Design example: a USB pointing device

Implementing a USB mouse using a generic microcontroller and a USB Interface IC, Philips PDIUSBD11.

Introduction

Adding a USB interface to your peripheral device may not be the easiest task, but with the PDIUSBD11 it can be made simpler.

This document shows by example how you can implement a USB system using the PDIUSBD11.

Selecting a USB Solution

There are already many USB IC devices available on the market. They can be collectively classified into three categories. The first category would be a USB microcontroller. The second type is a one-chip solution that implements a specific USB function. The third and last category belongs to the USB interface IC.

The data throughput, endpoint requirements and data delivery constraints impact what kind of solution you can choose. The following table displays the maximum effective data rate in increasing order of data throughput.

Table I : Effective Data transfer rate of USB device

Buffer Size	Low Speed data rate (bytes per frame)		Full Speed data rate (bytes per frame)			
	Control	Interrupt	Control	Interrupt	Bulk	ISO
1	3	1 (0.1Kbytes/sec)	32	1 (1 Kbytes/sec)	107	150
2	6	2 (0.2 Kbytes/sec)	62	2 (2 Kbytes/sec)	200	272
4	12	4 (0.4 Kbytes/sec)	120	4 (3 Kbytes/s)	352	460
8	24	8 (0.8 Kbytes/sec)	224	8 (8 Kbytes/s)	568	704
16	----------	----------	384	16 (16 Kbytes/s)	816	960
32			608	32 (32 Kbytes/s)	1056	1152
64			832	64 (64 Kbytes/s)	1216	1280
128			--------	----------	----------	1280
256						1280
512						1024
1023						1023

When operating at full speed, a shielded USB cable must be used. Apart from that, the only difference the physical layer dictates is the allowable bandwidth each device can use on the bus - at full speed, the useful data throughput rate is higher.

The colored region shows what the PDIUSBD11 can deliver. However, the user must also consider the data rate of I^2C which may bottleneck the throughput. PDIUSBD11 can run at an I^2C clock speed of 1 MHz. Allowing for the overhead in accessing the data registers of the D11, this translates to a maximum effective data rate of 40Kbytes/sec.

The benefits of using a USB interface IC are:

1. Lower costs of USB interface ICs versus high-end USB micro-controllers (full speed microcontrollers generally belong to the higher end of micro-controllers)
2. Specific features required of the micro-controllers that have been tailored to fit your specific system.
3. Lower development costs and avoidance of risks involved in re-writing code for a new microcontroller.

The trade-offs in using a USB interface IC like PDIUSBD11 are:

1. The requirement to use I^2C to communicate with the PDIUSBD11, including the required I/O pins for Interrupt and Suspend signals.
2. Requirement for some additional ROM space on your micro-controller.
3. Possible limitations in the I^2C data rate

The hardware

The hardware for this project was selected to target an audience whose wishes are to learn about developing USB peripheral devices. For tutorial purposes, mouse functionality was selected as the design goal, because the hardware involved is simple and host device drivers are already available on Windows 98.

The schematic diagram for the application, consisting mainly of a microcontroller, a scan matrix, the USB interface, and finally a supply regulator, is as shown in Fig.1.

A standard scan matrix implementing 9 buttons is used. The scan outputs are connected to Port2.5 through Port2.7 of the microcontroller, and the scanned inputs are read in from Port2.2 through Port 2.4.

This device will be designed for bus-powered operation. Therefore, the system will have to be limited to consume no more than 500μA of current when it is in the suspend state. To achieve that, the micro-controller has to be in the Power-down state. The current consumption for the micro-controller under the power-down state goes to a 50 μA maximum value. Adding the suspend current for PDIUSBD11 and the mandatory internal pull-up resistor on the D+ line, we are well under the 500 μA limit. When the device is in suspend state, the I2C lines are inactive. Yet, all interrupts will have to be serviced, so the INT_N should be high. SUSPEND is an active high signal. The pull-up resistors on these lines do not contribute to the total current consumption.

Waking up the micro-controller from the power-down state involves pulling the INT0 (pin 3.2) of the micro-controller LOW. There are two scenarios in which this occurs. The first possibility is a user-initiated wakeup which is done through the WAKEUP button. After the micro-controller senses a low on its external interrupt INT0 (pin P1.5 on microcontroller), it wakes the micro-controller. The micro-controller will need to issue a Remote Wakeup command to the USB host. This is performed via a command register afforded by the PDIUSBD11. The second possibility for a wakeup is one where the host, or another device on the system, initiates the wakeup. When that happens, PDIUSBD11 detects the event and holds down the SUSPEND pin as soon as traffic is resumed on the USB bus. This wakes up the micro-controller, which can then continue its normal operation.

The clock of the micro-controller is supplied by the PDIUSBD11. The frequency at power-up is 4MHz whereafter the microcontroller can change the operating frequency to a value as high as 48MHz. The frequency of operation used here is 12MHz. An inverter is used to buffer between the PDIUSBD11, which gives out the clock at 3.3V logic, and the micro-controller. A PicoGate™ is used to save area in the PCB layout.

