AN73503

PSoC® USB HID Bootloader

Authors: Robert Murphy, Keith Mikoleit

Associated Part Family: All PSoC 3, PSoC 4 L-series, and PSoC 5LP parts with USB

Related Code Examples and Other Documents: For a complete list, click here.

More code examples? We heard you.

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

AN73503 describes how to implement a USB bootloader for PSoC devices by using the USB Human Interface Device (HID) class. It also shows how to build a Windows-based USB bootloader host program.

Contents

1 Introduction

Bootloaders are a common part of MCU system design. A bootloader makes it possible for a product's firmware to be updated in the field. At the factory, initial programming of firmware into a product is typically done through the MCU's Joint Test Action Group (JTAG) or Serial Wire Debugger (SWD) interface. However, these interfaces are usually not available in the field.

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

In this application note, you will learn how to:

- Add a USB bootloader to a PSoC 3, PSoC 4 L-series, or PSoC 5LP device
- Prepare an application project for bootloading
- Use the bootloader host program provided with PSoC Creator
- Create your own Windows-based bootloader host program

This application note assumes that you are familiar with PSoC and the PSoC Creator IDE. If you are new to PSoC, you can find introductions in the Getting Started with PSoC application notes AN54181, AN79953, and AN77759. If you are new to PSoC Creator, see the PSoC Creator home page.

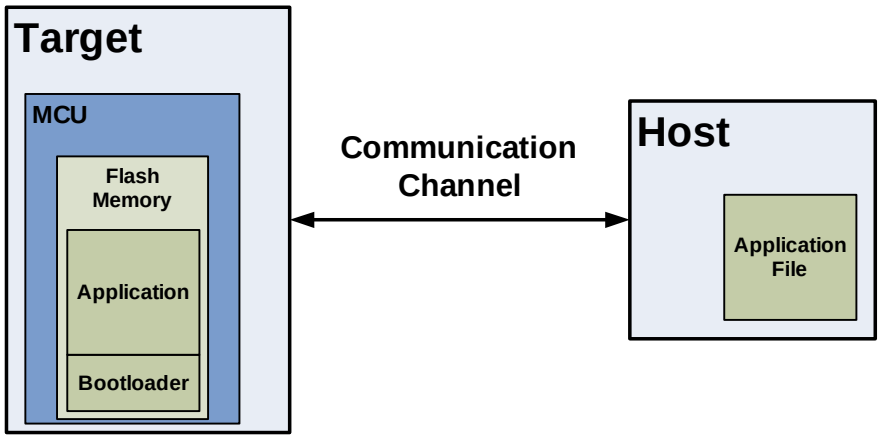
This application note also assumes that you are familiar with bootloader concepts. If you are new to these concepts, see application note AN73854, PSoC Introduction to Bootloaders.

Finally, this application note assumes that you are familiar with USB. If you are new to USB or the USB HID class, see application notes AN57294, USB 101: An Introduction to Universal Serial Bus 2.0 or AN57473, USB HID Basics with PSoC.

1.1 Terms and Definitions

Figure 1 shows that a product's embedded firmware uses the communication port for two different purposes – normal operation and updating the application. That portion of the embedded firmware that updates the application is called a **bootloader**.

Figure 1. Bootloader System Block Diagram



Typically, the system that provides the data to update the flash is called the **host**, and the system being updated is called the **target**. The host can be an external PC or another MCU on the same PCB as the target.

The act of transferring data from the host to the target flash is called **bootloading**, or a **bootload operation**, or just **bootload** for short. The data that is placed in flash is called the **application** or **bootloadable**.

Another common term for bootloading is **in-system programming (ISP)**. Cypress has a product with a similar name called In-System Serial Programmer (ISSP) and an operation called Host-Sourced Serial Programming (HSSP). For more information, see application notes AN73054 or AN84858, PSoC Programming Using an External Microcontroller (HSSP).

1.2 Using a Bootloader

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

When the bootloader is running, the host can send a "start bootload" command over the communication channel. If the target sends an "OK" response, bootloading can begin.

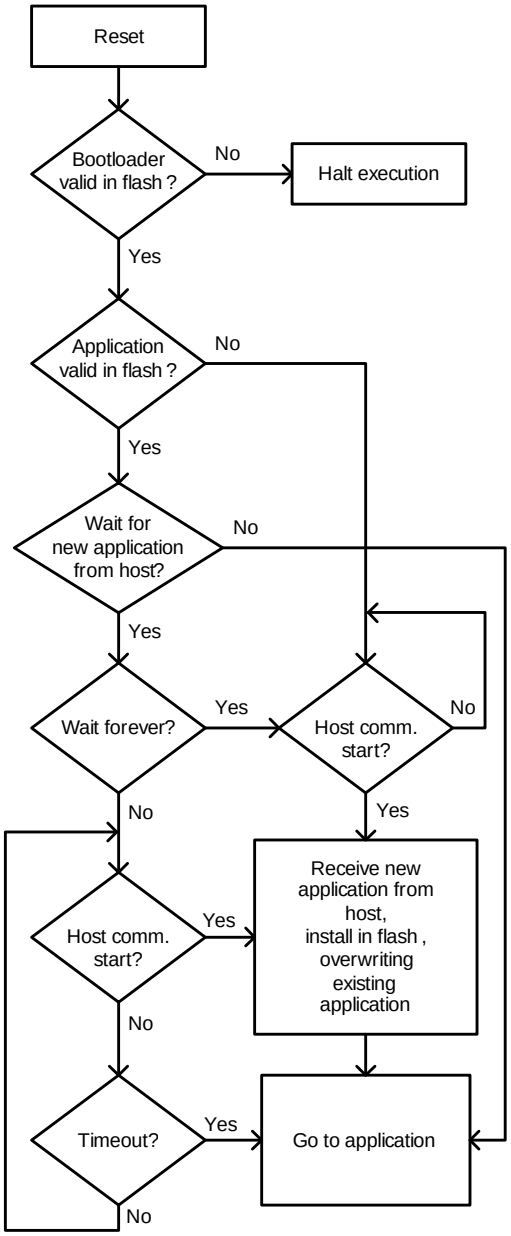
During bootloading, the host reads the file for the new application, parses it into flash write commands, and sends those commands to the bootloader. After the entire file is sent, the bootloader can pass control to the new application.

1.3 Bootloader Function Flow

A bootloader typically executes first at reset. It can then perform the following actions, as Figure 2 shows:

- Check the application's validity before letting it run
- Manage the timing to start host communication
- Do the bootload / flash update operation
- And finally, pass control to the application

Figure 2. Bootload Process Flowchart



1.4 USB Bootloader Considerations

There are some issues to consider when using a USB port as the bootloader communication interface. The USB HID class gives the benefit of not needing a driver, and the device can work with any host operating system.

Keep in mind that the bootloader waits for a limited amount of time before passing control to the application, and USB devices take time to enumerate. If USB enumeration takes too long, the host may miss the opportunity to initiate a bootload operation after target device reset.

Once the application is in control, there are some ways to pass control back to the bootloader.

1.4.1 Bootloadable API

The PSoC Creator Bootloadable Component has an API function to start the bootloader: `Bootloadable_Load()`. This enables the application to pass control back to the bootloader. This is a good way to allow a product user to initiate a firmware upgrade.

The problem with this method is that you now depend on the application code to perform an application upgrade. What happens if the application has a bug that doesn't allow successful transfer to the bootloader? To ensure the ability to upgrade and fix firmware, it is a good idea to put in a failsafe method for starting the bootloader in the bootloader project.

1.4.2 Launching Bootloader at Startup with an Infinite Wait Period

To address issues such as corrupt application or long USB enumeration times, you can add a method to stay in the bootloader until a host command is received. You can customize the bootloader project to check for some user input before calling Bootloader_Start() and running through its normal routine.

Refer to the Bootloader Firmware section for more information.

1.5 Code Example

The following sections show you the steps to create PSoC Creator bootloader and bootloadable projects. You can get the completed projects at code example CE95391.

2 Bootloader Project

A PSoC bootloader system consists of at least two PSoC Creator projects – one bootloader project and at least one bootloadable (application) project. This section shows how to create a bootloader project. The next section shows how create a bootloadable project.

A bootloader project design includes the PSoC Creator Bootloader and USBFS Components, as Figure 3 shows. The USBFS Component communicates with the PC host to get commands and a new application image. The Bootloader Component does flash programming, host command / response protocol, and launches the bootloadable application.

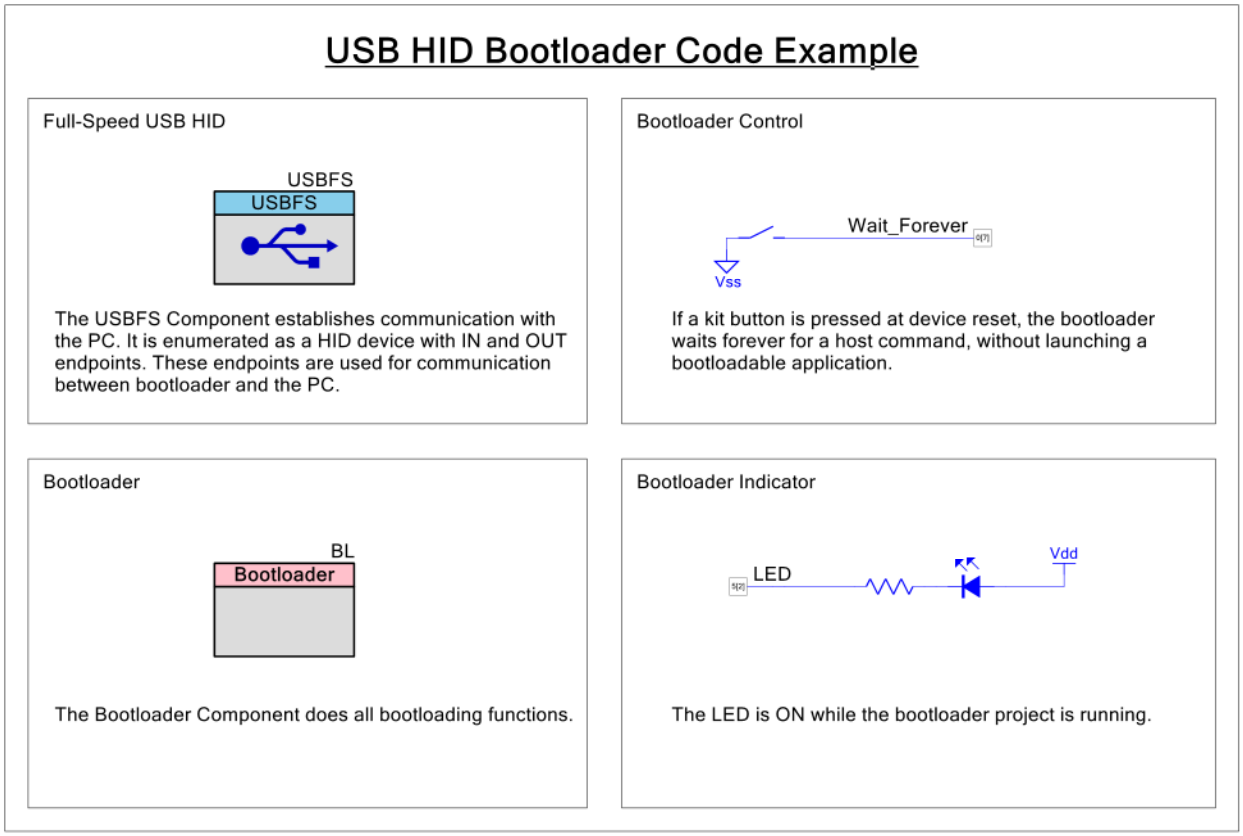
2.1 Create the Project

To start, create a new PSoC Creator project. Select your target hardware or device in the Create Project dialog.

