Guide to the USB code in the MSA Basic application

This document summarises the changes made to the Basic code to implement a USB interface.

Basics

The USB interface is accessed by calling a DLL – msadll.dll. This DLL exists to provide an interface that Liberty basic is capable of using and codes a small amount of functionality to get around some of Liberty's limitations

In the code, a call to the DLL looks like this.

```
if USBdevice <> 0 then CALLDLL #USB, "UsbMSADeviceWriteString", USBdevice as
long, USBwrbuf$ as ptr, 4 as short, result as boolean
```

In this call, USBdevice is a Windows "handle" used to access the interface. If zero, then the device has not been found. So, if the handle is not zero, call the USB interface using the handle opened elsewhere in the program as #USB.

The function accessed in the DLL is called `UsbMSADeviceWriteString` and it takes 3 parameters and returns a single result. The parameters passed here are as follows:

```
USBdevice as long,
USBwrbuf$ as ptr,
4 as short
```

And the result is

```
result as boolean
```

At the moment there is precious little checking of the return codes – the application just carries on regardless. Must change this sometime.

In all calls to this DLL other than the basic open/close functions, we will pass the handle USBdevice and will have a Boolean result of the function (zero for fail, not zero for success). The other parameters vary by function. In this case there is a String called USBwrbuf$ and an integer value of 4.

What do the functions do? That will become clear later.

The interface is opened thus

```
if uVerifyDLL("msadll") then bUsbAvailable = 1 else bUsbAvailable = 0
if bUsbAvailable then call UsbOpenInterface 'USB:01-08-2010
```

Having checked that the interface DLL is available, call the open function.

```
' ------------------------------------------------------------------------
' call this to try to open the USB interface

sub UsbOpenInterface  'USB:01-08-2010
    if UsbInterfaceOpen = 0 then
        USBdevice = 0
        on error goto [UsbInterfaceOpenError]
        open "msadll" for dll as #USB
        UsbInterfaceOpen = 1
    end if
    if USBdevice = 0 then
        on error goto [UsbInterfaceInitError]
        CALLDLL #USB, "UsbMSAInitialise", USBdevice as long
    end if
    exit sub
[UsbInterfaceOpenError]
    exit sub
[UsbInterfaceInitError]
    close #USB
    UsbInterfaceOpen = 0
    exit sub
end sub 'UsbOpenInterface
```

A similar function is called on exit

```
' ------------------------------------------------------------------------
' call this to close the USB interface

sub UsbCloseInterface 'USB:01-08-2010
    if USBdevice <> 0 then
         CALLDLL #USB, "UsbMSARelease", USBdevice as long, result as boolean
         USBdevice = 0
    end if
    if UsbInterfaceOpen <> 0 then
        close #USB
        UsbInterfaceOpen = 0
    end if
    exit sub
end sub 'UsbCloseInterface
```

ADC Input

There is one small routine that handles ass ADC input.

```
   ' Generic code for USB ADC input regardless of number of bits and ADC type
[Read22wSlimCBUSB] 'USB:01-08-2010
    USBwrbuf$ = "B201040A01"
    goto [ReadCommonwSlimCBUSB]

[Read16wSlimCBUSB] 'USB:01-08-2010
    USBwrbuf$ = "B200021001"
    ' fall through
[ReadCommonwSlimCBUSB] 'USB:01-08-2010
    if USBdevice = 0 then return
    UsbAdcCount = 0
    UsbAdcResult1 = 0
    UsbAdcResult2 = 0
    CALLDLL #USB, "UsbMSADeviceReadAdcs", USBdevice as long, USBwrbuf$ as ptr,_
                              5 as short, USBrBuf as struct, result as boolean
    if( result ) then
        UsbAdcCount = USBrBuf.numreads.struct
        UsbAdcResult1 = USBrBuf.magnitude.struct
        UsbAdcResult2 = USBrBuf.phase.struct
    end if
    return
```

That's all there is to it. The DLL and interface handle variable bit sizes and clocking schemes; at the moment just the AD7685 and LT1860 are supported

The code shows one function; write to the MSA to instruct the USB controller to perform an ADC conversion as defined by the USBwrbuf$ string then read the ADCs. The string being written is explained later but essentially it tells the USB interface to perform an ADC conversion of a given type. The result is then read within the DLL and the result returned as one or two integers.

