# USB1 communications interface for controlling instruments

## 1. Introduction

Numerous optical instruments and indeed many other types of instruments use a variety of motorised parts which require some from of control and interfacing to host computers. Devices which use motors etc. rarely require a fast communications link. In the 'old' days, the speed of links based on parallel printer ports or serial links such as RS232 and RS485 were quite sufficient. These days, most computers no longer have such ports and the ubiquitous presence of the Universal Serial Bus, USB, in its various forms, USB1, USB2, USB3 (http://en.wikipedia.org/wiki/USB, http://www.usbmadesimple.co.uk/, http://www.howstuffworks.com/usb.htm) cannot be ignored.

We describe here some hardware which we first developed in the mid-late 1990's with the intention to use this for the vast majority of projects undertaken by our group where relatively low speeds are required, i.e. for 'control' rather than for 'data acquisition'. Of course, when data rates are low, the system described here can be used for data acquisition as well. In some cases large and complex instruments are placed on the end of this hardware. The largest system used with this is a linear accelerator, but in general, various forms of optical instrumentation are commonly used, for example automated microscopes.

As its name implies, USB is a serial bus, while instruments require a large number of parallel lines, or individual lines. If the number of lines can be defined, then other approaches may be more suitable; in our work however, we do not have access to a crystal ball (!) and we wanted to develop a system which was inherently expandable since it is very rare that we know ahead of time what we might need. USB is a point-to point bus and what is really needed is a bus that can be daisy-chained or operated in a 'star' arrangement. A convenient bus to achieve inter-instrument control at moderate speeds is the $I^2C$ bus (http://en.wikipedia.org/wiki/I2C, http://www.i2c-bus.org), originally developed by Philips for 'Inter-Integrated Circuit' applications in consumer and other applications. This bus protocol is extremely rugged and can in fact operate over significant distances. Addition of cables results in increased capacitive load but the degradation is graceful: speed drops, but operation is maintained. Cable length should of course always be minimised, but in practice, most drivers allow operation at 100 kb/s without problems.

We chose to use a 6 pin mini DIN cable to interconnect separately cased devices. There are few things in life less interesting than making up cables and our choice of cable was partly based on the availability of low cost 'keyboard' cables which use such a mini-DIN connector. We also standardised on providing a +15V power supply in the cable to allow remote powering of devices. This choice was based on the fact this voltage is suitable for operating most motors, solenoids etc. The supplies to remote electronics can be locally regulated to +5V or to +12V for powering operational amplifiers or other signal processing circuits (we generate negative voltages using charge pumps or DC-DC converters. The $I^2C$ bus requires a serial bidirectional clock line and a serial bidirectional data line and operates in an unbalanced mode. We are thus left with two uncommitted lines which can be used for specific purposes (e.g. when fast trigger signals are required). The slave $I^2C$ devices are provided with two 6 pin sockets, such that additional devices can be daisy-chained.

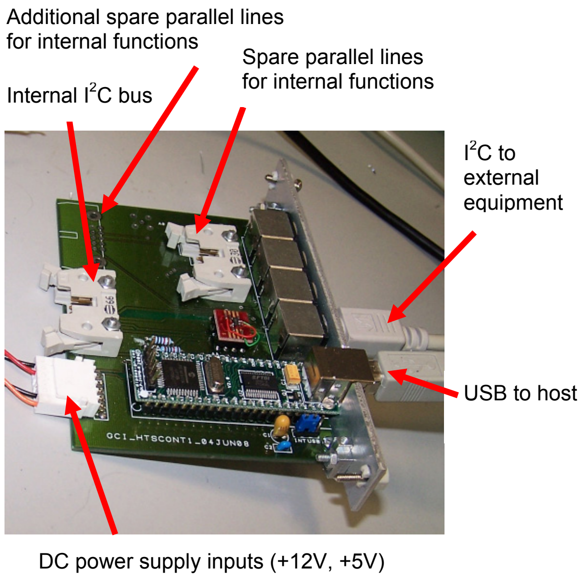



Figure 1: The USB interface board

However, this is not always possible due to space limitations so we provide multiple sockets at the main interface, as shown in Figure 1. This interface is constructed on a printed circuit attached to a 3U ‘Eurocard’ panel (5HP width, 25.4 mm) so as to make it compatible with the rack systems which we routinely use.

## 2. Interface circuit

The full circuit of the interface is shown in Figure 2. We use a readily available sub-assembly, DLP245, a simple USB-FIFO interface which includes a PIC Microchip 16F877A microcontroller (http://www.dlpdesign.com/usb/245pb.shtml), available from Mouser (http://uk.mouser.com) or Digikey (http://search.digikey.com). These modules are also available at reduced cost from http://www.ftdichip.com/Products/Modules/DLPModules.htm.

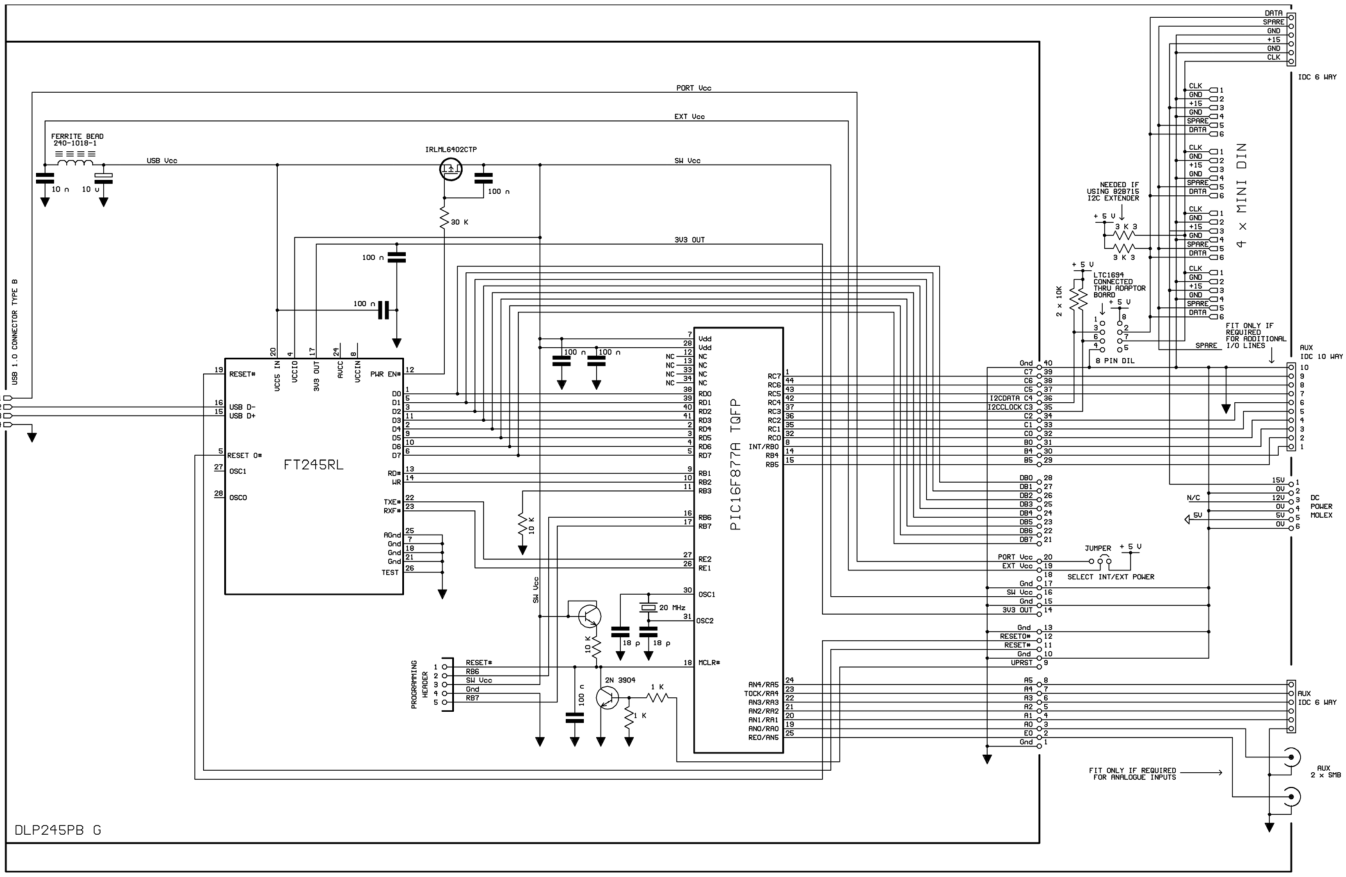
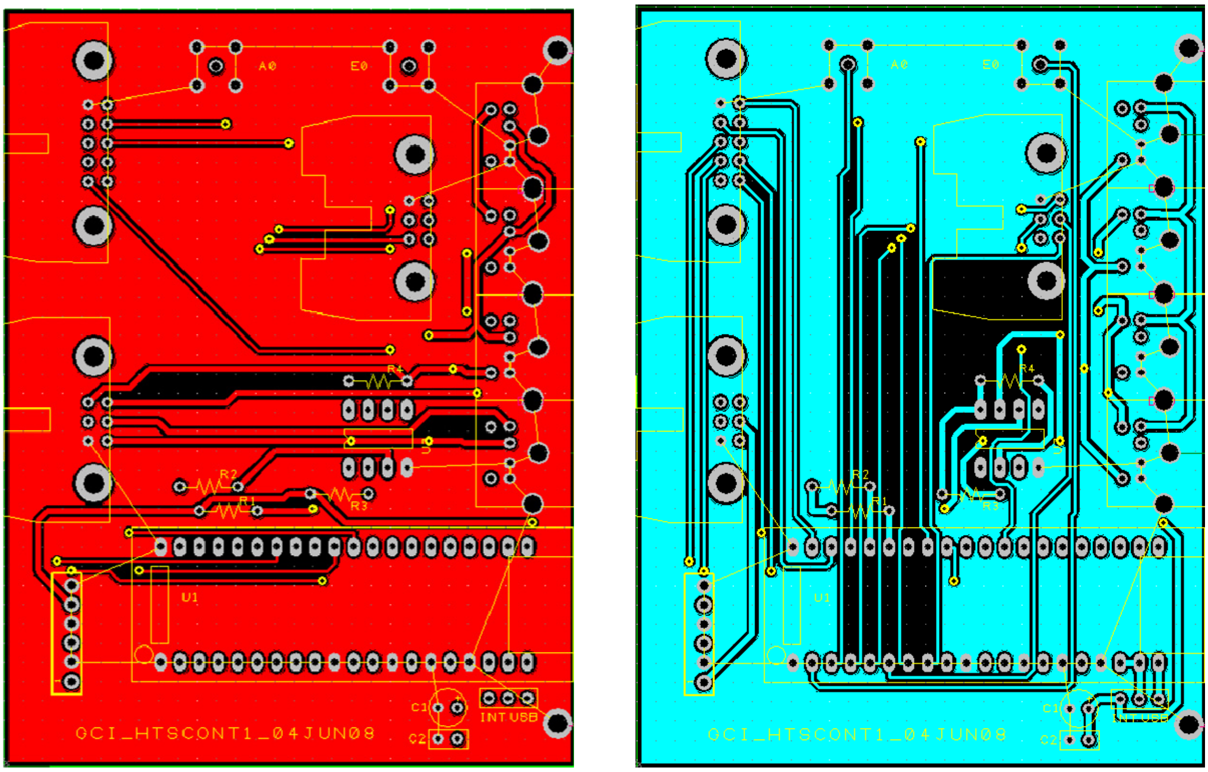


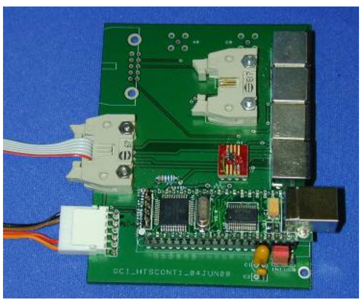

Figure 2: Circuit diagram of the USB-I$^2$C interface.

The 8 pin DIL socket on the diagram can be used in a number of ways. An NXP P82B715 bus extender chip may be used if the board is used to drive long capacitive cables but needs a corresponding chip to convert back to I$^2$C signals, a Linear technology LTC1694 bus accelerator may also be used to enhance data transmission with longer cables by providing active pull-up on the control lines. The board may also be used by putting a jumper across the pins and using directly on short cables, although a resistor (~100 Ω- 1 kΩ) may be used as a link giving some protection from a higher voltage being accidentally applied to the data lines and damaging the PIC. Diodes from Vdd and from ground may are also suggested. A small DIL interface printed circuit board header/plug is used to insert the required components.

## 3. Printed circuit board

The printed circuit board is double-sided, 100 x 75 mm, designed with Number One Systems EasyPC (http://www.numberone.com/). We use PCB pool for board manufacture (http://www.pcb-pool.com/ppuk/info.html) and board assembly is straightforward. Although rarely used, additional parallel i/o lines are made available on this board. In addition, we have provided two SMB coaxial inputs which can be used to inject analogue voltages into two of the PIC's internal analogue to digital converters. However, the code described later does not support these lines as to date we have not had to use them. When we do, code will be released!



