NEREUS Nemertes: Embedded Mass Spectrometer Control System

by

Adam Samuel Champy

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

Master Of Engineering in Computer Science and Electrical Engineering

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

May 2005 [June 2005]

Author .....
Department of Electrical Engineering and Computer Science
May 19, 2005

Certified by....
.....
Harold F. Hemond
William E. Leonhard Professor of Civil & Environmental Engineering
Thesis Supervisor

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

NEREUS Nemertes: Embedded Mass Spectrometer Control System

by

Adam Samuel Champy

Submitted to the Department of Electrical Engineering and Computer Science
on May 19, 2005, in partial fulfillment of the
requirements for the degree of
Master Of Engineering in Computer Science and Electrical Engineering

Abstract

In this thesis, I present Nemertes System, a software suite to control an embedded autonomous mass spectrometer. I first evaluate previous control systems for the hardware and evaluate a set of software design goals. The NSystem software builds upon the previous functionalities by offering text-file based scheduling and subroutines, as well as customizable scanning and data reporting. I also implement a new calibration technique that is suitable for auto-calibration while in autonomous operation. Overall, the system is designed to be modular and flexible with the expectation of hardware upgrades and changing needs.

Thesis Supervisor: Harold F. Hemond
Title: William E. Leonhard Professor of Civil & Environmental Engineering

Acknowledgments

I would first like to thank my mother and father, who have inspired me to dare to take on challenges and given me the work ethic to find success.

Second, I would like to thank Prof. Hemond for introducing me to environmental engineering and welcoming me onto a project with such great impact. Thank you for your mentorship and guidance.

Third, I would like to thank the NSF for making this research project possible and for enabling such important work for scientists across the land.

Contents

List of Figures

Chapter 1

Introduction

In Physics, Heisenberg argued that the more information you measure the position of an atom, the less you know about its momentum[4]. In Environmental Engineering, up to this point, the more you want to know about the chemical composition of a body of water, the *more* you know where graduate students will have to be: in the field, taking samples. With the vastness of our oceans, you can understand that there simply aren't enough researchers or resources to canvas the hydrosphere for data. Given that 37% of the world's population lives within 100 km of an ocean[3], and that many live along rivers and tributaries, mankind interacts with this poorly understood chemical transport system on a daily basis. We actively deposit pollutants, perturb sediments, and use water as a thermal sink. We feel the affects of our actions as well as those that occur simply due to nature, including temperature stratification, methane generation, and denitrification. Our vision is to develop a model for chemical transport in bodies of water, To achieve our vision, we're going to have to change the way we collect data. Discrete sampling isn't feasible and it isn't accurate. We need to build an *in-situ* sensor network capable of gathering the world's continuous data.

This thesis will take you through a design of a system that will gather continuous data about the hydrosphere to help us build a 3 dimensional model of chemical transport. Chapter One gives you a background of Upper Mystic Lake, our testing ground, and the peculiar nature of something called a pycnocline. Here, I outline the major components of our overall project, a sensor network, and then go into detail

about the Autonomous Underwater Vehicle that carries the Nereus Mass Spectrometer. My major contribution to this project is the Nereus control software that runs on the embedded computer in the Nereus sphere. I will plunge, so to speak, into that in Chapters Two and Three, where I evaluate the system needs and present my solution. Chapter Four focuses on test routines and results, while Chapter Five finishes the thesis with a look towards the future of the system and a statement of my accomplishments. This thesis is designed for those of you who might use NSystem to control Nereus or will modify it to suit your architecture. Enjoy.

1.1 Background

1.1.1 Upper Mystic Lake

Researchers have been studying Upper Mystic Lake for years, ever since the factories along Massachusetts' Mystic River churned out pollutants in the early 1900's as byproducts of activities like leather tanning. Specifically, Upper Mystic Lake contains an abnormal amount of arsenic in it's sediment. All of the suburban lawns and industrial waste lead to there also being a large amount of nutrients in the lake. These nutrients lead to enhanced algae growth, making the lake strongly eutrophic. The sediment, with dead algae and decomposing organisms, generates a great deal of methane. In terms of global warming and human health, we'd prefer the arsenic and methane to never reach the surface or enter the water supply. Upper Mystic Lake serves as a strong test bed for a sensor system as it has peculiar chemistry of its own and undergoes many changes in temperature and chemical concentrations throughout the year.

1.1.2 Thermal Stratification

Lakes undergo drastic changes due to numerous meteorological factors including temperature, wind, and air humidity fluctuations. In temperate areas, including the Northeast, most inland bodies of water track an annual cycle of warming and cool-

ing. Such a cycle leads to stratification of temperature zones within the water, i.e. a sharp vertical gradient in water temperature known as a pycnocline. Thus lakes and reservoirs in temperate zones often have a surface layer of warm *lessdense* water and a sharply distinct lower layer of more dense cold water. The vertical density stratification inhibits vertical chemical transport in the water. Thus, two distinct chemical zones evolve, especially in respect to dissolved oxygen and methane. Since both oxygen and methane control a great deal of biological and chemical functions in the ecosystem, their flux across the density gradient interests our group quite a bit. Since their flux is, again, by no means discrete in respect to time, we need to implement a more continuous sensing network to capture important flux events. Only then can we gain a full picture of how oxygen and the greenhouse gas methane transport through and out of lakes and reservoirs.

Wind and air humidity changes have some very interesting affects on these bodies of water as well. Low temperatures and low humidity cause cooling along the surface of lakes. Since this pool of water is much denser than the surrounding warm water, the cold water falls down to the lower depths in columnar fluxes. Also, wind causes similar effects through the creation of turbulence. These plunging thermals carry the chemical concentrations of the surface layer, meaning high oxygen levels, down to the lower methane rich levels. This oxygen could oxidize, via methanotrophs, the methane before it ever reaches the surface and escapes from the water. Up to this point, limnologists have found capturing this process quite difficult, as their intermittent nature does not match well with a discrete sampling process.

1.1.3 Mass Spectroscopy

The science of mass spectroscopy is the ability to separate chemicals and count chemical concentrations based on mass-to-charge ratios after ionization. The fundamentals of mass spectroscopy are well understood. *J.J. Thomson, 1913* [6] combined his earlier work on electron deflection with W. Wein's 1898 discovery that Goldenstein rays, positively charged ion beams, could be deflected by electric fields, to establish that different hydrocarbons and polyatomic molecules could be separated based on

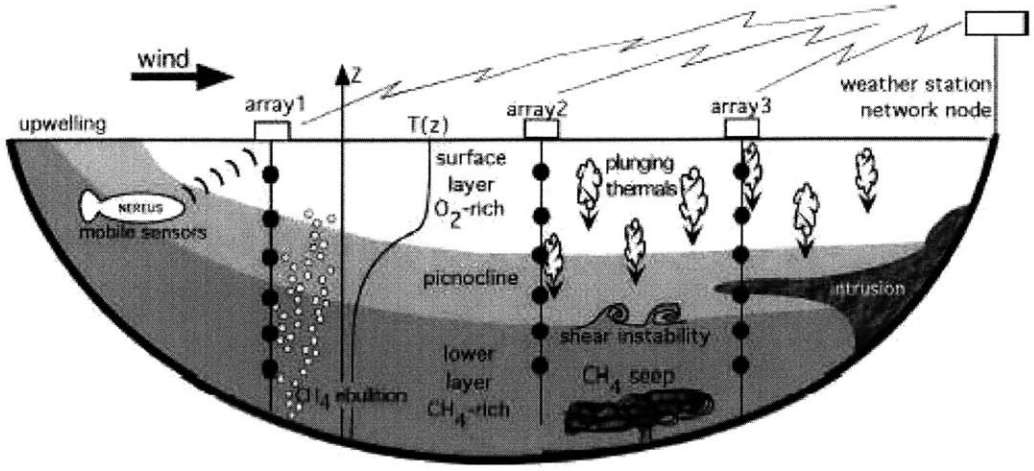

Figure 1-1: Sensor Network Architecture

their mass-to-charge ratio, $(\frac{m}{z})$. The result of his work was something called a mass spectrograph, a device that employed a photographic plate to record the presence of ion beams deflected at different angles. By replacing the photographic plate with a Faraday cup and an electrometer, you can actually measure the specific ion current for a specific mass-to-charge ratio. When you combine the relationship between ion concentration and the particular inlet membrane in use, you can generate very accurate data on the presence of even traces of many different chemical compounds. In the particular design of cycloid tube we are using, a modified CEC 21-620, all you have to do is change the potential on a set of plates to select of a particular $(\frac{m}{z})$. Its behavior is defined in *Bleakney and Hipple, 1938*[1]. This leads to a spectrum output, where each voltage applied to the system returns back a different ion current readout. Where $(\frac{m}{z})$'s are present, this spectrum exhibits a "peak", the size of which is related to the chemical concentration of a particular chemical. The relationship between voltages on the plates and the $(\frac{m}{z})$ differs for each mass spectrometer design.

1.2 Vision

Our vision is to create an *in-situ* chemical sensing network to capture the continuous and intermittent changes in the aqueous environment. Chemical sensing must occur in many different locations at the same time. A set of fixed sensors enables excellent vertical data capture when anchored on mooring lines. To capture horizontal data, Nereus employs a mobile AUV that can move throughout the sensor network to both capture data and calibrate the fixed sensors. In terms of cost, each fixed sensor must be relatively inexpensive if we plan to employ a lot of them. We can enable very high precision measurements by giving the AUV a suite of quite powerful sensors, including a mass spectrometer. Figure 1-1[1] displays a visual model of such a design. We hypothesize that the scale, cost, and performance limitations of each type of sensor can be overcome by networking small numbers of more sophisticated mobile sensors with extensive arrays of low-cost fixed sensors.

1.3 Implementation

1.3.1 Buoy Network

The foundation for our system is a constellation of 3 buoys deployed on Upper Mystic Lake. The buoy is a platform for a computer, communications, fixed sensors, and plays a large role in submarine navigation. Each buoy includes:

1. Technologix Systems TS-5000 Series 586 PC-104 Computer
2. Woods Hole Oceanographic Institute (WHOI) MicroModem Acoustic Modem
3. 802.11b Wireless With Hyperlink 1 Watt Power Amp and 6 dBi Omni Antenna
4. Thermistor String With 6 Thermistors At 2m Intervals And ADC Board
5. Xemics GPS Model RPSM002

[1]Courtesy Heidi Nepf, *Nereus NSF Proposal*, 2002

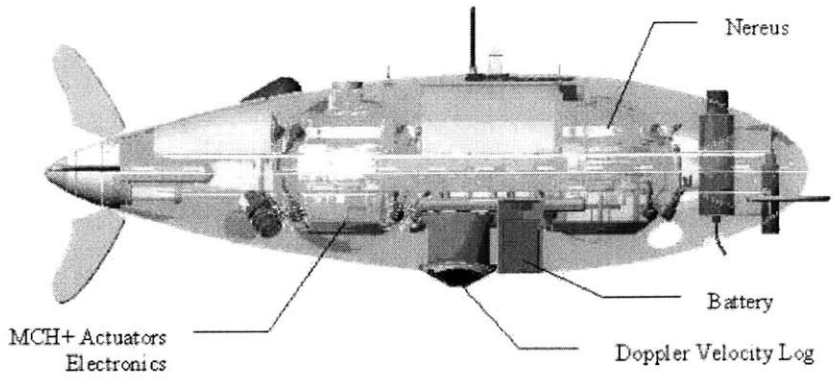

Figure 1-2: 2 Sphere Odyssey Vehicle With Mass Spectrometer Payload

6. Hydrolab MiniSonde With DO, pH, Turbidity, Conductivity Modules

Each buoy is capable of communicating with each other as well as back to the base station computer, graciously housed at the Medford Yacht Club, in order to post sensor data and execute remote commands. As the AUV also has a WHOI Micro-Modem, each buoy is capable of relaying information to and from the submarine. In these data transmissions, the modem is capable of calculating a range, enabling exact long baseline navigation with a network of at least 3 buoys. The network is not size limited much at all, as the wireless links to the buoys enable enormous bandwidth compared to transmitted data.

