NAVAL POSTGRADUATE SCHOOL
MONTEREY, CALIFORNIA

THESIS

MERGING VIRTUAL AND REAL EXECUTION LEVEL CONTROL SOFTWARE FOR THE PHOENIX AUTONOMOUS UNDERWATER VEHICLE

by

Michael L. Burns

September 1996

Thesis Advisors: Robert B. McGhee
Don Brutzman

Approved for public release; distribution is unlimited.

REPORT DOCUMENTATION PAGE

Form Approved OMB No. 0704-0188

1. AGENCY USE ONLY *(Leave blank)*	2. REPORT DATE September 1996	3. REPORT TYPE AND DATES COVERED Master's Thesis

4. TITLE AND SUBTITLE MERGING VIRTUAL AND REAL EXECUTION LEVEL CONTROL SOFTWARE FOR THE PHOENIX AUTONOMOUS UNDERWATER VEHICLE	5. FUNDING NUMBERS
6. AUTHOR(S) Burns, Michael L.	
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Naval Postgraduate School Monterey CA 93943-5000	8. PERFORMING ORGANIZATION REPORT NUMBER
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)	10. SPONSORING/MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government.

12a. DISTRIBUTION/AVAILABILITY STATEMENT Approved for public release; distribution is unlimited.	12b. DISTRIBUTION CODE

13. ABSTRACT

The Naval Postgraduate School (NPS) is developing an AUV, Phoenix. The Phoenix has the capability of precise navigation, however too much time is needed to validate a new section of code. NPS is also developing a virtual AUV, which has the capability of being networked, having flexible missions, and having a quick feedback of results when validating new portions of code. The virtual AUV has a drawback of never being tested for real world precision. This thesis discusses the steps taken to combine these two sets of control code to obtain the maximum functionality that will drive either the virtual or actual AUV and produce a faster feedback response to newly developed code.

As a result of this effort, the newly developed control code has successfully driven both the actual and virtual AUVs and provides a means for readily validating new code. Also this new control code has given the AUV research group the ability to perform distributed software development, test all AUV hardware from either the onboard or offboard computers, conduct flexible missions, and test missions in the virtual world prior to conducting them with the AUV.

14. SUBJECT TERMS Control Software, Under Robotics, AUV, Combining Control Code, Execution Level	15. NUMBER OF PAGES 243
	16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT Unclassified	18. SECURITY CLASSIFICATION OF THIS PAGE Unclassified	19. SECURITY CLASSIFICATION OF ABSTRACT Unclassified	20. LIMITATION OF ABSTRACT UL

NSN 7540-01-280-5500

Standard Form 298 (Rev. 2-89)
Prescribed by ANSI Std. 239-18 298-102

Approved for public release; distribution is unlimited.

MERGING VIRTUAL AND REAL EXECUTION LEVEL CONTROL SOFTWARE FOR THE PHOENIX AUTONOMOUS UNDERWATER VEHICLE

Michael L. Burns
Lieutenant, United States Navy
B.S., United States Naval Academy, 1990

Submitted in partial fulfillment
of the requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

from the

NAVAL POSTGRADUATE SCHOOL
SEPTEMBER 1996

Author: _____
Michael L. Burns

Approved by: _____
Robert B. McGhee, Thesis Advisor

_____
Don Brutzman, Thesis Advisor

_____
Ted Lewis, Chair
Department of Computer Science

ABSTRACT

The Naval Postgraduate School (NPS) is developing an AUV, Phoenix. The Phoenix has the capability of precise navigation, however too much time is needed to validate a new section of code. NPS is also developing a virtual AUV, which has the capability of being networked, having flexible missions, and having a quick feedback of results when validating new portions of code. The virtual AUV has a drawback of never being tested for real world precision. This thesis discusses the steps taken to combine these two sets of control code to obtain the maximum functionality that will drive either the virtual or actual AUV and produce a faster feedback response to newly developed code.

As a result of this effort, the newly developed control code has successfully driven both the actual and virtual AUVs and provides a means for readily validating new code. Also this new control code has given the AUV research group the ability to perform distributed software development, test all AUV hardware from either the onboard or offboard computers, conduct flexible missions, and test missions in the virtual world prior to conducting them with the AUV.

TABLE OF CONTENTS

ACKNOWLEDGMENTS

I would like to take this time to recognize my classmates LT Bradley Leonhardt, LT Michael Campbell, LT David McClarin, and LT Duane Davis. If not for the perseverance and the support of the group as a whole none of our individual projects would have been as successful. I would like to thank Russ Whalen for the support and the time spent teaching me an assortment of "hands on" hardware issues. Next, I would like to thank my thesis advisor, Professor Robert B. McGhee who initiated the idea of joining the AUV research group and has also given me incredible support during the research and writing of my thesis. Also, I would like to thank my other thesis advisor, Professor Donald P. Brutzman who has taken me under his wing and answered every possible question imaginable. He was always available to me and instead of just discussing a concept, often we would both sit at a terminal for hours finding the best way to employ ideas.

Lastly, but mostly, I would like to thank my wife Susan who has given me complete support during this entire project. She has had to single handedly raise our first son Tyler, who was born during the beginning of this project. I know she will be happy to have me back full time so that we can raise our second child, that is due in the near future, together.

This thesis was funded in part by the National Science Foundation under Grant BCS - 9306252.

I. INTRODUCTION

A. GOALS

An Autonomous Underwater Vehicle (AUV) is a self-contained unmanned vehicle used for underwater missions such as surveying, pollution detection, or mine detection and neutralization. In order to be deemed truly autonomous the vehicle must be able to independently follow a mission plan, interact with its environment, accomplish a goal and return to a predesignated destination.

The goal of this thesis is to merge two independent sets of control software for the NPS autonomous underwater vehicle, Phoenix. One of the software control codes used in this project is taken from an actual AUV that has been the work of Ph.D. candidate David Marco (Marco 96). The AUV is computer driven and has run successful missions in a test tank using this control code (Marco 96)(Marco, Healey, McGhee 96). The operating system for this code is OS-9, running on a GESPAC 68030 microprocessor. The other software control code is taken from a virtual AUV that has been the work of Professor Donald Brutzman at the Naval Postgraduate School (NPS). This is a computer-generated robust simulation of an AUV in a virtual environment. The robot control code runs on an SGI computer system (Brutzman 94). Although each set of control code is run on different operating systems, they are both written in C.

The intended result of this project is that the combined software for robot control can run identically on all pertinent systems. In other words, the exact same source code can be run in either the virtual or actual world and produce the same results. This is intended to enable robot research that can be tested and refined in the virtual AUV in a fraction of the time required to test the same ideas in the actual AUV. Once these ideas have been refined in the virtual world, the same code can be put directly into the actual AUV for real-world in-water verification and validation.

Different operating systems and issues of real-time performance present serious obstacles against effectively merging the two systems. The final test for validation of this project is an untethered "Moss Landing" mission. The Moss Landing mission is

explained in greater detail in Chapter VI. In short, the Moss Landing mission is a sea trial designed to test navigation, sonar search, sonar classification and path replanning, all performed by the AUV while it is free of an Ethernet connection. This thesis discusses and provides solutions to the problems encountered while combining the two software control systems to run in either the virtual or actual world. It also gives an assessment of experimental results obtained during the Moss Landing mission testing. A copy of the source code used for the Moss Landing mission is found in Appendix A-E and is explained in Chapter V.

B. MOTIVATION

The motivation for this project stems from seeing a slow but steady progression in the development of the actual NPS AUV *Phoenix*. Years have been spent refining small portions of code in the *Phoenix*. If there was a way to visualize and correct the refinements prior to taking the time required for testing the changes in the actual AUV, development of the AUV should progress substantially faster. In a sense, many hours are spent on small details with known or predictable solutions that are bottle-necked due to the limited time that can be afforded to validate these solutions in-water with the Phoenix. With the support of a virtual environment, most bugs can be worked out in a fraction of the time that might normally be required if testing was done in the actual AUV. If this project is successful the development of the Naval Postgraduate School AUV will progress at a speed only imagined by those that have been developing it thus far. Successful completion of this project can also ensure that the NPS research team will remain on the leading edge of technology. In fact the work undertaken in this thesis has been successful, and the hoped-for improvements in vehicle design methodology have also occurred.

C. ORGANIZATION

This thesis is presented in much the same manner as it was performed. Chapter I presents the motivation for taking on this project. Chapter II discusses previous work

done in three areas that are significant to this project: the robot software architecture used for this project, the actual AUV, and the virtual AUV. Chapter III states the research problem, identifying a series of questions and concerns that need to be addressed prior to accepting the project and that can ultimately be used to distinguish the validity of the project. Chapter IV discusses the specific methodology of steps that were taken while merging the control software. Chapter V gives an in-depth discussion on how the combined software control code actually works. Chapter VI presents and evaluates the logical progression of tests that were performed, as well as a basic analysis of test results for the Moss Landing mission. Chapter VII provides conclusions regarding this project and presents recommendations for future work.

II. PREVIOUS WORK

A. INTRODUCTION

Research on Autonomous Underwater Vehicle's (AUV) has been an ongoing project at the Naval Postgraduate School (NPS) since 1987 (Healey 90,92) (Brutzman 96). The need to further technology in the underwater arena is realized by most Naval officers serving at sea. A large contribution to the NPS AUV project development has been by graduate student officers that have returned from sea to study. This level of Naval experience applied to the design of underwater robots is unparalleled at any other educational institution.

This chapter discusses how earlier NPS AUV research was split into two categories, actual and virtual. It will also discuss the progression of each and the common link between the two. After understanding this background it is clear why the two entities need to merge.

B. ACTUAL AUV

1. Physical Description

The Naval Postgraduate School AUV *Phoenix* hull is approximately 2.4 meters long, 0.46 meters wide and 0.31 meters deep. It has the general shape of a miniature submarine with two aft propellers, two vertical thrusters, two horizontal thrusters, two forward rudders, two aft rudders, two forward plane surfaces, and two aft plane surfaces to control its movement through the water. It has a 2 psig pressurized hull with a free-flooding nose cone that houses the AUV's sonars and depth cell. While submerged it displaces approximately 197.3 Kilograms and is ballasted to be neutrally buoyant. Figure 1 and Figure 2 show the external components and a snapshot of the AUV in the test tank respectively.

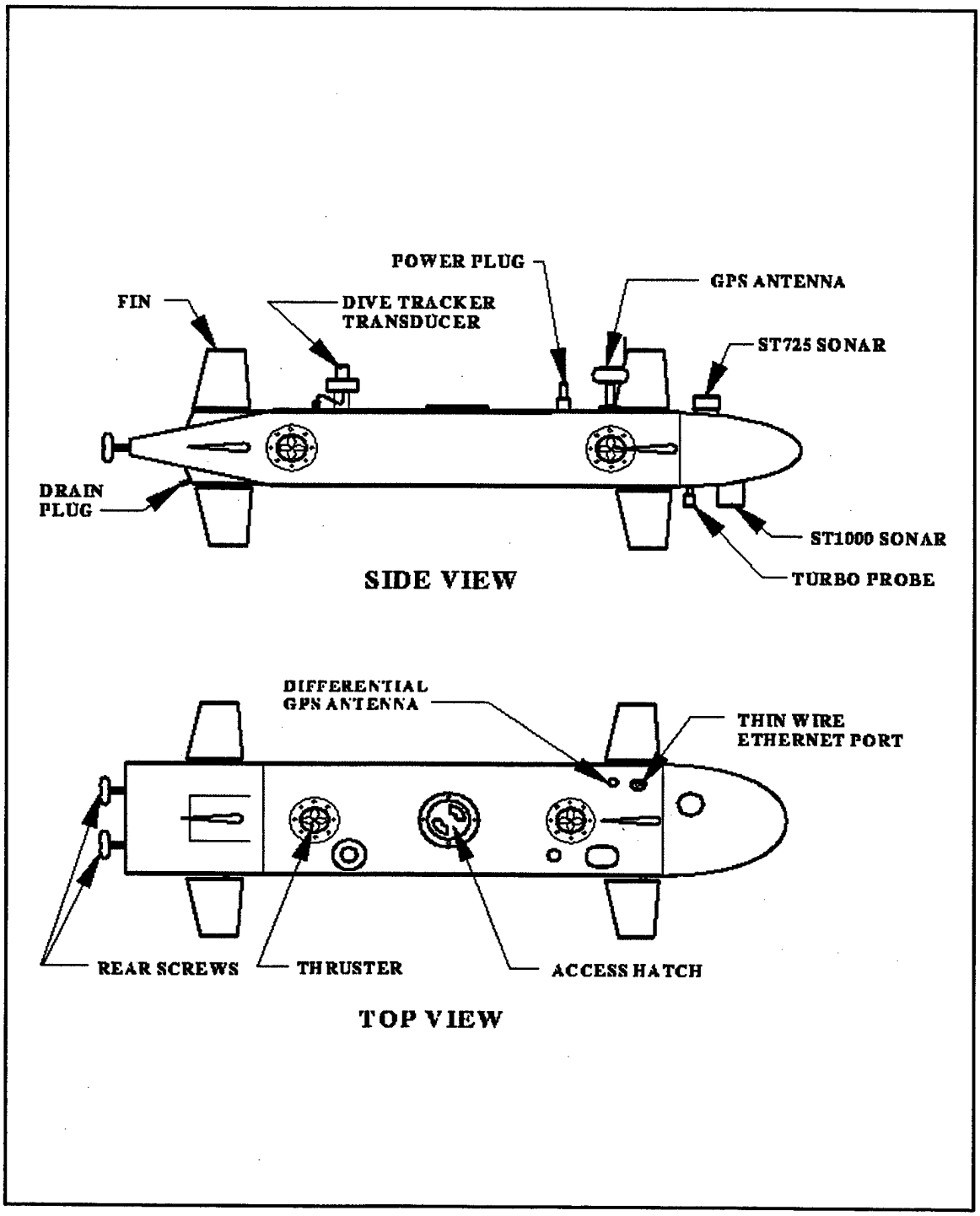

Figure 1 External components of the AUV (Marco 96)

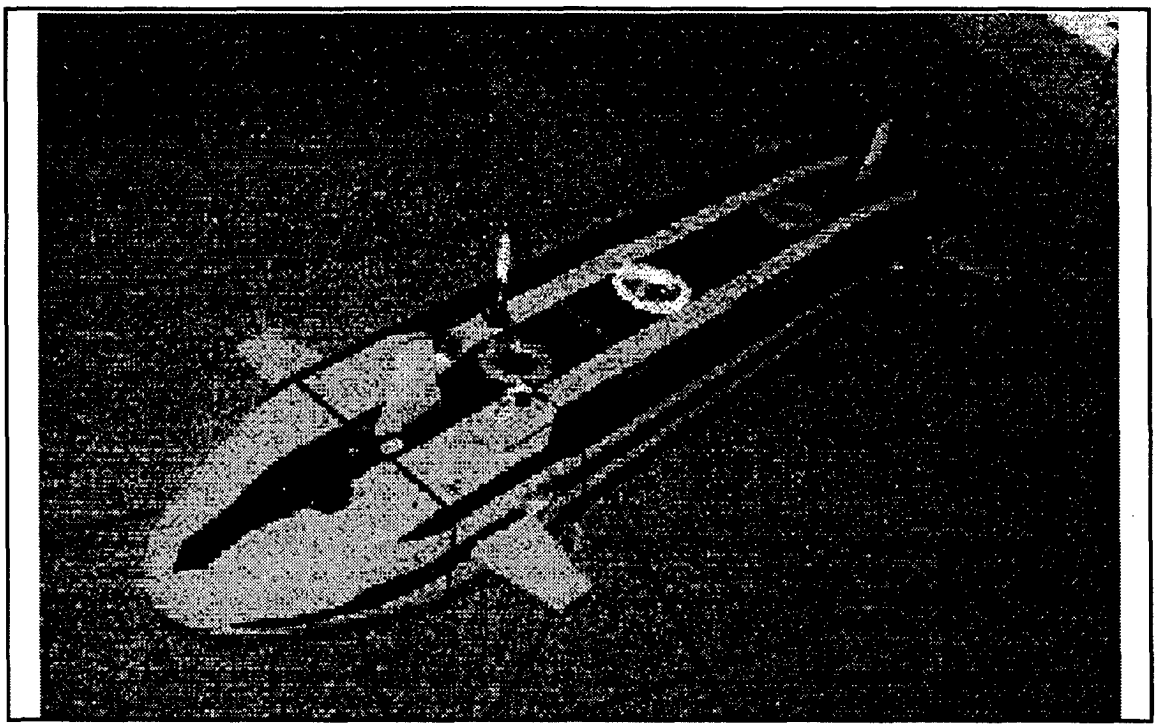

Figure 2 AUV in the test tank (Leonhardt 96)

Some of the hardware and their general purposes are listed in the following Figure 3:

1). A computer for controlling the AUV's stability, *execution* level
2). A computer for data storage and running *strategic* and *tactical* levels
3). Turbo Probe for sensing water speed
4). Three sonars
    a. downward looking (altimeter)
    b. overall environment sensing (ST725)
    c. obstacle classification (ST1000)
5). GPS for tracking the vehicle's latitude and longitude in open water
6). DiveTracker for tracking the vehicle precisely in small areas.
7). Gyro's for sensing the vehicle's orientation about three degrees of rotational freedom as well as three angular rates
8). Batteries to run the computer and motors
9). Depth cell to measure depth
10). A-to-D and D-to-A converters for computer-hardware interfaces

Figure 3 AUV internal hardware and brief description

Figure 4 depicts the internals of the AUV.

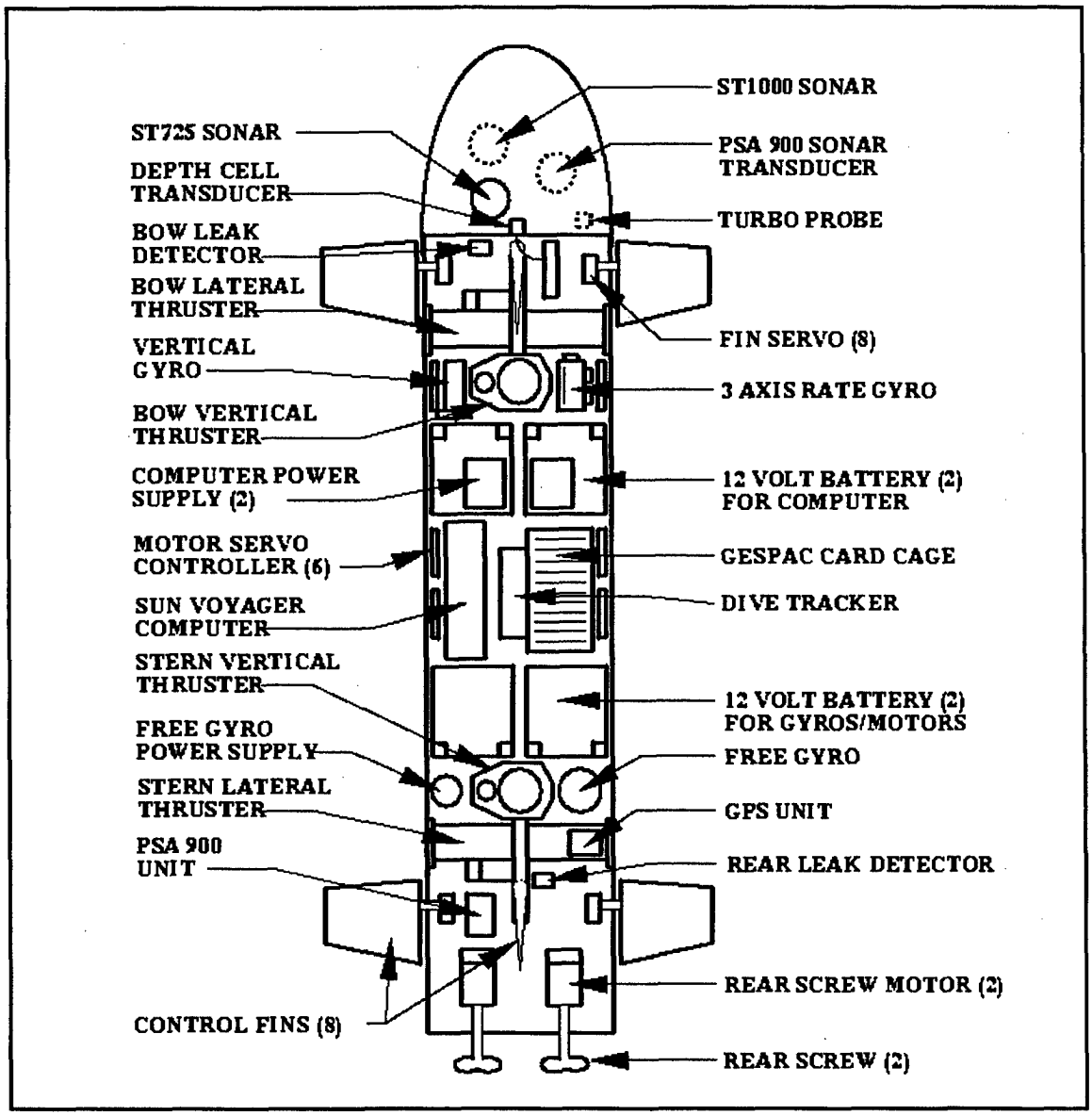

Figure 4 AUV internals (Marco 96)

After many work hours, the actual AUV became capable of station keeping (Healey 95). Station keeping is the ability for the actual AUV to be smoothly directed to

a point in a 3-dimensional environment and then maintain that position. This is an essential milestone in the development of the AUV control. Because station keeping had been achieved by the AUV, precise control in water was possible. This was a major accomplishment for the research team. Expansion to make the actual AUV more robust and flexible to changing scenarios was the next step in this progression. An issue that remains a work in progress is accurate navigation when surfaced and submerged, using dead reckoning, short-baseline acoustics and GPS (McClarin 96) (Bachmann 96).

