USER INTERFACE FOR MEMS CHARACTERIZATION SYSTEM

by

Erik J. Pedersen

A thesis submitted in partial fulfillment of the requirements for the degree of

Master of Electrical Engineering and Computer Science at the Massachusetts Institute of Technology

January 1999
[February, 1999]

Author _____
Department of Electrical Engineering and Computer Science

January 15, 1999

Certified by _____
Donald E. Troxel
Professor of Electrical Engineering
Thesis Supervisor

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

USER INTERFACE FOR MEMS
CHARACTERIZATION SYSTEM

by Erik J. Pedersen

Submitted to the Department of Electrical Engineering and Computer Science
on January 26, 1999 in partial fulfillment of the
requirements for the Degree of Master of Science in
Electrical Engineering and Computer Science

Abstract

The purpose of this work is to present a graphical user interface designed for remote operation of a MEMS (Micro Electromechanical Systems) characterization system. A MEMS characterization system is a system which is developed to collect and analyze data on a MEMS device. Presently, there are systems and tools for these purposes but they are not remotely accessible. Therefore, the work done here involves developing a graphical user interface that permits access to the system remotely. Furthermore, the graphical user interface architecture presented here allows a user to customize the user interface in terms of appearance and functionality. This paper explains how the architecture works and also describes how the user can create a custom graphical user interface in order to operate a MEMS characterization system remotely.

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

TABLE OF CONTENTS

LIST OF FIGURES

Number *Page*

ACKNOWLEDGMENTS

I wish to acknowledge and give thanks to some of the many people that have made the completion my graduate work here at MIT possible as well as enjoyable. In an effort to remember those who have impacted and helped me, I do not intend to discount the many who have impacted me in ways that I may not have seen or noticed.

First, I would like to thank Donald Troxel for inviting me to be a part of his research. His knowledge is deep and he has helped me to look at my work and the work of others in a more detailed manner. I have to thank him for allowing the research that I have done to be my own. His suggestions have always been insightful and his questions always hit the heart of the matter. Also, thanks goes out to Michael McIlrath for asking the tough questions during our weekly meetings.

I must also not forget to give thanks to my officemate, William Moyne, who has been a constant source of laughter and great conversation ever since I met him. His laughter is contagious and I look forward to working with him at Virtual Ink. Jared Cottrell was also a constant companion in my classes as well as in my research here. He is an invaluable source of knowledge and always has great insight on any subject. Danny Seth who joined our group just this past semester has contributed more than was required of him. His laughter and positive attitude have been a constant encouragement to me. Thomas Lohman and Myron Freeman (Fletch) with their wit and unique perspectives on life have been sources of laughter and fun for our entire office area. I will miss the many lunches we had together.

I thank my sister Cheri for all the meals she fed me and I cannot forget the constant encouragement of my bother-in-law Keith. They have both been gracious enough to allow me to live in their house and eat their food. I will never forget that. A huge thanks go out to my parents, Robert and Linda Pedersen, who never fail to welcome me home and who are my model of a true marriage commitment.

I have to thank my friends back home who have really walked the trenches with me during these years of school. Thanks to Bob, Frank, and Jamie for being the best roommates that I will ever have. Thanks to Jon for being an unfailing friend all of my life. The laughter I share with him is never ending.

Finally, but most importantly, I want to thank my friend, my Savior and my Lord, Jesus Christ for His unfailing love for me and for His grace. None of these words have been written without His allowance.

Chapter 1

INTRODUCTION

1.1 MEMS

MEMS (Micro Electromechanical Systems) are micro electromechanical devices incorporating small components that react to electric stimuli in a mechanical way. They are produced and manufactured using similar techniques as those used in the design of VLSI chips. More specifically, there are an abundance of materials used to create MEMS including semi-conducting and conducting materials. Also, MEMS are laid out in packages that look similar to an electronic microchip. The main difference is that there are actually small machines placed in the package of a MEMS device as opposed to an electronic device inside of a VLSI chip. The devices put into MEMS chips can include various components from small switches to gyroscopes that detect motion and tilt. As a result, the machines placed in the packages must be developed and tested for response and durability. Computer microvision is a good tool to use for the analysis during the testing and development stages of the design process.

1.2 Computer Microvision

Computer microvision is a new field that has been growing as the research in the area has continued. In essence, computer microvision is the use of a system incorporating a microscope device and a computer system. The computer system has the function of processing commands and data for the purpose of research or production. One example of the use of such a system can be seen in the study of the tectoral membrane of the inner ear [1,2]. A second example of where this type of system may be utilized is in the analysis and development of MEMS.

1.3 Components of a Microvision System

The components of a computer microvision system consists of a device that magnifies the appearance of an object under study, a computer to implement control and data processing, a device that can collect and transmit visual data, and a component that can initiate excitation of a MEMS device. These components imply, as is the case with the present applications, that the device object under study is too small for direct observation. Thus, in order to observe the object, or some aspect of it, the visual data collector must be able to sense the object motion with some minimal amount of accuracy. There has been research done to determine a fast and reliable algorithm in order to accomplish this sort of detection at a sub-pixel level [3].

The computer system must have the ability to interface with the visual data collector in order for the data to be utilized in understanding what the device maybe doing. The visual data collector consists of a camera having a limited pixel resolution and an interface to the computer system that must control the camera. 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 and to control the camera when it is taking images. Parameters such as exposure time and when the picture should be taken must be controllable by computer as well.

Another system component that is needed when a device is moving too fast for the camera to image correctly is a strobe pulse generator. The strobe pulse is used to illuminate the moving device for a short enough period of time so that the motion of the device will not blur the image taken with the camera. Furthermore, the strobe pulse must be synchronized with both the camera image acquisition and the excitation signal going into the device.

In addition, the computer must interface with a device that produces an excitation of the object under study. In the field of MEMS, this excitation often comes in the form of an excitation voltage. When a voltage is applied properly to a device, it should react in a desired manner. The way the device reacts to the excitation is therefore the most important aspect under study.

The computer is also in control of the stage position of the microscope. As a result, the computer has enough information about the system to aid in the testing process as vital tool. 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 result of the specified input since it has an interface with the camera system. Thus, the desired response to a specific input can be compared to the measured actual response of the MEMS device.

1.4 User Interaction

A very functional and useful prototype system is in use at the present time. The system is used to collect data and access the tools used to analyze the data taken. The main interface between the user and the hardware is a control kernel which has been labeled "Experiment Control Kernel" or ECK for short. There are projects underway that attempt to raise the level of abstraction above the level of a command line only interface (i.e. ECK). To use ECK requires the use of scripts to control the hardware and the analysis tools. Essentially, the newer interfaces still use scripts to execute programs on the hardware, but the scripts are not exposed to the user as in ECK [5].

The other aspect of the present system that is limiting is the fact that it can only be used as a local system. The fact that it has been designed to run locally makes it difficult to use the system outside of directly interacting with the computer to control the hardware. There are benefits to desiring to use to the system remotely. First of all, in an ideal system, there should be no contact between a 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 around the device. Thus, it is best to keep the device in a controlled environment. In order to avoid having to work inside 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 the collaboration of these fields into a group that can work on the design of the same device, remote access to the system is a much better alternative than having to gather all of the engineers in the same place. Furthermore, the architecture of the 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 system. Thus, remote access of the system naturally fit in with the present architecture.

1.5 Relevant Work

There are examples of work that has been completed by other students doing research in this area. It is important to understand the relevant work so that the ideas can be used to help with the present work as well as avoid the overlap of ideas among the research.

1.5.1 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 [6,7,8,12]. The primary differences between that system and the present microvision system have to do with the fact that with the microvision system, the devices under test are not in motion. Thus, the tools used to analyze the devices are different for the systems. In particular, the method of taking images of the MEMS devices requires careful timing of the excitation with the strobe pulse and image capture sequence. In addition, the remote microscope uses an algorithm to focus the image. As of now, there is no such algorithm for the microvision system. Also, there is an input device when it come to observing MEMS. Thus, the tools must interact with the data and with the input to the device. There are, however, some similarities between the systems. Most importantly, both involve the use of a microscope and camera system connected to a controlling computer. Thus, the model for remote access can be understood for the remote microscope and can then be applied to the MEMS system.

1.5.2 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 on that can detect sub-pixel movement of a 3D object and one that can analyze the motion in a certain region of interest

on a set of images given the proper information [3,4]. With these two tools working together, a set of data taken from a MEMS device can be analyzed for various factors of interest. Information can be gathered includes the motion of the device under test in the x, y, and z directions. Graphs can be produced to analyze the phase differences between parts as well as the relative amount of motion in the region under study [1].

1.5.3 Present User Interface

The original interface used for the present system is a shell command interface where all of the setup commands must be explicitly typed into a command line. This interface is called ECK [5]. The setup parameters include excitation frequency, amplitude, the placement of the strobe to capture a phase of the motion and the microscope system variables. There are also commands that interfaced with the camera exposure time and set the image timing. These commands allow the user to take sets of pictures at different levels of focus for a certain excitation frequency while capturing eight phases of the motion of the device. With this data, the separate tools used to analyze the device must be invoked to get the detailed information about the device behavior.

Presently, there are multiple interfaces being developed which build upon ECK. They are graphical interfaces that can be used to set all of the appropriate parameters for the system. Additionally, the interfaces can initiate the data collection as well as the data analysis process. The user is therefore able to use the system without having to write scripts and with less of an understanding of exactly what the hardware must do.

1.5.4 User Interface Reference Designs

There have been projects within the past few years that have dealt with developing user interfaces for accomplishing various tasks. The first, and likely the most relevant, is the project completed by Manuel Perez on implementing an interface for the remote microscope project mentioned previously [8]. He used a Java programming environment with the data collection and control of the microscope resting with the computer at the microscope side of the system.

Another reference source for understanding how interfaces can be implemented is work done by Matthew Verminski on an interface across the internet called EmSIm. The tool allows a user to input parameters on an interconnect line to be used in a small integrated device. After these parameters have been set , they are submitted across the network to a machine that handles the calculations to find the time to failure of the interconnect line or lines under study [9]. Since there is a user interface and server system utilized in EmSIm, the methods used to design the system can be used as reference ideas for the present project.

A third source of information on interfaces comes from Debashis Saha. He worked on a project that attempted to provide a standard specification for CGI based CAD tools. Thus, with the specifications, a client can invoke tools across a network using an HTTP call. This idea of creating standard specifications can be useful in the design of the user interface for this work.

1.6 Thesis Statement

The observation and manipulation of MEMS are two of the main steps in designing these devices. In order to improve upon this process, the next logical step in its evolution would be to implement these steps such that they can be done remotely and by multiple users. The purpose of this work is to address these issues from a user interface standpoint. There have been tools and methods developed that allow for the testing of MEMS. However, the implementation is local only and does not incorporate an interface that allows easy access to the analysis tools and controls of the system. Thus, this research focuses on implementing a method for creating a user interface to use the tools available on a MEMS characterization system.

1.7 Organization of Thesis

This chapter has provided a general overview of the tools required to observe and analyze MEMS. Also, it briefly discussed the idea of requiring an interface for the user interact with the system. It also discussed some previous uses of user interfaces that were relevant to the work done for this thesis.

The next chapter will provide an overview of the client and server architecture developed to create a MEMS characterization system. Most of the focus will be on the client side of the system because this work is involved in that particular aspect of the system. Only when it is necessary in the understanding of the client architecture will the server architecture be discussed. Chapter 3 will present some of the ideas behind developing user interfaces in Java. Chapter 4 will describe the idea of having a customizable user interface as well as how that is implemented in the client. Chapters 5 through 10 will present the components of the client software. The presentation is done in a logical grouping of the modules, not necessarily grouping based on code structure. Chapter 11 will briefly discuss some custom components written specifically for the client software. Chapter 12 will discuss another module dealing with the idea of having a programmatic interface for the system. Finally, Chapter 13 will conclude and discuss some areas that can be developed and improved upon in the future.

As a supplement to the information in this paper, there are two appendixes. Appendix A contains information for the user describing each of the functions that can be accessed within the client software when a user interface is created. Appendix B explains the installation process for the client software.

Chapter 2

SYSTEM OVERVIEW

2.1 Server Overview

The server can be divided into two distinct and crucial parts. First, there is the hardware of the server. Second, there is the software the must run on the hardware. These two integral parts of the server will be discussed in this section.

2.1.1 Hardware

The proposed 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 connected to 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. Using this architecture, there is still a collection of hardware components that are constant in any of these MEMS characterization systems. They consist of a computer system that is in control of a microscope, a camera system, and a strobe/excitation device for analysis of the MEMS part. The term computer system is used here in its most broad sense. This is true because these devices may not actually be directly connected to one computer. However, there must be some control center. In addition to the above components, there must be hardware that allows for storage of data as well as the means to retrieve the data when needed. Furthermore, there must be a system that can handle the analysis of the data taken from the MEMS devices. The system that is being put together at this time is not distributed since the hardware is actually driven by computer that runs the server. Also, the database presently resides on the same machine. The data analysis system may run on a different system. The hardware included in the server side of the system must encompass all of these separate subsystems through a transparent means of interaction.

2.1.2 Software

There are three main modules of software that enable the server 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 is what acts as the network server for the client. It allows the client to interact with the server using the standard HTTP protocol. The server software that is running as of now in the system is Sun's Java Web Server. The reason that Sun's web server has been used is that it is the only web server that supports the use of Java programs called servelets. It is these servelets that overcome the inability of the standard HTTP interactions to deal with state. They are Java classes that run on the server side instead of being downloaded over a network the way that a standard applet is downloaded and run locally. Thus, they extend the server's ability to accomplish tasks which can be initiated remotely. Thus, the server software is what controls the interface between a client and the server where the separation between them consists of a network.

The second software module is the sum of the Java classes that make up the functionality of the MEMS characterization system. They handle the remote login of a client as well as all of the subsequent interaction between client and server. For details on exactly how these modules are set up, see Jared Cottrell's thesis entitled "Server Architecture for MEMS Characterization System" [12]. In general, there is a module that handles any messages sent from the client to the server appropriately. 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. The final main module of functionality includes classes that have the ability to access a database that will eventually reside either locally or remotely. The database will contain images and analysis data which are required for the characterization of the MEMS devices.

The third software module that has been included in the server side of the system is the database system as well as the drivers that allow the Java classes to interface with the database. The server architecture has been laid out to allow the server to use any SQL (Structured Query Language) [17] database that can use the Java Database Connectivity (JDBC) API [16]. Many databases have drivers

that allow them to use the API. Thus, the idea is that any database can be used as long as it adheres to the protocols and works correctly using JDBC.

2.2 Client Introduction

The client software is the basis for the work described in this paper. It consists of a multiple module system written in Java and it is intended to be run remotely through a web browser. The link between the client and the server in the MEMS characterization system is a messaging protocol which will be described in detail in the next subsection. The part of the client that the user can see and interacts with is the graphical user interface (GUI). It results from a user written settings file which is used to create a custom user interface. This process will be described in detail later in this paper. However, some examples of user interfaces with their respective settings files will be presented in the last subsection of this main section. Prior to those examples, some of the higher level ideas that have been utilized in the design of the client will be presented.

2.2.1 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. For a more detailed description of this protocol, see the HTTP protocol specifications [13]. There are certain types of requests that a client is allowed using the HTTP protocol. The most often used requests are those made by web browsers when they request web pages. This request is called a GET where there are parameters outlined in the header of the request. There is also a request called a POST which is similar to a GET request except that there is more flexibility in the length of the information attached to it. It is through the POST that a client is expected to send information to the server. In accordance with this idea is the use of the POST command in the MEMS characterization system. The POST command is used to send messages from the client to the server so that the server can accomplish tasks as requested by the client.

The idea used is communication based on a messaging protocol. Previous work done using this idea of sending messages between a client and a server to accomplish work and to provide a standard

interface for communication includes work done by John Carney on CAFÉ [10] as well as work done by Manuel Perez on the Remote Microscope system [8]. This idea also has some roots in the development of large scale multi-processor computers where communication must occur between nodes of the system to keep the parallel processes synchronized. The main advantage of such an interface as used in the MEMS characterization system is that there is a well defined interface to connect the client and server. The interface is well defined by the previously mentioned HTTP protocol and by the message protocol definition as developed specifically for the MEMS characterization system.

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 simply text with well defined formatting. When the text is sent by 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 type of 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. On both sides of the system there are modules that accomplish message handling. The handling has to be done in both directions for the client and this will be explained in more detail later in this report. The messages that are included in the minimum message set as defined for the work done on this project allows a client to login, take control of the server's resources, take pictures of the device under test as directed by the client, and eventually initiate the analysis of the data.

2.2.2 Java GUI Interface

Java was developed by Sun Microsystems and is an example of an object oriented language that has been designed from the beginning as an object oriented language. In essence, the idea behind an object oriented language is that it allows programmers to create objects or modules that can be utilized and interacted with in ways that are cognitively in line with how we as people interact with objects around us. These objects are defined by classes in Java and when needed are instantiated within the

Java code. The classes that get instantiated can then be used through the methods defined in the classes in order to accomplish work. To find out more about Java, see the white papers on Java from Sun Microsystems [11]. Also, a good book to read is *Java In a Nutshell*, 2nd edition by David Flanagan [14].

The idea of creating a user interface using Java is not a new one. In fact, there are many applications that have already been written that utilize Java in order to create a graphical user interface. The reason that Java has been used to accomplish this task is that Sun has included an abstract windowing toolkit (AWT) in the Java language that can be used to create user interface components. These components are used primarily as the front end for the user interface for this work. In addition, Sun has developed an extension of the AWT which has components with enhanced functionality. This package is called *Swing* and it will be described in more detail in the Java basics section of this paper.

