I think the FTDI UM245R usb to parallel module works like this!

================

Robin McKay
September 2011
V1.0

I bought the FTDI UM245R usb to parallel module as an alternative to the now obsolete parallel printer port that I had used in the past to control electronic devices. At the time I new very little about usb and I incorrectly assumed that the module would be a simple alternative to the parallel port, but with just 8 data pins.

Unfortunately the documentation available from FTDI only provides glimpses of the capabilities and restrictions of the device and I could find no comprehensive explanation of how it works and how best to use it.

This note is the outcome of many hours of reading and experimenting and may be useful for others in the same situation. I am not an expert and I may have made some mistakes, but this is how the product seems to work. It would be so nice if FTDI wrote comprehensive documentation of the same quality as Microchip's.

Because the usb system claims very high speeds I have been trying to build a rudimentary oscilloscope that did not require a memory chip for temporary storage of sampled data. This is to replace a “nearly completed” project from some years ago when I had a PC with a parallel port. I had hoped that I could issue a "read" to the module and then wait until it had collected (say) 2000 samples. However now that I understand it a bit better, the usb system does not appear to facilitate this.

I have tried to use consistent terminology for clarity. When I use the word "module" I mean the whole UM245R gadget with its USB socket and its 24 DIP pins. I use the word "chip" to mean the FT245R chip which is the principal part of the module. I use the word "PC" to mean the computer that is connected to the module using a standard usb cable. And I use the word "device" to mean the thing that the module is plugged into - perhaps a breadboard during the experimental stage. In my case the module is collecting data from an ADC chip.

I found the FTDI buffer names confusing. The transmit buffer collects data from the device for transmission to the PC - I would call this receiving (but what do I know!). And the receive buffer holds data that comes from the PC. Hence the TXE# pin goes high when the transmit buffer is full of data sent by the device and I presume the RXF# pin goes high when there is no data in the receive buffer. (*TXF and RXE might have been more appropriate names!*)

> I suspect a lot of what I have written here is also relevant to the FT232R chip and the UM232 module.

The FT245R chip and consequently the UM245R module has three operating modes - normal, asynchronous bit-bang and synchronous bit-bang.

The documentation from FTDI provides most information about synchronous bit-bang. In that mode the chip will read data from the module pins after every byte is written. You can select which pins are for output and which for input. Obviously if some pins are for input only some bits in the written byte will be relevant. Also it seems that when the PC gets data from the module each byte will include the values on the pins that were set as output as well as the input values. The PC software will need to exclude the unwanted data from the output pins.

Assuming you send (write) 10 bytes to the module it will write those onto the module pins at the rate determined by the baudrate. And as each byte is written the module will read a byte into the on-chip buffer. You retrieve the incoming bytes with a read command. If you try to send more than 384 bytes you will get a write error because the buffer for incoming data can only hold 384 bytes and it will only output a byte if it also has space to input a byte. I have not checked to see if the RXF# and TXE# control pins are used in synchronous bit bang mode – they are probably irrelevant in this mode.

I have not done much with asynchronous bit bang. I think that in this mode the chip starts reading incoming data straight away without any need to send (write) data. Presumably the TXE# pin will signal that the buffer is full so that the device sending data to the module knows to wait until the PC reads (removes) the data from the buffer. Again, as far as I know the data is read or written at the rate determined by the baudrate. Perhaps I should explain that I need to know the data rate accurately and I don't have the equipment to check what speed the baudrate clock operates at so I haven't used this mode. The FTDI documentation seems to suggest it operates at 16 times the baudrate. Another problem for me is that I don't know if the baudrate clock could be interrupted by the usb system.

Normal mode (which is what I call it - I have not seen an FTDI name for it) differs from bitbang mode because it requires an external device to provide a clock to operate the RD pin to move the data from the receive buffer to the module pins and to operate the WR pin to move data from the module pins into the transmit buffer. In this mode the TXE# pin goes high after 256 bytes have been read into the buffer. From my limited experiments it seems that the module will accept data into the transmit buffer when RD is clocked at 1Mhz but not at 1.5Mhz. I have not experimented with sending data from the PC to the module in normal mode.

