Cal Sensors

ADIC LIBRARY DOCUMENTATION

User documentation for ADICLib_256R4.dll version 4.0.2.0

Cal Sensors

8/13/2008

TABLE OF CONTENTS

ARRAYADD

Description:

Adds two single dimension, single precision data arrays, array1 and array2, of equal length together, storing the resulting array in array1.

CALLING FORMAT:

ArrayAdd(array1, array2, #arrayelements)

VISUAL C++ DECLARE:

extern "C" __declspec(dllimport) long WINAPI ArrayAdd(float array1[], float array2[], long dataSize);

VISUAL C++ CALL:

{return value} = ArrayAdd(arrayname1, arrayname2, datasize);

VISUAL BASIC DECLARE:

Public Declare Function ArrayAdd Lib "ADICLib_256.dll" (ByRef Array1 As Single, _ ByRef array2 As Single, ByVal dataSize As Long) As Long

VISUAL BASIC CALL:

{return value} = ArrayAdd(arrayname1(0), arrayname2(0), datasize)

LABVIEW FUNCTION PROTOTYPE:

long ArrayAdd(float *arg1, float *arg2, long arg3);

PARAMETERS:

arrayname1 = array to be added to and the addition result array returned

arrayname2 = array to add to arrayname1

datasize = the number of elements in the single dimension array (128 or 256)

{return value} = Function evaluation status, A 1 indicates successful operation, a 0 indicates there was an error.

ALGORITHMN:

array1 = array1 + array2

PROGRAMMING NOTES:

1) The __stdcall calling convention is used for the function call.

2) Function returns no value (void), the result is passed back to the calling procedure in array1.

3) Arrays are always passed by reference. To pass an entire array by reference in Visual BASIC to a DLL written in C, only the first element of the array is passed by reference. The DLL will have full access to the array since arrays are stored sequentially in memory.

ARRAYPERCENTDIFF

DESCRIPTION:

Finds the percent difference of one single dimension, single precision array, array1, to a another single dimension, single precision reference array, array2. The resulting data in percent is returned in array1.

CALLING FORMAT:

ArrayPercentDiff(array1, array2, #arrayelements)

VISUAL C++ DECLARE:

extern "C" __declspec(dllimport) void WINAPI ArrayPercentDiff(float array1[], float array2[], long dataSize);

VISUAL C++ CALL:

{return value} = ArrayPercentDiff(arrayname1, arrayname2, datasize);

VISUAL BASIC DECLARE:

Public Declare Function ArrayPercentDiff Lib "ADICLib_256.dll" (ByRef Array1 As Single, _ ByRef array2 As Single, ByVal dataSize As Long) As Long

VISUAL BASIC CALL:

{return value} = ArrayPercentDiff(arrayname1(0), arrayname2(0), datasize)

LABVIEW FUNCTION PROTOTYPE:

long ArrayAdd(float *arg1, float *arg2, long arg3);

PARAMETERS:

arrayname1 = array to compute the percent difference from arrayname2

arrayname2 = the reference array

datasize = the number of elements in the single dimension array (128 or 256))

{return value} = Function evaluation status, A 1 indicates successful operation, a 0 indicates there was an error.

ALGORITHM:

array1 = 100 * (array1 - array2) / array2)

If an element of array2 = 0, it is arbitrarily set to 1E-7 prior to division to avoid divide by zero errors.

PROGRAMMING NOTES:

1) The __stdcall calling convention is used for the function call.

2) Function returns no value (void), the result is passed back to the calling procedure in array1.

3) Arrays are always passed by reference. To pass an entire array by reference in Visual BASIC to a DLL written in C, only the first element of the array is passed by reference. The DLL will have full access to the array since arrays are stored sequentially in memory.

ARRAYSUBTRACT

DESCRIPTION:

Subtracts two single dimension, single precision data arrays, array1 and array2, of equal length

from one another, storing the resulting array in array1.

CALLING FORMAT:

ArraySubtract(array1, array2, #arrayelements)

VISUAL C++ DECLARE:

```
extern "C" __declspec(dllimport) long WINAPI ArraySubtract(float array1[], float array2[], long dataSize);
```

VISUAL C++ CALL:

```
{return value} = ArraySubtract(arrayname1, arrayname2, datasize);
```

VISUAL BASIC DECLARE:

```
Public Declare Function ArraySubtract Lib "ADICLib_256.dll" (ByRef Array1 As Single, _
  ByRef array2 As Single, ByVal dataSize As Long) As Long
```

VISUAL BASIC CALL:

```
{return value} = ArraySubtract(arrayname1(0), arrayname2(0), datasize)
```

LABVIEW FUNCTION PROTOTYPE:

```
long ArrayAdd(float *arg1, float *arg2, long arg3);
```

PARAMETERS:

arrayname1 = array to be subtracted from and the subtraction result array returned

arrayname2 = array to subtract from arrayname1

datasize = the number of elements in the single dimension array (128 or 256))

{return value} = Function evaluation status, A 1 indicates successful operation, a 0 indicates there was an error.

ALGORITHMN:

array1 = array1 - array2

PROGRAMMING NOTES:

1) The __stdcall calling convention is used for the function call.

2) Function returns no value (void), the result is passed back to the calling procedure in array1.

3) Arrays are always passed by reference. To pass an entire array by reference in Visual BASIC to a DLL written in C, only the first element of the array is passed by reference. The DLL will have full access to the array since arrays are stored sequentially in memory.

CLOSEDEVICE

DESCRIPTION:

Closes communication with the mux controller board. This command should be executed prior to the top level software application exiting to cleanly close communications with the board.

CALLING FORMAT:

CloseDevice(deviceHandle)

VISUAL C++ DECLARE:

extern "C" __declspec(dllimport) long WINAPI CloseDevice(FT_HANDLE lngHandle);

VISUAL C++ CALL:

{return value} = CloseDevice(deviceHandle);

VISUAL BASIC DECLARE:

Public Declare Function CloseDevice Lib "ADICLib_256.dll" (ByVal lngHandle As Long) As Long

VISUAL BASIC CALL:

{return value} = CloseDevice(deviceHandle)

LABVIEW FUNCTION PROTOTYPE:

long CloseDevice(long arg1);

PARAMETERS:

deviceHandle = device handle address of type FT_HANDLE (see header file) returned from the **GetDeviceHandle** function.

{return value} = Passed status variable from the function call. If the function completed successfully status will return with a value of 1, otherwise an error code is returned, see the status error definition help topic.

PROGRAMMING NOTES:

1) The __stdcall calling convention is used for the function call.

DOCALIBRATE

DESCRIPTION:

Calibrates the array

CALLING FORMAT:

DoCalibrate(deviceHandle, coeffArray, arraySetVal, gskimVal, dacVhVal, dacVlVal, bPixMapArray, mxSize)

VISUAL C++ DECLARE:

extern "C" __declspec(dllimport) long WINAPI DoCalibrate(FT_HANDLE lngHandle, long coeff[], float arraySetVal, long* gskimVal, long* dacVhVal, long* dacVlVal, long bPixMap[], long mxSize,);

VISUAL C++ CALL:

{return value} = DoCalibrate(deviceHandle, coeffArray, arraySetVal, gskimVal, dacVhVal, dacVlVal, bPixMapArray, mxSize);

VISUAL BASIC DECLARE:

Public Declare Function DoCalibrate Lib "ADICLib_256.dll" (ByVal lngHandle As Long, ByRef coeff As Long, _ ByVal arraySetVal As Single, ByRef gskimVal As Long, ByRef dacVhVal As Long, _ ByRef dacVlVal As Long, ByRef bPixMap As Long, ByVal mxSize As Long) As Long

VISUAL BASIC CALL:

{return value} = DoCalibrate(deviceHandle, coeffArray(0), arraySetVal, gskimVal, dacVhVal, dacVlVal, _ bPixMapArray(0), mxSize)

LABVIEW FUNCTION PROTOTYPE:

long DoCalibrate(long arg1, long *arg2, float arg3, long *arg4, long *arg5, long *arg6, long *arg7, long arg8);

PARAMETERS:

deviceHandle = device handle address of type FT_HANDLE (see header file) returned from the **GetDeviceHandle** function.

coeffArray = Correction coefficient array returned that sets the per pixel DAC values for correction

arraySetVal = Target correction value for the array. This should not be set to either of the supply rails. Typical value is 1.2

gskimVal = Returned value for the global skim value set automatically by the DoCalibrate routine. Meaningless if the gskimUseVal has been set to anything other than -1 by the **SetGlobalSkimVal** function

dacVhVal = Upper DAC reference value returned from the DoCalibrate procedure if dacVhUseVal is set at -1 by the SetDacReferences procedure. Meaningless otherwise.

dacVlVal = Lower DAC reference value returned from the DoCalibrate procedure if dacVlUseVal is set at -1 by the SetDacReferences procedure. Meaningless otherwise.

bPixMapArray = Bad pixel array map. Bad pixels are marked with a value of "1", good pixels are marked as "0".

mxSize = Array size. Either 128 or 256.

{return value} = Passed status variable from the function call. If the function completed successfully status will return with a value of 1, otherwise an error code is returned, see the status error definition help topic.

PROGRAMMING NOTES:

1) The __stdcall calling convention is used for the function call.

2) Function returns a status variable as a long datatype, updated values for gskimVal, dacVhVal, and dacVlVal are returned in the passed variables.

3) Arrays are always passed by reference. To pass an entire array by reference in Visual BASIC to a DLL written in C, only the first element of the array is passed by reference. The DLL will have full access to the array since arrays are stored sequentially in memory.

FLUX_P

DESCRIPTION:

Integrates Planck's blackbody equation and computes the bandpass blackbody flux for a given temperature in photons/(sec-cm^2-sr).

CALLING FORMAT:

ReturnValue = Flux_P(lambdaLow, lambdaHigh, temperature)

VISUAL C++ DECLARE:

extern "C" __declspec(dllimport) double WINAPI Flux_P(double lambdaL, double lambdaH, double temperature);

VISUAL C++ CALL:

ReturnValue = Flux_P(lambdaLow, lambdaHigh, temperature);

VISUAL BASIC DECLARE:

Public Declare Function Flux_P Lib "ADICLib_256.dll" (ByVal lambda_lo As Double, _ ByVal lambda_hi As Double, ByVal temperature As Double) As Double

VISUAL BASIC CALL:

ReturnValue = Flux_P(lambdaLow, lambdaHigh, temperature)

LABVIEW FUNCTION PROTOTYPE:

double Flux_P(double arg1, double arg2, double arg3);

PARAMETERS:

lambdaLow = Lower bandpass wavelength in microns

lambdaHigh = Upper bandpass wavelength in microns

temperature = Temperature in Kelvin

ALGORITHMN:

Performs a trapizoidal integration of Planck's equation using a delta lambda of 100 picometers per micron.