2. Further Software Development

At the beginning of this thesis research, software had been written to support all three levels of control, as explained in the RBM architecture section. However, the need to have a stable foundation from which to work necessitated that the majority of time of this thesis research be spent on the *execution* level. At the time this work began the *strategic* and the *tactical* levels, although in place and functioning, were more or less only in place to support the *execution* level. If an AUV can not maintain basic navigational control in its environment, it is useless to attempt to make it perform other tasks. Thus the *execution* level is indispensable.

The most advanced mission conducted in the test tank by the AUV was a test on its ability to take station. This mission involved the *execution* level using its sonar system to locate a target then drive to a point (i.e. station) close to that target. The following is a description of how the test was conducted. For simplicity of explanation, because the AUV operated at a constant depth of two feet and the target was at a depth of two feet, the example will be explained using a 2-dimensional coordinate system with units being measured in feet. Figure 5 gives a graphical display of the example mission. Again for simplicity, Figure 5 is not to scale and merely illustrates the idea expressed in the following paragraph.

The AUV's starting position is at a point [-5, -5] relative to the target that was defined as the point [0, 0] and the AUV is told to face North. Next, the AUV submerged two feet and remained stationary while the active sonar mapped out the test tank walls and located a target. The target was a metal cylinder suspended by a rope and placed in

the test tank approximately two feet under the surface. It was required of the AUV to distinguish the target from the walls of the test tank. Once the target had been acquired the AUV was directed to a point close to the target and told to hover at that point for a period of time. The point that is used in this example is the point [1, -1], relative to the target. During the transition to the point, the AUV is restricted to maintaining its Northerly heading. This was done to test if the thrusters and the propellers could function together in transiting to a point. Once at the point [1, -1] the AUV was to remain there for a period of 30 seconds facing North. At the end of this period the AUV would be told to turn to its left 45 degrees but still maintain the point [1, -1]. From a birds-eye-view this would now appear that the AUV is facing the target and hovering at a point 1.41 feet from the target. Next the AUV would be told to transit to a point [-1, -1] while maintaining its northwest heading then hold that position for 30 seconds. This test was done to simulate the AUV's ability to navigate around a target. It was told to take the first station at a particular heading then take another position close to the target at another heading to simulate, if there was a fixed camera on the AUV, taking different angled snapshots of the target. After the second position was reached and held the AUV was programmed to return to the starting position, regain the Northerly heading and terminate.

This test validated both the AUV's ability use both the thrusters and propellers in an stable effort of transit through the water, and the sonars' ability to interact with the AUV's navigational software in finding contacts and holding positions relative to those contacts. What was also shown by this exercise was that this entire missions was hand-coded step-by-step and it is not easily adapted to other missions. (Marco 96)

C. RATIONAL BEHAVIOR MODEL (RBM) ARCHITECTURE

The notion of a tri-level architecture for the NPS AUV was first introduced in 1992 (Kwak 92). This idea was then further refined and expanded in later work (Byrnes 93) (Byrnes 96). The RBM at its highest level of abstraction, the *strategic* level, is concerned with goals to be accomplished without regard to precisely how they are

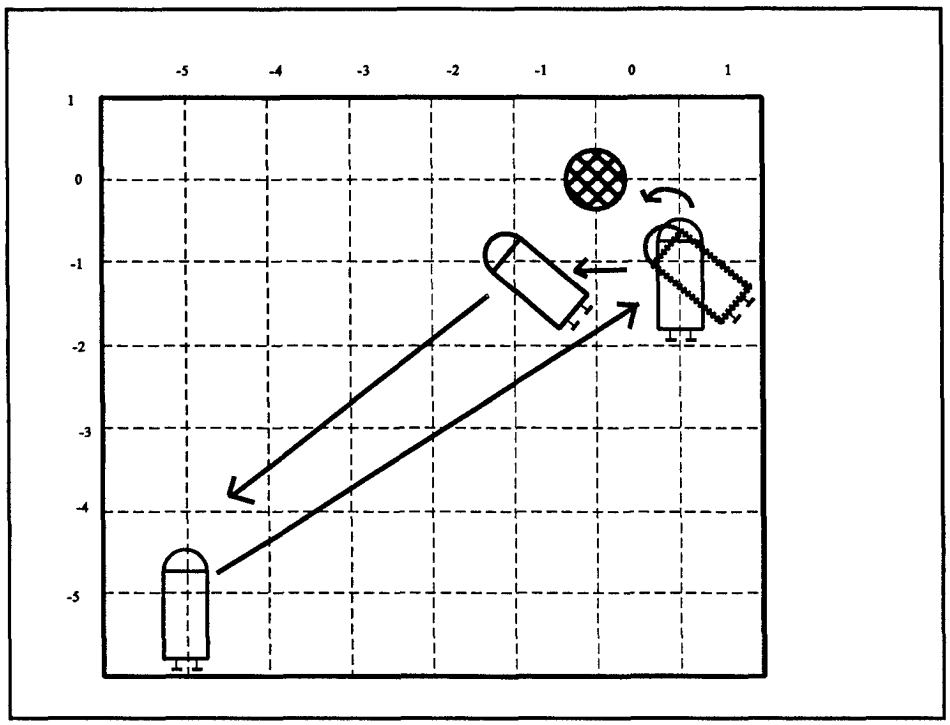

Figure 5 Non-scaled display of locating and operating around a target

achieved. Goals are then funneled down to an intermediate *tactical* level that can direct (via the *execution* level) the necessary pieces of robot software and hardware in the proper sequence to accomplish the directed goals.

The lowest (*execution*) level of the RBM software architecture is a real-time closed loop that maintains stability through direct software-to-hardware interfaces. This software level is able to accept commands from the intermediate level and have the hardware respond to accommodate intermediate-level demands. This lowest level has no idea of where it is going or why. It merely responds to the intermediate levels demands while keeping the AUV stable in the water.

The *strategic* software level interfaces with the *tactical* level in strictly a boolean fashion. A command is sent, and a "yes" or "no" answer is received. There are no calculations performed at the *strategic* level and no timing restrictions; therefore the *strategic* level operates in an asynchronous manner. The interface between the *tactical* and *execution* levels is confined to orders being sent down and telemetry of the vehicle

being sent up. The *tactical* level has the ability to manipulate numbers but, like the *strategic* level, is not required to meet hard time deadlines with its results. Therefore it too operates in an asynchronous manner. The *execution* level is responsible for the stability of the AUV and therefore functions synchronously through use of timed interrupts. That is, in order to keep a stable platform, the *execution* level must conform to hard real-time requirements. Also the *execution* level is given the ability, if needed, to have absolute control. If a critical dilemma is sensed such as flooding, loss of depth control, or loss of communications with the *tactical* level, the *execution* level has the ability to override the *tactical* level and perform self-preservation tasks. A detailed description of the RBM structure is provided in (Byrnes 96).

D. WHY DID ACTUAL AND VIRTUAL ORIGINALLY DIVERGE

Building an AUV to interact with its environment autonomously in six degrees of freedom is a difficult and time-consuming task. Prior to the virtual world work reported in (Brutzman 94), each time there was an advancement in the software, the AUV had to be placed into a test tank to verify that portion of code. It quickly became apparent that if there was a way to simulate AUV actions and reactions, the project would advance at a faster pace. Simulation reduces the time and effort required to put the AUV in the water for simple tests in control and logic. This simulation effort spawned a new research area that was developed in parallel with the development of the AUV.

The simulated AUV development first lagged the actual AUV development. This was due to the simulated AUV being required to reproduce all of the actual AUV's movements. During that original period, high-speed graphics programs and other technology tools were not available. However, capabilities of the simulated AUV in time surpassed the actual AUV. Once the simple tasks of the actual AUV were mimicked, the simulated AUV work went on to other maneuvers that the actual AUV could not yet perform. Along with simulating the movements of the actual AUV, the simulated AUV evolved into a fully operational virtual AUV. This was an early significant proof of value of the simulation-based design (SBD) approach.

Although the pace of the virtual and actual control code development did not progress at the same rate, they both shared a common link. The Rational Behavior Model (RBM) (Byrnes 96) was the software architecture that both versions were designed to accommodate. To facilitate adhering to this structure, the virtual world robot software used the actual robot source code as a basis. This approach was intended to insure that the virtual world AUV might perform the same tasks as the actual AUV. In early development of the virtual world, whenever possible, this corresponding software structure was maintained in order to minimize software divergence. In a relatively short amount of time (about a year) the logic of the virtual AUV grew far more sophisticated than that of the actual AUV. This was because it was far easier to test different ideas in the virtual world. However, not having the opportunity to encounter the real world, the control of the virtual AUV was not perfectly tuned for real world operation. Thus significant capabilities and drawbacks existed side by side in the real and virtual versions of the *execution* software.

E. VIRTUAL AUV

The early years of graphical simulators developed for the NPS AUV concentrated on particular aspects of the AUV. Each of these simulators served as a building block to investigate a different aspect. It was not until 1992 when multiple aspects of AUV simulation were combined to form a NPS AUV Integrated Simulator (Brutzman 92).

The underlying goal in the development of the integrated simulator was to be able to use the simulator as a tool for rapid development of the stand-alone AUV. In the creation of such a tool, three areas were clearly defined. If a simulator could simply and reliably demonstrate pre-mission testing, pseudo-mission testing, and post-mission playback of a recorded mission, the simulator would exercise the essence of the actual AUV (Brutzman 92). These three areas are discussed in subsections that follow along with a general flow chart of the operation of the *execution* level.

The integrated simulator provided a solid foundation for the implementation of the virtual world. However, the need became apparent for a networked 3-dimensional

(3D) virtual world allowing even more software engineers to test specific modules. Accompanying that idea was the concept that, in order to properly test an actual mission and reproduce it in a virtual world, hard-real-time response software must be developed.

The virtual world in Figure 6 includes a 3D graphical display of an AUV to scale, a test tank to scale, and a hydrodynamics model that correctly reproduces the AUV's movement through water.

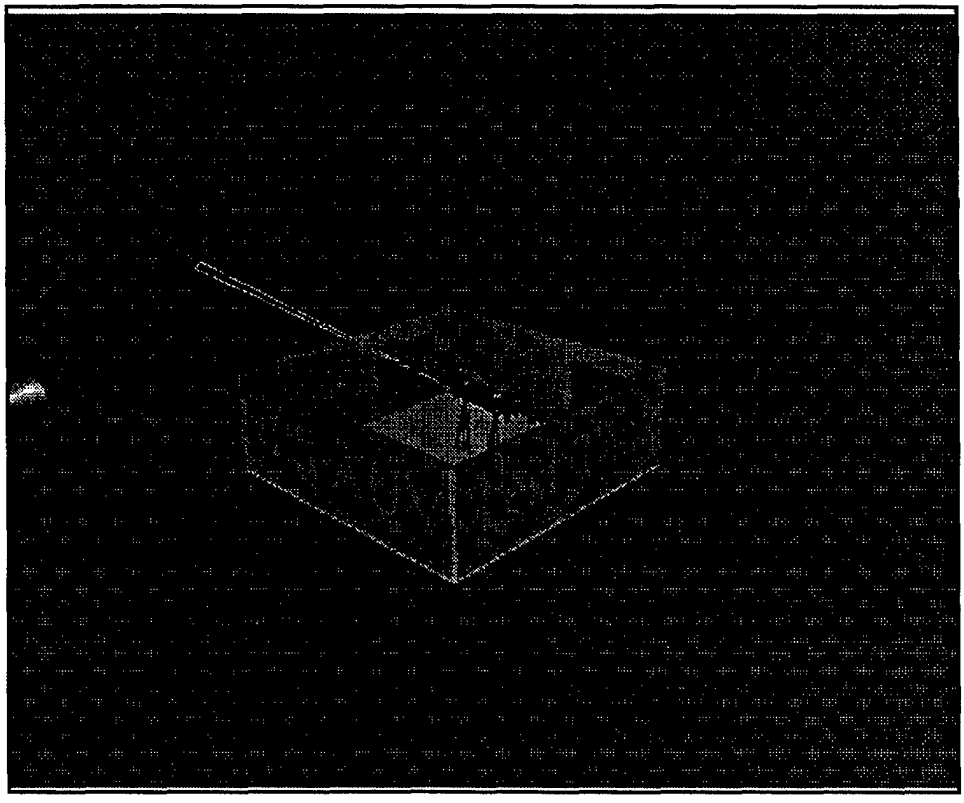

Figure 6 Virtual AUV and test tank

The graphics view into an underwater environment also gives the user the ability to see any aspect of the AUV. Because the virtual world is networked, it simultaneously allows many observers to either watch a mission that is ongoing or run separate missions. If many viewers are watching a single mission, each viewer is able to view a different aspect of the AUV. In other words each viewer has their own camera and can point it

anywhere in the virtual world to see what ever they want to see. Figure 7 shows the AUV diving while driving backwards. This is shown by the cones coming up from the vertical thrusters and the inverted cones behind the aft propellers. Figure 8 shows the AUV climbing and going forward. This is shown by the cones below the vertical thrusters and the cones behind the aft propellers. Also, the plane surfaces in Figure 8 are angled for ascending. The magnitude of the cones displays the amount of force being used.

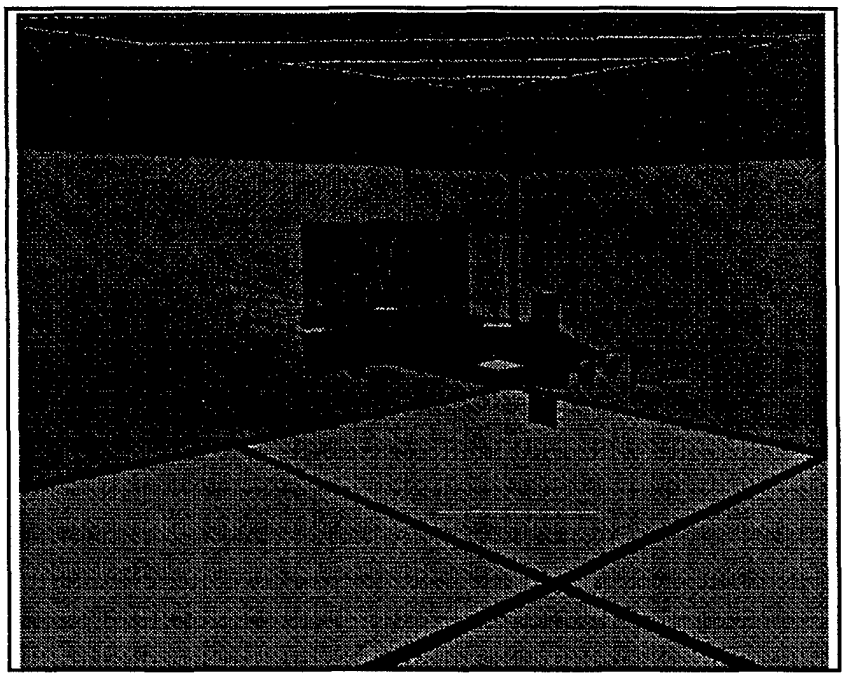

Figure 7 Virtual AUV backing down and descending

The virtual world developed in (Brutzman 94) was the turning point in simulation for the Phoenix project. It was designed to meet all of the concepts depicted by the integrated simulator; i.e. pre-mission testing, pseudo-mission testing, and post-mission playback. Also, hard real-time constraints were added and the virtual world was made available to all users over the NPS network. Nevertheless it is not bound to the NPS campus. As predicted, if general ideas for the robot can first be visualized on a graphics computer screen, implementation details can be worked out quickly and confidently. One particular concept of note that was considered in the early years was the idea of first developing the

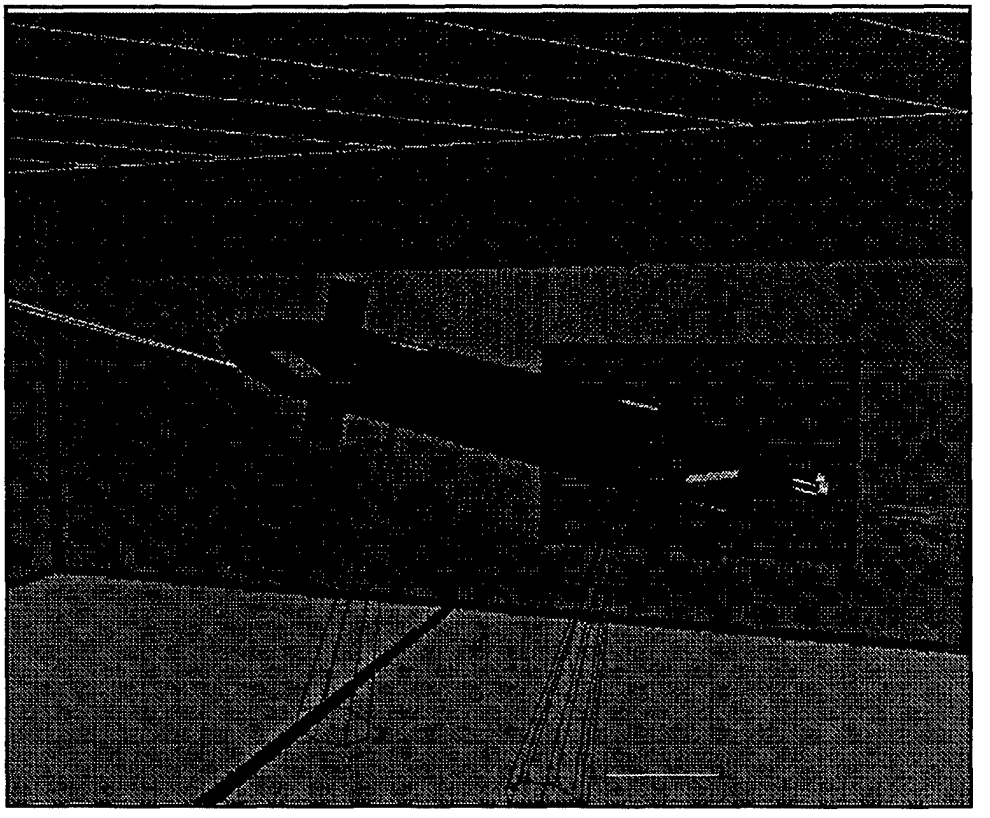

Figure 8 Virtual AUV propelling forward and ascending

physical AUV and then making the virtual AUV follow. "As the characteristics of the NPS Model 2 AUV become more well defined, and the simulation systems are upgraded to reflect these operating characteristics,..." (Rogers 89). It is now possible to incorporate a new piece of hardware into the virtual world inexpensively (without a complete rebuild of the actual AUV) in order to test its validity. Then if tests are successful, the research group can spend the time and money necessary to incorporate the piece of equipment into the actual AUV.

The source code for the AUV driving the virtual world was written in full confidence that it would be integrated with the actual AUV. Also it was written to run either on a SGI machine or a GESPAC OS-9 machine. Slots were created so that integration of actual AUV control source code might be implemented independently. It is the user's choice to either run the robot software on a networked SGI system, or the actual

AUV operated under the GESPAC OS-9 system.

1. Pre-Mission Testing

Pre-mission testing is most valuable because it gives several people the ability to work on software modules simultaneously that will later be fitted together. This distributed software development capability proved to be the integrated simulator's finest asset in the early development of a fully functional AUV (Brutzman 96). Logic and data flow throughout the software package can be developed and tested prior to the finished product that will ultimately run the actual AUV. Timing, software functionality and communications between processors can also be tested.

2. Pseudo-Mission Testing

Pseudo-mission testing in a virtual environment (e.g. a modeled test tank that is the same scale as the actual one) provides the capability to run missions in the 3D world while monitoring robot response to ensure mission tasks are completed as expected. Along with software testing, the simulator also allows the actual AUV hardware to be bench tested or end-to-end tested. The simulator was developed so that it could be run on a variety of computers. This enables different levels of the architecture to be run together in the boat, separately outside the boat, or some combination of inside and outside. Also it enables the use of either (or both) of the AUV's computers to be run. An example of this is the ability to run the *tactical* level on a networked SGI computer while using the GESPAC/OS-9 that is in the AUV to run the *execution* level. With the onboard computer running the *execution* level, the user is able to visually inspect the rudders, planes, propellers and gyros to see if these components are rotating in the proper directions with out actually immersing the AUV. With a powerful offboard computer running *tactical* or *strategic* level code, computationally intensive artificial intelligence (AI) modules can be tested and debugged quickly. By such means, all vehicle software and hardware can be tested in the pseudo-mission environment.

