

53

Design and Implementation of a Multiprocessor System for Position and Attitude Control of an Underwater Robotic Vehicle

by

Ella Marie Atkins

B.S., Massachusetts Institute of Technology (1988)

SUBMITTED IN PARTIAL FULFILLMENT OF THE
REQUIREMENTS FOR THE DEGREE OF

Master of Science

in

Aeronautics and Astronautics

at the

Massachusetts Institute of Technology

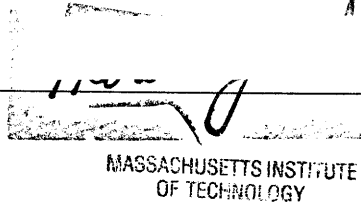
May, 1990

© Massachusetts Institute of Technology

Signature of Author _____
Department of Aeronautics and Astronautics
May 17, 1990

Certified by _____
Professor David Akin
Thesis Supervisor

Accepted by _____
Professor Harold Y. Wachman
Chairman, Department Graduate Committee



JUN 19 1990 **Aero**

LIBRARIES

Design and Implementation of a Multiprocessor System for Position and Attitude Control of an Underwater Robotic Vehicle

by

Ella Marie Atkins

Submitted to the Department of Aeronautics and Astronautics
on May 17, 1990, in partial fulfillment of the requirements
for the degree of Master of Science in
Aeronautics and Astronautics

ABSTRACT

The Multimode Proximity Operations Device (MPOD) is a neutral buoyancy simulation telerobot with the capability to fly in three dimensions and dock with an underwater satellite mockup. The vehicle may be flown from onboard, underwater remote, or surface control station. MPOD electronic systems are used to control motors, issue pneumatic commands, and read sensors. An onboard multiprocessor control system has been implemented. Five parallel processors are used to communicate with MPOD hardware and the pilot, read available sensors, calculate the vehicle state, and determine the desired control outputs.

MPOD position and attitude are determined in software via an extended Kalman filter. Sensors include a 3-axis rate sensor package, pressure transducer, pendulum inclinometers, and the 3-Dimensional Acoustic Positioning System (3DAPS), a group of underwater sound emitters and receivers.

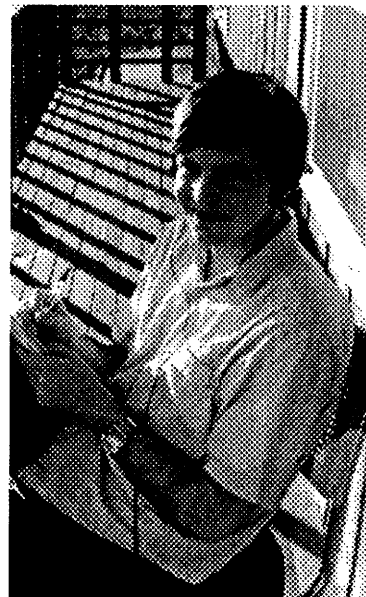
Once the vehicle state has been determined, the control computer calculates the motor commands required to implement the desired position and/or attitude changes. For attitude and position hold, the control equations are linearized. During large angle or position maneuvers, nonlinear terms are incorporated using feed-forward linearization.

This thesis describes the electronic design, sensor integration, and multiprocessor system implementation for MPOD flight control. Simulation and experimental results are presented for vehicle state calculation. The vehicle's estimate for its state vector consistently converged to within the expected error of MPOD's sensors.

Thesis Supervisor: David L. Akin
Assistant Professor of
Aeronautics and Astronautics

Acknowledgements

This thesis is dedicated to the three people who made the current MPOD and its exploits possible: Robert M. Sanner, Lisa K. Evelsizer, and Matt "I have a Llama" Machlis.



Rob waved his fingers over the computer keys, and proclaimed, "Let there be elegant, dependable MPOD software." And it was done. Besides writing PiVeCS and the filter software, he became a bottomless well of control system knowledge and guided me toward equations that often appeared to be written in a foreign tongue. He was a most excellent roommate, from providing fine kitchen smells and sugar-laden foods to shovelling the sidewalk during violent snow and ice storms. Perhaps someday he'll even learn to drive.

Lisa began her SSL life in the summer of 1988. Her expertise at building things and dedication to work many hours began immediately. She impressed all when she treaded water during an entire episode of Star Trek. In addition to displaying her tongue often during January 1989, she won the invisible "Most Sacrificial Student" award in Huntsville by stripping live wire with her teeth, while underwater. During the Spring of 1989, Lisa was led down the path of digital electronics and one Todd Barber. She was never the same, entering the world of circuit design, wire wrapping, and Barry Manilow. In the future, she may be remembered as the "Fiberglassing Goddess" of the laboratory. However, I will forever be thankful that her dedication to MPOD and the laboratory never disappeared, even during her final term at MIT, when I couldn't have supported pool tests without her.

Matt was once rumored to be "King of the One-Shots" among exclusive SSL circles. His work with 3DAPS both educated him about electronics and trapped him in the position of assisting needy lab types with 3DAPS. He learned FORTH and programmed MPOD's 68HC11's. Debugging MPOD's onboard 3DAPS with Matt was a true joy. It meant being exposed to his desert-like sense of humor and the fine Elmer Fudd laugh. Fortunately, his violent behavior with guns has to date only caused the destruction of paper and parking tickets.

In addition to Lisa, Rob, and Matt, Colleen Daugherty, a transient but wonderful member of the MPOD crew, is most heartily thanked. She was always excited about work, even when I moped about the lab. I hope she is happy in Ohio or wherever she may be. MPOD and I miss you.

Vicky, Vicky, (Vicky Rowley). I need to kiss your neck. From being the control station and initial software part of the MPOD team to building stereo cameras to help MPOD see in 3-D, she immensely aided my association with MPOD. But most importantly, she was my bestest friend, and always made me laugh. Then there was this weekend during June 1988, when she wanted to visit her friend Deano in Connecticut... She has made my life much happier and mushier. I hope to frequently meet her at random times and locations, even if she does live in Philadelphia.

From a purely emotional standpoint, I wish to express all my deepest feelings for one Mr. Deano R. Smith. Although he lives some 3000 miles away and was out to sea for many moons, he is my constant source of happiness, even if he is still a little too excited about physics and mathematics. After all, the human body may be described by equations. Of course, more experimentation may be necessary...

Merci beaucoup to all the main MPOD divers: Wisdom Franchot Coleman, the perfect fighter pilot, Paul Duncan, Karl von Ellenreider, and (Professor) Sandy Alexander. Much gratitude to the Fong Unit, SPAM owner, acronym maker, and programmer extraordinaire. Also, Russ Howard is worshipped for his valuable auto advice and assistance, although he may never get Fang to Alaska on its own power. Praise be to the still pure Karl "Killer" Kowalski, for initiating loud arguments and providing amusing but useful critiques of my thesis. Much gratitude and jealousy to Kurt "Smut-Man" Eberly, who typed my lost data into the Macintosh at the last minute, for \$1.75, and whose car is 6" longer than mine. Thanks to all the other lab types who made life so much better: Jud the engaged man, Ender, Anna my dive class buddy, Sayan, and Ali. Finally, thanks to Professor Dave, who trusted me with MPOD, provided an element of stability in the violent world of graduate students, and who is one of the nicest people I know. I hope both SSL's are tremendously successful.

Table of Contents

1.0	Introduction	8
1.1	MPOD: Previous Electronic Systems and Experiments	8
1.2	Scope of This Thesis	9
2.0	The MPOD System	10
2.1	Physical Description	10
2.2	Electronics	20
2.3	Computer Systems	33
2.4	Surface Control Station	39
3.0	MPOD Dynamics and Physical Parameters	44
3.1	MPOD Equations of Motion	44
3.2	MPOD Physical Constants	47
3.3	Sensor Feedback -- Calibration and Accuracy	49
4.0	The Control System	52
4.1	State Calculation	53
4.2	Position and Attitude Hold	56
4.3	Automated Maneuvers	63
5.0	Experiments and Test Results	67
5.1	MPOD Simulation	68
5.2	Underwater Static Tests	81
5.3	Underwater Dynamic Tests	84
6.0	Conclusions and Recommendations	110
6.1	System Accuracy and Robustness	110
6.2	Future Control and Human Factors Experiments	112
6.3	Conclusion	114
7.0	References	115
8.0	Appendices	117
8.1	Appendix A -- Circuit Diagrams	117
8.2	Appendix B -- MPOD and Control Station Software	125
8.3	Appendix C -- Parameter and Calibration Calculations	183
8.4	Appendix D -- Simulation Software	188
8.5	Appendix E -- Control System Parameter Calculations	199

List of Figures

Figure 1.	MPOD During Docking Task	11
Figure 2.	MPOD Assembled Side View	12
Figure 3.	MPOD Top Cutaway View	12
Figure 4a.	MPOD Disassembled Left Side View	14
Figure 4b.	MPOD Disassembled Right Side View	14
Figure 5.	Docking Probe and Target	15
Figure 6.	Pressurization System Diagram	17
Figure 7a.	3DAPS Thumper Locations -- MIT Pool	19
Figure 7b.	3DAPS Hydrophone Locations on MPOD	19
Figure 8.	Electronics System Diagram	21
Figure 9.	Control Box Internal Layout	22
Figure 10a.	Pneumatic System Diagram	24
Figure 10b.	Solenoid Circuit: Single Channel	24
Figure 11.	Motor Control Circuit: Single Channel	26
Figure 12a.	Pendula Diagram	27
Figure 12b.	Pendula Circuit	27
Figure 13.	3DAPS System Diagram	29
Figure 14.	Multiple Processor Interface Diagram	31
Figure 15.	Fiber Optics Connections	32
Figure 16.	Yoda Software Logic	34
Figure 17.	Obi-Wan Software Logic	36
Figure 18.	Lando Software Logic	38
Figure 19.	Control Station Layout	40
Figure 20.	Control Station Functional Diagram	41
Figure 21.	Luke Keyboard Functions	42
Figure 22.	Luke Software Diagram	43
Figure 23.	3DAPS Range Calibration Plot	52
Figure 24a.	Position Hold Block Diagram	59
Figure 24b.	Position Hold Control Diagram	59
Figure 25a.	Attitude Hold Block Diagram	60
Figure 25b.	Attitude Hold Control Diagram	60
Figure 26a.	Position Maneuver Block Diagram	64
Figure 26b.	Position Feed-Forward Control Diagram	64
Figure 27a.	Attitude Maneuver Block Diagram	66

Figure 27b.	Attitude Feed-Forward Control Diagram	66
Figure 28.	Static Simulation Position Plots with Low Initial Covariance	70
Figure 29.	Static Simulation Position Plots with High Initial Covariance	71
Figure 30.	Static Simulation Attitude Plots with Low Initial Covariance	73
Figure 31.	Static Simulation Attitude Plots with High Initial Covariance	74
Figure 32.	State Estimator Position Tracking	76
Figure 33.	State Estimator Linear Velocity Tracking	77
Figure 34.	State Estimator Attitude Tracking (q0 & q1)	78
Figure 35.	State Estimator Attitude Tracking (q2 & q3)	79
Figure 36.	State Estimator Angular Velocity Tracking	80
Figure 37.	Static Test Pool Locations	82
Figure 38.	Constrained (-y) Flight -- Positions	86
Figure 39.	Constrained (-y) Flight -- Linear Velocities	87
Figure 40.	Constrained (-y) Flight -- Quaternions and Angular Velocities	88
Figure 41.	Constrained (-y) Flight -- Thumper 0-3 Unblocked Ranges	89
Figure 42.	Constrained (-y) Flight -- Thumper 5-7 Unblocked Ranges	90
Figure 43.	Docking Run -- Positions	92
Figure 44.	Docking Run -- Linear Velocities	93
Figure 45.	Docking Run -- Attitudes (q0 & q1)	94
Figure 46.	Docking Run -- Attitudes (q2 & q3)	95
Figure 47.	Docking Run -- Angular Velocities	96
Figure 48.	Docking Run -- Thumper 0-3 Unblocked Ranges	97
Figure 49.	Docking Run -- Thumper 5-7 Unblocked Ranges	98
Figure 50.	Flight with Roll -- Positions	100
Figure 51.	Flight with Roll -- Linear Velocities	101
Figure 52.	Flight with Roll -- Attitudes (q0 & q1)	102
Figure 53.	Flight with Roll -- Attitudes (q2 & q3)	103
Figure 54.	Flight with Roll -- Angular Velocities	104
Figure 55.	Attitude Maneuvers -- Positions	105
Figure 56.	Attitude Maneuvers -- Linear Velocities	106
Figure 57.	Attitude Maneuvers -- Attitudes (q0 & q1)	107
Figure 58.	Attitude Maneuvers -- Attitudes (q2 & q3)	108
Figure 59.	Attitude Maneuvers -- Angular Velocities	109

1.0 Introduction

The LOOP (Lab of Orbital Productivity) Group of MIT's Space Systems Laboratory (MIT SSL) was begun in 1978 to study possible applications of space systems in an underwater environment. Zero-gravity is simulated by making all submerged hardware neutrally buoyant, both in depth and attitude. Currently, NASA projects such as space station assembly and maintenance can benefit from the use of either teleoperated or partially autonomous robots. As a major requirement, proposals are under development for the systems and algorithms required to enable automated docking of a free-flying space vehicle. Neutral buoyancy simulation tests are producing useful results for the development of this and other space hardware.

The MIT Space Systems Lab has studied robotics during neutral buoyancy simulation for the past ten years. The Beam Assembly Teleoperator (BAT) was the lab's first functional underwater robotic vehicle. Its main task was to assemble truss structures, such as the SSL's EASE (Experimental Assembly of Structures in EVA) project. During structural assembly, a person operating BAT from a surface control station received stereo video feedback and uses master/slave systems for manipulation, with joysticks for open-loop flight (Reference 1).

The Multimode Proximity Operations Device (MPOD) was built concurrently with BAT to study different modes of pilot operation and possible control algorithms during vehicle free flight. MPOD's main task was designed to be docking to an underwater mockup of a satellite. An onboard cockpit with removable controls allows the pilot to be at either an onboard or remote control site. The importance of various motion and visual feedback situations can be quantitatively studied through pilot performance comparisons during vehicle free-flight and docking runs.

Because docking was a relatively simple task, MPOD research evolved from primarily human factors studies to control system experimentation. The final goal of this research phase is autonomous underwater docking. Sensors which enable completion of this task include 3-axis rotational rate transducers, pendulum inclinometers, a depth sensor, and an acoustic emitter/receiver system (3DAPS) which would enable an estimate of vehicle position and attitude (Reference 2).

1.1 MPOD: Previous Electronic Systems and Experiments

When MPOD was first designed (Reference 3), it was flown open-loop, using two underwater discrete hand controllers. Next, a closed-loop control system was

implemented, which used a 3-axis rate gyro package as its sole sensor input. Since only angular velocity was measured, the system provided angular rate damping but no actual attitude control.

The second generation of MPOD development incorporated pendulum inclinometers with the rate gyros to enable a proportional-integral-derivative (PID) attitude control system (Reference 4). While using the PID system, human factors comparisons of onboard vs. remote pilot control were performed. Because of limited test time, the results were not definitive, but did suggest that remote control with direct visual feedback was the preferred mode of operation.

The third generation of tests used a similar PID control system (Reference 5). Pilot visual feedback comparisons were performed during docking maneuvers. An operator flew the vehicle from a surface control station, where graphics video overlays and stereo vision were used. Results showed that a cross-hair graphic overlay on the MPOD video screen enhanced pilot docking performance.

During the set of visual experiments, the idea for MPOD's autonomous flight and docking was first spawned. It was determined that multiple processors for calculations and integration of the 3DAPS data into MPOD's electronics were necessary. The current MPOD system is based on these requirements. All design and experimentation described in this thesis may be considered the fourth generation MPOD.

1.2 Scope of this Thesis

The sensing capabilities and calculation algorithms for autonomous MPOD flight required reliable hardware, fast computer processing, and extensive system integration. The old system had experienced numerous electronic failures primarily due to leaky connectors. To solve this problem, the electronic systems were integrated into a single control box, thus minimizing underwater connections. In addition to leakage problems, the old onboard computer possessed little RAM, could not support a math coprocessor, and had to be programmed by EPROM. This computer was replaced by a series of three single-board processors (one V40, two 80C286's with coprocessors). These computers run IBM-PC compatible software and may be programmed as would any IBM-standard device. The 3DAPS system and a depth sensor were added to MPOD's sensors to enable both position and attitude calculation.

This thesis describes the design and implementation of the current MPOD electronics and computer systems, software, and algorithms used for vehicle control. Sensor calibration tests are shown. Results displaying the accuracy of the estimated vehicle state

vector are presented. Because of time constraints, the final implementation of the automated docking control software was never completed. However, the algorithms and computer simulation test results are presented with the expectation that these algorithms will enable autonomous MPOD docking in future tests.

2.0 The MPOD System

2.1.0 Physical Description

The MPOD vehicle's primary task is satellite docking. Figure (1) shows the final stages of an underwater docking run. MPOD's octahedral configuration was a compromise between ease of construction and minimization of water drag. A perfect sphere would be the most efficient shape, but would be almost impossible to construct. An octagonal body approaches a sphere, but allows the frame and surface to be constructed from rectangular and triangular components. The vehicle's frame is composed of 1" aluminum box beams, inter-connected with riveted gusset plates and covered by foam and fiberglass panels. Overall dimensions are approximately 2 meters along each of the vehicle axes. MPOD's size was primarily determined by the requirement of an onboard cockpit. The cockpit and pilot entry area comprise the upper and central portion of the vehicle. Pressurization, electronic, and power supply components are contained below and to the sides of the open cockpit.

Figure (2) shows a simplified side view of the assembled MPOD with docking probe. With twelve motors driving ducted propellers, the vehicle has the ability to move in the six degrees of freedom associated with three-dimensional free-flight. Since the motors are located in pairs along the vehicle principal axes at a distance of 0.81m from MPOD's center, there are equal thrust levels for all translational directions and equivalent torques about all rotational axes.

The motors are driven by MPOD's onboard electronics and power system. Thruster magnitude and direction are determined by either the open-loop commands from a pilot or onboard closed-loop control routines. Sensors for the control system include: 3-axis fluidic rate transducers, pendulum inclinometers, a depth sensor, and the Three-Dimensional Acoustic Positioning System (3DAPS). The sensor information is converted

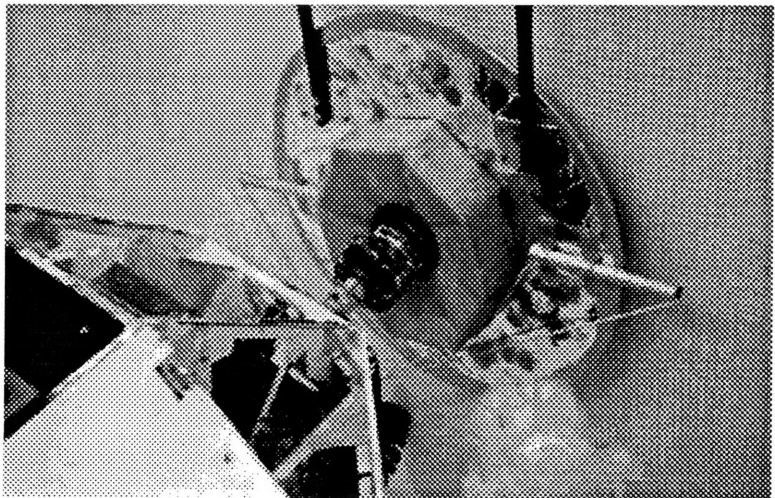
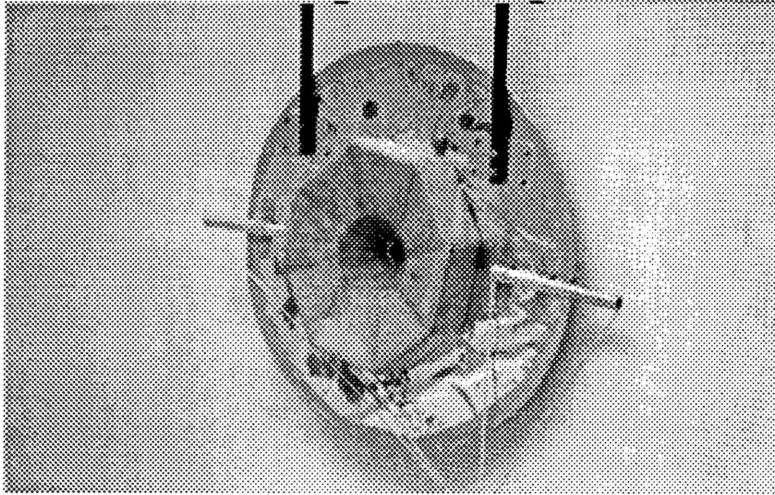
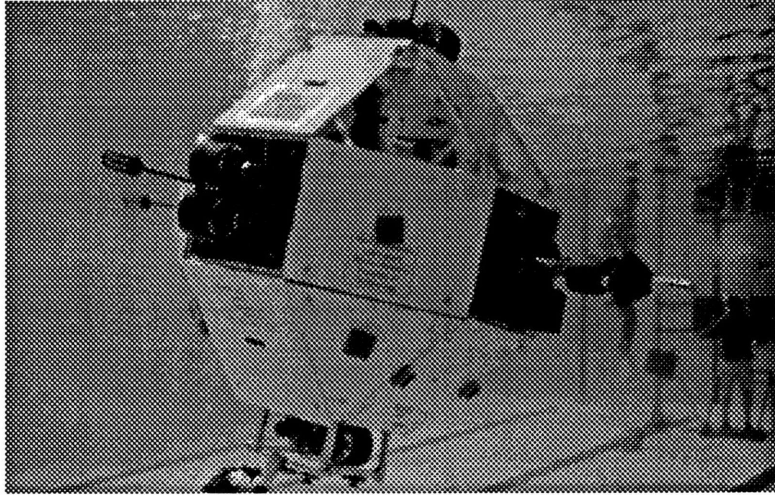


Figure 1. MPOD During Docking Task

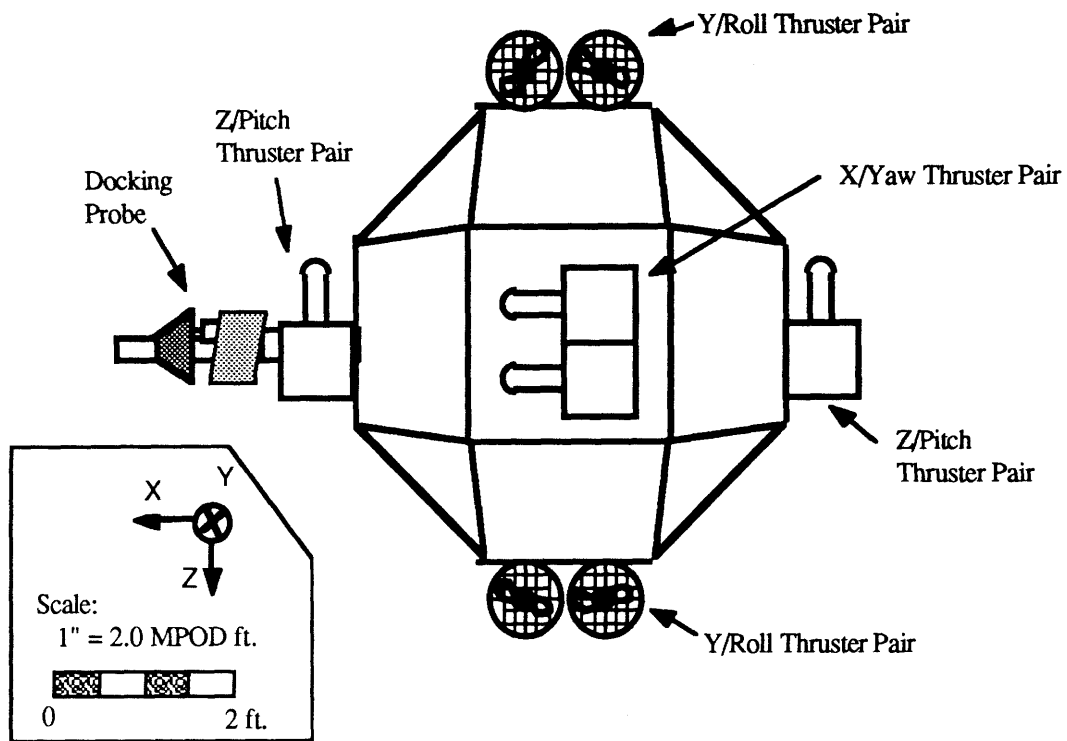


Figure 2. MPOD Assembled Side View

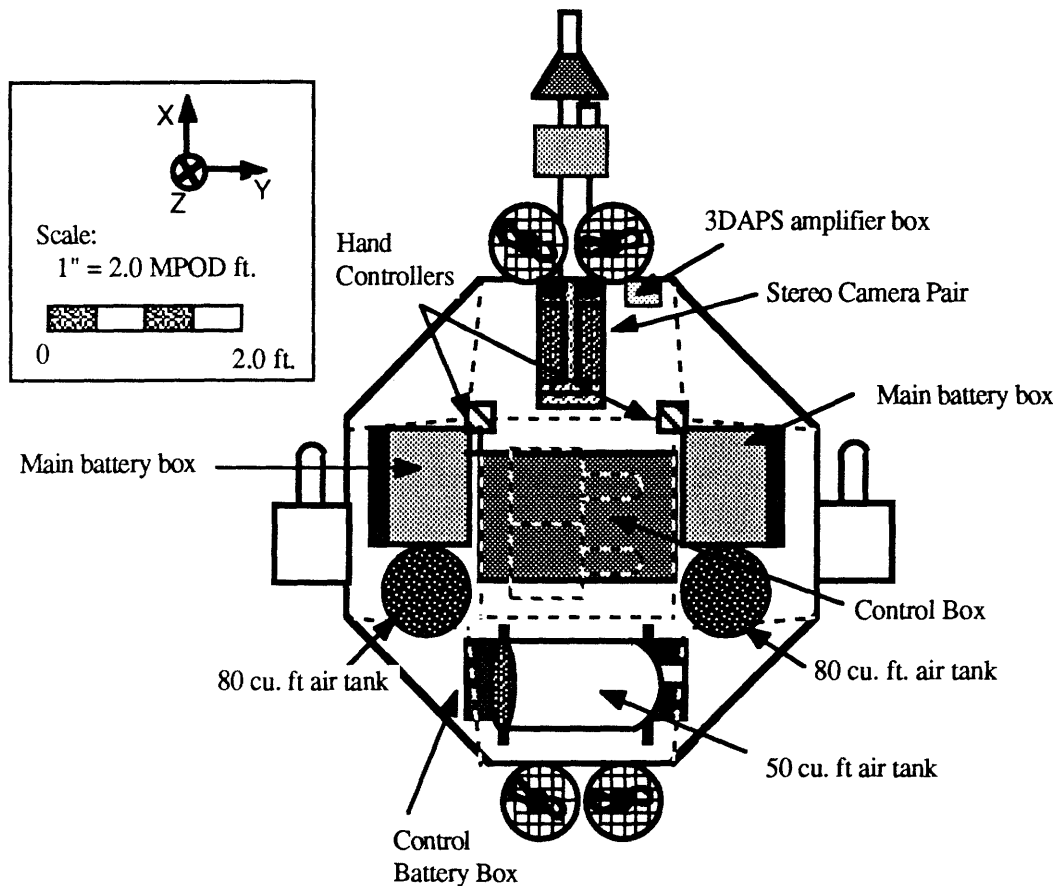


Figure 3. MPOD Top Cutaway View

into a state vector estimate by the state calculation computer, hereafter referred to as *Lando*. *Obi-Wan*, the control computer, uses the estimated state to calculate appropriate thrust values. A third processor, *Yoda*, communicates with MPOD hardware and continually looks for new instructions from the operator. See Section 2.3 for computer system details.

2.1.1 Components and their Locations

Since MPOD is submerged during operation, all electrical components must be housed in water-tight enclosures. Figures (3), (4a), and (4b) show the locations and relative sizes of MPOD's various boxes and air tanks. Most waterproof housings were constructed from foam and fiberglass (see Reference 6 for construction details). The side main battery boxes each contain three lead-acid battery packs supplying +18V for driving the vehicle's motors. This power is switched on and off by a pneumatically-driven relay, housed inside the cylindrical plexiglass power relay box. Since the instantaneous current draw on this system approached ~100A, all cables and connections were constructed accordingly.

The control battery box, located under MPOD's rear air tank, holds three +12V battery packs which run all onboard electronics and computer systems. These batteries also power the cockpit when the vehicle is flown from onboard. Two switches attached to the battery box enable divers to turn electronics on and off from underwater. Current draw from the control batteries during normal operation is ~5.5A.

MPOD's control box rests on rails under the pilot seat. Nearly all onboard electronics are contained within this box, with the exception of the solenoids for pneumatic cylinders, the 3DAPS hydrophone amplifier circuit, and the heat-generating motor power transistor block, which is mounted under the pilot seat right arm. Contained within the control box are: 3 computers, 2 microcontroller boards, the pendulum inclinometers, rate sensors, motor control circuitry, and 3 wire-wrapped boards which interface all the hardware I/O and the 3 processors. Note that the 3DAPS receivers and the depth sensor are not shown in Figures (2) - (4), but will be discussed in Section 2.2.

Two 80 ft³ side tanks supply air for an onboard pilot. They are strapped to MPOD's sides with scuba belts. Connected in parallel, they supply the pilot air through a standard scuba regulator. The 50 ft³ rear air tank provides pressurization for all MPOD systems. A high pressure line runs to the solenoid box for driving the pneumatic cylinders, while a modified second-stage regulator is used to pressurize the vehicle's waterproof boxes.

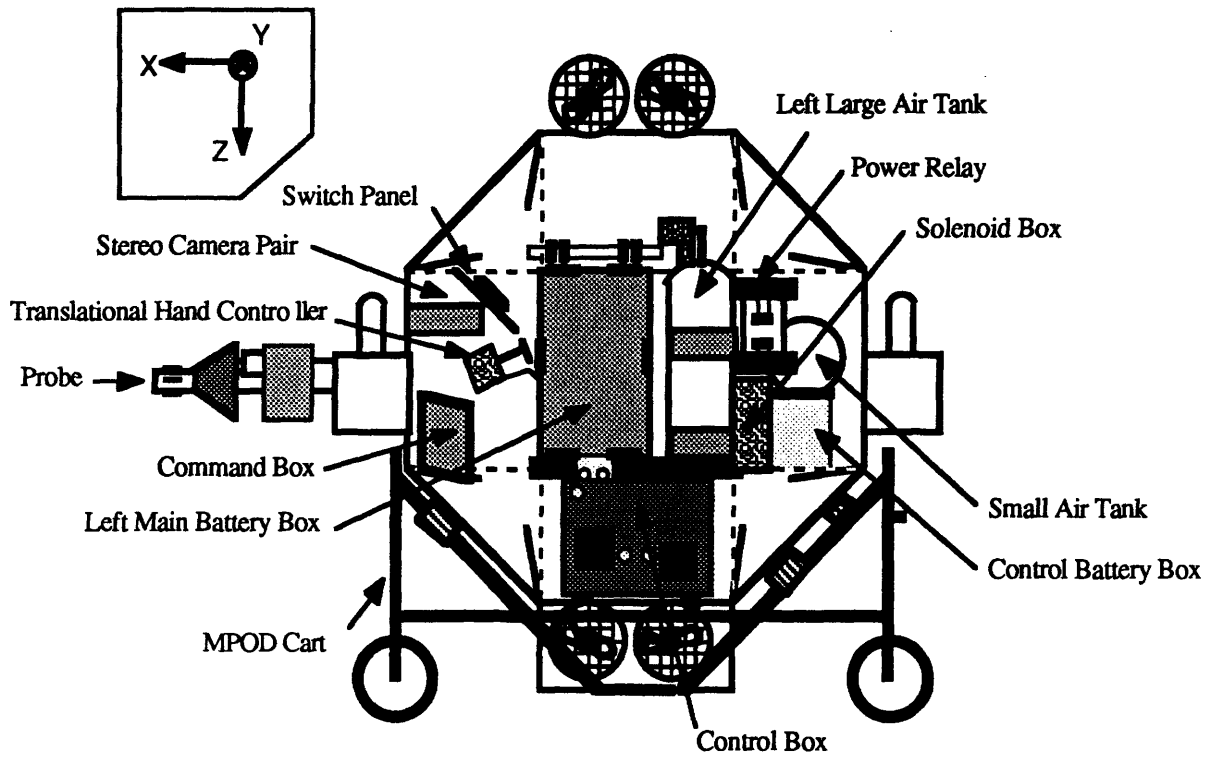


Figure 4a. MPOD Disassembled Left Side View

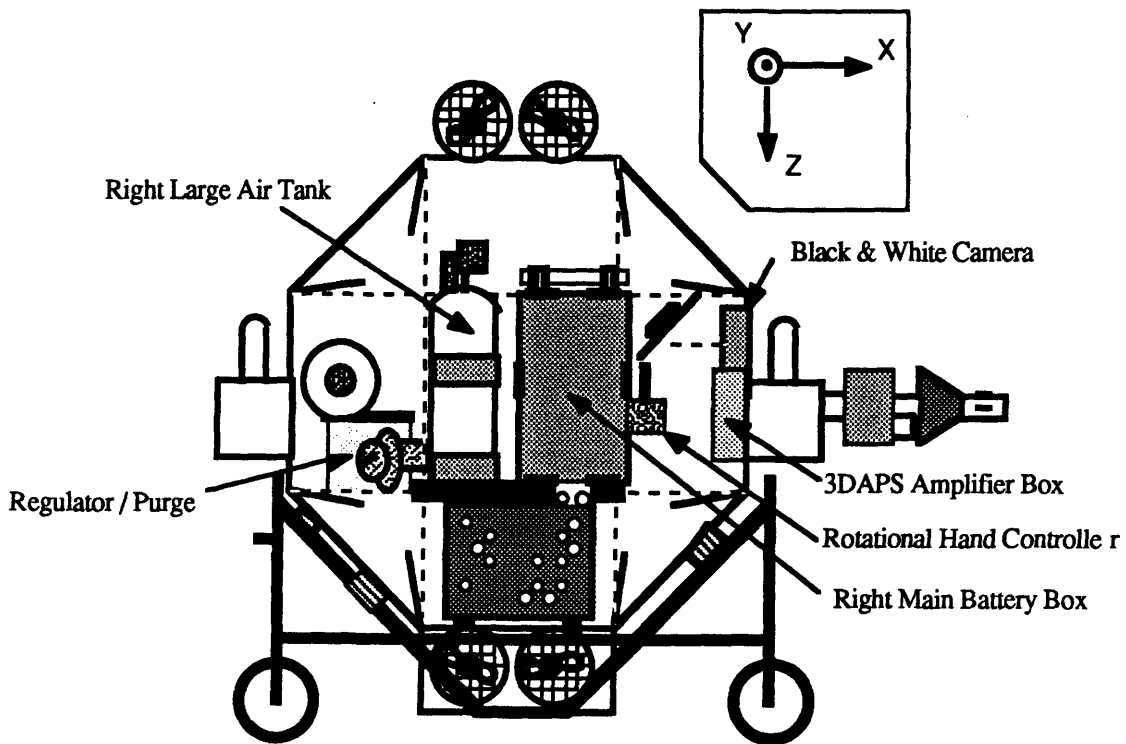


Figure 4b. MPOD Disassembled Right Side View

2.1.2 Docking Mechanism

As mentioned in previous sections, MPOD's primary task is docking to an underwater satellite mockup. This is accomplished via a removable probe, which latches and then rams its target for secure attachment. Figure (5) shows a close-up view of the probe and its mating drogue.

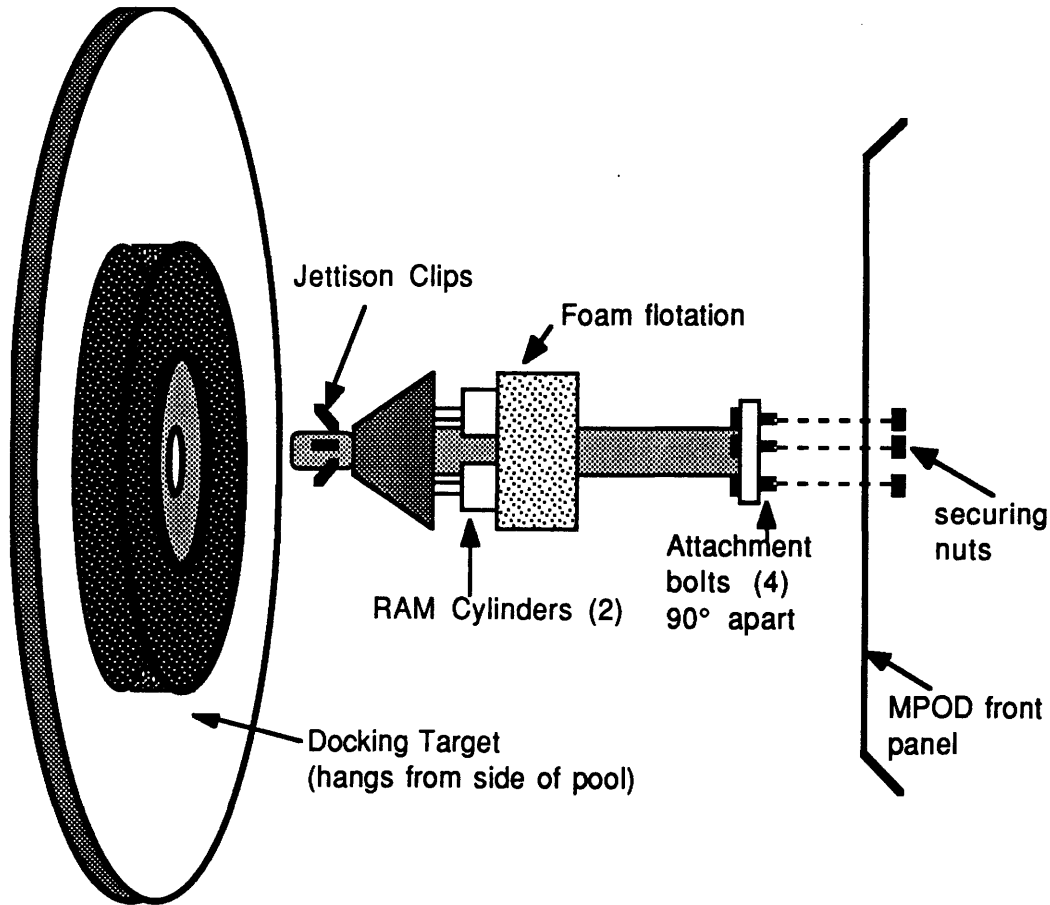


Figure 5. Docking Probe and Target

During flight, the conical ram fixture remains in a retracted state. The spring-loaded latches are set to capture and engage the circular target opening. Upon MPOD's penetration of the docking fixture, the ram is activated to extend into the target's conical opening. The secure fit of the probe prevents all relative motion between MPOD and the satellite. Both the latches and ram are activated by pneumatic cylinders. The latch cylinder and driving mechanism are contained within the probe, while the ram is activated by the two cylinders visible on the probe's exterior.

2.1.3 Pressurization System

Air pressurization systems play an important role during vehicle operation. Shown in Figure (6) is a diagram of all MPOD pressure systems. Onboard air for a pilot is required primarily because the cockpit is not large enough for a diver in scuba equipment to fit. The double tank system allows even a heavy breather to fly for an entire test session.

The small air tank supplies pressure to all MPOD systems. As a precaution against leakage, all water-tight boxes that do not contain hydrogen-venting batteries are pressurized at 2-3 psi above the outside water pressure. This system has two major advantages: the air leaks out instead of allowing water to leak into the boxes, and the bubbles exuded by a box quickly show the presence and location of leakage before the contents become wet. Air is constantly used, but it has been noted that an onboard operator breathes much faster than MPOD. Depending on the frequency of depth changes, a typical test run in the MIT Pool will use approximately 200 psi of air per hour from the small tank.

High pressure lines from the small air tank's first stage reducer activate the vehicle's pneumatic cylinders. These solenoid-activated cylinders operate the main power relay and docking probe. Because pneumatic state changes are activated infrequently during testing, they do not cause a significant drain on MPOD's air supply.

2.1.4 Video

A pilot flying MPOD from the surface needs visual information to determine an appropriate command sequence. Currently, two video systems are available for MPOD: a set of color stereo cameras with surface signal decoding circuitry (Reference 23), and one black and white camera with wide-angle lens. Figures (4a) and (4b) show the camera positions on MPOD. Each video system's waterproof housing is mounted inside the vehicle's front panel, with each lens pointing along the $+x$ axis. This location enables the pilot to view the docking probe and the target during approach. Power is sent from the surface to each camera. The video signal runs back through a shielded cable to a surface monitor.

Since an operator's ability to fly the vehicle was not the emphasis of this thesis, sophisticated video feedback was unnecessary. The single black and white camera was used for all experiments. It provided a clear view of the target, and enabled the operator to

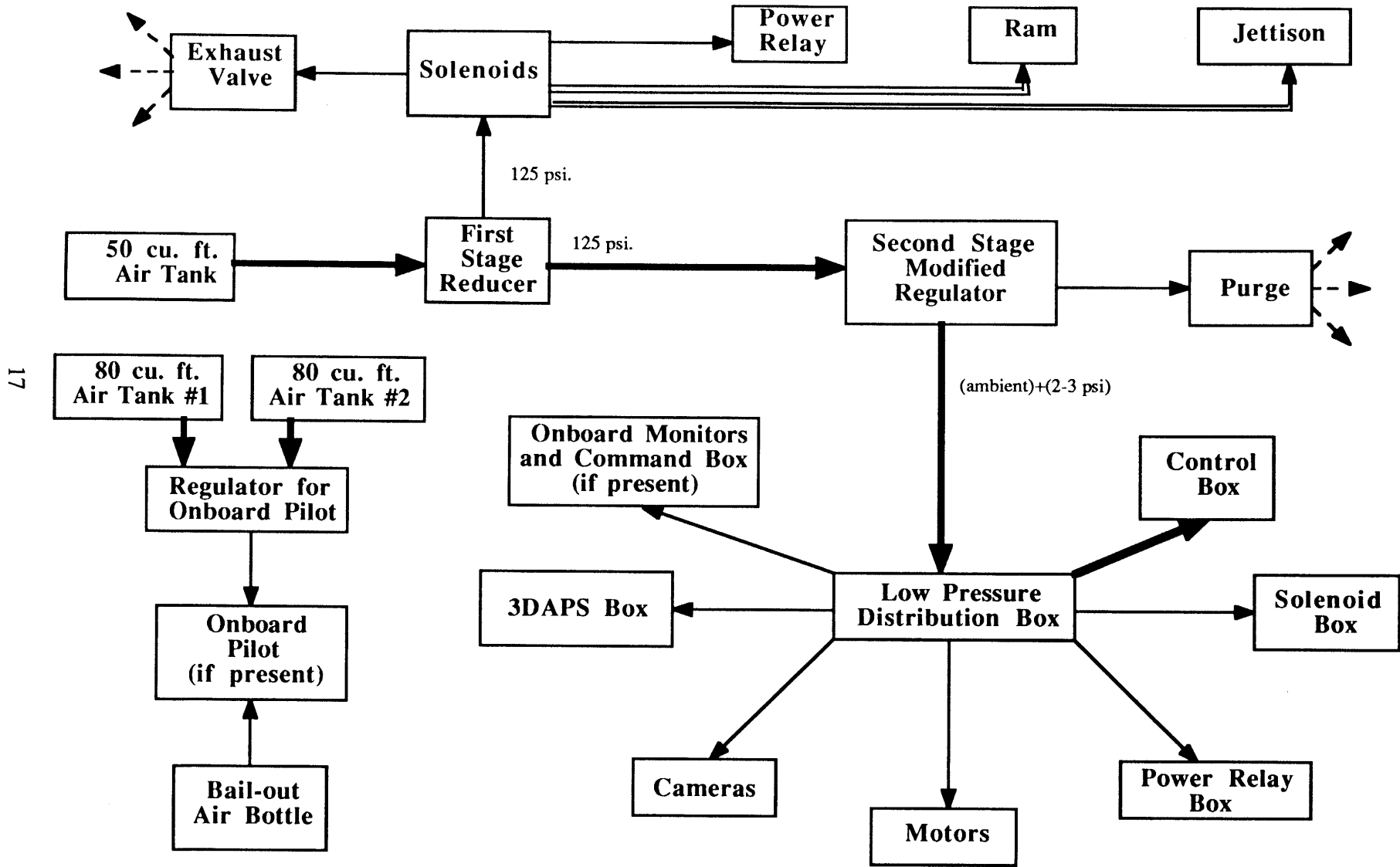


Figure 6. Pressurization System Diagram

keep a close watch over MPOD's flight progress. The video feedback was also used to determine the exact point at which MPOD engaged the docking target.

2.1.5 Control Box Description and Contents

The MPOD control box contains almost all the vehicle's onboard circuitry, and was built to enable relatively simple electronics modifications. Its size was determined by the vehicle frame dimensions, with the box filling the entire space below the pilot seat to within a few centimeters. The control box lid was made of anodized aluminum, serving the dual purposes of covering the box and dissipating the electronics-generated heat. Tension latches and a rubber gasket provided the seal between lid and box.

All external connections were made on the box ends, both for sizing and connectability reasons. Small gauge electrical wires were attached with Amphenol™ connectors. The larger (14 AWG) wires exit the box through resin and epoxy-filled threaded brass bulkheads, then terminate with Sure-Seal™ connectors. Fiber optic signals travel through the control box via plastic connectors with waterproofed internal box cables. Since a bulkhead connector significantly lowers the line's light intensity, the 3DAPS fast fiber optic line connects directly with the receiver chip, which has been insulated and mounted on the control box outer surface.

MPOD's control box is installed and removed by sliding it along mounting rails out the right side of the vehicle. When not inside MPOD, the box rests on a rolled cart which has the necessary sliding rail system and is the same height from the ground as MPOD's rails. The box lid may be removed and all connectors attached while the control box is not inside the vehicle. This is necessary for efficient debugging of electronics and software.

2.1.6 3DAPS Physical Aspects

The 3-Dimensional Acoustic Positioning System (References 2 and 7) was originally built to function independently of MPOD. 3DAPS mechanisms include a series of acoustic emitters, or thumpers, and acoustic receivers, or "hydrophones". Circuitry contained in a metal box at the surface called the sequencer serially fires the thumpers at regular intervals, set by the operator. It also detects when each thumper fires, and sends this information to the underwater receiver. The thumpers are positioned along the corners of a rectangular parallelepiped, and pointed toward the figure's central point. See Figure (7a) for the MIT Alumni Pool thumper locations.

Thumpers are constructed of rapid-firing solenoids which move a metal hammer into

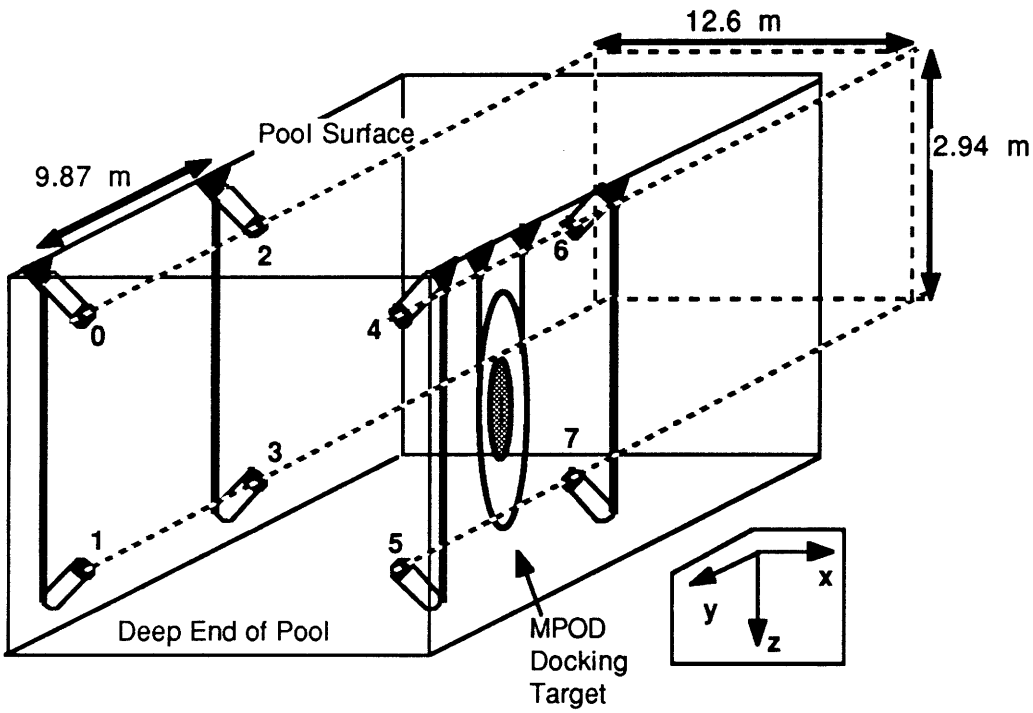


Figure 7a. 3DAPS Thumper Locations -- MIT Pool

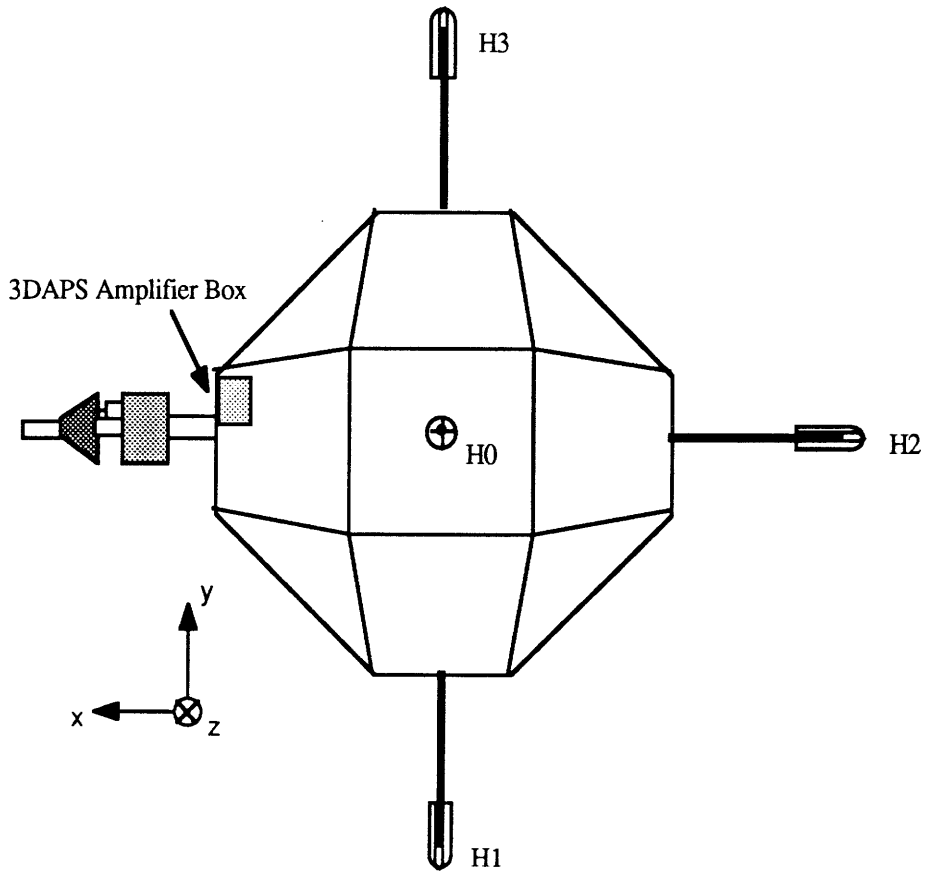


Figure 7b. 3DAPS Hydrophone Locations on MPOD

contact with a plate. This contact produces an acoustic pulse designed to be centered at a frequency of 100 kHz. Also, the solenoid and plate surfaces are connected to wires which produce an electrical pulse upon contact. This pulse is returned to the sequencer, which in turn sends the signal and corresponding thumper number via a 1 Mbaud serial fiber optic line to the 3DAPS receiver system.

Because MPOD processors needed real-time access to 3DAPS ranges, the original receiver system was redesigned and built to fit within MPOD's control box. Four Brüel and Kjaer hydrophones (Reference 8) send the received low-level (2-3 mV) acoustic vibration signals to the amplifier circuit.

After amplification, the hydrophone signals are sent to the control box, where distances between the four hydrophones and the fired thumper are calculated, and then sent to MPOD's computers for processing. Unfortunately, when the signal was being amplified inside MPOD's control box, RFI noise produced by the twelve motor relays corrupted the acoustic pulse. Thus the hydrophone amplifier circuit was moved to a separate small waterproof box outside the MPOD control box (see Figure (7b) for box location on MPOD). Because noise on the MPOD power and ground lines were also corrupting the low-level 3DAPS signals, a small 6V battery was placed in the 3DAPS box for powering the amplifier circuitry.

Since the hydrophones are fragile, they were enclosed in anodized aluminum "cages" to avoid damage from contact with pool walls, divers, and other obstacles. Figure (7b) shows the mounted locations on MPOD of the enclosed hydrophones. The large distance of each hydrophone from the vehicle center enables a better attitude estimate from 3DAPS. Because the long, narrow rods are easily bent, break-away bolts were used for hydrophone attachment to MPOD.

2.2.0 Electronics

MPOD's electronic systems may be divided into three categories: (1) hardware control electronics, (2) circuitry for interfacing all the hardware with all the computers, and (3) computers for communication and calculation. Figure (8) shows a block diagram of all MPOD electronic systems and their interconnections. RAM shared between the computers contains common variables and measurements. Intel 8255A multiplexers provide interfacing between all the hardware and computer data buses. A 12-bit A/D converter with multiplexer is used for reading the analog rate and depth sensor outputs, while HCTL-2000's read encoder counts for the pendula. 3DAPS ranges are determined by FORTH-programmed 68HC11 microprocessor boards. RS232 serial ports provide communication

with the surface computer and pilot via the fiber optic lines.

The following sections describe the MPOD electronic functions. First, a description of control box internal component locations and circuit card functional division is given to provide an overall picture of the actual hardware. Next, each major circuit is examined in detail.

2.2.1 Electronic Components and their Functions

The MPOD control box was designed to hold all onboard circuitry and computer systems. Shown in Figure (9) is the internal layout of the components. The three main computers, *Yoda*, *Obi-Wan*, and *Lando*, are mounted next to each other. The disk drive and NOVRAM (non-volatile RAM) cartridges used to store software are mounted near the computers to minimize ribbon cable length. Two 68HC11 microprocessor boards are mounted between the pendula and motor circuitry. They are connected only to the processor interface card, hence, the boards do not have the need to be near any supporting components.

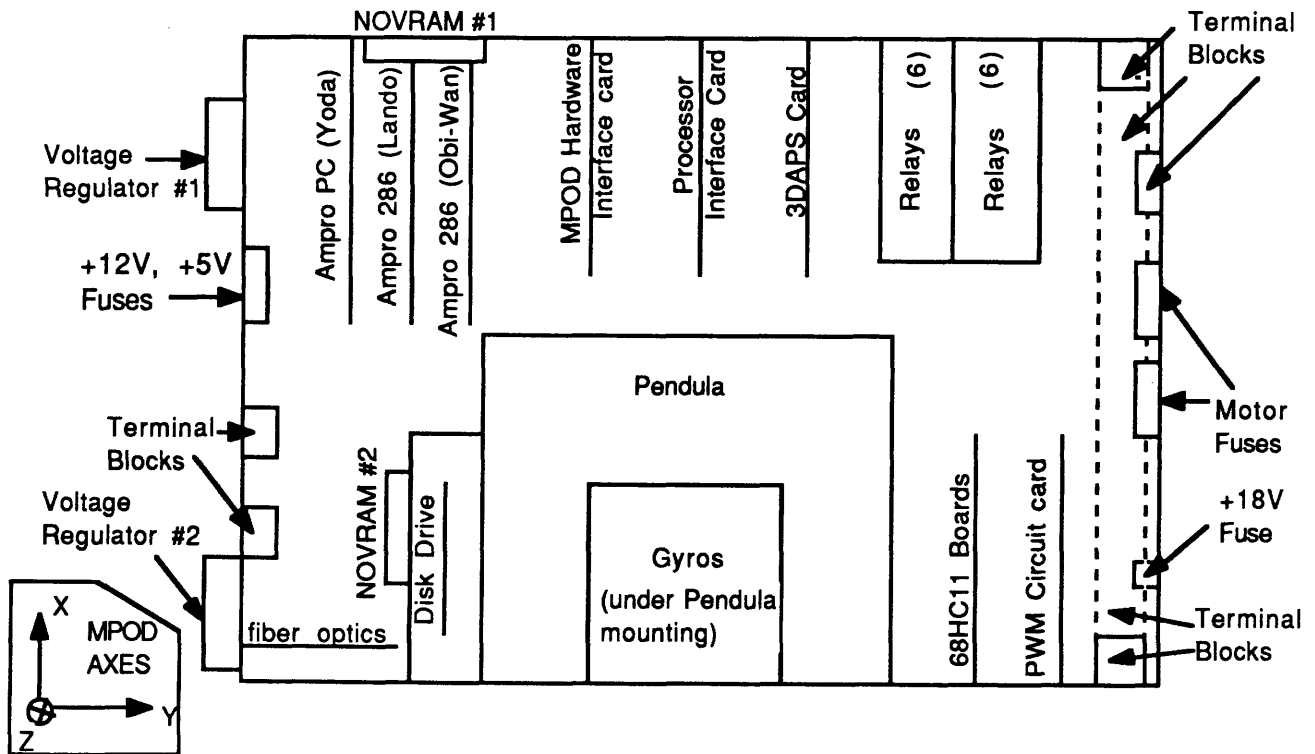


Figure 9. Control Box Internal Layout

As seen in Figure (9), there are five wire-wrapped circuit cards in MPOD's control box. The three cards opposite the pendula and rate sensors share information and power along a common edge connector. Together, they perform all the interfacing between the hardware and the processors.

The MPOD Hardware Interface Card, or MIC, is connected to *Yoda's* data bus. Motor magnitudes and directions as well as solenoids are commanded from this card. Also, MIC contains circuitry to process rate sensor, pendula, and depth information. The processed sensor readings are sent to *Yoda* via 8255's. Appendix A-1 shows the complete circuit diagram for MIC.

The Processor Interface Card, or PIC, has two primary functions. First, it connects the three computers (*Yoda*, *Obi-Wan*, and *Lando*) together with dual port RAM. Next, the 68HC11 ports are connected via 8255's to *Obi-Wan's* data bus for the reading of 3DAPS ranges. Appendix A-2 shows the complete circuit diagram for PIC.

The third card opposite the pendula is the 3DAPS Interface Card. The 3DAPS sequencer serial signal decoding circuitry is contained on this card. It also sends power to and receives signals from the 3DAPS hydrophone amplifier circuit (see Appendix A-6 for the amplifier circuit diagram). The 3DAPS Interface Card circuit diagram is shown in Appendix A-3.

Almost the entire +y section of the control box is devoted to motor circuitry and its wiring connections. The pulse-width modulation circuit is contained on the PWM circuit card, as labelled in Figure (9). Because of the high current requirement of each motor (20 amp maximum), 14-gauge wire was used for all power and motor signal connections. Relays are mounted in two groups of six (see Figure (8)). Each motor is fused, with the fuse mountings along the +y wall of the control box.

The final wire-wrapped circuit is used for the conversion of fiber optics light to/from electrical signals. There are five lines: two for *Yoda*, two for *Obi-Wan*, and one fast fiber optics line for 3DAPS. The conversion circuit card is mounted next to the onboard disk drive. Appendix A-5 shows the 5-channel fiber optics circuit diagram.

2.2.2 Solenoids

Both the docking probe and main power relay are driven by +12V pneumatic solenoids. Figure (10a) shows a single solenoid circuit. The TIP31 transistor is switched by a 1-bit signal from *Yoda*. The signal then travels from the control box to the solenoid box, where the appropriate device is activated. Upon switching of a solenoid, the pressurized

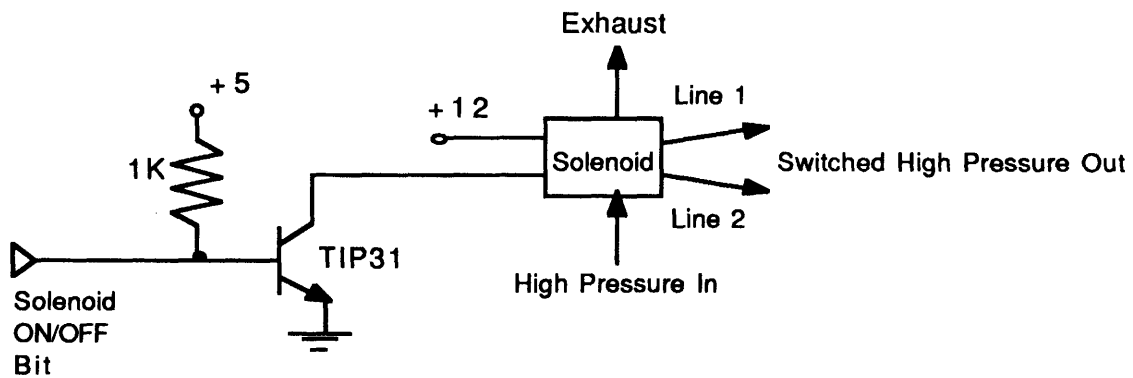


Figure 10a. Solenoid Circuit: Single Channel

24

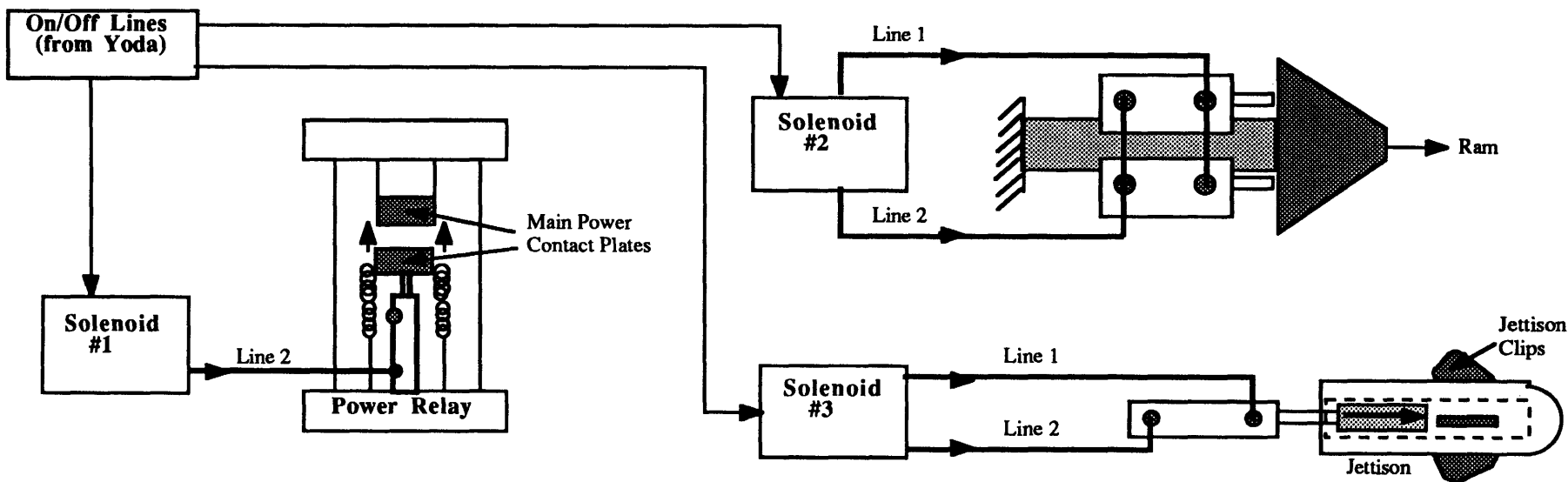


Figure 10b. Pneumatic System Diagram

line is vented, then the excess air escapes the solenoid box through a purge valve.

Mounted on a six-solenoid manifold, the solenoids switch high pressure from one line to another, depending on the current flow through the electrical lines. Figure (10b) shows the functions of the three solenoids currently used by MPOD. The main power relay is driven by one high pressure hose. Nominally, this line is not pressurized. Upon solenoid activation, the metal relay plate is driven up to meet another matching plate. An electrical connection is then made between the positive leads on the main batteries and the MPOD control box +18V lines. If no pressure is supplied to this line, the relay plate is held away from its mate by springs, and no electrical connection is made. The extension of the docking probe is accomplished by two cylinders driven by the same solenoid pressure lines. The default setting (i.e. with no current passing through the solenoid) is the ram retracted position. Finally, the disengagement of the probe's latches is accomplished by a single pneumatic cylinder. No solenoid current flow results in engaged latches, thus allowing target capture before the line is activated.

2.2.3 Motor Control

The motors are activated via a 4-bit magnitude, 1-bit direction command from Yoda, the onboard PC. The actual thrust output by each motor is obtained by pulse-width modulation of the driving voltage, determined by the computer's magnitude outputs. Each motor's direction is controlled by a relay. Shown in Figure (11) is a diagram of the circuit for one motor. The 4-bit magnitude command passes through 74HC85 channel A, and is compared to the constantly counting output of an HC163. Since the twelve motors are organized in pairs, only six independent commands are sent from the computer. After passing through the comparator, these six signals are buffered and split into two lines, one for each motor of a pair. Then, the signal passes through a resistor and transistors before reaching the relay. Each motor direction bit was buffered and split into two lines, as was the magnitude signal. The direction bit triggers the relay, which is connected so that the motor is running either forward or backward, for non-zero magnitudes.

Because of the high current requirement (maximum of 20A per motor), multiple transistors are necessary for TTL signal amplification. Motor power lines are fused in two separate locations to help prevent circuit components from overloading during the inevitable current spikes. Each 11028 power transistor dissipates a significant amount of heat, so the twelve transistors were waterproofed and mounted on a heat sink outside MPOD's control box. The control box internal layout (see Figure (9)) shows the box location of the motor circuitry. Appendix A-4 shows the complete PWM circuit card diagram.

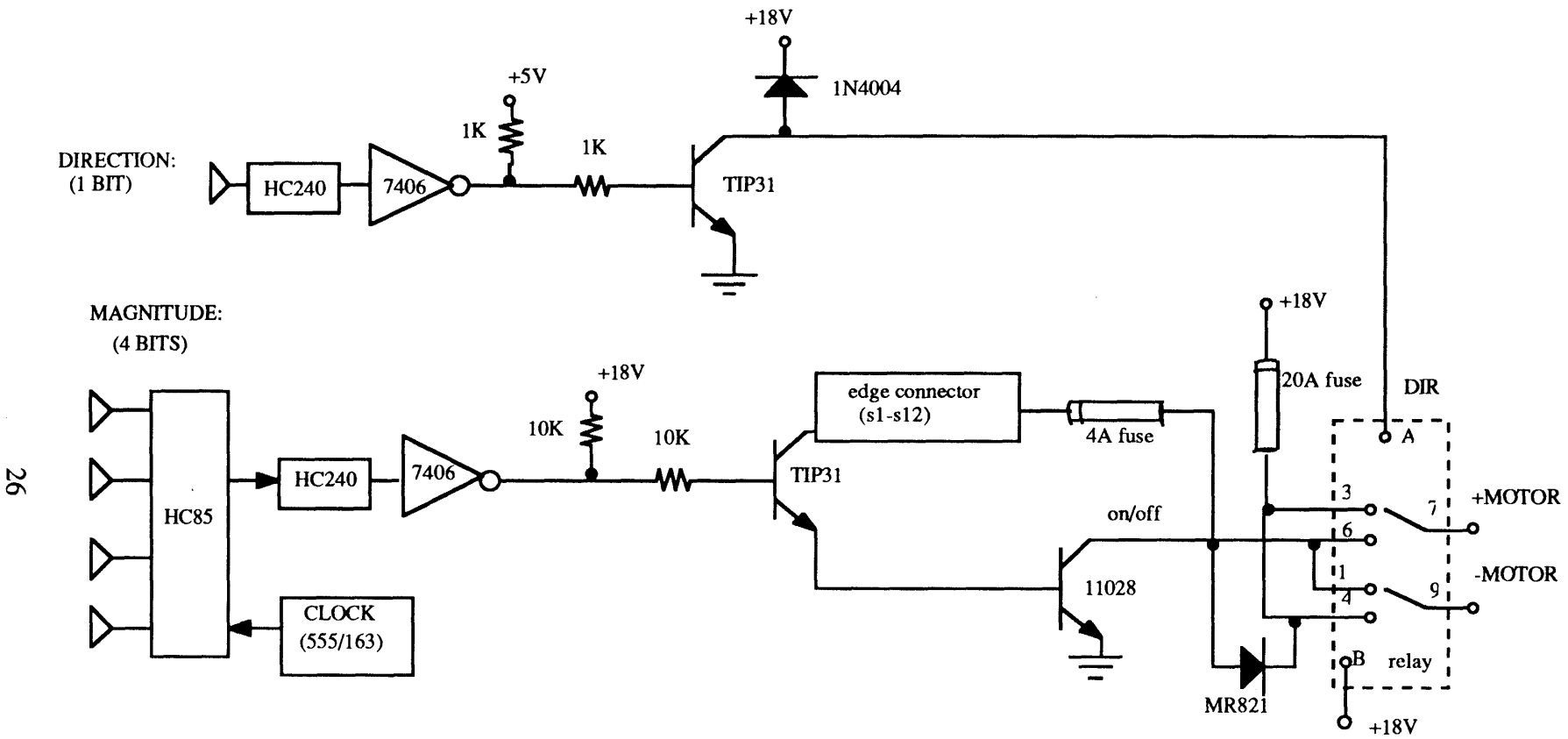


Figure 11. Motor Control Circuit: Single Channel

2.2.4 Pendulum Inclinometers

A pendulum inclinometer consists of an aluminum weight hung from a rod which is then attached to a freely rotating encoder shaft. One such device aligned with each of the MPOD vehicle's axes. Figure (12a) shows a 3-axis arrangement and possible pendulum positions when the vehicle and inertial axes are aligned. The pendula constantly move to follow the gravity vector, which is assumed to be much greater in magnitude than any feasible vehicle acceleration.

