USB in a NutShell

Making Sense of the USB Standard

Enumeration

Enumeration is the process of determining what device has just been connected to the bus and what parameters it requires such as power consumption, number and type of endpoint(s), class of product etc. The host will then assign the device an address and enable a configuration allowing the device to transfer data on the bus. A fairly generic enumeration process is detailed in section 9.1.2 of the USB specification. However when writing USB firmware for the first time, it is handy to know exactly how the host responds during enumeration, rather than the general enumeration process detailed in the specification.

A common Windows enumeration involves the following steps,

1. The host or hub detects the connection of a new device via the device's pull up resistors on the data pair. The host waits for at least 100ms allowing for the plug to be inserted fully and for power to stabilise on the device.
2. Host issues a reset placing the device is the default state. The device may now respond to the default address zero.
3. The MS Windows host asks for the first 64 bytes of the Device Descriptor.
4. After receiving the first 8 bytes of the Device Descriptor, it immediately issues another bus reset.
5. The host now issues a Set Address command, placing the device in the addressed state.
6. The host asks for the entire 18 bytes of the Device Descriptor.
7. It then asks for 9 bytes of the Configuration Descriptor to determine the overall size.
8. The host asks for 255 bytes of the Configuration Descriptor.
9. Host asks for any String Descriptors if they were specified.

At the end of Step 9, Windows will ask for a driver for your device. It is then common to see it request all the descriptors again before it issues a Set Configuration request.

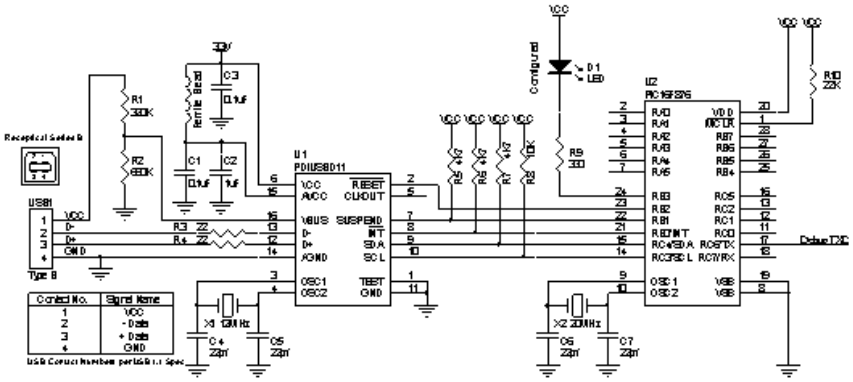
The above enumeration process is common to Windows 2000, Windows XP and Windows 98 SE.

Step 4 often confuses people writing firmware for the first time. The Host asks for the first 64 bytes of the device descriptor, so when the host resets your device after it receives the first 8 bytes, it is only natural to think there is something wrong with your device descriptor or how your firmware handles the request. However as many will tell you, if you keep persisting by implementing the Set Address Command it will pay off by asking for a full 18 bytes of device descriptor next.

Normally when something is wrong with a descriptor or how it is being sent, the host will attempt to read it three times with long pauses in between requests. After the third attempt, the host gives up reporting an error with your device.

Firmware - PIC16F876 controlling the PDIUSBD11

We start our examples with a Philip's PDIUSBD11 I2C Serial USB Device connected to a MicroChip PIC16F876 (shown) or a Microchip PIC16F877 (Larger 40 Pin Device). While Microchip has got two low speed USB PIC16C745 and PIC16C765 devices out now, they are only OTP without In-Circuit Debugging (ICD) support which doesn't help with the development flow too well. They do have four new full speed flash devices with (ICD) support coming. In the mean time the Philips PDIUSBD11 connected to the PIC16F876 which gives the advantage of Flash and In-Circuit Debugging.



Click here to enlarge

A schematic of the required hardware is shown above. The example enumerates and allows analog voltages to be read from the five multiplexed ADC inputs on the PIC16F876 MCU. The code is compatible with the PIC16F877 allowing a maximum of 8 Analog Channels. A LED connected on port pin RB3 lights when the device is configured. A 3.3V regulator is not pictured, but is required for the PDIUSBD11. If you are running the example circuit from an external power supply, then you can use a garden variety 78L033 3.3V voltage regulator, however if you wish to run the device as a Bus Powered USB device then a low dropout regulator needs to be sought.

Debugging can be done by connecting TXD (Pin 17) to a RS-232 Line Driver and fed into a PC at 115,200bps. Printf statements have been included which display the progress of enumeration.

The code has been written in C and compiled with the Hi-Tech PICC Compiler. They have a demo version (7.86 PL4) of the PICC for download which works for 30 days. A pre-compiled .HEX file has be included in the archive which has been compiled for use with (or without) the ICD.

```
#include <pic.h>
#include <stdio.h>
#include <string.h>
#include "usbfull.h"

const USB_DEVICE_DESCRIPTOR DeviceDescriptor = {
    sizeof(USB_DEVICE_DESCRIPTOR), /* bLength */
    TYPE_DEVICE_DESCRIPTOR,        /* bDescriptorType */
    0x0110,                        /* bcdUSB USB Version 1.1 */
    0,                             /* bDeviceClass */
    0,                             /* bDeviceSubclass */
    0,                             /* bDeviceProtocol */
    8,                             /* bMaxPacketSize 8 Bytes */
    0x04B4,                        /* idVendor (Cypress Semi) */
    0x0002,                        /* idProduct (USB Thermometer Example) */
    0x0000,                        /* bcdDevice */
    1,                             /* iManufacturer String Index */
    0,                             /* iProduct String Index */
    0,                             /* iSerialNumber String Index */
    1                              /* bNumberConfigurations */
};
```

