A Remotely Automated Microscope for Characterizing Micro Electromechanical Systems (MEMS)

by

Danny Seth

B.S., Electrical Engineering
Northeastern University, June 1999

Submitted to the
Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Science in Electrical Engineering and Computer Science

at the

MASSACHVSETTS INSTITVTE OF TECHNOLOGY

June 2001

© MMI Massachvsetts Institvte of Technology
All rights reserved.

BARKER
MASSACHUSETTS INSTITUTE OF TECHNOLOGY
JUL 11 2001
LIBRARIES

Author .....
Department of Electrical Engineering and Computer Science
May 1, 2001

Certified by .....
Donald E. Troxel
Professor of Electrical Engineering and Computer Science
Thesis Supervisor

Accepted by .....
Arthur C. Smith
Chairman, Department Committee on Graduate Students

A Remotely Automated Microscope for Characterizing Micro Electromechanical Systems (MEMS)

by

Danny Seth

Submitted to the
Department of Electrical Engineering and Computer Science
on
May 1, 2001,
in partial fulfillment of the requirements for the degree of
Master of Science in Electrical Engineering and Computer Science

Abstract

Designers of Micro Electromechanical Systems need tools to test the electrical as well as mechanical properties of the devices they fabricate. Computer microvision acts as a good analysis tool during the testing and development stages of the design process. Computer microvision involves the use of light microscopy and video imaging to acquire 3-dimensional images at multiple phases of motion. In this research, a computer microvision system is defined and implemented. The computer microvision system includes a PC, an automated X-Y-Z stage, a camera, and a piezo electric device. Custom hardware includes the design of a module for a PCI interface that acts as a central controller for stimulus and stroboscopic illumination.

There are benefits in being able to run the system remotely and support a multi-client environment. The computer microvision system uses an Apache web server to provide remote access and all communication is done via "messages". Java servlets form an integral part of the server side software in overcoming HTTP's inability to handle state. A client connects to the server's URL via a webbrowser and is presented with a Graphical User Interface (GUI) that acts as medium to access all aspects of the computer microvision system. The GUI, written in Java, also supports remote focusing which can be done either manually or automatically. The various hardware settings can be configured, an experiment or analysis can be launched, and the results can then be viewed.

Thesis Supervisor: Donald E. Troxel
Title: Professor of Electrical Engineering and Computer Science

Electronic Source

An electronic copy of this thesis can be obtained from http://www-mtl.mit.edu/~dseth. This web site also contains documents that have not been included in this thesis. They include the VHDL source code and associated block diagrams, the Java source code for the client and the server, the drivers that operate the hardware, and relevant data sheets. Further information can be obtained by sending an email to dseth@alum.mit.edu.

Acknowledgments

I would like to acknowledge the support of many people that have made this research possible and my MIT experience to be an enjoyable one.

First, I would like to thank Professor Donald Troxel for being the best advisor I could have asked for. I can never forget that Professor Troxel gave me his trust by allowing me to be part of his research group while I was still a special student at MIT. I would like to thank him for his support and advice that made this research my greatest learning experience. His insightful questions always adressed the heart of the matter and made me see things from presepectives I had overlooked. His understanding of the time commitment imposed by graduate H-level courses made this research experience a pleasurable one and allowed the best to come out of me. Over the past years, as a RA and then a TA, I have found Professor Troxel to be a great person who is caring and fun to talk to. I firmly believe that without his support I would not have been able to come this far, and I will be forever grateful to him.

I would like to thank Professor Denny Freeman and the members of his research group for providing guidance and support throughout this research. I would like to thank Mike McIlrath for helping out with several system integration issues and for asking tough questions that allowed me to gain better insight of the problem. Over the years, Francis Doughty has helped me get around MIT and has provided extensive support in the so many things such as Latex commands and 3-hole punchers that made life so much easier. Thanks for everything Fran.

Upon first joining the group, Erik Pedersen was instrumental in bringing me up to speed with the ongoing research. He took the time to make sure I understood everything and his positive attitude and laughter simplified everything. Xudong Tang and I worked closely for the longest time. The discussions on research and "life" were intriguing and a lot of fun. Xudong and I started with a dream to have a building named after us at MIT. Towards the end, we realized a conference room named after us would be sufficient.

I must also not forget to give thanks to my officemate, William Moyne, for his good humor, and giving me a lot of space when I moved in. I eventually became the heir to his office. Syed Alam was a constant companion in classes and constant source of laughter and great conversation. His being around made the office experience an enjoyable one. Tom Lohman and Myron Feeman (Fletch) were always there to help out with computer issues and share a few good laughs.

At Northeastern University, I would like to thank Professor Martin Schetzen and Professor Phillip Serafim who encouraged me to pursue graduate studies at MIT. They were always available whenever I needed their support and for that I'm grateful. In the Course VI graduate office, I would like to recognize Marilyn Pierce, who has helped me out so many times. She is committed to students and her dedication and skill is appreciated by every graduate student in the department.

MIT is a challenging place, and there were many times when I just felt overwhelmed. However, it was the support of friends that brightened my days. Fatih Yanik, now at Stanford, was my 6.111 final project partner, who convinced me that I could also succeed at MIT. The conversations I have with Fatih are enriching as well as inspiring and I cherish the friendship I share with him. Aatif Abbasi at Northeastern University was always there to point out that I had to cross the Charles River into the non-nerdy side to truly enjoy life. Thanks to him, the weekends at Avalon were a lot of fun. Raj Midha and Gary Hall have been excellent friends. Their laughter, positive attitude, and unique perspective has been a constant encouragement to me. The time I spent with them during both Spring breaks is quite memorable and hopefully we will continue this into an annual tradition. I would also like to acknowledge the friendship of Manish Bhardwaj, Hans Calebaut, Eric Caulfield, Yu Chen, Albert Chow, Karen Gonzalez, Linda Lin, Raj Rao, Sunil Rao, Oguz Silahtar and many others. The time spent with them made MIT a fun place for me.

I have to thank my friends and all the people that supported me back home in India. Thanks to Amit for being a friend I can always count on even though I haven't done a good job in staying in touch.

I'm grateful to MIT and everything that this place represents. Although, it has

been a lot of hard work and several all nighters, I have enjoyed every moment at MIT. My experiences here have been pleasurable as well as satisfying. The people here are the best and the brightest and their commitment to success (in research, life, or entrepreneurship) motivates me. MIT has had a strong impact on me that will shape my life in the years to come. I only hope I make the best of this opportunity that very few people are entitled to. While I don't want to leave, I believe the time is right for me to take a break and evaluate what I really want to do. I can always come back for my Ph.D.

I would like to express my gratitude to my grandparents, parents, and brother who believed in me and supported my struggle to get in and then get out of MIT. The blessings of my grandparents have enlightened me and made everything in my life possible. I have no words to express my thankfulness to my parents who worked very hard to fulfill their dream of getting their children an education in the U.S. Deepak has always been there for me and made sure I was eating okay (among the many other things). Thank you Deepak, I'm very lucky to have you as a brother. Pitaji, Chiji, Papa, Mommy, and Deepak, this one is for you - I love you all.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Micro Electromechanical Systems (MEMS) are devices that react mechanically to electrical stimuli. They can be used to create complex machines with micron feature sizes. MEMS is an enabling technology where current applications include accelerometers, pressure, chemical and flow sensors, micro-optics, optical scanners, and fluid pumps.

MEMS are fabricated using batch-processing techniques similar to those utilized in the design of digital/analog integrated circuits (IC's). Unlike electronics, however, simple methods for testing MEMS devices do not exist. This imposes limitations on their design and manufacture. Since designers cannot visualize the actual motion of the structures that they build, mechanical problems can go undiscovered. Furthermore, manufacturing costs can be high since mechanical testing may not be practical until late in the manufacturing process. This is where computer microvision acts as a good analysis tool to analyze the X,Y, and Z motion of a MEMS device during the development and testing stages of the design process.

1.1 Computer Microvision

Computer microvision is an evolving field where common machine vision algorithms are used to analyze microscopic devices. In computer microvision, a computer system works with a microscope and external hardware for data acquisition. It then processes

the results either locally or remotely. Previous applications of computer microvision include the study of the tectoral membrane of the inner ear [6].

The underlying principle of a computer microvision system is light microscopy and video imaging. External hardware stimulates the device to be tested and generates pulses that control the stroboscopic illumination. Images acquired from a camera at multiple phases of motion can then be inspected and also processed by software.

1.2 Components of a Microvision System

The components of computer microvision consist of a device that magnifies the appearance of an object under study (microscope), a computer to implement control and data processing, a device that can collect and transmit visual data, and a component that can initiate an excitation upon the MEMS device.

1.2.1 Computer System

The computer system must have the ability to interface with the visual data collector in order to understand what the device may be doing. The visual data collector must be able to sense the object or some aspect of it with some minimal amount of accuracy. There has been research done to determine a fast and reliable algorithm to accomplish precisely this sort of detection at the sub-pixel level [3, 4, 2].

The visual data collector consists of a camera having a limited pixel resolution and an interface to the computer system. Often, the interface will come in the form of a card that plugs into a slot in the computer system. The card allows the computer to capture images taken by the camera as well as control when the camera is to take a picture. Other parameters such as exposure time and camera gain (used to control dark current) can also be set via the computer.

In addition, the computer must interface with an apparatus to cause an excitation upon the object under study. In the field of MEMS, this is usually an excitation voltage. When a defined voltage is applied properly to the device, it should react/move in a certain manner. The way that the device reacts to the excitation is therefore the

most important aspect under study. This apparatus is part of the SPG (Strobe Pulse Generator) Module. The SPG Module is also used to illuminate the moving device by turning on the LED (Light Emitting Diode) during periods of time that correspond to a certain phase in motion. Furthermore, the strobe pulses must be synchronized with both the camera trigger and the excitation signal going to the device.

1.2.2 Microscope

The microscope consists of a computer controlled X,Y stage upon which the MEMS device is mounted. The computer system also has control over the focus knob of the microscope which allows large-scale, but crude motion (1 μm to 15 cm), in the Z-Axis. The microscope objective is mounted on a piezo-electric device that is also controlled by the computer. The piezo-electric device is used for sub-micron motion (1 nm to 80 μm) in the Z-axis. Unlike the microscope's focus knob, the piezo-electric device uses a closed loop DC-Servo motor to provide a very accurate displacement in the Z-axis. The microscope has an LED that illuminates the moving MEMS device. The LED is operated by the SPG Module and hence the computer can control it.

In essence, the computer system has full control over the microscope. As a result, the computer has enough information to aid in the test process. It has information about the signal used to excite the device since it is interfaced with the excitation device. In addition, the computer can determine to some degree of accuracy how the object behaves as a result of the specified input since it has an interface to the camera. Furthermore, this behavior can be compared to a desired response of a specific input.

1.3 Remote Access

There are benefits to being able to run the system remotely. First of all, in an ideal system, there should be no contact between the device under test and unfiltered air. In particular, the part of the system that houses the device under test should be in a clean room. The reason for this is that in the development stages of a MEMS device, there may not be a package for the device under test. In addition, once a device has

been packaged, there is no possible way to visualize its motion. Thus, in order to test and observe the device, it must be in a controlled environment. In order to avoid having to work inside of a clean room or a controlled environment, the system should be designed to handle remote access.

Also, in the process of designing and manufacturing devices such as MEMS, there are many fields of study and expertise involved. In order to allow collaboration of these fields into a group that can work on the design of the same MEMS device, remote access would be a much better alternative than having to gather all of the engineers in the same place.

The system architecture will have much to do with how they system should be accessed. Since there will be one computer acting as the controller, this system can easily be thought of as the server for all of the data and for all of the parameters of the present system.

1.4 Relevant Work

This section describes the relevant work that has been done by researchers in a similar area. It also explains the work that has been done for the current project by other graduate students. It is important to understand the relevant work so that the results from prior work can aid in the present work as well as avoid the overlap of ideas among the research.

1.4.1 Remote Microscope

At present there is a remote microscope in operation at MIT. The interface and the overall architecture has been refined by several graduate students at MIT [9, 13, 10] The primary differences between that system and the present microvision system have to do with the fact that with the remote microscope project, the devices under test are not in motion. Thus, the tools used to analyze the devices will be different. Also, MEMS devices require an excitation input. Thus, the tools must interact with the data and the input to the device. However, there are some similarities between

the systems. Most importantly, both involve the use of a microscope/camera system connected to a controlling computer.

1.4.2 Present System Tools

There has been work done on the present MEMS System to develop a way to examine and analyze MEMS devices. Two tools that have been used extensively for this work include one that can detect subpixel movement of a 3D object [3, 4] and one that can analyze the motion in a certain Region of Interest (ROI) on a set of images. With these two tools working together, a set of data taken from a MEMS device can be analyzed for various ROI's. Information that can be gathered includes the motion of the device under test in the X, Y and Z directions. Graphs can then be produced to show the phase and magnitude response in the region under study [2].

1.4.3 Present Remote Interface

The existing system architecture designed by Jared Cottrell [1] is based upon the server/client relationship. The server is the means through which somebody using the web can have access to hardware connected to the system or the analysis tools residing on the system. A computer system is currently in place that has a web server running on top of the operating system. The web server allows clients to interact with the system using the standard HTTP protocol. The server software that is currently running is the Apache Web Server as it supports the use of Java Programs called servlets. It is these servlets that overcome the inability of the standard HTTP interactions to deal with state. The computer supports a dual boot to the Linux and Windows NT system. Since the servlets compiled code is OS independent, it can be used for both operating systems.

At present, a Java module resides on the server that appropriately handles messages sent from the client. There are also Java modules that take care of inter-client communication as well as control and access of information for each client connected to the server.

Erik Pedersen [12] implemented a customizable Java based Graphical User Interface (GUI) to send and receive messages from the server, i.e. interact with the module described in the above paragraph. In this implementation, a client connects to the server via a web-browser, and obtains a GUI. Messages can be sent to the server to execute C-programs that control the hardware or query the state of the system.

1.5 Thesis Statement

Designers of Micro Electromechanical Systems need tools to test the electrical as well as mechanical properties of the devices they fabricate. Computer microvision acts as a good analysis tool during the testing and development stages of the design process. Computer microvision involves the use of light microscopy and video imaging to acquire 3-dimensional images at multiple phases of motion. In this research, a computer microvision system is defined and implemented. The computer microvision system includes a PC, an automated X-Y-Z stage, a camera, and a piezo electric device. Custom hardware includes the design of a module for a PCI interface that acts as a central controller for stimulus and stroboscopic illumination.

There are benefits in being able to run the system remotely and support a multi-client environment. The computer microvision system uses an Apache web server to provide remote access and all communication is done via "messages". Java servlets form an integral part of the server side software in overcoming HTTP's inability to handle state. A client connects to the server's URL via a webbrowser and is presented with a Graphical User Interface (GUI) that acts as medium to access all aspects of the computer microvision system. The GUI, written in Java, also supports remote focusing which can be done either manually or automatically. The various hardware settings can be configured, an experiment or analysis can be launched, and the results can then be viewed.

1.6 Thesis Work Involved

The research presented in this thesis defines, implements and *presents* a remotely operated MEMS Characterization System. The ultimate goal was to present a *working* system. Several graduate students and four theses over a span of two years have already contributed to this project [1, 12, 14, 16]. The work involved in this thesis required mastering their individual work and making it work together as a *system.*

Jared Cottrell presented the architecture for the servlets while Erik Pedersen presented a basic GUI that could remotely connect and excite these servlets. However, at the time of their thesis completion, there was no hardware connected to the server and only the Windows NT system was supported. All the software had to ported to Linux and an upgrade to latest version of Java and Swing was required. The camera and the corresponding driver was different and presented a new approach to image acquisition. The SPG Module was designed and implemented which could excite the MEMS device and we could now get real data.

As the hardware evolved and drivers were written, the messaging format was refined along with the addition of new messages. In addition, several modifications were made to the underlying architecture of the servlets to support multitasking and provide feedback on an executable's status. Erik's GUI was virtually rewritten to support multitasking and support new features while providing a user friendly operating environment.

All this involved being responsible for the entire hardware and filling several software gaps. Upon attaining an acceptable system, support was then added to the client and the server for essential features such as remote focusing, auto focusing, selecting a ROI, launching and viewing experimental results, etc.

1.7 Organization of the Thesis

Chapter 2 highlights the basic principles of computer microvision and the equipment that was assembled to accomplish the task. It will show how the hardware was

integrated and the various software drivers that were written to allow the computer to control the microscope.

Chapter 3 covers the design and architecture of the SPG Module. The SPG Module is the main controller for stroboscopic illumination. It interfaces with the camera and the light source while providing an excitation voltage to the MEMS device.

Beyond Chapter 3, the focus is on remote access. Chapter 4 highlights the architecture used for remote access where the notion of "messages" and "handlers" will be introduced. The emphasis will be on how the messaging protocol links the client and server which have different software development environments.

Chapter 5 discusses the basic hardware handlers needed to operate the microscope equipment, i.e call upon the executables established in Chapter 2. Chapter 6 will introduce the command module interface and its associated handlers.

Chapter 7 highlights the need for remote focusing and the various architectural issues that were considered in implementing it. The handlers used for auto focusing will also be discussed.

Building upon the foundation of the handlers, Chapter 8 provides an introduction of the client interface. Section 8.3 shows why we need a new user interface and provides and pictorial view of the user interface that was created.

A conclusion at the end provides a discussion on the accomplishments and possible future work.

Chapter 2

Principles of Microvision

The purpose of this chapter is to establish the fundamental principles of computer microvision and show how multidimensional motion analysis can be performed on a MEMS device.

2.1 Computer Microvision Method

Figure 2-1 [5] shows a test structure placed on the stage of a light microscope and driven with a periodic stimulus. The motion of the device in response to the stimulus is captured via stroboscopic illumination. The 3-D behavior of the device can be monitored by taking images at multiple focal planes [2, 6]. Motion estimates for X,Y, and Z can then be extracted from a set of images by using computer vision algorithms [3, 4].

2.2 Computer Microvision Hardware

Commercial microscopes from Olympus, Zeiss, Lica, and other vendors are expensive ($50,000 plus). These microscopes often provide nice but costly features which are not a necessity for the system we are trying to build. They may include complicated light interferometrics, multiple high-powered objectives, etc.

An underlying advantage of the computer microvision method of analysis is the

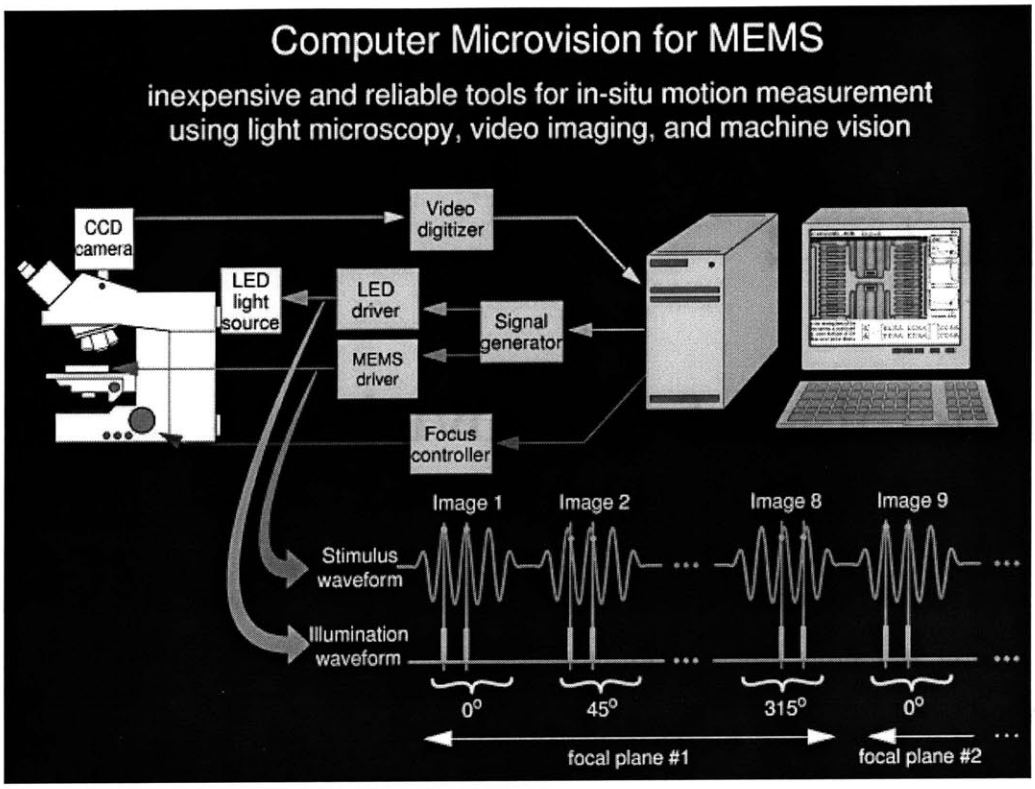

Figure 2-1: Computer Microvision for MEMS [5]

low cost of the computer system involved. In fact, the cost of a computer system is negligible ($ 2,000) compared to the cost of a microscope. A goal of this research is therefore to present the essential components required by a MEMS characterization system. This can only be possible if we create a generic microscope. Another advantage of the generic approach is that it doesn't restrict the microscope to any particular brand or type. Instead, it presents the basic building blocks that can be integrated or added on to an existing microscope.

Figure 2-2 and Figure 2-3 show the front and side views of the of the generic microscope that has been put together. A vertical rod is tightly screwed to a heavy base plate. To this rod is attached a Z-Axis stage, also known as the Modular Focus Unit (MFU). The MFU is provided by Nikon and allows manual focusing at 1 μm resolution while supporting a scanning range of 15 cm.

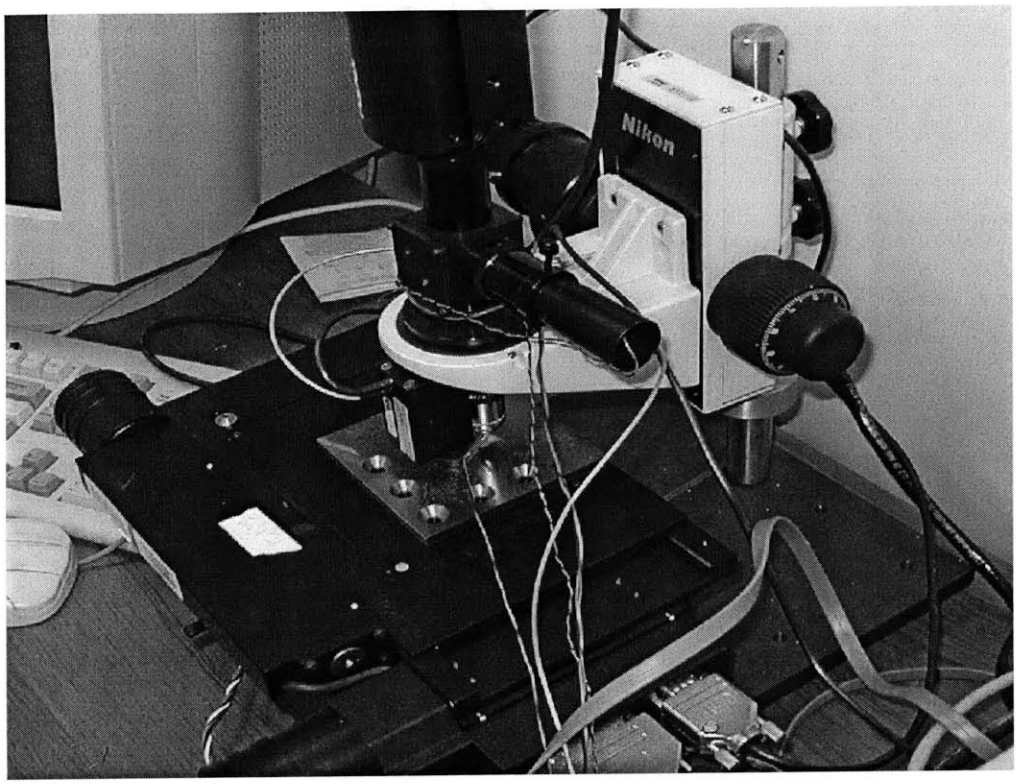

Figure 2-2: Side View of Microscope

2.2.1 Camera

A gray-scale CCD camera mounts on the MFU. The camera interfaces to the computer system via a frame grabber that occupies a PCI slot of the computer. A turret[1] provides the interface between the camera and the objective.

It is desirable that our microvision system support the Windows NT as well as the Linux platform. However, we were not able to find a camera that would work under both operating systems. We therefore used an 8-bit Depict type CCD camera from Opteon to be used for the Windows operating environment, and a 12-bit CCD camera from Pulnix to be used for the Linux OS. Both of these cameras were installed on the computer system which supports a dual-operating-system boot. The appropriate camera is selected by software depending on the OS in use. However, the appropriate camera has to be mounted onto the MFU by the system designer.

A driver for the camera was not written as it is part of the command module

[1] A rotating device holding various lenses.