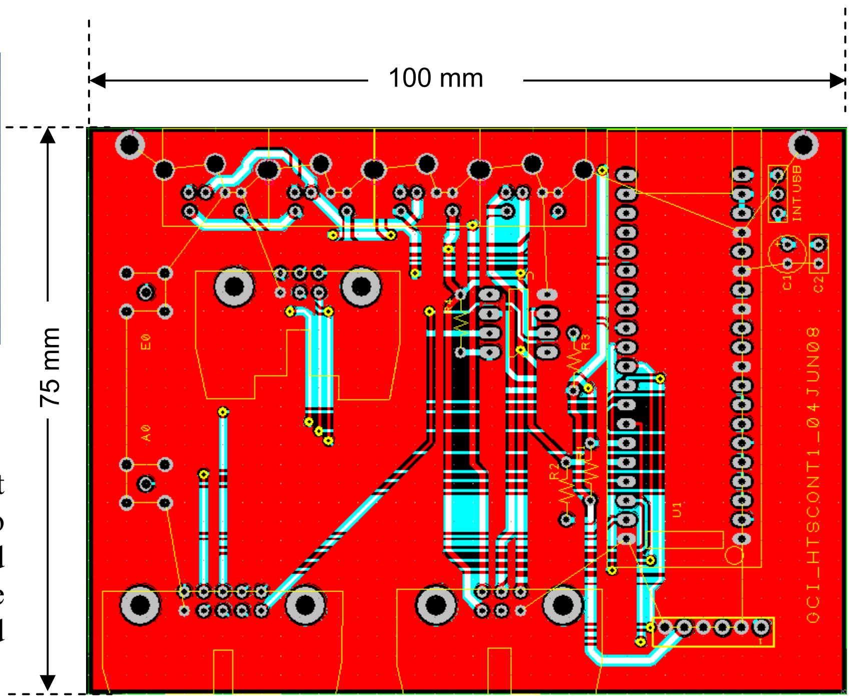



Figure 3: Printed circuit board layouts of the two sides of the double-sided printed circuit board and the USB-I$^2$C interface board during assembly.

## 4. Software and PIC firmware

Correct operation of this device is dependant on several software modules and an outline description of these is provided here. Figure 4 is a diagram of how the various components interact with the hardware. The software used with this device consists of three components: assembler code PIC firmware, USB driver software (downloaded from http://www.ftdichip.com), high level code running on the host computer and a header (.h) file which defines whether data sent or received is associated with $I^2C$, RS232 or dedicated lines. Although in this application, we do not use the serial RS232 output port, the .h and the assembler code allow for this.

In many instances, the high level C-code is developed using National Instruments' LabWindows environment; high level code examples are provided later.

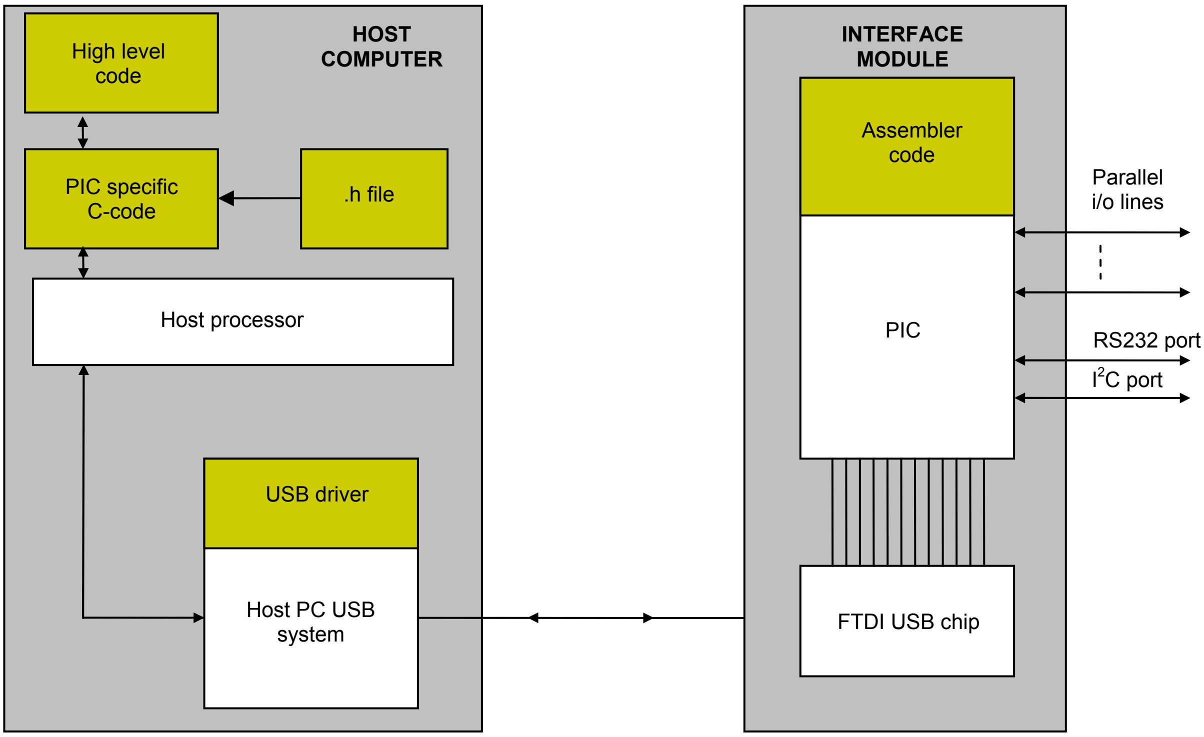



Figure 4: Arrangement of software components, shown in dark yellow, used in the interface unit and the host processor.

The code for the 16F877A microcontroller was written in assembler code using Microchip MPLAB software using an ICD2 programmer which allows debugging of the code. The listing is shown below. It was modified at various times, by various colleagues, to include additional features. We note that no interrupts can be handled and that the $I^2C$ devices placed on the bus can only act as slave devices. In practice this is not a serious limitation in our applications, where we expect the host PC to always act as master and keep track of the various instrument operations. However in some instruments, we reached the address limit of various $I^2C$ devices. In this instance, several busses can be made by using a $I^2C$ controlled analogue switch to route signals appropriately. A suitable device is the Maxim MAX4562 and an application using this can be found on the Maxim website: http://www.maxim-ic.com/app-notes/index.mvp/id/955.

After initializing the internal PIC registers and pin configurations the program cycles in a loop, continually checking if data have arrived from the USB connection or an interrupt has occurred from the $I^2C$ bus. If data are available from host computer, a command byte is read which will signal to the PIC whether I/O data lines, $I^2C$ or RS232 communications are required. The program then branches; if I/O data lines are required, another byte is read which will indicate how many more bytes will follow. These data bytes are subsequently read out in pairs, the first byte configures the port pins to be inputs or outputs. The second byte immediately reads from or writes to port pins. A byte is sent back to the host if any lines have been read. This is repeated until all the expected bytes have been processed and an acknowledge byte is returned at the end of the process.

If the command byte is for $I^2C$ communications, another byte is read which will signal to the PIC how many data bytes to expect. The next read byte is the address of the $I^2C$ device, incorporating whether we want to read from or write to the device. Subsequent expected bytes are read. An $I^2C$ sequence is activated to either write data to, or read from a device. At the end of a write sequence a byte is returned to the host indicating a successful write or any errors. At the end of a read sequence, several bytes are usually returned from the device to the host computer along with the success or error byte.

For RS232 communication the first byte read signals to the PIC the number data bytes to expect. These are subsequently read one byte at a time and sent out onto the RS232 bus, no handshaking is involved and no status byte is returned at the end. If returned RS 232 communication is expected this will be automatically available for reading by the host computer. The PIC can only buffer 3 bytes so care must be taken not to send too many commands. The baud rate is set to 9600 but can be changed in the code to other values. Handshaking can be implemented by un-commenting the #define handshake line, RTS CTS lines need to be connected and the host computer needs to read before writing to check whether the previous byte has been cleared. This is not usually needed for simple control systems.