The structures are all defined in the header file. We have based this example on the Cypress USB Thermometer example so you can use our USB Driver for the Cypress USB Starter Kit. A new generic driver is being written to support this and other examples which will be available soon. Only one string is provided to display the manufacturer. This gives enough information about how to implement string descriptors without filling up the entire device with code. A description of the Device Descriptor and its fields can be found here.

```
const USB_CONFIG_DATA ConfigurationDescriptor = {
    {                              /* configuration descriptor */
    sizeof(USB_CONFIGURATION_DESCRIPTOR), /* bLength */
    TYPE_CONFIGURATION_DESCRIPTOR, /* bDescriptorType */
    sizeof(USB_CONFIG_DATA),       /* wTotalLength */
    1,                             /* bNumInterfaces */
    1,                             /* bConfigurationValue */
    0,                             /* iConfiguration String Index */
    0x80,                          /* bmAttributes Bus Powered, No Remote Wakeup */
    0x32                           /* bMaxPower, 100mA */
    },
```

```
    {                           /* interface descriptor */
    sizeof(USB_INTERFACE_DESCRIPTOR), /* bLength */
    TYPE_INTERFACE_DESCRIPTOR,  /* bDescriptorType */
    0,                          /* bInterface Number */
    0,                          /* bAlternateSetting */
    2,                          /* bNumEndpoints */
    0xFF,                       /* bInterfaceClass (Vendor specific) */
    0xFF,                       /* bInterfaceSubClass */
    0xFF,                       /* bInterfaceProtocol */
    0                           /* iInterface String Index */
    },
    {                           /* endpoint descriptor */
    sizeof(USB_ENDPOINT_DESCRIPTOR),  /* bLength */
    TYPE_ENDPOINT_DESCRIPTOR,   /* bDescriptorType */
    0x01,                       /* bEndpoint Address EP1 OUT */
    0x02,                       /* bmAttributes - Interrupt */
    0x0008,                     /* wMaxPacketSize */
    0x00                        /* bInterval */
    },
    {                           /* endpoint descriptor */
    sizeof(USB_ENDPOINT_DESCRIPTOR),  /* bLength */
    TYPE_ENDPOINT_DESCRIPTOR,   /* bDescriptorType */
    0x81,                       /* bEndpoint Address EP1 IN */
    0x02,                       /* bmAttributes - Interrupt */
    0x0008,                     /* wMaxPacketSize */
    0x00                        /* bInterval */
    }
};
```

A description of the Configuration Descriptor and its fields can be found here. We provide two endpoint descriptors on top of the default pipe. EP1 OUT is an 8 byte maximum Bulk OUT Endpoint and EP1 IN is an 8 byte max Bulk IN Endpoint. Our example reads data from the Bulk OUT endpoint and places it in an 80 byte circular buffer. Sending an IN packet to EP1 reads 8 byte chunks from this circular buffer.

```
LANGID_DESCRIPTOR LANGID_Descriptor = { /* LANGID String Descriptor Zero */
    sizeof(LANGID_DESCRIPTOR),          /* bLenght */
    TYPE_STRING_DESCRIPTOR,             /* bDescriptorType */
    0x0409                              /* LANGID US English */
};

const MANUFACTURER_DESCRIPTOR Manufacturer_Descriptor = { /* ManufacturerString 1 */
    sizeof(MANUFACTURER_DESCRIPTOR),                      /* bLenght */
    TYPE_STRING_DESCRIPTOR,                               /* bDescriptorType */
    "B\0e\0y\0o\0n\0d\0 \0L\0o\0g\0i\0c\0"                /* ManufacturerString in UNICODE */
};
```

A Zero Index String Descriptor is provided to support the LANGID requirements of USB String Descriptors. This indicates all descriptors are in US English. The Manufacturer Descriptor can be a little deceiving as the size of the char array is fixed in the header and is not dynamic.

```
#define MAX_BUFFER_SIZE 80

bank1 unsigned char circularbuffer[MAX_BUFFER_SIZE];
unsigned char inpointer;
unsigned char outpointer;

unsigned char *pSendBuffer;
unsigned char BytesToSend;
unsigned char CtlTransferInProgress;
unsigned char DeviceAddress;
unsigned char DeviceConfigured;

#define PROGRESS_IDLE            0
#define PROGRESS_ADDRESS         3

void main (void)
{
    TRISB = 0x03;   /* Int & Suspend Inputs */
    RB3 = 1;        /* Device Not Configured (LED) */
    RB2 = 0;        /* Reset PDIUSBD11 */

    InitUART();
    printf("Initialising\n\r");
    I2C_Init();

    RB2 = 1;        /* Bring PDIUSBD11 out of reset */

    ADCON1 = 0x80;  /* ADC Control - All 8 Channels Enabled, */
                    /* supporting upgrade to 16F877 */

    USB_Init();
    printf("PDIUSBD11 Ready for connection\n\r");
    while(1)
        if (!RB0) D11GetIRQ(); /* If IRQ is Low, PDIUSBD11 has an Interrupt Condition */
}
```