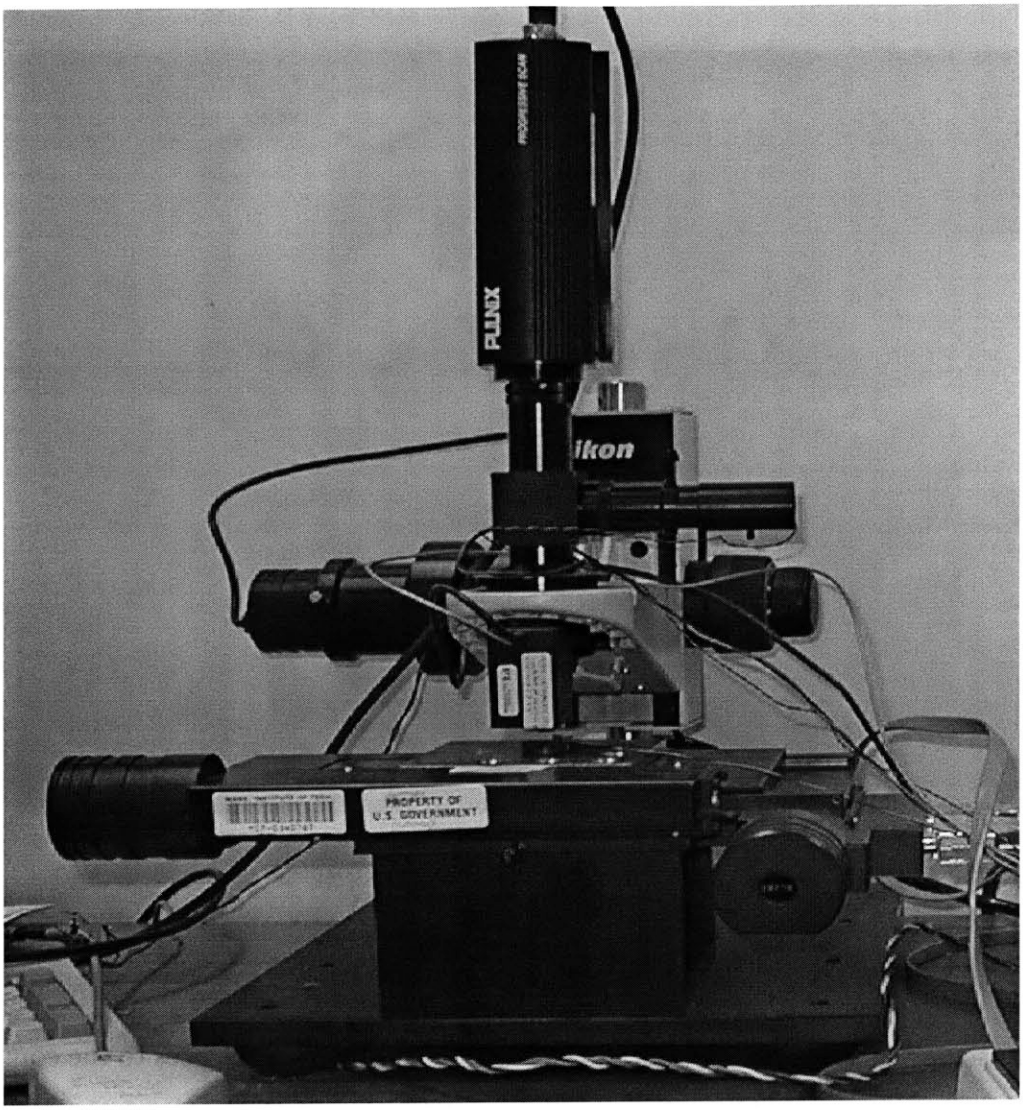

Figure 2-3: Front View of Microscope

interface. Chapter 6 will provide an introduction to the command module interface and show the subroutines used to acquire images from the camera.

2.2.2 PIFOC

A Piezoelectric Microscope Objective Nanopositioner (PIFOC), attaches between the turret/objective interface. The pifoc is used to focus images up to 80 μm along the Z-axis with a 10 nm resolution. Since the pifoc is made up of piezoelectric material, its length can be controlled by applying a voltage. A Position Servo Controller, which interfaces to the computer system via an RS-232 interface, is used to generate the voltage needed by the pifoc. The pifoc and the controller are both supplied by Physik

Instruments.

A C-program was written to control the pifoc. The arguments to the program are as follows :

`piezo argument` ***xyz***

where `argument` can be any one of the following shown in Table 2.1 and ***xyz*** is an integer that can be positive or negative with the units of μm.

Table 2.1: Arguments for the Piezo Driver

init	Initializes the controller. Must be done every time the computer is restarted.
goto_position	Puts the piezo at the displacement specified by *xyz*.
goto_ origin	Takes the piezo to location 0.
step	Sets the step size of the controller to *xyz*. Can be a positive or negative number.
move	Adds the value of "step" to current position. The piezo will clamp to it maximum or minimum value if asked to exceed its range of motion.
report_step	Returns the current step size.
report	Returns the current piezo position.

2.2.3 Light Source

The turret described in the camera subsection allows the insertion of a light source in its middle. The light source for the microscope is green light (500 nm) from an LED. Green light is used because Silicon is opaque to green light and this allows the camera to only view the top structure. Nevertheless, a sequence of images obtained from multiple planes contains information about out-of-plane motions.

2.2.4 SPG Module

The SPG Module is used to illuminate the moving device by turning on an LED during periods of time that correspond to a certain phase in motion. Since a MEMS device can move faster than the camera can respond, the SPG Module is needed to synchronize the phase of the motion with the LED strobes. The SPG module is

also capable of opening/closing the camera shutter at the right intervals. The SPG Module was custom designed to support various requirements which include arbitrary frequency and waveform generation, the ability to select the number of strobes per phase, etc. The details of the hardware can be found in Chapter 3. The executable "strobe" was written to operate this hardware. The arguments to the executable are as follows :

`strobe argument` *xyz xyz2*

where *argument*, *xyz*, and *xyz2* are described in Table 2.2.

Table 2.2: Arguments for the SPG Module Driver

led_on	Force the LED to be on.
led_off	Turn the LED off. However, the LED can turn on while acquiring an image.
integrate_on	Set the integration line of the camera low (Linux Only).
integrate_off	Set the integration line of the camera high (Linux Only).
frequency	Set the frequency of the output waveform to *xyz* Hz. The capacitor selection must also be updated.
divisions	Set the desired phase divisions of the sinusoid to *xyz*. The capacitor selection must also be updated.
phase	Select the phase at which we want to sample the motion. Must be less that the number of phase divisions specified.
flash	Programs the flash count register.
delay	The time delay after asserting pixstart to starting strobe pulses.
start	Initiate the process to take an image. Upon receiving this signal, pixstart is asserted for 1 Clock Cycle and the flashing starts after waiting for the specified delay count. A "Stop" character is sent to the computer system upon the completion of the process. The image can then be copied from the camera's CCD buffer. The flash count depends on the programmed value.
cap	Select the appropriate capacitor for the VCO circuitry. Refer to Table 3.4 for the appropriate selection.
wfon	Enable the output waveform.
wfoff	Turn off the output waveform. Output voltage is not necessarily at 0 Volts.
init	Resets the SPG Module. Sets the frequency to *xyz* and divisions to *xyz2*. Since this argument takes in the frequency and the division, it knows what capacitor selection to choose and does so accordingly.
image	Sets the phase to *xyz* and the flash to *xyz2*.

2.2.5 X-Y-Z Control

An automated X-Y stage provided by Prior Scientific is mounted to the base plate directly underneath the objective as shown in Figure 2-3. It is on this X-Y stage that we mount the device to be tested as seen in Figure 2-3. Unlike the pifoc, the X-Y stage uses stepper motors and allows a 0.1 μm resolution in either direction. The X-Y stage motion is limited to 4.5 x 11 inches respectively.

The stepper motors of the X-Y stage receive their control signals from Prior's Proscan H128 series motor controller. This family of motor controllers can be programmed via an RS-232 interface. In addition, it provides a cup-shaped stepper motor module with a 0.1 μm resolution that can mount on the focus knob of the MFU as seen in Figure 2-3. In this way, the MFU can be controlled via the computer system. The MFU is used to provided crude large scale motion (15 cm scanning range with 1 μm resolution) in the Z-axis, while the pifoc is used to provide a limited range (80 μm) motion with a very accurate resolution (10 nm).

The executable "stage" was written to operate this hardware. The arguments to the executable are as follows :

`stage argument` *`arg2 arg3 arg4`*

where *argument* can be any one of the following shown in Table 2.3. This table also highlights the significance of *arg2*, *arg3*, and *arg4* which have units of μm and can be either positive or negative integers.

It is important to talk about the resolution of the MFU. While the motor that interfaces to the Z-axis of the MFU has a resolution of 0.1 μm, Nikon's MFU only supports a resolution of 1 μm. Hence the large scale motion in the Z-Axis is limited to a resolution of 1 μm. The large scale motion in the Z-Axis is primarily needed to put the objective into focus and therefore resolution is not an issue. In fact, this is mandatory if the system is to be operated remotely.

Table 2.3: Arguments for the Stage Driver

set_origin	Set the internal X-Y-Z reference counter to (0,0,0). This defines the location of the *origin.*
goto_origin	Take the stage to (0,0,0), i.e the defined *origin.*
delta_x	Increment the X-axis by the number specified in *arg2* (μm). *arg2* can be a negative number.
delta_y	Increment the Y-axis by the number specified in *arg2* (μm). *arg2* can be a negative number.
delta_z	Increment the Z-axis by the number specified in *arg2* (μm). *arg2* can be a negative number.
delta_xyz	Increment the X,Y and Z position as specified by *arg2*, *arg3*, and *arg4* (μm) respectively. The increment variables can be negative.
goto_xy	Move the stage to the X,Y coordinates specified by *arg2* and *arg3* (μm) respectively.
goto_xyz	Move the stage to the X,Y,Z coordinates as specified by *arg2*, *arg3*, and *arg4* (μm) respectively.
report	Report the current X,Y,Z position.

2.3 Measuring In Plane Motion

Motion estimation can be done by taking a sequence of images at evenly spaced phases (typically 8) of the sinusoidal stimulus. The images are acquired by the camera while the sinusoidal stimulus and camera trigger are generated by the SPG Module. The X-Y stage is used to find the device to be tested and is focused by the MFU and pifoc. The following lines represents an example on how the various drivers described in this chapter can be called upon to setup the hardware prior to acquiring an image.

```
stage init  % initialize the stage
stage goto_origin  % Find a device
piezo goto 25  % Focus the image
strobe -init 2500 7 % Initialize SPG Mod. and set Freq. and Div.
strobe -flash 20  % Set the number of flashes
strobe -phase 5  % Select the phase at which to take the image
strobe -delay 2  % Delay to flashing upon receipt of start.
strobe start  % Obtain the sample image
copy_image() % Transfer the image from the camera to the hard-disk
```

The steps show that once the stage has been initialized and an appropriate ROI has been found, it can be brought into focus by adjusting the height of the pifoc. The SPG Module is programmed with the desired frequency and phase-divisions, and in this case it is 2500 Hz and 8 phase-divisions respectively. The flash count is chosen to limit the exposure of light and the phase at which the LED is to flash is selected. In this case, we selected 20 flash counts and the 6th phase[2]. The delay counter is set to some non-zero value and the start command is sent to the SPG Module to initiate the LED flashing.

The next image will most likely be at a different phase or plane, and not all of the above steps need to be repeated. The displacement between two images for a specified ROI can then be estimated directly from the images by analyzing the changes in pixel brightness. The image resolution is limited by the optics [2] to distances on the order of 550 nm. However, the displacement between two such images can be measured in the order of a few nanometers [2].

Furthermore, the sequence of images that represent the motion at various phases can be used to create a movie that shows a time displacement waveform.

2.4 Measuring Out of Plane Motion

3-D motion estimation can be done by taking pictures in various planes. All images, besides the one at the best focal plane, will have a certain degree of blurriness. In-plane motion estimation algorithms can be used to extract 3-D motion estimates from such images. More information on this can be found on page 17 in [14].

2.4.1 Computer Microvision Software

The computer and its operating system must be capable of controlling all the hardware involved, namely the pifoc's position servo controller, X-Y-Z stage motor controller, the SPG Module, and the frame grabber. With the exception of the camera, all the

[2]Refer to Table 3.3 for all such offset issues with the SPG Module.

hardware is controlled via an RS-232 interface. This is in fact quite desirable since it makes the computer microvision system OS independent. The computer used is a Pentium Pro 200 with 128 Megs of RAM. A Cyclades expansion card is installed to provide 8 extra RS-232 serial lines as dictated by the hardware requirements.

2.5 Ending Remarks

This chapter highlighted the various hardware components of a MEMS characterization system. Software was developed that allows the computer to change the state of the hardware. In essence, the computer has full control over the microscope. It can excite the device and measure the behavior of any region of interest. Furthermore, with the appropriate software, these measurements can even be done remotely.

Before proceeding on to the remote access architecture, it is appropriate at this time to present the fundamental details of the SPG Module. That will be the focus of the next chapter. The reader can choose to skip the next chapter without loss of continuity.

Chapter 3

SPG Module

The goal of this chapter is to present the underlying hardware details of the Strobe Pulse Generator (SPG) Module. This module serves as the main controller for stroboscopic illumination. Since a high level description of the SPG Module was presented in Section 2.2.4, this chapter can be skipped without loss in continuity.

3.1 Objective

The SPG Module has several objectives. First, it must allow the user to specify an arbitrary waveform shape to excite the MEMS device and not restrict the user to known forms such as sinusoid, triangular, etc. It must also not quantize the output frequency and allow the user to specify as precise a frequency as possible. Second, the SPG Module must be able to section this waveform, i.e. divide this waveform into phases. The number of sections, i.e phase-divisions, must depend upon the user specification. Third, the SPG Module must be able to turn on the LED at a user-specified phase for a number of user-specified cycles. Fourth, it must provide control signals to operate the camera. Fifth, it must be a "user friendly" module that provides acknowledge signals, status, etc.

All this will be clear once we define the architecture and establish the means to communicate to the SPG Module.

3.2 Architecture Overview

The architecture of the SPG Module comprises of three main units : Serial Unit, Singen Unit, and the VCO Unit. The Serial Unit acts as the interface for data communication between the computer system and the other units of the SPG Module, i.e. the Singen Unit and the VCO unit. It converts serial data from the computer system into parallel data and facilitates the communication protocol.

The Singen Unit is responsible for generating the stimulus that excites the MEMS device. The Singen Unit contains a specialized device that can be programmed to generate a very precise square wave, i.e. "clock". The Singen Unit allows the generation of waveforms with arbitrary shape and frequency by allowing a user to load a waveform into the RAM and to vary the rate at which the data points in the RAM are accessed. The "clock" generated from the specialized device determines the speed at which the data points in the RAM are cycled.

The VCO Unit is responsible for frequency multiplication of the arbitrary waveform (typically sinusoid) being generated by the Singen Unit. The multiplication factor depends upon the *phase-division's* requested by the user. The VCO Unit uses a Phase Lock Loop (PLL) based circuitry to generate the multiplied output which is in phase with the input. The VCO Unit uses logic that works with the reference input signal and the multiplied signal to turn the LED on, i.e. illuminate the moving MEMS device at the right phases.

3.3 Prototype Implementation

The SPG Module was implemented on a nerd kit[1]. The nerd kit was used because it provides a convenient development environment and has provision to access 4 Cypress CY7C374i CPLD's and a serial interface via the CPLD-Module[2] [15]. The Serial Unit and the Singen Unit each occupy one CPLD of the CPLD-Module. The VCO Unit

[1] A nerd kit is a sophisticated protoboard that supports a NUbus interface and is used by students at MIT that take course 6.111 which is an introductory course in digital design.

[2] The CPLD-Module plugs into the NUbus interface of the nerd kit and was designed by the author.

was put on the external proto strips due to clock and pin limitations of the CPLD-Module. The external components are wired on the space available on the protoboard. The limitation of the prototyping environment, such as availability of pins and clocks, affected the architecture of the design. This will be explained when necessary.

Figure 3-1 illustrates the interconnection among the various units. The significance of each unit will be discussed next.

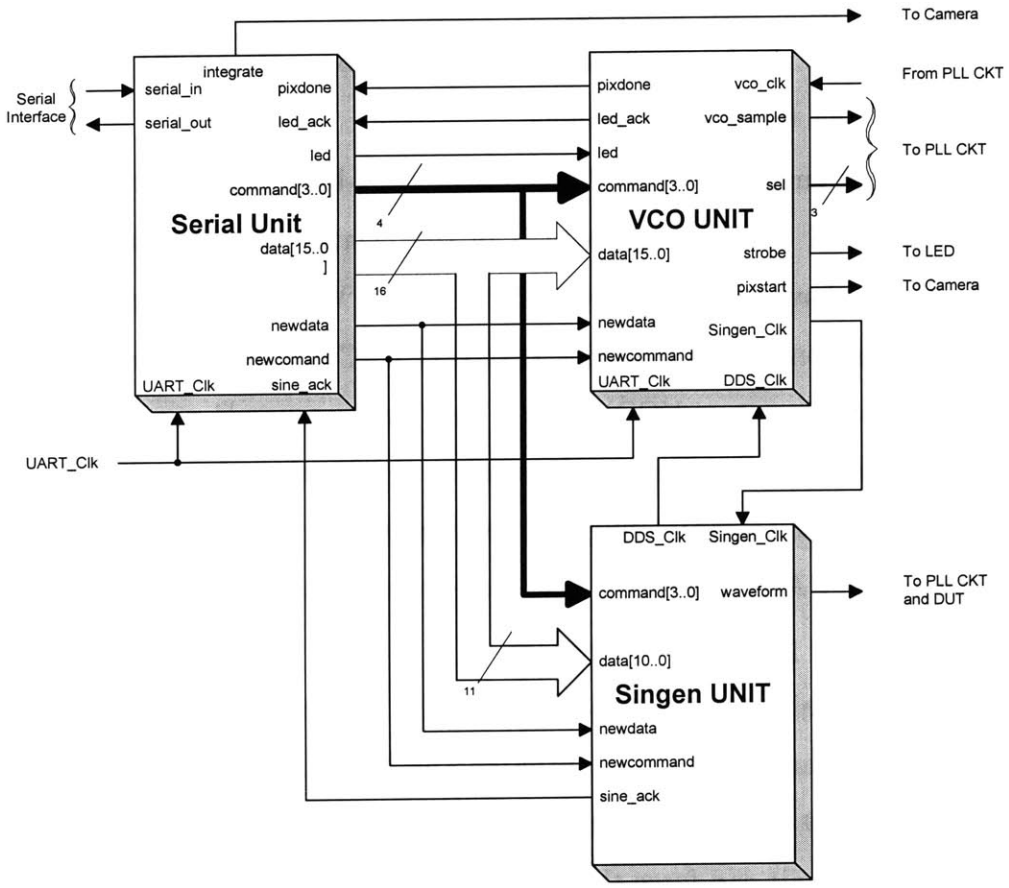

Figure 3-1: SPG Module Top-level Architecture

3.4 Serial Unit

The Serial Unit comprises of a Universal Asynchronous Receiver Transmitter (UART) that facilitates the serial transfer of data between the computer system and the SPG Module. It is desired that the SPG Module be operating system independent which is why a serial interface was chosen. The ultimate goal is that the SPG Module would

occupy a PCI[3] slot of the computer system from which it will only obtain power and ground. The functionality of the SPG Module can then be accessed via a serial interface.

The goal of the Serial Unit is to support and facilitate the established communication protocol. The communication protocol can be found in Appendix A and the reader is strongly encouraged to read it before proceeding. The protocol outlines how someone can go about accessing the various features of the SPG Module and program it. It shows the format in which data must be sent and the impact of the serial characters sent to the SPG Module.

The Serial Unit emulates a UART to convert the 7-bit serial data to a byte. This byte is then checked to see if the byte is of the "Data" category or "Command" category[4]. "Data" have to be grouped to form 16-bit "Words" and "newdata" flag is raised upon formation. If the incoming byte is of a "Command" type and is meant for the other units (Singen or VCO), then a flag "newcommand" must be raised and the appropriate value for a "Command" must be sent to the other units. The Serial Unit must wait for the corresponding unit to acknowledge the command (sine_ack or led_ack) before the UART transmits a character to inform the computer system that the "Command" was processed successfully.

The Serial Unit also asserts signals needed by the camera and the VCO Unit. These are the "led" and "integrate" outputs. The "strobe" output of the VCO unit which drives the LED can be forced high if the Serial Unit asserts the "led" signal. In this way, the Serial unit can force the LED to turn on irrespective of the VCO Unit settings. The "integrate" output is part of the control signals needed to operate the camera and is explained in Section 3.6.5.

Table 3.1 highlights the commands processed by the Serial Unit. The Serial Unit shown in Figure 3-1 is implemented in VHDL and occupies a single CPLD. The interconnections to the serial interface shown in the figure represent the interface to

[3]We selected PCI because ISA is gradually being faded away and it is very likely that newer computers will not have an ISA slot.

[4]Data is transmitted by nibbles represented by 0 - 9 and a - f or 0 - 9 and A - F. That is, case is not distinguished for hex characters. All other 7 bit characters are special command characters.

the RS-232 transceiver which converts the 12 Volt signals to TTL levels. The VHDL code can be found in Appendix D. In essence, the VHDL code can be separated into two broad categories - "Serial" and "Echo". The Echo block uses the "Serial" interface to implement the communication protocol.

Table 3.1: Commands Processed by Serial Unit

Command	Serial Char	Significance
Echo	"@"	A Null command which does nothing except the required echo of @. Used to test the RS-232 connection.
Turn_LED_On	"O"	Turn the LED on.
Turn_LED_Off	"o"	Turn the LED off (Default). The VCO unit can still turn on the LED.
Integrate_high	"i"	Set the integrate line of the camera high.
Integrate_low	"I"	Set the integrate line of the camera low.

3.5 Singen Unit

The Singen Unit uses a CPLD, RAM, DDS chip, and a D/A converter to generate a waveform with an arbitrary shape and precision frequency. It uses the CPLD, referred to as the Singen CPLD, to address and load data onto the RAM in addition to programming the DDS chip that generates a precise "clock". This clock is then used to loop through the various points of the RAM which interfaces to the D/A converter. The system architecture of the Singen Unit is shown in Figure 3-2.

3.5.1 Singen CPLD

The role of the Singen CPLD is to interface with the Serial Unit and listen to the "newdata" and "newcommand" signals and then act according to the values presented on the the 4 bit "command" bus and the 16 bit "data" bus. The architecture of the Singen CPLD is based on an FSM that polls these signals coming from the Serial Unit and asserts signals for the RAM, D/A and the DDS chip accordingly. Table 3.2 highlights the commands recognized by the Singen CPLD.

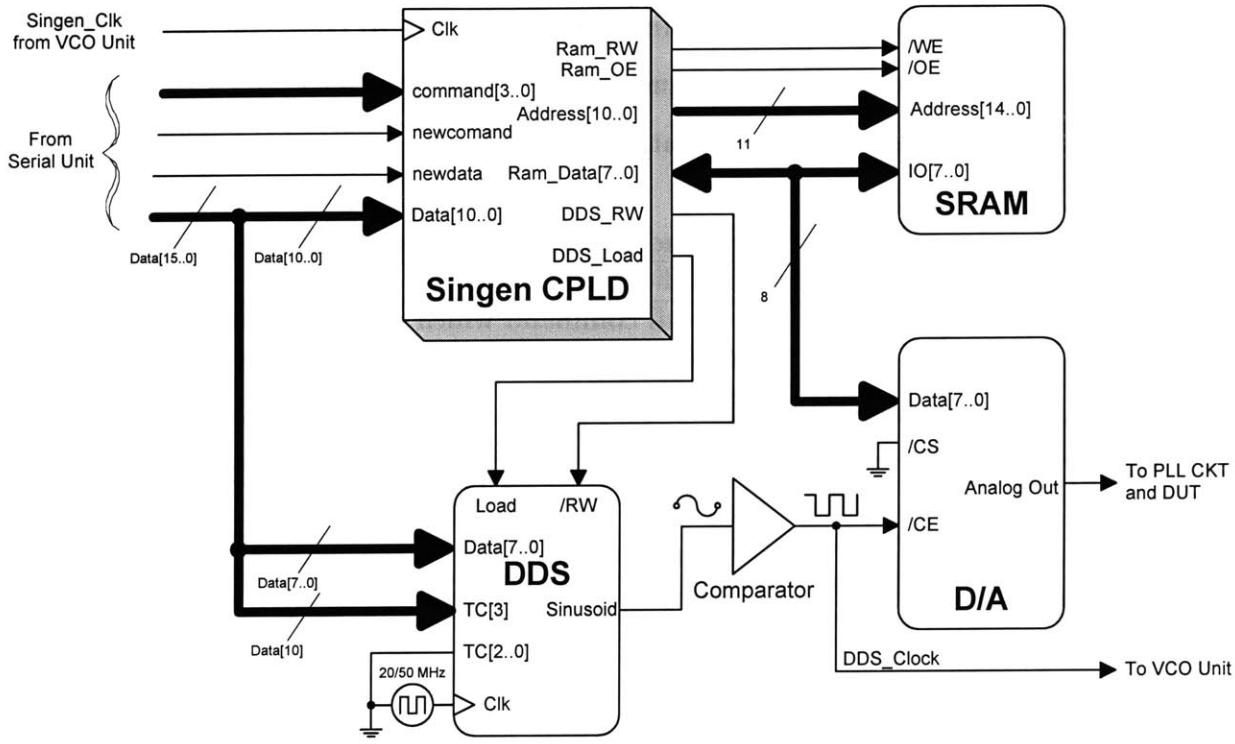

Figure 3-2: Singen Unit Architecture

3.5.2 Static RAM

The RAM used by the Singen Unit is an IDT71256SA manufactured by Integrated Device Technology, Inc. which is a 256 K (32K x 8) CMOS Static RAM [8]. It is the fastest SRAM available in the DIP package with access times in the order of 15ns.

Due to the pin limitations of the prototyping environment, only 11 bits of the Singen CPLD are used to the address the RAM[5]. Furthermore, the data bits coming from the Serial Unit are 16 bits wide, while the RAM only looks at the first 8 bits. This is because 8 bits are sufficient to produce a high quality waveform from the D/A converter. If a higher resolution D/A is needed, then the RAM can be replaced with one that is wider, or another one can be added for the higher order bits. Increasing the address and data bus width is a simple modification which requires re-compilation of the VHDL code for the Singen CPLD. This flexibility offered by programmable devices is the primary reason for their popularity in modern digital designs.

[5]The SRAM supports up to 15 bits of address lines.

Table 3.2: Singen Unit Commands

Command	Serial Char	Significance
Load_Stop_Register	"n"	This command is used to specify the value for the 11 bit "Stop Register" internal to the Singen CPLD. The "Stop Register" contains the final address of the waveform stored in the RAM. Once this command is detected via the newcommand assertion, the data is loaded from the Serial Unit data bus upon the receipt of a newdata signal.
Reset_Address_Loop	"r"	Set the RAM Address to 0.
Load_RAM_Block	"L"	This command is used to load a block of data into the RAM. The size of the block is specified by the value of the Stop Register and it is assumed that the user will first reset the RAM address before issuing this command. Upon the receipt of this command, the CPLD module will write data into the RAM each time the newdata signal is received from the Serial Unit. The address is also incremented after writing to the RAM. This process of writing and incrementing continues until the RAM has been written to the address specified by the value of the Stop Register.
Start_Address_Loop	"G"	The RAM data outputs are enabled and the RAM Address increments by one on each clock edge. The Address roles-over to zero after the value of "Stop Register" has been reached.
Load_DDS_Cmd	"l"	This command is used to program the DDS chip that generates a precise clock. Once this command is received, the action depends on the receipt of the newdata signal from the Serial Unit and the value of Data(8). If the 9th bit, i.e. Data(8), is Logic '1', then DDS_WR is asserted for 1 clock cycle, else DDS_Load is asserted for one clock cycle. The significance of these signals are explained in Section 3.5.4.