The basic idea behind creating user interfaces using Java is that there are components created such as buttons and menus which reside in windows. These buttons and menus are meant to be acted upon by the user. As a result the interface is supposed to react to the user input. This is the way a user interface works at a fundamental level. Exactly how this was done in this work will be explained in more detail later in this report.

2.2.3 Example User Interface

When dealing with user interfaces, it is the user that ultimately has to utilize the interface. Because of this fact, it would be desirable for the user to have a way to control the interface that they must use. In light of this, a simple method of modifying the user interface for the MEMS characterization client has been developed. The user can modify the interface through the use of a script file that must be located on the server. The details of the script and its use by the client will be explained later in this paper. Basically, the user can create menus with items, buttons, and data windows by using the proper grammar in the setup script file. Each item in the menus and each button can be assigned to a function from a list of primitive functions available. Furthermore, each data window is assigned a name by the user.

This work is based on this idea. It can be theoretically expanded to allow the user much more control over the interface. However, the idea that the user can have this sort of control is important particularly for this present application. The reason for this is that there are basically two ways in which the MEMS characterization system can be used. The first is that it can be used to acquire data from the system and store it in a database. The second use is the actual analysis of the data taken at another time. Thus, the interface can be setup to accommodate the different tasks.

In order to help the reader understand what this client software allows the user to do, an example of a user interfaces along with the settings file that creates the user interface will be presented here. The example shown in figure 2-1c is a user interface with buttons to control the microscope parameters as well as the server hardware settings. Furthermore, there are two menus included in the interface that allow the user to initiate some control oriented functions as well some useful tools.

The figure below contains three main parts. The first part of figure 2-1 is the what the user would see after pointing the browser to the MEMS server URL and allowing the client to load. This is figure 2-1a. As can be seen, there is a login/logout applet in the browser's window. When the login button is pressed, the user sees the a desktop that looks like the second part of figure 2-1. This is figure 2-1b. At this point, the user is presented with the settings files that are present on the server and the user must choose one and push load to continue. After that is done, the result is figure 2-1c where the interface is loaded and ready for use.

Looking at figure 2-1c of the user interface and the settings file below figure 2-1c, it is easy to see how each frame is created. The first line in the settings file creates the frame that contains all of the control buttons and the menus. As can be seen in both the settings file frame creation line and the frame, the title of the frame is the same - *Demo Interface.* Further down in the settings file, there is another frame creation. The second frame is the frame that holds the data. As with the *Demo Interface* frame, both the line in the settings file that creates the data frame and the data frame in the interface have matching titles in them – *Data Window.* These facts are merely being pointed out so the reader can connect how the lines in the settings file correspond to components in the interface. The second line in the settings file creates a menu bar and places it in the *Demo Interface* frame. As a side note, the menu bar definition

in the settings file also has a title that is not actually displayed. Next, there is a menu created followed by the menu items in the menu. Each menu item contains a third argument which indicates the function associated with menu item. The second menu that is created in the settings file and the associated menu items can be connected visually with the menu that is pulled down in the user interface in figure 2-1c. After the menus are created in the settings file, the next component created is a panel frame that is created to hold all of the button panels. As can be seen in the definition of the panel frame in the settings file, the panel frame is added to the *Demo Interface* frame. Furthermore, the panel frame has additional arguments that represent rows and columns of panels that will appear in the interface. Then, the settings file defines each of the panels with their respective titles as well as the buttons that are added to the panels. As with the menu items, each button has a final argument that associates the button with a function. Skipping past the rest of the panels and buttons, the last four lines of the settings file defines two user defined functions. These functions are actually associated with two menu items in the tools menu in the interface. Each of these components will be explained in more depth later in this paper. This example has been given just to help the reader see an example of a user interface along with its associated settings file. As a final note, every interface displays the *MEMS Status Window* frame. Thus, there is no line the in the settings file that creates the *MEMS Status Window* frame. This will also be explained in more detail later in this paper.

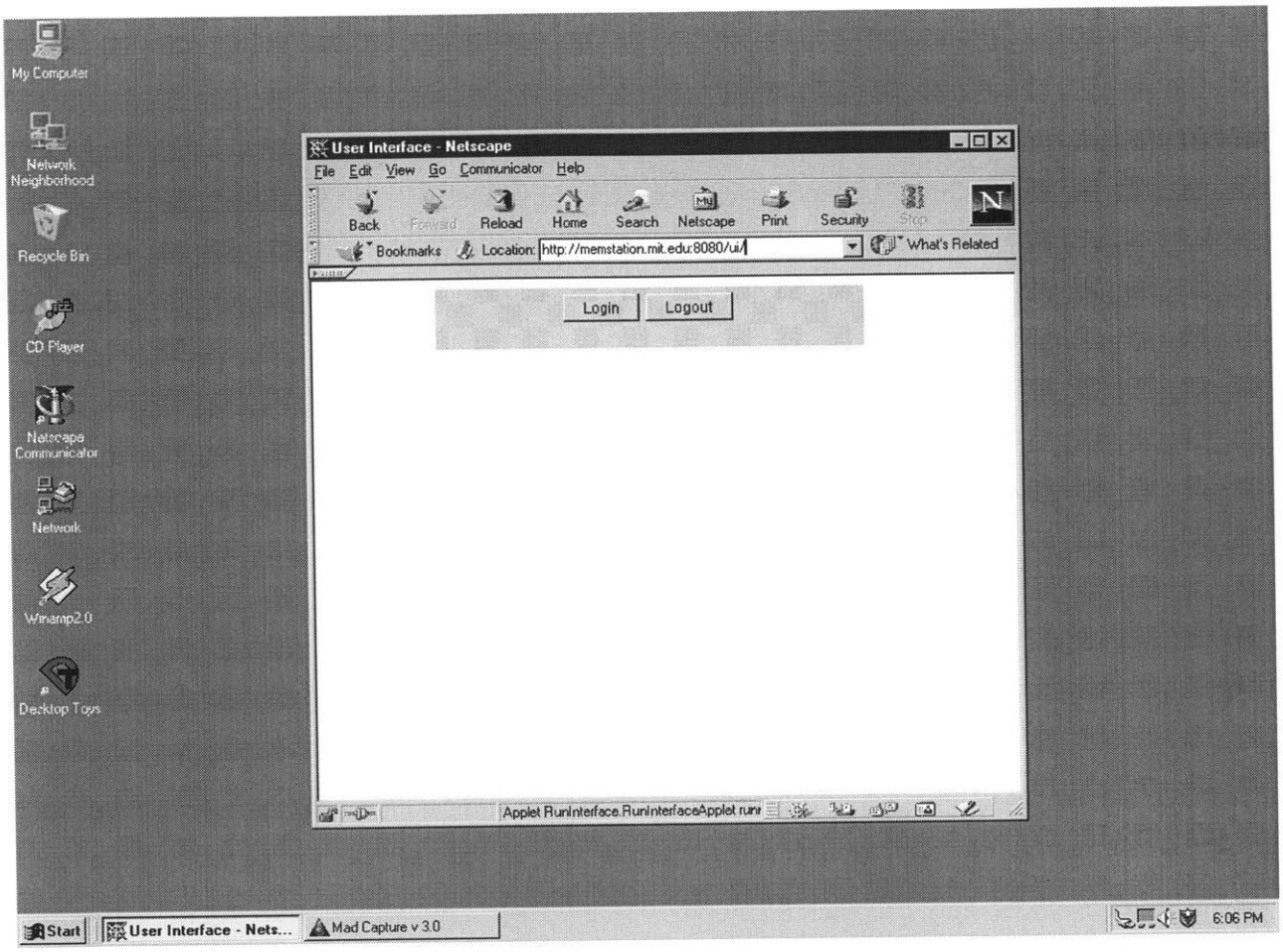

Figure 2-1a: Login Step 1

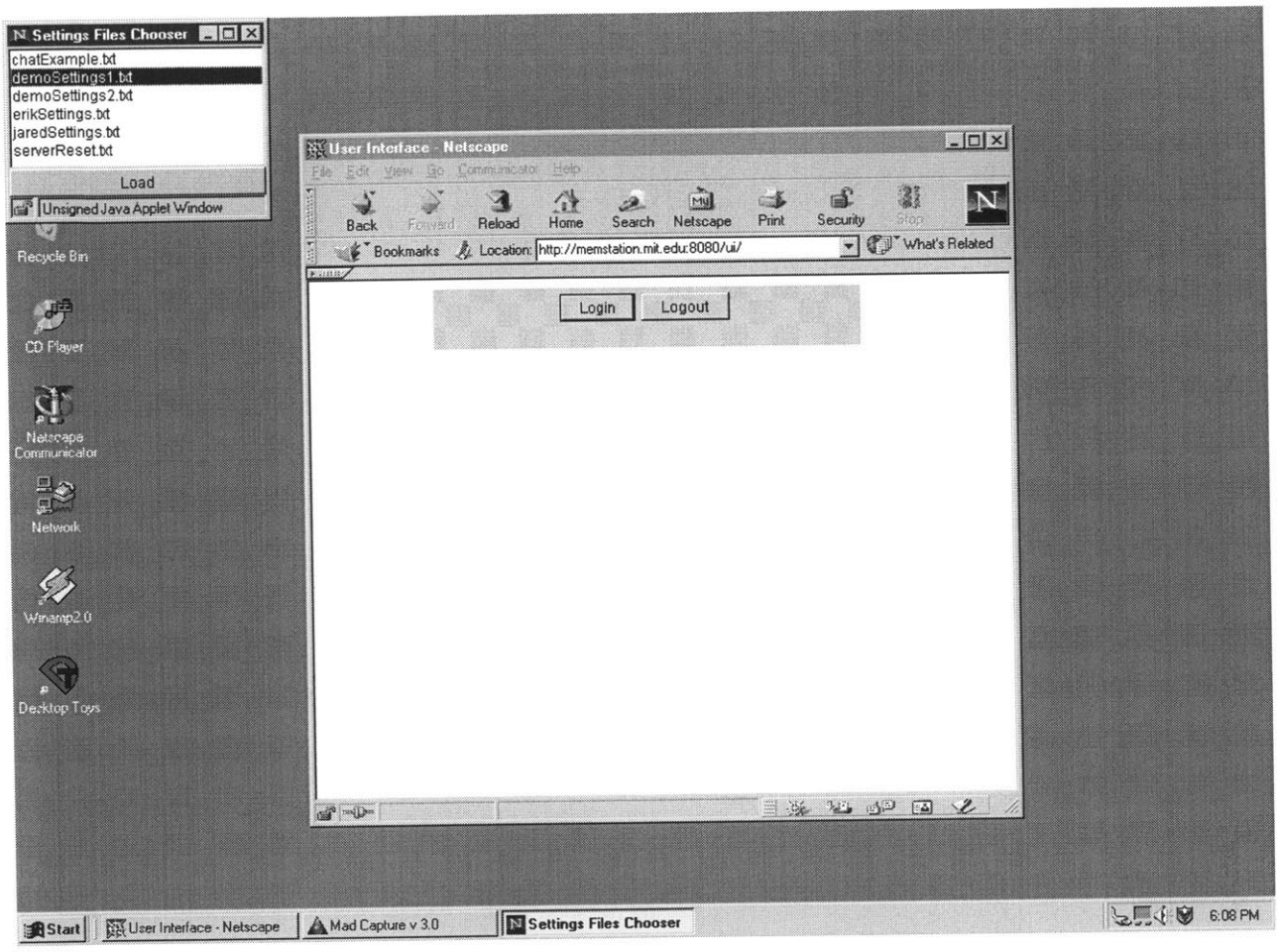

Figure 2-1b: Login Step 2

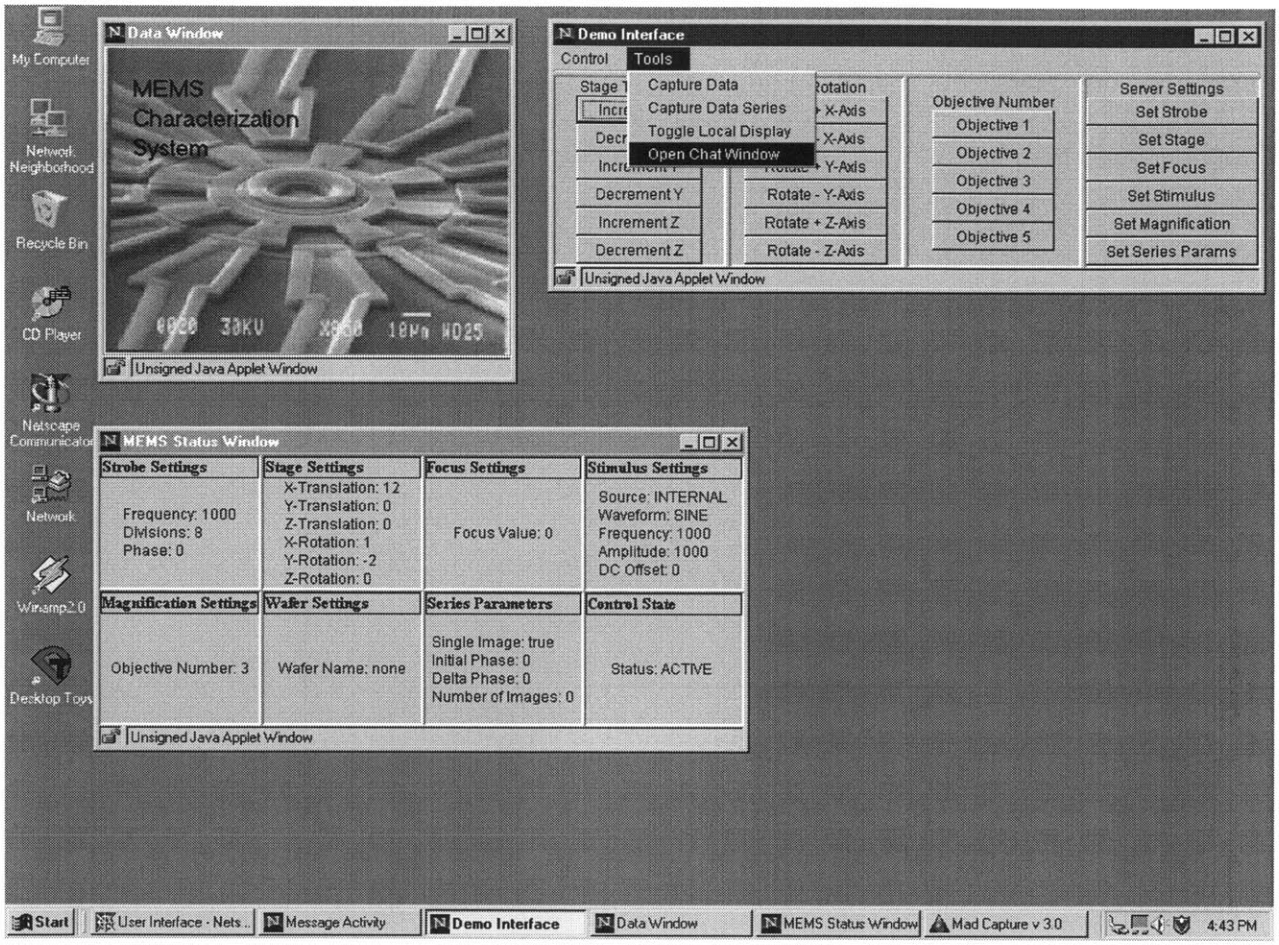

Figure 2-1c: Example Interface

Settings File for user interface in figure 2-1c:

```
mainFrame = Frame("Demo Interface");

mainMenuBar = MenuBar("Main MenuBar", mainFrame);

control = Menu("Control", mainMenuBar);
takeControl = MenuItem("Take Control", control, takeControl());
cedeControl = MenuItem("Cede Control", control, cedeControl());
checkControl = MenuItem("Check Control State", control, checkControl());
reset = MenuItem("Reset Server", control, reset());
shutdown = MenuItem("Shutdown Server", control, shutdown());

tools = Menu("Tools", mainMenuBar);
```

```
captureData = MenuItem("Capture Data", tools, captureSingle());
captureDataSeries = MenuItem("Capture Data Series", tools, captureDataSeries());
activeDisplay = MenutItem("Set Active Window", tools, setActiveDisplayWindow());
toggleDisplay = MenuItem("Toggle Local Display", tools, toggleCaptureDisplay());
chat = MenuItem("Open Chat Window", tools, openChatWindow(10, 3));

mainPanels = PanelFrame("Main Panels", mainFrame, 1, 4);

scopeStageTranslationPanel = Panel("Stage Translation", mainPanels);
incStageX = Button("Increment X", scopeStageTranslationPanel, stageTransIncX());
decStageX = Button("Decrement X", scopeStageTranslationPanel, stageTransDecX());
incStageY = Button("Increment Y", scopeStageTranslationPanel, stageTransIncY());
decStageY = Button("Decrement Y", scopeStageTranslationPanel, stageTransDecY());
incStageZ = Button("Increment Z", scopeStageTranslationPanel, stageTransIncZ());
decStageZ = Button("Decrement Z", scopeStageTranslationPanel, stageTransDecZ());

scopeStageRotationPanel = Panel("Stage Rotation", mainPanels);
incRotX = Button("Rotate + X-Axis", scopeStageRotationPanel, stageRotIncX());
decRotX = Button("Rotate - X-Axis", scopeStageRotationPanel, stageRotDecX());
incRotY = Button("Rotate + Y-Axis", scopeStageRotationPanel, stageRotIncY());
decRotY = Button("Rotate - Y-Axis", scopeStageRotationPanel, stageRotDecY());
incRotZ = Button("Rotate + Z-Axis", scopeStageRotationPanel, stageRotIncZ());
decRotZ = Button("Rotate - Z-Axis", scopeStageRotationPanel, stageRotDecZ());

scopeObjectivePanel = Panel("Objective Number", mainPanels);
objectiveOne = Button("Objective 1", scopeObjectivePanel, setMagnificationValue(1));
objectiveTwo = Button("Objective 2", scopeObjectivePanel, setMagnificationValue(2));
objectiveThree = Button("Objective 3", scopeObjectivePanel, setMagnificationValue(3));
objectiveFour = Button("Objective 4", scopeObjectivePanel, setMagnificationValue(4));
objectiveFive = Button("Objective 5", scopeObjectivePanel, setMagnificationValue(5));

serverSettings = Panel("Server Settings", mainPanels);
setStrobe = Button("Set Strobe", serverSettings, setStrobe());
setStage = Button("Set Stage", serverSettings, setStage());
setFocus = Button("Set Focus", serverSettings, setFocus());
setStimulus = Button("Set Stimulus", serverSettings, setStimulus());
setMagnification = Button("Set Magnification", serverSettings, setMagnification());
setSeriesParams = Button("Set Series Params", serverSettings, setSeriesParameters());

dataFrame = Frame("Data Window");
dataWindow = DataWindow("Data Window", dataFrame, 300, 100);

captureSingle() = Function(setSeriesParamsSingleImage(true));
```

captureSingle() = Function(capture());

captureDataSeries() = Function(setSeriesParamsSingleImage(false));
captureDataSeries() = Function(capture());

2.3 Modules in Client

The client side of the MEMS characterization system is the focus of this work. As a result, there are modules that have been developed for the client side of the system. They will be outlined briefly here and then detailed later in this paper. Figure 2-2 shows how each of the modules in the client software interacts. It will be helpful to the reader to refer to this figure when reading through the descriptions of each of the modules.