1.3.2 Autonomous Underwater Vehicle

To enable our sensor system to deliver a capable high-accuracy mass spectrometer to continuous locations in Upper Mystic Lake, we work in conjunction with MIT's SeaGrant AUVLab to redeploy an Odyssey IId to house a sphere-enclosed mass spectrometer. The current design for this craft includes such environmental sensors as a Hach Environmental Hydrolab MiniSonde 5 that includes dissolved oxygen, turbidity, pH, temperature, and conductivity probes. The craft is about 2 meters long, is powered by shore-rechargeable batteries, and is positively buoyant. The craft shell floods, so all components, including the payload, must be in water tight housing rated to the

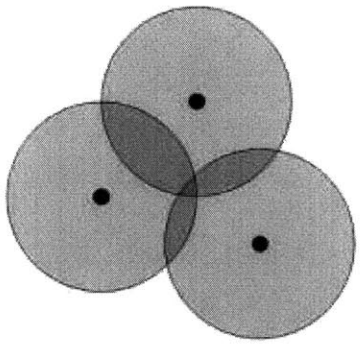

Figure 1-3: Long Baseline Navigation: Each Black Dot Represents A Buoy. The Center Darkest Area Represents The Best Guess Location Of The Submarines Based On An Intersection Of 3 Ranges. Note Too That This Is An Intersection In 3-Space So The Sub Position Is Fully Defined

depth of a mission. This Odyssey holds two 17" spheres, one housing the Nereus mass spectrometer, the other holding the computer system which controls the submarine. This PC-104 stack handles communication, submarine navigation, and gathers data from attached sensors over serial connections. The sub also communicates with the Nereus sphere via a tx-rx RS-232 serial line connection.

The submarine navigates using a long baseline navigation algorithm updated by measurements taken from acoustic pings in between WHOI micromodems. This requires a constellation of at least 3 known points within range of the submarine. Essentially, LBL intersects 3 range circles and finds the single common point, as shown in Figure 1-3. This has the capability to locate the submarine in 3 dimensions, when in sight of 3 buoys. The SeaGrant group has developed an implementation of an algorithm for updating a location guess for when the submarine may be out of range of one of the MicroModems or is simply dead reckoning. The temperature gradient causes much of the acoustic signal to either be reflected, absorbed, or corrupted when traveling through the thermocline. Therefore, routing the submarine with a constellation of 3 MicroModems requires thoughtful mission commands and possible different vertical positioning of the MicroModems on in the water in order to guarantee at least one capable communication and navigation link. The submarine is also capable of communicating over 802.11b when at the surface and navigating via its own on

board GPS - yet, for those of you unfamiliar with GPS, it does sometimes take some delay to get a locked position.

1.3.3 Nereus Mass Spectrometer

The critical payload aboard Kemonaut II is the Nereus mass spectrometer. The hardware behind this system was designed and specified in *R. Camilli, 2002* [2]. Camilli, a PhD student in the Hemond Lab, repackaged a "backpack" mass spectrometer based on the work of *Hemond, 1991* [5] published in the *Review of Scientific Instruments.* This repackaging enabled the system to be compatible with the Odyssey Class AUV, fit inside a 17 inch diameter Benthos sphere, and travel to depths of 100 meters or more (with minor modifications.) The system, as designed by Camilli, is capable of 240 samples per hour, weighs 22 kilograms, has no moving parts, and had a basic somewhat adaptable software package. Drawing 20 watts when running, and much less when off, this system suits the deployability requirements for week long missions aboard an AUV.

As I mentioned before, one controls $(\frac{m}{z})$ by tuning the voltages on a series of plates. In this design, the architecture has been designed to output a single voltage from a DAC conversion and have a Scan Board establish a set of 8 potentials based on that input. These potentials set the plates up in the cycloid tube of the mass spectrometer to scan for a particular $(\frac{m}{z})$. The relationship between the scan voltage and $(\frac{m}{z})$ for this design is

$$V_{scan} = \frac{slope}{\frac{m}{z}} + offset \tag{1.1}$$

Chapter 2

Software Design

The major contribution I have made in this thesis is the software that controls and guides the Nereus mass spectrometer in its scans, calibrations, reports, and interaction with other computers. With the understanding of the science from the last chapter, I now examine the previous work done on the mass spectrometer control software and analyze the user needs for a mass spectrometer control suite. The result of this chapter is a set of design principles, priorities, and usage patterns that serve as constraints for my engineering of the system.

2.1 Previous Systems

The previous systems that controlled Nereus were lightweight, easy to use, and required very little time to learn how to use them. Their major drawbacks are that they are difficult to modify, are somewhat monolithic processes, and offer the user only a limited amount of flexibility.

2.1.1 Backpack Mass Spectrometer

In *Hemond, 1991* and in unpublished data, Hemond designed a system to allow a human user to operate a "Backpack" based mass spectrometer. The compact and effective 629 lines of QBasic control code enabled the user to do the following tasks:

1. *Full Spectrum Scan*

   Scans the $(\frac{m}{z})$ ratios between two user specified inputs at a requested interval.

2. *Single Ion Monitor*

   Allows the user to continuously monitor a single $(\frac{m}{z})$ with the ability to change that value up or down using the keyboard of the control computer. This is particularly useful if trying to find a peak, as you can scan $(\frac{m}{z})$'s and do your own first derivative test.

3. *Calibrate*

   Requests information on two $(\frac{m}{z})$ peaks from the user. This requires the user to have looked at a scan of two spectrum peaks and to identify two relationships between a particular scan voltage and $(\frac{m}{z})$ ratio. The from these two peaks, the system calculates two variables, *slope* and $offset$ where the relationship between V_{scan} and $(\frac{m}{z})$ is exactly Equation 1.1.

   With two unknowns, *slope* and $offset$, the relationship between $(\frac{m}{z})$ and V_{scan} for two peaks fully specifies a calibration.

4. *Display Spectrum From Disk*

   This function acts somewhat like a GUI, enabling the user to visualize on a monitor a saved spectrum scan.

5. *Repeat Previous Scan*

   The system does remember its previous scan request for a full scan, so if you want to repeat the same range and interval, you can command the system to do this.

These commands are entered at a console-like interface, requiring constant user attention and command. Considering one may want to calibrate every few minutes, you cannot let the system just continue to scan without attention or you will suffer from drift.

This software, though, was simple and designed with a sizeable amount of functionality in a compact package. It required no programming knowledge by a system user, but it did require their constant attention, and not just in passively monitoring, but actively identifying calibration peaks. It can sit on one peak and scan it continuously, monitoring a particular chemical for example. It is limited, though, to either scanning a particular requested range or a single ($\frac{m}{z}$), but for what the system was designed to do, enable scanning, this software fits the bill.

2.1.2 Nereus Mass Spectrometer

With the *Camilli* design came the requirement that the system exist in embedded environments. These were mainly alongside a boat as a float or aboard an autonomous underwater vehicle. The link to the system would be a RS-232 serial communication TX-RX pair, allowing user supervision when attached to a computer. Aboard the AUV, which does not necessarily have a viable communications channel to the surface, the system had to be able to execute routines and scan accurately without human input. This was a much larger QBasic program, encompassing over 1100 lines of code. The changes from the Backpack included:

1. *Automatic Calibration*

   In an embedded environment, the system had to calibrate itself based on peaks it found during scans. In the aqueous environment the Nereus is designed for, 3 peaks are guaranteed to be present. Aqueous calibration peaks include

$$(\frac{m}{z})_{Water} = 16 \tag{2.1}$$

$$(\frac{m}{z})_{Nitrogen} = 20 \tag{2.2}$$

$$(\frac{m}{z})_{Argon} = 40 \tag{2.3}$$

   The nitrogen, in case you're curious, is dissolved N_2. *Camilli's* technique for finding a calibration differed from *Hemond* in that he attempted to minimize the error of the sum of predicted peak voltage location across all three peaks

simultaneously. His method was to calculate a set of 3 predicted voltages using something similar to Equation 1.1. His equation was

$$V_{scan} = (slope(\frac{m}{z}))^{warp} \tag{2.4}$$

He then used a global *slope* and *warp* variables and attempt to minimize the difference between the sum of the three predictions and the actual V_{scan} for each peak. To find this minimum, *Camilli* took the approach of starting by scanning a range of *warp*'s from 0.995 to 1.020 and *slope*'s from 45.00 to 47.50, attempting to find the first minimum of the error function. Upon discovery, the algorithm checked the error against a reference maximum allowable error, and upon success, returned the global calibration variables. To note, the $SmallestError$ was 0.01 and the system would try again up to 5 times before using the previous calibration.

2. *Routines*

   This feature gives you the ability to send a small serial string to the system and have the system go into a mode of executing a specific routine repetitively without any further user interaction. The Backpack design was capable of looping and remembering the previous scan, but this system establishes different scanning modes and continues to repetitively scan in that mode until ordered to stop.

3. *Peak Jumping*

   One of *Camilli's* routines is that of "peak jumping." After calibration, we are curious about the data returned from a full spectrum scan, but there are only a certain set of chemicals we're really interested in for every specific mission. His set was hard coded into an array and included 31 listings.

4. *Gradient Tracking*

   This feature the $(\frac{m}{z})$ with the largest change in between multiple scans and reports back to the user about that information.

5. *Noise Reduction: Signal Averaging*

   In development, *Camilli* documents several issues with noise in the readings coming off the electrometer. His technique to solve this problem involved signal averaging to try to capture the DC component of any noise. In routine commands, the user could specify 8 different averaging amounts, depending on the time they were willing to have the system sacrifice in every scan versus scanning more different locations.

6. *Thetis Graphical User Interface*

   As a complement to the embedded software, *Camilli* also presents a Thetis GUI interface to view the data generated by the embedded system. This software is designed in Visual Basic and interacts with the embedded system via a serial communication link.

2.2 Evaluation Of Previous Systems

2.2.1 Calibration

The *Hemond* algorithm fully specifies the calibration based on 2 peaks, but is limited in that it specifies the calibration for the entire scan range based on *only* two peaks. The *Camilli* algorithm looks at 3 peaks, and uses the *warp* variation instead of a simple exponent of -1 allowed for a better chance of finding some local minimum of the calibration error using his global search algorithm, as the $offset$ in Equation 1.1 changes along the $(\frac{m}{z})$. While this method produced a suitable approximation, using Equation 1.1 and 2 measured peaks completely specifies a calibration near those peaks. A better design for calibrating on 3 or more peaks would lead to $n-1$ different calibrations for n peaks depending on the requested $(\frac{m}{z})$.

2.2.2 Scan Routines

The Backpack design is much more basic but serves as a foundation for the codebase in the Nereus embedded system. Its routines are quite basic as discussed. The Nereus design builds in some hard coded routine options as well as some noise reductions using signal averaging. The system is only capable of simple gradient tracking, one peak-jumping array, and full scans at user specified intervals. The user cannot specify a departure from any of those routines without changing the code itself.