3. Post-Mission Playback

Post-mission playback is most helpful for evaluation of the finished product. It is instrumental in the fine tuning of the control constants used for the motion of the AUV. Determining exactly how much voltage is required for components such as the thrusters and propellers to obtain a desired motion is one of the most time-consuming areas in the development of the AUV. Without post-mission playback, engineers fall into guess work and eyeball approximations of how much voltage is needed to get a desired movement. This often requires many immersed tests to get close to what is believed to be the correct values. Post-mission playback in real time uses vehicle telemetry (system state values) that were recorded during an actual mission. Because these are updated in a synchronous manner, they depict exactly where the AUV was and its orientation at any interval of time. With this knowledge of where the vehicle actually was and where the vehicle thought it was, designers are better able to modify the control constants that govern vehicle motion.

4. Execution Flow Chart

A detailed flow description for the combined *execution* level can be found in Chapter V. Figure 9 is a block flow chart that generalizes the operation of the preliminary virtual world *execution* level. This is the version that was obtained at the start of this thesis research.

F. SUMMARY

During the development of the *Phoenix* AUV, virtual-environment and actual-environment versions of the *execution* level control software were created. Although stemming from a common architecture these two versions progressed at different rates. This thesis describes building a software bridge between the AUV and its virtual world so that the control and logic of the actual AUV uses the same control and logic as the virtual AUV. With consistent control and logic for both worlds, the same code can be used to run in either the virtual environment or the actual world. Therefore, changes to the code in the virtual world can be tested and corrected in the virtual world. When modifications

1.) When the main execution program is started, arguments can be included. These arguments are read and parsed by `parse_command_line_flags`.

1.) Read and parse command-line arguments

2.) A virtual world host is identified or a mission telemetry file is identified for playback.

2.) Set virtual world host or identify playback file

3.) The next operation that takes place is a time step. This it the amount of time between updates to the "state vector" that is explained later. The default value for `dt` is .1 seconds or 10 Hz. Going slower then this may cause the vehicle to become unstable. A faster time step is unnecessary and promotes over kill.

3.) Set `dt` = .1 (default for timestep)

4.) `parse_command_line_flags` optionally sets numerous toggle variables identified in Table 1. Normally these values are set to default values. Also prints a list of valid keywords if help is needed.

4.) `parse_command_line_flags`

Figure 9a *execution* level flow chart

5.) `get_control_constants` allows the user to set each of the control constants. Alternatively, these constants can be read in from the default file `control.constants.input`.

5.) `get_control_constants`

6.) `inialize_dacs` takes the default values and sets planes, rudders, motors, etc. to zero and opens digital-analog (DA) channels for communication.

6.) `initialize_dacs`

7.) `initialize_adcs` opens analog-digital (AD) channels to make it possible to take readings from planes, rudders, motors, etc...

7.) `initialize_adcs`

8.) opens GESPAC ports "t1" for a serial-path or "tt" for a high baud rate. This is opened for both reading and writing. If a sonar is installed it also opens "t3" for sonar communication.

8.) `open_device_paths`

9.) `record_data_on` opens two files to write the telemetry vector or "state vector" into. `mission.output.telemetry` updates at a rate of `dt`. `mission.output.1_second` updates every second so that a shorter version is available.

9.) `record_data_on`

10.) opens the socket to the virtual world.

10.) `open_virtual_world_socket`

11.) Gets an address to output a recorded mission report.

11.) Get an E-mail address

Figure 9b *execution* level flow chart (continued).

12.) Time given to move the in-water AUV to the starting position.

12.) Position the AUV

13.) parse_mission_script_commands takes its inputs from the movement orders. It opens a file to output a state vector whenever a block of data is collected. This block of data is brought in one line at a time, or order at a time, through the command_buffer and each time a line is read toggles are set and variables are set pertaining to that command. A block of data consists of a series of orders followed by a time to wait or a length of time given to perform those tasks. When a block of data is collected parse_mission_script_command is halted. It waits until it is called upon again by the closed loop module to alter another block.

13) parse_mission_script_commands

13a.) Open script file for reading

13b.) Set orders file for writing

13c.) get line, put in command_buffer

13d.) Set toggles and move some variables

14.) Zero pitch, roll, roll_rate, pitch_rate, yaw_rate, z_val0, and dg_offset according to how the AUV is positioned.

14) zero_gyro_data

15.) Centers the sonar relative to the AUV longitunal centerline.

15) center_sonar

16.) Used to set up the module that actually sends the commands to the hardware to make the AUV move.

16) initialize closed_loop_control_module

17.) Get a clock time to reference the waits or runs from parse_mission_script_commands.

17.) clock( )

Figure 9c *execution* level flow chart (continued).

18.) The `closed_loop_control_module` is the heart of the execution level. This is where the AUV is actually moved. It uses the variables and toggles that were altered in the block of data by the `parse_mission_script_command`. These variables and toggles are manipulated and passed to the lowest level of the program. The variables are then sent to the "dacs" where they are converted to voltages. These voltages are then applied to the proper components of the AUV. The AUV then moves.

Readings from the components are then taken by "adcs" and compared to what they should be. The errors are then corrected and the AUV travels in manner commanded. The AUV continues to travel as directed while the `closed_loop_control_module` loops until the time for the next command. When this time is reached `parse_mission_script_command` is again called and the next block of data replaces the block of data that is currently be performed and the AUV carries out the next command. This process continues until the mission is completed.

18.) `closed_loop_control_module ( )`
(Described in detail in Chapter V)

19.) At the end of each scripted mission is the toggle QUIT, STOP, or EXIT. This toggle sets the `end_test` variable to TRUE and the program is allowed to exit the `closed_loop_control_module`. Gyro's are centered, sockets are closed, device paths are closed, recorded data files are closed and stored. If in keyboard mode the user inputs one of these toggles.

19.) `End_test`

Figure 9d *execution* level flow chart end.

test satisfactorily in the virtual world, the same code with the corrections can be loaded into the actual AUV and run. This enables most coding errors to be corrected prior to taking the time and deployment effort necessary to repeat tests in the actual world. Combining the codes ensures that the advancement in one domain will be an advancement in the other. The intention of this thesis is to reintegrate both the control and the logic of the virtual and actual AUV's so that, following this thesis, both versions may advance together at a rapid pace.

III. PROBLEM STATEMENT

A. INTRODUCTION

This chapter discusses the different obstacles that had to be overcome in combining virtual and actual robot control code. The question in section B was the most important to answer and served as the basis for taking on the project. Design questions in subsequent sections were posed prior to actually performing the project. It was originally expected that if the robot control code for each domain (real and virtual) was properly combined, the results of the project might answer and validate the first question. These questions and potential answers provide the design criteria for this project.

B. WILL THE VIRTUAL WORLD BE HELPFUL?

At the root of this project is the question, "Will integrating the software control system of a virtual world AUV and the software control system for an actual AUV be beneficial to the ongoing AUV research being done at NPS?" Theoretically, if the virtual world AUV can realistically perform the tasks described in Chapter II (pre-mission testing, pseudo-mission testing, and post-mission playback) it might prove to be essential to the research. In fact, if integrating the virtual world AUV software with the actual AUV software might achieve any of these three capabilities, the project is well worth the time and effort invested. This project attempts to utilize the pre-mission and pseudo-mission testing capabilities in preparing the AUV for the Moss Landing mission (Leonhardt 96).

C. WHICH VERSION (ACTUAL OR VIRTUAL) WILL BE THE BASE FOR THE COMBINED ROBOT SOFTWARE?

A major consideration was which of the two *execution* control codes to use as a base for the combined software. The actual AUV code (Marco 96) was working and limited experiments on in-water stability were successful. Although the actual AUV code was precise and operated in an optimal 10 Hz range, the code was very large,

unmodularized and hard to understand due to cryptic variable naming conventions. The virtual AUV code (Brutzman 94) was also working and experiments in the virtual world were successful. Although the virtual AUV code was robust and modularized, it was untested in a real environment. Even though easier to read, the large size and overall complexity of the virtual AUV *execution* level code also left it hard to understand. Despite originating from a single code base, two years of separate incremental changes led to significant differences within a similar structure. This led to the question "Which code ought to be imported into the other code in order to complete the project?"

D. MESSAGE PASSING TO TACTICAL

Message passing between the *tactical* and *execution* level was a crucial specification that needed to be addressed prior to the start of the project for two reasons. The first reason was to free the other software developers to design their individual modules without a need or dependency on the other engineers and students. The individual developers might simulate inputs and outputs to their modules and test the coherency of their code prior to integrating the code into the complete package. Because of the large number of people in the group developing different portions of the software, identifying exactly which people needed what information to run their code was essential.

The second reason was to quickly determine if the project is feasible in the sense that message passing takes time, and in the *execution* level the total time needed to complete a closed loop control cycle is critical. Too much message passing slows the system and causes the AUV to become physically unstable. Not enough information getting passed causes the AUV to operate blindly for excessively long intervals of time which again causes the vehicle to become unstable. These ideas led to the question "How will the *tactical* and *execution* levels communicate and what information will be passed?"

E. WHAT ABOUT 10 Hz STABILITY?

As shown empirically in previous tests, 10 Hz enables the *execution* level to maintain vehicle stability. One of the biggest questions that needed to be answered

through testing was, "Will using the robust *execution* structure used in the virtual world code allow the vehicle to maintain a 10 Hz *execution* level update rate?" This speed problem alone might cause this project to fail. The only way to answer this question was to combine the codes and run a mission experiment. This was a gamble justified by the fact that each version had a similar structure, a common basis in source code (albeit two years old), reliance on virtual world testability, and faith in the authors ability to write effective source code. It was not expected that it would be possible to "buy out of" slow processor problems due to lack of research funding, time constraints, and a problematic set of hardware interfaces and software drivers for analog components.

F. SUMMARY

There were many questions that needed to be addressed prior to taking on the project. The biggest question remains "Will integrating the virtual world benefit the ongoing AUV research?" These questions were addressed by the Phoenix research group one by one until there was enough evidence to support combining the code. Once it was agreed how the project was to be undertaken, these questions served as a guide on the design of the new software package.

IV. SOFTWARE INTEGRATION METHODOLOGY

A. INTRODUCTION

There are many requirements for the real/virtual software integration challenge; consequently the task was broken down into to distinct problems. The first problem was to collate and integrate the combined software and get it to a stage that it might compile and run using both SGI and OS-9/GESPAC operating systems and also act as a driver to the virtual world. The second problem was to put the software into the actual AUV and watch it control the boat hardware, first on a test bench then in the water. This chapter discusses the process of integrating the combined software so that it will compile and run on a GESPAC system. Chapter VI discusses in detail what later happens with the code after it is generated and put into the AUV.

B. COMBINING THE SOFTWARE

As discussed in Chapter III, the decision was made to start with the virtual world code and extend it using the actual world code. The virtual code was already organized in such a way that sections of the actual AUV code might be transported and upgraded one at a time in corresponding modules. Also prior to a flooding accident that delayed the AUV project by a year in 1994, the majority of these slots already had actual functions written that theoretically still sent the correct signals to the correct hardware (and visa versa). However, during the rebuild of the physical AUV, additional components were set into place and hardware had to be rewired to account for circuit boards that were lost or upgraded as a result of the mishap. For these reasons many hardware interface functions that were written in the virtual code structure were no longer valid and needed to be updated so that the correct signals would go to the correct hardware components. Also some of the functions in the virtual code did not account for the fact that an option to run in either virtual or actual worlds needed to be present. Functions involving the *execution* level sonar were excluded from the initial conversion. The sonar functions were deferred due to a separate sonar residing in the *tactical* level that was being

developed at the same time. The *tactical* level sonar was able to produce all necessary information needed for the AUV's initial voyages (Campbell 96). A sonar in the *execution* level was planned to be developed after the basic *tactical* sonar code was proved successful. These decisions were made in the interest of time and to help focus on the overall goal. The ultimate conversion goal was to combine the software to run identically in the virtual and actual world, then run the Moss Landing mission.

The first order of business was to obtain the complete reference versions of the control source code for both the virtual and the actual AUV's . Once both sets of code were gathered they were separately analyzed to get an understanding of information flow (virtual source code is presented in block form in Figure 9). After there was a thorough understanding of how each source code worked, the two were compared line-by-line in order to get the maximum combined functionality; i.e. the best of both worlds.

The next step was to isolate individual functions in the actual code and abstract them to a level of movement. For example, if rotating the rudders was the function at hand, the function in the actual code that rotated the rudders was found, then all supporting functions to accomplish that task were also isolated and as a package the function and its support functions were brought into the virtual code. The abstracted function went into the virtual rotate rudders function as a condition of the virtual function, and the support functions went at the end of the source code as complete functions. The support functions went to the end of the source code unchanged instead of being inserted into the virtual abstracted function in order to minimize the amount of code that needed to be rewritten or redesignated. This was done because often the support functions would be called on by other abstracted functions. Working on one function set at a time (e.g. rudder control) allowed for incremental testing of combined results.

As actual code functions were being imported to the virtual code, the names of the functions were improved for readability and consistency. After each function was updated with the actual version of code, the function was marked with a comment as being verified that a match had been found for it in the actual code. All other functions in

the virtual code were marked as "not yet updated." This was to ensure that all functions were visited at least once during the combining stage. After each function had been marked as being updated, the code was compiled under the SGI compiler (to ensure at least in the static condition) all components of each function were present. At a minimum of once at the end of each update session the new code was both compiled and a mission was run in the virtual world to check for blatant run-time errors. Once all essential functions had been updated compiled on a SGI computer and tested satisfactorily in the virtual world, the time came to attempt to compile the code on a OS-9/GESPAC computer. Table 1 gives a step by step approach to combining the software.

1) Obtain a complete set of both source codes.
2) Understand both source codes.
3) Maximize the functionality of both source code.
4) Mark all virtual functions as "not yet updated".
5) Isolate each abstracted movement function in the actual code with the support functions that are needed.
6) Find the slot in the virtual code for that abstracted function then import it.
7) Improve the function name for clarity and consistency.
8) Change "not yet updated" to "verified match and updated" for the imported function.
9) Compile on an SGI system to ensure all support functions are present, updated, and variables are consistent.
10) Perform periodic run tests at the end of each update session to check for blatant run time errors.
11) Ensure all necessary functions in the virtual world were updated.
12) Compile on an OS-9 system.
13) Run on an OS-9 system.

Table 1 Steps to combine the software

C. COMPILING THE SOFTWARE

Although the *execution* source code would ultimately have to run on an OS-9/GESPAC operating system, it was thought best to get it to compile on an SGI system

first because it took significantly less time to access an SGI system and compile. If the code compiles on the SGI system then it is at least syntactically correct. However, because of some small differences between the SGI and OS-9/GESPAC systems there would need to be minor adjustments made to allow the OS-9 system to execute the code. Some of these small differences due to the operating systems are: OS-9 does not recognize "rm" or "cp" as remove or copy a file therefore, "del" and "copy" need to be used in their place. Other subtle differences are "cat" in SGI is "list" in OS-9, and the SGI does not have the same `tsleep()` function as the OS-9 system, instead the SGI has the `sleep()` function. A further set of complications were that the OS-9 C compiler is not ANSI C compliant (e.g. library names are different, input format for reading a double is "`%F`" instead of "`%lf`", etc.). Other details that had to be strictly observed: the OS-9-specific code had to be kept completely isolated from the SGI-specific code or fatal errors would occur. Such errors might be due to the virtual world version mistakenly trying to take readings from GESPAC circuit boards in the actual AUV, instead of calculating the readings via its virtual world resources. Similarly, if not completely isolated, the virtual code might try to open paths of communication that were intended for the OS-9 system that are not necessary and unavailable to the virtual world. Such mistakes typically cause run-time system crashes.

The AUV research group had two OS-9/GESPAC systems at its disposal. One of the systems was aboard the AUV and a second spare was networked in the AUV group laboratory. The first attempt for compilation of the combined source code on the OS-9 system was done on the spare, because that system was networked (os9.me.nps.navy.mil) and had a compiler installed. Initially the AUV GESPAC had neither network connections nor a compiler. The thought here was that if a successful compilation of executable code could be obtained from the spare OS-9, that code could then be loaded directly via diskette into the OS-9 that was onboard the AUV and bench tests might then be attempted.

For remote OS-9 compilation, the spare AUV was remotely logged onto and a copy of the combined source code was downloaded to be compiled. Syntactic errors were

corrected and paths to store information were established. Despite OS-9 C compiler noncompliance and poor documentation, eventually the source code was able to compile successfully. Once successfully compiled, running the executable code on the spare OS-9 was attempted. This was to test for run time errors in the code prior to putting the code into the actual AUV. The majority of errors that were found during this time were expected since the lab GESPAC was not able to take readings from hardware that is normally present when the code is in the actual AUV. Several runs on the lab GESPAC were made replacing hardware reads with dummy values in order completely execute an entire mission. Eventually, a mission with dummy input/output was completed successfully using the spare OS-9 to run the *execution* level.

The work described in this chapter was extremely challenging, taking four months to complete. The next step was to load this executable code into the OS-9 that was in the AUV and begin bench tests. That sequence of events is described in Chapter VI.

D. SUMMARY

After separating software integration and verification tasks into two distinct problems, combining the source code was started. The goal of this portion of the project was to get the code to compile and run on the spare GESPAC operating system. After many trial and errors due to dissimilar operating systems, the two source code versions were successfully merged into one version which ran on the spare GESPAC OS-9 system.

The code was now at a point where it was ready for bench tests in the actual AUV. This follow-on part of the project is explained in Chapter VI, Experimental Results. The next chapter, Source Code Description, reviews in detail the combined *execution* level source code produced by the software integration methodology presented in this chapter.

V. SOURCE CODE DESCRIPTION

A. INTRODUCTION

This chapter discusses in depth the integrated virtual world/real world software that controls the AUV. At the heart of the *execution* level control code is the function `closed_loop_control_module`. This is a closed-loop function that runs continuously either in a real-time or non-real-time domain. This control loop calls on several smaller functions to read, update, and command the AUV's controlling hardware. Once through each loop, the `closed_loop_control_module` allows commands to be entered or other parameters to be updated by calling on the `parse_functions` module.

As part of the *execution* source code prior to entering the `closed_loop_control_module` the DiveTracker process is spawned. The Dive_tracker process runs concurrent with the `closed_loop_control_module` and receives navigational data that is passed up to the *tactical* level. This chapter is explained by first introducing large functions called by the control loop, then gives definitions of terms that will be used in this chapter. The last sections explain the progression of the main program prior to the `closed_loop_control_module`, the `closed_loop_control_module` itself, and what happens after the `closed_loop_control_module` is exited.

B. PARSING COMMANDS

There are two large modules other than the `closed_loop_control_module` that make up a majority of the *execution* program. The first module consists of several functions that parse commands and update global variables. The second is the DiveTracker process that runs concurrent with the closed loop.

The parsing functions are divided into two categories. Both categories operate on the same principle. They take an input string, perform a keyword comparison against a list of known commands, and when a match is made set toggle variables and/or update

system parameters as appropriate. A toggle variable is defined as a unique string corresponding to a global flag variable that can either be set to a 0 (FALSE) or a 1 (TRUE). The difference between the two parsing functions is that `parse_command_line_flags` is performed once prior to the control loop at initial program invocation, while `parse_mission_script_commands` is performed repeatedly for each `closed_loop_control_module` loop.

The function `parse_command_line_flags` is used one time during a mission. It sets up the initial conditions that are intended to be used during the mission by parsing command line commands entered at invocation. It describes how the mission is to be performed and where the orders will be coming from. For logical reasons, `parse_command_line_flags` includes only a subset of the commands found in `parse_mission_script_commands`. As an example, to start the *execution* level the user might type: ***execution keyboard real-time silent***, then the user would press enter. The user needs to type ***execution*** first because it is the name of the *execution* level program (executable code), and therefore starts the main program. After ***execution*** is typed the user can add as many (or no) additional attributes desired for the intended mission. If no attribute keywords are set by the user, default toggle variable values are used. The attributes that are used in this example are: ***keyboard*** lets the system know that the user will be inputting commands from the keyboard instead of the default `mission.script` file, ***real-time*** lets the system know that a strict time schedule must be adhered to by the `closed_loop_control_module`, and ***silent*** lets the system know to turn off its added feature of having all orders echoed through the system speakers via a Netherlands-based voice synthesis server (Brutzman 94). A list of possible toggles that can be used in initializing a mission can be found in Table 2. This table gives a listing of the commands used, toggles that are reset, globals that are updated, and a brief description of what happens. All keywords and values are case insensitive. Commands and default values are described in more detail in Appendix F, `mission.script.HELP`.