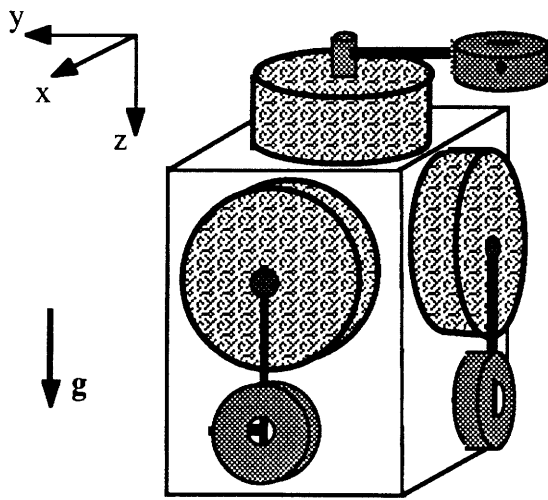


Figure 12a. Pendula Layout

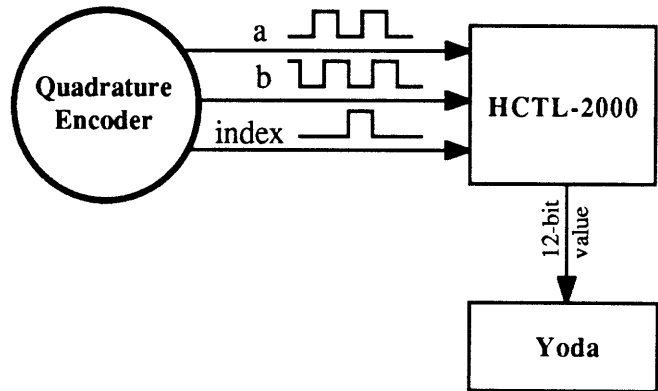


Figure 12b. Encoder Circuit

Pendula positions are determined by BEI Motion Systems quadrature encoders with 12-bit resolution. Figure (12b) shows the basic circuit for one channel. The HCTL-2000 (Reference 9) decodes the a and b pulsing lines, which are 90° out of phase with each other. This phase shift enables the determination of the direction of pendulum motion. Also, an index pulse once per shaft resolution is used to reset the counter in case some counts are missed. At system startup, the HCTL-2000 begins counting from zero. To ensure proper encoder readings after startup, each pendulum must be rotated such that its index pulse is passed. The zero-count for each encoder is then properly initialized.

2.2.5 Depth and 3-axis Rate Sensors

A 12-bit Analog Devices A/D converter and multiplexer combination (Reference 10) is used to read a pressure sensor and 3-axis rate transducer package. The pressure sensor, Omega PX240 Series (Reference 11), is located at approximately the vehicle's center of mass, and is used to determine MPOD's depth in the water. This sensor receives a +15V input signal and emits an analog voltage between 0 and +7V. The Omega PX240 package is capable of measuring depths up to 20 meters in the water. MIT's Alumni Pool is less than five meters deep.

The 3-axis rate transducer package measures MPOD's angular velocity about all three vehicle axes. Driven by +12V input, a fluidic sensor package, Humphrey, Inc., Series RT02, measures rates between $\pm\pi/2$ rad/sec (Reference 12). Its analog output voltage ranges from -5V to +5V for full scale angular velocities.

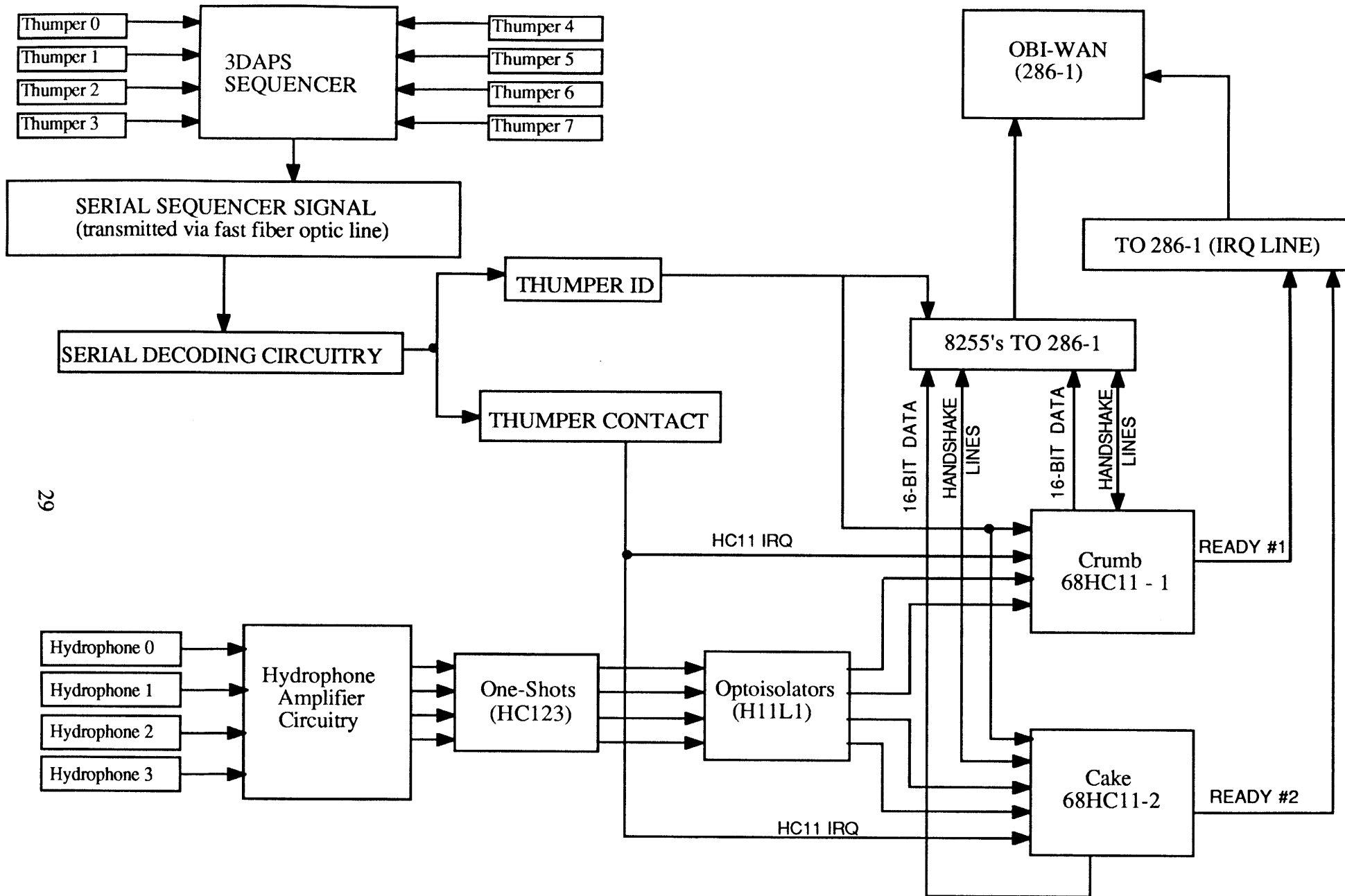
Calibrations and accuracy of the rate and pressure sensors are described in Chapter 3. Because the sensors were converted by the same A/D, its gain and range were required to be compatible with both sensor outputs. Since the lower bit of depth and lower 3-bits of the rate sensors were noise, the maximum input range of the A/D was set at $\pm 10V$.

The A/D and multiplexer system is controlled by Yoda. The process of reading an analog sensor value includes the following steps: (1) output desired multiplexer channel, (2) sample and hold the chosen channel, (3) begin A/D conversion process, (4) upon completion of conversion, read the 12-bit converted value. See "yodafuns.c" in Appendix B-1 for the software implementation of the sensor reading.

2.2.6 3DAPS Receiver Electronics and Microprocessor Software

3DAPS, the 3-Dimensional Acoustic Positioning System, consists of both surface and underwater electronics. Figure (13) shows a complete system diagram of 3DAPS. The sequencer drives and receives signals from the eight thumpers, while the receiver system analyzes hydrophone signals and receives contact and thumper identification signals from the sequencer. The sequencer, thumpers, and hydrophones are identical to those in the systems described in References 2 and 7. However, the receiver system was designed and built to function inside MPOD's control box.

The previous 3DAPS receiver was primarily hardware-based, with individual counters and gates determining thumper-to-hydrophone ranges. In an attempt to simplify circuitry



29

Figure 13. 3DAPS System Diagram

and facilitate system modifications, most of the prior receiver electronics were replaced by 68HC11 microprocessors (Reference 13) programmed in MAX-FORTH. Upon thumper activation, the sequencer sends a serial signal to MPOD. This signal is decoded into a contact signal bit and 3-bit thumper identification number. Appendix A-3 shows the serial decoding circuit, which is contained on the 3DAPS Interface Card inside MPOD's control box. The contact signal bit triggers a hardware interrupt line on the two 68HC11's. Then, the microprocessors begin counting until they receive a hydrophone pulse on a specified counter interrupt line. Note that each 68HC11 handles two of the four acoustic receivers.

Hydrophone signals are amplified by a series of AD521 instrumentation amplifiers, then sent through one-shots to produce a TTL-level pulse. H11L1 Schmitt Trigger optoisolators are used to convert the 3DAPS one-shot signals into MPOD control box signals. This eliminates all electrical connections between the amplifier circuitry and all other MPOD systems. See Appendix A-6 for the complete diagram of the 4-channel hydrophone amplification circuit. Amplifier gains are set such that acoustic interference from other systems is minimized, but all thumpers are received from anywhere in the specified rectangular parallelepiped of flight. 74HC123 pulse lengths are adjusted to prevent post-pulse triggering from acoustic reflections.

After the 68HC11's have received all hydrophone signals or counter rollover has occurred, they interrupt *Obi-Wan* in a declaration of new data. *Obi-Wan* then reads the current thumper's data, and each 68HC11 awaits the next contact signal. See Appendix B-5 for a complete listing of the software used by the 68HC11's, named *Crumb* and *Cake*. The program USMV6811.TXT is used for downloading a program to the 68HC11 for booting and running from the onboard 8Kbyte EPROM (Erasable Programmable ROM). Because a programmed EPROM may not be modified, run-time data is stored on an 8K NOVRAM (Dallas Semiconductor 1225Y non-volatile RAM). A program called INT6811.TXT is stored on the EPROM to perform the 3DAPS counting functions. Each 68HC11 was programmed from RS232 serial lines from an IBM PC. The serial communication program PC-Talk was used on the surface computer for 68HC11 program downloading.

2.2.7 Multiprocessor Interfacing

The 68HC11 microprocessors are interfaced to *Obi-Wan* via 8255A's, arranged in parallel to accommodate *Obi-Wan's* 16-bit data expansion bus. When data is passed between the two systems, handshaking must occur via two bits on an 8255 line. A block

diagram of connections is shown in Figure (14). Appendix A-2 shows the complete interface circuit between *Obi-Wan* and the 68HC11's.

MPOD's three main onboard computers, *Yoda*, *Obi-Wan*, and *Lando*, share common RAM. An Advanced Micro Devices AM2130 Dual Port Ram (DPR) 1 Kbyte device is used for each processor interconnection (Reference 14). Figure (14) also shows the expansion bus connections for the three computers. *Yoda* and *Obi-Wan* are connected via one DPR chip, while *Obi-Wan* and *Lando* are connected via a second chip. For parameter sharing between *Yoda* and *Lando*, *Obi-Wan* must perform a memory transfer. This was an acceptable solution, due to the fact that *Obi-Wan* uses all the *Yoda* and *Lando* shared data in its own control calculations.

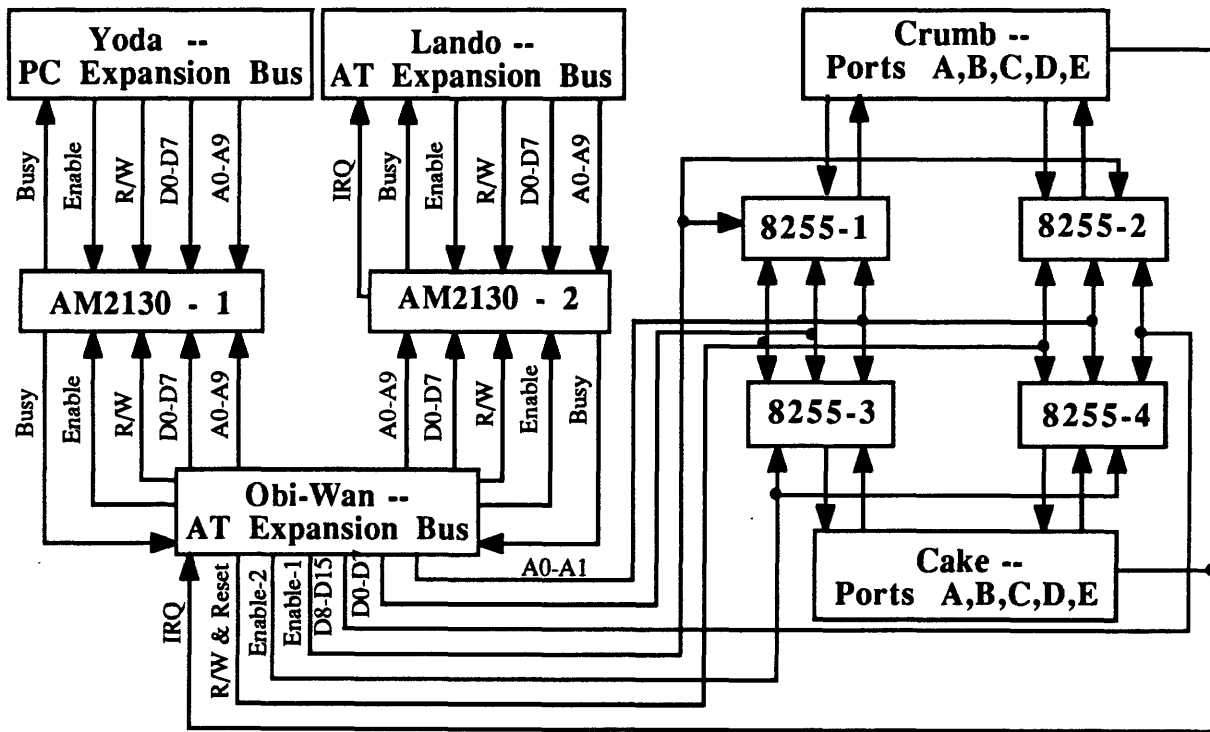


Figure 14. Multiple Processor Interface Diagram

2.2.8 Fiber Optics

Serial communications between MPOD and the surface station are performed through fiber optic cables. This technology allows relatively thin umbilical lines running from the vehicle to the control station. Also, communication problems may be diagnosed by unplugging the cable and examining the light being transmitted through the cables, even

while the vehicle is underwater. A bright transmitted light signal indicates proper cable operation, while constant darkness or dull ambient light may suggest a cut in the cable.

Hewlett-Packard optical sensing chips, connectors, and cables were chosen (Reference 9). Their products were relatively inexpensive and included bulkhead connectors which could easily be mounted on waterproof boxes. The RS232 serial signals travel at the relatively slow rate of 9600 baud, so the optical transmitter/receiver chips chosen for this system were low-speed, high sensitivity devices. It has been noticed that small cuts through the cable insulation do not affect this system.

The 3DAPS sequencer serial signal is transmitted at 250 Kbaud. Hence, the low-speed devices were not sufficient. The required 3DAPS transmission rate meant using high-speed transmitter/receiver chips which, unfortunately, are not sensitive to low level light. Shown in Figure (15) is a diagram of MPOD and surface serial communication connections. Note that the lines from Yoda and Obi-Wan can both connect with the control station computer's (Luke's) COM1 lines. Since Obi-Wan's serial line is used only for data transmission between test runs, the lines may be switched by the surface operator as needed. Also, because the sequencer does not receive data from the 3DAPS receiver system, only one line is needed.

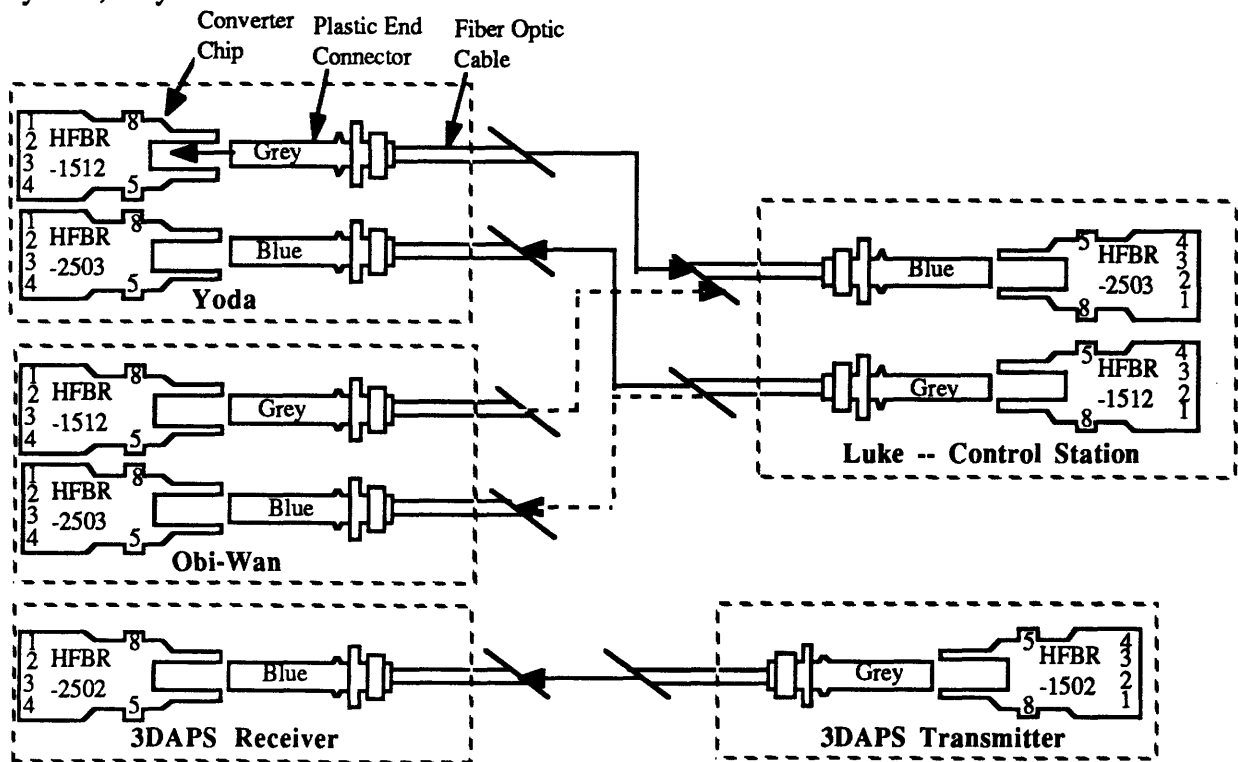


Figure 15. Fiber Optics Connections

For a complete circuit diagram for all five fiber optics channels, see Appendix A-5. The RS232 signals are converted to fiber optic transmitter levels by a +5V to +12V voltage conversion chip, while received signals are converted back to RS232 levels by a +12V to +5V voltage conversion IC. As depicted in Figure (15), the light conversion chips are shaped to accommodate the snap-in fiber optic end connectors.

2.3.0 Computer Systems

MPOD onboard computer processing tasks are divided among three AMPRO single board computers (References 15-18). One processor, *Yoda*, performs communication tasks, both with the surface operator and all MPOD onboard hardware. A second computer, *Obi-Wan*, provides interfacing between the three computers, reads 3DAPS data, computes control outputs, and saves all data. Finally, the third computer in the series, *Lando*, calculates the MPOD state vector from all the available sensor measurements.

The three computers have slightly different components and operate at different clock speeds, but the interfacing and support components are identical for the two 286 boards, and similar for the PC. All the boards run under MS-DOS, and are programmed with Microsoft C, version 5.1. The programs execute within a startup batch file, so that the programs will continuously run without keyboard commands.

Each single-board computer may be attached to a standard IBM monitor and AT keyboard (PC keyboard for *Yoda*). Video signals for each board are provided by a mono/CGA card that attaches directly to each computer's expansion bus. Expansion bus signals are sent through ribbon cables to the MPOD wire-wrapped circuit boards. Because of the CMOS computer components and lengthy ribbon cables required inside MPOD's control box, expansion bus signals are terminated by buffers, except for the low-power dual port RAM.

The AMPRO computers are powered by +5V, with a current draw of ~1A, including video card. With maximum dimensions of 6" x 8" x 1" thick, the boards mount easily within MPOD's control box. Because the computers are constructed from CMOS components, the power and heat dissipation requirements are far lower than those for conventional computers.

2.3.1 *Yoda* the Communications PC

Running at a clock speed of 8MHz, the single-board PC communicates with both the surface pilot and MPOD hardware. *Yoda* software is stored on the 3 1/2", 720Kbyte disk

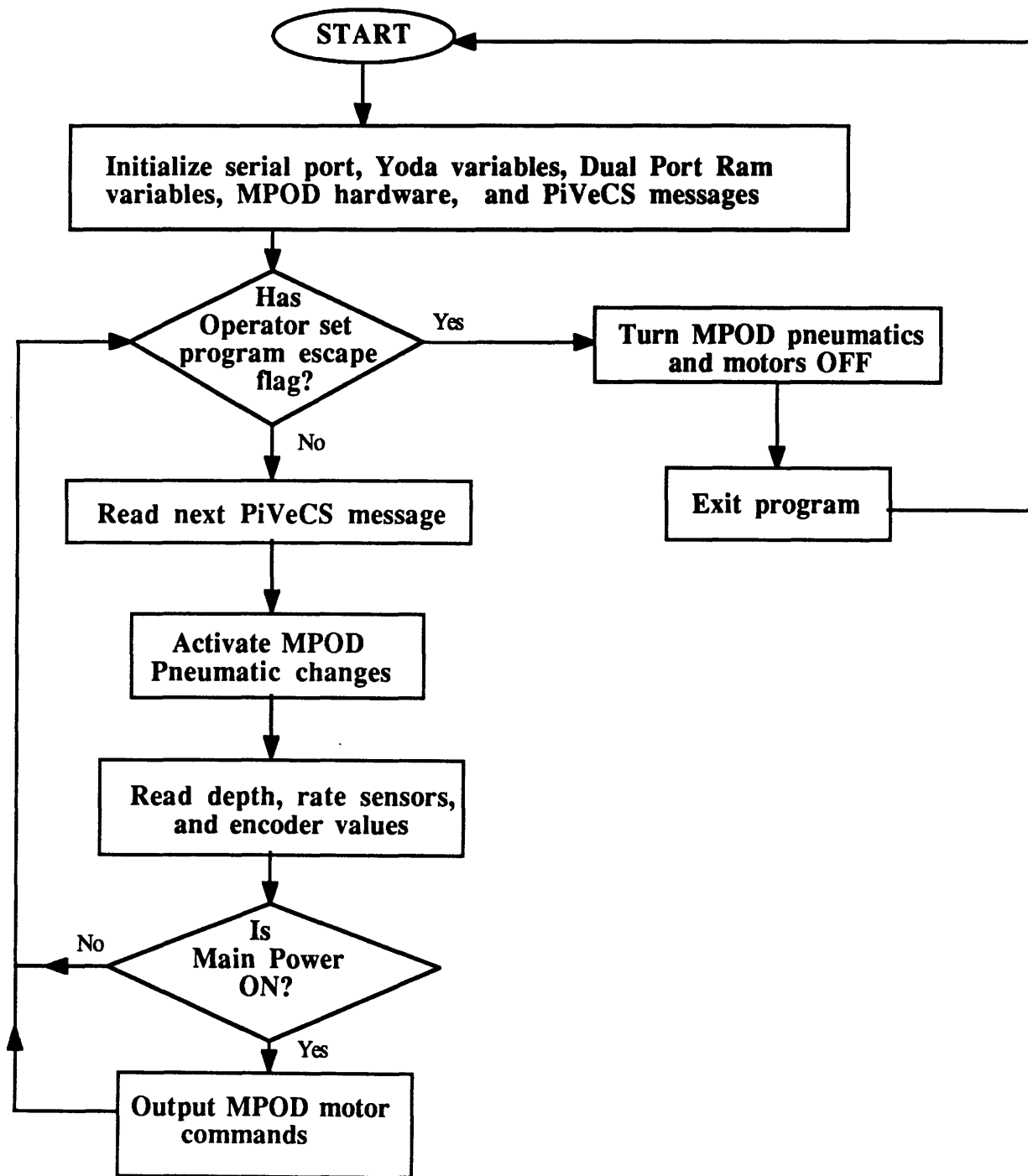


Figure 16. Yoda Software Diagram

drive mounted within MPOD's control box. A relatively slow PC was acceptable for communication tasks, due to the minimal number of mathematical computations and speed limitation of RS232 serial ports. Figure (16) shows the logic diagram of *Yoda* software.

Serial communications are performed at a rate of 9600 baud through the fiber optics lines. A software package known as PiVeCS (Pilot-Vehicle Communication System) was written within the Space Systems Laboratory to enable reliable data exchange between robots and their pilots. Operator commands or data streams are transmitted in "Messages", groups of seven bytes or less. The messages begin with an identification header byte and are sent as requested by the user. The total length of the serial data stream may be varied during run-time by turning on or off a particular message transmission. PiVeCS contains a special function called "ShutDown" which turns off all hardware during periods of communications loss between the surface computer and *Yoda*. This feature is especially useful for the avoidance of a runaway vehicle due to communications problems during flight.

In addition to communicating with the surface operator, *Yoda* performs all the MPOD hardware input/output. Thruster values, calculated by *Obi-Wan*, are sent by *Yoda* to the PWM card. The six motor commands corresponding to MPOD's pairs of motors are sent in four bytes: three magnitude bytes (4-bits per motor pair) and one direction byte (1-bit per motor pair). Pneumatic settings requested by the surface operator are output in one byte. Pendulum encoder values are read by *Yoda* from the HCTL-2000's. Depth and rate transducer data is read by *Yoda* through the A/D converter. All the data is stored in dual port RAM to enable easy access by *Obi-Wan* and *Lando*. A complete listing of all *Yoda* software is shown in Appendix B-1. PiVeCS source code is presented in Reference 19.

2.3.2 *Obi-Wan* the Control 286 Board

Obi-Wan has four primary functions: (1) saving data on NOVRAM, (2) reading 3DAPS ranges, (3) transferring dual port RAM data between *Yoda* and *Lando*, and (4) calculating control outputs from the estimated state vector. Figure (17) shows the complete software diagram for *Obi-Wan*. The 80286-based computer runs at a clock speed of 12MHz, with an 80C287 CMOS math coprocessor. A 512Kbyte Dallas Semiconductor NOVRAM cartridge (DS1217M/4) connects directly with a 25-pin socket on the AMPRO computer board. Because of its speed advantages over a disk drive, all data during MPOD runs is stored on *Obi-Wan's* NOVRAM. After each run's completion, the data is uplinked via Kermit (Reference 20) to *Luke*, the surface control station computer, for

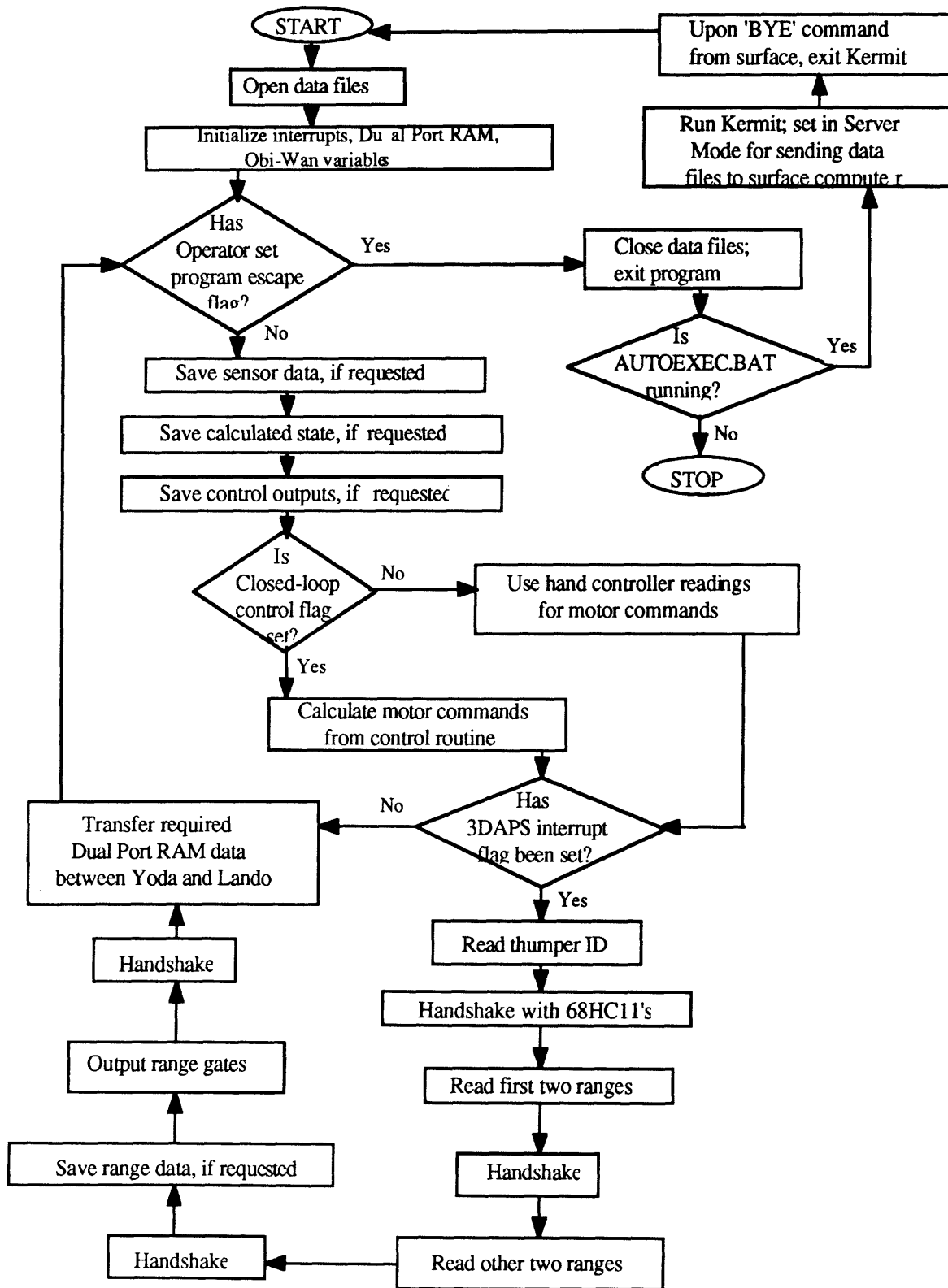


Figure 17. Obi-Wan Software Diagram

storage on its hard disk drive. Data uplinking is necessary due to the limited memory available on the NOVRAM.

3DAPS range reading is triggered by hardware IRQ9 on *Obi-Wan*. The interrupt handler routine sets a flag, then when the main driver program sees the flag, the 3DAPS range reading routine is called. First, the thumper ID is read from the sequencer decoding circuit. Then the ranges are read. Handshaking with the two 68HC11's is necessary because of the limited I/O ports available for the 16-bit range values. Each time a 3DAPS interrupt is generated, four ranges are read. To conserve memory and time, range data is only written to the NOVRAM when new values arrive.

The third function performed by *Obi-Wan* is that of dual port RAM data transfer. During closed-loop runs, the only necessary transfers between *Yoda* and *Lando* should be the pendula, depth, and rate sensor values. Note that when the surface operator wishes to observe the state, the values must be transferred from the *Lando* dual port RAM to *Yoda's* dual port RAM.

The primary calculations performed by *Obi-Wan* involve converting the state estimate into control outputs for the MPOD thrusters. First, *Obi-Wan* determines the desired state, dependent on which part of a control path MPOD is traversing. Next, the control routine determines MPOD motor commands by multiplying the feed-forward linearized state error values by gains. See Chapter 4 for a detailed description of the control algorithms. The complete listing of *Obi-Wan* software is shown in Appendix B-2.

2.3.3 *Lando* the State Calculation 286 Board

The sole purpose of *Lando* is to estimate the current position, attitude, and rates of the MPOD vehicle. Figure (18) shows a logic diagram of the *Lando* software. *Lando* is an 80286-based computer running at a clock speed of 16MHz. Like *Obi-Wan*, *Lando* has an 80C287 coprocessor and runs from a 512Kbyte NOVRAM cartridge. 3DAPS ranges and the depth sensor are used for position determination. The pendulum encoders and 3DAPS provide attitude measurements, while the rate sensors directly measure vehicle angular velocity about each of the three axes. Unfortunately, no direct measurement of linear velocity is available. An extended Kalman filtering routine with state propagation is used for the estimation process. Chapter 4 describes in detail the implementation of this state calculation algorithm.

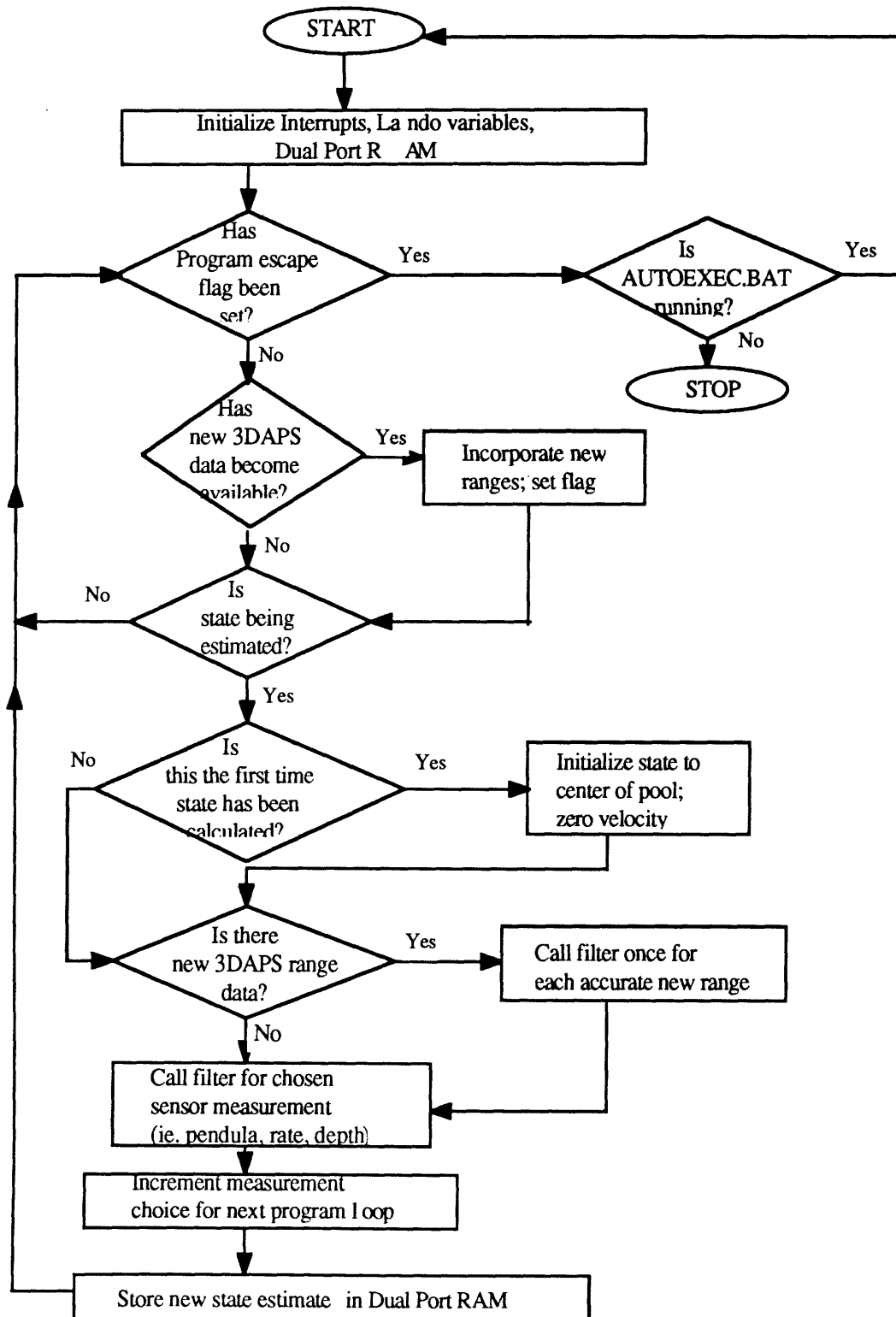


Figure 18. Lando Software Diagram

The filter is called with one measurement at a time. Since 3DAPS ranges are available relatively infrequently, Lando hardware interrupt IRQ9 is activated by Obi-Wan upon receipt of new 3DAPS data. Lando only calls the filter for range measurements when its interrupt line has been triggered and the new ranges are non-zero. Since new pendula, rate, and depth measurements are available as frequently as Lando updates the state for one measurement, Lando cycles through these sensor readings in order, except when interrupted with 3DAPS data.

State calculation may be turned "off" and "on", and the state estimate may be re-initialized by the surface operator. Because the filter initially guesses that MPOD is at the center of the 3DAPS thumper parallelopiped, the state calculation routines should be turned "on" when MPOD is near that location.

2.4.0 Surface Control Station

For MPOD human factors testing, an attempt was made to make a remote control station as similar as possible to the onboard cockpit. However, the tests performed in this thesis had no human factors component. Therefore, the control station was designed to be easy to assemble and centered around the fastest available IBM-compatible computer. This computer was used for developing all the MPOD onboard programs as well as the control station software.

2.4.1 Pilot Interface and Hardware Description

An operator flies MPOD with two hand controllers and the surface computer keyboard. During open loop flight, the left hand controls MPOD translational motion, while the right hand dictates rotational maneuvers. The keyboard is used for controlling MPOD pneumatics, commanding data transmission, and turning on and off the various calculations being performed on MPOD.

The operator receives two types of feedback from MPOD: (1) Black and white video from MPOD's camera, and (2) the computer display of PiVeCS, the current switch settings, and uplinked data values. Shown in Figure (19) is a diagram of the surface control station used for the MPOD experiments.

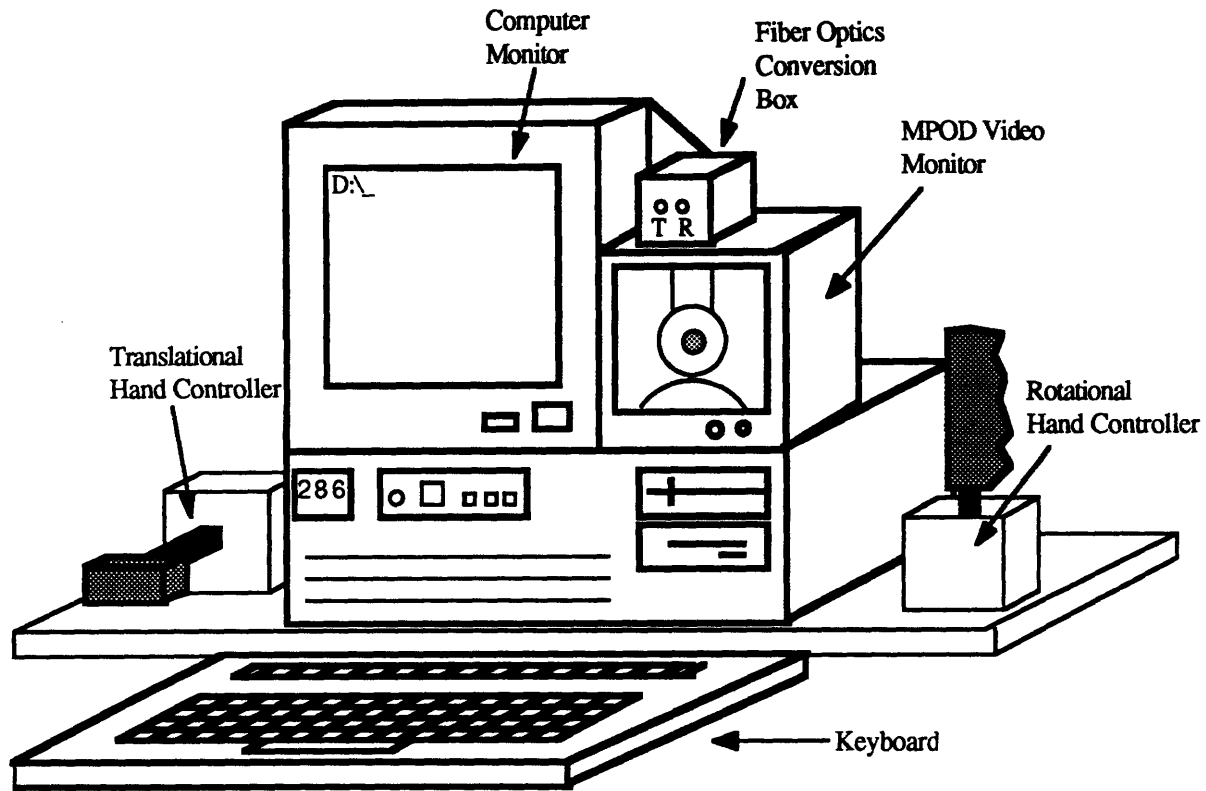


Figure 19. Control Station Layout

The computer used for these experiments was a 10MHz 286-clone, hereafter referred to as *Luke*. *Luke* has a 40Mbyte hard disk, one 5 1/4" floppy drive, and a 3 1/2" 720K disk drive. An EGA monitor and video card are used. One RS232 serial port is connected to the external fiber optics conversion box for serial transmissions to and from MPOD.

A wire-wrapped circuit card, attached to *Luke's* data bus, connects the hand controllers with the rest of the system. See Appendix A-7 for the card's circuit diagram. The hand controllers used are "bang-bang", meaning they provide on/off signals but no variable magnitude. They use magnets and redundant magnetic field sensors to produce the electrical signals sent to *Luke*. For a more detailed description, see Reference 6. Note that these hand controllers are waterproof and are also used in MPOD's cockpit during onboard flight control. Figure (20) shows a complete diagram of the surface control station components and their functions.

2.4.2 Software

During underwater testing, *Luke* is used to interface with a surface operator and store data between MPOD runs. Besides communicating with *Yoda*, *Luke* displays switch

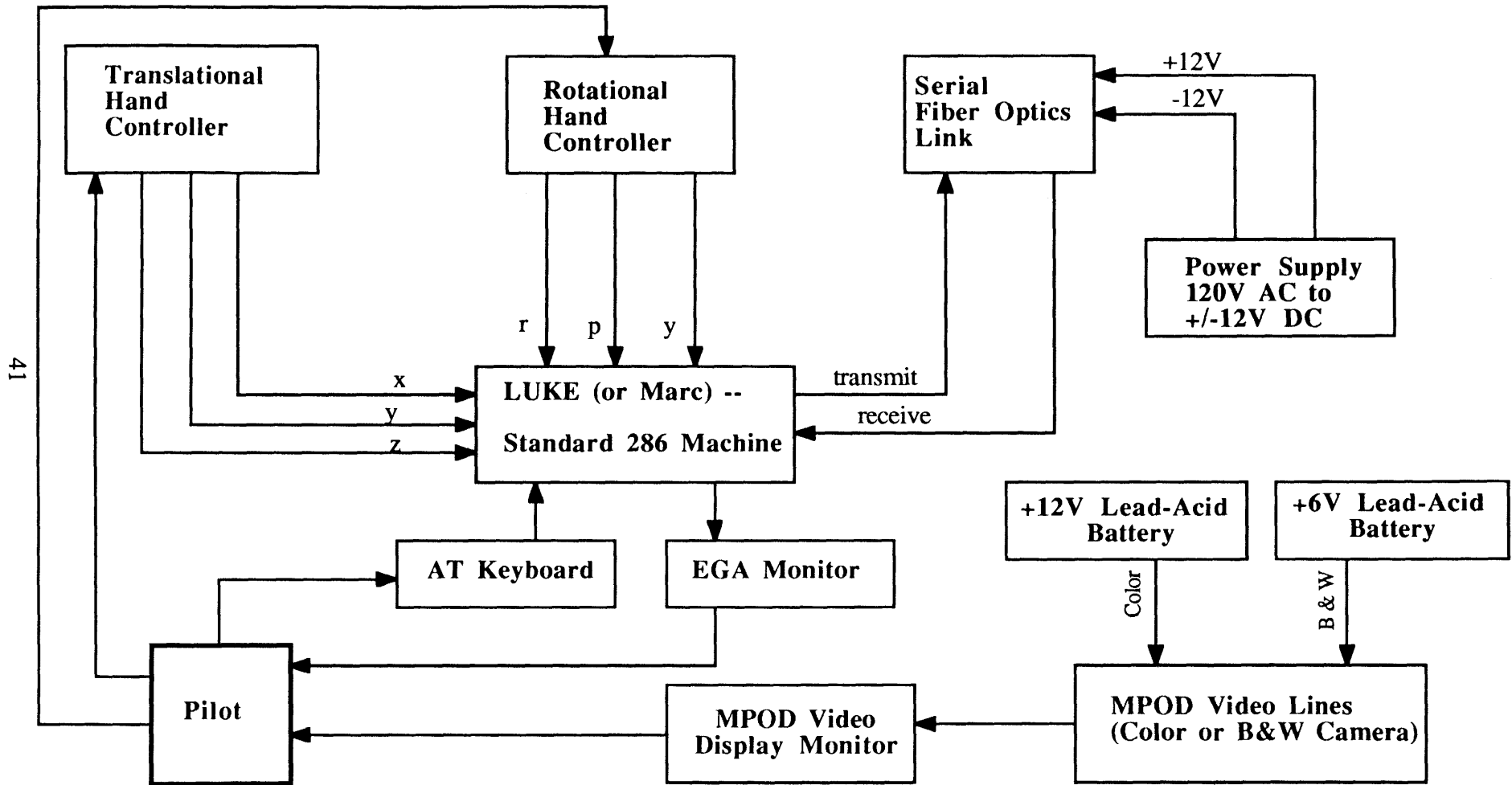


Figure 21. Control Station Functional Diagram

settings and MPOD data on its monitor. The computer constantly looks for operator commands from the keyboard and hand controllers. Figure (21) shows the keyboard layout of commands a pilot may initiate. Figure (22) is a block diagram of the control station software logic. Appendix B-4 shows a complete source code listing of the *Luke* software.

Serial communications with *Yoda* are performed using the PiVeCS protocols. Messages are passed between *Luke* and *Yoda* via COM1, running at a rate of 9600 baud. The operator controls the quantity of serial data transmission through *Luke's* keyboard. During many debugging situations, it is important for an operator to have the ability to see certain sensor readings, the calculated state, and/or control outputs. However, viewing of MPOD data at the surface significantly slows the loop times of *Yoda* and *Luke*. Also, a large data stream delays serial exchange of important parameters, such as hand controller commands. Therefore, during closed-loop control runs, the operator should view only a minimal number of the data parameters.

A pilot may use *Luke's* display to judge the performance of MPOD in real-time. Besides printing the requested vehicle data, the monitor also provides the user with communication and message passing status. In addition to serial port handling, PiVeCS provides a graphic display atop *Luke's* monitor which enables the pilot to constantly view the status of communications and the current message being transmitted or received.

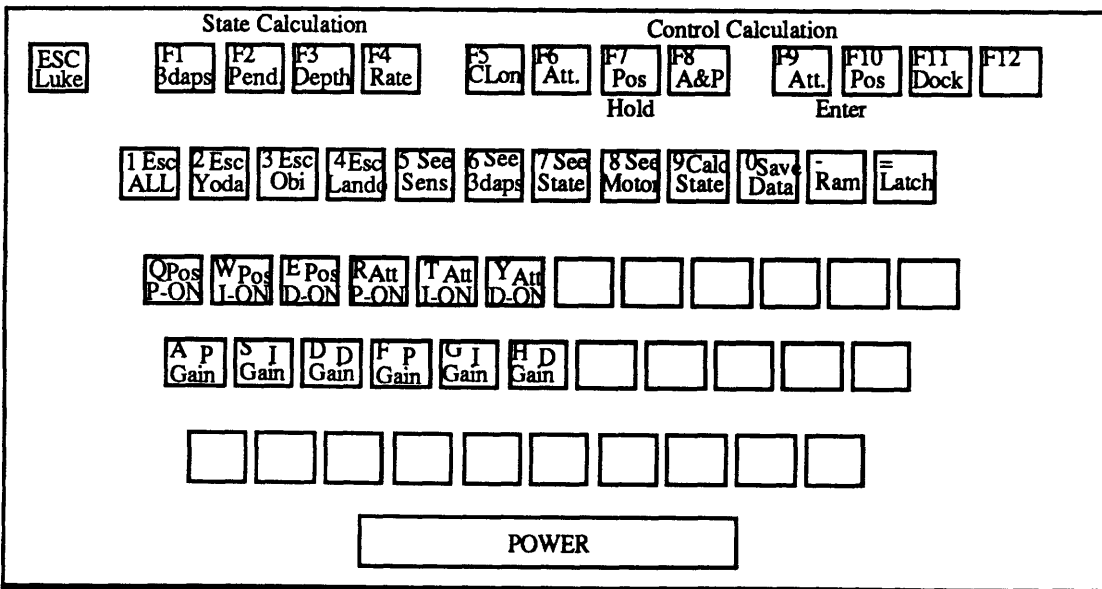


Figure 21. Luke Keyboard Functions

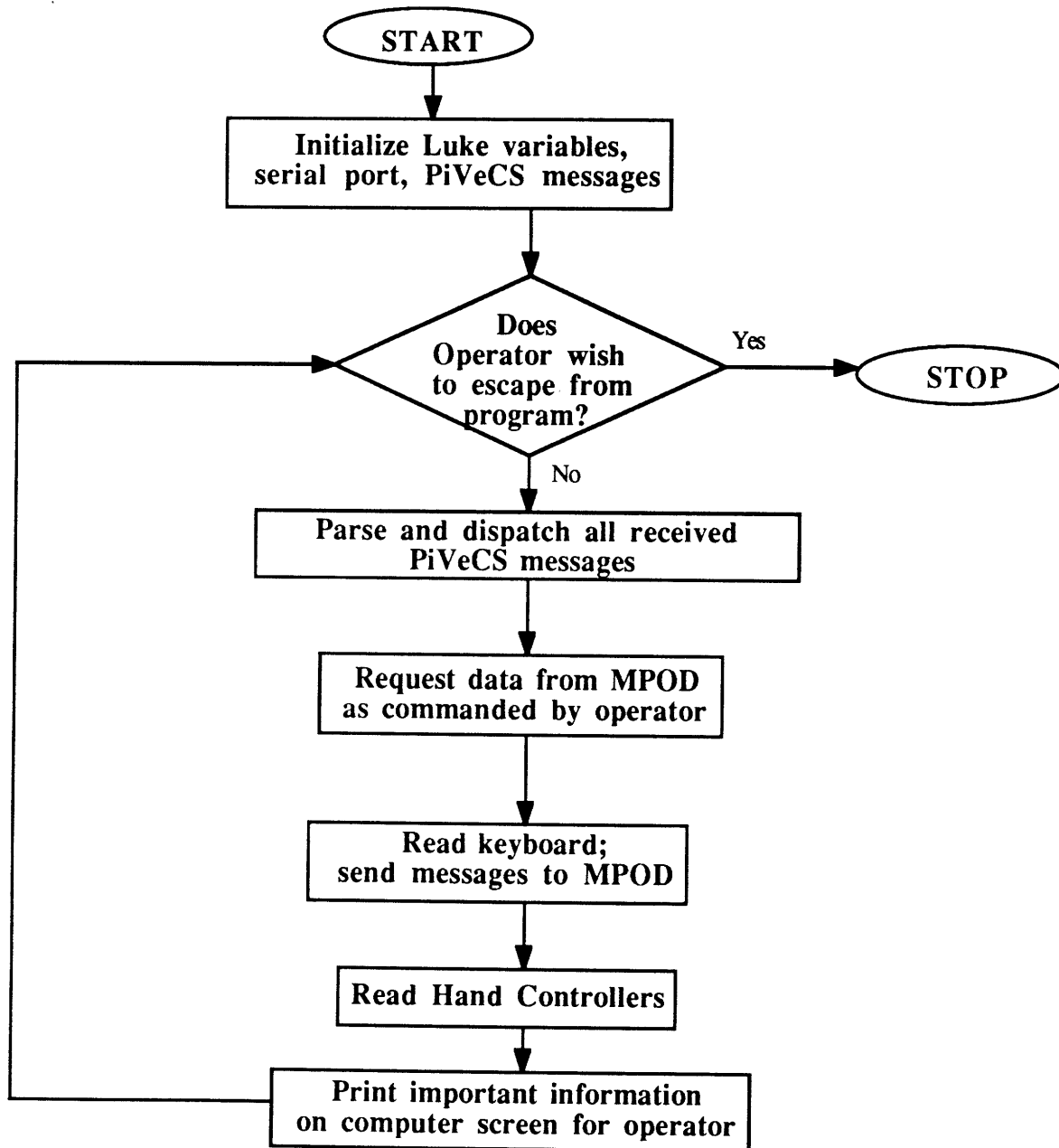


Figure 22. Luke Software Diagram

3.0 MPOD Dynamics and Physical Parameters

Before the implementation of a control system, sensors must be calibrated, relevant vehicle physical parameters determined, and an estimated state vector formed using these values and the available sensor measurements. All equations and calculations were standardized with SI units. This chapter describes the vehicle equations of motion, MPOD physical parameters, and results from the sensor calibration tests.

3.1.0 MPOD Equations of Motion

MPOD travels through the water in 3 dimensions with no constraining tether, and has no significant dynamic modes during free-flight. Therefore, MPOD may be treated as a rigid body moving through a viscous fluid. During vehicle motion, hydrodynamic drag significantly affects MPOD's behavior, both in translation and rotation. Because the drag is proportional to the square of velocity, both the translational and rotational equations of motion are nonlinear. MPOD's motors provide the only known linear forces and torques.

Unmodelled effects on MPOD include buoyancy offsets from the vehicle's center and water currents produced by MPOD motors, divers, or pool water jets. Because MPOD is balanced each time it enters the water, a buoyancy term in the equations of motion would be a function of balancing success by the divers. MPOD is usually balanced within 0.5 kg of neutral (i.e. one 1-lb. balancing weight), thus buoyancy offsets may be assumed negligible and considered as constant disturbances in the control system. Water currents may also cause significant MPOD motion, but they are unknown in direction and strength. Hence, they cannot be adequately modelled in the equations of motion and were therefore ignored..

3.1.1 Rigid Body Translation

MPOD locomotive force is provided by twelve motors. Linear acceleration is a function of the instantaneous forcing and vehicle linear velocity. The equations of motion along the inertial translational axes are shown in Equations (3.1-1).

$$\begin{aligned}\ddot{x} &= \frac{1}{m} (F_x - C_{dt} |\dot{x}| \dot{x}) \\ \ddot{y} &= \frac{1}{m} (F_y - C_{dt} |\dot{y}| \dot{y}) \\ \ddot{z} &= \frac{1}{m} (F_z - C_{dt} |\dot{z}| \dot{z})\end{aligned}\quad (3.1-1)$$

where x, y, z = inertial translation state variables, C_{dt} = the MPOD translational drag coefficient, F_n = external forcing about the n inertial axis, and m = total apparent vehicle mass. These equations determine the location of the MPOD vehicle's center with respect to the inertial coordinate system. The mass term includes water that is accelerated with the vehicle. The external force, F_n , is produced by MPOD's thrusters. Later in Chapter 3, it is determined that the drag coefficients are the same about each axis.

Because the coordinates x , y , and z are in inertial space, each forcing term F_n is a function of both MPOD motor commands and vehicle attitude. The twelve motors arranged in pairs along the vehicle axes are the control actuators. Linear force in body coordinates is provided by two motor pairs aligned with each of the vehicle x , y , and z axes. The direction cosine matrix, $[C]$, is used to transform the motor forces along MPOD body coordinates into linear force values along the inertial x , y , and z axes. In Equation (3.1-2), $x_{n,i}$ and $x_{n,b}$ are positions along the inertial and body n -axes, respectively. The $F_{n,i}$ and $F_{n,b}$ in Equation (3.1-3) are forces along the inertial and body n -axes, respectively.

$$\begin{Bmatrix} x_{x,i} \\ x_{y,i} \\ x_{z,i} \end{Bmatrix} = [C] \begin{Bmatrix} x_{x,b} \\ x_{y,b} \\ x_{z,b} \end{Bmatrix} = \begin{bmatrix} c_{11} & c_{12} & c_{13} \\ c_{21} & c_{22} & c_{23} \\ c_{31} & c_{32} & c_{33} \end{bmatrix} \begin{Bmatrix} x_{x,b} \\ x_{y,b} \\ x_{z,b} \end{Bmatrix} \quad (3.1-2)$$

$$\begin{Bmatrix} F_{x,i} \\ F_{y,i} \\ F_{z,i} \end{Bmatrix} = \begin{bmatrix} c_{11} & c_{12} & c_{13} \\ c_{21} & c_{22} & c_{23} \\ c_{31} & c_{32} & c_{33} \end{bmatrix} \begin{Bmatrix} F_{x,b} \\ F_{y,b} \\ F_{z,b} \end{Bmatrix} \quad (3.1-3)$$

3.1.2 Rigid Body Rotation

Vehicle rotation is described in a similar manner. Torques due to motor thrust and hydrodynamic drag dominate the equations of motion. The important difference between

the rotation and translation equations is axial cross-coupling due to the moment of inertia matrix, $[I]$. The moment of inertia matrix is approximated as diagonal, although there are inevitably small off-diagonal terms which may be considered as "noise" in the equations of motion. From calculations described in Section 3.2 and Appendix C.1, it is shown that the diagonal terms of the $[I]$ matrix are not equal. The equations of motion for MPOD's 3-axis rigid body rotation are hence given by Equations (3.1-4).

$$\begin{aligned}\dot{\omega}_x &= \frac{1}{I_{xx}} (T_x + (I_{yy} - I_{zz}) \omega_y \omega_z - C_{drx} |\omega_x| \omega_x) \\ \dot{\omega}_y &= \frac{1}{I_{yy}} (T_y + (I_{zz} - I_{xx}) \omega_x \omega_z - C_{dry} |\omega_y| \omega_y) \\ \dot{\omega}_z &= \frac{1}{I_{zz}} (T_z + (I_{xx} - I_{yy}) \omega_x \omega_y - C_{drz} |\omega_z| \omega_z) \quad (3.1-4)\end{aligned}$$

where I_{nn} = moment of inertia about the n axis, T_n = motor torque about the n axis, C_{dm} = rotational coefficient of drag about the n body axis, and ω_n = angular velocity about the n body axis.

A method of describing attitude is in terms of Euler angles, roll (ϕ), pitch (θ), and yaw (ψ). Using this method, the final attitude is dependent on the order of command execution. For this thesis, the standard aircraft system was used in which the Euler angles describe the following sequence of maneuvers: (1) yaw, (2), pitch, then (3) roll. If the maneuvers were performed in any other sequence, a different final vehicle attitude would result.

Relating changes in the angular velocities to changes in the Euler angles results in a system of coupled nonlinear equations populated with trigonometric terms. A great simplification can be achieved if the attitude is instead expressed in the quaternion system. Expressed in this manner, the evolution of the attitude is still a set of coupled nonlinear ODE's, but involving only arithmetic computations. Reference 21 contains a complete discussion of quaternions and their applications. The four quaternion elements uniquely describe the three-dimensional attitude of an object. They consist of three coordinates describing an axis in 3-D space and a fourth describing a rotation about that axis, as shown in Equation (3.1-5).

$$\vec{q} = q_0 + \bar{q} = q_0 + q_1\mathbf{i} + q_2\mathbf{j} + q_3\mathbf{k} \quad (3.1-5)$$

In terms of the Euler angles ϕ , θ , and ψ , the quaternion vector is defined by:

$$\begin{aligned} q_0 &= \cos \frac{\theta}{2} \cos \frac{1}{2}(\psi + \phi) \\ q_1 &= \sin \frac{\theta}{2} \cos \frac{1}{2}(\psi - \phi) \\ q_2 &= \sin \frac{\theta}{2} \sin \frac{1}{2}(\psi - \phi) \\ q_3 &= \cos \frac{\theta}{2} \sin \frac{1}{2}(\psi + \phi) \end{aligned} \quad (3.1-6)$$

The derivatives of the quaternion vector in terms of the vehicle angular velocities, ω_n , are:

$$\begin{aligned} \dot{q}_0 &= -\frac{1}{2}(\omega_x q_1 + \omega_y q_2 + \omega_z q_3) \\ \dot{q}_1 &= \frac{1}{2}(\omega_x q_0 - \omega_y q_3 + \omega_z q_2) \\ \dot{q}_2 &= \frac{1}{2}(\omega_x q_3 + \omega_y q_0 - \omega_z q_1) \\ \dot{q}_3 &= \frac{1}{2}(-\omega_x q_2 + \omega_y q_1 + \omega_z q_0) \end{aligned} \quad (3.1-7)$$

For all state and most control calculations, the quaternion coordinate system is used. The direction cosine matrix, $[C]$, is calculated from the quaternion estimate using the following conversion matrix:

$$[C] = \begin{bmatrix} c_{11} & c_{12} & c_{13} \\ c_{21} & c_{22} & c_{23} \\ c_{31} & c_{32} & c_{33} \end{bmatrix} = \begin{bmatrix} 1 - 2(q_2^2 + q_3^2) & 2(q_1 q_2 - q_0 q_3) & 2(q_1 q_3 + q_0 q_2) \\ 2(q_1 q_2 + q_0 q_3) & 1 - 2(q_1^2 + q_3^2) & 2(q_2 q_3 - q_0 q_1) \\ 2(q_1 q_3 - q_0 q_2) & 2(q_2 q_3 + q_0 q_1) & 1 - 2(q_1^2 + q_2^2) \end{bmatrix} \quad (3.1-7)$$

3.2.0 MPOD Physical Constants

3.2.1 Apparent Mass

For an underwater object, the dynamic equations are best modeled by the use of an "apparent" mass, not the actual mass of the object. In Reference 22, it was shown that the apparent mass of a body moving in water is approximately twice the actual mass of the body. This increase in apparent mass is due to induced water velocity around the moving object.

Upon construction, MPOD's dry mass was determined to be approximately 1100 lbs, or 500 kg (Reference 30). This value may have changed slightly with the addition of the new control box. However, due to the inability to reassess the MPOD vehicle's mass, the approximate value of 500 kg is assumed still valid. Multiplying the actual vehicle mass by a factor of two, the apparent vehicle mass is:

$$m = 1000 \text{ kg} \quad (3.2-1)$$

3.2.2 Moments of Inertia

The moment of inertia matrix, $[I]$, is assumed to be diagonal. This simplifies the equations of motion considerably, and is a reasonable approximation given MPOD's symmetric properties. It would further simplify the equations to assume that all diagonal elements of $[I]$ were equal; however, calculations show that this is not the case.

The most accurate way of calculating MPOD's moments of inertia would be to create a finite element model of MPOD, then compute each element's moment of inertia with respect to the vehicle axes. Due to limited time and computer resources, the $[I]$ values for MPOD were determined using "lumped masses". Major MPOD components were measured, weighed, then modelled as boxes or cylinders. Components used in moment of inertia calculations included: main battery boxes (2), control battery box, 80 ft³ air tanks (2), 50 ft³ air tank, control box, motors (12), docking probe, and solid aluminum panels. The remaining MPOD frame mass was modelled as a spherical shell of radius 0.5 m to simplify moment of inertia calculations. Appendix C-1 shows the calculation breakdown. The composite results used in the MPOD equations of motion are:

$$[I] = \begin{bmatrix} I_{xx} & 0 & 0 \\ 0 & I_{yy} & 0 \\ 0 & 0 & I_{zz} \end{bmatrix} = \begin{bmatrix} 80.5 & 0 & 0 \\ 0 & 85.9 & 0 \\ 0 & 0 & 94.1 \end{bmatrix} \text{ (kg-m}^2\text{)} \quad (3.2-2)$$

3.2.3 Maximum Thrust and Torque

The twelve MPOD motors each provide an equal amount of thrust. Although the propellers were designed to provide peak thrust at a non-zero velocity, only static tests could completely separate MPOD thrusters from dynamic effects. The maximum output of the thrusters was determined by attaching a spring scale to the MPOD vehicle, then measuring the full-scale force output along each vehicle axis. Each group of four motors produce a maximum thrust of 120 N, or 30 N/motor. The maximum torque was calculated

by multiplying the maximum thrust of the four motors by the moment arm from the vehicle center to the MPOD motor location. This moment arm was measured as 0.85 m about each axis. The maximum thrust and torque available for each axis are shown in Equation (3.2-3).

$$F_{\max} = 120 \text{ N}$$

$$T_{\max} = F_{\max} (0.85 \text{ m}) = 102 \text{ N-m} \quad (3.2-3)$$

3.2.4 Translational Coefficients of Drag

MPOD translational coefficients of drag were experimentally determined during terminal velocity tests. Shortly after a constant thrust command was initiated, the vehicle reached a steady state in which the acceleration approached zero. Then, the only terms remaining in translational equations of motion (3.1-1) were the known forcing and drag. It was experimentally determined that the maximum translational velocity along each of MPOD's axes is approximately 0.5 m/sec. Because all axes also have the same maximum thrust value, all translational coefficients of drag are equal. Thus the single translational coefficient of drag, C_{dt} , is given by Equation (3.2-4).

$$C_{dt} = \frac{|F_{\max}|}{v_{\max}^2} = \frac{120}{(0.5)^2} = 480.0 \quad \frac{\text{kg}}{\text{m}} \quad (3.2-4)$$

3.2.5 Rotational Coefficients of Drag

MPOD rotational drag coefficients were also evaluated with terminal velocity tests. Analogous with the translation experiments, a constant torque command produced a rotational steady state in which the angular acceleration was zero. Because the tests were done one axis at a time, the cross-coupling velocity term in each equation also vanished. The maximum thrust terminal velocities about the x, y, and z axes are 0.74, 0.62, and 0.62 rad/sec, respectively. The resulting drag coefficients are:

$$C_{d_{rx}} = \frac{|T_{\max}|}{\omega_{x\max}^2} = \frac{102}{(0.74)^2} = 186.3 \quad \text{kg-m}^2$$

$$C_{d_{ry}} = C_{d_{rz}} = \frac{|T_{\max}|}{\omega_{y\max}^2} = \frac{102}{(0.62)^2} = 265.3 \quad \text{kg-m}^2 \quad (3.2-5)$$

3.3.0 Sensor Feedback -- Calibration and Accuracy

3.3.1 3-axis Rate Transducer Package

The rate sensors, as described in Section 2.2, measure angular velocities between $\pm \pi/2$ radians/sec. These rates correspond with rate sensor voltage outputs of $\pm 5V$ and A/D readings ranging from 1024 - 3071, with a theoretical null reading of 2048. However, through all tests, the zero reading of the rate sensors was not consistently 2048, as shown in Appendix C-2. The expected offsets were countered in the state calculation software listed in Appendix B-3.

Rate sensor calibration was determined by dividing the maximum A/D reading by the maximum rate sensor reading. Timed terminal velocity tests agreed with the theoretical calibration. The following calibration factor to SI units was used in the MPOD software:

$$\text{Rate Factor} = \frac{\text{Max A/D count} - \text{A/D zero}}{\text{Maximum Angular Velocity}} = \frac{(3096 - 2048)}{\frac{\pi}{2}} = 651.9 \frac{\text{counts}}{\left(\frac{\text{rad}}{\text{sec}}\right)} \quad (3.3-1)$$

The rate sensor noise was within a two-bit count, or 0.006 rad/sec. See Appendix C-2 for static test results showing rate sensor null offset and noise.