PROGRAMMING NOTES:

1) The __stdcall calling convention is used for the function call.

2) Function returns a double precision value

FLUX_W

DESCRIPTION:

Integrates Planck's blackbody equation and computes the bandpass blackbody flux for a given temperature in Watts/(cm^2-sr).

CALLING FORMAT:

ReturnValue = Flux_W(lambdaLow, lambdaHigh, temperature)

VISUAL C++ DECLARE:

extern "C" __declspec(dllimport) double WINAPI Flux_W(double lambdaL, double lambdaH, double temperature);

VISUAL C++ CALL:

ReturnValue = Flux_W(lambdaLow, lambdaHigh, temperature);

VISUAL BASIC DECLARE:

Public Declare Function Flux_W Lib "ADICLib_256.dll" (ByVal lambda_lo As Double, _ ByVal lambda_hi As Double, ByVal temperature As Double) As Double

VISUAL BASIC CALL:

ReturnValue = Flux_W(lambdaLow, lambdaHigh, temperature)

LABVIEW FUNCTION PROTOTYPE:

double Flux_W(double arg1, double arg2, double arg3);

PARAMETERS:

lambdaLow = Lower bandpass wavelength in microns

lambdaHigh = Upper bandpass wavelength in microns

temperature = Temperature in Kelvin

ALGORITHMN:

Performs a trapizoidal integration of Planck's equation using a delta lambda of 100 picometers per micron.

PROGRAMMING NOTES:

1) The __stdcall calling convention is used for the function call.

2) Function returns a double precision value

GETBOARD_HANDLE

DESCRIPTION:

Returns a PVOID type number which represents the device handle of the Mux Controller Board for the given board index number, opens and sets up all communication settings to the mux controller board "boardnumber". The number returned is the address number to be passed to all other API functions requiring a device handle when communicating to that particular board. The GetBoard_Handle function should be the first function called for any communications to the hardware.

CALLING FORMAT:

boardHandle = GetBoard_Handle(boardnumber)

VISUAL C++ DECLARE:

extern "C" __declspec(dllimport) PVOID WINAPI GetBoard_Handle(long boardnumber);

VISUAL C++ CALL:

boardHandle = GetBoard_Handle(boardnumber);

VISUAL BASIC DECLARE:

Public Declare Function GetBoard_Handle Lib "ADICLib_256.dll" (ByVal boardnumber As Long) As Long

VISUAL BASIC CALL:

boardHandle = GetBoard_Handle(boardnumber)

LABVIEW FUNCTION PROTOTYPE:

long GetBoard_Handle(long arg1);

PARAMETERS:

boardHandle = device handle address of type FT_HANDLE (see header file) returned from the GetBoard_Handle function.

boardnumber = index number reference for each controller board attached to the computer where boardnumber is a number from 1 to NumberOfBoards, where NumberOfBoards is the number returned back from the GetNumBoards functions (see GetNumBoards function). To get the boardhandle and open the first board attached to the system, pass boardnumber = 1 to the GetBoard_Handle function, to get the boardhandle and open the second board attached to the system pass boardnumber = 2 to the GetBoard_Handle function, and so on up to the maximum number of currently attached controller boards to the system determined from the GetNumBoards function. Any number outside the range 1 to NumberOfBoards will produce an error.

PROGRAMMING NOTES:

1) The __stdcall calling convention is used for the function call.

GETDATA

Note: It is recommended that the **GetData2** command be used when operating 256 element arrays using the RevG multiplexers

DESCRIPTION:

Retrieves an array of data values (16 bit conversions) from all channels (pixels) of the detector/multiplexer array. The format of the data returned from the **GetData** procedure is a single precision, single dimension data array of length 128 or 256 elements depending on array size. The data format is in volts with a range of 0v to Vmax, where Vmax is typically ~ 5.00 Volts when the conversion factor is at it's default value of 13107 (see SetConversionRef procedure).

CALLING FORMAT:

GetData(deviceHandle, numAvg, arrayOrderFlag, dataSize, MxData, bPixMap)

VISUAL C++ DECLARE:

extern "C" __declspec(dllimport) long WINAPI GetData(FT_HANDLE lngHandle, long numAvg, long arrayOrderFlag, long dataSize, float MxData[], long bPixMap[]);

VISUAL C++ CALL:

{return value} = GetData(deviceHandle, numAvg, arrayOrderFlag, dataSize, MxData, bPixMap);

VISUAL BASIC DECLARE:

Public Declare Function GetData Lib "ADICLib_256.dll" (ByVal lngHandle As Long, ByVal numAvg As Long, _ ByVal ArrayOrderFlag As Long, ByVal dataSize As Long, ByRef MxData As Single, _ ByRef bPixMap As Long, ByRef status As Long) As Long

VISUAL BASIC CALL:

{return value} = GetData(deviceHandle, numAvg, arrayOrderFlag, dataSize, MxData(0), bPixMap(0))

LABVIEW FUNCTION PROTOTYPE:

long GetData(long arg1, long arg2, long arg3, long arg4, float *arg5, long *arg6);

PARAMETERS:

deviceHandle = device handle address of type FT_HANDLE (see header file) returned from the **GetDeviceHandle** function.

numAvg = This value is included for legacy software support. It is ignored when using the R4 version of the controller board.

arrayOrderFlag = Flag which tells the GetData procedure whether to return the data in either mux die order or detector array order. A "0" is defined as mux order, a "1" is defined as array order. This paramater is only used when operating Rev F multiplexers, for Rev G multiplexers this parameter is ignored.

datasize = the number of elements in the single dimension array (128 or 256)

MxData = Single precision data array of detector pixel values returned by the GetData procedure.

bPixMapArray = Bad pixel array map. Bad pixels are marked with a value of "1", good pixels are marked as "0".

{return value} = Passed status variable from the function call. If the function completed successfully status will return with a value of 1, otherwise an error code is returned, see the status error definition help topic.

PROGRAMMING NOTES:

1) The __stdcall calling convention is used for the function call.

2) Function returns a status variable as a long datatype, detector data is returned in the passed variable MxData

3) Arrays are always passed by reference. To pass an entire array by reference in Visual BASIC to a DLL written in C, only the first element of the array is passed by reference. The DLL will have full access to the array since arrays are stored sequentially in memory.

GETDATA2

Note: This command can only be used with 256 element arrays using the RevG multiplexers. The type of mux is indicated by the muxtype bit returned by **GetQuery3**. A muxtype = 1 indicates a RevG multiplexer

DESCRIPTION:

Retrieves an windowed array of data values (16 bit conversions) from all channels (pixels) of the detector/multiplexer array. The format of the data returned from the **GetData** procedure is a single precision, single dimension data array of variable length from 2 to 256 elements depending on array window size. The data format is in volts with a range of 0v to Vmax, where Vmax is typically ~ 5.00 Volts when the conversion factor is at it's default value of 13107 (see SetConversionRef procedure).

CALLING FORMAT:

GetData2(deviceHandle, LAdd, RAdd, RoDir, MxData, bPixMap)

VISUAL C++ DECLARE:

extern "C" __declspec(dllimport) long WINAPI GetData(FT_HANDLE lngHandle, long* LAdd, long* RAdd, long* RoDir, float MxData[], long bPixMap[]);

VISUAL C++ CALL:

{return value} = GetData(deviceHandle, LAdd, RAdd, RoDir, MxData, bPixMap);

VISUAL BASIC DECLARE:

Public Declare Function GetData Lib "ADICLib_256.dll" (ByVal lngHandle As Long, ByRef LAdd As Long, _ ByRef RAdd As Long, ByRef RoDir As Long, ByRef MxData As Single, _ ByRef bPixMap As Long) As Long

VISUAL BASIC CALL:

{return value} = GetData(deviceHandle, LAdd, RAdd, RoDir, MxData(0), bPixMap(0))

LABVIEW FUNCTION PROTOTYPE:

long GetData(long arg1, long *arg2, long *arg3, long *arg4, float *arg5, long *arg6);

PARAMETERS:

deviceHandle = device handle address of type FT_HANDLE (see header file) returned from the **GetDeviceHandle** function.

LAdd = Passed value sets (and returns) the start pixel of the readout window, if the array is reading out left to right. If the readout is right to left this value sets the end pixel. Regardless of the direction of readout the LAdd (or Left Address) is the number of pixels to skip on the left side of the array. The valid range is 0 to 127, with 0 being the value to readout the entire left side of the array, and 127 reading out only a single pixel from the left side (the 128th pixel). On return the value represents the value returned by the array. This value should match the value sent, but if it does not it, is a clear indicator that the software and the hardware no longer have the same settings.

RAdd = Passed value sets (and returns) the end pixel of the readout window, if the array is reading out left to right. If the readout is right to left this value sets the start pixel. Regardless of the direction of readout the RAdd (or Right Address) is the number of pixels to skip on the right side of the array. The valid range is 0 to 127, with 0 being the value to readout the entire right side of the array, and 127 reading out only a single pixel from the right side (the 129th pixel). On return the value represents the value returned by the array. This value should match the value sent, but if it does not, it is a clear indicator that the software and the hardware no longer have the same settings.

RoDir = Readout direction control flag. A low sets a Left to Right readout and a high is the reverse direction The returned value is the direction control setting inside the hardware, it should always match the value sent. If it does not, it is a clear indicator that the software and the hardware no longer have the same settings.

MxData = Single precision data array of detector pixel values returned by the **GetData** procedure.

bPixMapArray = Bad pixel array map. Bad pixels are marked with a value of "1", good pixels are marked as "0".

{return value} = Passed status variable from the function call. If the function completed successfully status will return with a value of 1, otherwise an error code is returned, see the status error definition help topic.

PROGRAMMING NOTES:

1) The __stdcall calling convention is used for the function call.

2) Function returns a status variable as a long datatype, detector data is returned in the passed variable MxData array

3) Arrays are always passed by reference. To pass an entire array by reference in Visual BASIC to a DLL written in C, only the first element of the array is passed by reference. The DLL will have full access to the array since arrays are stored sequentially in memory.

GETDEVICEHANDLE

DESCRIPTION:

Returns a PVOID type number which represents the device handle of the Mux Controller Board and sets up all communication settings to the mux controller board. The number returned is the address number to be passed to all other API functions requiring a device handle. The GetDeviceHandle function should be the first function called for any communications to the hardware.

