ADC conversion

Here is the initial coding for the ADC conversion routines. Only serial connected ADCs are supported.

The PC sends a command to the USB controller formatted thus

Command | ClockOptions | ClockDelay | Bits | Average

And then reads from the USB controller chip until it gets the result. Typically, this will be on the 3rd read due to double buffering in the controller

The Command is one of these

B0 – Read the magnitude ADC only
B1 – Read the phase ADC only
B2 – Read both ADCs

The clock options is one of these

00 – clock as per AD7685
01 – clock as per LT1860

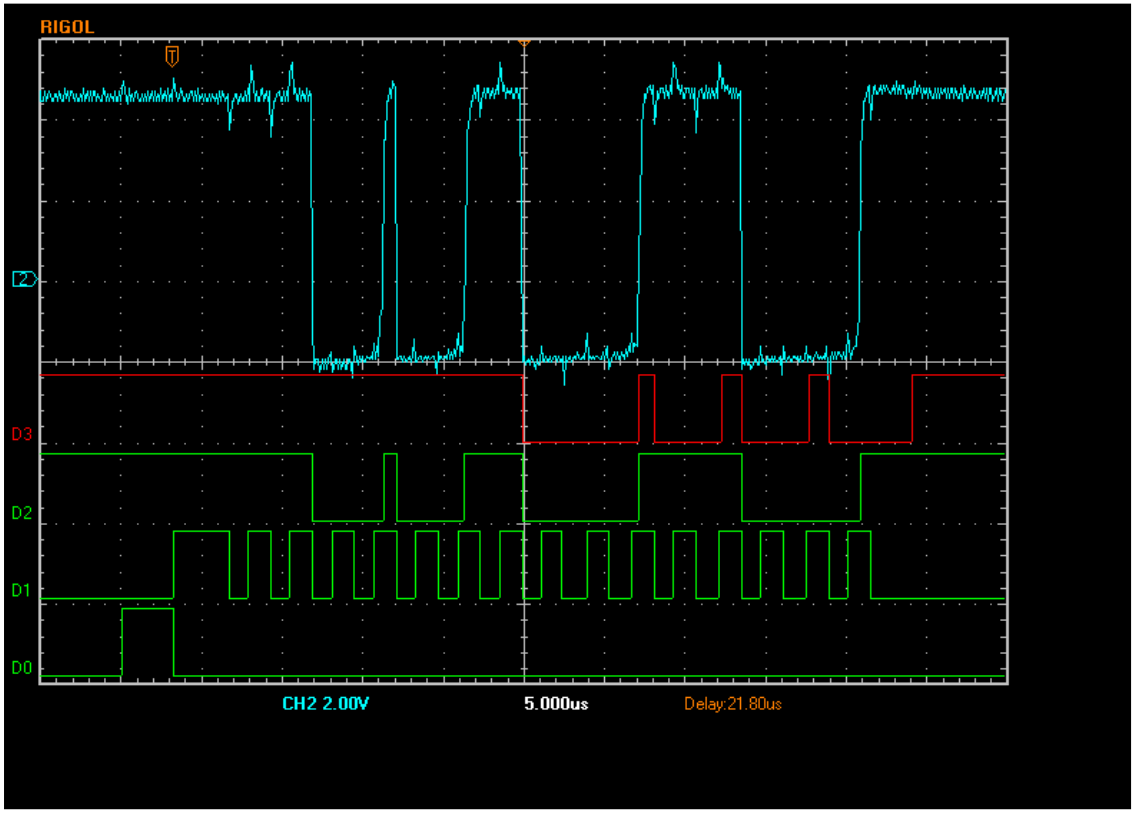
Clock delay is the conversion time delay after raising the convert line until it is lowered and clocking the result starts. The delay obtained is approximately N+(0.585D) used where N is 2.47 and D is the value of ClockDelay. The value to use for the AD7865 is therefore 01 to give a small margin and 03 for the LT1865.

Bits is the number of bits of data; so 0C for the LT1865 and 10 for the AD7685

Average is used to perform multiple ADC conversions and average the results.

