Interfacing an USB Slave with Linux

Written by Jac Kersing.

In this article we will implement an USB device using the Flowcode (version 3) USB Slave component and access the device from Linux. The simple example application we'll implement accepts strings of up to 16 characters from the host and displays those on an LCD display. It also allows the host to query the state of some push buttons.

The components used for the example application are:

- ECIO-28 mounted on an ECIO base board
- E-blocks LCD board
- E-Blocks Switch board
- Power supply
- Linux system with CentOS 5 installed

PIC Software

First we will create the Flowcode application. Start your copy of Flowcode, you'll need the professional edition for this example, and start a new project. Set the target to ECIO-28.

Add the following components to your project:

1. USB Slave
2. LCDDisplay
3. Switches

Check the pin connections of the LCDDisplay to make sure it is connected to Port B bits 0 (Data 1) to 5 (Enable). The switches should be connected to Port A bits 0 to 5.

The USB Slave component needs a little more configuration. Open the properties screen and in the “Properties” tab enter the information as shown in figure 1.

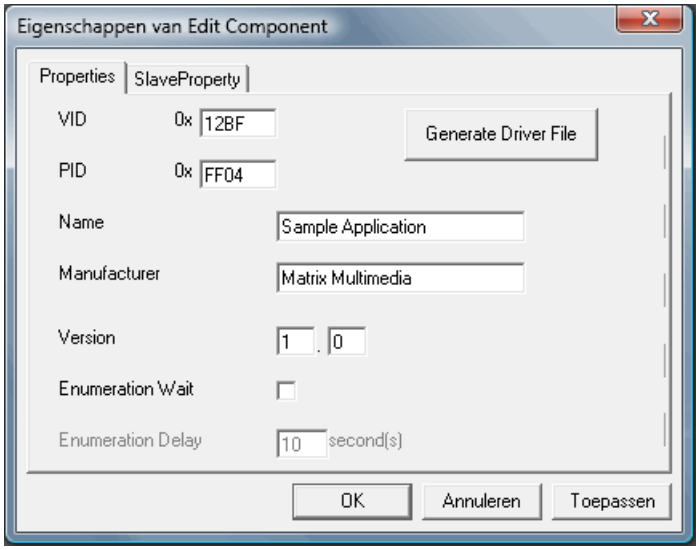


Figure 1 - USB Slave component properties

The VID (Vendor ID) and PID (Product ID) identify the USB device to the host system. The

combination needs to be unique for the host computer to be able to distinguish between different devices with different functions.
Matrix Multimedia has a number of PIDs available for customers for use in projects. Contact Matrix Multimedia if you want a PID for your project.

Next, Switch to the “SlaveProperty” tab and enter “usb_service” next to the the “Slave Service Macro Name” and choose “char, string” from the “Slave Service Macro Parameters” drop down. The result should look like the dialog shown in figure 2.