Give the workspace a name such as "PSoCx_USB_Bootloader". Give the project a name such as “USB _Bootloader”.

Now add PSoC Creator Components – USBFS, Bootloader, and other Components – to the schematic. You may optionally rename the Components as Figure 3 shows.

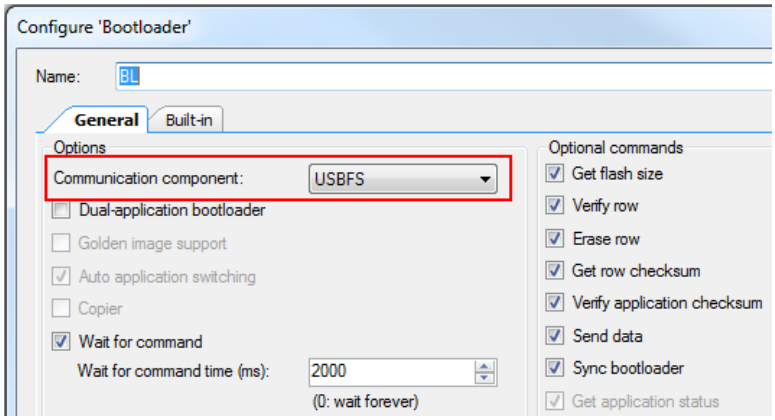
Figure 3. Bootloader Project Schematic from Code Example CE95391



2.2 Configure the Bootloader Component

The only required change to the Bootloader Component is to set the Communication Component to USBFS, as Figure 4 shows. This tells PSoC Creator to generate the functions required for the bootloader to interface with the USB.

Figure 4. Bootloader Component Customizer



You should also review the **Wait for command** settings. They are particularly important for USB bootloaders because you must consider USB device enumeration time.

- **Wait for command:** On device reset, the Bootloader can wait for a command from the host, or immediately jump to the application code. If this option is enabled, the Bootloader waits for a command from the host until the timeout period specified by the **Wait for command time** parameter. If the Bootloader does not receive a command from the host within the timeout interval, it jumps to the application code.
- **Wait for command time (ms):** If the above option is enabled, this parameter is the amount of time that the Bootloader waits before jumping to the application code. A zero value is interpreted as wait forever. The default value is a 2-second timeout.

If a valid application is not installed in the device flash, the Bootloader waits forever for a host command regardless of these settings.

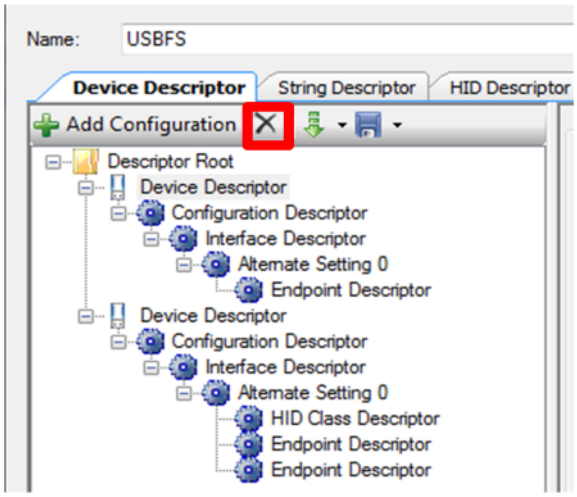
For complete information on the Bootloader Component settings, refer to the Bootloader Component datasheet.

2.3 Configure the USB Component

Next, configure the USBFS Component to support bootloader communication. Remember that we are using the USB HID class to avoid the need for externally provided driver files in the host PC.

To begin configuring the USBFS Component, double-click it. Then, in the configuration dialog, delete the default Device Descriptor root. Click the **Device Descriptor** tab, and then click the **'X'** button, as Figure 5 shows.

Figure 5. Delete Default USB Configuration

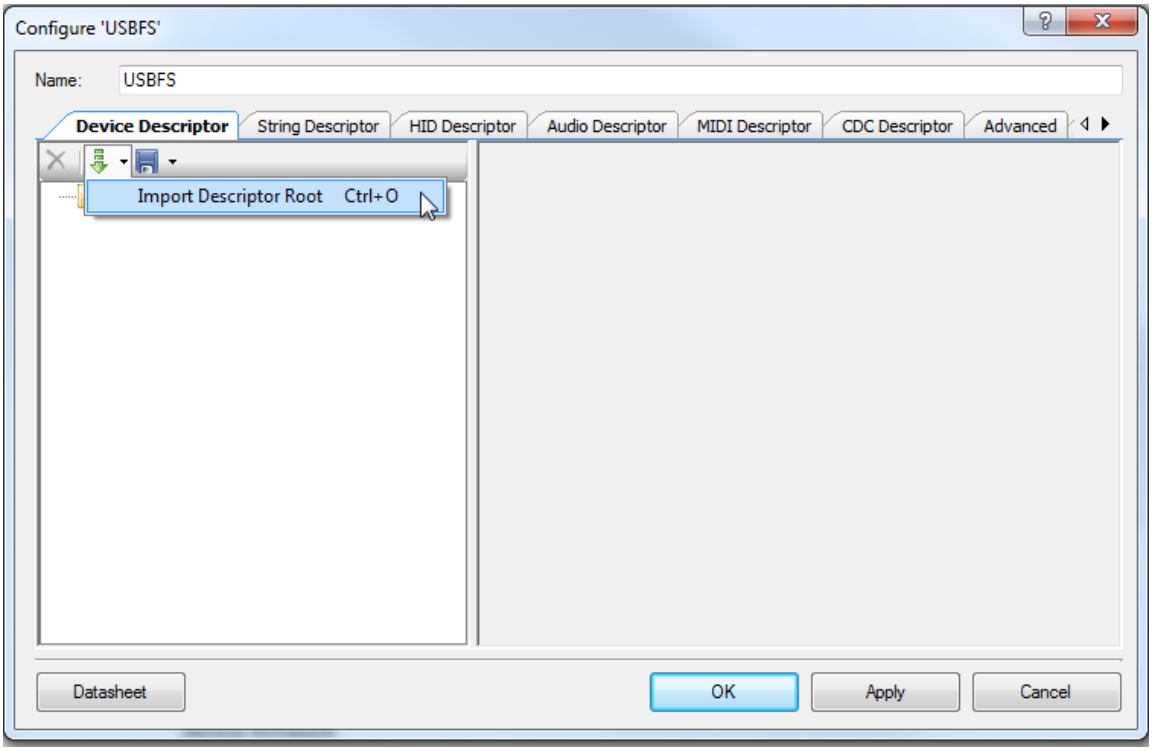


Then, use the Insert Configuration option to import a USB configuration for the bootloader, as Figure 6 shows. The complete USB configuration for the bootloader is saved as an *.xml* file that is included with PSoC Creator:

<PSoC Creator InstallDir> \ psoc \ content \ cycomponentlibrary \ CyComponentLibrary.cylib \ USBFS_v_<version of USBFS Component> \ Custom \ template \ Bootloader.root.xml

After the configuration is imported, PSoC Creator proceeds to load the required USB descriptors automatically. See Appendix A for USB configuration details.

Figure 6. USB Insert Configuration



2.4 Configure the Input Pin

As mentioned previously, it is a good idea to provide a backdoor option to start the bootloader. One way to do this is to check the state of a pin at startup. This works well if the end application has a user interface such as a switch.

Use the Digital Input Pin Component to read the state of a pin. In code example CE95391, the pin is connected to a kit button. The button shorts to ground when pressed, therefore the Pin Component is configured for resistive pull up. This causes the Pin input state to be 0 when the button is pressed and 1 when it is released.

2.5 Place Pins and Configure Clocks

After all of the Components are configured, set up the clocking resources and place the pins.

In the Workspace Explorer window, locate and double-click *USB_Bootloader.cydwr*. The Pin Selection tab appears. Assign the Wait_Forever and other Pin Components to the device physical pins used in your kit. See CE95391 for an example.

Note: The USB D+ and D- data lines are automatically assigned by PSoC Creator.

Note: You can assign a Pin Component in a bootloader project and a Pin Component in an application to the same physical pin. You can even use the same pin name, because the bootloader and application are separate projects.

Next, click the **Clocks** tab, and double-click one of the clocks to open the clock configuration wizard. Configure the clocks for USB, as Figure 7 shows for PSoC 4 L-series, and Figure 8 shows for PSoC 3 and PSoC 5LP.

Figure 7. PSoC 4 L-series USB Clock Configuration

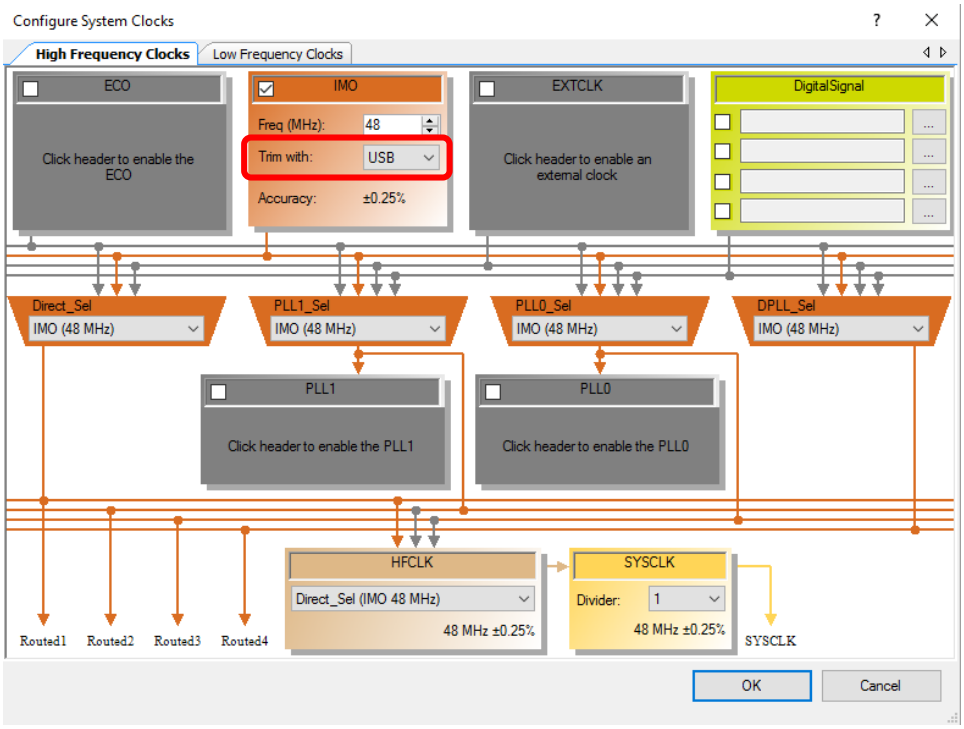
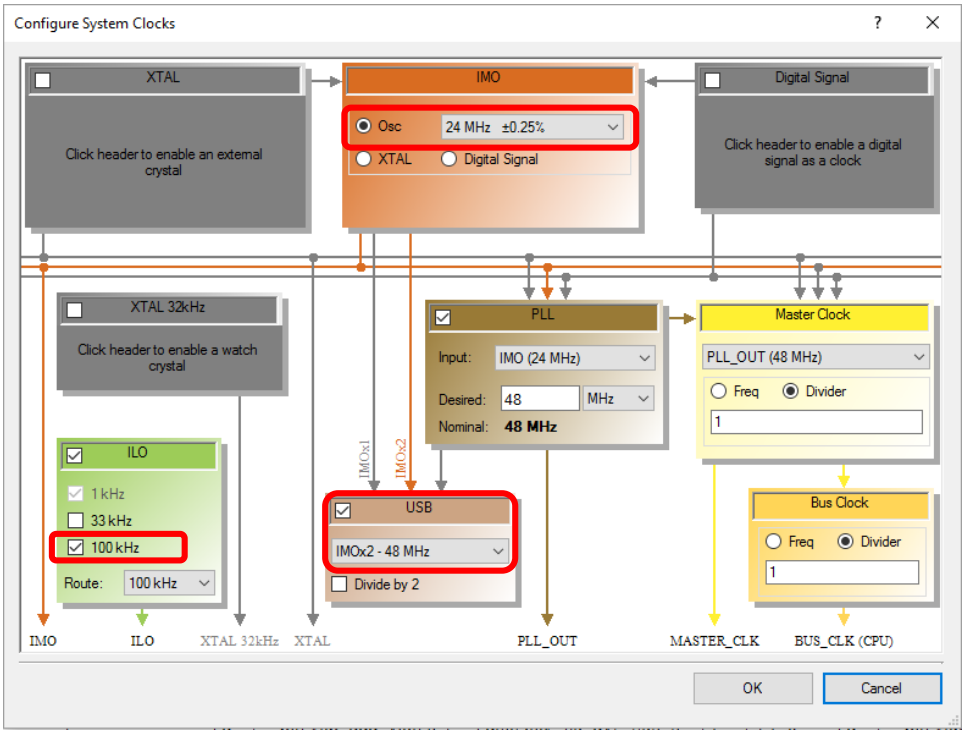


