ADICLib_256R5
SOFTWARE API
PROGRAMMING MANUAL

Version 2

FOR THE ADIC LINEAR ARRAY CONTROLLER R5 HARDWARE

(Not for Use with Prior Hardware Board Revisions)

A/DIC Inc.
740 Florida Central Pkwy
Suite 1024
Longwood, FL 32750
(407) 834-9981

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_256R5.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_256R5.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.

Algorithmn:

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_256R5.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_256R5.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_256R5.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 1.4
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_256R5.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_256R5.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_256R5.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 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.

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 GetData2 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_256R5.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 the LAdd (or Left Address) is the number of pixels to skip on the left side of the array. The valid 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 the RAdd (or Right Address) is the number of pixels to skip on the right side of the array. The valid 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.

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_256R5.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_256R5.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_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_256R5.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_256R5.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_256R5.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.

ReadTecADChannel

Note: New function for R5 interface electronics ONLY

Description:

Reads one of the analog monitor voltages available on the TE controller. The available analog signals are:

0) ITEC = Current through theTE cooler element
1) TMON = Measure of temperature stability
2) THRMB = Additional thermistor voltage (if installed)
3) VTEC = Voltage across the TE cooler element
4) VREF = TE controller reference voltage

Calling format:

ReadTecADChannel(devHandle, adchan, NumAvg, ADValue)

Visual C++ declare:

extern "C" __declspec(dllimport) long WINAPI ReadTecADChannel(FT_HANDLE lngHandle, long adchan, double NumAvg, double* ADValue);

Visual C++ call:

{*return value*} = ReadTecADChannel(devHandle, adchan, NumAvg, ADValue);

Visual BASIC declare:

Public Declare Function ReadTecADChannel Lib "ADICLib_256R5.dll" (ByVal lngHandle As Long, ByVal adchan as Long, ByVal NumAvg as Double, ByRef ADValue As Double) As Long

Visual BASIC call:

{*return value*} = ReadTecADChannel(devHandle, adchan, NumAvg, ADValue)

LabView Function Prototype:

long ReadTecADChannel(long arg1, long arg2, double arg3, double *arg4);

Parameters:

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

adchan = measurement channel to read, 0 to 4.

0 = ITEC = current through the TE cooler element
1 = TMON = measure of temperature stability
2 = THRMB = Additional thermistor voltage (if installed)
3 = VTEC = Voltage across the TE cooler element
4 = VREF = TE controller reference voltage

NumAvg = Number of samples to average together. Must be a number between 1 and 15, Where 1 represents no averaging

ADValue = Returned value of the analog signal in digital counts.

The A/D converter is 12 bits spanning a 0 to 5 Volt input range.

To convert counts to voltage use the equation: {AD Voltage} = ADValue * 5 / 4095.

To convert the returned voltages to values use the following equations

0) ITEC = [{AD Voltage} - VREF / 2] * 4 in Amps (positive for cooling, negative for heating)

1) TMON = [{AD Voltage} - VREF / 2] * 1000 / 1.57 in miliKelvin TMON is the voltage at the output of the bridge error amp. It provides an approximate look at how far from the set point temp the loop is controlling to. The sensitivity of the cooler loop is based on the value at -10C which is 1.57mV/mK at the TMON node. This is based on default circuit values.

2) THRMB - as of the authoring of this help file no second thermistor is installed

3) VTEC = [{AD Voltage} - VREF / 2] * 4.5 in Volts (positive for cooling, negative for heating)

4) VREF = {AD Voltage} in Volts

{*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.

ReadTecStatus

Note: New function for R5 interface electronics ONLY

Description:
Returns the status byte from the TE controller.