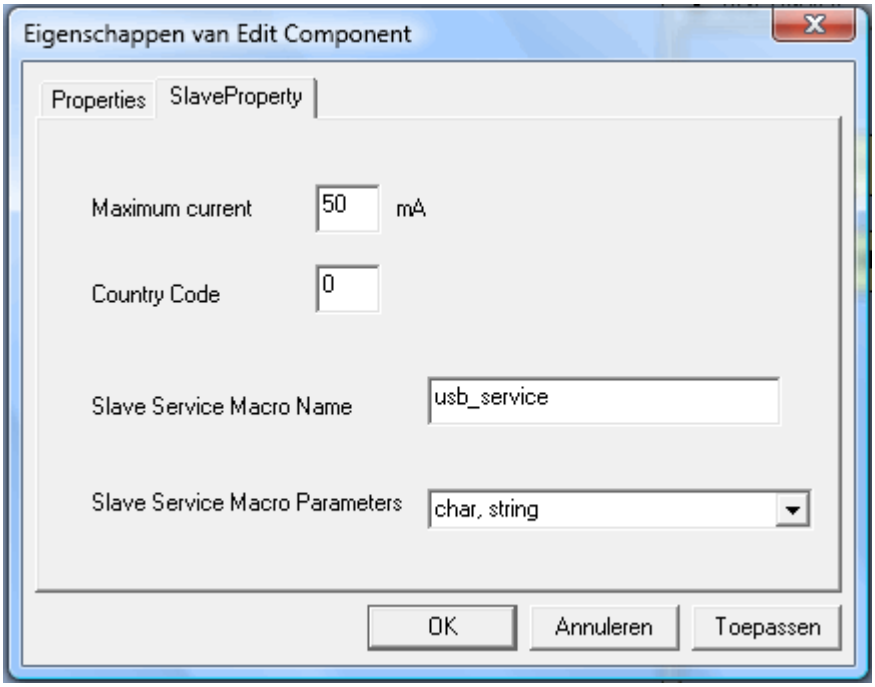


Figure 2 - USB Slave slaveproperty

Now we'll create the main routine. Drag 5 component macro's between begin and end and starting with the topmost one adjust them to read:

Display name	**Component**	**Macro**	**Parameters**	**Return Value**
Initialize Display	LCDDisplay(0)	Start	n.a.	n.a.
Initialize USB	USBSlave(0)	Initialize_Slave	n.a.	n.a.[1]
Clear LCD Display	LCDDisplay(0)	Clear	n.a.	n.a.
Display message	LCDDisplay(0)	PrintString	“App started”	n.a.
Start USB Service	USBSlave(0)	Start_Slave_Service	n.a.	n.a.

1 We're ignoring the result from this call.

The result should look like this:

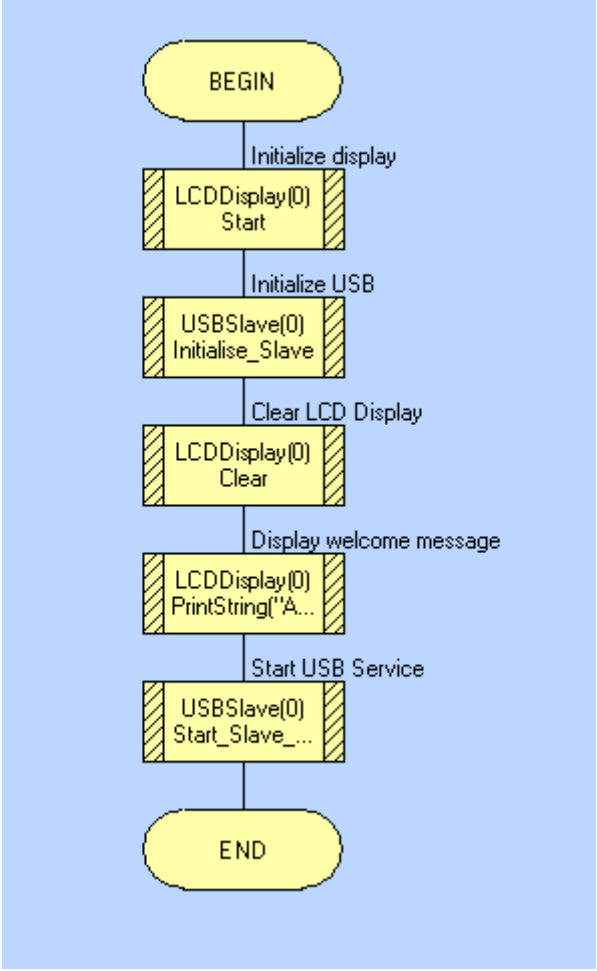


Figure 3 - Main program

Next the usb_service macro needs to be created. Create a new macro with the following details:
Name: usb_service (this needs to match the name specified in the USBSlave component properties)
Description: Service USB request (free format text, entering a useful description is recommended)
Parameters:

command	BYTE
arguments[20]	STRING

Local variables:

response[20]	STRING
response_length	BYTE
i	BYTE

Return type: No return variable.