3.3.2 Depth Sensor

The depth sensor was calibrated by taking MPOD to known water depths and calculating the corresponding sensor readings. The depth sensor has a maximum output of approximately +8V at a depth of 20 or more meters. Because the same 12-bit A/D converter was used for both the angular rate (0- $\pm 5V$) and depth (0-+8V) measurements, the full-scale readings were set at $\pm 10V$. The zero depth value of (2430 ± 2) counts was determined by the surface depth sensor reading. The conversion factor from these experiments is calculated in Equation (3.3-2).

$$\text{Depth Factor} = \frac{\text{Depth A/D Count} - \text{Zero depth A/D count}}{\text{Depth in Meters}} = 60.2 \frac{\text{counts}}{\text{m}} \quad (3.3-2)$$

The depth sensor reading was accurate to two bits, or 0.066 m. See Appendix C-2 for the depth sensor standard deviation during static testing with MPOD on various areas of the pool floor.

3.3.3 Pendulum Inclinometers

The pendulum encoders and HCTL-2000's provide a 12-bit value which ranges from 0 to 4095, as described in Section 2.2. This encoder count corresponds to angles of 0 - 2π radians, thus the calibration factor is given by:

$$\text{Pendula Factor} = \frac{\text{12-bit HCTL-2000 Count}}{2\pi \text{ radians per revolution}} = 651.9 \frac{\text{counts}}{\text{radian}} \quad (3.3-3)$$

Pendulum inaccuracies are produced by dynamic modes of the swinging pendulum weights during MPOD motion. These pendulum modes have a maximum magnitude no greater than four bits, or (0.024 radians), as determined by studying dynamic data.

3.3.4 3DAPS Ranges

The range calibration equation depends on the speed of sound in water and the delay between thumper contact and counter initialization. Experimental calibration is the most practical method for precisely determining these parameters. To perform the calibration, MPOD was weighted to the bottom of the pool, then a series of range data was collected. Next, divers manually measured the distances for each thumper-hydrophone combination (8 thumpers x 4 hydrophones).

After the experiments were completed, averages and standard deviations of 3DAPS-determined counts for each thumper-hydrophone combination at each static location were determined. The average ranges were plotted against the measured ranges. Next, a linear curve fit was performed to determine the offset and range factor for the conversion from counts to meters. Equation (3.3-4) shows the calibration equation used in the state calculation software. Figure (23) shows the composite range calibration plot for three different MPOD locations. The standard deviation of the ranges averaged 42.7 counts, or 0.032 meters.

$$\text{Range (m)} = 1.011 + (0.000748) \text{Range (counts)} \quad (3.3-4)$$

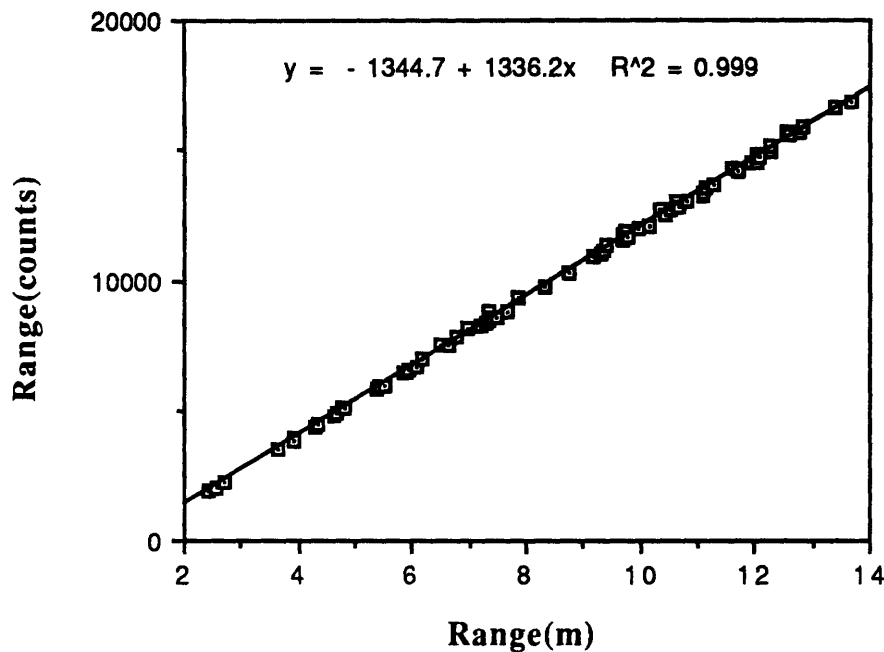


Figure 23. Static Range Calibration Plot

4.0 The Control System

The goal for the current MPOD multiprocessor system is automated docking from any initial starting point. A fully operational control system would include: 3-axis position hold, 3-axis attitude hold, and inertial trajectory following. Any combination position and attitude maneuver should be possible. The controller should be sufficiently robust to counter slight MPOD buoyancy offsets and water currents, which may be considered disturbances by the control system.

This thesis does not present a thoroughly tested and proven controller. However, algorithms and MATLAB linearized control results are presented for possible future implementation. With gain adjustments, it is expected that the current algorithms are capable of performing many of the control system goals, from station-keeping to automated flight.

Before the implementation of a control system, sensor measurements must be used to estimate vehicle position, attitude, and velocity. This chapter first describes the state estimation equations and their implementation. Next, a position and attitude hold control system is developed by linearizing the equations of motion about a set point. Finally, the

full nonlinear dynamic model is incorporated into a feed-forward linearization control scheme for large position and attitude maneuvers.

4.1.0 State Calculation

4.1.1 State Vector Elements and their Derivatives

In order to fully examine the dynamic properties, MPOD's translational and rotational position and velocity must be estimated. Also, the unpredictable null offset in the rate sensor readings, (\hat{b}_n) , must be estimated for each measurement sequence. Equation (4.1-1) shows the components of the 16-dimensional estimated state vector.

$$\begin{aligned} \vec{y} &= [y_1 \ y_2 \ \dots \ y_{16}]^T \\ &= [x \ y \ z \ q_0 \ q_1 \ q_2 \ q_3 \ v_x \ v_y \ v_z \ \omega_x \ \omega_y \ \omega_z \ b_x \ b_y \ b_z]^T \end{aligned} \quad (4.1-1)$$

A new estimate of this state vector is determined each time the filter incorporates a new measurement. Because the equations of motion are integrated during the state estimation process, the derivatives of all state vector elements must be calculated. These equations were described in Chapter 3. Equation (4.1-2) shows the complete state derivative vector.

$$\vec{\dot{y}} = \begin{pmatrix} \vec{\dot{x}} \\ \dot{q}_0 \\ \vec{\dot{q}} \\ \vec{\dot{v}} \\ \vec{\dot{\omega}} \\ \vec{\dot{b}} \end{pmatrix} = \begin{pmatrix} \vec{v} \\ \frac{1}{2} (-\vec{q}^T \vec{\omega}) \\ q_0 \vec{\omega} + \vec{q} \times \vec{\omega} \\ \frac{1}{m} (\vec{F} - [c_{dt}] (\vec{v}^2) \text{sgn}(\vec{v})) \\ [\mathbf{I}]^{-1} (\vec{T} - [c_{dr}] (\vec{\omega}^2) \text{sgn}(\vec{\omega}) - \vec{\omega} \times [\mathbf{I}] \vec{\omega}) \\ 0 \end{pmatrix} \quad (4.1-2)$$

4.1.2 Extended Kalman Filter Equations

Reference 23 describes in detail the derivation and implementation of the extended Kalman filter equations for MPOD. Equations (4.1-3) show the extended Kalman filter equations used in the MPOD state calculation software.

$$\begin{aligned} [\mathbf{P}]^+ &= ([\mathbf{I}] - \vec{\mathbf{k}}_i \vec{\mathbf{h}}_i^T) [\mathbf{P}]^- \\ \vec{\mathbf{y}}^+ &= \vec{\mathbf{y}}^- + \vec{\mathbf{k}}_i (g_i - z_i) \end{aligned} \quad (4.1-3)$$

where

$[\mathbf{P}]$ = error covariance matrix

g_i = i^{th} expected measurement

z_i = i^{th} actual measurement

$\vec{\mathbf{y}}$ = State vector estimate

$\vec{\mathbf{h}}_i = \frac{\partial g_i}{\partial \vec{\mathbf{y}}} =$ Sensitivity of i^{th} measurement to variations in $\vec{\mathbf{y}}$

$\vec{\mathbf{k}}_i = \frac{1}{a} [\mathbf{P}]^- \vec{\mathbf{h}}_i =$ Measurement gain

where $a = \sigma^2 + \vec{\mathbf{h}}_i^T [\mathbf{P}]^- \vec{\mathbf{h}}_i$ and

$\sigma^2 =$ variance of sensor i output

+/- superscripts = quantities after/before measurement incorporation

These equations are updated upon receipt of each new measurement, z_i . MPOD filter software is listed in Appendix B.3, "filter.c". The filter update rate during MPOD runs is approximately 50 measurements per second.

4.1.3 Measurement Incorporation and State Propagation

Each of the measurements gives information about a combination of state vector elements. To predict MPOD's 39 sensor measurements, correlation between these measurements and state vector elements must be determined. Equations (4.1-4) show the relationship between the thirty-two 3DAPS range readings and state vector elements.

$$g_{ij} = (\vec{d}_{ij})^T \vec{d}_{ij} = (r_{ij})^2$$

$$\vec{d}_{ij} = \vec{x} + [C] \vec{l}_i - \vec{t}_j \quad (4.1-4)$$

where r_{ij} = range value from hydrophone i to thumper j , \vec{d}_{ij} = range vector from hydrophone i to thumper j , $[C]$ = direction cosine matrix defined in Chapter 3, \vec{l}_i = hydrophone i position vector with respect to MPOD body coordinates, and \vec{t}_j = thumper j position vector with respect to inertial coordinates. Note that \vec{l}_i and \vec{t}_j are constant vectors.

The pendula produce the most complex combinations of state vector measurements. Because they measure only the gravity vector, the inertial yaw reading cannot be determined with pendulum feedback. The following equations relate the three pendulum measurements to the inertial space. Note the loss of yaw angle sign information due to taking the cosine of pendulum angles. To restore the yaw angle sign information, the pitch angle sign is incorporated into the measurement. Equation (4.1-5) shows the state parameters relating to the pendulum measurements.

$$g_\phi = \frac{c_{33}}{\sqrt{(c_{32})^2 + (c_{33})^2}} = \cos(\phi), \quad g_\theta = \frac{c_{33}}{\sqrt{(c_{31})^2 + (c_{33})^2}} = \cos(\theta),$$

$$g_\psi = \frac{-\text{sgn}(g_\theta) c_{31}}{\sqrt{(c_{31})^2 + (c_{32})^2}} = \cos(\psi) \quad (4.1-5)$$

where the c_{ij} are elements of the direction cosine matrix.

The three-axis rate sensors proportionally affect only the angular velocity (ω) elements of the state vector, while the depth sensor provides a direct inertial z - coordinate measurement. Equations (4.1-6) show the correlation between measurements and the corresponding state vector elements.

$$\vec{g}_\omega = \vec{\omega} + \vec{b} = \begin{pmatrix} \omega_{x, meas} \\ \omega_{y, meas} \\ \omega_{z, meas} \end{pmatrix}, \quad g_z = z = (\text{depth}) \quad (4.1-6)$$

In order to determine, at each iteration, how to adjust the current state vector estimate, it is necessary to evaluate the effect on the expected measurements of first order perturbations in the estimated state. For this reason, it is necessary to calculate all measurement

derivatives with respect to the estimated state vector elements, $\left(\frac{\partial \vec{g}}{\partial \vec{y}}\right)$. Equation (4.1-7) shows the elements of this measurement derivative matrix. Because of the complexity of the 624 results, they will not be presented in this thesis. Reference 23 shows the complete set of derivatives used in the MPOD filter calculations.

$$\frac{\partial \vec{g}}{\partial \vec{y}} = \begin{bmatrix} \frac{\partial \vec{g}_r}{\partial \vec{x}} & \frac{\partial \vec{g}_r}{\partial \vec{q}} & \frac{\partial \vec{g}_r}{\partial \vec{v}} & \frac{\partial \vec{g}_r}{\partial \vec{\omega}} & \frac{\partial \vec{g}_r}{\partial \vec{b}} \\ \frac{\partial \vec{g}_p}{\partial \vec{x}} & \frac{\partial \vec{g}_p}{\partial \vec{q}} & \frac{\partial \vec{g}_p}{\partial \vec{v}} & \frac{\partial \vec{g}_p}{\partial \vec{\omega}} & \frac{\partial \vec{g}_p}{\partial \vec{b}} \\ \frac{\partial \vec{g}_z}{\partial \vec{x}} & \frac{\partial \vec{g}_z}{\partial \vec{q}} & \frac{\partial \vec{g}_z}{\partial \vec{v}} & \frac{\partial \vec{g}_z}{\partial \vec{\omega}} & \frac{\partial \vec{g}_z}{\partial \vec{b}} \\ \frac{\partial \vec{g}_\omega}{\partial \vec{x}} & \frac{\partial \vec{g}_\omega}{\partial \vec{q}} & \frac{\partial \vec{g}_\omega}{\partial \vec{v}} & \frac{\partial \vec{g}_\omega}{\partial \vec{\omega}} & \frac{\partial \vec{g}_\omega}{\partial \vec{b}} \end{bmatrix} \quad (4.1-7)$$

where \vec{g}_r = range measurement vector, \vec{g}_p = pendula measurement vector, \vec{g}_z = depth sensor measurement vector, and \vec{g}_ω = rate sensor measurements.

In addition to incorporating measurements during the state estimation process, MPOD equations of motion must also be integrated. Estimates of vehicle positions, velocities, and applied forces are used to calculate vehicle acceleration components. See Equations (4.1-2) for the state vector derivative components, \hat{y}_i . After calculating all the state vector derivatives, the \hat{y}_i equations are integrated using a second-order Runge-Kutta algorithm. The integration procedure uses an intermediate estimate $\hat{y}(t + \Delta t / 2)$ between the times t and $(t + \Delta t)$. See Reference 23 for details. Appendix B.3 shows the MPOD state propagation software in "pstate.c".

4.2.0 Position and Attitude Hold

4.2.1 Linearized Equations of Motion

Because more theory is available for linear control systems analysis, an attempt was made to linearize MPOD equations of motion whenever possible. For small attitude or

position changes, the second order terms may be ignored, thus producing linear equations about which a linear PID control system may be implemented.

MPOD maneuvers may be considered small during station-keeping, or set point regulation. If the desired position is given by $\vec{x}_{n,0}$ and the current position is \vec{x}_n , then position perturbations from the nominal value are given by $(\vec{x}_{n,0} - \vec{x}_n)$. The linear velocity is given by $\delta\vec{v}$. For attitude, small perturbations about a set point may easily be represented by the use of Euler angles instead of quaternions. The conversion from quaternions to the Euler angles is given by:

$$\begin{aligned}\theta &= -\arcsin(c_{13}) = -\arcsin(2*(q_1q_3 - q_2q_0)) \\ \psi &= -\arctan2\left(\frac{c_{23}}{\cos(\theta)}, \frac{c_{33}}{\cos(\theta)}\right) = -\arctan2\left(\frac{2(q_2q_3 + q_0q_1)}{\cos(\theta)}, \frac{1 - 2(q_1^2 + q_2^2)}{\cos(\theta)}\right) \\ \phi &= -\arctan2\left(\frac{c_{12}}{\cos(\theta)}, \frac{c_{11}}{\cos(\theta)}\right) = -\arctan2\left(\frac{2(q_1q_2 + q_3q_0)}{\cos(\theta)}, \frac{1 - 2(q_2^2 + q_3^2)}{\cos(\theta)}\right)\end{aligned}\quad (4.2-1)$$

Under the small perturbation assumption, second order drag and cross-coupled inertia terms in the equations of motion may be neglected, such that both the translational and rotational equations described in Chapter 3 become double integrators. The state-space form of linear equations of motion is given by:

$$\dot{\vec{x}} = [\mathbf{A}] \vec{x} + [\mathbf{B}] \vec{u} \quad (4.2-2)$$

The MPOD linearized equations of motion are:

$$\begin{pmatrix} \delta\dot{\hat{x}}_n \\ \delta\ddot{\hat{x}}_n \end{pmatrix} = \begin{bmatrix} 0 & 1 \\ 0 & 0 \end{bmatrix} \begin{pmatrix} (x_{n,0} - \hat{x}_n) \\ \delta\dot{\hat{x}}_n \end{pmatrix} + \begin{bmatrix} 0 \\ \frac{1}{m} \end{bmatrix} F_n \quad (4.2-3a)$$

$$\begin{pmatrix} \delta\dot{\hat{\omega}}_n \\ \delta\ddot{\hat{\Phi}}_n \end{pmatrix} = \begin{bmatrix} 0 & 1 \\ 0 & 0 \end{bmatrix} \begin{pmatrix} (\Phi_{n,0} - \hat{\Phi}_n) \\ \delta\dot{\hat{\omega}}_n \end{pmatrix} + \begin{bmatrix} 0 \\ \frac{1}{I_m} \end{bmatrix} T_n \quad (4.2-3b)$$

where Equations (4.2-3a) represent translation and (4.2-3b) rotations. For convenience, the matrices in (4.2-3a) will be denoted by $[A_x]$ and $[B_x]$ while the matrices in (4.2-3b) will be $[A_\phi]$ and $[B_\phi]$.

4.2.2 Proportional-Integral-Derivative (PID) Control Structure

For a double integrator system, the actuator outputs are determined by multiplying position and velocity estimate errors with "proportional" and "derivative" controller gains, respectively. To prevent steady-state errors, the position is integrated with respect to time, then multiplied by a gain to provide the "integral" term of the controller.

Because each position loop and each attitude loop is completely independent, the PID controller was designed for a single-axis system, then the individual gains were applied to all axes of the multivariable system. A complete position control diagram is shown in Figure (24a). For position regulation, the set point, \vec{x}_0 , is initialized upon control system activation. Errors from the nominal position value are fed into the PID system. Note that the control system only receives the estimated state determined from sensor measurements.

The detailed PID position diagram is shown in Figure (24b). The proportional gain corrects the current position error, while the derivative gain is used to damp non-zero velocities. The integral gain is used to keep steady-state errors in line. With a nonzero position error, the integral terms grow with each time step, inducing MPOD to return to its set point position. For software implementation, a maximum limit was placed on the integral feedback term, due to the possibility of an unstable system.

During attitude regulation, MPOD must point in a constant direction with respect to the inertial coordinate system. Figure (25a) shows the complete attitude system diagram. As with position, the initial attitude upon control system initiation is used as $\vec{\Phi}_0$, the set point. Then errors between the estimated attitude, $\vec{\Phi}$, and set point attitude are fed to the PID controller, which is depicted by Figure (25b).

The structure of the open-loop MPOD system is shown in Equations (4.2-4), where $T(s)$ is the PID open loop transfer function, where the K_n are gains to be determined.

$$T(s) = \frac{1}{s^2} \left(\frac{K_I}{s} + K_P + K_D s \right) = \frac{(K_I + K_P s + K_D s^2)}{s^3} \quad (4.2-4)$$

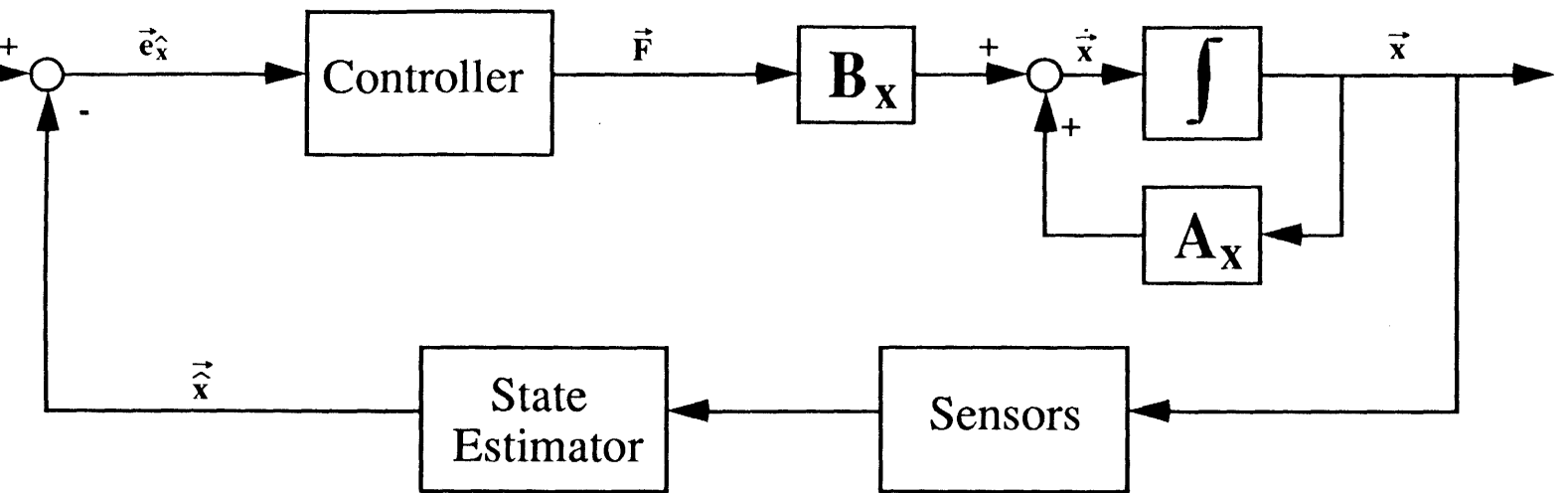


Figure 24a. Position Hold Block Diagram

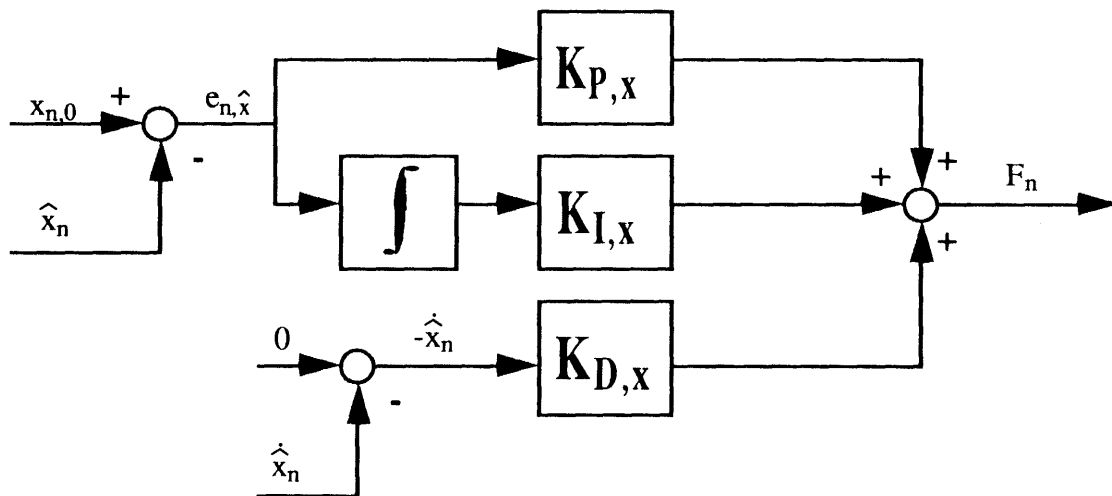


Figure 24b. Position Hold Control Diagram

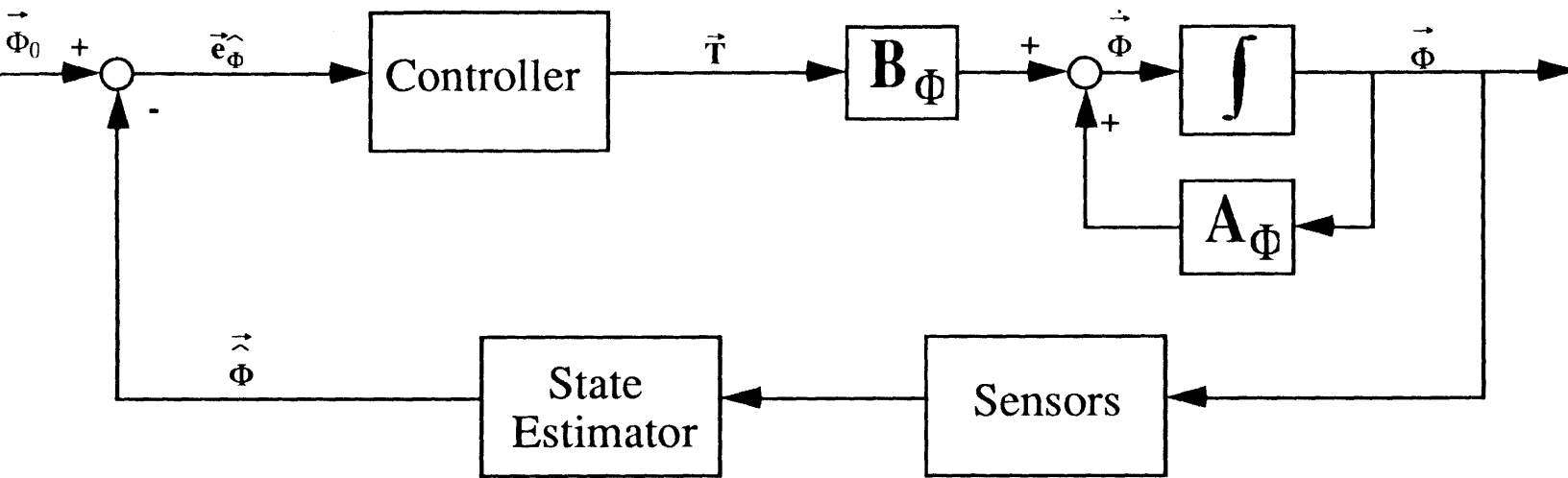


Figure 25a. Attitude Hold Block Diagram

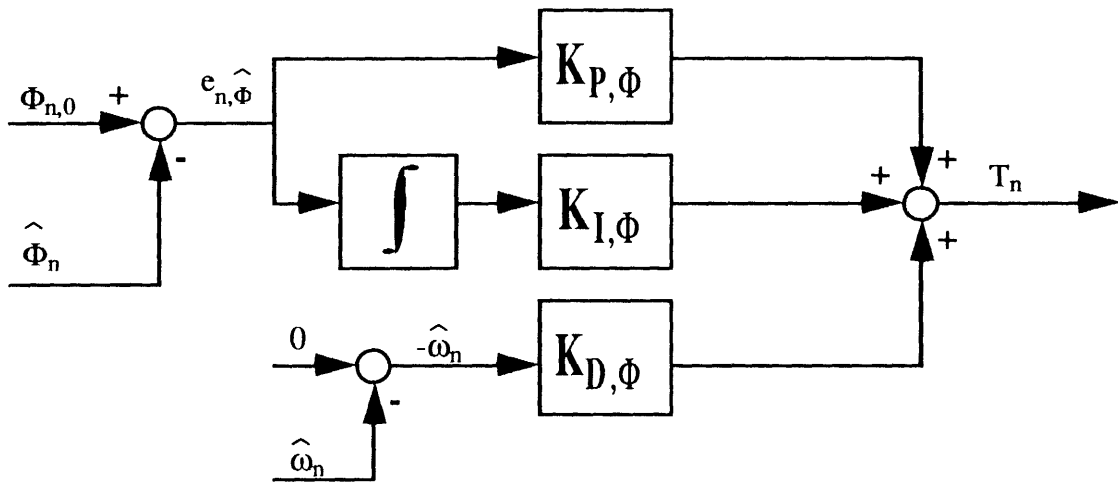


Figure 25b. Attitude Hold Control Diagram

Note that for certain gains, the system may contain unstable right half-plane poles. The PID closed loop poles and zeros of a double integrator dynamic system are given by the transfer function, $C(s)$, shown in Equation (4.2-5). The next section describes one method by which control system gains may be determined. The results are only an example of many possible methods for gain calculation, and should be considered accordingly.

$$C(s) = \frac{T(s)}{1 + T(s)} = \frac{(s^2 K_D + s K_P + K_I)}{(s^3 + s^2 K_D + s K_P + K_I)} \quad (4.2-5)$$

4.2.3 LQ-Servo with Integrator Results

The Linear Quadratic Servo problem (Reference 24) provides an optimal stable solution for the gains of a linear PID control problem. The solution relies on the minimization of a cost function, J , which is described in Equation (4.2- 6).

$$J = \int_0^{\infty} (\vec{y}^T [\mathbf{Q}] \vec{y} + \vec{u}^T [\mathbf{R}] \vec{u}) dt \quad (4.2- 6)$$

The matrices $[\mathbf{Q}]$ and $[\mathbf{R}]$ are to a certain extent arbitrary. For simplicity, the $[\mathbf{Q}]$ matrix was chosen to be the identity matrix. Because MPOD has only one forcing input per equation, $[\mathbf{R}]$ was one-dimensional, hereafter labelled ρ .

To incorporate the integrator into the state-space equations, the state vector is augmented. Equations (4.2- 7a) and (4.2- 7b) show the position and attitude augmented state equations for control gain analysis. Note the addition of two new variables: w , the position integral variable, and a , the attitude integral variable.

$$\begin{pmatrix} \dot{w} \\ \delta \dot{x} \\ \delta \ddot{x} \end{pmatrix} = \begin{bmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ 0 & 0 & 0 \end{bmatrix} \begin{pmatrix} w \\ (x_{n,0} - \hat{x}_n) \\ \delta \dot{x} \end{pmatrix} + \begin{bmatrix} 0 \\ 0 \\ \frac{1}{m} \end{bmatrix} F_n \quad (4.2- 7a)$$

$$\begin{pmatrix} \dot{a} \\ \delta\omega \\ \delta\dot{\omega} \end{pmatrix} = \begin{bmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ 0 & 0 & 0 \end{bmatrix} \begin{pmatrix} a \\ (\Phi_{n,0} - \hat{\Phi}_n) \\ \delta\omega \end{pmatrix} + \begin{bmatrix} 0 \\ 0 \\ \frac{1}{I_m} \end{bmatrix} T_n \quad (4.2- 7b)$$

For a closed loop LQ-Servo system, the forcing is determined by the estimated state feedback and desired set point. The control and closed-loop state-space equations are:

$$\begin{aligned} \vec{u} &= - [\mathbf{K}] (\vec{x} - \vec{x}_{set}) \\ \dot{\vec{x}} &= ([\mathbf{A}] - [\mathbf{B}] [\mathbf{K}]) \vec{x} + [\mathbf{K}] \vec{x}_{set} \end{aligned} \quad (4.2- 8)$$

Note that \vec{x}_{set} in the above equations replaces the x_0 and Φ_0 elements of \vec{x} in Equations (4.2-7).

The LQ-Servo gain matrix is calculated with the Control Algebraic Ricatti Equation (CARE). Equation (4.2- 9a) shows the CARE. The solution of CARE, $[\mathbf{L}]$, is used for determination of the control system gain matrix, $[\mathbf{K}] = [K_I \quad K_P \quad K_D]$. Equations (4.2-9b) shows solution for $[\mathbf{K}]$.

$$- [\mathbf{L}] [\mathbf{A}] - [\mathbf{A}]^T [\mathbf{L}] - [\mathbf{Q}] + [\mathbf{L}] [\mathbf{B}] [\mathbf{R}]^{-1} [\mathbf{B}]^T [\mathbf{L}] = 0. \quad (4.2- 9a)$$

$$[\mathbf{K}] = [\mathbf{R}]^{-1} [\mathbf{B}]^T [\mathbf{L}] \quad (4.2- 9b)$$

Appendix E.1 shows the position LQ-servo results. The Macintosh MATLAB program was used for all control system analysis. For the "cheap" control problem with $\rho = 0.0001$, optimal position gains were determined to be:

$$[\mathbf{K}_x] = [K_{I,x} \quad K_{P,x} \quad K_{D,x}] = [100.0 \quad 447.5 \quad 951.3] \frac{N}{m} \quad (4.2- 10)$$

The limiting factor in control system gains was the maximum available actuator output. The value of ρ was chosen such that the thrusters would saturate when MPOD was a distance of 0.3 m from the desired set point. The system gains were tested by simulations of the linearized equations with an initial position offset of 0.3 m from the desired set point, then with a constant disturbance of magnitude 10 N, to represent water currents. Resulting plots are shown in Appendix E.1. The response time of the system, due to control gains

limited by available control authority, was approximately 10 seconds. Overshoot was 50%. This value is quite large, indicating the need to introduce more system damping.

Appendix E.2 shows the attitude LQ-servo results. Note that this sample single-axis LQ-Servo used the average of the three moment of inertia values for gain determination. For $\rho = 0.0001$, optimal rotation set point gains were determined to be:

$$[\mathbf{K}_\phi] = [K_{I,\phi} \ K_{P,\phi} \ K_{D,\phi}] = [100.0 \ 242.0 \ 242.8] \ \text{N-m} \quad (4.2-11)$$

As with position, the value of ρ , and hence the system gains, were limited by thruster power. Gains were chosen such that the thrusters would saturate at 0.5 radians from the set point. The system gains were tested with an initial attitude offset of 0.5 radians, then with a constant disturbance of magnitude 10 N-m which represents a buoyancy offset. Note that this disturbance value is much larger than would be expected in real applications. Plots of the tests are shown in Appendix E.2. The response time was approximately 5 seconds, half that of the position response. The maximum overshoot is 25%. As with the position response, this high value should be countered with more system damping.

4.3.0 Automated Maneuvers

4.3.1 Feed-Forward Linearization Structure

When performing large magnitude position and attitude maneuvers, velocities are not small, so the equations of motion may not be linearized as in Section 4.2. A control scheme known as feed-forward linearization has been developed to enable nonlinear dynamic systems to be controlled as if they were linear. Reference 25 describes the rationalization behind this linearization procedure.

The main purpose of feed-forward linearization is the cancellation of nonlinear dynamics by feeding forward these terms through the control output. For position, the only nonlinear term in the equations of motion is drag, which is proportional to the square of velocity. Because the state estimator provides a value for the linear velocity, the drag may be estimated. By subtracting the drag term from the control force, the net drag force entering the MPOD plant is zero. Thus the system behaves as if drag were not present.

Figure (26a) shows the position feed-forward control diagram. Note the addition of drag to the plant dynamics. The control block of Figure (26a) is expanded in Figure (26b). Except for gain values, the PID portion of the controller is identical to that described in

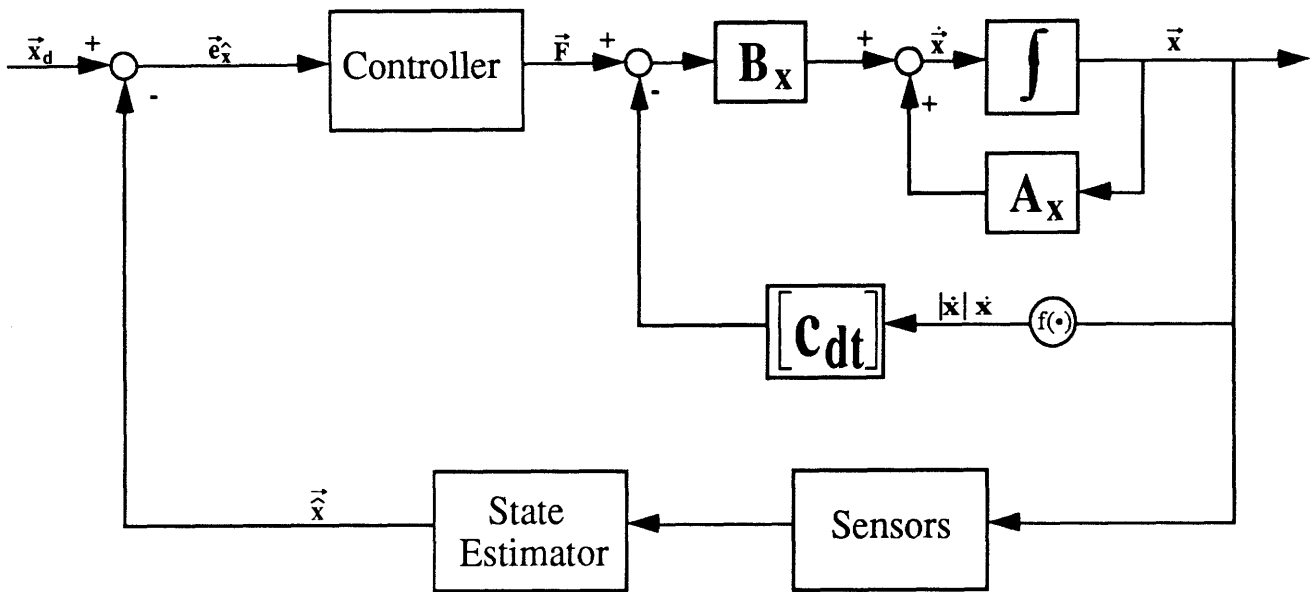


Figure 26a. Position Maneuver Block Diagram

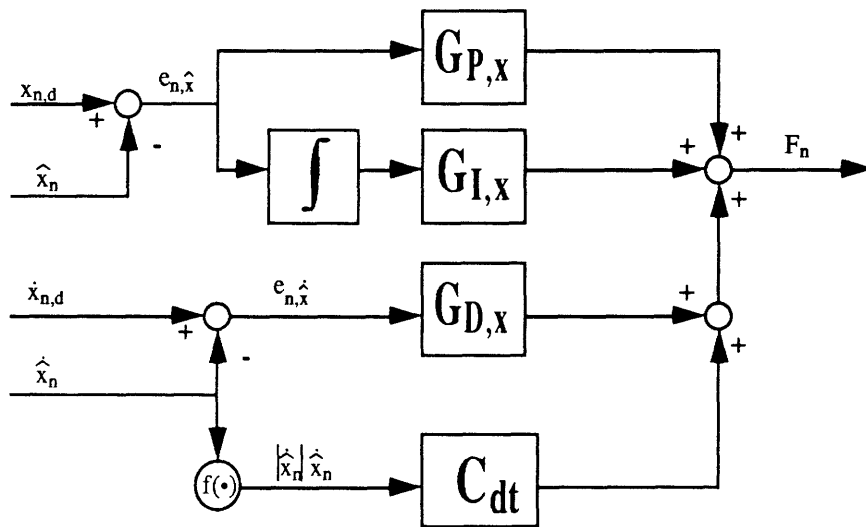


Figure 26b. Position Feed-Forward Control Diagram

Section 4.2 for regulation about a set point. However, a new feedback path is introduced. Equation (4.3-1) describes the control forcing output, F_n , given estimated states values of $(\hat{x}_n, \hat{\dot{x}}_n)$ and desired state values of $(x_{n,d}, \dot{x}_{n,d})$.

$$F_n = G_{P,x}(x_{n,d} - \hat{x}_n) + G_{I,x} \int (x_{n,d}(t) - \hat{x}_n(t)) dt + G_{D,x}(\dot{x}_{n,d} - \hat{\dot{x}}_n) + C_{dt} |\hat{\dot{x}}_n| \hat{\dot{x}}_n \quad (4.3-1)$$

For attitude, drag is only one of two nonlinear terms in the equations of motion. The other term arises from cross-coupling produced by the term $\vec{\omega} \times [I] \vec{\omega}$. As with drag, this nonlinear inertia term may be cancelled by the actuator torque, T_n . Figure (27a) shows the complete system diagram for large attitude maneuvers. The plant dynamics are affected by both the drag and cross-coupled inertia terms.

Because the attitude is expressed in inertial coordinates while the angular velocity and acceleration are in body coordinates, the nonlinear kinematic inversion depicted in Figure (27a) must be performed on the desired and estimated vehicle attitude elements. The attitude controller then receives the angular velocity discrepancy from a desired value, so that a proportional-derivative control system may be implemented with respect to each of the body axes.

Figure (27b) shows the detailed control scheme with the feed-forward nonlinear terms. Hydrodynamic drag and inertia cross-coupling are fed through the control system to cancel the nonlinear dynamics. Equation (4.3-2) shows the elements of the control torque, T_n , with angular velocity estimate $\hat{\omega}_n$ and desired angular velocity $\omega_{n,c}$.

$$T_n = G_{P,\phi} \int (\omega_{n,d}(t) - \hat{\omega}_n(t)) dt + G_{D,\phi}(\omega_{n,d} - \hat{\omega}_n) + C_{dm} |\hat{\omega}_n| \hat{\omega}_n - (\vec{\omega} \times [I] \vec{\omega})_n \quad (4.3-2)$$

4.3.2 Feed-Forward System Gains

The PID feed-forward system gains may be determined as for any linear PID system. The only limiting factor is the control authority used by the feed-forward terms. The maximum expected force required to cancel the translational drag term would occur when MPOD is moving at terminal velocity, $\dot{x} = 0.5$ m/sec. The required force to completely cancel this term is equal to the MPOD maximum thrust value of 120 N, by the definition of terminal velocity in Chapter 3. This means that the feed-forward drag term will always

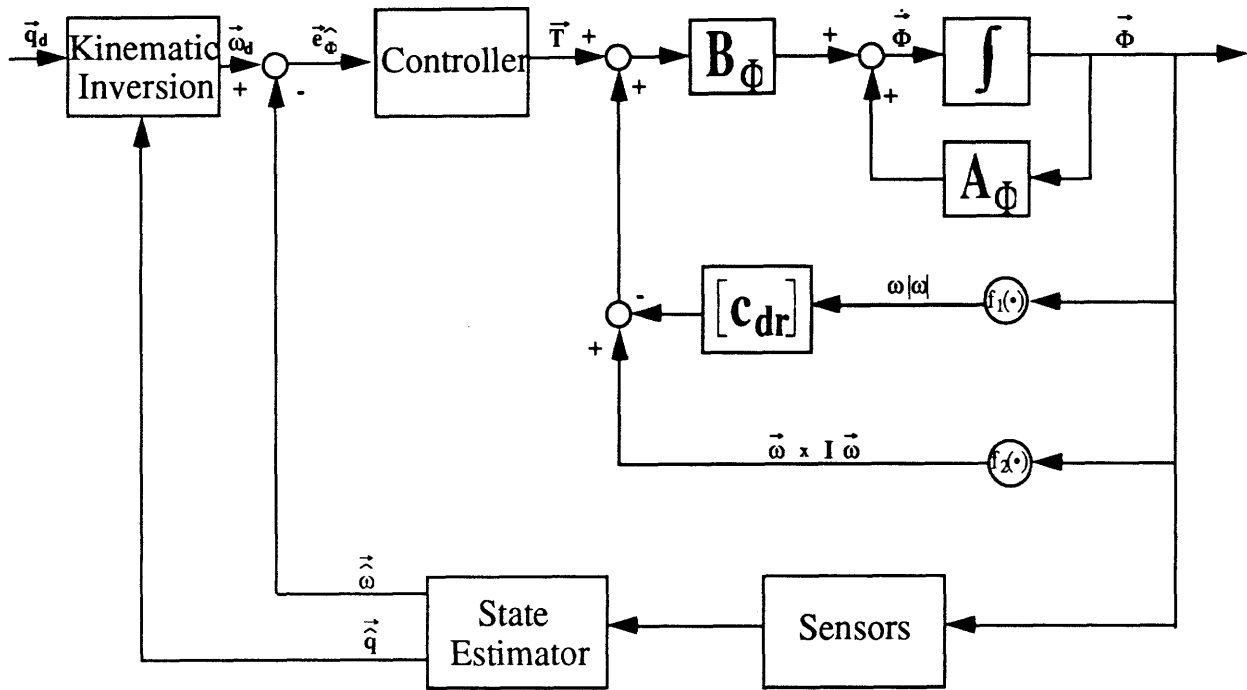


Figure 27a. Attitude Maneuver Block Diagram

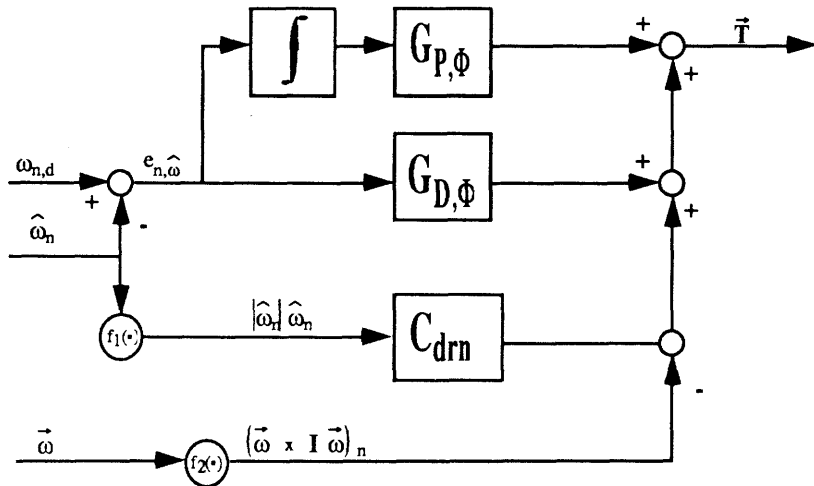


Figure 27b. Attitude Feed-Forward Control Diagram

have the capability to cancel nonlinear plant dynamics, but may also limit the PID control authority.

For attitude, the maximum required torque to completely cancel rotational drag is also equal to the terminal angular velocity about each axis. The same thruster saturation condition is true for attitude as was true for position. Finally, the nonlinear inertia term is always small in magnitude, due to only slight differences in axis moments of inertia. Therefore, there is no chance that the cross-coupled inertia term will alone cause thruster saturation.

The feed-forward linearization scheme is easy to implement, while still accounting for nonlinear terms in the equations. This system should be considered for the MPOD control system used for large attitude maneuvers.

5.0 Experiments and Test Results

MPOD underwater tests were conducted at the MIT Alumni Pool. Because the pool was used for swimming during normal hours, tests were held every 2-3 weeks on Saturday nights and the following Sunday mornings. The main disadvantage to such a schedule was the infrequency of testing and inability to debug many problems in time for the rest of the test session. For example, 3DAPS on MPOD could only be tested while both were completely submerged, so that with each modification, a long wait followed during which there could be little debugging. MPOD and 3DAPS would then be placed back in the pool, only to find that the supposed "solution" to the problem had not actually been effective. Many MPOD pool tests were used for debugging 3DAPS in this manner, primarily because of the inability to find errors between test sessions.

All pool test data presented in this thesis was gathered during the weekend of April 21-22, 1990. Static data from previous pool tests was valid; however, the 3DAPS calibration offset had changed with the conversion from NOVRAM to EPROMs in the microcontroller boards. This range offset will be the same so long as the 68HC11 software is not modified. The calibration parameters in Chapter 3 were valid for all pool test data presented in this thesis.

The experiments were conducted and analyzed in the following order. First, static tests were performed in which MPOD's location could be determined empirically without relying on the data. Second, to distinguish translation maneuvers from attitude movements, a balanced MPOD was flown in an upright attitude between two fixed points. Finally, attitude maneuvers were included in free flight testing.

Before studying the state estimator's performance in underwater tests, relatively ideal surface experiments were performed in simulation. In addition to providing unlimited time for data collection, the simulator provided its own state calculations, with which the state estimations were quantitatively compared. This chapter presents a description of the simulation algorithms and test results for the study of state estimator performance. Following simulation results, state calculation data from underwater MPOD tests is presented with a qualitative performance analysis.

5.1.0 MPOD Simulation

Two simulation programs were written to run on the MPOD computers. They were performed on *Obi-Wan* because it was a 286 processor with math coprocessor, had access to all dual port RAM, and had the least responsibility during state calculation tests.

5.1.1 Dynamic Simulation Description

The simulation software was implemented on MPOD because of the speed of the available parallel processing and the ability to debug most of the software in its flight configuration. The first of the two simulations was written to enable testing of both the state calculation and future control systems. The operator may choose any initial position and attitude, assuming a zero initial velocity. For dynamic tests, the control station hand controllers may then be used to "fly" the simulator as the vehicle would be flown. Both simulated and estimated data were viewed on the screen, with the option of data saving on *Obi-Wan's* NOVRAM.

The dynamic simulation software is shown in Appendix D.1. The hand controller commands are converted to thrust and torque values which are used in the simulation equations, then sent to the filter. For integration of the equations of motion, the dynamic simulator uses a fourth order Runge-Kutta algorithm. Reference 26 describes the software used in the MPOD simulation. After the simulator calculates the expected MPOD states, sensor measurements are calculated. Because the real sensors contained a known amount of uncertainty, the simulated sensor measurements were each corrupted by random noise proportional to twice the corresponding actual expected sensor noise. This value was chosen to account for unexpected sensor behavior, and examine the filter response when suboptimal conditions were present.

Because the simulation and filter software were running on different computers, the loop times were slightly different. Therefore, the simulation and filter time steps used in the equation of motion integrations did not precisely match. This could have been

advantageous for analysis purposes, due to the fact that the state estimation loop does not exactly integrate at real-time.

The simulation software contained the same dynamic parameters as the filter software. Because the actual MPOD parameters most likely do not agree with the numerical estimates, MPOD dynamics could later be modified in the simulator in order to observe the resultant estimated state discrepancies. It is expected that at least the linear velocity estimate will not match nearly so well, due to the fact that no sensor feedback is available.

Another useful addition to the simulator would be disturbance modelling. By applying disturbances to the simulator equations of motion only, the filter would be forced to rely on the sensor measurements for disturbance rejection. In this manner, MPOD's ability to calculate its state despite water currents or buoyancy offsets may be examined.

5.1.2 Simulation Static Tests

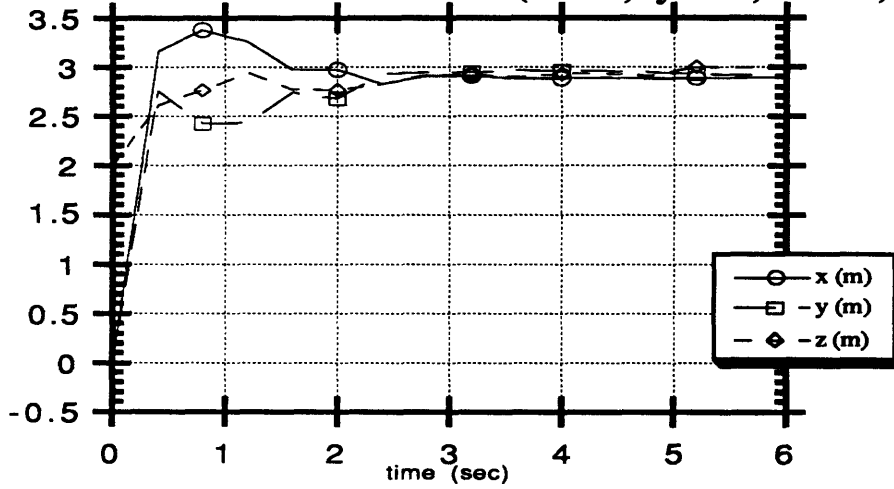
Although the filter should ideally be started when inertial and MPOD body axes are aligned, this is not always feasible. When starting near the filter initialization point, the state estimate consistently converges to the proper value. However, the filter may not converge to the actual state if MPOD is sufficiently far from the initial estimate. The simulator was used to test the filter's ability to converge to a new state, given the initial state estimate of the pool center, or $x=0$, $y=0$, $z=2$ m, with inertial and body axes aligned.

Figures (28) and (29) show the position results for different static locations. The first figure shows estimates derived from a low initial covariance ($P = 0.2$ m), which indicates to the filter that the state estimate is "close" to the actual state. Figure (29) shows the state estimates when a high initial covariance ($P = 3.0$ m) is passed to the filter, indicating a high level of uncertainty in the current state estimate. Three possible static positions are shown.

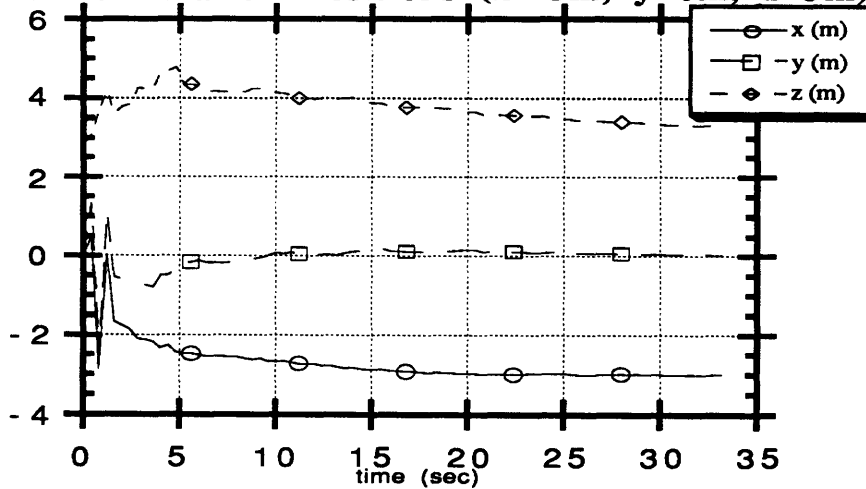
With all the runs, the large initial covariance produces a large overshoot in state estimate. However, the low initial covariance runs require much more time to stabilize to the final value; for example, in the second run, with $x=-3$ m, $y=0$ m, $z=3$ m, the low initial covariance barely converges after 35 seconds, while the high initial covariance run for the same static point overshoot by up to a meter, but settles to the correct value within 6 seconds. Except for the $z=1$ m coordinate in the third static position run, all the static positions shown in these figure settle to their correct steady-state values. In both the high and low initial covariance runs, the $z=1$ m location in the third run settles to ~ 1.2 m. Perhaps with more testing and further covariance parameter tests, the position behavior

Figure 28. Static Simulation Position Plots with Low Initial Covariance

Static Simulation Positions ($x=3\text{m}$, $y=3\text{m}$, $z=3\text{m}$)



Static Simulation Positions ($x=-3\text{m}$, $y=0\text{m}$, $z=3\text{m}$)



Static Simulation Positions ($x=2\text{m}$, $y=-4\text{m}$, $z=1\text{m}$)

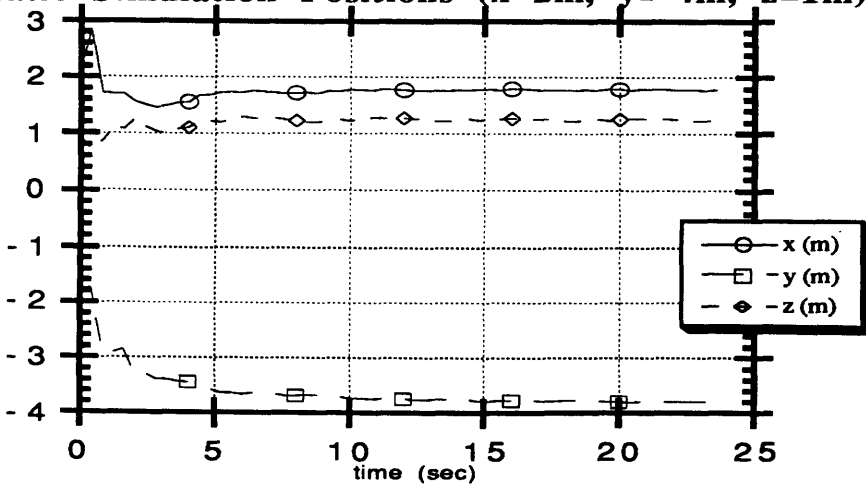
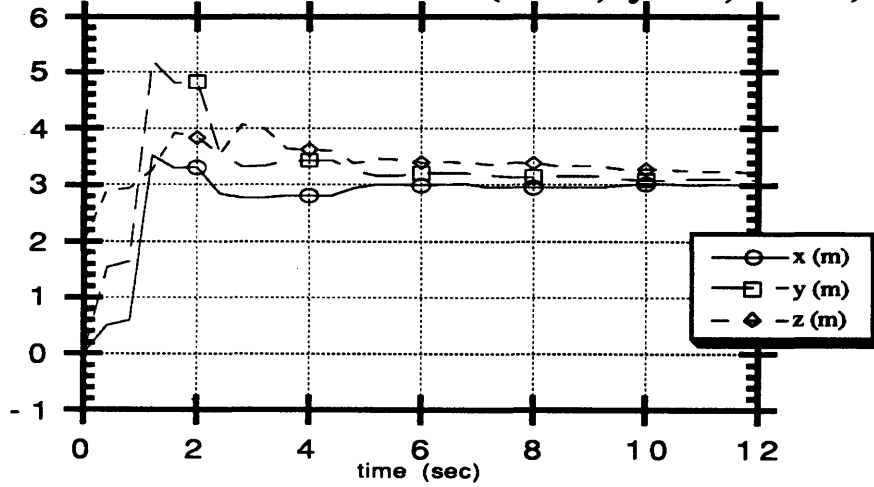
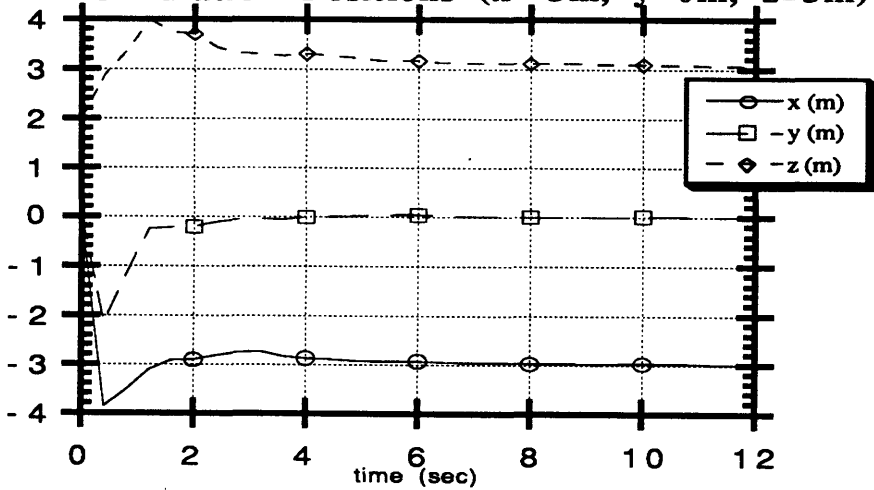


Figure 29. Static Simulation Position Plots with High Initial Covariance

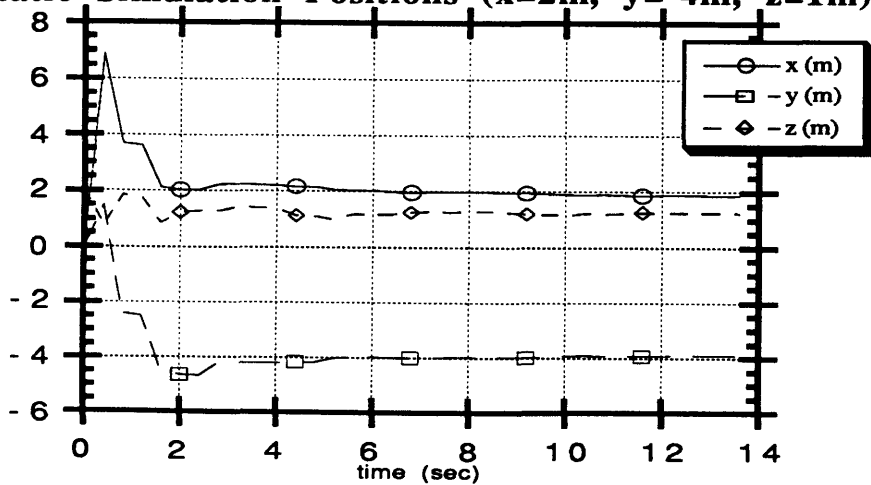
Static Simulation Positions ($x=3\text{m}$, $y=3\text{m}$, $z=3\text{m}$)



Static Simulation Positions ($x=-3\text{m}$, $y=0\text{m}$, $z=3\text{m}$)



Static Simulation Positions ($x=2\text{m}$, $y=-4\text{m}$, $z=1\text{m}$)



may be assessed and better predicted. However, the current recommendation remains to start the filter near the pool center, which is the defined zero point in the software.

Figures (30) and (31) show the static test results for attitude convergence to a known value. Three distinct convergence cases were shown in static tests, one with the low initial covariance value of $P=0.02$, and the other with the high initial covariance value of 1.0. For a 180° pitch angle, the initially high covariance system converges to the proper attitude within 6 seconds, while the initially low covariance run shows the beginnings of convergence after 30 seconds of testing.

The worst steady-state error in the static simulations was observed during the second test, representing a -90° yaw attitude. This discrepancy may partially be due to the fact that pendula do not assist in the state estimation, while in the other two runs, the pendula do provide information about the altered vehicle state. The quaternion elements q_1 and q_3 converge to the proper values within 5 seconds, however, q_0 is low by approximately 0.1 in both runs, while q_2 is unexpectedly nonzero, at a steady-state magnitude of -0.3. A long-duration run of more than a minute was conducted which confirmed that the quaternion elements were indeed experiencing steady-state errors, not just slow transient responses. No solution to this problem has yet been found, although it is suggested that initial covariance and state variance parameters be varied, with the hope of finding some relation between convergence and filter parameters.

The final attitude test shown in Figures (30) and (31) is a static 90° roll angle. Yet a third convergence case was observed. The low initial covariance run produced quaternion elements to within 0.1 of the desired value within 4 seconds, but the high initial covariance run never converged. Note the interesting sinusoidal behavior of the zero quaternion elements.

As previously stated, the best guarantee of a filter with no steady-state error is to start the state calculations with the vehicle near the initialization point of the filter software. From the static tests, it appears that position converges to the proper steady-state value more easily than attitude, and is not as sensitive to the choice of initial covariance. However, further tests need to be performed to quantify the affects of filter parameters on static state estimate convergence.

5.1.3 Simulation Dynamic Tests

By using the simulator to generate dynamic MPOD data, the "actual" vehicle state is always known. Performance of the state estimator may then be determined by comparing its estimate with the actual state as a function of time. Figure (32) shows the positions

Figure 30. Static Simulation Attitude Plots with Low Initial Covariance

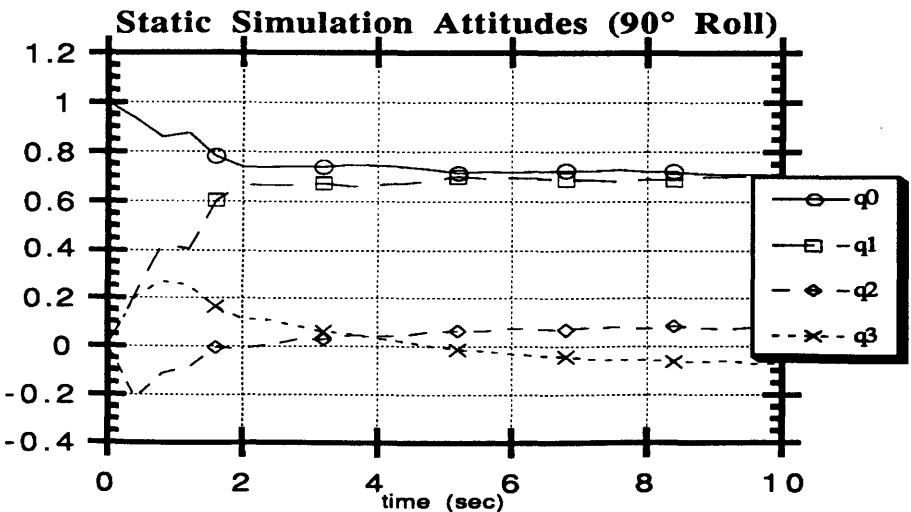
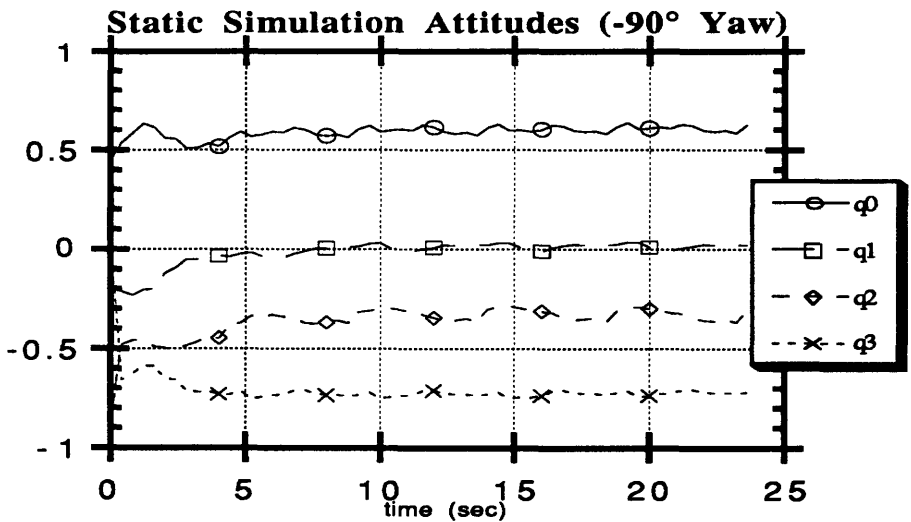
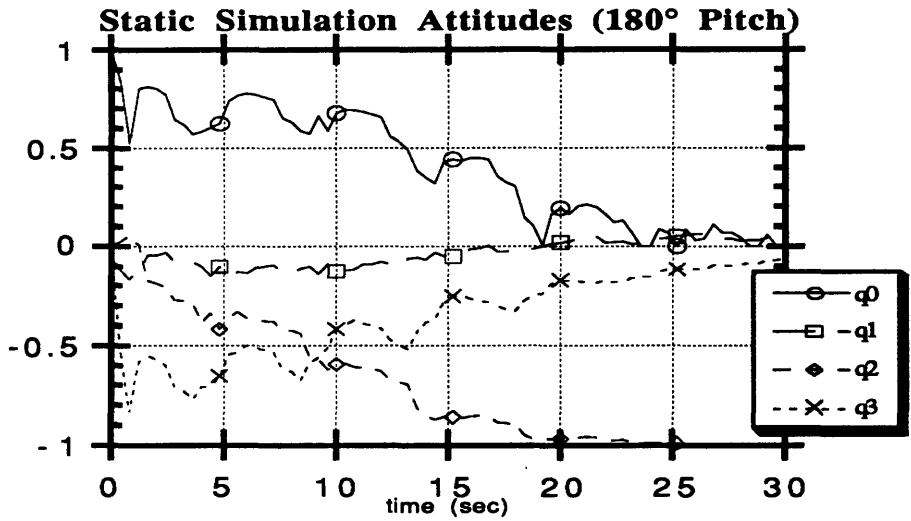
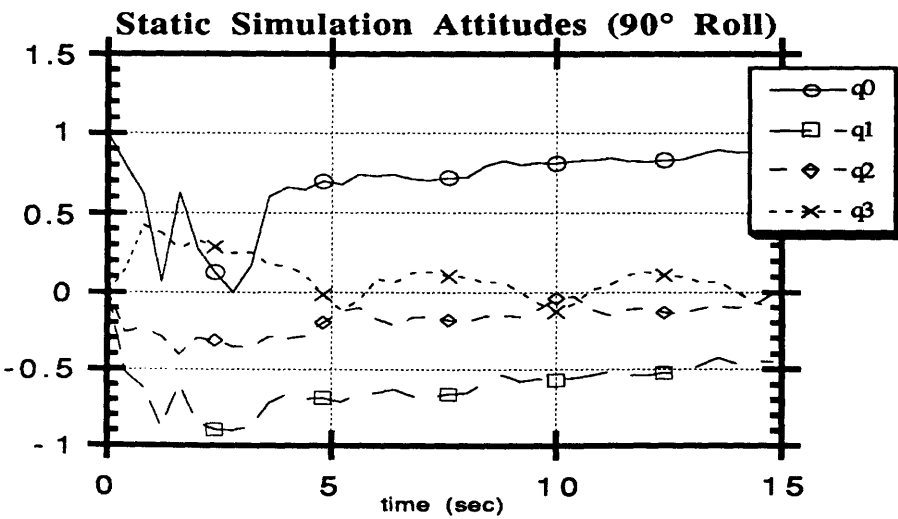
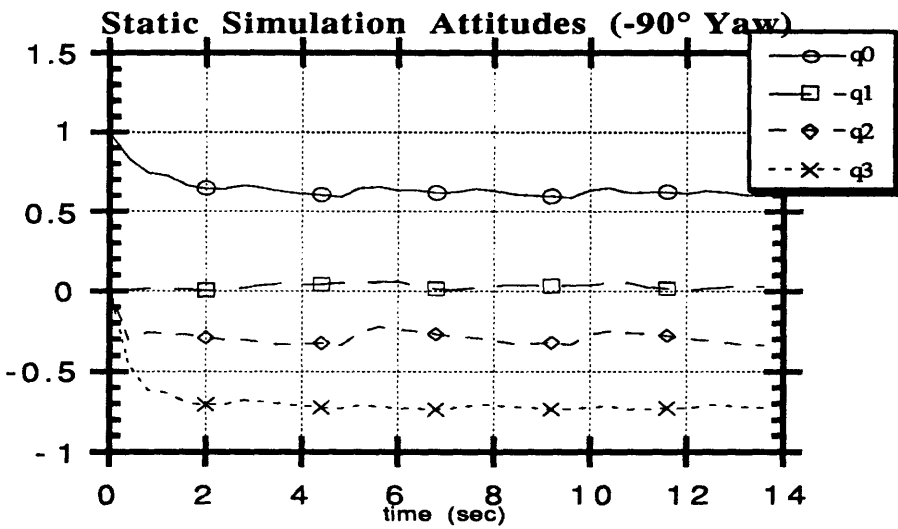
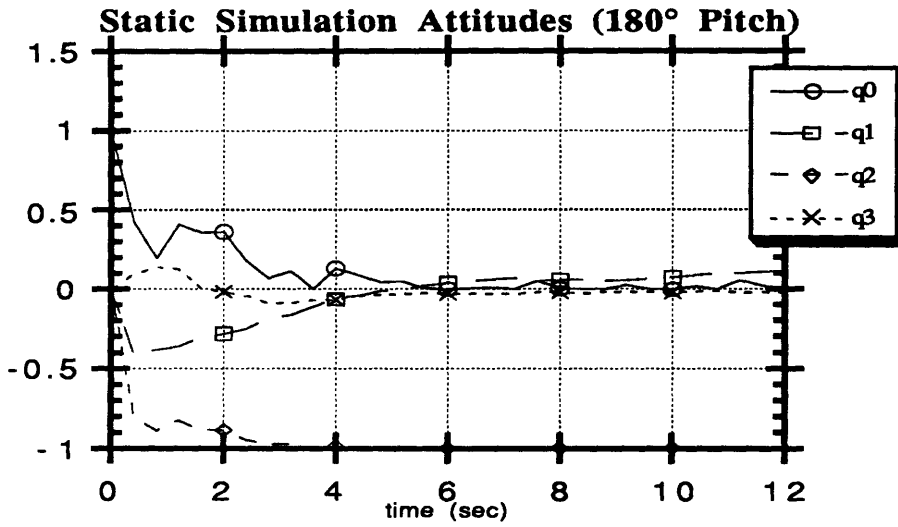


Figure 31. Static Simulation Attitude Plots with High Initial Covariance



during a typical dynamic state calculation run. The x and y estimated position curves match the actual state curves in both shape and magnitude. However, the estimated z value does not agree, by up to a ~1.7 m discrepancy.

