HUMAN INTERFACE DEVICE TUTORIAL

Relevant Devices

This application note applies to the following devices:
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. With its on-chip integration of 25 MIPs CPU, robust USB interface, and numerous peripherals, the C8051F32x product family is versatile enough to suit just about any HID-Class device.

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 protocol. The document concentrates on highlighting the benefits of designing with HID and techniques for creating and optimizing HID-based systems that use a C8051F32x microcontroller.

This document includes the following:

- Discussion on HID
- A Firmware Template that can be used as a starting point for HID device firmware
- Explanation of two 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 included in this application note were created using the included Firmware Template as a starting point. The sections titled “USB Mouse Example” and “HID Blinky Firmware and Software Example” describe how the Firmware Template was modified to create each example.

Copyright © 2006 by Silicon Laboratories

2. HID 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 with a computer system.

2.1. Universal Serial Bus

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 native operating system or customized drivers.

2.1.1. 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.

2.1.2. C8051F32x Capabilities

The C8051F32x microcontrollers can support a control endpoint and up to three additional endpoints. 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 can trigger the servicing of an interrupt service routine (ISR) if interrupts have been enabled.

2.1.3. 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 to determine how best to interact with an attached USB device.

2.2. Human Interface Device Class

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

2.2.1. 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 great detail later in this document.
- HID devices must respond to standard HID Requests in addition to all standard USB requests.

2.2.2. Device Operation

All HID devices follow the same basic steps during their operation. Upon connection to the host’s USB port, the host initiates an enumeration sequence. During this sequence, the host retrieves Descriptors from the device to determine basic information about the device. HID devices must also transfer descriptors defining the structure of HID Reports.

After enumeration, the host determines what drivers to use when interacting with the device. Since all HID devices use a standard HID driver, system designers do not need to develop a custom USB driver.

3. 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. The device is required to respond with descriptors that follow a standard format. The next section briefly discusses the descriptor structure a host expects to receive. The two sections after that describe the responsibilities of the device and the host during the process of enumeration.

These sections refer to sections of the HID Firmware Template, which is discussed in greater detail later in this document.

3.1. Descriptor Structure

Descriptors contain all basic device information about a device. The USB specification defines some of the descriptors retrieved, and the HID specification defines other required descriptors.

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 so that the host can 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. If enumeration fails, the device will not be allowed to interact with the host.

In the C8051F32x family of devices, 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.

3.1.1. Descriptor Declaration Example

```
A declaration might look as follows:
//-------------------------------------------
// 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.

3.1.2. Descriptor Definition Example

The definition might look as follows:

```
const 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.

3.1.3. A Reminder About Descriptors

Using descriptors to communicate the device's capabilities to the host might seem like an intimidating task, but many HID devices have very similar descriptor contents. The HID Firmware Template includes descriptor settings to create a device that meets minimum HID requirements. 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.

3.2. 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, which is located in the *USB_ISR.c* file.

3.2.1. The Control Endpoint Handler

If a Control Endpoint-related event triggers the USB ISR, the 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 *USB_STD_REQ.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.

3.3. 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 must then locate the device by checking information stored in the operating system.

3.4. 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.