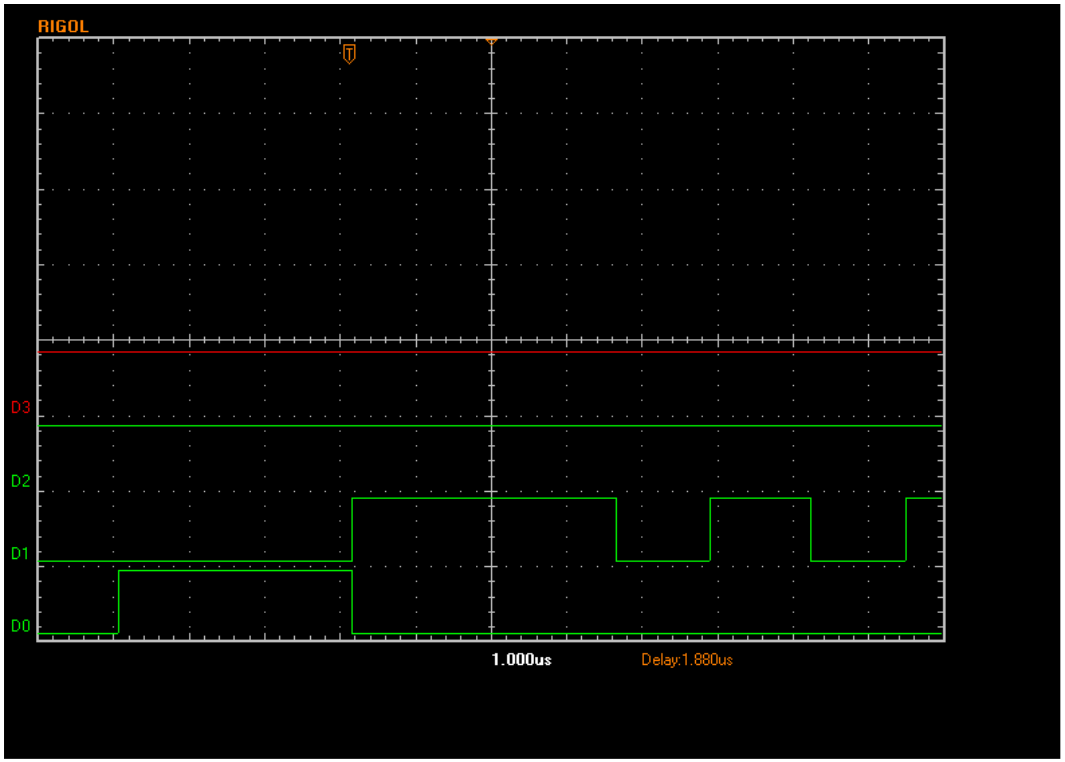
Here is a single ADC conversion with the following command string, which is a request to read the magnitude ADC only with an AD7685