The PIC assembler code is shown below:

```
;************************************************************************************************************************************
;   This file is the source code to the general purpose i2c expander- board.                    *
;   with fast line return values                                                                *
;   Code converted to use DLP2456PB with 16F877 using different control lines                   *
;  Acknowledge return bytes sent after fast read and writes added                               *
;************************************************************************************************************************************
;                                                           *
;    Filename:      DLP_v2.asm                              *
;    Date:          14.12.2007                              *
;    File Version:  initial                                 *
;    Author:        Manser Andreas/Rob Newman               *
;    Company:       Graylab                                 *
;                                                           *
;                                                           *
;************************************************************************************************************************************
list      p=16f877                                          ; list directive to define processor
#include <p16f877.inc>                                      ; processor specific variable definitions
errorlevel -302                                             ; suppress bank warning

__CONFIG _CP_OFF & _WDT_ON & _BODEN_OFF & _PWRTE_ON & _HS_OSC & _WRT_ENABLE_OFF & _LVP_OFF & _CPD_OFF

; '__CONFIG' directive is used to embed configuration data within .asm file.
                                                            ; The labels following the directive are located in the respective .inc
file.
                                                            ; See respective data sheet for additional information on configuration
word.
#define baudrate    19200

;**** DEFINES
freq_osc equ 20000000                                       ; define constants
                                                            ; calculates baudrate when BRGH = 1, adjust for rounding errors
#define CALC_HIGH_BAUD(baudrate)(((10*freq_osc/(16*baudrate))+5)/10)-1
                                                            ; calculates baudrate when BRGH = 0, adjust for rounding errors
#define CALC_LOW_BAUD(baudrate)(((10*freq_osc/(64*baudrate))+5)/10)-1
                                                            ; for now this doesn't work so you have to calculate with above formulas
                                                            ; if the result you get is bigger then 255 so use LOW formula and define
low_br below

#define baud                  d'129'                        ; (10 is 115200) in hsmode 129->9600
#define high_br                                             ; mode
;#define            handshake                               ; with or without hardware handshake
#defineRXF                    d'1'                          ; pin defines
#define TXE                   d'2'                          ; for usb2mod
#defineWR           d'2'
#define READ        d'1'
#define RTS                   d'2'                          ; RS232 pins
#defineCTS                    d'1'

#define interrupt   d'0'                                    ; flags defines
#define ack_error   d'1'
#define done                  d'2'
#define bus_coll    d'3'
#define busy                  d'7'

#define C_FAST      0x01                                    ; define commands
#define C_I2C       0x02
#define C_RS232     0x03

file
```

```
i2cSizeMask         EQU       0x0F

;#include baudmacro.inc

;***** VARIABLE DEFINITIONS
w_temp              EQU       0x20                          ; variable used for context saving
status_temp         EQU       0x21                          ; variable used for context saving
flags     equ       0x7f                                    ; flags
i2c_status          equ       0x7e                          ; i2c_status variable
command   equ       0x7d                                    ; holds the command
count     equ       0x7c                                    ; databytes count
ptr       equ       0x7b                                    ; pointer to data
address   equ       0x69                                    ; i2c device address
datacount equ       0x79                                    ; number of bytes written/read on i2c
tdatacount          equ       0x76                          ; may be destroyed
tcount    equ       0x75                                    ; same
retdata   equ       0x70                                    ; returned data
                                                            ; reserved: 0x6d 0xeb-0xf0 (used for icd)
                                                            ; used for datastorage: 0xa0-0xec= 74 bytes
;**************************************************************************************************************************************
                    ORG        0x000                        ; processor reset vector
                    nop                                     ; used for icd!
                    clrf       PCLATH                       ; ensure page bits are clear
                    goto      init                          ; go to beginning of program

                    ORG       0x004                         ; interrupt vector location
                    movwf      w_temp                       ; save off current W register contents
                    movf      STATUS,w                      ; move status register into W register
                    bcf       STATUS,RP0                    ; ensure file register bank set to 0
                    movwf     status_temp                   ; save off contents of STATUS register

                    ; TEST FOR COMPLETION OF VALID I2C EVENT
                    bsf       STATUS,RP0                    ; select SFR bank
                    btfss     PIE1,SSPIE                    ; test is interrupt is enabled
                    goto      test_buscoll                  ; no, so test for Bus Collision Int
                    bcf       STATUS,RP0                    ; select SFR bank
                    btfss      PIR1,SSPIF                           ; test for SSP H/W flag
                    goto      otherints                     ; test for other interrupts
                    bcf       PIR1,SSPIF                    ; clear SSP H/W flag
                    pagesel   service_i2c                   ; select page bits for function
                    call       service_i2c                  ; service valid I2C event
                    goto      test_buscoll                          ; see if i2c event started while other not completed
otherints
                    ; TEST FOR COMPLETION OF RS232 EVENT this routine may require modification, but works well-enough for simple
RS232 communication!
                    bsf       STATUS,RP0                    ; select bank1
                    btfsc     PIR1,TXIF                     ; if event=Transmitted
                    call      service_Trs232                ; service event
                    nop
                    btfsc     PIR1,RCIF                     ; if event=received
                    call      service_Rrs232                ; then service receive event

                    ; TEST FOR I2C BUS COLLISION EVENT
test_buscoll
                    bcf       STATUS,RP0                    ; select SFR bank
                    btfss     PIR2,BCLIF                    ; test if Bus Collision occurred
                    goto      exit_isr                      ; no, so go on
                    bcf       PIR2,BCLIF                    ; yes: clear Bus Collision H/W flag
                    call      service_buscoll               ; service bus collision error
exit_isr
                    bcf       STATUS,RP0                    ; ensure file register bank set to 0
                    movf      status_temp,w                 ; retrieve copy of STATUS register
                    movwf     STATUS                        ; restore pre-isr STATUS register contents
                    swapf     w_temp,f
                    swapf     w_temp,w                      ; restore pre-isr W register contents
                    retfie                                  ; return from interrupt
;*******************Start of Program**************************************************************************************************
init                                                        ; Initialize Program after startup, Errors
                    clrf      flags                         ; initialize variables
                    clrf      i2c_status
                    clrf      count
                    clrf      SSPBUF
                    movlw     h'A0'
                    movwf     ptr
                    movwf     FSR
                    bcf       STATUS,IRP                    ; ensure that INDF points to bank0/1
                    call      INIT_UART
                                                            ; initialize port directions
                    clrf      PORTE                         ; Reset Control port
                              clrf      PORTB               ; Reset Control port
                    clrf      PORTA                         ; Reset fast lines
                    bsf       PORTB,READ                    ; initially high
                    bcf       PORTB,WR  ; initially low
                    bsf       STATUS,RP0                    ; select bank1
                    movlw     b'01011011'                   ; enable RB-pullup, enable wdt
                    movwf     OPTION_REG                    ; time out is 112-528 ms (was '011'timeout is 56-264 ms)

                    movlw     h'0E'                         ; all pins of PORTA
                    movwf     ADCON1                        ; are used as ditital i/o except A0 (A/D)
                    movlw     b'00000110'                   ; TXE and RXF are inputs
                    movwf     TRISE                         ; RE0 output
                    movlw     b'11111001'                   ; RD and WR outputs rest
                    movwf     TRISB                         ; are inputs

                    movlw     h'ff'
                    movwf     TRISD                         ; initiate PORTD as input
                    movwf     TRISA                         ; initiate PORTA as input
                    movlw     b'11111011'                   ; 0:interrupt=in, 1:cts=in, 2:rts=out, 3,4:i2c=in
                    ;MOVLW    B'11011000'                   ; 5:dtr=unused, 6,7:rs232=in
                    movwf     TRISC                         ;

                                                            ; initialize i2c
                    movlw     b'00101000'                   ; enable MSSP & mode=3 -> i2c-master
                    bcf       STATUS,RP0                    ; select bank0
                    movwf     SSPCON                        ; write it
```

```
                    bsf      STATUS,RP0                    ; select bank1
                    movlw    d'49'                         ; set bitrate=100kBit/s
                    movwf    SSPADD                        ; SSPADD=((Fosc/100k)/4)-1 = 49
                    bsf      SSPSTAT,SMP                   ; no slewrate control
                    bcf      STATUS,RP0                    ; select bank0

                                                           ; initialize interrupts
                    bsf      STATUS,RP0                    ; select bank1
                    bsf      PIE1,SSPIE                    ; allow MSSP int
                    bsf      PIE2,BCLIE                    ; allow BusColl int
                    bsf      INTCON,PEIE                   ; enable peripheral ints
                    bsf      INTCON,GIE                    ; general int enable
                    bcf      STATUS,RP0                    ; select bank0
;***************Main
Loop*************************************************************************************************************************
****************
main_loop
                    clrwdt                                 ;clear watchdog timer
                    btfss    flags,busy
                    call     checkCommand                  ; see if new command redy
                    call     checkBusInt                   ; was there an interrupt?
                    btfsc    flags,interrupt               ; if (interrupt detected)
                    call     handle_interrupt              ; then handle interrupt
                    goto     main_loop                     ; ok, lets do it again

;=======================see if there was an Interrupt on I2C Bus (no interrupt for pic) ============================
checkBusInt
                    btfss    PORTC,RP0                     ; if (RP0=0)  (int is low-active)
                    bsf      flags,interrupt               ; then set Interrupt Flag
                    return                                 ; else return
;=======================Check which Command has been sent (if any) ======================================
checkCommand
                    btfsc    PORTE,RXF                     ; if (no data available)
                    return                                 ; then go back to main_loop
                    bcf      PORTB,READ                    ; else start reading
                    movfw    PORTD                         ; read data
                    movwf    command                       ; store data
                    bsf      PORTB,READ                    ; get next databyte
                    movlw    C_FAST                        ; get pattern of command FAST
                    subwf    command,w                     ; storage-C_FAST
                    btfsc    STATUS,Z                      ; if (command=fast)
                    goto     handle_fast                   ; then handle fast
                    movlw    C_I2C                         ; else check other commands
                    subwf    command,w                     ; storage-C_I2C
                    btfsc    STATUS,Z                      ; if (command=i2c)
                    goto     handle_i2c                    ; then handle i2c
                    movlw    C_RS232                       ; else check oder commands
                    subwf    command,w                     ; storage-RS232
                    btfsc    STATUS,Z                      ; if (command=rs232)
                    goto     handle_rs232                  ; then handle rs232
                    return                                 ; else return (if none found)

          IFDEF     handshake
checkCTS232
                    btfsc    PORTC,CTS                     ; if cts=1
                    bsf      PORTC,RTS                     ; then set RTS
                    return
          ENDIF

;======================Routines to handle Each command go here==========================================
handle_done
                    return

handle_fast                                                ; code for fast lines
                    btfsc    PORTE,RXF                     ; if (data not ready)
                    goto     $-1                           ; then test again
                    bcf      PORTB,READ                    ; read number of databytes
                    movfw    PORTD                         ; fetch data
                    movwf    count                         ; save data
                    bsf      PORTB,READ                    ; end readdata
                    btfsc    PORTE,RXF                     ; if (data not ready)
                    goto     $-1                           ; then test again
                    bcf      PORTB,READ                    ; read "FAST-byte"
                    movfw    PORTD                         ; fetch it
                    movwf    address                       ; save.
                    movwf    retdata                       ; save to return data.
                    bsf      PORTB,READ                    ; end reading
                    btfsc    address,0                     ; if (config)
                    goto     do_config                     ; then configure directions
test_fast                                                  ; else get next byte(s)
                    decfsz   count                         ; if (--count!=0)
                    goto     rw                            ; read or write
                    call     ackret                        ; acknowledgement returned at end
                    return                                 ; else done
rw
                    btfsc    PORTE,RXF                     ; see if module ready
                    goto     $-1                           ; else try again
                    bcf      PORTB,READ                    ; start reading
                    movfw    PORTD                         ; get data from USB
                    btfss    PORTD,0                       ; see if read or write
                    goto     read_fast
                    bsf      PORTB,READ                    ; stop reading
                    btfsc    PORTE,RXF                     ; see if module ready
                    goto     $-1                           ; else try again
                    bcf      PORTB,READ                    ; start reading
                    movfw    PORTD                         ; get data from USB
                    movwf    address                       ; save in temp
                    ;movwf   retdata                       ; save returned data.
                    movwf    PORTA                         ; ok as only writing to RA1 to RA5
                    rrf      address                       ; shift RB5, RB4 back
                    rrf      address
                    bsf      address,READ                  ; keep READ high
                    bcf      address,WR                    ; keep WR low
                    movfw    address
                    movwf    PORTB                         ; output RB5, RB4
```

```
                nop
                bsf       PORTB,READ                     ; stop reading TEST!!!!
                bcf       STATUS,RP0                     ; select bank0
                decf      count                          ; we have read 2 bytes!
                goto      test_fast                      ; (more to be done?)

do_config
                bsf       address,0                      ; RA0 is input analogue input possible
                btfsc     address,7                      ; if (dirB5=1)
                goto      SetriB5                        ; then set TRISB,5
                bsf       STATUS,RP0                     ; select bank1
                bcf       TRISB,5                        ; RB5 is output
                bcf       STATUS,RP0                     ; select bank0
                btfsc     address,6                      ; if (dirB4=1)
                goto      SetriB4                        ; then set TRISA,4
                movfw     address                        ; place data in w
                bsf       STATUS,RP0                     ; select bank1
                movwf     TRISA                          ; set TRISA
                bcf       TRISB,4                        ; RB4 is output
                bcf       STATUS,RP0                     ; select bank0
                movlw     0x01                           ; move 1 into w
                subwf     count                          ; subtract from count
                btfsc     STATUS,Z                       ; checks if zero
                call      ackret                         ; acknowledgement returned at end
                bcf       STATUS,Z                       ; clears z bit
                movlw     0x01                           ; move 1 into w
                bcf       STATUS,RP0                     ; select bank0
                addwf     count                          ; restore count
                return
SetriB5
                bsf       STATUS,RP0                     ; select bank1
                bsf       TRISB,5                        ; RB5 is input
                bcf       STATUS,RP0                     ; select bank1
                btfsc     address,6                      ; if (dirB4=1)
                goto      SetriB4                        ; then set TRISB,4
                movfw     address                        ; place data in w
                bsf       STATUS,RP0                     ; select bank1
                movwf     TRISA                          ; set TRISA
                bcf       TRISB,4                        ; RB4 is output

                bcf       STATUS,RP0                     ; select bank0
                movlw     0x01                           ; move 1 into w
                subwf     count                          ; subtract from count
                btfsc     STATUS,Z                       ; checks if zero
                call      ackret                         ; acknowledgement returned at end
                bcf       STATUS,Z                       ; clears z bit
                movlw     0x01                           ;move 1 into w
                bcf       STATUS,RP0                     ; select bank0
                addwf     count                          ;restore count
                return
SetriB4
                movfw     address                        ; place data in w
                bsf       STATUS,RP0                     ; select bank1
                movwf     TRISA                          ; set TRISA
                bsf       TRISB,4                        ; RB4 is input
                bcf       STATUS,RP0                     ; select bank0
                movlw     0x01                           ; move 1 into w
                subwf     count                          ; subtract from count
                btfsc     STATUS,Z                       ; checks if zero
                call      ackret                         ; acknowledgement returned at end
                bcf       STATUS,Z                       ; clears z bit
                movlw     0x01                           ; move 1 into w
                bcf       STATUS,RP0                     ; select bank0
                addwf     count                          ; restore count
                return                                   ; done
read_fast
                bsf       PORTB,READ                     ; stop reading
                movfw     PORTA                          ; fetch port A
                movwf     address
                btfsc     PORTB,5                        ; if (RB5=1)
                goto      setrb5                         ; set it in outbyte
                bcf       address,7                      ; else clear in outbyte
                btfsc     PORTB,4                        ; if (RB4=1)
                goto      setrb4                         ; set it
                bcf       address,6                      ; else clear it
                bsf       STATUS,RP0                     ; select bank1
                clrf      TRISD                          ; portd is output now
                bcf       STATUS,RP0                     ; select bank0
                btfsc     PORTE,TXE                      ; see if module ready
                goto      $-1                            ; else try again
                bsf       PORTB,WR                       ; start writing
                movfw     address
                movwf     PORTD                          ; send data to USB
                bcf       PORTB,WR                       ; ACTUALLY WRITE
                bsf       STATUS,RP0                     ; select bank1
                movlw     0xff                           ; load 0xff
                movwf     TRISD                          ; portd is input now
                bcf       STATUS,RP0                     ; select bank0
                goto      test_fast
setrb5
                bsf       address,7
                btfsc     PORTB,4                        ; if (RB4=1)
                goto      setrb4                         ; set it
                bcf       address,6                      ; else, clear it
                bsf       STATUS,RP0                     ; select bank1
                clrf      TRISD                          ; portd is output noaddress
                bcf       STATUS,RP0                     ; select bank0
                btfsc     PORTE,TXE                      ; see if module ready
                goto      $-1                            ; else try again
                bsf       PORTB,WR                       ; start writing
                movfw     address
                movwf     PORTD                          ; send data to USB
                bcf       PORTB,WR                       ; ACTUALLY WRITE
                bsf       STATUS,RP0                     ; select bank1
                movlw     0xff                           ; load 0xff
```

```
                movwf   TRISD                   ; portd is input now input
                bcf     STATUS,RP0              ; select bank0
                goto    test_fast

setrb4          bsf     address,6
                bsf     STATUS,RP0              ; select bank1
                clrf    TRISD                   ; portd is output now
                bcf     STATUS,RP0              ; select bank0
                btfsc   PORTE,TXE               ; see if module ready
                goto    $-1                     ; else try again
                bsf     PORTB,WR                ; start writing
                movfw   address
                movwf   PORTD                   ; send data to USB
                bcf     PORTB,WR                ; ACTUALLY WRITE
                bsf     STATUS,RP0              ; select bank1
                movlw   0xff                    ; load 0xff
                movwf   TRISD                   ; portd is input now
                bcf     STATUS,RP0              ; select bank0
                goto    test_fast

ackret          bsf     STATUS,RP0              ; select bank1
                clrf    TRISD                   ; portd is output now
                bcf     STATUS,RP0              ; select bank0
                btfsc   PORTE,TXE               ; see if module ready
                goto    $-1                     ; else try again
                bsf     PORTB,WR                ; start writing
                movlw   0x04                    ; load 0x04 (acknowledge)
                movwf   PORTD                   ; send data to USB
                bcf     PORTB,WR                ; ACTUALLY WRITE
                nop
                bsf     STATUS,RP0              ; select bank1
                movlw   0xff                    ; load 0xff
                movwf   TRISD                   ; portd is input now
                bcf     STATUS,RP0              ; select bank0
                return                          ;
handle_rs232                                    ; code for handling rs232 requests
                bsf     flags,busy              ; now we're busy
                bcf     PIR1,TXIF               ; clear before enable
                btfsc   PORTE,RXF               ; if (data not ready)
                goto    $-1                     ; then test again
                bcf     PORTB,READ              ; read number of databytes
                movfw   PORTD                   ; fetch data
                movwf   count                   ; save data
                movwf   tcount                  ; copy into temp-count
                bsf     PORTB,READ              ; prepare next read
                bsf     STATUS,RP0              ; select bank1
                bsf     PIE1,TXIE               ; enable tx interrupt -> int is served
                bcf     STATUS,RP0              ; select bank0
                return                          ; rest goes int-driven
handle_interrupt                                ; i2c interrupt handling goes here
                bcf     flags,interrupt         ; after handling interrupt, clear flag
                return
handle_i2c                                      ; code for handling i2c-request
                btfsc   flags,busy              ; busy with i2c?
                return                          ; yes, so return
                bsf     flags,busy              ; no, so now we are
                btfsc   PORTE,RXF               ; if (data not ready)
                goto    $-1                     ; then test again
                bcf     PORTB,READ              ; read number of databytes
                movfw   PORTD                   ; fetch byte
