Ajile Suite
Software Users Guide

Contents

Chapter 1

Introduction

Ajile has developed a suite of hardware and software products that cooperatively support fast image creation and projection by Ajile DMD structured light projectors as well as coordinated fast image capture by Ajile smart cameras. These products have broad applicability in areas such as machine vision, automated inspection, vision testing and spectroscopy.

The Ajile suite is designed to make management of light a simpler process. Users can begin with an easy to use graphical user interface (GUI) to quickly and easily create projects which create, project and capture images. The GUI workflow will not only give one familiarity with the project structure and the way in which components interact, but will also enable users to create and run fully functional experiments to accomplish meaningful tasks.

For greater levels of control, flexibility and power, one can make use of the Ajile software development kit (SDK) which exposes in a programmatic form the same object-oriented project structure and overall workflow which are available in the GUI, available in either Python or C++ programming languages.

The tight integration of both hardware and software within a single coherent suite makes many tasks straightforward which were previously either extremely difficult or even impossible with other existing solutions.

1.1 Overview of Features

The following list provides an overview of some of the more interesting, unique, or otherwise nice to have features which are found in the Ajile software suite. These features along with many others will be described in detail throughout the documentation.

- Object-oriented project model describes images, sequences, frames and their relationships with one another.
- Projects created by the GUI and/or SDK, then loaded and run directly by the controller hardware.
- Control of nearly every imaging parameter (e.g. frame time, lighting settings, region of interest, etc.) on a frame by frame (per frame) basis.
- Tight synchronization of components and external devices by graphically specifying trigger rules in software which are then evaluated in hardware with nanoseconds of latency.
- Images preloaded, stored and run from controller memory, or streamed continuously from a PC over a number of available interfaces.
- Symmetric view of both projectors (which consume/display images) and cameras (which produce/-capture images). Both types of components follow the same overall project model.

- Projects easily ported between GUI and SDK applications.
- SDK applications can be run on a host PC, or run directly on the embedded Linux controller with minimal code changes.
- Embedded image processing algorithms which run in FPGA hardware but configured in software allow smart image pipelines for processing or generating images at high speeds.

1.2 Software Installation

Currently supported operating systems for the Ajile GUI and SDK include Windows 7 / 8.1 / 10 and Ubuntu Linux 16.04. Other distributions of both Windows and Linux may be possible but are not officially supported and so their operation cannot be guaranteed.

1.2.1 Obtaining Ajile Software

To obtain the Ajile software suite packages, go to the Ajile downloads section of the Ajile website at `http://ajile.ca/downloads` and select the files for your operating system of choice. You will need to enter in the login username and password which were supplied to you.

For Windows installations, download the most recent .exe installation package (e.g. ajile_installer_win64_2017-07-11_1.0-4.exe). For Ubuntu installations, download the most recent .tar.gz archive for your Linux distribution (e.g. for Ubuntu 16.04, ajile_suite_linux64_16.04_x86_64_2017-07-11_1.0-4.tar.gz).

1.2.2 Software Installation in Windows

Installing Prerequisites

To install the Ajile software in Windows we first need to install a few prerequisites which allow the software to run. Prior to installing the Ajile software please install all Windows updates by opening the Windows Update tool in Windows, checking for new updates, and selecting and installing all updates which are recommended by Windows. Note that you may need to restart your computer several times to complete the updates and each time after a restart you should again check for updates in the Windows Update tool as new updates may become available once the previous updates have been installed.

Installing Ajile Software

Once all Windows updates have been installed, double-click and open the obtained Ajile software package which is a .exe Windows installer (i.e. ajile_installer_win64_xxx.exe). This will open an installer program. Follow the prompts to install the Ajile GUI and Ajile SDK Driver to the Program Files directory into the Ajile directory (e.g. to C:\Program Files\Ajile). The installer also creates a shortcut in the start menu for the Ajile GUI.

Starting the GUI

To run the Ajile GUI, find the Ajile folder in the start menu and select the Ajile GUI application. Alternately, use the Windows search tool to search for Ajile GUI and select it from the Windows search. This will launch the Ajile GUI and it will be ready for use.

Loading SDK Libraries

Python Libraries

Installing Python and NumPy

To use the Ajile SDK for Python in Windows, you will first need to install Python and the NumPy package for Python. Currently supported versions of Python are Python 2.7.x and 3.x, 64-bit versions only. Download and install the latest 64-bit version of either Python 2.7.x or 3.x from `http://www.python.org`.

Next, the NumPy package must be installed in your Python distribution since it is used as the main tool to pass images between Python and the Ajile SDK. Details for installing NumPy can be found at `http://www.python.org`. The easiest way to install NumPy however is with Python pip tool. If your Python executable is installed at 'C:\Python\python.exe', then you can install NumPy from the command line terminal with:

```
C:\> c:\Python\python.exe -m pip install numpy
```

Installing the Ajile Python Library

With Python and NumPy installed, the Ajile Python libraries are located at 'C:\Program Files\Ajile\AjileDriver\lib_python' by default for Python 2.7.x, and at 'C:\Program Files\Ajile\AjileDriver\lib_python3' for Python 3.x. To use them with your existing Python2.7 (or Python3) installation, copy the files `_ajiledriver.pyd` and `ajiledriver.py` to your Python installation folder at %PYTHON_HOME%\Lib\site-packages. Also copy all the .dll libraries from 'C:\Program Files\Ajile\AjileDriver\lib_thirdparty' to %PYTHON_HOME%\Lib\site-packages. These are third party libraries (Pthreads and OpenCV) which the AjileDriver uses and needs to load. Once this is done you should be able to load and use the ajiledriver from you Python programs. To test it try:

```
C:\> python
>>> import ajiledriver
>>> project = ajiledriver.Project("Test")
>>> print project.Name()
Test
```

If you get similar output and the name of the Project is printed then your Python installation was successful.

C++ Libraries

The Ajile C++ libraries are located at 'C:\Program Files\Ajile\AjileDriver\lib' by default. In addition the include headers are located at 'C:\Program Files\Ajile\AjileDriver\include. For building C++ applications using the Ajile SDK we use Visual Studio 2017. In addition we use the freely available CMake (`https://cmake.org/`) to generate Visual C++ projects that use the Ajile SDK. It is possible to use the include files and library files with other build tools, but we show an example here based on CMake and Visual C++ 2017.

To build with CMake we first need a CMakeLists.txt file. The following is a minimal CMakeLists.txt file which uses the AjileDriver.

```
CMAKE_MINIMUM_REQUIRED(VERSION 2.8)
PROJECT(project_test)
SET(AJILEDRIVER_HOME "C:/Program Files/Ajile/AjileDriver/")
ADD_DEFINITIONS(-DWIN32_LEAN_AND_MEAN)
SET(SOURCE main.cpp)
INCLUDE_DIRECTORIES(
  ${CMAKE_CURRENT_BINARY_DIR}
  ${CMAKE_CURRENT_SOURCE_DIR}
  ${AJILEDRIVER_HOME}/include)
ADD_EXECUTABLE(project_test ${SOURCE})
TARGET_LINK_LIBRARIES(
  project_test
  ${AJILEDRIVER_HOME}/lib/ajiledriver.lib )
```

A minimal main.cpp source file to accompany it would be as follows:

```
#include <ajile/AJObjects.h>
#include <iostream>
void main() {
    aj::Project project("Test");
    std::cout << project.Name() << endl;
}
```

You can create the preceding CMakeLists.txt file and main.cpp file into the same directory, then run CMake and point the source directory where you created those files (i.e. run Configure followed by Generate, see Figure 1.1). You can then open the generated Visual Studio solution and build it. This will output an executable, project_test.exe, into the Release or Debug directory. If you try to run the executable you will likely get an error indicating that the ajiledriver.dll library is missing. To fix this problem, copy the file 'C:\Program Files\Ajile\AjileDriver\lib\ajiledriver.dll as well as all the .dll libraries from 'C:\Program Files\Ajile\AjileDriver\lib_thirdparty' to the location of your executable (e.g. in Release or Debug). (Note that you can install the .dll libraries to a system directory instead of the executable location.) You should then be able to run the executable in a console window and see the name of the project displayed, see Figure 1.2 which shows a console window running the created program alongside a Windows Explorer window which shows all files in the same directory.

1.2.3 Software Installation in Ubuntu

Installing Prerequisites

A number of Ubuntu packages are required to run the Ajile Software Suite. These are all installed automatically by the install.sh script found in the Ubuntu software package archive. For details of which packages will be installed please review this script.

Installing Ajile Software

To install the Ajile software in Ubuntu you will need to first extract the files from the .tar.gz archive which you downloaded. This will extract a number of .deb Debian packages and an install.sh installaer script in the same directory as the archive. The prerequisite packages and each of these Ajile packages needs to installed by running the install.sh script, which will install the Ajile software into the proper system directories. The steps to install the Ajile software in Ubuntu is therefore as follows:

1. Open a new Terminal window to get a system command line interface. On Ubuntu systems this may be called Terminal in the Application → Accessories menu.
2. Change directories to the location where you downloaded the .tar.gz archive. For example if you download the file to the /Downloads directory, the command to type in the terminal is `cd /Downloads`.

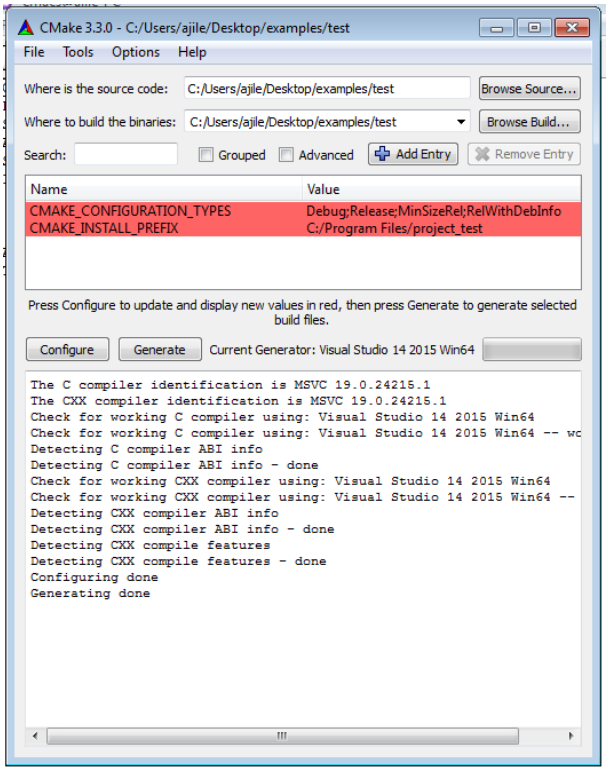

Figure 1.1: Screenshot of CMake output.

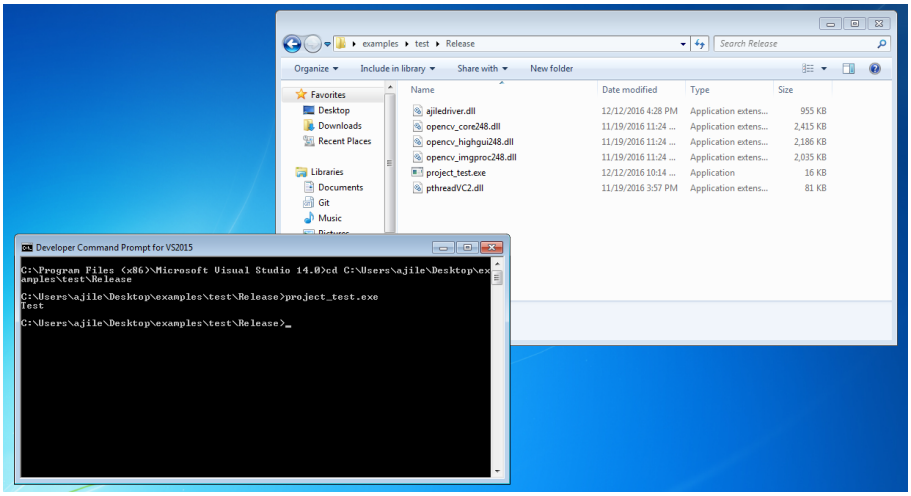

Figure 1.2: Screenshot of Running a test program.

3. Extract the files from the .tar.gz archive with

   ```
   tar xvf ajile_suite_linux64_xxx_xxxx.tar.gz
   ```

   where _xxx_xxxx should be replaced with the characters in the file you just downloaded.

4. Install the Debian packages from the archive with the command

   ```
   sudo bash ./install.sh
   ```

Starting the GUI

To start the Ajile GUI, enter the following command at the Terminal prompt:

```
ajile-gui
```

You can also start the Ajile GUI with Run Application dialog by pressing Alt-F2 in Ubuntu and entering the command `ajile-gui`.

Loading SDK Libraries

The Debian packages installs the C++ Ajile driver library to /usr/lib, the C++ Ajile driver headers to /usr/lib/ajile, and the Python Ajile driver to /usr/lib/python2.7/dist-packages. On most systems these paths are automatically found by the C++ compiler (GCC) and by the Python interpreter and you should be ready to start designing your own software using the Ajile SDK.

1.3 Device Driver Installation and Configuration

Ajile controllers come with a variety of communication interfaces including USB3, Ethernet and PCIe. Depending on the communication interface that you are using and your operating system there may need to be additional device drivers installed on the system to connect to the device. This section goes through the driver installation for each communication interface, and any additional settings that need to be made by the user in order to get the driver to work and to connect to the Ajile device.

1.3.1 USB 3.0 Drivers

For users with devices that have a USB 3.0 interface, the device drivers for USB 3 are built into the Ajile installer and will be installed automatically when the installer is run. If you experience difficulty with connecting to your device over USB 3, please refer to the Windows USB3 Troubleshooting Guide. Contact Ajile technical support for further details.

1.3.2 PCIe Drivers and Configuration

For users with devices that have a PCIe interface, additional device drivers must be installed in order to connect to the device. Please see the Ajile PCIe User Guide for further details.

1.3.3 Ethernet Configuration

For users with devices that have an Ethernet interface, all device drivers are already installed in the operating system, however the IP address settings of your network interface card (NIC) must be updated so that the NIC can connect to the Ajile device. Please see the documentation for your specific hardware for details on what its IP address should be and how to set your IP address.

1.4 Running the Examples

The Ajile Software Suite installation comes with a number of example programs/projects so that you can get started right away with running your Ajile devices. Most of the examples can be run in the GUI,

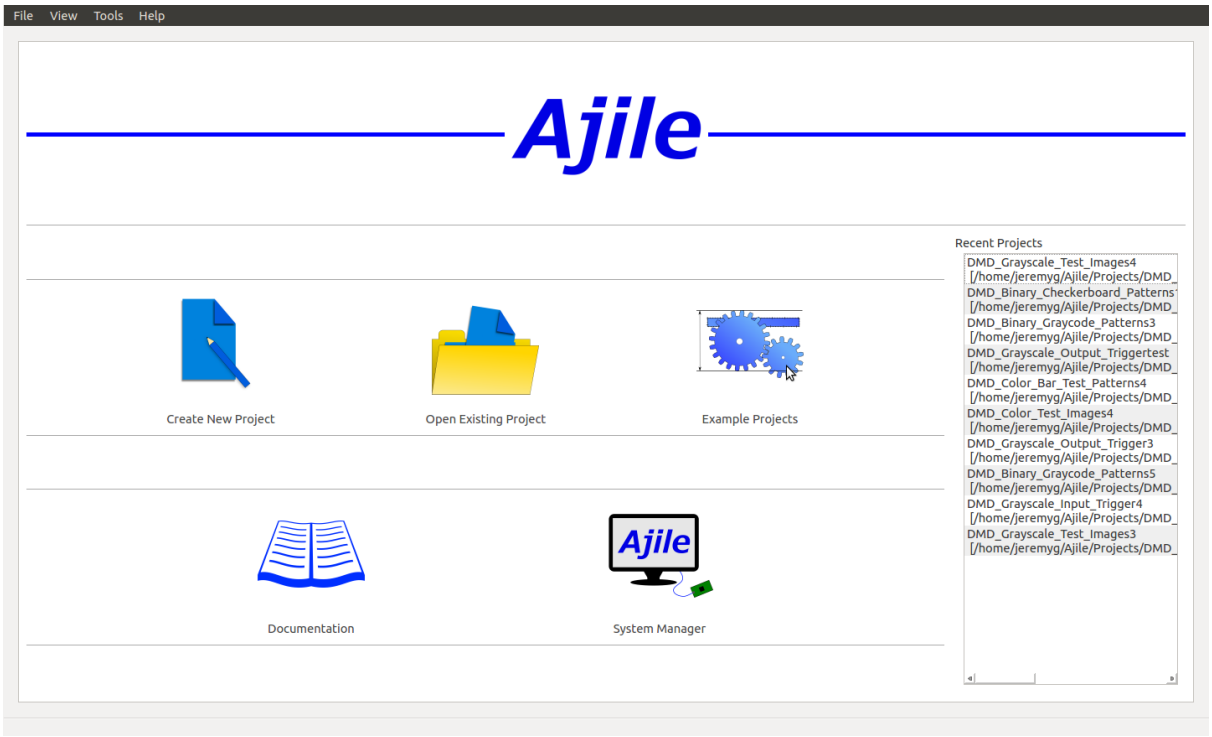

Figure 1.3: Screenshot of the Start Window, where we click on Example Projects to open the examples.

in Python and in C++. In this section we will see how to run the GUI examples, and how to install the Python/C++ examples to a suitable location and build/run them. For demonstration purposes we show how to load and run a DMD project which displays a generated checkerboard pattern and its inverse. Other examples can be run in the same way.

1.4.1 Running the Examples in the GUI

When you first open the Ajile GUI you will be presented with the 'Start' screen which shows a number of icons to get you started. Click on 'Example Projects' to open a new example project, see Figure 1.3.

Next, the list of available example projects will be shown in a new dialog box. The projects are categorized starting with the device type (DMD, Camera, etc.) then with their subcategories. For this tutorial select the 'DMD Binary Checkerboard Patterns' example then click Next, see Figure 1.4.

Clicking Next will bring up a New Project dialog. Select the AJD-4500 kit and enter any Project Name and Working Path (or leave them to the defaults) then click OK, see Figure 1.5. See Chapter 3 for more details on creating Projects and Chapter 4 for information on selecting Components (i.e. kits).

This will create a new project using the checkboard pattern example project. You can view or even edit the project using the Ajile GUI. To run the checkerboard pattern sequence on the connected DMD device we use the Run Environment. Click on the Run Environment button on the left Menu buttons to open in, see Figure 1.6, number 1. Next we must connect to the hardware device. This is done by clicking on the 'Connect to HW' button, see Figure 1.6, number 2. If successful you should see the connected components and their properties displayed. If not then you will likely need to configure your connection settings, see Chapter 9 for details.

When we are connected to the hardware device we load the project by clicking on the 'Load' button, see Figure 1.6, number 3. The load should be nearly instantaneous, then the 'Run' button will appear. Click the 'Run' button (see Figure 1.6, number 4) and the checkerboard pattern should be displayed by the device. Finally, click the 'Stop' button to stop the sequence from being run.

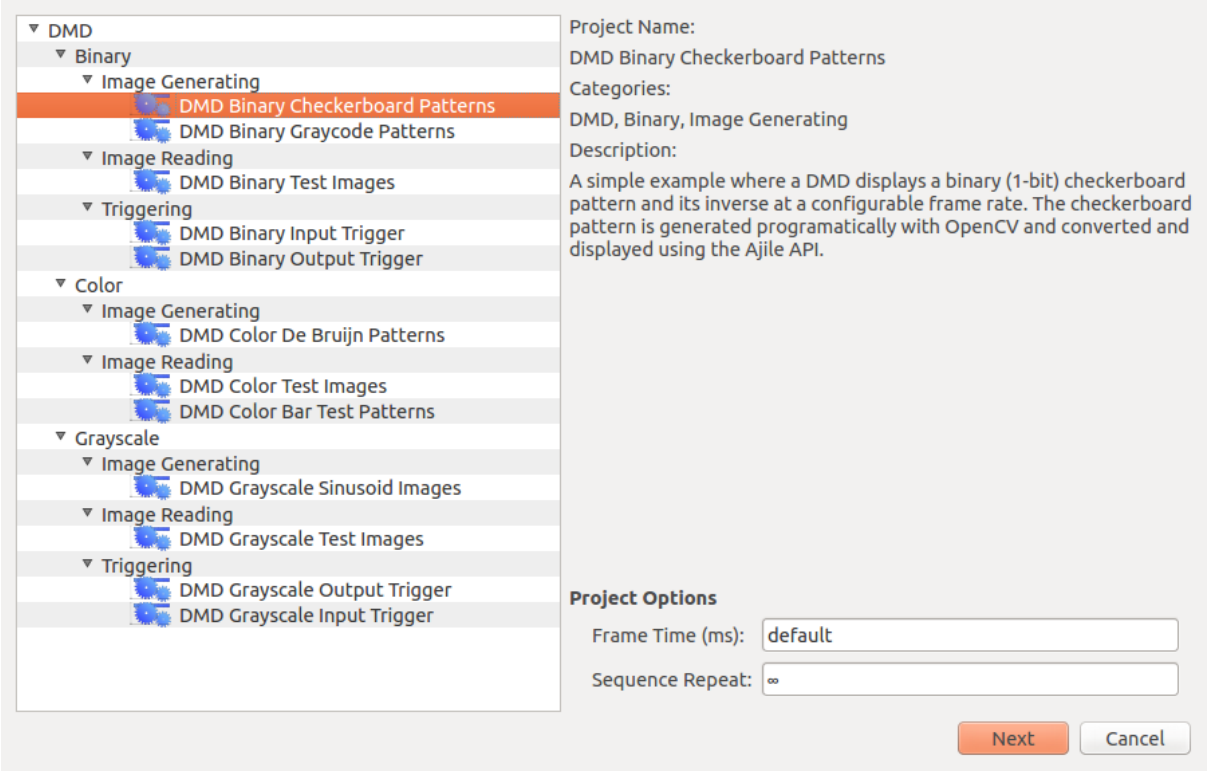

Figure 1.4: Screenshot of the Example Projects Dialog where we can select an example project to load.

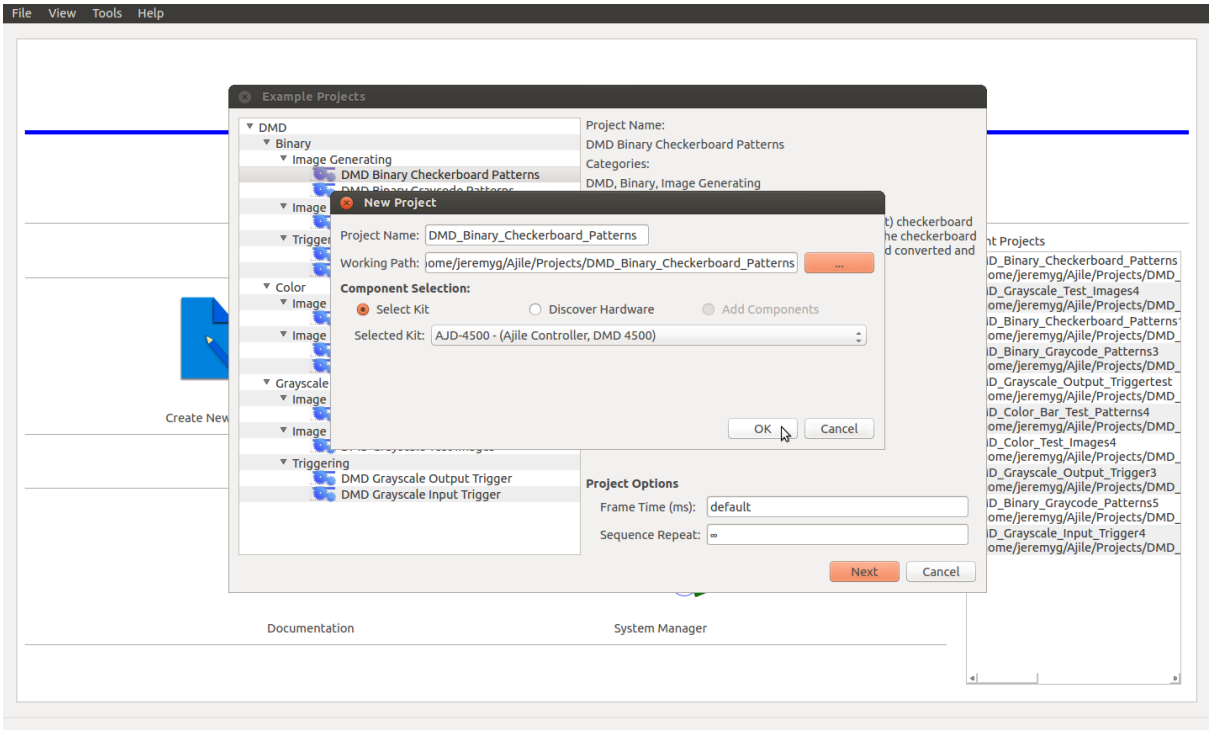

Figure 1.5: Screenshot of the creating a new Example Project.

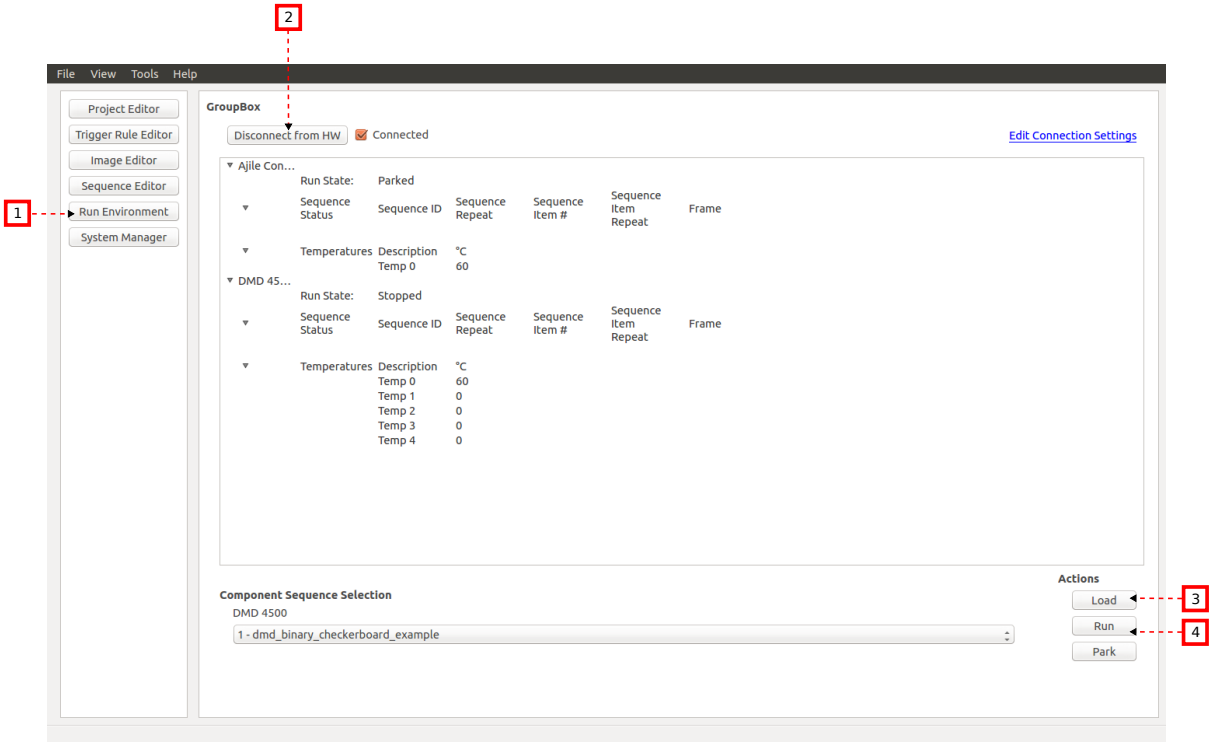

Figure 1.6: Screenshot of running an example project.

1.4.2 Running the Examples with the SDK

The Ajile SDK examples are available in both Python and C++. On Windows they are located at 'C:\Program Files\Ajile\Documentation\examples' and on Ubuntu Linux they are located at '/usr/share/doc/ajiledriver-doc/examples/'. The first step is to therefore copy the 'examples' directory to a location which is user-writable, e.g. on Windows copy 'C:\Program Files\Ajile\Documentation\examples' to your Desktop or on Linux copy '/usr/share/doc/ajiledriver-doc/examples/' to your home directory '~'.

Running the Examples in Python

Assuming that the Ajile Python libraries are installed correctly as above it is straightforward to run the Python examples. Open a Command Prompt (Windows) or Terminal (Linux) and change directories to the examples directory:

On Windows:

```
> cd C:\Users\ajile\Desktop\examples\dmd_binary_checkerboard\python
> python dmd_binary_checkerboard_example.py
```

On Linux:

```
$ cp -R /usr/share/doc/ajiledriver-doc/examples/ ~/
$ cd ~/examples/dmd_binary_checkerboard/python/
$ python dmd_binary_checkerboard_example.py
```

This should result in a checkerboard pattern being displayed by the connected DMD device.

Running the Examples with C++

Running the examples in C++ has one additional step over the Python examples which is that they must be compiled first. To do this we use CMake. Assuming that you have CMake installed in the system

path you will run the following commands to generate the CMake project:

On Windows:

```
> cd C:\Users\ajile\Desktop\examples\dmd_binary_checkerboard\cpp
> cmake .
```

In Windows this creates a Visual Studio Solution .sln. Open the `dmd_binary_checkerboard_example.sln` file with Visual Studio and build it. If we build it in Release mode the executable will be located in the Release subfolder. Finally, to run the compiled example do the following:

```
> cd C:\Users\ajile\Desktop\examples\dmd_binary_checkerboard\cpp\Release
> dmd_binary_checkerboard_example.exe
```

If you get an error message which says that you have missing .dll files, copy the .dll files from 'C:\Program Files\Ajile\AjileDriver\lib' and 'C:\Program Files\Ajile\AjileDriver\lib_thirdparty' to the Release directory, as was explained in Section 1.2.2.

On Linux:

```
> cd ~/examples/dmd_binary_checkerboard/cpp
> cmake .
```

In Linux this creates a Makefile. You can build the program then with make, then run the program:

```
> make
> ./dmd_binary_checkerboard_example
```

This should result in a checkerboard pattern being displayed by the connected DMD device.

1.5 Upgrading Ajile Software and Firmware

Ajile regularly releases new software and firmware versions which improve on performance, add new features and fix certain bugs from previous versions. For nearly all new software releases it will be required to first upgrade your device firmware before upgrading your PC software. The steps to upgrading your Ajile software are as follows:

1. Upgrade device firmware using currently installed PC software. For example, if software version 3 is installed on the PC and firmware release 3 is installed on the device and we want to upgrade the software/firmware to release 4, use the installed release 3 to first upgrade the device firmware, since it is compatable with the currently running firmware.
2. Restart Ajile device.
3. Uninstall currently installed software on the PC.
4. Install new software release (e.g. software release 4) on the PC. Connect to the device and run projects as usual.

The firmware and software upgrade steps are detailed below.

1.5.1 Upgrading Ajile Firmware

To upgrade the Ajile device firmware, first download the most recent firmware release from `http://ajile.ca/downloads`. The downloaded filename will be of the format BOOT_DMD_release_X.ajb, where X is the release version number.

Next open the Ajile GUI and open the System Manager (which is available from the Start screen or from the Navigation side bar.) With the System Manager open, physically connect the Ajile device which will receive the firmware upgrade, and connect to the device with the 'Connect to HW' button (see Section

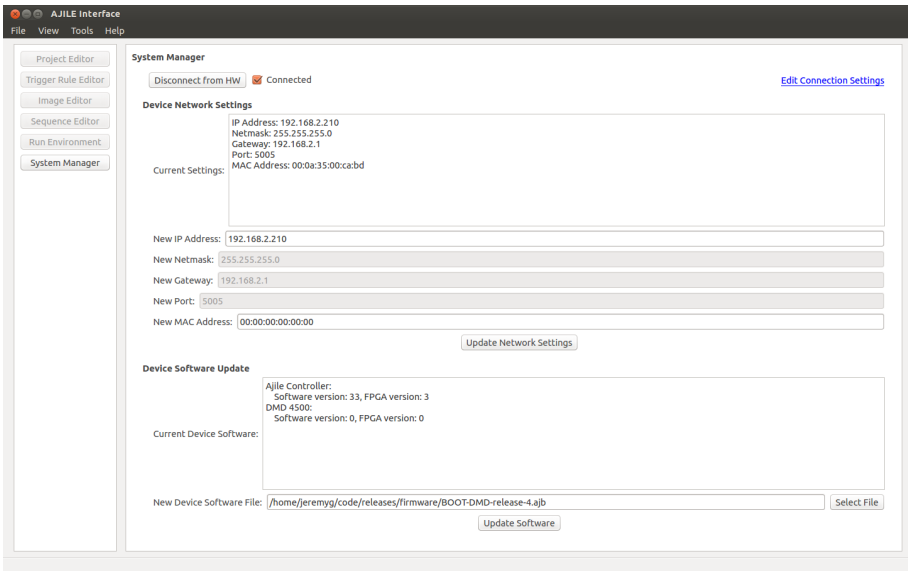

Figure 1.7: Screenshot of connecting with the System Manager.

9.1 for details on connecting.) When you are connected to the device you will see a screen similar to Figure 1.7 where the currently connected device firmware version is shown.

With the device connected, select the downloaded firmware image (i.e. BOOT_DMD_release_X.ajb) with the 'Select File' button. Finally, click the 'Update Software' button. The system will ensure that the selected firmware image is valid for the target device, then a dialog will appear to confirm whether or not to proceeed with the firmware upgrade as in Figure 1.8. Click 'Yes' then a progress indicator bar will appear, shown in Figure 1.9. The firmware update can take around 2 minutes to complete so do not close the GUI or power off the device during this procedure. The progress percentage will increment during this update. If the progress percentage stays at 0% for 2 minutes or more then there was likely a communcation error when sending the firmware image to the device. If this happens power cycle the device then reconnect and try again, or if using USB 3.0 try again but connecting using USB 2 instead. If neither of these work then please contact Ajile support. Finally, when the firmware upgrade completes a new prompt will appear and you will be instructed to power cycle the device. Do so and proceed with the PC software upgrade detailed in the next section.

1.5.2 Upgrading Ajile Software

Upgrading the Ajile PC software involves first uninstalling the exisitng Ajile software from the system, then installing the newly downloaded software release (from `http://ajile.ca/downloads`) in the same way as described in Section 1.2.

To uninstall the Ajile software suite in Windows, use the 'Add or remove programs' Windows utility, select the 'Ajile Software Suite', and click the 'Uninstall' button (note that there may be minor variations depending on your Windows version). When the uninstall completes you can re-install the newly downloaded software release on the PC.

To uninstall the Ajile software suite in Ubuntu Linux we use the apt-get command from the command line. Open a Terminal window to get a system command line interface and enter the following command:

```
sudo apt-get remove ajiledriver-doc ajile-gui python-ajiledriver libajiledriver
```

When complete the Ajile software suite will be removed from the system, after which you can re-install the newly downloaded software release.

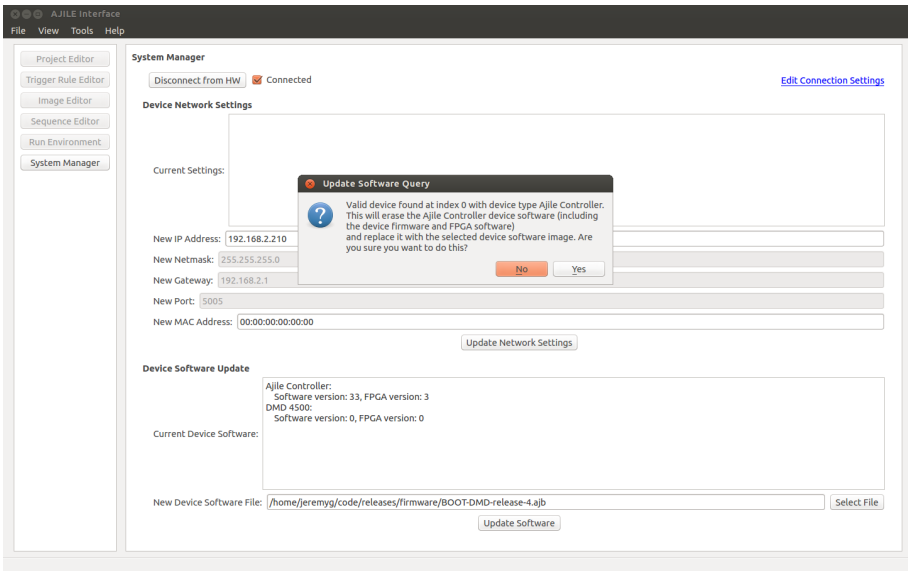

Figure 1.8: Screenshot of a valid firmware image file.

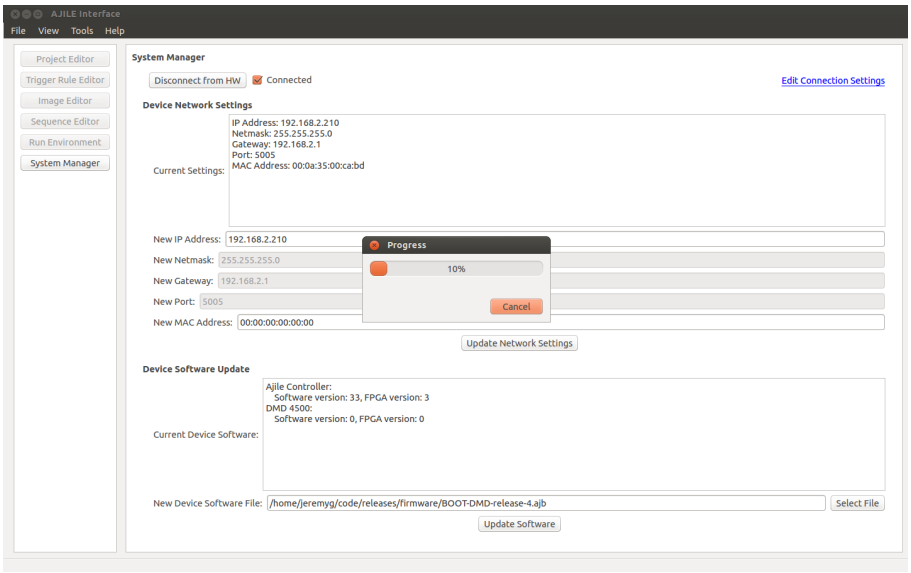

Figure 1.9: Screenshot of firmware upgrade progress.

Chapter 2

Project Model Overview

At the core of the Ajile software framework is a hierchichal group objects which describe projects that are to be run by Ajile hardware. These projects allow flexible image creation and projection by Ajile structured light projectors as well as coordinated image capture by Ajile smart cameras.

The overall project model structure is shown in a tree-like structure in Figure 2.1. The central object at the root of the hiarchy is the Project. This is the top level object which contains everything that is needed to load and run sequences of images on multiple components. A brief overview of the core objects within the project model will be described in this section. More detailed descriptions of these objects and how to work with them are found further on in the documentation.

2.1 Projects

The Project contains multiple objects which are used for controlling Ajile components. These include the descriptions of the Components themselves which map to physical hardware components, a store of Images which can be randomly accessed by frames, and most importantly, highly controllable and configurable Sequences and Sequence Items of Frames. Details about creating Projects is described in Chapter 3.

2.2 Components

The Ajile suite is somewhat unique in that it is designed from the ground up to be able to control and coordinate multiple imaging devices such as cameras, projectors and lighting controllers within a single environment.

In the Ajile Project model, physical hardware devices (i.e. controller boards) are described by objects called Components. There is one Component object per physical device. The first part of a Component is its device descriptor which tells the system what kind of device it is. Examples of devices are the DMD4500 DMD device, CMV4000 camera device and the application processor device.

A brief summary of some of the parameters which are stored inside Component objects is shown in Figure 2.2. Each Component holds its maximum image dimensions in rows and columns (in the case of imaging devices), its default, minimum and maximum LED settings (i.e. for a DMD projector) and its current trigger settings (e.g. rising/falling edge triggers, hold time, etc.). Components are typically either read out from the currently connected device, or read from hardware description files which come with the Ajile software suite. More details about setting up and working with Components will be described in Chapter 4.

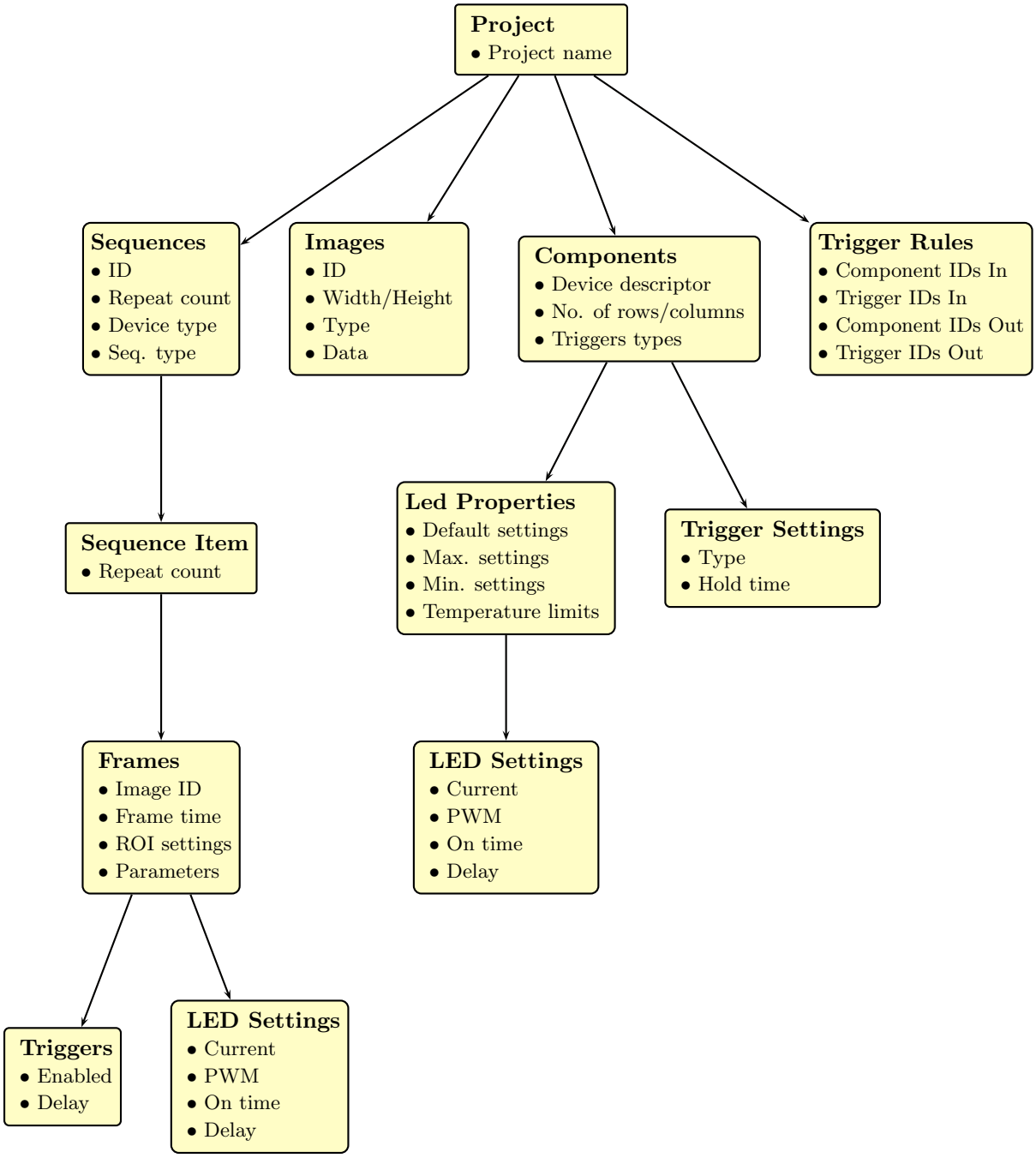

Figure 2.1: Overview of the Ajile Project model. Projects are made up of the lists of components and triggering rules along with images and sequences which will be run by those components.