The parameters are pushed from the computer to the PIC, the local variables are used while processing the command (i) and to pass results back to the computer (response and response_length)

Insert the following elements:

1. Calculation
   Name: Initialize response length
   Calculations:
   usb_service.response_length = 0
2. Decision
   Display name: Version Command?

If: usb_service.command == 0

3. Decision
   Display name: Display message command?
   If: usb_service.command == 1
4. Decision
   Display name: Read switch command?
   If: usb_service.command == 2
5. Component Macro
   Component: USBSlave(0)
   Macro: Send_String
   Parameters: usb_service.response, usb_service.response_length

The decisions compare the command byte (the first byte of the bytes the computer sends to the PIC) to a predefined value. This can be any value starting at 0 up to 255, use different values for the commands if you do not want to complicate things unnecessarily. I decided to keep things simple and choose a sequence of numbers starting at zero for the commands.

The current macro does not do anything if a command is recognized. Let's change that right now. Insert a calculation into the 'yes' flow of the first 'if' element and change the attributes as follows.
Display name: Set response
Calculations:

```
usb_service.response_length = 2
usb_service.response[0] = 1
usb_service.response[1] = 0
```

This sets the first two bytes of the response string (array) and defines the length of the response the component macro at the end of this macro needs to send.

Insert the following in the 'yes' flow of the 'Display message command' - 'if'.

1. Component Macro
   Display name: Clear display
   Component: LCDDisplay(0)
   Macro: Clear
   (No parameters)
2. Calculation
   Display name: Initialize counter to first character
   Calculations:
   ```
   usb_service.i = 1
   ```
3. Loop
   Display name: Loop while characters left
   Loop while: usb_service.i < usb_service.arguments[0]
   Test loop at the: Start

The next two elements are to be placed inside the loop:

4. Component Macro
   Display name: Display character
   Component: LCDDisplay(0)
   Macro: PrintASCII
   Parameters: usb_service.arguments[usb_service.i]
5. Calculation
   Display name: Increment counter
   Calculations:
   ```
   usb_service.i = usb_service.i + 1
   ```

After the end of the loop

6. Calculation
   Display name: Signal success
   Calculations:
     usb_service.response_length = 1
     usb_service.response[0] = 1