As expected, the estimated linear velocity curves shown in Figure (33) are not quite so accurate as the position plots. The estimated z-velocity matches the actual value up to approximately 20 seconds into the run, at which point it appears to mirror the actual value. This seems to indicate that the z-thrust is being used in an inverse sense by the estimator. Currently, the source of this error is unknown.

Figures (34) and (35) show the quaternion elements of the state vector. The first quaternion shows two distinct spikes at times of approximately 38 and 44 seconds into the run. It is suspected that these are associated with the "state variance" being added to the covariance during each filter loop. Although its magnitude is only 0.0001 for these calculations, the state variance boosts the covariance values by 0.005 every second. This may be significant in attitude calculations.

Except for the initial startup transient and two spikes of maximum amplitude 0.4, the quaternion estimates q_0 , q_1 , and q_2 match the actual state to within 0.006 of their actual values, while q_3 has a maximum error of magnitude 0.12. By studying Figure (36), the q_3 discrepancy may at least partially be explained by the yaw velocity error. This error is due to the simulated null offset of the yaw rate sensor. Starting at a simulated initial offset of 0.15 rad/sec from the filter expected offset value, the estimated yaw velocity underwent transient behavior to account for the unexpected rate sensor null offset. Note that the roll and pitch angular velocities shown in Figure (36) are accurate to within 0.05 rad/sec for the entire simulation run.

Although no definite conclusions may be drawn from the dynamic simulation results performed to date, it seems that the state estimator is performing its job reasonably well. Two positions and three quaternion elements were acceptably accurate, with one quaternion possibly corrupted by rate sensor null offset and one position corrupted from an undetermined source. Hopefully with insight gained by more covariance and state variance analysis, all elements of the state vector will consistently agree with dynamic simulation results.

5.1.4 Real-time Simulation using Pool Test Data

The state calculation software had not been thoroughly debugged before the MPOD pool test in which the data was collected. Therefore, a simulation was developed in which the pool test data could be fed into the state estimation software exactly as it would have

Figure 32. State Estimator Position Tracking

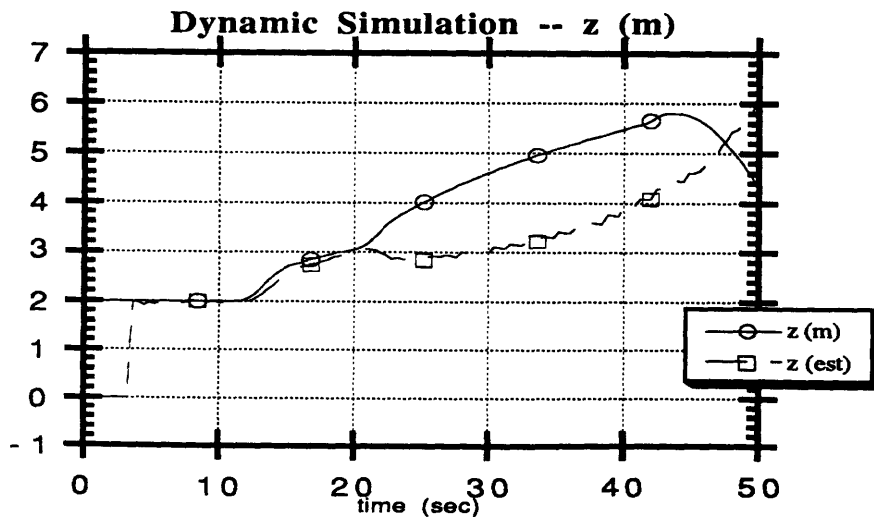
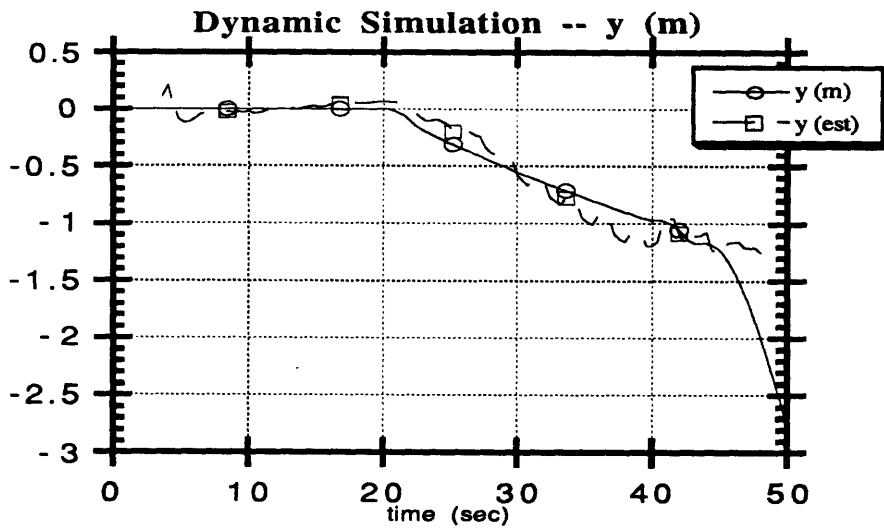
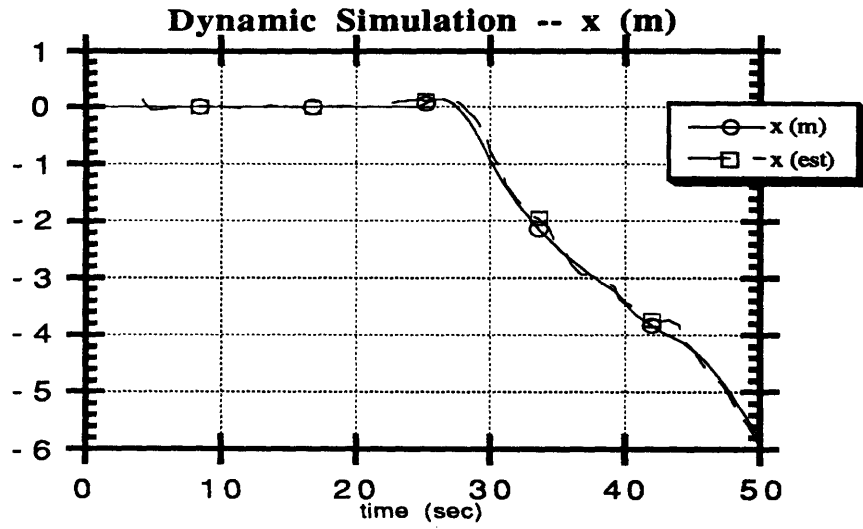


Figure 33. State Estimator Linear Velocity Tracking

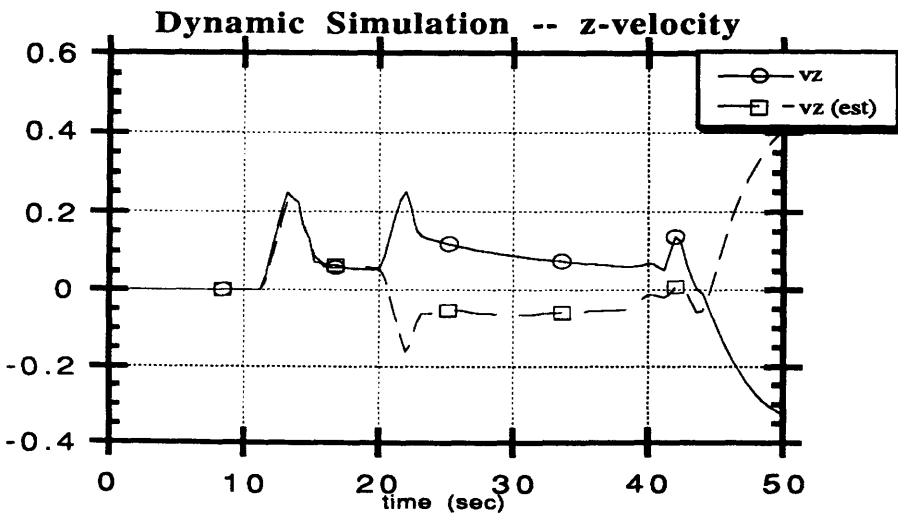
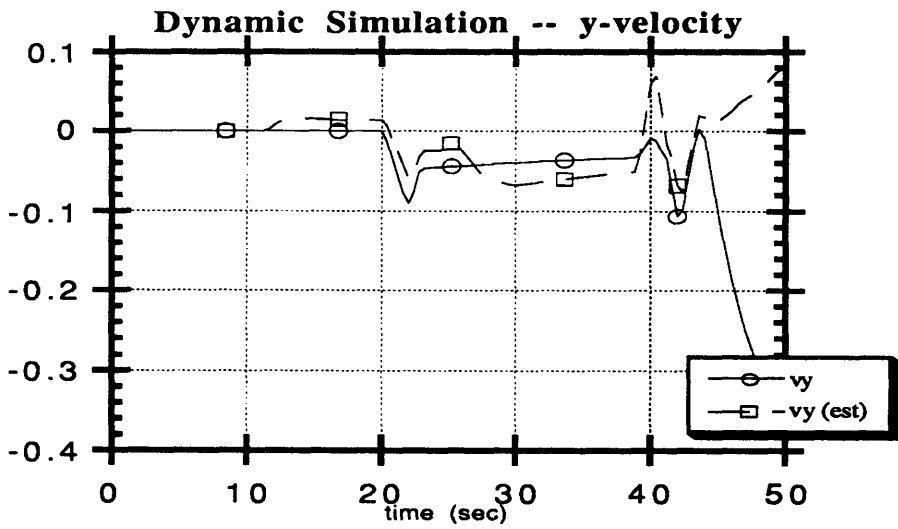
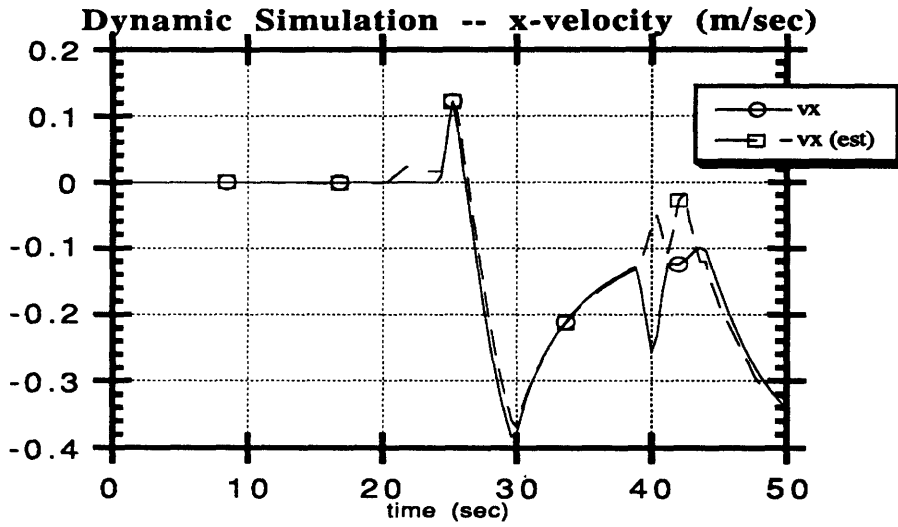


Figure 34. State Estimator Attitude Tracking (q_0 & q_1)

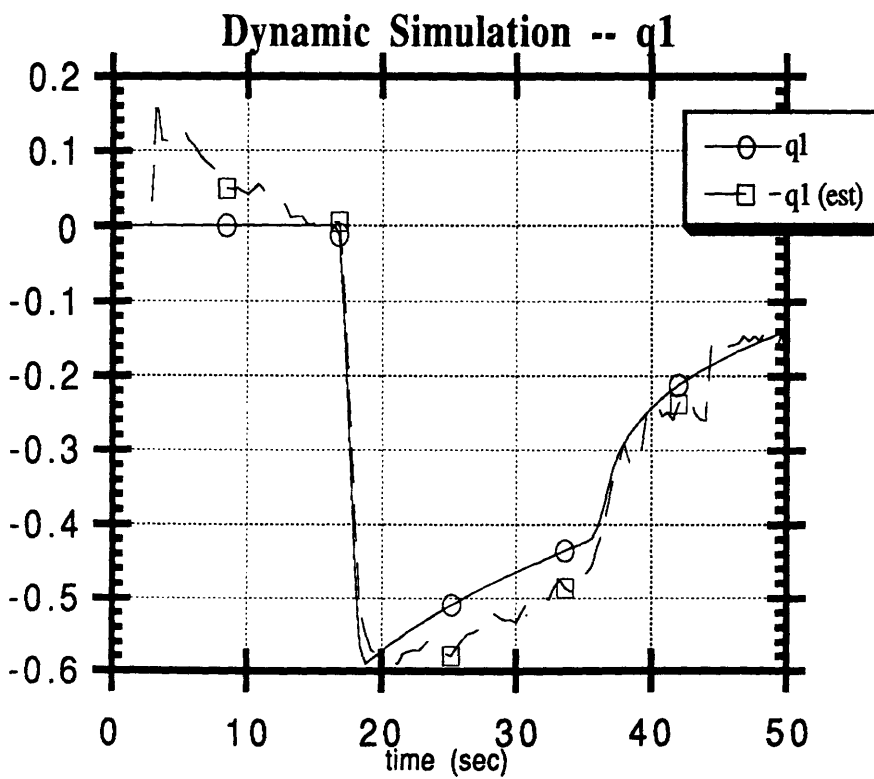
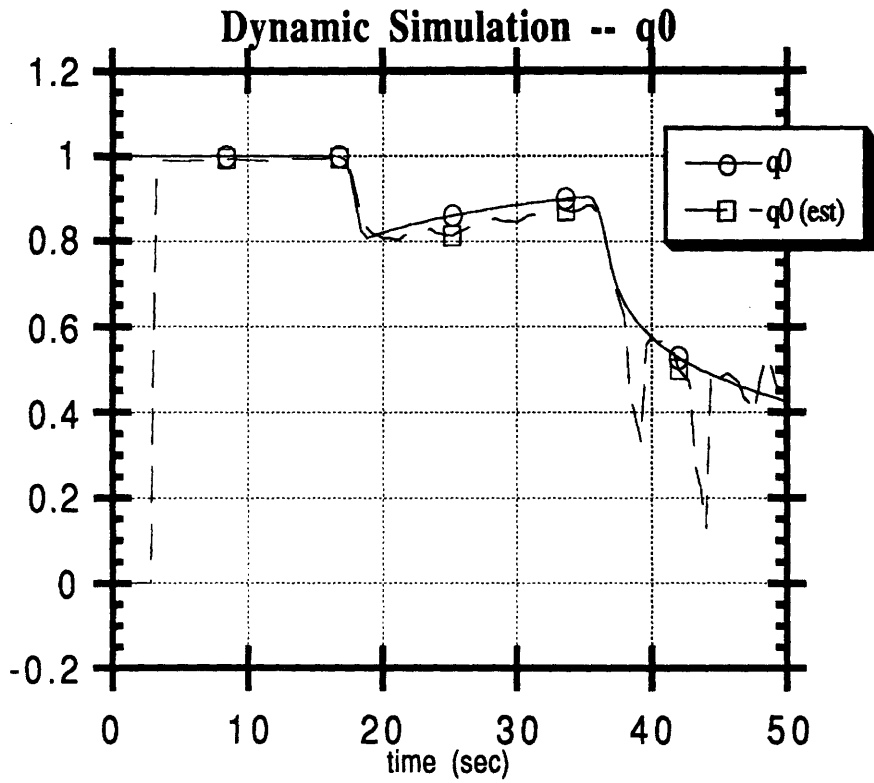


Figure 35. State Estimator Attitude Tracking (q2 & q3)

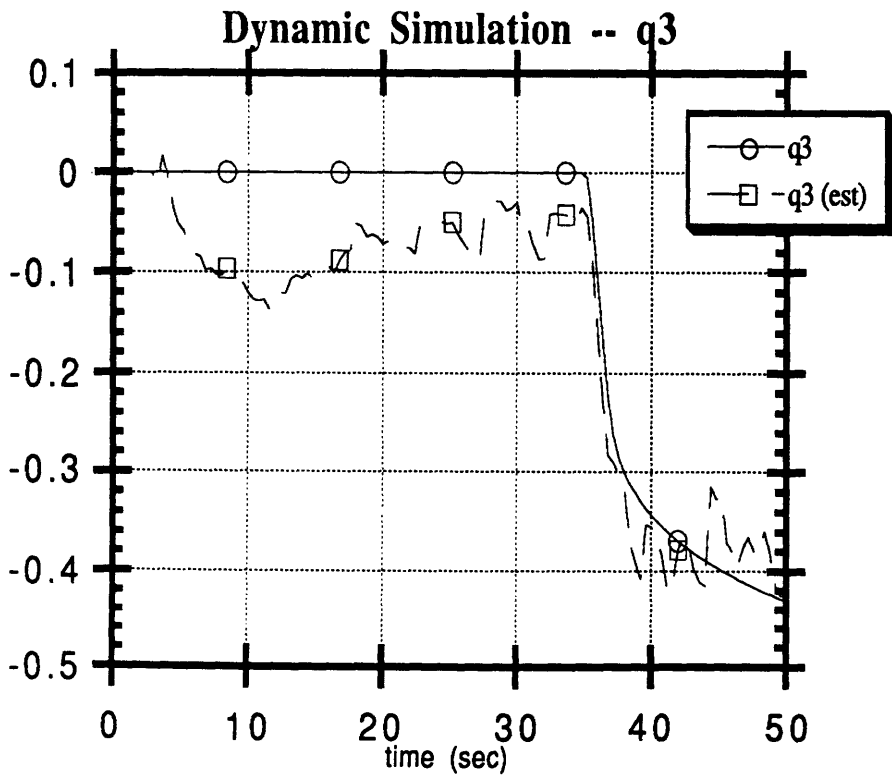
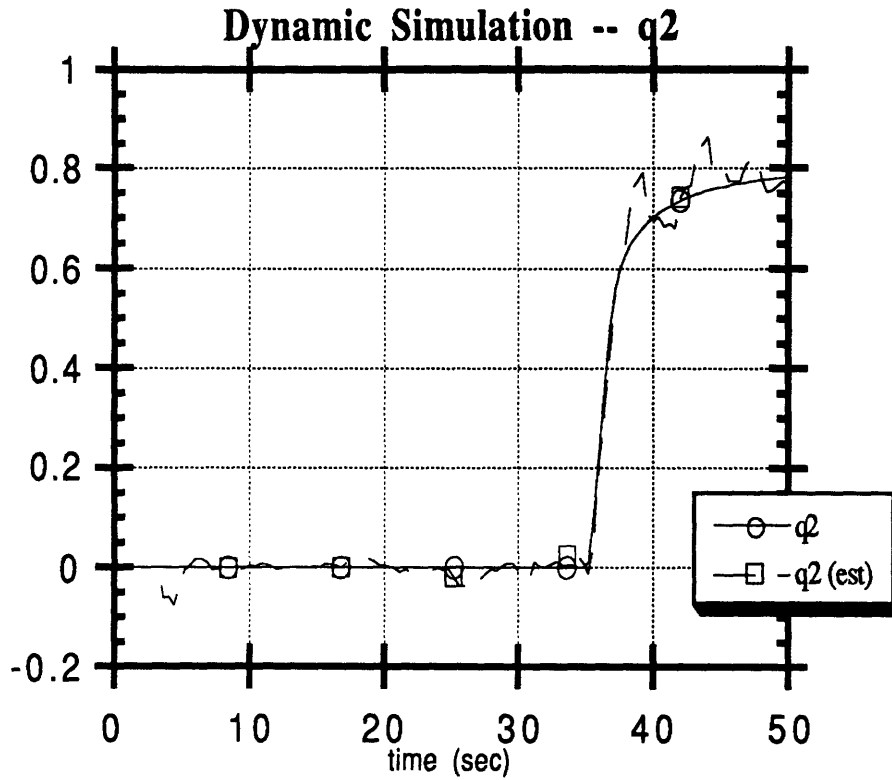
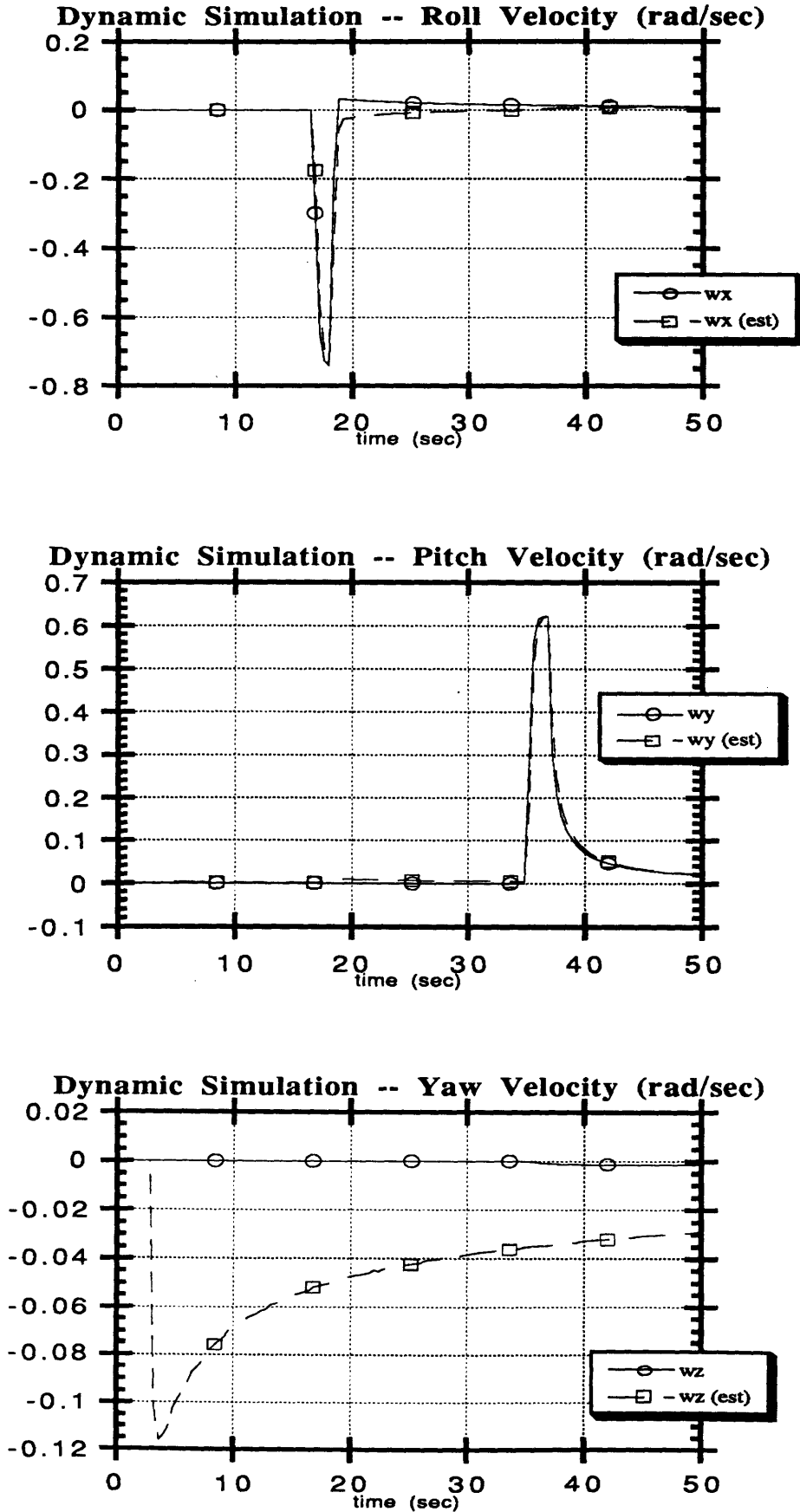


Figure 36. State Estimator Angular Velocity Tracking



occurred during real-time underwater experiments. Because MPOD was flown open-loop, sensor data and thrust commands could be used to recreate each underwater test.

All MPOD state calculation results presented in the remainder of this chapter were produced during post-test simulation with the pool data. The only software changes from the flight *Obi-Wan* software involved reading motor commands the sensor data instead of calculating or writing them. Because less 3DAPS data was generated than for the other sensors, the data was read from the range data file at an infrequent interval. By trial and error, the software was set such that 3DAPS ranges were read at approximately the same rate as they were originally generated. The minimal changes to the *Obi-Wan* software meant a program which ran at approximately the same speed with the same sensor and thrust values during the actual underwater tests. Note that the *Lando* software was not modified for the simulation, so that the state calculations occur precisely as they would at the pool. Appendix D.2 shows the main driver program for *Obi-Wan* as modified for these simulation runs.

5.2.0 Underwater Static Tests

5.2.1 Test Procedure

Sensor calibration and underwater static state estimation tests could only be assessed if MPOD's position was known and remained constant during data collection. For the pool test, three different physical locations were chosen to assess MPOD sensor and state calculation performance in a variety of static situations. To prevent MPOD from drifting, the cage surrounding MPOD's bottom motor ducts was anchored to the pool floor with 50 lbs. of diving weights. Figure (37) shows the three locations used for static tests. They will hereafter be referred to as *Loc1*, *Loc2*, and *Loc3*.

Static testing was performed in the following sequence: (1) MPOD was firmly stabilized on the pool floor, (2) sensor data was taken, (3) divers measured from each thumper to each hydrophone while the static test data was uplinked to *Luke*. Between static tests, data was collected as MPOD flew in a triangular pattern from *Loc1* to *Loc2*, from *Loc2* to *Loc3*, then back to *Loc1*.

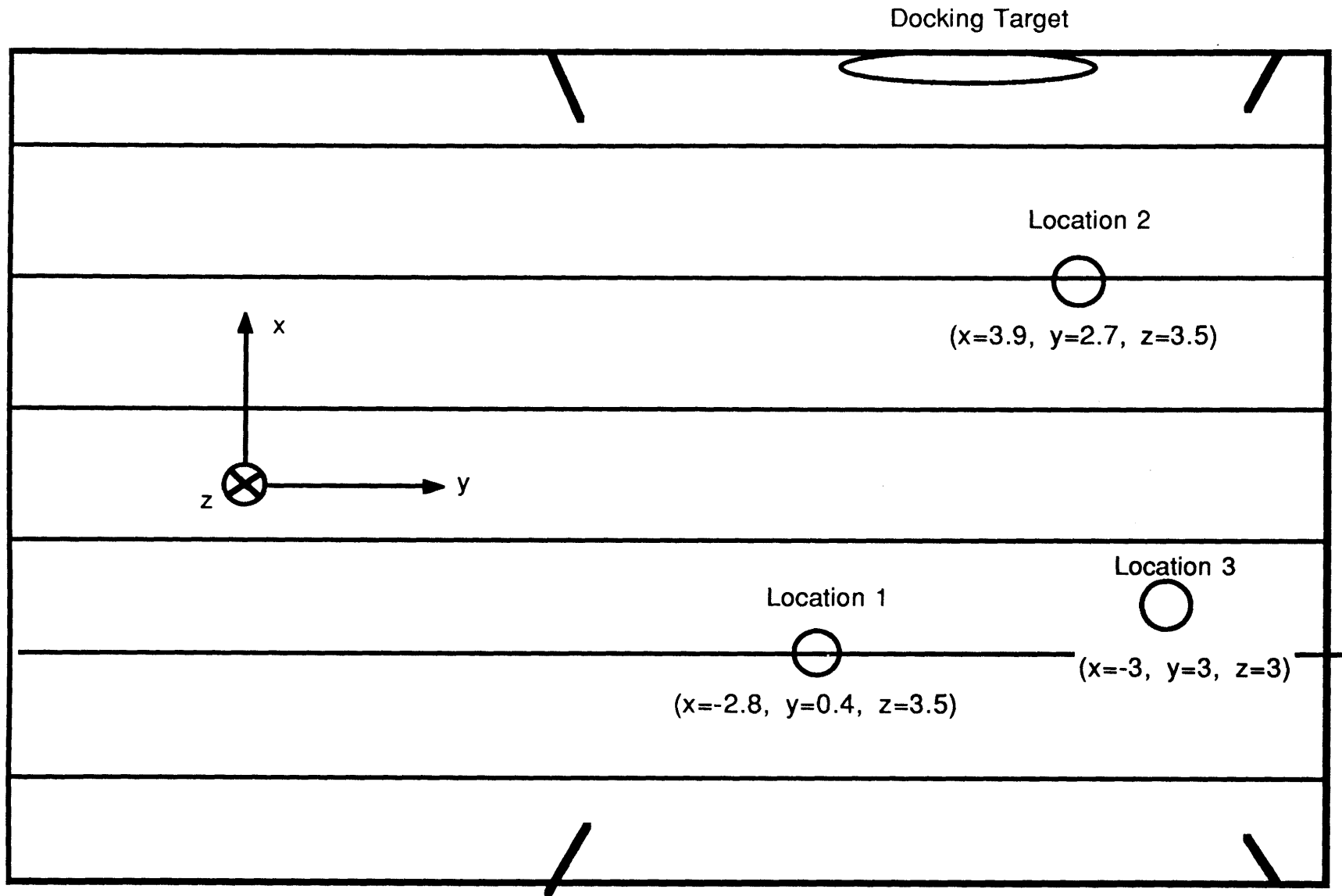


Figure 37. Static Test Pool Locations

5.2.2 Static Testing for Sensor Calibrations and Accuracies

The easiest method for calibrating 3DAPS was to take static data from a constant underwater locations, then perform a linear curve fit on a plot of range data vs. measured ranges. The static testing described in this section was used for the chapter 3 range calibration. Appendix C.3 shows the range calibration plots for each location. Note that the curve fits are approximately the same for each of the three locations. In calibrating the ranges, hydrophones that were blocked from a thumper by MPOD were removed from the data. Blocked hydrophone-thumper combinations produced either a completely ridiculous range value, or a range which was consistently wrong by a few thousand counts. The consistently erroneous range is suspected to result from either a reflection or non-straight line component of the thumper acoustic pulse.

Other sensors examined during the static tests include the rate and depth sensors. Although the rate sensor static properties were also examined at the surface, the underwater static tests were used to verify that a closed control box underwater did not affect their properties. The depth sensor was calibrated during MPOD static tests. By measuring the distance from the water surface to the MPOD center, the sensor reading was calibrated. Also, it was observed that the depth sensor had far more static error during underwater tests than for the zero depth at the surface. The source of this error is suspected to be water currents which produce pressure changes across the depth sensor inlet. Appendix C.2 shows a chart of observed averages and standard deviations of MPOD's sensors during static tests. Significant non-zero rate sensor null offsets were present in roll and yaw rates. Depth standard deviation was less than 0.1m, while the average range standard deviation was 42.7 counts, which corresponds to a 0.05 m standard deviation.

5.2.3 State Estimation Properties

Because only ranges were measured during static tests, there was no method for precisely determining MPOD's location in the water. By visual inspection, the upright MPOD attitude was determined to within a few degrees, while the 3-axis position elements were assessed to within a meter. Note that the depth was deterministic because MPOD was resting on the pool floor.

Table 5.2-1 shows the static locations as determined by the filter. Except for the x value at *Loc3*, the estimated locations, both quaternions (± 0.1) and positions (± 0.5 m),

agreed with actual values. Also shown in the table are the standard deviations of state calculations during each test.

		x (m)	y (m)	z (m)	q0	q1	q2	q3
Location 1:	average	-2.79	0.438	3.601	0.087	0.066	0.07	0.99
	standard deviation	0.025	0.034	0.027	0.052	0.018	0.012	0.007
Location 2:	average	3.935	2.68	3.736	0.095	0.019	-0.07	-0.98
	standard deviation	0.11	0.133	0.201	0.136	0.049	0.04	0.033
Location 3:	average	-3.21	3.137	2.94	0.793	0.232	-0.19	-0.52
	standard deviation	0.082	0.169	0.197	0.047	0.06	0.026	0.05

Table 5.2-1 State Estimate Averages and Standard Deviations

5.3 Underwater Dynamic Tests

5.3.1 Test Procedure and Limitations

For dynamic state estimation tests during MPOD flight, the vehicle was flown from the surface control station. Data was recorded on MPOD, then uplinked to the surface between runs. At the beginning of each test, the vehicle was held in place for two complete 3DAPS iterations to enable some stabilization of the state estimation software. Then, MPOD was flown in some maneuver which was visually assessed as well as possible.

Because there was no way to measure MPOD movements during flight, the actual vehicle position and attitudes are completely unknown. Therefore, the state estimation results cannot numerically be assessed. They may, however, be qualitatively studied from the general pattern and directions of motion.

5.3.2 Constrained Straight and Level Flight

The first series of dynamic underwater tests was performed with the assistance of divers. The MIT Pool is separated into lanes, each marked by a black stripe along the "y-axis" of the pool floor. To assess MPOD's ability to calculate a known single-axis translation maneuver, divers prevented the vehicle from drifting as the pilot flew along a black stripe. The test results presented in Figures (38) - (42) were made from a flight beginning near the +y pool wall and continuing along the black stripe until ~2 m past the plane containing thumpers 2,3,6, and 7.

Figures (38) and (39) show the position and linear velocity plots, respectively, for the constrained test run. The y-position performs as expected, with a linear slope corresponding to the predicted y-velocity. The x and z curves are difficult to assess. The only position problem observed in the state plots is the unexpected increase in depth near the end of the run. This currently cannot be explained, but corresponds with the velocity estimate.

Attitude plots are shown in Figure (40). The known direction MPOD was facing corresponded with quaternion values of $q_0 = 0.7$, $q_1 = 0$, $q_2 = 0$, and $q_3 = -0.7$. This attitude was the same as in the -90° yaw static simulation tests described in Section 5.2. It is useful to note that the -0.3 offset of quaternion element q_2 was present in both the simulated static calculations and the actual pool test attitude calculations. If nothing else, this result lends an assurance of accuracy of the simulator to predict actual system behavior. Note that the angular velocities are zero except for an unexpected transient due to a yaw rate sensor null offset.

Figures (41) and (42) show ranges from hydrophones to thumpers. Because of their inaccurate values, the hydrophone-thumper paths which were blocked by the MPOD vehicle were not included in these plots. It may be of interest to note the local minima on the thumper 2, 3, 6, and 7 plots. The range values reach a minimum as the vehicle passes through the plane containing the thumpers. Then, the range values again increase. Ideally, all motion should be completed within the 3DAPS area, due to both thumper pointing and calculation accuracy. However, in this test, no instabilities were noticed after exiting the defined 3DAPS area.

Figure 38. Constrained (-y) Flight -- Positions

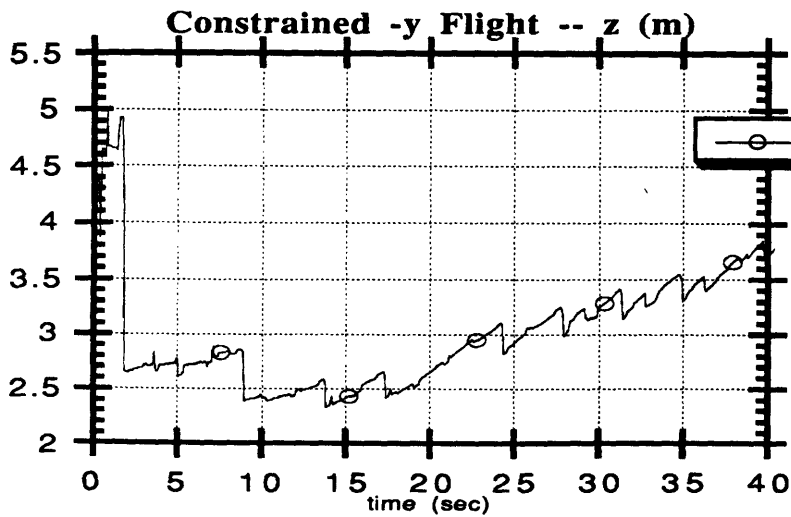
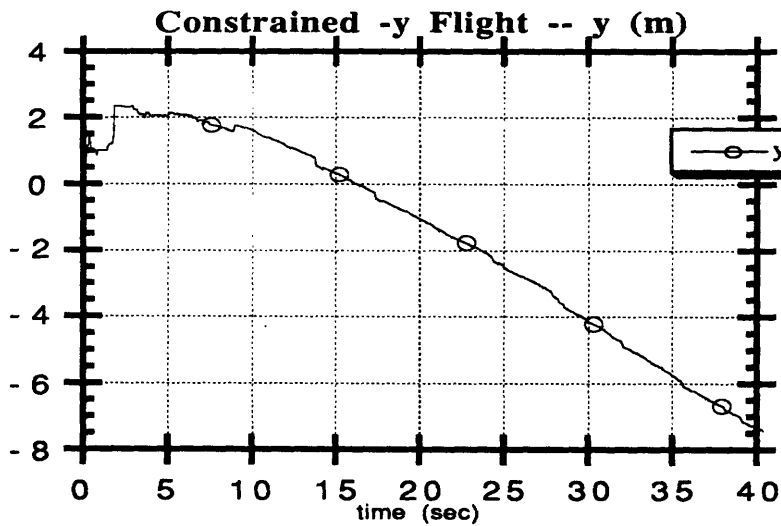
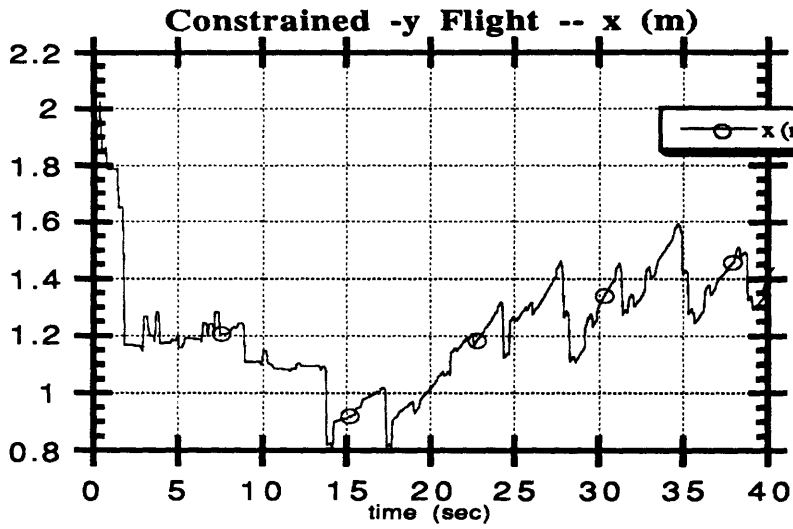


Figure 39. Constrained (-y) Flight -- Linear Velocities

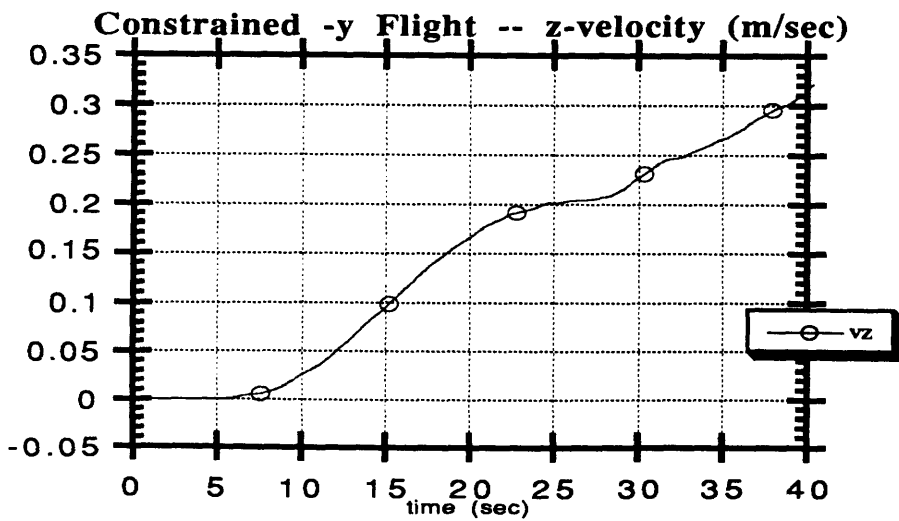
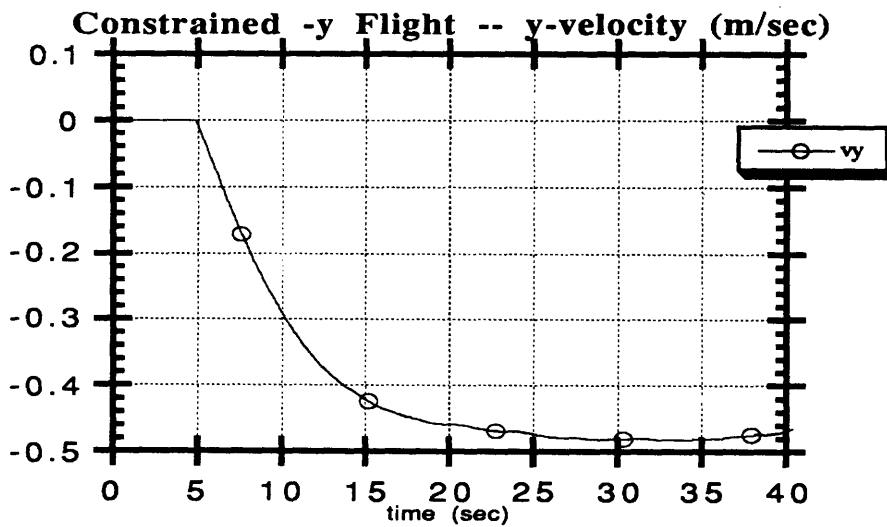
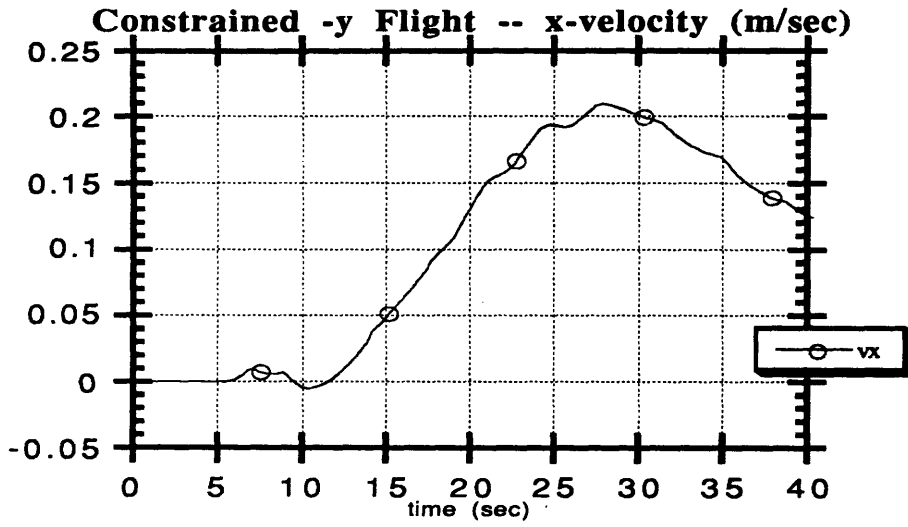


Figure 40. Constrained (-y) Flight -- Quaternions and Angular Velocities

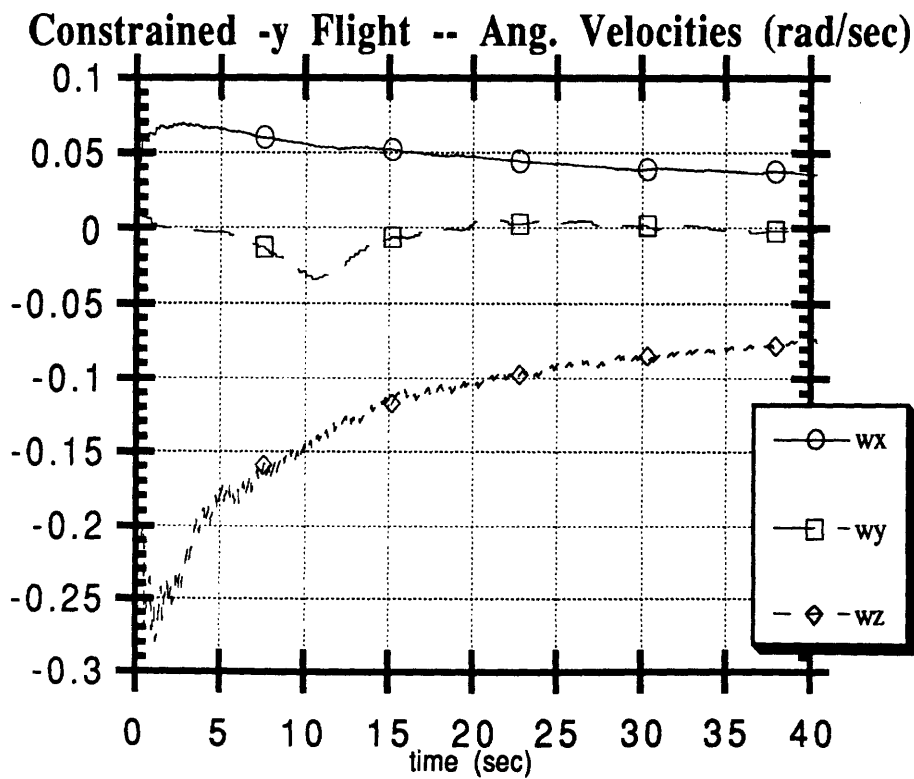
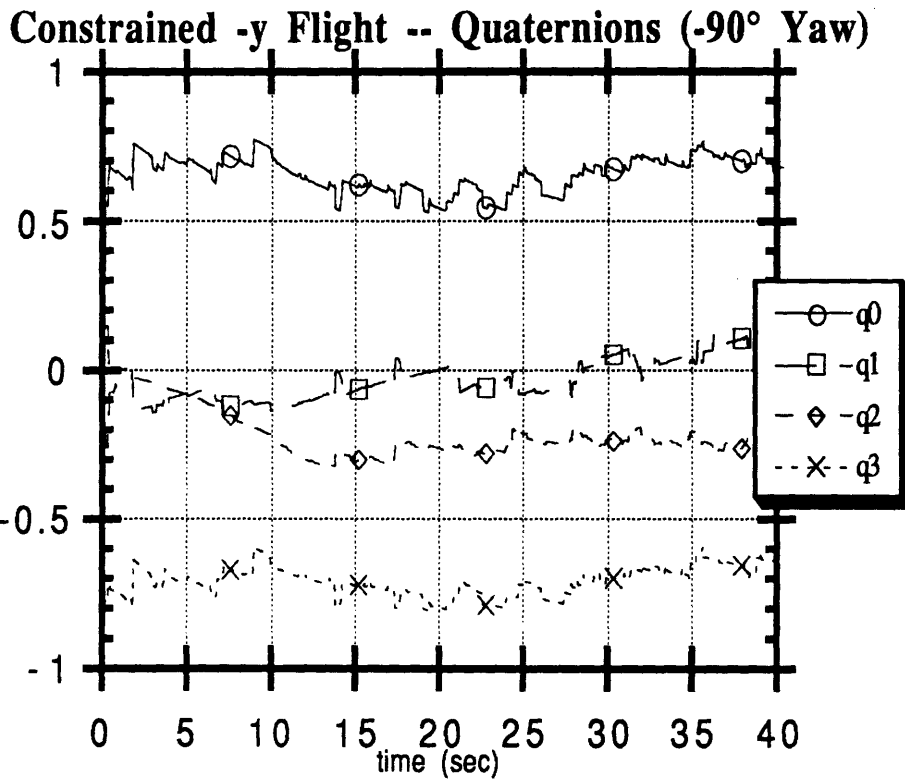


Figure 41. Constrained (-y) Flight -- Thumper 0-3 Unblocked Ranges

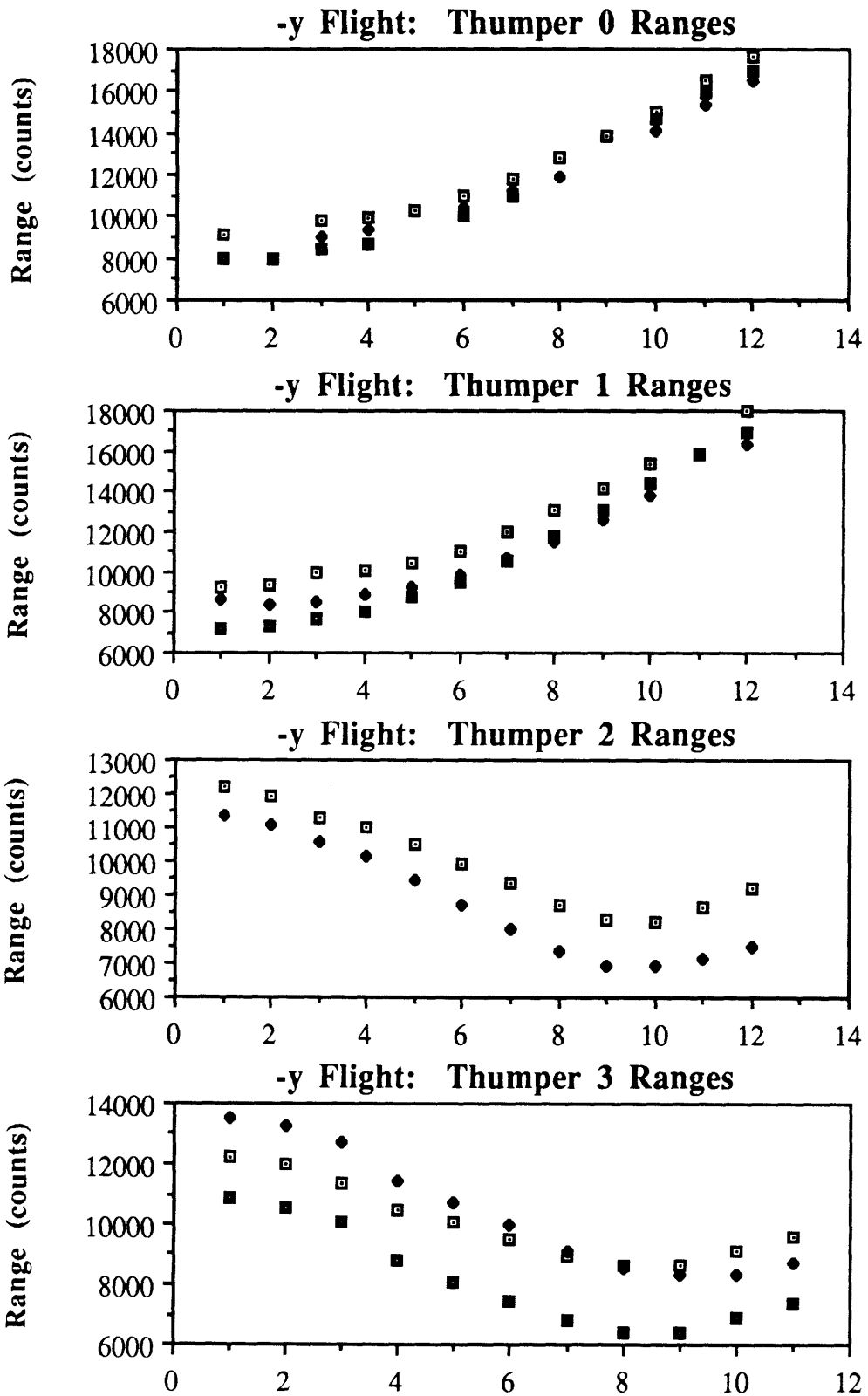
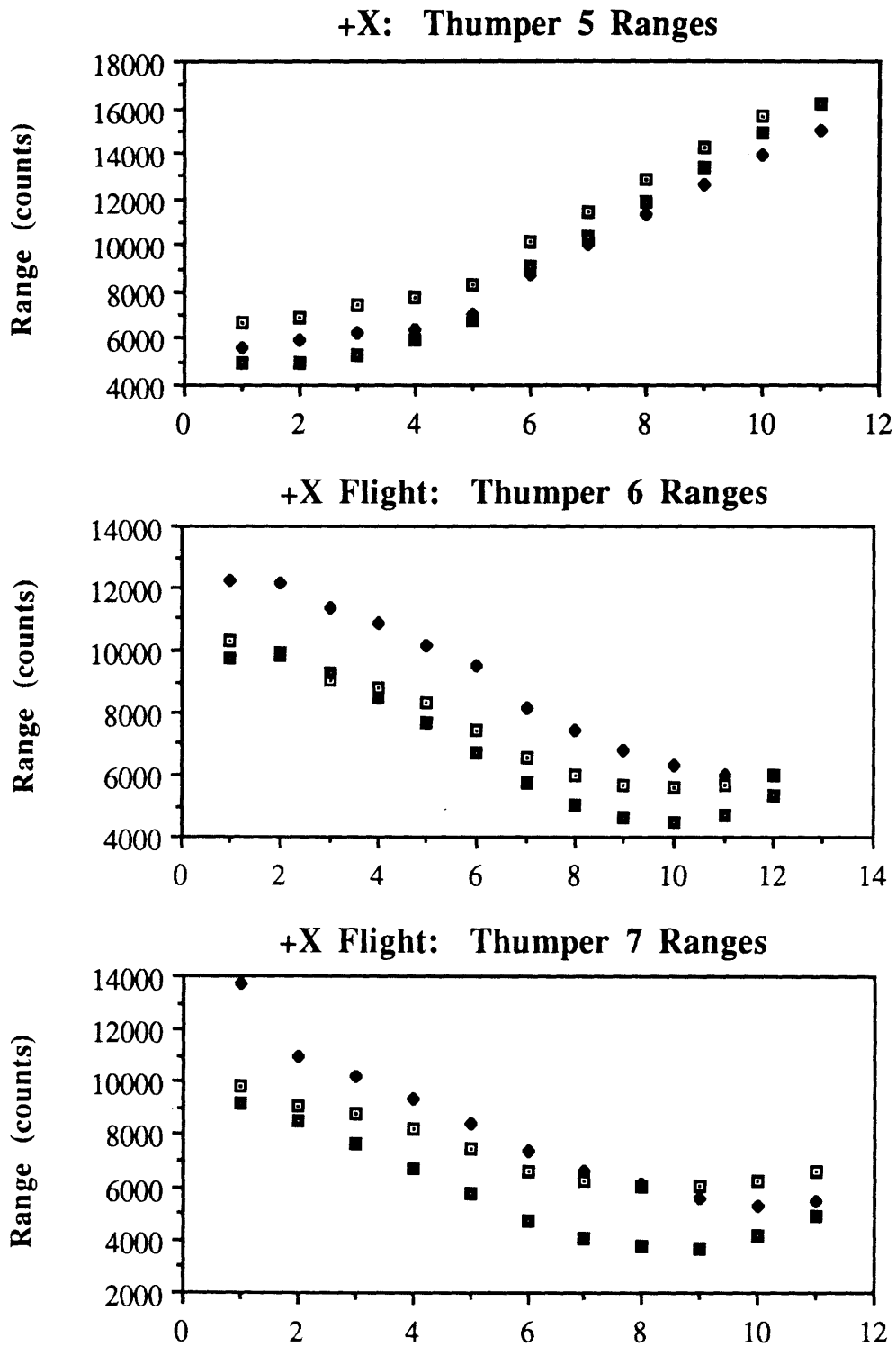


Figure 42. Constrained (-y) Flight -- Thumper 5-7 Unblocked Ranges



5.3.3 Free-Flight

To assess the state calculation performance during position maneuvers only, several flight tests were performed in which MPOD was kept at an upright attitude. Straight and level flight was achieved through good balancing and minimal use of motor torque commands. Also, because vehicle control is most intuitive from a straight and level attitude, this orientation was desirable for tests conducted through pilot flight with hand controllers. After completion of the constrained tests, straight and level flight was performed with only pilot control. Several flights were performed, including docking to the underwater target, flying along the x and y axes, and flying straight and level with yaw.

Figures (43) - (49) show the results of an MPOD docking run. The positions move in the expected pattern, with the x and y positions levelling off near the docking site. Because data collection stopped due to *Obi-Wan* NOVRAM overflow before the final satellite docking was completed, no precise comparisons of known target position and MPOD state calculations are possible. The final levelled y-position of MPOD, $y=2.5$ m, agrees with the measured target y-location. However, the x-estimate of 5.0 m is too large by approximately one meter from the expected docked value. As with the depth value in the constrained -y flight, the docking run z value increases to greater than expected near the end of the run. With multiple runs producing the same problem, it may be suspected that some consistent calculation error is occurring. This could be due to either filter parameters or some other software calculation which has not yet been discovered.

The attitude during the straight and level docking run should approximately hold the quaternion coordinates (1,0,0,0), indicating alignment of body and inertial axes. Although the actual flight cannot be numerically compared with this value, the results show some general agreement with this trend. During the Sunday MPOD tests, the balancing was such that MPOD wanted to roll slightly to one side. The quaternion elements and angular velocity plots document accidental roll during the supposed straight and level flight. The angular velocity plots in pitch and roll depict brief attitude commands dictated by the operator to correct for attitude drift during the docking run.

Figures (48) and (49) show ranges during the docking run. Note the convergence upon the target location, apparent in the ranges by the levelling of range value slope. The final horizontal "line" of ranges for each thumper roughly corresponds to a straight and level MPOD preparing to dock.

After completion of the straight and level flight tests, single-axis attitude maneuvers were incorporated into flight. Figures (50) - (54) show the results of -x cross-pool flight with an intermediate roll maneuver. The x and y positions follow the correct MPOD

Figure 43. Docking Run -- Positions

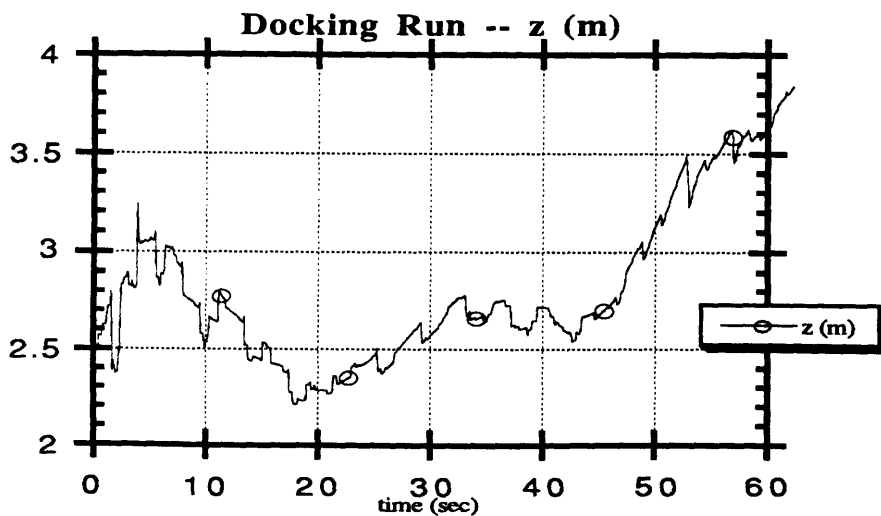
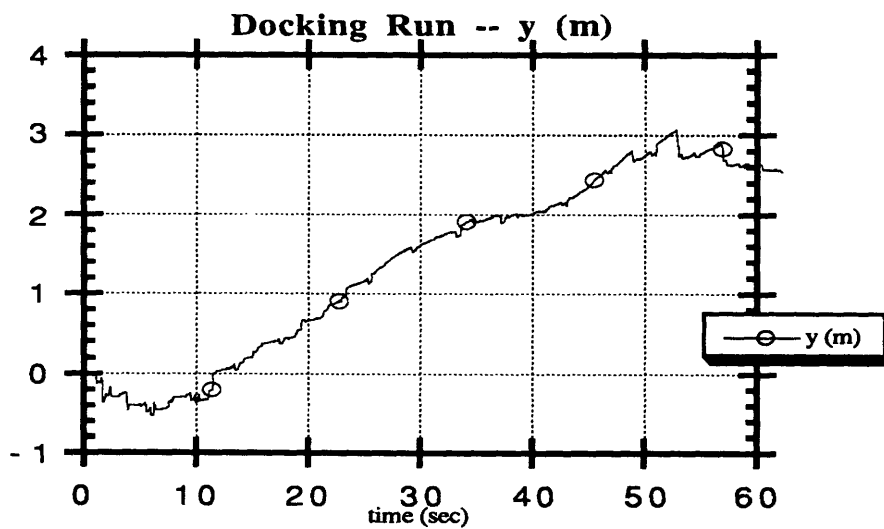
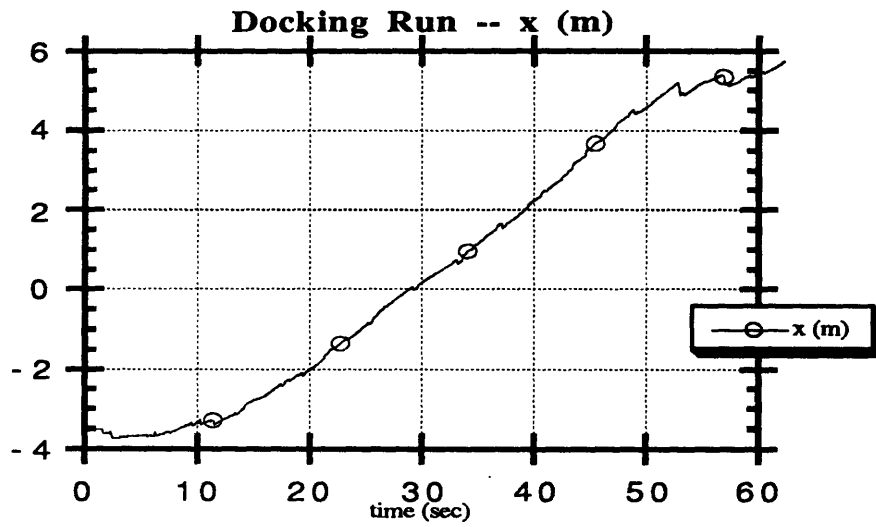


Figure 44. Docking Run -- Linear Velocities

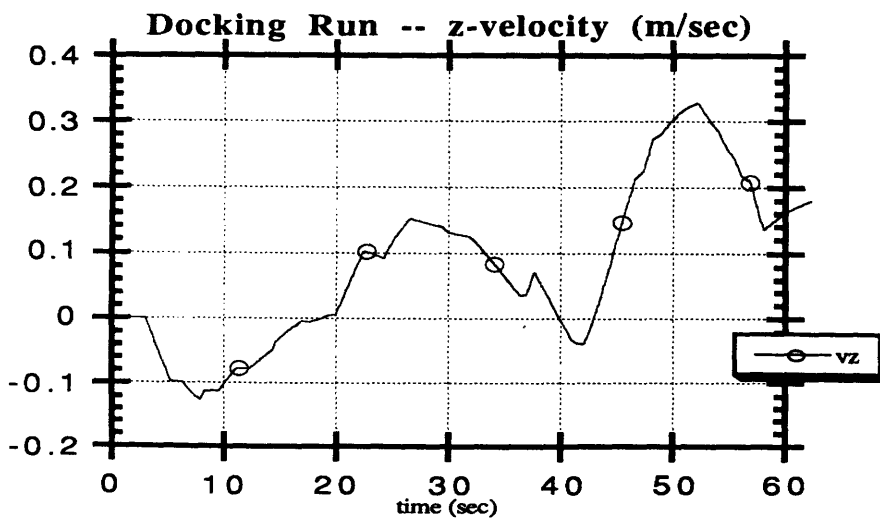
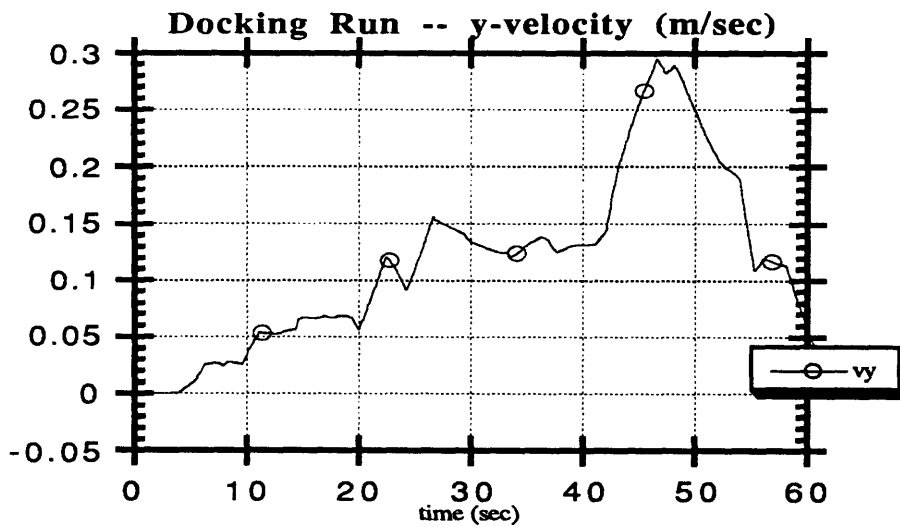
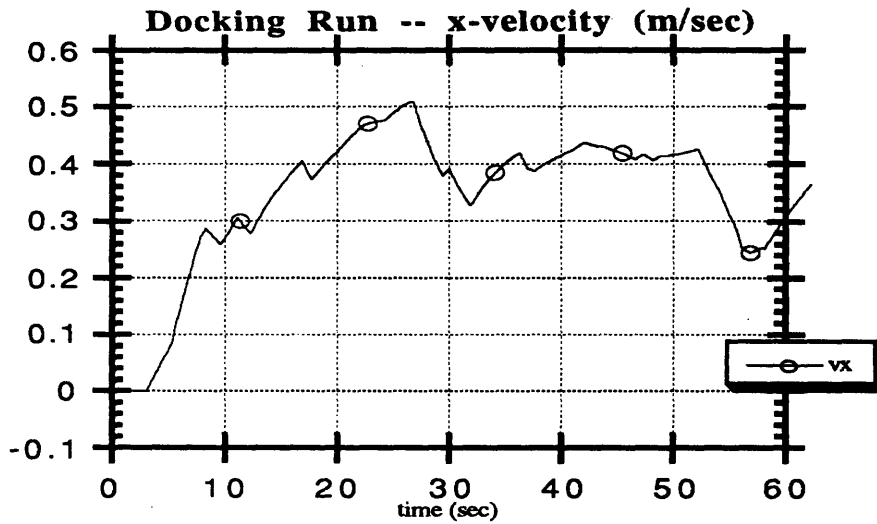


Figure 45. Docking Run -- Attitudes (q0 & q1)