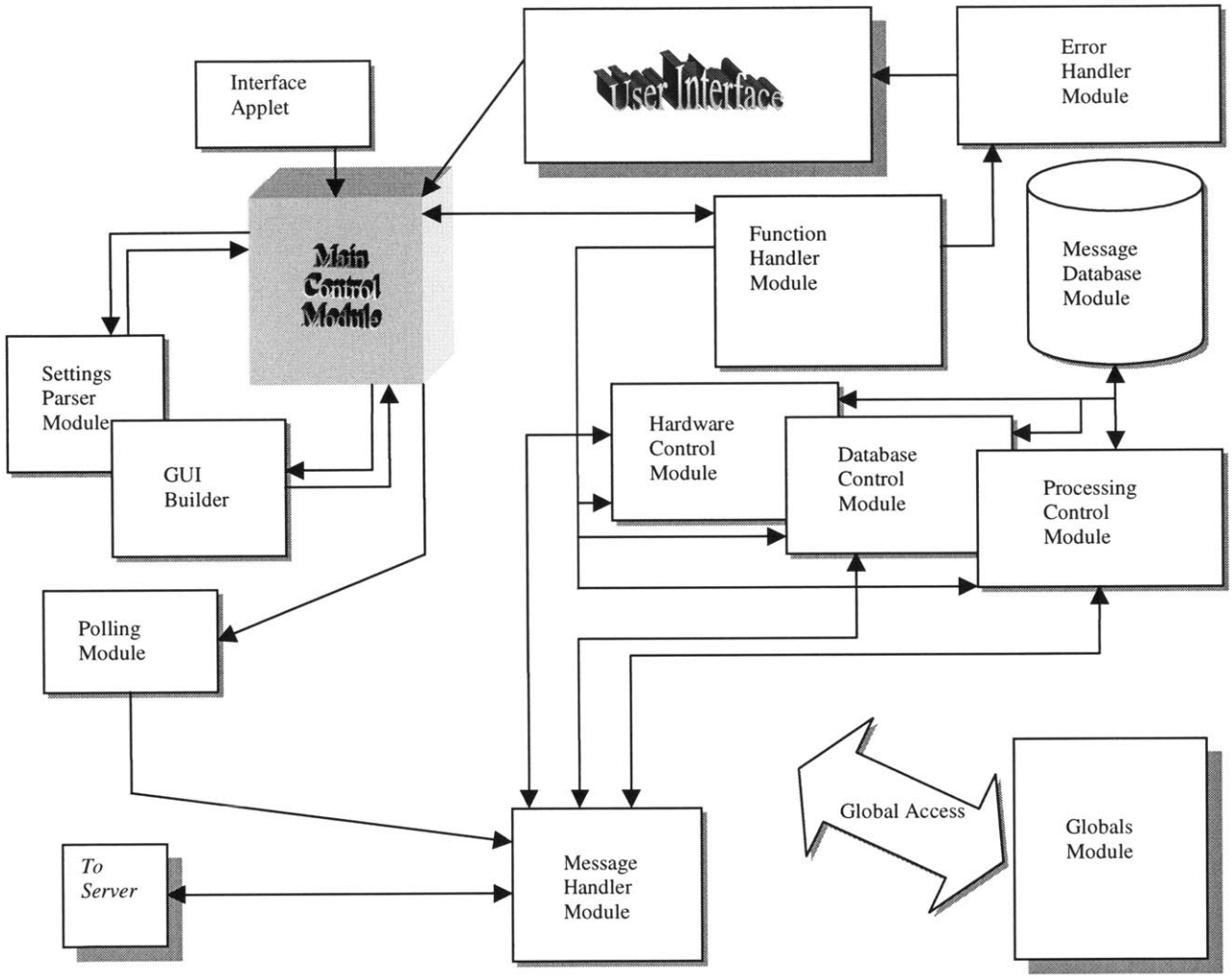

Figure 2-2: Client Modules

2.3.1 Main Control

The main control module is what starts the client modules. Also, any interaction by the user at the interface side must go through the main control. This centralizes all user interaction to this main module. This module also performs all of the initialization involved in starting the client up. It acts as the control center for the client interface for most aspects. Other modules are self contained and are not actually controlled by this module. However, it is this module that will have a part in spawning all other aspects of the interface.

2.3.2 Settings Parser and GUI Builder

These two modules are also used in the startup process for the client side interface. The settings parser performs the parsing of the text file that is used to modify the user interface. The GUI builder is employed following the parsing of the file. It performs the actions required to put together the interface as required from the information in the settings file loaded. Once the interface is built with the builder module, it can be displayed and used.

2.3.3 Polling

Polling is one of the most important aspects of the client. The reason for this is that by polling, the client keeps in touch with the server side of the system. This is how the server sends any information to the clients. The inability of the HTTP protocol to "push" information to a location has forced the use of this polling scheme. Also, the server side uses the polling to determine whether a client is still active. Interclient communication uses the polling module because the communication messages between clients are all sent through the server and the clients must be present according to the server in order for communication to take place. The polling scheme used in the client is based on five and eight second intervals to the server. This timing will be explained in more detail when the polling module is detailed later in this paper.

2.3.4 Function Handler

The function handler module in the client is the basis for performing all of the translation from desired function calls made by the user interface into messages that are sent to the server. According to the function called, the function handler will initiate communication with the server and will handle all responses from the server appropriately. Furthermore, the functionality implemented by the function handler module can be divided into three separate classes of functional modules. These are described briefly next.

2.3.4.1 Hardware Control Module

Hardware control is the largest aspect of the client's functional ability. It must be able to tell the server to change the state of its hardware in order for the client to be able to accomplish work. The hardware control module is where the indirection ends for the functionality in the system. More specifically, this is where the messages are actually sent by the client to the server in order to control the hardware on the server side of the system.

2.3.4.2 Database Control Module

Similar in idea to the hardware control module, this module handles all of the database management functionality as required by the client. Again, the database control module actually sends the messages to the server in order to manage the database. Also, some functions require the use of these database control abilities. In such cases, the function handler may use functionality in both the hardware control module and the database control module.

2.3.4.3 Processing Control Module

This module is what initiates processing data on the server side. At this point it is quite primitive and merely does the initiation of the processing. In the future, this can be expanded in functionality to be able to do more sophisticated processing on the data.

2.3.5 Image Tools

The image tools included in the client side software are not extensive. Basically, these tools consist of the allowing the client interface to display images in the data windows of the interface. There are presently no graphic filters to deal with different graphic formats. Supported graphic types are presently GIFs and JPEGs.

2.3.6 Message Database

The message database is where all of the messages used during a session are stored. The database has been setup so that it can be accessed by all of the modules in the client software. Essentially, it is a constant storage space for all of the message skeletons. These skeletons can be accessed and then filled in appropriately by the modules that use them to communicate with the server.

2.3.7 Globals Module

The globals module holds information about the state of the client which must be kept central. This module can be accessed and used by any of the modules in the system. It allows modules to share information at a global level. This centralization of certain variables is helpful since these variables do not have to be explicitly passed around to the different modules. It is also helpful for the access of variables that never change throughout a session.

2.3.8 Message Handler

The message handler module functionality includes the handling of messages that are both going to and coming from the server. The handler is the front end of the client from the server's point of view. The message handler deals with taking messages in the proper format from the client and sending them to the server utilizing the HTTP protocol. In addition, the handler accepts the response from the server and formats it so that the client can understand the response. Basically, all communication that occurs between the client and the server passes through the message handler.

2.4 Custom GUI Component Classes

The basis for the graphical user interface components used for the client software are included in Java's Abstract Windowing Toolkit (AWT). This will be explained in more depth later. However, the components actually used are part of Java's Swing package. The Swing package is a collection of graphical components that extend the normal AWT components in certain areas of functionality.

Certain advantages and disadvantages exist when utilizing these Swing components. Thus, in order to create all of the components used in the interface, custom graphical user interface components were created. The noteworthy aspect of these component classes is that they are instantiated by the client software. Thus, in order to change how the components are rendered and which components should be used, only the custom component classes must be changed as long as the interface remains intact for the rest of the client access. These custom components were added to enable this flexibility.

2.5 Programmatic Interface

The final aspect of the client to be mentioned here in the overview is the programmatic interface for the system. This means that the client has the ability to run a script file that contains information about how a data set should be collected by the server. The reason for including this type of interface is that data sets often require hundreds of pictures to be taken. To do this automatically through the use of a script file makes the client much more suited to be able to do full scale data collections without requiring a user to point and click on the interface for an extended period of time.

Chapter 3

JAVA GUI DEVELOPMENT

3.1 General Java

In this paper, I do not intend to describe Java and all that it can do. The idea is to cover some of the main ideas about Java that were used for this work extensively. For a more accurate and complete description of Java, Sun's Java technology web page may be referenced. That can be found at http://www.javasoft.com. There are tutorials as well as any new updates and advances in Java located there for easy access. The following information is intended to convey general background information as well as to indicate the reasons for using Java.

3.1.1 Compiler Version

In order to understand how the client software modules work in general, it is important to understand a little bit about the language that was used to create the client. The language chosen was Java. To relate it to a compiler version for reference, Sun's Java Developer's Kit version 1.1x was used. This is important for a few reasons. First, there are older compiler versions that attempt to compile code that has been written using an old and now obsolete event module. Any of the compiler versions prior to version 1.1 use the old event model. In addition, the Swing components used require the use of a Java 1.1x compiler. Finally, as of this work, the latest non-beta release of the developer's kit from Sun is for Java 1.1x. Thus, any enhancements to the compiler included by Sun were utilized because of the use of the latest compiler version.

3.1.2 Object Orientation

Java is an object oriented programming language. An object-oriented language differs from other types of languages in that the paradigm for development is based on creating objects that can be manipulated to accomplish work. Every object used in a program must first be defined in a class. A class is the definition of an object. It is called a class because as a definition of an object, it can include a class of objects. For instance, there may be a class that defines a circle. The class may include a parameter for radius value. In addition, the class may define methods to calculate the area or the circumference of the circle. These methods are essentially functions. With the class written and compiled, it can be instantiated. This means that another class may name a circle and then create an instance of the circle class identified by the name of the circle. In order to create an instance of the circle class, the value of the radius of the circle must be defined according to the present example. Once the circle is instantiated, the methods defined in the class can be called through the instance of the circle. This method of using classes and instantiations of these classes reduces the need to repeat code because classes can be carefully defined to allow them to be instantiated in many different ways. However, most importantly, these classes can be written such that they define these real objects. Thus, all the programming can be thought of as creating objects with which to interact in order to accomplish work. Furthermore, the way that these objects interact with each other is totally up the programmer. Thus, object oriented programming represents a unique way of designing software.

3.1.3 Runtime

Java is a unique language in that it has been designed from the start to be as portable as possible. In light of this goal, Sun Microsystems chose to make it necessary for Java code to be run within a virtual machine. This Java Virtual Machine (JVM), as it is called, is the interface from the Java code to the operating system on a computer. When Java code is compiled, its final form is in what is known as byte codes. These are not machine level instructions but they are also not code level instructions. The level of abstraction of these byte codes lies somewhere in-between these two extremes. These byte codes are given to the JVM for interpretation. The JVM transforms these byte codes into instructions

that can be done within the present operating system. Thus, when code is compiled, it only needs to be run on top of the correct virtual machine in order to work. The JVM must be rewritten for each particular operating system. Fortunately, Sun Microsystems has done this work for us. Therefore, compiled code can be run on any supported operating system. Furthermore, as will be explained next, a JVM can be written to run within a browser. This gives Java another unique feature – its ability to be run through a web browser over a network.

3.1.4 Applets

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). The virtual machine is actually a part of the web browser. Most of the newest browsers have a JVM to run Java code. One note of caution is that there are different versions of web browsers that actually employ different versions of the JVM. Thus, it is important to have the correct version of the web browser in order to run the Java code correctly. As of this work, the focus has been on getting the code to run properly within *Netscape Navigator* and *Microsoft's Internet Explorer.*

The applet class simply defines methods that are often used to create small graphical interfaces within the area of the web browser. It contains methods that can be overridden in a subclass of the applet class. These methods include an initiation method to perform any tasks upon initiation. In addition, methods are included that repaint the applet when necessary. Furthermore, there are methods to perform animation as well as play sounds. The basic idea is that applets are tailored for performing small-scale multimedia programs. For this work, only the initiation and the repaint methods have been overridden.

3.2 Java's GUI Capabilities

An interface is simply a way to interact with an entity. Ultimately, the idea is to cause the entity to react to the interaction through the interface. The same ideas are true for a graphical user interface (GUI). The term *graphical* implies that that the interface involves some sort of visual indication of how

to interact with an entity. The term *user* refers to the fact that there will be a user interacting with the entity through the interface. Finally, the term *interface* reiterates the fact that the idea is to create a connection between the user and the entity through a means of communication common to both user and entity, namely the graphical interface.

3.2.1 Java's Abstract Windowing Toolkit

Sun's JDK comes with core Java classes that define graphical components for use in development of interfaces. The idea of using graphical interfaces has been attractive for a long time because of the ease with which they can be used. They can be designed so the user can learn to use the interface using information given within the interface. This is true because the components in the interface have functionality that can be indicated on the components. The core Java classes that allow this design to occur are called the Abstract Windowing Toolkit (AWT). The components that these classes define and allow the programmer to use include windows, frames, buttons and menus with menu items.

3.2.2 Swing

Swing is Sun's name for the extension to Java's Abstract Windowing Toolkit. The extension provides for greater control over the graphical components that can be used in creating graphical user interfaces and it also includes new components not included with the standard components that come with Java. One of the main features utilized for this work is the ability for Swing components to look different according to the operating system on which they are running. This is called "look and feel" customization. There are looks and feels for Windows 9X/NT, metal (a unique Java look and feel), and motif. There is also support for a Mac look and feel, however, this has not been tested. The result is that the components look more like they were created by the operating system and it makes integration with different operating systems less distracting in terms of appearance. In addition, Sun claims that the new components have been optimized for better performance. For more information on the Swing package, see Sun's *Swing Connection* [15].

Chapter 4

CUSTOM USER INTERFACE MODULES

4.1 Customizable Interface

The graphical user interface developed in this work has been done so with the idea of customization in mind. More specifically, the interface has been designed so that it can be modified for use during each session. The reason that this idea is useful for the MEMS characterization system is that usage can be divided into two categories. The system may to used to collect data or to analyze data. Thus, in order to make the interface less cumbersome in both circumstances, it can be modified such that only those functions that will be used in a given session can be included in the interface. For instance, in order to collect data on a device, the stage of the microscope must be positioned properly and the excitation must be setup as well. However, when analyzing, none of these controls are necessary and in fact are quite useless. Thus, the interface can be modified according to the application. This idea is not only useful in this project, however. It can be useful in other situations where an interface is required for different applications at different times.

4.1.1 Interface Component Organization

In order to understand how the client side user interface can be modified, it is important to understand what components can be used to customize and create the interface. It is also important to understand how they are organized within the interface. The user interface from a graphical standpoint is made up of frames, menus, menu items, data windows and buttons. All components must be placed inside of a frame. More specifically, menus and menu items are added to the top of a frame. Data windows are assigned space in a frame. Finally, buttons are assigned space in the frame as well. Buttons are organized into groups of buttons and placed within panels within a main frame. Furthermore, if there is a data window and a group of buttons within the same frame, the data window

will be placed above the groups of buttons. There is no real reason for this besides pure aesthetics. The assigning of space within the interface is not done by the user. It is done by the interface setup components of the client interface software. In addition, any of the components are optional in a frame. Because of this freedom in design, it is up to the user to create an interface that looks and acts as desired.