In normal mode (and presumably in asynchronous bit bang mode) you must also take account of the peculiarities of the USB system. The booklet EUDBE2.01.pdf available on FTDI's website provides a good introduction. Also the formal USB documentation is worth reading and is not difficult.

When the PC program issues a read or write request it will be queued by the USB scheduling system which works in 1 millisecond intervals - so, for example, it seems to be impossible to issue read requests more frequently and it is not possible to know exactly when a request will be executed. A single read request may request several (maybe thousands of) bytes - indeed using the usb system for single-byte reads or writes is very inefficient. At its heart the USB system works on 64byte blocks and a large read will continue until a block is returned which is not full or no block is returned and the timeout period has elapsed (default 5000 milliseconds). If a partial block is returned there seems to be no mechanism to make a read wait for the full timeout until all the requested bytes have been supplied.

The FTDI chip appears to interact with the USB system in the following way. In response to a read it sends data to the PC as soon as it has a block of 64 bytes of data or when an on-chip timeout expires. The default for this is 16 milliseconds and you can vary it between

2 and 255 milliseconds. The 64 byte block comprises 62 bytes of your data preceded by 2 bytes inserted by the FTDI chip. Those two bytes get stripped out and I don't know what they contain. If there is less than 62 bytes in the transmit buffer when the timeout expires the chip sends whatever is in the buffer (which could be empty) preceded by the 2 FTDI bytes. This partial block will terminate the USB read.

Programming Functions

I am using Ubuntu 10.10 and libftdi which, in turn uses libusb_0.1.x.

The relevant libftdi functions for reading data from the module are

ftdi_set_bitmode
ftdi_set_baudrate
ftdi_set_latency_timer
ftdi_read_data
ftdi_read_data_set_chunksize

There is also an attribute called **usb_read_timeout** (and usb_write_timeout).

Ftdi_set_bitmode is used to set the mode and which pins are for input and output. The FTDI Bit-bang and D2XX Programmers Guide documentation describe this adequately.

Ftdi_set_baudrate may not be as obvious as its simple name suggests. The FTDI documentation seems to say that the on chip clock operates at 16 times the baudrate but I don't have the equipment that would be necessary to verify this. The baudrate is relevant for bit-bang modes.

Ftdi_set_latency_timer sets the on-chip timeout. The default is 16 milliseconds and this can be changed to anything from 2 to 255 milliseconds.

Ftdi_read_data does what is expected except that the requested number of bytes is a maximum and fewer (including 0) may be returned. The number retrieved is given by the return value - e.g. n = ftdi_data_read(....).

Ftdi_read_data_set_chunksize sets a value that is used by libusb. The default is 4k bytes but libusb has a default of 16k. So unless you use a value greater than 16k it seems to be irrelevant.

Finally there is the attribute **usb_read_timeout** which has a default value of 5000 milliseconds. I program in Ruby (using the ftdi gem) and I'm not sufficiently familiar with C to know how you change this in C. In any case this is only used by libusb to end a read if there is no response from the module. And as noted above the on-chip timeout is much shorter and will be the relevant timeout in virtually all cases.

Further Reading

For people with insominia the above information was gleaned from the following:

FTDI documents (http://www.ftdichip.com/Support/FTDocuments.htm)

AN232B-03_D2XXDataThroughput.pdf
AN232B-04_DataLatencyFlow.pdf
AN232B-05_BaudRates.pdf
AN_232R-01_V2_02_Bit_Bang_Mode_Available_For_FT232R_and_Ft245R.pdf
EUDBE2.01.pdf (Technical Publication Embedded USB Design by Example)
D2XX_Programmer's_Guide(FT_000071).pdf
DS_FT245R.pdf
DS_UM245R.pdf

USB specification (http://www.usb.org/developers/docs/)
usb_20.pdf

Source code

lbftdi – ftdi.c	(http://www.intra2net.com/en/developer/libftdi/)
libusb_0.1.4 – linux.c	(http://www.libusb.org/)

END