Figure 8. PSoC 3 and PSoC 5LP USB Clock Configuration



2.6 Bootloader Firmware

After all of the project Components are configured, add firmware to control the bootloader operation. (A recommended best practice is to do a project **Build** > **Generate Application** before adding your firmware.)

In many cases, all you have to do is add to your *main.c* a call to the Bootloader Component API function `Bootloader_Start()`. (Replace the "Bootloader" portion of the function call with the name of your Bootloader Component, if you changed it from the default.)

The `Bootloader_Start()` function does the entire bootload function. It does not return – it resets the PSoC device before transferring control to the application. For more information on the bootloader and device reset, see Appendix B.

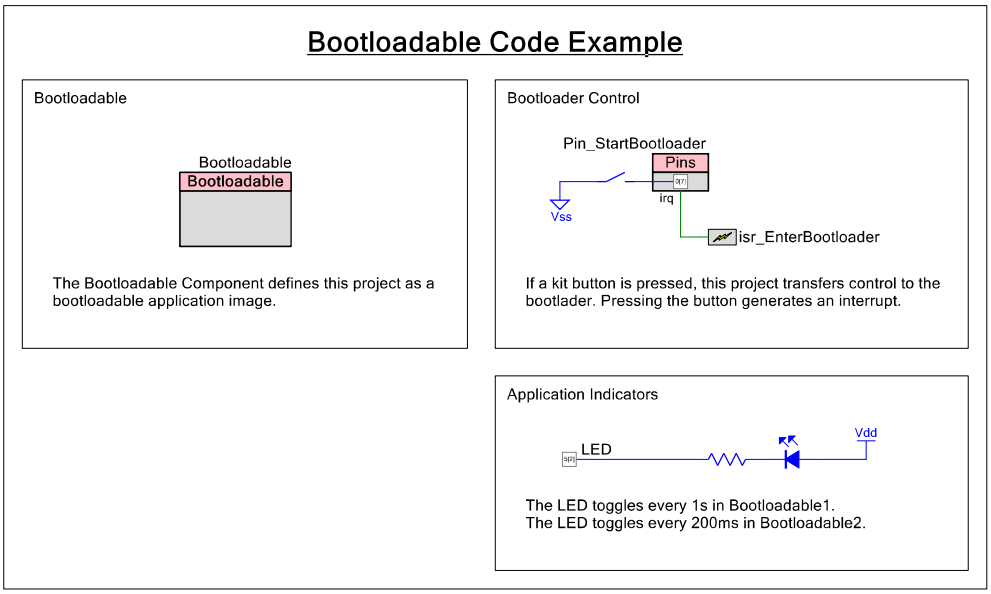
For a more complex example of how to use the bootloader indicator and control pins shown in Figure 3, see code example CE95391.

3 Bootloadable (Application) Project

A bootloadable project includes the PSoC Creator Bootloadable Component, as Figure 9 shows. Other Components implement the desired application.

Code example CE95391 has two bootloadable projects which blink a kit LED at different rates. The projects also read a kit button switch. When the switch is closed, the project transfers control to the bootloader.

Figure 9. Bootloadable Project Schematic



3.1 Create the Project

Create another new PSoC Creator project. Select your target hardware or device to be the same as the bootloader project. Give the project a name such as "USB _Bootloadable". You can add it to the bootloader project's workspace or save it in a different workspace.[1]

Now add PSoC Creator Components – Bootloadable and other Components – to the schematic. You may optionally rename the Components as Figure 9 shows.

[1] A PSoC Creator workspace can have multiple projects. In many cases, a bootloader project exists in the same workspace as its associated bootloadables. However, this doesn't have to be the case—bootloaders and bootloadables can exist in separate workspaces and separate locations on your PC. Before getting started with PSoC, it is a good idea to work out a workspaces / projects plan for your overall system development needs.

3.2 Configure the Bootloadable Component

A bootloadable project always has a dependency on the output *.hex* and *.elf* files of an associated bootloader project, as Figure 10 shows. Selecting the *.hex* file automatically selects the associated *.elf* file.

Before configuring a Bootloadable Component, you should completely build the associated bootloader project.

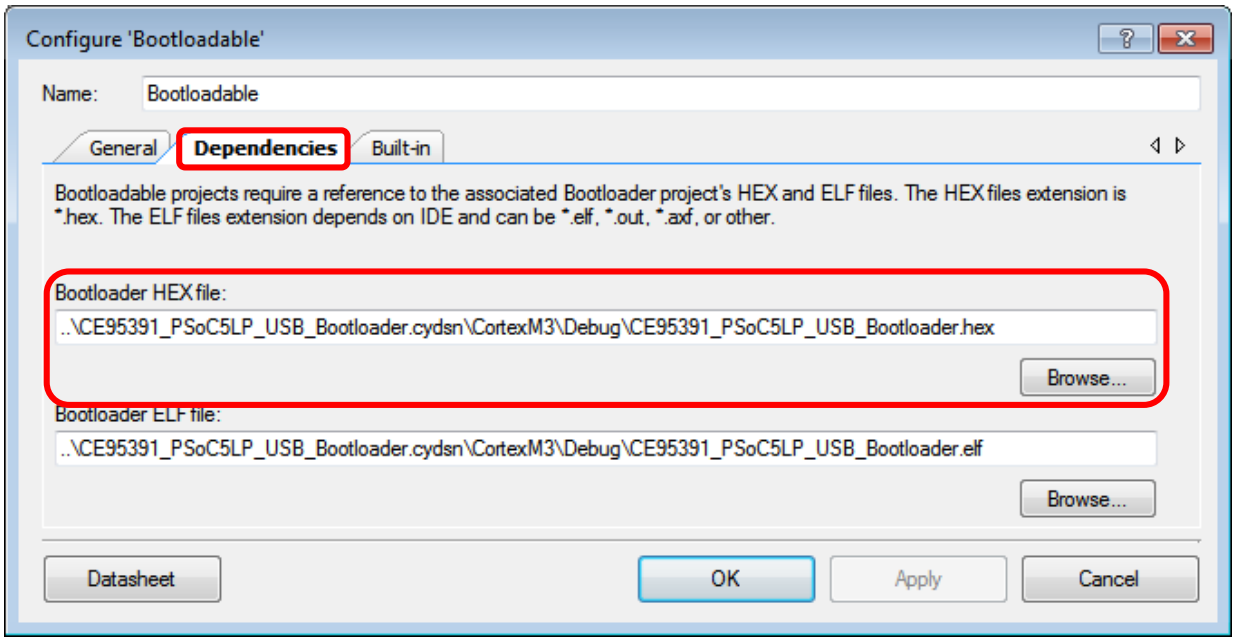
Figure 10 shows the bootloader *.hex* file at a relative path within a PSoC Creator workspace. However, this need not be the case; bootloader and bootloadable projects can be in different workspaces. In either case, find the *.hex* file in the folder for the bootloader project's compiler output:

<project folder>\USB_Bootloader.cydsn\DP8051\DP8051_Keil_903\Debug\ (For PSoC 3)
<project folder>\USB_Bootloader.cydsn\CortexM0\Debug\ (For PSoC 4 L)

<project folder>\USB_Bootloader.cydsn\CortexM3\Debug\ (For PSoC 5LP)

For more information on bootloader and bootloadable files, see AN73854, PSoC Introduction to Bootloaders.

Figure 10. Bootloadable Component Configuration



3.3 Configure Remainder of Project

After the Bootloadable Component is configured, you can build the remainder of a bootloadable project in the same manner as a standard (non-bootloadable) project. Add other Components to the schematic as needed, configure them, and add your firmware. See code example CE95391 for an example.

You can create as many bootloadable projects as you want, and associate each of them with the previously created bootloader project. Then, after the bootloader project is installed in your target hardware, you can change the target function by downloading different bootloadable projects.

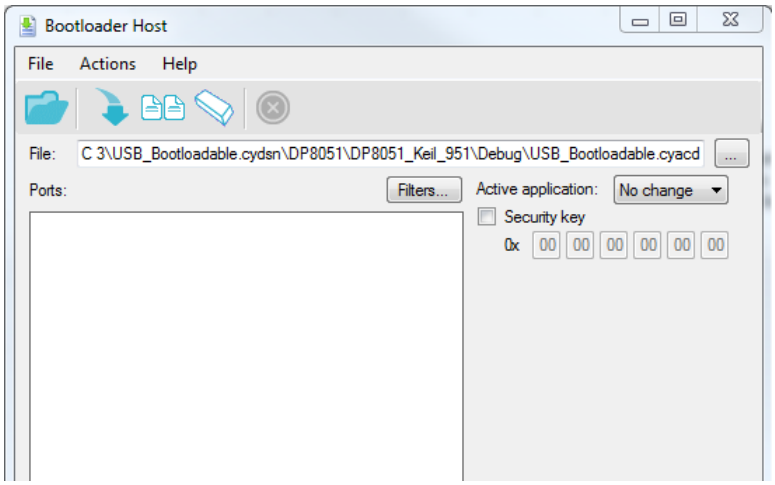
The remainder of this application note shows how to use the Bootloader Host program provided with PSoC Creator, and how to build your PC-based bootloader host.

4 PSoC Creator Bootloader Host

PSoC Creator provides a Bootloader Host program to facilitate use of your bootloader and bootloadable projects. Do the following:

1. Build your bootloader project, and program it into your target PSoC.
2. Open the Bootloader Host tool (Figure 11). Select PSoC Creator menu item **Tools** > **Bootloader Host**.

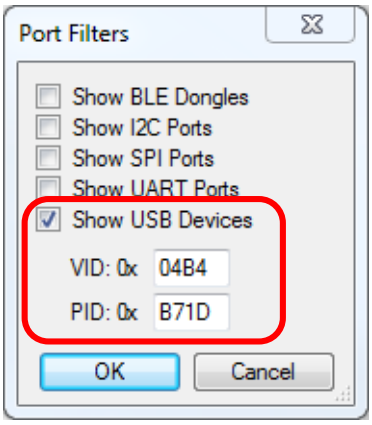
Figure 11. PSoC Creator Bootloader Host Program



3. Click **Filters** and add a filter to identify the USB device (i.e., your PSoC target) that you intend to bootload, as Figure 12 shows.

