User's Manual

Scott Streit
Astronomical Research Cameras, Inc.
3547 Camino del Rio South, Suite A
San Diego, CA 92108

Updated October 24, 2006

I. Requirements

Owl is available for Windows 2000/XP, Linux, and Solaris. The software should be compatible with all controller versions, but has only been tested on Generation II, III, and IV systems. The program contains a bundled copy of version 5 of the Java Runtime Environment. So the user is not required to download and install Java. Owl is only compatible with Java version 5 or higher. Also, the controller API is a C++ based library that may require some Linux and Solaris users to download and install the GNU C++ standard libraries if your system does not already contain them. The libraries are available here: http://gcc.gnu.org/. The installation of the software is platform dependent, but each version contains an install script or setup program to aid in the installation.

II. Main Application Bar

The main application bar is the basic control center for the application. All sub-windows and scripts can be accessed through the main application bar. See figure 1.

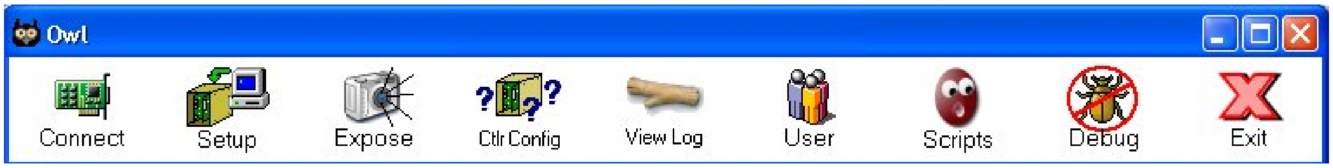


Figure 1. The Owl main application bar.

The main application bar can be made to be horizontal or vertical by “grabbing” a corner of the window and dragging it toward the desired orientation. The window will then “snap” into place. See figure 1b for an example.

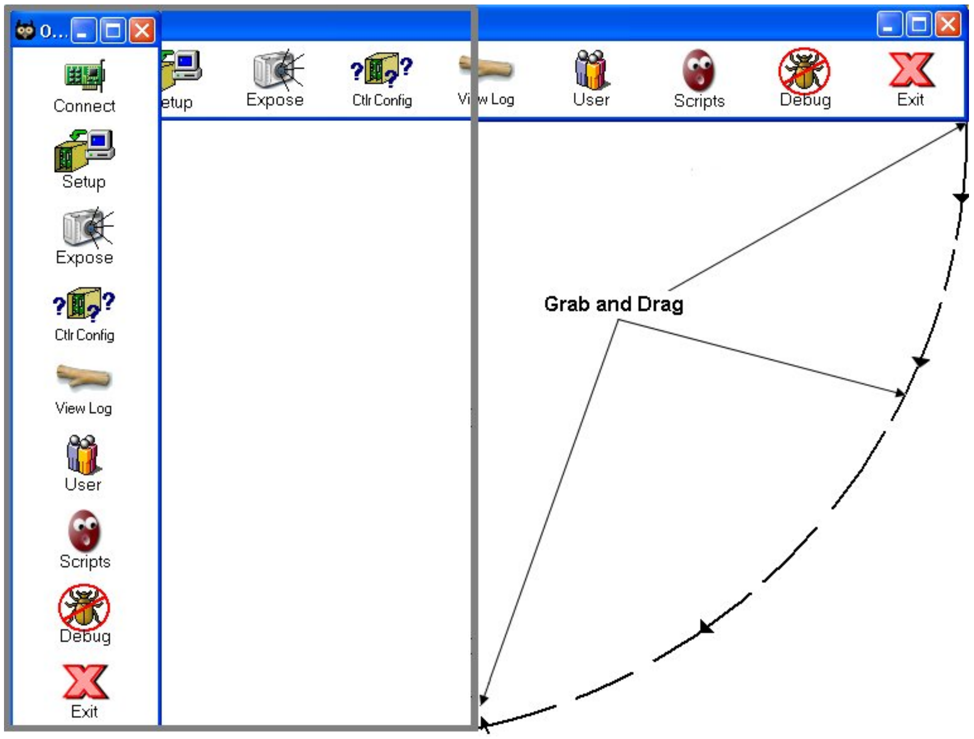


Figure 1b. The primary toolbar can have it’s orientation set by dragging the window in the appropriate direction.

Connect

Controls which driver the application will use if multiple PCI boards are installed in the system. Note: The PCI board must be physically in the computer in order to have a driver appear as an option for that board. See figure 2. To determine which driver belongs to which board can be done by connecting a controller to one of the boards and sending a Test Data Link (TDL) command to the timing board (using the Debug window). If the command succeeds, then you know which board belongs to which driver.

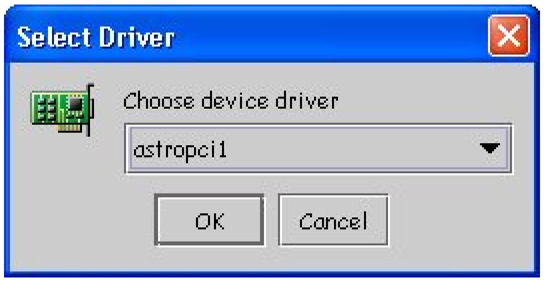


Figure 2. The driver connect window.

Setup

This is one of two "dual purpose" buttons on the main application bar. More generically, this button is used to apply a camera controller setup. Clicking on the button will either open a standard controller setup window or directly apply a setup using an attached script. The selection between script and standard window is done by *right-clicking* on the button and selecting the desired option from the popup menu. See figures 3 and 4. Please refer to the scripts section for details on using scripts. Note: the Owl/Scripts directory contains two standard setup scripts called *StandardSetup.bsh* and *ARC22_Setup.bsh* that can be used as examples for writing your own setup scripts.

Another important feature of the standard setup window is in the save sub-window. If you click on save and give a filename that ends with *.bsh*, then the parameters in the setup window will be used to create a Bean Shell script that can be used like any other script. This can be useful if you want to quickly create a script that can be attached directly to the setup window or if you need to create multiple setup scripts. Note that the default save operation is to create a standard setup text file that can be loaded back into the window using the *load* button. This text file is designed to be compatible with the older Voodoo setup; thus any existing setup file you have from Voodoo should be loadable here. You must specify a *.bsh* extension for the file if you wish to save the setup as a script.

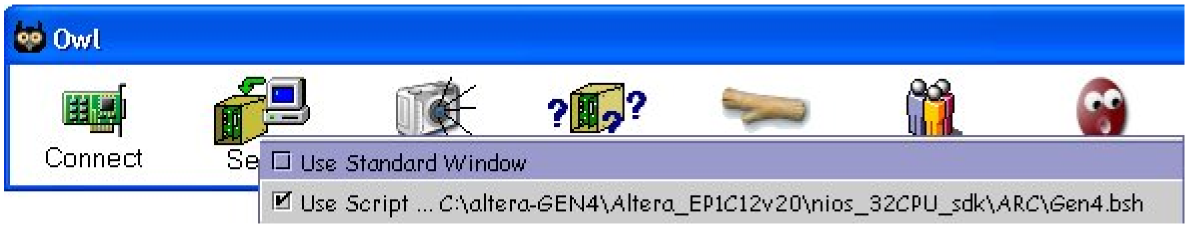

Owl
Connect
Se
☐ Use Standard Window
☑ Use Script ... C:\altera-GEN4\Altera_EP1C12v20\nios_32CPU_sdk\ARC\Gen4.bsh

Figure 3. Selection of standard window or script as target for the setup button.

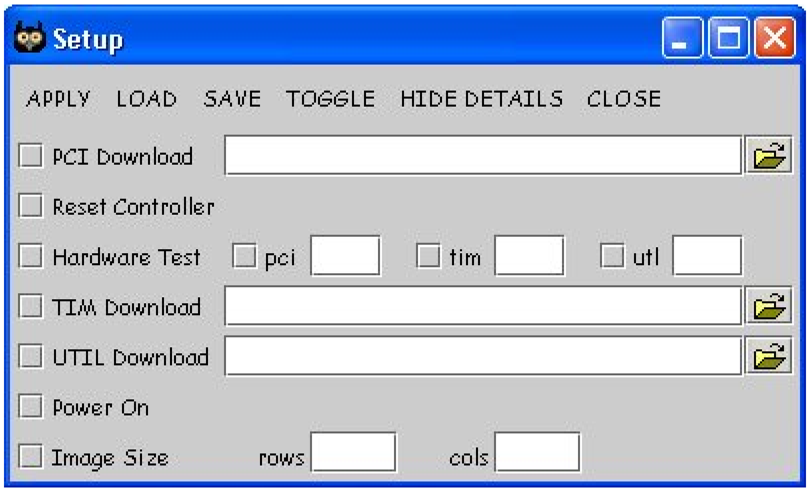


Figure 4. Standard camera controller setup window.

<u>Expose</u>

This is one of two "dual purpose" buttons on the main application bar. More generically, this button is used to take an image exposure. Clicking on the button will either open a standard exposure control window or directly make an exposure using an attached script. The selection between the script and standard window is done by *right-clicking* on the button and selecting the desired option from the popup menu. See figures 5 and 6. Please refer to the scripts section for details on using scripts.