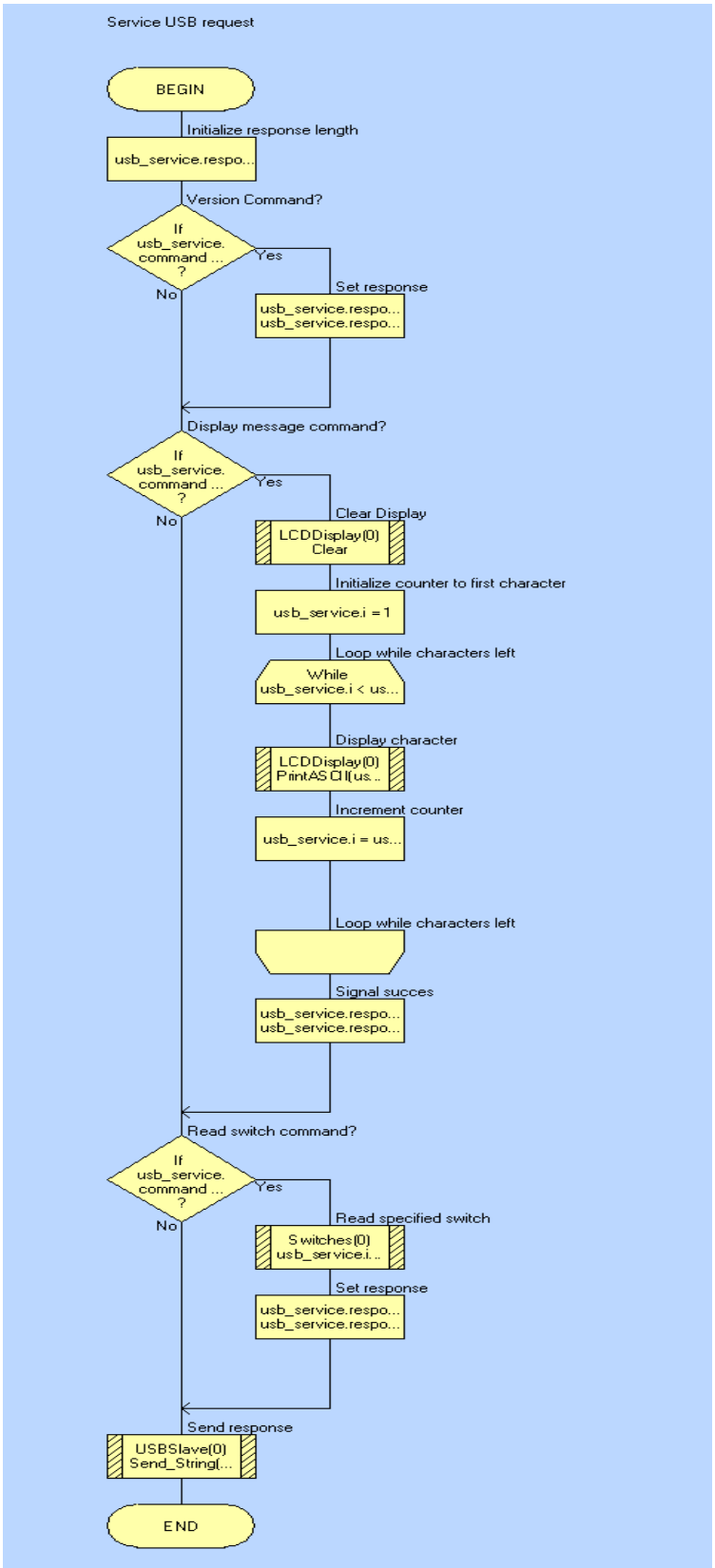
The flow we just added assumes the first byte of the argument string contains the length of the string to display. It then 'prints' the characters in the loop and finally sets the response to a value that signals the computer the command has been processed.

In the 'yes' flow of the final ('Read switch command') 'if' we need to add the following elements:

1. Component macro
   Component: Switches(0)
   Macro: ReadState
   Parameters: usb_service.arguments[0]
   Return Value: usb_service.i
2. Calculation
   Display name: Set response
   Calculations:
     usb_service.response_length = 1
     usb_service.response_[0] = usb_service.i

This flow reads the state of the switch specified in the first byte of the argument string and sets the response to the switch state.

The completed macro should look like figure 4.



Our flowcode program is now complete. (It is included in the archive as 'usbslave_example.fcf')

Hardware

Now the PIC software has been created the hardware needs to be set up. Connect the LCD board to

Port B of the ECIO base board. Connect the switch board to Port A of the ECIO base board. Wire the +V inputs of the LCD and switch boards to the VDD OUT of the ECIO base board.
Put the ECIO-28 in the ECIO base board and set the jumper on the ECIO-28 to the EXT position. Next connect the power supply to the ECIO base board and plug it into a power socket. Connect an USB cable between ECIO-28 and the system running Flowcode. If you press the reset button at this point the LED on the ECIO should start to flash indicating the bootloader is ready to receive your code. (The bootloader will automatically abort after a couple of seconds and start your application once it has been loaded)

To load the program at this point, select “Compile to chip” in Flowcode and follow the instructions on screen. Upon successfull completion you are ready for the next step, the Linux software.

