Learning USB by Doing.

John.Hyde@intel.com

The question that I am asked most often is "how do I start a USB project?" There are many alternate starting points for the design of a USB I/O device and this paper focuses on what I consider the simplest, and in many ways the lowest cost, method of ***using a PC as an I/O device***. It is possible to use USB-to-parallel, USB-to serial or pre-built "byte-mover" products but I feel that these solutions mask much of the flexibility and value of a USB solution. The "PC as an I/O device" method presented here will expose many of the details of USB so that you may exploit its capabilities. This paper first covers some essential USB theory[1] and then presents a solution using a low-cost adaptor[2] and a PC. Several example solutions are presented to get you going and all source code and schematics are downloadable for your experimentation.

Essential Theory.

A USB I/O device is a combination of software and hardware. Figure 1 shows the software model of an I/O device attached to a Host PC – this comprehensive diagram is best described from the bottom up.

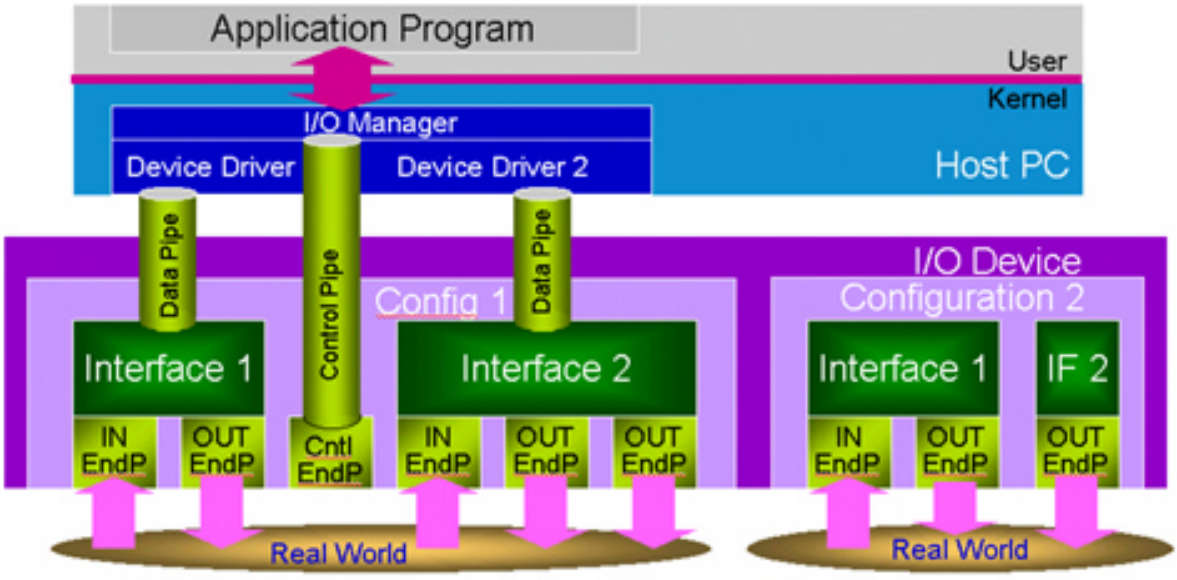

Figure 1 - USB Software Model

An I/O device interfaces to the real world. It will use sensors to collect real-world information such as temperature, speed, color, density and size and will use actuators and transducers to control the real world. Data collected from the real world is placed in a local buffer area called an **IN endpoint** – the Host PC will collect this data later. Similarly, data is delivered by the host PC into a local buffer called an **OUT endpoint,** this data is then distributed to the real world by the I/O device.

A USB I/O device ***requires*** a CPU – this could be a microcontroller, microprocessor, digital signal processor or a fixed-function state machine. This CPU is responsible for translating the signals from the real world into data in the IN endpoint and for translating the data in the OUT endpoint into signals for the real world. This arrangement of having a local CPU which can pre- and post-process data on behalf of the Host PC is one of the key elements of flexibility and standardization within a USB-based system implementation. Think of this as an extension to BIOS – the Host PC operating system provides some generic data to a generic I/O device via this OUT endpoint and the specifics of handling a particular I/O device is managed by the CPU in the I/O device. Similarly specific actions needed for particular data input devices are handled by the I/O device CPU and generic data is provided to the Host PC via an IN endpoint. A typical PC-BIOS will handle keyboard, mouse, display and computer-type I/O devices. The USB model is extensible to handle ***any*** type of data acquisition or distribution system and ***any*** type of data format including sound and video.