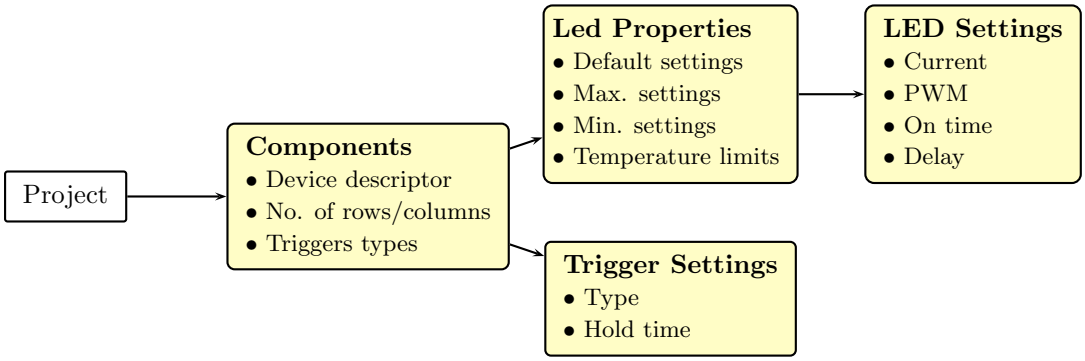

Figure 2.2: Component objects.

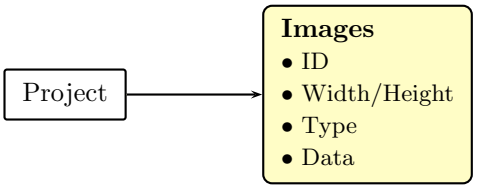

Figure 2.3: Image objects.

2.3 Images

Projects contain a store of multiple Images where each individual Image can be randomly accessed by referring to its unique Image ID. As summarized in Figure 2.3, Image objects describe the dimensions (width/height) of each image, the type of image which it referrs to (e.g. 1-bit monochrome, 8-bit grayscale, etc.), and of course a reference to the image data itself which is stored as a 2-dimensional array of pixel values. The Ajile software suite has tools for reading, writing and manipulating Images. These are described in detail in Chapter 5.

2.4 Sequences, Sequence Items, Frames

One of the the most useful features of the Ajile suite is the ability to seperate control and timing instructions from the bulk image data transfers involved in image display and capture. To accomplish this, Projects contain one or more Sequences of Frames which describe the display/capture parameters such as frame time (exposure time) and lighting values. In addition, each Frame is linked to an individual Image which is stored in the Project via its unique Image ID. In this way we have a compact set of objects which describe the frame control information which is transmitted and handled seperately from image data, allowing for much tighter and lower latency control than would be possible if this control information were embedded within image data.

Another feature which the Sequence and Frame structure enables is the ability to control individual parameters involved in image display or capture on a frame by frame basis, since each Frame object contains all of the imaging parameters needed for the current frame and is completely independant from neighbouring frames. Some of the available parameters under user control in Sequences of Frames are shown in Figure 2.4. Each Sequence is referred to by its Sequence ID, and within each sequence are a number of Sequence Item objects. Sequences and Sequence Items can repeat one or more times by modifying its repeat count. Within each Sequence Item is contained one or more Frame objects, where each Frame corresponds to an individual image display or image capture event. A user can adjust any of the Frame parameters, such as the associated Image ID, the frame time, the region of interest (ROI) as well as lighting and trigger settings.

By offering a hiearchy of Frames which are contained in Sequence Items that can repeat multiple times, which are then contained in top level Sequences which can also repeat, we can enable very sophisticated

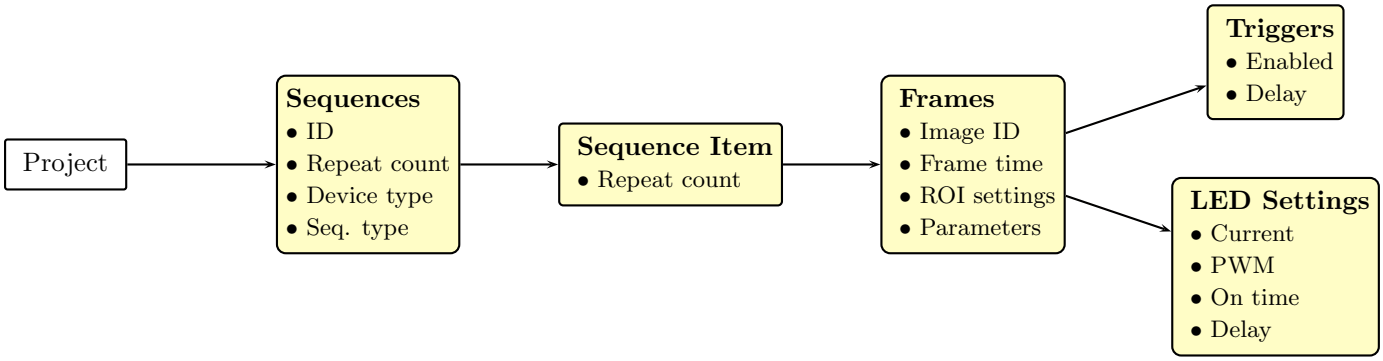

Figure 2.4: Sequence, Sequence Item and Frame objects.

sequences of frames. Figure 2.5 shows an example Sequence which contains multiple Sequence Items and Frames, along with repeat counts. When this example Sequence is run on the target component, the first Sequence Item and all of the Frames within it are run. The Frames within the first Sequence Item are repeated for whatever number of times are specified in the Sequence Item repeat count. Once the first Sequence Item has been run for its desired number of repeats, the second Sequence Item is run, and so on, until the final Sequence Item is reached. Once the final Sequence Item in the Sequence has completed, the entire Sequence then gets repeated for the desired repeat count stored in the Sequence. Also note from Figure 2.5 that each Frame has its own independant parameters which get executed sequentially, and each Frame refers to an Image in the Project using the Image ID parameter. Details of working with Sequences is described in Chapter 6.

2.5 Lighting

The Ajile suite includes high performance lighting controllers which allow users a great deal of flexibility for defining LED settings. Lighting parameters can be set up on a frame by frame basis for each individual LED channel. These lighting parameters are contained in Frame objects as shown in Figure 2.4 and they take effect during the period of the Frame in which they are contained. Lighting parameters which are under per frame control include LED current, LED pulse-width modulation (PWM) percentage, total LED on time for the frame, and the delay time after the start of the frame when the LEDs should turn on. While an incredible amount of control is possible for the Ajile lighting controllers, these settings can also simply be set to the Project level defaults for each component by using the Component LED Properties (see Figure 2.2) for cases when this additional control is not necessary. Specific details of setting up lighting, from basic settings to more advanced usage, is covered in Chapter 7.

2.6 Triggers

Since the Ajile suite at its core deals with multiple components and the interactions between them, the system includes not only a wide range of different hardware and software triggering options, but also has a sophisticated Trigger Rules engine to enable a level of control and synchronization between devices not previously possible.

Each imaging component (e.g. camera, DMD) has a set of device state signals which it outputs and a number of device control signals which are inputs. Examples of device state output signals include the beginning or end of a frame, the beginning or end of frame lighting, and a signal to indicate that the next frame is ready. Examples of device control input signals include start the next frame, end the current frame, and start or end the frame lighting.

Device states are output from the device whenever the associated event occurs. For example, the 'beginning of frame' event for a DMD device can be observed immediately when the DMD micromirrors are updated to the next image, or likewise for a camera device the 'beginning of frame' event appears when the camera exposure begins. On the other hand, device controls are inputs into a device where

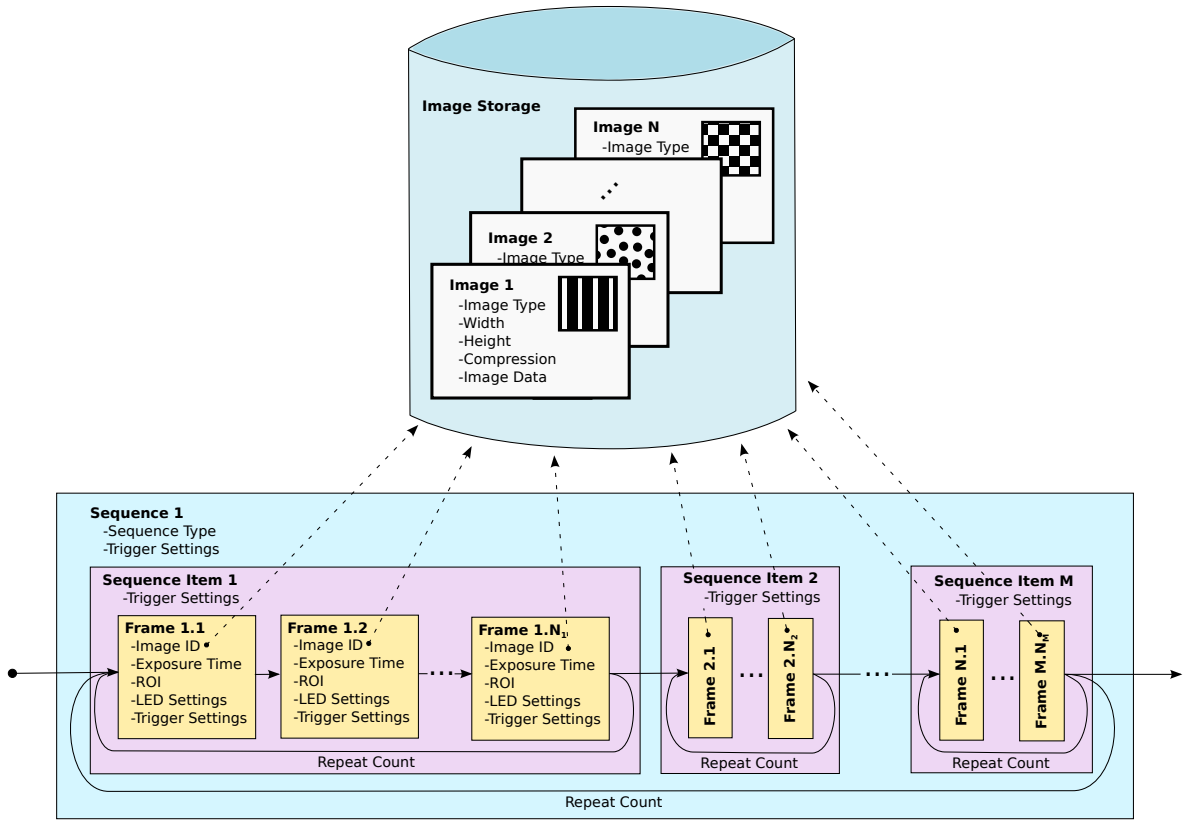

Figure 2.5: Sequence object model in a timeline view. Sequences contain one or more Sequence Items and each Sequence Item contains one or more frames. Each frame contains its own display/capture parameters including frame time, image ID, region of interest and lighting settings which can be flexibly varied on a per frame basis.

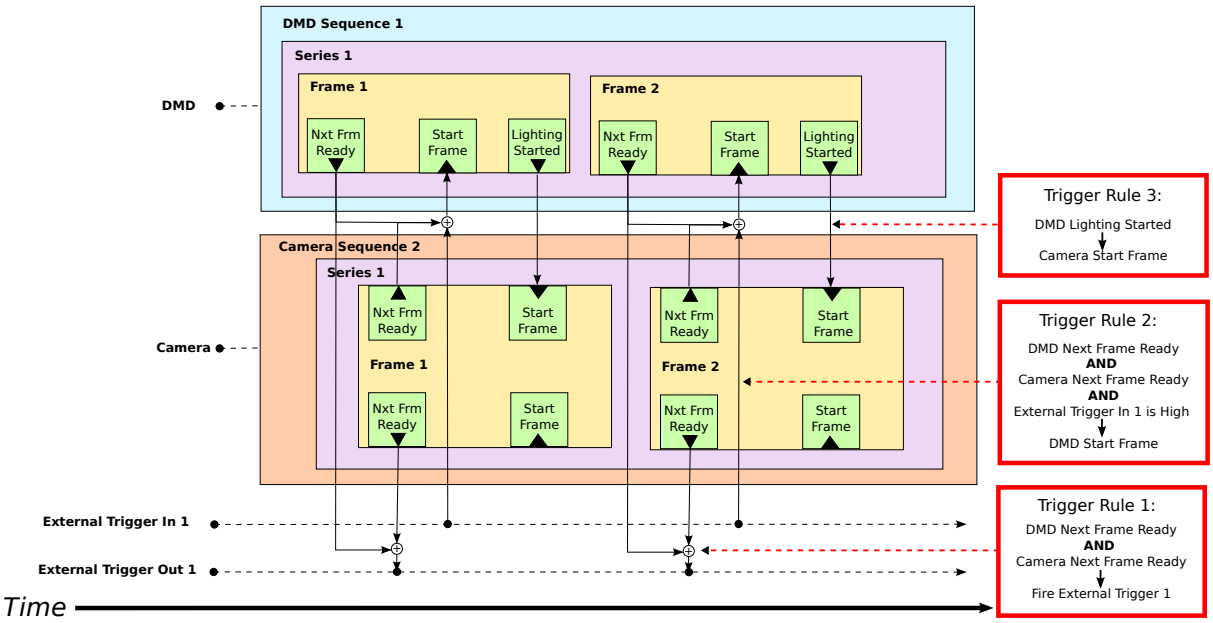

Figure 2.6: A set of three Trigger Rules defined by the Project which are evaluated in hardware to enable very tight synchronization between devices.

it pauses its operation until the control signal is observed. For example, if a 'start next frame' device control is enabled for a DMD device, the DMD will remain paused with the previous frame's image until a 'start next frame' signal is received, after which the DMD performs a reset and the micromirrors are immediately updated. Similarly for a camera, a received 'start next frame' signal immediately begins the exposure of the next frame.

Any of the device state outputs of a component can be connected to any of the device control inputs of another or the same component in order to provide tight synchronization between components. In addition, Ajile hardware devices include a number of input/output (IO) ports which provide external trigger signals for synchronizing Ajile components with external hardware devices such as other light sources or cameras. Device state outputs can be mapped to external output triggers where they are transformed into external signals, either rising or falling edge sensitive pulses or level sensitive states. External input triggers can be mapped to device control inputs, so that either edge sensitive or level sensitive signals from external devices can trigger Ajile components to take action.

Finally, the Ajile suite lets users specify multiple Trigger Rules per Project which allow for device controls and device states to be logically combined together to create a very high level of control over synchronization. These Trigger Rules are defined within the Project model, seen in Figure 2.1, using tools in the Ajile GUI and SDK. Each rule is evaluated in FPGA hardware in just 10 nanoseconds per rule while Sequences are running, resulting in very low latency synchronization. An example of a set of Trigger Rules is shown in the diagram of Figure 2.6, where the Sequences being run on both a DMD and a Camera as well as external trigger ports are drawn horizontally with respect to the time axis. In this example there are three Trigger Rules present. Trigger Rule 1 states that when the DMD ready for reset signal (i.e. the next frame is ready device state) is present *AND* when the camera next frame is ready state is also present, then fire an external output trigger signal on trigger output port 1. Trigger Rule 2 states that when both the DMD *AND* camera next frame ready states are present *AND* when an external input trigger signal on input port 1 is high, then reset the DMD device (i.e. begin the frame). Finally, Trigger Rule 3 states that when the DMD lighting turns on, then begin the camera frame. The overall result of this set of three trigger rules is a DMD and camera synchronized perfectly with each other and with an external hardware device. Details of working with Triggers and Trigger Rules is described in Chapter 8.

Chapter 3

Projects

In Chapter 2 we saw the Project object model framework in the Ajile software suite. The Project is the root of the hierchy of objects. This top level container has the list of the Components representing the physical hardware, a store of Images which will be used by the Components, a store of Sequences will be run by devices, a list of Trigger Rules connecting and synchronizing components, and a store of image pipeline results.

3.1 Project Members

Table 3.1 describes the most important members that belong to Projects. For further details of each of these and any additional Project members please refer to later sections of this guide or to the Ajile SDK Reference.

3.2 Creating New Projects

The first step to working with Projects in the Ajile suite is to create one. Creating projects is simple; here we describe how to create new projects using either the GUI or programmatically with the SDK.

3.2.1 Creating New Projects in the GUI

When the GUI is first opened (see Section 1.2.2 for starting the GUI) there will be no project opened and all GUI buttons are disabled until a new project is created or loaded. To create a new project, simply click the 'File' menu in the top menubar of the GUI and select the 'New Project' (Figure 3.1). This opens a new dialog which lets the user create a new project. As seen in Figure 3.2, with this dialog we can change the Project name to any text name (less than 255 characters). We can also change the working path where the Project and all of its files including any imported images will be stored. By default the working path is set to point to the user's home directory under $home/Ajile/Projects/*user_project_name*, but can be changed to any permissable path on the filesystem. Beneath the working path is the Components selection. Components must be configured right up front when creating the Project because different types of components will change the way in which Images and Sequences are created. Users can either select from one of several pre-defined sets of components, called kits, or connect to the hardware device and discover the list of components by querying the hardware. Components will be discussed in detail in Chapter 4. For now, we simply select any of the pre-defined kits which matches our current hardware (e.g. the Microzed Controller, DMD 4500 and CMV4000 Mono Camera). After clicking 'OK' the new project will be created and opened and the GUI buttons will become active.

The Project Editor widget will initally open which lists the current Project name, working path and components in the system. Certain properties within the components can also be edited in the Project Editor, which will be described in other sections.

Name	Description
Project Name	An optional human readable text string which can be used to identify projects. Can be any text string with a length between 0 and 255 characters. Can be left blank if not needed.
Working Path	Working path (directory) on the filesystem where the project and its associated files are stored. This is a text string absolute path. For projects that are completely generated and stored in memory and do not use the filesystem, this can be left blank.
Components	List of components that describe the hardware which the project uses.
Images	Store of images which are used by the project. An individual image with a specific Image ID is retrieved from the Images store by using its unique Image ID. Image IDs start at Image ID 1, and are typically incremented sequentially up to a maximum Image ID of 65535.
Trigger Rules	List of trigger rules which allow components to synchronize with each other and with external hardware.
Image Pipeline Results	Store of image pipeline results which have been evaluated in the project. Similarly to the image store, each individual image pipeline results is referred to by its Image Pipeline Results ID.

Table 3.1: Description of members that are in a Project.

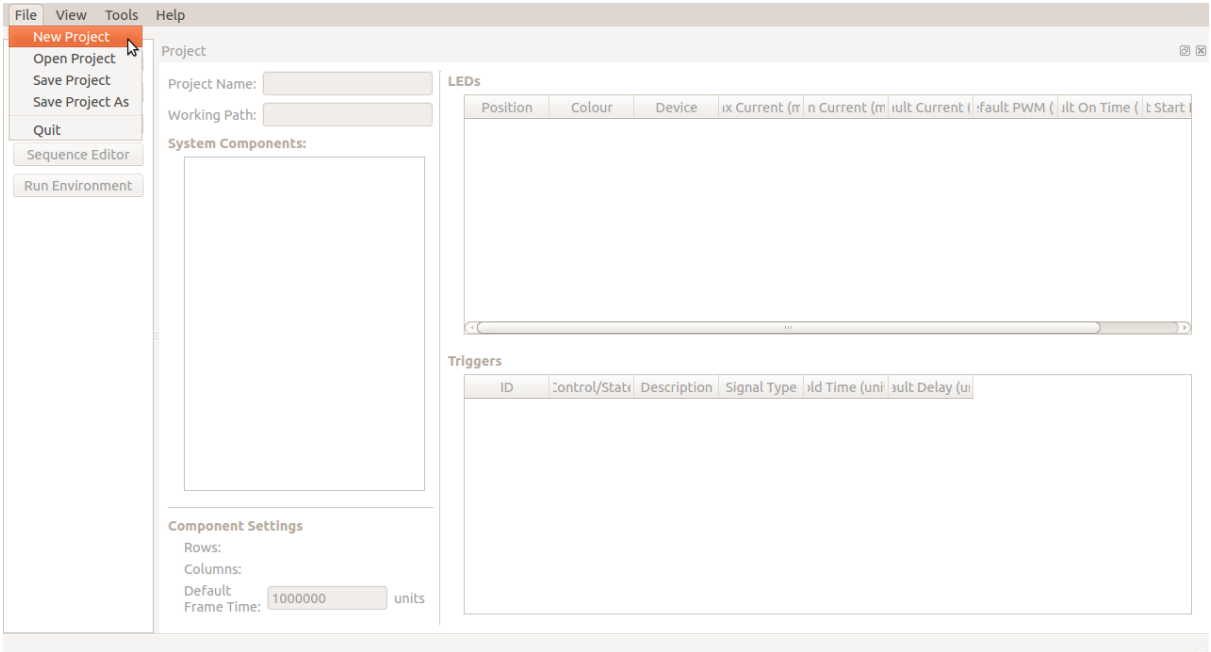

Figure 3.1: Screenshot of creating a new Project.

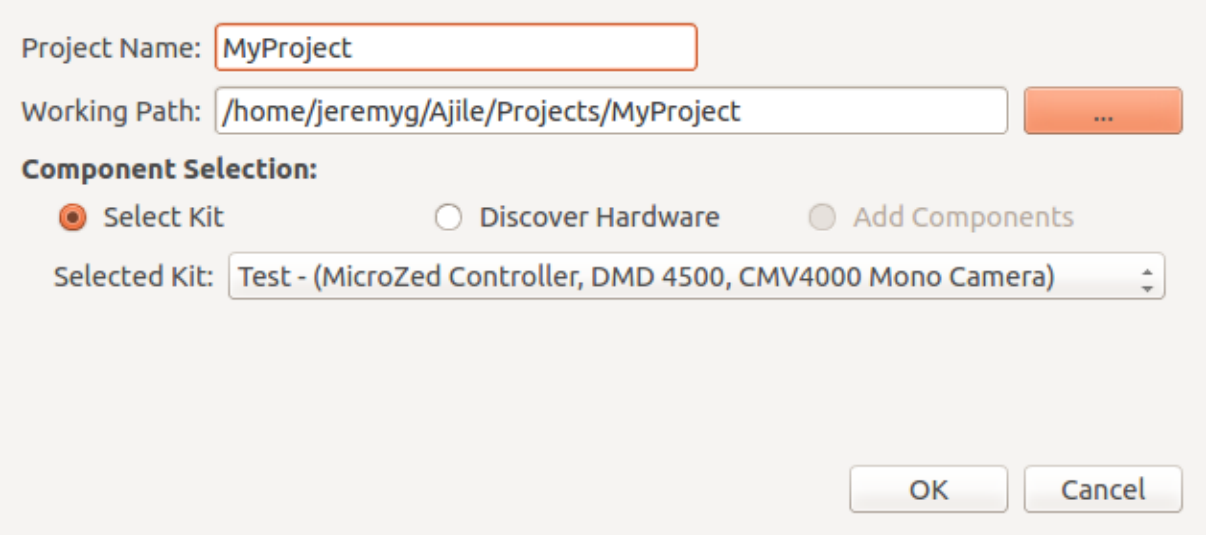

Figure 3.2: Screenshot of setting the initial Project parameters.

3.2.2 Creating New Projects in the SDK

Creating components programmatically using the Ajile SDK is accomplished by instantiating a new Project object and changing any necessary members which were described in Section 3.1. After loading the necessary SDK libraries into the source file (in either Python or C++ languages, see Section 1.2.2), we create a new Project object in Python in Listing 3.1 and in C++ in Listing 3.2. We first create the Project object and set the project name and working path with the constructor. We then modify the project name and working path and output their values.

```
from ajiledriver import * # import Ajile SDK library
# create a new project and set the name and workingPath
myProject = Project("My First Project", "path/to/my/project")
# change the project name and working path
myProject.SetName("New Project")
myProject.SetWorkingPath("~/Ajile/Projects/New_Project")
print "myProject.name: " + myProject.Name()
print "myProject.workingPath: " + myProject.WorkingPath()
```

Listing 3.1: Python example of creating and modifying a basic project.

```
#include "AJObjects.h" // import Ajile SDK library
using namespace aj;
#include <iostream> // import cout for printing
using namespace std;
void FirstProjectExample() {
    // create a new project and set the name and workingPath
    Project myProject("My First Project", "/path/to/my/project");
    // change the project name and working path
    myProject.SetName("New Project");
    myProject.SetWorkingPath("~/Ajile/Projects/New_Project");
    cout << "myProject.name: " << myProject.Name() << endl;
    cout << "myProject.workingPath: " << myProject.WorkingPath() << endl;
}
```

Listing 3.2: C++ example of creating and modifying a basic project.

3.3 Saving and Opening Projects

After creating and modifying Projects, we will naturally need to be able to save the Project to the filesystem and later re-open it and continue to work with it.

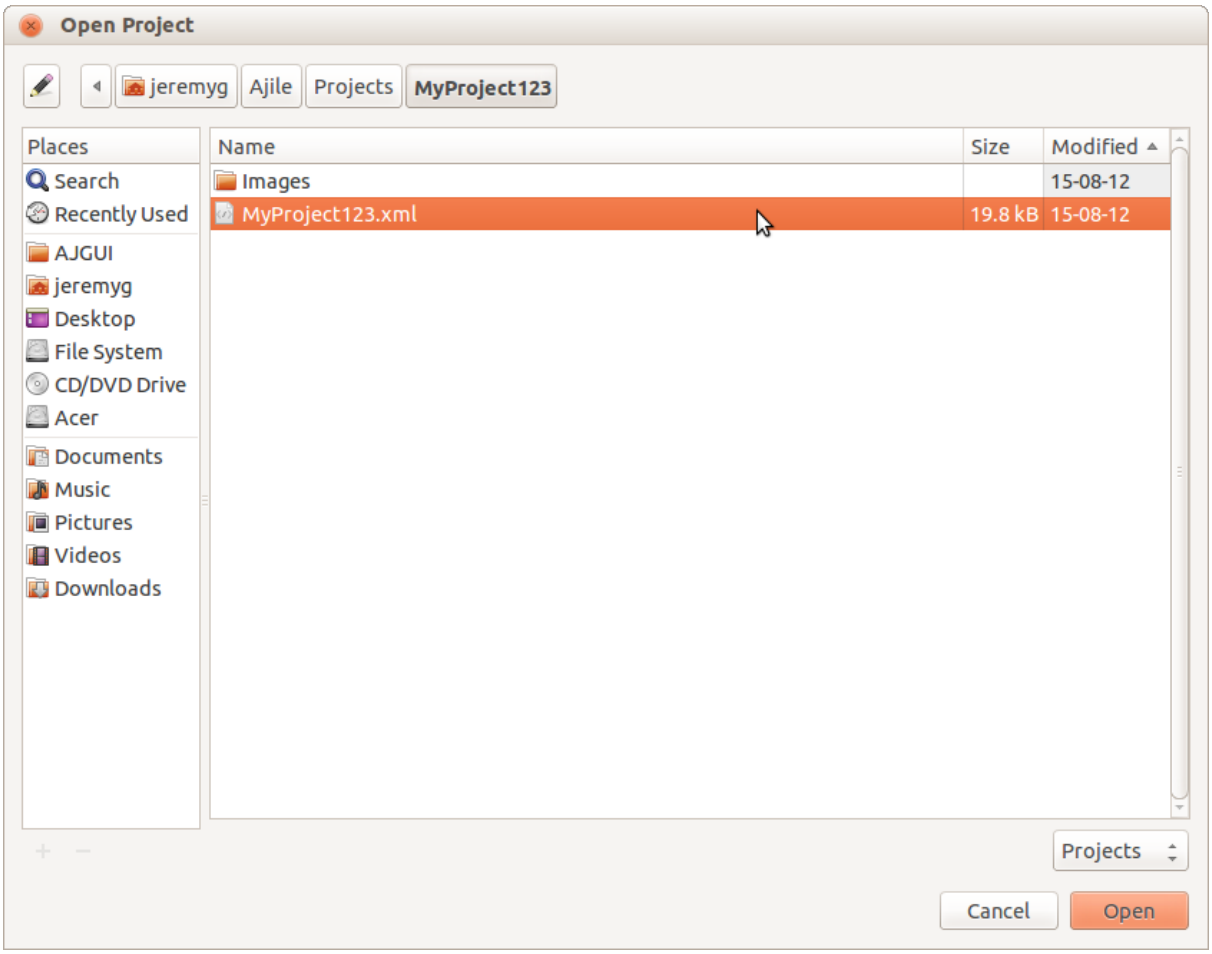

Figure 3.3: Screenshot of opening an existing Project.

3.3.1 Saving and Opening Projects in the GUI

Saving a project is accomplished in the Ajile GUI by clicking on the 'File' menu in the top menubar as we did for creating a new Project, and selecting the 'Save Project' menu item. Note that the top titlebar of the GUI main window shows the Project name which we entered earlier with an asterisk ('*') next to it indicating that changes have been made to the Project without saving it. Selecting 'Save Project' stores the Project on the filesystem in the project working path with the filename *projectName*.xml, where *projectName* is the name which we entered for our project. The 'Save Project As' allows one to save a copy of the currently open project to a new location and optionally with a new project name. The Save As dialog has the same appearance as the create project dialog which was seen in Figure 3.2.

To open an existing project, we click on the 'File' menu and this time select 'Open Project'. This opens a file browser dialog to the default location in the user home directory where projects reside (i.e. $home/Ajile/Projects/). Navigate to the working path where the desired project resides, then open the *projectName*.xml file within that directory. For example, in Figure 3.3 we open a project named MyProject123 by selecting 'MyProject123.xml' located in /home/jeremyg/Ajile/Projects/MyProject123/ on the author's Linux system (note on a Windows 7 system the home directory would instead typically be C:\Documents and Settings\users\jeremyg\Ajile\Projects). After clicking the 'Open' button the project will be opened in the GUI and we can continue to work with it.

3.3.2 Saving and Opening Projects in the SDK

Saving and opening projects in the SDK is done via two functions inside the Project - `Project.SaveToFile`(*filename.xml*) and `Project.LoadFromFile`(*filename.xml*). An example of saving a project to a filename of MyProject123.xml in the local directory, then opening it into a new Project object, and verifying that it is was properly loaded by comparing the opened project name against the saved project name, is given in Python in Listing 3.3 and in C++ in Listing 3.4.

```
# save the project to an XML file
myProject.SaveToFile("MyProject123.xml")
# open a project from a file
newProject = Project()
newProject.LoadFromFile("MyProject123.xml")
print (newProject.Name() == myProject.Name())
```

Listing 3.3: Python example of saving a Project and re-opening it as new Project.

```
void SaveOpenProjectExample(Project myProject) {
    // save the project to an XML file
    myProject.SaveToFile("MyProject123.xml");
    // open a project from a file
    Project newProject;
    newProject.LoadFromFile("MyProject123.xml");
    cout << (newProject.Name() == myProject.Name()) << endl;
}
```

Listing 3.4: C++ example of saving a Project and re-opening it as new Project.

Chapter 4

Components

After creating an initial Project, the next step is to define the types of physical hardware components which the project is intended to run on. The description of the hardware devices in our system is defined by a list of objects within the Ajile project called Components. Each physical controller board, such as a DMD controller, camera controller or application processor board, has a Component within the project to describe its parameters.

4.1 Component Members

The most important members of Components are listed in Table 4.1. A device descriptor has an identifier which indicates the type of hardware that the component describes. The device states and device controls are listed so that users know the ways in which external devices can interact with the component. The maximum number of rows (width) and columns (height) of images on the device is given. The LEDs are listed and described, along with their default, maximum and minimum settings. Lastly, the external trigger interface signals are listed and their properties and triggering conditions can be read and set.

4.2 Initializing Components

There are a few ways to initialize the list of Components which describe the target hardware. The first is to read the list of components from a hardware description file which is included with the Ajile software suite. The next is to connect to the hardware over one of the communication interfaces (e.g. Gig-E, USB 3.0, etc.) and query the hardware for the list of components. The third option is the manally create the list of components by specifying each of its properties.

Once the list of components has been set up, they are added to the Project so that they can be used. This section will describe the different ways of initializing the system components.

4.2.1 Reading Components Kits from File

Probably the simplest way of initializing the list of hardware components is to read them from a hardware description file. The hardware description files are simply Project files stored in .xml format which have been included with the Ajile software suite to make configuring the list of components more straightforward. The hardware description Project files have within them lists of components which correspond the the most common configurations of Ajile devices. These common configurations are known as *kits*, and a list of the current kits is summarized in Table 4.2. The basic idea is to read the hardware description Project from its '.xml' file (see Section 3.3) then copy the list of components from the hardware description project into our own Project.

Name	Description
Device Descriptor	A unique identifier which maps to the type of device which this component defines. Also included with the device descriptor are the device hardware and software revisions.
Device States	The list of device state output signals which are available to be monitored, such as frame started/ended and lighting started/ended.
Device Controls	The list of device control input signals which are available to synchronize the component, such as start/end the frame and start/end lighting.
Number of Rows/Columns	The maximum number of image rows and columns (resolution) available for the device. The DMD4500 device has 912 × 1140 rows by columns for example.
LED Properties	The list of device LEDs and their minimum, maximum and default settings such as current settings and on times.
External Trigger Settings	The list of input and output external trigger ports available on the device and their properties.
Image Pipeline Descriptor	The list of image pipeline operators which can be applied to images going in and out of the device.

Table 4.1: Description of members that are in a Component.

Reading Component Kits in the GUI

We already had a brief introduction with reading a component kit in the GUI when we created a new Project in Section 3.2.1 where we selected a kit in the new project dialog seen in Figure 3.2 in order to create the project. To select a specific hardware configuration to match the target hardware components, simply click the Selected Kit dropdown menu and select the kit of components which best matches the target hardware.

Reading Component Kits in the SDK

Reading components from kit project files using the SDK is done by reading the desired .xml project file corresponding to the target hardware as a new project, then copying the list of components from the read project into our existing project. This is shown in Python in Listing 4.1 and in C++ in Listing 4.2. The kit project with filename *'hw_AJ8500M.xml'* is read into the project `kitProject`, then we set the list of components in our existing project, `myProject`, to the components in `kitProject`. This is done by using the Project functions `Project.SetComponents()` and `Project.Components()` to set and get the list of components, respectively. The number of components in our project is then printed, which in this case will be 3 since the AJ8500M kit configuration has within it the Microzed Controller, DMD 4500 and CMV4000 Mono camera as shown in Table 4.2.

Kit Name	Component List	Filename
AJ4500	Microzed Controller, DMD 4500	hw_AJ4500.xml
AJ4000M	Microzed Controller, CMV4000 Mono Camera	hw_AJ4000.xml
AJ8500M	Microzed Controller, DMD 4500, CMV4000 Mono Camera	hw_AJ8500M.xml

Table 4.2: List of a number of kits (sets of components) which are provided with the Ajile software suite.

```
# load the kit file as a new project
kitProject = Project()
kitProject.LoadFromFile("./Config/hw_AJ8500M.xml")
# copy the components which were read into our own project
myProject.SetComponents(kitProject.Components())
print "Number of components: " + str(len(myProject.Components()))
```

Listing 4.1: Python example of reading components from a kit Project file and copying the components list into our own Project.

```
void ReadComponentKit(Project myProject) {
    // load the kit file as a new project
    Project kitProject;
    kitProject.LoadFromFile("./Config/hw_AJ8500M.xml");
    // copy the components which were read into our own project
    myProject.SetComponents(kitProject.Components());
    cout << "Number of components: " << myProject.Components().size() << endl;
}
```

Listing 4.2: C++ example of reading components from a kit Project file and copying the components list into our own Project.

4.2.2 Retrieving Components from the Hardware

Reading the list of components from a hardware description kit file as shown in the previous section is simple and convenient since it allows one to set up projects offline without access to the physical hardware. If the hardware is connected to our PC however, another good approach to getting the list of components into our project is to connect to the hardware device and retrieve the component list directly from the device. This has the advantage that we can be sure that the list of components is completely accurate and up to date since we are retrieving them directly.

This method of retrieving the component list does require one to connect to the hardware and interact with the device drivers. These topics are covered in detail in Chapter 9 so the reader may wish to familiarize with the overall architecture of device drivers and connecting to the hardware in Chapter 9 and Section 9.1 before continuing with this section.

Hardware Discovery in the GUI

To retrieve the list of components from the hardware in the GUI, when we are creating the project and the new project dialog is present (see Figure 3.2) we change the Component Selection radio button choice from its default of 'Select Kit' to 'Discover Hardware', as seen in Figure 4.1. Initially we are disconnected from the hardware device. When we click on the 'Connect' button the GUI software will automatically connect to the hardware device using the current connection settings and will retrieve the list of components. This retrieved component list will then appear in the 'Discoverd Kit' text box, after which we will be able to press 'OK' to accept the components and create the project using them. It is possible to change the connection settings before pressing the 'Connect' button by clicking on 'Edit Connection Settings' - for specific details on how to do this however, please refer to Section 9.1.1.

Retrieving Components from Hardware in the SDK

To retrieve components from the hardware in the SDK we need to have a basic understanding of the device drivers and how to connect to the device, which is explained in detail in Chapter 9. For now, it is sufficient to know that interaction with the hardware takes place through a central object called the HostSystem which runs on the host PC. Within this HostSystem module there is an internal Project object which keeps track of the state of the connected device including the list of components which are currently present.

To retrieve the component list from the hardware in the SDK we need to first start the HostSystem. As will be explained in Chapter 9, starting the HostSystem causes it to connect to the device using the

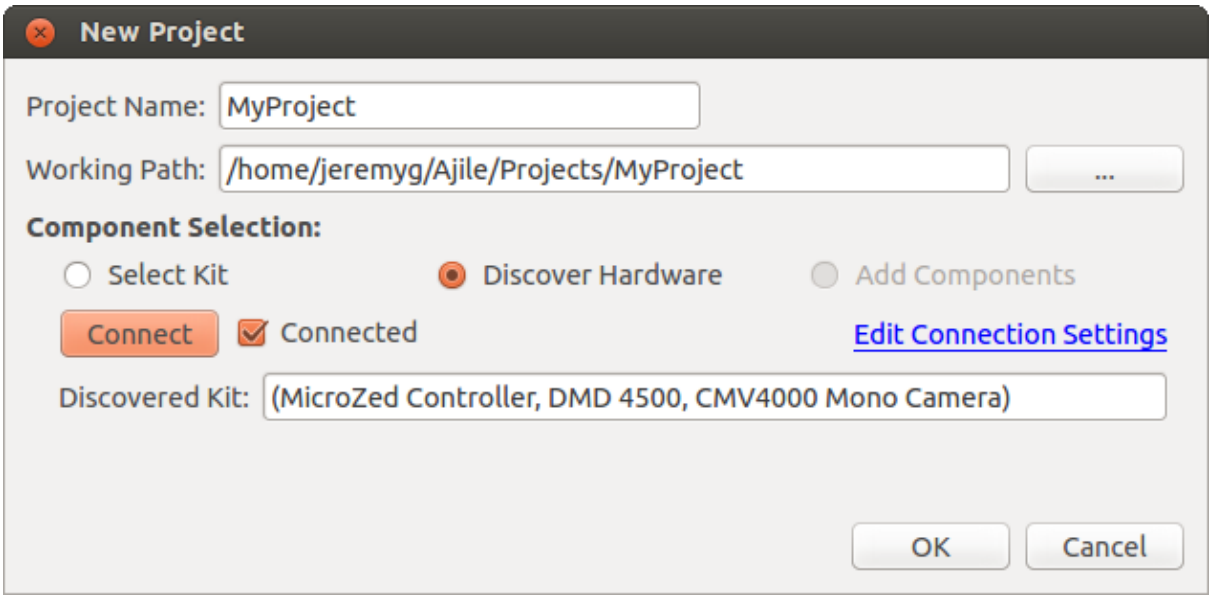

Figure 4.1: Screenshot of hardware discovery of the component list.

current connection settings, then retrieve the current state of the connected device along with its list of components.

Once the HostSystem has been successfully started, to get the list of components from the connected hardware we simply need to copy the component list from the internal Project of the HostSystem into our own project. This is accomplished with the function `HostSystem.GetProject()` which gives us access to the internal system Project, followed by getting the list of components from that returned project via `Project.Components()` as we saw in Section 4.2.1. An example of retrieving the components from the hardware by using the HostSystem is given in Listing 4.3 in Python and in Listing 4.4 in C++. The important thing to observe in this section is that setting and getting the list of components is identical to what we have already seen when reading projects from hardware kit files, with the only difference now being that the component list now comes from a project within the device driver instead of from a file on the filesystem.

```
# initialize the system module and start it
system = HostSystem.Instance()
system.StartSystem()
# copy the components from the system into our own project
myProject.SetComponents(system.GetProject().Components())
print "Number of components: " + str(len(myProject.Components()))
```

Listing 4.3: Python example of retrieving components from the connected hardware by using the Project inside the HostSystem module.

```
#include "HostSystem.h"
void RetrieveComponents(Project myProject) {
    // initialize the system module and start it
    HostSystem system;
    system.StartSystem();
    // copy the components from the system into our own project
    myProject.SetComponents(system.GetProject()->Components());
    cout << "Number of components: " << myProject.Components().size() << endl;
}
```

Listing 4.4: C++ example of retrieving components from the connected hardware by using the Project inside the HostSystem module.

4.2.3 Creating Custom Components

In most cases it will be sufficient to read the list of components from a hardware kit file or retrieve the list of components from the physical hardware. However for completeness we will briefly discuss how to create custom components from scratch if needed. This functionality is not available in the GUI but can be accessed using the SDK.

The example in Listing 4.5 and 4.6 (in Python and C++) shows how we create a new component and set its device type to a DMD 4500 device on initialization, then add the component the our project's list of components using the function `Project.AddComponent()`. In a practical example, the created component would likely need many more of its data structures need to be set in order to be useful, such as the LED properties, the device states/controls and so on. These topics will be covered in later sections and so we do not go into their details here.

```
# create a new component and set its device type
myComponent = Component(DeviceDescriptor(DMD_4500_DEVICE_TYPE))
# add the component to the project
myProject.AddComponent(myComponent)
```

Listing 4.5: Python example of creating a Component and adding it to a Project.

```
void CreateComponent(Project myProject) {
    // create a new component and set its device type
    Component myComponent = Component(DeviceDescriptor(DMD_4500_DEVICE_TYPE));
    // add the component to the project
    myProject.AddComponent(myComponent);
}
```

Listing 4.6: C++ example of creating a Component and adding it to a Project.

4.3 Configuring Components

Components have a number of different properties which were seen in Table 4.1. When reading the list of components from a kit file (Section 4.2.1) or directly from the hardware (Section 4.2.2) most of the properties should typically only be read and not changed. A few of the properties however, such as the default LED settings and external trigger settings will need to be modified by users in many instances. The details of LED settings are discussed further in Chapter 7 while details of triggers are given in Chapter 8, however in this section we introduce the basic tools in the GUI or SDK which allow us to configure these settings at the component level.

4.3.1 Configuring Components in the GUI

Configuring components in the GUI is done in the Project Editor which is shown in Figure 4.2. When a project is first created the Project Editor is the first interface to appear, however navigating to the Project Editor is done by clicking on the Project Editor button on the left navigation bar (Figure 4.2, number 1). The list of components present in the project is shown in the System Components list (Figure 4.2, number 2) and any of the components in the project can be selected in order to inspect and modify its properties. In Figure 4.2 the DMD 4500 component is selected. Basic component settings are shown in the Component Settings box (Figure 4.2, number 3) such as the number of available image rows and columns on the device. The details of each of the LEDs is shown in the LEDs table (Figure 4.2, number 4) which includes the LED descriptions and the minimum, maximum and default LED curents, PWM settings, on time and delay. Most LED properties are read only, however the default values can be adjusted which will be discussed in Chapter 7. Finally, the external trigger settings and device state outputs and device control inputs are listed in the Triggers table (Figure 4.2, number 5). The DMD 4500 component does not have any external triggers so only its device states and controls are listed.