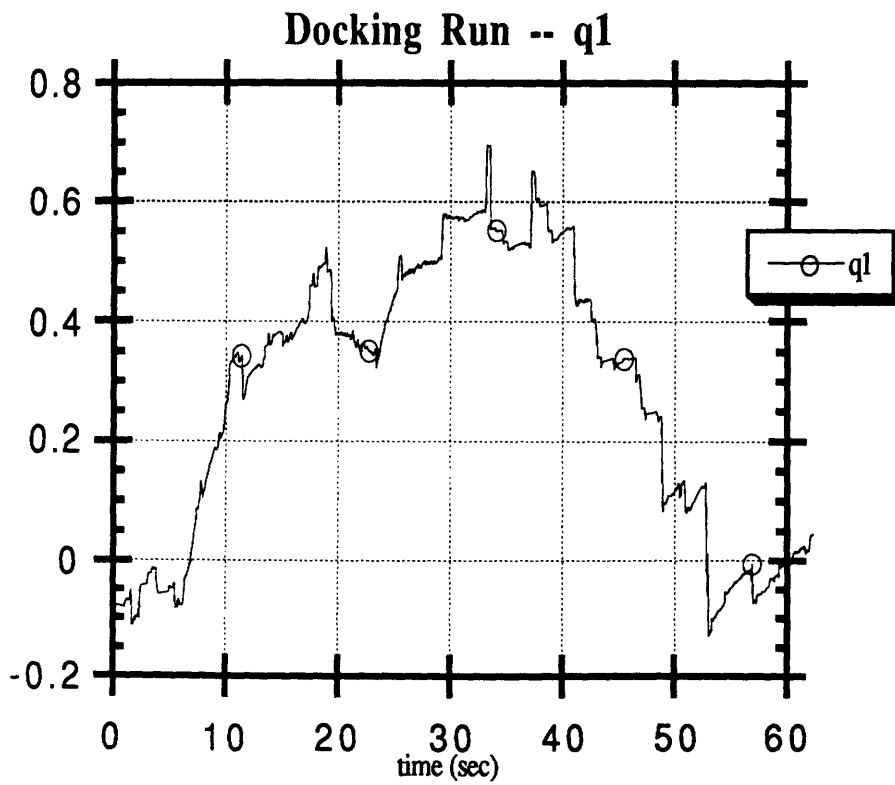
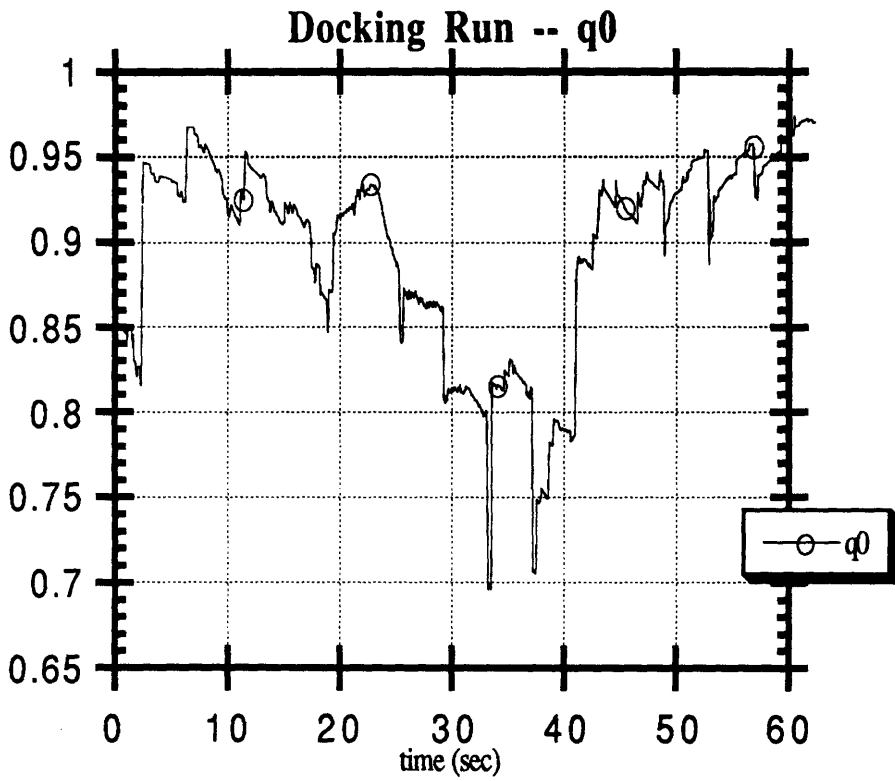


Figure 46. Docking Run -- Attitudes (q2 & q3)

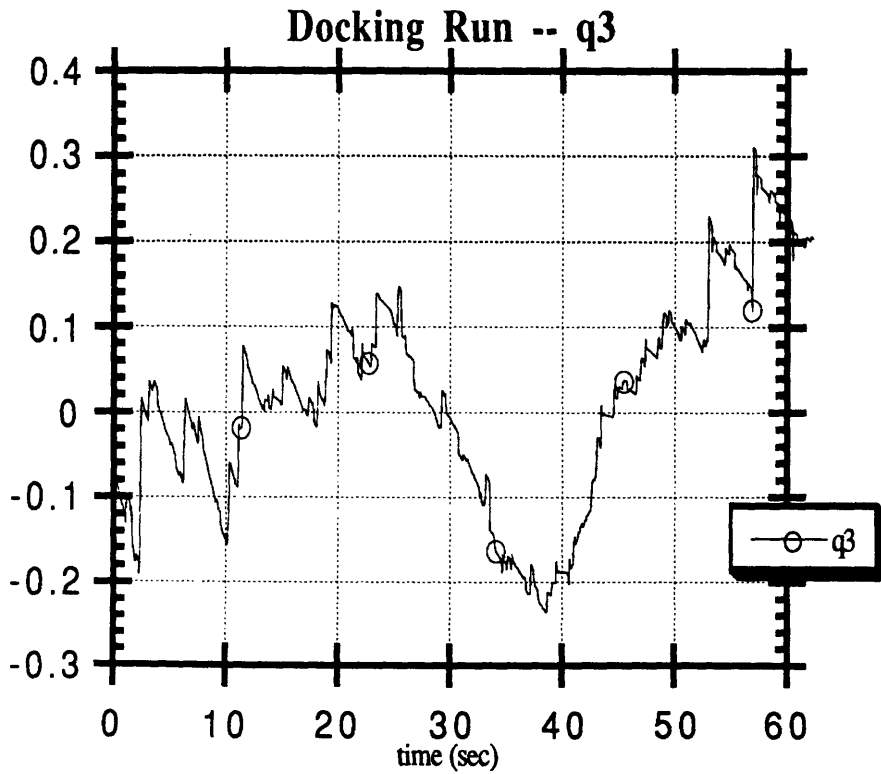
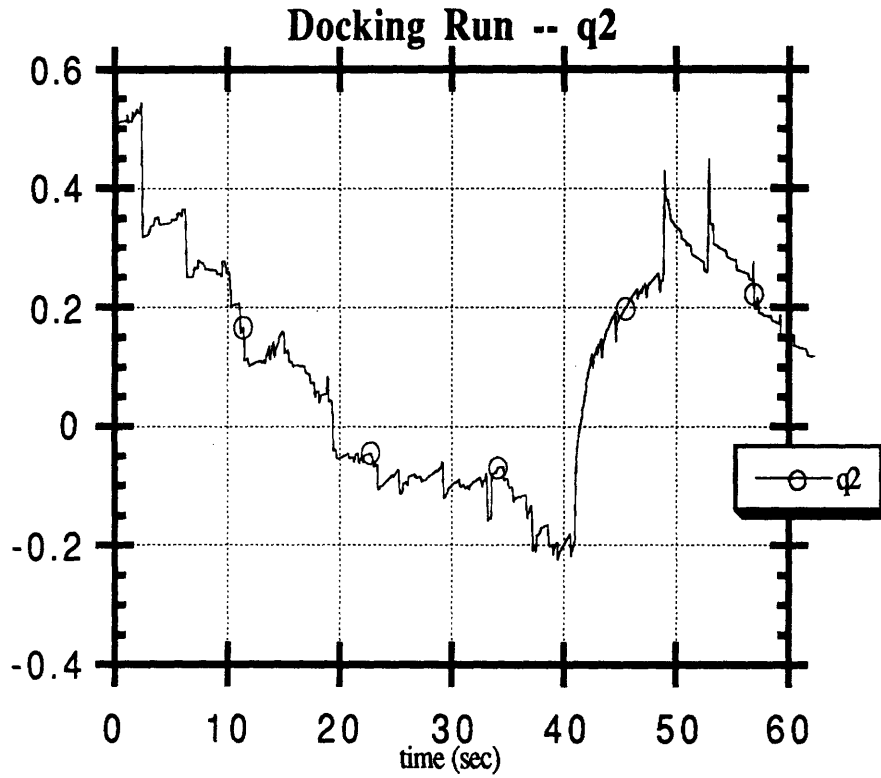


Figure 47. Docking Run -- Angular Velocities

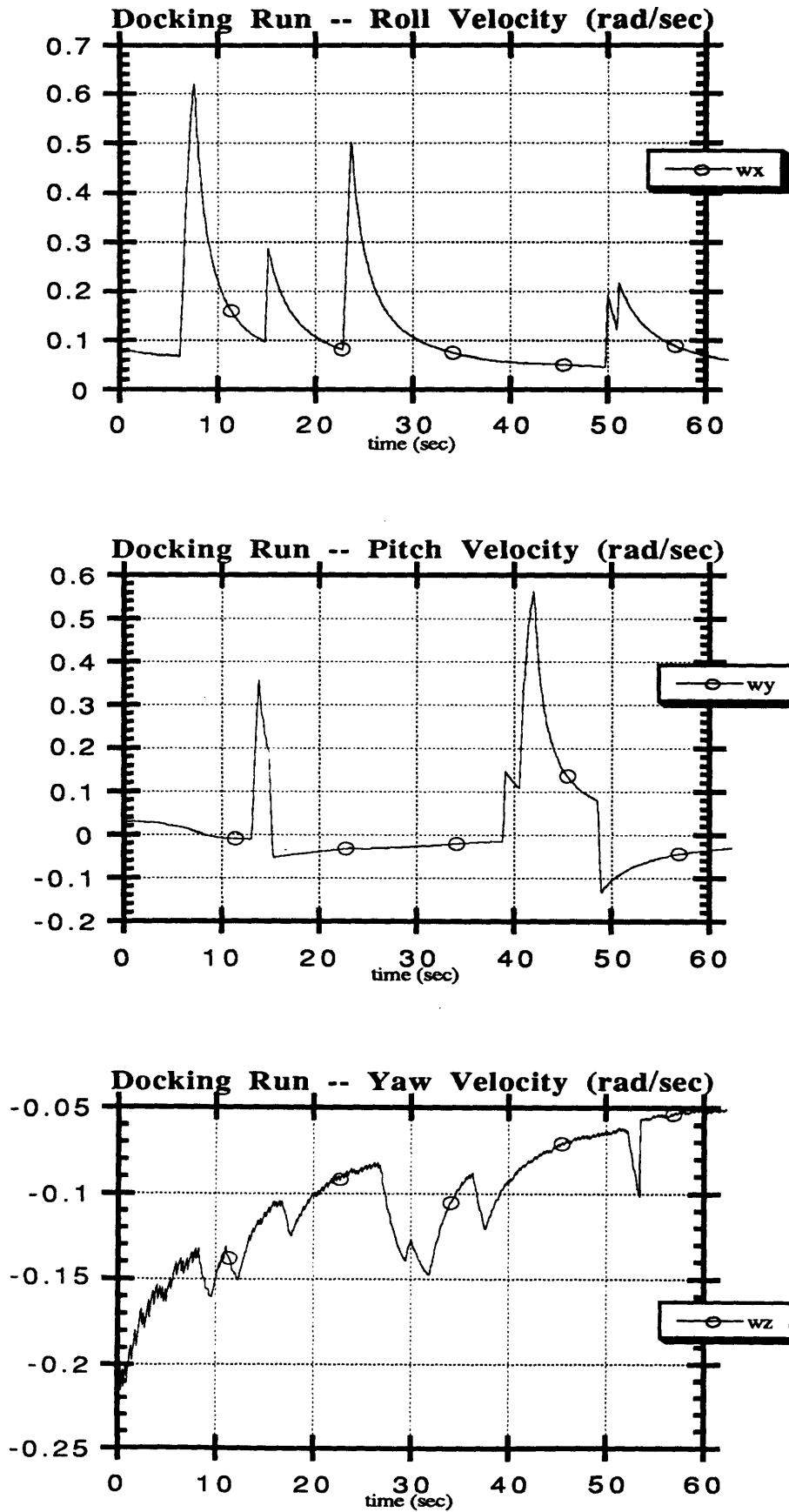


Figure 48. Docking Run -- Thumper 0-3 Unblocked Ranges

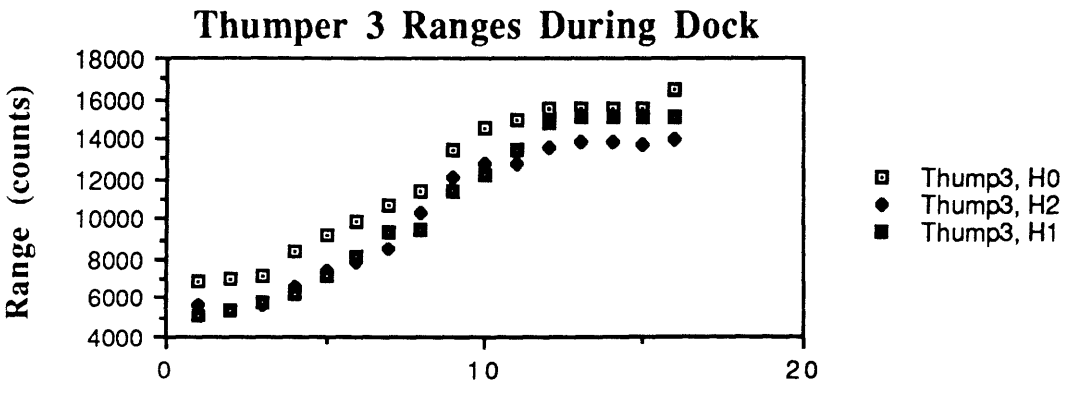
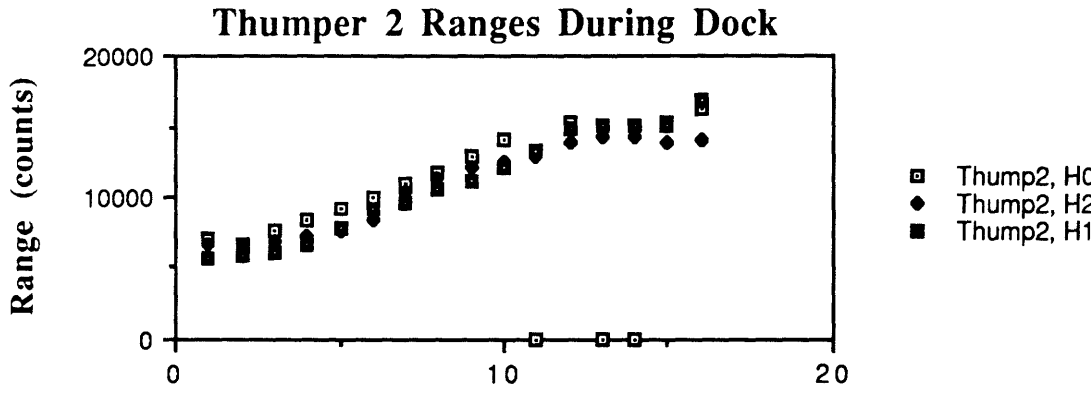
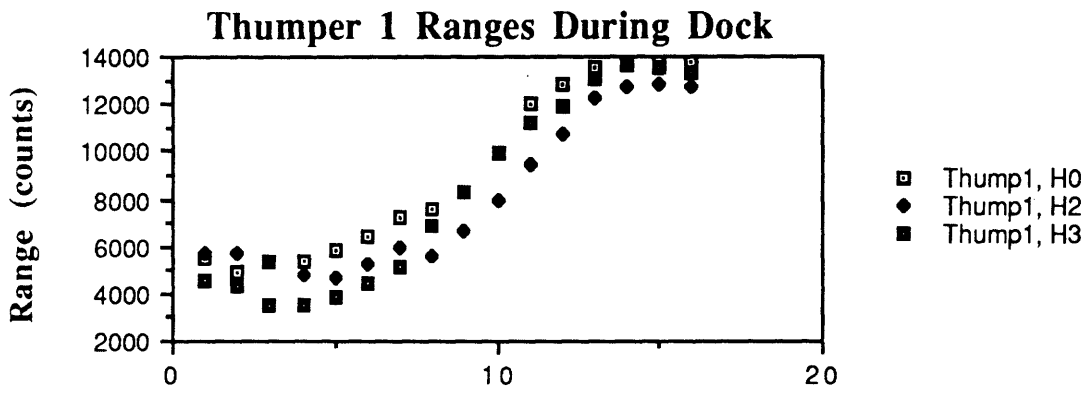
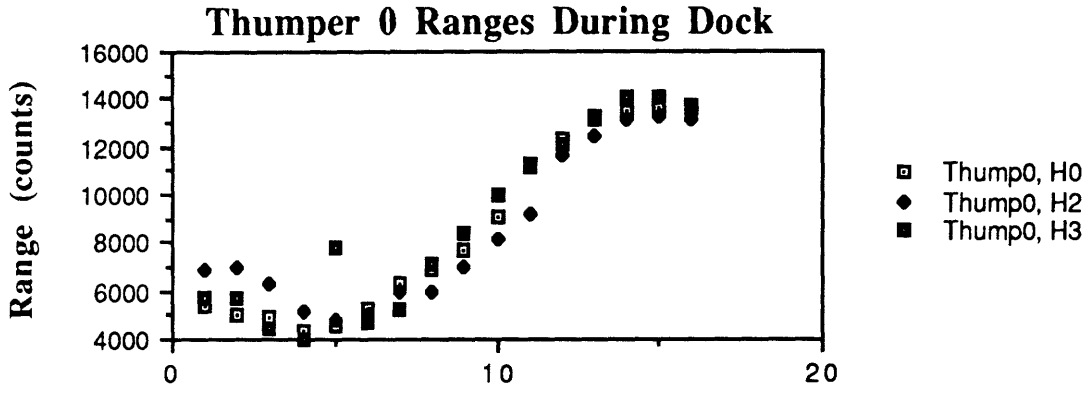
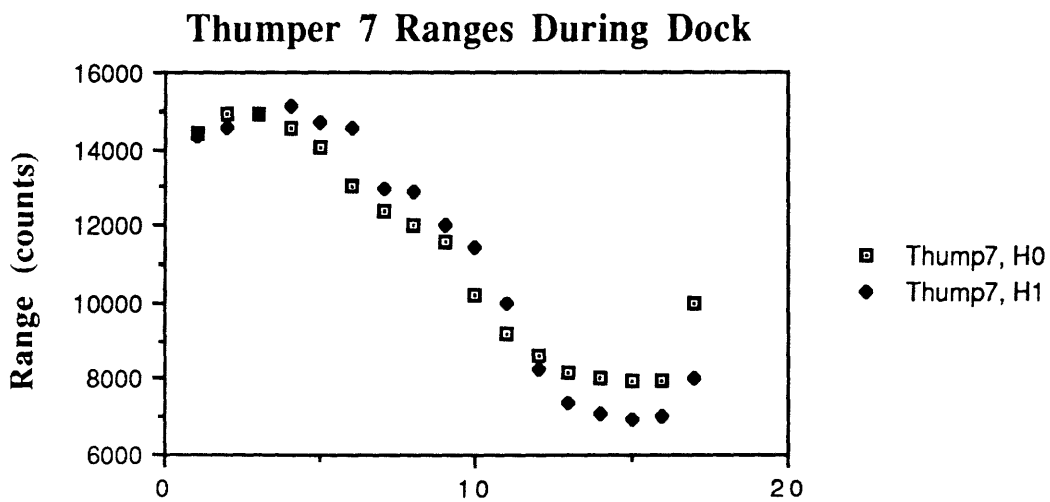
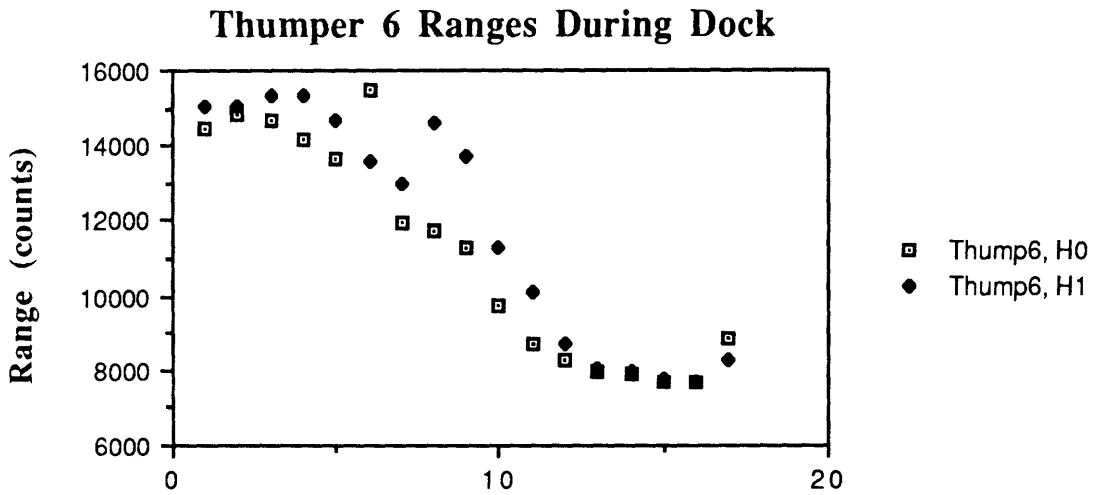
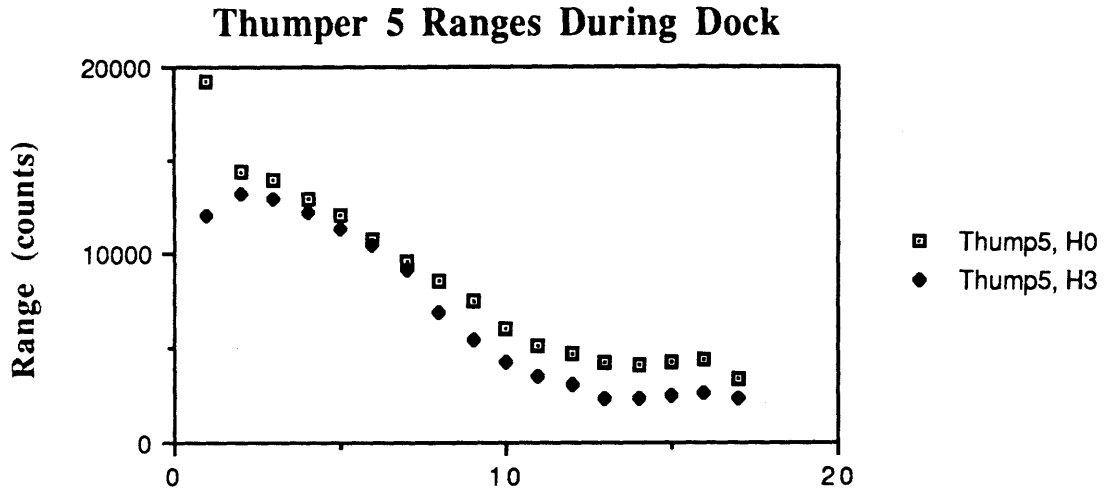


Figure 49. Docking Run -- Thumper 5-7 Unblocked Ranges



motion pattern, with x decreasing as a function of time, and y located near the center of the coordinate system. The z plot shows the characteristic increase in estimated depth near the end of the data run. During the first 30 seconds, the z position is located between 1.8 and 2.0 m. This is a reasonable estimate due to the fact that the roll maneuver was performed near the water surface to prevent hydrophone contact with the pool floor. However, the z estimate then depicts an increasing depth which may not have occurred.

The quaternion plots are shown in Figures (52) and (53). There are three distinct components to the plots, all of which agree with expected results. Because MPOD is flying in the $-x$ direction, the initial attitude corresponds to a 180° yaw angle, or quaternion value of $(0,0,0,1)$. Then, at a time of 20 seconds into the run, the roll maneuver is initiated. At a time of 40 seconds, the roll maneuver concludes. Then, MPOD yawed such that the vehicle and body axes are aligned, corresponding to a quaternion value of $(1,0,0,0)$. The angular velocity plots shown in Figure (54) shows the roll and yaw velocities corresponding to the MPOD attitude maneuvers.

The final test results presented in this thesis were conducted solely for attitude analysis. Figures (55) - (59) show the state estimation results. The test run was begun from a near-central pool location to prevent pool wall or floor contact during the inevitable vehicle drift. Torque commands dominated the translational force commands applied to the vehicle. For comparison purposes, an attempt was made to keep the maneuvers simple enough to allow a visual record of maneuvers which could later be compared with the state estimations. Therefore, large attitude maneuvers were performed one axis at a time, starting with roll, then pitch, and ending with yaw.

The position and velocity plots are shown in Figures (55) and (56). Note that the x and y positions are near the pool center, while the z position indicates a near constant depth after the first 10 seconds. Note that the z and z -velocity plots do not show their characteristic increase near the end of the run. This suggests that the z -computation problem involves either translational movements or z -thrust commands.

Quaternion and angular velocity plots are shown in Figures (57) - (59). The angular velocity plots show the roll, pitch, and yaw maneuvers in the sequence they were performed. Each was done for approximately 20 seconds, with the yaw maneuver in its final phases as data collection ended. The quaternion plots are difficult to analyze, but the rapidly changing q_0 plot indicates complex attitude motion, while the q_1 , q_2 , and q_3 plots show curves corresponding with commanded roll, pitch, then yaw maneuvers.

Figure 50. Flight with Roll -- Positions

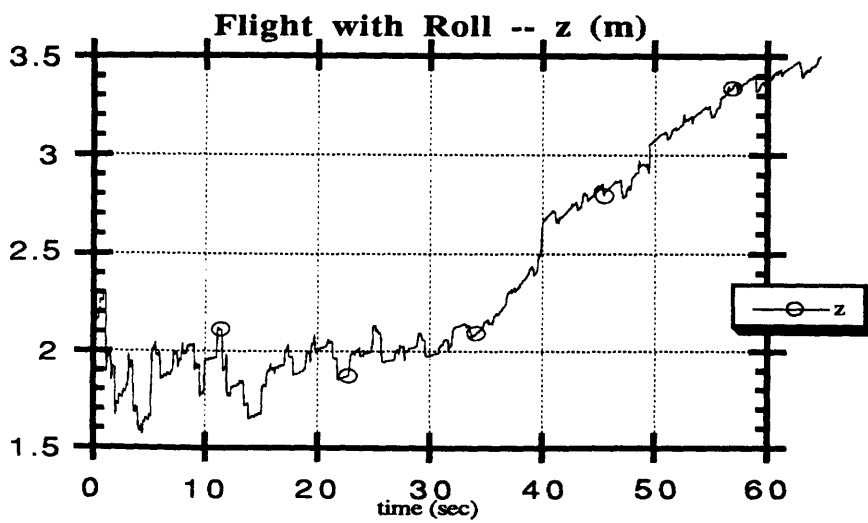
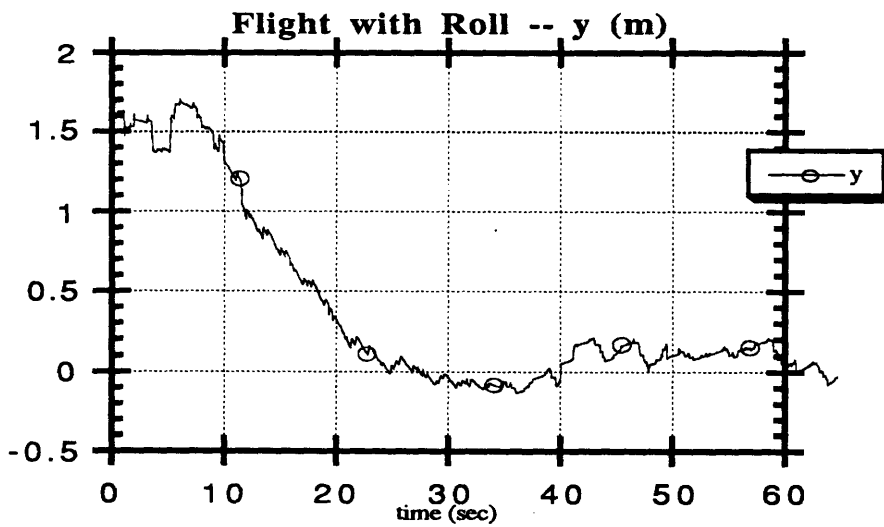
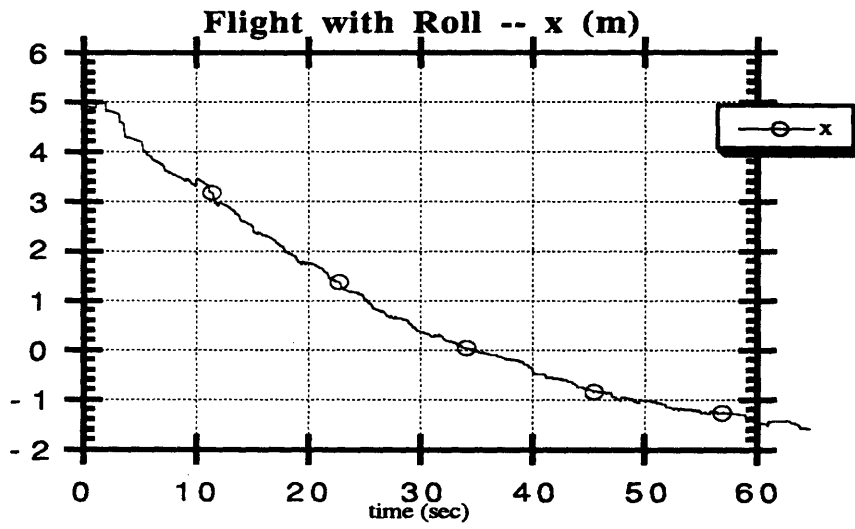


Figure 51. Flight with Roll -- Linear Velocities

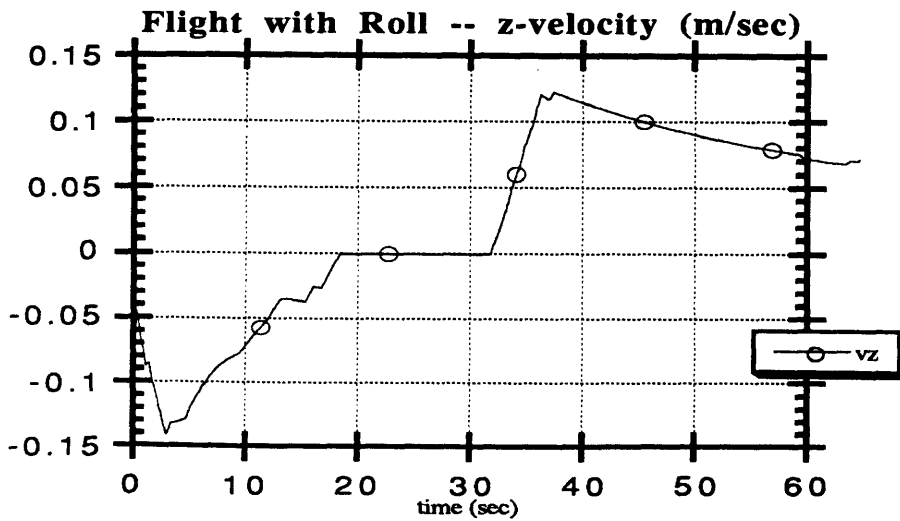
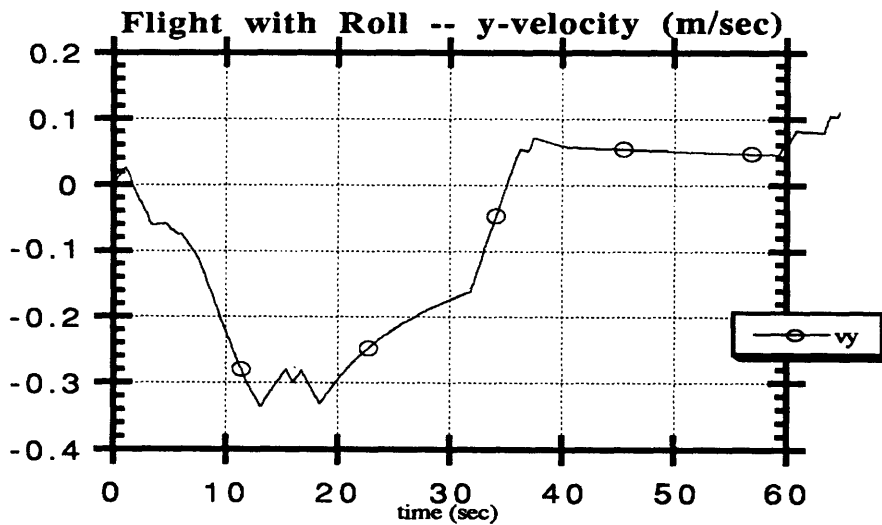
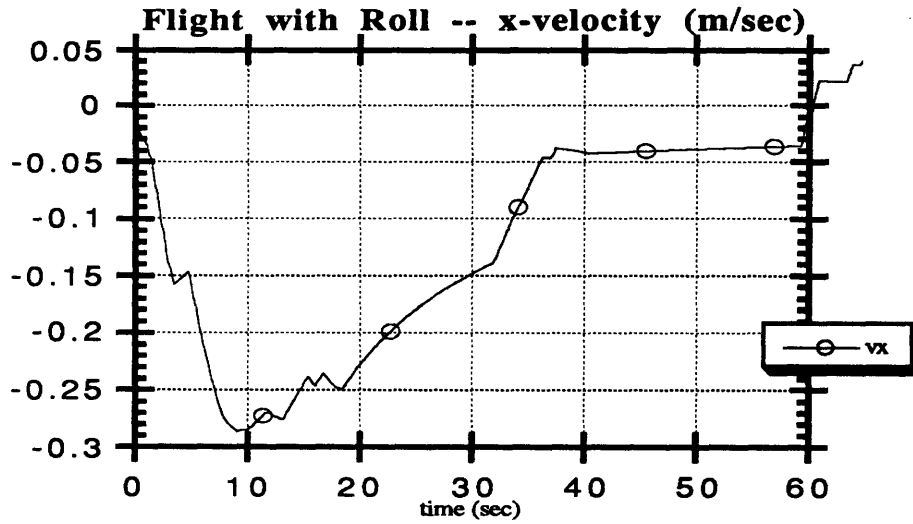


Figure 52. Flight with Roll -- Attitudes (q_0 & q_1)

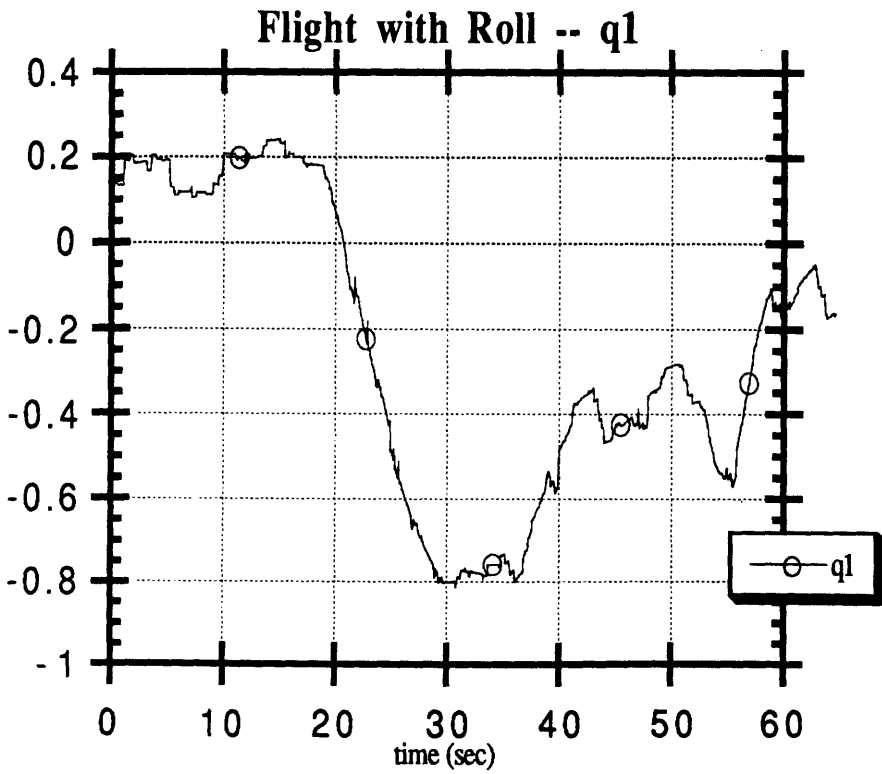
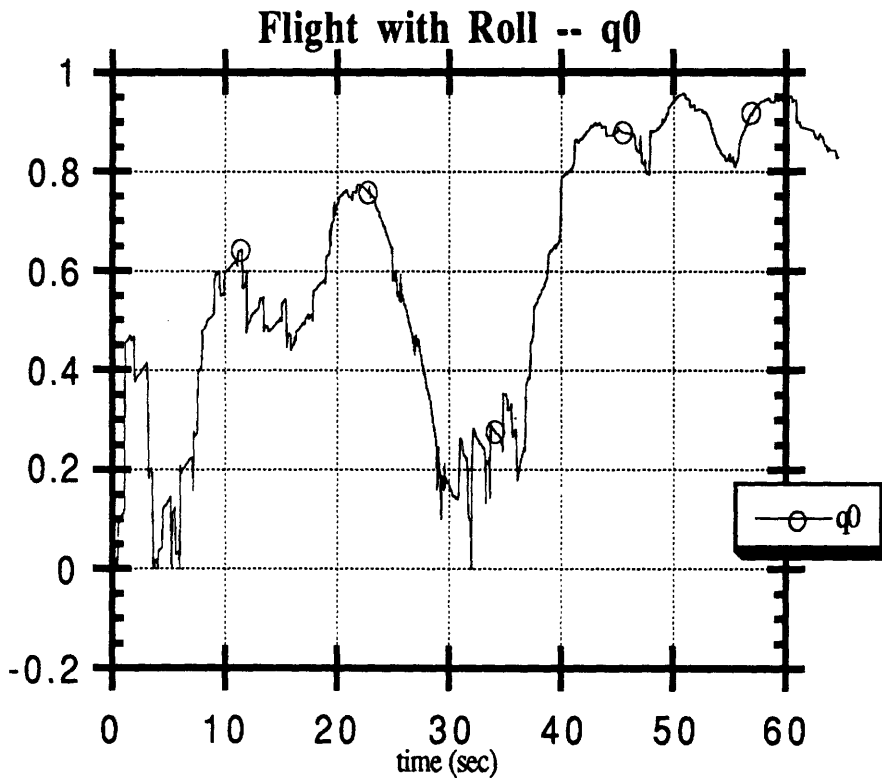


Figure 53. Flight with Roll -- Attitudes (q2 & q3)

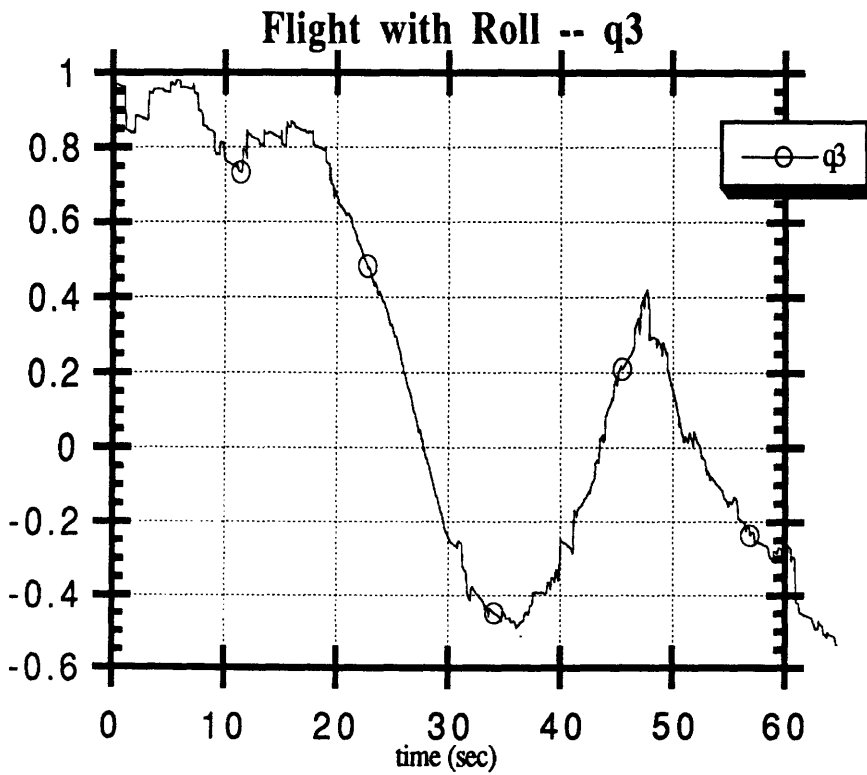
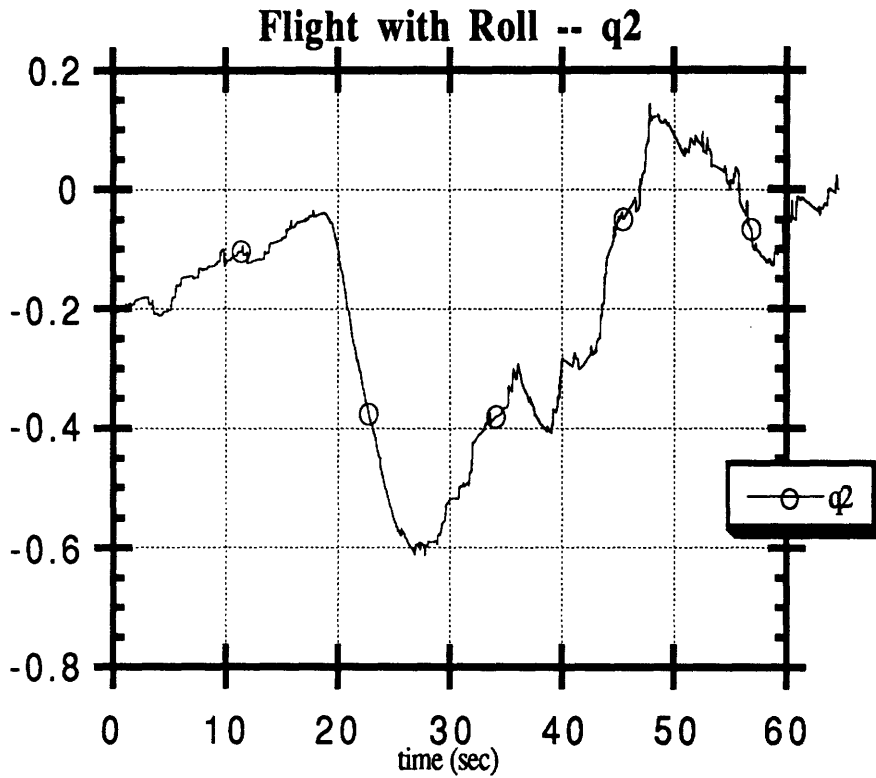


Figure 54. Flight with Roll -- Angular Velocities

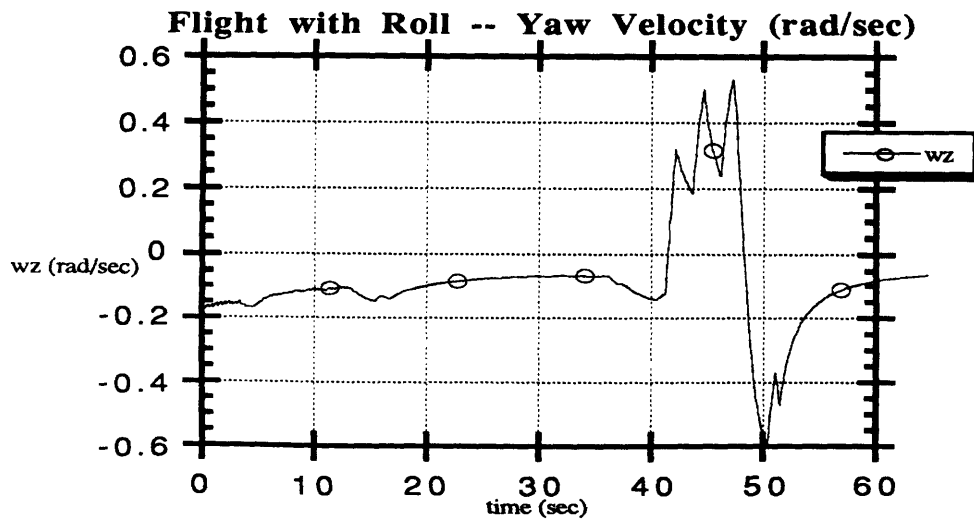
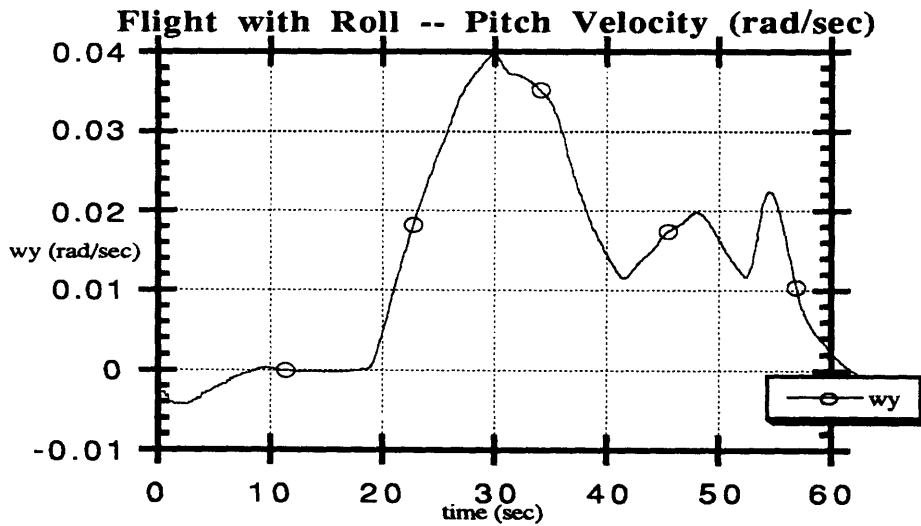
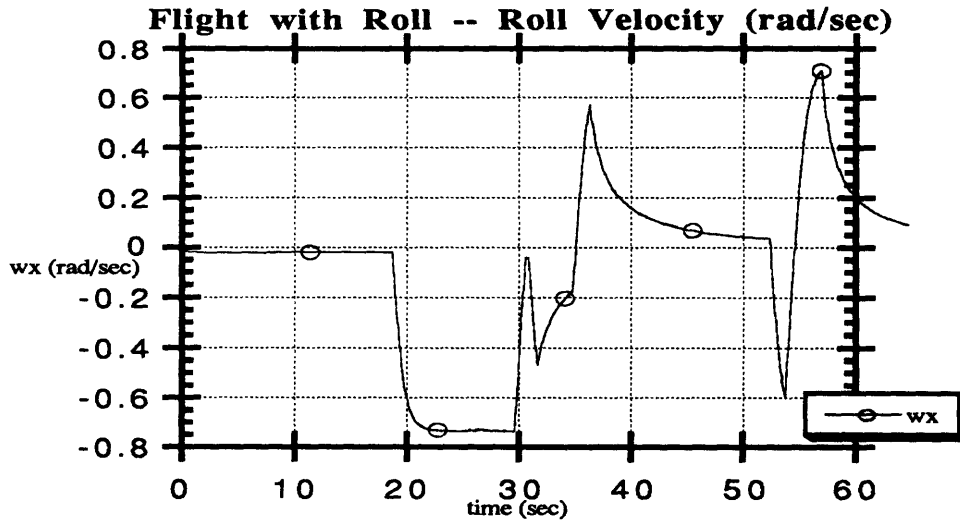


Figure 55. Attitude Maneuvers -- Positions

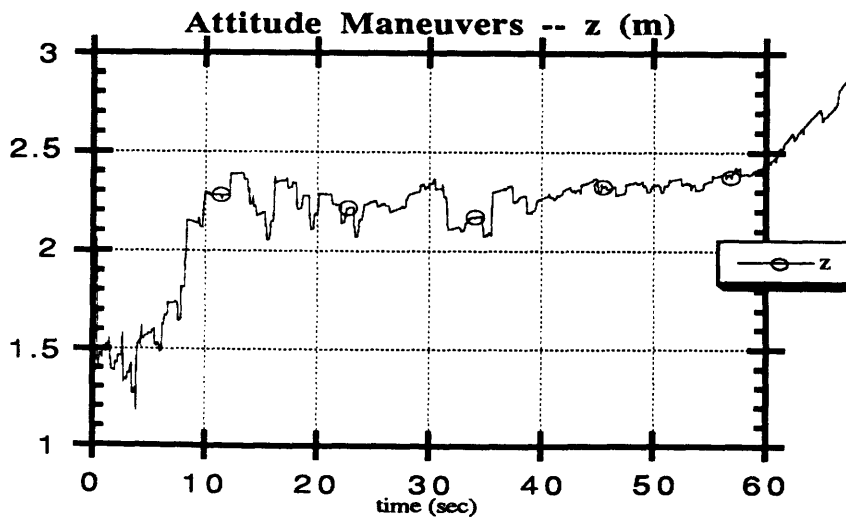
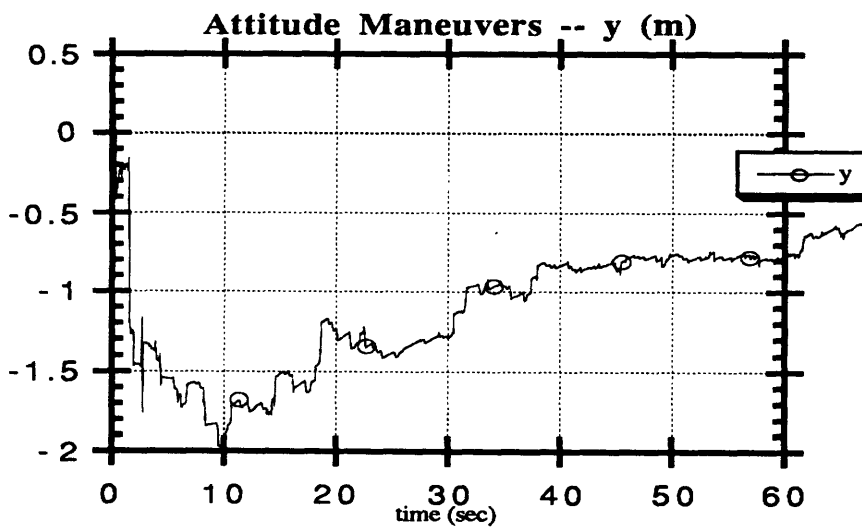
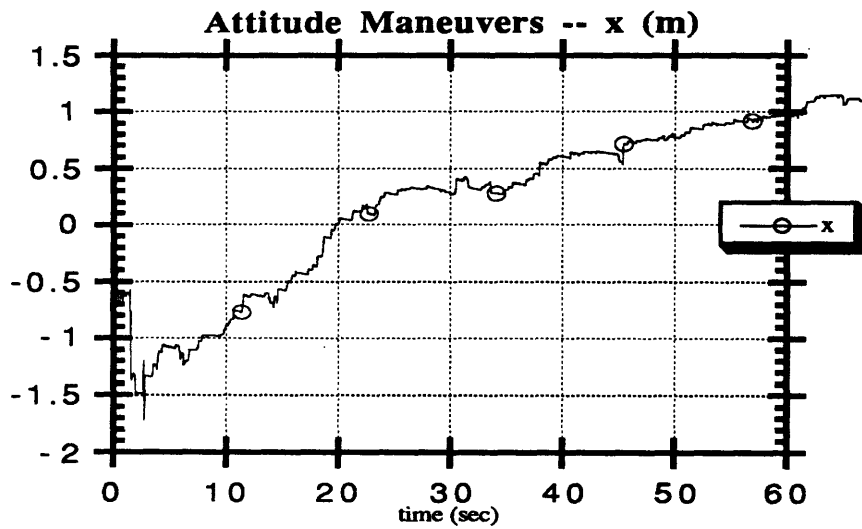


Figure 56. Attitude Maneuvers -- Linear Velocities

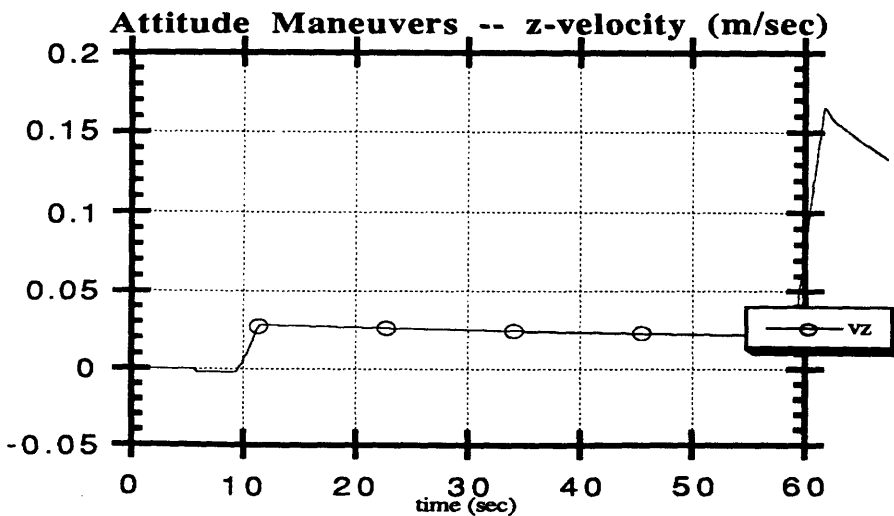
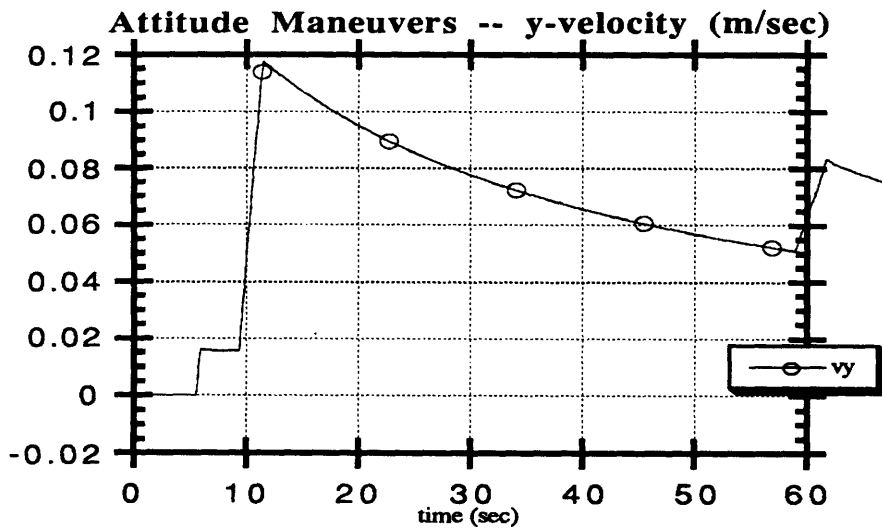
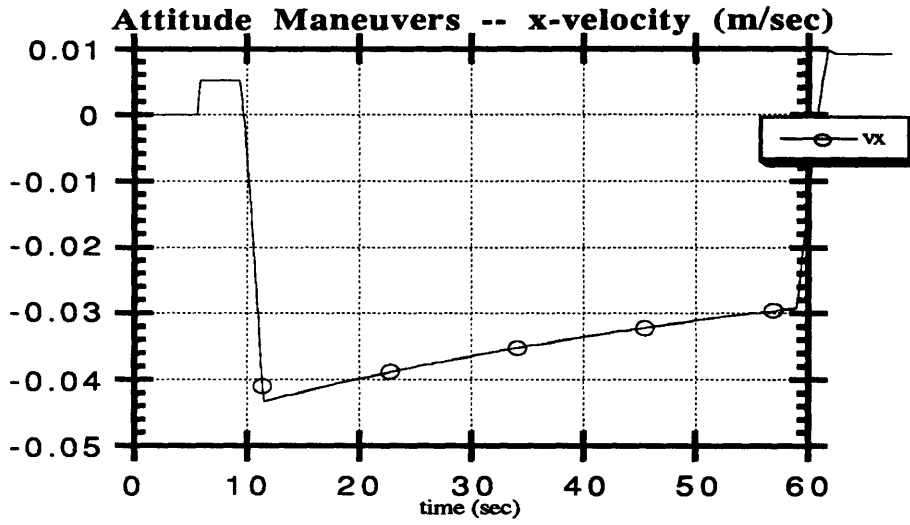


Figure 57. Attitude Maneuvers -- Attitudes (q0 & q1)

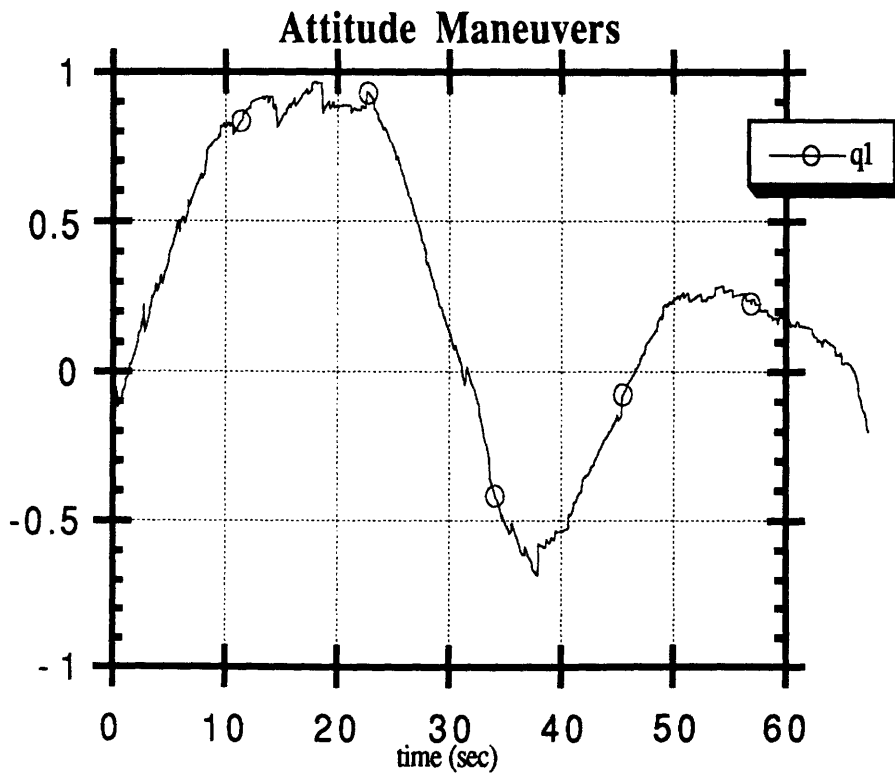
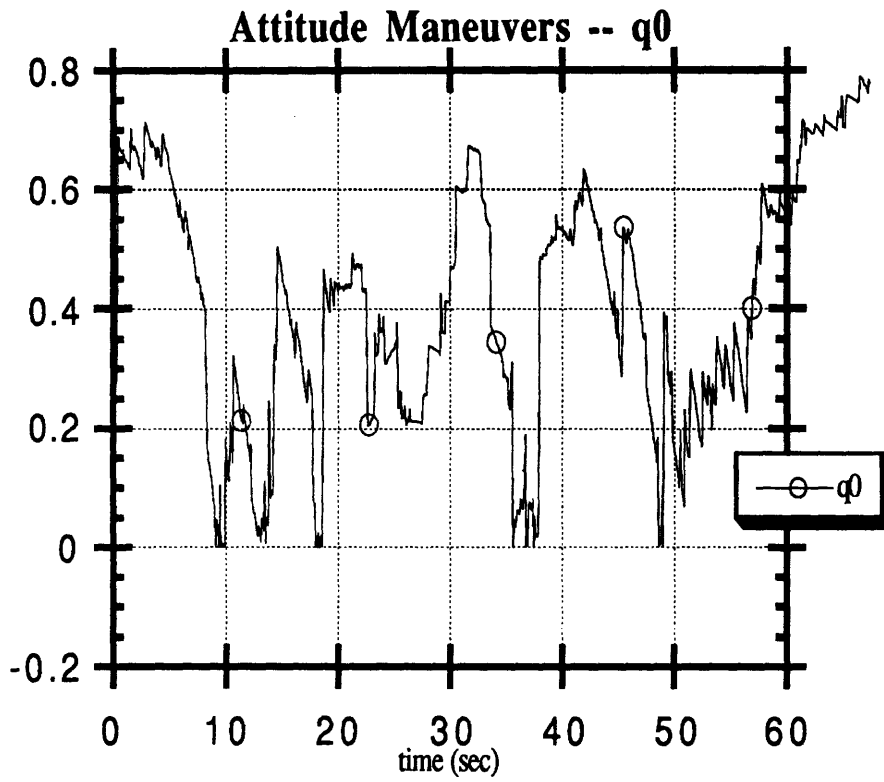


Figure 58. Attitude Maneuvers -- Attitudes (q2 & q3)

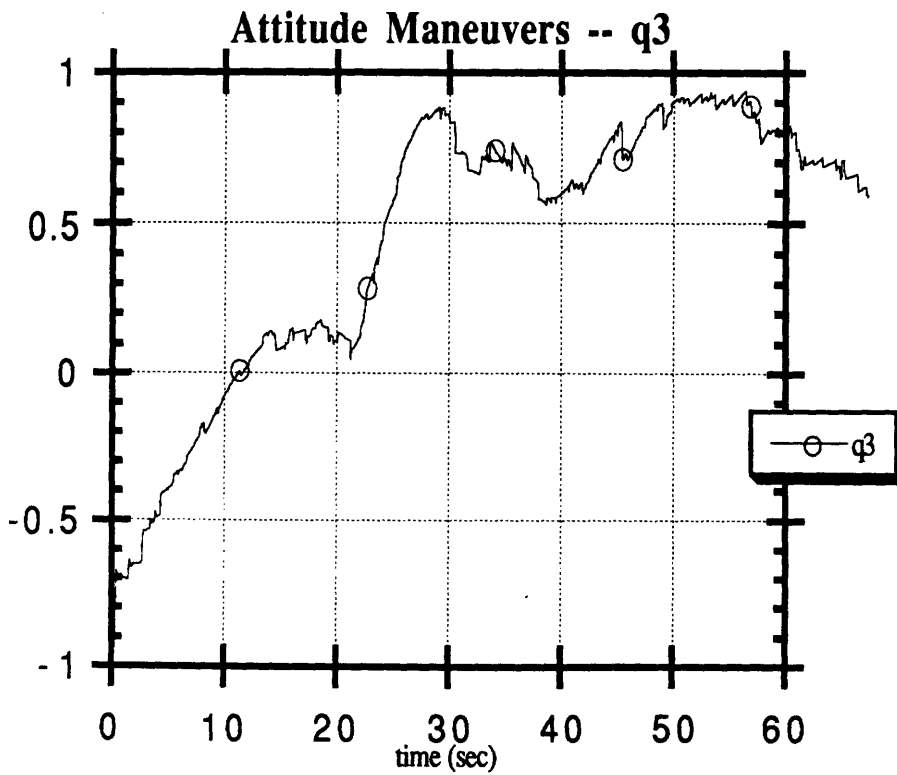
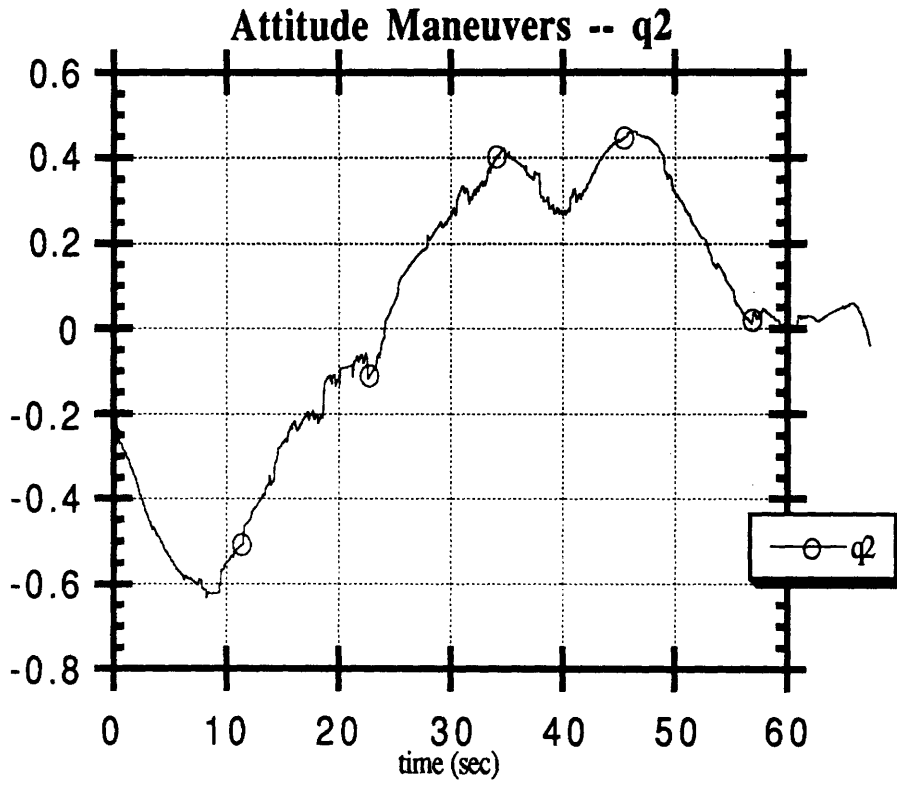
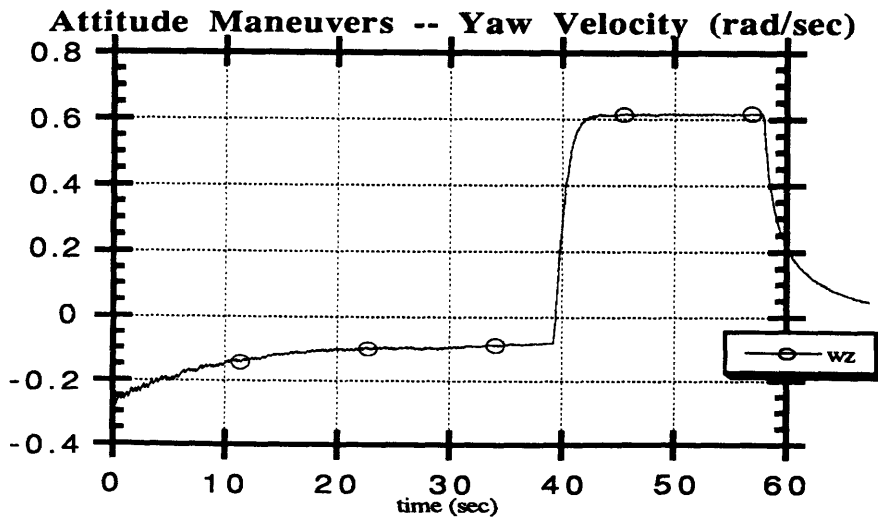
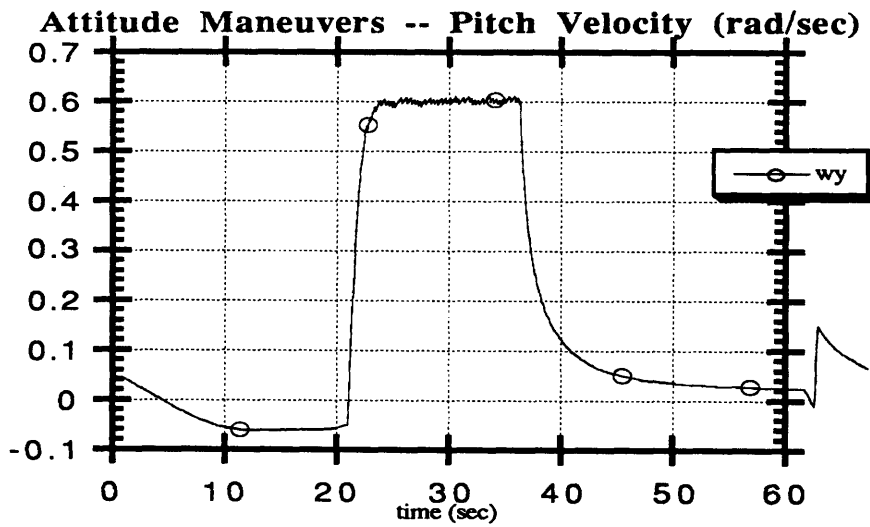
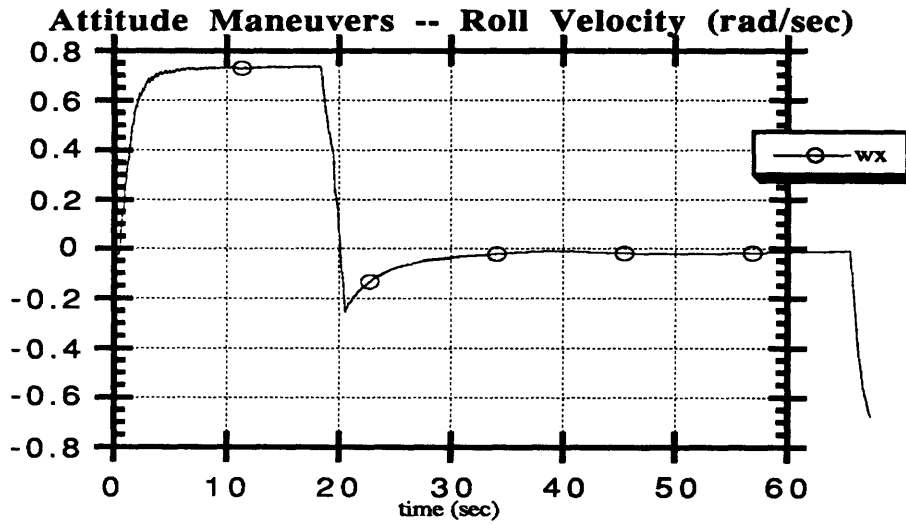


Figure 59. Attitude Maneuvers -- Angular Velocities



6.0 Conclusions and Recommendations

6.1.0 System Accuracy and Robustness

6.1.1 Hardware

The MPOD hardware systems were much more reliable than during past generations of MPOD testing. After the initial debugging tests, the computers and computer interface circuitry systems performed flawlessly. The main key to hardware debugging became knowing each subsystem's weakness. For example, an MPOD motor failure indicated either failure of the motor or of an 11028 power transistor. Pneumatic leakage from the solenoid box was always traced to a faulty jettison cylinder on the docking probe.