CALLING FORMAT:

devHandle = GetDeviceHandle()

VISUAL C++ DECLARE:

extern "C" __declspec(dllimport) PVOID WINAPI GetDeviceHandle(void);

VISUAL C++ CALL:

devHandle = GetDeviceHandle();

VISUAL BASIC DECLARE:

Public Declare Function GetDeviceHandle Lib "ADICLib_256.dll" () As Long

VISUAL BASIC CALL:

devHandle = GetDeviceHandle()

LABVIEW FUNCTION PROTOTYPE:

long GetDeviceHandle(void);

PARAMETERS:

deviceHandle = device handle address of type FT_HANDLE (see header file) returned from the **GetDeviceHandle** function.

PROGRAMMING NOTES:

1) The __stdcall calling convention is used for the function call.

GETNUMBOARDS

DESCRIPTION:

Returns the number, of data type long, of Multiplexer Controller boards currently attached to the computer system.

CALLING FORMAT:

NumberOfBoards = GetNumBoards()

VISUAL C++ DECLARE:

```
extern "C" __declspec(dllimport) long WINAPI GetNumBoards(void);
```

VISUAL C++ CALL:

```
NumberOfBoards = GetNumBoards();
```

VISUAL BASIC DECLARE:

```
Public Declare Function GetNumBoards Lib "ADICLib_256.dll" () As Long
```

VISUAL BASIC CALL:

```
NumberOfBoards = GetNumBoards()
```

LABVIEW FUNCTION PROTOTYPE:

```
long GetNumBoards(void);
```

PARAMETERS:

NumberOfBoards = The number of Multiplexer Controller boards currently powered up and attached to the computer system returned by the GetNumBoards function. There is no fixed limit in the API as to the number of boards allowed to be attached to the system and is only limited by the number of available USB interfaces on the computer system and the number of unique controller boards handled by the top level user developed application.

PROGRAMMING NOTES:

1) The __stdcall calling convention is used for the function call.

GETQUERY3

DESCRIPTION:

Returns the Mux controller board's firmware checksum in the cksum variable passed to the routine, the serial number of the Mux controller board in the serialnumb variable passed to the routine, the multiplexer type in the muxtype variable passed to the routine, and the board description as an ASCII code array from the USB EEPROM data.

CALLING FORMAT:

GetQuery3(devHandle, cksum, serialnumb, muxtype, Descript)

VISUAL C++ DECLARE:

extern "C" __declspec(dllimport) long WINAPI GetQuery3(FT_HANDLE lngHandle, long* cksum, long* serialnumb, long* muxtype, long Descript[]);

VISUAL C++ CALL:

{return value} = GetQuery3(devHandle, cksum, serialnumb, muxtype, Descript);

VISUAL BASIC DECLARE:

Public Declare Function GetQuery3 Lib "ADICLib_256.dll" (ByVal lngHandle As Long, _ ByRef cksum As Long, ByRef serialnumb As Long, ByRef muxtype As Long,_ ByRef Descript As Long) As Long

VISUAL BASIC CALL:

{return value} = Call GetQuery3(devHandle, cksum, serialnumb, muxtype, Descript(0))

LABVIEW FUNCTION PROTOTYPE:

long GetQuery3(long arg1, long *arg2, long *arg3, long *arg4, long *arg5);

PARAMETERS:

deviceHandle = device handle address of type FT_HANDLE (see header file) returned from the **GetDeviceHandle** function.

cksum = Returned value of the checksum of the Mux Controller Board's firmware

serialnumb = Returned value of the serial number of the Mux Controller Board

muxtype = Returned value that identifies which revision of multiplexer is attached to the interface board. A 0 identifies the Rev F multiplexer and a 1 identifies the Rev G multiplexer.

Descript = Returned array of ASCII codes representing the text string description of the board. Returned as a long datatype ASCII code array to get around string passing differences between Visual C++ and Visual BASIC. For Visual BASIC, rebuild the text string in a For|Next loop of 32 loops with the code:

```
Description = ""

For x = 0 to 31

Description = Description + CHR(Descript(x))

Next x
```

{return value} = Passed status variable from the function call. If the function completed successfully status will return with a value of 1, otherwise an error code is returned, see the status error definition help topic.

PROGRAMMING NOTES:

1) The __stdcall calling convention is used for the function call.

HIDEBADPIXELS

DESCRIPTION:

Flag that toggles whether bad pixels are shown or masked in the data returned by the **GetData** procedure. A value of "1" will mask bad pixels, a "0" will show bad pixels. The default value of the HideBadPixels flag is internally set to "1" to perform bad pixel replacement. If the HideBadPixels procedure is never called, bad pixels will always be replaced based on the bad pixel map stored in the PIC. For multiple array controller board applications, this function operates on the first detected board in the system, board 1, only. For multi-board applications use the HideBadPixels_Brd function.

CALLING FORMAT:

HideBadPixels(numb)

VISUAL C++ DECLARE:

extern "C" __declspec(dllimport) long WINAPI HideBadPixels(long numb);

VISUAL C++ CALL:

{return value} = HideBadPixels(numb);

VISUAL BASIC DECLARE:

Public Declare Function HideBadPixels Lib "ADICLib_256.dll" (ByVal numb As Long) As Long

VISUAL BASIC CALL:

{return value} = HideBadPixels(numb)

LABVIEW FUNCTION PROTOTYPE:

long HideBadPixels(long arg1);

PARAMETERS:

numb = flag that sets whether bad pixels are masked or show in the data from the **GetData** procedure. A value of "1" will mask bad pixels, a "0" will show bad pixels.

{return value} = Returned function error status. A "1" indicates successful completion of the function call.

PROGRAMMING NOTES:

1) The __stdcall calling convention is used for the function call.

HIDEBADPIXELS_BRD

DESCRIPTION:

Flag that toggles whether bad pixels are shown or masked in the data returned by the **GetData** procedure. A value of "1" will mask bad pixels, a "0" will show bad pixels. The default value of the HideBadPixels flag is internally set to "1" to perform bad pixel replacement. If the HideBadPixels procedure is never called, bad pixels will always be replaced based on the bad pixel map stored in the PIC. This function operates on individual array controller boards in a multi-board system by passing the desired board's device handle to the function.

CALLING FORMAT:

HideBadPixels_Brd(devHandle, numb)

VISUAL C++ DECLARE:

extern "C" __declspec(dllimport) long WINAPI HideBadPixels_Brd(FT_HANDLE lngHandle, long numb);

VISUAL C++ CALL:

{return value} = HideBadPixels_Brd(devHandle, numb);

VISUAL BASIC DECLARE:

Public Declare Function HideBadPixels_Brd Lib "ADICLib_256.dll" (ByVal devHandle as Long, ByVal numb As Long) As Long

VISUAL BASIC CALL:

{return value} = HideBadPixels_Brd(devHandle, numb)

LABVIEW FUNCTION PROTOTYPE:

long HideBadPixels_Brd(long arg1, long arg2);

PARAMETERS:

deviceHandle = device handle address of type FT_HANDLE (see header file) returned from the **GetDeviceHandle** function.

numb = flag that sets whether bad pixels are masked or show in the data from the **GetData** procedure. A value of "1" will mask bad pixels, a "0" will show bad pixels.

{return value} = Returned function error status. A "1" indicates successful completion of the function call.

PROGRAMMING NOTES:

1) The __stdcall calling convention is used for the function call.

MARKBADPIXEL

DESCRIPTION:

Sends the bad pixel array map to the PIC processor on the Mux Controller Board. To store the bad pixel map into the controller board's EEPROM for permanent setting, use the StoreAll procedure.

CALLING FORMAT:

MarkBadPixel(devHandle, mxSize, numBPix, bpMap)

VISUAL C++ DECLARE:

extern "C" __declspec(dllimport) long WINAPI MarkBadPixel(FT_HANDLE lngHandle, long mxSize, long numBPix, long bpMap[]);

VISUAL C++ CALL:

{return value} = MarkBadPixel(devHandle, mxSize, numBPix, bpMap);

VISUAL BASIC DECLARE:

Public Declare Function MarkBadPixel Lib "ADICLib_256.dll" (ByVal lngHandle As Long, ByVal mxSize As Long, _ ByVal numBPix As Long, ByRef bpMap As Long) As Long

VISUAL BASIC CALL:

{return value} = MarkBadPixel(devHandle, mxSize, numBPix, bpMap(0))

LABVIEW FUNCTION PROTOTYPE:

long MarkBadPixel(long arg1, long arg2, long arg3, long *arg4);

PARAMETERS:

deviceHandle = device handle address of type FT_HANDLE (see header file) returned from the **GetDeviceHandle** function.

mxSize = Size of the array, either 128 or 256

numBPix = Number of bad pixels in the array. Must equal the number of pixels marked bad in the bad pixel array map

bpMap = Array indicating which pixels are bad. Bad pixels are marked with a "1", good pixels are marked with a "0"

{return value} = Passed status variable from the function call. If the function completed successfully status will return with a value of 1, otherwise an error code is returned, see the status error definition help topic.

PROGRAMMING NOTES:

1) The __stdcall calling convention is used for the function call.

READE2BLOCK

DESCRIPTION:

Allows the user to read data previousl stored in the unused portion of the PIC's NVRAM using the SaveE2Block command. There are 400 unused bytes of memory available inside the PIC. This memory is in the form of E2PROM and is rated typically at 1 million erase/write cycles with data retention of typically 40 years. This memory is intended for storage of calibration data or other semi permanent configuration data. Data can be read in any block size from 1 byte to 400 bytes. The address offset is in the range of 0 to 399, and offset + number of bytes must not exceed 400 or the routine will kick out with an error and bypass execution.

CALLING FORMAT:

ReadE2Block(devHandle, numByte, baseAddress, e2data)

VISUAL C++ DECLARE:

```
extern "C" __declspec(dllimport) long WINAPI ReadE2Block(FT_HANDLE lngHandle, long numByte, long baseAddress,
  long e2Data[]);
```

VISUAL C++ CALL:

```
{return value} = ReadE2Block(devHandle, numByte, baseAddress, e2Data);
```

VISUAL BASIC DECLARE:

```
Public Declare Function ReadE2Block Lib "ADICLib_256.dll" (ByVal lngHandle As Long, _ ByVal numByte As Long, ByVal baseAddress,
ByRef e2Data As Long) As Long
```

VISUAL BASIC CALL:

```
{return value} = ReadE2Block(devHandle, numByte, baseAddress, e2Data(0))
```

LABVIEW FUNCTION PROTOTYPE:

```
long ReadE2Block(long arg1, long arg2, long arg3, long *arg4);
```

PARAMETERS:

deviceHandle = device handle address of type FT_HANDLE (see header file) returned from the **GetDeviceHandle** function.

numBytes = The number of bytes to read from the NVRAM memory. Valid values are 1 to 400

baseAddress = The address of the first byte to read. Valid values are 0 to 399 (0x18F hex).