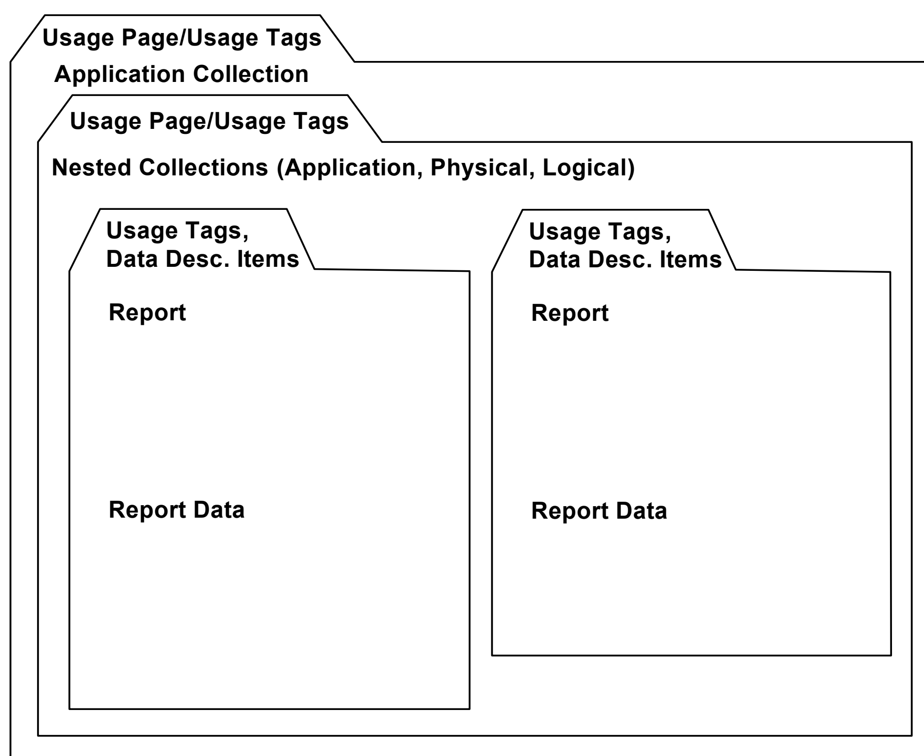

Figure 1. Report Descriptor Example Structure

3.4.1. 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 1 for a Report Descriptor's example structure.

3.4.1.1. 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.

3.4.1.2. 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 on the USB.org website.

3.4.1.3. 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, logical collections, and physical collections. Application Collections group variables that carry out a common purpose. All report items must be contained inside an Application Collection. A Logical Collection groups 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.

3.4.1.4. 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 depending on how the system is designed.

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 on the items in a Report Descriptor, see the latest revision of the HID specification.

SILICON LABORATORIES

3.4.2. 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 `Get_Report()` and `Set_Report()`. 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.

3.4.2.1. 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.

Two of these HID-specific requests, `Set_Report()` and `Get_Report()`, allow host applications to send and receive reports. Parameters passed in with the call to Set Report include the handle to the HID device, the buffer containing the report, and the number of bytes to transmit. Similarly, calls to `Get_Report()` 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.

In firmware systems that define more than one Report structure, host applications must set the first byte of the buffer passed as a parameter to the Report ID of the report the host wishes to retrieve. Similarly, the first byte of the buffer transferring a report during a call to `Set_Report()` should be set to the Report ID corresponding to the report being sent.

The API calls first send a packet to the Control Endpoint of the device that contains reserved command byte corresponding to `Set_Report()` or `Get_Report()`. In the case of the `Set_Report()` command, the system then transmits a second packet containing the report. In the case of the `Get_Report()` 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.

3.4.2.2. 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 `Set_Report()` 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 `Get_Report()` request has been transmitted, the C8051F32x switches the handler into `EP_GetReport` mode. The host then transmits a report to the device. After the report has been transmitted, the C8051F32x signals firmware of the availability of the report, and the report can be retrieved from the buffer.

3.4.2.3. 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.

3.4.3. 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 C8051F320 signals the firmware and the OUT Endpoint handler retrieves the bytes from the OUT Endpoint buffer.

4. 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.

4.1. 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.

4.2. Template Files

The HID Firmware Template includes the following files.

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

4.3. 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 *USB_Descriptor.c*, modify descriptors as needed.
- In *USB_ReportHandler.c*, add routines to handle all input and output data packets and populate the Report Handler Vector tables.
- In *USB_Main.c*, add any necessary foreground routines for processing USB traffic.
- Add code to control any other peripherals needed to implement the system.

4.4. Default Descriptor Declaration Values

The file *USB_Descriptor.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 LABORATORIES

4.4.1. 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 Sub Class 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 *USB_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.

SILICON LABORATORIES

4.4.2. 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
 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.

