART Bootloader Host introduced the building blocks of a bootloader host. With this background, [Figure 39](#) 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.

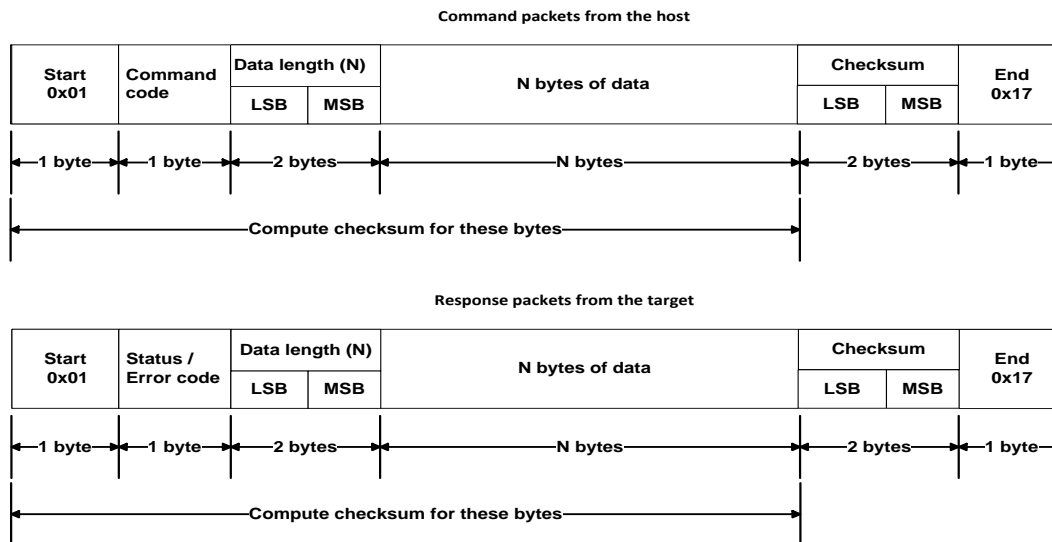
Figure 39. Communication Flow During Bootloading



## 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 40](#) shows.

Figure 40. 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 multi byte 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 8](#).

### Command and Status / Error Codes

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

[Table 8](#) provides a list of commands and their expected responses. [Table 9](#) provides a list of status and error codes.

Table 8. Bootloading Commands

Command Byte	Command	Data Byte in the Command Packet	Expected Response Data Bytes
0x31	Verify checksum	N/A	1 byte: Non zero or '0'. If it is a non-zero byte, then the application checksum matches and it is a valid application. If it is a zero byte, then the checksum is bad and the application is invalid.
0x32	Get flash size	Flash array ID, 1 byte	First row number of the bootloadable flash, 2 bytes; Last row number of the bootloadable flash, 2 bytes. 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.

Command Byte	Command	Data Byte in the Command Packet	Expected Response Data Bytes
0x34	Erase row	Flash array ID, 1 byte; Flash row number, 2 bytes	N/A. Erases the contents of the specified flash row.
0x35	Sync bootloader	N/A	N/A. Resets the bootloader to a clean state. Any data which was buffered in will be thrown out. This command is needed only if the bootloader and the host go out of sync with each other.
0x36	Set active application (valid only for 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 9. 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 of 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 bootloader operation. It has functions to open and close files. It invokes the functions of the *cybtldr\_api.c / .h* API for the bootloader 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 bootloader operation, erasing a row, programming a row, verifying a row and ending the bootloader operation. [Table 10](#) describes in detail the functions of this API file.

