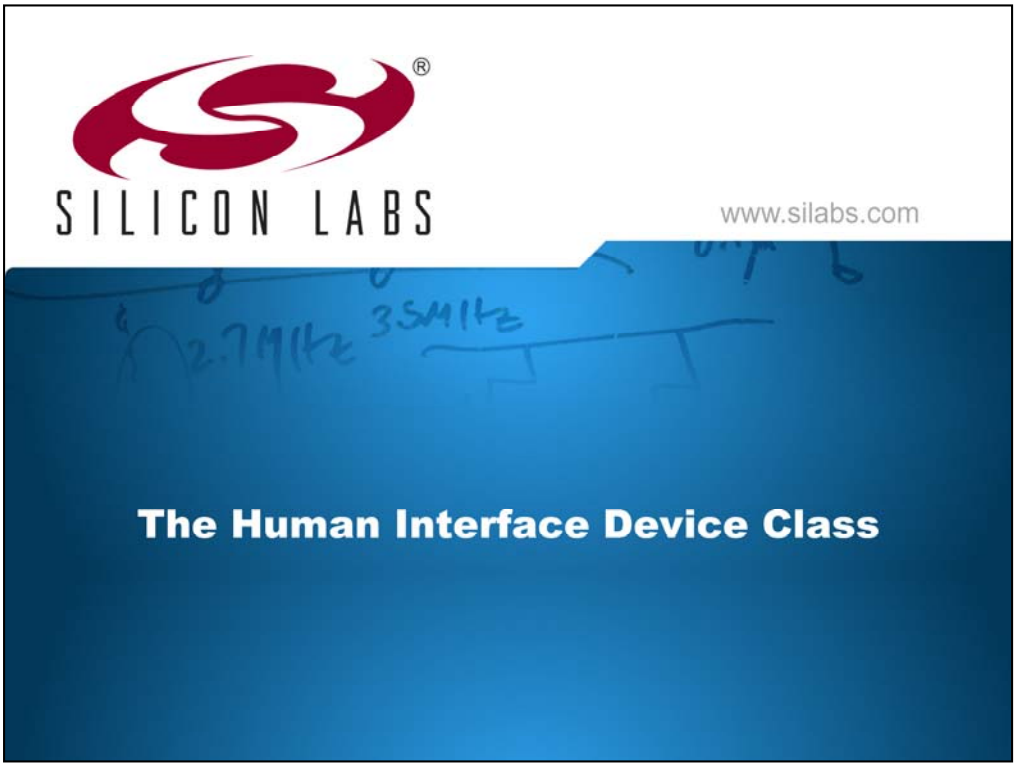

SILICON LABS
www.silabs.com
The Human Interface Device Class

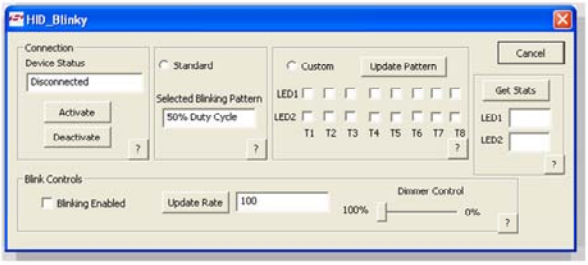
Agenda

- **USB Human Interface Device Class (HID) overview**
- **USB HID firmware examples**
  - USB mouse example
  - Generic application example
- **USB host application example**

SILICON LABS

HID Interface

- **HID is a defined USB class that most operating systems support natively**
  - The end customer does not need to install drivers
  - Digitizer usage included in Windows 7 and Windows Vista
- **The HID class definition is flexible enough to accommodate many different kinds of USB designs**
- **The class also defines a number of HID subclasses, such as the mouse and keyboard subclasses**



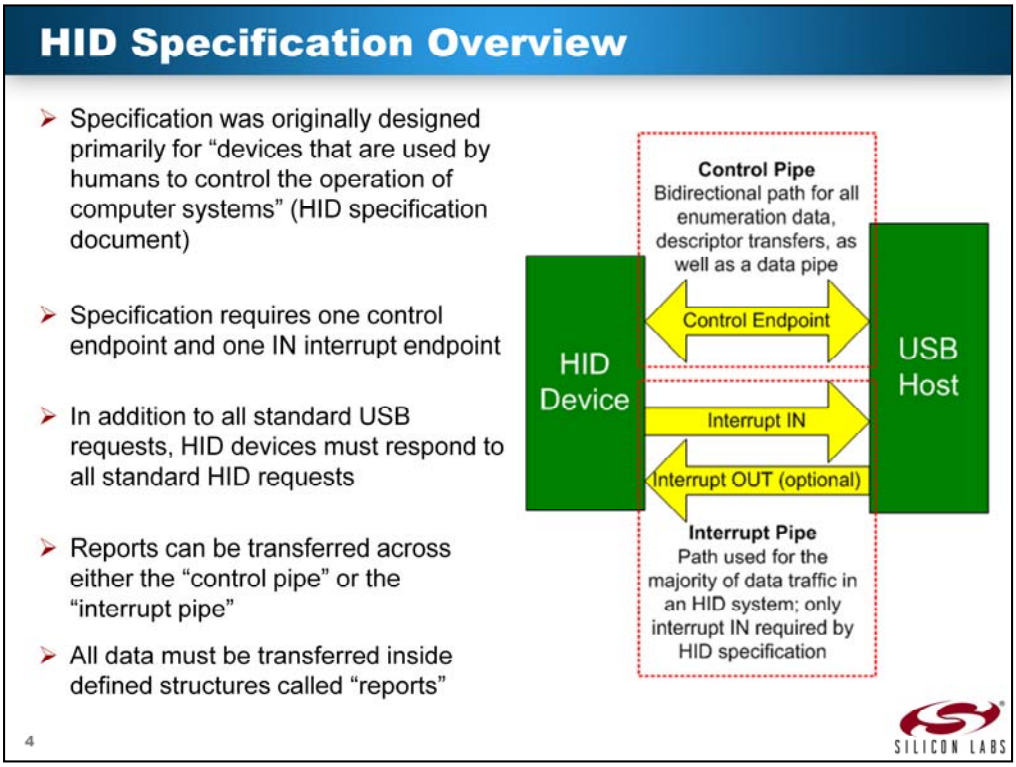
Sample HID Based Application

3

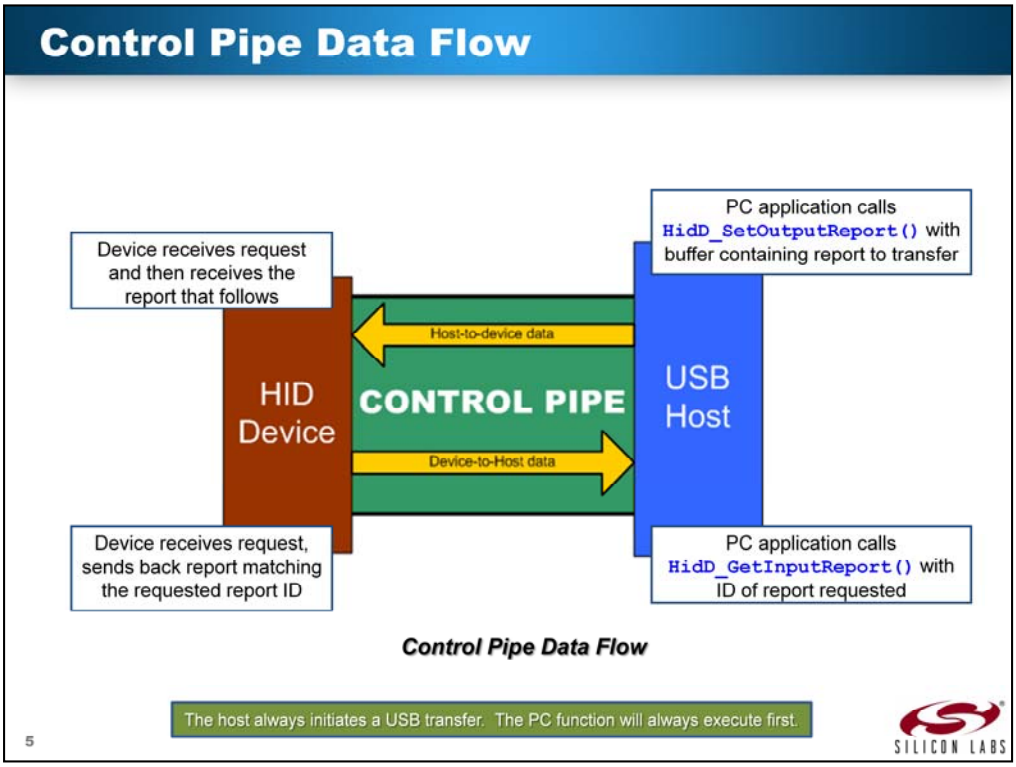
SILICON LABS

Universal Serial Bus (USB) is a communications architecture that gives a personal computer (PC) the ability to interconnect a variety of devices using a simple four wire cable. The USB is actually a two-wire serial communication link that runs at either 1.5 or 12 megabits per second (Mbps) for a full speed device. Higher data transfer rates are available for devices that support 480 Mbps high speed. USB protocols can configure devices at startup or when they are plugged in at run time. These devices are broken into various device classes. Each device class defines the common behavior and protocols for devices that serve similar functions.

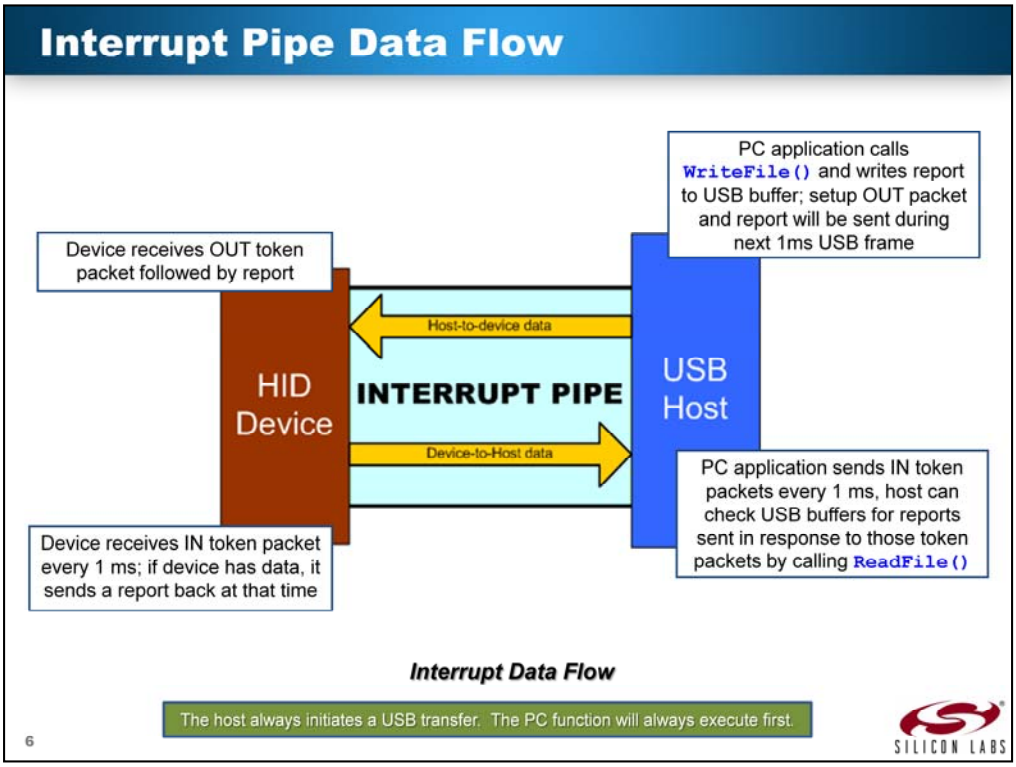
The HID class consists primarily of devices that are used by humans to control the operation of computer systems like a mouse, keyboard or touch screen. HID also provides provisions for output directed to the user.



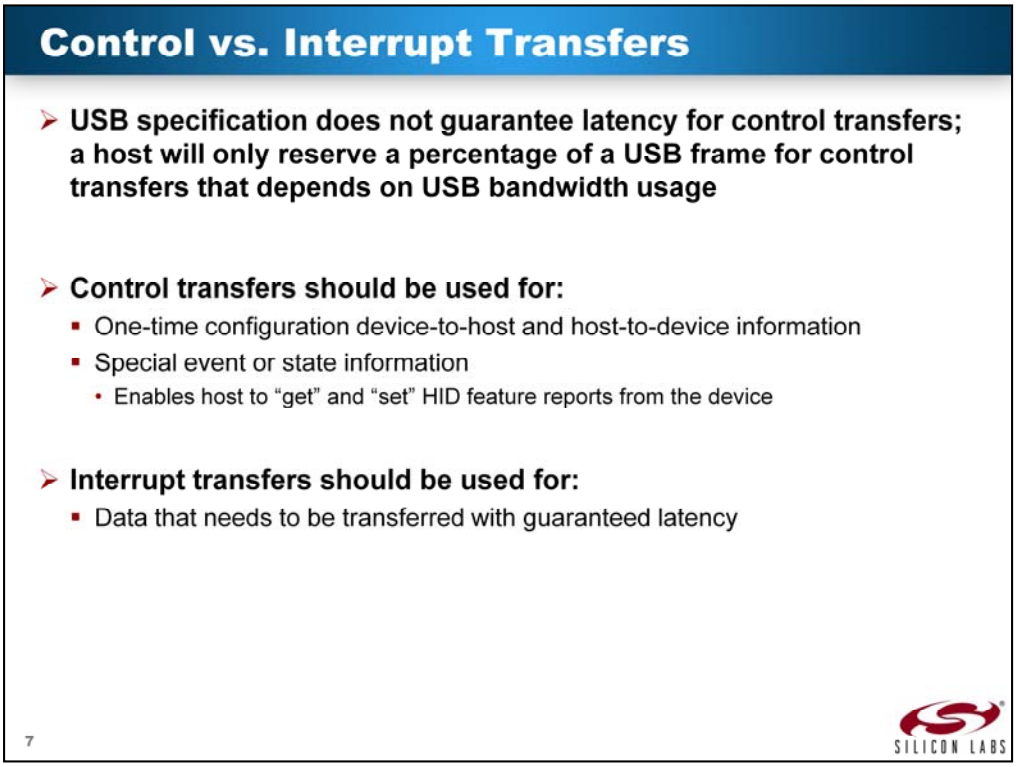
To completely understand the HID class users should be familiar with the USB specification. The USB specification defines the protocol the host controller uses to identify and learn how to communicate with devices on the bus. The USB Core Specification defines the HID class code. The *bInterfaceClass* member of an interface descriptor is always 3 for HID class devices. Using this definition the host knows to ask for additional descriptor information from the device. The specification defines the transfer methods a connected device must support. From the diagram it can be seen that the HID class device is required to support control pipe and interrupt transfers. The control pipe is typically used for enumeration and device configuration whereas the interrupt pipe is used for the data transfer. The devices descriptors tell the host what endpoints and transfer types it supports, which must be the control pipe and at least one interrupt IN endpoint. The USB specification defines the differences between the transfer types and the HID class specifications defines the data structures called "reports." These reports get passed over any of the transfer methods described.



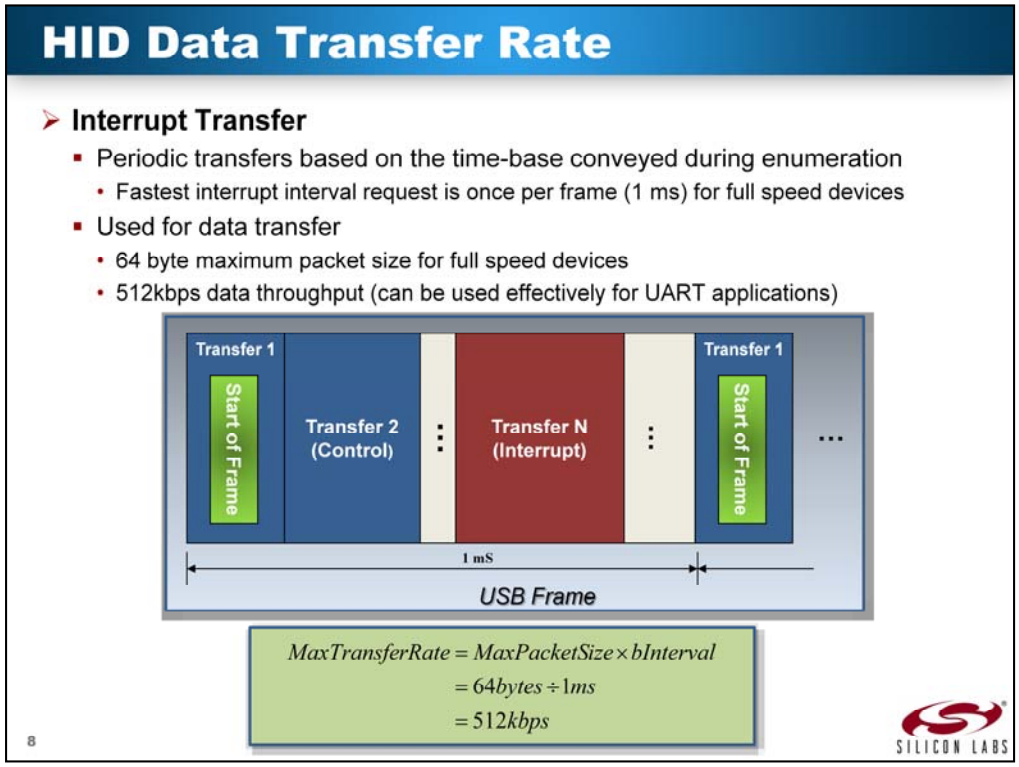
Once enumeration is complete, HID devices can communicate across the control pipe to exchange information stored in reports. This is typically reserved for either configuration information or device status such as application version numbers. The control pipe has a defined transfer sequence defined in the USB specification that has a higher level of error checking than the other transfer methods. In addition, the host API calls to send data via the control pipe are different than those for the interrupt pipe.