The variable UsbAdcCount holds the number of ADC results returned (which should be 2 given the USB controller command 'B2') and the results are in the two variables.

The Liberty basic code that clocks the result in bit by bit is skipped as is the code that reads these bits back into an integer result – for example in this function

```
[Process16MagPha]'ver111-33a
    if cb = 3 then 'USB:01-08-2010
        magdata = UsbAdcResult1
        phadata = UsbAdcResult1
        return
    end if
```

Setting the MSA outputs

There are several different methods used to drive the hardware. One would suffice but we need to code around Liberty basic. These are probably not the best ways to do it but it works……

Why landscape format? Some of these code lines are a bit long!

Method 1 – using integers

This is the method used for the CommandPLLSlim function.

Here is the coding for the parallel port SLIM

```
[CommandPLLslim]'needs:datavalue,levalue,N23-N0,control,Jcontrol,port,contclear,LEPLL ; commands N23-N0,SLIM ControlBoard ver111-28
    'used during initialization of PLL1, PLL2, and PLL3.  PDM will get set to "0" during Initializations
    'selt word = 1 common clock, 4 datas, plus 3 (filtbank). entering this sub, selt word should = filtbank only
    'init word = 5 latch lines plus 2 pdm commands. entering this sub, init word should = pdmcmd + pdmclk only.ver111-39d
    'two steps to do: command data and clock without disturbing Filter Bank, then send LE without disturbing PDM
  'step 1. Command the PLL without changing the filter bank.
    'For PLL1,datavalue=2, for PLL2,datavalue=16, for PLL3,datavalue=8
    'following code lines changed in ver113-3c
    a=filtbank + N23*datavalue:out port, a:out control, SELT:out control, contclear:out port, a+1:out control, SELT:out control, contclear
    a=filtbank + N22*datavalue:out port, a:out control, SELT:out control, contclear:out port, a+1:out control, SELT:out control, contclear
    a=filtbank + N21*datavalue:out port, a:out control, SELT:out control, contclear:out port, a+1:out control, SELT:out control, contclear
    a=filtbank + N20*datavalue:out port, a:out control, SELT:out control, contclear:out port, a+1:out control, SELT:out control, contclear
    a=filtbank + N19*datavalue:out port, a:out control, SELT:out control, contclear:out port, a+1:out control, SELT:out control, contclear
    a=filtbank + N18*datavalue:out port, a:out control, SELT:out control, contclear:out port, a+1:out control, SELT:out control, contclear
    a=filtbank + N17*datavalue:out port, a:out control, SELT:out control, contclear:out port, a+1:out control, SELT:out control, contclear
    a=filtbank + N16*datavalue:out port, a:out control, SELT:out control, contclear:out port, a+1:out control, SELT:out control, contclear
    a=filtbank + N15*datavalue:out port, a:out control, SELT:out control, contclear:out port, a+1:out control, SELT:out control, contclear
    a=filtbank + N14*datavalue:out port, a:out control, SELT:out control, contclear:out port, a+1:out control, SELT:out control, contclear
    a=filtbank + N13*datavalue:out port, a:out control, SELT:out control, contclear:out port, a+1:out control, SELT:out control, contclear
    a=filtbank + N12*datavalue:out port, a:out control, SELT:out control, contclear:out port, a+1:out control, SELT:out control, contclear
    a=filtbank + N11*datavalue:out port, a:out control, SELT:out control, contclear:out port, a+1:out control, SELT:out control, contclear
    a=filtbank + N10*datavalue:out port, a:out control, SELT:out control, contclear:out port, a+1:out control, SELT:out control, contclear
    a=filtbank + N9*datavalue:out port, a:out control, SELT:out control, contclear:out port, a+1:out control, SELT:out control, contclear
    a=filtbank + N8*datavalue:out port, a:out control, SELT:out control, contclear:out port, a+1:out control, SELT:out control, contclear
```

```
    a=filtbank + N7*datavalue:out port, a:out control, SELT:out control, contclear:out port, a+1:out control, SELT:out control, contclear
    a=filtbank + N6*datavalue:out port, a:out control, SELT:out control, contclear:out port, a+1:out control, SELT:out control, contclear
    a=filtbank + N5*datavalue:out port, a:out control, SELT:out control, contclear:out port, a+1:out control, SELT:out control, contclear
    a=filtbank + N4*datavalue:out port, a:out control, SELT:out control, contclear:out port, a+1:out control, SELT:out control, contclear
    a=filtbank + N3*datavalue:out port, a:out control, SELT:out control, contclear:out port, a+1:out control, SELT:out control, contclear
    a=filtbank + N2*datavalue:out port, a:out control, SELT:out control, contclear:out port, a+1:out control, SELT:out control, contclear
    a=filtbank + N1*datavalue:out port, a:out control, SELT:out control, contclear:out port, a+1:out control, SELT:out control, contclear
    a=filtbank + N0*datavalue:out port, a:out control, SELT:out control, contclear:out port, a+1:out control, SELT:out control, contclear
    out port, filtbank:out control, SELT:out control, contclear 'leaving lines latched to filter bank
    out port, 0
  'step 2. Command the PLL without changing the PDM
    pdmcommand = phaarray(thisstep,0)*64 'do not disturb PDM state, this may be used during Spur Test
    out port, pdmcommand + levalue  'levalues: PLL1=1, PLL2=16, PLL3=4
    out control, INIT
    out port, pdmcommand
    out control, contclear  'leaving lines latched, and unchanged, to PDM
    out port, 0
    return 'to [CommandPLL]
```

