

# **ArrayBoardR6\_x64 Software API Reference Manual**

**ArrayBoardR6\_Lib\_x64.dll**

**Version 2023.xx.xx**

**A/DIC Inc.**

**740 Florida Central Pkwy**

**Suite 1024**

**Longwood, FL 32750**

**(407) 834-9981**

## **INTRODUCTION**

The ArrayBoardR6\_Lib\_x64.DLL is a library of functions to communicate with the ArrayBoardR6 hardware which runs the 256 pixel PbS/PbSe linear detector arrays. This programming API is written in the Microsoft .NET Framework and the API functions can be interfaced to and called directly from top level client programs written in Microsoft Visual C# and Microsoft Visual BASIC. Functions in the DLL when coding the top level client application appear as any other built in function within the .NET Framework after being properly declared.

Additionally, the DLL functions can be called from Microsoft Visual C++ using the COM interoperability functionality which is much less straightforward than calling the DLL functions from Visual C# or Visual BASIC. This is because the .NET Framework is “Managed” code which runs on top the Microsoft Common Language Runtime (CLR), whereas Visual C++ is “Native” code and more processor specific. Managed and Native code binaries are not directly interchangeable and must communicate through an interface such as COM. The ArrayBoardR6\_Lib\_x64.DLL has been compiled with the “Make assembly COM-Visible” option turned on so that the DLL functions are visible through COM interoperability.

Managed code DLLs that run on top of the Microsoft CLR can also be called from top level LabView programs according to the LabView specifications, but this has not been tried and is beyond the scope of this document. Refer to the documentation that is included with the LabView development platform.

The ArrayBoardR6\_Lib\_x64.DLL has been tested with both Visual C# and Visual BASIC top level clients and verified to be fully functional being called from these languages.

Modifications to the underlying DLL code have been made in order to be properly callable from C# and C++ starting with version 1.30. Only version 1.30 and later of the DLL should be used for calling from C# and C++. DLL version number 2023.xx.xx and higher are compiled as 64 bit DLL for 64 bit code development.

To verify the version of the DLL you are using for top level application development, right mouse click on ArrayBoardR6\_Lib\_x64.dll in Windows Explorer, chose “Properties” in the dialog box that pops up, and then choose the “Details” tab.

## **CALLING THE DLL FUNCTIONS FROM VISUAL C#**

To call the ArrayBoardR6\_Lib\_x64.dll functions from Visual C# you must first add a reference to the DLL in your project and then declare the namespace using the following steps:

1. Right mouse click on the top level application project in the Solution Explorer in Visual Studio and choose “Add Reference”. Next choose the “Browse...” button in the lower right of the dialog box that pops up and point to the ArrayBoardR6\_Lib\_x64.DLL file and add the DLL as a reference to your application project
2. At the top of the C# code file that will be calling the DLL, add the line:

```
using ArrayBoardR6_Lib_x64;
```

3. Within the class module of the code file that will be calling the DLL functions add the following line, where “BoardComs” is any name the software developer wants to define:

```
ArrayBoardR6_x64 BoardComs = new ArrayBoardR6_x64 ();
```

4. Functions are then called with the following syntax:

```
result = BoardComs.DLLfunction(param1, param2, ...);
```

where result is an int data type

Since communications are opened with the Connect\_Boards function (see function reference) and all further function calls rely on open communications, all DLL functions that communicate with the array board hardware should be called from the same code class module using the “BoardComs” function prefix name from where it was declared. Opening the same board using the Connect\_Boards function from different modules at the same time will cause the DLL to loose data synchronization with the hardware and data communication errors will occur. To call the DLL functions from different code modules in a top level application, create your own function call “wrappers” within the board communication C# module that can be called from anywhere and then the wrapper function makes the actual local hardware function calls. For example, a wrapper function to get pixel data from the array board:

```
public void GetPixelData( int boardnumber, ref int[] PixelData)
{
    result = BoardComs.GetData(boardnumber, ref PixelData);
}
```

Here the function BoardComs.GetData is called from the same class module where BoardComs was declared, communications opened, and all DLL function calls are called from, but the GetPixelData function can be called from any code module or class (once properly declared) assuming that the board communications have already been opened with the Connect\_Boards function.

An alternate method of calling the array board functions across multiple code files is to use the “partial” class modifier when declaring classes and define the same namespace and classes across multiple code files. This method has not been tested by ADIC.

## **CALLING THE DLL FUNCTIONS FROM VISUAL BASIC**

To call the ArrayBoardR6\_Lib\_x64.dll functions from Visual BASIC you must first add a reference to the DLL in your project and then declare the namespace using the following steps:

1. Right mouse click on the top level application project in the Solution Explorer in Visual Studio and choose “Add Reference”. Next choose the “Browse...” button in the lower right of the dialog box that pops up and point to the ArrayBoardR6\_Lib\_x64.DLL file and add the DLL as a reference to your application project
2. At the top of the BASIC code file that will be calling the DLL, add the line:

```
Imports ArrayBoardR6_Lib_x64
```

3. Within the class module of the code file that will be calling the DLL functions add the following line, where “BoardComs” is any name the software developer wants to define:

```
Dim BoardComs as New ArrayBoardR6_x64
```

4. Functions are then called with the following syntax:

```
result = BoardComs.DLLfunction(param1, param2, ...)
```

where result is an integer data type

Since communications are opened with the Connect\_Boards function (see function reference) and all further function calls rely on open communications, all DLL functions that communicate with the array board hardware should be called from the same code class module using the “BoardComs” function prefix name from where it was declared. Opening the same board using the Connect\_Boards function from different modules at the same time will cause the DLL to loose data synchronization with the hardware and data communication errors will occur. To call the DLL functions from different code modules in a top level application, create your own function call “wrappers” within the board communication class module that can be called from anywhere and then the wrapper function makes the actual local hardware function calls. For example, a wrapper function to get pixel data from the array board:

```
public sub GetPixelData( byval boardnumber as integer, byref PixelData() as integer)
{
    result = BoardComs.GetData(boardnumber, PixelData)
}
```

Here the function BoardComs.GetData is called from the same class module where BoardComs was declared, communications opened, and all DLL function calls are called from, but the GetPixelData function can be called from any code module or class (once properly declared) assuming that the board communications have already been opened with the Connect\_Boards function.

An alternate method of calling the array board functions across multiple code files is to declare the interface public with the “Public” modifier within the class where the bulk of the array board communication functions will reside (like the code that initially opens communications with the Connect\_Boards function):

Public BoardComs as New ArrayBoardR6\_x64

To then call the array board functions from another code module, you would then call the DLL function with the following syntax:

*ClassName*.BoardComs.*DLLfunction*(param1, param2, ...)

Where:

*ClassName* = name of the public class where BoardComs was declared

ADIC has not tried that method of calling the array board functions across multiple classes/code modules.

## **CALLING THE DLL FUNCTIONS FROM VISUAL C++**

Visual C++ is considered “native” code and is more processor specific than the code architecture of the Microsoft CLR which is “managed” code. Native code and the managed code of the CLR do not interact directly and must communicate through an interface such as COM.

The ArrayBoardR6\_Lib\_x64.DLL has been compiled with the “Make assembly COM-Visible” option turned on using the GUID number “ccf7ff41-474c-4597-9cfd-766248702000” and the functions exposed through a public interface block following the example from Microsoft:

<https://msdn.microsoft.com/en-us/library/2w30w8zx%28v=vs.110%29.aspx?cs-save-lang=1&cs-lang=csharp#code-snippet-2>

The interface block name for the array board functions exposed through COM is “iArrayBoardR6\_x64”

Actual “how-to” implementing of calling COM exposed objects of managed CLR code in Visual C++ code is beyond the scope of this document and left for the C++ developer to work out.