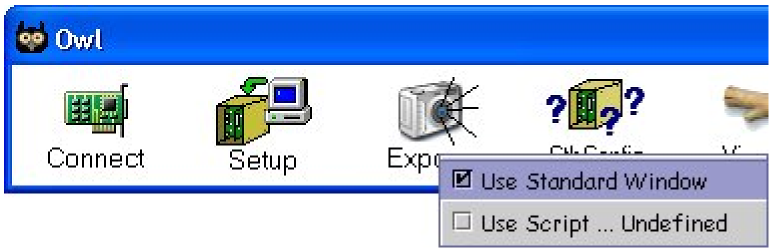
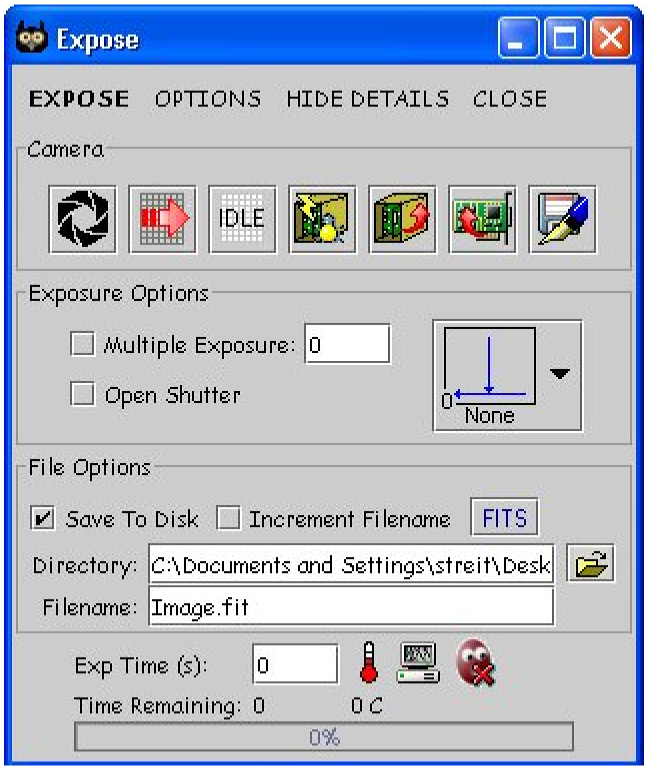


Figure 5. Selection of standard exposure window or script for the expose button.

The standard expose window contains three "hide able" option panels that can be shown or hidden using the "Options" menu button on the window. The sections are "Camera", "Exposure Options", and "File Options". The **"Camera"** panel contains buttons for the following actions (from left to right): *open/close shutter, clear array, idle on/off, controller power on/off, reset controller, reset PCI board, write current contents of image buffer to disk.* Note: Because the controller does not report its current status, some buttons may not match the current state of the controller. The **"Exposure Options"** panel contains options that are generally self explanatory and includes the image de-interlace option. The difference between the "Camera" panel open/close shutter button and the "Exposure Options" panel open shutter option is that the latter will open and close the shutter for the specified exposure time when the expose button is pressed. Whereas the former button will open the shutter immediately and hold it open until the button is pressed again. The **"File Options"** panel provides details for saving images in FITS format. The FITS button on this panel allows a custom header using XML. Please see the FITS and XML section for details.

A fourth panel on the standard expose window provides input for the exposure time (in seconds), a pixel readout progress bar, and a temperature button that displays the current controller temperature, an image display button for viewing the last image in a simple frame, and a BeanShell script toggle button for automatically executing a script at the end of an exposure sequence. The BeanShell script can be edited and/or set by right-clicking on the toggle button itself. See figure 6b.



Directory: C:\Documents and Settings\streit\Desk

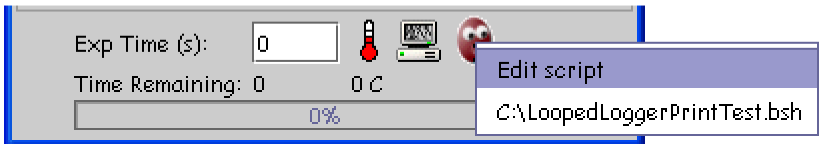
Filename: Image.fit

Multiple Exposure: 0

Exp Time (s): 0

Time Remaining: 0   0 C

Figure 6. The standard expose window.



C:\LoopedLoggerPrintTest.bsh

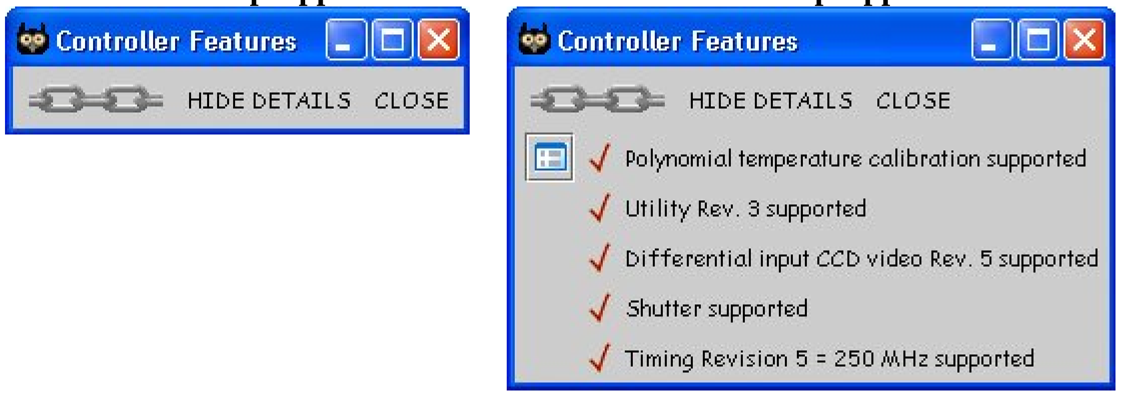
Figure 6b. Editing or selecting an exposure exit script.

Controller Configuration

This button opens a window that displays all available controller options. Note that controller options are only available after a controller setup has been applied. A checkmark appears next to all available options. A button will also appear to the left of a checkmark if the option provides user selectable values. The button will run a script, which is typically a window that will show user editable values or the script could directly perform a specific task. The association of scripts with controller parameters is fully customizable through the *CCScriptList.ini* file in the Owl installation directory.

Please refer to the *CCScriptList.ini* file for details on customizing the scripts. Finally, the menu button will allow you to view (only) the links between scripts and configuration parameters. It provides a sort of graphical version of what's in the *CCScriptList.ini* file. See figure 7.

Before Setup Applied **After Setup Applied**



✓ Polynomial temperature calibration supported
✓ Utility Rev. 3 supported
✓ Differential input CCD video Rev. 5 supported
✓ Shutter supported
✓ Timing Revision 5 = 250 MHz supported

Figure 7. Left: The controller configuration window before a setup has been applied. Right: After a setup has been applied.

<u>View Log</u>

This button opens the log window. All output from the program goes to this window. See figure 8. *Right-clicking* on the text area of the window provides a popup menu with the following options:

Log API commands – Results in every command being sent to the controller or PCI board to be shown in detail.

For example: DEBUG - 0x103 TDL 0x112233 -> 0x112233

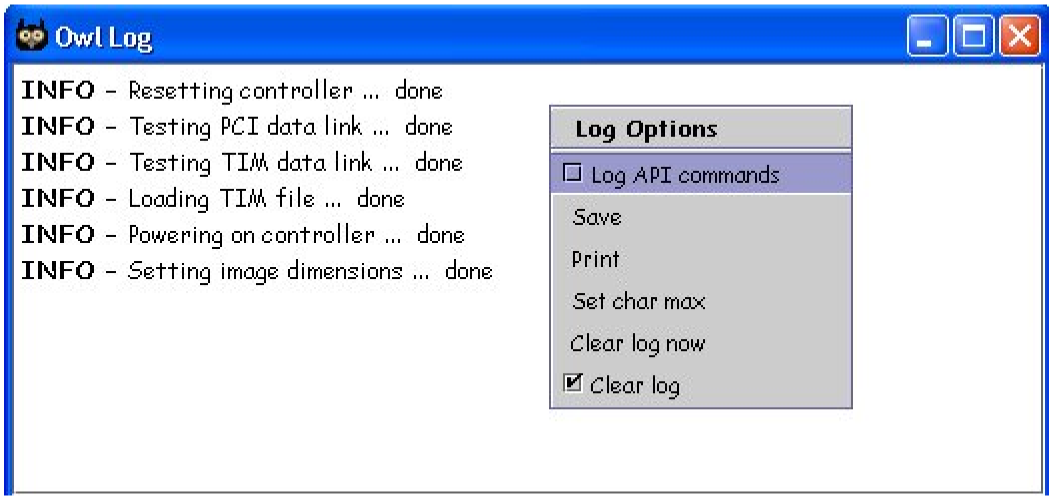
Save – Allows the current contents of the log window to be saved as a text file.

Print – Allows the current contents of the log window to be printed.

Set char max – Allows the user to set the maximum number of characters displayed in the window before being cleared. The default is 2048 characters.

Clear log now – Instantly clears the current contents of the log.

Clear log – If checked, causes the log to be cleared automatically every "Set char max" characters. The log does not clear if unchecked.



```
INFO - Resetting controller ... done
INFO - Testing PCI data link ... done
INFO - Testing TIM data link ... done
INFO - Loading TIM file ... done
INFO - Powering on controller ... done
INFO - Setting image dimensions ... done
```

Figure 8. The Owl log window with options shown by *right-clicking* in the window.

User

This User button opens a customizable toolbar that can be used to provide a set of buttons for running scripts. Buttons can be added or deleted by clicking on the add/del button. The user will be prompted to add a text or icon button. Finally, the user will be prompted to attach a script to the button. The script will then execute whenever the new button is pushed. Once created, the script can be changed to a different one by *right-clicking* on the desired button. See figures 9 and 10.

