HUMAN INTERFACE DEVICE TUTORIAL

Relevant Devices

This application note applies to all Silicon Labs USB MCUs.

1. Introduction

The Human Interface Device (HID) class specification allows designers to create USB-based devices and applications without the need for custom driver development. Their high levels of on-chip integration and robust USB interfaces make Silicon Laboratories microcontrollers ideal devices for HID designs.

1.1. About this Document

This application note assumes that the reader has a basic understanding of the USB specification, including some knowledge of endpoints, descriptors, and transfer protocols. This document concentrates on highlighting the benefits of designing with HID and techniques for creating and optimizing HID-based systems that use Silicon Laboratories microcontrollers.

This document includes the following:

- Discussion of HIDs
- Firmware template that can be used as a starting point for HID device firmware
- Description of the HID DLL that can be used by host applications to communicate with HID devices
- Explanation of three example HID firmware systems implemented by modifying the template firmware
- Firmware source for each HID example discussed in the text
- Example code for host-side application software

1.2. HID Examples

Both HID examples in this application note were created using the included firmware template as a starting point. The sections titled "7. USB Mouse Example" on page 29, "8. HID Blinky Firmware and Software Example" on page 36, and "9. HID to UART Firmware and Software Example" on page 44 describe how the firmware template was modified to create each example. Sections 8 and 9 also provide examples of how the HID class object can be used to communicate with an HID device.

2. Introduction To HID

USB devices communicate with PCs as shown in Figure 1. Creating a USB interface between an embedded system and a PC requires the writing of code for the following software subsystems:

- Embedded device firmware
- Host-side operating system drivers
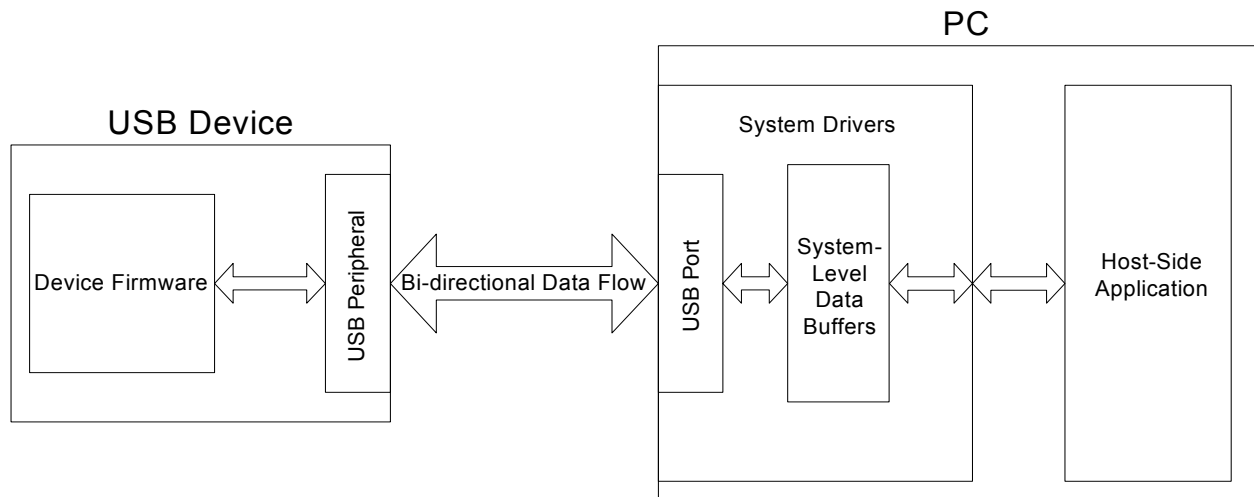- Host-side PC application

Figure 1. USB Interface between a PC and an Embedded System

2.1. USB System Development

The USB specification defines a number of USB classes, such as HID, mass storage devices, etc. Developers creating a USB system that does not fit into one of the predefined USB classes must develop custom drivers along with device firmware and PC applications. For these systems, developers can use a software package, such as Silicon Laboratories' USBXpress, which includes a set of custom drivers, firmware routines, and host routines that enable USB communication. HID-class devices simplify USB communication one step further by using a standardized, flexible driver that comes pre-installed with all commonly-used operating systems.

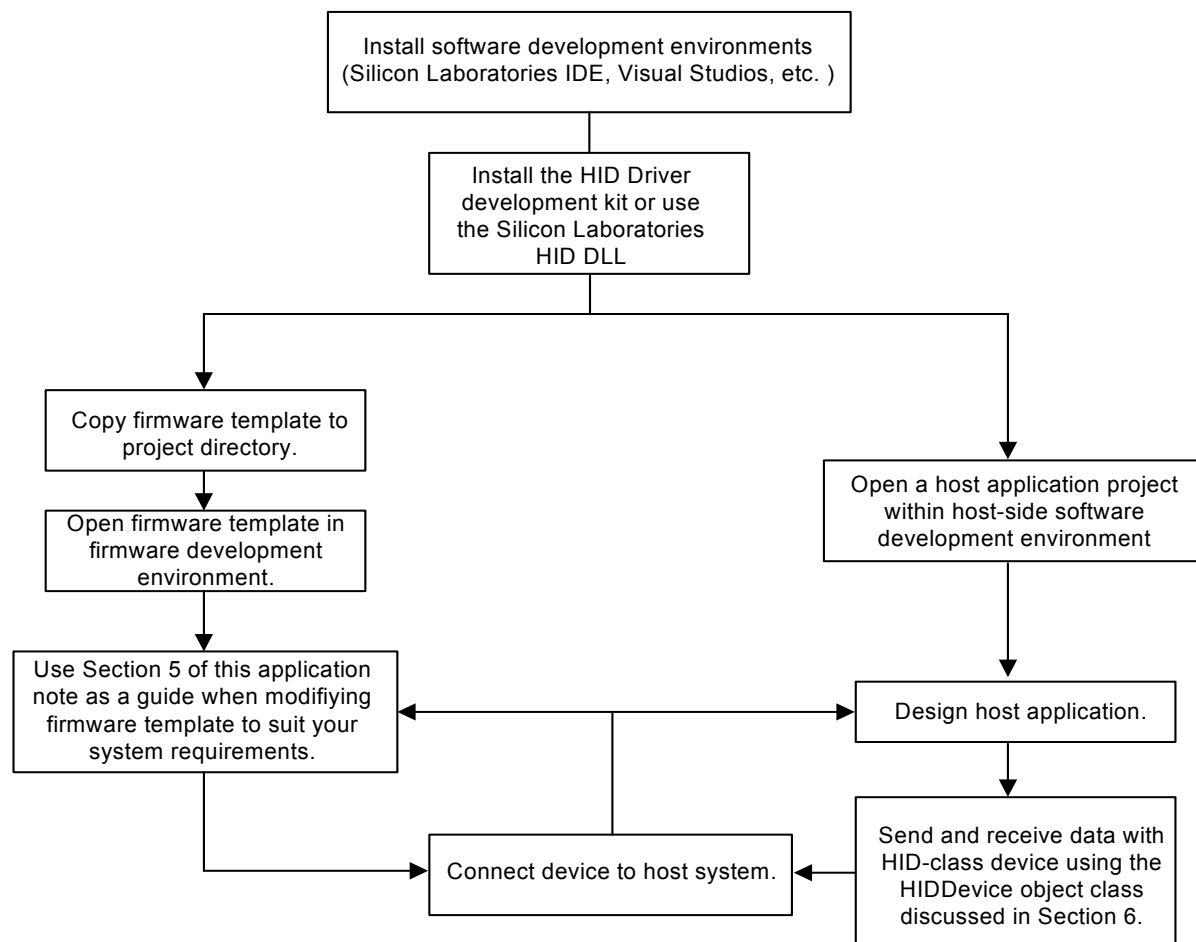Benefits of developing with HID include:

- Compatibility with all commonly used operating systems (XP, 2000, Mac, Linux)
- No need for driver development or distribution
- Streamlined device/host interface due to standardized (but flexible) HID specifications

2.2. Getting Started with HID

Designers can create embedded HID firmware using the Silicon Laboratories IDE, the evaluation version of the Keil compiler, and the target board included in the Silicon Laboratories development kit. Designers developing host-side PC applications must install the Windows Driver Development Kit (DDK), which can be downloaded from http://www.microsoft.com/whdc/devtools/ddk/. Once the DDK is installed, the developer will be able to take advantage of HID-specific API calls to send and receive data with HID-based devices.

Note: The HID DLL included with this application note's software package encapsulates all HID-related Windows DDK functionality into a pre-compiled library. This DLL allows host application programmers to develop code without installing the Windows DDK.

For designers creating an HID-based system using this application note's associated firmware template as a starting point, the typical system design flow is shown in Figure 2.

Install software development environments
(Silicon Laboratories IDE, Visual Studios, etc.)

Install the HID Driver
development kit or use
the Silicon Laboratories
HID DLL

Copy firmware template to
project directory.

Open firmware template in
firmware development
environment.

Use Section 5 of this application
note as a guide when modifiying
firmware template to suit your
system requirements.

Open a host application project
within host-side software
development environment

Design host application.

Connect device to host system.

Send and receive data with
HID-class device using the
HIDDevice object class
discussed in Section 6.

Figure 2. Project Work Flow Chart

SILICON LABS

3. HID Specification Overview

The HID class of the universal serial bus (USB) protocol was created to define devices that interact to some degree with humans and transfer data within a computer system.

3.1. Universal Serial Bus

The USB protocol presents significant advantages over other PC interfaces in versatility, speed, and reliability. USB systems communicate under device/host relationships where a device is attached to a USB port of a host PC or a hub that is then connected to a PC. Host-side application software interacts with device-side firmware through the native operating system or customized drivers.

3.1.1. Device Endpoints

In USB-based systems, all data travels to or from device endpoints. The USB specification requires that all devices have a control endpoint. The host uses this endpoint to retrieve information about the device through data packets called descriptors. Many USB devices also support additional endpoints that transfer data to and from the host. IN endpoints transfer data from the device to the host while OUT endpoints transfer data from the host to the device.

3.1.2. Silicon Laboratories Microcontroller Capabilities

Silicon Laboratories microcontroller families with USB functionality can support a control endpoint and at least one additional endpoint. USB hardware controls low-level data transfer to and from the host. The hardware sends and receives data through user-accessible buffers. The microcontroller signals firmware about USB events, including data reception and transmission-related events, by setting flags. These flags trigger the servicing of an interrupt service routine (ISR) if interrupts have been enabled.

3.1.3. USB Device Classes

The USB specification and supplemental documents define a number of device classes that categorize USB devices according to capability and interface requirements. When a host retrieves device information, class classification helps the host determine how to communicate with the USB device.

3.2. Human Interface Device Class

The HID class devices usually interface with humans in some capacity. HID-class devices include mice, keyboards, printers, etc. However, the HID specification merely defines basic requirements for devices and the protocol for data transfer, and devices do not necessarily depend on any direct human interaction.

3.2.1. Class Requirements

HID devices must meet a few general requirements that are imposed to keep the HID interface standardized and efficient.

- All HID devices must have a control endpoint (Endpoint 0) and an interrupt IN endpoint. Many devices also use an interrupt OUT endpoint. In most cases, HID devices are not allowed to have more than one OUT and one IN endpoint.
- All data transferred must be formatted as reports whose structure is defined in the report descriptor. Reports are discussed in detail later in this document.
- HID devices must respond to standard HID requests in addition to all standard USB requests.

SILICON LABS

4. Enumeration and Device Detection

Before the HID device can enter its normal operating mode and transfer data with the host, the device must properly enumerate. The enumeration process consists of a number of calls made by the host for descriptors stored in the device that describe the device's capabilities.

The device must respond with descriptors that follow a standard format. Descriptors contain all basic information about a device. The USB specification defines some of the descriptors retrieved, and the HID specification defines other required descriptors. The next section discusses the descriptor structure a host expects to receive. The two sections after that describe the responsibilities of the device and the host during enumeration. These sections refer to sections of the HID firmware template, which is discussed in detail later in this document.

4.1. Descriptor Structure

Descriptors begin with a byte describing the descriptor length in bytes. This length equals the total number of bytes in the descriptor including the byte storing the length. The next byte indicates the descriptor type, which allows the host to correctly interpret the rest of the bytes contained in the descriptor. The content and values of the rest of the bytes are specific to the type of descriptor being transmitted. Descriptor structure must follow specifications exactly; the host will ignore received descriptors containing errors in size or value, potentially causing enumeration to fail and prohibiting further communication between the device and the host.

Descriptor contents are typically stored in FLASH memory space. The file named *USB_Descriptor.h* in the HID firmware template declares each value of every descriptor. The file, *USB_Descriptor.c*, defines the contents for each descriptor.

4.1.1. Descriptor Declaration Example

A declaration might look like the following:

```
//-----------------------------------------
// Standard Device Descriptor Type Definition
//-----------------------------------------
typedef struct
{
   BYTE bLength;                  // Size of this Descriptor in Bytes
   BYTE bDescriptorType;          // Descriptor Type (=1)
   WORD bcdUSB;                   // USB Spec Release Number in BCD
   BYTE bDeviceClass;             // Device Class Code
   BYTE bDeviceSubClass;          // Device Subclass Code
   BYTE bDeviceProtocol;          // Device Protocol Code
   BYTE bMaxPacketSize0;          // Maximum Packet Size for EP0
   WORD idVendor;                 // Vendor ID
   WORD idProduct;                // Product ID
   WORD bcdDevice;                // Device Release Number in BCD
   BYTE iManufacturer;            // Index of String Desc for Manufacturer
   BYTE iProduct;                 // Index of String Desc for Product
   BYTE iSerialNumber;            // Index of String Desc for SerNo
   BYTE bNumConfigurations;       // Number of possible Configurations
} device_descriptor;             // End of Device Descriptor Type
```

This declaration exactly conforms to the USB specification's requirements for the size and content order of the device descriptor. Some contents are stored in single bytes while others require two bytes.

