

# Microscopy Software Architecture

G Pierce, P Barber

August 2011

## Contents

1	Introduction.....	3
2	Building the Project .....	3
2.1	Prerequisites .....	3
2.2	Subversion (SVN) Code Repositories.....	3
2.3	Using CMAKE to Build a Visual Studio Project.....	4
3	Hardware Setup.....	6
3.1	The Config.ini File .....	7
3.2	Microscope Mode Configuration Files .....	7
4	Code Structure .....	7
4.1	Generic and Specific Module Architecture.....	8
4.2	Hardware module creation, initialization and the success dialog .....	10
4.3	UIModule .....	11
4.4	UILogger.....	11
4.5	Exception Handler.....	12
4.6	XYStage and Z Drive Separation.....	12
4.7	Modules.....	12
5	Coding Methods.....	14
5.1	Object Orientated C basics.....	14
5.2	Signals Between Modules .....	15
5.3	DeviceListConfigModule.....	17
6	High-level Modules .....	17
6.1	Microscope Modes .....	17
6.2	RegionScan.....	17
6.3	Time-lapse.....	18
6.4	Region of Interest .....	19
6.5	Real-time Overview .....	19

6.6	Stage Scan .....	19
6.7	Stage Plates .....	20
6.8	Single Photo Counting (SPC) operation .....	20
6.8.1	SPC User Interface.....	21
6.8.2	Scanner.....	22
6.9	Camera triggering via signals.....	22
7	External Libraries.....	24
7.1	FreeImageAlgorithms / FreeImageIcs.....	24
7.2	Python.....	24
7.3	External dll dependencies.....	25
7.4	BasicWin32Window for image display and debugging.....	25
8	References.....	26

## 1 Introduction

This document outlines some of the details of the software architecture of the high level code of the Gray Institute Open Microscopes. It starts with the prerequisite dependencies and how to compile the projects and moves on to describe the structure of the software and how modular components can be integrated to form a working system. Settings, configuration and initialisation files are also described followed by details of top level operations, such as the region scanning and time lapse functions.

## 2 Building the Project

### 2.1 Prerequisites

- Microsoft Visual Studio – Currently known to work on VS Express 2008 (aka v. 9)
- LabWindows/CVI 8.0.1 – CVI libraries are used as is the UIR editor.
- Python - <http://www.python.org/download/releases/2.5.4/>
- CMAKE - <http://www.cmake.org/cmake/resources/software.html>
- An SVN Client such as TortoiseSVN - <http://tortoisesvn.net/downloads>

### 2.2 Subversion (SVN) Code Repositories

There are five repositories of code and libraries required to build our microscopy applications. They are:

- ATD\_Microscopy ([https://subversion.assembla.com/svn/ATD\\_Microscopy/](https://subversion.assembla.com/svn/ATD_Microscopy/)) contains a folder for each of our microscopy systems. Each system has a CMake build file and the correct configuration file (config.ini.in) for each system. The documentation for all the systems is contained in the Microscope Systems\Documents folder. ATD\_Microscopy also contains a folder for Microscope Modules. These contain the code for the high level microscope functions, such region scan.
- ATD\_Applications ([https://subversion.assembla.com/svn/ATD\\_Applications/](https://subversion.assembla.com/svn/ATD_Applications/)) contains various utilities we have written. The only code used for the microscopes in this repository is IcsViewer which displays images acquired.
- ATD\_Hardware ([https://subversion.assembla.com/svn/ATD\\_Hardware/](https://subversion.assembla.com/svn/ATD_Hardware/)) contains all the code for objects that abstract and implement our hardware devices.
- ATD\_Libraries ([https://subversion.assembla.com/svn/ATD\\_Libraries/](https://subversion.assembla.com/svn/ATD_Libraries/)) various common libraries used by all of our code.
- ATD\_Dlls ([https://subversion.assembla.com/svn/ATD\\_Dlls/](https://subversion.assembla.com/svn/ATD_Dlls/)) contains pre compiled dll's and lib's that are needed to compile and run the code.

These repositories do not contain the code for all our individual modules the developer is expected to use CMake to generate a build file that references all the modules in ATD\_Hardware, ATD\_Applications and ATD\_Libraries.

ATD\_Hardware also has its own CMake file for building stand alone hardware modules that are not integrated into a larger program.

The ATD\_Microscopy top-level directory contains two directories "Microscope Modules" and "Microscope Systems". The "Microscope Modules" directory contains modules common to all our microscopes. This includes code to perform timelapse experiments and region scans. The "Microscope Systems" directory contains the folders for each of our systems. These are essentially directories containing only build files. There is a small amount of code in the MainApplication directory of each system that includes the main() method which creates and runs our microscope objects.

There is one module in ATD\_Microscopy called Dummy which is a dummy microscope that can be used for testing and development away from the microscope hardware. This is a system that is intended to fully test our microscope code where there is no hardware present. It should be able to be compiled on any computer and can test all of our higher level modules.

ATD\_Dlls should contain files for

- FreeImage (<http://freeimage.sourceforge.net/>)
- FreeImageAlgorithms - home grown image processing code built in FreeImage – (<https://code.launchpad.net/~glennpierce/+junk/main>, src in ATD\_Libraries)
- FreeImageICS – home grown code to load ics files into FreeImage structures (src in ATD\_Libraries)
- ImageViewer – Basic image display component of ICSViewer (src in ATD\_Libraries)
- StringUtils – home grown string utilities (src in ATD\_Libraries)
- Python and Python\_d – natural and debug compiles of python

And the path to ATD\_Dlls\trunk should be placed in the path environment variable.

### ***2.3 Using CMAKE to Build a Visual Studio Project***

Our microscopy applications are ANSI c (Not labwindows dependant) and are provided with CMake build files. CMake is a tool which generates make files or IDE project files for a variety of tools. In our case usually CMake is used to generate a visual studio project that then compiles these projects.

To build the dummy microscope application you would perform the following steps.

Checkout ATD\_Applications, ATD\_Hardware, ATD\_Libraries, ATD\_Microscopy into a directory. For us this is usually C:\Devel\ATD\

Check out ATD\_Dlls and add C:\Devel\ATD\ATD\_Dlls\trunk to the system path environment variable.

Run cmake-gui from the windows start menu (see Figure 1).

In the edit box that asks where the source code is, enter: C:\Devel\ATD\ATD\_Microscopy\Microscope Systems\Dummy\trunk (This is where the CMakeLists.txt file is).

For the edit box asking where to build the binaries enter: C:\Devel\ATD\ATD\_Microscopy\Microscope Systems\Dummy\trunk\build. (This could be anywhere but this is the convention).

You can then press configure until all the red info messages turn to grey (locate directories or files if some are not found, if the convention is followed they are probably all found automatically). You then press generate and your build files will have been generated.

To get rid of the red info messages you may need to specify the paths to the other three repositories.

For ATD\_Hardware this would be C:\Devel\ATD\ATD\_Hardware

For visual studio you can then go into

C:\Devel\ATD\ATD\_Microscopy\Microscope Systems\Dummy\trunk\build and open Dummy.sln.

Remember in visual studio to right-click the project and select “Set as start up project” so that dummy runs when the play button is clicked.