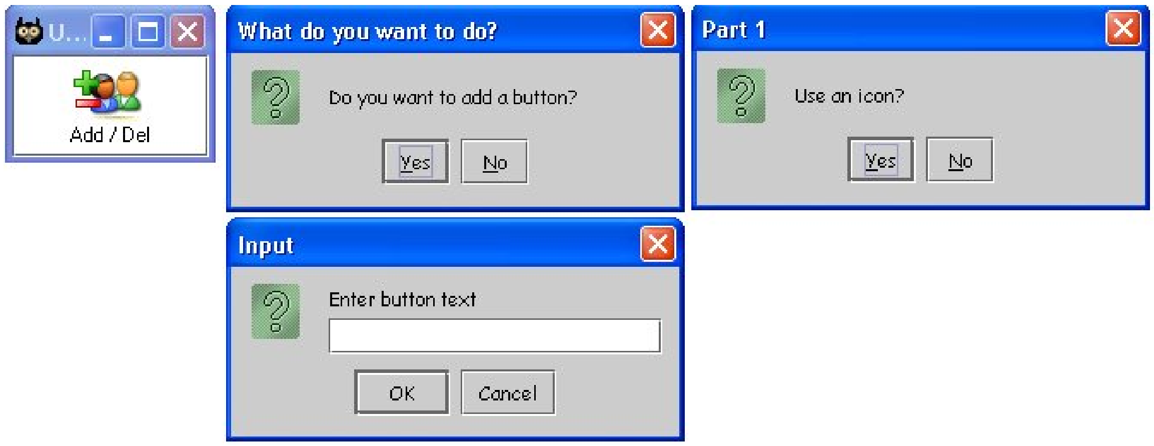


Figure 9. The add/del button and sub-windows for the “user” toolbar.

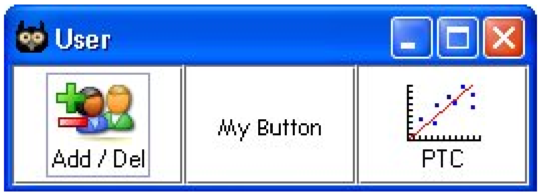
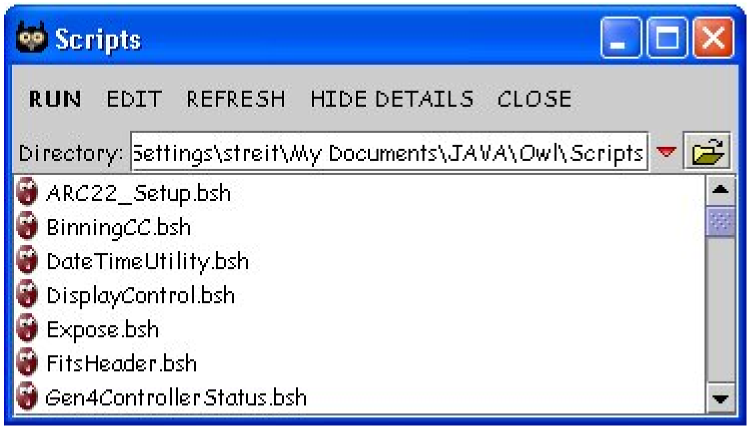


Figure 10. A customized “user” toolbar showing both text and icon buttons.

Scripts

This button opens the scripts window that allows the user to select, edit, and run custom scripts. Please refer to the scripts section for details on writing scripts. See figure 11. The *edit* menu option will open the selected script in WordPad on Windows, gedit on Linux, and dtpad on Solaris. The *run* menu option causes the selected script to be executed within a thread. A Boolean variable, called *stop*, exists in the Bean Shell interpreter and should be used within a script to determine if the user pressed the abort button, so the appropriate action can be taken. The small, red, down-arrow provides a drop-down list containing the last four script directories accessed. Selecting any directory from the drop-down list will cause the window to switch and display script files from that directory.



Settings\streit\My Documents\JAVA\Owl\Scripts

ARC22_Setup.bsh
BinningCC.bsh
DateTimeUtility.bsh
DisplayControl.bsh
Expose.bsh
FitsHeader.bsh
Gen4ControllerStatus.bsh

Figure 11. The Owl scripts window.

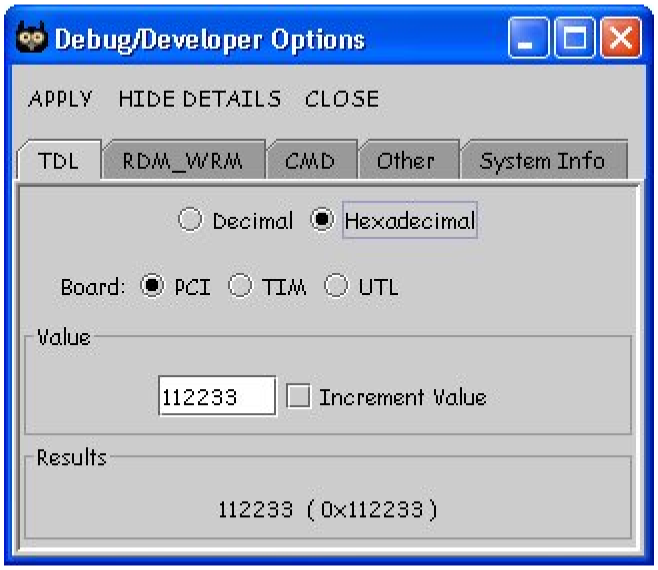
Debug

The debug button opens a window with options for debugging a camera system, saving and restoring program preferences, and provides program information (such as the info related to the Java Runtime Environment being used). The debug window has several tabbed panes that will be discussed individually.

TDL – This tab is used to verify communication between the host computer and the PCI board, controller timing board, or controller utility board. The *decimal/hexadecimal* option allows the input and output to be in the selected radix. Selecting this option will automatically change all values to the newly selected radix. The *increment value* checkbox increments the value after each click of the apply button. See figure 12.

RDM_WRM – This tab is used to read and write memory locations on the PCI, timing, or utility board. The *decimal/hexadecimal* option allows the input and output to be in the selected radix. Selecting this option will automatically change all values to the newly selected radix. The address section has its own radix option that allows the addresses to be input in a separate radix. Multiple or single addresses may be selected. The *memory* section allows R (ROM), P (program), X, and Y DSP memory spaces to be accessed. See figure 13. The resulting output is shown in the following form:

000001 >>> FCE5FA (address >>> value)



Decimal [✓ Hexadecimal]

Board: [✓ PCI] TIM UTL

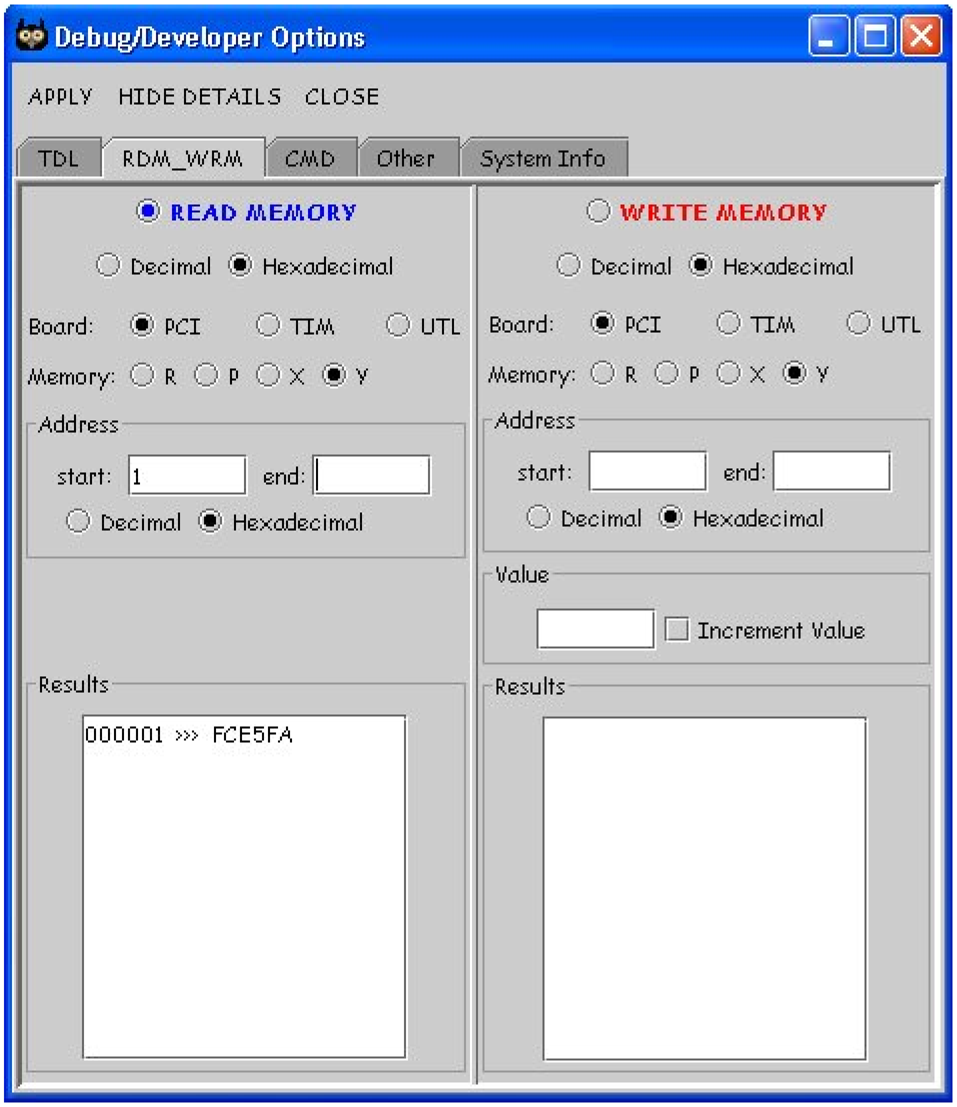
Value

112233 Increment Value

Results

112233 (0x112233)

Figure 12. The debug TDL option tab.