3DAPS problems plagued MPOD for a period of months. However, a permanent solution was devised and implemented for each failure mode. Sequencer communication problems were solved by mounting the fiber optic receiver chip directly on MPOD's control box. Electrical noise on the amplifier circuitry was minimized by removing the circuit from MPOD's control box and isolating its power supply from the MPOD control power.

Working electronic systems required no maintenance. However, the software was often changed due to modifications in state calculation parameters and additions to the control system. So long as MPOD's control box was open, reprogramming the AMPRO computers was relatively simple. However, compiling on MPOD computers was inefficient due to the absence of hard disk drives. Fortunately, all of the single-board computers could be attached to a floppy drive for executable file transfer. The software was compiled and stored on *Luke*, the control station computer.

6.1.2 State Calculation

During static simulation tests, the state estimator has been shown to always converge to the correct value when the actual static value is approximately the same as the filter's initial guess. However, when the state estimator's guess is far from the actual state, the state estimator sometimes converges to a wrong answer. Varying the initial covariance values has been shown to have an impact on state convergence, but more tests are required to determine a concrete relationship between filter parameters and state convergence.

The dynamic simulation tests showed that the x and y positions and velocities were tracked well; however, z position and velocity were not nearly so accurate. The quaternion

and angular velocity state estimates showed good agreement with simulated values, with the exception of offsets produced by the non-zero simulated yaw rate null offset.

Pool tests also revealed a problem with z position and velocity calculations. Although the error source has not yet been found, it is suspected that a software bug is the culprit. The pool test x and y positions usually agreed with the expected results. Quaternion elements and angular velocities also agreed reasonably well with expected results. It is important to remember that all pool test results are qualitative only, due to the inability to know exact underwater position during dynamic flight.

The MPOD underwater testing could have been made more ideal in many ways. First, the sensor and state data was saved too often, resulting in limited test runs of approximately a minute in duration. After that point, the NOVRAM would ignore data writing commands. Because MPOD motion was begun relatively quickly after data saving, there was a minimal motionless transient during dynamic tests. In this manner, most runs were completed to satisfaction. MPOD could fly across the pool in this time, or perform a series of three attitude maneuvers, as described in the previous section. However, data saving at such a frequent rate (approximately 20 points / sec) was not necessary. A data acquisition rate of ~2 Hz should be sufficient to examine MPOD dynamics accurately.

Another non-ideal factor in the April 21-22 pool test experiments was the running of 3DAPS at the relatively slow rate of approximately 3 seconds / iteration. The system has the capability of running faster and should be used to capacity. The static tests were not affected, due to the similarity of ranges at each point. However, fast dynamic range updates are imperative for position and attitude. It is a tribute to the state estimation software that it performed as well as it did, considering the slow 3DAPS update rate.

One of the most major problems with the underwater dynamic testing of the MPOD state calculation was the inability to know precise MPOD attitude or position at any time. For future testing of the state calculation software, it would be advantageous to devise a method for constrained attitude and position flight, with an accurate method of determining the vehicle path for comparison. For straight line translation, MPOD might be attached by a rope to a pulley system, as described in the vehicle parameter determination tests in Reference 27. With some sort of rails guiding MPOD in a straight line, pulley encoder positions could be used to determine exact MPOD position as a function of time. The only suggestion for constrained attitude maneuvers is somehow attaching MPOD along its axis of rotation to a fixed line which is free to rotate, but rigid in translation. The vehicle central position may be measured without the vehicle in place, then by knowing the locations of line attachment to MPOD, the exact axis of rotation may be determined. Although this

attitude test sounds good on paper, this thesis presents no practical way of implementing the described constraint procedure.

Modifications may be needed in the state calculation process. One suggestion for improving measurement quality is the identification of blocked 3DAPS ranges before the filter is called. For any MPOD position and attitude, approximately 38% of the thumper-hydrophone combinations may produce erroneous ranges due to MPOD blockage. With an accurate real-time state estimate, possible blocked ranges might be identified. A suspect range would then either not be used by the filter or would be passed with a high measurement variance, indicating a possible bad numerical value.

Other modifications will center around intelligent filter parameter determination. The current values were determined by trial and error. Not nearly enough simulations and actual test runs were completed to decide which covariance, state variance, and measurement variance values to use. Only through more extensive simulations will a concrete evaluation of parameter effects be determined.

6.2.0 Future Control and Human Factors Experiments

6.2.1 Implementing the Control System with the State Estimator

One of the first priorities for MPOD should be development of a control system which uses the current hardware and state calculation system to enable automated flight and docking. Due to unmodelled dynamics, it is unlikely that the gains like those determined by MATLAB in Chapter 4 will be optimal. Working gains may be determined by the operator during MPOD underwater test runs. The current software allows the pilot to enter a 16-bit gain, which is then translated into hardware commands. Also, each element of a PID control system may be tested separately, allowing the determination of an unstable control system's faulty component.

It may prove impossible for an operator to determine workable control system gains. If such a case, an adaptive control system might be the most intelligent method of determining MPOD gains. Such a system would allow MPOD to calculate its own gains during assessment of its performance compared with the ideal case.

6.2.2 Pilot Onboard Flight

Although a robust position and attitude control system may seem to make the pilot expendable, it is important to make the vehicle as user-friendly as possible. Historically,

only well-trained SSL pilots could fly and dock MPOD consistently and accurately. However, with the full position and attitude control system, different elements of automated control may be used with the pilot to enhance overall performance. Human factors tests have only been performed with open loop and PID attitude hold tests. MPOD should soon have the ability to provide more helpful functions, such as station-keeping. Although one wouldn't use the full automatic docking capability with an onboard pilot, it would decrease the required pilot skill level to have, for example, an accurate 3-axis attitude hold system during docking runs. Also, the onboard pilot could perform more intricate flight maneuvers than docking and strive for optimal paths during flight.

Another interesting experiment would be to study kinesthetic feedback by blocking MPOD's windshield, having MPOD fly known attitude and position maneuvers, then asking the pilot to describe the MPOD maneuver. Also, MPOD could be used for simulated astronaut rescue, with MPOD flying to the known position of an injured diver, holding its position while the diver climbs onboard and takes the regulator, then either flying itself to the water surface or turning over flight control to the diver. These and many other flight scenarios could be enhanced by an "intelligent" MPOD.

6.2.3 Mounting an Astronaut on the Vehicle

In past Marshall Space Flight Center test sessions, a test subject in a space suit has been mounted on the front of the MPOD vehicle, replacing the docking probe. Flying the vehicle was difficult, due to the loss of video feedback and changes in MPOD dynamic behavior (i.e. altered moments of inertia and different drag coefficients) due to the astronaut's presence. During such flight, a completely new set of parameters govern the equations of motion. The MPOD center of mass would no longer be at the vehicle's center, so attitude-translation cross-coupling would be introduced. Also, the mass addition would decrease the power and response time of the vehicle. However, with knowledge of the dynamic changes and the sensor measurements which are available in the current system, a working controller might be designed.

In addition to performing station-keeping, the control system could be designed such that the pilot could enter new position and attitude coordinates, and MPOD would fly itself between these locations. Upon reaching the new location, MPOD would automatically initiate position and attitude hold control routines. For a project with known coordinates, such as structural assembly by an astronaut, the sequence of maneuvers could be calculated and stored on MPOD's computers. MPOD would follow the preprogrammed pattern, moving between stations upon the pilot's commands. Note that with such a

system, the astronaut performing the task could also be the vehicle pilot, given access to MPOD's underwater switches.

6.3.0 Conclusion

The MIT Space Systems Laboratory has performed a variety of neutral buoyancy simulation experiments to study human factors and control system applications for use in space operations. The development of a neutral buoyancy knowledge base will facilitate the development and construction of an orbital robotic vehicle that could perform analogous functions to the thoroughly tested underwater vehicle described in this thesis.

A major drawback to the MPOD system is the sensor modifications which would be required for space. Acoustic systems such as 3DAPS have proven to be reliable sensors in water; however, sound waves do not travel in a vacuum. Also, there is little gravity in space, so the pendula would be useless. Of course, the depth sensor would be meaningless, as well. Conversely, an equivalent set of space sensors is impossible to use in an underwater environment. Currently, video sensing studies are being undertaken with the hope that they will provide underwater position and attitude feedback that could also be applied in space applications (Reference 28). Fortunately, the MPOD rate sensors would work in space as well as in the water.

Because of the sensor discrepancies, the applicability of MPOD systems to space is concentrated in its multiprocessor system and control algorithm implementation. Through the implementation of the multiprocessor system and state estimation procedure described in this thesis, design of a similar system for a space robotic vehicle might be simplified. After all, the nonlinear drag terms present in underwater dynamics are not felt in space.

At its new home in College Park, Maryland, MPOD will continue to provide valuable human factors and control system results which will be applicable to space robotic control problems. May both the University of Maryland and MIT Space Systems Laboratories live long and prosper.

7.0 References

- [1] Viggh, Herbert, "Artificial Intelligence Applications in Teleoperated Robotic Assembly of the EASE Space Structure", S.M. Thesis, Dept. of Aeronautics and Astronautics and Dept. of Electrical Engineering, MIT, February 1988.
- [2] Kowalski, K. G., "Applications of a Three-Dimensional Position and Attitude Sensing System for Neutral Buoyancy Space Simulation", S.M. Thesis, Dept. of Aeronautics and Astronautics, MIT, October 1989.
- [3] Vyhnalek, G. G., "A Digital Control System for an Underwater Space Simulation Vehicle Using Rate Gyro Feedback", S.M. Thesis, Dept. of Aeronautics and Astronautics, MIT, June 1985.
- [4] Tarrant, J. M., "Attitude Control and Human Factors Issues in the Maneuvering of an Underwater Space Simulation Vehicle", S.M. Thesis, Dept. of Aeronautics and Astronautics, MIT, August 1987.
- [5] Rowley, V. M., "Effects of Stereovision and Graphics Overlay on a Teleoperator Docking Task", S.M. Thesis, Dept. of Aeronautics and Astronautics, MIT, August, 1989.
- [6] Atkins, E. M., "A User's Guide to MPOD -- the Multimode Proximity Operations Device", Space Systems Laboratory Report in Progress, MIT.
- [7] Spofford, J. R., "3-D Position and Attitude Measurement for Underwater Vehicles", Space Systems Lab Report #21-86, MIT, December 1986.
- [8] Brüel and Kjaer, "Instruction Manual -- Hydrophone Types 8101, 8103, 8104, 8105", October 1986.
- [9] Hewlett Packard, Inc., "HP Optoelectronics Designer's Catalog", 1987.
- [10] Analog Devices, Inc., "Analog Devices Databook", Vol. 1, 1984.
- [11] Omega Engineering, Inc., "Series PX240 Pressure Transducer", 1986.
- [12] Humphrey, Inc., "Operating Manual for Rate Transducers, RT02 Series", 1988.
- [13] Motorola, Inc., "HCMOS Single-Chip Microcontroller", 1988.
- [14] Advanced Micro Devices, Inc., "Memory Products: 1989/1990 Data Book", 1989.
- [15] Ampro Computers, Inc., "Technical Manual -- Little Board / PC", 1988.

- [16] Ampro Computers, Inc., "Technical Manual -- Little Board / 286", 1989.
- [17] Ampro Computers, Inc., "Solid State Disk Utilities", 1988.
- [18] Ampro Computers, Inc., "Technical Manual -- MONO/CGA Video Card", 1988.
- [19] Sanner, R. M., "A User's Guide to PiVeCS", Space Systems Laboratory Report in Progress, MIT.
- [20] Douppnik, J. R., "MS-DOS Kermit Reference Guide", Utah State University, July, 1988.
- [21] Battin, R. H., "An Introduction to the Mathematics and Methods of Astrodynamics", American Institute of Aeronautics and Astronautics, New York, 1987.
- [22] Paines, J.D.B., "A Review of Hydrodynamic Forces in Neutral Buoyancy Simulation of Microgravity EVA of IVA", SSL Report # 17-86, July 1986.
- [23] Sanner, R. M., "Optimal State Vector Determination of a Submersible Telerobot", Space Systems Laboratory Report in Progress, MIT.
- [24] Athans, M., "Lecture Notes for Multivariable Control Systems II", MIT, Cambridge, MA, January, 1988.
- [25] Slotine, J. J., "Lecture Notes for 2.152", MIT, Cambridge, MA, 1990.
- [26] Press, W. H. et al., "Numerical Recipes in C: The Art of Scientific Computing", Cambridge University Press, New York, 1988.
- [27] Parrish, J. C., "Trajectory Control of Free-Flying Space and Underwater Vehicles", S.M. Thesis, Dept. of Aeronautics and Astronautics, MIT, August 1987.
- [28] St. John-Olcayto, E., "Machine Vision for Space Robotic Applications", S.M. Thesis, Dept. of Aeronautics and Astronautics, MIT, May 1990.

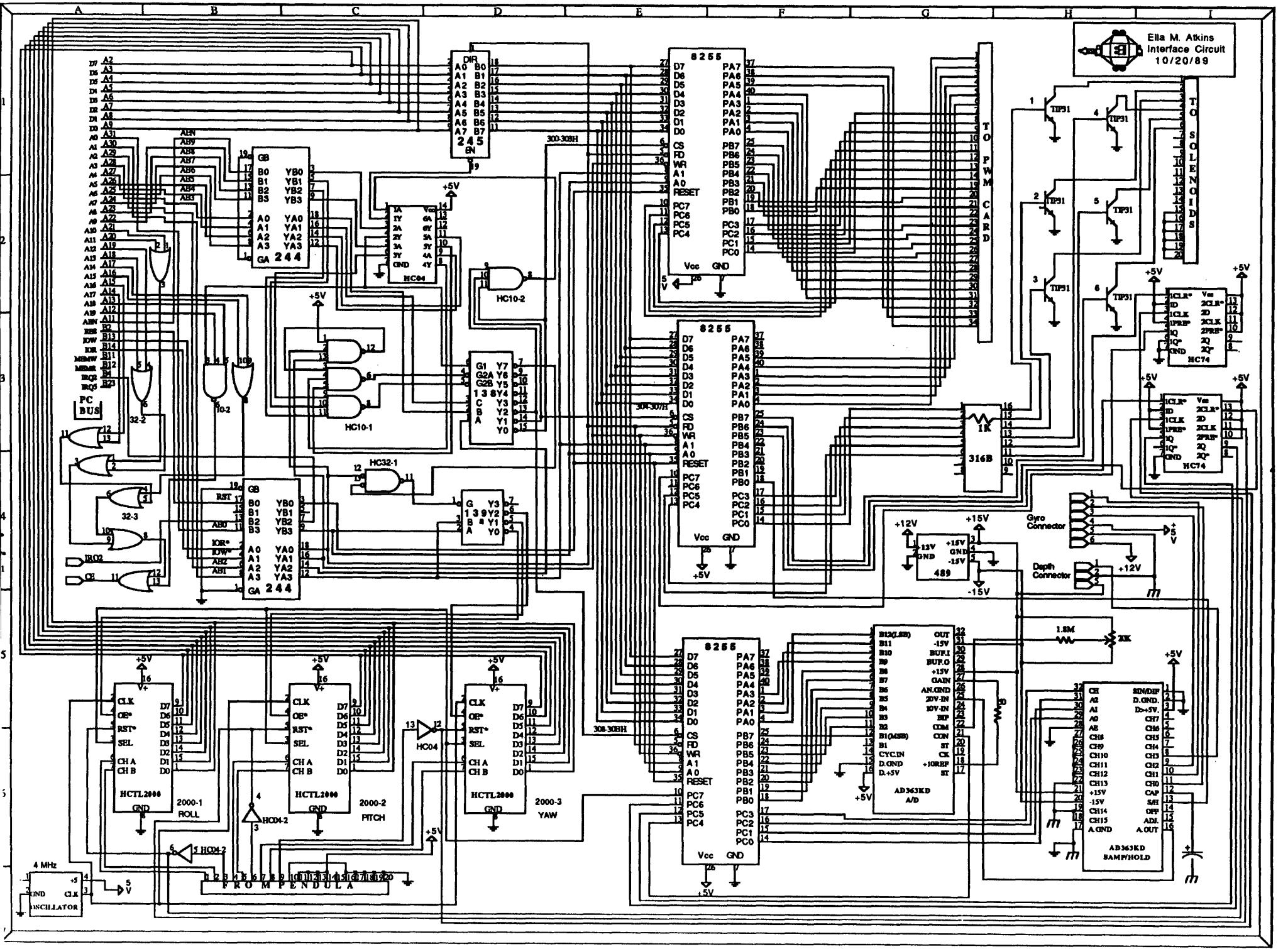
Appendices

Appendix A.0

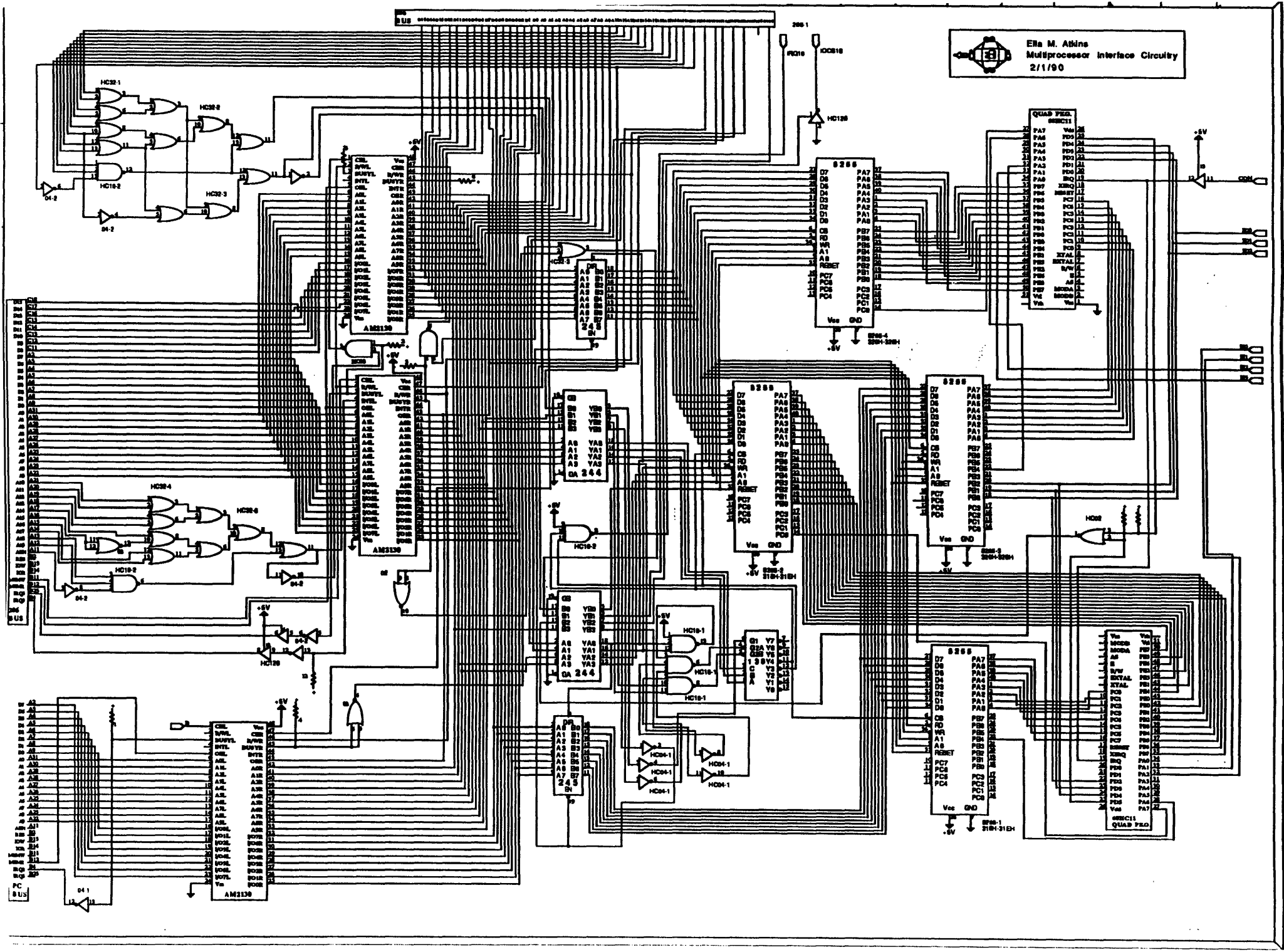
The MPOD circuit diagrams are shown in this appendix. Each describes one of the circuit cards. The first five circuits are contained within MPOD's control box. The following table of contents describes the diagrams:

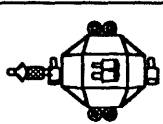
<u>Appendix</u>	<u>Description</u>	<u>Page</u>
Appendix A.1	MPOD Hardware Interface Circuit	118
Appendix A.2	Multiprocessor Interface Circuit	119
Appendix A.3	3DAPS Serial Decoding Circuit	120
Appendix A.4	Pulse Width Modulation Circuit	121
Appendix A.5	Fiber Optics Circuit	122
Appendix A.6	3DAPS Hydrophone Amplifier Circuit	123
Appendix A.7	Luke Hardware Interface Circuit	124

Ella M. Atkins
Interface Circuit
10/20/89

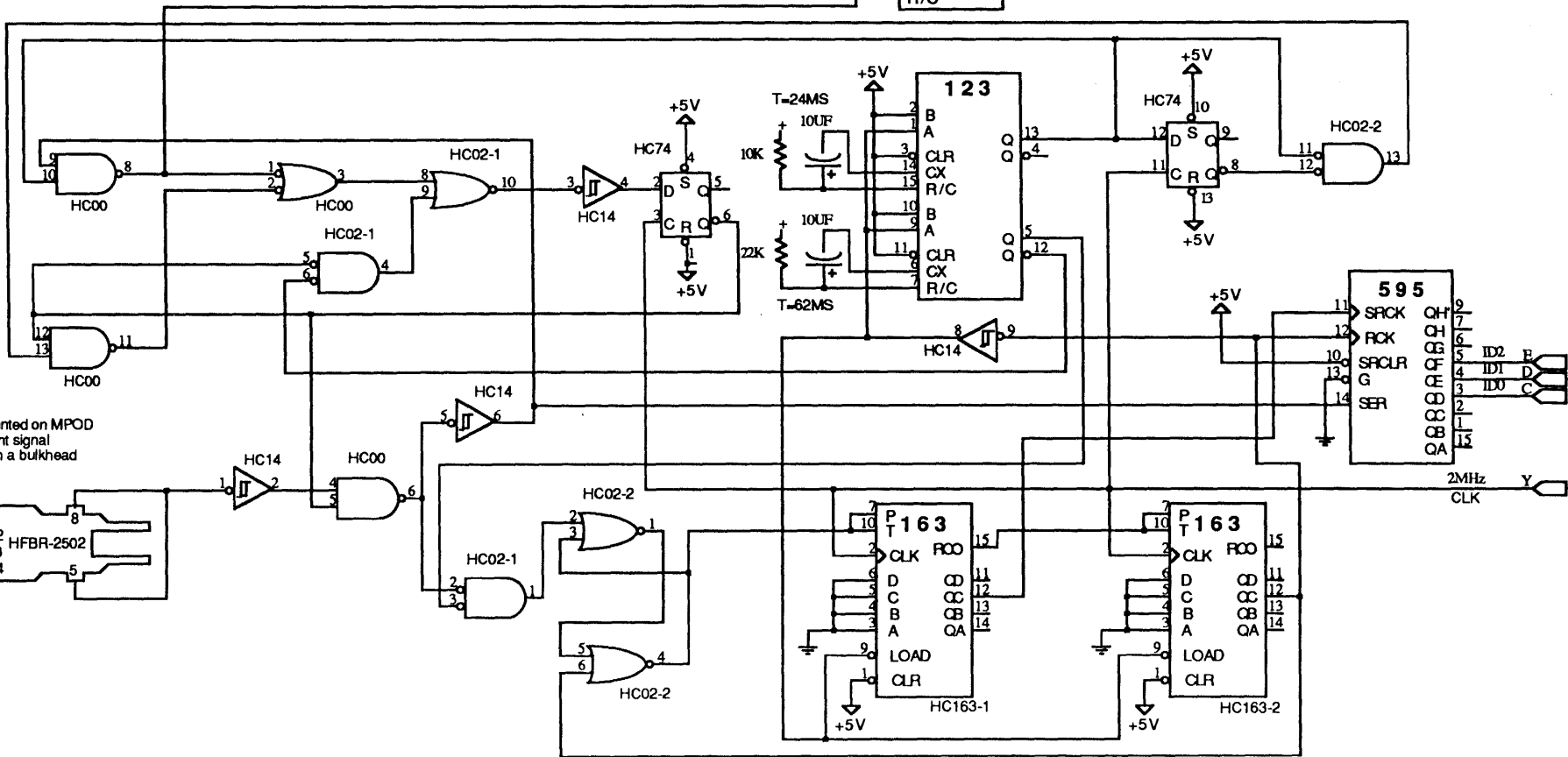
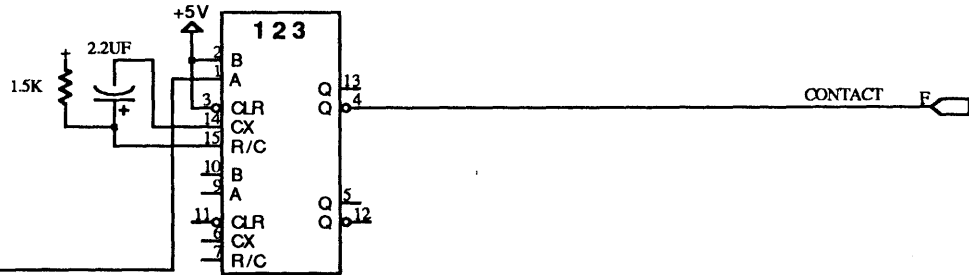


Ella M. Atkins
 Multiprocessor Interface Circuitry
 2/1/90

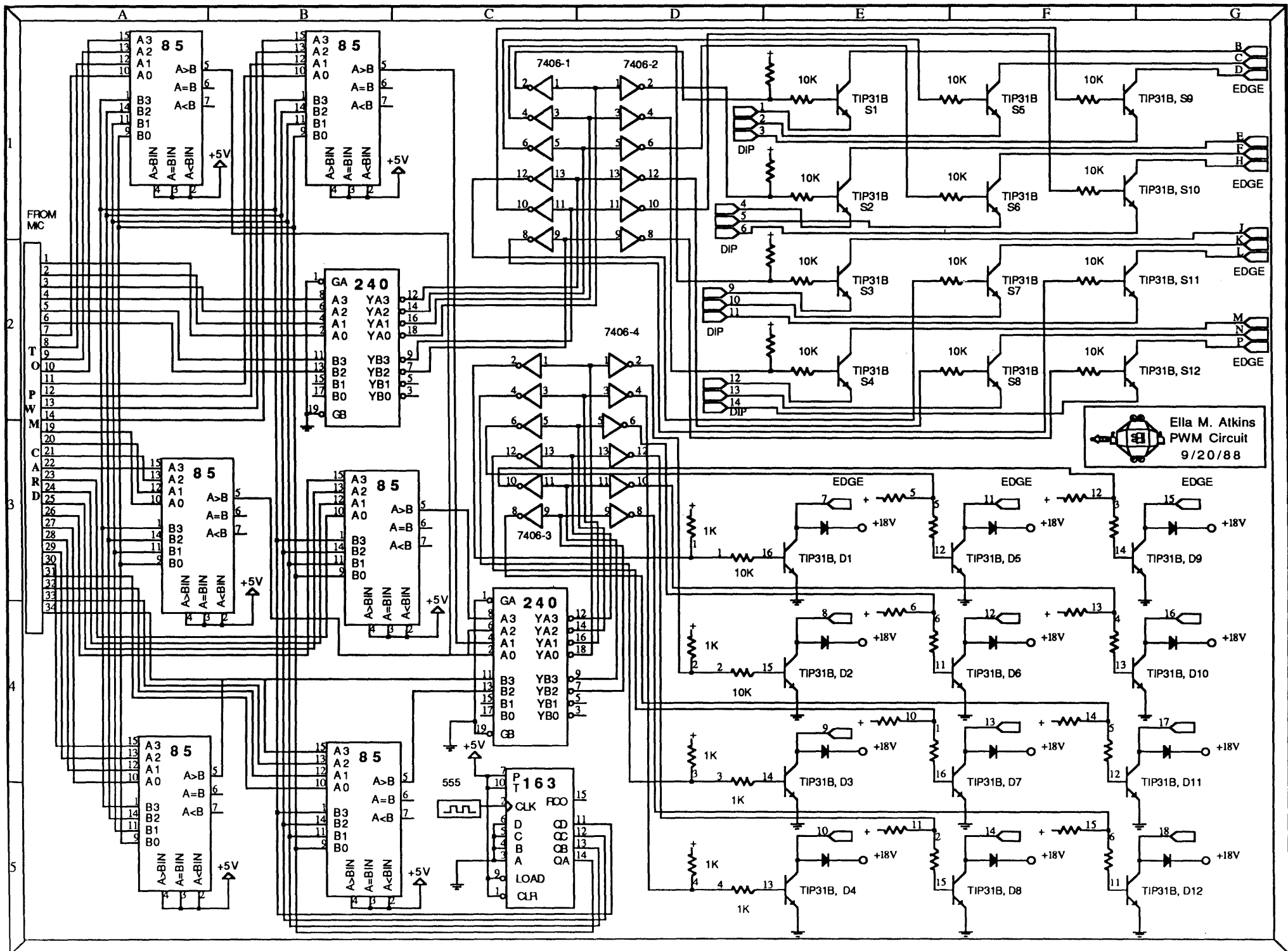




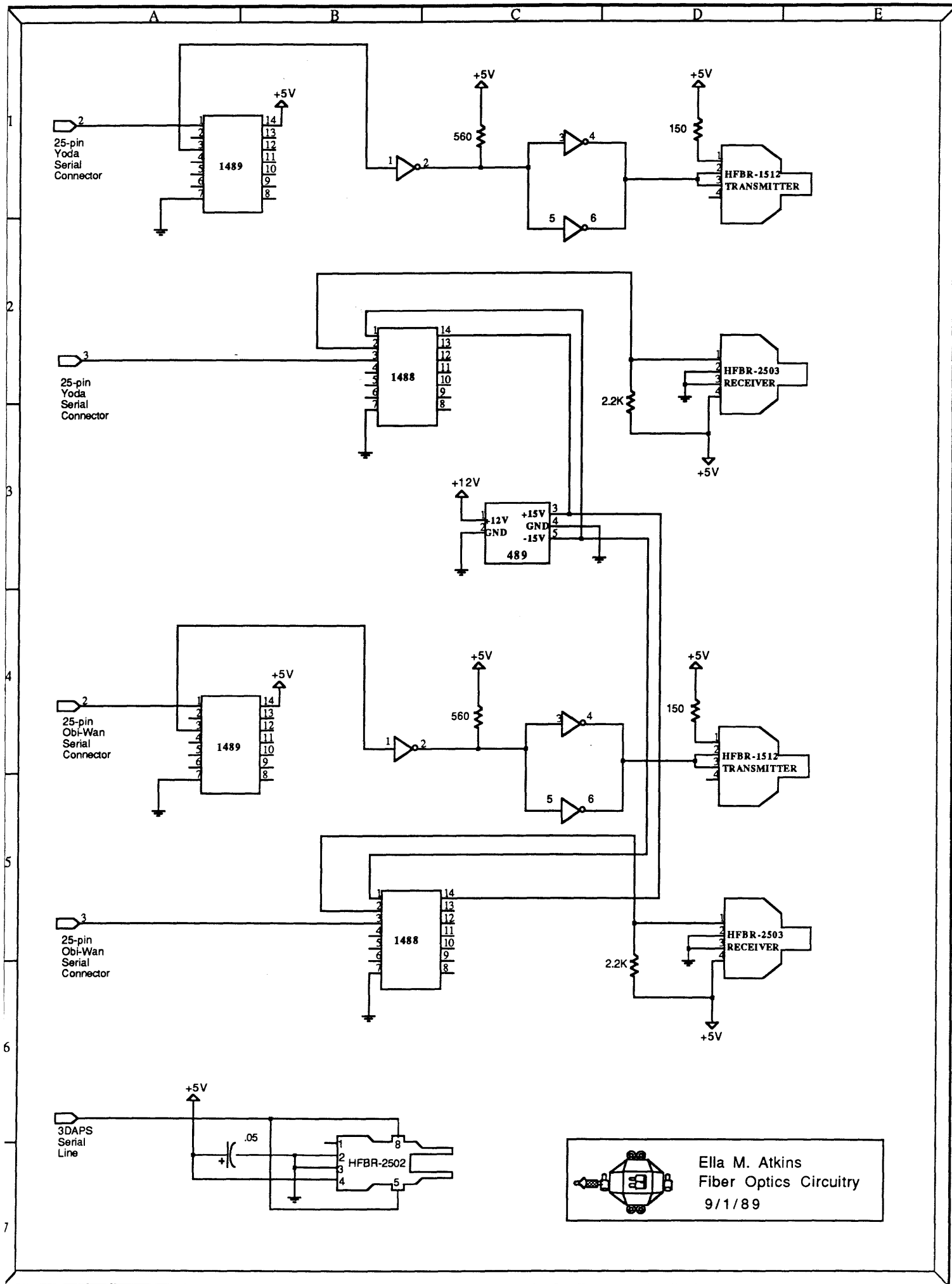
Ella M. Atkins
 Sequencer Serial Decoding Circuitry
 2/1/90

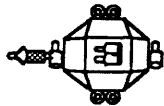


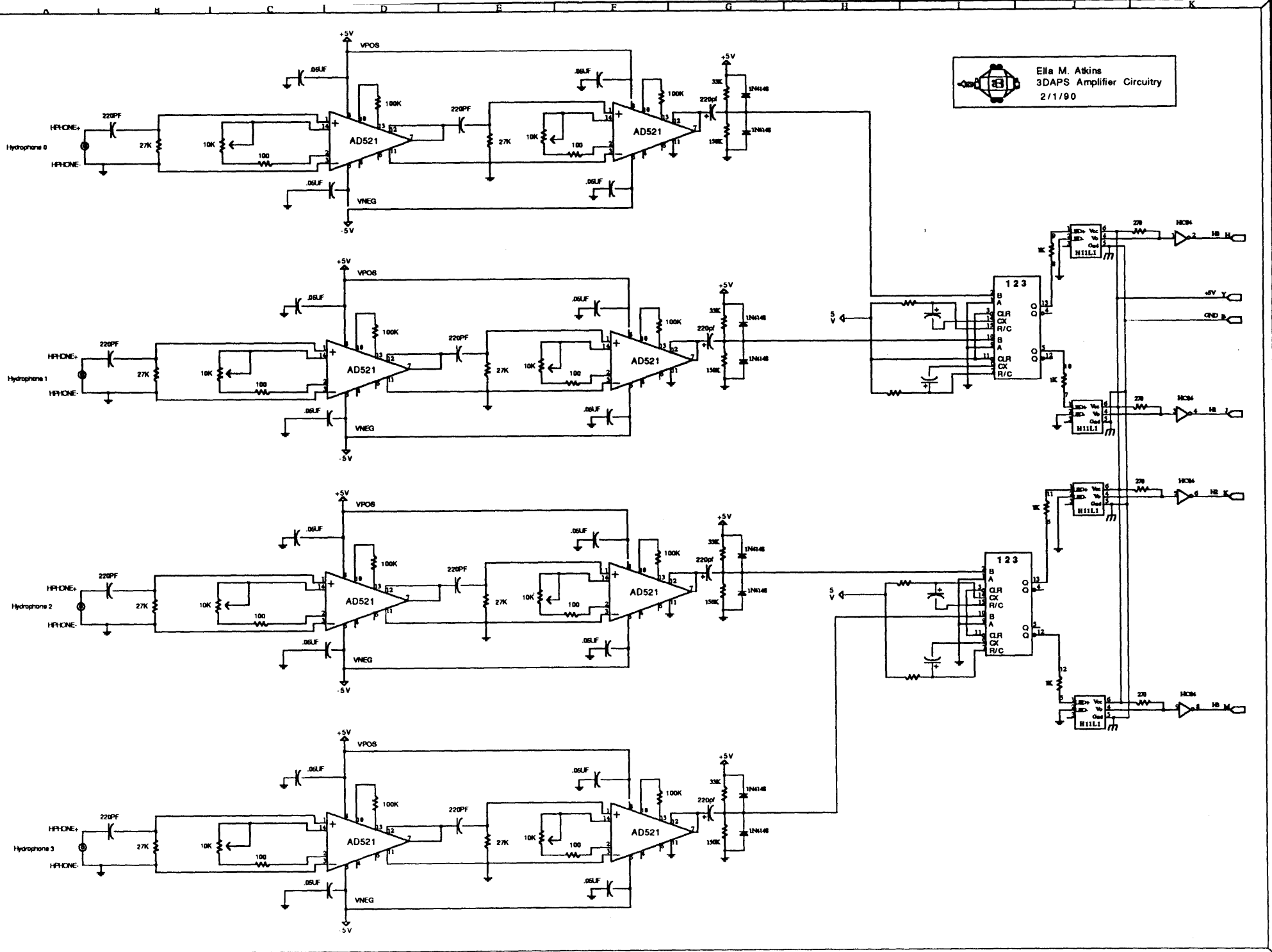
NOTE: 2502 is mounted on MPOD control box to prevent signal degradation through a bulkhead



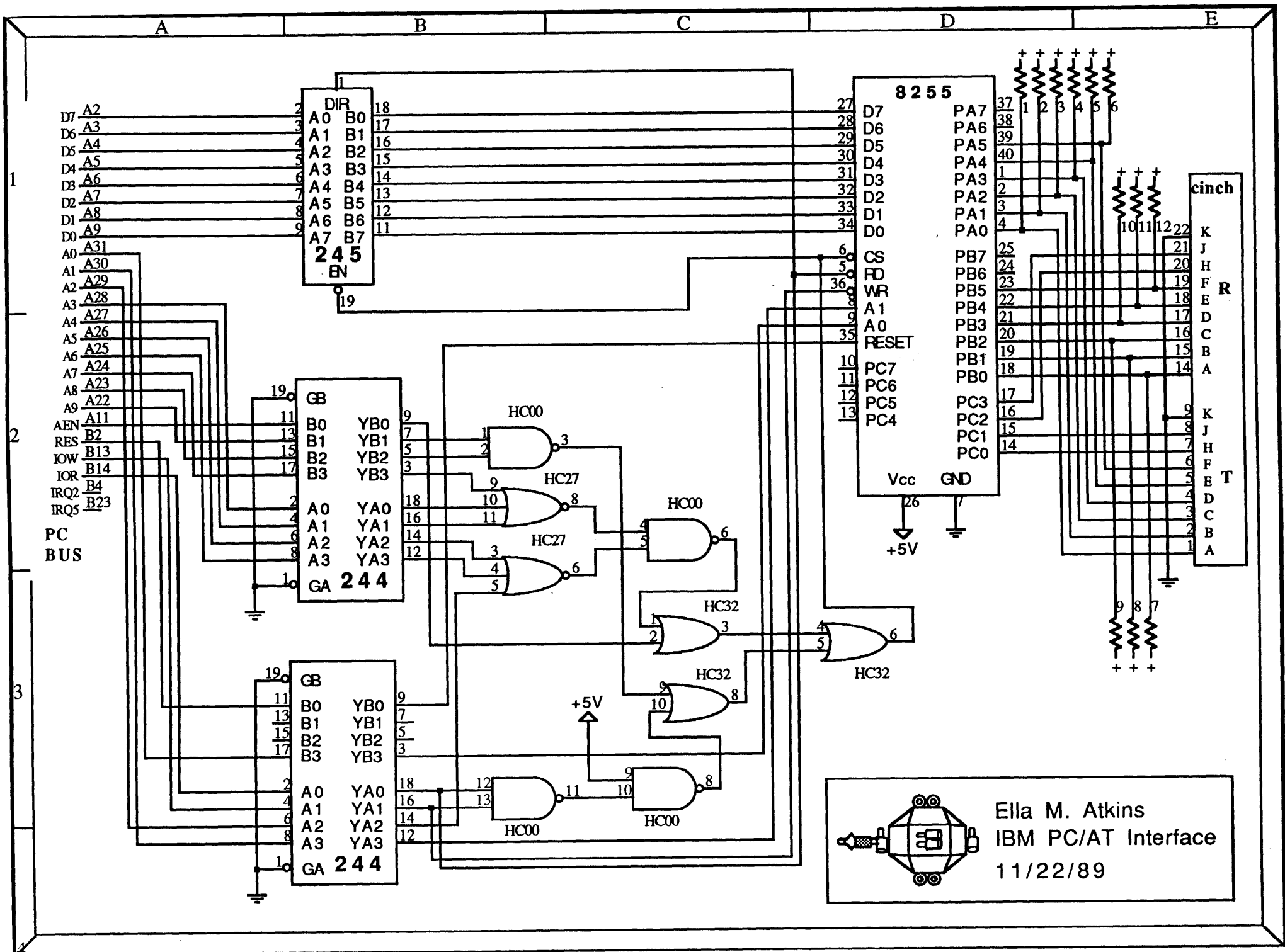
Ella M. Atkins
 PWM Circuit
 9/20/88




 Ella M. Atkins
 Fiber Optics Circuitry
 9/1/89



Ella M. Atkins
3DAPS Amplifier Circuitry
2/1/90



Appendix B.0 MPOD Software

All MPOD software, excluding simulation programs in Appendix D, are presented in this appendix. An index of programs is shown below. (NOTE: Currently 58 pages)

<u>Resident Computer</u>	<u>Program Name</u>	<u>Page</u>
Yoda	pvyoda	126
	pvyoda.msg	126
	pvyoda.h	127
	pvyoda.c	129
	yodamsgs.c	131
	yodafuns.c	136
Obi-Wan	obi	139
	yodaobi.h	139
	obilando.h	140
	yodaobi.c	140
	obilando.c	141
	obiwan.h	141
	obiwan.c	143
	obifuns.c	147
ctrlcalc.c	149	
Lando	lando	151
	lando.h	151
	filter.h	152
	lando.c	153
	landfuns.c	155
	filter.c	157
	pstate.c	162
Luke	pvluke	166
	pvluke.msg	166
	pvluke.h	167
	pvluke.c	169
	lukemsgs.c	171
	lukefuns.c	176
Crumb and Cake	USMV6811.TXT	180
	INT6811.TXT	180

Appendix B.1 Yoda Software

```

/ *          pvyoda          * /
/*          Microsoft C Make file          * /

pvyoda.obj: pvyoda.h ..\pvluke.msg pvyoda.c
            cl /c pvyoda.c

yodafuns.obj: pvyoda.h yodafuns.c
            cl /c yodafuns.c

yodamsgs.obj: ..\pivecs\pvddata.h pvyoda.h ..\pvluke.msg yodamsgs.c
            cl /c yodamsgs.c

pvyoda.exe: pvyoda.obj yodamsgs.obj yodafuns.obj ..\yodaobi.obj
            link /NOD $**, pvyoda.exe,,SLIBCE+GFCS+GFS+..\pivecs\PIVECS

/ *          pvyoda.msg          * /
/ *          Written by: Robert M. Sanner          * /
/*          Last modified by: Ella M. Atkins, 4/11/90          * /

#ifndef YODAMSGS
#define YODAMSGS

/*****
Recognized Message List *****/
#define COMTEST      0x00 /* Msg 0, No Data          */
#define COMAOK       0x08 /* Msg 1, " "            */
#define SHUTDN       0x10 /* Msg 2, " "            */
#define YODAESC      0x20 /* Msg 4, " "            */

#define TXMOTORS     0x30 /* Msg 6, " "            */
#define TXPENDULA    0x38 /* Msg 7, " "            */
#define TXGYROS      0x40 /* Msg 8, " "            */
#define TXHYDRO01    0x48 /* Msg 9, " "            */
#define TXHYDRO23    0x50 /* Msg 10, " "           */

#define TXPOSITION   0x60 /* Msg 12, " "           */
#define TXATTITUDE   0x68 /* Msg 13, " "           */
#define TXVELOCITY   0x70 /* Msg 14, " "           */
#define TXOMEGA      0x78 /* Msg 15, " "           */
#define TXBIAS       0x80 /* Msg 16, " "           */

#define RXTHC        0xA1 /* Msg 20, 1 Data Byte */
#define RXRHC        0xA9 /* Msg 21, 1 Data Byte */
#define RXSWITCH     0xB3 /* Msg 22, 3 Data Bytes */
#define RXCTRL       0xBA /* Msg 23, 2 Data Bytes */
#define RXSTATE      0xC5 /* Msg 24, 5 Data Bytes */
#define RXGAINS      0xCB /* Msg 25, 3 Data Bytes */

#define BADMSG       0xFF /* Msg 31, 7 data. Placeholder for Bad msgs */
/*****
#endif

```

```

/ *          pvyoda.h          * /
/ *    Written by: Robert M. Sanner    * /
/*  Last modified by: Ella M. Atkins, 4/11/90    * /

#ifndef PIVECS
#include "..\pivecs\pivecs.h"
#endif
#ifndef YODAMSGS
#include "..\pvpyoda.msg"
#endif
#ifndef YODAOBI
#include "..\yodaobi.h"
#endif

#ifndef YODA
#define YODA

extern HandlerFunc    BadMsg, ShutDown, ComCheck, ComAOK;
extern HandlerFunc    RX_THC, RX_RHC, RX_Switch, Yod_ESC;
extern HandlerFunc    RX_Ctrl, RX_State, RX_Gains;
extern HandlerFunc    TX_Motors, TX_Pendula, TX_Gyros;
extern HandlerFunc    TX_Hydro01, TX_Hydro23, TX_Bias;
extern HandlerFunc    TX_Position, TX_Attitude, TX_Velocity, TX_Omega;

static Handlers        YodaHandlers =
{ComCheck,    ComAOK,    ShutDown,    BadMsg,
 Yod_ESC,    BadMsg,    TX_Motors,    TX_Pendula,
 TX_Gyros,    TX_Hydro01, TX_Hydro23,    BadMsg,
 TX_Position, TX_Attitude, TX_Velocity, TX_Omega,
 TX_Bias,    BadMsg,    BadMsg,    BadMsg,
 RX_THC,    RX_RHC,    RX_Switch,    RX_Ctrl,
 RX_State,    RX_Gains,    BadMsg,    BadMsg,
 BadMsg,    BadMsg,    BadMsg,    BadMsg};

static Headers YodaMsgs =
{COMTEST,    COMAOK,    SHUTDN,    BADMSG,
 YODAESC,    BADMSG,    TXMOTORS,    TXPENDULA,
 TXGYROS,    TXHYDRO01, TXHYDRO23,    BADMSG,
 TXPOSITION, TXATTITUDE, TXVELOCITY,    TXOMEGA,
 TXBIAS,    BADMSG,    BADMSG,    BADMSG,
 RXTHC,    RXRHC,    RXSWITCH,    RXCTRL,
 RXSTATE,    RXGAINS,    BADMSG,    BADMSG,
 BADMSG,    BADMSG,    BADMSG,    BADMSG};

static Byte    YodaHiPri = 5;
Byte    STOP;

/ *    Global MPOD State information    * /

#define WLEN    0x05

/*  Components of *Switches used by Yoda */

```

```

#define SAVE_DATA          0x01
#define STATE_CALC        0x02
#define OBI_ESCAPE        0x10
#define LANDO_ESCAPE      0x20
#define ESCAPE            0x40

/* Components of *Pneu_View used by Yoda */

#define POWER             0x10
#define RAM               0x04
#define LATCH            0x02
#define SENSOR_SEE       0x20
#define HYDRO_SEE        0x40
#define STATE_SEE        0x80

/ *    Ports for the I/O Protoboard    */

#define P0X              0x303
#define P1X              0x307
#define P2X              0x30B

#define INIT0            0x80
#define INIT1            0x82
#define INIT2            0x92

/ *    Ports for the pneumatics and motors    */

#define PNEUPORT         0x306
#define SGNPORT          0x304
#define XMOTPORT         0x302
#define YMOTPORT         0x301
#define ZMOTPORT         0x300
#define PNEUMASK         0x16 /* output if all pneumatics activated */

/ *    Ports and equates for the gyros, depth sensor, and pendula    */

#define AD_PORT          0x30A
#define AD_STATUS        0x305
#define AD_BUSY          0x00
#define AD_READ_LO       0x308
#define AD_READ_HI       0x309
#define AD_LO            0x80
#define AD_HI            0x00
#define DELAY            0x100

#define ROLL_ANGLE       0x31C
#define PITCH_ANGLE      0x31D
#define YAW_ANGLE        0x31E

#define ROLL_ADDR        0x08
#define ROLL_LATCH       0x00
#define ROLL_START       0x10
#define ROLL_END         0x00

```



```

#define PITCH_ADDR      0x09
#define PITCH_LATCH    0x01
#define PITCH_START    0x11
#define PITCH_END      0x01

#define YAW_ADDR       0x0A
#define YAW_LATCH     0x02
#define YAW_START     0x12
#define YAW_END       0x02

#define DEPTH_ADDR     0x0B
#define DEPTH_LATCH   0x03
#define DEPTH_START   0x13
#define DEPTH_END     0x03

        /* Yoda specific control subroutines */

void  MpodInit(), FireMotors(), FirePneu();
void  ReadGyros(), ReadPendula();

void  interrupt far  Busy();

char  FOO_C;
#define charswap(x,y) FOO_C = (x); (x) = (y); (y) = FOO_C;
Byte  FOO_B;
#define byteswap(x,y) FOO_B = (x); (x) = (y); (y) = FOO_B;

#endif

/ *          pvyoda.c          * /
/ *    Written by: Robert M. Sanner    * /
/* Last modified by: Ella M. Atkins, 3/23/90    * /

#include "pvyoda.h"
#include "..\pvluke.msg"
#include <stdio.h>
#include <gf.h>
#include <dos.h>
#include <ibmkeys.h>
#include <asiports.h>

main() {          /* Begin PiVecs driver program */

    unsigned  getkey();
    unsigned long  CurrMsg = 0;
    register      i;
    Byte          messtat;

    pvInitCom( COM1, 9600, P_ODD, 1 );
    pvInitMsg( YodaHandlers, YodaMsgs, YodaHiPri );
    messtat = pvWorry(RXSWITCH, (Byte) 100, TXSWITCH);

```

```

/* Enable and initialize interrupt IRQ2 */

    outp(0x21, 0xA8);
    _dos_setvect(0x0A, Busy);
    outp(0x20, 0x20);

    Dual_Init_1();
    MpodInit();
    pvRequest( TXSWITCH );

    while (!(STOP || (*status & ESCAPE))) { /* Begin main driver loop */

        if (kbhit()) if ( getkey() == ESC ) break;

        CurrMsg = pvRecv();

        / *      Activate changes in pneumatics          * /

        if ((*Pstatus) & PNEUMASK) {
            FirePneu();
            if (((*Pstatus) & POWER) && ((*Pneu_View) & POWER)) {
                pvRequest(TXTHC);
                pvRequest(TXRHC);
                messtat = pvWorry(RXTHC, (Byte) 10, TXTHC);
                messtat = pvWorry(RXRHC, (Byte) 10, TXRHC);
            } else if ((*Pstatus) & POWER) {
                messtat = pvWorry(RXTHC, (Byte) 255, TXTHC);
                messtat = pvWorry(RXRHC, (Byte) 255, TXRHC);
            }
            *Pstatus &= (~PNEUMASK);
        }

        /* Read sensors for state calculation, data view, or data storage */

        if (((*Switches) & (STATE_CALC | SAVE_DATA))
            || ((*Pneu_View) & SENSOR_SEE)) {
            ReadGyros();
            ReadPendula(); }

        if ((*Pneu_View)&POWER) {
            for (i = 0; i < 6; i++)
                FireMotors();
        }

    } /* End main driver WHILE loop */

    pvRequest( YODAESC );
    if (*status & ESCAPE) {
        *status |= ((~ESCAPE) & (OBI_ESCAPE | LANDO_ESCAPE));
    } /* Escape from Obi-Wan and Lando */
    *Switches = 0x00;
    *Pneu_View = 0x00;
    *State_Stuff = 0x0F;
    for (i = 0; i < 6; i++)

```

```

        *(Motors+i) = 0x00;
    FirePneu();
    FireMotors();

    printf("No problems on loop exit\n");
    pvExit();

} /* End program */

void MpodInit() {

    register i, j;

    /*          Initialize the 8255's on the I/O protoboard      */

    outp(P0X, INIT0);
    outp(P1X, INIT1);
    outp(P2X, INIT2);

    /*          Zero out all the state variables and toggle flags      */
    STOP = 0x00;
    for (i = 0; i < 6; i++) {
        if (i < 3) {
            *(Gyros+i) = 0x800;
            *(Pendula+i) = 0x000;
        }
        *(Motors+i) = 0x00;
    }
}

/*          yodamsgs.c          */
/*          Written by: Robert M. Sanner and Ella M. Atkins      */
/*          Last modified by: 4/11/90          */

#include "pvyoda.h"
#include "..\pvluke.msg"
#include "..\pivecs\pvdata.h"

MsgHandler ShutDown(msg)
    MsgPtr msg;
{
    register i;
    *Switches = 0x00;
    *Pneu_View = 0x00;
    *State_Stuff = 0x0F;
    for (i = 0; i < 6; i++) {
        *(Motors+i) = 0x00;
    }
    *thc = *rhc = 0x00;
    *status = 0x00;
    *Pstatus = 0x00;
    FirePneu();
}

```

```

    FireMotors();
    return( OK );
}

MsgHandler Yod_ESC(msg)
    MsgPtr msg;
{
    STOP = 0x01;
    return( OK );
}

MsgHandler RX_Ctrl(msg)
    MsgPtr msg;
{
    BytePtr datptr = msg->data;
    *path = *datptr++;
    *PID = *datptr;
    return( OK );
}

MsgHandler RX_State(msg)
    MsgPtr msg;
{
    register i;
    BytePtr    datptr = msg->data;
    unsigned char far *point;
    point = stateval;
    *stateID = *datptr++;
    for (i = 0; i < 4; i++)
        *point++ = *datptr++;
    return( OK );
}

MsgHandler RX_Gains(msg)
    MsgPtr msg;
{
    register i;
    BytePtr    datptr = msg->data;
    unsigned char far *point;
    point = gainval;
    *gainID = *datptr++;
    for (i = 0; i < 2; i++)
        *point++ = *datptr++;
    return( OK );
}

MsgHandler RX_THC(msg)
    MsgPtr msg;
{
    BytePtr datptr = msg->data;
    *thc = *datptr;
    if ((*Pneu_View)&POWER) pvRequest( TXTHC );
    return( OK );
}

```

```

MsgHandler RX_RHC(msg)
    MsgPtr msg;
{
    BytePtr datptr = msg->data;
    *rhc = *datptr;
    if ((*Pneu_View)&POWER) pvRequest( TXRHC );
    return( OK );
}

```

```

MsgHandler RX_Switch(msg)
    MsgPtr msg;
{
    BytePtr datptr = msg->data;
    if ((*Switches) != *datptr) {
        *status |= ((*Switches)^( *datptr));
        *Switches = *datptr;
    }
    *datptr++;
    if ((*Pneu_View) != *datptr) {
        *Pstatus |= ((*Pneu_View)^( *datptr));
        *Pneu_View = *datptr;
    }
    *datptr++;
    if ((*State_Stuff) != *datptr)
        *State_Stuff = *datptr;
    pvRequest( TXSWITCH );
    return( OK );
}

```

```

MsgHandler TX_Motors(msg)
    MsgPtr msg;
{
    Byte array[5];
    array[0] = RXMOTORS;
    array[1] = ((*Motors) << 4) | (*(Motors+1));
    array[2] = ((*Motors+3) << 4) | (*(Motors+2));
    array[3] = ((*Motors+4) << 4) | (*(Motors+5));
    array[4] = *MotorSigns;
    pvSend(array, 5);
    return( OK );
}

```

```

MsgHandler TX_Pendula(msg)
    MsgPtr msg;
{
    register i;
    Byte array[8];
    array[0] = RXPENDULA;
    for (i = 0; i < 3; i++) {
        array[2*i+1] = (Byte) *(Pendula+i);
        array[2*(i+1)] = (((Byte) *(Pendula+i) >> 8) & 0x0F); }
    array[7] = (Byte) ((*Depth) & 0x00F);
    pvSend(array, 8);
}

```

```

    return( OK );
}

MsgHandler TX_Gyros(msg)
MsgPtr msg;
{
    register i;
    Byte array[8];
    array[0] = RXGYROS;
    for (i = 0; i < 3; i++) {
        array[(2*i)+1] = (Byte) (*(Gyros+i));
        array[2*(i+1)] = (((Byte) (*(Gyros+i) >> 8)) & 0x0F); }
    array[7] = (Byte) ((*Depth) >> 4);
    pvSend(array, 8);
    return( OK );
}

MsgHandler TX_Hydro01(msg)
MsgPtr msg;
{
    register i;
    static unsigned char q=0;
    unsigned short j;
    Byte array[6];
    array[0] = RXHYDRO01;
    array[1] = q;
    j = q << 2;
    for (i = 1; i < 3; i++) {
        array[2*i] = (Byte) *(arange+i+j-1));
        array[(2*i)+1] = (Byte) ((*arange+i+j-1)) >> 8); }
    if (q < 7) q++;
    else q = 0;
    pvSend(array, 6);
    return( OK );
}

MsgHandler TX_Hydro23(msg)
MsgPtr msg;
{
    register i;
    static unsigned char u=0;
    unsigned short j;
    Byte array[6];
    array[0] = RXHYDRO23;
    array[1] = u;
    j = u << 2;
    for (i = 1; i < 3; i++) {
        array[2*i] = (Byte) *(arange+i+1+j));
        array[(2*i)+1] = (Byte) ((*arange+i+1+j)) >> 8); }
    if (u < 7) u++;
    else u = 0;
    pvSend(array, 6);
    return( OK );
}

```

MsgHandler TX_Position(msg)

```
    MsgPtr msg;
{
    register i;
    Byte array[8];
    array[0] = RXPOSITION;
    for (i = 0; i < 3; i++) {
        array[(2*i)+1] = (Byte) (*(astate+i));
        array[2*(i+1)] = (Byte) (*(astate+i) >> 8);
    }
    array[7] = (Byte) (*(astate+6)); /* First half of last quaternion */
    pvSend(array, 8);
    return( OK );
}
```

MsgHandler TX_Attitude(msg)

```
    MsgPtr msg;
{
    register i;
    Byte array[8];
    array[0] = RXATTITUDE;
    for (i = 0; i < 3; i++) {
        array[(2*i)+1] = (Byte) (*(astate+i+3));
        array[2*(i+1)] = (Byte) (*(astate+i+3) >> 8);
    }
    array[8] = (Byte) (*(astate+6) >> 8); /* High Byte of last quaternion */
    pvSend(array, 8);
    return( OK );
}
```

MsgHandler TX_Velocity(msg)

```
    MsgPtr msg;
{
    register i;
    Byte array[7];
    array[0] = RXVELOCITY;
    for (i = 0; i < 3; i++) {
        array[(2*i)+1] = (Byte) (*(astate+i+7));
        array[2*(i+1)] = (Byte) (*(astate+i+7) >> 8);
    }
    pvSend(array, 7);
    return( OK );
}
```

MsgHandler TX_Omega(msg)

```
    MsgPtr msg;
{
    register i;
    Byte array[7];
    array[0] = RXOMEGA;
    for (i = 0; i < 3; i++) {
        array[(2*i)+1] = (Byte) (*(astate+i+10));
        array[2*(i+1)] = (Byte) (*(astate+i+10) >> 8);
    }
}
```

```

    }
    pvSend(array, 7);
    return( OK );
}

MsgHandler TX_Bias(msg)
    MsgPtr msg;
{
    register i;
    Byte array[7];
    array[0] = RXBIAS;
    for (i = 0; i < 3; i++) {
        array[(2*i)+1] = (Byte) (*(astate+i+13));
        array[2*(i+1)] = (Byte) (*(astate+i+13) >> 8);
    }
    pvSend(array, 7);
    return( OK );
}

```

```

/ *          yodafuns.c          * /
/ *   Written by: Robert M. Sanner      * /
/* Last modified by: Ella M. Atkins, 1/8/90 * /

```

```

#include "pvyoda.h"

```

```

#include <dos.h>

```

```

void ReadGyros() {

```

```

    Byte hi_byte, lo_byte;
    short i = 0;

```

```

    outp(AD_PORT, ROLL_ADDR);          /* send address for x */
    outp(AD_PORT, ROLL_LATCH);         /* latch address for x */
    outp(AD_PORT, ROLL_START);        /* send START for x */
    outp(AD_PORT, ROLL_END);          /* end START pulse for x */
    while ( ( inp(AD_STATUS) != AD_BUSY) && (i++<DELAY) );
    hi_byte = inp(AD_READ_HI);
    lo_byte = inp(AD_READ_LO) & 0x0F;
    *Gyros = ( ((unsigned short) hi_byte) << 4) | lo_byte;
    i = 0;

```

```

    outp(AD_PORT, PITCH_ADDR);        /* send address for y */
    outp(AD_PORT, PITCH_LATCH);       /* latch address for y */
    outp(AD_PORT, PITCH_START);       /* send START for y */
    outp(AD_PORT, PITCH_END);         /* end START pulse for y */
    while ( ( inp(AD_STATUS) != AD_BUSY) && (i++<DELAY) );
    hi_byte = inp(AD_READ_HI);
    lo_byte = inp(AD_READ_LO) & 0x0F;
    *(Gyros+1) = ( ((unsigned short) hi_byte) << 4) | lo_byte;
    i = 0;

```

```

    outp(AD_PORT, YAW_ADDR);          /* send address for z */
    outp(AD_PORT, YAW_LATCH);         /* latch address */

```



```

    outp(AD_PORT, YAW_START);          /* send START for z */
    outp(AD_PORT, YAW_END);           /* end START pulse for z */
    while ( ( inp(AD_STATUS) != AD_BUSY) && (i++<DELAY) );
    hi_byte = inp(AD_READ_HI);
    lo_byte = inp(AD_READ_LO) & 0x0F;
    *(Gyros+2) = ( ((unsigned short) hi_byte) << 4) | lo_byte;
    i = 0;

    outp(AD_PORT, DEPTH_ADDR);        /* send address for depth */
    outp(AD_PORT, DEPTH_LATCH);      /* latch address for depth */
    outp(AD_PORT, DEPTH_START);      /* send START for depth */
    outp(AD_PORT, DEPTH_END);        /* end START for depth */
    while ( ( inp(AD_STATUS) != AD_BUSY) && (i++<DELAY) );
    hi_byte = inp(AD_READ_HI);
    lo_byte = inp(AD_READ_LO)&0x0F;
    *Depth = ( ((unsigned short) hi_byte) << 4) | lo_byte;
}

void ReadPendula() {

    Byte      hi_byte, lo_byte;
    register  i;

    outp(AD_PORT, AD_HI);
    hi_byte = inp(ROLL_ANGLE)&0x0F;
    outp(AD_PORT, AD_LO);
    lo_byte = inp(ROLL_ANGLE);
    *Pendula = ( ((unsigned short) hi_byte) << 8) | lo_byte;

    outp(AD_PORT, AD_HI);
    hi_byte = inp(PITCH_ANGLE)&0x0F;
    outp(AD_PORT, AD_LO);
    lo_byte = inp(PITCH_ANGLE);
    *(Pendula+1) = ( ((unsigned short) hi_byte) << 8) | lo_byte;

    outp(AD_PORT, AD_HI);
    hi_byte = inp(YAW_ANGLE)&0x0F;
    outp(AD_PORT, AD_LO);
    lo_byte = inp(YAW_ANGLE);
    *(Pendula+2) = ( ((unsigned short) hi_byte) << 8) | lo_byte;
}

void FirePneu() {
    register i;

    outp( PNEUPORT, ((*Pneu_View)&PNEUMASK) );
    for (i = 0; i < PAUSE; i++);
}

void FireMotors() {

```

```

Byte   X_out = 0x00, Y_out = 0x00, Z_out = 0x00;
char   x1, x2, y1, y2, z1, z2;
/ *    Check motor signs and adjust sign byte accordingly
        output only absolute motor mags to the I/O ports    * /

*MotorSigns = 0x00;
x1 = *(Motors+1);
x2 = *Motors;
y1 = *(Motors+2);
y2 = *(Motors+3);
z1 = *(Motors+5);
z2 = *(Motors+4);

/* X Motors */

if (x1 > 0) *MotorSigns |= 0x01;
    else x1 = -x1;
if ( x2 > 0) *MotorSigns |= 0x02;
    else x2 = -x2;

/* Y Motors */
if (y1 > 0) *MotorSigns |= 0x20;
    else y1 = -y1;
if (y2 > 0) *MotorSigns |= 0x04;
    else y2 = -y2;

/* Z Motors */
if (z1 > 0) *MotorSigns |= 0x08;
    else z1 = -z1;
if (z2 > 0) *MotorSigns |= 0x10;
    else z2 = -z2;

X_out = (x2 << 4) | x1;
Y_out = (y2 << 4) | y1;
Z_out = (z2 << 4) | z1;

outp(SGNPORT, *MotorSigns);    /* Output motor directions */
outp(XMOTPORT, X_out);        /* Output motor magnitudes */
outp(YMOTPORT, Y_out);
outp(ZMOTPORT, Z_out);

}

void interrupt far   Busy() {
    /* Interrupt causes delay for Dual Port Ram */
    outp(0x20, 0x20);    /* Reset 8259A */
    return;
}

```

Appendix B.2 Obiwan Software

```
/ *          obi          * /
/ *    Microsoft C Make file    * /

obiwan.obj: obiwan.h obiwan.c
            cl /c /FPI87 obiwan.c

pathcalc.obj: obiwan.h pathcalc.c
              cl /c /FPI87 pathcalc.c

ctrlcalc.obj: obiwan.h ctrlcalc.c
              cl /c /FPI87 ctrlcalc.c

obifuns.obj: obiwan.h obifuns.c
             cl /c /FPI87 obifuns.c

obiwan.exe: obiwan.obj pathcalc.obj ctrlcalc.obj obifuns.obj
            ..\yodaobi.obj ..\obilando.obj
            link /NOD $**, obiwan.exe,,SLIBC7+GFCS+GFS

/ *          yodaobi.h          * /
/ *    Written by: Ella M. Atkins    * /
/ *    Last modified: 4/11/90        * /

#ifndef YODAOBI
#define YODAOBI

/* Pointer addresses set for Dual Port Ram between Yoda and Obi-Wan */

unsigned char far *Switches;
unsigned char far *Pneu_View;
unsigned char far *State_Stuff;
unsigned char far *MotorSigns;
unsigned char far *thc;
unsigned char far *rhc;
unsigned char far *status;
unsigned char far *Pstatus;
unsigned short far *Depth;
unsigned short far *Pendula;
unsigned short far *Gyros;
unsigned char far *path;
unsigned char far *PID;
unsigned char far *gainID;
unsigned short far *gainval;
unsigned char far *statelD;
float far *stateval;
char far *Motors;
int far *astate;
unsigned short far *arange;
unsigned char far *athumpID;
```

```
void Dual_Init_1();
#endif
```

```
/*      obilando.h      */
/*      Written by: Ella M. Atkins      */
/*      Last modified: 3/23/90      */
```

```
#ifndef OBILANDO
#define OBILANDO
```

```
/* Pointer addresses set for Dual Port Ram between Obi-Wan and Lando */
```

```
unsigned char far *bSwitches;
unsigned char far *bState_Stuff;
unsigned char far *bthc;
unsigned char far *brhc;
unsigned char far *bstatus;
unsigned short far *bDepth;
unsigned short far *bPendula;
unsigned short far *bGyros;
double far *state;
unsigned short far *range;
unsigned char far *thumpID;
unsigned char far *actuator;
unsigned short far *gate;
unsigned short far *INT_DUAL;
unsigned short far *DPR_num;
```

```
void Dual_Init_2();
#endif
```

```
/*      yodaobi.c      */
/*      Written by: Ella M. Atkins      */
/*      Last modified: 4/11/90      */
```

```
#include "yodaobi.h"
```

```
void Dual_Init_1() {
```

```
/* Pointer addresses set for Dual Port Ram between Yoda and Obi-Wan */
```

```
*(Switches = (0xC000L << 16)) = 0x00;
*(Pneu_View = ((0xC000L << 16) + 0x01L)) = 0x00;
*(State_Stuff = ((0xC000L << 16) + 0x201L)) = 0x0F;
*(MotorSigns = ((0xC000L << 16) + 0x02L)) = 0x00;
*(thc = ((0xC000L << 16) + 0x09L)) = 0x00;
*(rhc = ((0xC000L << 16) + 0x0CL)) = 0x00;
*(status = ((0xC000L << 16) + 0x1EL)) = 0x00;
*(Pstatus = ((0xC000L << 16) + 0x1FL)) = 0x00;
*(Depth = ((0xC000L << 16) + 0x10L)) = 0x800;
*(Pendula = ((0xC000L << 16) + 0x12L)) = 0x000;
*(Gyros = ((0xC000L << 16) + 0x18L)) = 0x800;
```

```

*(path      = ((0xC000L << 16) + 0x300L)) = 0x00;
*(PID       = ((0xC000L << 16) + 0x301L)) = 0xFE;
*(gainID    = ((0xC000L << 16) + 0x302L)) = 0x00;
*(gainval   = ((0xC000L << 16) + 0x303L)) = 0x0000;
*(stateID   = ((0xC000L << 16) + 0x305L)) = 0x00;
*(stateval  = ((0xC000L << 16) + 0x306L)) = 0x00;
*(Motors    = ((0xC000L << 16) + 0x03L))  = 0x00;
*(astate    = ((0xC000L << 16) + 0x100L)) = 0x0000;
*(arange    = ((0xC000L << 16) + 0x1A0L)) = 0x0000;
*(athumpID  = ((0xC000L << 16) + 0x200L)) = 0x00;
}

/ *          obilando.c          * /
/ *  Written by: Ella M. Atkins  * /
/ *  Last modified: 3/23/90      * /

#include "obilando.h"

void Dual_Init_2() {

/* Pointer addresses set for Dual Port Ram between Obi-Wan and Lando * /

*(bSwitches = ((0xC000L << 16) + 0x8000)) = 0x00;
*(bState_Stuff = ((0xC000L << 16) + 0x8201)) = 0x0F;
*(bthc       = ((0xC000L << 16) + 0x8009)) = 0x00;
*(brhc       = ((0xC000L << 16) + 0x800C)) = 0x00;
*(bstatus    = ((0xC000L << 16) + 0x801E)) = 0x00;
*(bDepth     = ((0xC000L << 16) + 0x8010)) = 0x800;
*(bPendula   = ((0xC000L << 16) + 0x8012)) = 0x00;
*(bGyros     = ((0xC000L << 16) + 0x8018)) = 0x800;
*(state      = ((0xC000L << 16) + 0x8100)) = 0x0000;
*(range      = ((0xC000L << 16) + 0x81A0)) = 0x0000;
*(thumpID    = ((0xC000L << 16) + 0x8200)) = 0x00;
*(actuator   = ((0xC000L << 16) + 0x8210)) = 0x00;
*(gate       = ((0xC000L << 16) + 0x8280)) = 0x8FFF;
INT_DUAL     = ((0xC000L << 16) + 0x83FE);
*(DPR_num    = ((0xC000L << 16) + 0x83F0)) = 0x00;

}

/ *          obiwan.h          * /
/ *  Written by: Ella M. Atkins  * /
/*  Last modified: 4/2/90      * /

#ifndef YODAABI
#include "..\yodaobi.h"
#endif
#ifndef OBILANDO
#include "..\obilando.h"
#endif

#include <stdio.h>