The USB Interface IC used here, the Philips PDIUSBD11, takes care of the USB hardware and protocol layers and communicates with a micro-controller via the I^2C Bus. The user need only to connect the usual SDA, SCL pins for the I²C, as well as pins for Interrupt, INT_N and SUSPEND (assuming the system is

power-constrained). The USB signals consist mainly of the D+ and D- lines which the user connects to the USB Type B connector. As the PDIUSBD11 is ESD-robust to 8kV, no additional ferrite beads or diodes are needed to protect the D+/D- lines. In fact, the ferrite beads are unwanted since they would cause ringing on the signals during EOP (End-Of-Packet) signaling. The 1.5 Kohms pull-up resistor is not seen on the D+ line on the schematics because the pull-up is done internally through software control. This feature is termed SoftConnect[tm].

The Vbus pin from the type B connector is to be connected via a resistor to the Vbus sensing pin of the PDIUSBD11. When a 0V is sensed on this pin, the D+ pull-up resistor will automatically be disconnected. However in this bus-powered example, Vbus sensing is not critical because when Vbus is removed, so is the power for all the devices!

The power regulator is needed to convert the available 5V from the USB to a 3.3V supply voltage needed to operate the PDIUSBD11. The USB specification states that a bus-powered peripheral remains operational even when the Vbus input voltage drops to 4.4V. Since PDIUSBD11 can operate at a minimum voltage of 3V, the design constraints on the regulator are somewhat relaxed.

The firmware

The firmware was written in C and compiled using the Keil C DOS compiler. A pictorial representation of the component files used is shown below.

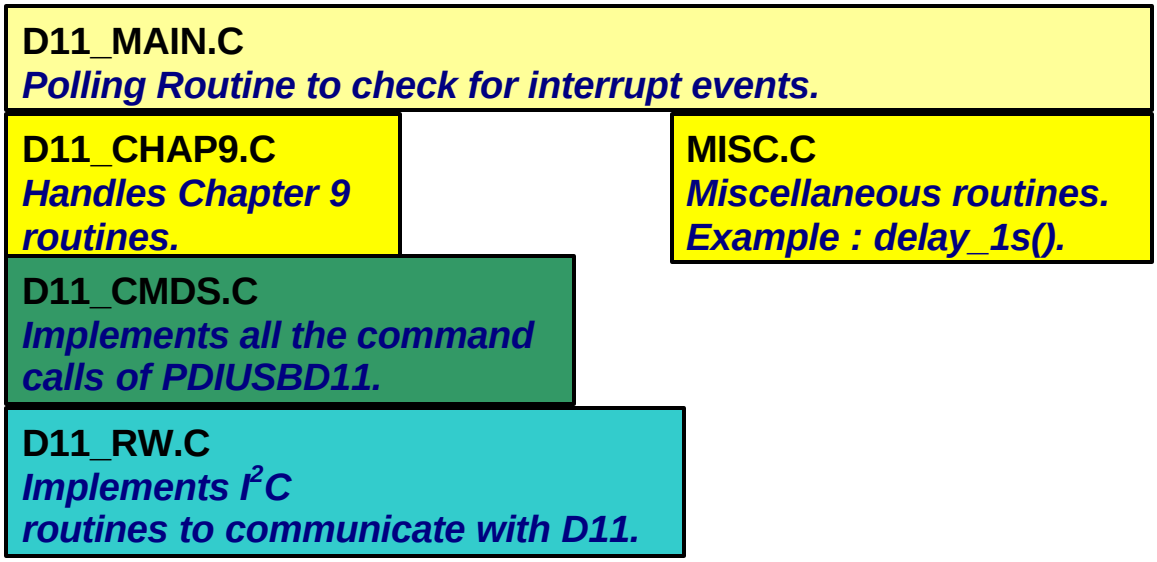


Diagram 1 : Layered functions and filenames for the firmware

The funtion calls are built in layered fashion. D11_RW.C contains some "bit-bang" routines to emulate the I^2C protocol. The header files for all function calls are listed in the file D11_RW.H. Below is a listing and brief explanation of all the functions used in the program:

```
extern void Wr_command(unsigned char D11_CDATA);
```

Sends a single command data to the I^2C command address at 0x1B. The command data is contained in the argument D11_CDATA.

```
extern void Rd_Ndata(unsigned char *POINTER, unsigned char N);
```

Sends the Read N data from the I2C Data Address at 0x1A. All the data read is stored in the array referenced by POINTER. N indicates the number of data bytes to read.

extern void Wr_Ndata(unsigned char *POINTER, unsigned char N);

Writes N data referenced by POINTER to I2C Data Address at 0x1A.

extern void Rd_Bufdata(unsigned char *POINTER);

Reads Data from the data buffer at 0x1A. However, the number of data bytes to strobe in is derived from the second byte of the I2C data. All the data bytes (D1,D2.., DN) read are stored into the array referenced by POINTER.

I2C stream – 0x35, 0x??, 0xNN, D1,..,DN.

extern void Wr_Bufdata(unsigned char *POINTER);

Writes Data contained in the array referenced by POINTER to the I2C address 0x1A. The number of data bytes to strobe in is derived from the 1st byte.

Example : POINTER[5] = {3,1,2,3}

I2C Stream : 0x34, 0x00, 0x03, 0x01, 0x02, 0x03.