Table 10. Functions of *cybtldr\_api.c / .h*

Function	Description
CyBtldr_StartBootloadOperation	<ul style="list-style-type: none"> <li>Enables the communication interface and sends an Enter Bootloader command to the target.</li> <li>From the response packet received, verifies the silicon ID, silicon revision of the target device, and bootloader version.</li> </ul>
CyBtldr_ProgramRow	<ul style="list-style-type: none"> <li>First validates a row, i.e., sends a Get Flash Size command to the target for a particular array ID of the target flash. In response to this, the target returns the start and end row numbers of the bootloadable flash portion in that array. The host reads this response and checks whether the specified row is in the bootloadable area of the flash.</li> <li>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.</li> <li>Along with the last portion of row data, sends a Program Row command to the target.</li> </ul>
CyBtldr_VerifyRow	<ul style="list-style-type: none"> <li>This function also first validates a row for a particular array ID and row number.</li> <li>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.</li> <li>The returned checksum is verified against the expected checksum value.</li> </ul>
CyBtldr_EraseRow	<ul style="list-style-type: none"> <li>This function also first validates a row for a particular array ID and row number.</li> <li>If row validation is successful, sends an Erase Row command for the validated flash row.</li> </ul>
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 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 PSoC's digital and analog routing features. When you configure the registers and routing, you must make sure that in addition to setting the bits for the new configuration, you reset the bits for the old configuration. Otherwise, the new configuration may not work, and may even damage the device.

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

### Effect on Device I/O Pins

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

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

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

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

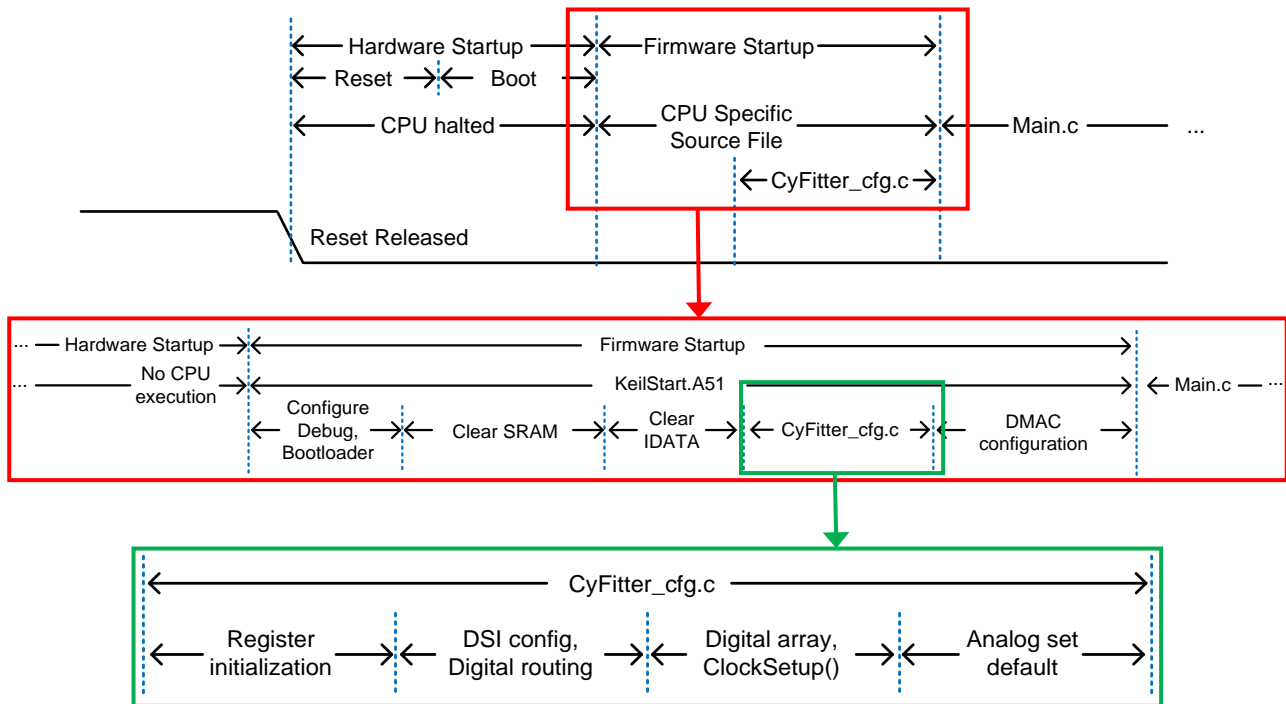
- The **Reset** tab for I/O ports, the individual Pin Component configurations
- The **System** tab for all other NVLs, the design-wide resources (DWR) window

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

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

[Figure 41](#) 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 41. 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 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<sup>2</sup>C, 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 41). 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 12](#) shows.