                movwf   datacount               ; save to temporary count
                movwf   count                   ; count for i2c
                decf    datacount               ; address--
                bsf     PORTB,READ              ; get next databyte
                btfsc   PORTE,RXF               ; if (data not ready)
                goto    $-1                     ; then test again
                bcf     PORTB,READ              ; read adress
                movfw   PORTD                   ; fetch byte
                bsf     PORTB,READ              ;
                movwf   address                 ; save address
                btfsc   address,0               ; if mode=read
                goto    read_i2c                ; then read it
retrieve_data
                movfw   count                   ; else save rest of data
                movwf   tcount                  ; restore tcount
retdat1
                decfsz  tcount                  ; if (--count=!0) //first byte is adrs so ok
                goto    save_data               ; then save further data
                clrf    i2c_status              ; write i2c
                bsf     PIR1,SSPIF              ; simulate i2c interrupt
                clrf    command                 ; clear command variable
                return                          ; return to main loop
save_data
                btfsc   PORTE,RXF               ; if (data not ready)
                goto    $-1                     ; then test again
                bcf     PORTB,READ              ; read number of databytes
                movfw   PORTD                   ; fetch byte
                movwf   INDF                    ; save to ram
                incf    FSR                     ; ptr++
                bsf     PORTB,READ              ; get next byte
                goto    retdat1                 ; see if there's more
read_i2c
                movlw   d'05'                   ;
                movwf   i2c_status              ; i2c_status = 5 means reading
                bsf     PIR1,SSPIF              ; simulate i2c interrupt
                clrf    command                 ; command done
                return                          ; return to mainloop
                banksel ptr                     ; select GPR bank
                movf    ptr,W                   ; retrieve ptr address
                movwf   FSR
                movfw   datacount
                movwf   tcount
                return                          ; return to mainloop
;***********************End Handle I2C*****************************************************************************************
```

```
;*********************Interrupt Service Routines are located here*************************************************************
;-----------------------------------------------------------------------------------------------------------------------------------
;**********************  I2C Service  ************************************************************************************
;-----------------------------------------------------------------------------------------------------------------------------------

service_i2c
                    movlw     high  I2CJump               ; fetch upper byte of jump table address
                    movwf     PCLATH                      ; load into upper PC latch
                    movlw     i2cSizeMask
                    banksel   i2c_status                  ; select GPR bank
                    andwf     i2c_status,w                ; retrieve current I2C state
                    addlw     low  (I2CJump + 1)          ; calculate state machine jump address into W
                    btfsc     STATUS,C                    ; skip if carry occurred
                    incf      PCLATH,f                    ; otherwise add carry
I2CJump                                                   ; address were jump table branch occurs, this addr also used in fill
                    movwf     PCL                         ; index into state machine jump table
                                                          ; jump to processing for each state= i2c_status value for each state

                    goto      WrtStart                    ; write start sequence                  =  0
                    goto      SendWrtAddr                 ; write address, R/W=1                  =  1
                    goto      WrtAckTest                  ; test ack, write data                  =  2
                    goto      WrtStop                     ; do stop if done                       =  3
                    goto      WrtDone                     ; send Data to USB                      =  4
                    goto      ReadStart                   ; write start sequence                  =  5
                    goto      SendReadAddr                ; write address, R/W=0                  =  6
                    goto      ReadAckTest                 ; test acknowledge after address        =  7
                    goto      ReadData                    ; read more data                        =  8
                    goto      ReadStop                    ; generate stop sequence                =  9
                    goto      ReadDone                    ; send Data to US                       = 10
                    goto      WrtStart                    ; write start sequence                  = 11
                    goto      SendWrtAddr                 ; write address, R/W=0                  = 12
                    goto      WrtAckTest                  ; test ack,write data                   = 13
                    goto      WrtRestart_5842             ; write restart sequence                = 14
                    goto      WrtDone                     ; send Data to USB                      = 15


I2CJumpEnd
        Fill (return),  (I2CJump-I2CJumpEnd) + i2cSizeMask

service_buscoll
                    bsf       flags,bus_coll
                    movlw     d'5'                        ;
                    subwf     i2c_status,f                ; i2c_status-5
                    btfss     STATUS,C                    ; if (i2c_status-5>0)
                    goto      ReadDone                    ; then Send "requested data back"
                    goto      WrtDone                     ; else just return the flags
                    goto      init                        ; re-initialize variables
                                                          ; bus_coll occurred during read
;-----------------------------------------------------------------------------------------------------------------------------------
;   ******************** Write data to Slave   ***************************************************************************
;-----------------------------------------------------------------------------------------------------------------------------------
; Generate I2C bus start condition              [ I2C STATE -> 0 ]
WrtStart
                    banksel   ptr                         ; select GPR bank
                    movf      ptr,w                       ; retrieve ptr address
                    movwf     FSR                         ; initialize FSR for indirect access
                    movfw     datacount
                    movwf     tcount                      ; restore tcount
                    incf      i2c_status,f                ; update I2C state variable
                    banksel   SSPCON2                     ; select SFR bank
                    bsf       SSPCON2,SEN                 ; initiate I2C bus start condition
                    return

; Generate I2C address write (R/W=0)           [ I2C STATE -> 1 ]
SendWrtAddr
                    banksel address                       ; select GPR bank
                    bcf       STATUS,C                    ; ensure carry bit is clear
                    movfw     address                     ; compose 7-bit address
                    incf      i2c_status,f                ; update I2C state variable
                    banksel   SSPBUF                      ; select SFR bank
                    movwf     SSPBUF                      ; initiate I2C bus write condition
                    return                                ;

; Test acknowledge after address and data write [ I2C STATE -> 2 ]
WrtAckTest
                    banksel   SSPCON2                     ; select SFR bank
                    btfss     SSPCON2,ACKSTAT             ; test for acknowledge from slave
                    goto      WrtData                     ; go to write data module
                    banksel   flags                       ; select GPR bank
                    bsf       flags,ack_error             ; set acknowledge error
                    movlw     d'4'                        ; goto "WrtDone" with next int
                    movwf     i2c_status                  ;
                    banksel   SSPCON2                     ; select SFR bank
                    bsf        SSPCON2,PEN                ; initiate I2C bus stop condition
                    return

; Generate I2C write data condition
WrtData
                    movfw     INDF                        ; fetch data
                    banksel   tcount                      ; select GPR bank
                    decfsz    tcount,f                    ; test if all done with writes
                    goto      send_byte                   ; not end of string
                    incf      i2c_status,f                ; update I2C state variable
send_byte
                    banksel   SSPBUF                      ; select SFR bank
                    movwf     SSPBUF                      ; initiate I2C bus write condition
                    incf      FSR,f                       ; increment pointer
                    return


; Generate I2C bus stop condition              [ I2C STATE -> 3 ]
WrtStop
                    banksel   SSPCON2                     ; select SFR bank
                    btfss     SSPCON2,ACKSTAT             ; test for acknowledge from slave
                    goto      no_error                    ; bypass setting error flag
```

```
                banksel   flags                        ; select GPR bank
                bsf       flags,ack_error              ; set acknowledge error
                clrf      i2c_status                   ; reset I2C state variable
                goto      stop
no_error
                banksel   i2c_status                   ; select GPR bank
                incf      i2c_status                   ; prepare next step

stop
                banksel   SSPCON2                      ; select SFR bank
                bsf       SSPCON2,PEN                  ; initiate I2C bus stop condition
                return                                                    ;
WrtRestart_5842
                banksel   SSPCON2                      ; select SFR bank
                btfss     SSPCON2,ACKSTAT              ; test for acknowledge from slave
                goto      no_error_5842                ; bypass setting error flag
                banksel   flags                        ; select GPR bank
                bsf       flags,ack_error              ; set acknowledge error
                clrf      i2c_status                   ; reset I2C state variable
                goto      stop
no_error_5842
                banksel   i2c_status                   ; select GPR bank
                incf      i2c_status                   ; prepare next step

ReStart_5842
                banksel   SSPCON2                      ; select SFR bank
                bsf       SSPCON2,RSEN
                return

WrtDone
                btfss     flags,bus_coll               ; if (no bus_coll occurred)
                bsf       flags,done                           ; set flag done
                btfsc     flags,ack_error              ; else check ack_error
                bcf       flags,done                   ; there was one!
                bsf       STATUS,RP0
                clrf      TRISD
                bcf       STATUS,RP0
                clrf      command
                clrf      i2c_status                   ; to main loop
                bcf       flags,busy
                btfsc     PORTE,TXE                    ; see if module ready
                goto      $-1
                bsf       PORTB,WR                     ; start writing
                movfw     flags
                movwf     PORTD
                bcf       PORTB,WR                     ; ACTUALLY WRITE
                clrf      flags
                bsf       STATUS,RP0                   ; do cleanup & return
                movlw     0xff
                movwf     TRISD
                bcf       STATUS,RP0
                movfw     ptr
                movwf     FSR
                return

;-----------------------------------------------------------------------------------------------------------------------------------
;********************Read data from Slave*********************************************************************************
;-----------------------------------------------------------------------------------------------------------------------------------
-------------------------------------------------------------
; Generate I2C start condition                 [ I2C STATE -> 5 ]
ReadStart
                banksel   ptr                          ; select GPR bank
                movf      ptr,W                        ; retrieve ptr address
                movwf     FSR                          ; initialize FSR for indirect access
                movfw     datacount                    ; restore tcount
                movwf     tcount
                incf      i2c_status,f                 ; update I2C state variable
                banksel   SSPCON2                      ; select SFR bank
                bsf       SSPCON2,SEN                  ; initiate I2C bus start condition
                return

; Test acknowledge and generate I2C restart condition
Restart
                banksel   SSPCON2                      ; select SFR bank
                btfss     SSPCON2,ACKSTAT              ; test for acknowledge from slave
                goto      no_error_restart             ; bypass setting error flag
                banksel   flags                        ; select GPR bank
                bsf       flags,ack_error              ; set acknowledge error
                clrf      i2c_status                   ; reset I2C state variable
                goto      stop


no_error_restart
                incf      i2c_status,f                 ; update I2C state variable
                clrf      i2c_status                   ; reset I2C state variable
                banksel   SSPCON2                      ; select SFR bank
                bsf       SSPCON2,RSEN                 ; initiate I2C bus restart condition
                return

; Generate I2C address write (R/W=1)           [ I2C STATE -> 6 ]
SendReadAddr
                banksel address                        ; select GPR bank
                bsf       STATUS,C                     ; ensure carry bit is clear
                movfw     address                      ; compose 7 bit address
                incf      i2c_status,f                 ; update I2C state variable
                banksel   SSPBUF                       ; select SFR bank
                movwf     SSPBUF                       ; initiate I2C bus write condition
                return                                 ;

; Test acknowledge after address write         [ I2C STATE -> 7 ]
ReadAckTest
                banksel   SSPCON2                      ; select SFR bank
                btfss     SSPCON2,ACKSTAT              ; test for not acknowledge from slave
                goto      StartReadData                ; good ack, go issue bus read
                banksel   flags                        ; ack error, so select GPR bank
```

```
                    bsf       flags,ack_error              ; set ack error flag
                    movlw     d'10'                        ; goto ReadDone with next int
                    movwf     i2c_status                   ; error handling
                    banksel   SSPCON2                      ; select SFR bank
                    bsf       SSPCON2,PEN                  ; initiate I2C bus stop condition
                    return

StartReadData
                    bsf       SSPCON2,RCEN                 ; generate receive condition
                    banksel i2c_status                     ; select GPR bank
                    incf      i2c_status,f                 ; update I2C state variable
                    return

; Read slave I2C                            [ I2C STATE -> 8 ]
ReadData

                    banksel   SSPBUF                       ; select SFR bank
                    movf      SSPBUF,w                     ; save off byte into W
                    banksel   tcount                       ; select GPR bank
                    decfsz    tcount,f                     ; test if all done with reads
                    goto      SendReadAck                  ; not end of string so send ACK


; Send Not Acknowledge
SendReadNack
                    movwf     INDF                         ; save off null character
                    incf      i2c_status,f                 ; update I2C state variable
                    banksel   SSPCON2                      ; select SFR bank
                    bsf       SSPCON2,ACKDT                ; acknowledge bit state to send (not ack)
                    bsf       SSPCON2,ACKEN                ; initiate acknowledge sequence
                    return

; Send Acknowledge
SendReadAck
                    movwf     INDF                         ; no, save off byte
                    incf      FSR,f                        ; update receive pointer
                    banksel   SSPCON2                      ; select SFR bank
                    bcf       SSPCON2,ACKDT                ; acknowledge bit state to send
                    bsf       SSPCON2,ACKEN                ; initiate acknowledge sequence
                    btfsc     SSPCON2,ACKEN                ; ack cycle complete?
                    goto      $-1                          ; no, so loop again
                    bsf       SSPCON2,RCEN                 ; generate receive condition
                    btfsc     SSPCON2,RCEN                 ; check RCEN cycle complete
                    goto      $-1                          ; no, so loop again
                    return                                 ;


; Generate I2C stop condition               [ I2C STATE -> 8 ]
ReadStop
                    banksel   SSPCON2                      ; select SFR bank
                    bsf       SSPCON2,PEN                  ; initiate I2C bus stop condition
                    banksel   i2c_status                   ; select GPR bank
                    movlw     d'12'
                    incf      i2c_status
                    return

ReadDone
                    btfss     flags,bus_coll               ; if (no bus_coll occurred)
                    bsf       flags,done                   ; set flag done
                    btfsc     flags,ack_error              ; else check ack_error
                    bcf       flags,done                   ; there was one!
                    bsf       STATUS,RP0                   ; else don't set it
                    clrf      TRISD
                    bcf       STATUS,RP0
                    movfw     ptr                          ; get beginning of data
                    movwf     FSR                          ; init pointer
                    movfw     count
                    movwf     tcount                       ; restore tcount
checkbyte
                    decfsz    tcount
                    goto      send2usb
                    bcf       flags,busy
                    btfsc     PORTE,TXE                    ; write flags to usb
                    goto      $-1
                    bsf       PORTB,WR                     ; start writing
                    movfw     flags
                    movwf     PORTD
                    bcf       PORTB,WR                     ; ACTUALLY WRITE
                    clrf      command
                    clrf      i2c_status                   ; to main loop
                    bsf       STATUS,RP0
                    movlw     0xff                         ;do cleanup & return
                    movwf     TRISD
                    bcf       STATUS,RP0
                    movfw     ptr
                    movwf     FSR
                    clrf      flags
                    return

send2usb
                    btfsc     PORTE,TXE                    ; see if module ready
                    goto      $-1
                    bsf       PORTB,WR                     ; start writing
                    movfw     INDF
                    movwf     PORTD
                    bcf       PORTB,WR                     ; ACTUALLY WRITE
                    incf      FSR
                    goto      checkbyte                    ; see if one more byte

;=======================End I2Cservice===============================================================================
service_Trs232                                             ; Transmit interrupt service
                    bsf       STATUS,RP0
                    btfss     PIE1,TXIE                    ; if not enabled
                    return                                 ; then return
                    bcf       STATUS,RP0                   ; select bank0
```

```
                    bcf       PIR1,TXIF                       ; clear interrupt flag
                    decfsz    tcount                          ; if (!all sent)
                    goto      tx_232                          ; send next
                    bsf       flags,done                      ; else say done & send last
tx_232

          IFDEF     handshake
hs_again
                    bsf       PORTC,RTS                       ; assert rts
                    btfsc     PORTC,CTS
                    goto      hs_again                        ; hangs if no cts!!!!
          ENDIF
                    btfsc     PORTE,RXF                       ; if (data not ready)
                    goto      $-1                             ; then test again
                    bcf       PORTB,READ                      ; read number of databytes
                    movfw     PORTD                           ; fetch data
                    movwf     TXREG                           ; send byte
                    bsf       PORTB,READ                      ; prepare next read
                    btfss     flags,done                      ; if !done
                    return                                    ; return and wait for next byte
                                                              ; else send statusbyte done
                    bsf       STATUS,RP0                      ; select bank1
                    bcf       PIE1,TXIE                       ; clear tx-int enable
                    bcf       STATUS,RP0                      ; select bank0
                    bcf       flags,busy                                ; no more busy
                    bsf       STATUS,RP0                      ; select bank1
                    clrf      TRISD                           ; portd is output
                    bcf       STATUS,RP0                      ; select bank0
                    btfsc     PORTE,TXE                       ; see if module ready
                    goto      $-1
                    bsf       PORTB,WR                        ; start writing
                    movfw     flags                           ;
                    movwf     PORTD
                    bcf       PORTB,WR                        ; ACTUALLY WRITE
                    bcf       flags,done
                    bsf       STATUS,RP0                      ; select bank1
                    decf      TRISD                           ; 0x00->0xff
                    bcf       STATUS,RP0                      ; select bank0

          IFDEF     handshake
                    bcf       PORTC,RTS                       ; clear rts
          ENDIF
                    return                                    ; done

service_Rrs232                                                ; Receive interrupt service
                    bcf       STATUS,RP0                      ; select bank1
                    btfss     PORTE,RXF                       ; if byte in fifo
                    return                                    ; then return! (rxf=low active)

                    btfsc     flags,busy                      ; if busy
                    return                                    ; then return

                    btfsc     RCSTA,OERR                      ; if overrun
                    goto      handle_or                       ; handle overrun

                    btfsc     RCSTA,FERR                      ; if framing error
                    goto      byte_void                       ; this byte is void
send_Rrs232
                    bsf       STATUS,RP0                      ; select bank1
                    clrf      TRISD                           ; portd is output
                    bcf       STATUS,RP0                      ; select bank0
                    btfsc     PORTE,TXE                       ; see if module ready
                    goto      $-1
                    bsf       PORTB,WR                        ; start writing
                    movfw     RCREG                           ; get Byte
                    movwf     PORTD                           ; send to usb
                    bcf       PORTB,WR                        ; write
                    btfsc     PIR1,RCIF                       ; if (rc not empty)
                    goto      send_Rrs232                     ; send next byte
                    bsf       STATUS,RP0                      ; select bank1
                    decf      TRISD                           ; 0x00->0xff
                    bcf       STATUS,RP0                      ; select bank0
                    return                                    ; else done
handle_or
                    bcf       RCSTA,CREN
                    bsf       RCSTA,CREN
byte_void
                    movfw     RCREG                           ; (clear rcflag)
                    movfw     RCREG
                    return                                    ; then return RXIF not cleared->what happens???


INIT_UART
; make sure pins are setup before calling this routine
; TRISC:6 and TRISC:7 must be set ( as for input, but operates as input/output )
; furthermore its advised that interrupts are disabled during this routine
                    bcf       INTCON,GIE                      ; disable interrupts
                    bsf       STATUS,RP0                      ; select bank1
                    movlw     baud                            ; fetch baudradt
                    movwf     SPBRG                           ; set baudrate
                    ;bcf      STATUS,RP0                      ; select bank0

; enable transmitter
                    IFDEF     high_br
                              movlw     (1<<TXEN) |(1<<BRGH); preset enable transmitter and high speed mode
                              MOVWF     TXSTA                 ; and set it
                    ENDIF
                    IFDEF     low_br
                              movlw     (1<<TXEN)             ; preset enable transmitter and high speed mode
                              MOVWF     TXSTA                 ; and set it
                    ENDIF
                    bcf       STATUS,RP0                      ; select bank0

; enable recevier
                    movlw     (1<<SPEN)|(1<<CREN)             ; preset serial port enable and continuous receive
```

```
                    movwf     RCSTA                         ; set it
                    movfw     RCREG
                    movfw     RCREG                         ; dummy read (needed)

; enable reciever interrupt
                    bsf       STATUS,RP0                    ; select bank1
                    bsf       PIE1,RCIE                     ; enable receiver irq
                    bsf       INTCON,PEIE                   ; and peripheral irq must also be enabled
                    bsf       INTCON,GIE                    ; re-enable interrupts
                    bcf       STATUS,RP0                    ; select bank0
                    return

;==============================================================================================================
;======================End of Program ===========================================================================
;==============================================================================================================

                    END                           ; directive 'end of program'
```