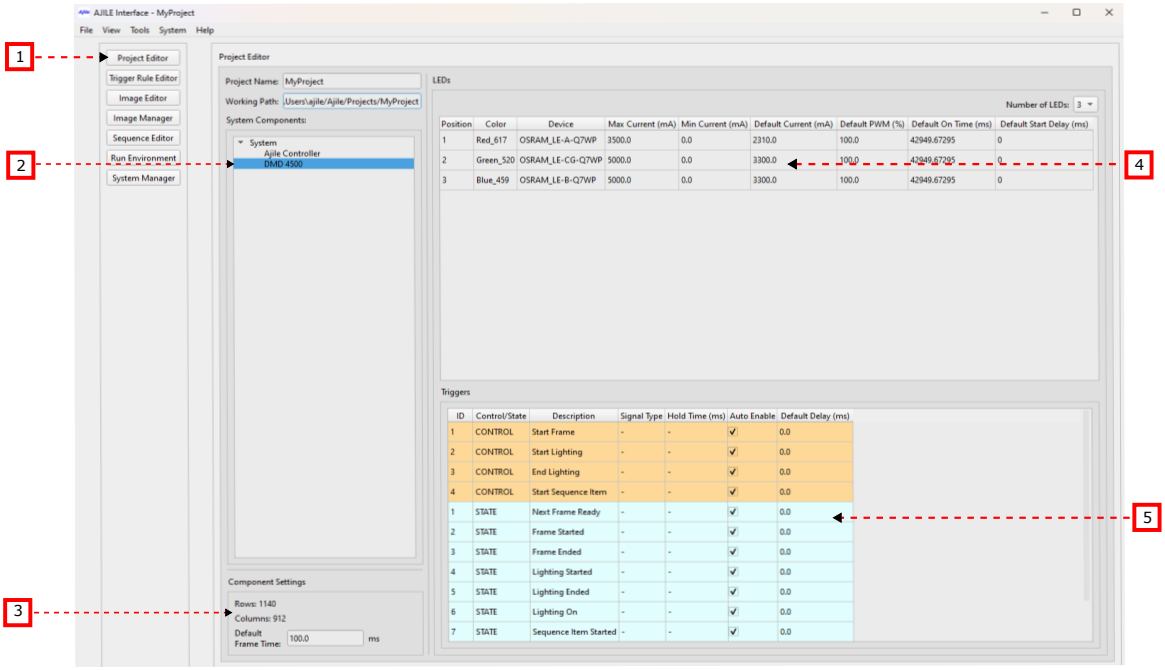

Figure 4.2: Screenshot of the Project Editor in the Ajile GUI. Project settings and components can be configured with this editor.

4.3.2 Configuring Components in the SDK

In the Ajile SDK a number of functions are available in the Component object to get and set any of its members. In most cases when it is necessary to modify a component in the project, the component must be modified externally to the project then added back into the project using the `Project.AddComponent()` function since the component list is read-only within the Project object itself. There are however a few Project level functions which allow users to more easily configure component settings. For example the function `Project.SetDefaultLedSetting()` sets the default LED setting of a specific LED for a specific component within the project without having to read, edit the re-add the component each time. These functions will be described in the coming chapters as well as in the Ajile SDK Reference.

Chapter 5

Images

The Ajile suite specializes in creating and projecting images with DMD structured light projectors and capturing and processing images with smart cameras. Both cameras and projectors treat images in a very similar way, the only difference is of course the direction of image data movement. For cameras the captured images are created on the camera, transferred to the Ajile Controller and finally to a PC, whereas for projectors the images are created on a PC then are transmitted to the Ajile controller and finally to the DMD device for display. As will be seen in later chapters, it is also possible for Ajile suite to omit the PC from the equation completely and perform all imaging with the embedded Ajile Controller directly controlling the DMD or camera in a self-contained smart system.

Ajile projects have within them a random access image store. Each Image data structure can be retrieved and updated from the project by using its unique Image ID. Frames within sequences refer to an individual Image in the project by its Image ID. Frames therefore hold all the timing and control information for performing the image display (for a DMD) or image capture (for a camera) operations, while the Images which they refer to hold the actual image data.

5.1 Image Members

Images have a number of parameters which describe what type of image it is and its dimensions (width $\times$ height), similar to an image header found in common image formats. Images also of course contain the actual image data, which is an array of pixel values, and if applicable the filename of the image. A listing and description of the most important members of an Image is given in Table 5.1.

5.2 Image Data Format

Different image based devices work with images in a variety of different formats. For example, DMD projectors use an array of binary (1 bit per pixel) images, some camera devices use grayscale 10-bit or 12-bit per pixel images, while typical commerical monitors can display 24-bit colour images. Converting images to and from these different formats can sometimes be labourous, which is why the Ajile software suite has functions in both the GUI and SDK which make importing and exporting images from and to different formats an easy process. Before going into the details of these image handling functions we will describe some of the typical image formats which will be needed by Ajile devices.

Images are stored as an array of pixel data values. Depending on the system there are many parameters which define how the pixel data values are ordered in the image array. The width, or number of columns, and the height, or number of rows, in the image, are the first parameters which must be known. Typical values are a width of 912 columns and a height of 1140 rows for the DMD 4500 device, or a width and height 2048 by 2048 for a 4 megapixel camera. The width and height (or the number of columns and rows) are shown along the axes in Figure 5.1, which shows an image represented as a rectangular array of pixels which is 8 columns wide and 6 columns high.

Name	Description
ID	A unique identifier to the project which allows each image to be referenced and retrieved within the project image store. Image IDs must be greater than zero (0) since the image with ID zero refers to the special NULL (unassigned) image.
Width/Height	The image dimensions given by the width and height, in number of pixels, of the image.
Bit Depth	The number of bits per pixel for the image.
Number of Channels	The number of colour channels for the image (e.g. 1 for grayscale, 3 for RGB colour).
Image Major Order	The major order of pixels within the image. Possible values are row-major order or column-major order.
Image Name	An optional human readable text string which can be used to help describe the image. Can be left blank if not needed.
Filename	The file name on the filesystem of the image. Can be an absolute path file name, or a relative path filename to the project working path. Can also be left blank if the image is stored completely in memory without using the filesystem.
Memory Address	Used in the SDK only, this is a pointer to the first pixel of the image pixel data array which can be used to read or manipulate the raw image data.
Size	Used in the SDK only, indicates the size, in bytes, of the image data memory.

Table 5.1: Description of members that are in an Image.

The next parameter which must be known is the number of colour channels in the image. Monochrome devices that use or produce grayscale or binary images have a single (gray) colour channel, whereas colour device will typically have three colour channels (red, green and blue). Figure 5.1 (a) represents an image with a single channel per pixel, while Figure 5.1 (c) shows an image with three channels (red, green and blue) per pixel.

The bit depth per pixel (i.e. the number of bits which represent each given pixel value) is the next parameter which needs to be determined. Most typical images found tend to have a bit depth of 8-bits per pixel since each 8-bit pixel value conveniently fits within a single byte, making these images the easiest to work with on most systems. The native image format of many hardware devices however do not use 8-bit images. The DMD 4500 device for example uses a bit depth of 1-bit per pixel, since each DMD micromirror can only have two possible states. Ajile cameras with CMOSIS image sensors on the other hand use 10 bits or 12 bits per pixel. In Figure 5.2 (a) three ordered pixels an 8-bit per pixel image is shown. The pixel indices are given by the values p0, p1, p2, ..., while the bit indices within each individual pixel are given by the values b0, b1, b2, ... In Figure 5.2 (b) on the other hand, 24 pixels of a 1-bit per pixel image are shown - that is each pixel only uses a single bit (either 1 or 0) for its value.

Finally the ordering of pixels values within the image array, known as the image major-order, needs to be known. Major-order can be one of two possible choices: row-major-order or column-major-order. Row-major-order means that pixels are ordered along rows, so that the top-left pixel at image row 0 and column 0 appears first, followed by the pixel beside it at row 0 and column 1, and so on until we reach the last pixel in row 0 at which point the next pixel which appears is from the next row at row 1 and column 0. Most systems use row-major-order, however some systems, including the DMD 4500 device, use column-major-order for its pixel ordering. In column-major-order pixels are ordered along columns where we again begin with the pixel at image row 0 and column 0, but the next pixel in the image array is the pixel below it at row 1 and column 0, followed by the pixel at row 2 and column 0, and so on until we reach the last pixel in column 0, after which the pixel at row 0 and column 1 appears. An example of row-major-order is shown in Figure 5.1 (a) and an example of column-major-order is shown in Figure 5.1 (b). Here the numbers within the pixel squares indicate its pixel index within the image

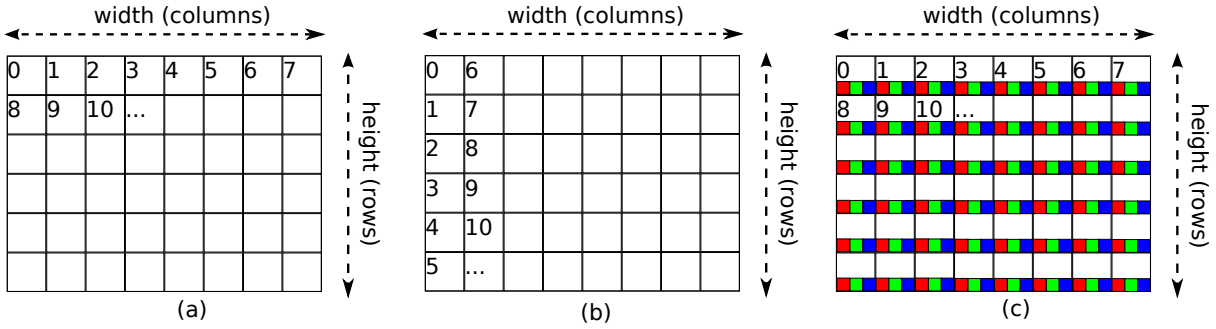

Figure 5.1: Image data formats showing (a) a single-channel, row-major-order image with 8×6 columns by rows, (b) a single-channel, column-major-order image with 8×6 columns by rows, and (c) a three-channel red, green, blue (RGB), row-major-order image with 8×6 columns by rows.

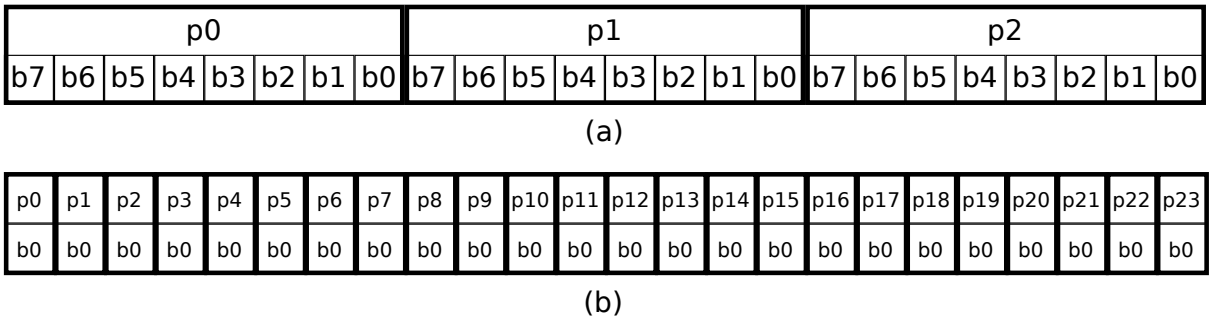

Figure 5.2: Image pixel bit depths with (a) three pixels from an image with an 8-bit per pixel bit depth, and (b) 24 pixels from an image with a 1-bit per pixel bit depth.

array, which increments on a row by row basis for row-major-order, or on a column by column basis for column-major-order.

A list of the image parameters for the current Ajile devices are shown in Table 5.2. Fortunately, for most users the details of image formats can be mostly ignored because the Ajile SDK and GUI tools can take care of all necessary image conversions to make sure that every device gets what is expected. These tools and functions will be described in the remainder of this chapter.

5.3 Creating DMD Images

Once we know what types of components are in our project and we have a DMD component included in the system, added images to our project is the first step in working with DMDs. There are two steps in

Device	Width	Height	Bit Depth	Number Channels	Major Order
DMD 4500	912	1140	1	1	Column
DMD 3000	608	684	1	1	Column
CMV 4000M	2048	2048	10	1	Row
CMV 4000C	1024	1024	10	3	Row
CMV 2000M	2048	1088	10	1	Row
CMV 2000C	1024	544	10	3	Row
CMV 300M	640	480	10	1	Row
CMV 300C	320	240	10	3	Row

Table 5.2: Native image properties used by Ajile devices.

creating a new image to add to a project. The first is the actual generating of the image data. Most users will generate their images using their own drawing or graphics programs which are found on most PCs, for example, 'Paint' on Windows systems, or the open-source GIMP (GNU Image Manipulation Program, `http://www.gimp.org`) available for Linux, Mac or Windows. Working with these image creation tools is beyond the scope of this guide, users should refer to their specific documentation. Nearly all of these programs however will have a variety of options for saving images to standard image file formats so that they can be opened by other programs. This is then where the Ajile software begins - with image files which have been created by external programs. It is *strongly* recommended to use a lossless image file format, such as PNG or BMP, when saving images in external programs for importing into DMD devices to avoid unwanted compression related image artifacts, however most standard file formats are supported by the Ajile suite. Once we have a suitable image file which we want to import into our project for display by a DMD device, the second step is to create an Image object which has the correct properties which match the given device listed in Table 5.2. In addition, an Image ID must be set for the image so that Frames can later refer to it.

5.3.1 Creating DMD Images in the GUI

To create DMD images in the GUI we use the Image Editor to open existing images from file, adjust their display characteristics, then finally accept and add them into the current project. The Image Editor is shown in Figure 5.3. To switch to the Image Editor, click on the Image Editor button on the left navition bar (Figure 5.3, number 1) in the GUI. The first thing that needs to be done is to open an image from file by clicking on the Open Image button (Figure 5.3, number 4). This brings up a file browser which can be used to locate and open an image file of a common format (such as PNG, BMP, JPG, etc). Once the image file has been opened it is displayed in the image viewer (Figure 5.3, number 3). By hovering the mouse over the image viewer and using the mouse wheel the current image can be zoomed in and out, and the view can be moved by holding the left mouse button and dragging.

If the currently open image is of the correct dimensions for the components in the system (i.e. see Table 5.2) then it can be accepted and added to the project immediately by clicking on the Accept Image button (Figure 5.3, number 2). The Accept Image button uses the currently selected Output Image Type and will add the current image to the project using the output image type. Available output image types include 1-bit mono which is the native DMD format, as well as 8-bit grayscale and 24-bit RGB colour. It is recommended to start with 1-bit mono images as the output image type while familiarizing with the system since they are the most straightforward format for a DMD. This is because a DMD is natively a 1-bit device, and therefore DMD images with higher bit depths must use multiple 1-bit images that get modulated at very high speeds in order to achieve the effect of 8-bit or 24-bit images. This will be described further in Chapter 6. Once the image has been accepted into the project, it is displayed in the list of project images along with its Image ID and image name (Figure 5.3, number 11).

For opened images which are do not have the correct dimensions or other characteristics, the Image Editor has basic tools to help with preparing images for proper display by a DMD. Since DMDs are 1-bit binary image devices and many images which are encountered are 8-bit or 24-bit, there must be some procedure for truncating the number of bits per pixel of the source image to make it suitable for a DMD. The default behavior is to simply keep only the most significant bit of a multi-bit per pixel image as the output image for the DMD, which for an 8-bit image is the same as applying a threshold with a value of 128. Other more intelligent thresholding functions are possible by using the Binraize Image box (Figure 5.3, number 6). The theshold type control can be a standard fixed value, which can be changed from the default of 128 in the Value text box, or can use Otsu, Adaptive thresholding, or image dithering. The selected threhold is applied by clicking on the Binarize button.

Resizing images to the correct dimensions to fit on the DMD array will also often be needed. Using the Ajile GUI two options for resizing are possible: image cropping and image scaling. To crop an image, the Crop Image box is used (Figure 5.3, number 7). The size of the crop selection can be set with the Crop Selection drop down list. Once selected, a red rectangle showing the selected crop area will be visble in the image viewer (Figure 5.3, number 3). The user can change the position of the cropped area by clicking anywhere in the image viewer. Once a suitable crop selection is chosen, the Crop button is clicked to apply the crop. To resize an image, the Scale Image box is used (Figure 5.3, number 8).

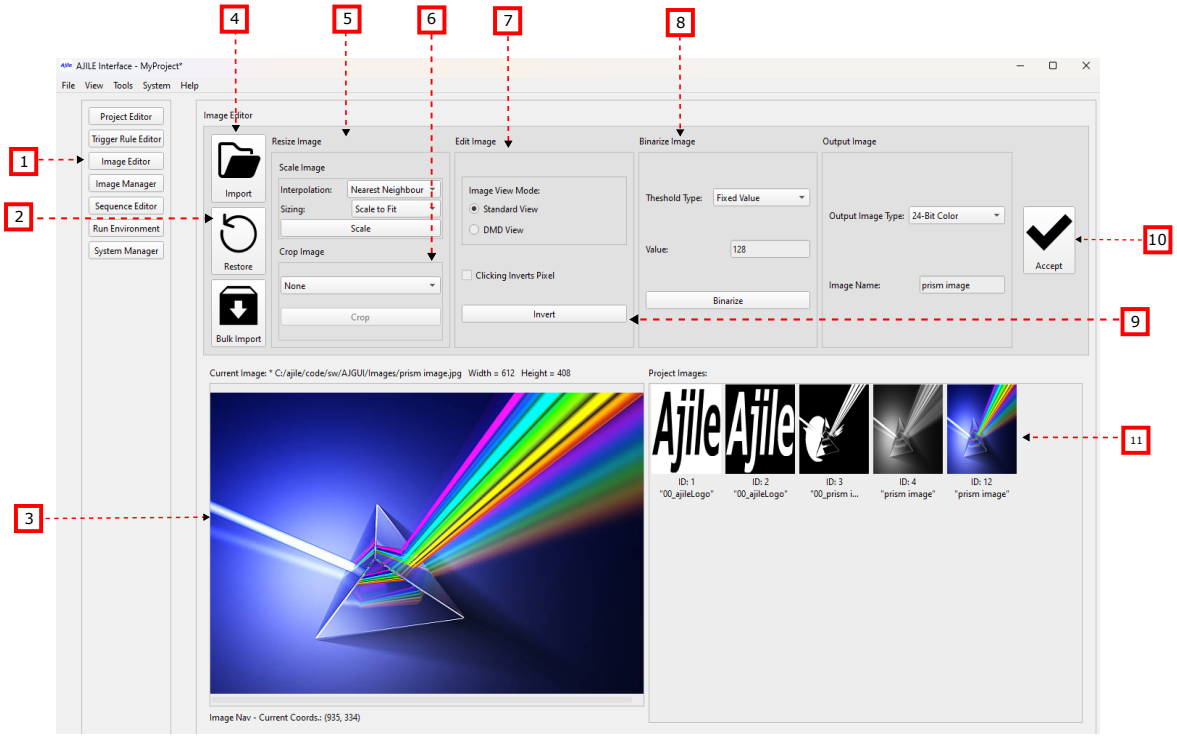

Figure 5.3: Screenshot of the Image Editor in the Ajile GUI. Images can be imported from standard image files and prepared for display with this editor.

If Scale to Fit is chosen the width and height of the image will be stretched to fit in the DMD native dimensions, while Keep Aspect Ratio will crop the image before scaling in order to maintain the aspect ratio. The type of interpolation when resizing can be adjusted as well in the Scale Image box. When the scaling settings have been set to their desired values, the Scale button is clicked to apply the scaling function.

Inverting, also known as negating, an image is another operation which is often needed for images. This can be done by clicking the Invert Image button (Figure 5.3, number 9). Finally, the currently loaded image can be reloaded from its original file at any time by clicking on the Reload Image button (Figure 5.3, number 10) which will discard any modifications made to the image.

5.3.2 Creating DMD Images in the SDK

Initializing Image Objects

The first step in creating a usable DMD image is to create a new empty Image object and set its Image ID to a positive non-zero integer between 1 and 65535 which will later be used by Frame objects to refer to it. The Image ID can be set either using the Image constructor or can be set and read using the `Image.SetID()` and `Image.ID()` functions. An example of creating two new images and setting their Image IDs to 1 and 2 using both the Image constructor method and the `Image.SetID()` method is given in Python in Listing 5.1 and in C++ in Listing 5.2.

```
# create two new images with Image IDs 1 and 2
myImage1 = Image(1)
myImage2 = Image()
myImage2.SetID(2)
print "myImage1.ID: " + str(myImage1.ID())
print "myImage2.ID: " + str(myImage2.ID())
```

Listing 5.1: Python example of creating two new Image Objects.

```
void CreateImagesExample() {
    // create two new images with Image IDs 1 and 2
    Image myImage1 = Image(1);
    Image myImage2 = Image();
    myImage2.SetID(2);
    cout << "myImage1.ID: " << myImage1.ID() << endl;
    cout << "myImage2.ID: " << myImage2.ID() << endl;
}
```

Listing 5.2: C++ example of creating two new Image Objects.

Setting Image Data

Once the Image object has been created and the Image ID has been set, there are a number of ways to set the image data which the Image object uses. The simplest way to set the image data is to use the function `Image.ReadFromFile()` which takes as its first argument the filename of any valid image file (such as a PNG or BMP). As discussed previously however, DMD images must have a specific format in order to be displayed by the DMD device (Table 5.2). Fortunately however it is easy to convert any image file into the required native format of a DMD. This is done by supplying a second argument to `Image.ReadFromFile()` which is the hardware type of the device which will use the image. The effect of passing in the device hardware type will be to automatically apply resizing and bit depth conversion functions to the input image, similar to those found in the GUI, so that it fits properly on the DMD array. Instead of using the device hardware type, it is also possible to manually specify any of the image properties by passing them into an alternate version of the `Image.ReadFromFile()`. Users that need this specialized functionality however can refer to the SDK reference manual for details. An example of reading an image from file with the filename *'ajileLogo.png'* and converting the file to the format required for a DMD 4500 device is given in Listing 5.3 in Python and in 5.4 with C++. If the input image filename is valid, then after the `Image.ReadFromFile()` function call the image `myImage` will therefore have the properties shown for the DMD 4500 in Table 5.2.

```
# create an image with ID 1
myImage = Image(1)
# read image file and convert it to DMD 4500 format
myImage.ReadFromFile("ajileLogo.png", DMD_4500_DEVICE_TYPE)
print "Image width: %d, height: %d, bitDepth: %d, channels: %d" \
    % (myImage.Width(), myImage.Height(), myImage.BitDepth(), myImage.NumChannels())
```

Listing 5.3: Python example of reading an image from file into an Image object.

```
void ReadImageFromFileExample() {
    // create an image with ID 1
    Image myImage = Image(1);
    // read image file and convert it to DMD 4500 format
    myImage.ReadFromFile("ajileLogo.png", DMD_4500_DEVICE_TYPE);
    printf("Image width: %d, height: %d, bitDepth: %d, channels: %d\n", myImage.Width(), myImage.Height(),
     myImage.BitDepth(), myImage.NumChannels());
}
```

Listing 5.4: C++ example of reading an image from file into an Image object.

While reading images from file and loading them into the project is the easiest way to work, there are cases where this is not desirable, particularly when higher performance is needed during image streaming and additional hard drive access should be avoided. The function `Image.ReadFromMemory()` has a very similary interface to `Image.ReadFromFile()` except that it accepts a raw image pixel array which resides in a memory location in RAM instead of a filename. The example in Listing 5.5 in Python and in Listing 5.6 in C++ uses the OpenCV library (`http://opencv.org/`) and NumPy (Python only, `http://www.numpy.org/`) to generate an 8-bit black background image with 100×100 pixel wide white rectangle at pixel coordinates $(100, 100)$. The generated image is then loaded into our Image object by using the `Image.ReadFromMemory()` function without using the file storage system. Just as in the `Image.ReadFromFile()` example, the generated image is converted to the format required for a DMD 4500 by supplying `DMD_4500_DEVICE_TYPE`.

```
# import OpenCV and NumPy to generate an image programmatically
import cv2
import numpy
npImage = numpy.zeros(shape=(DMD_IMAGE_HEIGHT_MAX, DMD_IMAGE_WIDTH_MAX, 1), dtype=numpy.uint8)
cv2.rectangle(npImage, (100, 100), (200, 200), 255)
# create an image with ID 1
myImage = Image(1)
# load the NumPy image into the Image object and convert it to DMD 4500 format
myImage.ReadFromMemory(npImage, 8, ROW_MAJOR_ORDER, DMD_4500_DEVICE_TYPE)
print "Image width: %d, height: %d, bitDepth: %d, channels: %d" \
    % (myImage.Width(), myImage.Height(), myImage.BitDepth(), myImage.NumChannels())
```

Listing 5.5: Python example of reading an image from memory created with the NumPy and OpenCV libraries into an Image object.

```
#include "dmd_constants.h"
using namespace aj;
#include <opencv2/opencv.hpp>
void ReadImageFromMemoryExample() {
    // import OpenCV to generate an image programmatically
    cv::Mat cvImage = cv::Mat::zeros(DMD_IMAGE_HEIGHT_MAX, DMD_IMAGE_WIDTH_MAX, CV_8U);
    cv::rectangle(cvImage, cv::Point(100, 100),
                  cv::Point(200, 200), 255);
    // create an image with ID 1
    Image myImage = Image(1);
    // load the NumPy image into the Image object and convert it to DMD 4500 format
    myImage.ReadFromMemory(cvImage.data,cvImage.rows,cvImage.cols,
                           1, 8, ROW_MAJOR_ORDER,
                           DMD_4500_DEVICE_TYPE);
    printf("Image width: %d, height: %d, bitDepth: %d, channels: %d\n", myImage.Width(), myImage.Height(),
     myImage.BitDepth(), myImage.NumChannels());
}
```

Listing 5.6: C++ example of reading an image from memory created with the NumPy and OpenCV libraries into an Image object.

Adding and Reading Images to and from Projects

When an Image object has been created and its image data has been set we can finally add the image to our project. This is done using the `Project.AddImage()` function and passing in our new image. The image is adding to the project image store at the image storage location (in memory) given by the Image ID. If the project already has an image with the same Image ID then the old image with the same ID will be overwritten. To later access the images in a project, the function `Image.FindImage()` can be used where a specific Image ID is passed in and a reference to the image with that ID is returned. An example of adding an image to a project and getting it back again is given in Listing 5.7 in Python and in Listing 5.8 in C++.

```
myProject = Project()
# create an image with ID 123
myImage = Image(123)
# add the image to our project
myProject.AddImage(myImage)
# get the image from the project
foundImage, wasFound = myProject.FindImage(123)
print str(myImage.ID() == foundImage.ID() == myProject.Images()[123].ID())
```

Listing 5.7: Python example of adding an Image to a Project and getting it back again.

```
void AddImageToProjectExample(Project myProject) {
    // create an image with ID 123
    Image myImage = Image(123);
    // add the image to our project
    myProject.AddImage(myImage);
    // get the image from the project
    bool wasFound = false;
    const Image& foundImage = myProject.FindImage(123, wasFound);
    cout << (myImage.ID() == foundImage.ID()) << endl;
}
```

Listing 5.8: C++ example of adding an Image to a Project and getting it back again.

Chapter 6

Sequences

The Ajile suite uses Sequences of Frames to control the various available timing, lighting, and imaging properties of Ajile devices on a frame by frame basis. Unlike many systems which must set all device properties in advance of running and each of the frame has identical properties, the Ajile suite allows describing complex sequences of frames where each frame can have unique properties independant of the previous frame such as exposure/display time, lighting, region-of-interest (ROI), and so on.

Each Ajile project can have one or more Sequences. Each sequence has a unique sequence ID which is used to refer to it, and one sequence which is composed of multiple frames gets run on a device at a time. An overview of Sequences, which each contain multiple Sequence Items, which each contain multiple Frames, was given in Section 2.4.

6.1 Sequence, Sequence Item and Frame Members

As was seen in the overview chapter in Section 2.4, Sequences can repeat one or more times, and contain a list of Sequence Items. Sequence Items can also be repeated one or more times, and contain a list of one or more Frames. Each individual Frame controls the display or capture parameters for a single image.

6.1.1 Sequence Members

In addition to a unique ID, a repeat count and a list of Sequence Items, Sequences have a number of other parameters which can be configured. The members of a Sequence are given in Table 6.1.

6.1.2 Sequence Item Members

Sequences have a list of Sequence Items which are run on a device in their listed order, and within each Sequence Item is a list of Frames which can be repeated one or more times. The list of members of a Sequence Item are given in Table 6.2.

6.1.3 Frame Members

Each Frame within a Sequence Item (which is within a Sequence) stores all imaging parameters for an individual frame display or capture event by a device. With the exception of only a few global project-level parameters, each frame can have complete control over each of the imaging parameters on a frame-by-frame basis. These parameters include frame time, region of interest (ROI), lighting settings, and many other which are listed in Table 6.3.

Name	Description
ID	A unique identifier to the project which allows each sequence to be referenced and retrieved within the project sequences store. Sequence IDs must be greater than zero (0) since the sequence with ID zero refers to the special NULL (unassigned) sequence.
Name	Optional human-readable name to describe the sequence. Can be left empty if not needed.
Hardware Type	The type of hardware device which the sequence is meant to run on, for example a DMD 4500 device or a CMV 4000 device.
Sequence Type	The sequence type, which can be either a 'preload' sequence or a 'streaming' sequence.
Repeat Count	The number of times that the sequence will be repeated when it is run. A repeat count of zero (0) is treated as a special value and means that the sequence will repeat forever (infinitely) until a stop command is sent to the device to stop the sequence.
Sequence Items	A list of sequence items within the sequence which will be run one after another in order of the list.
Out of Data Action	For 'streaming' sequence types only, this defines the behavior when the sequence is running on a device but the device runs out of data (i.e. more streaming sequence items).
Out of Data Item	For 'streaming' sequence types only, a sequence item which is displayed when the device runs out of data.

Table 6.1: Description of members that are in a Sequence.

Name	Description
Sequence ID	The Sequence ID which this Sequence Item belongs to.
Repeat Count	The number of times that the list of frames in the Sequence Item will be repeated before moving on to the next Sequence Item. A repeat count of zero (0) is treated as a special value and means that the Sequence Item will repeat forever (infinitely) until a stop command is sent to the device to stop the sequence, or 'Next Sequence Item' command is sent to advance to the next Sequence Item.
Frames	A list of frames within the sequence item which will be run one after another in order of the list.
Start Sequence Item Control Input	The sequence item can either start automatically after the previous sequence item finished, or can wait for a control input trigger signal if enabled. A trigger delay time can also be provided.
Sequence Item Started State Output	A status signal indicating that a new sequence item has started. Can be enabled or disabled and a delay time can be provided.
Sequence Item Ended State Output	A status signal indicating that the current sequence item has ended. Can be enabled or disabled and a delay time can be provided.

Table 6.2: Description of members that are in a Sequence Item.

Name	Description
Sequence ID	The Sequence ID which this Frame belongs to.
Image ID	An Image ID which refers to an image within the project. For image display devices such as a DMD, the Image ID determines which image data will be displayed during this frame. For image capture devices such as a camera, the Image ID refers to a storage location where the captured image data will be stored. For 'streaming' frames, the Image ID may be zero (0) and thus ignored, and the included 'Streaming Image' will be used instead.
Frame Time	The amount of time that the frame will be displayed (for image display devices such as DMDs), or the length of the image exposure time (for image capture devices such as cameras).
Region of Interest (ROI)	A rectangular region of interest which describes a subset of the imaging device area which will be used for the frame.
LED Settings	A list of LED settings which describe the LED behavior for the frame. LED settings include the LED current, pulse-width modulation (PWM) duty cycle, LED on time, and LED delay time (after the start of the frame).
Control Input Settings	The list of control input settings (e.g. start/end frame/lighting, etc.) for the frame. Each control input for the device can be enabled/disabled and can have a delay time per frame.
State Output Settings	The list of state output settings (e.g. frame/lighting started/ended, etc.) for the frame. Each state output for the device can be enabled/disabled and can have a delay time per frame.
Imaging Parameters	A list of extended imaging parameters for the frame. Each imaging parameter is a key-value pair and is specific to the intended device which will run the frame.
Streaming Image	For 'streaming' sequence types only, an entire 'streaming image' can be included inline with the frame. This image will be used once only for the current frame to allow for streaming new image data along with frames.
Image Pipeline Settings	A list of image pipeline settings which can be customized per frame.
Image Pipeline Results	A list of image pipeline results which were obtained after running the frame on the device.

Table 6.3: Description of members that are in a Frame.

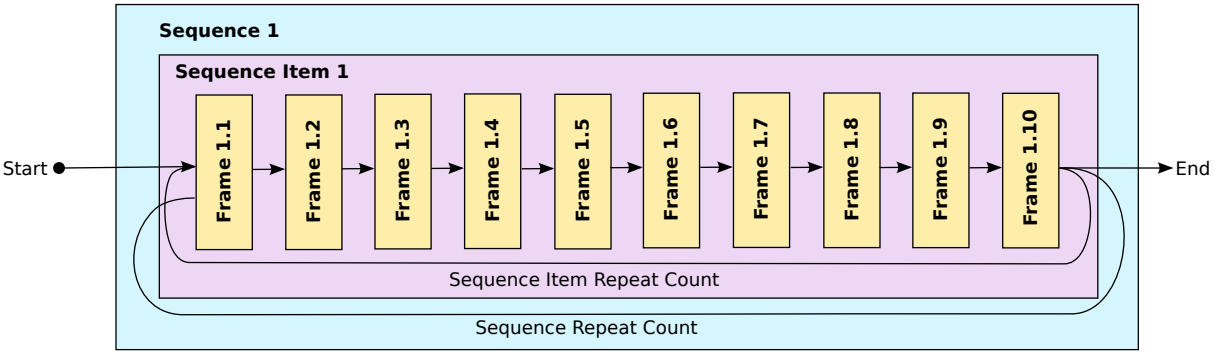

Figure 6.1: Sequence with a single sequence item which contains all frames (10 frames total in this example.)

6.2 Sequence Structure

With the available hiearchy of Sequences which contain a list of Sequence Items which themselves contain a list of Frames, a wide amount of flexibility is available to the user for constructing sequences.

One should be aware that the main use of Sequence Items is to be able to repeat sets of Frames within a Sequence. In cases where repeating sets of frames is not needed users may wish to either have a single Sequence Item which contains all frames for the sequences, or else one Sequence Item for each frame in a Sequence. Here we show a few examples of typical ways in which sequences can be structured.

Example 1 (Single Sequence Item, Multiple Frames):

Probably the simplest way to construct a sequence is to simply have within it a single sequence item which contains all frames for the sequence. This is useful in cases where repeating subsets of frames within a sequence item is not needed. An example of a sequence with a single sequence item that contains all frames is shown in Figure 6.1. When this example sequence is run on a device (e.g. on a DMD) the first (and only) sequence item will be executed first. Within this sequence item, the first frame (labelled Frame 1.1 in Figure 6.1) will be executed by the device. Once the frame time of Frame 1.1 has elapsed, Frame 1.2 is executed, followed by Frame 1.3 and so on until Frame 1.10 is run. At the end of Frame 1.10, the sequence item is either repeated if the sequence item repeat count is greater than one for the number of times specified by the repeat count, or else the sequence item is finished. Once all frames in the sequence item have been run for the specified repeat count, the entire sequence may also be repeated for the sequence repeat count in Sequence 1, in which case Sequence Item 1 begins again from the start with Frame 1.1.

Example 2 (One Sequence Item per Frame):

Another way to construct a simple sequence of frames is to create one sequence item for each and every frame, or in other words create several sequence items where each one has within it a single frame. An example of this approach is given in Figure 6.2 which shows a sequence made up of six sequence items and six frames, where each frame is within its own sequence item. The flow of this sequence would then be for the device to run Frame 1.1 in Sequence Item 1, followed by Frame 2.1 in Sequence Item 2, then Frame 3.1 in Sequence Item 3, and so on until Frame 6.1 in Sequence Item 6 is executed. If sequence item repeat counts greater than one are given in the sequence items then of course the single frame within the sequence item will be repeated for that count. As well, the entire sequence of sequence items can be repeated for the sequence repeat count.

Example 3 (Multiple Sequence Items, Multiple Frames):

If greater flexibility and control is needed, one can construct sequences which have multiple sequence items which each have multiple frames. In this way we can repeat subsets of frames within a sequence using sequence items. An example of multiple sequence items which each have multiple frames is shown in Figure 6.3. In this example, Sequence Item 1 is executed and Frames 1.1, 1.2 and 1.3 are run in order and are repeated for the repeat count specified in Sequence Item 1. Following this, Frames 2.1 and 2.2

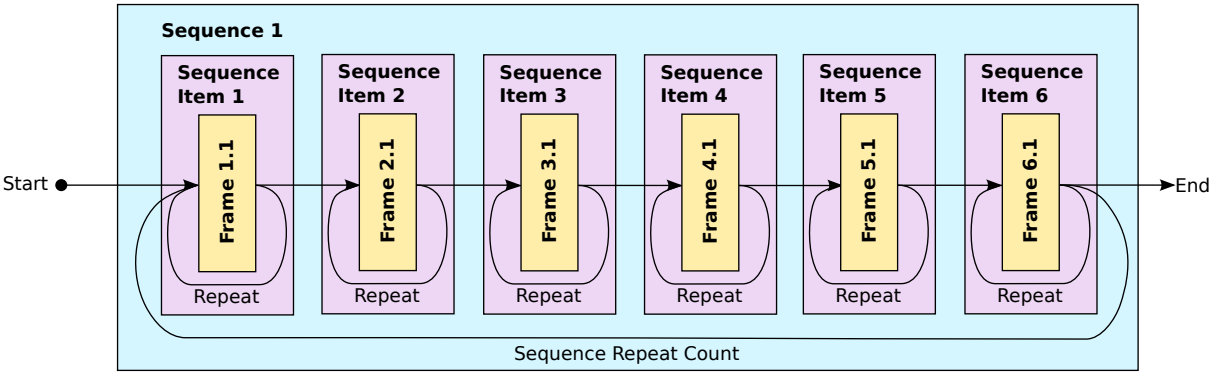

Figure 6.2: Sequence with a each Frame contained within its own Sequence Items (6 frames with 6 sequence items total in this example.)

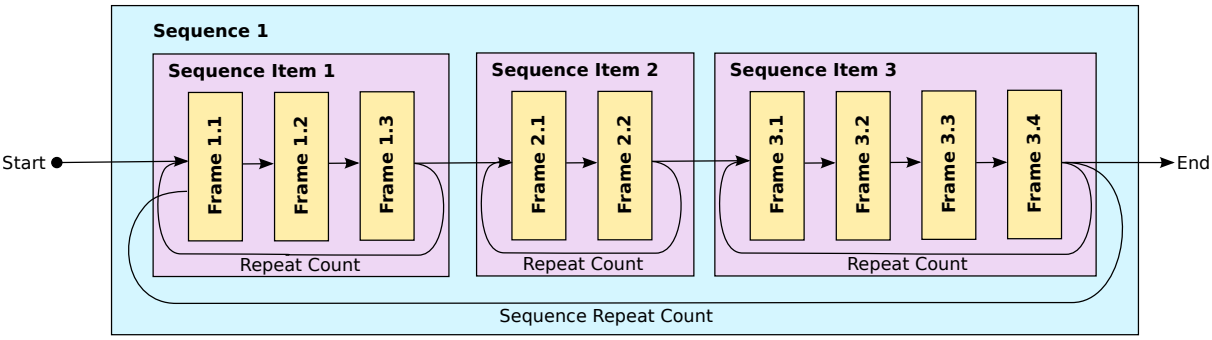

Figure 6.3: Sequence with multiple Sequence Items which have multiple Frames each.

in Sequence Item 2 are run for the number of times in the Sequence Item 2 repeat count, and finally Frames 3.1, 3.2, 3.3 and 3.4 in Sequence Item 3 are run for the Sequence Item 3 repeat count. The entire sequence may also repeat according the the Sequence 1 repeat count.

As will be described elsewhere in this manual, a common use of multiple sequence items which have multiple frames can be found in sequences of composite images on a DMD device, such as grayscale or colour images. Since a DMD is a binary device, in order to display images with greater bit depths such as 24-bit colour or 8-bit grayscale images we create sequence items where each frame corresponds to an individual bit plane of the complete image. For example, a sequence of 8-bit grayscale images will be made up of sequence items where each sequence item defines to a complete 8-bit image and within each sequence item there will be 8 frames which correspond to the 8 individual bit planes of the 8-bit image.

6.3 Creating Sequences

After we have decided on the basic structure which our sequence will follow, we are ready to start constructing the actual sequence which is composed of sequence items and frames. Creating sequences involves creating and adding sequence items to the sequence and creating and adding frames to the sequence items. The parameters of sequences, sequence items and frames can be easily modified using the Ajile GUI sequence editor or with the Ajile SDK API. A sequence verifier tool is also included with the Ajile software suite which can check over sequences and spot or even fix certain common errors which can come up when creating new seqences.

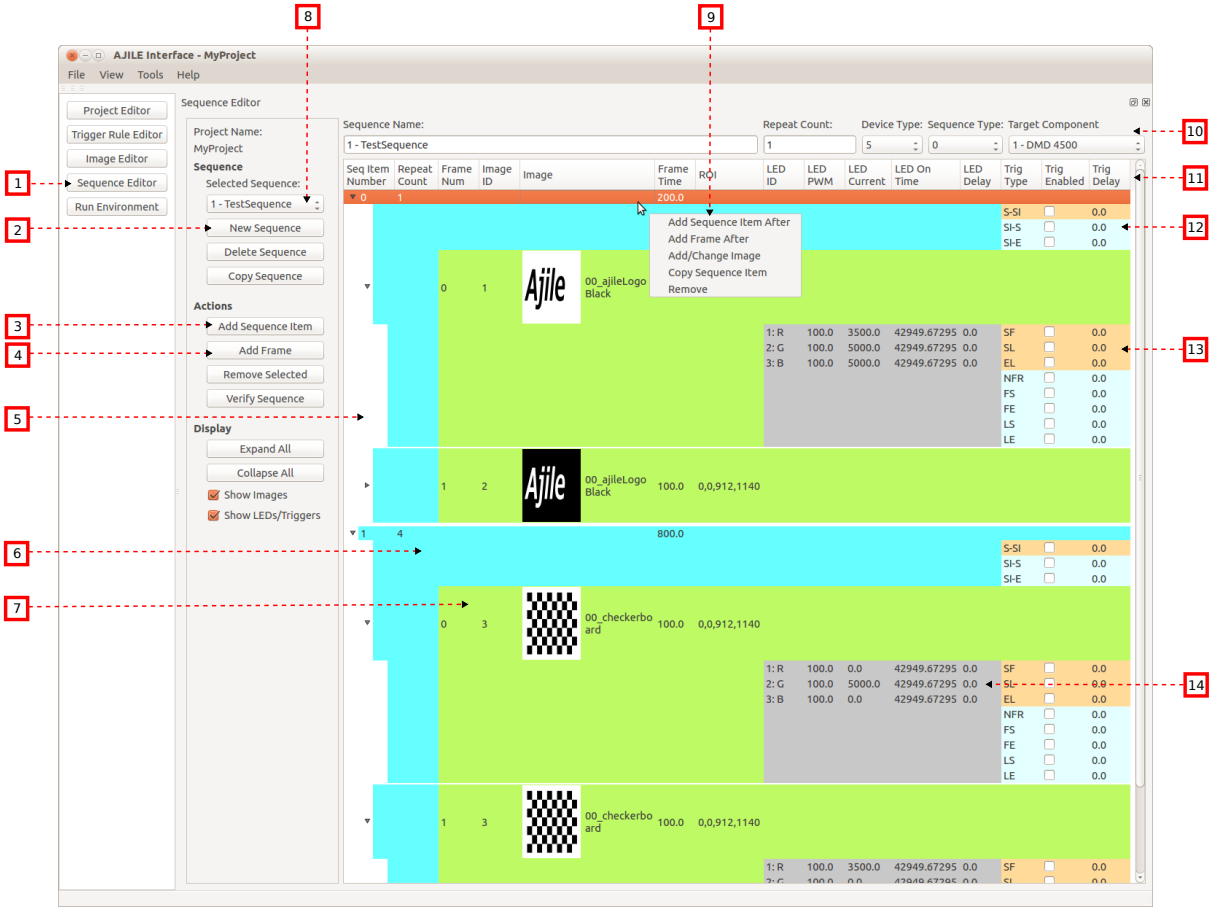

Figure 6.4: Screenshot of the Sequence Editor in the Ajile GUI where sequences, sequence items and frames can be created and edited.