A collection of endpoints is called an **interface**. An interface describes the device CLASS. Classes are an important concept that I should explain. A typical operating system (such as Windows or Linux) supports a wide array of I/O devices – it broadly groups similar devices into classes and will use a generic class device driver to interact with this group of similar devices. A typical operating system will include a range of class drivers, such as printer, audio, human interface device (HID) and mass storage, to support a wide collection of I/O devices. All operating systems allow you to write your own device driver if your particular I/O device is not supported by an included class device driver but, be warned, it is a LOT of work to create and maintain a device driver. The strategy that I shall adopt in this paper is to make best use of pre-existing class drivers.

There is a one-to-one mapping of I/O device interface to Host PC device driver. Or, to put it another way, an interface will specify which device driver it needs to operate. The INF file mechanism is used in the Windows operating system to bind an interface to a device driver. The Windows operating system also provides default INF files to bind generic devices to generic class device drivers. My examples will use these default files.

Notice in Figure 1 that an I/O device can have more than one interface. A keyboard with built-in mouse/track ball would be two interfaces in a single device. A telephone is an audio class device (for the speech) and a HID class device (for the buttons). Each interface is supported by a matching device driver. The USB specification also allows an I/O device to have multiple configurations but this is not supported (yet) by the Windows operating system.

An I/O device also includes a **control endpoint**. This endpoint is bi-directional and is used by the operating system to discover the identity and capabilities of the I/O device and also to control its operation. Following the initial attachment of a USB I/O device on to a Host PC, a long "conversation" is held on this control endpoint so that the operating system can integrate the I/O device into its operating environment. This process is called **enumeration** in the USB literature and involves a series of pre-formatted standard USB requests – these look quite verbose from a hardware designers perspective but, the good news, is that this is code that may be reproduced on all devices (and is included with this article).

Before leaving Figure 1, I did want to stress that it is a MODEL – the three pipes shown in the diagram are implemented on the same USB cable using different packet types and addresses.

Applications programs open named devices and the operating system implements all of the low-level communications to the I/O device. We, on the I/O device, will see the result of this as data arriving at OUT endpoints and request for us to supply data on IN endpoints.

The minimal hardware for a USB I/O device is shown in Figure 2. A transceiver is required to meet the electrical requirements of the bus and a Serial Interface Engine is required to meet the bit timing of the bus. The Serial Interface Engine is responsible for converting the serial USB packets into valid bytes for the Protocol Controller – these bytes are provided as Endpoints within the SIE. Similarly, the SIE will transmit bytes from endpoints to USB.

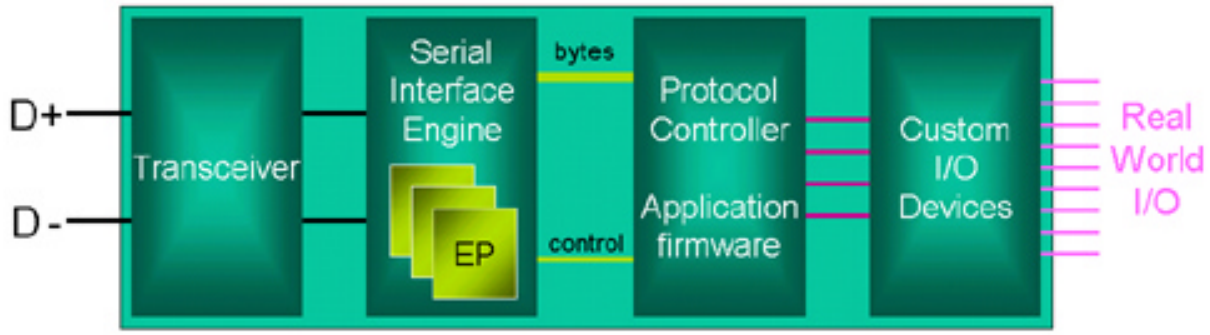

Figure 2 - USB Hardware Model

An I/O device can have up to 16 IN and 16 OUT endpoints through components are typically manufactured with less. All components **must** implement Endpoint 0 and our examples will also use an IN endpoint 1 and an OUT endpoint 1. The protocol controller is the I/O device CPU. It executes a program to respond to requests from the Host PC. A USB I/O device is always a slave and, as such, responds to requests from the Host PC. These requests could be configuration and setup requests on endpoint 0 or data requests on endpoint 1. The operation of an I/O device is fixed – it always responds in the same way. Data requests are manipulated by the application firmware – endpoint OUT data is sent through custom I/O devices to the real world and real-world inputs are prepared in IN endpoints for collection by the Host PC.