This function takes a set of bits N0..N23, a datavalue that corresponds to the bit used for the specific IO line and a fixed value filtbank. It clocks the data sequentially.

The USB equivalent is this

```
[CommandPLLslimUSB] 'USB:01-08-2010
    if USBdevice = 0 then return
    CALLDLL #USB, "UsbMSADeviceWriteInt64MsbFirst", USBdevice as long, 161 as short, Int64N as ptr, 24 as short, 1 as short, _
                                                    filtbank as short, datavalue as short, result as boolean  'USB:11-08-2010
    pdmcommand = phaarray(thisstep,0)*64 'do not disturb PDM state, this may be used during Spur Test
    USBwrbuf$ = "A30200"+ToHex$(pdmcommand + levalue)+ToHex$(pdmcommand)
    CALLDLL #USB, "UsbMSADeviceWriteString", USBdevice as long, USBwrbuf$ as ptr, 5 as short, result as boolean
    Return
```

This function shows the main integer output function, `UsbMSADeviceWriteInt64MsbFirst` as well as another method used, `UsbMSADeviceWriteString`. Let's look at the first one.

```
    CALLDLL #USB, "UsbMSADeviceWriteInt64MsbFirst", USBdevice as long, 161 as short, Int64N as ptr, 24 as short, 1 as short, _
                                                    filtbank as short, datavalue as short, result as boolean  'USB:11-08-2010
```

All the output methods all get to the USB controller via the same interface but the DLL provides several ways to get these, and these are two of them.