2.2.3 Design Flexibility

The previous systems are relatively inflexible in the face of the possibility that the mass spectrometer hardware, its interface, and logging requirements may change in the very near future. When the Backpack was originally envisioned, computing was a much different stage. DOS systems were the norm, wireless networking was barely on the horizon, and the design of even the chips on the Nereus hardware components differed greatly than what you can get today. Faster processors and larger amounts of memory have opened up more opportunities to do software-based analysis and signal processing, whereas previously much of that has always been done in analog circuitry. Current self-monitoring outputs, such as emission regulator current, may be converted into digital signals so the computer system could react to abnormalities.

The inflexibility lie in the fact that the code is one major module that shares dependencies on hardware, disk storage format, and I/O. If you ever want to change a single interface, you have to understand this entire monolithic block to know what might fail due to the change. Also, the code is limited by the DOS operating system, forcing a single thread and lacking much of the network support possibly required in the future.

2.3 User Analysis

2.3.1 User Skills

To be on the safe side, many of the users of the Nereus mass spectrometer will probably be more expert in either water chemistry or programming, probably not both. Therefore, the ability to scan for a particular chemical and receive information about its concentration should not be a programming chore nor should it return data that cannot immediately be used by a water chemist without serious post-processing.

The normal user should not have to be an expert in how to program in any particular language to use the system. They should know how to execute a program on a computer as well as use serial terminal emulation programs such as Hyperterminal or Minicom when the system is connected to a PC.

2.3.2 Desired Tasks

The user should be able to scan whatever ($\frac{m}{z}$) they want, whenever they want, and receive the data they care about and only the data they choose to care about along their I/O link to the sphere. There are usually only a specific group of possible scans one might care about, and it should be easy to look at specific chemicals or groups of chemicals. You should not have to change actual code to change a scan routine and if you ever increase the sensitivity of the machine, you should be able to add scan points easily.

There are a few basic tasks, but it should also be possible to schedule a routine of tasks and reports. For example, you want the system to scan for the Nitrogen, Carbon Dioxide, Argon, and Methane peaks 5 times, tell you about the Methane each time, calibrate, do a full scan, log the full scan on disk but don't tell you about it over the I/O link, and repeat. A good metaphor to apply is how easy shampoo manufacturers make it to "rinse, lather, repeat," and apply that to routine scheduling.

Calibration should be automatic and as exact as possible. The user will be required to interact with the system on the initial calibration, or at least give the system a

guess where to look for a the center of a peak, and the routine should be stable from there on out. If the system becomes un calibrated, it should be because of system failure and not due to the instability of a calibration routine. This would make it appropriate to notify the user or the host system that assistance is needed. The system should be able to notify the user or host system of emergency conditions and the specific assistance needed.

2.3.3 Upgrade Process Requirements

Only a power user would consider upgrading the system. By power user, I mean someone with an intimate understanding of the theory of operation (to get that, keep on reading the thesis.) The upgrader should not have to understand all the code that was written before, though. He or she should only need to understand how to interface to the previous functionalities that they need.

2.4 Modular Design

The major issues with the previous code were that we needed a better automatic calibration routines for the full spectrum, the ability to schedule a routine without knowledge of programming, and the ability to easily upgrade the system if necessary. At first, I considered designing the system along the same lines as the Backpack and Nereus Embedded. The major problem with that was that I would be directly interacting with functionalities that already work and could be encapsulated into their own foundational modules. The inspiration of control systems like MOOS and the projects I had worked on in object oriented software led me to a different approach.

2.4.1 An A/V Example

Let's use an example, if you want to add a CD player to a stereo system, you don't want to have to change the way the tape deck works too at the same time. All you want to know is what type of wire to plug into the amplifier, the interface, and leave

the tape deck be. Now this new component may be a CD-R recorder as well, so it should know how to receive an input from other A/V equipment. The tape deck records, so it knows how to receive those A/V inputs as well. The tape deck doesn't need to know when, if, or how the CD-R plugs in. The two work in very different ways, but share a similar interface. But, say you want to transfer tapes onto CD's, now, and only now, do you have to make sure your tape player and CD-R recorder know how to communicate with each other. When you start doing this, you recognize a dependency between the CD-R and the tape player's function. But, if you abstract out the functionality of reading a particular media, whether it be a tape, CD, record, or even 8-Track, and settle on a common interface, multiple A/V companies can design all sorts of components without calling each other up to see how to connect them together.

2.4.2 Modules In Software

So, one of the things I want to do is design in the capability to cache a set of recent ($\frac{m}{z}$) results so that the user can schedule a report of a specific group of them, I shouldn't have to change the way the system interacts with the mass spectrometer hardware to do that. If I want to design a new calibration routine, all I should have to do is know how to request the interface to the Nereus hardware scan at a specific voltage and return back a result.

Therefore, if I migrate the system to a modular design, adding functionality without affecting the stable base is simple. Each module contains abstractable functionalities. That means you don't have to know how the system works internally, that information is abstracted away by the well specified interface. As long as the module meets the interface specification, it can exist within the system framework.

The result is that I took the existing functionalities in the Backpack code and Nereus Embedded and ported them over to an object oriented language, C++, which could better support the module architecture that I'd like to implement. I added functionalities to the system beyond those in the previous systems by creating new components. There are many major advantages to this type of system, not just

making it easier to do my own work. For example, if a new user wants to create a component, the rest of this thesis will tell them exactly how simple that is and the interface specification that these components have to meet. Second, from a debugging standpoint, if ever there is a problem with a component in the system, you don't have to ever go into code that doesn't relate at all to the problem to find a solution. Since each functionality is abstracted, each potential internal issue is isolated. Think of what happens when your CD player breaks - you don't have to fix your tape player too. The trickiness comes in the interconnect between components, so I will spend a good deal of time detailing potential pitfalls that could occur.

Chapter 3

NSystem Implementation

In the previous chapters, I presented a scientific foundation for the project as a whole and the background for the engineering decisions I took. In this chapter, I present the result of these evaluations, along with a user and system analysis, is a modular component based design. Here, I present a set of 3 elements of the system, how they interact, and how they enable the scheduling and function of a mass spectrometer. In this chapter, I stay above the nitty gritty code level and talk about what the code actually does. The coding has all been done in C++, and there are Classes which represent NMessage, NComponent, and NControl. To know what's on the horizon, the next chapter details the components I built for this system, how they interact with each other, and some examples of how to use the system. All of the NComponents, as well as all of the code I wrote is in the "nemertes" folder in the Hemond group's web locker, available publicly online.[1] This locker also contains example code and turnkey instructions with expected outputs in case you somehow miss place this thesis.

3.1 Nereus Nemertes System: NSystem

I named the system Nemertes as she was one of the daughters of the Greek figure Nereus. While Thetis was their leader, and one of Camilli's projects, Nemertes was the wisest of the sisters.

[1] http://web.mit.edu/hemond/nemertes

3.1.1 Their Interface: NMessages

The most important aspect of a modular design is the interface between the modules. In this design, I wanted to try follow the notion of having the interface be a tangible item so that non-programmers could understand what's going on internally and even debug problems by looking at the inter-component communication. The result is that I decided to make the inter component communication a "Message." Each component is capable of receiving messages and a few are capable of generating them. For example, a log component is a passive element that watches actions and calculations done by the other components. The interface to the mass spectrometer is active as it must report back to the system the result of a scan at a requested voltage. There are multiple types or names of NMessages, each one referring to a different type of data communication within the system. The entire list of messages is enumerated in Appendix A. That list may serve as a valuable tool for the rest of the thesis, so if you can make a copy of it and keep it alongside, it will be easier to reference.

Specifically, each NMessage is capable of holding:

1. *MessageName* Name of that type of message - Required
2. *CreatorName* Name of the creator of the message - Optional
3. *DestinationName* Name of the destination of the message - Optional
4. *StringArguments* A variable length list of string arguments who's size is determined by the type of message
5. *DoubleArguments* A variable length list of double[2] arguments who's size is determined by the type of message
6. *Embedded NMessage* This is a message contained within a message, allowing for messages themselves to be passed if necessary - Optional

[2]A double is a C++ term for a double precision floating point variable, represented in memory with 8 bytes, and essentially a +/- 15 digit floating decimal point number.

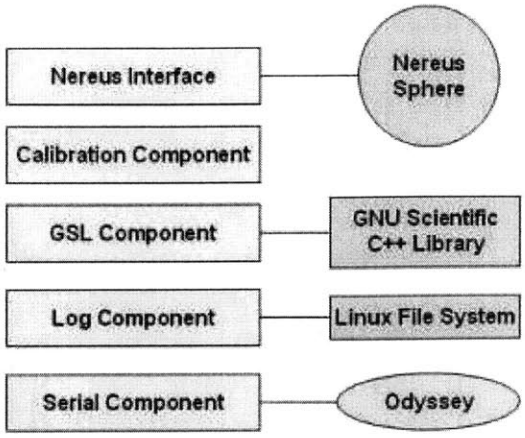

Figure 3-1: Dependencies of a few NComponents

If messages have a specific name for the type of data they carry, any component that receives a message can check what type of message has been sent and react to that message. Each component also has the ability to send messages. So, for example, there is a message that contains an $(\frac{m}{z})$ report, namely MZR. An MZR contains such information as the $(\frac{m}{z})$ requested, the scan time, and the V_{scan} actually requested to get that value. Multiple different components could be able to receive this message and do what they wish. One could log all the MZR's to disk. Another could cache them and upon request send out specific ones. Since components react to messages as well as create them, that request would also be in the form of a message, to be specific a MZC message.

One of the nice things about these messages is that a person can understand the data that is contained within. Our speed requirements and the scale of the software is not so great that the overhead of packaging messages outweighs the ability to debug and have knowledge of what's happening in the system when a problem occurs. The idea is to design to debug.

3.1.2 The Modules: NComponents

The idea is that each NComponent contains an abstractable set of functionality or dependencies. Figure 3-1 gives you a basic idea of the individual dependencies of each

module. Note how no two components share dependencies to the outside world, and that the CalibrationComponent, being a purely code-based component, does not rely upon any outside interface.

To keep the idea of components as simple as possible, There are only two specifications an NComponent has to meet.

1. The NComponent must be able to accept an NMessage. Upon accepting an NMessage, it must be able to respond with messages of its own if it would like to communicate back to the system.

2. The NComponent must be able to advertise the types of NMessages it would like to/can receive.

So far, I have decided that NComponents should not be able to randomly send messages out into the system without being messaged/polled themselves. This reduces the complexity and possible randomness that may occur if each component was timed differently, all outputting information to a shared "bus." That bus is the message distribution and collection system.

The cool thing about using messages is that the logging options are quite infinite as building passive components, components that do not send any messages, is trivial. Simply look at the DelayComponent as an example.

3.1.3 NControl: NMessage Distribution and Collection

So, we have all these messages, and a bunch of components. We need a distribution system between the messages to make sure that each component gets the messages it needs and can then send out proper responses. NControl handles the distribution of messages so that the certain components that only "care" about certain messages receive only those messages. If you would like to create an "omniscient" component, meaning it gets passed and therefore knows every message, that is a special allowance. As long as NControl has registered every NComponent and knows what messages each one wants to receive, data and commands are guaranteed to get to all the parts of the system that require them.

