Astronomical
Research
Cameras, Inc.

ARC API User Manual

Version 3.5

April 2013

General Information

Arc API Features

The API contains software for Windows, Linux & Mac OSX host environments. This software is provided for development of custom applications and has the following features:

- Set of C++ classes used to command host interface and controller boards.
- C-interface for each C++ class.
- Libraries for device control, image deinterlacing, display (via DS9), statistics, and saving via FITS and TIFF.

Version Compatibility

Only the newest version should be used for new applications. It is important that all components are of the same version. The older Versions 2.0 and 3.0 are not compatible with version 3.5.

Customer Support

Prior to contacting ARC customer support, please be prepared to provide the following information:

- Host Operating System and version
- ARC API version
- ARC device (ARC-63/64 PCI, ARC-66/67 PCIe, etc)
- Detailed description of your problem
- Any error messages that may have been reported
- Steps to recreate the problem

If you have comments, corrections, or suggestions, you may contact ARC at:

Address: Astronomical Research Cameras, Inc.
2247 San Diego Ave Ste 135, CA 92110

Phone: 619-278-0865

Web: http://www.astro-cam.com

Getting Started

Development Tools

Various tools were used to build the libraries included in the ARC API. There are many compatible alternative tools available for the various build environments. Customers are free to use their own preferred sets of compatible development tools; however, ARC has only verified the tools listed below and, as a result, cannot support tools not listed here. The API libraries were built using the following development environments:

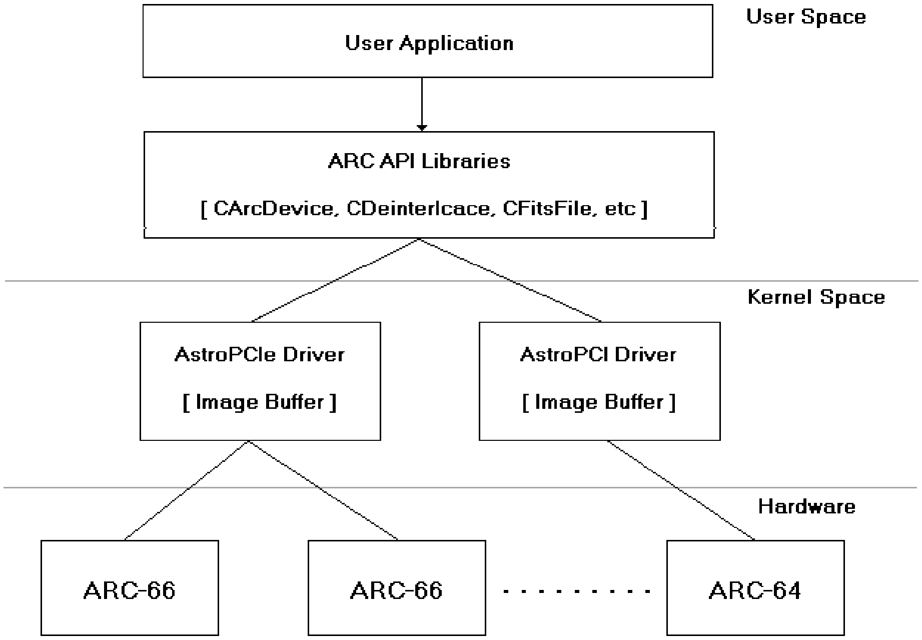
Windows**:** Microsoft Visual C/C++ 2012

Linux: *GCC (4.4.7)*

MAC OS X: XCode

API Architectural Overview

The ARC API has three main components, the kernel drivers, user API and user applications. The following figure demonstrates the various components and how they fit together. The API is provided to handle most of the low-level functionality so users can concentrate on building their applications.



API Libraries

The ARC API libraries are provided to communicate with the ARC device drivers. When an API function is called by an application, the API library handles the call and translates it to an I/O control message that is sent to the driver. Once the driver completes the request, control returns to the API and then back to the calling application; typically returning a reply to the application.

The API consists of a set of classes, from which multiple ARC devices can be accessed and used. The API covers most features of all ARC devices and controllers, such as sending commands, receiving replies and DMA access.

The API libraries are implemented as Dynamically Linked Libraries (DLL); .dll on windows, .so on linux, and .dylib on mac os x. Applications linked with these libraries will attempt to load the DLL when started; therefore, the DLLs must be found somewhere in the system path. DLLs are typically placed in a system directory (System32 on windows, /usr/local/lib on linux/mac), but can be placed anywhere as long as their location is in the system path environment variable.

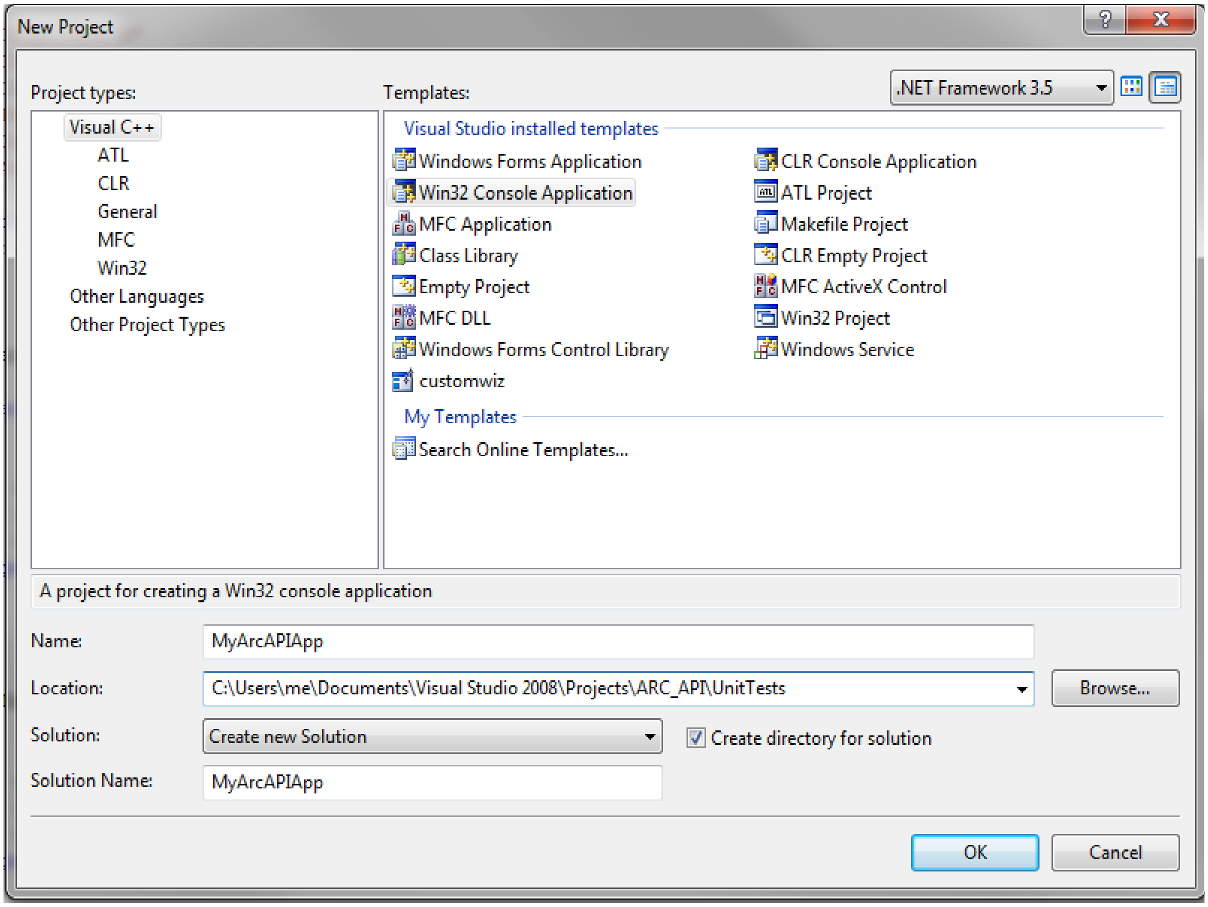
Device Driver

The appropriate ARC device driver must be installed before the API can access any device. The appropriate driver for your system can be downloaded from www.astro-cam.com. Be sure to download the driver version that matches your hardware and API. If no matching driver version exists, then download the latest version.

On some systems, such as linux, the driver may need to be recompiled before use. See the device driver READ_ME file for more details.

Building a Windows Application Using the API

The first step in creating a Windows host application is to create a Microsoft Project File. Typically, a *Win32 Console application* is used to create a project, but any C or C++ project, such as *MFC AppWizard*, is compatible with the API. The following figure demonstrates the new project dialog.



A project for creating a Win32 console application

Name: MyArcAPIApp

Location: C:\Users\me\Documents\Visual Studio 2008\Projects\ARC_API\UnitTests

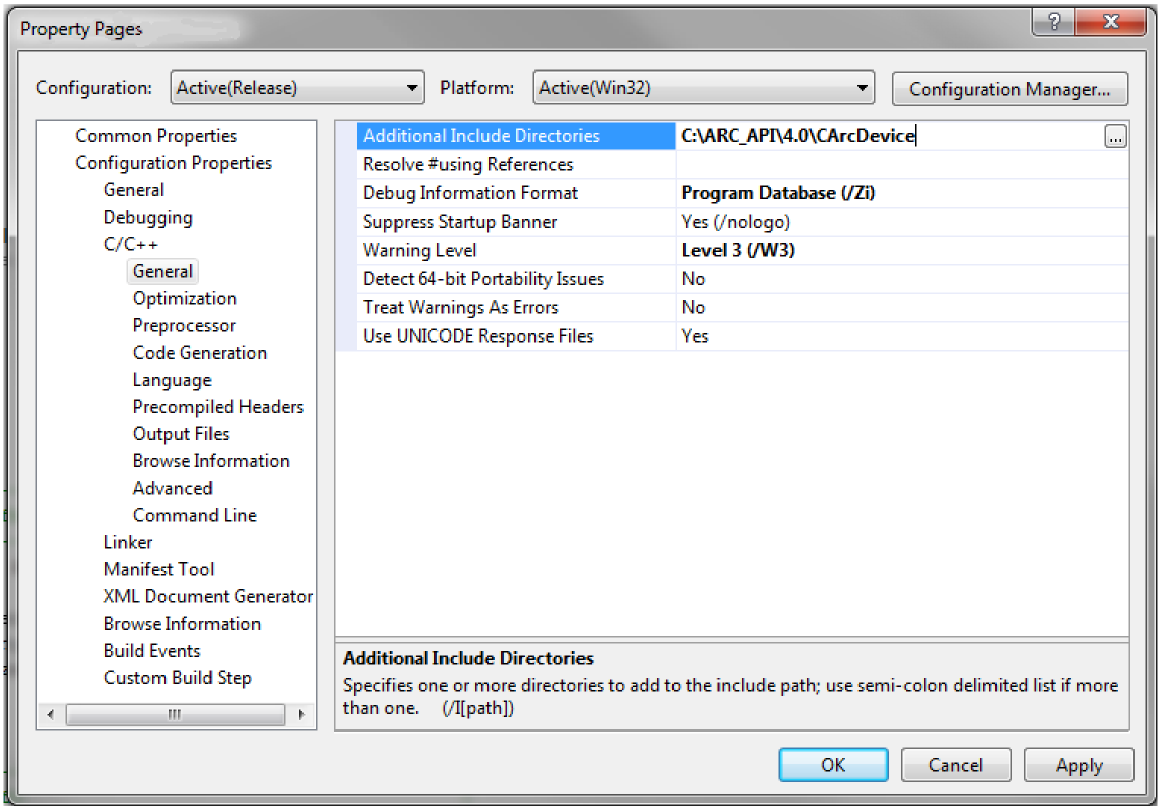
Solution: Create new Solution ☑ Create directory for solution

Solution Name: MyArcAPIApp

Once the project has been opened, source code can be written and inserted into the project. Before an application can be built successfully, however, the steps below must be completed. The following figures demonstrate a typical Visual C or C++ project that is configured for the ARC API. Be sure to include the directory for each library you wish to use.

- **Add the API include directories**

  This ensures that the development tools refer to and can find the correct version of the API C/C++ header files. In Visual C/C++, for example, the directory is specified in the *Options* dialog, as shown in the following figure.



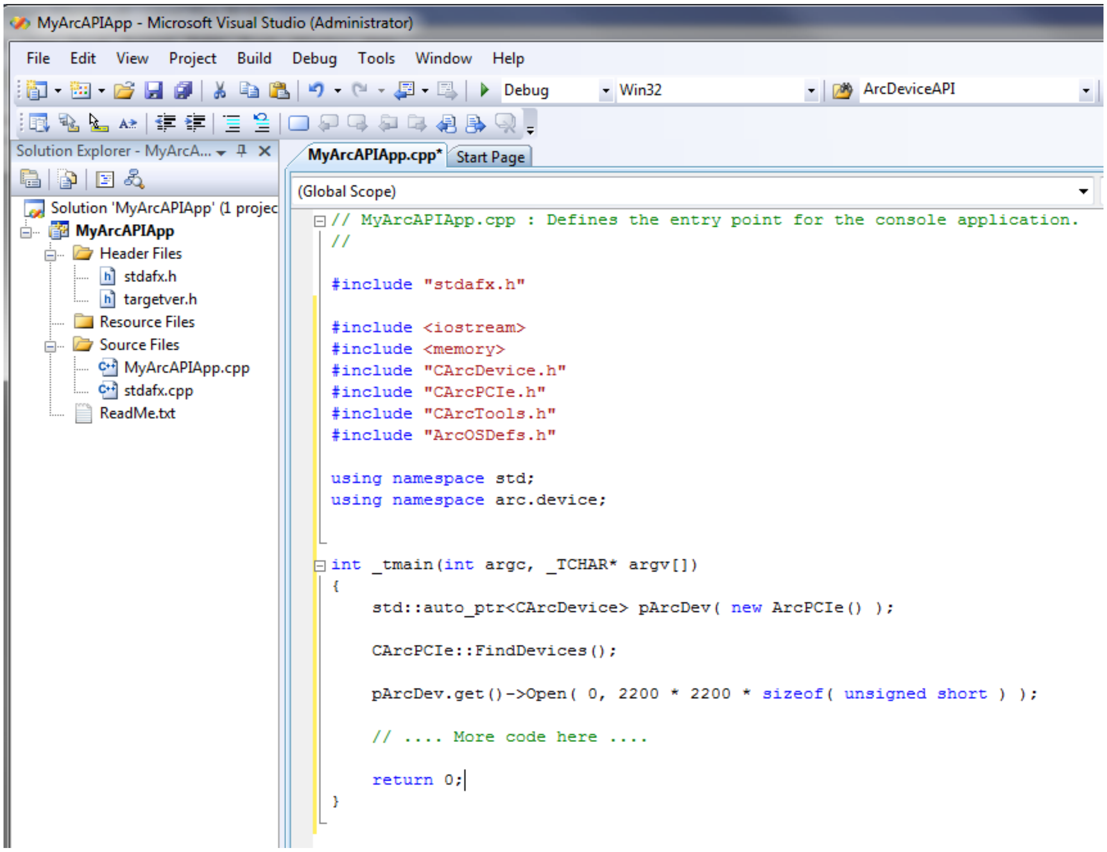
Additional Include Directories	C:\ARC_API\4.0\CArcDevice
Resolve #using References	
Debug Information Format	**Program Database (/Zi)**
Suppress Startup Banner	Yes (/nologo)
Warning Level	**Level 3 (/W3)**
Detect 64-bit Portability Issues	No
Treat Warnings As Errors	No
Use UNICODE Response Files	Yes

Additional Include Directories

Specifies one or more directories to add to the include path; use semi-colon delimited list if more than one. (/I[path])

- **Add the API headers**

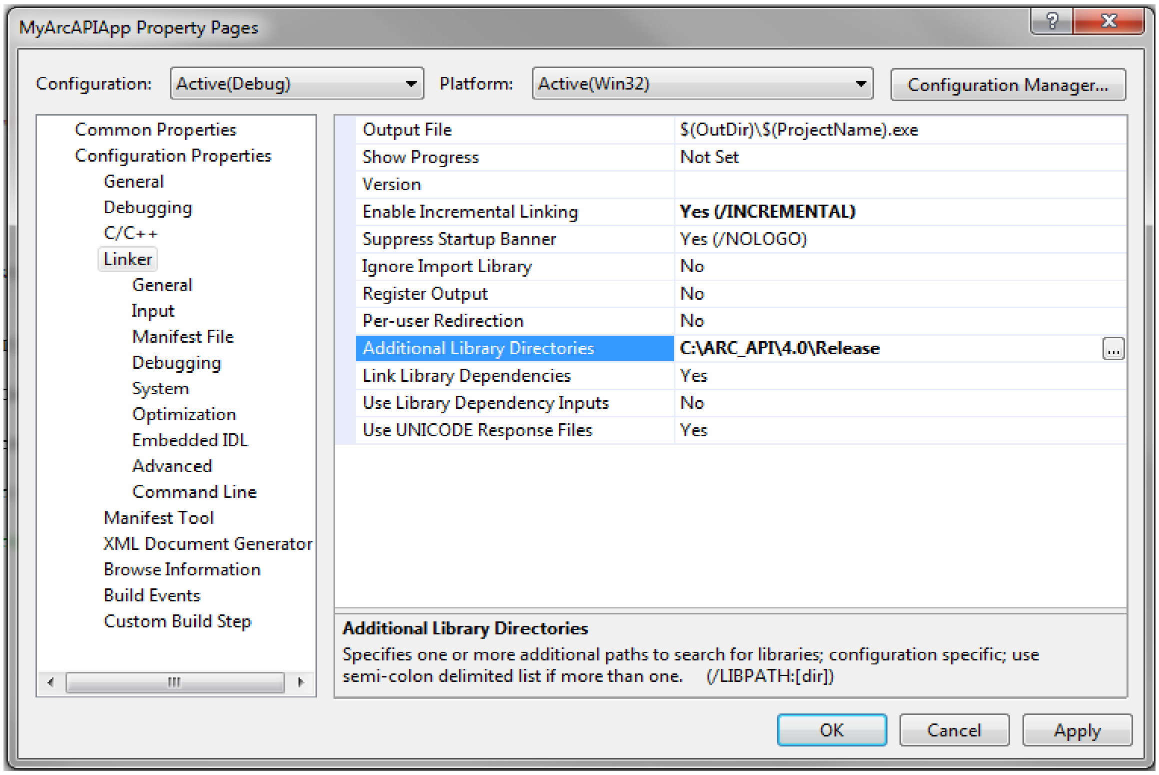
The necessary API header files must be included in your source code.



```
// MyArcAPIApp.cpp : Defines the entry point for the console application.
//

#include "stdafx.h"

#include <iostream>
#include <memory>
#include "CArcDevice.h"
#include "CArcPCIe.h"
#include "CArcTools.h"
#include "ArcOSDefs.h"

using namespace std;
using namespace arc.device;

int _tmain(int argc, _TCHAR* argv[])
{
    std::auto_ptr<CArcDevice> pArcDev( new ArcPCIe() );

    CArcPCIe::FindDevices();

    pArcDev.get()->Open( 0, 2200 * 2200 * sizeof( unsigned short ) );

    // .... More code here ....

    return 0;
}
```

- **Add the API libraries and directories**

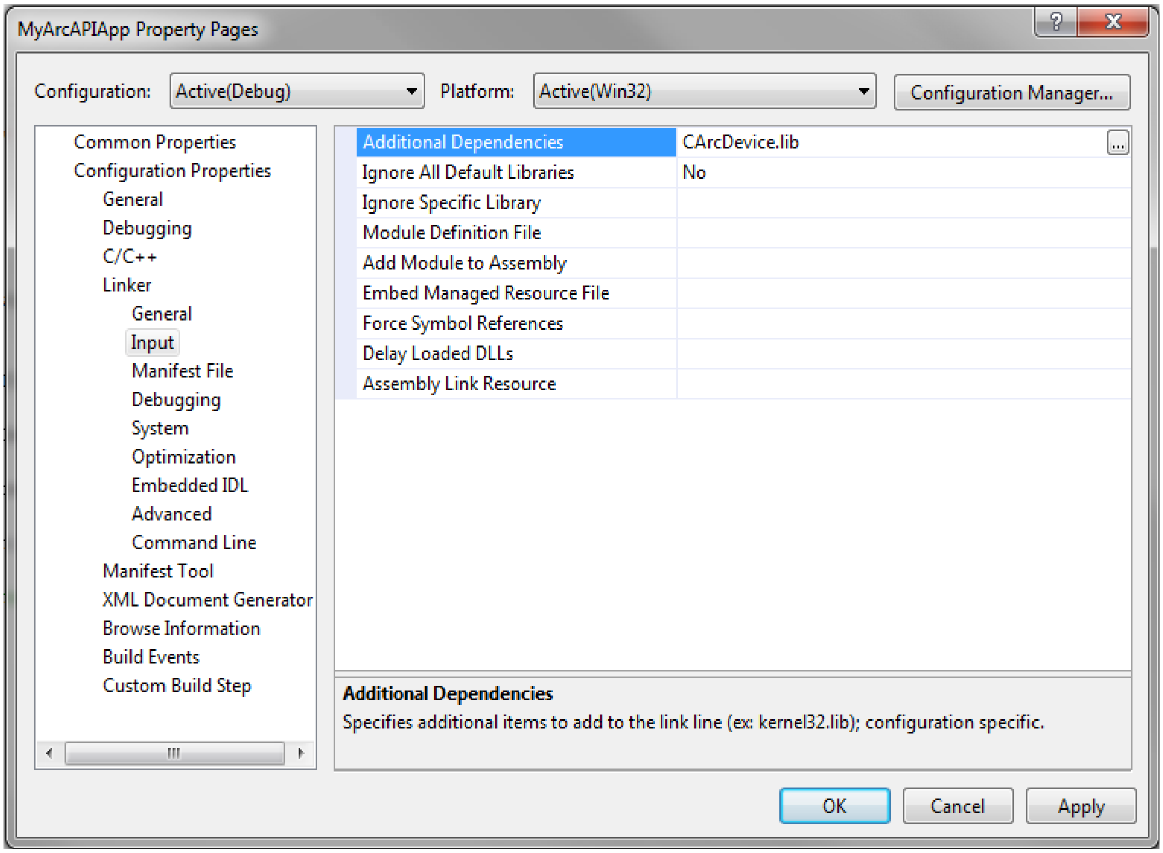
When the application is launched, the API DLLs will automatically be loaded by Windows. The library file as well as the location must be set in the Linker section of the properties window. The API library files are provided in the *<API_Install_Dir>\ARC_API\version\Release* directory.



