Closed-Loop Depth and Attitude Control of an Underwater Telerobotic Vehicle

by

WENDY MARIE POWER

S.B. Aeronautics and Astronautics, Massachusetts Institute of Technology (1987)

Submitted to the Department of Aeronautics and Astronautics
in Partial Fulfillment of the Requirements of the Degree of
Master of Science in Aeronautics and Astronautics

at the

Massachusetts Institute of Technology

June 1990

Signature of Author [illegible]
Department of Aeronautics and Astronautics
March 7, 1990

Certified by
Professor David L. Akin
Thesis Supervisor

Accepted by [illegible]
Professor Harold Y. Wachman
Chairman, Department Graduate Committee

Aero

Closed-Loop Depth and Attitude Control of an Underwater Telerobotic Vehicle

by

WENDY MARIE POWER

Submitted to the Department of Aeronautics and Astronautics on March 7, 1990 in partial fulfillment of the requirements of the Degree of Master of Science in Aeronautics and Astronautics

ABSTRACT

Underwater teleoperators are used in research to simulate task performance by robots in the space environment. In order to effectively use the teleoperators, they must be easily and accurately controlled. Difficulties in implementing an accurate closed-loop attitude controller arise due to the torque moments resulting from the application of the teleoperator's thrust vectors, the difficulty in quantifying the dynamics of the teleoperator due to the effects of the water inside and outside of the robot, water drag, and the inability to measure the yaw angle about the local vertical. This research deals with the development of one teleoperator, Apparatus for Space Tele-Robotic Operations (ASTRO), and the implementation of a proportional-integral-derivative attitude controller for roll and pitch angles based on an equivalent angle-axis representation of the teleoperator orientation. A proportional depth controller was also implemented. The controller was tested in attitude regulation, step response to a 45 degree change in roll or pitch angle, and depth and attitude disturbances. The controller successfully held the desired attitude to within ±5 degrees, though the rotational drift about the gravity vector could not be completely eliminated. The depth controller was accurate within ± 0.5 feet.

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

Dedication

This thesis is dedicated to my parents, whose constant love, understanding, and encouragement has enabled me to set and reach ever-higher goals.

Acknowledgements

I would like to thank Professor Dave Akin for having the foresight to hire me as an untried but persistent freshman and to encourage and support me through my many years of growth in the Space Systems Lab. Thanks also go to Professor Sandy Alexander for his encouragement, help, and explanations that made even the most muddled and confused problems become clear. I would also like to recognize the many UROPs who have had a hand in the ongoing development of ASTRO and the divers whose patience during ASTRO's endless balancing, zeroing, and pool tests in general is greatly appreciated. Special thanks go to those volleyball studs, Matt and Russ, whose enthusiasm for playing gave me the opportunity to vent my frustrations on a defenseless little ball. Recognition also goes to the residents of 33-407, both past and present, for their friendship and willingness to share ideas and knowledge. Thanks also to the many friends who have shared hopes, fears, and classes. Special thanks and love go to Dana Johnson whose eternal patience, humor, and love has made the bad times better and the good times extraordinary.

Special acknowledgement goes to the Hughes Aircraft Company, Space and Communications Group, and especially the Control Electronics Department for sponsoring part of my graduate study and providing me with the opportunity to work in the "real world".

This research was supported by NASA grant NAGW-21.

Table of Contents

List of Figures

List of Tables

1.0 Introduction

Human presence in space has opened up new and exciting fields of research. One of these is the search for viable alternatives to the exclusively human performance of tasks in this environment. The desire for alternatives is due to the cost of training and equipping an astronaut, and to the potential danger of exposing a human to the space environment for extended periods of time. Another factor is human fatigue, which limits the amount of time available for any particular task. A robot can better perform the boring and repetitious tasks that sometimes result in human error. The use of robots in space to assist the astronauts or perhaps replace them for certain tasks may alleviate some of these concerns. Thus, a new field of research is the study of man-machine systems in space, and their capabilities and relative performance.

One of the main research interests of the MIT Space Systems Laboratory is the investigation of man-machine systems in a simulated zero-gravity environment. For the purpose of this research, underwater teleoperated robots have been constructed. They were designed to test new ideas in man-machine systems, and are currently used as testbeds for equipment relevant to tasks that will be performed in space. Testing is performed underwater to simulate the zero-gravity of space, thus utilizing the buoyancy of the robots to counteract the effects of gravity. To this end, three teleoperators have been developed by the Space Systems Laboratory. The first, the Beam Assembly Teleoperator (BAT), has been used as a testbed for several research projects related to tasks performed in space. These include dexterous manipulation, space assembly, partial automation, and stereo teleoperator vision. The second teleoperator developed, the Multimode Proximity Operations Device (MPOD), was designed to perform docking tasks and to be operated onboard as well as remotely. The most recent teleoperator, called Apparatus for Space TeleRobotic Operations (ASTRO), was developed as a multi-purpose testbed for satellite servicing. It will be discussed in detail below. Testing for this research was performed at the MIT Alumni Pool and the NASA Marshall Space Flight Center Neutral Buoyancy Simulator in Huntsville, Alabama. The general experimental system is shown in Figure 1.0.1.

ASTRO was designed to be a multi-purpose, reconfigurable second-generation teleoperator. The physical design of the robot was intended to facilitate maintenance and to shorten the time necessary to prepare it for testing. It was also intended to allow easy reconfiguration of the teleoperator for future research experiments. In addition, an onboard computer and an advanced sensor system were implemented.

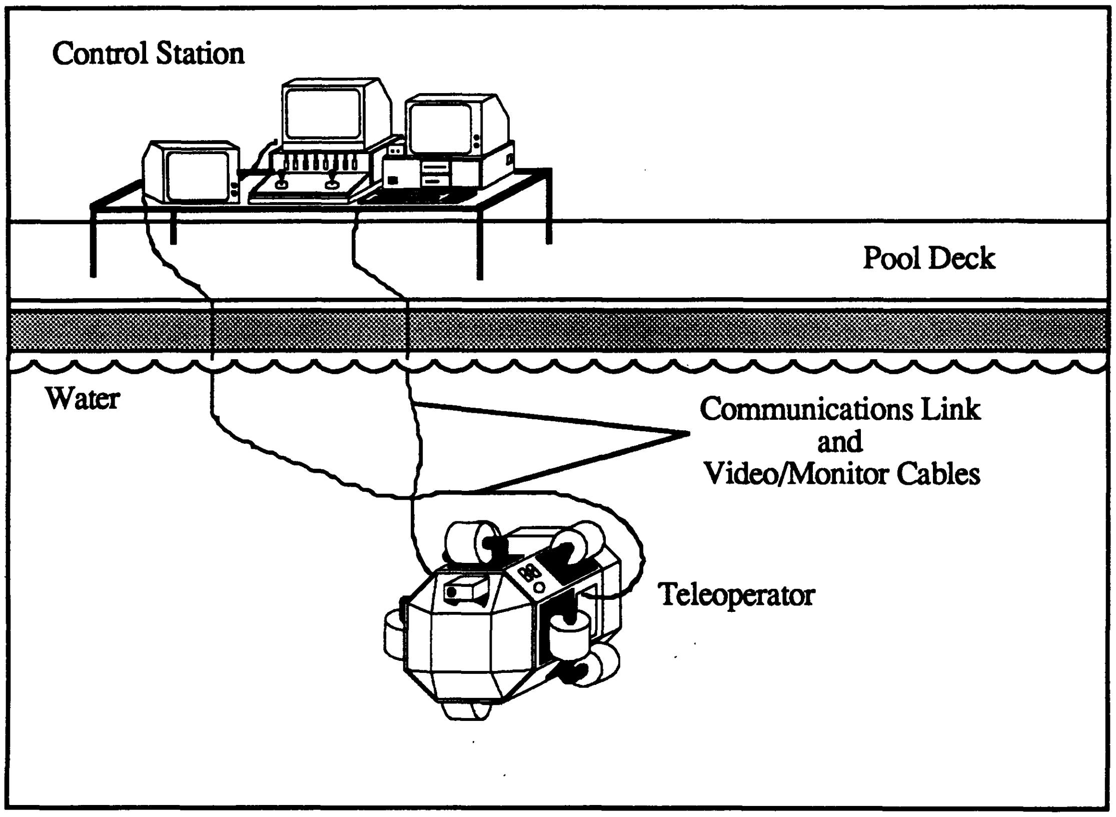


Figure 1.0.1 Pool Test Set-Up

The primary goal of the sensor system was to provide information on the teleoperator's orientation. A simple three-axis rate sensor was the easiest way to obtain direct rate feedback data. Several options existed for measuring the teleoperator's attitude. However, all of these were unable to measure the rotation angle about the local $\hat{z}$ axis. This will be discussed in more detail in Section 2.2. The sensors chosen to provide attitude data were pressure transducers. These provided depth information for specific points in the teleoperator. This depth data was then used to calculate the roll and pitch angles of the teleoperator.

An efficient and reliable closed-loop controller for the underwater robot was necessary so it could be used as an easily and accurately controlled testbed. Using hand controllers to specify XYZ translation and RPY (roll, pitch, and yaw) rotation was not sufficient. Problems with open-loop (hand controller) control were apparent when attempts were made to "fly" the teleoperator around the pool. A specific attitude was very difficult to maintain. Pure rotation or translation commands from the hand controllers resulted in torques exerted on the teleoperator during flight due to the misalignment of the thrusters with the teleoperator center of mass. In addition, the physical restrictions of the thrusters resulted in unequal thrust levels for any given command. During testing, a subject should not be required to deal with correcting all the coupling via the hand controllers. This distracts the test subject and takes away energy and concentration from the primary test objective. Therefore, an accurate closed-loop attitude controller was needed.

2.0 ASTRO Development

The original purpose for developing ASTRO was to provide a new testbed for current research. The physical design of the teleoperator was based on its primary function of servicing satellites. The necessary requirements for a satellite servicer were obtained from reports and specifications of the Solar Maximum Mission and the Hubble Space Telescope [1]. The results of this background research yielded many specifications on manipulators for the robot, but only general size requirements for the robot itself. The conclusion drawn was that the robot needed to be highly serviceable and reconfigurable. This indicated a need for modularity. In essence, ASTRO was designed to be a generalized testbed that could be adapted to various new manipulators (as well as to other research projects). In addition to the above design criteria, problems associated with the two older teleoperators were taken into account and eliminated during the design phase. These problems included numerous separate access panels on the outside of the robot that greatly increased preparation and repair time. In addition, the use of many sealed boxes to house electronics, sensors, and batteries was not space efficient and provided opportunities for leaks. The resultant design included modularity and easy access for maintenance, with simple and easily isolated subsystems and interchangeable parts. Advanced electronics in the form of onboard computers and sensors were also included.

2.1 ASTRO Physical Design

The basic construction of underwater teleoperators consisted of an internal aluminum frame surrounded by relatively smooth exterior buoyancy panels made of fiberglass and high density foam. Vital electronics and sensors, and the batteries used as the power sources for the thrusters and electronics, were sealed inside several separate boxes mounted at various points to the internal frame. A color video camera was sealed inside another box and mounted on the front of the teleoperator. A scuba tank used to pressurize the thrusters and the sealed boxes, as well as to provide high pressure for pneumatic devices, was mounted to the internal frame. The regulator attached to the scuba tank was used to keep the pressure in the sealed boxes (control, camera, and solenoid) one to two pounds per square inch (psi) over the water pressure outside of them to prevent leaks or failure of the box structure at greater depths. The solenoid box contained high pressure solenoids that were used to operate the main power relay and pneumatic devices.

ASTRO's body was separated into three basic components: the back, which consisted solely of the battery box; the main body, which consisted of the center propulsion unit shell and the internal frame; and the front which was used for mounting manipulators and cameras. Like the other teleoperators, the main (central) section of ASTRO was built

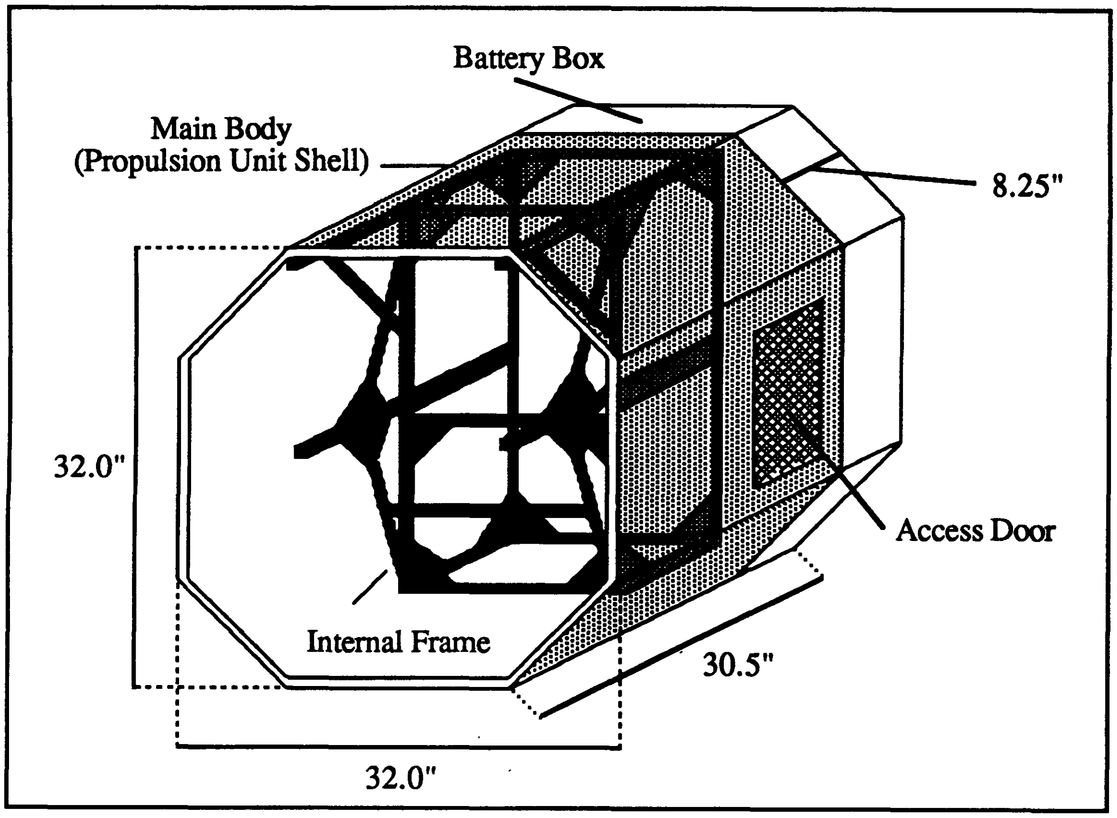


Figure 2.1.1 ASTRO Shell With Internal Frame

around an internal aluminum frame. However, instead of multiple panels around the exterior, ASTRO consisted of a single octagonal shell mounted to the internal frame (Figure 2.1.1). This shell was a tube (32.0 inches in diameter and 22.25 inches long), that left the front and back ends free for adding the other sections (the battery box and the front lid or manipulator mount). This shell was removed when extensive structural work was necessary, but for the everyday use and maintenance of the teleoperator it remained attached. Additional access to the interior of the robot was provided by two side doors.

A single large battery box was mounted to the back end of the teleoperator. This battery box contained all of the batteries required to meet the power demands of the teleoperator and was removed from the main body of the robot only when necessary (Figure 2.1.2). It was the same shape and size as the cross-section of the main body of the teleoperator and was 8.25 inches deep. Internal dividers separated the interior of the box into 12 square sections and four triangular ones. Each square section contained a 12 volt lead acid gel cell battery, while the triangular ones provided surface area for mounting connectors and fuses. The ten main batteries supplied power to the eight thruster units and the pneumatic systems, while the control batteries powered all of the electronics and sensors. The battery box was designed to allow charging and maintenance while attached

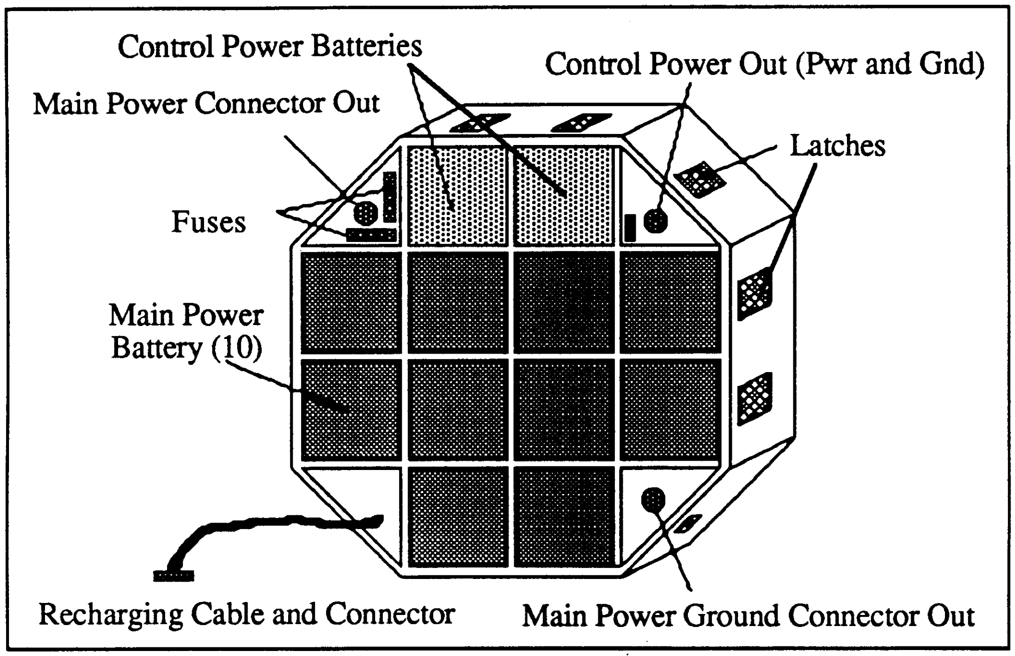

Control Power Batteries
Control Power Out (Pwr and Gnd)
Main Power Connector Out
Latches
Fuses
Main Power Battery (10)
Recharging Cable and Connector
Main Power Ground Connector Out

Figure 2.1.2 ASTRO Battery Box

to the teleoperator, and simply required removal of the lid. The lid was held on by latches placed around the sides of the battery box. When the box was sealed, it was purged of air by flushing it with inert nitrogen gas to eliminate the possibility of fires. The three power and ground connectors (separate connectors for main power and ground and a single connector for control power and ground) passed through the back of the battery box into the interior of the robot.

The front end of the teleoperator allowed access to all the interior systems, and was closed during testing by a large removable lid. The control box was mounted to the internal frame from the front of the teleoperator (Figure 2.1.3). It contained all of the critical electronics and sensors. It could be easily removed so that the interior components could be serviced. Figure 2.1.4 shows the teleoperator when closed. The entire configuration made the robot very easy to work on and service, and left a significant amount of space inside the teleoperator for later system expansion.

The shape of the teleoperator was chosen to supply a large amount of symmetrical surface area for mounting thrusters and other hardware, and also to reduce water drag, making underwater motion more efficient. This was important because symmetry enabled a more equal force balance from the thrusters. The thrusters could thus be mounted in such a way that the thrust vectors from each unit were parallel to the principal axes of the teleoperator. This is shown in Figure 2.1.5. Each thruster could be operated at 16 different thrust levels (15 thrust levels and off) in the forward and reverse directions.

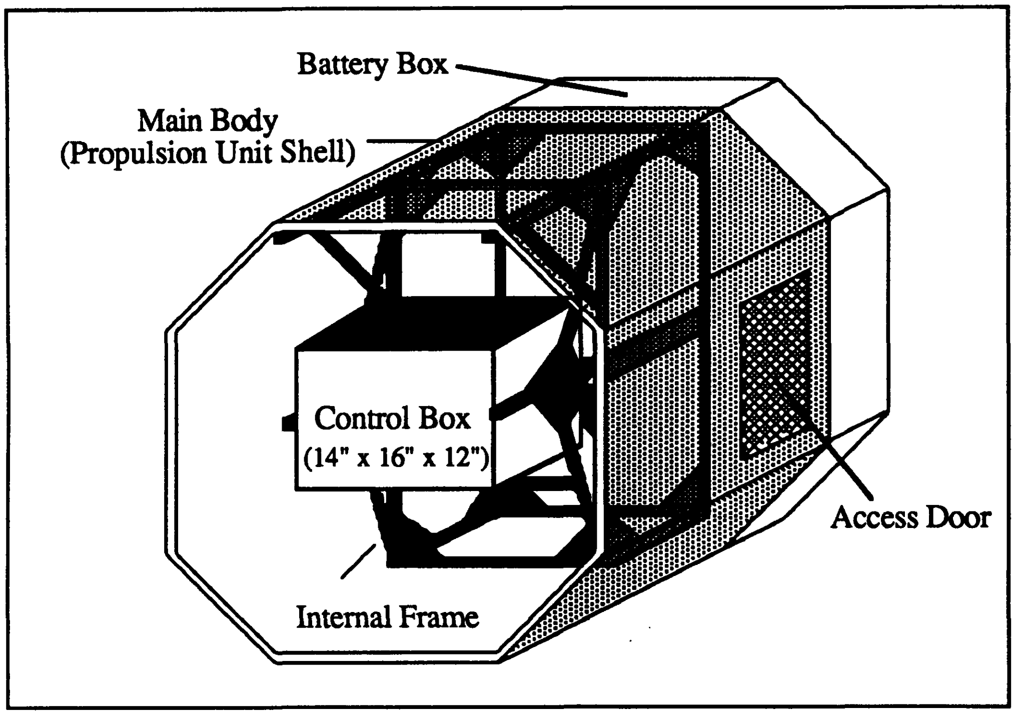


Figure 2.1.3 ASTRO Control Box in Teleoperator

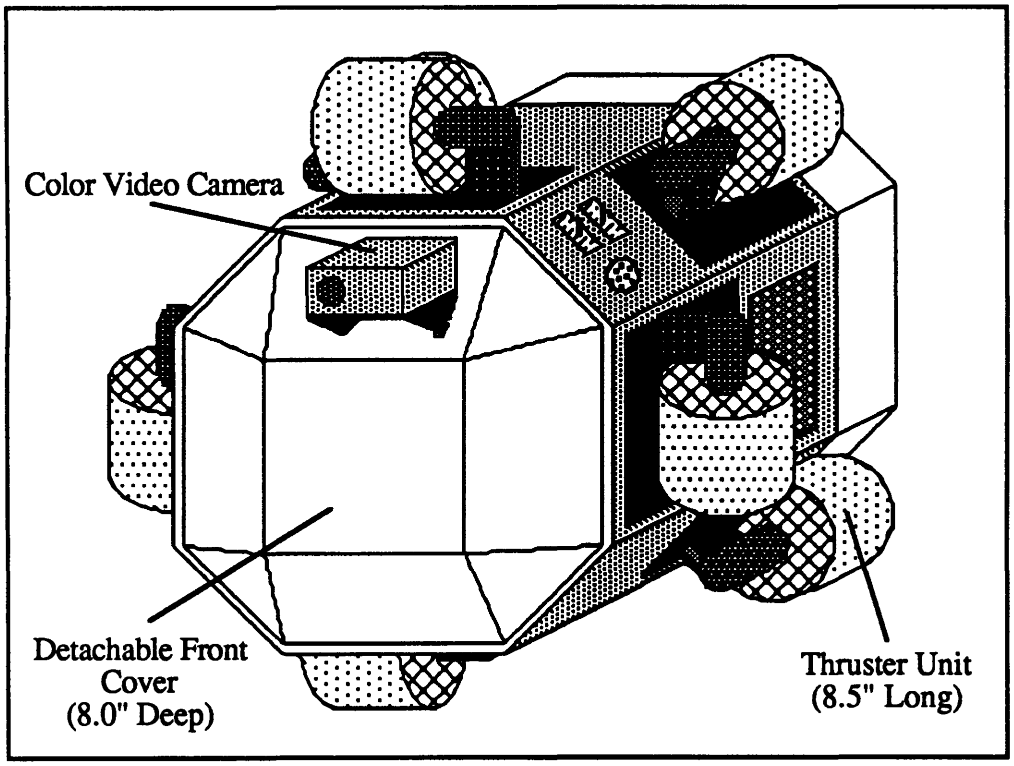


Figure 2.1.4 ASTRO Front View

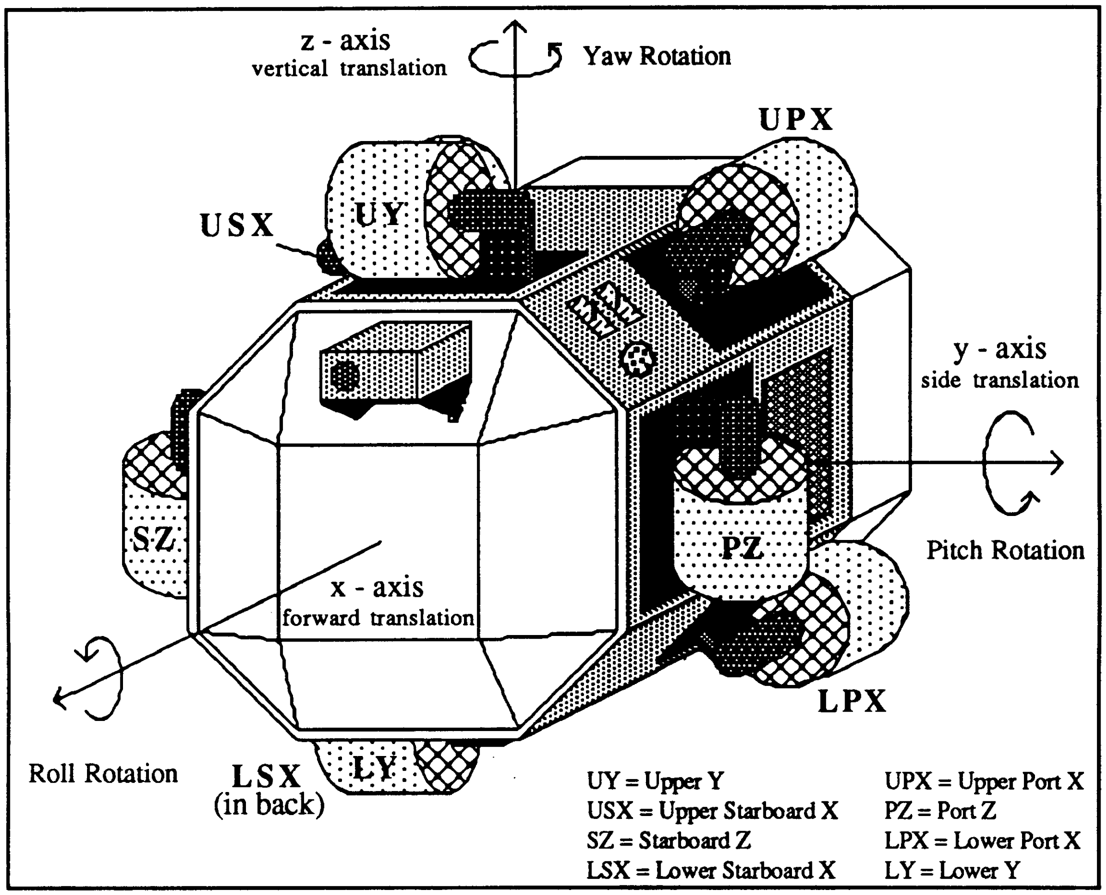


Figure 2.1.5 ASTRO Principal Axes, Rotations, and Thrust Vectors

There were four x-thrusters (lined up so that the thrust vector was parallel to the teleoperator's principal x-axis), two y- and two z-thrusters. Past experience with operating underwater teleoperators showed that there was a tendency to rely mostly on x-translation to maneuver about the pool. This was primarily due to the robot operator's reliance on the video camera mounted on the front of the robot for a view of the worksite and flight path. In addition, y and z translations were generally used only for minor adjustments at a worksite, so smaller amounts of thrust were needed. Teleoperator rotations about the principal axes were produced by using two symmetric pairs of thrusters, with one pair operating in reverse. Figure 2.1.5 also shows the principal axes of the teleoperator and the corresponding rotations. The right hand rule was used to determine the sign (positive/negative) of the rotation. The thrusters were labeled according to their locations on the robot and the directions of the output thrust vectors. Table 2.1.1 lists the thruster combinations used for each translation and rotation. Each thruster unit was made up of a Minnkota trolling motor, a screened duct, and driving electronics.

Table 2.1.1 Thruster Combinations

Translation/Rotation	Forward Thrust Motors	Reverse Thrust Motors
+X	UPX, LPX, USX, LSX	---
-X	---	UPX, LPX, USX, LSX
+Y	UY, LY	---
-Y	---	UY, LY
+Z	PZ, SZ	---
-Z	---	PZ, SZ
+Roll	LY, PZ	UY, SZ
-Roll	UY, SZ	LY, PZ
+Pitch	UPX, USX	LPX, LSX
-Pitch	LPX, LSX	UPX, USX
+Yaw	USX, LSX	UPX, LPX
-Yaw	UPX, LPX	USX, LSX

To adjust the neutral buoyancy of the teleoperator, lead weights and flotation blocks were attached to the robot by thumb screws. Screw inserts were imbedded or attached to the robot on all of the outer surfaces. Adjustments were made, if necessary, at the beginning of each test session when the teleoperator was first put in the water. Attempts were made to balance it in depth (i.e. it stayed at the depth it was placed at) and in rotation about all its axes (i.e. it stayed in any given orientation). The balance was never perfect, but the controller derived in Section 3 was intended to compensate for the remaining error.

2.2 ASTRO Avionics

The most important sensors on ASTRO were those that provided the data for the closed-loop attitude controller. There were several major factors that influenced the choice of the sensors. The cost of the system needed to be relatively affordable, approximately in the $5000 range, which was constrained by the research budget. The power requirements of the system were also very important. Due to the way the batteries were connected, the maximum input voltage to the sensors, computers, and other electronics was 12 volts. The sensors also needed to have a low current draw so that the power supply would last for a reasonable length of time, at least 2 hours. In addition, the sensors also had to be accurate enough to measure the range of rates and attitudes of the teleoperator system. Another factor that constrained the sensor choice was the size of the sensor package. There was limited space inside of the control box to house the sensors and it would have been inconvenient to contain them in a separate sealed box.

Sensor systems installed on previous teleoperators included a three-axis gyro package and pendulum inclinometers. The signal-to-noise ratio on the gyro package had degraded due to age and extensive use. This indicated a need for a more robust and enduring rate sensing system. The pendulum system installed for attitude determination was unreliable when attempting to measure rotations about the local vertical. In addition, it was a fairly bulky system (~7" x 7" x 7"), and would have accounted for about a third of the space in ASTRO's control box. Since there were no sensors that could accurately measure the rotation angle about the local vertical in the underwater environment, the decision was made to try a new way of measuring the attitude by using pressure transducers.

The 3-axis rate transducer package installed on ASTRO (Humphrey RT02-0608-1) was chosen for its small size, high accuracy, and conformance to the power restrictions of the system. Because of its small size (2.00" x 2.75" x 3.30"), the rate sensor package was mounted inside the control box. This made the interface between the sensor and the rest of the control system very simple. The supply voltage needed for the rate transducer was 12 volts, which was supplied directly from the control power batteries. The sensors were capable of measuring a rotation up to ±90 degrees per second about each axis, with an accuracy of ±1 percent of the full scale output. The linear output of the sensors ranged from -5.0 volts (maximum rotation rate in the negative direction) to +5.0 volts (for positive rotations). The only problem encountered when using these sensors was the need to constantly check the zero value and calculate the offset number. This drift was a result of the length of time the sensor had been running and seemed to be dependent on the temperature of the sensor and the environment in which it was sealed. The zero values

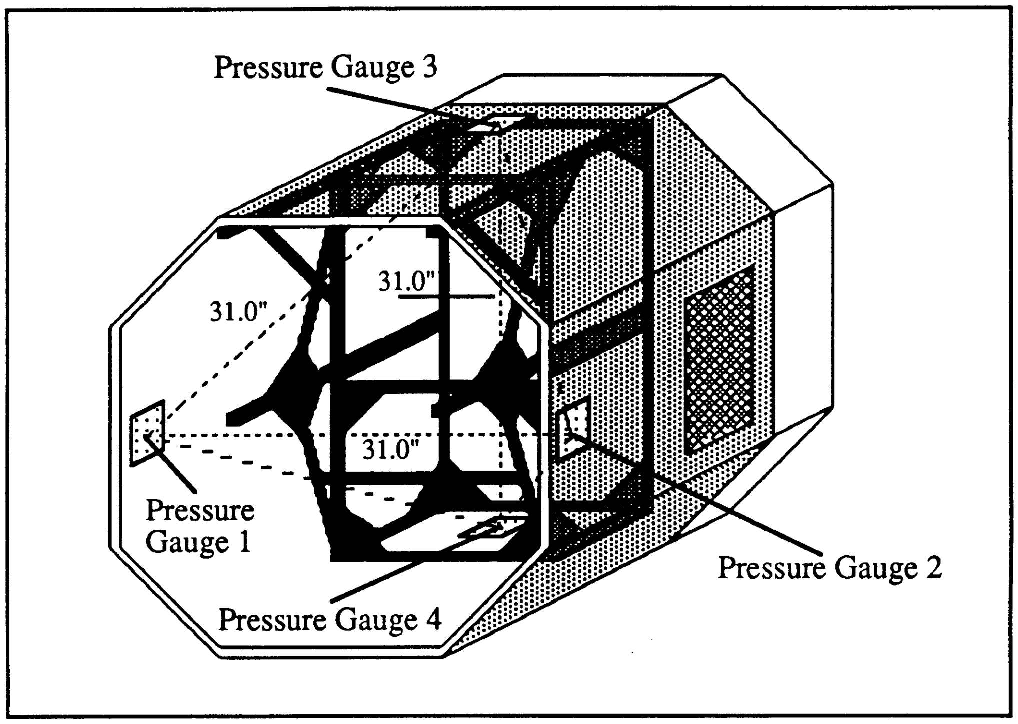


Figure 2.2.1 Pressure Gauge Locations

were checked and adjusted periodically by simply holding the teleoperator motionless and recalculating the zero value.

A set of four piezoresistive pressure transducers (Omega PX242-030G-5V) were chosen to provide depth and attitude data to the closed-loop controller. These devices were also relatively inexpensive ($132 each). The gauges required an 8 volt (80mA) power supply for the set of gauges. This was provided by a voltage regulator supplied by the 12 volt control power batteries. The four gauges were sealed inside a waterproof compound to protect the electronic connections, leaving only the input port exposed to measure water pressure. The gauges chosen were capable of measuring a pressure range of 0 to 30 psi. The linear output of the gauges ranged from 0 to 6 volts (at full pressure). In essence, the pressure transducers measured the depth of the teleoperator at the points where they were mounted. Figure 2.2.1 shows the locations of the gauges. They were mounted equal distances from each other about the approximate center point of the teleoperator. Averaging the four gauge values provided data on the depth of the center of the robot. This data was utilized in a proportional depth controller. The locations of the pressure transducers were specifically chosen to provide information on the attitude of the teleoperator. A change in the roll or pitch attitude resulted in a measurable change of depth between some combination of the pressure transducers. Using rotation matrices, these differences in depth were resolved into specific orientation angles. It should be noted that a rotation about the local vertical could not be detected, since there was no change in depth. However, this rotation could not be measured by any of the other sensor options investigated, so there was no loss of information. The calculations involved with the pressure transducers are discussed in Section 3. Additional information and specification sheets for the pressure and rate transducers are contained in reference [2].

The leak sensors were designed to take the guess work and worry out of the sealed boxes. Since the electronics, sensors, and batteries could not be exposed to water, it was very important to retrieve the teleoperator from underwater as soon as possible once water leakage had been detected (previous robots had no such detection ability). To this end, leak sensors were installed on the interior surfaces of the battery, control, and camera boxes. The leak sensors consisted of an etched circuit board and simple circuitry that detected a change in voltage across the circuit board. Software was written to trigger an alarm if any leaks were detected.