B1 00 01 10 01

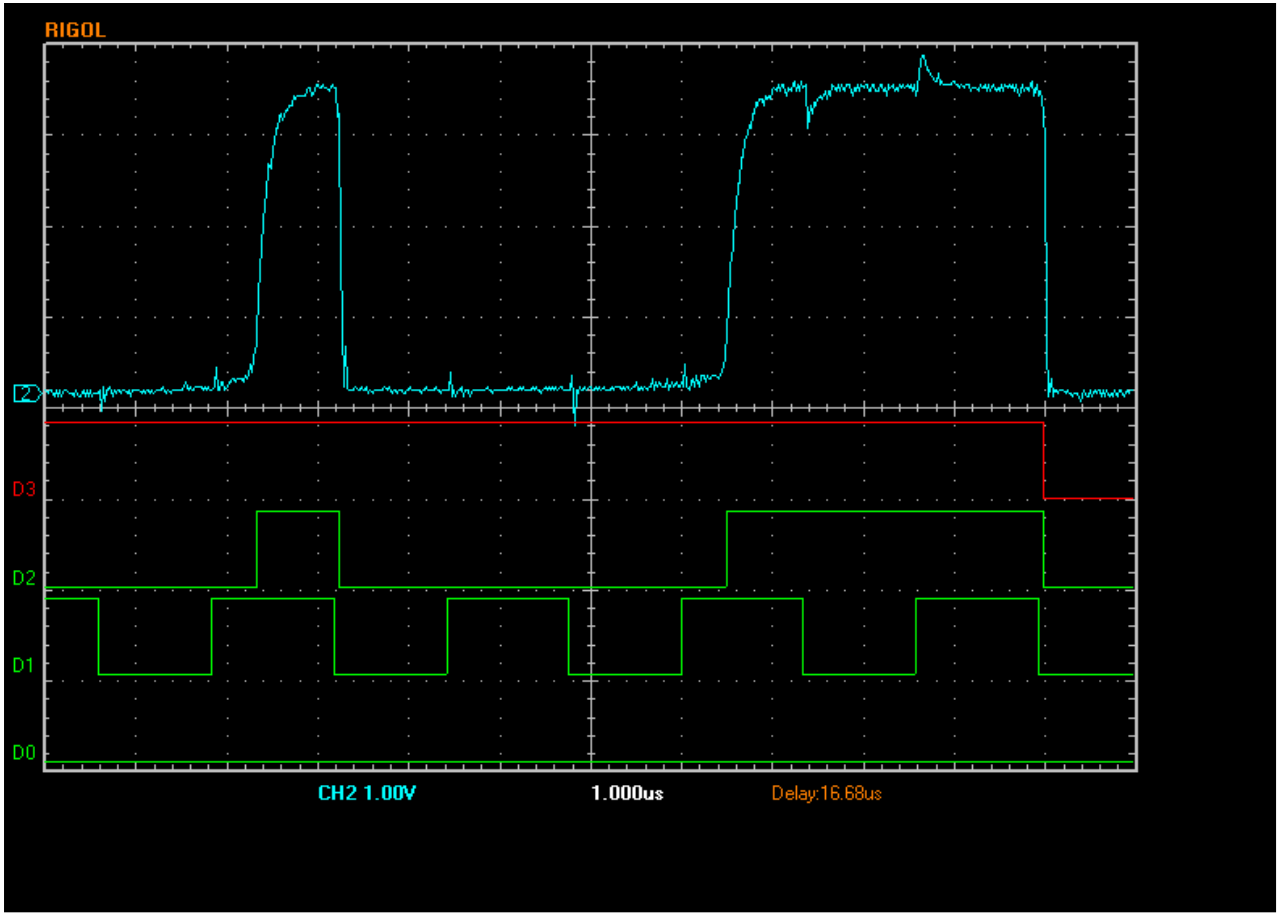


The bottom trace is the convert line, the line above the clock line. D2 is the magnitude data and D3 is phase, present still but the code is ignoring it. The top trace is a scope rather than a logic analyser trace on the ADC data out line. More of this later. In the trace can be seen the convert line go high followed by the clock line high whih then goes low 16 times, one for each bit. The data can be seen clocking out above this as D2.

Zooming in on the first part of the trace yields this. The convert line was high for just over 3 usec against a specification time requirement from the datasheet of 2.2 usec. There is a short delay to the first clock low edge then the data is clocked out at a cycle rate of just under 3 usec per bit.