4.1.2 Functional Based Interface

The graphical side of the user interface is based on assigning primitive functions to each menu item and button. These primitive functions have been chosen carefully such that they allow the user to access the functionality of the system. In addition to the ability to access these primitive functions in the client, there is the ability to build more complex functionality using combinations of the primitive functions. This allows the user to build entirely new functions from the existing primitive ones available. One of the most useful applications of this feature is the ability to repeat primitive functions multiple times. The reason that this is useful is that the user can automate certain functions making the interaction with the system less tedious. Of course, one disadvantage of allowing the user to have this sort of control over the functionality of the interface is that the user has the ability to chain together certain functions that cannot be assigned together within a user defined function. However, the flexibility that this functional based interface contains outweighs this particular disadvantage.

The primitive functions will be detailed later in this paper when the function handler module is described in detail. In general, however, the primitive functions are based on the messages used in the message passing protocol for communication between the client and the server. As briefly explained previously, these messages request certain actions to be taken at the server side. More specifically, the messages correspond to requesting the server to change the stage position, set up the image capture parameters, define the excitation signal to the device under test, and modify and manage the database. Some of the messages are not in one-to-one correspondence with the primitive messages that can be assigned to each of the elements in the interface. In these cases, the primitive functions actually handle a multiple message exchange session between client and server.

Thus, the idea behind the functional based interface is that buttons and menu items can be assigned to correspond to primitive functions. In addition, user defined functions can be created that are built up from a series of primitive functions. All of this function assignment occurs as a result of a file that can be defined by the user. This will be explained next.

4.2 Settings File

The user interface for the MEMS characterization system requires a setup file to be read by the client from the server in order to define the graphical interface. Thus, it is important to understand the format of the settings files and how the user can modify them. Also, it is important to understand how the settings files are used by the client so that they can be utilized properly. As a reference, the reader can refer back to the example of a user interface and the respective settings file that was presented earlier in this writing when a client overview was presented.

4.2.1 File Format from OS Perspective

The file format from the perspective of the operating system is text. At the most fundamental level, the file is in basic ASCII format with some additional formatting characters such as newline characters. The reason that this file format was chosen is that it is universally editable. Any operating system has the ability to read and edit files in text format. This allows the user to edit a settings file easily and without any special tools. Also, since the settings files are not generally very large in number of characters, these text files are not generally large. This has the side effect of making file transfers easy.

4.2.2 File Format from Grammar Perspective

There is another aspect of the format of the settings file that is more important on some levels than how the operating system sees the file. This aspect is the file format according to how the user sees it. Fundamentally, the file is designed to allow the user to customize the user interface for different uses of the MEMS characterization system. Of course, this customization occurs within certain limits. In order for the user to be able to utilize the customization capabilities of the system, there must be an

understanding from a grammatical perspective, of how the file needs to be formatted. In general, a grammar defines how objects can be combined by laying down rules to follow. In this case, it defines how the user can define utilize text to setup the user interface.

In general, all of the characters used in the settings file must be ASCII defined characters. It is important to keep in mind that the parser of this file, which will be explained shortly, looks for certain characters while parsing. Thus, in order to avoid a situation where the parser cannot do its job, these formatting characters must be included in the proper places. Each defined item must be terminated with a semicolon. Furthermore, for readability, each defined item should be on a new line. It is important that there are no spaces following items because the parser will attempt to use the spaces and errors will occur before the interface even gets loaded.

There are three visible components that can be created and manipulated within the settings file. They are menu items, data windows, and buttons. However, in order to create and define how these components are used, there are other constructs that must be created within which the three main components must reside. In particular, there is a hierarchy of constructs which define the interface. Everything must reside within a frame. There may be multiple frames created, but all components must be placed within a frame. When the hierarchy tree is traced back to its origination, the top component is always a frame. Furthermore, all menu items must be put within a menu and each menu must be placed within a menu bar. A menu bar is merely the area at the top of the frame that holds the actual menus. The trace from frame to menu item ends after going through these four levels: frame, menu bar, menu, and menu item. Data windows are different because they are placed directly into a frame with no other levels in between. Buttons, like menu items, have two levels in between the frame and the button. First, a panel frame must be placed within a frame. Then, a panel is placed within a panel frame. Finally, a button (or a group of buttons) is placed in a panel. Thus, there are four levels of components involved in properly defining a button on the interface: frame, panel frame, panel, and button. There is one final component that actually is not placed within anything. This component is a user defined function. Exactly how this component is used will be explained along with each of the other components mentioned. Finally, each definition ends with a semicolon. This is

used by the settings file parser to figure out where each component definition ends and a new one begins. Figure 4-1 below illustrates how each component relates to other components.

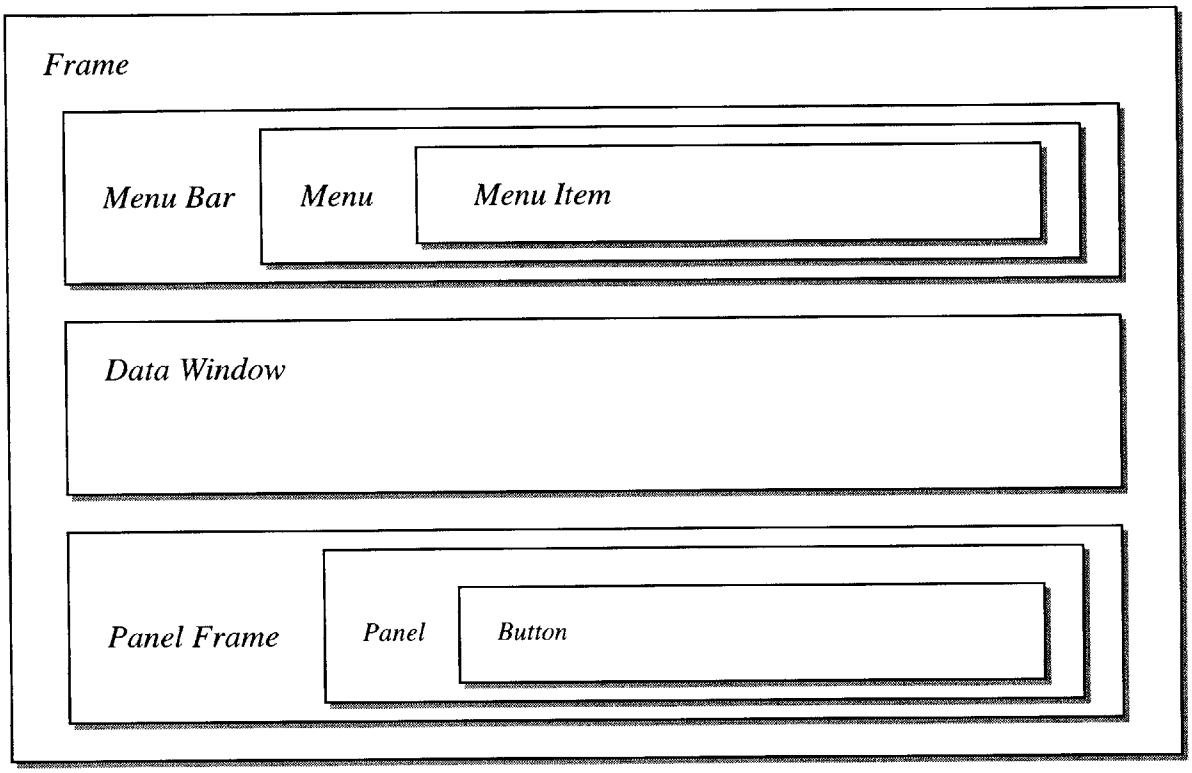

Figure 4-1: Interface Component Hierarchy

4.2.2.1 Frames

A frame definition adheres to the following general format. In each of the definitions that follow, text in italics indicates that the user must choose what these words are. Text that is not in italics indicates key words that must remain where they are and they must be typed exactly as they appear in the definitions.

frameName = Frame("*Frame Title*");

The above rule for defining a frame begins with a frame name. This name must be one word consisting of any ASCII character (except white space) and it is used to indicate where to add components for the interface. This name must be unique among any other frame names that may exist in the settings file. This is a result of the fact that the name is used as an identifier for other components to use in the interface building process. Following the name is a space, then an equal sign and then another space. Although it may seem unnecessary to require such strict formatting, the parser is not robust enough in its present form to deal with deviation from this well defined structure. The next part of the above rule is the key word "Frame." The parser uses this word to decide what to do with the rest of the line of text that is fed to it. Following the keyword are parenthesis which contain a title for the frame within quotes. This title is displayed on the frame when the interface is displayed. So although any string can be used (including multiple words without white space), it makes sense to title the frame with a descriptive title.

4.2.2.2 Menu Bars

Now that frames have been defined, the rest of the components can be defined since they are all added to frames. Menu bars are created using the following statement structure.

menubarName = MenuBar("*Menu Bar Title*", *frameName*);

The above rule has a format very similar to the rule for defining a frame. First, there is a menu bar name which must be unique so that it can be identified by the interface builder. This is followed by a space, an equal sign, and another space. Then, the keyword "MenuBar" must be added. After the keyword, there are parenthesis that contain information about the menu bar. The two pieces of information are separated by a comma. The first part of the information is the name of the menu bar. Actually, this piece of information is not really needed since it is not displayed anywhere on the interface. The reason that this was included was to keep the structure of each rule very similar. The second piece of information that must be included is the name of the frame to which the menu bar must be added. The frame that the menu bar is to be added to must have been created at some point in the settings file. Otherwise, the interface builder will have no place to put the menu bar.

4.2.2.3 Menus

Menus are added to menu bars and they follow a structure exactly like menu bars except that they have a different keyword. The rule for the construction of a menu is as follows.

menuName = Menu("*Menu Label*", *menubarName*);

As can be observed the format is exactly the same as in the definition of a menu bar. The main difference is the keyword. For a menu, the keyword "Menu" must be used to indicate that the component is to be a menu. The menu label, shown in quotes in the above definition should be chosen carefully because it will displayed on the interface. Furthermore, the menu bar name field within the parenthesis indicates the menu bar to which the menu is to be added. As a result, the menu bar name must point to a valid, already defined menu bar. If it does not, the interface builder will not know where to add the menu. The order of the menus in the settings file represents the order which the menus appear in the interface. In particular, the first menu in a list of menus added to a particular menu bar appears on the left side of the menu bar. The rest of the menus are added to right of the previous menu.

4.2.2.4 Menu Items

Menu items are leaf components. What this means is that if a tree was constructed to represent the structure of an interface, the menu items would reside at the endpoints or leaves of the tree. This means that no components can be added to menu items. Also, menu items are unique among the components presented thus far because they can be assigned to functions for use in the interface. The definition for the construction of a menu item is as follows.

menuItemName = MenuItem("*Menu Item Label*", *menuName*,*functionName*);

The structure of a menu item is the same as both the menu bar and the menu. However, there are some key differences to note. First, the keyword "MenuItem" is used to indicate that the component

to be created is a menu item. Second, there is another field of information contained within the parenthesis. This field is used to connect the menu item with a primitive function. The primitive functions will be explained in more detail in the function handler chapter of this writing.

The first item in the menu item definition is the identifier for the menu item. Since this identifier is not actually ever used again, it is not really essential. However, as stated before, the structure of each object has been designed to be similar enough so that new structures will not need to be learned for each component. The identifier must be unique, as with all components. The parser and the interface builder actually use the identifier so it is still important to keep it unique. The keyword used for a menu item creation is "MenuItem." Then, contained within the parenthesis is the information needed to construct the menu item. First, the menu item label is what the user will see when a menu is opened in the interface. Thus, this title should be indicative of what the item is going to do within the interface. This title is contained in quotes just to remind the user that this will be a displayed string. The next field within the parenthesis is the menu name field. This indicates where the interface builder is supposed to put the item. It is required that the menu is defined at some point in the settings file so that the parser can correctly place the item. Finally, the last piece of information in the definition is the function name. This connects the menu item to one of the primitive functions given to the user or even a user defined function. How this is done will be explained in the function handler chapter. The requirement put upon the function name in the definition is that it is either a primitive function or it is a user defined function. As with the menus created in the interface, the menu items appear in the interface in the order they are written in the settings file. More specifically, the first menu item added to a particular menu is added to the top of the menu when it opens up in the interface. The rest of the menu items are added below the previously defined item.

4.2.2.5 Data Windows

The structure of a data window definition is unique among all of the components because it actually contains setup arguments within the definition. Following is the definition for a data window.

dataWindowName = DataWindow("*Data Window Title*", *frameName*, *width*, *height*);

As with all the definitions, the construct starts off with an identifier for the component. It must be unique, at least among data window names. This is because the interface must be able to identify an active data window among all of the data windows in the interface. Thus, in order to be able to do that, each data window must be unique in its identification. The next element in the definition for a data window is the common sequence of a space, an equal sign, then another space. The keyword "DataWindow" appears next in the definition. This is what signifies that a data window is to be created. Within the parenthesis, there are four different pieces of information. First, within quotes, there is a title string for the data window. Even though this string is not displayed in the interface, it has been included to keep the format consistent with the rest of the components. Following the title for the data window, there is a frame name field. This value identifies the frame within which the data window is to be placed. The frame must be a valid frame as defined elsewhere in the settings file. The last two fields within the parenthesis define the height and width of the data window. The values are entered as numbers and they represent the pixel width and height of the data window. These values are initial values, meaning that if an image gets placed within the data window that is larger than that size, the window will expand to contain the image.

4.2.2.6 Panel Frames

In general, a panel frame is a frame that is created to hold panels. The panels within the panel frame then hold buttons. This means that the panel frame component is at the same level within the interface as the menu bar described previously. As a result, the format is similar. However, there are some important differences. Following is the syntax for the definition of a panel frame.

panelFrameName = PanelFrame("*Panel Frame Title*", *frameName*, *rows*, *columns*);

As with all components, there is an identifier for the panel frame. All panel frames within a frame must have a unique identifier in the settings file. It is also good to keep this identifier unique among all components for readability. After the equal sign, the keyword for the component is needed. In this case, it is "PanelFrame" to indicate that a panel frame is to be created. Within the parenthesis, there are parameters used for the creation of the panel frame. First, contained in quotes, there is the title for

the panel frame. Even though this is never actually displayed on the interface, it was included to keep the formats consistent among different components. Following the title, there is a field that identifies where to put the panel frame. The frame must be defined in the settings file in order for the panel frame to be put into the frame. The last two fields within the parenthesis represent the layout of the panel frame in general. Entering this information requires some planning on the user's part. This is because the values represent how many rows and columns to setup in a grid to hold the panels that are added to the panel frame.

4.2.2.7 Panels

Panels are at the same level of construction within the interface as menu components. The format for panels are therefore very much like the format for menus. They have exactly the same structure except for the keywords used. Following is the definition for the creation of a panel component.

paneName = Panel("*Panel Title*", *panelFrameName*);

The panel identifier constitutes the first part of the construction statement. This identifier must be unique among the other panels defined in the settings file and it should be unique among all other components defined. After the equal sign in the definition statement, the keyword "Panel" must be typed. This tells the interface builder that the component is to be a panel. Following the keyword are the parameters to be fed to the interface builder. The first parameter is the title of the panel. The title is contained in quotes to indicate that it will be displayed on the interface. The title will be displayed on the interface so it should be chosen to reflect accurately the functions that it contains. Of course, this is not a requirement, but it makes the use of the interface more intuitive. The second parameter contained within the parenthesis is a panel frame name. This indicates the panel frame to which the panel is to be added. In general, the panel frame must be defined in the settings file so that the interface builder knows where to put the panel. The order of the panels added to the panel frame corresponds to the order in which they are defined in the settings file. In general, panels are added from left to right and from top to bottom according to the grid defined in the panel frame where the panels are added.

4.2.2.8 Buttons

Buttons are like menu items with regards to the way that functions can be assigned to them. Buttons are used to perform work for the user. The definition of a button follows the proceeding structure.

buttonName = Button("*Button Title*", *panelName*, *functionName*);

The first part of the definition is the button identifier. The identifier must be unique among buttons in the settings file so that the interface builder will be able to store the buttons properly. Following the equal sign is the keyword "Button" which indicates to the interface builder that a button is to be created. After the keyword, there must be parenthesis that contain the parameters necessary to build the button. The first parameter is the button title. In this case, the text is displayed on the interface. Therefore, it is important to choose the title carefully so that it accurately reflects what the button is supposed to do in the interface. After the title, a panel name must be included in the parameters. The panel name indicates the panel to which the button is to be added. As with the other components added to the interface, buttons are added to the panel in the order which they are placed in the settings file. More specifically, buttons are added to their respective panels from the top to the bottom. The last parameter in the button definition is a function name. This attaches the button to a function. The function can either be a primitive function or a user defined function. Any primitive function can be attached to a button. Also, primitive functions can be used to create a user defined function. This process will be explained in more detail later in this paper.

4.2.2.9 Functions

Functions are the final component that can be created from the settings file. They are only used when a user defined function is to be created. The definition for a function is as follows.

functionName = Function(*functionName*);

The format is simple but it is a powerful definition. The function name identifier at the beginning of the definition is the name of the user defined function that is to be created. The general form for the function identifiers follows the form for the names of the primitive functions. What this means is that they look like *function()* where the function is identified and then parenthesis are placed at the end of the function name. It is this identifier that is used when user defined functions are attached to either menu items or buttons. The identifier has to be unique so that it can be found within the function handler when it is called. Following the identifier is that standard space, equal sign, and another space. After the equal sign, the keyword "Function" must be used to tell the interface builder that a function is to be created. Within the parenthesis following the keyword is the name of a function that is to be added to the user defined function. The powerful part of the definition is that the function name within the parenthesis can either be a primitive function or a user defined function. This allows these functions to call each other which is how the user interacts with the server from a programmatic perspective. This will be explained in more detail in the programmatic interface chapter of this paper.