4.1.2. Descriptor Definition Example

The definition might look like the following:

```
const code device_descriptor DeviceDesc =
{
    18,                     // bLength
    0x01,                   // bDescriptorType
    0x1001,                 // bcdUSB
    0x00,                   // bDeviceClass
    0x00,                   // bDeviceSubClass
    0x00,                   // bDeviceProtocol
    EP0_PACKET_SIZE,        // bMaxPacketSize0
    0xC410,                 // idVendor
    0x0001,                 // idProduct
    0x0000,                 // bcdDevice
    0x01,                   // iManufacturer
    0x02,                   // iProduct
    0x00,                   // iSerialNumber
    0x01                    // bNumConfigurations
}; //end of DeviceDesc
```

The definition exactly follows the declaration for the `struct device_descriptor`. All contents in this definition must be valid at the firmware system's compile time because all of these values will be stored in nonvolatile memory. Descriptor values stored in multiple bytes must follow the "little endian" style of formatting, where the least significant byte is stored first. For example, a value of 300 or 0x012C, would be stored as 0x2C01.

4.1.3. A Reminder about Descriptors

Many HID devices have very similar descriptor contents, and, in many cases, the descriptors defined in the firmware template need to be changed in only a few places in order to create a customized HID-class device that suits the needs of a given application. For detailed discussions on each descriptor's contents, read the sections of this document describing the firmware template and the two examples created by modifying the template.

4.2. Device Responsibilities during Enumeration

A device's main responsibility during enumeration is to respond to requests for information made by the host system in a timely and accurate manner. The device transfers all enumeration data across the control endpoint. In the firmware template, this endpoint is handled during execution of the USB ISR.

SILICON LABS

4.2.1. The Control Endpoint Handler

The USB ISR examines USB registers to determine the cause of the interrupt. If the ISR finds that an Endpoint 0 transaction caused the interrupt, the ISR calls the control endpoint handler. The Endpoint 0 handler parses the Setup Packet sent by the host and stored in the Endpoint 0 buffer to determine what standard USB request has been made by the host system. The handler then calls the appropriate function. The firmware template file named *F3xx_USB0_Standard_Requests.c* defines all standard requests.

Some of these standard requests require the device to transmit information back to the host. One such standard request, `Get_Descriptor`, allows the host to gather all basic information about the newly-attached device. Other standard requests require the device to receive additional packets of information before the transaction terminates.

4.3. Device Detection after Successful Enumeration

Standard requests sent during enumeration by the host system are not controlled by user-level code. When a device connects with a host's USB port, the host system software will automatically retrieve descriptors and determine whether to enable communication with the device. Host-side application software wishing to interface with the device can then begin communicating with the device using standard API calls.

4.4. Application Communications

Once a device has successfully enumerated, the host can begin sending and receiving data in the form of reports. All data passed between an HID device and a host must be structured according to specifications found in the report descriptor. These reports can be transmitted across either the "Control" pipe (endpoint 0) or the "Interrupt" pipe (endpoints configured to be IN or OUT).

The following sections discuss how to define the report structure inside the report descriptor and how to transfer these reports across either the "Control" pipe or the "Interrupt" pipe.
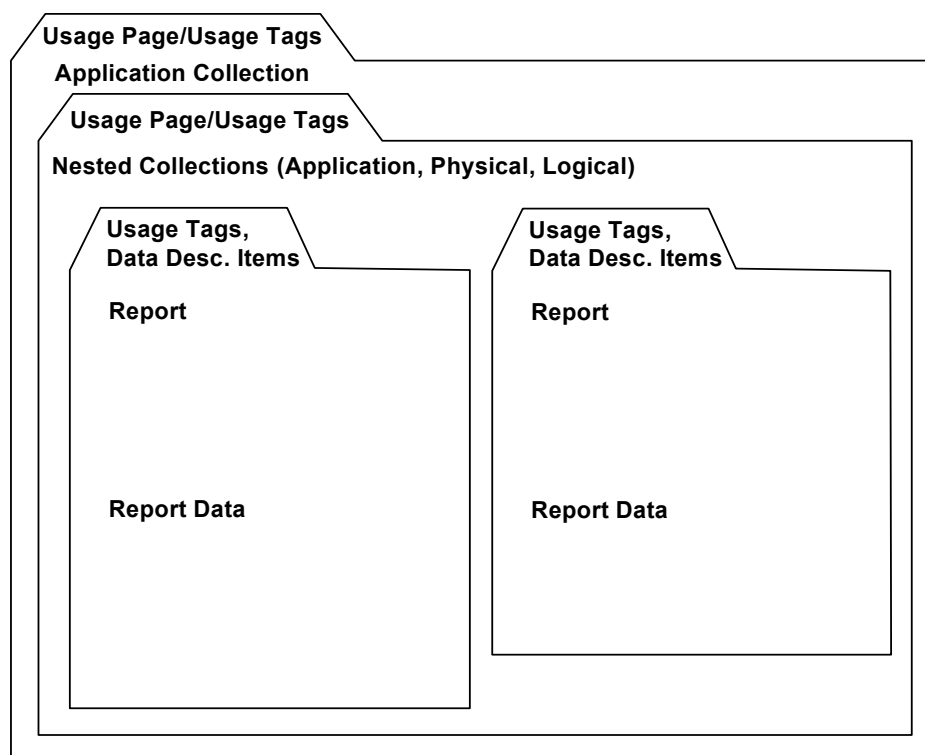

Figure 3. Report Descriptor Example Structure

4.4.1. Report Descriptors

All data transferred to and from an HID device must be structured in the form of reports. The report descriptor defines the report structure, which contains all the information a host needs to determine the data format and how the data should be processed. See Figure 3 for a report descriptor example structure.

4.4.1.1. Report Structure Overview

Although the report structure must follow a few constraints and guidelines, the HID specification purposefully allows for a high degree of customization. This potential for customization gives HID device designers freedom to create a wide variety of HID-class devices.

4.4.1.2. Usage Page and Usage Items

A report descriptor begins with a usage page item that describes the general function of all of the reports that follow. For instance, in a report descriptor describing reports for a USB keyboard or a USB mouse, such as the one found in the USB mouse example in this document, designers would use the "Generic Desktop" usage page.

Reports contained in defined usage pages have defined usages, which provide more specific information about the report contents. For example, a keyboard would use the "Keyboard" usage for its "Generic Desktop" usage page.

For a complete list of defined Usage Pages, check the "HID Usage Tables" document found at USB.org.

4.4.1.3. Collections

"Collections" group similar data. Every report descriptor must have at least one top-level collection in which the data is contained, but the descriptor can define more than one top-level collection. A keyboard with an attached mouse would have two top-level collections, one describing mouse reports and one describing keyboard reports. Each collection must have a usage tag. For example, the "Keyboard" usage can tag a collection of USB keyboard-related data. Also, collections can be nested.

The HID specification defines three types of collections:

- *Application Collections* group variables that carry out a common purpose. All report items must be contained inside an application collection.
- *Logical Collections* group variables of different types that form a composite data structure. Think of logical collections as a collection designator for "struct" data types, such as a struct that groups a buffer with index variables for that buffer.
- *Physical Collections* group data describing a single data point. For instance, a physical collection of data could contain readings from a temperature sensor.

4.4.1.4. Data Types

Report descriptors also contain extensive information describing the characteristics of each data item. Logical Minimum and Logical Maximum items describe the boundary conditions the data contained in reports can reach. The Report Size item describes how many bits each data item uses, and Report Count describes how many data items are contained inside the report. A report of Size 8 and Count 2 would contain 16 bits, or 2 bytes of data.

Data values are further described by designating each data item as Input, Output, or Feature. Input items travel from device to host; Output items travel from host to device, and Feature items can travel in either direction.

Data items can also be designated as Variable, meaning that the values can be read and written, or Constant, meaning that the values are read-only. Another often-used designation indicates whether the value is Absolute, meaning that the value contained in a report is measured from a fixed origin, or Relative, meaning that the value has no fixed reference and instead describes the change of value since the last report.

Systems using more than one defined report structure also need to give each report a unique Report ID tag. This Report ID precedes reports during transfer and signals to the receiver which report is being transmitted. For an example of a USB system that uses more than one report structure, see the HID Blinky Firmware and Software example. For a more detailed discussion about the items in a report descriptor, see the latest revision of the HID specification.

SILICON LABS

4.4.2. Two Transfer Types

Data traffic contained in HID reports can be transferred between device and host through one of two methods. Reports travel across the "Control Pipe" when control endpoint transfers are initiated by host calls to the Windows API functions, `HidD_SetOutputReport()`, `HidD_GetInputReport()`, `HidD_GetFeatureReport()`, and `HidD_SetFeatureReport()`. Reports travel across the "Interrupt Pipe" when data is made available for transfer across endpoints configured as Interrupt IN or Interrupt OUT. The next subsections examine Control Pipe and Interrupt Pipe data transfers.

4.4.2.1. Control Transfers from the Perspective of the Host

The HID specification defines six HID-specific requests. For a complete list of HID-specific requests, see the relevant section in the HID specification.

`HidD_SetOutputReport()` and `HidD_GetInputReport()` allow host applications to send and receive IN and OUT reports. Parameters passed in with the call to `HidD_SetOutputReport()` include the handle to the HID device, the buffer containing the report, and the number of bytes to transmit. Similarly, calls to `HidD_GetInputReport()` require parameters for the handle to the HID device, a buffer where the incoming report will be stored, and the number of packets that the system expects to receive. `HidD_GetFeatureReport()` and `HidDSetFeatureReport()` receive and send Feature reports, which are bidirectional.

The API calls first send a packet to the control endpoint of the device that contains reserved command byte corresponding to `HidD_SetOutputReport()` or `HidD_GetInputReport()`. In the case of the `HidD_SetOutputReport()` command, the system then transmits a second packet containing the report. In the case of the `HidD_GetInputReport()` command, the API transmits a second packet containing the Report ID of the report the application wishes to retrieve. The host then expects the device to ready that packet for transmission to the host, and the host makes an attempt to retrieve that packet.

4.4.2.2. Control Transfers from the Perspective of the Device

After the host initiates a control endpoint transfer, the device's control endpoint handler parses the bytes of this setup packet to determine what request has been transmitted to the device.

If a `HidD_SetOutputReport()` request has been transmitted to the device, the Report ID of the report to be retrieved will be included as part of the setup packet. The device then transmits that report back to the host.

If a `HidD_GetInputReport()` request has been transmitted, the firmware switches the handler into `EP_GetReport` mode. The host then transmits a report to the device. After the report has been transmitted, the microcontroller signals firmware of the availability of the report, and the report can be retrieved from the buffer.

4.4.2.3. Interrupt Transfers from the Perspective of the Host

During enumeration, host system software learns about the interface of the attached USB device. After reception of endpoint descriptors, the system polls any endpoint configured as interrupt IN or interrupt OUT at an interval defined in the endpoint descriptor.

To retrieve IN Endpoint reports transmitted across the interrupt pipe by the device after a poll from the host, the application calls a Windows API function called `Readfile()`. This function requires parameters for the handle of the device, a buffer to store the information, the number of bytes requested, and a variable where the number of bytes successfully retrieved will be stored.

To transmit an OUT Endpoint Report across the Interrupt Pipe, an application calls the Windows API routine named `Writefile()`, and passes into this function parameters including the device handle, a buffer containing the report to be transmitted, the number of bytes to be transmitted, and a variable where the number of bytes successfully transmitted will be stored.

4.4.3. Interrupt Transfers from the Perspective of the Device

Until a device has data to send across the IN endpoint, it should simply NAK the host's polled requests for data by not signaling that data is ready to be received. Once data has been collected into a report structure and placed onto the IN endpoint's buffer, firmware signals that data is ready to transmit.

When the host sends a packet across the OUT endpoint, the microcontroller signals the firmware, and the OUT Endpoint handler retrieves the bytes from the OUT endpoint buffer.

5. HID Firmware Template

The remainder of this document describes the example code from the HID firmware template and the included derivative example systems. This section gives a brief description of the firmware template and general usage techniques.

5.1. Firmware Template Goals

The firmware template takes advantage of the fact that many HID systems differ only in the way they handle data contained in incoming and outgoing reports. By separating standard USB operation, which remains constant across almost all USB-based systems, from custom USB operation, which varies according to the needs of a particular firmware system, the template localizes customized operation to as few points as possible. The user needs to modify only a few sections of code to create a custom HID-class firmware system.

5.2. Template Files

The HID firmware template includes the following files.

- *F3xx_USB0_Main.c*—contains all global variable declarations, the `main()` function, and all routines for peripherals other than USB.
- *F3xx_InterruptServiceRoutine.c*—contains the USB Interrupt Service Routine and handler routines needed to service data on all endpoints.
- *F3xx_USB0_InterruptServiceRoutine.h* —includes prototypes for all USB routines.
- *F3xx_USB0_Register.h*—includes all USB core register addresses, register access macros, and register bit masks.
- *F3xx_USB0_Standard_Requests.c*—contains all standard USB-related functions called by the `USB_ISR` and associated handler routines.
- *F3xx_USB0_Descriptor.c*—where all descriptors are declared.
- *F3xx_USB0_ReportHandler.c*—contains all code for handling input and output packets and the declaration of the Input Vector table and Output Vector table.
- *F3xx_USB0_ReportHandler.h*—includes definition of vector tables.

5.3. Using the Template

Below is a checklist of the locations in the firmware template that must be modified to create a customized HID firmware solution. The modifications on this list are all discussed in detail later in this document.

- In *F3xx_USB0_ReportHandler.c*, modify descriptors as needed.
- In *F3xx_USB0_ReportHandler.c*, add routines to handle all input and output data packets and populate the report handler vector tables.
- In *F3xx_USB0_Main.c*, add any necessary foreground routines for processing USB traffic.
- Add code to control any other peripherals needed to implement the system.

5.4. Default Descriptor Declaration Values

The file, *F3xx_USB0_Main.c*, declares values for each descriptor. Some of these values will remain the same across all HID-class devices, while others will need to be altered depending on the application. The following subsections discuss the values of each descriptor. Items most commonly modified to implement custom devices are highlighted in bold.

