HIGH SPEED DIGTIAL IMAGE CAPTURE METHOD FOR A DIGTIAL IMAGE-BASED ELASTO-TOMOGRAPHY BREAST CANCER SCREEING SYSTEM

A thesis

submitted in partial fulfilment

of the requirements for the Degree

of

Master of Mechanical Engineering

in the

University of Canterbury

By

Crispen James Berg

University of Canterbury

2006

ABSTRACT

Digital Image-based Elasto-Tomography (DIET) is an emerging technology for non-invasive breast cancer screening. This technology relies on obtaining high resolution images of a breasts surface under high frequency actuation; typically 50-100Hz. Off-the-shelf digital cameras are unable to capture images directly at these speeds and current digital camera set-ups that are potentially capable of high speed image capture are either low in resolution, expensive, or occupy a volume too large to have them placed about the breast in a dense array. A method is presented for obtaining the required high speed image capture at a resolution of 1280x1024 (1.3 mega-pixels) and actuation frequency of 100Hz. The apparatus uses two Kodak CMOS KAC-9648 imaging sensors in combination with frame grabbers and the dSpace™ control system, to produce an automated image capture system.

The final working system produced images that enabled effective 3D motion tracking of the surface of a silicon phantom actuated at 100Hz. The surface of the phantom was strobed at pre-selected phases from 0 to 360 degrees, and an image was captured for each phase. The times at which image capture occurred were calculated for a phase lag increment of 10 degrees resulting in an image effectively every 0.00028s for the actuator cycle of 0.01s. The comparison of the actual trigger times and pre-selected ideal trigger times gave a mean absolute error of 1.4%, thus demonstrating the accuracy of the final system.

ACKNOWLEDGEMENTS

Firstly, I would like to thank Dr Geoff Chase, Dr Chris Hann and Rodney Elliot for their upmost support and guidance. Their foresight and expertise were invaluable in completing my work and the knowledge and experience I have gained from their leadership has been immeasurable. I would also like to thank my mother and father, my nana, my brother Timothy and my girlfriend Jessica for their never ending support and encouragement, without which I would not have gotten very far. Finally, I would like to thank my mates in Christchurch for the good times, which will only continue in the years to come.

"They are able who think they are able"

Virgil, 70-19 BC
Roman poet

Table of Contents

Chapter 4 Image Capture Software Structure

Chapter 5 Image Capture Results

Chapter 6 Conclusion and Future Work

Appendices A Visual C++ Code

List of Figures

PART I

INTRODUCTION

Chapter 1

Introduction

1.1 Motivation

Breast cancer is the second leading cause of death in women after lung cancer. It is estimated that each year the disease is diagnosed in over one million women worldwide [1]. In New Zealand, breast cancer accounts for the highest mortality rate of all cancers among women and it has the sixth highest death rate out of 173 developed countries.

The key to surviving breast cancer is early detection and treatment. One of the common methods of detection is mammography. Mammography works on the principles of x-ray attenuation differences between normal tissue and diseased tissue [2]. This means that malignant tissue will absorb a different amount of radiation by comparison to its healthy counterpart. As a result, the contrast between the two will appear different on the film. However, the contrast between the two types of tissue is only about 10-15% and small tumours often go undetected, as mammogram analysis is done by humans who may miss such small differences. The mammogram procedure is also quite unpleasant leading to a less than ideal compliance rate among eligible women. More specifically it involves compressing the breast, to achieve a smaller uniform thickness, in order to get the best possible image of the entire tissue volume.

Digital Image-based Elasto-Tomography (DIET) is an emerging technology for non-invasive breast cancer screening. The DIET system relies on the mechanical properties of the breast and looks for regions of high stiffness since cancerous tissue is between 3 and 10 times stiffer than healthy tissue in the breast [3-5].

The DIET system uses digital imaging of a dynamically actuated breast surface to determine tissue surface motion. It then reconstructs the 3D internal tissue stiffness distribution for that motion using advanced inverse finite boundary element methods [13-14]. This process can thus be broken down into four fundamental steps it is also shown in Figure 1.1:

(1) <u>Actuation:</u> Sinusoid motion is induced in the breast via a controlled actuator.

(2) <u>Image Capture:</u> A set of images is captured of the breast through a full range of motion, and co-ordination with the actuator.

(3) <u>Motion Tracking and Measurement:</u> The captured images are analysed to track and determine breast motion and amplitude over the entire tissue surface in 3D space.

(4) <u>Tissue Stiffness Reconstruction:</u> Using the known actuated frequency and phase and the measured breast tissue motion can be used to determine the distribution of tissue stiffness. This distribution is re-constructed by a finite element method.

Presently, there are other elasto-tomographic methods based on magnetic resonance [11] and ultrasound [12] modalities. Both methods are capable of measuring the tissue and are undergoing rapid development across the globe. However, they are also costly in terms of equipment and take significant time to use. They are therefore limited for practical screening applications and are still primarily research activities instead of begin in regular clinical use. Another elastography method was investigated by Kirkpatrick and Duncan (2001) [10]. In their experiments they used a laser or “coherent optical radiation” to create a backscattered speckle pattern. This pattern is then read into a computer using a linear array CCD camera with a telecentric lens. The movement of this speckle pattern is a result of relative sample surface movement, making it possible to determine the surface strain.

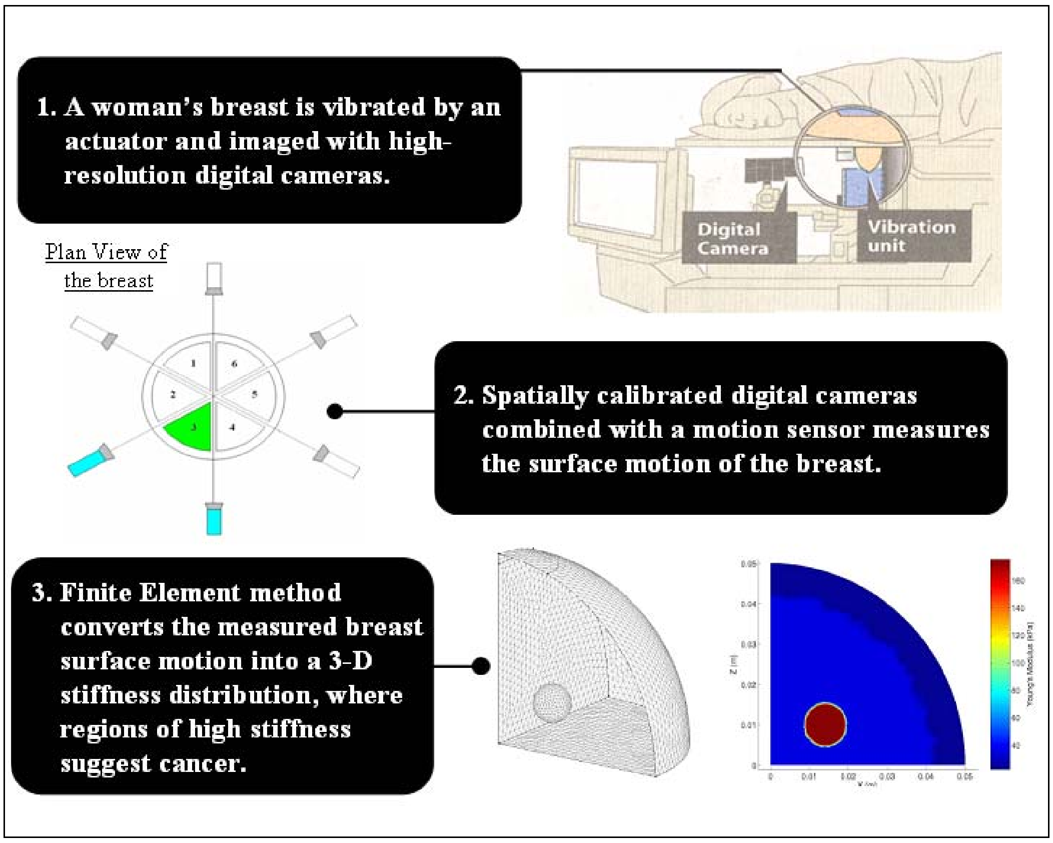

1. A woman's breast is vibrated by an actuator and imaged with high-resolution digital cameras.
Digital Camera
Vibration unit
Plan View of the breast
2. Spatially calibrated digital cameras combined with a motion sensor measures the surface motion of the breast.
3. Finite Element method converts the measured breast surface motion into a 3-D stiffness distribution, where regions of high stiffness suggest cancer.

Figure 1.1: A General overview of the DIET system

The benefit of using the laser speckle pattern is that it increases the spatial resolution of strain measurements. The disadvantage is it only acts on the surface of the sample, where ultrasound or MRI provides deeper, full volume strain measurements, but at reduced spatial resolution. Furthermore the laser can only image small parts of the sample a time. The size of the Kirkpatrick experimental set-up also makes it cumbersome beyond the confines of the laboratory, and it is thus not yet suitable for clinical breast cancer screening.

The DIET system, in contrast, is silicon based and is thus potentially low cost low size and portable technology could therefore potentially be used in any medical centre, or transported to remote areas. In addition, the use of silicon technology ensures that as silicon technology improves and scales upward in capability, so will the DIET system performance. This scalability of performance is not true for wave-based X-Ray or ultrasound approaches.

1.2 Image Capture

For optimal 3D tissue reconstruction, the breast is actuated at 50-100 Hz [13-14]. This frequency is well outside the frequencies of biological processes, such as breathing and heart rate. The amplitude of actuation is about 1-5 mm, which takes into account patient comfort, and limitations on actuator and motion measurements. At these high frequencies, image capture is therefore a challenging task, since clear, crisp images at high resolution are required for high density, accurate velocity and displacement vectors to be obtained. This requirement for the cameras puts the array of pixels required in the SVGA range (1264x1016), at minimum. Based prior analysis of field of view size and desired spatial resolution, images of 4-16 Mpixels will be required [15]

This project develops and implements a method for combining a stroboscope with “off the shelf” CMOS imaging sensors to enable high frequency high-resolution image capture for the DIET system. In particular, the KAC-9648 SVGA CMOS imaging sensors from Kodak are used and the image capture method developed in this research is shown to efficiently and automatically grab images from the breast with actuation frequencies of 50-100 Hz.

As a result, the need for very expensive high speed, high frame rate image capture, which often comes only at lesser resolution, is avoided. In particular, the approach presented allows low cost standard imaging sensors to be used. These sensors are growing in size (Mpixels) and speed on an annual base, so the approach presented in this thesis allows this technology to be utilized as it appears rather than waiting for it to be used in high speed image capture systems.

This project uses CMOS imaging sensors due to their reduced size over their commercial CCD counterparts. This difference would allow the freedom of placing more cameras in a dense array to capture all actuated breast motion with very high resolution, as shown in Figure 1.2. This choice thus also enables greater imaging resolution to be obtained while maintaining simplicity or a lack of greater complexity, in the silicon technology used.

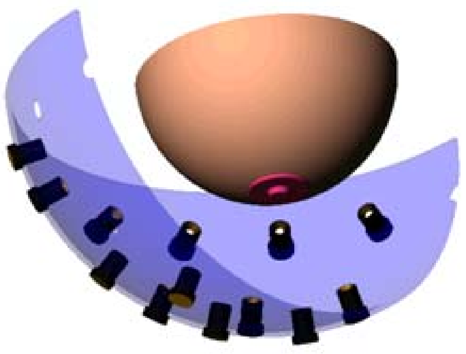


Figure 1.2: Showing a possible set-up for digital cameras about the breast. The cameras are indicated here as the black cylindrical objects about the breast and are in a dome like configuration. The reduced size of the CMOS imaging sensors allows for a denser array of cameras.

1.2.1 Direct Imaging

The last six years has seen the CMOS imagining sensors develop significantly, as an alternative to the CCD. As a result, more research is being done to increase the rate of image capture at ever increasing resolution. For example, Lauxtermann etal (1999) produced a CMOS imaging sensor that could capture images at a rate of 5000 picture/second at a resolution of 256x256 [6]. Kleinfelder etal (2001) produced an imaging sensor capable of 10,000 frames/s at a resolution of 352x288 [7], which streams data at 1 Gpixels/s with each pixel being represented in 8-bits. The frame rates on these imaging sensors is very impressive, however their lack of resolution makes them impracticable for this project.

A CMOS imaging sensor was developed by Krymski etal (2003) in which they produced a sensor with the capacity to capture 240 frames/s at a resolution of 2352x1728 [8]. This sensor however, had trouble finding optics that would fit the large chip size. Thus, currently there are no direct imaging sensors that are able to satisfy the unique high density and speed requirements of the DIET system. What is needed is high speed image capture from standard off-the-shelf CMOS imaging sensors or a means of obtaining it with less stringent speed requirements.

1.2.2 Indirect Imaging

A group of researchers at the University of Stanford [9] achieved high-speed image capture using off-the-shelf CMOS imaging sensors of mid-range resolution. They took 52 off-the-shelf CMOS imaging sensors of resolution 640x480 and a frame rate of 30fps, and arranged them in a circular array, as shown in Figure 1.3. They where mounted at three points on the body of the camera, to allow them to be adjusted independently of one another. One of the central cameras was chosen to be the reference camera in order to align the other 51 cameras to the same field of view. The cameras are triggered in a certain sequence one after the other to produce an overall very high frame rate. This sequence is shown in Figure 1.4.

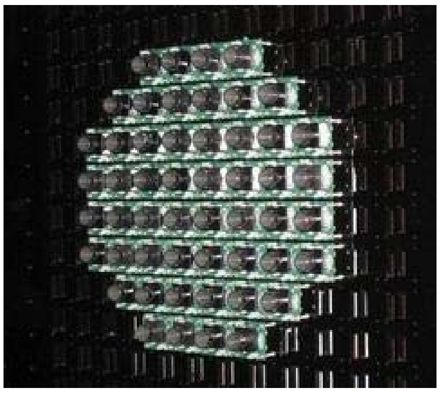


Figure 1.3: The set-up of the 52 CMOS imaging sensors

The resulting images produce seamless high speed capture at the centre of the image with image inconsistencies around the edges. The authors are able to produce frame rate of up to 1560 fps. The images from each camera are processed to correct image distortion from the cameras pixel readout.

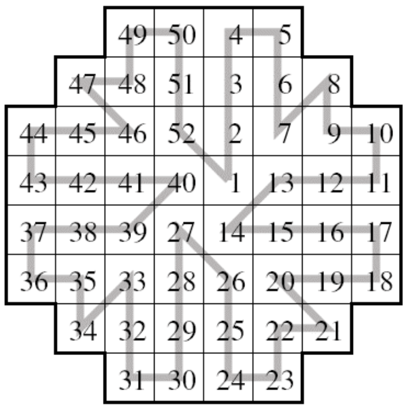


Figure 1.4: This illustrates the firing sequence of the 52 CMOS imaging sensors

However, the method is limited, because the footage being captured must lie far from the array of cameras. Furthermore, an array that large to would have to be placed far from the breast, making the integration of this research into the DIET system impracticable if a portable system is required or desirable

1.3 Summary

The objective of this thesis to use standard off-the-shelf CMOS cameras, and capture frame rates of 50-100fps at a resolution of at least 1264x1016. Specifically, this will involve controlled actuation of the breast, strobing the background light and co-ordinating image capture with a triggered electronic shutter. Capturing images at different phases of a sinusoidal actuation and response enable motion to be captured without high speed sensors. The approach will thus also be shown to be readily generalised to larger sensors as well.

Chapter 2 will discuss the major requirements for getting the overall system working, including a short summary of an earlier prototype that was done prior to this thesis and a focus on the problems encountered. Chapter 3 will describe how a user would set-up the system to produce a set of captured images, and will discuss the settings available to gain the desired results. This chapter will also include specific examples to demonstrate the procedure. Chapter 4 will look at the how the code is structured between the different pieces of equipment. Chapter 5 will examine the results of the image capture system and discuss system limitations. Finally, Chapter 6 will summarize the effectiveness of the overall system, and discuss the potential areas for improvement in the system.

PART II
METHODOLOGY

Chapter 2

Image Capture Apparatus

This chapter describes the fundamental image capture systems in this research.

2.1 Prior Image Capture System

The image capture system used prior to this research was a very simple set-up made of off-the-shelf commercial products. Specifically it consisted of the following fundamental items:

- Two Canon PowerShot™ Digital Cameras
- Electromagnetic Actuator
- dSpace™ Control System
- Laser Interferometer
- Triggerable Stroboscope

The layout for this equipment can be seen in Figure 2.1.

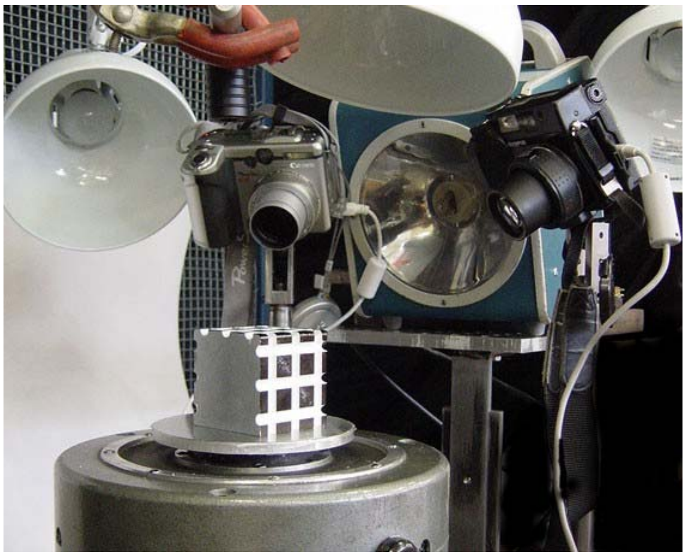


Figure 2.1: Previous image capture set-up [16]

The two commercial digital cameras were set-up on mounts, so that their positions remained fixed after camera calibration. A computer controls these commercial cameras via a USB link. Each camera has its own GUI, which enables the triggering of image capture at times defined by the user. However, this triggering is a manual process, requiring the user to move the mouse cursor between GUI’s. The resulting images are encoded to a specific image format, usually JPEG, and saved to a local memory card.

While the image capture is occurring, the dSpace™ control system drives the triggerable stroboscope and electromagnetic actuator in a sinusoidal pattern. Constant feedback from the laser interferometer allows precise control of the actuator and strobe. The strobe is triggered to flash at a specific predefined point in the actuators motion. The result is that this lighting provides the effect of a stationary actuator as it is the only from of light for image capture with all other light blocked by a black curtain. This enables the motion of the object under actuation to be imaged at discrete user defined intervals in the actuators cycle. The actual position of the actuator relative to the commanded position is monitored using the laser interferometer, which obtains velocity information of the actuator and thus provides actuator displacement after integration of the velocity. This separate use of the laser sensor for feedback control ensures the motion of the actuator maintains a precise, user-defined frequency and amplitude over the image capture process.

The overall effect is that the sinusoidal tissue response can be imaged at specific points in the response without high speed imagery, as several cycles may be used to create a single image. This approach assumes linear, or largely linear, response to these small amplitude inputs, as seen in previous elastography studies [13]. It also allows high resolution imaging at any sensor speed. Several points of the actuator cycle can be captured so that magnitude and phase (relative to actuation) can be determined for the tissue motion.

2.2 Motivation for the New System

The manual triggering of the cameras in the initial set-up is a very time consuming task, especially when multiple images are required to be taken to capture a good image at each phase shift. In addition, the use of these commercial cameras limits the options for adjusting the properties of the camera. For example, the option of decreasing the size of the active window reduces the exposure time per frame and consequently increases the frame rate that the camera is capable of producing. However, there is of course also a trade off between increasing the frame rate and decreasing the resolution. Thus, the control of these attributes amongst others will make the camera system more adaptive and robust to changes in or, eventually, clinical laboratory conditions.

Another advantage of using purpose built cameras in this system instead of commercial cameras is the ability to attach the stroboscope directly to the CMOS imaging sensor itself. This feature would allow the camera to synchronize the strobe flash with the frame integration, to optimise the frame exposure time as shown in Figure 2.2. It would also dramatically shrink the overall system package and provide better overall lighting quality [16].

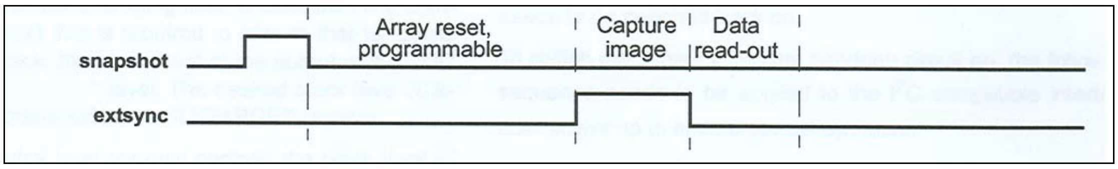


Figure 2.2: Shown here are the timing diagrams for the strobe synchronization with image sensor snap. This is a brief representation of where the triggering of the strobe occurs.

Furthermore, the CMOS imaging sensors can be automatically triggered externally to perform a single frame snap. This feature provides the advantage of capturing image data at more precisely controlled times. In the case of this system, the external trigger is provided by the dSpace™ control system, itself a precise electronic, programmed real time system.

Finally, the CMOS imaging sensor has a significant size advantage over the commercial camera, in that the CMOS is almost a third the size of the commercial camera. This reduction in size is because the CMOS imaging sensor does not require onboard image processing, image storage, or a local power source. This size difference also means a denser array of cameras can be placed about the test piece to obtain larger amounts of data without increasing in system size.

The electromagnetic actuator used in the initial system was the Derritron vibration electromagnetic exciter, which operates on the same principles as a voice coil. To gain position information for the actuator, a laser interferometer was used to measure the velocity of the actuator, and then integrated to find the displacement. An improved version of a “voice-coil” actuator was developed at the university as a final year project [17]. The result was an actuator that performed as well as its predecessor, but included a linear transducer, which provides continuous position data to dSpace™. This new actuator and sensor system enables the actuator position to be automatically monitored, rather than using the laser interferometer, which requires extensive external user set-up takes further space.

2.3 New Image Capture System Overview

The setup for the new image capture system developed and presented in this thesis is divided into two main sections:

- Image Capture
- Actuator and Trigger Control

The overall layout for the image capture and related trigger and data lines is illustrated in Figure 2.3.

2.3.1 Image Capture

The image capture computer (ICC) handles all the capturing and storing of the digital images. The ICC contains two PCI frame grabbers and an I^2C adapter. The PCI frame grabbers capture all the image data arriving from the imaging sensors along the pixel data lines, as shown in Figure 2.3.

Two Kodak KAC-9648 colour image sensors are used in this apparatus. Each sensor produces image data output in the form of 10-bits per pixel at a resolution of 1280 x 1024. Communication between the image capture computer and the camera is carried out via the I^2C adapter. The adapter has two digital lines coming out of it, the first is the serial data line (SDL) and the second is the serial clock line (SCL).

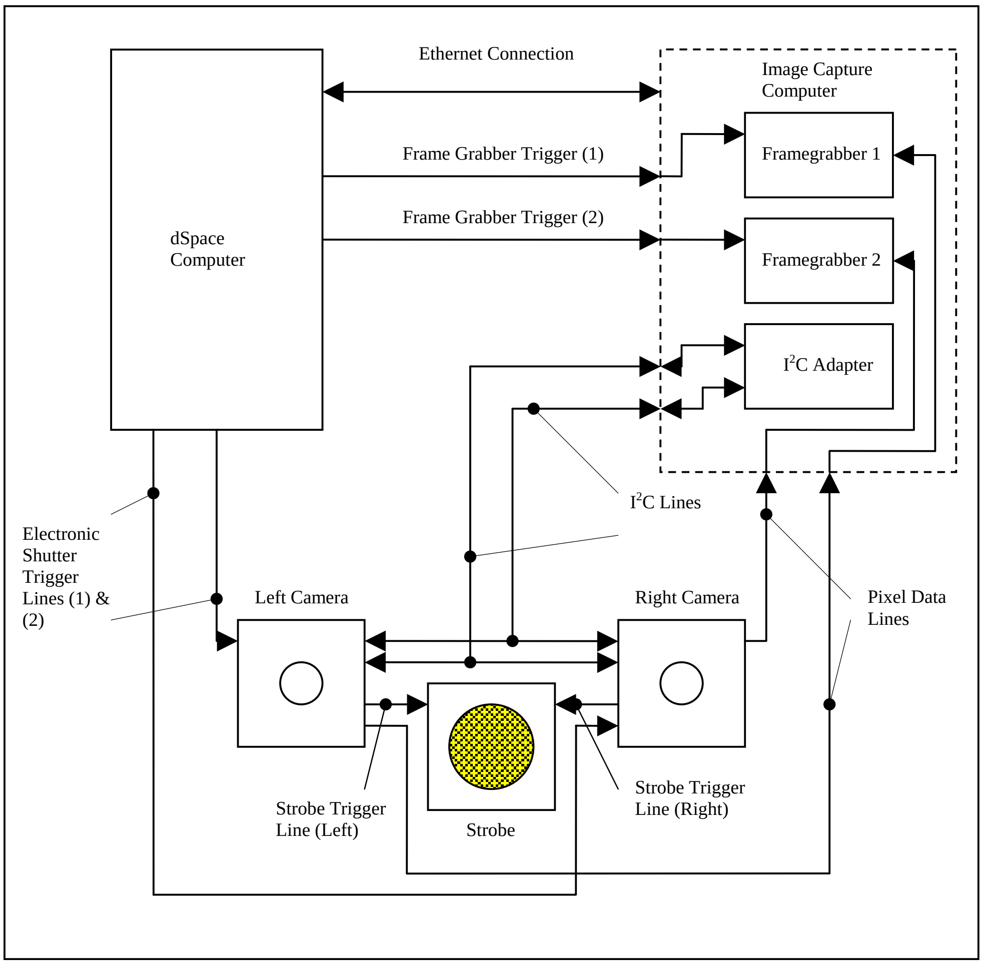


Figure 2.3: The layout for the digital image capture system

Both cameras are connected in parallel to these two serial lines and each has a unique bus address as shown in Figure 2.4. When the I^{2}C adapter communicates with the left camera, for example, it first transmits the bus address, of the left camera. This puts the camera in a state to listen for any information arriving down the two serial lines. This information could be a change in the active window size required by the user or same other input. The right camera then ignores this information since the I^{2}C adapter is only “addressing” the left camera.

There are two camera configurations required, involving a communication between the I^{2}C adapter and the cameras. The first camera configuration is the initialisation of the digital cameras, which enables them to be compatible with the frame grabbers. The result of this initialisation is a continuous stream of video data, which is displayed on the screen and enables the user to adjust colour gains, focus, camera position and aperture size as required.

The second configuration puts the cameras into a state where they are able to receive a digital pulse from dSpace™, which triggers the frame exposure and strobe activation. Specifically, there is an input pin and an output pin on the sensor that is automatically configured after instructions from the I^{2}C adaptor, as shown in Figure 2.5.

The first pin is called a triggered snap pin, which receives the pulse from dSpace™ and starts frame exposure. The second pin is called an external sync, which supplies a pulse to activate the strobe. The precise timing of the strobe trigger from the camera is preset and cannot be changed by the user.

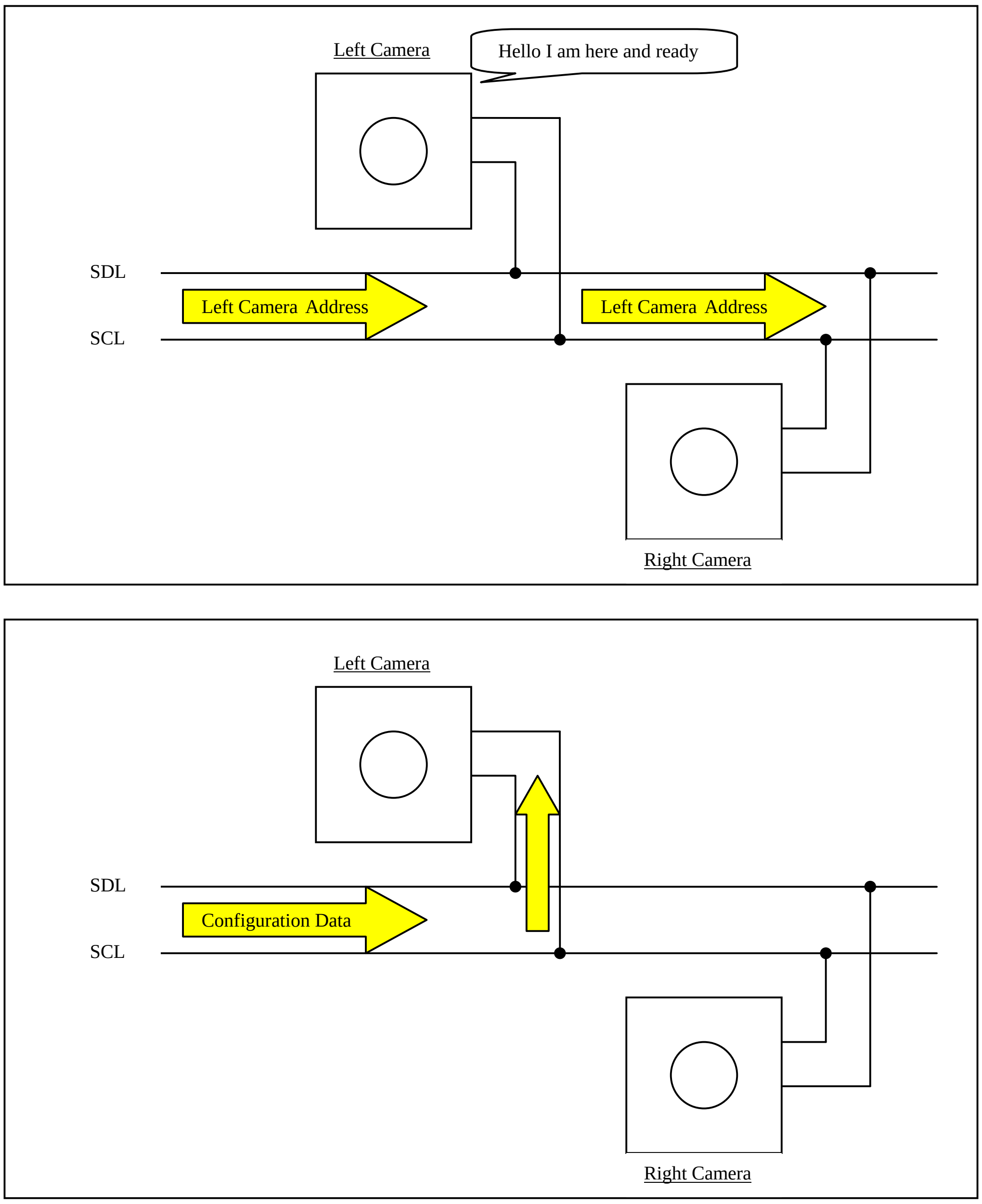


Figure 2.4: (Top) The cameras address being sent out from the I^{2}C adapter. (Bottom) The configuration data being sent to the left camera.

After both camera configurations are performed, enabling compatibility with the frame grabbers and triggering of the frame exposure and strobe activation, the cameras are ready for image capture of the actuated test phantom. More details on these configurations and image capture are given in Chapter 4.

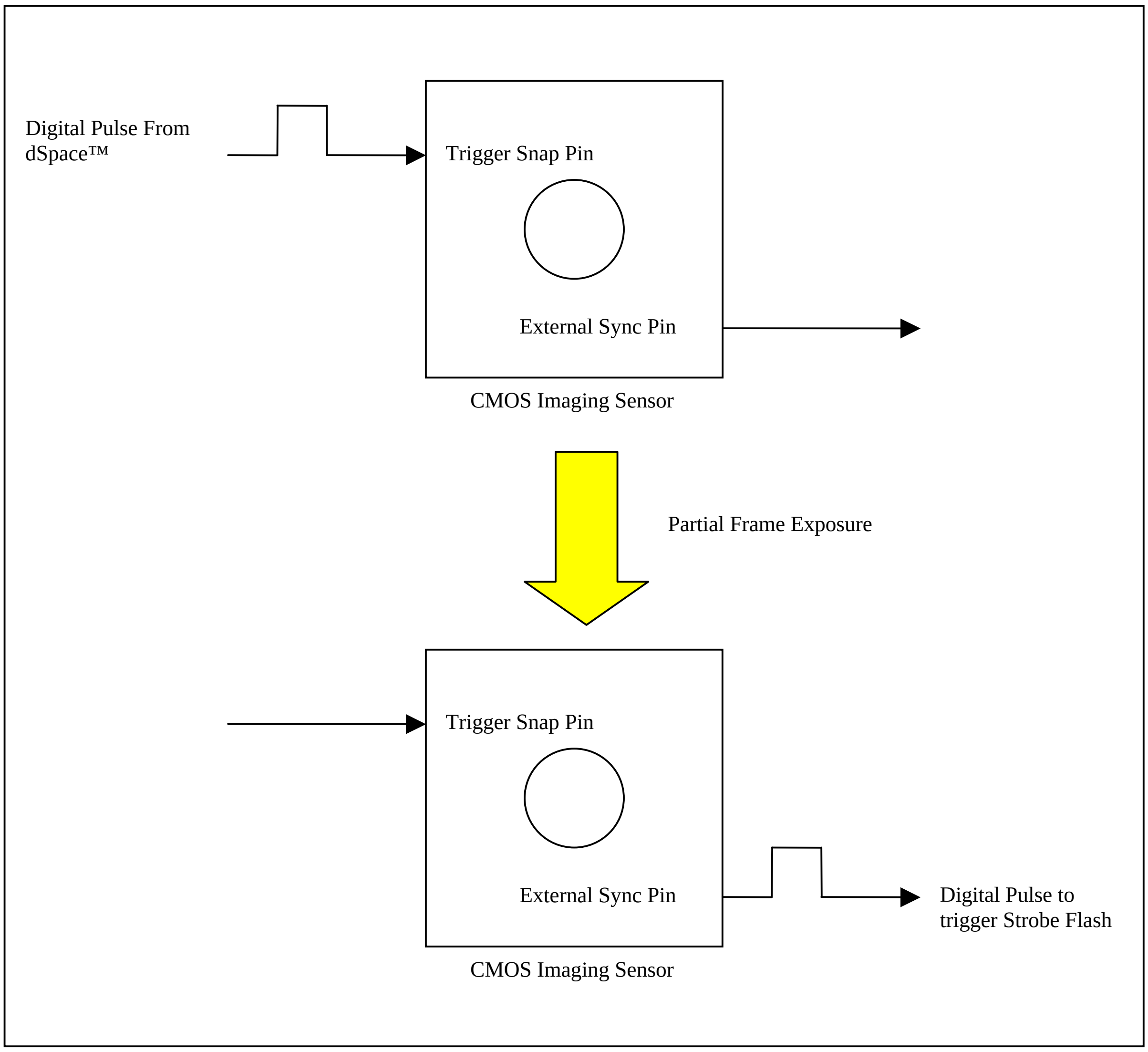


Figure 2.5: The process of strobe trigger by the camera

The cameras used in this system produce images that are in colour. To get the coloured images the pixels on the image sensor itself are arranged in a pattern known as a Bayer Pattern, as shown in Figure 2.6. The pixels in this pattern are sensitive to the colours of green, blue and red respectively, as shown in Figure (a), (b) and (c). This sensitivity is achieved by filtering the light and only allowing the required colours of the incoming image to register at the pixel sites. To obtain a complete colour image to be rendered to an image file and to the screen, the Bayer Pattern data the needs to be processed into three complete colour arrays.