4.2.3 Illustrated Example

To help the reader understand what each of the above components looks like in the user interface, figure 4-2 below shows an example interface with at least one of each of the above components. Each component has been pointed out so that it is obvious how they appear in the interface after it is built by the client software. The only definition that cannot be pointed out in this illustrated example is the function definition. The functions are attached to each of the buttons and menu items in the interface. The rest of the components can each be clearly seen in the figure.

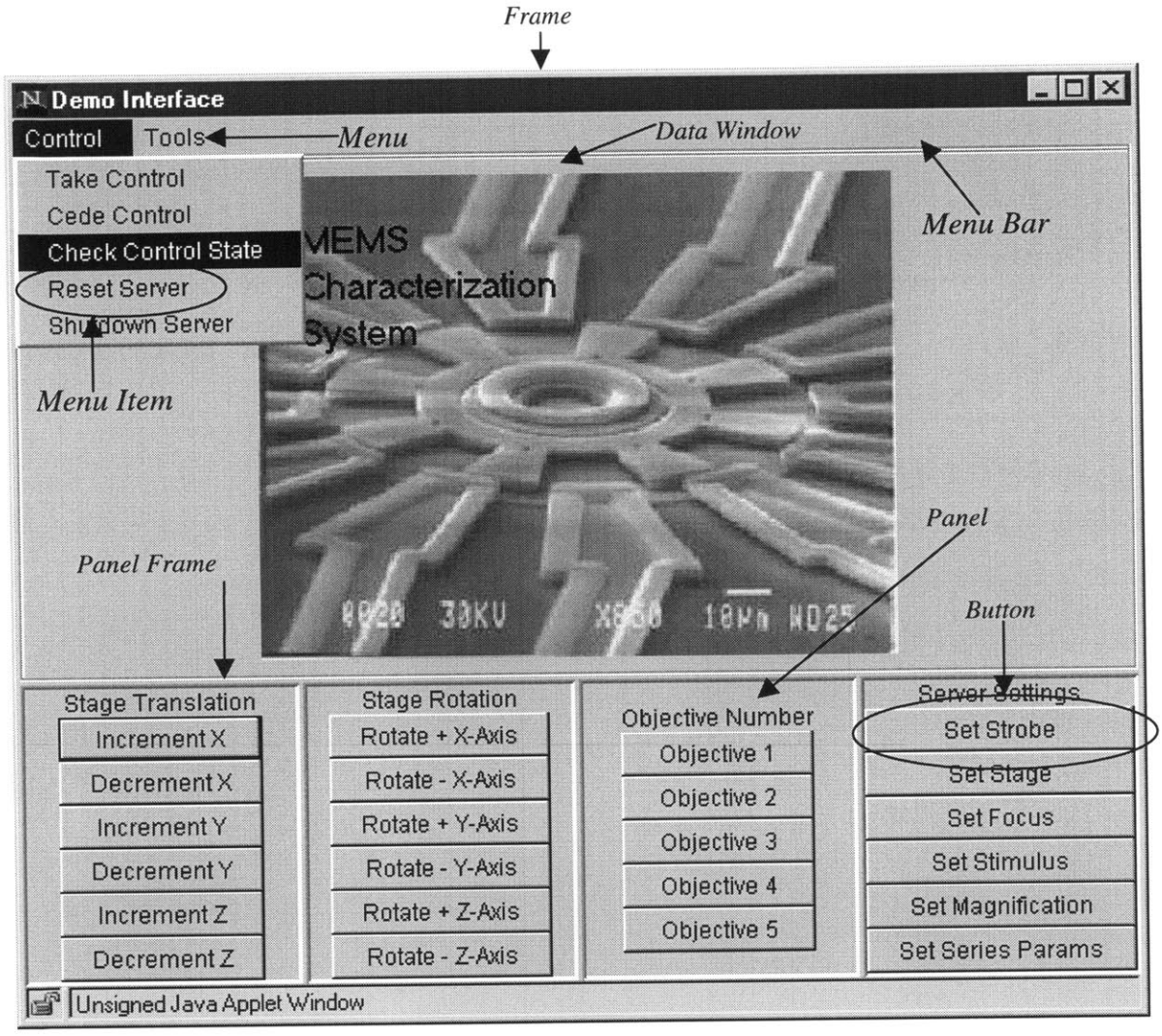

Figure 4-2: Illustrated Example of Components

4.3 Settings File Parser

One of the main modules in the client is the settings file parser. The format of the settings file was detailed in the previous sections of this chapter. The function of the settings file is to allow a user to define a custom interface. The settings file parser takes the settings file as input and outputs an organized data construct that is used by the GUI builder to create the user interface. That is why it is

called a parser since it takes in properly formatted text, interprets the format and returns the information in another format.

The parser works to separate out the different components that can be defined in the settings file. These components were explained in detail in the preceding sections. Therefore, they will not be repeated here. The parser works by going through the settings file from the top to the bottom. That is one of the reasons why it is important to format the settings file with order in mind. Initially, each component definition is separated through the use of the semicolons that end each definition. The parser makes use of Java's string tokenizer utility which allows the parser to split text into tokens by looking for key symbols in the text. After each line is found from the settings file, the parser utilizes the fact that each component is defined by first having an identifier followed by an equal sign. By assuming that this is the case for each definition, the parser stores the identifier and then finds the keyword after the equal sign in the definition. After figuring out the keyword being used for the present identifier, the parser knows how many parameters are supposed to be in the definition and also what type of parameter is supposed to be in each position. This allows the parser to store the information in a data structure for the GUI builder to use.

The data structure that is created utilizes another Java utility that allows the use of a hashtable. This makes it possible to store any type of data into a structure with an association connected with the stored object. The association is a string identifier that can be used to load the object from the hashtable for use in a class. The most useful aspect of the hashtable is that it can store different types of object within the same hashtable. For instance, there are both vectors and hashtables stored in the data structure created by the settings file parser. With this data structure created, the GUI builder is invoked to make use of that data.

Each type of component is stored in hashtables in the order they are encountered in the settings file. Furthermore, the dependency between each component is stored in other hashtables. Thus, the GUI builder is passed enough information to be able to identify each component within the data structure and it also knows how each component fits together. This is all a result of the parser placing the data correctly into the data structure passed to the GUI builder.

The settings file parser produces a data structure that contains all of the components defined in the settings file. More specifically, the structure contains frames, menu bars, menus, menu items, data windows, panel frames, panels, and buttons. Furthermore, the structure also contains any user defined functions. However, there is more information stored than simply these components. This is because of the fact that each component has to be added to a higher level component, except for user defined functions. As a result, there are dependencies that exist between the defined components and these dependencies are stored in the data structure along the components. Even more subtle, there is one element of the data structure that contains all of the component and function identifiers. This is for bookkeeping so that the interface knows what the user has defined.

4.4 GUI Builder

As the name implies, the GUI builder module in the client builds the graphical user interface from a data structure that is passed from the settings file parser. The data structure passed to the constructor for the GUI builder module contains all of the information necessary to build the interface for the user.

4.4.1 Using the Data Structure

As explained in the previous section, the settings file parser creates the data structure that contains all of the components and relevant information to be used by the GUI builder module. As a result, as soon as the GUI builder receives the data structure, it first finds the appropriate information. It does this by using a construct called a hashtable. As was explained, the data structure created by the settings file parser is a hashtable and it stores objects using associations. In this case, the associations are strings that describe the object to which they are pointing. The term for the string is the key and the associated object is the value. Thus, in order for the GUI builder module to get the appropriate objects from the data structure, it must utilize the keys appropriately. The keys that the GUI builder uses to access each value are consistent with the keys used by the settings file parser to store the values. They consist of descriptive names for each type of value within the data structure. Also, as each value is pulled out of the data structure, it is saved in a variable within the GUI builder module. This is done

so that using each value would be easier. Another noteworthy point to understand is that each value within the data structure may also be a hashtable. In fact, there are both hashtables and vectors contained within the main data structure. After each value is pulled out of the data structure, the GUI builder the continues to build the interface with the information that it has.

4.4.2 Building the Interface

After each useful value is stored locally, the GUI builder begins to create the interface as defined in the settings file. The first thing that the GUI builder does is that it creates each of the frames that are defined in the settings file. The reason that each frame is created before anything else is that every graphical component is fundamentally a part of a frame. This is because the frame is at the first level of the interface meaning that each component is, through a series of dependencies, added to a frame. The builder knows how many frames to create because it has a vector that contains the identifications for each frame defined in the settings file. Each of these frames are added to a global hashtable. The construct of the global variables module is detailed in another section of this writing. Essentially, once added to the global frames hashtable, they are static frames and exist over the lifetime of the user interface.

4.4.2.1 Adding Menu Items

Once each frame is created, the next part of the interface that is built is the menus. This includes creating a menu bar for each frame where one is defined and then adding the respective menus and menu items to the menu bars. The GUI builder begins the process by looking at a vector that contains all of the identifiers for menu items to be added to the interface. By going through each identifier, the builder module knows which menu a menu item belongs to as well as which menu bar each menu is supposed to be placed. Thus, for each menu item, the builder traces the dependencies all the way back to the frame and then adds a menu bar to the frame with a new menu item added with each iteration of the process. This is not the most efficient method of accomplishing the task because of the fact that the menu bar for each frame is copied, modified with a new menu item or even a new

menu, and then the original menu bar is overwritten with the new menu bar. However, this process makes the building mechanism very general.

Information that is used to create each of the menu items and menus includes the name given to each menu item and menu as defined in the settings file. The GUI builder carefully adds these text labels to each menu item and menu. Furthermore, the dependency of each component upon then next higher level component is also known by the GUI builder. This information is all contained in hashtables pulled out of the main data structure from the settings file parser.

4.4.2.2 Adding Data Windows

A method similar to the way the menu items are added to interface is used to add data windows to the interface frames. One of the objects that is extracted from the data structure passed to the GUI builder is a vector that contains the identifiers for all of the data windows defined in the settings file. In order for the GUI builder to add each of the data windows to the proper frames, the vector containing the identifiers for the data windows is used. Each element of the vector is examined, and the data window information is extracted from a hashtable using the data window identifier as the key. Using the information from the hashtable object, the GUI builder can identify to which frame the data window is to be added. When the data window is added to the frame, the modified frame is put back into the hashtable that contains all of the frames for user interface. This process replaces the previously saved frame with the frame that was modified by adding the data window. One thing to note is that no data windows need to be added to a frame. The GUI builder checks to see if there are any data windows defined in the settings file before it tries to add them to the frames.

A noteworthy concept for the user is that there can only be one data window per frame. The reason for this has to do with the type of layout tool used for the frame. The layout manager that is used for the frames is called a border layout manager. This is a standard layout tool included in Java's Abstract Windowing Toolkit and it allows components to be added in all four directions as defined on a compass (i.e. north, south, east and west). Furthermore, there is a center space on the frame. The

data windows are added to the center of the frame. Thus, if more than one data window is added to the center, only the last data window added will actually be visible in the frame.

4.4.2.3 Adding Buttons

In a method similar to adding menu items and data windows to the interface, buttons are added to the interface. However, there are some key differences in the way that the buttons are added to the interface. One of the object values in the main data structure produced by the settings file parser is a vector that contains all of the identifiers for the panels to be included in the user interface. Thus, the GUI builder has a record of the panels that it must add to the interface as it is built. The panels are created before any of the other necessary button related components such as the panel frames and the buttons. Following the creation of the panels, they are placed in a hashtable using the panel identifiers as keys for the panels objects. Then, the GUI builder uses another identifier vector containing all the identifiers for the buttons defined in the settings file in order to add the buttons to panels. At the same time, the panels created beforehand are added to the proper panel frames and the panel frames are added to the proper frames.

The method for adding buttons to the interface has the same inefficiency as the method for adding menu items to the interface. More specifically, the panel frames are extracted from a hashtable, modified with the appropriate panels and then replaced by the modified panel frames. At the same time, the panels get extracted from another hashtable, are modified and then replaced by new panels. Thus, there is the extraction, modification and then replacement process that occurs for each button added to the interface. This inefficiency is not in a critical path of the interface since all of this is done at startup and not while the interface is actually doing work. Thus, although the efficiency can be improved upon, it is not critical. Also, this method allows the adding process to be very general.

4.5 Result of Parsing and Building

After the settings file is parsed and the interface is built, the output of the process is in the form of a hashtable that contains all of the frames which were built by the GUI builder. There is a method

within the GUI builder that allows another module to get the frames hashtable from within the GUI builder module. The method is used by the main control module, which will be detailed in the next section. The advantage of only returning the built frames in a hashtable to the main control module is that it can decide when to display the frames to the user. The result of the settings file parser and the GUI builder is therefore used by the main control as it starts up the rest of the client and displays the user interface. After these modules are done with their functions upon startup, they are not invoked again during a MEMS characterization session.

Chapter 5

MAIN CONTROL MODULE

5.1 Overview

The main control module is responsible for all control within the user interface. It starts all of the necessary modules for the client and it is the center of all control while the user interface is running. There are many functions that the main control module performs and those will be described in detail later in this paper. The main idea behind the main control module is that it is the center for all action that occurs in the client software.

5.2 Client Login Applet

The reason that the client login applet must be discussed before the main control module is discussed is that that the client applet is what actually starts the main control module. The client applet is the first point of contact that the user has with the user interface developed in this work. As discussed before, applets are programs that are downloaded from a server and run locally through a Java Virtual Machine. Using this definition of an applet, everything about the user interface for the system could actually be considered an applet. However, there is an applet class that is defined in the Java language that has methods dealing with internet media. However, only the initialization method is used in the interface. As can be observed in figure 5-1, the applet simply consists of a box that contains two buttons. The buttons are labeled "Login" and "Logout." The applet appears in the center of the browser window and it is the first point of contact between the user interface and the user. When the login button is depressed, the interface begins and a window is created that lists the settings files available on the server. It is at the point before the settings files window is created that the applet is not used anymore. This is because the applet starts up the main control module and everything after the login is handled by the main control module.

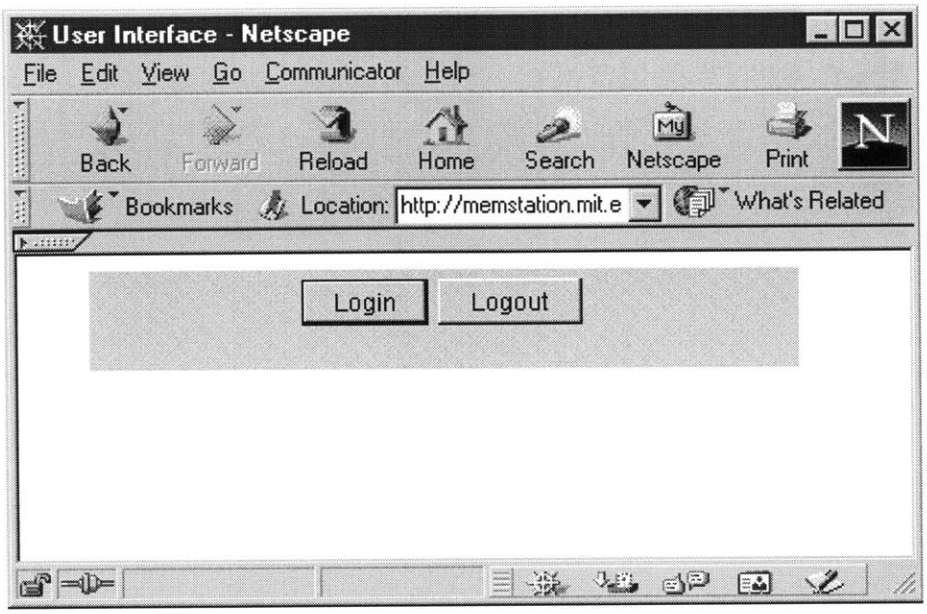

Figure 5-1: Client Login Applet

The applet is obviously not used to accomplish too many tasks. However, there are some subtle but vital tasks that it accomplishes. First of all, it sets the look and feel of the graphical components drawn by Java. As described previously in this paper, Swing has different looks and feels that can be used to display the graphical components. What this means is the Swing attempts to make the components look like certain popular operating systems. For instance, there is a windows look and feel that makes all the components look like they belong in a Windows 9X/NT environment. There is also a metal look and feel which is a Java original as well as a motif look and feel to accommodate X-windows users. The goal of the ability to change the look and feel of the graphical components is to create more of a sense of continuity between Java programs and the operating system in use on a given machine. The other task that startup applet accomplishes is to create the URL necessary for access to the server. The message server URL, which is the message interface access point to the server, is created. This URL defines where the message servlet entry point resides on the server. All messages are posted to this URL from the client.

With those tasks done, the applet passes all control over to the main control module and then waits for a logout from the user. When the applet registers that the logout button has been depressed, it

sends a logout message to the server and server ends the client session. Thus, in order to log back on, the login button must be depressed again.

5.3 Tasks of the Main Control Module

The main control module is the center for all control of the user interface. It starts fundamental modules and it is the one place through which all user interaction must pass to some degree. Thus, it remains as the most essential module among all client modules.

5.3.1 Polling

The main control module starts two polling timer listener modules within the client. Polling is an important function for the client. In fact, from the server's perspective, a requirement is that the client must poll every five seconds or else the client will be killed. Another effect of this polling is that the client can check to see if there are any messages in the message queue that should be delivered to the client doing the polling. As mentioned, there are two polling timer listener modules started by the main control module. The first listener is synched to a five second timer that is continuously running as long as the client is running. Similarly, the second listener is synched to an eight second timer that runs while the client is running. The polling timer listeners are therefore waiting for signals from these two timers indicating that the specified time has elapsed. When the timer listeners are notified that the time has elapsed, they send a message to the server.