Configuration: Active(Debug)   Platform: Active(Win32)

Output File	$(OutDir)\$(ProjectName).exe
Show Progress	Not Set
Version	
Enable Incremental Linking	**Yes (/INCREMENTAL)**
Suppress Startup Banner	Yes (/NOLOGO)
Ignore Import Library	No
Register Output	No
Per-user Redirection	No
Additional Library Directories	**C:\ARC_API\4.0\Release**
Link Library Dependencies	Yes
Use Library Dependency Inputs	No
Use UNICODE Response Files	Yes

Additional Library Directories
Specifies one or more additional paths to search for libraries; configuration specific; use semi-colon delimited list if more than one. (/LIBPATH:[dir])



Configuration: Active(Debug)   Platform: Active(Win32)

Additional Dependencies	CArcDevice.lib
Ignore All Default Libraries	No
Ignore Specific Library	
Module Definition File	
Add Module to Assembly	
Embed Managed Resource File	
Force Symbol References	
Delay Loaded DLLs	
Assembly Link Resource	

Additional Dependencies
Specifies additional items to add to the link line (ex: kernel32.lib); configuration specific.

- Declare required **namespace**:

Each of the ARC API libraries is contained within its own namespace under the top-level “arc” namespace. See the library header files to determine the namespaces, which are listed here:

CArcDevice library namespace – arc::device

CArcImage library namespace – arc::image

CArcFitsFile library namespace – arc::fits

CArcTiffFile library namespace – arc::tiff

CArcDeinterlace library namespace – arc::deinterlace

CArcDisplay library namespace – arc::display

These namespaces must be included within your code by using either the “using” directive or prefixing all class names with the full namespace.

To include the namespace globally with the “using” directive:

```
#include “CArcDevice.h”
#include “CArcFitsFile.h”

using namespace arc::device;
using namespace arc::fits;
```

To prefix class names:

```
#include “CArcDevice.h”

arc::device::CArcDevice* pArcDev;
. . . .
```

Building a Linux Application Using the API

The following command line can be used to compile an application called *prog.cpp* against the *CArcDevice* and *CArcDeinterlace* libraries (assuming the API's installed in /xxx):

```
g++ --std=c++11 -I/xxx/ARC_API/3.5/CArcDevice/src -I/xxx/ARC_API/3.5/CArcDeinterlace/src
-L/xxx/ARC_API/3.5/Release prog.cpp -lCArcDevice -lCArcDeinterlace
```

The libraries make use of C++11, which requires the `--std=c++11` option. If this option doesn't work, then try the older `-std=c++0x` option. Some C++11 features won't be implemented in compilers older than GCC version 4.6.

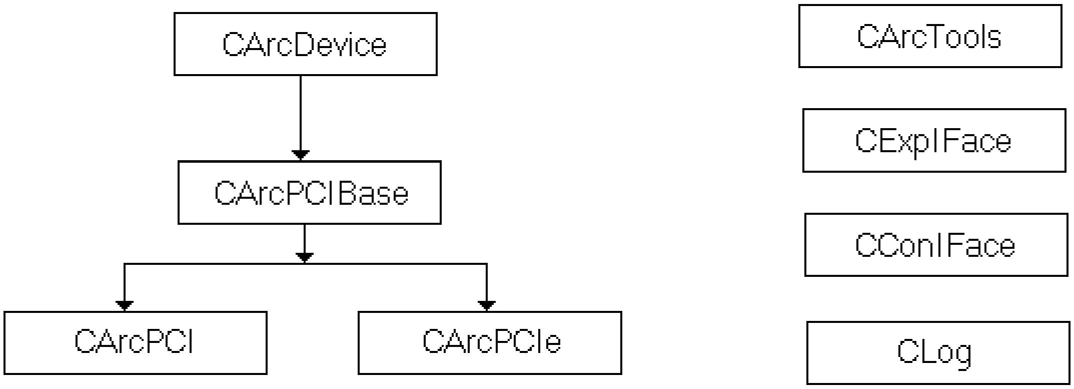
Building a Mac OS X Application Using the API

The libraries can be included in an xcode project to build an application. The library header file paths need to be set in the “Header Search Paths” build settings option. Likewise, the library paths need to be set in the “Library Search Paths” option. Finally, the libraries themselves need to be included in the library project. This can be done by using the “Add Files to .. “ menu option.

How to Access Devices (Class Structure)

Device access is accomplished using one of two classes, depending on which devices you wish to support. Current support is for PCI (ARC-63/64) and PCIe (ARC-66/67). All ARC controllers are accessed via these classes.

ARC API Class Hierarchy



Class Namespace

Each C++ class within an API library is contained within its own namespace. All classes within a specific library are generally contained within the same namespace. For example, most classes within the CArcDevice library are contained within the “arc::device” namespace; there are a few exceptions, such as the CArcTools class, which is in the arc namespace only. The top-level namespace across all ARC API library classes is “arc”. The following namespace structure exits within the libraries:

CArcDevice library namespace – arc & arc::device
CArcImage library namespace – arc::image
CArcFitsFile library namespace – arc::fits
CArcTiffFile library namespace – arc::tiff
CArcDeinterlace library namespace – arc::deinterlace
CArcDisplay library namespace – arc::display

These namespaces must be included within your code by calling the “using” directive or prefixing all class names with the full namespace.

To include the namespace globally with the “using” directive:

```
#include “CArcDevice.h”
#include “CArcFitsFile.h”

using namespace arc::device;
using namespace arc::fits;
```

To prefix class names:

```
#include “CArcDevice.h”

arc::device::CArcDevice* pArcDev;
. . . .
```

FindDevices Class Method

The CArcPCIe and CArcPCI classes contain a set of static methods, one of which (*FindDevices()*) must be called before any device can be opened (accessed). The *FindDevices()* method searches the system for installed devices (drivers) of the appropriate type. Any devices found is maintained in a list that can be used to open a device. The set of class methods used for this purpose are: *FindDevices()*, *DeviceCount()*, and *GetDeviceStringList()*. See the *CArcPCIe* and *CArcPCI* method descriptions for details.

Before calling the *Open()* method on any device, the *FindDevices()* method must be called first. For example:

```
#include “CArcDevice.h”
#include “CArcPCIe.h”

using namespace arc::device;

. . . .

CArcPCIe::FindDevices();

if ( CArcPCIe::DeviceCount() > 0 )
{
      CArcDevice* pArcDev = new CArcPCIe();

      pArcDev->Open( 0, dBufferSize );
}

. . . .
```

Device Class Instantiation

The *CArcDevice* class is abstract and cannot be instantiated directly and should be used to instantiate one of the sub-classes *CArcPCI* or *CArcPCIe*. Using the *CArcDevice* class provides the ability to easily switch between devices without code changes.

For example, the following shows how to access a PCIe device:

```
#include “CArcDevice.h”
#include “CArcPCIe.h”

using namespace arc::device;

. . . .

CArcDevice* pArcDev = new CArcPCIe();

pArcDev->Open( 0, dBufferSize );

. . . .
```

To access a PCI device:

```
#include “CArcDevice.h”
#include “CArcPCI.h”

using namespace arc::device;

. . . .

CArcDevice* pArcDev = new CArcPCI();

pArcDev->Open( 0, dBufferSize );

. . . .
```

To support both PCI and PCIe, user applications can reassign a *CArcDevice* to the desired board during runtime.

For example, suppose the user may select the device as a parameter (*std::string sDev*) that is passed into the user application. The proper device may then be selected as follows:

```
#include “CArcDevice.h”
#include “CArcPCIe.h”
#include “CArcPCI.h”

using namespace arc::device;

. . . .

CArcDevice* pArcDev = NULL;

if ( sDev == “PCIe” )
{
      pArcDev = new CArcPCIe();
}
else
{
      pArcDev = new CArcPCI();
}

pArcDev->Open( 0, dBufferSize );

. . . .
```

If only one device is required, PCIe or PCI, then the appropriate class may be instantiated directly. However, it is still recommended that the *CArcDevice* class be used instead.

For example, if only PCIe will be used, the following is allowed:

```
#include “CArcDevice.h”
#include “CArcPCIe.h”

using namespace arc::device;

. . . .

CArcPCIe cArcDev;

cArcDev.Open( 0, dBufferSize );

. . . .
```

but this is preferred:

```
#include “CArcDevice.h”
#include “CArcPCIe.h”

using namespace arc::device;

. . . .

CArcDevice* pArcDev = new CArcPCIe();

pArcDev->Open( 0, dBufferSize );
```

Header Listing

CArcDevice.h

CArcDevice class definition. Primary class that should be used for device access. This is an abstract class that must point to one of the sub-classes *CArcPCI* or *CarcPCIe*.

CArcPCIBase.h

CArcPCIBase class definition. Provides PCI(e) configuration space access only. Abstract class; not useful for user applications.

CArcPCIe.h

PCIe class definition. Provides PCIe device access and can be instantiated directly by user applications.

CArcPCI.h

PCI class definition. Provides PCI device access and can be instantiated directly by user applications.

CExpIFace.h

CExpIFace class definition. Abstract interface class that provides exposure callbacks for user applications. A user defined class extending this interface can be passed into the *CArcDevice::Expose()* method for elapsed time and pixel count information.

CConIFace.h

CConIFace class definition. Abstract interface class that provices continuous readout callbacks for user applications. A user defined class extending this interface can be passed into the *CArcDevice::Continuous()* method for frame count information.

ArcDefs.h

Command, reply, board id's, command parameter and controller configuration parameter constants.

ArcOSDefs.h

Generic re-mappings of system functions for cross-platform compatibility. Not useful for user applications.

CArcTools.h

CArcTools class definition. Defines a general set of utility methods for string and command conversions and throwing descriptive exceptions.

CLog.h

Clog class definition. Defines logger class used internally by *CArcDevice* to store commands. Used for debugging only.

PCIRegs.h

PCI(e) configuration space constant and macro definitions. Used by *CArcPCIBase* class and not useful for user applications.

Reg9056.h

PLX PCIe register definitions. Used by *CArcPCIe* class and not useful for user applications.

TempCtrl.h

Temperature calibration constants and default values.

CStringList.h

CStringList class definition. Used internally and not exported by library. User applications cannot access this class.

AstroPCIeGUID.h and astropciGUID.h

PCIe and PCI Windows driver id files respectively. Used by *CArcPCIe* and *CArcPCI* classes to identify device drivers. Not useful for user applications.

CArcDevice Methods

This section documents details of the methods available through the CArcDevice class (see CArcDevice.h). These methods define the standard interface for the sub-device classes (CArcPCIe and CArcPCI). The following is a list of these methods; with details to follow on subsequent pages:

NOTE: The term “device” refers to an ARC-64 (PCI) or ARC-66/67 (PCIe board). The term “controller” refers to the camera controller (ARC-22, etc).

CArcDevice Method Name	C Interface Name	Description
ToString	ArcDevice_ToString	Returns a device specific string
IsOpen	ArcDevice_IsOpen	Returns true if a device is open and can be accessed
Open	ArcDevice_Open ArcDevice_Open_I ArcDevice_Open_II	Opens a device
Close	ArcDevice_Close	Closes a device
Reset	ArcDevice_Reset	Resets a device
MapCommonBuffer	ArcDevice_MapCommonBuffer	Maps the kernel device buffer so the user application may access it
UnMapCommonBuffer	ArcDevice_UnMapCommonBuffer	Removes access to a mapped kernel device buffer
FillCommonBuffer	ArcDevice_FillCommonBuffer	Fill the kernel device buffer with a specific value
CommonBufferVA	ArcDevice_CommonBufferVA	Returns the kernel device buffer virtual address. Applications use this address to access the buffer.
CommonBufferPA	ArcDevice_CommonBufferPA	Returns the kernel device buffer physical address. For informational purposes only.
CommonBufferSize	ArcDevice_CommonBufferSize	Returns the size, in bytes, of the kernel device buffer
GetId	ArcDevice_GetId	Returns the controller id or 0 if no id exists
GetStatus	ArcDevice_GetStatus	Returns the device status (meaning is device dependent)
ClearStatus	ArcDevice_ClearStatus	Clears the device status
Set2xFOTransmitter	ArcDevice_Set2xFOTransmitter	Toggles 2x fiber transmission if available (most devices do not support this)
LoadDeviceFile	ArcDevice_LoadDeviceFile	Loads a device file, such as PCI DSP lod file
Command	ArcDevice_Command ArcDevice_Command_I ArcDevice_Command_II ArcDevice_Command_III ArcDevice_Command_IIII	Sends an ASCII command with arguments to the device or controller
GetControllerId	ArcDevice_GetControllerId	Returns the controller id or 'ERR' (0x455252) if not supported
ResetController	ArcDevice_ResetController	Resets the controller
IsControllerConnected	ArcDevice_IsControllerConnected	Returns true of a controller is connected and turned-on
SetupController	ArcDevice_SetupController	Performs the tasks required to initialize a controller
LoadControllerFile	ArcDevice_LoadControllerFile	Loads a controller file, such as a TIMING DSP lod file
SetImageSize	ArcDevice_SetImageSize	Sets the image dimensions to be used on the controller
GetImageRows	ArcDevice_GetImageRows	Returns the row image dimension as set on the controller
GetImageCols	ArcDevice_GetImageCols	Returns the column image dimension as set on the controller
GetCCParams	ArcDevice_GetCCParams	Returns the controller configuration parameter value
IsCCParamSupported	ArcDevice_IsCCParamSupported	Returns true of the specified configuration parameter is supported on the controller
IsCCD	ArcDevice_IsCCD	Returns true of the attached array is a CCD; false if IR
IsBinningSet	ArcDevice_IsBinningSet	Returns true if binning is set on the controller
SetBinning	ArcDevice_SetBinning	Sets binning on the controller (must be supported by DSP lod file)
UnSetBinning	ArcDevice_UnSetBinning	Un-sets binning on the controller
IsSyntheticImageMode	ArcDevice_IsSyntheticImageMode	Returns true if synthetic image mode is set on the controller

SetSyncheticImageMode	ArcDevice_SetSyncheticImageMode	Toggles synthetic image mode on the controller
SetOpenShutter	ArcDevice_SetOpenShutter	Toggles the shutter to open or stay closed during an exposure
Expose	ArcDevice_Expose	Performs basic exposure and readout handling (best if called from a thread)
StopExposure	ArcDevice_StopExposure	Aborts the current exposure/readout
Continuous	ArcDevice_Continuous	Performs basic continuous readout (best if called from a thread)
StopContinuous	ArcDevice_StopContinuous	Aborts the current continuous exposure/readout
IsReadout	ArcDevice_IsReadout	Returns true if the controller is currently reading out an image
GetPixelCount	ArcDevice_GetPixelCount	Returns current pixel count during readout
GetCRPixelCount	ArcDevice_GetCRPixelCount	Returns current pixel count during continuous readout
GetFrameCount	ArcDevice_GetFrameCount	Returns current frame count during readout
ContainsError	ArcDevice_ContainsError ArcDevice_ContainsError_I	Returns true if the specified command reply contains an error reply ('ERR', 'SYR', 'TOUT', 'CNR', 'RST', 'ROUT' or 'HERR') or does not fall within the specified range.
GetNextLoggedCmd	ArcDevice_GetNextLoggedCmd	Returns the next logged command (for debug)
GetLoggedCmdCount	ArcDevice_GetLoggedCmdCount	Returns the logged command count (for debug)
SetLogCmds	ArcDevice_SetLogCmds	Toggle command logging on/off
GetArrayTemperature	ArcDevice_GetArrayTemperature	Returns the current average array temperature (in celcius)
GetArrayTemperatureDN	ArcDevice_GetArrayTemperatureDN	Returns the current average array temperature digital number
SetArrayTemperature	ArcDevice_SetArrayTemperature	Sets the array target temperature
LoadTemperatureCtrlData	ArcDevice_LoadTemperatureCtrlData	Loads a temperature control data file
SaveTemperatureCtrlData	ArcDevice_SaveTemperatureCtrlData	Saves a temperature control data file

CArcDevice::ToString

Syntax:

```
const std::string ToString();
```

Namespace:

arc::device

Description:

Returns a descriptive string that represents the device controlled by this library.

Parameters:

N/A

Throws Exception:

N/A

Return Value	Description
const std::string	Device dependent string

Notes:

The string returned by this method is device dependent and may change at any time.

Current PCIe String: "PCIe [ARC-66 / 67]"

Current PCI String: " PCI [ARC-63 / 64]"

Usage:

```
#include <iostream>
#include “CArcDevices.h”
#include “CArcPCIe.h”

using namespace std;
using namespace arc::device;

CArcPCIe::FindDevices();

CArcDevice *pArcDev = new CArcPCIe();

pArcDev->Open( 0 );

cout << “Device in use: “ << pArcDev->ToString() << endl;

pArcDev->Close();
```

CArcDevice::IsOpen

Syntax:

```
bool IsOpen();
```

Namespace:

arc::device

Description:

Returns true if an application has called *CArcDevice::Open* successfully.