[✓ READ MEMORY]

Decimal [✓ Hexadecimal]

Board: [✓ PCI] TIM UTL

Memory: R P X [✓ Y]

Address

start: 1 end:

Decimal [✓ Hexadecimal]

Results

```
000001 >>> FCE5FA
```

WRITE MEMORY

Decimal [✓ Hexadecimal]

Board: [✓ PCI] TIM UTL

Memory: R P X [✓ Y]

Address

start: end:

Decimal [✓ Hexadecimal]

Value

Increment Value

Results

Figure 13. Read and write memory tab.

CMD – This command tab allows the user to send any allowable ASCII command to the PCI board, controller timing board, or controller utility board. The command argument may be a three letter string (upper or lower case) or the ASCII numerical value for the letters in the command. For example, 'TDL', 0x54444C, and 846876 are all valid (minus the '' and 0x). The *decimal/hexadecimal* option allows the input and output to be in the selected radix. Selecting this option will automatically change all values to the newly selected radix. Clicking the down arrow button will pop up a menu from which the user may select a command. The selected command will fill in the appropriate fields as specified in the command initialization file (OwlCmdList.ini). The menu will also contain the last four commands entered (along with their arguments). The commands available in the list are customizable through the *OwlCmdList.ini* file, which can be found in the Owl installation directory. See "Customizing the Debug Command List" section for further details. See figure 14.

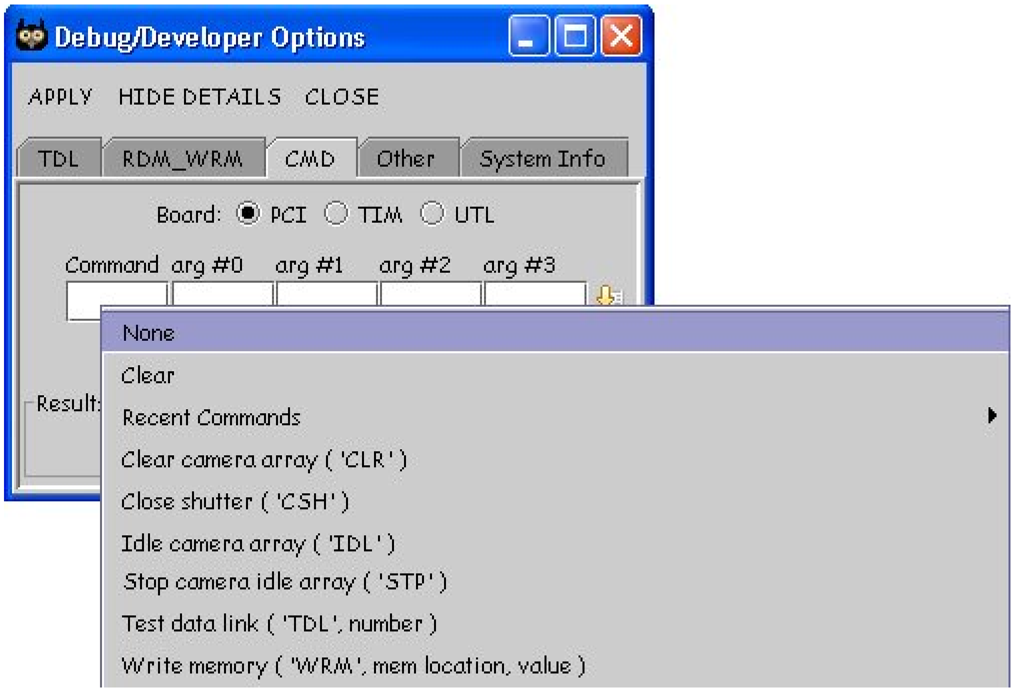


Figure 14. The general command window.

Other – This tab contains miscellaneous debug and program options. The *Export and Restore* preference buttons allow the user to create a backup of the program preferences that are stored in hidden system dependent location. It's possible that a crash of the program could cause the preferences to be lost, so creating a backup when you have things setup the way you like is advisable. Preferences include window size and location, file names and locations, and script attachments to buttons. See figure 15. The Create Synthetic Image button causes the controller to bypass the A/D converters and create a synthetic image. See figure 16 for an example of the synthetic image.

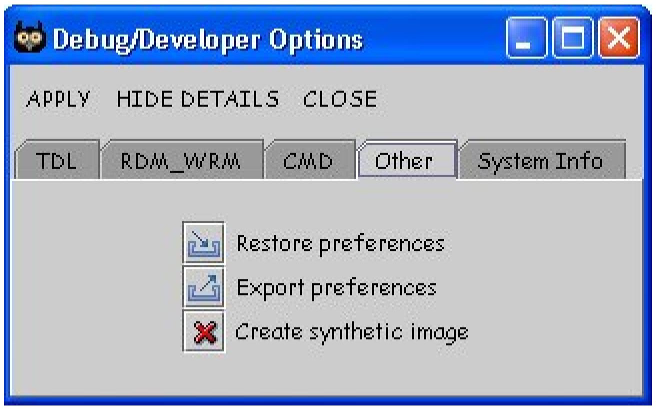


Figure 15. The miscellaneous debug and program option window.

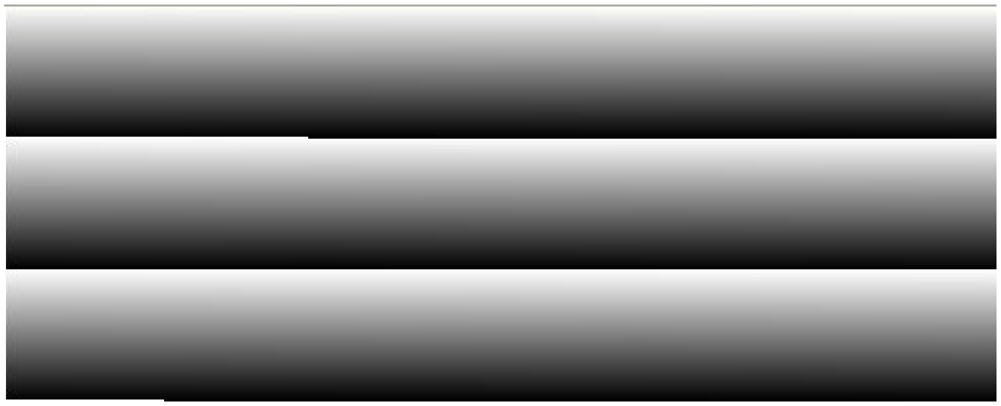
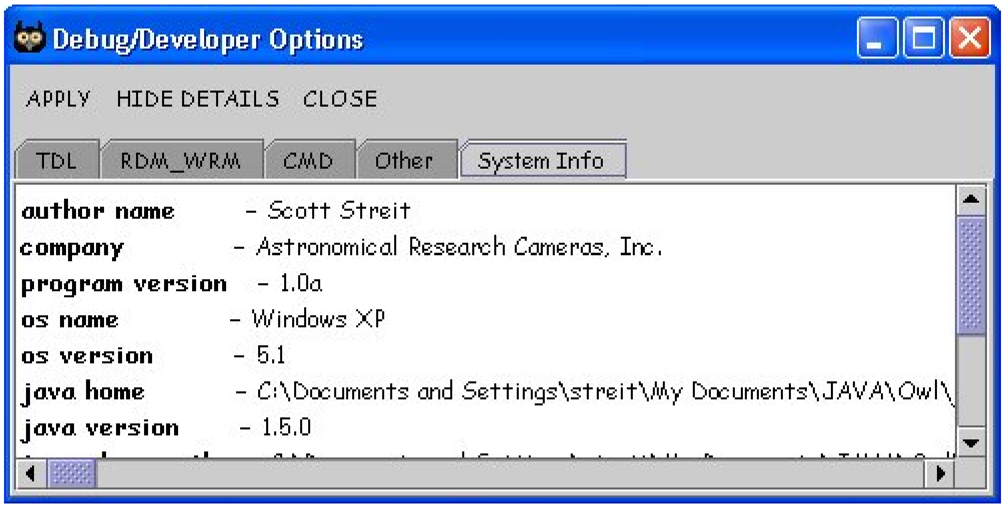


Figure 16. The synthetic image mode produces an image similar to this, but of the correct image dimensions.

Other – This tab contains info on the application and the Java Runtime Environment. See figure 17.



```
author name      - Scott Streit
company          - Astronomical Research Cameras, Inc.
program version  - 1.0a
os name          - Windows XP
os version       - 5.1
java home        - C:\Documents and Settings\streit\My Documents\JAVA\Owl\
java version     - 1.5.0
```

Figure 17. The system info tab in the debug window.

III. Basic Window Properties

Every standard window in Owl has contains two basic menu buttons. The *Hide/Show Details* menu button allows the entire window except for the menu and frame to be hidden. This helps to preserve space on the desktop and allows windows to be tiled. If the window contains an action button, such as “Apply”, the action button can still be used regardless of whether or not the window details are in view. The second basic menu button is *Close*, which I hope is self explanatory. Note: Closing a window does not cause the contents of the window to be lost, unless the window is generated by a script. Also, if a window contains an action button, such as “Apply”, it is always the leftmost button on the window menu. See figure 18.