Note: baseAddress + numBytes should never exceed 400, to do so will exit the function prematurely without executing the EEPROM read.

e2Data = Data array of bytes read from the PIC's NVRAM

{return value} = Passed status variable from the function call. If the function completed successfully status will return with a value of 1, otherwise an error code is returned, see the status error definition help topic.

PROGRAMMING NOTES:

1) The __stdcall calling convention is used for the function call.

2) Function returns a data type of long.

3) Arrays are always passed by reference. To pass an entire array by reference in Visual BASIC to a DLL written in C, only the first element of the array is passed by reference. The DLL will have full access to the array since arrays are stored sequentially in memory.

READPWM

DESCRIPTION:

Reads the current pulse width modulation duty cycle for the TE cooler control

CALLING FORMAT:

ReadPWM(devHandle, PWMval)

VISUAL C++ DECLARE:

extern "C" __declspec(dllimport) long WINAPI ReadPWM(FT_HANDLE lngHandle, long* PWMval);

VISUAL C++ CALL:

{return value} = ReadPWM(devHandle, PWMval);

VISUAL BASIC DECLARE:

Public Declare Function ReadPWM Lib "ADICLib_256.dll" (ByVal lngHandle As Long, ByRef PWMval As Long) As Long

VISUAL BASIC CALL:

{return value} = ReadPWM(devHandle, PWMval)

LABVIEW FUNCTION PROTOTYPE:

long ReadPWM(long arg1, long *arg2);

PARAMETERS:

deviceHandle = device handle address of type FT_HANDLE (see header file) returned from the **GetDeviceHandle** function.

PWMval = returned value representing the duty cycle of the pulse width modulation control of the TE cooler. The number is in the range 0 to 100, where "0" represents a duty cycle of 0% and 100 represents a duty cycle of 100%.

{return value} = Passed status variable from the function call. If the function completed successfully status will return with a value of 1, otherwise an error code is returned, see the status error definition help topic.

PROGRAMMING NOTES:

1) The __stdcall calling convention is used for the function call.

READTEMP

DESCRIPTION:

Reads the current value of the thermistor bridge circuit. When the value of the thermistor equals to the value of the RSET resistor installed on the controller pc board, the bridge will be balanced and ReadTemp will return a value of 2048.

CALLING FORMAT:

ReadTemp(devHandle, teTemp)

VISUAL C++ DECLARE:

extern "C" __declspec(dllimport) long WINAPI ReadTemp(FT_HANDLE lngHandle, long* teTemp);

VISUAL C++ CALL:

{return value} = ReadTemp(devHandle, teTemp);

VISUAL BASIC DECLARE:

Public Declare Function ReadTemp Lib "ADICLib_256.dll" (ByVal lngHandle As Long, ByRef teTemp As Long) As Long

VISUAL BASIC CALL:

{return value} = ReadTemp(devHandle, teTemp)

LABVIEW FUNCTION PROTOTYPE:

long ReadTemp(long arg1, long *arg2);

PARAMETERS:

deviceHandle = device handle address of type FT_HANDLE (see header file) returned from the **GetDeviceHandle** function.

teTemp = returned value representing the temperature from the array. The range is 0 to 4095. When the array is cooled with the TE Cooler and the temperature stabilizes, the returned value should be approximately 2048.

{return value} = Passed status variable from the function call. If the function completed successfully status will return with a value of 1, otherwise an error code is returned, see the status error definition help topic.

PROGRAMMING NOTES:

1) The __stdcall calling convention is used for the function call.

RESTOREALL

DESCRIPTION:

Retrieves all stored operational configuration data from the MUX controller board's NVRAM such as stored integration time, DAC VH & VL settings, global skim setting, mux size, well size, detector bias setting, bad pixel map array, and per pixel correction coefficients so that a prior operational configuration can be applied for the detector array/mux assembly.

CALLING FORMAT:

RestoreAll(devHandle, BaseSettings, bpMap, coeff)

VISUAL C++ DECLARE:

extern "C" __declspec(dllimport) long WINAPI RestoreAll(FT_HANDLE lngHandle, long BaseSettings[], long bpMap[], long coeff[]);

VISUAL C++ CALL:

{return value} = RestoreAll(devHandle, BaseSettings, bpMap, coeff);

VISUAL BASIC DECLARE:

Public Declare Function RestoreAll Lib "ADICLib_256.dll" (ByVal lngHandle As Long, _ ByRef BaseSettings As Long, ByRef bpMap As Long, ByRef coeff As Long) As Long

VISUAL BASIC CALL:

{return value} = RestoreAll(devHandle, BaseSettings(0), bpMap(0), coeff(0))

LABVIEW FUNCTION PROTOTYPE:

long RestoreAll(long arg1, long *arg2, long *arg3, long *arg4);

PARAMETERS:

deviceHandle = device handle address of type FT_HANDLE (see header file) returned from the **GetDeviceHandle** function.

BaseSettings = A 16 element array containing operational configuration data for the mux.

The array elements are defined as follows:

BaseSettings(0) = Integration Time

BaseSettings(1) = NOT USED, value will ALWAYS be 0

BaseSettings(2) = Global Skim value

BaseSettings(3) = NOT USED, value will ALWAYS be 0

BaseSettings(4) = DACVh value

BaseSettings(5) = NOT USED, value will ALWAYS be 0

BaseSettings(6) = DACVl value

BaseSettings(7) = NOT USED, value will ALWAYS be 0

BaseSettings(8) = NOT USED, value will ALWAYS be 0

BaseSettings(9) = Detector Bias

BaseSettings(10) = Mux Size

BaseSettings(11) = Direction Flag

BaseSettings(12) = Number of Bad Pixels

BaseSettings(13) = Integration Well Size

BaseSettings(14) = Left wndow address

BaseSettings(15) = Right window address

bpMap = Array indicating which pixels are bad. Bad pixels are marked with a "1", good pixels are marked with a "0"

coeffArray = Correction coefficient array returned that sets the per pixel DAC values for correction

{return value} = Passed status variable from the function call. If the function completed successfully status will return with a value of 1, otherwise an error code is returned, see the status error definition help topic.

BASE SETTINGS ARRAY NOTES:

1) GlobalSkim value in volts is computed by:

GblSkimValue = 2.083 * (BaseSettings(2) / 1023) + 0.417

2) DacVH value in volts is computed by:

DacVHValue = 1.786 * (BaseSettings(4) / 1023) + 0.714

3) DacVL value in volts is computed by:

DacVLValue = 1.786 * (BaseSettings(6) / 1023) + 0.714

4) DetBias value in volts is computed by:

DetBiasValue = 6.0 * (BaseSettings(9) / 1023) + 6.0

5) Integration time in us is computed by:

IntTime = BaseSettings(0) * 3.333 (in us)

6) BaseSetting(10) = 256 for a 256 array, and 128 for a 128 array

7) BaseSetting(11) = 0 for left to right, and 1 for right to left

8) BaseSetting(13) = index value for Wellsize. See the wsize parameter in the SetWellSize help

9) BaseSetting(14) = Left window address: 0 to 127. Equal to the number of pixels on the left side of the array to skip during readout

10) BaseSetting(15) = Right window address: 0 to 127. Equal to the number of pixels on the right side of the array to skip during readout.

PROGRAMMING NOTES:

1) The __stdcall calling convention is used for the function call.

2) Function returns a status variable of datatype long, the results are passed back to the calling procedure in the passed variables

3) Arrays are always passed by reference. To pass an entire array by reference in Visual BASIC to a DLL written in C, only the first element of the array is passed by reference. The DLL will have full access to the array since arrays are stored sequentially in memory.

SAVEE2BLOCK

DESCRIPTION:

Allows the user to save data to the unused portion of the PIC's NVRAM. There are 400 unused bytes of memory available inside the PIC. This memory is in the form of E2PROM and is rated typically at 1 million erase/write cycles with data retention of typically 40 years. This memory is intended for storage of calibration data or other semi permanent configuration data. Data can be stored and read in any block size from 1 byte to 400 bytes. To read the memory see the ReadE2Block command.

Note: Each byte read/write cycle takes 4ms, so this command can take in the range of a second or more to execute when writing a 400 byte block (1.6 seconds for all 400 bytes). The address offset is in the range of 0 to 399, and offset + number of bytes must not exceed 400 or the routine will kick out with an error and bypass execution.

CALLING FORMAT:

SaveE2Block(devHandle, numByte, baseAddress, e2data)

VISUAL C++ DECLARE:

extern "C" __declspec(dllimport) long WINAPI SaveE2Block(FT_HANDLE lngHandle, long numByte, long baseAddress, long e2Data[]);

VISUAL C++ CALL:

{return value} = SaveE2Block(devHandle, numByte, baseAddress, e2Data);

VISUAL BASIC DECLARE:

Public Declare Function SaveE2Block Lib "ADICLib_256.dll" (ByVal lngHandle As Long, _ ByVal numByte As Long, ByVal baseAddress, ByRef e2Data As Long) As Long

VISUAL BASIC CALL:

{return value} = SaveE2Block(devHandle, numByte, baseAddress, e2Data(0))

LABVIEW FUNCTION PROTOTYPE:

long SaveE2Block(long arg1, long arg2, long arg3, long *arg4);

PARAMETERS:

deviceHandle = device handle address of type FT_HANDLE (see header file) returned from the **GetDeviceHandle** function.

numBytes = The number of bytes to write to the NVRAM. Valid values are 1 to 400

baseAddress = The address of the first byte to store. Valid values are 0 to 399 (0x18F hex).

Note: baseAddress + numBytes can never exceed 400, to do so will exit the function prematurely without executing the EEPROM write.

e2Data = Data array of bytes to store into the PIC's NVRAM

{return value} = Passed status variable from the function call. If the function completed successfully status will return with a value of 1, otherwise an error code is returned, see the status error definition help topic.

PROGRAMMING NOTES:

1) The __stdcall calling convention is used for the function call.

2) Function returns a data type of long.

3) Arrays are always passed by reference. To pass an entire array by reference in Visual BASIC to a DLL written in C, only the first element of the array is passed by reference. The DLL will have full access to the array since arrays are stored sequentially in memory.

SENDALLCOEFF

DESCRIPTION:

Sends the per pixel correction coefficient array to the mux via the PIC processor. The PIC processor applies the per pixel correction coefficients to the Mux for the on-plane offset correction. The coefficient array resident in the PIC's run time memory is not stored to the PIC's NVRAM with the SendAllCoeff command, to store the coefficients to the PIC's NVRAM use the StoreAll command.