Parameters:

N/A

Throws Exception:

N/A

Return Value	Description
true	The device is already open
false	The device is not open

Usage:

```
#include <iostream>
#include “CArcDevices.h”
#include “CArcPCIe.h”

using namespace std;
using namespace arc::device;

CArcPCIe::FindDevices();

CArcDevice *pArcDev = new CArcPCIe();

pArcDev->Open( 0, BUFFER_SIZE );

if ( !pArcDev->IsOpen() )
{
      cerr << “Device failed to open!” << endl;
}

. . . .
```

CArcDevice::Open

Syntax:

```
void Open( int dDeviceNumber );
void Open( int dDeviceNumber, int dBufferSize );
void Open( int dDeviceNumber, int dRows, int dCols );
```

Namespace:

arc::device

Description:

Opens a connection to the specified host interface device.

Parameters:

dDeviceNumber

Device number in the range 0 to N (N-th host interface board)

dBufferSize

The size (in bytes) of the common image buffer to allocate

Throws Exception:

std::runtime_error

Return Value	Description
N/A	N/A

Notes:

The number of host interface boards can be found using the FindDevices and DeviceCount class methods.

Usage:

```
#define BUFFER_SIZE     2200 * 2200 * sizeof( unsigned short )

CArcDevice *pArcDev = new CArcPCIe();

//
// Open device 0 with a 2200 x 2200 pixel common image buffer
//
pArcDev->Open( 0, BUFFER_SIZE );
```

To open a device without allocating an image buffer or if you intend to call *CArcDevice::MapCommonBuffer* separately:

```
CArcDevice *pArcDev = new CArcPCIe();

//
// Open device 0 with NO common image buffer
//
pArcDev->Open( 0 );
```

CArcDevice::Close

Syntax:

```
void Close();
```

Namespace:

arc::device

Description:

Closes a host interface device connection.

Parameters:

N/A

Throws Exception:

N/A

Return Value	Description
N/A	N/A

Usage:

```
#include <iostream>
#include “CArcDevices.h”
#include “CArcPCIe.h”

using namespace std;
using namespace arc::device;

CArcPCIe::FindDevices();

CArcDevice *pArcDev = new CArcPCIe();

pArcDev->Open( 0 );

if ( !pArcDev->IsOpen() )
{
      cerr << “Device failed to open!” << endl;
}

pArcDev->Close();
```

CArcDevice::Reset

Syntax:

```
void Reset();
```

Namespace:

arc::device

Description:

Resets the host interface device.

Parameters:

N/A

Throws Exception:

std::runtime_error

Return Value	Description
N/A	N/A

Notes:

May not be implemented for all host interface devices.

Usage:

```
      //
      // Reset the PCI board
      //
      CArcDevice* pArcDev = new CArcPCI();

      . . . .

      pArcDev->Reset();

      . . . .


      //
      // Reset the PCIe board
      //
      CArcDevice* pArcDev = new CArcPCIe();

      . . . .

      pArcDev->Reset();

      . . . .
```

CArcDevice::MapCommonBuffer

Syntax:

```
void MapCommonBuffer( int dBufferSize = 0 );
```

Namespace:

arc::device

Description:

Maps a common buffer of the specified size (in bytes) into user virtual space.

Parameters:

dBufferSize

The size (in bytes) of the common image buffer to allocate

Throws Exception:

std::runtime_error

Return Value	Description
N/A	N/A

Notes:

Mapping of the common buffer into user virtual space may fail due to insufficient contiguous memory. How and when the common buffer is actually allocated is operating system dependent. The size of the buffer should be verified by calling *CArcDevice::CommonBufferSize*.

The buffer should be unmapped by calling *CArcDevice::UnMapCommonBuffer or CArcDevice::Close*. The virtual address will cease to be valid after closing the device or after unmapping the buffer. Refer to *CArcDevice::UnMapCommonBuffer*.

The virtual address for the common buffer can be had by calling *CArcDevice::CommonBufferVA*. The returned pointer can be used to directly access the buffer. This pointer should not be freed by the user; the CArcDevice class will handle this.

Usage:

```
#define BUFFER_SIZE     2200 * 2200 * sizeof( unsigned short )

CArcDevice *pArcDev = new CArcPCIe();

//
// Open device 0
//
pArcDev->Open( 0 );

//
// Map a 2200 x 2200 pixel common image buffer
//
pArcDev->MapCommonBuffer( BUFFER_SIZE );

if ( pArcDev->CommonBufferSize() != BUFFER_SIZE )
{
<continued next page>
```

```
      cerr << “Failed to map image buffer!” << endl;

      return 1;
}
```

<continued next page>

```
//
// Get the virtual address to 16-bit data
//
unsigned short* pU16Buf =
                  ( unsigned short * )pArcDev->CommonBufferVA();

//
// Print the first ten values
//
for ( int i=0; i<10; i++ )
{
      cout <<  “Buffer[ “ << i << “ ]: “ << pU16Buf[ i ] << endl;
}

//
// UnMap buffer or just call Close
//
pArcDev->UnMapCommonBuffer();

pArcDev->Close();
```

CArcDevice::UnMapCommonBuffer

Syntax:

```
void UnMapCommonBuffer();
```

Namespace:

arc::device

Description:

Unmaps the common buffer from user virtual space.

Parameters:

N/A

Throws Exception:

N/A

Return Value	Description
N/A	N/A

Notes:

The buffer should be unmapped by calling *CArcDevice::UnMapCommonBuffer or CArcDevice::Close*. The virtual address will cease to be valid after closing the device or after unmapping the buffer.

Usage:

```
CArcDevice *pArcDev = new CArcPCIe();

pArcDev->Open( 0 );

pArcDev->MapCommonBuffer( 1024 * 1200 * 2 );

//
// Do some stuff here
//
. . . .

pArcDev->UnMapCommonBuffer();
```

CArcDevice::ReMapCommonBuffer

Syntax:

```
void ReMapCommonBuffer( int dBufferSize = 0 );
```

Namespace:

arc::device

Description:

Re-Maps the common buffer to have the specified size (in bytes) .

Parameters:

dBufferSize

The size (in bytes) of the common image buffer

Throws Exception:

std::runtime_error

Return Value	Description
N/A	N/A

Notes:

Re-Mapping of the common buffer into user virtual space may fail due to insufficient contiguous memory. The size of the buffer should be verified by calling *CArcDevice::CommonBufferSize*.

Any previous virtual addresses retrieved by calling *CArcDevice::CommonBufferVA* should no longer be used and a new address should be had by re-calling *CArcDevice::CommonBufferVA*.

Usage:

```
#define BUFFER1_SIZE    2200 * 2200 * sizeof( unsigned short )
#define BUFFER2_SIZE    1200 * 1200 * sizeof( unsigned short )

CArcDevice *pArcDev = new CArcPCIe();

//
// Open device 0
//
pArcDev->Open( 0 );

//
// Map a 2200 x 2200 pixel common image buffer
//
pArcDev->MapCommonBuffer( BUFFER1_SIZE );

if ( pArcDev->CommonBufferSize() != BUFFER1_SIZE )
{
      cerr << “Failed to map image buffer!” << endl;
      return 1;
}

<continued next page>
```

```
//
// Get the virtual address to 16-bit data
//
unsigned short* pU16Buf =
                  ( unsigned short * )pArcDev->CommonBufferVA();

//
// Print the first ten values
//
for ( int i=0; i<10; i++ )
{
      cout <<  “Buffer[ “ << i << “ ]: “ << pU16Buf[ i ] << endl;
}

//
// ReMap the buffer to a smaller one
//
pArcDev->ReMapCommonBuffer( BUFFER2_SIZE );

if ( pArcDev->CommonBufferSize() != BUFFER2_SIZE )
{
      cerr << “Failed to re-map image buffer!” << endl;
      return 1;
}

//
// Get the NEW virtual address to 16-bit data
//
pU16Buf = ( unsigned short * )pArcDev->CommonBufferVA();

//
// Print the first ten values
//
for ( int i=0; i<10; i++ )
{
      cout <<  “Buffer[ “ << i << “ ]: “ << pU16Buf[ i ] << endl;
}

//
// UnMap buffer or just call Close
//
pArcDev->UnMapCommonBuffer();

pArcDev->Close();
```

CArcDevice::GetCommonBufferProperties

Syntax:

```
bool GetCommonBufferProperties();
```

Namespace:

arc::device

Description:

Calls the host interface driver to retrieve the common buffer properties: *user virtual address*, *physical address*, and *size* (in bytes).

Parameters:

N/A

Throws Exception:

N/A

Return Value	Description
true	The function was successful
false	The function failed

Notes:

The properties are maintained by the CArcDevice class and can be retrieved by calling the following methods: *CArcDevice::CommonBufferVA*, *CArcDevice::CommonBufferPA*, and *CArcDevice::CommonBufferSize*.

For PCI and PCIe host interfaces this function is automatically called within *the CArcDevice::MapCommonBuffer*.

Usage:

```
CArcDevice *pArcDev = new CArcPCIe();

//
// Open device, etc.
//
. . . .

if ( pArcDev->GetCommonBufferProperties() )
{
      cout << "Image buf virt addr: " << pArcDev->CommonBufferVA() << endl;
      cout << "Image buf phys addr: " << pArcDev->CommonBufferPA() << endl;
      cout << "Image buf size: " << pArcDev->CommonBufferSize() << endl;
}
else
{
      cerr << "Failed to read buffer properties!" << endl;
}
```

CArcDevice::FillCommonBuffer

Syntax:

```
void FillCommonBuffer( unsigned short u16Value = 0 );
```

Namespace:

arc::device

Description:

Fills the common buffer with the specified 16-bit value.

Parameters:

u16Value

The value to fill the common image buffer with; default = 0

Throws Exception:

std::runtime_error

Return Value	Description
N/A	N/A

Usage:

```
CArcDevice *pArcDev = new CArcPCIe();

//
// Open device, etc
//
. . . .

//
// Fill the buffer with 0xBEEF
//
pArcDev->FillCommonBuffer( 0xBEEF );
```

CArcDevice::CommonBufferVA

Syntax:

```
void* CommonBufferVA();
```

Namespace:

arc::device

Description:

Returns the common buffer user virtual address.

Parameters:

N/A

Throws Exception:

N/A

Return Value	Description
void *	The buffer base virtual address
NULL	No buffer exists or GetCommonBufferProperties has not been called

Notes:

The user virtual address can only be valid after calling *CArcDevice::GetCommonBufferProperties*.

Usage:

```
CArcDevice *pArcDev = new CArcPCIe();

//
// Open device, etc.
//
. . . .

//
// Get the virtual address to 16-bit data
//
unsigned short* pU16Buf =
                  ( unsigned short * )pArcDev->CommonBufferVA();

//
// Print the first ten values
//
for ( int i=0; i<10; i++ )
{
      cout <<  “Buffer[ “ << i << “ ]: “ << pU16Buf[ i ] << endl;
}
```

CArcDevice::CommonBufferPA

Syntax:

```
unsigned long CommonBufferPA();
```

Namespace:

arc::device

Description:

Returns the common buffer physical address.

Parameters:

N/A

Throws Exception:

N/A

Return Value	Description
unsigned long	The buffer base physical address
0	No buffer exists or GetCommonBufferProperties has not been called

Notes:

The physical address is an invalid address for the user application. It is only available for reference and validation and should never be called upon. The returned address is only valid after calling *CArcDevice::GetCommonBufferProperties*.

Usage:

```
CArcDevice *pArcDev = new CArcPCIe();

//
// Open device, etc.
//
. . . .

if ( pArcDev->GetCommonBufferProperties() )
{
      cout << “Image buf virt addr: “ << pArcDev->CommonBufferVA() << endl;
      cout << “Image buf phys addr: “ << pArcDev->CommonBufferPA() << endl;
      cout << “Image buf size: “ << pArcDev->CommonBufferSize() << endl;
}
else
{
      cerr << “Failed to read buffer properties!” << endl;
}
```

CArcDevice::CommonBufferSize

Syntax:

```
int CommonBufferSize();
```

Namespace:

arc::device

Description:

Returns the common buffer size (in bytes).

Parameters:

N/A

Throws Exception:

N/A

Return Value	Description
int	The buffer size (in bytes)
0	No buffer exists or GetCommonBufferProperties has not been called

Notes:

The size (in bytes) of the allocated common image buffer. The returned size is only valid after calling *CArcDevice::GetCommonBufferProperties*.

Usage:

```
CArcDevice *pArcDev = new CArcPCIe();

//
// Open device, etc.
//
. . . .

if ( pArcDev->GetCommonBufferProperties() )
{
      cout << "Image buf virt addr: " << pArcDev->CommonBufferVA() << endl;
      cout << "Image buf phys addr: " << pArcDev->CommonBufferPA() << endl;
      cout << "Image buf size: " << pArcDev->CommonBufferSize() << endl;
}
else
{
      cerr << "Failed to read buffer properties!" << endl;
}
```

CArcDevice::GetId

Syntax:

```
int GetId();
```

Namespace:

arc::device

Description:

Returns the hardware device ID.

Parameters:

N/A

Throws Exception:

std::runtime_error (PCIe only)

Return Value	Description
int	The hardware device ID
0	No hardware device ID exists

Notes:

The *CArcPCIe* class contains a static constant against which the return value can be compared.

Usage:

```
CArcDevice *pArcDev = new CArcPCIe();

if ( pArcDev->GetId() == CArcPCIe::ID )
{
      cout << “Found PCIe board!” << endl;
}
```

CArcDevice::GetStatus

Syntax:

```
int GetStatus();
```

Namespace:

arc::device

Description:

Returns the hardware device status.

Parameters:

N/A

Throws Exception:

std::runtime_error

Return Value	Description
int	The hardware device status

Notes:

The returned value is device specific.

Usage:

```
    CArcDevice *pArcDev = new CArcPCIe();

    . . . .

    cout << “Device status: ” << pArcDev->GetStatus() << endl;
```

CArcDevice::ClearStatus

Syntax:

```
void ClearStatus();
```

Namespace:

arc::device

Description:

PCIe only - Clears the device status.

Parameters:

N/A

Throws Exception:

std::runtime_error

Return Value	Description
N/A	N/A

Notes:

This method is valid for PCIe only; does nothing on other host interface devices.

Not generally useful in user applications.

Usage:

```
      CArcDevice *pArcDev = new CArcPCIe();

      . . . .

      pArcDev->ClearStatus();
```

CArcDevice::Set2xFOTransmitter

Syntax:

```
void Set2xFOTransmitter( bool bOnOff );
```

Namespace:

arc::device

Description:

Enables/disables dual fiber optic transmitters on the camera controller.

Parameters:

bOnOff

> *true* to enable dual transmitters; *false* to disable

Throws Exception:

std::runtime_error

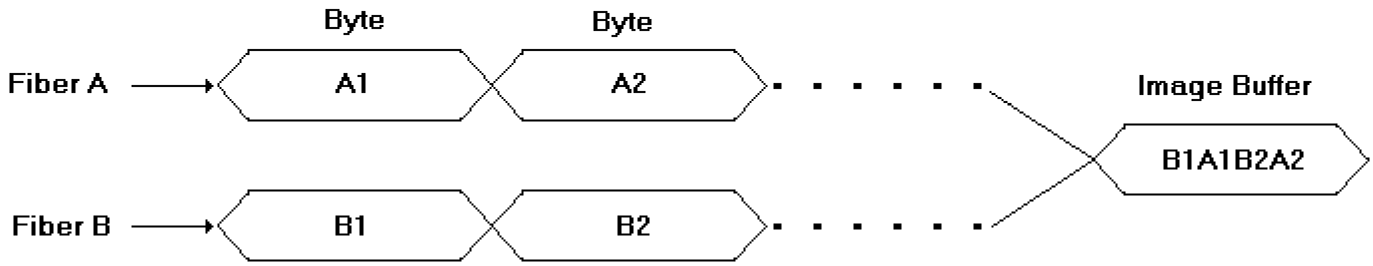
Return Value	Description
N/A	N/A.

Notes:

For **PCIe devices** dual receivers must be installed on the board.

For **PCIe devices** this method enables/disables dual transmitters on the controller and enables/disables dual receivers on the PCIe board.

PCIe Dual Receiver Data Format



For **PCI devices** this method enables/disables dual transmitters on the controller only.

Usage:

```
CArcDevice *pArcDev = new CArcPCIe();

. . . .

//
// Enable dual FO transmitters on the controller.
//
pArcDev->Set2xFOTransmitter( true );
```

CArcDevice::LoadDeviceFile

Syntax:

```
void LoadDeviceFile( const std::string sFilename );
```

Namespace:

arc::device

Description:

PCI Only – Loads a PCI ‘.lod' file into the boards DSP for execution.

Parameters:

pszFile

The PCI ‘.lod’ file to load; includes path (relative or full)

Throws Exception:

std::runtime_error

Return Value	Description
N/A	N/A.

Notes:

Execution of the DSP file begins immediately following upload completion. This method does nothing on non-PCI boards.

Usage:

```
CArcDevice *pArcDev = new CArcPCI();

. . . .

//
// Load a PCI file
//
pArcDev->LoadDeviceFile( “C:\\User\\DSPFiles\\pci.lod” );
```

CArcDevice::Command

Syntax:

```
int Command( int dBoardId, int dCommand, int dArg1, int dArg2, int dArg3, int dArg4 );
```

Namespace:

arc::device

Description:

Sends an ASCII command to the specified board.

Parameters:

dBoardId

The board ID; PCI_ID, TIM_ID or UTIL_ID

dCommand

A valid ASCII controller command. See *ArcDefs.h* for command and reply definitions.

dArg1 – dArg4

Arguments for the command; default = -1

Throws Exception:

std::runtime_error

Return Value	Description
int	The command reply. This is command dependent, but is typically the ASCII word 0x444F4E ('DON').
0x455252 ('ERR')	The command is invalid or failed
0x544F5554 ('TOUT')	Timeout occurred while processing the command

Usage:

```
CArcDevice *pArcDev = new CArcPCIe();

. . . .

// Send a series of Test Data Links ( 'TDL' )
// to timing board.
for ( int i=0; i<123; i++ )
{
      int dRetVal = pArcDev->Command( TIM_ID, TDL, i );

      if ( dRetVal != i )
      {
            throw runtime_error( "TDL failed!" );
      }
}

// Send Power On to the controller
if ( pArcDev->Command( TIM_ID, PON ) != DON )
{
      throw runtime_error( "PON failed!" );
}
```

CArcDevice::GetControllerId

Syntax:

```
int GetControllerId();
```

Namespace:

arc::device

Description:

Returns the hardware ID from the timing board.

Parameters:

N/A

Throws Exception:

std::runtime_error

Return Value	Description
int	The controller ID
0x455252 (‘ERR’)	The timing board doesn’t support a controller ID in hardware

Notes:

Currently (2011) only the SmallCam timing board contains a hardware ID. The SmallCam ID is 0x534330 (‘SC#’, where # is currently 0).

ArcDefs.h defines a macro called *IS_ARC12(id)* that can be called to verify that the ID matches that of SmallCam. The macro returns bool *true* or *false*.

Usage:

```
CArcDevice *pArcDev = new CArcPCIe();

. . . .

int dId = pArcDev->GetControllerId();

if ( IS_ARC12( dId ) )
{
      cout << “Found SmallCam!” << endl;
}
else
{
      cout << “Controller ID: ” << hex << dId << dec << endl;
}
```

CArcDevice::ResetController

Syntax:

```
void ResetController( bool bDSPOnly );
```

Namespace:

arc::device

Description:

Resets the controller.

Parameters:

bDSPOnly

SmallCam only. True to only reset the SmallCam DSP and not the entire controller. Default = false

Throws Exception:

std::runtime_error

Return Value	Description
N/A	N/A

Notes:

Usage:

```
CArcDevice *pArcDev = new CArcPCIe();

. . . .

pArcDev->ResetController();
```

CArcDevice::SetupController

Syntax:

```
void SetupController( bool bReset, bool bTdl, bool bPower, int dRows, int dCols, const
std::string sTimFile, const std::string sUtilFile, const std::string sPciFile, const bool&
bAbort );
```

Namespace:

arc::device

Description:

Convenience function to initialize a camera controller.