```

```

#ifndef OBIWAN
#define OBIWAN

typedef unsigned char Byte, *BytePtr, Logical;

/ *      Obi-Wan Definitions      * /

Byte          Torque[3], Trans[3], DapsFlag, gainflag;
unsigned short gyroup[3], gyrolow[3];
FILE          *rangefile;

/ *      8255 initializations for MIC      * /

#define P0A    0x318
#define P0B    0x31A
#define P0C    0x31C
#define P0X    0x31E

#define P1A    0x320
#define P1B    0x322
#define P1C    0x324
#define P1X    0x326

/* Definitions in *Switches used by Obi-Wan * /

#define SAVE_DATA        0x01
#define STATE_CALC       0x02
#define OBI_ESCAPE       0x10
#define LANDO_ESCAPE     0x20
#define ESCAPE           0x40

/* Definitions in *Pneu_View used by Obi-Wan */

#define POWER            0x10 /* main power          * /
#define RAM              0x04 /* flag to STOP CLCNTL routine */
#define HYDRO_SEE       0x40 /* Uplink sensor data * /
#define STATE_SEE       0x80 /* Uplink calculated state vector */

/* Control definitions in *path */

#define ATT_HOLD        0x02
#define POS_HOLD        0x04
#define AP_HOLD         0x08
#define ENTER_ATT       0x10
#define ENTER_POS       0x20
#define DOCK            0x80

/* Control definitions in *PID */

#define CL_CTRL         0x01
#define PPOS_CTRL       0x02
#define IPOS_CTRL       0x04
#define DPOS_CTRL       0x08
#define PATT_CTRL       0x20

```

```

#define IATT_CTRL          0x40
#define DATT_CTRL          0x80

#define TRUE              1
#define FALSE             0
#define NULL              0

/ *    Obi-Wan specific routines    * /

void          Daps(), Obilnit(), JetSelect(), CtrlInit();
void          path_calc(), ctrl_calc(), cosine_calc();
void interrupt far  DapsInt();
void interrupt far  Busy();

#endif

/ *          obiwan.c          * /
/ *    Written by: Ella M. Atkins    * /
/ *    Last modified: 4/2/90    * /

#include "obiwan.h"
#include <dos.h>
#include <gf.h>
#include <asiports.h>
#include <math.h>

main() {    /* Begin Obi-Wan main driver program */

    register i, j;
    static unsigned char count = 0;
    unsigned getkey();
    double far *ydex;
    int far *aydex;
    FILE    *sensorfile, *statefile;

    if ((rangefile = fopen("range.dat","a")) == NULL)
        exit(0);
    if ((sensorfile = fopen("sensor.dat","a")) == NULL)
        exit(0);
    if ((statefile = fopen("state.dat","a")) == NULL)
        exit(0);

/* Enable and initialize IRQ9 and IRQ10 */

    outp(0x21, 0xB8);
    outp(0xA1, 0xD8);
    _dos_setvect(0x71, DapsInt);    /* Hardware IRQ9 */
    _dos_setvect(0x72, Busy);    /* Hardware IRQ10 */
    outp(0xA0,0x20);
    outp(0x20,0x20);

/* Initialize variables and Dual Port Ram */

```

```

Dual_Init_1();
Dual_Init_2();
Obilnit();
CtrlInit();

/* Begin program loop */

while (!( (*status) & OBI_ESCAPE)) {
    if (kbhit()) if (getkey() == ESC) break;

/* Save data if requested */

    if (( *Switches) & SAVE_DATA) {
        if (( *status) & SAVE_DATA) {
            fprintf(rangefile, "Range Data... \n");
            fprintf(sensorfile, "Sensor Data... \n");
            fprintf(statefile, "State Data... \n");
            *status &= (~SAVE_DATA);
        }
        if (count >= 75) {
            fprintf(sensorfile, "%05u \t %05u \t %05u \t %05u \t %05u \t %05u \t %05u \t
                %03d \t %03d \t %03d \t %03d \t %03d \t %03d \n", *Gyros,
                *(Gyros+1), *(Gyros+2), *Depth, *Pendula, *(Pendula+1), *(Pendula+2),
                *Motors, *(Motors+1), *(Motors+2), *(Motors+3), *(Motors+4),
                *(Motors+5));
            ydex = state;
            for (i = 0; i < 16; i++)
                fprintf(statefile, "%7.4f \t", *ydex++);
            fprintf(statefile, "\n");
            count = 0;
        } else count++;
    }

/* Thruster Control calculations */

    if (*Pneu_View & POWER) {
        if (*PID & CL_CTRL) { /* Closed loop control */
            path_calc();
            ctrl_calc();
        } else { /* open loop control from hand controllers */
            for (i = 0; i <= 2; i++) {
                if (*rhc & (0x01 << (i+i))) Torque[i] = 0xFF;
                else if (*rhc & (0x01 << (1+i+i))) Torque[i] = 0x00;
                else Torque[i] = 0x80;
                if (*thc & (0x01 << (i+i))) Trans[i] = 0xFF;
                else if (*thc & (0x01 << (1+i+i))) Trans[i] = 0x00;
                else Trans[i] = 0x80;
            }
            JetSelect(Trans[0], Torque[2], Motors); /* x-motors */
            JetSelect(Trans[1], Torque[0], Motors+2); /* y-motors */
            JetSelect(Trans[2], Torque[1], Motors+4); /* z-motors */
        }
    }
}

```



```

/* 3DAPS reading. Flag set from 3DAPS interrupt */

    if (DapsFlag) { /* Flag set from 3DAPS interrupt */
        Daps();
        DapsFlag = 0x00;
    }

/* Dual Port Ram transfers between Yoda and Lando */

if (*status & LANDO_ESCAPE) {
    *bstatus |= LANDO_ESCAPE;
    *status &= (~LANDO_ESCAPE);
} else if (*status & STATE_CALC) {
    *bstatus |= STATE_CALC;
    *status &= (~STATE_CALC);
}
*bSwitches = *Switches;
if (*Switches & STATE_CALC) {
    *bState_Stuff = *State_Stuff;
    for (i = 0; i < 3; i++) {
        *(bPendula+i) = *(Pendula+i);
        *(bGyros+i) = *(Gyros+i);
        *(actuator+i) = Trans[i];
        *(actuator+3+i) = Torque[i];
    }
    *bDepth = *Depth;
}
if (*Pneu_View & STATE_SEE) { /* Convert to 2 Byte messages */
    ydex = state;
    aydex = astate;
    for (i = 0; i < 3; i++)
        (*aydex++) = (int) (100.0*(*ydex++));
    for (i = 0; i < 4; i++)
        (*aydex++) = (int) (1000.0*(*ydex++));
    for (i = 0; i < 3; i++)
        (*aydex++) = (int) (100.0*(*ydex++));
    for (i = 0; i < 6; i++)
        (*aydex++) = (int) (57.296*(*ydex++));
}
} /* End main driver loop */

if (*status & OBI_ESCAPE) {
    *status ^= OBI_ESCAPE;
    *Switches &= (~OBI_ESCAPE); }
outp(0x21, 0xBC); /* Disable Hardware IRQ9 & IRQ10 */
outp(0xA1, 0xDE);
fclose(rangefile);
fclose(sensorfile);
fclose(statefile);
if (*status & ESCAPE) {
    *bSwitches = *Switches;
    *bstatus = *status; }
exit(0);

```

```

}

void Obilnit() {

    register i, j, temp;

    /* Initialize the 8255's on MIC */

    outpw(P0X, 0x9292); /* Initialized with Port A, B input, */
    outpw(P1X, 0x9292); /* Port C output */

    /* Handshake with 68HC11's in case thumper has already activated HC11 */

    outpw(P0C, 0x0100);
    outpw(P1C, 0x0100);
    for (i = 0; i < 0x8888; i++); { }
    outpw(P0C, 0x0000);
    outpw(P1C, 0x0000);
    for (i = 0; i < 8888; i++) { } /* Delay to give HC11 proc. time */
    outpw(P0X, 0x8282);
    outpw(P1X, 0x8282);
    outpw(P0A, 0x0000);
    outpw(P1A, 0x0000);
    outpw(P0C, 0x0100);
    outpw(P1C, 0x0100);
    for (i = 0; i < 8888; i++) { }
    outpw(P0X, 0x9292);
    outpw(P1X, 0x9292);
    outpw(P0C, 0x0000);
    outpw(P1C, 0x0000);

    /* Initialize variables to inert values */

    DapsFlag = 0x00;
    for (i = 0; i < 3; i++) {
        Trans[i] = 0x80;
        Torque[i] = 0x80; }

    for (j = 0; j < 8; j++) { /* Initialize DPR ranges and gates */
        temp = j<<2;
        for (i = 0; i < 4; i++) {
            *(gate+i+temp) = 0x00;
            *(range+i+temp) = 0xEEEE;
            *(arange+i+temp) = 0xEEEE;
        }
    }

    for (j = 0; j < 16; j++) { /* Initialize DPR states */
        *(astate+j) = 0x0000;
        *(state+j) = 0.00;
    }
}

```

```

/ *          obifuns.c          * /
/* Written by: Ella M. Atkins and Robert M. Sanner    * /
/* Last modified: 4/2/90          * /

#include "obiwan.h"
#include <dos.h>
#include <gf.h>

#define STRIP(X)  ( ( (X)&127 ) >> 3 )
#define TMAX  0x0F

void JetSelect(xlat, rotat, mot)
    Byte      xlat, rotat;
    char far  *mot;
{
    char StripXlat, StripRotat, tmp;

    if ( (StripXlat = (xlat >> 3) - 16) < 0) StripXlat++;
    if ( (StripRotat = (rotat >> 3) - 16) < 0) StripRotat++;

    if ( (tmp = StripXlat - StripRotat) < -TMAX) tmp = -TMAX;
        else if (tmp > TMAX) tmp = TMAX;
    *mot++ = tmp;
    if ( (tmp = StripXlat + StripRotat) > TMAX) tmp = TMAX;
        else if (tmp < -TMAX) tmp = -TMAX;
    *mot = tmp;
}

void Daps() { /* Routine for reading 3DAPS data from 68HC11's */
    short i;
    short j;
    unsigned short gate1, gate2;

/* Read thumper ID    * /

    *thumpID = (Byte) ((inpw(P1B)) & 0x0007);
    j = (*thumpID) << 2;
    if ((*Switches) & SAVE_DATA) {
        fprintf(rangefile, "%02u \t", *thumpID);
    }

/* Read H0 and H2 of current thumper    * /
    *(range+j) = ((inpw(P0A) & 0x00FF) | (inpw(P0B) & 0xFF00));
    *(range+j+2) = ((inpw(P1A) & 0x00FF) | (inpw(P1B) & 0xFF00));

/* Signal 68HC11's to prepare for 286 read of H1 and H3 */
    outpw(P0C, 0x0100);
    outpw(P1C, 0x0100);
    if ((*Switches) & SAVE_DATA) {
        fprintf(rangefile, "%05u \t %05u \t", *(range+j), *(range+j+2));
    }
    while (!(inpw(P0B) & 0x0010) || !(inpw(P1B) & 0x0010)) {
}

```

```

/* Read H1 and H3      */
*(range+j+1) = ((inpw(P0A) & 0x00FF) | (inpw(P0B) & 0xFF00));
*(range+j+3) = ((inpw(P1A) & 0x00FF) | (inpw(P1B) & 0xFF00));

/* Signal 68HC11's to prepare for range gate receipt */

    outpw(P0C, 0x0000);
    outpw(P1C, 0x0000);
    if ((*Switches) & SAVE_DATA) {
        fprintf(rangefile, "%05u \t %05u \n",*(range+j+1), *(range+j+3));
    }

    *INT_DUAL = 0x00; /* Signal Lando about new hydrophone data */
    if ((*Pneu_View) & HYDRO_SEE) {
        for (i = 0; i < 4; i++) {
            *(arange+j+i) = *(range+j+i); }
        }
    while ((inpw(P0B) & 0x0010) || (inpw(P1B) & 0x0010)) {
        }

/* Output Range gates to 68HC11's      */
    outpw(P0X, 0x8282); /* Initialize 8255's with Port A output */
    outpw(P1X, 0x8282);

    gate1 = (unsigned short) ((*gate+j)<<8 | *(gate+j+2));
    gate2 = (unsigned short) ((*gate+j+1)<<8 | *(gate+j+3));

    outpw(P0A, gate1);
    outpw(P1A, gate2);

    outpw(P0C, 0x0100); /* Signal HC11's to read range gates */
    outpw(P1C, 0x0100); /* You're so forward, Matt... */

/* Wait until HC11 has read gates */

    while (!(inpw(P0B) & 0x0010) || !(inpw(P1B) & 0x0010)) {
        }
        outpw(P0X, 0x9292); /* Re-Initialize 8255's with Port A input */
        outpw(P1X, 0x9292);
        outpw(P0C, 0x0000);
        outpw(P1C, 0x0000);
    }

void interrupt far DapsInt() {
    DapsFlag = 0x01; /* Set flag for 3DAPS routine */
    outp(0xA0, 0x20); /* Reset 8259A's */
    outp(0x20, 0x20);
    return;
}

void interrupt far Busy() {
    /* Interrupt produces delay for Dual-Port Ram writing */
    printf("Busy");
}

```

```

    outp(0xA0, 0x20); /* Reset 8259A's */
    outp(0x20, 0x20);
    return;
}

/* ctricalc.c */
/* Written by: Ella M. Atkins */
/* Last modified: 4/13/90 */

#include "obiwan.h"
#include <math.h>
#include <stdio.h>

static unsigned char pathflag;
static double Gain[2][9], C[3][3];
static double *Cib, yaw, pitch, roll;
static double xwanted[13], xerror[13], steadyx[3], deltax[12];

void path_calc() /* MPOD trajectory planning routine */
/* Note: This program is incomplete. */
{
    register i;
    unsigned char newpath, dock_stage;

    if (pathflag != *path) {
        newpath = 0x01;
        if (pathflag <= 0x08) gainflag = 0x00; /* Att/Pos Hold gains */
        else gainflag = 0x01; /* Att/Pos Maneuver gains */
        pathflag = *path;
    }

    if (pathflag & ATT_HOLD) { /* Attitude Hold */
        if (newpath) { /* Compute Euler angles, direction cosines */
            cosine_calc();
            newpath = 0x00;
        }
        if (*thc != 0x00) { /* Incorporate THC commands with Attitude hold */
        }
    }

    } else if (pathflag & POS_HOLD) { /* Position Hold */
        if (newpath) {
            for (i = 0; i < 3; i++)
                steadyx[i] = *(state+i);
            newpath = 0x00;
        }
        if (*rhc != 0x00) { /* Incorporate RHC commands with Position hold */
        }
    }

    } else if (pathflag & AP_HOLD) { /* Attitude & Position Hold */
        if (newpath) {
            for (i = 0; i < 3; i++)
                steadyx[i] = *(state+i);
            cosine_calc();
        }
    }
}

```

```

    newpath = 0x00;
}

} else if (pathflag & ENTER_ATT) { /* User-defined final Attitude */
    if (newpath) { /* Hold position during attitude change */
        for (i = 0; i < 3; i++)
            steadyx[i] = *(state+i);
        newpath = 0x00;
    }

} else if (pathflag & ENTER_POS) { /* User-defined final Position */
    if (newpath) { /* Hold attitude during position change */
        cosine_calc();

        newpath = 0x00;
    }

} else if (pathflag & DOCK) { /* MPOD Automatic Docking */
    if (newpath) {
        dock_stage = 0x01;
        newpath = 0x00;
    }
    if (dock_stage == 0x01) { /* Obtain upright attitude */

    } else if (dock_stage == 0x02) { /* Move to 2m in front of target */

    } else if (dock_stage == 0x03) { /* Translate in x until docked */
        if (*Switches & RAM) { /* Flag signifying docking completion */
            pathflag = 0x00;
            *path = 0x00;
        }
    }
}
}

void CtrlInit() /* MPOD control parameter initialization */
{
    register i, j;
    for (i = 0; i < 2; i++) {
        for (j = 0; j < 9; j++)
            Gain[i][j] = 10.0;
    }
    pathflag = 0x00;
    for (i = 0; i < 13; i++) {
        xwanted[i] = 0.00;
        xerror[i] = 0.00;
        if (i < 12)    deltax[i] = 0.00;
        if (i < 3)    steadyx[i] = 0.00;
    }
}

```

Appendix B.3 Lando Software

```
/*          lando          * /
/ *      Microsoft C Make file      * /

lando.obj: lando.h filter.h lando.c
           cl /c /FPi87 lando.c

filter2.obj: filter.h lando.h filter2.c
            cl /c /FPi87 /Od filter2.c

pstate.obj: filter.h lando.h pstate.c
            cl /c /FPi87 /Od pstate.c

landfuns.obj: lando.h filter.h landfuns.c
             cl /c /FPi87 landfuns.c

lando.exe: lando.obj filter2.obj pstate.obj landfuns.obj ..\obilando.obj
           link /NOD $**, lando.exe,SLIBC7+GFCS+GFS
```

```
/ *      lando.h          * /
/* Written by: Ella M. Atkins      * /
/* Last modified: 5/2/90          * /
```

```
#ifndef OBILANDO
#include "..\obilando.h"
#endif
```

```
#include <stdio.h>
```

```
#ifndef LANDO
#define LANDO
```

```
unsigned char  newdata;
```

```
/* Definitions in *bSwitches used by Lando */
```

```
#define STATE_CALC          0x02
#define LANDO_ESCAPE       0x20
```

```
/* Definitions in *bState_Stuff used by Lando */
```

```
#define USE_RANGES         0x01
#define USE_PENDULA        0x02
#define USE_DEPTH          0x04
#define USE_RATE           0x08
```

```
/* Useful definitions */
```

```
#define READ_DATA          0x01
#define USE_DATA           0x02
#define ROLLBIAS           -0.26
```

```

#define MAX_ERROR      5.0          /* Maximum allowable range error */
#define RANGE_OFFSET   1.011       /* distance before counter begins */
#define RANGE_FACTOR   0.000748   /* Ranges -- counts to meters */
#define PEND_FACTOR    0.001534    /* Pendula -- counts to radians */
#define RATE_FACTOR    0.001534    /* Gyros -- 12-bit count to rad/sec */
#define DEPTH_FACTOR   0.01662     /* Depth Sensor -- 12-bit count to m */
#define DEPTH_ZERO     0x97E       /* Depth Sensor -- Surface reading */
#define THRUST_FACTOR  0.9449      /* Converting #'s to Newtons */
#define TORQUE_FACTOR  0.8031      /* Converting #'s to N-m's */

```

```
/* Subroutines used in main driver loop */
```

```
void LandInIt();
void FilterInIt();
```

```
void interrupt far Busy();
void interrupt far New_3DAPS();
```

```
#endif
```

```

/ *          filter.h          * /
/*      Written by: Robert M. Sanner      * /
/* Last modified by: Ella M. Atkins 4/11/90 * /

```

```
#ifndef FILTER_H
#define FILTER_H
```

```
double yhat[16], P[16][16], yVar[16];
```

```
/* Filter subroutine prototypes */
```

```
void theFilter(); /* double dt, int measNum, double meas, double measVar,
                  double *stateVar); */
```

```
void propState(); /* double dt, double *stateVar); */
```

```
/* 3DAPS configuration information (from KGK thesis) */
```

```

#define T0 {-6.300, 4.935, 0.15}
#define T1 {-6.300, 4.935, 3.09}
#define T2 {-6.300, -4.935, 0.15}
#define T3 {-6.300, -4.935, 3.09}
#define T4 {6.300, 4.935, 0.15}
#define T5 {6.300, 4.935, 3.09}
#define T6 {6.300, -4.935, 0.15}
#define T7 {6.300, -4.935, 3.09}

```

```
#define EPS 1.0e-02
```

```

#define L0 {0.0, 0.0, -1.43}
#define L1 {0.0, -1.43, 0.0}
#define L2 {-1.44, 0.0, 0.0}
#define L3 {0.0, 1.434, 0.0}

```



```

/ *      MPOD physical parameter data (From Joe P. and Ella tests)   */

#define      CDTX  480.0 /* Estimated from Cd(trans) = (Fmax/((Vt)**2)) */
#define      CDTY  480.0 /* N/((m/sec)**2) */
#define      CDTZ  480.0

#define      CDRX  186.3 /* Estimated from Cd(rot) = (Tmax/((wmax)**2)) */
#define      CDRY  265.3 /* N*m / ((rad/sec)**2) */
#define      CDRZ  265.3

#define      MT    1000.0

#define      IXX   80.5 /* Estimated from lumped box and */
#define      IYY   85.9 /* thin spherical shell model */
#define      IZZ   94.1

/ *      Derived constants      */

#define      INVMT      1./MT
#define      I2MI3      (IYY - IZZ)
#define      I1MI2      (IXX - IYY)
#define      I3MI1      (IZZ - IXX)
#define      INVIX      1./IXX
#define      INVIY      1./IYY
#define      INVIZ      1./IZZ

#endif

/ *      lando.c      */
/ *      Written by:  Ella M. Atkins      */
/ *      Last modified: 3/16/90      */

#include "lando.h"
#include "filter.h"

#include <dos.h>
#include <ibmkeys.h>
#include <gf.h>
#include <math.h>

double ut[3] = {0.0}, ur[3] = {0.0};

main() {
    register i, j, k;
    static unsigned char choice = 0x00, measflag = 0x0F;
    unsigned short temp;
    static int measID;
    static double dt = 0.02, pitchsign = 1.0;
    static double newmeas[32]; /* Variable for range data */
    double meas, *ydex, temp2;
    double far *DPRydex;

```

```

/ *      Enable and initialize IRQ5 and IRQ9      * /

outp(0x21, 0x98);
outp(0xA1, 0xDC);
_dos_setvect(0x0D, Busy);          / *      Hardware IRQ5      * /
_dos_setvect(0x71, New_3DAPS);    / *      Hardware IRQ9      * /
outp(0xA0, 0x20);
outp(0x20, 0x20);

Dual_Init_2();
Landolnit();
for (i = 0; i < 32; i++) newmeas[i] = 8.0;

while (!((*bstatus) & LANDO_ESCAPE))      {
if (kbhit()) if (getkey() == ESC) break;
if ((*bState_Stuff) != measflag) measflag = (*bState_Stuff);
if (newdata & READ_DATA) {
    temp = (*thumpID)<<2;
    for (i = 0; i < 4; i++) {
        measID = (*thumpID) + (i << 3);
        newmeas[measID] = (((double) (*(range+i+temp)))
            *RANGE_FACTOR)+RANGE_OFFSET;
    }
    newdata = USE_DATA;
}
if ((*bSwitches) & STATE_CALC) {
    if (*bstatus & STATE_CALC) {
        FilterInit();      / *      Initialize state and variances */
        *bstatus &= (~STATE_CALC); }
    if ((newdata & USE_DATA) && (measflag & USE_RANGES))      {
        newdata ^= USE_DATA;
        for (i = 0; i < 4; i++) {
            measID = (*thumpID) + (i << 3);
            if ((newmeas[measID] >= 0.3) &&
                (newmeas[measID] <= 16.0)) {
                ut[0] = (THRUST_FACTOR)*(((double) (*actuator))-128.0);
                ur[0] = (TORQUE_FACTOR)*(((double) *(actuator+3)))-128.0;
                for (j = 1; j < 3; j++) {          /* Read Forces from DPR */
                    ut[j] = (-THRUST_FACTOR)*(((double) *(actuator+j)))-128.0;
                    ur[j] = (-TORQUE_FACTOR)*(((double) *(actuator+3+j)))-128.0;
                }
                meas = newmeas[measID]*newmeas[measID];
                theFilter(dt, measID, meas, 0.5, yVar);
                ydex = yhat;
                DPRydex = state;
                for (j = 0; j < 16; j++)          / *      Store state in DPR */
                    *DPRydex++ = *ydex++;
            }
        }
    }
}
ut[0] = (THRUST_FACTOR)*(((double) (*actuator))-128.0);
ur[0] = (TORQUE_FACTOR)*(((double) *(actuator+3)))-128.0;
for (i = 1; i < 3; i++) { /* Read Control Signals from DPR */
    ut[i] = (-THRUST_FACTOR)*(((double) *(actuator+i)))-128.0;
}

```

```

        ur[i] = (-TORQUE_FACTOR)*(((double) (*(actuator+3+i)))-128.0);
    }
    if ((choice < 0x03) && (measflag & USE_PENDULA)) {
    if (choice == 0x00) {
    temp2 = 3.14159-(((double) *(bPendula+choice))*PEND_FACTOR);
    meas = cos(temp2);
    } else if (choice == 0x01) {
    temp2 = 3.14159+(((double) *(bPendula+choice))*PEND_FACTOR);
    pitchsign = -fabs(temp2-6.28)/(temp2-6.28);
    meas = cos(temp2);
    } else {
    temp2 = pitchsign*(3.14159+(((double) *(bPendula+choice))*PEND_FACTOR));
    meas = pitchsign*cos(temp2); }
    theFilter(dt, (int) (32+choice), meas, 0.05, yVar);
    } else if ((choice == 0x03) && (measflag & USE_DEPTH)) {
    meas = ((double) (*(bDepth)-DEPTH_ZERO))*DEPTH_FACTOR;
    theFilter(dt, 35, meas, 0.1, yVar);
    } else if (measflag & USE_RATE) {
    meas = (((double) *(bGyros+choice-4))-2048.0)*RATE_FACTOR;
    if (choice > 4) meas = -1.0*meas;
    theFilter(dt, (int) (32+choice), meas, 0.1, yVar);
    }

    if (choice >= 0x06) choice = 0x00;
    else choice++;
    ydex = yhat;
    DPRydex = state;
    for (i = 0; i < 16; i++)          / *      Store state in DPR */
    *DPRydex++ = *ydex++;
    }
}
*bstatus &= (~LANDO_ESCAPE);
outp(0x21, 0xBC);          / *      Disable hardware interrupt lines */
outp(0xA1, 0xDE);
exit(0);
}

```

```

/ *      landfuns.c          * /
/* Written by: Ella M. Atkins      * /
/* Last modified: 4/11/90          * /

```

```

#include "lando.h"
#include "filter.h"

```

```

#include <dos.h>
#include <gf.h>

```

```

void Landolnit() {
    register i, j;
    double *Vdex, *ydex, *pdex;
    newdata = *INT_DUAL;      /* In case an interrupt has activated DPR */
    newdata = 0x00;
}

```

```

/* Initialize variables */

Vdex = yVar;
ydex = yhat;
pdex = P[0];

for (i = 0; i < 16; i++) {
    *(state+i) = 2.00; /* DPR state variable initialization */
    *Vdex++ = 0.00001; /* State variance initialization */
    *ydex++ = 0.00; /* State variable initialization */
    for (j = 0; j < 16; j++) /* Variance initialization */
        *pdex++ = 0.0;
}
}

void FilterInit() {
    register i;
    double *ydex, *Pdex;
    ydex = yhat;
    Pdex = yVar;

/* Assumes MPOD is at rest near defined inertial coordinate origin and axes */

    for (i = 0; i < 16; i++) {
        *ydex++ = 0.00;
        *Pdex++ = 0.00001; }

    for (i = 0; i < 3; i++) P[i][i] = 0.2;
    for (i = 3; i < 7; i++) P[i][i] = 0.02;
    for (i = 7; i < 10; i++) P[i][i] = 0.5;
    for (i = 10; i < 13; i++) P[i][i] = 0.05;
    for (i = 13; i < 16; i++) P[i][i] = 0.05;

    *(yhat+13) = ROLLBIAS; /* roll rate sensor bias */
    *(yhat+3) = 1.0; /* Initial quaternion value */
    *(yhat+2) = 2.0; /* expected z value at pool center, not surface
* /
}

void interrupt far New_3DAPS() { /* Alerts Lando to new 3DAPS data */
    newdata = *INT_DUAL; /* Clear Interrupt flag */
    newdata = READ_DATA; /* Flag indicating new data to read */
    outp(0xA0, 0x20); /* Reset 8259A's -- IRQ9 */
    outp(0x20, 0x20);
    return;
}

void interrupt far Busy() { /* Interrupt causes delay for DPR */
    outp(0x20, 0x20); /* Reset 8259A -- IRQ5 */
    return;
}

```

```

/ *      filter.c                * /
/* Written by: Robert M. Sanner  * /
/* Last modified: 2/9/90        * /

#include <math.h>
#include <stdio.h>
#include "lando.h"
#include "filter.h"

double t[8][3] = {T0, T1, T2, T3, T4, T5, T6, T7 };
double l[4][3] = {L0, L1, L2, L3};
static double Ph[16], Cib[3][3], h[16], d[3];

static double x1, x2, x3, q0, q1, q2, q3;
static double q11,q12,q13,q22,q23,q33,q01,q02,q03;

static double Cbi13, Cbi23, Cbi33, Cbi13_2, Cbi23_2, Cbi33_2, alpha;
static double Cbi13_23, Cbi23_33, Cbi13_33, alpha_2;
static double g, a, temp, scaledDiff;

static int thumpNum, hydroNum;

void theFilter(double dt, int measNum, double meas, double measVar,
              double *stateVar)
{
    register          ii, jj, index, index2;
    double *cdex, *ddex, *hdex, *idex, *ldex, *pdex, *ph1dex, *ph2dex;
    double *xdex, *ydex, *thet, *thel;

/ *      Set up local registers with the current estimates      * /

    ydex = yhat;
    x1 = *ydex++;
    x2 = *ydex++;
    x3 = *ydex++;
    q0 = *ydex++;
    q1 = *ydex++;
    q2 = *ydex++;
    q3 = *ydex;

    /* Compute direction cosines only if necessary */

    if (measNum < 35) {

/ *      Set up some useful combinations of state vector elements. */

        q11 = q1*q1;
        q01 = q0*q1;
        q12 = q1*q2;
        q22 = q2*q2;
        q02 = q0*q2;
        q13 = q1*q3;
        q33 = q3*q3;

```

```

q03 = q0*q3;
q23 = q2*q3;

/ *      Compute Cib      * /

idex = Cib[0];

*idex++ = 1.0 - 2.0*(q22 + q33);
*idex++ = 2.0*(q12 - q03);
*idex++ = 2.0*(q13 + q02);

*idex++ = 2.0*(q12 + q03);
*idex++ = 1.0 - 2.0*(q11 + q33);
*idex++ = 2.0*(q23 - q01);

*idex++ = 2.0*(q13 - q02);
*idex++ = 2.0*(q23 + q01);
*idex  = 1.0 - 2.0*(q11 + q22);
}

/ *      Determine which measurement was taken, and compute the
        necessary measurement geometry quantities      */

if (measNum < 32) {          /* Range Measurements */

    thumpNum = (measNum%8);
    hydroNum = measNum/8;

    /* Select the correct thumper-hydrophone pair for this meas. */

    thet = t[thumpNum];
    thel = l[hydroNum];

    /* Compute the expected measurement using our current estimate
       of the state vector. */

    cdex = *Cib; ddex = d;

    *ddex++ = x1 - (*thet++);
    *ddex++ = x2 -(*thet++);
    *ddex  = x3 - (*thet);

    for (ii = 0, ddex = d; ii < 3; ii++, ddex++)
        for (jj = 0, ldex = thel; jj < 3; jj++)
            *ddex += (*cdex++) * (*ldex++);

    g = 0.0;
    for (ii = 0, ddex = d; ii < 3; ii++, ddex++)
        g += (*ddex)*(*ddex);

    / *      Compute the meas. geom. sensitivity vector      * /

    hdex = &h[3];
    ddex = d;
    temp = *ddex++;

```

```

*hdex++ = (q2*thel[2] - q3*thel[1])*temp;
*hdex++ = (q2*thel[1] + q3*thel[2])*temp;
*hdex++ = (q1*thel[1] + q0*thel[2] - 2.0*q2*thel[0])*temp;
*hdex   = (q1*thel[2] - q0*thel[1] - 2.0*q3*thel[0])*temp;

hdex = &h[3];
temp = *ddex++;
*hdex++ += (q3*thel[0] - q1*thel[2])*temp;
*hdex++ += (q2*thel[0] - q0*thel[2] - 2.0*q1*thel[1])*temp;
*hdex++ += (q1*thel[0] + q3*thel[2])*temp;
*hdex   += (q2*thel[2] + q0*thel[0] - 2.0*q3*thel[1])*temp;

hdex = &h[3];
temp = *ddex;
*hdex++ += (q1*thel[1] - q2*thel[0])*temp;
*hdex++ += (q3*thel[0] + q0*thel[1] - 2.0*q1*thel[2])*temp;
*hdex++ += (q3*thel[1] - q0*thel[0] - 2.0*q2*thel[2])*temp;
*hdex   += (q1*thel[0] + q2*thel[1])*temp;

hdex = h; ddex = d;
for (ii = 0; ii < 3; ii++)
    *hdex++ = 2.0 * (*ddex++);
for (ii = 0; ii < 4; ii++) *hdex++ *= 4.0;
for (ii = 0; ii < 9; ii++) *hdex++ = 0.0;
}

else if (measNum < 35) {          /* Angle measurements */

/ *      Set up some useful combinations of direction cosines.      * /

cdex = *Cib;

Cbi13 = *cdex++;
Cbi23 = *cdex++;
Cbi33 = *cdex;

Cbi23_2 = Cbi23*Cbi23;
Cbi13_2 = Cbi13*Cbi13;
Cbi33_2 = Cbi33*Cbi33;

hdex = &h[3];

/ *      Compute the expected measurement and local gradient,
          given our current state estimate.      */

if (measNum == 32) {            /* "Roll" pendulum */

    alpha_2 = Cbi23_2 + Cbi33_2;
    alpha = sqrt(alpha_2);
    if (alpha < EPS) return;
    g = Cbi33/alpha;
    Cbi23_33 = Cbi23*Cbi33;

```

```

        *hdex++ = Cbi23_33*q1;
        *hdex++ = 2.0*q1*Cbi23_2 + Cbi23_33*q0;
        *hdex++ = 2.0*q2*Cbi23_2 + Cbi23_33*q3;
        *hdex  = Cbi23_33*q2;

        temp = -2.0/( alpha_2 * alpha );
    }

    else if (measNum == 33) {          /* "Pitch" pendulum */

        alpha_2 = Cbi13_2 + Cbi33_2;
        alpha = sqrt(alpha_2);
        if (alpha < EPS) return;
        g = Cbi33/alpha;
        Cbi13_33 = Cbi13*Cbi33;

        *hdex++ = Cbi13_33*q2;
        *hdex++ = 2.0*q1*Cbi13_2 + Cbi13_33*q3;
        *hdex++ = 2.0*q2*Cbi13_2 + Cbi23_33*q3;
        *hdex  = Cbi13_33*q2;

        temp = -2.0/( alpha_2 * alpha );
    }

    else if (measNum == 34) {          /* "Yaw" pendulum */

        alpha_2 = Cbi13_2 + Cbi23_2;
        alpha = sqrt(alpha_2);
        if (alpha < EPS) return;
        g = Cbi13/alpha;
        Cbi13_33 = Cbi13*Cbi33;
        Cbi23_33 = Cbi23*Cbi33;

        *hdex++ = -( q2*Cbi23_2 + Cbi13_23*q1 );
        *hdex++ =  q3*Cbi23_2 - Cbi13_23*q0;
        *hdex++ = -(q0*Cbi23_2 + Cbi13_23*q3);
        *hdex  =  q1*Cbi23_2 - Cbi13_23*q2;

        temp = 2.0/( alpha_2 * alpha );
    }

    /* xcg does not effect these measurements at all */

    for (ii = 0, hdex = h; ii < 3; ii++) *hdex++ = 0.0;
    for (ii = 0; ii < 4; ii++) *hdex++ *= temp;
    for (ii = 0; ii < 9; ii++) *hdex++ = 0.0;
}

else if (measNum == 35) {          /* Depth Sensor */

    g = x3;

```



```

        for (ii = 0, hdex = h; ii < 16; ii++) *hdex++ = 0.0;
        h[2] = 1.0;
    }

    else if (measNum < 39) {          / *      Gyro Readings      * /

        index = measNum - 26;
        index2 = index + 3;

        g = yhat[index] + yhat[index2];
        for (ii = 0, hdex = h; ii < 16; ii++) *hdex++ = 0.0;
        h[index] = h[index2] = 1.0;

    }

    else {

        printf("Invalid measurement selector in theFilter.\n");
        return;

    }

    / *
    * /      first, propagate the state and covariance:

    if (dt > 0.0) propState(dt, stateVar);

    / *
    * /      ...now compute Ph:

    ph1dex = Ph; pdex = P[0];
    for (ii = 0; ii < 16; ii++, ph1dex++) {
        *ph1dex = 0.0;
        for (jj = 0, hdex = h; jj < 16; jj++)
            *ph1dex += (*pdex++) * (*hdex++);
    }

    / *
    * /      ...next compute a:

    a = measVar;
    hdex = h; ph1dex = Ph;
    for (ii = 0; ii < 16; ii++) a += (*hdex++) * (*ph1dex++);

    / *
    * /      ...and update the state estimate, yhat:

    scaledDiff = (meas - g)/a;
    ph1dex = Ph; ydex = yhat;
    for (ii = 0; ii < 16; ii++)
        (*ydex++) += scaledDiff * (*ph1dex++);

    /* ...last, but hardly least, update the covariance matrix:

```

NOTE: since we assume P symmetric we can reuse the vector Ph in the update equation for P!

```

* /

pdex = P[0]; ph1dex = Ph;
for (ii = 0; ii < 16; ii++, ph1dex++) {
    temp = (*ph1dex)/a;
    for (jj = 0, ph2dex = Ph; jj < 16; jj++)
        *pdex++ -= temp * (*ph2dex++);
}

/ *    With luck, that's it...give the caller the vector and exit */

idex = ydex = &yhat[4];
q1 = *ydex++;
q2 = *ydex++;
q3 = *ydex;
temp = q1*q1 + q2*q2 + q3*q3;
if (temp > 1.0) {
    temp = sqrt(temp);
    yhat[3] = 0.0;
    for (ii = 0; ii < 3; ii++) *idex++ /= temp;
} else
    yhat[3] = sqrt(1.0 - temp);

*DPR_num = measNum;
}

/ *    pstate.c                * /
/* Written by: Robert M. Sanner * /
/* Last Modified: 2/9/90      * /

#include <stdio.h>
#include <math.h>
#include "filter.h"

#define ABS(X) ( (X) < 0 ? -(X) : (X) )
#define QBIGTOL    1.01
#define QSMTOL    0.99

extern double ut[3], ur[3];

void propState(double dt, double *stateVar)
{
    static double    halfdt, k1[13] = {0.}, k2[13] = {0.};
    static double    x1, x2, x3, v1, v2, v3;
    static double    q0, q1, q2, q3, w1, w2, w3;
    static double    absv1, absv2, absv3, absw1, absw2, absw3;
    static double    uix, uiy, uiz;
    static double    cib11, cib12, cib13, cib21, cib22, cib23;
    static double    cib31, cib32, cib33;
    static double    q11,q22,q33,q01,q02,q03,q12,q13,q23;

```

```

static double  check, correct;

static double  *idx, *jdx, *kdx;
register       ii;

halfdt = dt/2.0;

/ *          Set up local storage for estimated variables      * /

idx = yhat;

x1 = *idx++; x2 = *idx++; x3 = *idx++;
q0 = *idx++; q1 = *idx++; q2 = *idx++; q3 = *idx++;
v1 = *idx++; v2 = *idx++; v3 = *idx++;
w1 = *idx++; w2 = *idx++; w3 = *idx;

absv1 = ABS(v1); absv2 = ABS(v2); absv3 = ABS(v3);
absw1 = ABS(w1); absw2 = ABS(w2); absw3 = ABS(w3);

q11 = q1*q1; q22 = q2*q2; q33 = q3*q3;
q03 = q0*q3; q13 = q1*q3; q23 = q2*q3;
q02 = q0*q2; q12 = q1*q2; q01 = q0*q1;

cib11 = 1 - 2.*(q22 + q33);
cib12 = 2.*(q12 + q03);
cib13 = 2.*(q13 - q02);

cib21 = 2.*(q12 - q03);
cib22 = 1. - 2.*(q11 + q33);
cib23 = 2.*(q23 + q01);

cib31 = 2.*(q13 + q02);
cib32 = 2.*(q23 - q01);
cib33 = 1. - 2.*(q11 + q22);

uix = cib11*ut[0] + cib12*ut[1] + cib13*ut[2];
uiy = cib21*ut[0] + cib22*ut[1] + cib23*ut[2];
uiz = cib31*ut[0] + cib32*ut[1] + cib33*ut[2];

/ *      Compute the derivatives and store in k1          * /

idx = k1;

/ *      fx      * /

*idx++ = v1; *idx++ = v2; *idx++ = v3;

/ *      fq      * /

*idx++ = -0.5*(q1*w1 + q2*w2 + q3*w3);
*idx++ = 0.5*(q0*w1 + q2*w3 - q3*w2);
*idx++ = 0.5*(q0*w2 - q1*w3 + q3*w1);
*idx++ = 0.5*(q0*w3 + q1*w2 - q2*w1);

```

```

/ *   f v   * /

*idx++ = INVMT*(uix - CDTX*v1*absv1);
*idx++ = INVMT*(uiy - CDTY*v2*absv2);
*idx++ = INVMT*(uiz - CDTZ*v3*absv3);

/ *   f w   * /

*idx++ = INVIX*(ur[0] + I2MI3*w2*w3 - CDRX*w1*absw1);
*idx++ = INVYI*(ur[1] + I3MI1*w1*w3 - CDRY*w2*absw2);
*idx++ = INVIZ*(ur[2] + I1MI2*w2*w1 - CDRZ*w3*absw3);

/ *   Reset registers to yhat + (dt/2)*k1   * /

idx = yhat; jdx = k1;

x1 = *idx++ + halfdt>(*jdx++);
x2 = *idx++ + halfdt>(*jdx++);
x3 = *idx++ + halfdt>(*jdx++);
q0 = *idx++ + halfdt>(*jdx++);
q1 = *idx++ + halfdt>(*jdx++);
q2 = *idx++ + halfdt>(*jdx++);
q3 = *idx++ + halfdt>(*jdx++);
v1 = *idx++ + halfdt>(*jdx++);
v2 = *idx++ + halfdt>(*jdx++);
v3 = *idx++ + halfdt>(*jdx++);
w1 = *idx++ + halfdt>(*jdx++);
w2 = *idx++ + halfdt>(*jdx++);
w3 = *idx  + halfdt(*jdx);

absv1 = ABS(v1); absv2 = ABS(v2); absv3 = ABS(v3);
absw1 = ABS(w1); absw2 = ABS(w2); absw3 = ABS(w3);

q11 = q1*q1; q22 = q2*q2; q33 = q3*q3;
q03 = q0*q3; q13 = q1*q3; q23 = q2*q3;
q02 = q0*q2; q12 = q1*q2; q01 = q0*q1;

cib11 = 1 - 2.*(q22 + q33);
cib12 = 2.*(q12 + q03);
cib13 = 2.*(q13 - q02);

cib21 = 2.*(q12 - q03);
cib22 = 1. - 2.*(q11 + q33);
cib23 = 2.*(q23 + q01);

cib31 = 2.*(q13 + q02);
cib32 = 2.*(q23 - q01);
cib33 = 1. - 2.*(q11 + q22);

uix = cib11*ut[0] + cib12*ut[1] + cib13*ut[2];
uiy = cib21*ut[0] + cib22*ut[1] + cib23*ut[2];
uiz = cib31*ut[0] + cib32*ut[1] + cib33*ut[2];

```

```

/ *      Repeat the derivatives to compute k2      * /

idx = k2;

      / *      f x      * /

*idx++ = v1; *idx++ = v2; *idx++ = v3;

      / *      f q      * /

*idx++ = -0.5*(q1*w1 + q2*w2 + q3*w3);
*idx++ = 0.5*(q0*w1 + q2*w3 - q3*w2);
*idx++ = 0.5*(q0*w2 - q1*w3 + q3*w1);
*idx++ = 0.5*(q0*w3 + q1*w2 - q2*w1);

      / *      f v      * /

*idx++ = INVMT*(uix - CDTX*v1*absv1);
*idx++ = INVMT*(uiy - CDTY*v2*absv2);
*idx++ = INVMT*(uiz - CDTZ*v3*absv3);

      / *      f w      * /

*idx++ = INVIX*(ur[0] + I2MI3*w2*w3 - CDRX*w1*absw1);
*idx++ = INVYI*(ur[1] + I3MI1*w1*w3 - CDRY*w2*absw2);
*idx++ = INVIZ*(ur[2] + I1MI2*w2*w1 - CDRZ*w3*absw3);

/ *      Now increment the state...      * /

idx = yhat; jdx = k2;
for (ii = 0; ii < 13; ii++) *idx++ += dt>(*jdx++);

/ *      ...and increment (minimally) the covariance:      * /

idx = P[0];
for (ii = 0; ii < 16; ii++, idx+=17)
    *idx += dt(*stateVar++);

/ *      Make sure the quaternion does not drift excessively      * /

idx = jdx = &yhat[4];
check = 0.;
for (ii = 0; ii < 3; ii++, idx++) check += (*idx)*(*idx);

if (check >1.0) {
    correct = sqrt(check);
    yhat[3] = 0.0;
    for (ii = 0; ii < 3; ii++) *jdx++ /= correct;
} else
    yhat[3] = sqrt(1.0 - check);
}

```

Appendix B.4 Luke Software

```

/ *          pvluke          * /
/ *  Microsoft C Make file  * /

```

```

pvluke.obj: ..\pivecs\pivecs.h pvluke.h ..\pvyoda.msg pvluke.c
           cl /c pvluke.c

```

```

lukefuns.obj: ..\pivecs\pivecs.h pvluke.h lukefuns.c
            cl /c lukefuns.c

```

```

lukemsgs.obj: ..\pivecs\pivecs.h ..\pivecs\pvdata.h pvluke.h ..\pvyoda.msg lukemsgs.c
            cl /c lukemsgs.c

```

```

pvluke.exe: pvluke.obj lukemsgs.obj lukefuns.obj
           link /NOD $**, pvluke.exe,,SLIBCE+GFCS+GFS+..\pivecs\PIVECS

```

```

/ *          pvluke.msg          * /
/ *  Written by: Robert M. Sanner * /
/ *  Last modified by: Ella M. Atkins, 4/11/90 * /

```

```

#ifndef LUKEMSGS
#define LUKEMSGS

```

```

/***** Recognized Message List *****/
#define COMTEST      0x00 /* Msg 0, No Data      * /
#define COMAOK       0x08 /* Msg 1, " "          * /
#define SHUTDN       0x10 /* Msg 2, " "          * /
#define YODAESC      0x20 /* Msg 4, " "          * /

#define RXMOTORS     0x34 /* Msg 6, 4 Data Bytes */
#define RXPENDULA    0x3F /* Msg 7, 7 " " " " * /
#define RXGYROS      0x47 /* Msg 8, 7 " " " " * /
#define RXHYDRO01    0x4D /* Msg 9, 5 " " " " * /
#define RXHYDRO23    0x55 /* Msg 10, 5 " " " " * /

#define RXPOSITION   0x67 /* Msg 12, 7 " " " " * /
#define RXATTITUDE   0x6F /* Msg 13, 7 " " " " * /
#define RXVELOCITY   0x76 /* Msg 14, 6 " " " " * /
#define RXOMEGA      0x7E /* Msg 15, 6 " " " " * /
#define RXBIAS       0x86 /* Msg 16, 6 " " " " * /

#define TXTHC        0xA0 /* Msg 20, No Data     * /
#define TXRHC        0xA8 /* Msg 21, " " " "     * /
#define TXSWITCH     0xB0 /* Msg 22, " " " "     * /
#define TXCTRL       0xB8 /* Msg 23, " " " "     * /
#define TXSTATE      0xC0 /* Msg 24, " " " "     * /
#define TXGAINS      0xC8 /* Msg 25, " " " "     * /

#define BADMSG       0xFF /* Msg 31, 7 data. Placeholder for Bad msgs */
/*****
#endif

```

```

/ *          pvluke.h          * /
/ *    Written by: Robert M. Sanner    * /
/* Last modified by: Ella M. Atkins, 4/11/90    * /

#ifndef PIVECS
#include "..\pivecs\pivecs.h"
#endif
#ifndef LUKEMSGS
#include "..\pvluke.msg"
#endif

#ifndef LUKE
#define LUKE

extern HandlerFunc    BadMsg, ShutDown, ComCheck, ComAOK, Yod_ESC;
extern HandlerFunc    TX_THC, TX_RHC, TX_Switch, TX_Ctrl, TX_State, TX_Gains;
extern HandlerFunc    RX_Motors, RX_Pendula, RX_Gyros;
extern HandlerFunc    RX_Hydro01, RX_Hydro23, RX_Bias;
extern HandlerFunc    RX_Position, RX_Attitude, RX_Velocity, RX_Omega;

static Handlers      LukeHandlers =
{ComCheck,    ComAOK,    ShutDown,    BadMsg,
Yod_ESC,    BadMsg,    RX_Motors,    RX_Pendula,
RX_Gyros,    RX_Hydro01,    RX_Hydro23,    BadMsg,
RX_Position,    RX_Attitude,    RX_Velocity,    RX_Omega,
RX_Bias,    BadMsg,    BadMsg,    BadMsg,
TX_THC,    TX_RHC,    TX_Switch,    TX_Ctrl,
TX_State,    TX_Gains,    BadMsg,    BadMsg,
BadMsg,    BadMsg,    BadMsg,    BadMsg};

static Headers      LukeMsgs =
{COMTEST,    COMAOK,    SHUTDN,    BADMSG,
YODAESC,    BADMSG,    RXMOTORS,    RXPENDULA,
RXGYROS,    RXHYDRO01,    RXHYDRO23,    BADMSG,
RXPOSITION,    RXATTITUDE,    RXVELOCITY,    RXOMEGA,
RXBIAS,    BADMSG,    BADMSG,    BADMSG,
TXTHC,    TXRHC,    TXSWITCH,    TXCTRL,
TXSTATE,    TXGAINS,    BADMSG,    BADMSG,
BADMSG,    BADMSG,    BADMSG,    BADMSG};

static Byte    LukeHiPri = 5;

Byte    thc, rhc, path, PID, Motors[6], MotorSigns;
Byte    stateID, gainID;
Byte    Switches, status, progesc, STOP, thumpID;
Byte    Pneu_View, Pstatus;
Byte    State_Stuff, Sstatus;
float    stateval;
unsigned short    Pendula[3], Gyros[3], Depth;
unsigned short    range[4][8], gainval;
int    pos[3], quat[4], veloc[3], omega[3], bias[3];

void    LukeInit(), ReadTHC(), ReadRHC(), ReadSW();

```

```

/* Sent to Yoda via Pneu_View */

#define POWER          0x10
#define RAM            0x04
#define LATCH          0x02
#define MOTOR_SEE      0x08
#define SENSOR_SEE     0x20
#define HYDRO_SEE      0x40
#define STATE_SEE      0x80

/* Sent to Yoda via Switches */

#define SAVE_DATA      0x01
#define STATE_CALC     0x02
#define YODA_ESCAPE    0x08 /* Sent to Yoda via YODAESC message */
#define OBI_ESCAPE     0x10
#define LANDO_ESCAPE   0x20
#define ESCAPE          0x40 /* Escape from all MPOD computers */

/* State Calculation parameters; Sent to Yoda via State_Stuff */

#define USE_RANGES     0x01
#define USE_PENDULA    0x02
#define USE_DEPTH      0x04
#define USE_RATE       0x08

/* Control Parameters; Sent to Yoda via path variable */

#define ATT_HOLD       0x02
#define POS_HOLD       0x04
#define AP_HOLD        0x08
#define ENTER_ATT      0x10
#define ENTER_POS      0x20
#define DOCK           0x80

/* Control Parameters; Sent to Yoda via PID variable */

#define CL_CTRL        0x01
#define PPOS_CTRL     0x02
#define IPOS_CTRL      0x04
#define DPOS_CTRL      0x08
#define PATT_CTRL     0x20
#define IATT_CTRL     0x40
#define DATT_CTRL     0x80

/* Define 8255 port addresses and control word */

#define P0A  0x300  / *   Input from THC           */
#define P0B  0x301  / *   Input from RHC           */
#define P0C  0x302  / *   C0-C3 = THC/RHC controls; */
#define P0X  0x303  / *   Control Port             */
#define INIT0 0x92  /* Ports A & B input, Port C output * /
#endif

```



```

/ *      pvluke.c          * /
/* Written by: Ella M. Atkins      * /
/* Last modified: 4/11/90          * /

#include "pvluke.h"
#include "..\pvyoda.msg"
#include <gf.h>
#include <stdio.h>

main() {
    register i;
    unsigned long CurrMsg = 0;
    Byte          messtat;
    static unsigned char count;
    pvInitCom( COM1, 9600, P_ODD, 1);
    pvInitMsg( LukeHandlers, LukeMsgs, LukeHiPri);
    LukeInit();

    while (!STOP) {

        CurrMsg = pvRecv();

        if (Pstatus & MOTOR_SEE) { /* View data from topside */
            if (Pneu_View & MOTOR_SEE) {
                pvRequest(TXMOTORS);
                messtat = pvWorry(RXMOTORS, (Byte) 10, TXMOTORS);
            } else {
                messtat = pvWorry(RXMOTORS, (Byte) 255, TXMOTORS); }
        }
        if (Pstatus & SENSOR_SEE) {
            if (Pneu_View & SENSOR_SEE) {
                pvRequest(TXPENDULA);
                pvRequest(TXGYROS);
                messtat = pvWorry(RXPENDULA, (Byte) 10, TXPENDULA);
                messtat = pvWorry(RXGYROS, (Byte) 10, TXGYROS);
            } else {
                messtat = pvWorry(RXPENDULA, (Byte) 255, TXPENDULA);
                messtat = pvWorry(RXGYROS, (Byte) 255, TXGYROS); }
        }
        if (Pstatus & HYDRO_SEE) {
            if (Pneu_View & HYDRO_SEE) {
                pvRequest(TXHYDRO01);
                pvRequest(TXHYDRO23);
                messtat = pvWorry(RXHYDRO01, (Byte) 10, TXHYDRO01);
                messtat = pvWorry(RXHYDRO23, (Byte) 10, TXHYDRO23);
            } else {
                messtat = pvWorry(RXHYDRO01, (Byte) 255, TXHYDRO01);
                messtat = pvWorry(RXHYDRO23, (Byte) 255, TXHYDRO23); }
        }
        if (Pstatus & STATE_SEE) { /* View state vector from topside */
            if (Pneu_View & STATE_SEE) {
                pvRequest(TXPOSITION);
                pvRequest(TXATTITUDE);
            }
        }
    }
}

```

```

    pvRequest(TXVELOCITY);
    pvRequest(TXOMEGA);
    pvRequest(TXBIAS);
    messtat = pvWorry(RXPOSITION, (Byte) 10, TXPOSITION);
    messtat = pvWorry(RXATTITUDE, (Byte) 10, TXATTITUDE);
    messtat = pvWorry(RXVELOCITY, (Byte) 10, TXVELOCITY);
    messtat = pvWorry(RXOMEGA, (Byte) 10, TXOMEGA);
    messtat = pvWorry(RXBIAS, (Byte) 10, TXBIAS);
} else {
    messtat = pvWorry(RXPOSITION, (Byte) 255, TXPOSITION);
    messtat = pvWorry(RXATTITUDE, (Byte) 255, TXATTITUDE);
    messtat = pvWorry(RXVELOCITY, (Byte) 255, TXVELOCITY);
    messtat = pvWorry(RXOMEGA, (Byte) 255, TXOMEGA);
    messtat = pvWorry(RXBIAS, (Byte) 255, TXBIAS); }
}
ReadSW();    /* Reads keyboard, transmits control changes */
ReadTHC();   /* Reads translational hand controller */
ReadRHC();   /* Reads rotational hand controller */

/* Data saving, state calculation user flags */

if ((status & SAVE_DATA) && (Switches & SAVE_DATA))
    onscreen(7, 40, 0, "Saving Data.... ");
else if (status & SAVE_DATA)
    onscreen(7, 40, 0, "NOT Saving Data");
if ((status & STATE_CALC) && (Switches & STATE_CALC))
    onscreen(7, 5, 0, "Calculating State Vector...");
else if (status & STATE_CALC)
    onscreen(7, 5, 0, "NO CALCULATION of State....");

/* Program escape flags */

if (status & YODA_ESCAPE) {
    onscreen(6, 5, 0, "YODA ESCAPE.....");
    pvRequest( YODAESC ); }
if (status & OBI_ESCAPE)
    onscreen(6, 5, 0, "OBI-WAN ESCAPE.....");
else if (status & LANDO_ESCAPE)
    onscreen(6, 5, 0, "LANDO ESCAPE.....");
else if (status & ESCAPE)
    onscreen(6, 5, 0, "ESCAPING ALL SYSTEMS");

} /* End main driver loop */
printf("Exiting program");
pvExit();

} /* End program */

void Lukelnit() {

    register i;

    /* initialize 8255 on I/O protoboard */

```

```

    outp(P0X,INIT0);      /* ports A & B input, port C output    * /

    thc = 0x00;
    rhc = 0x00;
    status = 0x00;
    Pstatus = 0x00;
    Sstatus = 0x00;
    Switches = 0x00;
    Pneu_View = 0x00;
    State_Stuff = 0x0F; /* Using all sensors to calculate state */
    PID = 0xFE; /* All PID on, CL_CTRL off */
    gainID = gainval = 0x00;
    STOP = 0x00;

    onscreen(8, 5, 0, "THC:");
    onscreen(9, 5, 0, "RHC:");
    onscreen(10, 5, 0, "Switches:");
    onscreen(12, 5, 0, "Gyros/Depth:");
    onscreen(13, 5, 0, "Pendula:");
    onscreen(14, 5, 0, "Gains:");
    onscreen(15, 5, 0, "Motors:");
    onscreen(17, 5, 0, "Pos/Quatern:");
    onscreen(18, 5, 0, "Veloc/Omega:");
    onscreen(19, 5, 0, "Gyro Bias:");
    onscreen(20, 5, 0, "Ranges:");
}

/ *          lukemsgs.c          * /
/ *  Written by: Robert M. Sanner and Ella M. Atkins  * /
/*    Last modified by:    4/11/90    * /

#include "pvluke.h"
#include "..\pvyoda.msg"
#include "..\pivecs\pvdata.h"

MsgHandler ShutDown(msg)
    MsgPtr msg;
{
    return( OK );
}

MsgHandler Yod_ESC(msg)
    MsgPtr msg;
{
    Lukelnit();
    return( OK );
}

MsgHandler TX_Ctrl(msg)
    MsgPtr msg;
{
    Byte    array[3];

```

```

array[0] = RXCTRL;
array[1] = path;
array[2] = PID;
onscreen(10, 55, 0, "%02X \t %02X", path, PID);
pvSend(array, 3);
return( OK );
}

```

```

MsgHandler TX_State(msg)

```

```

MsgPtr msg;
{
    Byte array[6], *point;
    register i;
    array[0] = RXSTATE;
    array[1] = stateID;
    point = &stateval;
    for (i = 2; i < 6; i++)
        array[i] = *point++;
    pvSend(array, 6);
    return( OK );
}

```

```

MsgHandler TX_Gains(msg)

```

```

MsgPtr msg;
{
    Byte array[4], *point;
    array[0] = RXGAINS;
    array[1] = gainID;
    point = &gainval;
    array[2] = *point++;
    array[3] = *point;
    pvSend(array, 4);
    return( OK );
}

```

```

MsgHandler TX_THC(msg)

```

```

MsgPtr msg;
{
    register i;
    Byte array[2];

    array[0] = RXTHC;
    array[1] = thc;
    onscreen(8, 20, 0, "%02X", thc);
    pvSend(array, 2);
    return( OK );
}

```

```

MsgHandler TX_RHC(msg)

```

```

MsgPtr msg;
{
    register i;
    Byte array[2];
}

```

```

    array[0] = RXRHC;
    array[1] = rhc;
    onscreen(9, 20, 0, "%02X", rhc);
    pvSend(array, 2);
    return( OK );
}

MsgHandler TX_Switch(msg)
    MsgPtr msg;
{
    Byte array[4];

    array[0] = RXSWITCH;
    array[1] = Switches;
    array[2] = Pneu_View;
    array[3] = State_Stuff;
    onscreen(10, 20, 0, "%02X \t %02X \t %02X",
        Switches, Pneu_View, State_Stuff);
    if (status & (ESCAPE | OBI_ESCAPE | LANDO_ESCAPE)) {
        LukeInit(); }
    pvSend(array, 4);
    return( OK );
}

MsgHandler RX_Motors(msg)
    MsgPtr msg;
{
    register i;
    BytePtr datptr = msg->data;
    for (i= 0; i< 3; i++) {
        Motors[i] = *datptr++;
        onscreen(15, 20+10*i, 0, "%02X", Motors[i]); }
    MotorSigns = *datptr;
    onscreen(15, 55, 0, "%02X", MotorSigns);
    if (Pneu_View & MOTOR_SEE) pvRequest(TXMOTORS);
    return( OK );
}

MsgHandler RX_Pendula(msg)
    MsgPtr msg;
{
    register i;
    BytePtr datptr = msg->data;
    for (i = 0; i < 3; i++) {
        Pendula[i] = (unsigned short)(*datptr++);
        Pendula[i] |= (((unsigned short)(*datptr++))<<8);
        onscreen(13, 20+5*i, 0, "%04X", Pendula[i]); }
    Depth &= 0xFF0;
    Depth |= (unsigned short) ((*datptr) & 0x0F);
    if (Pneu_View & SENSOR_SEE) pvRequest(TXPENDULA);
    return( OK );
}

```

MsgHandler RX_Gyros(msg)

```
MsgPtr msg;
{
    register i;
    BytePtr datptr = msg->data;
    for (i = 0; i < 3; i++) {
        Gyros[i] = (unsigned short)(*datptr++);
        Gyros[i] |= (((unsigned short)(*datptr++))<<8);
        onscreen(12, 20+5*i, 0, "%04X", Gyros[i]); }
    Depth &= 0x00F;
    Depth |= (unsigned short) ((*datptr) << 4);
    onscreen(12, 40, 0, "%02X", Depth);
    if (Pneu_View & SENSOR_SEE) pvRequest(TXGYROS);
    return( OK );
}
```

MsgHandler RX_Hydro01(msg)

```
MsgPtr msg;
{
    register i;
    BytePtr datptr = msg->data;
    thumpID = ((0x07) & (*datptr++));
    for (i = 0; i < 2; i++) {
        range[i][thumpID] = (unsigned short)(*datptr++);
        range[i][thumpID] |= (((unsigned short)(*datptr++))<< 8);
        onscreen(20+i, 20+5*(thumpID), 0, "%04X", range[i][thumpID]); }
    if (Pneu_View & HYDRO_SEE) pvRequest(TXHYDRO01);
    return( OK );
}
```

MsgHandler RX_Hydro23(msg)

```
MsgPtr msg;
{
    register i;
    BytePtr datptr = msg->data;
    thumpID = ((0x07) & (*datptr++));
    for (i = 2; i < 4; i++) {
        range[i][thumpID] = (unsigned short)(*datptr++);
        range[i][thumpID] |= (((unsigned short)(*datptr++))<< 8);
        onscreen(20+i, 20+5*(thumpID), 0, "%04X", range[i][thumpID]); }
    if (Pneu_View & HYDRO_SEE) pvRequest(TXHYDRO23);
    return( OK );
}
```

MsgHandler RX_Position(msg)

```
MsgPtr msg;
{
    register i;
    BytePtr datptr = msg->data;
    for (i = 0; i < 3; i++) {
        pos[i] = (int) (*datptr++);
        pos[i] |= (((int) (*datptr++)) << 8);
        onscreen(17, 20+7*i, 0, "%05d", pos[i]); }
    quat[3] &= 0xFF00; /* Clear low bits */
}
```

```

    quat[3] |= (int) (*dataptr++);
    if (Pneu_View & STATE_SEE) pvRequest(TXPOSITION);
    return( OK );
}

```

```

MsgHandler RX_Attitude(msg)
    MsgPtr msg;
{
    register i;
    BytePtr dataptr = msg->data;
    for (i = 0; i < 3; i++) {
        quat[i] = (int) (*dataptr++);
        quat[i] |= (((int) (*dataptr++)) << 8);
        onscreen(17, 45+7*i, 0, "%05d", quat[i]); }
    quat[3] &= 0x00FF; /* Clear high bits */
    quat[3] |= (((int) (*dataptr++)) << 8);
    onscreen(17, 66, 0, "%05d", quat[3]);
    if (Pneu_View & STATE_SEE) pvRequest(TXATTITUDE);
    return( OK );
}

```

```

MsgHandler RX_Velocity(msg)
    MsgPtr msg;
{
    register i;
    BytePtr dataptr = msg->data;
    for (i = 0; i < 3; i++) {
        veloc[i] = (int) (*dataptr++);
        veloc[i] |= (((int) (*dataptr++)) << 8);
        onscreen(18, 20+7*i, 0, "%05d", veloc[i]); }
    if (Pneu_View & STATE_SEE) pvRequest(TXVELOCITY);
    return( OK );
}

```

```

MsgHandler RX_Omega(msg)
    MsgPtr msg;
{
    register i;
    BytePtr dataptr = msg->data;
    for (i = 0; i < 3; i++) {
        omega[i] = (int) (*dataptr++);
        omega[i] |= (((int) (*dataptr++)) << 8);
        onscreen(18, 45+7*i, 0, "%05d", omega[i]); }
    if (Pneu_View & STATE_SEE) pvRequest(TXOMEGA);
    return( OK );
}

```

```

MsgHandler RX_Bias(msg)
    MsgPtr msg;
{
    register i;
    BytePtr dataptr = msg->data;
    for (i = 0; i < 3; i++) {
        bias[i] = (int) (*dataptr++);

```

```

        bias[i] |= (((int) (*datptr++)) << 8);
        onscreen(19, 45+7*i, 0, "%05d", bias[i]); }
if (Pneu_View & STATE_SEE) pvRequest(TXBIAS);
return ( OK );
}

/ *          lukefuncs.c          * /
/ *   Written by: Ella M. Atkins   * /
/ *   Last modified: 4/11/90      * /

#include "pvluke.h"
#include <gf.h>
#include <asiports.h>
#include <stdio.h>
#include <ibmkeys.h>

void ReadTHC()          / *   Bang-bang hand controller   * /
{
    Byte  thcpos, thcneg;
    thc = 0x00;
    outp(P0C,0x01);      / *   C+ for THC   * /
    thcpos = inp(P0A);
    outp(P0C,0x02);      / *   C- for THC   * /
    thcneg = inp(P0A);

/* For the magnitude: bit0=xmotors on; bit1=ymotors on; bit2=zmotors on */
/* Both hand controller sensors must be activated for ON command   */
/* For the signs (0=+;1=-): bit4=x-direction; bit5=y-dir; bit6=z-dir */

    if (!(thcpos & 0x02)                thc |= 0x01;
        else if (!(thcneg & 0x01) && !(thcneg & 0x02)) thc |= 0x02;
    if ( !(thcpos & 0x04) && !(thcpos & 0x08)  thc |= 0x04;
        else if (!(thcneg & 0x04) && !(thcneg & 0x08)) thc |= 0x08;
    if ( !(thcpos & 0x10) && !(thcpos & 0x20)  thc |= 0x10;
        else if (!(thcneg & 0x10) && !(thcneg & 0x20)) thc |= 0x20;
}

void ReadRHC()          / *   Bang-bang hand controller   * /
{
    Byte  rhcpos, rhcneg;
    rhc = 0x00;
    outp(P0C,0x04);      / *   C+ for RHC   * /
    rhcpos = inp(P0B);
    outp(P0C,0x08);      / *   C- for RHC   * /
    rhcneg = inp(P0B);
    outp(P0C,0x00);

/* For rhc magnitude: bit0=roll on; bit1=pitch on; bit2=yaw on      */
/* Both hand controller sensors must be activated for ON command   */
/* For rhc signs (0=+;1=-): bit4=roll direction;bit5=pitch-dir;bit6=yaw-dir */

    if ( !(rhcpos & 0x01) )                rhc |= 0x01;
        else if (!(rhcneg & 0x01) && !(rhcneg & 0x02)) rhc |= 0x02;
}

```



```

    if ( !(rhcpow & 0x04) && !(rhcneg & 0x08)) rhc |= 0x04;
    else if (!(rhcneg & 0x04) && !(rhcpow & 0x08)) rhc |= 0x08;
    if ( !(rhcpow & 0x10) && !(rhcneg & 0x20)) rhc |= 0x10;
    else if (!(rhcneg & 0x10) && !(rhcpow & 0x20)) rhc |= 0x20;
}

```

```

void ReadSW()
{

```

```

    register i;
    unsigned inkey;
    status = 0x00;
    Pstatus = 0x00;
    Sstatus = 0x00;
    if (gfkbit()) {
        if ((inkey = getkey()) == ESC) STOP = 0x01;

```

```

/* Keys for Switch variables */

```

```

    else if (inkey == '1') status |= ESCAPE;
    else if (inkey == '2') status |= YODA_ESCAPE;
    else if (inkey == '3') status |= OBI_ESCAPE;
    else if (inkey == '4') status |= LANDO_ESCAPE;
    else if (inkey == '9') status |= STATE_CALC;
    else if (inkey == '0') status |= SAVE_DATA;

```

```

/* Keys for Pneu_View variables */

```

```

    else if (inkey == ' ') Pstatus |= POWER;
    else if (inkey == '.') Pstatus |= LATCH;
    else if (inkey == '=') Pstatus |= RAM;
    else if (inkey == '5') Pstatus |= SENSOR_SEE;
    else if (inkey == '6') Pstatus |= HYDRO_SEE;
    else if (inkey == '7') Pstatus |= STATE_SEE;
    else if (inkey == '8') Pstatus |= MOTOR_SEE;

```

```

/* Keys for State_Stuff variables */

```

```

    else if (inkey == F1) Sstatus |= USE_RANGES;
    else if (inkey == F2) Sstatus |= USE_PENDULA;
    else if (inkey == F3) Sstatus |= USE_DEPTH;
    else if (inkey == F4) Sstatus |= USE_RATE;

```

```

/* Path control calculation elements */

```

```

    else if (inkey == F6) {
        path = ATT_HOLD;
        TX_Ctrl(NULL); }
    else if (inkey == F7) {
        path = POS_HOLD;
        TX_Ctrl(NULL); }
    else if (inkey == F8) {
        path = AP_HOLD;
        TX_Ctrl(NULL); }
    else if (inkey == F9) {

```

```

        onscreen(5, 5, 0, "Enter Final Attitude:      ");
        for (i = 0; i < 4; i++) {
            stateID = i;
            onscreen(5, 40, 0, "Q%01d", i);
            scanf("%f", &stateval);
            TX_State(NULL); }
        path = ENTER_ATT;
        TX_Ctrl(NULL); }
    else if (inkey == F10) {
        onscreen(5, 5, 0, "Enter Final Position:      ");
        onscreen(5, 69, 0, " ");
        for (i = 0; i < 3; i++) {
            stateID = i;
            onscreen(5, 40, 0, "X%01d", i);
            scanf("%f", &stateval);
            TX_State(NULL); }
        path = ENTER_POS;
        TX_Ctrl(NULL); }
    else if (inkey == F11) {
        path = DOCK;
        TX_Ctrl(NULL); }

/* PID Control On/Off and gain settings */

    else if (inkey == F5) {
        path = 0; /* Reset path */
        PID ^= CL_CTRL;
        TX_Ctrl(NULL); }
    else if (inkey == 'q') {
        PID ^= PPOS_CTRL;
        TX_Ctrl(NULL); }
    else if (inkey == 'w') {
        PID ^= IPOS_CTRL;
        TX_Ctrl(NULL); }
    else if (inkey == 'e') {
        PID ^= DPOS_CTRL;
        TX_Ctrl(NULL); }
    else if (inkey == 'r') {
        PID ^= PATT_CTRL;
        TX_Ctrl(NULL); }
    else if (inkey == 't') {
        PID ^= IATT_CTRL;
        TX_Ctrl(NULL); }
    else if (inkey == 'y') {
        PID ^= DATT_CTRL;
        TX_Ctrl(NULL); }

/* Enter new Gains * /

    else if (inkey == 'a') {
        gainID = 0x00;
        onscreen(5,5,0,"Enter Proportional Position Gain:");
        scanf("%d", &gainval);
        TX_Gains(NULL);

```

```

        onscreen(14,12,0,"%04X", gainval); }
else if (inkey == 's') {
    gainID = 0x01;
    onscreen(5,5,0,"Enter Integral Position Gain:  ");
    scanf("%d", &gainval);
    TX_Gains(NULL);
    onscreen(14,17,0,"%04X", gainval); }
else if (inkey == 'd') {
    gainID = 0x02;
    onscreen(5,5,0,"Enter Derivative Position Gain: ");
    scanf("%d",&gainval);
    TX_Gains(NULL);
    onscreen(14,22,0,"%04X", gainval); }
else if (inkey == 'f') {
    gainID = 0x04;
    onscreen(5,5,0,"Enter Proportional Attitude Gain:");
    scanf("%d",&gainval);
    TX_Gains(NULL);
    onscreen(14,37,0,"%04X", gainval); }
else if (inkey == 'g') {
    gainID = 0x05;
    onscreen(5,5,0,"Enter Integral Attitude Gain:  ");
    scanf("%d",&gainval);
    TX_Gains(NULL);
    onscreen(14,42,0,"%04X", gainval); }
else if (inkey == 'h') {
    gainID = 0x06;
    onscreen(5,5,0,"Enter Derivative Attitude Gain: ");
    scanf("%d",&gainval);
    TX_Gains(NULL);
    onscreen(14,47,0,"%04X", gainval); }
else if (inkey == 'z') {
    gainID = 0x03;
    onscreen(5,5,0,"Enter Feed-Forward Position Gain:");
    scanf("%d",&gainval);
    TX_Gains(NULL);
    onscreen(14,27,0,"%04X", gainval); }
else if (inkey == 'v') {
    gainID = 0x07;
    onscreen(5,5,0,"Enter Feed-Forward Attitude Gain:");
    scanf("%d",&gainval);
    TX_Gains(NULL);
    onscreen(14,52,0,"%04X", gainval); }
else if (inkey == 'b') {
    gainID = 0x08;
    onscreen(5,5,0,"Enter Feed-Forward C-Coupling Gain");
    scanf("%d",&gainval);
    TX_Gains(NULL);
    onscreen(14,57,0,"%04X", gainval); }
}
Switches ^= status;
Pneu_View ^= Pstatus;
State_Stuff ^= Sstatus;
}

```

Appendix B.5 Crumb and Cake Software

```
/*          USMV6811.TXT          */
/*      Written by:  Matt Machlis  */
/* *      Last Modified:  4/11/90  */
```

HEX

0000 2F00 100

CMOVE

2F00 2F0E +!