The driver for Windows computers ships with two variants in a combined driver model (CDM). Included are the D2XX driver for direct communication with the PIC and a Virtual COM Port (VCP) driver for communication via a COM port which become available on the host PC when the VCP driver is enabled. Only the D2XX driver is enabled by default. Below is code for both the VCP and D2XX drivers.

The .h file used with the PIC-specific C-code for the VCP driver is shown below:

```
////////////////////////////////////////////////////////////////////////////
//Export functions for usbdetectors.c
//Module of GCI MultiPhotonTimeResolved system
//
//Rosalind Locke - January 2003
////////////////////////////////////////////////////////////////////////////
// RJL 23 December 2003 - Added function GCI_EnableLowLevelErrorReporting()
////////////////////////////////////////////////////////////////////////////
typedef unsigned char byte;
int GCI_initI2C(void);
void GCI_EnableLowLevelErrorReporting(int enable);
void GCI_closeI2C(void);
int GCI_writeRS232(byte data[],int nbr_bytes);
int GCI_readRS232(byte data[]);
int GCI_writeI2C(int nbrbytes, byte data[],int i2cbus);
int GCI_readI2C(int nbrbytes, byte data[],int i2cbus);
int GCI_writeFAST(byte data[],int nbr_bytes);
int GCI_readFAST(byte data[],int nbr_bytes);
int GCI_writereadFAST(byte data[],int nbr_bytes,int ret_bytes);
```