Using a PC as an I/O device.

We must use specialized silicon for the USB transceiver and Serial Interface Engine (to meet USB requirements) but we are free to implement the protocol controller and application firmware however we desire. While people typically think of a microcontroller for this task, I am going to use a PC, because of its superior development environment. I will use the adaptor shown in Figure 3 to turn my PC into a USB device. This is not the typical role of the PC in a USB environment so I have also included a host PC in figure 3 – note that there are two PC's, the traditional host PC and now a device PC.

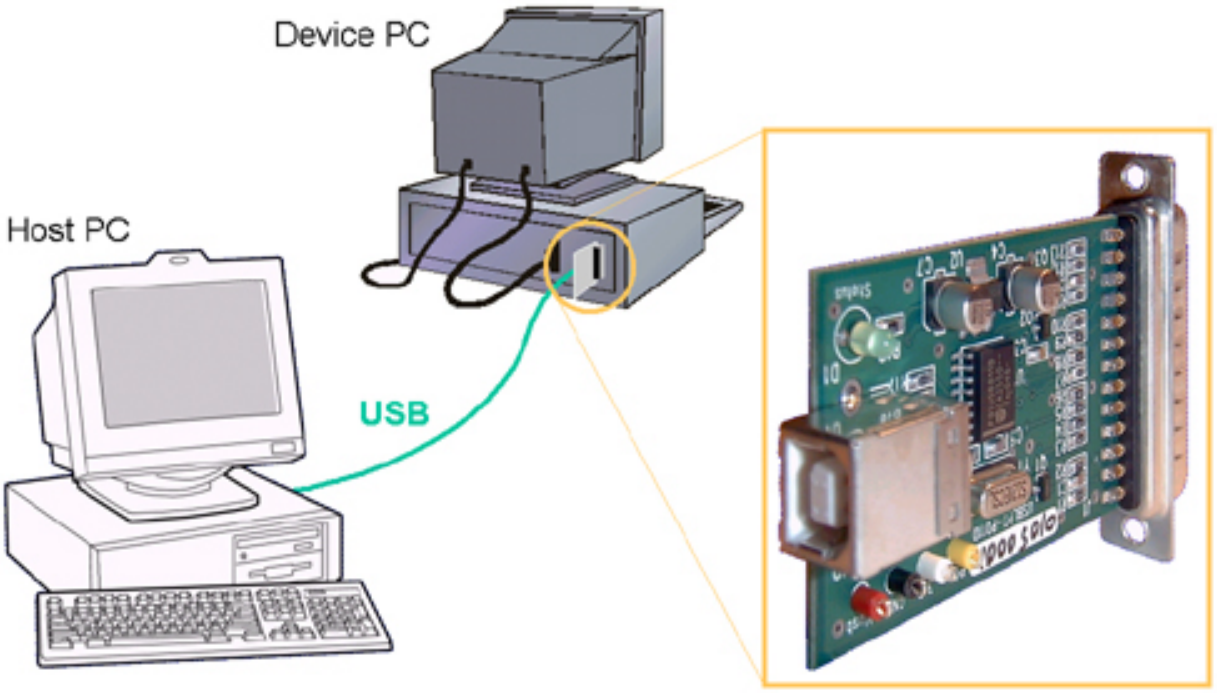

Figure 3 - Using an adaptor to create a device PC

The adaptor shown in Figure 3 uses a USB peripheral device from Philips Semiconductor. The PUSBD11 interfaces between USB and I2C. A standard PC does not have a defined I2C connector but the adaptor creates one using the parallel printer port. Software on the device PC will "bit-bang" the parallel port to implement the I2C connection. This is not super fast – I chose it because it is simple and low cost. You can see exactly what the software is doing and no details are hidden. It is plenty fast enough to learn USB and the low cost allows everyone to have one!

The operation of the PC I/O device is straight forward. It must monitor the state of the endpoints and respond to any requests sent by the Host PC. It must also respond to real-world events and create data for the Host PC if necessary. A flowchart is shown in Figure 4.