The main function is example dependent. It's responsible for initialising the direction of the I/O Ports,

initialising the I2C interface, Analog to Digital Converters and PDIUSBD11. Once everything is configured it keeps calling D11GetIRQ which processes PDIUSBD11 Interrupt Requests.

```
void USB_Init(void)
{
    unsigned char Buffer[2];

    /* Disable Hub Function in PDIUSBD11 */
    Buffer[0] = 0x00;
    D11CmdDataWrite(D11_SET_HUB_ADDRESS, Buffer, 1);

    /* Set Address to zero (default) and enable function */
    Buffer[0] = 0x80;
    D11CmdDataWrite(D11_SET_ADDRESS_ENABLE, Buffer, 1);

    /* Enable function generic endpoints */
    Buffer[0] = 0x02;
    D11CmdDataWrite(D11_SET_ENDPOINT_ENABLE, Buffer, 1);

    /* Set Mode - Enable SoftConnect */
    Buffer[0] = 0x97; /* Embedded Function, SoftConnect, Clk Run, No LazyClk, Remote Wakeup */
    Buffer[1] = 0x0B; /* CLKOut = 4MHz */
    D11CmdDataWrite(D11_SET_MODE, Buffer, 2);
}
```

The USB Init function initialises the PDIUSBD11. This initialisation procedure has been omitted from the Philips PDIUSBD11 datasheet but is available from their FAQ. The last command enables the soft-connect pull up resistor on D+ indicating it is a full speed device but also advertises its presence on the universal serial bus.

```
void D11GetIRQ(void)
{
        unsigned short Irq;
        unsigned char Buffer[1];

        /* Read Interrupt Register to determine source of interrupt */

        D11CmdDataRead(D11_READ_INTERRUPT_REGISTER, (unsigned char *)&Irq, 2);

        if (Irq) printf("Irq = 0x%X: ",Irq);
```

Main() keeps calling the D11GetIRQ in a loop. This function reads the PDIUSBD11's Interrupt Register to establish if any interrupts are pending. If this is the case it will act upon them, otherwise it will continue to loop. Other USB devices may have a series of interrupt vectors assigned to each endpoint. In this case each ISR will service the appropriate interrupt removing the if statements.

```
        if (Irq & D11_INT_BUS_RESET) {
            printf("Bus Reset\n\r");
            USB_Init();
        }

        if (Irq & D11_INT_EP0_OUT) {
            printf("EP0_Out: ");
            Process_EP0_OUT_Interrupt();
        }

        if (Irq & D11_INT_EP0_IN) {
            printf("EP0_In: \n\r");
            if (CtlTransferInProgress == PROGRESS_ADDRESS) {
                D11CmdDataWrite(D11_SET_ADDRESS_ENABLE,&DeviceAddress,1);
                D11CmdDataRead(D11_READ_LAST_TRANSACTION + D11_ENDPOINT_EP0_IN, Buffer, 1);
                CtlTransferInProgress = PROGRESS_IDLE;
                }
        else {
                D11CmdDataRead(D11_READ_LAST_TRANSACTION + D11_ENDPOINT_EP0_IN, Buffer, 1);
                WriteBufferToEndPoint();
                }
        }
```

The If statements work down in order of priority. The highest priority interrupt is the bus reset. This simply calls USB_Init which re-initialises the USB function. The next highest priority is the default pipe consisting of EP0 OUT and EP1 IN. This is where all the enumeration and control requests are sent. We branch to another function to handle the EP0_OUT requests.

When a request is made by the host and it wants to receive data, the PIC16F876 will send the PDIUSBD11 a 8 byte packet. As the USbus is host controlled it cannot write the data when ever it desires, so the PDIUSBD11 buffers the data and waits for an IN Token to be sent from the host. When the PDIUSBD11 receives the IN Token it generates an interrupt. This makes a good time to reload the next packet of data to send. This is done by an additional function WriteBufferToEndpoint();

The section under CtlTransferInProgress == PROGRESS_ADDRESS handles the setting of the device's address. We detail this later.

```
        if (Irq & D11_INT_EP1_OUT) {
            printf("EP1_OUT\n\r");
            D11CmdDataRead(D11_READ_LAST_TRANSACTION + D11_ENDPOINT_EP1_OUT, Buffer, 1);
            bytes = D11ReadEndpoint(D11_ENDPOINT_EP1_OUT, Buffer);
            for (count = 0; count < bytes; count++) {
               circularbuffer[inpointer++] = Buffer[count];
               if (inpointer >= MAX_BUFFER_SIZE) inpointer = 0;
              }
            loadfromcircularbuffer(); //Kick Start
        }

        if (Irq & D11_INT_EP1_IN) {
            printf("EP1_IN\n\r");
            D11CmdDataRead(D11_READ_LAST_TRANSACTION + D11_ENDPOINT_EP1_IN, Buffer, 1);
            loadfromcircularbuffer();
        }
```

EP1 OUT and EP1 IN are implemented to read and write bulk data to or from a circular buffer. This setup allows the code to be used in conjunction with the BulkUSB example in the Windows DDK's. The circular buffer is defined earlier in the code as being 80 bytes long taking up all of bank1 of the PIC16F876's RAM.

```
        if (Irq & D11_INT_EP2_OUT) {
            printf("EP2_OUT\n\r");
            D11CmdDataRead(D11_READ_LAST_TRANSACTION + D11_ENDPOINT_EP2_OUT, Buffer, 1);
            Buffer[0] = 0x01; /* Stall Endpoint */
            D11CmdDataWrite(D11_SET_ENDPOINT_STATUS + D11_ENDPOINT_EP2_OUT, Buffer, 1);
        }

        if (Irq & D11_INT_EP2_IN) {
            printf("EP2_IN\n\r");
            D11CmdDataRead(D11_READ_LAST_TRANSACTION + D11_ENDPOINT_EP2_IN, Buffer, 1);
            Buffer[0] = 0x01; /* Stall Endpoint */
            D11CmdDataWrite(D11_SET_ENDPOINT_STATUS + D11_ENDPOINT_EP2_IN, Buffer, 1);
        }

        if (Irq & D11_INT_EP3_OUT) {
            printf("EP3_OUT\n\r");
            D11CmdDataRead(D11_READ_LAST_TRANSACTION + D11_ENDPOINT_EP3_OUT, Buffer, 1);
            Buffer[0] = 0x01; /* Stall Endpoint */
            D11CmdDataWrite(D11_SET_ENDPOINT_STATUS + D11_ENDPOINT_EP3_OUT, Buffer, 1);
        }

        if (Irq & D11_INT_EP3_IN) {
            printf("EP3_IN\n\r");
            D11CmdDataRead(D11_READ_LAST_TRANSACTION + D11_ENDPOINT_EP3_IN, Buffer, 1);
            Buffer[0] = 0x01; /* Stall Endpoint */
            D11CmdDataWrite(D11_SET_ENDPOINT_STATUS + D11_ENDPOINT_EP3_IN, Buffer, 1);
        }

}
```