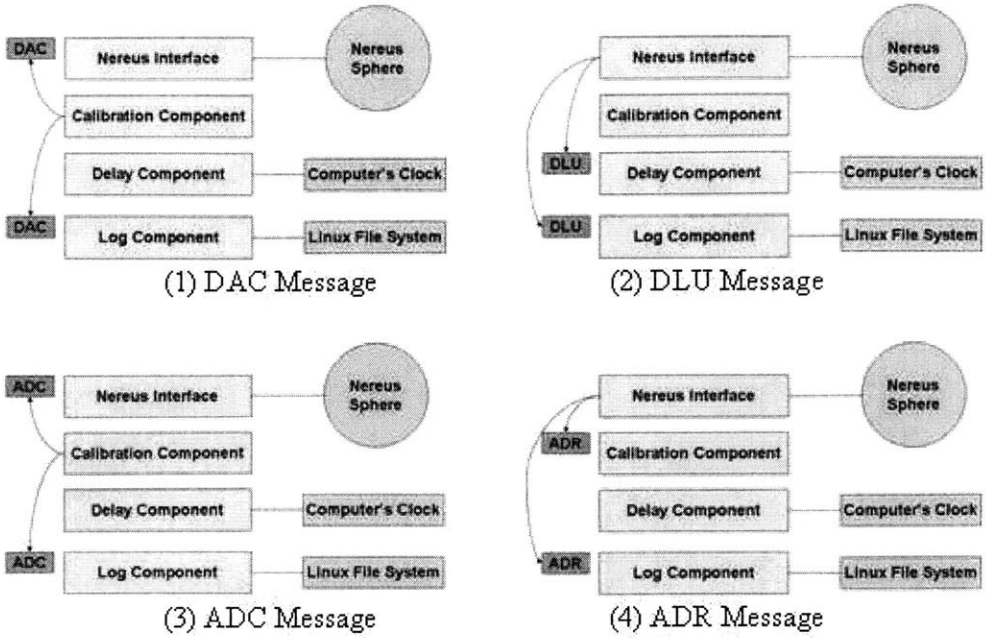

Figure 3-2: Message Passing Example: Note the "Omniscient" LogComponent receives every message

For example, the CalibrationComponent will need to request that the mass spectrometer scan a particular set of voltages to figure out the location of a peak. This is all illustrated in Figure 3-2. This means it sends at least one DAC message that the NereusComponent, and only the NereusComponent, will want to receive. This DAC message requests that the NereusComponent set a specific voltage on its DA converter. The Nereus responds with a message, DLU, which says how many microseconds the system should wait to be able to accurately scan at this new location, based on its knowledge of where the last scan was in relation to this new request. If there is a component that knows how to delay the system for a specific time, the DelayComponent, it will handle that message, but not send any new messages out. After the delay, the CalibrationComponent will want to make sure the NereusComponent does an analog-to-digital conversion of the return voltage from the electrometer. This takes the form of an ADC message request. This message is read in by the Nereus Interface, which responds using a report of that AD conversion, an ADR mes-

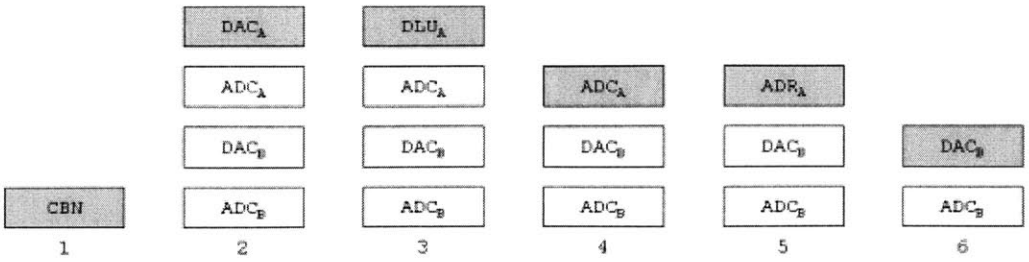

Figure 3-3: Stack Example Related To Figure 3-2

sage. The CalibrationComponent wants to hear about the ADR message and when it does it gets one of the data points necessary to perform a calibration. *Note,* the LogComponent is "omniscient", so it receives every message.

3.1.4 NComponent Interaction: The NMessage Stack

The next important architecture decision is to decide the organization of how NControl should distribute the messages. Whenever a component receives a message, it has the opportunity to respond with messages of its own. Three issues come up: multiple systems may respond to input messages, multiple components are often needed in a process, and the order of one sub-process should not affect the order of another unless intentionally. For example, the CalibrationComponent should be able to pretty much control the NereusComponent during calibration, without too much interruption and extraordinary efforts by the component designer.

To solve the problem of two components interacting with each other, I have decided to utilize a "Stack" system in order to give individual processes the ability to occur. What I mean by this is that the last message put on the stack is the first one distributed out to the components. Let's go back to the example used in Figure 3-2 and look at this example in terms of the NMessage Stack. Say we request a calibrate now from the CalibrationComponent, or CBN. For simplicity, let's assume that the CalibrationComponent is trying to figure out if the calibration peak for a specific $(\frac{m}{z})$ occurs at V_A or V_B. Figure 3-3 shows what happens on the stack. Each grayed

out message is the current message being passed. After the CalibrationComponent receives CBN, it sends out 4 messages which are pushed onto the stack. These are the DAC's and ADC's associated with scanning at 2 different voltages and returning the results. After the DAC_A is received by the NereusComponent, a DLU_A for delaying before scanning at V_A is returned. That is pushed onto the stack. Since it was the last one on, it is the first one to get "popped" and distributed. Since the DelayComponent doesn't respond to DLU's with any return messages, the next request that is processed is the ADC_A that the CalibrationComponent had returned T = 1 upon receiving the CBN. As you can see, the NereusComponent returns an ADR_A, which contains information relating $(\frac{m}{z}) = A$ to V_A.

As a component designer, having a stack design makes it possible for a process to know that if it sends a group of messages, those messages will be the first one distributed in the order that they were sent out. While this could cause some headaches, at least the NComponent designer knows exactly what to expect from each message being sent. He or she does not have to worry about unknown other actions being taken before their sent messages are delivered. This is specifically designed around the idea of calibration, during which the system should not also be scanning for particular $(\frac{m}{z})$'s. There is a slight danger in this design, in that if you design a "malicious" component, it could take over the system by continuously asking itself to do things.

3.1.5 NComponent Interaction: Order Of Distribution

Notice, though, that if another process also responds to DAC_A messages, that message would get pushed onto the stack in some relation to the DLU_A. Say another hypothetical component, the XComponent, responded to the DAC_A with an XXX_A message. The order of distribution depends on the order you register NComponents with the NControl - it's as simple as that. If XComponent was registered after the DelayComponent, you would see a T = 3 in Figure 3-4 as the last on was the message response from XComponent. The designer determines the order. You may start to notice a pattern that I am trying to reduce complexity and enforce deterministic system behavior. That effort is intentional.

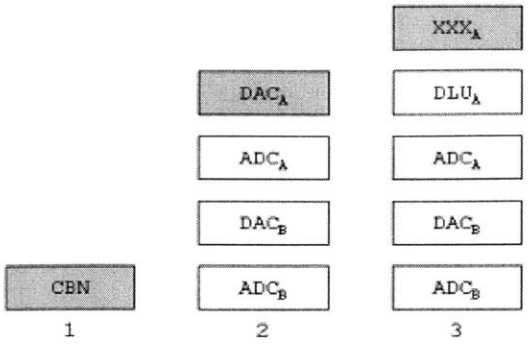

Figure 3-4: Stack Example With Component X Responding To DAC_A

3.1.6 Making or Inserting New NComponents

Making and inserting new NComponents has the following levels of difficulty:

1. *New Passive NComponent* A passive component pretty much just listens to messages sent to it. This does not mean that it is passive in function. A good example of such a component is an EREnable component. It listens to Emission Regulator Enable messages and if it receives them, turns on the emission regulator. If it receives and ER Off message, it turns that off. It is only passive because, as an NComponent, it does not send out any messages into the system. A good example template to use for a passive component is the DelayComponent.

2. *New Active NComponent* If you are creating a new component that actively responds to existing messages, you must make sure not to place it in the wrong order with other components that react to the same existing messages. Otherwise, if it reacts only to a new message type that you are defining, just make sure this new NComponent meets the specifications of any of the existing types of messages it receives/sends. The most basic active component to use as an example if you are building an NComponent is actually the GSLComponent. It simply takes in 2 messages, passes that information to a backend, and then

returns a single message.

3. *Modify Active NComponent* All you have to do is make sure that the specification for the messages that come in and out of the system looks the same. A good example of a modifiable component is the Peak Max Component which does the peak maximum calculation for peak-finding. You could replace the internal algorithm and not touch any of the external interface.

3.1.7 NSystem As A Platform

From what you've read, you might ask how is this control system specific to mass spectroscopy. The answer is that the system is customized only by the components one decides to use. If you would like to have an interface to a thermistor/temperature gauge added to the system, and have that component be incorporated into your calibration routine by sending a message to the calibration system upon request, it's relatively simple. All you have to do is build a component that accepts requests for information in the form of an NMessage and is capable of returning the desired information. The actual interface to the temperature gauge could be a serial port, digital I/O, or something even more exotic. All that is abstracted away from the rest of the system using the component idea.

3.2 NMessages Enable Scheduling and Commands

There are a few questions that must have arisen by now, including, "if the components only respond to messages, how does the whole process start?" Or, "if the NMessage stack is empty, what happens then?" One of the most important features of the idea of messages being commands is that you can create a schedule of messages to be sent to the system whenever the local stack is empty. In fact, if you build a component that holds a list of messages and how many times that they should be sent out, you have yourself a scheduling system. It is actually the NMessages serving as the interface amongst components that enable the user of the system to schedule exactly

Instruction	# Times
Calibrate	1
Scan MZ's [16,18,28,32,40]	∞
Report on CH_4 and O_2	∞
Scan Integer MZ's From 10 to 45	∞
Report on CH_4 and O_2	∞

Figure 3-5: Schedule Of Desired Functions

what they want to happen. In this section, I also show that the user can designate routines without ever going into the code of the actual executable. In fact, all you have to know is the specification of the command messages in order to change the scanning and reporting pattern.

3.2.1 ScheduleComponent And The GNE Message

Take a look at Figure 3-5. This is a short example of what the user might want the system to do. When it starts up, the system should calibrate itself one time and then continue doing the remaining 4 items until turned off. The system should automatically check its calibration during runs, so as a schedule, all the user should have to tell the system to do is the types of scans a calibrated system should do. Now, imagine a component that stores a list of messages, ready to dispense them somewhat like a PEZ-dispenser. Simply send this component a message, Get Next Event (GNE,) and receive the next message. This component keeps track of an order that the user has specified and how many times each item in that order should be "dispensed" before not dispensing that item.

I have implemented this listing component as the ScheduleComponent. This is a special component that is always registered with the NSystem as the NSystem itself can send it a message whenever the message stack is empty. The ScheduleComponent responds to the GNE by sending any message that the user has loaded into the

schedule.

3.2.2 ScheduleFileLoader And The LSF Message

To load a message into the ScheduleComponent, the system can either read schedule files from disk or get them from the serial terminal. The NComponent that knows how to parse my specification for a schedule file is ScheduleFileLoader. This takes files in text form, parses them to create a schedule, and sends these items to the ScheduleComponent to be listed in the schedule. Each line in a schedule file represents a message to be sent to the system and how many times the message should be sent. A specification for schedule files appears in the appendix

To load a schedule file, all you have to do is send a LSF, or Load Schedule File message to the component. Schedule files themselves can contain "sub-schedules", or LSF messages to load other schedule files. For example, you could create a routine that scans one set of $(\frac{m}{z})$'s related to each other another to look at the shape of the peaks themselves. These can be kept separate and called somewhat like subroutines or macros.