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

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

## Appendix F – Miscellaneous Topics

### Bootloader Versus HSSP

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

### What Happens If Power Fails During the Bootload Operation?

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

### Why Do We Need a Reset to Jump Between the Bootloader and the Bootloadable Projects?

PSoC 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](#).

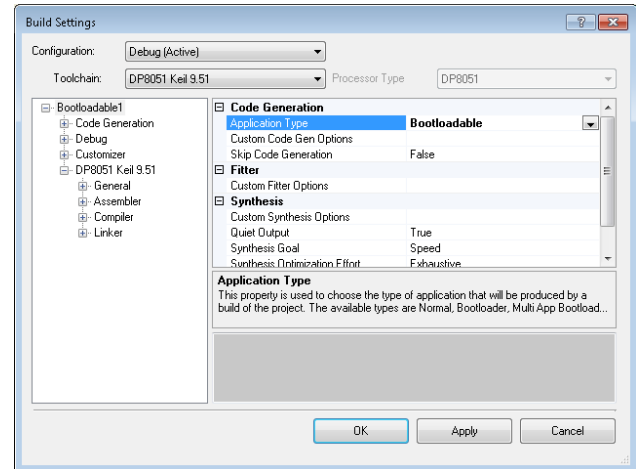
### Converting a Normal Application Project to a Bootloadable Project

If you have already created a standard (Normal) project and want to convert it to a bootloadable project, change the application type of the project to Bootloadable. To do this right-click on the project > Build setting > Code Generation > General tab and change Application Type as [Figure 42](#) shows.

After changing the application type, you must add a Bootloadable Component onto the top design and add the bootloader project's .hex file as a dependency as [Figure 15](#) on page 8 shows.

If a project is created as a normal project, and then later changed to a bootloader project by changing the application type as in [Figure 42](#), you should insert the `CyBldr_Start()` function call in `main.c` for the bootloader project to work as expected.

Figure 42. Changing Application Type to Bootloadable



### 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 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 onto 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

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 37](#) on page 18 shows the placement of two applications in flash memory.

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

1. Create a new MABL bootloader project. Set the application type as Multi-App Bootloader – check the Multi-App bootloader checkbox in the 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 – 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 .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 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 GCC compiler optimization set to "size". And, 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 43](#) shows.

Set the Device Configuration Mode to 'Compressed' in `.cydwr > System` tab, as [Figure 44](#) shows, to minimize flash memory usage. Set Device Configuration Mode to DMA if startup time is more important than code size.