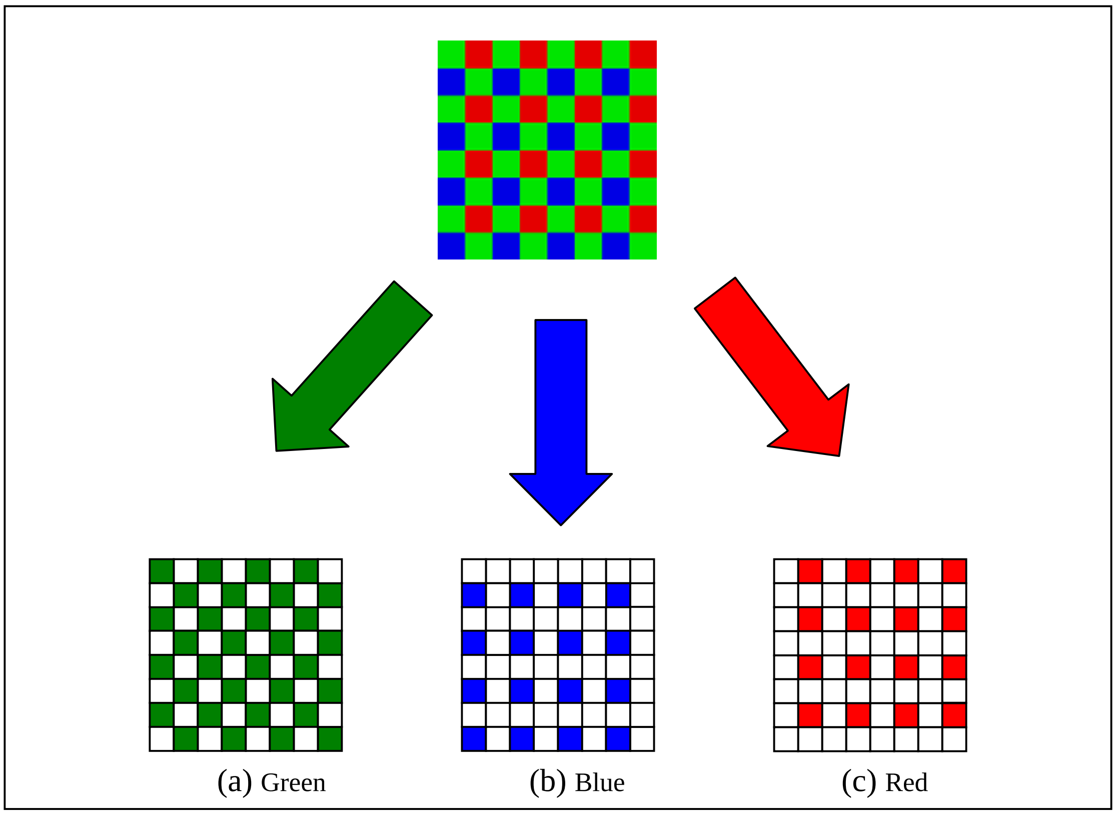


Figure 2.6: The Bayer Pattern and the separate colour arrays

Specifically, the white spaces in the colour arrays of Figure 2.6(a), (b) and (c) are filled by interpolating the pixel values of the pixels adjacent to the white space. This colour interpolation is performed on every frame arriving at the ICC. The reason for the dominant green in the Bayer Pattern is because the human eye is most sensitive to the colour green.

2.3.2 Actuator and Strobe Trigger Control

The set-up for the actuator and trigger control is shown below in Figure 2.7. The dSpace™ computer uses Simulink from Matlab™ to create a system for controlling the input and output signals. The system for processing the signals is built up from blocks, similar to a wiring diagram, where Simulink blocks are connected together to perform its portion of the image capture task on the dSpace™ module.

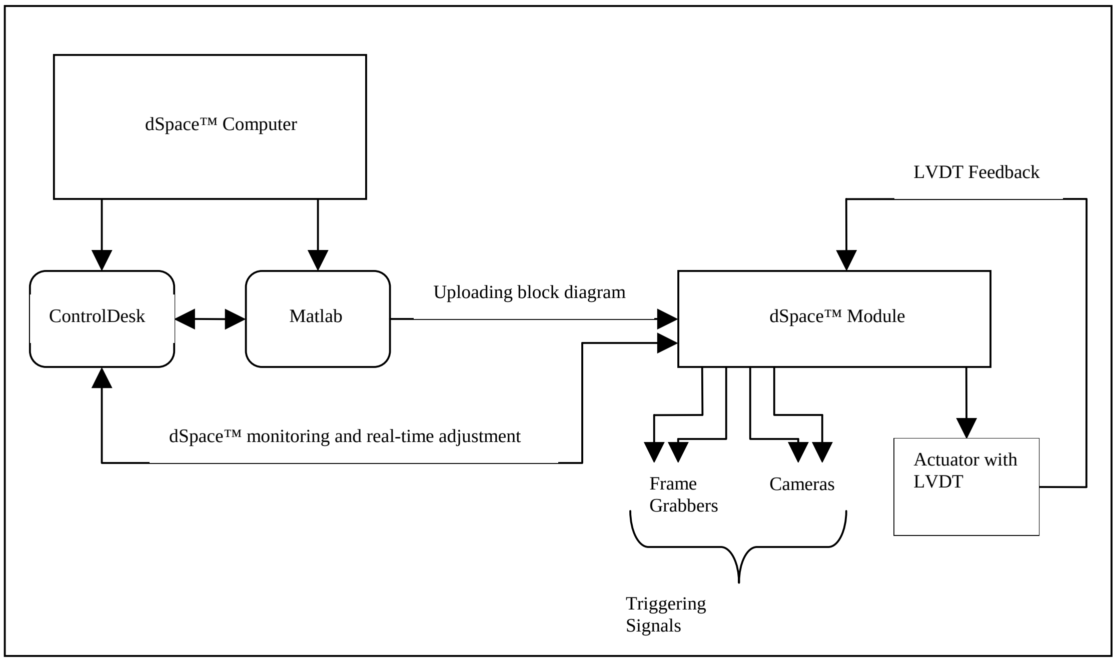


Figure 2.7: An outline of the dSpace control system set-up. The rounded boxes indicate the software contribution.

The portion of the image capture process for which the dSpace™ is responsible, is the generation and synchronising of signals sent to the actuator and necessary trigger signals. Once the diagram is ready it is automatically transferred to a C code format, uploaded to dSpace™, and then run by dSpace™ in real-time inside the dSpace™ module. The settings in the Simulink™ diagram can be adjusted in real-time using the dSpace™ software ControlDesk™. ControlDesk™ makes it possible to automatically perform real time adjustments of the working embedded code in the dSpace™ module. For example, resetting the trigger signals to the frame grabbers and cameras, and real time adjustment of the actuator amplitude.

ControlDesk ™ also allows trigger settings to be modified via a user built project interface, which will be discussed in detail in Chapter 3. The rounded boxes in Figure 2.7 represent the programs interactions with the hardware where ControlDesk™ and Matlab™ are constantly talking to one another and adjusting the settings in the dSpace™ module.

The programming language used to automate ControlDesk™ is known as Python. Python is a high level scripting, interpreted and interactive object-oriented programming language. The Python™ code is used to talk to the ICC and automate the sending of the trigger signals. The hierarchy for the operation of Simulink™, ControlDesk™ and Python™ can be seen in Figure 2.8

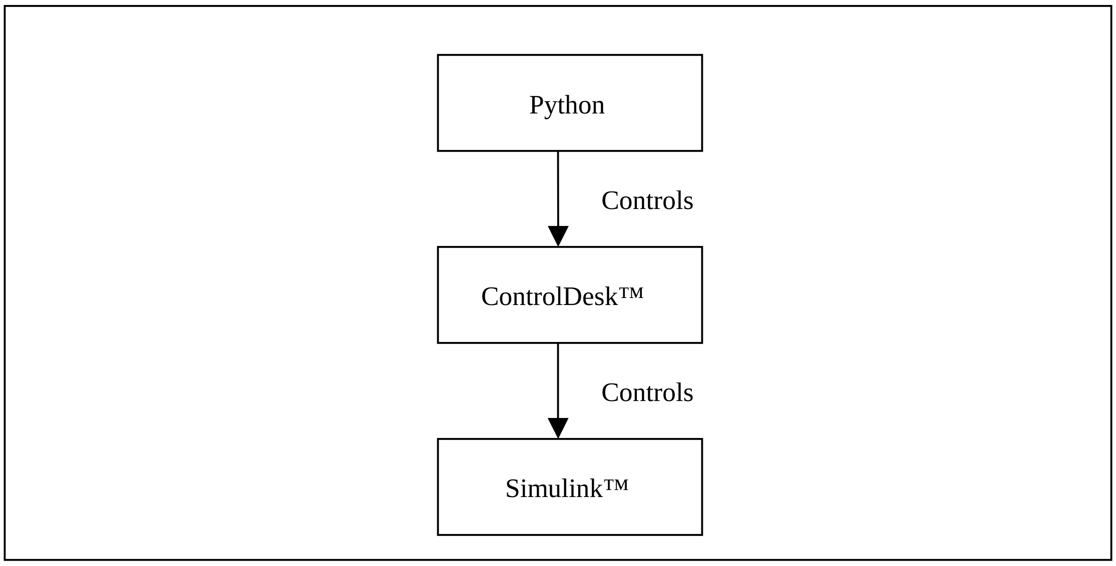


Figure 2.8: The hierarchy of control for the dSpace™ set-up

A 50-100 Hz sinusoidally (or periodically) actuated silicon phantom would require 50-100 fps in the imaging sensors. Since the frame rates of the CMOS imaging sensors for this project have a maximum rate of 18 fps at full resolution, it is therefore not possible to directly image the phantom. To overcome this problem the high-speed phantom is strobed at specific points in its motion, thus effectively rendering the object “stationary” at that point in its resulting sinusoidal periodic response.

Shown in Figure 2.9 is an example of 12 different phase angles in the actuator’s cycle where a user requires an image of the phantom. By introducing a phase shift between the actuators motion and the point of triggering the strobe, the object can be made “stationary” at each of these 12 user defined points in its response and thus an image taken.

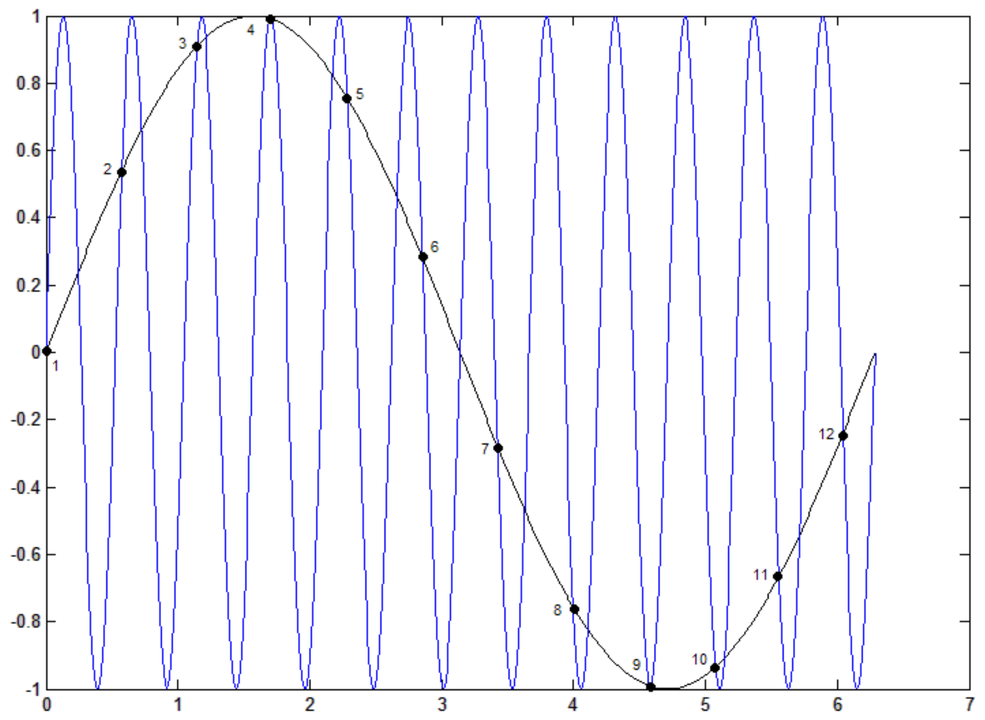


Figure 2.9: Shows an example of a 12 Hz command signal that would drive the sinusoidal motion of an actuator.

In this example, the phase shifts are at increments of 30 degrees labelled 1 to 12 in Figure 2.9. A similar process could be used to capture images of an object at any predefined points in an actuator cycle for any actuator frequency.

For the DIET system, this provides the required ability to capture a sequence of high-resolution images of a 50-100 Hz actuated silicon breast phantom describing the displacement response throughout a 360-degree cycle. At each point the tissue surface motion is imaged and captured. From this data the magnitude and phase of the response relative to the input, can be readily obtained as it is assumed the small sinusoidal inputs result in a sinusoidal response at the (steady state) frequency.

The actuator used in this system has a linear transducer (LVDT), built into the core of the actuator, which sends data back to the dSpace™ module, as shown in Figure 2.7. However, the form of this data is a voltage potential thus further processing is required to obtain position information.

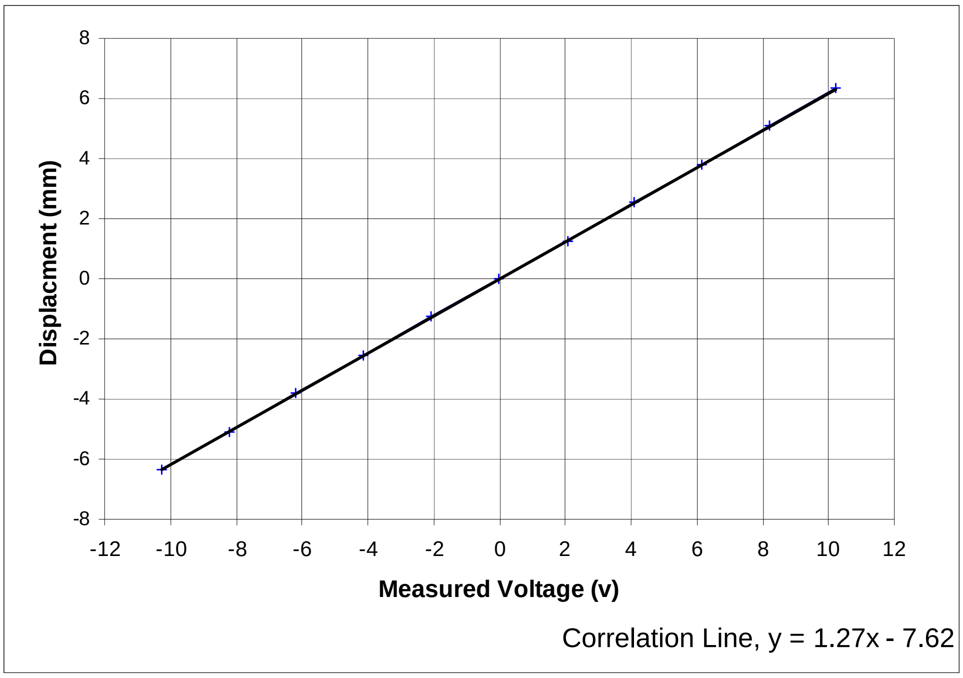


Figure 2.10: The plotted LVDT voltage with measured displacement

This task was achieved by taking a set of measured LVDT displacement positions and their resulting voltages and calculating the linear correlation line which is shown in Figure 2.10. The voltage arriving from the LVDT to dSpace™ can then be converted to displacement using the formula:

$$Displacement = (1.27 \times Voltage - 7.62)\ \text{mm} \tag{1}$$

Equation (1) can then be used to tune the actuator to the commanded displacement amplitude signal using a proportional controller. Note that a PID controller was not used due to limitations on the actuator resulting in the control system becoming unstable with the introduction of the integral and differential gains. However, a proportional controller gave sufficiently accurate results in this work.

2.4 Summary

The solution for capturing high speed images from cameras with a low frame rate is based on the predicable motion of the controlled actuator, moving the phantom in a sinusoidal motion. Tracking this motion allows the capturing of a single image frame at user-defined intervals. The current image capture apparatus has introduced the following changes over the prior image capture set-up:

(1) Introduction of CMOS imaging sensor, reducing camera envelope.

(2) The integration of a new electromagnetic actuator, with an internal linear position sensor (LVDT).

(3) Automated image capture with the “hand shaking” between actuator control and image capture control, reducing the overall image capture time.

These improvements have been made, to provide the initial steps to a completely automated breast cancer screening system.

Chapter 3

Image Capture Applications Set-up

This chapter covers how a user operates the image capture system to produce images for surface motion tracking. The first section will discusses the set-up of dSpace™ to provide actuator control and image capture. The second section discusses the way to change camera settings and image storage preferences on the ICC.

Note that the ICC application caters to each camera independently, so the user operation of the camera settings will be explained for the left camera. The process is identical for the right camera. This independent approach to settings accounts for the use of different and/or specialized cameras over an entire system.

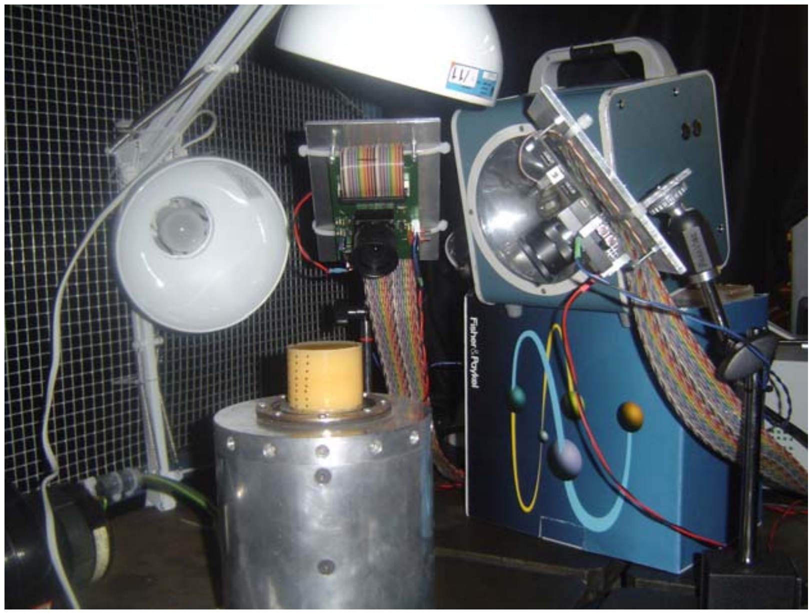


Figure 3.1: The current image capture set-up

3.1 dSpace™ User Set-Up

The dSpace™ application is driven from ControlDesk™, a program designed to interface with the code running on the dSpace™ module shown in Figure 2.8 of Chapter 2. Setting up of the dSpace side of the image capture operation consists of the following main steps:

- Loading the Simulink™ block diagram
- Starting the ControlDesk™ layout
- Zeroing the LVDT and setting the desired actuator amplitude
- The setting and tuning of the proportional gain on the LVDT signal (if required)
- Starting the dSpace™ server

The end result of this process is a running actuator and a computer that is prepared to receive data from the ICC and drive the image capture process.

Once the ControlDesk™ application is running an associated Simulink™ block diagram that defines how the dSpace™ module will run the actuator and activate the triggers, is enabled. This Simulink™ system is then built and automatically downloaded to dSpace™ system by pressing Ctrl+B. As soon as the diagram is built to the module, the actuator and trigger signal control layout is loaded, as shown in Figure 3.2. Figure 3.2 illustrates all the different options available to the user in the ControlDesk™ layout.

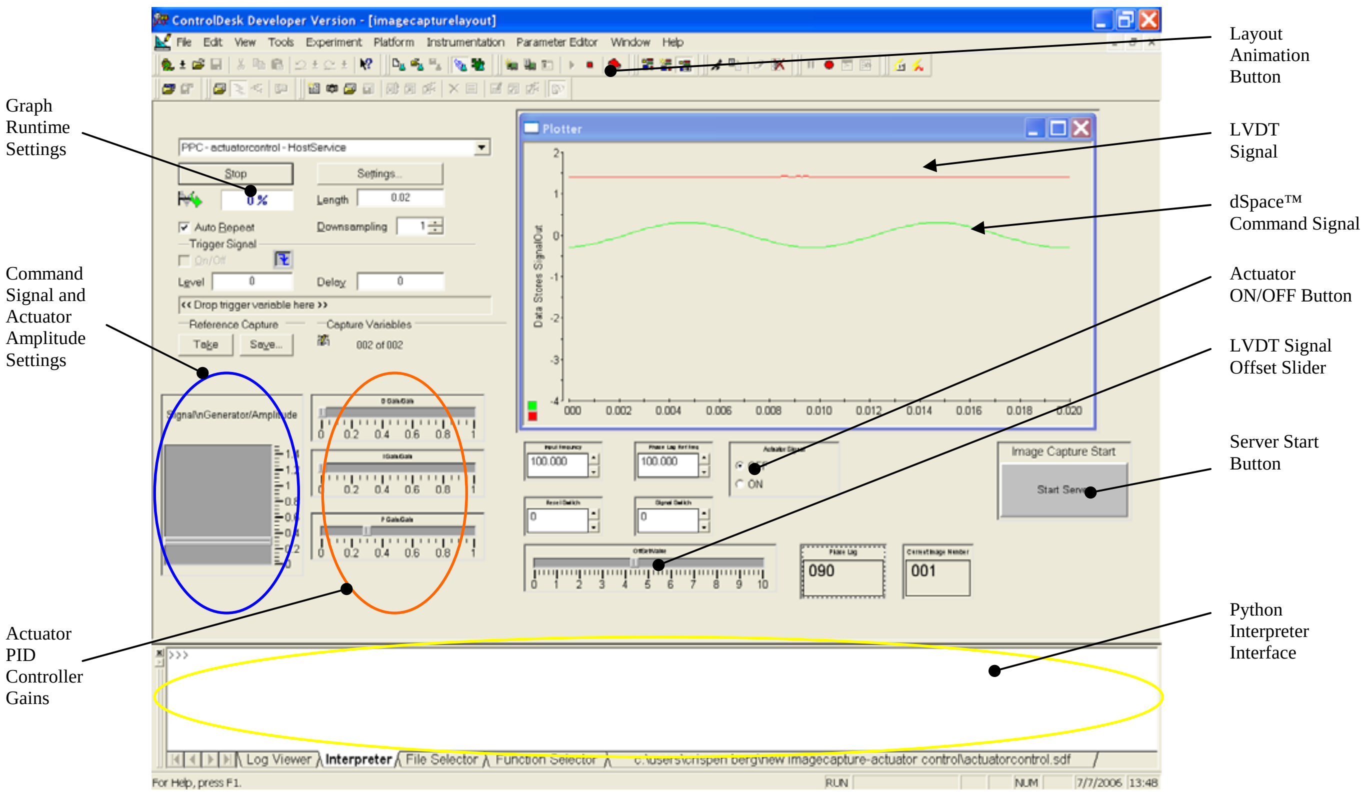
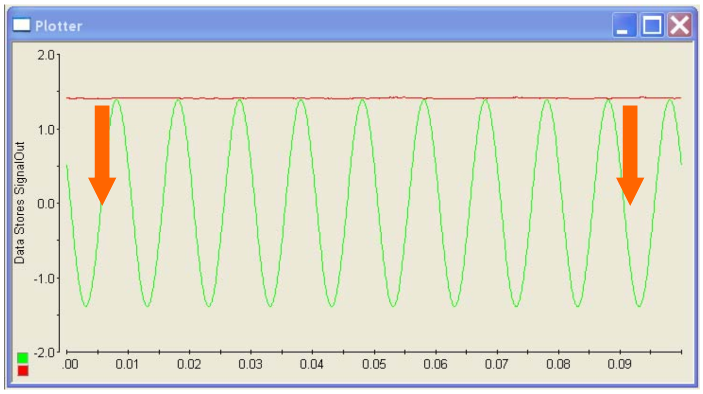
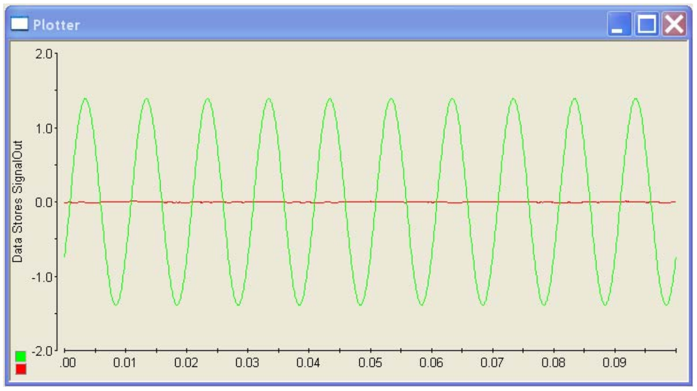

ControlDesk Developer Version - [imagecapturelayout]
File Edit View Tools Experiment Platform Instrumentation Parameter Editor Window Help
Layout Animation Button
Graph Runtime Settings
PPC - actuatorcontrol - HostService
Stop
Settings...
Length 0.02
Auto Repeat
Downsampling 1
Trigger Signal
On/Off
Level 0
Delay 0
<< Drop trigger variable here >>
Reference Capture
Take
Save...
Capture Variables
002 of 002
Plotter
Data Stores SignalOut
LVDT Signal
dSpace™ Command Signal
Command Signal and Actuator Amplitude Settings
Signal\nGenerator/Amplitude
Actuator ON/OFF Button
OFF
ON
LVDT Signal Offset Slider
Image Capture Start
Start Server
Server Start Button
090
001
Actuator PID Controller Gains
Python Interpreter Interface
Log Viewer
Interpreter
File Selector
Function Selector
c:\users\crispen berg\new imagecapture-actuator control\actuatorcontrol.sdf
For Help, press F1.
RUN
NUM
7/7/2006 13:48

Figure 3.2: The ControlDesk™ Image Capture Layout

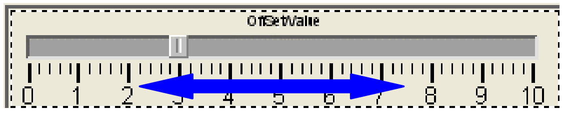
After the layout has been animated, using the layout animation button in Figure 3.2, the signals from the actuator can be read and the values of the gains changed. Once this is achieved, the user can freely interact with the variables in the diagram built to the module, including changes to the proportional (‘P’) gain, command signal amplitude and frequency, and the feedback of actuator position information in volts. However, the downloaded Simulink™ diagram does not start sending the command signal to the actuator until the user clicks on the ON/OFF button. The zeroing of the LVDT involves moving the offset slider also shown in Figure 3.2 until the horizontal line (LVDT signal) is sitting on zero in the signal plot. This process is shown in Figure 3.3.



(a)



(b)



(c)

Figure 3.3: Zeroing the LVDT signal. (a) The signal is before offset is applied. (b) The signal after offset is applied. (c) The slider used to achieve the signal offset.

3.1.1 Controlling the LVDT signal

The actuator in this image capture system, uses a proportional or ‘P’ controller to precisely control the actuator motion during the image capture process. Note that due to the dynamics of the actuator, the addition of significant integral or derivative control produces an unstable result. Thus, these gains are typically set to zero on the ControlDesk™ layout.

Once the LVDT is zeroed, the proportional gain is slowly increased until the signal amplitude matches the amplitude of the command signal. At this point, the LVDT signal needs to be monitored, because as the actuator warms-up the amplitude increases, requiring the gain to be decreased until the system reaches equilibrium. This process usually takes five minutes. Once this equilibrium is achieved the signals should look like those illustrated in Figure 3.4, where it is not required that the phases match perfectly, depending on what phase lag offset has been set.

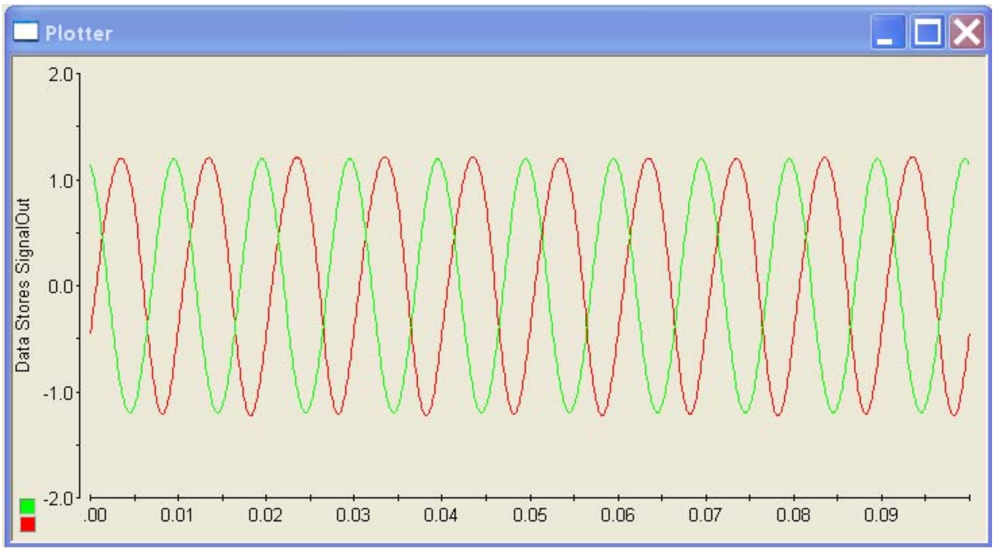


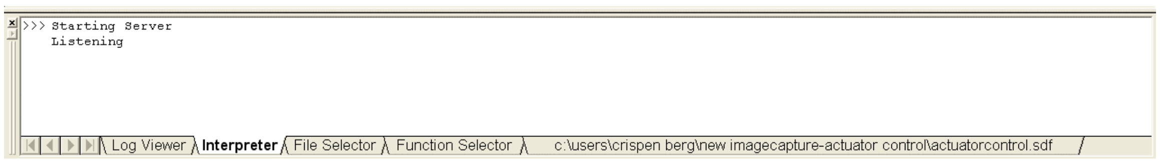
Figure 3.4: The two signals with synchronized amplitude, the command signal in green (light) and the LVDT signal in red (dark)

However at zero degrees of lag these signals would overlap. Finally, any residual phase lag between the command signal and the LVDT signal is not a problem since the image capture triggering system uses the LVDT signal, which is giving a true reading of actuator displacement, rather that the command signal

3.1.2 Starting of the dSpace™ server

In the image capture system, the server is the dSpace™ computer. It receives the information from the ICC and relays that information to the dSpace™ module via the ControlDesk™ layout. Once the user's settings are finalized in the ControlDesk™ layout of Figure 3.2, the user activates the "Start Server" button.

When the server is started, a message is sent to the user in the Python™ interpreter interface, as shown in Figure 3.5. This message indicates that the dSpace™ server is running. The Python™ interface also displays the progress of the image capture process by showing the user the current image number and phase lag. Once the image capture cycle has been completed the Python code that drives the image capture on the dSpace™ computer also switches off the actuator.



```
>>> Starting Server
    Listening
```

Log Viewer
Interpreter
File Selector
Function Selector
c:\users\crispen berg\new imagecapture-actuator control\actuatorcontrol.sdf

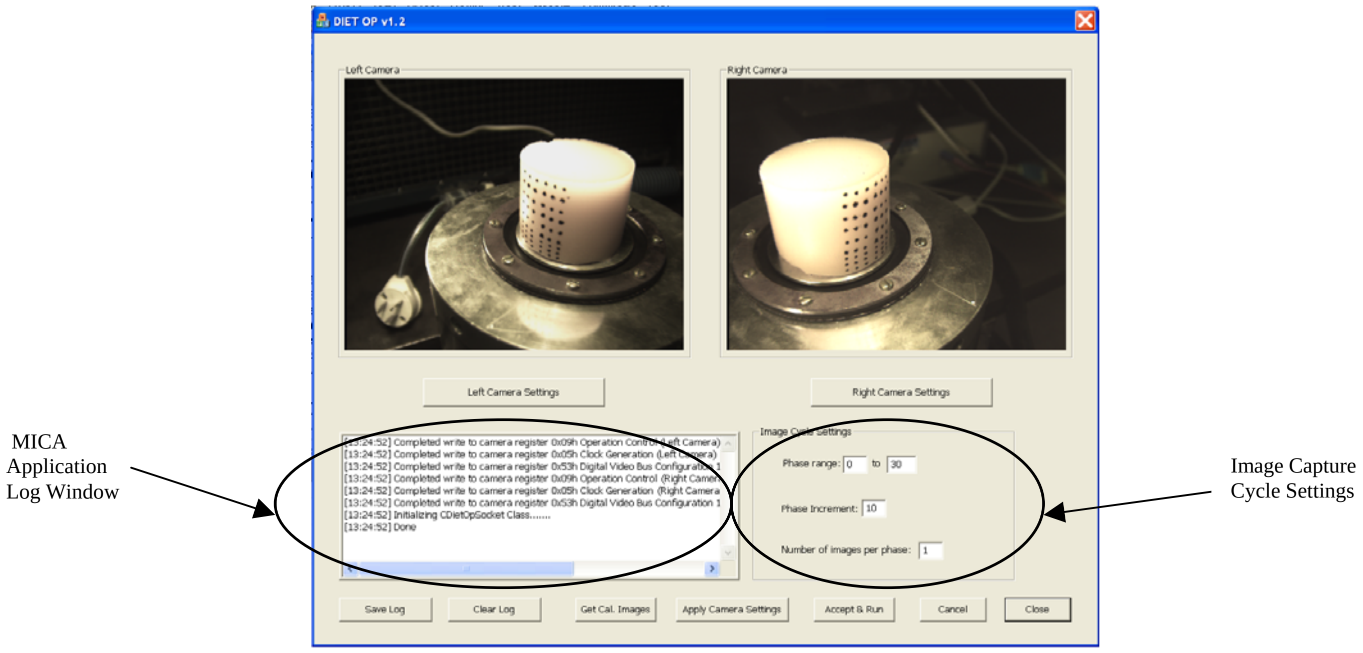
Figure 3.5: The Python™ interface after the "Start Server" button is pressed

3.2 The ICC Image Capture Application

The image capture application consists of two main dialog boxes:

- The main start-up dialog
- The camera settings tabbed dialog

The main start-up dialog is activated from the desktop and is shown in Figure 3.6. All other dialogs are activated from the main dialog in Figure 3.6 at the request of the user. Before starting the main image capture application (MICA), the cameras must first be powered up. Upon activation of the MICA the cameras are placed into the mode where they stream video directly to the main dialog.



```
[13:24:52] Completed write to camera register 0x09h Operation Control (Left Camera)
[13:24:52] Completed write to camera register 0x05h Clock Generation (Left Camera)
[13:24:52] Completed write to camera register 0x53h Digital Video Bus Configuration 1
[13:24:52] Completed write to camera register 0x09h Operation Control (Right Camera
[13:24:52] Completed write to camera register 0x05h Clock Generation (Right Camera
[13:24:52] Completed write to camera register 0x53h Digital Video Bus Configuration 1
[13:24:52] Initializing CDietOpSocket Class.......
[13:24:52] Done
```

Phase range: 0 to 30

Phase Increment: 10

Number of images per phase: 1

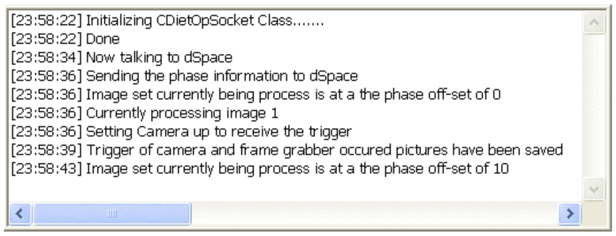
Figure 3.6: The Main Image Capture Application (MICA)

The I^{2}C communications between the cameras and the adapter are registered in the log window that is also shown in Figure 3.6. During the I^{2}C communications, the status of the image capture process is also passed to this window. The messages indicate at what phase and image number the capture process is currently operating as well as any errors that may occur with socket communications and frame grabber interface.