CALLING FORMAT:

SendAllCoeff(devHandle, mxSize, coeff)

VISUAL C++ DECLARE:

extern "C" __declspec(dllimport) long WINAPI SendAllCoeff(FT_HANDLE lngHandle, long mxSize, long coeff[]);

VISUAL C++ CALL:

{return value} = SendAllCoeff(devHandle, mxSize, coeff);

VISUAL BASIC DECLARE:

Public Declare Function SendAllCoeff Lib "ADICLib_256.dll" (ByVal lngHandle As Long, _ ByVal mxSize As Long, ByRef coeff As Long) As Long

VISUAL BASIC CALL:

{return value} = SendAllCoeff(devHandle, mxSize, coeff(0))

LABVIEW FUNCTION PROTOTYPE:

long SendAllCoeff(long arg1, long arg2, long *arg3);

PARAMETERS:

deviceHandle = device handle address of type FT_HANDLE (see header file) returned from the **GetDeviceHandle** function.

mxSize = the number of elements in the single dimension array (128 or 256)

coeff = Correction coefficient array sent that sets the per pixel DAC values for correction

{return value} = Passed status variable from the function call. If the function completed successfully status will return with a value of 1, otherwise an error code is returned, see the status error definition help topic.

PROGRAMMING NOTES:

1) The __stdcall calling convention is used for the function call.

2) Function returns a status variable of datatype long

3) Arrays are always passed by reference. To pass an entire array by reference in Visual BASIC to a DLL written in C, only the first element of the array is passed by reference. The DLL will have full access to the array since arrays are stored sequentially in memory.

SETCONVERSIONREF

DESCRIPTION:

Sets the software D/A conversion factor used in the **GetData** procedure for the conversion of the 16 bit (0 to 65535) sampled data retrieved from the PIC by the **GetData** command to format the data into an analog representation in volts. The default value of the conversion factor is 13107 and ordinarily does not need to be changed. The **GetData** command returns data in volts in the range of 0v - Vmax. For multiple array controller applications, this function operates on the first detected board in the system, board 1, only. For multi-board applications use the **SetConversionRef_Brd** function.

The A/D conversion factor is used internally by the **GetData** routine via the following formula:

{floating point value) = {16 bit A/D conversion value from the PIC} / D/A conversion factor

which becomes {16 bit A/D conversion value from the PIC} / 13107 when the default value is used.

CALLING FORMAT:

SetConversionRef(numb)

VISUAL C++ DECLARE:

extern "C" __declspec(dllimport) long WINAPI SetConversionRef(long numb);

VISUAL C++ CALL:

{return value} = SetConversionRef(long numb);

VISUAL BASIC DECLARE:

Public Declare Function SetConversionRef Lib "ADICLib_256.dll" (ByVal numb As Long) As Long

VISUAL BASIC CALL:

{return value} = SetConversionRef(numb)

LABVIEW FUNCTION PROTOTYPE:

long SetConversionRef(long arg1);

PARAMETERS:

numb = D/A convert value for the software conversion of the digitally sampled data retrieved from the PIC in the **GetData** procedure to convert the data to an analog representation in volts. The default value is set internally at 13107 until it is changed by the **SetConversionRef** command.

{return value} = Returned function error status. A "1" indicates successful completion of the function call.

PROGRAMMING NOTES:

1) The __stdcall calling convention is used for the function call.

SETCONVERSIONREF_BRD

DESCRIPTION:

Sets the software D/A conversion factor used in the **GetData** procedure for the conversion of the 16 bit (0 to 65535) sampled data retrieved from the PIC by the **GetData** command to format the data into an analog representation in volts. The default value of the conversion factor is 13107 and ordinarily does not need to be changed. The **GetData** command returns data in volts in the range of 0v - Vmax. This function operates on individual array controller boards in a multi-board system by passing the desired board's device handle to the function.

The A/D conversion factor is used internally by the **GetData** routine via the following formula:

{floating point value) = {16 bit A/D conversion value from the PIC} / D/A conversion factor

which becomes {16 bit A/D conversion value from the PIC} / 13107 when the default value is used.

CALLING FORMAT:

SetConversionRef_Brd(devHandle, numb)

VISUAL C++ DECLARE:

extern "C" __declspec(dllimport) long WINAPI SetConversionRef_Brd(FT_HANDLE lngHandle, long numb);

VISUAL C++ CALL:

{return value} = SetConversionRef_Brd(devHandle, long numb);

VISUAL BASIC DECLARE:

Public Declare Function SetConversionRef_Brd Lib "ADICLib_256.dll" (ByVal devHandle as Long, ByVal numb As Long) As Long

VISUAL BASIC CALL:

{return value} = SetConversionRef_Brd(devHandle, numb)

LABVIEW FUNCTION PROTOTYPE:

long SetConversionRef_Brd(long arg1, long arg2);

PARAMETERS:

deviceHandle = device handle address of type FT_HANDLE (see header file) returned from the **GetDeviceHandle** function.

numb = D/A convert value for the software conversion of the digitally sampled data retrieved from the PIC in the **GetData** procedure to convert the data to an analog representation in volts. The default value is set internally at 13107 until it is changed by the SetConversionRef_Brd command.

{return value} = Returned function error status. A "1" indicates successful completion of the function call.

PROGRAMMING NOTES:

1) The __stdcall calling convention is used for the function call.

SETDACGSKIM

DESCRIPTION:

Sets the value of the global skim applied to the multiplexer. The global skim value is represented by a 10bit digital word, the valid range is 0 to 1023, which equates to a global skim voltage range of 0.417 to 2.50Volts. A value "0" is no skim, a value of 1023 is maximum skim.

CALLING FORMAT:

SetDacGSkim(devHandle, setValue)

VISUAL C++ DECLARE:

extern "C" __declspec(dllimport) long WINAPI SetDacGSkim(FT_HANDLE lngHandle, long setValue);

VISUAL C++ CALL:

{return value} = SetDacGSkim(devHandle, setValue);

VISUAL BASIC DECLARE:

Public Declare Function SetDacGSkim Lib "ADICLib_256.dll" (ByVal lngHandle As Long, _ ByVal setValue As Long, ByVal setValue As Long) As Long

VISUAL BASIC CALL:

{return value} = SetDacGSkim(devHandle, setValue)

LABVIEW FUNCTION PROTOTYPE:

long SetDacGSkim(long arg1, long arg2);

PARAMETERS:

deviceHandle = device handle address of type FT_HANDLE (see header file) returned from the **GetDeviceHandle** function.

setValue = Digital word representing the value of the global skim to set in the mux. The valid range for setValue is 0 to 1023

{return value} = Passed status variable from the function call. If the function completed successfully status will return with a value of 1, otherwise an error code is returned, see the status error definition help topic.

PROGRAMMING NOTES:

1) The __stdcall calling convention is used for the function call.

SETDACREFERENCES

DESCRIPTION:

Sets override values for the DAC references for the per pixel DACs for the **DoCalibrate** calibration procedure. The values for the DAC references are set by a 10bit digital word, the valid range is -1 and 0 to 1023. The DAC references are normally determined automatically by the **DoCalibrate** command, set by inputing a -1 for numbVH and numbVL with the SetDacReferences command or left set to the DLL initialized values of -1 by never calling the SetDacReferences command. Inputing values of 0 to 1023 for numbVH and numbVL allows overriding of the automatic calculation of the references to be set manually to the input numbVH and numbVL values. For multiple array controller applications, this function operates on the first detected board in the system, board 1, only. For multi-board applications use the **SetDacReferences_Brd** function.

Setting numbVH and numbVL to -1 (or leaving the DLL initialization state at -1) sets the internal flags dacVhUseVal and dacVlUseVal to -1 which tells the **DoCalibrate** procedure to automatically compute the DAC references.

Generally the DAC references should be allowed to be computed automatically by the **DoCalibrate** command so the SetDacReferences should not normally ever be needed and dacVhUseVal and dacVlUseVal left to their initialized values of -1.

CALLING FORMAT:

SetDacReferences(numbVH, numbVL)

VISUAL C++ DECLARE:

extern "C" __declspec(dllimport) long WINAPI SetDacReferences(long numbVH, long numbVL);

VISUAL C++ CALL:

{return value} = SetDacReferences(numbVH, numbVL);

VISUAL BASIC DECLARE:

Public Declare Function SetDacReferences Lib "ADICLib_256.dll" (ByVal numbVH As Long, _ ByVal numbVL As Long) As Long

VISUAL BASIC CALL:

{return value} = SetDacReferences(numbVH, numbVL)

LABVIEW FUNCTION PROTOTYPE:

long SetDacReferences(long arg1, long arg2);

PARAMETERS:

numbVH = Digital word set value for the upper DAC reference for use during the calibration procedure. The valid range for setValue is -1 and 0 to 1023.

numbVL = Digital word set value for the lower DAC reference for use during the calibration procedure. The valid range for setValue is -1 and 0 to 1023.

{return value} = Returned function error status. A "1" indicates successful completion of the function call.

PROGRAMMING NOTES:

1) The __stdcall calling convention is used for the function call.

SETDACREFERENCES_BRD

DESCRIPTION:

Sets override values for the DAC references for the per pixel DACs for the **DoCalibrate** calibration procedure. The values for the DAC references are set by a 10bit digital word, the valid range is -1 and 0 to 1023. The DAC references are normally determined automatically by the **DoCalibrate** command, set by inputing a -1 for numbVH and numbVL with the SetDacReferences_Brd command or left set to the DLL initialized values of -1 by never calling the SetDacReferences_Brd command. Inputing values of 0 to 1023 for numbVH and numbVL allows overriding of the automatic calculation of the references to be set manually to the input numbVH and numbVL values. This function operates on individual array controller boards in a multi-board system by passing the desired board's device handle to the function.

Setting numbVH and numbVL to -1 (or leaving the DLL initialization state at -1) sets the internal flags dacVhUseVal and dacVlUseVal to -1 which tells the DoCalibrate procedure to automatically compute the DAC references.

Generally the DAC references should be allowed to be computed automatically by the **DoCalibrate** command so the SetDacReferences_Brd should not normally ever be needed and dacVhUseVal and dacVlUseVal left to their initialized values of -1.

CALLING FORMAT:

SetDacReferences_Brd(devHandle, numbVH, numbVL)

VISUAL C++ DECLARE:

extern "C" __declspec(dllimport) long WINAPI SetDacReferences_Brd(FT_HANDLE lngHandle, long numbVH, long numbVL);

VISUAL C++ CALL:

{return value} = SetDacReferences_Brd(devHandle, numbVH, numbVL);

VISUAL BASIC DECLARE:

Public Declare Function SetDacReferences_Brd Lib "ADICLib_256.dll" (ByVal devHandle as Long, ByVal numbVH As Long, _ ByVal numbVL As Long) As Long

VISUAL BASIC CALL:

{return value} = SetDacReferences_Brd(devHandle, numbVH, numbVL)

LABVIEW FUNCTION PROTOTYPE:

long SetDacReferences_Brd(long arg1, long arg2, long arg3);

PARAMETERS:

deviceHandle = device handle address of type FT_HANDLE (see header file) returned from the **GetDeviceHandle** function.

numbVH = Digital word set value for the upper DAC reference for use during the calibration procedure. The valid range for setValue is -1 and 0 to 1023.

numbVL = Digital word set value for the lower DAC reference for use during the calibration procedure. The valid range for setValue is -1 and 0 to 1023.