SILICON LABS

5.4.1. Device Descriptor

The template defines the device descriptor as follows:

```
{
 18,        // bLength
 0x01,      // bDescriptorType
 0x1001,    // bcdUSB
 0x00,      // bDeviceClass
 0x00,      // bDeviceSubClass
 0x00,      // bDeviceProtocol
 EP0_PACKET_SIZE, // bMaxPacketSize0
 0xC410,    // idVendor
 0x0000,    // idProduct
 0x0000,    // bcdDevice
 0x00,      // iManufacturer
 0x00,      // iProduct
 0x00,      // iSerialNumber
 0x01       // bNumConfigurations
};
```

```
 18,        // bLength
```

The first item describes the descriptor length and should be common to all USB device descriptors.

```
 0x01,                   // bDescriptorType
```

0x01 is the constant one-byte designator for device descriptors and should be common to all device descriptors.

```
 0x1001,                 // bcdUSB
```

This BCD-encoded two-byte item tells the system which USB specification release guidelines the device follows. This number might need to be altered in devices that take advantage of additions or changes included in future revisions of the USB specification, as the host will use this item to help determine what driver to load for the device.

```
 0x00,                   // bDeviceClass
```

If the USB device class is to be defined inside the device descriptor, this item would contain a constant defined in the USB specification. However, this firmware template assumes that the device will be defined in other descriptors. Device classes defined in other descriptors should set the device class item in the device descriptor to 0x00.

```
 0x00,                   // bDeviceSubClass
```

If the device class item discussed above is set to 0x00, then the device subclass item should also be set to 0x00. This item can tell the host information about the device's subclass setting.

```
 0x00,                   // bDeviceProtocol
```

This item can tell the host whether the device supports high-speed transfers. If the above two items are set to 0x00, this one should also be set to 0x00.

```
EP0_PACKET_SIZE,      // bMaxPacketSize0
```

This item tells the host the maximum number of bytes that can be contained inside a single control endpoint transfer. For low-speed devices, this byte must be set to 8, while full-speed devices can have maximum endpoint 0 packet sizes of 8, 16, 32, or 64. `EP0_PACKET_SIZE` is defined in the *F3xx_USB0_Descriptor.h* header file.

Reports can be larger than the maximum packet size. In this case, the report will be transferred across multiple packets.

```
0xC410,   // idVendor
```

This two-byte item identifies the vendor ID for the device. Vendor IDs can be acquired through the USB.org website. Devices using C8051F32x microcontrollers are allowed to use Silicon Laboratories' vendor ID, which is 0xC410, after applying for a Silicon Labs-issued PID.

Host applications will search attached USB devices' vendor IDs to find a particular device needed for an application.

```
0x0000,   // idProduct
```

Like the vendor ID, this two-byte item uniquely identifies the attached USB device. Product IDs can be acquired through the USB.org web site. Alternatively, Silicon Laboratories has reserved a block of product IDs to be used by customers designing products with Silicon Laboratories USB products. Contact Silicon Laboratories technical support to allocate one of these reserved product IDs for your design. This service is free of charge.

```
0x0000,   // bcdDevice
```

This item is used along with the vendor ID and the Product ID to uniquely identify each USB device.

```
0x00,               // iManufacturer
```

The next three one-byte items tell the host which string array index to use when retrieving UNICODE strings describing attached devices that are displayed by the system on-screen. This string describes the manufacturer of the attached device. For example, the string could read "Silicon Laboratories."

A string index value of 0x00 indicates to the host that the device does not have a value for this string stored in memory.

```
0x00,               // iProduct
```

This index will be used when the host wants to retrieve the string that describes the attached product. For example, the string could read "USB Keyboard".

```
0x00,               // iSerialNumber
```

The string pointed to by this index can contain the UNICODE text for the product's serial number.

```
0x01               // bNumConfigurations
```

This item tells the host how many configurations the device supports. A configuration is the definition of the device's functional capabilities, including endpoint configuration. All devices must contain at least one configuration, but more than one can be supported. For this example, only one configuration will be defined.

5.4.2. Configuration Descriptor

After the host retrieves the device descriptor, it can request other descriptors, including the configuration descriptor. The following is the firmware template's configuration descriptor:

```
{
  0x09,      // Length
  0x02,      // Type
  0x0000,    // Totallength
  0x01,      // NumInterfaces
  0x01,      // bConfigurationValue
```

SILICON LABS

```
   0x00,        // iConfiguration
   0x80,        // bmAttributes
   0x00         // MaxPower (in 2 mA units)
}
```

0x09,   // Length

This defines the length of the configuration descriptor. This is a standard length and should be common to all HID devices.

0x02,   // Type

0x02 is the constant one-byte designator for configuration descriptors.

0x2200,   // Totallength

This two-byte item defines the length of this descriptor and all of the other descriptors associated with this configuration. The length of this example is calculated by adding the length of the configuration descriptor, the interface descriptor, the HID descriptor, and one endpoint descriptor. This two-byte item follows a "little endian" data format.

This two-byte item defines the length of this descriptor and all of the other descriptors associated with this configuration. The length of this example is calculated by adding the lengths of the 9-byte configuration descriptor, the 9-byte interface descriptor, the 9-byte HID descriptor, and one 7-byte endpoint descriptor. Note that this two-byte length value follows a "little endian" format, where the value is stored least-significant byte first.

0x01,   // NumInterfaces

This item defines the number of interface settings contained in this configuration.

0x01,   // bConfigurationValue

This item gives this particular configuration a designation of 0x01, which can be used in the standard USB requests `Get_Configuration` and `Set_Configuration` to identify this configuration. This number must be higher than 0.

0x00,   // iConfiguration

This item defines the string index for a string that describes this configuration. This example defines no configuration string and sets the index to 0x00 to indicate this condition to the host.

0x80,   // bmAttributes

This item tells the host whether the device supports USB features such as remote wake-up. Item bits are set or cleared to describe these conditions. Check the USB specification for a detailed discussion on this item.

0x00   // MaxPower (in 2 mA units)

This item tells the host how much current the device will require to function properly at this configuration.

5.4.3. Interface Descriptor

The firmware template defines the interface descriptor as follows:

```
{
   0x09,    // bLength
   0x04,    // bDescriptorType
   0x00,    // bInterfaceNumber
   0x00,    // bAlternateSetting
   0x01,    // bNumEndpoints
```

```
   0x03,      // bInterfaceClass (3 = HID)
   0x00,      // bInterfaceSubClass
   0x00,      // bInterfaceProcotol
   0x00       // iInterface
},
```

0x09,   // bLength

This item describes the size of the interface descriptor and is common to all devices' interface descriptors.

0x04,   // bDescriptorType

0x04 is the constant one byte designator for Interface descriptors.

0x00,   // bInterfaceNumber

This item distinguishes between interfaces of a configuration. Composite devices, such as a keyboard with an embedded mouse, have more than one active interface. Each interface must be distinguished using this designation item. The firmware template only defines a single interface, so the Interface number can be set to 0x00.

0x00,   // bAlternateSetting

This item is useful for devices that define multiple interfaces. This firmware template assumes that only one primary interface will be defined for the device, and sets this item to 0x00, which tells the host that the device defines no alternate setting.

0x01,   // bNumEndpoints

This item tells the host how many endpoints, not counting the control endpoint, will be active in this configuration. Remember that the HID specification requires at least one IN interrupt endpoint be defined in every device.

0x03,   // bInterfaceClass

This item is used to define the device's subclass. A value of 0x03 designates this device's class as HID.

0x00,   // bInterfaceSubClass

This item further describes the device by defining which subclass of the above-defined class the device falls under. For many HID devices, this item will be set to set to 0x00. See the HID Specification for a list of defined HID subclasses.

0x00,   // bInterfaceProcotol

This item can be used to define protocol settings for a USB device. For many HID devices, this item will be set to set to 0x00. See the HID Specification for a list of defined HID protocols.

0x00   // iInterface

This item tells the host the string index of the interface string that describes the specifics of the interface and can be displayed on-screen. The firmware template defines no such string and so sets the index to 0x00.

5.4.4. IN Endpoint Descriptor

The firmware template defines the IN endpoint descriptor as follows:

```
{
   0x07,                 // bLength
   0x05,                 // bDescriptorType
   0x81,                 // bEndpointAddress
```

```
    0x03,                    // bmAttributes
    EP1_PACKET_SIZE_LE,  // MaxPacketSize (LITTLE ENDIAN)
    10        // bInterval
}
```

0x07,   // bLength

This item defines the length of this endpoint descriptor.

0x05,   // bDescriptorType

0x05 is the constant one-byte designator for Endpoint descriptors.

0x81,   // bEndpointAddress

This item describes the address and data flow direction of the endpoint. Bits 0 through 3 define the endpoint's address, and bit 7 describes the data flow direction, with 1 meaning IN and 0 meaning OUT. The item in this descriptor defines an "IN" endpoint with Endpoint Address 1.

0x03,   // bmAttributes

This item describes the type of data transfer the device is configured to use. "0x03" designates this endpoint as using the Interrupt data transfer method.

EP1_PACKET_SIZE_LE, // MaxPacketSize (LITTLE ENDIAN)

This item tells the host the maximum packet size for the endpoint. The maximum packet size should be at least as large as the largest Report. `EP1_PACKET_SIZE_LE` is defined in the *F3xx_USB0_Descriptor.h* header file.

10 // bInterval

The value contained in this item determines how often the endpoint will be polled for data by the system software. Units of the value vary depending on the speed of the device.

5.4.5. OUT Endpoint Descriptor

HID-class devices are not required to use an OUT endpoint. The firmware template declares an OUT endpoint in case custom systems require one. Most item values in this descriptor will be identical to the IN endpoint descriptor. The descriptor is defined as follows:

```
{
    0x07,                    // bLength
    0x05,                    // bDescriptorType
    0x02,                    // bEndpointAddress
    0x03,                    // bmAttributes
    EP2_PACKET_SIZE_LE,    // MaxPacketSize (LITTLE ENDIAN)
    10                      // bInterval
}
```

0x07,   // bLength

This item describes the size of the descriptor.

0x05,   // bDescriptorType

0x05 is the constant one-byte designator for endpoint descriptors.

0x02,   // bEndpointAddress

This item configures the endpoint to be OUT at address 02.

0x03,   // bmAttributes

This item describes the type of data transfer the device is configured to use. "0x03" designates this endpoint as using the Interrupt data transfer method.

EP2_PACKET_SIZE_LE,   // MaxPacketSize (LITTLE ENDIAN)

This item tells the host the maximum packet size for the endpoint. The maximum packet size should be at least as large as the largest Report. EP2_PACKET_SIZE_LE is defined in the *F3xx_USB0_Descriptor.h* header file.

10   // bInterval

The value contained in this item determines how often the endpoint will be polled for data by the system software. Units of the value vary depending on the speed of the device. For the firmware template, which transfers data full speed rates, the units for bInterval are 125 ms. This descriptor defines a polling speed of 125 ms x 10 or once every 1.25 seconds.

5.4.6. HID Descriptor

This class-specific descriptor gives the host information specific to the device at the above-defined configuration.

The descriptor looks like this:

```
{ // class_descriptor hid_descriptor
 0x09,      // bLength
 0x21,      // bDescriptorType
 0x0101,    // bcdHID
 0x00,      // bCountryCode
 0x01,      // bNumDescriptors
 0x22,      // bDescriptorType
 HID_REPORT_DESCRIPTOR_SIZE_LE // wItemLength (total length of report descriptor)
}
```

0x09,   // bLength

This length describes the size of the HID descriptor. It can vary depending on the number of subordinate descriptors, such as report descriptors, that are included in this HID configuration definition.

0x21,   // bDescriptorType

0x21 is the constant one-byte designator for device descriptors and should be common to all HID descriptors.

0x0101,   // bcdHID

This two-byte item tells the host which version of the HID class specification the device follows. USB specification requires that this value be formatted as a binary coded decimal digit, meaning that the upper and lower nibbles of each byte represent the number '0'...9'. In this case, 0x0101 represents the number 0101, which equals a revision number of 1.01 with an implied decimal point.

0x00,   // bCountryCode

If the device was designed to be localized to a specific country, this item tells the host which country. Setting the item to 0x00 tells the host that the device was not designed to be localized to any country.

SILICON LABS

```
0x01,      // bNumDescriptors
```

This item tells the host how many report descriptors are contained in this HID configuration. The following two-byte pairs of items describe each contained report descriptor. The firmware template is configured to contain a single report descriptor, which can define multiple reports.

```
0x22,      // bDescriptorType
```

This item describes the first descriptor which will follow the transfer of this HID descriptor. "0x22" indicates that the descriptor to follow is a report descriptor.

```
HID_REPORT_DESCRIPTOR_SIZE_LE  // wItemLength  (total  length  of  report
descriptor)
```

This item tells the host the size of the descriptor that is described in the preceding item. The value for `HID_REPORT_DESCRIPTOR_SIZE_LE` can be set in the *F3xx_USB0_Descriptor.h* header file.

If the HID descriptor contains more than one subordinate descriptor, those descriptors would be defined at this point, in two-byte pairs like the report descriptor declared above.

5.4.7. String Descriptors

The USB device stores character strings defining the product, the manufacturer, the serial number, and other descriptive texts. A string descriptor table stores the memory addresses of these strings. The host retrieves strings through a standard request call that passes the index of the requested string.

The firmware template defines the following string description table:

```
BYTE* const StringDescTable[] =
{
   String0Desc,
   String1Desc,
   String2Desc
};
```

```
The first String Descriptor of the String Descriptor Table, shown above to be defined
as String0Desc, contains special information about the strings contained within the
table. The Firmware Template defines String0Desc as follows:
```

```
code const BYTE String0Desc[STR0LEN] =
{
   STR0LEN, 0x03, 0x09, 0x04
}; //end of String0Desc
```

The first byte of this descriptor defines the descriptor's length, and the second byte defines this array as a string descriptor. The next two bytes form the language ID code, which the host can use to determine how to interpret characters in the other strings of the string descriptor table. "0x09" indicates that the strings use the English language, and the "0x04" subcode indicates that the English language type is U.S. English.