The "Image Cycle Settings" box contains the variables that can be changed with respect to where in sinusoidal the actuator cycle the user requires images. These settings are taken from the MICA and sent to the dSpace™ computer. Therefore, they must be set by the user before the image capture process begins.

Initially, the cameras must first be calibrated. This task requires static images from both cameras where the calibration objects three faces are clearly visible to both cameras while it rest on top of the actuator. The “Get Cal. Images” button in Figure 3.6 takes images from the cameras at the same time and saves them to a path specified by the user.

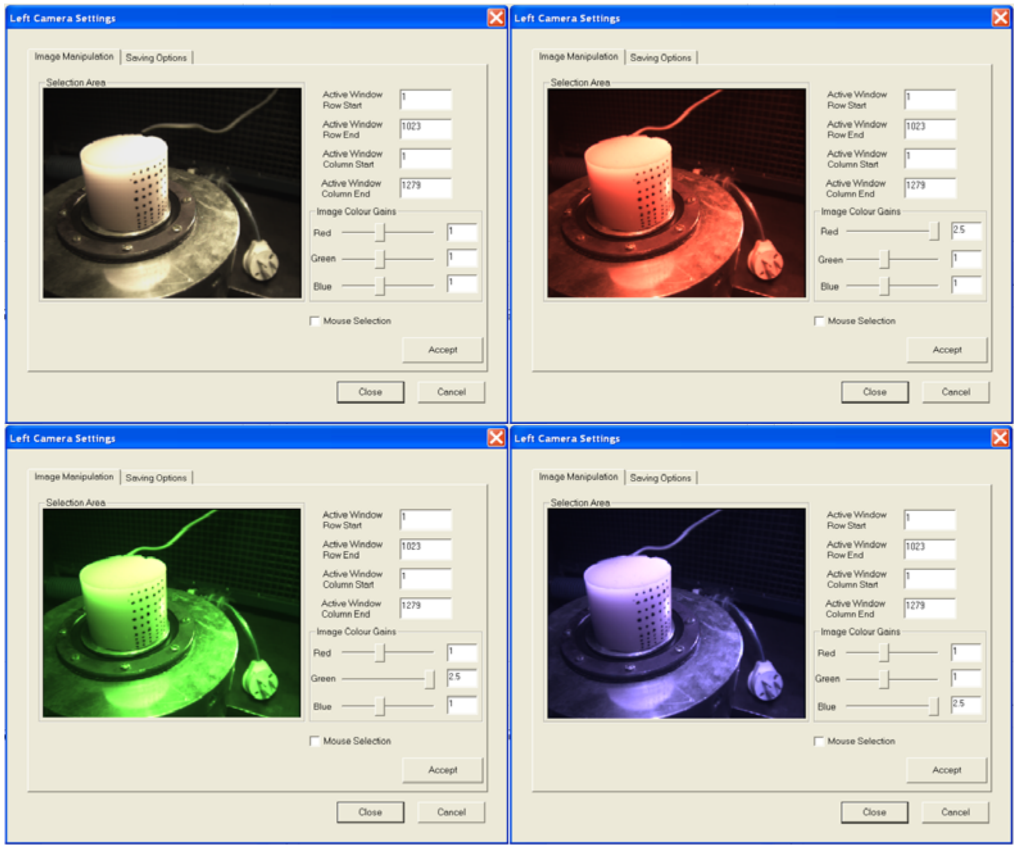
The other features of the MICA are the option of saving the application log for later examination, and clearing of the log for each image capture cycle. The application log provides a running feedback to the user on the progress of the image capture cycle including any errors that occur. The application log in operation is shown in the example given in Figure 3.7.



```
[23:58:22] Initializing CDietOpSocket Class.......
[23:58:22] Done
[23:58:34] Now talking to dSpace
[23:58:36] Sending the phase information to dSpace
[23:58:36] Image set currently being process is at a the phase off-set of 0
[23:58:36] Currently processing image 1
[23:58:36] Setting Camera up to receive the trigger
[23:58:39] Trigger of camera and frame grabber occured pictures have been saved
[23:58:43] Image set currently being process is at a the phase off-set of 10
```

Figure 3.7: The application log in operation

The camera settings can be changed for each camera by clicking on the left (or right) camera settings button in Figure 3.6. Pressing these buttons activates the camera settings dialog box shown in Figure 3.8. In the dialog of Figure 3.8, there are three sliders, one for each of the primary colours that make up the Bayer pattern on the CMOS image sensor. The movement of these sliders adjusts the colours in the image providing adjustment to laboratory light conditions. This function is also illustrated in Figure 3.8 over the range of colours. Alternatively, the values of the colour gain can be added in the edit box to the right of the respective slider. The cameras themselves have a fine adjustment that takes place on the lens, which is attached on the front of the sensor. The front portion of the lens is for the focus and, further back is the adjustment of the iris, which allows more or less light onto the imaging sensor. These adjustments are shown in Figure 3.9 for the camera hardware used in this study.



Active Window Row Start 1
Active Window Row End 1023
Active Window Column Start 1
Active Window Column End 1279
Image Colour Gains: Red 1, Green 1, Blue 1

Active Window Row Start 1
Active Window Row End 1023
Active Window Column Start 1
Active Window Column End 1279
Image Colour Gains: Red 2.5, Green 1, Blue 1

Active Window Row Start 1
Active Window Row End 1023
Active Window Column Start 1
Active Window Column End 1279
Image Colour Gains: Red 1, Green 2.5, Blue 1

Active Window Row Start 1
Active Window Row End 1023
Active Window Column Start 1
Active Window Column End 1279
Image Colour Gains: Red 1, Green 1, Blue 2.5

Figure 3.8: The adjusted colour gains at different extremes. The first image is the camera settings dialog upon activation

The active window of the camera can be changed in Figure 3.8, as well. The active window on the cameras, is the part of the image sensor array where the pixel data is collected. The mouse selection check box at the bottom of the camera settings shown in Figure 3.10 can be selected to activate a set of cross hairs, which are moved using the mouse to select the active window area. Alternatively, the values/coordinates of the new active window can be manually entered into the dialog. Once an active window selection is made, a message box pops up asking whether the user would like to keep the current selection or try again. Clicking on the “Accept” button shown in Figure 3.8 accepts the chosen colour gain and active window selection. This decision loads all the requested values to a ‘.dat’ file that is then stored in the local directory of the MICA.

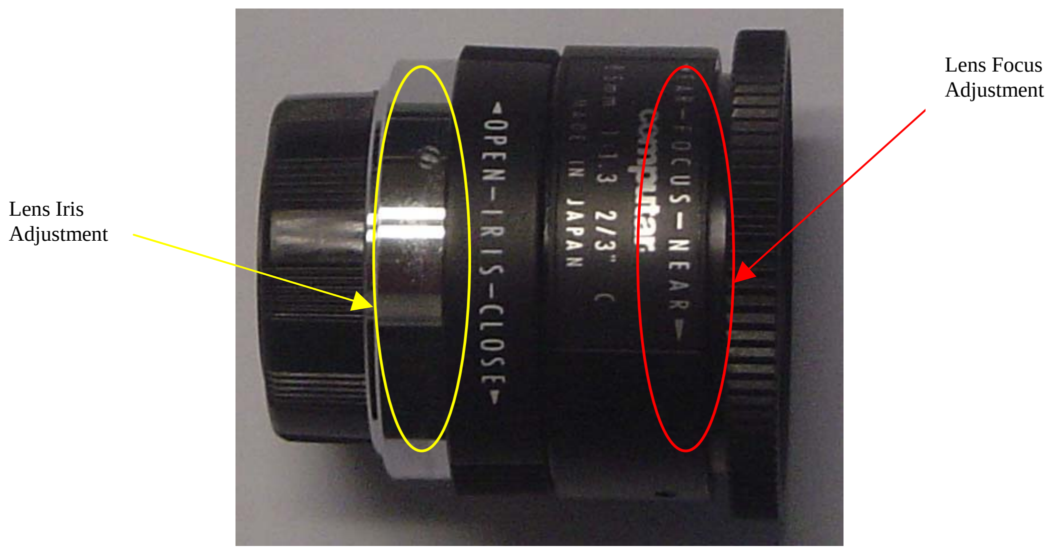


Figure 3.9: Position of lens fine adjustment

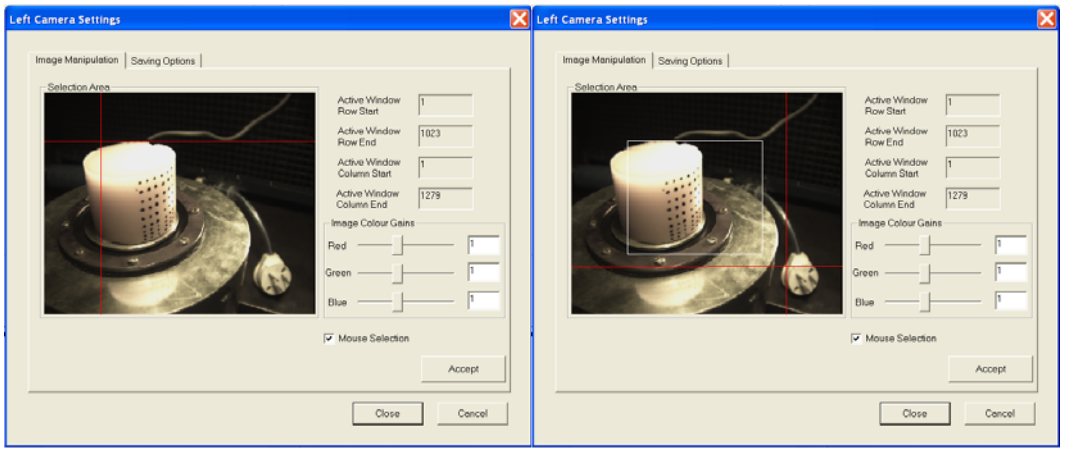

Left Camera Settings

Image Manipulation | Saving Options

Selection Area

Active Window Row Start	1
Active Window Row End	1023
Active Window Column Start	1
Active Window Column End	1279

Image Colour Gains

Red	1
Green	1
Blue	1

☑ Mouse Selection

Left Camera Settings

Image Manipulation | Saving Options

Selection Area

Active Window Row Start	1
Active Window Row End	1023
Active Window Column Start	1
Active Window Column End	1279

Image Colour Gains

Red	1
Green	1
Blue	1

☑ Mouse Selection

Figure 3.10: The process of mouse selection

The second tab of the camera settings dialog is used to store the path selection, for the captured images. There are two separate paths to be chosen. The first is for the calibration images and the second is for actuated image capture. These dialogs are shown in Figure 3.11. The path is chosen by clicking the browse button and selecting a path. Up to five previous paths can be stored in the drop down list. Along with the path selection, it is also possible to select the format of the image being saved. Once the path and image formats have been chosen, again the “Accept” button is pressed to store the path settings to another ‘.dat’ file.

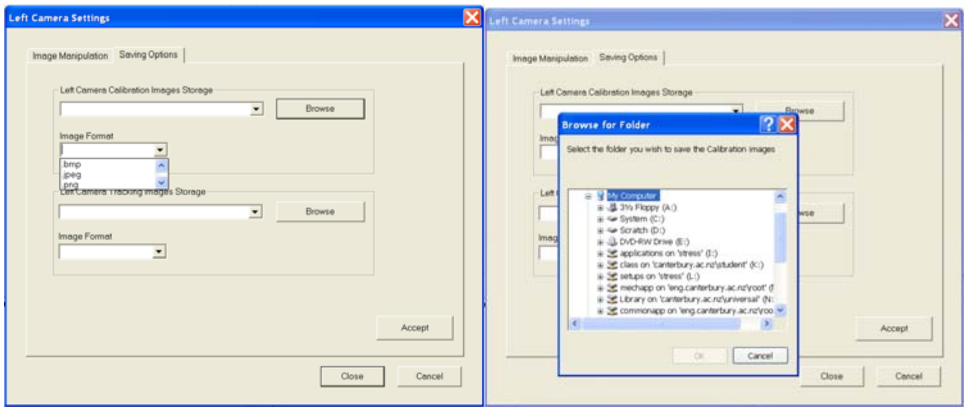

Left Camera Settings
Image Manipulation
Saving Options
Left Camera Calibration Images Storage
Browse
Image Format
.bmp
.jpeg
.png
Left Camera Tracking Images Storage
Browse
Image Format
Accept
Close
Cancel
Browse for Folder
Select the folder you wish to save the Calibration images
My Computer
3½ Floppy (A:)
System (C:)
Scratch (D:)
DVD-RW Drive (E:)
applications on 'stress' (I:)
class on 'canterbury.ac.nz\student' (K:)
setups on 'stress' (L:)
mechapp on 'eng.canterbury.ac.nz\root' (
Library on 'canterbury.ac.nz\universal' (N:
commonapp on 'eng.canterbury.ac.nz\roo
OK
Cancel

Figure 3.11: The process of image format selection and saving path selection

With all the users settings satisfied the "Accept & Run" button in Figure 3.6 is pressed to start the image capture process. The ControlDesk™ layout, at this point should be listening for the connection from the image capture computer. The progress of the image capture process is communicated to the user via the MICA application log window in Figure 3.6.

3.3 Summary

The dSpace™ module acts as the overall controller for the image capture process and is started before the set-up on the ICC begins. The dSpace™ module runs the actuator and activates the camera triggers with user instruction from the ControlDesk™ image capture layout. The user can change the proportional control gain of the LVDT, and the command signal amplitude and frequency for the actuator. The ICC image capture application is activated from the desktop and the user has access to a range of camera and image cycle settings, and can change the active window of the camera. Once all the settings are saved, the image capture process is be started from the main image capture application on the desktop.

Chapter 4

Image Capture Software Structure

This chapter covers the main structure of the code used in the image capture system and how different parts of the code interact with each other in order to capture an image sequence. The description of the code will be simplified to block diagram form, and the more detailed aspects of the code can be found in Appendices A1 and B1. The chapter is thus divided into two main sections:

(1) The operation of the Simulink diagram and Python™ code that is running on the dSpace™ computer, which drives the actuator and supplies trigger signals to coordinate the image capture process.

(2) The main image capture application (MICA), which is responsible for the capturing, processing and storing of the images

4.1 Simulink™ Diagrams and Python™ Code

The Simulink™ diagram is used to generate the trigger pulses for the image capture process and to drive the actuator at the required amplitude and frequency. The diagram is divided up into three main parts:

- Trigger pulse and actuator signal generation
- Data Storage
- Python™ Code Structure

Each section will be explained in more detail with there respective part of the Simulink™ diagram.

4.1.1 Trigger Pulse and Actuator Signal Generation

This part of the diagram deals with incrementing the phase lag on the trigger signals and the generation of the trigger signals themselves. It is this part of the overall diagram on which the Python™ code acts directly to switch the trigger signals on and off. The phase lag is there to introduce a controlled delay in degrees to the trigger signals, enabling sampling of different portions of a sinusoidal response.

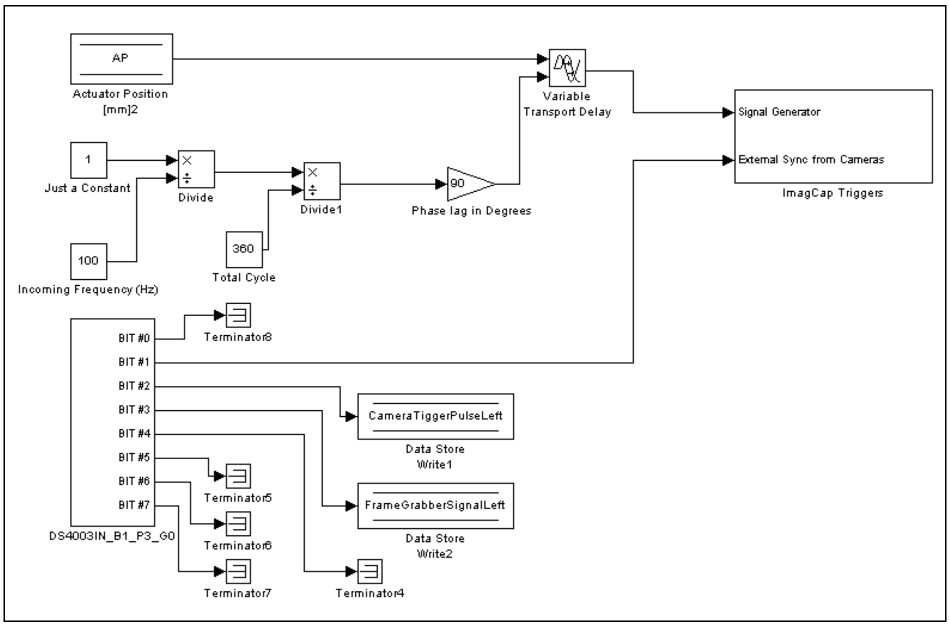


Figure 4.1: The first parts of the trigger generation

The first parts of the trigger generation shown in Figure 4.1 illustrate where the phase lag is introduced into the image capture process. The signal being fed into the trigger generation arrives directly from the LVDT in the actuator. The reason for using the LVDT signal to drive the trigger signal generation is so the triggering of the camera snap can be achieved at pre-defined points in the actuators sinusoidal input cycle.

The variable transport delay is used to introduce the phase lag to the signal being passed to the trigger signal generation block. The Simulink™ block corresponding to the variable transport delay buffers the incoming signal and then feeds it out again, but at a delayed time, defined by the required phase lag. The variable transport delay, requires the delay to be in seconds, and so the phase lag needs to be converted from degrees to fractions of a second. When the LVDT signal enters the trigger signal generation block, it is converted to a square wave oscillating at the actuators frequency between the values of zero and one as shown in Figure 4.2.

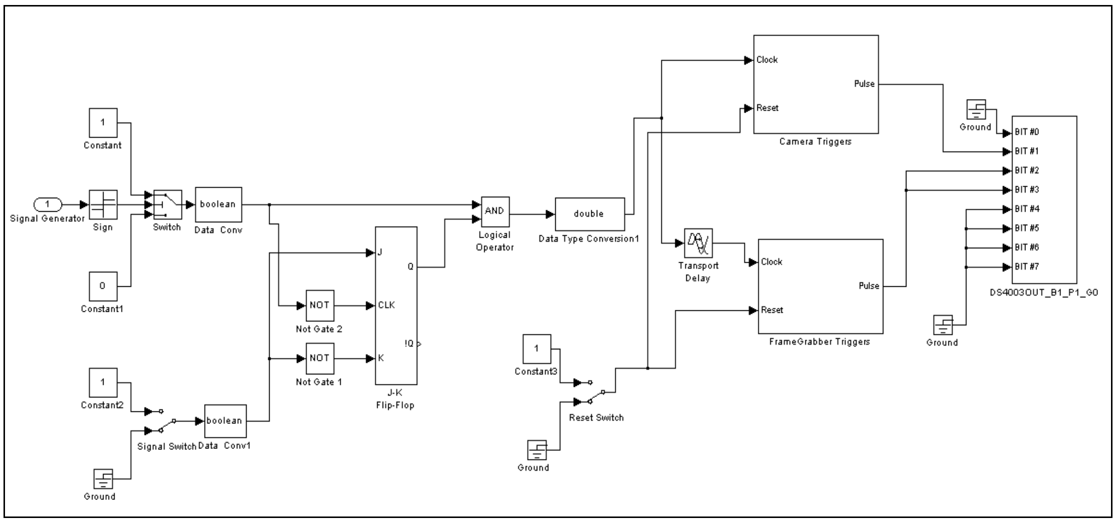


Figure 4.2: The contents of the trigger generation block

Also shown in Figure 4.2, are two switches labelled "Start" and "Reset" that are turned on and off using ControlDesk™, which is driven by the Python™ code. When the "Start" switch is set to the value of '1', the transformed square wave, or clock signal which comes from the sinusoidal LVDT signal, begins to pass to the pulse generation sub-block via the "AND" gate, in Figure 4.2. The "AND" is a logical operator that provides an output of '1' when both inputs are '1' and provides an output of '0' if either input is '0'. The JK flip-flop in Figure 4.2 is another logical operator that is used to align the activation of the "Start" switch with the rising edge of the clock signal to achieve the full pulse width, as illustrated in Figure 4.3. Hence, when the "Start" switch is turned on, and the clock signal is high, there will be two '1's at the "AND" gate allowing the clock signal to pass through, and thus producing a camera snap.

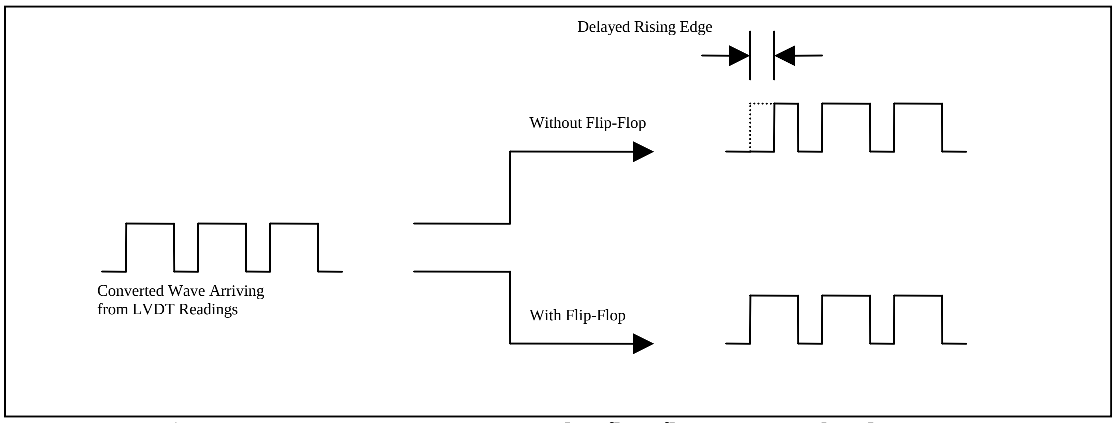


Figure 4.3: The effect of adding the flip-flop gate to the diagram

The camera snap results from a single pulse that is sent to the cameras and the frame grabbers. The pulse is produced in the triggering sub-section block, the contents of which are displayed in the Figure 4.4.

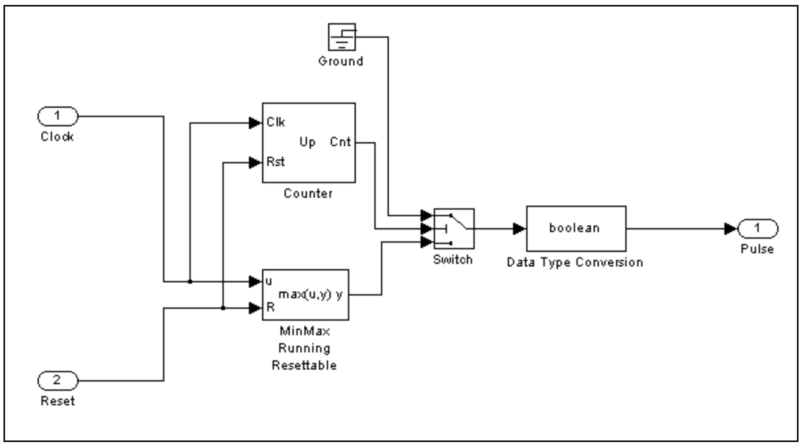


Figure 4.4: The processing of the square LVDT signal into a single trigger pulse of variable duration

This single pulse is taken from the aligned clock signal shown in Figure 4.3, and it is the first rising edge into the triggering sub-section block. Hence, it is important that it is aligned with the correct delay governed by the variable transport delay and not by the random nature of switching the output of the “AND” on half way through an already risen pulse, producing a delay rising edge at the output of the “AND” gate. The process of producing a trigger pulse starts with the clock signal that is fed into a counter, and a maximum value block, which grabs the maximum value of the clock signal, see Figure 4.4.

The switch to the right of the counter and maximum value blocks is a threshold switch that controls the trigger pulse width to the camera. The switch takes the highest value of the incoming signal, which is the output of the maximum value block of Figure 4.4, and holds it there for a specified number of counts, in this case 50 counts. Each count is registered by the counter after a rising edge in the clock signal, which occurs at the frequency of the LVDT signal and consequently the actuator frequency. Once the 50 counts are delivered to the threshold switch, the output is changed to ground completing the trigger pulse. The counter and the maximum value block undergo a reset via the "Reset" switch shown in Figure 4.2. The threshold of the switch can be changed to achieve a longer pulse width by increasing the number of counts it requires to do so. A longer pulse width may be desired if more than one frame from the camera is required, in which case the trigger to the camera must be held high for a longer period of time.

The generation of the signal for the actuator is achieved from the signal generation block in Figure 4.5. The incoming signal is passed through proportional, integral and derivative gains; as shown in Figure 4.5, where the integral and derivative control gains are set to zero for stability reasons as discussed in Chapter 3.

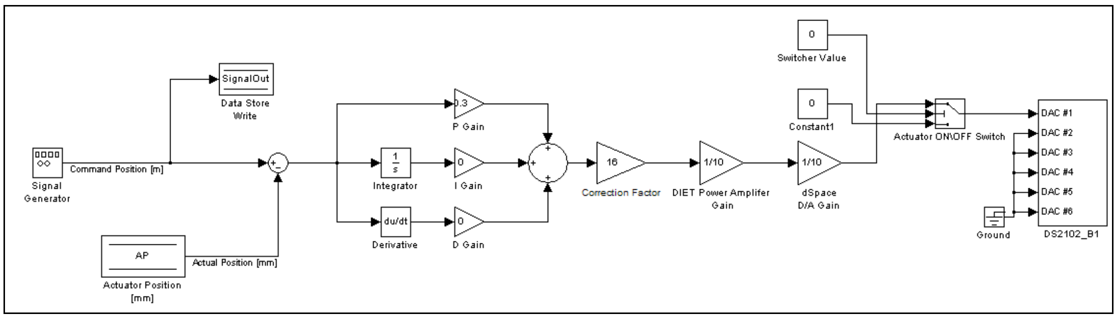


Figure 4.5: The actuator signal generation and control part of the Simulink™ diagram

4.1.2 Data Storage

Voltage data arrives from the LVDT into the dSpace™ module via the A/D (Analogue to Digital) lines, and enters the Simulink™ diagram via the 'DS2001_B1' block shown in Figure

4.6. The incoming signal is stored in a local memory block making the data available to all blocks that may require it. However, before the data reaches the storage block, the voltage data is converted to displacement in millimetres using Equation (1) of Chapter 2. The block representation of Equation (1) is shown in Figure 4.6.

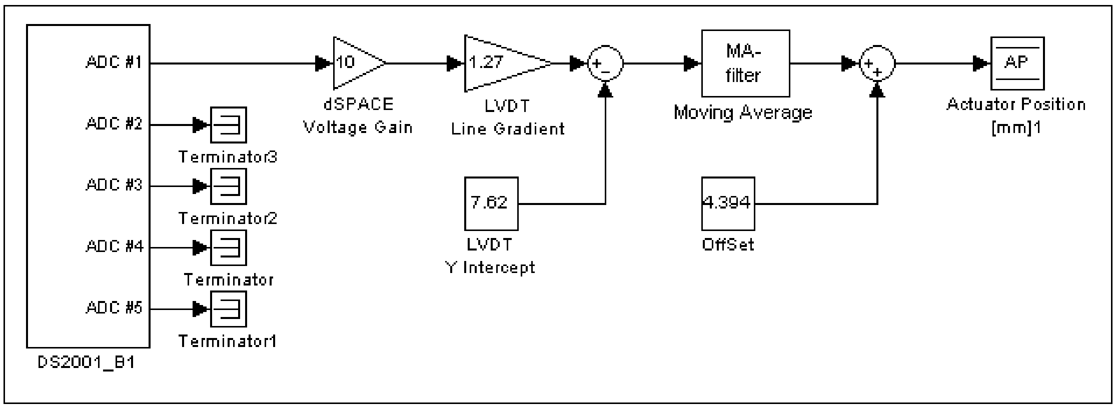


Figure 4.6: The blocks that deal with data arriving back from the LVDT

4.1.3 Python™ Code Structure

The Python™ code drives the operation of the ControlDesk™ in Figure 3.2 of Chapter 3. The Python™ code's operation is detailed in Figure 4.7.

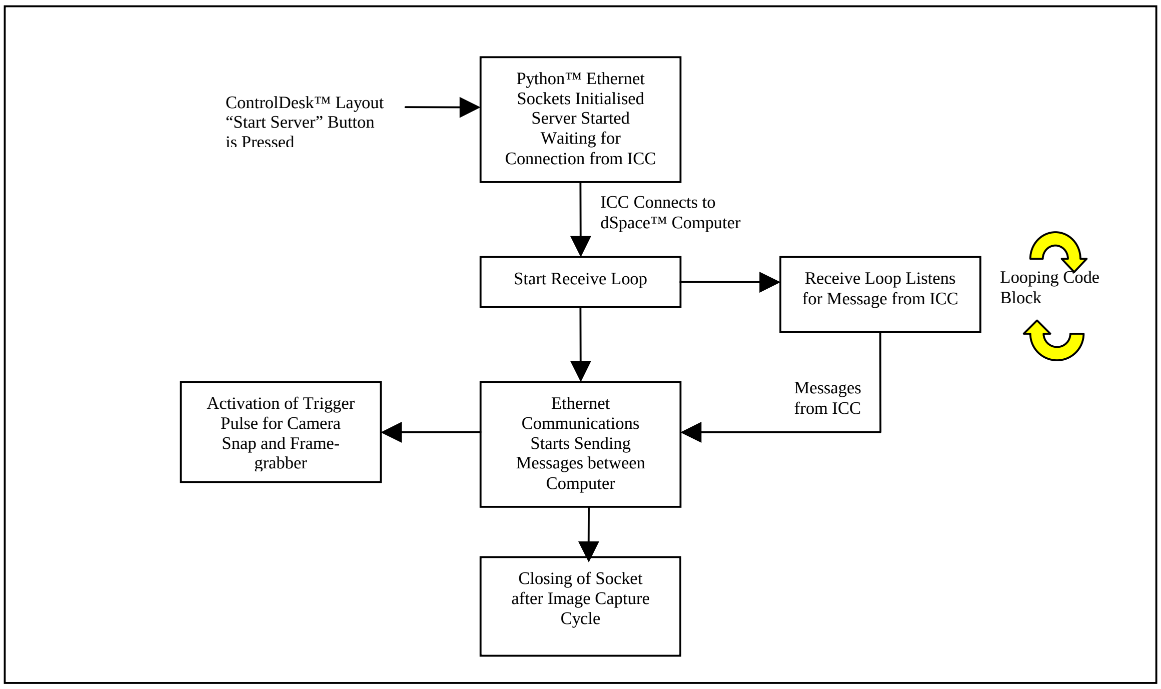

ControlDesk™ Layout
"Start Server" Button
is Pressed
Python™ Ethernet Sockets Initialised Server Started Waiting for Connection from ICC
ICC Connects to dSpace™ Computer
Start Receive Loop
Receive Loop Listens for Message from ICC
Looping Code Block
Messages from ICC
Activation of Trigger Pulse for Camera Snap and Frame-grabber
Ethernet Communications Starts Sending Messages between Computer
Closing of Socket after Image Capture Cycle

Figure 4.7: The main structure for the Python™ code

Once the “Start Server” button is pressed the main part of the Python™ code creates a communication socket and waits for a connection attempt from the ICC. When the ICC connects with the dSpace™ computer the receive part of the code is started and the main Ethernet communications begin. Upon notification from the ICC that the cameras and frame-grabbers are ready, the “Start” and “Reset “ switch are turned on and off, to produce the trigger pulse to the camera and frame-grabbers. After each trigger pulse is sent, the phase lag between the trigger and the actuator position is incremented in degrees in the gain block shown in Figure 4.1, using the same Python™ code. Once the image capture cycle has ended the Python™ Ethernet sockets close, signalling to the ICC to do the same.

4.2 The Main Image Capture Application (MICA)

The settings window for each camera, shown in Figures 3.8 and 3.10 of Chapter 3, are activated from the MICA. The MICA code structure is broken down into the following sections:

- Video Streaming
- Ethernet Communications
- Triggered Image Set-up
- Camera Settings Dialog

These sections of code are arranged as shown in the flow diagram in Figure 4.8. The video streaming starts as soon as the MICA is started and continues to stream video data from the cameras until the user starts the image capture process. The sections of code corresponding to the camera settings begin to run when the user wishes to change the colour gains or the size of the active capture window. This process is shown in Figure 3.8 of Chapter 3.

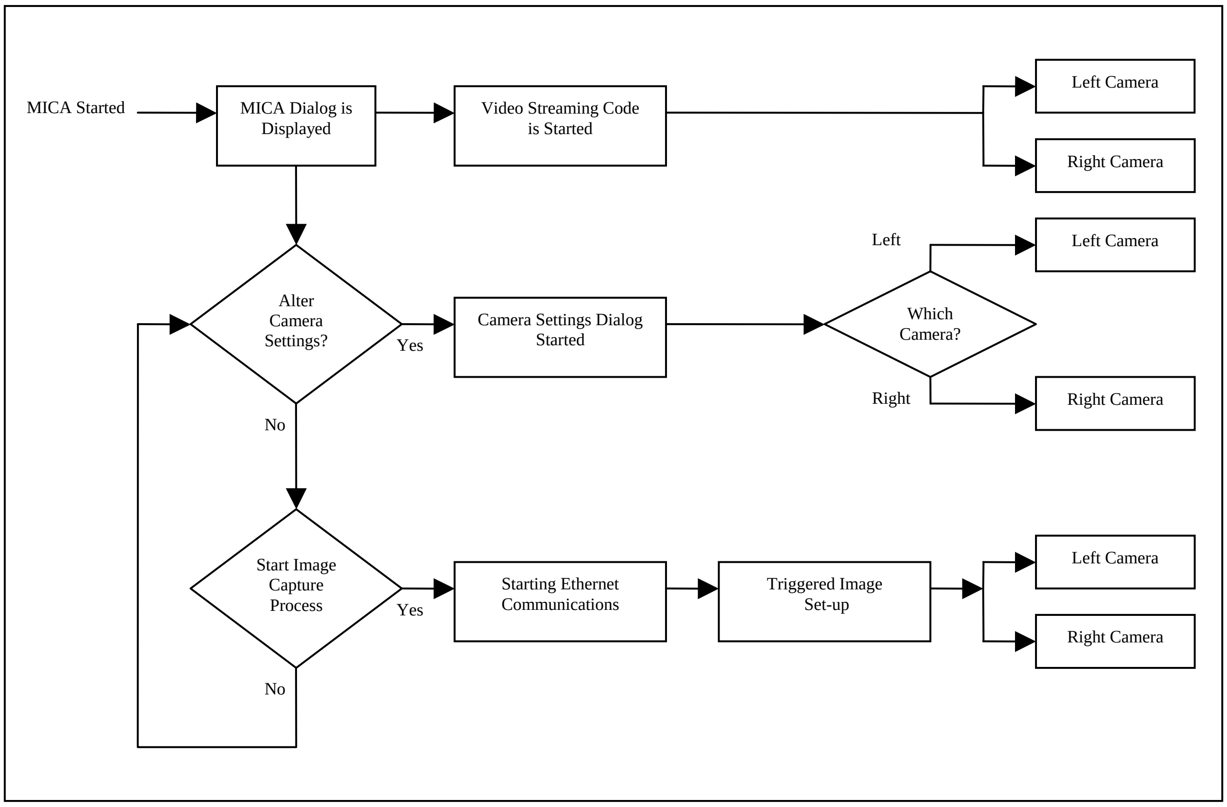


Figure 4.8: The overview of MICA code structure

The video streaming portion of the code can be further broken down into the flow diagram shown in Figure 4.9. Once the video streaming is started the I^{2}C adapter applies the first of two configurations to the cameras, which puts them in a state to stream pixel data to the frame grabbers in a form that they can process. In the image capture system, OpenGL™ is used to scale and display the pixel data arriving from the cameras. This approach requires the OpenGL™ libraries to be initialised first, whereby windows are created that are attached to the OpenGL™ objects. These windows are where OpenGL™ places the scaled pixel data.