For onboard control, an IBM-PC-compatible AMPRO computer board was installed. This was an 8088 equivalent processor running at 7.16MHz with a total of 512K RAM. A 3.5" floppy disk drive attached to the processor facilitated program portability and decreased the software development time. The presence of a fully operational onboard computer made the programming and debugging process much simpler. The software was programmed in a high level language (C). The only additional systems necessary were the I/O expansion ports, an A/D converter, and the specialized circuitry necessary to command the thrusters. Figure 2.2.2 shows the structure and flow of the onboard processor system (for detailed diagrams of the circuits refer to the ASTRO Guide to Operations and Systems [2]). A fiberoptic communications link for transmitting signal commands to and from the surface was chosen to ensure a clear signal (no noise or voltage drop like that associated with a long, approximately 200 feet, wire cable). The fiberoptic cable was also very light weight, reducing the amount of cable drag on the teleoperator. The communications link operated at 9600 baud, limited by the onboard computer. The fiberoptic circuitry converted the light signal to one compatible with the AMPRO/PC RS232 serial port. Additional electronics were mounted on the interface card. Communications between the AMPRO board and the additional circuitry were carried out via the PC expansion bus and Intel Programmable Peripheral Interface chips (8255). Pressure and rate transducer data was acquired through a 12-bit analog to digital converter. Pulse width modulated thruster commands were generated on the interface card and sent to each thruster with a direction command (forward or reverse). The circuitry at each thruster was responsible for amplifying the command and switching the thruster on and off. The light aperture on the

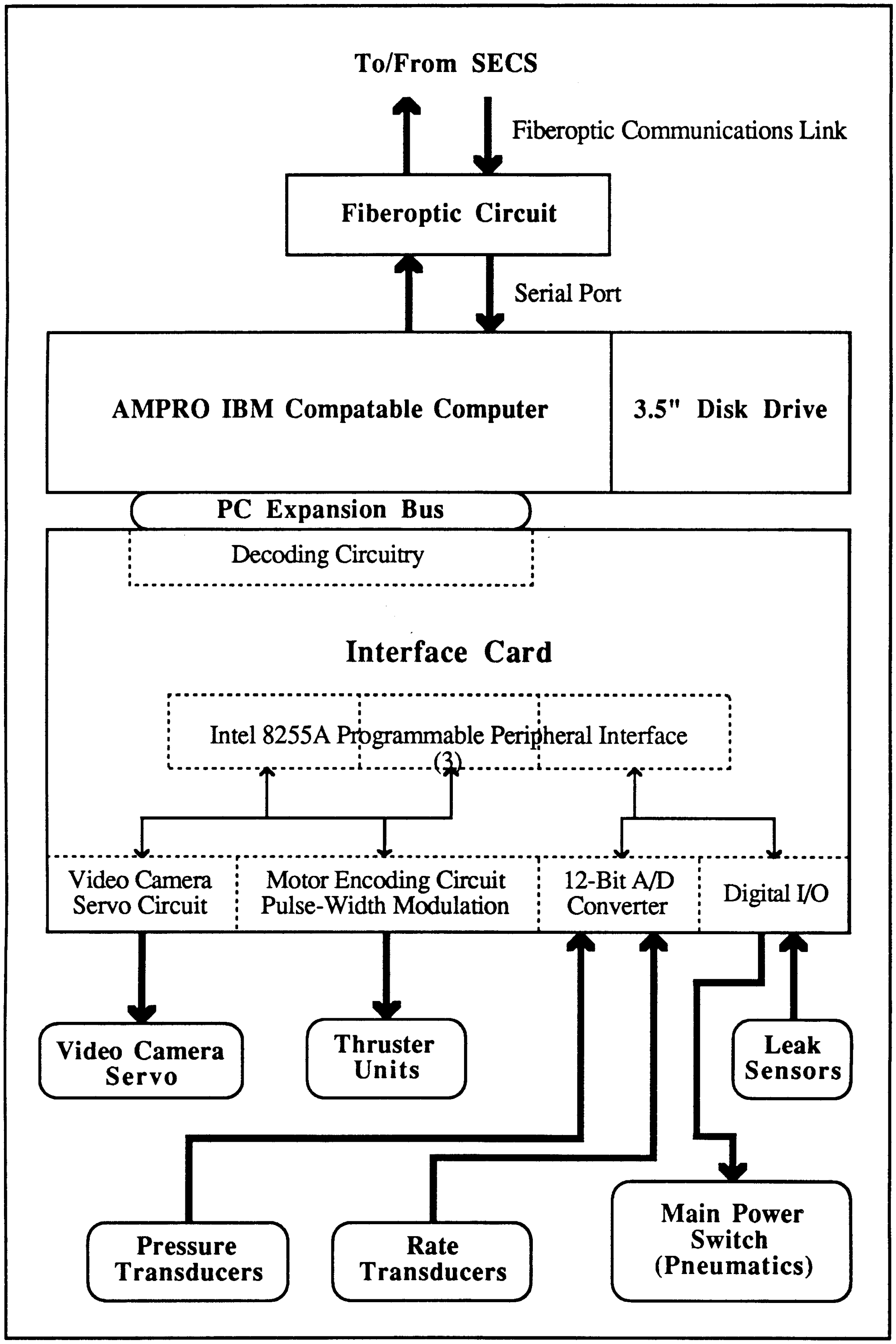


Figure 2.2.2 ASTRO Onboard Processor System

video camera was adjusted by an actuator servo. The servo was controlled by a frequency signal converted from a 4-bit digital command. As can be seen from the diagram, the various components of this system were relatively easy to isolate, thus making the system easy to service and upgrade. Power for the control system (electronics and sensors) was provided by two 12 volt lead acid gel cell batteries. The batteries were used to power the onboard disk drive, the 3-axis rate transducers, and the 8 volt regulator that supplied power to the pressure transducers. The 5 volts necessary for the PC board and the additional circuitry were supplied by a 5 volt linear (3 amp) voltage regulator, which in turn was powered by the 12 volts of control power.

A flowchart of the onboard software is shown in Figure 2.2.3. Communications with the surface via the fiberoptics was interrupt driven. It consisted of a formalized message structure called PIVECS [3] which could be easily reconfigured for different combinations of teleoperators and control stations. The main program served as a shell to call individual functions and to handle data acquisition. These functions were divided up along logical lines of separation and served to increase the clarity of the software and facilitate development and debugging. Appendix A contains a complete listing of the onboard software.

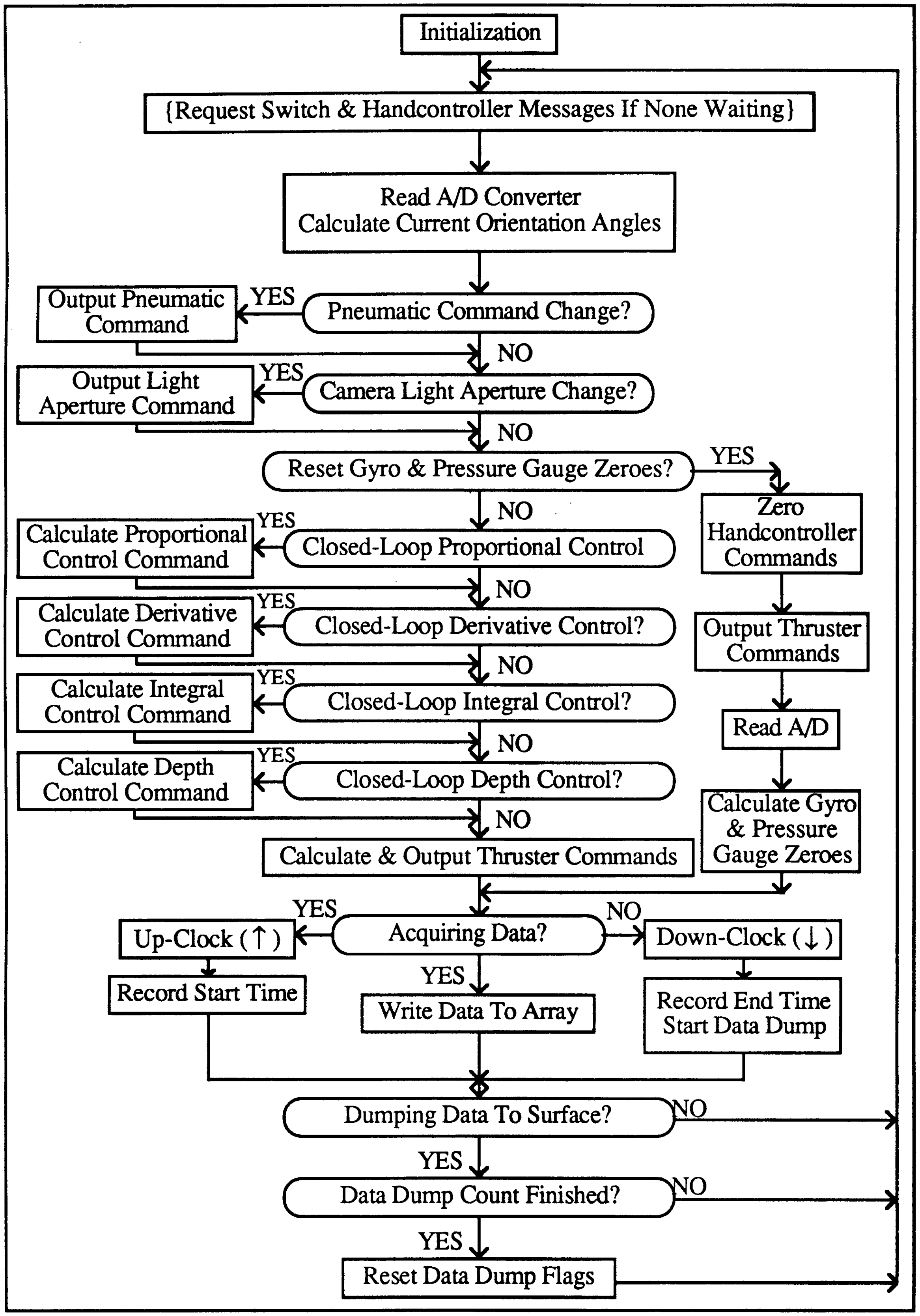

Initialization
{Request Switch & Handcontroller Messages If None Waiting}
Read A/D Converter
Calculate Current Orientation Angles
Pneumatic Command Change?
YES
Output Pneumatic Command
NO
Camera Light Aperture Change?
YES
Output Light Aperture Command
NO
Reset Gyro & Pressure Gauge Zeroes?
YES
Zero Handcontroller Commands
Output Thruster Commands
Read A/D
Calculate Gyro & Pressure Gauge Zeroes
NO
Closed-Loop Proportional Control
YES
Calculate Proportional Control Command
NO
Closed-Loop Derivative Control?
YES
Calculate Derivative Control Command
NO
Closed-Loop Integral Control?
YES
Calculate Integral Control Command
NO
Closed-Loop Depth Control?
YES
Calculate Depth Control Command
NO
Calculate & Output Thruster Commands
Acquiring Data?
YES
Up-Clock (↑)
Record Start Time
NO
Down-Clock (↓)
Record End Time Start Data Dump
YES
Write Data To Array
Dumping Data To Surface?
NO
YES
Data Dump Count Finished?
NO
YES
Reset Data Dump Flags

Figure 2.2.3 Flowchart of ASTRO Onboard Software

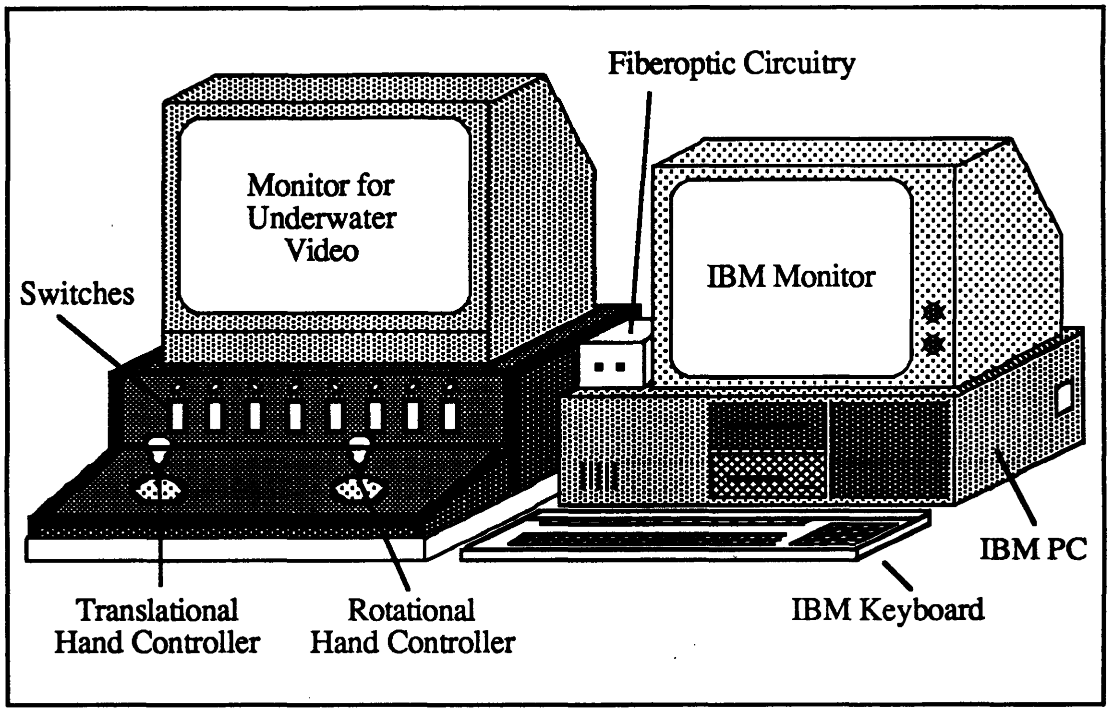

Fiberoptic Circuitry
Monitor for Underwater Video
Switches
IBM Monitor
IBM PC
Translational Hand Controller
Rotational Hand Controller
IBM Keyboard

Figure 2.3.1 Secondary Electronic Control Station (SECS)

2.3 Secondary Electronic Control Station - SECS

SECS is shown in Figure 2.3.1. It was not designed to be a final configuration, human-factored control station. Rather, it was built with the bare necessities of computers, hand controllers, and switches for running ASTRO and acquiring data. It has proved to be adequate for the job, and has provided insight into the basic necessities and configurations for future control stations.

SECS consists of an IBM personal computer, a color video monitor, two three degree-of-freedom hand controllers, and eight switches. The video monitor was directly linked to the underwater video camera mounted on the front of ASTRO, and provided the operator with a view of the area in front of ASTRO. This view provided visual cues regarding teleoperator orientation and rotation, using such details as the lines in the pool and stationary objects. The two hand controllers were used for specifying XYZ translation (left hand controller) and RPY (roll, pitch, yaw) rotation (right hand controller). The hand controllers acted like potentiometers, with the resistance in each direction of movement (left/right, up/down, and twist) proportional to displacement from the center (zero) position. This resistance was transformed into a voltage, amplified, and read into the computer through an analog to digital (A/D) converter. The eight switches were used to control various functions including main power, proportional, derivative, and integral control, pressure and rate transducer zero resets, and data acquisition. The switch value (on/off) was read directly into the personal computer. The system diagram is shown in Figure 2.3.2.

The software on the personal computer was used to acquire commands and to send control station information to the teleoperator. It was also used to provide the operator with teleoperator system command and status displays. A listing of the control station software is contained in Appendix B. Figure 2.3.3 shows the displays and their locations. These displayed parameters included information on leaks and whether they were in the control, camera, or battery boxes, the status of data being received, the light aperture command for the video camera, the proportional, derivative, integral, and depth gains, and the current data file name. Optionally displayed were the hand controller and switch commands, and the pressure and rate transducer uplink data. These optional displays were automatically turned off during data acquisition to increase the speed of the system. The personal computer keyboard was used to set new gains and orientation angles, to turn on and off the optional display and leak alarm, to control the light aperture of the camera, and to signal data acquisition and input data file names. Table 2.3.1 lists these keyboard commands. The monitor cable for the onboard computer was used during testing so that the state of the communications link between the two computers was monitored.

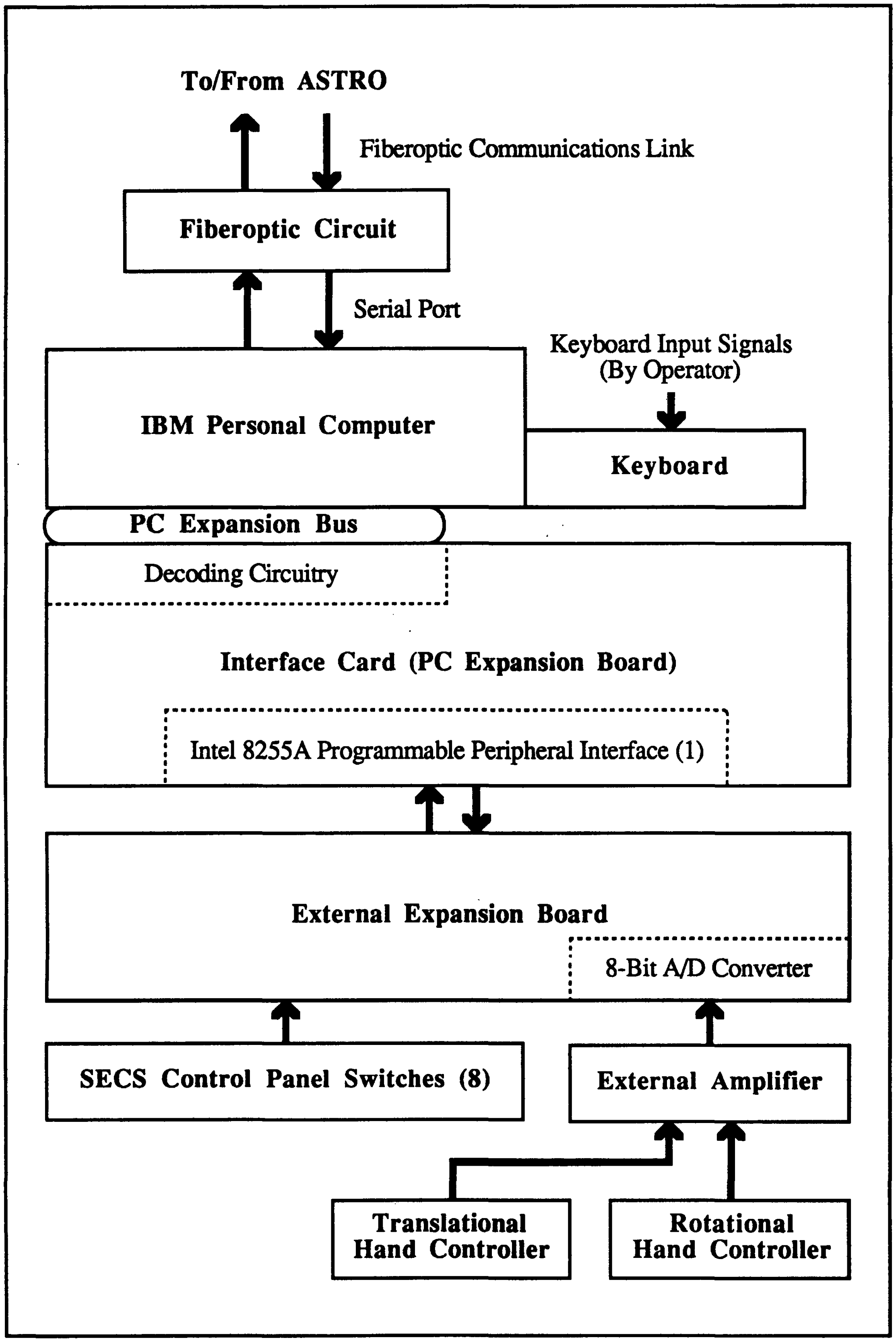
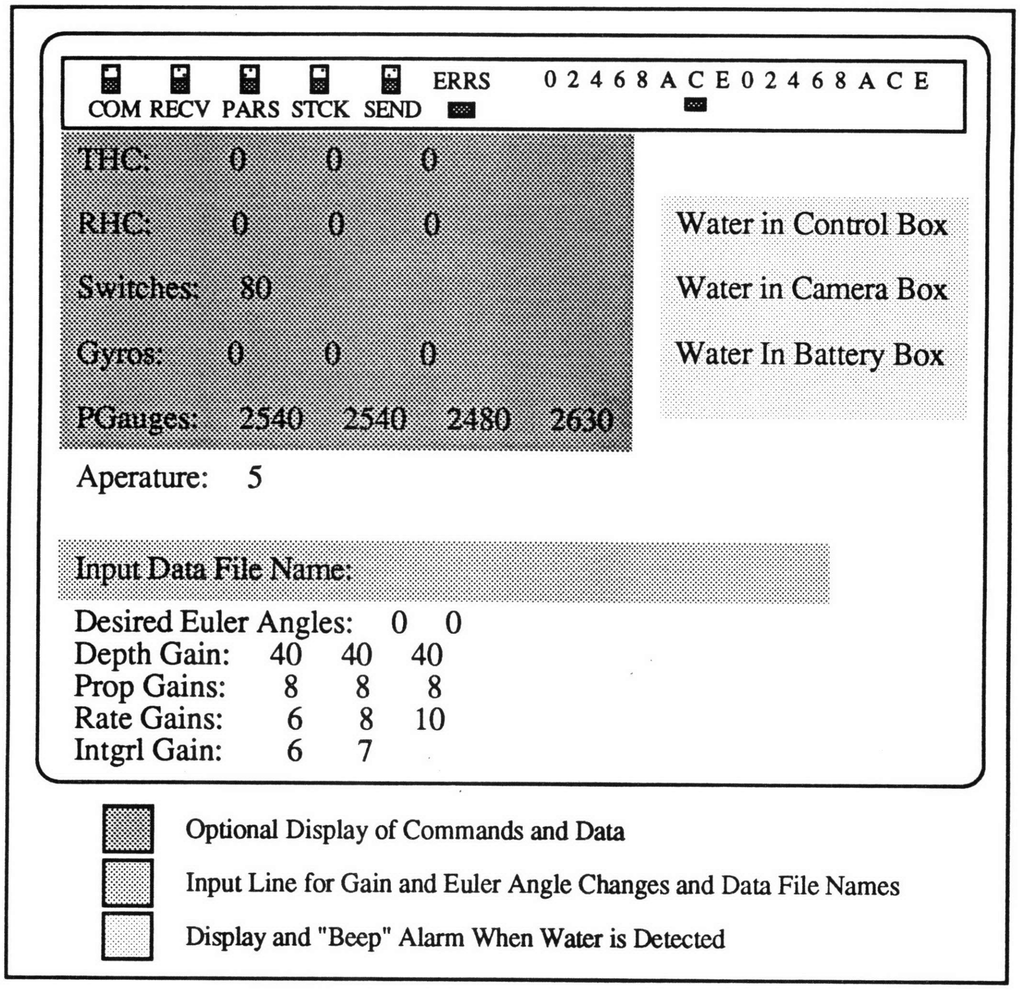

To/From ASTRO
Fiberoptic Communications Link
Fiberoptic Circuit
Serial Port
Keyboard Input Signals (By Operator)
IBM Personal Computer
Keyboard
PC Expansion Bus
Decoding Circuitry
Interface Card (PC Expansion Board)
Intel 8255A Programmable Peripheral Interface (1)
External Expansion Board
8-Bit A/D Converter
SECS Control Panel Switches (8)
External Amplifier
Translational Hand Controller
Rotational Hand Controller

Figure 2.3.2 SECS System Block Diagram



```
COM RECV PARS STCK SEND   ERRS     0 2 4 6 8 A C E 0 2 4 6 8 A C E

THC:        0      0      0
RHC:        0      0      0                 Water in Control Box
Switches:   80                               Water in Camera Box
Gyros:      0      0      0                  Water In Battery Box
PGauges:    2540   2540   2480   2630
Aperature:  5

Input Data File Name:
Desired Euler Angles:   0    0
Depth Gain:   40    40    40
Prop Gains:    8     8     8
Rate Gains:    6     8    10
Intgrl Gain:   6     7
```

▨	Optional Display of Commands and Data
▨	Input Line for Gain and Euler Angle Changes and Data File Names
▨	Display and "Beep" Alarm When Water is Detected

Figure 2.3.3 SECS Computer Displays

Table 2.3.1 SECS Keyboard Commands

i = Increment camera light aperture (darker)
d = Decrement camera light aperture (lighter)
s = Show command and uplink data display
o = leak sensor On/Off "beep" toggle
a = Acquire (input) data file name
e = set Euler angles
t = Transmit euler angles
p = set Proportional gains
r = set Rate (derivative) gains
j = set integral (j) gains
l = set depth (Level) gains

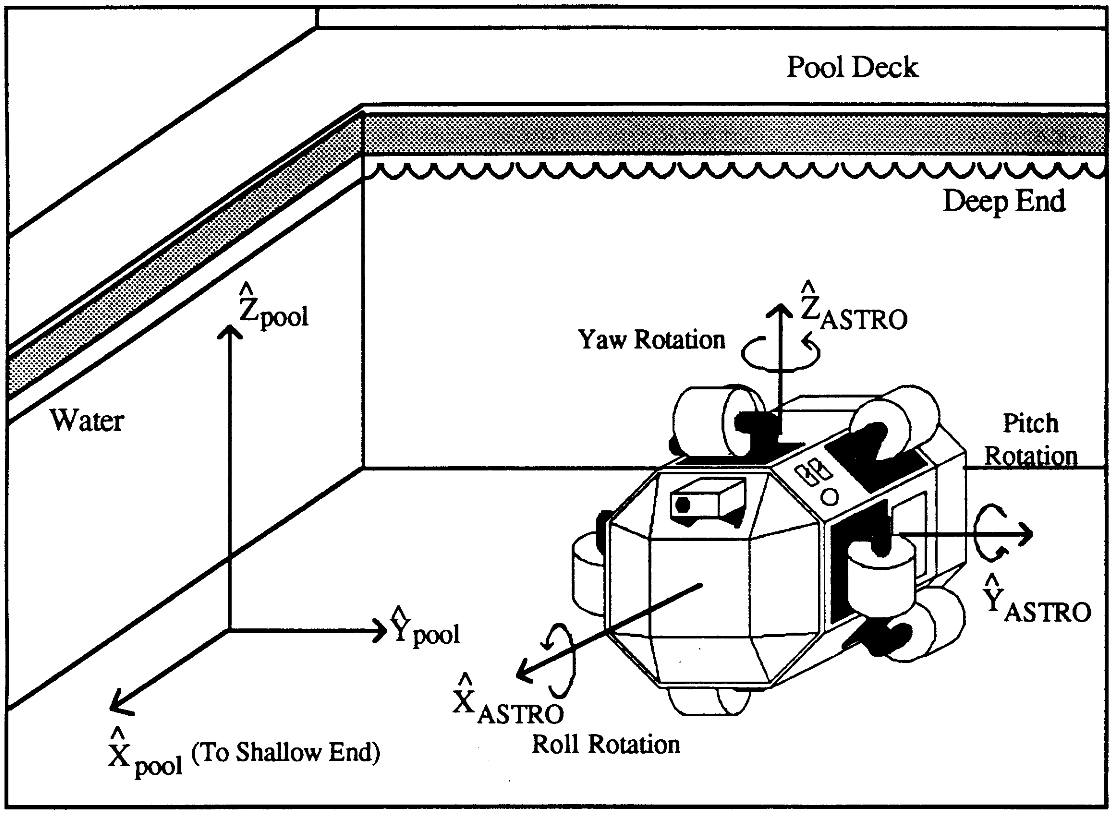


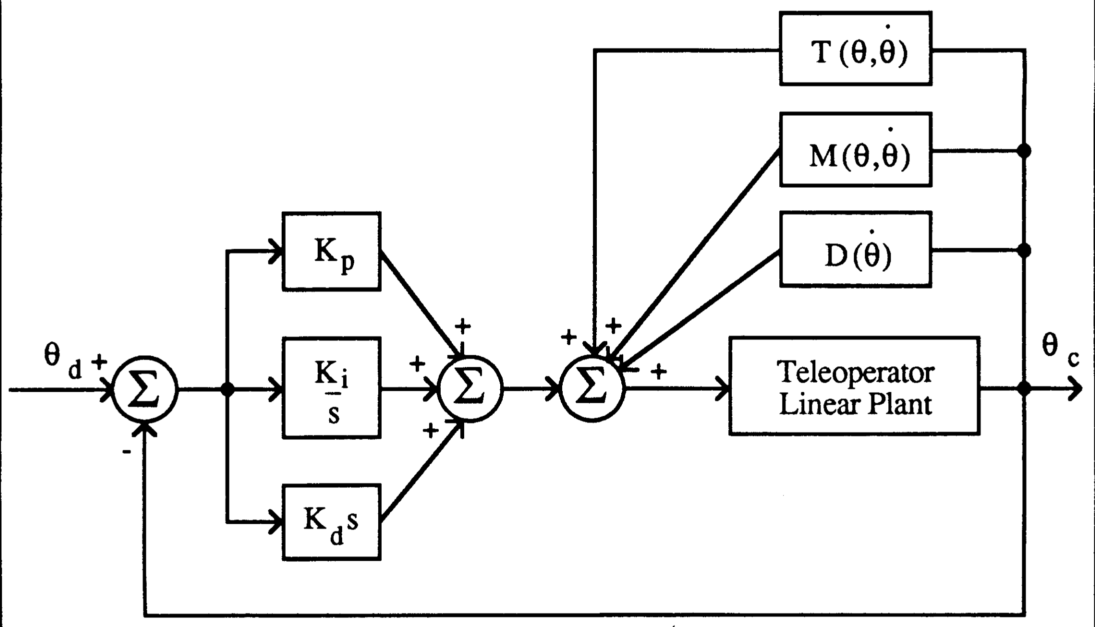
Figure 3.0.1 Pool and Teleoperator Coordinate Systems

3.0 Theory

The purpose of the closed-loop attitude controller was to accurately maintain a desired orientation in inertial roll and pitch and to hold the rate of rotation to zero when no rotation command was given. Because there was no way to measure the yaw orientation angle, the only direct control of rotations about the teleoperator's $\hat{z}$ axis consisted of rate feedback from angular rate sensors to prevent excessive yaw rotation. Figure 3.0.1 shows the definition of the robot axis and the related rotation angles. A proportional depth controller was also implemented to regulate the teleoperator's depth.

3.1 Theoretical PID Controller

A block diagram of the teleoperator system with a PID attitude controller is shown in Figure 3.1.1. The difficulty in implementing this attitude controller arose from the nonlinear components of the system. These components included the water drag on the outside of the teleoperator, and the applied moments due to the offset of the thrust vectors from the center of mass of the teleoperator. Another nonlinear component was contributed by the motion of the water inside of the teleoperator. All of these nonlinear effects were difficult to quantify. The decision was made to apply a PID type of attitude controller to the teleoperator. However, instead of basing it on the nonlinear equations of motion, an



T = Non-linear Thrust Coupling

D = Non-Linear Water Drag

M = Non-linear Effect of Water Motion Inside the Teleoperator

θ = Orientation State Vector

Figure 3.1.1 PID Block Diagram for the Teleoperator System

equivalent angle-axis representation of the system was used. The basic premise of this system was that a single rotation angle ϕ_e about a calculated axis $\hat{k}$ could be used to correct the orientation error. The details of this representation are discussed in the next section. Implementation of this system was much simpler and reduced the amount and complexity of the calculations involved. The success of this approach was, however, predicated on the assumption that the linearized controller could overcome the disturbance effects of the nonlinear components. This representation also encouraged thruster efficiency because the correction commands worked together to apply a net correction torque. Efficiency was desired to reduce the amount of power used and to extend the total flight time. The data used to calculate the error angle was supplied by the pressure transducers. The derivative term of the controller was based on the data provided by the 3-axis rate transducers. This system was implemented on ASTRO to prove the feasibility of this type of controller. The gains for each of the proportional, derivative, and integral components were set heuristically.

To simplify implementation on the teleoperator thruster systems, it was assumed that the desired correctional torque value was proportional to the torque produced by the discrete thruster commands. An arbitrary scaling factor (k_{scale}) was used to move the controller gains into the integer region. The best gains were chosen after experimentation and data analysis. It was also anticipated that the gains might be different for each axis because of the difference in number and efficiency of thrusters mounted parallel to each axis, and of the hydrodynamics of the teleoperator.

3.2 Equivalent Angle-Axis

The underlying idea of the equivalent angle-axis representation of the system, was that a single rotation of magnitude ϕ_e about a calculated unit vector $\hat{k}$ could achieve the desired orientation. To build up the angle-axis representation, rotation matrices were first defined. A rotation matrix relates the three principal axes of a system to another set of principal axes in terms of trigonometric functions of the rotation angle between the two. A rotation matrix can be written for a single rotation, or combined by multiplication to represent multiple rotations [4].

Successive rotations about non-fixed axes are called body axes rotations. Specific combinations of three of these rotations are called Euler angles. One typical set of Euler angles consists of ordered rotations about the $\hat{z}$, $\hat{y}$, and $\hat{x}$ axes. These are called ZYX Euler angles. The order of the rotations is very important to avoid singularities. The ZYX Euler angles were chosen to be implemented in the controller. A commonly used notation for expressing individual rotations is ROT(axis,angle). Thus the combined ZYX Euler angle rotation matrix can be expressed as

$$\text{Rotation Matrix} = \mathbf{R} = \text{ROT}(\hat{z},\alpha)\text{ROT}(\hat{y},\beta)\text{ROT}(\hat{x},\gamma) \qquad (1)$$

Note that the order of matrix multiplication occurs in the same order as the "rotations". The complete matrix is

$$\mathbf{R} = \begin{bmatrix} \cos\alpha*\cos\beta & \cos\alpha*\sin\beta*\sin\gamma-\sin\alpha*\cos\gamma & \cos\alpha*\sin\beta*\cos\gamma+\sin\alpha*\sin\gamma \\ \sin\alpha*\cos\beta & \sin\alpha*\sin\beta*\sin\gamma+\cos\alpha*\cos\gamma & \sin\alpha*\sin\beta*\cos\gamma-\cos\alpha\sin\gamma \\ -\sin\beta & -\cos\beta*\sin\gamma & \cos\beta*\cos\gamma \end{bmatrix} \qquad (2)$$

Since the yaw rotation angle α could not be measured, as previously discussed, it was set to zero. This simplifies the above rotation matrix to

$$\mathbf{R} = \begin{bmatrix} \cos\beta & \sin\beta*\sin\gamma & \sin\beta*\cos\gamma \\ 0 & \cos\gamma & -\sin\gamma \\ -\sin\beta & -\cos\beta*\sin\gamma & \cos\beta*\cos\gamma \end{bmatrix} \quad (3)$$

In order to use the equivalent angle-axis representation of the system, the current and desired orientation angles needed to be determined. For the purpose of the experiment, the desired angles were entered into the system via the control station. The current orientation angles were calculated from the data provided by the four pressure transducers mounted inside the teleoperator. The locations of the pressure gauges were shown in Figure 2.2.1. The gauges were mounted inside the robot with the pressure measuring ports pointing inward. The distances separating the gauges from each other were essentially the same, approximately 31.5 inches. Gauges 1 and 2 were parallel to the teleoperator's $\hat{y}$ axis, while gauges 3 and 4 were parallel to the $\hat{z}$ axis. The line defined by the average of gauges 3 and 4 and the average of gauges 1 and 2 was parallel to the teleoperator's $\hat{x}$ axis. The roll and pitch orientation angles were calculated in terms of the pressure gauges by using the ZYX Euler angle rotation matrix. Each column of the rotation matrix represented a principal axis in the teleoperator coordinate frame. Each row of the rotation matrix represented a principal axis in the pool coordinate frame. The pressure or depth differences between each of the gauges were related to specific terms of the rotation matrix. The depths measured by the pressure gauges were related to the $\hat{z}$ component of the pool coordinate frame in the rotation matrix. Therefore, the lines, Δp_{ij}, defined by the differences between gauges 3 and 4, 1 and 2, and the averages of 3 and 4 and 1 and 2, could be related to the $\hat{z}$ component of the appropriate body axis. Thus, from the third row of the rotation matrix:

$$\begin{aligned} {}^{P}(\hat{x}_r)_z &= -\sin\beta = C^{-1} * \Delta p_{(aver34\,-\,aver12)} \\ {}^{P}(\hat{y}_r)_z &= \cos\beta * \sin\gamma = C^{-1} * \Delta p_{12} \\ {}^{P}(\hat{z}_r)_z &= \cos\beta * \cos\gamma = C^{-1} * \Delta p_{34} \end{aligned} \quad (4)$$

Since the distances between the gauges were identical, the C^{-1} values in the equations were equal. To avoid ambiguity, the β and γ angles were found by calculating an atan2 value. Equating the $\hat{y}$ and $\hat{z}$ equations by the common C^{-1} value, yields:

$$\gamma = \operatorname{atan2}(\Delta p_{12}, \Delta p_{34}) \quad (5)$$

The value for β was found by isolating $\sin\beta$ from the $\hat{x}$ equation and $\cos\beta$ from either the $\hat{z}$ or $\hat{y}$ equations. This yields:

$$\beta = \operatorname{atan2}(-\Delta p_q, (\Delta p_{12}/\sin\gamma)) = \operatorname{atan2}(-\Delta p_q, (\Delta p_{34}/\cos\gamma)) \quad (6)$$

The use of both of these equations for ß eliminated problems associated with division by zero. The values of ß and Υ calculated from current pressure gauge readings represented the current orientation angles. It should be noted that due to the nature of the calculations, the roll orientation angles ranged from ±180° and the pitch angles, which were calculated as a result of the roll angle, ranged from ±90°.

The next step was to use these angles to calculate a matrix representative of the rotation away from the desired orientation. If $\mathbf{R}_{\text{desired}}$ represents the desired orientation matrix and $\mathbf{R}_{\text{current}}$ represents the teleoperator's current orientation, the correctional matrix can be expressed as:

$$\mathbf{R}_{\text{desired}} * \mathbf{R}_{\text{correction}} = \mathbf{R}_{\text{current}} \tag{7}$$

$$\mathbf{R}_{\text{correction}} = \mathbf{R}_{\text{desired}}^{-1} * \mathbf{R}_{\text{current}} = \mathbf{R}_{\text{desired}}^{T} * \mathbf{R}_{\text{current}} \tag{8}$$

In general, the transpose of a rotation matrix is equivalent to the inverse of the rotation matrix because the matrices are orthonormal. Appendix C contains the matrix calculations that result in the correctional matrix, $\mathbf{R}_{\text{correction}}$, as a function of the desired orientation angles $\Upsilon_{\text{desired}}$ and $ß_{\text{desired}}$ and the current angles $\Upsilon_{\text{current}}$ and $ß_{\text{current}}$. Shorthand notations of these will be Υ_d, $ß_d$, Υ_c, and $ß_c$. The matrix, $\mathbf{R}_{\text{correction}}$ (or $\mathbf{R}_c$), looks like

$$\mathbf{R}_c = \begin{bmatrix} c(ß_d-ß_c) & s\Upsilon_c*s(ß_c-ß_d) & c\Upsilon_c*s(ß_c-ß_d) \\ s\Upsilon_d*s(ß_d-ß_c) & s\Upsilon_c*s\Upsilon_d*c(ß_d-ß_c) + c\Upsilon_d*c\Upsilon_c & s\Upsilon_d*c\Upsilon_c*c(ß_d-ß_c) - c\Upsilon_d*s\Upsilon_c \\ c\Upsilon_d*s(ß_d-ß_c) & c\Upsilon_d*s\Upsilon_c*c(ß_d-ß_c) - s\Upsilon_d*c\Upsilon_c & c\Upsilon_d*c\Upsilon_c*c(ß_d-ß_c) + s\Upsilon_d*s\Upsilon_c \end{bmatrix} \tag{9}$$

where c = cosine and s = sine.