3.5.3 D/A Converter

The SPG Module uses the Analog Devices, Inc. AD558 Digital to Analog Converter which supports a microprocessor interface. The D/A converter is operated in a mode where it is always enabled and listens to data on the RAM data bus. Since it takes time for the RAM data to become valid after an address change, this can result in the D/A converter giving out inconsistent data for a short period after every address change. This problem is solved by taking advantage of the latch based architecture of the AD558. When /CE or /CS goes to Logic '1', the input data is latched into the registers and held until both /CE and /CS return to zero. Therefore, we tie clock to /CE and wire /CS to ground[6]. The timing diagram shown in Figure 3-3 assists in explaining this behavior.

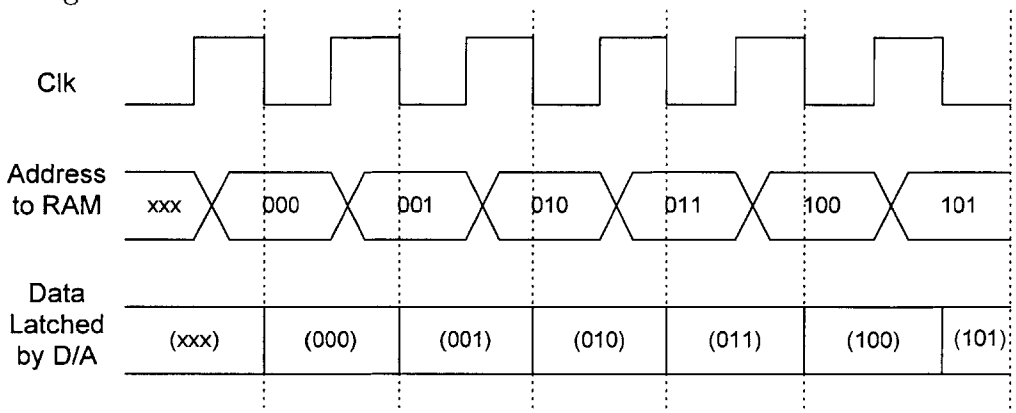

Figure 3-3: D/A Timing Diagram

3.5.4 DDS Chip

In order to generate a very precise clock, we use an AD7008 from Analog Devices, Inc. The AD7008 Direct Digital Synthesis (DDS) chip is a numerically controlled oscillator employing a 32-bit phase accumulator, sine and cosine look-up tables and a 10-bit D/A converter integrated on single CMOS chip. The DDS accepts an input clock of up to 50 MHz and generates a sinusoidal output with a frequency that can be controlled to one part in 4 billion. A comparator is used to convert the sinusoidal output into a square wave. The fidelity of the output sinusoid decreases at higher

[6]Reversing the role of /CE and /CS makes no difference.

frequencies due to the phase-accumulator architecture. Experimental results show that a decent looking sinusoid is attained when the requested output frequency is less than 1/10 the clock frequency of the DDS chip.

The DDS chip can be programmed either through an 8-bit or 16-bit parallel interface or via a serial interface. Although, the Serial Unit supplies a 16 bit data bus, we use the 8-bit parallel interface to program the DDS chip as the upper pins of the data bus work in conjunction with the Singen CPLD to assert control signals required by the DDS chip. The DDS chip supports a wide variety of functions including phase modulation. However, for our application, we only need to generate a fixed frequency which makes programming relatively trivial. Basically, we have to load data into a COMMAND register which specifies the mode of operation and the FREQ_0 register which specifies the output frequency.

The DDS chip has a 32 bit temporary register to which we write data, 8 bits at a time, by asserting the /RW line. Upon each assertion of /RW, the data is shifted left by 8 bits. Hence, 4 /RW are needed to fill up this temporary register. The next step is to assert LOAD while specifying the destination register via TC(3). This transfers data from the temporary register to the COMMAND register if TC(3) is Logic 0, else to the FREQ_0 register.

Figure 3-2 shows that TC(3) is connected to the 10th bit of the data coming from the Serial Unit while the first 8 bits serve as the parallel data to be loaded into the temporary register. The DDS_RW and DDS_Load are generated from the Singen Unit based on the value of the 9th bit as explained in Table 3.2.

Therefore, the process of programming the DDS chip first involves having the computer system send a Load_DDS_Cmd to the Singen Unit. This is followed by the computer then sending data 4 times. Recall that each data is 16 bits wide but the DDS only looks at the first 8 bits. In all 4 instances, the 9th bit should be '1' so the Singen Unit asserts a DDS_Write each time a newdata signal is received. Now that the temporary register has been filled, the next step involves transferring the data either to the COMMAND register or to the FREQ_0 register. The computer system now sends the Load_DDS_Cmd again followed by 16 bits of data. The 10-bit is tied

to TC3 so if it is Logic '0' the COMMAND register gets selected, else the FREQ_0 register is selected. This time the 9th bit is a Logic '0' so the Singen Unit asserts a DDS_Load and the data gets transferred to the selected register. Note that this time the lower order 8 bits were ignored. It is important to realize that the COMMAND register must be loaded before the FREQ_0 register. The details on this process from a software point of view can be found in the Set_DDS_Frequency() function of the driver strobe.cpp. From a users perspective, the driver takes care of the underlying details and the ordering in which the signals must be sent.

The Singen CPLD is implemented with a Finite State Machine (FSM) architecture and fitting constraints impacted the style of code. The VHDL code for the Singen CPLD can be found Appendix E.

3.5.5 Clocking Strategy

The Singen CPLD uses a fairly complex clocking strategy. It involves the multiplexing of two clock sources, namely the UART_Clk and the DDS_Clk. The UART_Clk runs the Serial Unit and VCO Unit. Hence, *all* signal communication among *all* units must be synchronized to the UART_Clk. Furthermore, this also implies that the Singen CPLD program the DDS chip using the UART_Clk. However, when the time comes to do the address looping, i.e. play out the waveform, we would like to use the DDS_Clk.

The switching of clocks is tricky and requires absolute precision to avoid any glitches on the clock line. Figure 3-4 shows the multiplexor design where signal 'A' is synchronized with the UART_Clk and 'B' is synchronized with the DDS_Clk. Furthermore, we must ensure that 'A' and 'B' are never asserted at the same time.

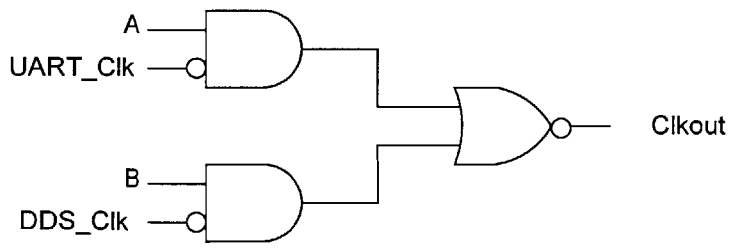

Figure 3-4: Clock Multiplexing Circuitry to generate Singen Clock

An FSM, shown in Figure 3-5, is devised to ensure that 'A' and 'B' are never asserted simultaneously and to provide a smooth transition whenever it is time to switch. Since the FSM requires 2 clock sources, the prototype limitations force us to implement this module on the VCO CPLD. This is because the VCO CPLD is external and doesn't have all 4 of its clocks tied together.

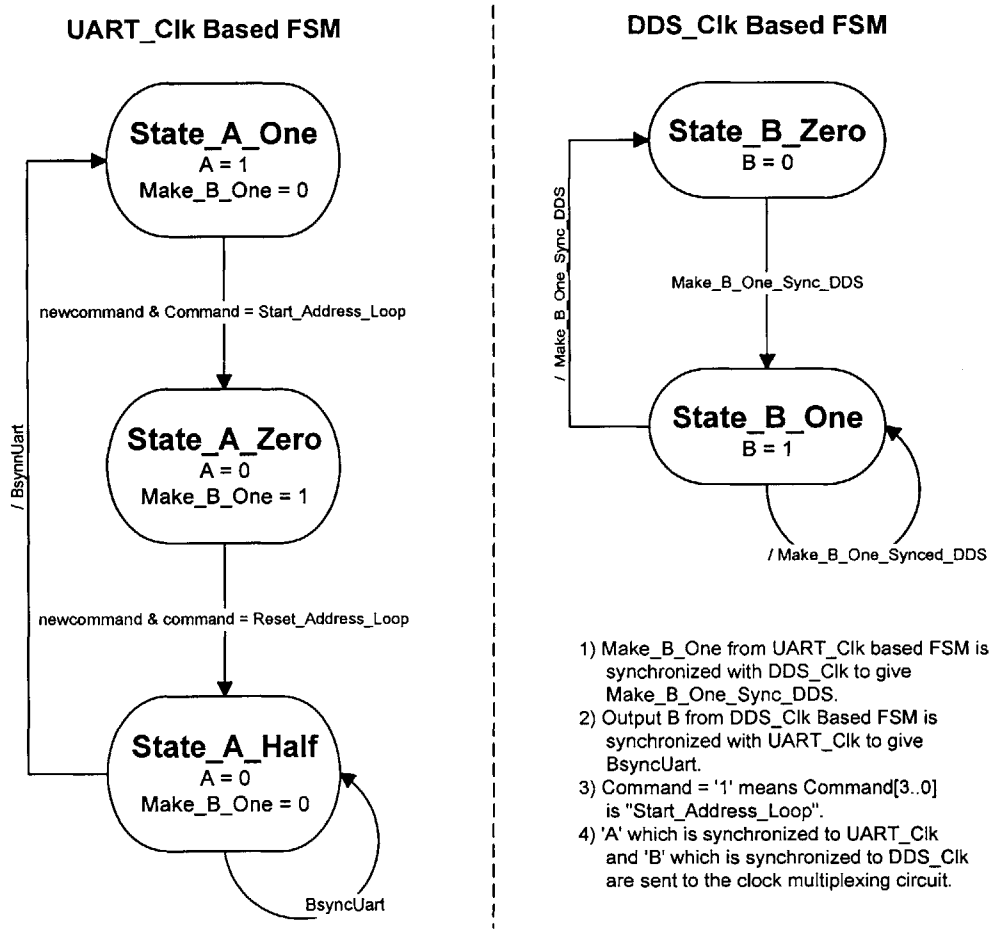

Figure 3-5: FSM used to Multiplex the Singen Unit Clock

Figure 3-6 shows a timing diagram of a simulation to show the process in which the FSM transitions the clocks. As a simulation simplification, the command signal when high means that the command is Start_Address_Loop else it is Reset_Address_Loop. It is important not to forget the assertion of the Sine_Ack signal by the Singen CPLD upon the receipt of newcommand. Once this signal is asserted by the Singen CPLD, the Serial Unit will de-assert newcommand. We must ensure that in such a situation there will be no timing problems. Recall that the assertion and de-assertion of the

newcommand signal from the Serial Unit are done on the UART_Clk. The FSM also checks for the newcommand on the UART_Clk, so it is bound to catch it. If the Singen CPLD is running on DDS_Clk and asserts Sine_Ack, newcommand will not be de-asserted till the next UART_Clk event. Furthermore, the Singen Unit keeps asserting the Sine_Ack signal till newcommand returns to Logic '0'. Hence, the assertion of Sine_Ack signal to the Serial Unit is compatible with this clock multiplexing scheme.

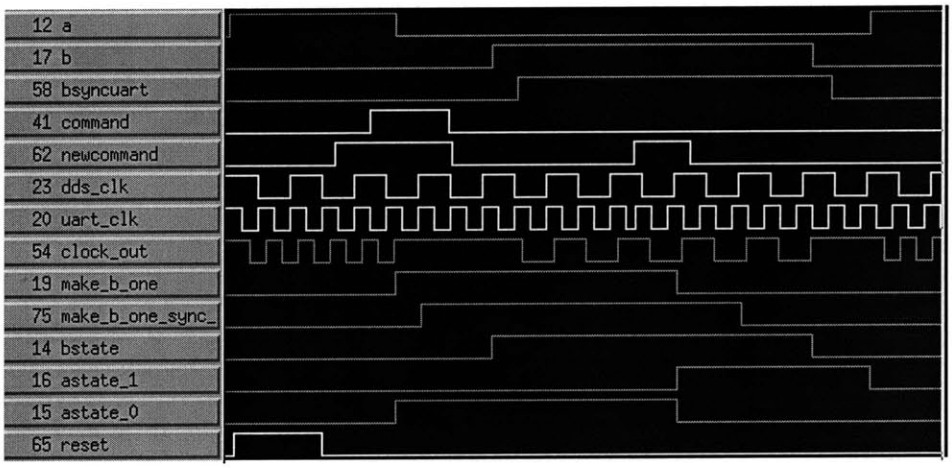

Figure 3-6: FSM Timing Simulation

3.6 VCO Unit

The previous section described the Singen Unit which provided the stimulus to the MEMS device. The MEMS device responds to this electrical stimulus with mechanical motion. This motion, which can be non-linear, is however periodic in nature. That is, a periodic input waveform will result in a periodic response/motion from the MEMS device. The goal of the VCO Unit is to use stroboscopic illumination to take advantage of this periodicity and capture this motion through the camera. The architecture of the VCO unit is shown in Figure 3-7. In this section, the functionality of the CPLD and the external components will be explained. However, the reader is encouraged to reference Appendix F for the VHDL implementation of the VCO CPLD.

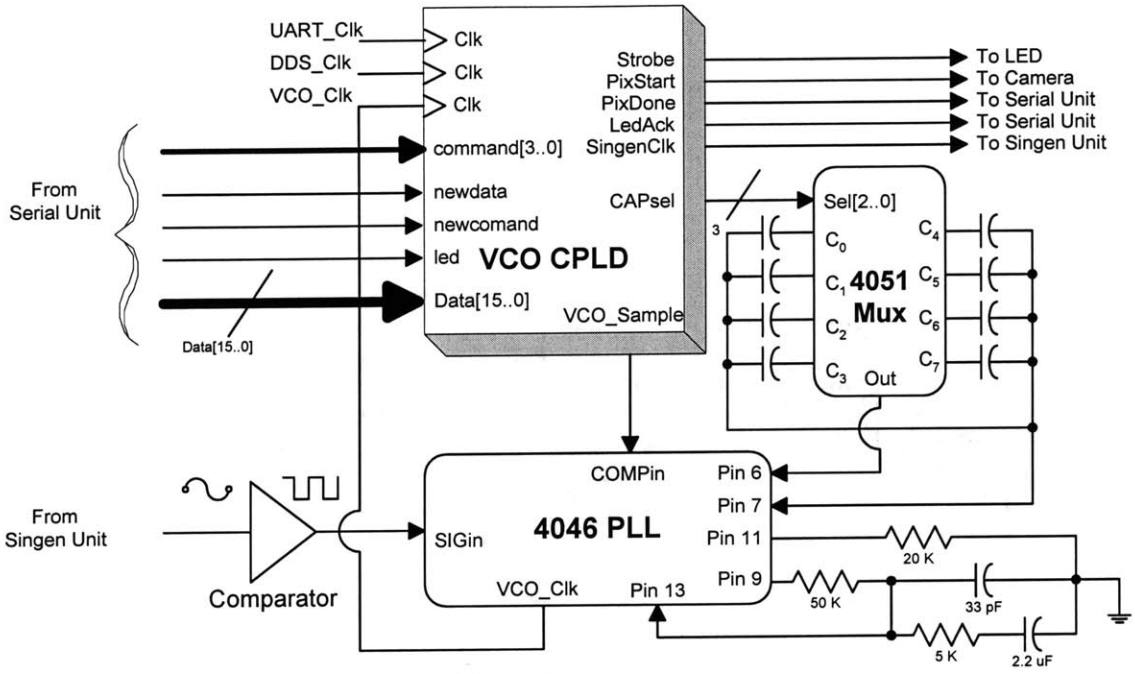

Figure 3-7: VCO Unit Architecture

3.6.1 Phase and Phase-Divisions

The response of a MEMS device is generally characterized with respect to the *phase* of the applied stimulus. A periodic waveform can be partitioned into multiple sections, where each section of time is considered a *phase*. The total number of sections in one cycle, i.e the number of possible phases, is defined as *phase-divisions*. Figure 3-8 shows a sinusoid with 4 phase-divisions. The region corresponding to the third phase-division has been highlighted.

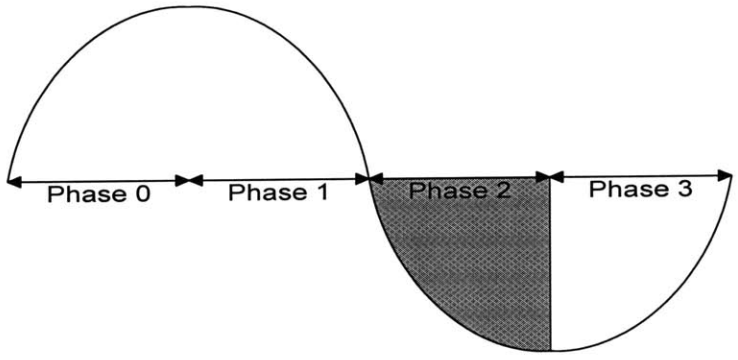

Figure 3-8: Illustration of Phase Correspondence to Stimulus

The partitioning of the stimulus waveform into phases is accomplished using a Phase Lock Loop (PLL) based circuitry that does frequency multiplication. The

objective is to produce a digital waveform that is in phase with the sinusoid exciting the MEMS device, but has a frequency that is multiplied by the number of *phase-divisions*. A PLL chip serves as an excellent means to accomplish such a task. There have been several books written on this subject and the reader can refer to any one them for details [7]. A PLL can be wired as shown in Figure 3-9 to do frequency multiplication. In essence, a frequency divider is placed in the feedback loop of the PLL. The PLL chip ensures that the two input signals will be in phase which requires the PLL chip to produce an output that is "n" times the input frequency. Here, "n" is the frequency division ratio which is achieved using a digital counter.

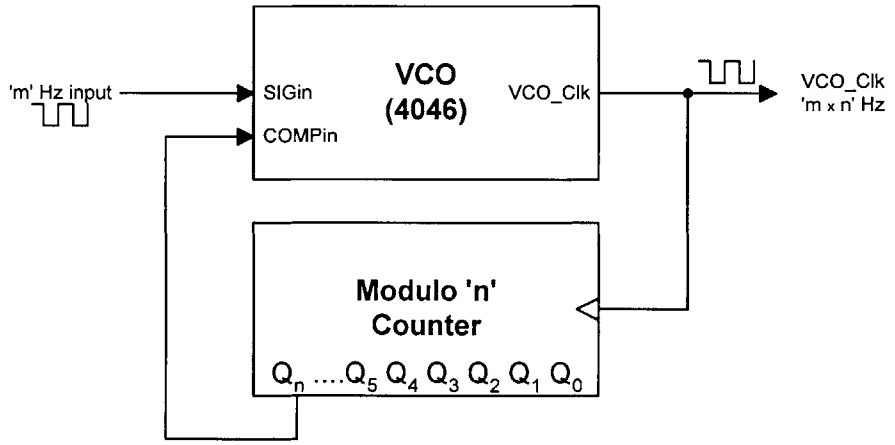

Figure 3-9: PLL Configuration for Frequency Multiplication

Figure 3-7 show that the stimulus from the Singen Unit is converted into a square wave which serves as the first input to the PLL chip. The output of the PLL chip, VCO_Clk, is sent to the VCO CPLD for frequency division to produce the second input to the PLL chip. Figure 3-10 illustrates the output for 4 *phase-divisions*, i.e. a frequency multiplication ratio of 4.

3.6.2 Phase Counter

Since the VCO CPLD has knowledge of VCO_Clk and controls the frequency division counter, it knows the correspondence of every rising-edge of VCO_Clk to the *phase* number[7]. This is labeled in Figure 3-10. Hence, a user can specify a phase number

[7]See the VHDL code and block diagrams in Appendix F for the implementation.

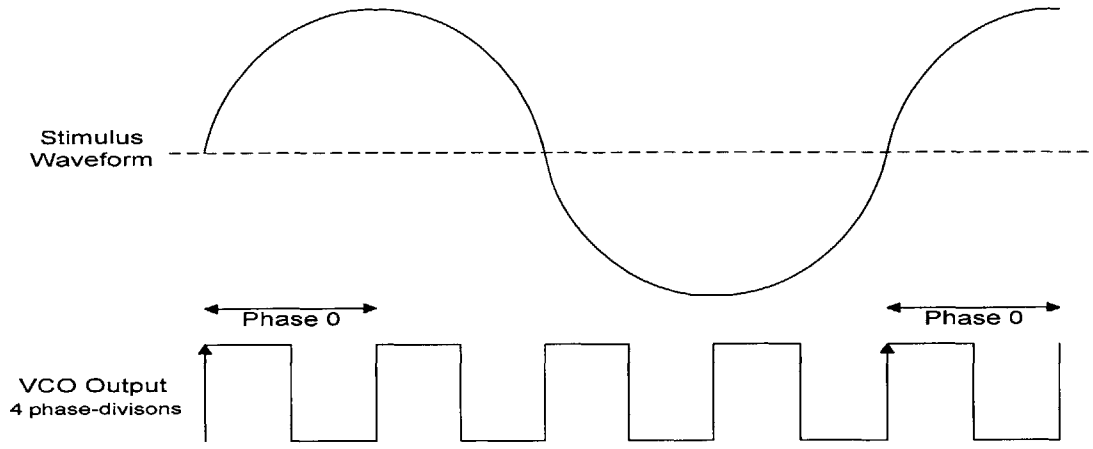

Figure 3-10: Stimulus Waveform versus PLL Output

and the VCO CPLD will be able to identify that in a stream of VCO_Clk's.

This knowledge is essential because when we analyze motion, i.e take an image, we do so one phase at a time. Thus, we must be able to select the phase of the stimulus at which we want the LED to illuminate. The details of LED illumination will be postponed till Section 3.6.4.

3.6.3 Strobe Counter

When the time comes to acquire an image, the VCO CPLD can turn on the LED corresponding to a particular phase of interest. However, the principles of imaging are based on the amount of light that gets exposed onto the CCD of the camera. The VCO CPLD provides a user programmable solution to the problem by making provision for a Strobe Counter. Upon a request for a picture, the LED turns on for a full VCO_Clk cycle and it does this for the number of times specified by the Strobe Counter.

This is the essence of stroboscopic illumination. We are taking advantage of the periodicity of the response and stimulus, to acquire the same image and multiple instances of time.

3.6.4 Image Acquisition

Table 3.3 highlights the serial commands of the SPG Module that are processed by the VCO Unit. Once the stimulus waveform is established, and the phase-divisions and

strobe counter values specified, it is time to acquire an image. The 'S' character when sent to the VCO unit will flash the LED at the desired phase for a number of times specified by the Strobe Counter. Upon completion, "pixdone" signal is asserted by the VCO CPLD which is processed by the Serial Unit. The Serial Unit, upon detecting the "pixdone" signal, transmits a special character to the computer system to notify completion of the image acquisition. This is important from a timing standpoint, as the computer system can now fetch the image from the camera buffer and write it to disk.

The "strobe" output of the VCO CPLD is sent to the LED via an open collector buffer (74S38 chip). When "strobe" is Logic 1, then the LED in on, else it is off. The "led" signal from the Serial Unit can be used to force the "strobe" output high irrespective of the VCO CPLD settings. This is needed for applications that control exposure via software.

Table 3.3: VCO Unit Commands

Serial Char	Significance
"M"	The number of phase-divisions.
"N"	The number of strobe pulses starting from 0 required to produce the desired exposure.
"P"	This is used to specify the phase of the strobe pulse.This number must be offset by -1. Suppose, there is an 8 phased-divisons system, 'M' should be '7'. To select the 2nd phase, 'P' should be '0' and to select the first phase, 'P' must specify '7'.
"S"	Start taking a picture.
"V"	The capacitor value selection which specifies the range of the VCO.
"W"	The time delay before starting strobe pulses. Cannot be zero.

3.6.5 Camera Support

It is important to realize that the primary role of the VCO Unit is to flash the LED. However, it is important to have the camera in the *expose*[8] mode before the LED begins to flash. To minimize the amount of dark current, it is important to minimize

[8]The word *expose* has been used in a very general sense. In analog camera cameras, this may correspond to opening the shutter, or in a digital camera flushing the CCD, and so on.

unnecessary exposure. Furthermore, to simplify timing and maximize throughput, it is desirable to take care of all timing sensitive issues in hardware. The "pixdone" signal described in the previous section can notify the completion of the imaging process. However, the issue of *exposing* the camera before the LED begins to flash needs to be addressed.

Chapter 2 showed that the MEMS characterization system requires a proper camera be mounted based on the operating system in use. Since the cameras are supplied by different vendors, they have different techniques of controlling the *exposure* process. The Linux camera supports a level-sensitive trigger via by the assertion of the "integrate" line. As long as the line is pulled low, the camera CCD is exposed to the incoming light. Once the "integrate" line is pulled high, the exposure process is over. Hence, for the Linux camera, the integrate line is pulled low prior to sending the 'S' character to the SPG Module. Once a "pixdone" is received from the SPG Module, the integrate line can then be pulled high.

However, the NT camera requires an edged pulse to specify the start of the *exposing* event. The amount of exposure time is programmed in software prior to the start of the event. In order to support camera types that require an edge triggered event, the VCO CPLD provides a "pixstart" signal. The pixstart signal is raised for 1 UART_Clk cycle upon the receipt of the 'S' char from the computer system. This triggers the camera and the VCO unit then starts the process of illuminating the LED.

It is very likely that once a camera receives the edge-triggered event, a certain processing delay occurs before the camera really starts to capture the image. This delay time is specified in the camera data sheet and it is very important to take this into account. To overcome problems that may arise due to such a delay, the VCO CPLD makes provisions for a delayed image acquisition. This is done by allowing a programmable delay from the time the *pixstart* is sent to the camera to the first flash. A user can specify the value for an 8 bit Delay Counter which runs at the UART_Clk via the "W" character [9]. The Delay Counter does not have to be updated

[9]The UART_Clk is used because its value is fixed compared to the variable VCO_Clk.

by the computer system for each image acquisition process. In fact, the VCO CPLD remembers the last programmed value and reloads the counter each time an image acquisition is requested. The Delay Counter then counts down and upon reaching zero the LED illumination sequence begins.

3.6.6 Capacitor Selection

The brief introduction to the PLL circuit in Section 3.6.1 did not highlight the main components of the PLL circuitry. That is not the intention of the thesis, however we have to understand one very important component of the PLL chip for it to operate properly. A PLL contains a phase detector, low-pass filter, amplifier and a voltage controlled oscillator (VCO) that represents a blend of digital and analog techniques all in one package. Figure 3-11 (below) shows the three important components that make up the PLL.

