PSoC® 3 and PSoC 5LP USB General Data Transfer with Standard HID Drivers

Author: Robert Murphy

Associated Project: Yes

Associated Part Family: All PSoC 3 and PSoC 5

Software Version: PSoC Creator™ 3.3 and higher

Related Application Notes: For a complete list of the application notes, click here.

AN82072 discusses how to use PSoC® 3 and PSoC 5LP devices to transfer generic data across USB using native OS drivers included with Windows, Mac OS, and Linux. These drivers are part of the Human Interface Device (HID) class, which is commonly used to support devices such as mice and keyboards, but can also be used for generic data transfers. A PSoC project and a program for each operating system (with source code) demonstrating generic data transfers are included with this application note.

Contents

1 Introduction

When developing a USB device to communicate with a PC for general data transfer, consider three primary device classes: Human Interface, Communication, and Vendor Specific. Many developers adopt the Communication Device class because it creates an easy-to-use COM port interface (which many PSoC users will better recognize as a USBUART). Others elect to use the Vendor Specific Device class due to its extreme configurability with limitations based solely on the restrictions of the USB specification and a company's development resources.

The largest downside to Communication and Vendor Specific device types is that frequently, when you attach a peripheral to a computer, regardless of the operating system, the computer prompts for a device driver in the form of an INF and/or a SYS file. To support multiple operating systems, the developer must maintain multiple driver files. This requirement not only adds effort and frustration to the end-user but also development time and cost to the company that developed the device.

However, the requirement changes if your USB data transfer needs are simple, specifically in an instance where you do not need to send a lot of data very quickly. In that case, you can use the USB Human Interface Device (HID) class for simple data transfer. This is the same device class used for keyboards and mice. The advantage to a HID method is that you can create a simple and reliable way to communicate with a PC, without the need to provide a driver file. This implementation is multi-platform without adding additional USB descriptor code to the device firmware.

Figure 1. HID and Operating System Synergy

Configuring a PSoC to accomplish a HID transfer is relatively simple if you follow a few instructions. It requires that you configure a USBFS component, create a HID descriptor that defines two generic multiple byte containers to send and receive data, and call several USB API functions in PSoC Creator to handle the data. This application note explains the conceptual framework, describes how to configure PSoC 3 and PSoC 5LP devices as a generic HID, and demonstrates how one PSoC device can communicate with a computer running Microsoft® Windows®, Apple® Macintosh®, or Linux™.

Note: Some entry-level knowledge of USB and the HID class is recommended prior to reading this application note. To prepare yourself for the concepts discussed in this document, see the Cypress application notes AN57294 – USB 101: An Introduction to Universal Serial Bus 2.0 and AN57473 – PSoC® 3 / PSoC 5LP USB HID Fundamentals.

2 The HID Class

The HID class is probably the most recognized of all the standardized USB classes available. It includes commonly known devices such as mice, keyboards, and joysticks. In fact, AN57473 and AN58726 - PSoC® 3 / PSoC 5LP USB HID Intermediate (with Keyboard and Composite Device) use these as examples for the HID class on PSoC 3 and PSoC 5LP devices.

All HID data is transferred across the bus in a report. Reports are essentially containers of information in a structured format sent and received by the host and device. Typically, in well-known HIDs such as a keyboard or mouse, the host knows exactly what to do with the data it receives and what kind of data it should send. The host knows this because the HID Report Descriptor has Usage labels that define specific data in the report as information about button and switch state, LED status, or cursor position.

The greatest benefit of a HID is that for many years computer operating systems have included a standard class HID driver to support various HID peripherals directly out of the box. This makes HID a great choice for any USB device that contains a low data rate and can handle a limited latency. There are few additional considerations when evaluating HID:

- All data transactions must use either control or interrupt transfers.
- The maximum transfer size is 64 bytes.
- The maximum transfer rate is one for every 1 ms frame, which limits the throughput to 64 KB/s.
- It supports only one Input Endpoint and one Output Endpoint.
- The host periodically polls the HID for data.

If all these constraints are acceptable, then HID may prove a viable option for your next USB design.

3 What is a Generic HID?

Support for HID has been in computer operating systems for many years. Specifically, Windows 98 was the first operating system to include support for the USB class. Because of this, many manufacturers with devices requiring USB connectivity are considering HID for their generic data transfer needs. The term "Human Interface Device" does not imply that you need to have some type of human interface to use this class. In fact, many devices, such as UPSs and programmable remotes, use the HID class for cross platform support. A HID configured to transfer generic data is known as a Generic HID. These devices use the integrated HID drivers that operating systems already contain to send any data that fits into a packet up to 64 bytes.

Custom HIDs are very similar to the traditional vendor-specific devices discussed in AN56377 - PSoC 3 and PSoC 5LP USB Transfer Types, which require a custom driver. Configuration of a generic HID is identical to the configuration of a more traditional HID such as a keyboard. The difference is in the HID Report Descriptor. The HID Report Descriptor has Usage labels, which define the data sent in the reports. However, rather than defining the data as buttons, switches, or LEDs, the Usage labels define the data as Vendor Specific. This allows us to create a generic container of data that does not contain a specific function or purpose. Undefined bytes of data are sent in the HID Reports between the device and host. It is then the responsibility of the host application and device firmware to know what to do with the data.

With a Generic HID, you can develop a single PSoC USB device that works on all computing platforms without the need to issue firmware updates later in the project roadmap or to plan for the support in advance. It requires only the development of a host software application interface for each supported operating system. Another way to significantly reduce development time is to choose a common programming language and USB library to interface with the USB device. This reduces development time primarily by creating a common ground that can easily be ported to the various operating systems.

Figure 2. Examples of HID and Generic HIDs

Uninterruptable Power Supply
Tablets
Joysticks/Game Controllers
Fingerprint Scanners
Programmable Universal Remote
Barcode Scanners

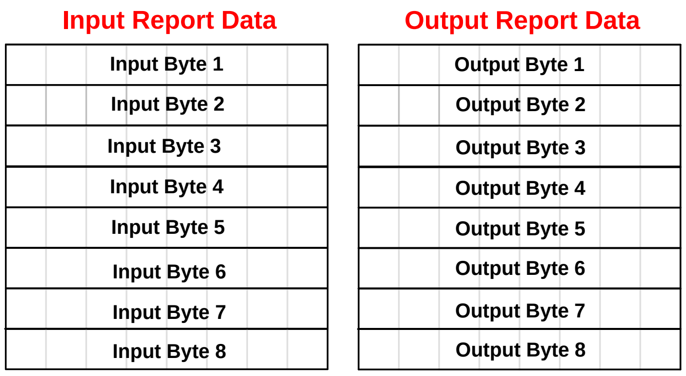
4 Generic HID Report Descriptor

A report descriptor describes each piece of data that the device generates and receives as well as what that data represents. For example, the report descriptor may define items that relate to button status or position information such as the cursor movement of a mouse. For a Generic HID, the report descriptor creates generic data structures that are used as containers to transmit data between the device and the host. The content of the data inside the container is irrelevant. As a result, it is the responsibility of the device and the host application to know what the data is, where to store it in the report, and determine proper endianness.

Note: With a HID, the configuration of the report descriptor is independent of the host computer operating system. One Report Descriptor functions across multiple operating systems.

For the example project in this application note, we configure the Report Descriptor to address several requirements. First, there are two reports: one to send data to the host (Input Report) and another to receive data from the host (Output Report). Additionally, we send and receive only 8 bytes of data, although it is possible to expand this to 64 bytes. Figure 3 is a conceptualization of the container based on those requirements.

Figure 3. Input and Output Data Container



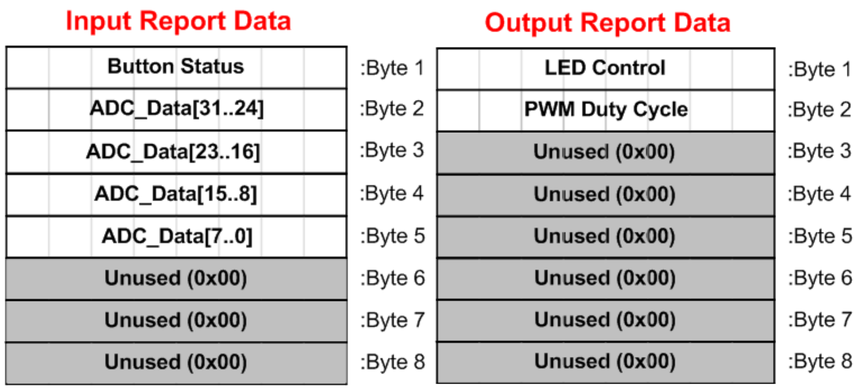
For the project, we also define the data sent to and received from the host. Table 1 lists the function and data requirements of this project. MM

Table 1. Project Functions and Data Requirements

Report	Function	Bytes
Input	Push button status (asserted vs. de-asserted)	1
Input	ADC measurement result	4
Output	LED control signal (on vs. off)	1
Output	PWM duty cycle	1

Knowing the data size and direction, we can now choose our organization scheme for the containers. Because the definition is for a generic data container, it does not matter where in the container we place the data. Figure 4 shows the structure of the data. In Figure 4, the bytes of the report used in the example application are shaded in white while the unused bytes are shaded in gray. The unused bytes represent data used by neither the PSoC nor the host application. All the unused areas of the report are available to transfer additional data if you wish to expand the example application.

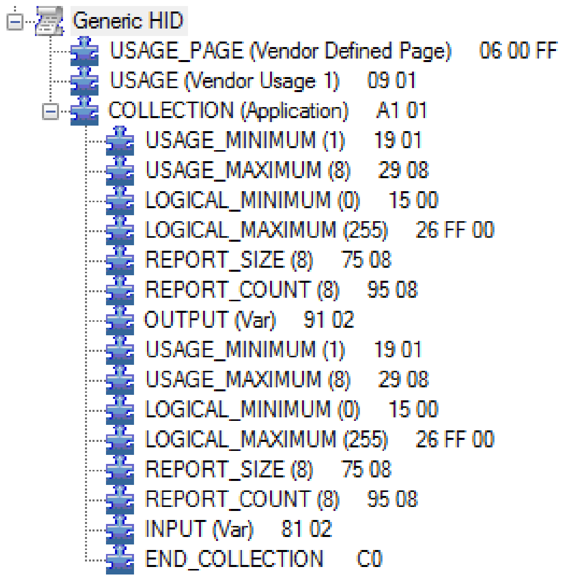
Figure 4. Data Format for Input and Output



Input Report Data		Output Report Data	
Button Status	:Byte 1	LED Control	:Byte 1
ADC_Data[31..24]	:Byte 2	PWM Duty Cycle	:Byte 2
ADC_Data[23..16]	:Byte 3	Unused (0x00)	:Byte 3
ADC_Data[15..8]	:Byte 4	Unused (0x00)	:Byte 4
ADC_Data[7..0]	:Byte 5	Unused (0x00)	:Byte 5
Unused (0x00)	:Byte 6	Unused (0x00)	:Byte 6
Unused (0x00)	:Byte 7	Unused (0x00)	:Byte 7
Unused (0x00)	:Byte 8	Unused (0x00)	:Byte 8

The next step is to define the HID Report Descriptor used to implement these reports. While the structure used to format the data is important, note that we do not care about the organization structure when defining the HID Report Descriptor. Figure 5 shows a HID Report Descriptor that creates 8-byte Input and Output Reports.

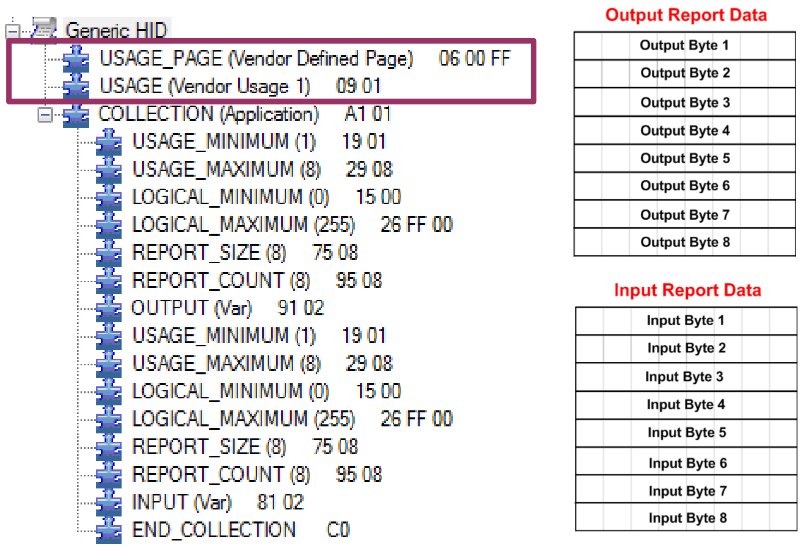
Figure 5. Report Descriptor for Generic HID



```
Generic HID
    USAGE_PAGE (Vendor Defined Page)    06 00 FF
    USAGE (Vendor Usage 1)    09 01
    COLLECTION (Application)    A1 01
        USAGE_MINIMUM (1)    19 01
        USAGE_MAXIMUM (8)    29 08
        LOGICAL_MINIMUM (0)    15 00
        LOGICAL_MAXIMUM (255)    26 FF 00
        REPORT_SIZE (8)    75 08
        REPORT_COUNT (8)    95 08
        OUTPUT (Var)    91 02
        USAGE_MINIMUM (1)    19 01
        USAGE_MAXIMUM (8)    29 08
        LOGICAL_MINIMUM (0)    15 00
        LOGICAL_MAXIMUM (255)    26 FF 00
        REPORT_SIZE (8)    75 08
        REPORT_COUNT (8)    95 08
        INPUT (Var)    81 02
        END_COLLECTION    C0
```

Next, we step through the HID Report Descriptor to better understand it. From an initial glance, it appears that most of the information is just for bi-directional communication. Because our device does not conform to a specific, predefined HID class, we must create a descriptor for a generic, vendor-defined HID. In previous HID examples such as those in AN57473 and AN58726, we relied on a Usage Page of Generic Desktop Control whose function is for applications such as mice, keyboards, joysticks, controllers, and so on. However, for a device whose only purpose is to move data via the HID class, we use the **Vendor Defined Page**. We assign a **Usage** value of ‘01’ to indicate a Vendor Usage label (Figure 6).

Figure 6. Vendor Defined Usage Page



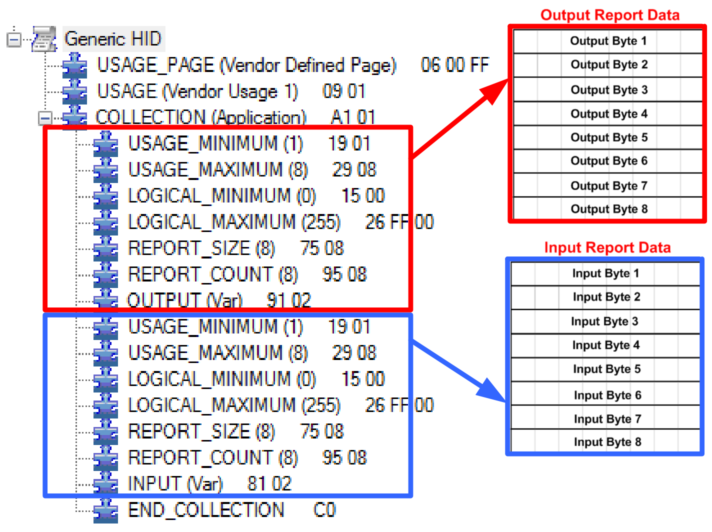
```
Generic HID
    USAGE_PAGE (Vendor Defined Page)    06 00 FF
    USAGE (Vendor Usage 1)    09 01
    COLLECTION (Application)    A1 01
        USAGE_MINIMUM (1)    19 01
        USAGE_MAXIMUM (8)    29 08
        LOGICAL_MINIMUM (0)    15 00
        LOGICAL_MAXIMUM (255)    26 FF 00
        REPORT_SIZE (8)    75 08
        REPORT_COUNT (8)    95 08
        OUTPUT (Var)    91 02
        USAGE_MINIMUM (1)    19 01
        USAGE_MAXIMUM (8)    29 08
        LOGICAL_MINIMUM (0)    15 00
        LOGICAL_MAXIMUM (255)    26 FF 00
        REPORT_SIZE (8)    75 08
        REPORT_COUNT (8)    95 08
        INPUT (Var)    81 02
        END_COLLECTION    C0
```

Output Report Data
Output Byte 1
Output Byte 2
Output Byte 3
Output Byte 4
Output Byte 5
Output Byte 6
Output Byte 7
Output Byte 8

Input Report Data
Input Byte 1
Input Byte 2
Input Byte 3
Input Byte 4
Input Byte 5
Input Byte 6
Input Byte 7
Input Byte 8

Because this application sends data to a PC host and receives data from the PC, there are two sections of the HID report descriptor: a portion that configures the Output Report and another portion that configures the Input Report. Figure 7 outlines the sections responsible for the Output and Input.

Figure 7. Output and Input Sections of Report Descriptor



```
Generic HID
    USAGE_PAGE (Vendor Defined Page)    06 00 FF
    USAGE (Vendor Usage 1)    09 01
    COLLECTION (Application)    A1 01
        USAGE_MINIMUM (1)    19 01
        USAGE_MAXIMUM (8)    29 08
        LOGICAL_MINIMUM (0)    15 00
        LOGICAL_MAXIMUM (255)    26 FF 00
        REPORT_SIZE (8)    75 08
        REPORT_COUNT (8)    95 08
        OUTPUT (Var)    91 02
        USAGE_MINIMUM (1)    19 01
        USAGE_MAXIMUM (8)    29 08
        LOGICAL_MINIMUM (0)    15 00
        LOGICAL_MAXIMUM (255)    26 FF 00
        REPORT_SIZE (8)    75 08
        REPORT_COUNT (8)    95 08
        INPUT (Var)    81 02
        END_COLLECTION    C0
```

Output Report Data
Output Byte 1
Output Byte 2
Output Byte 3
Output Byte 4
Output Byte 5
Output Byte 6
Output Byte 7
Output Byte 8

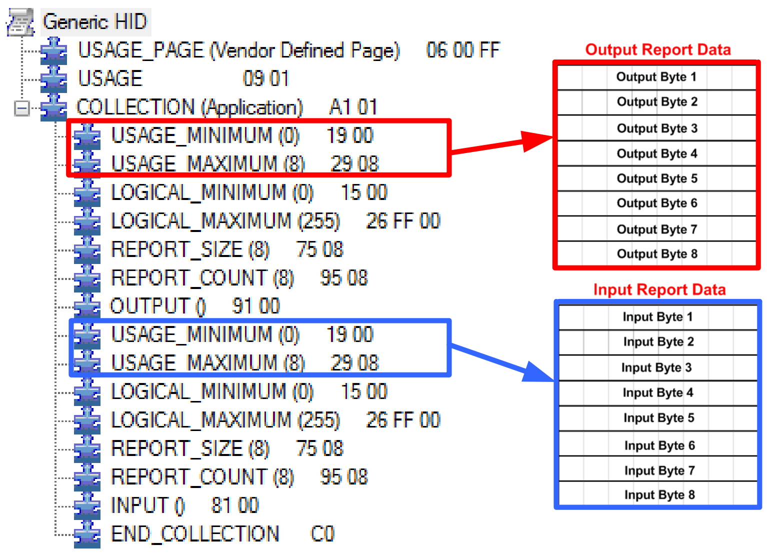
Input Report Data
Input Byte 1
Input Byte 2
Input Byte 3
Input Byte 4
Input Byte 5
Input Byte 6
Input Byte 7
Input Byte 8

We start with configuration of the Usage Minimum and Maximum of the report descriptor. AN57473 describes these parameters as follows:

“Usage Minimum and Maximum: Used to link Usage IDs to the data in an array or bitmap. The Usage Minimum defines the starting point and the Usage Maximum defines the ending point.”

Therefore, we use the Usage Minimum and Usage Maximum to give each element (each byte) in the Input Report and Output Report a vendor-defined usage (as opposed to a usage that defines the element as a button). To assign all 8 bytes in each report, we set the Usage Minimum value to 1 and the Usage Maximum value to 8. Figure 8 shows this configuration of the HID report descriptor. The red outline (upper box) indicates the configuration for the Output Report, and the blue outline (lower box) indicates the configuration for the Input Report.

Figure 8. Usage Maximum/Minimum Configuration



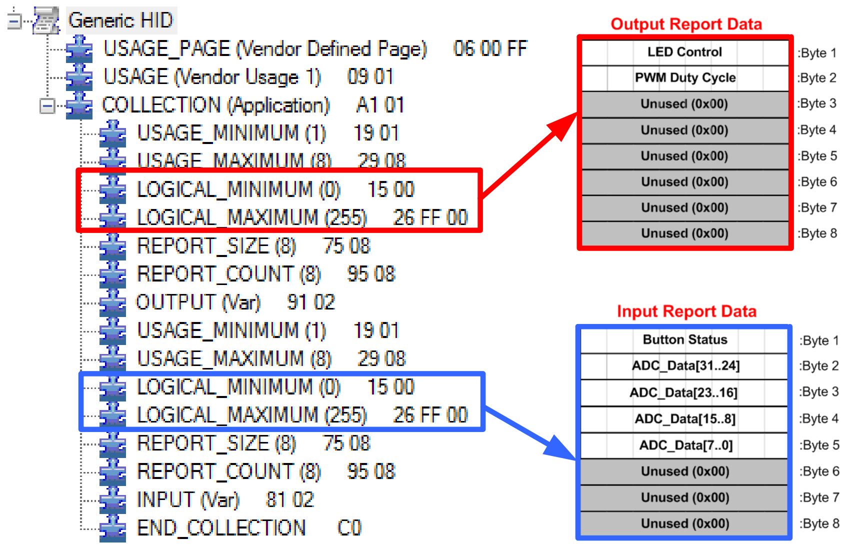
```
Generic HID
  USAGE_PAGE (Vendor Defined Page)    06 00 FF
  USAGE                09 01
  COLLECTION (Application)    A1 01
    USAGE_MINIMUM (0)    19 00
    USAGE_MAXIMUM (8)    29 08
    LOGICAL_MINIMUM (0)    15 00
    LOGICAL_MAXIMUM (255)    26 FF 00
    REPORT_SIZE (8)    75 08
    REPORT_COUNT (8)    95 08
    OUTPUT ()    91 00
    USAGE_MINIMUM (0)    19 00
    USAGE_MAXIMUM (8)    29 08
    LOGICAL_MINIMUM (0)    15 00
    LOGICAL_MAXIMUM (255)    26 FF 00
    REPORT_SIZE (8)    75 08
    REPORT_COUNT (8)    95 08
    INPUT ()    81 00
    END_COLLECTION    C0
```

Next, we configure the Logical Minimum and Maximum for the generic HID. AN57473 describes these parameters as follows:

“Logical Maximum and Minimum: Determines the limits for reported values in an array or variable. For example, a mouse that reports position values from 127 to -127 has a logical maximum of 127 and a logical minimum of -127."

Because each element is one byte in length, we define a Logical Minimum of ‘0x00’ and a Logical Maximum of ‘0xFF’ to allow a full-scale range of 8 bits. Figure 9 shows this configuration of the HID report descriptor. The red outline indicates the configuration for the Output Report, and the blue outline indicates the configuration for the Input Report.

Figure 9. Logical Maximum/Minimum for Output Report



```
Generic HID
    USAGE_PAGE (Vendor Defined Page)    06 00 FF
    USAGE (Vendor Usage 1)    09 01
    COLLECTION (Application)    A1 01
        USAGE_MINIMUM (1)    19 01
        USAGE_MAXIMUM (8)    29 08
        LOGICAL_MINIMUM (0)    15 00
        LOGICAL_MAXIMUM (255)    26 FF 00
        REPORT_SIZE (8)    75 08
        REPORT_COUNT (8)    95 08
        OUTPUT (Var)    91 02
        USAGE_MINIMUM (1)    19 01
        USAGE_MAXIMUM (8)    29 08
        LOGICAL_MINIMUM (0)    15 00
        LOGICAL_MAXIMUM (255)    26 FF 00
        REPORT_SIZE (8)    75 08
        REPORT_COUNT (8)    95 08
        INPUT (Var)    81 02
        END_COLLECTION    C0
```

Output Report Data

LED Control	:Byte 1
PWM Duty Cycle	:Byte 2
Unused (0x00)	:Byte 3
Unused (0x00)	:Byte 4
Unused (0x00)	:Byte 5
Unused (0x00)	:Byte 6
Unused (0x00)	:Byte 7
Unused (0x00)	:Byte 8

Input Report Data

Button Status	:Byte 1
ADC_Data[31..24]	:Byte 2
ADC_Data[23..16]	:Byte 3
ADC_Data[15..8]	:Byte 4
ADC_Data[7..0]	:Byte 5
Unused (0x00)	:Byte 6
Unused (0x00)	:Byte 7
Unused (0x00)	:Byte 8

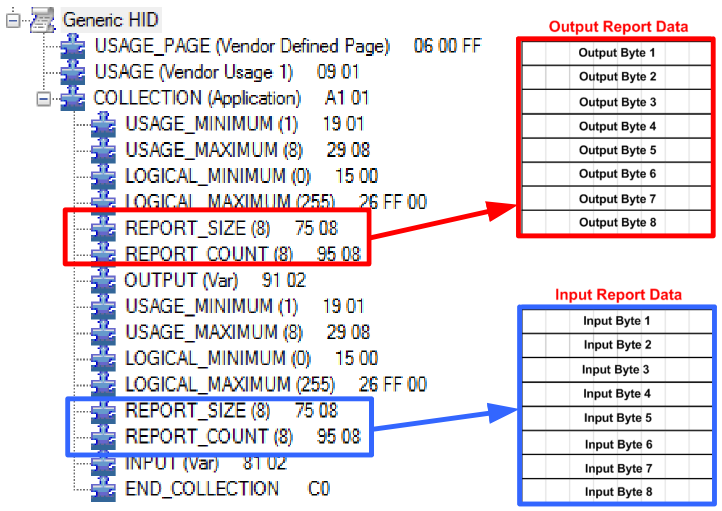
Now, we configure the report descriptor Report Size and Report Count. These two parameters determine the size of the Input and Output Reports. AN57473 describes these parameters as:

“Report Size: Determines the size in bits of a field in an Input, Output, or Feature item.