Table 2 `parse_command_line_flags` keywords

Commands	Toggle Variables Reset	Other Global Variables Updated	Description
HELP ?	N/A	print_valid_keywords	Offers possible toggles
KEYBOARD KEY-BOARD	KEYBOARDINPUT ← TRUE	N/A	Commands will come from the keyboard
TRACEOFF TRACE-OFF NO-TRACE	TRACE ← FALSE	N/A	Disables the ability to trace a running program
TRACE TRACE-ON	TRACE ← TRUE	N/A	Enables a trace of the running program for error correction
LOOPFOREVER	LOOPFOREVER ← TRUE	N/A	Program cycles through multiple times without stopping
LOOPONCE LOOP-ONCE	LOOPFOREVER ← FALSE	N/A	Program runs mission one time
LOOPFILEBACKUP	LOOPFILEBACKUP ← TRUE	N/A	Allows a backup copy of the running program results
ENTERCONTROLCONS TANTS	ENTERCONTROLCONSTANT S ← TRUE	N/A	Allows the user to manually enter the control constants for AUV motion
TACTICAL TACTICAL-HOST	TACTICAL ← TRUE KEYBOARDINPUT ← FALSE	get tactical hostname/IP address	Allows AUV execution level to communicate with tactical level
SILENT SILENCE	AUDIBLE ← FALSE	send SILENT to virtual world buffer	Disables the audio playback
TIMESTEP TIME-STEP	N/A	dt ← TIMESTEP	Allows the user to alter the time required for a single closed-loop cycle
VIRTUALHOST REMOTE DYNAMICS	N/A	virtual_world_remote_host_na me ← VIRTUALHOST	Identifies host running dynamics (Virtual World)

Table 2 `parse_command_line_flags` keywords (continued)

Commands	Toggle Variables Reset	Other Global Variables Updated	Description
REALTIME REAL-TIME	REALTIME ←TRUE	N/A	Allows the mission to run in real-time (busy/wait as necessary)
NOREALTIME NOPAUSE NO-WAIT	REALTIME ←FALSE	N/A	Allows the mission to run as fast as possible
LOCATIONLAB	LOCATIONLAB ← TRUE	N/A	Allows AUV to be bench tested (no gyros or sonars)
LOCATIONWATER	LOCATIONLAB ← FALSE	N/A	Allows AUV to be fully functional

The function `parse_mission_script_commands` is called at the end of each of the `closed_loop_control_module` cycles. This is where the majority of the toggles are changed. One time per cycle the control loop pauses to get its next command. Once it has received a new order, it continues in the control loop to direct the necessary functions to accomplish the new command. If input commands in the keyboard mode are provided, this break in the control loop gives the user the ability to manipulate any portion of the hardware. A list of possible toggles that can be used to direct the AUV during a mission can be found in Table 3. This table gives a listing of the commands used, toggles that are reset, globals that are updated, and a brief description of what happens. All keywords and values are case insensitive. Commands and default values are described in more detail in Appendix F, `mission.script.HELP`.

Table 3 `parse_mission_script` keywords

Commands	Toggle Variables Reset	Other Global Variables Updated	Description
HELP ?	N/A	print_valid_keywords	Offers possible toggles

Table 3 `parse_mission_script` keywords (continued).

Commands	Toggle Variables Reset	Other Global Variables Updated	Description
WAIT RUN	read_another_line ← FALSE	time_next_command = t + parameter	Stops orders for a specified amount of time
TIME WAITUNTIL	read_another_line ← FALSE	time_next_command = t + parameter if parameter < t print warning reset all velocities to 0	Tells when the next command will be accepted
TIMESTEP	N/A	dt ← TIMESTEP	can alter the time required for a single closed-loop cycle
PAUSE	N/A	getchar ( )	momentary stop of the program
REALTIME REAL-TIME	REALTIME ← TRUE	N/A	Allows realtime mission (busy/wait as necessary)
MISSION SCRIPT FILE	KEYBOARDINPUT ← FALSE	AUVSCRIPTFILENAME← parameter	read the mission commands in from a script file
KEYBOARD KEYBOARD-ON	KEYBOARDINPUT ← TRUE	N/A	Commands will come from the keyboard
KEYBOARD-OFF NO-KEYBOARD	KEYBOARDINPUT ← FALSE	N/A	keyboard will not be used to get commands
NOWAIT NO-REALTIME NO-PAUSE	REALTIME ← FALSE	N/A	allows the mission to run as fast as possible
ABORT	HALTSCRIPT ← TRUE	N/A	stop the program now
QUIT STOP EXIT	end_test ← TRUE read_another_line ← FALSE	N/A	stop the program, shutdown normally
RPM SPEED PROPS	WAYPOINTCONTROL ← FALSE ROTATECONTROL ← FALSE HOVERCONTROL ← FALSE	port_rpm_command stbd_rpm_command	tell system how fast to rotate the shafts. 700 RPM produces steady state speed of ~ 1.5 feet/sec

Table 3 `parse_mission_script` keywords (continued).

Commands	Toggle Variables Reset	Other Global Variables Updated	Description
COURSE HEADING YAW	DEADSTICKRUDDER ← FALSE WAYPOINTCONTROL ← FALSE ROTATECONTROL ← FALSE LATERALCONTROL ● FALSE if (HOVERCONTROL is true) set REPORTSTABLE ←TRUE	psi_command psi_command_hover rotate_command = 0.0 lateral_command = 0.0	a course the AUV will maintain, using current control mode
TURN CHANGE-COURSE	DEADSTICKRUDDER ← FALSE WAYPOINTCONTROL ← FALSE ROTATECONTROL ← FALSE	psi_command	conduct a turn in degrees relative to the heading, using current control mode
RUDDER	DEADSTICKRUDDER ● TRUE WAYPOINTCONTROL ← FALSE ROTATECONTROL ← FALSE HOVERCONTROL ← FALSE	rudder_command	turn rudders to a specified angle

DEADSTICKRUDDER	if an angle is sent then
	DEADSTICKRUDDER ← TRUE
	WAYPOINTCONTROL ← FALSE
	ROTATECONTROL ← FALSE
	HOVERCONTROL ← FALSE
	else
	DEADSTICKRUDDER ← TRUE
	WAYPOINTCONTROL ← FALSE
	ROTATECONTROL ← FALSE
		if an angle is included then
		rudder_command = parameter
		else
		rudder_command = 0.0
			keep rudder at a specified angle (open loop control)

Table 3 `parse_mission_script` keywords (continued).

Commands	Toggle Variables Reset	Other Global Variables Updated	Description
DEPTH	DEADSTICKPLANES ← FALSE if (HOVERCONTROL) is true then REPORTSTABLE ← TRUE	z_command	reach and maintain ordered depth using current control mode
PLANES	DEADSTICKPLANES ← TRUE	command_planes	turn planes to specified angle
DEADSTICKPLANES	DEADSTICKPLANES ← TRUE	if an angle is sent then command_planes else command_planes = 0.0	keep planes at a specified angle (open loop control)
THRUSTERS THRUSTERS-ON	THRUSTERCONTROL ←TRUE	N/A	allow the thrusters to be used
NOTHRUSTERS THRUSTERSOFF	THRUSTERCONTROL ← FALSE ROTATECONTROL ← FALSE HOVERCONTROL ← FALSE LATERALCONTROL ← FALSE	N/A	discontinue thruster use
ROTATE	THRUSTERCONTROL ← TRUE ROTATECONTROL ← TRUE HOVERCONTROL ● FALSE LATERALCONTROL ← FALSE WAYPOINTCONTROL ← FALSE	rotate_command lateral_command = 0.0	spin the vehicle around its z-axis using thrusters at a specified rate (open loop control)
NOROTATE ROTATE-OFF	ROTATECONTROL ← FALSE	rotate_command = 0.0	stop rotating
LATERAL	THRUSTERCONTROL ← TRUE ROTATECONTROL ← FALSE HOVERCONTROL ● FALSE LATERALCONTROL ← TRUE WAYPOINTCONTROL ← FALSE	rotate_command = 0.0 lateral_command	slide the AUV laterally using thrusters (open loop control)

Table 3 `parse_mission_script` keywords (continued).

Commands	Toggle Variables Reset	Other Global Variables Updated	Description
NOLATERAL LATERAL-OFF	LATERALCONTROL ← FALSE	lateral_command = 0.0	stop sliding the vehicle laterally
DIVETRACKER1	N/A	DiveTracker1_x DiveTracker1_y DiveTracker1_z	world position of divetracker transponder1
DIVETRACKER2	N/A	DiveTracker2_x DiveTracker2_y DiveTracker2_z	world position of divetracker transponder 2
GPS GPS-FIX	if TACTICALPARSE ← FALSE then GPSFIXINPROGRESS = TRUE	if TACTICALPARSE ← FALSE then previous_z_command = z_command time_gps_complete = t + 30.0 time_postgps_dive = t + 60.0 time_next_command = time_gps_complete	get a GPS fix
GPS-COMPLETE GPS-FIX-COMPLETE	if GPSFIXINPROGRESS read_another_line ← FALSE	if GPSFIXINPROGRESS z_command = previous_z_command time_postgps_dive = t + 30.0 time_next_command = time_postgps_dive time_gps_complete = time_postgps_dive + 1	finished getting a GPS fix before reading further orders
GYROERROR GYRO-ERROR	N/A	gyro_error	correct offset error in the gyro (gyro + error = true)
DEPTH-CELL-BIAS DEPTHCELLERROR DEPTHERROR	N/A	depth_cell_bias	an offset for the depth cell to obtain accurate readings in saltwater and fresh water (depth cell + error = true)

Table 3 `parse_mission_script` keywords (continued).

Commands	Toggle Variables Reset	Other Global Variables Updated	Description
LOCATIONLAB LOCATIONVIRTUAL	LOCATIONLAB ← TRUE	N/A	Allows the AUV to be bench tested or run virtually (No gyros or sonars)
LOCATIONWATER LOCATIONREAL	LOCATIONLAB ← FALSE	N/A	Allows AUV to be fully functional
LOCATION POSITION FIX	N/A	x y z	reset vehicle position estimate
ORIENTATION ROTATION	N/A	phi theta psi	reset vehicle orientation estimate
POSTURE	kal_init_z ← TRUE	x, y, z, phi, theta, psi, start_psi	how the vehicle is oriented in a fixed position
OCEANCURRENT OCEAN-CURRENT	N/A	AUV_oceancurrent_x AUV_oceancurrent_y AUV_oceancurrent_z	set and drift estimated value
CONTINUE GO	N/A	N/A	continue the loop without stopping
STEP	read_another_line ← FALSE	time_next_command = t + dt	do only one closed loop
TRACE TRACE-ON	TRACE ● TRUE	N/A	Enables a trace of the program for error diagnostics
TRACE-OFF NOTRACE	TRACE ← FALSE	N/A	Disables the ability to trace a running program
LOOP-FOREVER	LOOPFOREVER ● TRUE	N/A	mission cycles through multiple times without stopping
LOOPONCE	LOOPFOREVER ← FALSE	N/A	run the mission one time
LOOPFILEBACKUP	LOOPFILEBACKUP ← TRUE	N/A	make a backup copy of the mission results

Table 3 `parse_mission_script` keywords (continued).

Commands	Toggle Variables Reset	Other Global Variables Updated	Description
ENTERCONTROL-CONSTANTS	ENTERCONTROLCONSTANTS ← TRUE	get_control_constants ()	manually input new control constants
CONTROL-CONSTANTS-INPUT-FILE	LOADCONTROLCONSTANTS ← TRUE	get_control_constants ()	read the control constants in from a file
SLIDINGMODECOURSE	SLIDINGMODECOURSE ← TRUE ROTATECONTROL ← FALSE HOVERCONTROL ← FALSE WAYPOINTCONTROL ← FALSE	N/A	not yet implemented
SLIDINGMODEOFF	SLIDINGMODECOURSE ← FALSE	N/A	not yet implemented
TACTICAL TACTICAL-HOST	TACTICAL ← TRUE KEYBOARDINPUT ← FALSE	get tactical hostname/IP address	Allows AUV execution level to communicate with tactical
NOTACTICAL	TACTICAL ← FALSE	N/A	no orders will come from the tactical level
SONARINSTALLED	SONARINSTALLED ← TRUE	N/A	indicate that the sonar is installed
AUDIBLE AUDIO SOUND-ON	AUDIBLE ←TRUE	send_buffer_to_virtual_world_socket ()	echo over the speakers a voice representation of all orders
SILENT QUIET NO-SOUND	AUDIBLE ← FALSE	send silent to virtual world buffer	Disables the audio playback
EMAIL-ON	EMAIL ● TRUE	N/A	send a copy of mission results to the recipient
EMAIL-OFF	EMAIL ← FALSE	N/A	no email is sent after mission completion

Table 3 `parse_mission_script` keywords (continued).

Commands	Toggle Variables Reset	Other Global Variables Updated	Description
WAYPOINT WAYPOINT-ON	WAYPOINTCONTROL ← TRUE FOLLOWWAYPOINTMODE - ←TRUE HOVERCONTROL ← FALSE ROTATECONTROL ← FALSE LATERALCONTROL ← FALSE REPORTSTABLE ← TRUE DEADSTICKRUDDER ← FALSE	x_command y_command z_command port_rpm_command = fabs(port_rpm_command) stbd_rpm_command = fabs(stbd_rpm_command) detect_death_spiral (true)	go through a waypoint and carry on to the next one when ordered
WAYPOINTFOLLOW WAYPOINT-FOLLOW	FOLLOWWAYPOINTMODE - ← TRUE DEADSTICKRUDDER ← FALSE	N/A	go through a waypoint and carry on to the next one
WAYPOINTFOLLOWOFF WAYPOINT-FOLLOW-OFF	FOLLOWWAYPOINTMODE - ← FALSE	N/A	do not go to next waypoint without being ordered
STANDOFF STANDOFF-DISTANCE	if (HOVERCONTROL) then REPORTSTABLE ← TRUE	standoff_distance	predetermined distance from a point to be reached for hoverpoint/waypoint success
HOVEROFF HOVER-OFF	WAYPOINTCONTROL ← FALSE FOLLOWWAYPOINTMODE ←FALSE HOVERCONTROL ← FALSE read_another_line ← FALSE	port_rpm_command = 0.0 stbd_rpm_command = 0.0 rudder_command = 0.0	Turn off the hover mode

Table 3 `parse_mission_script` keywords end.

Commands	Toggle Variables Reset	Other Global Variables Updated	Description
HOVER HOVER-ON	HOVERCONTROL ← TRUE REPORTSTABLE ← TRUE WAYPOINTCONTROL ← FALSE ROTATECONTROL ← FALSE LATERALCONTROL ← FALSE THRUSTERCONTROL ← TRUE DEADSTICKRUDDER ← TRUE	x_command y_command z_command rudder_command = 0.0 psi_command psi_command_hover standoff_distance	use thrusters to get into position and stay there

C. BEGIN THE MAIN PROGRAM

To begin the main program, the timestep variable `dt` is set to the default setting. The timestep is the most important factor in maintaining real-time response. A timestep is the maximum fraction of a second that is required to run one complete control loop. As an example, through many iterations and tests it was found that the revised `closed_loop_control_module` requires a maximum of .15 seconds to complete the most complex loop. Because the control loop is never expected to take more than .15 seconds to complete, the timestep is then set to .15 seconds. If the REALTIME toggle is set by the user, each time a loop is completed it is held idle in a busy-wait loop until .15 seconds has transpired. This is how the AUV maintains a 6.66 Hz control loop. Therefore, all positioning and control algorithms and filters have a constant time duration between loops in which they use to calculate how fast the AUV is moving in a given direction and consequently where the AUV should be with respect to time. If the user does not select the REALTIME option the computer is free to run without pausing through the loops. Without pausing at the end of each loop, the overall mission takes a shorter time to complete. This is ideal for testing logic in the 3D virtual world, because the user does not have to wait as long for results.

Next the initial conditions for the mission are read by `parse_command_line_flags` (explained earlier). These initial conditions indicate to the system all the appropriate paths and sockets to open for either transfer or storage of recorded data.

If needed, the next step is to spawn the DiveTracker process. This is only used if the AUV is in water and the user wishes to navigate via DiveTracker. The DiveTracker process is a three- beacon acoustic interrogation system that is used for navigation (Scrivener 96). It consists of a baseline of two beacons at a known distance from each other and a third beacon that is mobile. The baseline pings the mobile unit and awaits a response. The time required to receive a response is then translated into a distance. The mobile unit receives the distances to each of the baseline beacons and by triangulation can determine where it is in the relative coordinate system.

Continuing execution, the AUV begins its hardware initialization. Time cards are first initialized and then the gyros are uncaged. Uncaging the gyros allows them to spin freely. While uncaged the gyros are highly susceptible to damage, therefore whenever they are not being used they are left caged. All rudder and plane surfaces are zeroed, and propellers and thrusters are zeroed and turned off. This is to prevent accidentally starting a piece of hardware due to uninitialized values left in registers. A source of power is then given to the propellers and thrusters. Analog-to-Digital and Digital-to-Analog converters are made ready and the gyros are zeroed. A set of control constants is next retrieved. All of these control constants are parameterized. They are normally read in during the initialization from the `control.constants.input` file. These constants are used in the control response formulas that dictate the movement of the AUV.

A do-while loop is then entered. This gives the code between the "do" and the "while" the flexibility to be executed multiple times (if necessary) but the loop is always executed at least once. The LOOPFOREVER toggle is the condition of the while. If set to TRUE this portion of code which includes the control loop will be executed repetitively. The primary purpose of this loop is to test the AUV's logic endurance and robustness, particularly with respect to memory leaks; i.e. failure to deallocate allocated

memory (garbage collection). The LOOPFOREVER toggle is normally used with a prepared `mission.script` for the 3D world. In this mode the user can test how long a mission will run continuously. Also, during these continuous runs, sockets, pipes and networks are being established and terminated. This gives the user the ability to repeatedly test all connections in the network. Such testing also exercises hardware for extended periods and has proved very useful in improving vehicle robustness. While in this mode, simulation time is reset to zero at the end of each mission so that multiple mission results may be produced identically.

Next, if the user is working in the 3D world remotely and the correct toggles have been set, the user is given the opportunity to be e-mailed a copy of the recorded data after the mission. When prompted, the user need only enter an e-mail address.

If the sonar is installed it is centered at this time. The closed-loop control parameters are also set. The principal parameter for the closed loop is "end_test" which must be set to FALSE. The last thing done prior to entering the closed loop is to calculate a time for the next loop. This will be used later. Once this is done, the closed-loop commences and will continue to loop until end_test is set to TRUE.

D. CLOSED_LOOP_CONTROL_MODULE

This is the function that coordinates all motion and maintains the stability of the AUV. Each function and logic sequence that is used either in or by the `closed_loop_control_module` is written to accommodate two modes of operation. These two modes are clearly separated by the toggle variable LOCATIONLAB. If LOCATIONLAB is TRUE all readings are computer generated by the virtual world and all commands go to the virtual world. If LOCATIONLAB is FALSE readings are taken from the hardware components of the AUV and all commands go to the hardware components of the AUV. Command results are viewed by looking at the virtual world viewer when LOCATIONLAB is TRUE, or the actual AUV (either on a test bench or in the water) when LOCATIONLAB is FALSE.

The first priority of the `closed_loop_control_module` is to test for

critical conditions that mandate mission termination. The loop first checks to see if the AUV is in the water and a terminal condition has not already been cited. If it is in the water and there is no previously triggered crisis, it then checks the computer battery and the motor battery. If both of these voltages are above 20 volts, operation continues. Next it checks a leak detector. If no moisture is sensed inside the hull of the AUV the program checks the depth cell. The maximum depth is currently set at six feet for testing purposes. Currently if the AUV goes deeper then six feet the mission will be terminated. This prevents an uncontrolled or unordered dive as well as system shutdown if a flooding casualty occurs. If the maximum depth has not been violated, the program then makes a final critical check to determine whether or not the DiveTracker transducer has lost communication with the baseline transponders. For testing purposes the code is written so that if there is no DiveTracker update within 10 seconds, it is then assumed communication with the base station has been lost. If communication with the base station is lost the AUV can not reliably tell where it is under the water (due to current unsatisfactory dead reckoning). If the AUV does not know its location in the water it does not have a reference point from which to navigate, therefore the program terminates for safety reasons. If any of these checks show that there is a problem, the program automatically enters a shutdown script that forces the AUV to the surface. Figure 10 gives a quick reference to critical checks.

Computer Battery	>	More than 20 Volts
Motor Battery	>	More than 20 Volts
Leak Detector	→	Must be dry
Depth Cell	<	Register less than 6 Feet
DiveTracker	<	Update less than 10 seconds

Figure 10 Critical checks for mission termination

If there are no critical problems encountered, the loop will continue and either simulated or actual readings are taken from the appropriate functions. Each of these functions have also been designed to test whether LOCATIONLAB is TRUE or FALSE. These readings update the vehicles telemetry vector, and allow the vehicle to evaluate its motion during the previous timestep. These readings are also used for dead reckoning which predicts vehicle position based on course, speed and the current time. Functions that update state variables are in Figure 11.

STATE VARIABLE		SENSOR FUNCTION
speed	←	read_speed ( )
port_rpm	←	read_port_motor_rpm ( )
stbd_rpm	←	read_stbd_motor_rpm ( )
x	←	XY_model_set ( )
y	←	XY_model_set ( )
z	←	read_depth ( )
phi	←	read_roll_angle ( )
theta	←	read_pitch_angle ( )
psi	←	read_psi ( )
u	←	XY_model_set ( )
v	←	XY_model_set ( )
w	←	z_dot_kal
kalman_z (z)	------	to smooth this transition and get other values
z_dot ←	z_dot_kal	
p	←	read_roll_rate_gyro ( )
q	←	read_pitch_rate_gyro ( )
r	←	psi - last psi / dt
		normally from read_yaw_rate_gyro ( ), but yaw rate gyro is inoperative at present time.