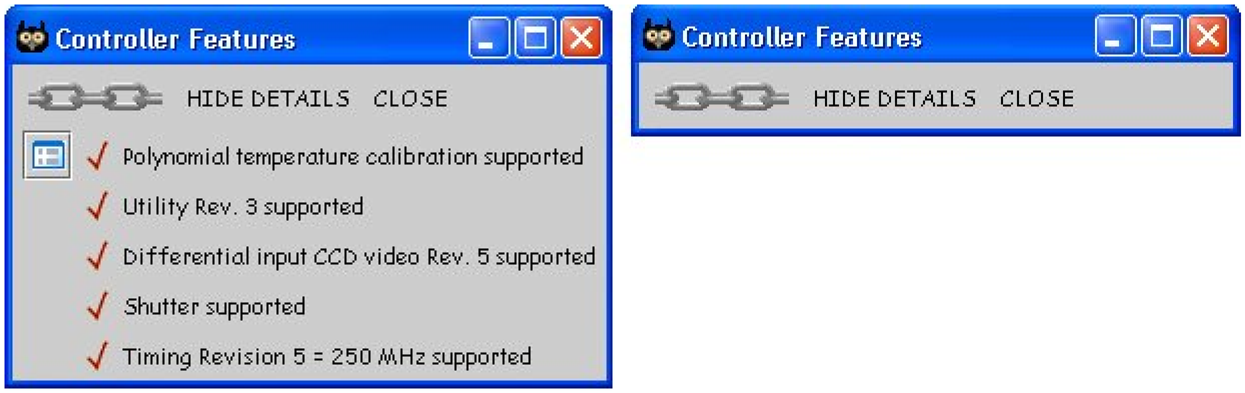

Controller Features
HIDE DETAILS
CLOSE
Polynomial temperature calibration supported
Utility Rev. 3 supported
Differential input CCD video Rev. 5 supported
Shutter supported
Timing Revision 5 = 250 MHz supported
Controller Features
HIDE DETAILS
CLOSE

Figure 18. Shows the hiding/showing of window details.

Window Preferences

Every window (well, almost every window) has a set of preferences that are saved on exit and re-used the next time the program is instantiated. Default preferences include the size and location of the window and which script is attached to which button. The preferences are an essential part of the program and must be cared for. Under some circumstances, however, if Owl should crash the preferences may be lost. To help prevent a complete loss of preferences, it’s advisable to backup your preferences once you have the program set up as you like. The preferences can be backed-up using the *Export Preferences* button on the *Other* tab in the *Debug* window. This way, if the preferences are lost, they can be restored using the *Import Preferences* button in the *Debug* window. The storage location of the preferences is platform dependent; the *registry* on Windows, in */home/username/.java/.userPrefs/owl* on Solaris, and I don’t remember on Linux but it’s probably something similar to Solaris.

IV. Application Initialization File

Owl contains a startup initialization file called *startup.ini* that can be found in the Owl installation directory. The filename and location is the same across all platforms. The initialization file is self-documented and will not be repeated here. It will be noted, however, that the program startup audio can be disabled by setting the [Audio] keyword to false.

V. Customizing the Debug Command List

The command tab within the debug window allows a user to select a command by right-clicking in the command parameter text field and choosing a command from the popup menu. The commands available on the menu are specified in the file OwlCmdList.ini, which is located in the installation directory on all platforms. The file contains the single keyword [CommandList], which has values of the following form:

<text description>< [><comma separated default text values for arguments><] >

Where the default arguments in brackets [] are optional. The default arguments will be put in order into the argument text fields of the command window. For an example see figure 19. Also refer to the file itself for details.

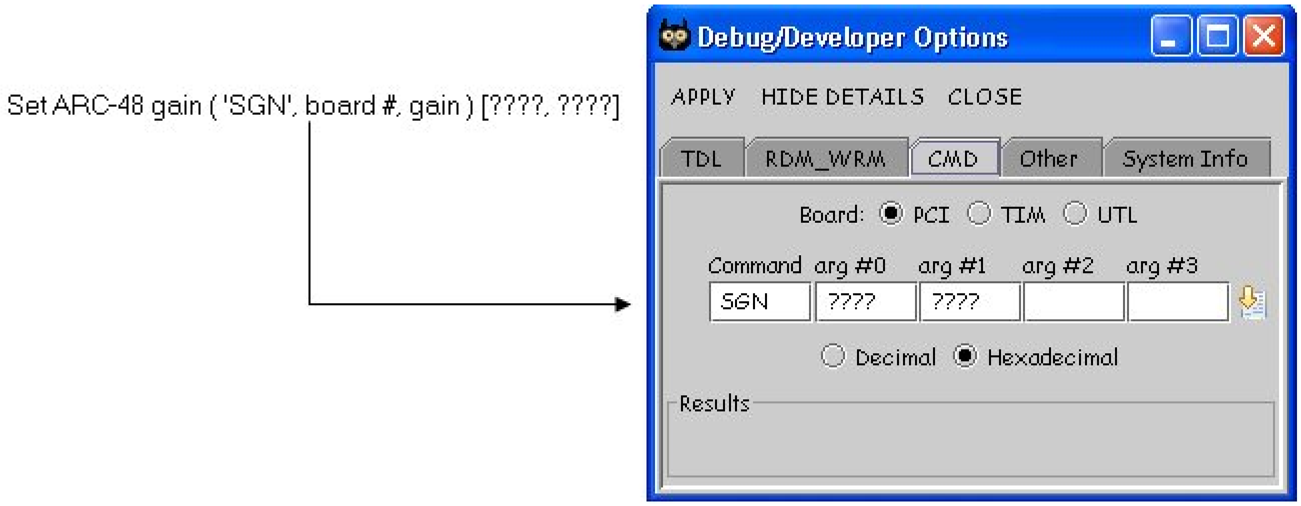


Figure 19. Shows how a command value in the OwlCmdList.ini file affects the parameters in the command tab of the debug window.

VI. FITS and XML

Owl allows customization of the FITS header through the use of an XML initialization file and a user defined script file. To start, open the Expose window and click on the “FITS” button in the “File Options” section. Initially, the Fits window that opens will be empty. See figure 20. A header XML initialization file can now be loaded into the window by clicking on the “Load” menu option. Once a file is loaded it will be reloaded on subsequent executions of Owl. However, before any of this can be done the user must first create an XML initialization file.

Figure 20. Initial (empty) Fits window.

The XML initialization file is fairly straight forward to create. The file is a simple text file that consists of a set of HTML like tags and values (strings, integers, decimals, etc). The first tag that the file should contain is the following: `<?xml version="1.0"?>`. This merely identifies the file as containing XML. The file should then contain one, and only one, `<header></header>` tag pair. In between the `<header>` and `</header>` tags there can be any number of `<card></card>` tag pairs. Each `<card></card>` tag pair must contain the following tags: `<type></type>`, `<name></name>`,

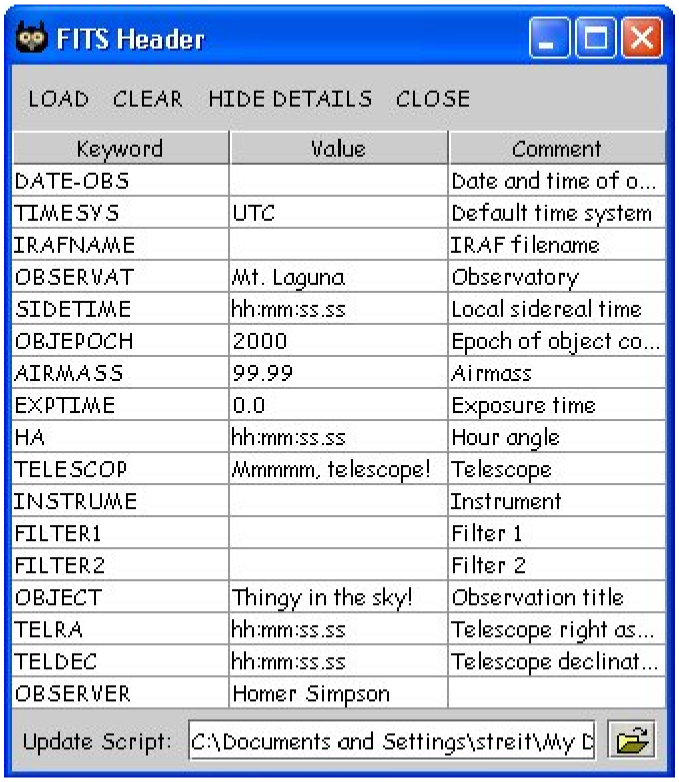
`<value></value> (optional)`, and `<comment></comment> (optional)`. Actually, only the *type* and *name* tags are required. The values contained within these tags represent valid FITS keywords. See http://heasarc.gsfc.nasa.gov/docs/fcg/standard_dict.html for a list of valid FITS keywords. An invalid keyword will likely be rejected by the FITS library. The value contained between the *type* tags can be one of the following: *string*, *integer*, or *real*. Example: `<type>string</type>`. The value contained between the *name* tags must be a string that represents a valid FITS keyword name. Example: `<name>SIDETIME</name>`. The *value* tags are optional and contain a default value for the specified FITS keyword. Example: `<value>hh:mm:ss.ss</value>`. Additionally, the *comment* tags are optional and can be used to provide a description of the keyword. The comment will appear in the header, but it must be less than 70 characters long. Example: `<comment>Local sidereal time</comment>`. See figure 21 for a more complete example.

```
<?xml version="1.0"?>

<header>

      <card>
            <type>string</type>
            <name>DATE-OBS</name>
            <comment>Date and time of observation</comment>
      </card>

      <card>
            <type>string</type>
            <name>OBSERVAT</name>
            <value>Mt. Laguna</value>
            <comment>Observatory</comment>
      </card>

      <card>
            <type>string</type>
            <name>SIDETIME</name>
            <value>hh:mm:ss.ss</value>
            <comment>Local sidereal time</comment>
      </card>

</header>
```

Figure 21. The contents of a FITS initialization file (SomeFile.xml).