6.3.1 Creating Sequences in the GUI

To create sequences in the GUI we need to open the Sequence Editor, add a new sequence, and set up its initial sequence parameters. The Sequence Editor is shown in Figure 6.4. The Sequence Editor is opened by clicking on the 'Sequence Editor' button in the GUI navigation bar (Figure 6.4, number 1). Initially the project will have no sequences. To create a new sequence, click on the 'New Sequence' button in the Sequence Editor (Figure 6.4, number 2). This causes the New Sequence dialog to appear, shown in Figure 6.5, which prompts us to enter a sequence ID, sequence name, sequence repeat count, device type that the sequence will run on, and the sequence type ('preload' or 'streaming'). With the exception of the sequence repeat count which can be changed later, these sequence parameters will be fixed for the lifetime of the project so it is important that they are initialized properly. If a mistake is made when entering these parameters the sequence will need to be deleted, by clicking the 'Delete Sequence' button beneath 'New Sequence', and starting over with another new sequence.

After entering the initial sequence parameters and pressing OK in the New Sequence dialog a new sequence is created and opened in the Sequence Editor. The sequence will automatically have a single sequence item added to it, which will be displayed as the top-most row of the sequence tree (Figure 6.4, number 5). The properties of the sequence are also displayed at the top of the Sequence Editor (Figure 6.4, number 10) and the sequence repeat count can be modified in the Repeat Count text box.

6.3.2 Creating Sequences in the SDK

The steps required for creating a new sequence and making use of it involves constructing a new Sequence object, setting any of its parameters either using the Sequence constructor or the Sequence member

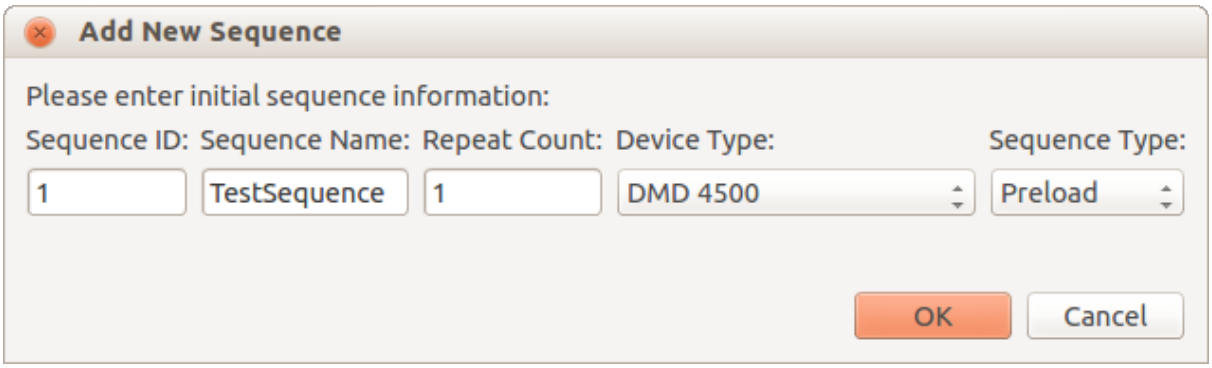

Figure 6.5: Screenshot of setting the initial Sequence parameters when creating a new sequence.

accessor functions (e.g. `Sequence.SetID()`, `Sequence.SetRepeatCount()`, etc.), then adding it to the Project using `Project.AddSequence()`. An example of this is shown in Listing 6.1 in Python and in Listing 6.2 in C++. A new Sequence object, `mySequence`, is initialized with a sequence ID of 1, with a sequence name of "My Sequence", a device type of the DMD_4500 device, and which is a 'preload' type sequence with a repeat count of 1. The sequence repeat count is later changed to 5, then the sequence is added to the project.

```
myProject = Project()
# create a new sequence with ID 1
mySequence = Sequence(1, "My Sequence", DMD_4500_DEVICE_TYPE, SEQ_TYPE_PRELOAD, 1)
print "mySequence.ID: " + str(mySequence.ID()) + ", Repeat Count: " + str(mySequence.RepeatCount())
# change the repeat count
mySequence.SetRepeatCount(5);
print "Repeat Counter after: " + str(mySequence.RepeatCount())
# add the sequence to the project
myProject.AddSequence(mySequence)
```

Listing 6.1: Python example of creating a new Sequence and adding it to the Project.

```
void CreateSequenceExample(Project myProject) {
    // create a new sequence with ID 1
    Sequence mySequence(1, "My Sequence", DMD_4500_DEVICE_TYPE,
                        SEQ_TYPE_PRELOAD, 1);
    cout << "mySequence.ID: " << mySequence.ID()
         << ", Repeat Count: " << mySequence.RepeatCount() << endl;
    // change the repeat count
    mySequence.SetRepeatCount(5);
    cout << "Repeat Counter after: " << mySequence.RepeatCount() << endl;
    // add the sequence to the project
    myProject.AddSequence(mySequence);
}
```

Listing 6.2: C++ example of creating a new Sequence and adding it to the Project.

6.4 Adding Sequence Items and Frames

Once one or more sequences have been created and added to the project, the next step is to add sequence items and frames to the sequences to make them useful.

6.4.1 Adding Sequence Items and Frames in the GUI

The list of sequence items and frames for the currently selected sequence are displayed in the sequence tree in the Sequence Editor (Figure 6.4, number 5). Note that the selected sequence can be changed by clicking on the 'Selected Sequence' drop down box (Figure 6.4, number 8). Recall that sequences, sequence items and frames are stored in a tree-like hiearchy. This tree structure can be seen in the

sequence tree where the (invisible) root of the tree is the current sequence. Beneath the sequence is the list of sequence items, shown as the blue items in the sequence tree (Figure 6.4, number 6), and beneath each sequence item is a list of frames, shown as the green items (Figure 6.4, number 7).

There are two ways to add new sequence items and frames to the current sequence. One way is to left-click on any sequence item or frame in the sequence tree to select it, then click the 'Add Sequence Item' (Figure 6.4, number 3) or 'Add Frame' (Figure 6.4, number 4) buttons. This will insert a new sequence item or a new frame into the current sequence after the currently selected sequence item or frame. The second way to add sequence items or frames is to right-click on any sequence item or frame in the sequence tree, which brings up a new pop-up menu next to the clicked item (Figure 6.4, number 9). This pop-up menu presents the same actions to add a frame or sequence item after the item which was clicked on. When a new sequence item or frame has been added to the sequence it is shown in the sequence tree and its parameters are populated with default parameters which can later be modified. Many of these default parameters such as lighting and frame times can be adjusted in the Project Editor to help speed up the process of creating sequences.

6.4.2 Adding Sequence Items and Frames in the SDK

There are several ways that sequence items and frames can be assembled and added to sequences/projects using the Ajile SDK. The example in Listing 6.3 in Python and Listing 6.4 shows the most straightforward and recommended way of adding sequence items and frames. After the sequence is created and added to the Project, the recommended approach to is to essentially add the Sequence Items and Frames in the order in which they are listed in the Sequence. In the example, a Sequence Item is created and added to the project with `Project.AddSequenceItem()`. The function `Project.AddSequenceItem()` uses the Sequence ID which is in the Sequence Item to deteremine which Sequence the Sequence Item is to be added to, then the Sequence Item is added to the end of the sequence item list for that sequence. Once the sequence item has been added, two Frames are then added in a similar manor using `Project.AddFrame()`. As with `Project.AddSequenceItem()`, `Project.AddFrame()` inspects the Sequence ID which is in the Frame to determine which sequence to add the frame to. Once the sequence with that ID was found, the Frame is then added to the end of the frame list of the last sequence item in the sequence.

While adding each new sequence item and frame to the end of the sequence is the simplest method, finer control over where sequence items and frames are added is possible by optionally supplying a sequence item 'index' as a second argument to `Project.AddSequenceItem()`, or a sequence item index and a frame 'index' as additional arguments to `Project.AddFrame()`. These index arguments insert the sequence item or frame after the item in the sequence item or frame list with the given index (note that indices start at zero). See the Ajile SDK reference manual for further details of manipulating the sequence item and frame lists.

```
myProject = Project()
# Create sequence and add it to the project
sequenceID = 1
sequence = Sequence(sequenceID)
myProject.AddSequence(sequence)
# create sequence item and add it to the project
seqItem = SequenceItem(sequenceID, 1)
myProject.AddSequenceItem(seqItem)
# create two frames and add them to the project
# (added to the last sequence item in the sequence)
frame1 = Frame(sequenceID)
myProject.AddFrame(frame1)
frame2 = Frame(sequenceID)
myProject.AddFrame(frame2)
# get the sequence from the project
foundSequence, wasFound = myProject.FindSequence(sequenceID)
print "Number of Sequence Items: " + str(len(foundSequence.SequenceItems()))
print "Number of Frames: " + str(len(foundSequence.SequenceItems()[0].Frames()))
```

Listing 6.3: Python example of creating a Sequence with one Sequence Item and two Frames and adding them to the Project.

```
void AddSequenceItemsFramesExample(Project myProject) {
    // Create sequence and add it to the project
    int sequenceID = 1;
    Sequence sequence(sequenceID);
    myProject.AddSequence(sequence);
    // create sequence item and add it to the project
    SequenceItem seqItem(sequenceID, 1);
    myProject.AddSequenceItem(seqItem);
    // create two frames and add them to the project
    // (added to the last sequence item in the sequence)
    Frame frame1(sequenceID);
    myProject.AddFrame(frame1);
    Frame frame2(sequenceID);
    myProject.AddFrame(frame2);
    // get the sequence from the project
    bool wasFound = false;
    const Sequence& foundSequence = myProject.FindSequence(sequenceID, wasFound);
    cout << "Number of Sequence Items: "
         << foundSequence.SequenceItems().size() << endl;
    cout << "Number of Frames: "
         << foundSequence.SequenceItems()[0].Frames().size() << endl;
}
```

Listing 6.4: C++ example of creating a Sequence with one Sequence Item and two Frames and adding them to the Project.

6.5 Modifying Sequence Item and Frame Parameters

Most sequences will require some amount of modification of the sequence item and/or frame parameters from the defaults. In this section we look at how to change the most fundamental sequence item and frame parameters such as repeat counts, image selection, frame time. Changing more advanced frame-by-frame parameters such as lighting and trigger settings will be covered seperately in their own chapters.

6.5.1 Modifying Sequence Item and Frame Parameters in the GUI

Once one or more sequence items and frames are added to the sequence and appear in the sequence tree of the Sequence Editor, we can modify most of the sequence item and frame parameters by clicking on the value in the sequence tree and typing in a new value. Each of the parameter names are listed in the sequence tree column headers (Figure 6.4, number 11).

Sequence Item Repeat Count

The sequence item repeat count specifies the number of times that frames in the given sequence item will be repeated. Click on the cell in the sequence item row under the repeat count column to enter a valid input number in the range from 1 to $2^{32}-1$. A special repeat count of 0 indicates that the sequence item will be repeated forever (i.e. infinitely) until the sequence is stopped or a command is sent to explicitly advance to the next sequence item. A 'Frame Time' is also displayed for the sequence item. This frame time is not editable by the user, but is for display purposes only and reports the total additive frame time of all frames within the sequence item, times the sequence item repeat count. For example, a sequence item with a repeat count of 5 which contains 3 frames of 100 ms each will have a 'Frame Time' displayed as 5 repeats $\times$ 3 frames $\times$ 100 ms = 500 ms.

Images

Each frame refers to an image by its unique image ID in the project. There are two ways to set the image ID of a given frame. The first is to click on the cell in the frame row under the Image ID column and explicitly enter in a valid Image ID number which was previously added in the Image Editor. An error message will appear if the entered Image ID does not refer to a valid image in the project. The second way to set the image in the frame is to right-click the frame row in the sequence tree to bring up the pop-up menu which we have already seen (Figure 6.4, number 9) and clicking on the "Add/Change

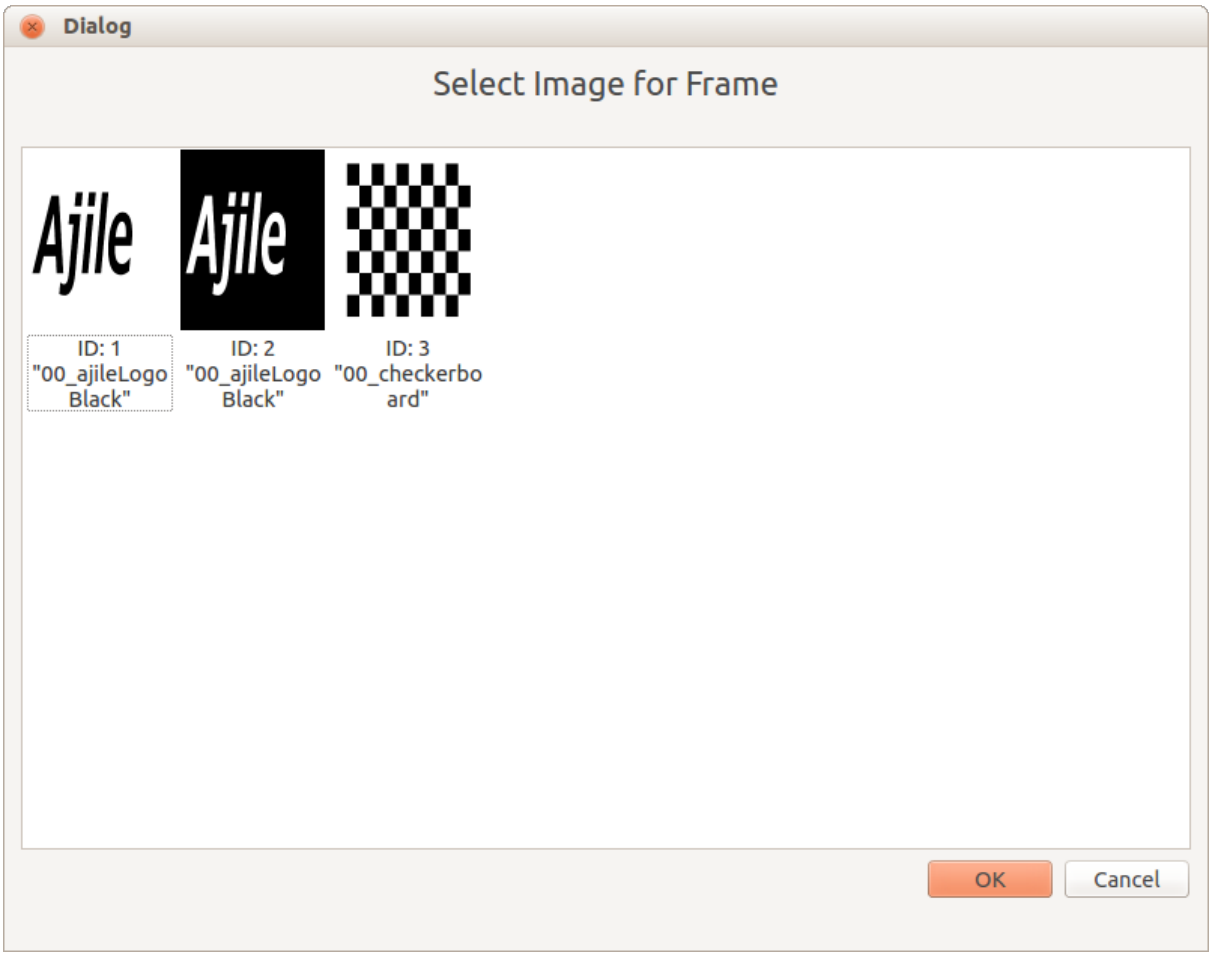

Figure 6.6: Dialog which allows setting the image for the frame.

Image" item in the menu. This brings up a Select Image Dialog, seen in Figure 6.6, which shows all valid images which can be used for the given frame in the sequence. Left clicking on the chosen image and clicking OK or double clicking the image will change the selected image for the frame.

It is also possible to set an image for a sequence item for the case of composite images such as 24-bit RGB or 8-bit grayscale image which are composed of 24 or 8 individual 1-bit frames. Setting a composite image for a sequence item (by clicking on "Add/Change Image" in the pop-up after right-clicking on the sequence item) results in 24 (or 8) frames being automatically created and added to the sequence item for each of the corresponding bitplanes of the composite image. The Image IDs of each frame along with the frame timing and lighting parameters are also automatically set to produce a sensible RGB colour (or grayscale) image. The individual parameters of each frame can of course still be modified to enable custom colour or grayscale lookup tables, which users can explore further.

Frame Time

Setting the frame time of an individual frame is accomplished by clicking on the cell in the frame row under the Frame Time column and entering a valid frame time. Valid frame times, particularly minimum values, are device specific and may depend on other imaging parameters as the selected region of interest. Note that the time units of the frame time depends on the GUI preference, which can be changed to suit the needs of the sequence timescale by opening the top menu item under Tools ⇒ Preferences. Milliseconds and microseconds are typical time units, however the lowest level of raw clock ticks (where each count corresponds to 10 nanoseconds) gives the finest granularity of precision if needed. As will be discussed in Chapter 7, it is important to also consider the On Time of the lighting to make sure that it

is changed along with the frame time as needed.

6.5.2 Modifying Sequence Item and Frame Parameters in the SDK

Sequence item and frame parameters must be set before they are added to the project in the SDK. If parameters need to be modified after being added then the modified sequence items or frames must be removed and re-added to the project with the new parameters.

Any sequence item or frame parameters which were listed in Table 6.2 and Table 6.3 can be set either by defining them initially in their constructors, or by using the accessor functions to modify them (e.g. `SetRepeatCount`, `SetFrameTime`, etc.) For full details of the constructor parameters and the list of accessor functions for the Sequence Item and Frame objects one can refer to the Ajile SDK reference manual.

6.6 Verifying Sequences

Users have a great deal of flexibility when creating sequences when using the Ajile software suite. While this flexibility allows a great deal of power for solving the widest possible range of problems, it can also potentially make it tricky for beginning users ensure that everything is set up correctly. To help with identifying and solving any potential issues with sequences, the Ajile suite has included with it a **sequence verifier** which checks over created sequences and can even automatically fix certain issues. The sequence verifier should be run on a sequence against a target component once it has been finished by the user. This exact same sequence verifier which is available to the user in the GUI/SDK is in fact also running on all Ajile devices, and so even if one forgoes running the verifier it will still be run internally by the target device to ensure that any sequence being run on it will not cause problems for the device.

6.6.1 Verifying Sequences in the GUI

Fortunately for the most part when creating sequences in the Ajile GUI, most potential sequence problems are prevented from occuring right up front while inputting in the various parameters. The sequence verifier is however available to spot any remaining issues. To run the sequence verifier in the GUI, first select the sequence to be verified then click on the "Verify Sequence" button in the Sequence Editor which is in the "Actions" set of buttons. This presents the user with the Sequence Verifier dialog box, shown in Figure 6.7. By clicking on the "Verify Only" button, the selected sequence is verified against the target component and a list of any errors or warnings are printed in the dialog. Warnings are printed in orange and are preceded by (WW) and errors are printed in red and are preceded by (EE). Warning are non-critical and can be fixed by clicking on the "Verify & Fix" button. Any unfixed warnings will be automatically fixed when they are run on the target device regardless. Errors on the other hand are deemed critical and cannot be fixed automatically. The example in Figure 6.7 shows one critical error - one of the frames in the sequence does not have a valid Image ID assigned, and so the sequence will not be able to run. One will need to manually fix any errors found in the verifier - the fix in this example would be to select a valid image for the frame in question.

6.6.2 Verifying Sequences in the SDK

The sequence verifier is most useful when used within the SDK environment since users are fairly free to create sequences however they want up front without limitations. As already mentioned, the same sequence verifier is used by the GUI, SDK and Ajile devices. As in the GUI, the SDK sequence verifier also has the ability to either only verify a sequence, or to verify and fix a sequence (thus modifying its contents). Verifying a sequence is accomplished with a Project function `Project.VerifySequence()` which takes as arguments a sequence ID which is in the project, a component ID (i.e. index) which is in the project, a flag which indicates whether or not the fix the sequence and finally a string which receives the human readable error output from the verifier. The example in Listing 6.5 in Python and in Listing 6.6 in C++ shows how to use the sequence verifier, where it doesn't modify the sequence in the first call to `Project.VerifySequence()` and fixes (and thus modifies) the sequence in the second call. Note

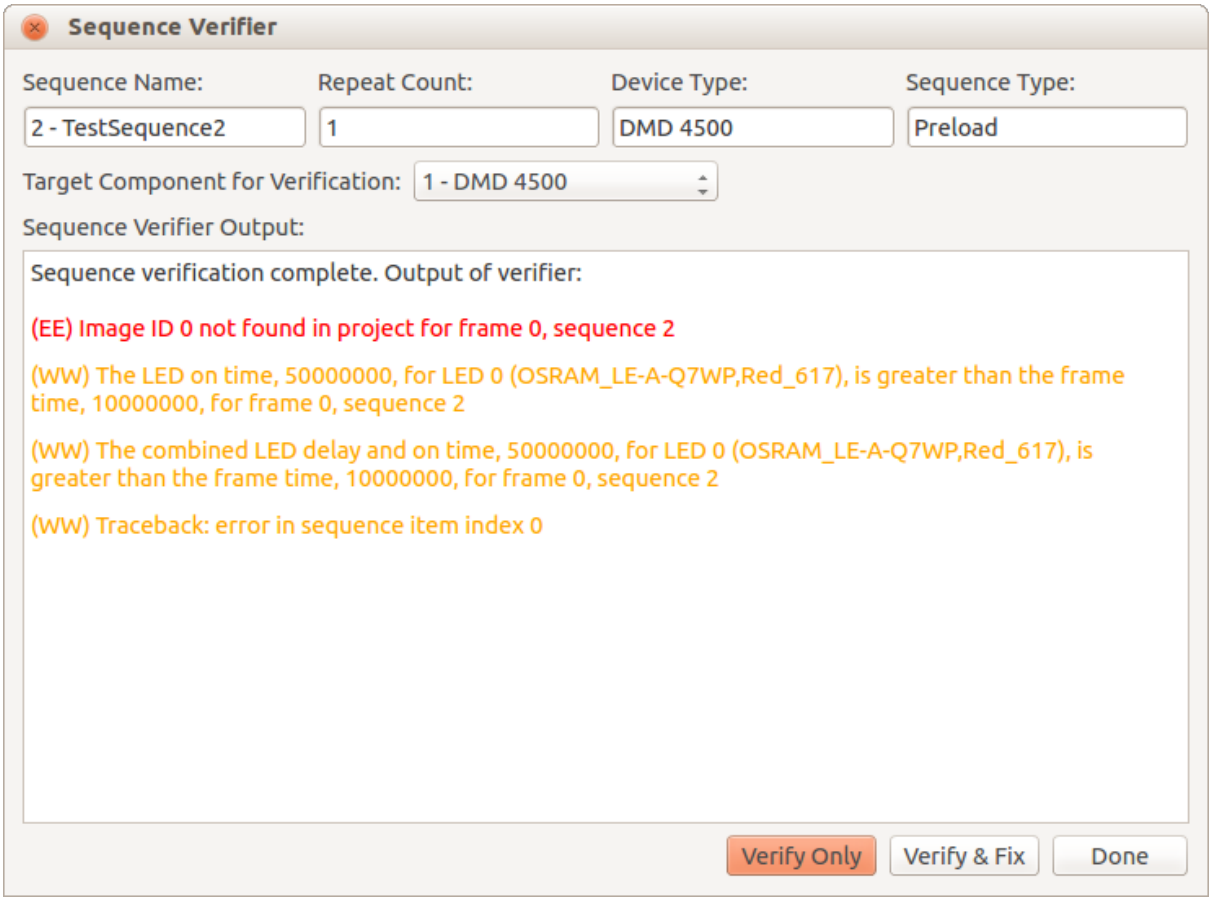

Figure 6.7: Screenshot of the Sequence Verifier dialog.

that the Python version of the sequence verifier call is actually `Project.VerifySequenceStr()` instead of `Project.VerifySequence()`, but it behaves identically to the C++ version. Based on the sequence verifier output, users can either print and analyze the verifier output and manually fix any problems, use the verifier to fix warnings automatically, and/or develop custom functions to fix any remaining sequence errors.

```
# verify the sequence only (without fixing it)
errorString = myProject.VerifySequenceStr(sequenceID, dmdComponentID, False)
print "Verification complete without fix, output is: " + errorString
# verify the sequence and fix it
errorString = myProject.VerifySequenceStr(sequenceID, dmdComponentID, True)
print "Verification complete with fix, output is: " + errorString
```

Listing 6.5: Python example of verifying a Sequence in a Project against a DMD target component.

```
void VerifySequenceExample(Project myProject, u16 sequenceID, u8 dmdComponentID) {
    // verify the sequence only (without fixing it)
    string errorString = "";
    ErrorType_e ret = myProject.VerifySequence(sequenceID, dmdComponentID, false, &errorString);
    cout << "Verification complete without fix, error code is "
         << ret << " output is: " << endl << errorString;
    // verify the sequence and fix it
    errorString.clear();
    ret = myProject.VerifySequence(sequenceID, dmdComponentID, true, &errorString);
    cout << "Verification complete with fix, error code is "
         << ret << " output is: " << endl << errorString;
}
```

Listing 6.6: C++ example of verifying a Sequence in a Project against a DMD target component.

Chapter 7

Lighting

7.1 Lighting Introduction

7.1.1 Lighting Controller Overview

A high performance lighting controller is available with the Ajile suite. This multi-channel lighting controller can be combined with other devices, such as the DMD controller or camera controller, to provide precisely controlled lighting on a frame-by-frame basis.

The primary use of the lighting controller is as a 3 channel LED controller for a DMD projector optical engine. For example, the standard Ajile DMD 4500 projector has in it three high powered OSRAM LEDs in each the red, green and blue.

The lighting controller is however not limited to just DMD lighting control. Ajile cameras can also have a lighting controller attached to provide, for example, frame-by-frame multispectral scene lighting with multiple LED channels. In addition, multiple lighting controllers can be daisy chained together so that more than three LED channels can be controlled from a single master device (e.g. a DMD controller or camera controller). The standard offering allows for 2 lighting controllers to be chained together to provide 6 LED channels for one device (3 channels per board), but for specific applications up to 8 lighting controllers may be used together to provide 24 LED channels if needed.

Maximum output current and voltage ratings per channel for the lighting controller can be found in the specific data sheets. The current implementation allows for around 85 W of electrical output across 3 channels. Of course, with such high powered operation of LEDs high temperatures can also be an issue, both with the LEDs themselves or with the actual lighting controller board. Proper heat sinking and air flow is thus needed to allow operation at maximum power. Furthermore, to ensure the safety of the attached LEDs and the lighting controller electronics, the controller is equipped with an on-board thermocouple and has inputs for 3 additional thermocouples, one per lighting channel, which can monitor temperatures. The thermocouple outputs can be monitored by the attached master device and individual LEDs or the entire lighting controller can be shut down when certain critical temperatures are reached to protect the hardware.

7.1.2 Lighting Control Software Overview

There are two main places where users interact with lighting in the Ajile software suite. The first is the list of LED properties which are contained in the components within a project. The second are the LED settings in each individual frame in the sequences within a project.

The list of LED properties in each component first of all tells us how many LEDs are in available in the component by looking at the length of the LED properties list. Each individual LED property then tells us about the minimum, maximum and default LED settings such as the LED currents, as well as descriptive text strings which identify the LED types and colours/wavelengths.

Name	Description
Device Name	A human readable text string which describes the specific LED part. This typically relates to the manufacturer part number of the device, for example OSRAM_LE-A-Q7WP defines the OSRAM Red LED which is found in standard Ajile projector units.
Colour	A human readable text string which describes the colour and/or centre wavelength of the LED. For example, Red_617 defines a red LED with centre wavelength at 617 nm (e.g. the OSRAM_LE-A-Q7WP).
Maximum Settings	The maximum drive settings for the LED. This is stored as an LED Setting (and therefore holds the maximum current, PWM, on time and delay for the LED).
Minimum Settings	The minimum drive settings for the LED. This is stored as an LED Setting. Frames that have LED Settings which are outside of the range of the maximum and minimum settings are not allowed, and are automatically clamped to the minimum or maximum settings when evaluated by the sequence verifier.
Default Settings	The default drive settings for the LED. This is stored as an LED Setting. Frames that do not specify all LED Settings will automatically be given the default settings when evaluated by the sequence verifier.
Warning Temperature	A temperature, in degrees Celcius, which is still safe for the LED device but should ideally not be exceeded for long periods of time.
Critical Temperature	A temperature, in degrees Celcius, which should never be exceeded by the LED device. The controlling device may automatically shut down the LED to protect it if this temperature is exceeded since it usually indicates a hardware problem with the electronics or the system heat conduction.

Table 7.1: Description of members that are in an LED Property.

The list of LED settings in each frame indicate the LED settings which will be used for that specific frame. This includes the per-channel LED current, LED pulse-width modulation (PWM) percentage, LED on time and LED delay time. If any or all of the LED settings are not included with the frame then the default LED settings from the component will be used.

7.2 Lighting Members

Within a project, Frames have a list of LED Settings while Components have a list of LED Properties (which in turn have LED Settings for the LED minimum, maximum and default values). This structure was previously seen in Figure 2.1.

7.2.1 LED Property Members

LED Properties in components store the minimum, maximum and default LED settings along with, a description of the LEDs, and warning/critical temperatures which will be monitored to protect the LEDs. The list of members of an LED Property are given in Table 7.1.

7.2.2 LED Setting Members

A list of LED Settings are stored in each Frame to specify the per-channel LED current, PWM, on time, and delay time for the frame. LED Settings are also used by the LED Properties to store the minimum, maximum and default LED Settings per LED. The list of members of an LED Setting are given in Table 7.2.

Name	Description
Current	The current of the LED, which controls the amount of electrical power delivered to the LED. Internally this value is specified in raw counts which will be used by LED controller to control the LED channel. However, convenience functions are provided to specify current in amps (A) or milliamps (mA) and the conversion to raw counts is done internally be the Ajile SDK.
Pulse-Width Modulation (PWM)	The PWM percentage of the LED. A default PWM period is used by the lighting controller, and so the PWM value specifies the duty cycle within that PWM period which the LED is on for. Internally this is specified as an 8-bit value with 0 being a PWM of 0%, 255 a PWM of 100% and 127 a PWM of $\sim$ 50%. However, convenience functions are available to specify PWM values as percentages with conversion to raw PWM values done internally by the Ajile SDK.
On Time	The amount of time during the frame that the LED will be active for. LED on time can be less than or equal to the frame time. On times longer than the frame time will be clamped to the frame time by the sequence verifier when run.
Delay Time	A delay time, which is measured from the start of the frame, after which the LED will be activated.

Table 7.2: Description of members that are in an LED Setting.

7.3 LED Settings Detailed Description

Since there are a number of different LED Setting parameters that be changed per frame, here we take a closer look at how their values change the effect of the LEDs by way of example.

The example in Figure 7.1 shows the behavior of three LEDs, red, green and blue, over the time period of two frames in a sequence. As we saw in Chapter 6, Frames have a frame time which defines how long the frame is exposed for, and has a list of LED settings, one for each channel. In the example, each of the three channels have different LED Settings including LED current, PWM percentage, LED on time, and LED delay time, and these LED Settings are varied between the two frames shown.

LED Current:

The LED current controls the brightness of the LED for the duration of the frame. The Ajile lighting controller has high speed control of LED current so that new current settings for a frame (which may be vastly different from the previous frame) take effect almost immediately. In Figure 7.1, we see that the red LED current remains a constant 1.0 over both frames. The green LED has a current value of 1.5 in the first frame while in the second frame its current drops to half its value to 0.75. We can observe this drop in current represented as a drop in green LED output amplitude (which will correspond to brightness of the actual phsysical LED). Similarly, the blue LED current setting drops from 1.25 to 1.0 over the two frames.

Note that the current in this example is of arbitrary units but is eventually mapped to amperes (A). Time units are also arbirary in Figure 7.1, but can easily be mapped to seconds, milliseconds, etc.

LED PWM:

The LED PWM is expressed as a percentage of the LED on time over a fixed repeating PWM period. In Figure 7.1 we see that the PWM period is $\frac{1}{8^{\text{th}}}$ the frame time of the first frame since there are 8 individual PWM periods seen for the red LED which has a PWM of 50%, and its on time is for the full frame time. Frame 1 has a frame time of 1000 time units, therefore the PWM period in this example is $\frac{1000}{8} = 125$ time units. With a PWM percentage of 50% for the red LED in Frame 1, we therefore see that for half of each PWM period the red LED is emitting light, and for the other half of each 125 time unit length PWM period the LED is not outputting light. In Frame 2, the red LED has a PWM of 100%, meaning that the red LED will not be pulse-width modulated and so is on for the entire duration

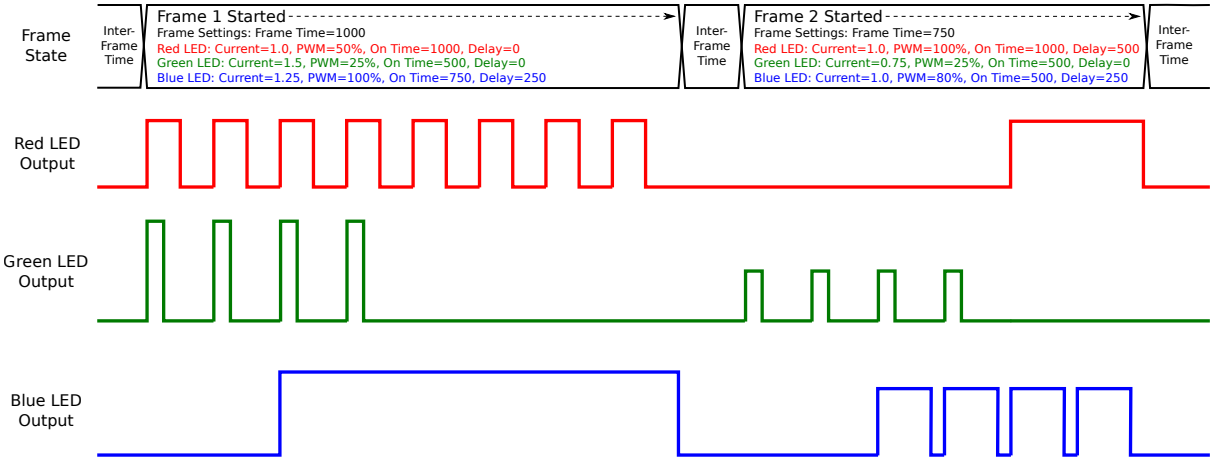

Figure 7.1: 3-channel LED output over two frames, with different LED Settings per channel and per frame.

of each PWM period. In the green LED channel we have a PWM of 25% for both frames, therefore the green LED is emitting light for a quarter of each 125 time unit PWM period. The blue LED on the other hand again has a 100% PWM for Frame 1 and so is continuously on, while it has a PWM of 80% for Frame 2 and therefore is mostly on during each PWM period.

In this example the PWM period is arbitrary. In actual hardware the PWM period is typically $\frac{1}{100\text{kHz}} = 10$ microseconds. Note that this 10 μs PWM period works well for frame times which are of a comparitively much longer timescale than the PWM period (e.g. 1 ms or longer). For very short frame times which are of microsecond time scales (which are completely possible with Ajile devices), it is *highly* recommended to *not use* the PWM setting and to *always* leave it at 100%. Instead, the LED current, LED on time, and delay time can be used to control the LED light output in order to obtain predictable results at these time scales.

LED On Time:

LED on time is the amount of time that the LED will be left on from the time that it is first enabled during a frame. When an LED delay setting is not used (i.e. the delay is zero), then the LED will be turned on immediately at the start of the frame. In Figure 7.1, a short inter-frame time is shown in the frame state timeline to indicate the gap in between adjacent frames. After this inter-frame time, the frame begins. When no delay time is set, the LED is turned on immediately at this frame begin event. From the initial turning on event of the LED, the LED will remain enabled for the LED on time. In the example, the red LED on time is set to the frame time of 1000 time units for Frame 1 and so is on for the entire frame. The green LED however has on on time of 500 time units for Frame 1 which is half the total frame time, therefore the green LED is enabled for only the first half of the frame. The blue LED has an on time of 750 units for Frame 1 and so is on for 3/4 of the frame time. The blue LED also has a delay time set and so is not immediately enabled when the frame starts; delays will be discussed next.

The given example on LED on times and delays holds for frames which are based on internal frame timing. It is also possible to use external trigger signals to control the state of the LEDs, which is described in detail in Chapter 8.

LED Delay:

The last LED Setting which is available under user control is the LED delay time. The LED delay allows us to specify the amount of time after the lighting start event when the LEDs are meant to be enabled. When using internal frame timing (i.e. not using triggers to control the lighting) this lighting start event is the start of the frame. The LED delay therefore causes the LED to be enabled after the start of the frame, at the time specified by the delay value. When external triggering is used, the LED delay allows for the LEDs to be enabled after a delay time has elapsed from the time the external trigger event is

observed (which is described further in Chapter 8).

In the examle of Figure 7.1, the red and green LEDs have the delays set to the default of zero for Frame 1, and therefore are enabled immediately at the start of the frame. The blue LED on the other hand has a delay of 250 time units and so is enabled 250 units after the start of the frame. For Frame 2, the green and blue LED channels use the same delay values of 0 and 250 time units, respectively. The red channel on Frame 2 now has a delay of 500 time units specified and so it is enabled 500 after the beginning of the frame.

Note that in Figure 7.1, the red LED settings specifiy a delay of 500 and an on time of 1000, which would mean that the LED would actually be on for longer than the total frame time of 750 time units. This is not allowed by sequences. As a rule the lighting for each frame is independant of the last. The sequence verifier therefore will detect such cases and will clamp the total LED time, defined as the total on time + delay time, to be the frame time. Also it should be noted that delay times greater than the frame time result in the LED never being enabled for the frame. These cases will be flagged as warnings by the sequence verifier.

7.4 Configuring Component LED Properties

The starting point of working with LED settings in Ajile projects is with the list of LED Properties which are in each component. These LED properties define the overall types of LEDs and their characteristics.

7.4.1 Configuring LED Properties in the GUI

Viewing LED Properties which are in the project components and configuring their default values is done via the Ajile GUI Project Editor. The Project Editor was introduced in Chapter 3, which we show in greater detail in Figure 7.2. With the Project Editor open, we first need to select the component which has the LED properties that we need to view or edit (Figure 7.2, number 1). Provided that the selected component actually has LEDs, the LED table will be populated with the current LED Properties for the component. In Figure 7.2, the DMD 4500 component is selected and in this case has three LEDs associated with it. The LED colours and device descriptions can be seen in the LED table (Figure 7.2, number 5), as well as the LED minimum and maximum current values (Figure 7.2, number 6). The LED colour, device, minimum and maximum are read-only since they are constraints derived directly from the component and device. The default LED settings are also in the LED table (Figure 7.2, number 7), which include the default current, PWM, on time and delay values. The default values *can* be edited by the user. These default values will be used by the Ajile GUI when creating frames in the sequence editor to automatically populate newly created frames with the default LED settings.

Note that the units displayed in the GUI for currents and times depend on the chosen GUI preference. To change the GUI preferences, select 'Preferences' under the 'Tools' menu (Figure 7.2, number 4). This presents a Preferences dialog (Figure 7.2, number 2) from which the user can choose the desired time and current units to best suit the project needs.

7.4.2 Configuring LED Properties in the SDK

A list of LED Property objects is contained in each Component in the Project. The LED Properties of an individual Component can be accessed by `Component.LedProperties()`. From this returned list we can see how many LEDs belong to the component from the list size, and see each device description, colour, and minimum/maximum/default settings. As with the GUI, most of the LED Properties of a component are read only, with the exception being the default LED Settings. To easily update the default LED Settings, the project function `Project.SetDefaultLedSetting()` is available. The function `Project.SetDefaultLedSetting()` takes as arguments the component index where the LED resides and the index of the individual LED, as well as an LedSetting object which contains the new default LED values.

An example of configuring LED Properties is given in Listing 7.1 in Python and in Listing 7.2 in C++.

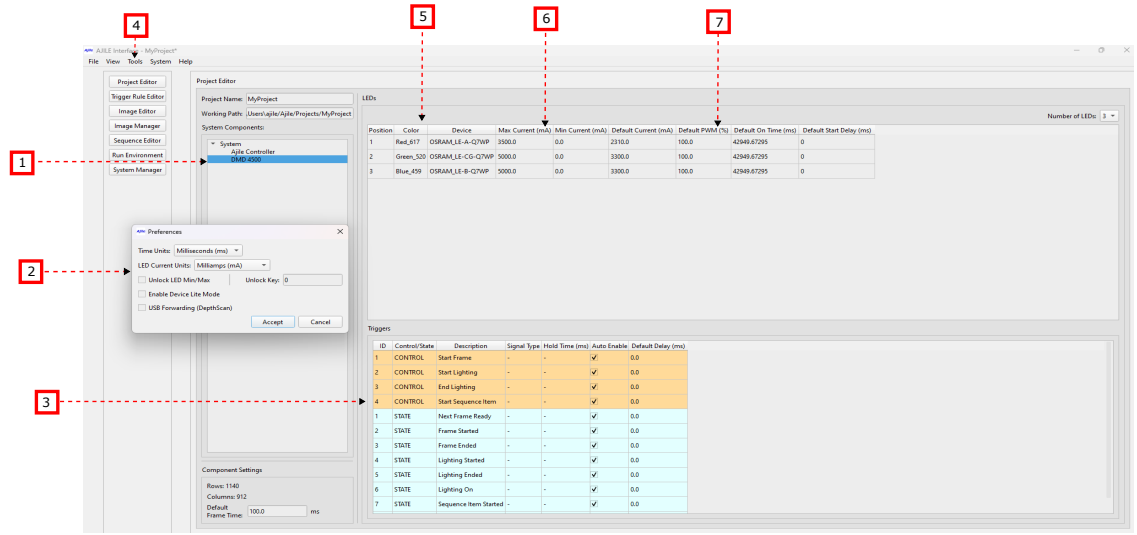

Figure 7.2: Screenshot of the Project Editor, which allows for component settings including LED and Trigger settings to be viewed and configured.

The list of LED Properties are iterated through and the descriptions and colours are output. Three new LEDSetting objects are then created to be used as default values for the red, green and blue LED channels. The values of the defaults are identical to those seen in the example of Figure 7.1. Note that the current units (the first argument of the constructor) is specified in milliamps, and the on time and delay times are given in milliseconds. Internally, all times are stored in 10 ns units, therefore a helper function `FromMSec` is used to easily convert from ms values to internal 10·ns units. Once the default LED Settings have been created, the defaults of the component can be updated using `Project.SetDefaultLedSetting()` for each of the three LED Settings. Finally, the updated default settings of the component are output. Note that the LED time accessor functions have the suffix `MSec()`, for example `LedSetting.OnTimeMSec()`. This convenience function performs the conversion of internal 10·ns units to ms. Other conversion functions to/from ns, μs, and seconds are also available.

```
# output the list of LED colours/names for the component
for ledProperty in myProject.Components()[componentIndex].LedProperties():
    print "LED Colour: " + ledProperty.Colour() + ", Device: " + ledProperty.DeviceName()
# create new default LED settings
newDefaultRed = LedSetting(1000, 50, FromMSec(50), 0)
newDefaultGreen = LedSetting(1500, 25, FromMSec(50), 0)
newDefaultBlue = LedSetting(1250, 100, FromMSec(75), FromMSec(25))
# update the default LED settings in the project
myProject.SetDefaultLedSetting(componentIndex, 0, newDefaultRed)
myProject.SetDefaultLedSetting(componentIndex, 1, newDefaultGreen)
myProject.SetDefaultLedSetting(componentIndex, 2, newDefaultBlue)
# output the LED defaults
for ledProperty in myProject.Components()[componentIndex].LedProperties():
    defaultSetting = ledProperty.DefaultSettings()
    print "Current: " + str(defaultSetting.Current())
    print "PWM: " + str(defaultSetting.PWM())
    print "On Time: " + str(defaultSetting.OnTimeMSec())
    print "Delay: " + str(defaultSetting.DelayMSec())
```

Listing 7.1: Python example of reading and setting LED Properties in the Project.

```
void LEDPropertiesExample(Project myProject, u8 componentIndex) {
    // output the list of LED colours/names for the component
    for (u8 i=0; i<myProject.Components()[componentIndex].LedProperties().size(); i++) {
        const LedProperty& ledProperty =
            myProject.Components()[componentIndex].LedProperties()[i];
        cout << "LED Colour: " << ledProperty.Color()
             << ", Device: " << ledProperty.DeviceName() << endl;
    }
    // create new default LED settings
    LedSetting newDefaultRed(1000, 50, FromMSec(50), 0);
    LedSetting newDefaultGreen(1500, 25, FromMSec(50), 0);
    LedSetting newDefaultBlue(1250, 100, FromMSec(75), FromMSec(25));
    // update the default LED settings in the project
    myProject.SetDefaultLedSetting(componentIndex, 0, newDefaultRed);
    myProject.SetDefaultLedSetting(componentIndex, 1, newDefaultGreen);
    myProject.SetDefaultLedSetting(componentIndex, 2, newDefaultBlue);
    // output the LED defaults
    for (u8 i=0; i<myProject.Components()[componentIndex].LedProperties().size(); i++) {
        const LedSetting& defaultSetting =
            myProject.Components()[componentIndex].LedProperties()[i].DefaultSettings();
        cout << "Current: " << defaultSetting.Current() << endl
             << "PWM: " << defaultSetting.PWM() << endl
             << "On Time: " << defaultSetting.OnTimeMSec() << endl
             << "Delay: " << defaultSetting.DelayMSec() << endl;
    }
}
```

Listing 7.2: C++ example of reading and setting LED Properties in the Project.