Figure 11 State variables and sensor functions

The next operation that takes place allows the previously spawned DiveTracker process to update the telemetry state vector. The DiveTracker ranges from beacon 1 and beacon 2 are accepted only if the AUV depth is greater than 1 foot, since ranges may be

inaccurate when the vehicle is surfaced. The telemetry vector is then sent to the *tactical* level. These ranges are filtered in the *tactical* level and passed back down to the *execution* level as the vehicles updated x and y positions.

This portion of code (like many others) is encapsulated by `#if, #else, #endif` preprocessor directives. These sections of code are setup to accommodate variations in the different operating systems and compilers that are used in running the combined source code. The OS-9 GESPAC system has a unique style and a different set of reserved words than both the SGI and SUN systems. However, all three systems recognize the left justified `#if, #then, #else` "C" language preprocessor directives. Periodically in the *execution* code the format in Figure 12 is found.

```
#if (defined(sun) || defined(sgi))
#else
      /* Dive Tracker Stuff */
      if (DIVETRACKER)
      {
         createdmod();

         /* Fork Dive Tracker Process */
         if( (dt_pid =
            OS-9fork("/r0/div_trac",0,dt_fork_parmptr,0,0,0,0)) > 0)
         {
            printf("[Dive Tracker Process %d forked]\n",dt_pid);
         }
         else
         {
            printf("[Can't Fork Dive Tracker Process]\n");
         }
      }
#endif
```

Figure 12 Operating system independence: permitting syntax variations using preprocessor directives.

This means if the compiler is running on a SUN or SGI workstation compile the "if" portion of the source code (in this case do nothing), but if the compiler is not one of these, compile the "else" portion of the source code (spawn the DiveTracker). This is one way to segregate the source code so that it is accepted by all of the compilers.

The telemetry state vector (mentioned earlier), is an array containing 37 global

variables. Originally it was constructed to record the AUV's telemetry as the AUV conducted a mission. At the end of a mission, the state vector can be played back in the virtual world to determine if the AUV performed as expected. If a fault occurred during a mission, the state vector would help determine the cause of the fault. Later the state vector proved to be an even more valuable tool, by providing the means to transfer all essential data from process to process and computer to computer. The information contained in the state vector not only includes the vehicle telemetry, but also includes all inputs and outputs needed by individual processes at the *tactical* level. All processes needing to communicate with other processes get a continually updated copy of the state vector. The individual processes receive the entire state vector and extract the information they need. Thus from the perspective of the *tactical* level processes, the telemetry vector is acting as though it is shared memory which is updated solely by the *execution* level. The rationale of state vector communications is described in (Brutzman 96) (Leonhardt 96).

Next a waypoint_distance is calculated. A waypoint_distance is defined as the distance between current AUV position, and the next waypoint the AUV is trying to reach. This waypoint_distance is used by both HOVERCONTROL and WAYPOINTCONTROL. Also used by these two modes of operation is the standoff_ distance. The standoff_distance is defined as a predetermined depth and radial distance away from a waypoint or hoverpoint in a 3D coordinate system. When the vehicle comes within the predetermined distance from the waypoint or hoverpoint it is trying to reach, the control algorithm declares success in obtaining that point.

Although many toggles are set and many variables updated through the next portion of code, for clarity this discussion focuses on the source code which updates shaft rpm while in HOVERCONTROL or WAYPOINTCONTROL. The major difference between these two modes of transit is that WAYPOINTCONTROL uses a fixed positive rpm (clamped at a minimum of 200), and the HOVERCONTROL uses a rpm amount that is nonlinearly proportional to how far away from the target the AUV is located. The hover code is written so that while the next hoverpoint is far away from the

AUV, WAYPOINTCONTROL will be used to rapidly approach the goal. HOVERCONTROL mode then resumes when the range to the hoverpoint becomes practical.

1. Hover Control Logic

If there is a HOVER command and the waypoint_distance is greater than the standoff_distance, an additional check is needed. This check is to see if the waypoint_distance is greater than the standoff_distance by more than 15 feet. If such is the case, the AUV temporarily switches from hover mode to waypoint mode. This switch is made because longer distances can be traveled faster in WAYPOINTCONTROL mode. Therefore the following toggles are temporarily set while HOVERCONTROL remains TRUE:

WAYPOINTCONTROL is set to TRUE
DEADSTICKRUDDER is set to FALSE
DEADSTICKPLANES is set to FALSE
port_rpm_command = 700
stbd_rpm_command = 700
psi_command is set to psi_command_hover

Temporarily changing to waypoint mode ensures the AUV will efficiently travel to the next point at maximum speed using rudders and planes to guide its way. Once the waypoint_distance is less then standoff_distance plus 15 feet, the AUV switches the WAYPOINTCONTROL mode off and makes its way to the point using HOVERCONTROL and the following toggles are set:

WAYPOINTCONTROL is set to FALSE
DEADSTICKRUDDER is set to TRUE
DEADSTICKPLANES is set to TRUE

In this version of HOVERCONTROL, the AUV acknowledges that it is too close to the point of destination to use a set rpm speed that would be produced by the WAYPOINTCONTROL and therefore sets that toggle to FALSE. It then resumes HOVERCONTROL to maneuver into position. HOVERCONTROL allows the rpms to be varied proportionally to how far away from the target the AUV is located. Also the

AUV will be traveling slow enough that it may use cross-body thrusters to obtain cross-track position and correction.

If in HOVERCONTROL and the waypoint_distance is not greater than the standoff distance, this means the AUV is close to its target. If WAYPOINTCONTROL prior to this was temporarily set to TRUE, it is now set to FALSE and

DEADSTICKRUDDER is set to TRUE
DEADSTICKPLANES is set to TRUE
rudder_command = 0.0
psi_command = psi_command_hover

This will ensure the AUV returns to a true HOVER mode using thruster and/or propellers to maneuver.

The next check in the HOVER mode is the cylinder proximity check. This check ensures that the AUV is close enough to the waypoint in three ways. The Figure 13 displays a graph that helps visualize the proximity check the AUV must comply with:

waypoint_distance	<	standoff_distance
|depth_error|	<	standoff_distance
|psi_error|	<	10.0 (degrees)

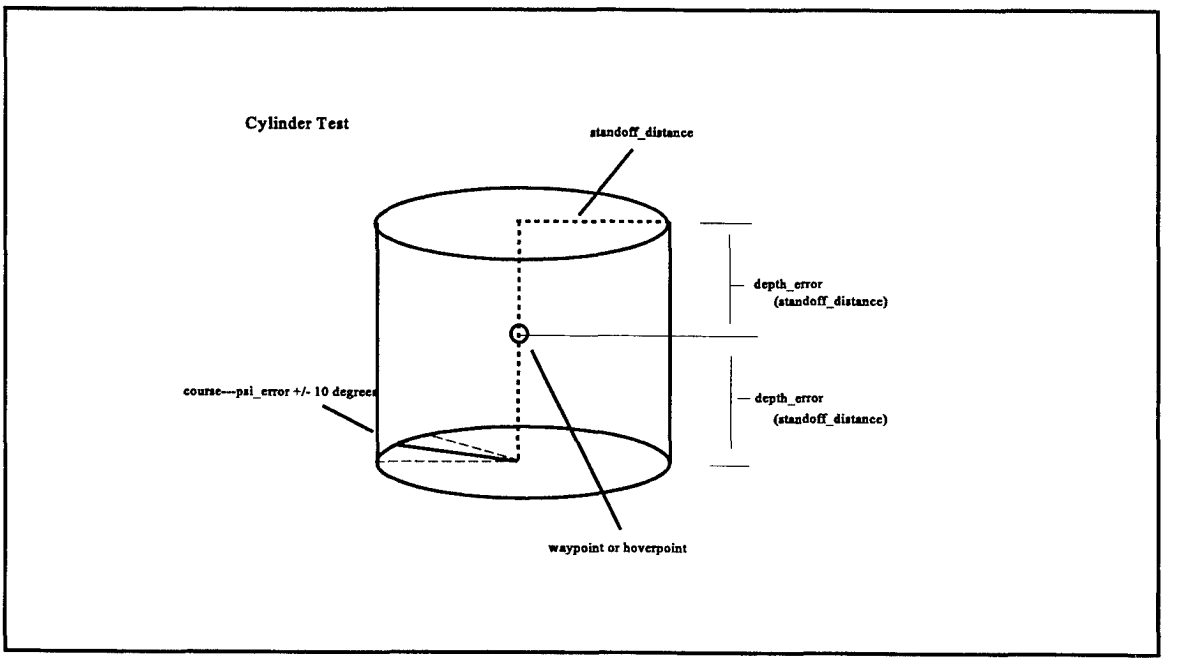

Figure 13 Cylinder test for waypoint/hoverpoint success

This states that the AUV is within standoff_distance in the radial-X-Y directions, within standoff_distance in depth error, and that actual heading is within 10 degrees of commanded heading. If these criteria are met, the AUV declares success in obtaining the predetermined waypoint/hoverpoint. Thus the intercept target is a cylindrical volume coupled with a direction requirement.

If the AUV is traveling using HOVERCONTROL, a track_angle is later calculated. A track_angle is (waypoint angle - current heading), which is the difference between the angle that the AUV needs to obtain waypoint and the AUV's actual heading. This angle is split into its X and Y components and named along_track_distance and cross_track_distance and shown in Figure 14.

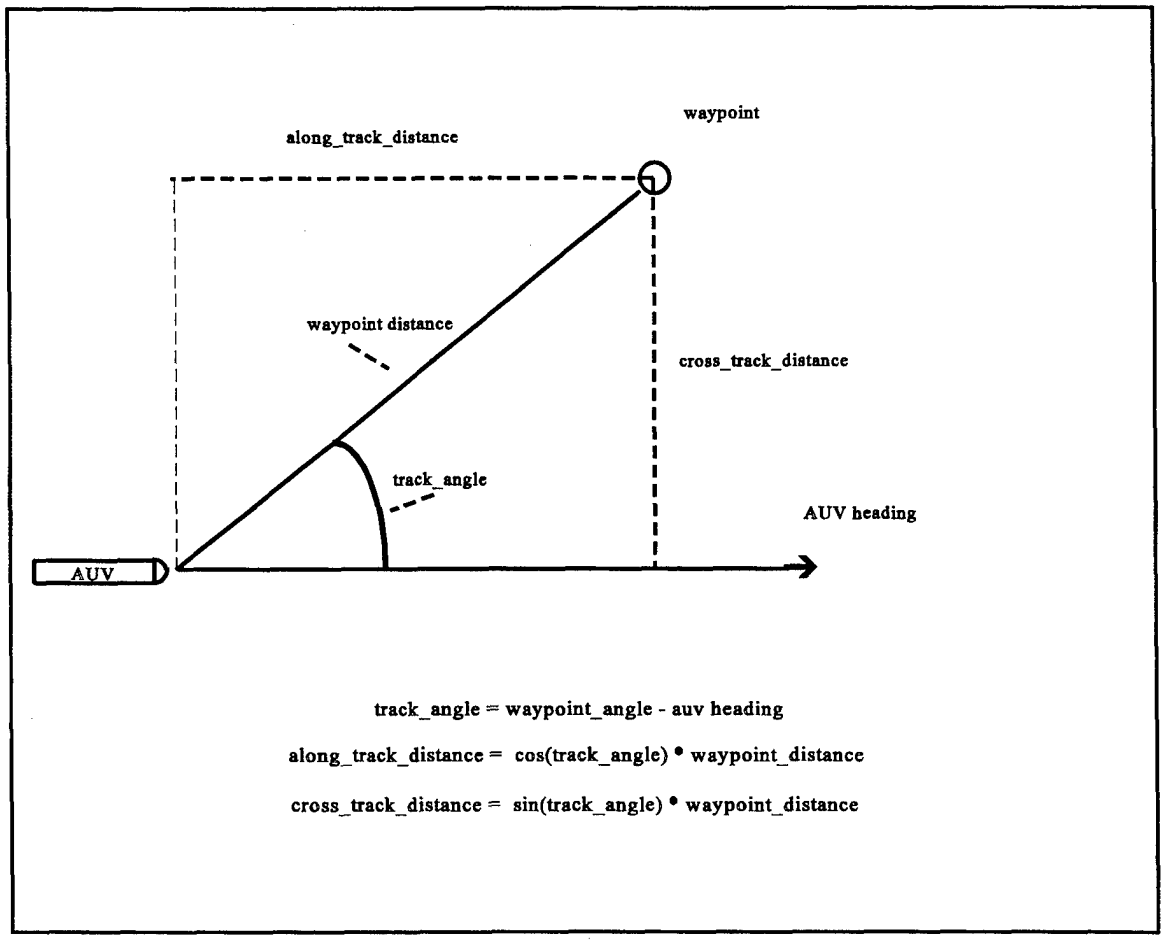

Figure 14 Closing waypoints independent of ordered heading.

These two components are used later as inputs to the formulas that drive the propellers and lateral thrusters respectively. This method permits approaching the hoverpoint satisfactorily independent of ordered AUV heading.

2. The Death Spiral

Before discussing the WAYPOINTCONTROL mode of operation it is important to understand one of the possibilities that can happen while in WAYPOINTCONTROL mode. A "death spiral" describes the behavior where the AUV drives continuous circles around a point. This situation is caused by the AUV's inability to make a sharp enough turn (using only rudders and planes while in WAYPOINTCONTROL) to get within a standoff_distance and meeting waypoint success criteria. In this pathological scenario the AUV continues to turn and propel itself at a constant rate towards the point but it will never intersect the cylinder volume which produces waypoint success. Through many tests it was determined that the AUV traveling at full speed (700 rpms), can acquire any waypoint that was more than 15 feet away. Determining factors for the AUV to acquire a point are how fast the AUV is traveling and how sharp of a turn it is able to make. Although more testing in this area may be prudent, the general formula:

$$\text{death\_spiral\_radius} = \text{abs}\,(\sin\,(\text{waypoint\_angle} - \text{heading}) * (1/700) * \text{rpms} * 15)$$

will keep the AUV from entering into a death spiral. It is based on how far away from a point (a radius of the point) the AUV is currently located. This distance is called the death_spiral_radius and is updated on every cycle while in WAYPOINTCONTROL mode. The death_spiral_radius represents the minimum circle radius that the AUV can currently achieve. The *execution* level code uses this formula as an alternative to standoff _distance to determine if it is possible for the AUV to achieve a waypoint. If the waypoint is closer to the AUV than this number, it is assumed that the AUV will get caught in a death spiral if it tries to acquire this point using WAYPOINTCONTROL. Figure 15 illustrates a scenario that the AUV will not obtain a given waypoint.

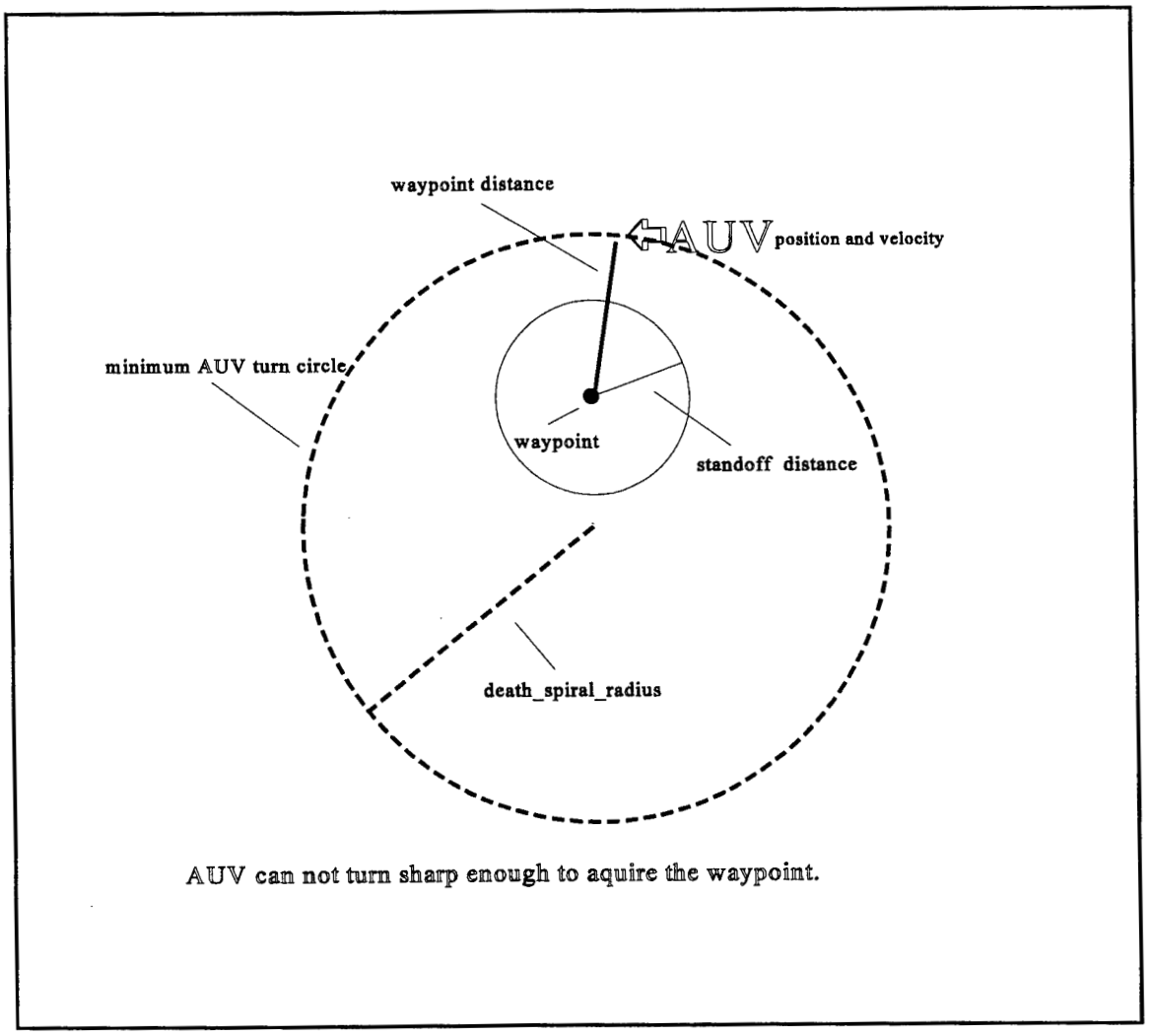

Figure 15 AUV can not reach waypoint

Therefore, in order to not get trapped in a death spiral, if the number that is produced from this formula is more than the distance to a waypoint, the waypoint is assumed to be achieved and the AUV continues on its mission. A further precaution is taken in addition to this check. Prior to making a turn, the AUV's heading is stored. If at any time during the turn the current heading varies more than 360 degrees from the stored heading, a death spiral is sensed, the waypoint is assumed to be achieved, and once again the AUV continues on its mission. The death spiral check is the reason that in the previous section, Hover Control Logic, a switch from HOVERCONTROL to

WAYPOINTCONTROL only occurs if the waypoint distance is more than standoff distance plus 15 feet.

3. Waypoint Control Logic

If in WAYPOINTCONTROL and the AUV is ordered to go less than 200 rpms, 200 rpms replaces that order. This ensures that the minimum rpms while in WAYPOINTCONTROL is clamped to 200 rpms. This minimum requirement is set so that the AUV will maintain adequate headway, i.e. be traveling fast enough to make effective use of the rudders and planes. Next a waypoint angle is calculated. This angle is converted to a turn direction that is needed to get from where the AUV is headed to where the AUV is going. Next a death_spiral_radius calculation is made. This calculation, as explained earlier in the Death Spiral subsection, is made to ensure that the AUV can attain the commanded waypoint. There are four tests for properly achieving the waypoint. Success for this cylinder test are identical to those explained previously in Figure 13, with the exception that orientation test is replaced by the death spiral checks. If these tests pass, the waypoint has been reached. Otherwise, the AUV continues to make its way to the waypoint uninterrupted by other orders. These are the four tests:

|depth_error| < standoff_distance
waypoint_distance < standoff_distance
waypoint_distance < death_spiral_radius
detect_death_spiral (FALSE) -- means that the AUV is not currently in a death spiral

When the waypoint has been reached, WAYPOINTCONTROL and FOLLOWWAYPOINT are both set back to FALSE and the AUV is then ready to receive its next order.

4. Rudders and Planes

Next delta_rudder, or the change in rudder angle needed to obtain the ordered course, is calculated. Delta_rudder is calculated by multiplying a heading constant to the difference between the ordered course and the current course. Factors of v and r are also added to this multiple to account for the rate of movement in the lateral direction and rate of turn of the AUV respectively. Then delta_rudder is modified as necessary to account

for the AUV rolling excessively in a turn. If too much roll is being encountered then delta_rudder is reduced to help keep the AUV stable through sharp turns.