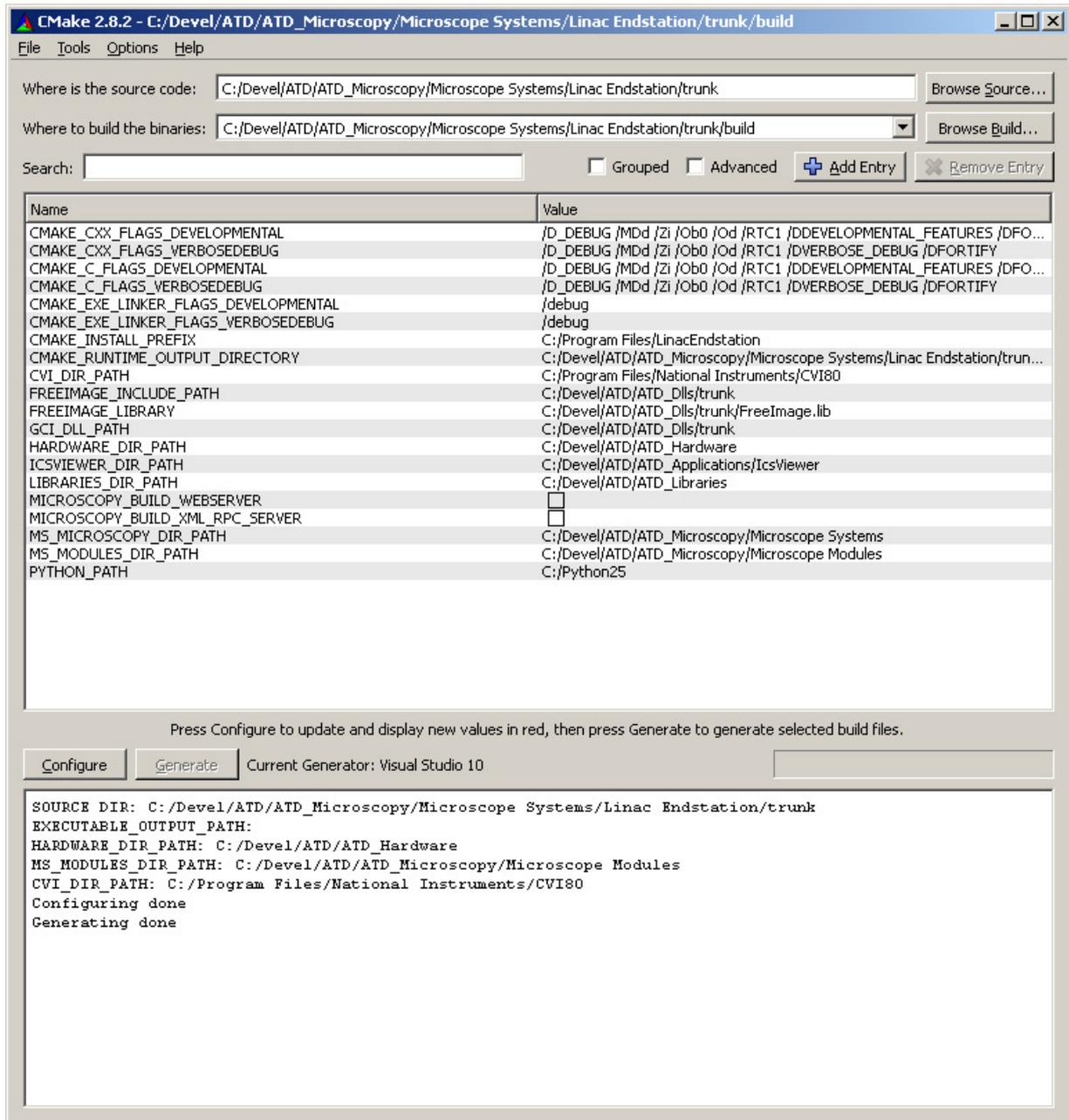


Figure 1, Screenshot of CMAKE gui correctly configured to build the project.

### 3 Hardware Setup

We have mainly used *ini* files for storing hardware setup information. This was mainly due to the simplicity but also due to a CVI incompatibility. We originally used xml but the CVI xml library seems to use msxml4. However, some new libraries that use xml such as the Imperx lynx camera library use msxml5 and you can not link msxml4 and msxml5 into the same application without problems.

Hardware configuration is held in 2 types of file. One file, config.ini, holds microscope wide

configuration necessary to communicate with the hardware and setup that should not change during use. The second type of file is also an ‘ini’ file, but there are many hardware specific ini files (and the occasional txt file) that hold settings for individual modules, the contents of which may change during use.

### ***3.1 The Config.ini File***

The config.ini file is created by CMAKE from the config.ini.in file as it contains computer specific paths. It is important to make any necessary changes to config.ini.in as well as config.ini (or just to config.ini.in and then rebuild the project).

The software communicates with the main hardware through either RS232 COM port commands or native FTDI calls (D2XX). Most microscopes use the native call method but this has led to problems on some hardware (maybe due to conflicts with other connected hardware) and so some use the Virtual COM Port (VCP) for RS232 commands.

Computers that use the native call method must have the serial number of the FTDI device in config.ini. The FTDI serial number can be obtained from one of the test programs that identifies the devices at startup and enters them into the drop down menu.

It can also be obtained from the Device Manager in the Windows Control Panel. In Device Manager, open up the Universal Serial Bus Controllers tree. The device will be listed as a USB Serial Converter. Right-click on this and select Properties. On the Details tab, select Device Instance Id from the drop down menu. You will see a string of characters similar to “USB\VID\_0403&PID\_6001\ELBSAT12”. The serial number is the last 8 characters of this string (after the ‘\’).

Use of the VCP requires that the VCP driver is loaded (see device manager for the USB Serial Converter) such that there is an entry in device manager under Ports (COM & LPT) for USB Serial Port (COM4), which indicates the COM port in use. Computers that use this method must have the COM port number entered in config.ini.

Most config.ini files have both bits of information added for native or VCP use. The test programs for individual hardware components all use the VCP.

### ***3.2 Microscope Mode Configuration Files***

As well as per module configuration will have configuration data for the modes of the microscope. For example the software stores data for the fluorescence, bright field and laser scanning modes. This data configuration overrides any per module configuration. There are menu items in the microscope code that allow the user to save all configuration data at that time or they can reset the data to default values. All this does is copy over the corresponding data file with the prefix “Default” to the current configuration.

## **4 Code Structure**

We can now describe the architecture of the microscope software. The Microscopy applications are made up of a number of diverse modules which can control all the various forms of hardware.

## 4.1 Generic and Specific Module Architecture

The majority of the microscopy application revolves around the microscope object. This object contains references to all the generic objects that make up a microscope. This includes objects like Camera, Stage and Cube Slider etc. As these are the generic objects the code in the microscope object should work with any implementation of a module. The Microscope object is itself intended to be inherited from. For example for “Abbe” we have an implementation AbbeMicroscope. This objects code creates all the specific module objects which represent specific hardware on the system (see Figure 2).

Each of these modules derives from a common base object called UIModule (see below) that provides common functionality to all modules. The modules talk to one another by the use of signals (see below) which is a small C library that mimics events and call-backs in other languages.

The entry point to the program is in Microscopy.c. This relatively small file is microscope specific and also handles the main user interface panel that has high level function and the ‘Exit’ button. The creation of module objects is always from the top down, but calling has to be done from the bottom up so the correct module for the specific objects are created, as follows:

- The generic `microscopy_app_new` in `Microscopy.c` calls `microscope_new` of the specific microscope.
- The specific `microscope_new` calls `microscope_constructor` for the generic microscope.
- The generic microscope is created by `microscope_constructor`.
- The specific microscope is then created by `microscope_new` which creates all the virtual methods required by the generic microscope (VTABLE).