7.5 Configuring LED Settings per Frame

A unique feature of the Ajile suite is the ability to control any LED setting on a frame-by-frame basis. Each Frame has a list of LED Settings, one for each physical LED, which can be used to set the LED settings which will be used for that individual frame.

7.5.1 Configuring LED Settings in the GUI

Configuring the per frame LED settings in the GUI is accomplished using the Sequence Editor which was introduced in Chapter 6. The LED Settings for an individual frame can be found in the sequence tree beneath the given frame (Figure 6.4, number 14). When new frames are created in the Sequence Editor they are automatically populated with a list of LED Settings using the default settings of the Target Component (Figure 6.4, number 10). These default LED Settings for the Frame could of course be left alone, or they could be customized per frame by clicking on the LED setting to be modified and inputting a new value. The units for current and time in the Sequence Editor follow the units selected in the GUI preferences. It is also possible to collapse or expand the LED settings for frames in the sequence tree by clicking on the arrow icon next to the frame.

7.5.2 Configuring LED Settings in the SDK

We have already seen LEDSetting objects when configuring the LED defaults for a component. Configuring LED Settings for a frame is similar. We create new LED Settings, then add or update them to the frame which they belong to.

Unlike in the GUI, Frames created in the SDK do not automatically have their list of LED Settings initialized to the defaults in the target component since frames do not explicity know which component they are designed for when they are created. However, frames with unspecified or missing LED Settings have their LEDs defaulted to the default LED settings of the target component by the sequence verifier when they are run on the component.

The recommended way to configure the LED Settings for a frame is to first create a list of LED Settings by making a copy of the default LED Settings from the component, then modify the copied LED settings to the customized values per frame. This way we can be sure that the list of LED Settings is created

properly for the target component and we only need to alter the values of interest. Once the LED Settings have been changed in the desired way, the LED Settings can assigned to the frame in which they will be used with the function `Frame.SetLedSettings()`.

An example of configuring per frame LED settings is given in Listing 7.3 in Python and in Listing 7.4 in C++. A new list of LED Settings is created by copying over the default LED Settings from the list LED Properties in the Component. The new list of LED Settings is then modified by using the the `LedSetting.SetCurrent()` and `LedSetting.SetOnTime()` functions. A new frame is then created and the LED Settings for the frame are set to our new list of LED Settings. Finally, the frame is added to the project and we output the list of LED settings.

```
# start with list of default LED settings from the component
ledSettings = LedSettingList()
for ledProperty in myProject.Components()[componentIndex].LedProperties():
    ledSettings.append(ledProperty.DefaultSettings())
# update the current of LED 0 to 2 A
ledSettings[0].SetCurrent(2000)
# update the on time of LED 1 to 500 microseconds
ledSettings[1].SetOnTimeUSec(500)
# update the on time of LED 2 to 0, disabling it
ledSettings[2].SetOnTime(0)
# create a new frame
myFrame = Frame(1)
# set the LED settings for the frame
myFrame.SetLedSettings(ledSettings)
# add the frame to the project
myProject.AddFrame(myFrame)
# output the LED settings
for ledSetting in myFrame.LedSettings():
    print "Current: " + str(ledSetting.Current())
    print "PWM: " + str(ledSetting.PWM())
    print "On Time: " + str(ledSetting.OnTimeMSec())
    print "Delay: " + str(ledSetting.DelayMSec())
```

Listing 7.3: Python example of changing the LED Settings of a Frame by modifying the defaults from the Component.

```
void LEDSettingsExample(Project myProject, u8 componentIndex) {
    // start with list of default LED settings from the component
    vector<LedSetting> ledSettings;
    const vector<LedProperty>& ledProperties =
        myProject.Components()[componentIndex].LedProperties();
    for (u8 i=0; i<ledProperties.size(); i++)
        ledSettings.push_back(ledProperties[i].DefaultSettings());
    // update the current of LED 0 to 2 A
    ledSettings[0].SetCurrent(2000);
    // update the on time of LED 1 to 500 microseconds
    ledSettings[1].SetOnTimeUSec(500);
    // update the on time of LED 2 to 0, disabling it
    ledSettings[2].SetOnTime(0);
    // create a new frame
    Frame myFrame(1);
    // set the LED settings for the frame
    myFrame.SetLedSettings(ledSettings);
    // add the frame to the project
    myProject.AddFrame(myFrame);
    // output the LED settings
    for (u8 i=0; i<myFrame.LedSettings().size(); i++)
        cout << "Current: " << myFrame.LedSettings()[i].Current() << endl
             << "PWM: " << myFrame.LedSettings()[i].PWM() << endl
             << "On Time: " << myFrame.LedSettings()[i].OnTimeMSec() << endl
             << "Delay: " << myFrame.LedSettings()[i].DelayMSec() << endl;
}
```

Listing 7.4: C++ example of changing the LED Settings of a Frame by modifying the defaults from the Component.

Chapter 8

Triggers

One of the main advantages that industrial imaging devices such as those in the Ajile suite have over standard imaging devices is the ability tightly syncrhonise devices to each other and to external devices. This syncrhonisation and control across multiple devices is accomplished through triggering, where signals with precise timing are emitted from one device and received by the other device. The signal which is being emitted is called the ***output trigger*** signal, and it is output by the ***master*** timing device. The signal which is received is called the ***input trigger*** signal, and it is an input to the ***slave*** device. The slave device which receives the input trigger then uses the timing of the signal to syncrhonize its imaging with the master device.

Triggering is often one of the more difficult concepts to set up when working with imaging devices since it deals with precise control with hardware devices. The Ajile suite tries to make triggering easier by offering graphical tools to set up triggers and to simulate their timing in the GUI, and by having a consistent object oriented interface in the SDK to set up triggers between devices.

Components in the Ajile suite can output their current state information with signals called ***device state outputs***. For example when a DMD starts displaying a frame, a device state output, called `FRAME_STARTED`, will be output from the component. These device states are output with very low latency (around 10 ns) which other devices can use to synchronize with. Components also have inputs which can accept external signals, called ***device control inputs***, which when observed cause certain actions to occur in the component. For example, a DMD (or camera) device can be configured to only started displaying (or exposing) the next frame when a `START_FRAME` device control signal is observed by the component.

Components such as the Ajile Controller also have external input and output triggers which correspond to hardware pins which can be used to physically connect and synchronize Ajile devices to external hardware. These external triggers can be configured by the software to be active high or active low, to accept rising or falling edge trigger signals, and to have a programmable trigger hold time to allow interfaces with devices on any time scale.

Ajile projects have within them a list of 'trigger rules' which can be created by the user. Trigger rules connect the state of one or more device state outputs or input triggers to a device control input or output trigger. For example, a trigger rule can connect the output of the `FRAME_STARTED` signal of a DMD to an external output trigger. Another trigger rule could be created alongside the first to connect the DMD `FRAME_STARTED` device state *and* an external input trigger to the `START_FRAME` device control input of a camera component. The list of trigger rules allows for a great deal of flexibility to achieve a high degree of synchronization. The trigger rules, once loaded into the project, are evaluated by a Trigger Rules Engine which runs on FPGA hardware. Each trigger rule is evaluated in around 10 ns, so that very low triggering latency can be achieved with the trigger rules list.

8.1 Device State Outputs

The device state outputs expose the internal running state of components to the external world so that external devices can synchronize with those states. As was seen in Chapter 4, each component has a list of device states which indicate what types of device state outputs that the component makes available. The list of possible of device states are described in Table 8.1.

8.2 Device Control Inputs

The device control inputs allow internal component actions to take place synchronously with signals observed from the outside world. This allows for Ajile components to coordinate with external devices by synchronizing with the control input signals. Each component has a list of device controls which indicate what types of device control inputs that the component makes available. The list of possible of device controls are described in Table 8.2. Note however that the described behavior only happens if the given device control is enabled for a given frame, which will be described in section 8.8.

8.3 Trigger Members

There are three types of objects in the Ajile software suite which we must be aware of when working with triggering. These are External Trigger Settings which determine how external trigger signals will behave, Trigger Rules which link state outputs and external trigger inputs with control inputs and external trigger outputs, and Frame Trigger Settings which allow triggers to be enabled and trigger delays to be set on a per frame basis.

8.3.1 External Trigger Setting Members

External Trigger Settings define how external trigger signals will behave. This includes the trigger type (i.e. rising edge, falling edge) and trigger hold time. Components have two lists of External Trigger Settings, one for the external input triggers and one for the external output triggers. Each External Trigger Setting in the list correspondeds to a physical external trigger, so the lengths of the lists indicate how many external triggers are available. The members of External Trigger Settings are given in Table 8.3.

8.3.2 Trigger Rule Members

Trigger Rules connect the signals from device state outputs and external input triggers of components to device control inputs and external output triggers. The Project has a list of Trigger Rule Pairs which is added to. Trigger Rules are made up of tuples of (Component Index, Trigger Type) pairs, which are called Trigger Rule Pairs. An individual Trigger Rule has a list of Trigger Rule Pairs which take output signals and map them to a single trigger rule pair which takes an input signal. Examples of Trigger Rules will be seen later in this Chapter. For now, the members of a Trigger Rule and Trigger Rule Pair are given in Table 8.4 and Table 8.5, respectively.

8.3.3 Frame Trigger Setting Members

Each frame has a list of Frame Trigger Settings which allow for the device trigger signals (device control inputs and device state outputs) to be enabled or disabled on a per frame basis. Frame Trigger Settings also have a delay time included which allows for trigger signals to or from the device to be delayed from the time of the actual trigger event. The members of Frame Trigger Settings are given in Table 8.6.

8.4 Trigger Timing

The behavior of device state output and device control inputs is best explained by looking at the timing diagram of these signals along with the device frame state and lighting state. The timing diagram of

Device State Name	Description
NEXT_FRAME_READY	Before the component is ready to start the next frame, the data for the next frame must be loaded into the device. This loading time takes a finite amount of time and depends on the type of device. Once the next frame data has finished loading to the device, the next frame is then ready to be run (ready for the next display for a DMD, ready for the next exposure for a camera.) When this happens, this state output signal is emitted.
FRAME_STARTED	When the frame begins running (i.e. display begins for DMD or exposure begins for a camera), this state output signal is emitted.
FRAME_ENDED	When the frame is running the frame time begins to increment for that frame. When the given frame time for the frame has elapsed then the frame is finished and this state output signal is emitted.
LIGHTING_STARTED	If the component has a lighting controller attached, when the lighting is enabled this device state signal is emitted.
LIGHTING_ENDED	Once the lighting has been started, the lighting is enabled for the lighting on time defined for the frame. When this lighting on time for the frame has elapsed the lighting is disabled and this device state signal is emitted.
SEQUENCE_ITEM_ STARTED	When the first frame of a sequence item starts, this state output signal is emitted. Internally, this signal is in fact actually the `FRAME_STARTED` signal, but it is only enabled for the first frame of the sequence item rather than for each frame.

Table 8.1: List of possible device state outputs which components can expose in their list of device states.

Device Control Name	Description
START_FRAME	Start the next frame when this control input signal is observed. Note that the next frame must be ready for display/capture, see the `NEXT_FRAME_READY` device state.
END_FRAME	Stop the current frame when this control input signal is observed. The device must be currently running a frame for this to have effect.
START_LIGHTING	Start the lighting for the current frame when this control input signal is observed. The device must be currently running a frame, and the lighting must not have already completed, for this to have effect.
END_LIGHTING	Stop the lighting for the current frame when this control input signal is observed. This only has effect if the lighting for the frame has already started and has not already ended.
START_SEQUENCE_ ITEM	Start the first frame of a sequence item when is this control input signal is observed. Internally, this signal is in fact actually the `START_FRAME` signal, but it is only enabled for the first frame of the sequence item rather than for each frame.

Table 8.2: List of possible device control inputs which can be used to synchronize the behavior of components.

Name	Description
Trigger Type	The type of trigger which will be observed. Possible values include **rising edge** where we are sensitive to a low-to-high signal transition, **falling edge** where we are sensitive to a high-to-low signal transition, **any edge** which is sensitive to either rising or falling edge signals, **active high level** which is sensitive to a high logic level, and **active low level** which is sensitive to a low logic level.
Hold Time	Relevant for external output triggers only, this is the amount of time that an output signal will be held in the active state before being reset to the inactive state. For example, for a rising edge signal the hold time defines how long after the transition from a low to high signal that the output signal will remain in the high state before being reset back to the inactive low state.

Table 8.3: Description of members that are in an ExternalTriggerSetting.

Name	Description
Triggers From Device	A list of Trigger Rule Pairs which are output from the device.
Trigger To Device	A single Trigger Rule Pair which is input to the device.

Table 8.4: Description of members that are in an TriggerRule.

Name	Description
Component Index	The index of the component within the project components list for this trigger.
Trigger Type	The type of the trigger signal, which can be one of the state outputs or control inputs which are present in the component.

Table 8.5: Description of members that are in an TriggerRulePair.

Name	Description
Trigger Type	The type of the trigger signal, which can be one of the available state outputs or control inputs.
Enabled	Flag which sets the trigger signal to be enabled or disable for the given frame.
Delay	The delay after the triggering event is observed when the trigger will take effect.

Table 8.6: Description of members that are in an Frame Trigger Setting.

the device state ouptuts are shown in Figure 8.1. In this example the device is running in 'internal timing mode', meaning that the frames run with the timing given by the frame time (and the lighting is on for time given by the lighting timing) and when the frame time ends the next frame automatically starts. In the diagram, when the frame begins a FRAME_STARTED state output signal is observed. Similarly when the lighting has started a LIGHTING_STARTED state output signal can be observed. In conjunction with the start of the frame, the data for the next frame begins to get loaded (as is the case for a display device such as a DMD.) When the data for the next frame has completed loaded a NEXT_FRAME_READY state output signal is emitted which tells the outside world that the device is ready to start the next frame. Finally, when the frame ends and the lighting ends the FRAME_ENDED and LIGHTING_ENDED state outputs are emitted indicating these end events.

The timing diagram of the device control inputs are shown in Figure 8.2. In this example the device is running in 'external timing mode', meaning that the frame time and the lighting on time is controlled by the external device control input signals. In the diagram, the START_FRAME control input signal is first sent to the device, which starts the first frame. Once the frame has started, a START_LIGHTING control input signal is sent to the device to turn on the lighting. Next, an END_LIGHTING control input is sent to turn off the lighting. Finally, and END_FRAME control input signal ends the current frame. Since we are in 'external timing mode', when the previous frame ends the next frame will not start until a START_FRAME control input signal is observed by the device. Therefore, the device will wait for an arbitrary amount of time until the next START_FRAME control input signal is observed, after which the next frame begins. Note that it is important to also monitor the NEXT_FRAME_READY state output when externally triggering the device frames with the START_FRAME control input. If a START_FRAME control input is sent without the next frame being ready, the input signal will be ignored by the device and will effectively be missed. Therefore an external master should wait for a NEXT_FRAME_READY device state to be observed before emitting a START_FRAME device control.

8.5 Trigger Rule Structure

Trigger rules connect, or map, the state of one or more device state outputs or input triggers to a device control input or output trigger. Each trigger rule has a list of trigger rule pairs called the 'Triggers From Device' which define the device states or external input triggers of components, as well as a single trigger rule pair called the 'Trigger To Device' which defines the device control or externla output trigger of a component.

The trigger rules in a project are loaded onto the device and are evaluated by a *trigger rules engine* in FPGA hardware. The trigger rules engine performs the logical AND of each of the Triggers From Device. When each of the device state outputs or external input triggers in the Triggers From Device are satisfied in parallel (i.e. their logical AND evaluates to 1) then the *trigger rules engine* fires a singal to the Trigger To Device which is a device control input or external output trigger. If the Trigger To Device is an external trigger then the output signal will be described by the ExternalTriggerSetting for the component (which includes the trigger type and hold time). A trigger rule can be expressed mathematically with the following equation:

$$(c_1, ds_1|ei_1) \wedge (c_2, ds_2|ei_2) \wedge ... \wedge (c_n, ds_n|ei_n) \rightarrow (c_i, dc_j|eo_j) \tag{8.1}$$

where c_i refers to the component with index i, ds_i refers to a device state output with index i, ei_i is an external input trigger with index i, dc_i is a device control input and eo_j is an external output trigger with index j.

We now look at two examples of trigger rules. The first trigger rule example is shown in Figure 8.3 which maps external input and output triggers to and from the control inputs and state outputs of a single DMD component. Four trigger rules are in this example which are as follows:

$$(\text{DMD\_component}, \text{NEXT\_FRAME\_READY}) \rightarrow (\text{controller}, eo_1) \tag{8.2}$$

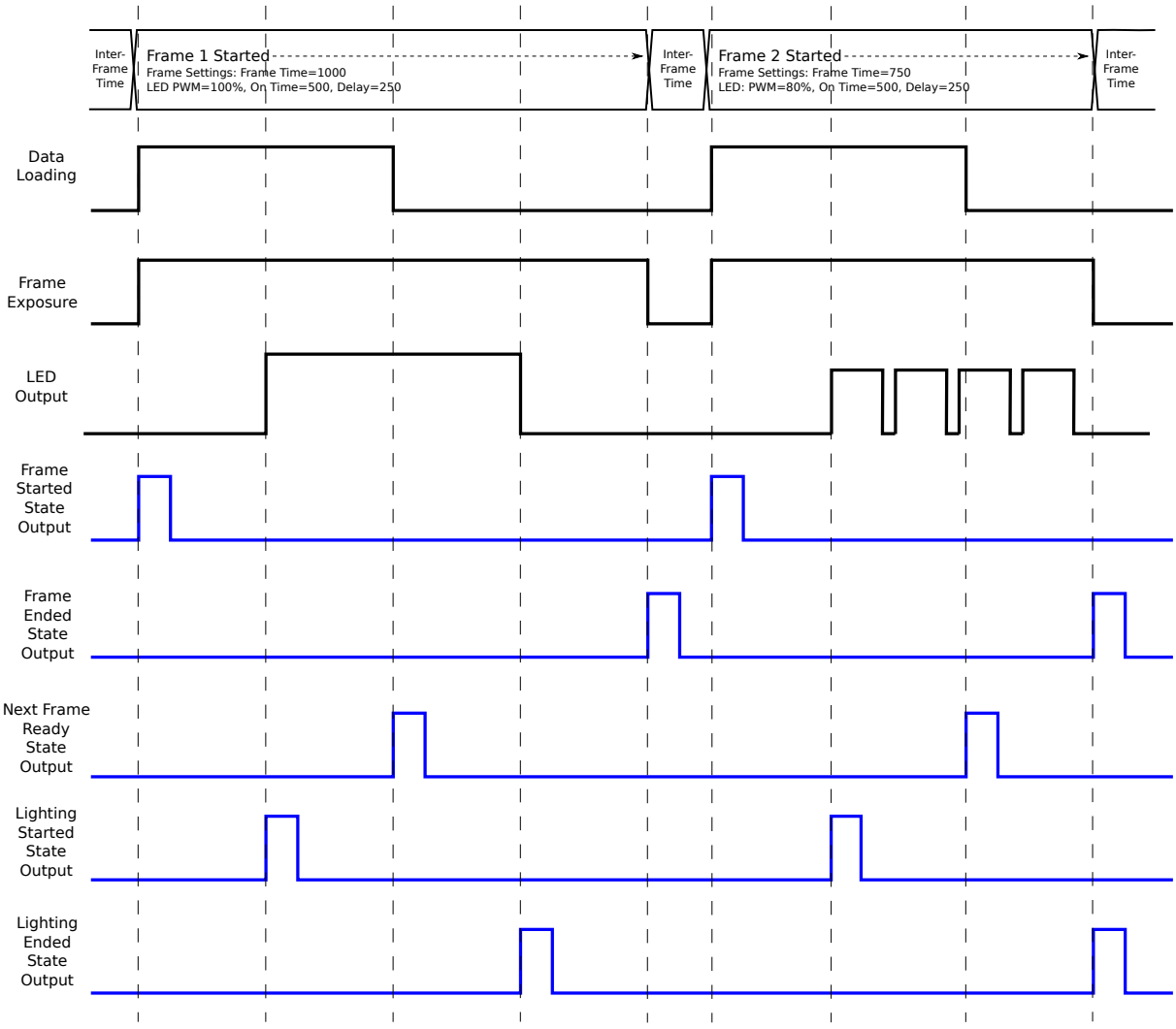

Figure 8.1: Device state output signals over two frames, showing the FRAME_STARTED, FRAME_ENDED, NEXT_FRAME_READY, LIGHTING_STARTED and LIGHTING_ENDED signals which correspond to the device frame and lighting states.

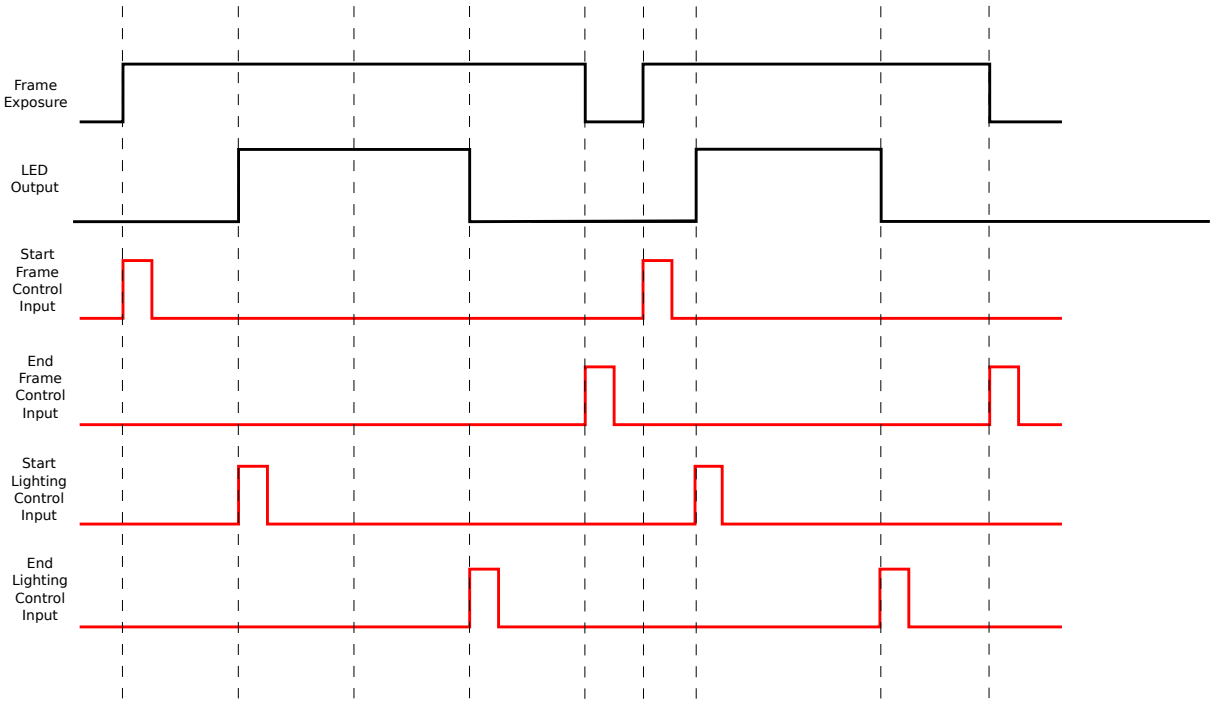

Figure 8.2: Device control input signals over two frames. The START_FRAME and END_FRAME input signals control the frame timing, while the START_LIGHTING and END_LIGHTING input signals control the lighting timing.

$$(\text{controller}, ei_1) \rightarrow (\text{DMD\_component}, \text{START\_FRAME}) \tag{8.3}$$

$$(\text{controller}, ei_2) \rightarrow (\text{DMD\_component}, \text{START\_LIGHTING}) \tag{8.4}$$

$$(\text{controller}, ei_2) \rightarrow (\text{DMD\_component}, \text{END\_LIGHTING}) \tag{8.5}$$

Note that external input trigger 2 is used to trigger both the start lighting and end lighting events. This can be done by setting the external trigger to be sensitive to any edge (i.e. both rising and falling edge).

The four trigger rules in this first example are very simple and only map external trigger signals to/from device states/controls. A more involved trigger rules example is shown in Figure 8.4 which consists of two components, a DMD component and a camera component, and three trigger rules. Some of the trigger rules involve multiple trigger rule pairs in a single rule which are logically AND'ed together. The trigger rules in this example are:

$$(\text{DMD}, \text{NEXT\_FRAME\_READY}) \wedge (\text{Camera}, \text{NEXT\_FRAME\_READY}) \rightarrow (\text{controller}, eo_1) \tag{8.6}$$

$$(\text{DMD}, \text{N\_F\_READY}) \wedge (\text{Camera}, \text{N\_F\_READY}) \wedge (\text{controller}, ei_1) \rightarrow (\text{DMD}, \text{START\_FRAME}) \tag{8.7}$$

$$(\text{DMD}, \text{LIGHTING\_STARTED}) \rightarrow (\text{Camera}, \text{START\_FRAME}) \tag{8.8}$$

The first trigger rule in this example is designed to allow an external master device to be informed when the next frame is ready to run. When the external device observes that both the DMD and Camera next frames are ready, the second trigger rule is used to allow the external device to start the next DMD frame

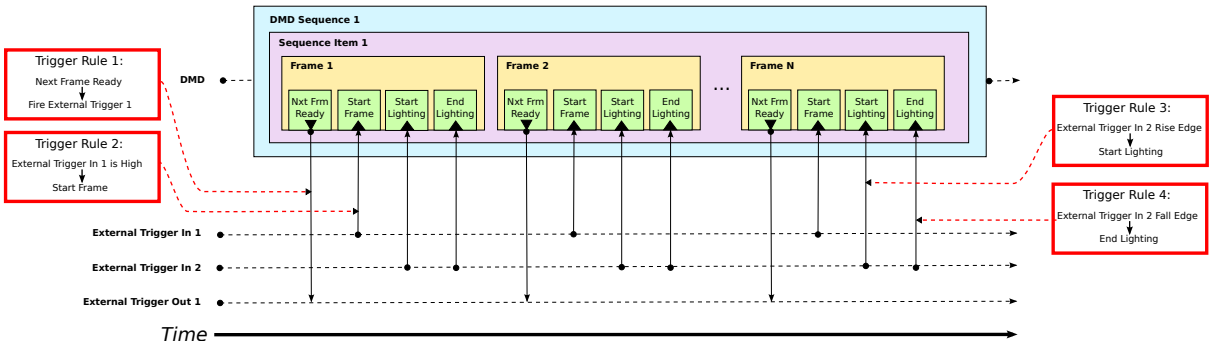

Figure 8.3: A set of Trigger Rules which connect external trigger signals to internal device states and controls.

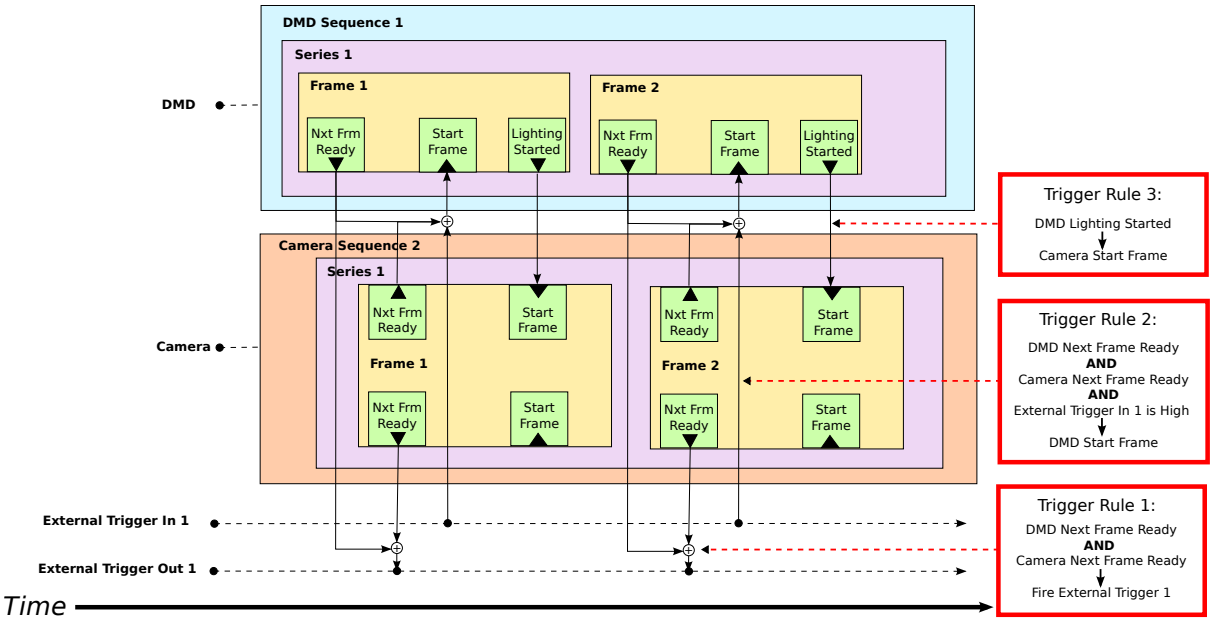

Figure 8.4: A set of three Trigger Rules defined by the Project which are evaluated in hardware to enable very tight synchronization between devices.

using the first external trigger in port. Finally, the third trigger signal is an internal trigger between two components, the DMD and the camera. After the DMD frame has started the lighting will automatically start which causes a LIGHTING_STARTED signal to be emitted. The third trigger rule connects this to the camera START_FRAME signal so that the camera exposure will start when the DMD lighting is on. The combined effect of all three trigger rules is then a DMD and Camera synchronized as slaves to an external device, with both the DMD and Camera frames running in perfect lockstep.

8.6 Configuring Trigger Settings

At the component level, components have lists of control inputs and state outputs which can be read, and list of External Trigger Settings for the input and output triggers. In this section we look at how to read and modify these component level trigger settings in the GUI and SDK.

8.6.1 Configuring Trigger Settings in the GUI

Viewing the lists of state outputs and control inputs along with the external trigger settings is done via the Ajile GUI Project Editor which was seen in Chapter 3. After opening the Project Editor, select the

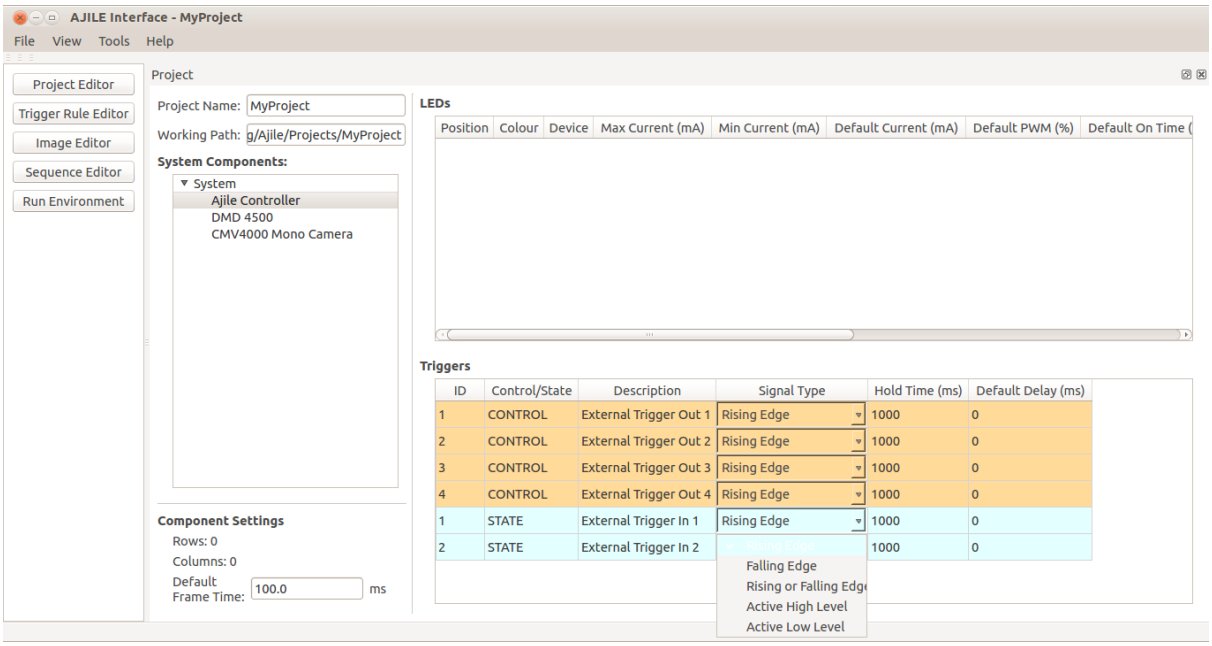

Figure 8.5: Screenshot of configuring trigger settings for the Ajile controller component.

component with the trigger settings of interest from the list of System Components. The list of triggers and their settings for the component are then shown in the Triggers table.

Figure 8.5 shows the list of triggers for the Ajile Controller component. These are all external triggers and we can modify the External Trigger Settings whose members were listed in Table 8.3. The trigger type can be selected with the drop down menu under the Signal Type header, the hold time can be set to a valid time for the trigger hol time, and the default delay for the trigger can be set, which will be used in the sequence editor when setting delays for frame triggers.

Figure 8.6 shows the list of triggers for a DMD component. For a DMD (and for a camera) the device states and controls do not include external triggers and therefore the Triggers table does not allow modifying the External Trigger Settings which include the Signal Type and Hold Time. We can however see what triggers are available to the component, which will be useful when designing trigger rules and sequences.

8.6.2 Configuring Trigger Settings in the SDK

Lists of device state outputs, device control inputs, input trigger settings and output trigger settings are contained in each Component in the Project. The lists of state outputs and control inputs for a component can be read from the accessor methods `Component.StateOutputs()` and `Component.ControlInputs()`, while the external input and output trigger settings can be read from the methods `Component.InputTriggerSettings()` and `Component.OutputTriggerSettings()`. To update the input or output trigger settings, such as the trigger type (e.g. rising or falling edge) and trigger hold time, a special function in the Project is available, `Project.SetTriggerSettings()`, which accepts as arguments the component index or component type of the target component which holds the trigger settings, and the updated lists of input and output trigger settings.

An example of configuring trigger settings is given in Listing 8.1 in Python and in Listing 8.2 in C++. The control inputs and state outputs of the DMD component are first output by iterating through the lists `Component.ControlInputs()` and `Component.StateOutputs()`. The lists of input trigger settings and output trigger settings are then modified and output by iterating through the lists `Component.InputTriggerSettings()` and `Component.OutputTriggerSettings()`. Finally, the input and output trigger settings were updated in a copy of our component but were never updated in the Project. The last line therefore updates the input and output trigger settings of the compo-

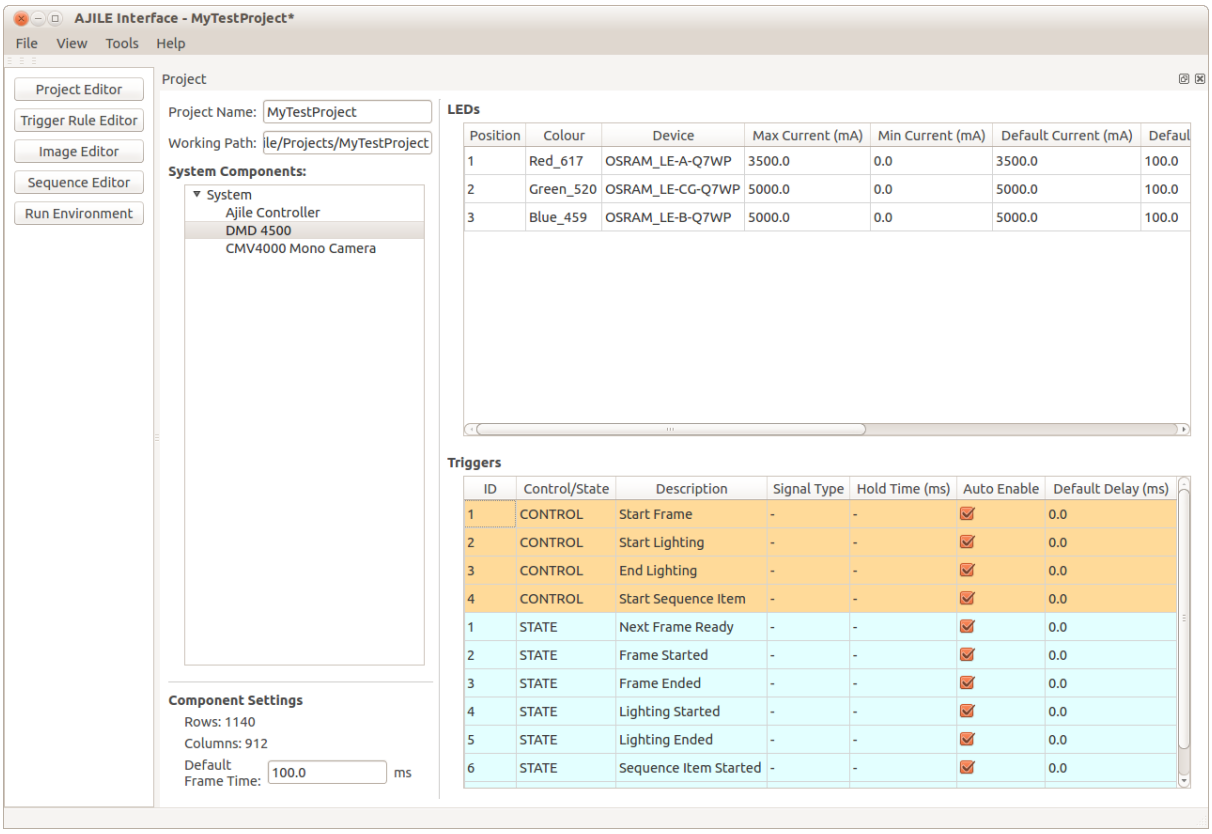

Figure 8.6: Screenshot of configuring trigger settings for the DMD component.

nent in our Project to be the same settings which are in our copied component using the function `Project.SetTriggerSettings()`.

```
controllerComponent = myProject.GetComponentWithDeviceType(MZED_DEVICE_TYPE)
dmdComponent = myProject.GetComponentWithDeviceType(DMD_4500_DEVICE_TYPE)
# output the list of device controls and states
print "Device Control Inputs list:"
for controlInput in dmdComponent.ControlInputs():
    print controlInput
print "Device State Outputs list:"
for stateOutput in dmdComponent.StateOutputs():
    print stateOutput
# update and print the input trigger settings
print "Input Trigger Settings: "
for i in range(len(controllerComponent.InputTriggerSettings())):
    controllerComponent.SetInputTriggerSetting(i, ExternalTriggerSetting(RISING_EDGE))
    print controllerComponent.InputTriggerSettings()[i].TriggerType()
# update and print the output trigger settings
print "Output Trigger Settings: "
for i in range(len(controllerComponent.OutputTriggerSettings())):
    controllerComponent.SetOutputTriggerSetting(i, ExternalTriggerSetting(FALLING_EDGE, 1000))
    print str(controllerComponent.OutputTriggerSettings()[i].TriggerType()) + " " + str(controllerComponent.
      OutputTriggerSettings()[i].HoldTimeUSec())
# update the trigger settings in the project
myProject.SetTriggerSettings(MZED_DEVICE_TYPE, controllerComponent.InputTriggerSettings(),
      controllerComponent.OutputTriggerSettings());
```

Listing 8.1: Python example of reading and setting device control inputs, device state outputs, input trigger settings and output trigger settings.

```
void TriggerSettingsExample(Project myProject) {
    Component controllerComponent = myProject.GetComponentWithDeviceType(
     AJILE_CONTROLLER_DEVICE_TYPE);
    Component dmdComponent = myProject.GetComponentWithDeviceType(DMD_4500_DEVICE_TYPE);
    // output the list of device controls and states
    cout << "Device Control Inputs list:" << endl;
    for (u8 i=0; i<dmdComponent.ControlInputs().size(); i++)
        cout << dmdComponent.ControlInputs()[i] << endl;
    cout << "Device State Outputs list:" << endl;
    for (u8 i=0; i<dmdComponent.StateOutputs().size(); i++)
        cout << dmdComponent.StateOutputs()[i] << endl;
    // update and print the input trigger settings
    cout << "Input Trigger Settings: " << endl;
    for (u8 i=0; i<controllerComponent.InputTriggerSettings().size(); i++) {
        controllerComponent.SetInputTriggerSetting(i, ExternalTriggerSetting(RISING_EDGE));
        cout << controllerComponent.InputTriggerSettings()[i].TriggerType() << endl;
    }
    // update and print the output trigger settings
    cout << "Output Trigger Settings: " << endl;
    for (u8 i=0; i<controllerComponent.OutputTriggerSettings().size(); i++) {
        controllerComponent.SetOutputTriggerSetting(i, ExternalTriggerSetting(FALLING_EDGE, 1000));
        cout << controllerComponent.OutputTriggerSettings()[i].TriggerType() << " "
             << controllerComponent.OutputTriggerSettings()[i].HoldTimeUSec() << endl;
    }
    // update the trigger settings in the project
    myProject.SetTriggerSettings(AJILE_CONTROLLER_DEVICE_TYPE,
                                 controllerComponent.InputTriggerSettings(),
                                 controllerComponent.OutputTriggerSettings());
}
```

Listing 8.2: C++ example of reading and setting device control inputs, device state outputs, input trigger settings and output trigger settings.

8.7 Creating Trigger Rules

The next step in setting up triggering to and from components is to create trigger rules and add them to the project. Trigger rules are designed to connect the outputs of one or more device states or external input triggers to the input of a device control or external output trigger.

8.7.1 Creating Trigger Rules in the GUI

Creating trigger rules in the GUI is done via the Trigger Rule Editor, which is shown in Figures 8.7 and 8.8. To open the Trigger Rule Editor click on the Trigger Rule Editor button (Figure 8.7, number 1). The Trigger Rule Editor is a visual editor which allows connecting the outputs of device states and external input triggers to the inputs of device controls and external output triggers. The components in the project and their device states and controls are displayed in the visual editor, represented by blocks which are labelled by the component device type name (Figure 8.7, number 2). On the left hand side of each component are the device states and/or external triggers in (Figure 8.7, number 3), and on the right hand side of each component are the device controls and/or external triggers out (Figure 8.7, number 4). For display purposes, the device state and control ports are labelled by their initials, for example 'NFR' is the `NEXT_FRAME_READY` state output. By holding the mouse over an input or output trigger port the full name of the trigger will be displayed as a mouse tooltip.