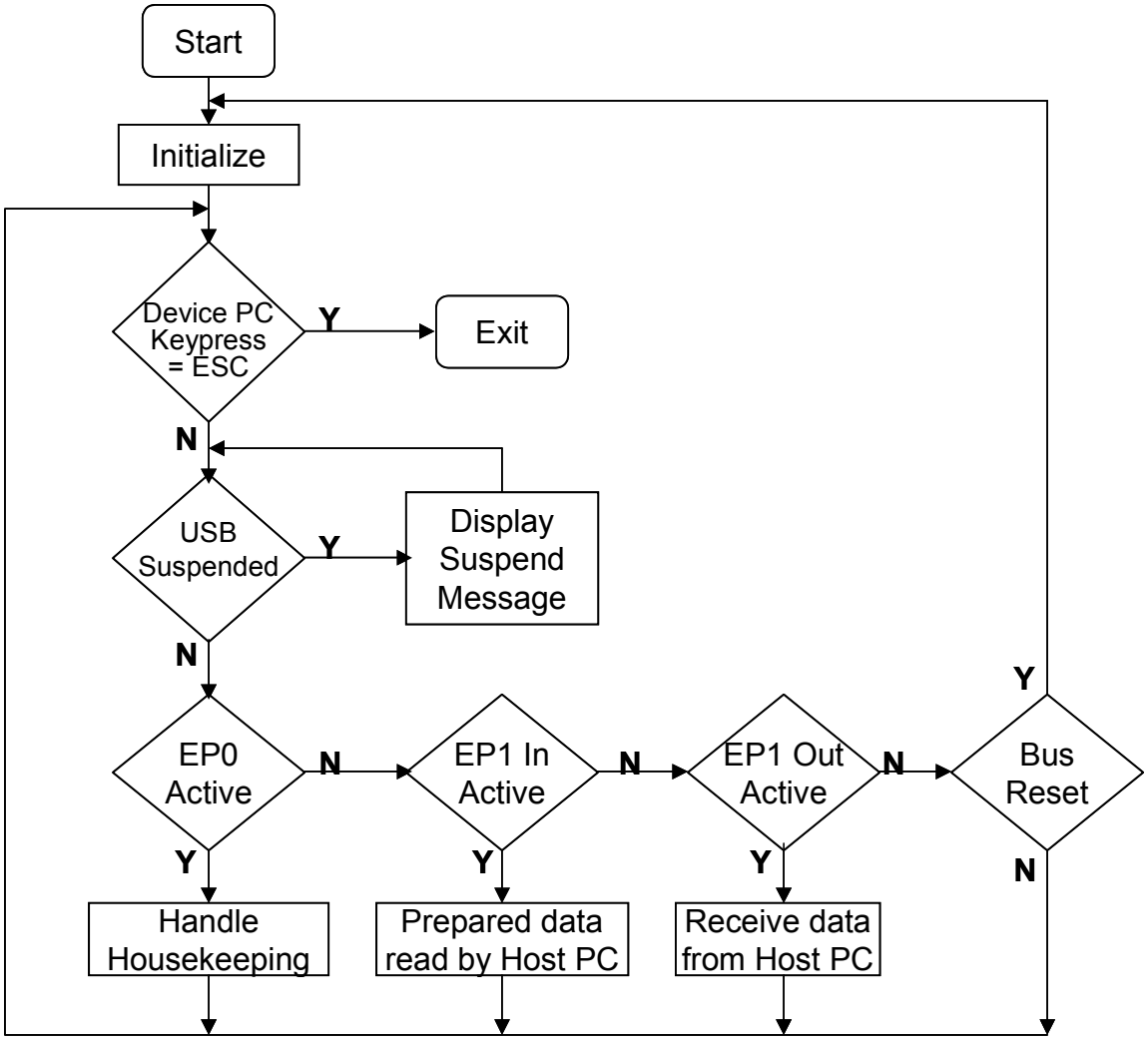

Figure 4 - Endless task of I/O device PC

I included an escape out of this "forever" loop via the device PC keyboard. I also manage any suspend requests locally (but do not actually suspend my device PC). Most of the work deals with the standard request housekeeping – handling the enumeration and device configuration. This software example will be reusable across all of the examples. Each example will use different descriptor tables to implement the different functionality of device.

The response to data requests on Endpoint 1 will be different for each example – note that only a small amount of application code changes between each example. All software is written in Visual C++ so that it is easy to read and port to another target.

Example 1 – a keyboard

The simplest example that I could think of was a USB keyboard. We will convert our device PC into a USB keyboard – this is easy to demonstrate and needs no additional hardware. The following discussion assumes that you have Visual C++ and the Win2000 DDK installed on the device PC.

Open the "PC as keyboard" project file and observe the program modules. They are partitioned into "portable" routines and "hardware dependant" routines as shown in Figure 5.