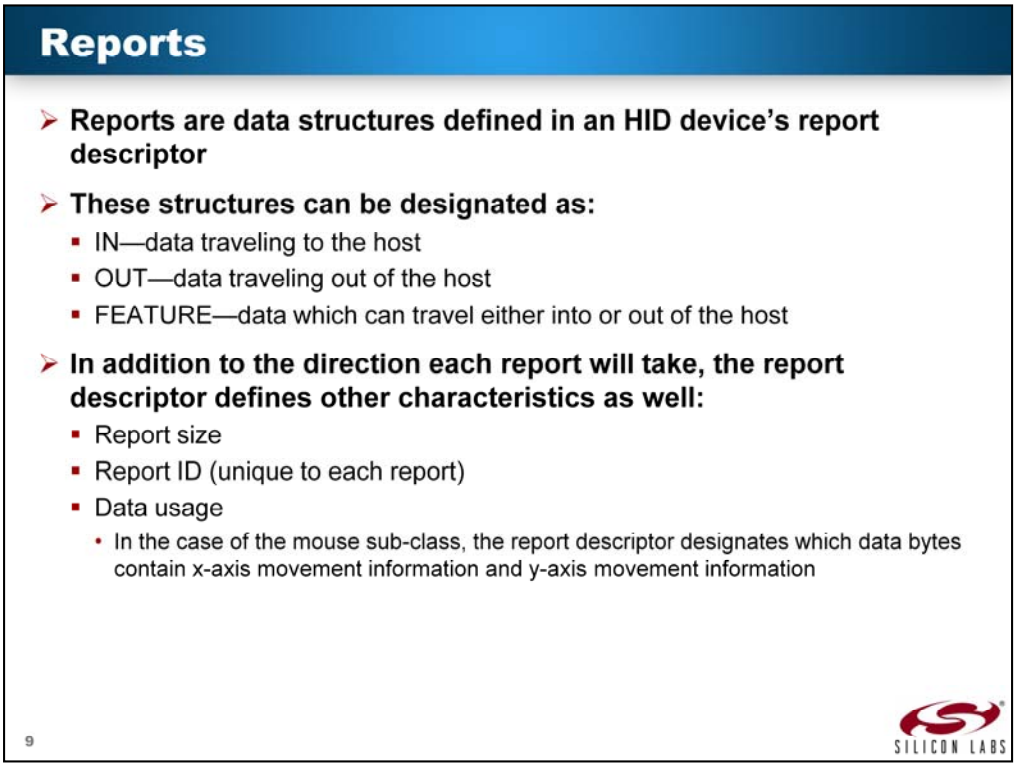
The interrupt transfer method is a periodic data transfer. During enumeration, the device requests how often it wants the host to ask for the data (IN) or send the data (OUT). Once enumeration is complete the host schedules the IN or OUT transfers on the periodic basis. If the device that enumerated was a mouse, for example, it assembles X and Y data into a buffer that gets transmitted to the host when asked. When using touch screens the same principle applies where the X and Y data as well as any required elements get transmitted back to the host when the host asks.



As mentioned, the control transfer and the interrupt transfer are very different and their operation is defined in the USB specification. Control transfers are used for the enumeration process whereby the host finds out how the device wants to communicate and what driver the host needs to be loaded. A dedicated pair of endpoints are allocated to the device enumeration called endpoint 0. In addition, the control transfers are used to pass configuration information and read status information contained in feature reports. The feature report is defined in the HID specification. The control transfer is not typically used for data communications. Data communications happens across the interrupt transfer.



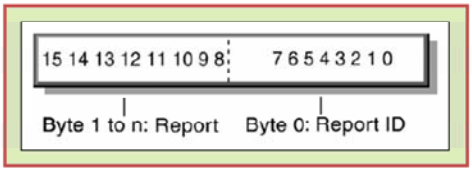
The interrupt transfer is used for periodic transfers where a time period is requested by the device and the host will guarantee that the data transfer will be scheduled within that time period. It does not guarantee that the data is transferred on a consistent time basis all the time, just that it will schedule the transfer prior to the time period expiration. The data throughput is lower for interrupt transfers versus a transfer type like bulk since it schedules a single transfer to the endpoint per frame. This transfer type is useful for applications like mice, keyboards where user input is continually needing to be sent to the host. In addition, the interrupt transfer can provide a timely throughput which can make it useful for UART applications as well. The smallest interval a device can request for the interrupt transfer to occur is 1 frame (1 ms). Since the maximum packet size is for full speed transfers is 64 bytes then the maximum throughput for the interrupt transfer is 64000 bytes/second. This translates to 512000 bits per second. In the UART example, these throughput rates allow HID to UART applications sufficient bandwidth for many systems.



A HID report is used to transmit HID control data to and from a HID Class device. A report descriptor defines the format of a report. Input and output reports specify control data and feature reports specify configuration data. If a device supports more than one report of the same type, each report is assigned a unique report ID. Input and output reports specify HID control data. Input controls are sources of data relevant to an application, for example, X and Y data obtained from a pointing device. Output controls are a sink for application data, for example, an LED that indicates the state of a device. A feature report specifies configuration information for a device. A user-mode application can obtain and set feature information by using this report designation.

Report Format

- **The report format is composed of an 8-bit report identifier followed by the data belonging to the report**
  - Amount of data for each report is conveyed during enumeration using the report descriptors (report size and report count)
- **Report ID**
  - The report ID is used by the firmware to determine what data has been received or what data needs to be transmitted
  - The report ID field is 8 bits in length
- **Report data**
  - The data fields are variable-length fields that report the state of an item



Report Format

10

SILICON LABS

Part of the HID enumeration requires the device to send an HID descriptor. This descriptor contains information about all of the data and its formats. The report format is determined by the report descriptors sent to he host. All reports are preceded by a report ID. The report ID is how the host and the device identify the particular report and are able to parse the data between multiple reports. Each bit in the report is specified in the report descriptor using a report size and report count identifier. For example, a report size of one and a report count of two would identify two bits. This could be the definition of a set of mouse buttons. To send a complete byte the report size would be set to eight.

Example Report Descriptor

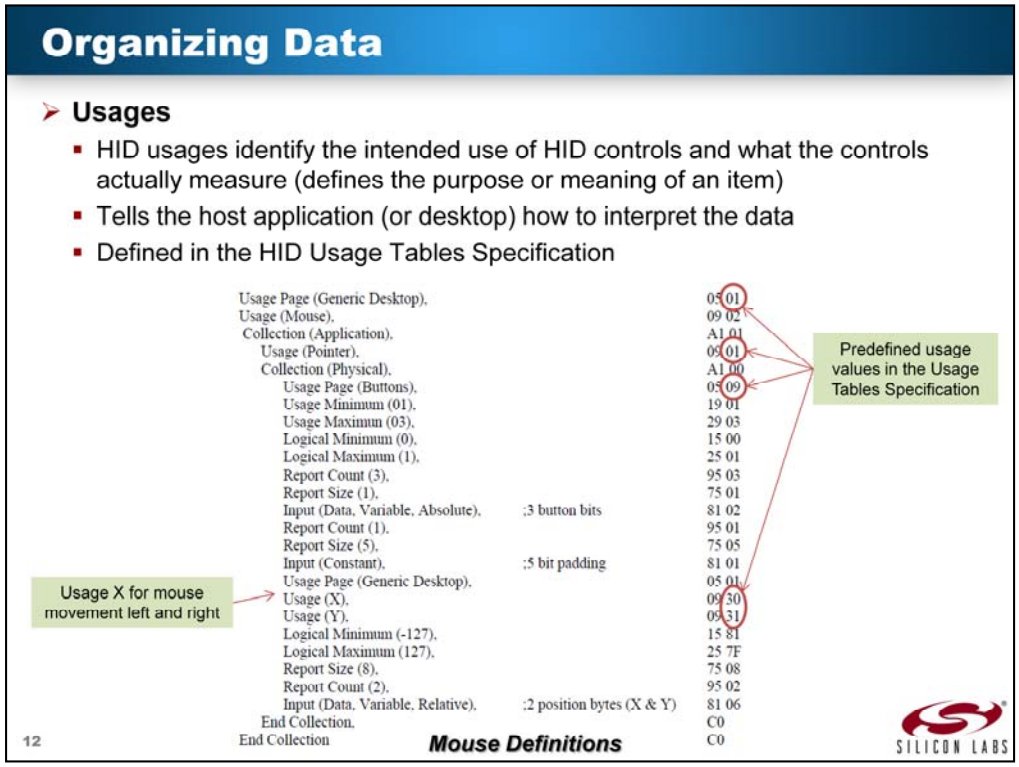
- **IN_BLINK_SELECTOR Example:**
  - Report count is the number of elements transferred
  - Report size is the number of bits for each element
    - Assume report count = 3
    - Report size = 8 bits
    - IN_BLINK_SELECTOR report contains 3 bytes of data

- **Mouse button example**
  - Report count = 2
  - Report size = 1 bit
  - The mouse report contains 2 bits of data

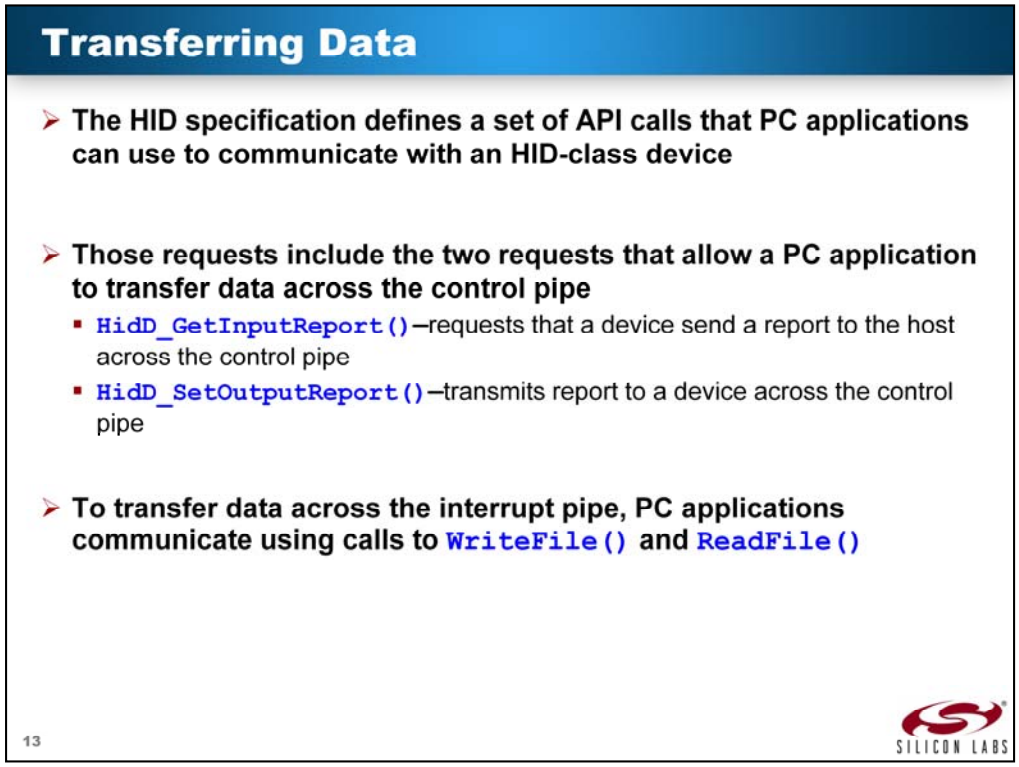
```
code const hid_report_descriptor HIDREPORTDESC *
{
    0x06, 0x00, 0xff,              // USAGE_PAGE (Vendor Defined Page 1)
    0x09, 0x01,                    // USAGE (Vendor Usage 1)
    0xa1, 0x01,                    // COLLECTION (Application)

    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)

    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
};
```

Sample HID Descriptor

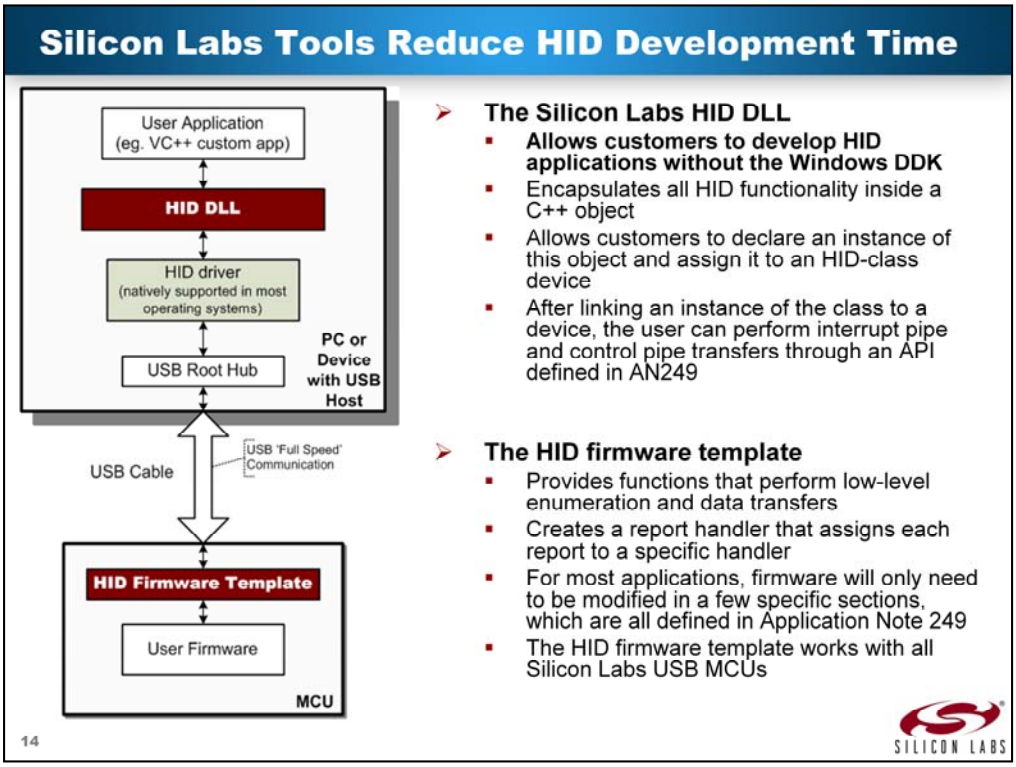
11



Formatting the data into reports helps to organize the data in terms of data widths so that the host and end devices know how to parse the data. Moving up a level the HID specification defines usages. These are predefined values that tell the host what to do with the data. For example, one usage defined for a mouse is usage(X). When the host receives a descriptor that has the report ID and the associated data tied to the X usage then it knows how to handle that data. For a touch screen the same usage exists since we are trying to get the same data. Some additional usages exist for touch screens like tip switch which is a one bit value. There are many usages for HID devices and their definitions can be found in the HID Usage Tables Specification document.



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). HidD_SetOutputReport() and HidD_GetInputReport() allow host applications to send and receive IN and OUT reports over the control pipe. During enumeration, host system software learns about the interface of the attached USB device. After reception of the 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(). 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 a buffer that contains the report to be transmitted.

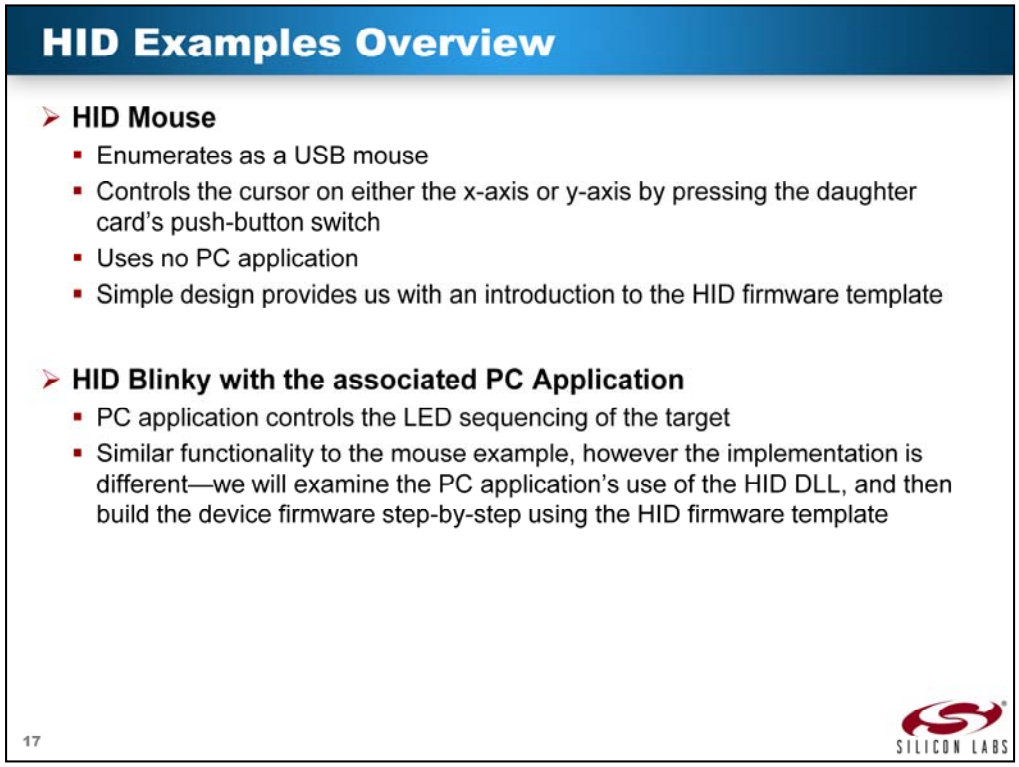


Since HID alleviates the need to write any host side drivers and doesn't require an install process it is an attractive means for implementing USB connectivity. When using the Silicon Labs library for the HID class API and the HID class firmware template for the USB MCU, designers can reduce the burden of implementing USB device and host applications.