A rotation matrix for a single rotation ϕ_e about an arbitrary axis $\hat{k}$ is

$$\text{Rot}(\hat{k},\theta) = \text{Rot}(\begin{Bmatrix} k_x \\ k_y \\ k_z \end{Bmatrix},\theta) =$$

$$\begin{bmatrix} k_x*k_x*v\theta+c\theta & k_x*k_y*v\theta-k_zs\theta & k_x*k_z*v\theta+k_ys\theta \\ k_x*k_y*v\theta+k_zs\theta & k_y*k_yv\theta+c\theta & k_y*k_z*v\theta-k_xs\theta \\ k_x*k_y*v\theta-k_ys\theta & k_y*k_z*v\theta+k_xs\theta & k_z*k_z*v\theta+c\theta \end{bmatrix} \tag{10}$$

where c = cosine, s = sine, v = versine = 1 - cosθ. This matrix would reduce to the more familiar rotation matrix previously discussed if $\hat{k}$ were one of the principal axes of the system. The derivation of this matrix can be found in various texts [4,5]. For the purposes of the closed-loop attitude controller, the inverse result was desired, i.e. the rotation angle ϕ_e and axis $\hat{k}$ needed to be found [5]. If the rotation matrix is rewritten so that each component is represented by r_{ij}, where i is the row number and j is the column number, the matrix looks like

$$\mathrm{Rot}(\hat{k},\theta) = \begin{bmatrix} r_{11} & r_{12} & r_{13} \\ r_{21} & r_{22} & r_{23} \\ r_{31} & r_{32} & r_{33} \end{bmatrix} \tag{11}$$

The rotation angle ϕ_e can then be written as:

$$\phi_e = \mathrm{acos}((r_{11} + r_{22} + r_{33} - 1.0) / 2.0) \tag{12}$$

The axis of rotation $\hat{k}$ (a unit vector) is written as:

$$\hat{k} = \begin{Bmatrix} k_x \\ k_y \\ k_z \end{Bmatrix} = \frac{1}{(2 * \sin\phi_e)} * \begin{Bmatrix} r_{32} - r_{23} \\ r_{13} - r_{31} \\ r_{21} - r_{12} \end{Bmatrix} \tag{13}$$

Substituting in actual values for r_{ij} from $\mathbf{R}_{\text{correction}}$ and simplifying, yields

$$\phi_e = \mathrm{acos}(\ (\cos(\beta_d-\beta_c)*(1.0+\cos(\gamma_d-\gamma_c)) + \cos(\gamma_d-\gamma_c) - 1.0) / 2.0 \tag{14}$$

$$\hat{k} = \begin{Bmatrix} k_x \\ k_y \\ k_z \end{Bmatrix} = \frac{1}{(2 * \sin\phi_e)} * \begin{Bmatrix} -\sin(\gamma_d-\gamma_c)*(\cos(\gamma_d-\gamma_c)+1.0) \\ -\sin(\beta_d-\beta_c)*(\cos\gamma_d + \cos\gamma_c) \\ \sin(\beta_d-\beta_c)*(\sin\gamma_d+\sin\gamma_c) \end{Bmatrix} \tag{15}$$

Implementing these equations in the attitude controller was fairly straightforward. The simplest way to obtain an output torque was to command a rotation or combination of rotations. Since the onboard software was configured to calculate the thruster commands from a roll, pitch, or yaw rotation command, the attitude controller commands also needed to be put in this form. Each component of the $\hat{k}$ vector corresponded to one of the teleoperator's principal axes. Thus the desired net rotation ϕ_e about the vector $\hat{k}$ was achieved by multiplying the error angle by each weighted component of the $\hat{k}$ vector. This results in rotation commands equal to

$$\begin{aligned} \text{roll} &= \phi_e * k_x \\ \text{pitch} &= \phi_e * k_y \\ \text{yaw} &= \phi_e * k_z \end{aligned} \tag{16}$$

Although the attitude controller was only attempting to control the roll and pitch orientation angles, the correctional rotation included a yaw component as well. When applied together, these rotational components yielded the single desired correctional rotation.

3.3 PD Controller

The single rotation ϕ_e about the calculated axis $\hat{k}$ that could correct the orientation of the teleoperator was now known. The next step was to incorporate this knowledge into a proportional controller.

The proportional control equation is

$$\tau = K_p * K_{pscale} * e_\phi \qquad (17)$$

where τ is the correctional torque vector, K_p is the diagonal proportional gain matrix, e_ϕ is the vector of orientation errors, and K_{pscale} is the diagonal scaling matrix that moves the values in the gain matrix into the integer region. However, instead of simply using the error between the desired and current orientation angles ($e_\phi = \phi_{desired} - \phi_{current}$), the correctional angle ϕ_e, weighted by the $\hat{k}$ vector, from the equivalent angle-axis representation was used. The resulting proportional control equation becomes

$$\tau_c = \begin{Bmatrix} \tau_r \\ \tau_p \\ \tau_y \end{Bmatrix} = K_p * K_{pscale} * \phi e * \begin{Bmatrix} k_x \\ k_y \\ k_z \end{Bmatrix} \qquad (18)$$

Since simple proportional control is rarely adequate for accurate closed-loop control, a derivative feedback term was also included. The control equation then becomes

$$\tau_c = \begin{Bmatrix} \tau_r \\ \tau_p \\ \tau_y \end{Bmatrix} = K_p * K_{pscale} * \phi e * \begin{Bmatrix} k_x \\ k_y \\ k_z \end{Bmatrix} + K_d * K_{dscale} * \begin{Bmatrix} \dot{\gamma} \\ \dot{\beta} \\ \dot{\alpha} \end{Bmatrix} \qquad (19)$$

where K_d is the diagonal derivative gain matrix and K_{dscale} is the diagonal scaling matrix that moves the derivative gains into the integer region. The rate term was determined by the rate of change of each of the orientation angles because the desired rotation rate was zero. The rate of change of each of the orientation angles was provided directly by the three-axis rate transducers installed in the teleoperator.

To simplify the control equations used in the flight software, the gains were chosen to make the final output of the controller in the same form as regular rotational thruster commands. This necessitated the use of a dividing factor to restrict the magnitude of the correctional commands to the range of regular hand controller commands (0 to 15). For both the proportional and the derivative gains, the maximum allowable error before correction (the deadband) and the error at which the maximum thrust would be applied were used to calculate preliminary gain values. The initial deadbands were chosen to be 5° for the proportional term and 5°/second for the rate feedback term. The errors at which the maximum correctional thrust would be applied were chosen as 45° and 45°/s. These values were then adjusted according to the results shown by the data gathered during testing. Because the yaw orientation could not be directly controlled and a side effect of the closed-loop controller was a small induced yaw rotation, it was anticipated that the derivative gain for the yaw feedback would be significantly higher than the gains for the other two rotations.

3.4 PID Controller

Initially, only a PD controller was tested. However, data gathered as a result of testing revealed the presence of a constant offset angle from the desired orientation in each of the axes. A detailed discussion of this data will be presented in the Data and Analysis Section 5.1. Therefore, to improve the system performance, an integral term was added. The equation for the correctional torque then became:

$$\tau_c = K_p * K_{pscale} * \text{ø}e * \begin{Bmatrix} k_x \\ k_y \\ k_z \end{Bmatrix} + K_d * K_{dscale} * \begin{Bmatrix} \dot{\gamma} \\ \dot{\beta} \\ \dot{\alpha} \end{Bmatrix} + K_I * K_{iscale} * \int (e_{\text{ø}} * \Delta t) \quad (20)$$

where K_I is the diagonal integral gain matrix and K_{iscale} is the diagonal scaling matrix that moves the integral gains into the integer region. The error was taken to be the difference between the desired orientation angle and the current orientation angle for each axis ($e = \text{ø}_{desired} - \text{ø}_{current}$). Because the orientation about the $\hat{z}$ axis (yaw angle) could not be specified or measured, the integral term was only calculated for the roll and pitch orientations. The change in time, Δt, is the time between each sample or correctional calculation. This cycle time was measured in the laboratory prior to data acquisition by utilizing the computer system clock. The sample time found for the flight software with complete closed-loop control was 0.40 seconds. For faster program execution and ease of calculation the integral was simplified to a summation. The correctional torque equation then becomes:

$$\tau_c = K_p * K_{pscale} * \text{ø}e * \begin{Bmatrix} k_x \\ k_y \\ k_z \end{Bmatrix} + K_d * K_{dscale} * \begin{Bmatrix} \dot{\gamma} \\ \dot{\beta} \\ \dot{\alpha} \end{Bmatrix} + K_I * K_{iscale} * \sum_{t=0}^{t} \left(\begin{Bmatrix} e_{\gamma} \\ e_{\beta} \\ 0 \end{Bmatrix} * \Delta t \right)$$

(21)

The final values chosen along the diagonals for the scaling factor matrices were 10 for each axis in the proportional term of the controller, 250 for each axis in the derivative term, and 60 for the roll and pitch terms of the integral component. The scaling factor for the yaw term of the integral component was zero because the orientation angles could not be measured.

3.5 Depth Controller

Another observation made while testing the initial PD controller was that the applied correctional torques for the orientations also affected the depth of the teleoperator. In other words, thruster commands intended to correct orientation also had the side effect of propelling the teleoperator up and down in the pool. Since the pressure gauges measured depth, the average of the gauge readings provided an estimate of the depth of the center of the teleoperator. Because the depth of the robot at any given time was the only information available, a simple proportional depth controller was added to the closed-loop system. The proportional control equation is

$$F_{depth} = K_{depth} * K_{scale} * e_{depth} \tag{22}$$

where the error in depth is the preset or desired average pressure gauge value minus the current average pressure gauge value. The pressure gauge values were used in the integer (bit) format rather than the depth format (floating point) to simplify calculations and decrease processing time.

The z-axis of the teleoperator did not always coincide with the z-axis of the pool. Thus, a rotation matrix reflecting the current orientation with respect to the pool coordinate system, specifically the $\hat{z}$ (pool) axis, was needed. This matrix is the same as the rotation matrix originally calculated for the attitude equations, and is restated here as

$$\begin{bmatrix} \cos\beta & \sin\beta*\sin\gamma & \sin\beta*\cos\gamma \\ 0 & \cos\gamma & -\sin\gamma \\ -\sin\beta & -\cos\beta*\sin\gamma & \cos\beta*\cos\gamma \end{bmatrix}$$

The components of this matrix that show the relationship between the teleoperator's axes and the pool's $\hat{z}$ axis are the components in the third row. Thus the vector that is needed is

$$\hat{z}_{pool} = \begin{Bmatrix} -\sin\beta \\ \cos\beta*\sin\gamma \\ \cos\beta*\cos\gamma \end{Bmatrix} \tag{23}$$

The implemented depth controller is therefore

$$F_{depth} = \begin{Bmatrix} F_x \\ F_y \\ F_z \end{Bmatrix} = K_{depth} * K_{scale} * e_{depth} * \begin{Bmatrix} -\sin\beta \\ \cos\beta*\sin\gamma \\ \cos\beta*\cos\gamma \end{Bmatrix} \tag{24}$$

where F_{depth} is the force vector required to correct the depth error, K_{depth} is the diagonal proportional depth gain matrix, K_{scale} is the diagonal scaling factor matrix used to move the depth gain into the integer region (the values along the diagonal are equal to 75), e_{depth} is the error in depth, and the components of the $\hat{z}_{pool}$ vector are used to weight the error and thus the correctional force.

The initial gains for the depth controller were chosen so that the deadband was approximately 0.25 feet and the depth error that triggered the maximum thrust value for correction was approximately 3.0 feet. It was anticipated that these values would change after data was taken and analyzed.

<u>4.0 Test Set-Up</u>

The equipment used in the experiment were SECS and ASTRO. Communication between the control station and the teleoperator was performed via a fiberoptic link. The communications software shell, PIVECS, was used to transfer information up and down the link. Color video signals were sent to the monitor on SECS via another cable. The overall test set-up is shown in Figure 4.0.1.

In order to obtain accurate sensor values the pressure and rate transducers first needed to be zeroed. This was done by taking a set of readings that were used as adjustment values for the data taken during testing. These readings were taken on the pool deck before the teleoperator was put into the water and between tests when it was underwater. Zeroing the values while in the water required holding the robot as still as possible for the rate readings, and leveling each pressure gauge with respect to the others for each gauge reading. Small levels were placed on three faces of the teleoperator (the top, port side, and 45° between them) to assist with this. The information provided by the rate sensors and pressure gauges were read into the onboard computer through a 12-bit analog-to-digital converter. These numbers were then converted into a usable form and utilized in the closed-loop controller calculations. The thruster commands were sent out of the onboard computer in the form of a 4-bit digital command. Interface circuitry converted those commands into a pulse-width modulated signal which was then sent to the appropriate thruster. Amplification and current switching was performed in the power modules attached to each thruster unit.

The closed-loop controller ran directly in the onboard software. The calculated contribution of each part of the attitude controller was added to the appropriate rotational thruster command. The regular hand controller commands were disabled for the bulk of the experiments in order to speed up the system execution time. The depth controller contributions were added to the thruster x, y, and z translation commands. The output command magnitude was limited to the maximum value of 15.

During the data run, all of the raw (12-bit) pressure and rate transducer values were stored on ASTRO in a large array (1000 samples). Data was continuously stored until the data acquisition was completed. At this time, the thrusters on ASTRO were disabled (main power was turned off) and the number of samples was sent to the surface computer. The pressure and rate transducer values were then sent to the computer on the surface in the same manner as regular communications, except that all of the other transmissions were overridden. When all the data was received the control station computer wrote the data into

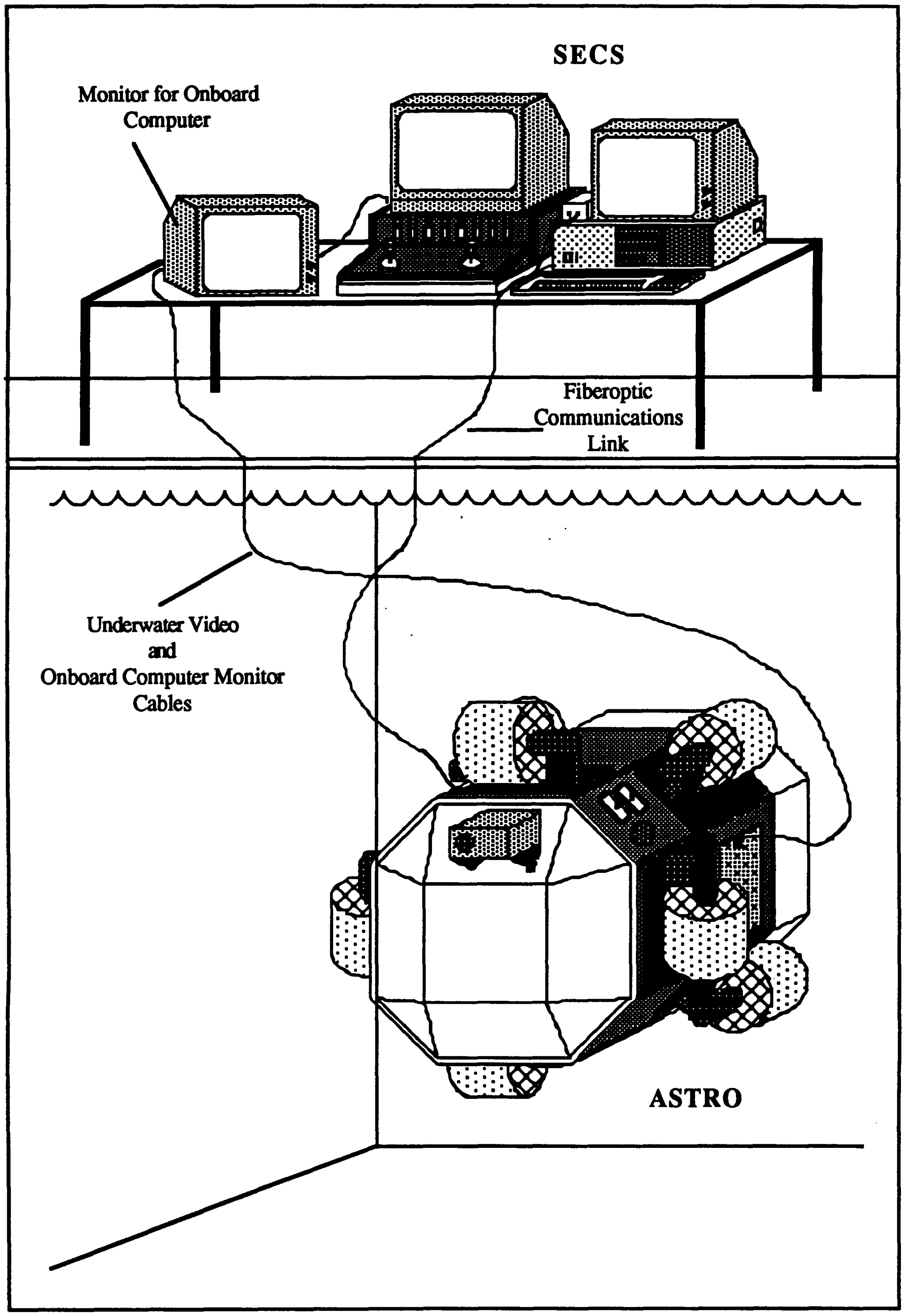

SECS
Monitor for Onboard Computer
Fiberoptic Communications Link
Underwater Video and Onboard Computer Monitor Cables
ASTRO

Figure 4.0.1 Test Set-UP: ASTRO and SECS System

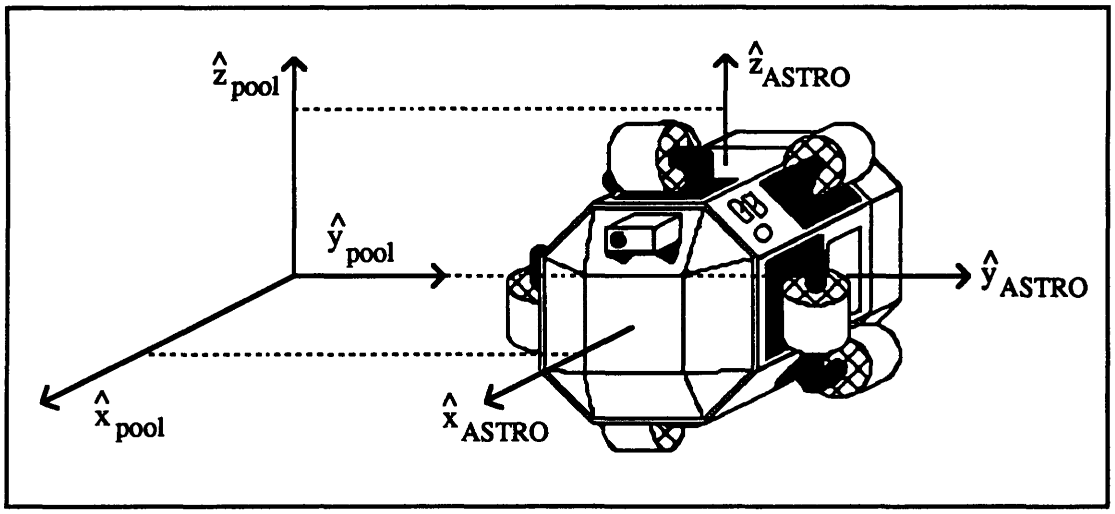
the previously specified data file along with the number of samples and the values of the desired orientation angles. The time between samples, or cycle time, on ASTRO was measured in previous experiments as 0.40 seconds. This measurement was made by using the system clock to calculate the total time passed during a data run. The length of time was then divided by the number of samples taken during the run.

5.0 Data and Analysis

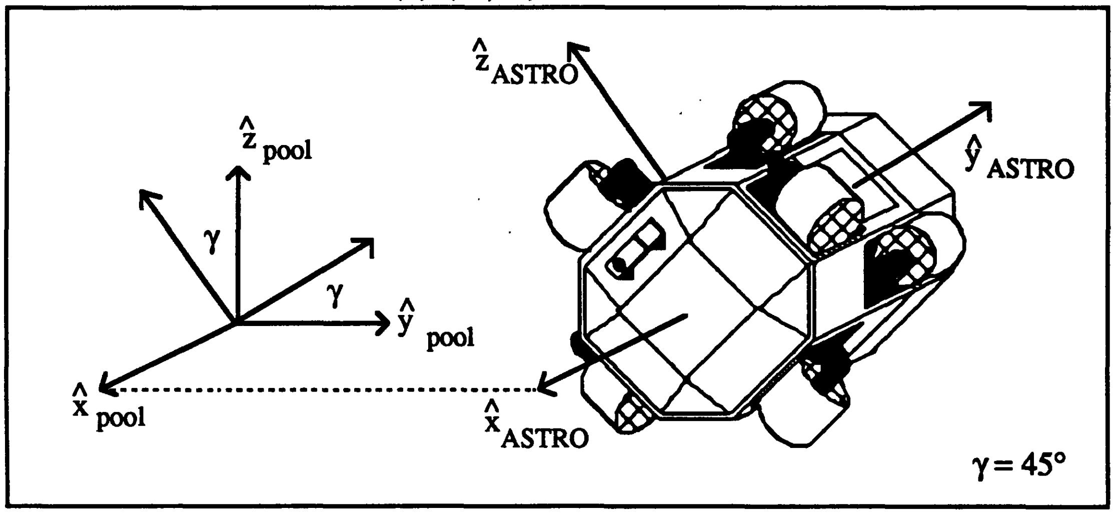
As with any controller, the desired system results should demonstrate a stable controller that is robust and can compensate for reasonable outside disturbances. The disturbances tested in the research were approximately the maximum disturbances that could be reasonably expected in this type of experimental research (i.e. those caused by collision with the pool wall, another teleoperator, or a diver). Other desired features were a quick response time, low overshoot, small or non-existent steady-state error, and no limit cycling. In addition, the minimization of uncontrolled yaw rotation, despite the fact that it could not be directly measured, was also desired.

Data files were obtained for straight regulation of (0°,0°), (45°,0°), and (0°,45°) orientation angles, where the first number was the roll angle and the second was the pitch angle. Each of these is pictured in Figure 5.0.1. Responses to step inputs and pseudo-random disturbances, as well as attempts at free-flight were also obtained. The step inputs were generated by changing the desired orientation angles. Disturbances were generated by underwater divers applying a torque to each axis singly and in combination. Disturbance files were taken by allowing the teleoperator 45 seconds to regulate (determined from previous data to be sufficient), imparting a disturbance, and then allowing approximately 60 to 90 seconds for the system to return to equilibrium. Tests to demonstrate the overall accuracy of the system were performed by specifying a specific orientation and then attempting to fly the teleoperator along a straight line translation.

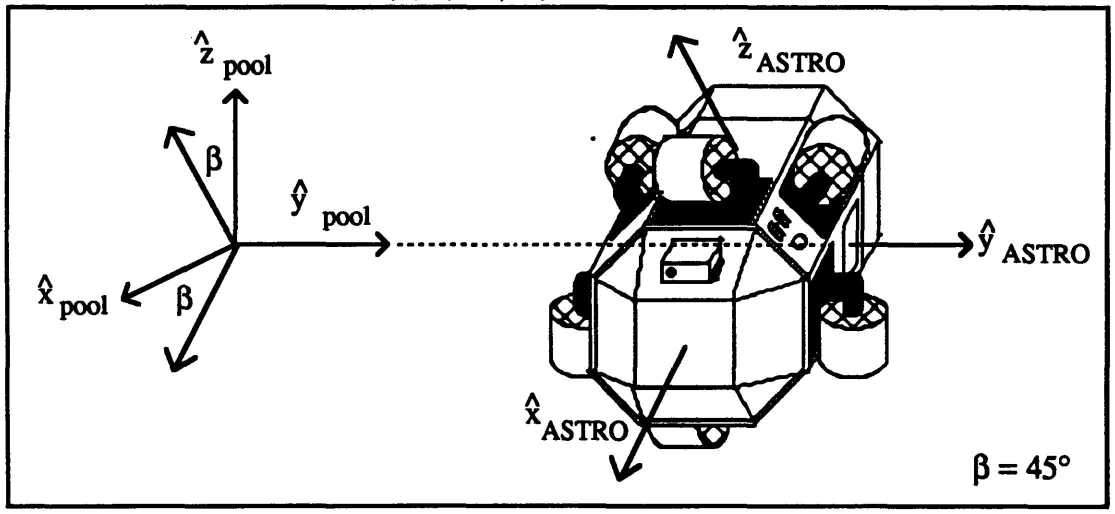
The performance of the depth controller was tested by setting the teleoperator in different attitudes and applying a depth disturbance in either direction. The depth response was also evaluated based on the data from the attitude regulation, step response, and disturbance files. The depth disturbance files contained disturbances in both directions (the force applied driving the robot deeper, allowing 60 seconds or more to settle, and another force applied driving the robot shallower). The data obtained in the experiments are presented below.



(a) (0°,0°) Orientation



(b) (45°,0°) Orientation



(c) (0°,45°) Orientation

Figure 5.0.1 Teleoperator Orientations

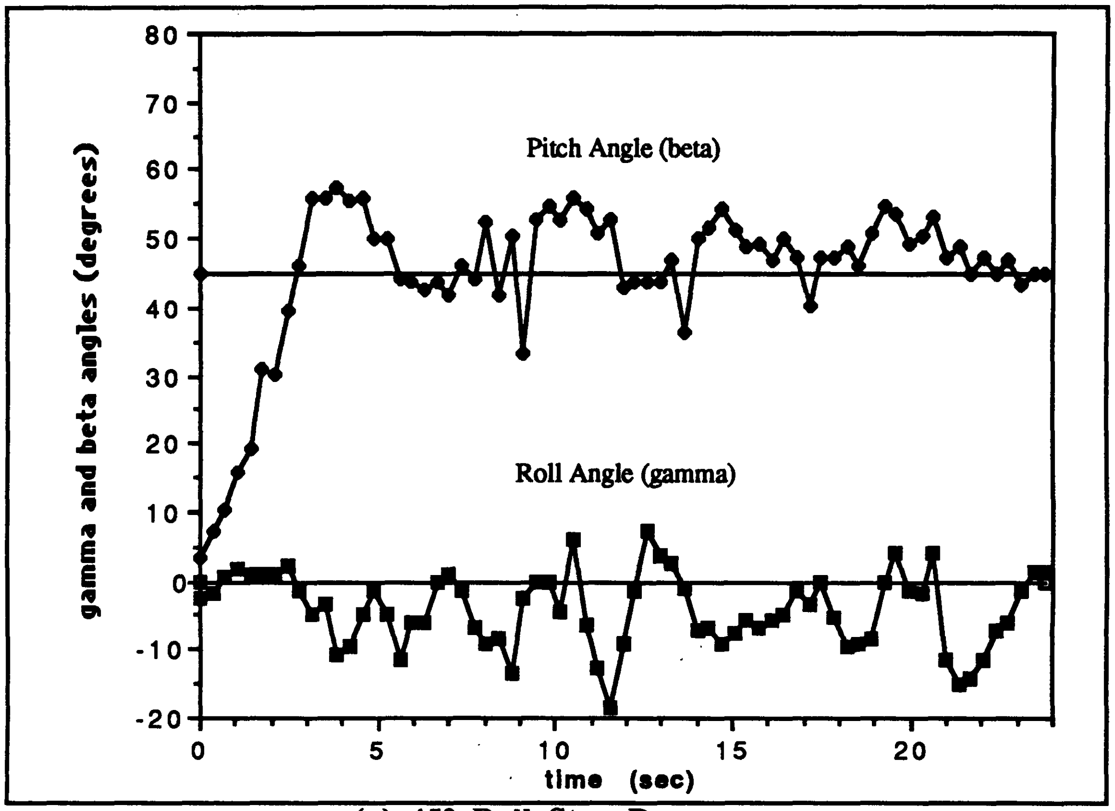
5.1 PD Controller Results

The first closed-loop attitude controller implemented was the PD controller. The tests performed with this controller were to serve several purposes. First, the gains were systematically adjusted in different combinations to give numerous data files that could be used to determine the best gains for each axis. Second, the tests were to provide qualitative data on the controller's disturbance rejection characteristics and free-flight capabilities.

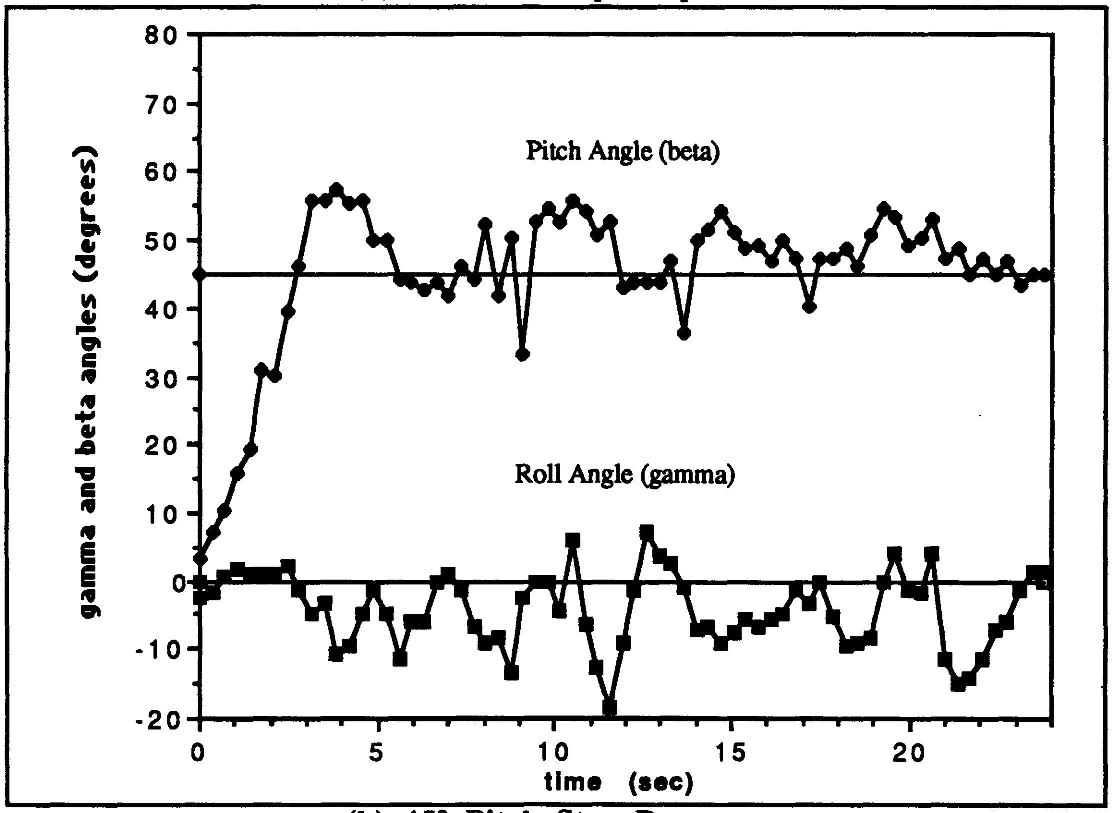
The data acquired were step response files. Initially, the teleoperator was programmed to hold an attitude of zero roll and zero pitch. A step input was obtained by changing either the desired roll or pitch angle to positive 45 degrees. This change was transmitted to the teleoperator along with an acquire data flag. The data files tended to be rather small, on the average about 45 seconds. This was due to the robot's tendency to thrust towards the bottom or surface of the pool due to incidental vertical thrust vectors generated by the attitude hold system. A range of gains (5 to 12) was identified for the proportional and derivative terms that varied from very slight control to limit cycling. During testing, observations made from the video monitor served to indicate when a limit cycle was reached. Not all combinations of gains were tested due to the shortage of time available for the initial testing, though an attempt was made to test some combinations of different gains for each axis. However, the data gathered was sufficient to indicate several necessary changes.

Representative graphs of the roll and pitch angle data for roll and pitch step inputs are shown in Figure 5.1.1. Both graphs show fairly good responses to the step input. The response time was approximately 3.0 to 5.0 seconds (time to when the response was within ~10 percent of the desired value or ± 5 degrees) with an overshoot of 5 to 12 degrees. The exact size of the overshoot was difficult to determine because of the offset error angle. The orientation angles oscillated about points slightly offset from the desired orientations by approximately 3 to 5 degrees. An examination of the graphs of the error angles calculated from the equivalent angle-axis representation, ϕ_e, in Figure 5.1.2 shows the presence of an offset error angle. The magnitude of these error angles varied from 3 to 20 degrees. These offset angles and the oscillations about them were present in all of the data files taken. However, the ones shown in the figures represent the best responses of the data acquired for the PD controller (the gains used in the graphs are 9 and 7 for the proportional and derivative controllers, respectively).

From this set of data, the best combination of gains are a proportional gain of 9 and a derivative gain of 7 for each of the roll, pitch, and yaw axes. These gains were slightly less than the ones that resulted in limit cycling. The gains corresponded to a deadband of ~1° with an error of ~17° corresponding to saturation of the correctional thrust.

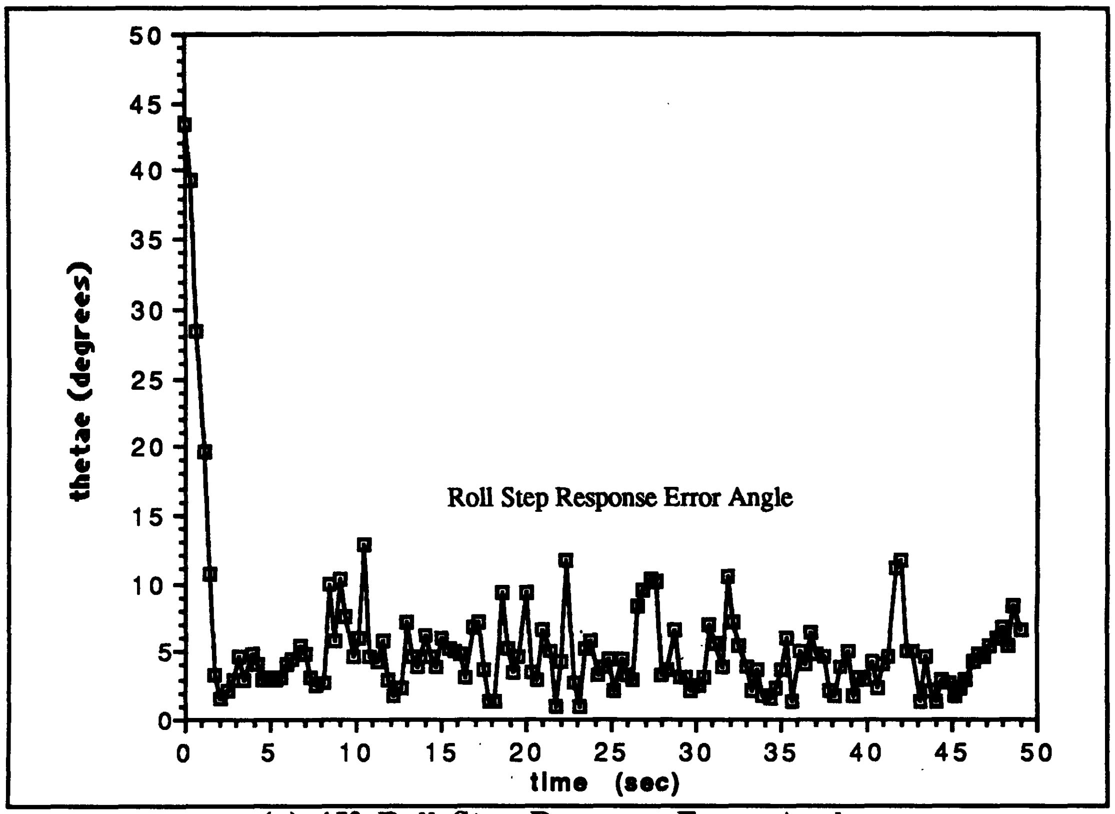


(a) 45° Roll Step Response

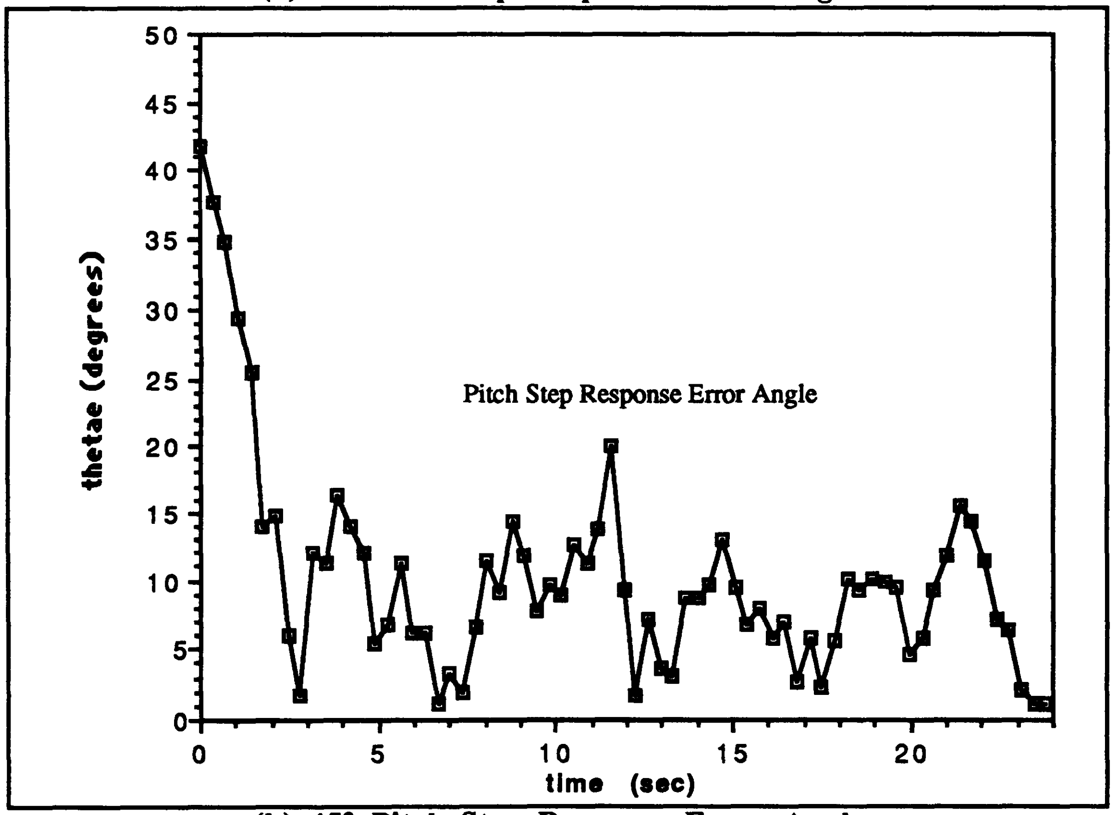


(b) 45° Pitch Step Response

Figure 5.1.1 Representative Step Response Data (PD Controller)



(a) 45° Roll Step Response Error Angle



(b) 45° Pitch Step Response Error Angle

Figure 5.1.2 Representative Error Angle Data (PD Controller)

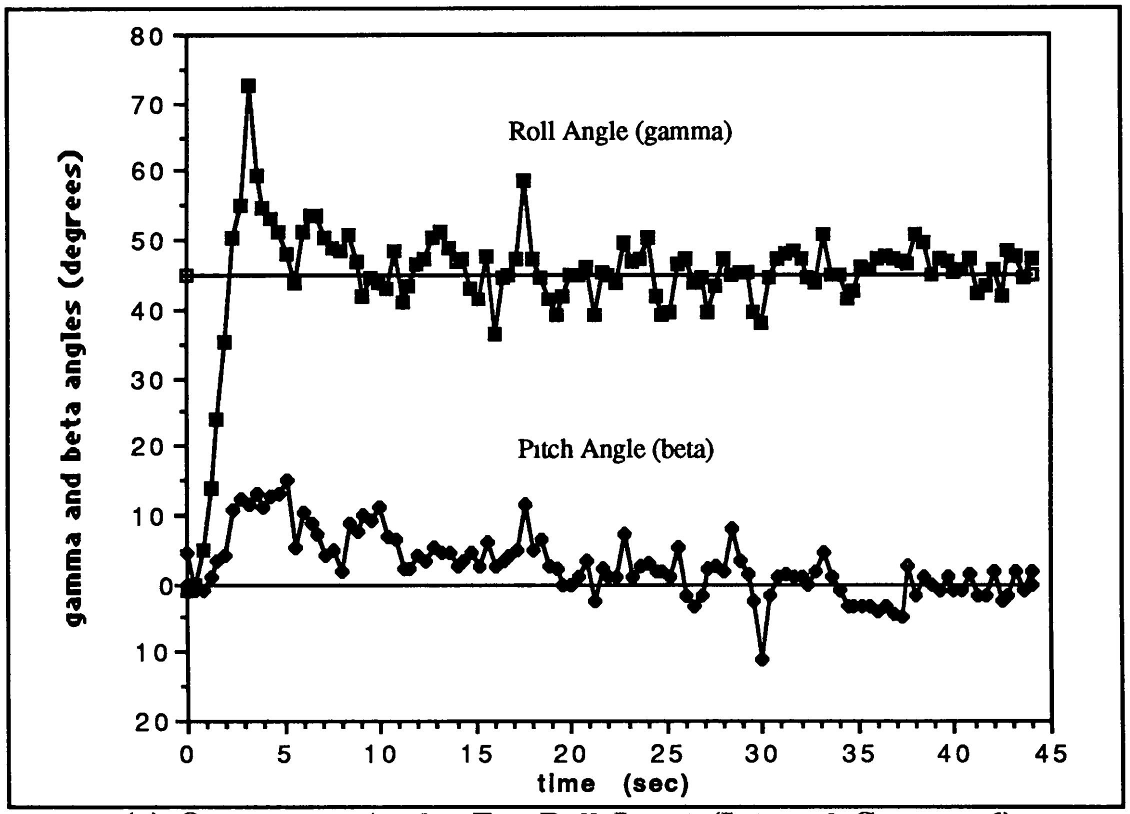
Problems with the data acquired include the relatively small sample size of each run and the errors introduced into the data by the teleoperator impacting the bottom of the pool or surfacing. Teleoperator surfacing corrupted data because when the robot was on the surface at least one thruster was not thrusting against the water. For the situations when the teleoperator hit the bottom, the bottom of the pool interfered with some attempted rotation or else imparted a disturbance force from the force of impact. There were also disturbances generated by the water jets at the sides of the pool which caused additional torques and translations.