Endpoints two and three are not used at the moment, so we stall them if any data is received. The PDIUSBD11 has a Set Endpoint Enable Command which can be used to enable or disable function generic endpoints (any endpoints other than the default control pipe). We could use this command to diable the generic endpoints, if we were planning on not using these later. However at the moment this code provides a foundation to build upon.

```
void Process_EP0_OUT_Interrupt(void)
{
    unsigned long a;
    unsigned char Buffer[2];
    USB_SETUP_REQUEST SetupPacket;

    /* Check if packet received is Setup or Data - Also clears IRQ */
    D11CmdDataRead(D11_READ_LAST_TRANSACTION + D11_ENDPOINT_EP0_OUT, &SetupPacket, 1);

    if (SetupPacket.bmRequestType & D11_LAST_TRAN_SETUP) {
```

The first thing we must do is determine is the packet we have received on EP0 Out is a data packet or a Setup Packet. A Setup Packet contains a request such as Get Descriptor where as a data packet contains data for a previous request. We are lucky that most requests do not send data packets from the host to the device. A request that does is SET_DESCRIPTOR but is rarely implemented.

```
        /* This is a setup Packet - Read Packet */
        D11ReadEndpoint(D11_ENDPOINT_EP0_OUT, &SetupPacket);

        /* Acknowlegde Setup Packet to EP0_OUT & Clear Buffer*/
        D11CmdDataWrite(D11_ACK_SETUP, NULL, 0);
        D11CmdDataWrite(D11_CLEAR_BUFFER, NULL, 0);

        /* Acknowlegde Setup Packet to EP0_IN */
        D11CmdDataWrite(D11_ENDPOINT_EP0_IN, NULL, 0);
        D11CmdDataWrite(D11_ACK_SETUP, NULL, 0);

        /* Parse bmRequestType */
        switch (SetupPacket.bmRequestType & 0x7F) {
```

As we have seen in our description of Control Transfers, a setup packet cannot be NAKed or STALLed. When the PDIUSBD11 receives a Setup Packet it flushes the EP0 IN buffer and disables the Validate Buffer and Clear Buffer commands. This ensures the setup packet is acknowledged by the microcontroller by sending an Acknowledge Setup command to both EP0 IN and EP0 OUT before a Validate or Clear Buffer command is effective. The recept of a setup packet will also un-stall a STALLed control endpoint.

Once the packet has been read into memory and the setup packet acknowledged, we being the parse the request starting with the request type. At the moment we are not interesting in the direction, so we AND off this bit. The three requests all devices must process is the Standard Device Request, Standard Interface Request and Standard Endpoint Requests. We provide our functionality (Read Analog Inputs) by the Vendor Request, so we add a case statement for Standard Vendor Requests. If your device supports a USB Class Specification, then you may also need to add cases for Class Device Request, Class Interface Request and/or Class Endpoint Request.

```
case STANDARD_DEVICE_REQUEST:
    printf("Standard Device Request ");
    switch (SetupPacket.bRequest) {
        case GET_STATUS:
            /* Get Status Request to Device should return */
            /* Remote Wakeup and Self Powered Status */
            Buffer[0] = 0x01;
            Buffer[1] = 0x00;
            D11WriteEndpoint(D11_ENDPOINT_EP0_IN, Buffer, 2);
            break;

        case CLEAR_FEATURE:
        case SET_FEATURE:
            /* We don't support DEVICE_REMOTE_WAKEUP or TEST_MODE */
            ErrorStallControlEndPoint();
            break;
```

The Get Status request is used to report the status of the device in terms of if the device is bus or self powered and if it supports remote wakeup. In our device we report it as self powered and as not supporting remote wakeup.

Of the Device Feature requests, this device doesn't support DEVICE_REMOTE_WAKEUP nor TEST_MODE and return a USB Request Error as a result.

```
        case SET_ADDRESS:
            printf("Set Address\n\r");
            DeviceAddress = SetupPacket.wValue | 0x80;
            D11WriteEndpoint(D11_ENDPOINT_EP0_IN, NULL, 0);
            CtlTransferInProgress = PROGRESS_ADDRESS;
            break;
```

The Set Address command is the only command that continues to be processed after the status stage. All other commands must finish processing before the status stage. The device address is read and OR'ed with 0x80 and stored in a variable DeviceAddress. The OR'ing with 0x80 is specific to the PDIUSBD11 with the most significant bit indicating if the device is enabled or not. A zero length packet is returned as status to the host indicating the command is complete. However the host must send an IN Token, retrieve the zero length packet and issue an ACK before we can change the address. Otherwise the device may never see the IN token being sent on the default address.