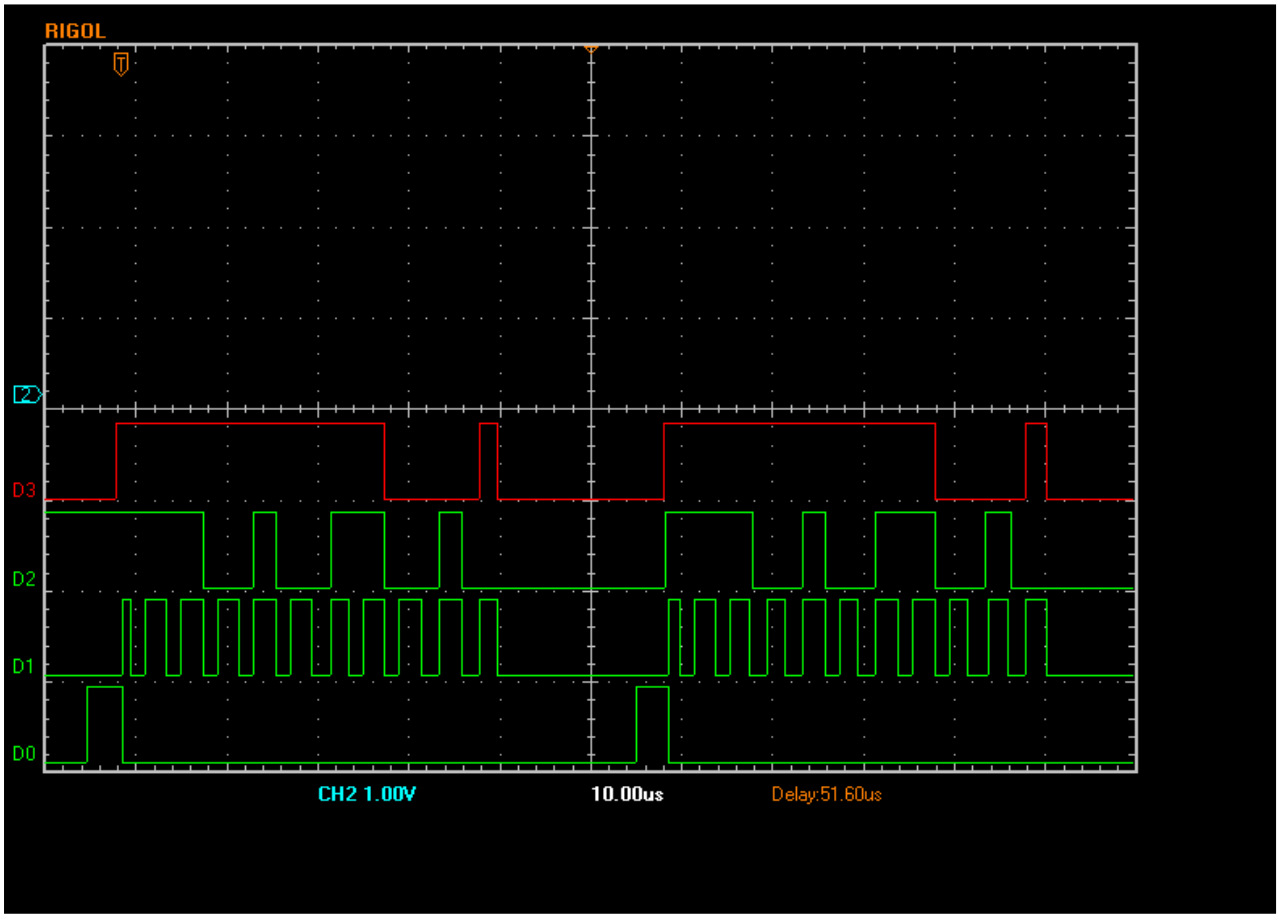
Now looking slightly later as the data is being clocked out.



The data line is shown at the top. The data changes state on falling edges of the clock; check for example the blue trace falling lines that follow slightly the falling clock edges. The rising edges though show a problem. There is a delay of about 2 usec from a falling edge to when the trace starts to rise. This corresponds to the ADC line going low and the digital transistor turning off. It gets pulled up by a 1k2 pullup to 3.3V. Why this is is not yet known but Scotty finds the same on his hardware and is investigating. In the meantime the code has been structured to read the data lines just before the clock goes low as this seems to give at least 500 ns margin.

For a command string thus (LT1865, average = 2, convert both) the trace is as follows, showing the extra clock transition at the start and two sequential conversions. The code reads both ADCs in both conversions and averages them before returning the two results.