The specific `microscope_new` then creates all the required hardware objects for that microscope, again using the same method.

- The specific `microscope_new` calls `specific_hardware_new` for the specific hardware item (e.g. `ATD_cubesliderA_new`, or `dcam_camera_new`).
- The `specific_hardware_new` calls the constructor for the generic hardware type, `hardware_constructor` or `hardware_new`.
- The generic hardware object is then created by `hardware_constructor`, including any generic user interface.
- The specific hardware object is then created by `specific_hardware_new`, including any specific user interface (usually accessed by a ‘more’ or ‘settings’ button and the creation of all virtual methods (VTABLE) required by the generic object.

Hardware initialisation then proceeds, the virtual specific method being called indirectly from `microscopy_app_new` before the user interface is allowed to run. Although some initialisation methods are threaded to allow simultaneous initialisation and some user interaction (enter user name etc.)

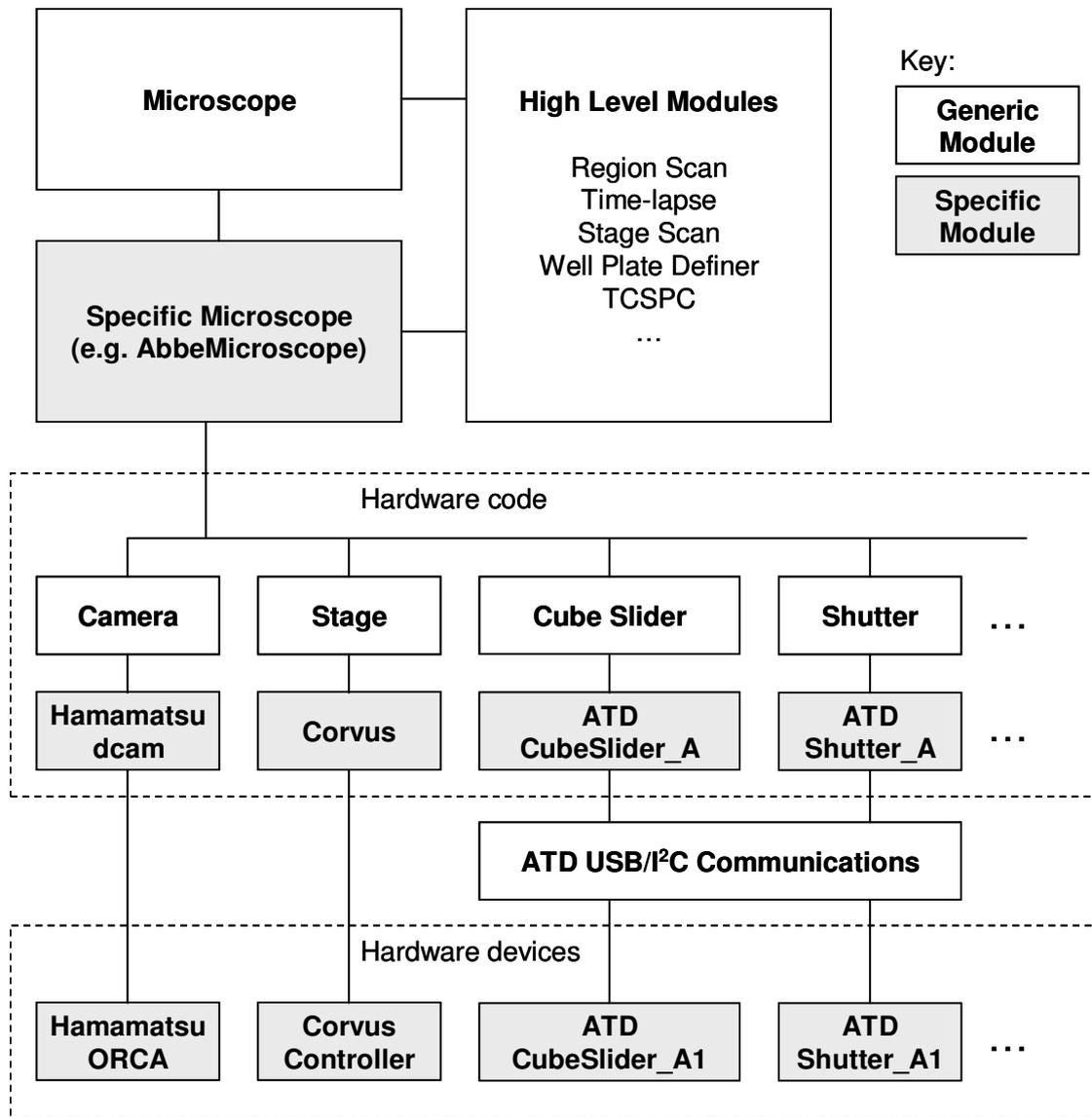


Figure 2, Microscope software architecture. The generic code modules remain the same for all microscopes (although different modules may be included or not on different microscopes) and the specific module code contains specific implementations for specific hardware devices. (ATD denotes Advanced Technology Development Group and relates to in-house built hardware described in other accompanying technical notes).

In each of the hardware module there are a few virtual methods always defined. They are

- Initialise
- Hardware\_initialise
- Destroy

The initialise method is used to setup any UI elements or reading configuration files. This could

also be done in the modules constructor.

The hardware initialise method is solely for initialising hardware. This has its own method as this function is intended to be thread safe. Often our hardware initialisation takes place in a separate thread so please make sure your implementation is thread safe.

The destroy method is there to provide any necessary cleanup like returning hardware to a safe shutdown state, closing files or freeing memory.

## **4.2 Hardware module creation, initialization and the success dialog**

Each hardware module has a ‘\_new’ function to setup the software object. The ‘new’ function is called first and sets up the table of virtual functions and signals for that module. These calls are made from the `microscope_new` function which is usually placed at the end of the specific microscope C file (e.g. `AbbeMicroscope.c`) and is a point of reference for all the modules required and created for the particular microscope. It ensures all code modules are created before any higher level modules can be created and any initialization takes place.

Early in this process the user login panel is displayed. The purpose of this panel is to encourage users to supply some information for image metadata (future auditing procedures and databases such as OME will require experimenter and study information) and so the system can setup a directory structure for them, in the Microscope Data directory, in an attempt to keep data in one place on the host pc. This aids integration with any central storage or archiving server.

Next a call to `microscope_initialise` is made. This is in the generic `Microscope.c` file. This calls in turn `microscope_initilise_devices`, `microscope_load_ui` and `microscope_post_initialise`. This displays the hardware initialization ‘success’ panel and calls the microscope specific `initialise_hardware` function which calls initialization functions for all the hardware (and some purely software) modules required for that microscope. Some of the initialization, such as for the camera, stage, cube slider and optical path switch occur in separate threads for speed as they can proceed in parallel.

Then higher level modules such as region scan and time lapse that depend on other modules can be created. This is followed by user interface display and the manual hardware setup panel. This panel is to encourage users to check the hardware components that are not under computer control, and as such the computer cannot know the state of these devices. This information is also required for image metadata. Examples of manual components are the objective lens, the PMT and the laser scanning filter set. The panel shows the current setting in software and a shortcut button to display the UI for the relevant module for them to change if required. The manual hardware setup panel does not update when settings are changed and is intended as a user prompt at start up and should be closed after use (no harm will come if it is left open, it just may be confusing as it does not update with settings changes – a whole new section of connected signals would be required to update this panel, it has been seen as not worth the coding effort so far).