The completion of the status stage is signalled by an interrupt on EP0 IN. In order to differentiate between a set address response and a normal EP0_IN interrupt we set a variable, CtlTransferInProgress to PROGRESS_ADDRESS. In the EP0 IN handler a check is made of CtlTransferInProgress. If it equals PROGRESS_ADDRESS then the Set Address Enable command is issued to the PDIUSBD11 and CtlTransferInProgress is set to PROGRESS_IDLE. The host gives 2ms for the device to change its address before the next command is sent.

```
        case GET_DESCRIPTOR:
            GetDescriptor(&SetupPacket);
            break;

        case GET_CONFIGURATION:
            D11WriteEndpoint(D11_ENDPOINT_EP0_IN, &DeviceConfigured, 1);
            break;

        case SET_CONFIGURATION:
            printf("Set Configuration\n\r");
            DeviceConfigured = SetupPacket.wValue & 0xFF;
            D11WriteEndpoint(D11_ENDPOINT_EP0_IN, NULL, 0);
            if (DeviceConfigured) {
                RB3 = 0;
                printf("\n\r *** Device Configured *** \n\r");
                }
            else {
                RB3 = 1; /* Device Not Configured */
                printf("\n\r ** Device Not Configured *** \n\r");
                }
```

```
                break;

            //case SET_DESCRIPTOR:
            default:
                /* Unsupported - Request Error - Stall */
                ErrorStallControlEndPoint();
                break;

        }
        break;
```

The Get Configuration and Set Configuration is used to "enable" the USB device allowing data to be transferred on endpoints other than endpoint zero. Set Configuration should be issued with wValue equal to that of a bConfigurationValue of the configuration you want to enable. In our case we only have one configuration, configuration 1. A zero configuration value means the device is not configured while a non-zero configuration value indicates the device is configured. The code does not fully type check the configuration value, it only copies it into a local storage variable, DeviceConfigured. If the value in wValue does not match the bConfigurationValue of a Configuration, it should return with a USB Request Error.

```
    case STANDARD_INTERFACE_REQUEST:
        printf("Standard Interface Request\n\r");
        switch (SetupPacket.bRequest) {

            case GET_STATUS:
                /* Get Status Request to Interface should return */
                /* Zero, Zero (Reserved for future use) */
                Buffer[0] = 0x00;
                Buffer[1] = 0x00;
                D11WriteEndpoint(D11_ENDPOINT_EP0_IN, Buffer, 2);
                break;

            case SET_INTERFACE:
                /* Device Only supports default setting, Stall may be */
                /* returned in the status stage of the request */
                if (SetupPacket.wIndex == 0 && SetupPacket.wValue == 0)
                    /* Interface Zero, Alternative Setting = 0 */
                    D11WriteEndpoint(D11_ENDPOINT_EP0_IN, NULL, 0);
                else ErrorStallControlEndPoint();
                break;

            case GET_INTERFACE:
                if (SetupPacket.wIndex == 0) { /* Interface Zero */
                    Buffer[0] = 0; /* Alternative Setting */
                    D11WriteEndpoint(D11_ENDPOINT_EP0_IN, Buffer, 1);
                    break;
                 } /* else fall through as RequestError */

            //case CLEAR_FEATURE:
            //case SET_FEATURE:
                /* Interface has no defined features. Return RequestError */
            default:
                ErrorStallControlEndPoint();
                break;

        }
        break;
```

Of the Standard Interface Requests, none perform any real function. The Get Status request must return a word of zero and is reserved for future use. The Set Interface and Get Interface requests are used with alternative Interface Descriptors. We have not defined any alternative Interface Descriptors so Get Interface returns zero and any request to Set an interface other than to set interface zero with an alternative setting of zero is processed with a Request Error.

```
    case STANDARD_ENDPOINT_REQUEST:
        printf("Standard Endpoint Request\n\r");
        switch (SetupPacket.bRequest) {

            case CLEAR_FEATURE:
            case SET_FEATURE:
                /* Halt(Stall) feature required to be implemented on all Interrupt and */
                /* Bulk Endpoints. It is not required nor recommended on the Default Pipe */

                if (SetupPacket.wValue == ENDPOINT_HALT)
                {
                    if (SetupPacket.bRequest == CLEAR_FEATURE) Buffer[0] = 0x00;
                    else                                       Buffer[0] = 0x01;
                    switch (SetupPacket.wIndex & 0xFF) {
                        case 0x01 : D11CmdDataWrite(D11_SET_ENDPOINT_STATUS + \
                                      D11_ENDPOINT_EP1_OUT, Buffer, 1);
                                    break;
                        case 0x81 : D11CmdDataWrite(D11_SET_ENDPOINT_STATUS + \
                                      D11_ENDPOINT_EP1_IN, Buffer, 1);
                                    break;
                        case 0x02 : D11CmdDataWrite(D11_SET_ENDPOINT_STATUS + \
                                      D11_ENDPOINT_EP2_OUT, Buffer, 1);
                                    break;
```

```
                        case 0x82 : D11CmdDataWrite(D11_SET_ENDPOINT_STATUS + \
                                      D11_ENDPOINT_EP2_IN, Buffer, 1);
                                    break;
                        case 0x03 : D11CmdDataWrite(D11_SET_ENDPOINT_STATUS + \
                                      D11_ENDPOINT_EP3_OUT, Buffer, 1);
                                    break;
                        case 0x83 : D11CmdDataWrite(D11_SET_ENDPOINT_STATUS + \
                                      D11_ENDPOINT_EP3_IN, Buffer, 1);
                                    break;
                        default   : /* Invalid Endpoint - RequestError */
                                    ErrorStallControlEndPoint();
                                    break;
                    }
                    D11WriteEndpoint(D11_ENDPOINT_EP0_IN, NULL, 0);
                } else {
                    /* No other Features for Endpoint - Request Error */
                    ErrorStallControlEndPoint();
                }
                break;
```