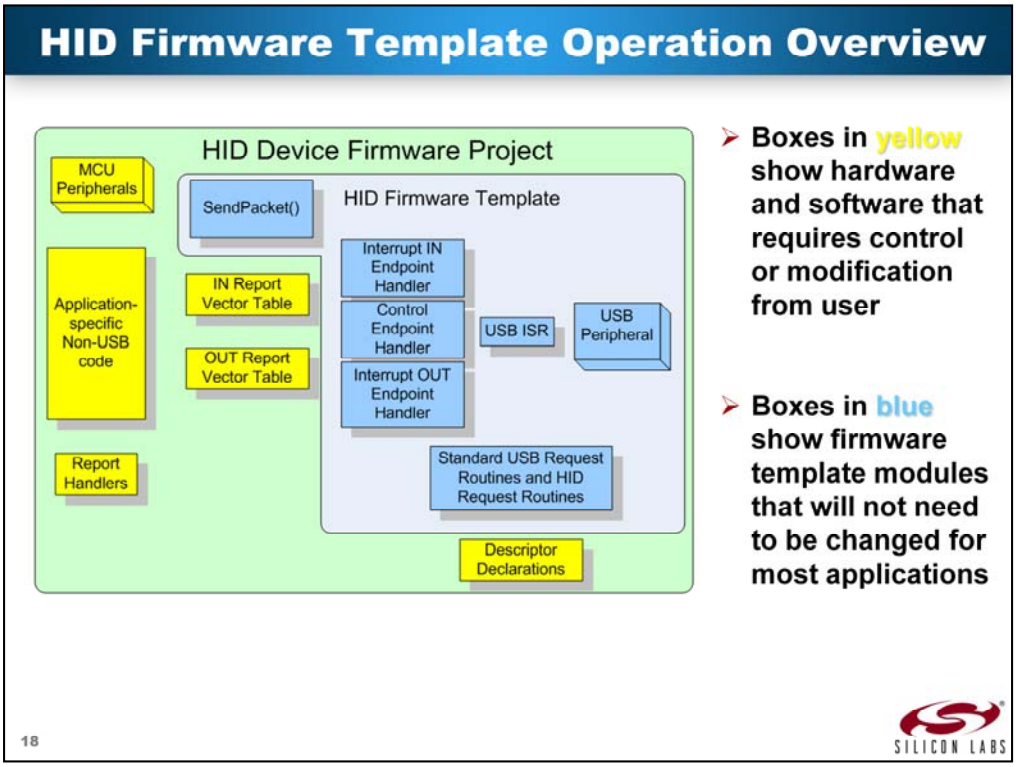
HID Summary

- **It's easier for end customers to use**
  - Designing with HID can be more challenging than designing USB using other methods, but the end customer doesn't have to install a driver
- **The HID specification gives designs lots of flexibility**
  - Customizable data structures and a standardized API make HID suitable for many USB-based applications
- **Silicon Labs tools make development easy**
  - The HID DLL encapsulates the HID API into a simpler C++ class object
  - The HID firmware template gives customers a nearly-complete firmware solution, with clearly defined areas in code that need to be customized for each application's unique requirements

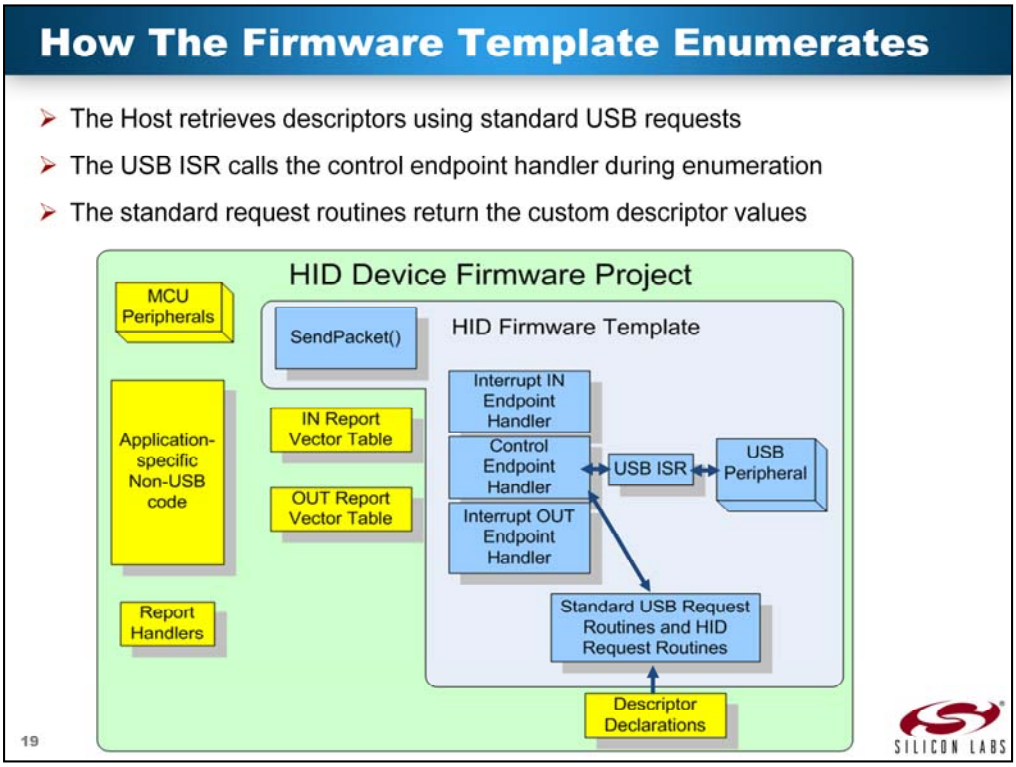

SILICON LABS
www.silabs.com
HID Examples



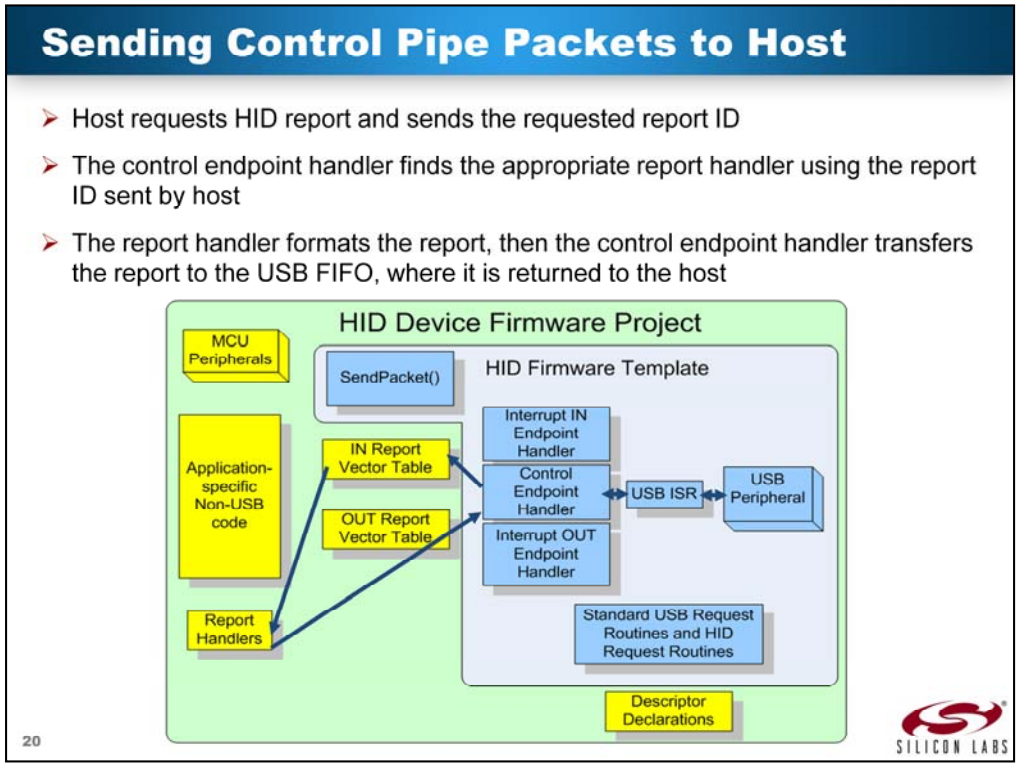
In this presentation we will take a look at two example applications. The first is a standard mouse implementation that requires no host side software. In this case the device enumerates as a mouse and passes x and y data. In many applications the device needs to send data that doesn't fit the standard HID implementation. An example would be a device that needs to send voltage, temperature or any other information that is not necessarily used by the PC for human interface. The second example uses the template code to implement the firmware as well as shows the steps required for the host application to read the data via the HID DLL.



The Silicon Labs firmware template contains the base firmware for USB HID communications. All of the low level handler functions read and write data across the USB and require no modifications (for most applications). The user application code is responsible for generating the report descriptors based on the application needs as well as handlers for those reports., which are already available as a stub function. By using the firmware template the integration of the USB into an end product is greatly simplified.



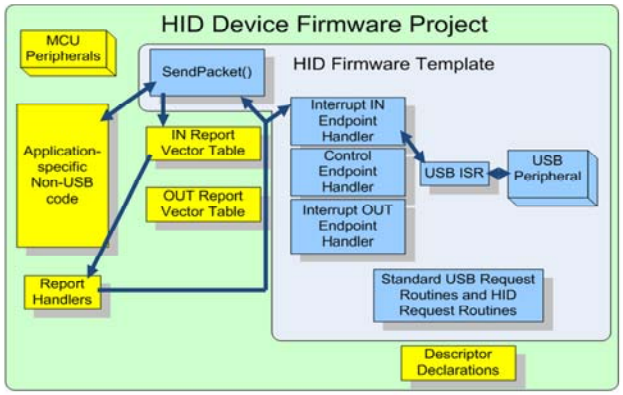
During enumeration the host asks the device about its capabilities. This process is covered by the USB specification as well as additional training modules. During development, designers define the data format for the application and determines how the reports will be structured with respect to the size (number of bits) and count. From there they can generate the report descriptors and store them in the flash of the MCU. When the device is plugged into the host it will identify itself as a HID class device. At that point the host will ask for the HID class descriptors and all of the lower level, pre-written firmware handles passing the descriptors to the host.



Once enumeration is complete, the host can then begin sending data. As discussed previously there are two ways to transfer data: control and interrupt transfers. Here is a flow diagram of the control transfer. The host initiates the request to the control endpoint. The endpoint handler calls the appropriate report handler which formats the data and writes it to the USB FIFO so that it is ready for when the host initiates the data phase of the control transfer.

Sending Interrupt Pipe Packets to Host

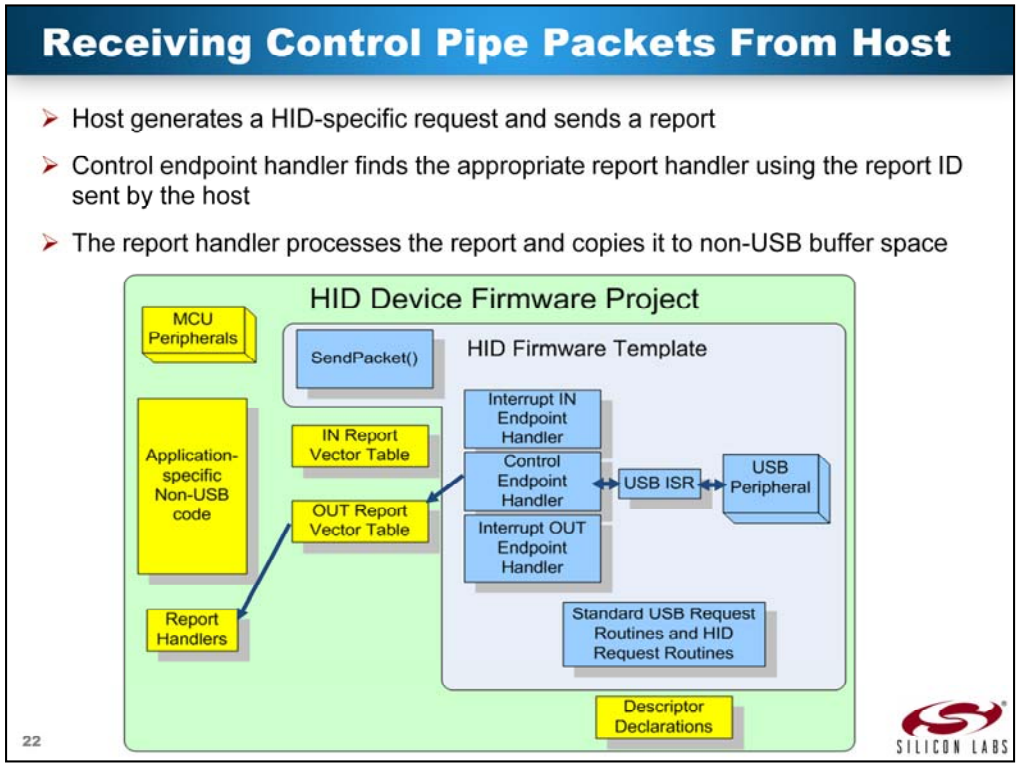
- App calls `SendPacket()` and passes in the appropriate report ID
- `SendPacket()` calls the appropriate report handler to format the IN report
- `SendPacket()` then writes to the USB buffer and signals the hardware to transmit when it receives the next IN token packet
- After hardware receives an IN token packet during the 1 ms USB frame
  - Transmits contents of the buffer
  - Signals when transfer is complete



21

SILICON LABS

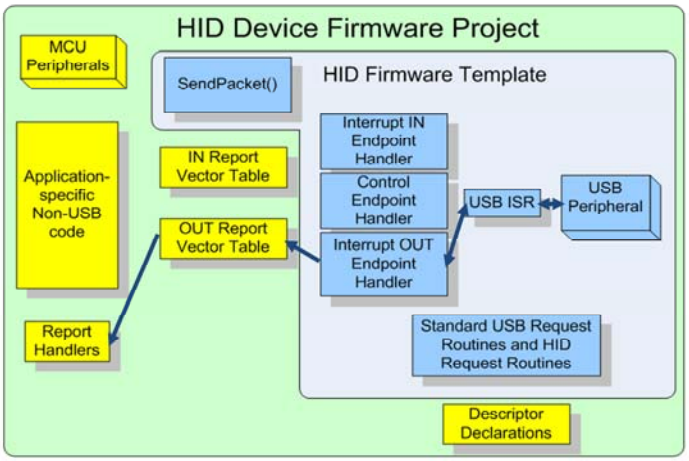
The second method to transfer data to the host is via the interrupt transfer. These transfers are scheduled by the host on an interval determined during enumeration. The data flow diagram above starts with a function in the firmware template called SendPacket which includes the report ID for the handler functions. The low level handler functions retrieve the data and place it into the USB buffers so that it is ready for when the host sends a data token for an IN transfer.



Receiving a packet from the host (OUT) via a control transfer is handled via the endpoint 0 handler. The host generates the request and the device endpoint 0 handler determines the report ID and calls the appropriate report handler that stores the data for use by the application.

Receiving Interrupt Pipe Packets From Host

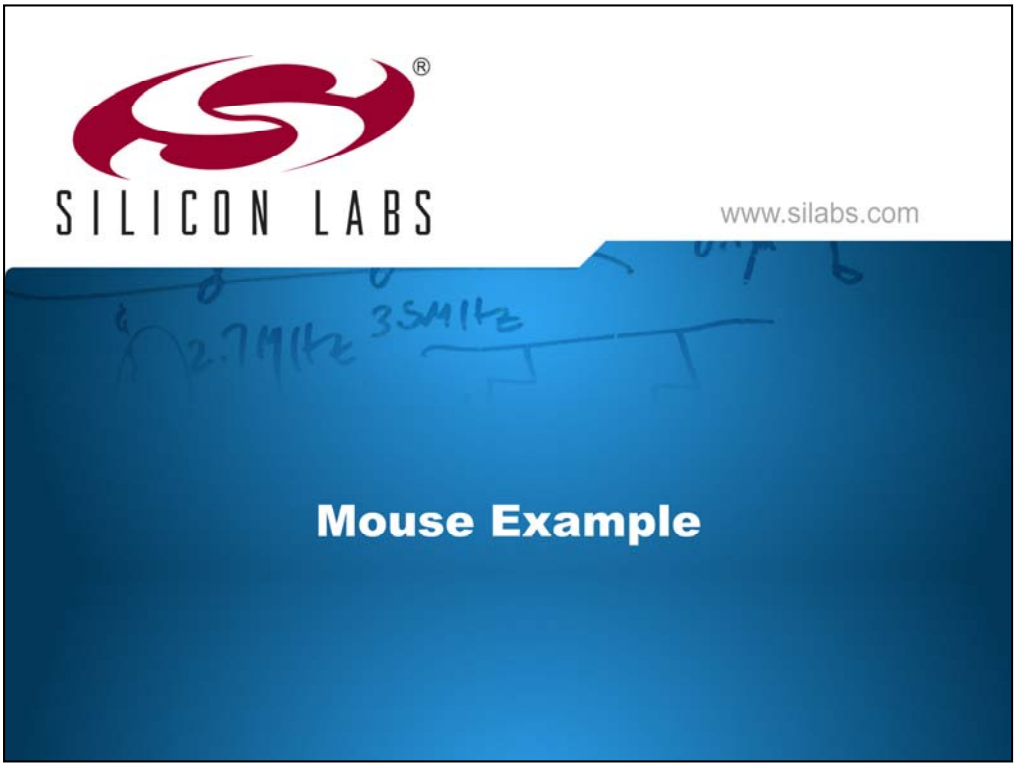
- Device receives OUT token packet and the report sent by host
- Interrupt OUT endpoint handler finds the appropriate report handler using the report ID sent by the host
- The report handler processes the received report and copies it to non-USB data space



23

SILICON LABS

Similar to sending the data to the host via interrupt transfers the device receives data from the host via scheduled data transfers (OUT token). Via the endpoint handler the device finds the appropriate report handler to store the data to memory for use by the application.