Each element that follows the first special descriptor holds an address to a string. The first byte of each string descriptor defines the length of the string descriptor, and the second byte tags the array as a string descriptor. After the first two bytes, all remaining bytes are two-byte unicode-formatted string characters. For most strings, the unicode values will store the ANSI character in the low byte and a "0" in the high byte. Following the USB's little endian format requirement, each character will appear in the string descriptor with the ANSI character followed by a 0. For example, the firmware template would define the string "Silicon Laboratories" as follows:

```
code const BYTE String1Desc[STR1LEN] =
{
   STR1LEN, 0x03,
   'S', 0,
   'I', 0,
   'L', 0,
   'I', 0,
   'C', 0,
   'O', 0,
   'N', 0,
   ' ', 0,
   'L', 0,
   'A', 0,
   'B', 0,
   'O', 0,
   'R', 0,
   'A', 0,
   'T', 0,
   'O', 0,
   'R', 0,
   'I', 0,
   'E', 0,
   'S', 0
}; //end of String1Desc
```

The index values to USB-specification standard descriptive texts, such as the product string, the manufacturer string, and the serial string, are defined in the device descriptor. For instance, the firmware template defines the product string index as "0x02" in the device descriptor. At any point after the host retrieves the device descriptor, the host can retrieve the product string by making a standard Get Descriptor request for a string descriptor of index "0x02". When the firmware template receives this request, it returns the value of the string held in memory addressed in indexed element "0x02" of the string descriptor table.

5.4.8. Report Descriptor

By default, the report descriptor declaration of the firmware template is left blank. The contents of this descriptor depend on device requirements. The next two sections provide examples of how the report descriptor can look and how reports can be structured.

5.5. F3xx_USB0_ReportHandler.c File

All report preparation and formatting takes place inside report handlers contained within the *USB_ReportHandler.c* file. Locations in the .c file that must be modified by the user are clearly commented. The following subsections describe how the firmware system functions and discuss each location where modifications will be necessary.

5.5.1. Behavior of the F3xx_USB0_ReportHandler.c File

Firmware calls handlers defined inside *F3xx_USB0_ReportHandler.c* every time a newly received output report has been received and every time an input report needs to be formatted before transmission.

The *F3xx_USB0_ReportHandler.c* file defines two report vector tables, one for input reports and one for output reports. Each element of a vector table is composed of a struct linking a report ID to its corresponding custom report handler. Designers are responsible for entering their report IDs and report handler function names into these tables. When a report needs to be prepared or processed, firmware calls either `ReportHandler_IN()` or `ReportHandler_OUT()`, with the report ID passed as a parameter. These functions search the report vector tables for a matching report ID and then call the corresponding report handler.

SILICON LABS

5.5.2. Handler Prototypes And Declarations

Toward the top of the file, designers will find a few sections where custom code must be added. A function prototype for each report handler must be placed in the Local Function Prototype section. Designers should link each of these functions to their corresponding report IDs inside the report vector tables. Input reports should be added to `IN_VectorTable`, while output reports should be added to `OUT_VectorTable`. Designers must also set the vector table array sizes correctly by setting the pre-compiler directives `IN_VectorTableSize` and `OUT_VectorTableSize`.

In the case where the HID system requires only one input or output report, the vector tables must link the report handler to report ID of 0x00. For an example of this, see "7. USB Mouse Example" .

5.5.3. Report Handler Requirements

Designers must define a function body for each report handler, and these handlers must follow a few simple guidelines. Input report handlers must set the `IN_Buffer` field "`Ptr`" to the buffer containing data to be transferred, and must set the `IN_Buffer` field "`length`" to the number of bytes the USB system should attempt to transfer.

Before the firmware system calls the appropriate output report handler, the system calls a function defined in *F3xx_USB0_ReportHandler.c* called `Setup_OUT_Buffer()`. This routine points the `OUT_Buffer` field "`Ptr`" to a buffer where the received report can be stored. The routine must also set `OUT_Buffer's` field "`Length`" to the size of the buffer.

Output report handlers must assume that data contained in `OUT_Buffer` will be overwritten after the handler exits and should copy any data that needs to be preserved elsewhere in memory.

5.5.4. Including Report IDs

If the report descriptor defines report IDs, report handlers must include report IDs in the `IN_Buffer.Ptr` and adjust the `IN_Buffer.Length` accordingly. The first byte of the input buffer should be set to the report ID of the report about to be transmitted. The number of bytes transmitted should be increased by 1 to include the report ID byte. The USB mouse example uses only one report, does not define any report IDs, and so the report handler does not include any report ID information in the buffer. The HID Blinky example does use more than one report; so, all of its report handlers must take report IDs into account. For systems that use more than one report, each report must be tagged with a unique report ID.

6. HID DLL and CHIDDevice Object Classes

The HID DLL includes the definition for the CHIDDevice object class, which can be declared and associated with an HID-class device in a Visual C++ project. The DLL also includes routines that are not declared within the CHIDDevice class that can be useful for finding and communicating with HID devices connected to the system.

These functions are included with the HID DLL to search the system for the desired device:

`GetNumHidDevices()`	– Returns count of connected HID devices
`GetHidString()`	– Returns serial string of connected HID device matching VID and PID
`GetHidGuid()`	– Used to register for windows messages relating to HID devices

Host-side applications can communicate with HID-class devices using the following member functions included with this class:

`Open()`	– Initiates communication with the specified connected device
`IsOpened()`	– Returns whether class instance is connected to device
`GetDeviceSerialString()`	– Returns specified string of connected device
`SetFeatureReport_Control()`	– Sends feature report
`GetFeatureReport_Control()`	– Requests feature report
`SetOutputReport_Interrupt()`	– Sends OUT report using interrupt pipe
`GetInputReport_Interrupt()`	– Receives IN report across interrupt pipe
`SetOutputReport_Control()`	– Sends OUT report across control pipe
`GetInputReport_Control()`	– Requests IN report across control pipe
`GetInputReportBufferLength()`	– Returns size of biggest input report
`GetOutputReportBufferLength()`	– Returns size of biggest output report
`GetFeatureReportBufferLength()`	– Returns size of biggest feature report
`GetMaxReportRequest()`	– Returns maximum number of reports that can be retrieved
`FlushBuffers()`	– Clear all of the object's buffers
`GetTimeouts()`	– Returns read and write time-out values
`SetTimeouts()`	– Sets read and write time-out values

6.1. Using the DLL

To use the CHIDDeviceclass, the library file SLABHIDDevice.lib must be added to the host application's build, and files using the class must include the HIDDevice.h header file. Because all HID functionality is encapsulated inside this pre-compiled library, the Windows DDK is not required to be installed when building projects with the DLL.

6.2. Other Routines

The sections below describe the other routines that are included in the DLL but are not member functions of the CHIDDevice class. These functions should be used to search the system for a specific device to be associated with a CHIDDevice object.

SILICON LABS

6.2.1. GetNumHidDevices()

Description: This function returns the number of devices found on the USB that have PIDs and VIDs matching the function's input parameters. If NULL is supplied for both of these parameters then it will return the entire number of HID devices connected. If a VID and PID are specified here, then they should also be specified in the Open member function of the HID Device class.

Prototype: `GetNumHidDevices(WORD vid, WORD pid);`

Parameters:

1. `vid` - WORD value containing the desired VID for the device in question (or zero for all devices).
2. `pid` - WORD value containing the desired PID for the device in question (or zero for all devices).

Return Value: Number of HID devices attached to the USB

6.2.2. GetHidString()

Description: This routine obtains the serial string of an attached HID device by its index. The strings available are the VID string, PID string, Serial string, and the device path of the indexed device.

Prototype: `BYTE GetHidString(DWORD deviceIndex, WORD vid, WORD pid, BYTE hidStringType, LPSTR deviceString, DWORD deviceStringLength);`

Parameters:

1. `deviceIndex` - zero based index identifier for device; 0 unless multiple devices are attached to the USB interface.
2. `vid` - this is a WORD value containing the desired VID for the device in question (or zero for all devices)
3. `pid` - this is a WORD value containing the desired PID for the device in question (or zero for all devices)
4. `hidStringType` - this can be any of the constants, `HID_VID_STRING`, `HID_PID_STRING`, `HID_SERIAL_STRING`, or `HID_PATH_STRING`
5. `serialString` - pointer where retrieved string will be stored.
6. `serialStringLength` - length of string to be retrieved.

Return Value: Results = HID_DEVICE_NOT_FOUND or HID_DEVICE_SUCCESS

6.2.3. GetHidGuid()

Description: This routine returns the GUID information for the HID class.

Prototype: `VOID GetHidGuid(LPGUID hidGuid)`

Parameters:

1. `hidGuid` - pointer to GUID variable where returned HID GUID value is to be stored.

6.3. CHIDDevice Class Member Functions

The sections below describe each member function of the CHIDDevice class.

6.3.1. Open ()

Description: This routine opens a communication link to an attached HID device identified by the index into the list of connected HID devices. To reference against all of the HID devices connected to the system, supply NULL, or zero, for the VID and PID. To reference against all HID devices with a specific VID and PID supply the specified VID and PID. This routine also adjusts the number of input reports that can be received by the system's HID class driver buffer if desired. By default this parameter can be ignored and will use the operating system default.

Prototype: `BYTE Open(DWORD deviceIndex, WORD vid, WORD pid, WORD numInputBuffers);`

Parameters:

1. `deviceIndex` - zero-based index identifier for device; 0 unless multiple devices are attached to the USB interface.
2. `vid` - WORD value containing the desired VID for the device in question (or zero for all devices)
3. `pid` - WORD value containing the desired PID for the device in question (or zero for all devices)
4. `numInputBuffers` - changes the size of the input report buffer. In most systems, it is not necessary to use this parameter. When the parameter is not used, a 0 is passed automatically meaning "use the operating system default".

Return Value: Results = HID_DEVICE_SUCCESS, HID_DEVICE_ALREADY_OPENED, or HID_DEVICE_NOT_FOUND

6.3.2. GetDeviceSerialString

Description: This routine returns the specified string of the connected device. The strings available are the VID string, PID string, Serial string or the device path of the opened device represented by the object.

Prototype: `GetString(BYTE hidStringType, LPSTR deviceString, DWORD deviceStringLength)`

Parameters:

1. `hidStringType` - this can be any of the constants HID_VID_STRING, HID_PID_STRING, HID_SERIAL_STRING or HID_PATH_STRING
2. `deviceString` - pointer to buffer of characters where retrieved string is to be stored
3. `deviceStringLength` - length of buffer pointed at by deviceString

6.3.3. IsOpened()

Description: This routine is used to determine when a call to `Open()` has successfully opened a communication link with an attached HID device.

Prototype: `BOOL IsOpened()`

Parameters: None.

Return Value: TRUE if communication link is open, or FALSE if no link has been opened.

6.3.4. SetFeatureReport_Control()

Description: This routine sends a feature report across the control pipe using the HID API function `HidD_GetFeatureReport()`.

Prototype: BYTE SetFeatureReport(BYTE* buffer, DWORD bufferSize)

Parameters:

1. `buffer` – pointer to the feature report to be transmitted. This buffer must be able to store at least bufferSize bytes.

SILICON LABS

2. `bufferSize` - size of the feature report in bytes.

Return Value: Results = HID_DEVICE_SUCCESS, HID_DEVICE_TRANSFER_FAILED, or HID_DEVICE_NOT_OPENED

6.3.5. GetFeatureReport_Control()

Description: This routine retrieves a feature report across the control pipe using the HID API function, `HidD_GetFeatureReport()`.

Prototype: `BYTE GetFeatureReport(BYTE* buffer, DWORD bufferSize)`

Parameters:

1. `buffer` - pointer to location where retrieved feature report will be stored. This buffer must be able to store at least bufferSize bytes.

Note: The first byte of this buffer should be the report ID of the report to be retrieved.

2. `bufferSize` - should be set to the size of the largest feature report listed in the device's report descriptor. This information can be obtained by calling the member function `GetFeatureReportBufferLength()`.

Return Value: Results = HID_DEVICE_SUCCESS, HID_DEVICE_TRANSFER_FAILED, or HID_DEVICE_NOT_OPENED

6.3.6. SetOutputReport_Interrupt()

Description: This function sends an OUT report across the interrupt pipe using a call to `WriteFile()`.

Prototype: `BYTE SetReport_Interrupt(BYTE* buffer, DWORD bufferSize)`

Parameters:

1. `buffer` - pointer to the OUT report to be transmitted. This buffer must be able to store at least bufferSize bytes.
2. `bufferSize` - size of the OUT report in bytes.

Return Value: Results = HID_DEVICE_SUCCESS, HID_DEVICE_TRANSFER_TIMEOUT, HID_DEVICE_TRANSFER_FAILED, HID_DEVICE_NOT_OPENED, or HID_DEVICE_INVALID_BUFFER_SIZE

6.3.7. GetInputReport_Interrupt()

Description: This function attempts to retrieve an IN report across the interrupt pipe using a call to `ReadFile()`.

Prototype: `BYTE GetReport_Interrupt (BYTE* buffer, DWORD bufferSize, WORDnumReports, DWORD* bytesReturned)`

Parameters:

1. `buffer` - pointer to the IN report to be transmitted. This buffer must be able to store at least bufferSize bytes.
2. `bufferSize` - should be set to the size of the largest feature report listed in the device's report descriptor. This information can be obtained by calling the member function, `GetInputReportBufferLength()`.

Return Value: Results = HID_DEVICE_SUCCESS, HID_DEVICE_TRANSFER_TIMEOUT, HID_DEVICE_TRANSFER_FAILED, HID_DEVICE_NOT_OPENED, or HID_DEVICE_INVALID_BUFFER_SIZE

6.3.8. SetOutputReport_Control()

Description: This function sends a report across the control pipe using the HID API function, `HidD_SetOutputReport()`.

Prototype: `BYTE SetReport_Control(BYTE* buffer, DWORD bufferSize)`

Parameters:

1. `buffer` - pointer to the OUT report to be transmitted. This buffer must be able to store at least bufferSize bytes.
2. `bufferSize` - size of the OUT report in bytes.

Return Value: Results = HID_DEVICE_SUCCESS, HID_DEVICE_TRANSFER_FAILED, HID_DEVICE_NOT_OPENED, or HID_DEVICE_INVALID_BUFFER_SIZE

6.3.9. GetInputReport_Control()

Description: This function receives a report across the control pipe using the HID API function `HidD_GetInputReport()`.