Now we can start all timers that update the UI and set the microscope mode (Fluorescence or Brightfield etc.) and load the default settings for that mode. See the relevant section of this document for how the microscope settings for different modes are handled.

`microscope_post_initialise` is used finally if there are some modules whose settings depend

on another part of the microscope. For instance, on the 90i microscopes (e.g. Zernike) the illumination apertures must be set for the given objective. Settings for these additional modules are loaded and set here according to other modules.

### **4.3 *UIModule***

Each module is intended to derive from an object called `UIModule`. This object provides methods that display, hide and close panels while remembering the size, position and last visibility status. The properties of size and position are stored in the registry for each panel. Note that although these properties are stored in the registry for the microscope code the positions of panels are re-read from the file that defines all the properties for a particular microscope mode i.e. Fluorescence mode. So in this case the registry entries won't seem to take effect. Instead of using the CVI methods such as `LoadPanel` it is expected that the programmer uses the method `ui_module_add_panel` which allows all panels and their state to be kept track of. It also provides instance variable which keeps track of any signals that at module defines.

This module allows the developer to search for resources such as uir files by name. To do this the module makes use of a file called `resources.h` which is created via the following method. The CMAKE script creates a file called `resources.dat` by searching all source folders for files of type "uir", "ico", "bmp" and "txt". A VS project called `BuildResourceTable` then takes this file and creates `resources.h` which is used in the main project.

This file also lists the relative paths to the uir files used by the `UIModule` code. The reason for this is that `labwindows` sometimes had problems embedding the uir file into an executable, and indeed Visual Studio cannot embed the UIR files. When loading a uir from the `uimodule` code the search path is as follows. The directory containing the executable is first searched, if no uir file is found then all the directories in the system path are searched, if still no file is found the paths defined in the `resources` file are searched. This allows us to have a dll that uses uir files in the system path and this can be linked into our application without copying the uir file to multiple places. This also helps a lot when wrapping out code into another language.

### **4.4 *UILogger***

Each `UIModule` has a `UI Logger` module. This module allows any module to log errors or info to the user. The developer can specify whether the log message is critical or just information and the logger will deal with that in the appropriate way. The log function is also re-entrant and so can be used from different threads. As it would be a waste of resources for each module to instantiate a `UI Logger` object through the use of `UIModule` the `UI Logger` is reference counted. The first module in our microscope code that gets instantiated also creates a `UILogger` object and the reference count gets increment. Each subsequent module that uses the logger increments the reference count and reuses the same original logger. When the modules are destroyed the reference count is decremented. Only when the reference count goes to zero is the logger module destroyed. In addition to providing output to the user at runtime this module also writes the log to a text file. Each new run of the application results in a new log file with a unique identifier appended. If the log files are greater than three months old then they are deleted.

## **4.5 Exception Handler**

Since starting to use Visual Studio we have made use of the exception handler extension. By surrounding a section of critical code with ‘try’ and ‘catch’ blocks we can provide code to handle exceptional circumstances.

This is a Visual Studio enhancement to C. We may wish to use C++ try and catch handlers on our code in the future as they are more feature full and portable. In our exception handler which surrounds all our code we write a stack trace and any details of the computer to disk. We also ask the user to email this to the developers. Note this does not work at the moment if an exception occurs from another thread other than the main one. Similar to the UI logger the stack trace files are written to an incremented file.

## **4.6 XYStage and Z Drive Separation**

Originally we had an abstracted stage module that abstracted all stage functions including movement in the z direction (which some stages are able to perform). However, there tends to be more than one way to move in the z direction. For example, our microscopes have motor driven-coarse and fine piezo-electric focus drives. To cope with all these z devices we created an XYStage module, removing the z movement. We also created a generic Zdrive object that we implement for all our devices. Of course if a microscope has more than one way to move in the z direction we still need to know which device to move. For this scenario we have a “master zdrive” object in our microscope. This object is the one that is used to perform focus moves.

## **4.7 Modules**

The microscope code has many modules for various purposes, this semi-exhaustive list attempts to give brief details about some of them.

- IcsViewer - Stand alone image viewer and can be used in a larger application for display of images.
- ATD\_UsbInterface - Defines methods used by our i2c communication protocol.
- BatchCounter – Hardware module for batch counting fast events such as photon counting.
- Camera – Hardware module that abstracts all of our cameras.
- Cube Slider – Hardware module that abstracts and controls fluorescence cube changers.
- DeviceFinder – Deprecated (superseded but still included for backwards compatibility) module used by the old standalone module code. Don’t use in microscope code.
- HardwarePy – Module that wraps our microscope functions in a Python method.
- Lamp – Hardware module used for all our light sources.
- Microscope – This is a hardware module that provides a common interface to our individual microscopes systems and is also where individual microscope features are defined. If you were creating a new microscope you would start here.
- MotorMike – Hardware module to control z axis lift for Surrey system.

- Objectives – Hardware module to control automatic objective changers.
- OpticalLift – Hardware module to control optical Lift.
- OpticalPath – Hardware module to control OpticalPath / Mirror Stepper.
- Power Switch – Hardware module to abstract controllable power supplies.
- Scanner – Hardware module used to control laser scanners.
- Shutter – Hardware module to control shutters.
- SPC – Module which defines time resolved photon counting features.
- Stage – Hardware module used to abstract stage controllers.
- StepperMotor – A module to control a stepper motor. This module is not to be used directly. Currently it is used by the Optical Path module.
- TemperatureMonitor – Hardware module used to monitor thermostats.
- WhiteLight Laser – Hardware module that controls out white light laser.
- ZDrive – Hardware abstraction for our ZDrive hardware.
- CHARM – Image processing code for the segmentation of cell images.
- CVI stubs – Code needed when including the CVI toolslib objects in Visual Studio.
- DeviceListConfigModule – Common code that helps modules that save a list of devices. I.e. the cube manager and objectives.
- Dictionary – Hash table implementation in C.
- Excel Automation – Allows passing data to Excel.
- Exception Handler – Module that invokes our own exception dialog and saves to text file.
- FilePrefixSaveDialog – Module that asks the user for a file prefix when saving timelapse data and regionscan data.
- FreeImageAlgorithms – Image processing library written by us.
- FreeImageICS – Integrates ICS file support into FreeImage
- GCI Registry – Allows one to save and read values to the registry.
- GCI Utils – Various utility functions.
- ImageViewer – Win32 image control for showing an image and providing zoom and panning. This is the control used in IcsViewer.
- Password – Module that pops up a dialog for a password entry. Does not provide proper security but is suitable for stopping accidental configuration changes.
- Profile – Module for measuring time spent in code for debugging.
- Signals – A general call-back mechanism.
- Status – A generic feedback dialog for the user.

- String Utils – Various string utility functions.
- UIModule – A module that provides saving of UI properties and managing of multiple panels.

## 5 Coding Methods

### 5.1 Object Orientated C basics

The vast majority of our software was written using the National Instruments CVI programming environment and is therefore ANSI C (C89 standard) compliant and uses the CVI GUI toolkit. As we are working in a C based environment we are unable to use built in language features for creating object orientated code. Fortunately, objected orientation is simply a means of arranging the code and can be simulated in C with a bit of work. It is worth it because it allows the interchange of like modules into the same microscope framework (i.e. a manual cube changer, the ATD cube slider and a Nikon cube turret can all offer the same API and so be interchangeable).