In this first one, we have a simple but flexible helper in the DLL that handles the things Liberty does badly but keeps the flexibility in the MSA Basic program for making changes. The parameters are as follows

USBdevice as long,	This is the handle to the USB device – we need this always
161 as short,	This is the hex command code 0xA1 which tells the USB controller that this is a write to port B
Int64N as ptr,	This is a 64 bit integer formed as two unsigned longs in a structure. We'll cover this in a moment
24 as short,	This is the number of bits in the 64 bit integer to clock out
1 as short, _	This is the clock bit – toggle port B bit 0 to clock the data
filtbank as short,	This is the fixed value for the port data
datavalue as short,	This is the variable bit to add to the fixed part for every '1' bit in the 64 bit integer
result as boolean	And the mandatory true / false success flag

So that 64 bit structure. It looks like this

```
struct Int64N, msLong as ulong, lsLong as ulong
```

and the basic program just bungs the data value in instead of turning it into bits in N0-N23 for example as shown below. If we are using the USB interface, put the value required into Int64, otherwise set the N0..23 values. This is the simplest example, some get more complex.

```
'[CreateIFNbuffer2350]'needed:nothing,since IF section is turned off(N22=1)
  if cb = 3 then
      Int64N.lsLong.struct = 4472833 ' 4472833 = 0x444001 = 10001000100000000000001
      Int64N.msLong.struct = 0
  else
      N23=0     'IF counter reset, 0=normal operation
      N22=1     'Power down mode for IF section, 1=powered down, 0=powered up
      N21=0     'PWN Mode,  0=async  1=syncro
      N20=0     'Fastlock, 0=CMOS outputs enabled 1= fastlock mode
      N19=0     'test bit, leave at 0
      N18=1     'OUT 0,  1
      N17=0     'OUT 1,  0
```

```
    N16=0     'IF N Bcounter 12 Bits MSB bit 11
    N15=0     'IF N Bcounter, bit 10, '512 = 0010 0000 0000
    N14=1     'IF N Bcounter, bit 9
    N13=0     'IF N Bcounter, bit 8
    N12=0     'IF N Bcounter, bit 7
    N11=0     'IF N Bcounter, bit 6
    N10=0     'IF N Bcounter, bit 5
    N9=0      'IF N Bcounter, bit 4
    N8=0      'IF N Bcounter, bit 3
    N7=0      'IF N Bcounter, bit 2
    N6=0      'IF N Bcounter, bit 1
    N5=0      'IF N Bcounter, 12 Bits, LSB bit 0
    N4=0      'bit 2, IF N Acounter 3 Bits MSB
    N3=0      'bit 1, 0 = 000 thru 7 = 111
    N2=0      'bit 0, IF N Acounter 3 Bits LSB
    N1=0      '2350 IF_N register, 2 bits, must be 0
    N0=1      '2350 IF_N register, 2 bits, must be 1
end if
gosub [CommandPLL]'needs:N23-N0,control,Jcontrol,port,contclear,LEPLL ; commands N23-N0,old ControlBoard ver111
```

Ok, that's the integer method. Remember the string one in the above example?

```
pdmcommand = phaarray(thisstep,0)*64 'do not disturb PDM state, this may be used during Spur Test
USBwrbuf$ = "A30200"+ToHex$(pdmcommand + levalue)+ToHex$(pdmcommand)
CALLDLL #USB, "UsbMSADeviceWriteString", USBdevice as long, USBwrbuf$ as ptr, 5 as short, result as boolean
```

This is a simple interface where we create a hex string with the USB controller command and ask the DLL to just send it direct. Command A3 is "write to port D", 02 is the number of bytes to write (2) the clock line is 00 so the interface will not do autoclocking. Thre are then two bytes added which are the values in the ToHex$() functions. These two bytes are written one after the other to port D by the USB controller.

Those are the easy interfaces. However there is one more complex one. In fact it is unnecessary for two reasons. The above functions could be used but then Liberty basic would make it really very slow as it can't handle bytes efficiently. Secondly it would be possible to rewrite the code to just calculate the parameters for each step in the scan on the fly but this would need to be removed from the Basic code and done in the DLL for speed and then we would start to lose flexibility in the basic code, so for both these reasons a method has been used that is very similar to the existing code design.