“Report Count: Specifies number of data fields for an Input, Output, or Feature item.”

Figure 10 on page 6 shows the Report Size and Report Count for both the Input and the Output Reports. Based on the definitions, the Report Size defines that each report field is 8 bits in length. To elaborate, if each report was an array, the Report Size parameter would define each array element as 8 bits in size. Additionally, the Report Count indicates that there are eight reports (also known as data fields). The result is an Input Report comprised of eight, 8-bit values. The Output Report is the same.

Figure 10. Report Size and Report Count Configuration



```
Generic HID
    USAGE_PAGE (Vendor Defined Page)    06 00 FF
    USAGE (Vendor Usage 1)    09 01
    COLLECTION (Application)    A1 01
        USAGE_MINIMUM (1)    19 01
        USAGE_MAXIMUM (8)    29 08
        LOGICAL_MINIMUM (0)    15 00
        LOGICAL_MAXIMUM (255)    26 FF 00
        REPORT_SIZE (8)    75 08
        REPORT_COUNT (8)    95 08
        OUTPUT (Var)    91 02
        USAGE_MINIMUM (1)    19 01
        USAGE_MAXIMUM (8)    29 08
        LOGICAL_MINIMUM (0)    15 00
        LOGICAL_MAXIMUM (255)    26 FF 00
        REPORT_SIZE (8)    75 08
        REPORT_COUNT (8)    95 08
        INPUT (Var)    81 02
        END_COLLECTION    C0
```

Output Report Data

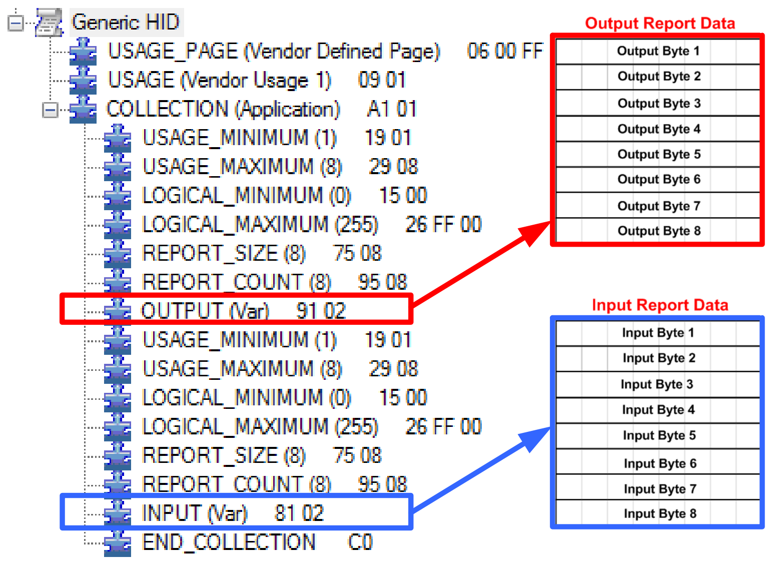
Output Byte 1
Output Byte 2
Output Byte 3
Output Byte 4
Output Byte 5
Output Byte 6
Output Byte 7
Output Byte 8

Input Report Data

Input Byte 1
Input Byte 2
Input Byte 3
Input Byte 4
Input Byte 5
Input Byte 6
Input Byte 7
Input Byte 8

The final step for the HID Report Descriptor is to configure the Input and Output Items. Until now, all the parameters for the Input and Output Reports are the same. It is the Item parameter that distinguishes between input and output as shown in Figure 11.

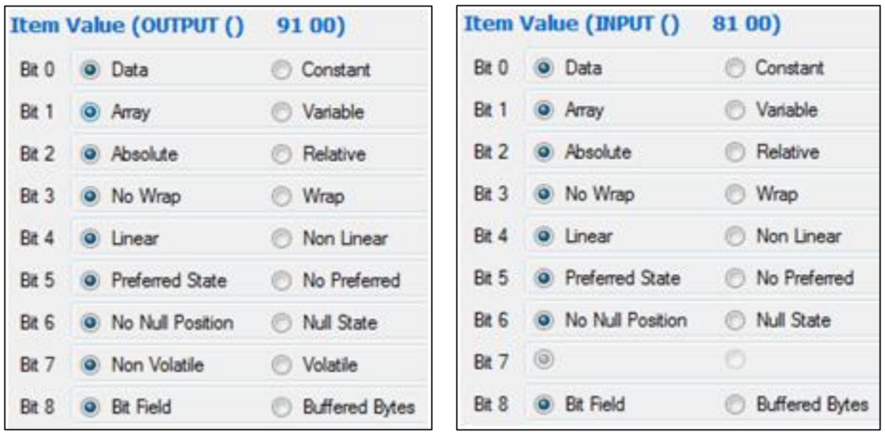
Figure 11. Input and Output Item Configuration



```
Generic HID
  USAGE_PAGE (Vendor Defined Page)   06 00 FF
  USAGE (Vendor Usage 1)   09 01
  COLLECTION (Application)   A1 01
    USAGE_MINIMUM (1)   19 01
    USAGE_MAXIMUM (8)   29 08
    LOGICAL_MINIMUM (0)   15 00
    LOGICAL_MAXIMUM (255)   26 FF 00
    REPORT_SIZE (8)   75 08
    REPORT_COUNT (8)   95 08
    OUTPUT (Var)   91 02
    USAGE_MINIMUM (1)   19 01
    USAGE_MAXIMUM (8)   29 08
    LOGICAL_MINIMUM (0)   15 00
    LOGICAL_MAXIMUM (255)   26 FF 00
    REPORT_SIZE (8)   75 08
    REPORT_COUNT (8)   95 08
    INPUT (Var)   81 02
    END_COLLECTION   C0
```

Normally with non-generic HIDs, we make specific configuration settings to the Input and Output Items. Because this application only creates containers for data, we can ignore these adjustments. Figure 12 demonstrates the lack of adjustments to the Input and Output Items. It shows the configuration window as it appears in PSoC Creator. The bit fields are all set to their default value of 0'b. Any change to these parameters does not affect the overall functionality of the project.

Figure 12. OUTPUT/INPUT Item Bit Configuration



Item Value (OUTPUT () 91 00)

Bit 0	[✓ Data]	Constant
Bit 1	[✓ Array]	Variable
Bit 2	[✓ Absolute]	Relative
Bit 3	[✓ No Wrap]	Wrap
Bit 4	[✓ Linear]	Non Linear
Bit 5	[✓ Preferred State]	No Preferred
Bit 6	[✓ No Null Position]	Null State
Bit 7	[✓ Non Volatile]	Volatile
Bit 8	[✓ Bit Field]	Buffered Bytes

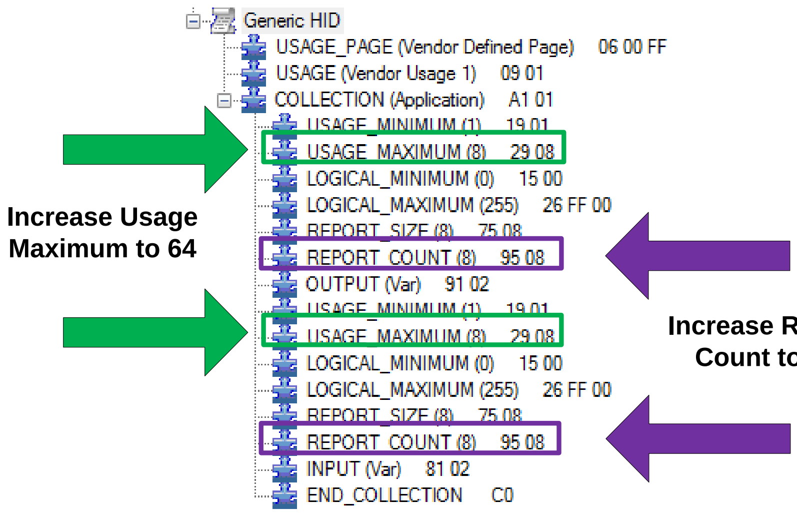
Item Value (INPUT () 81 00)

Bit 0	[✓ Data]	Constant
Bit 1	[✓ Array]	Variable
Bit 2	[✓ Absolute]	Relative
Bit 3	[✓ No Wrap]	Wrap
Bit 4	[✓ Linear]	Non Linear
Bit 5	[✓ Preferred State]	No Preferred
Bit 6	[✓ No Null Position]	Null State
Bit 7	[✓]	
Bit 8	[✓ Bit Field]	Buffered Bytes

5 Supporting 64-Byte HID Reports

It only requires four edits to the HID report descriptor in order to expand from 8-byte reports to 64-byte reports. Change the Usage Maximum and the Report Count items from a value of 8 to a value of 64 for both the Input and Output portion of the descriptor (Figure 13). Additionally, you need to change the Input and Output Endpoint Maximum Packet Size to 64 (from its default configuration of 8). We will look at how to accomplish this later.

Figure 13. How to Edit the Report Descriptor for 64-Bytes



```
Generic HID
  USAGE_PAGE (Vendor Defined Page)   06 00 FF
  USAGE (Vendor Usage 1)   09 01
  COLLECTION (Application)   A1 01
    USAGE_MINIMUM (1)   19 01
    USAGE_MAXIMUM (8)   29 08
    LOGICAL_MINIMUM (0)   15 00
    LOGICAL_MAXIMUM (255)   26 FF 00
    REPORT_SIZE (8)   75 08
    REPORT_COUNT (8)   95 08
    OUTPUT (Var)   91 02
    USAGE_MINIMUM (1)   19 01
    USAGE_MAXIMUM (8)   29 08
    LOGICAL_MINIMUM (0)   15 00
    LOGICAL_MAXIMUM (255)   26 FF 00
    REPORT_SIZE (8)   75 08
    REPORT_COUNT (8)   95 08
    INPUT (Var)   81 02
    END_COLLECTION   C0
```

6 Example Project: Generic Data Transfer via HID

This example project demonstrates both sides of the application. It demonstrates how to use and create a host application to communicate between a computer running the Windows, Mac, or Linux operating system. It also demonstrates how to use and configure the USBFS component in PSoC Creator to use the HID protocol. We transfer information such as an ADC measurement and the status of a push button to the host via an Input transfer. The host uses an Output transfer to send data to the PSoC to control the ON/OFF state of an LED along with the duty cycle of a PWM connected to another LED. We demonstrate the process with the user interfaces for all three operating systems.

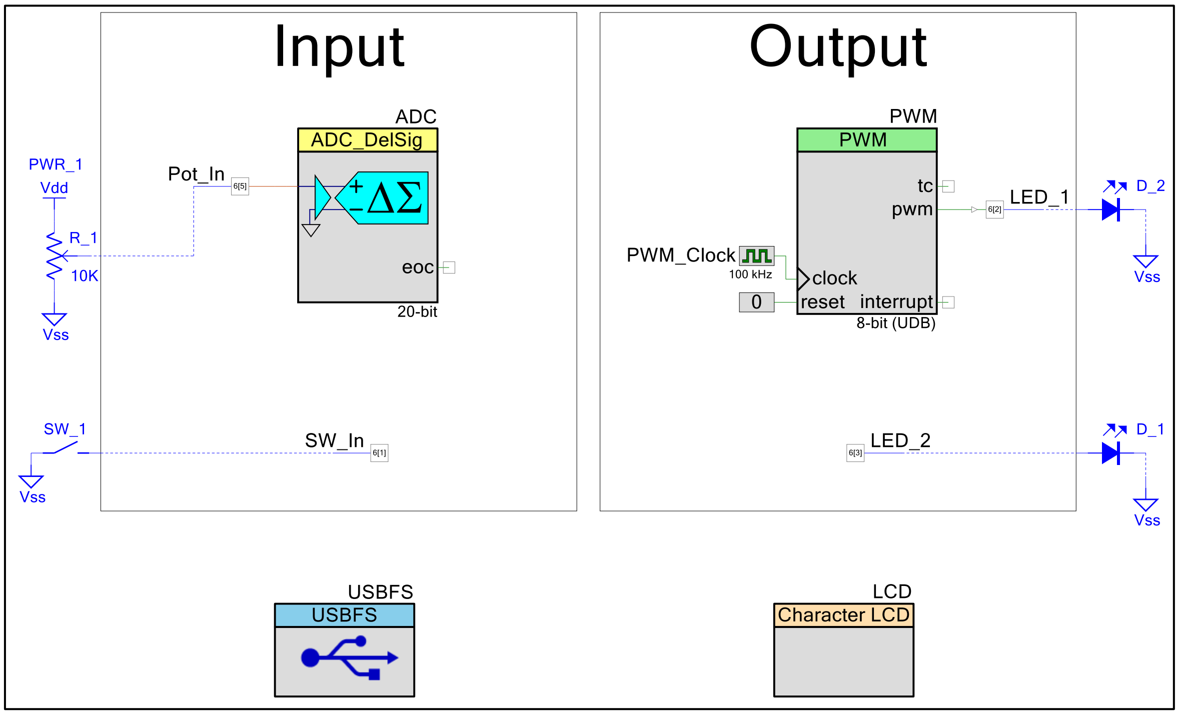
6.1 PSoC Creator Configuration

6.1.1 Create the Project

1. Open PSoC Creator.
2. Create an empty project, selecting either PSoC 3 or PSoC 5LP, and name it *MyFirstGenericDevice*.
3. Place the following components from the PSoC Creator Component Catalog on to the schematic entry page (TopDesign.cysch):
   - USBFS
   - Character LCD
   - Delta Sigma ADC
   - PWM
   - Pin Components
     - Analog Pin (Quantity 1)
     - Digital Input Pin (Quantity 1)
     - Digital Output Pin (Quantity 2)
4. Use the wiring tool to arrange and connect the components as shown in Figure 14.

The gray boxes and annotation components are for clarification and not required for functionality. In addition, we changed default names of the components. Modify the component names to ensure proper building and functionality of the PSoC Creator project.

Figure 14. PSoC Creator Project Schematic Layout



After arranging and placing all the components appropriately, the next step is to configure them so that they function properly in our application. We start by configuring the USBFS component because it is the core component and most important component in this design.

6.1.2 Configure the USBFS Component

1. Right-click the USBFS component and select **Configuration**.

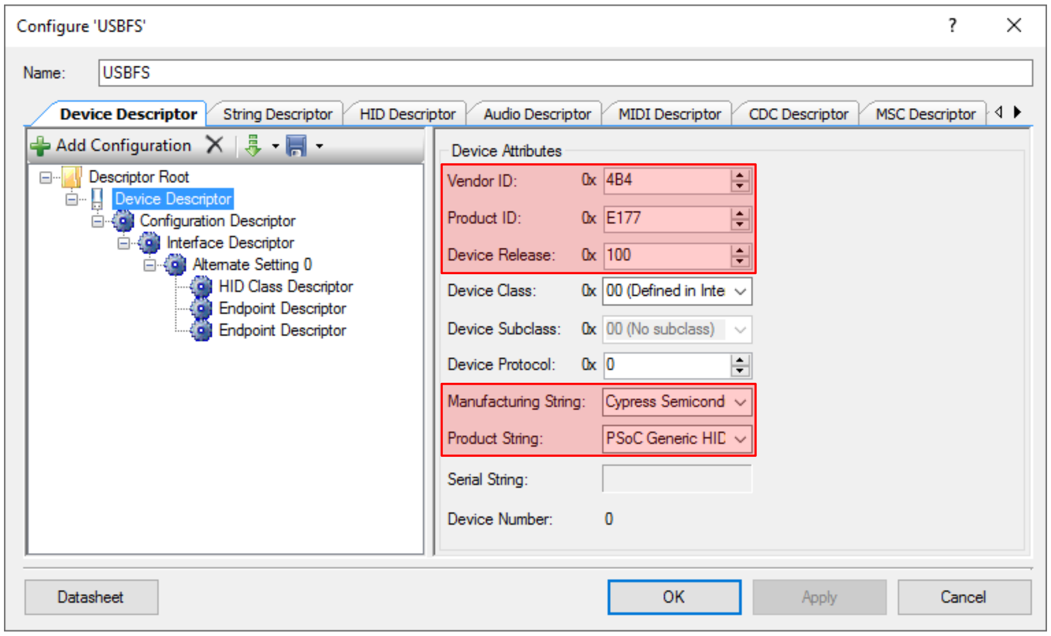
   The Configuration dialog box appears displaying the **Device Descriptor** tab.

2. In the **Descriptor** tree, click **Device Descriptor**. Configure options for the **Device Attributes** as follows and shown in Figure 15:

   - **Vendor ID**: 0x04B4
   - **Product ID**: 0xE177
   - **Device Release**: 0x100
   - **Manufacturing String**: Cypress Semiconductor
   - **Product String**: PSoC Generic HID Data Transfer

   In this 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 it in 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. You can optionally change the various strings such as the Product String and the Manufacturing String.

Figure 15. USBFS Device Descriptor Configuration

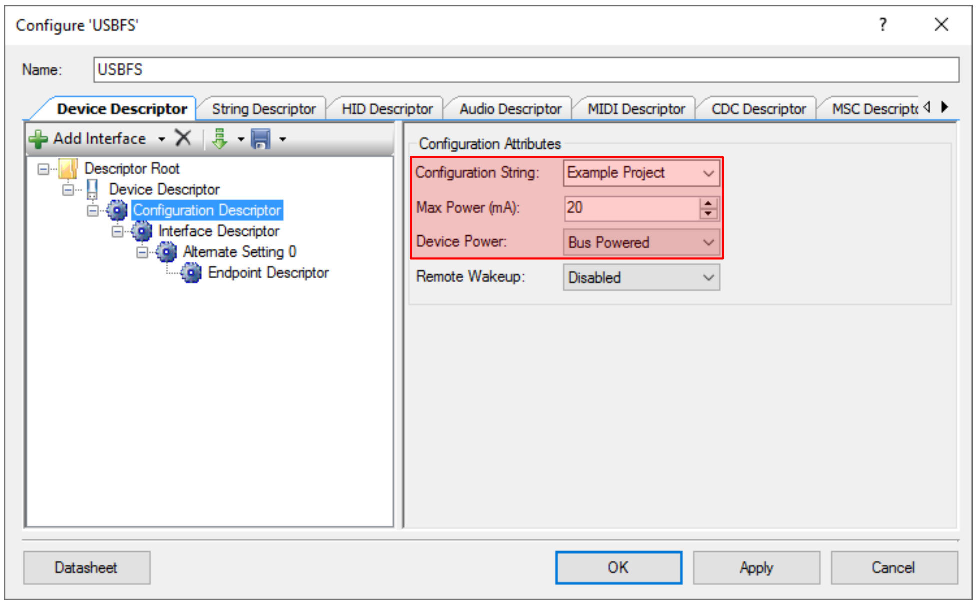


Name: USBFS

Vendor ID:	0x 4B4
Product ID:	0x E177
Device Release:	0x 100
Device Class:	0x 00 (Defined in Inte
Device Subclass:	0x 00 (No subclass)
Device Protocol:	0x 0
Manufacturing String:	Cypress Semicond
Product String:	PSoC Generic HID
Serial String:	
Device Number:	0

3. In the **Descriptor** tree, click **Configuration Descriptor**. Configure options for **Configuration Attributes** as follows and shown in Figure 16:
   - **Configuration String**: Example Project
   - **Max Power (mA)**: 20
   - **Device Power**: Bus Powered

Figure 16. USBFS Configuration Descriptor Configuration



Configure 'USBFS'

Name: USBFS

Configuration Attributes

Configuration String: Example Project

Max Power (mA): 20

Device Power: Bus Powered

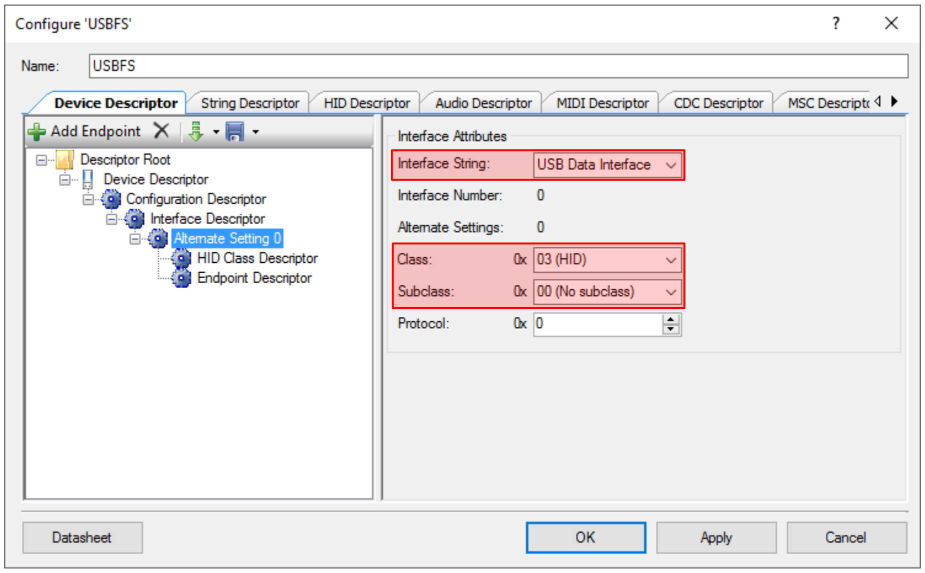
Remote Wakeup: Disabled

The key settings in this section 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 we cannot exceed this requirement. Because we do not require Remote Wakeup functionality, we disable it.

4. In the **Descriptor** tree, click **Alternate Setting 0**. Configure options for **Interface Attributes** as follows and shown in Figure 17:
   - **Interface String**: USB Data Interface
   - **Class**: HID
   - **Subclass**: No subclass

   Each interface can have multiple Alternate Settings to allow for multiple endpoint configurations. In this application, we only have one alternate setting (the default one). If we had more, we would configure all the Alternate Settings via the Interface Descriptor. Here, we 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.

Figure 17. USBFS Interface Descriptor Configuration



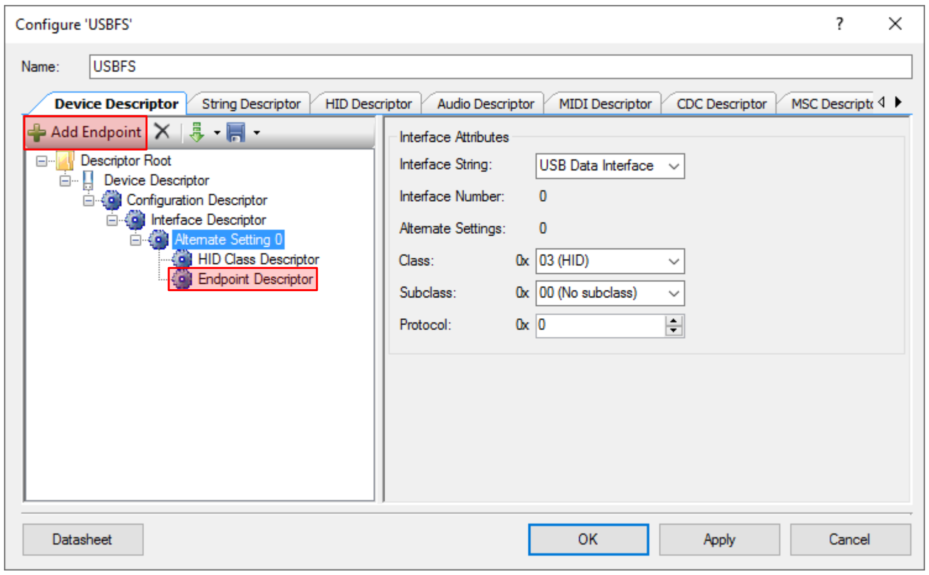
Interface Attributes

Interface String: USB Data Interface
Interface Number: 0
Alternate Settings: 0
Class: 0x 03 (HID)
Subclass: 0x 00 (No subclass)
Protocol: 0x 0

5. Click the **Add Endpoint** button. The location of this button is shown in Figure 18.

   An additional **Endpoint Descriptor** appears in the **Descriptor** tree.