The PIC-specific C-code for the VCP driver is shown below. Some parts are used only when multiple $I^2C$ bus systems are used, for example MAX4562 analogues switches can be used to increase the number of $I^2C$ busses when address limits of bus-controlled parts have been reached.

```
#ifndef _USBCONVERTER_AM
#define _USBCONVERTER_AM

#include <userint.h>
#include <rs232.h>
#include <stdio.h>
#include <utility.h>
#include <formatio.h>
#include "usbconverter_v2.h"

///////////////////////////////////////////////////////////////////////////////
// Routines for FTDI USB -> parallel/serial devices
// Andreas Manser 2002
///////////////////////////////////////////////////////////////////////////////
// RJL 18 December 2003
// This is the most recent version as used in the GCI MP system
//
// Changes from A Manser's original
// 1. Commented out error messages in usberr() function as errors should be reported at a higher level
// 2. Made it conform to software guidlines, i.e. export functions are named GCI_
//
///////////////////////////////////////////////////////////////////////////////
// RJL 23 December 2003
// Added function GCI_EnableLowLevelErrorReporting() as error messages may be required
// during initial hardware tests. By default it is disabled.
///////////////////////////////////////////////////////////////////////////////
// RJL 14 April 2004
// Integrated changes from Rob's version of the program
// Note only PICs with Rob's new firmware return data when using the FAST functions
///////////////////////////////////////////////////////////////////////////////
// RJL July 2004
// Add functions for multiple port operation.
// The single port functions remain for backward compatibility.
// The multi port functions should be used for new projects.
///////////////////////////////////////////////////////////////////////////////

//#define I2C_STANDALONE_EXAMPLE
```

```c
typedef unsigned char byte;
static int mi2cbus = GCI_I2C_SINGLE_BUS;        // No switching of the I2C bus

//Try to cope with all the different ways we've defined the port number in the past
#ifdef PORT
static int gSerialPort = PORT;                  // for single port operation
#else
    #ifdef I2CPORT
    static int gSerialPort = I2CPORT;           // for single port operation
    #else
    static int gSerialPort = 3;                 // for single port operation
    #endif
#endif

static int reportErrors = 0;
//Will we ever get higher than COM10 ?
static int gPortOpen[10] = {0,0,0,0,0,0,0,0,0,0};

////////////////////////////////////////////////////////////////////////
static int usberr(int stat)
{
char msg[256];
    switch (stat){
        case 4:     return 0;  //OK
                        case 5:     return 0;  //OK
                    case 2:     sprintf(msg, "I2C-ACK-ERROR");
                    if (reportErrors) MessagePopup("USB-Converter: ",msg);
                    return -2;
                    case 8:     sprintf(msg, "OOPS I2C-BUS-Collision=>we're in trouble");
                    if (reportErrors) MessagePopup("USB-Converter: ",msg);
                    return -3;

        case 1:     sprintf(msg, "I2C-Interrupt");
                    if (reportErrors) MessagePopup("USB-Converter: ",msg);
                    return -4;

        case -99:   sprintf(msg, "Cannot access I2C controller. Is it switched on?");
                    if (reportErrors) MessagePopup("USB-Converter: ",msg);
                    return -99;

        default:    sprintf(msg, "Unknown I2C error code %i",stat);
                    if (reportErrors) MessagePopup("USB-Converter: ",msg);
                    break;
    }
    return -1;
}

static int selectBus(int port, int i2cbus)
{
    unsigned char senddata[10];

    //convert_i2cbus to uchar[]  for selecting it
    senddata[0]=0x02;   //i2c
    senddata[1]=0x04;   //nbr_bytes
    senddata[2]=0x68;   //address
    senddata[3]=0xc0;   //command"switchset"
    switch (i2cbus){
        case 8: senddata[4]=0x00;   //all switches open
                senddata[5]=0x00;   //that's it
                break;
        case 7: senddata[4]=0x30;   //sw8 sw5
                senddata[5]=0x00;   //that's it
                break;
        case 6: senddata[4]=0x01;   //sw6a
                senddata[5]=0x20;   //sw3b
                break;
        case 5: senddata[4]=0x02;   //sw6b
                senddata[5]=0x10;   //sw3a
                break;
        case 4: senddata[4]=0x04;   //sw7a
                senddata[5]=0x08;   //sw2b
                break;
        case 3: senddata[4]=0x08;   //sw7b
                senddata[5]=0x04;   //sw2a
                break;
        case 2: senddata[4]=0x00;   //none
                senddata[5]=0x82;   //sw4b sw1b
                break;
        case 1: senddata[4]=0x00;   //none
                senddata[5]=0x41;   //sw4a sw1a
                break;
    }
    ComWrt(port, senddata, 6);   //send it
    if (usberr(ComRdByte(port))) return -1;;

    mi2cbus = i2cbus;
    Delay(0.1);                                 //ensures that occasional ack err do not occur
    return 0;
}

static int write_usbconverter(int port, byte command, int nbr_bytes, byte data[], int i2cbus)
{
    int i, stat;
    unsigned char senddata[255];

    while (GetInQLen (port)) {                  //dump bytes which shoudn't be there
        if (rs232err < 0) return -1;
        ComRdByte(port);
    }

    switch (command){
        case 0x03:                              //RS232
            i2cbus = mi2cbus;                   //don't switch the i2c bus
                                                //don't break!
```

```
        case 0x02:                              //I2C
            if (i2cbus != mi2cbus){             //switching of i2cbus needed
                if (selectBus(port, i2cbus) < 0) return -1;
            }
            senddata[0] = command;
            senddata[1] = nbr_bytes;
            for (i=0; i<nbr_bytes; i++)
                senddata[i+2] = data[i];
            if (ComWrt(port, senddata, nbr_bytes+2) < 0) return -1;
            stat = ComRdByte (port);
            break;
        case 0x01:                                //FAST write
            senddata[0] = command;
            senddata[1] = nbr_bytes;
            for (i=0; i<nbr_bytes; i++)
                senddata[i+2] = data[i];
            if (ComWrt(port, senddata, nbr_bytes+2) < 0) return -1;
            //older PICs don't return anything, so if nothing returned assume success
            if (GetInQLen(port) < 1)
                stat = 4;
            else {
                ComRd(port, data, 1);             //just read one byte
                if (data[0] != 0x04)              //returns 0x04 for I2CFAST PIC
                    stat = 2;
                else
                    stat = 4;
            }
            break;
        case 0x04:                              //FAST read and write
            senddata[0] = 0x01;                 //0x01 for fast command
            senddata[1] = nbr_bytes;
            for (i=0; i<nbr_bytes; i++)
                senddata[i+2] = data[i];
            if (ComWrt(port, senddata, nbr_bytes+2) < 0) return -1;

            stat = GetInQLen(port);
            if (stat>0) ComRd(port, data, i2cbus);  //just read using i2c bus integer
            stat = 4;                               //as number of bytes returned
            break;
    }
    return usberr(stat);//stat;
}


static int read_usbconverter(int port, byte command, int nbr_bytes, byte  data[], int i2cbus)
{
    int i, stat;
    unsigned char senddata[128];

    switch (command) {
        case 0x02:                              //I2C
            while(GetInQLen (port)) {           //dump bytes which should not be there
                if (rs232err < 0) return -1;
                ComRdByte(port);
            }

            if (i2cbus != mi2cbus){             //switching of i2cbus needed
                if (selectBus(port, i2cbus) < 0) return -1;
            }
            senddata[0] = command;
            senddata[1] = nbr_bytes+1;
            for (i=0; i<nbr_bytes; i++)
                senddata[i+2] = data[i];
            if (ComWrt(port, senddata, 3) < 0) return -1;   //3 bytes for command, nbr_bytes and address (valid only for i2c)

            for (i=0; i<nbr_bytes; i++)
                data[i] = 0x00;
            if (ComRd(port, data, nbr_bytes+1) < 0) return -1;  //data + 1 for status, was used for address while sending
            stat = data[nbr_bytes];
            break;
        case 0x01:                                        //FAST
            senddata[0] = command;
            nbr_bytes = nbr_bytes+1;                      //number of bytes + 1 for RW command
            senddata[1] = nbr_bytes+1;
            for (i=0; i<nbr_bytes; i++)
                data[i] = 0x00;

            for (i=0; i<nbr_bytes; i++)
                senddata[i+2] = data[i];
            if (ComWrt(port, senddata, nbr_bytes+2) < 0) return -1;
            stat = GetInQLen(port);

            ComRd(port, data, nbr_bytes-1);               //just read correct number of bytes
            stat = 4;                                     //no status byte returned
            break;

        default:                                          //RS232
            stat = GetInQLen(port);
            if (stat == 0) break;
            ComRd(port, data, stat);                      //just read
            stat = 4;                                     //no status byte returned
            break;

        }
    return usberr(stat);//stat;
}

////////////////////////////////////////////////////////////////////////////////

void GCI_EnableLowLevelErrorReporting(int enable)
{
    reportErrors = enable;
}

////////////////////////////////////////////////////////////////////////////////
```

```
// Legacy export functions for single port operation

void GCI_setI2Cport(int port)
{
    gSerialPort = port;
}

int GCI_initI2C()
{
    char port_string[10];

    if (gPortOpen[gSerialPort]) return 0;                           //already initialised

    sprintf(port_string, "COM%d", gSerialPort);
    if (OpenComConfig (gSerialPort, port_string, 9600, 0, 8, 1, 164, 164) < 0) {
        MessagePopup("i2c Error", "Is control box switched on?");
        return -1;
    }

    SetXMode (gSerialPort, 0);
    SetCTSMode (gSerialPort, LWRS_HWHANDSHAKE_OFF);
    SetComTime (gSerialPort, 3);                                    //timeout = 3sec
    Delay(0.1);

    while(GetInQLen (gSerialPort)) {                                //dump bytes which shoudn't be there
        if (rs232err < 0) return -1;
        ComRdByte(gSerialPort);
    }

    gPortOpen[gSerialPort] = 1;
    return 0;
}

void GCI_closeI2C()
{
    CloseCom(gSerialPort);
    gPortOpen[gSerialPort] = 0;
}

int GCI_writeRS232(byte data[], int nbr_bytes)
{
    return write_usbconverter(gSerialPort, 0x03, nbr_bytes, data, 0);
}

int GCI_readRS232(byte data[])
{
    return  read_usbconverter(gSerialPort, 0x03, 0x00, data, 0);
}

int GCI_readI2C(int nbrbytes, byte data[], int i2cbus)
{
    int i;

    i = read_usbconverter(gSerialPort, 0x2, nbrbytes, data, i2cbus);

    return i;
}

int GCI_writeI2C(int nbrbytes, byte data[], int i2cbus)
{

    return  write_usbconverter(gSerialPort, 0x2, nbrbytes, data, i2cbus);
}

int GCI_writeFAST(byte data[], int nbr_bytes)
{
    return  write_usbconverter(gSerialPort, 0x01, nbr_bytes, data, 0);
}

int GCI_readFAST(byte data[], int nbr_bytes)
{
    return  read_usbconverter(gSerialPort, 0x01, nbr_bytes, data, 0);
}

int GCI_writereadFAST(byte data[], int nbr_bytes, int ret_bytes)
{
    return  write_usbconverter(gSerialPort, 0x04, nbr_bytes, data, ret_bytes);
}

////////////////////////////////////////////////////////////////////////////////
// Multi-port operation

int GCI_initI2C_multiPort(int port)
{
    char port_string[10];

    if (port > 10) {
        MessagePopup("USB Error", "Sorry, too many com ports");
        return -1;
    }

    if (gPortOpen[port]) return 0;                                  //already initialised

    sprintf(port_string, "COM%d", port);
    if (OpenComConfig (port, port_string, 9600, 0, 8, 1, 164, 164) < 0) {
        MessagePopup("i2c Error", "Is control box switched on?");
        return -1;
    }

    SetXMode (port, 0);
    SetCTSMode (port, LWRS_HWHANDSHAKE_OFF);
    SetComTime (port, 3);                                           //timeout = 3 sec
    Delay(0.1);

    while(GetInQLen (port)) {                                       //dump bytes which should not be there
```

```
        if (rs232err < 0) return -1;
        ComRdByte(port);
    }

    gPortOpen[port] = 1;

    return 0;
}

void GCI_closeI2C_multiPort(int port)
{
    CloseCom(port);
    gPortOpen[port] = 0;
}

int GCI_writeRS232_multiPort(int port, byte data[], int nbr_bytes)
{
    return write_usbconverter(port, 0x03, nbr_bytes, data, 0);
}

int GCI_readRS232_multiPort(int port, byte data[])
{
    return  read_usbconverter(port, 0x03, 0x00, data, 0);
}

int GCI_readI2C_multiPort(int port, int nbrbytes, byte data[], int i2cbus)
{
    int i;

    i = read_usbconverter(port, 0x2, nbrbytes, data, i2cbus);
    return i;
}

int GCI_writeI2C_multiPort(int port, int nbrbytes, byte data[], int i2cbus)
{
    return  write_usbconverter(port, 0x2, nbrbytes, data, i2cbus);
}

int GCI_writeFAST_multiPort(int port, byte data[], int nbr_bytes)
{
    return  write_usbconverter(port, 0x01, nbr_bytes, data, 0);
}

int GCI_readFAST_multiPort(int port, byte data[], int nbr_bytes)
{
    return  read_usbconverter(port, 0x01, nbr_bytes, data, 0);
}

int GCI_writereadFAST_multiPort(int port, byte data[], int nbr_bytes, int ret_bytes)
{
    return  write_usbconverter(port, 0x04, nbr_bytes, data, ret_bytes);
}

////////////////////////////////////////////////////////////////////////////////

#ifdef  I2C_STANDALONE_EXAMPLE

int main (int argc, char *argv[])
{
    byte data[255]={0x40,0x55};

    GCI_initI2C();

    GCI_writeI2C(2, data,6);
    GCI_readI2C(1, data,4);                                   //possible (but hardware has to be there!)
    printf("read %s\n",data);

    while (!KeyHit()){}
    CloseCom (gSerialPort);

    return 0;
}
#endif
#endif /* _USBCONVERTER_AM */
```