Conclusions drawn from this preliminary data included the need for an integral term in the controller to eliminate or minimize the constant offset angles (steady-state error). In addition, a closed-loop depth controller was needed to provide the ability for longer data runs without interference with the bottom or surface of the pool. Finally, more accurate gains were obtained for the proportional and derivative terms of the controller (though these values changed slightly with the addition of the integral term).

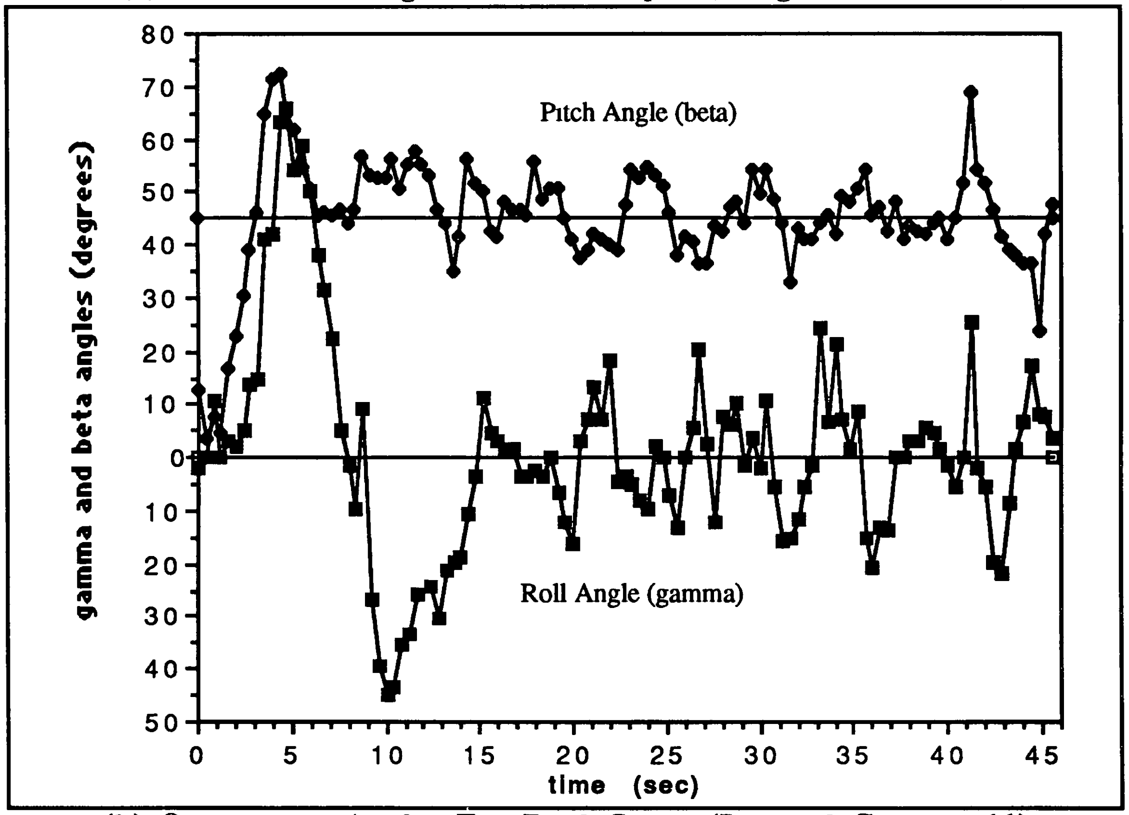
5.2 PID Controller Results

An integral term was added to the previously tested PD controller to eliminate the offset error from the desired orientation angles. The new PID controller was tested in three separate test sessions. In the first one, step responses to a separate 45° change in roll and pitch were used to determine the best integral gain. In addition, longer data files were taken. The second session was used to fine-tune all of the gains to the step responses and also to obtain pure orientation regulation files. The third test session was used to collect data during free-flight (pure x and y translations along the length of the pool) as well as for orientation disturbance responses.

In the first test session, the integral gains were systematically varied over a range of values (5 to 12). The proportional and derivative gains used during these tests were the ones previously shown to have the best response: proportional gains equal to 9 and derivative gains equal to 7. Small offset angles were still apparent in the orientation angles with integral gains of 5 in roll and 5 and 6 in pitch. Integral gains of 8 or greater seemed to cause excessive noise in the system and instability in gains greater than 10. Therefore, integral gains of 6 for the roll axis and 7 for the pitch axis were chosen. Figure 5.2.1(a) shows a graph of the orientation angles for the step response to a 45° roll command using integral gains of 6. Figure 5.2.1(b) shows the orientation angles of the step response to a 45° pitch command using integral gains of 11. The latter system was much noisier and appeared unstable.



(a) Orientation Angles For Roll Input (Integral Gains = 6)

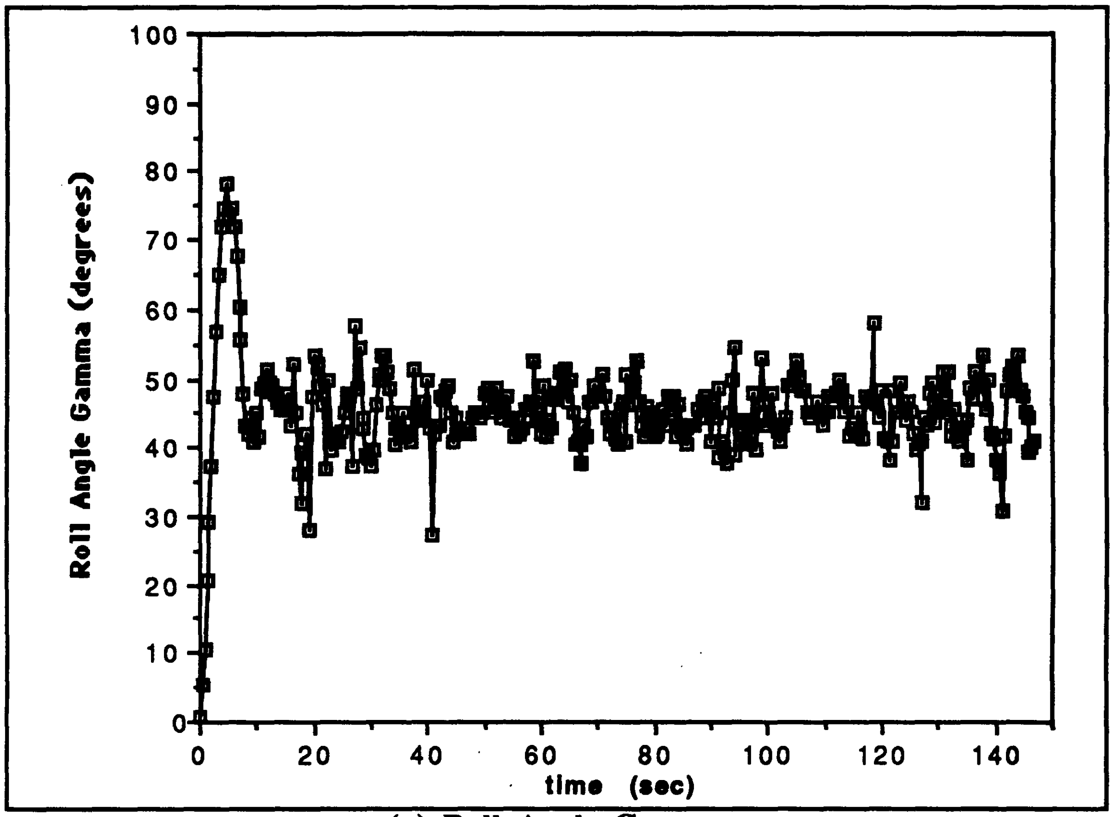


(b) Orientation Angles For Pitch Input (Integral Gains = 11)

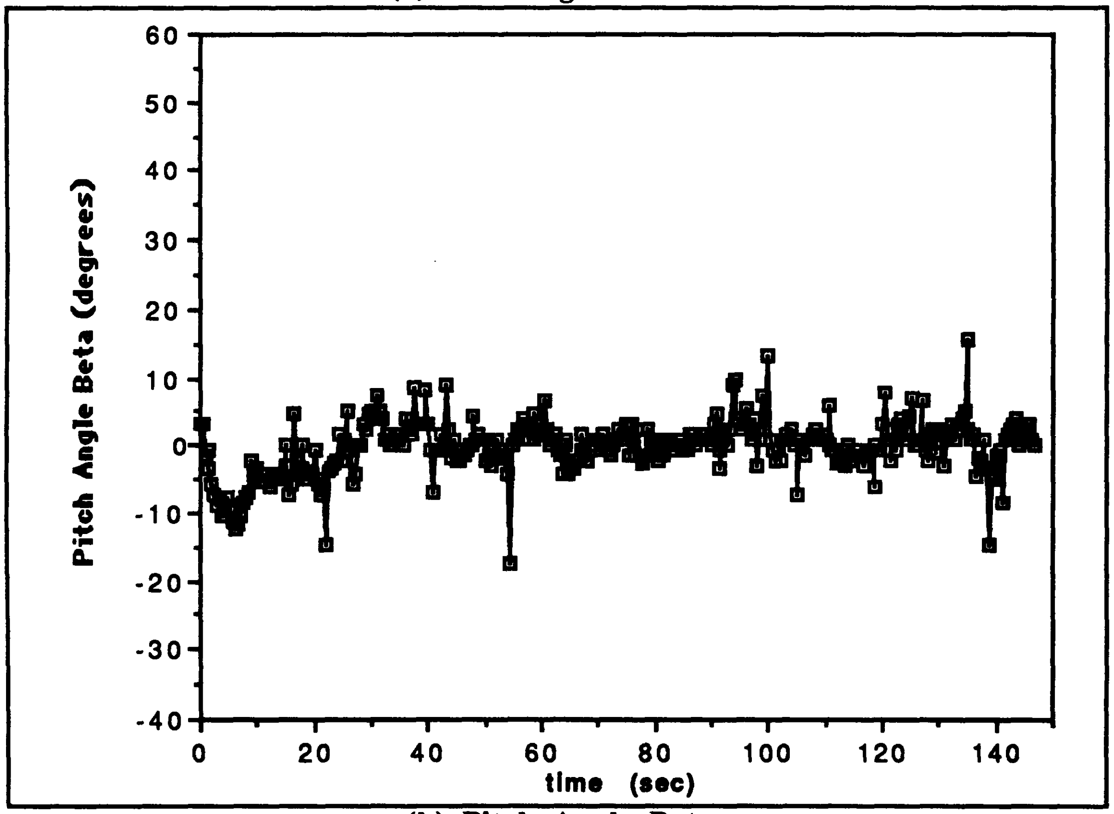
Figure 5 2 1 Sample Step Response Graphs (PID Controller)

In the second test session, all of the gains were fine tuned to the step response. In addition, pure orientation regulation files were taken. These files showed that the oscillations about the desired orientation angles were partially due to the noise in the system (noise and resultant errors are discussed in Section 5.4). Additional oscillations were due to angle "spikes" (random and large orientation angle changes) which in turn were caused by noise or spikes in the pressure transducer readings. In order to reduce the effects of the spikes on the overall performance of the system, the proportional gains were reduced to 8. With the proportional gains set to 8 and the integral gains set to 6 and 7 for the roll and pitch axes respectively, the derivative gains were varied for each axis. The best gains for the roll and pitch axes were determined to be 6 and 8 respectively. These gains provided the most steady orientation angles during regulation and for the step responses. Choosing the best gain for the yaw axis was more difficult since derivative feedback is the only control about the pool yaw axis. When the teleoperator was in the (0°,0°) orientation, high derivative feedback for the yaw axis greatly damped out unwanted yaw rotations that occurred due to the controller correctional torques. However, if this gain was too high, it affected the other two axes adversely. Data files were taken to determine the maximum acceptable derivative gain for the robot's yaw axis. Graphing the data showed that a gain of 10 minimized excessive yaw rotations while not interfering with the control of the other axes. Figure 5.2.2 shows the step response of the system to a 45° roll input with the final gains. The response time of the system was within 10 seconds. The large overshoot in the roll angle (~35°) was due in part to the magnitude of the step input. The system settled fairly quickly with an error of approximately ±5 degrees. Figure 5.2.3 shows the step response to a 45° pitch input. The response time was slightly longer, a little over 10 seconds, and the magnitude of the error oscillations were closer to 7 to 10 degrees. In retrospect, a tradeoff should be made between the response time and the overshoot. The final gains chosen for the controller favored fairly quick response time, but allowed a large overshoot. Further experimentation should investigate the step response to a system with reduced proportional and integral gains and increased derivative gains.

Data files were also taken for simple regulation of the (0°,0°), (45°,0°), and (0°,45°) orientations. These are presented in Figures 5.2.4, 5.2.5, and 5.2.6, respectively. Except for an occasional spike, the (0°,0°) orientation was easily held within ±5° of the desired values. The (45°,0°) orientation was slightly more noisy, though the desired value still ranged within ±5 degrees. The (0°,45°) orientation was also a little noisy, with the roll angle range approximately ±7° and the pitch angle range approximately ±5 degrees. Overall, it appeared that the attitude regulation was fairly consistent and that the response to a 45° step input reached the desired values in approximately 10 seconds.