A type library file (TLB file) is needed for Visual C++ developers to call the DLL, the C++ developer can either create a type library file using the Tlbexp.exe tool installed with the Visual Studio Tools using the DLL or use the one already created and included with the array board software distribution.

<https://msdn.microsoft.com/en-us/library/hfzzah2c%28v=vs.110%29.aspx>

Here are a few other reference links if needed:

An Overview of Managed/Unmanaged Code Interoperability:

<https://msdn.microsoft.com/en-us/library/ms973872.aspx>

Assembly Registration Tool (COM objects need to be registered with the registry to be called):

<https://msdn.microsoft.com/en-us/library/tzat5yw6%28v=vs.110%29.aspx>

### Notes on calling the DLL functions:

1. The Connect\_Boards function must be called prior to any of the other functions being called. All functions assume communications have been previously opened with the array board by the Connect\_Boards function.
2. All functions are written to not return until the data has been sent/retrieved from the array board in order to maintain data communication synchronization. The top level client application should not advance and call new array board communication functions until the current function has returned or else data handshaking will get out of order and a data communication error will occur. Often the error will clear itself and communications will re-sync back up, but it is highly recommended to wait for a function to return before proceeding with the next communication function call to the array board. Example code that will wait for the function to return is:

```
fresult = 99; // 99 is an arbitrary number

fresult = BoardComs.DLLFunction(param1, param2, ....);

while(fresult = 99)
{
    // do nothing but wait for function to return

    // 1 = success, 0 = fail
}
```

This type of coding is needed only if the top level client application doesn't wait for the function return in the original function call and advances to subsequent code statements.

3. **The “GetData\_ExtTrigger” and “GetDataFrames\_ExtTrigger” functions are a special case of communication with the array board.** Data retrieval from the array board is not initiated with a software function call but by an external timing input to the array board hardware. Each time a trigger is received by the array board the array board will send a 256 pixel stream of data to the computer via the USB. When in external trigger mode, the top level client application should be running in a loop to continually receive pixel data from the array board and **should not** send any other commands to the array board during external trigger mode or data synchronization errors will occur. To call other array board API functions, first disengage external trigger mode, send and receive the other function call/data, then re-engage external trigger mode. The LMAC top level client application supplied with the array board shows an example of this, when external data mode is enabled all other array board control functions are un-selectable.

## **Connect\_Boards**

### Description:

Finds and opens communications to array boards on the USB bus. Up to 8 boards are opened at once if connected to the USB

### Calling formats:

#### VB .NET call:

*{return value}* = BoardComs.Connect\_Boards()

#### Visual C# call:

*{return value}* = BoardComs.Connect\_Boards();

### Parameters:

*{return value}* = Number of array boards found and opened. Will return 0 if no boards are found on the USB bus. Data type is Integer

### Programming Notes:

This has to be the very first command to send to the array board as all other commands rely on having communications already opened and referenced by board index number.



## **Close\_Boards**

### Description:

Closes communication with **all** attached array controller boards.

### Calling formats:

#### VB .NET call:

*{return value}* = BoardComs.Close\_Boards()

#### Visual C# call:

*{return value}* = BoardComs.Close\_Boards();

### Parameters:

*{return value}* = Passed status variable from the function call. If the function completed successfully will return with a value of 1, otherwise a 0 is returned indicating the function failed. Data type is integer.

### Programming Notes:

This should be the last function sent to the array board to cleanly close and detach board communications from the USB bus. To reopen communications after they have been closed, recall the Connect\_Boards function.

## **GetData**

### Description:

Gets a single array scan of data from the system.

### Calling formats:

#### VB .NET call:

{*return value*} = BoardComs.GetData(BoardNumber, DataArray)

#### Visual C# call:

{*return value*} = BoardComs.GetData(BoardNumber, ref DataArray)

### Parameters:

{*return value*} = Passed status variable from the function call. If the function completed successfully status will return with a value of 1, otherwise a 0 will be returned, indicating a failure. Data type is integer

BoardNumber = the board number as found during the Connect\_Boards command.

Allowable values are 0 to 7. Data type is integer.

DataArray = Data array of detector pixel values returned by the GetData procedure.

This is a single dimension array that contains 256 elements. Data type is integer.

To get the data in volts, the default conversion value is 16000, so divide the DataArray values by 16000 and the result is in volts

### Programming Notes:

## **GetDataFrames**

### Description:

Grabs a block of frames under software control.

### Calling format:

#### VB .NET call:

{return value} = BoardComs.GetDataFrames(BoardNumber, Frames2Grab, BigDataArray, GrabTime)

#### Visual C# call:

{return value} = BoardComs. GetDataFrames (BoardNumber, Frames2Grab, ref BigDataArray, ref GrabTime);

### Parameters:

{return value} = Passed status variable from the function call. If the function completed successfully status will return with a value of 1, otherwise a 0 will be returned, indicating a failure. Data type is integer

BoardNumber = the board number as found during the Connect\_Boards command.

Allowable values are 0 to 7. Data type is integer.

Frames2Grab = Integer number of frames to grab. Valid range is 1 to 65535. Data type is integer

BigDataArray = Two dimensional data array large enough to hold all requested frames of data. Dimensionally the array is BigDataArray(frames, pixels), for example a grab of 1000 full array frames, the minimum array size would be

BigDataArray(999,255). Data type is integer. To get the data in volts, the default conversion value is 16000, so divide the BigDataArray values by 16000 and the result is in volts.

GrabTime = The time GetDataFrames took to execute is returned in this variable as elapsed seconds. Data type is double.

### Programming Notes:

- 1) Depending on the integration time the array may be operating in either a readout after integration or readout during integration. The board firmware decides which mode to be in to maximize throughput. In general the array will readout during integration for integration times greater than 650us.

## **GetData\_ExtTrigger**

### Description:

Gets a single array scan of data from the system; when using external triggering to control when integration begins.

### Calling format:

#### VB .NET call:

```
{return value} = BoardComs.GetData_ExtTrigger(BoardNumber, DataArray,  
    TriggerPolarity)
```

#### Visual C# call:

```
{return value} = BoardComs.GetData_ExtTrigger(BoardNumber, ref DataArray, ref  
    TriggerPolarity);
```

### Parameters:

{return value} = Passed status variable from the function call. If the function completed successfully status will return with a value of 1, otherwise a 0 will be returned, indicating a failure. Data type is integer

BoardNumber = the board number as found during the Connect\_Boards command. Allowable values are 0 to 7. Data type is integer.

DataArray = Integer data array of detector pixel values returned by the GetData\_ExtTrigger procedure. This is a single dimension array that contains a minimum of 256 elements. Data type is integer. To get the data in volts, the default conversion value is 16000, so divide the BigDataArray values by 16000 and the result is in volts.

TriggerPolarity = Trigger polarity to use for the data set, 1 = rising, 0 = falling. Allowed values are 0 and 1. Data type is integer. Useful when dual edge triggering is enabled.

### Programming Notes:

## **GetDataFrames\_ExtTrigger**

### Description:

Grabs a block of frames from the array when using external triggering mode to control when integration begins.

### Calling format:

#### VB .NET call:

```
{return value} = BoardComs.GetDataFrames_ExtTrigger(BoardNumber, Frames2Grab, BigDataArray, GrabTime, TriggerPolarityArray)
```

#### Visual C# call:

```
{return value} = BoardComs.GetDataFrames_ExtTrigger(BoardNumber, Frames2Grab, ref BigDataArray, ref GrabTime, ref TriggerPolarityArray);
```

### Parameters:

{return value} = Passed status variable from the function call. If the function completed successfully status will return with a value of 1, otherwise a 0 will be returned, indicating a failure. Data type is integer

BoardNumber = the board number as found during the Connect\_Boards command. Allowable values are 0 to 7. Data type is integer.