Parameters:

bReset

True to reset the controller. Typically be set to true.

bTdl

True to test the data link between the host computer and the host device (PCI, PCIe), and the host device and the camera controller. Typically set to true.

bPower

True to power-on the camera controller. Typically set to true.

dRows

Image row dimension (in pixels)

dCols

Image column dimension (in pixels)

pszTimFile

DSP timing board file (.lod)

pszUtilFile

DSP utility board file (.lod). Default = NULL

pszPciFile

DSP PCI board file (.lod). Default = NULL

bAbort

Reference variable to allow external program to exit this method. Default = false

Throws Exception:

std::runtime_error

Return Value	Description
N/A	N/A

Notes:

This method must be called before any exposures or commands other than test data link (‘TDL’) and read/write memory (‘RDM’/’WRM’) can occur.

Usage:

```
CArcPCIe::FindDevices();

if ( CArcPCIe::DeviceCount() > 0 )
{
      CArcDevice *pArcDev = new CArcPCIe();

      int dRows = 1024;
      int dCols = 1200;

      pArcDev->Open( 0, dRows * dCols * 2 );

      pArcDev->SetupController( true,           // reset controller
                                true,           // test data links
                                true,           // power on
                                1024,           // row size
                                1200,           // col size
                                “tim.lod” );    // DSP timing file


      pArcDev->Expose( 1.5f, 1024, 1200 );

      pArcDev->Close();
}
```

CArcDevice::LoadControllerFile

Syntax:

```
void  LoadControllerFile(  const  std::string  sFilename,  bool  bValidate,  const  bool&
bAbort );
```

Namespace:

arc::device

Description:

Loads a DSP timing or utility file onto the camera controller.

Parameters:

pszFilename

The DSP timing or utility file to load onto the controller. Typically tim.lod (timing board) or util.lod (utility board).

bValidate

True to verify that each data word is written successfully. Default = true

bAbort

Reference variable to allow external program to exit this method. Default = false

Throws Exception:

std::runtime_error

Return Value	Description
N/A	N/A

Notes:

Calling this method will effectively wipe out any existing controller settings. This method is called from within the SetupController() method.

Usage:

```
CArcDevice *pArcDev = new CArcPCIe();

. . . .

// The following is essentially what SetupController() does:
pArcDev->ResetController();

for ( int i=0; i<123; i++ ) {
      pArcDev->TestDataLink();

      if ( pArcDev->Command( TIM_ID, TDL, 0x123456 ) != 0x123456 )
            throw runtime_error( “TIM TDL failed!” );
}

pArcDev->LoadControllerFile( “tim.lod” );
pArcDev->Command( TIM_ID, PON );
```

```
      pArcDev->SetImageSize( 1024, 1200 );
```

CArcDevice::SetImageSize

Syntax:

```
void SetImageSize( int dRows, int dCols );
```

Namespace:

arc::device

Description:

Set the image dimensions on the camera controller.

Parameters:

dRows

The row image dimension (in pixels).

dCols

The column image dimension (in pixels).

Throws Exception:

std::runtime_error

Return Value	Description
N/A	N/A

Notes:

This method is called from within the SetupController() method.

Usage:

```
      CArcDevice *pArcDev = new CArcPCIe();

      . . . .

      //  Set the image size to 1200x1024
      // +--------------------------------------+
      pArcDev->SetImageSize( 1024, 1200 );

      . . . .
```

CArcDevice::GetImageRows

Syntax:

```
int GetImageRows();
```

Namespace:

arc::device

Description:

Get the image row dimension from the camera controller.

Parameters:

N/A

Throws Exception:

std::runtime_error

Return Value	Description
int	The image row dimension (in pixels)

Notes:

N/A

Usage:

```
      CArcDevice *pArcDev = new CArcPCIe();

      . . . .

      int dRows = pArcDev->GetImageRow();
      int dCols = pArcDev->GetImageCols();

      cout << "Current Image Size: " << dRows << "x" << dCols << endl;

      . . . .
```

CArcDevice::GetImageCols

Syntax:

```
int GetImageCols();
```

Namespace:

arc::device

Description:

Get the image column dimension from the camera controller.

Parameters:

N/A

Throws Exception:

std::runtime_error

Return Value	Description
int	The image column dimension (in pixels)

Notes:

N/A

Usage:

```
      CArcDevice *pArcDev = new CArcPCIe();

      . . . .

      int dRows = pArcDev->GetImageRow();
      int dCols = pArcDev->GetImageCols();

      cout << "Current Image Size: " << dRows << "x" << dCols << endl;

      . . . .
```

CArcDevice::GetCCParams

Syntax:

```
int GetCCParams();
```

Namespace:

arc::device

Description:

Get the controller configuration parameter value from the camera controller.

Parameters:

N/A

Throws Exception:

std::runtime_error

Return Value	Description
int	The current controller configuration parameter value.

Notes:

The controller configuration parameter value bits specify the DSP firmware capabilities. The capabilities include binning, sub-array, temperature readout, shutter existence, which ARC boards are in the system, etc. The current bit definitions can be found in *ArcDefs.h*.

Call method *CArcDevice::IsCCParamSupported(int)* to determine if individual capabilities are available.

Usage:

```
#include “ArcDefs.h”

CArcDevice *pArcDev = new CArcPCIe();

. . . .

int dCCParam = pArcDev->GetCCParam();

if ( pArcDev->IsCCParamSupported( ARC22 ) ) {
      cout << “ARC-22 board in system!” << endl;
}
else if ( pArcDev->IsCCParamSupported( SHUTTER_CC ) ) {
      cout << “Shutter support!” << endl;
}
else if ( pArcDev->IsCCParamSupported( SPLIT_SERIAL ) ) {
      cout << “Serial readout supported!” << endl;
}
else if ( pArcDev->IsCCParamSupported( BINNING ) ) {
      cout << “Binning supported!” << endl;
}
else if ( pArcDev->IsCCParamSupported( SUBARRAY ) ) {
      cout << “Sub-Array supported!” << endl;
}
. . . .
```

CArcDevice::IsCCParamSupported

Syntax:

```
bool IsCCParamSupported( int dParameter );
```

Namespace:

```
arc::device
```

Description:

Determines if the specified controller configuration parameter is available on the camera controller.

Parameters:

dParameter

The controller configuration parameter to check. A list of parameters can be found in *ArcDefs.h*.

Throws Exception:

N/A

Return Value	Description
true	The specified parameter is supported
false	The specified parameter is NOT supported

Notes:

N/A

Usage:

```
#include “ArcDefs.h”

CArcDevice *pArcDev = new CArcPCIe();

. . . .

int dCCParam = pArcDev->GetCCParam();

if ( pArcDev->IsCCParamSupported( ARC22 ) ) {
      cout << “ARC-22 board in system!” << endl;
}
else if ( pArcDev->IsCCParamSupported( SHUTTER_CC ) ) {
      cout << “Shutter support!” << endl;
}
else if ( pArcDev->IsCCParamSupported( SPLIT_SERIAL ) ) {
      cout << “Serial readout supported!” << endl;
}
else if ( pArcDev->IsCCParamSupported( BINNING ) ) {
      cout << “Binning supported!” << endl;
}
else if ( pArcDev->IsCCParamSupported( SUBARRAY ) ) {
      cout << “Sub-Array supported!” << endl;
}
. . . .
```

CArcDevice::IsControllerConnected

Syntax:

```
bool IsControllerConnected();
```

Namespace:

```
arc::device
```

Description:

Determines if a camera controller is connected and powered-on.

Parameters:

N/A

Throws Exception:

std::runtime_error

Return Value	Description
true	A camera controller is connected and powered-on
false	No camera controller is connected or is not powered-on

Notes:

N/A

Usage:

```
#include “ArcDefs.h”

CArcPCIe::FindDevices();

CArcDevice *pArcDev = new CArcPCIe();

pArcDev->Open( 0, dBufferSize );

if ( pArcDev->IsControllerConnected() )
{
      cout << “Yeah! A controller is connected!” << endl;
}

else
{
      cout << “Hmmm, maybe we forgot to turn it on!” << endl;
}

. . . .
```

CArcDevice::IsCCD

Syntax:

```
bool IsCCD();
```

Namespace:

```
arc::device
```

Description:

Determines if the camera controller is for a CCD or IR system.

Parameters:

N/A

Throws Exception:

N/A

Return Value	Description
true	The camera controller is for a CCD system
false	The camera controller is for a IR system

Notes:

This method searches the current controller configuration parameter for the existence of IR boards. The method returns true if no IR boards are found.

Usage:

```
        CArcPCIe::FindDevices();

        CArcDevice *pArcDev = new CArcPCIe();

        pArcDev->Open( 0, dBufferSize );

        pArcDev->SetupController( true, true, true, 1024, 1200, “tim.lod” );

        if ( pArcDev->IsCCD() )
        {
              cout << “This is a CCD system!” << endl;
        }

        else
        {
              cout << “This is an IR system!” << endl;
        }

        . . . .
```

CArcDevice::IsBinningSet

Syntax:

```
bool IsBinningSet();
```

Namespace:

```
arc::device
```

Description:

Determines if the camera controller is currently set for binning.

Parameters:

N/A

Throws Exception:

std::runtime_error

Return Value	Description
true	The camera controller is set for binning
false	The camera controller is NOT set for binning

Notes:

N/A

Usage:

```
CArcDevice *pArcDev = new CArcPCIe();

. . . .

//  Set binning mode to 2x2
// +---------------------------------------------+
pArcDev->SetBinning( dRows, dCols, 2, 2 );

if ( pArcDev->IsBinningSet() )
{
      cout << "Binning is SET!" << endl;
}

. . . .

//  Un-Set binning mode
// +---------------------------------------------+
pArcDev->UnSetBinning( dRows, dCols );

if ( pArcDev->IsBinningSet() )
{
      cout << "Binning is NO LONGER SET!" << endl;
}

. . . .
```

CArcDevice::SetBinning

Syntax:

```
void SetBinning( int dRows, int dCols, int dRowFactor, int dColFactor, int* dBinRows, int*
dBinCols );
```

Namespace:

```
arc::device
```

Description:

Sets the camera controller to binning mode.

Parameters:

dRows

The number of rows in the un-binned image.

dCols

The number of columns in the un-binned image.

dRowFactor

The row binning factor.

dColFactor

The column binning factor.

dBinRows

Optional pointer to return the binned image row size to the caller. Default = NULL

dBinCols

Optional pointer to return the binned image column size to the caller. Default = NULL

Throws Exception:

std::runtime_error

Return Value	Description
N/A	N/A

Notes:

Binning is used to combine pixels together on the chip and results in a smaller image. The number of pixels that are combined is determined by the row and column parameters, which do not need to match. A binning factor of 1 means no binning occurs along that image axis.

Usage:

```
CArcDevice* pArcDev = new CArcPCIe();

//   Set the binning to 4x2
pArcDev->SetBinning( dRows, dCols, 2, 4 );

. . . .
```

CArcDevice::UnSetBinning

Syntax:

```
void UnSetBinning( int dRows, int dCols );
```

Namespace:

```
arc::device
```

Description:

Sets the camera controller from binning mode back to normal image readout.

Parameters:

dRows

The number of rows in the un-binned image.

dCols

The number of columns in the un-binned image.

Throws Exception:

std::runtime_error

Return Value	Description
N/A	N/A

Notes:

N/A

Usage:

```
CArcDevice *pArcDev = new CArcPCIe();

. . . .

//  Set binning mode to 2x2
// +--------------------------------------------+
pArcDev->SetBinning( dRows, dCols, 2, 2 );

if ( pArcDev->IsBinningSet() )
{
      cout << "Binning is SET!" << endl;
}

. . . .

//  Un-Set binning mode
// +--------------------------------------------+
pArcDev->UnSetBinning( dRows, dCols );

if ( pArcDev->IsBinningSet() )
{
      cout << "Binning is NO LONGER SET!" << endl;
}
. . . .
```

CArcDevice::SetSubArray

Syntax:

```
void SetSubArray( int& dOldRows, int& dOldCols, int dRow, int dCol, int dSubRows, int
dSubCols, int dBiasOffset, int dBiasWidth );
```

Namespace:

```
arc::device
```

Description:

Sets the camera controller into sub-array mode.

Parameters:

dOldRows

> The current number of image rows set on the camera controller (in pixels).

dOldCols

> The current number of image columns set on the camera controller (in pixels).

dRow

> The row number of the sub-array center (in pixels).

dCol

> The column number of the sub-array center (in pixels).

dSubRows

> The number of rows in the sub-image (in pixels).

dSubCols

> The number of columns in the sub-image (in pixels).

dBiasOffset

> The pixel offset to the start of the bias region.

dBiasWidth

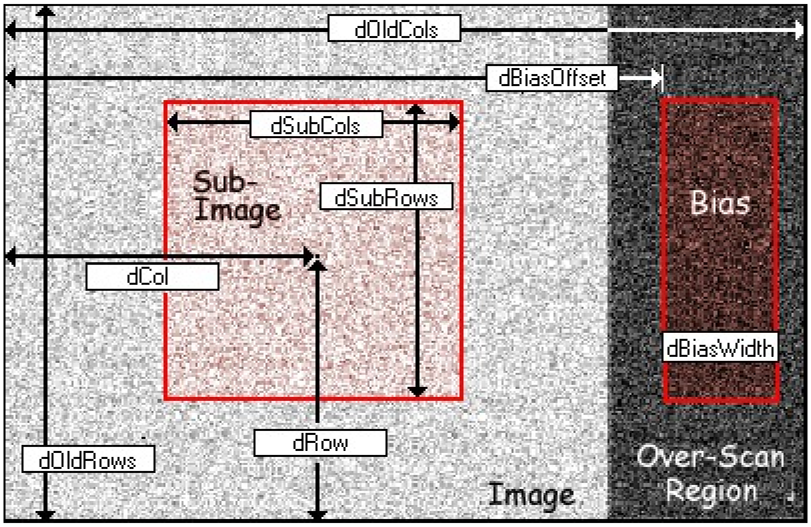
> The width of the bias region (in pixels).

Throws Exception:

std::runtime_error

Return Value	Description
N/A	N/A

Notes:



Usage:

```
CArcDevice *pArcDev = new CArcPCIe();

. . . .

int dOldRows = 0, dOldCols = 0;

//  Create a 600x500 pixel sub-array centered at row 150, col 200
//  with a 100x500 pixel bias region located at an offset of 1100 pixels
// +------------------------------------------------------------------+
pArcDev->SetSubArray( dOldRows,
                      dOldCols,
                      150,
                      200,
                      500,
                      600,
                      1100,
                      100 );

. . . .
```

CArcDevice::UnSetSubArray

Syntax:

```
void UnSetSubArray( int dRows, int dCols );
```

Namespace:

```
arc::device
```

Description:

Removes the camera controller from sub-array mode.

Parameters:

dRows

The number of rows (in full image) to set on the camera controller (in pixels).

dCols

The number of columns (in full image) to set on the camera controller (in pixels).

Throws Exception:

std::runtime_error

Return Value	Description
N/A	N/A

Notes:

N/A

Usage:

```
CArcDevice *pArcDev = new CArcPCIe();

. . . .

int dOldRows = 0, dOldCols = 0;

//  Create a 600x500 pixel sub-array centered at row 150, col 200
//  with a 100x500 pixel bias region located at an offset of 1100 pixels
// +------------------------------------------------------------------+
pArcDev->SetSubArray( dOldRows, dOldCols, 150, 200, 500, 600, 1100, 100 );

. . . .

//  Expose in sub-array mode
// +----------------------------------------------+
pArcDev->Expose( 1.5f, 500, 600 );

. . . .

//  Un-Set sub-array mode
// +----------------------------------------------+
pArcDev->UnSetSubArray( dOldRows, dOldCols );

. . . .
```

CArcDevice::IsSyntheticImageMode

Syntax:

```
bool IsSyntheticImageMode();
```

Namespace:

```
arc::device
```

Description:

Determines if the camera controller is currently set for synthetic image mode.

Parameters:

N/A

Throws Exception:

std::runtime_error

Return Value	Description
true	The camera controller is set for synthetic image mode
false	The camera controller is NOT set for synthetic image mode

Notes:

See *CArcDevice::SetSyntheticImageMode()* notes for more details.

Usage:

```
CArcDevice *pArcDev = new CArcPCIe();

. . . .

if ( pArcDev->IsSyntheticImageMode() )
{
      cout << "Synthetic image mode is SET!" << endl;
}

. . . .
```

CArcDevice::SetSyntheticImageMode

Syntax:

```
void SetSyntheticImageMode( bool bMode );
```

Namespace:

```
arc::device
```

Description:

Sets the camera controller into synthetic image mode.

Parameters:

bMode

True to turn synthetic image mode on; false to turn off.

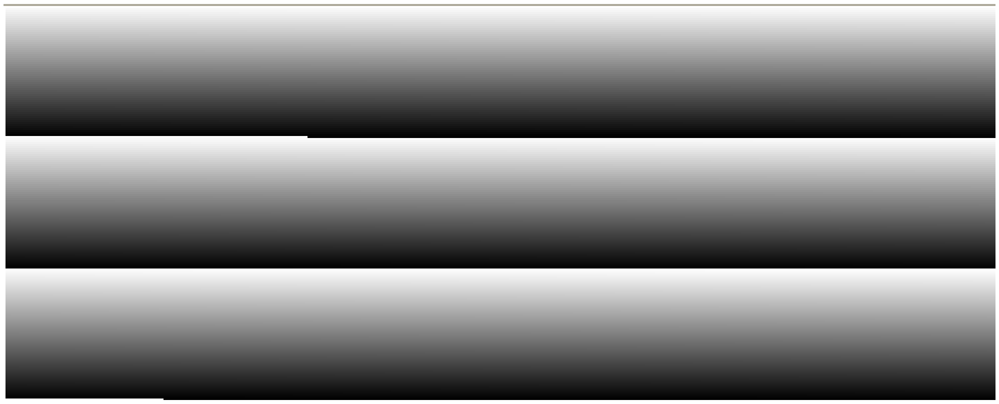
Throws Exception:

std::runtime_error

Return Value	Description
N/A	N/A

Notes:

Synthetic image mode causes the controller DSP to bypass the A/D converters and generate an artificial image pattern. The image data will have the following pattern: 0, 1, 2, 3… 65535, 0, 1, 2, 3… 65535, 0, 1, 2, 3… 65535 … See the figure below for an example of the pattern. The number and size of the pattern depends on the image dimensions.



Usage:

```
CArcDevice *pArcDev = new CArcPCIe();

. . . .

pArcDev->SetSyntheticImageMode( true );

if ( pArcDev->IsSyntheticImageMode() )
{
      cout << “Synthetic image mode is SET!” << endl;
}
. . . .
```

CArcDevice::VerifyImageAsSynthetic

Syntax:

```
void VerifyImageAsSynthetic ( int dRows, int dCols );
```

Namespace:

```
arc::device
```

Description:

Verifies that the data in the image buffer matches the expected pattern for a synthetic image.

Parameters:

dRows

The number of rows in the image.

dCols

The number of columns in the image.

Throws Exception:

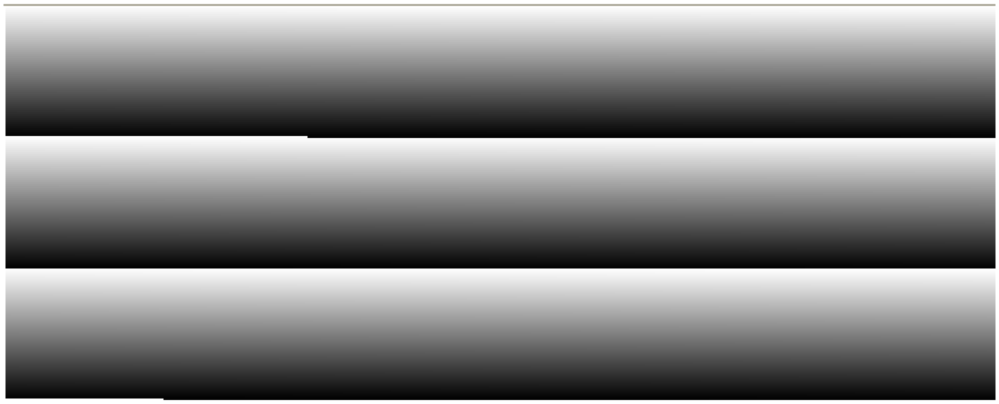
std::runtime_error

Return Value	Description
N/A	N/A

Notes:

Checks that the artificial image pattern generated by the DSP has the following pattern: 0, 1, 2, 3… 65535, 0, 1, 2, 3… 65535, 0, 1, 2, 3… 65535 …

An exception is thrown on the first mismatched value.