Figure 5 - I/O device software is partitioned

The hardware dependant routines match the specific adaptor that we are using and are common across all examples. The examples will differ only in the Main () program and in the descriptor table entries. The descriptors for the first example are shown in Figure 6.

```
// Descriptors.h :the Descriptor Tables for this Keyboard Example
//
const byte ReportDescriptor[] = {
       5, 1,          // Usage_Page (Generic Desktop)
       9, 6,          // Usage (Keyboard)
       0xA1, 1,       // Collection (Application)
       5, 7,          //   Usage page (Key Codes)
       0x19, 224,     //   Usage_Minimum (224)
       0x29, 231,     //   Usage_Maximum (231)
       0x15, 0,       //   Logical_Minimum (0)
       0x25, 1,       //   Logical_Maximum (1)
       0x75, 1,       //   Report_Size (1)
       0x95, 8,       //   Report_Count (8)
       0x81, 2,       //   Input (Data,Var,Abs) = Modifier Byte
       0x81, 1,       //   Input (Constant) = Reserved Byte
       0x19, 0,       //   Usage_Minimum (0)
       0x29, 101,     //   Usage_Maximum (101)
       0x15, 0,       //   Logical_Minimum (0)
       0x25, 101,     //   Logical_Maximum (101)
       0x75, 8,       //   Report_Size (8)
       0x95, 6,       //   Report_Count (6)
       0x81, 0,       //   Input (Data,Array) = Keycode Bytes(6)
       5, 8,          //   Usage Page (LEDs)
       0x19, 1,       //   Usage_Minimum (1)
       0x29, 5,       //   Usage_Maximum (5)
       0x15, 0,       //   Logical_Minimum (0)
       0x25, 1,       //   Logical_Maximum (1)
       0x75, 1,       //   Report_Size (1)
       0x95, 5,       //   Report_Count (5)
       0x91, 2,       //   Output (Data,Var,Abs) = LEDs (5 bits)
       0x95, 3,       //   Report_Count (3)
       0x91, 1,       //   Output (Constant) = Pad (3 bits)
       0xC0           // End_Collection
       };

const DeviceDescriptorType DeviceDescriptor = {
       18, 1, 0, 1, 0, 0, 0, EP0Size, 0x42, 0x42, 16, 1, 0, 1, 1, 2, 0, 1
       };
typedef struct {
       ConfigurationDescriptorType Configuration;
       InterfaceDescriptorType Interface;
       ClassDescriptorType Class;
       EndPointDescriptorType EPin;
       EndPointDescriptorType EPout;
       } KeyboardConfigurationDescriptorType;

const KeyboardConfigurationDescriptorType My = {
// Base Configuration Descriptor Follows
       9, 2, sizeof(KeyboardConfigurationDescriptorType), 0, 1, 1, 0, 0xC0, 50,
// Interface Descriptor Follows
       9, 4, 0, 0, 2, 3, 0, 0, 3,
// HID Class Descriptor Follows
       9, 0x21, 0x10, 1, 0, 1, 0x22, sizeof(ReportDescriptor), 0,
// Endpoint Descriptors Follows, this HID example uses INT IN and INT OUT
       7, 5, 0x81, 3, EP1Size, 0, 100,
       7, 5,    1, 3, EP1Size, 0, 100
       };

// Define Strings as UNICODE and create descriptors at runtime
wchar String0[] = {0x0409, 0x0000};
wchar String1[] = L"USB Design By Example";
wchar String2[] = L"Keyboard using a PC ";
wchar String3[] = L"HID interface";
wchar *String[] = {String0, String1, String2, String3};
byte StringSize[] = { sizeof(String0), sizeof(String1), sizeof(String2), sizeof(String3) };
```

Figure 6 - Descriptors for a USB Keyboard

A keyboard is a Human Interface Device and, as such, must follow the definition of a HID as expected by the operating system. A HID is required to have a control endpoint and an interrupt IN endpoint. It can optionally have an Interrupt OUT endpoint so that the operating system can

send information to the keyboard (to change the status of the LEDs). A HID exchanges data with the operating system using **reports** – a report descriptor defines the type, format and size of theses reports. The descriptor in Figure 6 defines an 8 byte input report {modifier keys, reserved = 0, scan code (6)} and a 1 byte output report {LEDvalue}.

Most of the code of the "PC as Keyboard" module deals with responding to the Host PC requests on the Control Endpoint 0. I wrote this to be table driven so that it is easy to follow and to enhance when more features are required.