When the user starts the system up and run the executable on the embedded computer, he or she supplies an initial schedule file. This is automatically parsed by the ScheduleFileLoader and the events are scheduled in order by the ScheduleComponent. If the user requests an LSF in the initial schedule file, the events inside this new schedule file will be inserted into the schedule.

Note, and this is important, do not make an LSF an infinitely run event and any events in the file loaded infinite as well. If you do, the LSF will keep on adding an event that never gets exhausted from the ScheduleComponent's schedule and the system will eventually creep towards only performing those events that are infinitely added. For examples of schedule files, please see Appendix B.

3.2.3 Serial Control: LSR Messages

The SerialComponent is polled with a Load Serial message before every GNE message is sent to the ScheduleComponent. This sees if the system should change its behavior by following a command sent over the serial line. The user can send a message into the system via this I/O Channel, such as a LSF message to perform a particular routine, or simply a 1 message command. A pair of instructions could have the ScheduleComponent erase its current schedule and then load a new LSF, changing the behavior of the system entirely to reflect the new schedule file. I will describe this in the appendix. One has a great deal of control over the system with the serial line as currently designed. That does put a burden on the operator not to send malicious messages into the system, but that would mean the operator also has physical access to the sphere.

3.3 Design Decisions And Considerations

There were a few alternate designs that I did consider, including a few that would go along the same lines as the previously written code.

3.3.1 The Same Interface: NMessages

You may be wondering why I made the interface between all the components using the same data type - NMessages. I did this to make it easy to add new functionalities that we might not be even able to envision. The other major benefit is that it is *very* simple to design passive components to log exactly what you want to log because everything speaks the same language. If I made the interface to the Nereus different than the interface to the calibration system, the logging scheme would have to be dependent on exactly those specifications or both the calibration system and the Nereus interface would have to be dependent on the single logging scheme. If a developer ever wanted to extend the system, their logging scheme would have to either become cognoscenti of the new way to interact with the new functionality or the old systems would have

to become aware of how to send data to the new system.

There is also a strength in having a set of similar code examples for new developers to use. Judging by the usual years of programming experience amongst new graduate students, having to learn one specification will get the designers off the ground much faster than learning multiple systems.

3.3.2 The Same Interface: Viewing Messages

Using the MessageViewComponent, you can set the message that the system outputs to the serial console. These settings are dynamic. A curious user can change the settings to see how a specific process is being executed. By settings, I mean the user can select the NMessages which are broadcast to the user as well as the system. The MessageViewComponent is one of those "Omniscient" systems, and could output every message if you told it to with the keyword ALL, but that is probably very rarely useful, so by default it's silent. A GUI interface on the other end of a serial line can request information and turn on the toggle for the types of message output it wants to see for visual representation.

3.3.3 Speed and Size

By using "messages," I do slow down the communication between different functionalities. Size is not increased dramatically unless the messages a component generates uses very long lists of strings. Please avoid using lots of strings in any message type you design. In terms of numbers, at some point you're going to have to pass data between different functions anyways, so using the messages to encapsulate that data shouldn't take up too much more space over the lifetime of the system (which is longer than the lifetime of the messages.) Some messages do stay in the system for a great deal of time, and those are the ones in the schedule with unexhausted run-cycles. That information would be kept in any scheduling system, so the software is still relatively lightweight.

The message stack should never get that big if you design components to not spit

out infinite amounts of messages. Again, as I reminded in the section on schedule files, do not schedule any infinite LSF's with non-limited events in the schedule file to which they refer. This will cause major size issues as the schedule will grow at an infinitely/unbounded rate. I may design a way to check for that, so please refer to the code and just avoid doing it if you can.

3.3.4 MOOS

MOOS is the Mission Oriented Operating System employed by SeaGrant's AUV lab to run many of the processes aboard Odyssey class submarines and other computer controlled devices. It was initially designed by Paul Newman for underwater vehicles. My colleagues in the Hemond Lab also use it to run our buoy system as described in Chapter 1. It is essentially a multithreaded lightweight OS designed to host multiple sensor processes. In many ways its better features - in terms of modularity and upgradeability - inspired what I did. MOOS is specifically designed to control a number of asynchronous and synchronous sensors, gather and log that data, and have the capability of transmitting using built in network support. I decided to avoid using MOOS for a few reasons. First, it has a steep learning curve for beginning programmers, and the major support for the system is an AUV Lab research engineer. This is a dangerous route for support if the Nereus hardware is not specifically always going to be used with SeaGrant support. The independent documentation is relatively poor and experience working with MOOS' experts is the best way to learn the system. It is, though, an excellent system design. I did not desire new researchers on the project who wanted to develop for the Nereus sphere to *have* to learn MOOS. While they have to learn a little bit about the system I developed, the NSystem's basic functionality is at a more tangible lower level with less overhead. It is therefore less powerful, but that is intentional. Second, I felt that the Nereus sphere itself, being essentially 1 sensor, did not need the complexities associated with a multithreaded data gathering system designed for multiple instruments. I wanted the system to be as transparent as possible to the novice user with easy system scheduling and debugging. The last major reason I designed my own system is to be able to have

simple access to schedule any particular functionality. The tangible idea of NMessages and ScheduleComponent enable this power.

3.3.5 Threading

I decided not to use a multithreaded main system due to the fact that I wanted to reduce complexity and enforce deterministic behavior in this first version of the code. I have done nothing to prevent a new designer from adding threaded systems in, in fact the SerialInterface accepts asynchronous communication in a hardware based cache and polls that for new incoming data whenever asked. An NComponent could create its own background thread for processing, especially if trying to do Digital Signal Processing which could take up a lot of main system time. Also if asynchronous threaded components wanted to be designed, the only addition a developer might make is to have a special class of asynchronous components that are polled for messages even if they do not receive a message. It's not a difficult modification to make and I have commented the lines of code in NSystem where you would most likely make that change.

Chapter 4

System Tests, News, And Results

In this section I take you through our first scans, some data, and some of the procedures and commands I used to generate that data. For specifics on the commands used and example commands, please see the appendix. For some interesting spectral data, keep on reading here.

4.1 Bench Testing

4.1.1 The Simulator

During April and May 2005, I was able to start testing my system on the lab bench. The first major set of tests I did was to make sure I could control V_{scan} aptly and record voltages using the analog-to-digital conversion module. This pretty much was an isolation test of the "DAQ" data acquisition board that Camilli had designed and built. The board is designed to take two byte (8 bit) values off the parallel port in series and multiplex them on the input of an 16 bit Analog Devices AD569 DAC. The simulator itself is a voltage measuring multi-meter connected to a DOS based PC over a GPIB interface. The DOS computer runs a QBasic program which reads in the voltage and reports the M/Z it believes that voltage to be referring to onscreen while simultaneously outputting a simulated electrometer voltage signal. This signal is interpreted by a digital-to-analog module and fed into the analog input of the DAQ

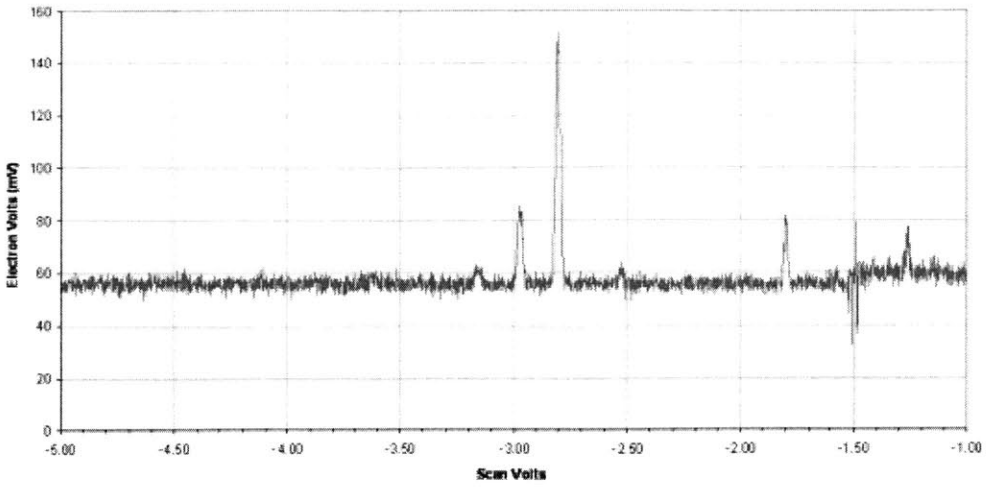

Figure 4-1: Result of Scan Voltage Range Command, SVR, Between -1 And -5 Volts. The Tall Peak Around -2.81 Is Mainly Water Vapor. The Peak At -1.26 Is Argon

Board. My system is then able to read in this return voltage by commanding an Analog Devices AD7884 ADC to multiplex 16 bits of two's-complement output onto the parallel bus. To command the system to set a scan voltage and read a voltage back, please reference the DAC and ADC requests in the appendix.

4.1.2 The First Spectrum

On May 13^{th}, 2005, we started the mass spectrometer hardware for the first time in over 2 years and 2 major laboratory moves. Some issues that arose included flaky edge connectors to the scan board, degassing effects, and a leak was confirmed in the vacuum envelope. The leak itself is confirmed by the scan we took as shown in Figure 4-1. The peaks, especially those at voltages near those we'd expect for water vapor, CO_2, and Argon gas, are prime components of the atmosphere.

At first, our output was very noisy and the DAQ board did not seem to be acquiring the signal properly. Our response was to do as all engineers should do, break out the chart recorder and oscilloscope and use an analog method to see if the scanning hardware itself was working properly. Having never used a chart recorder, let me tell you, it is a work of genius if you're trying to prove whether a digitally acquired signal

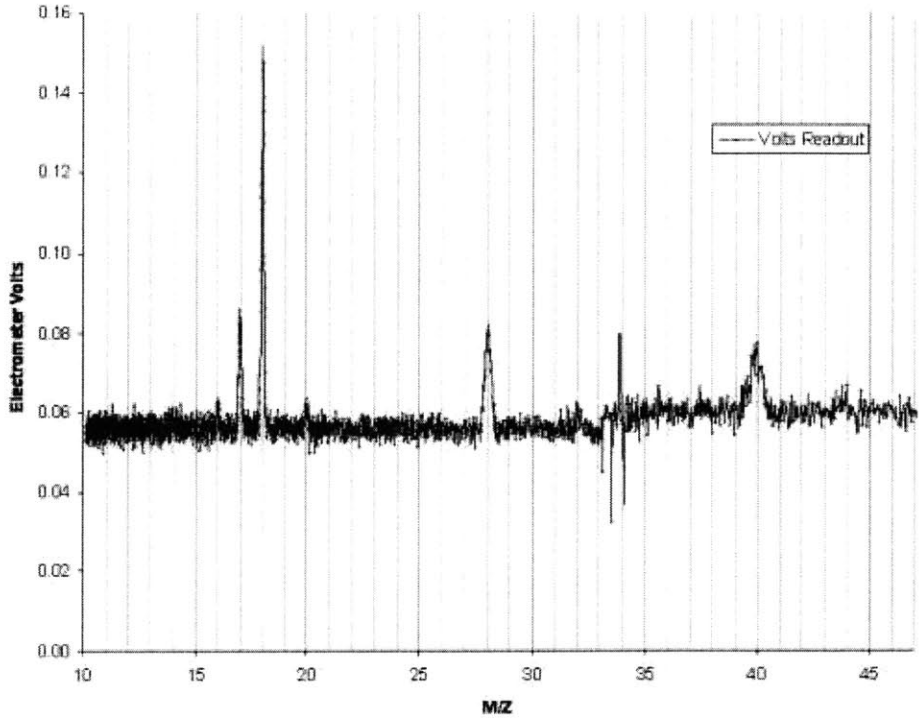

Figure 4-2: Result of Calibrating On The Water And Carbon Dioxide Peaks In Microsoft Excel

is what is actually the same as what an electrometer is putting out.