To mimic objects in C we use structures and function pointers to implement object inheritance and polymorphism. In object orientation speak, inheritance is where one object can inherit from another. The object that inherits gains all the methods that the parent object had. It may also add any methods of its own or redefine any of the parent's methods. The latter is known as polymorphism.

For example let's say we were to implement a simple object relationship of a circle inheriting from a shape object. The shape object defines the method Area. We are unable to implement this Area method as only a specific shape can have an area calculated. The shape object would be known as an abstract class in object orientated languages. In C we would create this relationship as:

```
typedef struct _Shape Shape;
typedef struct
{
    void (*Area) (Shape* shape);
} ShapeVtbl;

struct _Shape
{
    ShapeVtbl vtable;
    // Any Shape instance variables can go here
};

// Circle Object
typedef struct
{
    Shape parent;
    // Any Circle instance variables can go here
} Circle;

// Defined method to create the object
Shape* circle_new(void);
```

The inheritance is replicated by always placing the direct parent object structure as the *first* variable of the derived object. When doing this in C, the structures are laid out in memory in this sequential order. I.e. any pointer to an instance of a Circle structure will also point to the Shape parent variable of the Circle struct. Thus when we wish to refer the shape instance variables when we have a pointer to a derived object such as a circle we can simply cast the pointer like:

```
Shape *shape = (Shape *) circle;
```

This is legal as the pointers refer to the same memory location.

The vtable variable is meant to contain function pointers to functions representing methods which are expected to be overridden. In the above example the method

```
void (*Area) (Shape* shape);
```

is defined. The Circle object is expected to override this method by pointing this function pointer to its own implementation.

This object orientation is very basic. There are no representations of classes and all instances share the same vtable which is strictly not necessary. To understand object orientation in C fully I recommend looking at the GTK library ([www.gtk.org](http://www.gtk.org)).

In each new module we often define some convenience macros. As an example for the Lamp module we define the macros as:

```
#define LAMP_VTABLE_PTR(ob, member) ((ob)->vtable.member)
#define LAMP_VTABLE(ob, member) (*(ob)->vtable.member)
#define CHECK_LAMP_VTABLE_PTR(ob, member) if(LAMP_VTABLE_PTR(ob, member) == NULL) { \
    ui_module_send_error(UIMODULE_CAST(ob), "Lamp Error", "member not implemented"); \
    return LAMP_ERROR; \
}

#define CALL_LAMP_VTABLE_PTR(ob, member) if( LAMP_VTABLE(ob, member)(ob) == LAMP_ERROR ) { \
    ui_module_send_error(UIMODULE_CAST(ob), "Lamp Error", "member failed"); \
    return LAMP_ERROR; \
}
```

Here we have defined LAMP\_VTABLE\_PTR to return a pointer to the vtable function; LAMP\_VTABLE returns the deference pointer; CHECK\_LAMP\_VTABLE\_PTR checks to see if a particular vtable method has been implemented by a derived object, and finally CALL\_LAMP\_VTABLE\_PTR executables the desired vtable method. If it fails then the error is sent to the user.

## 5.2 Signals Between Modules

To ease module and c-object communication a signal / call-back mechanism was developed which allows modules to send out signals without care for how many modules are listening (if any) and which types of module they are. In that way modules can communicate without having excessive dependencies – there is no consequence if there are no modules listening to the signals.

This code allows a programmer to register a call-back function or function pointer that gets executed when a particular signal from another module is emitted. Below is an example of emitting a signal from module1 and receiving it is module2. This is simply a C implementation of the publish/subscribe or observer design pattern ([http://en.wikipedia.org/wiki/Observer\\_pattern](http://en.wikipedia.org/wiki/Observer_pattern)).

Such a mechanism is necessary in this environment if we are to allow for differing hardware arrangements. In this way a camera module can send a signal to a shutter module to open the shutter just before an image is to be acquired, without the camera code being *dependant* on the shutter code. Then the camera code need not be modified on a microscope that does not have a shutter present – the camera publishes the same signals on every microscope, there may or may not be any shutters that are subscribers to those signals.

The following describes how the signal code should be used:

- 1) The emitting module (module1) must be setup to signal to any listeners:

*In module1.h*

Declare the prototype for the handler that is executed when a signal is emitted

```
typedef void (*MODULE1_CHANGE_EVENT_HANDLER) (Module1* module1, int data1, void *data2);
```

Declare the function that allows another module to connect to the signal with a handler

```
int module1_signal_changed_handler_connect (Module1* module1, MODULE1_CHANGE_EVENT_HANDLER handler, void *callback_data);
```

*In module1.c*

A marshaller function is required to marshal the data passed by the programmer to GCI\_Signal\_Emit to the callback function defined by the programmer who wishes connect to a signal. Only the programmer who is creating a signal knows the format that the call-back has to take, therefore they need to create a marshaller to pass data to this specific call-back definition.

```
static int MODULE_PTR_INT_MARSHALLER (void *handler, void *callback_data, GCI_Signal_Arg* args)
{
    typedef void (*HANDLER) (Module1*, int, void *);
    HANDLER func;
    assert(handler != NULL);
    func = (HANDLER) handler;
    func ( (Module1 *) args[0].void_ptr_data, (int) args[1].int_data, callback_data);
    return SIGNAL_SUCCESS;
}
```

Create a new signal that uses the marshaller above.

```
GCI_Signal_New (UIMODULE_SIGNAL_TABLE(module1), "ModuleChanged", MODULE_PTR_INT_MARSHALLER);
```

Define the function that allows another module to connect to the signal with a handler. This will simply be a wrapper for GCI\_Signal\_Connect.

```
int module1_signal_changed_handler_connect (Module1 *module1, MODULE1_CHANGE_EVENT_HANDLER *handler, void *callback_data)
{
    if( GCI_Signal_Connect (UIMODULE_SIGNAL_TABLE(module1), "ModuleChanged", handler, callback_data) == SIGNAL_ERROR) {
        return MODULE1_ERROR;
    }
    return MODULE1_SUCCESS;
}
```

To emit the signal to execute all the handlers in other modules you call

```
GCI_Signal_Emit (UIMODULE_SIGNAL_TABLE(module1), "ModuleChanged", GCI_VOID_POINTER, module1, GCI_INT, int_data);
```

2) When connecting a call-back to a signal the following needs to be carried out in the receiving module (module2).

*In module2.c*

Call the function to connect our handler, usually in the initialise function

```
module1_signal_changed_handler_connect (microscope->module1, OnModule1Changed, microscope);
```

The handler executed when signal is emitted

```
static void OnModule1Changed (Module1* module1, int int_data, void *data)
{
    Microscope* microscope = (Microscope*)data;
    // Do Stuff based on the ptr to module1, int_data or microscope.
}
```

### **5.3 DeviceListConfigModule**

There are some devices which have a reference to a DeviceListConfigModule module. This includes the fluorescence cube manager, objective manager and the optical path manager. This module allows other modules to manage a list of hardware items and their details. For example the cube manager manages a list of fluorescence cubes that has excitation and emission wavelengths. The deviceListConfigModule not only manages this list but it allows the user to select between hardware items are in use at any particular time. This allows us to ship a config listing all types of objectives or cubes and the user can simply select which device is present on a system. Internally the DeviceListConfigModule has two CVI List controls: one manages the items present/in use on a system and the other contains items that are not currently on the system. These lists contain Node objects which point to a specific structure such as a cube that the client developer defines.