A depth_error is next calculated. The depth error is simply the difference between the depth the AUV is at currently and the ordered depth it is trying to obtain. If the absolute value of depth_error is greater than 15 feet, it is clamped to a corresponding +/- 15 feet. This clamp is a simple control law constraint to keep the AUV from ascending or descending too steeply.

Next delta_planes is calculated by multiplying a z constant to the depth_error then adding both a factor to account for theta, the angle of pitch, and a factor to account for q, the rate of the pitch. Also taken into consideration is the rate at which the AUV is ascending or descending. Delta_planes (like delta_rudder) is further governed by how much roll the AUV is experiencing. If there is too much roll the angle of deflection for the planes is reduced to help the roll subside. Once calculated, both delta_rudder and delta_planes are clamped not to exceed +/-22.5 degrees. This is done to keep stable transitions through the water and to avoid excessive force on the AUV plane surfaces.

5. Thrusters

The lateral and vertical thruster voltages are set depending upon the mode of operation. If in ROTATECONTROL the AUV to spins about a vertical axis through the middle of the AUV in an open-loop mode. Therefore the lateral thrusters need a voltage applied that is proportional to steady-state open-loop spin rate (e.g. +5 degrees/second or -10 degrees/second). To arrive at the necessary voltage needed, the computer multiplies how fast the user wants to rotate the AUV by an empirically determined coefficient.

If in LATERALCONTROL the user is ordering the AUV to slide laterally either to the right or left. The user is given the opportunity to command steady-state lateral speed, (e.g.+0 .5 feet/second or -1.0 foot/second). This lateral command value is then multiplied by an empirically determined movement coefficient and an open-loop control voltage is applied.

If neither of these modes of operation are needed by the user, a default voltage is given to the lateral thrusters. This voltage is derived by using the psi_error and the

turning rate. These two readings combine to influence the thrusters and maintain a heading. Control equations are fully specified in (Brutzman 94).

The vertical thruster voltages derived from the depth_error and the rate that the AUV is traveling in the z direction. These voltages are next manipulated depending on which mode is in use. If THRUSTERCONTROL, HOVERCONTROL, ROTATECONTROL, or LATERALCONTROL are in use, the vertical voltage is set for both the bow and the stern vertical thrusters. The lateral thrusters depend on the particular mode of operation, (one of the preceding four). If in LATERALCONTROL the same voltage is applied to both bow and stern lateral thrusters at the same time and in the same direction. If in HOVERCONTROL, the voltage given to the bow and stern lateral thrusters is calculated using the heading error, the distance away from the point in a lateral direction, ocean currents, and the rate the AUV is traveling in the lateral direction. If in THRUSTERCONTROL or ROTATECONTROL, the lateral thrusters are operated similarly. A voltage is applied to the bow lateral thruster and an equal but opposite voltage is applied to the stern lateral thruster. This produces the spin about the z-axis. A larger voltage while in ROTATECONTROL allows the AUV spin at a higher rate. If none of the above four control mode toggles are set prior to getting to this portion of the control loop, a zero voltage is applied to all thrusters to ensure they are turned off.

6. Clamp and Send Commands

The thrusters are clamped to +/-24 volts or 3820 rpms no-load and the propellers are clamped to +/- 700 rpms no-load. These software governors ensure that the motors will not be overdriven during a mission. Next all calculated voltages and rpms are sent to their respective motors. Next the changes in the rudders and the planes that were calculated earlier are sent to the surfaces. The AUV makes the appropriate changes and continues on its way.

7. Record, Timestep, and Loop

At this point all of the commands that were passed to the hardware are recorded and sent to the *tactical* level for evaluation and storage. A time step increment is added to the current vehicle time. Next, a test is performed to see if the mission has been

completed. If the mission is not completed the control loop is repeated. If the mission is completed all motors are zeroed, end_test is set to TRUE and the `closed_loop_control_module` is exited.

E. SHUTTING DOWN

In shutting down the system, first there is a check made to see whether the AUV is in water. If so, the gyros are centered, caged and locked so that they will not be broken during handling of the AUV. Next, if the AUV is in water, the DiveTracker process that was spawned is terminated and the memory that was allocated to that process is reclaimed.

If the LOOPFOREVER mode was selected during virtual world operation, the "do-while" loop resets the clock, gets a new set of control constants, and starts again from the "do" prior to the control loop. If the user has also chosen the LOOPFILEBACKUP mode of operation, a counter is placed and incremented on each successful run. Also, orders and telemetry files that were made during the mission are backed-up and stored. This enables the user to have access to the telemetry files of the most current mission and the previous mission. This information is then used for a comparison and analysis. After all files are copied, control constants are then replaced, and the output files are ready to record the next mission.

If LOOPFOREVER is not TRUE communication sockets are closed, all device paths are closed, recorded data files are closed and stored in memory, and the program terminates.

F. SUMMARY

This chapter describes the successfully integrated virtual world/real world AUV software in detail, emphasizing vehicle control. The `closed_loop_control_module` is the heart of the *execution* level. It maintains stability of the vehicle while accomplishing the tasks that are sent down from the *tactical* level. Commands that are sent to the *execution* level are first parsed in the *execution* level parsing functions. At the end of each cycle the `closed_loop_control_module` checks the parsing functions to see if there is a

new order waiting. If there is no new order, the cycle begins again using the same flags that were sent by the previous command. These cycles continue until the ultimate goal is reached, at which point the *execution* records all stored data and terminates.

VI. EXPERIMENTAL RESULTS

A. INTRODUCTION

This chapter examines the test progression which validated the newly integrated software. It discusses how pre-mission testing and pseudo-mission testing were used in making the AUV ready to perform the Moss Landing mission. It also describes in detail how the *execution* level software fared during the final Moss Landing mission experiments.

B. PRE-MISSION TESTING AND EVALUATION

Pre-mission testing includes the ability to decouple the efforts of individual software developers (i.e. students and staff) so that they are not constrained by the results of other's progress prior to evaluating their own modules. The software structure used in the virtual environment did allow the programmers to do their work individually and independently. Over a period of five months, this structure permitted not only the *execution* level, but also the *tactical* level to be simultaneously developed. It also allowed for Sonar and a Navigation modules within the *tactical* level to be developed along with the *tactical* level. Each of these software pieces were able to be constructed and tested separately despite dependence on other students' work.

The *tactical* program is the lead process at the *tactical* level. After *tactical* is started, it spawns the Navigation, Sonar, and the *Execution* level as separate processes (Leonhardt 96). During development, each of these subordinate processes were "stubbed" (i.e. replaced by dummy calls) by the *tactical* level so that they did not interfere with the logic and construction of the *tactical* level. Once each process was tested satisfactorily, the stub for that process was removed and the new process was then allowed to interact with the *tactical* level. The key to this success was an approach that allowed for a natural progression when merging processes, and communication between the software developers prior to the completion of each package. The project team members spent a great deal of time discussing exactly what information was needed and in what form that

information was useful, as well as how often information was needed and how often a new result was produced from their calculations.

This methodology proved to be successful. The team members were free to write code at their own pace with predefined knowledge of exactly what was expected as a result of their efforts. This led to a rapid software development effort. Regularly throughout the development, the coders would converse with each other to ensure that none of the originally discussed ideas had changed, and also to ensure that no one was getting left behind. If any of the developers were baffled by a particular problem within their code, they were free to consult with any or all of the other students, staff and faculty to work out any difficulties. This communication during development proved to be as important as communication prior to development. Open dialog kept each of the writers up-to-date on the others' status, so that when the time came to merge separate modules each was ready. Because pre-mission testing successfully enabled rapid progress, the originally posed thesis question about combining the control software and the benefits of doing so was immediately answered. Although the work to this stage had nothing to do with the actual control of the vehicle, the leap in software development was enough to validate the effort attempting to combine the control structures. This distributed software development idea was a large asset to the AUV research group efforts but only a small part of the potential of the project. This effort to combine two *execution* level control codes unquestionably benefitted the group. Even if no other portion of this project were to be deemed successful, the AUV research group is now catapulted into a whole new arena based on its new-found ability to perform distributed software development.

C. PSEUDO-MISSION TESTING AND EVALUATION

Pseudo-mission testing was the next hurdle that needed to be addressed. Now that all of this code had been generated, did it work together and how could it be properly tested to confirm that is working together?

1. Versatility in Running the Software

Fundamental to the approach of this thesis is the idea that the *execution* level code and the *tactical* level code are separate, and each might run separately and independently on either on the SGI systems or the GESPAC OS-9 systems. This assumption gave birth to many testing ideas. Combinations of running this diverse architecture include running the *execution* level alone on either the actual AUV via a GESPAC 0S-9 system, or in the virtual world via an SGI system. In either case, the inputs to the *execution* level can come from a mission script file or from a keyboard input, and the outputs can be viewed either by watching the actual AUV perform movements or by viewing the virtual world AUV perform the movements. Another combination is to run the *execution* in either machine, but have the input come from either the SGI operating system's *tactical* level for the virtual world or the SUN operating system's *tactical* level from within the actual AUV. Again these outputs can be viewed as above on either the actual AUV or the virtual world. Prior to the start of this project it was known that the *execution* level was able to run in the virtual world successfully using the SGI system. Also the *execution* level had the capability of taking its input from the *tactical* level, a script file of a mission, or directly from the keyboard. What was not known was if it was possible to run a modified version of the *execution* level code on a GESPAC OS-9. When running on the SGI system, the *execution* level could be used as a driver to a dynamics model that could be observed on a viewer to the virtual world. The next test was to see if the *execution* level running on the GESPAC could be used as the driver to the virtual world. Many tries and modifications were needed due to a spare GESPAC system not being able to make reads on components that would normally be attached and are attached in the working AUV GESPAC. Eventually the GESPAC OS-9 system was able to drive the dynamics model and be viewed in the virtual world. The key to success in debugging a long and mysterious series of errors was the flexibility provided by distributed processes and socket communications. This versatility provided fast compile cycles and improved diagnostics in a variety of workstation environments. It is unlikely that success would have been achieved without such flexibility.

2. End-to-End Hardware Tests

End-to-end hardware testing was one of the capabilities that the new *execution* level software needed to conduct tests on the actual AUV. To this point, all that was known was that the *execution* level was written, able to compile, and ran in the virtual world as the driver to the dynamics model. The next step was to find out if the newly developed code could actually move the intended components of the actual AUV by the correct amounts and in the correct directions.

A copy of the *execution* level executable code was imported to the AUV research center where the actual AUV was stored. It was rapidly sent via Airlan (wireless bridge), but the process of recompiling remotely proved to be quite slow. The delay in compiling the C source code came from having to send the source code back to the campus from the remote test tank location, have it compiled on the (very slow) spare GESPAC system, then retrieve it from the spare GESPAC as an executable code. Next the code was downloaded into the AUV GESPAC while the AUV was on the test bench, not in the water. There were minor complications that involved locating the correct working directories within the actual AUV's GESPAC and where in the GESPAC to store data that would be retrieved while doing testing. Although these complications were minor, a great deal of time was used in making simple corrections to code then recompiling the code on the compiler using the slow machine (68020 processor) on the other side of campus. At this point it was learned that the compilation could be done locally in the AUV center via a DOS-based cross compiler that was in the research centers' PC. Files and scripts used to compile on the DOS-based compiler are found in (Marco 96). This was the answer that was needed for making the minor software corrections. Tests from this point went more rapidly. The compiler for the networked spare GESPAC had served its purpose in getting the project to this point, but was no longer needed as long as development took place next to the AUV. A series of minor corrections ensued and eventually all communication paths to hardware devices were established reliably.

Next a series of tests were performed without the AUV being submerged. These were open-loop order tests that included rotating the rudders, rotating the planes, rotating

the screws and rotating the thrusters. The open-loop order tests are defined as giving a specific angle for the rudders and planes (such as rudders +20 degrees or planes -10 degrees). For these tests to be successful, the rudders or planes needed to come to these predetermined angles and stop. For the screws and thrusters the criteria was that they had to spin in the correct direction. These tests required small adjustments such as changing the wiring polarity or the rotational directions in the rudder orders. Eventually, these tests were satisfactorily completed.

The next series of simple tests involved the AUV's ability to operate in a closed-loop manner. Once again these tests were conducted while the AUV was on the test bench and not in the water. This series of tests determined if the AUV was able to come to a particular heading (using as much rudder as necessary) then maintain that heading. As the AUV approached the desired heading the rudders smoothly decreased the amount of rudder angle needed to achieve the heading. To perform this test the AUV was placed on a test bench that had wheels. A heading order was given to the AUV and the rudders made the necessary deflections to bring the AUV to that heading. After a rudder deflection was made the test table was manually moved in the correct direction to respond to the vehicles rudder deflections as though the vehicle was actually in water and coming to the new heading. As the bench was rotated into the turn, the rudders were closely watched to ensure (as the vehicle came closer to its desired heading) that the amount of deflection in the rudder angle decreased. After achieving the desired heading the vehicle was rotated through the commanded heading to test if it might deflect the rudders correctly to regain the heading. A similar test was done by lifting the nose of the AUV to test if the planes might react to changes in depth. When these tests were satisfactorily completed, both vertical and horizontal thrusters were tested in a similar fashion. Similarly, while still on the bench, orders were given to move the AUV in a lateral direction to see if the lateral thrusters would spin faster or slower depending on the distance away from its desired location. The vertical thrusters were also observed in the same fashion depending on how far away the vehicle was from the desired depth. These tests were all completed satisfactorily.

3. Virtual to Actual Test Tank Mission

Although testing in the test tank might have proceeded more slowly, it was decided that the first tank test should be the mini-Moss Landing mission. The mini-Moss Landing mission was the Moss Landing mission, discussed in the next section, only scaled down to fit into a test tank. Transit delay times were shortened to accommodate the relative size of the test tank versus the size of the pier and slip at Moss Landing. To successfully carry out the mini-Moss Landing mission, nearly all of the *execution* level capabilities are exercised. However there were two particular areas in the *execution* level that could not be tested during a mini-Moss Landing mission. These two areas were the DiveTracker module and the positional update that is passed down from the *tactical* level. The DiveTracker could not be tested because the area in the test tank was too small. The beacon pings resonated the sides of the tank, making it impossible to get an accurate and consistent reading on where the AUV was positioned.

The *execution* level also could not get positional updates from the *tactical* level because these updates were based on GPS, DGPS, and DiveTracker readings (McClarin 96). The GPS and DGPS updates could not be retrieved because the test tank was located inside a building so there was no chance of obtaining satellite fixes. Despite these major portions of the code that were not tested, there was still a lot to gain by conducting the mini-Moss Landing mission.

The first step in attempting the mini-Moss Landing mission test was to plan out the exact course of the AUV. This was quickly agreed on. The group decided to make this test as close the Moss Landing mission as possible. The AUV was to be placed in the test tank at position 1 in Figure 16 and told its relative coordinates and given a start heading of 180 degrees. Standoff distance was set at 1 foot, propellers were turned off and thrusters were enabled. The first task was to have the AUV perform a GPS fix while on the surface. Because the AUV was in the test tank it was impossible for the AUV to obtain a fix, therefore it only went through the motions of getting a fix. The AUV was then instructed to submerge to a depth of 3 feet and point towards position 2. When stable, proceed to position 2 using 300 rpms. Once at position 2 the AUV was instructed

to hover at that point while changing its heading to face position 3. Once stable, again the AUV was ordered to 300 rpms and told to go to position number 3. At position number 3 the AUV aligned with position number 4 then told to acquire position number 4. At position number 4 the AUV would then normally conduct a sonar search and then get another GPS fix. Because the sonar search code was not yet completed, the AUV surfaced and went directly into a simulated GPS fix. Once stable from getting the GPS fix and submerging back to 3 feet, the AUV was ordered to position number 5. Normally the AUV would have to navigate around a target placed in the water between positions 4 and 5, but due size constraints of the test tank this replanning exercise was not done. At position number 5 the AUV was to conduct two separate 360 degree turns. While turning, the AUV would conduct a sonar search that would map out the test tank. The first turn was to be done by giving course changes in 90 degree increments. After stabilizing from that turn, the AUV conduct another turn by rotating at a rate of 10 degrees per second. After the second circle was completed and the AUV was again stable it would do another simulated GPS fix. Next the AUV would align with position 6 (same as 3) and transit to that point using 300 rpms for a propeller order. Positions 7 and 8 were to be acquired in the same manner of aligning with the position first, then propelling towards the position. Once at position 8 (same as position 1 where the mission started), the AUV would obtain the correct starting heading get yet another simulated GPS fix and terminate. This course was mapped out on paper as shown in Figure 16.

Next, the group was able to utilize one of the new features of the AUV. The course that was mapped out by hand was then made into a script file that was later used by the virtual world. A test tank was created to scale in the virtual environment, then the virtual AUV was adapted to run in that test tank. Next, the script that was written was used to drive the virtual AUV through the mini-Moss Landing mission. It was during this time that the group could appreciate the use of the virtual environment. It was quickly learned that the derived courses for the mini-Moss Landing mission were not sufficient. Watching the virtual AUV drive the mission in the virtual test tank showed that the courses and intermediate points needed be recalculated. At first the virtual AUV

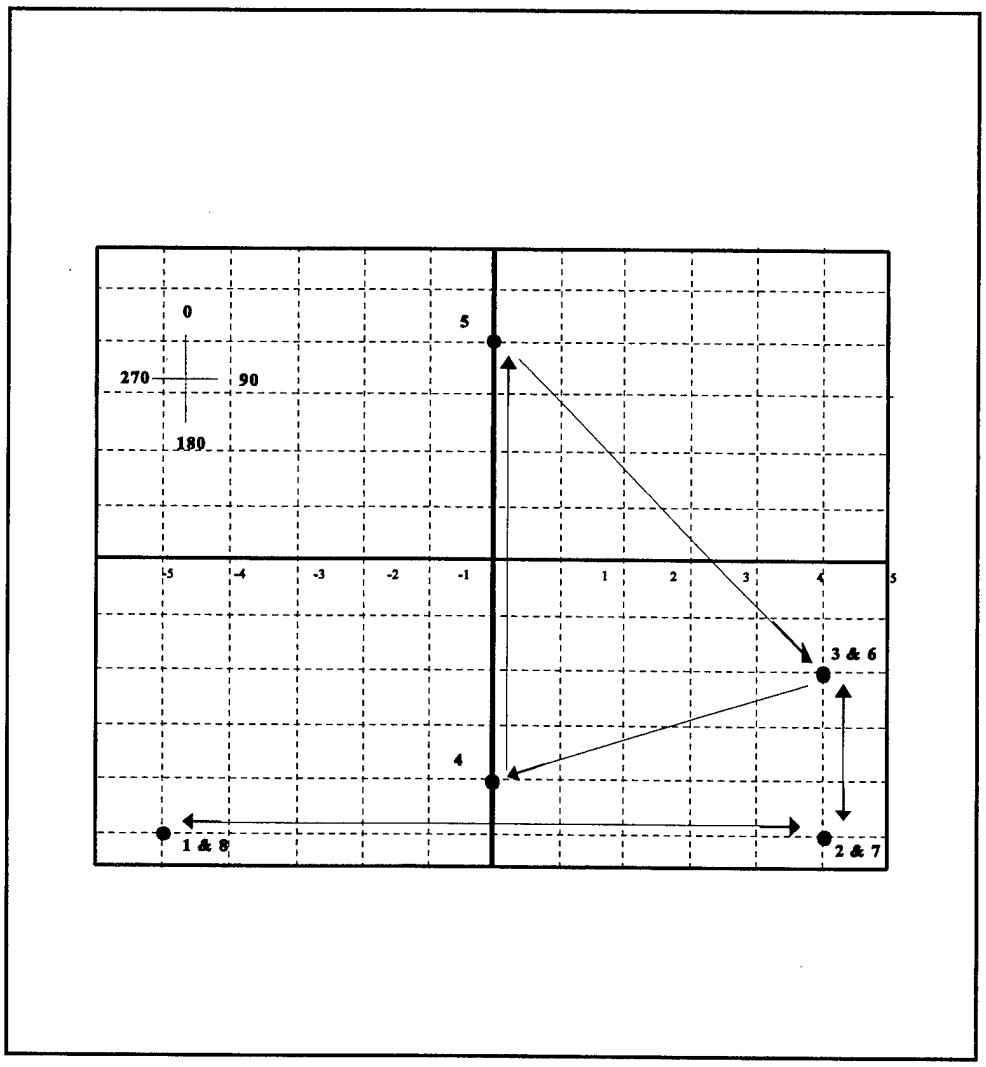

Figure 16 The mini-Moss Landing mission conducted in the test tank

sometimes attempted to run out of the test tank (typically during turns). Although this was not expected, the problem was easy to see and corrected in minutes using the virtual world. The same simple corrections would have taken much longer to diagnose (perhaps days) if the test were first performed in the actual test tank. Another benefit of having the virtual world was improved AUV safety. If the actual AUV had attempted to run this mission prior to the changes, the possibility of damage to the vehicle existed because it would have tried to turn improperly and hit the sides of the test tank.