Frames2Grab = Integer number of frames to grab. Valid range is 1 to 65535. Data type is integer

BigDataArray = Two dimensional data array large enough to hold all requested frames of data. Dimensionally the array is BigDataArray(frames, pixels), for example a grab of 1000 full array frames, the minimum array size would be BigDataArray(999,255). Data type is integer. To get the data in volts, the default conversion value is 16000, so divide the BigDataArray values by 16000 and the result is in volts.

GrabTime = The time GetDataFrames\_ExtTrigger took to execute is returned in this variable as elapsed seconds. Data type is double.

TriggerPolarityArray = A single dimension array that defines the trigger polarity for each frame to be acquired. The trigger polarity to use for a data set is 1 = rising or 0 = falling. Allowed values for each array index are 0 and 1. The minimum size of TriggerPolarityData is a dimension of Frames2Grab. Data type is integer array. Useful when dual edge triggering is enabled.

### Programming Notes:

- 1) Depending on the integration time the array may be operating in either a readout after integration or readout during integration. The board firmware decides which mode to be in to maximize throughput. In general the array will readout during integration for integration times greater than 650us.

## **ConfigureTrigger**

### Description:

Configures the operation of the external triggered acquisition mode.

### Calling format:

#### VB .NET call:

{*return value*} = BoardComs.ConfigureTrigger (BoardNumber, TriggerPolarity, TriggerEdgeMode)

#### Visual C# call:

{*return value*} = BoardComs.ConfigureTrigger (BoardNumber, TriggerPolarity, TriggerEdgeMode);

### Parameters:

BoardNumber = the board number as found during the Connect\_Boards command.

Allowable values are 0 to 7. Data type is integer.

TriggerPolarity = A digital flag setting which edge to trigger on when in single edge mode, and setting which edge to capture first in dual edge mode. 0 = falling edge, 1 = rising edge. Allowed values are 0 and 1. Data type is integer.

TriggerEdgeMode = A digital flag setting dual edge trigger mode. 0 = single edge, 1 = dual edge. Allowed values are 0 and 1. Data type is integer

{*return value*} = Passed status variable from the function call. If the function completed successfully status will return with a value of 1, otherwise a 0 will be returned, indicating a failure. Data type is integer

### Programming Notes:

## **ConfigureTriggerDelay**

### Description:

Configures the operation of the triggered acquisition mode.

### Calling format:

#### VB .NET call:

{return value} = BoardComs.ConfigureTriggerDelay (BoardNumber, TrigDelayValue, TrigDelayMode)

#### Visual C# call:

{return value} = BoardComs.ConfigureTriggerDelay(BoardNumber, TrigDelayValue, TrigDelayMode);

### Parameters:

BoardNumber = the board number as found during the Connect\_Boards command.

Allowable values are 0 to 7. Data type is integer.

TrigDelayValue = Sets the delay between the trigger event and the start of integration.

Allowed values are  $0 \leq \text{TrigDelayValue} \leq 65535$ . Data type is integer. Note:

Trigger delay only functions when enabled. For TrigDelayValue = 0 the delay is

1.02us. For  $1 \leq \text{TrigDelayValue} \leq 65535$  then delay in us is given by:  $2.26\text{us} + (\text{TrigDelayValue} - 1) * 0.2\text{us}$ . From the equation give, the maximum delay is 13.11ms.

TrigDelayMode = A digital flag that enables or disables the external trigger delay mode.

0 = disabled, 1 = enabled. Allowed values are 0 and 1. Data type is integer.

{return value} = Passed status variable from the function call. If the function completed successfully status will return with a value of 1, otherwise a 0 will be returned, indicating a failure. Data type is integer

### Programming Notes:

## **SetExternalTriggerMode**

### Description:

Enables or disables the external trigger mode.

### Calling format:

#### VB .NET call:

```
{return value} = BoardComs.SetExternalTriggerMode (BoardNumber,  
    ExternalTriggerState)
```

#### Visual C# call:

```
{return value} = BoardComs.SetExternalTriggerMode (BoardNumber,  
    ExternalTriggerState);
```

### Parameters:

BoardNumber = the board number as found during the Connect\_Boards command.

Allowable values are 0 to 7. Data type is integer.

ExternalTriggerState = A digital flag that enables or disables the external trigger operaton. 0 = disabled, 1 = enabled. Allowed values are 0 and 1. Data type is integer.

{return value} = Passed status variable from the function call. If the function completed successfully status will return with a value of 1, otherwise a 0 will be returned, indicating a failure. Data type is integer

### Programming Notes:

- 1) Once the external trigger mode is enabled all data grabs from the array must be done using either GetData\_ExtTrigger, or GetDataFrames\_ExtTrigger running in a loop. Data sending from the array board is initiated by the external trigger and not a software command. GetData\_ExtTrigger and GetDataFrames\_ExtTrigger only receive data. The external trigger mode must be disabled before any other array board communications commands are sent to the board otherwise a data synchronization error will occur. See note 3 in the section "Notes on calling the DLL functions".
- 2) The trigger delay and delay mode should be configured prior to enabling the trigger mode.



## **SetIntegrationTime**

### Description:

Sets the charge well integration time of the array. The integration time is represented by a 16 bit number in the range 1 to 65535, where 65535 will give the maximum integration time and 1 will give the minimum integration time.

### Calling format:

#### VB .NET call:

*{return value}* = BoardComs.SetIntegrationTime(BoardNumber, IntegrationTime)

#### Visual C# call:

*{return value}* = BoardComs.SetIntegrationTime(BoardNumber, IntegrationTime);

### Parameters:

BoardNumber = the board number as found during the Connect\_Boards command.

Allowable values are 0 to 7. Data type is integer.

IntegrationTime = Digital word representing the value of the array integration time. The valid range for IntegrationTime is 1 to 65535. A value of 1 will give an integration time of approximately 4uS which is the minimum allowable integration time. Data type is integer

*{return value}* = Passed status variable from the function call. If the function completed successfully status will return with a value of 1, otherwise a 0 will be returned, indicating a failure. Data type is integer

### Algorithmn:

Integration time =  $3.2\mu\text{s} * (\text{IntegrationTime} - 1) + 4.025\mu\text{s}$

### Programming Notes:

## **SetWellDepth**

### Description:

Sets the integration charge depth. This is a global setting for all pixels of the array.

Valid charge well sizes are 1pF, 4pF, 7pF, 10pF, 11pF, 14pF, 17pF, and 20pF. The charge well sizes are set by an index number (see below).

### Calling format:

#### VB .NET call:

*{return value}* = BoardComs.SetWellDepth(BoardNumber, WellDepth)

#### Visual C# call:

*{return value}* = BoardComs.SetWellDepth(BoardNumber, WellDepth);

### Parameters:

BoardNumber = the board number as found during the Connect\_Boards command.

Allowable values are 0 to 7. Data type is integer.

WellDepth = 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

Allowed values are 0 to 7. Data type is integer.

*{return value}* = Passed status variable from the function call. If the function completed successfully status will return with a value of 1, otherwise a 0 will be returned, indicating a failure. Data type is integer

### Programming Notes:

## **SetPOTValue**

### Description:

Sets the bias voltage digital pots on the array board. One of four bias voltages may be changed at a time.

### Calling format:

#### VB .NET call:

*{return value}* = BoardComs.SetPOTValue(BoardNumber, POT2set, POTvalue)

#### Visual C# call:

*{return value}* = BoardComs.SetPOTValue(BoardNumber, POT2set, POTvalue);

### Parameters:

BoardNumber = the board number as found during the Connect\_Boards command.