   Make sure that the Vendor ID (VID) and Product ID (PID) match those defined in Configure the Bootloader Component.

Figure 12. USB Filter Settings



4. Click the **Open File** icon, and browse to the location of your bootloadable file. The file is of type *.cyacd*, and is in the folder for the bootloadable project's compiler output.

   For more information on bootloader and bootloadable files, see AN73854, PSoC Introduction to Bootloaders.

5. Click **Program** or press the F5 key to start bootloading.

After bootloading is complete, control is passed from the bootloader to your bootloadable application, after a possible timeout wait by the bootloader.

5 Build a Bootloader Host

This section shows how to create a graphical user interface (GUI) application for Windows that implements a custom USB bootloader host. Only the key steps are described here. If you are unfamiliar with Windows application development or Microsoft Visual Studio, refer to Related Resources.

A completed bootloader host application Visual Studio project, *USBBootloaderHost_VC2015.zip*, is attached to this application note, for reference. You can also copy and paste code from the project source files to your project.

5.1 Required Resources

You need a few things to create the bootloader host:

5.1.1 Visual Studio

Visual Studio editions provide variety of development tools and support for programming languages such as C# and C++.

Note that the instructions in this application note are for Visual Studio Community 2015. Other versions of Visual Studio may require different steps.

5.1.2 Visual C# Express 2015

This is Microsoft's free IDE for developing .NET applications using the C# programming language. The instructions in this application note are for the free version; you can also use the full version (Visual Studio).

5.1.3 Visual C++ Express 2015

This is another IDE from Microsoft for developing .NET applications; it uses the C++ programming language. In this application note, Visual C++ Express is used to generate a Dynamic Link Library (DLL), using C modules.

5.1.4 CYUSB.dll

CYUSB.dll is a Cypress developed and maintained .NET dynamic-link library (DLL) for interfacing Visual Studio applications with Cypress USB devices. This DLL is bundled with the Cypress SuiteUSB package, which is a complete set of USB development tools for Visual Studio. Download SuiteUSB from Cypress at Cypress SuiteUSB.

5.1.5 Bootloader APIs Provided with PSoC Creator

The four API modules used to create the host program are included with PSoC Creator in:

<install folder> \ *PSoC Creator* \ *<PSoC Creator Version>* \ *PSoC Creator* \ *cybootloaderutils*.

These modules include all of the code required for host-side interface with a PSoC Creator Bootloader Component, using the Cypress bootloader command / response protocol. For more information on this protocol, see the Bootloader Component datasheet or the System Reference Guide.

There are four modules, each of which is a .c / .h file pair:

- cybtldr_api.c / .h
  This module contains low-level functions to start a bootload operation, program a flash row, erase a row, verify a row, and end the bootload operation.
- cybtldr_api2.c / .h
  This module is a higher-level API that manages the entire bootload process. It has functions to program the device, erase the device, verify the device, and abort the bootload operation.
- cybtldr_command.c / .h
  This module constructs command packets to send to the bootloader, and parses response packets from the bootloader.
- cybtldr_parse.c / .h
  This module parses the **.cyacd* file that contains the bootloadable image to send to the device.

5.2 Create the Bootloader Host Application

This process has the following key stages:

Stage 1: Create a DLL (dynamic-link library) with the bootloader utility functions provided with PSoC Creator.

Stage 2: Create a C# Windows form application (i.e., a GUI).

Stage 3: Define the communication functions.

Stage 4: Import essential bootloader functions from the DLL created in Stage 1.

Stage 5: Modify form functions.

Stage 6: Define host error codes.

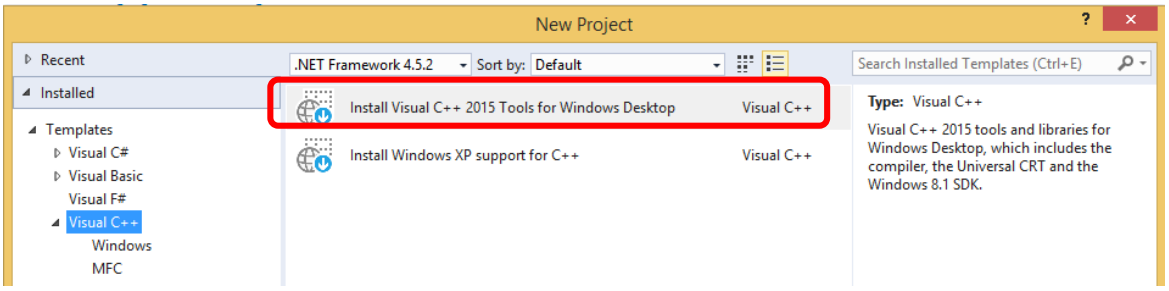
The following sections explain each stage in detail:

5.2.1 Stage 1: Create a DLL

First, create a DLL with the PSoC Creator bootloader C files. For more information, see Appendix C – Host Core APIs. This step can be skipped by using the pre-built dll available in the example project.

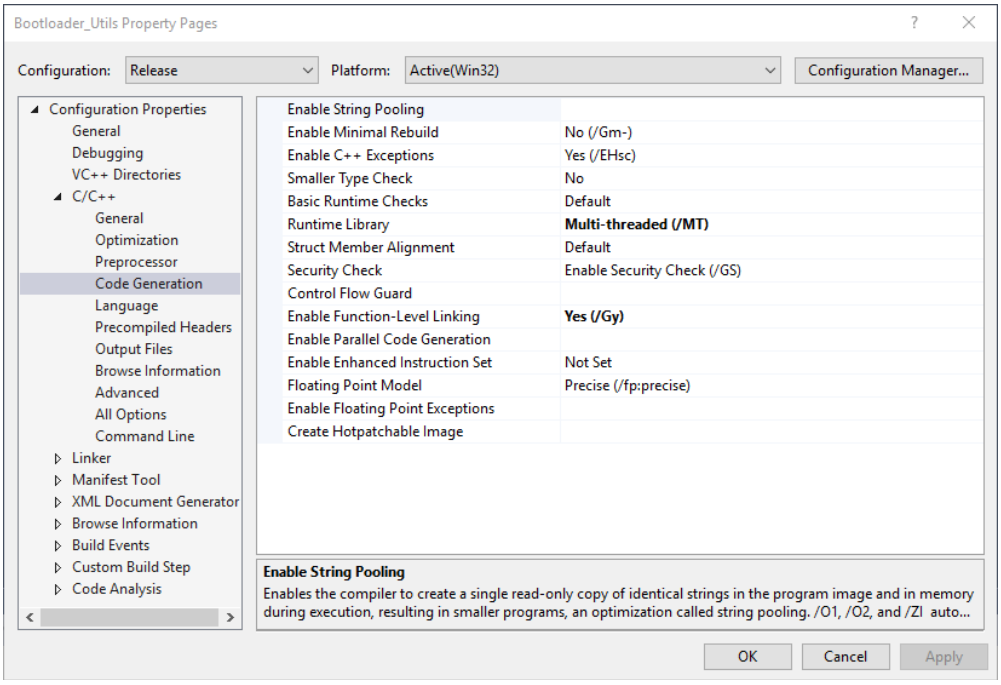
1. Run Visual Studio Express.
2. Create a new C++ DLL project.
   You may need to install Visual C++ 2015 Tools for Windows Desktop to see the Win32 console application, as Figure 13 shows. Select menu item **File** > **New** > **Project…** > **Visual C++** > **Win32 Console Application**.

Figure 13. Install Visual C++ 2015 Tools for Windows Desktop



3. Provide a suitable name to the project, such as Bootloader_Utils. Click **OK**, then **Next**.
4. Set **Application type** to DLL, and **Additional options** to Empty project. Click **Finish** to apply these settings and create a new project.
5. In the Solution Explorer window, right-click the project name (“Bootloader_Utils”) and select **Add** > **Existing Item…**. Add the following files, from the PSoC Creator folder *<install folder>* \ *PSoC Creator* \ *<PSoC Creator Version>* \ *PSoC Creator* \ *cybootloaderutils*:
   - cybtldr_api.h, cybtldr_api.c
   - cybtldr_api2.h, cybtldr_api2.c
   - cybtldr_parse.h, cybtldr_parse.c
   - cybtldr_command.h, cybtldr_command.c
   - cybtldr_utils.h
6. Right-click the Solution name (“Bootloader_Utils”), and click **Configuration Manager** to set the “Release” build option:
   Solution > **Configuration Manager…** > **Active solution configuration** > **Release**
7. Right-click the project name (“Bootloader_Utils”), and add a preprocessor definition to the project properties. Select menu item:
   Project Properties > **Configuration Properties** > **C/C++** > **Preprocessor** > **Preprocessor Definitions** > **Edit…**
   Insert “_CRT_SECURE_NO_WARNINGS” in the bottom of Preprocessor Definitions.
8. Change the Runtime Library option to “Multi-threaded (/MT)”, as Figure 14; otherwise, *msvcr100.dll* is required for using *Bootloader_Utils.dll.*
   Project Properties > **Configuration Properties** > **C/C++** > **Code Generation** > **Runtime Library >Multi-threaded (/MT)**

Figure 14. Runtime Library Option



9. Compile the project – select menu item **Build** > **Build Solution**. This creates the file *Bootloader_Utils.dll*.

5.2.2 Stage 2: Create a Windows Form Application

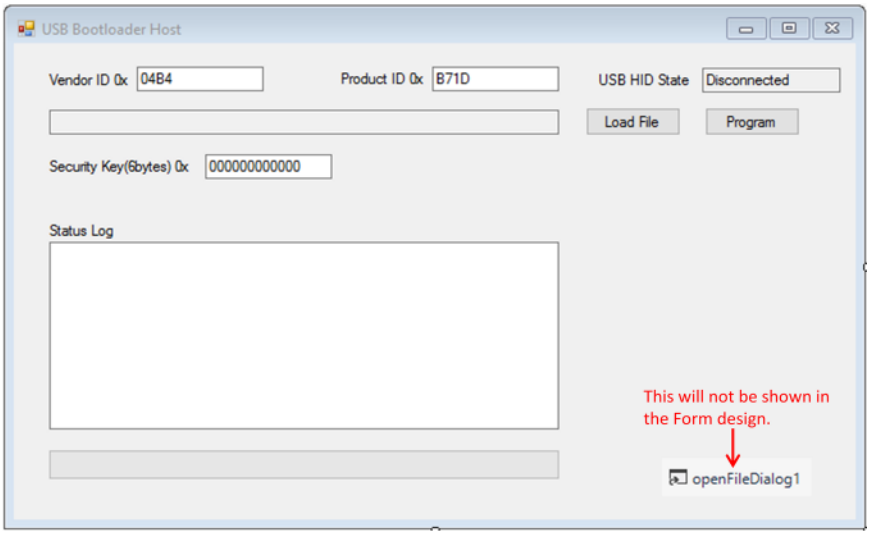
The next stage is to create a GUI for the bootloader host. You can reuse or modify the example GUI project instead of following all steps. The GUI is created using Visual C#. The main requirements are:

- Recognize and attach to an HID device, based on the device's VID and PID. For more information see the USB application notes in Related Resources.
- Place an input box for the user to enter a security key
- Allow the user to select a new *.cyacd* file to bootload onto the device
- Program the *.cyacd* file onto the device
- Use the x86 Platform target (**Project Properties** > **Build** = > **Platform target** > **x86**)
- Some systems may have a build error with .NET Framework 4.x, therefore use .NET Framework 3.5. Select the menu item **Project Properties** > **Application** > **Target framework** > **.NET Framework 3.5**.