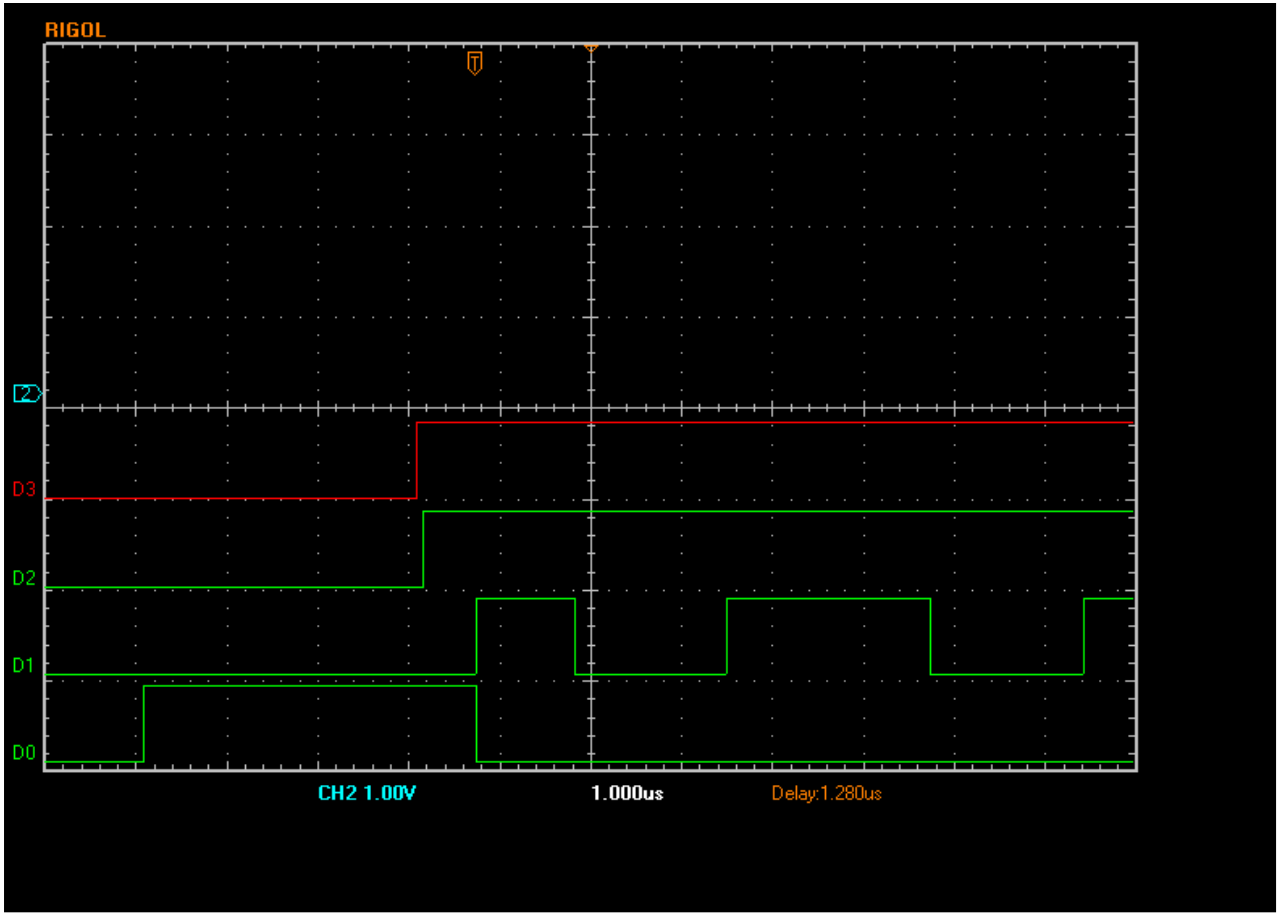
B2 01 02 0A 02



The trace starts with a convert signal, followed by an extra clock edge that the lt1865 apparenlty needs followed by 10 falling edges. It is not quite as the datasheet but follows the basic code so some testing may be needed (I don't have an LT1865 to try).