Prototype: `BYTE GetReport_Control(BYTE* buffer, DWORD bufferSize)`

Parameters:

1. `buffer` - pointer to the OUT report to be transmitted. This buffer must be able to store at least bufferSize bytes.
2. `bufferSize` - size of the OUT report in bytes.

Return Value: Results = HID_DEVICE_SUCCESS, HID_DEVICE_TRANSFER_FAILED, HID_DEVICE_NOT_OPENED, or HID_DEVICE_INVALID_BUFFER_SIZE

6.3.10. GetInputReportBufferLength()

Description: This function returns the size of the largest IN report listed in the Report Descriptor of the opened HID device.

Prototype: `WORD GetInputReportBufferLength()`

Parameters: None.

Return Value: The size of the largest IN report.

6.3.11. GetOutputReportBufferLength()

Description: This function returns the size of the largest OUT report listed in the Report Descriptor of the opened HID device.

Prototype: `WORD GetOutputReportBufferLength()`

Parameters: None.

Return Value: The size of the largest OUT report.

6.3.12. GetFeatureReportBufferLength()

Description: This function returns the size of the largest Feature report listed in the Report Descriptor of the opened HID device.

Prototype: `WORD GetFeatureReportBufferLength()`

Parameters: None.

Return Value: The size of the largest Feature report.

6.3.13. GetMaxReportRequest()

Description: This function returns the maximum allowable reports to request for read during one `GetReport_Control()`.

Prototype: `WORD GetMaxReportRequest()`

Parameters: None.

Return Value: Number of reports that can be retrieved during one call to GetReport_Control().

6.3.14. FlushBuffers()

Description: This routine flushes the input queue of the opened HID device by calling the HID API function `HidD_FlushQueue()`.

Prototype: `BOOL FlushBuffers()`

Parameters: None.

SILICON LABS

Return Value: True if buffers were successfully flushed and false if they were not successfully flushed.

6.3.15. SetTimeouts()

Description: This routine sets read and write timeout values for the CHIDDevice object. The read timeout affects all report reads (`GetFeatureReport_Control()`, `GetInputReport_Control()`, and `GetInputReport_Interrupt()`), and the write timeout affects all report writes (`SetFeatureReport_Control()`, `SetOutputReport_Control()`, and `SetOutputReport_Interrupt()`).

Prototype: `void SetTimeouts(UINT getReportTimeout, UINT setReportTimeout)`

Parameters:

1. `getReportTimeout` - sets timeout value for report reads in milliseconds
2. `setReportTimeout` - sets timeout value for all report writes in milliseconds

6.3.16. GetTimeouts()

Description: This routine retrieves configured read and write timeout values from the CHIDDevice object.

Prototype: `void GetTimeouts(UINT* getReportTimeout, UINT* setReportTimeout);`

Parameters:

1. `getReportTimeout` - pointer to UINT variable where input report timeout is to be stored
2. `setReportTimeout` - pointer to UINT variable where output report timeout is to be stored

6.4. Open()

Description:	This routine opens a communication link to an attached HID device identified by index, VID, and PID. This routine also adjusts the number of input reports that can be received by the system's HID class driver buffer.
Prototype:	`BYTE Open(DWORD deviceIndex, WORD vid, WORD pid, WORD numInputBuffers)`
Parameters:	1. `deviceIndex` - identifier for device, 0 unless multiple devices with matching pid and vid are attached to the USB.
	2. `vid` - device VID used in search.
	3. `pid` - device PID used in search.
	4. `numInputBuffers` - changes the size of the input report buffer. In most systems, it is not necessary to use this parameter.
Return Value:	Results = HID_DEVICE_SUCCESS or HID_DEVICE_ALREADY_OPENED or HID_DEVICE_NOT_FOUND

6.5. IsOpened()

Description:	This routine is used to determine when a call to Open() has successfully opened a communication link with an attached HID device.
Prototype:	`BOOL IsOpened()`
Parameters:	None.
Return Value:	TRUE if communication link is open, or FALSE if no link has been opened.

AN249

6.6. SetFeatureReport()

Description: This routine sends a feature report across the control pipe using the HID API function `HidD_GetFeatureReport()`.

Prototype: `BYTE SetFeatureReport(BYTE* buffer, DWORD bufferSize)`

Parameters: 1. `buffer` - pointer to the feature report to be transmitted. This buffer must be able to store at least bufferSize bytes.

2. `bufferSize` - size of the feature report in bytes.

Return Value: Results = HID_DEVICE_SUCCESS or
HID_DEVICE_TRANSFER_FAILED or
HID_DEVICE_NOT_OPENED

6.7. GetFeatureReport()

Description: This routine retrieves a feature report across the control pipe using the HID API function `HidD_GetFeatureReport()`.

Prototype: `BYTE GetFeatureReport(BYTE* buffer, DWORD bufferSize)`

Parameters: 1. `buffer` - pointer to location where retrieved feature report will be stored. This buffer must be able to store at least *bufferSize* bytes.
IMPORTANT NOTE: The first byte of this buffer should be the report ID of the report to be retrieved.
2. `bufferSize` - should be set to the size of the largest feature report listed in the device's report descriptor. This information can be obtained by calling the member function `GetFeatureReportBufferLength()`.

Return Value: Results = HID_DEVICE_SUCCESS or
HID_DEVICE_TRANSFER_FAILED or
HID_DEVICE_NOT_OPENED

6.8. SetReport_Interrupt()

Description: This function sends an OUT report across the interrupt pipe using a call to `WriteFile()`.

Prototype: `BYTE SetReport_Interrupt(BYTE* buffer, DWORD bufferSize)`

Parameters: 1. `buffer` - pointer to the OUT report to be transmitted. This buffer must be able to store at least bufferSize bytes.
2. `bufferSize` - size of the OUT report in bytes.

Return Value: Results = HID_DEVICE_SUCCESS or
HID_DEVICE_TRANSFER_TIMEOUT or
HID_DEVICE_TRANSFER_FAILED or
HID_DEVICE_NOT_OPENED or
HID_DEVICE_INVALID_BUFFER_SIZE

6.9. GetReport_Interrupt()

Description:	This function attempts to retrieve an IN report across the interrupt pipe using a call to `ReadFile()`.
Prototype:	`BYTE GetReport_Interrupt(BYTE* buffer, DWORD bufferSize, WORD numReports, DWORD* bytesReturned)`
Parameters:	1. `buffer` - pointer to the IN report to be transmitted. This buffer must be able to store at least bufferSize bytes. 2. `bufferSize` - should be set to the size of the largest feature report listed in the device's report descriptor. This information can be obtained by calling the member function `GetInputReportBufferLength()`.
Return Value:	Results = HID_DEVICE_SUCCESS or HID_DEVICE_TRANSFER_TIMEOUT or HID_DEVICE_TRANSFER_FAILED or HID_DEVICE_NOT_OPENED or HID_DEVICE_INVALID_BUFFER_SIZE

6.10. SetReport_Control()

Description:	This function sends a report across the control pipe using the HID API function `HidD_SetOutputReport()`.
Prototype:	BYTE SetReport_Control(BYTE* buffer, DWORD bufferSize)
Parameters:	1. `buffer` - pointer to the OUT report to be transmitted. This buffer must be able to store at least bufferSize bytes. 2. `bufferSize` - size of the OUT report in bytes.
Return Value:	Results = HID_DEVICE_SUCCESS or HID_DEVICE_TRANSFER_FAILED or HID_DEVICE_NOT_OPENED or HID_DEVICE_INVALID_BUFFER_SIZE

6.11. GetReport_Control()

Description:	This function receives a report across the control pipe using the HID API function `HidD_GetInputReport`().
Prototype:	`BYTE GetReport_Control(BYTE* buffer, DWORD bufferSize)`
Parameters:	1. `buffer` - pointer to the OUT report to be transmitted. This buffer must be able to store at least bufferSize bytes. 2. `bufferSize` - size of the OUT report in bytes.
Return Value:	Results = HID_DEVICE_SUCCESS or HID_DEVICE_TRANSFER_FAILED or HID_DEVICE_NOT_OPENED or HID_DEVICE_INVALID_BUFFER_SIZE

6.12. GetInputReportBufferLength()

Description:	This function returns the size of the largest IN report listed in the report descriptor of the opened HID device.
Prototype:	`WORD GetInputReportBufferLength()`
Parameters:	None.
Return Value:	The size of the largest IN report.

6.13. GetOutputReportBufferLength()

Description: This function returns the size of the largest OUT report listed in the report descriptor of the opened HID device.

Prototype: `WORD GetOutputReportBufferLength()`

Parameters: None.

Return Value: The size of the largest OUT report.

6.14. GetFeatureReportBufferLength()

Description: This function returns the size of the largest Feature report listed in the report descriptor of the opened HID device.

Prototype: `WORD GetFeatureReportBufferLength()`

Parameters: None.

Return Value: The size of the largest Feature report.

6.15. GetMaxReportRequest()

Description: This function returns the maximum allowable reports to request for read during one `GetReport_Control()`.

Prototype: `WORD GetMaxReportRequest()`

Parameters: None.

Return Value: Number of reports that can be retrieved during one call to GetReport_Control().

6.16. FlushBuffers()

Description: This routine flushes the input queue of the opened HID device by calling the HID API function `HidD_FlushQueue()`.

Prototype: `BOOL FlushBuffers()`

Parameters: None.

Return Value: True if buffers were successfully flushed and false if they were not successfully flushed.

SILICON LABS

7. USB Mouse Example

This section examines the steps taken to emulate a USB Mouse using the Silicon Laboratories target board. This example touches on all necessary firmware modifications that need to be made for HID-class device designs derived from the firmware template. The USB mouse example needs no host-side application to operate correctly. An example of host-side application software can be found in "8. HID Blinky Firmware and Software Example" on page 36.

7.1. Overview

This example contains two versions of low-level mouse emulation code. Code running on a C8051F320/1 or a C8051F34x microcontroller emulates mouse movement by measuring the potentiometer's position. The axis of moment (whether motion on the potentiometer translates to up-down motion or left-right motion) is controlled by one of the switches found on the target board. Every press of the button switches between X-axis movement and Y-axis movement. The other switch on the target board acts as the left mouse button. Code running on a C8051F326/7 emulates movement by setting mouse vector values in a predefined pattern.

The USB Mouse example was created by taking the following steps:

1. Configure all descriptors so that the host recognizes the attached device as a USB mouse.
2. Initialize Timer 2 and (for the C8051F320/1 and C8051F34x device builds) ADC peripherals.
3. Format captured data into a report structure defined in the report descriptor.
4. Add the report handler to transmit these reports to the host.

7.2. Descriptors

This section describes how each descriptor located in the firmware template file, *USB_Descriptor.c*, has been modified to create a device that appears to the host to be a USB mouse. This subsection describes only items in descriptors that have been modified from their default firmware template values.

7.2.1. Device Descriptor

The device descriptor for this example looks like the following code. Changes from the firmware template are highlighted in bold.

```
{
 18,        // bLength
 0x01,      // bDescriptorType
 0x1001,    // bcdUSB
 0x00,      // bDeviceClass
 0x00,      // bDeviceSubClass
 0x00,      // bDeviceProtocol
 EP0_PACKET_SIZE, // bMaxPacketSize0
 0xC410,    // idVendor
 0x0000,    // idProduct
 0x0000,    // bcdDevice
 0x01,      // iManufacturer
 0x02,      // iProduct
 0x00,      // iSerialNumber
 0x01       // bNumConfigurations
```

```
};                        //end of DeviceDesc
```

```
 0x01,      // iManufacturer
```

This example declares a string describing the hardware manufacturer at string table index 1. This item will allow the host to retrieve that string.

```
 0x02,      // iProduct
```

This example also declares a string describing the product at string table index 2.

7.2.2. Configuration Descriptor

```
 0x09,      // Length

 0x02,      // Type

 0x2200,    // Totallength (= 9+9+9+7)

 0x01,      // NumInterfaces

 0x01,      // bConfigurationValue

 0x00,      // iConfiguration

 0x80,      // bmAttributes

 0x20       // MaxPower (in 2 mA units)
```

```
 0x2200,    // Totallength (= 9+9+9+7)
```

This item was calculated by adding together the lengths of the configuration descriptor (9) and all of the other descriptors associated with this particular configuration, specifically the Interface Descriptor (9), one IN Endpoint descriptor (7), and the HID Descriptor (9).

```
 0x20       // MaxPower (in 2 mA units)
```

This item tells the host that the device will require 64 mA to function properly. This should be sufficient current to supply the microcontroller.

7.2.3. Interface Descriptor

```
{ // endpoint_descriptor hid_endpoint_in_descriptor

 0x09,      // bLength

 0x04,      // bDescriptorType

 0x00,      // bInterfaceNumber

 0x00,      // bAlternateSetting

 0x01,      // bNumEndpoints

 0x03,      // bInterfaceClass (3 = HID)

 0x01,      // bInterfaceSubClass

 0x02,      // bInterfaceProcotol

 0x00       // iInterface

};
```

```
 0x01,      // bInterfaceSubClass
```

This item has been set to the boot interface subclass, which is one of the defined HID subclass types. The boot interface is a standard transfer mode that can be used by a system's BIOS at boot time. This device must tell the firmware that it is compatible with the boot interface subclass in order to be recognized as a USB Mouse.

SILICON LABS

```
0x02,      // bInterfaceProcotol
```

This item chooses the "Mouse" protocol for the boot interface HID subclass. It must be set to 0x02 so that the system will know how to interpret the incoming reports.

7.2.4. IN Endpoint Descriptor

```
// IN endpoint (mandatory for HID)

{ // endpoint_descriptor hid_endpoint_in_descriptor

   0x07,               // bLength

   0x05,               // bDescriptorType

   0x81,               // bEndpointAddress

   0x03,               // bmAttributes

   EP1_PACKET_SIZE_LE,  // MaxPacketSize (LITTLE ENDIAN)

   10                  // bInterval

}
```

EP1_PACKET_SIZE_LE

In the *USB_Descriptor.h* header file, this pre-compiler directive is defined to be "3", which will be big enough to allow transfers of the example's only defined report structure.