4.4.3. 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.

4.4.4. 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 *USB_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.

4.4.5. 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 *USB_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.

SILICON LABORATORIES

4.4.6. HID Descriptor

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

The descriptor looks as follows:

```
{ // 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 a 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.

`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 *USB_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.

4.4.7. 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 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.

4.4.8. 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.

4.5. USB_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.

4.5.1. Behavior of the USB_ReportHandler.c File

The USB firmware system calls handlers defined inside *USB_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 *USB_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 the firmware system reaches a point where a report needs to be prepared or processed, it calls either `ReportHandler_IN()` or `ReportHandler_OUT()`, passing into these functions only the Report ID. These functions search the Report Vector Tables for a matching Report ID and then call the corresponding Report Handler.

4.5.2. 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 "5. USB Mouse Example" .

4.5.3. 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 *USB_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.

4.5.4. 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.

5. USB Mouse Example

This section examines the steps taken to emulate a USB Mouse using the C8051F32x target board. While the firmware system of the mouse will be relatively simple, this example still touches on all necessary firmware modifications that need to be made for all HID-class device designs derived from the Firmware Template.

The information contained in the descriptors tells the host system that the attached device is a mouse, and the system will read reports sent by the device. The USB Mouse Example needs no host-side application to operate correctly. For an example of host-side application software, read the next section, titled “USB Blinky Device And Host Firmware Example.”

5.1. Overview

The goal of this example is to emulate the behavior of a USB mouse using the C8051F32x target board. The example contains two versions of low-level mouse emulation code. Code running on a C8051F320/1 microcontroller will emulate mouse movement by reading movements of the potentiometer on the target board. The axis of moment (whether motion on the potentiometer translates to up-down motion or left-right motion) will be controlled by one of the switches found on the target board. The switch will act as a one shot, with every press of the button switching between X-axis movement or Y-axis movement. The other switch on the target board will be used to emulate the left mouse button. Code running on a C8051F326/7 will emulate movement by setting mouse vector values in a set pattern.

Create a “USB Mouse” 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 device build) 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.

5.2. 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.

5.2.1. Device Descriptor

The device descriptor for this example looks as follows. 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
```


SILICON LABORATORIES

```
 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.

5.2.2. 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 enough current to supply the microcontroller.

5.2.3. 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.

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.

5.2.4. 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.

5.2.5. 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.

5.2.6. 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 7 bits of padding with two bytes of data describing the X- and Y- axis positions of the pointer.

The report descriptor for the Mouse Emulation Example looks as follows:

```
{
 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 would 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, which is in this case the system software.

```
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.

5.3. 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.

5.3.1. 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.

5.3.2. 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 LABORATORIES

5.4. 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 *USB_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 `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.

5.5. 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.

6. USB 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 pipes. Like the first example, this one takes advantage of the switches, LEDs, and the potentiometer on C8051F32x target boards.

6.1. Overview

The data transfers in this example are written to exercise all possible data paths between the device and the host. Data travels through the IN and OUT endpoints, and travels in both directions through the Control Endpoint. This system also provides an example of how multiple reports can be created and employed to optimize data transfer.

In this example, a user scrolls with the potentiometer to choose from a list of different blinking patterns listed on the application window. The firmware system then transmits the potentiometer value to the host across the IN interrupt pipe. The application then sends reports containing the blinking sequence of the selected pattern using the OUT interrupt pipe. The speed at which the device runs through the blinking pattern is controlled by data transferred to the device across the Control Output pipe. Device measurements and statistics can be retrieved across the Control Input pipe.

Create a USB Blinky Example by taking 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 5 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.

6.2. Firmware System

The firmware system will blink LEDs according to patterns received from the host application. It will also save measurements concerning these patterns that can be retrieved by the host.

6.2.1. Descriptors

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

6.2.1.1. Device Descriptor

The device descriptor for this example looks as follows:

```
{
 18,               // bLength
 0x01,             // bDescriptorType
 0x1001,           // bcdUSB
 0x00,             // bDeviceClass
 0x00,             // bDeviceSubClass
 0x00,             // bDeviceProtocol
 EP0_PACKET_SIZE,  // bMaxPacketSize0
 0xC410,           // idVendor
 0x0002,           // idProduct
 0x0000,           // bcdDevice
 0x01,             // iManufacturer
```


SILICON LABORATORIES

```
 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 with 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.