The current code creates a series of arrays such as dim PLL1array(800,48), so we create a memory block equivalent for __int64 values

```
hSPLL1Array = GlobalAlloc( DeviceArrayBlockSize ) 'USB:06-08-2010
ptrSPLL1Array = GlobalLock( hSPLL1Array ) 'USB:06-08-2010
```

The first variable is a handle that is used for management, the second is a pointer to a block in memory. We can pass this to the DLL and the DLL will store the binary data in it, so for example in the Basic code we find this (comments removed for ease of visualising the code). The first part of the code up to the line before the w0 setting calculates the desired data – an integer value 'base'. We then store it in 4 integers w0..w4 for the parallel interface (this could also be skipped for USB but has been left in for diagnostics). Now the USB code. Put the value of base into a 64 bit integer so we can store it. Then the else clause – for the parallel interface serial control chop it into individual bits sw0..sw39. This is why we use a 64 bit integer – we have more than 32 bits of data even if lots are zero.

```
[CreateBaseForDDSarray]'needed:ddsoutput,ddsclock ; creates: base,sw0thrusw30,w0thruw4
        fullbase=(ddsoutput*2^32/ddsclock) 'decimal number, including fraction
        if ddsoutput >= ddsclock/2 then
                beep:message$="Error, ddsoutput > .5 ddsclock" : call PrintMessage :goto [Halted] 'ver114-4e
        end if
         base = int(fullbase) 'rounded down to whole number
        if fullbase - base >= .5 then base = base + 1 'rounded to nearest whole number
        w0= 0 'a "1" here will activate the x4 internal multiplier, but not recommended
        w1= int(base/2^24)  'w1 thru w4 converts decimal base code to 4 words, each are 8 bit binary
        w2= int((base-(w1*2^24))/2^16)
        w3= int((base-(w1*2^24)-(w2*2^16))/2^8)
        w4= int(base-(w1*2^24)-(w2*2^16)-(w3*2^8))
        if cb = 3 then
                Int64SW.msLong.struct = 0
                Int64SW.lsLong.struct = int( base )
        else
                'Create Serial Bits'needed:base ; creates serial word bits; sw0 thru sw39
                 b0 = int(base/2):sw0 = base - 2*b0  'LSB, Freq-b0.  sw is serial word bit
                 b1 = int(b0/2):sw1 = b0 - 2*b1:b2 = int(b1/2):sw2 = b1 - 2*b2
                 b3 = int(b2/2):sw3 = b2 - 2*b3:b4 = int(b3/2):sw4 = b3 - 2*b4
                 b5 = int(b4/2):sw5 = b4 - 2*b5:b6 = int(b5/2):sw6 = b5 - 2*b6
                 b7 = int(b6/2):sw7 = b6 - 2*b7:b8 = int(b7/2):sw8 = b7 - 2*b8
                 b9 = int(b8/2):sw9 = b8 - 2*b9:b10 = int(b9/2):sw10 = b9 - 2*b10
                 b11 = int(b10/2):sw11 = b10 - 2*b11:b12 = int(b11/2):sw12 = b11 - 2*b12
                 b13 = int(b12/2):sw13 = b12 - 2*b13:b14 = int(b13/2):sw14 = b13 - 2*b14
                 b15 = int(b14/2):sw15 = b14 - 2*b15:b16 = int(b15/2):sw16 = b15 - 2*b16
                 b17 = int(b16/2):sw17 = b16 - 2*b17:b18 = int(b17/2):sw18 = b17 - 2*b18
                 b19 = int(b18/2):sw19 = b18 - 2*b19:b20 = int(b19/2):sw20 = b19 - 2*b20
```

```
            b21 = int(b20/2):sw21 = b20 - 2*b21:b22 = int(b21/2):sw22 = b21 - 2*b22
            b23 = int(b22/2):sw23 = b22 - 2*b23:b24 = int(b23/2):sw24 = b23 - 2*b24
            b25 = int(b24/2):sw25 = b24 - 2*b25:b26 = int(b25/2):sw26 = b25 - 2*b26
            b27 = int(b26/2):sw27 = b26 - 2*b27:b28 = int(b27/2):sw28 = b27 - 2*b28
            b29 = int(b28/2):sw29 = b28 - 2*b29:b30 = int(b29/2):sw30 = b29 - 2*b30
            b31 = int(b30/2):sw31 = b30 - 2*b31  'MSB, Freq-b31
            sw32 = 0 'x4 multiplier, 1=enable, but not recommended
            sw33 = 0 'control bit
            sw34 = 0 'power down bit
            sw35 = 0 'phase data
            sw36 = 0 'phase data
            sw37 = 0 'phase data
            sw38 = 0 'phase data
            sw39 = 0 'phase data
        end if
    return
```