A session is then started with the frame grabbers. This session initialises the frame grabbers and gets them ready to receive the pixel data arriving from the cameras. The first part of the video streaming code structure deals with the initialisation of the components required to display the images on the screen. Subsequently, as shown in Figure 4.9, the code enters a while loop where each cycle of the loop one frame of pixel data is grabbed from the frame grabbers.

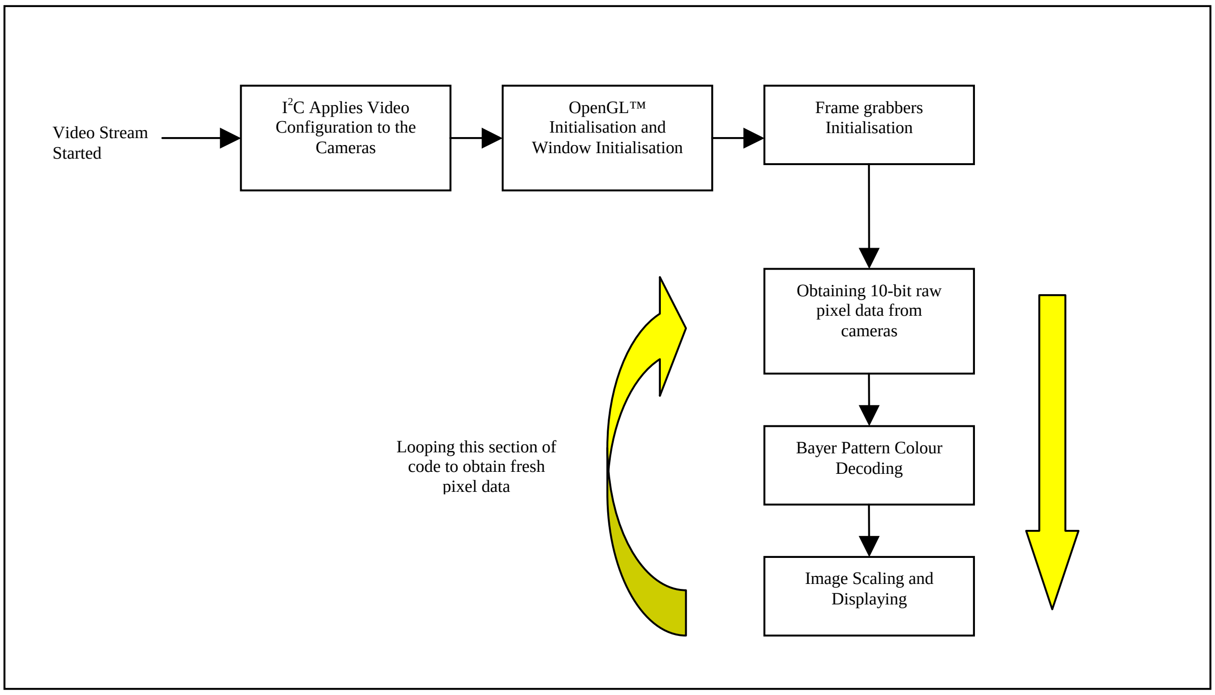

Video Stream Started
I²C Applies Video Configuration to the Cameras
OpenGL™ Initialisation and Window Initialisation
Frame grabbers Initialisation
Obtaining 10-bit raw pixel data from cameras
Bayer Pattern Colour Decoding
Image Scaling and Displaying
Looping this section of code to obtain fresh pixel data

Figure 4.9: Video Streaming code structure

Once the image capture process is started in Figure 4.8, the Ethernet communications that control the image capture process on the ICC begin. An overview of this section is shown Figure 4.10 and it is broken up into five main parts. The first part of the Ethernet communications deals with the initialisation and opening of the socket communications, which enables the dSpace™ computer and the ICC to communicate. The second part, is the starting of the receive loop, which runs constantly throughout the image capture process. It runs in a loop independent of the ethernet code block, and relays any message from the dSpace™ computer to the code block. One such message would be 'CamerasReady', which notifies the dSpace™ computer that the cameras are ready to receive the trigger pulse to 'snap' and image.

After the receive loop has begun the commanded events loop is started to allow other functions like the message log and the receive loop in the MICA to continue to run. The next block in Figure 4.10 is the triggered events code section that obtains processed commands from the receive loop block. Based on those commands the code block performs specific tasks, including the triggered image set-up and the sending of data and commands to the dSpace™ computer.

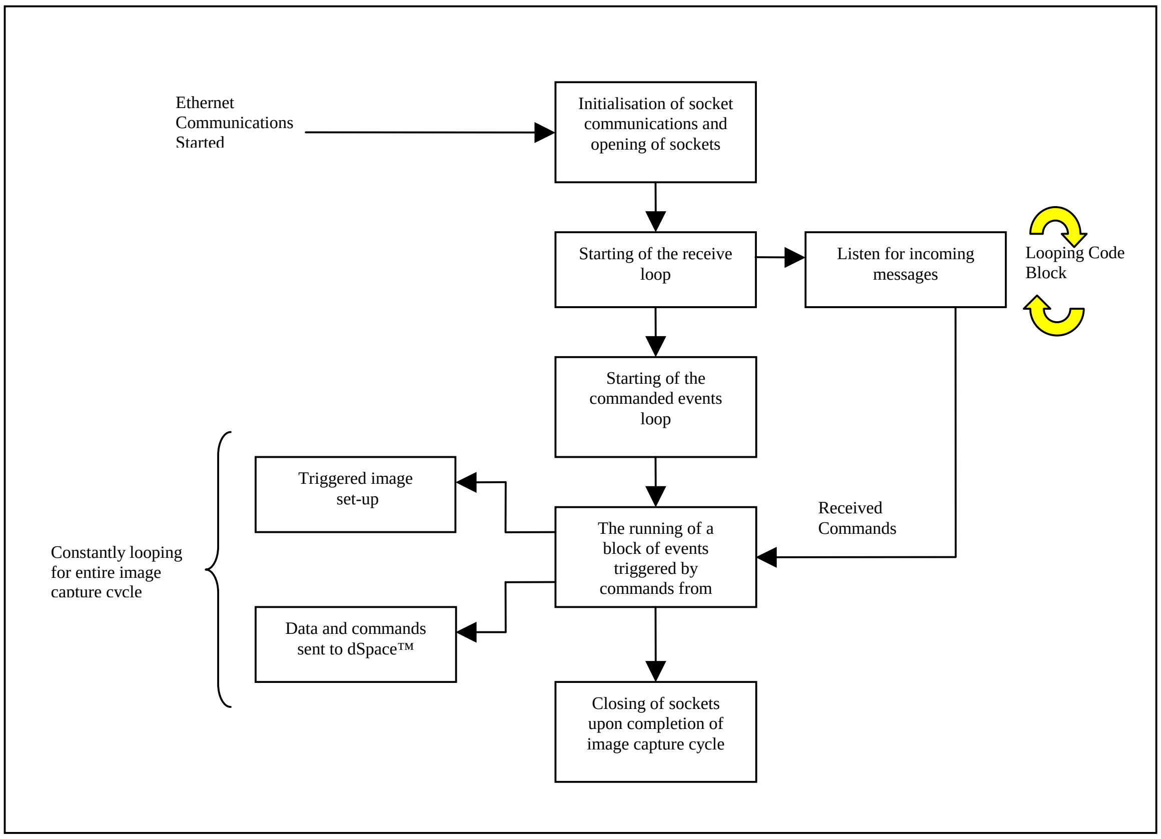

Ethernet Communications Started
Initialisation of socket communications and opening of sockets
Starting of the receive loop
Listen for incoming messages
Looping Code Block
Starting of the commanded events loop
Triggered image set-up
Received Commands
Constantly looping for entire image capture cycle
The running of a block of events triggered by commands from
Data and commands sent to dSpace™
Closing of sockets upon completion of image capture cycle

Figure 4.10: An overview of the Ethernet Communications code block

The final block is reached upon the completion of the image capture cycle where commands are first sent to the dSpace™ computer to close its communications sockets, and then the ICC waits and closes its sockets.

Following the starting of the Ethernet Communications, the triggered image set-up part of the code is begun, as shown in Figure 4.8. This block is explained in more detail in Figure 4.11. Before acquiring the triggered images the I^2C adapter applies the second of the two configurations to the cameras, which puts them into the state where they are listening for the trigger pulse from dSpace™. The frame-grabbers are then initialised.

Note that any one frame-grabber cannot run multiple sessions at once. Hence, before the triggered image set-up the session for streaming the video from the cameras must be terminated. A new session always requires fresh initialisation of the frame-grabbers. Following frame-grabber initialisation there is the initialisation of the CImage class. This class is produced by Microsoft Windows™, and is used to encode the pixel data to a chosen image format.

At this point, the frame-grabbers need to be configured to receive a trigger pulse from dSpace™ into one of their external trigger lines. The code block then stops and waits for the trigger from dSpace™, as shown in Figure 4.11. Once the trigger has been received, the pixel data is taken directly from the memory buffers on the frame-grabber cards themselves. The triggered pixel data is then passed to the CImage class to encode the image to a user-defined image format and stored in a user-defined path.

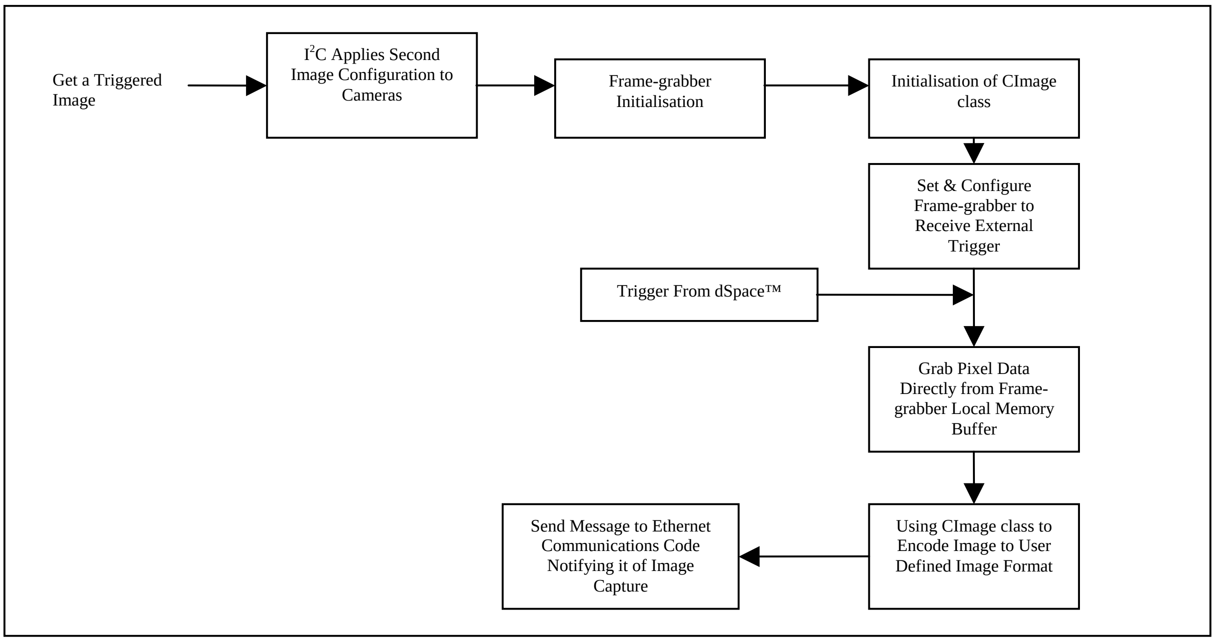

Get a Triggered Image
I²C Applies Second Image Configuration to Cameras
Frame-grabber Initialisation
Initialisation of CImage class
Set & Configure Frame-grabber to Receive External Trigger
Trigger From dSpace™
Grab Pixel Data Directly from Frame-grabber Local Memory Buffer
Using CImage class to Encode Image to User Defined Image Format
Send Message to Ethernet Communications Code Notifying it of Image Capture

Figure 4.11: The Triggered Image Set-up Code Block

When the entire process is successful, a message is sent to the Ethernet communication code block, notifying it of a successful capture and to proceed with the rest of the image capture process.

4.2.1 The Camera Settings Dialog

The camera-setting dialog can be broken down into the following sections of code:

- Image Manipulation Tab
- Image Storage Paths Tab

These two sections are activated at the request of the user as shown in Figure 4.12.

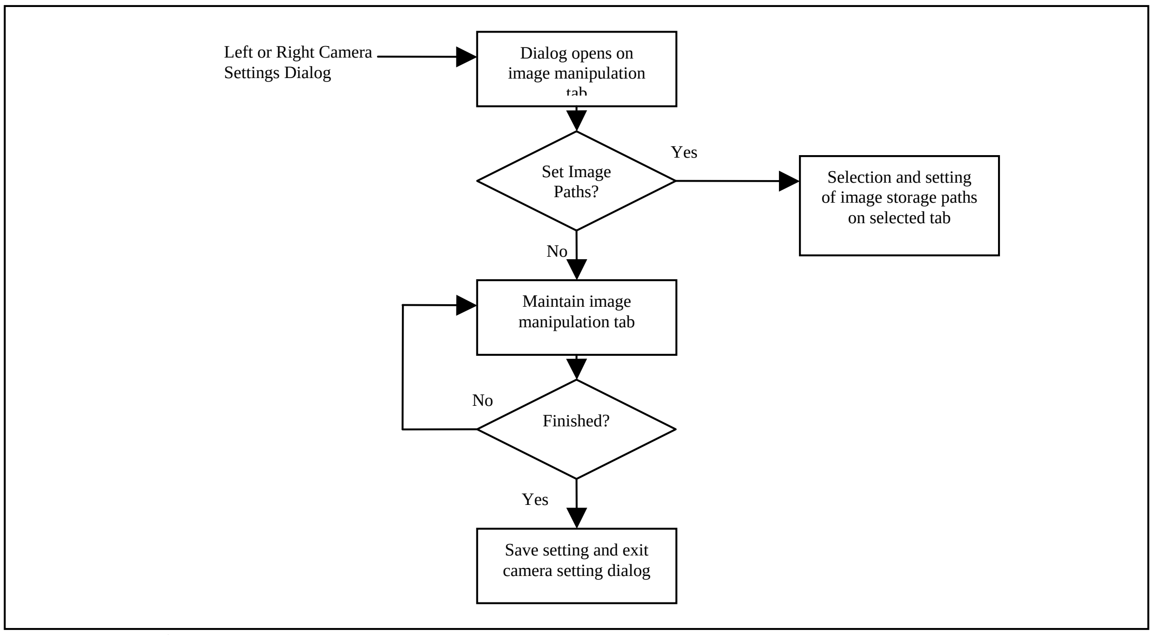

Left or Right Camera Settings Dialog
Dialog opens on image manipulation tab
Set Image Paths?
Yes
Selection and setting of image storage paths on selected tab
No
Maintain image manipulation tab
No
Finished?
Yes
Save setting and exit camera setting dialog

Figure 4.12: The overview of the camera setting dialog box code blocks

The code that runs the image manipulation tab can be further broken down into the components shown in Figure 4.13. The image manipulation tab is first started when the camera settings dialog is started and it begins to stream video from the camera that needs its settings changed. The video is displayed on the screen, in this case using the CImage class which is used to save the captured images. The colour gains for the image are changed by moving the sliders or by entering the value directly into the edit box to the left of the slider.

The sliders are a bar that is selected using the left mouse button, and while continuing to hold that button down allows the mouse to move the slider bar to the left or right of its designed limits to increment colour gain values, as shown in Figure 3.8. Any changes to these GUI controls trigger an event within the code. The triggered event then takes the values from the slider or edit box and applies them to the Bayer pattern decoding function in the main loop. This change happens in real time to allow the user to sample the different gains.

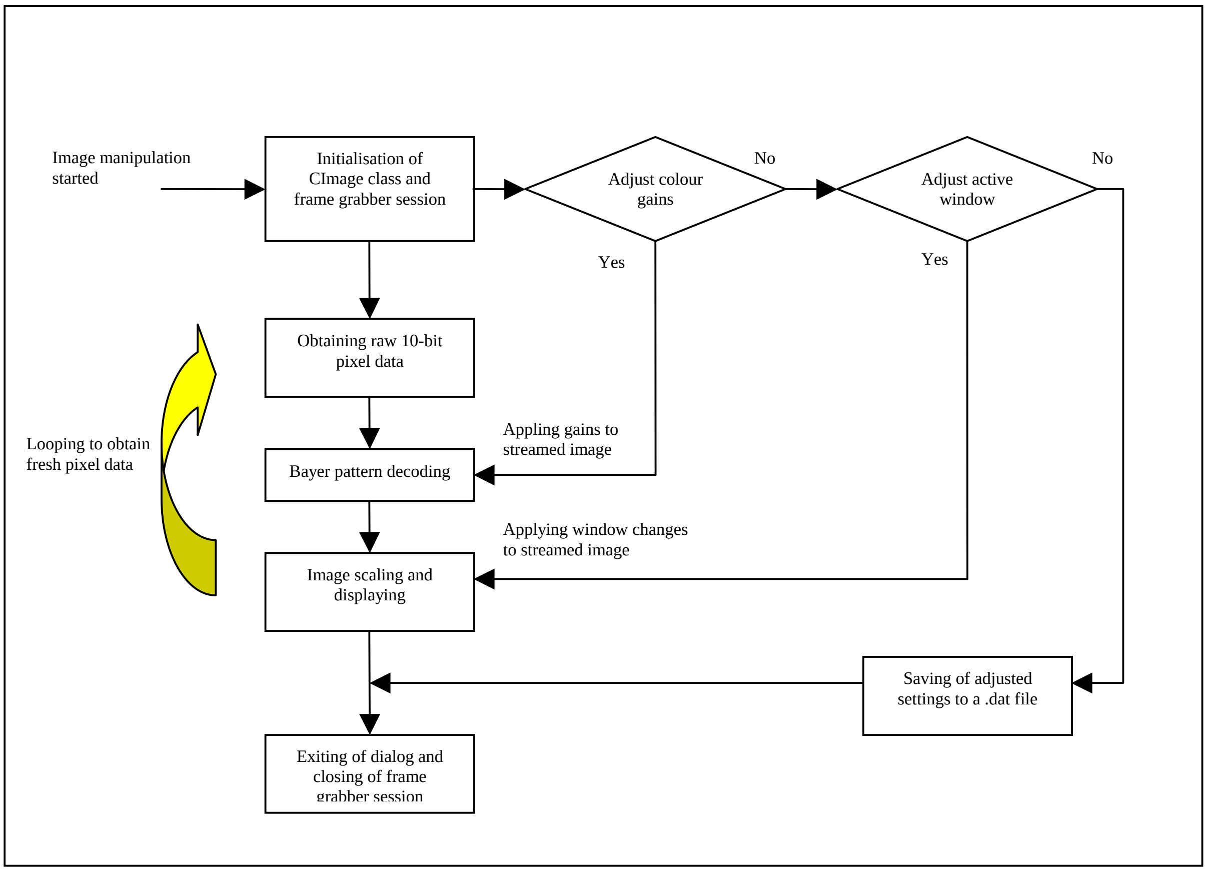


Image manipulation started
Initialisation of CImage class and frame grabber session
Adjust colour gains
No
Adjust active window
No
Yes
Yes
Obtaining raw 10-bit pixel data
Looping to obtain fresh pixel data
Appling gains to streamed image
Bayer pattern decoding
Applying window changes to streamed image
Image scaling and displaying
Saving of adjusted settings to a .dat file
Exiting of dialog and closing of frame grabber session

Figure 4.13: An overview of the Image Manipulation code block

If the user wants to change the size of the active window, the values can be entered into the dialog in the edit boxes for the active window. Alternatively the mouse selection check box can be ticked and the mouse can be used to perform this task. The launching of the mouse selection creates two cross hairs, as shown in Figure 3.10, that which follows the mouse cursor across the streamed video image. The cross hairs themselves are drawn directly to the incoming video streaming functions that are part of the CImage class.

When the left mouse button is double clicked it triggers an event that grabs the mouse cursor current position in screen co-ordinates, scales them to the streamed video window co-ordinates, and places a marker box at the recently chosen point on the image. The same process is done for a second point, usually in the bottom right hand side of the video window. These two points define a rectangle which is the new active window size. The second mouse point selection contains the new window size and triggers an event that updates the edit boxes for the active window selection.

A message box then appears asking the user whether they wish to update the video image. If yes, the image is scaled to the video window, giving the user a chance to sample different active window scenarios. Once the user has chosen the required settings, the accept button is pressed, and all the changed settings are stored in a .dat file under a name defined by the application. They are stored in this .dat file for two reasons. The first reason is it allows the settings to be transfer to the MICA, and the second reason is to give the user the option of applying the same settings to the cameras for repeated image capture cycles.

There are two image storage paths for each camera. The first path is for the storage of the calibration images, and the second path is for the captured images. The blocks of code arranged in Figure 4.14 illustrate their interaction with the user. The image storage dialog interacts with the tab class the same way as the image manipulation dialog. Once the image storage dialog is started, a .dat file containing five previous paths for the images is loaded into the drop down list box.

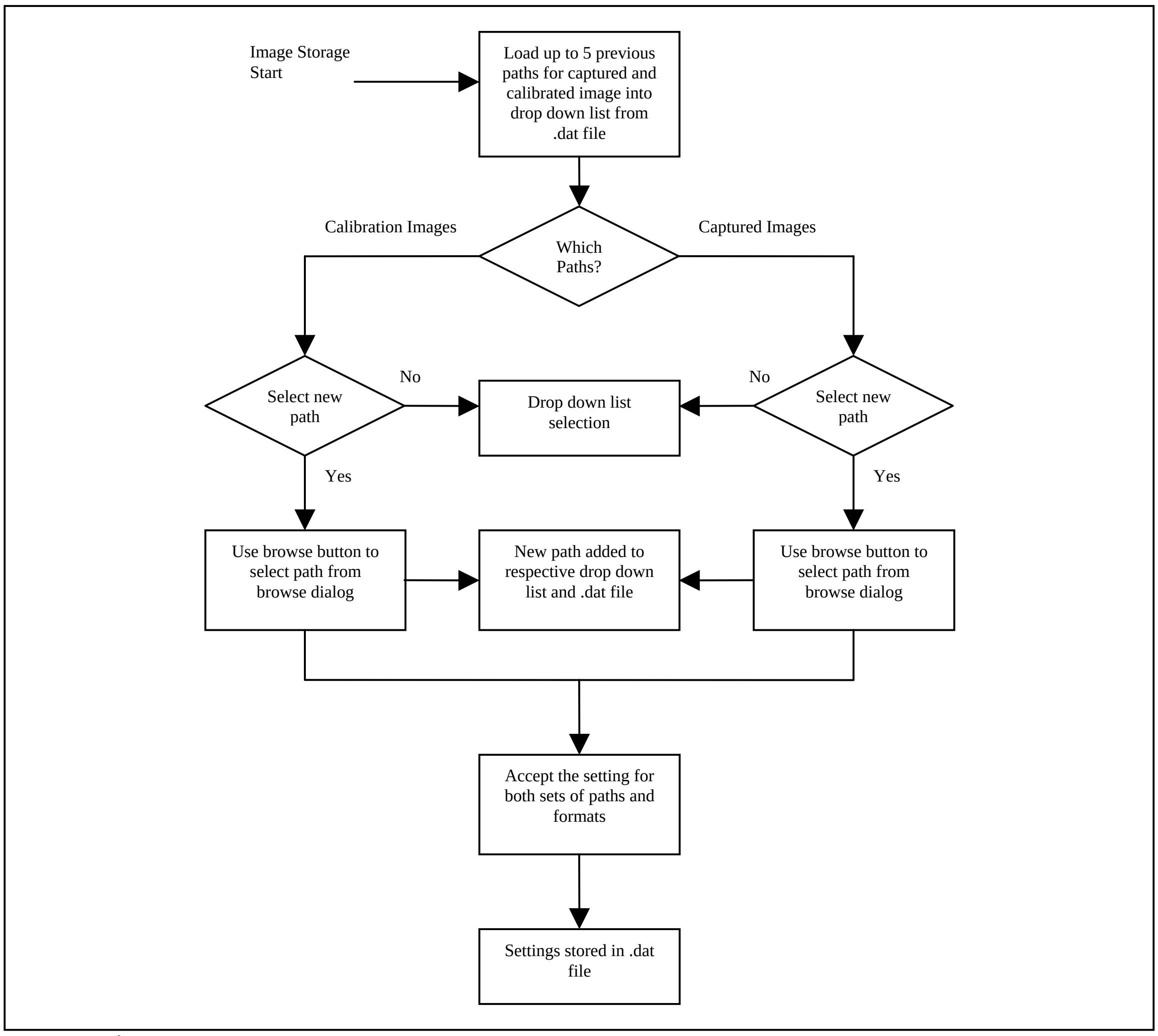


Image Storage Start
Load up to 5 previous paths for captured and calibrated image into drop down list from .dat file
Calibration Images
Which Paths?
Captured Images
Select new path
No
Drop down list selection
No
Select new path
Yes
Yes
Use browse button to select path from browse dialog
New path added to respective drop down list and .dat file
Use browse button to select path from browse dialog
Accept the setting for both sets of paths and formats
Settings stored in .dat file

Figure 4.14: An overview of Image Storage operation and code block interaction

Once the dialog has been initialised the user can select the path for the respective types of images. The process of path selection is the same for both image types.

4.3 Summary

The image capture software is divided up into two separate parts Simulink™ and Python™ running on the dSpace™ computer and Visual C++ running on the ICC. The software running on the dSpace™ computer drives the actuator and controls the trigger pulses that are sent for the cameras, the frame grabbers and the strobe. These functions are performed by the Simulink™ diagram built for the dSpace™ module and is controlled by the Python™ software. The Python™ software then "talks" to the Visual C++ software running on the ICC, which manages the running of the cameras, the frame grabbers, and the storing of captured images. The Visual

C++ software and the Python™ software communicate with one another to achieve a complete image capture cycle, and this communication occurs through an Ethernet link between the two computers.

PART III

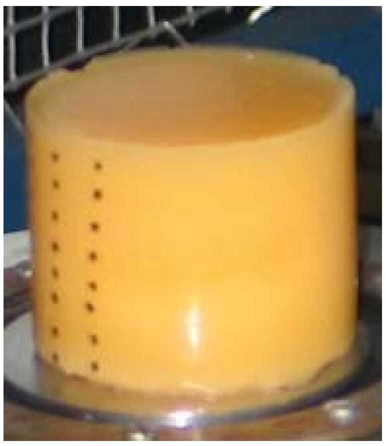
RESULTS

Chapter 5

Image Capture Results

5.1 Preparation

In this chapter, the application of the image capture system is examined using a silicon phantom, moulded in a cylindrical shape. The silicon phantom used in this experiment is a two-part mix solid silicon elastomer, and is as shown in Figure 5.1. The silicon gel formula and ingredients are sourced from Factor II Inc.



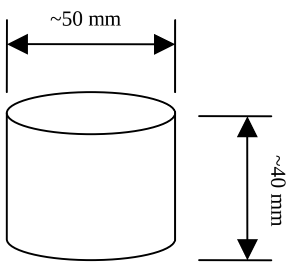


Figure 5.1: The silicon phantom used in the experiments and its dimensions

This silicon polymer was chosen because of similarities with the elastic properties of human tissue. The preparation of the silicon phantom begins by building a mould, which in this case comprises of a short section of PVC piping sealed at one end. The silicon polymer arrives as two components part ‘A’ and part ‘B’, mixed in a ratio of 10% of ‘A’ for the total volume of ‘B’ into which ‘A’ is being mixed. Part ‘B’ is the actual silicon rubber itself and arrives as a liquid rubber. The part ‘A’ solidifies part ’B’ into a silicon rubber solid of a specific stiffness.

The phantom and mould are is then placed under a vacuum to remove all air bubbles caused by mixing the two components together, and the solution is poured into the mould and allowed to set. The dots seen on the silicon phantom are applied manually using water-based paint and are used to aid in the tracking surface motion when actuated.

5.2 Preliminary Image Capture Results and Problems

The initial tests of the image capture set-up revealed an unforeseen problem. The problem was that the time period between when camera fired the strobe flash and the camera received the trigger pulse to snap a frame was inconsistent and unpredictable. This inconsistency makes it very difficult to align the strobe trigger with the phase position in the actuators cycle.

Specifically, consider the case of the actuator moving at a frequency of 100 Hz, which is the maximum frequency required of this image capture system. Thus, the actuator and silicon phantom move through a 360-degree cycle every 0.01 seconds. The problem is that while the time periods between the dSpace™ trigger pulse and the strobe trigger pulse are consistent at a time resolution of 0.1 seconds; it can fluctuate randomly over the finer resolution of 0.01 seconds. This behaviour is demonstrated in Figure 5.2 for the left camera over 7 images at a phase lag of 0-degrees and a 100Hz signal. In other words, the camera is not designed for precision strobing greater than 10 Hz image capture frequency.

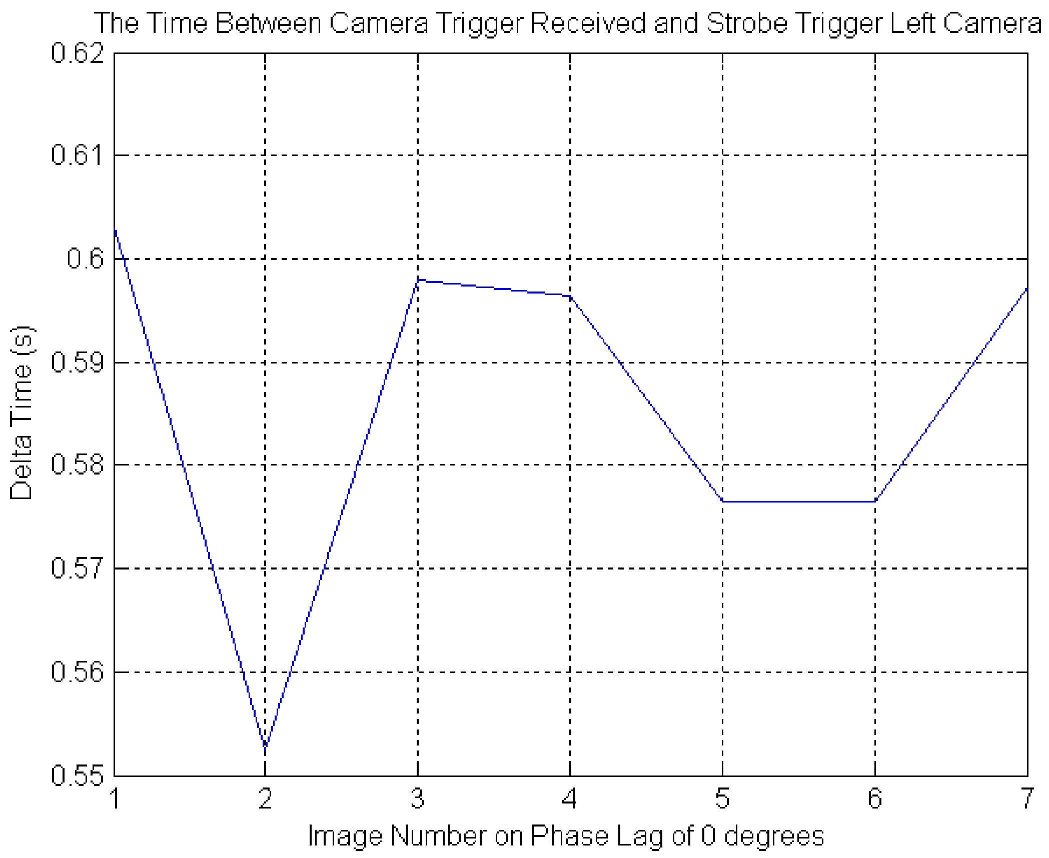
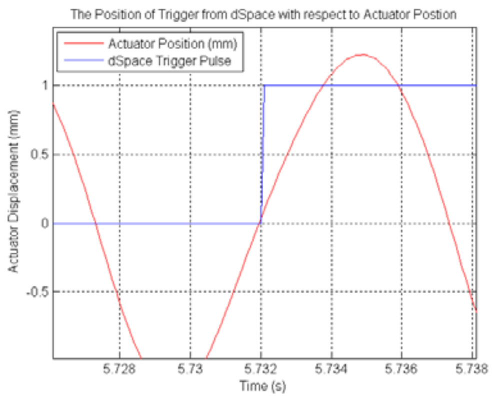
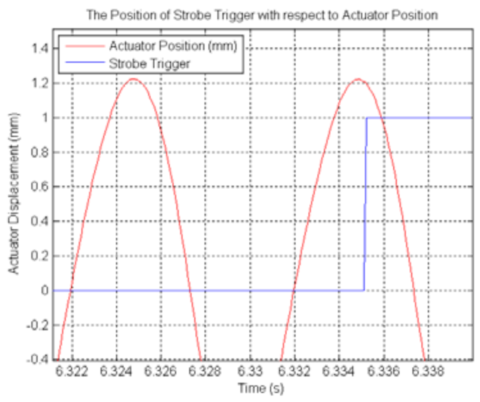


Figure 5.2: A graph of the dSpace trigger pulse/strobe trigger pulse time period

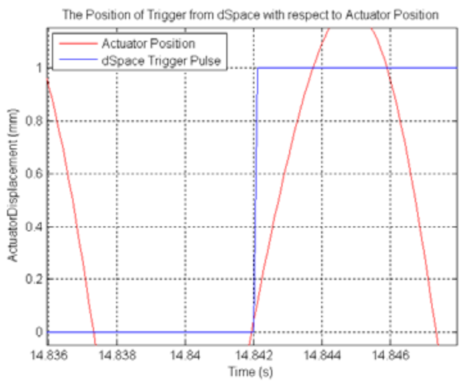
Figure 5.3 shows the result of the fluctuations in Figure 5.2 on the position of the actuator and consequently the silicon phantom for the first three images taken. Figures 5.3 (a), (c) and (e), illustrate that the dSpace™ module consistently sends the snap trigger at almost exactly zero degrees of phase lag. Figures 5.3 (b), (d) and (f) show the inconsistent triggering of the strobe from image to image and where it occurs in the actuators motion. The resulting images are therefore captured at almost random points of the actuators cycle.