The Set Feature and Clear Feature requests are used to set endpoint specific features. The standard defines one endpoint feature selector, ENDPOINT_HALT. We check what endpoint the request is directed to and set/clear the STALL bit accordingly. This HALT feature is not required on the default endpoints.

```
            case GET_STATUS:
                /* Get Status Request to Endpoint should return */
                /* Halt Status in D0 for Interrupt and Bulk */
                switch (SetupPacket.wIndex & 0xFF) {
                    case 0x01 : D11CmdDataRead(D11_READ_ENDPOINT_STATUS + \
                                  D11_ENDPOINT_EP1_OUT, Buffer, 1);
                                break;
                    case 0x81 : D11CmdDataRead(D11_READ_ENDPOINT_STATUS + \
                                  D11_ENDPOINT_EP1_IN, Buffer, 1);
                                break;
                    case 0x02 : D11CmdDataRead(D11_READ_ENDPOINT_STATUS + \
                                  D11_ENDPOINT_EP2_OUT, Buffer, 1);
                                break;
                    case 0x82 : D11CmdDataRead(D11_READ_ENDPOINT_STATUS + \
                                  D11_ENDPOINT_EP2_IN, Buffer, 1);
                                break;
                    case 0x03 : D11CmdDataRead(D11_READ_ENDPOINT_STATUS + \
                                  D11_ENDPOINT_EP3_OUT, Buffer, 1);
                                break;
                    case 0x83 : D11CmdDataRead(D11_READ_ENDPOINT_STATUS + \
                                  D11_ENDPOINT_EP3_IN, Buffer, 1);
                                break;
                    default   : /* Invalid Endpoint - RequestError */
                                ErrorStallControlEndPoint();
                                break;
                }
                if (Buffer[0] & 0x08) Buffer[0] = 0x01;
                else                  Buffer[0] = 0x00;
                Buffer[1] = 0x00;
                D11WriteEndpoint(D11_ENDPOINT_EP0_IN, Buffer, 2);
                break;

            default:
                /* Unsupported - Request Error - Stall */
                ErrorStallControlEndPoint();
                break;
        }
        break;
```

The Get Status request when directed to the endpoint returns the status of the endpoint, ie. if it is halted or not. Like the Set/Clear feature request ENDPOINT_HALT, we only need to report the status of the generic endpoints.

Any undefined Standard Endpoint Requests are handled by USB Request Error.

```
    case VENDOR_DEVICE_REQUEST:
    case VENDOR_ENDPOINT_REQUEST:
        printf("Vendor Device bRequest = 0x%X, wValue = 0x%X, wIndex = 0x%X\n\r", \
            SetupPacket.bRequest, SetupPacket.wValue, SetupPacket.wIndex);
        switch (SetupPacket.bRequest) {

            case VENDOR_GET_ANALOG_VALUE:
                printf("Get Analog Value, Channel %x :",SetupPacket.wIndex & 0x07);
                ADCON0 = 0xC1 | (SetupPacket.wIndex & 0x07) << 3;
                /* Wait Acquistion time of Sample and Hold */
                for (a = 0; a <= 255; a++);
                ADGO = 1;
                while(ADGO);
                Buffer[0] = ADRESL;
                Buffer[1] = ADRESH;
                a = (Buffer[1] << 8) + Buffer[0];
                a = (a * 500) / 1024;
                printf(" Value = %d.%02d\n\r",(unsigned int)a/100,(unsigned int)a%100);
                D11WriteEndpoint(D11_ENDPOINT_EP0_IN, Buffer, 2);
```

```
                    break;
```

Now comes the functional parts of the USB device. The Vendor Requests can be dreamed up by the designer. We have dreamed up two requests, VENDOR_GET_ANALOG_VALUE and VENDOR_SET_RB_HIGH_NIBBLE. VENDOR_GET_ANALOG_VALUE reads the 10-bit Analog Value on Channel x dictated by wIndex. This is ANDed with 0x07 to allow 8 possible channels, supporting the larger PIC16F877 if required. The analog value is returned in a two byte data packet.

```
                case VENDOR_SET_RB_HIGH_NIBBLE:
                    printf("Write High Nibble of PORTB\n\r");
                    PORTB = (PORTB & 0x0F) | (SetupPacket.wIndex & 0xF0);
                    D11WriteEndpoint(D11_ENDPOINT_EP0_IN, NULL, 0);
                    break;

                default:
                    ErrorStallControlEndPoint();
                    break;
            }
            break;
```

The VENDOR_SET_RB_HIGH_NIBBLE can be used to set the high nibble bits of PORTB[3:7].

```
        default:
            printf("UnSupported Request Type 0x%X\n\r",SetupPacket.bmRequestType);
            ErrorStallControlEndPoint();
            break;
    }
} else {
printf("Data Packet?\n\r");
/* This is a Data Packet */
}
}
```