Calling format:
ReadTecStatus(devHandle, StatusVal)

Visual C++ declare:
extern "C" __declspec(dllimport) long WINAPI ReadTecStatus(FT_HANDLE lngHandle, long* StatusVal);

Visual C++ call:
{*return value*} = ReadTecStatus(devHandle, StatusVal);

Visual BASIC declare:
Public Declare Function ReadTecStatus Lib "ADICLib_256R5.dll" (ByVal lngHandle As Long, ByRef StatusVal As Long) As Long

Visual BASIC call:
{*return value*} = ReadTecStatus(devHandle, StatusVal)

LabView Function Prototype:
long ReadTecStatus(long arg1, long *arg2);

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

StatusVal = returned value has eight status bits 0 through 7, corresponding to the LSb through MSb of the StatusVal byte. As of this code version only two bits have meaning: Bit 0 indicates the TEC operation status which is either ON (bit 0 = 1) or OFF (bit 0 = 0). Bit 4 indicates the temperature stability of the cooler which iseither stable (bit 4 = 1) or not stable (bit 4 = 0). During cool down the stable bit will be clear and upon reaching the set point temperature the stable bit will go high.

{*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.

ReadTeSetPoint

Note: New function for R5 interface electronics ONLY

Description:
Reads the current set point temperature of the TE controller.

Calling format:
ReadTeSetPoint(devHandle, SPTval)

Visual C++ declare:
extern "C" __declspec(dllimport) long WINAPI ReadTeSetPoint(FT_HANDLE lngHandle, long* SPTval);

Visual C++ call:
{*return value*} = ReadTeSetPoint(devHandle, SPTval);

Visual BASIC declare:
Public Declare Function ReadTeSetPoint Lib "ADICLib_256R5.dll" (ByVal lngHandle As Long, ByRef SPTval As Long) As Long

Visual BASIC call:
{*return value*} = ReadTeSetPoint(devHandle, SPTval)

LabView Function Prototype:
long ReadTeSetPoint(long arg1, long *arg2);

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

SPTval = returned value representing the setpoint of the TE contoller The number is in the range 0 to 255, which corresponds to the 8 bit DAC setting used to control the temperature (see the **SetTESetPoint** command).

{*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.

RecallTeSetPoint

Note: New function for R5 interface electronics ONLY

Description:

Forces the TE controller to load the setpoint temperature stored in EEPROM and make it the current setpoint temperature. The new value is then returned.

Calling format:

RecallTeSetPoint(devHandle, SPTval)

Visual C++ declare:

extern "C" __declspec(dllimport) long WINAPI RecallTeSetPoint(FT_HANDLE lngHandle, long* SPTval);

Visual C++ call:

{*return value*} = RecallTeSetPoint(devHandle, SPTval);

Visual BASIC declare:

Public Declare Function RecallTeSetPoint Lib "ADICLib_256R5.dll" (ByVal lngHandle As Long, ByRef SPTval As Long) As Long

Visual BASIC call:

{*return value*} = RecallTeSetPoint(devHandle, SPTval)

LabView Function Prototype:

long RecallTeSetPoint(long arg1, long *arg2);

Parameters:

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

SPTval = returned value representing the setpoint of the TE contoller The number is in the range 0 to 255, which corresponds to the 8 bit DAC setting used to control the temperature (see the **SetTESetPoint** command).

{*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_256R5.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_256R5.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_256R5.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_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_256R5.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_256R5.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_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_256R5.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_256R5.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_256R5.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_256R5.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 tothe 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_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_256R5.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_256R5.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 timeThe 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_256R5.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 PICprocessor 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.

SetTESetPoint

Note: New function for R5 interface electronics ONLY

Description:

Sets the TEC setpoint temperature. This is the temperature that the TEC will attempt to cool down to. The set value controls a 256 step digital pot that controls the temperature. An equation relating the digital set value to temperature depends on installed component values. See device data sheet for more detailed information

Calling format:

SetTESetPoint(devHandle, SPTval)

Visual C++ declare:

extern "C" __declspec(dllimport) long WINAPI SetTESetPoint(FT_HANDLE lngHandle, long SPTval);

Visual C++ call:

{*return value*} = SetTESetPoint(devHandle, SPTval);

Visual BASIC declare:

Public Declare Function SetTESetPoint Lib "ADICLib_256R5.dll" (ByVal lngHandle As Long, ByVal SPTval As Long) As Long

Visual BASIC call:

{*return value*} = SetTESetPoint(devHandle, SPTval)

LabView Function Prototype:

long SetTESetPoint(long arg1, long arg2);

Parameters:

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

SPTval = Digital word representing the value of the TEC set point temperature. The valid range for SPTval is 0 to 255. With 0 being warmest and 255 being coldest. The equation for converting the digital word to temperature is a bit complex. The equations below are simplified based upon the installed component values and default thermistor.