The content of the message sent to the server differs between the two timer listeners. One of the timer listeners is intended to be a simple poll to the server. The message content for the poll has a command field of "POLL" and the server responds by letting the client know whether there are any messages waiting in the queue for the client. If there are messages, the server responds with a message that is intended for the client. Furthermore, if there are multiple messages in the queue for the client, the server will indicate such a situation in the message response and the client will poll again immediately. When there are no messages, the server sends an acknowledgement that the client polled the server and the client waits another five seconds before polling again. The other timer listener has

the sole purpose of making sure that the list of clients logged into the server is up to date. Every eight seconds, the update client timer listener, as it is called, sends a message to the server that requests a list of all the clients logged into the server. This update is done for the purpose of inter-client communication. This is important so that all the clients know when a client has logged in or logged out so that the clients know whether a message can be sent to a client or not. These two timer listeners allow the server to send messages to the client and also keep the clients up to date on the status of clients logged into the server during a session.

The reason that the polling has been divided into two separate polls is because there are two distinct messages that must be sent for each poll. The first poll sends a message that the server recognizes as a poll message. The polling time is set to five seconds because that is one of the rules the server places on the clients. The poll initiation only causes the server to see if there are any messages waiting in the message queue for the client sending the poll. Thus, the client has no way of knowing from the poll message which clients are active on the server. Therefore, the second poll has been established to remedy this limitation of the poll response from the server. The second poll therefore sends a message that is used by the client for the sole purpose of see which clients are active on the server. The time of eight seconds was chosen because the server does not place a time restriction on this status check by the client. In fact, the server does not even expect the client to be polling in this way to the server. The longer interval was chosen so after some testing with multiple clients logged into the server. It was a good time to keep the clients list updated often enough for effective chatting across clients as well to keep the load on the server low enough with multiple clients logged in.

Since these two timer listeners are important to the operation of the client, it is essential that the main control module create and instantiate them at the beginning of the client session. What this means is that the main module creates objects appropriately named to indicate the type of timer listener that each object will be when instantiated. When the constructors are invoked for each of the timer listeners, they wait for the signal from the timers. The timers are also created by the main control module and they are setup so that they are registered to their respective timer listeners.

5.3.2 Settings File Window

After setting up polling for the client session, the main control module must continue to setup the interface for the user. In order for the client session to create the interface, the user must choose the settings file to use to create the interface. Thus, the main control module constructs a simple frame with a list containing all of the settings files that are available on the server. Refer to figure 5-2 below to see what the settings file window looks like to the user. It is a list of the settings files currently on the server. The placement of the settings files is described in appendix B of this paper explaining the installation process for the client software for the MEMS characterization system. The frame and list of the settings files that the main control module creates has special functionality as well. In particular, the list of settings files is a selectable list meaning that the user can highlight one of the settings files. There is also a button labeled "load" within the frame that must be pushed by the user to start the parsing of the highlighted settings file.

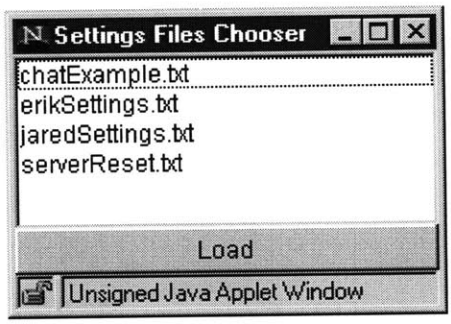

Figure 5-2: Settings File Window

In the process of constructing a frame for the list of settings files, the main control module also sets the look and feel of the interface for the remainder of the session. This allows the graphical components in the interface to look and feel like other graphical components within the current operating system. In addition, the main control module uses the message handler module within the client. This module will be explained in more detail later in this paper. Basically, it allows the client to send and receive properly formatted messages to and from the server. In order to get the list of settings files, the main control module sends a message that requests the information from the server.

5.3.3 Building the GUI

As mentioned above, when the load button on the frame that contains the settings files list, the client begins the parsing of the settings file selected by the user. The entire parsing process was detailed in the previous chapter of this paper. As a note, all of the computation involved in parsing and building the GUI is done on the client side of the system within downloaded client modules. The result of the parsing of the settings file is then used to build the GUI. At this point, the process of building the GUI consists of creating a hashtable that contains the frames that make up the user interface. When this process is complete, the hashtable is returned to the main control module.

At this point, displaying the user interface merely becomes a process of taking all the frames in the GUI hashtable and making them visible. The main control module is given the task of making the frames visible. However, there is one more step in the process that has been saved until this point in the process of running the interface. This step is the process of registering the buttons and menu items to an action listener. The method that Java uses to handle actions from the user is called its event model. In the case of Java version 1.1 and higher, the event model consists of creating an object that can be acted upon and then registering it to another object that listens for an action upon the previous object. Using this event model, it is possible for multiple objects to be listening for action upon a single object. This is not used in the client software, but is a noteworthy aspect of the Java event model. In light of the event model, all of the buttons and the menu items in the frames created from the GUI builder are not automatically registered to any action listeners. Thus, the main control module must make sure that each of the buttons and the menu items defined for the user interface are registered to an action listener. In this case, the main control module is the action listener to which each button and menu item must be registered. Thus, the main control module does an initialization step where each button and menu item in the user interface is registered to the main control module. The result of this process of registering all buttons and menu items to the main control module is that there is a central point through which all user interaction flows, namely the main control module. That is one of the reasons that it is called that main control module because all user interaction goes through the module.

Once all of the objects in the interface are registered to the main control module, the next step in building the GUI for the user is to display all of the frames. Thus, the main control module goes through each of the frames defined in the user interface and causes them to display. At this point, the user can see the interface as defined in the settings file chosen by the user.

5.3.4 Status Window

One part of the user interface that is always displayed when the client starts is the status window, which is shown in figure 5-3 below. The window can be minimized by the user if more space is needed on the desktop, but these settings are important for the user so that the status of the server hardware is known. The information in the status window details the state of the hardware at the server side as well as the parameters for taking data. More specifically, the status window displays the strobe parameters, the position of the microscope stage, the focus value, the stimulus parameters, the magnification value (i.e. the objective number), the wafer name, the series parameters for data acquisition, and the control status for the present client.

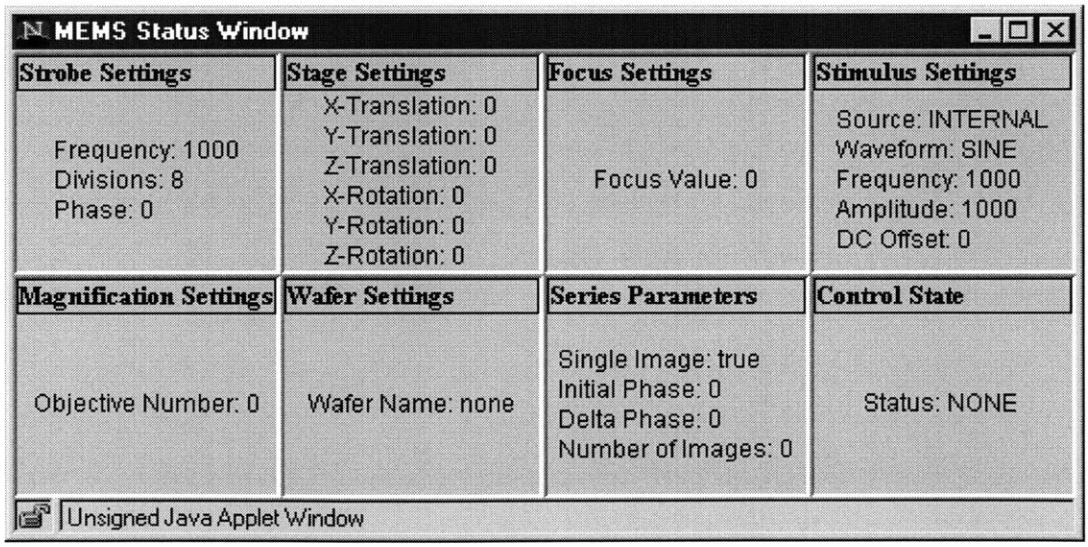

Figure 5-3: MEMS Status Window

In order for the client to obtain the information to display in the status window, it must send a message for each of the above categories of information. This is an initialization that must be done so that the state of the server is known by the client. After that initialization, these status messages do not need to be sent to the server since the server sends any state updates to each of the clients that are in session. However, the user can define a button or menu item that can check these status parameters at any time if desired. As a result of any status checking messages, the status window continues to receive any updates that result from these messages.

5.3.5 Handling Input

With all of the setup done as a result of the main control module, the client session has been started from both the server's perspective and the client's perspective. Thus, the user interface is at a point where it must wait for user input before anymore work is done by the system. Since each of the objects in the user interface are registered to the main control module, whenever there is user interaction, the main control module is notified. When this occurs, the main control module passes the function associated with the button or menu item to the function handler module in the client. This module is detailed in the next section of this paper. The function handler module is defined by a class within the client software and an object of that type is instantiated by the main control module at startup. It is this function handler object that is used by the main control module to handle all user interaction. Thus, with each user interaction, the main control module passes the function parameters as defined in the settings file to the function handler.

Chapter 6

FUNCTION HANDLER

6.1 Module Overview

The function handler is where all work that needs to be done by the client software is accomplished. The class that defines the function handler creates a method for the function handler object that is called "handle." The argument used for the handle method is a string that is used to distinguish the function that must be initiated when the handle method is called. The method is called by the main control module and the argument used is also passed by the main control module. When the handle method is called, the function handler has the task of using the information within the client software to determine how to handle the input given.

The architecture of the function handler module can be described as a control module, the function handler itself, with three helper modules that actually accomplish the work delegated by the control module. A close-up of the function handler section of the client modules is shown in figure 6-1 below. The three helper modules are used because the work that must be accomplished can be divided into three main categories. These categories are hardware control, data control, and data processing. Thus, the three modules each are suited to handle certain types of work that must be done for the user. As the function handler interprets a handle command from the main control module, it calls upon any of the three helpers as required.

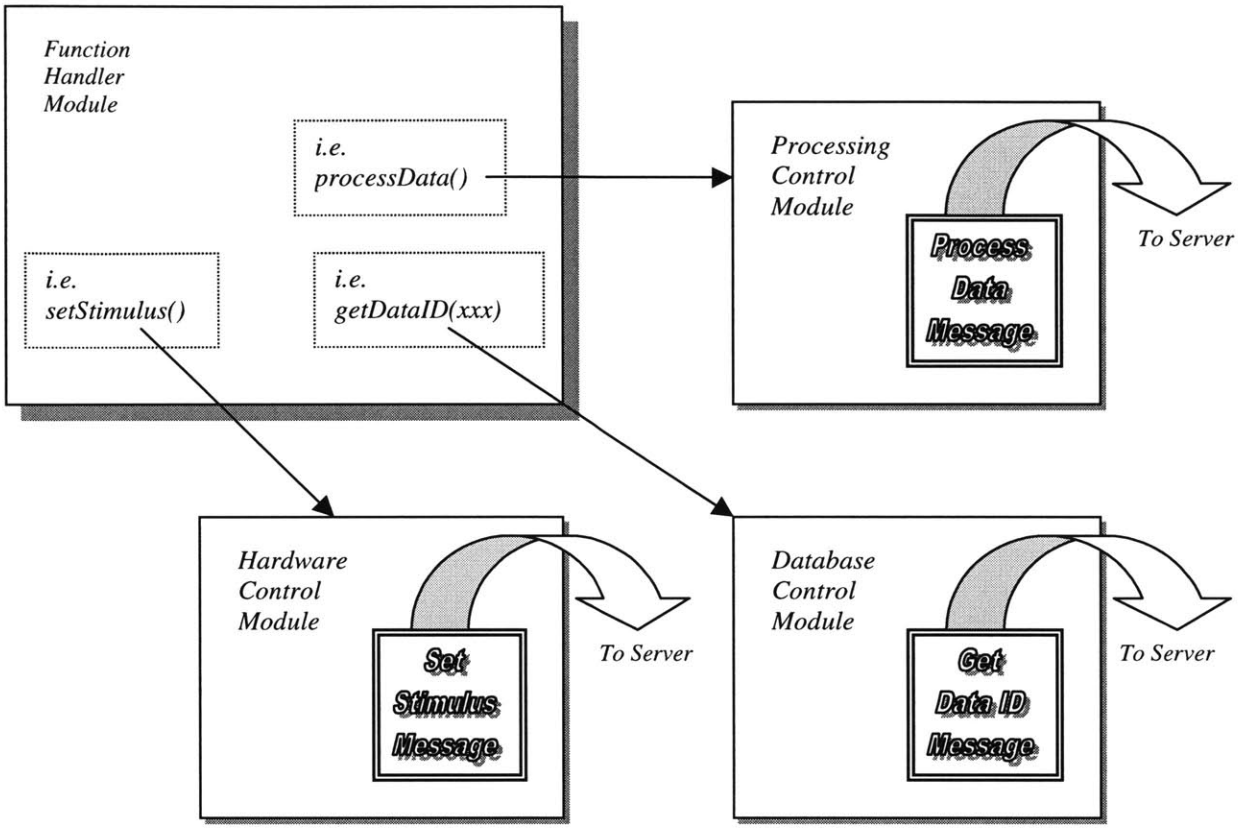

Figure 6-1: Function Handler Modules

6.2 Handling Functions

When the function handler is called upon to perform work within the system, it has to determine the type of function that is being initiated. There are two types of functions that can be initiated within the function handler. The first type is a primitive function while the second type is a user defined function. Primitive functions are called "primitive" because they are the fundamental building blocks of all functionality that can be accessed by the client. User defined functions are called "user defined" because they are defined by the user through the use of the settings file. User defined functions can be made up of either other user defined functions or primitive functions. Eventually, all functions must relate to a set of primitive functions. This way of defining functions allows reuse of common primitive function sets as well as for a method of defining a programmatic interface.

6.2.1 Primitive Functions

As explained briefly in the previous section, primitive functions are fundamental building blocks for the functionality offered in the MEMS characterization system. What this means is that the primitive functions must be utilized at some level when the client desires to accomplish work during a session. They represent the smallest pieces of work that can be accomplished from the user's perspective. In many cases these primitive functions line up with the messages that can be passed between client and server. Thus, the primitive functions allow the client to do such things as take control of the server's resources, change the parameters of the hardware and initialize the processing of the data. The advantage of dividing up the functional capabilities of the system into these fundamental components is that they can be sequenced together to form larger functional structures. Furthermore, these larger functional structures can be used as part of even larger functional structures. This process is the idea behind user defined functions which are explained in the next subsection. For the entire list of primitive functions available to the user as well as the work performed by each function, see appendix A.

Primitive functions are handled directly by the function handler. More specifically, when the function handler is asked to handle a function, it figures out whether the function is a primitive function or a user defined function. When the function handler finds that it has been asked to handle a primitive function, it makes a direct call to the desired function within itself. This is because all of the primitive functions have corresponding private methods within the function handler object. This means that the methods are not accessible outside of the function handler object. These methods will then make calls upon any of the three helper modules that are used by the function handler module. What this creates is an interface for accessing these functions that consists of going through the proper means of invoking a function through the handle method of the function handler.

As was stated earlier, all functions must eventually reach a point where they are not user defined and must be associated with a primitive function. Thus, if a function is encountered that is not defined among the user defined functions, it is treated as a primitive function. However, there are cases when

a user may define a function somewhere within the interface that may not be a user defined function or a primitive function. In such cases, the function handler treats them as non functions. The function handler will do nothing when it encounters a function that it cannot resolve. This allows the user interface to continue to run without causing errors if the user makes a mistake in defining functions in the settings file.

6.2.2 User Defined Functions

User defined functions are the cornerstone of the programmatic interface. Also, they are the means by which blocks of primitive functions can be grouped together. User defined functions are sequences of either other user defined functions or primitive functions. These user defined functions are defined by the user in the settings file that is used to create the interface for each client session. Thus, as required, these user defined functions can be modified to accommodate different sets of functions through the settings file.

The function handler takes care of user defined functions by calling the handle method recursively. This means is that if the function handler finds that it is called upon to handle a user defined function, it finds the first function within the set of functions that make up the user defined function and calls the handle function of the function handler for that function. If that function also happens to be a user defined function, the above process will be repeated. However, if the function is a primitive function, then the function handler will finally perform the native function as required. This process goes on for each function handled until there are no more functions left to be handled. This method of handling the functions allows user defined functions to contain other user defined functions because they are handled the same with each iteration. As a result, each function must eventually lead to a non-user defined function so that the function handler will accomplish work. Also, this process forms the programmatic interface which is explained in more detail later in this paper.

6.3 Hardware Control Module

The hardware control module is one of the helper modules used by the function handler module. It is also the most extensively used helper as well as the one with the most functions available of any of the helper modules. The reason for these two facts is that there are more messages having to deal with hardware control than with any other type functionality within the server.

The function handler calls upon the hardware control module to deal with the server in all areas concerning the hardware at the server side. Thus, the hardware control module is used in such cases when the microscope stage must be adjusted and when the series parameters for capturing data must be modified. Furthermore, the hardware control module takes care of all other parameters displayed in the status window of the user interface. See figure 5-3 for a diagram of the all of the parameters displayed in the status window. The structure of the hardware control module is defined in a class. The class defines an object with no initialization upon construction of the object. The whole object is made up of methods that are called by the function handler module. The function handler module creates a hardware control module when it starts up so that it has an object of that type to use to interface with the server hardware. Each of the methods in the hardware control module are of a similar structure. Each method must first check the control state for the client desiring to control the hardware. If the client has control of the hardware, then the hardware control module proceeds to send a message to the server to initiate a change in state of the server hardware. After that has been done, each method appropriately handles the response from the server whether it responds in error or merely acknowledges that the hardware state has been changed as requested. With that process complete, the hardware control module waits for another call from the function handler.