Linux Host Software

The first test to see if everything works correctly is to disconnect the USB cable of the newly programmed ECIO from the system used for Flowcode and connect it to a Linux system, wait for the led to stop flashing. Now open a terminal on the Linux system and run the command '/sbin/lsusb' to list all connected USB devices. The result should look like the output listed below.

```
1 Bus 002 Device 004: ID 12bf:ff04
2 Bus 002 Device 001: ID 0000:0000
3 Bus 005 Device 001: ID 0000:0000
4 Bus 003 Device 001: ID 0000:0000
5 Bus 004 Device 002: ID 108b:0005 Grand-tek Technology Co., Ltd
6 Bus 004 Device 001: ID 0000:0000
7 Bus 001 Device 001: ID 0000:0000
```

Searching the list for the vendor id (12BF) and product id (FF04) we specified in the USB slave component properties we find it at line 1. The additional descriptions provided earlier at the component properties tab are not shown, the 'lsusb' command uses a file to find the information it displays and our vendor/product combination is not available in this file.

To add the information, become root and edit '/usr/share/hwdata/usb.ids'. Add the next two lines at the end of the file:

```
12bf Matrix Multimedia,
        ff04 Example Application
```

The first line defines the vendor id. The second line starts with a 'tab' character and specifies the product id within this specific vendor id. The resulting 'lsusb' line for our device is now:

```
1 Bus 002 Device 004: ID 12bf:ff04 Matrix Multimedia, Example Application
```

In order to write code to talk to the USB device 'libusb' needs to be installed. Check if it has been installed by running 'rpm -q libusb'. If the output is 'package libusb is not installed' it needs to be installed by running 'yum install libusb', otherwise the full package name including version string will be displayed. The examples have been written for the libusb version 0.1.x, where x > 10.

USB Device permissions

At this point we'll examine how to configure your Linux system to allow non privileged users access to the PIC.

When we connect the ECIO-28 to the Linux host the system detects the new device and runs a number of processes. One of these processes creates a couple of new entries in the '/dev' directory.

```
1 crw------- 1 root root 442, 2053 Jun 23 22:08 usbdev2.4_ep00
2 crw------- 1 root root 442, 2053 Jun 23 22:09 usbdev2.4_ep01
3 crw------- 1 root root 442, 2053 Jun 23 22:09 usbdev2.4_ep81
```

Looking at the permissions it is pretty obvious only root can access the device. The way to change this is by adding creating a new file named '/etc/udev/rules.d/99-usb.rules' with the following line as content:

```
1 SYSFS{idVendor}=="12bf", SYSFS{idProduct}=="ff04", MODE="666"
```

This file tells the process creating the new entries in '/dev' to use the mode '666' (rw-rw-rw-) for USB devices with the vendor id and product id of our PIC program. If we unplug and replug (or reset) the ECIO-28 and take an other look in the '/dev' directory we will see:

```
1 crw-rw-rw- 1 root root 442, 2054 Jun 24 00:18 /dev/usbdev2.7_ep00
2 crw-rw-rw- 1 root root 442, 2054 Jun 24 00:18 /dev/usbdev2.7_ep01
3 crw-rw-rw- 1 root root 442, 2054 Jun 24 00:18 /dev/usbdev2.7_ep81
```

As expected the file permissions now allow anyone on the system to access the device.

SEARCH.C

All prerequisites have been met, now let us start with a simple program. This program will search the connected USB devices for the first device with the correct vid and pid, it will not invoke any of our custom macro code yet.

```
1 /*
2  * search.c : Find USB device
3  */
4
5 #include <stdio.h>
6 #include <stdlib.h>
7 #include <usb.h>
8
9 #define VID  0x12bf
10#define PID  0xff04
11
12int main(int argc, char *argv[]) {
13  struct usb_bus *bus;
14
15  usb_init();
16  usb_find_busses();
17  usb_find_devices();
18
19  for (bus = usb_get_busses(); bus != NULL; bus=bus->next) {
20    struct usb_device *device;
21
22    for (device = bus->devices; device != NULL; device = device->next) {
23      if (device->descriptor.idVendor == VID &&
24          device->descriptor.idProduct == PID) {
25            printf("Device found\n");
26            exit(0);
27      }
28    }
29  }
30  printf("Device not found\n");
31  exit(1);
32}
```

Focusing on the USB specifics of the program, line 7 includes the libusb data structures. Lines 15, 16 and 17 initialize the library, enumerate the USB buses and search for all attached devices. Line 19 starts the loop for all found buses and line 22 for all devices connected to that particular bus. In line 23 and 24 the vid and pid of the device are compared to our values, the program prints a message if they match and exits.