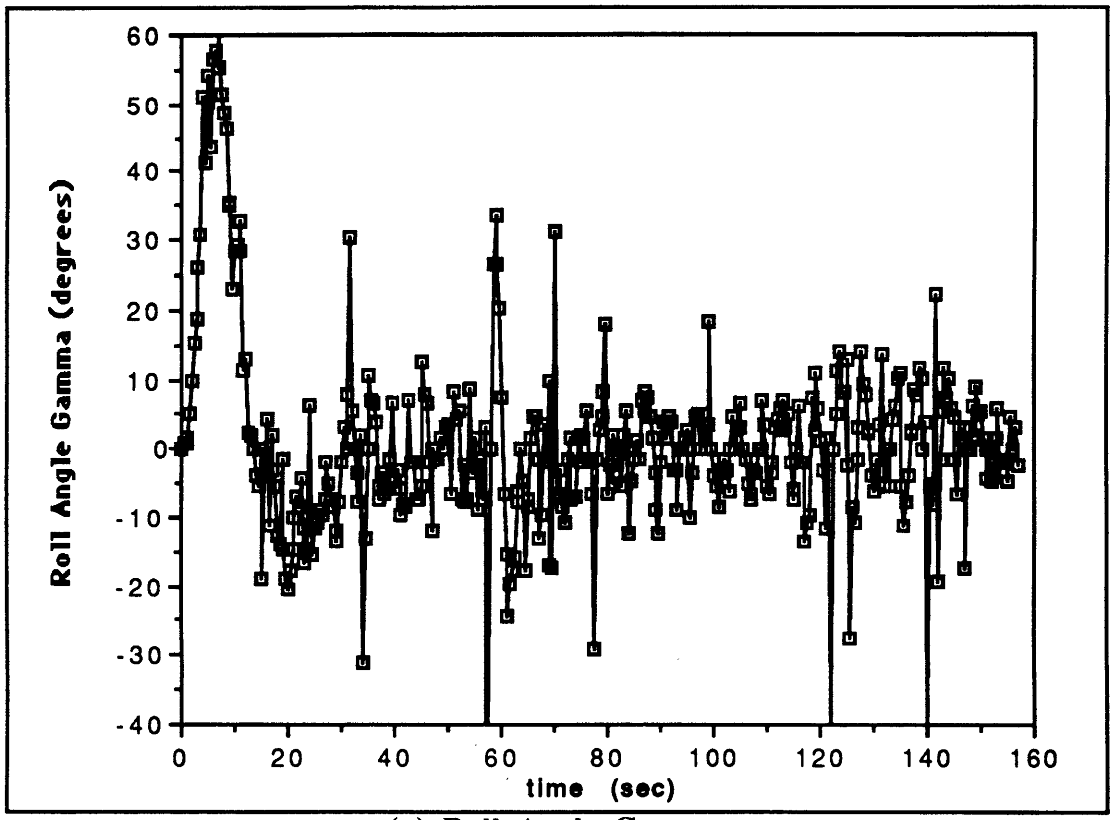


(a) Roll Angle Gamma

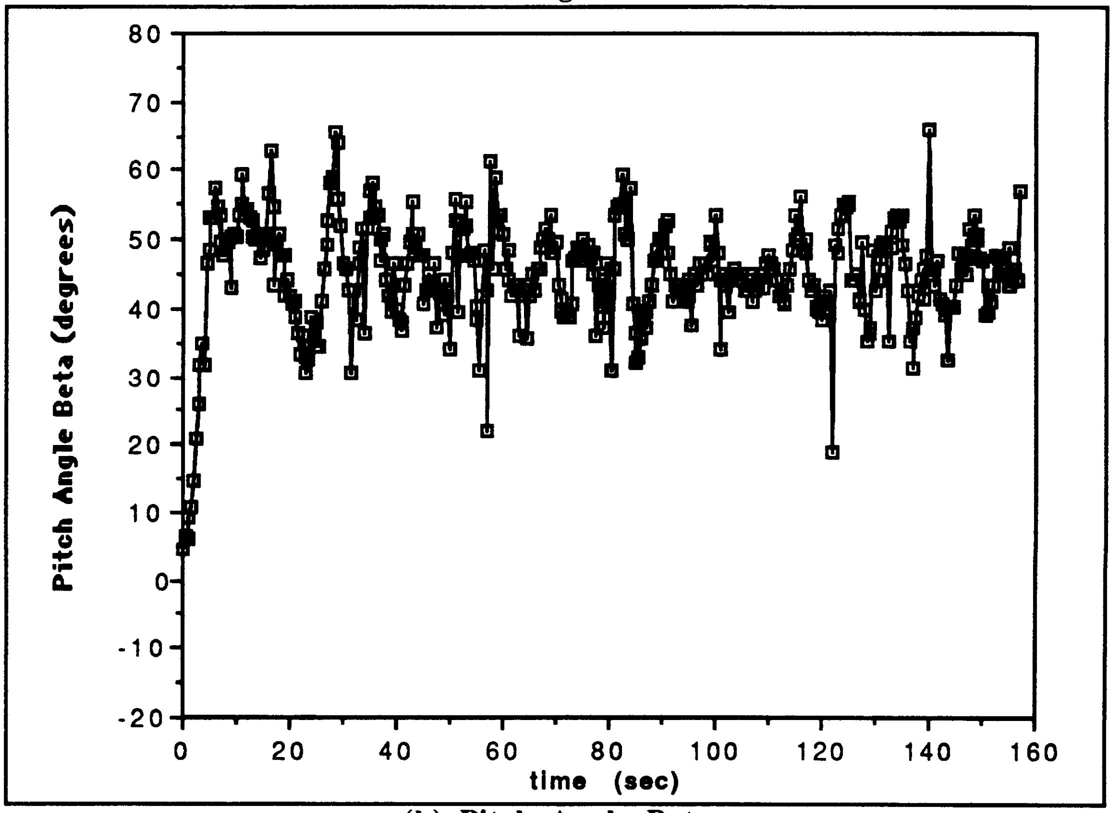


(b) Pitch Angle Beta

Figure 5.2.2 Orientation Angle Response To A 45° Roll Step Input

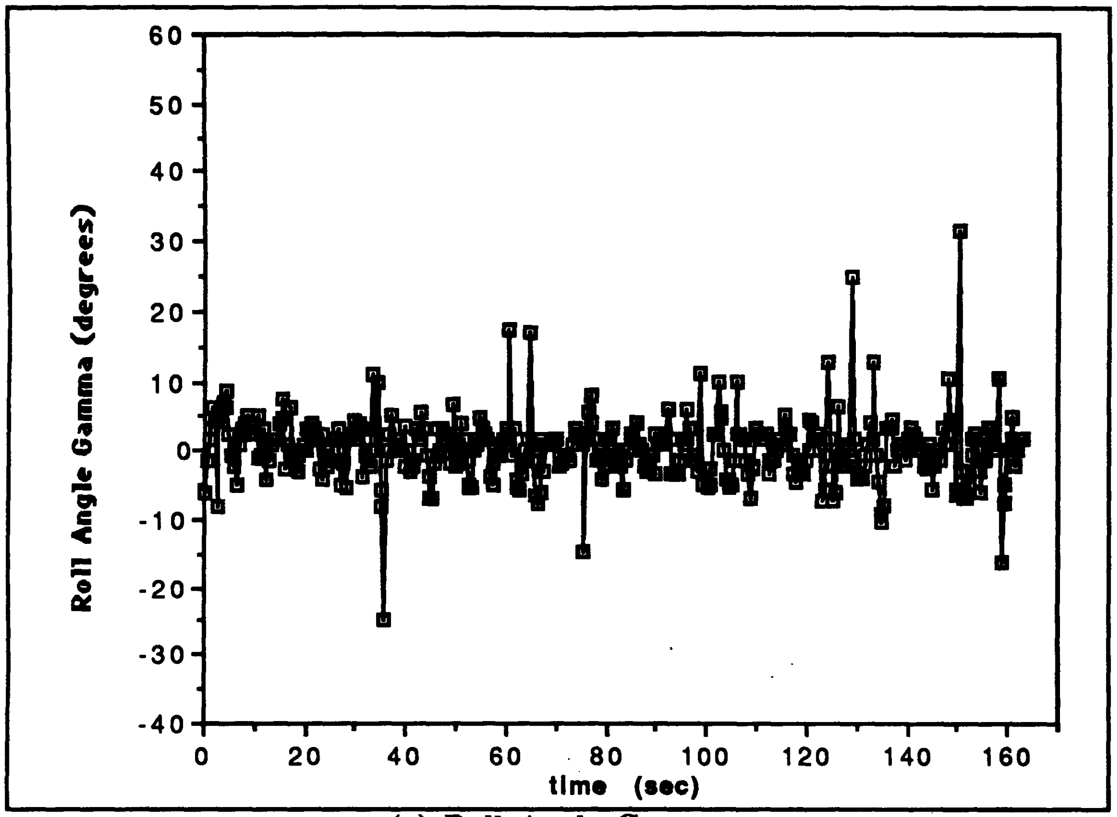


(a) Roll Angle Gamma

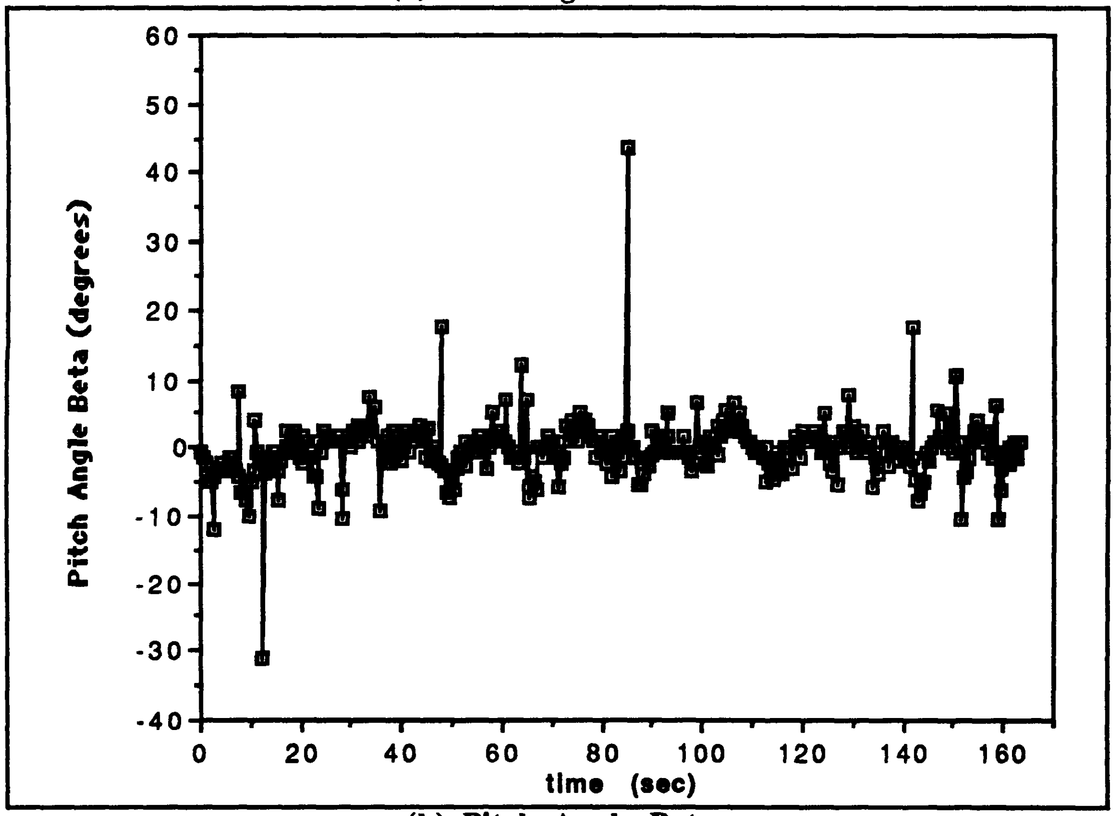


(b) Pitch Angle Beta

Figure 5.2.3 Orientation Angle Response To A 45° Pitch Step Input

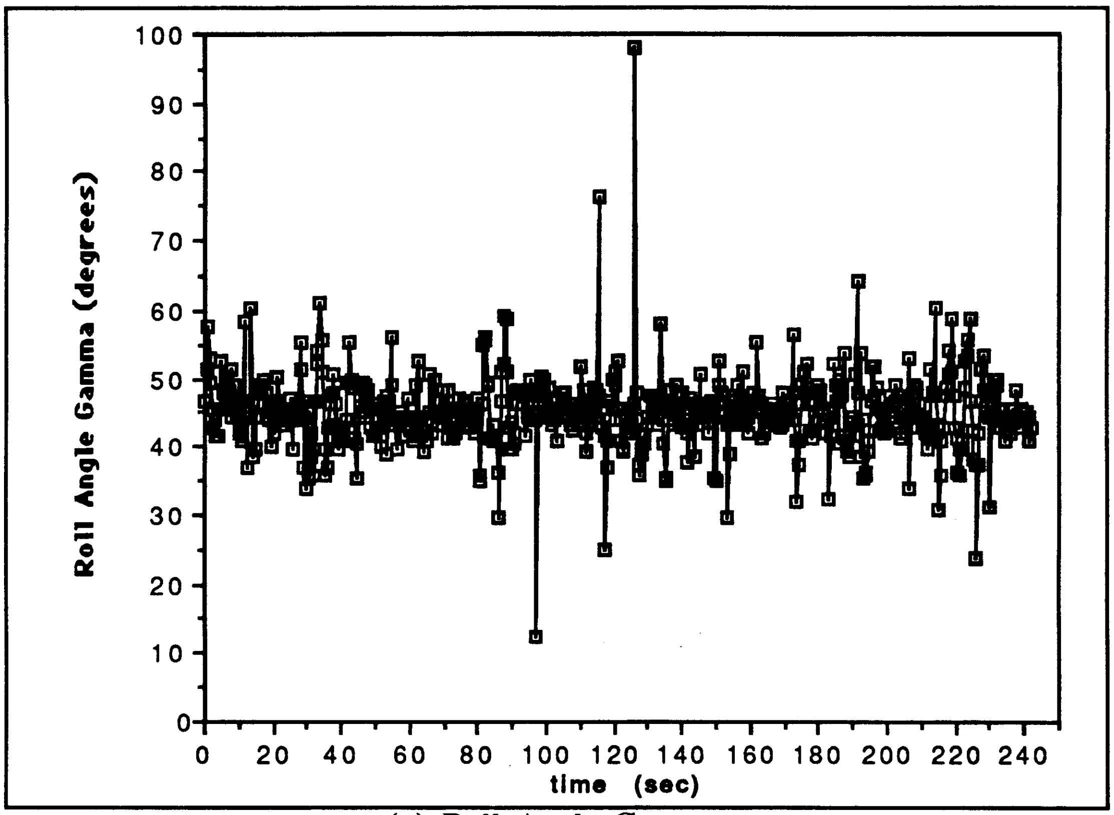


(a) Roll Angle Gamma

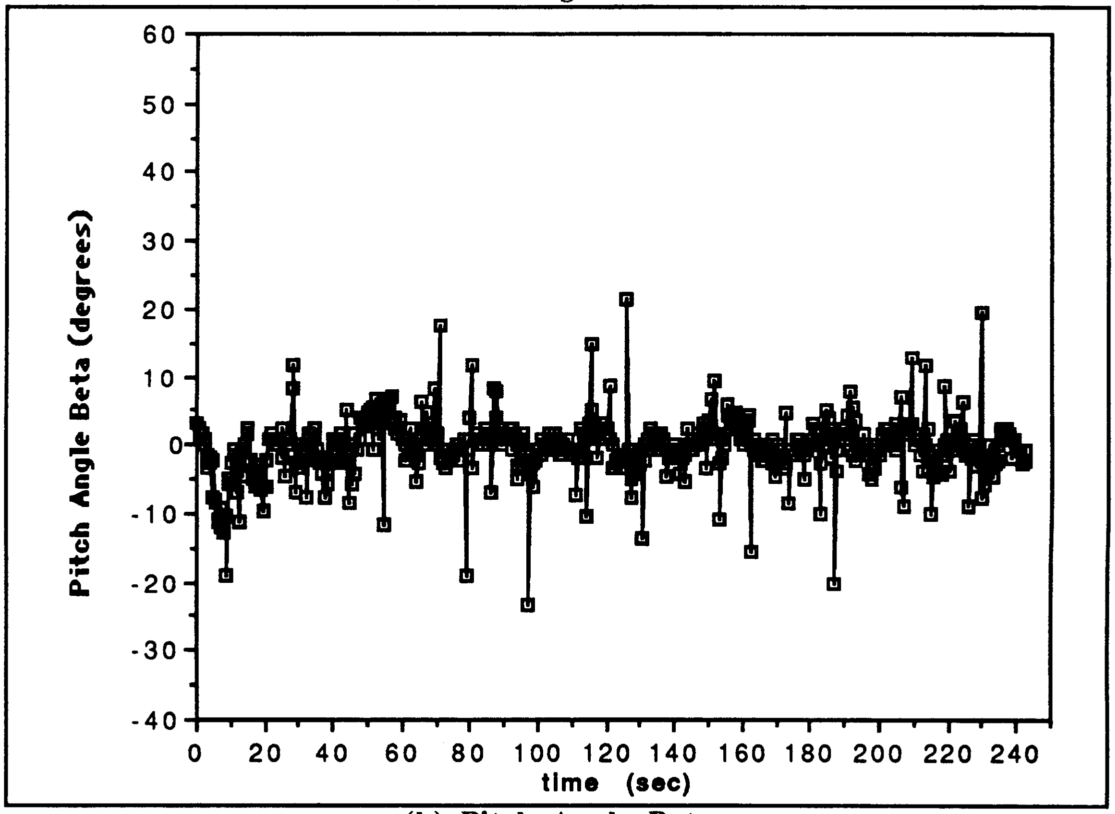


(b) Pitch Angle Beta

Figure 5.2.4 Orientation Angles During (0°,0°) Attitude Regulation

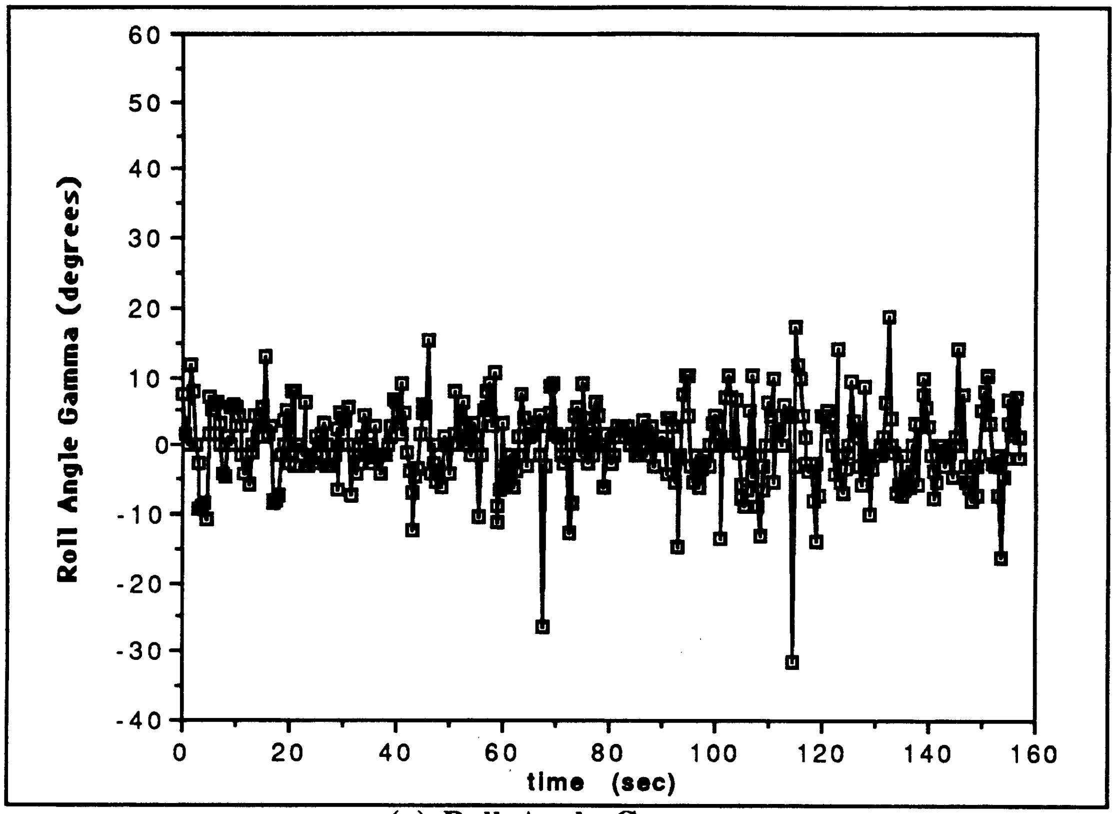


(a) Roll Angle Gamma

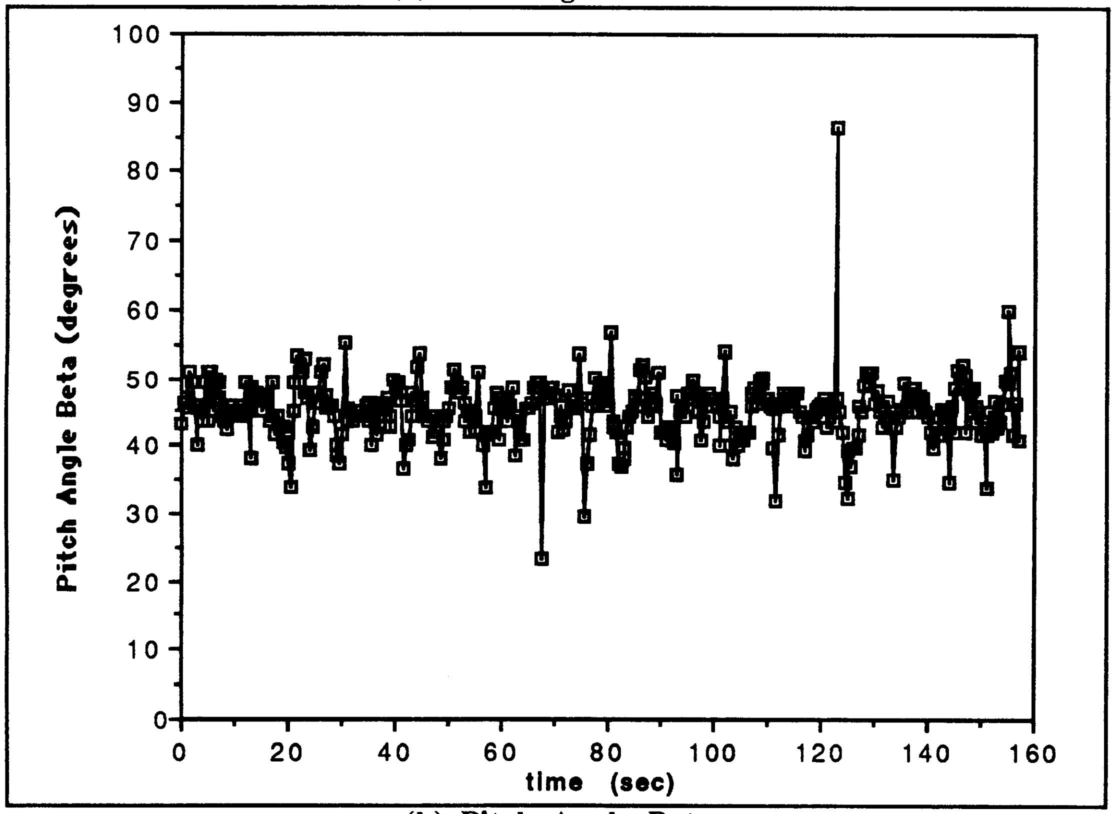


(b) Pitch Angle Beta

Figure 5.2.5 Orientation Angles During (45°,0°) Attitude Regulation



(a) Roll Angle Gamma



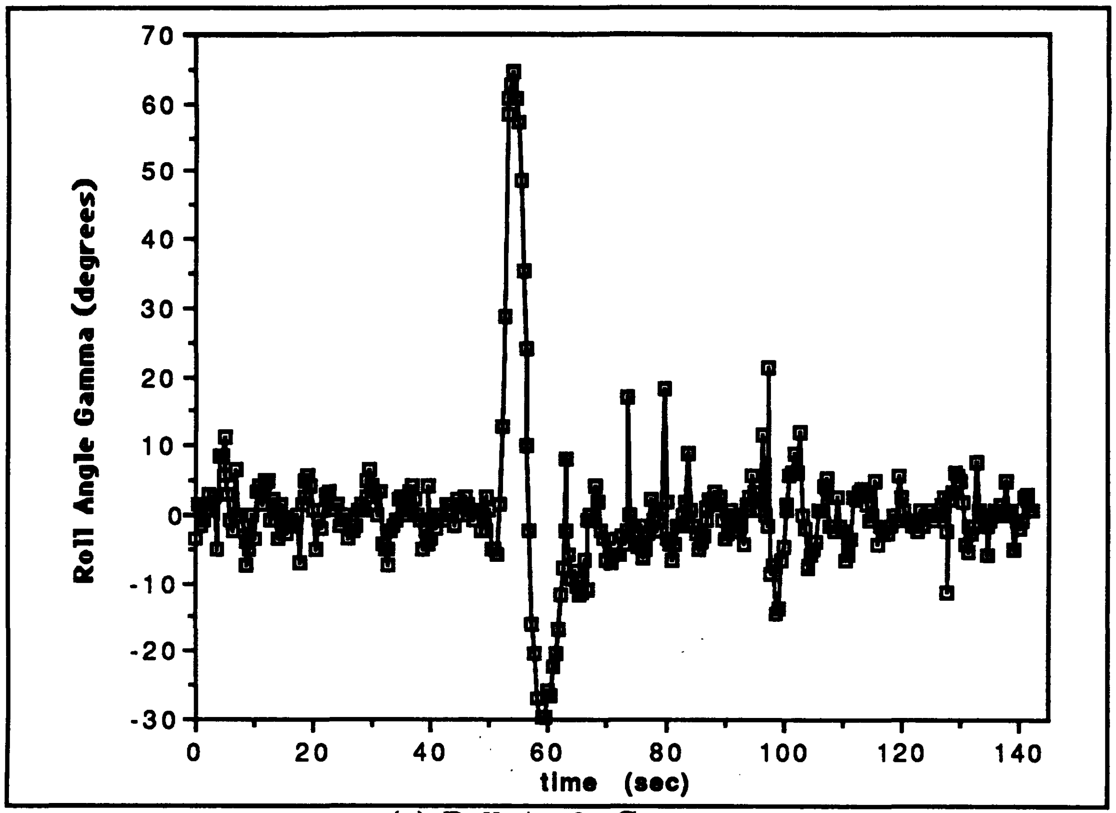
(b) Pitch Angle Beta

Figure 5.2.6 Orientation Angles During (0°,45°) Attitude Regulation

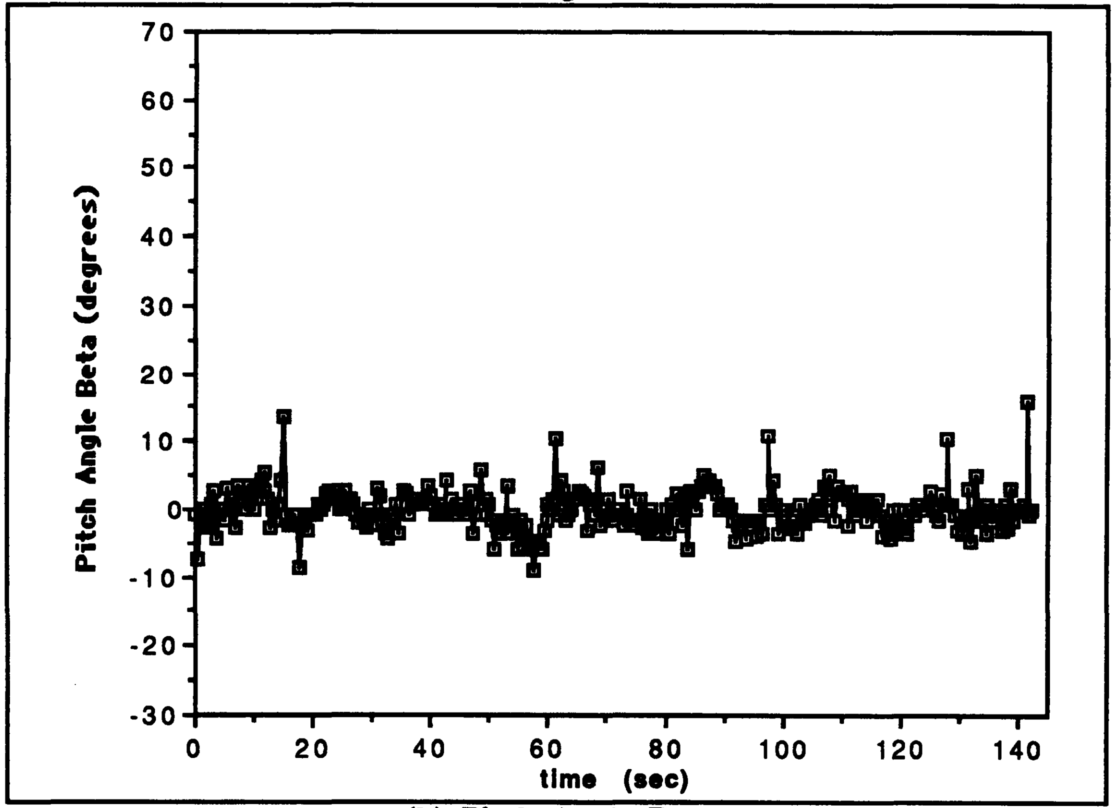
The last test session was used to examine the PID controller's response to orientation disturbances. Initially, the teleoperator was set in the (0°,0°) orientation and a pure roll disturbance was imparted (45 seconds after the data file was started to allow the robot to settle in the desired orientation). This data is shown in Figure 5.2.7. The controller responded very quickly, and the system returned to normal within about 5 seconds. Due to the large size of the disturbance, there was a fairly large overshoot. However, the overall response was fairly good. Figure 5.2.8 shows the system's response (from the (0°,0°) orientation) to a pure pitch disturbance. The response was again good, and closely resembled the step response data. Figure 5.2.9 shows the PID controller's response to a random orientation disturbance when the teleoperator was in the (0°,45°) orientation. The controller responded to the disturbance and damped it out within approximately 10 seconds. It should be noted that all of the disturbances in this section were larger than those likely to be encountered during the course of experimental research. Good system response to these disturbances guarantees good responses to disturbances encountered during normal operation.

The third test session was also used to show the effectiveness of the PID controller during free-flight translation. Before the controller was implemented, free-flight translation involved constant correction (with the rotational hand controller) in all of the rotational axes. In addition, pure translation was impossible since a y-translation resulted in a combined translation and yaw rotation, while a commanded x-translation resulted in a combined translation and pitch rotation. In the free-flight test, corrections with the rotational hand controller were allowed only for yaw rotations. Data for the pure x- and y-translations are presented in Figures 5.2.10 and 5.2.11, respectively. The teleoperator was flown up and down the length of the pool in the (0°,0°) orientation. The data shows that while there is a little more noise present than in the pure regulatory mode, the overall performance is quite good.

The final gains chosen for the system are listed in Table 5.2.1. The proportional gains correspond to a deadband of ~1° with an error angle of ~19° corresponding to maximum correctional torque. The derivative gains correspond to deadbands of ~1.5, 1, and .5 degrees per second (for roll, pitch, and yaw, respectively) with errors of ~27, 20, and 16 degrees per second corresponding to maximum correctional torques.

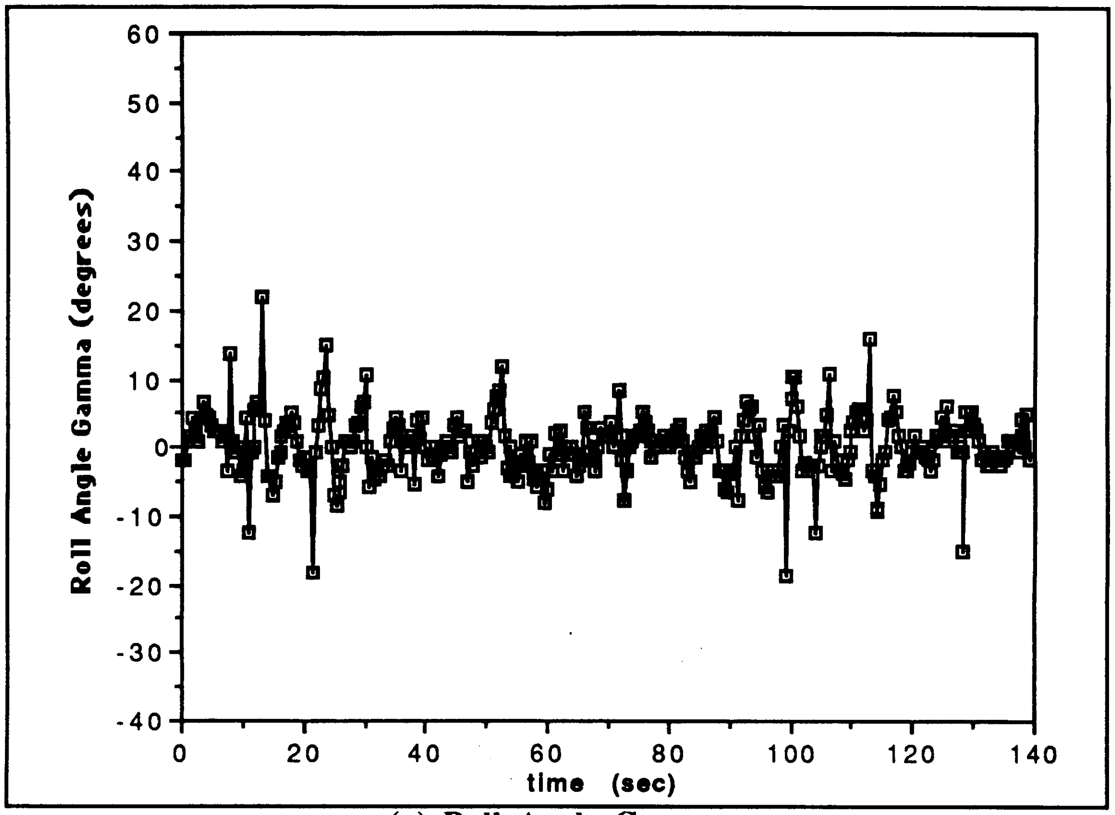


(a) Roll Angle Gamma

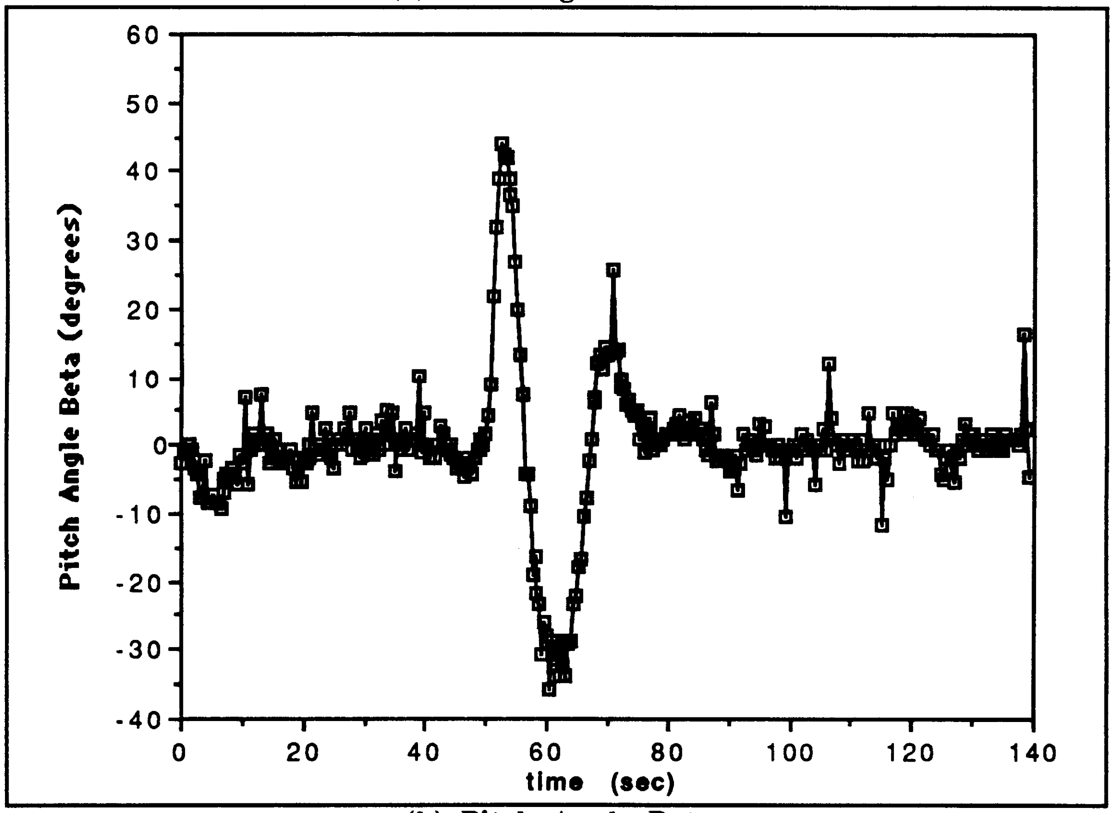


(b) Pitch Angle Beta

Figure 5.2.7 Orientation Angle Response To A Roll Disturbance

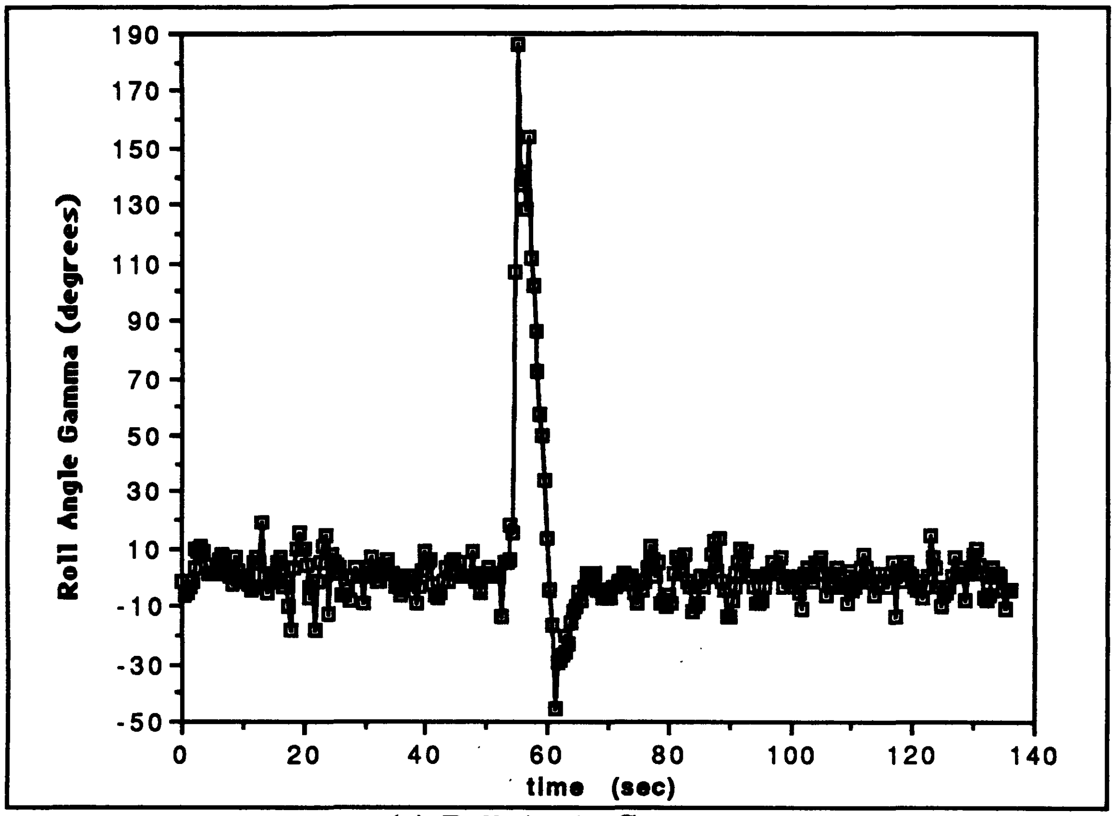


(a) Roll Angle Gamma

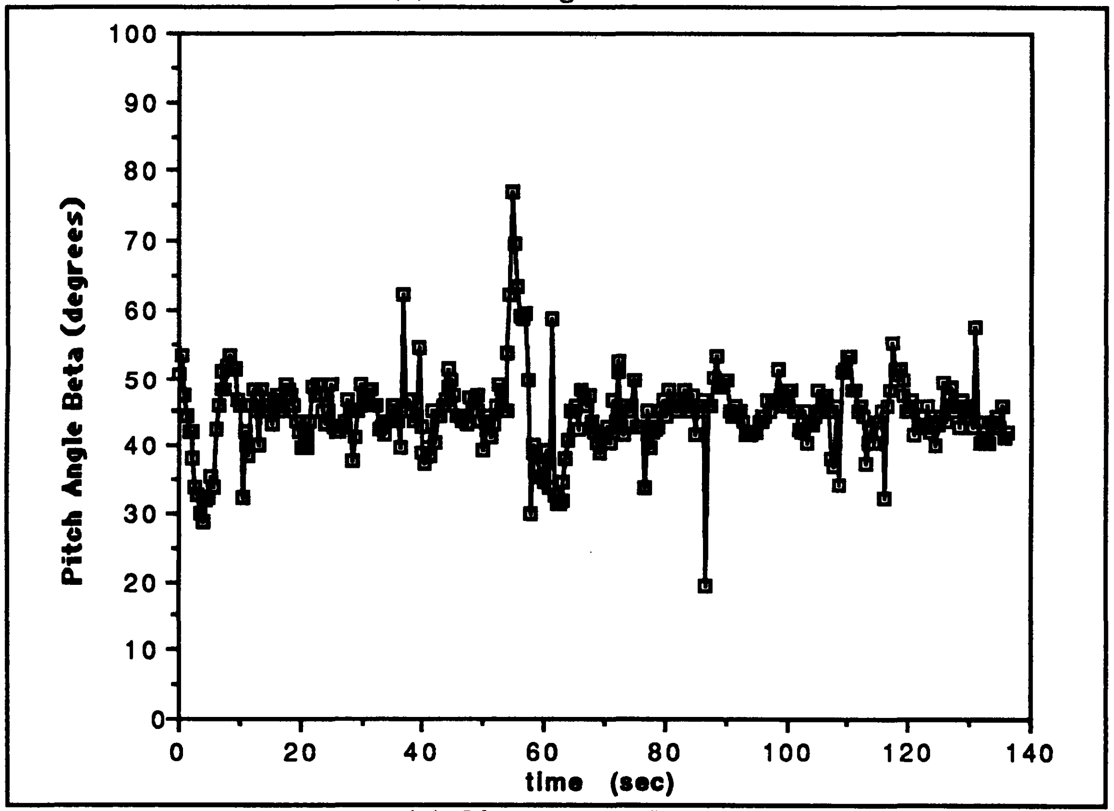


(b) Pitch Angle Beta

Figure 5.2.8 Orientation Angle Response To A Pitch Disturbance

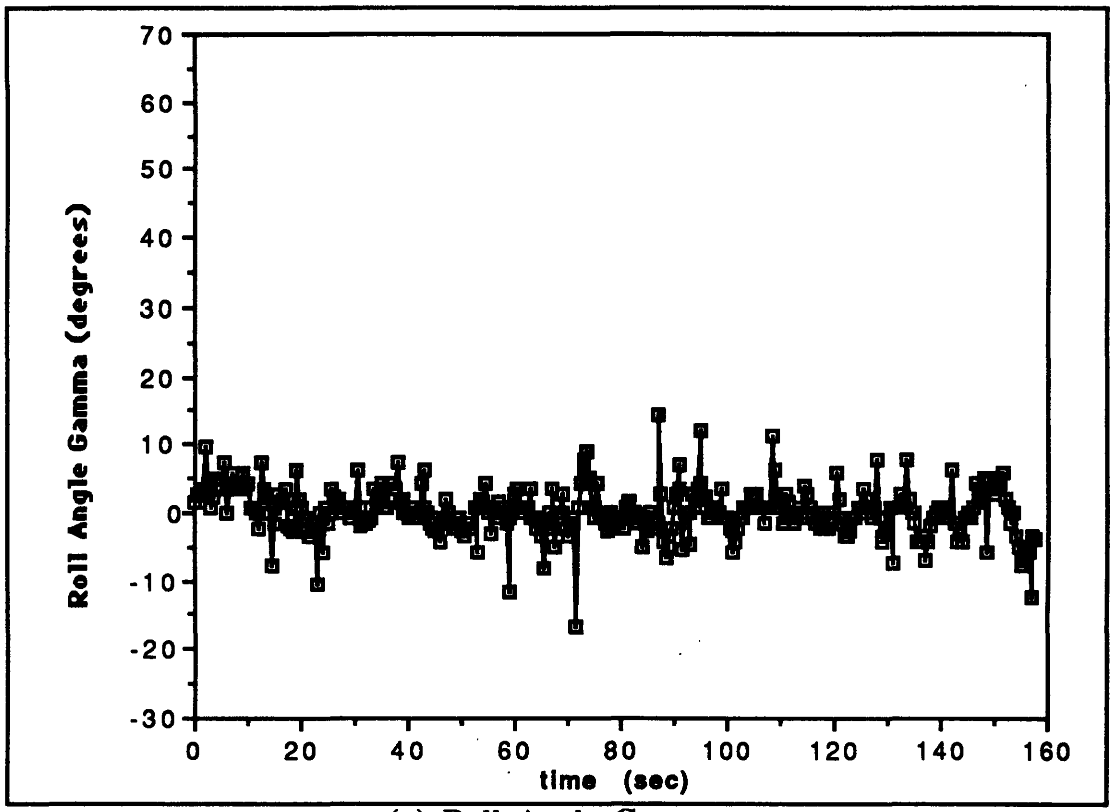


(a) Roll Angle Gamma

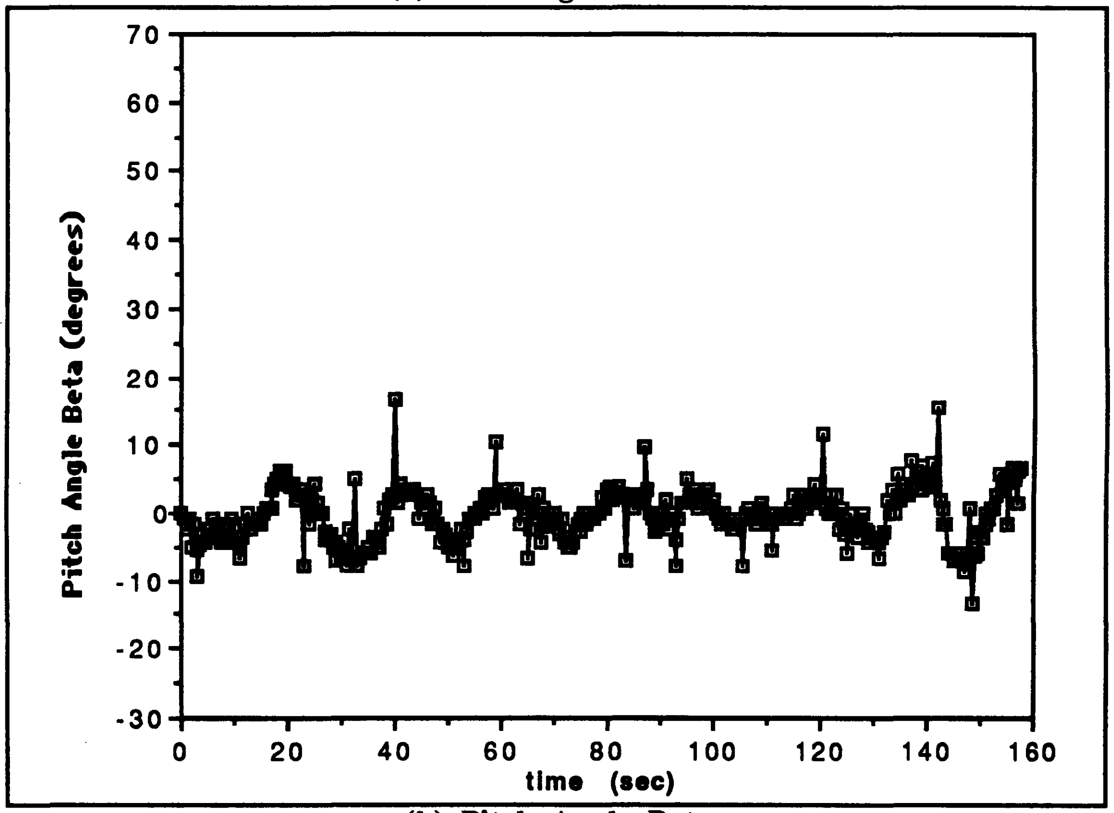


(b) Pitch Angle Beta

Figure 5.2.9 Orientation Angle Response To A Random Disturbance

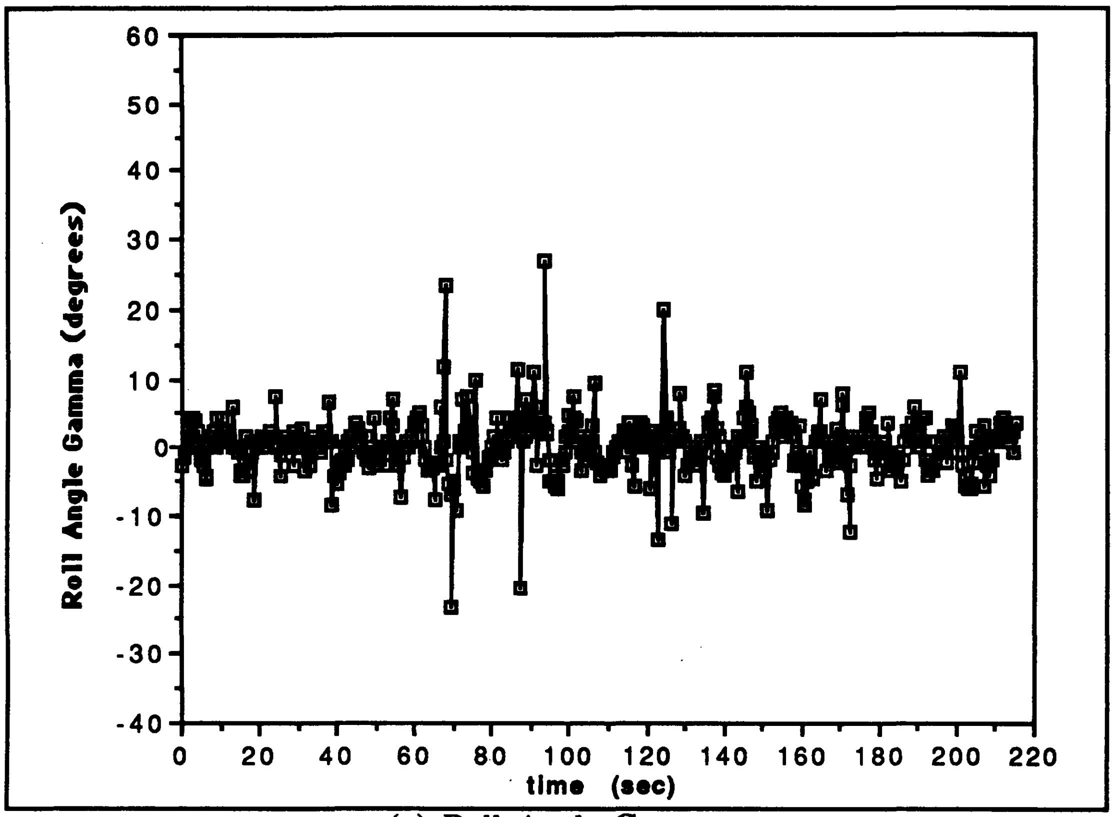


(a) Roll Angle Gamma

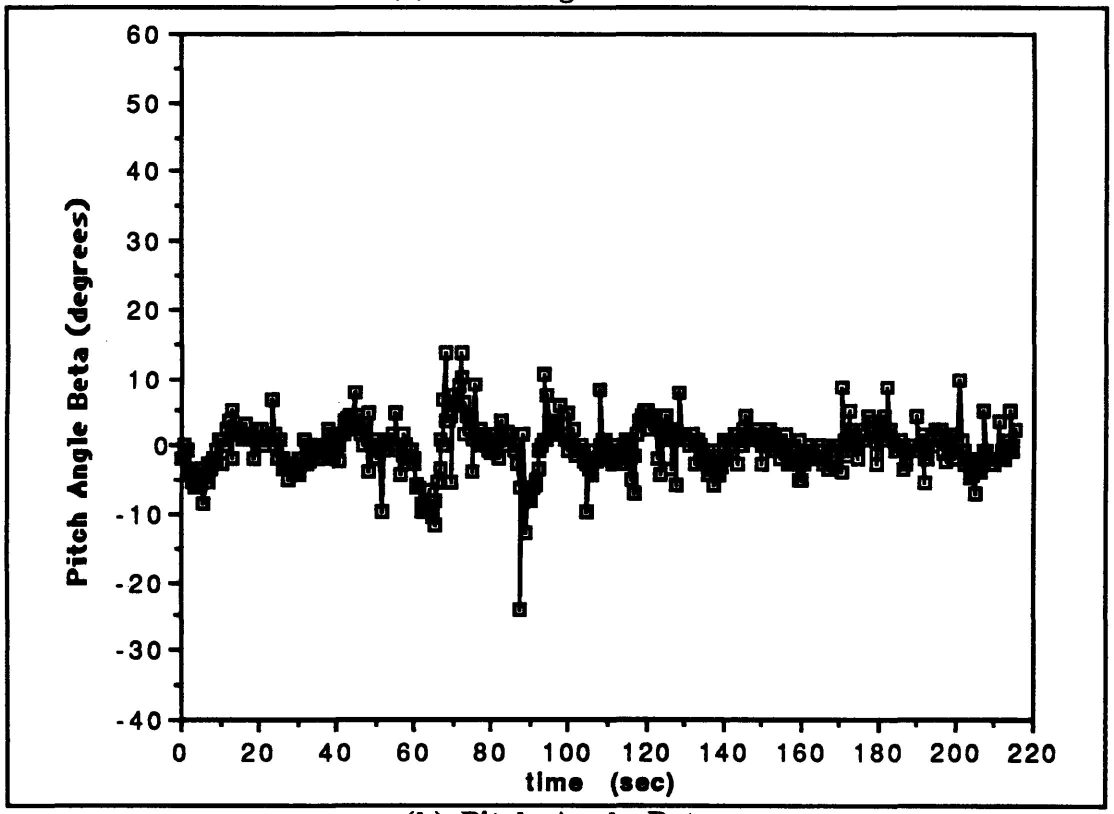


(b) Pitch Angle Beta

Figure 5.2.10 Orientation Angles During X-Translation



(a) Roll Angle Gamma



(b) Pitch Angle Beta

Figure 5.2.11 Orientation Angles During Y-Translation

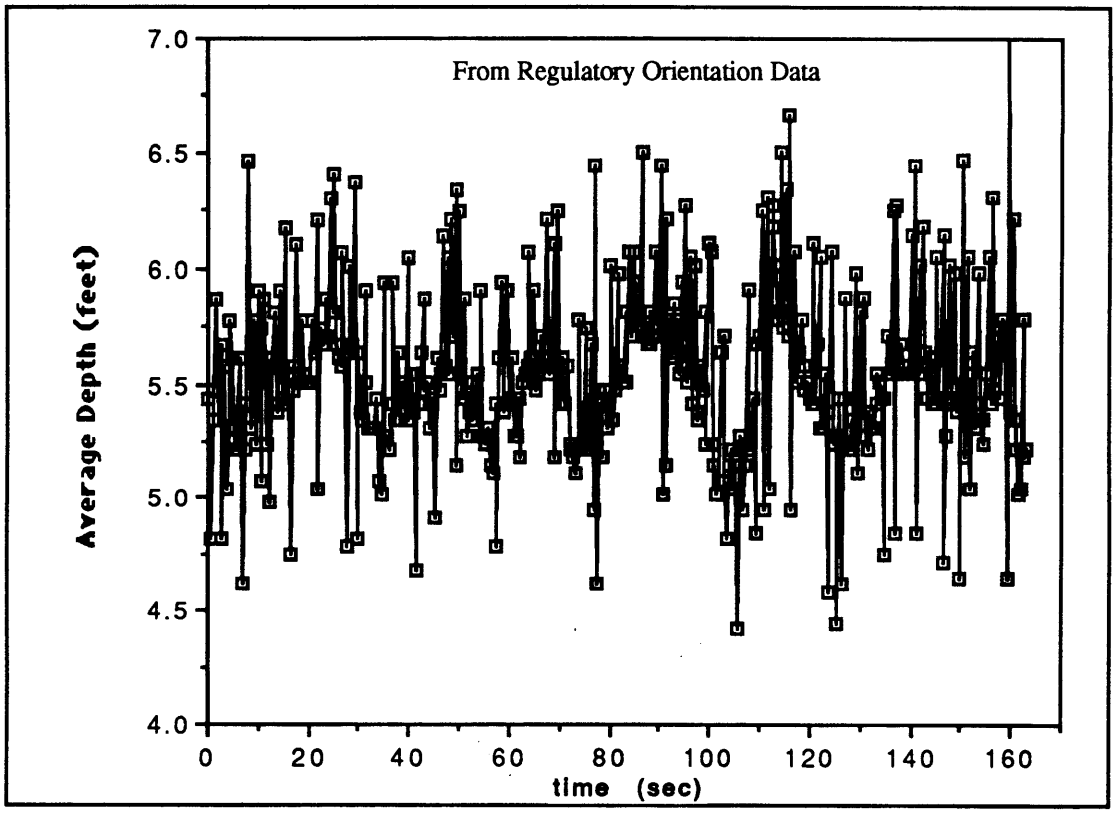
Table 5.2.1 Final PID Gains

Controller	Axes Roll	 Pitch	 Yaw
Proportional:	8	8	8
Derivative:	8	8	10
Integral:	6	7	---

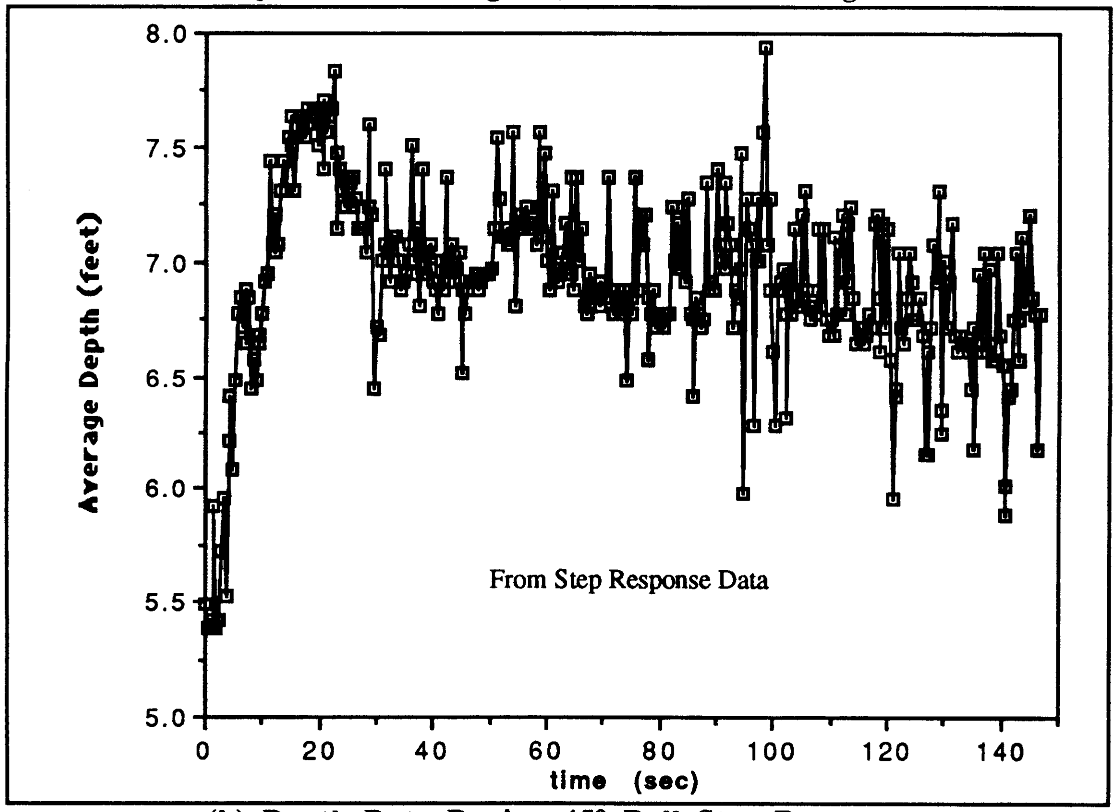
5.3 Depth Controller Results

The need for a depth controller was made apparent during the tests of the original PD controller. The effectiveness of the proportional depth controller was tested during all of the previously discussed tests as well as with depth disturbances. The depth controller worked very well overall during the regulatory, step response, disturbance, and free-flight tests performed for the orientation data. It kept the teleoperator within ± 0.5 feet of the desired depth most of the time. Sample graphs for each of these test cases is shown in Figure 5.3.1.