6.2.1.2. 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 7 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 enough current to function properly.

6.2.1.3. 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.

6.2.1.4. Report Descriptor

This example uses five 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 that are displayed in the application window. Three output reports are created: one to transfer selected LED blinking patterns, one for enable and disable LED blinking, and one command sends a two-byte blinking rate to the device.

The Report Descriptor for this example looks 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 ()
 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)

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)
```

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 *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 *Blink_Control.h* file. The next item tells the host that each data item will be 1 byte in size. The two next 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.

6.2.2. 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 *BlinkControl.h*. In the C8051F320/1 version 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 chosen by the potentiometer value, 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 on 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 makes a few simple calculations to measure 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.

6.2.3. 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 *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);
```

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.

The file defines the IN and OUT Vector Tables as follows:

```
#define IN_VectorTableSize 2
#define OUT_VectorTableSize 3
```

The IN Vector table contains the following elements:

```
IN_Blink_SelectorID, IN_Blink_Selector,
IN_Blink_StatsID, IN_Blink_Stats
```


SILICON LABORATORIES

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
```

The *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 to 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;
```

6.2.4. 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 *BlinkControl.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()`.

6.3. Software System

The creation of a complete Visual Studio software system is beyond the scope of this application note. However, this document does provide a working example application that can be modified to work with many custom USB devices. The main focus of this section is to highlight a few important HID-related functions in this software example and to explain how these functions communicate with the attached device.

6.3.1. On-Screen Display

The software system of this example allows users to enter data on-screen and have it sent to the device. Values for the blink rate and whether or not blinking has been enabled are controlled by buttons. When these buttons are pressed, information is sent to the device.

The firmware system also displays information retrieved from the device. The host application displays potentiometer or switch selections as they are received. Pressing the “Get Info” button causes the system to retrieve a report from the device containing statistical information and display that on-screen.

6.3.2. Device Detection

Located in *UsbHid.cpp*, the `HidOpen()` routine attempts to detect an attached HID-class device by searching Product IDs and Vendor IDs. The following line passes the Vendor ID and the Product ID into a routine that establishes a connection and returns Read and Write handles for the device.

```
if(hid_open(p,&hDevNotify,&hUSBRead,&hUSBWrite,/*VendorID*/0x10C4,/*ProductID*/0x020
0,&version,&usage,&usagepage,&ilen,&olen))
```

If the device is attached to the system, this function detects it and creates a “read handle” and a “write handle” used for data transfers during application communication.

6.3.3. Control Transfers

This software system provides an example of both input and output control transfers. The HID specification defines two routines that allow an application to send or receive reports with HID devices.

6.3.3.1. Input Reports

An input report, where data travels from the device to the host, can be requested using the Windows API routine `HidD_GetInputReport()`, which is called inside the routine `UsbHidRead()`.

The parameter passed into the function `UsbHidRead()` defines the Report ID of the report the host system wants to retrieve. `UsbHidRead()` stores this report ID in the first element of a buffer that is then passed into `HidD_GetInputReport()`. `GetInputReport()` commands the host system to send a request with the Report ID sent as part of the setup packet.

The firmware responds to the request and transmits the report back to the host system. If the request was successful, the `GetInputReport()` call returns a buffer containing the received Report.

The buffer containing this report is defined in the *UsbHid.cpp* file. This buffer “`buf`” should be large enough to hold the biggest Report plus a single byte for the Report ID. The declaration of this buffer uses the pre-compiler definition `LargestReportSize` to determine the size of the buffer.