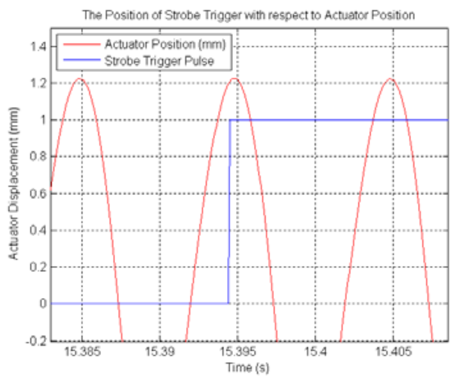
(a)



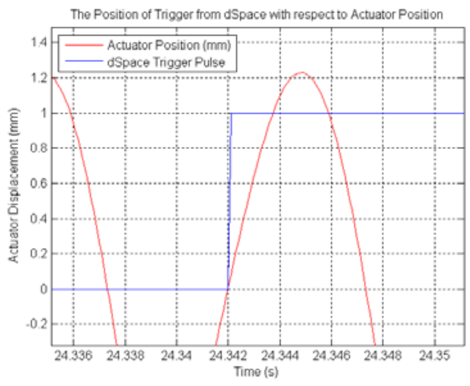
(b)



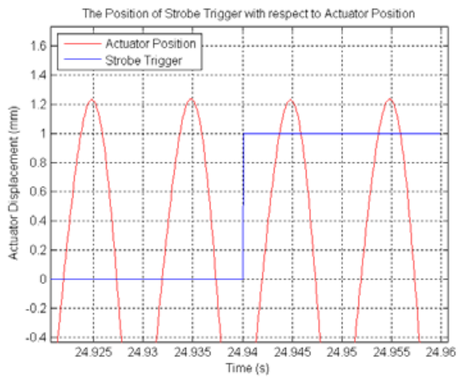
(c)



(d)



(e)



(f)

Figure 5.3: The dSpace trigger pulses and strobe trigger pulses on a phase lag of 0-degrees over three images with the strobe trigger times from Figure 5.2.

A second problem was also discovered involving external sync time variations between the two cameras. For example, triggering the strobe using the right camera resulted in partial images from the left camera on random phase lags. This problem was due to the left camera not being ready for the strobe to flash at the same time as the right camera, even though the dSpace trigger pulse was sent to the cameras at the same time.

5.3 Corrected Image Capture Results

In this section the following problems are addressed and results presented:

- Inconsistencies in required strobe trigger between cameras
- An inconsistent time period, at the required time resolution, between the dSpace trigger pulse and the strobe trigger pulse

Both of these problems are addressed simultaneously with the introduction of solid state AND gate and a feed back pulse to the dSpace™ module, which triggers the strobe flash. As shown in Figure 5.4, the solid state AND gate is attached to the two external sync lines from each camera, and this addition aligns the two pulses into one coherent pulse that is fed into the dSpace™ module.

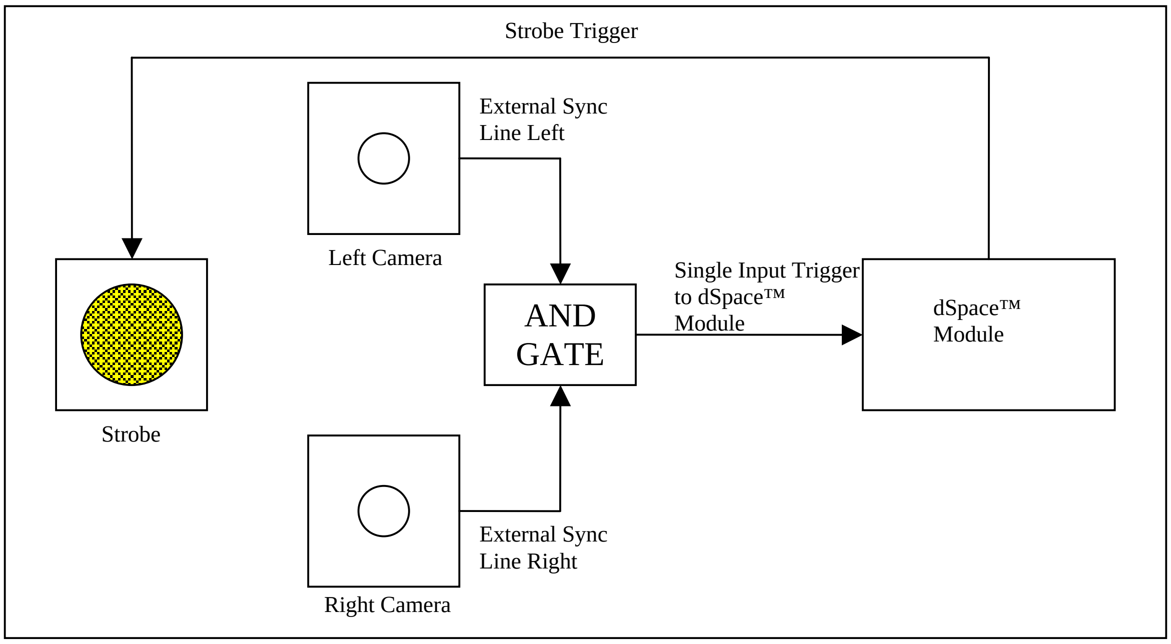


Figure 5.4: The new strobe trigger set-up

The single coherent external pulse sent to the dSpace™ module from the AND gate is then aligned with the rising edge of the now phase-lagged LVDT signal, which is then passed back out to trigger the strobe. The single external strobe trigger is aligned in much the same way as the camera snap trigger pulse using flip-flops Simulink™ blocks, as shown in Figure 5.5.

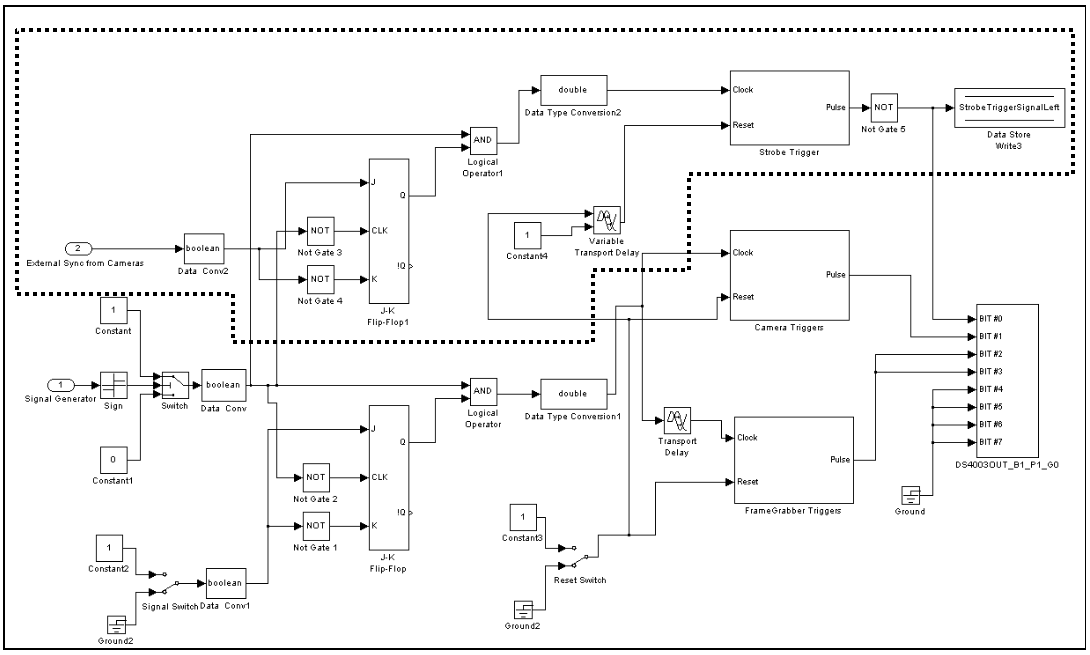


Figure 5.5: The Simulink™ blocks added (within the dashed region) to the trigger generation sub-block of Figure 4.2 to deal with firing the strobe

The strobe itself actually triggers on the falling edge of the trigger pulse and not the rising edge because of the way the strobe was designed by the manufacturer. Due to the falling edge trigger, a ‘NOT’ gate is used to invert the edge. The implementation of this ‘NOT’ gate achieves the result shown in Figure 5.6.

Each rising edge of the square wave LVDT lagged signal occurs at a specific point in the actuators motion. This rising edge is moved to other positions in the actuators motion, by adjusting the phase lag desired. It is this rising edge that the strobe must trigger on to capture the actuator at that specific position. The external sync from the cameras notifies the image capture system that the cameras are ready for the strobe flash, but the flash must occur on the rising edge of the lagged LVDT signal. Thus, the external sync must remain high until the next rising edge in the lagged LVDT signal, which is at 0.01s intervals for 100Hz actuation.

Since the external sync therefore may remain high for ~0.25-0.3s, there is plenty of time for the strobe to receive a rising edge and thus trigger a camera flash. The result of the solid state AND gate and feedback pulse to the dSpace™ module applied to the apparatus outlined in Chapter 2, allows the strobe to trigger at the points shown in Figure 5.7

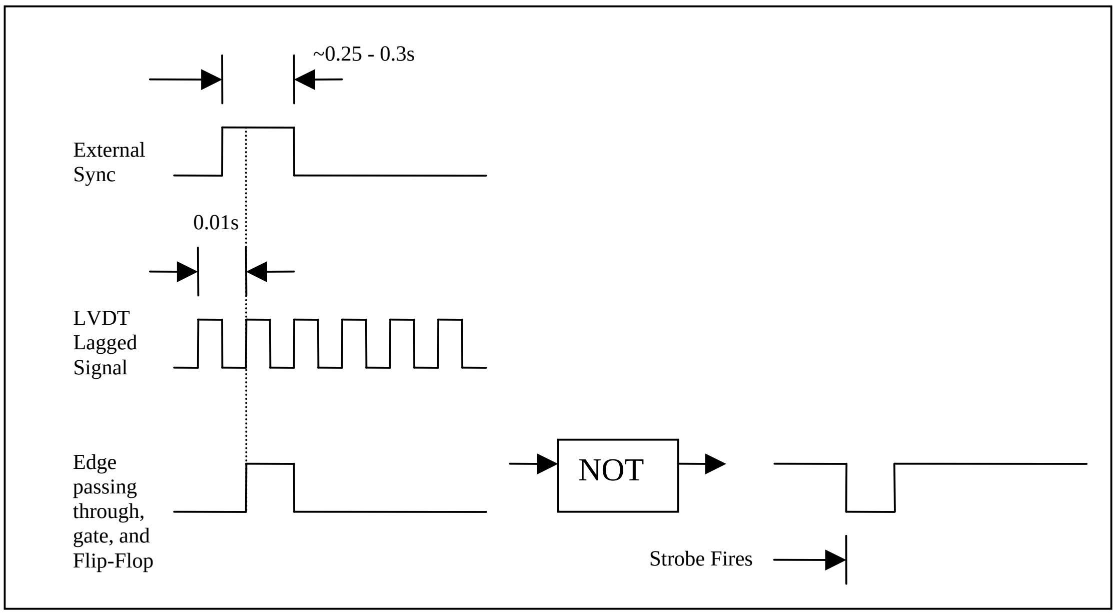


Figure 5.6: The timing of the modified strobe trigger

Figure 5.7 compares the actual strobe triggering time with the ideal, where the mean absolute error is approximately 1.4%. This demonstrates the accuracy of the timing of the strobe firing. The gradient of the graph shown in Figure 5.7 of the ideal strobe trigger times follows the experimental strobe triggers very closely. This result indicates that the progression of the phase lag over 0-360 degrees is evenly distributed. Furthermore, the complete phase lag occurs almost precisely in 0.01s, indicating the image capture process has occurred within a 360 degree waveform, oscillating at 100Hz.

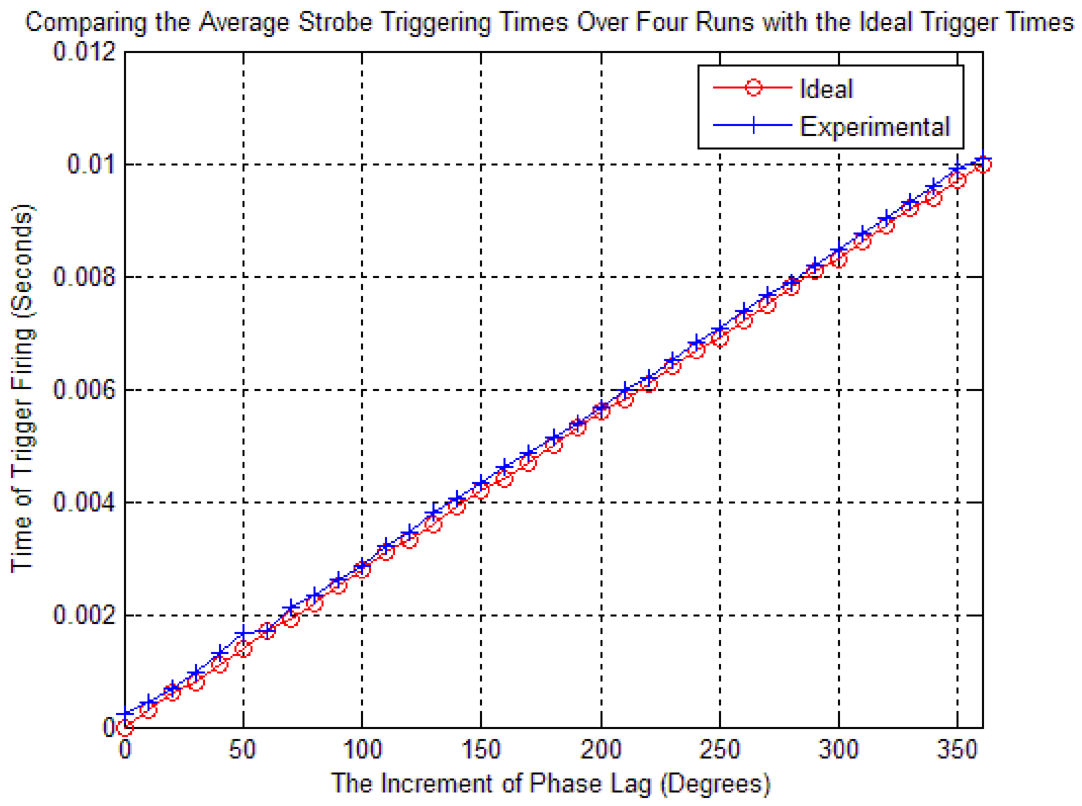


Figure 5.7: The comparison of the average strobe triggering times over the four image capture runs compared with the ideal triggering times.

It should also be noted that the actuator has imperfections or error in its motion. A snap shot of the actuators motion at t = 1.8207s compared with an ideal case shown in Figure 5.8(a), shows a time period where the difference in the two waveforms is at a minimum corresponding to the best achievable actuator motion. The variation in frequency for Figure 5.8(a) is between ~98 to 100Hz or less than 2%.

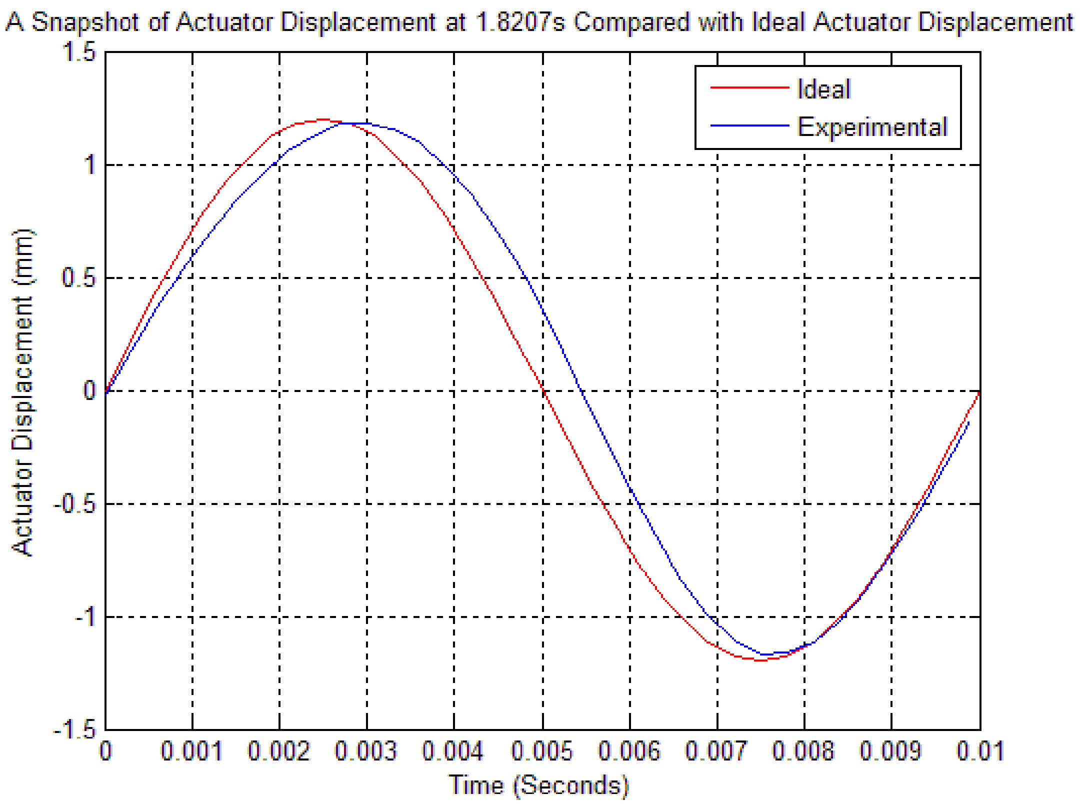


Figure 5.8(a): The actual position of the actuator compared to an Ideal position at a snapshot of the actuators motion taken at 1.8207 seconds.

An average of 20,000 actuator waveforms was taken, and the result is plotted in Figure 5.8(b) showing that a larger error dominates the motion. Since Figures 5.8(a) and (b) involve direct measurements of the actuator displacement, errors can be attributed to the dynamics of the actuator itself. In other words, there are physical limitations in the current actuator. Further experimental work and potential improvements need to be done in the future.

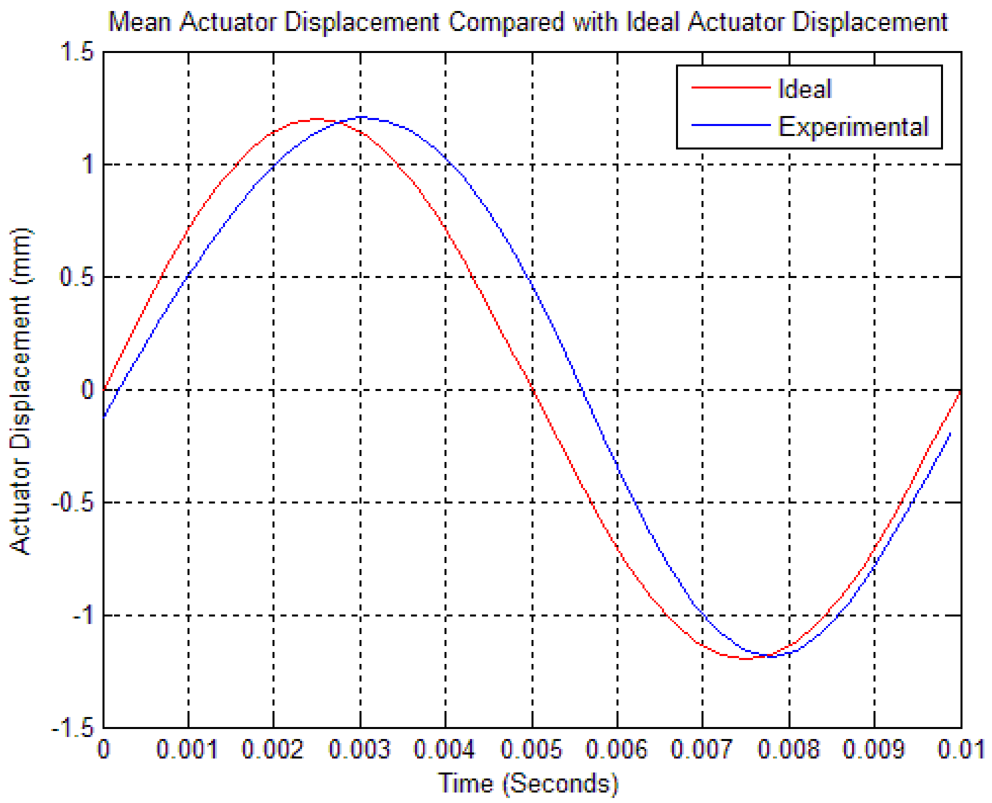


Figure 5.8(b): The mean actuator displacement compared with the ideal actuator displacement over 20,000 actuator cycles.

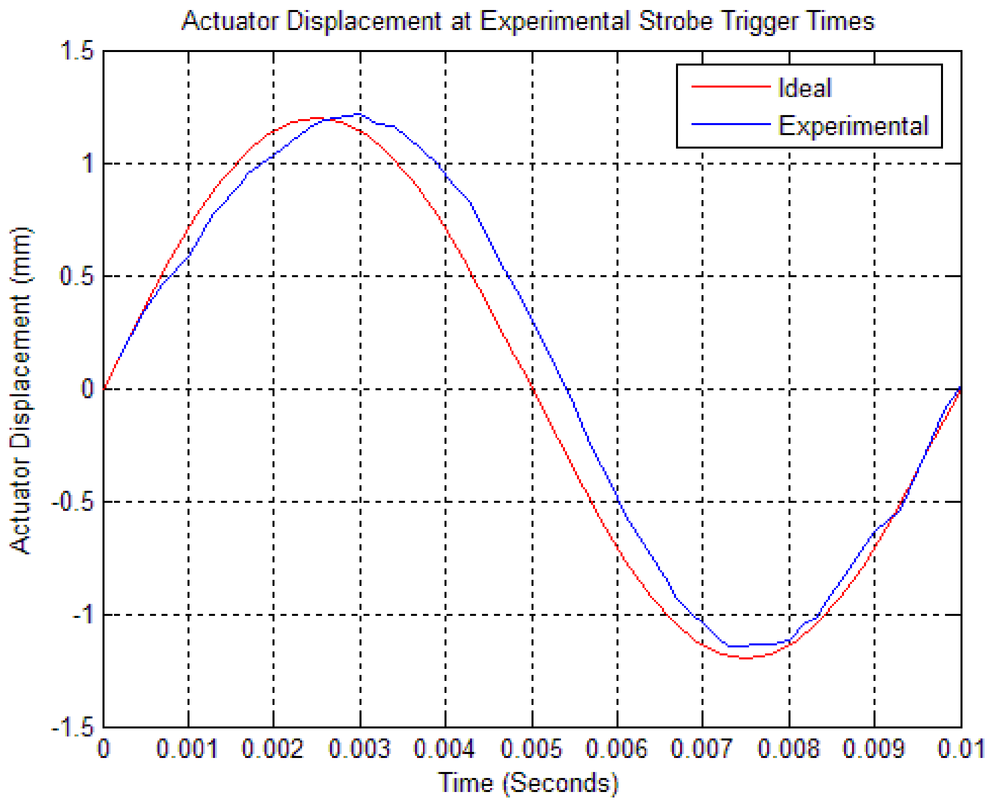


Figure 5.8(c): The compiled actuator displacements at the time of the strobe firing, compared with an ideal actuators displacement at that point.

In practice, images of a silicon phantom's displacement response to one specific period of the actuator, as in Figure 5.8(a) for example, cannot be achieved. As discussed in Chapter 2, the way to build up one complete cycle of the actuated silicon phantom, is to strobe at pre-defined phase angles and capture an image at each strobed point in time. To further validate the method the phase lag increment is chosen to be 10 degrees so that strobing occurs every 0.00028s of the 0.01s cycle.

The actuator displacements at the time of the strobe firing are plotted in Figure 5.8(c). Note that the specific times that the images are taken can vary significantly between runs. However, relative to the 0.01s cycle the image capture times are very consistent, as shown in Figure 5.7. For example, the first 10 image capture times in Figure 5.8(c) are:

t = [5.541, 14.5413, 24.3915, 34.1817, 44.0121, 53.8224, 63.6525, 73.4427, 83.2932, 93.0834]

With respect to the 0.01s cycle the 10 image capture times are effectively:

t = [0.0002, 0.0005, 0.0007, 0.001, 0.0013, 0.0016, 0.0017, 0.002, 0.0024, 0.0026]

These values are very close to multiples of 0.00028, as required. The results in Figure 5.8(c) show similar behaviour to Figures 5.8(a) and (b), further demonstrating the accuracy of the strobe and camera trigger system of Chapter 3.

5.4 Surface Motion Tracking

Six images of the silicon phantom corresponding to 60 degrees intervals from 0-300 degrees are shown in Figure 5.9. The frequency of actuation is 100Hz with amplitude of 1.2mm. On the face common to both cameras there are 54 black dots, which are used to help track the surface motion of the phantom.

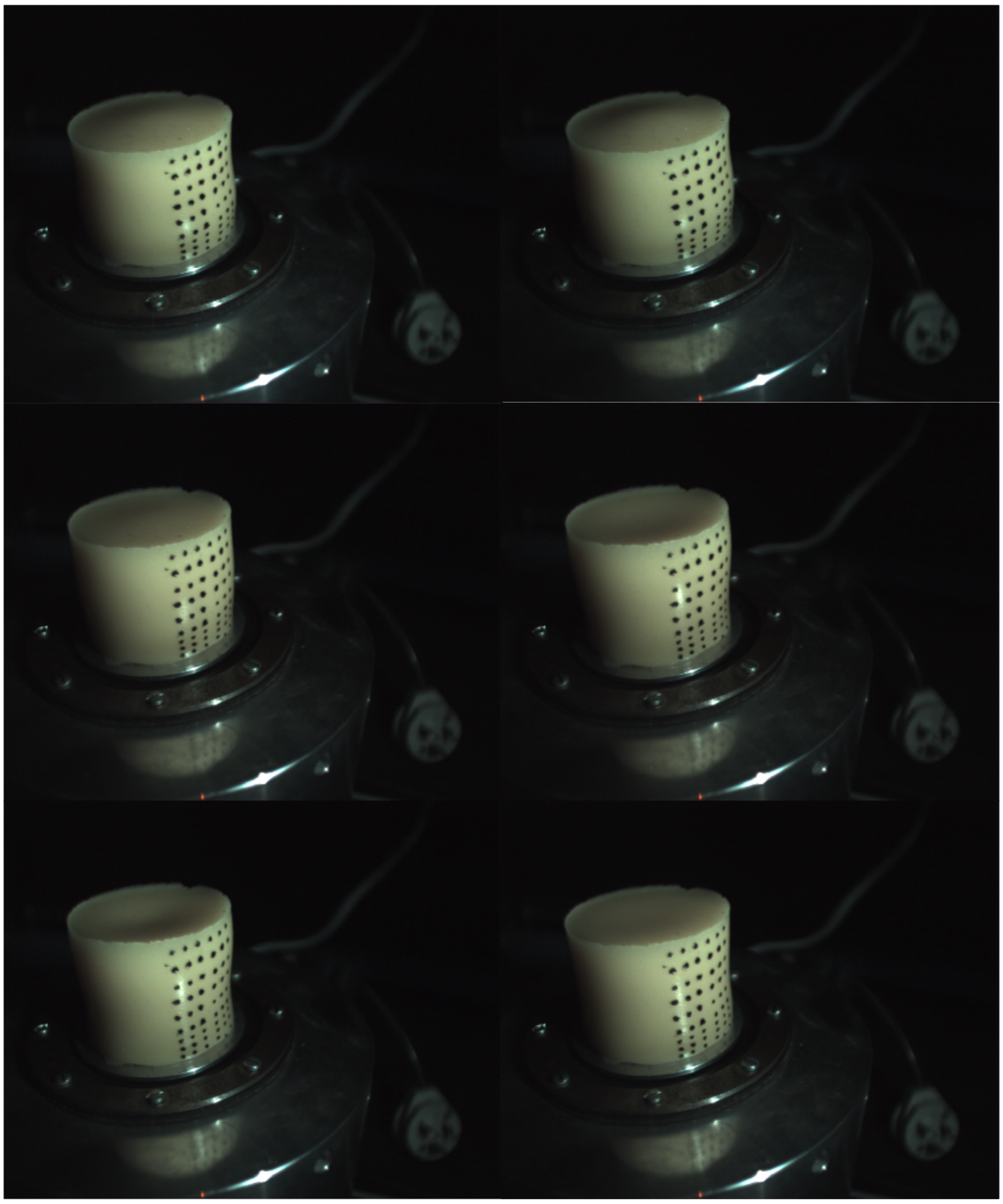


Figure 5.9: The silicon phantom with 54 black dots on the surface moving at 1.2 mm of amplitude starting at phase lag of 0 degrees and moving to a phase lag of 360 degrees from the left camera.

To validate the image capture system the black dots shown in Figure 5.9 are used to track the displacement of the moving surface of the phantom. The motion tracking is performed using software constructed by Richard Brown as part of his PhD thesis.

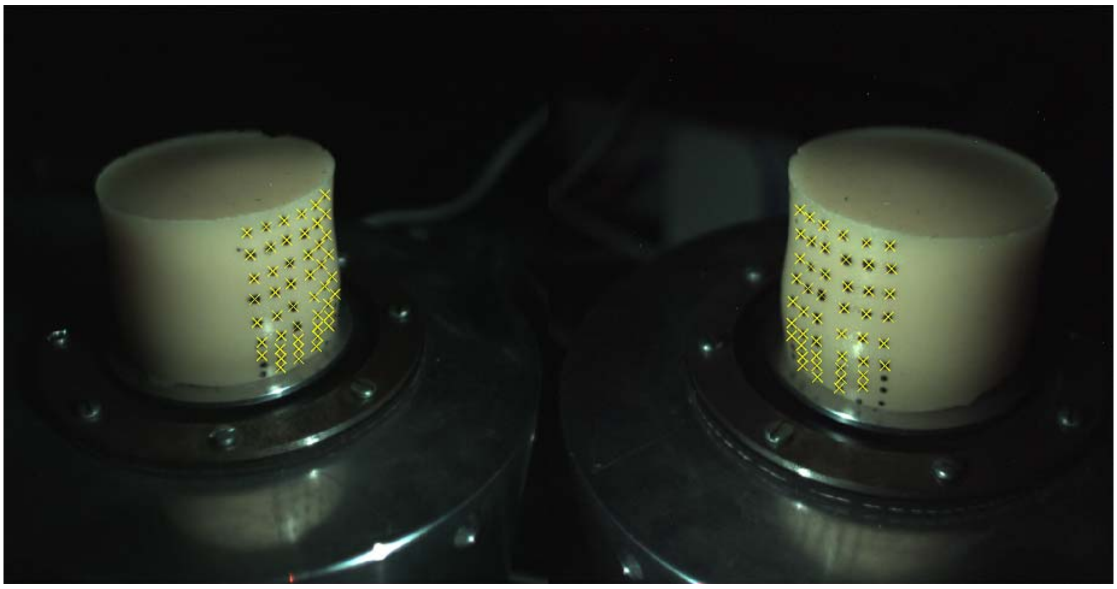


Figure 5.10: Tracking motion of the dots on the silicon phantoms surface, using images from the DIET image capture system. The identified dots are denoted by crosses.

Figure 5.10 shows the moving phantom and crosses overlaid by the image tracking algorithm. Figure 5.11 shows an example of the 3D tracked motion for one of the points in Figure 5.10. Note that a small number of the dots shown in Figure 5.10 on the surface of the silicon phantom could not be tracked because they move outside the field of view shared by both cameras.

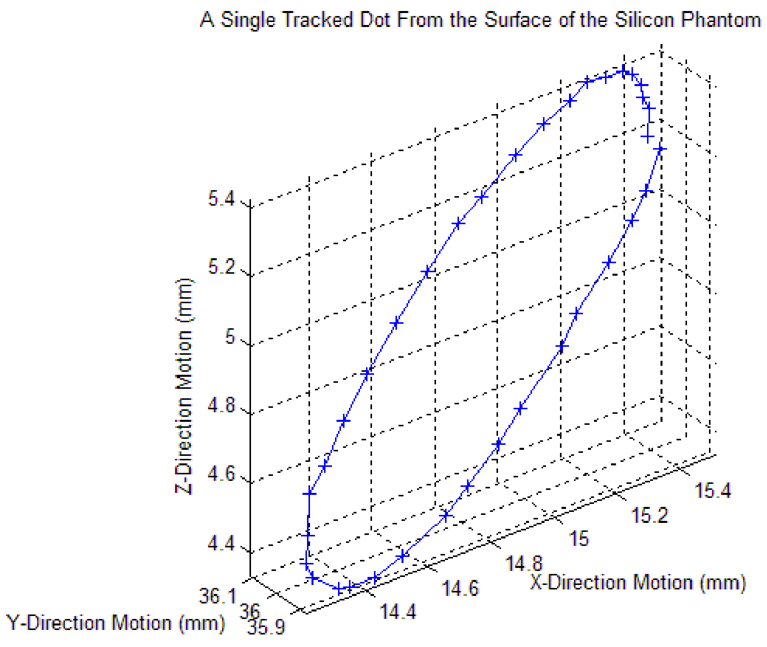


Figure 5.11: The motion of a single dot on the surface of the silicon phantom

An example of the reconstructed surface for one point in the actuators cycle is shown in Figure 5.12. The 3D mapped positions of the dots are then used to construct a virtual silicon phantom with the mapped dots displayed in red as shown in Figure 5.13. This 3D visualisation of the virtual phantom is generated by assuming that the silicon phantom is rotationally symmetric, and the final rendered phantom was created by rotating one column of dots about the central axis of the phantom and adding a flat top surface. The calculated 3D locations of the points were then overlaid on top of the phantom to create the final image. The software for generating Figure 5.13 was written by Richard Brown.

Note that Figure 5.13 is not an accurate 3D reconstruction of the whole surface rather a means of visually validating the results and demonstrating the application of the imaging system. The virtual silicon phantom visually agrees with the physical silicon phantom results shown in Figure 5.9 throughout the actuation cycle. This data further validates the successful operation of the camera system and, in particular, shows that images can be captured at high frequencies at a sufficient image quality that makes it possible to retrieve useful surface motion data.