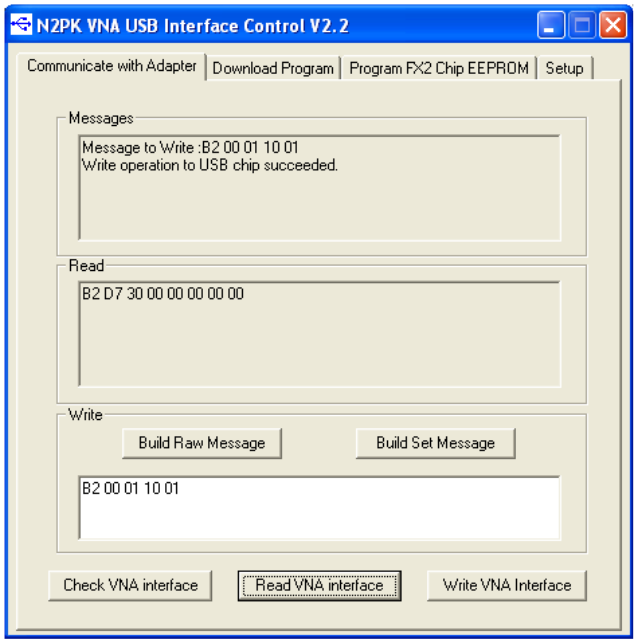
After the data is clocked out there is a 15 usec delay (whilst the code stores the results) then a second convert & clock. It would be possible to rearrange the code to remove some of this delay but it does not seem worthwhile.

Zooming in on the start of the trace….

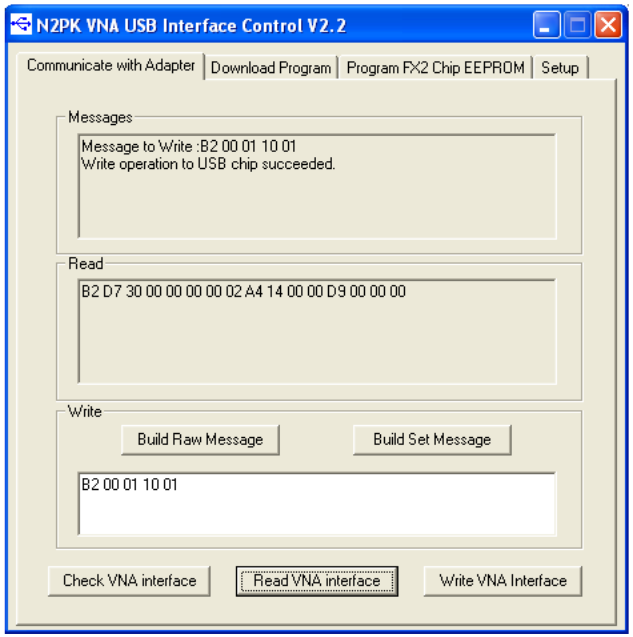


The convert line is high for about 3.7 usec against a specification sheet requirement of 3.3 usec worst case (in fact you can see the data lines changing earlier at about 3usec). The short clock pulse is the extra clock event, then clocking the data starts. This time it happens at a rate of about 3.9 usec per bit because here we are clocking out both ADCs at once. The code will also support 24 bit ADCs and the clocking will slow down slightly again due to the need to manage 32 bit results in that case.

A typical response from the USB interface to a dual ADC conversion is this before the result is ready (the result is available on the 3rd read of the interface due to buffering). B2 was the last command given, D7 is (at the moment) just random data; it is used for status flags but is not being set yet. The next 5 bytes (30 and 4 bytes of 00) are the state of the five I/O ports, A, B, C, D and E. The last byte is the number of ADC conversions present.



When the result is available it changes to this



The response (B2 D7 30 00 00 00 00 02 A4 14 00 00 D9 00 00 00) parses as follows:

B2 – last command processed
D7 – flags. No meaningful data yet (not coded anything so far here)
30 – port A data lines at the moment
00 – port B data lines at the moment
00 – port C data lines at the moment
00 – port D data lines at the moment
00 – port E data lines at the moment
02 – number of ADC results present
A4 14 00 00 Magnitude result (0x000014A4 = decimal 5284)
D9 00 00 00 Phase result (0x000000D9 = decimal 217)

All results are returned as unsigned long integers. Calculations internally in the USB chip are also performed as long integers so the code will support 24 bit ADC data, but for 16 bit or less it optimises its processing for speed. Hence the code can support averaging of up to 255 readings with 24 bit ADC words without overflow.