Usage:

```
CArcDevice *pArcDev = new CArcPCIe();

. . . .

try {
      pArcDev->SetSyntheticImageMode( true );
      pArcDev->Expose( 0, 1024, 1200 );
      pArcDev->VerifyImageAsSynthetic();
}
catch ( exception& e ) { cerr << e.what() << endl; }
```

CArcDevice::SetOpenShutter

Syntax:

```
void SetOpenShutter( bool bMode );
```

Namespace:

```
arc::device
```

Description:

Determines whether or not to open the shutter during an exposure.

Parameters:

bMode

True to open the shutter during exposure; false to keep it closed.

Throws Exception:

std::runtime_error

Return Value	Description
N/A	N/A

Notes:

N/A

Usage:

```
    CArcDevice *pArcDev = new CArcPCIe();

    . . . .

    pArcDev->SetOpenShutter( true );

    . . . .
```

CArcDevice::Expose

Syntax:

```
void Expose( float fExpTime, int dRows, int dCols, const bool& bAbort, CExpIFace*
pExpIFace, bool bOpenShutter );
```

Namespace:

arc::device

Description:

Starts an image exposure.

Parameters:

fExpTime

The exposure time (in seconds).

dRows

The number of rows in the image.

dCols

The number of columns in the image.

bAbort

External reference to allow the user to abort the method. Default = false

pExpIFace

A *CExpIFace* pointer that can be used to provide elapsed time and pixel count information. Default = NULL

bOpenShutter

Set to true to open the shutter during an exposure. Default = true

Throws Exception:

std::runtime_error

Return Value	Description
N/A	N/A

Notes:

This is a convenience method that handles both the exposure and readout of an image. The elapsed exposure time and pixel count callback methods of the *CExpIFace* parameter (provided it's not NULL) will be used to provide feedback to the user application. The user application may extend the *CExpIFace* class or implement a separate extension class to handle the callback methods.

Usage:

```
class CMyExpIFace : public CExpIFace
{
      void ExposeCallback( float fElapsedTime )
      {
            cout << "Elapsed Time: " << fElapsedTime << endl;
      }

      void ReadCallback( int dPixelCount )
      {
            cout << "Pixel Count: " << dPixelCount << endl;
      }
};


CArcDevice *pArcDev = new CArcPCIe();
CMyExpIFace cMyExpIFace;

. . . .

pArcDev->Expose( 0.5f, 1024, 1200, false, &cMyExpIFace );

. . . .
```

In the above example, the expose and read callbacks will be called from the *Expose()* method during exposure and readout respectively. The *CExpIFace* and *CArcPCIe* classes can be combined into a single class as follows:

```
class CMyPCIe : public CExpIFace, public CArcPCIe
{
      void ExposeCallback( float fElapsedTime )
      {
            cout << "Elapsed Time: " << fElapsedTime << endl;
      }

      void ReadCallback( int dPixelCount )
      {
            cout << "Pixel Count: " << dPixelCount << endl;
      }
};


CMyPCIe cMyPCIe;

. . . .

cMyPCIe.Expose( 0.5f, 1024, 1200, false, &cMyPCIe );

. . . .
```

CArcDevice::StopExposure

Syntax:

```
void StopExposure();
```

Namespace:

```
arc::device
```

Description:

Causes the current exposure to stop.

Parameters:

N/A

Throws Exception:

std::runtime_error

Return Value	Description
N/A	N/A

Notes:

N/A

Usage:

CArcDevice::Continuous

Syntax:

```
void Continuous( int dRows, int dCols, int dNumOfFrames, float fExpTime, const bool& bAbort, CConIFace* pConIFace, bool bOpenShutter );
```

Namespace:

arc::device

Description:

Starts continuous readout.

Parameters:

dRows

The number of rows in each image.

dCols

The number of columns in each image.

dNumOfFrames

The number of frames to read.

fExpTime

The exposure time (in seconds).

bAbort

External reference to allow the user to abort the method. Default = false

pConIFace

A *CConIFace* pointer that can be used to provide frame count information. Default = NULL

bOpenShutter

Set to true to open the shutter during an exposure. Default = true

Throws Exception:

std::runtime_error

Return Value	Description
N/A	N/A

Notes:

True continuous readout (i.e. video mode) does not exist on the camera controller. The number of frames parameter can be any number up to 16777216 (24-bits), which when coupled with an exposure time is generally ample enough to provide what is effectively “continuous” readout.

This is a convenience method that handles both the exposure and readout of a series of images. The frame count callback method of the *CConIFace* parameter (provided it’s not NULL) will be used to provide feedback to the user application. The user application may extend the *CConIFace* class or implement a separate extension class to handle the callback method.

The image buffer is divided into sub-buffers using the specified image size parameters (dRows, dCols). Each sub-buffer starts on a 1k boundary within the main image buffer.

The camera controller is automatically set back to single image mode at the end of this method.

Usage:

```
class CMyConIFace : public CConIFace
{
      CFitsFile* pFits;

      CMyConIFace( int dRows, int dCols )
      {
            pFits = new CFitsFile( “Image.fit”,
                                   dRows,
                                   dCols,
                                   CFitsFile::BPP16,
                                   true );
      }

      ~CMyConIFace() { delete pFits; }

      void FrameCallback( int dFPB, int dCount, int dRows, int dCols, void* pBuf );
      {
            cout << “Saving frame #“ << dCount << endl;

            pFits->Write3D( pBuf );
      }
};


CArcDevice *pArcDev = new CArcPCIe();

. . . .

int dRows = pArcDev->GetImageRows();
int dCols = pArcDev->GetImageCols();

CMyConIFace cMyConIFace( dRows, dCols );

. . . .

pArcDev->Continuous( dRows, dCols, 100, 0.5f, false, &cMyConIFace );

. . . .
```

In the above example, the frame callback will be called from the *Continuous()* method. The *CConIFace* and *CArcPCIe* classes can be combined into a single class as follows:

```
class CMyPCIe : public CConIFace, public CArcPCIe
{
      CFitsFile* pFits;

      CMyConIFace()  { pFits = NULL; }
      ~CMyConIFace() { delete pFits; }

      void FrameCallback( int dFPB, int dCount, int dRows, int dCols, void* pBuf );
      {
            cout << “Saving frame #“ << dCount << endl;

            if ( pFits == NULL )
            {
                  pFits = new CFitsFile( “Image.fit”,
                                          dRows,
                                          dCols,
                                          CFitsFile::BPP16,
                                          true );
            }

            pFits->Write3D( pBuf );
      }
};


CMyPCIe cMyPCIe;

. . . .

cMyPCIe.Continuous( cMyPCIe.GetImageRows(),
                    cMyPCIe.GetImageCols(),
                    100,
                    0.5f,
                    false,
                    &cMyPCIe);

. . . .
```

CArcDevice::StopContinuous

Syntax:

```
void StopContinuous();
```

Namespace:

```
arc::device
```

Description:

Causes the camera controller to stop running in continuous mode and return to single image mode.

Parameters:

N/A

Throws Exception:

std::runtime_error

Return Value	Description
N/A	N/A

Notes:

N/A

Usage:

CArcDevice::IsReadout

Syntax:

```
bool IsReadout();
```

Namespace:

```
arc::device
```

Description:

Returns true if the camera controller is currently reading out an image.

Parameters:

N/A

Throws Exception:

std::runtime_error

Return Value	Description
true	Image readout is in progress
false	Image readout is NOT in progress

Notes:

Except for stop exposure, no commands should be sent to the controller during image readout.

Usage:

```
        CArcDevice* pArcDev = new CArcPCIe();

        . . . .

        if ( !pArcDev->IsReadout() )
        {
              pArcDev->Command( TIM_ID, TDL, 0x112233 );
        }

        . . . .
```

CArcDevice::GetPixelCount

Syntax:

```
int GetPixelCount();
```

Namespace:

```
arc::device
```

Description:

Returns the current pixel count during image readout.

Parameters:

N/A

Throws Exception:

std::runtime_error

Return Value	Description
int	The current pixel count

Notes:

N/A

Usage:

```
CArcDevice* pArcDev = new CArcPCIe();

. . . .

int dPixCnt = 0;
int dRows   = 1024;
int dCols   = 1200;

//
// Start a 4.5 sec exposure
//
pArcDev->Command( TIM_ID, SET, 4500 );
pArcDev->Command( TIM_ID, SEX );

//
// Loop and print pixel count
//
while ( dPixCnt < ( dRows * dCols ) )
{
      dPixCnt = pArcDev->GetPixelCount();

      cout << "Pixel Count: " << dPixCnt << endl;

      Sleep( 500 );
}

. . . .
```

CArcDevice::GetCRPixelCount

Syntax:

```
int GetCRPixelCount();
```

Namespace:

```
arc::device
```

Description:

Returns the current pixel count during continuous readout.

Parameters:

N/A

Throws Exception:

std::runtime_error

Return Value	Description
int	The current pixel count

Notes:

This is the total pixel count across all frames. i.e. this value goes from 0 to (frame count * rows * cols).

Usage:

CArcDevice::GetFrameCount

Syntax:

```
int GetFrameCount();
```

Namespace:

```
arc::device
```

Description:

Returns the current frame count during continuous readout.

Parameters:

N/A

Throws Exception:

std::runtime_error

Return Value	Description
int	The current frame count

Notes:

N/A

Usage:

CArcDevice::SubtractImageHalves

Syntax:

```
void SubtractImageHalves( int dRows, int dCols );
```

Namespace:

arc::device

Description:

Subtracts the first half of an image from the second half.

Parameters:

dRows

The row image dimension (in pixels).

dCols

The column image dimension (in pixels).

Throws Exception:

std::runtime_error

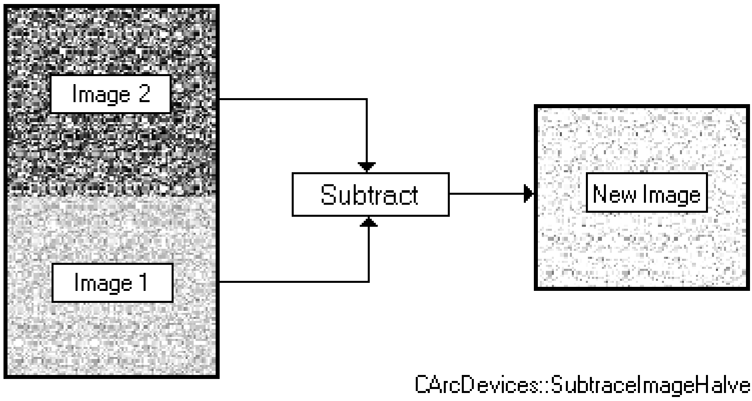
Return Value	Description
N/A	N/A

Notes:

This method is for infrared systems using correlated double sampling (CDS); where the first half of the image contains a read of the array immediately following a reset. This is a noise pattern that is then subtracted from the true image contained within the second half of the image.

The first half of the image is replaced with the new image.

The image must have an equal number of rows or an exception will be thrown.



CArcDevices::SubtraceImageHalves()

Usage:

```
. . . .

pArcDev->Expose( 0.5f, dRows, dCols );

pArcDev->SubtractImageHalves( dRows, dCols );

CFitsFile fits( “Image.fit”, dRows, dCols );
fits.Write( pArcDev->CommonBufferVA() );
```

CArcDevice::ContainsError

Syntax:

```
bool ContainsError( int dWord );

bool ContainsError( int dWord, int dWordMin, int dWordMax );
```

Namespace:

arc::device

Description:

The first version checks the specified value for error replies: timeout, readout, header error, error, system reset, and reset.

The second version checks that the specified value is within the specified range.

Parameters:

dWord

The value (usually a command reply) to check.

dWordMin

The minimum range value (not inclusive).

dWordMax

The maximum range value (not inclusive).

Throws Exception:

N/A

Return Value	Description
true	The value contains an error or is not within the specified range
false	The value doesn't contain any errors or is within the specified range

Notes:

Usage:

```
CArcDevice* pArcDev = new CArcPCIe();

int dReply = pArcDev->Command( TIM_ID, SET, 1000 );

if ( pArcDev->ContainsError( dReply ) ) {
      cerr << "Failed to set exposure time! Reply: 0x"
           << hex << dReply << dec << endl;
}

int dRows = pArcDev->GetImageRows();

if ( pArcDev->ContainsError( dRows, 0, 4200 ) ) {
      cerr << "Invalid row size!" << endl;
}
. . . .
```

CArcDevice::GetNextLoggedCmd

Syntax:

```
const std::string GetNextLoggedCmd();
```

Namespace:

arc::device

Description:

Pops the first message from the command logger and returns it.

Parameters:

N/A

Throws Exception:

N/A

Return Value	Description
const std::string	The first message in the command logger
NULL	The command logger is empty

Notes:

This method is used to log all commands sent to the controller. Logging uses large amounts of memory and should only be used for debugging.

Usage:

```
CArcDevice* pArcDev = new CArcPCIe();

. . . .

pArcDev->SetLogCmds( true );

pArcDev->Command( TIM_ID, SET, 1000 );
pArcDev->Command( TIM_ID, TDL, 0x123456 );
pArcDev->Command( TIM_ID, TDL, 0x112233 );

while ( pArcDev->GetLoggedCmdCount() > 0 )
{
      cout << pArcDev->GetNextLoggedCmd() << endl;
}

pArcDev->SetLogCmds( false );

. . . .
```

The above results in the following output:

```
[ 0x203 SET 1000 -> DON ]
[ 0x203 TDL 0x123456 -> 0x123456 ]
[ 0x203 TDL 0x112233 -> 0x112233 ]
```

CArcDevice::GetLoggedCmdCount

Syntax:

```
int GetLoggedCmdCount();
```

Namespace:

```
arc::device
```

Description:

Returns the available message count.

Parameters:

N/A

Throws Exception:

N/A

Return Value	Description
int	The available message count
0	The logger is empty

Notes:

This method should only be used within a “*while*” loop when used in conjunction with *GetNextLoggedCmd()*, or it will not function properly. The logger uses a queue which shrinks on each call to *GetNextLoggedCmd()*, thus reducing the value of *GetLoggedCmdCount()*. Using this method with a fixed “*for*” loop will result in messages being lost.

Correct Usage: `while ( pArcDev->GetLoggedCmdCount() > 0 ) { … }`

Incorrect Usage: `for ( int i=0; i<pArcDev->GetLoggedCmdCount(); i++ ) { … }`

Usage:

```
CArcDevice* pArcDev = new CArcPCIe();

. . . .

pArcDev->SetLogCmds( true );

pArcDev->Command( TIM_ID, SET, 1000 );
pArcDev->Command( TIM_ID, TDL, 0x123456 );
pArcDev->Command( TIM_ID, TDL, 0x112233 );

while ( pArcDev->GetLoggedCmdCount() > 0 )
{
      cout << pArcDev->GetNextLoggedCmd() << endl;
}

pArcDev->SetLogCmds( false );

. . . .
```

The above results in the following output:

```
[ 0x203 SET 1000 -> DON ]
[ 0x203 TDL 0x123456 -> 0x123456 ]
[ 0x203 TDL 0x112233 -> 0x112233 ]
```

CArcDevice::SetLogCmds

Syntax:

```
void SetLogCmds( bool bOnOff );
```

Namespace:

arc::device

Description:

Turns command logging on/off.

Parameters:

bOnOff

True to turn command logging on; false to turn it off.

Throws Exception:

N/A

Return Value	Description
N/A	N/A

Notes:

This logging can be used for debugging to see command details in the following form:

```
[ <header> <cmd> <arg1> ... <arg4> -> <controller reply> ]
```

Example: `[ 0x203 TDL 0x112233 -> 0x444E4F ]`

Usage:

```
CArcDevice* pArcDev = new CArcPCIe();

. . . .

pArcDev->SetLogCmds( true );

pArcDev->Command( TIM_ID, SET, 1000 );
pArcDev->Command( TIM_ID, TDL, 0x123456 );
pArcDev->Command( TIM_ID, TDL, 0x112233 );

while ( pArcDev->GetLoggedCmdCount() > 0 )
{
      cout << pArcDev->GetNextLoggedCmd() << endl;
}

pArcDev->SetLogCmds( false );

. . . .
```

<continued on next page>

The above results in the following output:

```
[ 0x203 SET 1000 -> DON ]
[ 0x203 TDL 0x123456 -> 0x123456 ]
[ 0x203 TDL 0x112233 -> 0x112233 ]
```

CArcDevice::GetArrayTemperature

Syntax:

```
double GetArrayTemperature();
```

Namespace:

arc::device

Description:

Returns the average array temperature (in Celcius).

Parameters:

N/A

Throws Exception:

std::runtime_error

Return Value	Description
double	The average array temperature (in Celcius)

Notes:

The temperature is read *CArcDevice::gTmpCtrl_SDNumberOfReads* (protected class member) times and averaged. Also, for a read to be included in the average, the difference between the target temperature and the actual temperature must be less than the tolerance specified by *CArcDevice::gTmpCtrl_SDTolerance* (protected class member).

Usage:

```
      CArcDevice* pArcDev = new CArcPCIe();

      . . . .

      double gTemp = pArcDev->GetArrayTemperature();

      cout << "The current array temperature ( C ): "
           << gTemp << endl;

      . . . .
```

CArcDevice::GetArrayTemperatureDN

Syntax:

```
double GetArrayTemperatureDN();
```

Namespace:

arc::device

Description:

Returns the digital number (ADU) associated with the current array temperature.

Parameters:

N/A

Throws Exception:

std::runtime_error

Return Value	Description
double	The current digital number

Notes:

Usage:

```
      CArcDevice* pArcDev = new CArcPCIe();

      . . . .

      double gTempDN = pArcDev->GetArrayTemperatureDN();

      cout << “The current temperature digital number: “
           << gTempDN << endl;

      . . . .
```

CArcDevice::SetArrayTemperature

Syntax:

```
void SetArrayTemperature( double gTempVal );
```

Namespace:

arc::device

Description:

Sets the array temperature to regulate around.

Parameters:

gTempVal

The temperature value (in Celcius).

Throws Exception:

std::runtime_error

Return Value	Description
N/A	N/A

Notes:

Usage:

```
CArcDevice* pArcDev = new CArcPCIe();

. . . .

pArcDev->SetArrayTemperature( -100.6 );

. . . .

double gTemp = pArcDev->GetArrayTemperature();

cout << "The current array temperature ( C ): "
     << gTemp << endl;

. . . .
```

CArcDevice::LoadTemperatureCtrlData

Syntax:

```
void LoadTemperatureCtrlData( const std::string sFilename );
```

Namespace:

arc::device

Description:

Loads temperature control constants from the specified file.

Parameters:

pszFilename

The file containing temperature control constants in the correct format.

Throws Exception:

std::runtime_error

Return Value	Description
N/A	N/A

Notes:

The default constants are stored in *TempCtrl.h* and cannot be permanently overwritten. This means any loaded file will need to be reloaded whenever a new *CArcDevice* object is created.

The file format is too detailed to show here. The best way to create a temperature control constant file is to save the existing constants using *CArcDevice::SaveTemperatureCtrlData()*. The saved file can then be modified and reloaded.