The following code uses the D2XX driver and is multi-thread safe through the use of thread locks as only one device should be communicated with at one time.

```
/*
    Communications interface using the D2XX driver for FTDI

    Filename: FTDI_Utils.h
    Date:     2010

    Author: Glenn Pierce
    Company: Gray Institute

*/

#ifndef _FTDI_UTILS_
#define _FTDI_UTILS_

#include <windows.h>
#include "FTD2XX.H"

#define FTDI_MAX_NUMBER_OF_DEVICES 10
#define FTDI_SERIAL_NUMBER_LINE_LENGTH 64
#define FTDI_DESCRIPTION_LINE_LENGTH 256

#define SETBITS(mem, bits)      (mem) |= (bits)
#define CLEARBITS(mem, bits)    (mem) &= ~(bits)
```

```c
#define TOGGLEBITS(mem)          (~(mem))
#define SETLSB(mem)              SETBITS(mem, BIN(0,0,0,0,0,0,0,1));
#define CLEARLSB(mem)            CLEARBITS(mem, BIN(0,0,0,0,0,0,0,1));
#define BIN(b7,b6,b5,b4, b3,b2,b1,b0)                          \
(BYTE)(                                                 \
    ((b7)<<7) + ((b6)<<6) + ((b5)<<5) + ((b4)<<4) +         \
    ((b3)<<3) + ((b2)<<2) + ((b1)<<1) + ((b0)<<0)           \
)

#define SET_BIT_ON(mem, bit) (mem |= 1 << bit)
#define SET_BIT_OFF(mem, bit) (mem &= ~(1 << bit))
#define TOGGLE_BIT(mem, bit) (mem ^= 1 << bit)

#define SET_BIT_WITH_MASK_TO_VALUE(mem, bit, value) (mem ^= (-value ^ mem) & bit)
#define SET_BIT_TO_VALUE(mem, bit, value) ( SET_BIT_WITH_MASK_TO_VALUE(mem, (1 << bit), value) )

#define GET_BIT_FROM_VALUE(value, bit) (value & (1 << bit))

//#define SET_BIT_TO_VALUE(mem, bit_mask, value) (if (value) mem |= bit_mask; else mem &= ~bit_mask;)

// Example int a = 0x123;
// SETBITS(a, BIN(0,0,0,1,1,1,1,0));
// printf("0x%x", a); // should be 0x13F

#define MSB(a) (a = (a | (1<<(sizeof(a)*8-1)) ) | (a))

typedef struct _FTDI FTDIController;

typedef FT_STATUS (WINAPI *PtrToOpen)(PVOID, FT_HANDLE *);
typedef FT_STATUS (WINAPI *PtrToOpenEx)(PVOID, DWORD, FT_HANDLE *);
typedef FT_STATUS (WINAPI *PtrToListDevices)(PVOID, PVOID, DWORD);
typedef FT_STATUS (WINAPI *PtrToClose)(FT_HANDLE);
typedef FT_STATUS (WINAPI *PtrToRead)(FT_HANDLE, LPVOID, DWORD, LPDWORD);
typedef FT_STATUS (WINAPI *PtrToWrite)(FT_HANDLE, LPVOID, DWORD, LPDWORD);
typedef FT_STATUS (WINAPI *PtrToResetDevice)(FT_HANDLE);
typedef FT_STATUS (WINAPI *PtrToPurge)(FT_HANDLE, ULONG);
typedef FT_STATUS (WINAPI *PtrToSetTimeouts)(FT_HANDLE, ULONG, ULONG);
typedef FT_STATUS (WINAPI *PtrToGetQueueStatus)(FT_HANDLE, LPDWORD);
typedef FT_STATUS (WINAPI *PtrToSetBaudRate)(FT_HANDLE, ULONG);
typedef FT_STATUS (WINAPI *PtrToSetBitMode)(FT_HANDLE, UCHAR, UCHAR);
typedef FT_STATUS (WINAPI *PtrToSetDataCharacteristics)(FT_HANDLE, UCHAR, UCHAR, UCHAR);
typedef FT_STATUS (WINAPI *PtrToSetDeadmanTimeout)(FT_HANDLE, DWORD);

typedef void (*FT_ERROR_HANDLER) (FTDIController *, const char *title, const char *error_string, void *callback_data);

typedef struct _DLLPointerTable
{
    PtrToOpen pOpen;
    PtrToOpenEx pOpenEx;
    PtrToWrite pRead;
    PtrToWrite pWrite;
    PtrToListDevices pListDevices;
    PtrToClose pClose;
    PtrToResetDevice pResetDevice;
    PtrToPurge pPurge;
    PtrToSetTimeouts pSetTimeouts;
    PtrToGetQueueStatus pGetQueueStatus;
    PtrToSetBaudRate pSetBaudRate;
    PtrToSetBitMode pSetBitMode;
    PtrToSetDataCharacteristics pSetDataCharacteristics;
    PtrToSetDeadmanTimeout pSetDeadmanTimeout;

} DLLPointerTable;

typedef struct _FTDI
{
    HMODULE module;
    FT_HANDLE device_handle;
    int debug;
    int debug_with_ints;

    DLLPointerTable dllPointerTable;

    void *callback_data;
    FT_ERROR_HANDLER error_handler;

    char device_id[FTDI_SERIAL_NUMBER_LINE_LENGTH];
};

FTDIController* ftdi_controller_new(void);
void            ftdi_controller_destroy(FTDIController* controller);
void            ftdi_controller_set_error_handler(FTDIController* controller, FT_ERROR_HANDLER handler, void *callback_data);
int             ftdi_controller_get_lock(FTDIController* controller);
int             ftdi_controller_release_lock(FTDIController* controller);
FT_STATUS       ftdi_controller_open(FTDIController* controller, const char *device_id);
FT_STATUS       ftdi_controller_close(FTDIController* controller);
FT_STATUS       ftdi_controller_purge_read_queue(FTDIController* controller);
FT_STATUS       ftdi_controller_purge_write_queue(FTDIController* controller);

FT_STATUS       ftdi_controller_set_deadman_timout(FTDIController* controller, DWORD timeout);
FT_STATUS       ftdi_controller_set_timouts(FTDIController* controller, DWORD readTimeout , DWORD writeTimeout);
FT_STATUS       ftdi_controller_set_baudrate(FTDIController* controller, unsigned long baudrate);
FT_STATUS       ftdi_controller_set_data_characteristics(FTDIController* controller, unsigned char word_length
    , unsigned char stop_bits, unsigned char parity);

FT_STATUS       ftdi_controller_set_bit_bang_mode(FTDIController* controller, unsigned char mask, unsigned char ucMode);
int             ftdi_controller_set_debugging(FTDIController* controller, int debug);
int             ftdi_controller_show_debugging_bytes_as_integers(FTDIController* controller);
int             ftdi_controller_show_debugging_bytes_as_hex(FTDIController* controller);
int             ftdi_controller_get_number_of_connected_devices(FTDIController* controller);
int             ftdi_controller_print_serial_numbers_of_connected_devices(FTDIController* controller);
FT_STATUS       ftdi_controller_add_serial_numbers_of_connected_devices_to_ring_control(FTDIController* controller, int panel, int
ctrl);
FT_STATUS       ftdi_controller_get_read_queue_status(FTDIController* controller, LPDWORD amount_in_rx_queue);
void            ftdi_print_byte_array_to_stdout(BYTE* array, int array_length, const char* prefix, int use_ints);
```

```
/* The address passed to these functions are the 8bit address. The original protocol spec for out i2c devices
says that this is shifted one to the left ie make a 7bit address. The LSB is then used to indicate writing
or reading. Rob when passing the address did the shifting at the wrong higher level code so he often gave
me the 7bit equivilent address. */

FT_STATUS   ftdi_controller_i2c_write_bytes(FTDIController* controller, int address, int data_length, BYTE *data);
FT_STATUS   ftdi_controller_i2c_read_bytes(FTDIController* controller, int address, int data_length, BYTE *data);

FT_STATUS   ftdi_controller_i2c_fastline_write_bytes(FTDIController* controller, BYTE *data, int data_length);
FT_STATUS   ftdi_controller_i2c_fastline_read_bytes(FTDIController* controller, int data_length, BYTE *data);

FT_STATUS   ftdi_controller_write_bytes(FTDIController* controller, int data_length, BYTE *data);
FT_STATUS   ftdi_controller_write_byte(FTDIController* controller, BYTE data);
FT_STATUS   ftdi_controller_write_string(FTDIController* controller, const char* fmt, ...);
FT_STATUS   ftdi_controller_read_bytes(FTDIController* controller, int data_length, BYTE *data);
FT_STATUS   ftdi_controller_read_bytes_availiable_in_rx_queue(FTDIController* controller, BYTE *data, LPDWORD bytes_read);

#endif

/* Communications interface using the D2XX driver for FTDI

    Filename: FTDI_Utils.c
    Date:     2010
    Author: Glenn Pierce
    Company: Gray Institute
*/

#include "FTDI_Utils.h"
#include "gci_utils.h"
#include "ThreadDebug.h"
#include "stdarg.h"
#include "dictionary.h"

static dictionary *devices = NULL;
static dictionary *device_open_count = NULL;

static int lock;    // Lock is global to all conroller instances as only one device can write at any one time.

#define DLL_POINTER(ptr_name) (controller->dllPointerTable.ptr_name)
#define DLL_POINTER_CREATE(function_name, ptr_type, ptr_name) \
{ \
    DLL_POINTER(ptr_name) = (ptr_type) GetProcAddress(controller->module, function_name); \
    if (DLL_POINTER(ptr_name) == NULL) \
    { \
        GCI_MessagePopup("Error", "Error: Can't find " function_name); \
        return NULL; \
    } \
}

static void ftdi_send_valist_error(FTDIController* controller, const char *title, const char *fmt, va_list ap)
{
    char message[500];

    SetSystemAttribute (ATTR_DEFAULT_MONITOR, 1);

    vsprintf(message, fmt, ap);

    if(controller->error_handler != NULL) {
        controller->error_handler(controller, title, message, controller->callback_data);
    }
}

static void ftdi_send_error(FTDIController* controller, const char *title, const char *fmt, ...)
{
    va_list ap;

    va_start(ap, fmt);

     ftdi_send_valist_error(controller, title, fmt, ap);

    va_end(ap);
}


void ftdi_controller_set_error_handler(FTDIController* controller, FT_ERROR_HANDLER handler, void *callback_data)
{
    controller->error_handler = handler;
    controller->callback_data = callback_data;
}

FTDIController* ftdi_controller_new(void)
{
    FTDIController *controller = (FTDIController *) malloc(sizeof(FTDIController));
    memset(controller, 0, sizeof(FTDIController));

    controller->module = LoadLibrary("Ftd2xx.dll");
    if(controller->module == NULL)
    {
        GCI_MessagePopup("Error", "Error: Can't Load ftd2xx dll");
        return NULL;
    }

    controller->debug_with_ints = 1;
    controller->debug = 0;

    DLL_POINTER_CREATE("FT_Read", PtrToRead, pRead);
    DLL_POINTER_CREATE("FT_Write", PtrToWrite, pWrite);
    DLL_POINTER_CREATE("FT_Open", PtrToOpen, pOpen);
    DLL_POINTER_CREATE("FT_OpenEx", PtrToOpenEx, pOpenEx);
    DLL_POINTER_CREATE("FT_ListDevices", PtrToListDevices, pListDevices);
    DLL_POINTER_CREATE("FT_Close", PtrToClose, pClose);
    DLL_POINTER_CREATE("FT_ResetDevice", PtrToResetDevice, pResetDevice);
    DLL_POINTER_CREATE("FT_Purge", PtrToPurge, pPurge);
    DLL_POINTER_CREATE("FT_SetTimeouts", PtrToSetTimeouts, pSetTimeouts);
```

```
    DLL_POINTER_CREATE("FT_GetQueueStatus", PtrToGetQueueStatus, pGetQueueStatus);
    DLL_POINTER_CREATE("FT_SetBaudRate", PtrToSetBaudRate, pSetBaudRate);
    DLL_POINTER_CREATE("FT_SetBitMode", PtrToSetBitMode, pSetBitMode);
    DLL_POINTER_CREATE("FT_SetDataCharacteristics", PtrToSetDataCharacteristics, pSetDataCharacteristics);
    DLL_POINTER_CREATE("FT_SetDeadmanTimeout", PtrToSetDeadmanTimeout, pSetDeadmanTimeout);

    if(devices == NULL) {
        devices = dictionary_new(20);
        device_open_count = dictionary_new(20);
    }

    return controller;
}

int ftdi_controller_get_lock(FTDIController* controller)
{
     return GciCmtGetLock (lock);
}

int ftdi_controller_release_lock(FTDIController* controller)
{
     return GciCmtReleaseLock (lock);
}

FT_STATUS ftdi_controller_open(FTDIController* controller, const char *device_id)
{
    FT_STATUS ftStatus;
    FT_HANDLE handle;

    if(strcmp(device_id, "") == 0)
        return FT_DEVICE_NOT_OPENED;

    handle = (FT_HANDLE) dictionary_getulong(devices, device_id, 01);

    if(handle != 01) {

        int count;

        controller->device_handle = handle;
        strncpy(controller->device_id, device_id, FTDI_SERIAL_NUMBER_LINE_LENGTH - 1);

        // Increment the open reference count for this device
        count = dictionary_getint(device_open_count, device_id, -1);
        if(count >= 0) {
            count++;
            dictionary_setint(device_open_count, device_id, count);
        }

        return FT_OK;
    }

    ftStatus = controller->dllPointerTable.pOpenEx((char*) device_id, FT_OPEN_BY_SERIAL_NUMBER, &(controller->device_handle));

    if (ftStatus != FT_OK)
    {
        if(controller->debug) {
            printf("Failed to open device %s\nAvailable devices are:\n", device_id);
        }

        ftdi_controller_print_serial_numbers_of_connected_devices(controller);

        return FT_DEVICE_NOT_OPENED;
    }

    GciCmtNewLock (device_id, 0, &(lock));
    strncpy(controller->device_id, device_id, FTDI_SERIAL_NUMBER_LINE_LENGTH - 1);
    dictionary_setulong(devices, device_id, (unsigned long) controller->device_handle);
    dictionary_setint(device_open_count, device_id, 1);

    // Set default timeouts
    ftdi_controller_set_timouts(controller, 1000.0, 1000.0);

    return FT_OK;
}

FT_STATUS ftdi_controller_set_timouts(FTDIController* controller, DWORD readTimeout , DWORD writeTimeout)
{
    FT_STATUS ftStatus = controller->dllPointerTable.pSetTimeouts(controller->device_handle, readTimeout, writeTimeout);

    return ftStatus;
}

FT_STATUS ftdi_controller_set_deadman_timout(FTDIController* controller, DWORD timeout)
{
    FT_STATUS ftStatus = controller->dllPointerTable.pSetDeadmanTimeout(controller->device_handle, timeout);

    return ftStatus;
}

FT_STATUS   ftdi_controller_set_baudrate(FTDIController* controller, unsigned long baudrate)
{
    FT_STATUS ftStatus = controller->dllPointerTable.pSetBaudRate(controller->device_handle, baudrate);

    return ftStatus;
}

FT_STATUS   ftdi_controller_set_data_characteristics(FTDIController* controller, unsigned char word_length
    , unsigned char stop_bits, unsigned char parity)
{
    return controller->dllPointerTable.pSetDataCharacteristics(controller->device_handle, word_length,
        stop_bits, parity);
}

FT_STATUS ftdi_controller_close(FTDIController* controller)
{
```

```
    int count = 0;

    if(controller->device_handle == NULL)
        return FT_DEVICE_NOT_OPENED;

    // Increment the open reference count for this device
    count = dictionary_getint(device_open_count, controller->device_id, -1);

    if(count > 0) {
        count--;
        dictionary_setint(device_open_count, controller->device_id, count);
    }

    if(count <= 0) {
        controller->dllPointerTable.pClose(controller->device_handle);
        dictionary_setulong(devices, controller->device_id, 0l);
    }

    return FT_OK;
}

void ftdi_controller_destroy(FTDIController* controller)
{
    ftdi_controller_close(controller);
}

int ftdi_controller_set_debugging(FTDIController* controller, int debug)
{
    controller->debug = debug;

    return FT_OK;
}

int ftdi_controller_show_debugging_bytes_as_integers(FTDIController* controller)
{
    controller->debug_with_ints = 1;

    return FT_OK;
}

int ftdi_controller_show_debugging_bytes_as_hex(FTDIController* controller)
{
    controller->debug_with_ints = 0;

    return FT_OK;
}

int ftdi_controller_get_number_of_connected_devices(FTDIController* controller)
{
    FT_STATUS ftStatus;
    int numDevs;

    ftStatus = controller->dllPointerTable.pListDevices(&numDevs, NULL, FT_LIST_NUMBER_ONLY);

    if (ftStatus == FT_OK)
        return numDevs;
    else
        return 0;
}


FT_STATUS ftdi_controller_set_bit_bang_mode(FTDIController* controller, unsigned char mask, unsigned char ucMode)
{
    return controller->dllPointerTable.pSetBitMode(controller->device_handle, mask, ucMode);
}

int ftdi_controller_print_serial_numbers_of_connected_devices(FTDIController* controller)
{
   FT_STATUS ftStatus;
   int i, numDevs;

   char **serial_numbers = (char**) malloc(sizeof(char*) * FTDI_MAX_NUMBER_OF_DEVICES);

   for(i=0; i < FTDI_MAX_NUMBER_OF_DEVICES; i++) {

        serial_numbers[i] = (char*) malloc(sizeof(char) * FTDI_SERIAL_NUMBER_LINE_LENGTH);
   }

   ftStatus = controller->dllPointerTable.pListDevices(serial_numbers, &numDevs,
                                                              FT_LIST_ALL|FT_OPEN_BY_SERIAL_NUMBER);
   if(ftStatus == FT_OK) {
       for(i=0; i < numDevs; i++) {
          printf("%s\n", serial_numbers[i]);
       }
   }

   for(i=0; i < FTDI_MAX_NUMBER_OF_DEVICES; i++) {

        free(serial_numbers[i]);
        serial_numbers[i] = NULL;
   }

   free(serial_numbers);

   return ftStatus;
}

FT_STATUS ftdi_controller_add_serial_numbers_of_connected_devices_to_ring_control(FTDIController* controller, int panel, int ctrl)
{
   FT_STATUS ftStatus;
   int i, numDevs;

   char **serial_numbers = (char**) malloc(sizeof(char*) * FTDI_MAX_NUMBER_OF_DEVICES);

   for(i=0; i < FTDI_MAX_NUMBER_OF_DEVICES; i++) {
```

```
        serial_numbers[i] = (char*) malloc(sizeof(char) * FTDI_SERIAL_NUMBER_LINE_LENGTH);
    }

    ftStatus = controller->dllPointerTable.pListDevices(serial_numbers, &numDevs,
                                                              FT_LIST_ALL|FT_OPEN_BY_SERIAL_NUMBER);
    if(ftStatus == FT_OK) {
        for(i=0; i < numDevs; i++) {
            InsertListItem (panel, ctrl, i, serial_numbers[i], serial_numbers[i]);
        }
    }

    for(i=0; i < FTDI_MAX_NUMBER_OF_DEVICES; i++) {

        free(serial_numbers[i]);
        serial_numbers[i] = NULL;
    }

    free(serial_numbers);

    return ftStatus;
}

static int i2c_error_check(FTDIController* controller, int stat)
{
    switch (stat){

        case 4:
        case 5:
            return FT_OK;  //OK

        case 1:
            ftdi_send_error(controller, "I2C Error", "Interrupt on I2C occurred");
            break;

        case 2:
            ftdi_send_error(controller, "I2C Error", "ACK error");
            break;

        case 8:
            ftdi_send_error(controller, "I2C Error", "I2C bus collision");
            break;

        default:
            break;
    }

    return FT_IO_ERROR;
}

static char * sprint_byte_array(BYTE* array, int array_length, const char* prefix, int use_ints, char *buffer)
{
    int i, written;
    char *ptr = buffer;
    const char* fmt = "0x%02x";

    written = sprintf(ptr, "%s [", prefix);
    ptr+=written;

    if (use_ints)
        fmt = "%d";

    for(i=0; i<array_length; i++) {

        written = sprintf(ptr, fmt, array[i]);
        ptr+=written;

        *ptr++ = ',';
    }

    *(--ptr) = ']'; // Get rid of last , and add a ]
    *(++ptr) = '\0';

    return buffer;
}

void ftdi_print_byte_array_to_stdout(BYTE* array, int array_length, const char* prefix, int use_ints)
{
    char buf[500] = "";

    char * b = sprint_byte_array(array, array_length, prefix, use_ints, buf);

    printf("%s\n", b);
    fflush(stdout);
}

FT_STATUS ftdi_controller_get_read_queue_status(FTDIController* controller, LPDWORD amount_in_rx_queue)
{
    FT_STATUS ftStatus;

    ftdi_controller_get_lock(controller);

    ftStatus = controller->dllPointerTable.pGetQueueStatus(controller->device_handle, amount_in_rx_queue);

    ftdi_controller_release_lock(controller);

    return ftStatus;
}

FT_STATUS ftdi_controller_purge_read_queue(FTDIController* controller)
{
    FT_STATUS ftStatus;

    ftdi_controller_get_lock(controller);
```

```
    ftStatus = controller->dllPointerTable.pPurge(controller->device_handle, FT_PURGE_RX);

    ftdi_controller_release_lock(controller);

    return ftStatus;
}

FT_STATUS ftdi_controller_purge_write_queue(FTDIController* controller)
{
    FT_STATUS ftStatus;

    ftdi_controller_get_lock(controller);

    ftStatus = controller->dllPointerTable.pPurge(controller->device_handle, FT_PURGE_TX);

    ftdi_controller_release_lock(controller);

    return ftStatus;
}

FT_STATUS ftdi_controller_i2c_write_bytes(FTDIController* controller, int address, int data_length, BYTE *data)
{
    FT_STATUS ftStatus;
    int i, total_raw_bytes;
    DWORD bytes_written_or_read = 0;
    BYTE raw_data[100] = "";

    ftdi_controller_get_lock(controller);

    raw_data[0] = 0x02;
    raw_data[1] = data_length + 1;
    raw_data[2] = address << 1;
    total_raw_bytes = data_length + 3;

    // Append data byte array
    for(i=0; i<data_length; i++)
        raw_data[i+3] = data[i];

    if(controller->debug) {
        ftdi_print_byte_array_to_stdout(raw_data, total_raw_bytes, "Writing actual data", controller->debug_with_ints);
    }

    ftStatus = controller->dllPointerTable.pWrite(controller->device_handle, raw_data, total_raw_bytes, &bytes_written_or_read);

    if (ftStatus != FT_OK || bytes_written_or_read != total_raw_bytes) {

        ftdi_controller_release_lock(controller);
        return ftStatus;
    }

    memset(raw_data, 0, sizeof(raw_data));

    ftStatus = controller->dllPointerTable.pRead(controller->device_handle, raw_data, 1, &bytes_written_or_read);

    if (ftStatus != FT_OK || bytes_written_or_read != 1) {

        ftdi_controller_release_lock(controller);
        return ftStatus;
    }

    ftdi_controller_release_lock(controller);

    return i2c_error_check(controller, raw_data[0]);
}


FT_STATUS ftdi_controller_i2c_fastline_write_bytes(FTDIController* controller, BYTE *data, int data_length)
{
    FT_STATUS ftStatus;
    int i, total_raw_bytes;
    DWORD bytes_written_or_read = 0;
    BYTE raw_data[100] = "";

    raw_data[0] = 0x01;
    raw_data[1] = data_length;
    total_raw_bytes = data_length + 2;

    // Append data byte array
    for(i=0; i<data_length; i++)
        raw_data[i+2] = data[i];

    ftdi_controller_get_lock(controller);

    if(controller->debug) {
        ftdi_print_byte_array_to_stdout(raw_data, total_raw_bytes, "Writing actual data", controller->debug_with_ints);
    }

    ftStatus = controller->dllPointerTable.pWrite(controller->device_handle, raw_data, total_raw_bytes, &bytes_written_or_read);

    if (ftStatus != FT_OK || bytes_written_or_read != total_raw_bytes) {

        ftdi_controller_release_lock(controller);
        return ftStatus;
    }

    controller->dllPointerTable.pGetQueueStatus(controller->device_handle, &bytes_written_or_read);

    // Old PICs don't return anything, so if nothing returned assume success
    if (bytes_written_or_read < 1) {
        ftdi_controller_release_lock(controller);
        return i2c_error_check(controller, 4);
    }

    memset(raw_data, 0, sizeof(raw_data));
```

```
    ftStatus = controller->dllPointerTable.pRead(controller->device_handle, raw_data, 1, &bytes_written_or_read);

    if (ftStatus != FT_OK || bytes_written_or_read != 1) {

        ftdi_controller_release_lock(controller);
        return ftStatus;
    }

    ftdi_controller_release_lock(controller);

    return i2c_error_check(controller, raw_data[0]);
}


FT_STATUS ftdi_controller_i2c_read_bytes(FTDIController* controller, int address, int data_length, BYTE *data)
{
    DWORD bytes_written_or_read = 0;
    FT_STATUS ftStatus;
    BYTE raw_data[100] = "";

    ftdi_controller_get_lock(controller);

    ftStatus = controller->dllPointerTable.pPurge(controller->device_handle, FT_PURGE_RX | FT_PURGE_TX);

    if (ftStatus != FT_OK) {
        ftdi_controller_release_lock(controller);
        return ftStatus;
    }

    raw_data[0] = 0x02;
    raw_data[1] = data_length + 1;
    raw_data[2] = address << 1;
    raw_data[2] = SETLSB(raw_data[2]);

    ftStatus = controller->dllPointerTable.pWrite(controller->device_handle, raw_data, 3, &bytes_written_or_read);

    if (ftStatus != FT_OK || bytes_written_or_read != 3) {

        ftdi_controller_release_lock(controller);
        return ftStatus;
    }

    if(controller->debug) {
        ftdi_print_byte_array_to_stdout(raw_data, 3, "Writing actual data from read", controller->debug_with_ints);
    }

    memset(raw_data, 0, sizeof(raw_data));

    ftStatus = controller->dllPointerTable.pRead(controller->device_handle, data, data_length + 1, &bytes_written_or_read);

    if (ftStatus != FT_OK || bytes_written_or_read != (data_length + 1)) {

        ftdi_controller_release_lock(controller);
        return ftStatus;
    }

    if(controller->debug) {
        ftdi_print_byte_array_to_stdout(data, data_length + 1, "Reading data", controller->debug_with_ints);
    }

    ftdi_controller_release_lock(controller);

    return i2c_error_check(controller, data[data_length]);
}

FT_STATUS ftdi_controller_i2c_fastline_read_bytes(FTDIController* controller, int data_length, BYTE *data)
{
    int i, total_raw_bytes;
    DWORD bytes_written_or_read = 0;
    FT_STATUS ftStatus;
    BYTE raw_data[100] = "";

    ftdi_controller_get_lock(controller);

    ftStatus = controller->dllPointerTable.pPurge(controller->device_handle, FT_PURGE_RX | FT_PURGE_TX);

    if (ftStatus != FT_OK) {
        ftdi_controller_release_lock(controller);
        return ftStatus;
    }

    raw_data[0] = 0x01;
    total_raw_bytes = data_length + 1;
    raw_data[1] = total_raw_bytes;

    // Append data byte array
    for(i=0; i<total_raw_bytes; i++)
        raw_data[i+2] = data[i];

    ftStatus = controller->dllPointerTable.pWrite(controller->device_handle, raw_data, total_raw_bytes + 3, &bytes_written_or_read);

    if (ftStatus != FT_OK || bytes_written_or_read != (total_raw_bytes + 3)) {

        ftdi_controller_release_lock(controller);
        return ftStatus;
    }

    if(controller->debug) {
        ftdi_print_byte_array_to_stdout(raw_data, total_raw_bytes + 3, "Writing actual data from read", controller->debug_with_ints);
    }

    memset(raw_data, 0, sizeof(raw_data));

    ftStatus = controller->dllPointerTable.pRead(controller->device_handle, data, total_raw_bytes - 1, &bytes_written_or_read);
```

```c
    if (ftStatus != FT_OK) {

        ftdi_controller_release_lock(controller);
        return ftStatus;
    }

    if(controller->debug) {
        ftdi_print_byte_array_to_stdout(data, total_raw_bytes - 1, "Reading data", controller->debug_with_ints);
    }

    ftdi_controller_release_lock(controller);

    return i2c_error_check(controller, 4);  // No status byte returned
}


FT_STATUS ftdi_controller_write_bytes(FTDIController* controller, int data_length, BYTE *data)
{
    FT_STATUS ftStatus;
    DWORD bytes_written_or_read = 0;

    if(controller->device_handle == 0)
        return FT_DEVICE_NOT_OPENED;

    ftdi_controller_get_lock(controller);

    if(controller->debug) {
        ftdi_print_byte_array_to_stdout(data, data_length, "Writing actual data", controller->debug_with_ints);
    }

    ftStatus = controller->dllPointerTable.pWrite(controller->device_handle, data, data_length, &bytes_written_or_read);

    if (ftStatus != FT_OK || bytes_written_or_read != data_length) {

        printf("Error sending data\n");
        ftdi_controller_release_lock(controller);
        return ftStatus;
    }
    else {
        if(controller->debug) {
            printf("Data send successfully\n");
        }
    }

    ftdi_controller_release_lock(controller);

    return ftStatus;
}

FT_STATUS   ftdi_controller_write_byte(FTDIController* controller, BYTE data)
{
    BYTE bytes[1] = {data};

    return ftdi_controller_write_bytes( controller, 1, bytes);
}

FT_STATUS ftdi_controller_write_string(FTDIController* controller, const char* fmt, ...)
{
    FT_STATUS ftStatus;
    char buffer[200] = "";
    va_list ap;
    va_start(ap, fmt);
    vsprintf(buffer, fmt, ap);
    va_end(ap);

    ftStatus = ftdi_controller_write_bytes(controller, strlen(buffer), buffer);

    if(ftStatus != FT_OK)
        return  ftStatus;

    return ftStatus;
}

FT_STATUS ftdi_controller_read_bytes(FTDIController* controller, int data_length, BYTE *data)
{
    DWORD bytes_written_or_read = 0;
    FT_STATUS ftStatus;

    if(controller->device_handle == 0)
        return FT_DEVICE_NOT_OPENED;

    ftdi_controller_get_lock(controller);

    ftStatus = controller->dllPointerTable.pRead(controller->device_handle, data, data_length, &bytes_written_or_read);

    if (ftStatus != FT_OK || bytes_written_or_read != (data_length)) {

        ftdi_controller_release_lock(controller);
        return ftStatus;
    }

    if(controller->debug) {
        ftdi_print_byte_array_to_stdout(data, data_length, "Reading data", controller->debug_with_ints);
    }

    ftdi_controller_release_lock(controller);

    return ftStatus;
}

FT_STATUS ftdi_controller_read_bytes_availiable_in_rx_queue(FTDIController* controller, BYTE *data, LPDWORD bytes_read)
{
    FT_STATUS ftStatus;
    DWORD amount_in_rx_queue = 0;
```

```
    ftStatus = ftdi_controller_get_read_queue_status(controller, &amount_in_rx_queue);

    if(ftStatus != FT_OK)
        return ftStatus;

    if(amount_in_rx_queue == 0)
        return FT_IO_ERROR;

    ftStatus = ftdi_controller_read_bytes(controller, amount_in_rx_queue, data);

    if(ftStatus != FT_OK)
        return ftStatus;

    *bytes_read = amount_in_rx_queue;

    return ftStatus;
}
```