Do the following:

1. Run Visual Studio.
2. Create a new C# project. Select menu item
   File > **New** > **Project…** > **Visual C#** > **Windows Forms Application**.
3. Provide a suitable name to the project, such as USBBootloaderHost, and click **OK**.
4. Design the form. Place 5 labels, 5 text boxes, 2 buttons, 1 progress bar and 1 openFileDialog from the toolbox onto the Main Form, as Figure 15 shows. Give the proper names and default values.

Figure 15. Custom Bootloader Host GUI



label1.Text = “Vendor ID 0x”
label2.Text = “Product ID 0x”
label3.Text = “USB HID State”
label4.Text = “Security Key(6bytes) 0x”
label5.Text = “Status Log”
textBox1.Text = “04B4”
textBox2.Text = “B71D”
textBox3.Text = “Disconnected”
textBox3.ReadOnly = true
textBox4.Text = “000000000000”
textBox5.Text = “”, textBox5.Multiline = true
textBox6.Text = “”, textBox6.ReadOnly = true

5.2.3 Stage 3: Define the Communication Functions

The next stage is to import the CyUSB functions from *CyUSB.dll* into the C# application. Note that the CyUSB functions are considered to be unmanaged code because they are not created by a .NET framework. For detailed information on unmanaged code, see Secure Coding Guidelines for Unmanaged Code.

To import the functions, follow these steps:

1. Copy *CyUSB.dll* to your project folder. If Cypress SuiteUSB is installed, *CyUSB.dll* is in:
   C:\Cypress\Cypress Suite USB 3.4.7\CyUSB.NET\lib.
2. Right click the project name or References, and select CyUSB.dll:
   References > **Add Reference…** > **Browse…** > **CyUSB.dll**
3. Check the check box in front of CyUSB.dll, then click **OK**.
4. Add a new class file: Right click the project name, then select **Add** > **New Item…** > **Class**. Give the class file a proper name such as *Bootloader_Utils.cs*.
5. Add code to *Bootloader_Utils.cs*, as Code 1 on page 15 shows. You can instead copy and paste code from the *Bootloader_Utils.cs* in the completed bootloader host Visual Studio project, *USBBootloaderHost_VC2015.zip*, which is attached to this application note.

5.2.4 Stage 4: Import the Bootloader Functions

The next stage is to import the Bootloader functions from the DLL (*Bootloader_Utils.dll*) into the C# application. This is accomplished using the **dllimport** modifier; see Code 1.

1. Add code to *Bootloader_Utils.cs*, as Code 1 shows. You can instead copy and paste code from the *Bootloader_Utils.cs* file in the completed bootloader host application Visual Studio project, *USBBootloaderHost_VC2015.zip*, which is attached to this application note.
2. Copy *BootLoader_Utils.dll*, created in stage 1, to your project output folder: ..\bin\x86\Debug

5.2.5 Stage 5: Modify Form Functions

Code for several events and processes must be added to the form. These include:

- USB HID connection and communication setup
- File management
- Display progress

It is easiest to copy and paste code from the *Form1.cs* file in the completed bootloader host application Visual Studio project, *USBBootloaderHost_VC2015.zip*, which is attached to this application note. You can then modify the code for your application.

5.2.6 Stage 6: Define Host Error Codes

Create a class file and give it a proper name such as *Bootloader_enum.cs*. The enum definitions must match those in *cybtldr_utils.h*. It is easiest to copy and paste code from the *Bootloader_enum.cs* file in the completed bootloader host application Visual Studio project, *USBBootloaderHost_VC2015.zip*, which is attached to this application note.

Code 1. Bootloader_Utils.cs

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Runtime.InteropServices;

namespace USBBootloaderHost
{
    class Bootloader_Utils
    {
        // Communication control structure
        [StructLayout(LayoutKind.Sequential)]
        public struct CyBtldr_CommunicationsData
        {
            public OpenConnection_USB OpenConnection;
            public CloseConnection_USB CloseConnection;
            public ReadData_USB ReadData;
            public WriteData_USB WriteData;
            public uint MaxTransferSize;
        };

        // Unmanaged code handling
        [UnmanagedFunctionPointer(CallingConvention.Cdecl)]
        public delegate int OpenConnection_USB();

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

        [UnmanagedFunctionPointer(CallingConvention.Cdecl)]
        public delegate int ReadData_USB(IntPtr buffer, int size);

        [UnmanagedFunctionPointer(CallingConvention.Cdecl)]
        public delegate int WriteData_USB(IntPtr buffer, int size);