Now the data is stored. For the parallel port, it is stored in the two dimensional DDS1array. For the USB interface we tell the DLL to store it in the memory block we allocated earlier (preSDDS1Array.

```
[FillDDS1array]
    if cb = 3 then
        if USBdevice <> 0 then CALLDLL #USB, "UsbMSADevicePopulateDDSArray", USBdevice as long, ptrSDDS1Array as ulong, _
                                                Int64SW as ptr, thisstep as short, result as boolean
    else
        DDS1array(thisstep,0) = sw0:DDS1array(thisstep,1) = sw1
        DDS1array(thisstep,2) = sw2:DDS1array(thisstep,3) = sw3
        DDS1array(thisstep,4) = sw4:DDS1array(thisstep,5) = sw5
        DDS1array(thisstep,6) = sw6:DDS1array(thisstep,7) = sw7
        DDS1array(thisstep,8) = sw8:DDS1array(thisstep,9) = sw9
        DDS1array(thisstep,10) = sw10:DDS1array(thisstep,11) = sw11
        DDS1array(thisstep,12) = sw12:DDS1array(thisstep,13) = sw13
        DDS1array(thisstep,14) = sw14:DDS1array(thisstep,15) = sw15
        DDS1array(thisstep,16) = sw16:DDS1array(thisstep,17) = sw17
        DDS1array(thisstep,18) = sw18:DDS1array(thisstep,19) = sw19
        DDS1array(thisstep,20) = sw20:DDS1array(thisstep,21) = sw21
        DDS1array(thisstep,22) = sw22:DDS1array(thisstep,23) = sw23
        DDS1array(thisstep,24) = sw24:DDS1array(thisstep,25) = sw25
        DDS1array(thisstep,26) = sw26:DDS1array(thisstep,27) = sw27
        DDS1array(thisstep,28) = sw28:DDS1array(thisstep,29) = sw29
        DDS1array(thisstep,30) = sw30:DDS1array(thisstep,31) = sw31
        DDS1array(thisstep,32) = sw32:DDS1array(thisstep,33) = sw33
```

```
        DDS1array(thisstep,34) = sw34:DDS1array(thisstep,35) = sw35
        DDS1array(thisstep,36) = sw36:DDS1array(thisstep,37) = sw37
        DDS1array(thisstep,38) = sw38:DDS1array(thisstep,39) = sw39
    end if
    DDS1array(thisstep,40) = w0
    DDS1array(thisstep,41) = w1
    DDS1array(thisstep,42) = w2
    DDS1array(thisstep,43) = w3
    DDS1array(thisstep,44) = w4
    DDS1array(thisstep,45) = base
    DDS1array(thisstep,46) = base*ddsclock/2^32
    return
```