Allowable values are 0 to 7. Data type is integer.

POT2set = Selection index to choose which bias pot is to be set by the command. The allowed values are 0 to 3. Data type is integer

POT2set = 0 => DAC VH bias voltage

POT2set = 1 => DAC VL bias voltage

POT2set = 2 => GSKIM bias voltage

POT2set = 3 => DETBIAS bias voltage

POTvalue = Digital word to set the 10 bit digital pot for a bias generator. The valid range is 0 to 1023. Data type is integer

*{return value}* = Passed status variable from the function call. If the function completed successfully status will return with a value of 1, otherwise a 0 will be returned, indicating a failure. Data type is integer

### Algorithmn:

DAC VH voltage =  $1.7857 * (\text{POTValue} / 1023) + 0.7143$  Volts

DAC VL voltage =  $1.7857 * (\text{POTValue} / 1023) + 0.7143$  Volts

GSKIM voltage =  $2.0833 * (\text{POTValue} / 1023) + 0.4167$  Volts

DETBIAS voltage =  $6 * (\text{POTValue} / 1023) + 6.053$  Volts

### Programming Notes:

Generally only the DETBIAS is set by the user, the other 3 biases are set automatically by the calibration routines.

## ReadBackPotValues

### Description:

Reads back the 4 current POT values from the PIC, useful for reading back new POT values after and on plane correction.

### Calling format:

#### VB .NET call:

{return value} = BoardComs.ReadBackPotValues (BoardNumber, DACVH\_val, DACVL\_val, GSKIM\_val, DETBIAS\_val)

#### Visual C# call:

{return value} = BoardComs.ReadBackPotValues (BoardNumber, ref DACVH\_val, ref DACVL\_val, ref GSKIM\_val, ref DETBIAS\_val);

### Parameters:

BoardNumber = the board number as found during the Connect\_Boards command.

Allowable values are 0 to 7. Data type is integer.

DACVH\_val = Returned value of the DAC\_VH pot setting. This is a 10 bit value with a range of 0 to 1023. Data type is integer

DACVL\_val = Returned value of the DAC\_VL pot setting. This is a 10 bit value with a range of 0 to 1023. Data type is integer

GSKIM\_val = Returned value of the GSKIM (global skim) pot setting. This is a 10 bit value with a range of 0 to 1023. Data type is integer

DETBias\_val = Returned value of the DETBIAS\_VH pot setting. This is a 10 bit value with a range of 0 to 1023. Data type is integer

{return value} = Passed status variable from the function call. If the function completed successfully status will return with a value of 1, otherwise a 0 will be returned, indicating a failure. Data type is integer

### Algorithmn:

DAC VH voltage =  $1.7857 * (\text{DACVH\_val} / 1023) + 0.7143$  Volts

DAC VL voltage =  $1.7857 * (\text{DACVL\_val} / 1023) + 0.7143$  Volts

GSKIM voltage =  $2.0833 * (\text{GSKIMVH\_val} / 1023) + 0.4167$  Volts

DETBias voltage =  $6 * (\text{DETBias\_val} / 1023) + 6.053$  Volts

### Programming Notes:

Generally only the DETBIAS is set by the user, the other 3 biases are set automatically by the calibration routines.

## **SetGlobalSkimVal**

### Description:

Sets the value of global skim to use during calibration when calibration is set to be performed with a user set global skim value (calibrate option #2, see the calibrate command).

### Calling format:

#### VB .NET call:

{*return value*} = BoardComs.SetGlobalSkimVal(BoardNumber, GlobalSkimValue)

#### Visual C# call:

{*return value*} = BoardComs.SetGlobalSkimVal(BoardNumber, GlobalSkimValue);

### Parameters:

BoardNumber = the board number as found during the Connect\_Boards command.

Allowable values are 0 to 7. Data type is integer.

GlobalSkimValue = Digital word to set the 10 bit digital pot for a bias generator. The valid range is 0 to 1023. Data type is integer

{*return value*} = Passed status variable from the function call. If the function completed successfully status will return with a value of 1, otherwise a 0 will be returned, indicating a failure. Data type is integer

### Algorithmn:

GSKIM voltage =  $2.0833 * (\text{GlobalSkimValue} / 1023) + 0.4167$  Volts

### Programming Notes:

Generally this function should not be needed as the best options to perform calibration is to either use no global skim or let the calibration routine automatically find the best global skim value

## CalibrateArray

### Description:

Calibrates the array by performing an on ROIC per pixel offset correction. Calibration involves setting the three biases, DACVH, DACVL, GSKIM to appropriate values and then determining the per pixel dac coefficients that will calibrate the array. The calibration routine can be used to calibrate with, without, or with a preset value of global skim.

### Calling format:

#### VB .NET call:

*{return value}* = BoardComs.CalibrateArray(BoardNumber, GSkim\_Option, CalTime)

#### Visual C# call:

*{return value}* = BoardComs.CalibrateArray(BoardNumber, GSkim\_Option, ref CalTime);

### Parameters:

BoardNumber = the board number as found during the Connect\_Boards command.

Allowable values are 0 to 7. Data type is integer.

GSkim\_Option = Flag that determines what will be done with global skim during calibration. Data Type is integer.

GSkim\_Option:

0 = No GSKIM, the gskim pot is set to 0

1 = GSKIM determined automatically

2 = Calibrate using value set by the SetGlobalSkimVal command

CalTime = Returned number of elapsed seconds. Data type is double

*{return value}* = Passed status variable from the function call. If the function completed successfully status will return with a value of 1, otherwise a 0 will be returned, indicating a failure. Data type is integer

### Programming Notes:

Array should be looking at a uniform "reference" such as a blackbody or some other uniform surface like a shutter blade in order to give the best offset correction results

## MarkBadPixels

### Description:

Sends the bad pixel map to the PIC processor on the array board. Upto 16 pixels can be marked bad.

### Calling format:

#### VB .NET call:

{return value} = BoardComs.MarkBadPixels (BoardNumber, NumbBadPixels, ReadoutDir, BadPixelArray)

#### Visual C# call:

{return value} = BoardComs.MarkBadPixels (BoardNumber, NumbBadPixels, ReadoutDir, ref BadPixelArray);

### Parameters:

BoardNumber = the board number as found during the Connect\_Boards command.

Allowable values are 0 to 7. Data type is integer.

NumbBadPixels = The number of bad pixels. Allowed values are 0 to 16. Data type is integer.

ReadoutDir = The readout direction of the array that matches the bad pixel array data.

0 = Left to Right, 1 = Right to Left. Data type is integer

BadPixelArray = 16 element array containing the pixel number for each bad pixel. Pixel numbers are from 0 to 255 for the 256 pixel array. Data type is integer

{return value} = Passed status variable from the function call. If the function completed successfully status will return with a value of 1, otherwise a 0 will be returned, indicating a failure. Data type is integer

### Programming Notes:

- 1) MarkBadPixels simply sets the bad pixel array into the PIC controller. Bad pixels shown or replaced is controlled by a separate function, HideBadPixels.
- 2) The directional descriptors, Left and Right, are based on viewing the ROIC die with the detector bond pads at the top and the control bond pads at the bottom.

## HideBadPixels

### Description:

Sets a flag to hide or show the bad pixels in the output data.

### Calling format:

#### VB .NET call:

{*return value*} = BoardComs.HideBadPixels (BoardNumber, HideFlag)

#### Visual C# call:

{*return value*} = BoardComs.HideBadPixels (BoardNumber, HideFlag);

### Parameters:

BoardNumber = the board number as found during the Connect\_Boards command.

Allowable values are 0 to 7. Data type is integer.

HideFlag = a 1 = hides bad pixels, a 0 = show bad pixels. Data type is integer.