After the corrections were made to the script file and the virtual test was

successful, the actual AUV was placed in the test tank and the script was used to run the mini-Moss Landing mission. Figure 17 is the script that was used to conduct the mini-Moss Landing mission. In Figure 17 "blank" lines and lines that begin with "#" are not used by the AUV. The results of this test showed weaknesses in three areas: the basic navigation system that resided in the *execution* level, the function that dictated the propellers revolutions, and the rate at which the batteries lost their charge. The navigational shortcomings were of the greatest importance. Although the AUV attempted to perform the mission, it had no accurate idea about where it was in the pool. The formulas derived to calculate AUV location (based on a known starting position and amount of propulsion used) were inaccurate and needed to be adjusted. Also contributing to this problem was the fact that the *execution* closed loop control module was not completing its cycles within the optimum 10 Hz rate. The latter problem was thought of as being the single most likely culprit and explored first.

The AUV was then placed back on the work bench and code was added to determine exactly how long each cycle was taking. It turned out that the rate of the closed loop was closer to 5 Hz than 10 Hz. Thus, the next portion of testing was streamlining and optimizing the *execution* level code to improve loop times. All unnecessary calculations were taken out and trigonometric function calculations were grouped together, calculated once and set aside in variables that were repeatedly used by all necessary functions requiring them. Prior to this there were many cases that variables were calculated repeatedly by different functions. This effort did not restore the 10 Hz rating but it did achieve 6.67 Hz. This 6.67 Hz was then evaluated by the group that designed the control filters, and was determined suboptimal but sufficient to run the *execution* level stablely. This *execution* level optimization is left as an avenue for future work. In order to consistently ensure hydrodynamic stability in all operational modes, the control loop needs to achieve the optimal 10 Hz frequency.

The AUV was placed back in the tank with better results, but still far from what was expected. The next changes were to import the actual navigational software and alter

```
# mission.script.mini-moss-landing

# ,,,,
#      16 December 95
# ,,,,

# moss landing mission scaled down to test tank


# dive tracker transducer locations

  DIVE-TRACKER1  -10  10 44
  DIVE-TRACKER2  -10 -10 44

  time 0

  no-pause

# test tank surface offset is
# 41 feet deep in virtual world

  posture -5 -5 41 0 0 180

  course 180

# standoff distance for waypoint-hoverpoint success

  standoff 1

# propellers off
  rpm   0

  thrusters-on

# start the clock
  step

# start point is
# hoverpoint  0
  hover   -5 -5 41 180

  GPS-FIX
# re-stabilize in case GPS-FIX recovery is poor
  hover   -5 -5 41 180

# using thrusters
# go to mission depth
```

Figure 17a Mini-Moss Landing script file

```
 hover   -5 -5 44 180

 rpm 300

 waypoint-follow

# waypoint  1
# eliminated due to proximity in test tank
# waypoint -5 -5 44

# ,,,,
# test tank is too small
# for effective waypoint testing
# ,,,,
# therefore add a hoverpoint
# to point at follow-on waypoints
# ,,,,

# line up for next waypoint
 hover  -5 -5 44 90
 rpm 300  restore speed

# waypoint  2
 waypoint -5  5 44

# line up for next waypoint
 hover   -5  5 44 0
 rpm 300  restore speed

# waypoint  3
waypoint -2  5 44

# line up for next waypoint
 hover   -2  5 44 270

# search phase

# ,,,,
# move to first search point

# hoverpoint 4
 hover    -4  0 44  0
```

Figure 17b Mini-Moss Landing script file (continued)

```
# rotate sonar
# not yet implemented
 wait  5

 GPS-FIX

# re-stabilize in case GPS-FIX recovery is poor
# hoverpoint 4
 hover   -4  0 44  0

# move to second search point

# hoverpoint 5
 hover   +4  0 44  0

# already aligned with north

 course  90

 course  180

 course  270

 course   0

# fully stabilize prior to sonar search
 wait   10

# rotate full AUV search
 rpm     0
 rotate  10 degrees per second
 wait   36

# stop rotating
 rotate  0
 course  0

# regain hoverpoint 5
 rpm 300  restore speed
 hover   +4  0 44  0

 GPS-FIX

# re-stabilize in case GPS-FIX recovery is poor
# still at hoverpoint 5
 hover   +4  0 44  0
```

Figure 17c Mini-Moss Landing script file (continued)

```
# line up for next waypoint
 hover    +4  0 44  150

# return transit phase

 rpm 300  restore speed

 waypoint-follow

# waypoint  3
 waypoint -2  5 44

# line up for next waypoint
 hover   -2  5 44 180
 rpm 300  restore speed

# waypoint  2
 waypoint -5  5 44

# line up for next waypoint
 hover   -5  5 44  270
 rpm 300  restore speed

# waypoint  1
 waypoint -5 -5 44

course 180

# finish point is the same as the
# start point
 hover   -5 -5 44 180

# surface
 hover   -5 -5 41 180

 GPS-FIX

# re-stabilize in case GPS-FIX recovery is poor
 hover   -5 -5 41 180

# all stop
 rpm      0
 thrusters-off
# sonar off
# ,,,,
# mission complete
# shutdown
# quit
```

Figure 17d Mini-Moss Landing script file end

the propeller rpms to account for interference and drag (shaft friction). Until this point the propeller shafts operated on a linear basis. One rpm command would send one voltage to the shafts. If the shaft was experiencing drag, the voltage sent would not be enough move the propeller as predicted. Changes were made in the software to allow for interference detection by the propeller shafts. A feedback loop was installed that monitored shaft rpms and if they were not rotating at the proper speed more voltage would be added until the unexpected drag was overcome. When placed back into the test tank with these changes installed, the AUV's performance improved. Next, the constants from the formulas that influenced the voltages sent to the propellers were altered and the AUV got to a point in tank testing that was acceptable. It was thought that during an actual test in Moss Landing these problems would not be visible because the *tactical* level would be sending down updated positions for the *execution* level.

The last important finding that was revealed during tank testing was the batteries' short lifetime. It was found that while operating in the water the AUV with a full charge had at least one good hour and possibly a half of an hour additional endurance. If problems occurred after the first hour of testing they were normally due to the batteries dying prematurely. It was thought prior to the test tank missions that three hours could be expected out of the batteries. This final problem had a simple solution of placing more batteries in parallel to gain a longer life time. Although this was a simple answer, it required a great deal of time to reconfigure the hardware inside of the AUV to make room for additional batteries. It was then decided that added lifetime was more of a luxury than a necessity at this point and future testing would be restricted to the shortened life time of the batteries that were in place. In time all corrections were made, constants were "tweaked" to the satisfaction of the AUV group, and the AUV ran successful mini-Moss Landing missions both in the virtual world and the test tank. The last test that was performed while still in the test tank was to run the mini-Moss Landing mission without the Ethernet connection. This became known as "untethered" mode. The programs were loaded into the SUN computer onboard for the *tactical* level and into the GESPAC

computer onboard for the *execution* level. A delay was set in the *execution* level prior to starting the mission that allowed for the safe disconnection of the Ethernet. Eventually this test (with adjusted operator expectations) ran successfully numerous times.

The concentration was on the fact that the AUV attempted to navigate to individual waypoints. It was known that the *execution* level navigation system was not developed enough to place the AUV in the tank with precision. Therefore, as long as the correct movements of both the thrusters and the propellers were viewed, testing personnel were allowed to physically assist the AUV in obtaining waypoints during the mission. However, if the AUV did not produce the correct movement (disregarding the magnitude of the movement), or if at any time the correct heading was not obtained, the test was deemed a failure. Performance was judged adequate during these tests. Calibrating the dead reckoning performance became a major goal for in-water testing.

4. The Moss Landing Mission

The next test was the one that all other tests were leading up to. This test would prove exactly how successful the project was. Here again, as in the test tank, the entire mission was tried instead of doing it in parts. This test initially failed for many reasons both in the *execution* level and in the *tactical* level. The *tactical* level results are discussed in (Leonhardt 96). The concentration of this thesis is strictly in the *execution* level.

The test started the AUV at position 1 in Figure 18. At position 1 the AUV was given a posture command that told it where it was in the new coordinate system and what direction it was heading. Next the positions of the baseline DiveTracker beacons were read. The standoff_distance for waypoint achievement was set to 1 foot and propellers and thrusters were turned off. The next task given to the AUV was to wave its rudders. This was done solely for the purpose of giving the test personnel a visual conformation that the AUV is still active. Thrusters are next enabled and the AUV is told to hover at the starting position on the surface and keep the prepositioned heading. Next the AUV is commanded to get a GPS fix and remain hovering in the same location. Once the GPS fix is obtained, the AUV is to submerge to a depth of 3 feet, hover at that the new depth

and point at the next waypoint. When the AUV is stable in this position, it is told to go to position number 2 in Figure 18 using 300 rpms for propeller speed. When position

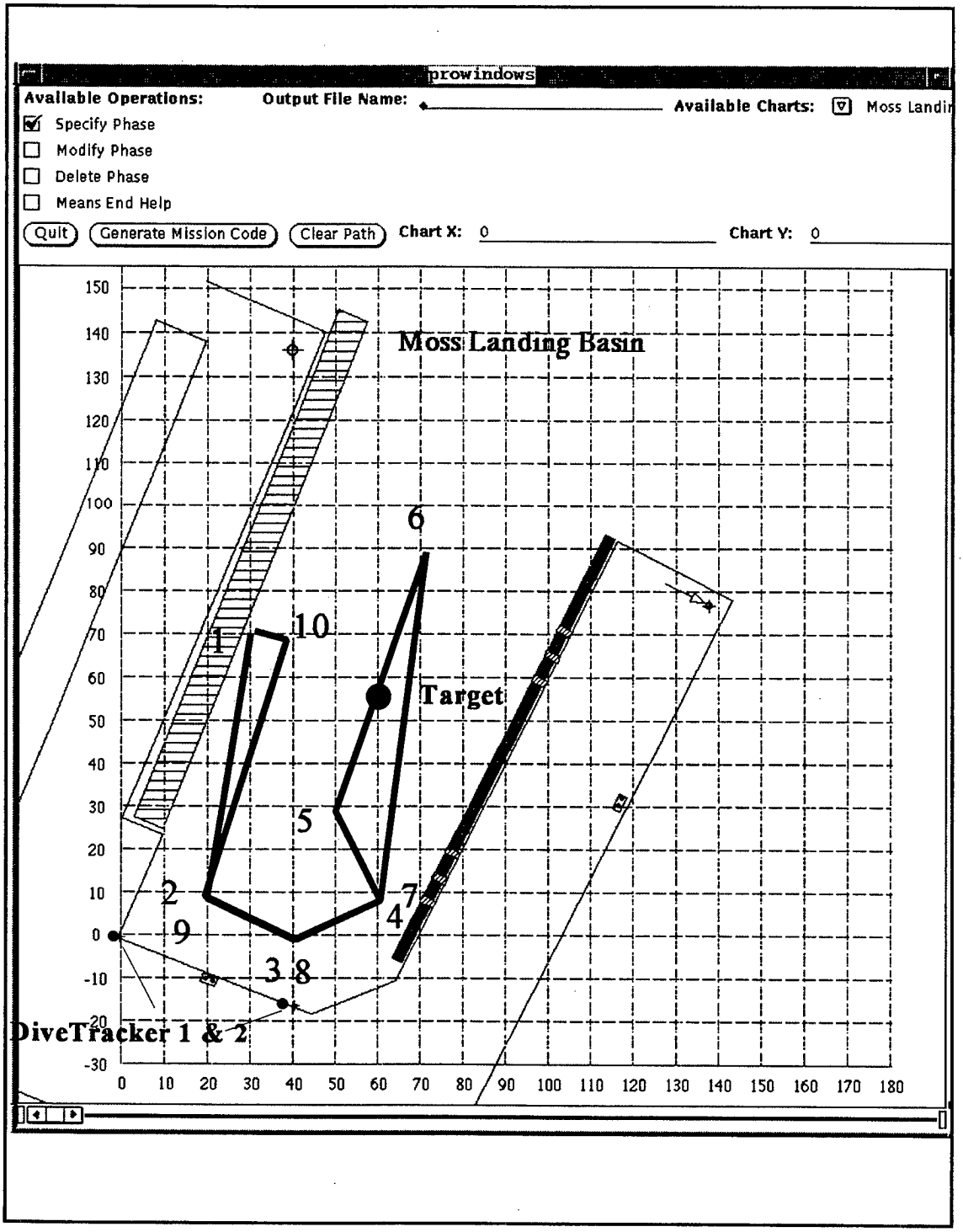

Figure 18 Moss Landing Mission

number 2 is acquired the AUV is to hover at that point while changing its heading to point to position number 3. When the vehicle has completed this maneuver, 300 rpms are again ordered to the propellers to in order to go to position number 4. Again the AUV is aligned to make its way to position number 5. At position number 5, the AUV is commanded to conduct a sonar search by rotating the sonar 360 degrees and holding the AUV steady facing North. Due to problems in that portion of code, a different search was conducted. Instead, the AUV was given course changes at 45 degree increments until a full circle was completed. During this search the sonar was held facing the front of the AUV. This task was done to map out the basin and find the target. After this search the AUV was ordered to surface and take another GPS fix (to be used for comparison after the mission). Once resubmerged and stable, the AUV was ordered to position number 6. The reason for this transit was to test the AUV's ability to replan its path. During the sonar search, the AUV was to find a target that was directly in the path of the AUV trying to acquire its next waypoint. The AUV would then have to plan a new route to the next position in order to avoid the target. Once at position number 6, the AUV would conduct another sonar search then take another GPS fix (to be compared with the last search after the mission). Next, the AUV was to align its heading with position number 7 (also #4) then transit to that position. It was to then repeat aligning and transiting to positions 8, 9, and 10. Position number 10 was adjusted away from the pier in order to account for possible dead reckoning errors. Once at position number 10, the AUV was then allowed to slowly make its way to the pier to position 1 where it had started the mission, and then terminate. Figure 18 shows an illustration of the mission that was set out to be accomplished and Figure 19 is a copy of the script file that the AUV used to try to accomplish the mission. In Figure 19 "blank" lines and lines that begin with "#" are not used by the AUV.

Many difficulties were encountered at Moss Landing. The first problem was due to ballast. The AUV was too light and the thrusters were not strong enough to overcome the buoyant effects of the salt water as opposed to the fresh water that was in the tank.

```
# mission.script.moss-landing
# ,,,,
#      27 January 96
# ,,,,
# moss landing mission
# demonstrate transit, minefield map and return
# launch position
# ,,,,
#        x  y  z  roll  pitch  yaw
# ,,,,

 posture  70  30  1   0    0   202
 time 0
 no-pause

# dive tracker transducer locations

 DIVE-TRACKER1   0  0  2
 DIVE-TRACKER2  -17  40  2

# set and drift current test
# oceancurrent 0.0 0.0 0

 GYRO-ERROR  0
 course   202

# standoff distance for waypoint-hoverpoint success

 standoff   1

# propellers off

 rpm       0
 thrusters-off

# start the clock
 step
# waggle rudders

 rudder  40
 wait    1
 rudder -40
 wait    1
 rudder   0
 wait    1
 thrusters-on
# start point is
# hoverpoint  0
```

Figure 19a Moss Landing mission script file

```
hover   70  30  1  202

 GPS-FIX

 # restabilize in case GPS-FIX recovery is poor
 hover   70  30  1  202

# using thrusters
# go to mission depth
 depth 3

# ,,,,
# for reliability
# add a hoverpoint
# to point at follow-on waypoints
# ,,,,
# line up for next waypoint
 hover   70  30  3  190
 rpm 300
 waypoint-follow

# waypoint  1
 waypoint 10  20  3

# line up for next waypoint
 hover   10  20  3  115
 rpm 300  restore speed

# waypoint  2
 waypoint  0  40  3

# line up for next waypoint
 hover    0  40  3  70
 rpm 300  restore speed

# waypoint  3
 waypoint 10  60  3

# line up for next waypoint
 hover   10  60  3  330

# search phase
# ,,,,
# move to first search point
# hoverpoint 4
 hover    30  50  3  0
 sonar_search
```

Figure 19b Moss Landing mission script file (continued)

```
# already aligned with north
# rotate while hovering

 course  45
 course  90
 course  135
 course  180
 course  225
 course  270
 course  315
 course   0

# fully stabilize prior to sonar search
 wait    10

# rotate sonar
# not yet implemented
 wait  5

 GPS-FIX

# restabilize in case GPS-FIX recovery is poor
# hoverpoint 4
 hover    30  50  3   0

# move to second search point

# hoverpoint 5
 hover    90  70  3   0
 rotate_search
 hover    90  70  3   0

# rotate full AUV search
 rpm      0
 rotate   10 degrees per second
 wait    36

# stop rotating
 rotate   0
 course   0

# regain hoverpoint 5
 rpm 300  restore speed
 hover    90  70  3   0
 GPS-FIX

# restabilize in case GPS-FIX recovery is poor
# still at hoverpoint 5
```

Figure 19c Moss Landing mission script file (continued)

```
 hover    90 70 3   0

# line up for next waypoint
 hover    90 70 3 190

# return transit phase

 rpm 300  restore speed
 waypoint-follow

# waypoint 3
 waypoint 10 60 3

# line up for next waypoint
 hover   10 60 3 250
 rpm 300  restore speed

# waypoint 2
 waypoint 0 40 3

# line up for next waypoint
 hover   0 40 3 295
 rpm 300  restore speed

# waypoint 1
 waypoint 10 20 3

# line up for next waypoint
 hover   10 20 3  10
 rpm 300  restore speed

# safety standoff prior to pier
 hover   67 40 3 202

# finish point is the same as the
# start point
# no waypoint due proximity to pier
 hover   70 30 3 202

# surface
 depth 1
 GPS-FIX

# restabilize in case GPS-FIX recovery is poor
 hover   70 30 1 202

# all stop
 rpm     0
```

Figure 19d Moss Landing mission script file (continued)

```
# waggle rudders

 rudder 40
 wait   1
 rudder -40
 wait   1
 rudder  0
 wait   1

# sonar off
# ,,,,
# mission complete

 shutdown

 quit
```

Figure 19e Moss Landing mission script file end

The vertical thrusters turned on when they were required to, but the program timed out when the AUV did not reach its operating depth of three feet. This was an easy fix but as testing continued from morning to afternoon and day to day, water temperature and salinity changed significantly and the ballast had to be continually altered. Therefore, before each mission in the morning or in the afternoon, weight was either added or taken away so that the AUV might maintain a neutrally buoyant status.

The next problem was with the DiveTracker. Although the *execution* level did no calculations with the data received by the DiveTracker, it still was responsible for spawning that process and receiving all data from the base stations. This turned out to be a three-part problem: intermittent signals received from the baseline, incorrect data communication between software levels, and ambiguous symmetry of the DiveTracker system. The first part of this problem was quickly realized. Often, at the beginning of a mission the DiveTracker unit would not initialize and ping the mobile unit that was located in the AUV. This caused a problem because if the *tactical* level does not receive a DiveTracker update within an arbitrary period of time it shuts down the system

(Leonhardt 96). After many tests that isolated this problem to the DiveTracker baseline, it was found that there was a possible bug in the software that the AUV research group received when the DiveTracker unit was purchased. The manufacturer was called in to assist and found that our group had an outdated version of the software that was likely producing the problem. An updated version of this code was installed and the first part of the problem was corrected.

The second problem, communications between the *tactical* and *execution* level, was not as easy to find. After several tests it was determined that in some cases there was not enough information getting passed and in other situations too much information was getting passed. Specifically, initially there was not enough data getting passed to the *tactical* level where the DiveTracker readings were filtered along with GPS readings from satellites (McClarin 96). The satellite readings gave a large margin of error (+/- 300 feet) while the AUV was on the surface. When the AUV submerged it began taking DiveTracker readings and no longer had the GPS readings available. These readings were more precise (+/- 6 inches) but they were filtered with the GPS readings. The problem at the start of each mission was that there were not enough DiveTracker readings filtered to get an accurate position of where the AUV actually was located. Therefore the AUV was starting in a place that it thought to be correct but was actually not close to where it should have been. When it started to make its transit, the positional updates were so far off that there was too much conflict in the control software. This disparity caused the AUV to become unstable. This problem was fixed by having the first order given to be a submerge followed by a 30 second wait. During this 30 seconds, the filters were able to get consistent inputs on where the AUV was or filter out the positions that were fixed by the GPS system. Diagnosis of these problems was hampered by lack of virtual world models for DGPS and DiveTracker.

The third part of DiveTracker problems was related to baseline symmetry and was found late in testing. On the final test the AUV was placed directly between the two DiveTracker baseline beacons. If placed away from the baseline and given a starting position of the AUV the DiveTracker system can distinguish which side of the baseline

the vehicle is on than track the mobile unit from that point. However, the DiveTracker system has a symmetric ambiguity as shown in Figure 20. The system only gives distances to the mobile unit from each of the home beacons. The actual position of the mobile unit can be in either one of two places that are equal distance from both baseline beacons. An example of this is; if there is a line segment drawn on a piece of paper and an "X" is placed on one side of the line segment, there is a distance that is unique from either of the segment endpoints to that "X". If that piece of paper is then folded along the line segment and the "X" is then traced on to the other side of the line segment, the traced "X" will also be the same distance from each of the line segments endpoints as the original "X." Therefore it cannot with certainty be determined which "X" is the correct one unless this information is known prior to writing the first "X".

