Revision 0.1 - 11th December 2009

Part 2: Develop the USB interface to the PS3 controller

1. USB Reduced Hosts

Full USB hosts such as Windows and Linux based computers can manage a large variety of different USB devices and load appropriate USB drivers for each device. There is an enumeration or discovery phase where the host gathers information on the attached USB device and uses this information for the driver selection and configuration. In small embedded applications this is not possible to provide this variety, so the application usually only supports a few devices, often only one. This means the discovery process can be much reduced since the results are already known. This will reduce the memory required for the application by hard coding the device configuration into the application. Though the configuration will be hard coded, we still need to initially gather the information from the device itself and other sources.

The device drivers mentioned above for Windows and Linux are important in the development process. The Windows drivers are usually complete, but not available in source form. Linux drivers are available in source form, though not always as complete. Device manufacturers are usually very reluctant to provide information on the information required to build a driver or embedded application. So we rely on copying Linux code or “sniffing” Windows code to give us guidance.

2. PS3 Game Controller USB Interface

The first thing is to get the device and configuration descriptors. This may be documented, but often not, so we will collect direct from the USB device. This may be done under Windows using a program such as USBView or using the Arduino and the host shield.

Using USBView:
http://www.ftdichip.com/Resources/Utilities/usbview.zip
The PS3 controller is connected to the PC and then selected in USBView. The following descriptor information is delivered for the Sony PS3 controller when connected via USB.

```
Device Descriptor:
bcdUSB:              0x0200
bDeviceClass:          0x00
bDeviceSubClass:       0x00
bDeviceProtocol:       0x00
bMaxPacketSize0:       0x40 (64)
idVendor:            0x054C (Sony Corporation)
idProduct:           0x0268
bcdDevice:           0x0100
iManufacturer:         0x01
iProduct:              0x02
iSerialNumber:         0x00
bNumConfigurations:    0x01

ConnectionStatus: DeviceConnected
Current Config Value: 0x01
Device Bus Speed:      Full
Device Address:        0x01
Open Pipes:               2

Endpoint Descriptor:
bEndpointAddress:      0x02
Transfer Type:    Interrupt
wMaxPacketSize:      0x0040 (64)
bInterval:             0x01

Endpoint Descriptor:
bEndpointAddress:      0x81
Transfer Type:    Interrupt
wMaxPacketSize:      0x0040 (64)
bInterval:             0x01
```

The report shows two additional endpoints besides the control endpoint (endpoint 0). The first of these is an input endpoint (0x02) and is where the host will receive reports from the PS3 game controller when anything changes. The second (0x81) is an output endpoint and is where the host can send reports to the PS3 controller, though these can also be sent via special messages on the control endpoint 0.

Taking the same report for the Madcatz PS3 shows that it also has almost identical information. It also shows with identical idVendor and idProduct. The only difference is in the MaxPacketSize0 which is 8 for the Madcatz. This difference is managed automatically by the USB library.

To read the device and configuration descriptors on the Arduino, we install the USB Host Shield and download the following USB_Desc sketch. This reports the following descriptors with a bit more detail: http://github.com/ribbotson/USB-Host/blob/master/examples/USB_desc.pde

```
Start
freeMemory() reports 683

Device descriptor:
Descriptor Length:      12
Descriptor type:        01
USB version:    0200
Device class:   00
Device Subclass:        00
Device Protocol:        00
Max.packet size:        40
Vendor  ID:     054C
Product ID:     0268
Revision ID:    0100
Mfg.string index:       01
Prod.string index:      02
Serial number index:    00
Number of conf.:        01
Configuration descriptor:
Total length:   0029
Num.intf:               01
Conf.value:     01
Conf.string:    00
Attr.:          80
Max.pwr:                FA
Interface descriptor:
Intf.number:    00
Alt.:           00
Endpoints:              02
Intf. Class:            03
Intf. Subclass:         00
Intf. Protocol:         00
Intf.string:    00
Unknown descriptor:
Length:         09
Type:           21
Contents:       110100012294000705
Endpoint descriptor:
Endpoint address:       02
Attr.:          03
Max.pkt size:   0040
Polling interval:       01
Endpoint descriptor:
Endpoint address:       81
Attr.:          03
Max.pkt size:   0040
Polling interval:       01

memoryMin() reports 336
```

Also here you see the reports from freeMemory and memoryMin(), which for this example were added to check the memory consumption of the USB stack. These are not usually present in the report. There is also an unknown report which is a class report for the HID device. The large difference between freeMemory and memoryMin in this example is due to a 256 byte buffer being defined for the configuration data plus there is also about 100 bytes of stack in data memory used by the USB library. The text strings were moved to program memory with a special print function, otherwise it would not fit into the available data memory of an ATMega168.

The sketch uses this USB Host library:

http://github.com/ribbotson/USB-Host/tree/master/usb_host/

The library was modified from the version from Oleg to give a longer time for the device to deliver the descriptors. However in the non blocking code we need later, we also need sometimes not to wait at all. Therefore an additional change is made to allow for waiting or not.