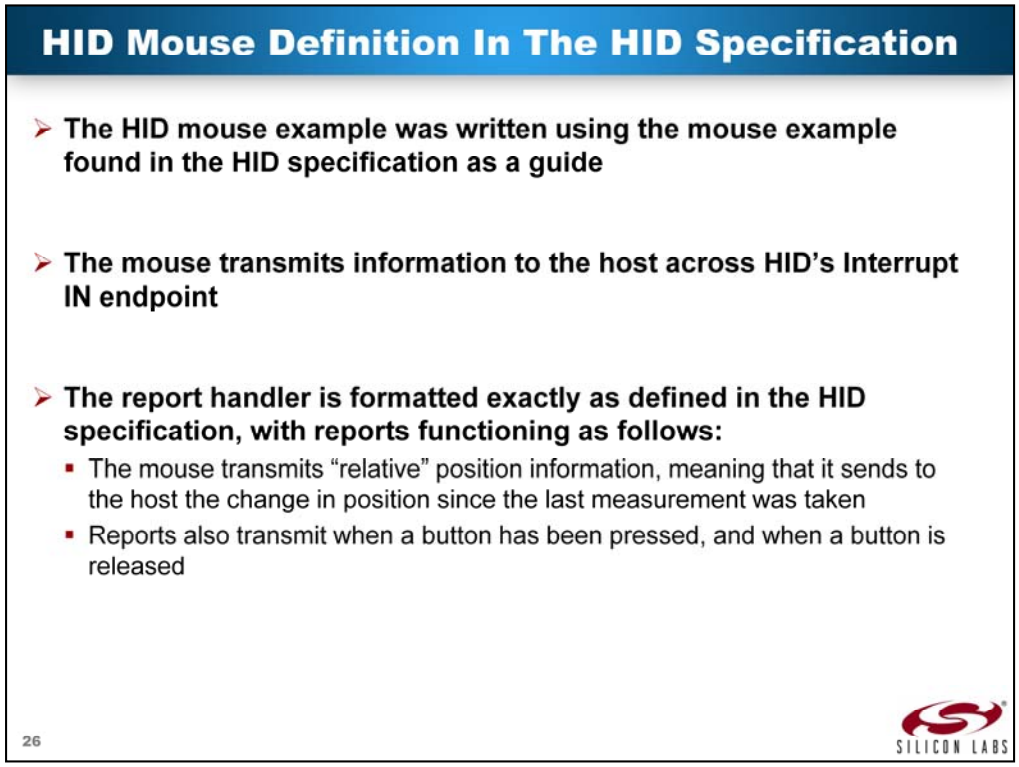

SILICON LABS
www.silabs.com
Mouse Example

HID Mouse Example Goals

- **Familiarize ourselves with the HID firmware template, with emphasis on:**
  - File structure
  - Report handler functionality
  - Descriptor definitions

images from wikipedia.org

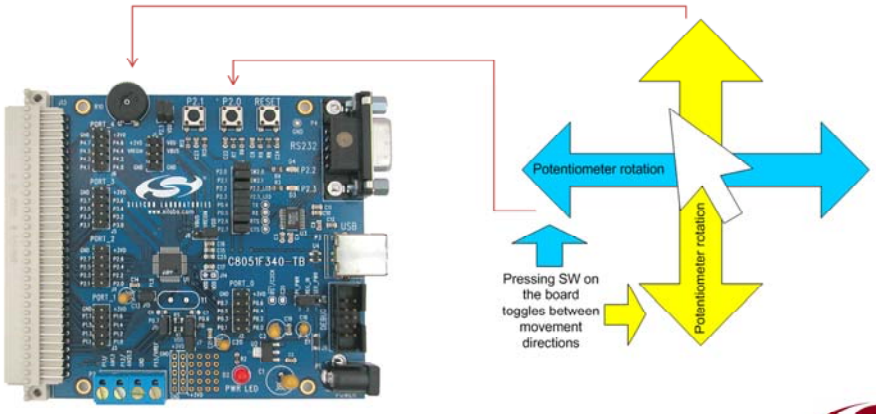
SILICON LABS



The HID class was developed originally as a means to provide human interface input to the PC. The mouse and keyboard were primary targets for generating the specification and it defined a way to process the data for keyboard keys, mouse buttons and position. The specification defines the data transfer for mice and keyboards to use the interrupt IN endpoint. Included in the specification were examples for the keyboard and the mouse and the example used here is derived from those in the specification. This example uses the firmware template to mimic mouse movement using development kit hardware.

How To Operate The Mouse

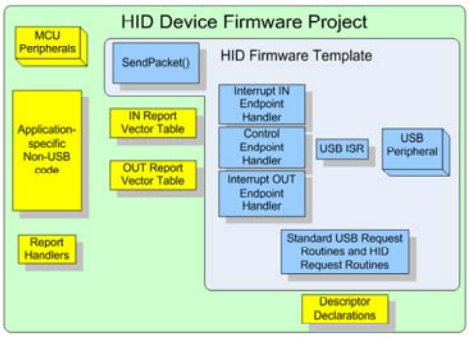
- The firmware will move the mouse along either the x-axis of the screen or the y-axis of the screen when the user rotates the potentiometer from its center position
- The user toggles between x-axis or y-axis movement by pressing the board's push-button switch



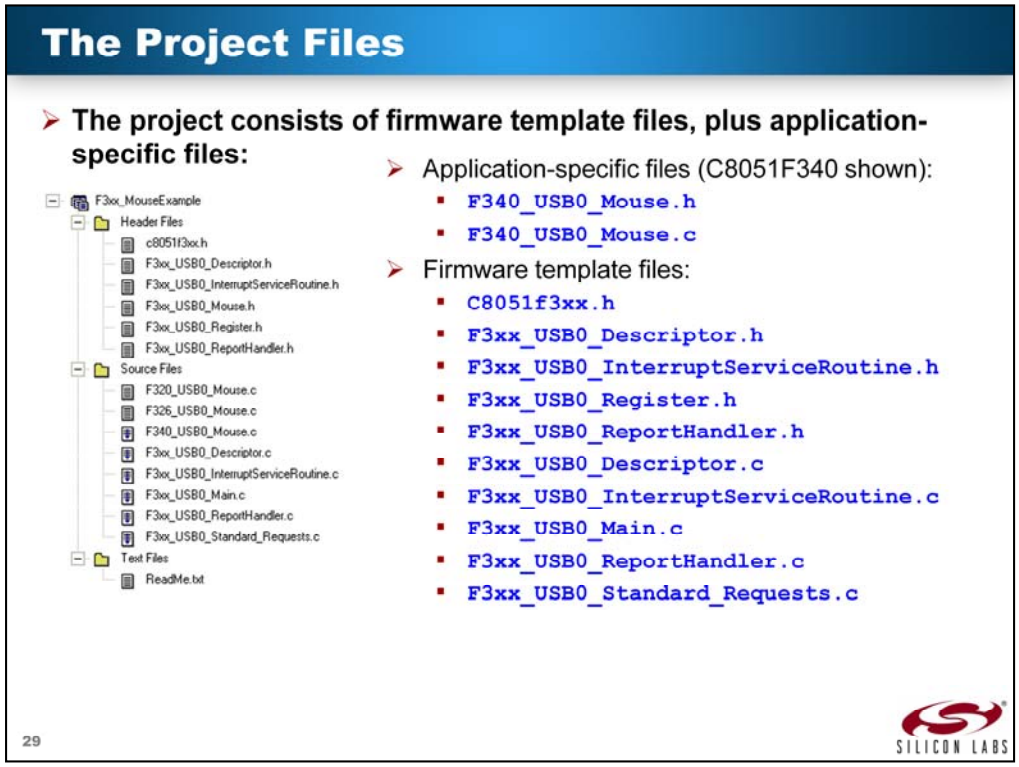
SILICON LABS

The Project's Files

- **In the next set of slides, we will look at each of the project's files and discuss the following:**
  - What each of the HID firmware template files do
  - How each application-specific file works
  - Where in the HID firmware template custom code was added in order to create an HID mouse
- **We will be highlighting sections of the HID firmware template drawing below:**



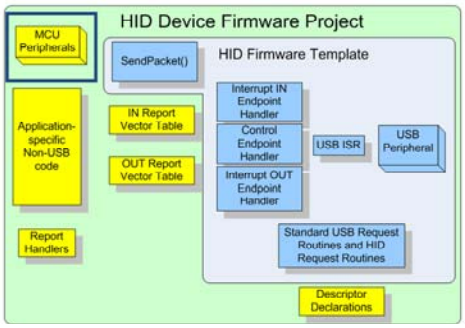
SILICON LABS



The picture on the left is a snapshot of the project files in the examples directory of the install for the IDE. C:\Silabs\MCU\Examples\C8051F34x\USB_HID is the directory where the example files can be found. In the next set of slides, we will discuss application-specific files, what each of the firmware template files do in a system, and where custom code was added to create an HID mouse.

F340_USB0_Mouse.c

- **Contains all code specific to this application that is not directly USB-related**
- **The example uses the following peripherals:**
  - ADC0—samples the potentiometer position
  - Timer 2—provides a start-of-conversion source for the ADC
- **The file contains initialization code for the above peripherals, along with port pin and system clock initialization**



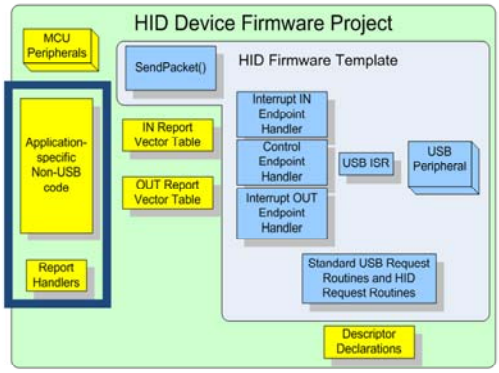
30

SILICON LABS

The F340_Mouse.c file contains all of the initialization routines for the MCU in order to operate as a HID class device. For example, this file contains the initialization routines for the USB peripheral and the oscillators such that it can provide a 48 MHz clock to the USB module. The application specific initialization also happens in this file such as the ADC configuration and interrupt service routines (ISR). These are used to generate the relative change for the X and Y position.

F3xx_USB0_Mouse.h

- Contains function prototypes and external variable definitions
- Many of the variables declared in F3xx_USB0_Mouse.c are used in F3xx_USB0_Report_Handler.c
- This header file allows F3xx_Report_Handler.c to access those variables



31

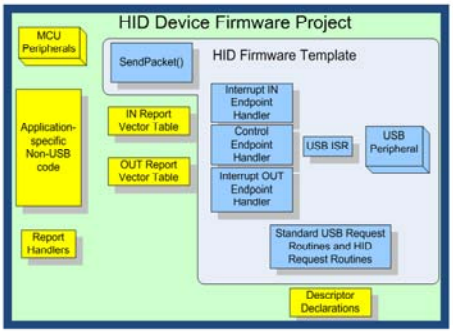
SILICON LABS

Included in this file are function prototypes as well as some variable declarations. An example variable declaration is the IN_PACKET array.

C8051F3xx.h Header File

- Allows all other files in the firmware template to be MCU-unspecific; each file just includes `C8051F3xx.h`
- User should add a #include for one of the Silicon Laboratories USB MCUs (`C8051F320.h`, `C8051F326.h`, `C8051F340.h`)
- Customization for the HID Mouse example
  - Open the `C8051F3xx.h` header file
  - This project includes the F340's header file, as shown below:

```
#include <c8051f340.h>
```



32

SILICON LABS

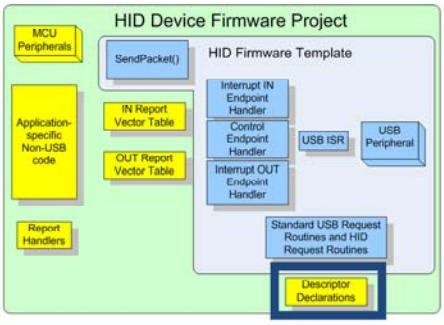
This file includes the specific USB MCU header file. The example shown above highlights the C8051F340 header file being added, however, if we targeted the C8051F320 we could modify this file to reflect that.

F3xx_USB0_Descriptor.h

- Contains definitions for descriptors, along with other definitions used by the HID firmware template
- The definition for the HID report descriptor size will need to be modified to show the size of the application's report descriptor, which is defined in `F3xx_USB0_Descriptor.h`
- Customization for the HID mouse example:

1. Scroll down until you find the #defines for `HID_REPORT_DESCRIPTOR_SIZE`:

```
#define HID_REPORT_DESCRIPTOR_SIZE 0x0032
#define HID_REPORT_DESCRIPTOR_SIZE_LE 0x3200
```



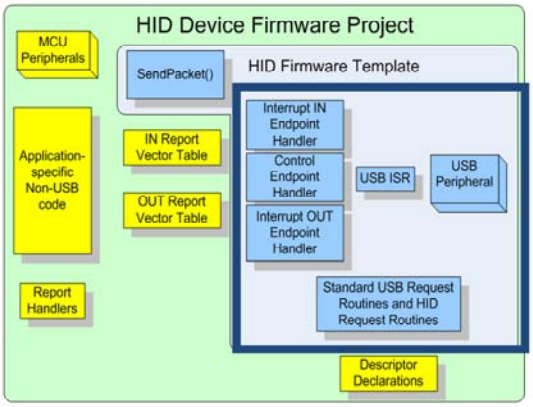
33

SILICON LABS

The F3xx_USB_Descriptor.h file contains all of the type definitions for the structures used for the descriptors stored in flash as well as the format for the setup packet used for the control transfers. One of the fields that would need to be modified is the HID_REPORT_DESCRIPTOR_SIZE parameter to match what the value required by the mouse example.

Header Files for USB Activity

- Contains definitions and function prototypes for low level USB module operation and other USB standard activity
  - F3xx_USB0_InterruptServiceRoutine.h, F3xx_USB0_Register.h, and F3xx_USB0_ReportHandler.h
- For most projects, these files will not need to be modified



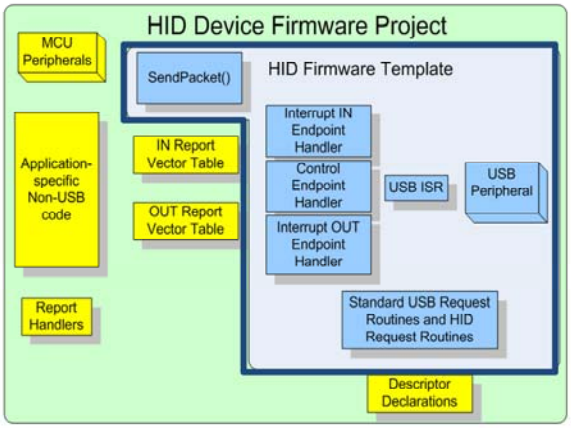
34

SILICON LABS

The header files listed are part of the lower level firmware template and typically requires no modification. These files support the low level communications with the USB. The register header file provides all of the addressing values required by the indirect access of the USB peripheral. The F3xx_USB0_InterruptServiceRoutine.h header file for the ISR provides a lot of the definitions required for the control and endpoint handlers. Many of these values are used for comparison with the received data in order to direct code execution to the proper functions. For example, GET_DESCRIPTOR is defined as a value of 0x06. This is used for comparison when the setup packet is received so the firmware can prepare a descriptor to be sent to the host.

C Files for USB Activity

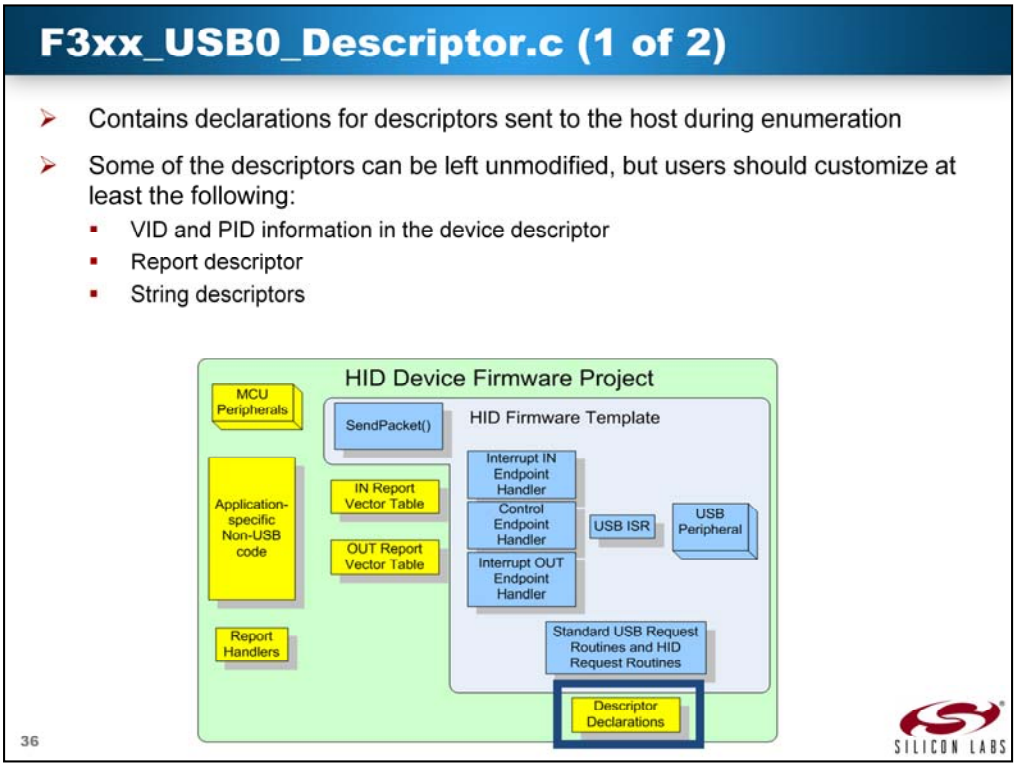
- Contains functions for handling HID traffic across both the control pipe and interrupt pipe, including the USB interrupt service routine
  - F3xx_USB0_InterruptServiceRoutine.c
  - F3xx_USB0_StandardRequests.c
- For most applications, these files do not need to be modified



35

SILICON LABS

The C files are used to carry out the low level communications with the USB. The ISR file contains all of the code that is required when the USB peripheral is ready to send or receive data. The standard requests file is used once the ISR has determined what action is required. An example function found in the F3xx_USB0_StandardRequests.c file is Set_Address which writes to the USB address register when the host assigns a new address to the device as part of enumeration.