PDIUSBD11 related commands are coded in the file D11_CMD.C. The associated header file is D11_CMD.H Most of the function names are self-explanatory. In this file, 9 bytes are data bytes reserved for the I2C buffer.

extern void SetMode(unsigned char D11_MDATA, unsigned char D11_CLKDIV);

Set the configuration byte and the clock divider byte for the PDIUSBD11. The new configuration byte is the first argument D11_MDATA and the second argument, D11_CLKDIV, contains the new CLOCK DIVIDER.

extern void SetAddress(unsigned char NewAddr);

This sets and enables the new address. The enable bit is the MSB of NewAddr.

extern void DisOtherAddress(void);

This disables the Hub address. The PDIUSBD11 shares the same digital core as that of the PDIUSBH11A which includes a hub. This command disables the (vacant) Hub address. It is essential that this command be used prior to activating the SoftConnect feature.

extern void SetEndptEnable(unsigned char Enable_data);

This enables the D11 endpoints. To enable, write 0x02; to disable, write 0x00.

extern void UpdateIntReg(void);

This reads the Interrupt register. The interrupt register is then stored into two registers: ENDPT_INT and OTHER_INT.

To check if either the main endpoint or endpoint 0 has an interrupt pending,you can “AND” the ENPT_INT with the following:

CTRL_ENDPT_OUT 0x04 // bit 2
CTRL_ENDPT_IN 0x08 // bit 3

```
ENDPT1_IN       0x10 // bit 4
ENDPT1_OUT      0x20 // bit 5
ENDPT2_OUT      0x40 // bit 6
ENDPT2_IN       0x80 // bit 7
```

Or if the interrupt comes from other interrupt sources:

```
#define ENDPT3_OUT      0x01 // bit 1
#define ENDPT3_IN       0x02 // bit 2
#define BUS_RESET       0x40 // bit 6
```

extern void ReadBuffer(void);

This reads the endpoint data Buffer into DBUFFER[9]. The first byte always stores the number of valid data bytes.

extern void WriteBuffer(void);

This writes the data to the data Buffer from DBUFFER[9]. The first byte stores the number of data bytes to be written.

extern void Ack_SETUP(void);

When a SETUP Token is received, the data in the endpoint buffer is locked. This means that a ClearBuffer command or a Validate command has no effect till the Ack_SETUP is issued. This is always issued after a SETUP token is received. Only then would ClearBuffer work.

extern void SendResume(void);

This issues a Remote Wakeup signal to the Host.

extern unsigned char SelectEndpoint(unsigned char ENDPT);

The SelectEndpoint(ENDPT) is used to select the endpoint to be read or to be written to. The possible arguments are:

```
#define CTRL_OUT 2 //Endpoint Index
#define CTRL_IN  3  //Endpoint Index
#define ENDPOINT1_OUT 5 //Endpoint Index
#define ENDPOINT1_IN  4 //Endpoint Index
#define ENDPOINT2_OUT 6 //Endpoint Index
#define ENDPOINT2_IN  7 //Endpoint Index
#define ENDPOINT3_OUT 8 //Endpoint Index
#define ENDPOINT3_IN  9 //Endpoint Index
```

extern unsigned char ReadEndptStatus(unsigned char ENDPT);

This reads the Last Transaction Status and clears the interrupt bit corresponding to the endpoint. The possible arguments are:

```
#define CTRL_OUT 2 //Endpoint Index
#define CTRL_IN  3  //Endpoint Index
#define ENDPOINT1_OUT 5 //Endpoint Index
#define ENDPOINT1_IN  4 //Endpoint Index
#define ENDPOINT2_OUT 6 //Endpoint Index
#define ENDPOINT2_IN  7 //Endpoint Index
#define ENDPOINT3_OUT 8 //Endpoint Index
```

#define ENDPOINT3_IN 9 //Endpoint Index

extern void SetEndptStatus(unsigned char ENDPT, unsigned char ENDPTDATA);

This allows an individual endpoint to be stalled in response to an unidentified or unsupported USB request. The ENDPT numbers are:

```
#define CTRL_OUT 2 //Endpoint Index
#define CTRL_IN  3  //Endpoint Index
#define ENDPOINT1_OUT 5 //Endpoint Index
#define ENDPOINT1_IN  4 //Endpoint Index
#define ENDPOINT2_OUT 6 //Endpoint Index
#define ENDPOINT2_IN  7 //Endpoint Index
#define ENDPOINT3_OUT 8 //Endpoint Index
#define ENDPOINT3_IN  9 //Endpoint Index
```

To stall, put 0x01 into ENDPTDATA. To un-stall the endpoint, put 0x00.

extern void ClearBuffer(void);

This clears the buffer such that the subsequent DATA can be “ACK”ed (received from the Host).

extern void ValidateBuffer(void);

This command is issued when the endpoint buffer has been filled with the data to be written to the Host in. The Validate command tells PDIUSBD11 that the data stored in the endpoint buffer is ready to be accepted upon issue of an “IN” token.

extern void SendZeroPacket(unsigned char ENDPT);