## 4. High level software

Higher level code communicates transparently with the device using the above code; appropriate strings for the device must obviously be sent. The COM port the device is connected to for the VCP driver variant or the serial number of the FTDI device for the D2XX variant must be known and suplied to the code.

Initialisation with the VCP variant:

```
#define PORT 3   //e.g. for COM3
GCI_initI2C();
```

Initialisation with the D2XX variant:

```
FTDIController* controller = ftdi_controller_new();
ftdi_controller_open(controller, "ELBSAT12");        //e.g. for chip with serial number ELBSAT12
```

With respect to our open microscopes, most use the native D2XX call method but this has led to problems on some hardware (maybe due to conflicts with other connected hardware that also use ftdi devices) and so some use the VCP for RS232 commands. Computers that use the native call method must have the serial number of the FTDI device in config.ini. The FTDI serial number can be obtained from one of the test programs that identifies the devices at startup and enters them into the drop-down menu. It can also be obtained from the Device Manager in the Windows Control Panel. In Device Manager, open up the Universal Serial Bus Controllers tree. The device will be listed as a USB Serial Converter. Right-click on this and select Properties. On the Details tab, select Device Instance Id from the drop down menu. You will see a string of characters similar to “USB\VID_0403&PID_6001\ELBSAT12”. The serial number is the last 8 characters of this string (after the ‘\’).

Use of the VCP requires that the VCP driver is loaded (see device manager for the USB Serial Converter) such that there is an entry in device manager under Ports (COM & LPT) for USB Serial Port (e.g. COM4), which indicates the COM port in use. Computers that use this method must have the COM port number entered in config.ini. Most config.ini files have both sets of information added for native or VCP use. The test programs for individual hardware components all use the VCP.

This note was prepared by B. Vojnovic, RG Newman and PR Barber in October 2007 and updated in September 2011. Thanks are due to A. Manser who developed the original $I^2C$ drivers and of course to Rosalind Locke and to RG Newman for enhancing them. Glenn Pierce was involved in high level code development.RG Newman also designed the printed circuit layouts and constructed numerous units used in several instruments. Board layouts are available on request (Number One Systems EasyPC (version 14 or below, http://www.numberone.com/).

We acknowledge the financial support of Cancer Research UK, the MRC and EPSRC.