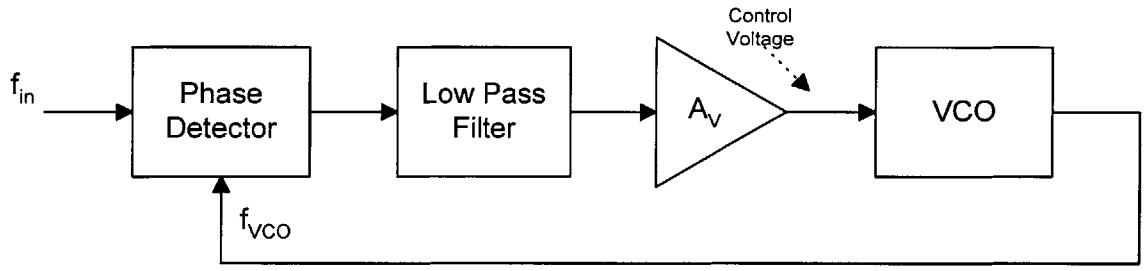

Figure 3-11: Stimulus Waveform versus PLL Output

The R and C values show in Figure 3-7 are used to specify the value for the loop filter, which directly influences the tradeoff between *settling time* and *locking range*. The *locking range* signifies the maximum output frequency that the VCO can generate in a closed loop application and settling time refers to the time it takes to *lock* after a perturbation, i.e. change in phase-divisions, etc. In typical VCO applications, the output is generally fixed or there are slight perturbations around this fixed value. Designers then work to minimize settling time and maximizing locking range.

Unlike Gigahertz applications, *settling time* is not a critical issue as given enough time the VCO will eventually settle. However, in our case, since the VCO_Clk is used to loop RAM addresses from the Singen CPLD, it can go all the way from less

than a Hertz to a couple of Megahertz. With such a large *locking range*, there is no one value of R and C that can meet the range we would like the PLL to support. We solve this problem by using an analog multiplexor (4051 chip) that can alter the filter settings. The VCO CPLD makes provision by allowing the user to select an appropriate capacitor as shown in Figure 3-7.

Since the VCO_Clk frequency is a product of the frequency of the stimulus and the desired phase-divisions, the multiplexor selection must be updated each time there is a change to one of these parameters. The various R and C values were found experimentally and are labeled in Figure 3-7. The analog multiplexor values can be selected based on Table 3.4 shown below. While the multiplexor supports up to 8 values, only 5 were needed for the range of interest.

Table 3.4: PLL Capacitor Selection

(Stimulus Freq x Phase-divisions)	Binary Selection	Capacitance
< 8000	000	0.5 nF
$8000 - 72000$	001	1 nF
$72000 - 176000$	010	5 nF
$176000 - 720000$	011	10 nF
$720000 - 1200000$	100	100 nF

3.7 Printed Circuit Board

Figure 3-12 shows the prototype implementation of the SPG Module on the nerd kit. The card plugged in the back of the kit is the CPLD-Module which houses the Serial and Singen CPLD's and provides an RS-232 interface. The schematics of the SPG Module were drawn with DesignWorks and are shown in Appendix C. The next step is to transfer the SPG Module implementation on a nerd kit to a card that occupies a PCI interface. The serial programming interface remains intact as the the SPG module only takes power, ground, and reset from the PCI interface. The extracted netlist can be sent to a facility for PCB layout and fabrication.

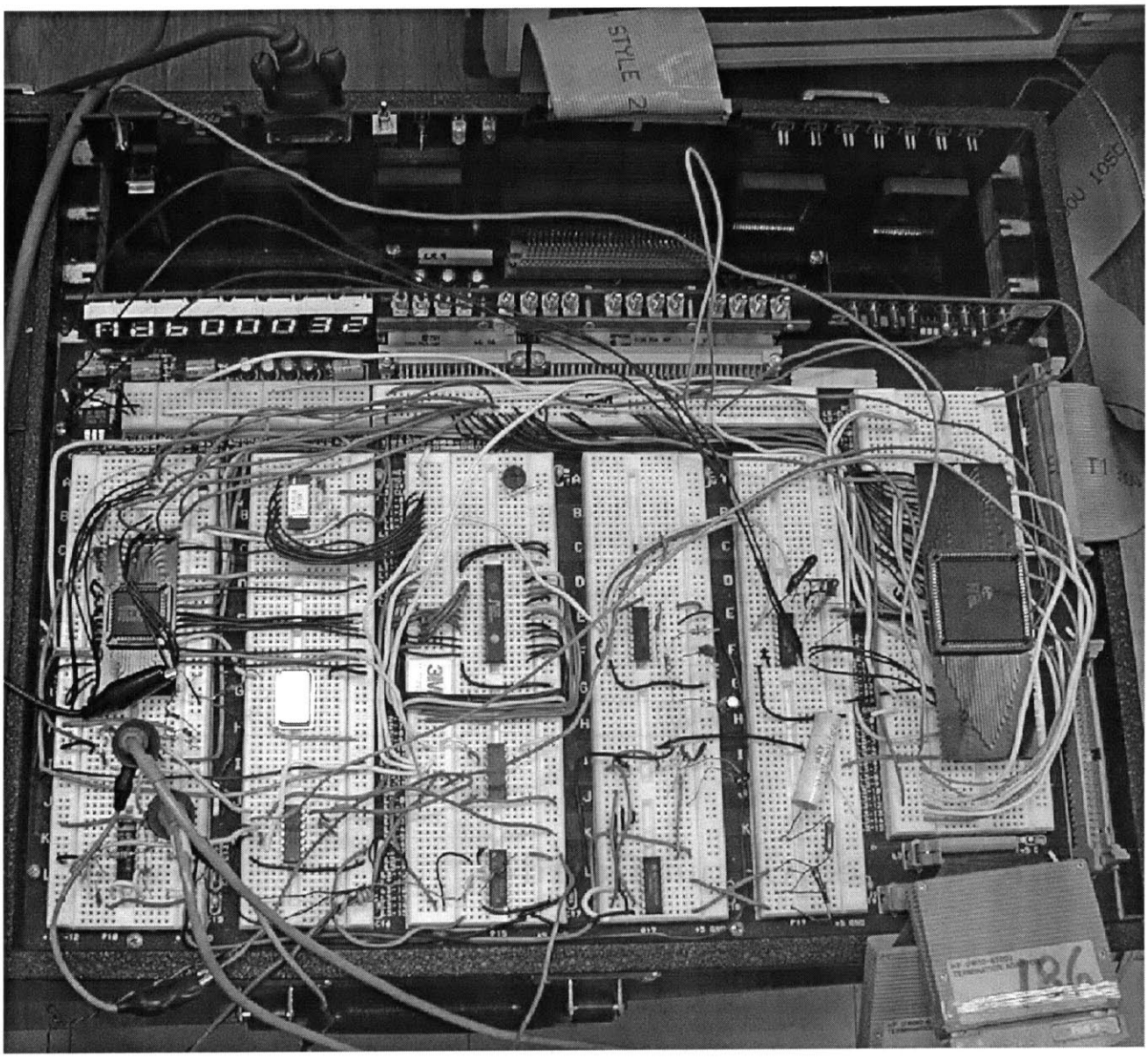

Figure 3-12: Image of SPG Module Prototype

Chapter 4

System Architecture for Remote Access

The previous chapters highlighted the various hardware components of the microvision system. It also showed the various software drivers that can be used to control the hardware and obtain an image of a moving device. *Obtaining an image of a moving MEMS device at a particular focal plane, frequency, phase, and light intensity is the most basic building block of a microvision system.* This basic building block can be extended into a script that can take a sequence of images and then run the motion estimation algorithm on a specified region of interest.

It is very desirable to raise the level of abstraction above the level of a command line interface. This calls for a user interface that shields the underlying details of the software and hardware involved. There are also benefits in desiring to use the system remotely as supposed to the user operating this interface locally. These benefits were discussed in Section 1.3.

The system architecture is based upon a server/client relationship. The server is the means through which somebody using the system can have access to the hardware and software installed on the system. This method of interaction allows the server to control access levels as well as abstract away from the user those details of the system that the user does not need to know about.

4.1 Server Overview

The design of the computer microvision system lends itself to being a remotely accessible system. This is because the computer, acting as the controller, can be thought of as the server for all of the data and for all of the parameters of the system. Figure 4-1 below shows the system architecture for the server.

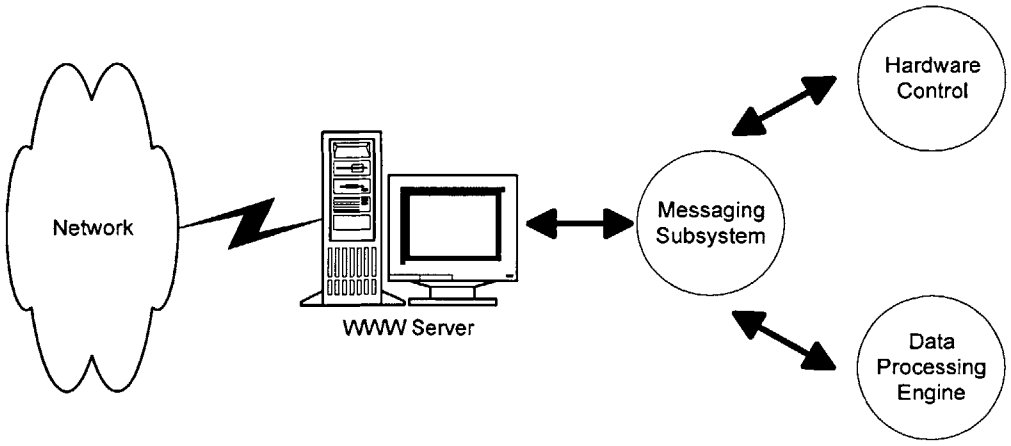

Figure 4-1: System Architecture for the Server

There are two main modules of software that enable the computer to act as a server for the MEMS Characterization System. First of all, there is a web server running on top of the operating system. This server acts as the network server for the clients. It allows the client to interact with the server using the standard HTTP protocol. The server software that has been used is Apache's web server. The reason for using Apache's web server is because it the most popular web server[1] and supports both the NT and Linux operating systems. Another reason for using Apache is because it supports the use of Java programs called servlets. It is these servlets that overcome the inability of the standard HTTP interactions to deal with state. Servlets are Java classes that run on the server side instead of being downloaded over a network the way a standard applet is downloaded and run locally. Thus, they extend the server's ability to accomplish tasks which can be initiated remotely.

[1]Apache has been the most popular webserver on the Internet since the April of 1996 according to the Netcraft Web Server Survey which can be found at http://www.netcraft.com/survey.

The second software module is the sum of Java classes that make up the functionality of the MEMS characterization system. They handle the remote login of a client and the subsequent interaction between client and server. For details on how these modules are set up, see Jared Cottrell's thesis entitled "Server Architecture for MEMS Characterization System" [1]. There are also modules that take care of interclient communication as well as control and access of information for each client connected to the server.

4.2 Client Overview

The client software consists of multiple modules written in Java and is intended to be run remotely through a web browser. For more details on the various modules, refer to Erik Pedersen's thesis, "User Interface for MEMS Characterization System" [12]. These modules were used to create a Graphical User Interface (GUI) which is described in Chapter 8. With this interface, the user is able to remotely operate the computer microvision system. The link between the client and the server is a messaging protocol which is described in the next section.

4.3 Messaging Protocol

The means through which the server and client interact is fundamentally based on the HTTP protocol used extensively in the World Wide Web. An advantage of this is that we can use an off-the-shelf web server to implement our messaging protocol. For a more detailed description of this protocol, see the HTTP protocol specification [11]. There are certain types of requests that a client is allowed to make using the HTTP protocol. The most often used requests are those made by web browsers when they request web pages. This request is called GET where there are parameters outlined in the header of the request. There is also a request called POST which is similar to a GET request except that there is more flexibility in the length of information attached to it. It is through a POST that a client is expected to send information to

the server.

The idea used in communicating between the client and a server is a communication protocol which spells out the rules for the format and transmission of data. The messaging protocol is the glue that binds the entire MEMS characterization system together. Again, it is the only interface through which the various clients and the server in the system can "talk".

The messaging protocol can be understood in terms of how the server and client determine how to handle the messages. The actual content of the messages sent between client and server is plain text with well defined formatting. When the text is sent by the either the client or the server, it is actually URL encoded to preserve all formatting. This also allows the corresponding agents to send data of any format. Specifically, it allows the transfer of any data including graphical data. Each message sent must contain a line of text consisting of the term "COMMAND = (command)". This command line is how the server and the client decide how to handle the messages. The messaging protocol also includes the transmission of session ID's and several other parameters. The details of the messaging protocol can be found in Chapter 3 of [1].

Figure 4-1, shows that the messaging subsystem sits directly behind the web-server and implements the messaging protocol. On both the client and the server, there are "handlers" that act upon the received message. As seen, there are two main categories of handlers. They are the *hardware control handlers* and the *data processing engine.* For example, a message requesting to move the stage would be processed by the SET-STAGE-HANDLER.

4.4 Managing Sessions

A client connects to the server by pointing a web browser[2] to the server's URL, i.e http://stage.mit.edu/UI/index-new.html. Upon connecting to the server, Java applets are automatically downloaded to the client's machine and get executed locally on the client's machine. The downloaded applet is the GUI to remotely operate the

[2]Our system works with either Netscape or Internet Explorer.

microvision system. The GUI operator is shielded from the underlying details of the messages that are exchanged back and forth between the client and server. The details of the GUI are presented in Chapter 8.

For various reasons, the server needs to keep track of the clients that are currently connected to it. It must also be able to differentiate between clients since there may be multiple users logged in at the same time. Furthermore, while all clients can query the state of the system (stage settings, etc.), only one user can have control over the hardware at any given time. Since HTTP is a stateless protocol, the server forces each client to log in before they can send any other messages. When a new client logs in, the server assigns it a unique session identification number. All subsequent messages between the client and server include this session ID in a field called SESSION-ID as specified in the messaging protocol. Once a client has logged in, the server keeps a reference to its session ID until either the client explicitly logs out or the servlet restarts.

It is important to emphasize, and it will be clear in the upcoming chapters, that the GUI operator has no idea what his/her session ID is. The applet keeps track of all the underlying details when communicating with the server.

4.5 Software Development Environment

It is important to note that there are two different sets of Java software. One that runs on the server i.e servlets, and the other that gets executed on the client, i.e. applets. They do not depend on each other and have no shared libraries. They only communicate via the specification set forth in the communication protocol. Furthermore, the first time a client connects to the server, the client is presented with a list of messages supported by the server. From time to time, it is required to expand the protocol. For example, send 3 arguments with the command rather than 2. Since this is a change it protocol, both the server software and client software must be recompiled to conform to the new specifications.

4.6 Polling

Sometimes the server needs to send a message to the client even though the client has not made a direct request for information. Because we use HTTP as a basis for the messaging protocol and HTTP requires that the client initiate all communications, the client is required to poll the server every several seconds and check for messages. In this case, the message from the client is CHECK-MESSAGE [1] while the handler at the servlet end is CHECK-MESSAGE-HANDLER. If there are any messages waiting for the client, the server will send them back in response to the poll.

Polling is also used to implement session timeouts. If the server sees that the client has not polled for a while, it will assume that the client is no longer active. It is also desirable for one client to send a message to another client in order to facilitate a collaborative session. Using the polling mechanism, a "chat" like program can be launched for clients to communicate among each other.

4.7 Chapter Summary

This chapter showed that there are two different sets of software that rely on the communication protocol. The communication is done via messages which adheres to a protocol understood by both sets of software. This also means that the client software can only be developed after the server protocol has been established. Similarly, it can be argued that the server protocol will depend on the messages that the client wants to send. This is a chicken and egg problem and I'll break the loop.

The strategy will be to first lay out the foundation of messages and the arguments encoded in them. With this information, the client messaging system can then call up the executables in whatever way it feels appropriate. The next two chapters will therefore describe the messaging handlers that were developed as part of this research, i.e. those that have anything to do with hardware. As we will see, this strategy simplifies the explanation of the client interface.

Chapter 5

Hardware Control Handlers

A majority of the work carried out in this research on the server side software involved the creation of messages that affect the status of the hardware. These messages and their corresponding handlers are part of the Hardware Control Module and serve two related functions. First, the Hardware Control Module can call up upon an executable program to change the state of the hardware. Second, the Hardware Control Module keeps track of the hardware status and updates the poll messages for other clients to inform them of any changes.

This chapter describes the hardware messaging format currently being used. Understanding these messages is essential to understanding how the client is able to remotely operate the computer microvision system.

5.1 Description

A handler and a message are closely related. According to the established protocol, every message must include in its header, arguments for COMMAND and SESSION-ID. Once a message has been received by the messaging subsystem, the COMMAND is decoded and the appropriate handler is executed. A handler is thus Java code which may include a call to a C-program to change the hardware state. The handlers also update the global variables with the latest hardware settings, and queue messages for other clients with updated hardware settings which they will pickup upon polling.

In the above description, certain details of the messaging subsystem were glossed over to maintain simplicity. For instance, while there may be many users logged onto the system, only one client may have control at any given time. A client requests controls with the GET-CONTROL message and cedes control with the CEDE-CONTROL message [1]. The messaging subsystem checks if the client requesting a change of hardware state has control before forwarding the request to the appropriate hardware handler. If a client does not have control, the server will return the CONTROL-ERROR message instead of the normal response. The messaging subsystem, among other things, also checks for valid SESSION-ID's and any formatting errors.

Since the handlers may include calls to C-programs to change the state of the hardware, it is almost a necessity to send some arguments along with the executable call. This can be accomplished by adding fields in the header which can be decoded by the handler. The following sections will describe the various hardware handlers and the corresponding messaging format that allows the client to remotely change the state of the hardware.

5.1.1 X-Y-Z Stage Handler

The following is the format of the SET-STAGE-FOCUS Message.

Table 5.1: Structure of the SET-STAGE-FOCUS Message

Client Request	COMMAND = SET-STAGE-FOCUS SESSION-ID = *sessionID* STAGECOMMAND = *stagecommand* XPOS = *xpos* YPOS = *ypos* ZPOS = *zpos*
Server Response	COMMAND = ACK SESSION-ID = *sessionID* MESSAGE = SET-STAGE-FOCUS X-TRANSLATION = *xpos* Y-TRANSLATION = *ypos* Z-TRANSLATION = *zpos*

When sent by a client, the SET-STAGE-FOCUS message tells the server to update

the position of the X-Y-Z stage. The message contains values for STAGECOMMAND, XPOS, YPOS, and ZPOS that will be used by the handler as arguments to a C-program. The following is the driver call with the appropriate arguments to the binary executable, "stage". It shows how the arguments can be passed from the client interface to be executed by the server.

```
if (stagecommand.equals(''goto_xyz'')) then
 execute (stage stagecommand xpos ypos zpos)
else
 execute (stage stagecommand)
end if
```

The significance of the arguments to this executable program can be found in Table 2.3 in Section 2.2.5. The *if* portion of the code is used to create motion after decoding the variables sent with the message. The *else* part is used to execute other commands such as "GOTO_ORIGIN" and "SET_ORIGIN". It is easy to realize that the *else* portion of the code is much more generic and can also accommodate what is being executed by the *if* part, and hence the *if* part can be eliminated. This is because the "*stagecommand*" argument can be a series of chars including "space". For example, "DELTA_XYZ 1 3 5", and even "GOTO_XYZ 1 3 5". Hence, we can incorporate knowledge of the executable arguments into the clients message rather than having the server decode the variables sent with the message. The later, generic approach, simplifies the design of the message handlers, but has some disadvantages. It is required for the server to be up-to-date with the current state of the hardware. If the message is a direct call to an executable, with no encoded variables, then the server does not know the position it just put the stage in. To solve this problem, the server will have to execute the "REPORT" command after every stage event to ask the stage for the latest position. This is a a slow process and not advised when the user wants to move the stage instantaneously, which is almost always the case. It is important to realize that the stage is a mechanical device which takes time to move as well as respond to commands.

The server response which includes the current X, Y, and Z settings is also sent to other clients to inform them of the latest stage position. They receive this information upon polling which was explained in Section 4.6.

5.1.2 Piezo Handler

When sent by a client, the SET-PIEZO messages tells the server to communicate with the pifoc controller and update the piezo position.

The following is the format of the SET-PIEZO Message.

Table 5.2: Structure of the SET-PIEZO Message

Client Request	COMMAND = SET-PIEZO SESSION-ID = *sessionID* PIEZOCOMMAND = *piezocommand* POS = *pos*
Server Response	COMMAND = ACK SESSION-ID = *sessionID* MESSAGE = SET-PIEZO POS = *pos*

The parameters from the message are extracted and executed by the server as follows :

```
execute (piezo piezocommand pos)
```

The significance of these arguments was explained earlier and can be found in Table 2.1 in Section 2.2.2. The most widely used *piezocommand* is "goto_position" which can be used to put the piezo in the position specified by *pos*. Anytime this message is received, the latest piezo position is also sent to the other clients.

5.1.3 Stroboscopic Settings Handler

The SET-STROBE message is used to establish the fundamental settings of the SPG Module. It sets the frequency of the output waveform, and the number of phase divisions needed in the analysis. The format of the message is shown in Table 5.3.

From the received parameters, the following is executed :

Table 5.3: Structure of the SET-STROBE Message

Client Request	COMMAND = SET-STROBE SESSION-ID = *sessionID* FREQUENCY = *frequency* DIVISONS = *divisions*
Server Response	COMMAND = ACK SESSION-ID = *sessionID* MESSAGE = SET-STROBE FREQUENCY = *frequency* DIVISONS = *divisions*

```
execute (strobe init frequency divisions)
```

The significance of the arguments to this executable program can be found in Table 2.2 in Section 2.2.4.

5.1.4 Image Settings Handler

The UPDATE-IMAGE-SETTINGS message is used to establish the image settings of the SPG Module. The format of the message is as follows :

Table 5.4: Structure of the UPDATE-IMAGE-SETTINGS Message

Client Request	COMMAND = UPDATE-IMAGE-SETTINGS SESSION-ID = *sessionID* PHASE = *phase* LEDONTIME = *ledontime*
Server Response	COMMAND = ACK SESSION-ID = *sessionID* MESSAGE = UPDATE-IMAGE-SETTINGS PHASE = *phase* LEDONTIME = *ledontime*

The PHASE parameter specifies the phase at which the picture should be taken. The *ledontime* is used to determine the exposure time of the camera. Basically, the camera's shutter is opened, and during a time interval that corresponds to a selected phase, we turn on the light source, i.e the LED. Hence, one flash of light per waveform cycle is sent to the camera. The amount of light per waveform cycle depends on the frequency of the sinusoid and the number of divisions. It is very likely that more than

one waveform cycle is needed to allow adequate light to be exposed onto the CCD of the camera. The *ledontime* parameter is thus used to specify the time that light should be shined on the CCD and by knowing the frequency and phase divisions, we can extract the flash count. The following C-code shows how the flash count is extracted and how the executable is called :

```
flashcount = ledontime * divisions * frequency;
execute (strobe -image phase flashcount)
```

Note that the *ledontime*, which is supplied by the client, is expected to have units of seconds. This is then converted to an integer called *flashcount* which represents the number of strobe flashes as required by the executable syntax. The parameters required for calculating flashcount, i.e. divisions and frequency, are extracted from the last known values of the server state. The details of this executable were explained in Table 2.2 in Section 2.2.4.

5.1.5 Obtain Sample Image Handler

The purpose of the OBTAIN-SAMPLE-IMAGE message is to expose the camera and trigger the SPG Module to obtain a sample image. The format of the message is as follows :

Table 5.5: Structure of the OBTAIN-SAMPLE-IMAGE Message

Client Request	COMMAND = OBTAIN-SAMPLE-IMAGE SESSION-ID = *sessionID*
Server Response	COMMAND = ACK SESSION-ID = *sessionID* MESSAGE = OBTAIN-SAMPLE-IMAGE

As seen from the server response, the server only returns an acknowledge signal. It does not return any image data with the reply. This is because it is more efficient for the server to simply return the URL of the image and have the client obtain it via the GET method. This procedure also simplifies the processing of the data at the client end.

It is interesting to observe that in this case we don't even return a URL. This is because we have fixed the URL to http://stage.mit.edu/sampleimage.gif. The executable used to obtain the image simply overwrites this files and the clients re-reads the URL once an ACK is received from the server. Since Java has the tendency to cache images, certain programming hurdles had to be overcome.

The executable used to obtain the image is as follows.

```
execute ( /usr/local/cm/bin/j_pixstart )
```

"j-pixstart" is a program used to send a "start" signal to the SPG Module described in Table 2.2 after activating the camera. It then copies the image from the camera buffer to the hard-disk once a "Stop" is received from the SPG Module. The image is then converted to GIF format. The details of this program can only be understood after we have studied the command module. We will return to this executable in Section 6.1.3.

It is also worth noticing that no ROI coordinates were supplied to the camera. The camera has a display size of about 1200 X 1600 pixels, and hence a full size image was taken. The resulting image size is about 1 Megabytes.

5.2 Single Message Approach

It is logical to ask why not have one message to accommodate SET-STROBE, UPDATE-IMAGE-SETTINGS, and OBTAIN-SAMPLE-IMAGE. After all, we can first set the SPG Module's parameters and then trigger the camera. This proposal is certainly viable, but there are advantages to doing this with multiple messages. First off, SET-STROBE message has an affect on the VCO circuitry described in Chapter 3, which requires updating the capacitor selection and has a certain settling time associated with it. In addition, these parameters are not changed very often, and it is intuitive for the user to set them and not have to worry about changing them very often. The next frequent occurrence is the change in phase and ledontime time. This only affects the hardware registers and thus happens instantaneously from a circuit point of view. The final trigger is the event when the user specifies he/she wants to obtain a sample

picture.

Having multiple messages also makes the system more manageable and expandable. After all, there are alternatives to using the SPG Module. Lastly, the transfer of messages between client and server is not a serious bottleneck in the operation of the MEMS characterization system. Hence, there is no benefit in designing a system with minimal message communication.

5.3 Chapter Summary

This chapter highlighted the basic handlers that can be used to update the hardware settings. It showed the message protocol through which the client can move the stage and set the piezo position. After setting the various parameters of the SPG Module, the client can then obtain a full-size image of the moving device. As mentioned earlier, the ability to obtain a sample image is the most fundamental building block of the MEMS characterization system. The next chapter will highlight some advanced handlers that extend the usability of the computer microvision system.