Figure 18. USBFS Adding Additional Endpoint Descriptor


Configure 'USBFS'

Name: USBFS

Interface Attributes

Interface String: USB Data Interface

Interface Number: 0

Alternate Settings: 0

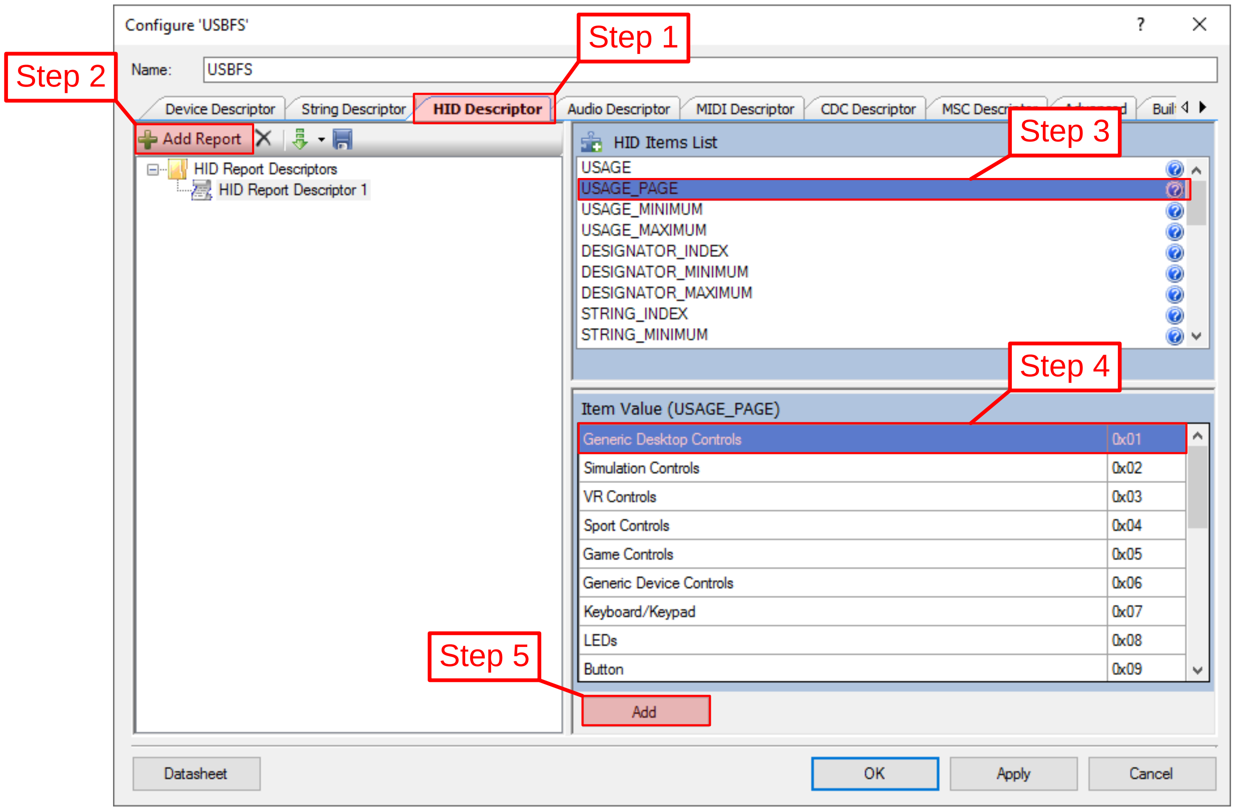
Class: 0x 03 (HID)

Subclass: 0x 00 (No subclass)

Protocol: 0x 0

6. Use the following steps to use the **HID Descriptor** to create a report descriptor. The layout of the HID descriptor tool is shown in Figure 19.

Figure 19. USBFS HID Descriptor Creation


Configure 'USBFS'

Name: USBFS

Item Value (USAGE_PAGE)

Generic Desktop Controls	0x01
Simulation Controls	0x02
VR Controls	0x03
Sport Controls	0x04
Game Controls	0x05
Generic Device Controls	0x06
Keyboard/Keypad	0x07
LEDs	0x08
Button	0x09

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

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

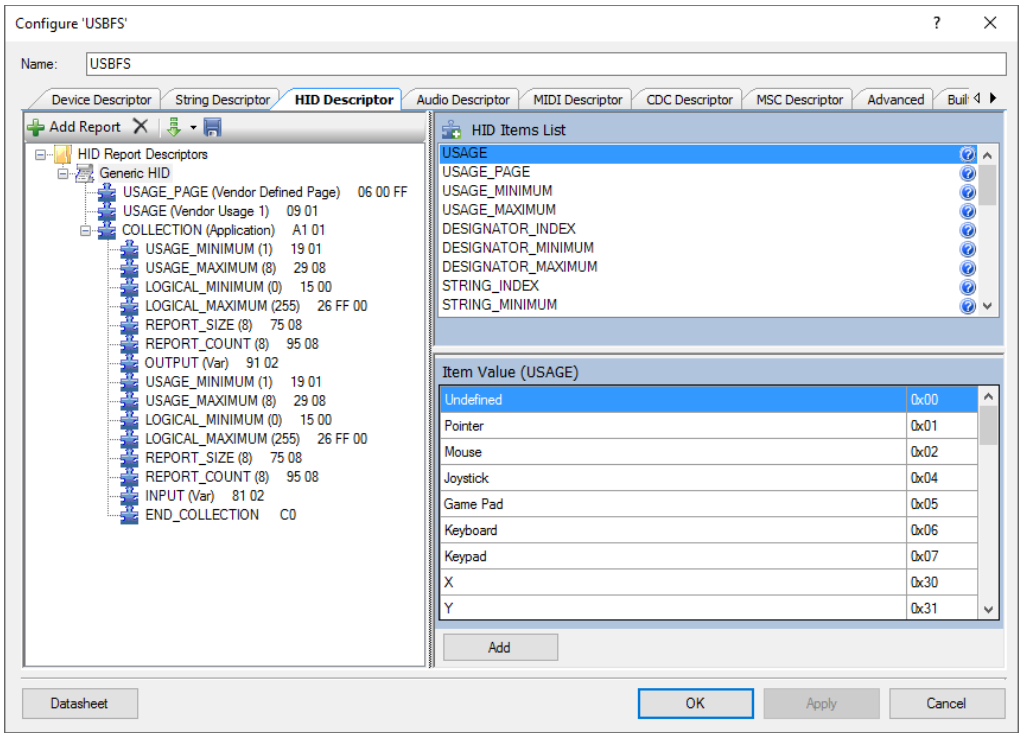
c. Select an item from the **HID Item List**.

d. Select a value from the **Item Value** list.

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 for each item in the report descriptor until it resembles Figure 20.

Figure 20. USBFS Completed HID Descriptor



```
HID Report Descriptors
  Generic HID
    USAGE_PAGE (Vendor Defined Page)   06 00 FF
    USAGE (Vendor Usage 1)   09 01
    COLLECTION (Application)   A1 01
      USAGE_MINIMUM (1)   19 01
      USAGE_MAXIMUM (8)   29 08
      LOGICAL_MINIMUM (0)   15 00
      LOGICAL_MAXIMUM (255)   26 FF 00
      REPORT_SIZE (8)   75 08
      REPORT_COUNT (8)   95 08
      OUTPUT (Var)   91 02
      USAGE_MINIMUM (1)   19 01
      USAGE_MAXIMUM (8)   29 08
      LOGICAL_MINIMUM (0)   15 00
      LOGICAL_MAXIMUM (255)   26 FF 00
      REPORT_SIZE (8)   75 08
      REPORT_COUNT (8)   95 08
      INPUT (Var)   81 02
      END_COLLECTION   C0
```

USAGE
USAGE_PAGE
USAGE_MINIMUM
USAGE_MAXIMUM
DESIGNATOR_INDEX
DESIGNATOR_MINIMUM
DESIGNATOR_MAXIMUM
STRING_INDEX
STRING_MINIMUM

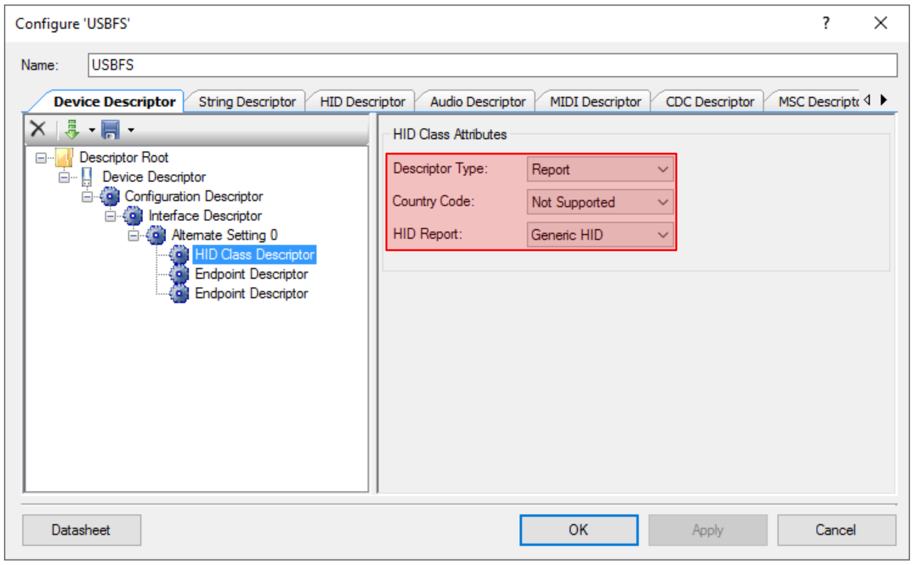
Undefined	0x00
Pointer	0x01
Mouse	0x02
Joystick	0x04
Game Pad	0x05
Keyboard	0x06
Keypad	0x07
X	0x30
Y	0x31

7. Click the **Device Descriptor** tab. In the **Descriptor** tree, click **HID Class Descriptor**. Configure options for **Device Attributes** as follows and shown in Figure 21:

- **Descriptor Type**: Report
- **Country Code**: Not Supported
- **HID Report**: Generic HID

Because the project uses a Report Descriptor, we set the Descriptor Type to Report. Because the project is not specific to any country, we set Country Code to Not Supported. Finally, the HID Class Descriptor must point to the HID Report Descriptor. To create this link, we set the HID Report to the name of our HID Report Descriptor, which in our example is Generic HID.

Figure 21. USBFS HID Class Descriptor



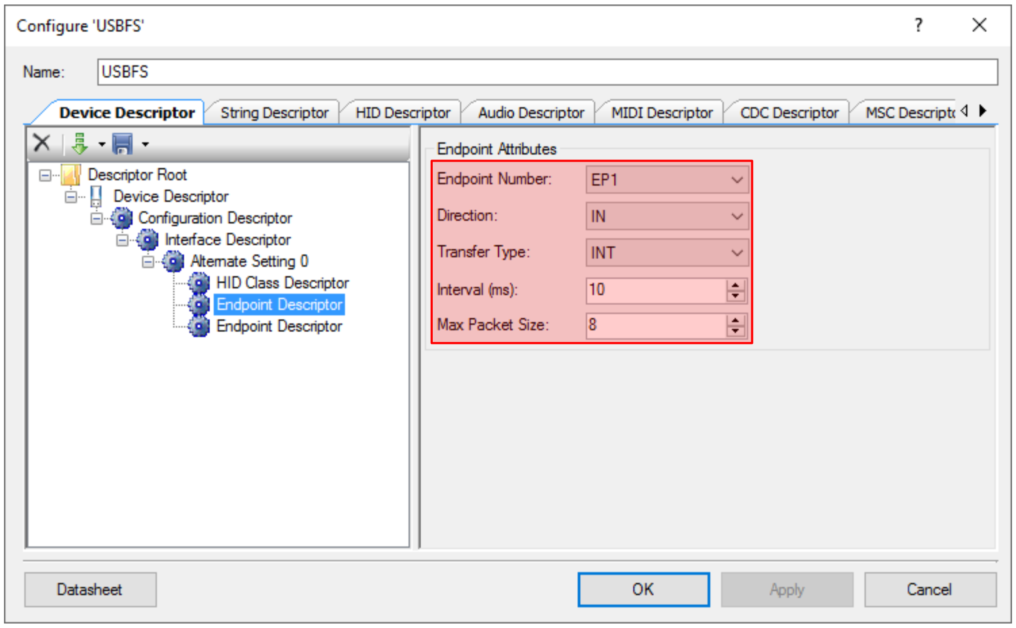
Descriptor Type: Report
Country Code: Not Supported
HID Report: Generic HID

8. In the **Descriptor** tree, click the first **Endpoint Descriptor** entry. Configure options for **Endpoint Attributes** as follows and shown in Figure 22:

- **Endpoint Number**: EP1
- **Direction**: IN
- **Transfer Type**: INT
- **Interval**: 10
- **Max Packet Size**: 8

The first entry is for the IN Endpoint. It acts as a buffer for data sent to the host. First, we define this endpoint as Endpoint 1 (EP1). Next, we set the direction of the endpoint as an Input. Because this application is an HID, the specification requires that we use Interrupt (INT) transfers. We then set the frequency (or Interval) the host polls the device to a value of 10 ms. Finally, we set our Maximum Packet Size to a value of 8 bytes.

Figure 22. USBFS IN Endpoint Descriptor



Endpoint Attributes
Endpoint Number: EP1
Direction: IN
Transfer Type: INT
Interval (ms): 10
Max Packet Size: 8

9. Repeat step 8 for the second endpoint to configure it as an OUT endpoint with the following attributes. The purpose of this endpoint is to act as a buffer for data that is received from the host.
   - **Endpoint Number**: EP2
   - **Direction**: OUT
   - **Transfer Type**: INT
   - **Interval**: 10
   - **Max Packet Size**: 8
10. Click the **Apply** button, and then click the **OK** button to close out the configuration wizard.

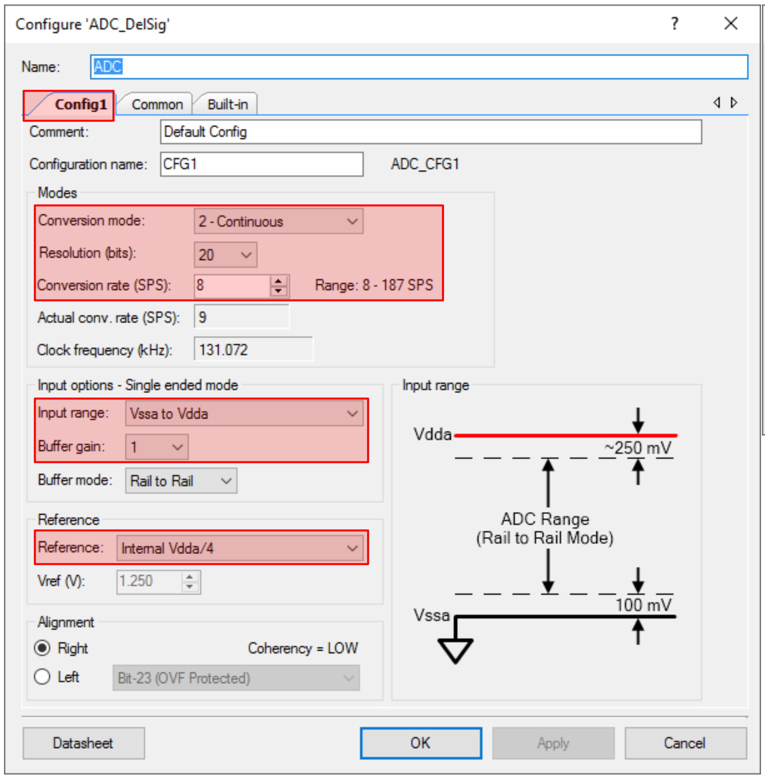
6.1.3 Configure Other Components

With the USBFS component fully configured, there are still configuration settings to enter for the other components in the project to finalize the application. We start with the ADC component.

1. From the PSoC Creator schematic, double-click the **ADC** component.
2. Configure the **ADC** component as shown in the following screenshot (Figure 23). Click **OK** to save and close the configuration for the ADC component.

   What we primarily need is to max out the resolution to 20 bits, adjust the sample rate, set the ADC to function in single-ended mode, and set the input range from V_{SSA} to V_{DDA}.

Figure 23. Delta Sigma ADC Configuration Wizard



Configure 'ADC_DelSig'

Name: ADC

Config1 | Common | Built-in

Comment: Default Config

Configuration name: CFG1 ADC_CFG1

Modes

Conversion mode: 2 - Continuous

Resolution (bits): 20

Conversion rate (SPS): 8 Range: 8 - 187 SPS

Actual conv. rate (SPS): 9

Clock frequency (kHz): 131.072

Input options - Single ended mode

Input range: Vssa to Vdda

Buffer gain: 1

Buffer mode: Rail to Rail

Reference

Reference: Internal Vdda/4

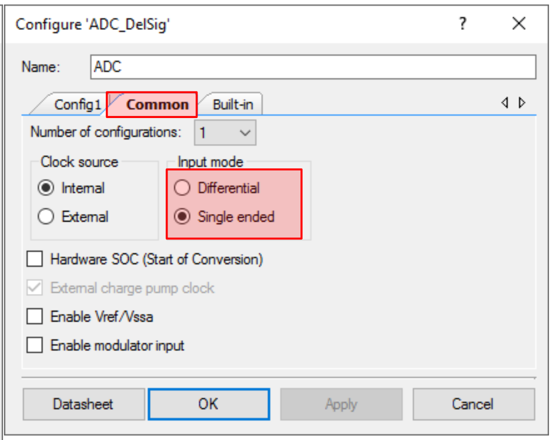
Vref (V): 1.250

Alignment

[✓ Right] Coherency = LOW

Left Bit-23 (OVF Protected)

Input range



Configure 'ADC_DelSig'

Name: ADC

Config1 | Common | Built-in

Number of configurations: 1

Clock source: [✓ Internal] External

Input mode: Differential [✓ Single ended]

☐ Hardware SOC (Start of Conversion)

☑ External charge pump clock

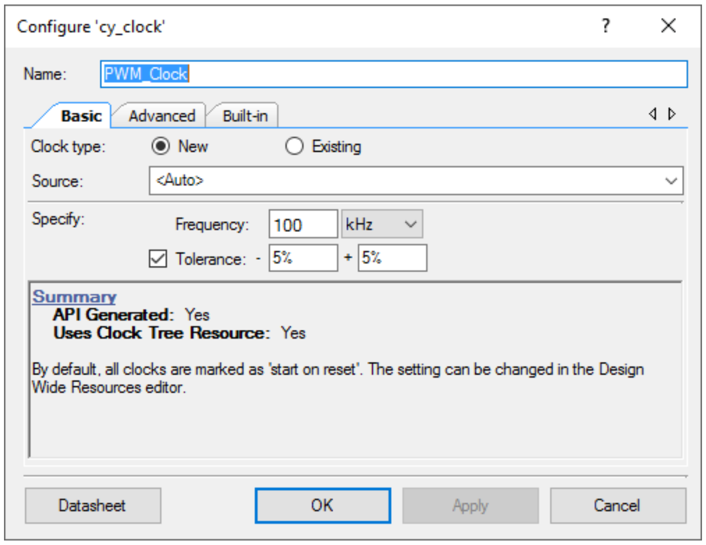
☐ Enable Vref/Vssa

☐ Enable modulator input

3. Double-click the PWM component. Configure the **PWM** component as shown in the following screenshot. After all changes are made, click **OK** to save and close the configuration for the PWM component.

4. Double-click the input clock to the PWM and configure it so that its frequency is set to 100 kHz as shown in Figure 24. Click **OK** to close the configure **PWM_Clock** dialog box.

Figure 24. PWM Clock Configuration



Name: PWM_Clock

Clock type: [✓ New] Existing

Source: <Auto>

Specify: Frequency: 100 kHz

☑ Tolerance: - 5% + 5%

Summary

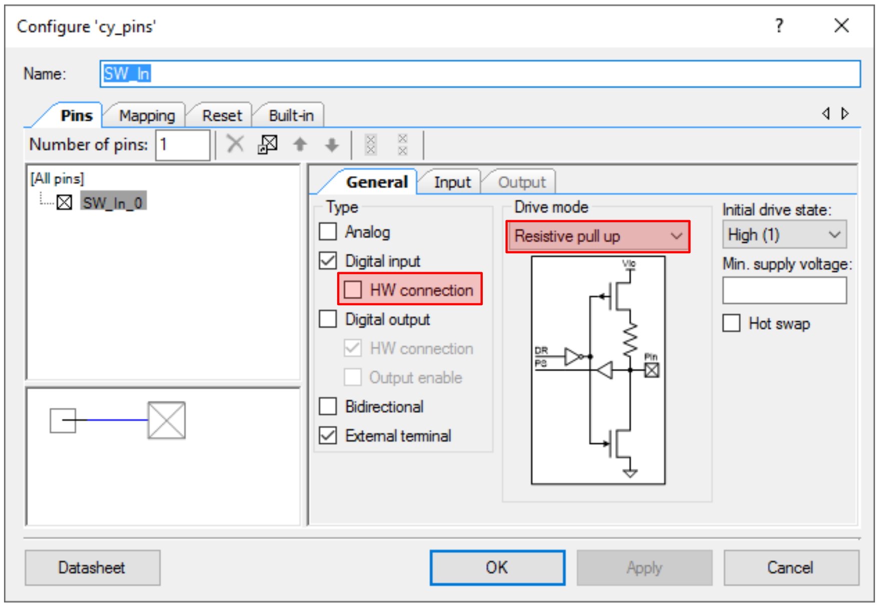
API Generated: Yes

Uses Clock Tree Resource: Yes

By default, all clocks are marked as 'start on reset'. The setting can be changed in the Design Wide Resources editor.

5. In our application, two of the digital pins, SW_In and LED_2, do not have hardware connections to other components in the schematic. Therefore, we remove the hardware connection terminals to avoid generating an error when the project builds. To remove the hardware connector from the pin, double-click the **SW_In** pin. On the **Type** groupbox, clear the **HW Connection** check box as shown in Figure 25.
6. Since **SW_In** is a digital input, we must also change the drive mode. Use the **Drive Mode** drop list to select **Resistive Pull Up** as shown in Figure 25. Click **OK** to close the dialog box.

Figure 25. SW_In Input Pin Configuration



Name: SW_In

Number of pins: 1

Type

☐ Analog

☑ Digital input

☐ HW connection

☐ Digital output

☑ HW connection

☐ Output enable

☐ Bidirectional

☑ External terminal

Drive mode: Resistive pull up

Initial drive state: High (1)

Min. supply voltage:

☐ Hot swap

7. Repeat step 5 for the **LED_2** pin.

   Note: The Character LCD component does not require any special configuration to ensure proper functionality. The default configuration for the other digital output pin, LED_1, is also acceptable in this application. Do not change these components.

6.1.4 Configure Pin and Clocking Resources

With all the PSoC Creator components configured, we now configure the clocking resources and pin placements.

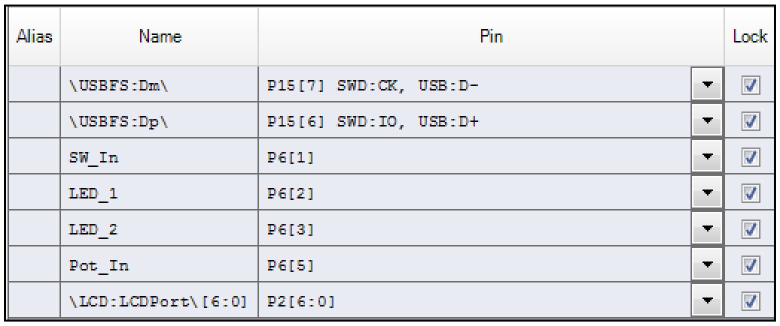
1. In the Workspace Explorer window, locate and double-click *MyFirstGenericDevice.cydwr*.

   The Pin Selection tab appears.

2. Change the pin placement to resemble Figure 26:

   This project functions on a CY8CKIT-030/CY8CKIT-050 without any jumper wires. A CY8CKIT-001 requires minimal jumper wires, as there are not any hardwired traces on the PCB to the various external components required.