4.1.3 Mass Scan Examples

When the system got running, I was able to calibrate on a scan and produce Figure 4-2. I calibrated this scan using Microsoft Excel to make sure I was actually getting legitimate data. Specifically, I calibrated the entire scan on 2 peaks to see how well the outermost peaks would line up. I chose the largest, $(\frac{m}{z}) = 18$ and $(\frac{m}{z}) = 28$. When I felt comfortable (and saw that there was in fact a peak lining up at 40, Argon,) I requested the system scan all masses between 15 and 45, using the Scan Mass Range, SMR, command. The system remembers the data for these peaks until queried or scheduled to report the peaks you want using the Mass List Request, or MLR. If I request peaks 15, 16, 18, 28, 32, 40, and 44 - referring to H_2O, CO_2, 0_2, and Argon

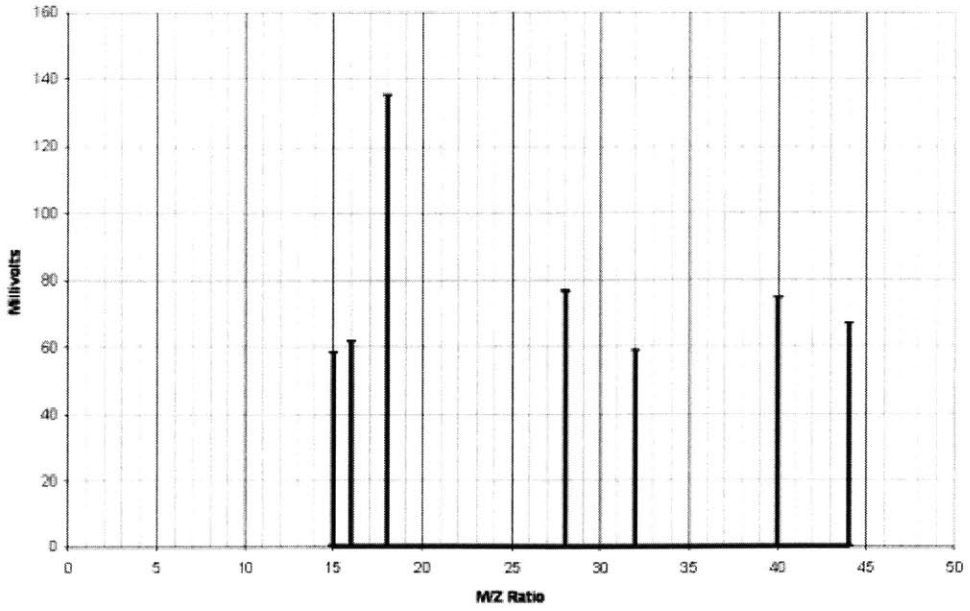

Figure 4-3: Result Of A Mass List Request, MLR, And System Autocalibration On Water And Argon Peaks

found in air vapor, we get the data displayed in Figure 4-3. This is a nice feature because while the system, if you want, could tell you about every mass it scans, can wait until you schedule or request specific information be sent to you (such as after the completion of a full scan.) In this way, you can easily track a set of compounds, either with a GUI, an interface to a host system (aka the Odyssey,) or simply on the command line of a terminal emulator. You are never bombarded with data you don't want to see.

For the scan in Figure 4-3, the system calibrated itself given initial guesses for the Argon peak and the H_2O peak taken from 4-1. The calibration was done by scheduling the add at least two calibration peaks, CBA command, then a calibrate now command, CBN, to have the system calibrate both of those peaks. The system *requires* at least 2 calibration peaks in order to perform any $(\frac{m}{z})$ related scans. You do not have to specify any calibration information if you are doing voltage only scans.

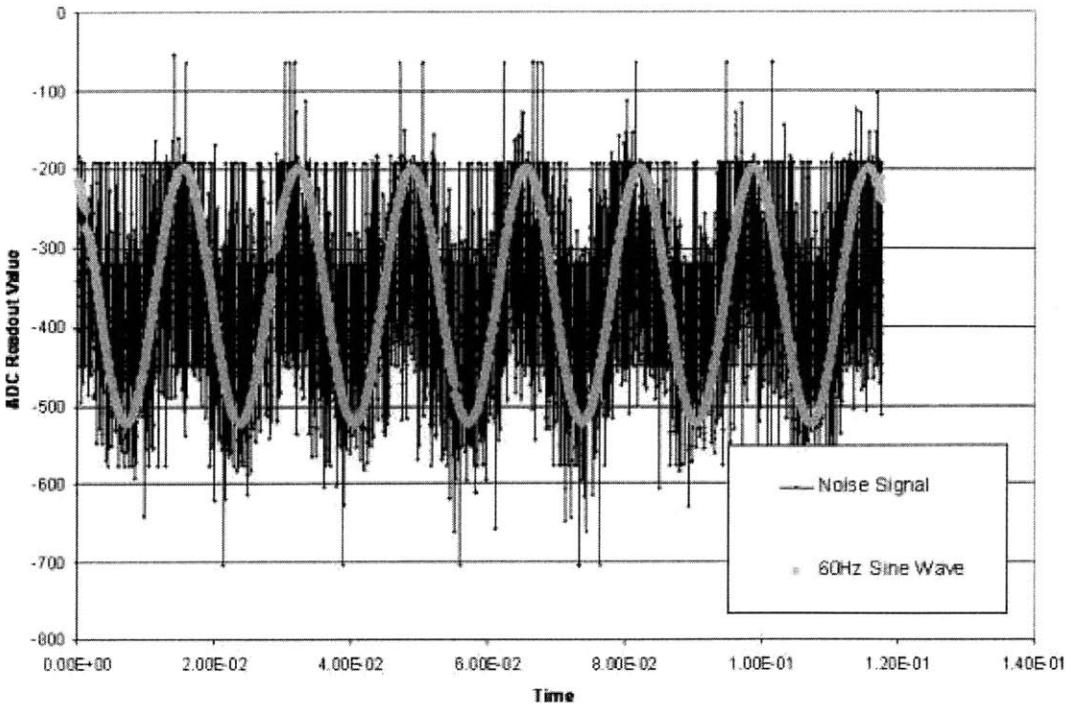

Figure 4-4: Electrometer Noise - Signal Versus Time In Seconds

4.1.4 Noise Profile

The major things you can notice are the baseline that exists in the scan and the amount of noise in the scan. I decided to try to sample the noise as fast possible to both see the DC component and possibly profile the noise in the frequency domain. Since we were doing bench testing, surrounded by labs, and plugged into a wall, Figure 4-4 displays a signal that is heavily affected by a near 60 Hz noise pattern. I have even plotted a 60 Hz signal to show you instead of simply displaying the frequency spectrum. The spectrum, as Camilli noted in his thesis, has constant low level noise throughout. The only major component that spiked was this 60Hz noise, but I do not believe it to be as much of a problem when out in the field.

4.1.5 Signal Averaging Versus Signal Processing

Due to the fact that there is this 60 Hz signal noise, one thing I had to consider is whether we wanted to do any digital signal processing on the incoming signal.

Having looked at the profile of the noise and done some averaging techniques, if we are simply trying to get rid of 60 Hz noise, then signal averaging works pretty well in comparison to running an FFT on a set of samples and selecting out the DC component, especially with the 66 MHz 486 processor we have aboard the Mass Spectrometer. Matrix operations are deceptively much faster on the 1.4 GHz Mobile Pentium 4 on which I also run test routines. I haven't built this out of the system though - if you wanted to process the information coming from the NereusComponent, simply change the message it returns after an Analog To Digital Conversion message, ADC, to something that contains a listing of all the samples it took and possibly time step information. Then, build another NComponent that reads in this message and releases the Analog to Digital Report as the current specification for the ADR message is - you won't have to change any components that rely on the ADR if you still meet its specification and make sure that message is sent back as a result of an ADC.

4.2 Calibration Evaluation

My particular method for calibrating was to scan around a previously known or user guessed peak and attempt to best fit a parabola around it using a least squares method. Upon completion of the linear algebra, which the system does using the GNU Scientific Library, the calibrated peak is set to the center of the best fit curve. While the peaks themselves are not actually parabolic in shape, but are actually flat-topped, this is a good first modeling to fitting some form of statistical shape around the curve to pick out the peak. You could easily substitute for the best-fit algorithm, which lies in GSLComponent, with another algorithm that takes in the scan data around the peak contained in the messages PM1 and PM2, and output your own Peak Max Report, PMR. Here is a set of results for what I did, and some techniques for getting even better performance from the approach I took.

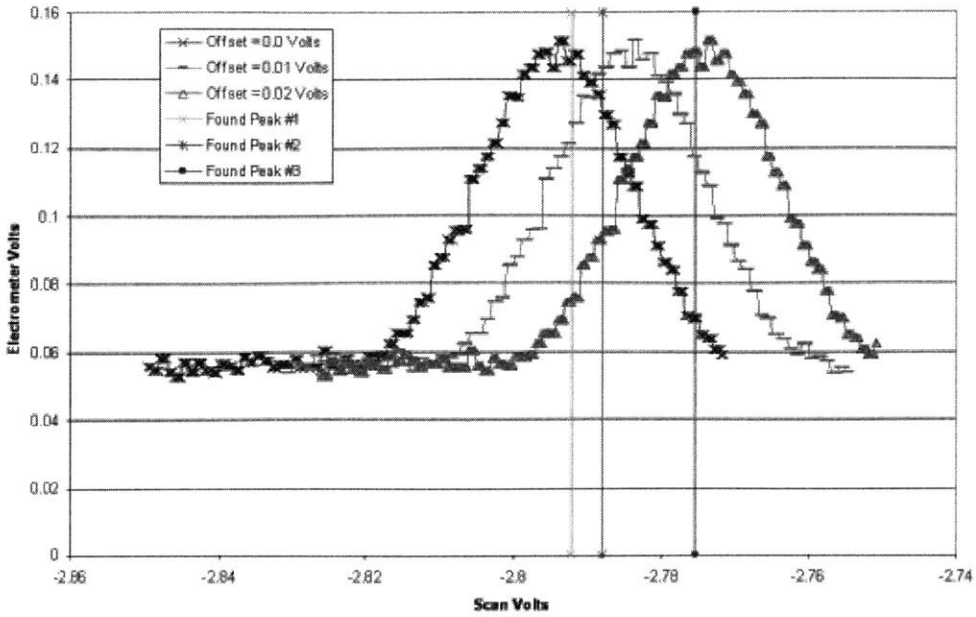

Figure 4-5: Calibrating Against Shifting Peaks - After Every Calibration, The Peak Is Shifted By 0.01 Volts - Note The "Tail Effect"

4.2.1 Calibration Performance

Overall, I was impressed by the calibration ability of the system. Its calibration stayed stable when the peak did not move, and was able to track peak movements in as little as one calibration. As a test, I forced the peak to move by 0.01 Volts around the $V_{18} \approx 2.81$ peak. I only let the system calibrate 1 time on each subsequent moved peak and then shifted again, somewhat of a worst case scenario for continuous shifting. Just as a reference, a change of $\Delta V_{scan} = 0.01$ around 2.18 means a $\Delta(\frac{m}{z}) = .05$. What was interesting is that the calibration actually caught up after getting what I would consider behind. This is all shown in 4-5. I ran these test routines on a set of cached static data, so if the results look quite discrete, they are. I did build a simulator component, called FakeMassSpec, that can fill in for the mass spectrometer itself if I need to try my procedures out while the hardware itself is unavailable.