        // DLL importing
        [UnmanagedFunctionPointer(CallingConvention.Cdecl)]
        public delegate int CyBtldr_ProgressUpdate(byte arrayID, ushort rowNum);
        [DllImport("BootLoader_Utils.dll", CallingConvention =
                   CallingConvention.Cdecl)]
        public static extern int CyBtldr_Program(string file, byte[] securityKey,
                                                 byte appId,
                                                 ref CyBtldr_CommunicationsData comm,
                                                 CyBtldr_ProgressUpdate update);
    }
}
```

6 Summary

Bootloaders are a useful way to do product field upgrades. Every bootloader needs a hardware communication interface to interact with a host. USB is widely available as a communication interface.

The PSoC USB HID bootloader described in this application note and in code example CE95391 provides a reliable solution to get you up and running quickly. The USBFS Component is configured as a HID class so that it can interface with any operating system with no custom driver.

A Windows Bootloader Host program is provided with PSoC Creator. This application note also shows how to use files provided by Cypress to create your own Windows bootloader host application.

7 Related Resources

Code Example

- CE95391 – USB HID Bootloader Code Example

Application Notes

- AN54181 – Getting Started with PSoC 3
- AN79953 – Getting Started with PSoC 4
- AN77759 – Getting Started With PSoC 5LP
- AN73854 – PSoC 3, PSoC 4, and PSoC 5LP Introduction to Bootloaders
- AN60317 – PSoC 3 and PSoC 5LP I^2C Bootloader
- AN86526 – PSoC 4 I^2C Bootloader
- AN68272 – PSoC 3, PSoC 4, and PSoC 5LP UART Bootloader
- AN84401 – PSoC 3 and PSoC 5LP SPI Bootloader
- AN57473 – USB HID Basics with PSoC 3 and PSoC 5LP (Fundamentals with Mouse and Joystick)
- AN58726 – USB HID Intermediate with PSoC 3 and PSoC 5LP (with Keyboard and Composite Device)
- AN56377 – PSoC 3 and PSoC 5LP - Introduction to Implementing USB Data Transfers

Additional Information

- SuiteUSB - USB Development tools for Visual Studio
- EZ-USB FX3 Software Development Kit

About the Authors

Name: Robert Murphy

Title: Applications Engineer Staff

Background: Robert Murphy graduated from Purdue University with a Bachelor's Degree in Electrical Engineering Technology.

Name: Keith Mikoleit

Title: Applications Engineer Sr.

Background: Keith Mikoleit graduated from Western Washington University with a Bachelor's Degree in Electrical Engineering Technology.

A Appendix A – USBFS HID Configuration

This section shows step-by-step how to set up a USB bootloader descriptor. Note that you can import the descriptor from the bootloader template file *bootloader.root.xml*, as described in Configure the USB Component.

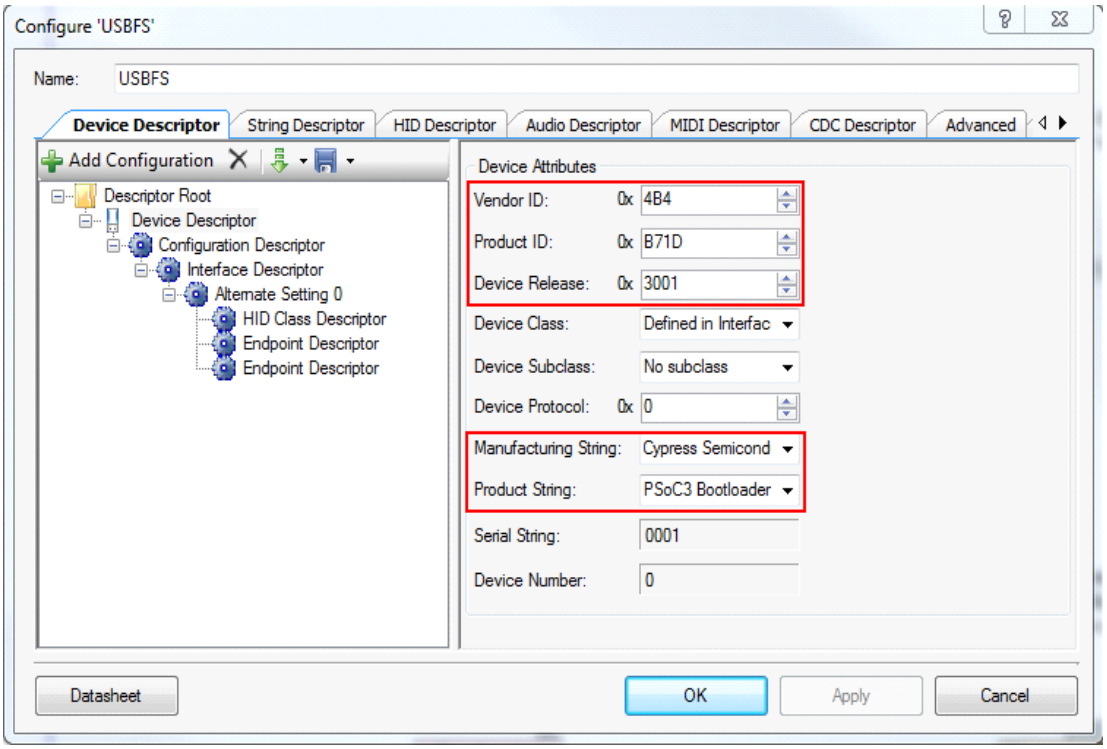
Note that this application note uses the USB HID class to avoid the need for externally provided driver files.

To begin configuring the USBFS Component, double-click the Component symbol in your PSoC Creator project schematic.

A.1 Create Device Descriptor

1. In the **Descriptor** tree, click **Device Descriptor**. Configure the options for the **Device Attributes**, as Figure 16 shows:
   - **Vendor ID**: 0x04B4
   - **Product ID**: 0xB71D
   - **Device Release**: 0x0101
   - **Manufacturing String**: Cypress Semiconductor
   - **Product String**: PSoCx Bootloader

Figure 16. USBFS Device Descriptor

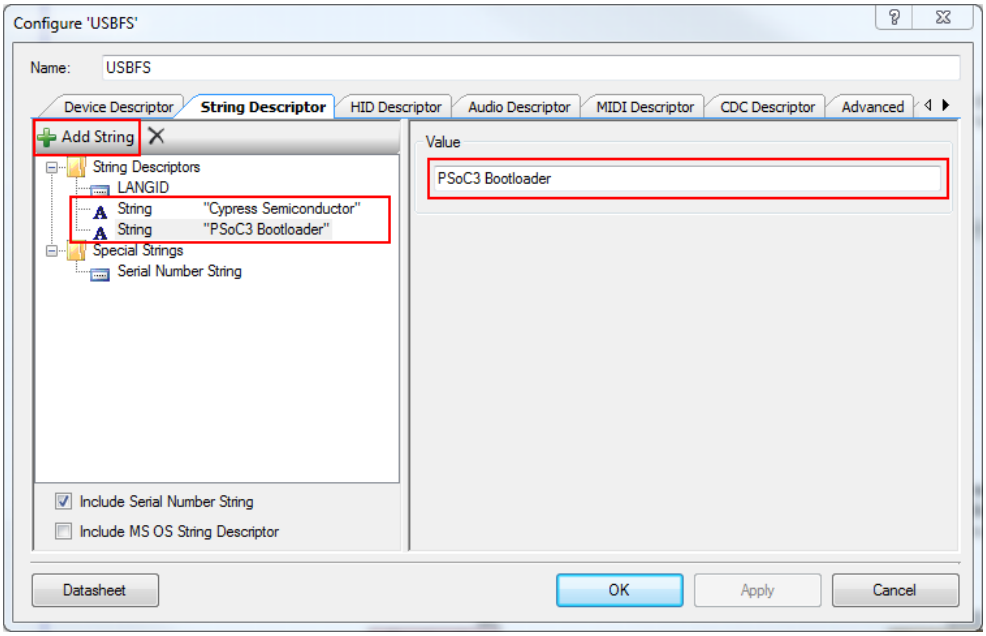


The only hard requirement is to change the Vendor ID (VID) and Product ID (PID). The VID used (0x04B4) is a specific Cypress Semiconductor VID. It is acceptable to use for this example. However, you must use a VID assigned to your company when you develop an application for production. The PID chosen is unique to this application. The bootloader host application uses the VID and PID to recognize the device.

You can optionally change strings such as the Manufacturing String and the Product String.

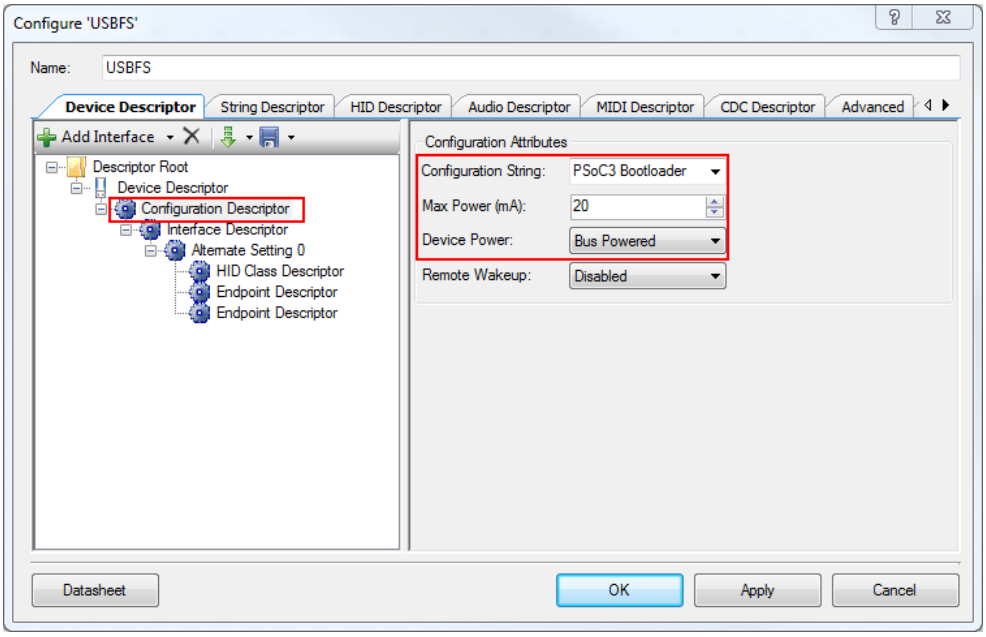
2. In the **String Descriptor** tree, click **Add String**. Add strings as Figure 17 shows:

Figure 17. String Descriptors



3. In the **Device Descriptor** tree, click **Configuration Descriptor**. Configure the options for **Configuration Attributes** as Figure 18 shows:

Figure 18. USBFS Configuration Descriptor



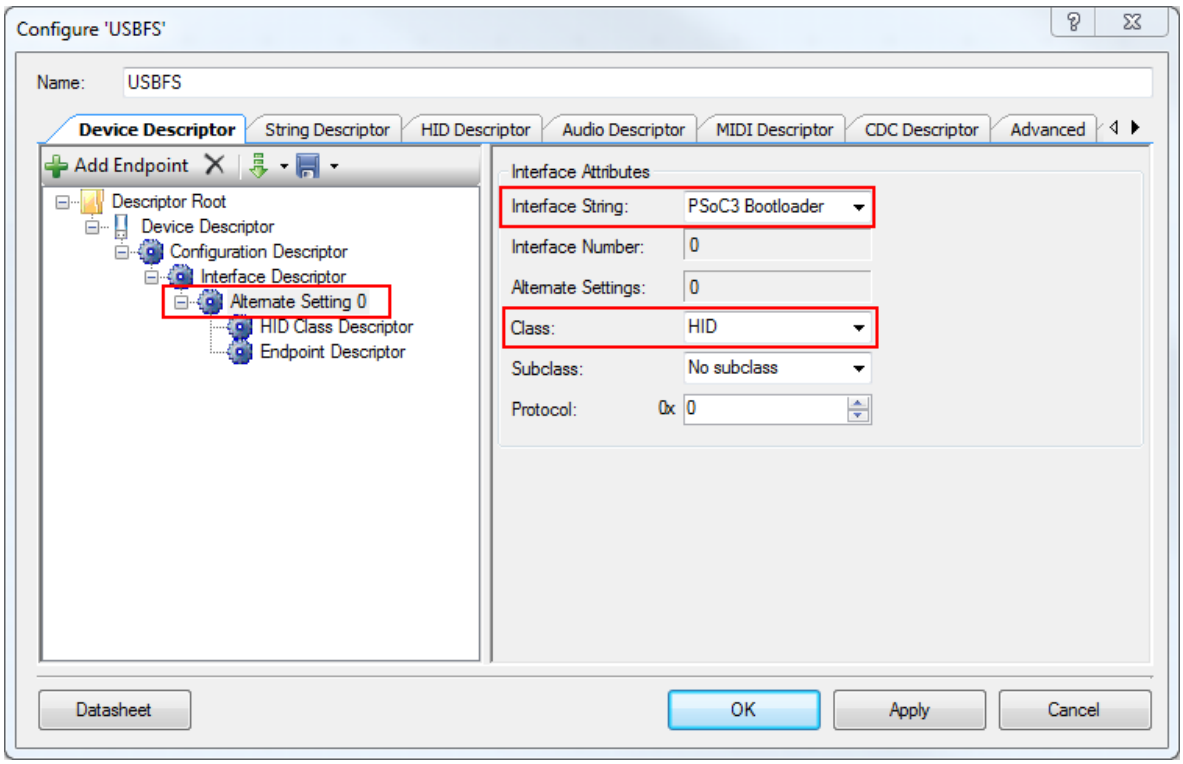
The key settings in this step are to define the USB device as Bus Powered, and to request a power budget of 20 mA from the host. Less than 20 mA for the application is acceptable, but you cannot exceed this requirement.

Remote Wakeup functionality is not required, and therefore disabled.

4. In the **Interface Descriptor** tree, click **Alternate Setting 0**. Configure the **Interface Attributes** as Figure 19 shows:

- **Interface String**: PSoCx Bootloader
- **Class**: HID
- **Subclass**: No subclass

Figure 19. USBFS Alternate Setting



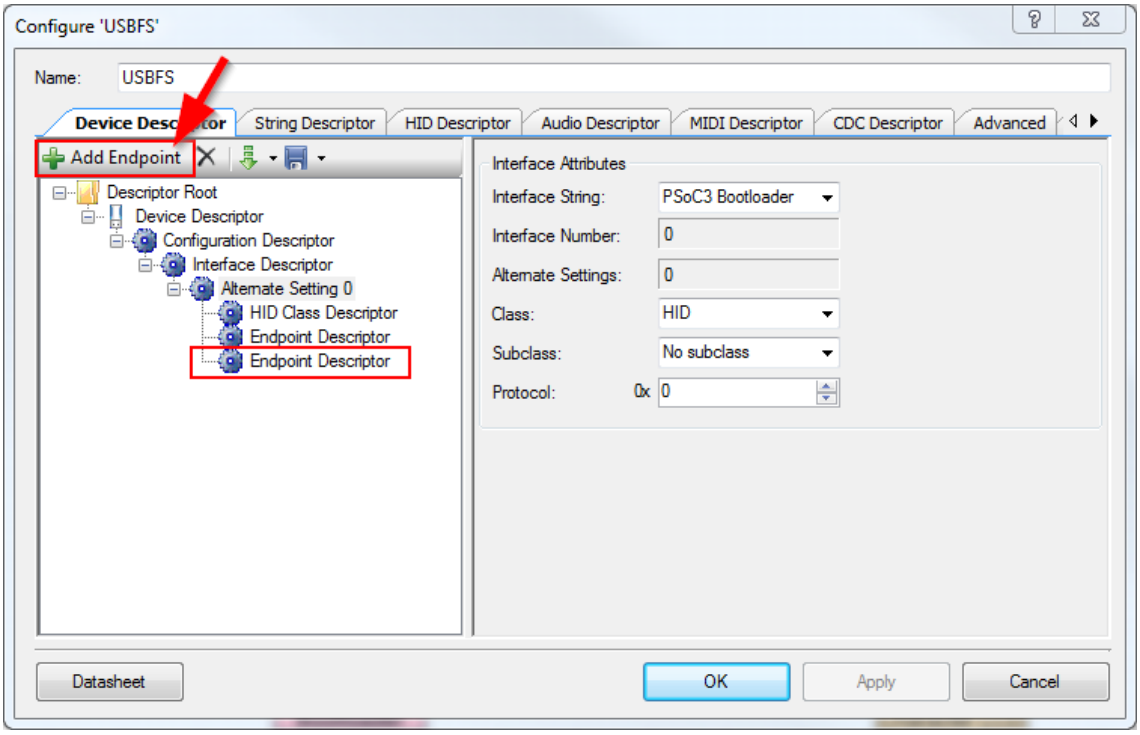
Each interface can have multiple Alternate Settings, for multiple endpoint configurations. Here you only need one alternate setting (the default one).

This is also where you specify that the device conforms to the HID class. Notice that as you change the Class from **Undefined** to **HID**, an additional Descriptor appears in the Descriptor Root tree: the HID Class Descriptor.

Note that you still have not set up the HID report. If you click OK or Apply at this point you will receive an error because no HID report is defined. The creation of this report is covered in the next few steps.

5. Click the **Add Endpoint** button. An additional **Endpoint Descriptor** appears in the **Alternate Setting 0** tree, as Figure 20 shows.

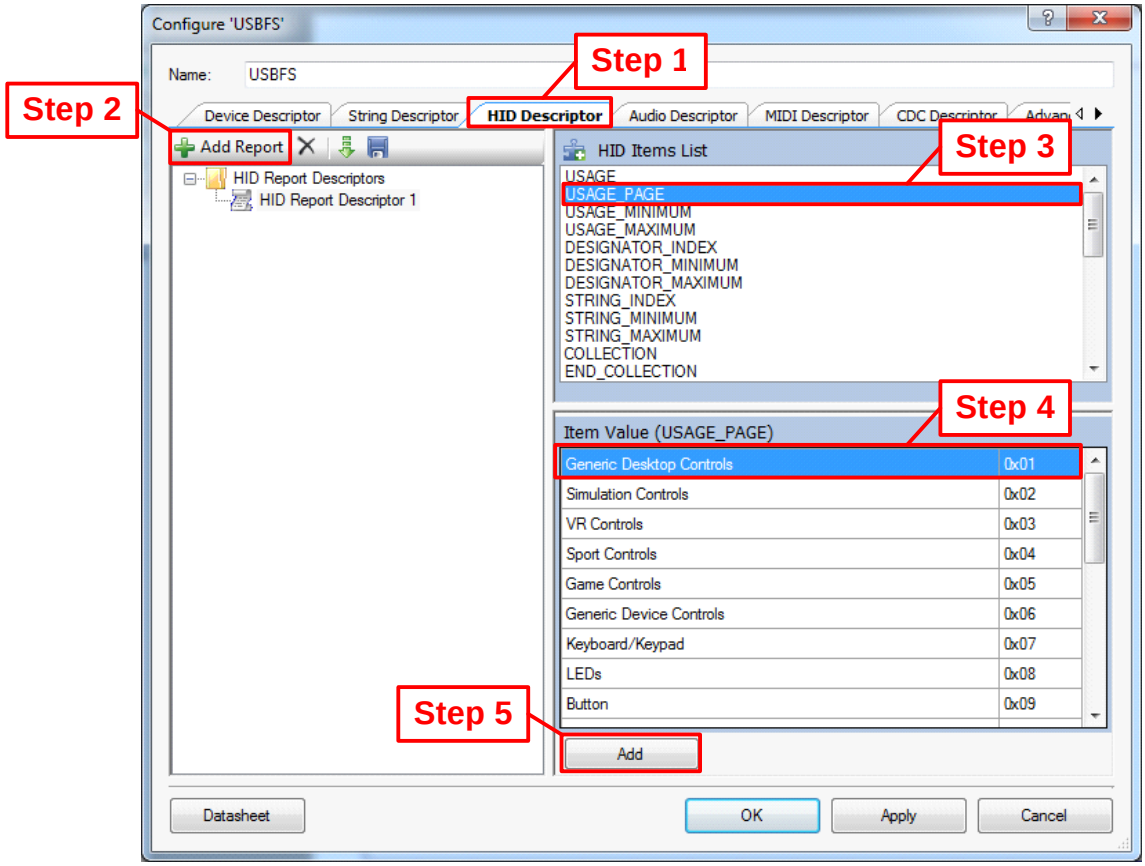
Figure 20. USBFS Adding Endpoints



A.2 Create HID Descriptor

1. Use the **HID Descriptor** interface to create an HID report descriptor, as Figure 21 shows:

Figure 21. HID Descriptor



a. Click the **HID Descriptor** tab.

b. Click the **Add Report** button.

   Follow Table 1 on page 22 for steps 3 to 5. The report items must be added in the order presented in the table.

c. Select an item from the **HID Item List**.

d. Select a value from the **Item Value** list according to the table.

e. When the Item Value box contains a text field for a particular Item selected, click either the Decimal or Hexadecimal radio option and enter the desired value in the field.

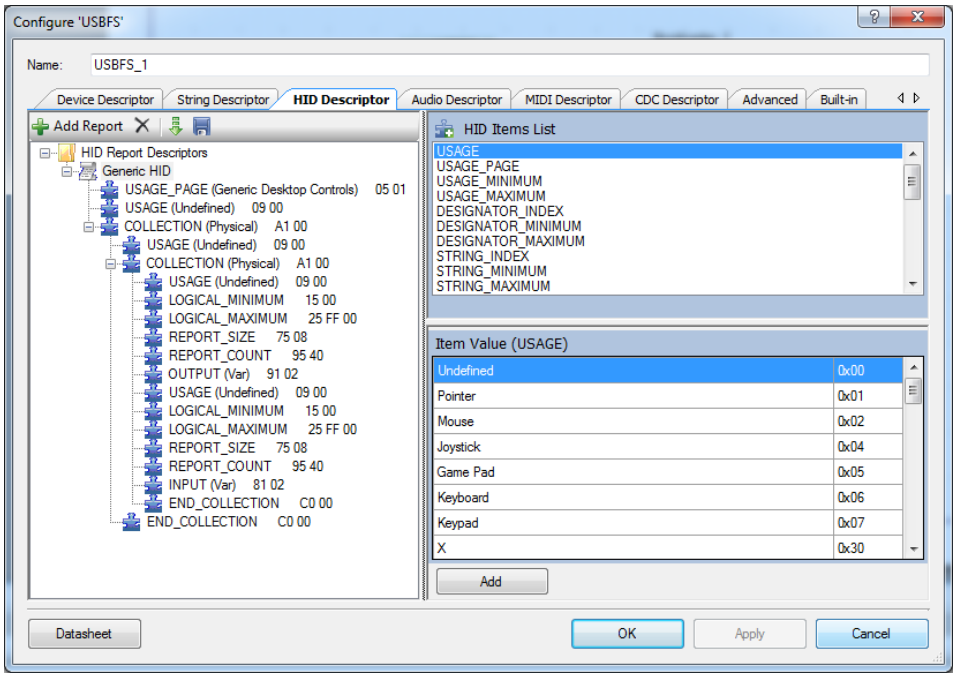
f. Repeat steps 3 to 5 until the report descriptor resembles the screenshot in Figure 22 on page 22.

g. Change HID description name to “Generic HID”.

Table 1. HID Descriptor Items

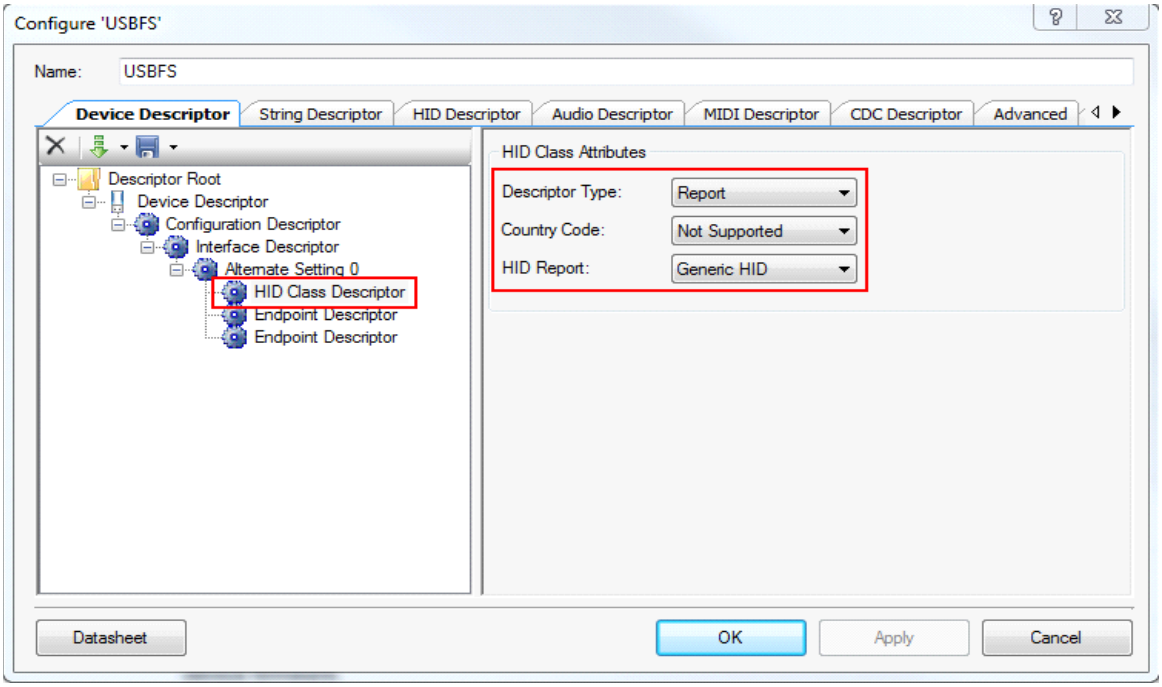
HID Item	Item Value (USAGE)
USAGE PAGE (05)	Generic Desktop Controls 0x01
USAGE (09)	Undefined 0x00
COLLECTION (A1)	Physical 0x00
USAGE (09)	Undefined 0x00
COLLECTION (A1)	Physical 0x00
USAGE (09)	Undefined 0x00
LOGICAL_MINIMUM (15)	0x00
LOGICAL_MAXIMUM (25)	0xFF
REPORT_SIZE (75)	0x08
REPORT_COUNT (95)	0x40
OUTPUT (91)	Bit 1 – Variable
USAGE (09)	Undefined 0x00
LOGICAL_MINIMUM (15)	0x00
LOGICAL_MAXIMUM (25)	0xFF
REPORT_SIZE (75)	0x08
REPORT_COUNT (95)	0x40
INPUT (81)	Bit 1 – Variable
END_COLLECTION (C0)	NA
END_COLLECTION (C0)	NA

Figure 22. Completed HID Descriptor



2. Click the **Device Descriptor** tab. In the **Descriptor** tree, click **HID Class Descriptor**. Configure options for **Device Attributes**, as Figure 23 shows:
   - **Descriptor Type**: Report
   - **Country Code**: Not Supported
   - **HID Report**: Generic HID

Figure 23. USBFS HID Device Attributes



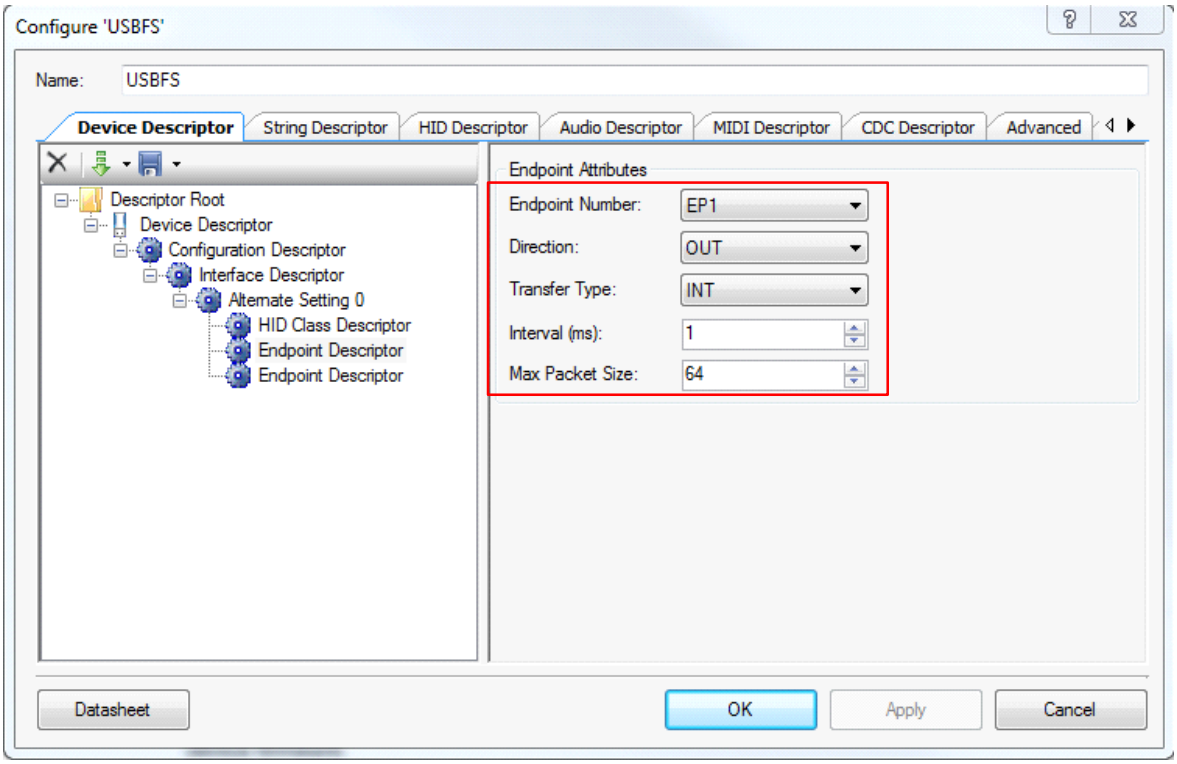
Since the project uses a Report Descriptor, set the Descriptor Type to Report.

Because the project is not specific to any country, set Country Code to Not Supported.

Finally, the HID Class Descriptor must point to the HID Report Descriptor created in the previous step. To create this link, match the HID Report to the HID Report Descriptor, which in this example is labeled Generic HID.

3. In the **Descriptor** tree, click the first **Endpoint Descriptor** entry. Configure options for **Endpoint Attributes** as Figure 24 shows:
   - **Endpoint Number**: EP1
   - **Direction**: OUT
   - **Transfer Type**: INT
   - **Interval**: 10
   - **Max Packet Size**: 64

Figure 24. USBFS Endpoint 1



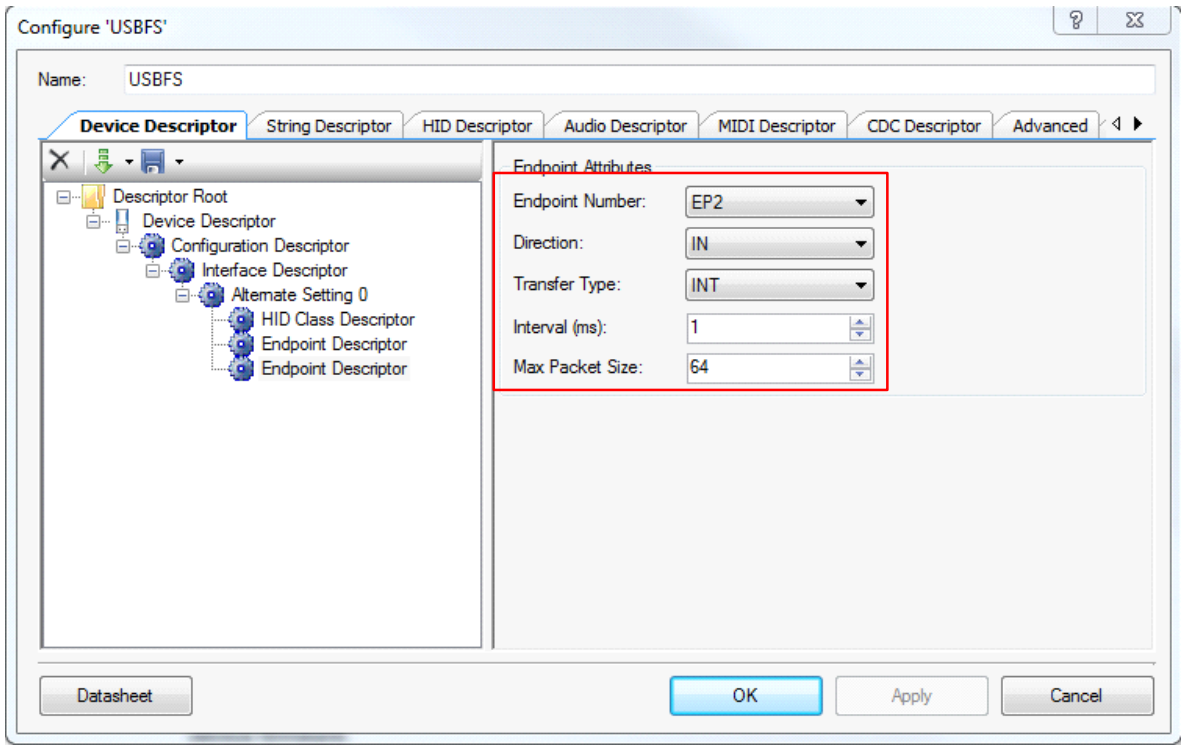
The first endpoint (EP) is used as the OUT Endpoint. It acts as a buffer for data received from the host. USB IN and OUT terminology is always relative to the USB host.

First, we define this endpoint as Endpoint 1 (EP1). Next, we set the direction of the endpoint as OUT.

Because this application is an HID, the specification requires that we use Interrupt (INT) transfers.

4. The purpose of EP2 is to act as a buffer for data that sent to the host. Repeat the previous step for the second endpoint to configure it as an IN endpoint with the following attributes (see Figure 25):
   - **Endpoint Number**: EP2
   - **Direction**: IN
   - **Transfer Type**: INT
   - **Interval**: 10
   - **Max Packet Size**: 64

Figure 25. USBFS Endpoint 2



5. Click the **Apply** button, and then click the **OK** button to close the configuration wizard.

B Appendix B – 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.

B.1 Why is Device Reset Needed?

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

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

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

B.2 Effect on PSoC 3 and PSoC 5LP Device I/O Pins