{*return value*} = Passed status variable from the function call. If the function completed successfully status will return with a value of 1, otherwise a 0 will be returned, indicating a failure. Data type is integer

### Programming Notes:



## **ReadbackBadPixels**

### Description:

Reads back the bad pixels from the PIC's runtime memory.

### Calling format:

#### VB .NET call:

{*return value*} = BoardComs.ReadbackBadPixels (BoardNumber, NumbBadPixels, ReadoutDir, BadPixelArray)

#### Visual C# call:

{*return value*} = BoardComs.ReadbackBadPixels (BoardNumber, ref NumbBadPixels, ref ReadoutDir, ref BadPixelArray);

### Parameters:

BoardNumber = the board number as found during the Connect\_Boards command.

Allowable values are 0 to 7. Data type is integer.

NumbBadPixels = Returns the number of bad pixels. Allowed values are 0 to 16. Data type is integer.

ReadoutDir = The readout direction of the array that matches the bad pixel array data.

0 = Left to Right, 1 = Right to Left. Data type is integer

BadPixelArray = 16 element array containing the pixel number for each bad pixel. Pixel numbers are from 0 to 255. Data type is integer

{*return value*} = Passed status variable from the function call. If the function completed successfully status will return with a value of 1, otherwise a 0 will be returned, indicating a failure. Data type is integer

### Programming Notes:

- 1) The directional descriptors, Left and Right, are based on viewing the ROIC die with the detector bond pads at the top and the control bond pads at the bottom.

## **UpdateDACCoeff**

### Description:

Copies the current run time DAC coefficients from the PIC memory to the ROIC.

### Calling format:

#### VB .NET call:

*{return value}* = BoardComs.UpdateDACCoeff (BoardNumber)

#### Visual C# call:

*{return value}* = BoardComs.UpdateDACCoeff (BoardNumber);

### Parameters:

BoardNumber = the board number as found during the Connect\_Boards command.

Allowable values are 0 to 7. Data type is integer.

*{return value}* = Passed status variable from the function call. If the function completed successfully status will return with a value of 1, otherwise a 0 will be returned, indicating a failure. Data type is integer

### Programming Notes:

This function is generally called after the WriteDACCoeff function which loads coefficients from the computer to the PIC processor on the array board. Useful when it is desired to send a known set of coefficients to the ROIC.

## **ZeroDACCoeff**

### Description:

Zeros the ROIC DAC coefficients in the ROIC and the PIC runtime memory coefficients.

### Calling format:

#### VB .NET call:

{*return value*} = BoardComs.ZeroDACCoeff (BoardNumber)

#### Visual C# call:

{*return value*} = BoardComs.ZeroDACCoeff (BoardNumber);

### Parameters:

BoardNumber = the board number as found during the Connect\_Boards command.

Allowable values are 0 to 7. Data type is integer.

{*return value*} = Passed status variable from the function call. If the function completed successfully status will return with a value of 1, otherwise a 0 will be returned, indicating a failure. Data type is integer

### Programming Notes:

This function clears the offset correction coefficients in both the PIC runtime memory and the ROIC at the same time, the UpdateDACCoeff command is not needed.

## **ReadBackDACCoeff**

### Description:

Reads back the PIC runtime memory DAC coefficients to the PC.

### Calling format:

#### VB .NET call:

*{return value}* = BoardComs.ReadBackDACCoeff (BoardNumber, DACCoeff)

#### Visual C# call:

*{return value}* = BoardComs.ReadBackDACCoeff (BoardNumber, ref DACCoeff);

### Parameters:

BoardNumber = the board number as found during the Connect\_Boards command.

Allowable values are 0 to 7. Data type is integer.

DACCoeff = Array of DAC coefficients. Data type is byte. This should be a 256 element array, with each element representing a single pixel's coefficient. Pixel coefficients can take any value from 0 to 255.

*{return value}* = Passed status variable from the function call. If the function completed successfully status will return with a value of 1, otherwise a 0 will be returned, indicating a failure. Data type is integer

### Programming Notes:

This function reads back the offset correction coefficients from the array board PIC processor to the PC if it is desired to display or store the coefficient data on the PC.

## **WriteDACCoeff**

### Description:

Writes the ROIC offset correction DAC coefficients from the PC into the PIC runtime memory. Coefficients are sent to the ROIC with the UpdateDACCoeff command.

### Calling format:

#### VB .NET call:

*{return value}* = BoardComs.WriteDACCoeff (BoardNumber, DACCoeff)

#### Visual C# call:

*{return value}* = BoardComs.WriteDACCoeff (BoardNumber, ref DACCoeff);

### Parameters:

BoardNumber = the board number as found during the Connect\_Boards command.

Allowable values are 0 to 7. Data type is integer.

DACCoeff = Array of DAC coefficients. Data type is byte. This should be a 256 element array, with each element representing a single pixel's coefficient. Pixel coefficients can take any value from 0 to 255.

*{return value}* = Passed status variable from the function call. If the function completed successfully status will return with a value of 1, otherwise a 0 will be returned, indicating a failure. Data type is integer

### Programming Notes:

This function only writes the coefficients from the PC to the array board PIC processor. To update the coefficients into the ROIC use the UpdateDACCoeff after this command is run. The WriteDACCoeff function is used to restore a previously stored set of coefficients from the PC to the PIC-ROIC.

## **Set\_ROIC\_Readout\_Window**

### Description:

Sets the ROIC readout window size and readout direction.

### Calling format:

#### VB .NET call:

{*return value*} = BoardComs.Set\_ROIC\_Readout\_Window (BoardNumber, LeftAdr, RightAdr, ReadoutDir)

#### Visual C# call:

{*return value*} = BoardComs.Set\_ROIC\_Readout\_Window (BoardNumber, LeftAdr, RightAdr, ReadoutDir);

### Parameters:

BoardNumber = the board number as found during the Connect\_Boards command.

Allowable values are 0 to 7. Data type is integer.

LeftAdr = Left window address, this is the number of channels to leave off from the left side of the array during readout. Data type is integer. Allowed values are 0 to 127.

To readout all pixels the value of LeftAdr must be zero.

RightAdr = Right window address, this is the number of channels to leave off from the right side of the array during readout. Data type is integer. Allowed values are 0 to 127. To readout all pixels the value of RightAdr must be zero.

ReadoutDir = The readout direction of the array. 0 = Left to Right, 1 = Right to Left.

Data type is integer. The readout window is independent of direction - left side is always the left side, and right side is always the right side – the ReadoutDir simply swaps which side the readout starts on and the direction the readout proceeds in.

{*return value*} = Passed status variable from the function call. If the function completed successfully status will return with a value of 1, otherwise a 0 will be returned, indicating a failure. Data type is integer

### Programming Notes:

- 1) The directional descriptors, Left and Right, are based on viewing the ROIC die with the detector bond pads at the top and the control bond pads at the bottom.

## ReadBackSettings

### Description:

Reads back the current array settings from the PIC runtime memory to the PC.

### Calling format:

#### VB .NET call:

{return value} = BoardComs.ReadBackSettings (BoardNumber, SettingsArray)

#### Visual C# call:

{return value} = BoardComs.ReadBackSettings (BoardNumber, ref SettingsArray);

### Parameters:

BoardNumber = the board number as found during the Connect\_Boards command.

Allowable values are 0 to 7. Data type is integer.