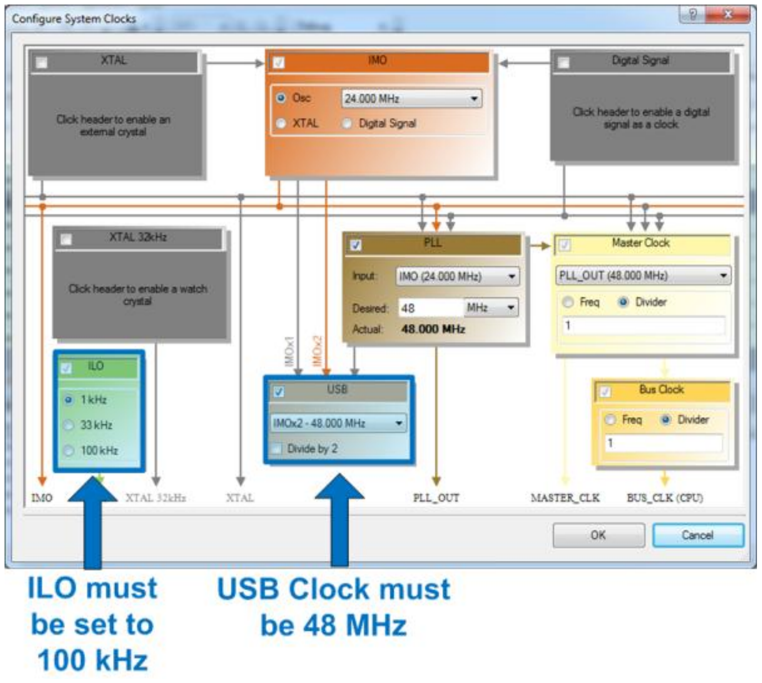
Figure 26. PSoC Project Pin Assignment



Alias	Name	Pin	Lock
	\USBFS:Dm\	P15[7] SWD:CK, USB:D-	✓
	\USBFS:Dp\	P15[6] SWD:IO, USB:D+	✓
	SW_In	P6[1]	✓
	LED_1	P6[2]	✓
	LED_2	P6[3]	✓
	Pot_In	P6[5]	✓
	\LCD:LCDPort\[6:0]	P2[6:0]	✓

3. Click on the **Clocks** tab. Double-click one of the clocks to open the configuration window. If using a PSoC 3 or PSoC 5LP device, adjust the clock settings so that the clock GUI appears as follows in Figure 27.

Figure 27. PSoC Clock Configuration



Configure System Clocks

XTAL: Click header to enable an external crystal

IMO: [✓ Osc] 24.000 MHz; XTAL; Digital Signal

Digital Signal: Click header to enable a digital signal as a clock

XTAL 32kHz: Click header to enable a watch crystal

PLL: Input: IMO (24.000 MHz); Desired: 48 MHz; Actual: **48.000 MHz**

Master Clock: PLL_OUT (48.000 MHz); Freq [✓ Divider] 1

ILO: [✓ 1 kHz]; 33 kHz; 100 kHz

USB: IMOx2 - 48.000 MHz; Divide by 2

Bus Clock: Freq [✓ Divider] 1

6.1.5 Place Code in *main.c*

With the PSoC internal hardware configured, we now configure the device firmware, specifically by adding code to the *main.c* file. Locate the code in (main.c) and place it into the *main.c* file located in PSoC Creator. *main.c* is in the **Source Files** folder in the Workspace Explorer.

With the code added to *main.c* and all the components configured, you can build the project. You should expect the process to complete successfully with no errors or warnings. Use a MiniProg3 Programmer to program the hex file into the PSoC 3 or PSoC 5LP device.

6.2 Development Kit Configuration

The project files included in this application note for PSoC 3 and PSoC 5LP devices work on two development kit platforms: CY8CKIT-030/050 and CY8CKIT-001. The CY8CKIT-003/014, which are the PSoC 3 and PSoC 5LP First Touch Kits, do not function with this project as the USB pins do not bond out to a USB connector. The USB connector present on the boards is used solely to program the device. Table 2 summarizes the kits supported by this document.

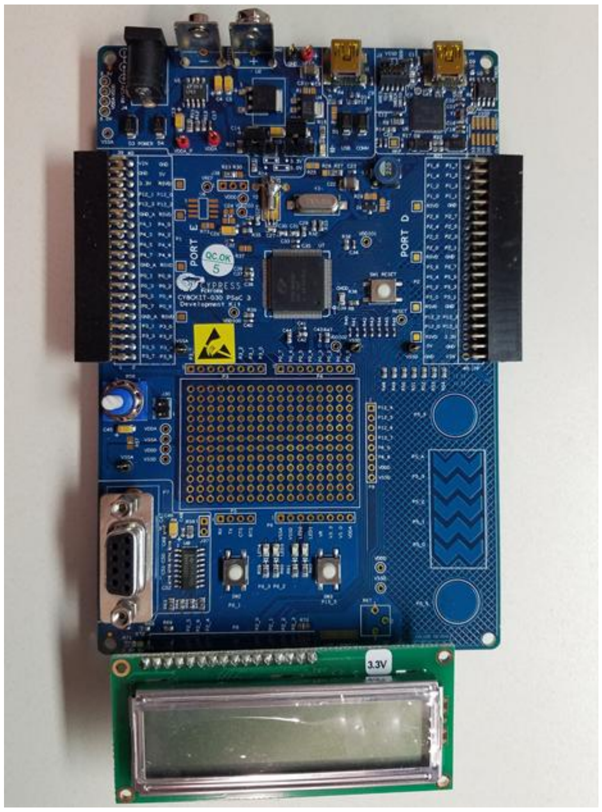
Table 2. Supported PSoC 3/PSoC 5LP Kits

Kit	Picture	Supported
CY8CKIT-003		No
CY8CKIT-001		Yes
CY8CKIT-030/050		Yes

6.2.1 Using the CY8CKIT-030/050

With CY8CKIT-030/050, the project does not require any external connections except for the USB cable. Verify that the LCD is in the LCD port (Port 2) on the bottom of the board as shown in Figure 28.

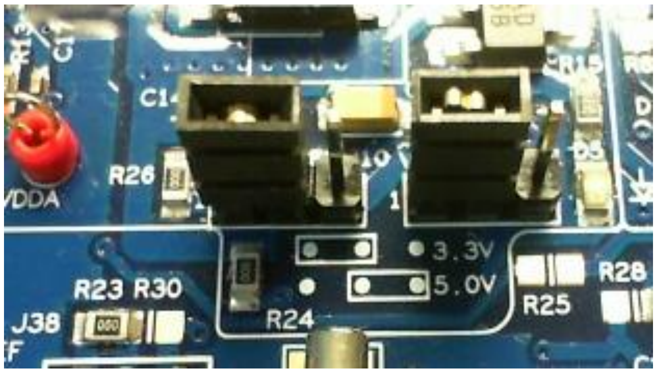
Figure 28. CY8CKIT-030/050 Configuration



Additionally, confirm that the following jumper settings are made to provide the proper power to the PSoC device. See Figure 29 for additional details.

- J10 - Move jumper to 3.3 V position
- J11 - Move jumper to 3.3 V position

Figure 29. CY8CKIT-030/050 J10 and J11 Jumpers



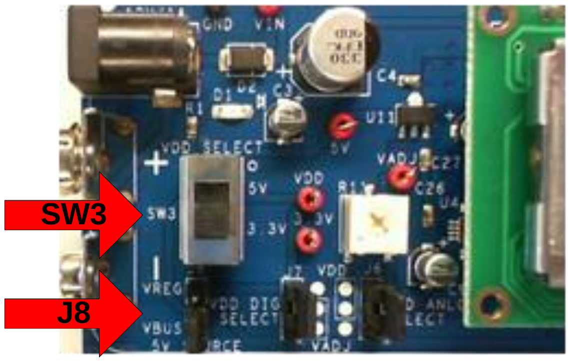
- J30 - Add jumper to provide power to potentiometer (VR)

6.2.2 Using the CY8CKIT-001

With the CY8CKIT-001, you must complete some wiring on the board prior to testing the application. The following changes are required for proper functionality. See Figure 30 for additional details:

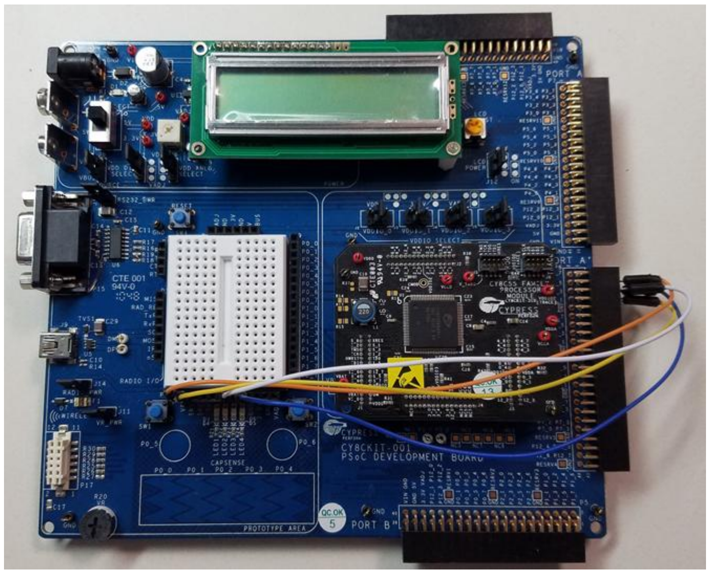
- J8 - Move jumper to VBUS position
- SW3 - Move switch to 3.3 V position
- P6[1] - Connect to SW1 using jumper wire
- P6[2] - Connect to LED1 using jumper wire
- P6[3] - Connect to LED2 using jumper wire
- P6[5] - Connect to VR using jumper wire

Figure 30. CY8CKIT-0010 J8 and SW3



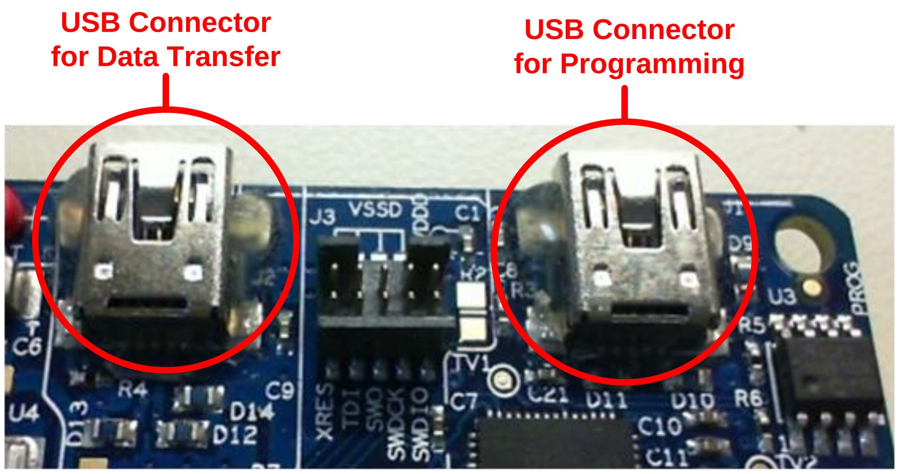
After these changes, the CY8CKIT-001 board should look similar to Figure 31. You must also verify that the Character LCD is in the LCD port socket (Port 2) at the top of the DVK as shown in Figure 31. Additionally, verify that the processor module is appropriate for evaluation with either PSoC 3 or PSoC 5LP.

Figure 31. CY8CKIT-001 Configuration



With your development kit (DVK) properly configured, attach a USB cable between a host computer and the USB interface connecter on the DVK board. Note that for the CY8CKIT-030/050 boards, which contain two USB connectors, the connector closest to the right side of the board is used to program the PSoC. The connector to the left of that is for USB data communication.

Figure 32. USB Connector Identification on CY8CKIT-030/050



6.3 Developing Desktop Applications

The next step is to develop a PC application to interface with the device so that the two can communicate. There are several decisions to make prior to development such as the selection of programming language, IDE, and USB library.

Programming Language – Selection often depends on the operating system for which you are developing. For example, Windows favors Microsoft .NET® languages such as C#, C++, and Visual Basic®. Mac OS X® favors its native object-oriented C programming language, Cocoa®, but it also includes mainstream support for Java®. While Linux® does not seem to play favorites; C is the more native language.

USB Library – The USB library interfaces with the USB device. A library is a compiled set of function calls that make it easy to interface with the device. While it is possible to write your own library to interface with the device, often this is neither necessary nor practical. There are many libraries available. Some have specific programming language requirements. Table 3 shows a list of some of the more popular USB libraries and the languages they support.

Table 3. List of Commonly Used USB Libraries

USB Library	Operating System Support	Programming Language Support
CyUSB	Windows	.NET Languages
LibUSB	Linux, Mac, Windows	C
IOKit	Mac	Objective-C
jUSB	Windows, Linux	Java
PyUSB	Linux (PyUSB), Windows (PyWinUSB)	Python®

To simplify things, this application note includes host applications for each operating system. Additionally, the USB library and programming language are dependent on the operating system. I chose the programming language based on the most commonly used language for the respective operating system. Additionally, I kept all the languages in the same C language family, using only C derivatives. Table 4 shows a list of each operating system and the programming language and library chosen.

Table 4. Application Note Example Project Usage

OS	Programming Language	USB Library
Windows	C#	CyUSB
Mac OS X	Cocoa (Objective-C)	IOKit
Linux	C	LibUSB

IDE – Recall that each host application, regardless of the operating system, displays two Input variables and controls two Output variables. With the Input Report, the application receives 20-bit ADC data and displays the decimal value on the host application. At the PSoC, the LCD can display this value. The GUI on the host also displays the status of a push button (asserted or de-asserted). With the Output Report, the host application controls the state of an LED (on or off) on a PSoC development board based on the state of a checkbox or radio button. The application also controls the Duty Cycle of a PWM; it reads a value from a text field and sends the value to the PSoC where the result appears on an LED and on a character LCD.

6.4 Microsoft Windows Host Application

I developed the Windows host application using the Microsoft Visual C#® Express IDE. It was written in the C# programming language using CyUSB as the USB library. The reason to select CyUSB for this project is its simplicity, its basis off of the Win32 API, and the readily available documentation and support from Cypress on interfacing CyUSB with Cypress products.

To run the application:

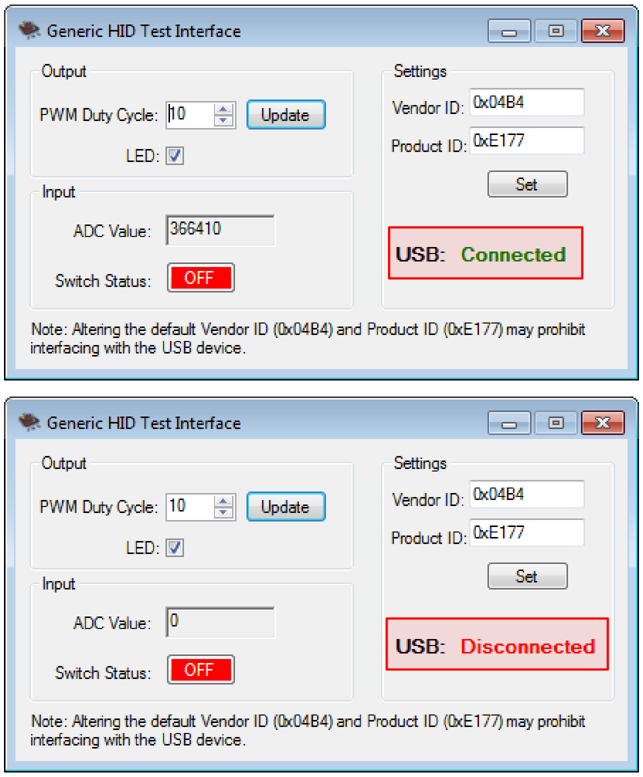
1. Verify that the Windows executable file (*Generic HID UI.exe*) and the CYUSB.dll are in the same folder.

The files are included in the project files associated with this application note. If the *.dll* and executable are not in the same folder, the application fails to launch correctly.

2. Double-click the Windows executable file in the project files.

Figure 33 shows the application when a device is connected and when it is disconnected.

Figure 33. Generic HID Test Interface in Windows



Generic HID Test Interface

Output
PWM Duty Cycle: 10 Update
LED: ☑

Input
ADC Value: 366410
Switch Status: OFF

Settings
Vendor ID: 0x04B4
Product ID: 0xE177
Set

USB: Connected

Note: Altering the default Vendor ID (0x04B4) and Product ID (0xE177) may prohibit interfacing with the USB device.

Generic HID Test Interface

Output
PWM Duty Cycle: 10 Update
LED: ☑

Input
ADC Value: 0
Switch Status: OFF

Settings
Vendor ID: 0x04B4
Product ID: 0xE177
Set

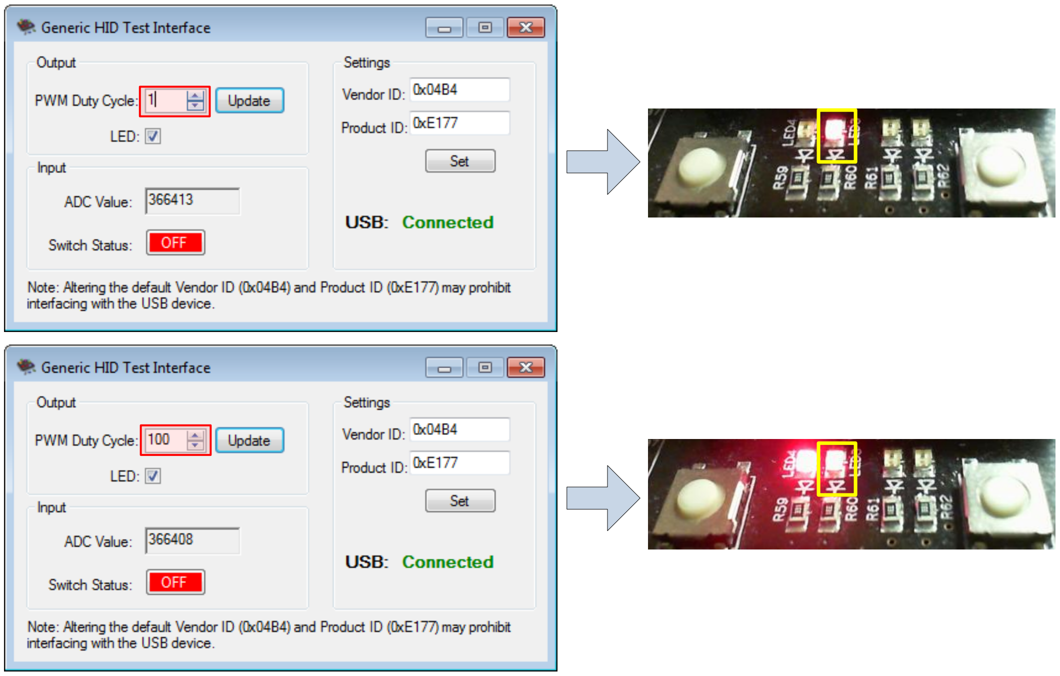
USB: Disconnected

Note: Altering the default Vendor ID (0x04B4) and Product ID (0xE177) may prohibit interfacing with the USB device.

3. By default, the application uses the VID and PID shown in Figure 33 (VID = 0x04B4 and PID = 0xE177). If you use the USBFS configuration wizard in the PSoC to change the VID and PID, edit the values in the **Settings** section to match your PSoC entries. Then click the **Set** button.

4. To control the PWM duty cycle and the LED state on the PSoC, use the **Output** section. In the **PWM Duty Cycle** text box, type a new value or use the arrows to increase or decrease the value as shown in Figure 34. Click the **Update** button to perform the Output transfer. An acceptable range for this field is between 1 and 100. The change takes effect on LED_3 on CY8CKIT-030/050 kits.

Figure 34. Host Tool Control of LED Duty Cycle


Generic HID Test Interface

Output

PWM Duty Cycle: 1 Update

LED: ☑

Input

ADC Value: 366413

Switch Status: OFF

Settings

Vendor ID: 0x04B4

Product ID: 0xE177

Set

USB: Connected

Note: Altering the default Vendor ID (0x04B4) and Product ID (0xE177) may prohibit interfacing with the USB device.

Generic HID Test Interface

Output

PWM Duty Cycle: 100 Update

LED: ☑

Input

ADC Value: 366408

Switch Status: OFF

Settings

Vendor ID: 0x04B4

Product ID: 0xE177

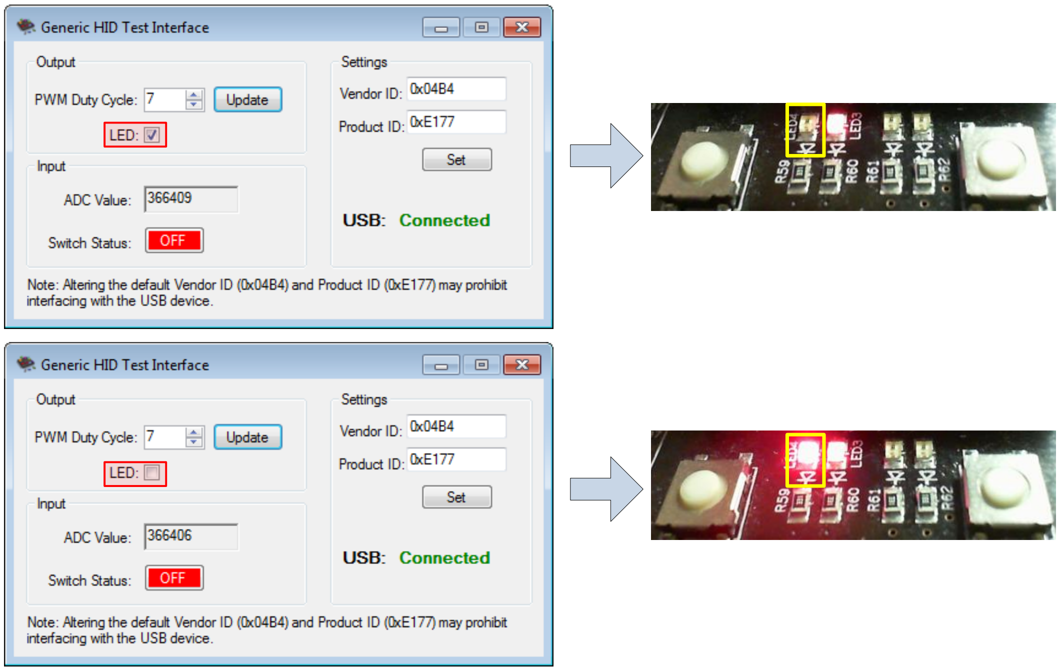
Set

USB: Connected

Note: Altering the default Vendor ID (0x04B4) and Product ID (0xE177) may prohibit interfacing with the USB device.

5. Underneath the PWM Duty Cycle field is the on/off control for LED_4 on the CY8CKIT-030. To turn the LED on, check the LED check box. To turn the LED off, clear the LED check box. Refer to Figure 35 for more details.

Figure 35. Host Tool Control of LED State (On/Off)


Generic HID Test Interface

Output

PWM Duty Cycle: 7 Update

LED: ☑

Input

ADC Value: 366409

Switch Status: OFF

Settings

Vendor ID: 0x04B4

Product ID: 0xE177

Set

USB: Connected

Note: Altering the default Vendor ID (0x04B4) and Product ID (0xE177) may prohibit interfacing with the USB device.

Generic HID Test Interface

Output

PWM Duty Cycle: 7 Update

LED: ☐

Input

ADC Value: 366406

Switch Status: OFF

Settings

Vendor ID: 0x04B4

Product ID: 0xE177

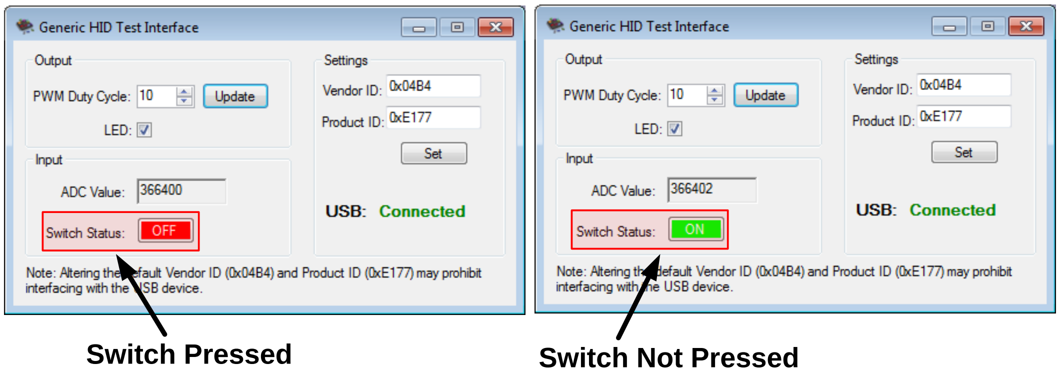
Set

USB: Connected

Note: Altering the default Vendor ID (0x04B4) and Product ID (0xE177) may prohibit interfacing with the USB device.