Chapter 6

Command Module and Associated Handlers

The previous chapter explained the concept of a handler and presented some basic handlers needed to obtain a sample image. In this chapter, the command module interface will be discussed along with the architecture used for data set gathering and slow motion analysis. An understanding of these handlers is essential to understanding the advanced features of the client's user interface and being able to launch an experiment to characterize the motion of a MEMS device.

6.1 Command Module

The handlers discussed in this chapter call upon executables that utilize the command module interface. The command module is a fairly-complicated set of C software written by Michael McIlrath that only works for the Linux OS. It serves as a script that can launch an experiment, i.e take a series of pictures, and organize all the data. So far in our understanding, we mentioned that the ability to take a single picture is the fundamental building block of the MEMS characterization system. The command module interface serves as an extension to that by acting as a script that can take a sequence of images.

The command module takes in as arguments frequency(s), piezo position(s), etc.

and simply loops through all combinations of possible images. The command module works at much higher level of abstraction then a simple script by allowing the dynamic loading of software modules. This implies that changing the hardware, for example the camera type, does not require any software compilation. A simple change in a text-based configuration file is sufficient. The command module interface thus requires that a generic function call be made to a camera driver, piezo driver, and strobe module driver when sequencing through the set of images it is asked to take. A software wrapper is thus written around the driver provided by the vendor to confirm to this generic function names. A wrapper, spg.c, was thus written for the SPG Module driver, strobe.exe, described in Section 2.2.4.

An advantage of the command module is the directory structure it creates when it takes a sequence of images. The vision algorithms were designed for such a directory structure. The command module also provides a mechanism to operate the Linux camera which significantly reduced the driver development time. The command module was not designed with the SPG Module in mind. This required changing the architecture and flow of the code, as the SPG Module provides a "picture done" signal and has the ability to control the "Integrate" line. It is essential to keep track of these signals to keep the timing in check.

The compilation of the command module, *cm_getdata.c*, results in the executable which can accept a whole range of arguments. The following is an example call to the executable :

```
cm_getdata -picname /user/danny/test -frequency 1000,2000
-phasedivisions 7 -phase 0,5 -objpos 0,50 -ledontime 0.05
-roi 1,87,518,579
```

where *objpos* refers to the piezo position. The above call highlights only a few switches. Table 6.1 shows a list of all possible arguments.

Table 6.1: Arguments for the Command Module Interface

-help	Displays all the switches.
-debug	Enables debugging mode (specified by level).
-exposuretime	Specifies the exposure time of the camera in seconds.
-gain	Specifies the gain of the camera, in dB.
-ledontime	Specifies the amount of time that the led should be on in seconds.
-roifile	Specifies a file containing the region of interest.
-roi	Specifies four coordinates defining the region of interest.
-roimax	Specifies the maximum region of interest provided by the camera.
-commentfile	Specifies a file containing the comment text.
-comment	Specifies the comment text.
-picname	Specifies a string that each pictures generated name will automatically be appended to.
-phase	Specifies the phase(s) to take data on.
-frequency	Specifies the frequency(s) to take data on, in Hertz.
-amplitude	Specifies the amplitude(s) to take data on, in volts.
-objpos	Specifies z position(s) of the piezo to take data on in microns.
-waveform	Specifies waveform type (sinusoid, square, triangle).
-offset	Specifies the DC offset, in volts.
-waveformcenter	Specifies the waveform center in volts.
-samplesperperiod	Specifies the number of samples per period in certain types of stimulus generators.
-phasedivisions	Specifies the number of divisions per phase.

6.1.1 Selecting a Region of Interest

The cm_getdata executable uses a *-roi* argument to specify a region of interest. This is very important because transporting a full size picture (1200 x 1600 pixels) is not bandwidth efficient. We will see in the next chapter that the client interface takes advantage of this in bandwidth critical applications such as "Live Video" used for remote focusing.

6.1.2 Get Data Messaging Format

The GET-DATA message shown Table 6.2 is used to call upon the cm_getdata executable and obtain a data set. The executable is called after decoding the GETDAT-

ACOMMAND variable which includes the specifications for frequencies, phase, etc. The pathname argument tells the executable where to store the files and create the directory if necessary. Since the pathname has to be unique, the handler generates a pathname by determining the current date and time. An example pathname would be `2001_1_13_14_51_12` which comprises of the year, month, day, hour, minute, and second of the handler call. This path name is then sent back to the client by encoding the information in the DATASETPATH variable.

Table 6.2: Structure of the GET-DATA Message

Client Request	COMMAND = GET-DATA SESSION-ID = *sessionID* GETDATACOMMAND = *getdatacommand* GIFCONVERSION = *gifconversion*
Server Response	COMMAND = ACK SESSION-ID = *sessionID* DATASETPATH = *datasetpath*

Web browsers and stand alone Java only support images that are WWW compatible. That is, they must be in the *gif* or *jpeg* format. The output of the camera is in *nd* format which is raw binary. Since a custom program was not written, 3 Unix scripts were used to attain the desired format. The process involved going from *nd-to-pgm*, *pgm-to-ppm*, and finally *ppm-to-gif*. These scripts are pretty CPU intensive and take a significant amount of time to accomplish their task.

The time spent in the format conversion routine is so much that it calls for having the user's consent before running them. Often times, the user may simply want to proceed with the direct processing of the vision algorithms which use the *nd* format. In such a circumstance, there is no need to perform the format conversion routines on the gathered data set. The GIFCONVERSION variable in the message header accommodates this feature. If is is equal to "YES" then the all the images in the data set are converted to the *gif* format.

It is very likely that a user may simple want to view one particular image in the whole data set. The CREATE-GIF message can be used for such a purpose. The format of the CREATE-GIF image is shown in Table 6.3.

Table 6.3: Structure of the CREATE-GIF Message

Client Request	COMMAND = GET-DATA SESSION-ID = *sessionID* IMAGEPATH = *imagepath*
Server Response	COMMAND = ACK SESSION-ID = *sessionID*

The client, in its message header, supplies the name of the image (only one) via the IMAGEPATH variable. The handler than converts the specified image which has an *nd* suffix to GIF format. The client display can then access the image with *gif* as the suffix.

6.1.3 Data Module

Careful analysis of the command module source code will show a call to a subroutine, data.c, that is responsible for sequentially firing commands to acquire and then store the images. The handlers to be discussed in this and the following chapter use customized versions of the cm_getdata executable. These executables differentiate by the module they load, i.e instead of the "data" module. The following fragment of code extracted from cm_getdata.c shows that only one module is loaded at compile time instead of the "data" module and this differentiates the type of executable created.

```
#ifdef JAVAREMOTEFOCUS
 load_module("javaremotefocus");
#elif JAVAPIXSTART
 load_module("javapixstart");
#elif JAVASLOMO
 load_module("javaslomo");
#elif XDT_AUTOFOCUS
 load_module("xdt_autofocus");
#else
 load_module("data");
#endif
```

The makefile used to compile the code is included in Appendix G. It compiles the same code, cm_getdata.c, to create multiple executables that serve a slightly different purpose. The *j_pixstart* command used by the OBTAIN-SAMPLE-IMAGE message described in Section 5.1.5 was compiled in this way and used the *javapixstart* module.

6.2 Slow Motion Handlers

Slow motion analysis of a moving MEMS device serves as an excellent visual analysis tools. The basic idea is to capture images at all phases and then play them sequentially. The result is a time-video of a moving device which proves to be an excellent analysis tool in addition to providing a visually satisfying experience. The RUN-SLOMO message was used and its format is as follows :

Table 6.4: Structure of the RUN-SLOMO Message

Client Request	COMMAND = RUN-SLOMO SESSION-ID = *sessionID* SLOMOCOMMAND = *slomocommand*
Server Response	COMMAND = ACK SESSION-ID = *sessionID* MESSAGE = AUTO-FOCUS SLOMOPATH = *slomopath*

The handler expects the SLOMOCOMMAND variable to specify the frequency, phase divisions, region of interest, and ledontime. The handler generates a unique file name based on the day and time stamp as discussed earlier. These arguments are sent to an executable that was obtained by compiling the command module with the javaslomo module, i.e. *javaslomo.c*. The following segment of code highlights an example of the executable call after decoding the value of SLOMOCOMMAND.

```
execute (j_slomo -frequency 1000 -roi XXXX -picname current_day_time)
```

The executable places all the images in a directory that can be accessed by a client, and returns this directory name to the client by encoding the SLOMOPATH variable.

6.2.1 Playback of Images

Once the client has been informed of the location of the images, the user interface can then *GET* the images and display then sequentially. After the first loop, the images are cached and the playback should be smooth. An alternate technique of creating an animated gif was considered using a share-ware program called *gifmerge*. This program, in essence, creates a movie out of a sequence of gif images. While some reduction in total image size takes place, the program at compression time requires a "playback" speed. It is quite desirable for a user to be able to vary this and this approach was therefore not used. Hence, it is up to the client interface to determine the rate at which the images should be sequenced.

6.3 Chapter Summary

This chapter provided insight into the command module software and the various executables that were derived from it. Carrying on in the same line of thinking, the next chapter will highlight the architecture used for remote focusing and the executables that were derived from the command module interface.

Chapter 7

Remote Focusing

Earlier chapters showed how a client can use the SET-STAGE message to align a MEMS device under the camera and then control the Z-axis of the MFU to put the device into the focusing range of the pifoc. The OBTAIN-SAMPLE-IMAGE message can be used to acquire a sample image, while the SET-PIEZO message can be used as a knob for fine-focusing.

7.1 The Need for Remote Focusing

Based on the capabilities of the system described so far, a technique for fine focusing would involve continuous repetitions of taking a sample image, adjusting the piezo position, and then taking the image again. This process of manually refreshing the image is tedious and most importantly, it is slow. We solve this problem by allowing the server to broadcast a stream of "Live Video" and then the user only has to tweak the piezo and *see* the effect.

7.2 Server Support for Broadcasting Live video

The server provides support for remote focusing by allowing a background process to continuously take images and overwrite a known filename, *LiveImage.gif*, with the latest image. The client at its own pace and processing power would then *GET* this

image. The original server architecture was not based on a multi-tasking architecture and the handler would not return till the executable had returned. To overcome this problem, the executable was changed so it would fork out a process. That is, once the executable was called, it would return after launching a background process that continuously acquired images.

Provisions were made to stop this process when needed. When the forked process is first launched it creates a dummy file. The forked process remains active as long as the dummy file exists. Hence, some external program can delete this file to stop the forked process.

7.2.1 Shortcomings and Other Techniques

The drawback of the implemented technique is predictable. Random flickering occurs on the clients screen if the server and client attempt to access the image at the same time. However the performance is better compared to all other techniques that were investigated. Appendix B provides a discussion on the various techniques that were investigated.

7.2.2 Remote Focusing Message Format

The FOCUS-IMAGE message was used to launch the background process on the server that would continuously acquire images. The message format used is :

Table 7.1: Structure of the FOCUS-IMAGE Message

Client Request	COMMAND = FOCUS-IMAGE SESSION-ID = *sessionID* ACTION = *action*
Server Response	COMMAND = ACK SESSION-ID = *sessionID* MESSAGE = FOCUS-IMAGE

After decoding the value of ACTION, the handler executes the following code :

```
rm -rf /tmp/javaremotefocus.looping
if ( !  ACTION.equals(''STOP''))
 execute (/usr/local/cm/bin/jrf_getpic action)
```

If the argument STOP is sent by the client, then the remote focusing would halt as the temporary file would be deleted. Any other argument would call the executable. If an on-going process is told to start, then the process would restart.

The *jrf_getpic* executable was obtained by compiling the command module with the *javaremotefocus.c* module. Several comments about this module are essential at this point. First, this module is unique because it does not excite the SPG Module as there is no need for an output waveform. In fact, in this mode, the SPG Module keeps the LED on at all times. Second, the amount of light is controlled by programming the camera with an exposure time. This software programmability feature of the camera allows the user to determine the right value for *ledontime*. Recall from Section 5.1.5 that *ledontime* determines the number of strobe flashes which must be programmed into the SPG Module. Hence, the decoding of the ACTION parameter sent by the the client would reveal something like :

```
ACTION = ''-exposuretime 0.05 -roi x1,x2,y1,y2''
```

It is important to realize that in this case the camera controls the amount of light being exposed onto the CCD. This is different from a true experimental setup where the SPG Module controls the light source. In the later case, the SPG Module opens the shutter and on the right phases turns on the LED. After the elapsed flash count, the shutter is closed and the image is copied from the CCD buffer to the hard disk. Nevertheless, this technique is useful for focusing and determining an appropriate value for *ledontime*.

The performance depends on the dimensions of the region of interest. On average 3-4 frames per second are achieved for a 250 x 250 size image.

7.2.3 Throughput Bottlenecks

A bottleneck that has been ignored so far has been the image format conversion that has to take place on the server. The three conversion scripts, *nd-to-pgm*, *pgm-to-ppm*, and finally *ppm-to-gif*, are a serious bottleneck to the overall throughput of the stream.

Furthermore, our approach of broadcasting high-resolution images is not feasible in bandwidth critical situations. Modern video stream broadcasting architectures and formats, such as Real Video need to be investigated.

7.3 Auto Focusing Handlers

To accompany the remote focusing ability of our system, Xudong Tang, developed an auto focusing mechanism as part of his Master's thesis [16]. His subroutine was integrated into the command module code with the ability to take a ROI argument. This algorithm sweeps through the entire piezo range and uses convolution to determine the piezo position with the maximum sharpness.

The AUTO-FOCUS message was used for this purpose. The idea being that a user would send a message to start the auto focus and the handler would return the optimum piezo position. The messaging format used is as follows :

Table 7.2: Structure of the AUTO-FOCUS Message

Client Request	COMMAND = AUTO-FOCUS SESSION-ID = *sessionID* AUTOFOCUSCOMMAND = *autofocuscommand*
Server Response	COMMAND = ACK SESSION-ID = *sessionID* MESSAGE = AUTO-FOCUS AUTOFOCUS = *plane*

Upon decoding the AUTOFOCUSCOMMAND argument, something similar to the following gets executed on the server.

```
xdt_autofocus -roi ....  -exposuretime ....
```

It is important to realize that the client is able to send the ROI he/she wants to focus upon. In fact, this is a requirement as a single image may have devices located at varying heights. The executable upon completion sets the pifoc to the optimum position and writes this position to a file. The handler reads this file and sends it to all clients by encoding it in the *plane* variable.

7.4 Chapter Summary

This chapter discussed the architecture used for remote focusing. The handler used to evoke the auto focusing mechanism was also introduced. The next chapter will highlight the client interface that will call upon all the handlers discussed so far to operate the MEMS characterization system.

Chapter 8

Client Interface

Prior chapters outlined the server architecture and established the messages a client can send to remotely operate the MEMS characterization system. The next challenge was the design of a client interface that fulfills several objectives. First and foremost, the client must be capable of connecting to the server. Second, the client must be able to send messages to the server as well as decode messages from the server. Third, the client must be able to to access and operate all aspects of the system without worrying about the underlying details of the back-and-forth communication.

The messaging protocol is the glue that binds the client and the server. This means that there is no requirement on the software used to develop the client interface. The client interface can be written in C or Java as longs it communicates with the server using the specifications set forth via the messaging protocol. Section 4.5 highlighted the difference between the client and the server software development environments.

8.1 Why Java

The language chosen to develop the client interface was Java. Java simplifies the creation of a GUI by having the Advanced Working Toolkit (AWT) class as part of it core classes. AWT provides a mechanism for creating windows, frames, buttons, menus, etc. Furthermore, by extending the AWT class with *Swing* components, extra gadgets such as a slide-rulers and split panes can be realized.

There is a special class in Java that is the applet class. It defines an object that allows Java code to be downloaded over a network and run within a Java Virtual Machine (JVM). A JVM can be thought of a program that simply executes compiled Java code. Nowadays, the JVM is part of all web browsers.

The applet environment thus provides a medium to download Java code over the Internet and execute it locally. This mechanism makes it possible for the client to simply point the browser to the server's URL and obtain the Java code to be executed by the browsers JVM.

8.2 Components of Executed Code

The Java code to be executed by the client can be divided in two main components - Core Unit and Visual Unit. Although they are closely bound to one another, it is important to study them individually.

The first part is responsible for the underlying details of the messaging protocol. It handles the details of putting the message in the right format, decoding information from the server, sending a poll message periodically, etc.

The second part of the Java code that gets executed by the client's JVM is the Visual Unit and is responsible for generating the GUI. This code draws the user interface on the clients screen and is capable of deciphering signals from the Core Unit. For example, the Core Unit after a poll may update a variable that contains the piezo position. The Visual unit picks up this information and updates the client interface to show the current piezo position.

The Visual Unit also generates function calls to the Core Unit to make a certain thing happen. For example, a user may type in the frequency and then click "update". The Visual Unit is only responsible for forwarding this request to the Core Unit. The Core Unit then massages this request into the format of the SET-STROBE message and sends it off to the server to make it happen.

8.3 The Need for a New Visual Unit

Erik Pedersen for his Masters thesis [12] laid out the foundation of the Core Unit and developed a Visual Unit to interface with it. However, at the time of his thesis completion, there was no hardware connected to the server. As the hardware evolved and drivers were written, the messaging format was refined and modifications were made to the Core Unit.

Erik's Visual Unit was based on a "Settings File Parser" approach. The idea involved having the user upload a settings file to provide a custom look and feel. Since this approach was based on a script - it had had limited abilities, i.e support for basic menu, frames, buttons, etc.

The evolving nature of this research project required the use of more components from the AWT toolkit and a different way to launch and interpret experimental results. This could only be accomplished by generating a custom GUI and not having to worry about the limitations of the settings parser script.

8.4 Interface Details

From now on, this chapter will provide a pictorial view of the GUI created to remotely operate the MEMS characterization system. The details of the several thousand lines of Java source code that make all this possible are irrelevant and will not be explained.

8.5 Login

The first step in connecting to the system is pointing the browser to the MEMS server URL. After the client has fetched the *web page*, a login/logout applet appears in the browsers window as shown in Figure 8-1.

The client *logs-in* by clicking on the "Login" button which initiates a contact with the server. The client then starts to download several Java classes to be executed within the browsers JVM. After a few seconds, two additional windows have appeared on the clients screen as shown in Figure 8-2.

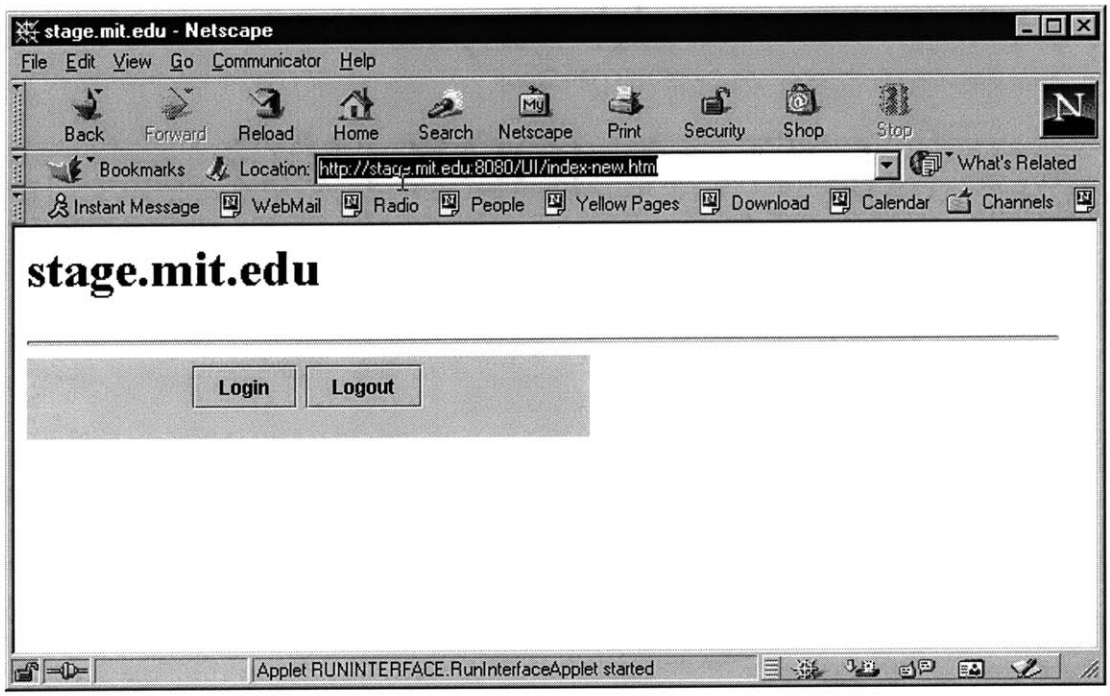

Figure 8-1: Login Interface - *Step 1*

The first window that has appeared is the Message Activity window which is an excellent development and diagnostic tool. It monitors and logs all *activity*[1], with the exception of polling, between the client and the server. To the user, it acts as the "What Just Happened" screen.

The second window that appears is actually waiting for the users feedback. The "Settings File Chooser" window provides the user with a list of setting files that exist on the server. The user may select anyone of the settings file and click "Load Settings File". This will launch the settings parser algorithms and create a GUI based upon the users request. Alternately, the user may click on the "Load Default" button and bypass the settings parser routine. The later option is the focus of this research and the user thus clicks this button to proceed.

[1]Here, the term *activity* refers to the bi-directional exchange of message between the client and the server

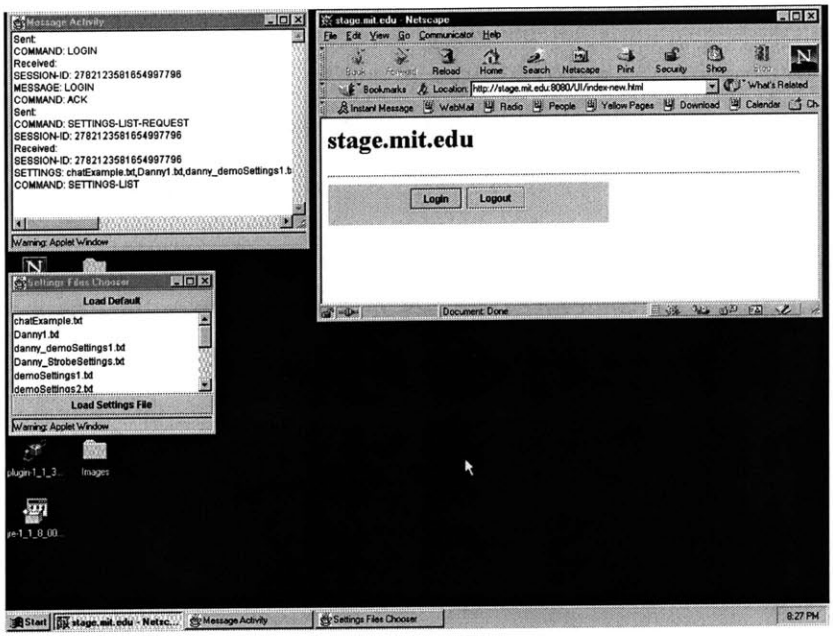

Figure 8-2: Interface After a Successful Login - *Step 2*

8.6 Main Window

Upon clicking the "Load Default" button, the "Main Window" pops up on the screen. This window, shown in Figure 8-3, acts as the central controller of the user interface by providing buttons and menus that fan-off different control interfaces. Lets look at the them in detail :

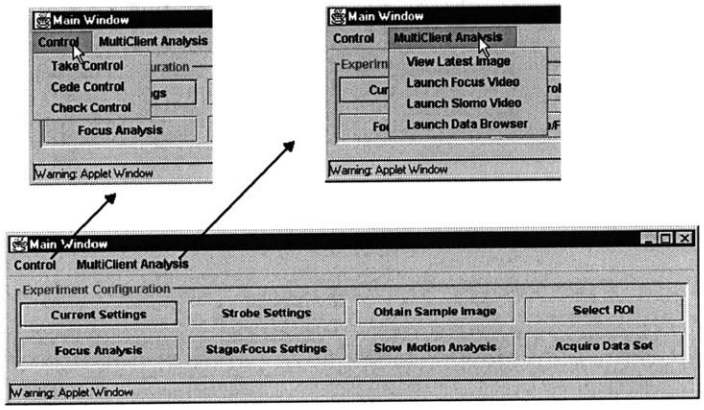

Figure 8-3: Main Window

8.6.1 Control

As seen in Figure 8-3, the control menu provides the user with three options. The user can either take control of the system, cede control of the system, or check for the control status. The SET-CONTROL message is used for this purpose [1].

The first logical step in operating the system is to take control. If the user fails to do this and continues on, then an error message will be generated as shown in Figure 8-4.

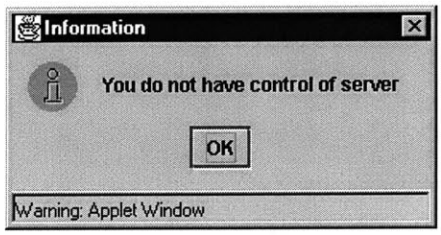

Figure 8-4: Control Error Message

8.6.2 Current Settings Window

The "Current Settings" window appears upon clicking the appropriate button on the Main Window shown in Figure 8-3. This window monitors the hardware settings and updates the screen upon receiving a poll message from the server. It therefore allows the user to monitor the system even in the absence of control.

The Strobe Settings box shows the frequency and phase division of the waveform driven on the MEMS device. The Stage Settings box shows the current stage position where (0,0,0) corresponds to the origin. The Focus Settings box shows the current piezo position and the results of the auto focus algorithm (if any). The Image Settings box shows the selected phase, ledontime, camera gain, and the corresponding flash count.

The Magnification and Wafer Settings hardware are not used and have been included for historical reasons. The Series Parameter box provides the latest path names for the slomo-motion and data-set images (if any). The control state, in this case, shows that the user has control over the system.

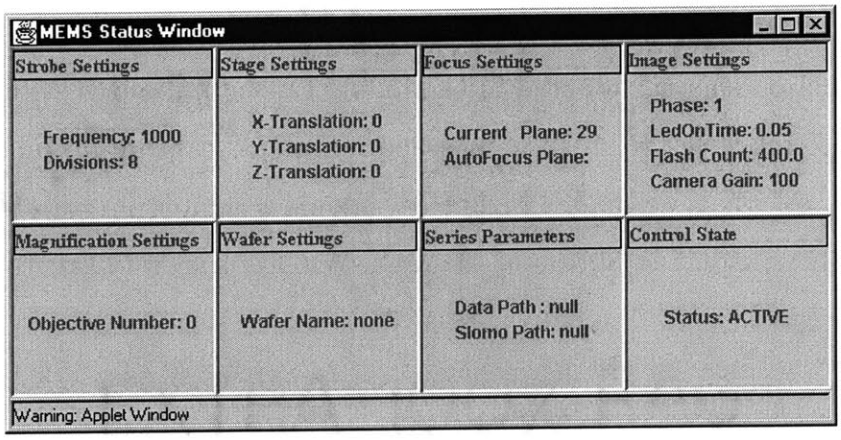

Figure 8-5: Status Window

8.6.3 Strobe Settings

The Strobe Settings interface launched from the Main Window shown in Figure 8-3 allows the user to specify the frequency, phase divisions, and waveform type. It also allows the user to specify the parameters which the user is most likely to vary from image to image, i.e phase, ledontime and camera gain.