SettingsArray = One dimensional settings array. The minimum size is 22 elements, the data type is integer. The array element meanings are:

- SettingsArray(0) = Left readout window address, LeftAdr
- SettingsArray(1) = Right readout window address, RightAdr
- SettingsArray(2) = Readout direction, ReadoutDir
- SettingsArray(3) = Charge well size, WellDepth
- SettingsArray(4) = Integration time, IntegrationTime
- SettingsArray(5) = DAC VH value, DACVH\_val
- SettingsArray(6) = DAC VL value, DACVL\_val
- SettingsArray(7) = Global Skim value, GSKIM\_val
- SettingsArray(8) = Detector bias value, DETBIAS\_val
- SettingsArray(9) = Trigger polarity, TriggerPolarity
- SettingsArray(10) = Trigger edge mode, TriggerEdgeMode
- SettingsArray(11) = Trigger delay count, TrigDelayValue
- SettingsArray(12) = Trigger delay mode, TrigDelayMode
- SettingsArray(13) = Hide bad pixels flag, HideFlag
- SettingsArray(14) = # of bad pixels, NumbBadPixels
- SettingsArray(15) = AtoD conversion factor
- SettingsArray(16) = GSkim value used for calibration\*
- SettingsArray(17) = DACVH value used for calibration\*
- SettingsArray(18) = DACVL value used for calibration\*
- SettingsArray(19) = Auto GSkim flag for calibration\*
- SettingsArray(20) = Auto DACVH flag for calibration\*
- SettingsArray(21) = Auto DACVL flag for calibration\*

{return value} = Passed status variable from the function call. If the function completed successfully status will return with a value of 1, otherwise a 0 will be returned, indicating a failure. Data type is integer

Programming Notes:

- 1) The last 6 entries in the settings array are marked with an asterisks, \*. These entries are for internal routine use and are not useful or needed by a top level application



## **RestoreFromEEPROM**

### Description:

Restores all the array settings from EEPROM memory on the array board to PIC runtime memory.

### Calling format:

#### VB .NET call:

*{return value}* = BoardComs.RestoreFromEEPROM (BoardNumber)

#### Visual C# call:

*{return value}* = BoardComs.RestoreFromEEPROM (BoardNumber);

### Parameters:

BoardNumber = the board number as found during the Connect\_Boards command.

Allowable values are 0 to 7. Data type is integer.

*{return value}* = Passed status variable from the function call. If the function completed successfully status will return with a value of 1, otherwise a 0 will be returned, indicating a failure. Data type is integer

### Programming Notes:

- 1) This command would need to be followed by a ReadBackSettings and ReadbackBadPixels in order for the PC to know the new settings restored from EEPROM.

## **StoreToEEPROM**

### Description:

Stores the current PIC runtime settings to the EEPROM on the array board.

### Calling format:

#### VB .NET call:

{*return value*} = BoardComs.StoreToEEPROM (BoardNumber)

#### Visual C# call:

{*return value*} = BoardComs.StoreToEEPROM (BoardNumber);

### Parameters:

BoardNumber = the board number as found during the Connect\_Boards command.

Allowable values are 0 to 7. Data type is integer.

{*return value*} = Passed status variable from the function call. If the function completed successfully status will return with a value of 1, otherwise a 0 will be returned, indicating a failure. Data type is integer

### Programming Notes:

## **ReadBoardInfo**

### Description:

Reads identification and interface data from the selected board.

### Calling format:

#### VB .NET call:

```
{return value} = BoardComs.ReadBoardInfo (BoardNumber, VID, PID,  
    DeviceDescription, Manufacturer, BoardSerial, FirmwareChecksum, BoardRev,  
    TECInstalled)
```

#### Visual C# call:

```
{return value} = BoardComs.ReadBoardInfo (BoardNumber, ref VID, ref PID, ref  
    DeviceDescription, ref Manufacturer, ref BoardSerial, ref FirmwareChecksum, ref  
    BoardRev, ref TECInstalled);
```

### Parameters:

BoardNumber = the board number as found during the Connect\_Boards command.

Allowable values are 0 to 7. Data type is integer.

VID = the USB vendor ID in hex format. Data type is string.

PID = the USB product ID in hex format. Data type is string.

DeviceDescription = USB device description string. Data type is string.

Manufacturer = USB manufacturer description string. Data type is string.

BoardSerial = the array board serial number. Data type is integer.

FirmwareChecksum = the checksum of the installed firmware. Data type is integer

BoardRev = the array PC board revision number. Data type is integer

TECInstalled = TE controller installed flag. 1 = TE controller installed, 0 = TE controller not installed. Data type is integer. Allowed values are 0 and 1.

{return value} = Passed status variable from the function call. If the function completed successfully status will return with a value of 1, otherwise a 0 will be returned, indicating a failure. Data type is integer

### Programming Notes:

## EEPROM\_UserData\_Write

### Description:

Allows the user to save arbitrary information to unused EEPROM area on the array board. A total of 2048 bytes of data may be written

### Calling format:

#### VB .NET call:

{return value} = BoardComs.EEPROM\_UserData\_Write (BoardNumber, StartAddress, NumBytes, ByteArray)

#### Visual C# call:

{return value} = BoardComs.EEPROM\_UserData\_Write (BoardNumber, StartAddress, NumBytes, ref ByteArray);

### Parameters:

BoardNumber = the board number as found during the Connect\_Boards command.

Allowable values are 0 to 7. Data type is integer.

StartAddress = The starting address of the EEPROM to write to. Allowed range is 0 to 2047. Data type is integer

NumBytes = Number of bytes of data to be written to EEPROM. Allowed range is 1 to 2048. Data type is integer

ByteArray = data to be written. Each element of the 1D ByteArray must be a value between 0 to 255 (inclusive). Data type is byte.

{return value} = Passed status variable from the function call. If the function completed successfully status will return with a value of 1, otherwise a 0 will be returned, indicating a failure. Data type is integer

### Programming Notes:

- 1) StartAddress + NumBytes can not be > 2048. Any data that StartAddress + NumBytes that is beyond 2048 is ignored and not written to the EEPROM
- 2) The size of ByteArray must at least be the same size as the NumBytes to be written or larger.
- 3) EEPROM writes take many milliseconds per block. This command can take a while to complete. It is important to wait for the command to return before sending more commands to the array board

## EEPROM\_UserData\_Read

### Description:

Reads user data from unused block of EEPROM memory on the array board. A total of 2048 bytes are available to read.

### Calling format:

#### VB .NET call:

{return value} = BoardComs.EEPROM\_UserData\_Read (BoardNumber, StartAddress, NumBytes, ByteArray)

#### Visual C# call:

{return value} = BoardComs.EEPROM\_UserData\_Read (BoardNumber, StartAddress, NumBytes, ref ByteArray);

### Parameters:

BoardNumber = the board number as found during the Connect\_Boards command.

Allowable values are 0 to 7. Data type is integer.

StartAddress = The starting address of the EEPROM to read from. Allowed range is 0 to 2047. Data type is integer

NumBytes = Number of bytes of data to read from EEPROM. Allowed range is 1 to 2048. Data type is integer

ByteArray = 1D array to store the read EEPROM data. Each element of the 1D

ByteArray will be a value between 0 to 255 (inclusive). Data type is byte.

{return value} = Passed status variable from the function call. If the function completed successfully status will return with a value of 1, otherwise a 0 will be returned, indicating a failure. Data type is integer

### Programming Notes:

- 1) StartAddress + NumBytes can not be > 2048. Any data that StartAddress + NumBytes that is beyond 2048 is ignored and not read from the EEPROM
- 2) The size of ByteArray must at least be the same size as the NumBytes to be read or larger.

## **TECooler\_Power**

### Description:

Turns the TE Cooler power on or off

### Calling format:

#### VB .NET call:

*{return value}* = BoardComs.TECooler\_Power (BoardNumber,TEPowerState)

#### Visual C# call:

*{return value}* = BoardComs.TECooler\_Power (BoardNumber,TEPowerState);

### Parameters:

BoardNumber = the board number as found during the Connect\_Boards command.