When the program is executed the device PC will enable the adaptor which will "attach" to the host PC. The host PC will run a standard device enumeration and will discover, via the device PC descriptors, that a keyboard has been attached. The operating system will use HIDDEV.INF to match our "PC as keyboard" device to the HID class driver. I included a global variable, DebugLevel, which controls the depth of routine where messages are displayed on the device PC console – a setting of 3 will cause the enumeration process to be displayed.

After the device driver is installed and activated it sends a series of output reports to set the state of our "PC as keyboard" LEDS.

When a key is entered on the device PC, an input report is created and this keystroke is sent to the host PC. Windows will merge this input with characters typed on the host PC keyboard and will send these to the currently active window. Output reports from the host PC are received and a local variable, LEDValue, is updated. The operation of this first example is simple and straightforward.

Lets recap the key lessons learned from this first example:

- A USB I/O device is a slave that responds to commands
- Data is exchanged using buffers called Endpoints
- Functionality of a USB I/O device is defined in descriptors
- Descriptors are provided to the host PC during enumeration
- Code to run enumeration sequence is standard (and provided with this paper!)
- Run-time software deals with data on non-EP0 Endpoints

Example 2 – Buttons and Lights

For my second example, I chose a generic device with a single 8 bit input value (8 buttons) and a single 8 bit output value (8 LEDS). If we can simply exchange data with this generic, custom I/O device then the design could be easily adapted to input and output ***any*** kind of data using appropriate sensors and transducers. The overall data rate of this application is quite low (less than 8 KB/sec) so I will use the HID class driver to exchange data between the host PC and the device PC; this means that we have no operating system software to write for this custom I/O device. We will need to write a custom application program to access this custom hardware and I will use Visual Basic on the host PC for this example.

I decided to attach my custom buttons and lights hardware to my adaptor using the I2C bus. A single Philips part, the PCF8575, was used to create 16 I/O lines from the I2C bus and read/write routines are included in CustomIO.cpp.

One of the first tasks in any USB I/O device design is the creation of the descriptors. Figure 7 shows the descriptors for this second example.

```
// Declare the Descriptor Tables for “Buttons and Lights” Example
//
const byte ReportDescriptor[] = {
       6, 0, 255,     // Usage_Page (Vendor Defined)
       9, 1,          // Usage (I/O Device)
       0xA1, 1,       // Collection (Application)
       0x19, 1,       //   Usage_Minimum (Button 1)
       0x29, 8,       //   Usage_Maximum (Button 8)
       0x15, 0,       //   Logical_Minimum (0)
       0x25, 1,       //   Logical_Maximum (1)
       0x75, 1,       //   Report_Size (1)
       0x95, 8,       //   Report_Count (8)
       0x81, 2,       //   Input (Data,Var,Abs) = Buttons Value
       0x19, 1,       //   Usage_Minimum (LED 1)
       0x29, 8,       //   Usage_Maximum (LED 8)
       0x91, 2,       //   Output (Data,Var,Abs) = LEDs (5 bits)
       0xC0           // End_Collection
       };

const DeviceDescriptorType DeviceDescriptor = {
       18, 1, 0, 1, 0, 0, 0, EP0Size, 0x42, 0x42, 16, 2, 0, 1, 1, 2, 0, 1
       };
typedef struct {
       ConfigurationDescriptorType Configuration;
       InterfaceDescriptorType Interface;
       ClassDescriptorType Class;
       EndPointDescriptorType EPin;
       } ButtonsAndLightsConfigurationDescriptorType;

const ButtonsAndLightsConfigurationDescriptorType My = {
// Base Configuration Descriptor Follows
       9, 2, sizeof(ButtonsAndLightsConfigurationDescriptorType), 0, 1, 1, 0, 0xC0, 50,
// Interface Descriptor Follows
       9, 4, 0, 0, 1, 3, 0, 0, 3,
// HID Class Descriptor Follows
       9, 0x21, 0x10, 1, 0, 1, 0x22, sizeof(ReportDescriptor), 0,
// Endpoint Descriptors Follows, this HID example uses INT IN
       7, 5, 0x81, 3, EP1Size, 0, 100
       };

// Define Strings as UNICODE and create descriptors at runtime
wchar String0[] = {0x0409, 0x0000};
wchar String1[] = L"USB Design By Example";
wchar String2[] = L"Buttons & Lights";
wchar String3[] = L"HID interface";
wchar *String[] = {String0, String1, String2, String3};
byte StringSize[] = { sizeof(String0), sizeof(String1), sizeof(String2), sizeof(String3) };
```

Figure 7 - Descriptors for Buttons and Lights example

The interface descriptor defines a HID class device so a report descriptor and one (two optional) data endpoints will be required. The report descriptor defines a vendor specific I/O device with a one byte input report and a one byte output report.