Usage:

```
CArcDevice* pArcDev = new CArcPCIe();

. . . .

pArcDev->LoadTemperatureCtrlData( “MyTempCtrlFile.txt” );

. . . .
```

CArcDevice::SaveTemperatureCtrlData

Syntax:

```
void SaveTemperatureCtrlData( const std::string sFilename );
```

Namespace:

arc::device

Description:

Saves the current temperature control constants to the specified file.

Parameters:

pszFilename

The file to save the data too.

Throws Exception:

std::runtime_error

Return Value	Description
N/A	N/A

Notes:

The default constants are stored in *TempCtrl.h* and cannot be permanently overwritten.

Usage:

```
CArcDevice* pArcDev = new CArcPCIe();

. . . .

pArcDev->SaveTemperatureCtrlData( “MyTempCtrlFile.txt” );

. . . . Modify the file contents . . . .

pArcDev->LoadTemperatureCtrlData( “MyTempCtrlFile.txt” );

. . . .
```

CArcPCIe Only Methods

This section documents details of the methods only available through the ARC PCIe class (see CArcPCIe.h). The following is a list of these methods; with details to follow on subsequent pages:

CArcPCIe Method Name	C Interface Name	Description
FindDevices	ArcDevice_FindDevices	Searches for all ARC-66/67 PCIe boards in the system. Note: C interface function returns all ARC PCI & PCIe boards found.
DeviceCount	ArcDevice_DeviceCount	Returns the number of ARC-66/67 PCIe boards found in the system. Note: C interface function returns the number of all ARC PCI & PCIe boards found.
GetDeviceStringList	ArcDevice_GetDeviceStringList	Returns the device list as an array of strings. Note: C Interface returns a list of all ARC PCI & PCIe boards found.
IsFiberConnected	N/A	Returns true if the specified fiber is connected correctly
WriteBar	N/A	Resets a device
ReadBar	N/A	Maps the kernel device buffer so the user application may access it

CArcPCIe::FindDevices

Syntax:

```
void FindDevices();
```

Namespace:

```
arc::device
```

Description:

Searches the system for available ARC PCIe (ARC-66/67) devices.

Parameters:

N/A

Throws Exception:

std::runtime_error

Return Value	Description
N/A	N/A

Notes:

This static class method MUST be called before any PCIe device can be opened (accessed).

The resulting list is stored and allows devices to be accessed via device number (0,1,2...) as a parameter to the *Open()* method. The list itself can be read via the *PCIe::DeviceCount()* and *PCIe::GetDeviceStringList()* methods.

Usage:

```
      CArcDevice* pArcDev = NULL;

      // Find all PCIe devices
      CArcPCIe::FindDevices();

      // List all PCIe devices found
      const string* pDevList = CArcPCIe::GetDeviceStringList();

      for ( int i=0; i<CArcPCIe::DeviceCount(); i++ )
      {
            cout << pDevList[ i ] << endl;
      }

      // Open the first PCIe device found
      if ( CArcPCIe::DeviceCount() > 0 )
      {
            pArcDev = new CArcPCIe();

            pArcDev->Open( 0, 2200 * 2200 * 2 );
      }

      . . . .
```

CArcPCIe::DeviceCount

Syntax:

```
int DeviceCount();
```

Namespace:

```
arc::device
```

Description:

Returns the number of ARC PCIe devices found in the system.

Parameters:

N/A

Throws Exception:

std::runtime_error

Return Value	Description
int	The device count

Notes:

Can be used to access *PCIe::GetDeviceStringList()* elements or verify that a device has been found.

Usage:

```
        CArcDevice* pArcDev = NULL;

        // Find all PCIe devices
        //
        CArcPCIe::FindDevices();

        // List all PCIe devices found
        //
        const string pDevList = CArcPCIe::GetDeviceStringList();

        for ( int i=0; i<CArcPCIe::DeviceCount(); i++ )
        {
              cout << pDevList[ i ] << endl;
        }

        // Open the first PCIe device found
        //
        if ( CArcPCIe::DeviceCount() > 0 )
        {
              pArcDev = new CArcPCIe();

              pArcDev->Open( 0, 2200 * 2200 * 2 );
        }

        . . . .
```

CArcPCIe::GetDeviceStringList

Syntax:

```
const string* GetDeviceStringList();
```

Namespace:

```
arc::device
```

Description:

Returns the list of ARC PCIe devices found in the system.

Parameters:

N/A

Throws Exception:

std::runtime_error

Return Value	Description
const string*	The device list
“No Devices Found”	The device list is empty

Notes:

The user should call *PCIe::FreeDeviceStringList()* when finished with the returned list.

Usage:

```
        CArcDevice* pArcDev = NULL;

        // Find all PCIe devices
        //
        CArcPCIe::FindDevices();

        // List all PCIe devices found
        //
        const string* sDevList = CArcPCIe::GetDeviceStringList();

        for ( int i=0; i<CArcPCIe::DeviceCount(); i++ )
        {
              cout << sDevList[ i ] << endl;
        }

        // Open the first PCIe device found
        //
        if ( CArcPCIe::DeviceCount() > 0 )
        {
              pArcDev = new CArcPCIe();

              pArcDev->Open( 0, 2200 * 2200 * 2 );
        }

        . . . .
```

CArcPCIe::IsFiberConnected

Syntax:

```
bool IsFiberConnected( int dFiber );
```

Namespace:

```
arc::device
```

Description:

Returns true if the specified PCIe fiber optic is connected to a powered-on controller.

Parameters:

dFiber

An integer identifying the fiber (A or B) to check. Default = CArcPCIe::FIBER_A

Throws Exception:

std::runtime_error

Return Value	Description
true	The specified fiber is connected correctly
false	The specified fiber is not connected correctly *or* no controller is connected and powered-on

Notes:

The parameter can be one of *CArcPCIe::FIBER_A* or *CArcPCIe::FIBER_B*.

NOT ALL PCIe boards have two receive fibers installed.

Usage:

```
CArcDevice* pArcDev = new CArcPCIe();

. . . .

if ( pArcDev->IsFiberConnected() )
{
      cout << "Controller connected properly!" << endl;
}
else
{
      cerr << "No controller connected, powered-on, or connected improperly!";
}

. . . .
```

CArcPCIe::WriteBar

Syntax:

```
void WriteBar( int dBar, int dOffset, int dValue );
```

Namespace:

```
arc::device
```

Description:

Write a value to a PCIe base address register (BAR).

Parameters:

dBar

The base address register number.

dOffset

The offset within the base address register.

dValue

The value to write.

Throws Exception:

std::runtime_error

Return Value	Description
N/A	N/A

Notes:

In general, this method should never be called by a user application.

The dBar parameter can be one of the values:

CArcPCIe::LCL_CFG_BAR	(Local Configuration Registers)
CArcPCIe::DEV_REG_BAR	(FPGA (Device) Registers)

The dOffset parameter can be on the values:

CArcPCIe::REG_CMD_HEADER	(Command Header Register)
CArcPCIe::REG_CMD_COMMAND	(Command Register)
CArcPCIe::REG_CMD_ARG0	(Command Argument #1 Register)
CArcPCIe::REG_CMD_ARG1	(Command Argument #2 Register)
CArcPCIe::REG_CMD_ARG2	(Command Argument #3 Register)
CArcPCIe::REG_CMD_ARG3	(Command Argument #4 Register)
CArcPCIe::REG_CMD_ARG4	(Command Argument #5 Register)
CArcPCIe::REG_INIT_IMG_ADDR	(Image Buffer Physical Address Register)
CArcPCIe::REG_STATUS	(Status Register)
CArcPCIe::REG_CMD_REPLY	(Command Reply Register)
CArcPCIe::REG_CTLR_ARG1	(Controller Argument Register #1)
CArcPCIe::REG_CTLR_ARG2	(Controller Argument Register #2)
CArcPCIe::REG_PIXEL_COUNT	(Image Pixel Count Register)

<continued on next page>

CArcPCIe::REG_FRAME_COUNT	(Continuous Readout Frame Count Register)
CArcPCIe::REG_ID_LO	(Device ID LSW Register)
CArcPCIe::REG_ID_HI	(Device ID MSW Register)
CArcPCIe::REG_CTLR_SPECIAL_CMD	(Controller Special Command Register)

Usage:

```
CArcDevice* pArcDev = new CArcPCIe();

. . . .

//
//  Send a TDL command to the controller
// +--------------------------------------------------+

//
// Send the command header
//
pArcDev->WriteBar( DEV_REG_BAR, REG_CMD_HEADER, 0x203 );

//
// Send the command
//
pArcDev->WriteBar( DEV_REG_BAR, REG_CMD_COMMAND, TDL );

//
// Send the argument
//
pArcDev->WriteBar( DEV_REG_BAR, REG_CMD_ARG0, 0x112233 );

. . . .

//
//  Instead, this should be done as follows:
// +--------------------------------------------------+
pArcDev->Command( TIM_ID, TDL, 0x112233 );

. . . .
```

CArcPCIe::ReadBar

Syntax:

```
int ReadBar( int dBar, int dOffset );
```

Namespace:

```
arc::device
```

Description:

Read a value from a PCIe base address register (BAR).

Parameters:

dBar

The base address register number.

dOffset

The offset within the base address register.

Throws Exception:

std::runtime_error

Return Value	Description
int	Value of base address register (dBar + dOffset)

Notes:

In general, this method should never be called by a user application.

The dBar parameter can be one of the values:

CArcPCIe::LCL_CFG_BAR	(Local Configuration Registers)
CArcPCIe::DEV_REG_BAR	(FPGA (Device) Registers)

The dOffset parameter can be on the values:

CArcPCIe::REG_CMD_HEADER	(Command Header Register)
CArcPCIe::REG_CMD_COMMAND	(Command Register)
CArcPCIe::REG_CMD_ARG0	(Command Argument #1 Register)
CArcPCIe::REG_CMD_ARG1	(Command Argument #2 Register)
CArcPCIe::REG_CMD_ARG2	(Command Argument #3 Register)
CArcPCIe::REG_CMD_ARG3	(Command Argument #4 Register)
CArcPCIe::REG_CMD_ARG4	(Command Argument #5 Register)
CArcPCIe::REG_INIT_IMG_ADDR	(Image Buffer Physical Address Register)
CArcPCIe::REG_STATUS	(Status Register)
CArcPCIe::REG_CMD_REPLY	(Command Reply Register)
CArcPCIe::REG_CTLR_ARG1	(Controller Argument Register #1)
CArcPCIe::REG_CTLR_ARG2	(Controller Argument Register #2)
CArcPCIe::REG_PIXEL_COUNT	(Image Pixel Count Register)
CArcPCIe::REG_FRAME_COUNT	(Continuous Readout Frame Count Register)
CArcPCIe::REG_ID_LO	(Device ID LSW Register)

<continued on next page>

CArcPCIe::REG_ID_HI	(Device ID MSW Register)
CArcPCIe::REG_CTLR_SPECIAL_CMD	(Controller Special Command Register)

Usage:

```
CArcDevice* pArcDev = new CArcPCIe();

. . . .

//
//  Send the status register
//
int dStatus = pArcDev->ReadBar( DEV_REG_BAR, REG_STATUS );

cout << “PCIe status: 0x” << hex << dStatus << dec << endl;

. . . .

//
//  Instead, this should be done as follows:
//
int dStatus = pArcDev->GetStatus();

cout << “PCIe status: 0x” << hex << dStatus << dec << endl;

. . . .
```

CArcPCIe Data Structures, Types and Constants

This section documents details of the structures, data types and constants used by the ARC PCIe class.

`CArcPCIe::FIBER_A`
`CArcPCIe::FIBER_B`

Type:

`Integer`

Namespace:

`arc::device`

Description:

Parameter for the *CArcPCIe::IsFiberConnected()* method. Fiber A is the standard receive fiber, while fiber B is the second receive fiber.

Note that not all PCIe boards have a second fiber installed, as this is for non-standard applications.

`CArcPCIe::ID`

Type:

`Integer`

Namespace:

`arc::device`

Description:

Static class constant that matches the MSW ID register (Reg@7CH) on the PCIe board.

The value of the ID constant is (‘ARC6’): 0x41524336

CArcPCI Only Methods

This section documents details of the methods only available through the ARC PCI class (see CArcPCI.h). The following is a list of these methods with details to follow on subsequent pages:

CArcPCI Method Name	C Interface Name	Description
FindDevices	ArcDevice_FindDevices	Searches for all ARC-62/64 PCI boards in the system. Note: C interface function returns all ARC PCI & PCIe boards found.
DeviceCount	ArcDevice_DeviceCount	Returns the number of ARC 62/64 PCI boards found in the system. Note: C interface function returns the number of all ARC PCI & PCIe boards found.
GetDeviceStringList	ArcDevice_GetDeviceStringList	Returns the device list as an array of strings. Note: C Interface returns a list of all ARC PCI & PCIe boards found.
SetHctr	N/A	Sets the value of the PCI Host Control (HCTR) register
GetHstr	N/A	Returns the vallue of the PCI Status (HSTR) register
GetHctr	N/A	Returns the value of the PCI Host Control (HCTR) register
PCICommand	N/A	Sends a vector command to the PCI board
IoctlDevice	N/A	Sends an I/O command to the PCI board

CArcPCI::FindDevices

Syntax:

```
void FindDevices();
```

Namespace:

```
arc::device
```

Description:

Searches the system for available ARC PCI (ARC-63/64) devices.

Parameters:

N/A

Throws Exception:

std::runtime_error

Return Value	Description
N/A	N/A

Notes:

This static class method MUST be called before any PCI device can be opened (accessed).

The resulting list is stored and allows devices to be accessed via device number (0,1,2...) as a parameter to the *Open()* method. The list itself can be read via the *PCI::DeviceCount()* and *PCI::GetDeviceStringList()* methods.

Usage:

```
      CArcDevice* pArcDev = NULL;

      // Find all PCI devices
      CArcPCI::FindDevices();

      // List all PCI devices found
      const std::string pDevList = CArcPCI::GetDeviceStringList();

      for ( int i=0; i<CArcPCI::DeviceCount(); i++ )
      {
            cout << pDevList[ i ] << endl;
      }

      // Open the first PCI device found
      if ( CArcPCI::DeviceCount() > 0 )
      {
            pArcDev = new CArcPCI();

            pArcDev->Open( 0, 2200 * 2200 * 2 );
      }

      . . . .
```

CArcPCI::DeviceCount

Syntax:

```
int DeviceCount();
```

Namespace:

```
arc::device
```

Description:

Returns the number of ARC PCI devices found in the system.

Parameters:

N/A

Throws Exception:

std::runtime_error

Return Value	Description
int	The device count

Notes:

Can be used to access *PCI::GetDeviceStringList()* elements or verify that a device has been found.

Usage:

```
CArcDevice* pArcDev = NULL;

// Find all PCI devices
//
CArcPCI::FindDevices();

// List all PCI devices found
//
const std::string pDevList = CArcPCI::GetDeviceStringList();

for ( int i=0; i<CArcPCI::DeviceCount(); i++ )
{
      cout << pDevList[ i ] << endl;
}

// Open the first PCI device found
//
if ( CArcPCI::DeviceCount() > 0 )
{
      pArcDev = new CArcPCI();

      pArcDev->Open( 0, 2200 * 2200 * 2 );
}

. . . .
```

CArcPCI::GetDeviceStringList

Syntax:

```
const std::string* GetDeviceStringList();
```

Namespace:

```
arc::device
```

Description:

Returns the list of ARC PCI devices found in the system.

Parameters:

N/A

Throws Exception:

std::runtime_error

Return Value	Description
const std::string*	The device list
“No Devices Found”	The device list is empty

Notes:

The user should call *PCI::FreeDeviceStringList()* when finished with the returned list.

Usage:

```
CArcDevice* pArcDev = NULL;

// Find all PCI devices
//
CArcPCI::FindDevices();

// List all PCI devices found
//
const std::string pDevList = CArcPCI::GetDeviceStringList();

for ( int i=0; i<CArcPCI::DeviceCount(); i++ )
{
      cout << pDevList[ i ] << endl;
}

// Open the first PCI device found
//
if ( CArcPCI::DeviceCount() > 0 )
{
      pArcDev = new CArcPCI();

      pArcDev->Open( 0, 2200 * 2200 * 2 );
}

. . . .
```

CArcPCI::SetHctr

Syntax:

```
void SetHctr( int dVal );
```

Namespace:

```
arc::device
```

Description:

Sets the DSP Host Control Register (HCTR).

Parameters:

dVal

The value to write to the register.

Throws Exception:

std::runtime_error

Return Value	Description
N/A	N/A

Notes:

N/A

Usage:

N/A

CArcPCI::GetHctr

Syntax:

```
int GetHctr();
```

Namespace:

```
arc::device
```

Description:

Reads the DSP Host Control Register (HCTR).

Parameters:

N/A

Throws Exception:

std::runtime_error

Return Value	Description
int	The HCTR value

Notes:

N/A

Usage:

N/A

CArcPCI::GetHstr

Syntax:

```
int GetHstr();
```

Namespace:

```
arc::device
```

Description:

Reads the DSP Host Status Register (HSTR).

Parameters:

N/A

Throws Exception:

std::runtime_error

Return Value	Description
int	The HSTR value

Notes:

The status bits are had by masking the return value from the HSTR with *CArcPCI::HTF_BIT_MASK*. This is what the *CArcPCI::GetStatus()* method does.

The status bits match one of the following:

```
enum {
      TIMEOUT_STATUS = 0,
      DONE_STATUS,
      READ_REPLY_STATUS,
      ERROR_STATUS,
      SYSTEM_RESET_STATUS,
      READOUT_STATUS,
      BUSY_STATUS
};
```

Usage:

```
CArcDevice* pArcDev = new CArcPCI();

. . . .

int dHstr = ( pArcDev->GetHstr() & CArcPCI::HTF_BIT_MASK ) >> 3;

if ( dHstr == CArcPCI::DONE_STATUS )
{
      cout << CArcTools::FormatString( "Status: 0x%X [ DON ]", dHstr );
}

else if ( dHstr == CArcPCI::ERROR_STATUS )
{
      cout << CArcTools::FormatString( "Status: 0x%X [ ERR ]", dHstr );
}
. . . .
```

CArcPCI::PCICommand

Syntax:

```
int PCICommand( int dCommand );
```

Namespace:

```
arc::device
```

Description:

Sends a command directly to the PCI board.

Parameters:

dCommand

A PCI specific command.

Throws Exception:

std::runtime_error

Return Value	Description
int	The command reply; typically ‘DON’.

Notes:

Valid PCI commands:

PCI Command	Description	Reply
PCI_RESET	Reset the PCI board	‘DON’
ABORT_READOUT	Stop image readout	‘DON’
RESET_CONTROLLER	Reset the camera controller	‘SYR’

Usage:

```
CArcDevice* pArcDev = new CArcPCI();

. . . .

// Reset the controller
//
int dReply = pArcDev->PCICommand( RESET_CONTROLLER );

if ( dReply != SYR )
{
      cerr << “Reset Controller Failed!” << endl;
}

. . . .

// Or could have just done the following
//
pArcDev->ResetController();

. . . .
```

CArcPCI::IoctlDevice

Syntax:

```
int IoctlDevice( int dIoctlCmd, int dArg );
```

Namespace:

```
arc::device
```

Description:

Sends a command to the PCI device driver.