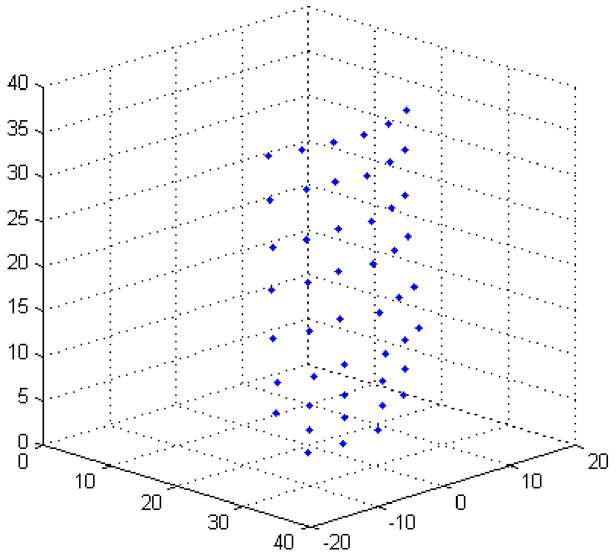


Figure 5.12: The 3D reconstruction of the tracked points from the phantoms surface

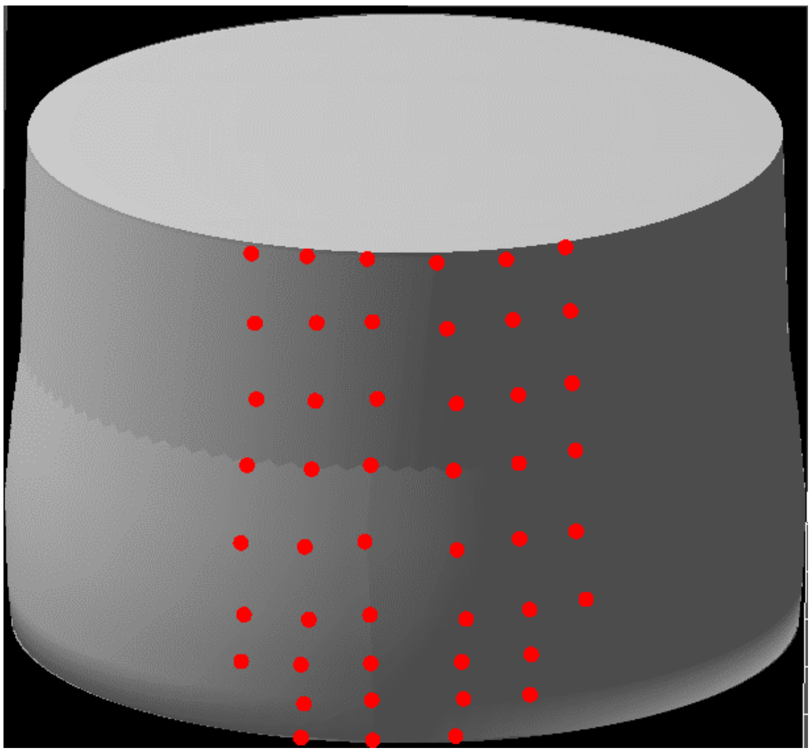


Figure 5.13: The virtual silicon phantom with the mapped points in red over the surface

5.5 Summary

Initial tests of the high-speed image capture system show that there were inconsistencies with the time period between the external sync pulse and the receiving of the snap trigger pulse from dSpace™. Furthermore, there were small inconsistencies between cameras as to when the external pulse was sent even though the cameras received the same snap trigger pulse at the same time. These two problems were addressed by gathering the external sync pulses from both cameras and feeding them into the dSpace™ module, via an AND gate. The AND gate allowed the inconsistencies of the external sync pulse between cameras to be combined into one coherent external pulse which was fed into dSpace™. This pulse was then used to trigger the strobe at the required time to obtain an image of the phantom at specified actuator position.

The resulting images were of a quality that allowed 3D motion tracking software to successfully track the surface motion of an actuated silicon phantom at a frequency of 100Hz and amplitude of 1.2 mm. However, analysis of the trigger time data and actuator position data showed that the actuator was not performing with as consistent a frequency as might be desired introducing some errors outside the control of this research.

PART IV

CONCLUSIONS

Chapter 6

Conclusion and Future Work

This chapter summarizes the tests undertaken on the image capture system, and discusses possible future work on the system. Future work is presented as possible directions for the improvement of the automated image capture system towards developing a final prototype.

6.0 Conclusion

A high-speed digital image capture system for Digital Image-based Elasto-Tomography (DIET) breast cancer screening has been presented. The final system satisfies the DIET system requirements of a completely automated relatively low cost method for capturing images of a silicon phantoms surface under sinusoid actuation at high frequencies up to 100 Hz. The image capture system was successfully tested on a silicon phantom moving at 100 Hz and amplitude of ~1.2 mm, providing accurate surface motion tracking at a high image resolution of 1280x1024. The image capture system also included functionality for the manipulation of colour gains and active windows making the system more adaptive to testing and laboratory conditions.

An important feature was the use of the dSpace™ control system module, which allowed the image capture process to take place outside of the Windows™ operating system message loops. This approach greatly increased the control over the timing of the events that go into capturing the high-speed images. It also more exactly matches any such commercial system, which would also use similar embedded operating system.

The construction of the digital cameras, in-house allowed a greater flexibility when it came to integrating them into the overall system. In addition, using Kodak's KAC-9648 CMOS imaging sensor, allows a reduction in complex circuitry in the camera design, simplifying making the future development and production of the digital cameras easier.

The comparison between the ideal and actual strobe triggering times, showed that the strobe was correctly triggered at the required predefined phase angles with a mean absolute error of ~1.4%. There were variations of the displacement of the actuator compared with the ideal actuator displacement at which the strobe triggers corresponding to a variation of 95-100 Hz within one image capture cycle. However, this displacement error was shown to be attributable to the dynamic properties of the actuator itself. For example, internal friction and the returning frequency of the LVDT signal varying slightly either side of the reference frequency for the introduction of the phase lag. A more exact next generation actuator will resolve these issues.

The time taken to complete an entire image capture run, of 37 images per camera took ~6 minutes upon review of the image capture log in the main application. This time can be reduced with the refining of the Ethernet protocols between the dSpace™ and image capture computer (ICC). The refining of the protocols could potentially reduce this image capture time by half (~3 minutes). Additionally, a custom designed system might reduce this test time by a further 2-10x.

6.1 Future Work

This section discusses possible directions for the continued development of the DIET image capture system towards achieving a robust and versatile image capture system. Possible next generation improvements for the image capture system are summarized as follows:

- The introduction of wider lenses with the ability to auto focus
- Replacement of the dSpace™ module with a self contained microcontroller
- Upgrade of the experimental apparatus and the introduction of ring flash devices for each camera
- Actuator development

6.1.0 Rebuild of the Camera Lens Arrangement

Currently, there is no zoom on the cameras. Thus, the object being imaged must be placed close to the lens, producing a fish eye effect that could potentially affect the accuracy of camera calibrations and motion sensing. The manual placement of the object could also produce varying fish eye effects between multiple cameras, complicating the identification of common points between images and introducing further sources of error. The solution would be to introduce a wider lens to the digital camera with auto focus properties. The auto focus could involve a motorized lens system that analyze the incoming images and adjust the focus to optimize high spatial frequencies in the stream data indicating a well-focused camera. Once calibration is achieved the focus could be locked by the user for the duration of the image capture process. This capability is commonly available a modern digital cameras and could likely be obtained “off the shelf” in future prototypes.

6.1.1 Replacing the dSpace™ module Concept

The dSpace™ module performs very well when tracking the actuators motion and triggering the camera snap and strobe. However, the Ethernet aspect of the system tends to slow the image capture process and is prone to corruption when insufficient bandwidth is made available to the process. One possible solution could be the introduction of one or more PSoC™ microcontrollers that would be communicated to, from the Image Capture Computer (ICC) over the same I^2C bus as the cameras as shown in Figure 6.1.

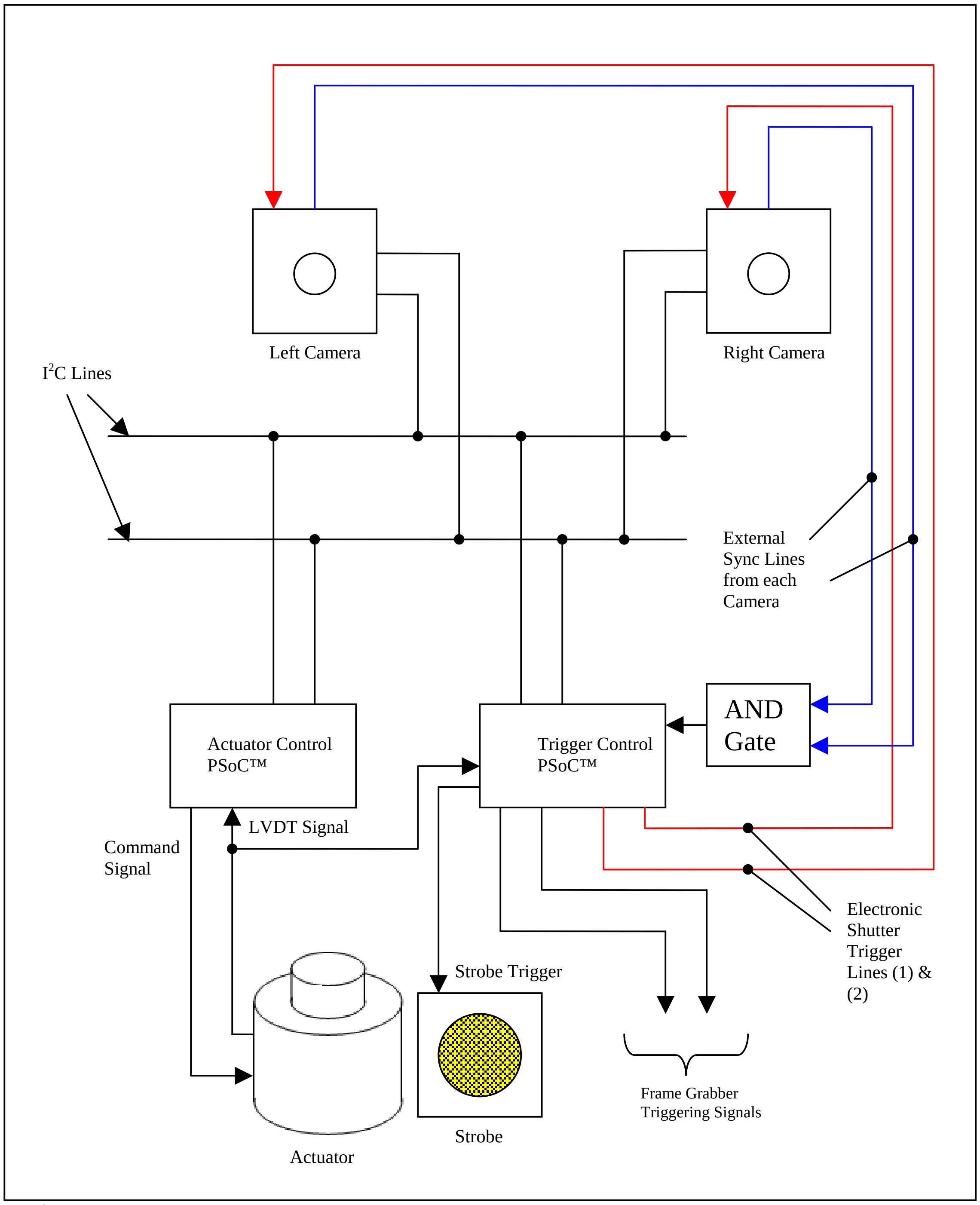


Figure 6.1: The set-up for the introduction of microcontrollers to the image capture system

The I^{2}C communications protocols are an industry standard and, as such, provides a robust and reasonably fast communication between the adapter and all devices on the bus. This will allow the ICC to more quickly and efficiently notify trigger control that the cameras and the frame grabbers are prepared for the triggers, and that the images have been successfully saved to allow the incrementing of the phase lag.

Furthermore, the Python™ code used to automate ControlDesk™ is a high level scripting, interpreted and interactive object-oriented programming language. It thus runs slower than the C++ software running on the ICC to which it is communicating. Hence, the C++ software needs to be slowed to effectively communicate with the dSpace™ computer. Finally, this approach would allow the whole image capture process to be driven from a single computer, rather than setting up two different computers. Again it is a next step towards a standard, commercial prototype for this type of mechatronic system.

6.1.2 Experimental Apparatus Concept

The upgrade of the experimental apparatus would see the introduction of four or five more cameras and the introduction of ring flash devices to each camera. The ring flashes would trigger from one single trigger and have all the cameras external syncs tied together. The ring flashes could be constructed from high intensity light emitting diodes, and positioned around the lens of each camera, as shown in Figure 6.2. This arrangement would provide and even distribution of light at high intensity. The current strobe flash, while providing adequate lighting for the capturing of the images, required the colour gains on the cameras to be increased with a completely opened lens iris.

Furthermore, the current strobe is bulky and restricts the placement of the cameras about the silicon phantom test piece. This change would thus significantly improve the quality of images captured.

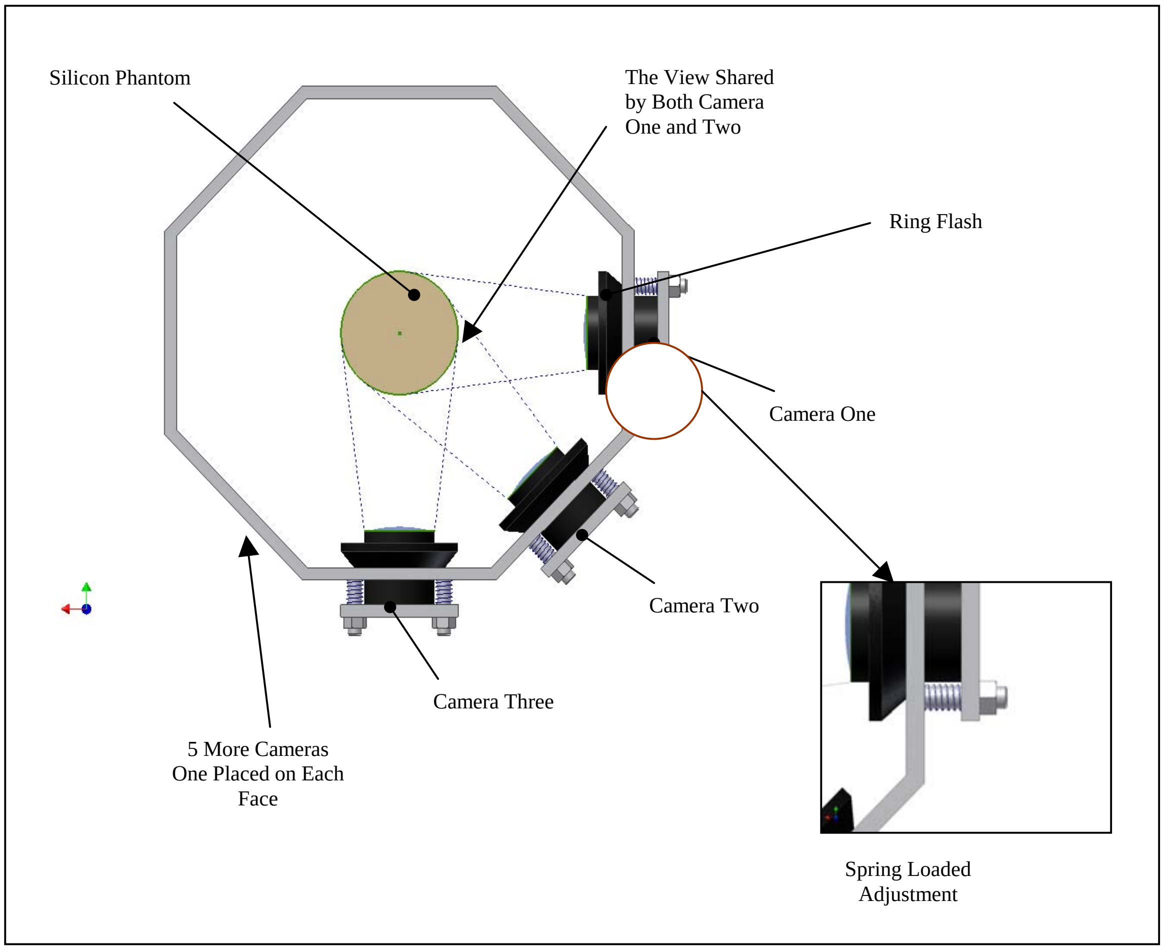


Figure 6.2: Possible new experimental set-up

The cameras could also be mounted via four spring loaded studs to allow the small movement of the cameras orientation, to set-up experimental tests. The number of cameras would depend on the angle of view of each camera to increase the size of the surface of the silicon phantom available to both cameras, and how many cameras it would take to capture the entire surface of the silicon phantom, in binocular vision. In summary, a more flexible and efficient prototype system to aid further development.

6.1.3 Actuator Development Concept

The current actuator performs well, within the design specifications. However, the actuator was difficult to control with a standard PID controller. It was found that the actuator also needed a period of time to "warm-up" before the system become stable. This behaviour was perhaps due to the expansion properties of the main nylon bearing, which appears to guide and align the actuator piston. Nylon bearings are not designed for high cyclic wear situations without regular lubrication. Furthermore, there did not seem to be any way of applying lubrication to the main bearing from the outside. In the report that accompanied the finished actuator, a sintered bronze bearing was suggested, but was not implemented due to high cost. A possible solution for the replacement of the current nylon bearing would be to remove the bearing all together and implement a second diaphragm, as shown in Figure 6.3.

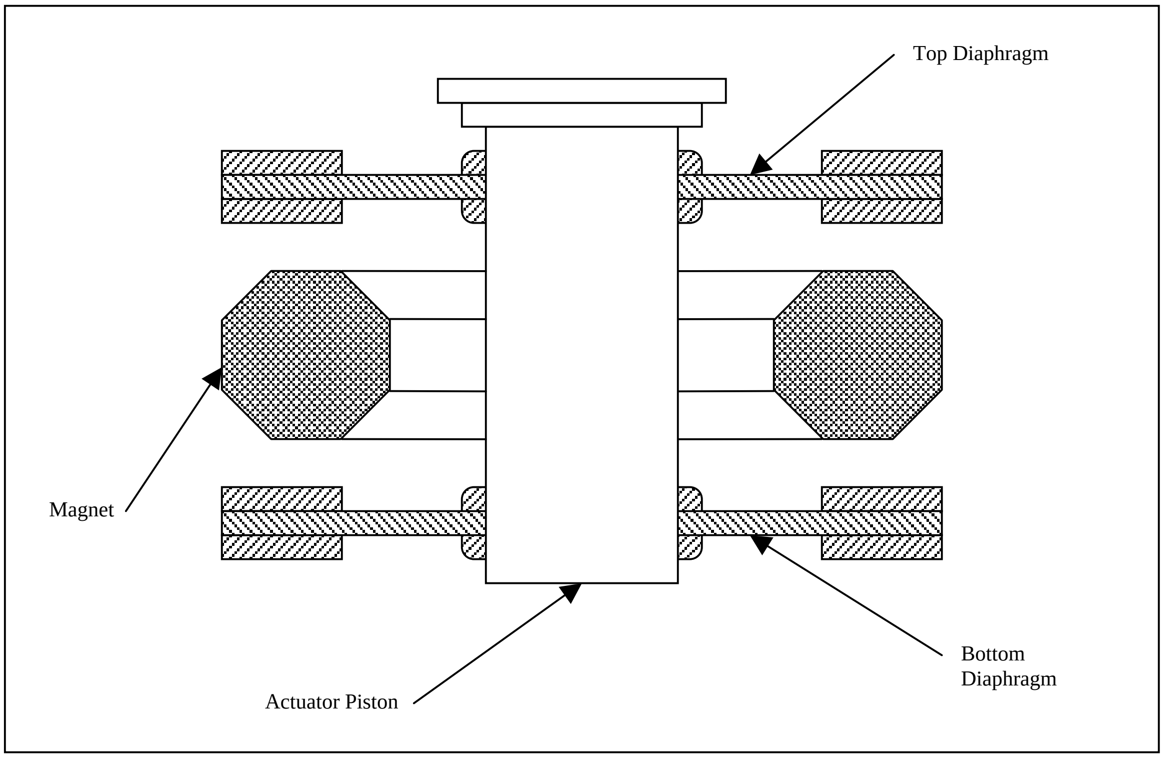


Figure 6.3: Possible solution for the removal of the main actuator bearing

The piston would thus be suspended between the two diaphragms one at the top and the other at the bottom with the magnetic field in the middle. This design would also do away with the need for lubrication. Furthermore, the heating that comes with moving the piston through the nylon bearing at high speeds does not allow the piston to return to the zero position easily, as the piston is perhaps being gripped by the warm nylon, due to the nylon expanding when heated.

References

[1] I. S. Fentiman, "*Breast Cancer in Older Women*", Breast Cancer Online, Vol. 5 2002

[2] D. B. Kopans, "*Breast Imaging*", 2nd Edition, Lippincott Williams & Wilkins, U.S.A, 1997

[3] Samani etal, "*Measuring the Elastic Modulus*", Phys. med. Biol, 48:2183-2198, 2003

[4] Kroustop etal, "*Elastic moduli of the Breast*", Ultrasonic Imaging, 20:260-274, 1998

[5] Wellman and Howe, "*Breast tissue Stiffness*", Harvard BioRobotics Laboratory Technical Report 2000

[6] S. Lauxtermann, P. Schwider, P. Setiz, H. Bloss, J. Ernst, H. Firla, *"A high speed CMOS imager acquiring 5000 frames/sec"*, IEEE, pp. 875-878 (1999)

[7] S. Kleinfelder, S. Lim, X. Liu, A. E. Gamal, *"A 10,000 Frames/s CMOS Digital Pixel Sensor"*, IEEE Journal of Solid-State Circuits, Vol. 36, No. 12, pp. 2049-2059 (December 2001)

[8] A. I. Krymski, N. E. Bock, N. Tu, D. V. Blerkom, E. R. Fossum, *"A High-Speed, 240-Frames/s, 4.1-Mpixel CMOS Sensor"*, IEEE Transactions on Electron Devices, Vol. 50, No. 1, pp. 130-135 (January 2003)

[9] B. Wilburn, N. Joshi, V. Vaish, M. Levoy, M. Horowitz, *"High-Speed Videography Using a Dense Camera Array"*, Presented at CVPR (2004)

[10] S. J. Kirkpatrick, D. D. Duncan, *"Acousto-optical Elastography"*, Proceedings of SPIE, Vol. 4257, pp. 426-432 (2001)

[11] Oida etal, *"Magnetic Resonance Elastography: in vivo measurements of elasticity for human tissue"*, International Conference in Informatics Research for Development of Knowledge Society Infrastructure, 2004

[12] Maurice etal, *"Non-Invasive Vascular Elastography: Theoretical Framework"*, IEE Transactions on medical Imaging, 23:164-180, 2004

[13] E.E.W. Van Houten, K.D. Paulsen, M.I. Miga, F.E. Kennedy and J.B. Weaver, "*An overlapping subzone technique for MR based elastic property reconstruction*", Magnetic Resonance in Medicine, 1999, 42(4), pp 779-786.
[14] E.E.W. Van Houten, J.B. Weaver, M.I. Miga, F.E. Kennedy and K.D. Paulsen, "*Elasticity reconstruction from experimental MR displacement data: Initial experience with an overlapping sub-zone finite element inversion process*", Medical Physics, 2000, 27(1), pp 101-107.

[15] A. Peters, A. Milsant, J. Rouze, L. Ray, J.G. Chase and E.E.W. Van Houten, *"Digitial Image-based Elasto-Tomography: Proof of Concept Studies for Surface Based Mechanical Property Reconstruction"*, Japanese Society of Mechanical Engineers (JSME) International Journal, Series C, Vol 47(4), pp 1117-1123, (2004).

[16] A. Peters, S. Wortmann, R. Elliott, M. Staiger, J.G. Chase and E.E.W. Van Houten, *"Digital Image-based Elasto-Tomography: First experiments in surface based mechanical property estimation of gelatine phantoms"*, Japanese Society of Mechanical Engineers (JSME) International Journal, Series C, Vol 48(4), pp 562-569, (2005).

[17] J. Fincher, A. Morrison, C. Murray, J. Steel, "*Diet System Actuator*", Final Year Project, (2005).

Figures

[Figure 1.1] A. Hii, "*Cluster Tracking Algorithms for a DIET System*", Masters Thesis, (2006)

[Figure 1.3 & 1.4] B. Wilburn, N. Joshi, V. Vaish, M. Levoy, M. Horowitz, "*High-Speed Videography Using a Dense Camera Array*", Presented at CVPR (2004)

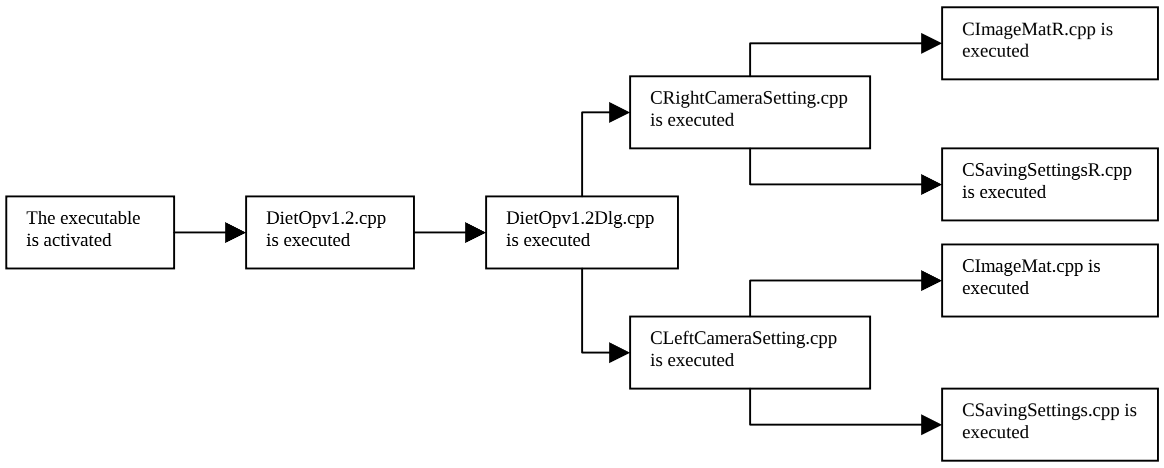
[Figure 2.1] Stefan Wortmann, "*Soft-Tissue Actuation and Imaging-based Motion Measurement System"*, Diploma-Thesis, (2006)

Appendix A

Visual C++ Code

A1: Detailed Visual C++ Code Explanation

What is outlined in appendix A is a descriptive explanation for the operation of the major parts of the code which operates on the ICC. References to blocks of code can be found after the text in sperate appendix blocks. Below in Figure A1.1 is the hierarchy for the dialog box activation. This gives an indication of what happens and when.



The executable is activated
DietOpv1.2.cpp is executed
DietOpv1.2Dlg.cpp is executed
CRightCameraSetting.cpp is executed
CLeftCameraSetting.cpp is executed
CImageMatR.cpp is executed
CSavingSettingsR.cpp is executed
CImageMat.cpp is executed
CSavingSettings.cpp is executed

Figure A1.1: Hierarchy for dialog box activation

A1.1 The Main Image Streaming

The main image streaming occurs upon the activation of the image capture program on the ICC, and streams video data from the frame grabbers to the main dialog box. A lot of the code that allows this to happen is repeated in other parts of the application. This section of the application uses OpenGL™ to scale the images and display them in the application. The image streaming is started in the initialisation of the main dialog code shown in Appendix A2. The lines in particular are (21) and (22) which start the functions that take the data from the frame grabbers and display it on the screen. However before this occurs, we must set-up windows, which are attached to the main dialog box, in which the image streaming functions can put the image data, this is done in line (19) of Appendix A2.

The process of streaming the images from the cameras to the screen can be assumed to be the same for both cameras, and so the explanation will always make reference to the left camera. Figure A1.2 shows a flow diagram for the operation of the image streaming process. The first operation to occur after the beginning of the function is the registering of the newly created window with the OpenGL™ processes. This occurs in Appendix A3 line (6) to (12). Then the frame grabbers are set-up using the functions that accompanied them in the frame grabber's driver software. Since there are two frame grabbers we must address the frame grabber to which the left camera is connected. In Appendix A3 line (13) this is done when setting up the interface session for the frame grabber, and the frame grabber being used is denoted in that line by "img1". The numbering of the frame grabbers begins at zero ("img0"), hence we are addressing the second frame grabber and consequently the left camera. What follows line (13) is just applying the settings for the size of the images the frame grabbers can expect to get from the CMOS imagers, and this continues until line (21). In line (30) we use the "grab" function which is used to always 'grab' new image data from the cameras and place it in the "LeftCamImaqBuffer".

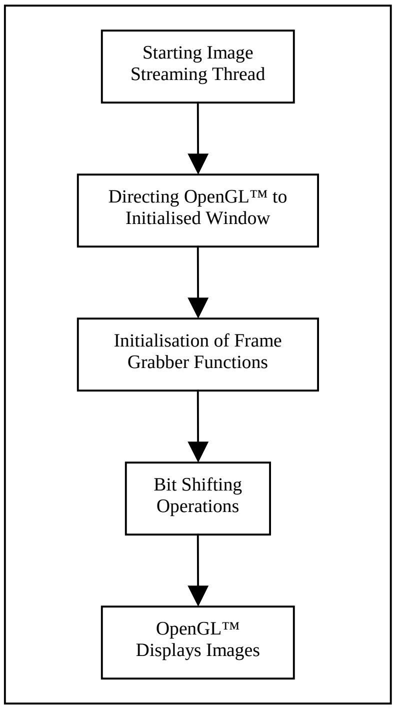

Starting Image Streaming Thread
Directing OpenGL™ to Initialised Window
Initialisation of Frame Grabber Functions
Bit Shifting Operations
OpenGL™ Displays Images

Figure A1.2: The operation of the image streaming function

At this point the image data is still at its original dimensions it arrives from the cameras at, and so it needs to be reduced in size in order to display the entire image data on the screen. For this we use OpenGL™ in particular OpenGL™'s function 'glPixelZoom' in line (50) of Appendix A3. By providing the function with a scale factor of 0.3 for the x and y dimensions, the image data being transferred to 8-bit is be reduce to a third of its original size, and displayed using the 'glDrawPixel' function in the main dialog box.

A1.2 The Apply New Camera Settings Event

An event is the triggering of an action by way of user interaction or an application process. Figure A1.3 shows the layout of the actions that occur upon the activation of the event, which applies new camera settings.

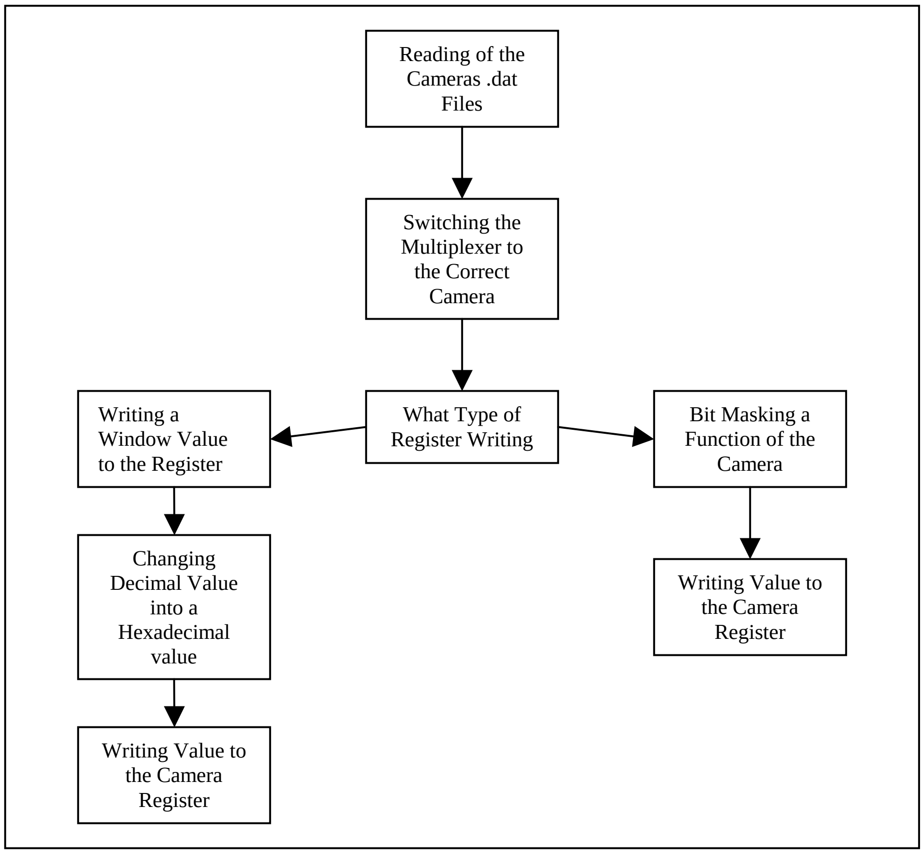

Reading of the Cameras .dat Files
Switching the Multiplexer to the Correct Camera
What Type of Register Writing
Writing a Window Value to the Register
Changing Decimal Value into a Hexadecimal value
Writing Value to the Camera Register
Bit Masking a Function of the Camera
Writing Value to the Camera Register

Figure A1.3: Layout of 'Applying New Camera Settings' to the Cameras

When applying new settings to the cameras, the data for the settings arrives from the .dat files created using the CImageMat class for each camera. The values entered into the dialog box that is ran using the CImageMat class, are the values for the camera window settings. The values are then saved to the LeftCameraSettings.dat file, using windows CArchive class. The code for retrieving these settings is outline for the left camera in Appendix A4. The values are read out in the order they were read into the .dat file, and are a read out into a local variable created by the programmer. The values read out of the .dat file are decimal values of base 10 that need to be converted into a hexadecimal value of base 16 format before they can be written to the cameras registers. All registers require the information to be in hexadecimal format.

Hexadecimal is shorthand for binary information. Each hexadecimal value represents 4-bits of binary data as shown in table A1.1.

Table A1.1: Binary Values and there respective Hexadecimal values

Binary	Hexadecimal
0000	0
0001	1
0010	2
0011	3
0100	4
0101	5
0110	6
0111	7
1000	8
1001	9
1010	A
1011	B
1100	C
1101	D
1110	E
1111	F

Hexadecimal can be used to represent 8-bits or one byte as ‘0x0E’ which has a decimal value of 14. The values of the window dimension to be written to the camera register are of base 10 and must be converted to base 16 in order for the camera to make sense of the data. The function built to carry out this conversion is outlined in Appendix A6. Since most of the values to be converted are larger than 255 they are represented in 16-bits or two bytes. When reading a hexadecimal value larger than 8-bits there are the ‘Most Significant Bits’ (MSB) and ‘Least Significant Bits’ (LSB) that determine the size of the resulting decimal value.

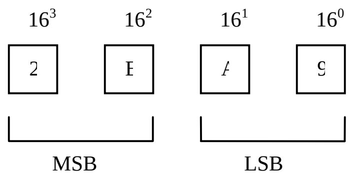


Figure A1.4: The Column Set-up for Base 16 Decimal Values

Normally we would see the hexadecimal value in Figure A1.4 as ‘0x2EA9’ but it is shown here to illustrate where the resulting decimal value comes from.

The value of the hexadecimal number in Figure A1.4 is:

$$\left(16^{3}\times 2\right)+\left(16^{2}\times 14\right)+\left(16^{1}\times 10\right)+\left(16^{0}\times 9\right)=11945$$

Once the camera window values have been converted to base 16 formats they are separated into their MSB and LSB components and sent to the I^{2}C interface code outline in Appendix A7 to be written to the camera registers. As well as windowing information being written to the cameras onboard factory settings can be changed, which includes the enabling of the external triggering pin on the CMOS imaging chip. This modification of factory settings is done using process call 'masking'. On the CMOS imaging chip itself each register is the size of 8-bits and has a unique address. Each register for the camera settings may contain more than one adjustable setting, and each setting is turned 'on' or 'off' by setting its respective value to either '1' or '0'. For example one register may resemble that in Figure A1.5, where some camera functions have been enable while others have not. If we only wish to affect one of the camera settings then we take the current value of the camera register and 'or' it with another byte, with a '1' in the position of the bit we wish to change.