6. There are two variables in the **Input** section: ADC Value and Switch Status. These are for observation only. You cannot edit these fields in the host application. By default, the configuration of the ADC in the PSoC project is for 20-bits of resolution. The Switch Status indicator is a bar that is green when SW2 on the CY8CKIT-030/050 is pressed and is red when the button is released, as shown in Figure 36.

Figure 36. Host Tool Display of Push Button Status



6.5 Mac OS X Host Application

We developed the Mac OS host application using XCode® (Version 4.2.1), which is the integrated development environment for Apple Mac and iOS® devices. It is written in the Cocoa programming language, an Objective-C based language, with IOKit as the USB library. Specifically, it uses a subset of IOKit called IOHIDLib.

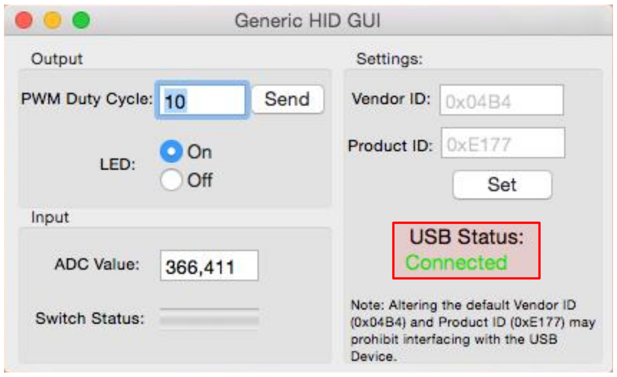
To run the application:

1. Double-click the application file *Generic HID Test Interface.app* in the project files.

   The file is included in the project files associated with this application note.

   Figure 37 shows the application when the device is connected.

Figure 37. Generic HID GUI in Mac OS X



Generic HID GUI

Output

PWM Duty Cycle: 10 Send

LED: [✓ On] Off

Input

ADC Value: 366,411

Switch Status:

Settings:

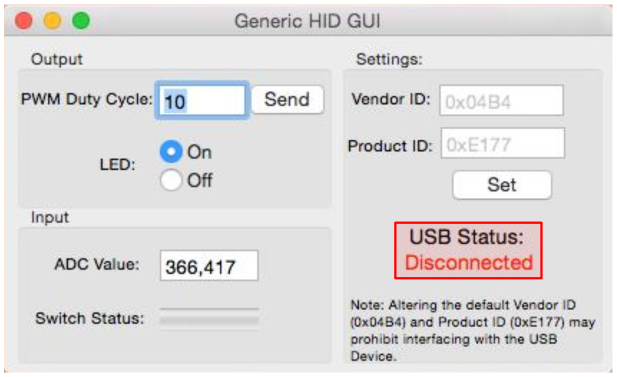
Vendor ID: 0x04B4

Product ID: 0xE177

Set

USB Status: Connected

Note: Altering the default Vendor ID (0x04B4) and Product ID (0xE177) may prohibit interfacing with the USB Device.



Generic HID GUI

Output

PWM Duty Cycle: 10 Send

LED: [✓ On] Off

Input

ADC Value: 366,417

Switch Status:

Settings:

Vendor ID: 0x04B4

Product ID: 0xE177

Set

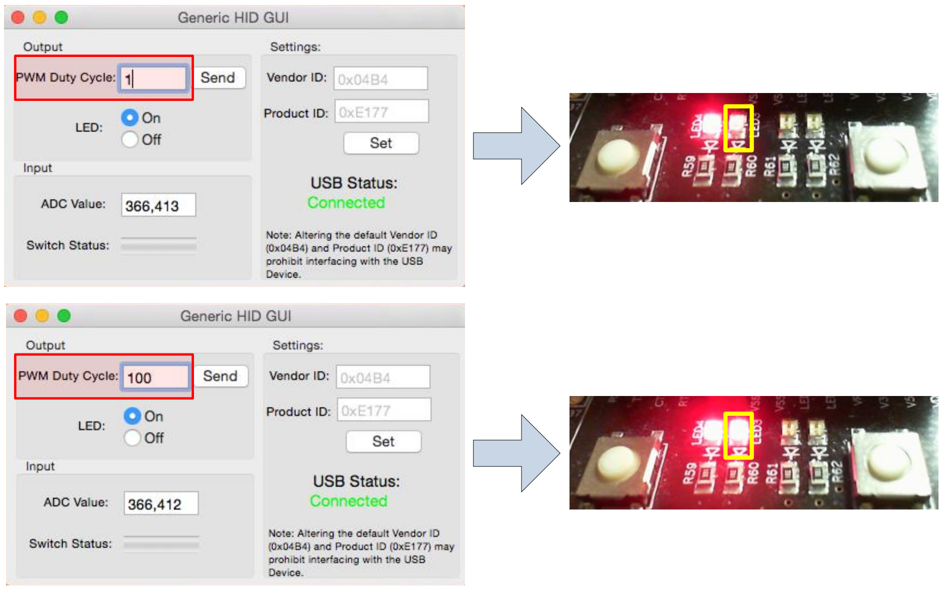
USB Status: Disconnected

Note: Altering the default Vendor ID (0x04B4) and Product ID (0xE177) may prohibit interfacing with the USB Device.

2. By default, the application uses the VID and PID described earlier in this application note (VID = 0x0484 and PID = 0xE177). If you use the USBFS configuration wizard in the PSoC to change the VID or PID, edit the values in the **Settings** section to match your PSoC entries. Then click the **Set** button.
3. Use the **Output** section to control the PWM duty cycle and LED state. In the **PWM Duty Cycle** text box, type a new value as shown in Figure 38. Click the **Send** button to perform the Output transfer.

   An acceptable range for this field is between 1 and 100. Do not use the % symbol. The change takes effect on LED_3.

Figure 38. Host Tool Control of LED Duty Cycle



Generic HID GUI

Output

PWM Duty Cycle: 1 Send

LED: [✓ On] Off

Input

ADC Value: 366,413

Switch Status:

Settings:

Vendor ID: 0x04B4

Product ID: 0xE177

Set

USB Status: Connected

Note: Altering the default Vendor ID (0x04B4) and Product ID (0xE177) may prohibit interfacing with the USB Device.

Generic HID GUI

Output

PWM Duty Cycle: 100 Send

LED: [✓ On] Off

Input

ADC Value: 366,412

Switch Status:

Settings:

Vendor ID: 0x04B4

Product ID: 0xE177

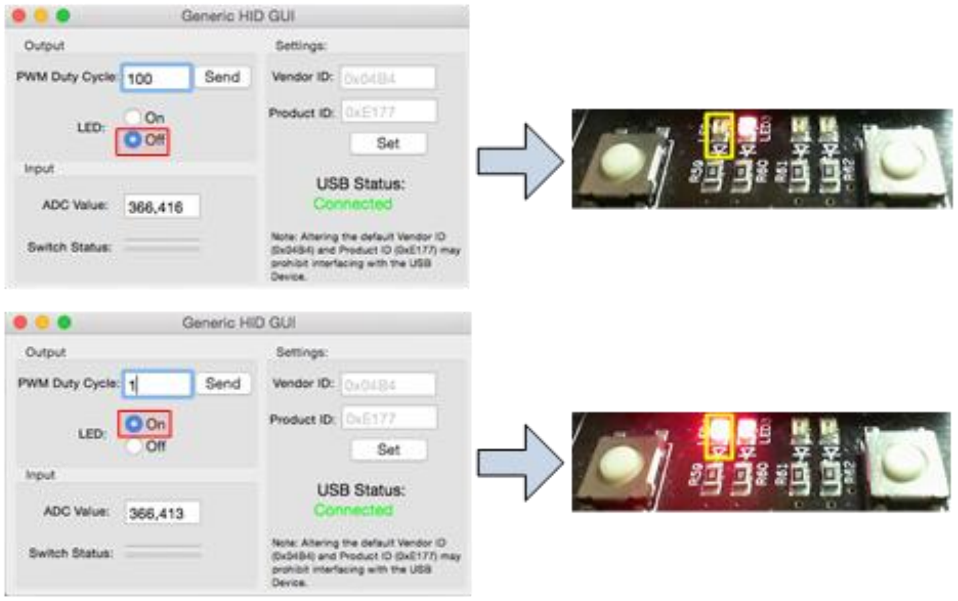
Set

USB Status: Connected

Note: Altering the default Vendor ID (0x04B4) and Product ID (0xE177) may prohibit interfacing with the USB Device.

4. Underneath the PWM Duty Cycle field is the on/off control for LED_4 on the CY8CKIT-030/50. To turn the LED on, select the **On** radio option. To turn the LED off, select the **Off** radio option. Refer to Figure 39 for more details.

Figure 39. Host Tool Control of LED State (On/Off)



Generic HID GUI

Output

PWM Duty Cycle: 100 Send

LED: On [✓ Off]

Input

ADC Value: 366,416

Switch Status:

Settings:

Vendor ID: 0x04B4

Product ID: 0xE177

Set

USB Status: Connected

Note: Altering the default Vendor ID (0x04B4) and Product ID (0xE177) may prohibit interfacing with the USB Device.

Generic HID GUI

Output

PWM Duty Cycle: 1 Send

LED: [✓ On] Off

Input

ADC Value: 366,413

Switch Status:

Settings:

Vendor ID: 0x04B4

Product ID: 0xE177

Set

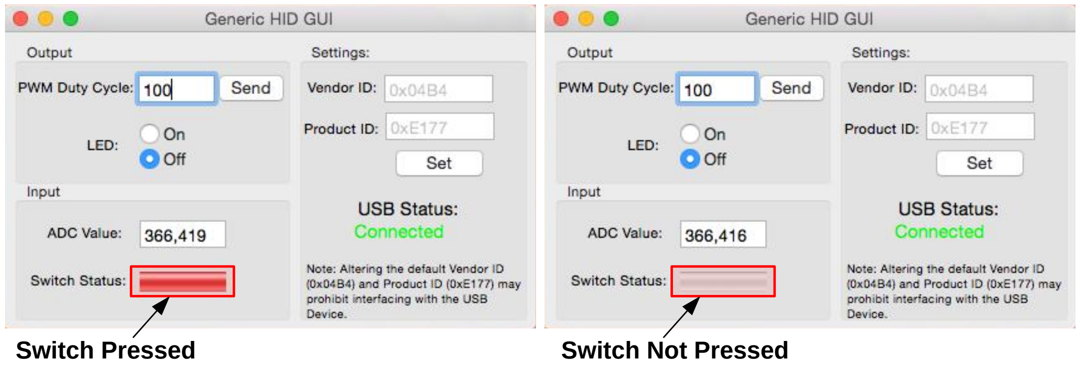
USB Status: Connected

Note: Altering the default Vendor ID (0x04B4) and Product ID (0xE177) may prohibit interfacing with the USB Device.

5. There are two variables in the **Input** section: ADC Value and Switch Status. These are for observation only. You cannot edit these fields in the host application. Use the PSoC application code to change these values.

   By default, the configuration of the ADC in the PSoC project is for 20-bits of resolution. The Switch Status indicator is a bar that is red when SW2 on the CY8CKIT-030/050 is pressed and is clear when the button is released, as shown in Figure 40.

Figure 40. Host Tool Display of Push Button Status



Generic HID GUI

Output

PWM Duty Cycle: 100 Send

LED: On [✓ Off]

Input

ADC Value: 366,419

Switch Status:

Settings:

Vendor ID: 0x04B4

Product ID: 0xE177

Set

USB Status: Connected

Note: Altering the default Vendor ID (0x04B4) and Product ID (0xE177) may prohibit interfacing with the USB Device.

Generic HID GUI

Output

PWM Duty Cycle: 100 Send

LED: On [✓ Off]

Input

ADC Value: 366,416

Switch Status:

Settings:

Vendor ID: 0x04B4

Product ID: 0xE177

Set

USB Status: Connected

Note: Altering the default Vendor ID (0x04B4) and Product ID (0xE177) may prohibit interfacing with the USB Device.

6.6 Linux PC Application

For the Linux implementation, rather than an application similar to the Windows and Mac platforms, we use a Command Line Interface (CLI) to ensure support across multiple Linux distributions (such as Ubuntu®, Gentoo™, Slackware®, and so on) and desktop environments (such as Gnome™, KDE®, XFCE™, Fluxbox and so on).The application runs entirely through the Linux terminal. To demonstrate how to stream data using a CLI, the host application performs a total of ten reads, one every second, and display the results on the terminal window. You can use the application code to change the interval between reads and the total number of reads.

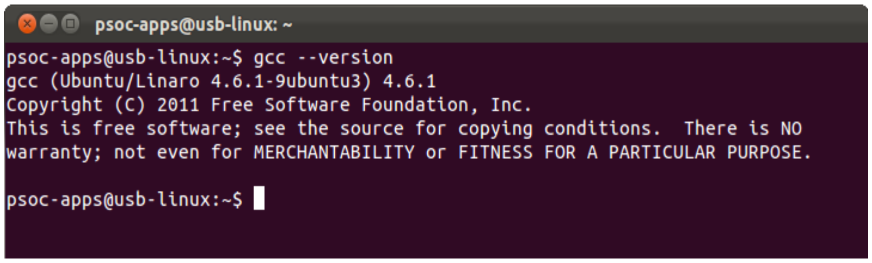
We developed the application initially using Ubuntu Linux. It is written in the C programming language using LibUSB as the USB library. You must compile the C code on your machine prior to running the application. Included with this application note is the C file *Linux_GenericHID_CLI.c*. See Linux_GenericHID_CLI.c *for Linux* for print copy of the code.

1. There are two prerequisites to run the application in Linux: GCC Compiler and LibUSB. These are available via your distributions package manager. Most distributions install a GCC compiler. On Ubuntu Linux, you can check for this dependency with the command seen in Code 1. If a GCC compiler is installed, a message appears similar to that seen in Figure 41.

Code 1. Command to Check for GCC Compiler

```
$ gcc –version
```

Figure 41. Checking for GCC Compiler in Linux



```
psoc-apps@usb-linux:~$ gcc --version
gcc (Ubuntu/Linaro 4.6.1-9ubuntu3) 4.6.1
Copyright (C) 2011 Free Software Foundation, Inc.
This is free software; see the source for copying conditions.  There is NO
warranty; not even for MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.

psoc-apps@usb-linux:~$
```

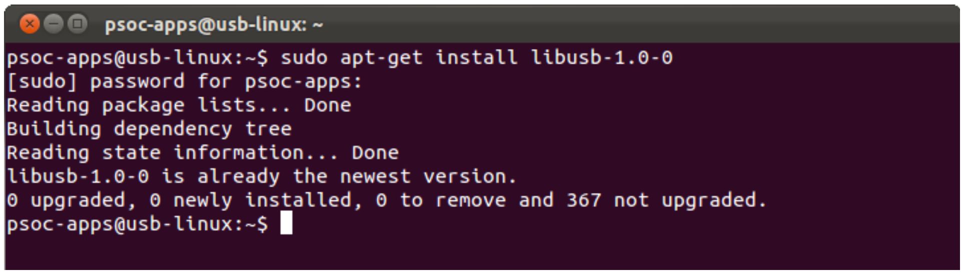
2. If not already on the host computer, you must install LibUSB. There are two methods to install LibUSB: download a copy directly from the SourceForge project page located at http://sourceforge.net/projects/libusb/ or install LibUSB via a package manager in the Linux distribution. Using Ubuntu Linux as a reference, install LibUSB with the command seen in Code 2.

   The actual terminal command varies based on the Linux distribution.

Code 2. Command to Install LibUSB

```
$ sudo apt-get install libusb-1.0-0-dev
```

Figure 42. Installing LibUSB in Ubuntu Linux



```
psoc-apps@usb-linux:~$ sudo apt-get install libusb-1.0-0
[sudo] password for psoc-apps:
Reading package lists... Done
Building dependency tree
Reading state information... Done
libusb-1.0-0 is already the newest version.
0 upgraded, 0 newly installed, 0 to remove and 367 not upgraded.
psoc-apps@usb-linux:~$
```

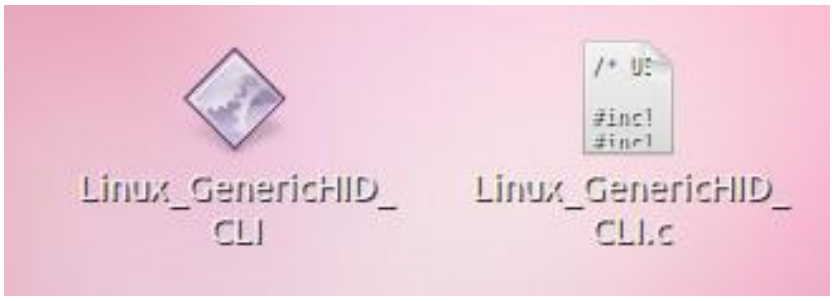
3. Use the Linux terminal to navigate to the *Linux_GenericHID_CLI.c* file. Compile the host application with the command as seen in Code 3. All of Code 3 should be placed on a single command line.

   If the code compiles correctly, a new executable file appears on the desktop as shown in Figure 43.

Code 3. Command to Compile Host Application

```
$ gcc Linux_GenericHID_CLI.c -o Linux_GenericHID_CLI -I/usr/local/include/libusb-1.0
-L/usr/local/lib -lusb-1.0
```

Figure 43. Compiled Linux Application



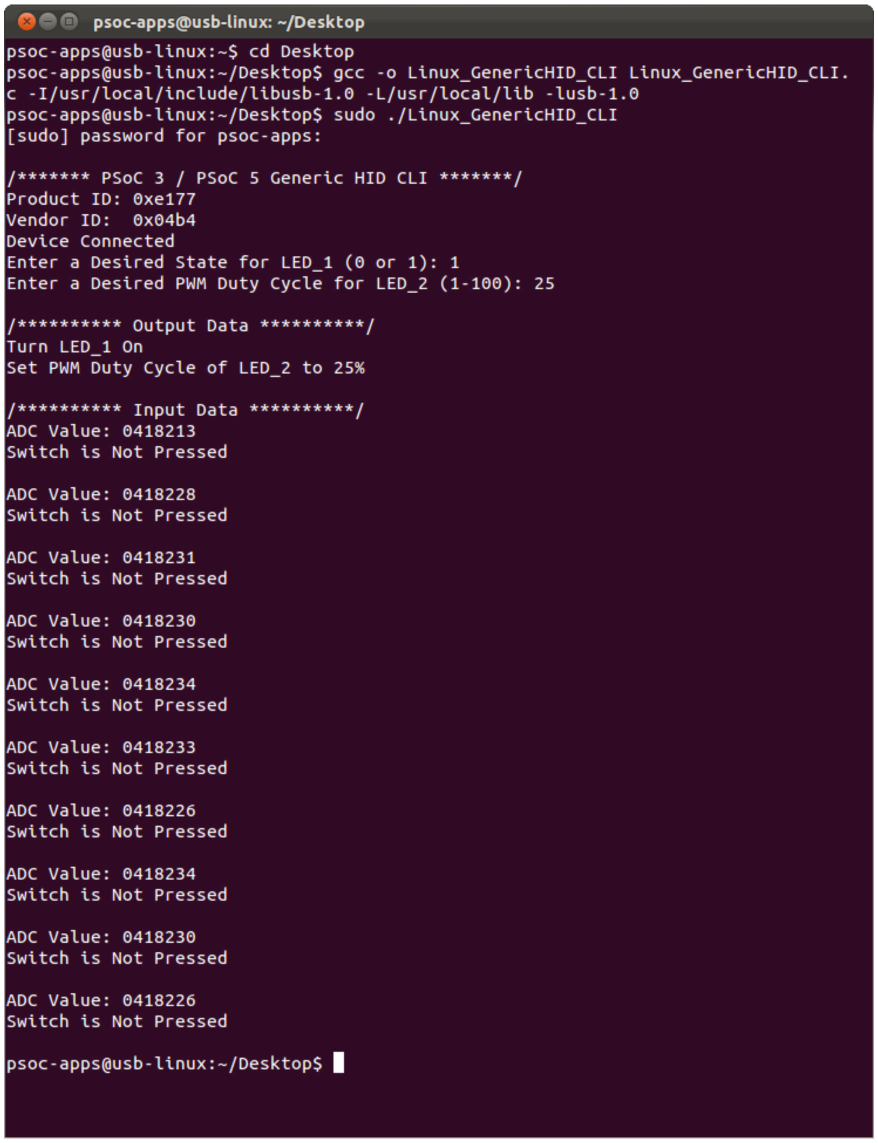
4. Run the host application from the terminal with the command as seen in Code 4.

Code 4. Command to Run Host Application

```
$ sudo ./Linux_GenericHID_CLI
```

Figure 44 shows the output of the CLI application. Use the **Output Data** section where you can view the data transferred to the PSoC. Use the **Input Data** section, you can view the data received from the PSoC device. The display also shows the multiple sequential reads which demonstrate the ability to stream continuous data.

Figure 44. Output from CLI Host Application in Linux



```
psoc-apps@usb-linux:~$ cd Desktop
psoc-apps@usb-linux:~/Desktop$ gcc -o Linux_GenericHID_CLI Linux_GenericHID_CLI.
c -I/usr/local/include/libusb-1.0 -L/usr/local/lib -lusb-1.0
psoc-apps@usb-linux:~/Desktop$ sudo ./Linux_GenericHID_CLI
[sudo] password for psoc-apps:

/******* PSoC 3 / PSoC 5 Generic HID CLI *******/
Product ID: 0xe177
Vendor ID:  0x04b4
Device Connected
Enter a Desired State for LED_1 (0 or 1): 1
Enter a Desired PWM Duty Cycle for LED_2 (1-100): 25

/********** Output Data **********/
Turn LED_1 On
Set PWM Duty Cycle of LED_2 to 25%

/********** Input Data **********/
ADC Value: 0418213
Switch is Not Pressed

ADC Value: 0418228
Switch is Not Pressed

ADC Value: 0418231
Switch is Not Pressed

ADC Value: 0418230
Switch is Not Pressed

ADC Value: 0418234
Switch is Not Pressed

ADC Value: 0418233
Switch is Not Pressed

ADC Value: 0418226
Switch is Not Pressed

ADC Value: 0418234
Switch is Not Pressed

ADC Value: 0418230
Switch is Not Pressed

ADC Value: 0418226
Switch is Not Pressed

psoc-apps@usb-linux:~/Desktop$ █
```

Note: If you prefer an IDE for the code and application development, I recommend Eclipse, a free and open-source code editor. To add support for C, you can download an add-on from Eclipse and set up the dependencies for LibUSB. It takes some extra effort; you can find out how to accomplish this by doing some Internet research.

7 Writing Host Applications to Interface with HIDs

While this application note includes the source code for the PC applications on multiple operating systems, this application note assumes a general understanding of the specific IDEs used along with their respective programming languages. If you need information on how to use any of the IDEs or how to learn C, C#, or Objective-C, please reference other resources. A list of books at the end of this application can assist you in learning these languages. The next few sections describe the key components of the source code for the developed GUIs to interface with USB. For simplification, we exclude portions of the code to handle button presses and to manipulate the data. Refer to the sample code in the Appendices and accompanying projects for more details. The projects also include detailed source code comments.

7.1 Windows Application Code

The *CYUSB.dll* makes interfacing with a PSoC HID very simple and requires minimal code from a developer's standpoint. The developer just needs to place a few lines of code into a C# application project to create an interface to a USB HID. Be aware that if opening the Windows application code provided with this application note, the files are intended to be used with Windows Visual C# Express 2010.

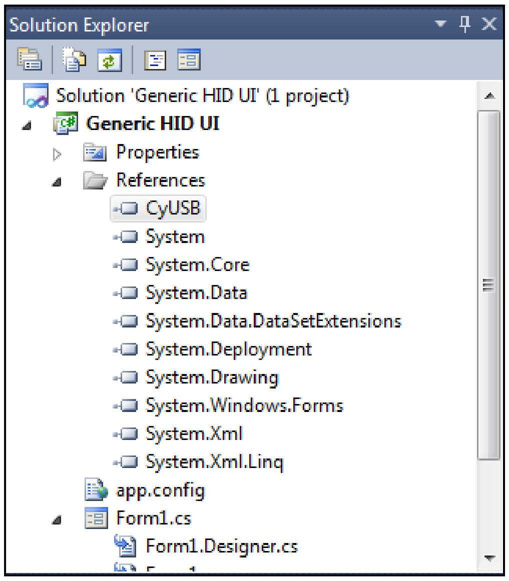
7.1.1 Add the CyUSB Reference

For the application to function properly, you must add the *CyUSB.dll* reference to your project. You can find *CyUSB.dll* either in the project files associated with this application note or in the Cypress SuiteUSB (http://www.cypress.com/?rID=34870).