7.2.5. HID Descriptor

```
{ // class_descriptor hid_descriptor

0x09,      // bLength

0x21,      // bDescriptorType

0x0101,    // bcdHID

0x00,      // bCountryCode

0x01,      // bNumDescriptors

0x22,      // bDescriptorType

HID_REPORT_DESCRIPTOR_SIZE_LE // wItemLength (total length of report descriptor)

}
```

HID_REPORT_DESCRIPTOR_SIZE_LE // wItemLength (total length of report descriptor)

This pre-compiler directive is defined in the *USB_Descriptor.h* file and reflects the size of the mouse example's report descriptor.

7.2.6. Report Descriptor

The report descriptor contains the definition for a report that includes one bit describing the state of the left mouse button and two bytes describing the relative X- and Y- axis positions of the mouse pointer. This example will require that only one report be defined.

This example's only report groups all data inside an application collection that contains generic desktop information pertaining to a mouse. Inside this application collection is a physical collection pertaining to a pointer that contains all information about a single data point (in this case, information about the mouse). This physical collection will group a byte of data containing the bit of data describing the left mouse button state and seven bits of padding with two bytes of data describing the X- and Y- axis positions of the pointer.

SILICON LABS

The report descriptor for the mouse emulation example looks like the following:

```
{
 0x05, 0x01, // Usage Page (Generic Desktop)
 0x09, 0x02, // Usage (Mouse)
 0xA1, 0x01, // Collection (Application)
 0x09, 0x01, // Usage (Pointer)
 0xA1, 0x00, // Collection (Physical)
 0x05, 0x09, // Usage Page (Buttons)
 0x19, 0x01, // Usage Minimum (01)
 0x29, 0x01, // Usage Maximum (01)
 0x15, 0x00, // Logical Minimum (0)
 0x25, 0x01, // Logical Maximum (1)
 0x95, 0x01, // Report Count (1)
 0x75, 0x01, // Report Size (1)
 0x81, 0x02, // Input (Data, Variable, Absolute)
 0x95, 0x01, // Report Count (1)
 0x75, 0x07, // Report Size (7)
 0x81, 0x01, // Input (Constant) for padding
 0x05, 0x01, // Usage Page (Generic Desktop)
 0x09, 0x30, // Usage (X)
 0x09, 0x31, // Usage (Y)
 0x15, 0x81, // Logical Minimum (-127)
 0x25, 0x7F, // Logical Maximum (127)
 0x75, 0x08, // Report Size (8)
 0x95, 0x02, // Report Count (2)
 0x81, 0x06, // Input (Data, Variable, Relative)
 0xC0,       // End Collection (Physical)
 0xC0        // End Collection (Application)
};
```

0x05, 0x01, // Usage Page (Generic Desktop)

This item tells the host that the controls (in this case, mouse data) contained within this report descriptor are relevant to the desktop. Mouse, keyboard, and joystick controls will all be found on the Generic Desktop usage page.

0x09, 0x02, // Usage (Mouse)

This item tells the host that the following collection contains data pertaining to a mouse.

0xA1, 0x01, // Collection (Application)

This item begins the top-level collection of data that can be used by the host application (in this case, the system software).

SILICON LABS

```
0x09, 0x01, // Usage (Pointer)
```

This item tells the host that the following collection contains data pertaining to the characteristics of a mouse pointer device.

```
0xA1, 0x00, // Collection (Physical)
```

This item begins the collection of data describing a single data point. In this case, the single data point is the state of the left mouse button and the relative position of the mouse along the X- and Y-axis.

```
0x05, 0x09, // Usage Page (Buttons)
```

This item tells the host that the following data should be interpreted as "buttons", meaning that each bit represents some indicator that is either in the "On" or "Off" state.

```
0x19, 0x01, // Usage Minimum (01)

0x29, 0x01, // Usage Maximum (01)
```

These two items function to give the button in the following data a usage. If the mouse supported multiple buttons, then the usage maximum would be increased, and each button would be assigned a unique usage.

```
0x15, 0x00, // Logical Minimum (0)

0x25, 0x01, // Logical Maximum (1)
```

These two items function to show the lowest and highest possible value of the data that will follow. Since the data can only take an "On" or "Off" state, the minimum can be set to 0, and the maximum can be set to 1.

```
0x95, 0x01, // Report Count (1)

0x75, 0x01, // Report Size (1)
```

These two items tell the host that one data point will follow and that this data point will take one bit of space. This is the data bit that will show whether or not the left mouse button has been pressed.

```
0x81, 0x02, // Input (Data, Variable, Absolute)
```

This is the item that describes the data itself. The item tells the host that the data can change and that its value is absolute (i.e., not relative to any other data or axis).

```
0x95, 0x01, // Report Count (1)

0x75, 0x07, // Report Size (7)
```

These two items set up the padding needed to fill in the other bits of the byte containing the data pertaining to the button. These items refer to 7 1-bit data points.

```
0x81, 0x01, // Input (Constant) for padding
```

This item designates the above 7 bits of data to be constant. This will effectively create a byte of data where bit 0 contains information about the left mouse button, and bits 2 through 7 are padding.

```
0x05, 0x01, // Usage Page (Generic Desktop)

0x09, 0x30, // Usage (X)

0x09, 0x31, // Usage (Y)
```

These items describe the next group of data. The usage page is set to Generic Desktop to tell the host that the following data will be relevant to system software. The next two usages tell the host that the Generic Desktop data to follow pertain to X- and Y-axis information controlled by the system. This usage definition informs the host that the following data will be formatted such that the data pertaining to the X-axis will be transferred before data pertaining to the Y-axis.

```
0x15, 0x81, // Logical Minimum (-127)
```

```
0x25, 0x7F, // Logical Maximum (127)
```

These items describe the minimum and maximum values that the following data can take.

```
0x75, 0x08, // Report Size (8)
```

```
0x95, 0x02, // Report Count (2)
```

These items describe the data that follows as being two bytes (with 8 bits per byte).

```
0x81, 0x06, // Input (Data, Variable, Relative)
```

This item tells the host about the data's direction of flow, that the data can change in value, and that it is relative to some axis. In this case, the data measured relative to the change in mouse position since the last measurement. This item applies to both bytes of data.

```
0xC0,      // End Collection (Physical)
```

```
0xC0       // End Collection (Application)
```

These two items should be considered "closing parentheses" on the above-defined collections. The first End Collection closes the physical collection containing information about the mouse data point, and the second End Collection closes the top-level application collection.

7.3. Mouse Emulation Data Sampling

The routines that capture and save mouse-related data are found in the file *Mouse.c*, and some global variables are included in the *Mouse.h* header file. In the C8051F320/1 build, the *Mouse.c* file contains two routines, the "Timer 2 ISR" and the "ADC Sample Complete ISR". In the C8051F326/7 build, the *Mouse.c* file contains only a "Timer 2 ISR".

Initialization routines, as well as port configuration, are also found in the *Mouse.c* file.

7.3.1. Timer 2 ISR

Once properly configured to overflow and enabled as an interrupt, Timer 2 is used to take readings from the two switches on the 'F320 target board. Inside `Timer2_ISR()`, both switch values are captured by saving the state of the port pin connected to each switch. Timer overflows allow for switch "debouncing" by comparing switch values across two consecutive Timer 2 ISR servicings. Switch states are saved in variables of *Mouse.c* file scope.

In the C8051F326/7 build of the project, the Timer 2 ISR also controls mouse movement by setting the `Mouse_Vector` and `Mouse_Axis` to a pattern that causes the cursor to move on-screen.

7.3.2. Adc_ConvComplete_ISR

In the C8051F320/1 version of the build, the ADC is configured to take samples at the port pin connected to the potentiometer. The captured value of the potentiometer is translated to relative mouse movement by first converting the unsigned potentiometer value to a signed character value by subtracting 128. The value is then divided by 16 to reduce the movement sensitivity of the potentiometer. Dividing by 16 makes cursor movement on-screen smoother.

At this point, the potentiometer value has been centered around 0 and reduced so that the maximum value is 128/16 = 8. This value is then saved as relative X- or Y- Axis movement, depending on whether X-Axis or Y-Axis movement is selected. Axis selection is accomplished by using one of the switches to toggle between the two axes.

Inside the ADC ISR, variables for the mouse button, mouse vector (relative movement), and active axis are saved in the global variables `Mouse_Button`, `Mouse_Vector`, and `Mouse_Axis`, respectively. These variables will be formatted into a report inside the report handler, which is discussed in the following section.

SILICON LABS

7.4. IN Report Handler

Since only one input report was defined in the report descriptor of this example, only one input report handler is needed. This report will transfer mouse information stored the global variables `Mouse_Button`, `Mouse_Vector`, and `Mouse_Axis` to the host.

The function prototype for the report handler, called `IN_Report()`, has been placed in the appropriate spot at the top of the *F3xx_USB0_ReportHandler.c* file. `IN_VectorTableSize` is set to 1. The report handler is linked to a Report ID of 0. In the case where only one report is defined in the report descriptor, no Report ID should be defined, and 0 should be placed in the Report ID field of the Vector Table element.

The `IN_Report()` body takes the mouse global variables and stores them in a buffer following the format defined in the report descriptor. Bit 0 of byte 0 of the buffer is set to the `Mouse_Button` state. The report handler examines `Mouse_Axis` to determine whether the X-Axis or the Y-Axis is being manipulated and then sets either byte 1 or byte 2 of the buffer to `Mouse_Vector`, depending on whether the X-Axis or the Y-Axis is selected.

`IN_Buffer.Ptr` is set to the newly-formatted buffer, and `IN_Buffer.Length` is set to "3" following the report descriptor's definition of a 3-byte report.

This example does not require the use of any output reports. If it did, the function, `Setup_OUT_Buffer()`, would need to set the `OUT_Buffer` struct elements so that the firmware would know where to store output reports. Reports stored in this output buffer would then be processed by defined output report handlers. For an example of how such a system could look, read the next section that describes the creation of an HID Blinky firmware and software system.

7.5. Alterations to Main()

The `main(void)` function of the firmware template requires only a few modifications. After peripheral initializations, the system enters a `while(1)` loop. Inside this loop, the `SendPacket()` function is repeatedly called, with a Report ID of 0 passed in as the report to transmit to the host.

`SendReport()`, a function found in the *USB_ISR.c* file, checks to see if the IN Endpoint is set to `IDLE`. If it is not, meaning that another transfer is already in progress, `SendReport` exits with an error code of 0. If the IN Endpoint is set to `IDLE`, a transmission of the report with ID "0" is initiated.

8. HID Blinky Firmware and Software Example

This example shows a system where device-side firmware and host-side software transfer data between each other using both the Interrupt and Control data pipes. Like the first example, this one takes advantage of the switches, LEDs, and the potentiometer on Silicon Laboratories target boards.

8.1. Overview

The data transfers in this example are written to exercise every data path between the device and the host. Data travels through the IN and OUT endpoints and in both directions through the control endpoint. This example also shows how multiple reports can be employed to optimize data transfer.

This example exercises the following data paths:

- LED blinking patterns transmitted to the device in OUT reports across the Interrupt OUT endpoint
- Potentiometer position, used to choose from a list of blinking patterns, transmitted to the host in IN reports across the Interrupt IN endpoint
- LED blinking rate configuration transmitted to the device in OUT reports across the control endpoint
- LED blinking enable/disable command transmitted to the device in OUT reports across the control endpoint
- Runtime-calculated statistical information transmitted to the host in IN reports across the control endpoint

LED brightness adjustment and command acknowledgement are transmitted between device and host in feature reports across the control endpoint.

The HID Blinky Example's firmware was created by performing the following steps:

1. Modify descriptors so that the microcontroller appears as a device of some vendor-defined purpose.
2. Write code that controls LED lighting patterns.
3. Create a report descriptor that defines five reports of different sizes and types.
4. Initialize ADC, Timer 2, and Timer 0 peripherals.
5. Write report handlers for each of the reports.
6. Modify main(void) to poll for changes in data and initiate transfers when necessary.

8.2. Firmware System

The firmware system blinks LEDs according to patterns received from the host application. It also saves measurements concerning these patterns that can be retrieved by the host. The C8051F320/1 and C8051F34x builds alter LED brightness by adjusting pulse-width modulation signals connected to the LEDs.

8.2.1. Descriptors

Each of the following subsections describes modifications that need to be made to the firmware template's descriptors.

8.2.1.1. Device Descriptor

The device descriptor for this example looks like the following:

```
{
 18,              // bLength
 0x01,            // bDescriptorType
 0x1001,          // bcdUSB
 0x00,            // bDeviceClass
 0x00,            // bDeviceSubClass
 0x00,            // bDeviceProtocol
 EP0_PACKET_SIZE, // bMaxPacketSize0
```

SILICON LABS

```
0xC410,          // idVendor
0x0002,          // idProduct
0x0000,          // bcdDevice
0x01,            // iManufacturer
0x02,            // iProduct
0x00,            // iSerialNumber
0x01             // bNumConfigurations
}; //end of DeviceDesc
```

0x0002,   // idProduct

When a device gets connected to the host system for the first time, the system saves a record that links the device's vendor ID and product ID to the drivers the system determines it should use for communication. If an 'F320 is loaded with the USB mouse example and connected to the host system, the system determines that the USB mouse drivers should be used to interact with attached devices matching this product and vendor ID. Later, if the 'F320 is loaded with HID firmware for this example but with the same product and vendor ID, the system expects that the attached device is a USB mouse and tries to communicate with it using those drivers. Host application software will not be able to communicate with the device. In this device descriptor, the firmware has been given a distinct product ID.

0x01,   // iManufacturer

0x02,   // iProduct

Manufacturer and Product strings have been added to this design so that some information appears on-screen when the device is attached.

8.2.1.2. Configuration Descriptor

```
0x09,      // Length
0x02,      // Type
0x2900,    // Totallength (= 9+9+9+7+7)
0x01,      // NumInterfaces
0x01,      // bConfigurationValue
0x00,      // iConfiguration
0x80,      // bmAttributes
0x20       // MaxPower (in 2 mA units)
```

0x2900,   // Totallength (= 9+9+9+7+7)

This example contains one more descriptor than the USB mouse example because the design uses an OUT endpoint. The last seven bytes added to the total length include this descriptor.