The user updates the text-boxes with desired values and presses "return". An appropriate message is sent to the server depending upon the section that was modified. An update in the *Stimulus* portion of the GUI is transmitted to the server via the SET-STROBE message described in Section 5.1.3. Similarly, the server is notified of the new *Image Settings* via the UPDATE-IMAGE-SETTINGS message described in Section 5.1.4

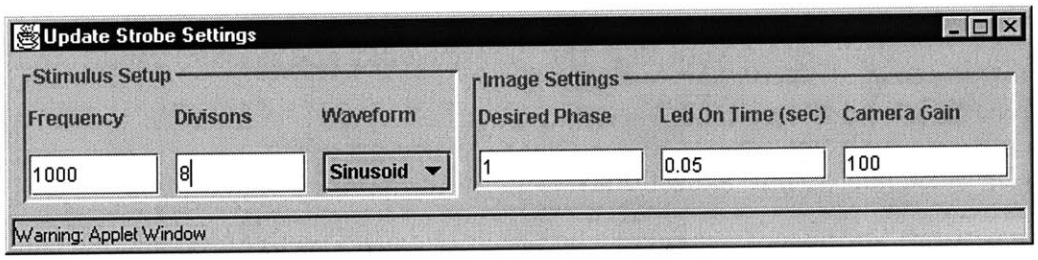

Figure 8-6: Strobe Settings Interface

8.6.4 Obtaining a Sample Image

The prior section showed how the SPG Module can be initialized with the desired stimulus and image settings. It is thus appropriate to capture an image from the camera to see "what we got". Figure 8-7 shows a sample image obtained upon clicking the corresponding button of the Main Window shown in Figure 8-3.

Figure 8-7: Sample Image of a MEMS device

The underlying process sends out an OBTAIN-SAMPLE-IMAGE message described in Section 5.1.5. Note that this interface did not ask for a ROI arguments and returned a full size image.

It is very likely that a user may not obtain a high fidelity image due to the image settings and after tweaking some parameters the user can try again. But most important of all, we have assumed that there is a device directly underneath the camera and in focus. Most likely, this will not be the case, and we will have to modify the stage and pifoc settings before attempting another image.

8.6.5 Stage and Piezo Settings

The stage and focus interface is also launched from the Main Window shown in Figure 8-3. Figure 8-8 reveals that it is a busy interface that provides slider and text-boxes to specify relative and absolute motion of the stage and piezo.

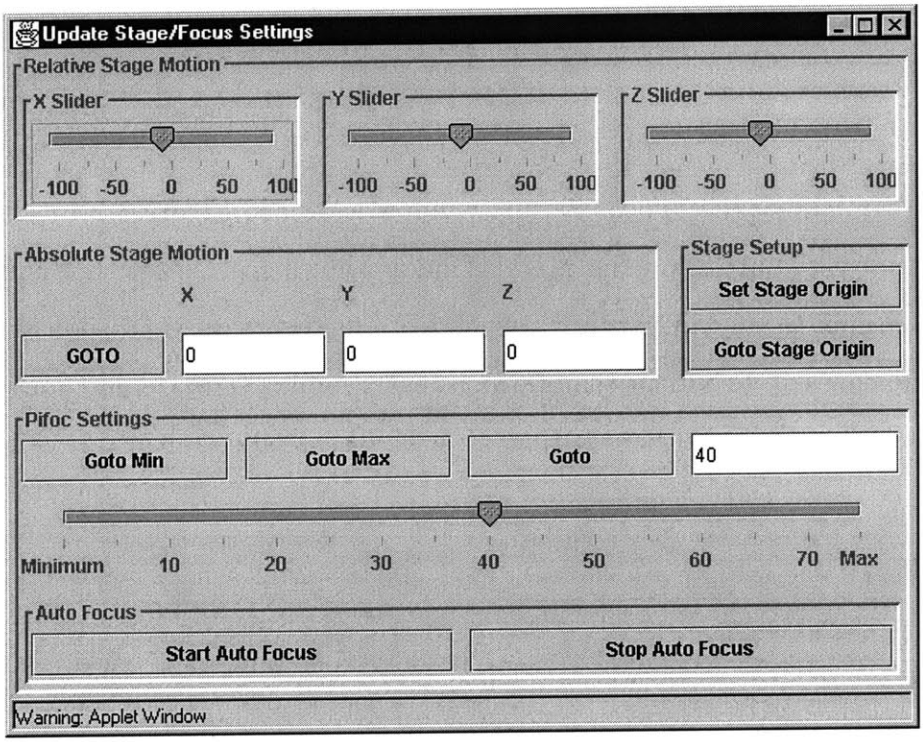

Figure 8-8: Stage and Focus Settings Interface

The X-slider, Y-slider, and Z-slider present in the *Relative Stage Motion* accommodate the differential motion of the axis. The 'Z' motion here refers to the action upon the MFU which we have considered to be the *large scale focus*. Alternatively, the user can specify the precise location in the *Absolute Stage Motion* section. The *Stage Setup* buttons allow the user to *set* any position to be the origin and then *return* to this position after wandering around. All these tasks are accommodated via the SET-STAGE-FOCUS message described in Section 5.1.1.

The *Goto-Min* and *Goto-Max* buttons in the *Pifoc Settings* do as their name suggests, while the *Goto* button puts the pifoc in the position specified by the text-box. Alternatively, the user may simply move the slider to specify the absolute

position. The auto focus buttons available in this interface will be discussed later on in the chapter.

With the limited introduction so far, the user has the ability to move the stage and obtain sample images. However, if the user is looking for a specific region on the device being viewed, then "Live Video" can be enabled to ease the process. The user will only have to modify the stage/pifoc settings and see the *"Live"* effect.

8.6.6 Live Video

The user can launch the "Focus Setup" window shown in Figure 8-9 by clicking the "Focus Analysis" button on the Main Window shown in Figure 8-3. The name of the button may seem odd versus "Live Video", however, we shall soon see that its primary purpose is to aid in remote focusing. Finding a particular location on a MEMS device is an added benefit that we have discussed so far in this chapter.

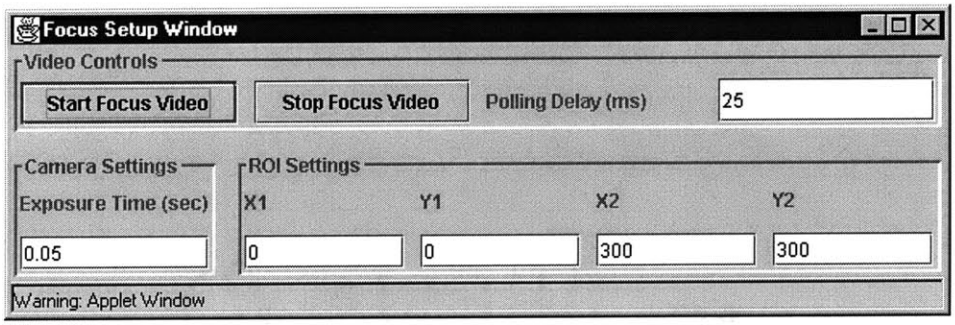

Figure 8-9: Focus Control Interface

Upon clicking the Start and Stop buttons, the FOCUS-IMAGE message described in Section 7.2.2 is sent to the server. The message includes in its arguments the parameters for the ROI and exposure-time. The architecture behind this feature was the basis of Chapter 7 which mentioned that flickering would occur if the client and the server accessed the file at the same time. The "Polling Delay" variable adds a delay before the client attempts to fetch the next image. A higher "Polling Delay" therefore result in a low refresh rate, but makes the user-interface faster as less processing-time is spent fetching images. Note that the "Polling Delay" term used here has no relation

with the client polling the server for messages described in Section 4.6.

Varying the exposure-time would vary the brightness of the image. This gives the user a sense of how much *ledontime* is appropriate for the strobe settings. The interface by default has a ROI which specifies the upper left quadrant. The user can manually update the ROI coordinates for a different quadrant.

Figure 8-11 is an example Live-Video screen. The user at this point can move the stage and pifoc and see the real-time affect of his/her actions.

The Live-Video application discussed so far involved having the user manually type in the ROI coordinates. More than often, the user would like to specify the region of interest by identifying a section of the image.

8.6.7 Remote Focusing

There are several reasons why a user may want to specify the ROI. Bandwidth constraints among other things restrict the size of the image that can be shown in the video screen. The prior section showed how a user would fix the quadrant and only look at that portion of the image. Suppose a user locates a region on the full-size image that is of interest. How would the user then go about specifying the *exact* ROI coordinates ? Do we expect him to move the stage ? Furthermore, specifying a section of a full-size image is a requirement for vision algorithms as an image may contain devices at multiple focal planes.

The MEMS user interface accommodates this by having the user click on the "Select ROI" button of the Main Window shown in Figure 8-3, *after* obtaining a *new* sample image. An image canvas pops up that shows the recent full-size image obtained in the background and the user specifies the ROI portion with a mouse. The mouse is used to superimpose a green rectangle, which the user draws, on top of the image. The underlying Java code then extracts the coordinates appropriately. Figure 8-10 shows a selected region of interest. The title of the image in the upper left-hand corner shows the selected ROI coordinates (431,384,638,606). To confirm the new ROI, the user closes the window and the client interface now remembers it for all future processes that require this information.

Figure 8-10: Specifying the Region of Interest on a Sample Image

With a ROI now specified, the user can now follow the steps described earlier to re-start the Live-Video application. Note that this time the ROI coordinates on the interface will not be the default values, i.e (0,0,300,300), but the ones we just specified. Figure 8-11 shows an example Live-Video window obtained after specifying the ROI. The user can now vary the pifoc settings, MFU settings of the stage, exposure-time of the camera to find the sharpest image.

8.6.8 Auto Focus

An alternate to remote focusing is auto focusing which was discussed in Section 7.3. Here, the AUTO-FOCUS message sent by the client includes the recently selected ROI coordinates which specifies the region to focus upon. The buttons to execute this command are part of the *AutoFocus* section shown in Figure 8-8. Since the autofocus command has to use the camera, an ongoing "Live-Video" process is temporarily

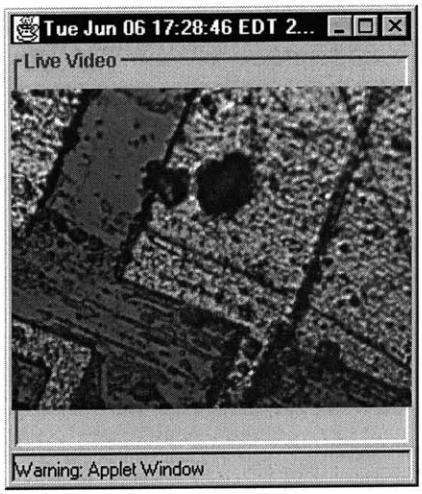

Figure 8-11: Focus Image Viewer

frozen. Furthermore, the AutoFocus command puts the pifoc into the "best" position upon completion. As a result, when "Live Video" continues, the video screen portrays a "before and after" effect which can be remarkable at times.

8.6.9 Slow Motion Analysis

Once the client has specified the ROI, focussed on the ROI, and has a feel for the exposure time needed to properly image the device, it is time to perform a slow motion analysis. The interface to launch such an activity is shown in Figure 8-12. This interface can be obtained from the Main Window shown in Figure 8-3.

Observe how the ROI and exposure time have already been filled in, but the provisions to change them for this particular experiment exist. The user specifies the frequency and phase-divisions and upon hitting the "Start Motion Analysis" button, the RUN-SLOMO message described in Section 6.2 is sent to the server. The message returns with the location where the images are stored and a viewer is launched that sequences through the images.

The viewer shown in 8-13 is silent for the first-loop as it caches the images. After that, a smooth playback is obtained and the speed can be varied via the slider. The playback can be stopped at anytime by closing the window or hitting the "Stop

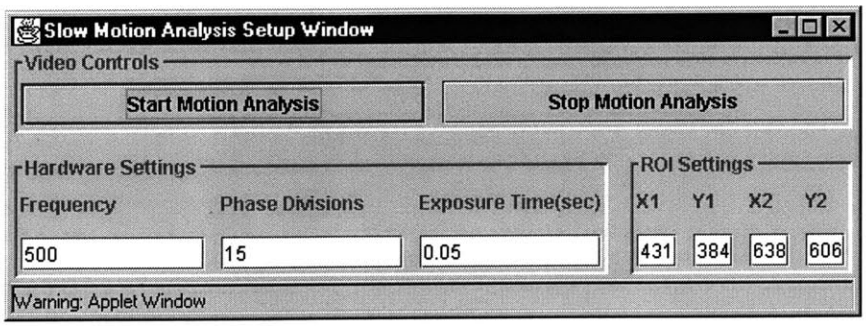

Figure 8-12: Slow Motion Setup

Motion Analysis" button in Figure 8-12.

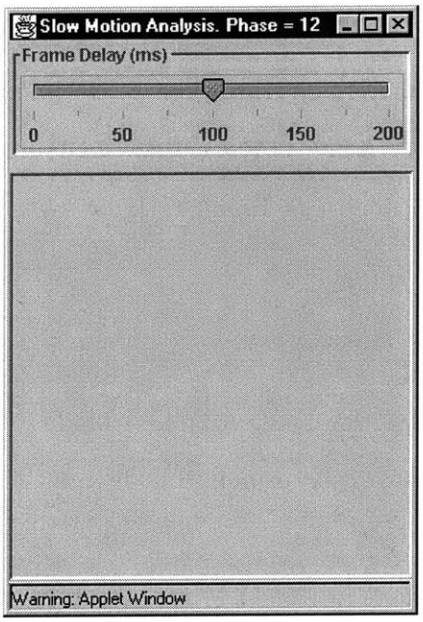

Figure 8-13: Slow Motion Viewer

8.6.10 Obtaining a Data Set

Once the user is comfortable with the MEMS device, the next step is to obtain a series of images. To do so, the user launches the "Obtain A Data Set" window shown in Figure 8-14 from the Main Window shown in Figure 8-3.

This window serves as a means to configure an experiment by allowing the user

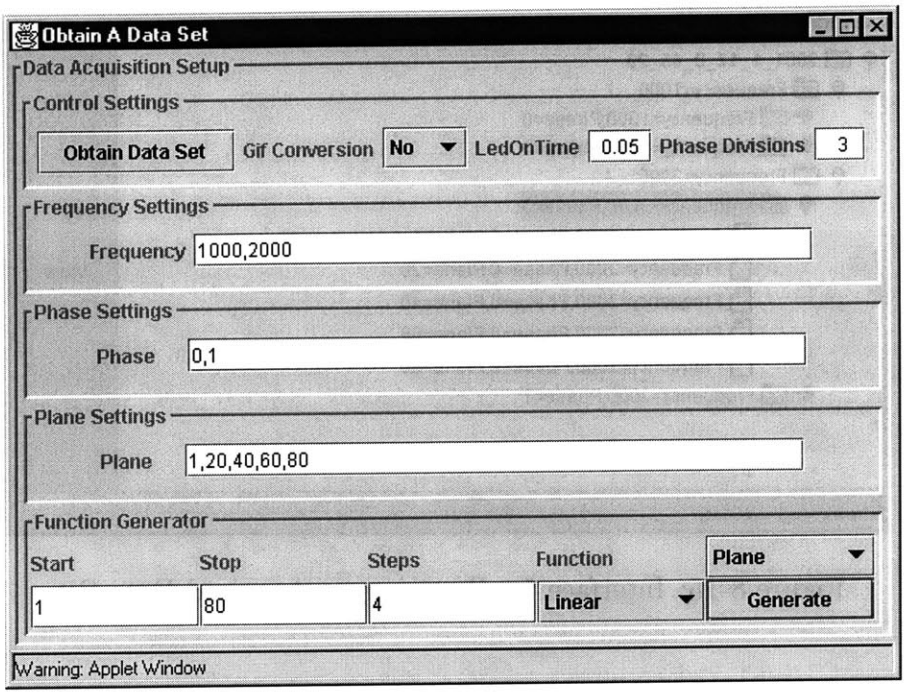

Figure 8-14: Setup for Gathering a Data Set

to specify the frequencies, phases, and planes at which images should be taken. The user can either type in these number manually, or use the *Function Generator* to make a logarithmic or linear series. The choice of linear or logarithmic depends upon the analysis and type of plot the user expects from the vision algorithms.

Ledontime and phase-divisions are parameters the users modifies if needed. The GIF conversion option requests that all images be converted from "nd" format to GIF and as Section 6.1.2 highlighted, this is a time-consuming operation.

Once the user has configured the experiment, the "Obtain Data Set" button is to be pressed. The OBTAIN-DATA-SET message described in Section 6.1.2 is sent to the server which returns with the pathname where the images are stored. A Data Browser Window, as shown in Figure 8-15, is then automatically launched so the user can view the images.

The user can navigate the browser and select the image he/she would like to see, and then click "View". If the GIF version of the image does not exist, then the interface automatically sends out the CREATE-GIF message described in Section

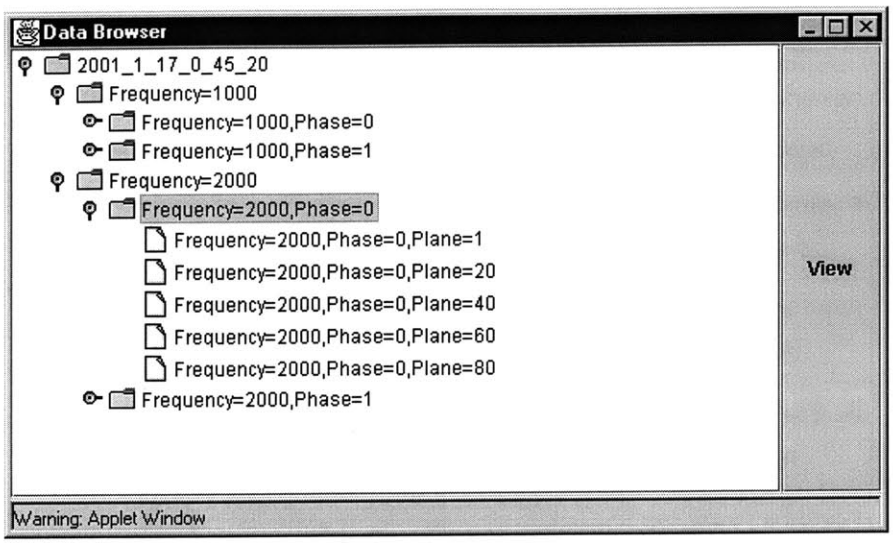

Figure 8-15: Interface for Browsing an Acquired Data Set

6.1.2 for that particular image. An image-canvas, similar to Figure 8-7, then pops up to display the image.

At this point, the obtained data set can be downloaded by the client for local analysis and/or transported to another host capable of performing the motion analysis.

8.7 Multi Client Analysis

It is often desirable that a user be able to share the data among other users. This was discussed in Section 1.3. The Multi-Client Analysis menu options shown in Figure 8-3 allows all clients connected to the system to view the latest image taken and see an ongoing "Live Video" session. In addition, all logged in clients can watch the latest motion video and browse the latest data set.

Chapter 9

Conclusion

Computer microvision acts as a good analysis tool during the testing and development stages of the design process. The computer microvision method involves driving the MEMS device with a periodic stimulus and capturing the motion via stroboscopic illumination. The 3-D behavior of the motion can be monitored by taking images at multiple focal lengths. Motion estimates for X,Y and Z can then be extracted from a set of images by using computer vision algorithms.

9.1 System Overview

In this research, a low cost computer microvision system is defined and implemented to characterize and test MEMS. The system includes a PC which has full control over a camera, a piezo electric device, a generic microscope with an X-Y-Z stage, and an LED to illuminate the moving MEMS device. Custom hardware includes the design of the Strobe Pulse Generator (SPG) Module for a PCI interface that acts as a central controller for stimulus and stroboscopic illumination. Obtaining an image of a moving MEMS device at a particular focal length, stimulus shape, frequency, phase, and light intensity is the most basic building block of the microvision system.

There are benefits in being able to operate the system remotely and support a multi-client environment. The design of the microvision system lends itself to being a remotely accessible system. This is because the computer, acting as the controller,

can be thought of as the server for all of the data and for all of the parameters of the system. There are two main modules of software that enables the computer to act as the server for the MEMS characterization system. First, there is the web server which includes servlets that overcomes the inability of the standard HTTP to deal with state. The second software module is the sum of Java classes that handle all the interaction between the client and the server. The client software consists of multiple modules written in Java and is intended to run remotely through a web browser.

The link between the client and the server is a messaging protocol and all communication is done via "messages". The messaging protocol is the glue that binds the entire system together which spells out the rules for the format and transmission of data between the client and the server. Each message has a handler, which is executed by the server upon the receipt of the corresponding message. These handlers are used to change the state of the hardware by calling executables, monitor the latest hardware settings, send messages to other clients, etc.

9.2 System Operation

A client connects to the server's URL via a web browser and is provided with a Graphical User Interface (GUI). This interface provides the client with full access to the system. The user can "login" and can then request control for the system. After setting the various strobe settings, the user can request a sample image. The user can move the X-Y-Z stage to find a Region of Interest (ROI) and alter the piezo settings to fine focus. To assist in this process, the user is provided with an option of "Live Video", which once enabled, allows the user to instantly see the effect of moving the stage or changing the focal length. Provisions are also made to focus on the ROI which the user can identify by simply drawing a rectangle on a previously acquired image. The user can also choose to auto focus on the selected ROI.

The system also supports slow motion analysis by showing a time waveform of the moving MEMS device for the requested number of phase divisions. The user can also configure an experiment and retrieve the data upon completion.

The computer microvision system can support multiple clients simultaneously and allows the sharing of data. However, at any given time, only one user can have control of the system, while the rest can only monitor the latest system settings. The multi-client analysis tools include shared live video, latest image acquisition, slow motion analysis, as well as viewing acquired data sets.

9.3 Future work

The current system architecture for remote access has some shortcomings. The problem is that HTTP, upon which the messaging protocol is layered, requires the client to initiate all communication. While this works well for most situations, there may be cases when this becomes a real issue. We try to get around this with a polling mechanism. But polling is far less efficient and scalable than a truly bidirectional protocol. Future work may try to develop a modified protocol that is a hybrid HTTP and some other protocol that is truly bidirectional.

An area of the user interface that could use much enhancements is the graphic manipulation abilities of the interface. More specifically, images cannot be cropped, filtered, marked, or saved back to the server in any modified format.

Remote focusing is essential for any remote microscopic application. The solution utilized in this research works only for high bandwidth networks. Even so, bottlenecks exist due to the image conversion subroutines and the lack of compression. Modern video stream broadcasting architectures and formats, such as Real Video, should be investigated and implemented.

The system presented in this thesis is truly one of a kind. Unfortunately, it has always stayed in the research lab. An effort can be made to house this in a real fabrication facility and feedback from *real users* can go a long way in improving and redefining the system architecture.

Before transporting our system to the real world, the issue of security needs to be addressed. Every effort must be made to ensure that a users data is not compromised.

9.4 Final Thoughts

This thesis started off in the mid stage of the research project and took it to completion. While a lot of the milestones were accomplished, there were some that weren't. Nevertheless, the ultimate goal, dictated by the project completion deadline, was to do everything, hardware or software, and demo a working system. With that accomplished, this research presented a first generation of remotely automated microscope to characterize and test MEMS.

Appendix A

Character Interface to the Led Flasher and Sine Wave Generator

```
21 Apr 1999 D. E. Troxel
31 May 1999 Rev.
21 Mar 2000 8 pm Rev.
01 Apr 2001 Last Rev.  by D. Seth
```

All communication is at 9600 baud.

All transmissions to the board require only 7 bit characters, the eighth bit is ignored. Thus, odd, even, or no parity is accepted. Only one stop bit is required, although two or more are acceptable. In short, any transmission at 9600 baud is acceptable.

All transmissions from the board are 8 bit characters, the eighth bit is zero and one stop bit is transmitted. Receivers that require two stop bits will work if characters are not received back to back (as is the normal case).

Data is transmitted by nibbles represented by 0 - 9 and a - f or 0 - 9 and A - F. That is, case is not distinguished for hex characters. All other 7 bit characters are special command characters.

Those characters which are not listed below are no-ops. They may be assigned a

meaning at a later time, so their use may result in unpredictable results.

All characters are echoed as received except for S (the pixstart character). When an S is received, there is no echo and a D is sent back when all the strobe (LED) pulses are finished for that picture.

Data characters that follow a special command character cause the data register to be shifted right by 4 and then the data character is used to represent the low four bits of the register. There are always either zero or four data characters except for the command character, L, which has 4 times the number of samples which is specified by the 16 bit value following the n command which must precede the L command. E.g., the following sequence of characters would load the hex number 012f in the data register.

< reg_load special character > 0 1 2 F

If the destination register has less than 16 bits then the high order bits of the data register are ignored.

In the table that follows the left column is the special command character which is also a printing character. Special command characters should always be a printing character. Remember, the characters 0 - 9, A - F, and a - f are not special command characters as they are hex data characters. The ASCII code in hex is given just to the right of each special command character. The next column is the length of the register in bits which is to be loaded after the special command character. The third column is the number of hex characters which are to follow the special character. The table entries are ordered by ascending values of the ASCII code for the command characters.

Char	**Numbits**	**Numchars**	
@ 40	0	0	A Null command which does nothing except the required echo of @.
G 47	0	0	Start the function generator.