Initially the project will have no trigger rules. To add a new trigger rule, click on the 'Add New Rule' button (Figure 8.7, number 5). This puts the Trigger Rule Editor into the Adding Rule mode, which will be shown in the Current Mode text box (Figure 8.7, number 7). Once in the Adding Rule mode we can start creating a new trigger rule. This is done by clicking on one or more device states or external trigger in ports to generate a list of Triggers From Device for the trigger rule, then click on any of the device controls or external trigger out ports to set the Trigger To Device for the rule. Clicking on a device state or external trigger in which is already included with the trigger rule will remove it from that rule. Finally, when the trigger rule creation is complete we finish editing the rule and add it to the project, which is done by clicking the Done Adding/Editing Rule button (Figure 8.7, number 6). Trigger rules

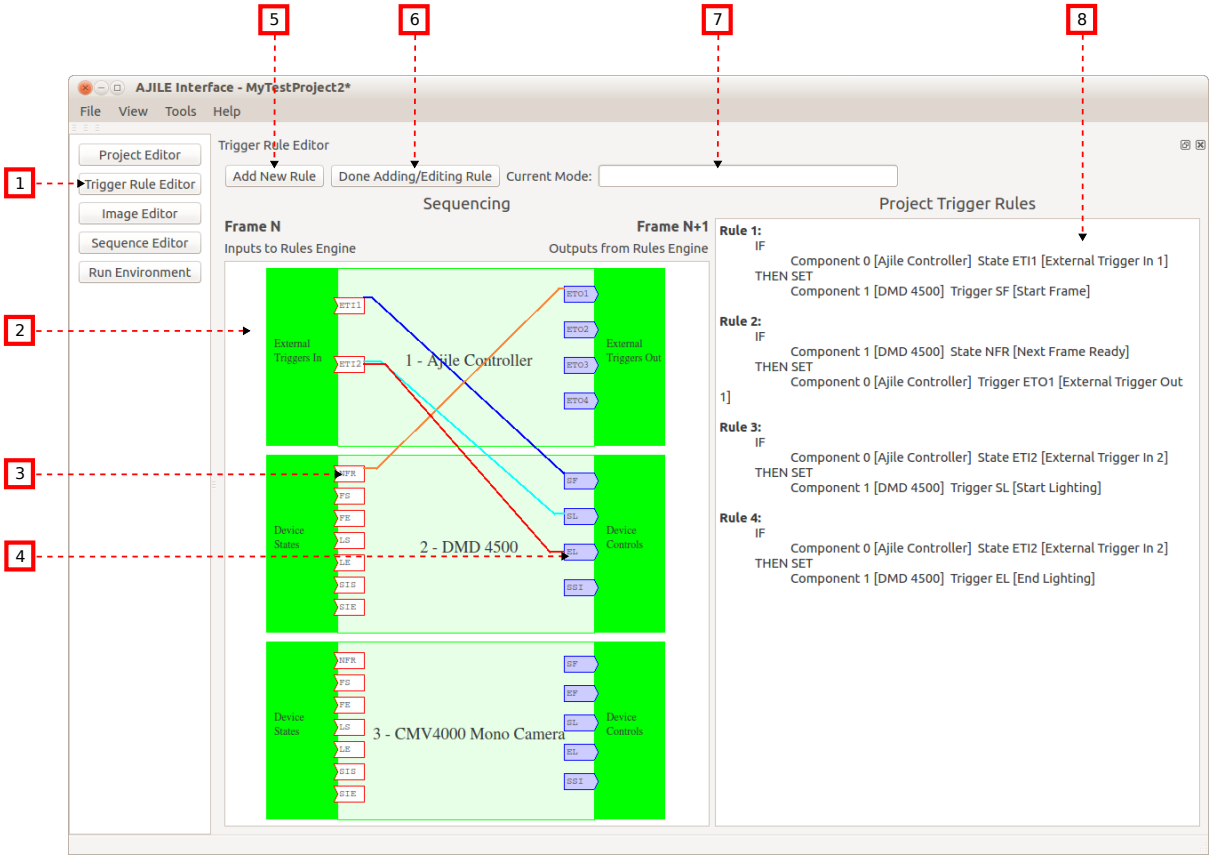

Figure 8.7: Screenshot of creating trigger rules for mapping external triggers to state outputs and control inputs. The four trigger rules in this screenshot correspond to the trigger rules shown in Figure 8.3.

are displayed in the visual editor has line segments connecting the input and output triggers (Figure 8.8, number 2). Once a rule has been added to the project, it is possible to edit or delete existing rules by left clicking on the trigger rule line segments and selecting Edit Rule or Delete Rule from the popup menu (Figure 8.8, number 1).

In Figure 8.7 four trigger rules were created in the Trigger Rule Editor. These four trigger rules are the same as those which we saw in the example of Figure 8.3 which map device states of a DMD component to external output triggers, and external input triggers to device controls of a DMD. Not only can each of the Trigger Rules be displayed visually with the connections between output and input triggers, but each rule is also displayed with a text representation which can help to summarize the trigger rules (Figure 8.7, number 8). The text representation is of the format 'IF (state1 AND state2 AND ...) THEN SET (control)'. In Figure 8.8 three trigger rules were created with the Trigger Rule Editor. These rules correspond to those in the example of Figure 8.4 which connects device states to external output triggers, and also connects device states and controls between DMD and camera components.

8.7.2 Creating Trigger Rules in the SDK

Creating trigger rules in the SDK involves creating `TriggerRule` objects, adding `TriggerRulePair` objects to the Triggers From Device list which are device state outputs or external triggers in using the function `TriggerRule.AddTriggerFromDevice()`, then setting a `TriggerRulePair` as the Trigger To Device which is a device control input or external output trigger using the function `TriggerRule.SetTriggerToDevice()`. When the `TriggerRule` has been created, it is then added to the project wtih the function `Project.AddTriggerRule()`.

An example of creating trigger rules and adding them to a project is given in Python in Listing 8.3 and in C++ in Listing 8.4. This example creates the three trigger rules which were seen in the example of

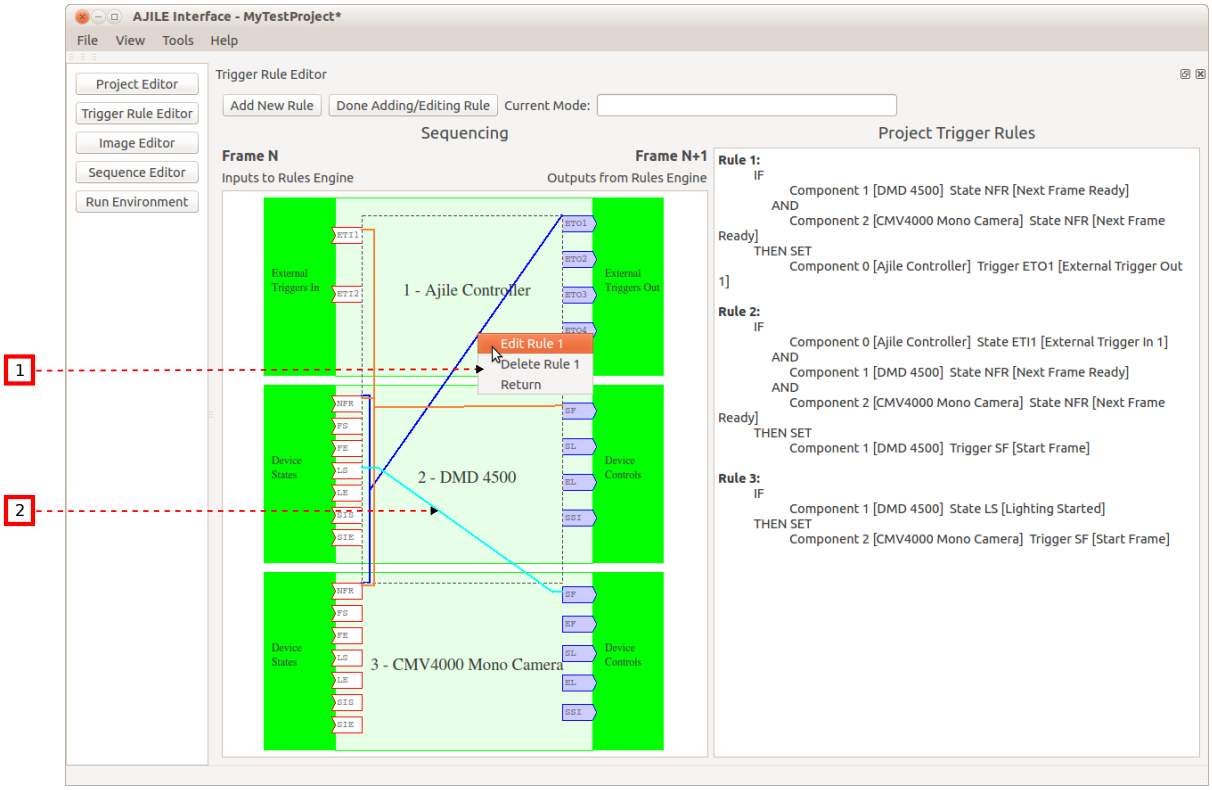

Figure 8.8: Screenshot of creating trigger rules to connect state outputs and control inputs between components and to/from external triggers. The three trigger rules in this screenshot correspond to the trigger rules shown in Figure 8.4.

Figure 8.4. Triggers from three different components are used and so the first step to creating the trigger rules is to obtain the component indices of the Ajile controller, DMD and camera components in the project using `Project.GetComponentIndexWithDeviceType()` which returns the first component that matches the given device type. The three trigger rules are then created by adding TriggerRulePairs for the trigger ports corresponding to the example of Figure 8.4. Finally, the three trigger rules are added to the project with `Project.AddTriggerRule()` and the number of trigger rules is output by observing the length of the trigger rule list with `Project.TriggerRules()`.

```
# get the component indices of the controller, DMD and camera
controllerIndex = myProject.GetComponentIndexWithDeviceType(MZED_DEVICE_TYPE)
dmdIndex = myProject.GetComponentIndexWithDeviceType(DMD_4500_DEVICE_TYPE)
cameraIndex = myProject.GetComponentIndexWithDeviceType(CMV_4000_MONO_DEVICE_TYPE)
# create three trigger rules
rule1 = TriggerRule()
rule1.AddTriggerFromDevice(TriggerRulePair(dmdIndex, NEXT_FRAME_READY))
rule1.AddTriggerFromDevice(TriggerRulePair(cameraIndex, NEXT_FRAME_READY))
rule1.SetTriggerToDevice(TriggerRulePair(controllerIndex, EXT_TRIGGER_OUTPUT_1))
rule2 = TriggerRule()
rule2.AddTriggerFromDevice(TriggerRulePair(controllerIndex, EXT_TRIGGER_INPUT_1))
rule2.AddTriggerFromDevice(TriggerRulePair(dmdIndex, NEXT_FRAME_READY))
rule2.AddTriggerFromDevice(TriggerRulePair(cameraIndex, NEXT_FRAME_READY))
rule2.SetTriggerToDevice(TriggerRulePair(dmdIndex, START_FRAME))
rule3 = TriggerRule()
rule3.AddTriggerFromDevice(TriggerRulePair(dmdIndex, LIGHTING_STARTED))
rule3.SetTriggerToDevice(TriggerRulePair(cameraIndex, START_FRAME))
# add the trigger rules to the project
myProject.AddTriggerRule(rule1)
myProject.AddTriggerRule(rule2)
myProject.AddTriggerRule(rule3)
print len(myProject.TriggerRules())
```

Listing 8.3: Python example of creating trigger rules and adding them to the project.

```
void TriggerRuleExample(Project myProject) {
    // get the component indices of the controller, DMD and camera
    int controllerIndex = myProject.GetComponentIndexWithDeviceType(AJILE_CONTROLLER_DEVICE_TYPE);
    int dmdIndex = myProject.GetComponentIndexWithDeviceType(DMD_4500_DEVICE_TYPE);
    int cameraIndex = myProject.GetComponentIndexWithDeviceType(CMV_4000_MONO_DEVICE_TYPE);
    // create three trigger rules
    TriggerRule rule1;
    rule1.AddTriggerFromDevice(TriggerRulePair(dmdIndex, NEXT_FRAME_READY));
    rule1.AddTriggerFromDevice(TriggerRulePair(cameraIndex, NEXT_FRAME_READY));
    rule1.SetTriggerToDevice(TriggerRulePair(controllerIndex, EXT_TRIGGER_OUTPUT_1));
    TriggerRule rule2;
    rule2.AddTriggerFromDevice(TriggerRulePair(controllerIndex, EXT_TRIGGER_INPUT_1));
    rule2.AddTriggerFromDevice(TriggerRulePair(dmdIndex, NEXT_FRAME_READY));
    rule2.AddTriggerFromDevice(TriggerRulePair(cameraIndex, NEXT_FRAME_READY));
    rule2.SetTriggerToDevice(TriggerRulePair(dmdIndex, START_FRAME));
    TriggerRule rule3;
    rule3.AddTriggerFromDevice(TriggerRulePair(dmdIndex, LIGHTING_STARTED));
    rule3.SetTriggerToDevice(TriggerRulePair(cameraIndex, START_FRAME));
    // add the trigger rules to the project
    myProject.AddTriggerRule(rule1);
    myProject.AddTriggerRule(rule2);
    myProject.AddTriggerRule(rule3);
    cout << myProject.TriggerRules().size() << endl;
}
```

Listing 8.4: C++ example of creating trigger rules and adding them to the project.

8.8 Per Frame Trigger Settings

It is possible for device state outputs and device control inputs to be enabled or disabled on a per-frame basis. Also, trigger delay times can be set frame by frame. In this section we show how to enable/disbale triggers and set their delays per frame in the GUI and SDK.

8.8.1 Per Frame Trigger Settings in the GUI

In the GUI, enabling triggers and setting their delays per frame is done using the Sequence Editor which was described in Chapter 6. The frame trigger settings are displayed in the columns labelled 'Trig Enabled' and 'Trig Delay'. The settings of the device states are displayed in blue (Figure 6.4, number 12) and the device controls are displayed in orange (Figure 6.4, number 13). The names of the states and controls are abreviated for display purposes but the full name can be seen by hovering the mouse of the label. The given device state or control can be enabled or disabled by checking or unchecking the checkbox in the 'Trig Enabled' column, and the trigger delay can be set for the device state or control for the frame by entering a time value in the 'Trig Delay' column.

When a new frame is added to the sequence, by default the device states and controls are enabled for the frame if they are used in any of the defined trigger rules. This prevents trigger rules from accidently being ignored due to forgetting to enable the per-frame trigger settings. This default enabling of frame trigger settings based on the current trigger rules can be turned off by the user if need be. This is done in the Project Editor, seen in Figure 8.6, by unchecking the 'Auto Enable' checkbox next to the device control or state in the Triggers table.

8.8.2 Per Frame Trigger Settings in the SDK

For many applications we do not need to adjust trigger settings on a per frame basis. In such cases, frame trigger settings can be ignored and the sequence verifier will automatically enable the device states and controls when a sequence is run by analysing the list of trigger rules in the project to determine which device states and controls are in use. For more advanced sequences we show here how to adjust trigger settings for individual frames.

To enable or disable control inputs and state outputs and set their delay times per frame in the SDK we create `FrameTriggerSetting` objects and add them to the frame using `Frame.AddControlInputSetting()` or `Frame.AddStateOutputSetting()`. An example of setting `FrameTriggerSetting` is given in Listing 8.5 in Python and in Listing 8.6 in C++. In this example, two frames are created then four `FrameTriggerSetting` objects are created where two of them are used for control inputs and two are for state outputs. The values of the `FrameTriggerSetting` are set using the constructor, with the trigger type (shown in Table 8.1 and 8.2) as the first argument, whether or not the state/control is enabled as the second argument, and the trigger delay time as the third argument. These `FrameTriggerSettings` are added the frames using `FrameTriggerSetting.AddControlInputSetting()` and `Frame.AddStateOutputSetting()` which adds them to the end of the control input and state output setting lists of the frame. The list lengths are retrieved and printed and finally the frames are added to the project.

```
sequenceID = 1
# create two frames
frame1 = Frame(sequenceID)
frame2 = Frame(sequenceID)
# create FrameTriggerSettings for the control inputs
startFrameSetting = FrameTriggerSetting(START_FRAME, False)
endLightingSetting = FrameTriggerSetting(END_LIGHTING, True, 10000)
# create FrameTriggerSettings for the state outputs
frameStartedSetting = FrameTriggerSetting(FRAME_STARTED, True, 500)
nextReadySetting = FrameTriggerSetting(NEXT_FRAME_READY, False, 0)
# add FrameTriggerSettings to the frames
frame1.AddControlInputSetting(startFrameSetting)
frame1.AddControlInputSetting(endLightingSetting)
frame1.AddStateOutputSetting(nextReadySetting)
frame2.AddControlInputSetting(startFrameSetting)
frame2.AddStateOutputSetting(frameStartedSetting)
print  str(len(frame1.ControlInputSettings())) + " " + \
    str(len(frame1.StateOutputSettings())) + " " + \
    str(len(frame2.ControlInputSettings())) + " " + \
    str(len(frame2.StateOutputSettings()))
# add the frames to the project
myProject.AddFrame(frame1)
myProject.AddFrame(frame2)
```

Listing 8.5: Python example of creating and setting FrameTriggerSettings to enable/disable or set delays for device states and controls for individual frames.

```
void FrameTriggerSettingsExample(Project myProject) {
    u16 sequenceID = 1;
    // create two frames
    Frame frame1(sequenceID);
    Frame frame2(sequenceID);
    // create FrameTriggerSettings for the control inputs
    FrameTriggerSetting startFrameSetting(START_FRAME, false);
    FrameTriggerSetting endLightingSetting(END_LIGHTING, true, 10000);
    // create FrameTriggerSettings for the state outputs
    FrameTriggerSetting frameStartedSetting(FRAME_STARTED, true, 500);
    FrameTriggerSetting nextReadySetting(NEXT_FRAME_READY, false, 0);
    // add FrameTriggerSettings to the frames
    frame1.AddControlInputSetting(startFrameSetting);
    frame1.AddControlInputSetting(endLightingSetting);
    frame1.AddStateOutputSetting(nextReadySetting);
    frame2.AddControlInputSetting(startFrameSetting);
    frame2.AddStateOutputSetting(frameStartedSetting);
    cout << frame1.ControlInputSettings().size() << " "
         << frame1.StateOutputSettings().size() << " "
         << frame2.ControlInputSettings().size() << " "
         << frame2.StateOutputSettings().size() << endl;
    // add the frames to the project
    myProject.AddFrame(frame1);
    myProject.AddFrame(frame2);
}
```

Listing 8.6: C++ example of creating and setting FrameTriggerSettings to enable/disable or set delays for device states and controls for individual frames.

Chapter 9

System Control

The previous chapters in this guide have focused on setting up Ajile Projects. To actually make use of the created projects we now look at how to load projects onto an Ajile system and to run sequences within projects.

For GUI users loading and running projects is a simple process which involves connecting to the hardware, loading the currently opened project then selecting a sequence to be run by a component.

Loading and running projects is also straightforward in the SDK. The top-level object which takes care of managing the system is called the `AjileSystem`. The `AjileSystem` takes care of communication between the user application and the Ajile Controller and makes device drivers available to the user to load projects to the controller and to run sequences.

9.1 Connecting to the Device

The first step to running projects on the device is to connect to the device. There are a number of different communication interfaces available to connect user applications and the GUI to Ajile devices, which include USB 2, USB 3.0, Gigabit Ethernet, and PCI-express. Start by powering on your Ajile device and connecting its communication interface cable to the PC (e.g. USB cable, Ethernet cable, PCIe cable). Note that depending on which communication interface you are using, additional drivers and settings must be installed in the operating system for it work, see Section 1.3 for instructions.

9.1.1 Connecting to the Device in the GUI

Connecting to the hardware device is done in the GUI with the Run Environment which is shown in Figure 9.1. To get to the Run Environment click on the 'Run Environment' button on the left navigation bar (Figure 9.1, number 1). The GUI starts in the disconnected state, as indicated by the 'Connected' status checkbox which is unchecked (Figure 9.1, number 3). The Ajile GUI remembers the connection settings from the previous session so that we can immediately connect to the device after the initial setup. To set up the connection settings, which will likely be required the first time connecting to the device, click on 'Edit Connection Settings' (Figure 9.1, number 6). This brings up the Connection Settings dialog (Figure 9.1, number 6) which allows us to select the connection interface type (i.e. USB 2, Ethernet, etc.) and to configure the settings for the selected connection type. The USB 2 connection is selected in the example (Figure 9.1, number 5), and the Ethernet connection settings for the device can be configured if needed (since the USB 2 device uses Ethernet over USB). Once the connection settings have been set we accept the settings by clicking on the 'Accept' button. Finally, we connect to the device by clicking on the 'Connect to HW' button (Figure 9.1, number 3). If successful, the Connected status checkbox will become checked - otherwise an error message will be displayed. Also, the device status window (Figure 9.2, number 1) will display the list of components and their device states once connected.

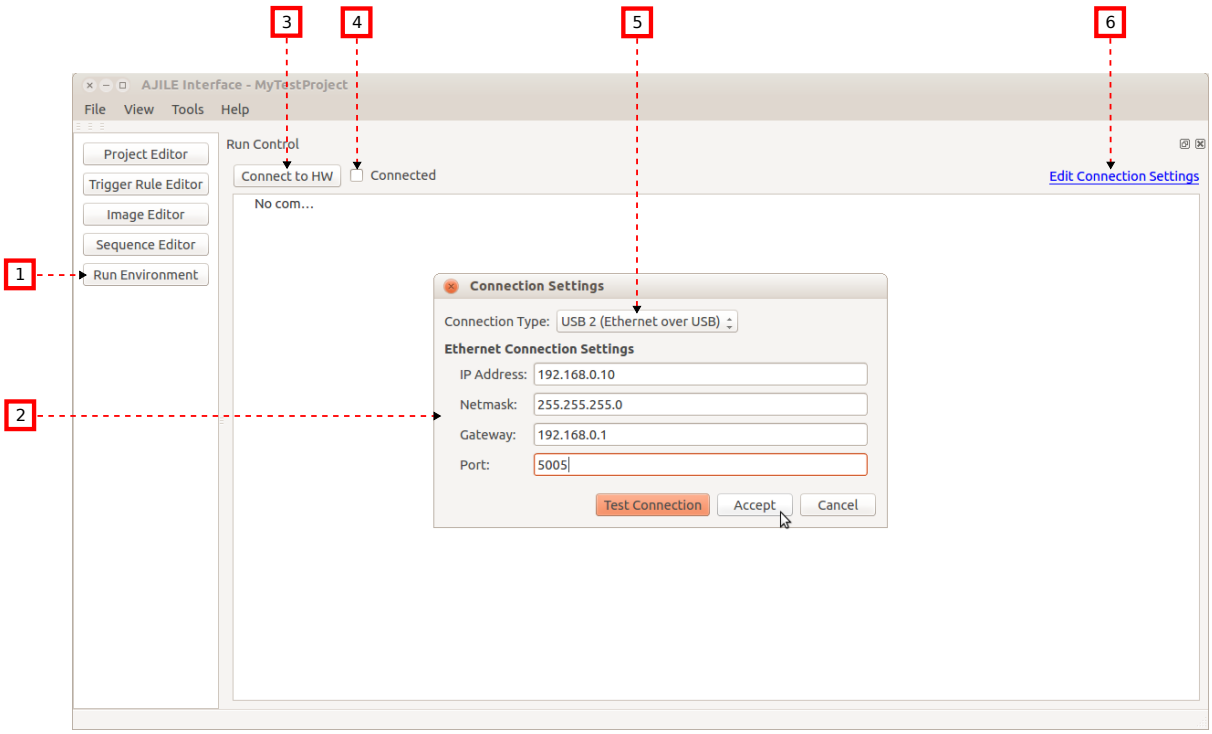

Figure 9.1: Screenshot of the Run Environment in the Ajile GUI before connecting to the device. Connection settings can be configured prior to connecting with the Connection Settings dialog.

9.1.2 Connecting to the Device in the SDK

Connections and communication with the hardware device is accomplished in the SDK using the `AjileSystem` interface. Most applications which use the Ajile SDK will be built on a PC system, which is the main focus of this guide. In this case, we use the `HostSystem` object to manage the device, which is implementation of the `AjileSystem` interface which runs on a host PC (Windows, Linux, Mac). Note that for advanced users, SDK applications can actually be run directly on the device in an embedded ARM-Linux environment by using the `MultiCoreSystem` implementation of the `AjileSystem`. This advanced usage will be the topic of future tutorials.

An example of setting up connection settings and connecting to the device using the `HostSystem` object is given in Listing 9.1 in Python and in Listing 9.2 in C++. A new `HostSystem` (which implements `AjileSystem`) is first created. The Ethernet connection settings for the `AjileSystem` are then set using `AjileSystem.SetConnectionSettingsStr()`. Strings are passed in for the connection settings which include the IP address, netmask, gateway address and port (the port is a 16-bit integer). Once the connection settings are configured, we can start the system with `AjileSystem.StartSystem()`. Starting the `AjileSystem` has the effect of connected to the hardware device using the given connection settings and also starting the message handling communication drivers internally to the `AjileSystem`. Note that the communication system inside the `AjileSystem` runs independantly of the user application so that user code does not need to worry about the details of that messaging system. Finally, we display whether the system is connected with `AjileSystem.IsConnected()` and return the `AjileSystem` to the caller function so that the rest of the application can continue to use it to communicate with the device for the lifetime of the program.

```python
def ConnectToDevice():
    # create a new HostSystem, which is an AjileSystem instance
    system = HostSystem()
    # set connection settings for the system
    system.SetConnectionSettingsStr("192.168.2.210",\
                                    "255.255.255.0",\
                                    "0.0.0.0", 5005)
    # start the system, which will establish the connection
    system.StartSystem()
    print system.IsConnected()
    # return the system so that it can be used by the application
    return system
```

Listing 9.1: Python example of creating an AjileSystem as a HostSystem, setting its Ethernet connection settings, and starting it (which connects it to the hardware device).

```cpp
AjileSystem* ConnectToDevice() {
    // create a new HostSystem, which is an AjileSystem instance
    AjileSystem* system = new HostSystem();
    // set connection settings for the system
    system->SetConnectionSettingsStr("192.168.2.210",
                                     "255.255.255.0",
                                     "0.0.0.0", 5005);
    // start the system, which will establish the connection
    system->StartSystem();
    cout << system->IsConnected() << endl;
    // return the system so that it can be used by the application
    return system;
}
```

Listing 9.2: C++ example of creating an AjileSystem as a HostSystem, setting its Ethernet connection settings, and starting it (which connects it to the hardware device).

9.2 Loading Projects

Once we are connected to the device the next step is to load our project onto the device so that it can be run.

9.2.1 Loading Projects in the GUI

After connecting to the device a list of Actions appear in the Run Environment which allows loading and running projects. By clicking on the 'Load' button (Figure 9.2, number 3) the currently opened project is sent (loaded) to the device Ajile Controller. Once complete the 'Run' button appears (Figure 9.2, number 4) and the sequences in the project are ready to be run by the components in the system.

9.2.2 Loading Projects in the SDK

Once we have an `AjileSystem` which is connected to the device, we need to get a `ControllerDriver` from it which is able to load projects onto the Ajile Controller and run sequences. The `ControllerDriver` object is obtained by using the function `AjileSystem.GetDriver()` which returns it from the running system. The `ControllerDriver` has numerous functions which enable controlling the Ajile devices and is the most important object for dealing with system control. Here we look mainly at the function `ControllerDriver.LoadProject()`, which as the name suggests loads (sends) an entire project to the device.

An example of loading a project to the device is given in Python in Listing 9.3 and in C++ in Listing 9.4. It is assumed that an `AjileSystem` has been passed into the example function which is already started and connected to the device, as well as a valid project for the device. We get access to the `ControllerDriver` object which is inside the `AjileSystem` with `AjileSystem.GetDriver()`. The `ControllerDriver` can then be used to load our project to the device with `ControllerDriver.LoadProject()`. Notice that

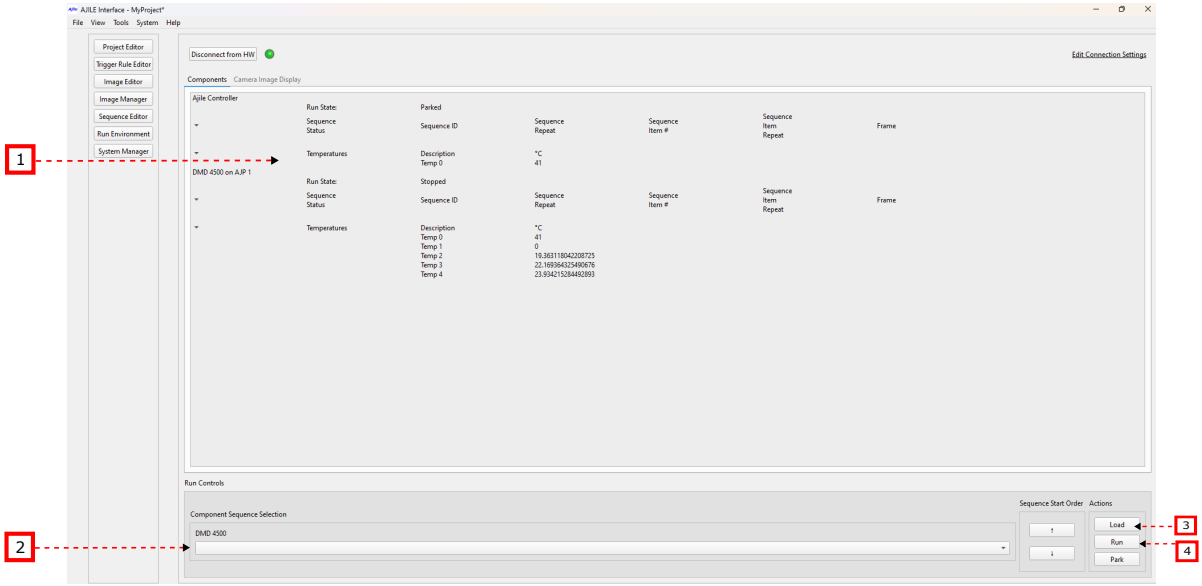

Figure 9.2: Screenshot of the Run Environment in the Ajile GUI once connected. The project is loaded and the sequence can be selected from our project to run.

`ControllerDriver.LoadProject()` is non-blocking, meaning that it returns immediately without waiting for the project to be completely transferred to the device. The actual transfer of the project is done in the background on a seperate thread by the `AjileSystem`, and this transfer may be almost instant or may take several seconds depending on the connection type used and on the size of the project. In order to know when the project has been fully loaded onto the device we use the function `ControllerDriver.WaitForLoadComplete` and pass in a timeout value (in milliseconds). Passing in a negative timeout value as shown in the example makes the application wait indefinitely until the transfer has completed.

```python
def LoadProjectExample(ajileSystem, myProject):
    # get the controller driver from the AjileSystem
    driver = ajileSystem->GetDriver()
    # load the project onto the device
    driver.LoadProject(myProject)
    # loading is non-blocking. Wait for the project load to complete
    driver.WaitForLoadComplete(-1)
```

Listing 9.3: Python example of getting a `ControllerDriver` from the `AjileSystem` and using it to load a project onto the Ajile Controller device.

```cpp
#include "ControllerDriver.h"
void LoadProjectExample(AjileSystem* ajileSystem, Project myProject) {
    // get the controller driver from the AjileSystem
    ControllerDriver* driver = ajileSystem->GetDriver();
    // load the project onto the device
    driver->LoadProject(myProject);
    // loading is non-blocking. Wait for the project load to complete
    driver->WaitForLoadComplete(-1);
}
```

Listing 9.4: C++ example of getting a `ControllerDriver` from the `AjileSystem` and using it to load a project onto the Ajile Controller device.

9.3 Running Sequences

After loading a project to the device, the next step is to run a sequence on the target component.

9.3.1 Running Sequences in the GUI

The first step to running a sequence is to select which sequence is to run on which component. This is done by selecting the sequence from the sequence selection drop down beneath the component on which it will run (Figure 9.2, number 2). Next, with the project loaded onto the device the 'Run' button appears in the Run Environment below the 'Load' button (Figure 9.2, number 4). Clicking on the 'Run' button will begin the selected sequence(s) on the target component(s). The sequence will run on the component until the end of the sequence, after which the component will go back into the stopped (idle) state. Running sequences can also be paused or stopped before they complete by clicking on the 'Pause' or 'Stop' buttons. Pausing has the effect of repeating the currently running sequence item indefinitely on the component until the sequence is resumed. In addition there is a 'Next' button which when clicked causes the running sequence to immediately advance to its next sequence item regardless of the current frame and sequence item repeat count. This allows us to effectively 'step' through the sequence one sequence item at a time by using the 'Next' sequence item button.

9.3.2 Running Sequences in the SDK

Running sequences on a target component in the SDK is accomplished using the function `ControllerDriver.StartSequen` where we pass in the sequence ID of the sequence in our project that we would like to run and the component index of the component which will run the sequence (which is the index of the component in the list of components in the project). An example of running a sequence is given in Python in Listing 9.5 and in C++ in Listing 9.6. In the example we re-use the example of Listing 9.3 and 9.4 to load the project onto the device. Following that, assuming that the loaded project has within it a sequence with sequence ID 1, and the connected device has a component at component index 1 which is compatible with that sequence, then we can run sequence 1 on component 1 by passing in the sequence ID and component index to `ControllerDriver.StartSequence()`. Once the sequence has started, we may wish to pause it mid sequence. This is done with the `ControllerDriver.PauseSequence()` function. Note that `ControllerDriver.PauseSequence()` only requires the component index and not the sequence ID since the sequence is already running on that component. While paused we can advance the current sequence item one at a time with the function `ControllerDriver.NextSequenceItem()`. Finally, we resume the sequence with `ControllerDriver.StartSequence()` and then stop it with `ControllerDriver.StopSequence()`.

```
def RunSequenceExample(ajileSystem, myProject):
    # load the project to the system using our previous example
    LoadProjectExample(ajileSystem, myProject)
    # get the controller driver from the AjileSystem
    driver = ajileSystem->GetDriver()
    # run the sequence with ID 1 on component 1
    sequenceID = 1
    componentIndex = 1
    driver.StartSequence(sequenceID, componentIndex)
    # wait the sequence on our component
    driver.PauseSequence(componentIndex)
    # advance the sequence items manually one by one
    driver.NextSequenceItem(componentIndex)
    driver.NextSequenceItem(componentIndex)
    # start the sequence again (i.e. un-pause the component)
    driver.StartSequence(sequenceID, componentIndex)
    # finally, stop the sequence
    driver.StopSequence(componentIndex)
```

Listing 9.5: Python example of loading a project onto the Ajile Controller, then running, pausing and stopping a sequence from that project on a target component.

```
void RunSequenceExample(AjileSystem* ajileSystem, Project myProject) {
    // load the project to the system using our previous example
    LoadProjectExample(ajileSystem, myProject);
    // get the controller driver from the AjileSystem
    ControllerDriver& driver = *ajileSystem->GetDriver();
    // run the sequence with ID 1 on component 1
    u16 sequenceID = 1;
    u8 componentIndex = 1;
    driver.StartSequence(sequenceID, componentIndex);
    // wait the sequence on our component
    driver.PauseSequence(componentIndex);
    // advance the sequence items manually one by one
    driver.NextSequenceItem(componentIndex);
    driver.NextSequenceItem(componentIndex);
    // start the sequence again (i.e. un-pause the component)
    driver.StartSequence(sequenceID, componentIndex);
    // finally, stop the sequence
    driver.StopSequence(componentIndex);
}
```

Listing 9.6: C++ example of loading a project onto the Ajile Controller, then running, pausing and stopping a sequence from that project on a target component.

9.4 Device Status Information

While we are connected to the device there is a variety of status information which can be retrieved from the device, including the state of each of the devices in the system and the current status of the running sequence (i.e. the current frame number, sequence item number, and repeat count).

9.4.1 Device State

Each component in the system has a device state which shows the current run state of the component (running, paused or stopped), the on-board temperature of the device in degrees Celcius, the temperatures of the connected LEDs and LED controller if applicable, the current status of the running sequence on the device, and the state of the device control inputs and device state outputs for the component.

Getting the Device State in the GUI

The device states of components are automatically retrieved and updated in the Ajile GUI while we are connected to the device. The device states are displayed in the device status window of the Run Environment (Figure 9.2, number 1) and can be used to see what components are available and to monitor their states and temperatures.

Getting the Device State in the SDK

Getting the device state of the components in the system is done in two steps in the SDK. First the device states must be retrieved from the connected device. This is done by using the `ControllerDriver.RetrieveDeviceState()` function to request the updated states from the device. `ControllerDriver.RetrieveDeviceState()` can accept an optional timeout argument to allow the caller to wait for the device states to be received. Once the device states have been retrieved they are stored in the `AjileSystem` and are obtained with `AjileSystem.GetDeviceState()`. The component index (as indexed in the project components list) is passed into `AjileSystem.GetDeviceState()` and a `DeviceState` object is returned which contains the device state information of that component. An example of retrieving the device state of a DMD device and outputting its temperature information is shown in Listing 9.7 in Python and in Listing 9.8 in C++.

```python
def DeviceStateExample(ajileSystem):
    dmdComponentIndex = 1
    # retrieve the device state and wait for it to return
    ajileSystem.GetDriver().RetrieveDeviceState(-1)
    # get the retrieved device state
    state = ajileSystem.GetDeviceState(dmdComponentIndex)
    # output the list of temperatures from the device state
    for temperature in state.Temperatures():
        print temperature
```

Listing 9.7: Python example of retrieving the device states of the connected components, then getting the DMD device state and outputting its temperature list.

```cpp
void DeviceStateExample(AjileSystem* ajileSystem) {
    u8 dmdComponentIndex = 1;
    // retrieve the device state and wait for it to return
    ajileSystem->GetDriver()->RetrieveDeviceState(-1);
    // get the retrieved device state
    DeviceState state = *ajileSystem->GetDeviceState(dmdComponentIndex);
    // output the list of temperatures from the device state
    for (u8 i=0; i<state.Temperatures().size(); i++)
        cout << state.Temperatures()[i] << endl;
}
```

Listing 9.8: C++ example of retrieving the device states of the connected components, then getting the DMD device state and outputting its temperature list.

9.4.2 Sequence Status

Each time a frame in a sequence is run by a component a status message is returned to the connected PC to inform it that the frame has completed. These are called sequence status messages and they indicate the last completed sequence, sequence item and frame number along with the current repeat count for the sequnece item and sequence. Sequence status message are queued by the Ajile software so that user applications can easily retrieve them and continuously monitor the current sequence status.

Getting the Sequence Status in the GUI

The sequence status messages are automatically updated in the Ajile GUI each time frames are executed by the components. The last completed frame number, sequence item number, and sequence item/sequence repeat counts are shown in the device status window of the Run Environment.

Getting the Sequence Status in the SDK

Sequence status messages are automatically sent to the Ajile SDK driver each time a frame completes on a component. These sequence status messages are received by the `AjileSystem` and are queued in the `ControllerDriver`. There is a queue of sequence status messages for each connected component in the system. These are stored as `SequenceStatusValues` objects and include the sequence ID, sequence repeat count, sequence item index, sequence item repeat count, and frame index of the frame which has completed on the component.

The next `SequenceStatusValues` in the queue for a given component retrieved with the function `ControllerDriver.GetNextSequenceStatus()`, which takes as an argument the index of that component. The queue of `SequenceStatusValues` is of finite size and when it is full the new `SequenceStatusValues` is added to the queue while the oldest `SequenceStatusValues` is removed. The maximum size of the `SequenceStatusValues` queue can be changed with `ControllerDriver.SetSequenceStatusMaxQueueSize()`. We can check if the queue is empty or not with `ControllerDriver.IsSequenceStatusQueueEmpty()` or we can wait for the next `SequenceStatusValues` to arrive with `ControllerDriver.WaitForSequenceStatus()`. An example of checking for and displaying `SequenceStatusValues` is shown in Listing 9.9 and 9.10 in Python and C++, respectively.

```
def SequenceStatusExample(ajileSystem):
    dmdComponentIndex = 1
    driver = ajileSystem.GetDriver()
    # while the DMD component is in the running state
    while ajileSystem.GetDeviceState(dmdComponentIndex).RunState() == RUN_STATE_RUNNING:
        # check if the sequence status queue is empty
        if not driver.IsSequenceStatusQueueEmpty():
            # get and output the next sequence status from the queue
            sequenceStatus = driver.GetNextSequenceStatus(dmdComponentIndex)
            print sequenceStatus.SequenceID() + " " \
                + sequenceStatus.SequenceRepeat() + " " \
                + sequenceStatus.SequenceItemIndex() + " " \
                + sequenceStatus.SequenceItemRepeat() + " " \
                + sequenceStatus.FrameIndex()
        else:
            # the sequence status queue is empty, so wait
            driver.WaitForSequenceStatus()
```

Listing 9.9: Python example of retrieving and outputting the sequence status values from the device while the component is running the sequence.

```
void SequenceStatusExample(AjileSystem* ajileSystem) {
    u8 dmdComponentIndex = 1;
    ControllerDriver& driver = *ajileSystem->GetDriver();
    // while the DMD component is in the running state
    while (ajileSystem->GetDeviceState(dmdComponentIndex)->RunState() == RUN_STATE_RUNNING) {
        // check if the sequence status queue is empty
        if (!driver.IsSequenceStatusQueueEmpty()) {
            // get and output the next sequence status from the queue
            SequenceStatusValues sequenceStatus =
                driver.GetNextSequenceStatus(dmdComponentIndex);
            cout << sequenceStatus.SequenceID() << " "
                 << sequenceStatus.SequenceRepeat() << " "
                 << sequenceStatus.SequenceItemIndex() << " "
                 << sequenceStatus.SequenceItemRepeat() << " "
                 << sequenceStatus.FrameIndex() << endl;
        } else
            // the sequence status queue is empty, so wait
            driver.WaitForSequenceStatus();
    }
}
```

Listing 9.10: C++ example of retrieving and outputting the sequence status values from the device while the component is running the sequence.

9.5 Streaming Sequences

The simplest way to run a sequence which we have seen up to this point is to preload the entire sequence along with all required images to the device in advance, then start the sequence after the entire project has been loaded to the device. This works well for the majority of sequences since most Ajile devices have 1 GB of on board memory which is plenty of storage most of the time. There are cases however where we need to run sequences that are larger than the on board memory store. In addition, it may be necessary to dynamically change frames and image data on the fly while the sequence is running, for example based on the output and analysis of images from a camera the DMD images need to be adapted. For these cases we move from the simple preloaded sequence approach and look at streaming sequences which allow an unlimited number of images to be streamed continuously from a PC.

The Ajile software suite makes streaming sequences easy to implement as all flow control and time synchronization is handled internally by the system. A first-in first-out (FIFO) queue interface is made available in the Ajile system which allows sequence items to be added to the queue. While the steraming sequence is running, the streaming sequence items are taken from the front of the queue one by one and sent to the device to be run, while user applications keep the queue topped up by continually adding streaming sequence items to the back of the queue.

9.5.1 Running Streaming Sequences in the SDK

Loading and running streaming sequences varies somewhat from preloaded sequences. Listing 9.11 and 9.12 in both Python and C++ shows an example of running a streaming sequence with images that are generated dynamically using OpenCV. First a streaming sequence is created by specifying its sequence type as `SEQ_TYPE_STREAM` which lets the Ajile SDK know how the sequence items in the sequence will be run (Python lines 9-14, C++ lines 11-16). Then a loop is run which will continually top up the streaming sequence items in the streaming queue of the `ControllerDriver` (Python lines 17-40, C++ lines 19-45). The function `ControllerDriver.GetNumStreamingSequenceItems()` tells the program how much space is available in the outgoing streaming sequence item queue (Python line 19, C++ line 21). If the number of items in the queue is less than our alotted amount a new streaming sequence item is created and added to the queue. OpenCV is used to draw a new image which has a 32-bit counter drawn on the image (Python lines 20-24, C++ lines 22-27). The OpenCV/NumPy image is converted to an Ajile Image object as we have previously seen in Chapter 5 (Python lines 25-28, C++ lines 28-32). Then a new sequence item and frame are newly created which will be streamed to the device (Python lines 29-31, C++ lines 32-34). Notice that the image ID passed into the `Frame` constructor is zero - this is because rather than the frame using an image which is in the project preloaded image store, the frame will instead use a streaming image which is attached to the frame itself. This process of attaching a streaming image to a frame is done with `Frame.SetStreamingImage` (Python line 33, C++ line 36). Images which are attached to streaming frames/sequence items are valid only for the lifetime of the sequence item. When a streaming sequence item has been run by the component the sequence item along with its frames and any streaming images are discarded. Finally, the newly created streaming sequence item is added to the queue of the `ControllerDriver` with `ControllerDriver.AddStreamingSequenceItem()` (Python line 36, C++ line 39).