Once a FITS initialization file has been created, it can be loaded into the Fits window. The window will then contain a table representing the data of the XML file. The window will look similar to that in figure 22. The keyword names within the table are not editable. The value and comment fields, however, can be edited by clicking on the field then typing.

As previously mentioned, a BeanShell script can be written and used to auto-update any of the fields within the FITS window. Clicking on the folder button next to the “Update Script” text field will allow the user to select a script that will perform the updating. This script, like all other BeanShell scripts, can do anything that the standard Java API can do; such as performing calculations or creating a TCP/IP connection and reading data from another computer. The script itself is called from Owl’s exposure sequence immediately before any FITS file is created.



Keyword	Value	Comment
DATE-OBS		Date and time of o...
TIMESYS	UTC	Default time system
IRAFNAME		IRAF filename
OBSERVAT	Mt. Laguna	Observatory
SIDETIME	hh:mm:ss.ss	Local sidereal time
OBJEPOCH	2000	Epoch of object co...
AIRMASS	99.99	Airmass
EXPTIME	0.0	Exposure time
HA	hh:mm:ss.ss	Hour angle
TELESCOP	Mmmmm, telescope!	Telescope
INSTRUME		Instrument
FILTER1		Filter 1
FILTER2		Filter 2
OBJECT	Thingy in the sky!	Observation title
TELRA	hh:mm:ss.ss	Telescope right as...
TELDEC	hh:mm:ss.ss	Telescope declinat...
OBSERVER	Homer Simpson	

Update Script: C:\Documents and Settings\streit\My D

Figure 22. The Fits window containing a header table after loading an XML initialization file.

V. Scripts

Be afraid … be very afraid! Just kidding. You should only be moderately afraid. The scripts used by Owl are written in "loosely" typed Java and interpreted using the BeanShell interpreter. Details on the BeanShell interpreter can be found at http://www.beanshell.org/ and general Java tutorials can be found at http://www.java.sun.com. This section will not discuss how to program in Java. Here are several links for basic Java tutorials:

1. http://java.sun.com/docs/books/tutorial/java/index.html
2. http://java.sun.com/docs/books/tutorial/essential/index.html
3. http://java.sun.com/docs/books/tutorial/collections/index.html

Note: The BeanShell interpreter conforms to Java SE1.4, which does not support generics in the collections classes (such as ArrayList, etc). The Owl installation directory contains a sub-directory called "Scripts", which contains a set of scripts used by Owl that can be used for reference. These scripts can also be called by other (user-defined) scripts. It is urged that you do not store your own scripts in this directory, as any future upgrades of the program may result in loss of your scripts.

I will, however, define the basic differences between "strictly" typed and "loosely" typed Java. In strictly-typed Java, all variables must have their data types defined, whereas loosely-typed Java does not require this.

For example:	*Strictly-Typed Java:*	`int someVariable = 0;`
	Loosely-Typed Java:	`someVariable = 0;`

Both define an integer variable called `someVariable` that is set to zero.

Classes, which are a collection of related variables and methods (functions) that can modify the variables, have the following basic form in strictly-typed Java:

Strictly-Typed Java:

```
public class MyClass
{
     int someVariable;
     public MyClass() { someVariable = 0; } // Constructor
     public SetSomeVariable( int val )
     {
          someVariable = val;
     }
}
```

In loosely-typed Java the class body itself is the constructor. So any code not contained within a method (function) belongs to the constructor. In other words, any code not contained within a method is executed when the `MyClass` class is instantiated. For example, the code `x = MyClass();` will create an instance of `MyClass` and assign it to `x`. In the process, `someVariable` will be set to zero (as it is not inside a method). Also, it is important to note that `someVariable` has global scope within the class. Finally, it is important to note that the line "`return this;`" is required to be at the end of all scripted classes. This statement is required by the BeanShell Java interpreter.

Loosely-Typed Java:

```
MyClass()
{
     someVariable = 0;

     SetSomeVariable( val )
     {
          someVariable = val;
     }

     return this;
}
```

Classes are useful, but not required for writing controller scripts. Any Java (SE1.4) class, including swing GUI classes, can be used within the scripts. This means a full graphical user interface can be scripted and multi-threaded.

Ptplot Plotting Package

Owl contains the Ptolemy Java plotting package. This package can be used to create various x-y, histogram, and line plots from within the java Bean Shell scripts. Details on this package can be found at: http://ptolemy.eecs.berkeley.edu/java/ptplot5.6-devel/ptolemy/plot/doc/index.htm

Basic plotting with the package is fairly straight forward to do. In fact, an example is probably the best way to describe the basic use of this package.

Example: The following example creates a Plot class, adds a title, x/y labels, and some data points that will be connected with a line. The result is shown in figure 23.

```
import ptolemy.plot.*;
```

```
TestPlot()
{
      frame = new JFrame();

      // Create ptplot ''Plot'' object
      ptplot = new Plot();
      ptplot.setTitle( "My Plot" );
      ptplot.setXLabel( "X Title" );
      ptplot.setYLabel( "Y Title" );
      ptplot.setBackground( Color.WHITE );

      // Add some data points. The first parameter associates
      // the other parameters with data set "0". The next two
      // are the x and y values. True means to connect the dots
      // with a line.
      ptplot.addPoint( 0, 0, 0, true );
      ptplot.addPoint( 0, 1, 2, true );
      ptplot.addPoint( 0, 2, 0, true );
      ptplot.addPoint( 0, 5, 4, true );
      ptplot.addPoint( 0, 7, 2, true );
      ptplot.addPoint( 0, 8, 7, true );

      // The Plot class is a sub-class of JPanel, so we can just
      // add it directly to a frame.
      frame.getContentPane().add( ptplot, BorderLayout.CENTER );
      frame.setSize( 400, 400 );
      frame.setVisible( true );

      return this;
}
```

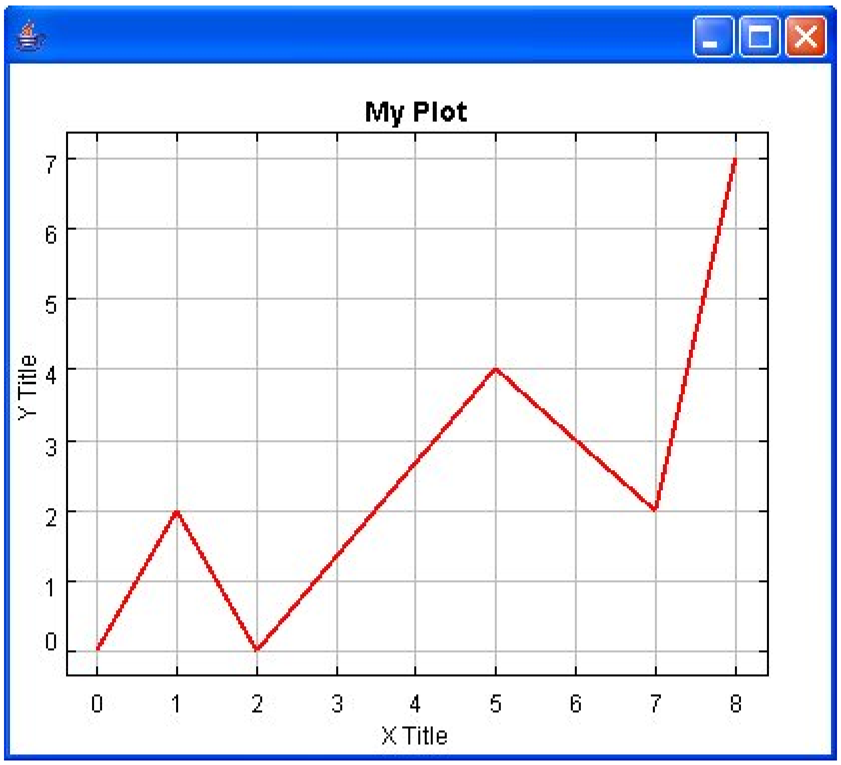


Figure 23. Example plot created using the Java plotting package Ptplot.

Hardware Communication APIs - Camera, Image, and ReplyException Packages

To communicate with a camera controller use the following imports (classes):

- *`import owl.cameraAPI.CameraAPI;`*
  The *`CameraAPI`* class provides all communication to the device driver, and thus, the PCI board and controller.

- *`import owl.cameraAPI.ImageAPI;`*
  The *`ImageAPI`* class provides methods to directly manipulate the contents of the device driver image buffer.

- *`import owl.cameraAPI.ReplyException;`*
  The `ReplyException` class provides methods to determine why a controller command failed.

`owl.cameraAPI.CameraAPI` Class Method List:

The *`CameraAPI`* class is used to access the device driver, and thus, the PCI board and controller (timing board, utility board). The class contains a fairly large list of methods and constants that can be used directly in scripts to access the hardware.

- *`public static int GetPCIStatus() throws Exception;`*

  The `GetPCIStatus()` method returns the value of the PCI board's status register (HSTR). Call `Exception getMessage()` method for error details.

- *`public static int GetPixelCount() throws Exception;`*

  The `GetPixelCount()` method returns the current number of pixels transferred during image readout. Call `Exception getMessage()` method for error details.

- *`public static boolean IsReadout() throws Exception;`*

  The `IsReadout()` method returns *true* of the controller is in image readout, *false* otherwise. Call `Exception getMessage()` method for error details.

- *`public static int PCICommand( int command ) throws Exception;`*

  The `PCICommand()` method takes a valid PCI command and returns any value associated with the command. Call `Exception getMessage()` method for error details.

- *`public static void PCICmd( int cmd, int expectedReply ) throws ReplyException, Exception;`*

  The `PCICmd()` method takes a valid PCI command and verifies that the reply equals the `expectedReply` parameter. The method throws a `ReplyException` if the actual reply doesn't equal the expected reply. Call `ReplyException` and `Exception getMessage()` method for error details.