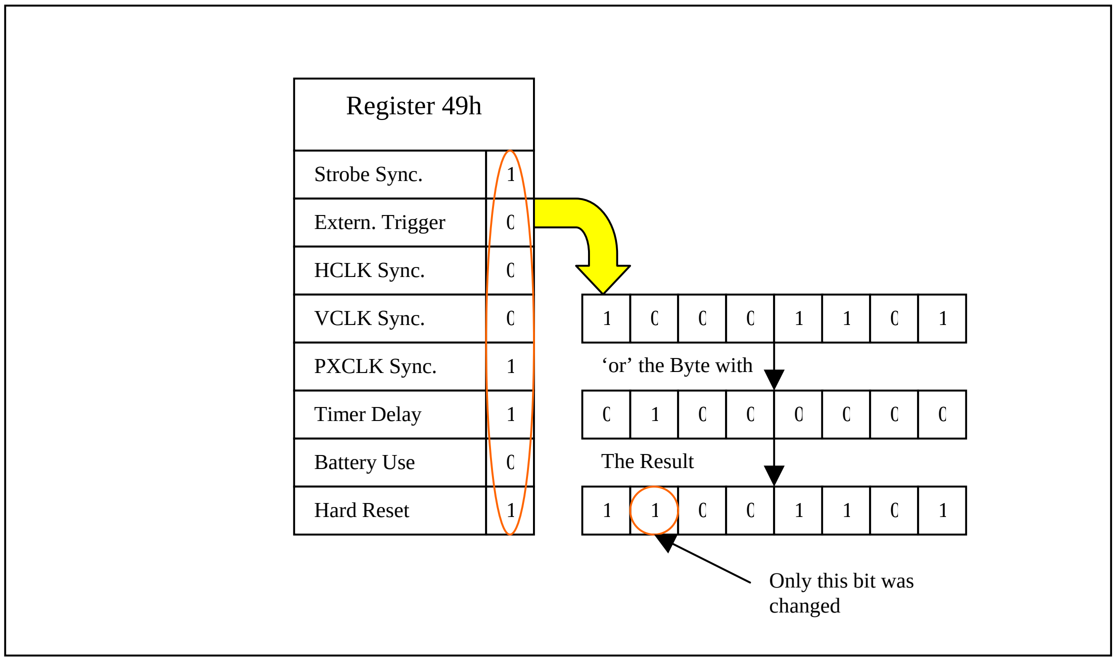


Figure A1.5: The operation of bit masking

Once it is known, which register the information is going, the I^{2}C interface code then communicates with the cameras to write the data there.

The I^{2}C interface code in Appendix A7 follows a specific format. The first part of the code (lines (5) to (33)) is for the bit mask and the second part (lines (36) to (53)) is for the writing of specific values. The code for actually writing the data to the cameras is the same over both sections.

The format for writing data to the camera registers is as follows:

- The 'WriteAddress' function is used with the 'KAC_9648_write_address' value sent. This is the address of the camera with the write-bit enabled. This occurs in line (8) and (36) of Appendix A7.
- Then the 'WriteData' function is used to tell the camera which register will be receiving the data. This occurs in line (37)

- The ‘WriteData’ function is used again to actually write the data to the camera register. This occurs in line (38)
- Then the ‘WriteStop’ function is used to halt communications with the CMOS image device. This occurs in line (39).

All of the functions described above use the adapter class functions. In Appendix A7 an object of the adapter class is taken and initialised, in line (3) and (4) respectively, before any communications occur. The I^{2}C interface code is built to check the data recently written to the specified register. This is done by the following:

- The ‘WriteAddress’ function is used with the ‘KAC_9648_write_address’ value sent. This is the address of the camera with the write-bit enabled. This occurs in line (40) of Appendix A7.
- The ‘WriteData’ function is used to tell the CMOS imaging device which register the data will be read from. This occurs in line (41).
- The ‘Restart’ function is used in line (42), with the ‘KAC_9648_read_address’ sent to the CMOS imaging sensor. The ‘Restart’ function is use to restart communications with the device, with the register of interest already been pre-selected for reading.
- The ‘ReadData’ function is used twice in lines (43) and (44). The reason for this is because the first ‘ReadData’, reads the data from the buffer on the I^{2}C communications adapter card in the computer, which just contains the recent register address. The second time the ‘ReadData’ function is called it grabs the data off the CMOS imaging device, from the specified register.

- The second call to 'ReadData' stores the contents of the register as 'Register_Value3' in line (44).
- Then the 'WriteStop' function is used to halt communications with the CMOS image device. This occurs in line (45).

Once the value of the current register has been obtained it is compared with the value that was intended to be written to the register to confirm the writing of the data has taken place. If the write was successful the I^2C interface code returns a value of '1', otherwise a value of '2' is return, indicating there has been an error in data writing to that register. This value is then used to prompt the user in the main dialogs, log window that an error has occurred with the camera registers.

A1.3 Start Image Capture Process

The automatic capturing of the triggered images occurs in this process. The function used to carry out this task uses the CSocket class, explained in chapter 3 to communicate between itself and the dSpace computer.

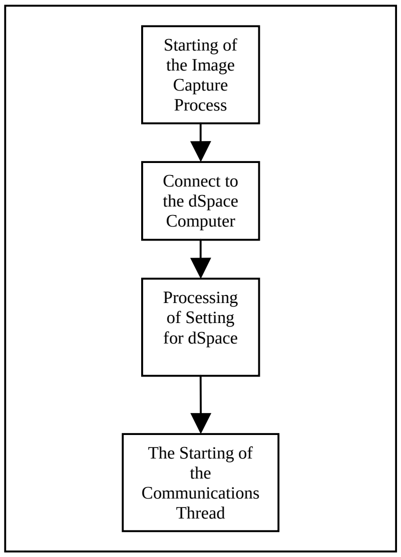

Starting of the Image Capture Process
Connect to the dSpace Computer
Processing of Setting for dSpace
The Starting of the Communications Thread

Figure A1.6: The Initial Stages of Running the Image Capture Function

Figure A1.6 outlines the initial stages of starting the image capture process. The connection to the dSpace computer is carried out with the use of the DietOpSocket class; the connection is made using the port number ‘2345’ as in Appendix A9 line (3). This port number has to be the same on the dSpace computer for the connection to be made.

The processing of the settings for dSpace occurs in Appendix A9. It starts with the updating of the data in the ‘dSpace Settings Field’, which occurs in line (13). The updating makes sure that the information for the settings is arriving directly from the main GUI. The data arriving from these fields are strings and are packaged up to be send to the dSpace computer in line (16) as ‘PhaseValues’. The spaces included in the combined string to be sent to dSpace are there to help the dSpace computer sort the data into its necessary places. The dSpace computer uses the ‘split’ function to break up the sent string.

A1.3.1 The Communications Thread

All communications carried out on the ICC are done inside a thread so that other functions, like the main image streaming, can continue to run within the main dialog code. The communications between the ICC and dSpace™ computer occur by sending message strings back and forth between the two computers. Once the communications thread begins, it starts a ‘while’ loop that loops continually until it is told to end by the dSpace computer. The dSpace computer is the driver of the image capture process, in that the ICC only acts when instructed by the dSpace computer and not before.

The ‘while’ loop that comprises the main part of the communications thread contains many different ‘if’ statements that respond to a given message string. The structure of the communication threads ‘while’ loop is outlined in Figure A1.7.

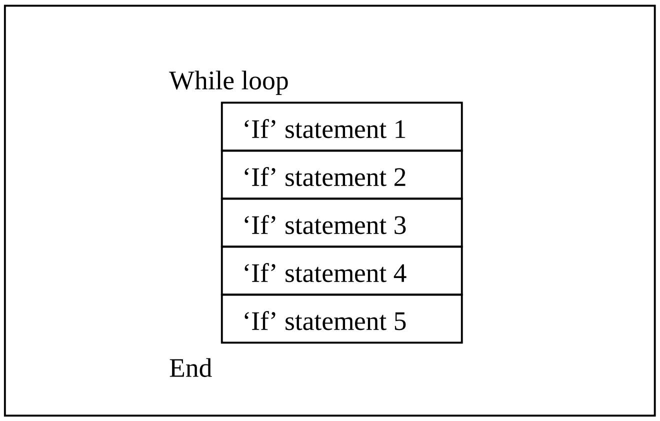


Figure A1.7: The layout of the communications thread ‘while’ loop

When the ‘while’ loop is running the ‘OnReceive’ event, which is part of the DietOpSocket class is constantly listening for new messages which are passed to the ‘while’ loop in Figure A1.7 during its operation from the ICC. The function of each ‘if’ statement shown in Figure A1.7is:

- <u>‘If’ Statement 1:</u> This statement is activated when the ‘GetPhases’ string is sent by the dSpace computer and received by the ICC. When asked for the settings for the dSpace computer, the ICC will send them repeatedly until the dSpace computer acknowledges there arrival.
- <u>‘If” Statement 2:</u> This statement is activated when the ‘SetCameras’ string is sent by the dSpace computer and received by the ICC. The statement then starts the frame grabbers waiting on the trigger signals (Appendix A10 line (37) and (38)). The cameras themselves begin waiting for the trigger signals once the external pin is enabled during the configuring of the cameras registers. The statement then sends a ‘CamerasReady’ string to the dSpace computer, so dSpace can send the trigger signal.

- <u>‘If’ Statement 3:</u> This statement is activated upon the successful triggering and saving of an image from the cameras. The statement then sends a message to the dSpace computer telling the computer that the images have been stored and to continue on with the image capture process.
- <u>‘If’ Statement 4:</u> This statement is activated when the capture of the image has not been successful. It then posts a message in the log window in the main dialog box. It does not alert the dSpace computer to the error, in order to maintain the image capture process as a whole.
- <u>‘If’ Statement 5:</u> This statement is activated when the ‘ResetCameras’ string is sent by the dSpace computer and received by the ICC. It is used to reset all ‘if’ statements in the ‘while’ loop in order to capture another image. Once the ICC computer has reset all its statements a string is sent to the dSpace computer tell it to reset its ‘if’ statements as well.

The reason why all the ‘if’ statement must undergo a reset is because, as well as acknowledging the message string, instructions sent from the dSpace computer, the ‘if’ statement also contains another condition that will only allow the contents of the ‘if’ statement to run if that condition is ‘TRUE’. Once inside the contents of the statement, that same condition is set to ‘FALSE’. This stops the same ‘if’ statement from being activated twice, during the running of the ‘while’ loop.

The trigger functions, which are started in line (37) and (38) of appendix A10, are expanded to the function shown in Appendix A11. As is can be seen there in Appendix A11 the first part of the trigger setting is the same over all the frame grabber functions. The important line is (30) as this puts the frame grabber in the mode to receive the trigger signal from the dSpace computer. Once the trigger has been sent the image data is read directly from the frame grabbers local memory which occurs in line (34) which then feeds the image to the ‘CopyBufferTriggerLeft’ image buffer which is then saved in line (42) to a folder specified by the user, and is denoted in this line as ‘m_filename’. This path is defined in the CSavingSettings dialog box.

Throughout the ‘while’ loop there are smaller ‘while’ loops which aid in the sending of the message string between the two computers. This is achieved by the loop continuing to send the same instruction message over and over again until or ‘while’ there is no acknowledgement of the messages arrival at the other computer. As soon as the instruction message makes it to the other computer that computer sends an acknowledgement message until the first computer stops sending the instruction message. This concept is outlined in figure A1.8.

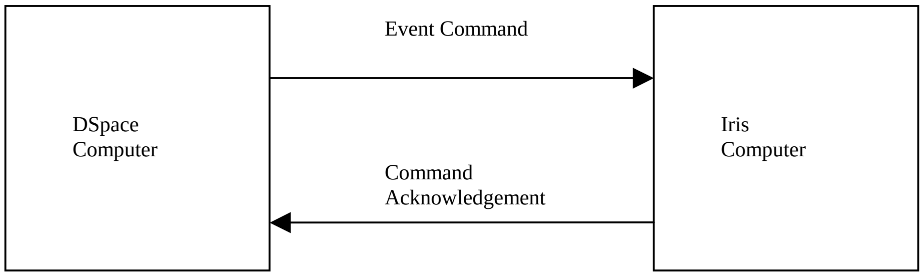


Figure A1.8: The string message sending layout

Once all the communications between the two computers has been completed the socket created on both computers are closed. The ICC lets the dSpace computer close its sockets first.

A1.4 Saving Information Log Process

The information log is a method of making the user aware of what is occurring within the workings of the image capture process. The messages are posted using the ‘Spit Message’ function shown in Appendix A8. This function takes message strings and posts them in the log window box shown in Chapter 3 Figure 3.6. The messages are time stamped.

The box to which the messages are posted to in Figure 3.6 is referred to as a ‘ListBox’. Once the ‘ListBox’ is drawn in the GUI a variable is attached to the ‘ListBox’. Attaching a variable to the ‘ListBox’ allows the programmer to control how it operates. In Appendix A8 the ‘Spit Message’ functions uses the ‘m_list’ variable to post the strings to the ‘ListBox’. Before the message is posted it is formatted using the CString class. This occurs in line (6) of Appendix A8. A snap shot of the system time is taken in line (4) and placed at the start of the string. Most strings sent to the ‘Spit Message’ function are loaded from the string table in the compliers resource folder. Each string has an I.D which is loaded into an intermediate string before it is passed to the ‘Spit Message’ function.

Upon pressing the ‘Save Log’ button in the GUI shown in figure 3.6, each line in the ‘ListBox’ is read into a rich edit file that is saved under a file name specified by the user. The log can then be cleared by clicking the ‘Clear Log’ button on the GUI shown in figure 4.3.

A1.5 The CLeftCameraSetting Class

The CLeftCameraSetting class is a dialog class that controls the workings of the ‘tab’ shift between the CImageMat dialog and the CSaveSettings dialog. The ‘tab’ shift is the selecting of either the CImageMat dialog or the CSaveSettings dialog, within the CleftCameraSetting dialog. The CLeftCameraSetting dialog is shown in Figure A1.9, at the top are two tabs, tab1 and tab2. Clicking on either tab changes the displayed dialog in the space below the tabs.

The displaying of CImageMat dialog and the CSaveSettings dialog is carried out by first taking an object of both classes, and placing them into a dialog array. Within the dialog array each dialog object can be accessed from the array by specifying the corresponding array number. The selection of each tab corresponds to the requested dialog object in the dialog array. The requested dialog is then displayed in the lower part of the CleftCameraSetting dialog using the 'SetWindowPos' with the '`SWP_SHOWWINDOW`' flag included in the function definition.

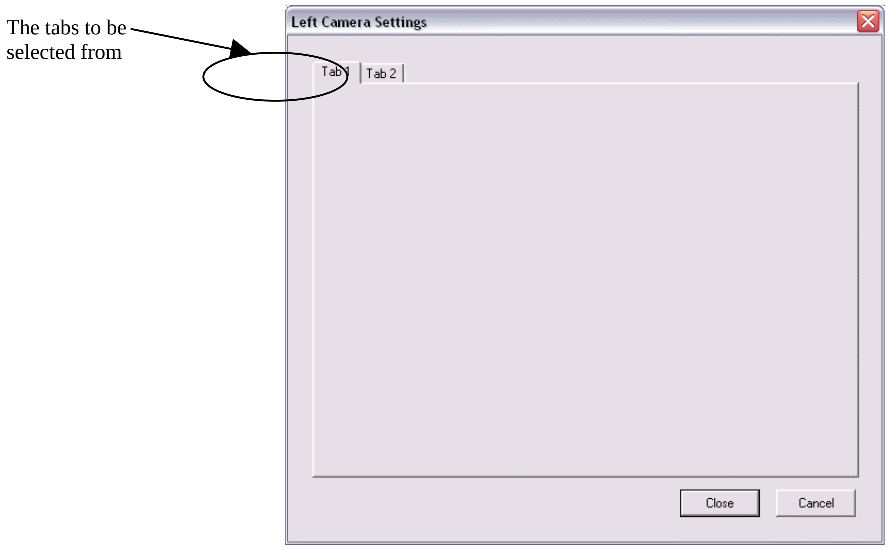


Figure A1.9: The dialog layout of the CleftCameraSetting dialog

The dialog displayed in the CLeftCameraSetting dialog then behaves exactly like it would if it where a stand alone dialog.

A1.6 The CSavingSettings Class

The CSavingSettings class operates the CSavingSettings dialog box which is responsible for the obtaining of the file paths, for saving the images, from the user. The dialog box shown in Figure A1.10 that is tied to the CSavingSettings class, uses Combo Boxes to allow the user to select from image formats and previously chosen file paths. Each ComboBox corresponds to a variable that will be archived to a .dat file upon the pressing of 'Accept' button in the dialog box of Figure A1.10.

The pressing of the 'Browse' button in Figure A1.10 starts the function in Appendix A13 and presents the user with the standard browser dialog box. Once the path is chosen by the user, the path is cast to a 'char' and saved as 'LeftCamCalPath' in line (22) of Appendix A13. The reason for casting the path to a 'char' was to make the file path easier to integrate with the frame grabber driver software functions. The lines (3) to (16) in Appendix A13 is the defining of the of browse dialog box.

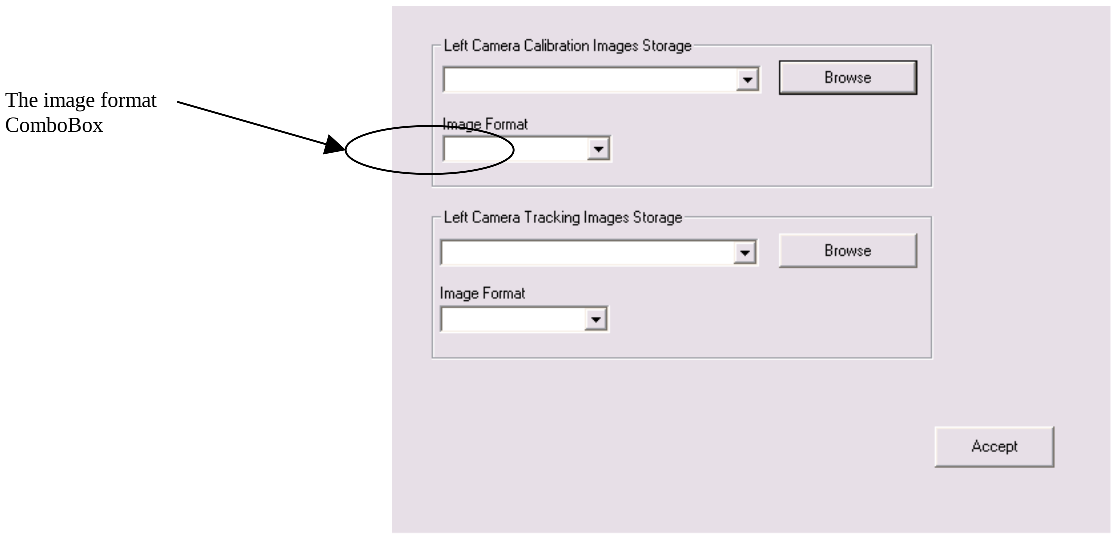

The image format ComboBox
Left Camera Calibration Images Storage
Browse
Image Format
Left Camera Tracking Images Storage
Browse
Image Format
Accept

Figure A1.10: The layout of the CSavingSettings dialog box

The CSavingSettings class, which runs the CSavingSettings dialog box, makes use of the combo boxes ‘OnCbnSelchange’ event or ‘on selection change’. An example of the ‘OnCbnSelchange’ event is shown in Appendix A14. This event allows the dialog box to update the variables for that particular ComboBox to the variable chose by the user from the ComboBox’s drop-down list. The drop-down list for the ‘Image Format’ variables are loaded in upon the initialisation of the CSavingSettings dialog box, and indexed to keep track of the selected ‘Image Format’. The image path combo boxes have their lists updated upon the selection of a path from the browse dialog box and the ‘OnCbnSelchange’ event for these combo boxes overwrites the path previously chosen by the browse dialog box with the newly selected path from the ComboBox’s drop-down list.

A2: Main Dialog Initialisation Code

```
BOOL CDIETOPv12Dlg::OnInitDialog()
{
        CDialog::OnInitDialog();

// Add "About..." menu item to system menu.

// IDM_ABOUTBOX must be in the system command range.
        ASSERT((IDM_ABOUTBOX & 0xFFF0) == IDM_ABOUTBOX);
        ASSERT(IDM_ABOUTBOX < 0xF000);

        CMenu* pSysMenu = GetSystemMenu(FALSE);
        if (pSysMenu != NULL)
        {
                CString strAboutMenu;
                strAboutMenu.LoadString(IDS_ABOUTBOX);
                if (!strAboutMenu.IsEmpty())
                {
                        pSysMenu->AppendMenu(MF_SEPARATOR);
                pSysMenu->AppendMenu(MF_STRING, IDM_ABOUTBOX, strAboutMenu);
                }
        }

// Set the icon for this dialog.  The framework does this automatically
//  when the application's main window is not a dialog
        SetIcon(m_hIcon, TRUE);                         // Set big icon
        SetIcon(m_hIcon, FALSE);                // Set small icon
        CamWindowInitialistion();
        CamStreamImageScale = 0.3;
        AfxBeginThread(StartUpLeftCamStream,(LPVOID)this);
        AfxBeginThread(StartUpRightCamStream,(LPVOID)this);
        ServerStatus = 1;
        CalledArray[0] = FALSE;
        CalledArray[1] = FALSE;
        CalledArray[2] = FALSE;
        CalledArray[3] = FALSE;
        CalledArray[4] = FALSE;
        ExecutedArray[0] = FALSE;
        ExecutedArray[1] = FALSE;
        ExecutedArray[2] = FALSE;
        ExecutedArray[3] = FALSE;
        ExecutedArray[4] = FALSE;

        Ethernet_Mess.Format("Initializing CDietOpSocket Class.......");

        spit_Message(Ethernet_Mess);

        if (!AfxSocketInit())
        {
                Ethernet_Mess.Format("Initializing Failed");
                spit_Message(Ethernet_Mess);
                CalledArray[1]=TRUE;
        }
        else
        {
                Ethernet_Mess.Format("Done");
                spit_Message(Ethernet_Mess);
        }

        return TRUE;  // return TRUE  unless you set the focus to a control
}
```

A3: The Main Image Streaming Code (Left Camera)

```
void CDIETOPv12Dlg::LeftCamStream()
{
        CDIETOPv12Dlg* LeftCamProp = this;
        int error, acqWinWidthLeft, acqWinHeightLeft;
        unsigned int   bufSize, bytesPerPixel;
        HWND hLeftCam;
        hLeftCam = LeftCamWindow->GetSafeHwnd();
        HGLRC hgLeftCam;
        HDC hdLeftCam = ::GetDC(hLeftCam);
        MySetPixelFormat(hdLeftCam);
        hgLeftCam = wglCreateContext(hdLeftCam);
        wglMakeCurrent(hdLeftCam, hgLeftCam);
```

Directing OpenGL™ to Initialised Window

```
        errChk(imgInterfaceOpen ("img1", &LeftCamInterfaceID));
        errChk(imgSessionOpen (LeftCamInterfaceID, &LeftCamSessionID));
        errChk(imgGetAttribute (LeftCamSessionID, IMG_ATTR_ROI_WIDTH,
                         &acqWinWidthLeft));
        errChk(imgGetAttribute (LeftCamSessionID, IMG_ATTR_ROI_HEIGHT,
                         &acqWinHeightLeft));

        errChk(imgSetAttribute (LeftCamSessionID, IMG_ATTR_ROI_WIDTH,
                         acqWinWidthLeft));
        errChk(imgSetAttribute (LeftCamSessionID, IMG_ATTR_ROI_HEIGHT,
                         acqWinHeightLeft));
        errChk(imgSetAttribute (LeftCamSessionID, IMG_ATTR_ROWPIXELS,
                         acqWinWidthLeft));
        errChk(imgGetAttribute (LeftCamSessionID, IMG_ATTR_BYTESPERPIXEL,
                         &bytesPerPixel));
        bufSize = acqWinWidthLeft * acqWinHeightLeft * bytesPerPixel;
```

Applying settings to the frame grabber capturing the left images

```
        errChkLeft(imgCalculateBayerColorLUT(redGainLeft, greenGainLeft,
                         blueGainLeft, LeftredLUT, LeftgreenLUT, LeftblueLUT,
                         bitsPerPixelLeft));
        errChkLeft(imgCreateBuffer(LeftMainVideoCamSessionID, FALSE, bufSizeLeft,
                         &RGBBufferLeft));
        BYTE* InterBufferLeft = new BYTE[bufSizeLeft];
        BYTE* FinalBufferLeft = new BYTE[bufSizeLeft];
        int i, Counter3, Counter37;
        for(i=0; i<NUM_RING_BUFFERS; i++)
                ImaqBuffersLeft[i] = NULL;
        // Setup and launch the ring acquisition
        errChkLeft(imgRingSetup(LeftMainVideoCamSessionID,
                         NUM_RING_BUFFERS, (void**)ImaqBuffersLeft, 0, TRUE));
        while(VideoLoopControlLeft==TRUE){
                if(::WaitForSingleObject(mythingLeft->m_EndVideoLeftCamera,0)
                                                             ==WAIT_OBJECT_0)
                {
                        VideoLoopControlLeft = FALSE;
                        ResetEvent(mythingLeft->m_EndVideoLeftCamera);
                }
                errChkLeft(imgSessionExamineBuffer (LeftMainVideoCamSessionID,
                         BufNumLeft, &currBufNumLeft, &bufAddrLeft));
                errChkLeft(imgBayerColorDecode(RGBBufferLeft, (void *)bufAddrLeft,
                         acqWinHeightLeft, acqWinWidthLeft, acqWinWidthLeft,
                         acqWinWidthLeft, LeftredLUT, LeftgreenLUT,
                         LeftblueLUT, IMG_BAYER_PATTERN_GRGR_BGBG,
                         bitsPerPixelLeft, 0));
                Counter3 = 0;
                Counter37 = bufSizeLeft;
                glClear (GL_COLOR_BUFFER_BIT);
                glColor3f (0.0,0.0,0.0);
                glMatrixMode(GL_PROJECTION);
                glViewport(0,0,0,0);
                glLoadIdentity();
                glRasterPos2i(0,0);
                glPixelZoom(CamStreamImageScale, CamStreamImageScale);
                glDrawPixels(acqWinWidthLeft,acqWinHeightLeft,GL_BGRA,
                         GL_UNSIGNED_BYTE, RGBBufferLeft);
                glFlush();
                SwapBuffers(hdLeftCam);
                errChkLeft(imgSessionReleaseBuffer (LeftMainVideoCamSessionID));
                BufNumLeft ++;

            }
            InterBufferLeft = NULL;
            FinalBufferLeft = NULL;
            delete InterBufferLeft;
            delete FinalBufferLeft;
            wglMakeCurrent(NULL, NULL);
            ::ReleaseDC (hLeftCam, hdLeftCam);
            wglDeleteContext(hgLeftCam);
    Error :
        if(error<0)
                DisplayIMAQError(error,LeftCamProp);

            // dispose of the buffer
            if (LeftCamImaqBuffer != NULL)
                imgDisposeBuffer(LeftCamImaqBuffer);

            // Close the interface and the session
        if(LeftCamSessionID != 0)
                imgClose (LeftCamSessionID, TRUE);
        if(LeftCamInterfaceID != 0)
                imgClose (LeftCamInterfaceID, TRUE);
    }
```

A4: Applying Camera Settings Code (Left Camera)

```
    void CDIETOPv12Dlg::OnBnClickedApplyCameraSettings()
    {
            Adapter Initialise_Adapter;
            int return_value;
            int return_value2;
            int RegWriteStatus;

            CFile LeftCameraFrameSettings;
            if(LeftCameraFrameSettings.Open("LeftCamSettings.dat",CFile::modeRead)
                                    ==FALSE){
                    AfxMessageBox("There is no LeftCamSettings.dat file
                            available",MB_OK);
                    return;};
            CArchive ar3(&LeftCameraFrameSettings,CArchive::load);
            ar3 >> LeftCam_woi_row_pointer >> LeftCam_woi_column_pointer >>
                    LeftCam_woi_row_depth >> LeftCam_woi_column_width >>
                    LeftCam_vf_row_depth >> LeftCam_vf_column_width;
            ar3.Close();
            LeftCameraFrameSettings.Close();
```

Retrieving Camera Window Information

A5 (a): Bit Masking For Camera Factory Setting (Left Camera)

```
            RegWriteStatus = CMOS_Register_Writer(BIT_MASKING,
                    KAC9648_FRAME_MODE_SINGLE_FRAME_MASK, 0x40);
            if(RegWriteStatus==1)
            {
                    RegMessage.LoadString(IDS_LEFT_CAM_SINGLE_SHUTTER);
                    spit_Message(RegMessage);
            }
            else if(RegWriteStatus==2)
            {
                    RegMessage.LoadString(IDS_NO_WRITE_LEFT_CAM_SINGLE_SHUTTER);
                    spit_Message(RegMessage);
            }
```

A5 (b): Writing Value to Camera (Left Camera)

```
IntegerToHex(LeftCam_woi_column_pointer);
TransferLSB = Bit_Array[0];
TransferMSB = Bit_Array[1];
RegWriteStatus = CMOS_Register_Writer(NO_BIT_MASKING, TransferMSB,
        0x49);
if(RegWriteStatus==1)
{
        RegMessage.LoadString(IDS_WRITE_TO_REG49HL);
        spit_Message(RegMessage);
        RegWriteStatus = 0;
}
else if(RegWriteStatus==2)
{
        RegMessage.LoadString(IDS_NO_WRITE_TO_REG49HL);
        spit_Message(RegMessage);
        RegWriteStatus = 0;
}
```

A6: Converter of Integer Value to Hexadecimal Value Function

```
void CDIETOPv12Dlg::IntegerToHex(int Integer)
{       counter = 0;
        value2 = 0;
        value1 = 0;
        power = 0.0;
        Hex_Array[0] = 0;
        Hex_Array[1] = 0;
        Hex_Array[2] = 0;
        Hex_Array[3] = 0;
        int LSB = 0;
        int MSB = 0;
        while((Integer-16)>0)
        {
                counter2 = 0;
                while(Integer>=16)
                {
                        Integer = Integer-16;
                        counter2++;
                };
                Hex_Array[counter] = Integer;
                counter++;
                Integer = counter2;
        }
        Hex_Array[counter] = Integer;
        counter++;
        for(i=0;i < counter;i++)
        {
                power = (i+0.0);
                Base16factor = (int)pow(16.0,power);
                value1 = Hex_Array[i]*Base16factor;
                i++;
                power = (i+0.0);
                Base16factor = (int)pow(16.0,power);
                value2 = Hex_Array[i]*Base16factor;

                if(i==1)
                {
                        LSB = value1 + value2;
                }
                else if(i==3)
                {
                        MSB = value1 + value2;
                }
        }
        Bit_Array[0] = LSB;
        Bit_Array[1] = MSB;
}
```

A7: I^{2}C Interface Code

```
int CDIETOPv12Dlg::CMOS_Register_Writer(int type, int Data, int Reg)
{
        Adapter Camera_Settings;
        Camera_Settings.Initialise();
        int WriteQuery = 0;
        if(type==1)
        {
                Camera_Settings.WriteAddress(KAC_9648_write_address
                        ,ACKNOWLEDGE);
                Camera_Settings.WriteData(Reg);
                Camera_Settings.Restart(KAC_9648_read_address
                        ,DO_NOT_ACKNOWLEDGE);
                Camera_Settings.ReadData(DO_NOT_ACKNOWLEDGE);
                int Register_Value2 = Camera_Settings
                        .ReadData(DO_NOT_ACKNOWLEDGE);
                Camera_Settings.WriteStop();
                int MaskApp = (Register_Value2 | Data);

                Camera_Settings.WriteAddress(KAC_9648_write_address
                        ,ACKNOWLEDGE);
                Camera_Settings.WriteData(Reg);
                Camera_Settings.WriteData(MaskApp);
                Camera_Settings.WriteStop();

                Camera_Settings.WriteAddress(KAC_9648_write_address
                        ,ACKNOWLEDGE);
                Camera_Settings.WriteData(Reg);
                Camera_Settings.Restart(KAC_9648_read_address
                        ,DO_NOT_ACKNOWLEDGE);
                Camera_Settings.ReadData(DO_NOT_ACKNOWLEDGE);
                int Register_Value3 = Camera_Settings
                        .ReadData(DO_NOT_ACKNOWLEDGE);
                Camera_Settings.WriteStop();
                if(Register_Value3==MaskApp)
                {
                        WriteQuery = 1; // There is a successful write to the
                                                camera register.
                }
                else
                {
                        WriteQuery = 2; // There is no write to the camera
                                                regsiter.
                }
```

Bit Masking the Registers

```
        }
        else if(type==0)
        {
                Camera_Settings.WriteAddress(KAC_9648_write_address
                        ,ACKNOWLEDGE);
                Camera_Settings.WriteData(Reg);
                Camera_Settings.WriteData(Data);
                Camera_Settings.WriteStop();

                Camera_Settings.WriteAddress(KAC_9648_write_address
                        ,ACKNOWLEDGE);
                Camera_Settings.WriteData(Reg);
                Camera_Settings.Restart(KAC_9648_read_address
                        ,DO_NOT_ACKNOWLEDGE);
                Camera_Settings.ReadData(DO_NOT_ACKNOWLEDGE);
                int Register_Value3 = Camera_Settings
                        .ReadData(DO_NOT_ACKNOWLEDGE);
```

Writing Specific Values to Registers

```
                Camera_Settings.WriteStop();

                if(Data==Register_Value3)
                {
                        WriteQuery = 1; // There is a successful write to the
                                                camera register.
                }
                else
                {
                        WriteQuery = 2; // There is no write to the camera
                                                regsiter.
                }

        };


 return WriteQuery;
}
```

A8: The ‘Spit Message’ Function

```
void CDIETOPv12Dlg::spit_Message(CString Message)
{
        LPSYSTEMTIME lpSystemTime = new _SYSTEMTIME;
        GetLocalTime(lpSystemTime);
        CString Message_Post;
        Message_Post.Format("[%d:%d:%d] %s",lpSystemTime->wHour,lpSystemTime-
                >wMinute,lpSystemTime->wSecond, Message);
        m_list.AddString(Message_Post);
        m_list.SetScrollPos(0,0,TRUE);
}
```

A9: The Initial Start-up of the Image Capture Process

```
void CDIETOPv12Dlg::OnBnClickedAcceptandrun()
{

        if(ConnectSocket("dSpace",2345)){
                Ethernet_Mess.LoadString(IDS_TALKING);
                spit_Message(Ethernet_Mess);
        }
        else
        {
                Ethernet_Mess.LoadString(IDS_NO_CONNECTION);
                spit_Message(Ethernet_Mess);
                return;
        }

        if(ServerStatus==1){
                UpdateData(TRUE);
                CString Space = " ";
                CString Identifier = "1";
                PhaseValues = Identifier + Space + m_LowerPhaseValue + Space +
                        m_UpperPhaseValue + Space +
                        m_PhaseIncrement + Space + m_NumImagesPerPhase + Space +
                        m_AcutatorFrequency;
                AfxBeginThread(IrisClientThread,(LPVOID)this);
                ServerStatus = 0;
        }
        else if(ServerStatus==0){
                Ethernet_Mess.LoadString(IDS_SERVER_IS_RUNNING);
                spit_Message(Ethernet_Mess);
        };
}
```