Figure 43. Unchecking Optional Commands in Bootloader Component

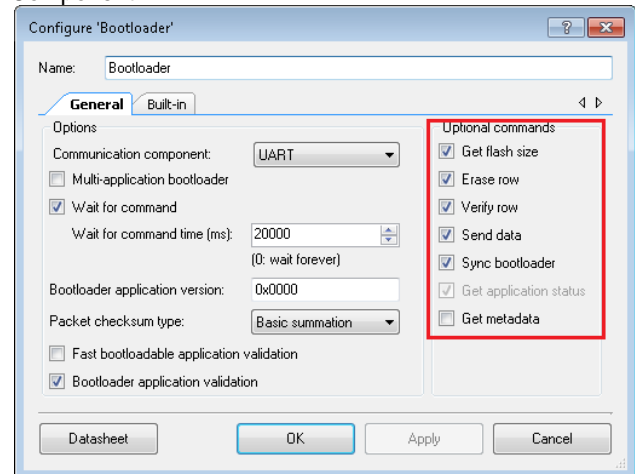
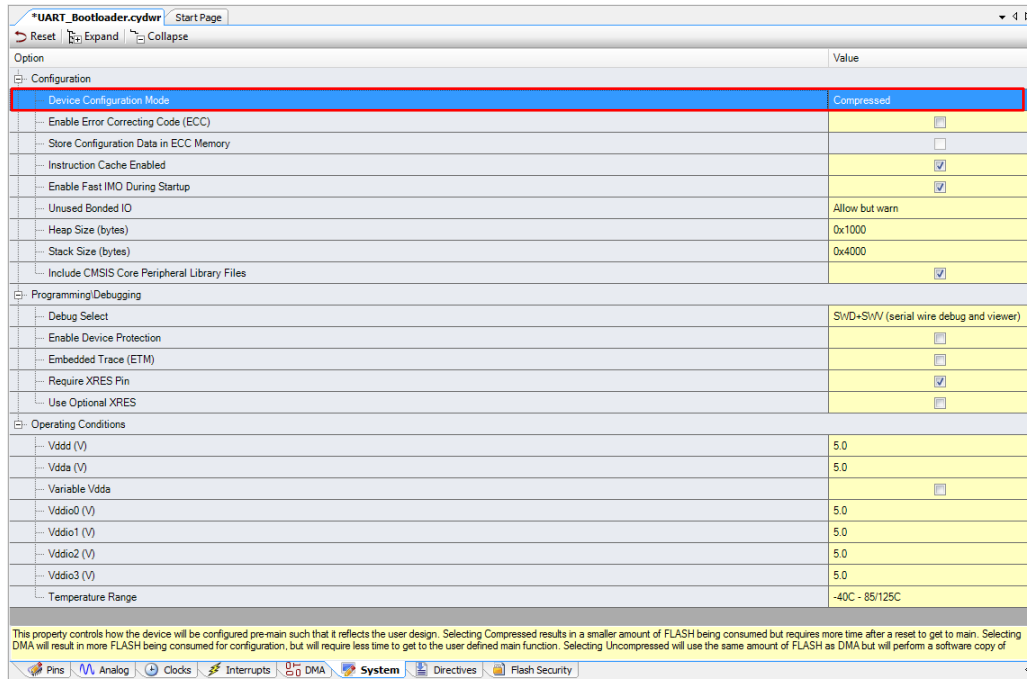


Figure 44. Device Configuration Mode



Option	Value
Device Configuration Mode	Compressed
Enable Error Correcting Code (ECC)	<input type="checkbox"/>
Store Configuration Data in ECC Memory	<input type="checkbox"/>
Instruction Cache Enabled	<input checked="" type="checkbox"/>
Enable Fast IMO During Startup	<input checked="" type="checkbox"/>
Unused Bonded IO	Allow but warn
Heap Size (bytes)	0x1000
Stack Size (bytes)	0x4000
Include CMSIS Core Peripheral Library Files	<input checked="" type="checkbox"/>
Debug Select	SWD+SWV (serial wire debug and viewer)
Enable Device Protection	<input type="checkbox"/>
Embedded Trace (ETM)	<input type="checkbox"/>
Require XRES Pin	<input checked="" type="checkbox"/>
Use Optional XRES	<input type="checkbox"/>
Vddd (V)	5.0
Vdda (V)	5.0
Variable Vdda	<input type="checkbox"/>
Vddio0 (V)	5.0
Vddio1 (V)	5.0
Vddio2 (V)	5.0
Vddio3 (V)	5.0
Temperature Range	-40C - 85/125C

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

## Appendix G – C# Bootloader Host Application

**Note** The following software must be installed to develop a GUI for a Bootloader Host Application:

- Visual C# 2010 Express Edition and Visual C++ 2010 Express Edition. The trial versions can be found [here](#), or
- Visual Studio 2010 Complete

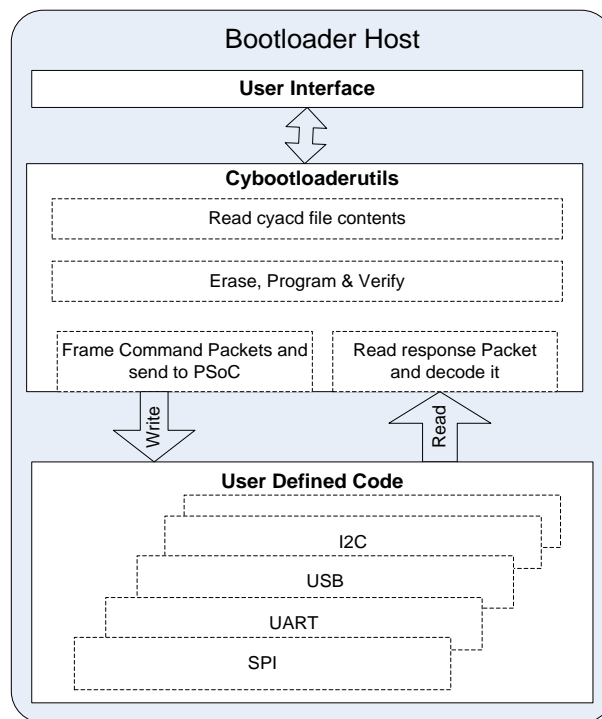
Professional editions of the software can be purchased from Microsoft.