Let S = ratio of the digital word to 255 => SPTValue / 255

Compute the thermistor variable

X = natural logarithm of the quantity { 3 * [{5*S + 2} / { 7 - 5*S }] }

The thermistor data sheet provides a polynomial in X for converting to temperature

Temperature = 1 / [A + B*X + C*X^2 + D*X^3 } in Kelvin

Where A = 0.0033538646
B = 0.0002565409
C = 0.0000019243889
D = 0.00000010969244

To convert to celcius just subtract 273.15 for the temperature computed above
{*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_256R5.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_256R5.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.

StoreTeSetPoint

Note: New function for R5 interface electronics ONLY

Description:

Stores the current TEC setpoint temperature into the EEPROM of the TEC controller. The TEC setpoint temperature stored in EEPROM is the temperature set point that will be reloaded on power up of the TEC controller.

Calling format:

StoreTeSetPoint(devHandle)

Visual C++ declare:

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

Visual C++ call:

{*return value*} = StoreTeSetPoint(devHandle);

Visual BASIC declare:

Public Declare Function StoreTeSetPoint Lib "ADICLib_256R5.dll" (ByVal lngHandle As Long) As Long

Visual BASIC call:

{*return value*} = StoreTeSetPoint(devHandle)

LabView Function Prototype:

long StoreTeSetPoint(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_256R5.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_256R5.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_256R5.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.

TecQuery

Note: New function for R5 interface electronics ONLY

Description:
Returns the firmware checksum of the TE controller board.

Calling format:
TecQuery(devHandle, teCksum)

Visual C++ declare:
```
extern "C" __declspec(dllimport) long WINAPI TecQuery(FT_HANDLE lngHandle, long*
    teCksum);
```

Visual C++ call:
{*return value*} = TecQuery(devHandle, teCksum);

Visual BASIC declare:
```
Public Declare Function TecQuery Lib "ADICLib_256R5.dll" (ByVal lngHandle As Long,
    ByRef teCksum As Long) As Long
```

Visual BASIC call:
{*return value*} = TecQuery(devHandle, teCksum)

LabView Function Prototype:
long TecQuery(long arg1, long arg2);

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

teCksum = Checksum of TE controller firmware. This value should be converted to HEX to check against published firmware revisions.

{*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.

TurnEnhCoolerOff

Note: New function for R5 interface electronics ONLY

Description:
Disables the enhanced TE controller output.

Calling format:
TurnEnhCoolerOff(devHandle)

Visual C++ declare:
```
extern "C" __declspec(dllimport) long WINAPI TurnEnhCoolerOff(FT_HANDLE
    lngHandle);
```

Visual C++ call:
{*return value*} = TurnEnhCoolerOff(devHandle);

Visual BASIC declare:
```
Public Declare Function TurnEnhCoolerOff Lib "ADICLib_256R5.dll" (ByVal lngHandle
    As Long) As Long
```

Visual BASIC call:
{*return value*} = TurnEnhCoolerOff(devHandle)

LabView Function Prototype:
long TurnEnhCoolerOff(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.

TurnEnhCoolerOn

Note: New function for R5 interface electronics ONLY

Description:
Enables the enhanced cooler controller output.

Calling format:
TurnEnhCoolerOn(devHandle)

Visual C++ declare:
```
extern "C" __declspec(dllimport) long WINAPI TurnEnhCoolerOn(FT_HANDLE
    lngHandle);
```

Visual C++ call:
{*return value*} = TurnEnhCoolerOn(devHandle);

Visual BASIC declare:
```
Public Declare Function TurnEnhCoolerOn Lib "ADICLib_256R5.dll" (ByVal lngHandle
    As Long) As Long
```

Visual BASIC call:
{*return value*} = TurnEnhCoolerOn(devHandle)

LabView Function Prototype:
long TurnEnhCoolerOn(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.

Status / Error Code Definitions

Almost all the DLL functions return a status indicator as the value of the function. The exceptions are; 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