To use DeviceListConfigModule the client developer has to code two methods: *save\_node\_as\_ini\_fmt* and *read\_node\_from\_ini\_fmt*. These simply read a *CMDeviceNode* from an ini file or write a *CMDeviceNode* to an ini file. This includes filling in the specific item data that node->device points at.

## **6 High-level Modules**

### **6.1 Microscope Modes**

Our microscope code allows the user to change the modes of the microscope. These modes are usually "Bright field", "Fluorescence" and "Laser Scanning". Not all systems will have all these modes. When the user changes mode the setting of the old mode are stored and reread when the user returns to this mode. These settings are the current value of any hardware and the positions and visibility of any user interface elements. The user can also save or restore settings for each mode manually by selecting load or save settings from the configuration menu. The user interface settings override any stored in the registry automatically by UI module. Defaults for each mode can be stored via the menus, these are loaded at start up.

### **6.2 RegionScan**

RegionScan is a module that allows one to select a region on a sample and the code moves the camera over that region by moving the stage. A series of images are then built up which are the stitched together. This code works in a multi-threaded way. A thread safe queue is first set-up. This is a simple queue that allows elements to be added or removed from multiple threads.

After this, an acquisition thread is created and run. This starts moving the stage and acquiring the data. Each frame acquired is stored with its position in the thread safe queue. When an element is added to the queue a resisted call-back in the main thread is called. At this point the acquired images and the positions can be removed from the queue and then processed. What we do is pass the position and image to the mosaic module which stitches the images together into a larger image and displays the result.

The region scan module will acquire a mosaic of the sample by moving the stage in a raster-like pattern and stopping to acquire an image from the camera in steps of the field of view (minus

some small overlap), to build up a large image in tiles. This has been well described elsewhere<sup>1</sup>. Region Scan uses the Region Of Interest module to allow the definition of the region to be scanned and Stage Plate, Focal Plane and Mosaic Window modules. What is worth documenting is the specific sequence of events and threads that handle image acquisition and storage during the scan.

After lots of setting up of the region to scan, the base filename to save to, the Mosaic Window to display an overview of progress and, if necessary, the map of found cells, all other timers that monitor various hardware components are stopped to dedicate resources to the region scan. Two threads are created: the acquisition thread (AT) and the processing thread (PT). They are linked by a CVI thread safe queue (TSQ), that is a *first in first out* queue that AT writes to and PT reads from. The TSQ will hold 'tiles' that have the image data and stage coordinates associated with them.

The AT proceeds in a loop that calculates the stage coordinates of the next position and moves the stage there, waiting for it to stop, then calls `gci_camera_get_image` (see the section on camera triggering) which handles the timing of the shutter opening etc. and then places the tile in the TSQ. If the AT proceeds much faster than the PT such that a large number of tiles are in the queue, a small delay is added to slow down the AT.

The PT proceeds in its own loop, checking that the AT is still running and then pulling tiles off the queue and performing background correction and cell finding if required. It then posts a windows message for the main thread to handle the UI by displaying the image in the main windows and placing a rescaled version of the image on the Mosaic Window. The image is also saved along with its metadata by the PT which can be the most time consuming part of the PT.

The use of these queues in this way does mean that the image displayed on the user interface may not correspond to the image last acquired nor the image currently being processed. This can be a little confusing in development but the everyday user it not usually concerned by this. The advantage of this method is that the user interface is consistent in the update of all windows; the main image display should not be out of sync with the Mosaic Window for instance. As with most GUI libraries, we believe the CVI GUI library to *not* be thread safe and hence the need to post a windows message to the main thread for it to update the user interface, but it will do this when it can!

### **6.3 Time-lapse**

The time-lapse code allows the users to define a number of points on a sample and then revisit those points and perform an action at each point. The revisiting and actions performed are defined by Python scripts. The microscope code has Python wrapper functions for the most important hardware functions and these can be called from the python scripts.

Time-lapse script structure consists of three methods that are called plus the abort method. The `timelapse` module assumes the following methods are defined in each timelapse script.

- `OnAbort()` - Called only if the user cancels the operation and is usually used to set a flag in the script for other methods to read and abort processing.
- `OnStart()` - `OnStart` is called once when the user presses start to begin the timelapse.
- `OnCycleStart()` - Called once at the start of each cycle of points. Should contain at least

one call to `microscope.MicroscopeVisitPoints()` to start the cycle.

- `OnNewPoint(x, y, z, position)` - Called each time the stage moves to a new point.

The microscope help documentation has a list of the possible calls that can be made from the scripts.

## **6.4 Region of Interest**

The region of interest is a simple module which allows the user to select a region. This module is used from a few different higher level modules. The region scan and stage scan both use this to ask the user to select a defined region on a sample. The module provides different means to set a region. The user can specify the top left and right bottom corners or they can specify a width and height. In the old microscopy code for cell finding there was also means for specifying a centre point and a radius for defining a cell dish region.

## **6.5 Real-time Overview**

The Real-time Overview (RTO) helps in navigating the sample by forming a rough, low resolution, mosaic image of the sample from all the images acquired by the camera. The user can use this overview image as a map of where they have already viewed making it easy to revisit these areas.

This module works by subscribing to the pre- and post-capture signals issued by the camera module. The post-capture signal tells us that a new image has been acquired; this is then copied at a low resolution and pasted into the overview window. The motorised stage position is also recorded on the pre- and post-capture signals; some weighted average of these gives the location at which the image was acquired as so the position at which to paste in the new image.

If the camera is left in live mode while the RTO is running then a map of the sample is built as the stage is moved around. Depending on the stage speed and camera shutter speed these images may be blurred by motion but they do give an indication of the sample structure none the less. The user will usually dwell over interesting areas, thus recording good images of these areas. Images that are out of focus usually form reasonable images when re-sampled to a low resolution for the map.

## **6.6 Stage Scan**

The Stage Scan module is designed to give the user a rough overview picture of the complete slide or plate on the stage. Since we have opted to only allow one objective lens, using images with different fields of view to navigate the sample is not an option, this module offers a map of the sample to navigate. The downside is that it may take several minutes to scan the complete plate by moving the stage.

This feature uses the RTO (above) but in an automated way. The stage is programmed to move in a raster pattern across the plate so that it is completely covered rather than have the user move it around. To speed things up the camera is placed in its fastest mode (highest binning mode) with high gain and circa 5 ms exposure. The stage moves at a speed, pre-calculated to move as

fast as possible but to minimise gaps between images. In future we may implement some feedback of the framing rate and adapt the stage speed to that (in some cases, such as during a routine virus scan, that PC may not be able to achieve its usual framing rate).

The Region of Interest module is used to select the scanned region but it is not necessary to set the focal plane as (again) images that are out of focus usually form reasonable images when re-sampled to a low resolution for the map.

## **6.7 Stage Plates**

The Stage Plates module is designed to store information about the various plates or slides the may be used on the microscope to allow, for example, the automatic generation of points for the time-lapse module to visit every well on the plate and for the setting of safe regions on the stage. The types and arrangements of plates and slides that can be used are determined by the stage insert. The user must tell the software what type of plate and or insert is in use via the Stage Plate module. The module is based on a Device Module selector.