1. In the Solution Explorer of Microsoft Visual C# Express, right-click the **References** folder, and select **Add Reference**.
2. Click the **Browse** tab, and then navigate to the location of the CyUSB.dll file. If you choose to use the DLL included with the application note, it is in the ../Host Applications/Windows Application folder of the associated project files.
3. Click to select *CyUSB.dll*, and then click **OK**.

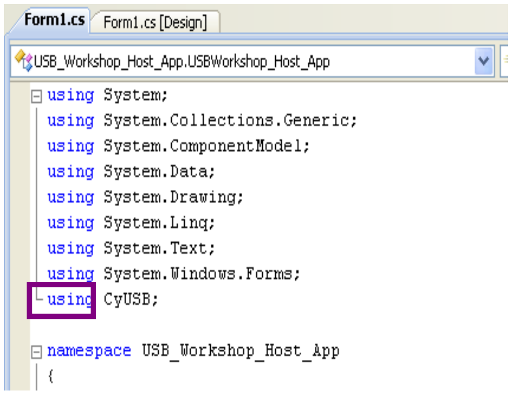
   The CyUSB reference appears in the Solution Explorer as shown in Figure 45.

Figure 45. CyUSB Reference in Solution Explorer



4. To easily access the Cypress USB library, add the CyUSB namespace. The line of code "using CyUSB", as shown in Figure 46, directs the compiler to look for functions in the CyUSB DLL without inclusion of the library name in each call. To add this, right click on the **Form1.cs** file in the Solution Explorer, click on **View Code**, and add the namespace at the top of the file.

Figure 46. CyUSB in C# Application Code



```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Linq;
using System.Text;
using System.Windows.Forms;
using CyUSB;

namespace USB_Workshop_Host_App
{
```

7.1.2 Initialize the USB HID Application

After adding a reference to the *CyUSB.dll* to the project, you can initialize the USB HID application.

1. Define four variables with the commands as seen in Code 5:
   - Vendor ID (integer)
   - Product ID (integer)
   - usbDevices (variable) – An instance of the USBDeviceList class and includes a dynamic list of USB devices that are accessible via the HID class library
   - myHidDevice – An instance of the CyHidDevice class used as a handle for devices attached to a HID driver

Code 5. Initialization Variables for Windows Example (C#)

```
      namespace WindowsFormsApplication1
      {
        public partial class
      GenericHidForm : Form
          {
          // Pointer to list of USB
      devices
          USBDeviceList usbDevices =
      null;
          // Handle of USB device
          CyHidDevice myHidDevice =
      null;
          // Cypress Vendor ID (VID)
   int VID = 0x04B4;

          // Example Project Product
      ID (PID)
```

```
            int PID = 0xE177;
        }
    }
```

2. Initialize the host application with the commands as seen in Code 6. The code completes the following tasks:
   - `InitializeComponent()` – A function call generated by the Visual C# tool that loads the GUI
   - `usbDevices.DeviceAttached` – A class instance that populates with the list of all active USB devices that conform to the standard HID class
   - `EventHandler(usbDevices_DeviceAttached)` – Creates an event handler for USB devices attached to the host
   - `EventHandler(usbDevices_DeviceRemoved)` – Creates an event handler for USB devices removed from the host
   - `GetDevice` – A function call that checks the Vendor ID and Product ID of attached devices to determine whether or not they belong with the host application.

Code 6. Application Initialization Code (C#)

```
public GenericHidForm()
{
   //Initialize the main GUI
   InitializeComponent();

   // Create a list of CYUSB devices for this application
   usbDevices = new USBDeviceList(CyConst.DEVICES_HID);

   //Add event handler for device attachment
   usbDevices.DeviceAttached += new EventHandler(usbDevices_DeviceAttached);

   //Add event handler for device removal
   usbDevices.DeviceRemoved += new     EventHandler(usbDevices_DeviceRemoved);

   //Connect to the USB device
   GetDevice();
}
```

7.1.3 Get the HID

The GetDevice function as shown in Code 7 looks through the usbDevices list for a device that matches the VID and PID defined previously in Code 5. If it finds a matching device, then myHidDevice contains the handle to that specific USB device, rather than a value of NULL. Be aware that the VID and PID are not the only way to identify a USB device. Once myHidDevice is no longer set to NULL, we then turn on the background timer and create a text label **Status** to say "Connected". The purpose of the timer is to create an interrupt where the firmware can periodically poll the device for data (typically every 10 ms based on timer configuration).

Code 7. Code to Acquire USB Device (C#)

```
public void GetDevice()
{
 //Look for device matching VID/PID
 myHidDevice = usbDevices[Vendor_ID, Product_ID]
  as CyHidDevice;

   if (myHidDevice != null)
   {
      InputTimer.Enabled = true;
      Status.Text = "Connected";
      Status.ForeColor = Color.Green;    }

   else
   {
      Status.Text = "Disconnected";
      Status.ForeColor = Color.Red;
      InputTimer.Enabled = false;
   }
}
```

7.1.4 Device Removal and Attachment Event Handlers

Code 8 shows the code for the event handler when a device is attached. The code checks to see if there is a device already attached. If there already is an attached device, then nothing needs to be done and the function exits. Otherwise, the system calls the GetDevice function to determine if the newly attached device matches the defined VID and PID.

Code 8. Event Handler for Device Attachment (C#)

```
//Handler for Device Attach
void usbDevices_DeviceAttached(object sender, EventArgs e)
{
   if (myHidDevice == null)

           {
      GetDevice();      // Process device status

           }
        }
```

Code 9 shows the code for the event handler when a device is removed. The code checks to determine whether or not the device removal affects the application. This happens when the device removed is the application device. The code casts the parameter “e” from EventArgs in the function prototype to USBEventArgs class and saves it as usbEvent. After that, it queries the PID and VID for the device removed. If it determines that the removed device was the application device, then the application no longer has a connection to the USB device and must take the appropriate actions. The application does not close. Instead, it disables the timer, updates the GUI to say “Disconnected” (by calling GetDevice), and resets the device handle to NULL.

Code 9. Event Handler for Device Removal (C#)

```
//Handler for Device Removal
void usbDevices_DeviceRemoved(object sender, EventArgs e)
{
  USBEventArgs usbEvent = e as USBEventArgs;
// Check that it was the expected device that was removed
if ((usbEvent.ProductID == PID) && (usbEvent.VendorID == VID))
  {
     // Disable interrupts for polling HID
     InputTimer.Enabled = false;
     // Set HID pointer to NULL
     myHidDevice = null;
     // Process device status
     GetDevice();
  }
}
```

7.1.5 Send and Receive Data

The host application must request data from the device as it is not sent automatically. This application uses a timer to trigger the host to send a request to the device for new data as shown in Code 10. The timer code performs a ReadInput that requests an Input Report from the device. After an Input transfer, the bytes transferred from the device are stored in the Inputs.DataBuf. DataBuf is a member of CyHidReport and serves as the data buffer for HID transfers. RptByteLen defines the length of DataBuf in bytes. RptByteLen reflects the ReportByteLength defined by the device report descriptor and is located in the XxxxReportByteLength field of the HIDP_CAPS structure reported for the Features, Inputs, or Outputs.

Code 10. InputTimer Code (Input Request) (C#)

```
private void InputTimer_Tick(object sender, EventArgs e)
{
   if (myHidDevice != null)
   {
   // Disable timer
   InputTimer.Enabled = false;

   //Query the device for new data
   myHidDevice.ReadInput();

   InputTimer.Enabled = true;
   }
}
```

ReadInput uses the Win32 ReadFile() function to read RptByteLen bytes from the device. The endpoint for this transaction is device dependent. It is also important to note that for most transfer operations DataBuf[0] contains the ReportID for the transfer. Report data begins at DataBuf[1]. This applies to Input, Output, and Feature Reports. Code 11 shows an example of how to request an Input report and then store it in a one-byte variable called myVariable.

Code 11. Code to Request Data from Device (Input) (C#)

```
myHidDevice.ReadInput();
myVariable =  myHidDevice.Inputs.DataBuf[1];
```

Sending data to the device is as simple as reading data from the device as shown in Code 12. Prior to transferring data to the device, pre-load the desired bytes into Outputs.DataBuf, and then call WriteOutput. WriteOutput uses the Win32 WriteFile() function to write RptByteLen bytes to the device. The endpoint for this transaction is device dependent.

Code 12. Code to Send Data to Device (Output) (C#)

```
//Load data into Output Buffer
myHidDevice.Outputs.DataBuf[1] = 50;
//Function call to send data to device
myHidDevice.WriteOutput();
```

7.1.6 Closing Thoughts for Windows

Now you should have the general idea of how to interface a PSoC with a Windows application using C# and the CyUSB driver. Be aware that the application code listed for Windows left out some of the code to handle the GUI functionality such as button presses, text fields, and so on. For more information, please refer to the accompanying C# project or the code shown in Form1.cs for Windows. To edit the project files, you must have a copy of Microsoft Visual C# Express®.

7.2 Mac OS X Application Code

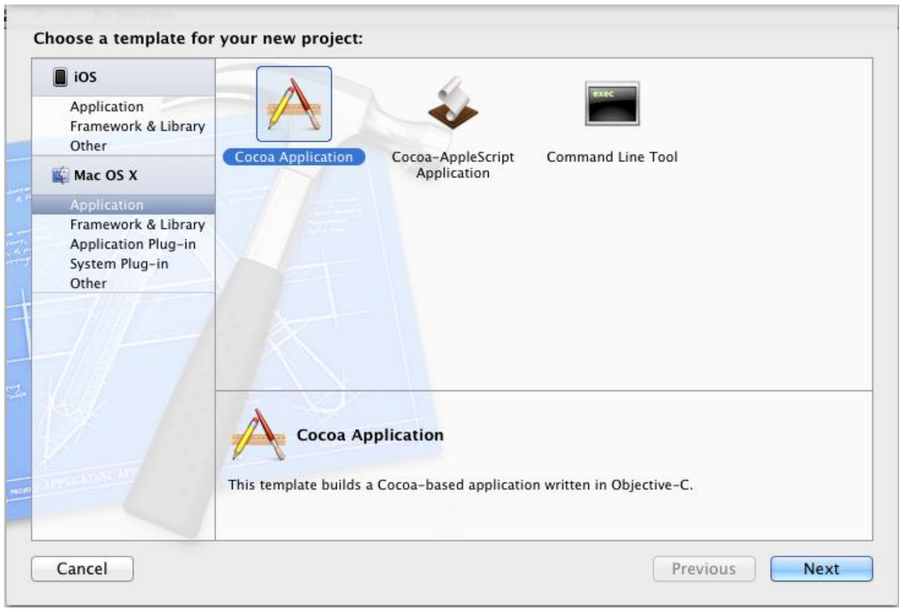
Configuring an Xcode project to properly interface with an HID is a multiple step process, and it is a bit more involved than with Windows. Upon loading the Xcode application, there are a few key steps required to interface with a HID. Be aware that if opening the OS X application code provided with this application note, the files are intended to be used with Xcode Version 4.3, which requires OS X 10.7 (Codename: Lion).

7.2.1 Add Toolkit Library and Configure HID Manager

For the application to function properly, it must include a reference to the IO Toolkit, which will act as our USB library for Mac OS X.

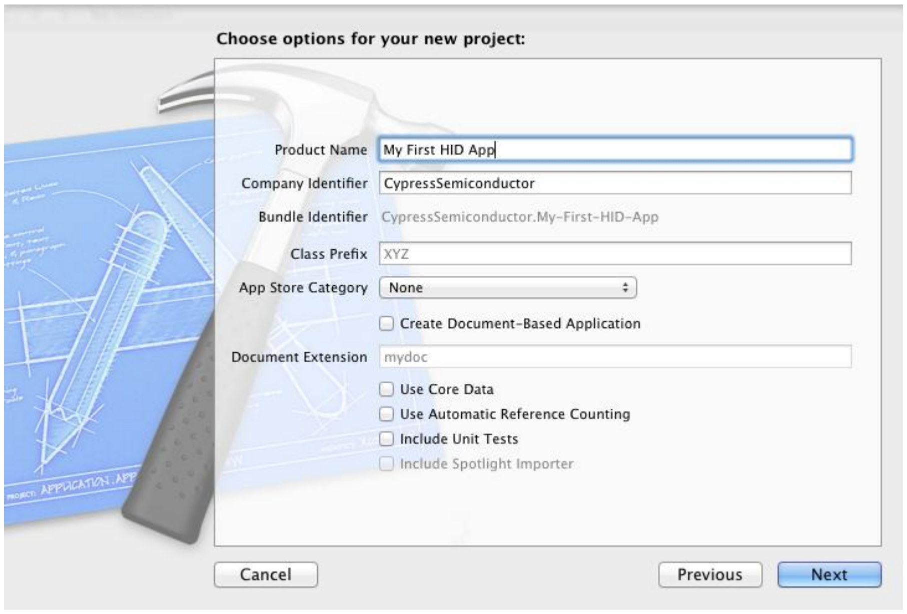
1. Presumably, we need to follow some simple steps here to open and begin configuring a new project in Xcode. To start, open Xcode, and create a new Cocoa project for Mac OS X as shown in Figure 47.

Figure 47. Creating a New Cocoa Application



Cocoa Application

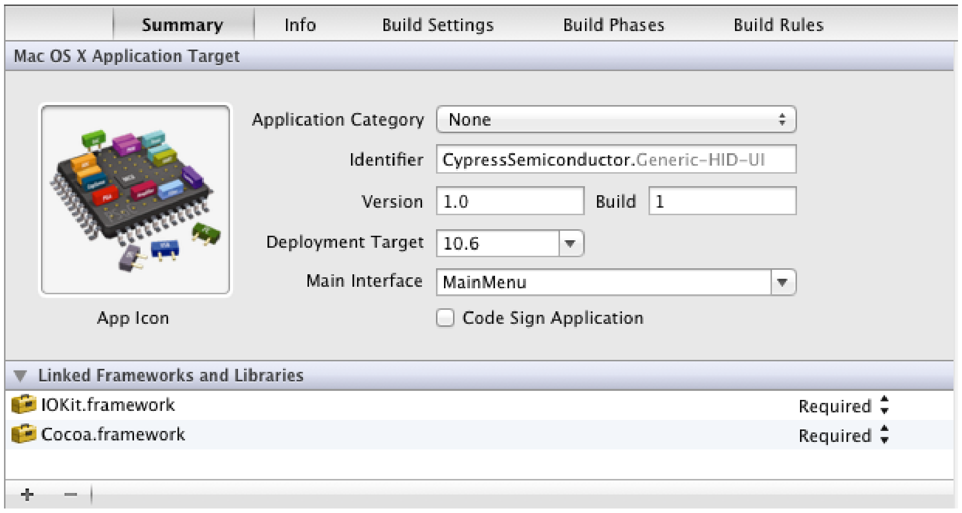
This template builds a Cocoa-based application written in Objective-C.



Product Name: My First HID App
Company Identifier: CypressSemiconductor
Bundle Identifier: CypressSemiconductor.My-First-HID-App
Class Prefix: XYZ
App Store Category: None
Create Document-Based Application
Document Extension: mydoc
Use Core Data
Use Automatic Reference Counting
Include Unit Tests
Include Spotlight Importer

2. When the new project is created, a project configuration window appears. Verify that the **Summary** tab (located on the top of Xcode) is selected. In the **Linked Framework and Libraries** section, click the **+** button and locate **IOKit.framework**. Upon selecting the proper framework file, the Xcode window looks similar to Figure 48 below.

Figure 48. Linking Apple Framework to New Application



Application Category: None
Identifier: CypressSemiconductor.Generic-HID-UI
Version: 1.0 Build: 1
Deployment Target: 10.6
Main Interface: MainMenu
Code Sign Application

Linked Frameworks and Libraries

IOKit.framework	Required
Cocoa.framework	Required

3. Once the IOKit framework has been added, the library needs to be included in the project file. To add this, click on in **AppDelegate.m** of the Xcode project. The following code snippet, shown in Code 13, is an excerpt from the application's .m file. Notice the inclusion of IOHIDLib from the IOKit library.

Code 13. Include Files for OS X Application

```
#import "AppDelegate.h"
#include <studio.h>
#include <CoreFoundation/CoreFoundation.h>
#include <IOKit/hid/IOHIDLib.h>
```

4. Setup a reference to the HID Manager with the commands in Code 14. Mac OS X includes the HID Manager to support the following:

   - HID Manager API – Provides definitions and functions an application can use to interface with HID class devices
   - Apple HID Drivers
   - In-Kernel Infrastructure – The base classes, kernel-user space memory mapping and queuing code, and the HID parser

   For application code to interface a HID with the Mac OS X, the only important consideration is the HID Manager API. The rest is handled behind the scenes by the Mac Kernel.

Code 14. HID Manager Reference Setup Code (Obj-C)

```
/**** Setup Manager and schedule it with main run loop ****/

//Define an IOHID Manager Reference
IOHIDManagerRef tIOHIDManagerRef;


//Create the HID Manager reference
tIOHIDManagerRef = IOHIDManagerCreate(kCFAllocatorDefault, kIOHIDOptionsTypeNone);


//Schedule a HID manager with current run loop.
IOHIDManagerScheduleWithRunLoop(tIOHIDManagerRef, CFRunLoopGetCurrent(),
kCFRunLoopDefaultMode);


//Open the HID manager reference and return status
tIOReturn = IOHIDManagerOpen(tIOHIDManagerRef, kIOHIDOptionsTypeNone);
```

5. The matching dictionary restricts the access of the HID Manager to only certain devices. In this case, we restrict access to only devices with a certain VID and PID. You can define a single matching dictionary or an array of dictionaries to add a more intense filtering process. Set up a matching dictionary with the commands shown in Code 15.

Code 15. Device Matching Dictionary Setup (Obj-C)

```
        // Variable Definition for VID/PID
unsigned long long VendorID = 0x04B4;

unsigned long long ProductID = 0xE177;


        //Create a dictionary
        myDictionary = [NSMutableDictionary dictionary];

        //Configure dictionary with matching criteria for Product ID
        [myDictionary setObject:[NSNumber numberWithLong:ProductID]forKey:[NSString
        stringWithCString:kIOHIDProductIDKey encoding:NSUTF8StringEncoding]];

        //Configure dictionary with matching criteria for Vendor ID
        [myDictionary setObject:[NSNumber numberWithLong:VendorID]forKey:[NSString
        stringWithCString:kIOHIDVendorIDKey encoding:NSUTF8StringEncoding]];

        //Set matching criteria (Vendor ID & Product ID)
        IOHIDManagerSetDeviceMatching(tIOHIDManagerRef, (CFMutableDictionaryRef) myDictionary);
```

Code 16. Define Buffer Pointers and Size (Obj-C)

```
        //Pointers to Input and Output Buffers in Memory
char *inputBuffer;

char *outputBuffer;

size_t bufferSize = 8;
```

7.2.2 Get the HID

With the matching criteria defined, we can acquire the HID with the commands shown in Code 17.

Code 17. Acquire Matching Device (Obj-C)

```
/******** Acquire the Device *********/

//Determine devices that match current matching criteria
NSSet * allDevices = [((NSSet *) IOHIDManagerCopyDevices(tIOHIDManagerRef))autorelease];

NSArray *deviceRefs = [allDevices allObjects];


deviceRef = ([deviceRefs count])? (IOHIDDeviceRef)[deviceRefs objectAtIndex:0]:nil;
```

Note: This code references the first device it finds that matches the criteria in the DeviceRefs array. It ignores subsequent devices with the same matching criteria (VID and PID). There is a way to handle the attachment of multiple devices that match the criteria. However, discussion of this condition is outside the scope of this application note.

1. Use the IOHIDManagerCopyDevices function to determine what devices attached to the host match the current matching criteria.
2. Create an array list of all the devices attached that match the criteria.
3. If a device matches the criteria, set the deviceRef reference to that device.
4. The last line of code acts as an IF/ELSE statement to evaluate if any devices are detected with the matching criteria. If so, then deviceRef points to the first element in the array. If no devices match the criteria, then deviceRef points to nil (equivalent to null).

7.2.3 Create Buffers for Data

Now, we set up buffers for the Input and Output data.

1. Create a pointer to the buffers as shown in Code 16.
2. Use the malloc() function to allocate memory for buffers with the commands shown in Code 18.

The *malloc()* function allocates bytes of memory and returns a pointer to the allocated memory. The parameter passed defines the size of the memory allocated.

Code 18. Allocate Input and Output Buffer Size (Obj-C)

```
/***** Allocate Memory Space for Buffers *****/
inputBuffer = malloc(bufferSize);

outputBuffer = malloc(bufferSize);
```

7.2.4 Set Up Application to Receive Data

Code 19 shows how to register a routine that is called when the HID receives an Input Report. This routine takes the form of a callback function. Because this function will continuously be called automatically, we only need to call this particular function shown below once in order to properly register it as a callback function. To aid in understanding what the variables being passed in the callback register function, refer to Table 5.

Code 19. HID Get Report Callback Setup (Obj-C)

```
/** Configure Callback for Input Reports **/
IOHIDDeviceRegisterInputReportCallback(deviceRef,(uint8_t *)inputBuffer, bufferSize,
MyInputCallback, NULL);
```

Table 5. IOHIDDeviceRegisterInputReportCallback()

Parameter	Name	Purpose
1st	deviceRef	The HID reference
2nd	inputBuffer	The address of where to store the Input report
3rd	MyInputCallback	The callback routine
4th	NULL	A user context parameter passed to the callback routine

After the host receives data from an Input report, you can reference the data in the array inputBuffer[7..0] to use it. The manipulation of the data is typically handled in the callback function.

7.2.5 Set Up Application to Send Data

Code 20 shows how to configure for an Output report. We use a synchronous IOHIDDeviceSetReport() function. You can also use the asynchronous IOHIDDeviceSetValueWithCallback(). To aid in the understanding of the variables being passed in the SetReport function, refer to Table 6.\

Code 20. Sending a HID Report Code Example (Obj-C)

```
      /***** Send the HID Report (Output) *****/
IOHIDDeviceSetReport(deviceRef, kIOHIDReportTypeOutput, 0, (uint8_t*)outputBuffer,
bufferSize);
```

Table 6. IOHIDDeviceSetReport()

Parameter	Name	Purpose
1st	deviceRef	The HID reference
2nd	kIOHIDReportTypeOutput	The IOHIDReportType object for the report
3rd	0	The Report ID
4th	outputBuffer	The address of the Output report buffer
5th	bufferSize	The size of the report being sent

7.2.6 Device Removal and Attachment Callbacks

While the initialization code works well for a device plugged in during the application launch, we must now implement code to accommodate an HID attached or removed while the application runs.

1. Declare two functions called when one a device is attached or removed with the commands in Code 21. This can be little tricky as it requires a mixture of C code with Objective-C. The first step is to create callback functions in C that in turn call class functions in Objective-C.

Code 21. C Callback Function for Attach and Removal Events (Obj-C)

```
       /* Function to execute initHID upon device matching */
static void Handle_DeviceMatchingCallback(void *inContext, IOReturn inResult, void
*inSender, IOHIDDeviceRef inIOHIDDeviceRef)

       {
           /* Call the Class Method */
    [(AppDelegate *) inContext deviceMatchingResult:inResult sender:inSender
device:inIOHIDDeviceRef];

       }
static void Handle_DeviceRemovalCallback(void *inContext, IOReturn inResult, void
*inSender, IOHIDDeviceRef inIOHIDDeviceRef)

       {
           /* Call the Class Method */
    [(AppDelegate *) inContext deviceRemovalResult:inResult sender:inSender
device:inIOHIDDeviceRef];

       }
```

2. Call the Objective-C class function that handles the appropriate action for attachment or removal with the commands in Code 22.

Code 22. Objective-C Function for Attach and Removal Events

```
       /* Function to execute initHID upon device matching */
- (void) deviceMatchingResult: (IOReturn) inResult sender: (void *) inSender device:
(IOHIDDeviceRef) inIOHIDDeviceRef

       {
   [self initHID];

       }


       /* Function to execute termHID upon device removal */
- (void) deviceRemovalResult: (IOReturn) inResult sender: (void *) inSender device:
(IOHIDDeviceRef) inIOHIDDeviceRef

       {
           [self termHID];
}
```

3. The Objective-C function `initHID` executes the code required to acquire a HID based on the matching criteria. The function `termHID` executes the code required to clear `deviceRef` when a formally connected and matched HID is removed. Once those functions are defined, the next step is to register the callbacks with the commands in Code 23.

Code 23. Register Callbacks for Attachment and Removal (Obj-C)

```
IOHIDManagerRegisterDeviceMatchingCallback(tIOHIDManagerRef, Handle_DeviceMatchingCallback,
self);


IOHIDManagerRegisterDeviceRemovalCallback(tIOHIDManagerRef, Handle_DeviceRemovalCallback, self);
```

7.2.7 Add and Respond to a Timer Event

The final step to develop a USB HID OS X application is to configure and use a timer. A timer in a USB application on the host serves multiple purposes. Commonly, a timer periodically polls for available data on the HID and then reads data from an endpoint if present. We adopted this method in our Windows application. Since we used a callback for the input data in the Mac OS application, we use the timer to periodically update the GUI with input data. Our example is set to refresh every 10 ms. Configure the timer with the commands in Code 24.