Compile the program with the command 'cc -o search -l usb search.c'. Run the program with './search'.

VERSION.C

The previous program does nothing you can't do by running 'lsusb' and checking the ouput, now we create a program that does call our custom macro to get the version number.

```
1 /*
2  * version.c : find device and query for version information
3  */
4
5 #include <stdio.h>
6 #include <stdlib.h>
7 #include <usb.h>
8
9 /* Some constants used later on */
10 #define TIMEOUT         2000
11 #define ENDPOINT        1
12 #define CONFIG          1
13
14 #define VID             0x12bf
15 #define PID             0xff04
```

The start of the program is similar to the last one. Data structure definitions are included and a number of constants needed later on are declared.

```
16 usb_dev_handle *findAndOpenUsb() {
17   struct usb_bus *bus;
18
19   usb_init();
20   usb_find_busses();
21   usb_find_devices();
22
23   for (bus = usb_get_busses(); bus; bus=bus->next) {
24     struct usb_device *device;
25
26     for (device = bus->devices; device; device = device->next) {
27       struct usb_dev_handle *dev;
28
29       if (device->descriptor.idVendor == VID &&
30           device->descriptor.idProduct == PID)
31       {
```

Above is the start of the function to find our USB device on the bus. It is very similar o the previous example up to this point, that will change in the next few lines as we need access to the device this time. So in the next few lines the device is opened and the first (and only) available configuration on the device is opened[2].

2 The USB specification allows for multiple configurations for a device. As Flowcode uses only one configuration we will not be exploring this in detail. Consult the USB specification or the Internet if you want more information.

```
32        dev = usb_open(device);
33        if (usb_set_configuration(dev,CONFIG) < 0) {
34          printf("Error setting configuration\n");
35          exit(1);
36        }
37        if (usb_claim_interface(dev, 0)) {
38          printf("Can't claim interface for communication.\n");
39          exit(1);
40        }
41        return dev;
```

Return to the caller if the device has been found and has been claimed succesfully. The program will be aborted if selecting the configuration or claiming the interface fails.

```
42      }
43    }
44  }
45  return (usb_dev_handle *)NULL;
46}
```

Return NULL to the caller if no device has been found.

The next function sends the 'version' command to the device. You might recall we compare the command byte received from the host in our Flowcode macro. For the 'version' command we're comparing the command to '0'...

```
47void getVersion(usb_dev_handle *dev) {
48  unsigned char buffer[2];
49  int len;
```

Buffer is used to store the bytes to send and the bytes we're receiving.

```
50  // write version command to device
51  buffer[0] = 0x00;
52  if (usb_bulk_write(dev,ENDPOINT,buffer,1,TIMEOUT) < 0) {
53    printf("Error writing.\n");
54    usb_close(dev);
55    exit(2);
56  }
```

The code above sets the command byte, which is the first byte of the data transfered to the PIC to the 'version' command (0). Then it sends the data to the device.

```
57  // read response
58  len = usb_bulk_read(dev,ENDPOINT,buffer,2,TIMEOUT);
59  if ( len != 2 ) {
60    printf("Error reading\n");
61    usb_close(dev);
62    exit(2);
63  }
64  printf("Firmware version: %d.%02d\n",buffer[1],buffer[0]);
65}
```

Once the command has been sent the PIC will respond with an answer. In the usb_service macro we defined the response to be two bytes, so we need to read two bytes into the buffer. We're aborting the program if the number of bytes does not match the expected number, if we successfully received

two bytes the version information will be displayed. I've chosen to send the major information in the second byte of the response and the minor version information in the first response byte, there are no requirements for the byte order, you only need to make sure both the host code and the device code use the same order.

```
66 int main(int argc, char *argv[]) {
67   usb_dev_handle *device;
68
69   device = findAndOpenUsb();
70   if (device == NULL) {
71     printf("Device not found.\n");
72     exit(1);
73   }
74   getVersion(device);
75 }
```

The main routine of the program simply calls the find routine and exits if it is not found. If the device is found the function to display the version information is called.