## **6.8 Single Photo Counting (SPC) operation**

This module is not available on all microscopes, currently Abbe, Galileo and Hooke have it. Time resolved acquisition is implemented using Time Correlated Single Photon Counting (TCSPC) and exploiting the Becker & Hickl (bh) SPC 830 board. The ATD microscopy code interfaces directly to this board via its SDK and the spcm32.dll, using the spcm32.lib and Spcm\_def.h files during compilation. We currently use a limited set of functions of this board, using it in “Scan sync in mode” mode to acquire an 3D (x, y, time) image of accumulated counts in each time bin. The number of time bins being set by the ADC resolution.

The total accumulated image of counts in each bin is usually acquired in sections. The size of these sections depends on how often the user wants an image to be displayed to show the progress of the acquisition. Sections will consist of a number of scanned frames. The first scanned frame is always displayed. A number of frames may then be acquired until the next time to display is reached.

There is a complication in acquiring a number of frames with the SPC830 board. It cannot be told to acquire a certain number of frames and then stop. It must be given a time to acquire for in seconds. This time has to be calculated, and this is done by the scanner code (since it knows the scanner speed and resolution etc.). The time to acquire for, or collection time, is given by:

```
collection_time = (frames-1) * spc->_frame_time + 0.1;
```

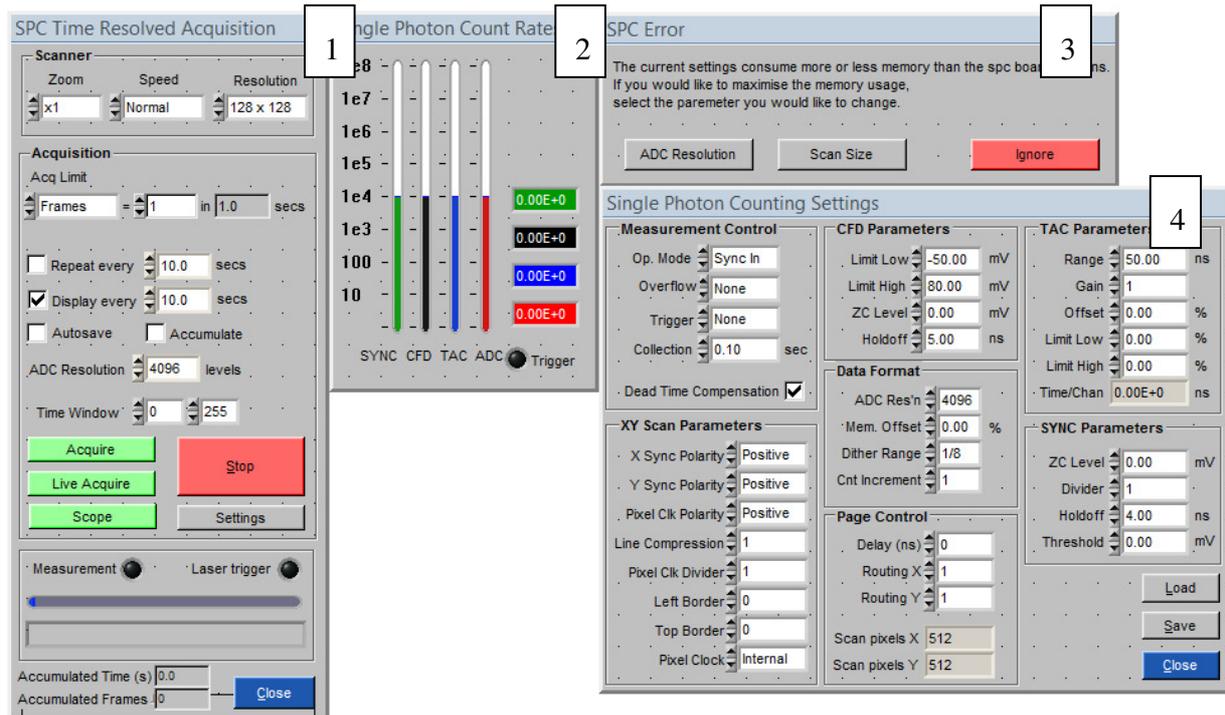
It must be just long enough to get into the last frame, the board will wait for end of last frame. A suitable time-out at twice the frame time times the number of frames is also calculated.

The code performs the following actions for each image or image section:

- SPC\_enable\_sequencer
- SPC\_start\_measurement
- SPC\_set\_parameters - read all params form the ui and send to the board.
- scanner\_start\_scan - start the scanners to scan the laser for the required number of frames.

- Wait for the board to finish acquiring using `SPC_test_state`.
- `SPC_stop_measurement` – If the scan completes successfully then the scanners will stop when the frames are complete.

The total acquisition time of photon collection is not the total acquisition time since time will be taken displaying the image and restarting the scanners when necessary. The actual acquisition time is read from the board using `SPC_get_time_from_start`.



### 6.8.1 SPC User Interface

Under normal imaging use the user interface controls have the following functions.

“Acq Limit” sets how the duration of the total scan is determined. Either by a number of frames or by a time in seconds. If the number of frames is set, the total acquisition time in seconds is calculated for the user.

“Repeat every” is a basic time-lapse function that will repeat the scan at given intervals. This has been superseded by the scripted time-lapse module. Associated with that is “Autosave” that will automatically save the image after acquisition.

“Display every” set how often the SPC intensity image will be displayed in the imaging window. The actual time used may differ from this value as it will be a whole number of frames.

“Accumulate” will cause the signal counts to be accumulated between acquisitions.

“ADC Resolution” set the number of time bins in the time-resolved measurement period.

The “Time window” function chooses some of these bins when making the SPC intensity image.

This does not affect the saved data.

“Acquire” will start the acquisition of the number of frames.

“Live Acquire” will repeatedly acquire one frame (regardless of the setting of Acq Limit) and display it.

“Scope” puts the board into oscilloscope mode such that it no longer records the x,y coordinate from the scanners and presents a 1D time-resolved signal trace. The scanner can scan when in this mode causing the trace to change as the laser is scanned across the sample, or be parked using the Zoom control.

At all times, Panel 2 shows the signal rates at different stages of single photon counting. The SYNC shows the laser repetition rate. The ADC rate shows the signal at the final counting stage and should not exceed  $10^5$  Hz to avoid pulse pile up effects.

The operation in “scope” mode for acquiring prompt (instrument response) traces.

## 6.8.2 Scanner

Zoom, speed and resolution set the appropriate parameter on the scanner module.

If the scanner zoom is changed from x1, the user is presented with a cached x1 image with a red box on it so that the user can choose the centre of the zoomed region (if there is no cached x1 image, the user is asked if they want to acquire one immediately). Similarly, if the Zoom control is set to park, an image with a cross hair is presented such that the park position can be chosen.

The resolution control has an interplay with the ADC resolution control (below) as the total memory requirement in bytes is set by X pixels x Y pixels x ADC resolution x 2 (16 bit number) and this must not exceed the memory of the board (32 Mbytes for the SPC 830 board). If the memory use is exceeded or non-optimal, Panel 3 above is displayed for the user to choose appropriate action.