Note that the sequence is started in the usual way with `ControllerDriver.StartSequence()`, however the example delays the start of the sequence until the streaming sequence item queue has been sufficiently filled (Python lines 38-39, C++ lines 41-43). Also note that for streaming sequences the actual frame rate of the sequence will be limited by the transfer rate of the communication type between the PC and the device. For example, Gigabit Ethernet and USB 2 typically achieve data rates of only a few hundred Mb/s which correspond to a few hundred frames per second for a DMD 4500 device. If frame rates which are faster than the communcation channel data rate are specified then the device will inevitably run out of data part way through the streaming sequence. When this happens one of two things can happen on the device which the user application has control of. The device can stop the currently running sequence when it runs out of streaming sequence items, or it can repeat the last streaming sequence item which it has received in its queue indefinitely until new sequence items are received. The out of data behavior of a sequence is controlled by setting the out of data action of the sequence using `Sequence.SetOutOfDataAction`. By setting the out of data action to `RUN_STATE_STOPPED` the sequence stops when it runs out of streaming sequence items, while setting it to `RUN_STATE_PAUSED` will repeat the last sequence items when it runs out of data. In the example the out of data behavior is set in the sequence constructor to `RUN_STATE_PAUSED` (Python line 11, C++ line 13).

```
runningStream = False
def StreamingSequenceExample(ajileSystem, myProject):
    dmdSequenceID = 1
    dmdComponentIndex = 1
    counter = 0
    sequenceRunning = False
    driver = ajileSystem.GetDriver()
    maxStreamingSequenceItems = driver.GetStreamingSequenceItemQueueSize()
    # create a streaming sequence and add it to the project
    myProject.AddSequence(
        Sequence(dmdSequenceID, "Streaming Sequence",
                 DMD_4500_DEVICE_TYPE, SEQ_TYPE_STREAM,
                 1, SequenceItemList(), RUN_STATE_PAUSED))
    driver.LoadProject(myProject)
    # add streaming sequence items in a continuous loop
    runningStream = True
    while runningStream:
        # check if the queue is full
        if driver.GetNumStreamingSequenceItems(dmdComponentIndex) < maxStreamingSequenceItems:
            # create an NumPy image the size of the DMD4500
            npImage = np.zeros(shape=(DMD_IMAGE_HEIGHT_MAX, DMD_IMAGE_WIDTH_MAX), dtype=np.uint8
    )
            # draw text on the NumPy image with OpenCV
            cv2.putText(npImage, str(counter), (50, 450), cv2.FONT_HERSHEY_PLAIN, 5, 255, 5)
            counter++
            # create an Image object
            streamingImage = Image()
            # convert the OpenCV image to an Ajile Image
            streamingImage.ReadFromMemory(npImage, 8, ROW_MAJOR_ORDER, DMD_4500_DEVICE_TYPE)
            # create a new sequence item and frame to be streamed
            streamingSeqItem = SequenceItem(dmdSequenceID, 1)
            streamingFrame = Frame( dmdSequenceID, 0, 10000000, 0, 0, DMD_IMAGE_WIDTH_MAX,
    DMD_IMAGE_HEIGHT_MAX)
            # attach the next streaming image to the streaming frame
            streamingFrame.SetStreamingImage(streamingImage)
            streamingSeqItem.AddFrame(streamingFrame)
            # add the streaming sequence item to the queue
            driver.AddStreamingSequenceItem(streamingSeqItem, dmdComponentIndex)
        elif not sequenceRunning:
            # start the sequence once the queue has been filled
            driver.StartSequence(dmdSequenceID, dmdComponentIndex)
            sequenceRunning = True
```

Listing 9.11: Python example of sending a streaming sequence.

```cpp
bool runningStream = false;
void StreamingSequenceExample(AjileSystem* ajileSystem, Project myProject) {
    u16 dmdSequenceID = 1;
    u8 dmdComponentIndex = 1;
    u32 counter = 0;
    char counterStr[32];
    bool sequenceRunning = false;
    cv::Mat cvImage;
    ControllerDriver& driver = *ajileSystem->GetDriver();
    u32 maxStreamingSequenceItems = driver.GetStreamingSequenceItemQueueSize();
    // create a streaming sequence and add it to the project
    myProject.AddSequence(
        Sequence(dmdSequenceID, "Streaming Sequence",
                 DMD_4500_DEVICE_TYPE, SEQ_TYPE_STREAM,
                 1, deque<SequenceItem>(), RUN_STATE_PAUSED));
    driver.LoadProject(myProject);
    // add streaming sequence items in a continuous loop
    runningStream = true;
    while (runningStream) {
        // check if the queue is full
        if ( driver.GetNumStreamingSequenceItems(dmdComponentIndex) < maxStreamingSequenceItems) {
            // create an OpenCV image the size of the DMD4500
            cvImage = cv::Mat::zeros(DMD_IMAGE_HEIGHT_MAX, DMD_IMAGE_WIDTH_MAX, CV_8U);
            // draw text on the OpenCV image
            sprintf(counterStr, "%08x", counter);
            cv::putText(cvImage, counterStr, cv::Point(50, 450), cv::FONT_HERSHEY_TRIPLEX, 5, 255, 5);
            counter++;
            // create an Image object
            Image streamingImage;
            // convert the OpenCV image to an Ajile Image
            streamingImage.ReadFromMemory((u8*)cvImage.data, cvImage.rows, cvImage.cols, 1, 8,
     ROW_MAJOR_ORDER, DMD_4500_DEVICE_TYPE);
            // create a new sequence item and frame to be streamed
            SequenceItem streamingSeqItem = SequenceItem(dmdSequenceID, 1);
            Frame streamingFrame = Frame( dmdSequenceID, 0, 10000000, 0, 0, DMD_IMAGE_WIDTH_MAX,
     DMD_IMAGE_HEIGHT_MAX);
            // attach the next streaming image to the streaming frame
            streamingFrame.SetStreamingImage(streamingImage);
            streamingSeqItem.AddFrame(streamingFrame);
            // add the streaming sequence item to the queue
            driver.AddStreamingSequenceItem(streamingSeqItem, dmdComponentIndex);
        } else if (!sequenceRunning) {
            // start the sequence once the queue has been filled
            driver.StartSequence(dmdSequenceID, dmdComponentIndex);
            sequenceRunning = true;
        }
    }
}
```

Listing 9.12: C++ example of sending a streaming sequence.

Chapter 10

Color and Grayscale Display

Natively the DMD is a purely binary device where each micromirror (pixel) can have two possible states, '1' (on) or '0' (off). In order to display images which higher bit depths than 1-bit we take advantage of the extremely high speed of the DMD to modulate each of the n bitplanes of an n-bit image according to their bit position.

As an example, consider an 8-bit grayscale image where each pixel in the image can be of $2^8 - 1$ possible gray levels. To display this 8-bit image with our binary DMD device we actually split the 8-bit image into 8 separate 1-bit images, one for each bit, called bitplanes. We then display each of the 1-bit bitplanes separately by the DMD one after another in a sequence for an amount of time proportional to their bit position. Thus the most significant bitplane will be displayed for $2^8/2$ units of time, the second most significant bitplane will be displayed for $2^8/4$ time units, and so on down to the least significant bit which is displayed for $2^8/256 = 1$ time unit.

Since DMDs have a minimum frame time which is limited by the data loading rate into the DMD, it is often the case that an individual time unit is too long in order to achieve reasonably high-speed grayscale or color display times. For example, the DMD 4500 device has a minimum frame time of 150 μs (or around 6600 frames/second). The 8 bitplanes of an 8-bit grayscale image would need to displayed for a total of $2^8 = 256$ time units, and so with a 150 μs minimum time unit we would end up with a minimum 8-bit grayscale image time of $256 \times 150\mu s = 38.4$ ms (or around 26 Hz). To achieve faster grayscale/color display times we therefore need to control the LED power in addition to the bitplane frame times to obtain time units which are less than the minimum DMD frame time. The simplest way to control the LED power is to reduce the LED on time as a fraction of the DMD frame time, since the Ajile LED controller is capable of turning the LEDs on for very short bursts (i.e. around 10 μs). In addition, LED current can also be controlled per frame to give even more control and performance for accurate display of grayscale and color.

Color images are displayed by the DMD by sequentially enabling the red, green and blue LEDs synchronized with the display of each of the RGB color channels of the image. The simplest type of color image is a 3-bit color image, where each color channel is a single 1-bit bitplane. In this case the 3 channels are simply displayed by the DMD sequentially, and the red, green and blue LEDs are enabled for the corresponding channel. For higher bit-depth color images, each channel is split into its individual bitplanes and displayed in the same way as a grayscale image but only using one of the LEDs per channel.

Fortunately the Ajile software suite is equipped with powerful tools which make it easy to automatically import and display grayscale and color images of arbitrary bit depths while keeping precise linearity of image intensity levels. The Ajile software allows users to split color and grayscale images into their individual list of bitplanes, and is also equipped with optimization functions which determine the ideal frame times and LED powers for each individual bitplane and arranges them in the correct order in a sequence to enable the proper displayed grayscale or color image by the DMD.

10.1 Displaying Color/Grayscale as a List of Bitplanes

There are two main aspects of displaying multi-bit (grayscale or color) images by a 1-bit binary DMD device. First we must split the multi-bit composite image into multiple 1-bit bitplanes so that they are compatible with the DMD. Second we must arrange the multiple 1-bit bitplanes into a sequence of frames with correct timing and lighting instructions to display the correct intensity level for every given pixel location.

10.1.1 Splitting Multi-Bit Images into Bitplanes

As we saw in Chapter 5, multi-bit images in row-major order are arranged starting from the top left pixel at row 0 and column 0, followed by the pixel to its right at row 0 and column 1, and so on. For an n-bit image, the n bits of each pixel are stored together at each pixel location. With the notation of $p_j b_i$ denoting the ith bit of pixel number j, an example 8-bit image with N columns and M rows would be arranged according to the matrix in equation 10.1 as follows:

$$\begin{bmatrix} p_0b_0 & p_0b_1 & \dots & p_0b_7 & p_1b_0 & p_1b_1 & \dots & p_{N-1}b_7 \\ p_Nb_0 & p_Nb_1 & \dots & p_Nb_7 & p_{N+1}b_0 & p_{N+1}b_1 & \dots & p_{2N-1}b_7 \\ \vdots & \vdots & \ddots & \vdots & \vdots & \vdots & \ddots & \vdots \\ p_{N(M-1)}b_0 & p_{N(M-1)}b_1 & \dots & p_{N(M-1)}b_7 & p_{(N+1)(M-1)}b_0 & p_{(N+1)(M-1)}b_1 & \dots & p_{NM-1}b_7 \end{bmatrix} \tag{10.1}$$

That is, each of the 8 bits of pixel 0 are listed, followed by the 8 bits of pixel 1 and so on up to the 8 bits of pixel $N-1$, followed by the 8 bits of pixel N which is at row 1, and so on. To split this image into each of its 8 bitplanes, we mean to create 8 × 1-bit images for each of the corresponding 8 bitplanes of the original image. For this example we would therefore have 8 individual bitplane images according to the matrix in 10.2 as follows:

$$\begin{bmatrix} p_0b_i & p_1b_i & p_2b_i & p_3b_i & \dots & p_{N-1}b_i \\ p_Nb_i & p_{N+1}b_i & p_{N+2}b_i & p_{N+3}b_i & \dots & p_{2N-1}b_i \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ p_{N(M-1)}b_i & p_{(N+1)(M-1)}b_i & p_{(N+2)(M-1)}b_i & p_{(N+3)(M-1)}b_i & \dots & p_{NM-1}b_i \end{bmatrix} \text{ where } i = 0, 1, \dots, 7 \tag{10.2}$$

The Ajile software suite has functions which easily take care of splitting n-bit images such as in Equation 10.1 into n individual 1-bit images such as in Equation 10.2. The bit-depth n can be an arbitrary depth, for example 1, 4, 8, 10, 12, 16 and even higher bit depths are fully supported with the tools.

10.1.2 Displaying Bitplanes of n-Bit Images

As was discussed in the introductory discussion, to display an n-bit grayscale image we must display each of the 1-bit bitplanes of the image sequentially by the binary DMD. In order to obtain the correct gray levels the most significant bitplanes must be displayed proportionally longer than the least significant bitplanes.

In the Ajile software suite, the n bitplanes of an n-bit image are arranged into a Sequence Item which is composed of n Frames (we have seen Sequence Items and Frames in detail in Chapter 6.) The i-th Frame in the color or grayscale Sequence Item therefore refers to the i-th bitplane in the n-bit composite image. The Ajile software facilitates creating color or grayscale Sequence Items where each of the n Frames have frame times and LED powers which are proportional to the bitplane location of the Frame's bitplane.

10.1.3 Grayscale Display: Frame Time Control Only

The simplest case for grayscale display is when the total display time of the image can be obtained by modulating the DMD frame times alone. This means that for this case the frame time for the least

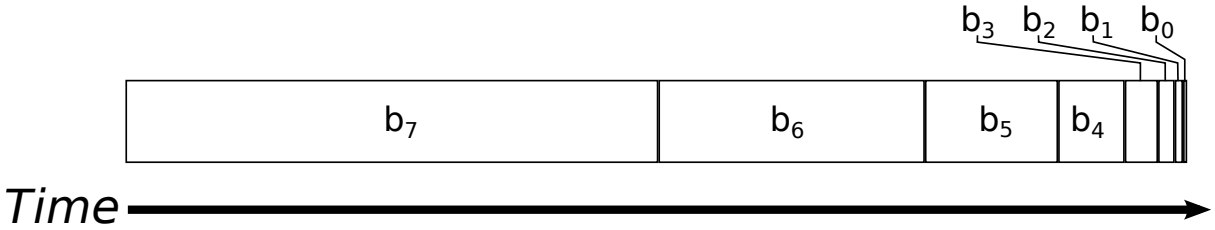

Figure 10.1: Proportional frame timing of the 8 bitplanes of an 8-bit image.

Bitplane Number	Frame Time	Red LED Time	Red LED Current	Green LED Time	Green LED Current	Blue LED Time	Blue LED Current
7	19.2	19.2	3500	19.2	5000	19.2	5000
6	9.6	9.6	3500	9.6	5000	9.6	5000
5	4.8	4.8	3500	4.8	5000	4.8	5000
4	2.4	2.4	3500	2.4	5000	2.4	5000
3	1.2	1.2	3500	1.2	5000	1.2	5000
2	0.6	0.6	3500	0.6	5000	0.6	5000
1	0.3	0.3	3500	0.3	5000	0.3	5000
0	0.15	0.15	3500	0.15	5000	0.15	5000

Table 10.1: Generated frame times and LED powers for an 8-bit image with a 38.4 ms display time. All times are in milliseconds and all currents are in milliamps.

significant bitplanes is greater than or equal to the minimum frame time of the DMD (i.e. ≥ 150 μs for the DMD 4500 device). The general principle of the proportional frame times for each of the bitplanes of an 8-bit image is shown in Figure 10.1. Each of the rectangles represents a bitplane of the 8-bit image, and is sized proportional to the amount of time that it will be displayed for by the DMD. The most bitplane at bit 7, b_7, is first with the longest time, followed by bit 6 which is half the time of b_7, then bit 5 which is half the time of b_6, and so on down to the least significant bit b_0 which is the smallest unit of time.

The specific frame times for each of the 8 bitplanes of an 8-bit image where only DMD frame times are used to modulate gray levels is shown in Table 10.1. Notice that the LED times and currents are set to the maximum possible power to fill the entire frame time since they are not needed to achieve the desired 8-bit image display time. Also notice that the LED power for each bitplane is precisely 1/2 the LED power of the previous bitplane so that the most significant frame (bitplane) has 1/2 of the total image power whereas the least significant frame (bitplane) has 1/256 of the total power.

10.1.4 Grayscale Display: Frame Time and LED Power Control

When higher speed grayscale display times are needed it will be necessary to reduce LED times and LED currents to achieve linear gray values for low order bitplanes. With a minimum DMD frame time of 0.150 ms, which is currently the minimum data load time for the DMD 4500 device, the shortest possible 8-bit grayscale display time is therefore 38.4 ms, as shown in Table 10.1. The example in Table 10.2 shows the frame times and LED powers (time $\times$ current) for a 10.0 ms total grayscale display time. Since this is below the minimum 38.4 ms some of the LED powers must be reduced below their maximum possible values. In this example, the most significant bitplanes (from bitplane numbers 2 to 7) have frame times above the minimum frame time of 0.150 ms and the LED times are set to the entire frame time. However, the lower order bitplanes (bitplane numbers 0 and 1) have LED on times which are a fraction of the frame time. Note that this preservation of gray level linearity therefore comes at the cost of slightly reduced total light output for the entire grayscale image.

We have so far seen an example in Table 10.1 where frame times alone are sufficient to achieve linear 8-bit grayscale output, and an example in Table 10.2 where frame time and LED on time are used to

Bitplane Number	Frame Time	Red LED Time	Red LED Current	Green LED Time	Green LED Current	Blue LED Time	Blue LED Current
7	4.927	4.927	3500	4.927	5000	4.927	5000
6	2.464	2.464	3500	2.464	5000	2.464	5000
5	1.232	1.232	3500	1.232	5000	1.232	5000
4	0.616	0.616	3500	0.616	5000	0.616	5000
3	0.308	0.308	3500	0.308	5000	0.308	5000
2	0.154	0.154	3500	0.154	5000	0.154	5000
1	0.15	0.077	3500	0.077	5000	0.077	5000
0	0.15	0.038	3500	0.038	5000	0.038	5000

Table 10.2: Generated frame times and LED powers for an 8-bit image with a 10.0 ms display time. All times are in milliseconds and all currents are in milliamps.

Bitplane Number	Frame Time	Red LED Time	Red LED Current	Green LED Time	Green LED Current	Blue LED Time	Blue LED Current
9	4.774	4.774	3500	4.774	5000	4.774	5000
8	2.387	2.387	3500	2.387	5000	2.387	5000
7	1.194	1.194	3500	1.194	5000	1.194	5000
6	0.597	0.597	3500	0.597	5000	0.597	5000
5	0.298	0.298	3500	0.298	5000	0.298	5000
4	0.15	0.149	3500	0.149	5000	0.149	5000
3	0.15	0.075	3500	0.075	5000	0.075	5000
2	0.15	0.037	3500	0.037	5000	0.037	5000
1	0.15	0.019	3500	0.019	5000	0.019	5000
0	0.15	0.01	3262	0.01	4660	0.01	4660

Table 10.3: Generated frame times and LED powers for an 10-bit image with a 10.0 ms display time. All times are in milliseconds and all currents are in milliamps.

achieve linear 8-bit grayscale at a slightly higher speed. For even higher speeds and/or bit-depths it may also be necessary to reduce LED currents to allow for a greater range of gray levels. The example in Table 10.3 shows the frame time, LED time and LED current used to display a 10-bit grayscale image with a 10 ms display time. In this example, bitplane numbers 1 through 9 are displayed using only the combination of frame time and LED on time to achieve the required gray levels as before. However for bitplane number 0 the LED currents are reduced as well since in this system the LED times cannot be reliably set below the minimum value of 0.01 ms.

10.1.5 Grayscale Display Optimization

As may be evident by the grayscale examples seen in Table 10.1, 10.2 and 10.3, determining the optimal frame time, LED on time and LED current for each of the n bitplanes of an n-bit image in order to display the image with the correct timing is not a trivial task. Fortunately the Ajile software suite provides functions to automatically optimize the frame times and LED powers for a given grayscale or color display time. We will see how to use these functions later in this Chapter, but for now we will just discuss that there are four different parameters that may be optimized, which are under user control as to whether or not they should be used.

Frame Time Adjustment

This is the first parameter that will be adjusted by the Grayscale display optimization functions. Adjusting the frame time of the DMD to obtain different gray levels is by far the most preferable option since the LED power is held constant for the entire frame and we get the highest possible light output

Bitplane Number	Frame Time	Red LED Time	Red LED Current	Green LED Time	Green LED Current	Blue LED Time	Blue LED Current
7	0.25	0.25	3500	0.25	5000	0.25	5000
6	0.15	0.125	3500	0.125	5000	0.125	5000
5	0.15	0.063	3500	0.063	5000	0.063	5000
4	0.15	0.031	3500	0.031	5000	0.031	5000
3	0.15	0.016	3500	0.016	5000	0.016	5000
2	0.15	0.01	2737	0.01	3910	0.01	3910

Table 10.4: Generated frame times and LED powers for an 8-bit image with a 1.0 ms display time. Note that only the most significant 6 bitplanes are displayed in order to meet the 1.0 ms timing requirement. All times are in milliseconds and all currents are in milliamps.

efficiency.

LED On Time Adjustment

When frame time adjusted alone is not enough to meet display timing requirements due to the fact that the frame times would need to fall below the minimum permissible frame times, the next most ideal parameter to adjust is the LED on time as a fraction of the frame time. LED on time is the next most preferable free parameter since adjusting LED time is still guaranteed to be very linear with light output and does not affect the temperate and/or wavelength characteristics of the LEDs (as adjusting LED currents would).

LED Current Adjustment

While LED on times can be set as a fraction of the minimum frame time, there is however a minimum permissible LED on time below which the LEDs cannot be reliably switched on at their desired currents. This minimum LED on time is currently 10 μs. When adjusting LED on times alone would result in times below the minimum LED time the grayscale display optimizer can adjust LED currents. This optimization will only occur when frame times and LED times are not sufficient for the desired n-bit grayscale display time since LED light output is not completely linear with drive current and also LED temperatures and wavelengths can be affected by current. However, this LED current setting per bitplane is still an option to display grayscale images with either very high bit depths or high speeds.

Dropping Bitplanes Adjustment

When it simply is not possible to achieve the desired grayscale display time with the given bit depth, the last resort which may or may not be feasible for the given application is to allow the grayscale display optimizer to drop the lower order bitplanes from the image. For example, if we want to display an 8-bit grayscale image with a 1 ms display time this is simply not possible with the amount of frame time, LED time and LED current adjustment available in the DMD 4500 controller. If we allow dropping bitplanes however, we would could achieve the 1 ms grayscale display time if we display the image using only 6 bits instead of 8 (i.e. a 6-bit grayscale image.) The optimized times and currents for such a 6-bit image are shown in Table 10.4 where only bitplanes 2 through 7 of the 8-bit image are displayed in the 1 ms total display time.

10.1.6 Color Display

Up until this point we have looked solely at splitting grayscale images into bitplanes and displaying them. Displaying RGB color images is identical to displaying three separate grayscale images, one for each color channel. Essentially the three color channels are split into three grayscale images for each channel, then the grayscale optimization for each of the three color channels is handled independently in the exact same way as standard grayscale images. The bitplanes for all three color channels are combined into a single sequence item with only one LED powered on for each of the three color channels.

Bitplane Number	Frame Time	Red LED Time	Red LED Current	Green LED Time	Green LED Current	Blue LED Time	Blue LED Current
R_7	1.458	1.458	3500	0	0	0	0
G_7	1.458	0	0	1.458	5000	0	0
B_7	1.458	0	0	0	0	1.458	5000
R_6	0.729	0.729	3500	0	0	0	0
G_6	0.729	0	0	0.729	5000	0	0
B_6	0.729	0	0	0	0	0.729	5000
R_5	0.364	0.364	3500	0	0	0	0
G_5	0.364	0	0	0.364	5000	0	0
B_5	0.364	0	0	0	0	0.364	5000
R_4	0.182	0.182	3500	0	0	0	0
G_4	0.182	0	0	0.182	5000	0	0
B_4	0.182	0	0	0	0	0.182	5000
R_3	0.15	0.091	3500	0	0	0	0
G_3	0.15	0	0	0.091	5000	0	0
B_3	0.15	0	0	0	0	0.091	5000
R_2	0.15	0.046	3500	0	0	0	0
G_2	0.15	0	0	0.046	5000	0	0
B_2	0.15	0	0	0	0	0.046	5000
R_1	0.15	0.023	3500	0	0	0	0
G_1	0.15	0	0	0.023	5000	0	0
B_1	0.15	0	0	0	0	0.023	5000
R_0	0.15	0.011	3500	0	0	0	0
G_0	0.15	0	0	0.011	5000	0	0
B_0	0.15	0	0	0	0	0.011	5000

Table 10.5: Generated frame times and LED powers for an 8-bit, 3-channel color image with a 10.0 ms display time. All times are in milliseconds and all currents are in milliamps.

An example bitplane timing of an 8-bit, 3 channel RGB color image is shown in Table 10.5. Notice that to obtain the three color channels the bitplanes alternate between the red, green and blue bitplanes sequentially, denoted by R_i, G_i and B_i for the i-th bitplane of the red, green and blue channels respectively. Also note that for each of bitplanes of any given channel, only the LED for that corresponding channel is enabled (with a non-zero LED time and current) while the other two channels are off.

10.2 Displaying Color and Grayscale Images

The first step to working with color and grayscale images is to split them into 1-bit bitplanes so that they are compatible with the native format of the DMD.

10.2.1 Displaying Color and Grayscale Images in the GUI

Displaying color and grayscale images in the GUI involves two steps: first we import the color or grayscale image into the project, and second we assign the image to a sequence item in a sequence so that it can be displayed.

Importing Color and Grayscale Images

Importing color and grayscale images in the GUI is nearly identical to the steps which we have already seen in Section 5.3.1. The only thing to keep in mind is to select the 'Output Image Type' in the drop down menu to be the output image format that is desired. That is, select '8-Bit Grayscale' to import the image as grayscale or '24-Bit Color' to import the image as color, as is shown in Figure 10.2. At this time

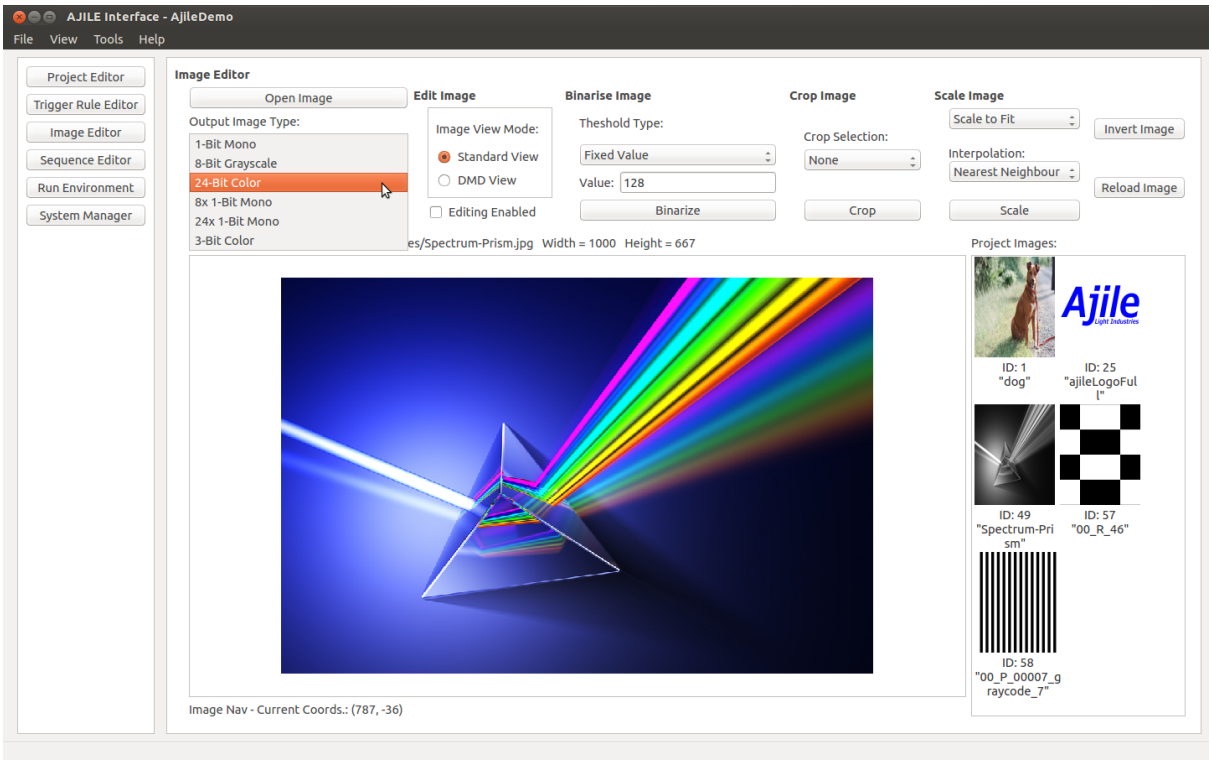

Figure 10.2: Screenshot of selecting the output image type in the Image Editor of the Ajile GUI.

the GUI only support 8-bit grayscale and 24-bit color (i.e. 3 channels of 8-bits each). For non-standard bit depths to SDK will need to be used, see the next section.

Creating Color and Grayscale Sequence Items

With grayscale or color images loading into the project, the next step is to assign the images to sequence items in a sequence. Previously in Chapter 6 we showed how to assign binary (1-bit) images to individual frames in a sequence. Color and grayscale images on the other hand are assigned to sequence items rather than frames, the reason being that color and grayscale images are actually composed of several frames, one for each bitplane of the composite image.

To create a color or grayscale sequence item for display we first open the Sequence Editor and create a new sequence and an empty sequence item (see Chapter 6). We then assign an image to the sequence item by right clicking on the sequence item which will display the image and selecting 'Add/Change Image' from the pop-up menu, as shown in Figure 10.3. The Image Selection dialog will then appear, as shown in Figure 10.4. Select a color or grayscale image and click OK. This will assign the image to the sequence item and automatically creates frames within the sequence item for each of the corresponding bitplanes of the image. As shown in Figure 10.5, the image thumbnail of the sequence item is updated to show the color or grayscale image.

Note that by default the color or grayscale sequence items are shown in collapsed mode, meaning that the frames within them are hidden. If needed, the sequence item can be expanded (by clicking on the arrow to the left of the sequence item) and the list of frames for each of the bitplanes are shown beneath the sequence item. The frames are automatically created with the lighting and timing parameters needed to correctly display the composite image.

The total frame time of the sequence item (and thus the total time that the color/grayscale image will be displayed) is shown under the Frame Time column of the sequence editor. This frame time can be edited to set the target time for which the sequence item / image will be displayed. A specific display time can be entered here and the frame times within the sequence item will automatically be adjusted

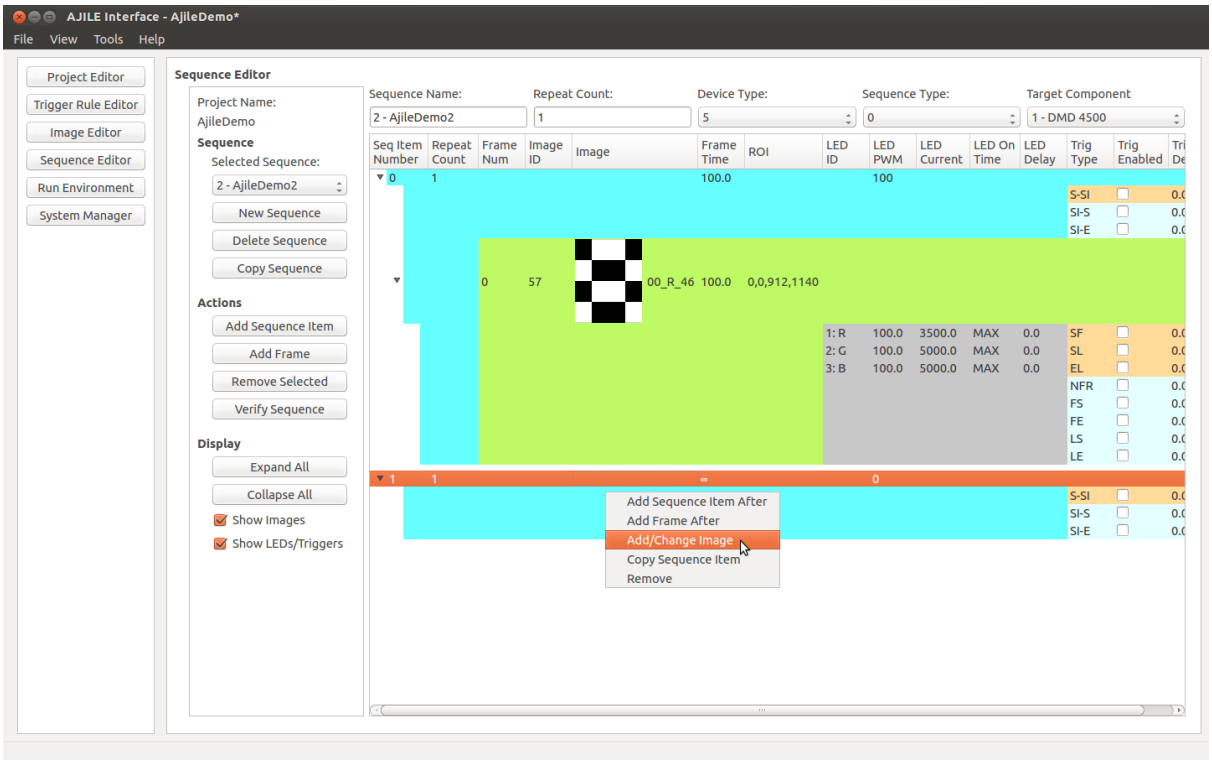

Figure 10.3: Screenshot of setting the color or grayscale image of a sequence item by right-clicking on its row and selecting 'Add/Change Image'.

to give the new time for the sequence item. Special values of '-1' and '0' can also be entered, and are described by the tool-tip shown in Figure 10.5. A setting of '0' will give the sequence item the minimum possible display time and allow LED on times and currents to be modulated below their maximums, whereas a setting of '-1' will give the sequence item the minimum possible display time while keeping the LEDs at their maximum outputs.

10.2.2 Displaying Color and Grayscale Images in the SDK

There are three possible ways to create color and grayscale sequence items for display depending on user requirements. They can be automatically created from standard image file formats (e.g. '.png', '.bmp', etc.) which are located on disk, they can be automatically created from existing Ajile Images which reside in program memory, or they can be created with full user control over the splitting of composite images into their individual bitplanes and the assignment of bitplanes to color/grayscale sequence items.

Creating Color and Grayscale Sequence Items From Image Files

The easiest way to create color or grayscale sequence items from the SDK is to simply specify an input image filename and let the SDK automatically split the image into its bitplanes and assign the bitplanes to frames in a sequence item with optimized frame timing and lighting parameters. To accomplish this there are functions available within the Project object to facilitate converting color and grayscale images into bitplanes and sequence items which can be displayed by the DMD device. These functions are

`Project.CreateGrayscaleSequenceItem_FromImage()`,
`Project.CreateGrayscaleSequenceItemWithTime_FromImage()`,

for grayscale conversion, and

`Project.CreateColorSequenceItem_FromImage()`,
`Project.CreateColorSequenceItemWithTime_FromImage()`,

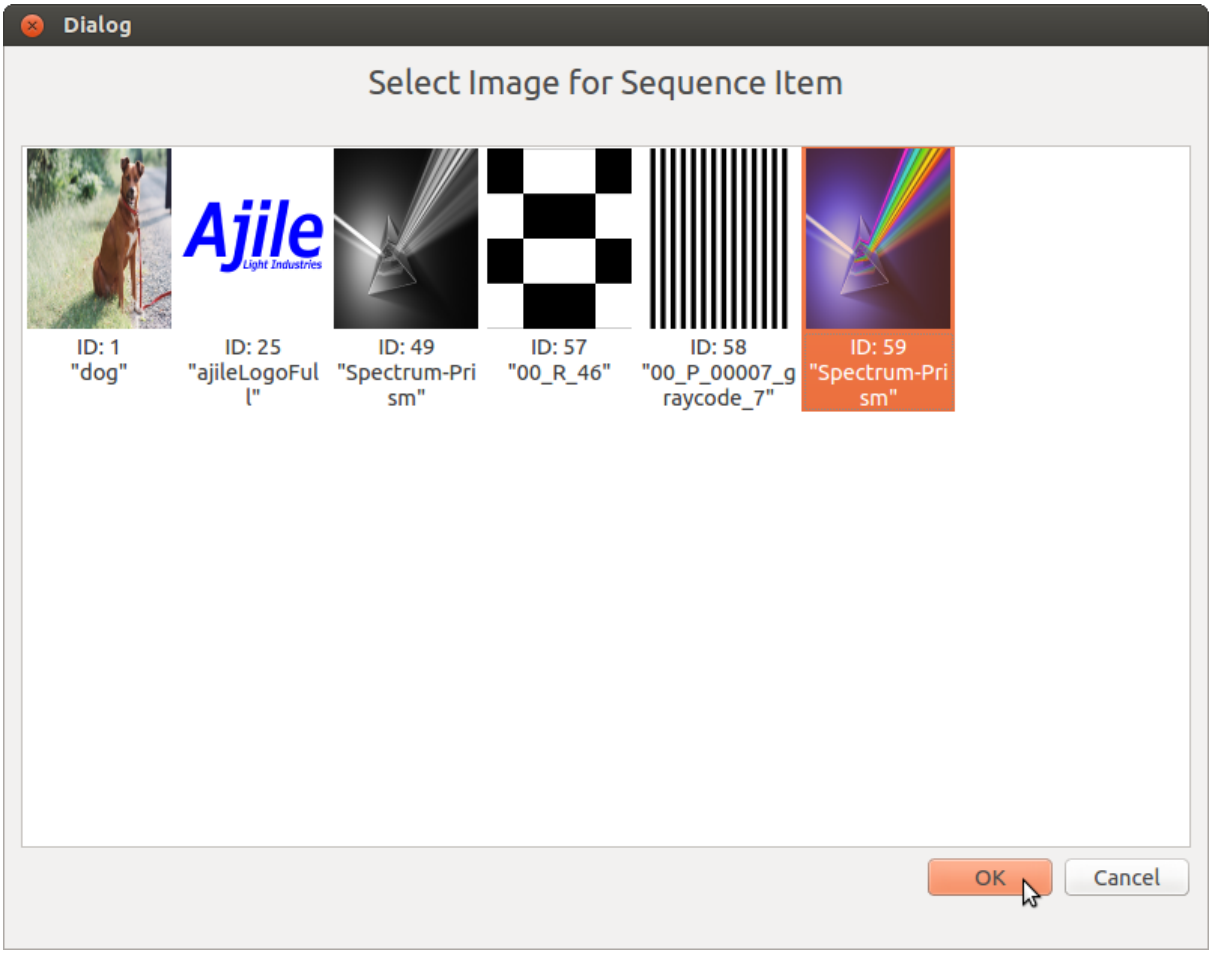

Figure 10.4: Screenshot of setting the color or grayscale image of a sequence item by right-clicking on its row and selecting 'Add/Change Image'.

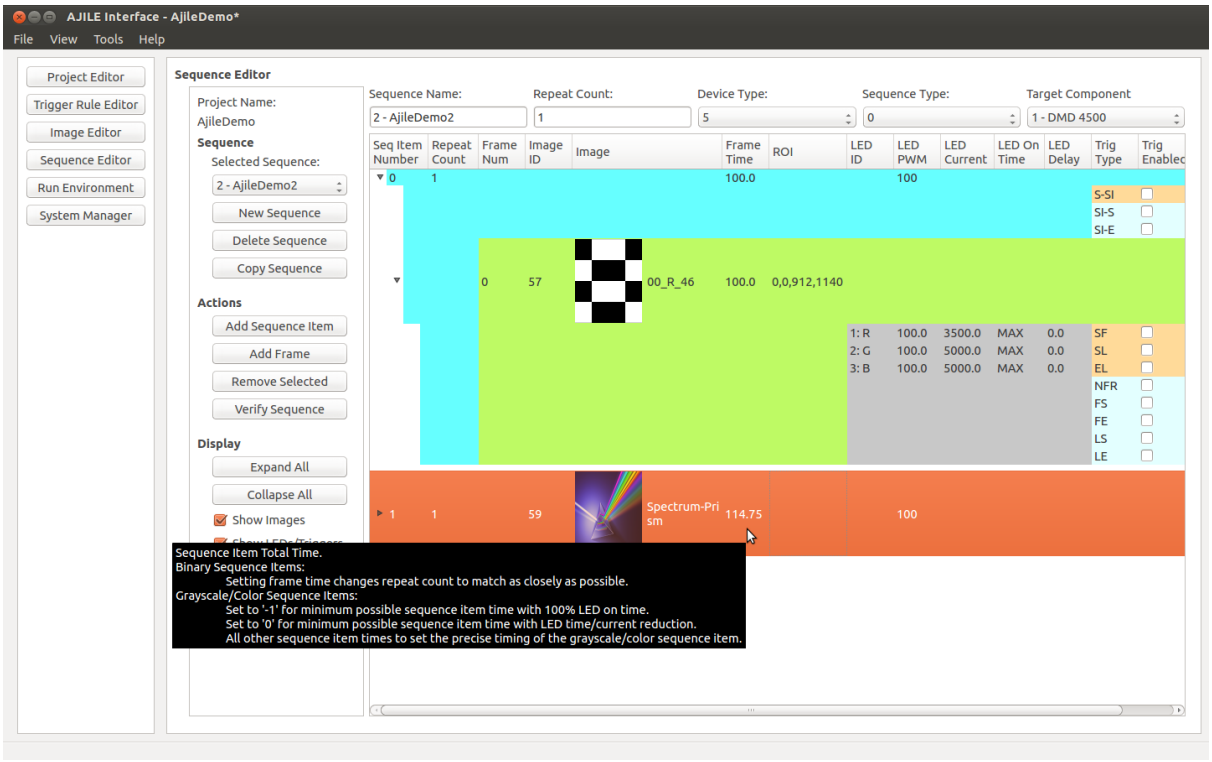

Figure 10.5: Screenshot of a color image attached to a sequence item.

for color conversion. For the '`WithTime`' versions of the function a target total time for the color/grayscale sequence item is specified, whereas for the other versions of the function the minimum possible total time for the sequence item will be used. Note that the target total time *is not necessarily the same* as the display time of the color or grayscale image, but instead defines the refresh rate of the image. The display time of the image can be configured by setting the repeat count of the color/grayscale sequence item.

A simple example demonstrating the use of the color and grayscale sequence item creation functions is shown in Listing 10.1 in Python and in Listing 10.2 in C++. The example function takes as arguments a project, sequence ID, image ID, input image filename and optional arguments to control which of the four color or grayscale functions listed above will be used. After calling the function the input project will have a color or grayscale sequence item added to it, along with the list of bitplanes for that image. An optional total frame time and display time can also be specified. The total frame time control how the sequence item is constructed to get this target time, whereas the display time controls the sequence item repeat count so that it is displayed for the correct amount of time. Note that more advanced usage of these functions is possible - please see the SDK reference manual for further details.

```python
def addCompositeToProject(myProject, sequenceID, imageID, filename, color, frameTimes=-1, displayTime=-1):
    # define an empty sequence item and list of images which will be populated with the new frames and bitplanes
    sequenceItem = SequenceItem(sequenceID)
    imageBitplanes = ImageList()
    if color:
        if frameTimes > 0:
            # create a sequence item to display the 24 bitplanes of a color image with the given timing
            myProject.CreateColorSequenceItemWithTime_FromFile(sequenceItem, imageBitplanes, filename, imageID,
 FromMSec(frameTimes))
        else:
            # create a sequence item to display the 24 bitplanes of a color image with the default minimum timing
            myProject.CreateColorSequenceItem_FromFile(sequenceItem, imageBitplanes, filename, imageID)
    else:
        if frameTimes > 0:
            # create a sequence item to display the 8 bitplanes of a grayscale image with the given timing
            myProject.CreateGrayscaleSequenceItemWithTime_FromFile(sequenceItem, imageBitplanes, filename,
 imageID, FromMSec(frameTimes))
        else:
            # create a sequence item to display the 8 bitplanes of a grayscale image with the default minimum timing
            myProject.CreateGrayscaleSequenceItem_FromFile(sequenceItem, imageBitplanes, filename, imageID)
    # set the display time of this sequence item by setting its repeat time
    sequenceItem.SetRepeatTimeMSec(displayTime)
    # add the image bitplanes to the project
    myProject.AddImages(imageBitplanes)
    # add the sequence item to the project
    myProject.AddSequenceItem(sequenceItem)
```

Listing 10.1: Python example of creating color and grayscale bitplanes and sequence items from an input color or grayscale file, and adding them to the project.

```cpp
void addCompositeToProject(Project& myProject, u16 sequenceID, u16 imageID, const char* filename, bool color, int
 frameTimes=-1, int displayTime=-1) {
    // define an empty sequence item and list of images which will be populated with the new frames and bitplanes
    SequenceItem sequenceItem(sequenceID);
    vector<Image> imageBitplanes;
    if (color)
        if (frameTimes > 0)
            // create a sequence item to display the 24 bitplanes of a color image with the given timing
            myProject.CreateColorSequenceItemWithTime_FromFile(sequenceItem, imageBitplanes, filename, imageID,
 FromMSec(frameTimes));
        else
            // create a sequence item to display the 24 bitplanes of a color image with the default minimum timing
            myProject.CreateColorSequenceItem_FromFile(sequenceItem, imageBitplanes, filename, imageID);
    else
        if (frameTimes > 0)
            // create a sequence item to display the 8 bitplanes of a grayscale image with the given timing
            myProject.CreateGrayscaleSequenceItemWithTime_FromFile(sequenceItem, imageBitplanes, filename,
 imageID, FromMSec(frameTimes));
        else
            // create a sequence item to display the 8 bitplanes of a grayscale image with the default minimum
 timing
            myProject.CreateGrayscaleSequenceItem_FromFile(sequenceItem, imageBitplanes, filename, imageID);
    // set the display time of this sequence item by setting its repeat time
    sequenceItem.SetRepeatTimeMSec(displayTime);
    // add the image bitplanes to the project
    myProject.AddImages(imageBitplanes);
    // add the sequence item to the project
    myProject.AddSequenceItem(sequenceItem);
}
```

Listing 10.2: C++ example of creating color and grayscale bitplanes and sequence items from an input color or grayscale file, and adding them to the project.