0x20   // MaxPower (in 2 mA units)

This amount of current will be sufficient to supply the 'F320 with sufficient current to function properly.

8.2.1.3. Interface, IN Endpoint, and OUT Endpoint Descriptors

No changes to the firmware template need to be made to implement this design other than modifying the endpoint sizes to be at least as large as the largest reports.

8.2.1.4. Report Descriptor

This example uses six different reports, each with a unique report structure. Two input reports are created: one to transmit the potentiometer value and one to transmit device statistics displayed in the application window. Three output reports are created: one transfers selected LED blinking patterns; one enables and disables LED blinking, and one command sends a two-byte blinking rate to the device. One feature report is created to transmit LED dimmer levels and dimmer acknowledgement signalling. The report descriptor for this example appears as follows:

```
{
0x06, 0x00, 0xff,                  // USAGE_PAGE (Vendor Defined Page 1)
0x09, 0x01,                        // USAGE (Vendor Usage 1)
0xa1, 0x01,                        // COLLECTION (Application)

0x09, 0x01,                        // USAGE (Vendor Usage 1)

0x85, OUT_Blink_PatternID,         // Report ID
0x95, OUT_Blink_PatternSize,       // REPORT_COUNT ()
0x75, 0x08,                        // REPORT_SIZE (8)
0x26, 0xff, 0x00,                  // LOGICAL_MAXIMUM (255)
0x15, 0x00,                        // LOGICAL_MINIMUM (0)
0x09, 0x01,                        // USAGE (Vendor Usage 1)
0x91, 0x02,                        // OUTPUT (Data,Var,Abs)

0x85, OUT_Blink_EnableID,          // Report ID
0x95, OUT_Blink_EnableSize,        // REPORT_COUNT ()
0x75, 0x08,                        // REPORT_SIZE (8)
0x26, 0xff, 0x00,                  // LOGICAL_MAXIMUM (255)
0x15, 0x00,                        // LOGICAL_MINIMUM (0)
0x09, 0x01,                        // USAGE (Vendor Usage 1)
0x91, 0x02,                        // OUTPUT (Data,Var,Abs)

0x85, OUT_Blink_RateID,            // Report ID
0x95, OUT_Blink_RateSize,          // REPORT_COUNT ()
0x75, 0x08,                        // REPORT_SIZE (8)
0x26, 0xff, 0x00,                  // LOGICAL_MAXIMUM (255)
0x15, 0x00,                        // LOGICAL_MINIMUM (0)
0x09, 0x01,                        // USAGE (Vendor Usage 1)
0x91, 0x02,                        // OUTPUT (Data,Var,Abs)

0x85, IN_Blink_SelectorID,         // Report ID
0x95, IN_Blink_SelectorSize,       // REPORT_COUNT ()
```

SILICON LABS

```
0x75, 0x08,                    // REPORT_SIZE (8)
0x26, 0xff, 0x00,              // LOGICAL_MAXIMUM (255)
0x15, 0x00,                    // LOGICAL_MINIMUM (0)
0x09, 0x01,                    // USAGE (Vendor Usage 1)
0x81, 0x02,                    // INPUT (Data,Var,Abs)


0x85, IN_Blink_StatsID,        // Report ID
0x95, IN_Blink_StatsSize,      // REPORT_COUNT ()
0x75, 0x08,                    // REPORT_SIZE (8)
0x26, 0xff, 0x00,              // LOGICAL_MAXIMUM (255)
0x15, 0x00,                    // LOGICAL_MINIMUM (0)
0x09, 0x01,                    // USAGE (Vendor Usage 1)
0x81, 0x02,                    // INPUT (Data,Var,Abs)
0x85, FEATURE_BLINK_DIMMERID,  // Report ID
0x95, FEATURE_BLINK_DIMMERSIZE, //   REPORT_COUNT ()
0x75, 0x08,                    //   REPORT_SIZE (8)
0x26, 0xff, 0x00,              //   LOGICAL_MAXIMUM (255)
0x15, 0x00,                    //   LOGICAL_MINIMUM (0)
0x09, 0x01,                    //   USAGE (Vendor Usage 1)
0xB1, 0x02,                    //   FEATURE (Data,Var,Abs)
0xC0                           // end Application Collection


  };


 0x06, 0x00, 0xff,             // USAGE_PAGE (Vendor Defined Page 1)
 0x09, 0x01,                   // USAGE (Vendor Usage 1)
 0xa1, 0x01,                   // COLLECTION (Application)
```

These items tell the host system that the usage pages and usages found in this report descriptor are all vendor-defined, which means that the reports follow no standard format (such as the one used in the mouse example). The top-level application collection, tagged with vendor-defined usage, assumes that a host-side application will have some knowledge of the report structure and will be able to communicate with the device.

```
 0x85, OUT_Blink_PatternID,    // Report ID
 0x95, OUT_Blink_PatternSize,  // REPORT_COUNT ()
 0x75, 0x08,                   // REPORT_SIZE (8)
 0x26, 0xff, 0x00,             // LOGICAL_MAXIMUM (255)
 0x15, 0x00,                   // LOGICAL_MINIMUM (0)
 0x09, 0x01,                   // USAGE (Vendor Usage 1)
 0x91, 0x02,                   // OUTPUT (Data,Var,Abs)
```

SILICON LABS

The rest of the report descriptor is composed of similarly-worded information describing each report used in this example. Each report description begins with a report ID, which is defined in the *F3xx_Blink_Control.h* file. The next item tells the number of data items contained in this particular report. This value is also defined in the *F3xx_Blink_Control.h* file. The next item tells the host that each data item will be 1 byte in size. The next two items indicate that these data bytes can contain any value from 0 to 0xFF. The usage item tags the data that follows as vendor-defined with the same usage as the rest of the data. The last item of each report description tells the host whether data contained in this report is input or output.

8.2.2. Data Capture and Processing Routines

The data capture and storage of this system takes place inside the *BlinkControl.c* file and its associated header file, *F3xx_Blink_Control.h*. In the C8051F320/1 and C8051F34x versions of the build, the potentiometer's value is captured by saving the high byte of the ADC0 during an ADC complete ISR servicing. In the C8051F326/7 version of the build, switch 1's state is saved instead. This value is sent to the host program, which uses it to select a blinking pattern. Effectively, the user will be able to scroll through LED lighting sequence choices on-screen using the potentiometer.

To optimize USB bandwidth usage, the firmware system transmits a report containing the potentiometer's value only if the value has changed since the last transmitted report. The ADC ISR sets a global variable called `Blink_SelectorUpdate` whenever the new ADC value does not match the next value. The firmware system's foreground `while(1)` loop polls `Blink_SelectorUpdate` and when it finds the variable set, the foreground initiates a transfer of potentiometer information.

The host application transmits the lighting pattern, and the device saves this pattern in the `Blink_Pattern` array. The host transmits a blinking rate to the device, and the device saves it to variable, `Blink_Rate`. For a detailed discussion of these data transfers, read the next subsection.

The firmware system configures Timer 0 ISR to interrupt at a rate of 0.5 ms. Inside the ISR, `Blink_Rate` multiplies this interrupt rate by being used in a comparison with a counter that is incremented once every interrupt servicing. For example, if `Blink_Rate` is equal to 2000, the LED pattern will be updated at a rate of 0.5 ms x 2000 or 1 s.

The Timer 0 ISR measures the percentage of time each LED is active during the sequence. The ISR stores these measurements in the global variables `Blink_Led1Active` and `Blink_Led2Active`. The host has access to these variables through the use of the Control data pipe.

The Timer 0 ISR updates the lighting pattern by checking the lower two bits of an element of the `Blink_Pattern` array, where bit 0 controls `Led1`, and bit 1 controls `Led2`. The ISR increments an index variable used to access the `Blink_Pattern` array after every LED pattern update. When the element selected by the incremented index equals 0xFF, the index is reset to 0, and the pattern repeats.

8.2.3. Report Handler Creation

The five reports defined in the report descriptor each need a report handler. Prototypes for each handler have been added to the *F3xx_USB0_ReportHandler.c* file as follows:

```
void IN_Blink_Selector(void);

void OUT_Blink_Enable(void);

void OUT_Blink_Pattern(void);

void OUT_Blink_Rate(void);

void IN_Blink_Stats(void);

void FEATURE_Blink_Dimmer_Input(void);

void FEATURE_Blink_Dimmer_Output(void);
```

Each of these functions corresponds to one of the reports. The IN and OUT vector tables establish a link between these handlers and their associated reports.

SILICON LABS

The file defines the IN and OUT vector tables as follows:

```
#define IN_VectorTableSize 3
#define OUT_VectorTableSize 4
```

The IN Vector table contains the following elements:

```
IN_Blink_SelectorID, IN_Blink_Selector,
IN_Blink_StatsID, IN_Blink_Stats
FEATURE_Blink_DimmerID, FEATURE_Blink_Dimmer_Input
```

The OUT Vector Table contains the following elements:

```
OUT_Blink_EnableID, OUT_Blink_Enable,
OUT_Blink_PatternID, OUT_Blink_Pattern,
OUT_Blink_RateID, OUT_Blink_Rate
FEATURE_Blink_DimmerID, FEATURE_Blink_Dimmer_Output
```

The *F3xx_Blink_Control.h* header file must be included for the compiler to recognize the report IDs listed in these tables.

The function bodies for each handler mostly perform simple data transfer, either from a global variable and into an input buffer, or from an output buffer to some global variable.

Since this example uses output reports, the firmware needs a body for the function, `Setup_OUT_Buffer`. This function is called before data is retrieved from a buffer storing data transferred during either a control OUT transfer or an OUT endpoint transaction. The body of the function only needs to set the *OUT_Buffer.Ptr* struct to a data buffer and save the size of this buffer in the struct element, `Length`.

This example configures the `Setup_OUT_Buffer` as follows:

```
OUT_Buffer.Ptr = OUT_PACKET;
OUT_Buffer.Length = 10;
```

8.2.4. Alterations to `main(void)`

Only a few modifications need to be made to the `main(void)` function of the firmware template. `Blink_Init()` must be called to initialize the timer used for blink rate measurement. Inside the `while(1)` loop, the global variable defined in *F3xx_USB0_Main.c*, named `Blink_SelectionUpdate`, is polled. If this variable is set, the potentiometer value has changed, and the `main(void)` function initiates a transmission of a report containing potentiometer information by calling the function, `SendPacket()`.

8.3. Host-side PC Application

The host-side software application uses the CHIDDevice class to communicate with the HID Blinky device. The following subsections describe how the application connects to the device and how data is transferred using the CHIDDevice class.

8.3.1. Project Overview

The application was created using dialog-based MFC classes in Visual C++. All application functionality is contained within the `HID_BlinkyDlg` dialog. Controls and communication are contained in *HID_BlinkyDlg.cpp*. All low-level HID communication is handled by the CHIDDevice, which is defined in *HIDDevice.cpp* and *HIDDevice.h*.

8.3.2. HID Device Initialization

The dialog class's header file, called *HID_BlinkyDlg.h*, includes the file, *HIDDevice.h*, which declares all member functions and variables for the CHIDDevice class. The `CHID_BlinkyDlg` definition in *HID_BlinkyDlg.h* contains the following line:

```
CHIDDevice HID_Blinky;
```

This line defines `HID_Blinky`, which is the variable that the application uses to communicate with the HID Blinky device.

8.3.3. Device Connection/Removal Detection

The application recognizes three different device connection states:

- Disconnected - Device not detected on the USB.
- Connected, Idle - Device detected, but the application has not opened a communications link.
- Connected, Active - Application has established a communications link with the device.

Device detection is handled using three notification routines. These routines enable the software to monitor the USB for devices and react anytime a device is connected or disconnected. The routines operate as follows:

- `RegisterNotification()` creates a notification handle that is signaled whenever a device is connected to or disconnected from the USB.
- `OnDeviceChange()` is called whenever a device is attached or removed from the bus; connect/disconnect event notification is stored in the function parameter, `nEventType`.
- `UnregisterNotification()` tells the system to no longer send notification of device connection/disconnection

`RegisterNotification()` is called when the dialog box is created, and `UnregisterNotification()` is called whenever the dialog box closes. `OnDeviceChange()` contains a switch statement on `nEventType` that recognizes two different notification event types:

- `DBT_DEVICEARRIVAL` verifies that an HID Blinky device was connected by calling the CHIDDevice member function, `GetConnectedDeviceNum()`, to search for a device with a VID and a PID that match HID Blinky's VID and PID.
- `DBT_DEVICEREMOVECOMPLETE` verifies that the HID Blinky device was disconnected by calling `GetConnectedDeviceNum()` to check that no device is attached to the bus with HID Blinky's VID and PID.

8.3.4. Opening a Communication Link with a Device

A communication link is established with a connected device by pressing the "Activate" control button, which calls `GetConnectedDeviceNum()` to check that the device is connected to the bus; it then calls the CHIDDevice member function, `IsOpened()`, to check that a communications link has not already been established with this device. If no link exists, `Open()` is called to establish one.

A link is terminated by pressing the "Deactivate" button. Link termination is accomplished by first checking that a link exists using `IsOpened()`. If a link exists, the software calls `Close()` to terminate the link.

SILICON LABS

8.3.5. Controls and Reports

Operating the buttons, check boxes, and sliders calls routines with names that begin "On…" (`OnBnClickedDisconnect()`, `OnBnClickedRate ()`, etc.). These functions then call routines with names beginning "`Update_...`" that modify variables and communicate with the attached HID device. The `Update_...` routines operate as follows:

- `Update_BlinkPattern()` transmits a BlinkPattern output report and transmits it by calling `SetReport_Interrupt()`.
- `Update_BlinkEnable()` transmits a `Blink_Enable` output report by calling `SetReport_Control()`.
- `Update_BlinkRate()` transmits a BlinkRate output report by calling `SetReport_Control()`.
- `Update_Stats()` receives a `Blink_Stats` input report by creating a buffer, setting the first element of that buffer to the `Blink_Stats` report ID and calling `GetReport_Control()`.
- `Update_Dimmer()` transmits a `Blink_Slider` feature report by calling `SetFeatureReport()` and then calls `GetFeatureReport()` to receive an acknowledgement from the device.