Allowable values are 0 to 7. Data type is integer.

TEPowerState = a 1 turns on the cooler, a 0 turns it off. Allowed values are 0 and 1, data type is integer.

*{return value}* = Passed status variable from the function call. If the function completed successfully status will return with a value of 1, otherwise a 0 will be returned, indicating a failure. Data type is integer

### Programming Notes:

## TECooler\_ReadA2D\_Data

### Description:

Reads back operational performance values from the TEC

### Calling format:

#### VB .NET call:

{return value} = BoardComs.TECooler\_ReadA2D\_Data (BoardNumber, Data2Read, NumAvgs, ReadBackData)

#### Visual C# call:

{return value} = BoardComs.TECooler\_ReadA2D\_Data (BoardNumber, Data2Read, NumAvgs, ref ReadBackData);

### Parameters:

BoardNumber = the board number as found during the Connect\_Boards command.

Allowable values are 0 to 7. Data type is integer.

Data2Read = Selection index to pick which piece of data to read back from TEC.

Allowed values are 0 to 4 (inclusive). Data type is integer.

0 = ITEC – a voltage related to the current in the TE element

1 = TMON – a voltage that provides temperature stability information

2 = not applicable

3 = VTEC – a voltage related to the voltage across the TE element

4 = VREF – the internal reference voltage for the TE controller

NumAvgs = The number of readings to average together before reporting a value.

Allowed values are 0 to 15. Note a value of 0 or 1 is equivalent to no averaging.

Data type is integer.

ReadBackData = The read back data. This data is based upon a 12bit A/D. Data type is integer.

{return value} = Passed status variable from the function call. If the function completed successfully status will return with a value of 1, otherwise a 0 will be returned, indicating a failure. Data type is integer

### Algorithmn:

VREF\_Meas = VREF \* 5 / 4095 (this voltage is used for the other computations)

ITEC\_Meas = { (ITEC \* 5 / 4095) – VREF\_Meas / 2 } \* 4 (in Amps)

VTEC\_Meas = { (VTEC \* 5 / 4095) – VREF\_Meas / 2 } \* 4.5 (in Volts)

TMON\_Meas = { (TMON \* 5 / 4095) – VREF\_Meas / 2 } \* 1000 / 1.57 (in mK from set point)

### Notes:

- 1) Positive values of VTEC\_Meas mean the TE is cooling
- 2) TMON\_Meas is a delta measurement, providing the delta temperature from the set point temperature. TMON\_Meas originates from the amplified/buffered output of the difference amplifier that is reading the thermistor bridge circuit.

## **TECooler\_Set\_TEC\_Setpoint**

### Description:

Sets the TEC setpoint temperature; which is the temperature the cooler will attempt to reach and maintain.

### Calling format:

#### VB .NET call:

*{return value}* = BoardComs.TECooler\_Set\_TEC\_Setpoint (BoardNumber, TECSetpoint)

#### Visual C# call:

*{return value}* = BoardComs.TECooler\_Set\_TEC\_Setpoint (BoardNumber, TECSetpoint);

### Parameters:

BoardNumber = the board number as found during the Connect\_Boards command.

Allowable values are 0 to 7. Data type is integer.

TECSetpoint = Value of the setpoint POT value, 0 to 255 (inclusive). (see TEC Setpoint Computation for conversion to Kelvin or Celsius). Data type is integer

*{return value}* = Passed status variable from the function call. If the function completed successfully status will return with a value of 1, otherwise a 0 will be returned, indicating a failure. Data type is integer

### Programming Notes:



## TEC Setpoint Computation

Computing the temperature from the setpoint integer requires detailed knowledge of the particular thermistor, and the circuit component values used to measure the temperature for the controller loop.

### Compute Temperature from Setpoint:

The setpoint temperature is controlled by an 8 bit digital pot in one leg of a bridge circuit. Buried in the equations provided are the circuit topology and values, if any of these are changed, either by the user, or by the manufacturer, then a modified set of equation coefficients will be needed. The equations provided below are based on the construction details as of July 2016.

Variables / Coefficients:

- 1) TECSetpoint = 8 bit digital word loaded to digital pot, values range from 0 to 255 (inclusive)
- 2) TEPotRatio = TECSetpoint / 255, this is the ratio of the pot. This has values of 0 to 1 as is a Double data type.
- 3) Thermistor Coefficient A = 0.0033538646, Data type is Double
- 4) Thermistor Coefficient B = 0.0002565409, Data type is Double
- 5) Thermistor Coefficient C = 0.0000019243889, Data type is Double
- 6) Thermistor Coefficient D = 0.00000010969244, Data type is Double
- 7) dtmp = temporary variable, data type is Double

$$\text{dtmp} = \text{LN} \{ 3 * [ (5 * \text{TEPotRatio} + 2) / (7 - 5 * \text{TEPotRatio}) ] \}$$

Note: LN is natural logarithm

$$\text{TempK} = 1 / ( A + B * \text{dtmp} + C * \text{dtmp}^2 + D * \text{dtmp}^3 ); \text{ in Kelvin}$$

$$\text{TempC} = \text{TempK} - 273.15; \text{ in Celsius}$$

## **TECooler\_Read\_TEC\_Setpoint**

### Description:

Reads the current TEC setpoint temperature POT value (see TEC Setpoint Computation for conversion to Kelvin or Celsius)

### Calling format:

#### VB .NET call:

{*return value*} = BoardComs.TECooler\_Read\_TEC\_Setpoint (BoardNumber, TECSetpoint)

#### Visual C# call:

{*return value*} = BoardComs.TECooler\_Read\_TEC\_Setpoint (BoardNumber, ref TECSetpoint);

### Parameters:

BoardNumber = the board number as found during the Connect\_Boards command.

Allowable values are 0 to 7. Data type is integer.

TECSetpoint = Value of the setpoint (see TEC Setpoint Computation for conversion to Kelvin or Celsius). Data type is integer

{*return value*} = Passed status variable from the function call. If the function completed successfully status will return with a value of 1, otherwise a 0 will be returned, indicating a failure. Data type is integer

### Programming Notes:

## **TECooler\_Store\_TEC\_Setpoint**

### Description:

Stores the current TEC setpoint into the EEPROM memory of the TEC board's PIC processor

### Calling format:

#### VB .NET call:

{*return value*} = BoardComs.TECooler\_Store\_TEC\_Setpoint (BoardNumber)

#### Visual C# call:

{*return value*} = BoardComs.TECooler\_Store\_TEC\_Setpoint (BoardNumber);

### Parameters:

BoardNumber = the board number as found during the Connect\_Boards command.

Allowable values are 0 to 7. Data type is integer.

{*return value*} = Passed status variable from the function call. If the function completed successfully status will return with a value of 1, otherwise a 0 will be returned, indicating a failure. Data type is integer

### Programming Notes:

## **TECooler\_Restore\_TEC\_Setpoint**

### Description:

Restores the TEC setpoint temperature from the non volatile memory in the TEC board to the TEC board current operating TEC setpoint and then returns that information to the PC.

### Calling format:

#### VB .NET call:

*{return value}* = BoardComs.TECooler\_Restore\_TEC\_Setpoint (BoardNumber, TECSetpoint)

#### Visual C# call:

*{return value}* = BoardComs.TECooler\_Restore\_TEC\_Setpoint (BoardNumber, ref TECSetpoint);

### Parameters:

BoardNumber = the board number as found during the Connect\_Boards command.

Allowable values are 0 to 7. Data type is integer.