Parameters:

dIoctlCmd

A PCI device driver command.

dArg

Any argument required by the command. Default = -1

Throws Exception:

std::runtime_error

Return Value	Description
int	The command reply

Notes:

<continued on next page>

Valid IOCTL commands:

IOCTL Command	Description
ASTROPCI_GET_HCTR	Read the DSP HCTR (Host Control Register)
ASTROPCI_GET_PROGRESS	Read the current pixel count
ASTROPCI_GET_DMA_ADDR	Read the image buffer physical address LSW (for info only)
ASTROPCI_GET_HSTR	Read the DSP HSTR (Host Status Register)
ASTROPCI_MEM_MAP	Map the image buffer into the user application (Windows Only)
ASTROPCI_GET_DMA_SIZE	Read the image buffer size (in bytes)
ASTROPCI_GET_FRAMES_READ	Read the current frame count (continuous readout only)
ASTROPCI_HCVR_DATA	Write data to the DSP HCVR data register
ASTROPCI_SET_HCTR	Write to the DSP HCTR (Host Control Register)
ASTROPCI_SET_HCVR	Write to the DSP HCVR (Host Vector Register)
ASTROPCI_PCI_DOWNLOAD	Set the PCI board into download mode
ASTROPCI_PCI_DOWNLOAD_WAIT	Wait for the PCI board to finish entering download mode
ASTROPCI_MEM_UNMAP	UnMap the image buffer from the user application (Windows Only)
ASTROPCI_ABORT	Stop the current image exposure/readout
ASTROPCI_CONTROLLER_DOWNLOAD	Set the camera controller into download mode
ASTROPCI_GET_CR_PROGRESS	Read the current pixel count (continuous readout only)
ASTROPCI_GET_DMA_LO_ADDR	Read the image buffer physical address LSW (for info only)
ASTROPCI_GET_DMA_HI_ADDR	Read the image buffer physical address MSW (for info only)
ASTROPCI_GET_CONFIG_BYTE	Read a byte from the configuration space header
ASTROPCI_GET_CONFIG_WORD	Read a word from the configuration space header
ASTROPCI_GET_CONFIG_DWORD	Read a double word from the configuration space header
ASTROPCI_SET_CONFIG_BYTE	Write a byte to the configuration space header
ASTROPCI_SET_CONFIG_WORD	Write a word to the configuration space header
ASTROPCI_SET_CONFIG_DWORD	Write a double word to the configuration space header

Usage:

```
CArcDevice* pArcDev = new CArcPCI();

. . . .

// Read the current pixel count
//
int dPixelCount = pArcDev->IoctlDevice( ASTROPCI_GET_PROGRESS );

cout << “Pixel Count: ” << dPixelCount << endl;

. . . .

// Should actually do it this way:
//
pArcDev->GetPixelCount();

. . . .
```

CArcPCI::IoctlDevice

Syntax:

```
int IoctlDevice( int dIoctlCmd, int dArg[], int dArgCount );
```

Namespace:

```
arc::device
```

Description:

Sends a command to the PCI device driver.

Parameters:

dIoctlCmd

A PCI device driver command.

dArg

Array of arguments required by the command. The size of the array depends on the command.

dArgCount

The number of elements in the dArg parameter.

Throws Exception:

std::runtime_error

Return Value	Description
int	The command reply

Notes:

<continued on next page>

Valid IOCTL commands:

IOCTL Command	Description
ASTROPCI_GET_HCTR	Read the DSP HCTR (Host Control Register)
ASTROPCI_GET_PROGRESS	Read the current pixel count
ASTROPCI_GET_DMA_ADDR	Read the image buffer physical address LSW (for info only)
ASTROPCI_GET_HSTR	Read the DSP HSTR (Host Status Register)
ASTROPCI_MEM_MAP	Map the image buffer into the user application (Windows Only)
ASTROPCI_GET_DMA_SIZE	Read the image buffer size (in bytes)
ASTROPCI_GET_FRAMES_READ	Read the current frame count (continuous readout only)
ASTROPCI_HCVR_DATA	Write data to the DSP HCVR data register
ASTROPCI_SET_HCTR	Write to the DSP HCTR (Host Control Register)
ASTROPCI_SET_HCVR	Write to the DSP HCVR (Host Vector Register)
ASTROPCI_PCI_DOWNLOAD	Set the PCI board into download mode
ASTROPCI_PCI_DOWNLOAD_WAIT	Wait for the PCI board to finish entering download mode
ASTROPCI_COMMAND	Send a command to the PCI or camera controller
ASTROPCI_MEM_UNMAP	UnMap the image buffer from the user application (Windows Only)
ASTROPCI_ABORT	Stop the current image exposure/readout
ASTROPCI_CONTROLLER_DOWNLOAD	Set the camera controller into download mode
ASTROPCI_GET_CR_PROGRESS	Read the current pixel count (continuous readout only)
ASTROPCI_GET_DMA_LO_ADDR	Read the image buffer physical address LSW (for info only)
ASTROPCI_GET_DMA_HI_ADDR	Read the image buffer physical address MSW (for info only)
ASTROPCI_GET_CONFIG_BYTE	Read a byte from the configuration space header
ASTROPCI_GET_CONFIG_WORD	Read a word from the configuration space header
ASTROPCI_GET_CONFIG_DWORD	Read a double word from the configuration space header
ASTROPCI_SET_CONFIG_BYTE	Write a byte to the configuration space header
ASTROPCI_SET_CONFIG_WORD	Write a word to the configuration space header
ASTROPCI_SET_CONFIG_DWORD	Write a double word to the configuration space header

Usage:

```
CArcDevice* pArcDev = new CArcPCI();

. . . .

// Send a TDL to the controller
//
int cmdData[ CTLR_CMD_MAX ] = { 0x203,
                                TDL,
                                0x112233,
                                -1,
                                -1,
                                -1 };

int dReply = pArcDev->IoctlDevice( ASTROPCI_COMMAND,
                                   cmdData,
                                   CTLR_CMD_MAX );

if ( dReply != 0x112233 )
{
      cerr << "TDL failed!" << endl;
}

. . . .

// Should actually do it this way:
//
pArcDev->Command( TIM_ID, TDL, 0x112233 );

. . . .
```

CExpIFace Interface

This section documents details of the methods available through the *CExpIFace* class (see CExpIFace.h). This class is an abstract interface that provides exposure callbacks for user applications. The user may extend this class and pass it into the *CArcDevice::Expose()* method for elapsed time and pixel count information.

The following is a list of these methods; with details to follow on subsequent pages:

CExpIFace Method Name	C Interface Name	Description
ExposeCallback	N/A	Called during an image exposure to pass back the elapsed time
ReadCallback	N/A	Called during image readout to pass back the current pixel count

CExpIFace::ExposeCallback

Syntax:

```
void ExposeCallback( float fElapsedTime );
```

Namespace:

```
arc::device
```

Description:

Called from the *CArcDevice::Expose()* method to supply the application with elapsed time info.

Parameters:

fElapsedTime

The current elapsed time.

Throws Exception:

std::runtime_error

Return Value	Description
N/A	N/A

Notes:

This class must be sub-classed by the user application. The sub-class can then be passed into the *CArcDevice::Expose()* method. This is the only way to get elapsed exposure time info from the *CArcDevice::Expose()* method.

Usage:

```
class CExpInfo : public CExpIFace
{
      void ExposeCallback( float fElapsedTime )
      {
            cout << "Elapsed Time: " << fElapsedTime << endl;
      }

      void ReadCallback( int dPixelCount )
      {
            cout << "Pixel Count: " << dPixelCount << endl;
      }

} cExpInfo;

. . . .

CArcDevice* pArcDev = new CArcPCIe();

. . . .

pArcDev->Expose( 0.5f, dRows, dCols, false, &cExpInfo );

. . . .
```

The *CExpIFace* and *CArcPCI(e)* classes can be simultaneously sub-classed.

For example, to sub-class *CArcPCIe*:

```
class CMyDev : public CExpIFace, public CArcPCIe
{
      void ExposeCallback( float fElapsedTime )
      {
            cout << "Elapsed Time: " << fElapsedTime << endl;
      }

      void ReadCallback( int dPixelCount )
      {
            cout << "Pixel Count: " << dPixelCount << endl;
      }

};

. . . .

CMyDev* pArcDev = new CMyDev();

. . . .

pArcDev->Expose( 0.5f, dRows, dCols, false, pArcDev );

. . . .
```

CExpIFace::ReadCallback

Syntax:

```
void ReadCallback( int dPixelCount );
```

Namespace:

```
arc::device
```

Description:

Called from the *CArcDevice::Expose()* method to supply the application with pixel count info.

Parameters:

dPixelCount

The current pixel count.

Throws Exception:

std::runtime_error

Return Value	Description
N/A	N/A

Notes:

This class must be sub-classed by the user application. The sub-class can then be passed into the *CArcDevice::Expose()* method. This is the only way to get pixel count info from the *CArcDevice::Expose()* method.

Usage:

```
class CExpInfo : public CExpIFace
{
      void ExposeCallback( float fElapsedTime )
      {
            cout << "Elapsed Time: " << fElapsedTime << endl;
      }

      void ReadCallback( int dPixelCount )
      {
            cout << "Pixel Count: " << dPixelCount << endl;
      }

} cExpInfo;

. . . .

CArcDevice* pArcDev = new CArcPCIe();

. . . .

pArcDev->Expose( 0.5f, dRows, dCols, false, &cExpInfo );

. . . .
```

The *CExpIFace* and *CArcPCI(e)* classes can be simultaneously sub-classed.

For example, to sub-class *CArcPCIe*:

```
class CMyDev : public CExpIFace, public CArcPCIe
{
      void ExposeCallback( float fElapsedTime )
      {
            cout << “Elapsed Time: ” << fElapsedTime << endl;
      }

      void ReadCallback( int dPixelCount )
      {
            cout << “Pixel Count: ” << dPixelCount << endl;
      }

};

. . . .

CMyDev* pArcDev = new CMyDev();

. . . .

pArcDev->Expose( 0.5f, dRows, dCols, false, pArcDev );

. . . .
```

CConIFace Interface

This section documents details of the methods available through the *CConIFace* class (see CExpIFace.h). This class is an abstract interface that provides continuous readout callbacks for user applications. The user may extend this class and pass it into the *CArcDevice::Continuous()* method for frame count and buffer information.

The following is a list of these methods; with details to follow on subsequent pages:

CConIFace Method Name	C Interface Name	Description
FrameCallback	N/A	Called during continuous readout to pass back the current frame count

CConIFace::FrameCallback

Syntax:

```
void FrameCallback( int dFPB, int dFrameCount, int dRows, int dCols, void* pBuffer );
```

Namespace:

```
arc::device
```

Description:

Called from the *CArcDevice::Continous()* method to supply the application with frame count and image buffer info.

Parameters:

dFPB

The frames-per-buffer.

dFrameCount

The frame count.

dRows

The number of rows in the frame image.

dCols

The number of columns in the frame image.

pBuffer

Pointer to frame image.

Throws Exception:

std::runtime_error

Return Value	Description
N/A	N/A

Notes:

This class must be sub-classed by the user application. The sub-class can then be passed into the *CArcDevice::Continous()* method. This is the only way to get frame info from the *CArcDevice::Continous()* method.

<continued on next page>

Usage:

```
class CExpInfo : public CConIFace
{
      void FrameCallback( int dFPB, int dFrameCount, int dRows, int dCols,
                          void* pBuffer )
      {
            cout << “Frame Count: “ << dFrameCount << endl;

            //  Save the image to FITS
            // +---------------------------------------+
            CFitsFile fits( “Image.fit”, dRows, dCols );
            fits.Write( pBuffer );
      }

} cExpInfo;

. . . .

CArcDevice* pArcDev = new CArcPCIe();

. . . .

pArcDev->Continuous( dRows, dCols, dNumOfFrames, 0.5f, false, &cExpInfo );

. . . .
```

The *CConIFace* and *CArcPCI(e)* classes can be simultaneously sub-classed.

For example, to sub-class *CArcPCIe*:

```
class CMyDev : public CConIFace, public CArcPCIe
{
      void FrameCallback( int dFPB, int dFrameCount, int dRows, int dCols,
                          void* pBuffer )
      {
            cout << “Frame Count: “ << dFrameCount << endl;

            //  Save the image to FITS
            // +---------------------------------------+
            CFitsFile fits( “Image.fit”, dRows, dCols );
            fits.Write( pBuffer );
      }

};

. . . .

CMyDev* pArcDev = new CMyDev();

. . . .

pArcDev-> Continuous( dRows, dCols, dNumOfFrames, 0.5f, false, &pArcDev );

. . . .
```

CArcTools Methods

This section documents details of the methods available through the *CArcTools* class (see CArcTools.h). This provides utility functions used by the library and user applications.

Note that all methods are class methods, that is, all methods are static.

The following is a list of these methods; with details to follow on subsequent pages:

CArcTools Method Name	C Interface Name	Description
ReplyToString	N/A	Converts a command reply to an ASCII string representation
CmdToString	N/A	Converts a command and its arguments to a string representation
StringToCmd	N/A	Converts an ASCII command string to its integer representation
FormatString	N/A	Formats a string; similar to sprintf
StringToUpper	N/A	Converts a string to uppercase
GetSystemMessage	N/A	Returns a system dependent error message for the specified error code
ConvertIntToString	N/A	Converts an integer to a string
ConvertWideToAnsi	N/A	Converts a unicode string to ansi
ConvertAnsiToWide	N/A	Converts an ansi string to unicode
StringToHex	N/A	Converts a string to a hexadecimal ingeter value
StringToChar	N/A	Converts a string to a single character
ThrowException	N/A	Throws a std::runtime_error with the class and method names included as part of the message.

CArcTools::ReplyToString

Syntax:

```
std::string ReplyToString( int dReply );
```

Namespace:

arc

Description:

Returns the std::string representation of the specified command reply.

Parameters:

dReply

The command reply to convert to a std::string.

Throws Exception:

std::runtime_error

Return Value	Description
std::string	A text version of the reply parameter

Notes:

The hexadecimal value of the reply is returned as a character string if the reply is not a standard value.

Example: dReply = 0x455252 -> returns “ERR”

Example: dReply = 0x112233 -> returns “0x112233”

Usage:

```
CArcDevice* pArcDev = new CArcPCIe();

. . . .

int dReply = pArcDev->Command( TIM_ID, WRM, ( X_MEM | 0x3 ) );

//
// Outputs “WRM reply: DON” on success
//
cout << “WRM reply: “ << CArcTools::ReplyToString( dReply );

. . . .

dReply = pArcDev->Command( TIM_ID, TDL, 0x123456 );

//
// Outputs “TDL reply: 0x123456” on success
//
cout << “TDL reply: “ << CArcTools::ReplyToString( dReply );

. . . .
```

CArcTools::CmdToString

Syntax:

```
std::string CmdToString( int dCmd );
```

Namespace:

```
arc
```

Description:

Returns the std::string representation of the specified command.

Parameters:

dCmd

The command to convert to a std::string.

Throws Exception:

N/A

Return Value	Description
std::string	A text version of the command parameter

Notes:

The hexadecimal value of the command is returned as a character string if the command is not a three letter ASCII command.

Example: dCmd = 0x54444C -> returns "TDL"

Example: dCmd = 0x112233 -> returns "0x112233"

Usage:

```
CArcDevice* pArcDev = new CArcPCIe();

. . . .

cout << "Sending " << CArcTools::CmdToString( WRM ) << endl;
int dReply = pArcDev->Command( TIM_ID, WRM, ( X_MEM | 0x3 ) );

. . . .

cout << "Sending " << CArcTools::CmdToString( TDL ) << endl;
dReply = pArcDev->Command( TIM_ID, TDL, 0x123456 );

. . . .
```

CArcTools::CmdToString

Syntax:

```
std::string CmdToString( int dReply, int dBoardId, int dCmd, int dArg1,
                         int dArg2, int dArg3, int dArg4, int dSysErr );
```

Namespace:

arc

Description:

Method used to bundle command values into a string.

Parameters:

dReply

The command reply.

dBoardId

The command board ID.

dCmd

The command.

dArg1

The command argument #1.

dArg2

The command argument #2.

dArg3

The command argument #3.

dArg4

The command argument #4.

dSysErr

The system error code if the command failed.

Throws Exception:

N/A

Return Value	Description
std::string	A text version of the command

<continued on next page>

Notes:

The command is returned as a character string of the following form:

```
[ CmdHeader   Cmd  Arg1  Arg2  Arg3  Arg4 ] -> Reply \n System message
```

Usage:

```
CArcTools::CmdToString( 0x112233, TIM_ID, TDL, 0x112233 );
```

Produces the following output:

```
[ 0x203 TDL 0x112233 ] -> 0x112233
```

```
CArcTools::CmdToString( ERR, TIM_ID, TDL, 0x112233 );
```

Produces the following output:

```
[ 0x203 TDL 0x112233 ] -> ERR
```

CArcTools::StringToCmd

Syntax:

```
int StringToCmd( string sCmd );
```

Namespace:

```
arc
```

Description:

Method to convert an ASCII command string, such as 'TDL' to the equivalent integer value.

Parameters:

sCmd

The command string to convert.

Throws Exception:

std::runtime_error

Return Value	Description
int	The integer version of the command string parameter

Notes:

Throws std::runtime_error if ASCII command parameter is not three characters in length.

Example: sCmd = “TDL” -> returns 0x54444C

Example: sCmd = “112233” -> returns 0x112233

Usage:

```
string sCmd;

cout << “Enter a command: “;
cin  >> sCmd;
cout << endl;

int dCmd = CArcTools::StringToCmd( sCmd );

cout << “The user entered the command: 0x”
     << hex << dCmd << dec << endl;
```

CArcTools::FormatString

Syntax:

```
std::string FormatString( const char *pszFmt, ... );
```

Namespace:

```
arc
```

Description:

Method to format a std::string using C printf-style formatting.

Parameters:

pszFmt

A printf-style format string, followed by variables.

Throws Exception:

N/A

Return Value	Description
std::string	The formatted string

Notes:

Acceptable format parameters:

Format Specifier	Description
%d	integer
%f	double
%s	char * string
%e	system message
%x or %X	lower or upper case hexadecimal integer

Usage:

```
cout << “Two plus Two equals: ”
     << CArcTools::FormatString( “%d”, ( 2 + 2 ) )
     << endl;
```

CArcTools::StringToUpper

Syntax:

```
const std::string StringToUpper( std::string sStr );
```

Namespace:

arc

Description:

Function to transform a string into all uppercase letters.

Parameters:

sStr

The string to convert.

Throws Exception:

N/A

Return Value	Description
const std::string	The converted string

Notes:

Usage:

```
cout << “The string \”lowercase\” as uppercase: ”
     << CArcTools::StringToUpper( “lowercase” )
     << endl;
```

CArcTools::GetSystemMessage

Syntax:

```
std::string GetSystemMessage( int dCode );
```

Namespace:

arc

Description:

Used to get a formatted message string from the specified system error code.

Parameters:

dCode

A system error code.

Throws Exception:

N/A

Return Value	Description
std::string	The system error code string

Notes:

Usage:

CArcTools::ConvertWideToAnsi

Syntax:

```
std::string ConvertWideToAnsi( wchar_t wcharString[] );
```

Namespace:

arc

Description:

Converts the specified wide char string (unicode) to an ANSI std::string.

Parameters:

wcharString

Wide character string to be converted to std::string.

Throws Exception:

N/A

Return Value	Description
std::string	The converted string

Notes:

Usage:

CArcTools::ConvertWideToAnsi

Syntax:

```
std::string ConvertWideToAnsi( const std::wstring& wsString );
```

Namespace:

arc

Description:

Converts the specified wide string (unicode) to an ansi std::string.

Parameters:

wsString

Wide string to be converted to std::string.

Throws Exception:

N/A

Return Value	Description
std::string	The converted string

Notes:

Usage:

CArcTools::ConvertAnsiToWide

Syntax:

```
std::wstring ConvertAnsiToWide( const char *szString );
```

Namespace:

arc

Description:

Converts the specified ANSI char string to a unicode std::wstring.

Parameters:

szString

ANSI C character string to be converted to std::wstring.

Throws Exception:

N/A

Return Value	Description
std::wstring	The converted wide string

Notes:

Usage:

CArcTools::ConvertIntToString

Syntax:

```
std::string ConvertIntToString( int dNumber );
```

Namespace:

arc

Description:

Converts the specified integer value to a std::string.

Parameters:

dNumber

The integer to convert to std::string.

Throws Exception:

N/A

Return Value	Description
std::string	The converted string

Notes:

This is a convenience method.