10.3 Creating High Bit-Depth (>8-bit) Color and Grayscale Sequence Items

Since the Ajile suite allows for controlling the DMD in its native 1-bit format and specifying frame times and LED settings on a per frame basis, it is possible to display high bit depth images using the Ajile DMD controller since there is no constraint on bit depth as there is with traditional video-based controllers. The only real constraint on bit depth is the required total time for all n bitplanes of an n-bit image to be displayed, which is limited by the minimum possible frame time of the DMD and minimum possible LED times and currents.

The Ajile SDK fully supports creating color and grayscale sequence items with arbitrary bit depths, such as 10-bit, 12-bit, 16-bit and beyond. Of course, for higher bit depth images such as 12-bit and above the minimum total time for all frames of the image to be displayed may be too long to be feasible for certain applications. Currently however 10-bit color can be feasibly displayed at over 80 frames per second and 10-bit grayscale can be displayed at over 240 Hz, with higher bit depths running proportionally slower.

The Ajile SDK interface which facilitates displaying color and grayscale images which are greater than 8 bits in depth is nearly identical to that which has already been shown. The only real differences are that input images of greater than 8-bit deep must be supplied (i.e. 16-bit images), and the resulting list of bitplanes will have more than 8 bitplanes.

10.4 Optimizing the Output Linearity of Color and Grayscale Images

As was discussed earlier in this chapter, displaying multi-bit images with the DMD (which is natively a 1-bit device) is accomplished by displaying the least significant bitplane for 1 time unit and each successive bitplane number i for 2^i time units. When DMD frame time control alone is sufficient to achieve the required bit depth and frame rate then the light output from the DMD is very much linearly proportional to the image pixel intensity values. In other words, an increase of 1 pixel intensity counts results in an increase of 1 light output unit and an increase of 100 pixel intensity counts results in an increase of 100 light output units.

Using frame time control only to achieve n-bit color and grayscale puts a significant limitation on the minimum image display rate however. To overcome this, LED time control below the minimum bitplane frame time must be used. This means that the LEDs are no longer on for 100% of the time and are switched on and off at high speeds in order to increase bit depths and frame rates. While this does give the desired effect of increasing color and grayscale display speed and/or bit depth, it has the unfortunate draw back that LED output is no longer perfectly linear with pixel intensity, due to complex thermal-dependant effects on the LED die.

Fortunately the Ajile software suite is equipped with a way to correct for non-linear light output with pixel intensity. The correction is a list of gain values, one gain value for each bitplane and color channel. When supplied, the frame time and/or LED on time for the given bitplane and color channel is multiplied by the gain value for that bitplane and channel. As an example, Table 10.6 shows the frame times and LED settings for the green channel of an 8-bit grayscale image. In addition, gain values for each of the 8 bitplanes of the green channel are specified. The resulting frame time and LED time after gain correction are shown in the final column.

The graphs in Figure 10.6 are a further demonstration of the usefulness of the gain correction settings. For these graphs, the pixel intensity of a 10-bit image was varied from 0 to 1023 (i.e. $2^{10}-1$) and the light output from the resulting projected image was measured by a light meter. The graph on the left hand side shows the image pixel intensity value versus the light output when no gain corrections were applied (that is, a gain value of 1.0 was applied to each bitplane). Notice that there are steps resulting in a non-straight line. The graph on the right hand side show the light output when a set of gain corrections were applied. We see on the right that the relationship between pixel intensity and light output has been successfully

Bitplane Number	Frame Time	Green LED Time	Green LED Current	Gain Setting	Gain Corrected Frame Time	Gain Corrected LED Time
7	4.927	4.927	5000	1.1	5.410	5.410
6	2.464	2.464	5000	1.3	3.203	3.203
5	1.232	1.232	5000	1.2	1.478	1.478
4	0.616	0.616	5000	1.35	0.832	0.832
3	0.308	0.308	5000	1.3	0.400	0.400
2	0.154	0.154	5000	1.25	0.193	0.193
1	0.15	0.077	5000	1.1	0.15	0.085
0	0.15	0.038	5000	1.0	0.15	0.038

Table 10.6: Generated frame times and LED powers for an 8-bit image with a 10.0 ms display time after applying a set of gain corrections per bitplane.

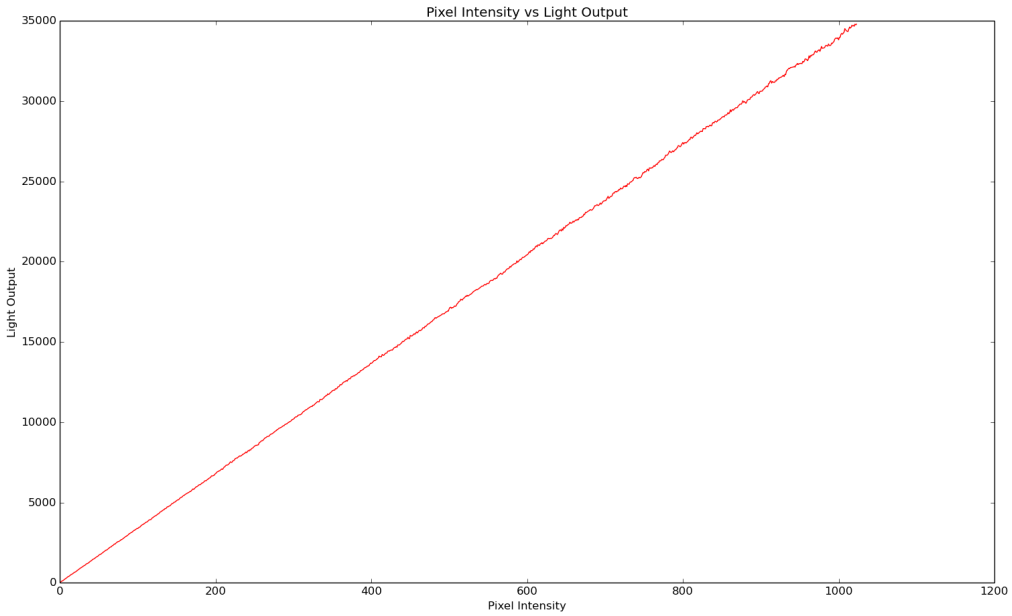

Figure 10.6: Graph of light output with varying pixel intensity.

linearized after applying gain corrections per bitplane. The gain values applied in this example, from most significant to least significant bit, were 1.410, 1.396, 1.40, 1.242, 1.558, 1.586, 1.53, 1.62, 1.259, 1.0.

The function `Project.SetGrayscaleColorGain()` is used to set the gain setting for each channel and bitplane. The functions takes as arguments a list of floating point gain values, one for each bitplane for the output image, along with the channel number to be set. Determining the correct values for the gain settings must be done by measurement with a suitable light meter. If needed, contact Ajile support for more information on this advanced feature.

10.5 Optimizing Color and Grayscale for Human Display

When displaying color and grayscale for non-human display purposes (e.g. when projected images are observed by a camera, or when images do not need to be observed at all), then the problem of total integration time of the multi-bit image by the DMD is not terribly important. However, when the images must be projected for human display then a number of subtleties can arise which can be perceived as 'artifacts' of the DMD display.

The most noticeable display 'artifact' is the apparent flickering of the DMD image as the multiple bitplanes are switched by the DMD at high speed. There are two ways that this flickering artifact can be reduced or eliminated.

1. Increasing the color or grayscale frame rate.
2. Split bitplanes which are displayed for longer periods of time into shorter times, and interleave bitplanes at high speeds.

If performing the splitting and interleaving of bitplanes (which is highly recommended for human display), then the ordering of the bitplanes can also have an impact on display artifacts.

The splitting and interleaving of bitplanes into smaller sub-bitplanes is facilitated by the Ajile SDK and GUI. This splitting is done automatically by the Ajile GUI when a special setting of '-1' is specified for the sequence item frame time (see Figure 10.5). In the SDK, the splitting is controlled by specifying the `maxBitplaneTime` in the `CreateColorSequenceItem()` and `CreateGrayscaleSequenceItem()` functions. See the Ajile SDK reference manual for details on its usage. The ordering of the frames within color or grayscale sequence items are currently not paramaterized by the SDK, but frames can be easily reordered and customized by user applications based on specific needs.

Chapter 11

Camera Control

Ajile cameras use the same project and sequence structure as the DMD components seen previously, and so offer the same level of frame by frame control and programming interface. The real difference between a camera component and a DMD component is that for the camera the images are not loaded from the host PC and sent to the device, but instead they are captured by the device and sent back to the host PC.

This chapter describes the capabilities of Ajile camera components, and the specific interface details for retrieving captured images from the camera.

11.1 Allocating Images

As with DMD components, camera components make use of the random access image store in the project structure which was already seen in Chapter 5. Images must be created and added to the project by the user as before, however the image data itself will typically be empty (i.e. zero size) since it will loaded with captured camera image data when a camera exposure has completed. Images are therefore slots of memory that are reserved ahead of time before running the capture sequence, and the camera will store its captured images at these image slots corresponding to their image ID.

To create camera images, we need to set a number of the Image properties which were shown in Table 5.1. Here we show in Table 11.1 the Image members which must be set, and their typical values for the CMV4000 and CMV2000 camera sensors.

Creating images for the camera is the same as was shown in Chapter 5. We show in Listing 11.1 two ways of creating images that may be used by the camera in C++. The first is the fully manual way where the image width, height, bit depth and number of channels are all specified. Note that constants which

Name	**Possible Values**
ID	1 to 65535
Width	2048
Height	1 to 2048 (CMV4000), 1 to 1088 (CMV2000)
Bit Depth	10, 8
Number of Channels	1
Image Major Order	ROW_MAJOR_ORDER=1
Image Name	Optional and unused
Filename	Unused
Memory Address	0 (will be non-zero after readout)
Size	0 (will be non-zero after readout)

Table 11.1: Image members that must be configured for a camera

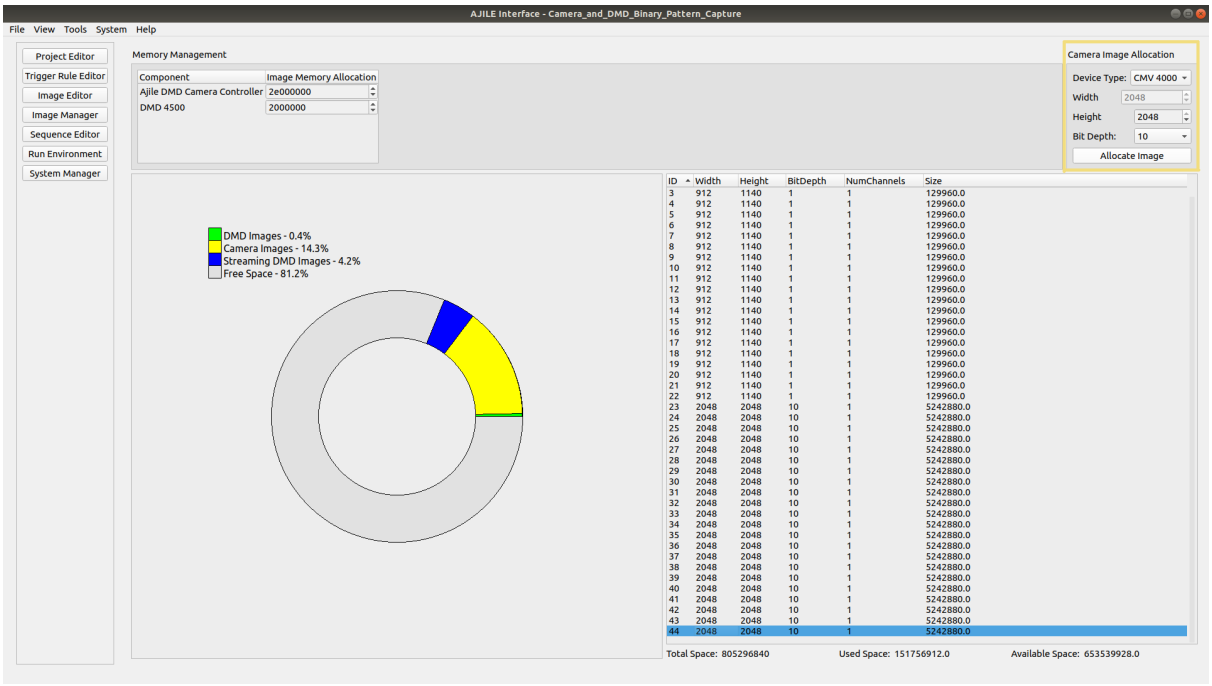

Figure 11.1: A view of the image Manager in the Ajile GUI.

are defined in the Ajile SDK (in camera_constants.h) are used for the width and height. The second way is to use a helper function, `Image.SetImagePropertiesForDevice()`, which automatically sets all image properties to match the given device type. Note also that the image size is set to 0 in the host user application. This is because no image data is associated with the image yet. After a camera image capture sequence has completed and the camera images have been retrieved from the device (which we will see in the next sections), the size and memory address will be non-zero.

```
// create camera image with ID 1 for CMV4000 manually
Image myImage1;
myImage1.SetID(1);
myImage1.SetBitDepth(10);
myImage1.SetNumChannels(1);
myImage1.SetWidth(CMV4000_IMAGE_WIDTH_MAX); // 2048
myImage1.SetHeight(CMV4000_IMAGE_HEIGHT_MAX); // 2048
myImage1.SetSize(0);
// create camera image with ID 2 for CMV4000 using helper function
Image myImage2(2);
myImage2.SetImagePropertiesForDevice(CMV_4000_MONO_DEVICE_TYPE);
Project myProject;
myProject.AddImage(myImage1);
myProject.AddImage(myImage2);
```

Listing 11.1: C++ example of creating two new Image Objects that may be used with a CMV4000 camera component.

11.1.1 Allocating Images in the GUI

The Image Manager is used to allocate camera images in the GUI. A view of the Image Manager can be seen in Figure 11.1. The Image Manager displays the total memory usage of allocate images and allows the user to allocate a new image using the 'Camera Image Allocation' section. The width, height and bit depth are adjustable properties. The width is fixed for certain devices types for compatibility reasons. Pressing the 'Allocate Image' button allocates a new image with the first available image ID. The allocated camera images and the DMD images can be seen in the image list in the Image Manager and the image pane in the Image Editor.

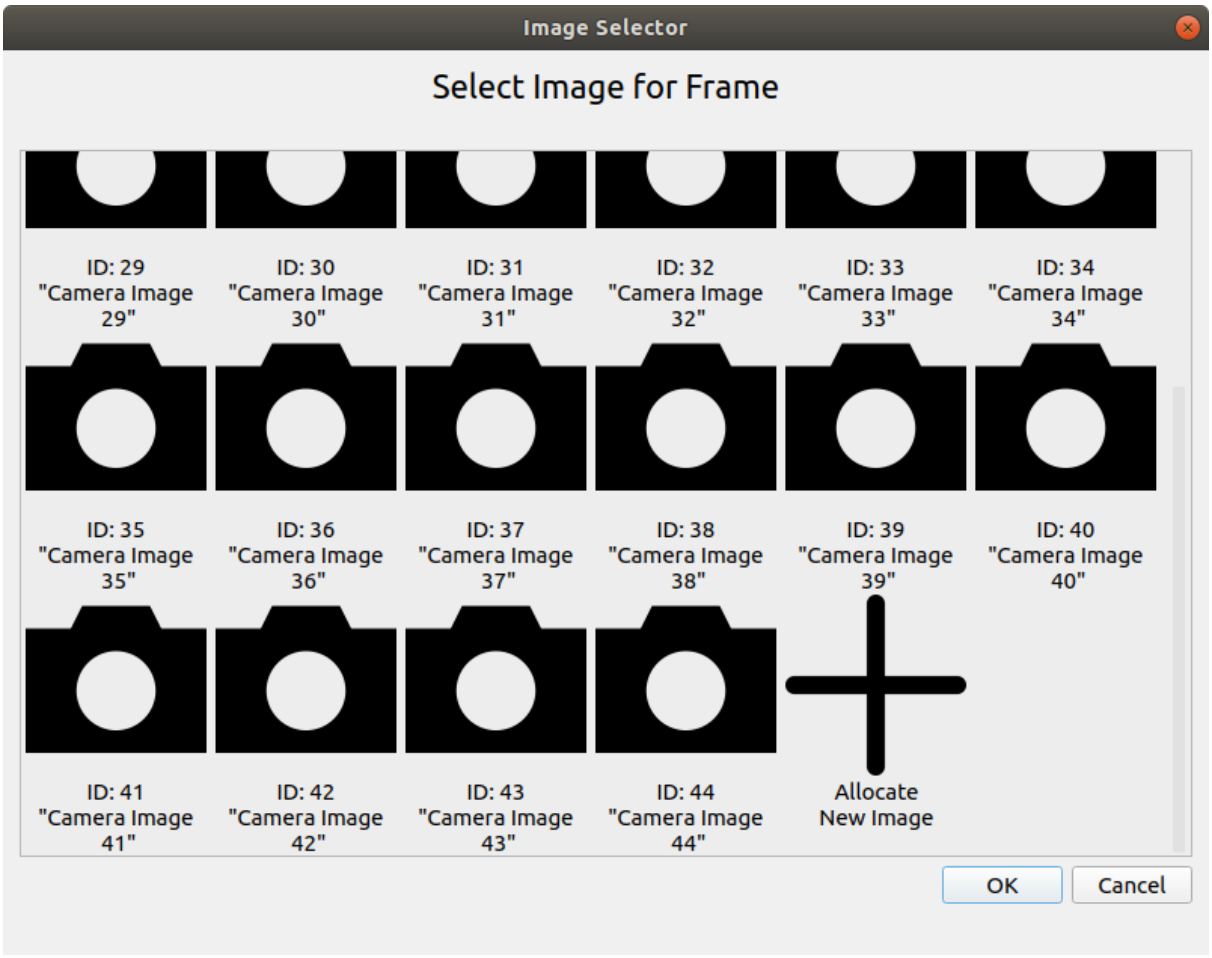

Figure 11.2: A view of the image selector for sequence items and frames.

11.2 Creating Sequences

When the images are added to the project the camera will now have a place to store its image data after the captures have been completed. What is still needed is a way to specify the camera exposures and other capture properties of the camera. This is done with Sequences, as were shown in Chapter 6. Most of the properties of interest are in the Frame object which were shown in Table 6.3. The Frame Image ID specificies which of the Image object slots should be used to store the captured camera image data for that frame. The Frame Time sets the exposure time for that frame. All frame settings, including the additional frame parameters of trigger settings and imaging parameters, can be changed on a frame by frame basis. For example, it is possible to have a sequence with as many different exposure times as desired in order to implement a high dynamic range capture mode.

11.2.1 Creating Sequences in the GUI

Sequences in the GUI are created as described in the sequences Chapter 6. With camera seqeunces, it is also possible to allocate camera images directory in a frame or sequence item by right-clicking the row and pressing 'Add/Change Image'. This will bring up a dialog for selecting the image as seen in Figure 11.2. This allows the user to change the image or allocate a new image directly in the Sequence Editor.

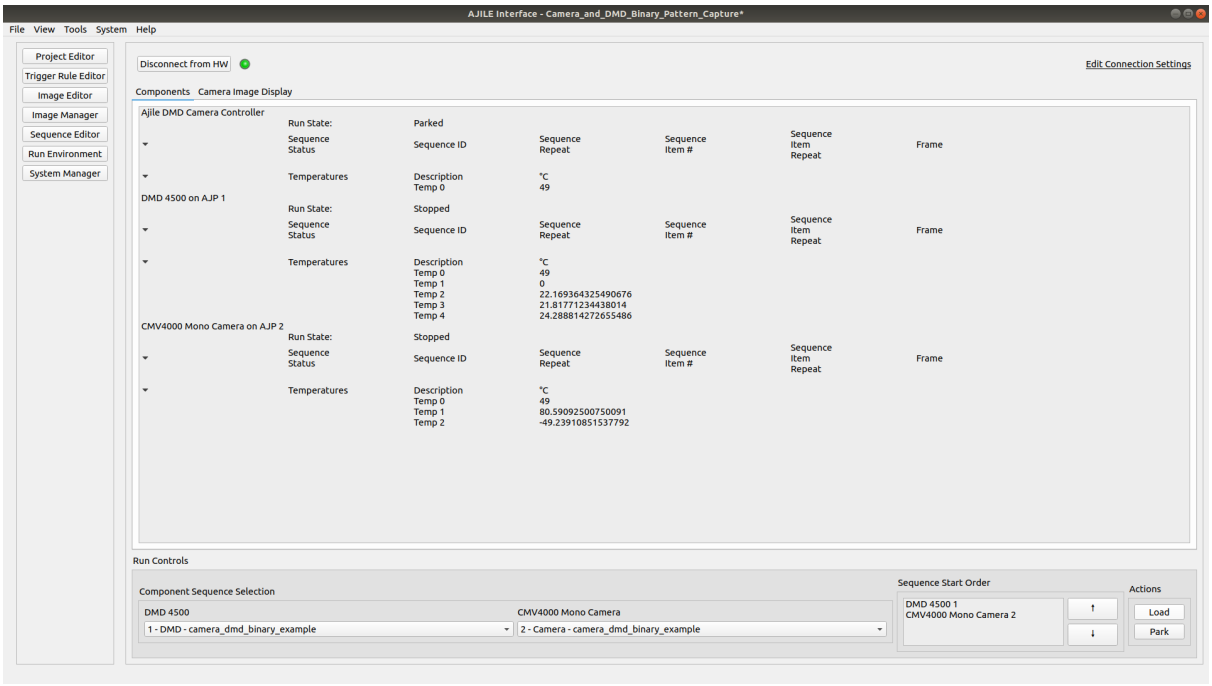

Figure 11.3: A view of the run Environment to control the device components.

11.3 Running Camera Capture Sequences

Running camera sequences is just as was shown in Chapter 9. We connect to the device with `AjileSystem.StartSystem()`, load the project onto the device with `AjileSystem.GetDriver().LoadProject()`, and start the sequence with `AjileSystem.GetDriver().StartSequence()`. We then monitor the running status of the device with `AjileSystem.GetDeviceState()` or read the SequenceStatusValues with `AjileSystem.GetDriver().GetNextSequenceStatus()` to check when the sequence has completed and all camera exposures have finished.

Note that it is possible to continuously capture images by setting the sequence or sequence item repeat count to zero, which means loop forever until `StopSequence()` is called. When combined with `RetrieveImage()` in the next secion, this is how one would implement a live camera image preview, which is very useful for camera setup steps such as moving the camera into the correct position on the scene and focusing the lens.

11.3.1 Running Camera Capture Sequences in the GUI

Running a camera capture sequence is accomplished in the Run Environment. The Run Environment allows the users to: connect to hardware; view connected components; load projects; run, stop and park components; change the sequences and their start order; and retrieve, view and save camera images.

First the hardware must be connected to the GUI by pressing the 'Connect to HW' button seen in Figure 11.3. Next the sequences must be loaded by pressing the 'Load' button and subsequently the 'Run' button located in the bottom right of the Run Environment. The status light indicator will change to blue while the sequences are running. More information about running sequences is available in the previous Chapter 6. The sequence can be stopped by pressing the 'Stop' button where the 'Run' button was previously located.

11.4 Retrieving Images

When camera exposures take place after a camera Frame has completed, the image data from the camera sensor will be stored at the Image memory address with the corresponding Image ID for the Frame. The image data will reside on the device, and is not automatically transferred back to the host PC. In order for the host PC to retrieve the camera images from the device so that they can be displayed, stored and processed, a function is provided to request images from the device. This function is `ControllerDriver.RetrieveImage()`. Images can be retrieved by specifying the Image ID, or they can be retrieved by specifying the Sequence, Sequence Item and Frame Index of a given frame, which may be useful if a Sequence Status is being used to know which camera exposure has just completed. An example of using RetrieveImage is shown in Listing 11.2, where we show retrieving an image based on the Frame index first when a sequence status message has been receive. Then we show the simpler way of retrieving an image by simply specifying its Image ID. The following items should be paid attention to in the example:

- We wait for a sequence status message before retrieving the image from the device to make sure that the camera exposure for that frame is complete, otherwise undefined memory will be returned to the host instead of valid image data.
- The Sequence Item and Frame indices used in `RetrieveImage()` have '1' subtracted from them since the indexing of the Sequence Item and Frame in the SequenceStatusValues begins at 1 (i.e. are 1-indexed), but the indexing within a SequenceItem or Frame begins at 0 (i.e. are 0-indexed) within a Project, and `RetrieveImage()` expects the latter (0-indexed).
- We check to make sure that the image has a valid width and height before processing it. If `RetrieveImage()` fails to return a valid image due to an error condition these will be 0.
- We save the image with `Image.WriteToFile()` with an output bit depth of 16-bit. Here we assume that a 10-bit image is being captured (CMV4000 default), which can we convert to 16-bit so that standard image formats (e.g. .png) can be used.

```
if (!ajileSystem.GetDriver()->IsSequenceStatusQueueEmpty(cameraIndex)) {
    // determine the last frame that was captured
    SequenceStatusValues sequenceStatus =
        ajileSystem.GetDriver()->GetLatestSequenceStatus(cameraComponentIndex);
    // retrieve the image based on the frame index
    const aj::Image& ajileImage = ajileSystem.GetDriver()->RetrieveImage(
        aj::RETRIEVE_FROM_FRAME, 0, sequenceStatus.FrameIndex()-1,
        sequenceStatus.SequenceItemIndex()-1, sequenceStatus.SequenceID());
    // save the image to file
    if (ajileImage.Width() > 0 && ajileImage.Height() > 0)
        ajileImage.WriteToFile(filename, 16);
}
// retrieve the image based on the image ID
const aj::Image& ajileImage = ajileSystem.GetDriver()->RetrieveImage(
    aj::RETRIEVE_FROM_IMAGE, 1);
if (ajileImage.Width() > 0 && ajileImage.Height() > 0)
    ajileImage.WriteToFile(filename, 16);
```

Listing 11.2: C++ example of retrieving images from the device.

11.4.1 Retrieving Images in the GUI

Retrieving images in the GUI is accomplished in three different ways, all of which lie in the 'Camera Image Display' tab of the Run Environment. Figure 11.4 provides an overview of the 'Camera Image Display' tab All three of the methods are accessed through the Camera Image Display tab in the Run Environment as seen in Figure 11.5. The next three paragraphs will describe each of the possible paths for retrieving images.

The first method is to retrieve a single image by index. The image index can be selected using the drop-down in Image Retrieval section of the Camera Image Display tab (part of the Run Environment). All camera images that appear in frames in the sequence will be displayed. 'Latest' will also be an option.

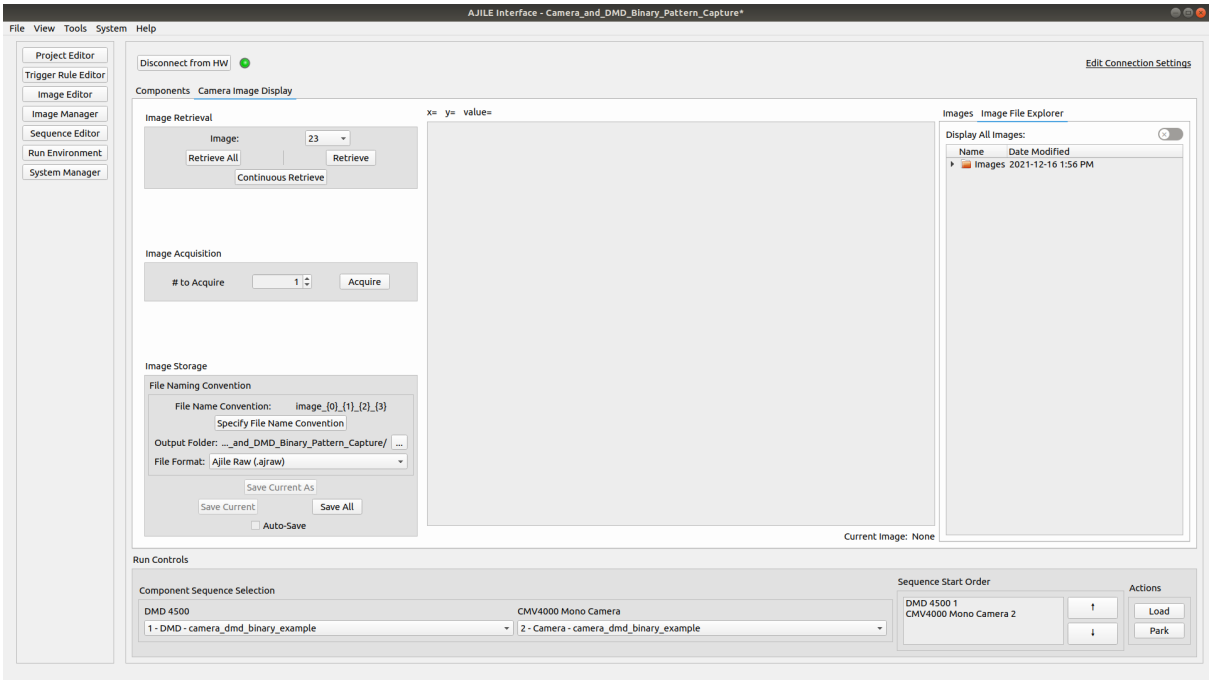

Figure 11.4: A screenshot of the camera image display tab of the Run Environment.

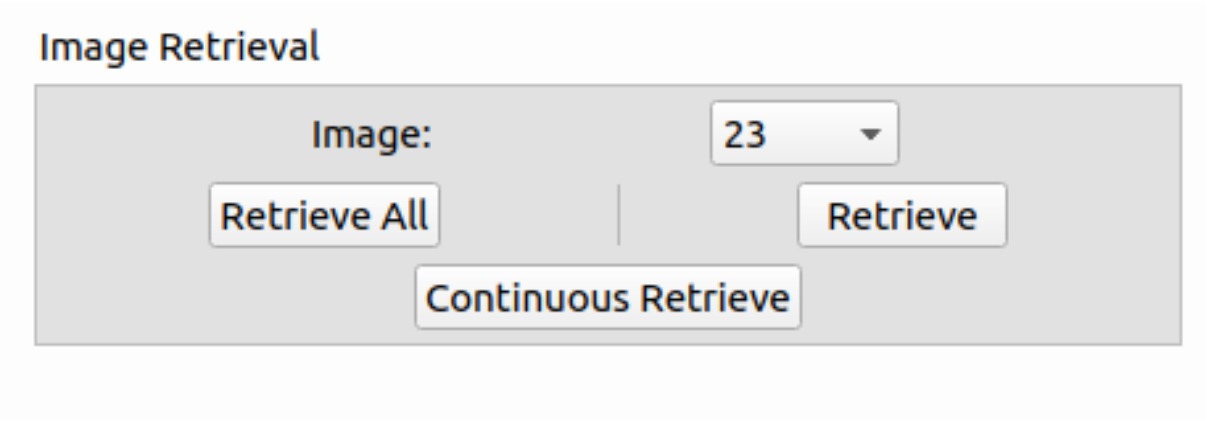

Figure 11.5: A screenshot of the image retrieval section of the Run Environment.

Pressing the 'Retrieve' button will retrieve one image from the device at the given index. If 'Latest' is used, the most recently captured image will be returned based on sequence status values. The retrieved image will be displayed in the Camera Image Display viewer. Here the image can be viewed and pixel values can be examined. The retrieved image will also be added to the Images pane. Note that only one image for each ID will be stored in the image pane, to retain images they should be saved to disk.

The second option is retrieving all images. This method will get one image from each image index - timing out after 5s for each image index. The 'Retrieve All' button initiates the retrieval process. The images will populate the image pane and the last image will be displayed in the viewer. The image ID of the displayed image can be seen underneath the image viewer.

The final method is to continuously retrieve images. Continuous retrieval uses the image index selected in the drop-down as previously described in the first method. However this method will continue to retrieve until it is stopped. Continuous retrieval is started by pressing the 'Continuous Retrieve' button and stopped by pressing it again.

11.5 Image Storage in the GUI

Camera image storage is available in the Image Storage section of the Run Environment as seen in Figure 11.6. The available file formats can be seen in the 'File Format' drop-down menu and apply to all methods of file storage. The methods can be generalized in two categories: manual filename specification or the filename convention. The 'Save Current As' button can be used to save a single image by manually specifying a filename. The alternative option is to specify a filename convention and use 'Save Current' to save a single image or 'Save All' to save all images. The filename convetion is used to specify a naming convention for saving files such as the Image ID, current time, current date or project name. When an image is saved it will use the specified convention and **overwrite** to the specified filename and path. The output folder can be specified to any folder and will default to the project path. The 'Auto-Save' checkbox provides the user the option to automatically save any retrieved or acquired image using the specified file name convention. Images on disk can be viewed in the Image File Explorer pane of the Run Environment's Camera Image Display tab. Double-clicking these images brings them into the viewer. By default, the current session's images will be displayed in the Image File Explorer but all images can be displayed by sliding the 'Display All Image' slider.

11.6 Acquiring Images

Waiting for sequences of image captures to complete then reading them out with `RetrieveImages()` is the simplest way to grab images, however for latency reasons we may want to have the captured camera images automatically returned to the PC immediately when they are captured so that they can be processed by the host application as soon as possible. Accomplishing this is done by the `AcquireImages` interface. The concept for acquiring images is the user application requests for the next captured images to be automatically sent to the PC as soon as they are captured by calling `ContollerDriver.AcquireImages()` and passing in the requested number of images. Typically this will be called before `StartSequence()` so that all images of a sequence are returned to the host. When the sequence is started and camera exposures take place, the captured images are sent to the host PC automatically as soon as possible, and they are added to a FIFO queue of images which resides on the host PC inside the Ajile driver, called the Acquired Image Queue. When the sequence is started, the user application then montiors the Acquired Image Queue with `ControllerDriver.IsAcquiredImageQueueEmpty()`, and the next image is grabbed from the front of the queue with `ControllerDriver.GetNextAcquiredImage()`. A simple example showing the use of the `AcquireImages()` interface is show in Listing 11.3.

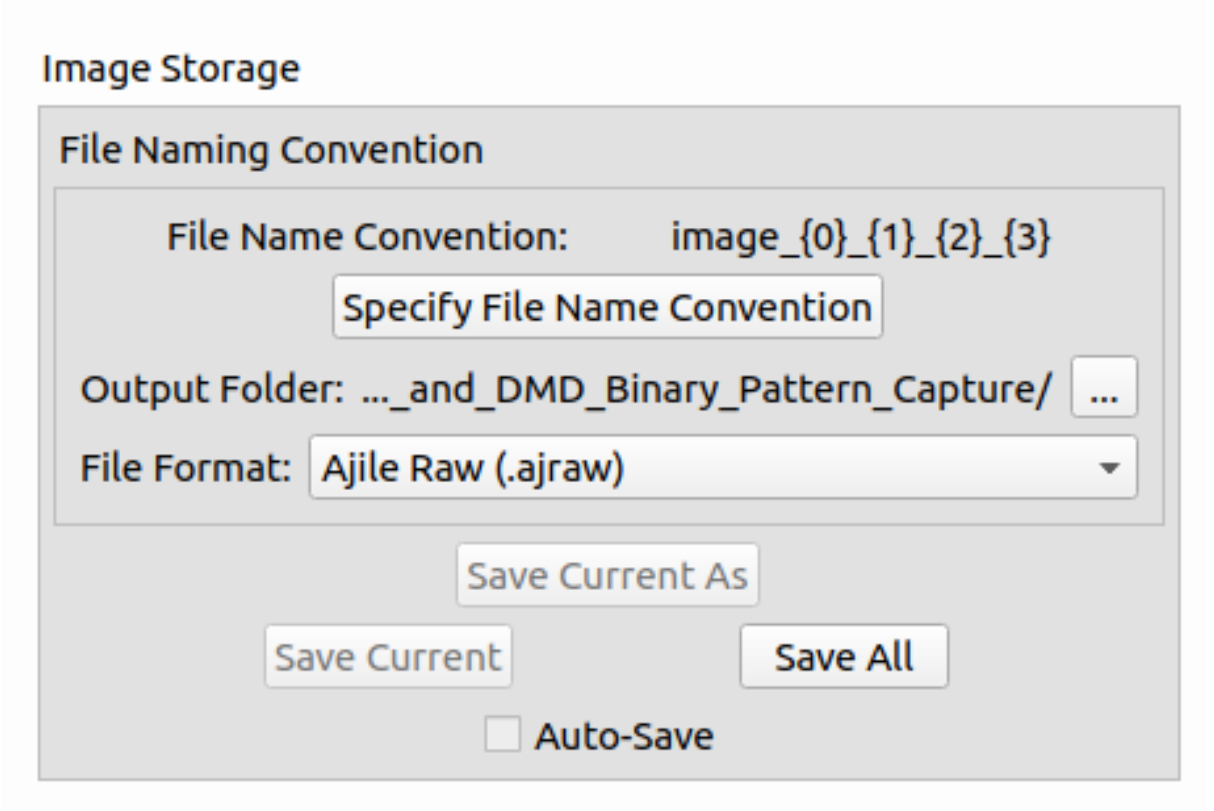

Figure 11.6: A screenshot of the image storage section of the Run Environment.

```
// acquire numImages images
ajileSystem.GetDriver()->AcquireImages(numImages, cameraIndex);
// start the capture
ajileSystem.GetDriver()->StartSequence(sequence.ID(), cameraIndex);
while (imagesRead < numImages) {
    // check if any images were acquired, i.e. are available on the PC
    if (!ajileSystem.GetDriver()->IsAcquiredImageQueueEmpty(cameraIndex)) {
        // get the next image from the acquired image queue
        const aj::Image& ajileImage =
            ajileSystem.GetDriver()->GetNextAcquiredImage(cameraIndex);
        // save the image to file
        if (ajileImage.Width() > 0 && ajileImage.Height() > 0)
            ajileImage.WriteToFile(filename, 16);
        // remove the acquired image from the queue so we can move the next
        ajileSystem.GetDriver()->PopNextAcquiredImage(cameraIndex);
        imagesRead ++;
    }
}
```

Listing 11.3: C++ example of acquiring images from the camera.

Note that in addition to lower latency camera capture read back, the acquire images interface is also useful for capturing more images than there is space for in device on-board memory. To do this, we set the number of images to acquire with `AcquireImages` to a number which is larger than the number of images allocated on the device, then either re-use the same image IDs in the sequence or use repeat counts to have more camera frames than images. Provided the communciations interface (e.g. USB3) can keep up with the transfer speed, it is possible to continuously capture images from the camera without missing a single frame.

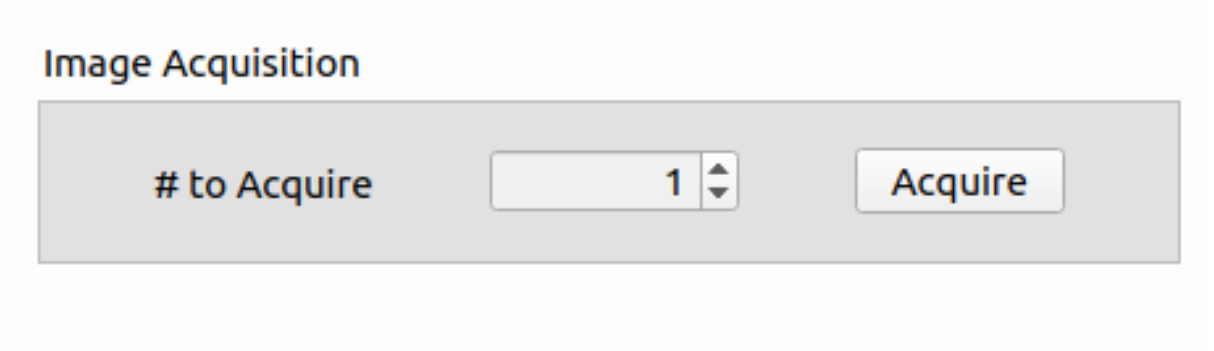

Figure 11.7: A screenshot of the image acquisition section of the Run Environment.

Example Name	Description
camera_sequence	Creates a sequence of 10 camera images. It is possible to configure the region of interest, bit depth and image subsampling parameter, with command line options.
camera_multi_exposure	Creates a sequence of 5 camera images with increasing exposure times.
camera_trigger(in—out)	The same as camera_sequence, except that output or input triggers are also added to the project to start or indicate the start of camera frames.
camera_dmd_binary	Creates a sequence of binary DMD images (Gray codes) and a corresponding camera image for each DMD image. A trigger rule is added to make the camera frame started state output trigger the DMD start frame control input.
camera_dmd_grayscale	Creates a sequence of grayscale DMD images (sinusoidal fringes) and a corresponding camera image for each grayscale DMD image. A trigger rule is added to make the camera frame started state output trigger the DMD start sequence item.
camera_dmd_color	Creates a sequence of 24-bit RGB color DMD images (test images from file) and a corresponding camera image for each color DMD image. A trigger rule is added to make the camera frame started state output trigger the DMD start sequence item.
camera_acquire	Demonstrates the `AcquireImages` API where 100 images are acquired from the camera and saved to file.

Table 11.2: Description of camera example projects.

11.6.1 Acquiring Images in the GUI

Images can be acquired in the GUI by setting the number of images to acquire and pressing the 'Acquire' button in the Image Acquisition section displayed in Figure 11.7. The acquired images will be displayed in the viewer and placed in the 'Images' pane.

11.7 DMD and Camera Synchronization

Synchronizing an Ajile camera with external devices is done via external triggers which was already detailed in Chapter 8. For cases where the DMD and camera components are connected to the same system, such as with the DMD and camera controller device, it is worth noting that internal triggers are used to synchronize the DMD and camera frames. A number of examples are provided with the software suite which demonstrate this for binary, grayscale and color image types.

11.7.1 DMD and Camera in the GUI

In Chapter 8, the process to add trigger rules and control trigger properties in the GUI is explained. A key aspect in synchronizing the DMD and camera is the sequence start order. The triggered component (input trigger component) must be started first while the triggering component (output trigger component) must be started second. The component start order can be adjusted in the Run Environment as seen in Figure 11.3. By default, the order will be determined automatically by project trigger rules.

11.8 Example Projects

A number of example projects are provided which show how to best use the camera and to syncrhonize it with a DMD. These examples are summarized in Table 11.2. The basic flow of the examples is for the project to be created, loaded onto the device and started. Then while running, images will be continuously retrieved from the device and displayed on screen to create a live camera preview from the camera which can be used for setting up the camera positioning and focus. The user must then select the camera image live disply window and press any key to stop the live camera preview, after which the complete sequence of camera images will be read out from the device and saved to file.