2F00 2F10 +!

2F00 2F1C +!

2F00 2F22 +!

2F00 2F2C +!

2F00 2F34 +!

2F00 2F38 +!

2F00 2F40 +!

2F00 2F46 +!

2F00 2F4A +!

2F00 2F6A +!

2F00 2F6C +!

2F06 04 !

ABORT

C030 TIB !

50 TIB 2+ !

2000 DP !

```
/*          INT6811.TXT          */
/*      Written by:  Matt "I have a llama" Machlis  */
/* *      Last Modified:  4/11/90  */
```

HEX

B000 CONSTANT PRTA

B003 CONSTANT PRTC

B004 CONSTANT PRTB

B007 CONSTANT DDRC

B008 CONSTANT PRTD

B009 CONSTANT DDRD

B00A CONSTANT PRTE

B00E CONSTANT TCNT

B010 CONSTANT TIC1

B012 CONSTANT TIC2

B016 CONSTANT TOC1

B021 CONSTANT TCTL2

B022 CONSTANT TMSK1

B023 CONSTANT TFLG1

B024 CONSTANT TMSK2

B025 CONSTANT TFLG2

B026 CONSTANT PACTL

C000 CONSTANT CNT1

C002 CONSTANT CNT2
 C004 CONSTANT NOTDN1
 C006 CONSTANT NOTDN2
 C008 CONSTANT GT+ID*2
 C00A CONSTANT BCNT
 C00C CONSTANT IRQFL
 C00E CONSTANT TIC1FL
 C010 CONSTANT TIC2FL
 C012 CONSTANT TOC1FL
 C014 CONSTANT G<S1
 C016 CONSTANT G<S2
 C018 CONSTANT C>G
 C01A CONSTANT C<S
 C01C CONSTANT GTCNT1
 C01E CONSTANT GTCNT2
 C020 CONSTANT GATES

: IRQHI TCNT @ BCNT ! -1 IRQFL ! ;
 CODE IRQLO CC C, ' IRQHI CFA , BD C, ATO4 , 3B C, END-CODE
 : TIC1HI IRQFL @ IF TMSK1 C@ 82 AND TMSK1 C! TCTL2 C@ 04 AND
 TCTL2 C! -1 TIC1FL ! THEN ;
 CODE TIC1LO CC C, ' TIC1HI CFA , BD C, ATO4 , 3B C, END-CODE
 : TIC2HI IRQFL @ IF TMSK1 C@ 84 AND TMSK1 C! TCTL2 C@ 10 AND
 TCTL2 C! -1 TIC2FL ! THEN ;
 CODE TIC2LO CC C, ' TIC2HI CFA , BD C, ATO4 , 3B C, END-CODE
 : TOC1HI 0 TMSK1 C! 0 TCTL2 C! -1 TOC1FL ! ;
 CODE TOC1LO CC C, ' TOC1HI CFA , BD C, ATO4 , 3B C, END-CODE

: SETIRQVC 7E B7E9 EEC! [' IRQLO @ >< FF AND] LITERAL
 B7EA EEC! [' IRQLO @ FF AND] LITERAL B7EB EEC! ;
 : SETXIRQVC 7E B7EC EEC! [' IRQLO @ >< FF AND] LITERAL
 B7ED EEC! [' IRQLO @ FF AND] LITERAL B7EE EEC! ;
 : SETTOC1VC 7E B7DA EEC! [' TOC1LO @ >< FF AND] LITERAL
 B7DB EEC! [' TOC1LO @ FF AND] LITERAL B7DC EEC! ;
 : SETTIC1VC 7E B7E3 EEC! [' TIC1LO @ >< FF AND] LITERAL
 B7E4 EEC! [' TIC1LO @ FF AND] LITERAL B7E5 EEC! ;
 : SETTIC2VC 7E B7E0 EEC! [' TIC2LO @ >< FF AND] LITERAL
 B7E1 EEC! [' TIC2LO @ FF AND] LITERAL B7E2 EEC! ;
 : SETINTVCS SETIRQVC SETTOC1VC SETTIC1VC
 SETTIC2VC SETXIRQVC ;

: INITL 20 PRD C! 22 DDRD C! 0 DDRC C! 0 PACTL C!
 0 TMSK1 C! 0 TMSK2 C! SETINTVCS 1F 0 DO 0 | GATES
 + C! LOOP 0 IRQFL ! ;

CODE-SUB CLEAR-CC-MASKS 86 C, EF C, 06 C, 39 C, END-CODE
 : PSE FFFF 0 DO LOOP CLEAR-CC-MASKS ;

: RSET 0 TMSK1 C! 0 CNT1 ! 0 CNT2 ! 0 TCTL2 ! FF TFLG1 C!
 -1 NOTDN1 ! -1 NOTDN2 ! 0 IRQFL ! 0 TIC1FL ! 0 TIC2FL !
 0 TOC1FL ! ;

: IRQWAIT FF TFLG1 C! 14 TCTL2 C! BEGIN IRQFL
 @ ?TERMINAL OR UNTIL ?TERMINAL IF ABORT THEN ;

```

: IRQCALC1 BCNT @ 1- TOC1 ! 80 TFLG1 C! 80 TMSK1 C!
PRTD C@ 1C AND 2/ GATES + DUP GT+ID*2 ! C@
100 * BCNT @ + DUP GTCNT1 ! ;
: IRQCALC2 BCNT @ < G<S1 ! GT+ID*2 @ 1+ C@ 100 * BCNT @ + DUP
GTCNT2 ! BCNT @ < G<S2 ! ;
: IRQCALC IRQCALC1 IRQCALC2 ;

: VALID1 TIC1 @ GTCNT1 @ > C>G ! TIC1 @ BCNT @ < C<S !
C>G @ C<S @ OR G<S1 @ 0= AND G<S1 @ C>G @ AND C<S @ AND OR ;
: VALID2 TIC2 @ GTCNT2 @ > C>G ! TIC2 @ BCNT @ < C<S !
C>G @ C<S @ OR G<S2 @ 0= AND G<S2 @ C>G @ AND C<S @ AND OR ;
: IS1DN TFLG1 C@ 04 AND IF VALID1 IF TIC1 @ BCNT @ - CNT1 !
0 NOTDN1 ! TCTL2 C@ 04 AND TCTL2 C! ELSE 04 TFLG1 C! THEN THEN ;
: IS2DN TFLG1 C@ 02 AND IF VALID2 IF TIC2 @ BCNT @ - CNT2 !
0 NOTDN2 ! TCTL2 C@ 10 AND TCTL2 C! ELSE 02 TFLG1 C! THEN THEN ;
: OVORDUN NOTDN1 @ 0= NOTDN2 @ 0= AND TOC1FL @ OR ;
: CNTWAIT BEGIN NOTDN1 IF IS1DN THEN NOTDN2 IF IS2DN THEN
OVORDUN UNTIL ;

: A7HI PRTA C@ 80 AND 0> ;
: A7LO PRTA C@ 80 AND 0= ;
: D5A6LO 0 PRTD C! 0 PRTA C! ;
: D5A6HI 20 PRTD C! 40 PRTA C! ;
: NOIRQ 0 TCTL2 C! 0 TMSK1 C! 0 TMSK2 C! ;
: OUT FF DDRC C! CNT1 C@ PRTB C! CNT1 1+ C@ PRTC C! D5A6LO
BEGIN A7HI UNTIL CNT2 C@ PRTB C! CNT2 1+ C@ PRTC C! D5A6HI
BEGIN A7LO UNTIL ;
: IN 0 DDRC C! 0 PRTA C! BEGIN A7HI UNTIL PRTE C@ GT+ID*2 @
C! PRTC C@ GT+ID*2 @ 1+ C! FF PRTA C! ;
: DOIO NOIRQ OUT IN ;

: DOIT INITL PSE BEGIN RSET IRQWAIT IRQCALC CNTWAIT DOIO
0 UNTIL ;

: STRT HEX
C030 TIB !
50 TIB 2+ !
2F00 CF00 100 CMOVE
A000 CF0E +!
A000 CF10 +!
A000 CF1C +!
A000 CF22 +!
A000 CF2C +!
A000 CF34 +!
A000 CF38 +!
A000 CF40 +!
A000 CF46 +!
A000 CF4A +!
A000 CF6A +!
A000 CF6C +!
CF06 04 !
DOIT
ABORT ;
3000 AUTOSTART STRT

```

Appendix C.0 Parameter and Calibration Calculations

This Appendix contains the MPOD physical parameter and sensor calibration calculations. The charts were developed in Macintosh Microsoft Excel v2.2, while the range calibration plots were produced in Macintosh Cricket Graph v1.3. The following analyses are included:

<u>Appendix</u>	<u>Description</u>	<u>Page</u>
Appendix C.1	MPOD Moments of Inertia	184
Appendix C.2	MPOD Sensor Averages and Standard Deviations	185
Appendix C.3	3DAPS Range Calibration Plots	186
Appendix C.4	Measured vs. Estimated Range Calculations	187

Appendix C.1 MPOD Moments of Inertia

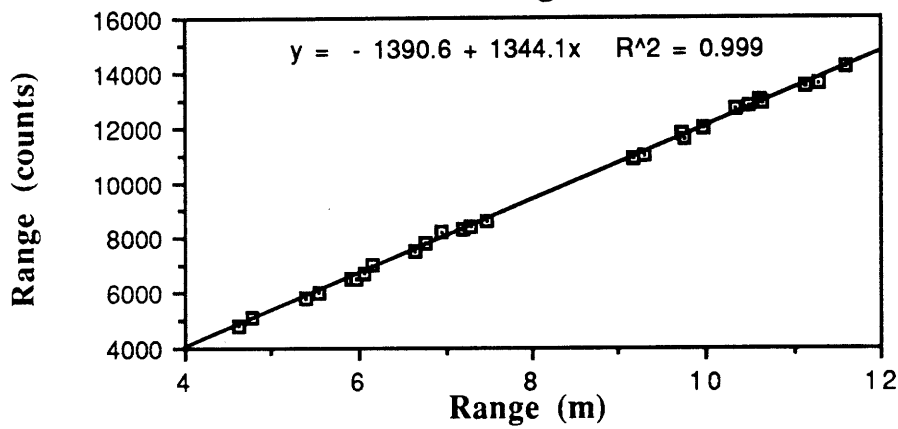
	mass (kg)	Dimensions (m)			Distance from origin (m)			Central Moment of Inertia			Parallel Axis Moment of Inertia			Total Moment of Inertia (kg-m-m)			
		x	y	z	x	y	z	Ixx	Iyy	Izz	Ixx	Iyy	Izz	Ixx	Iyy	Izz	
x-motor pairs (2)	8.18	0.254	0.0635	0.0635	0.85	0	0.85	0.0165	0.0261	0.0261	5.9101	0	5.9101	5.9266	0.0261	5.9362	
y-motor pairs (2)	8.18	0.0635	0.254	0.0635	0.85	0.85	0	0.0261	0.0165	0.0261	5.9101	5.9101	0	5.9362	5.9266	0.0261	
z-motor pairs (2)	8.18	0.0635	0.0635	0.254	0	0.85	0.85	0.0261	0.0261	0.0165	0	5.9101	5.9101	0.0261	5.9362	5.9266	
x-motor panels (2)	6.66	0.508	0.0048	0.508	0.762	0	0.762	0.1432	0.2864	0.1432	3.8671	0	3.8671	4.0103	0.2864	4.0103	
y-motor panels (2)	6.66	0.508	0.508	0.0048	0.762	0.762	0	0.1432	0.1432	0.2864	3.8671	3.8671	0	4.0103	4.0103	0.2864	
z-motor panels (2)	6.66	0.0048	0.508	0.508	0	0.762	0.762	0.2864	0.1432	0.1432	0	3.8671	3.8671	0.2864	4.0103	4.0103	
inner docking probe	4.4	0.356	0.07 (diam)		0	0.94	0.94	0.0108	0.0518	0.0518	0	3.8878	3.8878	0.0108	3.9396	3.9396	
outer docking probe	4	0.356	0.0508 (diam)		0	1.295	1.295	0.0052	0.0448	0.0448	0	6.7081	6.7081	0.0052	6.7529	6.7529	
arm weights (2)	9.1	0.15	0.08	0.025	0.305	0	0.305	0.0053	0.0175	0.0219	0.8465	0	0.8465	0.8518	0.0175	0.8684	
main battery box (2)	94.54	0.33	0.203	0.559	0.3556	0	0.3556	2.7864	3.3196	1.1826	11.9547	0	11.9547	14.7411	3.3196	13.1373	
control battery box	68	0.216	0.584	0.203	0	0.356	0.356	2.1661	0.4979	2.1969	0	8.618	8.618	2.1661	9.1159	10.8149	
control box	80	0.41	0.56	0.32	0.356	0.356	0	2.7732	1.8033	3.2112	10.1389	10.1389	0	12.9121	11.9422	3.2112	
large side air tanks (2)	35.4	0.1778	(diam)		0.6858	0.356	0	0.356	1.6667	0.8333	1.5735	4.4865	0	4.4865	6.1532	0.8333	6.06
small rear air tank	22.8	0.18	0.467	0.18	0	0.51	0.51	0.5813	1.026	0.2907	0	5.9303	5.9303	0.5813	6.9563	6.221	
leftover frame	137.24	0.04m thick	5m radius		0	0	0	22.8733333	22.8733333	22.8733333	0	0	0	22.8733333	22.8733333	22.8733333	
Totals:	500													80.4908333	85.9465333	94.0745333	

Appendix C.2 Sensor Averages and Standard Deviations

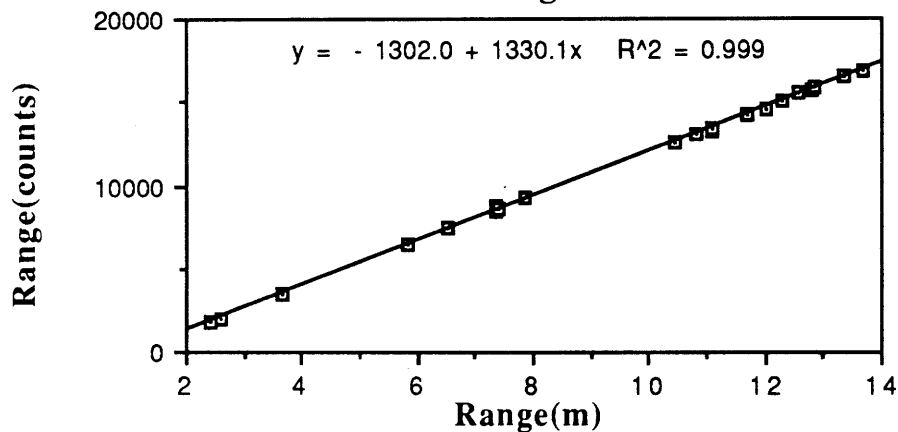
Static Data	Average:	(raw readings)			Std. Deviation (raw readings)				
	Roll Rate	Pitch Rate	Yaw Rate	Depth	Ranges	Roll Rate	Pitch Rate	Yaw Rate	Depth
Location 1	1959.8	2047.2	2199.2	2704.1	24.76	2.08	1.95	1.29	1.96
Location 2	1939.9	2047.6	2197.5	2705.1	55.6	1.86	1.79	1.07	1.87
Location 3	1899.7	2048.9	2193.6	2693.3	47.76	2.082	1.56	2.02	1.03
Overall Avg.	1933.13333	2047.9	2196.76667		42.7066667	2.00733333	1.76666667	1.46	1.62
Rate Drift	30.6160633	0.88881944	2.87112057						

Appendix C.3 3DAPS Range Calibration Plots

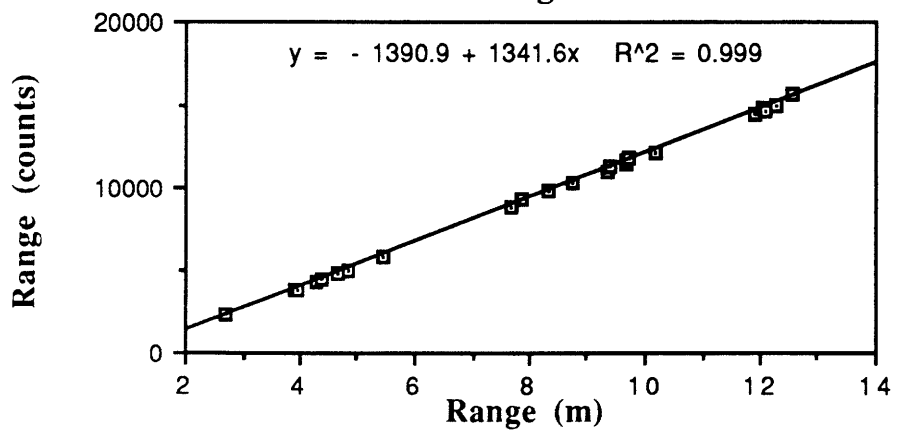
Location 1 Range Calibration



Location 2 Range Calibration



Location 3 Range Calibration



Appendix C.4 Measured vs. Estimated Range Calculations

Location 1:			Location 2:			Location 3:		
Estimate Range	Measured Range	Rest - Rmeas	Estimate Range	Measured Range	Rest - Rmeas	Estimate Range	Measured Range	Rest - Rmeas
6.083107	5.9	0.183107	10.811865	11.08	0.268135	4.479916	4.3	0.179916
5.84455	5.385	0.45955	10.573755	10.805	0.231245	3.655891	4.83	1.174109
7.580872	7.19	0.390872	12.293838	12.58	0.286162	3.73435	3.925	0.19065
7.669305	7.475	0.194305	11.720015	12.015	0.294985	5.644891	5.43	0.214891
5.797298	5.96	0.162702	10.619742	11.11	0.490258	4.380056	4.35	0.030056
4.8757	4.625	0.2507	10.025686	10.445	0.419314	3.222664	3.935	0.712336
6.842858	6.645	0.197858	11.768183	12.31	0.541817	2.594725	2.705	0.110275
6.775283	6.95	0.174717	11.124805	11.69	0.565195	4.39411	4.65	0.25589
6.485527	6.065	0.420527	12.889354	12.79	0.099354	9.312071	9.355	0.042929
8.37589	7.75	0.62589	13.814163	13.66	0.154163	8.084939	8.34	0.255061
7.807464	7.305	0.502464	14.471122	14.33	0.141122	10.092496	10.175	0.082504
6.327496	5.525	0.802496	12.808796	12.585	0.223796	10.185314	10.97	0.784686
6.218241	6.15	0.068241	12.728626	12.83	0.101374	9.264444	9.395	0.130556
7.731055	7.28	0.451055	13.399298	13.37	0.029298	7.898497	7.86	0.038497
7.093066	6.765	0.328066	14.027296	14.05	0.022704	9.728603	9.69	0.038603
5.208016	4.77	0.438016	12.266531	12.285	0.018469	9.549032	10.65	1.100968
10.605918	10.485	0.120918	3.990676	3.565	0.425676	9.010324	8.71	0.300324
9.930628	10.15	0.219372	4.389296	3.695	0.694296	10.986339	10.125	0.861339
9.623069	9.59	0.033069	4.252595	3.82	0.432595	10.428185	10.055	0.373185
11.620777	11.49	0.130777	5.617347	5.255	0.362347	9.281152	8.145	1.136152
10.444614	10.525	0.080386	3.436275	3.645	0.208725	8.961093	8.76	0.201093
9.393139	9.765	0.371861	2.824183	2.4	0.424183	10.84987	9.74	1.10987
9.053096	9.17	0.116904	2.331574	2.58	0.248426	10.076421	7.655	2.421421
11.051185	11.135	0.083815	4.237584	4.43	0.192416	8.578064	9.665	1.086936
10.841742	10.605	0.236742	8.072454	7.365	0.707454	12.158576	12.04	0.118576
11.602333	11.61	0.007667	9.91427	9.14	0.77427	13.141551	12.265	0.876551
9.802568	9.715	0.087568	8.73839	7.87	0.86839	14.023557	13.75	0.273557
10.782459	10.35	0.432459	7.632896	6.505	1.127896	12.570425	12.57	0.000425
10.683999	10.655	0.028999	7.813258	7.4	0.413258	12.122138	12.09	0.032138
11.145754	11.28	0.134246	9.327531	8.82	0.507531	13.027677	11.925	1.102677
9.243669	9.305	0.061331	7.981926	7.355	0.626926	13.763988	13.45	0.313988
10.166005	9.965	0.201005	6.683085	5.85	0.833085	12.060638	12.27	0.209362
	Loc. 1 Error:	0.24992766		Loc. 2 Error:	0.39796453		Loc. 3 Error:	0.492485
							Overall Average Error:	0.38012573

Appendix D.0 Simulation Software

Appendix D.1. Obiwan Dynamic Simulation Software

```
/*          simula.h          */
/* Written by: Ella M. Atkins */
/* Last modified: 3/13/90     */

#ifndef SIMULATE
#define SIMULATE

double    x[3], q[4], v[3], w[3];
double    xnew[3], qnew[4], vnew[3], wnew[3];
double    xprime[4][3], qprime[4][4], vprime[4][3], wprime[4][3];
double    Lthrust[3], Rthrust[3];

#define dt 0.06 /* Integration time step */
#define hh dt*0.5
#define h6 dt/6.0

/* 3DAPS configuration information (from KGK thesis) */

#define T0 {-6.300, 4.935, 0.15} /* Thumper coordinates */
#define T1 {-6.300, 4.935, 3.09}
#define T2 {-6.300, -4.935, 0.15}
#define T3 {-6.300, -4.935, 3.09}
#define T4 {6.300, 4.935, 0.15}
#define T5 {6.300, 4.935, 3.09}
#define T6 {6.300, -4.935, 0.15}
#define T7 {6.300, -4.935, 3.09}

#define L0 {0.0, 0.0, -1.43} /* Hydrophone coordinates */
#define L1 {0.0, -1.43, 0.0}
#define L2 {-1.44, 0.0, 0.0}
#define L3 {0.0, 1.434, 0.0}

/* MPOD physical parameter data (From underwater dynamic tests) */

#define CDTX 480.0 /* Estimated from Cd(trans) = (Fmax/((Vt)**2)) */
#define CDTY 480.0 /* N/((m/sec)**2) */
#define CDTZ 480.0

#define CDRX 186.3 /* Estimated from Cd(rot) = (Tmax/((wmax)**2)) */
#define CDRY 265.3 /* N*m / ((rad/sec)**2) */
#define CDRZ 265.3

#define MT 1000.0

#define IXX 80.5 /* Estimated from lumped box and */
#define IYY 85.9 /* thin spherical shell model */
```

```

#define    IZZ          94.1

/* Derived constants */

#define    INVMT        1./MT
#define    I2MI3        (IYY - IZZ)
#define    I1MI2        (IXX - IYY)
#define    I3MI1        (IZZ - IXX)
#define    INVIX        1./IXX
#define    INVIY        1./IYY
#define    INVIZ        1./IZZ

/* Sensor Data */

#define    PEND_NOISE   30 /* Noise in counts (30) */
#define    RATE_NOISE   10 /* Noise in A/D counts (10) */
#define    DEPTH_NOISE   6 /* Noise in A/D counts (6) */
#define    RANGE_NOISE  80 /* Noise in counts (80) */

#define    ROLLBIAS     -0.26

#define    RANGE_OFFSET 1.011 /* distance before counter begins */
#define    RANGE_FACTOR 0.000748 /* Ranges -- counts to meters */
#define    PEND_FACTOR   0.001534 /* Pendula -- counts to radians */
#define    RATE_FACTOR   0.001534 /* Gyros -- 12-bit count to rad/sec*/
#define    DEPTH_FACTOR  0.01662 /* Depth Sensor -- 12-bit count to m*/
#define    DEPTH_ZERO    0x97E /* Depth Sensor -- Surface reading */
#define    THRUST_FACTOR 0.9449 /* Converting #'s to Newtons */
#define    TORQUE_FACTOR 0.8031 /* Converting #'s to N-m's */

/* Simulator-specific routines */

void RungeKutta(), derivs();

#endif

/*      simula.c      */
/* Written by: Ella M. Atkins */
/* Last Modified: 3/22/90 */

#include "obisim.h"
#include "simula.h"
#include <math.h>
#include <stdlib.h>

static double time = 0.00;
static double t[8][3] = {T0, T1, T2, T3, T4, T5, T6, T7};
static double l[4][3] = {L0, L1, L2, L3};

static double c[3][3]; /* Direction cosine matrix */

void simulate() {

```

```

register i, j;
double denom, r[3], Linear[3];
static unsigned char count1 = 0x00, count2 = 0x00;
unsigned char temp;
unsigned short;
double *Cib;

/* Calculate direction cosine matrix */

Cib = c[0];

*Cib++ = 1.0 - 2.0*(q[2]*q[2] + q[3]*q[3]);
*Cib++ = 2.0*(q[1]*q[2] - q[3]*q[0]);
*Cib++ = 2.0*(q[1]*q[3] + q[2]*q[0]);

*Cib++ = 2.0*(q[1]*q[2] + q[3]*q[0]);
*Cib++ = 1.0 - 2.0*(q[1]*q[1] + q[3]*q[3]);
*Cib++ = 2.0*(q[2]*q[3] - q[1]*q[0]);

*Cib++ = 2.0*(q[1]*q[3] - q[2]*q[0]);
*Cib++ = 2.0*(q[2]*q[3] + q[1]*q[0]);
*Cib  = 1.0 - 2.0*(q[1]*q[1] + q[2]*q[2]);

/* Convert State Vector to Sensor Measurements */

if (c[2][2] != 0.00) { /* Conditional to prevent divide by zero */

denom = sqrt(c[2][1]*c[2][1] + c[2][2]*c[2][2]);
*Pendula = 2048 + (unsigned short) (((acos(c[2][2] / denom))/PEND_FACTOR)
+ PEND_NOISE*(((double) rand()) - 16383.0) / 32767.0));

denom = sqrt(c[2][0]*c[2][0] + c[2][2]*c[2][2]);
*(Pendula+1) = 2047 - (unsigned short) (((acos(c[2][2] / denom))
/PEND_FACTOR) + PEND_NOISE*(((double) rand()) - 16383.0) / 32767.0));

} else {
*Pendula = 2048;
*(Pendula+1) = 2048; }

if ((c[2][0] != 0) || (c[2][1] != 0)) {
denom = sqrt(c[2][0]*c[2][0] + c[2][1]*c[2][1]);
*(Pendula+2) = 2047 - (unsigned short) (((acos(c[2][0] / denom))
/PEND_FACTOR) + PEND_NOISE*(((double) rand()) - 16383.0) / 32767.0));
} else *(Pendula+2) = 2048;

*Gyros = (unsigned short) (((w[0] + ROLLBIAS)/RATE_FACTOR)+2048.0
+ RATE_NOISE*(((double) rand()) - 16383.0) / 32767.0));
*(Gyros+1) = (unsigned short) ((-1.0 * (w[1]/RATE_FACTOR))+2048.0
+ RATE_NOISE*(((double) rand()) - 16383.0) / 32767.0));
*(Gyros+2) = (unsigned short) ((-1.0 * (w[2]/RATE_FACTOR))+2048.0
+ RATE_NOISE*(((double) rand()) - 16383.0) / 32767.0));

*Depth = (unsigned short) ((x[2]/DEPTH_FACTOR)+DEPTH_ZERO
+ DEPTH_NOISE*(((double) rand()) - 16383.0) / 32767.0));

```

```

/* Calculate and send a set of range measurements to Lando every 8 loops */

if (count1 >= 0x08) {
count1 = 0x00;
*thumpID = count2;
temp = count2 << 2;

if (*Switches & SAVE_DATA) fprintf(rangefile, "%02u \t", count2);
for (i = 0; i < 4; i++) {
for (j = 0; j < 3; j++) {
r[j] = x[j] + c[j][0]*l[i][0] + c[j][1]*l[i][1] + c[j][2]*l[i][2]
- t[count2][j];
}
*(range+temp+i) = (unsigned short) (((sqrt(r[0]*r[0]
+ r[1]*r[1] + r[2]*r[2])-RANGE_OFFSET)/RANGE_FACTOR)
+ RANGE_NOISE*(((double) rand()) - 16383.0) / 32767.0));
if (*Switches & SAVE_DATA)
fprintf(rangefile, "\t %05d", *(range+temp+i));
if (*Pneu_View & HYDRO_SEE) *(arange+temp+i) = *(range+temp+i);
}
if (*Switches & SAVE_DATA) fprintf(rangefile, "\n");
*INT_DUAL = 0x00; /* Interrupt Lando with new ranges */
if (count2 <= 0x06) count2++;
else count2 = 0x00;
} else count1++;

/* Calculate new state vector */

/* Linear[i] = Thrusts in N along the defined positive axes */

Linear[0] = (((double) Trans[0]) - 128.0)*THRUST_FACTOR;
Linear[1] = (128.0 - ((double) Trans[1]))*THRUST_FACTOR;
Linear[2] = (128.0 - ((double) Trans[2]))*THRUST_FACTOR;
for (i = 0; i <= 2; i++) {
Lthrust[i] = c[i][0]*Linear[0] + c[i][1]*Linear[1] + c[i][2]*Linear[2];
}
Rthrust[0] = (((double) Torque[0]) - 128.0)*TORQUE_FACTOR;
Rthrust[1] = (128.0 - ((double) Torque[1]))*TORQUE_FACTOR;
Rthrust[2] = (128.0 - ((double) Torque[2]))*TORQUE_FACTOR;

RungeKutta();
time += dt; /* Increment simulated time value */

/* Print state vectors */

for (i = 0; i < 3; i++) {
onscreen((3+i), 20, 0, "%8.5f", x[i]);
onscreen((8+i), 20, 0, "%8.5f", q[i]);
onscreen((14+i), 20, 0, "%8.5f", v[i]);
onscreen((19+i), 20, 0, "%8.5f", w[i]);
onscreen((3+i), 30, 0, "%8.5f", *(state+i));
onscreen((8+i), 30, 0, "%8.5f", *(state+3+i));
onscreen((14+i), 30, 0, "%8.5f", *(state+7+i));
}

```

```

    onscreen((19+i), 30, 0, "%8.5f", *(state+10+i));
    onscreen((22+i), 30, 0, "%8.5f", *(state+13+i));
}
onscreen(1, 20, 0, "%9.4f", time);
onscreen(11, 20, 0, "%8.5f", q[3]);
onscreen(11, 30, 0, "%8.5f", *(state+6));
}

void RungeKutta() {
    register i;
    short j;
    double temp;

    for (i = 0; i <=3; i++) { /* Initialize 'new' values */
        if (i < 3) {
            xnew[i] = x[i];
            vnew[i] = v[i];
            wnew[i] = w[i]; }
        qnew[i] = q[i];
    }
    for (j = 0; j < 2; j++) {
        derivs(j);
        for (i = 0; i <= 3; i++) {
            if (i < 3) {
                xnew[i] = x[i] + hh*xprime[j][i];
                vnew[i] = v[i] + hh*vprime[j][i];
                wnew[i] = w[i] + hh*wprime[j][i];
            }
            qnew[i] = q[i] + hh*qprime[j][i];
        }
    }
    derivs(2);
    for (i = 0; i <= 3; i++) {
        if (i < 3) {
            xnew[i] = x[i] + dt*xprime[2][i];
            vnew[i] = v[i] + dt*vprime[2][i];
            wnew[i] = w[i] + dt*wprime[2][i];
        }
        qnew[i] = q[i] + dt*qprime[2][i];
    }
    derivs(3);
    for (i = 0; i <=3; i++) {
        if (i < 3) {
            x[i] += h6*(xprime[0][i]+2*(xprime[1][i]+xprime[2][i])+xprime[3][i]);
            v[i] += h6*(vprime[0][i]+2*(vprime[1][i]+vprime[2][i])+vprime[3][i]);
            w[i] += h6*(wprime[0][i]+2*(wprime[1][i]+wprime[2][i])+wprime[3][i]);
        }
        q[i] += h6*(qprime[0][i]+2*(qprime[1][i]+qprime[2][i])+qprime[3][i]);
    }
    temp = q[1]*q[1] + q[2]*q[2] + q[3]*q[3];
    if (temp > 1.0) {
        temp = sqrt(temp);
        q[3] = 0.0;
        for (i = 0; i < 3; i++)

```



```

    q[i+1] /= temp;
  } else
    q[0] = sqrt(1.0-temp);
}

void derivs(k)
  short k;
{
  register i;

  for (i = 0; i < 3; i++) xprime[k][i] = v[i];

  qprime[k][0] = (-0.5)*(q[1]*w[0] + q[2]*w[1] + q[3]*w[2]);
  qprime[k][1] = 0.5*(q[0]*w[0] + q[2]*w[2] - q[3]*w[1]);
  qprime[k][2] = 0.5*(q[0]*w[1] + q[3]*w[0] - q[1]*w[2]);
  qprime[k][3] = 0.5*(q[0]*w[2] + q[1]*w[1] - q[2]*w[0]);

  vprime[k][0] = INVMT*(Lthrust[0] - CDTX*v[0]*fabs(v[0]));
  vprime[k][1] = INVMT*(Lthrust[1] - CDTY*v[1]*fabs(v[1]));
  vprime[k][2] = INVMT*(Lthrust[2] - CDTZ*v[2]*fabs(v[2]));

  wprime[k][0] = INVIX*(Rthrust[0] + w[1]*w[2]*I2MI3 - CDRX*w[0]*fabs(w[0]));
  wprime[k][1] = INVYI*(Rthrust[1] + w[0]*w[2]*I3MI1 - CDRY*w[1]*fabs(w[1]));
  wprime[k][2] = INVIZ*(Rthrust[2] + w[0]*w[1]*I1MI2 - CDRZ*w[2]*fabs(w[2]));
}

void SimInit() {
  register i;
  float array[4];
  for (i = 0; i < 3; i++) {
    x[i] = xnew[i] = 0.00;
    v[i] = vnew[i] = 0.00;
    w[i] = wnew[i] = 0.00;
    q[i+1] = qnew[i+1] = 0.00;
  }
  q[0] = 1.0; /* For aligned attitude */
  x[2] = 2.0; /* Center of Pool */
  printf("Enter X:"); /* For static tests */
  for (i = 0; i < 3; i++) {
    scanf("%f", &array[i]);
    x[i] = (double) array[i];
  }
  printf("Enter q:");
  for (i = 0; i < 4; i++) {
    scanf("%f", &array[i]);
    q[i] = (double) array[i];
  }
}

```

Appendix D.2 Obiwan Simulation with Pool Test Data

```
/*          obisim2.c          */
/* Written by: Ella M. Atkins */
/* Last modified: 4/11/90    */

#include "obisim2.h"
#include <dos.h>
#include <gf.h>
#include <asiports.h>
#include <math.h>
#include <ibmkeys.h>

main() { /* Begin Obi-Wan main driver program */

    register i, j;
    static unsigned char count1 = 0, count2 = 0, count4 = 0, oldthump;
    static short count3 = 0;
    FILE *readsensors, *readranges, *statefile;
    unsigned short temp;

    double far *ydex;
    int far *aydex;
    unsigned short far *fdatptr;

    unsigned short *datptr, data[4];
    char *motptr, motodata[6], mototemp;
    static unsigned char *databyte;

    if ((readranges = fopen("range.dat","r")) == NULL)
        exit(0);
    if ((readsensors = fopen("sensor.dat","r")) == NULL)
        exit(0);
    if ((statefile = fopen("state.dat","a")) == NULL)
        exit(0);

/* Enable and initialize IRQ9 and IRQ10 */

    outp(0x21, 0xB8);
    outp(0xA1, 0xD8);
    _dos_setvect(0x71, DapsInt); /* Hardware IRQ9 */
    _dos_setvect(0x72, Busy); /* Hardware IRQ10 */
    outp(0xA0,0x20);
    outp(0x20,0x20);

/* Initialize variables and Dual Port Ram */

    Dual_Init_1();
    Dual_Init_2();
    ObiInit();
```

```

/* Begin program loop */

while (!( (*status) & OBI_ESCAPE)) {
if (kbhit()) if (getkey() == ESC) break;

/* Save state data if requested */

if (( *Switches) & SAVE_DATA) {
if (( *status) & SAVE_DATA) {
fprintf(statefile, "State Data Saving...");
*status &= (~SAVE_DATA); }
if (count1 >= 75) {
ydex = state;
for (i = 0; i < 16; i++)
fprintf(statefile, " %7.4f ", (*ydex++));
fprintf(statefile, "\n");
count1 = 0;
} else count1++;
}
/* Read Pool Test MPOD Data */

if ( *Switches & STATE_CALC) {
if (count2 >= 75) {
if (fscanf(readsensors, "%05u \t %05u \t %05u \t %05u \t",
&data[0], &data[1], &data[2], &data[3]) == EOF) {
printf("Sensor exit");
exit(0); }
datptr = data;
fdatptr = Gyros;
for (i = 0; i < 3; i++)
*fdatptr++ = *datptr++;
*Depth = *datptr;

if (fscanf(readsensors, "%05u \t %05u \t %05u \t", &data[0],
&data[1], &data[2]) == EOF) exit(0);
datptr = data;
fdatptr = Pendula;
for (i = 0; i < 3; i++)
*fdatptr++ = *datptr++;

if (fscanf(readsensors,
"%03d \t %03d \t %03d \t %03d \t %03d \t %03d \n",
&motodata[0], &motodata[1], &motodata[2],
&motodata[3], &motodata[4], &motodata[5]) == EOF)
{ printf("Sensor exit");
exit(0); }
for (i = 0; i < 3; i++) {
mototemp = motodata[i+i]+motodata[i+i+1];
if (mototemp != 0x00)
Trans[i] = -(((mototemp/2)<<4) + (mototemp/2));
else Trans[i] = 0x80;
mototemp = motodata[i+i] - motodata[i+i+1];
if (i == 0) j = 2;
else j = i-1;
}
}
}

```

```

if (mototemp != 0x00) {
Torque[j] = (((mototemp/2)<<4) + (mototemp/2));
} else Torque[j] = 0x80;
}
count2 = 0;
} else count2++;

if (count3 >= 450) {
if (fscanf(readranges, "%02u \n", databyte) == EOF) {
printf("Range exit");
exit(0); }
*thumpID = *databyte;
temp = (*databyte) << 2;
if (fscanf(readranges, "%05u \t %05u \t %05u \t %05u \n",
&data[0], &data[2], &data[1], &data[3]) == EOF) {
printf("Range exit");
exit(0); }
datptr = data;
fdatptr = (range+temp);
for (i = 0; i < 4; i++)
*fdatptr++ = *datptr++;
*INT_DUAL = 0x00; /* Tell Lando about new range data */
if (*Pneu_View & HYDRO_SEE) {
for (i = 0; i < 4; i++)
*(arange+temp+i) = *(range+temp+i);
}
if (oldthump > (*databyte)) count3 = -900; /* New set */
else count3 = count3-(450*((*databyte) - oldthump));
oldthump = *databyte;
printf("%02u \n", *thumpID); /* For test of thumper 'speed' */
} else count3++;
}

/* Dual Port Ram transfers between Yoda and Lando */

if (*status & LANDO_ESCAPE) {
*bstatus |= LANDO_ESCAPE;
*status &= (~LANDO_ESCAPE);
} else if (*status & STATE_CALC) {
*bstatus |= STATE_CALC;
*status &= (~STATE_CALC); }
*bSwitches = *Switches;
if (*Switches & STATE_CALC) {
*bState_Stuff = *State_Stuff;
for (i = 0; i < 3; i++) {
*(bPendula+i) = *(Pendula+i);
*(bGyros+i) = *(Gyros+i);
*(actuator+i) = Trans[i];
*(actuator+3+i) = Torque[i];
}
*bDepth = *Depth;
}
if (*Pneu_View & STATE_SEE) { /* Convert to 2 Byte messages */
ydex = state;

```

```

    aydex = astate;
    for (i = 0; i < 3; i++)
        *aydex++ = (int) (100.0*(*ydex++));
    for (i = 0; i < 4; i++)
        *aydex++ = (int) (1000.0*(*ydex++));
    for (i = 0; i < 3; i++)
        *aydex++ = (int) (100.0*(*ydex++));
    for (i = 0; i < 6; i++)
        *aydex++ = (int) (57.296*(*ydex++));
}
} /* End main driver loop */

if (*status & OBI_ESCAPE) {
    *status ^= OBI_ESCAPE;
    *Switches &= (~OBI_ESCAPE); }
outp(0x21, 0xBC); /* Disable Hardware IRQ9 & IRQ10 */
outp(0xA1, 0xDE);
fclose(readranges);
fclose(readsensors);
fclose(statefile);
if (*status & ESCAPE) {
    *bSwitches = *Switches;
    *bstatus = *status; }
exit(0);
}

void ObiInit() {

    register i, j;
    unsigned char temp;

    /* Initialize the 8255's on MIC */

    outpw(P0X, 0x9292); /* Initialized with Port A, B input, */
    outpw(P1X, 0x9292); /* Port C output */

    /* Handshake with 68HC11's in case thumper has already activated HC11 */

    outpw(P0C, 0x0100);
    outpw(P1C, 0x0100);
    for (i = 0; i < 0x8888; i++); { }
    outpw(P0C, 0x0000);
    outpw(P1C, 0x0000);
    for (i = 0; i < 8888; i++) { } /* Delay to give HC11 proc. time */
    outpw(P0X, 0x8282);
    outpw(P1X, 0x8282);
    outpw(P0A, 0x0000);
    outpw(P1A, 0x0000);
    outpw(P0C, 0x0100);
    outpw(P1C, 0x0100);
    for (i = 0; i < 8888; i++) { }
    outpw(P0X, 0x9292);
    outpw(P1X, 0x9292);
    outpw(P0C, 0x0000);

```

```

    outpw(P1C, 0x0000);

/* Initialize variables to inert values */

    for (i = 0; i < 3; i++) {
        Trans[i] = 0x80;
        Torque[i] = 0x80; }

    for (j = 0; j < 8; j++) { /* Initialize DPR ranges and gates */
        temp = j<<2;
        for (i = 0; i < 4; i++) {
            *(gate+i+temp) = 0x00;
            *(range+i+temp) = 0x00;
            *(arange+i+temp) = 0x00;
        }
    }

    for (j = 0; j < 16; j++) { /* Initialize DPR states */
        *(astate+j) = 0x0000;
        *(state+j) = 0.00;
    }
}

```

Appendix E.0 Control System Parameter Calculations

This Appendix contains the control system analyses and simulations. All the calculations and plots were done with Macintosh MATLAB. The following analyses were performed:

<u>Appendix</u>	<u>Description</u>	<u>Page</u>
Appendix E.1	Position Hold Gain Analysis	200
Appendix E.2	Attitude Hold Gain Analysis	203

Appendix E -- Control System Gain Calculations

```
% Position Hold Analysis
%Lqr with q = I, r = 0.0001

minv = 0.001;
a = [0 1 0; 0 0 1; 0 0 0];
b = [0; 0; minv];
q = eye(3);
r = 0.0001;
[g,s] = lqr(a,b,q,r);

% Note that for the given q and r,
% gi = 100, gp = 447.5, gd = 951.33

%Time responses to offset, constant disturbance

t = [0.0: 0.1: 20.0];
x0 = [0.0;0.3;0];
u = ones(201,1);
x1 = a-(b*g);
x2 = [0;0;0];
x3 = [0 1 0; 0 0 1; -g];
x4 = zeros(3,1);

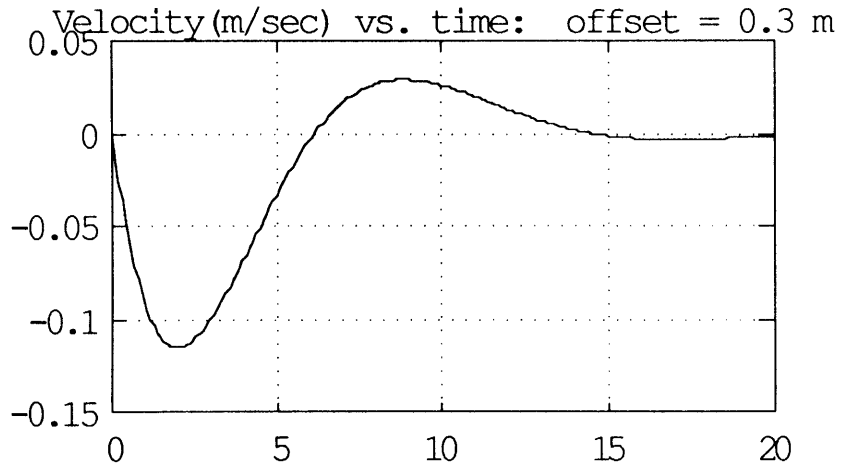
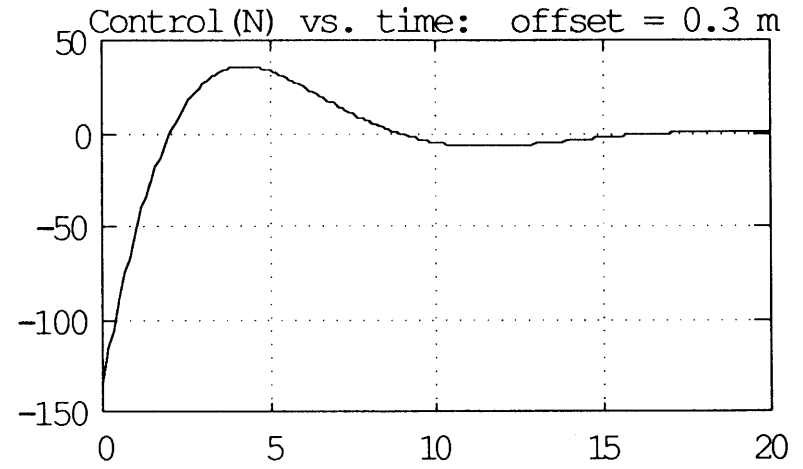
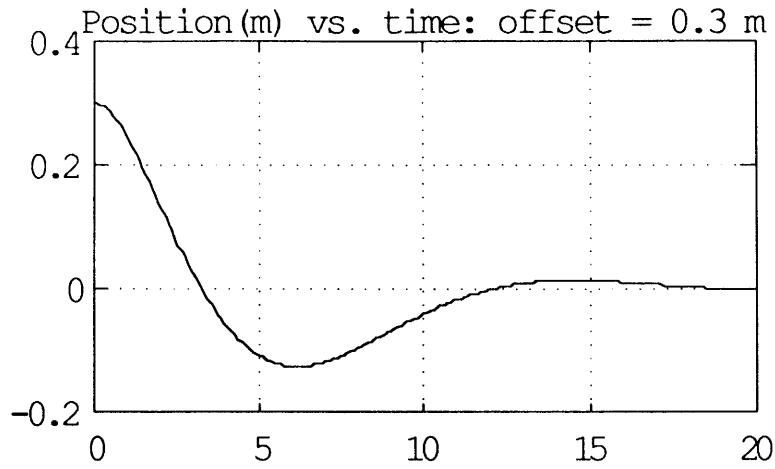
% Initial offset responses (offset of 0.3 m)

clg;
y = lsim(x1, x2, x3, x4, u, t, x0);
subplot(221), plot(t, y(:,1));
grid;
title('Position(m) vs. time: offset = 0.3 m');
subplot(223), plot(t, y(:,2));
grid;
title('Velocity(m/sec) vs. time: offset = 0.3 m');
subplot(222), plot(t, y(:,3));
grid;
title('Control(N) vs. time: offset = 0.3 m');
pause;

% Constant disturbance response (|d| = 20 N)
```

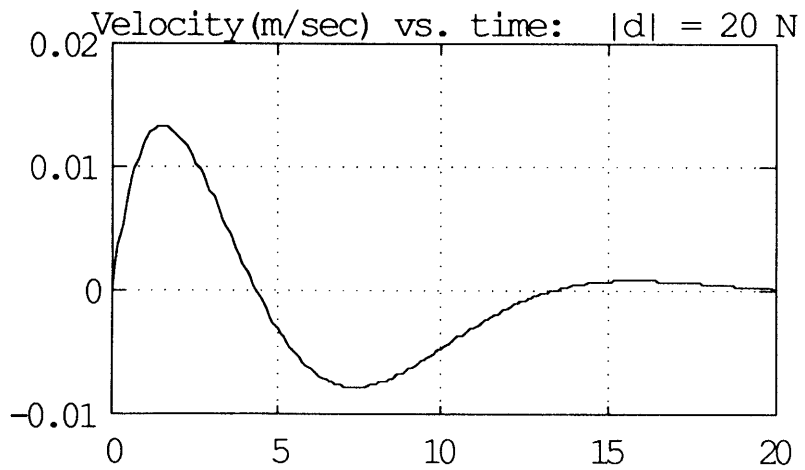
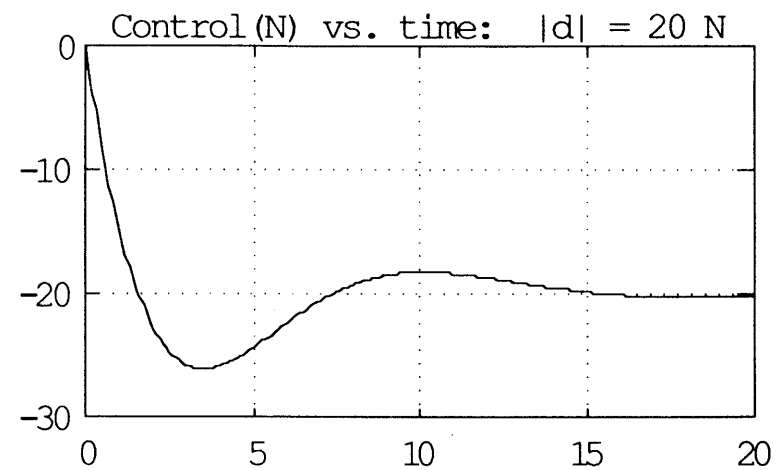
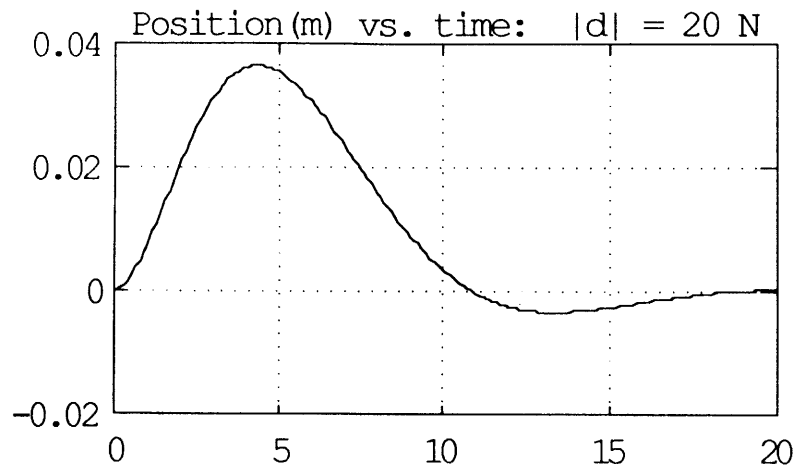
```
clg;
x0 = [0;0;0];
x2 = minv*[0;0;20];
y = lsim(x1, x2, x3, x4, u, t, x0);
subplot(221), plot(t, y(:,1));
grid;
title('Position(m) vs. time: |d| = 20 N');
subplot(223), plot(t, y(:,2));
grid;
title('Velocity(m/sec) vs. time: |d| = 20 N');
subplot(222), plot(t, y(:,3));
grid;
title('Control(N) vs. time: |d| = 20 N');
pause;
```


Appendix E -- Control System Gain Calculations



Position Response with 0.3m Initial Offset

Appendix E -- Control System Gain Calculations



Position Response with 10 N Disturbance

Appendix E -- Control System Gain Calculations

```
%Attitude Response Analysis
%Lqr with q = I, r = 0.0001

doverl = 0.85/86.0;
a = [0 1 0; 0 0 1; 0 0 0];
b = [0; 0; doverl];
q = eye(3);
r = 0.0001;

[G,s] = lqr(a,b,q,r);

% Note that for the given q and r,
% gi = 100.0, gp = 242.0, gd = 242.8

%Time responses to offset, constant disturbance

t = [0.0: 0.1: 20.0];
x0 = [0.0;0.4;0];
u = ones(201,1);

x1 = a-(b*g);
x2 = [0;0;0];
x3 = [0 1 0; 0 0 1; -g];
x4 = zeros(3,1);

% Initial offset responses (offset of .4 rad)

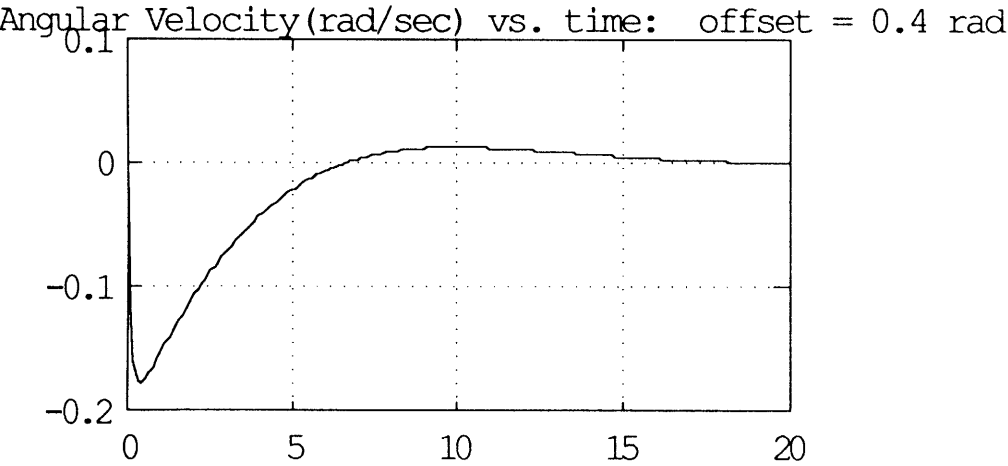
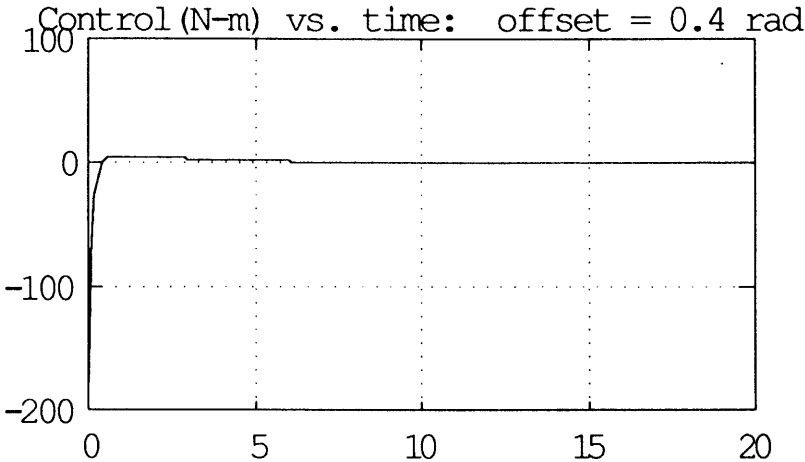
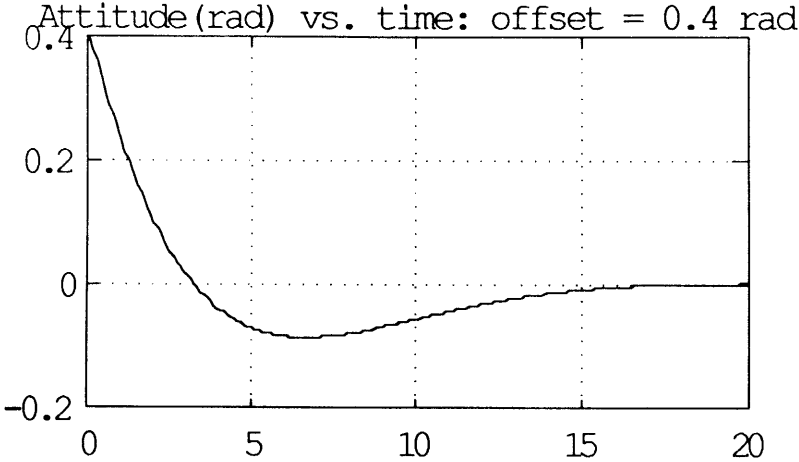
clg;
y = lsim(x1, x2, x3, x4, u, t, x0);
subplot(221), plot(t, y(:,1));
grid;
title('Attitude (rad) vs. time: offset = 0.4 rad');
subplot(223), plot(t, y(:,2));
grid;
title('Angular Velocity (rad/sec) vs. time: offset = 0.4 rad');
subplot(222), plot(t, y(:,3));
```

```
grid;
title('Control(N-m) vs. time: offset = 0.4 rad');
pause;

% Constant disturbance response (|d| = 10 N-m)

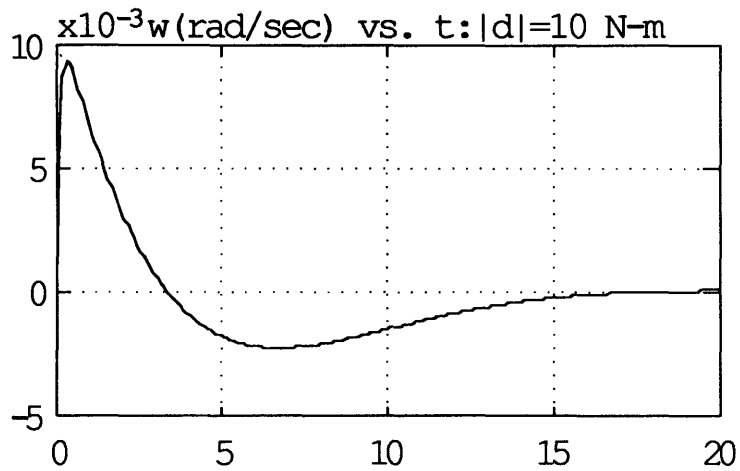
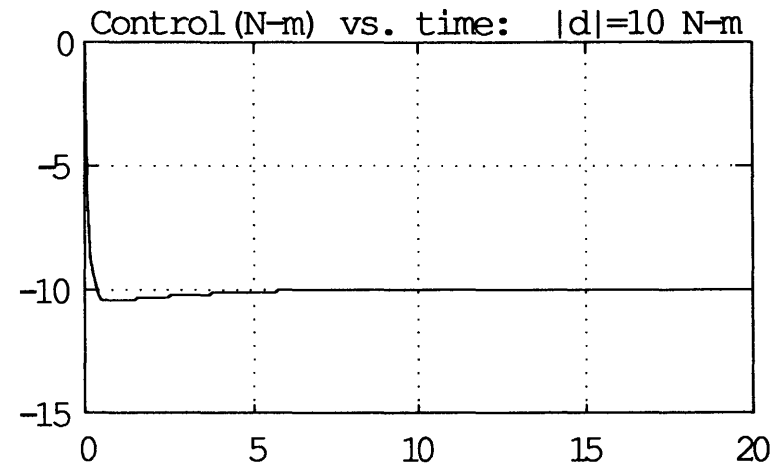
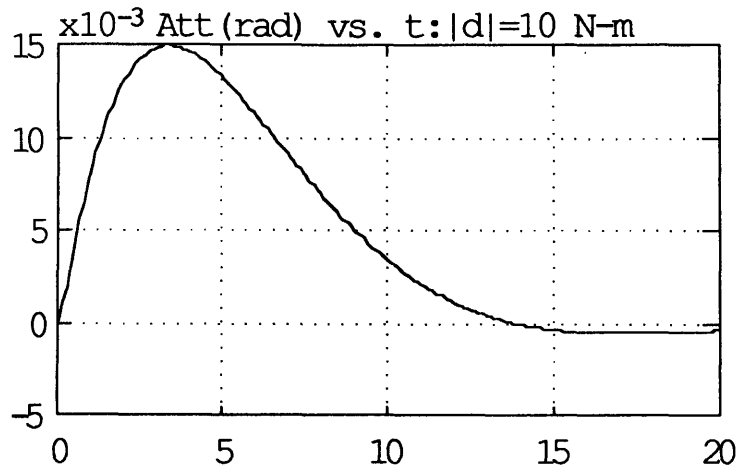
clg;
x0 = [0;0;0];
x2 = doverl*[0;0;10];
y = lsim(x1, x2, x3, x4, u, t, x0);
subplot(221), plot(t, y(:,1));
grid;
title(' Att(rad) vs. t:|d|=10 N-m');
subplot(223), plot(t, y(:,2));
grid;
title(' w(rad/sec) vs. t:|d|=10 N-m');
subplot(222), plot(t, y(:,3));
grid;
title('Control(N-m) vs. time: |d|=10 N-m');
pause;
```

Appendix E -- Control System Gain Calculations



Attitude Response to 0.4 rad Initial Offset

Appendix E -- Control System Gain Calculations



Attitude Response to 10 N-m Disturbance