The Fxx_USB0_Descriptor.c file takes the structure definitions found in F3xx_USB0_Descriptor.h file and declares the values for them. The specific values required for the device to enumerate properly are defined in this file. For example, the VID and PID are assigned values in the *const device descriptor code DEVICEDESC* declaration. Notice the const usage. This places these values in the non-volatile memory of the MCU. For HID devices, this file includes the HID descriptors for the application. This file needs to be modified to support the data needs of the system.

F3xx_USB0_Descriptor.c (2 of 2)

- Customization for the HID mouse example:
  1. In the device descriptor, the example is given a unique VID and PID:

```
0xC410,   // idVendor
0xB981,   // idProduct
```

Note that the VID and PID are stored in "little endian" format, with the LSB before the MSB; this is due to USB specification

  2. Because the MCU needs to be recognized as a HID Mouse, the interface descriptor needs to be modified as a HID subclass device

```
0x01, // bInterfaceSubClass
0x02, // bInterfaceProcotol
```

The Interface Descriptor tells the host that the device uses an HID interface

Setting `InterfaceSubclass` to 1 tells the host that the device is an HID mouse

Setting `InterfaceProtocol` to 2 tells the host that the device is bootable

  4. The Report Descriptor is defined exactly as shown in the Mouse section of the HID specification document

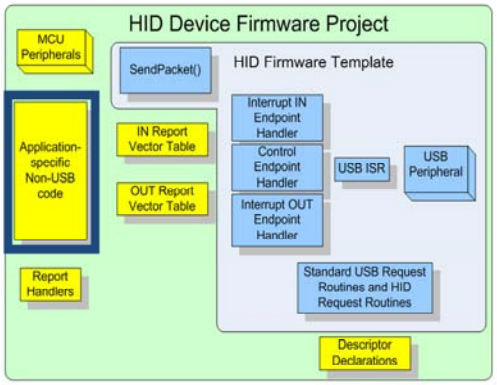
SILICON LABS

F3xx_USB0_Main.c

- Initializes the MCU, enables USB, and then enters a `while(1)` loop that continuously calls `SendPacket()`, which transmits a report containing mouse position information to the host
- This file will always need to be customized, especially the `main()` routine
- Customization for the HID Mouse example:

1. The main loop of the system simply transmits the only defined report in the report handler as fast as possible:

```
while (1)
{
   SendPacket (ReportID);
}
```



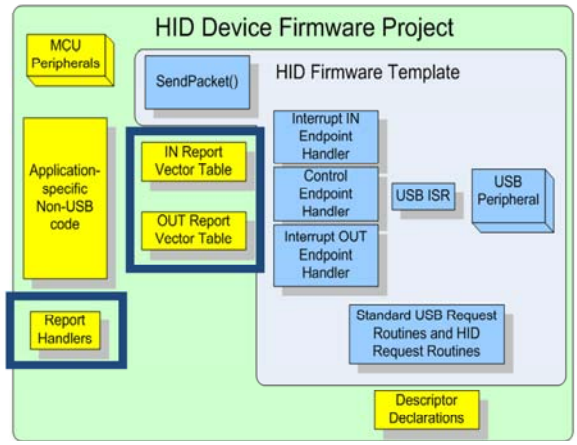
38

SILICON LABS

The main.c file requires modification to add application specific code. In this example, you can see that the main routine is calling a function SendPacket(Report ID) that prepares data to be sent to the host via IN transfers.

F3xx_USB0_ReportHandler.c (1 of 3)

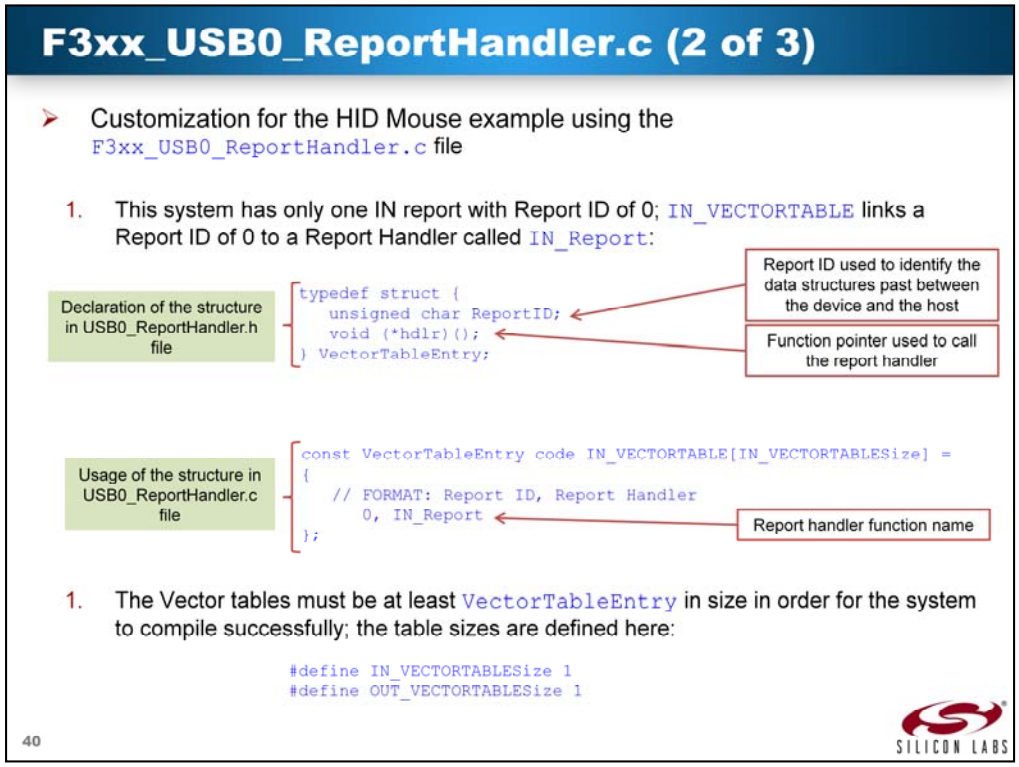
- Contains report vector tables which map each defined report ID to a corresponding report handler, also contains all of the report handlers
- This file will always need to be customized for an HID device.



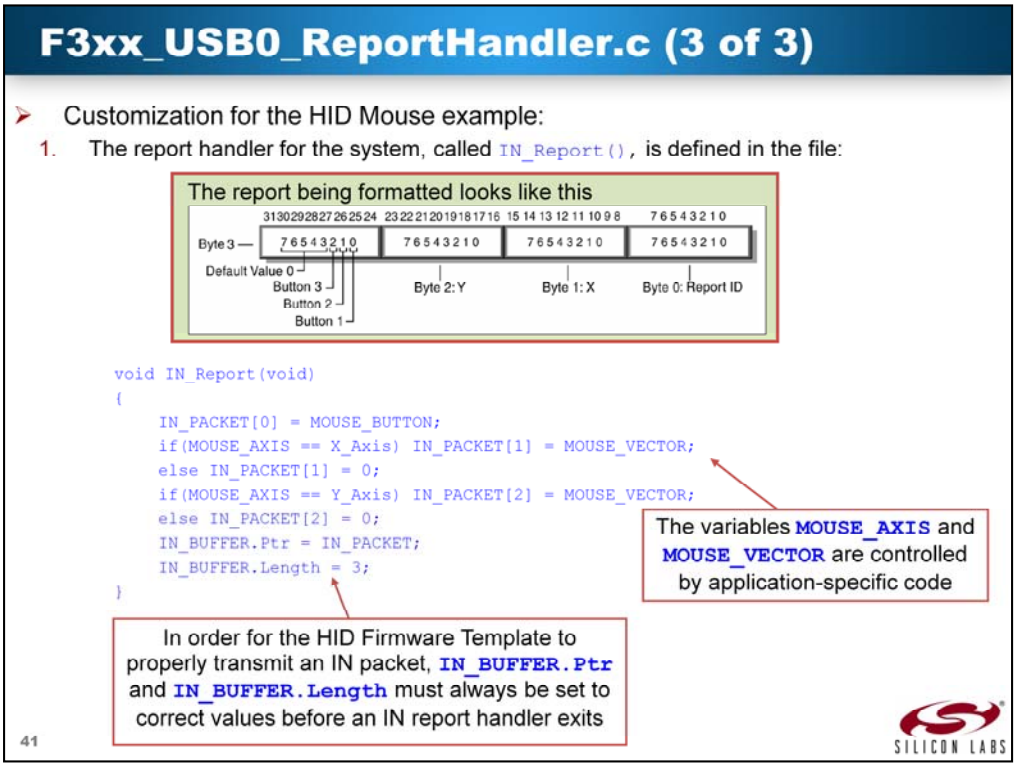
39

SILICON LABS

Another file that requires modification by the user contains the report handlers. These routines carry out functions based on the specific report being addressed. Typically, the device receives the request from the host for a specific report ID and the report handler functions call the specific routine to move the correct data to the buffers.



Here we see the progression of the report handlers and how they are declared in the firmware. The first snippet of code shows the type definition of the structure used for vectoring to the associated handler routine. The report ID and the function pointer are defined as part of this structure. In this example there was only one INPUT report type, therefore, the ReportID was set to 0 as shown. The input handler function name is defined to be IN_Report as shown as well.



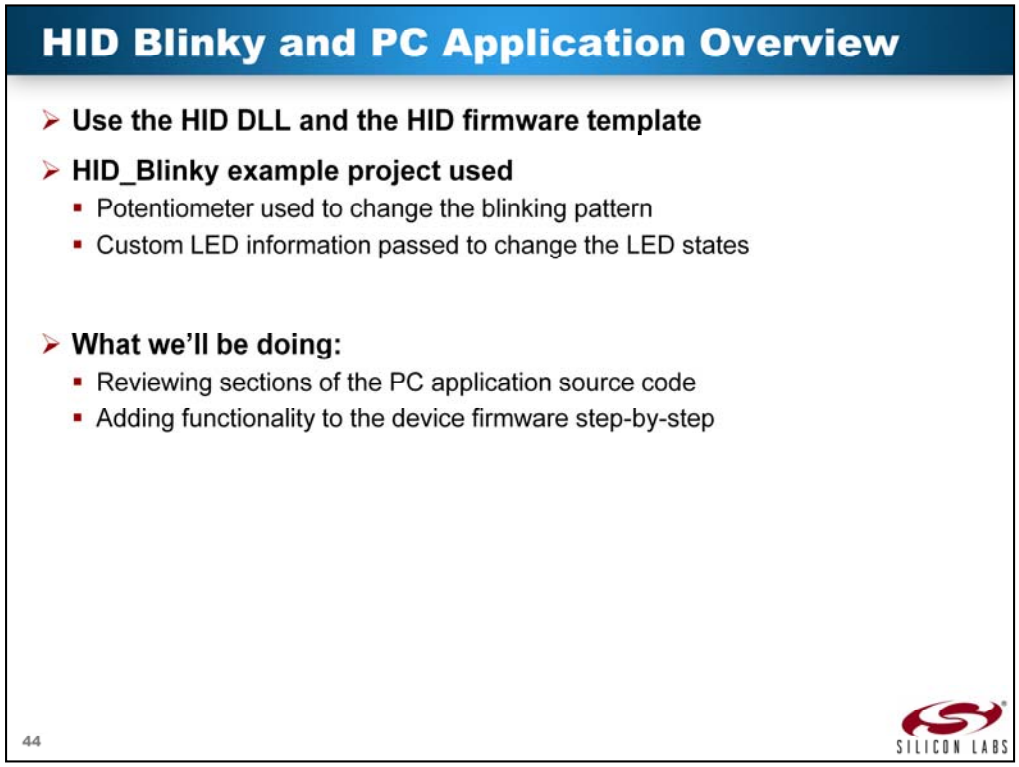
The data for the mouse movements must be stored in a specific order as determined during enumeration when the host asked for the HID descriptors. The diagram above shows the format of a mouse descriptor from the HID specification. The first byte is the report ID with the X and Y data following. The last byte contains the button information. On the development board we need to place the ADC results into the X and Y bytes and the push-buttons into the associated button locations. When the device gets the request to send the report for the data via the interrupt transfer it calls the IN_Report handler function. This function loads the data into the allocated buffer that gets loaded into the USB FIFO to be sent back to the host. In the function the mouse vector information is placed into the buffer. In addition, function is responsible for setting the length of the data packet is set along with the pointer to the correct buffer.

HID Mouse Example Summary

- Example uses the HID firmware template and follows the example shown in the HID specification
- Only a few modifications are needed to turn the firmware template into an HID Mouse
- Now we're prepared to look at a more challenging HID system, with both firmware and software components
- In the next example, we will build the firmware step-by-step, using what we've learned in the HID mouse example

SILICON LABS

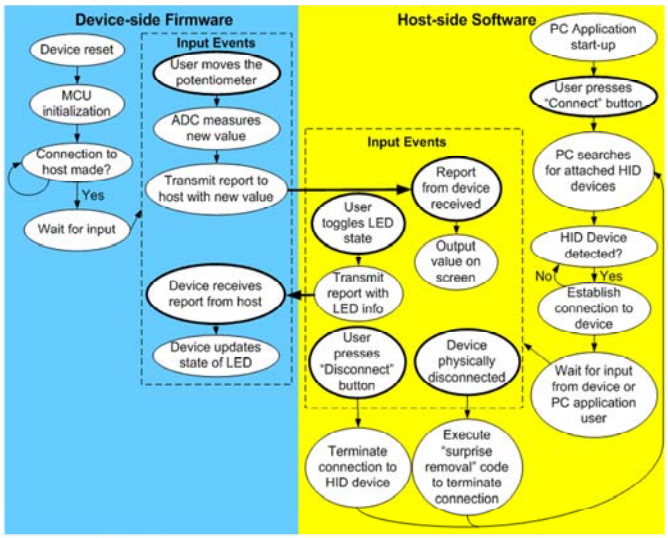

SILICON LABS
www.silabs.com
HID Blinky Application Example



In the previous example we looked at the firmware side of a HID device. Implementing a mouse requires no host side software since the operating system already understands how to interpret the data presented by the device. This was accomplished via the usages defined in the HID descriptor table. The next example highlights a non-standard HID application. The implementation is not for a human interface component to the PC, but instead is an application that passes data for controlling LEDs. This is representative of embedded devices that need to get data to the host without requiring a driver install.