Read this [MSDN page](#) if you are targeting the DLL for a 64-bit Windows platform.

You should have some experience with Visual C# or Visual C++ in order to implement the GUI explained in this section.

The architecture of a PSoC 3 or PSoC 5LP Bootloader Host is as shown in [Figure 45](#).

Figure 45. Bootloader Host Architecture



The steps to create a C# Bootloader Host Application are as follows:

1. Create a **dynamic link library (dll)** for the C functions (cybootloader Utils) provided with PSoC Creator.
2. Create a **C# GUI** (User Interface).
3. Import essential bootloader functions from the **dll** created in step 1.
4. Provide definition for the communication functions using a serial Port (UART) communication interface.
5. Complete the Windows Form Application.

Each step is explained in detail as follows:

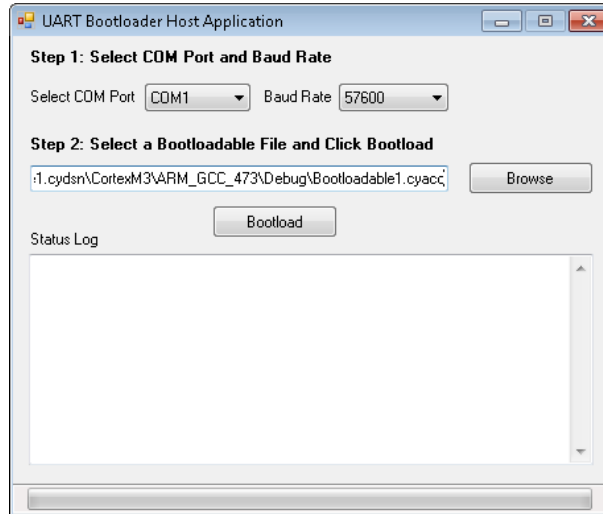
### Step 1: Create bootloaderutils.dll

Create a dynamic link library (dll) for the C functions provided with PSoC Creator. A copy of the dll (BootLoad\_Utils.dll) created using Visual C++ 2010 Express Edition for various windows platforms is attached with this application note (BootLoad\_Utils\_dll).

## Step 2: Create a C# Windows Form Application

Create a C# Windows Form Application. Include necessary Windows forms from the toolbox including serialPort. A snapshot of the GUI that was created for the UART Bootloader provided with this application note is shown in [Figure 46](#).

Figure 46. UART Bootloader Host GUI



## Step 3: Import Essential Bootloader Functions from the dll Created in Step 1

The methods and objects provided by the **BootLoad\_Utils.dll** are accessed from the C# application using the **Platform Invoke (pinvoke)** utility provided by Windows. Platform invoke is a service that enables managed code<sup>1</sup> to call unmanaged<sup>2</sup> functions implemented in dynamic link libraries (DLLs). It locates and invokes an exported function and marshals its arguments (integers, strings, arrays, structures, and so on). To use exported DLL functions:

- Identify functions in the DLL that are directly invoked by the C# host application. This includes `CyBtldr_Program()`, `CyBtldr_Erase()`, `CyBtldr_Verify()`, and `CyBtldr_Abort()`.
- Create a class to hold DLL functions. Specify the names of the functions and name of the DLL that contains them in the class.

You can use an existing class, create an individual class for each unmanaged function, or create one class that contains a set of unmanaged functions.

- Create prototypes in managed code.