Internally the code looks like this for a dual detector reading:

```
void DoAdcBothRead()
{
        BYTE temp;
        BYTE clock;
        BYTE nBits;
        BYTE nAverage = EP2FIFOBUF[CmdMsaReadAverage];
        if( !nAverage )
                nAverage = 1;
        nResultPhase = nResultMag = 0;
        do
        {
                temp = IOA | bmAdcConv;
                IOA = temp & ~bmAdcSerClk;
                clock = EP2FIFOBUF[CmdMsaReadClockDelay];
                nBits = EP2FIFOBUF[CmdMsaReadBits]-1;
                while( clock-- > 0 )
                        ;
                temp = IOA | bmAdcSerClk;
                IOA = temp & ~bmAdcConv;
                if( EP2FIFOBUF[CmdMsaReadClockOptions] & 1 )
                        IOA &= ~bmAdcSerClk;
                if( nBits <= 16 )
                {
                        nResultPhaseTempShort = nResultMagTempShort = 0;
                        do
                        {
                                IOA |= bmAdcSerClk;
                                nResultMagTempShort <<= 1;
                                nResultPhaseTempShort <<= 1;
                                temp = IOA;
                                IOA &= ~bmAdcSerClk;
                                if( (temp & bmMagData ) == 0 )
                                        nResultMagTempShort |= 1;
                                if( (temp & bmPhaseData ) == 0 )
                                        nResultPhaseTempShort |= 1;
                        } while ( nBits-- > 0 );
                        nResultMag += nResultMagTempShort;
                        nResultPhase += nResultPhaseTempShort;
                }
                else
                {
                        nResultPhaseTempLong = nResultMagTempLong = 0;
                        do
                        {
                                IOA |= bmAdcSerClk;
                                nResultMagTempLong <<= 1;
                                nResultPhaseTempLong <<= 1;
                                temp = IOA;
```

```
                        IOA &= ~bmAdcSerClk;
                        if( (temp & bmMagData ) == 0 )
                                nResultMagTempLong |= 1;
                        if( (temp & bmPhaseData ) == 0 )
                                nResultPhaseTempLong |= 1;
                } while ( nBits-- > 0 );
                nResultMag += nResultMagTempLong;
                nResultPhase += nResultPhaseTempLong;
        }
    } while ( --nAverage > 0 );
    nAverage = EP2FIFOBUF[CmdMsaReadAverage];
    if( !nAverage )
            nAverage = 1;
    if( nAverage != 1 )
    {
            nResultMag /= nAverage;
            nResultPhase /= nAverage;
    }
}
```

There are separate routines for reading the magnitude and phase separately that are optimised by removing the unnecessary lines. Unwrapping the loops into simple inline code would also be quicker but it is not clear that this is needed.

Ok, reading the data into the Basic program….

At the moment a crude string based interface. The USB controller returns a structure like this when read

```
typedef struct _MSA_RXBUFFER {
        unsigned char last_command;          // command type last received
        unsigned char return_status;  // see below for status flag defintions
        unsigned char ioa;                          // FX2 PortA data
        unsigned char iob;                          // FX2 PortB data
        unsigned char ioc;                          // FX2 PortC data
        unsigned char iod;                          // FX2 PortD data
        unsigned char ioe;                          // FX2 PortE data
        unsigned char ADC_reads_done; // Number of ADC reads performed
        unsigned char data[240];             // VARIABLE nuumber of ADC reads performed
} MSA_RXBUFFER;
```