I 49	0	0	Set the integrate line of the camera high.
L 4C	11	4*<n>	Load the ram with the number of samples loaded in the n register. Remember that four hex data characters are required for each sample.
M 4D	5	4	The Maximum number of strobe pulses.
N 4E	16	4	The number of strobe pulses required to produce the desired exposure.
O 4F	0	0	Turn the LED on.
P 50	5	4	The (phase - 1) of the strobe pulse. This must never be more than the maximum number of strobe pulses. '0' refers to the 2nd phase while the value used for 'M" selects the first phase.
R 52	0	0	All operations are stopped and the LED flasher and the function generator are Reset. All registers are set to all ones except for the ram and frequency generator chip which are unchanged. The LED is turned off.
S 53	0	0	Start taking a picture. There is no echo of S when this special character is received, a D (for Done) will be sent when the number of LED strobe pulse specified have occurred.
V 56	3	4	The Capacitor value selection which specifies the range of the VCO.
W 57	?	4	The time delay before starting strobe pulses. Cannot be zero.
g 67	0	0	Stop the function generator.
i 69	0	0	Set the integrate line of the camera low.
l 6C	8	4	Load the clock generator with a byte of data. The early bytes loaded into the clock generator determine whether the number of bytes loaded is five or eight. Remember that two hex data characters are required

			for each byte.
n 6E	11	4	Specifies the number of samples to be loaded in the ram starting with address zero of the ram.
o 6F	0	0	Turn the LED off (default).
r 72	0	0	The address lines of the ram and clock chip are reset to zero.

This character interface module will incorporate the receiver and transmitter modules and provide an interface to the led flasher module and the sine wave generator module. The following data comprise the interface between the character interface module and the led flasher and sine wave generator modules.

cmd A four bit command code as specified by the following table. This is not changed until a new command is received. Note that command characters are not listed in the same sequence as in the previous table. Note that three command codes are unassigned and reserved for possible future use.

Code	Char	Data Chars
0	V	4
1	W	4
2	M	4
3	P	4
4	N	4
5	n	4
6	l	4
7	L	4*<n>
8	*unassigned*	
9	S	0
A	R	0
B	G	0
C	g	0
D	r	0
E	*unassigned*	
F	*unassigned*	

data A 16 bit data value.

The next two signals are control signals which may be generated by clocks different from the led flasher and sine wave generator modules. Therefore, they should be synchronized by the receiving module.

newdat This is asserted high whenever new data is stable. It remains high until led_ack or sine_ack is asserted high.

newcmd This is asserted high whenever a new command is stable. It remains high until led_ack or sine_ack is asserted high.

The next two signals are control signals which may be generated by clocks different from the character interface module. Therefore, they are synchronized by the character interface module.

led_ack This is asserted high whenever the led flasher module or the sine wave generator module has recognized the newdat or newcommand signal assertion. It is de-asserted when both newdat and newcommand are low.

sine_ack This is asserted high whenever the led flasher module or the sine wave generator module has recognized the newdat or newcommand signal assertion. It is de-asserted when both newdat and newcommand are low.

Appendix B

Techniques Investigated for Broadcasting Live Video

Chapter 7 described the method used for remote focusing. However, it is important to discuss the other methods that were investigated and why they were not used.

Several approaches were investigated to accomplish the task. The first technique involved the client user interface running through a continuous loop of OBTAIN-SAMPLE-IMAGE messages in the background. In this way, the user would only have to worry about sending messages to modify the piezo position while the display of the current image would update automatically. A very important hurdle pertaining to multitasking of the user interface had to be overcome in order to accomplish this. Specifically, the existing user interface architecture was not designed to support multitasking and therefore did not allow "threads" [12]. After all the necessary modifications to the user interface architecture, this scheme was implemented and the shortcomings surfaced. First, the client interface would become very slow and just hang at times, thereby affecting the reliability of the user interface. Second, a strain was added on the server that had to process this flood of messages. All this led to a horrible refresh rate of about 1 frame every 3-4 seconds.

The next approach aimed at minimizing the transfer of messages between the

client and server to improve performance. The idea was that the client would send a message once to start the video, and during its poll interval it would pick up the path to the latest image. A reasonable refresh rate would be at least 2-3 frames a second, which meant that the poll timer would have to be reduced from the original 5 second to 0.33 seconds. This however led to different problems. First, the client GUI would became very slow as it would spend most of the time and processing power taking care of the poll messages, even when the poll process was launched as a separate thread. However, the main problem would develop when another client would connect to the server. Multiple clients polling at such high rate would reduce the overall system performance. This approach, at best, would yield a refresh rate of 1 frame every 2-3 seconds and a very slow client and server.

Clearly, an alternate technique that minimized the load on the client would be preferred. This concept involved launching a process on the server that would continuously overwrite a known file name, *LiveImage.gif*, with the latest image. The client at its own pace and processing power would then *GET* this image. The drawback of such a technique was predictable. Random flickering would occur if the server and client would attempt to access the file at the same time.

B.0.1 Launching a Background Process on the Server

With this latest approach in mind, the question involved how to remotely keep a process running. The trouble this time involved the server architecture [1]. Upon receiving a message, the Hardware Control Module would not return until the executable had exited. This was overcome by modifying the server architecture to support threads. In this way, the server in its background would regularly call on an executable to take a sample image. While this worked well, one aspect of system timing was overlooked. This involved the time it took to launch an executable; after all, the command module has to dynamically load all the software modules, but most importantly initialize the camera.

A much simpler alternative was soon realized. This involved having the executable

return but not stop what its doing, i.e have the executable fork out a process. That is, once the executable was called, it would start taking picture and create a dummy file. The forked process would keep acquiring images until some external program deleted the dummy file. This seemed to work very well.

Appendix C

SPG Module Schematics

23 Apr 2001 Last Rev. by D. Seth

The following pages illustrate the schematics of the SPG Module.