As described in application notes AN61290, PSoC 3, PSoC 5LP Hardware Design Considerations, and AN60616, PSoC 3 and PSoC 5LP Startup Procedure, during the reset and startup process the PSoC 3 and PSoC 5LP I/O pins are in three distinct drive modes, as Table 2 shows. PSoC 4 L-series I/O pins are similar.

Table 2. PSoC 3 and PSoC 5LP I/O Pin Drive Modes During Device Reset

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

For details on NVL usage in PSoC 3 and PSoC 5LP, see a device datasheet. In your PSoC Creator project, the NVL settings are established in two places:

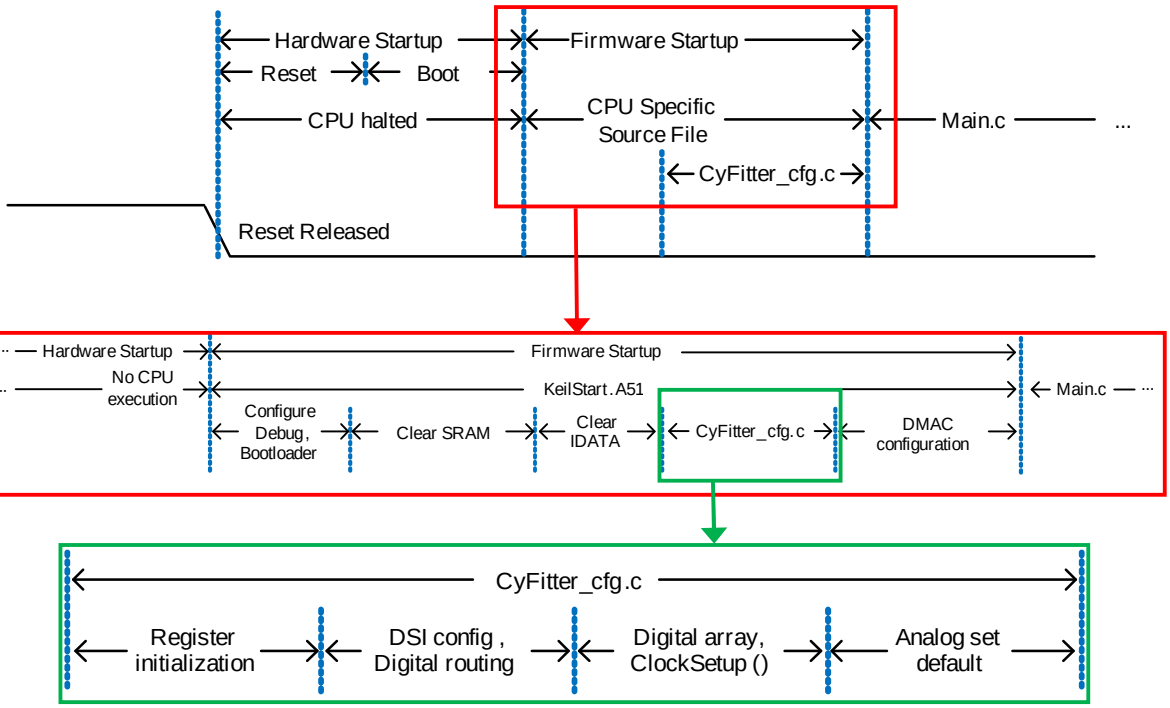
- The **Reset** tab for I/O ports, the individual Pin Component configurations
- The **System** tab for all other NVLs, the design-wide resources (DWR) window

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 26 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 L-series and PSoC 5LP. For more information, see AN60616, PSoC 3 and PSoC 5LP Startup Procedure.

Figure 26. Device Startup Process Diagrams



B.3 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, and CTBm blocks in PSoC 4 L-series.

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

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

All hardware-based functions are restored after configuration (see Figure 26). All firmware functions are restored when the project's main() function starts executing.

B.4 Example: Fan Control

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

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

The fan can also be kept running while the device is reset, during the transfer between the bootloader to the bootloadable, as Table 3 shows.

Table 3. PSoC I/O Pin Drive Modes During Device Reset for Fan Controller

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

C Appendix C – Host Core APIs

C.1 cybtldr_api2.c / .h

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

C.2 cybtldr_api.c / .h

This is a row-level API file for sending a single row of data at a time to the bootloader target. It has functions for setting up the bootload operation, erasing a row, programming a row, verifying a row and ending the bootload operation. Table 4 describes in detail the functions of this API.

Table 4. Functions of *cybtldr_api.c /.h*

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

C.3 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.

C.4 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.

C.5 cybtldr_utils.h

This header provides bootloader version information, and constants for status and error.

Document History

Document Title: AN73503 - PSoC® USB HID Bootloader

Document Number: 001-73503

Revision	ECN	Orig. of Change	Submission Date	Description of Change
**	3435382	UDAY	12/20/2011	New application note.
*A	3673053	UDAY	07/11/2012	Minor updates and edits. Updated template.
*B	3811883	ANCY	11/14/2012	Updated for PSoC 5LP
*C	3850054	KLMZ	12/21/2012	Rewritten for improved content and organization
*D	4034483	KLMZ	06/19/2013	Improved figure clarity, incorporated customer feedback to improve instructional steps, added CY USB related resource
*E	4435010	MKEA	07/17/2014	Added Appendix B – Bootloader and Device Reset
*F	4479382	KLMZ	08/20/2014	Added Cypress SuiteUSB to Related Resources
*G	4831774	KLMZ	07/10/2015	Removed bootloader host example projects, fixed links for new website, removed obsolete links, fixed minor text issues.
*H	5032648	SJLE	12/08/2015	Updated figures for current latest PSoC Creator 3.3 Updated the names of “Related Notes” Deleted obsolete application notes Added sub-headings for Appendix A Updated template Removed Visual studio Express 2010 hyper-link Added the Appendix C for a custom Host bootloader
*I	5101142	MKEA	01/29/2016	Deleted attached project; transferred it to code example CE95391. Added references to the code example. Updated for PSoC 4 L-series.
*J	5582269	SJLE	01/13/2017	Changed the folder path for USBFS component Added the information of pre-built dll file in section 5.2.1 Changed the folder path for cybootloaderutils in section 5.1.5 and 5.2.1 Added the information of the host program example in section 5.2.2 Updated the creating steps for Bootloader_Utils.dll Visual Studio Community project update – Fixed a communication failed after Windows system resumes – Provided both release and debug builds Updated template
*K	5705702	BENV	04/21/2017	Updated logo and copyright

Worldwide Sales and Design Support

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

Products

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

PSoC® Solutions

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

Cypress Developer Community

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

Technical Support

cypress.com/support

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