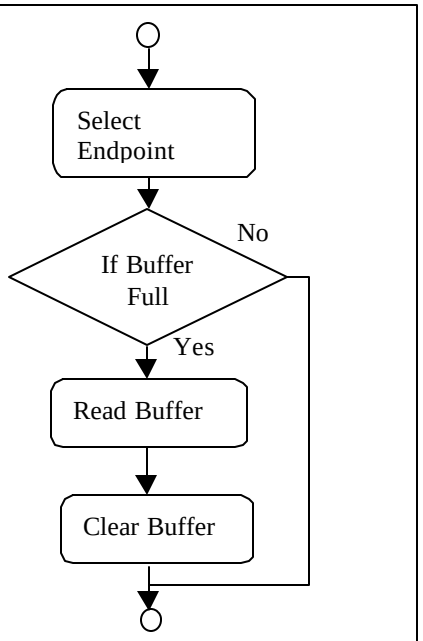
This command sends a zero packet data to the host. Basically it writes to an IN endpoint, then writes a zero length data packet, followed by writing the validate command.

extern unsigned char ReadEPStall(unsigned char ENDPT);

With this command you can read the status of an endpoint to check if it has been stalled.

Procedures to Read and Write data to Host

The PDIUSBD11 buffers the hardware handshaking. The Chapter 9 handshake protocol is to be implemented by the user. Hence, it is essential that the basic data reading/writing sequence is understood.



Flow Chart 1 : Reading Data from the Host

Reading Data

The Select Endpoint re-initializes the PDIUSBD11 internal pointer to where the endpoint buffer starts. The return status bit also shows if the endpoint if full or empty.

A ReadBuffer command followed by strobing in the data gets the actual data stored in the endpoint. The first byte received is a reserved byte. The second byte indicates the number of valid data bytes in the buffer, this is followed by the actual data.

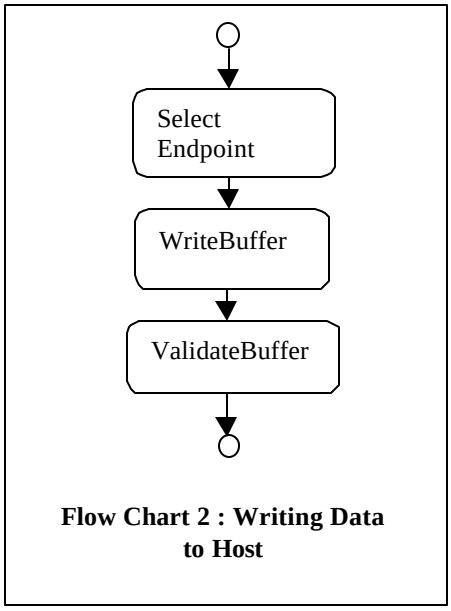
ClearBuffer allows the endpoint to be cleared such that the next packet that comes by through the USB traffic will be accepted and acknowledged.

Writing Data

The endpoint is first selected to re-initialize the internal pointer to the buffer.

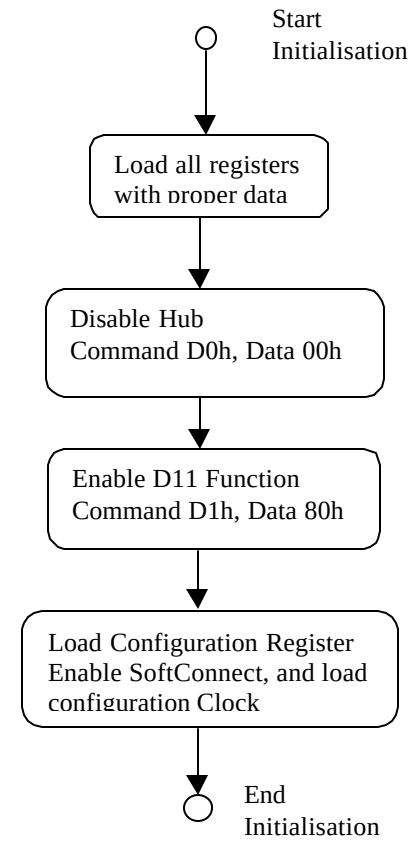
The data is strobed into the endpoint buffer. The first data byte is a reserved byte; the second byte indicates how many actual data bytes contained in the buffer are to be sent to the host.

ValidateBuffer informs PDIUSBD11 that the buffer is now fully loaded and ready to send the data out.



Flow Chart 2 : Writing Data to Host

The following shows a flow diagram of the initialization routine.