Any unsupported request types such as class device request, class interface request etc is dealt with by a USB Request Error.

```
void GetDescriptor(PUSB_SETUP_REQUEST SetupPacket)
{
    switch((SetupPacket->wValue & 0xFF00) >> 8) {

        case TYPE_DEVICE_DESCRIPTOR:
            printf("\n\rDevice Descriptor: Bytes Asked For %d, Size of Descriptor %d\n\r", \
                    SetupPacket->wLength,DeviceDescriptor.bLength);
            pSendBuffer = (const unsigned char *)&DeviceDescriptor;
            BytesToSend = DeviceDescriptor.bLength;
            if (BytesToSend > SetupPacket->wLength)
                BytesToSend = SetupPacket->wLength;
            WriteBufferToEndPoint();
            break;

        case TYPE_CONFIGURATION_DESCRIPTOR:
            printf("\n\rConfiguration Descriptor: Bytes Asked For %d, Size of Descriptor %d\n\r", \
                    SetupPacket->wLength, sizeof(ConfigurationDescriptor));
            pSendBuffer = (const unsigned char *)&ConfigurationDescriptor;
            BytesToSend = sizeof(ConfigurationDescriptor);
            if (BytesToSend > SetupPacket->wLength)
                BytesToSend = SetupPacket->wLength;
            WriteBufferToEndPoint();
            break;
```

The Get Descriptor requests involve responses greater than the 8 byte maximum packet size limit of the endpoint. Therefore they must be broken up into 8 byte chunks. Both the Device and Configuration requests load the address of the relevant descriptors into pSendBuffer and sets the BytesToSend to the length of the descriptor. The request will also specify a descriptor length in wLength specifying the maximum data to send. In each case we check the actual length against that of what the host has asked for and trim the size if required. Then we call WriteBuffertoEndpoint which loads the first 8 bytes into the endpoint buffer and increment the pointer ready for the next 8 byte packet.

```
        case TYPE_STRING_DESCRIPTOR:
            printf("\n\rString Descriptor: LANGID = 0x%04x, Index %d\n\r", \
                SetupPacket->wIndex, SetupPacket->wValue & 0xFF);
            switch (SetupPacket->wValue & 0xFF){

                case 0 :  pSendBuffer = (const unsigned char *)&LANGID_Descriptor;
                          BytesToSend = sizeof(LANGID_Descriptor);
                          break;

                case 1 :  pSendBuffer = (const unsigned char *)&Manufacturer_Descriptor;
                          BytesToSend = sizeof(Manufacturer_Descriptor);
                          break;

                default : pSendBuffer = NULL;
                          BytesToSend = 0;
            }
```

```
            if (BytesToSend > SetupPacket->wLength)
                BytesToSend = SetupPacket->wLength;
            WriteBufferToEndPoint();
            break;
```

If any string descriptors are included, there must be a string descriptor zero present which details what languages are supported by the device. Any non zero string requests have a LanguageID specified in wIndex telling what language to support. In our case we cheat somewhat and ignore the value of wIndex (LANGID) returning the string, no matter what language is asked for.

```
        default:
            ErrorStallControlEndPoint();
            break;
    }
}

void ErrorStallControlEndPoint(void)
{
    unsigned char Buffer[] = { 0x01 };
    /* 9.2.7 RequestError - return STALL PID in response to next DATA Stage Transaction */
    D11CmdDataWrite(D11_SET_ENDPOINT_STATUS + D11_ENDPOINT_EP0_IN, Buffer, 1);
    /* or in the status stage of the message. */
    D11CmdDataWrite(D11_SET_ENDPOINT_STATUS + D11_ENDPOINT_EP0_OUT, Buffer, 1);
}
```

When we are faced with an invalid request, invalid parameter or a request the device doesn't support, we must report a request error. This is defined in 9.2.7 of the specification. A request error will return a STALL PID in response to the next data stage transaction or in the status stage of the message. However it notes that to prevent unnecessary bus traffic the error should be reported at the next data stage rather than waiting until the status stage.

```
unsigned char D11ReadEndpoint(unsigned char Endpoint, unsigned char *Buffer)
{
    unsigned char D11Header[2];
    unsigned char BufferStatus = 0;

    /* Select Endpoint */
    D11CmdDataRead(Endpoint, &BufferStatus, 1);

    /* Check if Buffer is Full */
    if(BufferStatus & 0x01)
    {
        /* Read dummy header - D11 buffer pointer is incremented on each read */
        /* and is only reset by a Select Endpoint Command */
        D11CmdDataRead(D11_READ_BUFFER, D11Header, 2);
        if(D11Header[1]) D11CmdDataRead(D11_READ_BUFFER, Buffer, D11Header[1]);
        /* Allow new packets to be accepted */
        D11CmdDataWrite(D11_CLEAR_BUFFER, NULL, 0);
    }
    return D11Header[1];
}

void D11WriteEndpoint(unsigned char Endpoint, const unsigned char *Buffer, unsigned char Bytes)
{
    unsigned char D11Header[2];
    unsigned char BufferStatus = 0;
    D11Header[0] = 0x00;
    D11Header[1] = Bytes;

    /* Select Endpoint */
    D11CmdDataRead(Endpoint, &BufferStatus, 1);
    /* Write Header */
    D11CmdDataWrite(D11_WRITE_BUFFER, D11Header, 2);
    /* Write Packet */
    if (Bytes) D11CmdDataWrite(D11_WRITE_BUFFER, Buffer, Bytes);
    /* Validate Buffer */
    D11CmdDataWrite(D11_VALIDATE_BUFFER, NULL, 0);
}
```