Gains for the depth controller were chosen by the controller's response to depth disturbances when in the (0°,0°), (45°,0°), and (0°,45°) orientations. Sample graphs of the depth data from each of the orientations are shown in Figures 5.3.2, 5.3.3, and 5.3.4. The final depth gains were chosen so that depth disturbances would be quickly corrected without excessive overshoot. This goal was made slightly more difficult because of the expansion and compression of the gases in the sealed boxes and the scuba tank as the teleoperator moves up and down. The controller was required to not only overcome the disturbance, but also the buoyancy of the expanded gases. The final gains chosen were 40 for each of the x, y, and z axes. This corresponds to maximum correctional thrust when the depth error is about 2 feet.

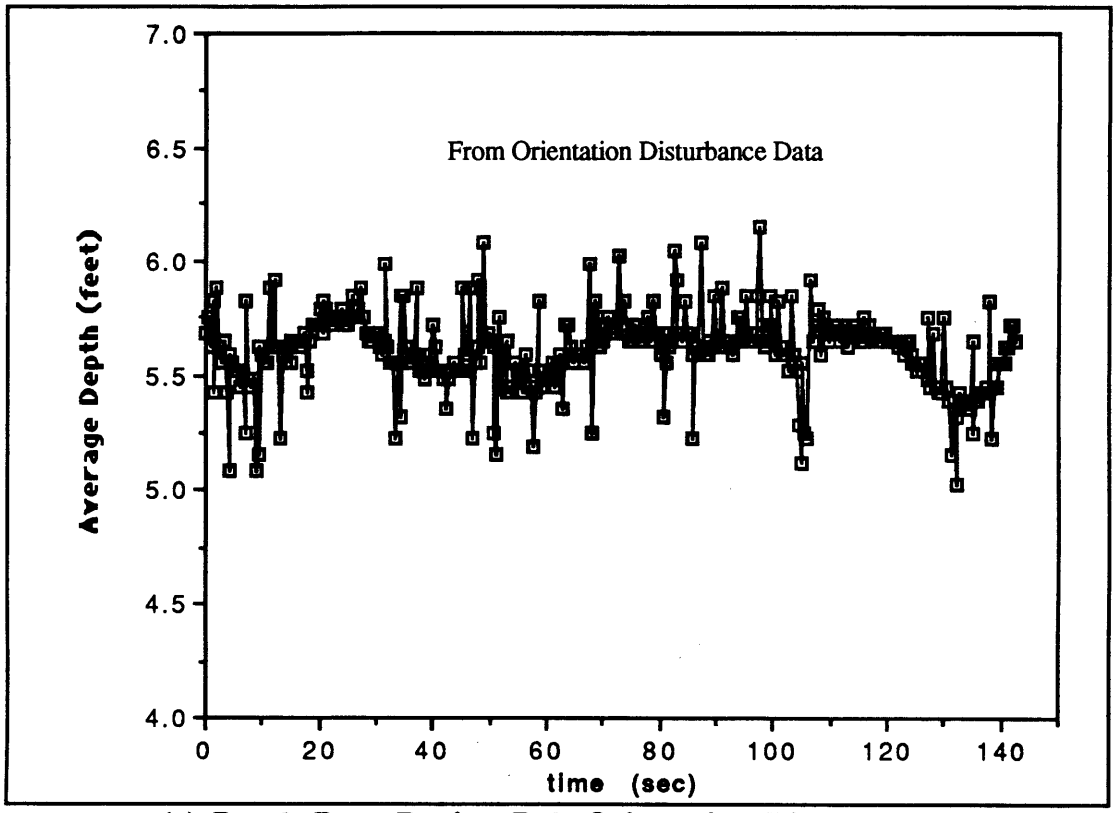


(a) Depth Data During (0°,0°) Orientation Regulation

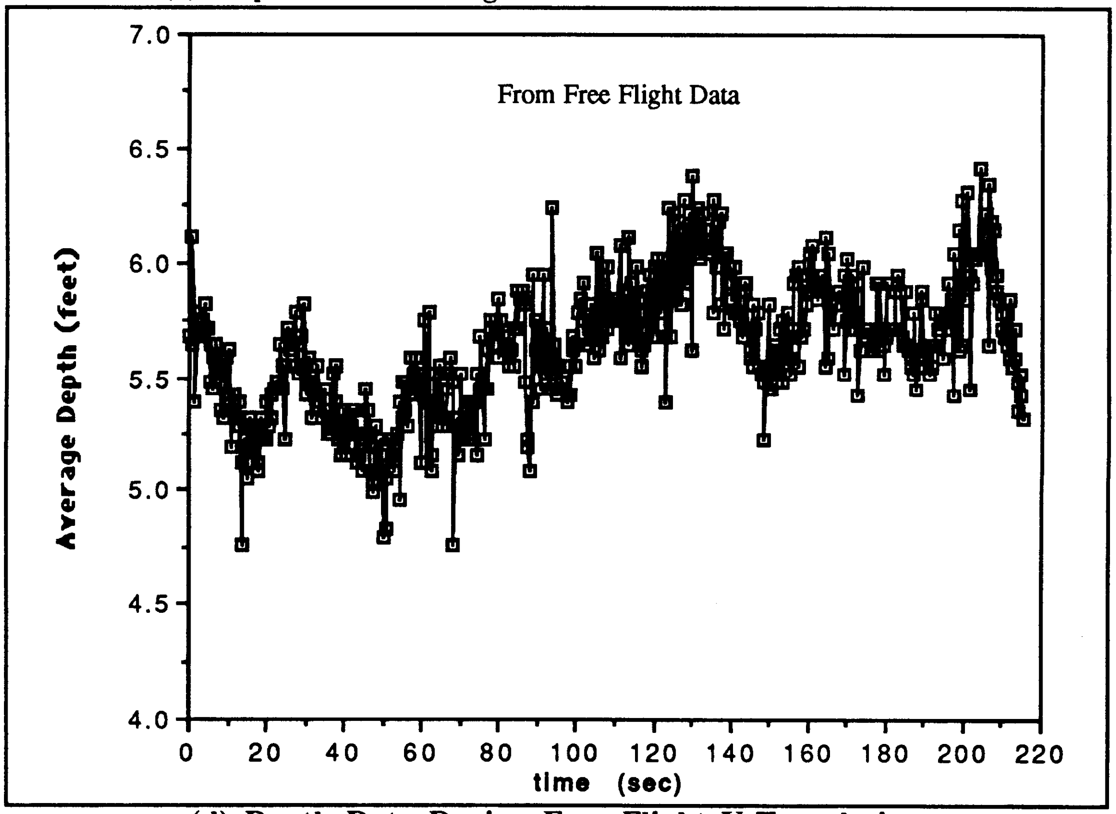


(b) Depth Data During 45° Roll Step Response

Figure 5.3.1 Average Depths During Orientation Data Acquisition



(c) Depth Data During Roll Orientation Disturbances



(d) Depth Data During Free Flight Y-Translation

Figure 5.3.1. Average Depths During Orientation Data Acquisition (cont.)

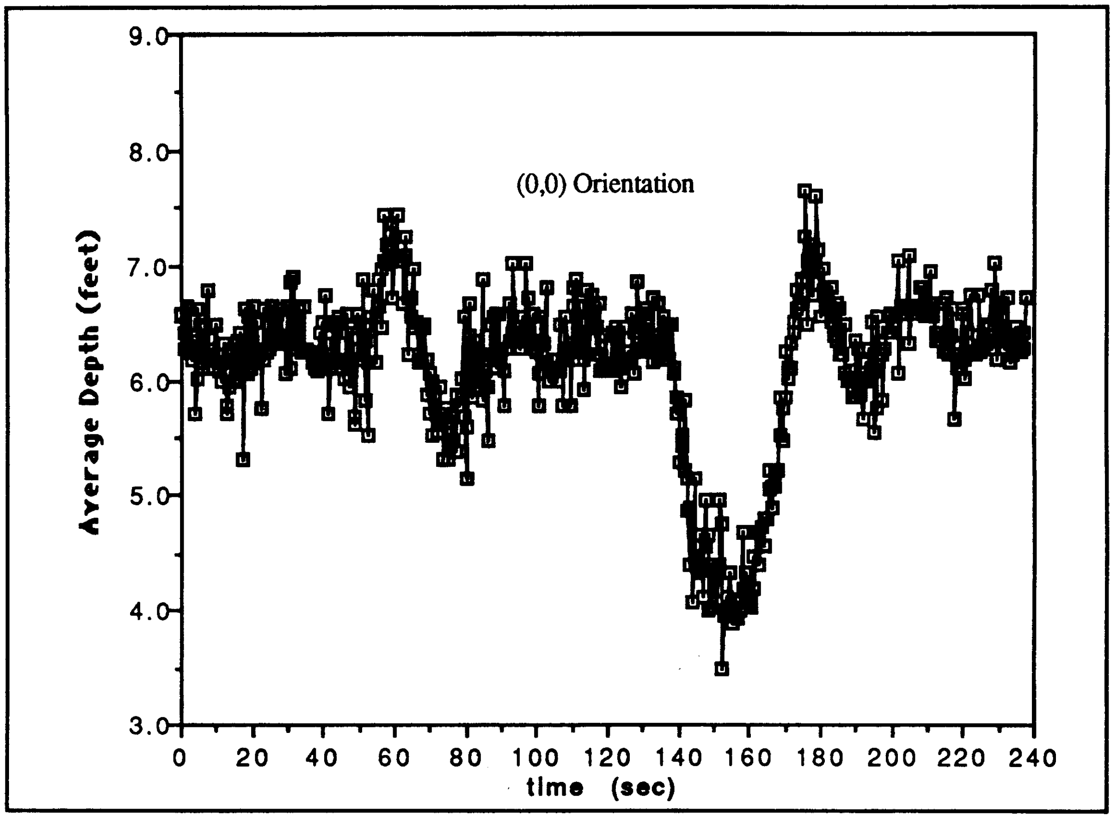


Figure 5.3.2 Depth Disturbances In (0°,0°) Orientation

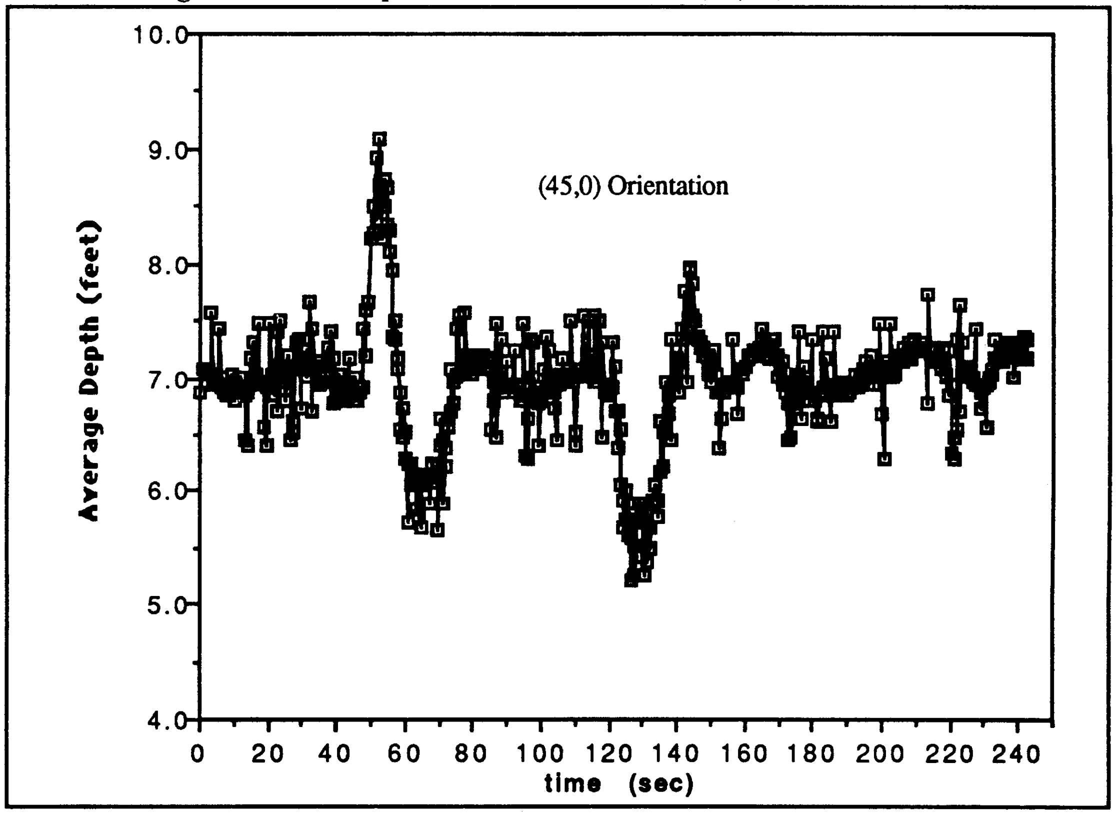


Figure 5.3.3 Depth Disturbances In (45°,0°) Orientation

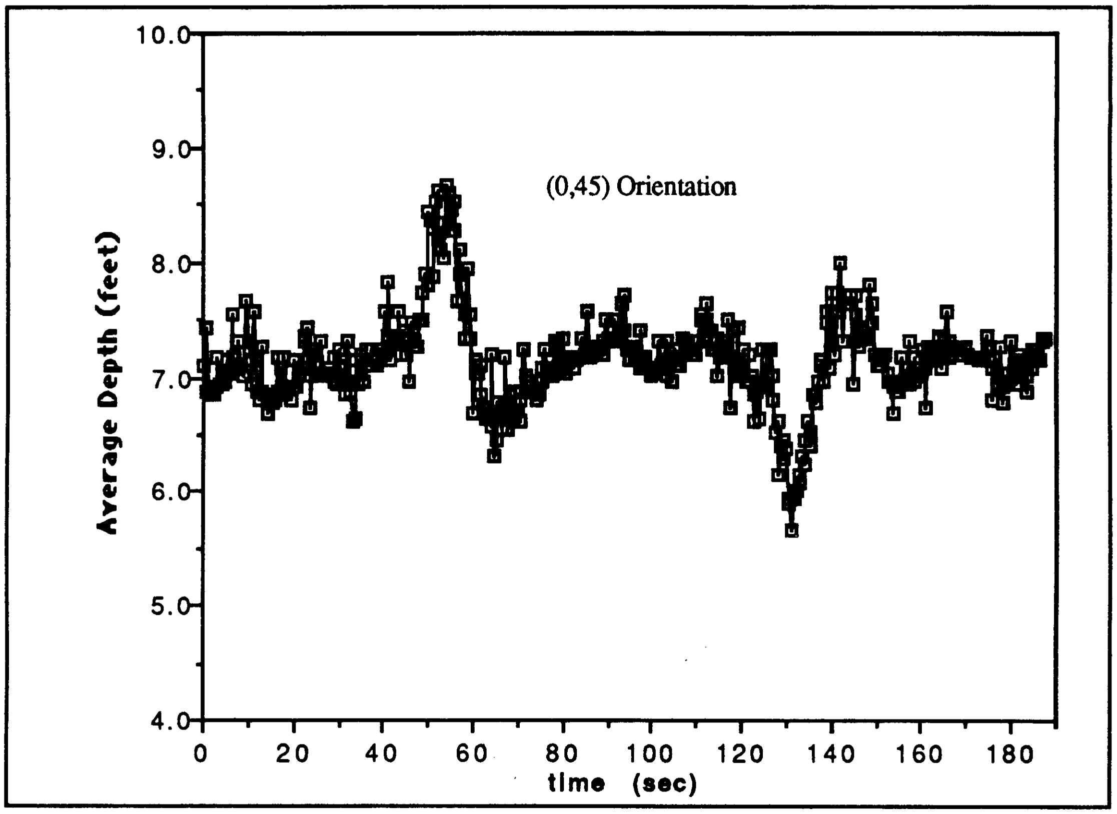


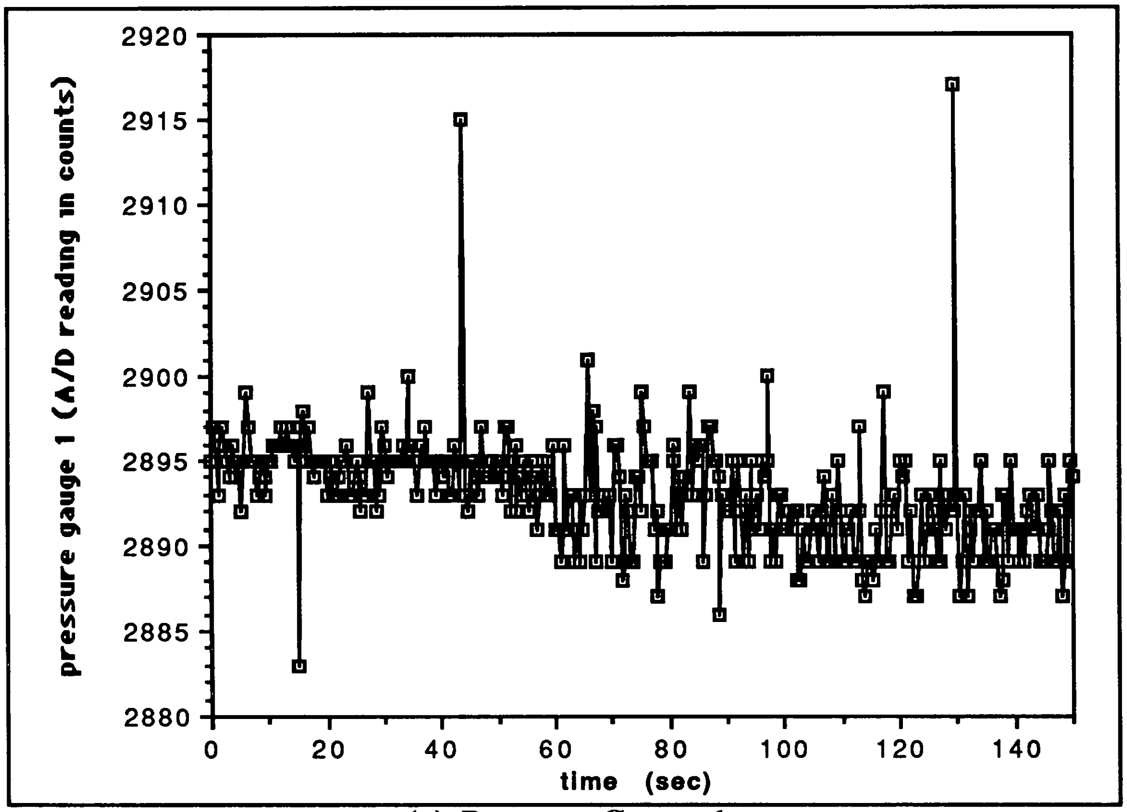
Figure 5.3.4 Depth Disturbances In (0°,45°) Orientation

5.4 Error Analysis

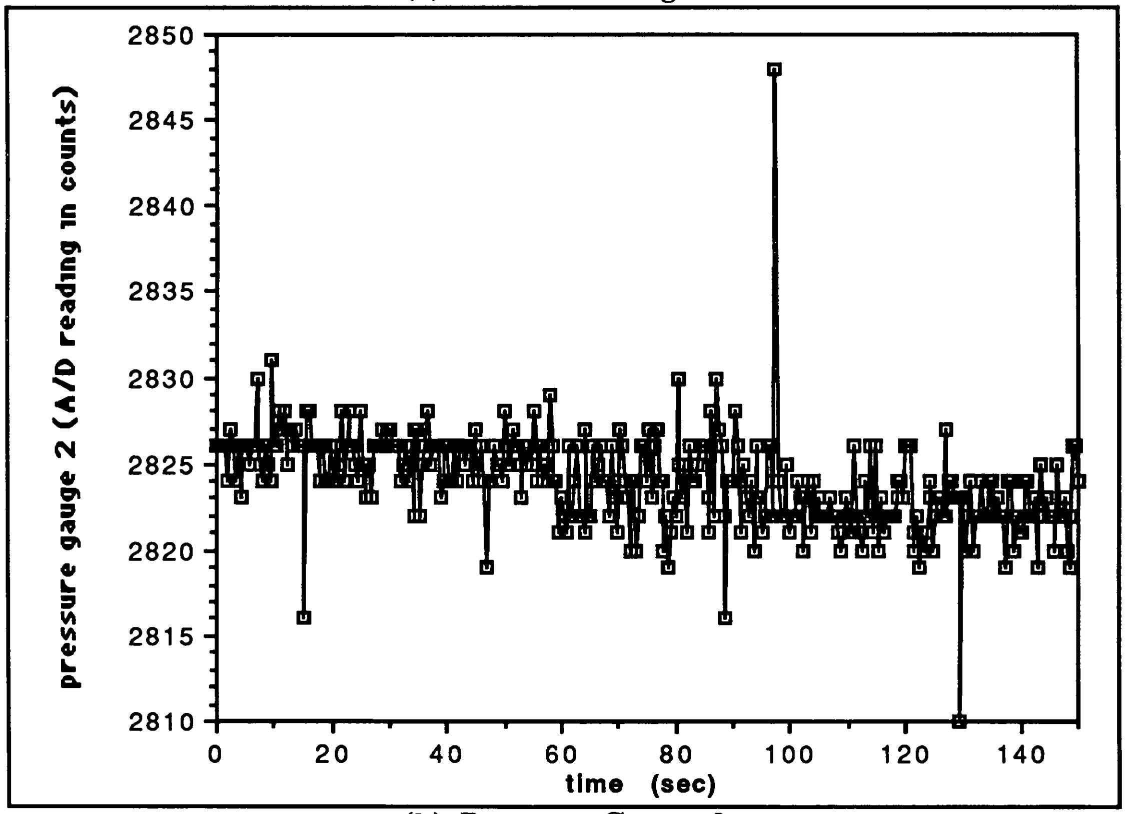
Data files were taken to determine the magnitude of the sensor noise and the expected accuracy of the PID and depth controller The teleoperator was weighted down and placed at the bottom of the pool in several different orientations Data files were acquired in exactly the same way as the regulator files were taken except that the main power switch on the robot was off Thus the software sampled the pressure and rate transducers and recorded the values to the data files at the same sample rate as when the robot orientation was actually being controlled The raw 12 bit, pressure and rate transducer data were examined as well as the orientation angles calculated from them

Figure 5 4 1 shows the raw (12 bit) data for each of the pressure transducers The range of values was within ±5 points with a few larger random spikes These particular data files were taken near the beginning of the test session Although the sensors were allowed to warm up in advance some drift in the values still occurred This drift was fairly small and the pressure transducers did not need to be re zeroed very often As can be seen in Figure 5 4 2 the orientation angles did not noticeably reflect the gauge value drift The range of the orientation angles appeared to fluctuate within ±2 to 3 degrees, with an occasional spike

Figures 5 4 3 5 4 4 and 5 4 5 show the rotation rate noise All of the values were within ±0 5 to 1 0 degrees per second However the values themselves were not zero indicating the need to reset the zero offset values These values had to be reset quite often despite the initial warm up time of approximately 15 minutes During data acquisition an attempt was made to check the rate transducer values between every third run Thus some rotation error and subsequent controller error can be attributed to the drift in the rate transducer values In general, the errors discussed here are the same magnitude as the errors of the attitude controller Thus the controller performed within the accuracy of the data provided by the pressure and rate transducers

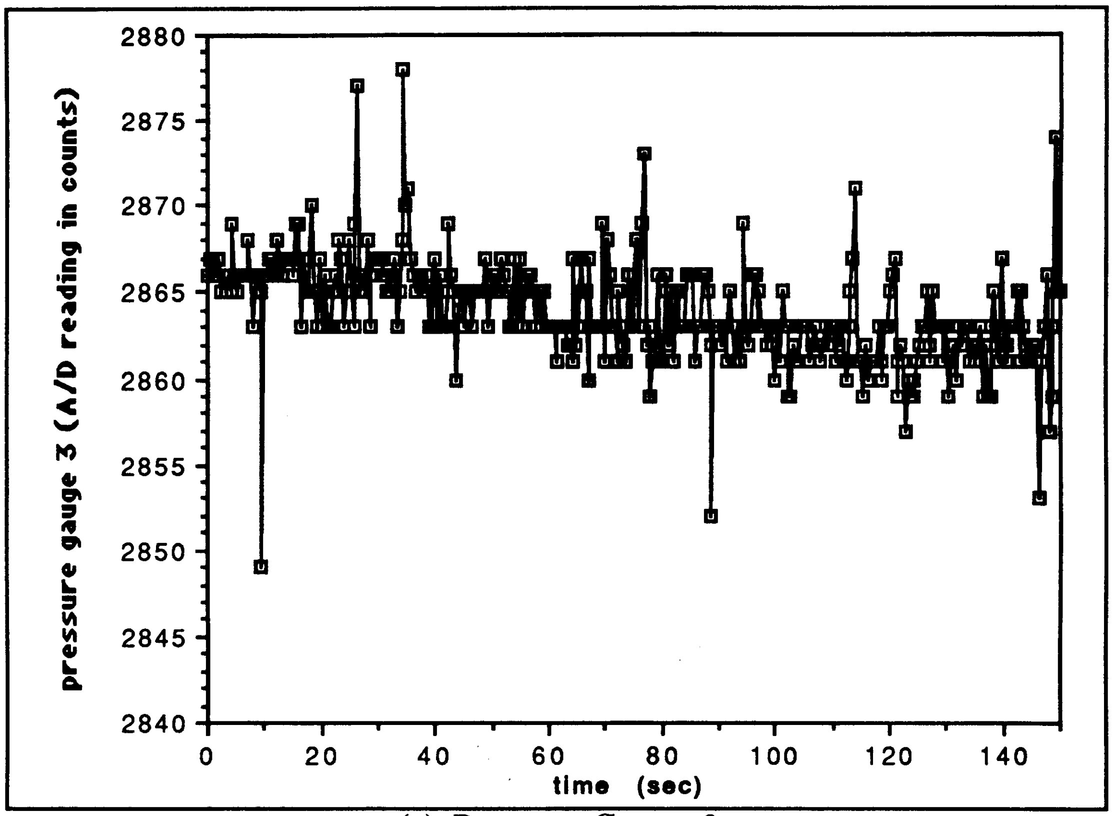


(a) Pressure Gauge 1

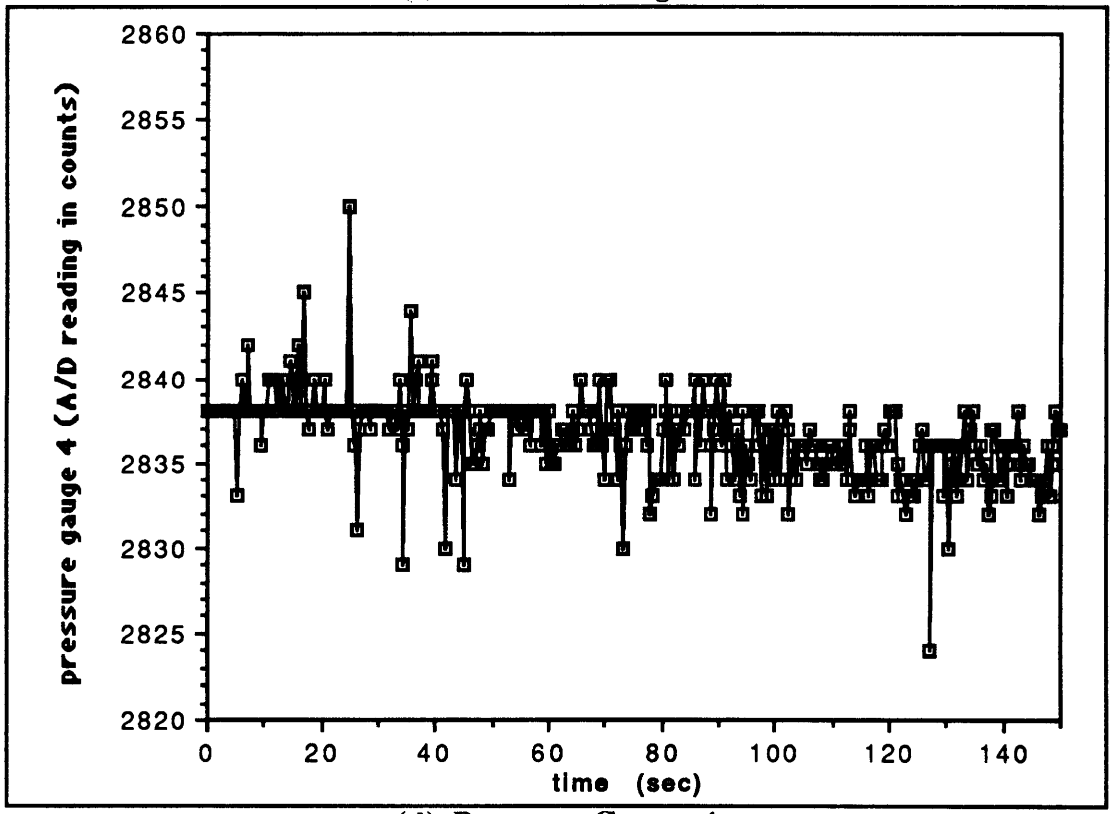


(b) Pressure Gauge 2

Figure 5 4 1 12 Bit Pressure Gauge Data

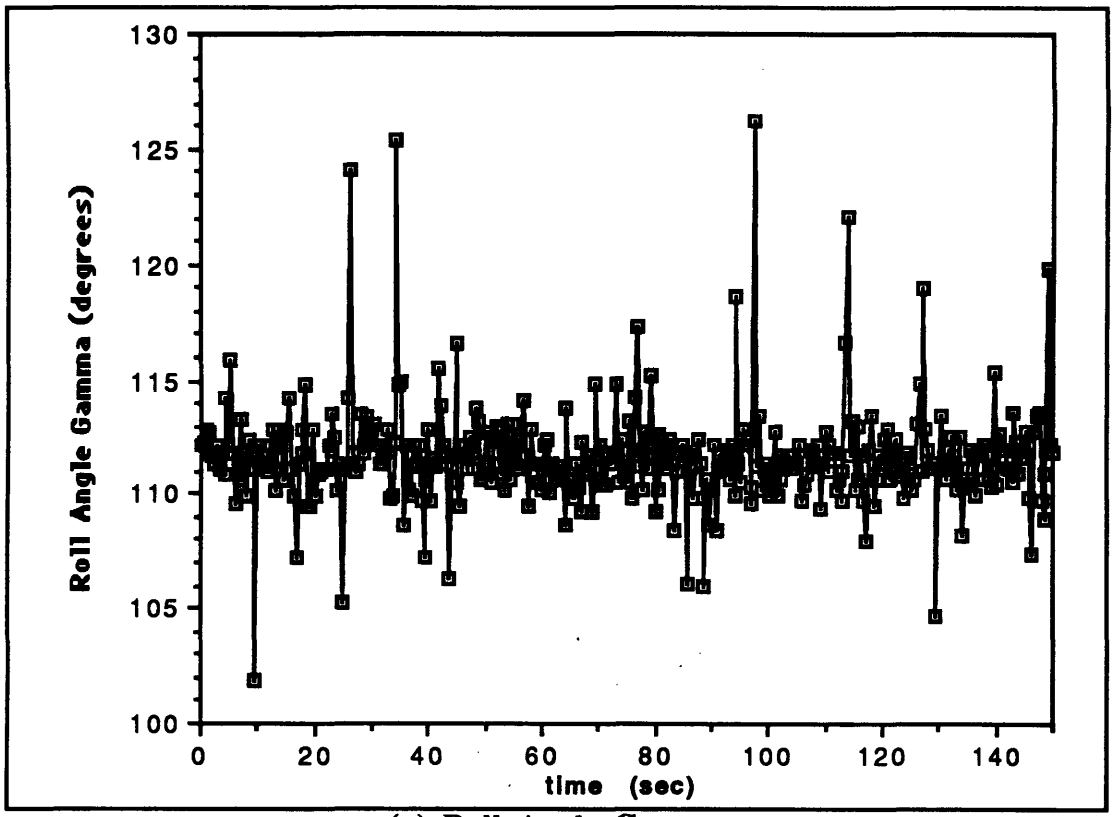


(c) Pressure Gauge 3

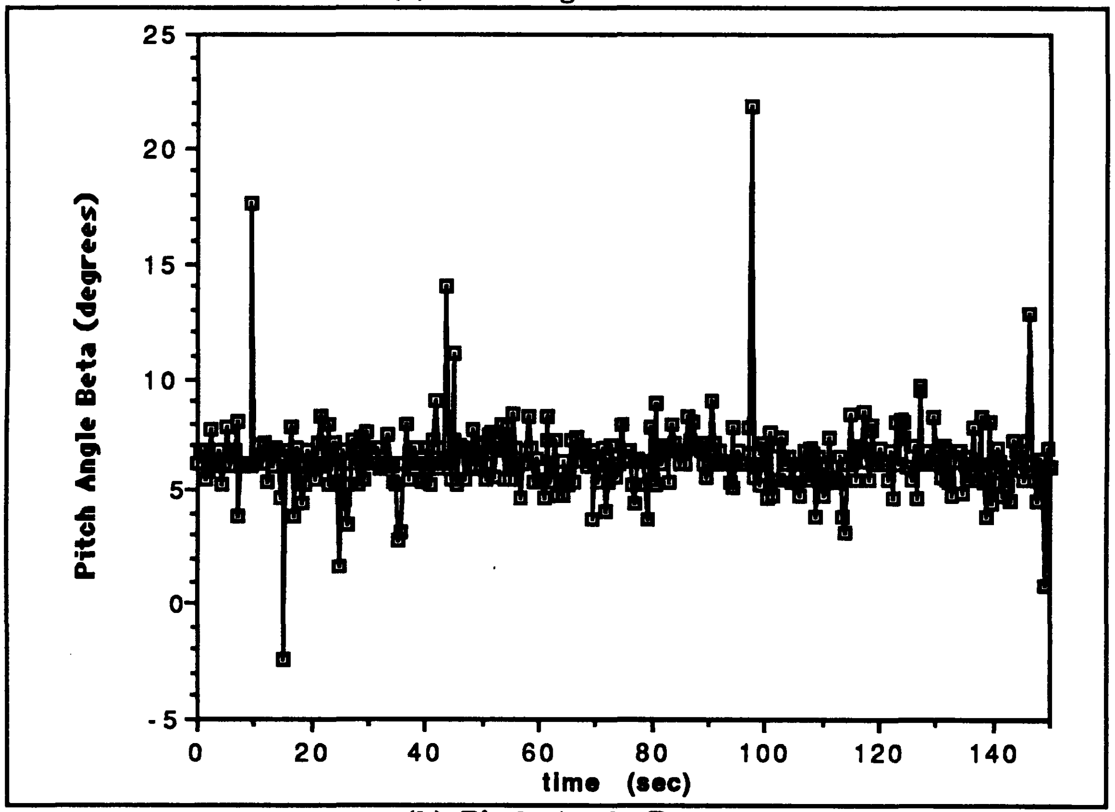


(d) Pressure Gauge 4

Figure 5.4.1 12-Bit Pressure Gauge Data (cont.)