## 6.9 Camera triggering via signals

There are several signals tied to the function of getting an image from the camera that handle the interplay with other hardware. On these microscope systems the decision was made to always operate the camera in a hardware triggered mode as on some cameras the performance differed when software triggering was used (this may relate to the old ImperX camera we tried). The hardware trigger comes from the shutter module that controls the epi-illumination (Fluorescence lamp). In Fluorescence (FL) mode this is straightforward, but in brightfield (BF) mode the shutter should not open as dia-illumination is being used. In BF mode, shutter module is placed in an “inhibit” mode, such that it still sends a hardware trigger to the camera but does not actually open the shutter.

The code path is as follows (common to all microscopes but dcam calls are shown, microscopes with different cameras will have the relevant calls for that camera):

```
gci_camera_snap_image
```

```
    logger_log – log that an image has been requested from the camera.
```

```
    GCI_ImagingWindow_SetFramesPerSecondIndicator – set indicator to 0
```

```

gci_camera_set_snap_mode
    gci_camera_deactivate_grab_timer – stop the thread used in live mode
    GCI_Signal_Emit "EnterSnapMode"
        shutter_close – make sure shutter is closed
        shutter_set_open_time – set to exposure time + 5.0 ms
    gci_camera_set_trigger_mode – ensure EXTERNAL TRIGGER
        logger_log – log that trigger mode requested
        camera_set_trigger_mode – camera specific, dcam code shown
            dcam_getstatus – check for IDLE, set IDLE if not so
            dcam_freeframe
            dcam_getstatus – wait for STABLE
            dcam_settriggermode
            dcam_settriggerpolarity
    camera_set_snap_mode – camera specific code, Hamamatsu dcam code shown
        dcam_getstatus – check for IDLE, set IDLE if not so
        dcam_freeframe
        dcam_getstatus – wait for STABLE
        dcam_precapture
        dcam_getstatus – wait for STABLE
        dcam_allocframe
        dcam_getstatus – wait for READY
gci_camera_get_image
    GCI_Signal_Emit "EnterGetImage" – used by real time overview (RTO)
    gci_camera_get_lock – get thread lock on camera
    camera_get_image – camera specific code, dcam code shown, tried 3 times
        dcam_idle – set IDLE
        GCI_Signal_Emit "PreCapture" – open the shutter on 90i microscopes
        dcam_capture
        GCI_Signal_Emit "TriggerNow" – open the shutter, triggers camera
        dcam_getstatus – wait for BUSY
        dcam_wait – wait for CYCLE_END
        GCI_Signal_Emit "PostCapture" – RTO
        dcam_gettransferinfo
        dcam_lockdata

```

```

dcam_getdatasize
LoadGreyScaleFIBFromArrayData – copy image to usable form
dcam_unlockdata

if camera_get_image failed 3 times do gci_camera_attempt_recovery, the
camera specific code to try and restart/re-initialise the camera

Flip or rotate the image if necessary

GCI_Signal_Emit "ExitGetImage" – background correction

gci_camera_release_lock – release the thread lock on the camera

```

## 7 External Libraries

### 7.1 *FreeImageAlgorithms / FreeImageIcs*

All image processing functions, including the code internal image format, are based on the open source Free Image library (<http://freeimage.sourceforge.net/>). We have extended this library with the Free Image Algorithms library which includes many standard image processing functions. We have released this library under an open source licence.

FreeImageAlgorithms and FreeImageIcs are written in C++ (Not labwindows dependant) and are provided with CMake build files. CMake is a tool which generates make files or IDE project files for a variety of tools. In our case usually CMake is used to generate a visual studio project that then compiles these two projects into a dynamic link library which is included in our CVI project.

These libraries have wrappers for other languages such as C# and were meant to be platform independent. Do not include any CVI functions or platform specific code into these libraries.

### 7.2 *Python*

Python is an interpreted scripting language that is very popular. One of its benefits is that it can easily be embedded within a C/C++ program. This allows one to extend the functionality of the C program with dynamic functions which allows the user to perform actions that the original program has not necessarily been designed to handle. In our case this is useful as we cannot predict exactly what actions an experimenter would wish to perform and designing for every possibility would quickly make the application overly complex and impossible to use for the user.

We have two Python modules at the moment one defines a user interface for quickly running scripts and prototyping. It includes Python functions to perform actions on the user interface elements of the program i.e. it can be controlled to "Press a button". It also includes various functions to interact with CVI. Note that these CVI wrappers are intent ally limited. Originally a full CVI wrapper was written so that anything that CVI could do could have been done within Python. However, as this allowed one to bypass purchasing the CVI development environment and National Instruments objected. Although this CVI wrapper (CVIPY) is present in the

repository we are not using it.

The second module that includes Python calls is HardwarePY which is present in the ATD\_Hardware repository. This defines the wrappers functions for the microscope module. These are the functions users with typically used to automate the microscope. Only when a particular method has not been implemented should the user resort to programmatically controlling user interface elements.

Detailed documentation on embedding Python and writing wrapper functions is located in the python documentation (<http://docs.python.org/extending>). Usually adding a wrapping function to the code is easiest by copying a previous defined wrapper function that takes and returns the same number of arguments.

### **7.3 External dll dependencies**

We use a number of external dlls in our code. Our accepted practice is to add C:\Devel\ATD\ATD\_Dlls\trunk to the system path and place these dlls there.

The dlls and descriptions are

- Python25 and Python25\_d - These are used as we embed the Python interpreter into our program.
- LStep4.dll - Used for the LStep stage controller (Marzhauser)  
<http://www.marzhauser.com/nc/en/service/downloads.html?0=>
- dcamapi.dll - Used for Hammamatsu DCam based cameras (Hammamatsu created these)
- FreeImage.dll - An image abstraction library. (<http://freeimage.sourceforge.net/>)
- freeimagealgorithms.dll - Our Image Processing Library based on FreeImage
- FreeImageIcs.dll -A library written by us that manipulates ICS files and converts them for use in FreeImage
- StringUtils.dll - Various simple string functions written by us.

### **7.4 BasicWin32Window for image display and debugging**

In the ImageViewer library that is located within the ATD\_Libraries repository there is a useful debug feature called BasicWin32Window. This function displays a FIBITMAP that you pass to it and it also displays useful debugging information about the image and its pixels. To use this function you must include the header BasicWin32Window.h You can then display an image with a function call to BasicWin32Window.

Its prototype is:

```
HWND BasicWin32Window(const char *title, int left, int top, int width, int height, FIBITMAP *dib);
```

The title parameter is used just to display in the titlebar of the window. The left, top, width, height parameters are for positioning absolutely on the screen. The final parameter is the image you wish to display. This function returns a handle to the window. So far this only has the one use of being passed to the function

```
BasicWin32Window_SetPalette(HWND hwnd, RGBQUAD *palette);
```

This function assigns a palette to this particular window so each loaded indexed image will be displayed using this palette.

## 8 References

<sup>1</sup> V. Rankov, R. J. Locke, R. J. Edens, P. R. Barber, and B. Vojnovic, "An algorithm for image stitching and blending," Proc. SPIE **5701**, 190-199 (2005).

We acknowledge the financial support of Cancer Research UK, the MRC and EPSRC.

© Gray Institute, Department of Oncology, University of Oxford, 2011.

This work is licensed under the Creative Commons Attribution-NonCommercial-NoDerivs 3.0 Unported License. To view a copy of this license, visit <http://creativecommons.org/licenses/by-nc-nd/3.0/> or send a letter to Creative Commons, 444 Castro Street, Suite 900, Mountain View, California, 94041, USA.