Typical Data Flow Diagram

- Because HID requires all device-host transfers be formatted in reports, the arrows shown transferring data between device and host must be defined as reports

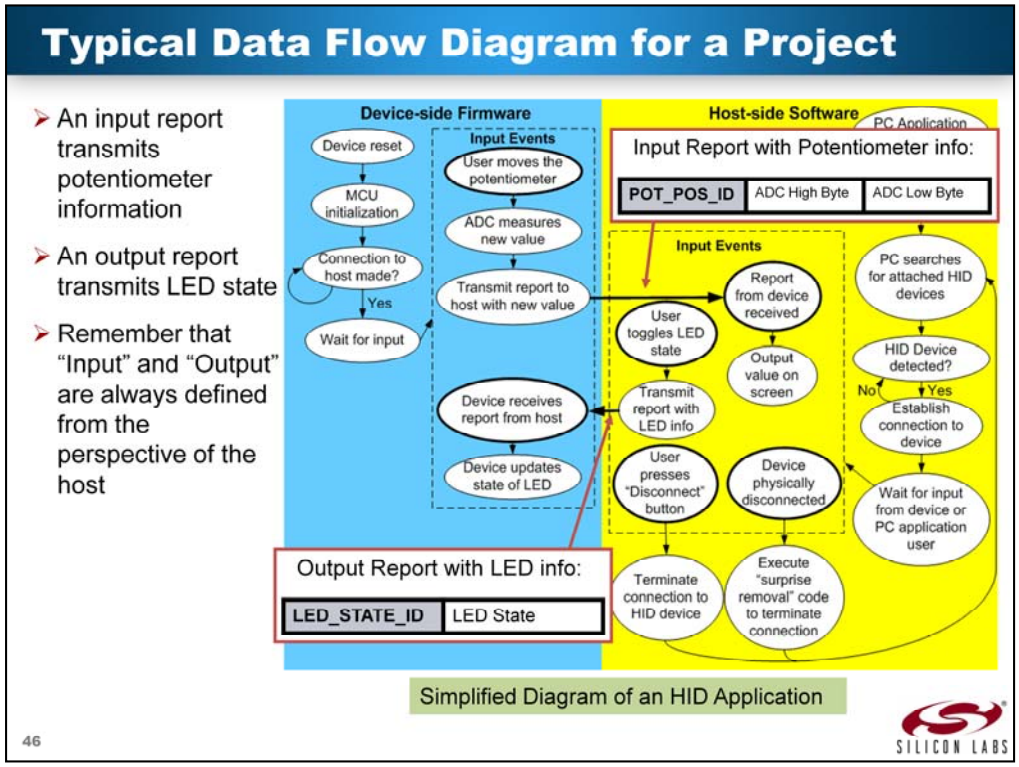


Simplified Diagram of an HID Application

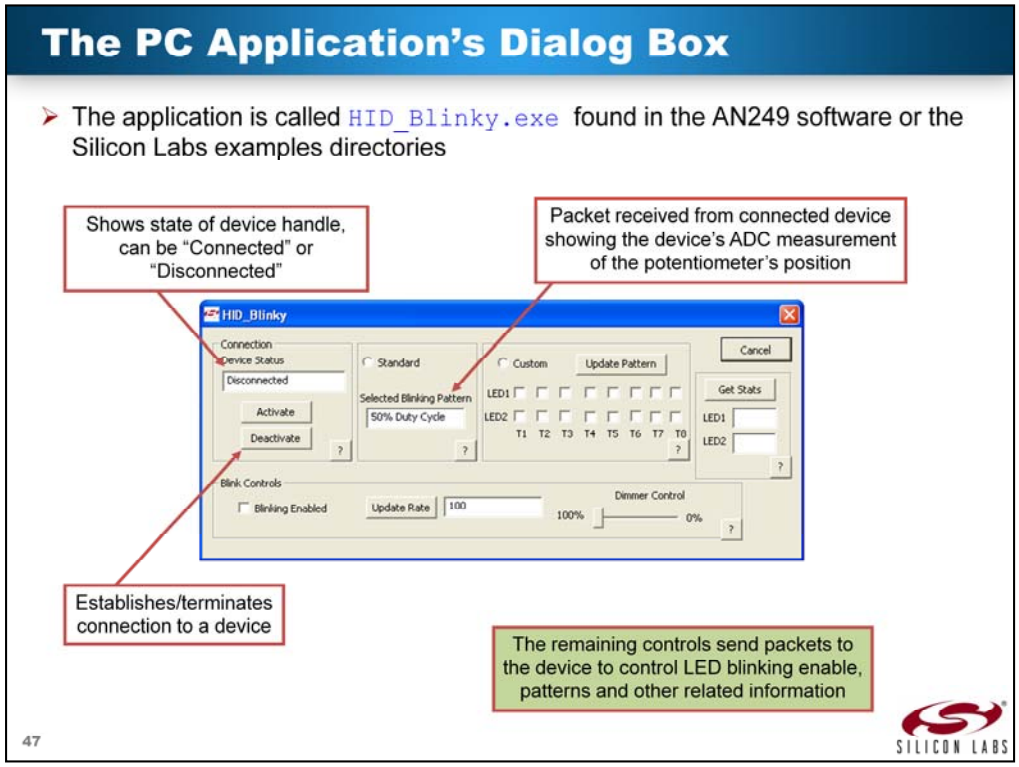
45

SILICON LABS

Here we see a flow diagram that is representative of many embedded applications that require an interface to a host. Unlike the last example, here we need to develop software for both the embedded system and the host PC. The communication channel used will be the USB HID class and the report formats will be pre-defined and transferred during enumeration.

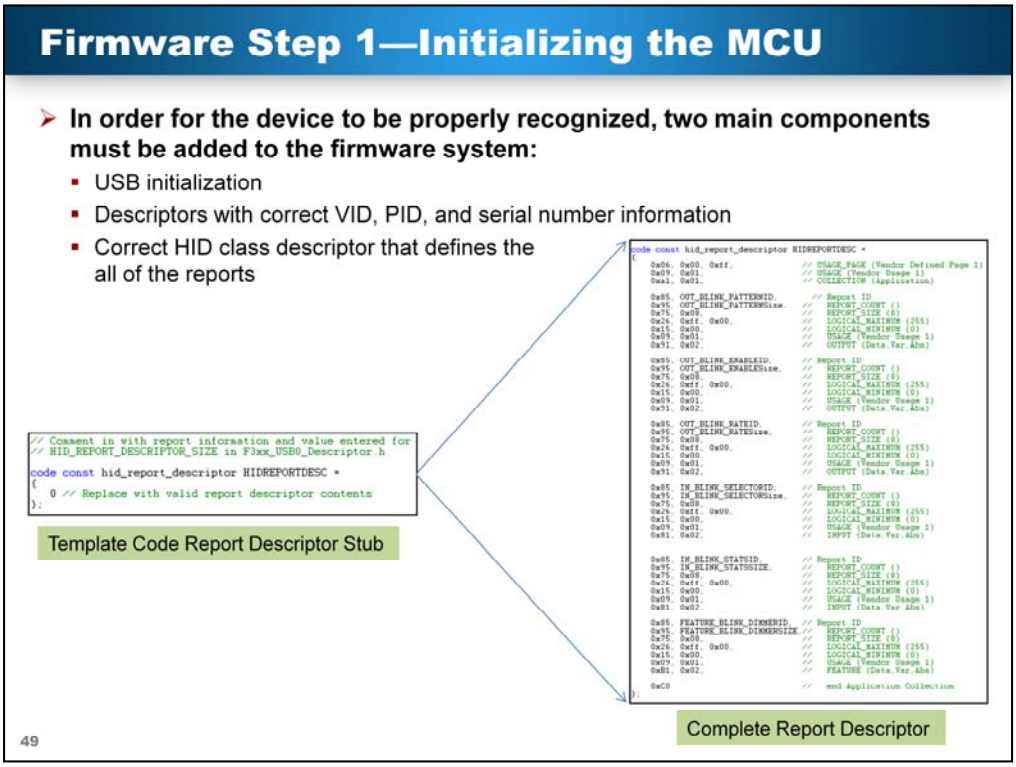


Here we see a flow diagram that is representative of many embedded applications that require an interface to a host. Unlike the last example, here we need to develop software for both the embedded system and the host PC. The communication channel used will be the USB HID class and the report formats will be pre-defined and transferred during enumeration. In the example shown above, there is a report defined that allows the PC to retrieve the ADC readings of the potentiometer using an INPUT report. It is the responsibility of the device to measure the ADC and place the data into the buffers used to send the data to the host. An OUTPUT report will be defined in order to send the LED parameters from the host to the device. It is the responsibility of the device to receive this data and set up the port I/O and timers accordingly.



In many systems a proprietary PC application is required to communicate the data to the endpoint device. Pictured above is an example application from the IDE install examples that uses the HID driver and an API to communicate with the driver. It communicates with the MCU on the Silicon Labs development kits, but is representative of applications that require the USB as the data interface. Let's take a look at how to implement a system that takes advantage of the HID class for the data transfer.


SILICON LABS
www.silabs.com
HID Firmware Overview



The first thing that is required is to configure the MCU. All of the initialization routines for the peripherals used are already configured via the firmware template code. The part of the initialization that is required to be modified is the USB descriptors since each device is different. For example, the VID and PID will change from the template. In the firmware the descriptors are provided with initial values and the user can modify them to suit their needs. The example above shows how the template code uses a “stub” as a place holder where the user can enter the data for the HID descriptors. The descriptors, as mentioned, are what the host uses to initialize the device during the enumeration process. The completed descriptor is defined at design time and the data formats and report IDs are defined within the HID descriptor as shown above. For example, the OUT_BLINK_RATE report controls the rate at which the LEDs blink. The data contained in this report will be used by the device to configure something like the PCA or general purpose timer. In the firmware, the report ID is defined as OUT_BLINK_RATEID with the data format definitions following the report ID. Concluding the specific report is the OUTPUT declaration of the report. Notice that there are two INPUT reports and one FEATURE report. There are three OUTPUT reports defined as well. We will take a look at how to declare these in the code.

Firmware Step 2—Transmitting a Report (1 of 4)

- Transmitting a report means getting the data ready for when the host asks for it
- Data is formatted according to the report descriptor

```
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)
```

Example Report

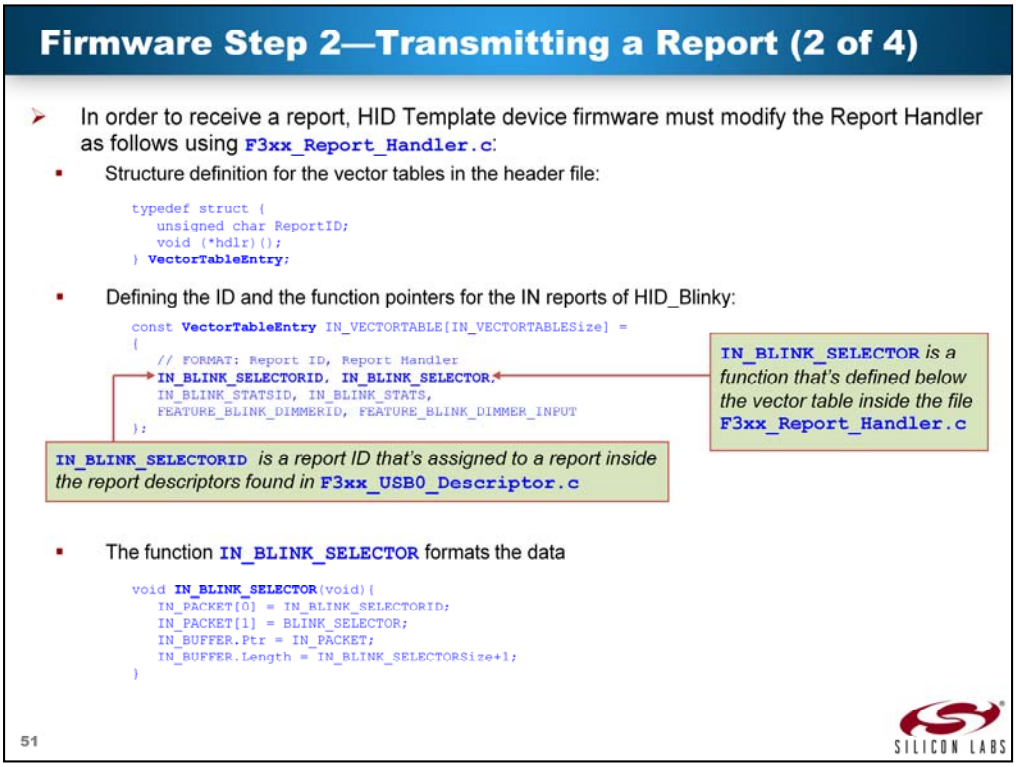
8 bit data (IN_BLINK_SELECTOR)	Report ID (IN_BLINK_SELECTORID)

Example Report Format

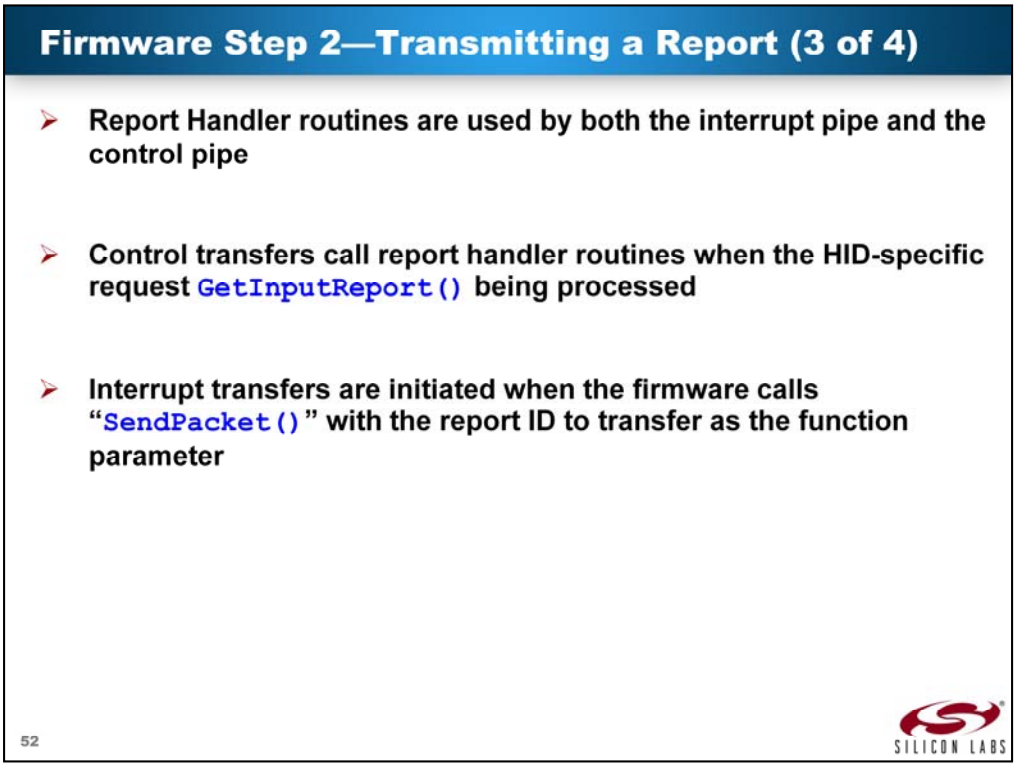
50

SILICON LABS

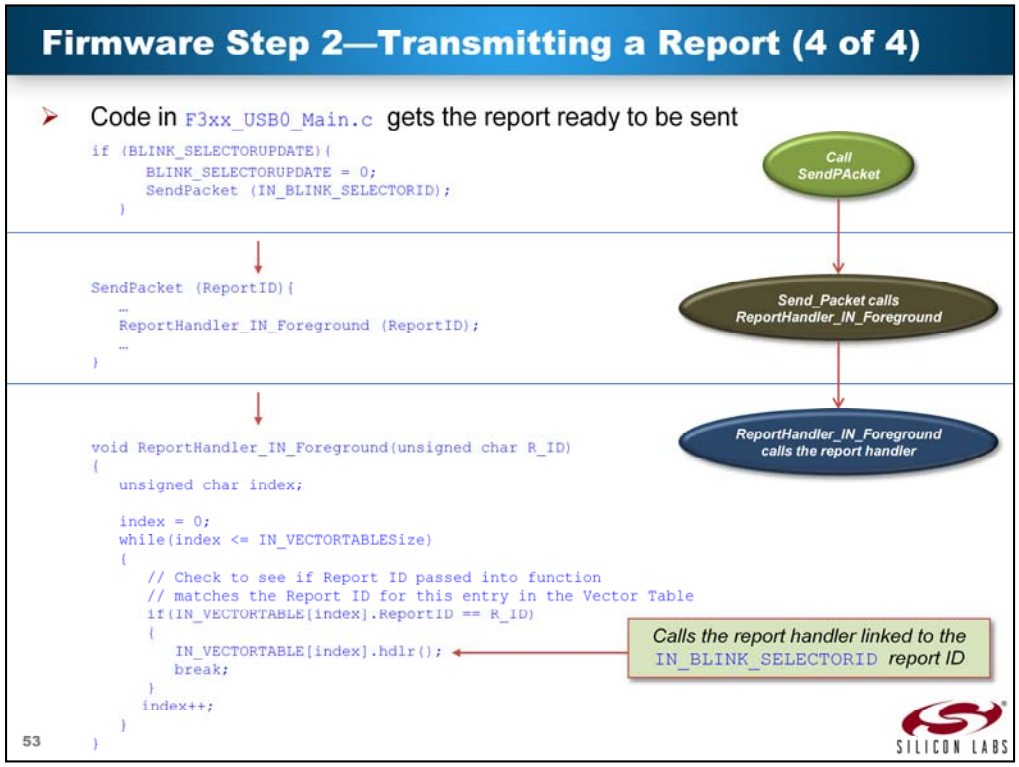
Once the data structure and descriptors are defined the host and device applications must format the data according to the report descriptor. In the example shown above, with IN_BLINK_SELECTORSize defined 1 the report would appear as shown in the Example Report Format figure containing the report ID and the single byte of data.



The first bullet shows the type definition of the structure that contains the report ID and the function pointer. When declaring the structure in the code it will be in this format. Next the structure is declared within the code in order to set up all of the report IDs and the report handler functions. In this case there are two INPUT reports and a FEATURE report defined as part of the IN_VECTORTABLE. The first entry in the structure is the IN_BLINK_SELECTORID which has IN_BLINK_SELECTOR as the report handler function. As can be seen in the last code snippet, IN_BLINK_SELECTOR adds the report ID and the data to the buffer as well as defines the buffer size and sets the pointer for the report to the packet containing the data.



The host can request the data either by generating a control or interrupt transfer and each is accomplished via a Windows API call. If the host retrieves data via the control pipe then it will call GetInputReport() with the request report descriptor. The device firmware then performs the required functions using the endpoint 0 handler routines to process the data. When using the interrupt pipe the firmware prepares the data to be sent via the SendPacket() call and the device firmware will have to have the data ready in the USB FIFO when the request is generated to the associated endpoint. If data is not ready then the device will NAK the requested transfer from the host.



We just saw how all of the structures are defined which set how the data is structured and how the report handler function formatted it. Here we see the code progression for sending the packet. In the main loop the firmware is continually looking to send the data. In this case, when the BLINK_SELECTORUPDATE is requested the firmware calls the SendPacket routine with the BLINK_SELECTORUPDATE report ID. From the SendPacket routine the report handler is indirectly called from the function that handles the IN foreground which searches the vector table to find a matching report ID. Once found the report handler is called (via the function pointer) using the index into the vector table. Remember from the previous section that the report handler puts the update data into the buffer, adds the report ID to the buffer, sets the buffer length and sets the pointer to the buffer. Once complete the data is pushed to the USB FIFO and sent to the host.