(a) Roll Angle Gamma



(b) Pitch Angle Beta

Figure 5.4.2 Orientation Angles From 12-Bit Pressure Gauge Data

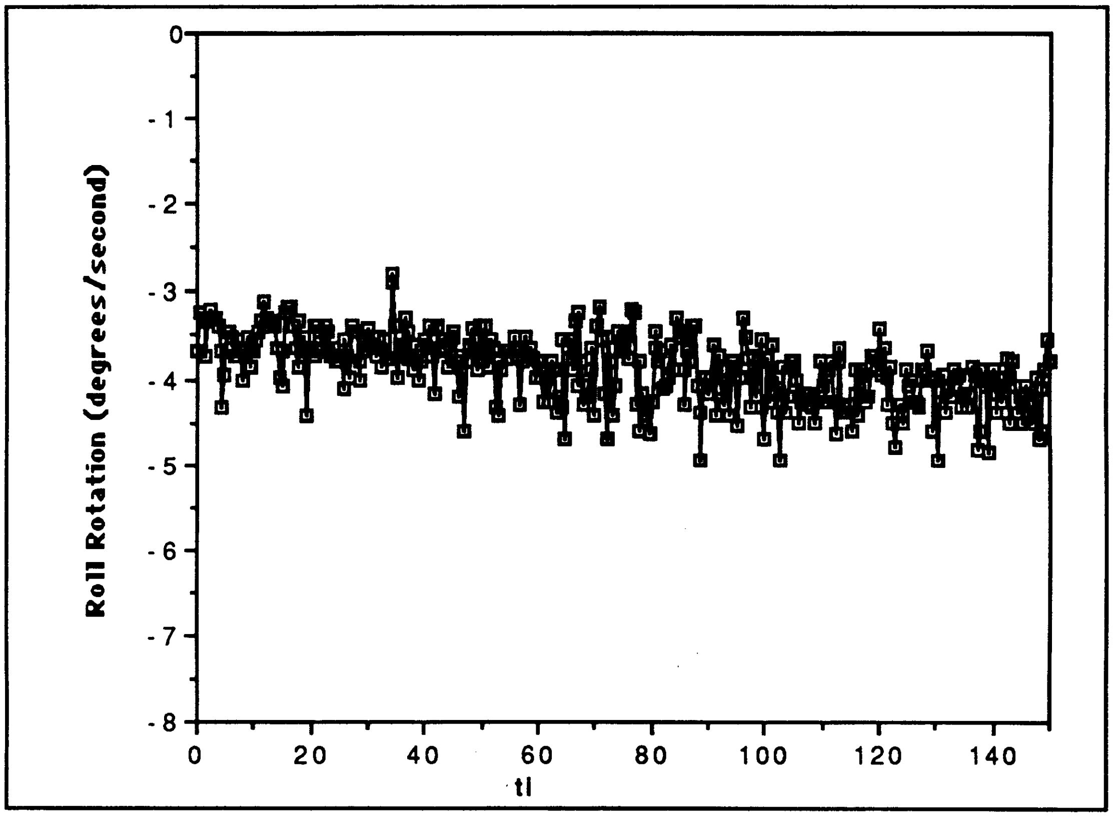


Figure 5.4.3 Rotation Rate Noise - Roll

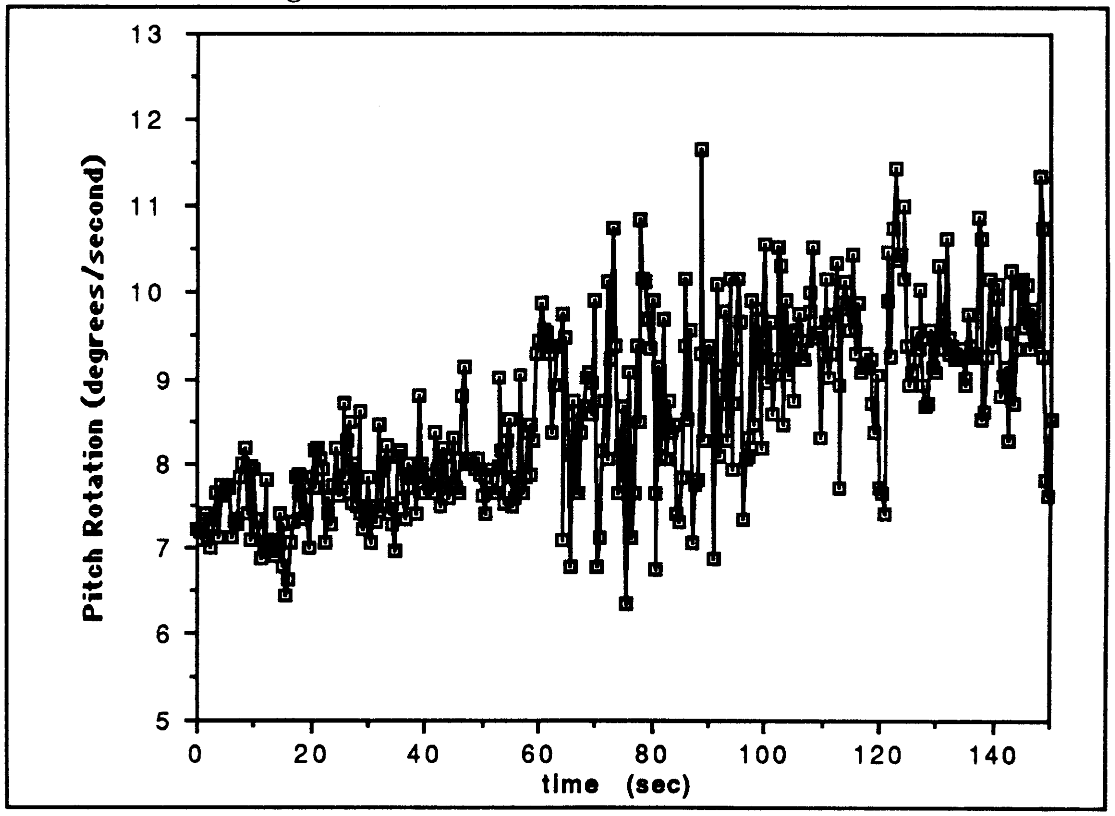


Figure 5.4.4 Rotation Rate Noise - Pitch

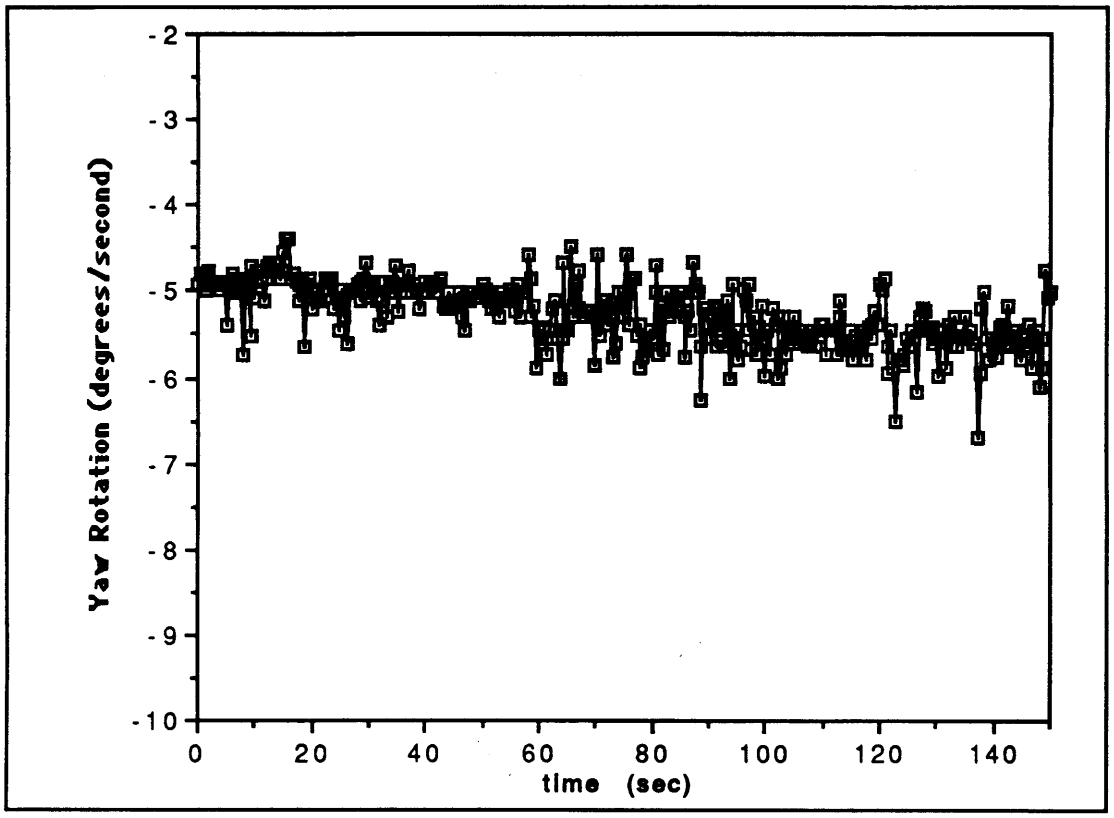


Figure 5.4.5 Rotation Rate Noise - Yaw

6.0 Conclusions and Recommendations

The implementation of a closed-loop PID attitude controller and proportional depth controller greatly improved the performance of the underwater teleoperator. Desired orientations in roll and pitch were maintained within ±5 degrees. The proportional depth controller maintained the teleoperator's depth within ±0.5 feet. These errors, when seen on the video monitor (through the camera mounted on the front of the robot) were of no major consequence. The system response to a 45° step input was fairly quick (~10 seconds), but the overshoot was rather large. Further experimentation should examine the effects of reduced proportional and integral gains and increased derivative gains on the response time and the overshoot.

There are several improvements to this control system that can be made which would significantly improve performance. First, a faster onboard computer, or a computer with a math co-processor would greatly decrease the loop execution time. The current cycle time for the complete controller is about 0.4 seconds or 2.5 hertz. Tests performed in the laboratory have shown that the most time consuming portions of the software are the proportional and depth control calculations. The calculation times of these functions were longer due to several floating point and trigonometric calculations. Accessing the 12-bit analog to digital converter to obtain the pressure and rate transducer data also took a significant amount of time. Portions of this software may be rewritten in assembly language and linked to C to further decrease the execution time.

To decrease the amount of noise in the attitude controller, a simple filter to eliminate spikes in the pressure transducer data, and thus the orientation angles, should be implemented. Further experimentation to determine the precise source of the spikes should be performed in order to determine whether the filter would be most effective implemented in software or hardware. Once a filter is installed and the spikes eliminated, the proportional attitude controller gains could be increased slightly (they were decreased to limit response to the spikes).

Another improvement to the attitude control system would be the development of a more complicated system to eliminate yaw rotations about the pool z-axis. This rotation was held to less than 1.0 degrees per second with the current controller in the (0°,0°) orientation due to the relatively high derivative feedback gain on the teleoperator yaw axis (which lines up with the pool yaw axis in this orientation). However, the rotation about the pool yaw axis was greater when the teleoperator was in other orientations. One possible method for correcting this rotation is to use the currently calculated attitude to determine an additional weighted rate feedback correction factor. Other methods should also be investigated.

Additional improvements to the overall teleoperator system should include expanding the internal systems monitor. On-line voltage readings for the main power as well as a digital pressure gauge to measure scuba tank pressure could be implemented. Thruster feedback data such as shaft rotation rate could also be monitored and tied into the control system.

While the closed-loop attitude and depth controller developed in the course of this research worked very well for the purposes of current teleoperator usage, a more expanded and advanced system based on the recommendations above would further increase the usefulness of this teleoperator for future research.

References

1. Quick, D.M. , "Apparatus for Space Telerobotic Operations (ASTRO)", SSL Guide, August 7,1987.
2. Power, W.M., "Guide to Operations and Systems: Apparatus for Space Tele-Robotic Operations (ASTRO)", SSL Report 1-90.
3. Sanner, R., "The MIT SSL Pilot-Vehicle-Control Station Communications Protocal (PIVECS)", SSL Report in progress.
4. Craig, J.J., *Introduction to Robotics: Mechanics and Control,* Addison-Wesley Publishing Company, Massachusetts, 1986.
5. Alexander, H.L., Class Notes for "Space Robotics", September 1988.

Asada, H. and Slotine, J.J., *Robot Analysis and Control,* Wiley and Sons, New York, 1986.

D'Azzo, J.J., and Houpis, C.H., *Feedback Control System Analysis and Synthesis,* McGraw-Hill Book Company, New York, 1966.

Paul, R.P., *Robot Manipulators: Mathematics, Programming, and Control,* MIT Press, Massachusetts, 1982.

Thomas, G.B., and Finney, R.L., *Calculus and Analytic Geometry,* Addison-Wesley Publishing Company, Massachusetts, 1982.

Appendix A: ASTRO Software

```
/* WPVASTRO */
/* Last Modified: 3/6/90 by Wendy Power */
/* This is the compile and link program used with the onboard flight software. */

wpvastro.obj: ..\pivecs.h pvastro.h ..\pvsecs.msg wpvastro.c
        cl /c /Od /Zp /Zi wpvastro.c

astrofn1.obj: ..\pivecs.h pvastro.h astrofn1.c
        cl /c /Od /Zp /Zi astrofn1.c

astrofn2.obj: ..\pivecs.h pvastro.h astrofn2.c
        cl /c /Od /Zp /Zi astrofn2.c

astromsg.obj: ..\pivecs.h ..\pvdata.h pvastro.h ..\pvsecs.msg astromsg.c
        cl /c /Od /Zp /Zi astromsg.c

wpvastro.exe: wpvastro.obj astromsg.obj astrofn1.obj astrofn2.obj
        link /NOD /CO  $**, wpvastro.exe,,SLIBCE+GFCS+GFS+..\PIVECS
```

```
/* PVASTRO.MSG */
/* Last Modified: 3/6/90 by Wendy Power */
#ifndef ASTROMSGS
#define ASTROMSGS

/* This is a listing of the message headers for ASTRO */

/********************** Recognized Message List **************************/
#define  COMTEST      0x00    /* Msg 0, No Data          */
#define COMAOK        0x08    /* Msg 1,  " "             */
#define SHUTDN        0x10    /* Msg 2,  " "             */
#define STOP          0x18    /* Msg 3,  " "             */

#define  TXSTATS      0x68    /* Msg 13,  " "    */
#define TXPGAUGE      0x70    /* Msg 14,        " "      */
#define TXGYROS       0x78    /* Msg 15,        " "      */
#define TXCOUNT       0x80    /* Msg 16,        " "      */

#define  RXTHC        0xA3    /* Msg 20, 3 Data Bytes    */
#define  RXRHC        0xAB    /* Msg 21, 3 Data Bytes    */
#define  RXSPANEL     0xB1    /* Msg 22, 1 Data Byte     */
#define RXCAMERA      0xB9     /* Msg 23, 1 Data Byte */
#define RXEULER       0xC4    /* Msg 24, 4 Data Bytes */
#define RXPGAIN       0xCB    /* Msg 25, 3 Data Bytes */
#define RXRGAIN       0xD3    /* Msg 26, 3 Data Bytes */
#define RXIGAIN       0xDA    /* Msg 27, 2 Data Bytes */
#define RXDPGAIN      0xE3    /* Msg 28, 3 Data Bytes */

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

/* The message structure is as follows:
#define  MSGname                0x??    /* Msg # (0-31) and # of data bytes (0-7)
The number of data bytes should be 0 if transmitting, and the appropriate
number if receiving.  ?? is a 2 digit hex representation of the message
number (most significant 5 bits) and the number of data bytes (3 least
significant bits).
Example:  receiving message number 26 with 3 data bytes
        26 = 11010 in bits, 3 = 011
        combining: 11010 011 -> 1101 0011 -> hex = D3               */
```

```
/* PVASTRO.H */
/* Last Modified: 3/6/90 by Wendy Power */
/* This routine contains all the message handler and header
        definitions, as well as all the global and fixed definition
        variables for ASTRO. */

#ifndef ASTRO
#define ASTRO
#ifndef PIVECS
#include "..\pivecs.h"
#endif
#ifndef ASTROMSGS
#include "..\pvastro.msg"
#endif

extern HandlerFunc          BadMsg, ShutDown, ComCheck, ComAOK;
extern HandlerFunc          TX_Stats, TX_PGauge, TX_Gyros, TX_Count;
extern HandlerFunc          RX_THC, RX_RHC, RX_SPanel, RX_Camera;
extern HandlerFunc          RX_Euler, RX_PGain, RX_RGain, RX_IGain, RX_DpGain;

static Handlers     AstroHandlers =
                    {ComCheck, ComAOK,  ShutDown,   BadMsg,
                    BadMsg,   BadMsg,   BadMsg,     BadMsg,
                    BadMsg,   BadMsg,   BadMsg,     BadMsg,
                    BadMsg,   TX_Stats, TX_PGauge,  TX_Gyros,
                    TX_Count, BadMsg,   BadMsg,     BadMsg,
                    RX_THC,   RX_RHC,   RX_SPanel,  RX_Camera,
                    RX_Euler, RX_PGain, RX_RGain,   RX_IGain,
                    RX_DpGain, BadMsg,  BadMsg,     BadMsg};

static Headers      AstroMsgs =
                    {COMTEST, COMAOK,  SHUTDN,    STOP,
                    BADMSG,  BADMSG,   BADMSG,    BADMSG,
                    BADMSG,  BADMSG,   BADMSG,    BADMSG,
                    BADMSG,  TXSTATS,  TXPGAUGE,  TXGYROS,
                    TXCOUNT, BADMSG,   BADMSG,    BADMSG,
                    RXTHC,   RXRHC,    RXSPANEL,  RXCAMERA,
                    RXEULER, RXPGAIN,  RXRGAIN,   RXIGAIN,
                    RXDPGAIN, BADMSG,  BADMSG,    BADMSG};

static Byte     AstroHiPri = 4;
static Byte       Bit[8] = {0x01, 0x02, 0x04, 0x08, 0x10, 0x20, 0x40, 0x80};

/* Define global variables (can be called from all routines). */
Byte                Motors[6], thc[3], rhc[3];
Byte                PwrChg, IrisChg, Iris, SPanel, PneuChg, DumpFlg, changes;
Byte                Done;
unsigned long       LastTHC, LastRHC, LastSPanel, count, p, g;
unsigned long       data[2000][7];
long                pgauge[4], gyros[3], gyroadj[3], pgadj[3], adnum[7];
int                 derivcmd[3], propcmd[3], rategain[3], propgain[3];
int                 intgain[2], intcmd[2], depthgain[3], depthcmd[3];
long                cdepth, ddepth;
double              kvector[3], esum[2], zvector[3];
double              thetae, thetaed, sintemp, gammad, betad, gammac, betac;
double              singc, cosgc;

/* Define fixed gloabal variables */
#define WLEN        0x05
#define ADMASK      0xE0
#define MSBMASK     0xF0
#define TMAX        0x0F
#define DELAY       0x100
#define  HTGGLE     0x09
```

```
#define   LTGGLE            0x08
#define   LKMASK            0x0E
#define tsample             0.395
#define pi                  3.14159
#define BIT7                0x0E
#define BON                 0x01
#define BOFF                0x00
#define MPON                0x80
#define   TogC6             0x40
#define TogC5               0x20

/* define SECS control switch names */
#define PWR_SWITCH          0x80
#define PropCtl             0x40
#define DerFdBk             0x20
#define GYRO_RESET          0x10
#define DepthCtl            0x08
#define PGADJ1              0x08
#define IntgrlCtl           0x04
#define PGADJ4              0x04
#define PGADJ23             0x02
#define NOHCCMD             0x02
#define TKDATA              0x01

/*        ASTRO Protoboard port addresses     */
#define PORT1A   0x300
#define PORT1B   0x301
#define   PORT1C  0x302
#define CNTRL1   0x303

#define PORT2A   0x304
#define PORT2B   0x305
#define   PORT2C  0x306
#define CNTRL2   0x307

#define PORT3A   0x308
#define PORT3B   0x309
#define   PORT3C  0x30a
#define CNTRL3   0x30b

/*        Astro functions       */
void      AstroInit(), FireMotors(), EngageMotors();
void      FirePneu(), FireCamera(), RdAtoD(), FPropCtl(), FDerFdBk();
void      FDepthCtl(), FIntgrlCtl();
Byte      JetSelect();

#endif
```

```
/* WPVASTRO.C */
/* Last Modified 3/6/90 by Wendy Power */
/* This is the main program routine running on ASTRO.  All
        function calls are made from this program. */

#include "..\pivecs.h"
#include "pvastro.h"
#include "..\pvsecs.msg"
#include <gf.h>
#include <stdio.h>
#include <time.h>
#include <math.h>
#include <sys\timeb.h>
#include <sys/types.h>

main() {            /* Begin PiVecs driver program */
        register i;
        unsigned long   CurrMsg = 0;
        double          time1, time2, deltat;
        struct timeb etime1;
        struct timeb etime2;

        /* Initialize Pivecs and Astro Computer System */
        pvInitCom( COM1, 9600, P_ODD, 1);
        pvInitMsg( AstroHandlers, AstroMsgs, AstroHiPri);
        AstroInit();
        pvRequest( TXSPANEL );

        /* Begin main driver loop */
        while (TRUE) {

        CurrMsg = pvRecv();
        /* Ask for current switch panel settings if it has timed out. */
        if ( (CurrMsg - LastSPanel) > WLEN ) {
                pvRequest( TXSPANEL );
                LastSPanel = CurrMsg;
                }

        if (!(DumpFlg)) {   /* if not dumping data to surface ... */
        /* If main power ON, request THC and RHC. */
        if (!(SPanel & NOHCCMD)) {          /* if hand controllers are not disabled */
        if (SPanel & PWR_SWITCH) {          /* if main power is on */
                if ( (CurrMsg - LastTHC) > WLEN ) {  /* ask for THC if timed out */
                        pvRequest( TXTHC );
                        LastTHC = CurrMsg;
                }
                if ( (CurrMsg - LastRHC) > WLEN ) {  /* ask for RHC if timed out */
                        pvRequest( TXRHC );
                        LastRHC = CurrMsg;
                }
        }
        }

        /* Additional data processing and data acquisition. */
        RdAtoD();                       /* Read Pressure Gauges and Gyros (ad363) */
        if (PneuChg) FirePneu();        /* Pneu = MP */
        if (IrisChg) FireCamera();      /* Change camera light aperture */

        /* Check for gyro adjust reset */
        if (!(SPanel & GYRO_RESET)) {       /* if not resetting zeroes ... */
        /* If main power ON, run motors (and closed loop attitude). */
                if (SPanel & PWR_SWITCH){
                        if (SPanel & PropCtl) FPropCtl(); /* calculate P of PID attitude */
                        else for (i=0; i<3; i++) propcmd[i] = 0;
                        if (SPanel & DerFdBk) FDerFdBk(); /* calculate D of PID attitude */
                        else for (i=0; i<3; i++) derivcmd[i] = 0;
                        if (SPanel & DepthCtl) FDepthCtl(); /* calculate depth controller */
                        else for (i=0; i<3; i++) depthcmd[i] = 0;
                        if (SPanel & IntgrlCtl) FIntgrlCtl(); /* calculate I of PID attitude */
                        else for (i=0; i<2; i++) intcmd[i] = 0;
                        FireMotors();       /* calculate and output thruster commands */
                        }
                }
        else { /* Reset gyro (and pgauge) adjusts */
                for (i=0; i<3; i++) {           /* clear old values */
                        thc[i] = rhc[i] = 0x80;
                        gyroadj[i] = 0;
                        }
                FireMotors();
                RdAtoD();           /* read in A/D */
                for (i=0; i<3; i++) {           /* calculate new gyro offsets */
                        gyroadj[i] = gyros[i] - 0x800;
                        if ((i == 0) || (i == 2)) gyroadj[i] = -(gyroadj[i]);
                        }
                /* calculate new pressure gauge adjustment values */
                if (SPanel & PGADJ1) pgadj[0] = adnum[1] - adnum[0];
                if (SPanel & PGADJ4) pgadj[1] = adnum[3] - adnum[2];
                if (SPanel & PGADJ23) pgadj[2] = adnum[2] - adnum[1];
                }

        /* Data Acquistion Stuff */
        if (changes & TKDATA) {
                if (SPanel & TKDATA) { /* starting data acquisition (upclock) */
                         ftime(&etime1); /* record start time */
                         count = 0;
                         }
                else { /* end of data acquisistion (downclock) */
                        ftime(&etime2); /* record end time and calculate duration */
                        time1 = etime1.time + (etime1.millitm/1000.0);
                        time2 = etime2.time + (etime2.millitm/1000.0);
                        deltat = time2 - time1;
                        onscreen(23,5,0,"deltat: %8.2lf\t\tcount: %5ld",deltat,count);
                        p = g = 0;          /* set-up for data dump */
                        TX_Count(NULL);
                        DumpFlg = TRUE;
                        outp(CNTRL3,(BIT7 | BOFF)); /* disable main power (thrusters) */
                        TX_Stats(NULL);
                        }
                }

        if (SPanel & TKDATA) { /* write data to array during acquisition */
                for (i=0; i<4; i++) data[count][i] = pgauge[i];
                for (i=0; i<3; i++) data[count][i+4] = gyros[i];
                count++;
                }
        } /* end of dumpflg loop */
        if (DumpFlg) {
                if ((p >= (count+1)) && (g >= (count+1))) DumpFlg = FALSE;
                for (i=0; i<1000; i++);
                TX_Stats(NULL);
                }

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

void AstroInit() {
        /* ASTRO initialization routine. */
        register   i;

        /* Initialize the 8255's on the I/O protoboard */
        outp(CNTRL1, 0x80);
        outp(CNTRL2, 0x80);
        outp(CNTRL3, 0x92);

        /* Zero out all the state variables and toggle flags */
        for (i = 0; i < 6; i++) {
                if (i < 3){
                        thc[i] = rhc[i] = 0x80;
                        gyros[i] = 0x800;
                        gyroadj[i] = pgadj[i] = 0;
                        propgain[i] = 9;
                        propcmd[i] = derivcmd[i] = depthcmd[i] = 0;
                        depthgain[i] = 40;
                        }
                Motors[i] = 0x00;
                }

        /* Zero out all other variables. */
        SPanel = Iris = 0x00;
        PwrChg = IrisChg = DumpFlg = FALSE;
        LastTHC = LastRHC = LastSPanel = 0;
        rategain[0] = 7;
        rategain[1] = 8;
        rategain[2] = 10;
        gammad = betad = gammac = betac = 0;
        intcmd[0] = intcmd[1] = 0;
        intgain[0] = 7;
        intgain[1] = 8;

        /* Print Onscreen Headers to Monitor (mostly for debugging) */
/*      onscreen(11, 5, 0, "Motors:    LSX USX LPX UPX  LY  UY  SZ  PZ");
        onscreen(9,5,0,"3-Axis Rate Gyros: ");
        onscreen(7,5,0,"Pressure Gauge (digital): ");
        onscreen(14,5,0,"DepthGain: ");
        onscreen(15,5,0,"PropGains: ");
        onscreen(16,5,0,"RateGains: ");
        onscreen(17,5,0,"IntglGain: ");*/
        onscreen(22,5,0,"Desired Angles: ");
        onscreen(22,20,0,"%4d\t%4d",gammad,betad);
/*      for (i=0; i<3; i++) {
                onscreen(15,20+4*i,0,"%4d",propgain[i]);
                onscreen(16,20+4*i,0,"%4d",rategain[i]);
                onscreen(14,20+4*i,0,"%4d",depthgain[i]);
                }
        for (i=0; i<2; i++) onscreen(17,20+4*i,0,"%4d",intgain[i]);*/
}
```

```
/* ASTROFN1.C */
/* Last Modified: 3/6/90 by Wendy Power */
/* This program contains all the nonchanging functions called in
        ASTRO's software (those dealing with motors, cameras, and
        pneumatics. This includes thruster calculations. */

#include "..\pivecs.h"
#include "pvastro.h"

void FireCamera() { /* control light aperture on underwater video camera */
        /* Output light apature control 4-bit word. */
        outp(PORT2B,Iris);
        IrisChg = FALSE;
}

void FirePneu() { /* controls pneumatics */
        if (PwrChg) {
                /* Main power On/Off switch. */
                if (SPanel & PWR_SWITCH) outp(CNTRL3, (BIT7 | BON) );
                else if (!(SPanel & PWR_SWITCH)) outp(CNTRL3, (BIT7 | BOFF) );
                }
        }

void FireMotors() {
/* Outer loop of thruster calculation routines. Calls JetSelect
        and EngageMotors. Displays thruster output commands and saves
        old data for the next set of calculations. */

  char        MotorCmds[8], TmpMotors[8];
  Byte          signs, MotorFlips, shift;
  static Byte   OldSigns;
  register    i;

  signs = JetSelect(MotorCmds);
  MotorFlips = signs^OldSigns;
        shift = 0x01;

        /* Display motor commands (for debugging only) */
/*   for (i = 0; i < 8; i++) {
        onscreen(12, (20+5*i), 0, "%2d ", MotorCmds[i]);
                if ( (signs & shift) == 0) onscreen(12, (19+5*i), 0, "+");
                else onscreen(12, (19+5*i), 0, "-");
                shift = shift << 1;
                } */

  if (MotorFlips != 0) {
    for (i = 0; i < 8; i++) {
       if (MotorFlips&Bit[i]) TmpMotors[i] = 0;
        else TmpMotors[i] = MotorCmds[i];
    }
      EngageMotors(TmpMotors, signs);
    for (i = 0; i < DELAY; i++);
  }

   EngageMotors(MotorCmds, signs);
  OldSigns = signs;
}

Byte JetSelect(motor)
        /* This function takes the THC and RHC commands and distributes
                them to the appropriate thruster. It also determines the
                direction of the thrust. */

  char *motor;
```

```
{
  register   i;
  char       StripTHC[3], StripRHC[3];
  Byte       signs = 0;

  /*    Strip the HC readings to +/- 0x00-0x0F    */
  for (i = 0; i < 3; i++) {
     if ( (StripTHC[i] = (thc[i] >> 3) - 16) < 0) StripTHC[i]++;
     if ( (StripRHC[i] = (rhc[i] >> 3) - 16) < 0) StripRHC[i]++;
  }

        /* write commands to motor array (X,Y,Z) */
  /* add in closed-loop depth control commands here */
  for ( i = 0; i <= 3; i++ ) motor[i] = StripTHC[0] + depthcmd[0];  /* X */
  for ( i = 4; i <= 5; i++ ) motor[i] = StripTHC[1] + depthcmd[1];  /* Y */
  for ( i = 6; i <= 7; i++ ) motor[i] = StripTHC[2] + depthcmd[2];  /* Z */

        /* write commands to motor array (Roll, Pitch, Yaw) */
        /* add in closed-loop attitude controller commands here */
        /* Roll */
  motor[4] += (propcmd[0] + derivcmd[0] + intcmd[0] + StripRHC[0]);
  motor[7] += (propcmd[0] + derivcmd[0] + intcmd[0] + StripRHC[0]);
  motor[5] -= (propcmd[0] + derivcmd[0] + intcmd[0] + StripRHC[0]);
  motor[6] -= (propcmd[0] + derivcmd[0] + intcmd[0] + StripRHC[0]);

        /* Pitch */
        motor[1] += (propcmd[1] + derivcmd[1] + intcmd[1] + StripRHC[1]);
  motor[3] += (propcmd[1] + derivcmd[1] + intcmd[1] + StripRHC[1]);
  motor[0] -= (propcmd[1] + derivcmd[1] + intcmd[1] + StripRHC[1]);
  motor[2] -= (propcmd[1] + derivcmd[1] + intcmd[1] + StripRHC[1]);

        /* Yaw */
  motor[0] += (propcmd[2] + derivcmd[2] + StripRHC[2]);
  motor[1] += (propcmd[2] + derivcmd[2] + StripRHC[2]);
  motor[2] -= (propcmd[2] + derivcmd[2] + StripRHC[2]);
  motor[3] -= (propcmd[2] + derivcmd[2] + StripRHC[2]);

  for (i = 0; i < 8; i++) { /* set direction bits */
    if (motor[i] < 0) {
       signs |= Bit[i];
       motor[i] = -motor[i];
    }
    if (motor[i] > TMAX) motor[i] = TMAX; /* limit max thruster values */
  }
  return(signs);
}

void EngageMotors(motormags, motorsgns)
        /* Routine configures and sends thruster commands and signs for
                and through the 8255 ports. */

  char*  motormags;
  Byte   motorsgns;
{
  Byte  mtrout[4];

  /* prepare output words 4 for the motors and 1 with directions */

  /* port 300H with UY and LY */
  mtrout[0]  = motormags[5] << 4;
  mtrout[0] |= motormags[4];

  /* port 301H with UPX and LPX */
  mtrout[1]  = motormags[3] << 4;
```

```
    mtrout[1] |= motormags[2];

   /* port 302H with USX and LSX */
    mtrout[2]  = motormags[1] << 4;
    mtrout[2] |= motormags[0];

   /* port 304H with PZ and SZ */
    mtrout[3]  = motormags[7] << 4;
    mtrout[3] |= motormags[6];

   /* output all values */
    outp(PORT1A, mtrout[0]);
    outp(PORT1B, mtrout[1]);
    outp(PORT1C, mtrout[2]);
    outp(PORT2A, mtrout[3]);
    outp(PORT2C, motorsgns);
}
```

```
/* ASTROFN2.C */
/* Last Modified: 3/6/90 by Wendy Power */
/* This program contains all the evolving functions called in
        ASTRO's software (those dealing with sensors and  control
        calculations. These include leak, gyro, and pressure sensors. */

#include "..\pivecs.h"
#include "pvastro.h"
#include <math.h>

void RdAtoD(){
        /* Routine to read a/d363 7 ports assigned to the pressure gauges
                (0,1,2,3) and rate gyros (4,5,6).  Also puts data into 12-bit
                datawords and adjusts them by the appropriate zero values. */

        register  i;
        Byte                adstat, tmp, wait, mask;
        Byte                inadnum[14];
        int                         temp;
        long                        x, y, q, q2;

        mask = 0;
        /* Set 8255-3-C 3 MSB Mask */
        if (SPanel & PWR_SWITCH) mask = PWR_SWITCH;

        /* Read 12-bit a/d (363) input ports 0 through 7 */
        for (i=0; i<7; i++){
                outp(PORT3C, i | mask);
                outp(CNTRL3,HTGGLE);
                outp(CNTRL3,LTGGLE);
                while((adstat = (0x01 & inp(PORT3B))) != 0x00) wait++;
                inadnum[2*i] = inp(PORT3B);
                inadnum[2*i+1] = inp(PORT3A);
                }
        /* Compress data for use on ASTRO (4 12-bit datawords) */
        for (i=0; i<7; i++)
                adnum[i] = ((tmp = inadnum[2*i]) << 4) + inadnum[2*i+1];

        /* Adjust pressure gauge values */
        pgauge[0] = adnum[0] + pgadj[0] + pgadj[2];
        pgauge[1] = adnum[1] + pgadj[2];
        pgauge[2] = adnum[2];
        pgauge[3] = adnum[3] - pgadj[1];
/*      for (i=0; i<4; i++) onscreen(7,35+i*7,0,"%4ld",pgauge[i]);*/

        /* Adjust and limit gyro values (and flip R and Y) */
        for (i=0; i<3; i++) {
                if (gyroadj[i] >= 0) {
                        if (adnum[i+4] <= gyroadj[i]) gyros[i] = 0x000;
                        else gyros[i] = adnum[i+4] - gyroadj[i];
                        }
                else gyros[i] = adnum[i+4] - gyroadj[i];
                if ((i == 0) || (i == 2)) {
                        temp = -(gyros[i]) + 0xFFF;
                        gyros[i] = temp;
                        }
                gyros[i] &= 0xfff;
                }

        /* Calculate current angles (roll and pitch) */
        y = pgauge[0] - pgauge[1];   /* pg1 - pg2 */
        x = pgauge[3] - pgauge[2];   /* pg4 - pg3 */
        gammac = atan2(y,x);                 /* roll angle */
        q = ((pgauge[2] + pgauge[3]) / 2) - ((pgauge[0] + pgauge[1]) / 2);
        if ( ( ( (gammac*180.0/pi) > 80.0) && ( (gammac*180.0/pi) <100.0) ) ||
                ( ( (gammac*180.0/pi) <-80.0) && ( (gammac*180.0/pi) >-100.0) ) )
                        q2=y/sin(gammac);
        else q2 = x / cos(gammac);
        betac = atan2(-q,q2);                   /* pitch angle */
        onscreen(22,45,0,"%8.2lf\t%8.2lf",(gammac*180.0/pi),(betac*180.0/pi));
        cosgc = cos(gammac);
        singc = sin(gammac);
}

void FPropCtl() {
        /* This routine calculates the proportional closed loop
                attitude Torques (commanded rotations) */

        register        i;
        double                  cosgd, singd, cdiffb,cdiffg,sdiffg,sdiffb;

        /* Calculate cosines and sines of gammac, betac, gammad, betad
                in the combinations necessary for the simplified control eqns */
        cdiffb = cos(betad-betac);
        cdiffg = cos(gammad-gammac);
        sdiffg = sin(gammad-gammac);
        sdiffb = sin(betad-betac);
        cosgd = cos(gammad);
        singd = sin(gammad);

        /* Calculate thetae and thetaed from the correction rotation matrix */
        thetae = acos((cdiffb*(1.0+cdiffg)+cdiffg-1.0)/2.0);
        thetaed = thetae*180.0/pi;

        /* Calculate the kvector[i] from the correction rotation matrix */
        if ((thetaed < 1.0) && (thetaed > -1.0)) sintemp = 1.0;
        else sintemp = 1.0 / (2.0 * sin(thetae));
        kvector[0] = sintemp*(-(sdiffg)*(cdiffb+1.0));
        kvector[1] = sintemp*(-(sdiffb)*(cosgc+cosgd));
        kvector[2] = sintemp*(sdiffb*(singd+singc));

        /* Calculate and limit the proportional feedback */
        for (i=0; i<3; i++) {
                propcmd[i] = -(kvector[i]*thetaed)*propgain[i]/10.0;
                if (propcmd[i] >= 15) propcmd[i] = 15;
                else if (propcmd[i] <= -15) propcmd[i] = -15;
/*              onscreen(19,20+i*6,0,"%4d",propcmd[i]);  */
                }
}

void FDerFdBk() {
        /* This routine calculates the derivative (rate) feedback
                contributions to the commanded rotations */

        register  i;
        int             tmpgyros[3];

        /* Calculate and limit the derivative commands */
        for (i=0; i<3; i++) {
                derivcmd[i] = (-(((int) gyros[i]) - 2048)) * rategain[i]/250.0;
                if (derivcmd[i] >= 15) derivcmd[i] = 15;
                else if (derivcmd[i] <= -15) derivcmd[i] = -15;
/*              onscreen(20,20+i*6,0,"%4d",derivcmd[i]);  */
                }
}

void FIntgrlCtl() {
        /* This routine calculates the integral term of the PID controller
```

```
                for Roll and Pitch */

        register  i;
        double              eangle[2];

        /* Calculate and limit the integral commands */
        eangle[0] = (gammad - gammac) * 180.0 / pi;
        eangle[1] = (betad - betac) * 180.0 / pi;
        for (i=0; i<2; i++) {
                esum[i] += (eangle[i]) * tsample;
                if (esum[i] >= 200.0) esum[i] = 200.0;
                else if (esum[i] <= -200.0) esum[i] = -200.0;
                intcmd[i] = esum[i] * intgain[i] / 60;
                if (intcmd[i] >= 15) intcmd[i] = 15;
                else if (intcmd[i] <= -15) intcmd[i] = -15;
                onscreen(21,20+i*6,0,"%4d",intcmd[i]);
                }
}

void FDepthCtl() {
        /* This routine provides proportional depth control */
        register  i;

        /* Calculate depths */
        cdepth = 0;                                             /* clear current depth */
        for (i=0; i<4; i++) cdepth += pgauge[i];
        cdepth = cdepth / 4;                    /* current depth */

        /* Calculate and limit depth control commands */
        zvector[0] = -(sin(betac));
        zvector[1] = cos(betac) * singc;
        zvector[2] = cos(betac) * cosgc;
        for (i=0; i<3; i++) {
                depthcmd[i] = (depthgain[i] * (cdepth-ddepth) * zvector[i]) / 75.0;
                if (depthcmd[i] >= 15) depthcmd[i] = 15;
                else if (depthcmd[i] <= -15) depthcmd[i] = -15;
/*              onscreen(18,20+i*6,0,"%4d",depthcmd[i]); */
                }
}
```

```
/* ASTROMSG.C */
/* Last Modified: 3/6/90 by Wendy Power */
/* This program contains all of the messages used for ASTRO.
         Some of these include small amounts of processing or
         data distribution. */
#include "..\pivecs.h"
#include "..\pvdata.h"
#include "pvastro.h"
#include "..\pvsecs.msg"

MsgHandler RX_Camera(msg) /* Receive camera aperture command */
   MsgPtr  msg;
{
   Iris = (*msg->data) & 0x0F;
   IrisChg = TRUE;
   return( OK );
}

MsgHandler ShutDown(msg)
   MsgPtr  msg;
{
   register i;

   SPanel = 0x00;
   for (i = 0; i < 3; i++) thc[i] = rhc[i] = 0x80;
   PwrChg = FALSE;
   FirePneu();
   FireMotors();
   return( OK );
}

MsgHandler RX_THC(msg)          /* receive translational hand controller command */
   MsgPtr msg;
{
   register i;
   BytePtr datptr = msg->data;

   for (i = 0; i < 3; i++) {
/*       if (thc[i] != *datptr) onscreen(7, (20+5*i), 0, "%4d", (*datptr-0x80)); */
      thc[i] = *datptr++;
   }
         if (!(SPanel & NOHCCMD)) {
            if (SPanel & PWR_SWITCH) pvRequest( TXTHC );
         }
   LastTHC = MsgNum;
   return( OK );
}

MsgHandler RX_RHC(msg)          /* receive rotational hand controller command */
   MsgPtr msg;
{
   register i;
   BytePtr datptr = msg->data;

         for (i = 0; i < 3; i++) {
/*       if (rhc[i] != *datptr) onscreen(9, (20+5*i), 0, "%4d", (*datptr)); */
      rhc[i] = *datptr++;
   }
         if (!(SPanel & NOHCCMD)) {
                  if (SPanel & PWR_SWITCH) pvRequest( TXRHC );
                  }
         LastRHC = MsgNum;
         return( OK );
}
```

```
MsgHandler RX_SPanel(msg)          /* receive switch panel commands */
   MsgPtr msg;
{
  register                      i;

   changes = SPanel^(*msg->data);
  SPanel = *msg->data;
  if (DumpFlg) SPanel & 0x00;
  if (changes & PWR_SWITCH) PwrChg = TRUE; else PwrChg = FALSE;
/*  if (changes & PropCtl) onscreen(19,5,0,"                    ");
  if (changes & DerFdBk) onscreen(20,5,0,"                    ");
  if (changes & DepthCtl) onscreen(18,5,0,"                   ");
  if (changes & IntgrlCtl) onscreen(21,5,0,"                  ");
          */
   if (changes & IntgrlCtl) for (i=0; i<2; i++) esum[i] = 0;
  if (changes & DepthCtl) {
                    ddepth = 0;
                    for (i=0; i<4; i++) ddepth += pgauge[i];
                    ddepth = ddepth / 4;
                    }
  pvRequest( TXSPANEL );
  LastSPanel = MsgNum;
  return( OK );
}

MsgHandler TX_Stats(msg)           /* transmit astro status to secs */
   MsgPtr msg;
{
          Byte      array[2];

          array[0] = RXSTATS;
          array[1] = (inp(PORT3B) & LKMASK) | DumpFlg;
          pvSend(array,2);
          return( OK );
}

MsgHandler RX_Euler(msg)           /* receive new desired orientation angles */
   MsgPtr msg;
{
   register           i;
          int                       tmp[4];
   BytePtr datptr = msg->data;

   for (i=0; i<4; i++) tmp[i] = *datptr++;
          for (i=0; i<2; i++) {
                    if (tmp[i+2] > 0) tmp[i] = -(tmp[i]);
              onscreen(22,20+5*i,0,"%4d",tmp[i]);
                    }
          gammad = tmp[0]*pi/180.0;
          betad = tmp[1]*pi/180.0;
          for (i=0; i<2; i++) esum[i] = 0;
          return( OK );
}

MsgHandler RX_PGain(msg)           /* receive new proportional gains */
   MsgPtr msg;
{
   register i;
   BytePtr datptr = msg->data;

   for (i=0; i<3; i++) propgain[i] = *datptr++;
  return( OK );
}
```

```
MsgHandler RX_RGain(msg)          /* receive new derivative (rate) gains */
   MsgPtr msg;
{
   register i;
   BytePtr datptr = msg->data;

  for (i=0; i<3; i++) rategain[i] = *datptr++;
  return( OK );
}

MsgHandler TX_PGauge(msg)         /* transmit 12-bit pressure gauge readings */
   MsgPtr msg;
{
         register  i;
         Byte              array[8];
         Byte              tmp;

         array[0] = RXPGAUGE;
         if (DumpFlg) for (i=0; i<4; i++) pgauge[i] = data[p][i];
         for (i=0; i<2; i++) {
                  array[2*i+2] = pgauge[i] & 0xFF;
                  array[2*i+1] = (pgauge[i] >> 4) & 0xF0;
                  }
         array[5] = pgauge[2] & 0xFF;
         array[6] = ((pgauge[2] >> 4)&0xF0) + ((pgauge[3] >> 8)&0x0F);
         array[7] = pgauge[3] & 0xFF;
         if (DumpFlg) p++;
         pvSend(array,8);
         return( OK );
}

MsgHandler TX_Gyros(msg)          /* transmit 12-bit rate gyro data */
   MsgPtr msg;
{
         register  i;
         Byte              array[7];

         array[0] = RXGYROS;
         if (DumpFlg) for (i=0; i<3; i++) gyros[i] = data[g][i+4];
         for (i=0; i<3; i++) {
                  array[2*i+2] = gyros[i] & 0xFF;
                  array[2*i+1] = (gyros[i] >> 4) & 0xF0;
                  }
         if (DumpFlg) g++;
         pvSend(array,7);
         return( OK );
}

MsgHandler TX_Count(msg)          /* transmit number of samples in datafile */
   MsgPtr msg;
{
         Byte     array[3];

         array[0] = RXCOUNT;
         array[1] = (count >> 8) & 0xFF;
         array[2] = count & 0xFF;
         pvSend(array,3);
         return( OK );
}

MsgHandler RX_IGain(msg)          /* receive new integral gains */
   MsgPtr msg;
{
   register i;
   BytePtr datptr = msg->data;

   for (i=0; i<2; i++) intgain[i] = *datptr++;
  return( OK );
}

MsgHandler RX_DpGain(msg)          /* receive new depth gains */
   MsgPtr msg;
{
          register              i;
   BytePtr datptr = msg->data;

          for (i=0; i<3; i++) depthgain[i] = *datptr++;
   return( OK );
}
```