6.4 Database Control Module

The database control module is the second of the three helper modules for the function handler. It is not as dense in functionality as the hardware control module since the messages in the database control area are not as developed as those in hardware control. However, the database control is an

important part of the client software since it allows the user to manage the database. It is also the link between the acquisition of images from the MEMS devices and analysis of those images.

The class file that defines the database control module is very much like the one that describes the hardware control module. There are no initialization steps to be done when a database control module object is constructed. The entire object consists of methods that can be called by the function handler. As with the hardware control module, a database control module object is created when the function handler is started so that it has access to the functionality provided by the database control module. The structure of each of the methods in the database control module are identical to those in the hardware control module with one major exception. Each method in the database control module does not check the control state of the client. This is because it is not necessary for the client to have control of the server's resources in order to access the database. Thus, a client can potentially log into the server and just do database management while another client actually has control of the server's hardware.

The functionality contained in the database control module allows the client to find out information about data present in the database as well as create and delete pieces of data from the database. The interface used in the database is SQL (Structured Query Language). This is a standard language understood by many major databases and allows access to tables which describe data in a flat database. For the MEMS characterization system, the images and other data are saved in a flat database while the information for each piece of data is stored in a table that can be accessed via SQL calls. That is how the database control module works with the database in the system to allow the user to do database manipulation.

6.5 Processing Control Module

The least developed of the helper modules for the function handler is the processing control module. The reason for this is that there are very few messages that can be passed between client and server that pertain to processing. The processing of data in the MEMS characterization system consists of telling the server which pieces of data to process. At this point, the only parameters included in the

processing are the region of interest for the images that are to be processed. However, the interface is such that more parameters can be used as desired when processing at the server side increases in functionality.

The class that defines the processing control module is very similar to the class the defines the database control module. The processing control module does not need to check the control state of the client since it does not need control of the server's hardware in order to perform its function. This allows multiple clients to be logged into the server while using the database management functionality and/or the processing functionality at the same time. Of course, all of this multiple use of the server's functionality will come at a cost of slowing the server down as it processes multiple requests from clients.

6.6 Error Handler

The error handler module is responsible for taking care of any errors that occur as the client and server interact with each other. It is actually a part of the polling package in the physical structure of the client software. However, the function handler, and more specifically the helper modules for the function handler, uses the error handler module extensively. The error handler module is one of the static modules within the client software. This means that all of the methods in the error handler are available to any other object in the client interface as long as the polling package is available to that object. The reason that this is done for the error handler is that since there are multiple modules within the client software that access the error handler, making the methods persistent allows them access without having to create an error handler object of their own.

The job of the error handler is to indicate to the user when an error has occurred during communication between the client and the server. Thus, when an error message is received from the server, the error handler displays an appropriate dialog box to the user to indicate that an error has occurred. Also, the message displayed is different for each type of error that occurs. Furthermore, when a message is sent by the server that the client does not understand, there is an indication to the user that the client does not know how to deal with that particular message. This is not supposed to

happen, but when it does, it is an indication of some sort of error in the system and that it must be fixed. The modules that use the error handler include the polling timer listeners as well as the three helper modules for the function handler.

Chapter 7

IMAGE TOOLS

7.1 Overview

Image tools in the client software make up a very small portion of the software. There are basically two classes that define objects that are used to load images and display images for the user interface. More functionality can be included in future versions of the software. Further development of this functionality would include adding the ability for the user interface to zoom and rotate images that are displayed to the user. Also, filters for the images could be added. Since, this was not the focus of this work, these functions were not added as part of the image tools. Instead, they simple consist of an image loader and image displayer.

7.2 Image Loader

The image loader is a simple object that has no initiation upon construction. Only one method exists in the object and it has an argument for the URL to use to load an image as well as an argument for the image name. When an image loader object is created and the method is called, the image as indicated by URL and name are stored locally within the client software. This is done so that the when image is to be displayed, the image displayer module can load it locally from within the client software. Thus, the image loader performs the process of loading an image from the server to the client.

7.3 Image Displayer

The image displayer works with the image loader. The constructor for the image displayer module contains nothing because nothing needs to be done when the object is created. There is only one method in the object that is used to display an image. There are three arguments that must be supplied

to the method when it is called. The first argument is the image that is to be displayed and this is usually available as a result of using the image loader module prior to displaying the image. The second argument is the name of the data window where the image is to be displayed. When there are multiple data windows in a user interface, the user can choose between frames to decide which one is the active frame. The active frame is where all images are displayed in the user interface. When the image displayer is used, any of the data windows can be selected for display. However, whenever it is called in the client software, the image is displayed in the active window. The third argument in the method within the image displayer is a boolean value that indicates whether the frame should be repacked after the image is placed in the data window. This is important so that the frame will be resized to make sure that the image will be displayed in its entirety. After the method is called, the image will be visible in the active data window in the user interface. When an image is displayed in a data window, it will be resized to fit the image. However, if the user resizes the window so that the whole image cannot be displayed in the designated area, scrollbars will appear so that the image can be viewed in its entirety by scrolling vertically or horizontally.

Chapter 8

MESSAGE DATABASE

8.1 Structure of the Database

The message database is actually a static object in the client software. It contains skeletons for all of the messages that must be used to communicate with the server. These skeletons consist of special arrays of objects that are name-value pairs where only the unchanging name-value pairs are included. The format of the messages is such that there is a series of field names associated with a values. Thus, the array of name-value objects encompasses this structure. Each message is a different length so each array of name-value objects is set according to each message. Furthermore, the database of messages consists of a hashtable with each message associated with a key value equal to the message name. This allows the message skeletons to be accessed from the message database through the use of the proper key value.

8.2 Accessing the Messages

The main advantage of setting up the messages so that they are in a database like structure is that they can be accessed and copied as if they were in a real database structure. As described in the previous section, the database is actually a hashtable where each message skeleton can be retrieved through the use of the proper key value. The message names for all of the messages supported in the present system can be found in appendix A of Jared Cottrell's thesis paper entitle *Server Architecture for MEMS Characterization* [12] which describes the server side of the MEMS characterization system. The same message names have been used to setup the hashtable keys for use in the message database for the client software. Thus, when a message skeleton is needed from the database, the process merely requires knowing the name of the message desired. Since the message database is a static structure within the client software, it can be accessed as long as the package is accessible for a class that needs

to use it. The modules in the client that use the message database most extensively are the helper modules for the function handler. The reason for this is that they are the modules in the client that must perform work meaning that they must send the proper messages as required.

Chapter 9

GLOBALS MODULE

9.1 Structure

The globals module in the client software is a small part of the overall structure of the software, but it servers a major purpose in providing a common communication point for various modules within the client. The globals module is a place where persistent session information is stored for use by any of the modules in the client software. Most of the classes import the globals package so that they have access to the information in the globals module. The globals module is basically a class that has public variables which allow access by any of the modules. As an example of the type of information stored in the module, the session identification for the current client session is stored there as well as any information pertaining to the state of the server hardware. This is the type of information that has a life span of the entire session of the client as it interacts with the server.

9.2 Usage of Globals Module

The globals module is used by many of the modules within the client software. The module allows information to be shared by any module within the client software by consisting purely of publicly accessible variables. The downside of allowing all access is that any of the modules can change the state of the these variables. This makes it important for each of the modules to be careful about the way that the global values are modified. One of the most obvious uses of the globals is through the helper modules for the function handler. These helper modules must send messages to the server in order to communicate and to accomplish work for the user. From the message database, these helper modules get copies of message skeletons that must be filled in. The most used piece of information from the globals modules is the session identification since it must be sent with every message sent to the server. Thus, after login, the globals module is accessed with every message sent to the server

from the client. Furthermore, when the helper modules need to modify the state of the server hardware, the values that it includes in the necessary messages come from the globals module. These local values in the globals module are the ones that get set by the user. Then, when the client has to tell the server to change its state, the local values are used to cause that change. In addition, the status window displays values from the globals module when the server returns messages to indicate that its settings have changed. In essence, the globals module maintains local state for the client.

Another purpose that the globals module serves is the synchronize and control message passing between the client and the server. This is done through a variable in the globals module that indicates whether the client is waiting for a response from the server or not. Thus, whenever a message is to be sent to the server, the module trying to send the message must check this lock variable within the globals module. This lines up the message passing so that there are no mistakes in message response handling. More specifically, each response can only line up with the most recent message sent because a new message cannot be sent until the current message has received a response. Thus, the globals module is the center point of communication in order to synchronize message passing from the client to the server.

Chapter 10

MESSAGE HANDLER

10.1 Overview

The message handler is the part of the client software that handles each message that is sent to and received from the server. It is used by each of the helper modules that work with the function handler. This is because each of the helper modules must send messages to the server in order to do work for the user. The necessity for a message handler in the client software is a result of the way that the client and server communicate with each other. Since all communication is done through the use of messages, it is necessary at some level to have a message handler to handle the communication and to centralize this function within the client.

10.2 Send Message Module

The message handler is made up of one module that is used to send the messages. It consists of two closely related public methods that can be used by any module that has access to the message handler package. The first of the methods with the object is one that is used to login to the server. It has arguments that require a message in the form of an array of name-value pairs and a URL to which the login must be done. The second of the methods in the send message module is the one that actually does the sending of the messages. The arguments that it requires also include a message in the form of an array of name-value pairs and a URL to which the message must be sent. The reason that these methods are closely related is that the login method actually uses the send method to send the login message to the server.

It may seem strange to include a method to just perform a login to the server. However, this method allows the main control module to simply call the method as one of the steps of initialization of the

user interface. Furthermore, the login message is unique because it does not contain a field for the session identification. This is because the session identification is assigned to each client upon login and is therefore unknown before login occurs. Thus, the method can be hard coded since nothing in the message changes over time.

The method in the message handler that actually sends the message is used to send all of the messages that client sends to the server. It also receives the response from the server from each message. Thus, it is one of the most important parts of the client software. From the input at the argument end of the method, information is known about the message content as well as the message destination. When a method is called upon to send a message, it is only done so in the case when the client is not already waiting for another method. As explained earlier, this is done though a variable in the globals module. As soon as the method is invoked, the variable in the globals module that indicates that the client is waiting for a response is modified to indicate that the client is indeed waiting for a response from the server. Then, there is a conversion done on the message that is to be sent so that it is actually URL encoded. With that done, a connection to the specified URL is established and the message is sent. As soon as the response is received from the server, the lock variable in the globals module is modified so that all other modules know that the message handler is no longer waiting for a response. This frees up the message handler to send another message. It should be noted that the only module that actually modifies the value of the lock variable is the message handler. This is to ensure that no other modules may make a mistake to throw off the synchronization of sent and received messages.

10.3 HTTP Usage

One important aspect to note about the communication link between the client and the server is that all communication travels on top of HTTP (Hyper Text Transfer Protocol). This protocol is a mature standard that is used to transfer information across the internet every day. All web page interaction is communicated using the protocol. Thus, rather than try to develop another protocol, the basis for communication was chosen to be HTTP. For the purpose of the MEMS characterization system, only a few of the aspects available in the HTTP protocol are used. More specifically, there are definitions in HTTP protocol that involve getting data from and posting data to servers. However, the only part of

these definitions used is the HTTP POST. This is because all messages sent to the server are through an HTTP POST. Furthermore, all information that is to be retrieved from the server is simply downloaded using built in Java functions. All information is passed using the POST from the client and the response of the POST from the server. The focus of this work was not on the HTTP protocol. Thus, for more information the protocol, see the references on HTTP [13,18].

10.4 HTTP Client Classes

Since the focus of this project was not on working with protocols and developing those aspects of communication that have been previously established, a package of classes for HTTP client communication was used. The classes actually defined much more than needed for the developed client, but using the classes proved much more efficient than trying to develop custom classes that would adhere to HTTP. Thanks goes out to Ronald Tschalär of Innovation GmbH of Zürich, Switzerland [18]. The classes were obtained from http://www.innovation.ch/java/HTTPClient/ free of charge and with very good documentation.

Chapter 11

CUSTOM GUI CLASSES

11.1 Explanation

The phrase "custom GUI classes" is used to describe a group of Java classes written specifically for the client software. A little background is required here. When the software for this work was in full swing, Sun's released Java Development Kit (JDK) was version 1.1.5. Furthermore, there was a separate package of classes that Sun had developed to enhance the graphical components of the Abstract Windowing Toolkit (AWT) that was the standard in Java. This package is called Swing and its function and purpose was briefly explained in the system overview section of this paper. Swing, with its extended functionality contains all of the graphical components that are being used right now for GUI development under Java. However, the status of the Swing components at the time when this software was started was not known. Thus, these custom GUI classes were created. They are simply classes that define each of the graphical components used in the client software such as buttons, frames, menu items, and panels. Each class is denoted as MEMS*x*.class where the *x* is the name of the component such as a button. As an example, the buttons that are used throughout the user interface are instances of the MEMSButton class. These custom classes are used so that no matter what type of new packages come out such as Swing, they can be incorporated into the user interface. The only modifications that must be made include having the custom classes extend the new component classes and making sure that the method calls and constructors line up correctly.

Chapter 12

PROGRAMMATIC INTERFACE

12.1 Overview

The idea behind a user interface is simple. It is fundamentally just a means for a user to interact with a system in some form. There are different ways that this can be accomplished. For the most part, the focus of this paper has been on the graphical side of the user interface that has been developed for the MEMS characterization system. However, there is another side of the interface that has been developed as a part of the client side of the system. This is the programmatic interface. The programmatic interface is meant to allow the user to perform work without having to push each of the buttons in the graphical interface to do the work. The purpose is to allow the user to run a custom "program" for use in the characterization system.

There are different ways to think of a programmatic interface. One way is to think of it as a method of interaction with the client through a user written program. In the case of the present client, this certainly can be done. This is because the process to perform work within the characterization system involves interfacing with the function handler module in the client software. However, the program would have to be written in Java and the user would have to know about the internal structure of the software in order to create the program. Thus, another way to think of the programmatic interface is to use the graphical user interface to establish a connection with the server and then perform work through the use of the user defined functions. This method requires the user to have no knowledge of how the functions are handled but it is not a "pure" programmatic interface. This is because the user must login and then load their user interface which performs the work.

12.2 Usage of Programmatic Interface

Since there are two different ways to approach the programmatic interface of the client, there are different ways to use the programmatic interface. The first method of interfacing with the server through a programmatic method involves writing an object that interacts with the client software. Since this method of interfacing in a programmatic way with the server requires knowledge of the internal structure of the client software, it will not be discussed in this paper. Furthermore, this method would introduce problems with control of the server resources that would have to be alleviated by the user. This makes the strategy quite difficult. The second method of programmatically interfacing with the server involves tools and skills that user has at their disposal. Thus, the usage of the second method of programmatically interfacing with the server will be covered here.

The idea used extensively in order to access the programmatic side of the client software is user defined functions. These were explained in the custom interface modules section of this paper. In order to understand how they are used to perform programmatic interface type actions with the user interface, only more complex uses of user defined functions must be presented here. To review, user defined functions are functions that the user can build from any number of primitive functions or other user defined functions. This allows the user to put functions in sequence that can perform the work required to accomplish an entire data acquisition session, just one primitive function or anything in between. Also, commonly used groups of primitive functions can be put together into a user defined function so that operation of the system can be streamlined. An important aspect of the programmatic interface available within the client is that the user usually needs to have control of the server's resources in order to accomplish work in terms of taking data. Thus, the user must provide a way within the interface to gain control before launching a "program" from the interface.

An important aspect using the programmatic interface is the use of nested user defined functions. The idea is that the user can define functions that contain other user defined functions. While this requires thought in order to build these, the process is simple. When using the nested user defined functions, it is important to make sure that eventually, there are primitive functions called or else there will be no actual work accomplished. If that requirement is not met, the user interface will not malfunction, but

it will end up not trying to communicate with the server at all. Thus, these aspects require planning and foresight on the part of the user creating the user defined functions. It is this flexibility that adds both complication and power to the interface and it can be used in creative ways that I have not even though of yet.

Thus, the programmatic interface requires nothing more than properly defining user defined functions. The important aspects to remember when utilizing this aspect of the user interface are that nested user defined functions can be used effectively and the user usually has to have control of the server's resources when running a user defined program.

Chapter 13

CONCLUSION

13.1 Overview of Interface

The user interface presented in this work is merely one of the many user interfaces that can be created for the MEMS characterization system. This is because any program that can send properly formatted messages to the server can interact with it. This is the first revision of a Java based GUI that has the ability to perform programmatic functions as well. The goal of this work was to create a user interface for the MEMS characterization that allows multiple users to access the server at the same time. This work accomplishes that task and gives the user tools with which to collect and analyze data. There are some future enhancements that may be added to the this work in the future. These are discussed in the next few sections.

13.2 Extended Processing Capabilities

The processing options for the data taken on the MEMS devices are quite limited. The message that gets passed from the client to the server in order to initiate the processing of data includes a field where the client must specify the data identifier for the data as well as an open field to add any analysis parameters required for the analysis process. At this point, the parameters may include the region of interest in the data taken from the MEMS device, as well as the type of information requested back from the server. These simple parameters will have to be enhanced in order to accomplish more complicated analysis on the data.

13.3 User Modification Enhancement

The amount that the user can modify the user interface has been limited so that the amount of arguments and parameters that the user must include in the settings file is not enormous. This also limits the amount of total control that the user has over the appearance of the user interface. Future enhancements to this interface may include a more mature user modification system to allow the user more control over appearance of the user interface.