The application receives input reports across the interrupt pipe by creating a thread that constantly attempts to retrieve reports. This thread is created when a communication link is established with the device, and is destroyed when the communication link is terminated.

The thread calls `InterruptThreadProc()`, which continuously calls `HID_InterruptGetReport ()`, which, in turn, calls the CHIDDevice member function, `GetReport_Interrupt()`. If `GetReport_Interrupt()` successfully retrieves a report, `HIDcallback()` processes received data. In this application, `HIDcallback()` handles `Blink_Selector` input report reception.

9. HID to UART Firmware and Software Example

This example creates a USB device that transfers bytes between an HID interface and a UART interface. The example also includes a dialog box-driven PC application that allows users to send and receive bytes through HID reports.

9.1. Overview

The HID to UART example has the following functionality:
- Device to PC application data transfer - The device buffers and sends data to a host across USB in reports; the PC application then retrieves these reports once it has established a handle with the device.
- PC application to device data transfer - The PC application takes input bytes and sends them to the device in reports, which the device then buffers and transmits across the UART interface.
- Device/Host control signaling - the host can request UART configuration settings from the device in a configuration report, and the host can modify those settings by sending a configuration report to the device.

Figure 4 shows the bidirectional data flow created by this example.

Figure 4. HID to UART Data Flow Chart

9.2. Firmware System

Figure 5 is a block diagram for the HID to UART firmware. The device firmware was created using the firmware template as a starting point. The following sections discuss the modifications that were made to the firmware template, and the files that were added to the project.

Note that the firmware contains a pre-compiler directive, BAUDRATE_HARDCODED, in F3xx_HIDtoUART.h. Including this directive in the build will hard-code UART communication to a baudrate defined by the pre-compiler directive, BAUDRATE. Additionally, including BAUDRATE_HARDCODED will prohibit the compiler from building any section of code that allows the baudrate to be changed at runtime. Removing BAUDRATE_HARDCODED from the firmware allows the compiler to include code that enables the baudrate to be configured at runtime. The firmware system will compile to less than 4K of code only if BAUDRATE_HARDCODED is included in the build.
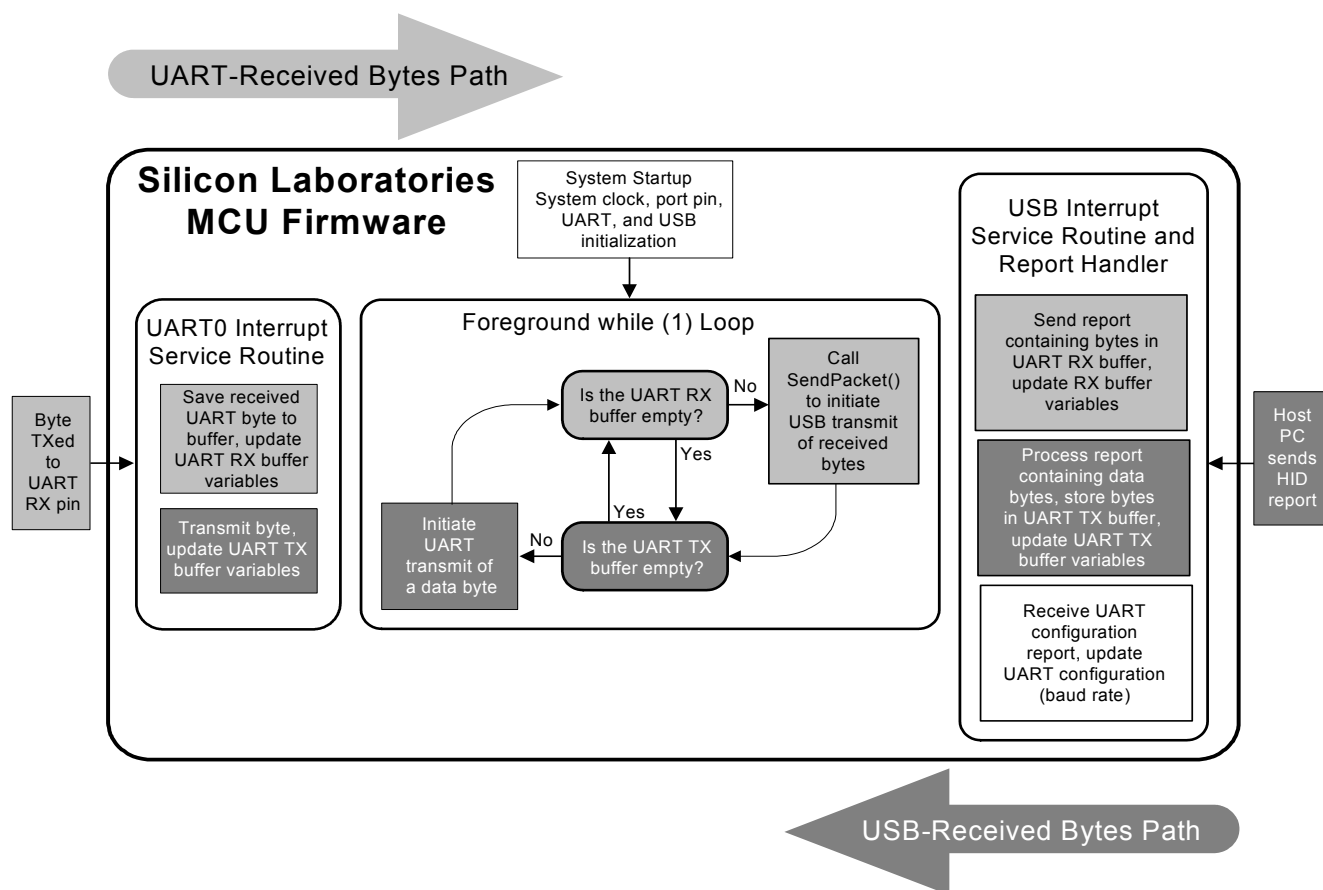
Figure 5. Firmware Data Flow Chart

9.2.1. Changes to `F3xx_USB0_Descriptor.c`

Two changes were made to the Firmware Template's F3xx_USB0_Descriptor.c file: the product string was changed to "HID to UART Example", and the Report descriptor was defined. The example uses four different reports to send and receive data and control information.

- `IN_DATA` – IN report for UART-received bytes the device sends to the host.
- `OUT_DATA` – OUT report for USB-received bytes the host sends to the device.
- `IN_CONTROL` – IN report containing device's current UART configuration information that the device sends to the host when requested.
- `OUT_CONTROL` – OUT report sent by the host to modify the device's UART configuration settings.

Note: The device will correctly respond to IN_CONTROL and OUT_CONTROL reports only if the pre-compiler directive, BAUDRATE_HARDCODED, is not defined in the project's build. See "9.2. Firmware System" for details.

F3xx_HIDtoUART.h contains the #define statements for Report IDs and Report sizes. IN and OUT data reports have the structure shown in Figure 6. IN and OUT control reports have the structure shown in Figure 7.

Report ID (IN = 0x01, OUT = 0x02)	N (# of Bytes in This Report)	Data buffer containing N bytes	Unused space in Report
Byte 0	Byte 1	Byte 2 Byte N + 2	Byte N + 3 to IN/OUT_DATA_SIZE

Figure 6. Data Report Structure

Report ID (IN = 0xFE, OUT = 0xFD)	Baud rate				Unused space in Report
Byte 0	Byte 1	Byte 2	Byte 3	Byte 4	Byte 5 to IN/OUT_CONTROL_SIZE

Figure 7. Control Report Structure

9.2.2. Changes to `F3xx_USB0_ReportHandler.c`

The `F3xx_USB0_ReportHandler.c` file was modified so that the firmware can process the four reports above. The input report vector table and output report vector table each contain two entries, one for the data report and one for the control report. These entries link the report IDs to the function bodies defined later in the file. The report handler functions perform the following tasks for each report type:

- `IN_DATA` writes bytes pulled from the UART-received data buffer into a data report to be transmitted to host.
- `OUT_DATA` copies USB-received data bytes from the report into a buffer to be transmitted out UART.
- `IN_CONTROL` stores the current UART baud rate in the control report to be transmitted to the host.
- `OUT_CONTROL` updates the UART baud rate using the baud rate stored in the report sent by the PC application.

9.2.3. Changes to F3xx_USB0_Main.c

`main()` initializes all peripherals and enters a `while(1)` loop where two conditional statements monitor the UART- and USB-received bytes buffers. When a conditional statement in the loop detects that the UART has buffered received bytes, the system initiates a USB transfer. Similarly, when a second conditional statement detects that the USB has buffered received bytes, the system initiates a UART transmit session.

9.2.4. Files Added to the Firmware Template

This example uses two files in addition to the files provided by the firmware template. The file named `F3xx_HIDtoUART.h` contains all the `#define` statements for report IDs and report sizes, as well as `extern` declarations for all buffer variables. The `F3[xx]_HIDtoUART.c` file set (only one of these files is included in the project's build, depending on the Silicon Laboratories microcontroller being used) contains all initialization routines and the UART interrupt service routine, which stores bytes in a received bytes buffer and transmits bytes stored in a transmit buffer.

9.2.5. Changes to Fxxx_USB0_InterruptServiceRoutine.c

In order to optimize throughput, achieve a high maximum UART baud rate, and prevent UART buffer overflow, some modifications needed to be made to USB-related routines.

In this example, the `Handle_Out2 ()` routine runs a check to detect whether the buffer used to store bytes received through USB has reached a size where receiving one more OUT_DATA report from a host would cause the buffer to overflow. When the buffer size crosses this threshold, the firmware will not set the Out Packet Ready bit in EOUTCSRL SFR. This forces the USB hardware to NAK all reports sent to the OUT endpoint. When the host's system drivers receive a NAK from a device on a transmitted report, the host will retry transmission periodically. When OUT reports are being NAKed, the firmware sets a flag called `USB_SUSPEND`.

A foreground test in `main()` checks to see whether `USB_SUSPEND` has been asserted. If `USB_SUSPEND` equals 1, the firmware then checks to see whether the buffer receiving USB bytes is still above the critical threshold. If the buffer size has fallen below the threshold, `main()` sets Out Packet Ready, which allows the USB peripheral to ACK incoming OUT reports.

`SendPacket()` was modified to maximize UART throughput. The version of `SendPacket()` included with the HID Firmware Template globally disables interrupts at the beginning of the function and re-enables them when the function's execution has completed. Leaving interrupts disabled prevents the interrupt-driven UART from sending or receiving bytes. To give the UART a chance to operate, the code was changed so that only USB interrupts are disabled inside the routine. This prevents a USB interrupt from altering USB SFRs and FIFOs while SendPacket() modifies them.

SILICON LABS

The example creates a flag called `SendPacketBusy` that is set whenever `SendPacket()` is executed and is cleared by the interrupt IN handler. The system uses this variable to avoid calling `SendPacket()` continuously. Without this flag, `SendPacket()` would be called in `main()` whenever that at least one byte exists in the UART-received bytes buffer. With this variable, `main()` can check to see if the last IN report written to USB FIFO space has been transmitted. `main()` will call `SendPacket()` again only after `SendPacketBusy` has been cleared.

9.3. Host-Side PC Application

Like the HID Blinky PC application, this example communicates with a device using the CHIDDevice class. The PC application in this example does the following:

- Searches for an attached HID to UART device and establishes a connection
- Reads and sets the device's baud rate
- Outputs received bytes to a text window
- Sends bytes typed into an input text window to the device
- Transmits a file selected by the user to the device

The dialog window created by the application is shown in Figure 8.

Figure 8. PC Application Dialog Window

9.3.1. Connecting to a Device

The PC application attempts to connect to an HID-to-UART device whenever a user presses the "Connect" button. The procedure the application uses to connect to the device is similar to the procedure used by the HID Blinky application, where the application checks for attached devices with the HID to UART device's VID and PID. If such a device is found, the app calls the CHIDDevice class member function `Open()` to establish a connection by creating a handle.

9.3.2. Receiving Data from the Device

When the application establishes a connection to a device, the app then creates a thread that repeatedly checks for interrupt IN reports from the device using the CHIDDevice class member function, `GetInputReport_Interrupt()`. When a device sends a data report to the host, the PC application retrieves the report using this thread and then calls `HIDcallback()`. In this routine, a switch statement checks the retrieved report's ID and processes data reports. Terminal mode retrieves these bytes and displays them in the 'Output from device" window. File Transfer Mode writes data bytes to the file shown in the "RXed from dev." window. The test pattern generator compares received bytes to transmitted bytes when the user presses "Check Pattern."

The application retrieves the baud rate used by the device by calling `GetInputReport_Control()`. The application writes the report ID for an `IN_CONTROL` report into the buffer passed into the routine. The device receives the request with the report ID and sends back the baud rate. The app then writes this value in the baud rate text box.

9.3.3. Sending Data to the Device

The PC application sends data to the device using the HID DLL's `SetOutputReport_Interrupt()` routine. Terminal Mode calls `SetOutputReport_Interrupt()` after collecting characters typed into the "Input to device." File Transfer mode calls the routine when "Transfer File" is pressed, after reading bytes from the designated input file. The pattern generator calls the routine to transfer the reports containing the data pattern. The application sends a command to change the baud rate using the HID DLL's `SetOutputReport_Control()` routine.

SILICON LABS

DOCUMENT CHANGE LIST

Revision 0.1 to Revision 0.2

- Added Section "2. Introduction To HID" on page 2.
- Added Section "6. HID DLL and CHIDDevice Object Classes" on page 20.
- Modified Section "8. HID Blinky Firmware and Software Example" on page 36 to conform to latest firmware and software example.
  - Minor formatting changes.

Revision 0.2 to Revision 0.3

- Changed section discussing CHIDDevice Class so that it refers to the HID DLL.
- Added section discussing the HID to UART example.
- Reworded introduction to reflect changes in document.

CONTACT INFORMATION

Silicon Laboratories Inc.
400 West Cesar Chavez
Austin, TX 78701
Email: MCUinfo@silabs.com
Internet: www.silabs.com

SILICON LABS