{return value} = Returned function error status. A "1" indicates successful completion of the function call.

PROGRAMMING NOTES:

1) The __stdcall calling convention is used for the function call.

SETDACVH

DESCRIPTION:

Sets the upper DAC reference for the per pixel correction DAC on the mux controller board. The the DACVh value is set by a 10 bit word, the valid range for the DACVh value is 0 to 1023, which equates to a DacVh voltage range of 0.714 to 2.5 volts.

CALLING FORMAT:

SetDacVH(devHandle, setValue)

VISUAL C++ DECLARE:

extern "C" __declspec(dllimport) long WINAPI SetDacVH(FT_HANDLE lngHandle, long setValue);

VISUAL C++ CALL:

{return value} = SetDacVH(devHandle, setValue);

VISUAL BASIC DECLARE:

Public Declare Function SetDacVH Lib "ADICLib_256.dll" (ByVal lngHandle As Long, _ ByVal setValue As Long) As Long

VISUAL BASIC CALL:

{return value} = SetDacVH(devHandle, setValue)

LABVIEW FUNCTION PROTOTYPE:

long SetDacVH(long arg1, long arg2);

PARAMETERS:

deviceHandle = device handle address of type FT_HANDLE (see header file) returned from the **GetDeviceHandle** function.

setValue = Digital word representing the value of the upper DAC reference to set on the mux controller board. The valid range for setValue is 0 to 1023.

{return value} = Passed status variable from the function call. If the function completed successfully status will return with a value of 1, otherwise an error code is returned, see the status error definition help topic.

PROGRAMMING NOTES:

1) The __stdcall calling convention is used for the function call.

SETDACVL

DESCRIPTION:

Sets the lower DAC reference for the per pixel correction DAC on the mux controller board. The the DACVl value is set by a 10 bit word, the valid range for the DACVl value is 0 to 1023, which equates to a DacVl voltage range of 0.714 to 2.50 volts.

CALLING FORMAT:

SetDacVL(devHandle, setValue)

VISUAL C++ DECLARE:

extern "C" __declspec(dllimport) long WINAPI SetDacVL(FT_HANDLE lngHandle, long setValue);

VISUAL C++ CALL:

{return value} = SetDacVL(devHandle, setValue);

VISUAL BASIC DECLARE:

Public Declare Function SetDacVL Lib "ADICLib_256.dll" (ByVal lngHandle As Long, _

ByVal setValue As Long) As Long

VISUAL BASIC CALL:

{return value} = SetDacVL(devHandle, setValue)

LABVIEW FUNCTION PROTOTYPE:

long SetDacVL(long arg1, long arg2);

PARAMETERS:

deviceHandle = device handle address of type FT_HANDLE (see header file) returned from the **GetDeviceHandle** function.

setValue = Digital word representing the value of the lower DAC reference to set on the mux controller board. The valid range for setValue is 0 to 1023.

{return value} = Passed status variable from the function call. If the function completed successfully status will return with a value of 1, otherwise an error code is returned, see the status error definition help topic.

PROGRAMMING NOTES:

1) The __stdcall calling convention is used for the function call.

SETDETBIAS

DESCRIPTION:

Sets the detector bias voltage applied to the detector substrate. The detector bias voltage is set by a 10 bit word by the SetDetBias procedure. The valid range for the set value is 0 to 1023, where "0" is approximately 6 volts and "1023" is approximately 12 volts, in linear voltage steps.

CALLING FORMAT:

SetDetBias(devHandle, setValue)

VISUAL C++ DECLARE:

extern "C" __declspec(dllimport) long WINAPI SetDetBias(FT_HANDLE lngHandle, long setValue);

VISUAL C++ CALL:

{return value} = SetDetBias(devHandle, setValue);

VISUAL BASIC DECLARE:

Public Declare Function SetDetBias Lib "ADICLib_256.dll" (ByVal lngHandle As Long, _ ByVal setValue As Long) As Long

VISUAL BASIC CALL:

{return value} = SetDetBias(devHandle, setValue)

LABVIEW FUNCTION PROTOTYPE:

long SetDetBias(long arg1, long arg2);

PARAMETERS:

deviceHandle = device handle address of type FT_HANDLE (see header file) returned from the **GetDeviceHandle** function.

setValue = Digital word representing the value for the detector bias voltage applied to the detector array substrate. The valid range for setValue is 0 to 1023.

Note: A rail to rail output opamp is used to drive the detector bias signal into the array. The max voltage applied with be the lesser of 12V or (Vsupply - 0.300V). It is recommended that the commanded voltage not exceed the linear output swing range of the amplifier. Vsupply is the DC supply voltage applied to the interface electronics board, typically 12V DC

{return value} = Passed status variable from the function call. If the function completed successfully status will return with a value of 1, otherwise an error code is returned, see the status error definition help topic.

PROGRAMMING NOTES:

1) The __stdcall calling convention is used for the function call.

SETGLOBALSKIMVAL

DESCRIPTION:

Sets the global skim variable, gskimUseVal, value to use during the **DoCalibrate** calibration procedure. The default value of the gskimUseVal value is "0" which tells the **DoCalibrate** procedure to turn off global skimming during calibration and calibration will be performed using only the per pixel skim functionality. Setting gskimUseVal to "-1" will tell the **DoCalibrate** procedure to automatically determine the value for the mux global skim during calibration, which is useful for low impedance detectors. Setting gskimUseVal to a none zero positive number will use that value during calibration. The gskimUseVal is represented by a 10 bit number, the valid range is -1 and 0 to 1023. The gskimUseVal is set default to "0" internally in the software, and if the SetGlobalSkimVal procedure is never called, global skim will be disabled and per pixel skim will be the only skim used for calibration. For multiple array controller applications, this function operates on the first detected board in the system, board 1, only. For multi-board applications use the **SetGlobalSkimVal_Brd** function.

CALLING FORMAT:

SetGlobalSkimVal(gskimUseVal)

VISUAL C++ DECLARE:

extern "C" __declspec(dllimport) long WINAPI SetGlobalSkimVal(long gskimUseVal);

VISUAL C++ CALL:

{return value} = SetGlobalSkimVal(gskimUseVal);

VISUAL BASIC DECLARE:

Public Declare Function SetGlobalSkimVal Lib "ADICLib_256.dll" (ByVal gskimUseVal As Long) As Long

VISUAL BASIC CALL:

{return value} = SetGlobalSkimVal(gskimUseVal)

LABVIEW FUNCTION PROTOTYPE:

long SetGlobalSkimVal(long arg1);

PARAMETERS:

gskimUseVal = Digital word representing the value for the global skim value to use during calibration. A "-1" will let the calibration procedure automatically determine the value for the global skim, "0" will turn global skim off, and a positive value will use that value during the calibration procedure. The the valid range is -1 and 0 to 1023

{return value} = Returned function error status. A "1" indicates successful completion of the function call.

PROGRAMMING NOTES:

1) The __stdcall calling convention is used for the function call.

SETGLOBALSKIMVAL_BRD

DESCRIPTION:

Sets the global skim variable, gskimUseVal, value to use during the **DoCalibrate** calibration procedure. The default value of the gskimUseVal value is "0" which tells the **DoCalibrate** procedure to turn off global skimming during calibration and calibration will be performed using only the per pixel skim functionality. Setting gskimUseVal to "-1" will tell the **DoCalibrate** procedure to automatically determine the value for the mux global skim during calibration, which is useful for low impedance detectors. Setting gskimUseVal to a none zero positive number will use that value during calibration. The gskimUseVal is represented by a 10 bit number, the valid range is -1 and 0 to 1023. The gskimUseVal is set default to "0" internally in the software, and if the SetGlobalSkimVal_Brd procedure is never called, global skim will be disabled and per pixel skim will be the only skim used for calibration. This function operates on individual array controller boards in a multi-board system by passing the desired board's device handle to the function.

CALLING FORMAT:

SetGlobalSkimVal_Brd(devHandle, gskimUseVal)

VISUAL C++ DECLARE:

extern "C" __declspec(dllimport) long WINAPI SetGlobalSkimVal_Brd(FT_HANDLE lngHandle, long gskimUseVal);

VISUAL C++ CALL:

{return value} = SetGlobalSkimVal_Brd(devHandle, gskimUseVal);

VISUAL BASIC DECLARE:

Public Declare Function SetGlobalSkimVal_Brd Lib "ADICLib_256.dll" (ByVal devHandle as Long, ByVal gskimUseVal As Long) As Long

VISUAL BASIC CALL:

{return value} = SetGlobalSkimVal_Brd(devHandle, gskimUseVal)

LABVIEW FUNCTION PROTOTYPE:

long SetGlobalSkimVal_Brd(long arg1, long arg2);

PARAMETERS:

deviceHandle = device handle address of type FT_HANDLE (see header file) returned from the **GetDeviceHandle** function.

gskimUseVal = Digital word representing the value for the global skim value to use during calibration. A "-1" will let the calibration procedure automatically determine the value for the global skim, "0" will turn global skim off, and a positive value will use that value during the calibration procedure. The the valid range is -1 and 0 to 1023

{return value} = Returned function error status. A "1" indicates successful completion of the function call.

PROGRAMMING NOTES:

1) The __stdcall calling convention is used for the function call.

SETINTEGRATION

DESCRIPTION:

Sets the charge well integration time of the mux/detector array. The integration time is represented by a 16 bit number in the range 3 to 65535, where "65535" will give the maximum integration time and "3" will give the minimum integration time.

CALLING FORMAT:

SetIntegration(devHandle, intValue)

VISUAL C++ DECLARE:

extern "C" __declspec(dllimport) long WINAPI SetIntegration(FT_HANDLE lngHandle, long intValue);

VISUAL C++ CALL:

{return value} = SetIntegration(devHandle, intValue);

VISUAL BASIC DECLARE:

Public Declare Function SetIntegration Lib "ADICLib_256.dll" (ByVal lngHandle As Long, _ ByVal intValue As Long) As Long

VISUAL BASIC CALL:

{return value} = SetIntegration(devHandle, intValue)

LABVIEW FUNCTION PROTOTYPE:

long SetIntegration(long arg1, long arg2);

PARAMETERS:

deviceHandle = device handle address of type FT_HANDLE (see header file) returned from the **GetDeviceHandle** function.

intValue = Digital word representing the value of the mux/detector array integration time The valid range for intValue is 0 to 65535. A value of "3" will give an integration time of approximately 10uS which is the minimum allowable integration time.