13.4 Graphic Filters and Processing Capabilities

One area of the user interface the could use much enhancement is the graphical manipulation abilities of the interface. More specifically, images cannot be cropped or filtered or saved back to the server in any modified format. This is not important at this stage of progress within the system, but it may become more important later on in the development of the system.

13.5 Final Thoughts

The work accomplished for the writing of this thesis has been and probably will be a work in progress for a long time. With each new step of progress made in order to get the user interface to act properly, there were new bugs and quirks that had to be sidestepped. Thus, as each hurdle was jumped there were certainly places where all the goals were not met. However, this represents a large portion of the functionality required to operate the MEMS characterization remotely and by multiple users. Thus, my goals for this work have been accomplished.

Appendix A

PRIMITIVE FUNCTIONS

A.1 Introduction

Primitive functions are the building blocks for all functionality in the client side of the software. Furthermore, each menu item and button must be associated with either a user defined functions or primitive functions. User defined functions can be further broken down into more user defined functions or primitive functions. When each user defined function is traced out, there must be primitive functions being called at the lowest level of the trace. Thus, it is important for the user to know what primitive functions are available and also what each function does within the MEMS characterization system. This appendix lists out each primitive function along with the work that the function accomplishes when called.

A.2 Primitive Functions

Function Name	*Function Description*
openChatWindow(*arg1*, *arg2*)	Opens the chat window in the current GUI. Used for interclient communication. Arguments correspond to the number of lines created for chat space and the number of lines for active clients space in the chat window respectively.
openSettingsWindow()	Opens the settings window in the current GUI. Used for setting the server hardware from client.
openActiveDisplayWindow()	Opens window to allow user to set active display window for displayed data.
setActiveDisplayWindow(*windowVariable*)	Sets the active display window. Argument *windowVariable* corresponds to a valid data window variable name as defined in the settings file. Otherwise, first data window defined in settings file is used as active display window.
takeControl()	Takes control of server's hardware. Must be done

	before any of the hardware parameters can be changed by the client.
checkControl()	Checks the control status of the server's hardware and updates the status window in GUI.
cedeControl()	Relinquishes control of the server's hardware. Should be done after resources are no longer needed by client. Automatic loss of control occurs upon logout of a client.
shutdown()	Initiates a shutdown of the server. Requires control of the hardware to initiate.
reset()	Resets the server's hardware to an initial state. Requires control of the hardware to initiate.
setStrobe()	Uses the current local strobe settings to set the server's strobe settings. Requires control of the hardware to initiate.
getStrobe()	Gets the server's strobe settings and updates status window in GUI.
openStrobePanel()	Used when opening the settings window in GUI. Tells the GUI to display the strobe panel when the settings window is opened. Must be called prior to calling the openSettingsWindow() function.
setStage()	Uses the current local stage settings to set the server's stage settings. Requires control of the hardware to initiate.
getStage()	Gets the server's stage settings and updates the status window in GUI.
openStagePanel()	Used when opening the settings window in GUI. Tells the GUI to display the stage panel when the settings window is opened. Must be called prior to calling the openSettingsWindow() function.
setFocus()	Uses the current local focus settings to set the server's focus settings. Requires control of the hardware to initiate.
getFocus()	Gets the server's focus settings and updates the status window in GUI.
openFocusPanel()	Used when opening the settings window in GUI. Tells the GUI to display the focus panel when the settings window is opened. Must be called prior to calling the openSettingsWindow() function.
autoFocus()	Initiates the server's auto focus routine.
setStimulus()	Uses the current local stimulus settings to set

	server's stimulus settings. Requires control of the hardware to initiate.
getStimulus()	Gets the server's stimulus settings and updates the status window in GUI.
openStimulusPanel()	Used when opening the settings window in GUI. Tells the GUI to display the stimulus panel when the settings window is opened. Must be called prior to calling the openSettingsWindow() function.
setMagnification()	Uses the current local magnification settings to set server's magnification settings. Requires control of the hardware to initiate.
getMagnification()	Gets the server's magnification settings and updates the status window in GUI.
openMagnificationPanel()	Used when opening the settings window in GUI. Tells the GUI to display the magnification panel when the settings window is opened. Must be called prior to calling the openSettingsWindow() function.
setWafer()	Uses the current local wafer settings to set server's wafer settings. Requires control of the hardware to initiate.
getWafer()	Gets the server's wafer settings and updates the status window in GUI.
calibrate()	Initiates a calibration routine on the server's hardware.
setSeriesParameters()	Uses the current local series parameters settings to set server's series parameters settings for the process of taking data. Requires control of the hardware to initiate.
getSeriesParameters()	Gets the server's series parameters settings and updates the status window in GUI.
openSeriesParametersPanel()	Used when opening the settings window in GUI. Tells the GUI to display the series parameters panel when the settings window is opened. Must be called prior to calling the openSettingsWindow() function.
capture()	Initiates a capture of data according to the series parameters settings. Requires control of the hardware to initiate.
toggleCaptureDisplay()	Toggles the display of captured data in the active data window of the GUI.

stageTransDecX()	Decrements the x-translation value of the stage of the microscope by one increment. Requires control of the hardware to initiate.
stageTransIncX()	Increments the x-translation value of the stage of the microscope by one increment. Requires control of the hardware to initiate.
stageTransDecY()	Decrements the y-translation value of the stage of the microscope by one increment. Requires control of the hardware to initiate.
stageTransIncY()	Increments the y-translation value of the stage of the microscope by one increment. Requires control of the hardware to initiate.
stageTransDecZ()	Decrements the z-translation value of the stage of the microscope by one increment. Requires control of the hardware to initiate.
stageTransIncZ()	Increments the z-translation value of the stage of the microscope by one increment. Requires control of the hardware to initiate.
stageRotDecX()	Decrements the x-rotation value of the stage of the microscope by one increment. Requires control of the hardware to initiate.
stageRotIncX()	Increments the x-rotation value of the stage of the microscope by one increment. Requires control of the hardware to initiate.
stageRotDecY()	Decrements the y-rotation value of the stage of the microscope by one increment. Requires control of the hardware to initiate.
stageRotIncY()	Increments the y-rotation value of the stage of the microscope by one increment. Requires control of the hardware to initiate.
stageRotDecZ()	Decrements the z-rotation value of the stage of the microscope by one increment. Requires control of the hardware to initiate.
stageRotIncZ()	Increments the z-rotation value of the stage of the microscope by one increment. Requires control of the hardware to initiate.
setStageTransX(*arg*)	Sets the x-translation value of the stage of the microscope to the value in *arg*. Requires control of the hardware to initiate.
setStageTransY(*arg*)	Sets the y-translation value of the stage of the microscope to the value in *arg*. Requires control of the hardware to initiate.

setStageTransZ(*arg*)	Sets the z-translation value of the stage of the microscope to the value in *arg*. Requires control of the hardware to initiate.
setStageRotX(*arg*)	Sets the x-rotation value of the stage of the microscope to the value in *arg*. Requires control of the hardware to initiate.
setStageRotY(*arg*)	Sets the y-rotation value of the stage of the microscope to the value in *arg*. Requires control of the hardware to initiate.
setStageRotZ(*arg*)	Sets the z-rotation value of the stage of the microscope to the value in *arg*. Requires control of the hardware to initiate.
setSeriesParamsSingleImage(*arg*)	Sets the value of single image parameter in the series parameters to the value in *arg*. The value of *arg* must be either *true* or *false*. Requires control of the hardware to initiate.
setSeriesParamsInitialPhase(*arg*)	Sets the value of the initial phase parameter in the series parameters to the value in *arg*. The value in *arg* must be an integer less than or equal to the number of phases in the strobe. Requires control of the hardware to initiate.
setSeriesParamsDeltaPhase(*arg*)	Sets the value of the delta phase parameter in the series parameters to the value in *arg*. The value in *arg* must be an integer less than or equal to the number of phases in the strobe. Requires control of the hardware to initiate.
setSeriesParamsNumberOfImages(*arg*)	Sets the value of the number of images to take in the series parameters to the value in *arg*. Requires control of the hardware to initiate.
setFocusValue(*arg*)	Sets the focus value of the microscope to the value in *arg*. Requires control of the hardware to initiate.
setMagnificationValue(*arg*)	Sets the magnification value of the microscope to the value in *arg*. The magnification value is equivalent to objective number so value in *arg* must be an integer. Requires control of the hardware to initiate.
setStrobeFrequency(*arg*)	Sets the frequency of the strobe hardware to the value of *arg*. The value of *arg* must be an integer. Requires control of the hardware to initiate.
setStrobeDivisions(*arg*)	Sets the number of divisions of the strobe hardware for data capture to the value of *arg*. The

	value of *arg* must be an integer. Requires control of the hardware to initiate.
setStrobePhase(*arg*)	Sets the strobe phase for data capture to value of *arg*. The value must be an integer. Requires control of the hardware to initiate.
setStimulusSource(*arg*)	Sets the source for the MEMS stimulus signal to the value of *arg*. The value of *arg* must be either *INTERNAL* or *EXTERNAL*. Requires control of the hardware to initiate.
setStimulusWaveform(*arg*)	Sets the stimulus waveform to the value of *arg*. The value of *arg* must be a valid waveform. Requires control of the hardware to initiate.
setStimulusAmplitude(*arg*)	Sets the stimulus waveform amplitude to the value of *arg*. The value of *arg* must be a valid voltage level. Requires control of the hardware to initiate.
setStimulusDCOffset(*arg*)	Sets the stimulus waveform DC offset to the value of *arg*. The value of *arg* must be a valid voltage level. Requires control of the hardware to initiate.
setWaferName(*arg*)	Sets the wafer name of the current MEMS device to the value of *arg*. The value of *arg* must be a string. Requires control of the hardware to initiate.
markData()	Initiates mode in user interface that allows the user to mark the region of interest on an image for analysis.
markDataDirect(*arg1*, *arg2*, *arg3*, *arg4*)	Sets the region of interest for image analysis. The arguments correspond to the x and y-coordinates of the top left corner and the bottom right corner of the region respectively.
openDataBrowser()	Opens the data browser in the user interface. Used to search for data in the database according to certain parameters.
openDataManagementWindow()	Opens the data management window in the user interface. Used to delete, replicate, create, move data elements in database.

Appendix B

CLIENT SOFTWARE INSTALLATION

B.1 Introduction

The software for the client side of the MEMS characterization system must be installed on the server in order for the user to be able to take advantage of its functionality. The process is simple, but it is beneficial for the reader to know how to accomplish the installation. That is the purpose of this appendix. It also points out the locations for some of the key files used when the system is being used such as where the settings files are located and where they need to be placed in order for the user to take advantage of them.

B.2 Installation Process

The installation process for the client software requires that there is already a web server running on computer where the client is to be installed. Furthermore, the computer where the server is installed must also have the server side of the MEMS characterization system installed. This is the software that was written by Jared Cottell that has the purpose of interacting with the client via messages. With both of those requirements taken care of, the next step is to install the client software.

Since the software was written in Java, the organization of the classes are in packages. These packages represent directories where the classes can be found off of a set root directory on the system. In its present structure, all of the directories and subdirectories are to be placed within a main directory called *UI* off of the *public_html* directory on the web server machine. The *public_html* directory is located within the software for the web server's main directory. Figure B-1 shows this structure in graphical form as it would appear in *Windows Explorer*. The figure shows what the *UI* directory looks like when all of the client classes have been copied there. It further shows where the settings files

must be placed on the server in order for the use to have access to them. They must be placed in the *guiSettings* directory. The installation, i.e. simply copying the classes, is complete once these steps have been taken. In order to run the client, a browser must point to the correct URL for the server machine and include a subdirectory at the end of the URL called *ui*. This will start the client as described earlier in this paper.

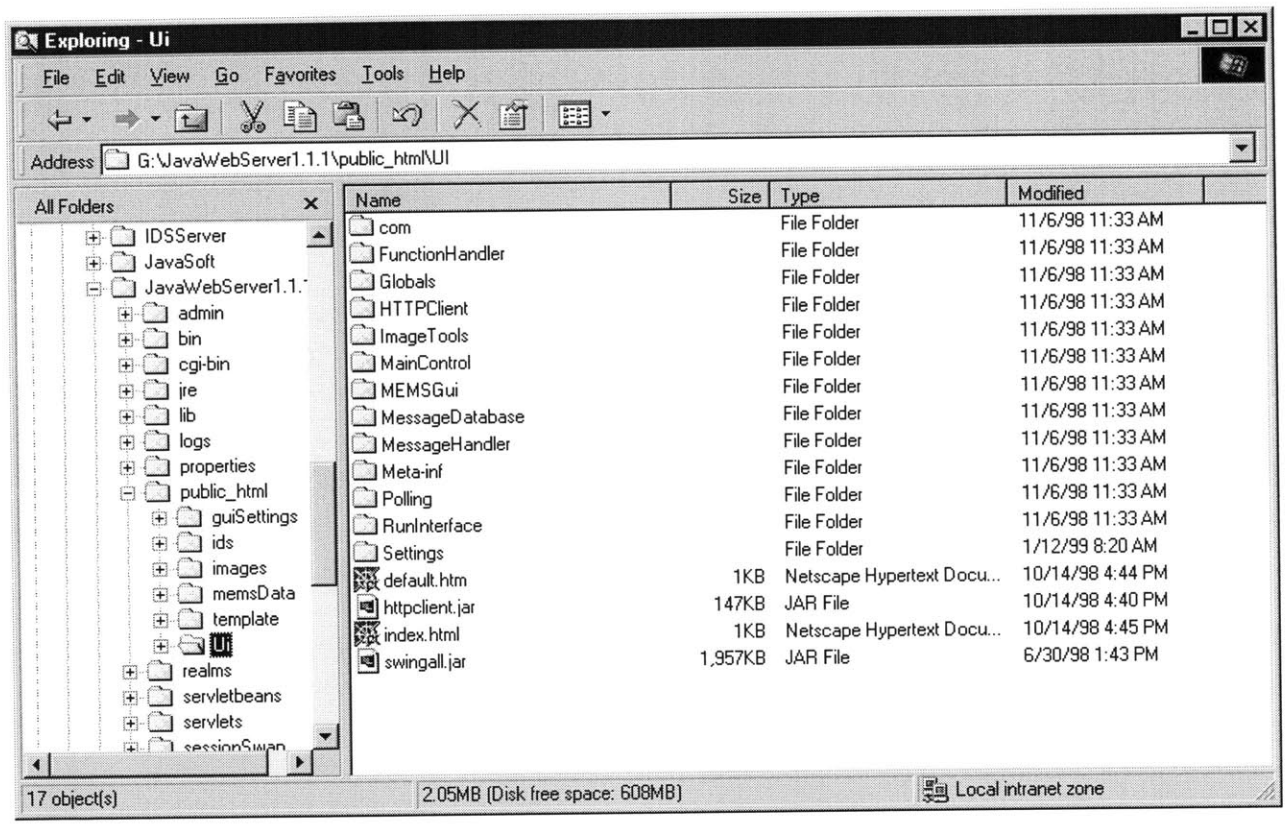

Figure B-1: Directory Structure of Server

Bibliography

[1] D.M. Freeman and C.Q. Davis, "Using Video Microscopy to Characterize Micromechanics of Biological and Man-Made Micromechanics (invited)," Technical Digest of the Solid-State Sensor and Actuator Workshop, Hilton Head Island, June 1996, pp161-167.

[2] http://umech.mit.edu/freeman/talks/sssaw96/talk.html, "Using Video Microscopy to Characterize Micromechanics of Biological and Man-Made Micromechanics (invited)" by Dennis Freeman and C. Quentin Davis, June 1996.

[3] Z.Z. Karu, "Fast Subpixel Registration of 3-D Images," MIT, September 1997.

[4] C.Q. Davis, "Measuring Nanometer, Three-Dimensional Motion with Light Microscopy," MIT, May 1997.

[5] http://umech.mit.edu/info/eck.html, "Instructions on Using Flick," 1997.

[6] James Kao, D.E. Troxel, and Somsak Kittipiyakul, "Internet Remote Microscope," CAPAM Memo No. 96-12.

[7] Somsak Kittipiyakul, "Automated Remote Microscope for Inspection of Integrated Circuits," CAPAM Memo No. 96-9.

[8] Manuel Perez, "Java Remote Microscope for Collaborative Inspection of Integrated Circuits," MIT, May 1997.

[9] http://nirvana.mit.edu/emsim/index.html, "MIT EmSIm Electromigration Simulator" by Mathew Verminski, 1997.

[10] J.C. Carney, "Message Passing Tools for Software Integration," MIT, June 1995.

[11] http://java.sun.com/docs/white/index.html, "The Java Language Environment White Paper" by James Gosling and Henry McGilton, May 1996.

[12] J.D. Cottrell, "Server Architecture for MEMS Characterization," Master's Thesis, September 1998.

[13] http://www.w3.org/Protocols/HTTP/HTTP2.html, "Basic HTTP as Defined in 1992" by Tim Berners-Lee, 1996.

[14] D. Flanagan, "Java in a Nutshell," 2nd Edition, O'Reilly Associates, May 1997.

[15] http://www.javasoft.com/products/jfc/tsc/index.html, "The Swing Connection" by Sun Microsystems, 1998.

[16] http://www.javasoft.com/products/jdbc/index.html, "The JDBC Data Access API" by Sun Microsystems, 1998.

[17] http://w3.one.net/~jhoffman/sqltut.htm, "Introduction to Structured Query Language" by Jim Hoffman, January 1998.

[18] http://www.innovation.ch/java/HTTPClient/index.html, "HTTPClient Version 0.3" by Ronald Tschalär, December 1998.