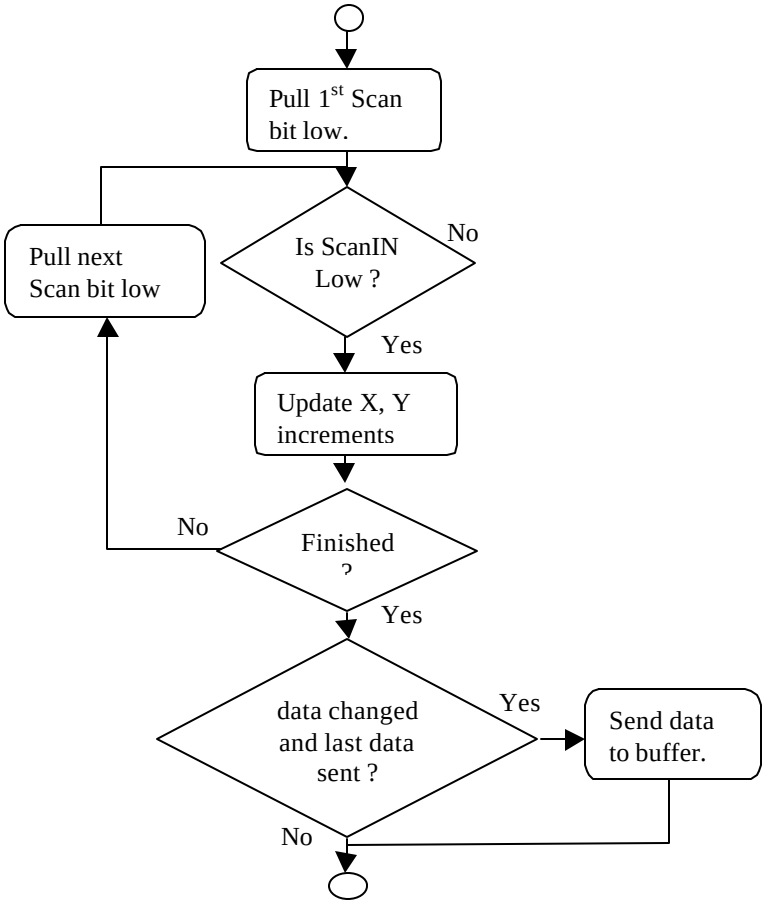


Flow Chart 3 : Initialisation Routine (D11_MAIN.C)

As the PDIUSBD11 shares the same digital core as the PDIUSBH11A but does not contain a hub, it is required to “disable” the hub through the Set Address command 0xD0. The next Set Address command 0xD1enables the D11 function.

The SoftConnect feature is enabled through the SetMode command 0xF3.

Scanning for User Keypressed



Flow Chart 4 : Button Scanning Routine (D11_MAIN.C)

A common scanning routine is used. Each scan line is pulled down and the scanned input is read to check If any key has been pressed.

On detection of a "button pressed" event, the corresponding increment or decrement of the X or Y movement is tracked.

Once a complete scan has been done, the data is sent to the Host (provided the previous data has been removed).

The data being sent should follow what was described on the Report Descriptor.

The first byte contains the information as to whether there has been a user "click", the second byte shows the amount of horizontal movement and the third byte shows the amount of vertical movement.

Chapter 9 standard requests

All the USB Standard requests are implemented based on D11_CHAP9.C and its associated file CHAP9.DES.

There are 2 main functions that will be called by the main program USB_MAIN.C. They are :

```
void ProcessCtrlInEP(void);
void ProcessCtrlOutEP(void);
```

Whenever there data is going through Endpoint 0 and when the direction is from the host to the device, `ProcessCtrlOutEP(void)` would be called. This includes the Setup Tokens. On receipt of a Setup Data, the information is re-packaged into a structure called REQUEST shown below:

```
typedef struct REQUEST { BYTE bmRequestType;
                  BYTE bRequest;
                  WORD wValue;
                  WORD wIndex;
                  WORD wLength;    // Data Phase's data length
                  };
```

This would be passed to the handling routines :

```
      Void D11StandardRequest(struct REQUEST *pReq)
      void D11ClassRequest(struct REQUEST *pReq)
      void D11VendorRequest(struct REQUEST *pReq)
```

In this example, in addition to implementing D11StandardRequest, a class handler for the HID class is also implemented. The function call to it is D11ClassRequest. Alternately, a D11VendorRequest could be implemented by the user if required.

The descriptors are stored in file CHAP9.DES. Therefore one can easily modify basic information such as the Device Descriptors, Configuration Descriptors, Endpoints Descriptors, HID descriptors and the String descriptors.