In C#, we use the **DllImport** attribute to identify the DLL and function. Mark the method with the **static** and **extern** modifiers. Note that these prototypes are simple and do not define the required **DllImport** attributes. Refer to the source code for details.

```
[DllImport("BootLoad_Utils.dll")]
int CyBtldr_Program(string file, ref CyBtldr_CommunicationsData comm,
CyBtldr_ProgressUpdate update);

[DllImport("BootLoad_Utils.dll")]
int CyBtldr_Erase(string file, ref CyBtldr_CommunicationsData comm,
CyBtldr_ProgressUpdate update);

[DllImport("BootLoad_Utils.dll")]
int CyBtldr_Verify(string file, ref CyBtldr_CommunicationsData comm,
CyBtldr_ProgressUpdate update);
```

<sup>1</sup> **Managed Code:** Code that executes under the control of the .NET runtime is called managed code.

<sup>2</sup> **Unmanaged Code:** Code that runs outside the runtime is called unmanaged code.



```
[DllImport("BootLoad_Utils.dll")]
int CyBtldr_Abort();
```

- d. Call the DLL function. You can invoke the method in your C# code as you invoke any other method.

#### Step 4: Provide Definitions for Communication Functions in Visual C#

A bootloader normally requires a particular communication protocol to send bootloader commands. The C files provided by PSoC Creator provide C code that has functions to read the *cyacd* file and frame bootloader packets. You only need to define the communication functions. Therefore, the following four unmanaged functions that correspond to the UART communication must be managed in C# using the desired communications Component – UART in this example.

- OpenConnection()
- CloseConnection()
- ReadData()
- WriteData()

To achieve this, first define in C# the functions that perform OpenConnection, CloseConnection, ReadData, and WriteData. Then use these functions as delegates to the functions OpenConnection(), CloseConnection(), ReadData(), and WriteData() in BootLoad\_Utils.dll.

For Example, to implement the CloseConnection function, do the following:

- a. Indicate that a delegate will be used to implement the function:

```
[UnmanagedFunctionPointer(CallingConvention.Cdecl)]
public delegate int OpenConnection_UART();
```

- b. Define the function in C#:

```
public int OpenConnection()
{
    /*Open communication channel*/
    serialPort.Open();
}
```

The above steps are required for all the delegated functions.

- c. The structure “CyBtldr\_CommunicationsData” present in *cybtldr.h* must be declared in the C# program as well. This is done inside a class. See the “Bootload\_Utils” class in the C# code attached.

```
[StructLayout(LayoutKind.Sequential)]
public struct CyBtldr_CommunicationsData
{
    public OpenConnection_UART OpenConnection;
    public CloseConnection_UART CloseConnection;
    public ReadData_UART ReadData;
    public WriteData_UART WriteData;
    public uint MaxTransferSize;
};
```

- d. Create an instance (comm\_data) of the structure that we defined above:

```
Bootload_Utils.CyBtldr_CommunicationsData comm = new
Bootload_Utils.CyBtldr_CommunicationsData();
```

- e. Delegate the members of the structure:

```
comm.OpenConnection = OpenConnection;
comm.CloseConnection = CloseConnection;
comm_data.ReadData = ReadData;
comm_data.WriteData = WriteData;
comm_data.MaxTransferSize = 64;
```

A brief explanation of what each function does for the UART Bootloader is given below.

```
public delegate int OpenConnection_UART();
```

For the UART Bootloader, this function makes a connection to the selected com port.

```
public delegate int CloseConnection_UART ();
```

This function closes the com port connection.

```
public delegate int ReadData_UART (IntPtr buffer, int size);
```

This function waits for a timeout period for the data to be available in the input buffer. The data length is specified as size in the above API and the pointer to the buffer is provided as IntPtr.

```
public delegate int WriteData_UART(IntPtr buffer, int size);
```

This function writes data to the selected serial port. The data to be sent is preloaded in a buffer. The data length is specified as size in the above API and the pointer to buffer is provided as IntPtr.

```
public delegate void CyBtldr_ProgressUpdate(byte arrayID, ushort rowNum);
```

This function provides a way to visualize the progress of the bootload operation. This function can be as simple as updating a text box showing the percentage of progress or it can be something that updates a progress bar.

The definitions for the communication functions is given in *delegated\_functions.cs* and *BootLoad\_Utils\_NativeCode.cs* in the attached C# project.