Code 24.Setup Timer Event Code Example (Obj-C)

```
//Start Timer
timer = [NSTimer scheduledTimerWithTimeInterval:0.01
                                   target:self
                                   selector:@selector(idleTimer☺
                                   userInfo:nil
                                   repeats:YES];
```

We must also define a function for the timer to jump to every time it ticks. Since we configured the timer to jump to itself, we used the following function (idleTimer). We can see this function in Code 25. Inside the function, the various GUI items that are to be updated, such as text fields and progress bars, would be placed here. Another key configuration is to set “repeats” to “YES” which will ensure that this timer is continuous and not just a single shot.

Code 25. Function for Timer Ticks (Obj-C)

```
        /**** Function for Timer Ticks ****/
- (void) idleTimer: (NSTimer *) inTimer

        {
            //Place code to update GUI
}
```

7.2.8 Closing Thoughts for Mac OS X

For more information on the implementation on Mac OS X, including the source code, please see AppDelegate.c for *Mac* OS X or take a look at the Xcode project included with the project files. You can also find additional resources in Xcode such as documentation and example projects maintained by Apple

7.3 Linux Application Code

Unlike the Windows and Mac host applications, the Linux host application does not use an IDE. We use a standard text editor to implement the C. When the time comes to build, compile, and run the application, we rely on the Linux terminal.

7.3.1 USB HID Application Initialization

1. Prior to writing the main application code, we must add four includes to the C file. While *stdio.h* and *stdlib.h* are standard for many commonly used C functions, the *libusb.h* is essential to this application. Access to the libusb API makes this application function. The application uses *time.h* to add a delay between the multiple Input transfers. Code 26 shows the include files used in the Linux application.

Code 26. Include Files for LibUSB Application

```
#include <stdio.h>
#include <stdlib.h>
#include <libusb.h>
#include <time.h>
```

2. Declare four constant values as shown in Code 27:
   - INTERFACE_NUMBER – The interface number of the device we want to communicate with. On a composite device, this can be one of a range of numbers. Since we only have a single interface on this device, the Interface Number is 0.
   - TIMEOUT_MS – The amount of time in milliseconds (after performing an Output or Input request) the application waits for a response before giving up and moving on.
   - NUMBER_INPUT_TRANSACTIONS – The number of times to perform an Input transaction before releasing the device.
   - TIME_DELAY – The delay interval in seconds between each Input request.

Code 27.Global Variables for Linux Application RECEIVING

```
static 51ons tint INTERFACE_NUMBER = 0;

static 51ons tint TIMEOUT_MS = 5000;

static 51ons tint NUMBER_INPUT_ITERATIONS = 10;

static 51ons tint TIME_DELAY = 1;
```

3. Next we need to define a few more key variables for our application. First, we need to define the VID and PID for the application. These variables have been preset and configured as constants, rather than asking for a VID and PID each time someone wants to run the application. We also need to define a device handle and a couple status flags to be used in the application as shown in Code 28.

Code 28. Local Variables for Linux Application RECEIVING

```
       /* Initialize and Set Default Vendor ID and Product ID */

const int VID = 0x04B4;

const int PID = 0xE177;


struct libusb_device_handle *devh = NULL;

int device_ready = 0;

int result = 0;
```

4. Use the Vendor ID and Product ID to initialize the device handler to a specific device. It returns a handle for the first matching device found. After verifying that the device handler is no longer NULL, detach the device from the kernel. Once the device is free from the kernel, the final step is to claim the device for our application. The commands to complete this series of tasks is shown in Code 29.

Code 29. Acquiring Device for Linux Application RECEIVING

```
result = libusb_init(NULL);


devh = libusb_open_device_with_vid_pid(NULL, VID, PID);
```

```
if (devh != NULL)

        {
            libusb_detach_kernel_driver(devh, 0);
            result = libusb_claim_interface(devh, 0);

            if(result != 0)
            {
               device_ready = 1;
            }
}
```

5. Further in the main loop, check to see if LibUSB successfully acquired the device. If so, call the function Perform_Input_and_Output_Transfers and pass the device handler so the application knows which device to communicate with. After executing the function, release the interface. This sequence is shown in Code 30.

Code 30. Performing Input/Output Transfers RECEIVING

```
if(device_ready)

        {
           //Send and receive data
           Perform_Input_and_Output_Transfers(devh);

           // Finished using the device.
   Libusb_release_interface(devh, 0);

        }
```

6. Table 7 describes additional variables required by the Perform_Input_Output_Transfers() function. We must also define a buffer to store the data prior to sending it to the device and after receiving it from the device. Finally, there are variables to declare for status flags and to store information on the button status and ADC result. Define these variables and buffers with the commands in Code 31.

Table 7. Variables Required by Perform_Input_and_Output_Transfers()

Variable	Purpose	Value
INTERRUPT_IN_ENDPOINT	Address of the Input endpoint	0x81
INTERRUPT_OUT_ENDPOINT	Address of the Output endpoint	0x02
MAX_INTERRUPT_IN_TRANSFER_SIZE	Maximum packet size for data received	8 (bytes)
MAX_INTERRUPT_OUT_TRANSFER_SIZE	Maximum packet size for data sent	8 (bytes)

Code 31. Linux Code for Variable Declarations of Transfers

```
        // Assign Endpoint Addresses
static 53ons tint INTERRUPT_IN_ENDPOINT = 0x81;

static 53ons tint INTERRUPT_OUT_ENDPOINT = 0x02;


        // With firmware support, transfers can be > the endpoint's max packet size.

Static 53ons tint MAX_INTERRUPT_IN_TRANSFER_SIZE = 8;

static 53ons tint MAX_INTERRUPT_OUT_TRANSFER_SIZE = 8;


unsigned char data_in[MAX_INTERRUPT_IN_TRANSFER_SIZE];

unsigned char data_out[MAX_INTERRUPT_OUT_TRANSFER_SIZE];


int bytes_transferred;

int result = 0;


int ADC_Result = 0;

int Switch_Status = 0
```

In Code 32, we load the OUT buffer based on code received from the CLI. With the buffer loaded, call libusb_interrupt_transfer() to perform an output transaction based on the parameters passed. The buffer stores the data in the data_out[ ] array prior to transfer.

Code 32. LibUSB Code for Out Transfer

```
data_out[0]= LED_State;
data_out[1] = PWM_DutyCycle;
result = libusb_interrupt_transfer(devh,
              INTERRUPT_OUT_ENDPOINT,
              data_out,
                        MAX_INTERRUPT_OUT_TRANSFER_SIZE,
              &bytes_transferred,
                        TIMEOUT_MS);
```

7. In Code 33, we can see a similar use of the libusb_interrupt_transfer function to perform an IN transaction. With the buffer loaded, call libusb_interupt_transfer() to perform an input transaction based on the parameters passed. The buffer stores the data after the transfer in the data_in[ ] array.

Code 33. LibUSB Code for In Transfer

```
result = libusb_interrupt_transfer(devh,
              INTERRUPT_IN_ENDPOINT,
              data_in,
                        MAX_INTERRUPT_OUT_TRANSFER_SIZE,
              &bytes_transferred,
                        TIMEOUT_MS);
```

Both the IN and OUT transfers use the function libusb_interrupt_transfer. The function prototype for libusb_interrupt_tranfser is shown in Code 34. In this function, the parameters are as follows:

- dev_handle – A handle for the device to communicate with.
- endpoint – The address for a valid endpoint to communicate with.
- data - Data buffer for either input or output data.
- length – In the case of an Output, the number of bytes to transfer in the data buffer. In the case of an Input, the number of bytes to receive in the data buffer.
- transfer – Output location for the number of bytes that were actually transferred.
- timeout – timeout in milliseconds that the function should wait before giving up due to no response being received.

Code 34. LibUSB Code for In Transfer

```
int libusb_interrupt_transfer ( struct libusb_device_handle *  dev_handle,
                                unsigned char                  endpoint,
                                unsigned char *                data,
                                int                            length,
                                int *                          transferred,
                                unsigned int                   timeout
                              )
```

8 Conclusion

This application note discusses the benefits of using the HID class for general data communication. After reading this application note, you will be more comfortable using the various operating applications with the PSoC device. Users who want to explore building upon the designs provided can adjust the maximum packet size to send more data. Another option is to build upon the host applications provided and the available space in the Input and Output reports to send data to perform additional functionality.

9 Related Resources

9.1 Application Notes

- AN57294 – USB 101: An Introduction to Universal Serial Bus 2.0
- AN57473 – PSoC® 3 / PSoC 5LP USB HID Fundamentals with Mouse and Joystick
- AN58726 – PSoC® 3 / PSoC 5LP USB HID Intermediate (with Keyboard and Composite Device)
- AN56377 – PSoC® 3 / PSoC 5LP USB Transfer Types
- AN023 – USB Compliance Testing Overview
- AN14557 – Introduction to CYUSB.dll Based Application Development Using C#
- AN61744 – Introduction to CYAPI.lib Based Application Development Using VC++

9.2 Programming Language Books

- **C#:**
  - Head First C#: A Learners Guide to Real-World Programming with Visual C# and .NET
  - C# Unleashed
  - Visual C# 2012 How to Program
- **C:**
  - The C Programming Language
- **Objective-C:**
  - Cocoa Programming for Mac OS X
  - Objective C Programming: The Big Nerd Ranch Guide

About the Author

Name:	Robert Murphy
Title:	Sr. Staff Systems Engineer
Background:	Robert Murphy graduated from Purdue University with a Bachelor Degree in Electrical Engineering Technology

Appendix A. (*main.c*)

```c
#include <device.h>

#include <stdio.h>


/* Declare defines for constant variables */

#define DEVICE_ID 0

#define IN_ENDPOINT 0x01

#define OUT_ENDPOINT 0x02

#define MAX_NUM_BYTES  8

#define UnassignedAddress 0


/* Array Positioning for IN Data */

#define Button_Status_Pos 0

#define ADC_Pos_1 1

#define ADC_Pos_2 2

#define ADC_Pos_3 3

#define ADC_Pos_4 4


/* Array Positioning for OUT Data */

#define LED_State_Pos 0

#define PWM_DC_Pos 1


/* Input and Output Data Buffers */

uint8 IN_Data_Buffer[8];  /* [0] = Button Status, [1..4] = ADC Result, [5..7] = Unused */

uint8 OUT_Data_Buffer[8]; /* [0] = LED State, [1] = PWM Duty Cycle, [2..7] = Unused */


/* Display buffer for Character LCD */

char ADC_Display_Data[10];

char PWM_Duty_Display_Data[5];


/* Various Application Variables */

int32 ADC_Result;

int8 PWM_DutyCycle;

uint8 OUT_COUNT;

uint16 LCD_Update_Timer = 0x00;
```

```
/* Function Prototypes */
void Process_EP2 (void);
void Process_EP1 (void);
void Update_LCD (void);

void main (void)
{
    /*Enable Global Interrupts */
    CYGlobalIntEnable;

    /*Initalize ADC */
    ADC_Start();
    ADC_StartConvert();

    /*Initalize LCD and place static text */
    LCD_Start();
    LCD_Position(0,0);
    LCD_PrintString("PSoC Generic HID");
    LCD_Position(1,0);
    LCD_PrintString("AD:");
    LCD_Position(1,11);
    LCD_PrintString("D:");

    /*Initalize PWM */
    PWM_Start();
    PWM_WriteCompare(10);

    /*Initalize LED Off */
    LED_2_Write(1);

    /* Start USBFS Operation for the DEVICE_ID and with 5V operation  */
    USBFS_Start(0, USBFS_DWR_VDDD_OPERATION);

    /* Ensure device is configured before running code */
```

```
while(USBFS_bGetConfiguration() == 0x00)
{
    /* Waiting for device to be configured */
}

for(;;)
{
    /* Prepare the push button and ADC data to be sent to the host */
    Process_EP1();

    /*Check to see if the IN Endpoint is empty. If so, load it with Input data to be tranfered */
    if(USBFS_GetEPState(IN_ENDPOINT) == USBFS_IN_BUFFER_EMPTY)
    {
        /* Load data located in IN_Data_Buffer and load it into the IN endpoint */
        USBFS_LoadEP(IN_ENDPOINT, IN_Data_Buffer, MAX_NUM_BYTES);
        /* Enable the OUT endpoint to recieve data */
        USBFS_EnableOutEP(OUT_ENDPOINT);
    }

    /* Check to see if the OUT Endpoint is full from a recieved transaction. */
    if(USBFS_GetEPState(OUT_ENDPOINT) == USBFS_OUT_BUFFER_FULL)
    {
        /* Get the number of bytes recieved */
        OUT_COUNT = USBFS_GetEPCount(OUT_ENDPOINT);
        /* Read the OUT endpoint and store data in OUT_Data_Buffer */
        USBFS_ReadOutEP(OUT_ENDPOINT, OUT_Data_Buffer, OUT_COUNT);
        /* Re-enable OUT endpoint */
        USBFS_EnableOutEP(OUT_ENDPOINT);
    }

    /* Process the data recieved from the host */
    Process_EP2();

    /* Impliment a perodic delay for updating the LCD */
    if(LCD_Update_Timer >= 0x08FF)
```

```
        {
            /* Call function to update Character LCD with ADC and PWM Duty Cycle value */
            Update_LCD();
        }

        /* Increment LCD Timer */
        LCD_Update_Timer ++;
    }
}

/***********************************************************************
* NAME: Process_EP2
*
* DESCRIPTION: Function to process the OUT data (data sent from the PC).
* This data includes the state of LED_2 (On or Off) and the duty cycle
* of the PWM attached to LED_1.
*
***********************************************************************/
void Process_EP2 (void)
{
    /* Update PWM Compare Value */
    if((OUT_Data_Buffer[PWM_DC_Pos] > 0) && (OUT_Data_Buffer[PWM_DC_Pos] <= 100))
    {
        PWM_DutyCycle = OUT_Data_Buffer[PWM_DC_Pos];
        PWM_WriteCompare(PWM_DutyCycle);
    }

    /* Check to see if LED_2 Should be on */
    if(OUT_Data_Buffer[LED_State_Pos] != 0)
    {
        LED_2_Write(1);
    }
    else
    {
        LED_2_Write(0);
    }
}

/***********************************************************************
* NAME: Process_EP1
*
* DESCRIPTION: Function to process the IN data (data sent to the PC).
* This data includes the value read from the DelSig ADC and the current
* status of push button connected to SW_In.
*
***********************************************************************/
void Process_EP1 (void)
{
    /* Check Status of Switch 1 */
    if(SW_In_Read() != 0)
    {
        IN_Data_Buffer[Button_Status_Pos] = 0x00;
    }
    else
    {
        IN_Data_Buffer[Button_Status_Pos] = 0xff;
    }

    /* Check if ADC data is ready */
    if(ADC_IsEndConversion(ADC_RETURN_STATUS) != 0x00)
    {
        /* Read ADC Sample */
        ADC_Result = ADC_GetResult32();

        /* Load Bits 7-0 of ADC_Value into the Input Buffer */
        IN_Data_Buffer[ADC_Pos_1] = (int8) 0x000000FF & (ADC_Result >>  24);

        /* Load Bits 15-8 of ADC_Value into the Input Buffer */
        IN_Data_Buffer[ADC_Pos_2] = (int8) 0x000000FF & (ADC_Result >>  16);
        /* Load Bits 23-16 of ADC_Value into the Input Buffer */
        IN_Data_Buffer[ADC_Pos_3] = (int8) 0x000000FF & (ADC_Result >>  8);

        /* Load Bits 31-24 of ADC_Value into the Input Buffer */
        IN_Data_Buffer[ADC_Pos_4] = (int8) 0x000000FF & ADC_Result;
    }
}

/***********************************************************************
* NAME: Update_LCD
*
* DESCRIPTION: Function periodically update the Character LCD with the
* ADC value and PWM duty cycle.
*
***********************************************************************/
void Update_LCD (void)
{
    /* Convert ADC Value to CHAR and display on Character LCD */
    sprintf(ADC_Display_Data, "%-7ld", ADC_Result);
    LCD_Position(1,3);
    LCD_PrintString(ADC_Display_Data);

    /* Convert PWM Duty Cycle to CHAR and display on Character LCD */
    sprintf(PWM_Duty_Display_Data, "%-3d", (int16)PWM_ReadCompare());
    LCD_Position(1,13);
    LCD_PrintString(PWM_Duty_Display_Data);

    /* Reset LCD Update Timer */
    LCD_Update_Timer = 0x00;
}
/* [] END OF FILE */
```

Appendix B. *Form1.cs* for Windows

```
using System;

using System.Collections.Generic;

using System.ComponentModel;

using System.Data;

using System.Drawing;

using System.Linq;

using System.Text;

using System.Windows.Forms;

using CyUSB;

namespace WindowsFormsApplication1
{
    public partial class GenericHidForm : Form
    {

        // *********************** START OF PROGRAM MAIN ****************************** //

        //Constant variables for locations in Input and Output Data Arrays

        //Constants fro Input Buffer Array
        const uint Switch_Status_Position = 1;
        const uint ADC_Byte1_Position = 2;
        const uint ADC_Byte2_Position = 3;
        const uint ADC_Byte3_Position = 4;
        const uint ADC_Byte4_Position = 5;

        //Constants fro Output Buffer Array
        const uint LED_State_Position = 1;
        const uint PWM_DutyCycle_Position = 2;

        USBDeviceList usbDevices = null;        // Pointer to list of USB devices
        CyHidDevice myHidDevice = null;         // Handle of USB device

        uint AdcCounts_1 = 0;                   // READ:  ADC count from PSOC, ADC_Result[7..0]
        uint AdcCounts_2 = 0;                   // READ:  ADC count from PSOC, ADC_Result[15..8]
        uint AdcCounts_3 = 0;                   // READ:  ADC count from PSOC, ADC_Result[23..16]
        uint AdcCounts_4 = 0;                   // READ:  ADC count from PSOC, ADC_Result[31..24]

        uint DutyCycle = 0;                     // WRITE: Duty cycle to set PSOC PWM

        int VID = 0x04B4;                       // Cypress Vendor ID
        int PID = 0xE177;                       // Example Project Product ID

        NumericUpDown DutyCycleUpDown;          // Handler for PWM up/down control


        /***********************************************************************
        * NAME: GenericHIDForm

        *

        * DESCRIPTION: Main function called initially upon the starting of the

        * application. Used to un-initialized variables, the GUI application, register

        * the event handlers, and check for a connected device.

        *
```

```
 ***********************************************************************/

 public GenericHidForm()
 {
     //Initialize the main GUI
     InitializeComponent();

     //Set initial values
     DutyCycleUpDown = new NumericUpDown();
     DutyCycleUpDown.Value = 10;
     DutyCycleUpDown.Minimum = 0;
     DutyCycleUpDown.Maximum = 100;

     //Callback to set numeric updown box value
     DutyCycleUpDown.ValueChanged += new EventHandler(DutyCycleUpDown_OnValueChanged);

     // Create a list of CYUSB devices for this application
     usbDevices = new USBDeviceList(CyConst.DEVICES_HID);

     //Add event handlers for device attachment and device removal
     usbDevices.DeviceAttached += new EventHandler(usbDevices_DeviceAttached);
     usbDevices.DeviceRemoved += new EventHandler(usbDevices_DeviceRemoved);

     //Connect to the USB device
     GetDevice();

 }


/**********************************************************************
* NAME: DutyCycleUpDown_OnValueChanged

*

* DESCRIPTION: Event Handler for the numeric up/down text field. This

* function is called when the value in the next field is updated with

* the arrows so that the value is the text field can be properly updated

* to reflect the change.

*

***********************************************************************/
 private void DutyCycleUpDown_OnValueChanged(object sender, EventArgs e)
 {
     Console.WriteLine(DutyCycleUpDown.Value);
 }

 /**********************************************************************
 * NAME: usbDevices_DeviceRemoved

 *

 * DESCRIPTION: Event handler for the removal of a USB device. When the removal

 * of a USB device is detected, this function will be called which will check to

 * see if the device removed was the device we were using. If so, then reset

 * device handler (myHidDevice), disable the timer, and update the GUI.

 *

 ***********************************************************************/

 public void usbDevices_DeviceRemoved(object sender, EventArgs e)
 {
     USBEventArgs usbEvent = e as USBEventArgs;
     if ((usbEvent.ProductID == PID) && (usbEvent.VendorID == VID))
     {
         InputTimer.Enabled = false;         // Disable interrupts for polling HID
         myHidDevice = null;
         GetDevice();                        // Process device status
     }
 }

 /**********************************************************************
 * NAME: usbDevices_DeviceAttached

 *

 * DESCRIPTION: Event handler for the attachment of a USB device. The function

 * first checks to see if a matching device is already connected by seeing

 * if the handler (myHidDevice) is null. If no device is previously attached,

 * the function will call GetDevice to check and see if a matching device was

 * attached.

 *

  **********************************************************************/

 public void usbDevices_DeviceAttached(object sender, EventArgs e)
 {
     if (myHidDevice == null)
     {
         GetDevice();                        // Process device status
     }
 }


 /**********************************************************************
 * NAME: GetDevice

 *

 * DESCRIPTION: Function checks to see if a matching USB device is attached

 * based on the VID and PID provided in the application. When a device is

 * found, it is assigned a handler (myHidDevice) and the GUI is updated to

 * reflect the connection. Additionally, if the device is not connected,

 * the function will update the GUI to reflect the disconnection.

 *

 **********************************************************************/

 public void GetDevice()
 {
     //Look for device matching VID/PID
     myHidDevice = usbDevices[VID, PID] as CyHidDevice;

     if (myHidDevice != null)                //Check to see if device is already connected
     {

         Status.Text = "Connected";
         Status.ForeColor = Color.Green;
         SwStatus.Enabled = true;
         InputTimer.Enabled = true;          //Enable background timer

         Update_LED();                    //Initialize the LED based on current GUI configuration
```

```
            Update_PWMDutyCycle();           //Initialize the PWM based on current GUI configuration

        }

        else
        {
            Status.Text = "Disconnected";
            Status.ForeColor = Color.Red;
        }
    }


    /**********************************************************************
     * NAME: Update_PWMDutyCycle

     *

     * DESCRIPTION: Function used to update the state of the PWM Duty Cycle

     * on the PSoC device end by reading the value on the PWM Duty Cycle text field

     * (DutyTextBox), applying the change to the Output Data Buffer, and

     * writing the OUT report.

     *

     **********************************************************************/

    public void Update_PWMDutyCycle()
    {
        //Update the Output Buffer for PWM based on selected checkbox setting
        DutyCycle = Convert.ToUInt32(DutyTextBox.Text);
        myHidDevice.Outputs.DataBuf[PWM_DutyCycle_Position] = (byte)DutyCycle;
        myHidDevice.WriteOutput();
    }

    /**********************************************************************
     * NAME: Update_LED

     *

     * DESCRIPTION: Function used to update the state of the LED on the PSoC

     * device end by checking the current status of the LED check box,

     * applying the change to the Output Data Buffer, and writing the OUT report.

     *

     **********************************************************************/

    public void Update_LED()
    {
        //Update the Output Buffer for LED based on selected checkbox setting
        if (LED_State_CheckBox.Checked)
        {
            myHidDevice.Outputs.DataBuf[LED_State_Position] = 0xFF;
        }

        else
        {
            myHidDevice.Outputs.DataBuf[LED_State_Position] = 0x00;
        }

        myHidDevice.WriteOutput();
    }

    /**********************************************************************
```

```
         * NAME: InputTimer_Tick

         *

         * DESCRIPTION: Function called by Timer (InputTimer) which is used to poll

         * for input data every 10ms. Function will check contents of Input Data

         * and change display of switch status and update the ADC Value field.

         *

         ***********************************************************************/

        private void InputTimer_Tick(object sender, EventArgs e)
        {
            if (myHidDevice != null)
            {
               // Disable timer so we don't get another interrupt until we service this interrupt
                InputTimer.Enabled = false;

               // This CyUSB.DLL method uses the Win32 ReadFile() function to read IN data transferred
               to our application from the device
                myHidDevice.ReadInput();

               // Check to see if Push Button is pressed. Update GUI based on results
                if (myHidDevice.Inputs.DataBuf[Switch_Status_Position] != 0x00)
                {
                    SwStatus.BackColor = Color.Lime;
                    SwStatus.Text = "ON";
                }
                else
                {
                    SwStatus.BackColor = Color.Red;
                    SwStatus.Text = "OFF";
                }

                //Unload the ADC Data from the Input Buffer to application variables
                AdcCounts_1 = myHidDevice.Inputs.DataBuf[ADC_Byte1_Position];
                AdcCounts_2 = myHidDevice.Inputs.DataBuf[ADC_Byte2_Position];
                AdcCounts_3 = myHidDevice.Inputs.DataBuf[ADC_Byte3_Position];
                AdcCounts_4 = myHidDevice.Inputs.DataBuf[ADC_Byte4_Position];

                //Compose the overall ADC value by shifting and adding the multiple ADC bytes
                String AdcValue = ((AdcCounts_1 << 24) + (AdcCounts_2 << 16) + (AdcCounts_3 << 8) +
AdcCounts_4).ToString();
                AdcValueBox.Text = AdcValue;

                //Re-enable the timer
                InputTimer.Enabled = true;
            }
        }

        /***********************************************************************
         * NAME: Update_PWM_Click

         *

         * DESCRIPTION: When "Update" button is pressed to update PWM value, function

         * checks to see if a matching USB device is currently connected. If so, function

         * calls another function to update the PWM Duty Cycle on the device.

         *

         ***********************************************************************/

        private void Update_PWM_Click(object sender, EventArgs e)
```

```
{
    //Respond to user pressing "Update" button. Make sure device is connected before hand.
    If (myHidDevice != null)
    {
        Update_PWMDutyCycle();
    }
}


/***********************************************************************
 * NAME: Set_VidPid_Click

 *

 * DESCRIPTION: Updates the applications Vendor ID and Product ID based on

 * user input when the "Set" button is clicked. This will cause the default VID

 * and PID of 0x04B4 and 0xE177 to be overwritten. The function will then

 * call GetDevice() to check for matching USB device.

 *

 ***********************************************************************/

private void Set_VidPid_Click(object sender, EventArgs e)
{
    //Respond to update of VID and PID value by pressing the "Set" button
    VID = Convert.ToInt32(VidTextBox.Text, 16);
    PID = Convert.ToInt32(PidTextBox.Text, 16);
    GetDevice();
}


/***********************************************************************
 * NAME: LED_State_CheckBox_CheckedChanged

 *

 * DESCRIPTION: When state of the LED State check box is changed, function

 * checks to see if a matching USB device is currently connected. If so, function

 * calls another function to update the LED state on the device.

 *

 ***********************************************************************/

private void LED_State_CheckBox_CheckedChanged(object sender, EventArgs e)
{
    //Respond to change in LED checkbox state. Make sure device is connected before hand.
    If (myHidDevice != null)
    {
        Update_LED();
    }
}

private void VidTextBox_MaskInputRejected(object sender, MaskInputRejectedEventArgs e)
{

}

private void AdcValueBox_TextChanged(object sender, EventArgs e)
{

}
```

```
        }
}
```