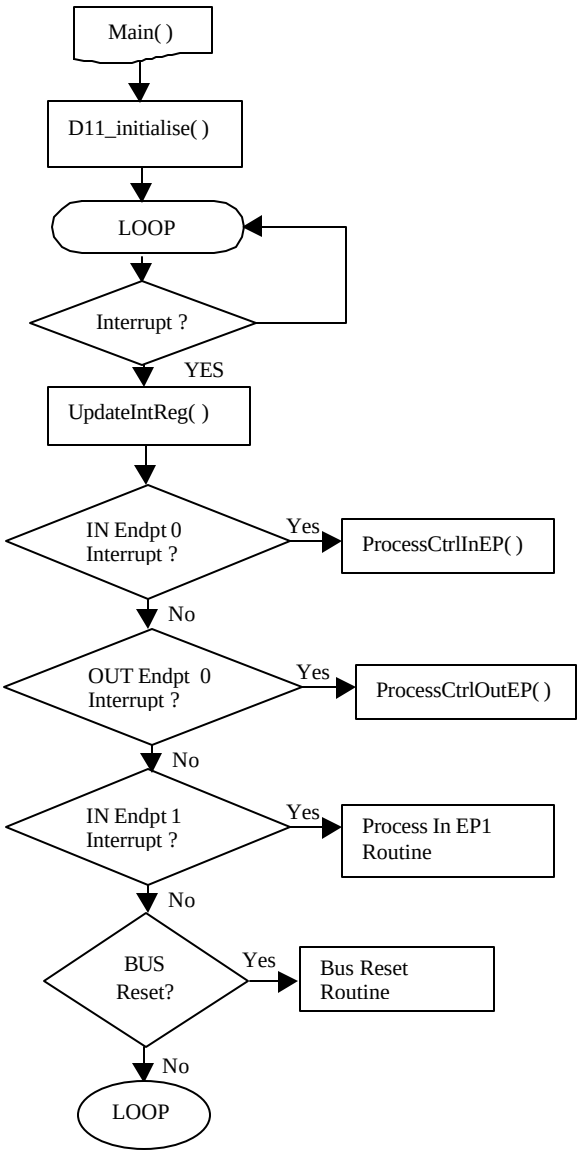
```
struct DEVICE { BYTE bLength;
             BYTE bDescriptorType;
             WORD bcdUSB;
             BYTE bDeviceClass;
             BYTE bDeviceSubClass;
             BYTE bDeviceProtocol;
             BYTE bMaxPacketSize;
             WORD idVendor;
             WORD idProduct;
             WORD bcdDevice;
             BYTE iManufacturer;
             BYTE iProduct;
             BYTE iSerialNumber;
             BYTE bNumConfigurations;
             };

struct CONFIGURATION { BYTE bLength;
             BYTE bDescriptorType;
             BYTE wTotalLength;
             BYTE wZero;
             BYTE bNumInterfaces;
             BYTE bConfigurationValue;
             BYTE iConfiguration;
             BYTE bmAttributes;
             BYTE MaxPower;
             };

struct INTERFACE { BYTE bLength;
             BYTE bDescriptionType;
             BYTE bInterfaceNumber;
             BYTE bAlternateSetting;
             BYTE bNumEndpoints;
             BYTE bInterfaceClass;
             BYTE bInterfaceSubClass;
             BYTE bInterfaceProtocol;
```

```
            BYTE iInterface;
            };

struct ENDPOINT { BYTE bLength;
            BYTE bDescriptorType;
            BYTE bEndpointAddress;
            BYTE bmAttributes;
            WORD wMaxPacketSize;
            BYTE bInterval;
            };

struct HIDDESC {
            BYTE bLength;
            BYTE bDescriptorType;
            WORD bcdHID;
            BYTE bCountryCode;
            BYTE bNumDescriptors;
            BYTE bReportDescriptorType;
            WORD wItemLength;
            };

struct CONFIG { struct CONFIGURATION sConfig;
                struct INTERFACE sInterface;
                struct HIDDESC sHIDDesc;
                struct ENDPOINT sEndpoint1;
             };
```

The report descriptor coded here uses the mouse report descriptor as documented in the HID1.0 - final. It is shown here for completeness. The data on the right column are sent to the Host on a Get_Report_Descriptor request.

E.10 Report Descriptor (Mouse)

Item	Value (hex)
Usage Page (Generic Desktop),	05 01
Usage (Mouse),	09 02
Collection (Application),	A1 01
Usage (Pointer),	09 01
Collection (Physical),	A1 00
Usage Page (Buttons),	05 09
Usage Minimum (01),	19 01
Usage Maximun (03),	29 03
Logical Minimum (0),	15 00
Logical Maximum (1),	25 01
Report Count (3),	95 03
Report Size (1),	75 01
Input (Data, Variable, Absolute), ;3 button bits	81 02
Report Count (1),	95 01
Report Size (5),	75 05
Input (Constant), ;5 bit padding	81 01
Usage Page (Generic Desktop),	05 01
Usage (X),	09 30
Usage (Y),	09 31
Logical Minimum (-127),	15 81
Logical Maximum (127),	25 7F
Report Size (8),	75 08
Report Count (2),	95 02
Input (Data, Variable, Relative), ;2 position bytes	81 06
(X & Y)	C0
End Collection,	C0
End Collection	



Flow Chart 5 : Main Loop for USB (D11_MAIN.C)

The main program is a polling routine. On retrieval of data, the interrupt pin of the PDIUSBD11 goes low.

The firmware checks for the interrupt source and diverts the program flow to the respective endpoint handler.

In the ProcessCtrlOutEP(), the steps to read a Setup command is as follows.