When we run the “PC as Buttons and Lights” program on the device PC it will attach to the host PC as a generic HID. (You could open the host PC’s “Control Panel” applet and view the “System” configuration to observe this attached device).

The changes to Main() for handling the data reports on endpoint 1 are small. In fact, we have a completely different I/O device and we changed less then 5% of the program. This re-usability of code will make the construction of other USB devices very efficient.

I wrote a custom application program to access our custom hardware using Visual Basic. The source code is provided and its human interface is shown in Figure 8. The program first searches the operating system tables for the device and then opens it as a standard HID device. Reports are sent using Writefile() and are read using Readfile() – the operating system handles all of the low-level communications for us.

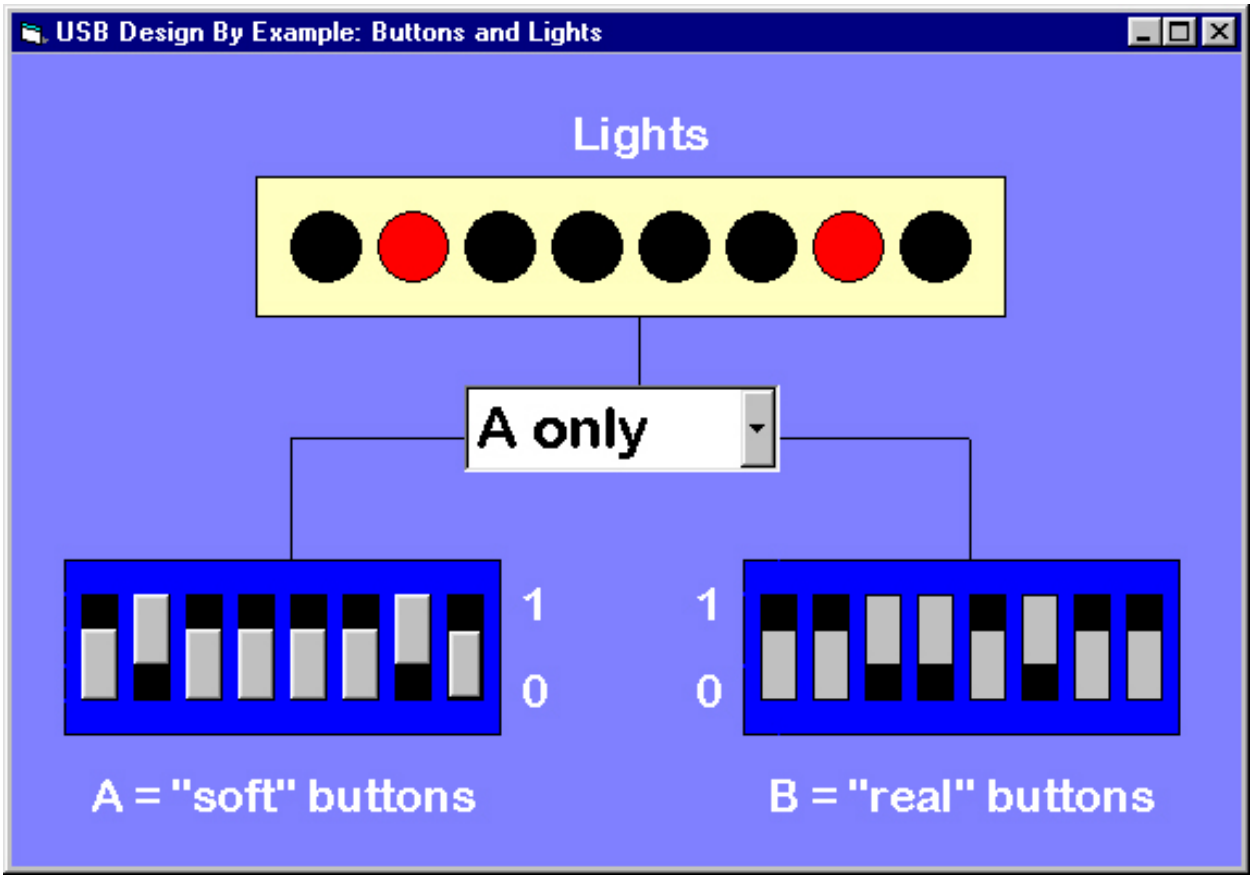

Figure 8 - Buttons and Lights Host PC application

You can click the "soft-buttons" on the host PC's screen to change the I/O device LED display or you can switch the buttons on the I/O device and see changes on the host PC's screen. We have software interacting with real-world hardware.