Appendix C. *AppDelegate.c* for Mac OS X

```
#import “AppDelegate.h”

#include <stdio.h>

#include <CoreFoundation/CoreFoundation.h>

       #include <IOKit/hid/IOHIDLib.h>

       /*****************************************************/
       /* Function Prototype */
       /****************************************************/

       @interface AppDelegate (private)
       - (void) initHID;
       - (void) termHID;
       - (void) idleTimer: (NSTimer *) inTimer;

       - (void) deviceMatchingResult: (IOReturn) inResult sender: (void *) inSender device:
       (IOHIDDeviceRef) inIOHIDDeviceRef;
       - (void) deviceRemovalResult: (IOReturn) inResult sender: (void *) inSender device:
       (IOHIDDeviceRef) inIOHIDDeviceRef;
       @end

       static void MyInputCallback (void *context, IOReturn result, void *sender,
       IOHIDReportType type, uint32_t reportID, uint8_t *report, CFIndex reportLength);

       static void Handle_DeviceMatchingCallback(void *inContext, IOReturn inResult, void
       *inSender, IOHIDDeviceRef inIOHIDDeviceRef);

       static void Handle_DeviceRemovalCallback(void *inContext, IOReturn inResult, void
       *inSender, IOHIDDeviceRef inIOHIDDeviceRef);

       @implementation AppDelegate

       @synthesize window;
       /*****************************************************/
       /* Variables for Application */
       /****************************************************/
uint8 AdcCounts_1;

uint8 AdcCounts_2;

uint8 AdcCounts_3;

       uint8 AdcCounts_4;

       double AdcValue;

uint8 SwStatus;

uint8 LedStatus = 0x01;

       double PWM_Duty = 10;

unsigned long long VendorID = 0x04B4;

       unsigned long long ProductID = 0xE177;
       size_t bufferSize = 8;
char *inputBuffer;

       char *outputBuffer;

       IOHIDDeviceRef deviceRef;
       IOHIDManagerRef tIOHIDManagerRef;
       NSMutableDictionary *myDictionary;

       uint8 FirstTimeConfig = 0x00;

       /***************************************************/
       /* Functions */
       /**************************************************/
       - (void)dealloc
       {
           [super dealloc];
       }

       - (void)applicationDidFinishLaunching☹NSNotification *)aNotification
       {
           // Insert code here to initialize your application
           [self initHID];
       }

       //Clean up application before closing
       -(void)applicationWillTerminate☹NSNotification *)notification
       {
           IOHIDManagerRegisterDeviceMatchingCallback(tIOHIDManagerRef, NULL, NULL);
           IOHIDManagerRegisterDeviceRemovalCallback(tIOHIDManagerRef, NULL, NULL);
           IOHIDManagerUnscheduleFromRunLoop(tIOHIDManagerRef, CFRunLoopGetCurrent(),
       kCFRunLoopDefaultMode);

           if (myDictionary)
           {
              CFRelease(myDictionary);
              myDictionary = NULL;
           }

           [timer release];
       }

       //Timer Function. Executed each timer tick
       - (void) idleTimer: (NSTimer *) inTimer
       {

           //Update the GUI with INPUT data
           if(SwStatus != 0x00)
           {
               [SwStatusField setIntValue:1];
           }
           else
           {
               [SwStatusField setIntValue:0];
           }
```

```
    [ADCValueField setDoubleValue:AdcValue];

}


/***************************************************/
/* Function to Initialize the HID */
/**************************************************/

-(void) initHID
{

    IOReturn tIOReturn;


    //Only run this section if it is the first time the application has been opened
    if(FirstTimeConfig == 0x00)
    {
        /************* Setup Manager and schedule it with main run loop *********/
        //Create the HID Manager reference
        tIOHIDManagerRef = IOHIDManagerCreate(kCFAllocatorDefault,
kIOHIDOptionsTypeNone);

        //Schedule a HID manager with current run loop.
        IOHIDManagerScheduleWithRunLoop(tIOHIDManagerRef, CFRunLoopGetCurrent(),
kCFRunLoopDefaultMode);

        //Open the HID manager reference
        tIOReturn = IOHIDManagerOpen(tIOHIDManagerRef, kIOHIDOptionsTypeNone);

        /************ Configure Matching Criteria for Device Detection *************/

        //Create a dictionary
        myDictionary = [NSMutableDictionary dictionary];

        //Configure dictionary with matching criteria for Product ID
        [myDictionary setObject:[NSNumber numberWithLong:ProductID]forKey:[NSString
stringWithCString:kIOHIDProductIDKey encoding:NSUTF8StringEncoding]];

        //Configure dictionary with matching criteria for Vendor ID
        [myDictionary setObject:[NSNumber numberWithLong:VendorID]forKey:[NSString
stringWithCString:kIOHIDVendorIDKey encoding:NSUTF8StringEncoding]];

        //Set matching criteria (Vendor ID and Product ID)
        IOHIDManagerSetDeviceMatching(tIOHIDManagerRef, (CFMutableDictionaryRef)
myDictionary);

        //Set flag indicating the completion of initial configuration
        FirstTimeConfig = 0xFF;
    }

    /************* Acquire the Device ****************/

    //Determine the devices attached that meet the matching criteria
    NSSet * allDevices = [((NSSet
*)IOHIDManagerCopyDevices(tIOHIDManagerRef))autorelease];
```

```
    //Create an array list of matching devices
    NSArray *deviceRefs = [allDevices allObjects];

    //Set our device reference/handler to the first device detected that meets the 
matching criteria
    deviceRef = ([deviceRefs count])? (IOHIDDeviceRef)[deviceRefs 
objectAtIndex:0]:nil;


    //If device is now connected
    if(deviceRef != 0x00)
    {
        //Update GUI to reflect connected device
        [StatusLabel setTextColor:[NSColor greenColor]];
        [StatusLabel setStringValue:@"Connected"];

        [PWMDutyField setIntValue:PWM_Duty];

        //Allocate RAM for Input and Output buffer
        inputBuffer = malloc(bufferSize);
        outputBuffer = malloc(bufferSize);

        //Register callback routine for when an Input report is received
        IOHIDDeviceRegisterInputReportCallback(deviceRef, (uint8_t *)inputBuffer, 
bufferSize, MyInputCallback, NULL);

        //Load initial Output data
        outputBuffer[0] = LedStatus;
        outputBuffer[1] = PWM_Duty;

        //Perform an Output transaction to the PSoC device
        IOHIDDeviceSetReport(deviceRef, kIOHIDReportTypeOutput, 0, 
(uint8_t*)outputBuffer, bufferSize);

    }

    else
    {
        //Update GUI to reflect a device disconnect
        [StatusLabel setTextColor:[NSColor redColor]];
        [StatusLabel setStringValue:@"Disconnected"];
    }

    //Start Timer
    timer = [NSTimer scheduledTimerWithTimeInterval:0.01
                                        target:self
                                      selector:@selector(idleTimer☺
                                      userInfo:nil
                                       repeats:YES];

    //Register callback routines for when a device is attached or removed from Mac
    IOHIDManagerRegisterDeviceMatchingCallback(tIOHIDManagerRef, 
Handle_DeviceMatchingCallback, self);
    IOHIDManagerRegisterDeviceRemovalCallback(tIOHIDManagerRef, 
Handle_DeviceRemovalCallback, self);
}
```

```
- (void) termHID
{
    //Check to make sure a device was connected
    if (tIOHIDManagerRef)
    {
        //Update GUI to reflect disconnect
        [StatusLabel setTextColor:[NSColor redColor]];
        [StatusLabel setStringValue:@"Disconnected"];
        //Disable Timer
        [timer invalidate];
        timer = nil;
        //Reset device handler to nill
        deviceRef = nil;
    }
}
/*****************************************************/
/* Function to execute initHID upon device matching */
/******************************************************/

- (void) deviceMatchingResult: (IOReturn) inResult sender: (void *) inSender device:
(IOHIDDeviceRef) inIOHIDDeviceRef
{

    [self initHID];

}//deviceRemovalResul

/*****************************************************/
/* Function to execute initHID upon device removal */
/*****************************************************/
 - (void) deviceRemovalResult: (IOReturn) inResult sender: (void *) inSender device:
(IOHIDDeviceRef) inIOHIDDeviceRef
{

    [self termHID];

}//deviceRemovalResult

/*********************************************/
/* Function to control LED ON/OFF           */
/* Tiggered by change in radio button state */
/*********************************************/
- (IBAction)LED_Change☹id)sender
{
    //Error catch to make sure device is connected before proceeding
    require(deviceRef, Oops);

    //Update Output buffer based on selected radio button (On or Off)
    switch ([[sender selectedCell] tag])
    {
        case 1:
            LedStatus = 0x01;
            outputBuffer[0] = LedStatus;
            break;

        case 0:
            LedStatus = 0x00;
            outputBuffer[0] = LedStatus;
            break;

        default:
            break;
    }

    //Perform Output transaction
   IOHIDDeviceSetReport(deviceRef, kIOHIDReportTypeOutput, 0, (uint8_t*)outputBuffer,
bufferSize);

//If no matching device is detected, do nothing.
Oops:;

}

/***************************************************/
/* Function to Update device VID and PID Matching */
/* Tiggered by pressing the "Set" button          */
/***************************************************/
-(IBAction)USB_Updateid)sender
{

    //Read the string from the text field and convert to hex value.

    //Update Product ID from text field
    NSString *myPID = [PIDField stringValue];
    NSScanner* myPidScanner = [NSScanner scannerWithString:myPID];
    [myPidScanner scanHexLongLong: &ProductID];

    //Update Vendor ID from text field
    NSString *myVID = [VIDField stringValue];
    NSScanner* myVidScanner = [NSScanner scannerWithString:myVID];
    [myVidScanner scanHexLongLong: &VendorID];


    //Create a new matching 74ons tint74 for the new VID and PID values
    myDictionary = [NSMutableDictionary dictionary];

    [myDictionary setObject:[NSNumber numberWithLong:ProductID]forKey:[NSString
stringWithCString:kIOHIDProductIDKey encoding:NSUTF8StringEncoding]];

    [myDictionary setObject:[NSNumber numberWithLong:VendorID]forKey:[NSString
stringWithCString:kIOHIDVendorIDKey encoding:NSUTF8StringEncoding]];

    //Set single matching criteria
    IOHIDManagerSetDeviceMatching(tIOHIDManagerRef, (CFMutableDictionaryRef)
myDictionary);

    //Initalize new USB device
    [self initHID];



}
```

```
/*******************************************/
/* Function to update PWM duty cycle       */
/* Triggered by pressing "Update" button   */
/*******************************************/
-(IBAction)PWM_Updateid)sender
{
    //Error check to make sure device is connected before proceeding
    require(deviceRef, Oops);

    //Convert String in textfield to Double
    PWM_Duty = [PWMDutyField  doubleValue];


    //Check to see if an out of range value receiving.
    //If so, change to closest acceptable value.
    If(PWM_Duty > 100)
    {
        PWM_Duty = 100;
        [PWMDutyField setIntValue:PWM_Duty];
    }

    if(PWM_Duty < 1)
    {
        PWM_Duty = 1;
        [PWMDutyField setIntValue:PWM_Duty];
    }

    //Update Output Buffer with PWM value
    outputBuffer[1] = PWM_Duty;

    //Perform Output transfer
    IOHIDDeviceSetReport(deviceRef, kIOHIDReportTypeOutput, 0, (uint8_t*)outputBuffer,
bufferSize);

//If no matching device is detected, do nothing.
Oops:;
}


@end

/***************************************************/
/* Call Back Functions */
/**************************************************/

/*******************************************/
/* Function to handle Input Reports       */
/* Triggered by receiving an Input Report */
/*******************************************/
static void MyInputCallback (void *context, IOReturn result, void *sender,
IOHIDReportType type, uint32_t reportID, uint8_t *report, CFIndex reportLength)
{
    //Unload the inputBuffer into specific variables
    SwStatus = inputBuffer[0];
    AdcCounts_1 = inputBuffer[1];
    AdcCounts_2 = inputBuffer[2];
    AdcCounts_3 = inputBuffer[3];
    AdcCounts_4 = inputBuffer[4];

    //Reconstruct the ADC value from 4 8-bit values
    AdcValue = (AdcCounts_1 << 24) + (AdcCounts_2 << 16) + (AdcCounts_3 << 8) +
AdcCounts_4;
}

//Handle for the Device Matching Callback
static void Handle_DeviceMatchingCallback(void *inContext, IOReturn inResult, void
*inSender, IOHIDDeviceRef inIOHIDDeviceRef)
{
    /* Call the Class Method */
    [(AppDelegate *) inContext deviceMatchingResult:inResult sender:inSender
device:inIOHIDDeviceRef];
}   // Handle_DeviceMatchingCallback


//Handle for the Device Removed Callback
static void Handle_DeviceRemovalCallback(void *inContext, IOReturn inResult, void
*inSender, IOHIDDeviceRef inIOHIDDeviceRef)
{
    /* Call the Class Method */
    [(AppDelegate *) inContext deviceRemovalResult:inResult sender:inSender
device:inIOHIDDeviceRef];
}   // Handle_DeviceRemovalCallback
```

Appendix D. *Linux_GenericHID_CLI.c* for Linux

```
/* USB HID App for Linux */
#include <stdio.h>

#include <stdlib.h>

#include <libusb-1.0/libusb.h>

#include <time.h>
#include <unistd.h>


/* Array positioning for IN Data */
#define Button_Status_Pos 0

#define ADC_Pos_1 1

#define ADC_Pos_2 2

#define ADC_Pos_3 3

#define ADC_Pos_4 4

/* Array positioning for OUT Data */
#define LED_State_Pos 0

#define PWM_DC_Pos 1

/* Miscellaneous Result Defines */
#define PASS 0
#define FAIL -1

/* Miscellaneous variables for timing and interface referencing */
static const int INTERFACE_NUMBER = 0;

static const int TIMEOUT_MS = 5000;

static const int NUMBER_INPUT_ITERATIONS = 10;

static const int TIME_DELAY = 1;

/* Initialize and Set Default Vendor ID (VID) and Product ID (PID) */
static const int VID = 0x04B4;

static const int PID = 0xE177;

/* Endpoint Addresses defined in Endpoint Descriptor */
static const int IN_ENDPOINT_ADDRESS = 0x81;

static const int OUT_ENDPOINT_ADDRESS = 0x02;

/* Maximum Input and Output Transfer size */
static const int MAXIMUM_IN_TRANSFER_SIZE = 8;

static const int MAXIMUM_OUT_TRANSFER_SIZE = 8;

/* Function prototype to handle Input and Output transactions */
void  Perform_Input_and_Output_Transfers(libusb_device_handle *MyHIDDevice, int LED_State, int
PWM_DutyCycle);
```

```
int main (void)
{
        struct libusb_device_handle *MyHIDDevice = NULL;
        int Device_Status = 0;
        int Result = 0;

        int LED_State;
        int PWM_DutyCycle;

        /* Initialize LibUSB */
        Result = libusb_init(NULL);

        /* Ensure that the LibUSB initializing was successful */
        if (Result == PASS)
        {
                printf("\n");
                printf("/******* PSoC 3 / PSoC 5LP Generic HID CLI *******/\n");

                printf("Product ID: 0x%.4x", PID);
                printf("\n");
                printf("Vendor ID:  0x%.4x", VID);
                printf("\n");

                /* Finds matching device and assigns device handle */
                MyHIDDevice = libusb_open_device_with_vid_pid(NULL, VID, PID);

                /* Check to see if HID was detected and properly assigned a handle */
                if (MyHIDDevice != NULL)
                {
                    /* Detach the Linux HID driver from the device to allow the use of libusb */
                    Result = libusb_detach_kernel_driver(MyHIDDevice, INTERFACE_NUMBER);

                    if(Result == PASS)
                    {
                        /* Claim USB interface for Application */
                        Result = libusb_claim_interface(MyHIDDevice, INTERFACE_NUMBER);

                        /* Ask user to provide parameters for Output Report */
                        switch(Result)
                        {
                                case LIBUSB_SUCCESS:
                                         Device_Status = 1;
                                         fprintf(stderr, "Device Connected \n");
                                         printf("Enter a Desired State for LED_1 (0 or 1): ");
                                         scanf("%d", &LED_State);

                                         printf("Enter a Desired PWM Duty Cycle for LED_2 (1-100): ");
                                         scanf("%d", &PWM_DutyCycle);
                                         break;

                                case LIBUSB_ERROR_NOT_FOUND:
                                       fprintf(stderr, "The requested interface does not exist. \n");
                                       break;

                                case LIBUSB_ERROR_BUSY:
```

```
                                fprintf(stderr, "Another program or driver has claimed the 
interface. \n");
                                break;

                        case LIBUSB_ERROR_NO_DEVICE:
                                fprintf(stderr, "The device has been disconnected. \n");
                                        break;

                        default:
                                fprintf(stderr, "An Error has occurred. Application cannot 
continue. \n");
                                        break;
                }

            }
        }

        else
        {
            fprintf(stderr, "Unable to locate an attached device that matches.\n");
        }

    }

    else
    {
        fprintf(stderr, "Unable to initialize LibUSB. Please ensure that it is properly 
Installed \n");
    }

    if(Device_Status)
    {
        /* Send and receive data */
        Perform_Input_and_Output_Transfers(MyHIDDevice, LED_State, PWM_DutyCycle);

        /* Finished using the device */
        libusb_release_interface(MyHIDDevice, 0);

        /* Turn control of device back over to kernel HID driver */
        libusb_attach_kernel_driver (MyHIDDevice, 0);
    }

    return 0;
}

void Perform_Input_and_Output_Transfers(libusb_device_handle *MyHIDDevice, int 
LED_State, int PWM_DutyCycle)
{
    /* Declare area in RAM for Input and Output Data Buffer */
    unsigned char In_Data_Buffer[MAXIMUM_IN_TRANSFER_SIZE];
    unsigned char Out_Data_Buffer[MAXIMUM_OUT_TRANSFER_SIZE];

    /* Various variables for application code */
    int ADC_Result = 0;
    int Switch_Status = 0;
    int Transfer_Count;
    int Result = 0;
    int In_Iteration_Count = 0;

    /* Check to ensure LED_State is either '1' or '0'. If not, variable will be
adjusted. */
    if(LED_State > 1)
    {
        LED_State = 1;
    }

    if(LED_State < 0)
    {
        LED_State = 0;
    }

    /* Check to ensure PWM_DutyCycle is between 1 and 100. If not, variable will be
adjusted. */
    if(PWM_DutyCycle > 100)
    {
        PWM_DutyCycle = 100;
    }

    if(PWM_DutyCycle < 1)
    {
        PWM_DutyCycle = 1;
    }


    /* Store output data in output buffer prior to prior to transferring */
    Out_Data_Buffer[LED_State_Pos]= LED_State;
    Out_Data_Buffer[PWM_DC_Pos] = PWM_DutyCycle;

    /* Perform Output Transfer to PSoC */
    Result = libusb_interrupt_transfer(MyHIDDevice,   OUT_ENDPOINT_ADDRESS,
Out_Data_Buffer, MAXIMUM_OUT_TRANSFER_SIZE,
&Transfer_Count, TIMEOUT_MS);

    /* Check to see if Output transaction was successful */
    if (Result == PASS)
    {
        printf("\n");
        printf("/********** Output Data **********/\n");

        if (LED_State == 0)
        {
            printf("Turn LED_1 Off");
            printf("\n");
        }

        else
        {
            printf("Turn LED_1 On");
            printf("\n");
        }

        printf("Set PWM Duty Cycle of LED_2 to %d%%", PWM_DutyCycle);
        printf("\n");
        printf("\n");
    }

    else
    {
        printf("Error Has Occurred with Output Transfer \n");
        printf("\n");
    }

    /* Request IN report from PSoC. Performs in a loop. Number of cycles dependent on
NUMBER_INPUT_ITERATIONS */
    printf("/********** Input Data **********/\n");

    for(In_Iteration_Count=0; In_Iteration_Count<NUMBER_INPUT_ITERATIONS;
In_Iteration_Count++)
    {

    Result = libusb_interrupt_transfer(MyHIDDevice,
                    IN_ENDPOINT_ADDRESS,
                    In_Data_Buffer,
                    MAXIMUM_IN_TRANSFER_SIZE,
                   &Transfer_Count,
                    TIMEOUT_MS);

        /* Check to see if Input request was successful */
        if (Result == PASS)
        {
            /* Unload data from Input Buffer and format ADC Data */
        ADC_Result = (In_Data_Buffer[ADC_Pos_1] << 24) + (In_Data_Buffer[ADC_Pos_2] <<
16) + (In_Data_Buffer[ADC_Pos_3] << 8) +
In_Data_Buffer[ADC_Pos_4];

            Switch_Status = In_Data_Buffer[Button_Status_Pos];

            /* Print Input Data in the Terminal */
            printf("ADC Value: %07d \n", ADC_Result);

            if(Switch_Status == 0)
            {
                printf("Switch is Not Pressed \n");
                printf("\n");
            }

            else
            {
                printf("Switch is Pressed \n");
                printf("\n");
            }
        }

        else
        {
            printf("Error Occurred with Input Transfer \n");
            printf("\n");

        }
```

```
            /* Time is seconds to wait between Input requests */
            sleep(TIME_DELAY);
        }
    }
```

Document History

Document Title: AN82072 - PSoC® 3 and PSoC 5LP USB General Data Transfer with Standard HID Drivers

Document Number: 001-82072

Revision	ECN	Orig. of Change	Submission Date	Description of Change
**	3749838	RLRM	09/21/2012	New Spec.
*A	3819496	RLRM	11/22/2012	Updated for PSoC 5LP.
*B	4497191	KLMZ	09/09/2014	Updated title to include in PSoC USB HID AN Family, Updated screenshots for Creator 3.0
*C	5067967	RLRM	01/06/2016	Updated Linux instructions, fixed typo in OS X application. Updated to new template.
*D	5220391	RLRM	04/20/2016	Updated hyperlinks across the document. Updated Related Resources: Updated Application Notes: Removed reference of AN52970. Updated to new template.
*E	5445335	RLRM	09/28/2016	Updated PSoC Creator projects Updated images in document Fixed typo in Linux commands Updated template
*F	5726423	BENV	05/04/2017	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

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

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