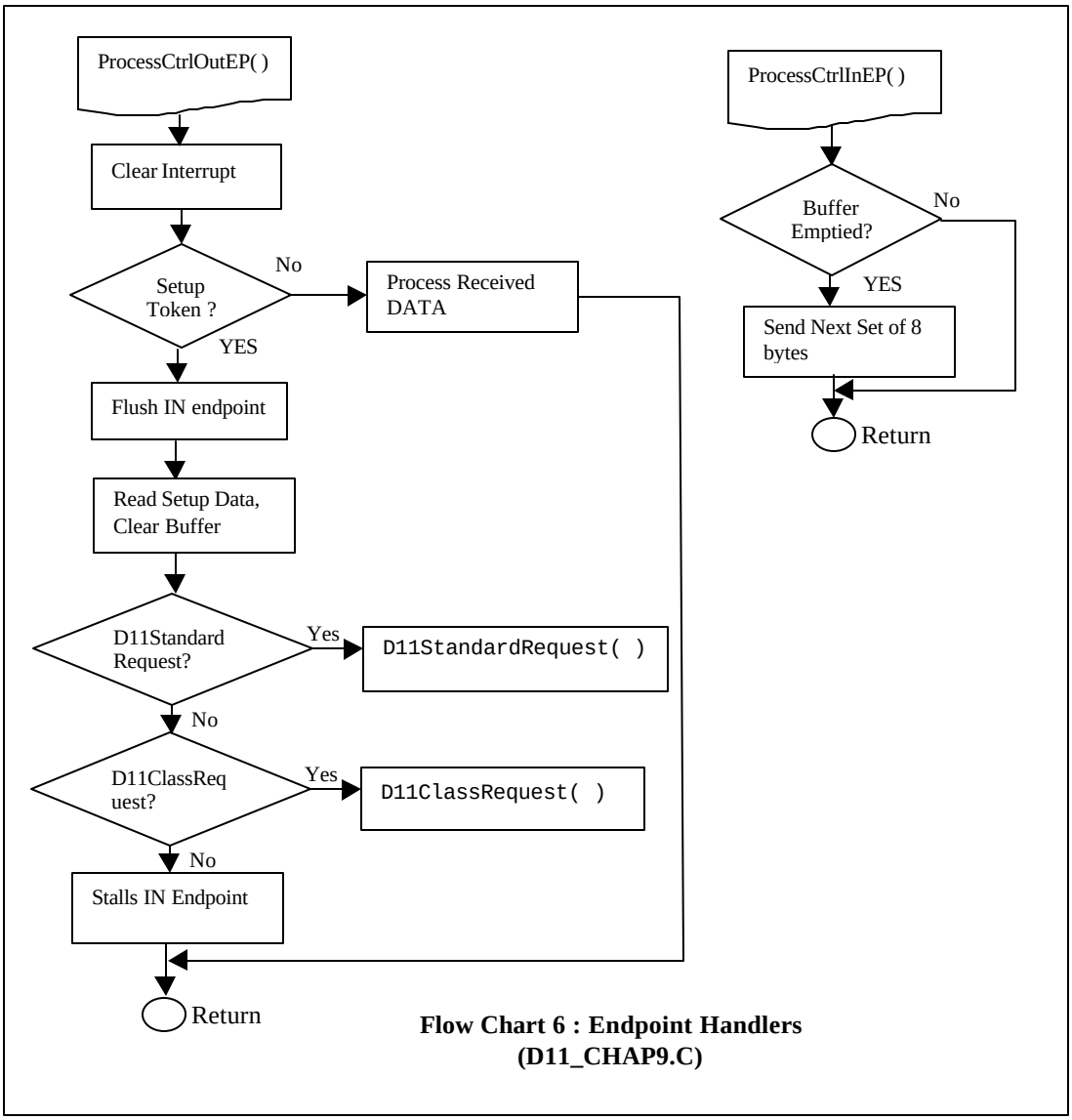
A ReadLastTransaction command is issued to clear the respectively endpoint.

It is checked to see if the current received status shows a Setup packet being given. If so, the Setup endpoints have to be "Ack"ed. This is to prevent already validated data from being sent in response to the current Setup token.

The justification for this is based on the USB protocol that allows any Setup token to supersede an uncompleted Setup request. Thus, to safeguard the accuracy of the handshake, once a Setup token is received, the data is received and locked. The other endpoint buffers are then frozen.