Let's recap the lessons learned from the second example

- Functionality of an I/O device is defined by descriptors
- The code to implement this functionality to common for all USB devices – this produces reusable code
- Controlling custom I/O devices is straightforward
- Writing a Host PC application to interact with custom hardware can use the HID class driver as an efficient, low data-rate, transport mechanism

Example 3 – a Printer

Many low data-rate devices (8KB/sec or less) can use the built-in HID drivers to simply implement a custom I/O device. Some applications need a higher data rate and will use the bulk data transfer mechanisms of USB. To illustrate this higher throughput application I chose to implement a printer I/O device that will use the built-in printer class driver. (Again, no operating system software to write!)

The changes to Main() are, again, quite small. I accumulate the received data in a local buffer and display this on the device PC screen once the data transfer is complete. There are several changes to the descriptors as shown in Figure 9.

```
// Descriptors.h : The Descriptor Tables for this Printer Example
//
const DeviceDescriptorType DeviceDescriptor = {
       18, 1, 0, 1, 0, 0, 0, EP0Size, 0x42, 0x42, 0x30, 1, 0, 1, 1, 2, 0, 1
       };
typedef struct {
       ConfigurationDescriptorType Configuration;
       InterfaceDescriptorType Interface;
       EndPointDescriptorType EPin;
       EndPointDescriptorType EPout;
       } PrinterConfigurationDescriptorType;

const PrinterConfigurationDescriptorType My = {
// Base Configuration Descriptor Follows
       9, 2, sizeof(PrinterConfigurationDescriptorType), 0, 1, 1, 0, 0xC0, 50,
// Interface Descriptor Follows: a bi-directional printer
       9, 4, 0, 0, 2, 7, 1, 2, 3,
// Endpoint Descriptors Follows, this Printer example uses Bulk IN1 and Bulk OUT1
       7, 5, 0x81, 2, EP1Size, 0, 0,
       7, 5,    1, 2, EP1Size, 0, 0
       };

// Define Strings as UNICODE and create descriptors at runtime
wchar String0[] = {0x0409, 0x0000};
wchar String1[] = L"USB Design By Example";
wchar String2[] = L"Printer using a PC ";
wchar String3[] = L"Printer interface";
wchar *String[] = {String0, String1, String2, String3};
byte StringSize[] = { sizeof(String0), sizeof(String1), sizeof(String2), sizeof(String3) };

byte IEEE1284String[] = "??MANUFACTURER:USB Design By Example;MODEL:PC as a
Printer;CLASS:PRINTER;";  // First two byte's will be overwritten
```

Figure 9 - Descriptors for Printer example

The interface descriptor defines a PRINTER class device which will be matched by MSPRINT.INF to point to USBPRINT.SYS. I chose to define a bi-directional printer so I must include a BULK IN and a BULK OUT endpoint descriptor.

When the "PC as Printer" program is started it will attach to the host PC which will enumenerate it as a standard printer device. Once the low-level, USBPRINT.SYS is loaded a request will be made for a high-level printer driver – I recommend that you specify a generic, text only driver for this example. The last stage of this high-level driver installation will ask if you would like to print a test page – selecting "yes" will cause the test page to be displayed on the device PC's screen. We have implemented a bulk transfer device!

Further experiments
You can edit the descriptors to create ***any*** USB I/O device type. I created the descriptors for an audio device (8 bit mono speaker) before I realized that the Philips D11 does not support isochronous data transfers. The partial example enumerates correctly as a USB audio device but data cannot be transferred due to the hardware limitations.

I would encourage you to define a Mass Storage Class driver using bulk-only transport. This will be similar to the printer example except that binary data transfer (as compared with text data transfer) will be possible. The I/O device will look like an attached disk drive to the host PC.

There is nothing that you can "break" with your experiments. The only precaution I would recommend is the frequent back up and restore of the registry on the Host PC. It is easy to create erroneous registry entries during experimentations and these are difficult to remove. It is safer to restore the registry from a known good backup.

Happy developing!

References:
[1] For a more detailed explanation of USB, I wrote a 500 page book called "USB Design By Example" – this is available online at www.intel.com/intelpress/usb. There are also many more examples on this web site.
[2] These adaptors are available from www.devasys.com.