The major detractor from calibrating was the "tail effect" that exists if you scan too large of a range in a calibration routine. There are a couple of methods to

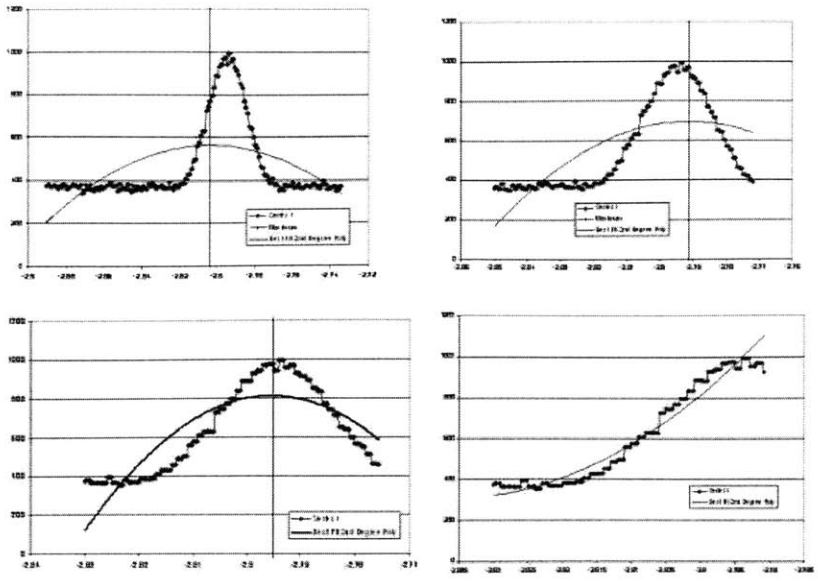

Figure 4-6: Graphical Example Of A Zoom In Calibration Routine And The "Zoom Too Far" Effect

eliminate the "tail effect". First, have a higher baseline on samples taken in the area where you expect the peak to be and only fit the curve to these samples, thus eliminating any weight the "tail" has on the parabolic fit. You have in fact eliminated the "tail" itself. Second, if you must scan a wide range, I have come up with a "zoom in technique."

4.2.2 Improved Calibration Routines

The calibration routine is stable and tracking performance is decent. The system would stabilize on a peak when I ran multiple calibrations in one place and the seed calibration was in the nearby ballpark of the original peak. Getting curious, and also looking at how wide the best fit curves were for some of my samplings, I wondered about a more effective calibration routine. Specifically, I thought about using the first attempt as a wide area attempt, to zone in on the basic area of the peak. I noticed that the flat area of any scan, the baseline, affected the least-squares quite a bit. Knowing that area, you can zoom in, so to speak, and the best fit curve becomes

more along the exact shape. I put together Figure 4-6 in Excel to get a graphical idea of whether this might would work. Each scan reduces the are sampled by one half. In fact, in this example, the number of samples taken per $(\frac{m}{z})$ was not increased in each scan, so the samples in each scan decreased by 1/2 as well. Since I allow you to set the calibration width and resolution using messages, you can write a schedule for a zoom in calibrate without ever going into the code itself. This seems to be much more effective, especially when the peaks could move around so much that they could get lost in between scheduled calibrations. These images don't even change the center of the scan, which would update and improve on each scan. An example of what a zoom in calibration schedule could look like is in the Appendix. You just have to make sure not to zoom in too much or you'll lose a peak and get a curve like the lower right element in Figure 4-6. This effect becomes visible if you don't get a proper recentering, something I wanted to show. Another interesting approach if the peak is lost might be to find the intersection of two parabolas fit to the outsides of a peak. If you make sure not to actually pick up the peak, the intersection may be over the area of the peak, allowing you to regain stability in the calibration if it gets lost.

If we used a zoom in technique, the system may still calibrate faster and more accurately than Camilli's approach. His system attempted to find a calibration 5 times and then started over. This system, in all trials, seems to be able to calibrate in the ballpark of the peak in one shot, and with a zoom in, may get progressively better in what could be logarithmically smaller samples. This would improve upon calibration time so much that you could conceive of running short and accurate calibrations more often instead of having to worry about sacrificing a great deal of time with a consuming routine.

Chapter 5

Accomplishments

5.1 Contributions

First, with the ScheduleComponent and Schedule Files, I have delivered user controlled scheduling of all system routines without any C++ coding knowledge. What I have given is somewhat like a specific language to very flexible scanning and reporting. All the user needs to do is create a text file with the NMessages they want sent to the system, the order in which the messages should be sent, and how many times they should be sent in an event loop. This also enables much simpler embedded routine creation, as the user does not ever have to work with the executable once it is loaded aboard the embedded flash. The user can simply upload over serial file transfer text based schedule files. The system does not have much I/O bandwidth, so this feature saves a lot of hassles and opening the sphere to change mission plans.

Second, I have enabled the system to report data in a scheduled and compact fashion, tunable to the user's desires. Specifically, a user can request items like full scans and then have the system report back only selected/priority masses. It also saves scan time because we only scan for what we need. This information is valuable both to the user at the terminal emulation screen and the interface designer who only wants specific information for their information processing needs. It reduces the complexity of the interface to exactly what is needed and nothing more. Hence, if a robot wants to scan for a certain set of chemicals, the system can be specified to only

output that information, and nothing else, if the interface wants that.

Third, I have demonstrated the scanning control capabilities of the system and its calibration routine performance. I have presented examples of the system scanning in Spring 2005, and delivered a calibration routine which, in its current basic form, works decently for small peak shifts. For larger peak shifts, I have suggested a routine, easily implementable in scheduling and not code, that could help with high drift situations.

Fourth, I have given the developer an easy way of adding functionality if they so choose to do it, and to modify existing pathways. As long as the current parts see the same interface to the current parts, whether that be getting an ADR for a sent ADC message or a CBR report coming back from a CBN calibration request, nothing should break or need changes upon addition.

5.2 The Future

5.2.1 Unfinished Business and New Ideas

If you've made it this far, there's a good chance you're probably taking this project even further. Some specific things I plan on doing are designing a Java based Swing GUI to control the system over a serial interface. Given that I've designed with a GUI in mind as a future project, the system can be put in a great deal of hosts and put out a plethora of information about how it's working. Because of that, I'd suggest improving the I/O link into the sphere by switching out the embedded PC for one with wireless capability. Second, I'd probably go for a processor with greater speed in order to enable full digital signal processing without any concern about clock cycles taken up. Considering a few cycles on chips running in the megahertz range is about the same time as a sample intake from the DAQ board, brute force wins out now if it means less processing burden.

I would like to put together some full mission length template schedule files, and subroutines for proper shutdown of the system. These will come when mission objectives are defined as soon as a week after this document's publish date. The other

addition to templates which would be cool is to make a Component that can take the argument of one message and make it an argument of an outgoing message where both are parts of a subroutine pair. Currently that is not implemented, but could be powerful if you would like to create high level routines based on the scheduling language as a core. It would also be cool to design in the ability for schedules to have inner loops, "for i" constructions, and "while" executions to make schedules not just lists but miniature programs.

5.2.2 Best Wishes

To finish up, I'd like to wish everyone good luck who's using the Nereus Sphere and NSystem. The oceans are a wonderful body of knowledge that we have not even come close to understanding. Through this project I've had the chance to learn a great deal about water sciences and the scale of some of the problems we're working on in environmental engineering. I'd highly recommend to anyone to work on an applied engineering project versus purely theoretical. Murphy's Law strikes more often but seeing tangible results is a great feeling. I've enjoyed working on this project quite a bit and wish Nereus safe travels on the sea.

Appendix A

NMessages

These appendices are mainly useful for how you might use the system yourself and to demonstrate the functionalities that I have built. This appendix gives you a good reference for the different types of NMessages available to use to command the system and those that are internally communicated and should only be used when testing the response of the system.

A.1 Message Specifications

In the following three Figures, Figures A.1 through A.1 you can see the specifications for the NMessages the system understands as of May 2005. These charts map out the commands you can use, either in designing your own NComponents, or in commanding the system to do certain routines. The specifications for what you can consider doing are pretty wide open using this instruction set. Please use this section as a reference in understanding what the arguments are for each message and a command set. I have ordered them in the probable order you will need them, Calibration, Scanning, and then miscellaneous utility functions.

Calibration

ExternalName	Full Name	String Args	Double Args	Current Target Components	Internally Generating Components	Embedded Message
CBA	Add Calibration Peak	None	Peak M/Z, Guess At Voltage	CalibrationComponent	None	None
	Adds a calibration peak to the routine - this MUST be called at least twice with 2 different peaks before performing any M/Z scans					
CBS	Set Peak Center	None	Peak M/Z, Voltage	CalibrationComponent	None	None
	Manually sets the value of a calibration peak center - useful on initial calibration as well as if the system becomes uncalibrated					
CBN	Calibrate Now	None	None	CalibrationComponent	CalibrationComponent	None
	Forces the system to autocalibrate all its calibration peaks					
CBR	Calibration Report	None	# N Peaks Calibrated, Peak0 M/Z, Peak0 Voltage.....PeakN M/Z, Peak N Voltage	All	CalibrationComponent	None
	The report generated after a CBN includes all the peaks calibrated and at what voltage they were found					
CSR	Set Calibration Resolution	None	Resolution in units of M/Z of calibration scan	CalibrationComponent	None	None
	Sets the resolution with which the CalibrationWidth area is scanned, in essence the stepsize of the search					
CSW	Set Calibration Width	None	Size of window in units of M/Z to search for peak	CalibrationComponent	None	None
	Sets the width of the window (in M/Z terms) where the peak is scanned for in calibration					
CCB	Check Calibration	None	None	CalibrationComponent	CalibrationComponent	None
	Forces a check of the calibration If something is wrong, such as a fallen calibration peak or a long time passing, calls a CBN					