D11ReadEndpoint and D11WriteEndpoint are PDIUSBD11 specific functions. The PDIUSBD11 has two dummy bytes prefixing any data read or write operation. The first byte is reserved, while the second byte indicates the number of bytes received or to be transmitted. These two functions take care of this header.

```
void WriteBufferToEndPoint(void)
{
    if (BytesToSend == 0) {
        /* If BytesToSend is Zero and we get called again, assume buffer is smaller */
        /* than Setup Request Size and indicate end by sending Zero Lenght packet */
        D11WriteEndpoint(D11_ENDPOINT_EP0_IN, NULL, 0);
    } else if (BytesToSend >= 8) {
        /* Write another 8 Bytes to buffer and send */
        D11WriteEndpoint(D11_ENDPOINT_EP0_IN, pSendBuffer, 8);
        pSendBuffer += 8;
        BytesToSend -= 8;
    } else {
        /* Buffer must have less than 8 bytes left */
```

```
        D11WriteEndpoint(D11_ENDPOINT_EP0_IN, pSendBuffer, BytesToSend);
        BytesToSend = 0;
    }
}
```

As we have mentioned previously, WriteBufferToEndPoint is responsible for loading data into the PDIUSBD11 in 8 byte chunks and adjusting the pointers ready for the next packet. It is called once by the handler of a request to load the first 8 bytes into the endpoint buffer. The host will then send an IN token, read this data and the PDIUSBD11 will generate an interrupt. The EP0 IN handler will then call WriteBufferToEndpoint to load in the next packet in readiness for the next IN token from the host.

A transfer is considered complete if all requested bytes have been read, if a packet is received with a payload less than bMaxPacketSize or if a zero length packet is returned. Therefore if the BytesToSend counter hits zero, we assume the data to be sent was a multiple of 8 bytes and we send a zero length packet to indicate this is the last of the data. However if we have less than 8 bytes left to send, we send only the remaining bytes. There is no need to pad the data with zeros.

```
void loadfromcircularbuffer(void)
{
   unsigned char Buffer[10];
   unsigned char count;

   // Read Buffer Full Status
   D11CmdDataRead(D11_ENDPOINT_EP1_IN, Buffer, 1);

   if (Buffer[0] == 0){
      // Buffer Empty
      if (inpointer != outpointer){
         // We have bytes to send
         count = 0;
         do {
             Buffer[count++] = circularbuffer[outpointer++];
             if (outpointer >= MAX_BUFFER_SIZE) outpointer = 0;
             if (outpointer == inpointer) break; // No more data
            } while (count < 8); // Maximum Buffer Size
         // Now load it into EP1_In
         D11WriteEndpoint(D11_ENDPOINT_EP1_IN, Buffer, count);
         }
   }
}
```

The loadfromcircularbuffer() routine handles the loading of data into the EP1 IN endpoint buffer. It is normally called after an EP1 IN interrupt to reload the buffer ready for the next IN token on EP1. However in order to send out first packet, we need to load the data prior to receiving the EP1 IN interrupt. Therefore the routine is also called after data is received on EP1 OUT.

By also calling the routine from the handler for EP1 OUT, we are likely to overwrite data in the IN Buffer regardless of whether it has been sent or not. To prevent this, we determine if the EP1 IN buffer is empty, before we attempt to reload it with new data.

```
void D11CmdDataWrite(unsigned char Command, const unsigned char *Buffer, unsigned char Count)
{
    I2C_Write(D11_CMD_ADDR, &Command, 1);
    if(Count) I2C_Write(D11_DATA_ADDR_WRITE, Buffer, Count);
}

void D11CmdDataRead(unsigned char Command, unsigned char Buffer[], unsigned char Count)
{
    I2C_Write(D11_CMD_ADDR, &Command, 1);
    if(Count) I2C_Read(D11_DATA_ADDR_READ, Buffer, Count);
}
```

D11CmdDataWrite and D11CmdDataRead are two PDIUSBD11 specific functions which are responsible for sending the I2C Address/Command first and then send or received data on the I2C Bus. Additional lower level functions are included in the source code but have not been reproduced here as it is the intend to focus on the USB specific details.

This example can be used with the bulkUSB.sys example as part of the Windows DDK's. To load the bulkUSB.sys driver either change the code to identify itself with a VID of 0x045E and a PID of 0x930A or change the bulkUSB.inf file accompanying bulkUSB.sys to match the VID/PID combination you use in this example.

It is then possible to use the user mode console program, rwbulk.exe to send and receive packets from the circular buffer. Use

```
rwbulk -r 80 -w 80 -c 1 -i 1 -o 0
```

to send 80 byte chunks of data to the PIC16F876. Using payloads greater than 80 bytes is going to overflow the PIC's circular buffer in BANK1.

This example has been coded for readability at the expense of code size. It compiles in 3250 words of FLASH (39% capacity of the PIC16F876).

Acknowledgments

A special acknowledgment must go to Michael DeVault of DeVaSys Embedded Systems. This example has been based upon code written by Michael and been effortlessly developed on the USBLPT-PD11 DeVaSys USB development board before being ported to the PIC.

Downloading the Source Code

- Version 1.2, 14k bytes

Revision History

- 6th April 2002 - Version 1.2 - Increased I2C speed to match that of comment. Improved PDIUSBD11 IRQ Handling
- 7th January 2002 - Version 1.1 - Added EP1 IN and EP1 OUT Bulk handler routines and made descriptors load from FLASH
- 31st December 2001 - Version 1.0.

Chapter 6 : USB Requests

- The Setup Packet
- Standard Device Requests
- Standard Interface Requests
- Standard Endpoint Requests

Chapter 8 : A Generic USB Driver

Coming Soon

- Chapter 8: Generic USB Driver
- Chapter 9: HID Class Description and Example

Comments and Feedback?

Comments :
Email Address : (Optional) Send