TECSetpoint = Value of the setpoint (see TEC Setpoint Computation for conversion to Kelvin or Celsius). Data type is integer

*{return value}* = Passed status variable from the function call. If the function completed successfully status will return with a value of 1, otherwise a 0 will be returned, indicating a failure. Data type is integer

### Programming Notes:

## **TECooler\_Read\_TEC\_Status**

### Description:

Reads the current TEC board status

### Calling format:

#### VB .NET call:

```
{return value} = BoardComs.TECooler_Read_TEC_Status (BoardNumber,  
    TEC_PowerState, TEC_CoolingState, TEC_StabilityState)
```

#### Visual C# call:

```
{return value} = BoardComs.TECooler_Read_TEC_Status (BoardNumber, ref  
    TEC_PowerState, ref TEC_CoolingState, ref TEC_StabilityState);
```

### Parameters:

BoardNumber = the board number as found during the Connect\_Boards command.

Allowable values are 0 to 7. Data type is integer.

TEC\_PowerState = Flag for the current state of the TEC power. 1 = Cooler is on, 0 = Cooler is off. Data type is integer. The TEC controller board is always powered, the power state referred to here is the power state of the output drivers to the actual TEC element. When the power state is off the H bridge driving the TE element is off and the TEC element is not being driven.

TEC\_CoolingState = Flag that indicates the direction that the TEC element is being driven (cooling or heating). TEC\_CoolingState => 1 = cooling, 0 = heating. Data type is integer.

TEC\_StabilityState = Flag that indicates the stability of the TE element temperature. TEC\_StabilityState => 1 = stable, 0 = not stable. Stability is determined within the TEC board's processor by monitoring the TEMPGOOD line from the TE controller chip, when the TEMPGOOD has been indicated for 5 to 6 seconds the stability state is set to stable. The stability state is monitored internally by the controller's processor to detect thermal runaway and to shut the cooler down automatically in case that occurs. So periodically calling this function will let you know if the cooler output control is still active

{return value} = Passed status variable from the function call. If the function completed successfully status will return with a value of 1, otherwise a 0 will be returned, indicating a failure. Data type is integer

### Programming Notes:

## TECooler\_Query

### Description:

Queries the TEC board's processor to see if it is there, and that comms are working.

This is a left over from the board development process.

### Calling format:

#### VB .NET call:

*{return value}* = BoardComs.TECooler\_Query (BoardNumber)

#### Visual C# call:

*{return value}* = BoardComs.TECooler\_Query (BoardNumber);

### Parameters:

BoardNumber = the board number as found during the Connect\_Boards command.

Allowable values are 0 to 7. Data type is integer.

*{return value}* = Passed status variable from the function call. If the function completed successfully status will return with a value of 1, otherwise a 0 will be returned, indicating a failure. Data type is integer

### Programming Notes:

## SetOutputTriggerLevel

### Description:

This command sets the digital state of the output trigger. The output trigger consists of a single open drain driver (NC7SZ05) which can operate from 1.65 to 5.5V of supply. This makes it compatible with 1.8V, 3.3V, and 5V logic. The user supplies the voltage to the driver, and also the pull up resistor for the open drain output. The drivers output can be set high or low with the SetOutputTriggerLevel command if the SyncTrig2Int option is set to 0. When the SyncTrig2Int option is set to 1, the output trigger level parameter is ignored and the output trigger signal from the array board replicates the ROIC integration control pulse.

### Calling format:

#### VB .NET call:

{return value} = BoardComs.SetOutputTriggerLevel (BoardNumber, OTrigLevel, SyncTrig2Int)

#### Visual C# call:

{return value} = BoardComs.SetOutputTriggerLevel (BoardNumber, OTrigLevel, SyncTrig2Int);

### Parameters:

BoardNumber = the board number as found during the Connect\_Boards command.

Allowable values are 0 to 7. Data type is integer.

OTrigLevel = the output digital trigger level state. Allowed values are 1 (for a logic high) and 0 for a logic low. Data type is integer.

SyncTrig2Int = output trigger mode selection option. 0 = output trigger level is set to the level specified by the OTrigLevel input parameter. 1 = the output trigger replicates the ROIC integration pulse. Allowed values are 1 and 0. Data type is integer.

{return value} = Passed status variable from the function call. If the function completed successfully status will return with a value of 1, otherwise a 0 will be returned, indicating a failure. Data type is integer

### Programming Notes:

If SyncTrig2Int is set to "1" to have the output trigger signal replicate the ROIC integration pulse then you do not need to keep calling the SetOutputTriggerLevel function, it will stay in the mode to have the output trigger replicate the ROIC integration pulse until SyncTrig2Int is set to 0 or the board is power cycled.

## **SetFastReadoutFlag**

### Description:

This command sets a flag that puts the PIC processor into fast readout mode, turning off the A/D conversion and running the ROIC at it's maximum readout rate of 4MHz. The feature is implemented for a particular customer that wished higher speed and was able to do their own external digitization of the analog pixel stream, taking the analog pixel stream signal directly off the array board.

### Calling format:

#### VB .NET call:

*{return value}* = BoardComs.SetFastReadoutFlag (BoardNumber, FastReadFlag)

#### Visual C# call:

*{return value}* = BoardComs.SetFastReadoutFlag (BoardNumber, FastReadFlag);

### Parameters:

BoardNumber = the board number as found during the Connect\_Boards command.

Allowable values are 0 to 7. Data type is integer.

FastReadFlag = Flag that sets fast readout mode, 1 = turn fast readout mode on, 0 = turn fast readout mode off. Allowed values are 0 and 1, data type is integer.

*{return value}* = Passed status variable from the function call. If the function completed successfully status will return with a value of 1, otherwise a 0 will be returned, indicating a failure. Data type is integer

### Programming Notes:

This function only needs to be used if bypassing all digitization and pixel data read back from the array board. When fast read out mode is enabled, no pixel data is transmitted back from the USB interface and all GetDataxxxx commands should not be used.



## **SetConversionRef**

### Description:

This command has been obsoleted and should not be used.

## **SetErrorDialogState**

### Description:

This command operates entirely within the DLL and does not do anything to the operation of the array as it does not send/receive any communications to the array board. The SetErrorDialogState command sets or clears a flag that allows internal communications errors to be presented on the screen via a pop-up message box. The default state within the DLL is to display communications errors when they occur with a pop-up msg box. This function allows the user to suppress the pop-up message box when communication errors occur. Generally showing the error message boxes should be left on. A well written top level application will not cause communication errors to occur under normal operating conditions.

### Calling format:

#### VB .NET call:

*{return value}* = BoardComs.SetErrorDialogState (ErrorEnable)

#### Visual C# call:

*{return value}* = BoardComs.SetErrorDialogState (ErrorEnable);

### Parameters:

ErrorEnable = When a 1 the errors are shown, when 0 the errors are suppressed.

Allowed values are 0 and 1. Data type is integer.

*{return value}* = Passed status variable from the function call. If the function completed successfully status will return with a value of 1, otherwise a 0 will be returned, indicating a failure. Data type is integer

### Programming Notes:

## **Flush\_USB\_ReadBuffer**

### Description:

Discards any data that is cached in the USB read buffer

### Calling format:

#### VB .NET call:

{*return value*} = Flush\_USB\_ReadBuffer (BoardNumber)

#### Visual C# call:

{*return value*} = Flush\_USB\_ReadBuffer (BoardNumber);

### Parameters:

BoardNumber = the board number as found during the Connect\_Boards command.

Allowable values are 0 to 7. Data type is integer.

{*return value*} = Passed status variable from the function call. If the function completed successfully status will return with a value of 1, otherwise a 0 will be returned, indicating a failure. Data type is integer

### Programming Notes:

## **TECooler\_Read\_Errorcode**

### Description:

This command is obsolete and no longer supported.

## **WriteBoardInfo**

### Description:

This command is for internal use by A/DIC only.