- *`public static void Cmd( int brdId, int cmd, int expectedReply ) throws ReplyException, Exception;`*

- *`public static void Cmd( int brdId, int cmd, int arg, int expectedReply ) throws ReplyException, Exception;`*

- *`public static void Cmd( int brdId, int cmd, int arg1, int arg2, int expectedReply ) throws ReplyException, Exception;`*

- *`public static void Cmd( int brdId, int cmd, int arg1, int arg2, int arg3, int expectedReply ) throws ReplyException, Exception;`*

- *`public static void Cmd( int brdId, int cmd, int arg1, int arg2, int arg3, int arg4, int expectedReply ) throws ReplyException, Exception;`*

  The `Cmd()` methods take a valid controller command and verifies that the reply equals the `expectedReply` parameter. The method throws a `ReplyException` if the actual reply doesn't equal the expected reply. Call `ReplyException` and `Exception getMessage()` method for error details. These methods are valid for use with the Timing and Utility boards. `brdId` is one of the following: 2 for Timing board, and 3 for Utility board.

- *`public static int Cmd2( int brdId, int cmd ) throws Exception;`*

- *`public static int Cmd2( int brdId, int cmd, int arg ) throws Exception;`*

- *`public static int Cmd2( int brdId, int cmd, int arg1, int arg2, int arg3, int arg4 ) throws Exception;`*

  The `Cmd2()` methods are a short notation for the longer `Command()` method. See the description of the `Command()` method for details.

- *`public static int Command( int boardId, int command, int arg1, int arg2, int arg3, int arg4 ) throws Exception;`*

  The `CommandNative()` method takes a valid controller command for the specified board and returns any value associated with the command. Call `Exception getMessage()` method for error details. `boardId` is one of the following: 1 for PCI board, 2 for Timing board, and 3 for Utility board.

- *`public static void LoadPCIFile( String filename ) throws Exception;`*

  The `LoadPCIFile()` method takes a valid PCI "lod" filename and uploads the contents to the PCI board associated with the current device driver connection. Call `Exception getMessage()` method for error details. `boardId` is one of the following: 1 for PCI board, 2 for Timing board, and 3 for Utility board.

- *`public static void LoadControllerFile( String filename, int validate ) throws Exception;`*

  The `LoadControllerFile()` method takes a valid Timing or Utility board "lod" filename and uploads the contents to the proper board. Call `Exception getMessage()` method for error details. `validate` should be set to 1 if each word uploaded to the board should be read back and verified.

- *`public static void SetImageSize( int rows, int cols ) throws Exception;`*

  The `SetImageSize()` method sets the current image size to be used during an image exposure. Call `Exception getMessage()` method for error details. `Rows` and `cols` should be given in pixels.

- *`public static int GetImageRows() throws Exception;`*

  The `GetImageRows()` method returns the number of image rows (in pixels) set by a previous call using `SetImageSize()`. Call `Exception getMessage()` method for error details.

- *`public static int GetImageCols() throws Exception;`*

  The `GetImageCols()` method returns the number of image columns (in pixels) set by a previous call using `SetImageSize()`. Call `Exception getMessage()` method for error details.

- *`public static int GetCCParams() throws Exception;`*

  The `GetCCParams()` method returns the controller configuration word for the current controller. The bit definitions for the word vary from controller to controller and can generally be found in the DSP assembly file *timhdr.asm*. Call `Exception getMessage()` method for error details.

- *`public static boolean IsCCParamSupported( int parameter ) throws Exception;`*

  The `IsCCParamSupported()` method returns true if the specified controller configuration parameter is supported by the controller. The bit definitions for the word vary from controller to controller and can generally be found in the DSP assembly file *timhdr.asm*. Call `Exception getMessage()` method for error details.

- *`public static boolean IsControllerSetup();`*

  The `IsControllerSetup()` method returns *true* if the currently opened controller has been "setup". That is, if the controller is executing a valid program, has been powered on, and has had it's image dimensions set. The method returns *false* otherwise.

- *`public static double GetArrayTemperature() throws Exception;`*

  The `GetArrayTemperature()` method returns the current temperature of the camera array. This method is currently only valid for use with the Omega CY7 series temperature sensor. Call `Exception getMessage()` method for error details.

- *`public static void SetArrayTemperature( double tempVal ) throws Exception;`*

The `SetArrayTemperature()` method sets the current temperature around which the camera array should be regulated. This method is currently only valid for use with the Omega CY7 series temperature sensor. Call `Exception getMessage()` method for error details.

- *`public static void SetBinning( int rowFactor, int colFactor ) throws Exception;`*

  The `SetBinning()` method sets the binning factors for all future images. The binning parameters *rowFactor* and *colFactor* specify how many pixels will be combined together into a "larger" single pixel in the row and column direction respectively. Set both parameters to 1 to revert back to no binning. Call `Exception getMessage()` method for error details.

- *`public static void SetOpenShutter( boolean shouldOpen ) throws Exception;`*

  The `SetOpenShutter()` method determines whether or not any shutter attached to the camera will be activated during an image exposure. Set `shouldOpen` to *true* to open the shutter during an exposure, *false* otherwise. Call `Exception getMessage()` method for error details.

- *`public static void SetSyntheticImageMode( boolean mode ) throws Exception;`*

  The `SetSyntheticImageMode()` method determines whether or not the controller will return a synthetic image instead of one from an array and A/D converter. Set `mode` to *true* to create a synthetic image during an exposure, *false* otherwise. Call `Exception getMessage()` method for error details.

- `public static void DeinterlaceImage( int rows, int cols, int algorithm ) throws Exception;`

  The `DeinterlaceImage()` method will de-interlace the current data in the device driver image buffer using the specified algorithm. The *rows* and *cols* parameters are in pixels and specify the size of the image buffer to de-interlace. When an array is read out using multiple output stages the image is stored with a specific pattern that must be decoded to restore the original image. See figure 24 for an example. The `rows` and `cols` parameters may be less than the current image buffer size, however, the result is undefined if they exceed the current image buffer size. Call `Exception getMessage()` method for error details.

- *`public static void WriteFitsFile( String filename, int rows, int cols ) throws Exception;`*

  The `WriteFitsFile()` method will write the current contents of the device driver image buffer to a FITS file with the specified `filename`. The `rows` and `cols` parameters are in pixels and specify the size of the image buffer to write to disk. The `rows` and `cols` parameters may be less than the current image buffer size, however, the result is undefined if they exceed the current image buffer size. Call `Exception getMessage()` method for error details.

- *`public static void WriteFitsKeyword( String key, String keyVal, String comment, String filename ) throws Exception;`*

The `WriteFitsKeyword()` method will write the specified key to the given FITS filename. Call `Exception getMessage()` method for error details. `Key` is the name of the FITS key, `keyVal` is the string value associated with the keyword, `comment` is any string comment associated with the keyword, and `filename` is the associated FITS file.

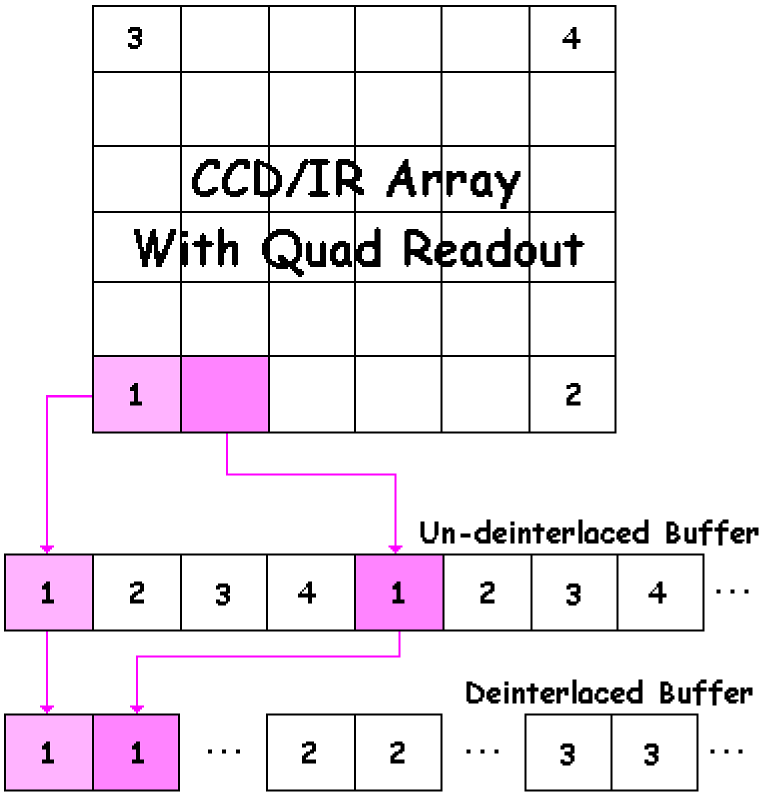


Figure 24. How a quad readout array is de-interlaced. The basic idea applies to other readout sequences as well.