### Step 5: Complete the Windows Form Application

After the necessary APIs are imported and definitions are provided to the communications APIs described above, these are used to implement various control operations. For example, pressing the Bootload button will initiate a bootload operation. Therefore, on a button-click event, the function "*Bootload\_Utils.CyBtldr\_Program*" must be invoked.

Check the UART Bootloader Host C# project to see the detailed implementation of the host.

## Document History

Document Title: PSoC<sup>®</sup> 3, PSoC 4, and PSoC 5LP UART Bootloader – AN68272

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

## 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

Automotive	<a href="http://cypress.com/go/automotive">cypress.com/go/automotive</a>
Clocks & Buffers	<a href="http://cypress.com/go/clocks">cypress.com/go/clocks</a>
Interface	<a href="http://cypress.com/go/interface">cypress.com/go/interface</a>
Lighting & Power Control	<a href="http://cypress.com/go/powerpsoc">cypress.com/go/powerpsoc</a>
Memory	<a href="http://cypress.com/go/memory">cypress.com/go/memory</a>
Optical Navigation Sensors	<a href="http://cypress.com/go/ons">cypress.com/go/ons</a>
PSoC	<a href="http://cypress.com/go/psoc">cypress.com/go/psoc</a>
Touch Sensing	<a href="http://cypress.com/go/touch">cypress.com/go/touch</a>
USB Controllers	<a href="http://cypress.com/go/usb">cypress.com/go/usb</a>
Wireless/Rf	<a href="http://cypress.com/go/wireless">cypress.com/go/wireless</a>

### PSoC<sup>®</sup> Solutions

[psoc.cypress.com/solutions](http://psoc.cypress.com/solutions)  
[PSoC 1](#) | [PSoC 3](#) | [PSoC 4](#) | [PSoC 5LP](#)

### Cypress Developer Community

[Community](#) | [Forums](#) | [Blogs](#) | [Video](#) | [Training](#)

### Technical Support

[cypress.com/go/support](http://cypress.com/go/support)

PSoC is a registered trademark of Cypress Semiconductor Corp. All other trademarks or registered trademarks referenced herein are the property of their respective owners.



Cypress Semiconductor    Phone : 408-943-2600  
198 Champion Court    Fax : 408-943-4730  
San Jose, CA 95134-1709    Website : [www.cypress.com](http://www.cypress.com)

© Cypress Semiconductor Corporation, 2011-2015. The information contained herein is subject to change without notice. Cypress Semiconductor Corporation assumes no responsibility for the use of any circuitry other than circuitry embodied in a Cypress product. Nor does it convey or imply any license under patent or other rights. Cypress products are not warranted nor intended to be used for medical, life support, life saving, critical control or safety applications, unless pursuant to an express written agreement with Cypress. Furthermore, Cypress does not authorize its products for use as critical components in life-support systems where a malfunction or failure may reasonably be expected to result in significant injury to the user. The inclusion of Cypress products in life-support systems application implies that the manufacturer assumes all risk of such use and in doing so indemnifies Cypress against all charges.

This Source Code (software and/or firmware) is owned by Cypress Semiconductor Corporation (Cypress) and is protected by and subject to worldwide patent protection (United States and foreign), United States copyright laws and international treaty provisions. Cypress hereby grants to licensee a personal, non-exclusive, non-transferable license to copy, use, modify, create derivative works of, and compile the Cypress Source Code and derivative works for the sole purpose of creating custom software and or firmware in support of licensee product to be used only in conjunction with a Cypress integrated circuit as specified in the applicable agreement. Any reproduction, modification, translation, compilation, or representation of this Source Code except as specified above is prohibited without the express written permission of Cypress.

Disclaimer: CYPRESS MAKES NO WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, WITH REGARD TO THIS MATERIAL, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. Cypress reserves the right to make changes without further notice to the materials described herein. Cypress does not assume any liability arising out of the application or use of any product or circuit described herein. Cypress does not authorize its products for use as critical components in life-support systems where a malfunction or failure may reasonably be expected to result in significant injury to the user. The inclusion of Cypress' product in a life-support systems application implies that the manufacturer assumes all risk of such use and in doing so indemnifies Cypress against all charges. Use may be limited by and subject to the applicable Cypress software license agreement.