Usage:

```
      cout << “The number 10 as a string: “
           << CArcTools::ConvertIntToString( 10 )
           << endl;
```

CArcTools::StringToHex

Syntax:

```
long StringToHex( std::string sStr );
```

Namespace:

arc

Description:

Converts the specified std::string to a long integer value.

Parameters:

sStr

The std::string to convert.

Throws Exception:

N/A

Return Value	Description
long	The converted long value

Notes:

This is a convenience method.

Usage:

```
      cout << "The string \"10\" as a hex value: "
           << CArcTools::StringToHex( "10" )
           << endl;
```

CArcTools::StringToChar

Syntax:

```
char StringToChar( std::string sStr );
```

Namespace:

arc

Description:

Converts the specified std::string, which represents a single character, to a C char value.

Parameters:

sStr

The std::string to convert to a char.

Throws Exception:

N/A

Return Value	Description
char	The converted character

Notes:

This is a convenience method.

Usage:

```
char c = CArcTools::StringToHex( “P” );

cout << “c = “ << c << endl;
```

CArcTools::ThrowException

Syntax:

```
void ThrowException( string sClassName, string sMethodName, string sMsg );
```

Namespace:

arc

Description:

Throws a std::runtime_error based on the supplied cfitsion status value.

Parameters:

sClassName

Name of the class where the exception occurred.

sMethodName

Name of the method where the exception occurred.

sMsg

The exception message.

Throws Exception:

std::runtime_error

Return Value	Description
N/A	N/A

Notes:

Throws a std::runtime_error exception with the message formatted as follows:

(ClassName::ClassMethod()): Message

If the sClassName parameter is empty, then the string “?Class?” will be used. Similarly, if the sMethodName parameter is empty, then the string “?Method?” will be used.

For Example: “ (CArcDevice::Command()): Incorrect reply: 0x112233”

For Example: “ (?Class?::?Method?()): Some message goes here”

Usage:

```
CArcDevice* pArcDev = new CArcPCIe();

int dReply = pArcDev->Command( TIM_ID, TDL, 0x112233 );

if ( dReply != 0x112233 )
{
      CArcTools::ThrowException( “CArcDevice”, “Command”, “TDL Failed!” );
}
```

CArcTools::ThrowException

Syntax:

```
void ThrowException( string sClassName, string sMethodName, const char *pszFmt, ... );
```

Namespace:

arc

Description:

Method uses printf-style formatting that is then used to throw a std::runtime_error exception.

Parameters:

sClassName

Name of the class where the exception occurred.

sMethodName

Name of the method where the exception occurred.

pszFmt

A C printf-style format string, followed by variables.

Throws Exception:

std::runtime_error

Return Value	Description
N/A	N/A

Notes:

Throws a std::runtime_error exception with the message formatted as follows:

(ClassName::ClassMethod()): Message

If the sClassName parameter is empty, then the string "?Class?" will be used. Similarly, if the sMethodName parameter is empty, then the string "?Method?" will be used.

Acceptable format parameters:

Format Specifier	Description
%d	integer
%f	double
%s	char * string
%e	system message
%x or %X	lower or upper case hexadecimal integer

For Example: " (CArcDevice::Command()): Incorrect reply: 0x112233"

For Example: " (?Class?::?Method?()): Some message goes here"

Usage:

```
CArcDevice* pArcDev = new CArcPCIe();

. . . .

int dReply = pArcDev->Command( TIM_ID, TDL, 0x112233 );

if ( dReply != 0x112233 )
{
      CArcTools::ThrowException( “CArcDevice”,
                                 “Command”,
                                 “TDL failed, reply: 0x%X”,
                                  dReply );
}

. . . .
```

CArcTools::CTokenizer Class

This section documents details of the methods available through the *CArcTools::CTokenizer* class (see CArcTools.h). This class provides a string tokenizer that uses string streams instead of the older C *strtok()* function. The string is split by whitespace only.

The following is a list of these methods; with details to follow on subsequent pages:

CTokenizer Method Name	C Interface Name	Description
CTokenizer	N/A	Class constructor
Victim	N/A	Sets the string to be split by whitespace
Next	N/A	Returns the next token from the split string
IsEmpty	N/A	Returns true if there are no more tokens to return

CArcTools::CTokenizer

Syntax:

```
CarcTools::CTokenizer();
```

Namespace:

arc

Description:

Default class constructor. Seperates a string into individual tokens deliminated by spaces.

Parameters:

N/A

Throws Exception:

N/A

Return Value	Description
N/A	N/A

Notes:

Class to separate a string deliminated spaces.

Usage:

```
CArcTools::CTokenizer tokenizer = new CArcTools::CTokenizer();

tokenizer.Victim( “This is a message!” );

while ( !tokenizer.IsEmpty() )
{
      cout << “Token: “ << tokenizer.Next() << endl;
}

. . . .
```

Results in the following output:

```
Token: This
Token: is
Token: a
Token: message!
```

CArcTools::CTokenizer::Victim

Syntax:

```
void Victim( std::string str );
```

Namespace:

arc

Description:

Method used to break a string into individual tokens.

Parameters:

str

The string to parse.

Throws Exception:

std::runtime_error

Return Value	Description
N/A	N/A

Notes:

Seperates the specified string deliminated by spaces.

Usage:

```
CArcTools::CTokenizer tokenizer = new CArcTools::CTokenizer();

tokenizer.Victim( "This is a message!" );

while ( !tokenizer.IsEmpty() )
{
      cout << "Token: " << tokenizer.Next() << endl;
}

. . . .
```

Results in the following output:

```
Token: This
Token: is
Token: a
Token: message!
```

CArcTools::CTokenizer::Next

Syntax:

```
std::string Next();
```

Namespace:

arc

Description:

Method used to return the next token.

Parameters:

N/A

Throws Exception:

std::runtime_error

Return Value	Description
std::string	The next token from the string

Notes:

Seperates the specified string deliminated by spaces.

Usage:

```
CArcTools::CTokenizer tokenizer = new CArcTools::CTokenizer();

tokenizer.Victim( “This is a message!” );

while ( !tokenizer.IsEmpty() )
{
      cout << “Token: ‶ << tokenizer.Next() << endl;
}

. . . .
```

Results in the following output:

```
Token: This
Token: is
Token: a
Token: message!
```

CArcTools::CTokenizer::IsEmpty

Syntax:

```
bool IsEmpty();
```

Namespace:

arc

Description:

Method used to determine if there are anymore tokens available.

Parameters:

N/A

Throws Exception:

std::runtime_error

Return Value	Description
true	More tokens are available
false	No more tokens are available

Notes:

Usage:

```
CArcTools::CTokenizer tokenizer = new CArcTools::CTokenizer();

tokenizer.Victim( “This is a message!” );

while ( !tokenizer.IsEmpty() )
{
      cout << “Token: “ << tokenizer.Next() << endl;
}

. . . .
```

<u>Results in the following output</u>:

```
Token: This
Token: is
Token: a
Token: message!
```

General Command and Controller Constants and Macros (ArcDefs.h)

This section documents details of the command and controller constants and macros as defined in *ArcDefs.h*.

PCI_ID

Type:

`Integer`

Namespace: arc

Description:

PCI(e) board id. Defined as 1.

TIM_ID

Type:

`Integer`

Namespace: arc

Description:

Timing board id. Defined as 2.

UTIL_ID

Type:

`Integer`

Namespace: arc

Description:

Utility board id. Defined as 3.

SMALLCAM_DLOAD_ID

Type:

`Integer`

Namespace: arc

Description:

SmallCam DSP download id. Defined as 3.

X_MEM

Type:

`Integer`

Namespace: arc

Description:

DSP X memory space. Used as part of the address parameter for read ('RDM') and write ('WRM') memory commands.

Y_MEM

Type:

`Integer`

Namespace: arc

Description:

DSP Y memory space. Used as part of the address parameter for read ('RDM') and write ('WRM') memory commands.

P_MEM

Type:

`Integer`

Namespace: arc

Description:

DSP program memory space. Used as part of the address parameter for read ('RDM') and write ('WRM') memory commands.

R_MEM

Type:

`Integer`

Namespace: arc

Description:

DSP ROM. Used as part of the address parameter for read ('RDM') and write ('WRM') memory commands.

DON

Type:

`Integer`

Namespace: arc

Description:

Success reply. Most device/controller commands return DON on success. See the command description document for details on commands and replies.

Defined as 0x444F4E

ERR

Type:

`Integer`

Namespace: arc

Description:

Error reply. See the command description document for details on commands and replies.

Defined as 0x455252

SYR

Type:

`Integer`

Namespace: arc

Description:

System reset reply. This reply means a system reset occurred. The *CArcDevice::ResetController()* method return SYR on success. See the command description document for details on commands and replies.

Defined as 0x535952

RST

Type:

`Integer`

Namespace: arc

Description:

Reset reply. See the command description document for details on commands and replies.

Defined as 0x525354

HERR

Type:

`Integer`

Namespace: arc

Description:

Header error reply. This reply means the command header is improperly formatted. See the command description document for details on commands and replies.

Defined as 0x48455252

TOUT

Type:

`Integer`

Namespace: arc

Description:

Timeout reply. This reply means the device or controller did not respond with a reply within a reasonable amount of time. See the command description document for details on commands and replies.

Defined as 0x544F5554

ROUT

Type:

`Integer`

Namespace: arc

Description:

Readout reply. This reply means the controller is currently reading an image. See the command description document for details on commands and replies.

Defined as 0x524F5554

IS_ARC12

Type:

```
Macro
```

Namespace: arc

Syntax:

```
IS_ARC12( int id );
```

Parameter:

Integer ID as returned from *CArcDevice::GetControllerId()*.

Description:

Macro that returns true if the ID parameter represents the SmallCam (ARC-12) controller. Returns false otherwise.

C Application Interface

This section documents details of the C interface part of each library. The C interface provides a wrapper to most of the ARC API library functions and is exported by each C++ library.

In general, with the exception of a status parameter (see below), the C function parameters match those of the “wrapped” class method, which are detailed in the class documentation. For example, to see the details of the parameters for the “ArcDevice_Command” function, see the CArcDevice “Command” method documentation.

In general, all functions in the C interface have the same name as the “wrapped” class counterpart, but are prefixed with a name matching the class. For example, the CArcDevice C functions all begin with “ArcDevice_”.

IMPORTANT NOTE: The `ArcDevice_FindDevices` function MUST be called before any `ArcDevice_OpenXXX` function, or open will fail. The `ArcDevice_GetDeviceStringList` function returns a list of device strings. See below for details.

ARC C Interface Status

The status parameter required by most functions is an integer pointer that is used to report any errors that may have occurred during the execution of the function. The status value will be `ARC_STATUS_OK` on success and `ARC_STATUS_ERROR` on failure. The status should be checked after every function call. If the status equals failure, then the error message may be retrieved by calling the `ArcXXX_GetLastError` function.

For example, the following shows how to check the status after sending a 'TDL' command:

```
int dStatus = 0;
int dResult = 0;

. . . .

dResult = ArcDevice_Command( TIM_ID, TDL, 0x112233, &dStatus );

if ( dStatus == ARC_STATUS_ERROR )
{
      printf( “Error: %s\n”, ArcDevice_GetLastError() );
}

. . . .
```

Device Initialization

The `ArcDevice_FindDevices` function MUST be called before any open function. The `ArcDevice_GetDeviceStringList` and its associated functions may also be useful for listing all available ARC devices if more than one exist in the system.

For example, the following shows how to correctly open a device:

<continued on next page>

```
std::string* pDevList = NULL;
int dStatus = 0;
int i = 0;

ArcDevice_FindDevices( &dStatus );

pDevList = ArcDevice_GetDeviceStringList( &dStatus );

if ( pDevList == NULL || dStatus == ARC_STATUS_ERROR )
{
      printf( “Error: %s\n”, ArcDevice_GetLastError() );

      return dStatus;
}

for ( i=0; i<ArcDevice_DeviceCount(); i++ )
{
      printf( “Device[ %d ]: %s\n”, i, pDevList[ i ] );
}

ArcDevice_Open( 0, ( 2200 * 2200 * 2 ), &dStatus );

if ( dStatus == ARC_STATUS_ERROR )
{
      printf( “Error: %s\n”, ArcDevice_GetLastError() );

      return dStatus;
}

. . . .
```

Simple Example

This section demonstrates a simple use of the ARC API libraries.

```
// +----------------------------------------------------------------------------------+
// |  File:  ArcAPITest.cpp
// +----------------------------------------------------------------------------------+
// |  Description:  This file demonstates a simple use of the ARC API 3.5 for both
// |  the PCI and PCIe interfaces. The first device found is used to setup an attached
// |  controller and take an exposure.
// |
// |  Author:       Scott Streit
// |  Date:         October 9, 2012
// +----------------------------------------------------------------------------------+
#include <iostream>
#include <iomanip>
#include <memory>
#include <string>
#include "CArcDevice.h"
#include "CArcPCIe.h"
#include "CArcPCI.h"
#include "CArcDeinterlace.h"
#include "CArcFitsFile.h"
#include "CExpIFace.h"

using namespace std;
using namespace arc::device;
using namespace arc::deinterlace;
using namespace arc::fits;


#define USAGE( x ) \
            ( cout << endl << "Usage: " << x \
                        << " [PCI | PCIe] [options]" << endl << endl \
                        << "\toptions:" << endl \
                        << "\t----------------------------------------" << endl \
                        << "\t-f [DSP lod filename : Default=tim.lod] " << endl \
                        << "\t-e [exp time (s) : Default=0.5]" << endl \
                        << "\t-r [rows : Default=512] " << endl \
                        << "\t-c [cols : Default=600]" << endl \
                        << "\t-d [deint alg : Default=CDeinterlace::DEINTERLACE_NONE]" \
                        << endl << endl << "\tDeinterlace Values:" \
                        << "\n\t----------------------------------------" \
                        << "\n\t0: None\n\t1: Parallel\n\t2: Serial" \
                        << "\n\t3: CCD Quad\n\t4: IR Quad\n\t5: CDS IR QUAD" \
                        << "\n\t6: Hawaii RG" \
                        << "\n\t7: STA1600" << endl << endl )


// ------------------------------------------------------
//  Function prototypes
// ------------------------------------------------------
std::string SetDots( const char *cStr );


// ------------------------------------------------------
//  Exposure Callback Class
// ------------------------------------------------------
class CExposeListener : public CExpIFace
{
      void ExposeCallback( float fElapsedTime )
      {
            cout << "Elapsed Time: " << fElapsedTime << endl;
      }

      void ReadCallback( int dPixelCount )
      {
            cout << "Pixel Count: " << dPixelCount << endl;
      }
};

// ------------------------------------------------------
//  Main program
// ------------------------------------------------------
int main( int argc, char **argv )
{
      std::string sTimFile     = "tim.lod";
      float       fExpTime     = 0.5;
      long        lNumOfFrames = 100;
      long        lRows        = 512;
      long        lCols        = 600;
      long        lDeintAlg    = CArcDeinterlace::DEINTERLACE_NONE;
      bool        bAbort       = false;

      CExposeListener cExposeListener;

      //
      // Set host device
      //
      if ( argc < 2 )
      {
            cout << "Error: Invalid number of minimum parameters!" << endl;
            USAGE( argv[ 0 ] );
            cout << endl;

            exit( EXIT_FAILURE );
      }

      string sDev = argv[ 1 ];

      if ( sDev.compare( "PCIe" ) != 0 && sDev.compare( "PCI" ) != 0 )
      {
            cout << "Error: Invalid device parameter: " << sDev << endl;
            USAGE( argv[ 0 ] );
            cout << endl;

            exit( EXIT_FAILURE );
      }

      //
      // Handle program arguments
      //
      for ( int i=2; i<argc; i++ )
      {
            std::string sArgv = argv[ i ];

            if ( sArgv.compare( "-f" ) == 0 && argc >= ( i + 1 ) )
            {
                  sTimFile = argv[ i + 1 ];
                  cout << "Timing File Set: " << sTimFile << endl;
            }
            else if ( sArgv.compare( "-e" ) == 0 && argc >= ( i + 1 ) )
            {
                  fExpTime = float( atof( argv[ i + 1 ] ) );
                  cout << "Exposure Time Set: " << fExpTime << endl;
            }

            else if ( sArgv.compare( "-r" ) == 0 && argc >= ( i + 1 ) )
            {
                  lRows = atol( argv[ i + 1 ] );
                  cout << "Number Of Rows Set: " << lRows << endl;
            }
            else if ( sArgv.compare( "-c" ) == 0 && argc >= ( i + 1 ) )
            {
                  lCols = atol( argv[ i + 1 ] );
                  cout << "Number Of Cols Set: " << lCols << endl;
            }
            else if ( sArgv.compare( "-d" ) == 0 && argc >= ( i + 1 ) )
            {
                  lDeintAlg = atol( argv[ i + 1 ] );
                  cout << "Deinterlace Set: " << lDeintAlg << endl;
            }
            else if ( sArgv.compare( "-h" ) == 0 )
            {
                  USAGE( argv[ 0 ] );
                  exit( EXIT_FAILURE );
            }
      }

      //
      // Create an instance of the ARC Device
      //
      auto_ptr<CArcDevice> pArcDev( new CArcPCIe );

      if ( sDev.compare( "PCI" ) == 0 )
      {
            pArcDev.reset( new CArcPCI );
      }

      cout << endl;

      try
      {
            //
            // Find all ARC PCI(e) device
            //
            cout << SetDots( "Finding devices" );
            if ( sDev.compare( "PCIe" ) == 0 ) { CArcPCIe::FindDevices(); }
            else { CArcPCI::FindDevices(); }
            cout << "done!" << endl;

            //
            // Open a driver/device connection
            //
            cout << SetDots( "Opening first device" );
            pArcDev.get()->Open( 0, 4200 * 4200 * sizeof( unsigned short ) );
            cout << "done! Image Buffer Size: " << pArcDev.get()->CommonBufferVA()
                 << endl;

            //
            // Setup the controller
            //
            cout << SetDots( "Setting up controller" );
            pArcDev.get()->SetupController( true,                  // Reset Controller
                                            true,                  // Test Data Link
                                            true,                  // Power On
                                            lRows,                 // Image row size
                                            lCols,                // Image col size
                                            sTimFile.c_str() );   // DSP timing file
            cout << "done!" << endl;

            //
            // Expose
            //
            pArcDev.get()->Expose( fExpTime, lRows, lCols, bAbort, &cExposeListener );

            //
            // Deinterlace the image
            //
            cout << SetDots( "Deinterlacing image" );
            CArcDeinterlace cDlacer;

            cDlacer.RunAlg( pArcDev.get()->CommonBufferVA(),
                            lRows,
                            lCols,
                            lDeintAlg );

            cout << "done!" << endl;

            //
            // Save the image to FITS
            //
            cout << SetDots( "Writing FITS" );
            CArcFitsFile cFits( "Image.fit", lRows, lCols );
            cFits.Write( pArcDev.get()->CommonBufferVA() );
            cout << "done!" << endl;

            //
            // Close the device connection
            //
            cout << SetDots( "Closing device" );
            pArcDev.get()->Close();
            cout << "done!" << endl;
      }
      catch ( std::runtime_error &e )
      {
            cout << "failed!" << endl;
            cerr << endl << e.what() << endl;

            if ( pArcDev.get()->IsReadout() )
            {
                  pArcDev.get()->StopExposure();
            }

            pArcDev.get()->Close();
      }
      catch ( ... )
      {
            cerr << endl << "Error: unknown exception occurred!!!" << endl;

            if ( pArcDev.get()->IsReadout() )
            {
                  pArcDev.get()->StopExposure();
            }

            pArcDev.get()->Close();
      }
}
```

```
// +--------------------------------------------------------------------------------
// |  SetDots
// +--------------------------------------------------------------------------------
// |  This function just prints dots (...) for the output.
// +--------------------------------------------------------------------------------
std::string SetDots( const char *cStr )
{
      std::string sStr( cStr );

      for ( int i=sStr.length(); i<40; i++ )
            sStr.append( "." );

      return sStr;
}
```