Figure A-1: Calibration NMessages

Scanning

ExternalName	Full Name	String Args	Double Args	Current Target Components	Internally Generating Components	Embedded Message
DAC	Set DA Voltage	None	Voltage To Be Set On DAC	NereusComponent	CalibrationComponent, ScanComponent	None
	Used to set the voltage on the Vucan DAC output, will cause a DLU to be sent if this is a large change from the previous scan					
ADC	AD Conversion	None	None	NereusComponent	CalibrationComponent, ScanComponent	None
	Used to get the value of electrometer volts currently read in off the DAQ board					
ADR	AD Conversion Report	None	Voltage On DAC, Electrometer Volts	Non Specific	NereusComponent	None
	Reply to an ADC request, currently signal averaged as set by SSA command (default = 500)					
SSA	Set Signal Averaging	None	Times to average a signal	NereusInterface	None	None
	Sets the averaging of samples before the NereusInterface returns an ADR					
SVR	Scan Voltage Range	None	Start Voltage, End Voltage, Increment	ScanComponent	None	None
	Scans the voltage range, generating an ADReport for each point that other components can watch for					
SMZ	Scans a M/Z Value	None	M/Z	ScanComponent	None	None
	Handles the request of a conversion between M/Z and Voltage scan and then requests the proper DAC and ADC before returning an MZR upon reciept of the ADR					
MZR	MZ Report	None	M/Z, Scan Voltage Used, Electrometer Voltage Readout	All	ScanComponent	None
	The report gerated after a Scan MZ request					
SMR	Scan Mass Range	None	Start M/Z, End M/Z, Increment	ScanComponent	None	None
	Scans the mass range, generating an MZReport for each point that other components, such as the Cache Component, can watch for					
CMV	Convert MZ into DA Volts	None	M/Z To Be Converted	CalibrationComponent	ScanComponent	None
	The request to convert a MZ into DA volts.					
CMO	Conversion Output	None	Voltage For Last CMZ Request	ScanComponent	CalibrationComponent	None
	The reply to CMV					

Returning Latest/Cached Scans

ExternalName	Full Name	String Args	Double Args	Current Target Components	Internally Generating Components	Embedded Message
MRR	Mass Range Request	None	Start M/Z, End M/Z, Increment	All	None	None
	The method to request the data from the last scans of a mass range at a certain increment from Start to End - returns 0 in any location not scanned					
MLR	Mass List Request	None	List Of Requested M/Z's Arbitrarily Long (eg. 18,20,28,40)	CacheComponent	None	None
	The method to request the last data scanned at the points specified in the list of masses, returns 0 in the place of anything not scanned					
MLO	Mass List Output	None	List Of Reported Electrometer Readings As Long As Last MLR Referring To Those M/Z's	None	CacheComponent	None
	The reply to either MRR or MLR, should follow right after MRR or MLR with the requested information					

Figure A-2: Scan NMessages

Utility Messages

ExternalName	Full Name	String Args	Double Args	Current Target Components	Internally Generating Components	Embedded Message
LSF	Load Schedule File	Filename	None	ScheduleFileLoader	NControl Calls This With The Init File	None
	Has the ScheduleFileLoader parse in the text file In Filename and sends ScheduleNewEvent messages for message line Filename					
MVS	Message View Set	Names Of Messages You Want To Toggle, Use "ALL" For All	1 or 0 depending if You Want To Toggle The Specific Messages On (1) or Off (0)	MessageViewComponent	None	None
	If using the system in console mode or on the command line, filters the outcoming messages based on a listing that you select					
ERE	ER Enable	None	None	ERComponent	None	None
ERD	ER Disable	None	None	ERComponent	None	None
DBT	Debug = True	None	None	Non Specific	None	None
	A reserved keyword to signal to components with Debug modes whether they should behave as if debug = true					
DBF	Debug = False	None	None	Non Specific	None	None
	A reserved keyword to signal to components with Debug modes whether they should behave as if debug = false					
DLS	Delay Seconds	None	Seconds	DelayComponent	None	None
	Has the DelayComponent delay the system for a period of Seconds seconds					
DLU	Delay Microseconds	None	Microseconds	DelayComponent	NereusComponent On Large M/Z Hops	None
	Has the DelayComponent delay the system for a period of Microseconds microseconds					

Internal Messages

ExternalName	Full Name	String Args	Double Args	Current Target Components	Internally Generating Components	Embedded Message
PM1	Peak Max Request	None	Contains List Of K D/A Volts	GSLComponent	CalibrationComponent	None
	The first half of a pair of requests that finds the center of a peak					
PM2	Peak Max Request	None	Contains Associated List of K Electrometer Readings	GSLComponent	CalibrationComponent	None
	The second half of a pair of requests that finds the center of a peak					
PMG	Peak Maximum Report	None	Center Voltage, Measured Current Closest To Center	Non Specific	GSL Component	None
	The report of a peak, where it was found for the last PM1 & PM2 messages					
GNE	Get Next Event	None	None	ScheduleComponent	NControl	None
	Sent by the NControl to get the next event from the ScheduleComponent					
SNE	Schedule New Event	None	#Times Embedded Message Should Run	ScheduleComponent	ScheduleFileLoader	Message To Be Scheduled
	Sent by the ScheduleFileLoader to schedule a new event in the event queue. The event is the embedded message to be run #Times					
LSR	Load Serial Commands	None	None	SerialComponent	Ncontrol uses this to probe for Serial Input	None
	The request made by NControl to the SerialComponent to load any new incoming commands					
SRO	Serial Output	String To Be Sent To Serial Port	None	SerialComponent	Non Specific	None
	The message sent that the SerialComponent listens to to put a message on the serial output					

Figure A-3: Miscellaneous NMessages

Appendix B

Schedule Files

B.1 Entry Format

To understand these with schedule files and specifically the ScheduleFileLoader, simply use the 3 letter initials for the message you would like to send, a semicolon, the string arguments or % if there are no string arguments, another semicolon, the double arguments, or a % to signify no double arguments, and the number of times you want the message to be sent in the event sending loop. This is illustrated in Figure B-1. If the number is initialized as a negative, it is sent an infinite amount of times.

B.2 Example Schedules

To see how I actually got the data displayed in Chapter 4, Figures B-2 to B-4 are some example schedules. Note the use of the MVS to select for specific information to be returned to the user. Figure B-2 scans all the voltages between -5 Volts And 0

```
MessageName;StringArg1,...StringArgN;DoubleArg1,...DoubleArgN;Times
MessageName;%;DoubleArg1,...DoubleArgN;Times
MessageName;StringArg1,...StringArgN;%;Times
MessageName;%;%;Times
```

Figure B-1: Schedule File Entry Format Templates

```
MSV;ADR;1;1
ERE;%;%;1
SVR;-5,0,0.001;1
```

Figure B-2: Scan Voltage Range

```
MSV;MLO,CBR;1;1
ERE;%;%;1
CBA;%;10,-2.18;1
CBA;%;40;-1.26;1
CBN;%;%;-1
SMR;%;15,100,0.5;-1
MLR;%;16,18,20,28,40,44;-1
```

Figure B-3: Calibration And Mass List Example

Volts, outputting *only* the ADR reports that record the voltage at each point. This is done only once, as you can see that the number-of-times argument is 1. Figure B-3 demonstrates the system's ability to add calibration points with initial guesses, calibrate, and scan many masses in a range. It then only reports what you specifically want to see as specified by the Mass List Request, MLR, message. The system can log the rest internally or wait until you ask about them. As you can see, the initial calibration items are done once, while the rest of the items are done with infinite repetition. Figure B-3 shows the sequence of commands to produce a zooming in calibration. Using CSW's to change the width, you see that the system is so flexible that you can even change the calibration routine in a meaningful way without ever getting down to the code level

```
CSR;%;0.01;1
CSW;%;0.5;1
CBN;%;%;1
CSW;%;0.25;1
CBN;%;%;1
CSW;%;0.125;1
CBN;%;%;1
```

Figure B-4: A Prototype Zoom In Subsequence

Appendix C

The Embedded System

C.1 Ampro CoreModule 410

The embedded computer we use to run the Nereus Sphere is an Ampro CoreModule 410 x86 compatible machine running an STPC Elite 486 processor with 16MB of RAM. It has 2 serial ports, one of which is used for terminal command of the PC due to its lack of video support and no Ethernet or wireless network hardware.

C.1.1 OS Choice: Debian Linux

We originally attempted to use the supplied Ampro TimeSys GNU/Linux distribution with the computer, but the entire support structure is designed around systems with network support and large hard drives. Our disk is a 512 MB CompactFlash. Specifically, we are using a high speed Sandisk UltraII CF card in order to deliver the read/write times we need. While the 410 does not have onboard CompactFlash support, The supplied distribution would not run on such a small disk so I attempted to find a linux distribution that could be easily loaded without network support and have a small footprint. Many installers like to have network access for embedded systems as it makes the task of selecting installed components dynamic.

The result of a lot of hard work trying other OS's was to get a very basic Debian GNU Linux running on the machine. In the end, the basic Debian disk based installer

was by far the easiest solution out there. A technique I used to transfer files to the Flash so the floppy could see them was to create the partitions on the flash using a desktop pc, load the proper files for the installer, and mount these partitions when the installer was running using the installer's interface.

You *must* note that if you want to use console mode with Debian, you have to specify the CONSOLE keyword in the kernel arguments. The Debian installer has a dialog for setting the kernel arguments run by LILO and you can find more information about the specific settings you want the serial to have at http://www.tldp.org/HOWTO/Remote-Serial-Console-HOWTO/ .

C.1.2 File Transfer

While the sphere is sealed, we do not have any io access into or out of the system. My choice of file transfer, when necessary, is to use ZModem over the serial console connection. If you download the lrzsz_0.12.21-4 or newer package onto the embedded PC, it can either "SZ" to you or "RZ" whatever you send it using Minicom or Hyperterminal on your host computer (depending on whether you're using linux or windows.) Note, the speeds may be shockingly low as serial UARTS are much slower than today's commonplace gigabit Ethernet.

C.2 Compiling NSystem

Since I only used C++ code from the Standard Template Library, with the exception of the GNU Scientific Library in the backend of GNUComponent, loading the glib C++ libraries such as the following list allowed for normal compilation of all but the GSL reliant code. For that, simply compile with the -static flag to make sure the compiler does not dynamically link to the GSL. You can install the GSL libraries on the embedded computer, which I will probably do before departing for the summer, so that you never have to worry about dynamic versus static linking while compiling

STL and GSL compliant code.[1]

1. libstdc++6 3.4.3
2. gcc-3.4-base 3.4.3
3. libc6_2.3.2

Do remember that the compilation target architecture is by no means i686, and must be at least i486. If you use the architecture compilation flag in gcc for i386, the compiler will also optimize for size and speed on limited resource machines.

C.3 Running NSystem

To run NSystem, all you have to do is, on the command line, send the string "./NSystem *initfile*" as super user, where initfile the name of a schedule file that contains the first things you want to run. If *initfile* does not exist, the system will still run but be in limbo just waiting for your command. To change the way the system behaves, simply change the schedule files to the routine you want and run the executable with the newer files. You do not have to recompile. You need to be super user to access control of the parallel and serial ports.

[1]The GSL is available as a library package download at http://packages.debian.org/testing/math/libgsl0 for Debian linux and documentation for what you can do with it, including FFT's and Linear Algebra, is at http://www.gnu.org/software/gsl/. Please be aware that the GSL is GNU Licensed Software so anything built with it must also be freely distributed as open source software under those license terms.

Bibliography

[1] W. Bleakney and J.A. Hipple. A new mass spectrometer with improved focusing properties. *Physical Review*, 53, 1938.

[2] Richard Camilli. Creation and deployment of the nereus autonomous underwater chemical analyzer. Master's thesis, Massachusetts Institute of Technology, June 2003.

[3] J.E. Cohen. Estimates of coastal populations. *Science*, 278, 1997.

[4] Werner Heiesenberg. Uber den anschaulichen inhalt der quantentheoretischen kinematik und mechanik. *Zeitschrift fr Physik*, 43, 1927.

[5] Harold F. Hemond. A backpack portable mass-spectrometer for measurement of volatile compounds in the environment. *Review Of Scientific Instruments*, 62, 1991.

[6] J.J. Thomson. *Rays of Positive Electricity And Their Application To Chemical Analyses*, page 132. Longmans, Green, and Co, 1913.