Appendix B: SECS Software

```
/* WPVSECS */
/* Last Modified: 3/6/90 by Wendy Power */
/* This file is used to compile and link the control station software */

wpvsecs.obj: ..\pivecs.h pvsecs.h ..\pvastro.msg wpvsecs.c
        cl /c /Od /Zp /Zi wpvsecs.c

secsfuns.obj: ..\pivecs.h pvsecs.h secsfuns.c
        cl /c /Od /Zp /Zi secsfuns.c

onscolor.obj: ..\pivecs.h pvsecs.h onscolor.c
        cl /c /Od /Zp /Zi onscolor.c

secsmsgs.obj: ..\pivecs.h pvsecs.h ..\pvastro.msg secsmsgs.c
        cl /c /Od /Zp /Zi secsmsgs.c

wpvsecs.exe: wpvsecs.obj secsmsgs.obj secsfuns.obj onscolor.obj
        link /NOD /CO  $**, wpvsecs.exe,,SLIBCE+GFCS+GFS+..\PIVECS
```

```
/* PVSECS.MSG */
/* Last Modified: 3/6/90 by Wendy Power */

#ifndef SECSMSGS
#define SECSMSGS

/* This is a listing of the message headers as used by SECS */

/*********************** Recognized Message List **************************/
#define COMTEST         0x00    /* Msg 0, No Data             */
#define COMAOK          0x08    /* Msg 1, " "                 */
#define SHUTDN          0x10    /* Msg 2, " "                 */
#define STOP            0x18    /* Msg 3, " "                 */

#define  TXTHC          0x50    /* Msg 10, No Data            */
#define  TXRHC          0x58    /* Msg 11, " "    */
#define TXSPANEL        0x60    /* Msg 12, " "    */
#define TXEULER         0x68    /* Msg 13, " "    */
#define TXPGAIN         0x70    /* Msg 14, " "    */
#define TXRGAIN         0x78    /* Msg 15, " "    */
#define TXIGAIN         0x80    /* Msg 16, " "    */
#define TXDPGAIN        0x88    /* Msg 17, " "    */

#define  RXSTATS        0xB1    /* Msg 22, 1 Data Byte  */
#define RXPGAUGE        0xBF    /* Msg 23, 7 Data Bytes */
#define RXGYROS         0xC6    /* Msg 24, 6 Data Bytes */
#define RXCOUNT         0xCA    /* Msg 25, 2 Data Bytes */

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

/* The message structure is as follows:
#define  MSGname                    0x??    /* Msg # (0-31) and # of data bytes (0-7)

The number of data bytes should be 0 if transmitting, and the appropriate
number if receiving.  ?? is a 2 digit hex representation of the message
number (most significant 5 bits) and the number of data bytes (3 least
significant bits).
Example:  receiving message number 24 with 6 data bytes
        24 = 11000 in bits, 3 = 110
        combining: 11000 110 -> 1100 0110 -> hex = C6                */
```

```
/* PVSECS.H */
/* Last Modified: by Wendy Power */
/* This routine contains all the message handler and header
        definitions, as well as all the global and fixed definition
        variables for SECS. */

#ifndef PIVECS
#include "..\pivecs.h"
#endif
#ifndef SECSMSGS
#include "..\pvsecs.msg"
#endif
#ifndef SECS
#define SECS

extern HandlerFunc          BadMsg, ShutDown, ComCheck, ComAOK;
extern HandlerFunc          RX_Stats, RX_PGauge, RX_Gyros, RX_Count;
extern HandlerFunc          TX_THC, TX_RHC, TX_SPanel;
extern HandlerFunc          TX_Euler, TX_PGain, TX_RGain, TX_IGain, TX_DpGain;

static Handlers     SecsHandlers =
                    {ComCheck, ComAOK,  ShutDown,   BadMsg,
                    BadMsg,   BadMsg,   BadMsg,     BadMsg,
                    BadMsg,   BadMsg,   TX_THC,     TX_RHC,
                    TX_SPanel, TX_Euler, TX_PGain,   TX_RGain,
                    TX_IGain, TX_DpGain, BadMsg,     BadMsg,
                    BadMsg,   BadMsg,   RX_Stats,   RX_PGauge,
                    RX_Gyros, RX_Count, BadMsg,     BadMsg,
                    BadMsg,   BadMsg,   BadMsg,     BadMsg};

static Headers      SecsMsgs =
                    {COMTEST, COMAOK,  SHUTDN,     STOP,
                    BADMSG,   BADMSG,  BADMSG,     BADMSG,
                    BADMSG,   BADMSG,  TXTHC,      TXRHC,
                    TXSPANEL, TXEULER, TXPGAIN,    TXRGAIN,
                    TXIGAIN,  TXDPGAIN, BADMSG,    BADMSG,
                    BADMSG,   BADMSG,  RXSTATS,    RXPGAUGE,
                    RXGYROS,  RXCOUNT, BADMSG,     BADMSG,
                    BADMSG,   BADMSG,  BADMSG,     BADMSG};

static Byte    SecsHiPri = 4;

/* Global Variables (can be called by all routines). */
Byte                Motors[6], thc[3], rhc[3], sndflg[3], sign[2];
Byte                Stats, Switches, Changed_THC, Changed_RHC, Iris;
Byte                SPanel, StateChange, Display, StatChg, nobeep;
int                 rategain[3], propgain[3], eulerdes[2], intgain[2];
int                 depthgain[3];
char                AdjTHC[3], AdjRHC[3];
unsigned long       LastStat, LastPGauge, LastGyros, count, j, p, g;
unsigned long       data[1000][7];
long                pgauge[4], gyros[3];

/* Name functions */
void                SecsInit(), ReadTHC(), ReadRHC(), ReadPanel(), Leak();
void                InRateGain(), InPropGain(), InEulerDes(), InIntGain();
void                InDepGain();
void                onscrnLR(), onscrnY(), onscrnLB(), onscrnMG();
void                onscrnWH(), onscrnGY(), onscrnBR();

/* List and define Fixed Variables */
#define  WLEN           0x05
#define DELAY           0x800
/* define SECS 8255 ports */
```

```
#define ADPORTA                 0x3E0
#define ADPORTB                 0x3E1
#define ADPORTC                 0x3E2
#define ADCNTRL                 0x3E3
/* Define switch names */
#define TKDATA          0x01
#define   LK_CNTRL      0x04
#define   LK_CAMERA     0x02
#define   LK_BATBOX     0x08
#define LEAKS           0x0E
#define DATADUMP        0x01

#endif
```

```
/* WPVSECS.C */
/* Last Modified: 3/6/90 by Wendy Power */
/* This program runs the main loop in the SECS (control station)
          software.  It is responsible for obtaining all handcontroller,
          switch, and keyboard commands.  It will also eventually be the
          sole input source for Sulu. */

#include "..\pivecs.h"
#include "pvsecs.h"
#include "..\pvastro.msg"
#include <gf.h>
#include <time.h>
#include <stdio.h>

unsigned    getkey();

main() {
          unsigned         inkey;
          unsigned long   CurrMsg = 0;
          register        i;
          char                              outname[20];
          FILE                              *fpout;

          /* Initialize Pivecs and Secs Computer System */
   pvInitCom( COM1, 9600, P_ODD, 1);
   pvInitMsg( SecsHandlers, SecsMsgs, SecsHiPri);
   SecsInit();

          /* Begin Main Driver Loop */
  while (TRUE) {

   /* Keyboard input processing: ESC and camera apature control. */
   if (kbhit()) {
             if ( (inkey = getkey()) == ESC) break;
             else if (inkey == 'd') {
                    if (Iris != 0x00) Iris--;
                    onscreen(15,20,0, "%02X", Iris);
                    TX_Camera(NULL);
                    }
             else if (inkey == 'i') {
                if (Iris != 0x0F) Iris++;
                  onscreen(15,20,0, "%02X", Iris);
               TX_Camera(NULL);
                    }
             else if (inkey == 'o') nobeep = !(nobeep);
             else if (inkey == 's') {
                    Display = !(Display);
                    if (!(Display)) {
                              onscreen(5,16,0,"              ");
                              onscreen(7,16,0,"              ");
                              onscreen(9,16,0,"    ");
                              onscreen(11,20,0,"                          ");
                              onscreen(13,20,0,"                          ");
                              }
                    }
             else if (inkey == 'e') InEulerDes();
                    else if (inkey == 't') TX_Euler(NULL);
             else if (inkey == 'p') InPropGain();
             else if (inkey == 'r') InRateGain();
                    else if (inkey == 'j') InIntGain();
                    else if (inkey == 'l') InDepGain();
                    else if (inkey == 'a') {
                    onscrnY(17,5,0,"Output filename: ");
                    scanf("%12s",outname);
```

```
				fclose(fpout);
			fpout = fopen(outname,"w+");
			}
	}

CurrMsg = pvRecv();
if (!(Stats & DATADUMP)) {
	/* Get Stat uplink */
		if ((CurrMsg - LastStat) > WLEN){
			pvRequest( TXSTATS );
			LastStat = CurrMsg;
			}
		}
	if (Display) {
		/* Get PGauge uplink */
		if ((CurrMsg - LastPGauge) > WLEN){
			pvRequest( TXPGAUGE );
			LastPGauge = CurrMsg;
			}
		/* Get Gyros uplink */
		if ((CurrMsg - LastGyros) > WLEN){
			pvRequest( TXGYROS );
			LastGyros = CurrMsg;
			}
		}

	/* Check for leaks and sound beep if leaks detected */
	if (StatChg) Leak();
	for (i=0; i<3; i++)		if ((sndflg[i]) && (nobeep)) printf("^G\n");

	if ((SPanel & TKDATA) && (Display)){
		Display = FALSE;
	  onscreen(5,16,0,"               ");
	  onscreen(7,16,0,"               ");
	   onscreen(9,16,0,"   ");
	  onscreen(11,20,0,"                        ");
	  onscreen(13,20,0,"                        ");
		}

	/* Data Acquisition */
	if (StateChange & TKDATA) {
		if (SPanel & TKDATA) { /* up clock pulse */
			fprintf(fpout,"\n%4d\t%4d",eulerdes[0],eulerdes[1]);
			TX_Euler(NULL);
			j = p = g = 0;
			}
		else { /* down clock pulse */
			for (i=0; i<2; i++) {
				if (eulerdes[i]) eulerdes[i] = 0;
				onscreen(20,30+i*8,0,"%4d",eulerdes[i]);
				}
			TX_Euler(NULL);
			onscreen(13,5,0,"                    ");
			}
		}

	if (!(Stats & DATADUMP)) {
		if (StatChg & DATADUMP) {
			fprintf(fpout,"\n%4ld\n",count);
			for (j=0; j<count; j++) {
				for (i=0; i<7; i++) fprintf(fpout,"%4ld\t",data[j][i]);
				fprintf(fpout,"\n");
				onscreen(17,60,0,"%4ld",j);
				}
```

```
                                fclose(fpout);
                                onscreen(17,5,0,"                              ");
                                }
                        }

            /* Read Handcontrollers */
            ReadTHC();
            ReadRHC();
            /* Read Switch Panel and Display */
            ReadPanel();

    } /* End main driver loop */

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

} /*  End program */

void SecsInit() {

            /* Control station initialization */
            register i;

            /* initialize SECS's 8255 */
            outp (ADCNTRL,0x92);

            /* Initialize RHC, THC, and motor arrays */
            for (i = 0; i < 6; i++) {
                        if (i < 3){
                                thc[i] = rhc[i] = 0x80;
                                sndflg[i] = 0;
                                propgain[i] = 9;
                                depthgain[i] = 40;
                                }
                        Motors[i] = 0x00;
                        }

            /* Set to zero all variables */
            SPanel = Stats = 0x00;
            Iris = count = j = 0x00;
            LastStat = LastPGauge = LastGyros = 0x00;
            nobeep = TRUE;
            rategain[0] = 7;
            rategain[1] = 8;
            rategain[2] = 10;
            for (i=0; i<2; i++) {
                        eulerdes[i] = 0;
                        intgain[i] = 7+i;
                        }

            /* Initialize handcontroller readings and set offset */
            for (i = 0; i < 2*DELAY; i++);
            ReadTHC();
            for (i = 0; i < DELAY; i++);
            ReadRHC();
            for (i = 0; i < 3; i++) {
               AdjTHC[i] = 0x80 - thc[i];
               AdjRHC[i] = 0x80 - rhc[i];
            }
            AdjTHC[1] = ~AdjTHC[1];

            /* Onscreen Headers */
            onscrnWH(5,5,0, "THC:");
            onscrnWH(7,5,0, "RHC:");
```

```
        onscrnWH(9,5,0, "Switches:");
        onscreen(15,5,0,"Aperature:");
        onscreen(20,5,0,"Desired Euler Angles: ");
        onscreen(21,5,0,"DepthGain:");
        onscreen(22,5,0,"Prop Gains: ");
        onscreen(23,5,0,"Rate Gains: ");
        onscreen(24,5,0,"Intgrl Gain: ");
        for (i=0; i<3; i++) {
                onscreen(23,30+i*8,0,"%4d",rategain[i]);
                onscreen(22,30+i*8,0,"%4d",propgain[i]);
                onscreen(21,30+i*8,0,"%4d",depthgain[i]);
                }
        for (i=0; i<2; i++) {
                onscreen(20,30+i*8,0,"%4d",eulerdes[i]);
                onscreen(24,30+i*8,0,"%4d",intgain[i]);
                }
        onscrnY(5,40,0,"Don't forget to zero pgauges and gyros!");
        onscreen(11,5,0,"Gyros: ");
        onscreen(13,5,0,"PGauges: ");
}
```

```
/* SECSFUNS.C */
/* Last Modified: 3/6/90 by Wendy Power */
/* Contains all functions called in SECS software */
#include "..\pivecs.h"
#include "pvsecs.h"
#include <gf.h>
#include <asiports.h>

void ReadPanel()  /* Read switch panel */
{
        /* Read Switch Panel */
        StateChange = SPanel; /* save old switch panel */
        SPanel = inp(ADPORTB);
        if (Display) onscrnWH(9,17,0,"%2X",SPanel);
        StateChange ^= SPanel; /* shows which switches have changed */
}

void ReadTHC()   /* read translational hand controllers */
{
  register  i;
  short     tmp;

  for (i = 0; i < 3; i++) {
        outp(ADPORTC, i);
        outp(ADCNTRL, 0x09);
        outp(ADCNTRL, 0x08);
        outp(ADCNTRL, 0x0B);
        tmp = inp(ADPORTA) + AdjTHC[i];
        outp(ADCNTRL, 0x0A);

        /* Keep the HC readings in bounds */
        if (tmp > 0xFF) tmp = 0xFF;
          else if (tmp < 0x00) tmp = 0x00;

        /* Flip the y reading to keep the coordinates right handed */
        if (i == 1) tmp = ~tmp;
        if ((thc[i] != tmp) && Display)
                onscrnWH(5, (17+6*i), 0, "%4d ", (Byte) tmp-0x80);
        thc[i] = tmp;
  }
}

void ReadRHC()   /* read rotational hand controller */
{
  register  i;
  short     tmp;

  for (i = 0; i < 3; i++) {
        outp(ADPORTC, i+3);
        outp(ADCNTRL, 0x09);
        outp(ADCNTRL, 0x08);
        outp(ADCNTRL, 0xb);
        tmp = inp(ADPORTA) + AdjRHC[i];
        outp(ADCNTRL, 0x0A);
        if (tmp > 0xFF) tmp = 0xFF;
          else if (tmp < 0x00) tmp = 0x00;
        if ((rhc[i] != tmp) && Display)
                onscrnWH(7, (17+6*i), 0, "%4d ", (Byte) tmp-0x80);
        rhc[i] = tmp;
  }
}

void Leak()        /* look for leaks and locations, trigger alarm if detected */
{
        /* Detect and display any water leaks in control or camera boxes */
        if (StatChg & LEAKS) {
                if (Stats & LK_CNTRL) {
                        onscrnLR(7,50,0,"Water in Control Box!");
                        sndflg[0] = TRUE;
                        }
                else {
                        onscreen(7,50,0,"                    ");
                        sndflg[0] = FALSE;
                        }
                if (Stats & LK_CAMERA) {
                        onscrnLR(9,50,0,"Water in Camera Box!");
                        sndflg[1] = TRUE;
                        }
                else {
                        onscreen(9,50,0,"                    ");
                        sndflg[1] = FALSE;
                        }
                if (Stats & LK_BATBOX) {
                        onscrnLR(11,50,0,"Water in Battery Box!");
                        sndflg[2] = TRUE;
                        }
                else {
                        onscreen(11,50,0,"                     ");
                        sndflg[2] = FALSE;
                        }
                }
}

void InRateGain() {            /* read in new derivative (rate) gains */
        /* Scanf rategains from keyboard (secs) when 'r' is typed */
        register        i;

        onscrnY(18,5,0,"Rate gain input:  roll = ");
        scanf("%d",&rategain[0]);
        onscrnLB(18,22,0,"pitch = ");
        scanf("%d",&rategain[1]);
        onscrnMG(18,22,0,"  yaw = ");
        scanf("%d",&rategain[2]);
        onscreen(18,5,0,"                          ");
        for (i=0; i<3; i++) onscreen(23,30+i*8,0,"%4d",rategain[i]);
        TX_RGain(NULL);
}

void InPropGain() {            /* read in new proportional gains */
        /* Scanf propgains from keyboard (secs) when 'p' is typed */
        register        i;

        onscrnY(18,5,0,"Prop gain input:  roll = ");
        scanf("%d",&propgain[0]);
        onscrnLB(18,22,0,"pitch = ");
        scanf("%d",&propgain[1]);
        onscrnMG(18,22,0,"  yaw = ");
        scanf("%d",&propgain[2]);
        onscreen(18,5,0,"                          ");
        for (i=0; i<3; i++) onscreen(22,30+i*8,0,"%4d",propgain[i]);
        TX_PGain(NULL);
}

void InEulerDes() {            /* read in new desired orientation angles */
        /* Scanf eulerdes from keyboard (secs) when 'e' is typed */
        register        i;

        onscrnY(18,5,0,"Desired euler angle input:  roll = ");
        scanf("%d",&eulerdes[0]);
        onscrnLB(18,32,0,"pitch = ");
        scanf("%d",&eulerdes[1]);
        onscreen(18,5,0,"                              ");
        for (i=0; i<2; i++) {
                onscreen(20,30+i*8,0,"%4d",eulerdes[i]);
                if (eulerdes[i] < 0) {
                        sign[i] = 1;
                        eulerdes[i] = 256 - eulerdes[i];
                        }
                else sign[i] = 0;
                eulerdes[i] &= 0x00FF;
                }
}

void InIntGain() {              /* read in new integral gain */
        /* Scanf integral gains from keyboard (secs) when 'j' is typed */
        register        i;

        onscrnY(18,5,0,"Integral gain input:  roll = ");
        scanf("%d",&intgain[0]);
        onscrnLB(18,26,0,"pitch = ");
        scanf("%d",&intgain[1]);
        onscreen(18,5,0,"                         ");
        for (i=0; i<2; i++) onscreen(24,30+i*8,0,"%4d",intgain[i]);
        TX_IGain(NULL);
}

void InDepGain() {              /* read in new depth gains */
        /* Scanf depth gains from keyboard (secs) when 'l' is typed */
        register        i;

        onscrnY(18,5,0,"Depth gain input: x = ");
        scanf("%d",&depthgain[0]);
        onscrnLB(18,23,0,"y = ");
        scanf("%d",&depthgain[1]);
        onscrnMG(18,23,0,"z = ");
        scanf("%d",&depthgain[2]);
        onscreen(18,5,0,"                    ");
        for (i=0; i<3; i++) onscreen(21,30+i*8,0,"%4d",depthgain[i]);
        TX_DpGain(NULL);
}
```

```
/* ONSCOLOR.C */
/* This program provides different color screen displays */

#include "..\pivecs.h"
#include "pvsecs.h"
#include <gf.h>
#include <asiports.h>
#include <color.h>

void onscrnLR(y,x,page,fmt,a0,a1,a2,a3,a4,a5,a6,a7,a8,a9)
int y,x,page;
char *fmt;
unsigned a0,a1,a2,a3,a4,a5,a6,a7,a8,a9;
{
        curset(y,x,page);
        rprintf(LTRED,page,fmt,a0,a1,a2,a3,a4,a5,a6,a7,a8,a9);
        return;
}

void onscrnY(y,x,page,fmt,a0,a1,a2,a3,a4,a5,a6,a7,a8,a9)
int y,x,page;
char *fmt;
unsigned a0,a1,a2,a3,a4,a5,a6,a7,a8,a9;
{
        curset(y,x,page);
        rprintf(YELLOW,page,fmt,a0,a1,a2,a3,a4,a5,a6,a7,a8,a9);
        return;
}

void onscrnLB(y,x,page,fmt,a0,a1,a2,a3,a4,a5,a6,a7,a8,a9)
int y,x,page;
char *fmt;
unsigned a0,a1,a2,a3,a4,a5,a6,a7,a8,a9;
{
        curset(y,x,page);
        rprintf(LTBLUE,page,fmt,a0,a1,a2,a3,a4,a5,a6,a7,a8,a9);
        return;
}

void onscrnMG(y,x,page,fmt,a0,a1,a2,a3,a4,a5,a6,a7,a8,a9)
int y,x,page;
char *fmt;
unsigned a0,a1,a2,a3,a4,a5,a6,a7,a8,a9;
{
        curset(y,x,page);
        rprintf(LTMAGENTA,page,fmt,a0,a1,a2,a3,a4,a5,a6,a7,a8,a9);
        return;
}

void onscrnWH(y,x,page,fmt,a0,a1,a2,a3,a4,a5,a6,a7,a8,a9)
int y,x,page;
char *fmt;
unsigned a0,a1,a2,a3,a4,a5,a6,a7,a8,a9;
{
        curset(y,x,page);
        rprintf(WHITE,page,fmt,a0,a1,a2,a3,a4,a5,a6,a7,a8,a9);
        return;
}
```

```
/* SECSMSGS.C */
/* Last Modified: 3/6/90 by Wendy Power */
/* Contains all message structures. Some calculations
          and message distributions also. */

#include "..\pivecs.h"
#include "..\pvdata.h"
#include "pvsecs.h"
#include "..\pvastro.msg"

MsgHandler TX_Camera(msg)        /* transmits camera aperture commands */
   MsgPtr msg;
{
  Byte   array[2];

  array[0] = RXCAMERA;
  array[1] = Iris;
  pvSend(array, 2);
  return( OK );
}

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

MsgHandler TX_THC(msg) /* transmits translational hand controller commands */
   MsgPtr msg;
{
  register           i;
  Byte   array[4];

  array[0] = RXTHC;
  for (i = 0; i < 3; i++) array[i+1] = thc[i];
  pvSend(array, 4);
  return( OK );
}

MsgHandler TX_RHC(msg)           /* transmits rotational hand controller commands */
   MsgPtr msg;
{
  register           i;
  Byte   array[4];

  array[0] = RXRHC;
  for (i = 0; i < 3; i++) array[i+1] = rhc[i];
  pvSend(array, 4);
  return( OK );

}

MsgHandler RX_Stats(msg)         /* receives ASTRO status */
   MsgPtr msg;
{

          StatChg = Stats ^ (*msg->data);
          Stats = *msg->data;
  return( OK );
}

MsgHandler TX_SPanel(msg)        /* transmits switch panel commands */
   MsgPtr msg;
{
   Byte array[2];

   array[0] = RXSPANEL;
   array[1] = SPanel;
   pvSend(array, 2);
   return( OK );
}

MsgHandler TX_Euler(msg)           /* transmits new desired orientation angles */
   MsgPtr msg;
{
        register  i;
        Byte              array[5];

        array[0] = RXEULER;
        for(i=0; i<2; i++) {
                array[i+1] = eulerdes[i];
                array[i+3] = sign[i];
                }
        pvSend(array, 5);
        return( OK );
}

MsgHandler TX_PGain(msg)           /* transmits new proportional gains */
   MsgPtr msg;
{
        register  i;
        Byte              array[4];

        array[0] = RXPGAIN;
        for(i=0; i<3; i++) array[i+1] = propgain[i];
        pvSend(array, 4);
        return( OK );
}

MsgHandler TX_RGain(msg)           /* transmits new derivative (rate) gains */
   MsgPtr msg;
{
        register  i;
        Byte              array[4];

        array[0] = RXRGAIN;
        for(i=0; i<3; i++) array[i+1] = rategain[i];
        pvSend(array, 4);
        return( OK );
}

MsgHandler RX_PGauge(msg)          /* receives pressure gauge data */
   MsgPtr msg;
{
        register          i;
        Byte                      tmp[7];
        Byte                      temp;
        long                      ltmp;
        BytePtr datptr = msg->data;

        for (i=0; i<7; i++) tmp[i] = *datptr++;
        for (i=0; i<2; i++) pgauge[i] = ((temp=tmp[2*i])<<4)+tmp[2*i+1];
        pgauge[2] = ((ltmp = tmp[5] << 4) & 0x0F00) + tmp[4];
        pgauge[3] = ((ltmp = tmp[5] << 8) & 0x0F00) + tmp[6];
        if (Display)
                for (i=0; i<4; i++) onscreen(13,20+i*8,0,"%4ld",pgauge[i]);
        if (Stats & DATADUMP) {
                for (i=0; i<4; i++) data[p][i] = pgauge[i];
                onscreen(15,60,0,"%4ld",p);
                p++;
                pvRequest( TXPGAUGE );
                }
        return( OK );
}

MsgHandler RX_Gyros(msg)            /* receives gyro data */
   MsgPtr msg;
{
        register            i;
        Byte                        tmp[6];
        Byte                        temp;
        float                       tmprpy[3];
        BytePtr datptr = msg->data;

        for (i=0; i<6; i++) tmp[i] = *datptr++;
        for (i=0; i<3; i++) gyros[i] = ((temp=tmp[2*i])<<4)+tmp[2*i+1];
        if (Display) {
                tmprpy[0] = -89.668 + (0.043714 * (double) gyros[0]);
                tmprpy[1] = -89.737 + (0.043595 * (double) gyros[1]);
                tmprpy[2] = -90.218 + (0.043846 * (double) gyros[2]);
                for (i=0; i<3; i++) onscreen(11,20+i*9,0,"%6.2f",tmprpy[i]);
                }
        if (Stats & DATADUMP) {
                for (i=0; i<3; i++) data[g][i+4] = gyros[i];
                onscreen(15,65,0,"%4ld",g);
                g++;
                pvRequest( TXGYROS );
                }
        return( OK );
}

MsgHandler RX_Count(msg)            /* receives number of data samples */
   MsgPtr msg;
{
        register            i;
        Byte                        tmp[2];
        BytePtr datptr = msg->data;

        for (i=0; i<2; i++) tmp[i] = *datptr++;
        count = (tmp[0] << 8) + tmp[1];
        onscreen(13,60,0,"count = %4ld",count);
        pvRequest( TXPGAUGE );
        pvRequest( TXGYROS );
  return( OK );
}

MsgHandler TX_IGain(msg)            /* transmits new integral gains */
   MsgPtr msg;
{
        register  i;
        Byte                array[3];

        array[0] = RXIGAIN;
        for(i=0; i<2; i++) array[i+1] = intgain[i];
        pvSend(array, 3);
        return( OK );
}

MsgHandler TX_DpGain(msg)           /* transmits new depth gains */
   MsgPtr msg;
{
        register  i;
        Byte            array[4];

        array[0] = RXDPGAIN;
        for (i=0; i<3; i++)         array[i+1] = depthgain[i];
        pvSend(array, 4);
        return( OK );
}
```

Appendix C: Matrix Calculations

This Appendix covers the matrix calculations that result in the correction matrix, $R_{correction}$, as a function of the desired orientation angles $\gamma_{desired}$ and ßdesired and the current angles $\gamma_{current}$ and $\beta_{current}$.

The matrix equations

$$R_{desired} * R_{correction} = R_{current}$$

$$R_{correction} = R_{desired}^{-1} * R_{current}$$

$$R_{correction} = R_{desired}^{T} * R_{current}$$

determine the form of the correction matrix, on which the form of the controller will be based. The desired and current matrices are simply the rotation matrix found in Appendix A, simplified to not include the yaw, or α, rotation. These matrices, represented in terms of the desired and current angles, are:

$$R_{desired} = \begin{bmatrix} \cos\beta_d & \sin\beta_d*\sin\gamma_d & \sin\beta_d*\cos\gamma_d \\ 0 & \cos\gamma_d & -\sin\gamma_d \\ -\sin\beta_d & \cos\beta_d*\sin\gamma_d & \cos\beta_d*\cos\gamma_d \end{bmatrix}$$

$$R_{desired}^{T} = \begin{bmatrix} \cos\beta_d & 0 & -\sin\beta_d \\ \sin\beta_d*\sin\gamma_d & \cos\gamma_d & \cos\beta_d*\sin\gamma_d \\ \sin\beta_d*\cos\gamma_d & -\sin\gamma_d & \cos\beta_d*\cos\gamma_d \end{bmatrix}$$

$$R_{current} = \begin{bmatrix} \cos\beta_c & \sin\beta_c*\sin\gamma_c & \sin\beta_c*\cos\gamma_c \\ 0 & \cos\gamma_c & -\sin\gamma_c \\ -\sin\beta_c & \cos\beta_c*\sin\gamma_c & \cos\beta_c*\cos\gamma_c \end{bmatrix}$$

Solving the equation for $R_{correction}$ yields:

$$\begin{bmatrix} c(\beta_d-\beta_c) & s\gamma_c*s(\beta_c-\beta_d) & c\gamma_c*s(\beta_c-\beta_d) \\ s\gamma_d*s(\beta_d-\beta_c) & s\gamma_c*s\gamma_d*c(\beta_d-\beta_c) + c\gamma_d*c\gamma_c & s\gamma_d*c\gamma_c*c(\beta_d-\beta_c) - c\gamma_d*s\gamma_c \\ c\gamma_d*s(\beta_d-\beta_c) & c\gamma_d*s\gamma_c*c(\beta_d-\beta_c) - s\gamma_d*c\gamma_c & c\gamma_d*c\gamma_c*c(\beta_d-\beta_c) + s\gamma_d*s\gamma_c \end{bmatrix}$$

This matrix is the basis for the equations implemented in the flight software.