Now the final bit – how is it used. In the parallel port code the data in each of the arrays like DDS1Array is combined into a per step overall array thus whereas in the Usb version the DLL does the same thing on the arrays of integers.

```
[CreateCmdAllArray] 'for SLIM CB only 'ver-31b
    rememberthisstep = thisstep 'remember where we were when entering this subroutine
    if cb <> 3 then
        for thisstep = 0 to steps
            for clmn = 0 to 15
                cmdallarray(thisstep,clmn) = DDS1array(thisstep,clmn)*4 + DDS3array(thisstep,clmn)*16
            next clmn
            for clmn = 16 to 39
                cmdallarray(thisstep,clmn) = PLL1array(thisstep,clmn-16)*2 + DDS1array(thisstep,clmn)*4 + _
                                             PLL3array(thisstep,clmn-16)*8 + DDS3array(thisstep,clmn)*16
            next clmn
        next thisstep
    else
        if USBdevice <> 0 then CALLDLL #USB, "UsbMSADevicePopulateAllArray", USBdevice as long, steps as short, 40 as short, _
                            0 as long, ptrSPLL1Array as long, ptrSDDS1Array as long, ptrSPLL3Array as long, _
                            ptrSDDS3Array as long, 0 as long, 0 as long, 0 as long, _
                            result as boolean 'USB:11-08-2010
    end if
    thisstep = rememberthisstep
    return
```

By the way, that function that put the data into the array? There are two forms of it – the other one bit reverses the data as it is put in – this is because the PLL data is clocked in the opposite direction.

And the final part – commanding all slims at once. For parallel port it looks like this (stripped of comments)

```
    [CommandAllSlims]'for SLIM Control and SLIM modules. Old PDM and old Filt Bank can be used 'ver111-31c
    for clmn = 0 to 39  'ver113-3c
        a= cmdallarray(thisstep,clmn)+ filtbank
        out port, a : out control, SELT:out control, contclear 'a is the data, without clock
        out port, a+1:out control, SELT:out control, contclear 'a+1 is data, plus clock
    next clmn
    out port, filtbank 'remove data, leaving filtbank data to filter bank.
    out control, SELT:out control, contclear 'disable buffer. filtbank signals will be latched to filter bank assembly
    pdmcmd = phaarray(thisstep,0)*64 'ver111-39d
    out port, le1 + fqud1 + le3 + fqud3 + pdmcmd 'present data to buffer input'ver111-39d
    out control, INIT: out control, contclear  'latch the buffer, moving the signals to the 5 modules'ver113-2a
    out port, pdmcmd + 32 'remove LEs and Fquds, leaving PDM data, but add a latch signal P2D5 for old PDM if used.'ver111-39d
    out control, INIT: out control, contclear  'sends latch signal to old PDM'ver113-2a
    out port, pdmcmd  'remove the added latch signal to PDM, leaving just the PDM's static data'ver111-39d
    out control, INIT: out control, contclear  'ver113-2a
    out port, 0    'bring all Data lines low. PDM data remains static
    lastpdmstate=phaarray(thisstep,0)   'ver114-6c
    return 'to [CommandThisStep]
```

and for USB this

```
[CommandAllSlimsUSB] 'USB:01-08-2010
    if USBdevice = 0 then return
    CALLDLL #USB, "UsbMSADeviceAllSlims", USBdevice as long, thisstep as short, filtbank as short, result as boolean 'USB:11-08-2010
    pdmcmd = phaarray(thisstep,0)*64 'ver111-39d
    USBwrbuf$ = "A30300"+ToHex$(le1 + fqud1 + le3 + fqud3 + pdmcmd)+ToHex$(pdmcmd + 32)+ToHex$(pdmcmd)
    CALLDLL #USB, "UsbMSADeviceWriteString", USBdevice as long, USBwrbuf$ as ptr, 6 as short, result as boolean
    lastpdmstate=phaarray(thisstep,0)   'ver114-6c
    return
```

Ok, the operation of the DLL will be the next document…..

73's
Dave
G8KBB