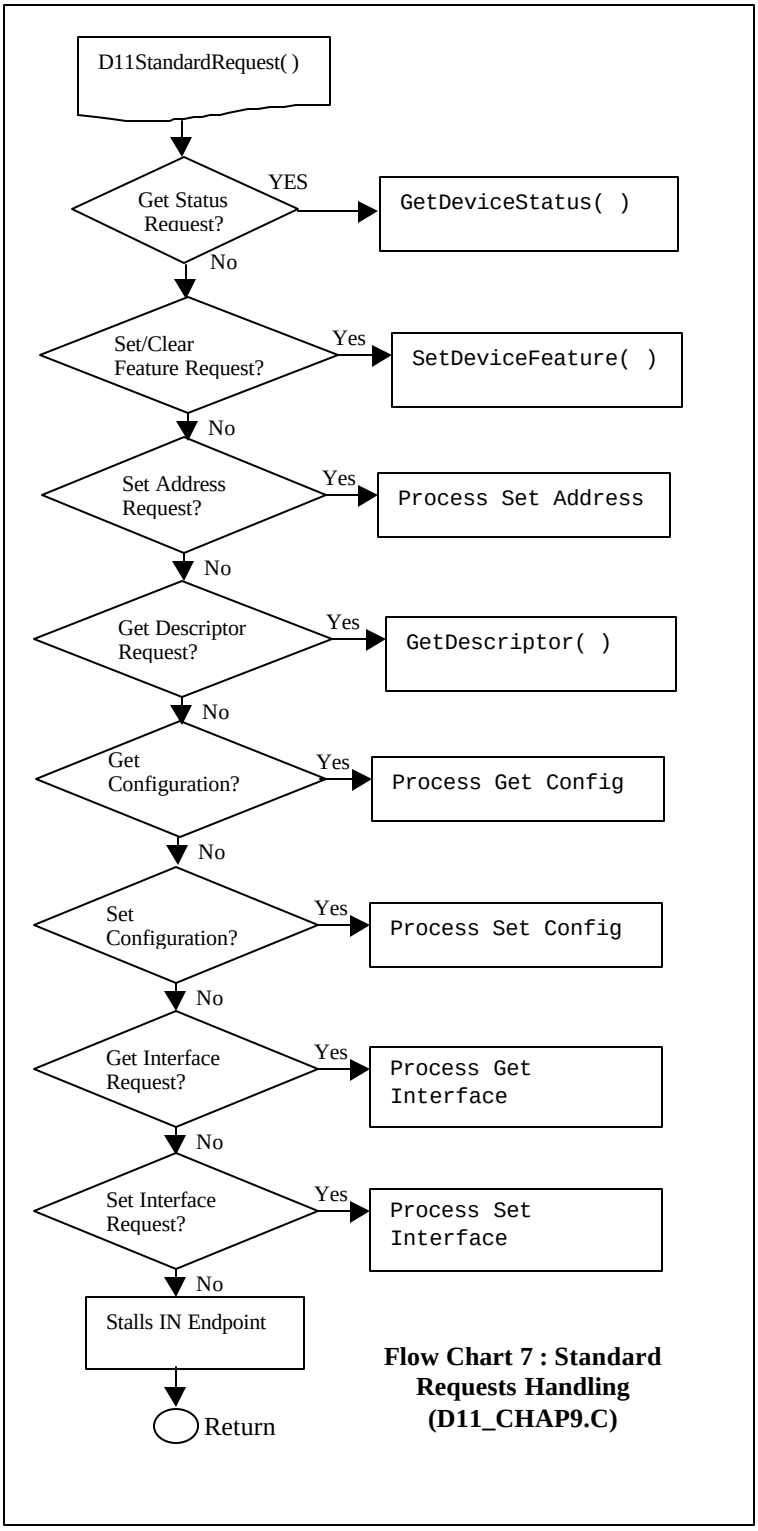
To unfreeze the endpoints, we use the Ack_Setup command. Hence, on every receipt of a Setup packet, the IN endpoint is selected. The Ack_Setup Command is given followed by a clearing of the buffer.

The Ack_Setup command is also issued on the CONTROL_IN buffer. The data is read and deciphered before routing it to either the StandardChapter9 or the ClassHandler.



Flow Chart 6 : Endpoint Handlers (D11_CHAP9.C)

Because the PDIUSBD11 has only buffer size of 8 bytes, long descriptors have to be divided into packets of 8 bytes and sent out one packet at a time. The firmware needs to keep track of the current data packet being sent out and to ensure that a zero packet data is sent on the final packet if the descriptor size is a factor of 8.



Flow Chart 7 : Standard Requests Handling (D11_CHAP9.C)

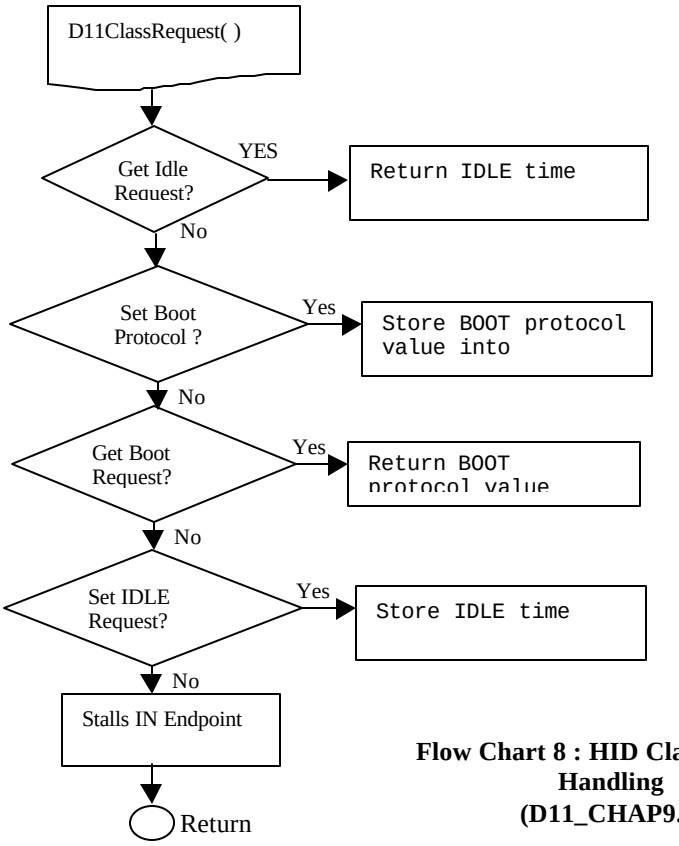
All the Chapter 9 standard requests are largely unchanged for all devices.

Except, a HID device is now required to handle the addition GetReportDescriptor. And also, the data expected from the GetConfigurationDescriptor includes the additional HID descriptor embedded between the Interface and the endpoint descriptors.

Conclusion

A board was wired to implement a USB mouse using the PDIUSBD11. The source code for the implementation is available.

The device was tested to enumerate and function properly on Windows 98.



Flow Chart 8 : HID Class Request Handling (D11_CHAP9.C