- Board Id Constants

```
public static int PCI_ID;   // PCI board
public static int TIM_ID;   // Timing board
public static int UTIL_ID;  // Utility board
```

- DSP Memory Spaces

```
public static int X_MEM;    // X memory space
public static int Y_MEM;    // Y memory space
public static int P_MEM;    // Program memory space
public static int R_MEM;    // ROM memory space
```

- PCI/Controller Command Replies

```
public static int DON;      // Done; standard ok response
public static int ERR;      // Error
public static int SYR;      // System reset; returned by PON
public static int TOUT;     // Timeout; no response
public static int READOUT;  // Readout; camera is reading out
```

- Image De-interlace Indexes. These are passed to certain API methods.

```
public static int DEINTERLACE_NONE;
public static int DEINTERLACE_PARALLEL;
public static int DEINTERLACE_SERIAL;
public static int DEINTERLACE_CCD_QUAD;
public static int DEINTERLACE_IR_QUAD;
public static int DEINTERLACE_CDS_IR_QUAD;
public static int DEINTERLACE_CCD_4X_SERIAL;
```

- Amplifier Output Source Selection

```
public static int AMP_0;   //  Ascii __A amp 0; lower left
public static int AMP_1;   //  Ascii __B amp 1; lower right
public static int AMP_2;   //  Ascii __C amp 2; upper right
public static int AMP_3;   //  Ascii __D amp 3; upper left
public static int AMP_L;   //  Ascii __L left amp; lower left
public static int AMP_R;   //  Ascii __R left amp; lower right
public static int AMP_LR;  //  Ascii _LR lower right two amps
public static int AMP_ALL; //  Ascii ALL four amps (quad)
```

- Controller Commands. Please refer to the document (available on our website: www.astro-cam.com) *ControllerCommandDescription.pdf* for more commands and their details.

```
public static int AEX;     // Abort exposure
public static int CLR;     // Clear camera array
public static int CSH;     // Close shutter (immediately)
public static int IDL;     // Idle array (clock without transmitting)
public static int OSH;     // Open shutter (immediately)
public static int POF;     // Power off the controller
public static int PON;     // Power on the controller
public static int RDM;     // Read DSP memory location
public static int RET;     // Read elapsed exposure time (ms)
public static int SET;     // Set exposure time (ms)
public static int SEX;     // Start exposure or it's what you want
public static int SOS;     // Set output source
public static int SSP;     // Set sub-array position
public static int SSS;     // Set sub-array size
public static int STP;     // Stop idle
public static int TDL;     // Test data link
public static int WRM;     // Write DSP memory location
```

- PCI Board Commands. Please refer to the document (available on our website: www.astro-cam.com) Pci*CommandDescription.pdf* for more commands and their details.

```
public static int RESET_CONTROLLER;
public static int RESET_PCI;
```

`owl.cameraAPI.ImageAPI` Class Method List:

The *`ImageAPI`* class contains methods for manipulating image data, which may be in a FITS file or the device driver image buffer. Note, the device driver image buffer is not directly accessible through java because the mapping of large images becomes uselessly slow. However, it can be manipulated directly by any customer C++ applications using the API. If you wish to investigate this further for yourself, then please see the *java.nio* package in the current Java Software Development Kit (www.java.sun.com).

- *`public static double[] ImageStats( String filename, int rowStart, int rowEnd, int colStart, int colEnd ) throws Exception;`*

  The *`ImageStats`*`()` method returns an array of doubles that contains the image statistics for the specified FITS file. The row and column parameters can be used to specify a sub-region within which to calculate the statistics. The entire image can be used by setting all the row and column parameters to 0 or you can specify the full image dimensions directly. The double array contains the image minimum, maximum, mean, variance, and standard deviation. The individual array values can be indexed using the following static constants that are defined in ImageAPI: *`MIN_STAT`*, *`MAX_STAT`*, *`MEAN_STAT`*, *`VAR_STAT`*, *`STD_DEV_STAT`*. Call `Exception getMessage()` method for error details.

- *`public static double[] ImageDiffStats( String filename1, String filename2, int rowStart, int rowEnd, int colStart, int colEnd ) throws Exception;`*

  The *`ImageDiffStats`*`()` is used to create a PTC (Photon Transfer Curve). The method takes two FITS files and calculates the following difference statistics: minimum, maximum, variance, and standard deviation for the individual images and the difference mean, variance, and standard deviation for the two images. The individual values of the returned array can be indexed using the following static constants that are defined in ImageAPI: *`MIN1_STAT, MIN2_STAT, MAX1_STAT, MAX2_STAT, MEAN1_STAT, MEAN2_STAT, VAR1_STAT, VAR2_STAT, STD_DEV1_STAT, STD_DEV2_STAT, DIFF_MEAN_STAT, DIFF_VAR_STAT, DIFF_STD_DEV_STAT`*. The row and column parameters can be used to specify a sub-region within which to calculate the statistics. The entire image can be used by setting all the row and column parameters to 0 or you can specify the full image dimensions directly. Call `Exception getMessage()` method for error details.

- `public native static double GetPixel( String filename, int row, int col );`

- `public native static double GetPixel( int memPtrAsInt, int row, int col, int imgCols );`

  Both *`GetPixel`*`()` methods are used to get the value of a pixel at location [row, col]. The first method uses the specified FITS file for the pixel data; while the second method uses a pointer (as an integer) to an image buffer for the pixel data. The *imgCols* parameter is the total number of column pixels in the image. Note, the *GetImageBufferReference()* method can be used to get an integer reference to the internal device driver image buffer.

- `public native static double[] GetPixels( String filename, int rowStart, int rowEnd, int colStart, int colEnd );`

- `public native static double[] GetPixels( int memPtrAsInt, int rowStart, int rowEnd, int colStart, int colEnd, int imgCols );`

Both `GetPixels()` methods are used to get a sub-image of pixels using the *rowStart*, *rowEnd*, *colStart*, and *colEnd* parameters. The first method uses the specified FITS file for the pixel data; while the second method uses a pointer (as an integer) to an image buffer for the pixel data. The *imgCols* parameter is the total number of column pixels in the image. Note, the *GetImageBufferReference()* method can be used to get an integer reference to the internal device driver image buffer.

- `public native static double[] GetRowPixels( String filename, int row );`

- `public native static double[] GetRowPixels( int memPtrAsInt, int row, int imgCols );`

Both `GetRowPixels()` methods are used to get a row of pixels using the *row index* parameter. The first method uses the specified FITS file for the pixel data; while the second method uses a pointer (as an integer) to an image buffer for the pixel data. The *imgCols* parameter is the total number of column pixels in the image. Note, the *GetImageBufferReference()* method can be used to get an integer reference to the internal device driver image buffer.

- `public native static double[] GetColPixels( String filename, int col );`

- `public native static double[] GetColPixels( int memPtrAsInt, int col, int imgRows, int imgCols );`

Both `GetColPixels()` methods are used to get a column of pixels using the *col index* parameter. The first method uses the specified FITS file for the pixel data; while the second method uses a pointer (as an integer) to an image buffer for the pixel data. The *imgRows* parameter is the total number of row pixels in the image. Similarly, the *imgCols* parameter is the total number of column pixels in the image. Note, the *GetImageBufferReference()* method can be used to get an integer reference to the internal device driver image buffer.

`owl.cameraAPI.ReplyException` Class Method List:

This class represents a reply error from the controller (timing board, utility board). The class contains a number of methods to determine what caused the error.

- *`public int getCommand();`*

The *`getCommand()`* method returns the integer value of the command that failed.

- *`public int getActualReply();`*

The *`getActualReply()`* method returns the integer value of the reply returned by the controller. This value will be different from the expected reply value.

- *`public String getActualHexString();`*

The `getActualHexString()` method returns a hexadecimal string representation of the reply returned by the controller. The represented value will be different from the expected reply value.

- `public int getExpectedReply();`

  The `getExpectedReply()` method returns the integer value of the expected reply. This is what the controller should have returned.

- `public String getExpectedHexString();`

  The `getExpectedHexString()` method returns a hexadecimal string representation of the expected reply, that is, the value the controller should have returned.

The following variables are pre-defined by Owl and can be used in scripts:

- **Interp**

  This is a reference to the Interpreter object and can be used to run a script from within a script. See the BeanShell API documentation (http://www.beanshell.org/javadoc/index.html) for details on the Interpreter class.

- **stop**

  Stop is a Boolean variable that is set to true when the user presses the *Abort* menu option on the Scripts window. The *Run* menu option switches to *Abort* when the script thread is started. The executing script should check this variable and take the appropriate action if it has been set to true.

- **bitmapPath**

  This is a String object that contains the full path to the bitmap directory in the Owl installation directory.

- **xmlPath**

  This is a String object that contains the full path to the XML directory in the Owl installation directory.

- **logger**

  This is a reference to an instance of the OwlLogger class, which sends text to the application log window. The class defines the following methods:

  `public void info( String message )`

  The `info()` method prints a basic message to the log window. The message will be preceded by "INFO –".

  `public void error( String message )`

  The `error()` method prints an error message to the log window using red text for the prefix. The message will be prefixed with "ERROR –".

  `public void warn( String message )`

The `warn()` method prints a warning message to the log window using yellow-orange text for the prefix. The message will be prefixed with "WARN –".

`public void debug( String message )`

The `debug()` method prints a debug message to the log window using blue text for the prefix. The message will be prefixed with "DEBUG –".

`public void infoStart( String message )`

The `infoStart()` method prints a basic message to the log window. The message will be preceded by "INFO –" and terminated with "**… waiting**". This method is used to "wrap" a command with a message and should be terminated with one of the following methods: `infoEnd()`, `infoCancel()`, or `infoFail()`.

For example:

```
1. logger.infoStart("Doing something");
2. CameraAPI.Cmd( 2, TDL, 12, 12 );
3. Logger.infoEnd();
```

Produces:

```
1. "Doing something … waiting."
2.
3. "Doing something … done."
```

`public void infoEnd()`

The `infoEnd()` method finalizes an `infoStart()` method by replacing the "**waiting**" text with "**done**". See infoStart() for example.

`public void infoCancel()`

The `infoCancel()` method finalizes an `infoStart()` method by replacing the "**waiting**" text with "**cancelled**". See infoStart() for example.

`public void infoFail()`

The `infoFail()` method finalizes an `infoStart()` method by replacing the "**waiting**" text with "**failed**". See infoStart() for example.