Compile this program using the command 'cc -o version -l usb version.c'. Now run it, the output should be:

```
1 # ./version
2 Firmware version: 0.01
```

LCDSWITCH.C

The previous program has shown us how to send a command to the PIC and read the result. In our final example we will be sending a string to display and read a button. Here I'll only go into the two routines handling these tasks. The full program is available in the archive with all the other example code.

Let us start with the readSwitch function:

```
1  int readSwitch(usb_dev_handle *dev,int swtch) {
2    unsigned char buffer[2];
3    int len;
4
5    // write read switch command to device
6    buffer[0] = 0x02;
7    buffer[1] = swtch;
8    if (usb_bulk_write(dev,ENDPOINT,buffer,2,TIMEOUT) < 0) {
9      printf("Error writing.\n");
10     usb_close(dev);
11     exit(2);
12   }
```

The code sets the command byte, the number of the switch to read and sends the command to the PIC. Now we need to read the response:

```
13   // read response
14   len = usb_bulk_read(dev,ENDPOINT,buffer,1,TIMEOUT);
15   if ( len != 1 ) {
16     printf("Error reading\n");
17     usb_close(dev);
18     exit(2);
19   }
20   if (buffer[0] == 1) {
21     printf("Switch closed\n");
```

```
22  } else {
23    printf("Switch open\n");
24  }
25  return (buffer[0]);
26}
```

The response should be one byte, if not an error is displayed and the program aborts. If one byte is received we check if it is 1 which signals the switch is closed. Otherwise the switch is open.

The sendString routine sets the command to display the string, sets the length of the string (limited to 16 characters), copies the string and sends it. To check if the PIC successfully completed the command a single byte valued 1 should be received:

```
1 void sendString(usb_dev_handle *dev,char *string) {
2   unsigned char buffer[20];
3   int len, i;
4
5   len = strlen(string);
6   if (len > 16) {
7     len = 16;
8   }
9
10  // write display message command to device
11  buffer[0] = 0x01;
12  buffer[1] = len+1;
13  for (i = 0; i < len; i++) {
14    buffer[2+i] = string[i];
15  }
16  if (usb_bulk_write(dev,ENDPOINT,buffer,2+len,TIMEOUT) < 0) {
17    printf("Error writing.\n");
18    usb_close(dev);
19    exit(2);
20  }
21
22  // read response
23  len = usb_bulk_read(dev,ENDPOINT,buffer,1,TIMEOUT);
24  if ( len != 1 ) {
25    printf("Error reading\n");
26    usb_close(dev);
27    exit(2);
28  }
29  if (buffer[0] == 1) {
30    printf("Displayed successfull\n");
31  }
32}
```

The other parts of this program search for the device, just like our previous example and parse the command line arguments. This program can be compiled with the command 'cc -o lcdswitch -l usb lcdswitch.c'. To display a string run './lcdswitch --display “Hello world”', read the status of switch SW0 with './lcdswitch --switch 0'.

A simple sample application would be to show the system load on the LCD (showload.sh):

```
1 #!/bin/sh
2 # Display system load
3 while :
4 do
5 ./lcdswitch --display "$(uptime | sed 's/.*average: //')"
6 sleep 5
7 done
```

There are a number of ways the code can be used and improved, I'll leave that as an exercise for the reader. Good luck with your own USB projects!

Please leave any feedback at the Matrix Multimedia user forums!