Firmware Step 3—Receiving a Report (1 of 4)

- Receiving a report requires the firmware to accept the data when the host sends it
- Data is formatted according to the report descriptor

```
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)
```

Example Report

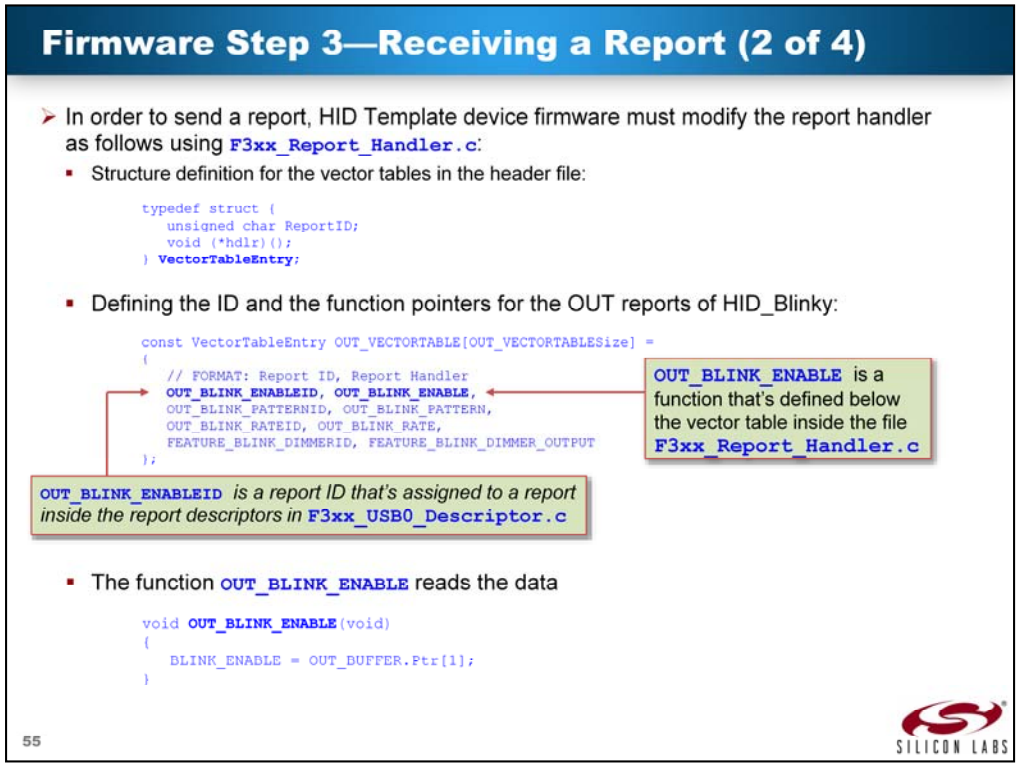
8 bit data (OUT_BLINK_ENABLE)	Report ID (OUT_BLINK_ENABLEID)

Example Report Format

54

SILICON LABS

Now let's take a look at when the host wants to send data to the device. There is a report defined as an OUTPUT report and the data format example is shown above. In this case the data size is one byte and the report ID is also one byte. The result of the report descriptor is shown in the Example Report Format.

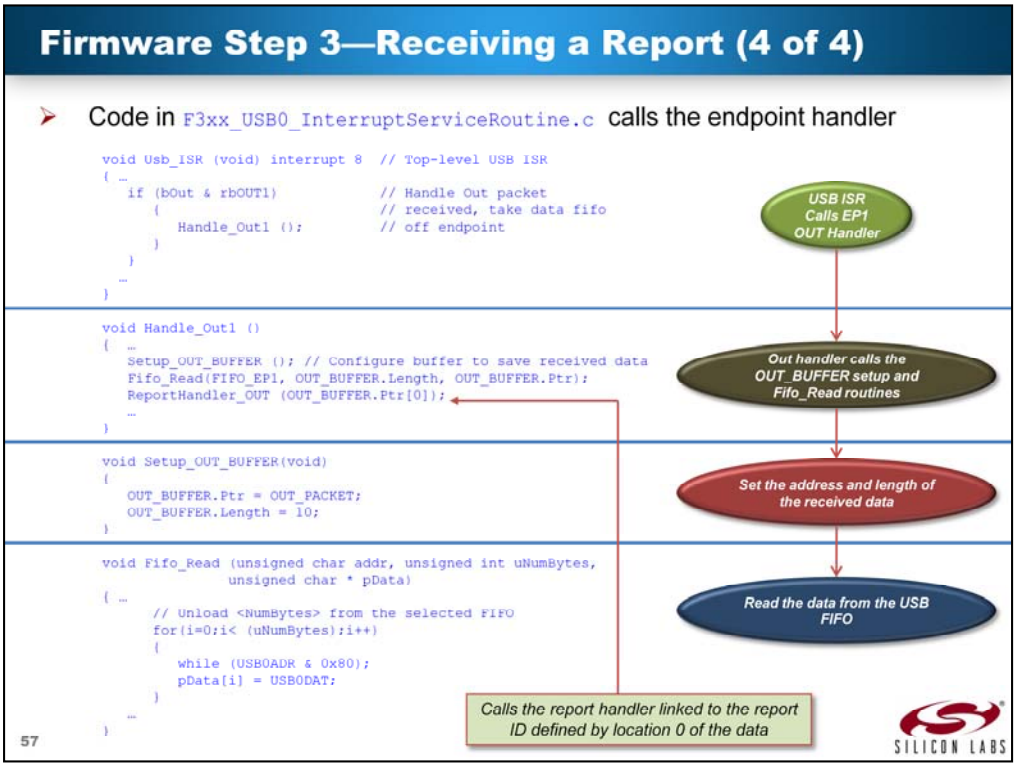


Like the receive direction, the OUT direction also has a vector table defined using the VectorTableEntry structure definition. In this example, there are three reports entered into the vector table. In this case, the firmware has placed the data into the OUT_BUFFER. In order to accept the data the firmware just reads the second byte in the buffer (the first byte is the report ID).

Firmware Step 3—Receiving a Report (3 of 4)

- Report Handler routines are used by both the interrupt pipe and the control pipe
- Control transfers call report handler routines when the HID-specific request `SetOutputReport()` being processed
- Interrupt transfers call the report handler routines when a report is received by the OUT interrupt endpoint

SILICON LABS

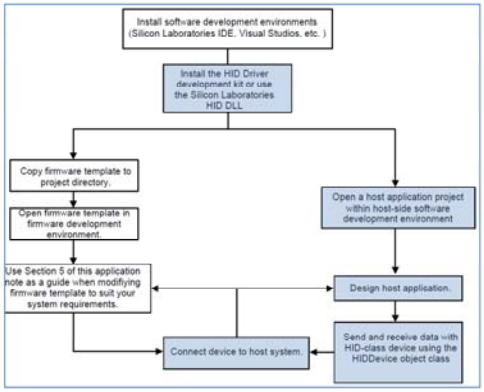


Here is a more detailed look at the firmware progression for reading data from the host. The data received by the USB peripheral generates an interrupt. Within the interrupt routine the firmware decodes the transfer request and determines that the OUT handler needs to be called, labeled Handle_Out1() in the slide. The Handle_Out1 function sets up the buffer pointer and size (using the Setup_OUT_BUFFER routine) and then reads the data from the FIFO and stores it to the OUT_BUFFER. After the data has been read and stored from the USB it is then processed by the report handler via the ReportHandler_OUT function which has the first byte of the data packet (the report ID) passed as a parameter.


SILICON LABS
www.silabs.com
Host HID Overview

Generating the Host Application

- Silicon Labs provides a DLL that has all of the APIs required to pass data to the USB HID driver
- 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.



59

SILICON LABS

Now that we have developed the code flow for the device side firmware, how do we communicate with it via the HID class driver and the host application? Silicon Labs simplifies the process by taking the DLLs from the Windows Driver Development Kit (Windows DDK) and provides them in a library. This enables the host application to call the functions without the developer having to download the DDK. In addition, the library contains functions that are not included in the Window DDK that simplifies finding devices on the USB and communicating with them.

HID DLL Member Functions (1 of 3)

- The HID DLL functions as a wrapper to HID-related functions included with the Windows DDK
- This allows customers to develop code that interfaces with HID devices without installing the Windows DDK

HID DLL Wrapper Function	Description
`HidDevice_GetInputReport_Control()`	This function receives a report across the control pipe using the HID API function `HidD_GetInputReport()`
`HidDevice_SetOutputReport_Control()`	This function sends a report across the control pipe using the HID API function, `HidD_SetOutputReport()`
`HidDevice_SetFeatureReport_Control()`	This routine sends a feature report across the control pipe using the HID API function `HidD_GetFeatureReport()`
`HidDevice_GetFeatureReport_Control()`	This routine retrieves a feature report across the control pipe using the HID API function, `HidD_GetFeatureReport()`

60

SILICON LABS

On this slide and some of the upcoming slides are the API calls from library. Listed in the table above are the API functions that use the control transfer to communicate the data over the USB. In this case there are functions for sending and receiving both the data and features of the device.

HID DLL Member Functions (2 of 3)

- The HID DLL saves information retrieved from the device's descriptors that can be accessed through member function calls, as well as encapsulating many non-HID specific calls

HID DLL Wrapper Function	Description
`HidDevice_GetInputReportBufferLength()`	Returns size of largest defined IN report
`HidDevice_GetOutputReportBufferLength()`	Returns size of largest defined OUT report
`HidDevice_GetFeatureReportBufferLength()`	Returns size of largest defined FEATURE report
`HidDevice_GetMaxReportRequest()`	Returns size of largest defined report
`HidDevice_FlushBuffers()`	Clear all system buffers storing report data
`HidDevice_GetTimeouts()`	Returns time that the object will wait for a read or write to execute in ms
`HidDevice_SetTimeouts()`	Sets time object will wait for read or write to execute in ms
`HidDevice_Open()`	Calls low-level system functions to establish handle to HID device
`HidDevice_Close()`	Calls low-level system functions to terminate handle with HID device

61

SILICON LABS

Here are some more functions and their descriptions from the library.

HID DLL Member Functions (3 of 3)

- The HID DLL also provides a number of functions which can be called independently of an instance of the HID object
- These functions allow applications to search for connected HID devices

HID DLL Wrapper Function	Description
`HidDevice_GetNumHidDevices()`	Returns number of HID devices connected that have VID and PID values matching input parameters
`HidDevice_GetHidString()`	Returns a string from a device matching an input VID, PID, and device number
`HidDevice_GetHidGuid()`	Returns HID information that is needed by surprise removal routines

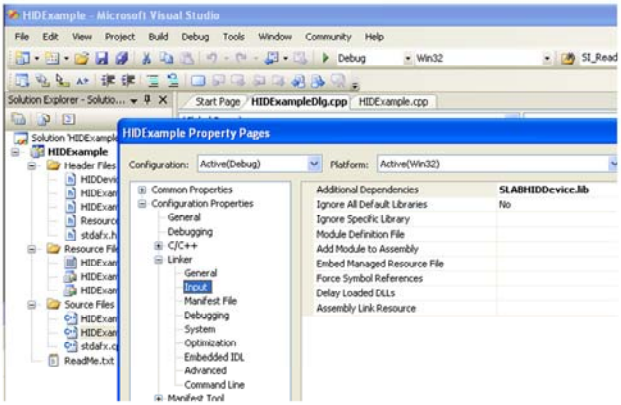
62

SILICON LABS

Some library functions are not related to the HID object, which means that a specific device has been found and a handle to that device has been assigned. For example, before a device is used and opened to get a handle, the host application can search the bus to find out how many devices are connected to the USB.

PC Application Step 1—Using the DLL

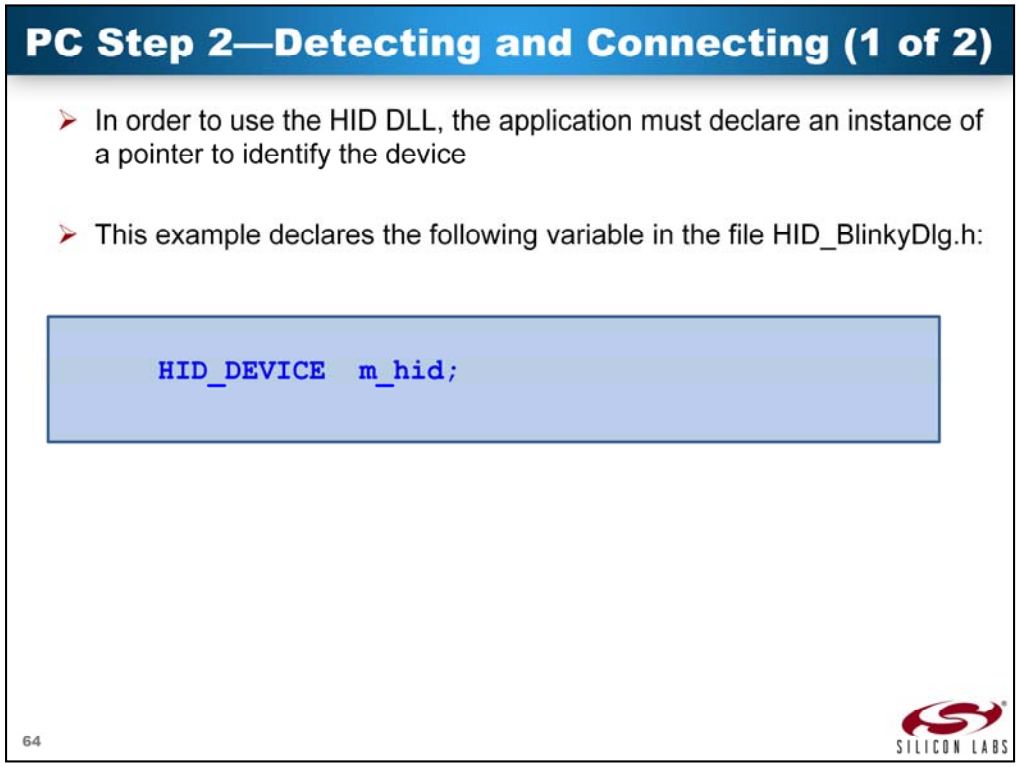
- **To use the HID DLL in a project, do the following:**
  - Copy the DLL, LIB, and header files to the project's directory
  - Add the library to the build (Visual Studio 2005 shown)
    - Click on "Project → Properties" to open the screen below:
    - In the window that appears, click Linker → Input and add `SLABHIDDevice.lib` to the "Additional Dependencies" list
    - Include the header file `SLABHIDAPI.h` in files that use DLL routines



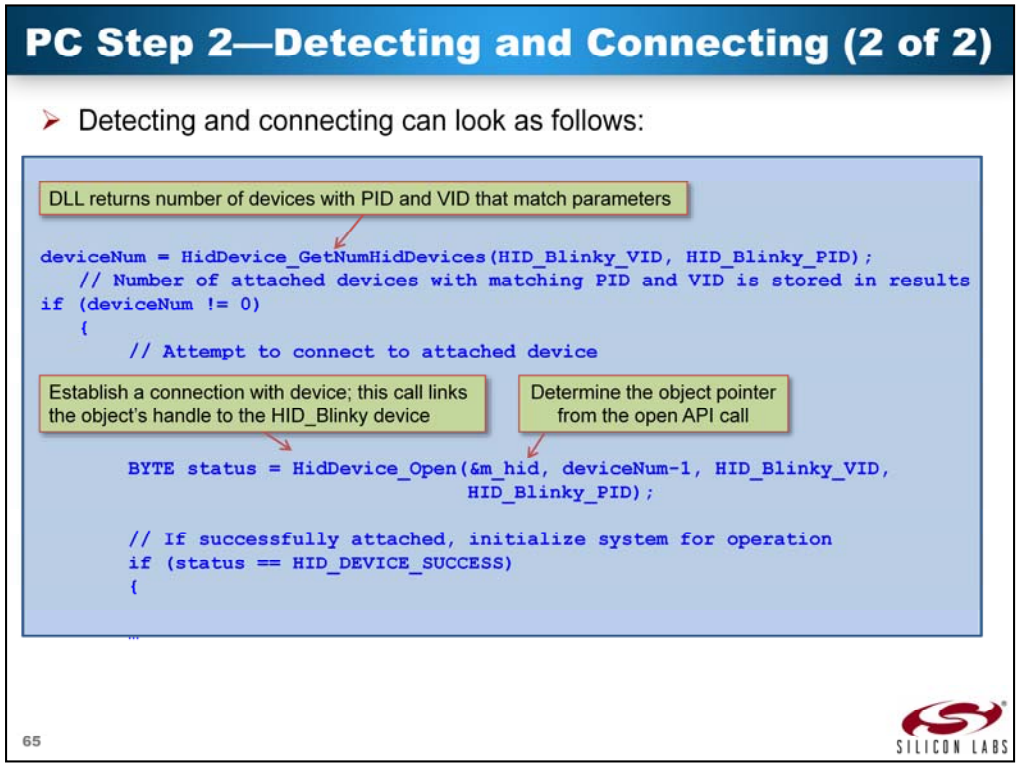
63

SILICON LABS

Now let's use the library to build the application. The first thing to do is to tell the tools what library to use. Within the development environment the developer will need to add the library to the build process. This is typically within the properties dialogs. The library filename is SLABHIDDevice.lib.



When using the HID class for data transfers the USB device will have to declare an instance of the object. HID_DEVICE is defined in the library and is used to declare a variable that identifies the specific HID object that is being referenced. In the header file for the HID_blinky example the statement shown above can be found. This declares m_hid as the pointer to the object and will be used in subsequent access to the HID object.