{return value} = Passed status variable from the function call. If the function completed successfully status will return with a value of 1, otherwise an error code is returned, see the status error definition help topic.

ALGORITHMN:

integrationtime = 3.333us * (intValue)

PROGRAMMING NOTES:

1) The __stdcall calling convention is used for the function call.

SETMUXSIZE

DESCRIPTION:

Sets the mux/detector array size for arrays constructed using 128 channel readout multiplexers. Only two sizes are valid, either 256 or 128. Setting mxSize to 256 tells the PIC processor that the detector array is two mux die and a 256 element detector array. Setting mxSize to any number other than 256 will tell the PIC processor that the mux/detector array is a 128 element array. If a 256 channel readout multiplexer is detected by the controller board then this function is ignored and should not be called.

CALLING FORMAT:

SetMuxSize(devHandle, mxSize);

VISUAL C++ DECLARE:

extern "C" __declspec(dllimport) long WINAPI SetMuxSize(FT_HANDLE lngHandle, long mxSize);

VISUAL C++ CALL:

{return value} = SetMuxSize(devHandle, mxSize);

VISUAL BASIC DECLARE:

Public Declare Function SetMuxSize Lib "ADICLib_256.dll" (ByVal lngHandle As Long, _ ByVal mxSize As Long) As Long

VISUAL BASIC CALL:

{return value} = SetMuxSize(devHandle, mxSize)

LABVIEW FUNCTION PROTOTYPE:

long SetMuxSize(long arg1, long arg2);

PARAMETERS:

deviceHandle = device handle address of type FT_HANDLE (see header file) returned from the **GetDeviceHandle** function.

mxSize = Value that tells the PIC processor the array size. A value of 256 tells the PIC processor that the array is a 256 element detector array. Any other value sets the PIC processor to the 128 detector element mode.

{return value} = Passed status variable from the function call. If the function completed successfully status will return with a value of 1, otherwise an error code is returned, see the status error definition help topic.

PROGRAMMING NOTES:

1) The __stdcall calling convention is used for the function call.

SETWELLSIZE

DESCRIPTION:

Sets the integration charge well size inside of the mux. Valid charge well sizes are 1pF, 4pF, 7pF, and 10pF for Rev F multiplexers, with 11pF, 14pf, 17pF, and 20pF also available on Rev G multiplexers. The charge well sizes are set by an index number (see below).

CALLING FORMAT:

SetWellSize(devHandle, wSize);

VISUAL C++ DECLARE:

extern "C" __declspec(dllimport) long WINAPI SetWellSize(FT_HANDLE lngHandle, long wSize);

VISUAL C++ CALL:

{return value} = SetWellSize(devHandle, wSize, status);

VISUAL BASIC DECLARE:

Public Declare Function SetWellSize Lib "ADICLib_256.dll" (ByVal lngHandle As Long, _ ByVal wSize As Long) As Long

VISUAL BASIC CALL:

{return value} = SetWellSize(devHandle, wSize)

LABVIEW FUNCTION PROTOTYPE:

long SetWellSize(long arg1, long arg2);

PARAMETERS:

deviceHandle = device handle address of type FT_HANDLE (see header file) returned from the **GetDeviceHandle** function.

wSize = index number used to set the integration charge well as follows:

0 = Set 1pF charge well

1 = Set 4pF charge well

2 = Set 7pF charge well

3 = Set 10pF charge well

4 = Set 11pF charge well

5 = Set 14pF charge well

6 = Set 17pF charge well

7 = Set 20pF charge well

{return value} = Passed status variable from the function call. If the function completed successfully status will return with a value of 1, otherwise an error code is returned, see the status error definition help topic.

PROGRAMMING NOTES:

1) The __stdcall calling convention is used for the function call.

STOREALL

DESCRIPTION:

Stores all mux/detector array operational data & settings currently resident in the PIC processor's run-time memory to the PIC processor's NVRAM for future retrieval to a known operating state after a mux controller board power cycle. For a list of stored parameters and settings, see the **RestoreAll** command.

CALLING FORMAT:

StoreAll(devHandle)

VISUAL C++ DECLARE:

extern "C" __declspec(dllimport) long WINAPI StoreAll(FT_HANDLE lngHandle);

VISUAL C++ CALL:

{return value} = StoreAll(devHandle);

VISUAL BASIC DECLARE:

Public Declare Function StoreAll Lib "ADICLib_256.dll" (ByVal lngHandle As Long) As Long

VISUAL BASIC CALL:

{return value} = StoreAll(devHandle)

LABVIEW FUNCTION PROTOTYPE:

long StoreAll(long arg1);

PARAMETERS:

deviceHandle = device handle address of type FT_HANDLE (see header file) returned from the **GetDeviceHandle** function.

{return value} = Passed status variable from the function call. If the function completed successfully status will return with a value of 1, otherwise an error code is returned, see the status error definition help topic.

PROGRAMMING NOTES:

1) The __stdcall calling convention is used for the function call.

SUPPRESSCALSTAT

DESCRIPTION:

Sets a flag that toggles whether pop-up status dialog boxes are enabled during the calibration procedure. The internal default value of the SuppressCalStat flag is "0" which enables pop-up status dialog boxes. Setting the SuppressCalStat flag to "1" will suppress the pop-up status dialog boxes, setting the SuppressCalStat flag to anything other than "1" will set the internal flag to "0", which is the default. If SuppressCalStat is never called, pop-up status dialog boxes are enabled.

CALLING FORMAT:

SuppressCalStat(numb)

VISUAL C++ DECLARE:

extern "C" __declspec(dllimport) long WINAPI SuppressCalStat(long numb);

VISUAL C++ CALL:

{return value} = SuppressCalStat(numb);

VISUAL BASIC DECLARE:

Public Declare Function SuppressCalStat Lib "ADICLib_256.dll" (ByVal numb As Long) As Long

VISUAL BASIC CALL:

{return value} = SuppressCalStat(numb)

LABVIEW FUNCTION PROTOTYPE:

long SuppressCalStat(long arg1);

PARAMETERS:

numb = flag that sets whether pop-up status dialog boxes during calibration are enabled or suppressed. A value of "1" will suppress pop-up status dialog boxes, any other number will enable the pop-up status dialog boxes. The internal default setting is to allow pop-up status dialog boxes.

{return value} = Returned function error status. A "1" indicates successful completion of the function call.

PROGRAMMING NOTES:

1) The __stdcall calling convention is used for the function call.

SUPPRESSERRORS

DESCRIPTION:

Sets a flag that toggles whether pop-up error dialog boxes are enabled for most all procedures. Setting the SuppressErrors flag to "1" will suppress pop-up error dialog boxes in all functions that have pop-up error dialog boxes except the **GetDeviceHandle** and **MarkBadPixel** routines. Setting the SuppressErrors flag to "0" will suppress pop-up error dialog boxes. Procedures that have pop-up error boxes suppressed will still return error codes in the status bit if errors are detected even though the pop-up error dialog boxes are suppressed, but pop-up error warning dialog boxes that will halt the program till the user clicks the "OK" button will be suppressed allowing error handling to be handled or ignored by the top level calling program via the status code returned from the data retrieval procedure call. The programmed default setting is to suppress pop-up error dialog boxes to occur if the SuppressErrors function is never called. To enable pop-up error dialog boxes, call the SuppressErrors function with parameter "0".

CALLING FORMAT:

SuppressErrors(numb)

VISUAL C++ DECLARE:

extern "C" __declspec(dllimport) long WINAPI SuppressErrors(long numb);

VISUAL C++ CALL:

{return value} = SuppressErrors(numb);

VISUAL BASIC DECLARE:

Public Declare Function SuppressErrors Lib "ADICLib_256.dll" (ByVal numb As Long) As Long

VISUAL BASIC CALL:

{return value} = SuppressErrors(numb)

LABVIEW FUNCTION PROTOTYPE:

long SuppressErrors(long arg1);

PARAMETERS:

numb = flag that sets whether pop-up error dialog boxes in the ReadTemp, **GetData**, and ReadPWM procedures are suppressed. A value of "1" will suppress pop-up error dialog boxes, any other number will enable the pop-up error dialog boxes. The internal default setting is to suppress pop-up error dialog boxes.

{return value} = Returned function error status. A "1" indicates successful completion of the function call.

PROGRAMMING NOTES:

1) The __stdcall calling convention is used for the function call.

SYNCPC

DESCRIPTION:

Retrieves all current operational configuration data from the PIC processor's run-time memory such as integration time, DAC VH & VL settings, global skim setting, mux size, well size, detector bias setting, bad pixel map array, and per pixel correction coefficients to synchronize the host computer's settings to the current mux controller board operational configuration.

This command is effectively the same as the **RestoreAll** command, but retrieves data from the PIC's run-time memory instead of the controller board's EEPROM. This command is useful in case of a computer crash and the mux controller board has not been power cycled and the current settings are desired to not be lost.

CALLING FORMAT:

SyncPC(devHandle, BaseSettings, bpMap, coeff)

VISUAL C++ DECLARE:

extern "C" __declspec(dllimport) long WINAPI SyncPC(FT_HANDLE lngHandle, long BaseSettings[], long bpMap[], long coeff[]);

VISUAL C++ CALL:

{return value} = SyncPC(devHandle, BaseSettings, bpMap, coeff);

VISUAL BASIC DECLARE:

Public Declare Function SyncPC Lib "ADICLib_256.dll" (ByVal lngHandle As Long, _ ByRef BaseSettings As Long, ByRef bpMap As Long, ByRef coeff As Long) As Long

VISUAL BASIC CALL:

{return value} = SyncPC(devHandle, BaseSettings(0), bpMap(0), coeff(0))

LABVIEW FUNCTION PROTOTYPE:

long SyncPC(long arg1, long *arg2, long *arg3, long *arg4);

PARAMETERS:

deviceHandle = device handle address of type FT_HANDLE (see header file) returned from the **GetDeviceHandle** function.

BaseSettings = A 16 element array containing operational configuration data for the mux.