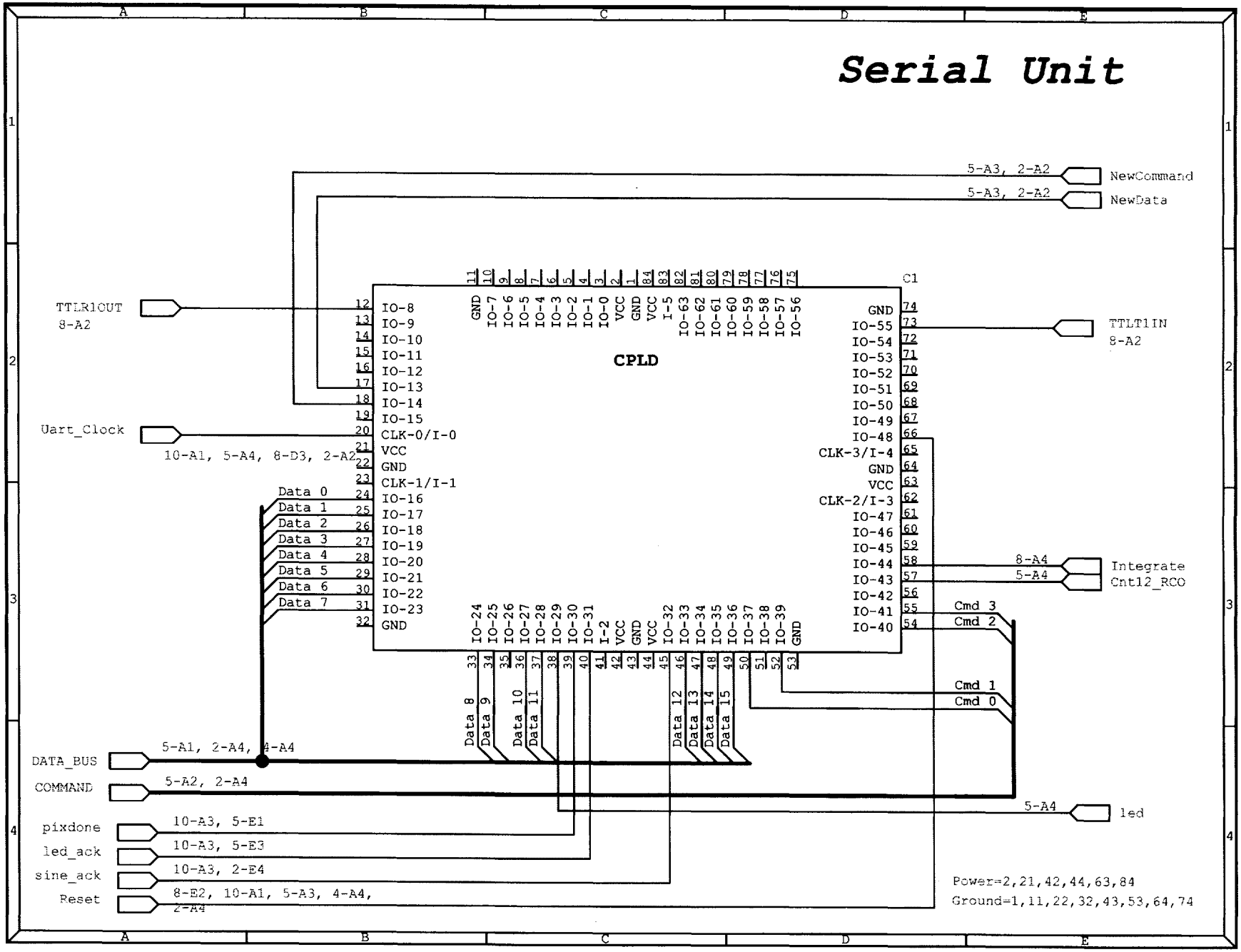

Figure C-1: Page 1 of the SPG Module Schematics

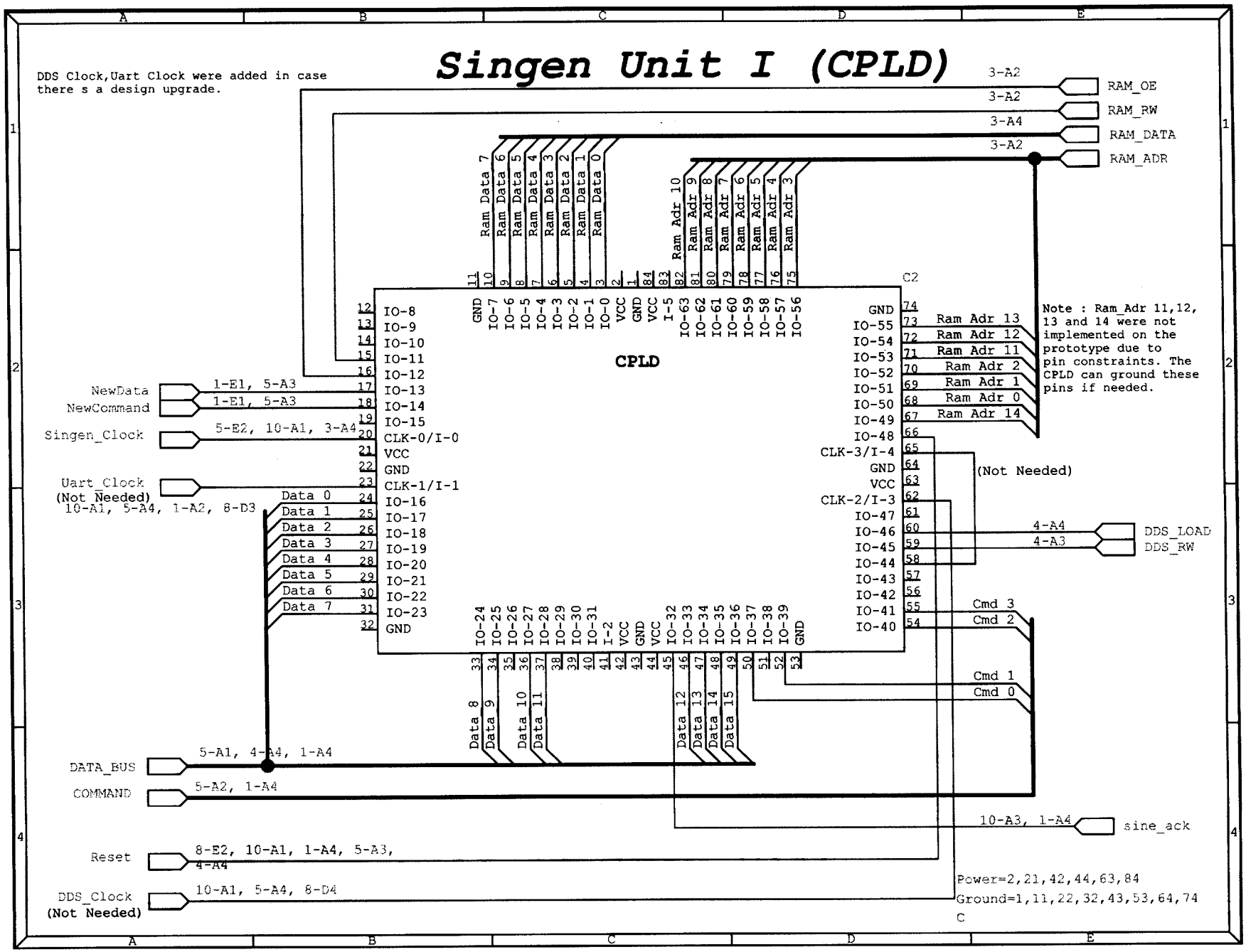

Figure C-2: Page 2 of the SPG Module Schematics

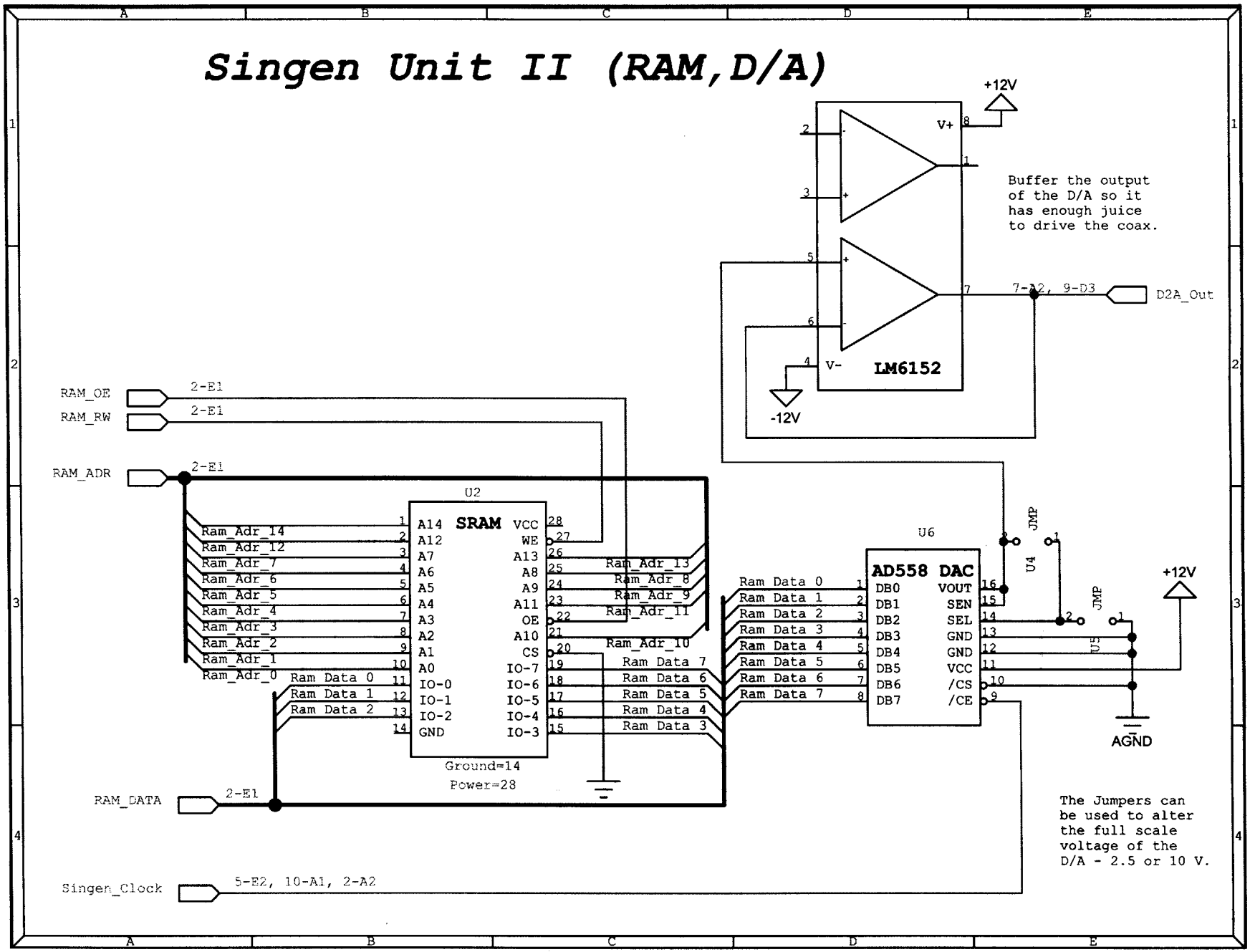

Figure C-3: Page 3 of the SPG Module Schematics

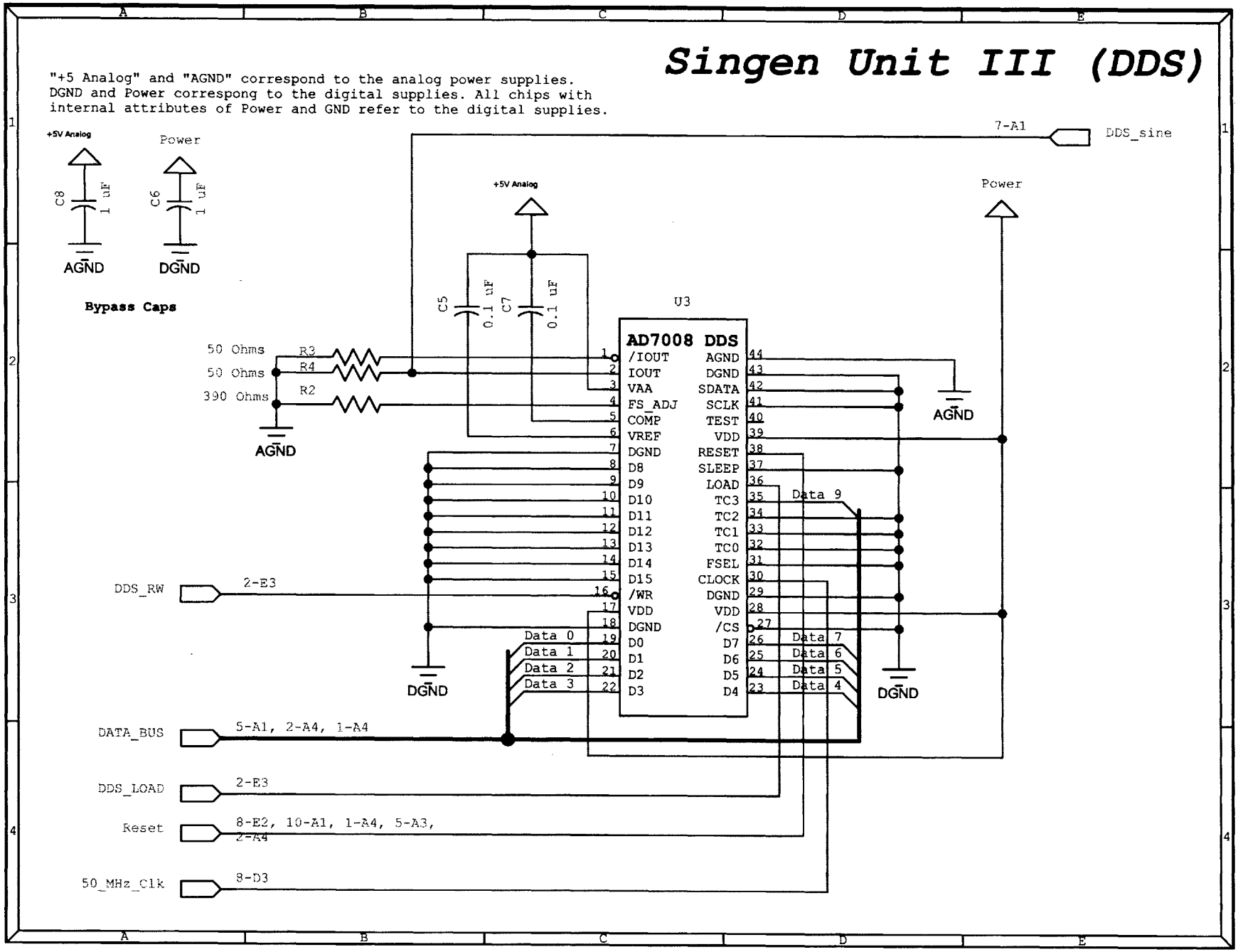

Figure C-4: Page 4 of the SPG Module Schematics

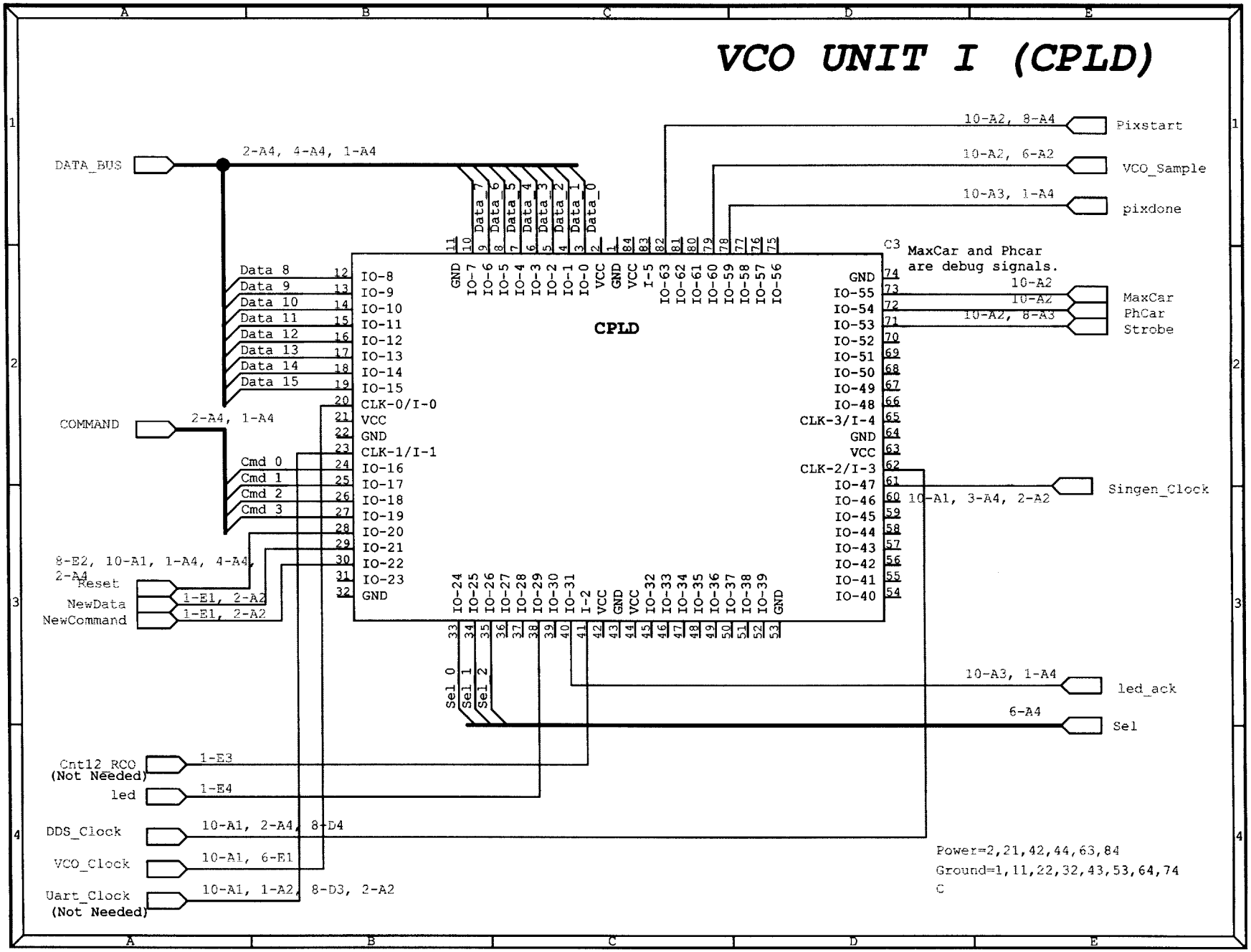

Figure C-5: Page 5 of the SPG Module Schematics

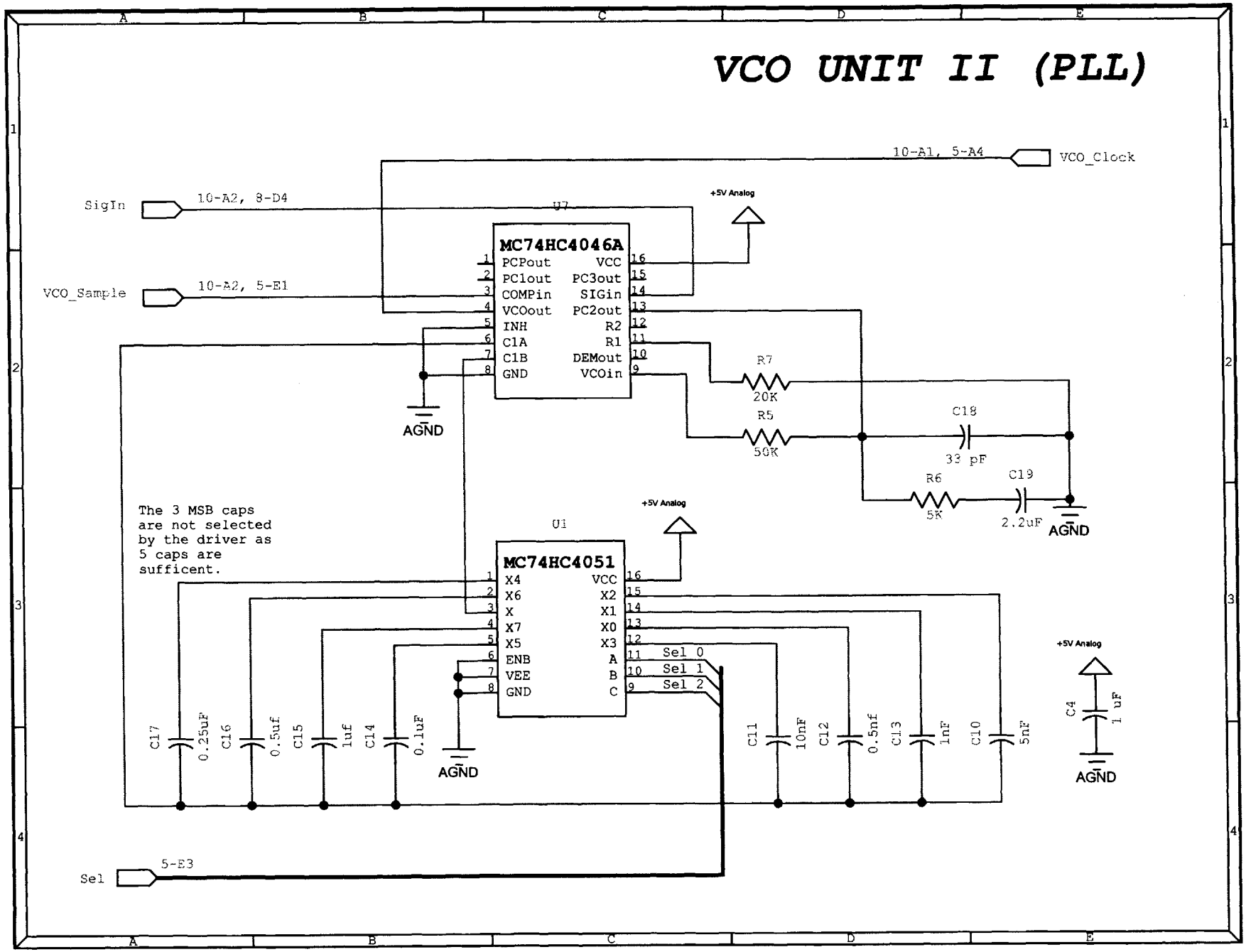

Figure C-6: Page 6 of the SPG Module Schematics

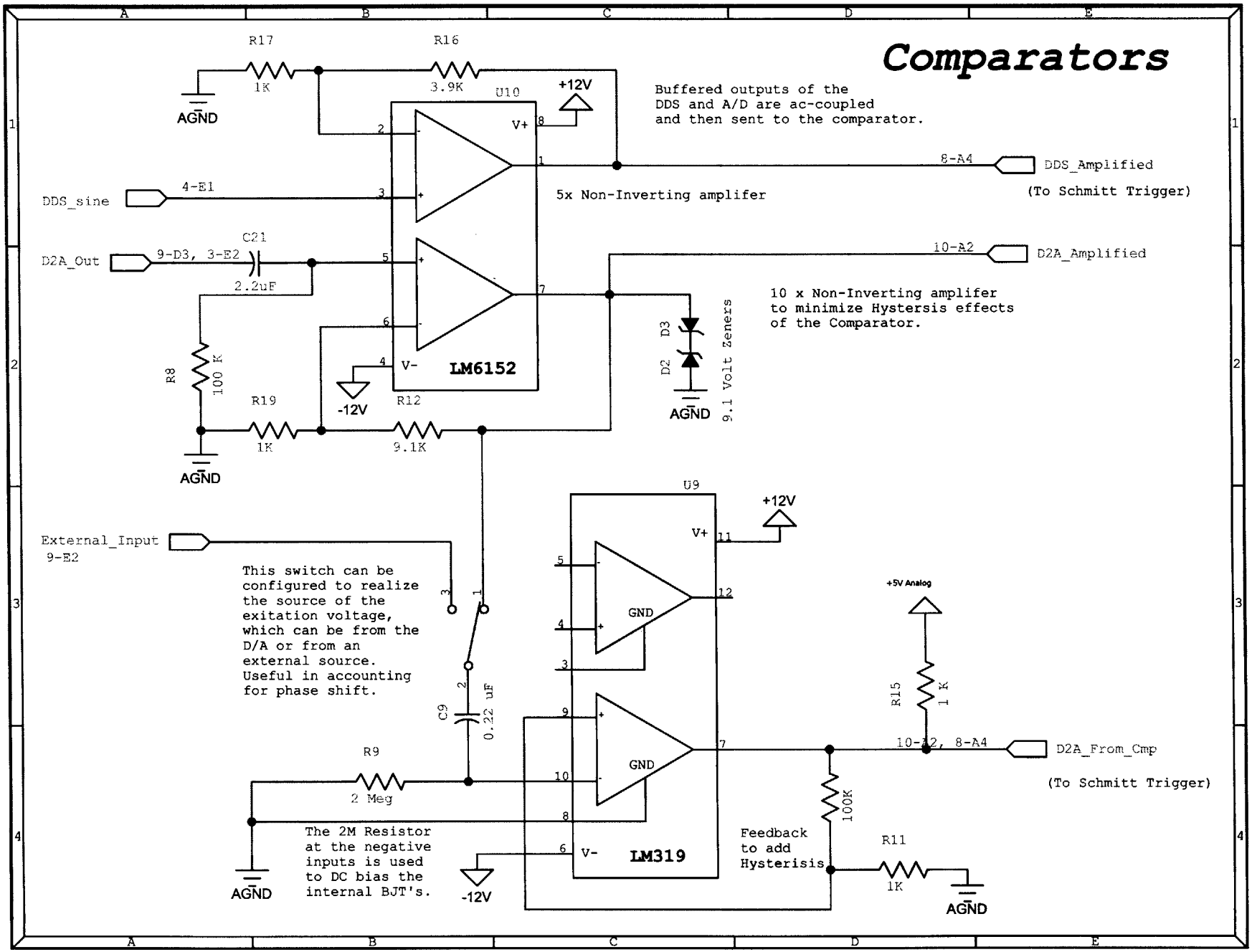

Figure C-7: Page 7 of the SPG Module Schematics

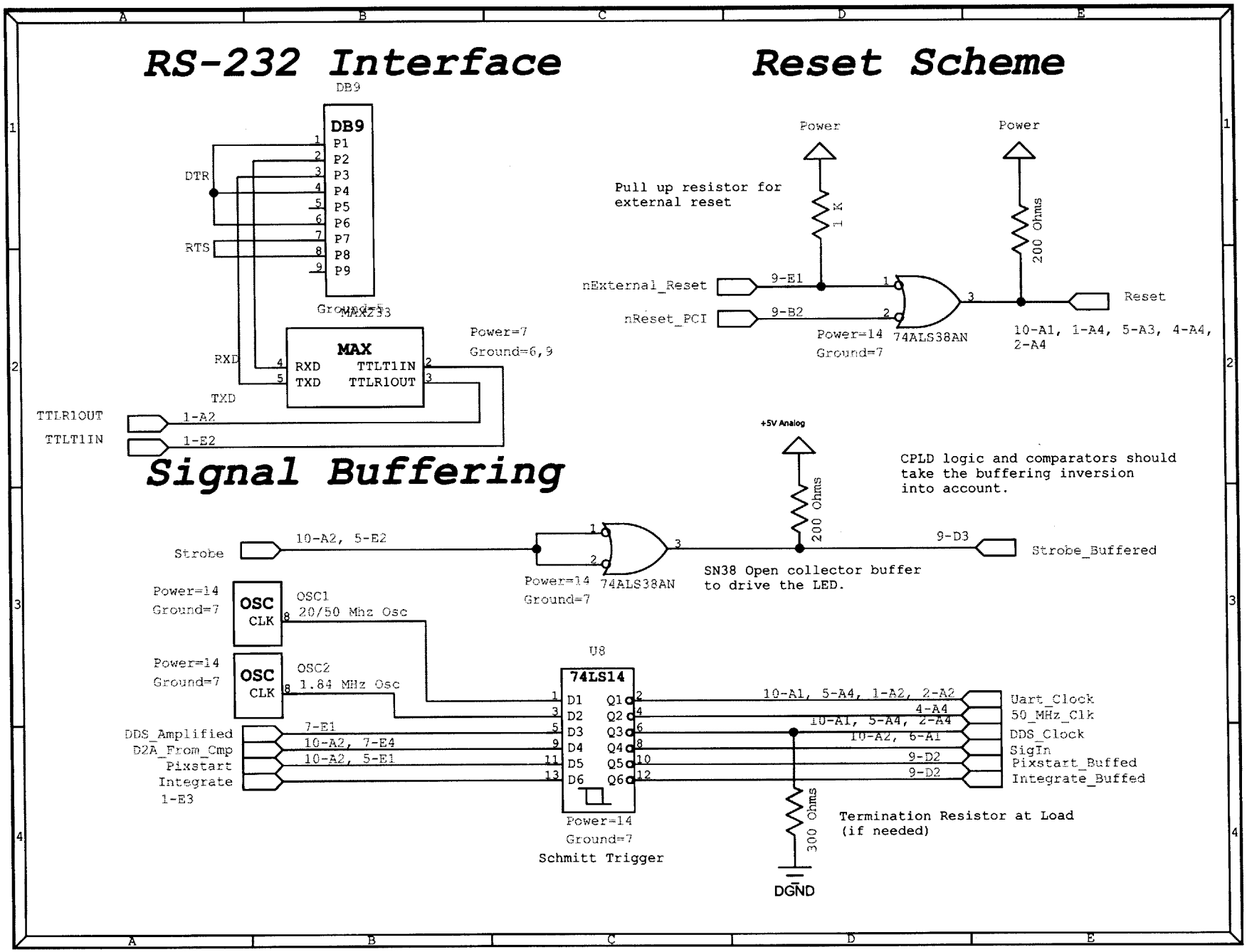

Figure C-8: Page 8 of the SPG Module Schematics

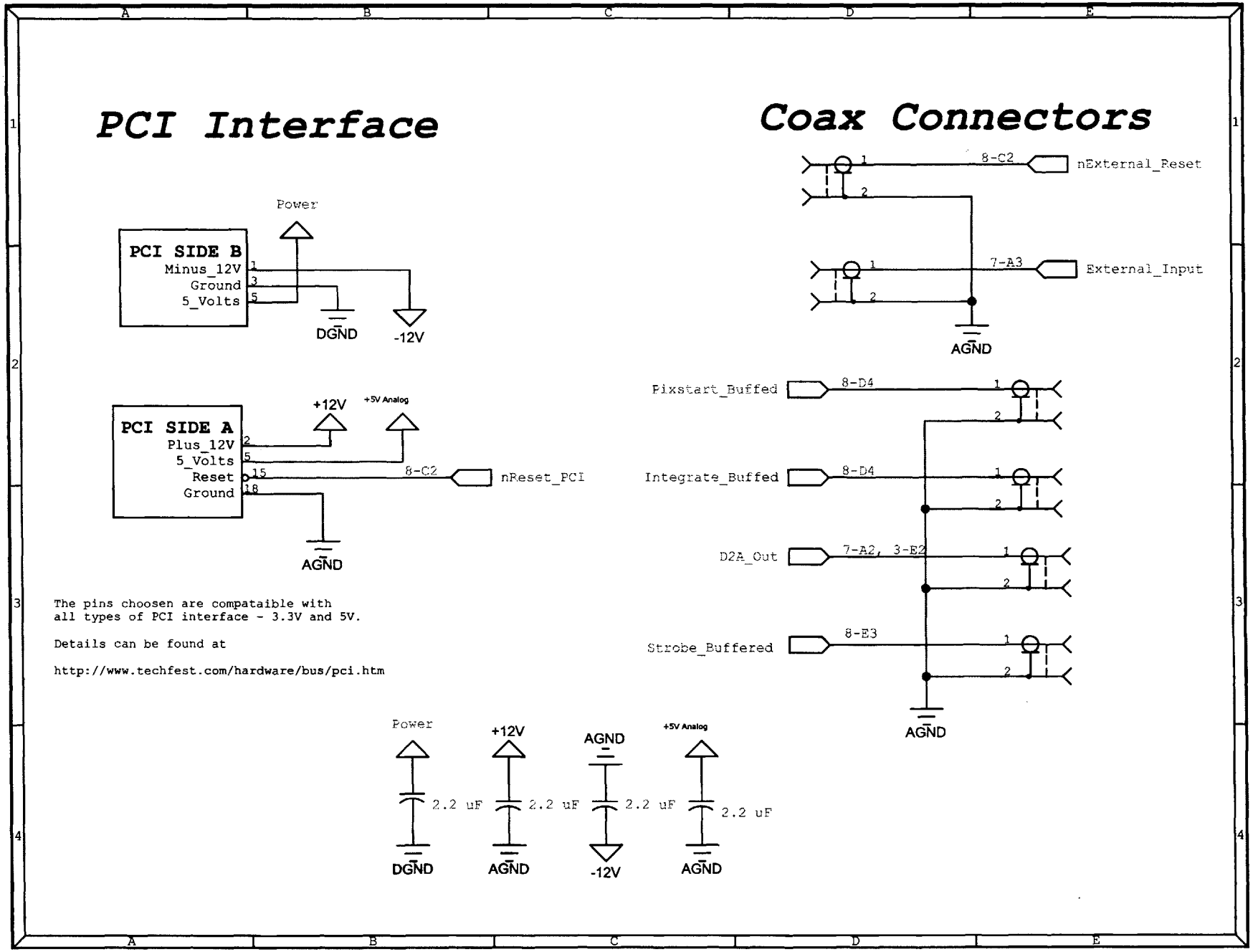

Figure C-9: Page 9 of the SPG Module Schematics

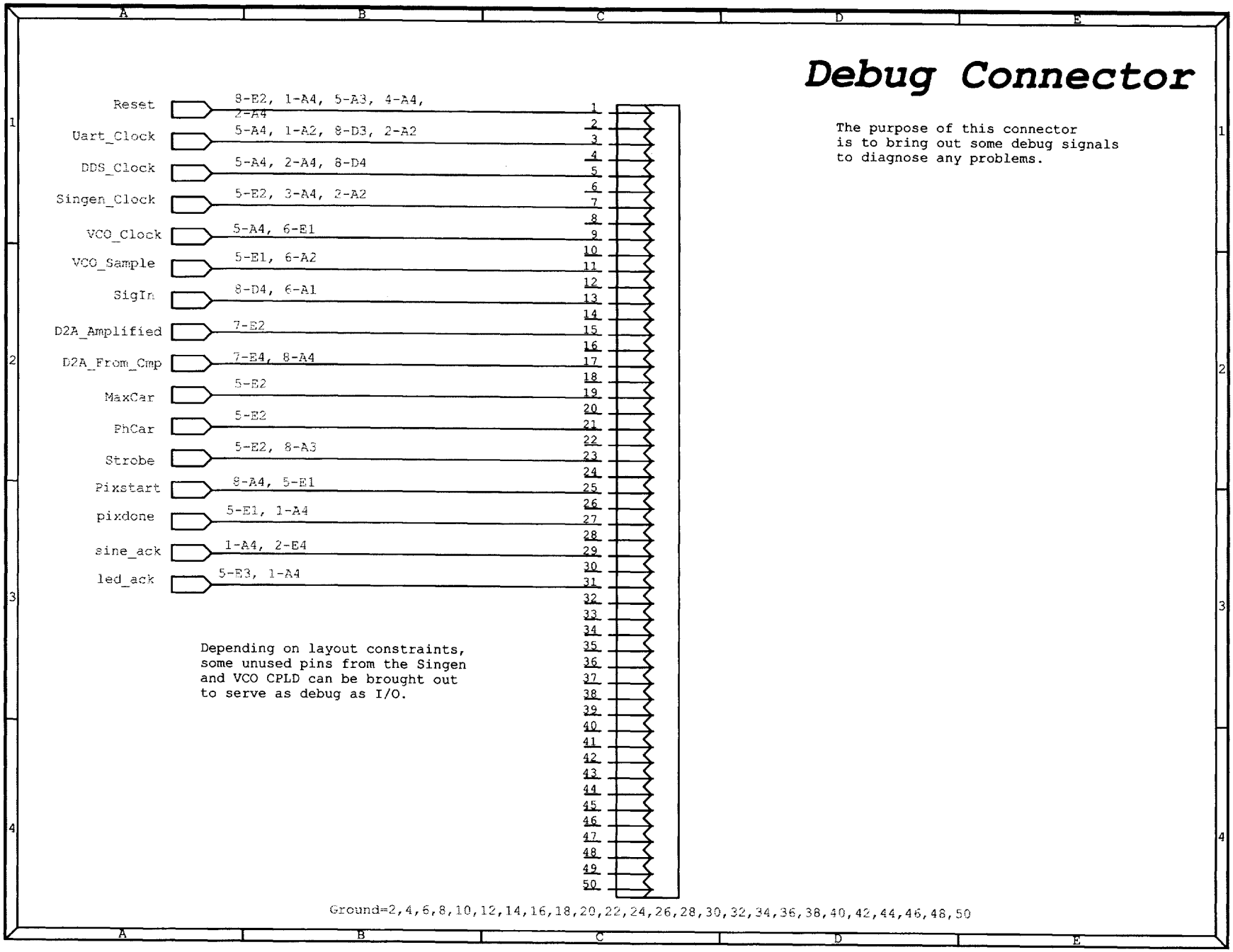

Figure C-10: Last Page (10) of the SPG Module Schematics

Appendix D

Serial Unit VHDL Code

The block diagram and VHDL code of the Serial Unit can be obtained from http://www-mtl.mit.edu/~dseth or by sending an email to dseth@alum.mit.edu. The following is the port description of the Serial Unit.

```
port (  reset         : in std_logic;
        clk           : in std_logic;
        si            : in std_logic;
        so            : buffer std_logic;
        pixdone       : in std_logic;
        led_ack       : in std_logic;
        sine_ack      : in std_logic;
        newdata       : buffer std_logic;
        led           : out std_logic;
        Integrate     : out std_logic;
        newcommand    : buffer std_logic;
        data          : buffer std_logic_vector(15 downto 0);
        command       : buffer std_logic_vector(3 downto 0);
        cnt12car_out  : out std_logic
    );
```

Appendix E

Singen CPLD VHDL Code

The VHDL code of the Singen Unit can be obtained from http://www-mtl.mit.edu/~dseth or by sending an email to dseth@alum.mit.edu. The following is the port description of the Singen Unit.

```
port
     ( clk              : in std_logic;        -- Global System Clock
       Reset            : in Std_logic;        -- Asychronous
       Data             : in Std_logic_vector(15 downto 0); -- Data
       Command          : in Std_logic_vector(3  downto 0); -- commmand
       newdata          : in std_logic;        -- Valid Data
       newcommand       : in std_logic;        -- Valid Command
       Ack              : out std_logic;       -- Acknowledge valid data/command
       Ram_RW           : buffer std_logic;    -- Ram Read/Write
       Ram_OE           : buffer std_logic;    -- Ram Output Enable
       DDS_RW           : buffer std_logic;    -- DDS Read/Write
       DDS_Load         : buffer std_logic;    -- DDS Load
       Ram_Address      : buffer std_logic_vector(10 downto 0);
       Ram_Data_Out     : buffer std_logic_vector(7 downto 0)
      )  ;
```

Appendix F

VCO CPLD VHDL Code

The block diagram and VHDL code of the VCO Unit can be obtained from http://www-mtl.mit.edu/~dseth or by sending an email to dseth@alum.mit.edu. The following is the port description of the VCO Unit.

```
port ( led              : in std_logic;
       reset            : in std_logic;
       clk              : in std_logic;
       DDS_clk          : in std_logic;
       vco_clk          : in std_logic;
       newdata          : in std_logic;
       newcommand       : in std_logic;
       command          : in std_logic_vector(3 downto 0);
       data             : in std_logic_vector(15 downto 0);
       sel              : buffer std_logic_vector(2 downto 0);
       led_ack          : buffer std_logic;
       pixstart         : buffer std_logic;
       strobe           : buffer std_logic;
       n_vco_sample     : buffer std_logic;
       pixdone          : buffer std_logic;
       phcar_out        : buffer std_logic;
       maxcar_out       : buffer std_logic;
       A                : buffer std_logic;
       B                : buffer std_logic;
       clock_out        : out    std_logic
  );
```

Appendix G

Command Module Compilation

The original Command Module loaded a module called data.c that was responsible for the image acquisition process. It established all the SPG Module settings and issued commands to acquire images. However, for the MEMS Characterization System, we need several variations of the cm_getdata executable. For instance, in the case of remote focus, we want the LED to be always on and control exposure via software while having the camera ignore the integrate settings. This case is very different from the slow-motion analysis case, and all cases differ significantly from one another. To make provisions for all these customized executables, the following lines was added to the original cm.getdata.c. Hence, only one module is loaded instead of the "data" module and this differentiates the type of executable created.

```
#ifdef JAVAREMOTEFOCUS
 load_module("javaremotefocus");
#elif JAVAPIXSTART
 load_module("javapixstart");
#elif JAVASLOMO
 load_module("javaslomo");
#elif XDT_AUTOFOCUS
 load_module("xdt_autofocus");
#else
 load_module("data");
#endif
```

The makefile shown on the next page is included to clarify the process of exectuable creation.

G.1 Makefile

```
ND=../nd
CFLAGS= -rdynamic -m486 -Wall -g -O2 -D_GNU_SOURCE -I../nd
#DEBUGLIB=/usr/lib/libefence.a
#DEBUGLIB=
BINDIR=/usr/local/cm/bin

#ALL=cm_getpos cm_setpos cm_getdata cm_focus
ALL= cm_getdata cm_focus cm_getpic cm_slomo

ALL: $(ALL)

cm_getpos: cm_getpos.c lib.c
  cc -o cm_getpos -rdynamic cm_getpos.c lib.c -L/usr/X11R6/lib -lX11 -ldl


cm_setpos: cm_setpos.c lib.c
  cc -o cm_setpos -rdynamic cm_setpos.c lib.c -L/usr/X11R6/lib -lX11 -ldl


cm_getdata: cm_getdata.c lib.c
  cc -o cm_getdata $(CFLAGS) cm_getdata.c lib.c -L$(ND) -lnd
     -L/usr/X11R6/lib -lX11 -ldl -lm $(DEBUGLIB)

cm_getpic: cm_getdata.c lib.c
  cc -o cm_getpic $(CFLAGS) -DGETPIC cm_getdata.c lib.c
     -L$(ND) -lnd -L/usr/X11R6/lib -lX11 -ldl -lm $(DEBUGLIB)

jrf_getpic: cm_getdata.c lib.c
  cc -o jrf_getpic $(CFLAGS) -DGETPIC -DJAVAREMOTEFOCUS cm_getdata.c lib.c
     -L$(ND) -lnd  -L/usr/X11R6/lib -lX11 -ldl -lm $(DEBUGLIB)

j_pixstart: cm_getdata.c lib.c
  cc -o j_pixstart $(CFLAGS) -DGETPIC -DJAVAPIXSTART cm_getdata.c lib.c
     -L$(ND) -lnd  -L/usr/X11R6/lib -lX11 -ldl -lm $(DEBUGLIB)

j_slomo: cm_getdata.c lib.c
  cc -o j_slomo $(CFLAGS) -DGETPIC -DJAVASLOMO cm_getdata.c lib.c
     -L$(ND) -lnd -L/usr/X11R6/lib -lX11 -ldl -lm $(DEBUGLIB)

cm_focus: cm_focus.c lib.c
  cc -o cm_focus $(CFLAGS) `gtk-config --cflags` cm_focus.c lib.c
     -L$(ND) -lnd  -L/usr/X11R6/lib -lXext -lX11  -ldl
     -lm $(DEBUGLIB) `gtk-config --libs`

cm_slomo: cm_focus.c lib.c
  cc -o cm_slomo $(CFLAGS)  `gtk-config --cflags` -DSLOMO cm_focus.c lib.c
     -L$(ND) -lnd -L/usr/X11R6/lib -lXext -lX11  -ldl
     -lm $(DEBUGLIB) `gtk-config --libs`

ndtest: ndtest.c
  cc -o -g ndtest -I$(ND) ndtest.c -L$(ND) -lnd -lm

i: i.c lib.c
  cc -o i -rdynamic i.c lib.c -L/usr/X11R6/lib -lX11 -ldl

clean:
  -rm $(ALL) *.o

install: $(ALL)
  -mkdir $(BINDIR)
  cp $(ALL) $(BINDIR)
```

Bibliography

[1] Jared Cottrell. Server architecture for mems characterization. Master of engineering thesis, Massachusetts Institute of Technololgy, Department of Electrical Engineering and Computer Science, September 1998.

[2] D. M. Freeman, A. J. Aranyosi, M. J. Gordon and S. S. Hong. Multidimensional motion analysis of MEMS using computer microvision. In *Technical Digest of Solid-State Sensor and Actuator Workshop*, pages 150–155, Hilton Head Island, SC, June 1998.

[3] C. Quentin Davis and Dennis M. Freeman. Using a light microscope to measure motions with nanometer accuracy. *Optical Engineering*, pages 1299–1304, 1998.

[4] C. Quentin Davis, Zohar Z. Karu, and Dennis M. Freeman. Equivalence of block matching and optical flow based methods of estimating sub-pixel displacements. *IEEE International Symposium for Computer Vision*, pages 7–12, 1995.

[5] Dennis M. Freeman. Computer microvision for mems. *MIT Micromechanics Group, http://umech.mit.edu/MEMS.html.*

[6] Dennis M. Freeman and C. Quentin Davis. Using video microscopy to characterize micromechanics of biological and man-made micromachines (invited). In *Technical Digest of the Solid-State Sensor and Actuator Workshop*, pages 161–167, Hilton Head Island, SC, June 1996.

[7] Paul Horwitz and Winfield Hill. *The Art of Electronics*, introduction to phase-locked loops, Chapter 9, pages 641–655. Cambridge University Press, New York, NY, second edition, 1998.

[8] Integrated Device Technology. IDT71256SA Datasheet. *http://www.idt.com/docs/71256L_DS_17524.pdf*, 2001.

[9] James Kao, Donald E. Troxel, and Somsak Kittipiyakul. Internet remote microscope. *Telemanipulator and Telepresence Technologies III, SPIE Proceedings*, November 1996.

[10] Somsak Kittipiyakul. Automated remote microscope for inspection of integrated circuits. *MIT, CAPAM Memo No. 96-9*, September 1996.

[11] Tim Berners Lee. Basic HTTP as Defined in 1992. *http://www.w3.org/Protocols/HTTP/HTTP2.html*, 1996.

[12] Erik J. Pedersen. User interface for mems characterization system. Master of science thesis, Massachusetts Institute of Technololgy, Department of Electrical Engineering and Computer Science, January 1999.

[13] Manuel Perez. Java remote microscope for collaborative inspection of integrated circuits. *MIT, CAPAM Memo No. 97-5*, May 1997.

[14] Ramón L. Rodríguez. Performance measurements of mems analysis system. Master of engineering thesis, Massachusetts Institute of Technololgy, Department of Electrical Engineering and Computer Science, June 1999.

[15] Danny Seth and Donald E. Troxel. CPLD Module. *http://sunpal2.mit.edu/6.111/s2001/cpld_module/cpld_module.html*, 2001.

[16] Xudong Tang. A complete mems analysis system and implementation. Master of science thesis, Massachusetts Institute of Technololgy, Department of Electrical Engineering and Computer Science, May 2000.