3. Configuring the USB library to connect the PS3 game Controller

Having obtained the descriptor information, this is coded into the USB configuration of the USB library. The library requires a structure for each endpoint to contain this data and this structure is then provided to the library using the Usb.setDevTableEntry() function.

```
/* Endpoint information structure               */
/* bToggle of endpoint 0 initialized to 0xff    */
/* during enumeration bToggle is set to 00      */
typedef struct {
    byte epAddr;        //copy from endpoint descriptor without bit 7 if set
    byte Attr;          // Endpoint transfer type.
    unsigned int MaxPktSize;    // Maximum packet size.
    byte Interval;      // Polling interval in frames.
    byte sndToggle;     //last toggle value, bitmask for HCTL toggle bits
    byte rcvToggle;     //last toggle value, bitmask for HCTL toggle bits
} EP_RECORD;
```

We then also make a check to confirm that the device connected is the one which we support by checking the VID and PID in the device descriptor.

```
device_descriptor = (USB_DEVICE_DESCRIPTOR *) &buf;
    if(
    (device_descriptor->idVendor != PS3_VID) ||(device_descriptor->idProduct != PS3_PID)  ) {
        Serial.println("Unsupported USB Device");
          while(1);  //stop
    }
```

If this is successful we set the configuration for the device and then we have three pipes connected to the PS3 device from the Arduino and communication is established.

4. Communication with the Game controller from a sketch

Having established communication we now have to send commands and receive reports in the format the PS3 game controller requires. This format is somewhat described in the documents mentioned in the first article and emulated in the software for linux and python. This information was probably obtained from “sniffing” the interaction of the game controller to a PS3. I do have a USB sniffer from Ellisys which could connect between a PS3 and the controller, but I do not have a PS3. So I will rely on published information and will also “sniff” the connection between the game controller and a PC running the motioninjoy software.

Motioninjoy is probably the most complete PC application to support the PS3 controller so should offer the best information. To sniff the connection I used a PC based program called SnoopyPro:

http://sourceforge.net/projects/usbsnoop/

Between the setting of the configuration and the receipt of data on the input pipe , there are two commands sent to the PS3 game controller. The first sets the game controller to send 01 type reports to the input pipe whenever any changes occur on the controller.

```
SetupPacket:
0000: 21 09 f4 03 00 00 04 00
bmRequestType: 21
  DIR: Host-To-Device
  TYPE: Class
  RECIPIENT: Interface
bRequest: 09
TransferBuffer: 0x00000004 (4) length
0000: 42 0c 00 00
```

This can be performed using the USB library by:

```
#define PS3_F4_REPORT_LEN 4
#define HID_REPORT_FEATURE 3
#define PS3_F4_REPORT_ID  0xF4
…
prog_char feature_F4_report[] PROGMEM = {0x42, 0x0c, 0x00, 0x00};
…
  /* Set the PS3 controller to send reports */
    for (i=0; i < PS3_F4_REPORT_LEN; i++) buf[i] = pgm_read_byte_near( 
feature_F4_report + i);
    rcode = Usb.setReport( PS3_ADDR, ep_record[ CONTROL_PIPE ].epAddr, 
PS3_F4_REPORT_LEN,  PS3_IF, HID_REPORT_FEATURE, PS3_F4_REPORT_ID , buf );
    if( rcode ) {
        Serial.print("Set report error: ");
        Serial.println( rcode, HEX );
        while(1);  //stop
    }
```

Here the report content was held in program memory.

The second command packet is not essential, but sets the LED on the game controller so that it stops flashing and goes to light LED 1 constantly. This command is developed later to control all the LED and rumble.

```
SetupPacket:
0000: 21 09 01 02 00 00 30 00
bmRequestType: 21
  DIR: Host-To-Device
  TYPE: Class
  RECIPIENT: Interface
bRequest: 09
TransferBuffer: 0x00000030 (48) length
0000: 00 ff 00 ff 00 00 00 00 00 02 ff 27 10 00 32 ff
0010: 27 10 00 32 ff 27 10 00 32 ff 27 10 00 32 00 00
0020: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
```

This can be performed using the USB library by:

```
#define PS3_01_REPORT_LEN 48
#define HID_REPORT_OUTPUT  2
#define PS3_01_REPORT_ID  0x01
…
prog_char output_01_report[] PROGMEM = {0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00,
                                     0x00, 0x02, 0xff, 0x27, 0x10, 0x00, 0x32, 0xff,
                                     0x27, 0x10, 0x00, 0x32, 0xff, 0x27, 0x10, 0x00,
                                     0x32, 0xff, 0x27, 0x10, 0x00, 0x32, 0x00, 0x00,
                                     0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
                                     0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00
};
…
   /* Set the PS3 controller LED 1 On */
    for (i=0; i < PS3_01_REPORT_LEN; i++) buf[i] = pgm_read_byte_near(
output_01_report + i);
    rcode = Usb.setReport( PS3_ADDR, ep_record[ CONTROL_PIPE ].epAddr,
PS3_01_REPORT_LEN,  PS3_IF, HID_REPORT_OUTPUT, PS3_01_REPORT_ID , buf );
    if( rcode ) {
        Serial.print("Set report error: ");
        Serial.println( rcode, HEX );
        while(1);  //stop

    }
```