The Processing of the Settings for dSpace

A10: The ‘While Loop’ of Communications Thread

```
while (CalledArray[1]==FALSE){

    if(CalledArray[3]==TRUE && ExecutedArray[3]==FALSE && CalledArray[1]==FALSE){
            Ethernet_Mess.LoadString(IDS_PHASE_INFO);
            spit_Message(Ethernet_Mess);
            if(SentValues==1){
                    pImageFileInfo->ImageNumber = 1;
                    while(*AckMessage != *Iris_Instruction){
                            dSpace_Instruction = "IncreasedPhase";
                            m_pSocket->Send(dSpace_Instruction,100);
                            Sleep(3);
                            };
            };

            if(SentValues==0){
                    pImageFileInfo->ImageNumber = 1;
                    dSpace_Instruction = PhaseValues;
                    while(*AckMessage != *Iris_Instruction){
                            m_pSocket->Send(dSpace_Instruction,100);
                            Sleep(3);
                    }
            SentValues=1;
            }

    IntermediateString.LoadString(IDS_PHASE_STATUS);
    Ethernet_Mess.Format(IntermediateString,pImageFileInfo->CurrentPhase);
    spit_Message(Ethernet_Mess);

    pImageFileInfo->CurrentPhase = pImageFileInfo->CurrentPhase + StepPhase;
    ExecutedArray[3]=TRUE;

    }

    if(CalledArray[2]==TRUE && ExecutedArray[2]==FALSE)
    {
            ExecutedArray[0]=FALSE;
            CalledArray[0]=FALSE;
            IntermediateString.LoadString(IDS_IMAGE_STATUS);
            Ethernet_Mess.Format(IntermediateString,pImageFileInfo->ImageNumber);
            spit_Message(Ethernet_Mess);
            Ethernet_Mess.LoadString(IDS_CAMERA_TRIGGER_SETUP);
            spit_Message(Ethernet_Mess);
            AfxBeginThread(LeftCamTrigThread,static_cast<LPVOID>(pImageFileInfo));
            AfxBeginThread(RightCamTrigThread,static_cast<LPVOID>(pImageFileInfo));

            char *StrChange2 = "NULL";
            strcpy(Iris_Instruction,StrChange2);
            while(*AckMessage != *Iris_Instruction){
                    dSpace_Instruction = "CamerasReady";
                    m_pSocket->Send(dSpace_Instruction,100);
                    Sleep(3);
            }
            if(*AckMessage==*Iris_Instruction){
                    CamerasReady = 1;
            }
            ExecutedArray[2] = TRUE;
            pImageFileInfo->ImageNumber++;
    }

    if(pImageFileInfo->ImageStatus==1 && CamerasReady == 1)
    {
            Ethernet_Mess.LoadString(IDS_CONFIRMED_TRIGGER_OCCURED);
            spit_Message(Ethernet_Mess);
            CamerasReady = 0;
            pImageFileInfo->ImageStatus = 0;
            char *StrChange = "NULL";
            strcpy(Iris_Instruction,StrChange);
```

```
        while(*AckMessage != *Iris_Instruction){
                dSpace_Instruction = "TriggerCompleted";
                m_pSocket->Send(dSpace_Instruction,100);
                Sleep(3);
        }
}

if(pImageFileInfo->ImageStatus==-1 && CamerasReady == 1)
{
        IntermediateString.LoadString(IDS_TRIGGER_FAILED);
        Ethernet_Mess.Format(IntermediateString,
                        pImageFileInfo->ImageNumber,
                        pImageFileInfo->CurrentPhase);
        spit_Message(Ethernet_Mess);
        CamerasReady = 0;
}

if(CalledArray[0]==TRUE && ExecutedArray[0]==FALSE)
{
        ExecutedArray[0]=TRUE;
        ExecutedArray[2]=FALSE;
        ExecutedArray[3]=FALSE;
        CalledArray[2]=FALSE;
        CalledArray[3]=FALSE;
        pImageFileInfo->ImageStatus = 0;

        while(*AckMessage != *Iris_Instruction){
                dSpace_Instruction = "ResetedCameras";
                m_pSocket->Send(dSpace_Instruction,100);
                Sleep(3);
        }
}
Ethernet_Mess.LoadString(IDS_COMPLETED_TIGGER);
spit_Message(Ethernet_Mess);
ServerStatus = 1;
CalledArray[1]=FALSE;
}
```

A11: The Trigger Set-Up Function

```
errChk(imgInterfaceOpen ("img1", &LeftCamTrigInterfaceID));
errChk(imgSessionOpen (LeftCamTrigInterfaceID, &LeftCamTrigSessionID));
errChk(imgGetAttribute (LeftCamTrigSessionID, IMG_ATTR_ROI_WIDTH,
                        &acqWinWidthLeft));
errChk(imgGetAttribute (LeftCamTrigSessionID, IMG_ATTR_ROI_HEIGHT,
                        &acqWinHeightLeft));
errChk(imgSetAttribute (LeftCamTrigSessionID, IMG_ATTR_ROI_WIDTH,
                        acqWinWidthLeft));
errChk(imgSetAttribute (LeftCamTrigSessionID, IMG_ATTR_ROI_HEIGHT,
                        acqWinHeightLeft));
errChk(imgSetAttribute (LeftCamTrigSessionID, IMG_ATTR_ROWPIXELS,
                        acqWinWidthLeft));
errChk(imgGetAttribute (LeftCamTrigSessionID, IMG_ATTR_BYTESPERPIXEL,
                        &bytesPerPixel));
errChkLeft(imgCreateBufList(20, &LeftMainVideoCamBufListIDTriggerLeft));
bufSize = acqWinWidthTriggerLeft * acqWinHeightTriggerLeft * 4;
BYTE* InterBufferTriggerLeft = new BYTE[bufSize];
BYTE* FinalBufferTriggerLeft = new BYTE[bufSize];
bufSize2 = acqWinWidthTriggerLeft * acqWinHeightTriggerLeft * bytesPerPixel;
ImaqBuffersTriggerLeft = (void **) malloc (10* sizeof(void*));
CopyBufferTriggerLeft = (uInt8 *) malloc (bufSize2 * sizeof(uInt8));
for (i = 0; i < 10; i++)
{
        errChkLeft(imgCreateBuffer(LeftMainVideoCamSessionIDTriggerLeft,
                IMG_DEVICE_FRAME, bufSize2, &ImaqBuffersTriggerLeft[i]));
        errChkLeft(imgSetBufferElement(LeftMainVideoCamBufListIDTriggerLeft, i,
                IMG_BUFF_ADDRESS, (uInt32)ImaqBuffersTriggerLeft[i]));
        errChkLeft(imgSetBufferElement(LeftMainVideoCamBufListIDTriggerLeft, i,
```

Generic Frame
Grabber Set-Up Code

```
                IMG_BUFF_SIZE, bufSize2));
        bufCmd = (i == (10 - 1)) ? IMG_CMD_LOOP : IMG_CMD_NEXT;
        errChkLeft(imgSetBufferElement(LeftMainVideoCamBufListIDTriggerLeft, i,
                IMG_BUFF_COMMAND, bufCmd));
}
errChkLeft(imgMemLock(LeftMainVideoCamBufListIDTriggerLeft));
errChkLeft(imgSessionConfigure(LeftMainVideoCamSessionIDTriggerLeft,
                LeftMainVideoCamBufListIDTriggerLeft));
errChkLeft(imgSessionAcquire(LeftMainVideoCamSessionIDTriggerLeft, TRUE, NULL));
errChkLeft(imgCalculateBayerColorLUT(redGainTriggerLeft, greenGainTriggerLeft,
                blueGainTriggerLeft, redLUTTriggerLeft, greenLUTTriggerLeft,
                blueLUTTriggerLeft, bitsPerPixel));
errChkLeft(imgCreateBuffer(LeftMainVideoCamSessionIDTriggerLeft, FALSE, bufSize,
                &RGBBufferTriggerLeft));
static int currBufNum, lastBufNum = 0xFFFFFFFF;
errChkLeft(imgSessionWaitSignal(LeftMainVideoCamSessionIDTriggerLeft,
                IMG_EXT_TRIG0,IMG_SIGNAL_STATE_FALLING,100000));
Sleep(1500);
errChkLeft(imgGetAttribute (LeftMainVideoCamSessionIDTriggerLeft,
                IMG_ATTR_LAST_VALID_BUFFER, &currBufNum));
if ((currBufNum != lastBufNum) && (currBufNum != 0xFFFFFFFF)){
        errChkLeft(imgSessionCopyBuffer (LeftMainVideoCamSessionIDTriggerLeft,
                        currBufNum, CopyBufferTriggerLeft, FALSE));
        errChkLeft(imgBayerColorDecode(RGBBufferTriggerLeft,
                        (void *)CopyBufferTriggerLeft, acqWinHeightTriggerLeft,
                        acqWinWidthTriggerLeft, acqWinWidthTriggerLeft,
                        acqWinWidthTriggerLeft, redLUTTriggerLeft,
                        greenLUTTriggerLeft, blueLUTTriggerLeft,
                        IMG_BAYER_PATTERN_GRGR_BGBG, bitsPerPixel, 0));
        m_hBitmapImageTriggerLeft = CreateBitmap(acqWinWidthTriggerLeft,
                        acqWinHeightTriggerLeft,1,32,(BYTE*)RGBBufferTriggerLeft);
        CString m_filename;
        m_filename.Format(TriggerImageFilePathLeft,pImageFileInfoLeft.ImageNumber,
                        pImageFileInfoLeft.CurrentPhase);
        CImage MyImage;
        MyImage.Attach(m_hBitmapImageTriggerLeft);
        Sleep(300);
        MyImage.Save(m_filename);
        MyImage.Destroy();
        InterBufferTriggerLeft = NULL;
        FinalBufferTriggerLeft = NULL;
        delete InterBufferTriggerLeft;
        delete FinalBufferTriggerLeft;
        delete MyImage;
        m_hBitmapImageTriggerLeft = NULL;
        delete m_hBitmapImageTriggerLeft;
        free(InterBufferTriggerLeft);
        free(FinalBufferTriggerLeft);
        free(MyImage);
        free(m_hBitmapImageTriggerLeft);}
```

A12: Kill Focus Event Example

```
void CImageMat::OnEnKillfocusEditwoirowpointer0()
{
        UpdateData(TRUE);
        if((woi_row_pointer_0 < 16) | (woi_row_pointer_0 > 1023))
        {
                AfxMessageBox("The WOI row pointer must be greater than 16 and
                               less than 1023");
                woi_row_pointer_0 = 16;
                UpdateData(FALSE);
                WOIROWPOINTER.SetFocus();
                WOIROWPOINTER.SetSel(0,-1);
        }
}
```

A13: File Path and Browse Dialog Box

```
void CSavingSettings::OnBnClickedCalBrowse()
{
        BROWSEINFO bi;
        m_LeftCamPathAccept.EnableWindow(TRUE);
        char szPathLeftCal[MAX_PATH + 1];
        LPITEMIDLIST pidl;
        BOOL bResult = FALSE;

        IMalloc *imalloc;
        SHGetMalloc(&imalloc);
        ZeroMemory(&bi, sizeof(bi));

        bi.hwndOwner = m_hWnd;
        bi.pidlRoot = NULL;
        bi.pszDisplayName = NULL;
        bi.lpszTitle = TEXT("Select the folder you wish to save the Calibration
                              images");
        bi.ulFlags = BIF_STATUSTEXT|BIF_NONEWFOLDERBUTTON|BIF_RETURNONLYFSDIRS;
        bi.lParam = NULL;

        pidl = SHBrowseForFolder(&bi);
        SHGetPathFromIDList(pidl,szPathLeftCal);
        m_LeftCameraCal.AddString(szPathLeftCal);
        m_LeftCameraCal.SetCurSel(0);
        m_LeftCameraCal.SetFocus();
        LeftCamCalPath = (char)szPathLeftCal;

        imalloc->Free(pidl);
        imalloc->Release();
}
```

A14: ComboBox Change Example

```
void CSavingSettings::OnCbnSelchangeCombo2()
{
// The extention for saving the Calibration images
        UpdateData();
        m_LeftCamPathAccept.EnableWindow(TRUE);
        if( m_nDropListIndex < 0 ) return;
        LPTSTR CalImagesExt = "NULL";
        m_LeftCameraCalImgFormat.GetLBText(m_nDropListIndex, CalImagesExt);
        LeftCamCalImgFormat = (char)CalImagesExt;

}
```

Appendix B

Python™ and ControlDesk™ Code

B1: Detailed Python™ and ControlDesk™ Operation Explanation

What is outlined in appendix B is a descriptive explanation for the operation of the Python™ and its interaction with ControlDesk™. References to blocks of code can be found after the text in sperate appendix blocks.

B1.0 ControlDesk™ Operation and Interfacing

ControlDesk™ is a program built to control the Simulink™ diagram while it is running in the dSpace module. An interface is built using ControlDesk™ in the form a GUI to talk to the Simulink™ code while it is running. Only once the diagram is uploaded to the module can it be access.

There are two signal switches outline in the ControlDesk™ GUI the reset switch and the send signal switch. These are both edit boxes, which are connected to a switch located in the Simulink™ diagram. They are numerical edit boxes have buttons located on the right side of the box to increment the value inside the box, but these have been disabled to stop the values being change by sources other that the automation code. The automation code changes the values inside these edit boxes, in order to control the sending of the pulse out of the dSpace module amongst other functions. The edit boxes correspond to the following tasks by inserting the following values:

- "Reset"→ Trigger Pulse Control; Reset Switch (1 = On, 0 = Off)
- "Send Signal" → Trigger Pulse Control; Signal Switch (1 = On, 0 = Off)
- "Actuator Frequency" → Strobe and Actuator Control; Signal Generator (Box Value = Frequency)
- "Phase Lag" → Strobe and Actuator Control; Phase Delay (Box Value = Phase Delay)
- "Image Number" → Python Script; Image Number Update

- "Main Frequency" → Strobe and Actuator Control; Main Frequency (Box Value = Frequency)

The plotter in the GUI plots the position of the actuator, from the memory stores in the block diagram. The buttons in the GUI trigger events in the automation code that is part of the GUI written in python. The events occur when the button is down. The "Start Server" button activates the python server that will be communicating with the ICC to co-ordinate image capture with trigger pulse generation.

B1.1 Python Script Layout and Operation

The python scripting language is a high level programming language. This means that a lot of the manual memory allocation that programmers usually deal with using lower level programming languages, is taken care of in python without the programming have to worry about it. Another difference between python, than say visual C++, is that in python strings are not terminated with the null character where as in visual C++ they are. Usually the null (\x00) must be added manually in the python script when ever an instruction is sent to the ICC. Another important difference between python and visual C++, is in python when dealing with "if" or "while" statements, their conditions are followed by a colon and the contents of the statement is indented below to indicate to the python interpreter what code is part of that statement.

The use of python to automate the ControlDesk™ GUI is an option offered by ControlDesk™ and due to the versatility and ease of programming in python it is possible to create socket communications between the dSpace computer and the ICC. If the correct PCI boards required to interface with the dSpace module, were in fact installed on the ICC the socket communications could still be used. The TCP/IP provide common ground for the image capture and camera controlling visual C++ program to interface with the python script controlling ControlDesk™, and hence send the trigger pulses, even though they are two completely different programming languages.

The python script running the ControlDesk™ GUI can be broken up into the following sections:

- The Python Header
- ControlDesk™ Automation
- Ethernet Communications
- ControlDesk™ GUI Button Events

B1.1.0 The Python Header

The header for the automation is shown in Appendix B2. The header includes all the libraries being used in the following script. The header in Appendix B2 is of a similar layout to that of a visual C++ ".h" file in that specific libraries are included, or in this case imported to make use of specific functions that are part of them. In the case where "from" is used the programmer only uses certain items from that library. For example: "*from* time *import* sleep", this is saying we only want to use the sleep function from the time library. This sleep function is used to temporary halt started threads, to give other programs time to catch-up or perform other functions first.

The header also includes definition of instructions to be sent to the ICC. The computer name of the ICC is called 'Iris' and the header includes all the pre-defined instructions for the ICC.

B1.1.1 ControlDesk™ Automation

The automation of ControlDesk™ is achieved using the using the header import libraries “cdacon” and “cdautomationlid” shown in Appendix B2. The script for automating the buttons on the ControlDesk™ GUI is shown in Appendix B3. The script in Appendix B3 follows a specific format. The 4 lines of script from (6) to (9) is the format for changing the value in one numerical edit box. The value that appears in the box, and hence has an influence on the running Simulink™ diagram, is place there during the running of line (7), where the value is changed to “1”.

B1.1.2 Ethernet Communications

The Ethernet communications are achieved using the socket capabilities of the python script programming language. Appendix B4 shows the layout of the python script for the Ethernet communications. The lines from (4) to (22) are the initialisation of variable and string commands used by the Ethernet Comms. The socket set-up is the almost the same as the visual C++ applications. First a socket is created, as shown in line (23) appendix B4, using the python function socket with the flags “AF_INET” and “SOCK_STREAM”. The flags are there to define the properties of the socket. The “AF_INET” or “Address Family: Internet” flag, means that the socket will being using the Internet Protocol (IP). The “SOCK_STREAM” flag assigns the use of the Transmission Control Protocol (TCP) to the created socket and is usually combined with the IP, since the TCP uses IP to find hosts on the network. The creation of the socket in line (23) can be thought of as the building of the type of cable that will be connecting the dSpace and ICC to one another. The structure of socket communications is discussed in chapter 3. Then in line (25) we bind the address format to the socket as defined in line (11) appendix B2. This address assigns a port number to the newly created socket. The computer connecting to the dSpace computer must connect using the same socket number or nothing will happen.

The created socket is then told to listen for any connection attempts made to the dSpace computer (appendix B4, line (27)). The dSpace is set-up as a server. Once the dSpace computer has accepted the connection from the ICC, the accept command creates another socket called “CommSocket”, by which all the sending and receiving of command strings is handled.

B1.1.3 The Receive Function

All instructions that travel between the two computers are communicated using strings. When a string is received by the dSpace computer it is then checked against the array of possible commands in the receive function shown in Appendix B5. The receive function is started in the Server function (appendix B4, line (32)) as a thread. The receive command, which is part of the socket library sits inside a “while” loop that is set to loop “forever”. Once the function arrives at the “recv” command in line (4) the function stops and waits for data to arrive from the ICC. It is for this reason that the “Receive” function is ran inside a thread because if it was not the script could not perform other functions while it is waiting for it next command. The set-up of this “Receive” function is use in much the same way as the “OnReceive” event is in the visual C++ application.

Once the Receive function has acquired data it is expected that this data is in the form of a string, and because the string is arriving from the ICC it will have a null character on the end of it. Anything after this null character is just rubbish that has filled up the remaining space in the message buffer. The null character is found using pythons, “find” command which returns an index in the string of the null position. The string command is then compared against all the commands the script has been programmed to accept. If the command has not already been called, then the “CalledArray” index (corresponding to the command called), is changed to a “1” (fig B1.1), which will be registered in the main server loops in the Server function.

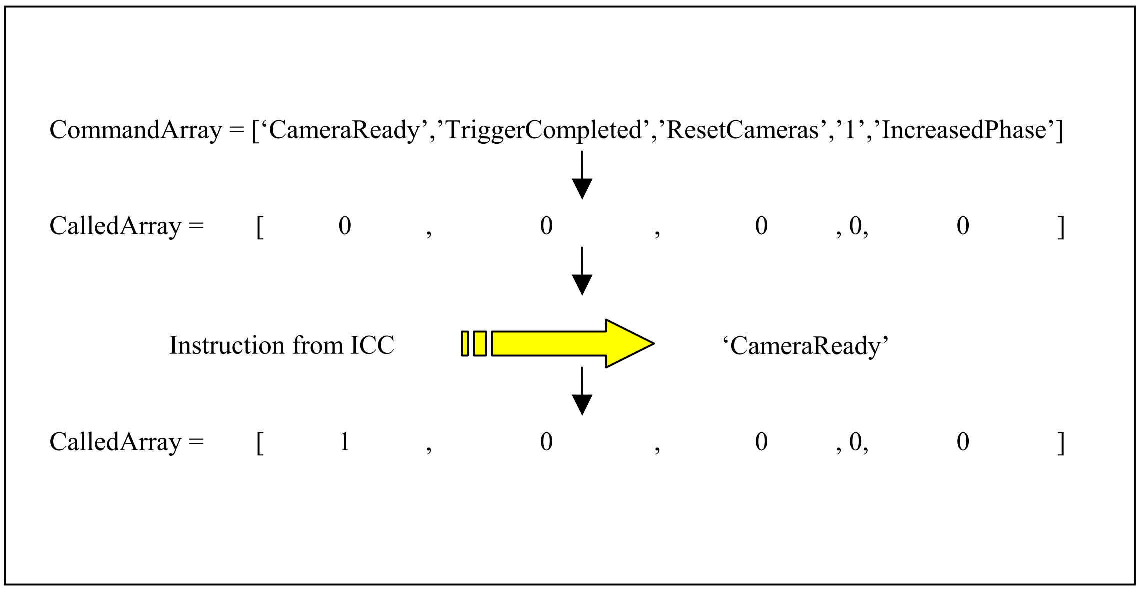

CommandArray = [‘CameraReady’,’TriggerCompleted’,’ResetCameras’,’1’,’IncreasedPhase’]
CalledArray = [0 , 0 , 0 , 0, 0]
Instruction from ICC
‘CameraReady’
CalledArray = [1 , 0 , 0 , 0, 0]

Figure B1.1: The changing of the CalledArray upon an instruction from the ICC

B1.1.4 The Server Function

The server function handles all the processing of the received instructions from the ICC. All the tasks for controlling the triggering sequence are contained within 3 “while” loops, arranged as shown in figure B1.2.

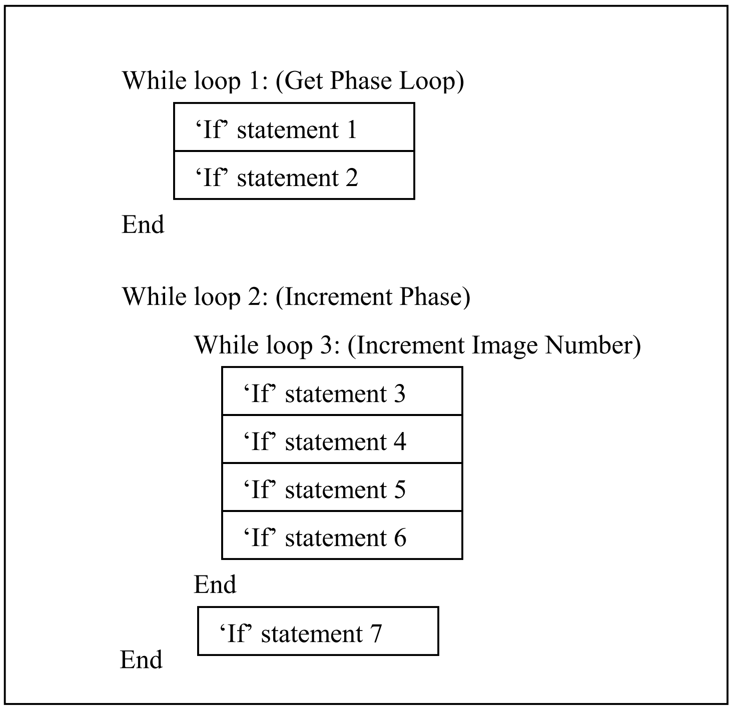


Figure B1.2: The pseudo layout of the server function script

The server script for handling the instructions from the ICC follows the initialisation of the socket script.

Throughout the server script there is smaller while loops that provide a type of “handshaking” between the two computers. An example of theses loops can be seen in appendix B4 lines (54) to (56). What is happening, is that any instruction being sent to the ICC, is being sent over and over again, in periods determined by the “sleep” function until the ICC sends an acknowledgment string to the dSpace computer which is then picked up by the receive function and then made available to the “while” loop so that it stops sending. The arrangement of this message sending is illustrated in figure B1.3.

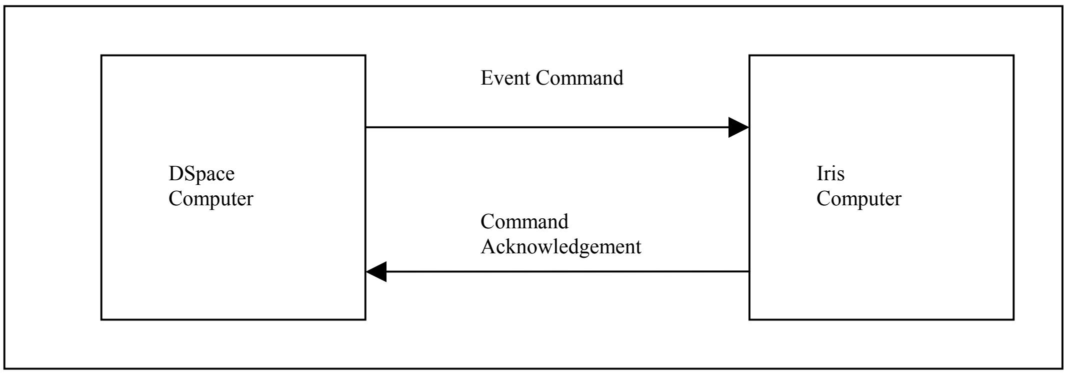


Figure B1.3: This shows the command string is being constantly sent to the ICC until the ICC send back the acknowledgment that the command has arrived

This process of handshaking is done in the visual C++ application as well in a similar format. The communications between the two computers occur as follows:

- The start button on the ControlDesk™ GUI is press starting the server on the dSpace computer, which then waits for a connection from the ICC.
- The connection is made from the ICC and accept by the dSpace computer at which point an instruction is sent to the ICC requesting information on the phases and actuator frequency (‘if’ statement 2).

- Once the phase and actuator information is received by the dSpace computer the information is processed inside “if” statement 1.
- The program then moves into “while” loop 2 and then into “while” loop 3 then to “if” statement 3 which send the instruction to the ICC to set-up the cameras and frame grabbers for receiving the triggers.
- Once the acknowledgement is sent from the ICC that the cameras are ready, the trigger function is executed in “if” statement 4.
- The ICC, upon receiving the trigger signal, then captures the images and saves them and then sends the ‘TriggerCompleted’ to the dSpace computer.
- Then in “if” statement 5 a reset instruction is sent to the ICC, to reset the entire ‘if’ statements on the ICC.
- The following ‘if’ statement 6 resets the entire ‘if’ statements on the dSpace computer.
- Once the “while” loop 3 has meet it conditions the program moves on to “if” statement 7 which sends the instruction to the ICC to increase its value of phase.

The above process continues for the user’s specified amount of images per phase increment and number of phase increments.

The “while” loop 2, provides the increasing of the phase from a user specified being, end and increment. The “while” loop 3 provides the increasing number of images per phase with the exception of ‘if’ statement 7, every statement within ‘loop 3 in figure B1.2 is repeated for a user defined number of images.

B1.1.5 ControlDesk™ GUI Button Events

The ControlDesk™ buttons are outlined in appendix B6. The functions in lines (1) and (3) are part of the ControlDesk™ automation library and are activated when the when either button on the GUI is press down. The first of the button events on line (1) starts the server thread, using the function shown in line (2) of appendix B6. The function shown in line (2) is part of the thread library and starts the server function as a thread. The second of the button events on line (3) runs the trigger function, which is used to test the response of the cameras and frame grabbers.

B2: <u>The Python Header</u>

```
(1)     import socket
(2)     import string
(3)     import thread
(4)     import cdacon
(5)     from cdautomationlib import *
(6)     from time import sleep
(7)     GrabbedPhases = 1
(8)     host = ''
(9)     port = 2345
(10)    Bufsize = 1024
(11)    ADDR = (host, port)
(12)    AckMessage = 'AckMessage'
(13)    AckMessage2 = 'AckMessage\x00'
(14)    Iris_Instruction1 = 'GetPhases\x00'
(15)    Iris_Instruction2 = 'SetCameras\x00'
(16)    Iris_Instruction3 = 'Reset\x00'
(17)    Iris_Instruction4 = 'End\x00'
(18)    global NumImagePerPhase
(19)    ImageCount=1
```

B3: <u>ControlDesk Automation</u>

```
(1)     def Trigger():
(2)       import pythoncom
(3)       pythoncom.CoInitialize()
(4)       sleep(0.5)
(5)       # triggering://dSPACE NumericInput Control_1:WriteData
(6)       Instrumentation().ConnectionController.DisableSystemPoll()
(7)       Instrumentation().Layouts.Item("c:\\users\\crispen berg\\diet_triggering_test_Testing\\Matlab
                Files\\triggering.lay").Instruments.Item("dSPACE NumericInput Control_1").Value =1.0000000000000000
(8)       Instrumentation().ConnectionController.ProcessAnimationEvent("triggering://dSPACE NumericInput
                Control_1","WriteData")
(9)       Instrumentation().ConnectionController.EnableSystemPoll()
(10)      sleep(0.5)
(11)      # triggering://dSPACE NumericInput Control_1:WriteData
(12)      Instrumentation().ConnectionController.DisableSystemPoll()
(13)      Instrumentation().Layouts.Item("c:\\users\\crispen berg\\diet_triggering_test_Testing\\Matlab
                Files\\triggering.lay").Instruments.Item("dSPACE NumericInput Control_1").Value =
                0.00000000000000000
(14)      Instrumentation().ConnectionController.ProcessAnimationEvent("triggering://dSPACE NumericInput
                Control_1","WriteData")
(15)      Instrumentation().ConnectionController.EnableSystemPoll()
(16)      sleep(0.5)
(17)      # triggering://dSPACE NumericInput Control:WriteData
(18)      Instrumentation().ConnectionController.DisableSystemPoll()
(19)      Instrumentation().Layouts.Item("c:\\users\\crispen berg\\diet_triggering_test_Testing\\Matlab
                Files\\triggering.lay").Instruments.Item("dSPACE NumericInput Control").Value =
                1.0000000000000000
(20)      Instrumentation().ConnectionController.ProcessAnimationEvent("triggering://dSPACE NumericInput
```

The script to change the value in one numerical edit

```
            Control","WriteData")
    Instrumentation().ConnectionController.EnableSystemPoll()
    sleep(0.5)
    # triggering://dSPACE NumericInput Control:WriteData
    Instrumentation().ConnectionController.DisableSystemPoll()
    sleep(0.5)
    Instrumentation().Layouts.Item("c:\\users\\crispen berg\\diet_triggering_test_Testing\\Matlab
            Files\\triggering.lay").Instruments.Item("dSPACE NumericInput Control").Value =
            0.00000000000000000
    Instrumentation().ConnectionController.ProcessAnimationEvent("triggering://dSPACE NumericInput
            Control","WriteData")
    Instrumentation().ConnectionController.EnableSystemPoll()
    sleep(0.5)
    Instrumentation().Layouts.Item("c:\\users\\crispen berg\\diet_triggering_test_Testing\\Matlab
            Files\\triggering.lay").Activate()
    Instrumentation().Layouts.Item("c:\\users\\crispen berg\\diet_triggering_test_Testing\\Matlab
            Files\\triggering.lay").Activate()
    pythoncom.CoUninitialize()
```

B4: Socket Communications

```
def Server():
    import pythoncom
    pythoncom.CoInitialize()
    global dSpace_Instruction
    dSpace_Instruction = 'NULL'
    GrabbedPhase = 1
    PhaseCount = 0
    global SetCameras
    global ExitReceive
    ExitReceive = 0
    SyncPhases = 0
    global CommandArray
    global CalledArray
    global ExecutedArray
    global BufferArray
    global QuitReceive
    CommandArray = ['CamerasReady','TriggerCompleted','ResetedCameras','1','IncreasedPhase']
    CalledArray = [0,0,0,0,0]
    ExecutedArray = [0,0,0,0,0]
    BufferArray = [0,0,0,0,0]
    QuitReceive = [0,0]
    SetCameras = 1
    ServerSocket = socket.socket(socket.AF_INET,socket.SOCK_STREAM)
    print'starting server'
    ServerSocket.bind(ADDR)
    print'Listening'
    ServerSocket.listen(5)
    global CommSocket
    CommSocket, addr = ServerSocket.accept()
    print'Connection accpeted'
    print'Starting the receive'
```

All the socket communication initialising

……[ControlDesk™ Automation Code]…………

```
    thread.start_new_thread(Receive,())
    ImageCount=1
    while(GrabbedPhase==1):
        if (dSpace_Instruction[0]=='1'):
            Values = dSpace_Instruction.split()
            LowerPhaseValue = int(Values[1])
            UpperPhaseValue = int(Values[2])
            PhaseIncrement = int(Values[3])
            NumImagePerPhase = int(Values[4])
            ActuatorFrequency = int(Values[5])
            NumberOfPhases = (UpperPhaseValue-LowerPhaseValue)/PhaseIncrement
```

‘If’ Statement 1

```
        GrabbedPhase = 0
………[ControlDesk™ Automation Code]…………
        print '\nJust got the Phases\n'
    if(dSpace_Instruction[0]!='1'):
        CommSocket.send(Iris_Instruction1)                          'If' Statement 2
        sleep(0.5)

    while(PhaseCount<=UpperPhaseValue):
………[ControlDesk™ Automation Code]…………
        print'Current strobe phase is:',PhaseCount
        while(ImageCount<=NumImagePerPhase):
            if(SetCameras ==1):
                print 'The image count is:',ImageCount
……....[ControlDesk™ Automation Code]……………
                dSpace_Instruction = 'NULL'
                while(cmp(dSpace_Instruction,AckMessage)!=0):       'If' Statement 3
                    CommSocket.send(Iris_Instruction2)
                    sleep(0.5)
                SetCameras = 0

            if(CalledArray[0]==1 and ExecutedArray[0]==0):
                ExecutedArray[0]=1
                Trigger()                                           'If' Statement 4
                ImageCount = ImageCount+1
                ExecutedArray[2]=0

            if(CalledArray[1]==1 and ExecutedArray[1]==0):
                ExecutedArray[1]=1
                dSpace_Instruction = 'NULL'                         'If' Statement 5
                while(cmp(dSpace_Instruction,AckMessage)!=0):
                    CommSocket.send(Iris_Instruction3)
                    sleep(0.5)
            if(CalledArray[2]==1 and ExecutedArray[2]==0):
                print'dSpace program has been reseted'
                ExecutedArray[2]=1
                ExecutedArray[0]=0
                ExecutedArray[1]=0                                  'If' Statement 6
                CalledArray[0]=0
                CalledArray[1]=0
                CalledArray[2]=0
                SetCameras=1
                dSpace_Instruction = 'NULL'
        if(PhaseCount!=UpperPhaseValue):
            while(cmp(dSpace_Instruction,AckMessage)!=0):           'If' Statement 7
                CommSocket.send(Iris_Instruction1)
                sleep(0.5)

        PhaseCount = PhaseCount + PhaseIncrement
        ImageCount = 1
    dSpace_Instruction = 'NULL'
    QuitReceive[0] = 1
    while(cmp(dSpace_Instruction,AckMessage)!=0):
        CommSocket.send(Iris_Instruction4)
        sleep(0.5)
    CommSocket.close()
    ServerSocket.close()
```

B5: Receive Function

```
def Receive():
    while 1:
        global dSpace_Instruction
        message = CommSocket.recv(Bufsize)
        Null_Position = string.find(message,'\x00')
        dSpace_Instruction = message[:Null_Position]
        Processed_dSpace_Instruction = dSpace_Instruction.split()

        for count in range(4):
            if(cmp(CommandArray[count],Processed_dSpace_Instruction[0])==0 and
                                            CalledArray[count]!=1):
                CalledArray[count]=1

        if(QuitReceive[0]!=0 and cmp(dSpace_Instruction,AckMessage)==0 ):
            break

        CommSocket.send(AckMessage2)
```

B6: ControlDesk™ GUI Button Events

```
def On_Instrumentation_triggering_dSPACEPushButtonControl_ButtonDown(OrderIndex):
    thread.start_new_thread(Server,())

def On_Instrumentation_triggering_dSPACEPushButtonControl_1_ButtonDown(OrderIndex):
    Trigger()
    print 'Just sent the trigger'
```