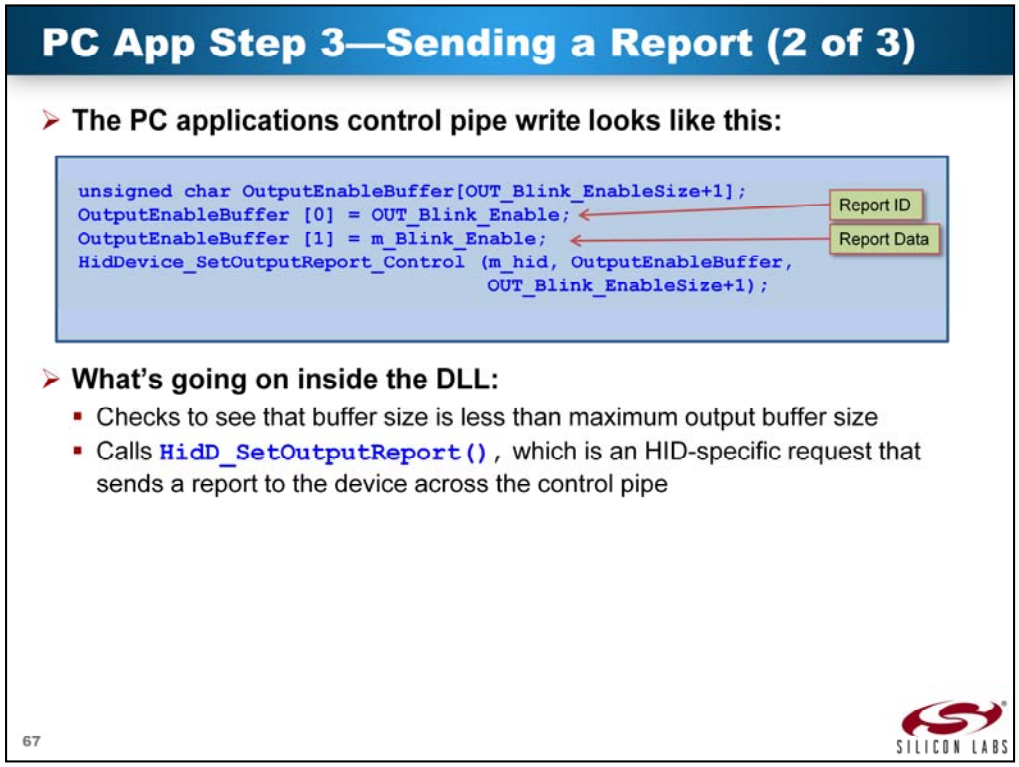


In the previous slides we mentioned that there are some functions in the library that are not HID object specific. Here we see an example of one of those functions, the HidDevice_GetNumDevices function. In this case the function uses the VID and PID to search for a specific type of device. The search can be made broad to where the call returns all connected devices as well. If the host application determines that there are connected devices that match the search criteria then it can then open and start a thread for that device. The open API call determines the pointer to the object with the matching VID and PID.

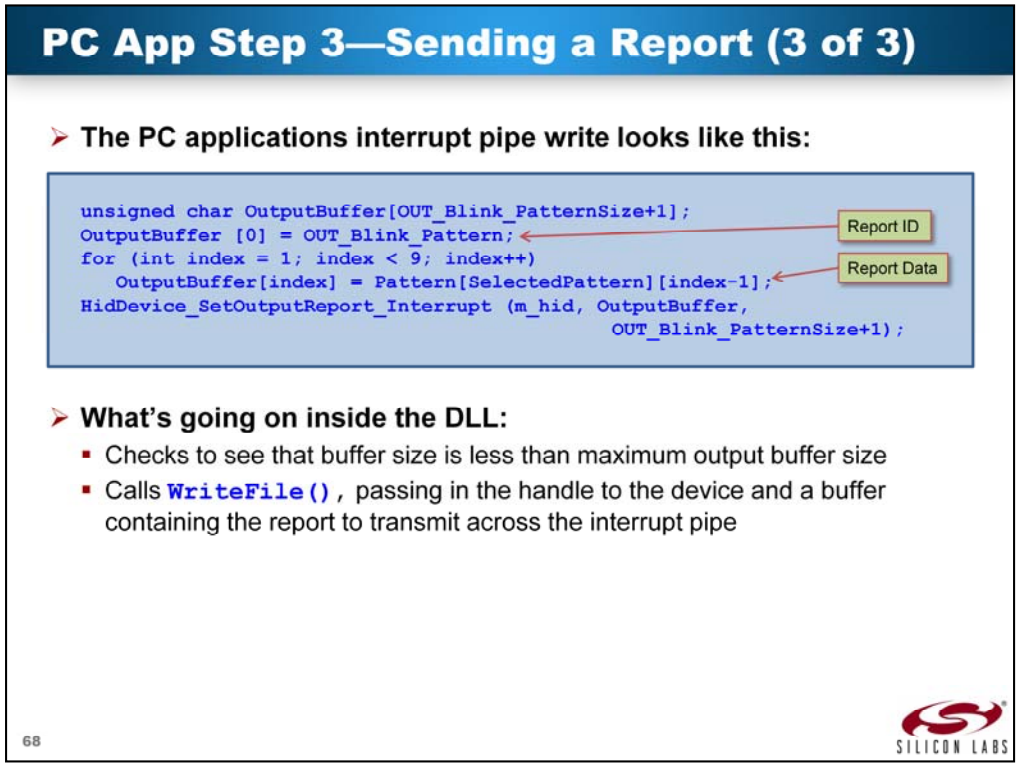
PC App Step 3—Sending a Report (1 of 3)

- HID allows users to transmit across either the control pipe or the interrupt pipe
- The HID DLL enables control pipe writes using the member function `HidDevice_SetOutputReport_Control()`, which takes the following parameters:
  - Pointer to the object
  - Buffer containing data to transmit
  - Size of buffer to transmit
- The DLL enables interrupt pipe writes using a similar function, `HidDevice_SetOutputReport_Interrupt()`, which takes the following parameters:
  - Pointer to the object
  - Buffer containing data to transmit
  - Size of buffer to transmit
- The PC application in this example switches between control and interrupt pipe transfers

SILICON LABS



In this example we see how data can be transferred from the host to the device using a control transfer. The data buffer is formatted with the report ID and then the data. Once that data buffer is ready it can be passed to the API call to initiate the transfer across the USB.



Sending data across the interrupt pipe is similar to sending across the control pipe, however, the transfer method across the USB is different. The buffer is allocated with the report ID and the data and then the API call is executed. The DLL then call WriteFile() to send the data in an interrupt transfer.

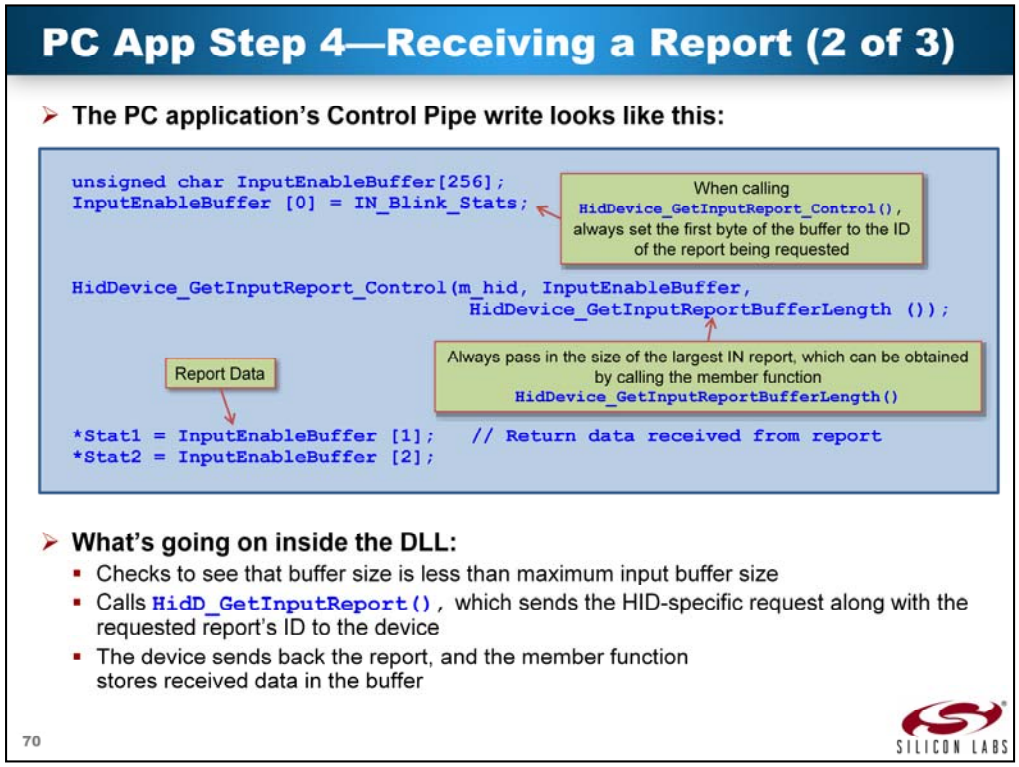
PC App Step 4—Receiving a Report (1 of 3)

- **HID allows users to receive reports across either the control pipe or the interrupt pipe**
- **The HID DLL enables control pipe receives a report using the member function `HidDevice_GetInputReport_Control()`, which takes the following parameters:**
  - Pointer to the object
  - Buffer where received data will be stored
  - Size of buffer
- **The DLL attempts interrupt pipe read using a similar function, `HidDevice_GetInputReport_Interrupt()`, which takes the following parameters:**
  - Pointer to the object
  - Buffer where received data will be stored
  - Size of buffer to receive
  - Number of reports to attempt to receive before returning
  - Number of bytes actually returned
- **The PC application can switch between control and interrupt pipe transfers**

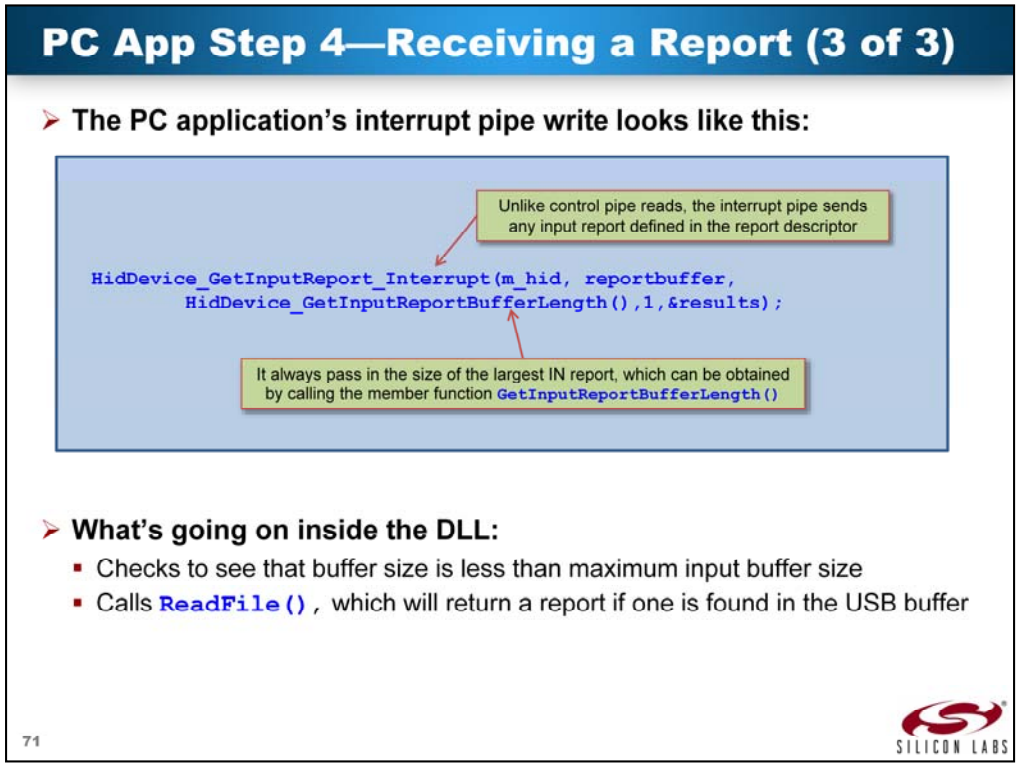
69

SILICON LABS

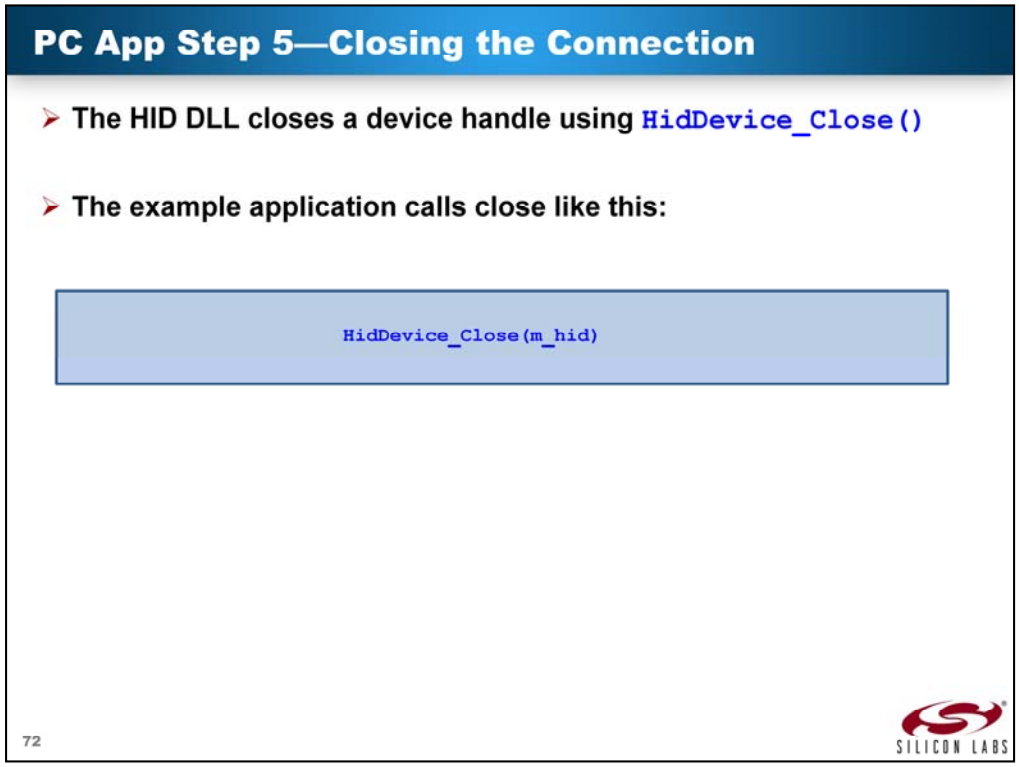
Here we see the two API calls used for receiving data from the device and the parameters used in each call.



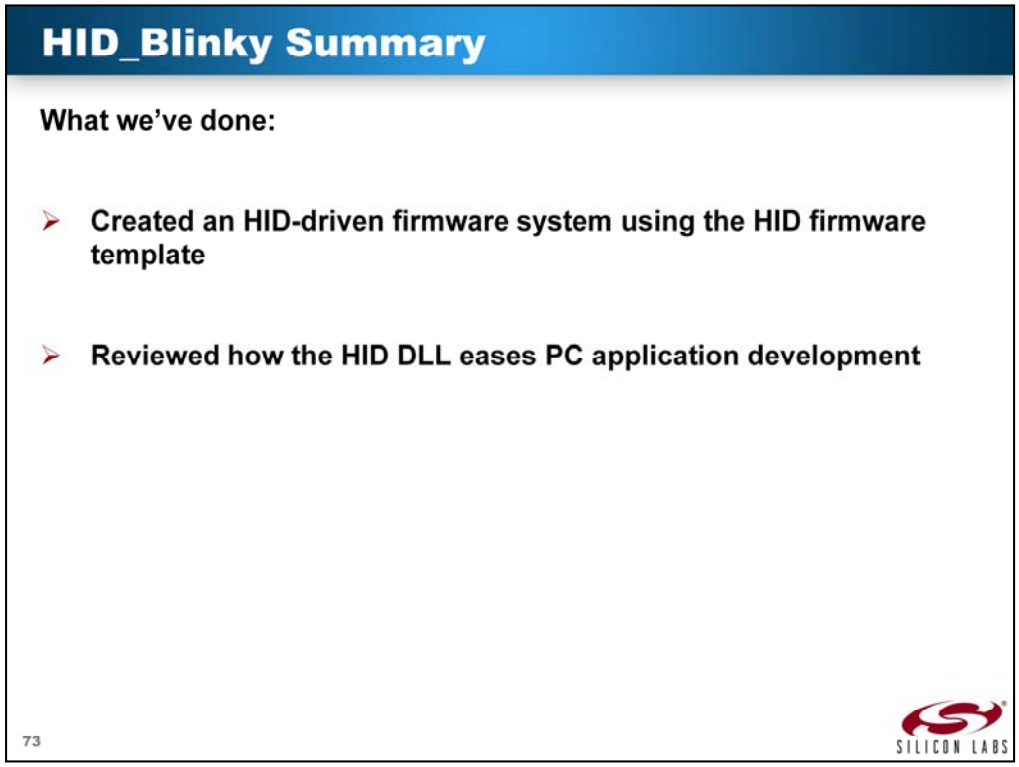
In this command the host formats the request by sending the requested report ID in the LSB of the input buffer. The device will receive this request and add the data to the report for the IN data phase. When the host receives the data it will copy the data to the variable location in memory.



The host can also retrieve data via the interrupt endpoint using ReadFile.



When closing an application or terminating the connection the host must close the connection.



The HID_Blinky example illustrates how to implement both the device firmware and the host application software to perform USB connectivity using the standard class driver for HID devices. Using the firmware template and existing example code developers can get HID implementations up and running quickly. In addition, the host side DLL alleviates the need to directly interface to the Windows DDK API libraries.


SILICON LABS
www.silabs.com
www.silabs.com/MCU