01 type reports can now be read from the input pipe using:

```
/* Poll PS3 interrupt pipe and process result if any */

void PS3_poll( void )
{

 byte rcode = 0;     //return code
    /* poll PS3 */
    rcode = Usb.inTransfer(PS3_ADDR, ep_record[ INPUT_PIPE ].epAddr,
PS3_01_REPORT_LEN, buf );
    if( rcode != 0 ) {
       return;
    }
    process_report();
    return;
}
```

5. Interpreting and using PS3 reports

The 01 type reports are formatted as follows:

```
//Structure which describes the type 01 input report
typedef struct {
    unsigned char ReportType;     //Report Type 01
    unsigned char Reserved1;      // Unknown
    unsigned int  ButtonState;    // Main buttons
    unsigned char PSButtonState;  // PS button
    unsigned char Reserved2;      // Unknown
    unsigned char LeftStickX;     // left Joystick X axis 0 - 255, 128 is mid
    unsigned char LeftStickY;     // left Joystick Y axis 0 - 255, 128 is mid
    unsigned char RightStickX;    // right Joystick X axis 0 - 255, 128 is mid
    unsigned char RightStickY;    // right Joystick Y axis 0 - 255, 128 is mid
    unsigned char Reserved3[4];   // Unknown
    unsigned char PressureUp;     // digital Pad Up button Pressure 0 - 255
    unsigned char PressureRight;  // digital Pad Right button Pressure 0 - 255
    unsigned char PressureDown;   // digital Pad Down button Pressure 0 - 255
    unsigned char PressureLeft;   // digital Pad Left button Pressure 0 - 255
    unsigned char PressureL2;     // digital Pad L2 button Pressure 0 - 255
    unsigned char PressureR2;     // digital Pad R2 button Pressure 0 - 255
    unsigned char PressureL1;     // digital Pad L1 button Pressure 0 - 255
    unsigned char PressureR1;     // digital Pad R1 button Pressure 0 - 255
    unsigned char PressureTriangle;   // digital Pad Triangle button Pressure 0 - 255
    unsigned char PressureCircle;     // digital Pad Circle button Pressure 0 - 255
    unsigned char PressureCross;      // digital Pad Cross button Pressure 0 - 255
    unsigned char PressureSquare;     // digital Pad Square button Pressure 0 - 255
    unsigned char Reserved4[3];   // Unknown
    unsigned char Charge;         // charging status ? 02 = charge, 03 = normal
    unsigned char Power;          // Battery status ?
    unsigned char Connection;     // Connection Type ?
    unsigned char Reserved5[9];   // Unknown
    unsigned int AccelerometerX;  // X axis accelerometer Big Endian 0 - 1023
    unsigned int Accelero         // Y axis accelerometer Big Endian 0 - 1023
    unsigned int AccelerometerZ;  // Z axis accelerometer Big Endian 0 - 1023
    unsigned int GyrometerX;      // Z axis Gyro Big Endian 0 - 1023

} TYPE_01_REPORT;
```

The state of the game controller buttons is contained in the ButtonState and PSButtonState variables mapped as follows:

```
#define buSelect    (PS3Report->ButtonState & 0x0001)
#define buLAnalog   (PS3Report->ButtonState & 0x0002)
#define buRAnalog   (PS3Report->ButtonState & 0x0004)
#define buStart     (PS3Report->ButtonState & 0x0008)
#define buUp        (PS3Report->ButtonState & 0x0010)
#define buRight     (PS3Report->ButtonState & 0x0020)
#define buDown      (PS3Report->ButtonState & 0x0040)
#define buLeft      (PS3Report->ButtonState & 0x0080)
#define buL2        (PS3Report->ButtonState & 0x0100)
#define buR2        (PS3Report->ButtonState & 0x0200)
#define buL1        (PS3Report->ButtonState & 0x0400)
#define buR1        (PS3Report->ButtonState & 0x0800)
#define buTriangle  (PS3Report->ButtonState & 0x1000)
#define buCircle    (PS3Report->ButtonState & 0x2000)
#define buCross     (PS3Report->ButtonState & 0x4000)
#define buSquare    (PS3Report->ButtonState & 0x8000)
#define buPS        (PS3Report->PSButtonState & 0x01)
```

The Accelerometer and Gyro values are in Big Endian format and need byte swapping before use as an unsigned integer value.

The following example program shows how the values can be used from the report. The example used the LCD display connected to the USB Host Shield. The example also controls the LED and rumble motors. This example does not use the library described below.

URL HERE!!

A menu item is provided in the example to read and set the Bluetooth Device Address of the Game controller. This is used in the pairing process for the game controller when in Bluetooth mode. This setup can alternatively be done with sixpair.c under linux, or motioninjoy under windows.

6. A PS3 Game Controller Library