The DLL needs to massage this into a form that the rather annoyingly limited basic can handle, so the interface at the moment is as follows. Liberty Basic passes a null terminated string buffer initialised as spaces to the DLL. The DLL reads the USB controller then fills in the string as shown

```
#define HEX(x) ( ((x)&0x0f) < 10 ? ((x)&0x0f)+'0' : ((x)&0x0f)+'A'-10)

extern "C" __declspec(dllexport) int  UsbMSADeviceReadString( void *pMSAData, char *data, int message_size )
{
        MSA_RXBUFFER readbuf;
        MSADevice* MSA = (MSADevice* )pMSAData;
        if( !MSA->Read( &readbuf ) )
                return false;
        data[0] = HEX( readbuf.last_command >> 4);
        data[1] = HEX( readbuf.last_command  );
        data[2] = HEX( readbuf.return_status >> 4);
        data[3] = HEX( readbuf.return_status  );
        data[4] = HEX( readbuf.ioa >> 4);
        data[5] = HEX( readbuf.ioa  );
        data[6] = HEX( readbuf.iob >> 4);
        data[7] = HEX( readbuf.iob  );
        data[8] = HEX( readbuf.ioc >> 4);
        data[9] = HEX( readbuf.ioc  );
        data[10] = HEX( readbuf.iod >> 4);
        data[11] = HEX( readbuf.iod  );
        data[12] = HEX( readbuf.ioe >> 4);
        data[13] = HEX( readbuf.ioe  );
        data[14] = HEX( readbuf.ADC_reads_done >> 4);
        data[15] = HEX( readbuf.ADC_reads_done  );
        for( int i=0; i<readbuf.ADC_reads_done; i++)
        {
                data[16+i*8] = HEX( readbuf.data[i*4+3] >> 4);
```

```
                data[17+i*8] = HEX( readbuf.data[i*4+3]  );
                data[18+i*8] = HEX( readbuf.data[i*4+2] >> 4);
                data[19+i*8] = HEX( readbuf.data[i*4+2]  );
                data[20+i*8] = HEX( readbuf.data[i*4+1] >> 4);
                data[21+i*8] = HEX( readbuf.data[i*4+1]  );
                data[22+i*8] = HEX( readbuf.data[i*4] >> 4);
                data[23+i*8] = HEX( readbuf.data[i*4]  );
        }
        return true;
}
```

Thus the basic program gets back a string. Here is the code to do it…

```
[Read16wSlimCBUSB]
    if USBdevice = 0 then return
    USBwrbuf$ = "B200021001"
    UsbAdcCount = 0
    UsbAdcResult1 = 0
    UsbAdcResult2 = 0
    CALLDLL #USB, "UsbMSADeviceWriteString", USBdevice as long, USBwrbuf$ as ptr, 5 as short, result as boolean
    for clmn = 0 to 39 'ver111-21
        USBrdbuf$ = space$(63)+chr$(0)
        CALLDLL #USB, "UsbMSADeviceReadString", USBdevice as long, USBrdbuf$ as ptr, 64 as short, result as boolean
        if result = 0 then return
        UsbAdcCount = HEXDEC( mid$( USBrdbuf$, 15, 2 ) )
        if UsbAdcCount > 0 then
            UsbAdcResult1 = HEXDEC( mid$( USBrdbuf$, 17, 8 ) )
            if UsbAdcCount > 1 then UsbAdcResult2 = HEXDEC( mid$( USBrdbuf$, 25, 8 ) )
            return
        end if
    next clmn
    return 'to [ReadMagnitude]or[ReadPhase]with status words
```

This code is called thus

```
        [ReadAD16Status]'For reading the 16 bit serial AtoD. Changed 4-27-10 ver115-9b
           'This routine modified to help reject noise glitching caused by newer, faster, cheaply made computers.
           'needed: port, control, status ; creates: 16 status port words (stat15-stat0)mag,(and, pha if two A/D's installed) 'ver111-33a
           'written for Analog Devices, AD7685, but other 16 bit serial AtoD's will probably work with this code
           'reads 16 bit serial using Original or Slim Control Board. ver111-33c
           'good for 16 Bit Original AtoD Module or SLIM-ADC-16
           'MAG is WAIT, PHASE is ACK, SCLK is BD6, CVN is BD7.
            if cb = 2 then goto[Read16wSlimCB]' if SLIM Contol Board, jump over [Read16wOrigCB]
            if cb = 3 then goto[Read16wSlimCBUSB]' if SLIM Contol Board, jump over [Read16wOrigCB]
```

And the conversion functions that generate the result from the status words is not needed so codes thus (same for magnitude only)

```
[Process16MagPha]'ver111-33a
    'process the stat15-0 for both magnitude and phase.Determines magdata bit (D7) and phadata bit (D6) in each word
    if cb = 3 then
        magdata = UsbAdcResult1
        phadata = UsbAdcResult1
        return
    end if
    magdata = 0
    phadata = 0
etc etc
```