The array elements are defined as follows:

BaseSettings(0) = Integration Time

BaseSettings(1) = NOT USED, value will ALWAYS be 0

BaseSettings(2) = Global Skim value

BaseSettings(3) = NOT USED, value will ALWAYS be 0

BaseSettings(4) = DACVh value

BaseSettings(5) = NOT USED, value will ALWAYS be 0

BaseSettings(6) = DACVl value

BaseSettings(7) = NOT USED, value will ALWAYS be 0

BaseSettings(8) = NOT USED, value will ALWAYS be 0

BaseSettings(9) = Detector Bias

BaseSettings(10) = Mux Size

BaseSettings(11) = NOT USED, value will ALWAYS be 0

BaseSettings(12) = Number of Bad Pixels

BaseSettings(13) = Integration Well Size

BaseSettings(14) = NOT USED, value will ALWAYS be 0

BaseSettings(15) = NOT USED, value will ALWAYS be 0

bpMap = Array indicating which pixels are bad. Bad pixels are marked with a "1", good pixels are marked with a "0"

coeffArray = Correction coefficient array returned that sets the per pixel DAC values for correction

{return value} = Passed status variable from the function call. If the function completed successfully status will return with a value of 1, otherwise an error code is returned, see the status error definition help topic.

BASE SETTINGS ARRAY NOTES:

1) GlobalSkim value in volts is computed by:

GblSkimValue = 2.083 * (BaseSettings(2) / 1023) + 0.417

2) DacVH value in volts is computed by:

DacVHValue = 1.786 * (BaseSettings(4) / 1023) + 0.714

3) DacVL value in volts is computed by:

DacVLValue = 1.786 * (BaseSettings(6) / 1023) + 0.714

4) DetBias value in volts is computed by:

DetBiasValue = 6.0 * (BaseSettings(9) / 1023) + 6.0

5) Integration time in us is computed by:

IntTime = BaseSettings(0) * 3.333 (in us)

6) BaseSetting(10) = 256 for a 256 array, and 128 for a 128 array

7) BaseSettings(13) = index value for Wellsize. See the wsize parameter in the **SetWellSize** help

PROGRAMMING NOTES:

1) The __stdcall calling convention is used for the function call.

2) Function returns a status variable as long datatype, the results are passed back to the calling procedure in the passed variables

3) Arrays are always passed by reference. To pass an entire array by reference in Visual BASIC to a DLL written in C, only the first element of the array is passed by reference. The DLL will have full access to the array since arrays are stored sequentially in memory.

TECOOLERPOWER

DESCRIPTION:

Turns the TE cooler ON or OFF. If teStatus is set to "1" the TE cooler will be turned ON. Any other number will turn the TE cooler OFF.

CALLING FORMAT:

TECoolerPower(devHandle, teStatus)

VISUAL C++ DECLARE:

extern "C" __declspec(dllimport) long WINAPI TECoolerPower(FT_HANDLE lngHandle, long teStatus);

VISUAL C++ CALL:

{return value} = TECoolerPower(devHandle, teStatus, status);

VISUAL BASIC DECLARE:

Public Declare Function TECoolerPower Lib "ADICLib_256.dll" (ByVal lngHandle As Long, _ ByVal teStatus As Long) As Long

VISUAL BASIC CALL:

{return value} = TECoolerPower(devHandle, teStatus)

LABVIEW FUNCTION PROTOTYPE:

long TECoolerPower(long arg1, long arg2);

PARAMETERS:

deviceHandle = device handle address of type FT_HANDLE (see header file) returned from the **GetDeviceHandle** function.

teStatus = Flag to turn the TE cooler ON or OFF. If teStatus is set to "1" the TE cooler will be turned ON. Any number other than "1" will turn the TE cooler OFF.

{return value} = Passed status variable from the function call. If the function completed successfully status will return with a value of 1, otherwise an error code is returned, see the status error definition help topic.

PROGRAMMING NOTES:

1) The __stdcall calling convention is used for the function call.

2) See the include file "ADICLIB_256R4_Functions.h" in the installation directory.

3) See the include file "ADICLIB_256R4.bas" in the installation directory.

STATUS / ERROR CODE DEFINITIONS

Almost all the DLL functions return a status indicator as the value of the function. The exceptions are; **GetDeviceHandle** and **GetBoard_Handle** which return the handle value, **GetNumBoards** which returns the number of attached boards, and **Flux_P**, **Flux_W** which return the double precision value of the flux curve numerical integration. If there are no errors detected in the communications to the controller board then the returned status value will be a value of "1". If an error is detected in communications, an error code indicating the type of error detected is returned. The code returned is a binary weighted code that can indicate when one or more errors have been detected, the value of the status code number is the binary weighted sum of all detected errors. Ordinarily though, it is believed if an error occurs in a function call, it will be the only error and it will be apparent from the error code returned as to what error was detected.

To determine what the errors detected are when there are multiple errors embedded in the status code number returned in the status variable, a logical AND mask function can be applied to determine which error code bits are set based on the following binary weighting:

Bit#	Binary Weighting	Code Description
Bit12	4096	USB EEPROM Read error
Bit11	2048	USB GetStatus error
Bit10	1024	Unknown error
Bit9	512	Extra Data error
Bit8	256	No ETX error
Bit7	128	Checksum Mismatch error
Bit6	64	Command Word Received wrong
Bit5	32	Received NACK
Bit4	16	Not STX error
Bit3	8	USB Read error
Bit2	4	USB Write error
Bit1	2	USB Purge error
Bit0	1	OK - Command completed successfully

ERROR DEFINITIONS:

USB EEPROM Read error = This error can occur in the **GetQuery** and **GetQuery2** commands and deals with the reading of the device information in the USB interface's data EEPROM

USB GetStatus error = This error can originate in the **GetQuery** and **GetQuery2** commands and deals with the statusing of the receive and transmit buffers of the USB interface

Unknown error = This error is for when it is any other error other than the specific error types that are trapped and defined within the DLL

Extra Data error = This error is detected when more data bytes are sent back to the computer than was requested for by the **GetData** function

No ETX error = This error is detected when the End Transmission byte is not detected by the computer at the end of a sequence of data bytes sent by the controller board to the computer

Checksum Mismatch error = This error is detected in any function that sends/receives data to the controller board and refers to the case when the checksum computed by the controller board of the data sent to the computer doesn't match the checksum computed by the computer of the data received by the computer

Command Word Received wrong = This error is detected when the command echoed back to the computer by the controller board doesn't match the command that was sent to the controller board. This is usually caused by calling a DLL function call before the prior DLL function call has completed and returned. In the programming enviroment you want to make sure that multi-threaded calls (calling DLL functions in parallel) is turned off to avoid getting this error

Received NACK = This error is detected when the controller board did not send back the ACKnowledge byte in the data byte header of the data being sent back to the computer and sends the Not ACKnowledge byte instead indicating the controller board communications is not functioning properly

Not STX error = This error is detected when the first byte received back from the controller board is not the Start Transmission byte which proceeds all data returned to the computer from the controller board.

USB Read error = This error is detected when the low level USB driver did not complete a USB read function properly.

USB Write error = This error is detected when the low level USB driver did not complete a USB write function properly.

USB Purge error = This error is detected when the low level USB driver did not complete a USB purge function properly. The USB read/write buffers are purged prior to any USB write function

WORKING WITH MULTIPLE CONTROLLER BOARDS

1). The **GetNumBoards** function reads back the number boards currently powered up and plugged into the USB communications interface. If you already know how many boards you have attached to the system, you don't necessarily need to call the **GetNumBoards** function, but it allows you to check that the intended boards are active on the communications bus before proceeding further in the top level application and helps to avoid generating errors with the **GetBoard_Handle** function.

2). All functions that communicate directly with the controller boards needs only the unique board/device handle for the target board to route the function call to the proper board connected to the computer system.

3). If the gskimUseVal set by the **SetGlobalSkimVal** / **SetGlobalSkimVal_Brd** functions, and/or the dacVhUseVal & dacVlUseVal set by the **SetDacReferences** / **SetDacReferences_Brd** functions are altered from their default values or it is desired to set them uniquely for each controller board, then the **SetGlobalSkimVal_Brd** and/or **SetDacReferences_Brd** functions should be called with the proper parameters for the board in question prior to calling the **DoCalibrate** function for that target controller board. Since the gskimUseVal, dacVhUseVal, & dacVlUseVal are generally left at their defaults to have the **DoCalibrate** function auto generate, then the **SetGlobalSkimVal_Brd** and/or **SetDacReferences_Brd** never need to be called, just as in the single controller board application programming.

4). If it is desired to have unique conversion reference numbers for each controller board when retrieving data with the **GetData** function, then when switching **GetData** calls between boards (by passing different handle numbers and parameters to **GetData**) the **SetConversionRef_Brd** function should be called to pass the conversion reference number to the DLL prior to calling the **GetData** function for the new target board. Since the conversion reference is rarely if ever changed, the **SetConversionRef_Brd** function never needs to be called unless it is desired to have a unique reference for each controller board connected to the system.

5). The unique device handle for each board connected to the system is determined by the **GetBoard_Handle** function. The **GetBoard_Handle** both returns the device handle for the target board and opens device communications to the target board based on the boardnumber parameter passed to **GetBoard_Handle**. The **GetBoard_Handle** only opens one controller board at a time for communications based on the passed variable "boardnumber". To get the unique serial number and/or device description for each board to relate to the boardnumber, use the GetQuery3 function to relate boardnumber, boardhandle, and board serial number and description.

6). The API calls are setup for calling only one board at a time, not multiple boards at once, thus it is up to the top level application developer to keep track of unique data and parameters for each board. For example, for the **GetData** function, the passed variables/arrays are: numAvg, arrayOrderFlag, dataSize, MxData, bPixMap. For this function call, usually only the MxData = multiplexer output data, and the bPixMap = bad pixel map, are unique to each board where the numavg, arrayOrderFlag, & dataSize could be common for all attached boards. For the unique data, if it is desired to keep the data between multiple boards unique within the top level application memory space, unique data arrays and variables for each controller board should be allocated in the top level application and used for each controller board function call based on the board communicated to with the board device handle. Using the same data array in, for example the **GetData** function, and calling board #1 to retrive the multiplexer output data, then using the same data array and calling **GetData** for board #2 will overwrite the board #1 multiplexer data in the data array with the output data from board #2. This is only an issue when running multiple boards in parallel controlling and retrieving data. If the top level application merely swaps between multiple attached boards and it doesn't matter if one board's data overwrites the previous data from another board, then obviously the same data arrays and variables can be used for all attached boards.

7). Each Multiplexer Controller Board opened with the **GetBoard_Handle** function using the boardnumber index should be closed individually using the **CloseDevice** function with it's board unique device handle. There is no mechanism in the API to close communications with all boards at once since each board has a unique identifier (device handle). So for example, having three boards open should have three **CloseDevice** calls prior to the top level application exiting. Not closing a controller board's communications without a board power cycle could cause an error when re-running the top level application when trying to open an already opened device.