In this system, an Input Control transfer is used to retrieve the two-byte Report containing statistics gathered by the device. When the user presses the “Get Info” on-screen button, a function called `OnBnClickedStats()` inside *USBTestDlg.cpp* is called. This function contains the following function call:

```
UsbHidRead(IN_Blink_Stats);
```

If this function successfully retrieves a Report, the `OnUsbRx()` routine processes the Report and displays the statistics on-screen.

SILICON LABORATORIES

6.3.3.2. Output Reports

The system uses Control Output Reports to send two different reports to the device. By pressing the "Set New Rate" button, the system sends the value typed into the adjacent box to the device. By pressing the "Enable/Disable" button, the system sends a one-byte enable or disable command to the device. Both of these transfers are accomplished by calling the `UsbHidWrite()` routine.

The `UsbHidWrite()` routine takes three parameters: the buffer containing the report to transmit, the number of bytes to transmit, and whether the transmit should use the "CONTROL" or "INTERRUPT" pipe. For "CONTROL" Pipe transfers, `UsbHidWrite()` calls the standard Windows API routine named `HidD_SetOutputReport()`. This routine sends a Set Report standard HID request to the device, a returns a value showing the number of bytes successfully transmitted across the Control Pipe.

6.3.4. Interrupt Transfers

This system provides an example of both IN and OUT Endpoint transfers. These transfers use the standard Windows API calls `ReadFile()` and `WriteFile()`.

6.3.5. IN Endpoint Transfers

When the firmware system captures a potentiometer reading and calls `USB_SendPacket_EP1()`, the host system retrieves this information and stores it in a system buffer. After retrieval, an application can retrieve this information from the system buffer and process it using the routine `Readfile()`. This example periodically runs a function called `UsbHidStep()`, which in turn calls `Readfile()`. This routine polls the system buffers for any newly-arrived reports. `Readfile()` parameters include the Read handle for the device and the number of bytes to be retrieved. If `Readfile()` exits successfully, it signals `OnUsbRx()`, the potentiometer reading is processed, and data is displayed on-screen.

6.3.6. OUT Endpoint Transfers

The system initializes an OUT Interrupt transfer when the potentiometer's value has risen or fallen to the point where a new blink pattern on the list has been selected. The new blink pattern is sent to the device by calling the `UsbHidWrite()` function and passing into it a buffer containing the new blink pattern, the size of the Report, and the "INTERRUPT" flag which tells the function to use the Interrupt Pipe.

The function `UsbHidWrite()` calls the standard Windows API routine named `WriteFile()`. This routine takes the handle for the USB device, the buffer to transmit, a number of bytes to transmit, and returns the number of bytes successfully transmitted.

Contact Information

Silicon Laboratories Inc.
4635 Boston Lane
Austin, TX 78735
Email: MCUinfo@silabs.com
Internet: www.silabs.com

The information in this document is believed to be accurate in all respects at the time of publication but is subject to change without notice. Silicon Laboratories assumes no responsibility for errors and omissions, and disclaims responsibility for any consequences resulting from the use of information included herein. Additionally, Silicon Laboratories assumes no responsibility for the functioning of undescribed features or parameters. Silicon Laboratories reserves the right to make changes without further notice. Silicon Laboratories makes no warranty, representation or guarantee regarding the suitability of its products for any particular purpose, nor does Silicon Laboratories assume any liability arising out of the application or use of any product or circuit, and specifically disclaims any and all liability, including without limitation consequential or incidental damages. Silicon Laboratories products are not designed, intended, or authorized for use in applications intended to support or sustain life, or for any other application in which the failure of the Silicon Laboratories product could create a situation where personal injury or death may occur. Should Buyer purchase or use Silicon Laboratories products for any such unintended or unauthorized application, Buyer shall indemnify and hold Silicon Laboratories harmless against all claims and damages.

Silicon Laboratories and Silicon Labs are trademarks of Silicon Laboratories Inc.

Other products or brandnames mentioned herein are trademarks or registered trademarks of their respective holders.