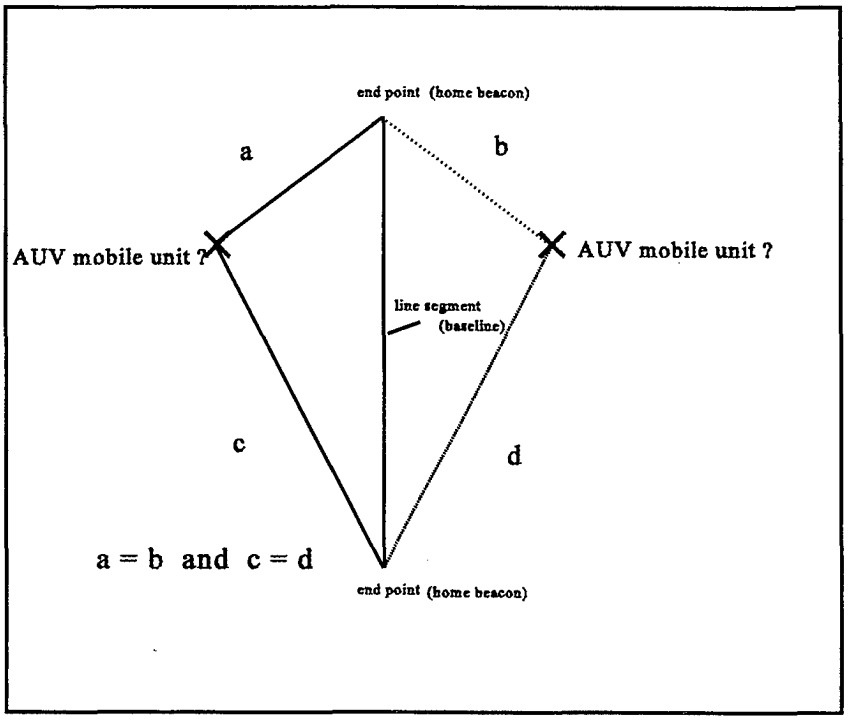

Figure 20 Reflexive 3-beacon system means AUV position relative to baseline is ambiguous.

The last day, when the vehicle was placed directly on the baseline and then moved, the software randomly picked which side the vehicle was on. In this particular case the system chose the wrong side of the base line and caused the AUV to become unstable because no matter how hard it tried to get to its destination, DiveTracker positions indicated that it was going away from the destination.

Another test that was done while at Moss Landing was to try to calibrate the speed sensor. While the AUV is traveling at a speed faster then .25 ft/sec, it uses the speed sensor to calculate its speed through the water. A script was made to allow for a straight speed run along the pier. The run was made and data collected it was then learned that while operating below .6 ft/sec a multiple of 4 was needed in the formula that calculates the speed through the water while above .6 ft/sec only a factor of two was needed. This difference in multiples is not precise due to insufficient test runs that were conducted in original speed wheel tests and this is another avenue for further research. Although these multiples will not produce exact measurements, they will allow for a closer approximation of where the AUV is while using the *execution* position calculations. Thus a major objective of these tests, obtaining an accurate dead reckoning model, remains incomplete.

One other test was conducted. The AUV was sent to attempt the mission untethered. This test failed due to lost communications between the *tactical* and the *execution* level. Specifically, the AUV started on its mission but then the *execution* level locked up. The last commands that were sent to the hardware remained active and the AUV needed to be physically retrieved and manually shutdown. The reason for this problem was not resolved, but the problem was likely due either to the batteries going dead or too many print statements with nowhere to be printed. The screen print problem was corrected by adding an "untethered" switch and in later tests this error was not encountered.

D. SUMMARY

The approach of pre-mission and pseudo-mission testing was used extensively in validating the newly developed software. Pre-mission testing provided numerous instrumental successes and validated a key goal of this project. The ability to distribute the software design proved to be a must in rapid development. Pseudo-mission testing, especially the ability to end-to-end test, also validates this thesis' worthiness.

Although multiple problems were encountered when testing in Moss Landing, a majority of the problems were solved on location. However, time constraints and other commitments made it impossible to continue Moss Landing testing. Nevertheless, prior to leaving Moss Landing, enough information was gathered to satisfy the project team that the AUV *Phoenix* will run and complete a full mission successfully. It appears that all systems worked most or part of the time, but reliability was never good enough that they all ran successfully at the same time. Successful completion of the end-to-end Moss Landing mission remains an unfulfilled experimental goal. Less complex untethered missions completed recently in the NPS test tank (Davis 96) have produced further evidence that this goal is attainable.

VII. CONCLUSIONS AND FUTURE WORK

A. INTRODUCTION

In this final chapter, thesis goals and results are evaluated. This is done by answering problem statement questions from Chapter III. As in all complex research projects, the work is never completed since new answers provoke newer questions. Therefore, this chapter will also discuss some short-term and long-term research goals that can further benefit the Phoenix project.

B. RESEARCH CONCLUSIONS

The questions posed in Chapter III were discussed at great length prior to taking on this project. These questions are now revisited and answered based on the experimental results achieved.

1. Will the Virtual World be Helpful?

At the root of this project is the question, "Will integrating the software control system of a virtual world AUV and the software control system for an actual AUV be beneficial to the ongoing AUV research being done at NPS?" This was the most important question from Chapter III. It is quite easily answered affirmatively. By integrating the software, many new capabilities were made possible. The obvious is true; one control software package can run either the virtual world or the actual AUV. However, there are other benefits that also come when combining these two control codes. This has allowed another dimension to the research being done at NPS. Prior to this project there was no distributed software development, and no way to visualize an entire mission prior to it to being tested. Both of these enhancements were major side effects of combining the code. They will both play a significant role in how future research will be conducted for the AUV *Phoenix*. This project has offered a concrete example of the value of simulation-based design for uncovering the true potential of Autonomous Underwater Vehicles.

2. Which Version (Actual or Virtual) Will be the Basis for the Combined Robot Software?

"Which code should be imported into the other code in order to complete the project?" There is a trade off for the development of the source code. If the actual control code was used as the base, precise stability might be inherited but the structure would need to be refined. Alternatively, if the virtual control code was used as the base, better structure was in place but physical stability would need to be refined. There is not a clear-cut answer to which might best be used as a base, since either final result ultimately ought to work.

The initial thought was to take the actual code that had already been tested in the water for stability, and expand it into a more robust form that could accommodate the many tasks that the virtual world could perform. However, after further review, it was decided to use the virtual code as the basis for the combined control code and import the actual functions into the virtual structure. The main reason for this decision was that the virtual world enabled much faster compilation and continuous feedback about the success of code changes. There were two other motivating reasons for this decision. The research group students working in the *tactical* level were also using the virtual AUV's structure, so when the time came to merge the two levels it would be an easier process if both structures were similar. Additionally, if the actual code was used as the basis, major rewrites would be necessary to comply with the formally defined hydrodynamics model of the virtual world. After weighing these many factors, it was decided that using the virtual control code as the basis was a more reliable and productive approach. This decision proved to be sound.

3. Message Passing to Tactical Level

"How will the *tactical* and *execution* levels communicate and what information will be passed?" It was decided that a single non-blocking read/write socket might efficiently maintain the connection between the *tactical* and *execution* levels.

As an input to the *execution* level, a single order can arrive on each cycle. In addition to orders were reports, such as a positional update, since one possible message is

a fix position. If the *execution* level receives a fix order, it updates the state vector instead of attempting to perform a new task. As an output of the *execution* level, a state vector is produced and sent to the *tactical* level on every cycle. State vector details are discussed in Chapter V. The idea behind the state vector communications to the *tactical* level is to allow any process access to current vehicle state. The critical calculations performed in the *execution* level include all calculations necessary to maintain stability, self preservation, and rudimentary navigational position calculations. All other calculations such as precise navigation, sonar classification, and path planning are done in the *tactical* level using the state vector values. Results from the *tactical* level are passed down to the *execution* level as orders. In summary, demonstrated results answering this question were affirmative. The proposed communication scheme works as anticipated.

4. What About 10 Hz Stability?

"Will using the robust *execution* structure used in the virtual world code allow the vehicle to maintain a 10 Hz *execution* level update rate?" It was found that the answer to this question is no. However, it was found that the AUV did operate at 6.66 Hz successfully. This new closed loop rate did work, but did not allow the AUV to operate at its full potential. The importance of this question is such that it is discussed in both the short and long-term goals. The research group ought not to be satisfied with the closed-loop rate that the AUV is maintaining. Correcting this problem is a major concern which could be resolved either by an *execution* level processor upgrade, or software improvements, or perhaps both.

C. RECOMMENDATIONS FOR SHORT-TERM FUTURE WORK

Some short-term projects that can be accomplished in a reasonable amount of time are: improve the post-mission playback, benchmark and optimize the *execution* level, and verify the effectiveness of the *execution* level sonar. Post-mission playback was first discussed in the AUV Integrated Simulator (Brutzman 92). This capability has been added, enabling missions that are conducted in the actual world to be played back in the virtual world for analysis. The output file containing the vehicles telemetry is used for

this operation and, because each state vector is time stamped, results can be played back in real time, depicting exactly what the AUV is doing. Making this feature easier to use and simplifying mission result archive procedures still needs to be done.

Benchmarking and optimizing the *execution* level is another short-term project. There are many instances where one variable is updated more than once during each cycle. Eliminating these duplications would enable the control cycle to go faster. Benchmarking ensures that optimization efforts are focused on the portions of the code that actually take the longest time to run. Also, a rewrite in the *execution* level is needed for clarity. Because of the complexity of the code and the time given for the project, the primary focus was in getting the code to run. Individual modules are understandable but the sheer size of the project makes the combined operation appear complex. One of the easiest solutions to making the code more readable is grouping of the toggles so that one action or task is only in one group, and that task is manipulated only once.

Adding an *execution* level sonar is another short-term project that was recently completed (Davis 96). The tests in this thesis were conducted using the *tactical* level sonar. This was necessary in order to rapidly complete the project and attempt an in-water mission. Now that a mission is known to work, the *execution* level must have its own sonar for obstacle detection. Improved obstacle detection capability in the *execution* level will enable the AUV to conduct self-preservation tasks more quickly. The code for the *execution* level software has been recently installed, but further testing is needed to verify its effectiveness.

D. RECOMMENDATIONS FOR LONG-TERM FUTURE WORK

The long-term goals that can be accomplished by future thesis students are to upgrade the GESPAC computer and reconstruct the internals of the AUV. There was a problem with maintaining a 10 Hz rate through the *execution* level control cycle discussed in Chapter VI. One of the ways to correct this is to upgrade the operating system. The GESPAC OS-9 was the original operating system used for this project in controlling the AUV. However, over the course of time new faster operating systems

have been developed and are now on the market. However, because of the time that would be required to completely install a new operating system into the AUV, a new system has not yet been incorporated. A valuable thesis might be to evaluate several alternative operating systems as to how they would best serve the AUV, and then actually install the best operating system into the AUV. The new operating system needs not only to regain the lost 10 Hz update rate, but also should increase the number of hardware controllers that might be used by the operating system. This in turn might allow the AUV to accomplish new tasks in an effort to adapt to changing scenarios. Certainly, versatility is key to the AUV being involved in many different assignments. One possible solution might be a Pentium laptop running a free Unix operating system, free g++ (C++) compiler, Ethernet connectivity and a compatible data acquisition interface.

Another long-term goal for future work is to rewire the existing vehicle and reposition its internals. If it is decided by the research group to keep the current operating system, rewiring is a must. The majority of time spent at Moss Landing related to fixing minor problems that took a great deal of time to physically access. Specifically, it would be possible to insert a central fuse box that can be readily accessed. This would cut delays immensely. Also, with that capability, sonars and gyros could be easily turned on and off. This would reduce the hazards of accidentally wearing out these components. Along with installing a fuse box, the internals of the AUV need to be cleaned up and arranged neatly. Because of the massive amount of new components, and the little amount of time to test, the components were added where they would fit. A rearrangement of the internal components could place components in logical positions that are easier to access. Again, this should cut down on delays during actual tests when locating defective components and accessing them for replacement.

E. SUMMARY

The ability to incorporate the virtual world with the actual world is now used as a foundation for all work done on the AUV. The progress of work has been greatly accelerated due to this capability. The tradeoff for long-term projects is time. Is the

group ready to take the time necessary now to build a better foundation for the project, or will taking that time stifle creative new ideas that are ready to be tested on the AUV now? Perhaps in the near future there will be a lull in new ideas that will prove to be the opportune time to rebuild the foundation. Based on experience gained in the conduct of this thesis, prompt action on future work is recommended in order to best enable reliable rapid progress by future students.

APPENDIX A - `execution.c` SOURCE CODE

```
/*****************************************************************************/
/*
 Program:          execution.c

 Description:      AUV execution level program

 Authors:          Don Brutzman, Mike Burns, Duane Davis,
                   Dave Marco & Walt Landaker

 Revised:          7 February 96

 System:           AUV Gespac 68020/68030, OS-9 version 2.4
 Compiler:         Gespac cc Kernighan & Richie (K&R) C (NOT ANSI C!)

 Compilation:      ftp>     put execution.c
                   auvsim1> chd execution
     [68020]       auvsim1> make -k2f execution
     [68030]       auvsim1> make      execution

     [Irix ]       ~brutzman/execution>>  make execution

 Execution:
     [Irix ]       ~brutzman/execution>> cd execution
                   ~brutzman/execution>>    execution  remote dynamics-hostname

                   where dynamics-hostname is the IP name of the host running
                   the dynamics (virtual world) program

 Plotting:         see gnuplot scripts 'auv_plot.gnu' and 'auv_plot_1_second.gnu'
                   ~brutzman/execution>> gnuplot  auv_plot.gnu

 Debugging:        ~brutzman/execution>> lint -lm execution.c

                   lint -lm -Iglobals.h -Idefines.h globals.c parse_functions.c \
                         execution.c

                   ~brutzman/execution>>  make warnings

 Description:      closed loop for operation during vehicle in-water
                              missions as well as in virtual world


 Active changes:   Don Brutzman    working lab/virtual world networked version
                                 & tactical level interface

                   Dave Marco      working vehicle code
                                 & interfacing physical devices

                   Mike Burns      Merging the two code sets


 Future work:      Update digital <==> analog access for new vehicle hardware

                   Retest code after vehicle repaired

                   Sonar/altimeter integration code reintegrated/retested

                   Audios seem to be generated differently by OS-9

                   standardize parsing of command line and script commands

                   finish sliding mode control

                   change serial/parallel comms to sockets once
                       tactical level gets an Ethernet card

                   DiveTracker and GPS code will be in tactical level
```

```
Testing interprocessor connections:

  parallel port   /P          OS-9 auvsim1> mfi_a3

                  LPT1:       DOS  auvsim2> portfix
                                          > print filename.txt

    serial port   (/T1) /TT   OS-9 auvsim1> wr2t1          then write text
                              OS-9 auvsim1> rdt1a          then read  text
                              DOS  auvsim2> C:\COMM\PROCOMM
                                                 then <alto> for chat mode
                                               <altF10> help, <altX> exit
Interfaces:

   Telemetry sent         via serial   port /tt [== /t1 at high baud rate]
   Telemetry received     via parallel port /P

   Telemetry is optionally passed to/from tactical level running on 80386

   Reads files:           mission.init  [mission initialization data file   ]
   Writes files:          output.data   [vehicle telemetry state vector data]
                          output.auv    [tactical order/executive report log]

   Sonar commands/replies via device port /t3

   Note that %F double formats are used instead of %lf on scanf() and sscanf()
             calls for OS-9 compatibility.  SGI C compiler does not complain.

*/

/*****************************************************************************/

#include "globals.h"
#include "statevector.h"
#include "defines.h"

/*****************************************************************************/
/* function prototypes                                                       */

/* is there some way to put parameter specifications in the prototypes??     */
/*    only if we buy the ANSI C compiler from Microware (or shift to VxWorks) */

/* thus following prototypes are missing parameters  :(                      */

   void          closed_loop_control_module          ();

   double        read_depth                          ();
   double        read_psi                            ();
   double        read_roll_rate_gyro                 ();
   double        read_pitch_rate_gyro                ();
   double        read_yaw_rate_gyro                  ();
   double        read_port_motor_rpm                 ();
   double        read_stbd_motor_rpm                 ();
   double        read_motor                          ();
   double        read_roll_angle                     ();
   double        read_pitch_angle                    ();
   double        read_heading                        ();
   double        read_speed                          ();

   void          kalman_z                            ();
   void          XY_psi_model_est                    ();

   double        read_computer_battery_voltage       ();
   double        read_motor_gyro_battery_voltage     ();
   void          XY_model_est                        ();
   int           leak_check                          ();

   void          zero_gyro_data                      ();
   void          zero_surfaces                       ();
   void          initialize_adcs                     ();
   void          init_pia                            ();
```

```
   void            init_tim1a                          ();
   void            thruster_power                      ();
   void            screw_power                         ();
   void            command_control_surface             ();
   void            command_rudder                      ();
   void            command_planes                      ();
   void            command_propellors_off              ();
   void            command_thrusters_off               ();
   void            command_motor                       ();
   void            test_alive                          ();

   void            get_init_avg                        ();
   void            get_avg_rng                         ();

   void             open_device_paths                  ();
   void            close_device_paths                  ();
   void            read_parallel_port                  ();

   unsigned char   read_tim1ac1                        ();
   void            write_tim1a                         ();

   void            send_dac1                           ();
   void            send_dac2b                          ();
   int             get_adc1                            ();
   int             get_adc2                            ();

   void            Init_PortA                          ();
   void            Init_PortB                          ();
   unsigned char   Read_PortA                          ();
   unsigned char   Read_PortB                          ();
   unsigned short  Read_PortAB                         ();
   void            set_bsyA                            ();
   void            rst_bsyA                            ();
   int             ck_sta                              ();

   void            center_sonar                        ();
   char            query_sonar_1_reply                 ();
   void            set_step_size                       ();
   void            tty_mode                            ();

   void            open_virtual_world_socket           ();
   void            shutdown_virtual_world_socket       ();
   void            send_buffer_to_virtual_world_socket ();
   void            get_string_from_virtual_world_socket ();

   void            record_data                         ();

   void            execute_shutdown_script                 ();


/* Functions added to implement Dave M's speed control */

   int             port_speed_control                      ();
   int             stbd_speed_control                      ();

/* Dive Tracker Functions */

   int             createdmod                              ();
   int             CLReaddmod                              ();
   void *          AttachMod                               ();
   int             DettachMod                              ();
   void *          CreateMod                               ();


/*****************************************************************************/
/* external function prototypes                                              */

/* from external_functions.c                                                 */

extern double          degrees                              ();
extern double          radians                              ();
extern double          normalize                            ();
```

```
extern double          normalize2                          ();
extern double          radian_normalize                    ();
extern double          radian_normalize2                   ();
extern void            clamp                               ();

extern double          atan2                               ();
extern double          sinh                                ();
extern double          cosh                                ();
extern double          tanh                                ();

extern double          sign                                ();

extern void            build_telemetry_string              ();
extern void            parse_telemetry_string              ();

extern void            open_tactical_socket                ();
extern void            shutdown_tactical_socket            ();
extern void            send_buffer_to_tactical_socket      ();
extern void            get_string_from_tactical_socket     ();

extern void            record_data_on                      ();
extern void            record_data_off                     ();

extern void            cage_dg                             ();
extern void            uncage_dg                           ();

extern int             detect_death_spiral                 ();

/* from parse_functions.c                                                    */

extern void            parse_command_line_flags            ();
extern int             parse_mission_script_commands       ();
extern void            parse_mission_string_commands       ();

extern void            print_valid_keywords                ();

extern void            get_control_constants               ();

extern double          dsign                               ();
extern double          dtanh                               ();


/*****************************************************************************/

/* File Scope Globals for use by the Thruster Speed Controller Routines */

double Int_rs = 0.0,
       Int_ls = 0.0;

/* DAC Values being sent 50 props and thrusters */
int   v_dls = 512,
      v_drs = 512,
      v_dblt = 512,
      v_dslt = 512,
      v_dbvt = 512,
      v_dsvt = 512;

double sin_psi,
       cos_psi,
       cos_phi;

/* Dive Tracker Variables */
#if (defined(sun) || defined(sgi))
#else

DT2CLMem  *dt_dmod;

#endif

/*****************************************************************************/
/*****************************************************************************/
```

```
main (argc, argv)          /* Note K+R C function prototyping is due to OS-9. */
   int argc; char **argv;                          /* command line arguments. */
{

   if (TRACE && DISPLAYSCREEN) printf ("[start main:  execution]\n");

   strcpy (virtual_world_remote_host_name, VIRTUAL_WORLD_REMOTE_HOST_NAME);

   strcpy (tactical_remote_host_name, TACTICAL_REMOTE_HOST_NAME);

   dt = TIMESTEP;

   parse_command_line_flags (argc, argv);

   kal_init_z = TRUE;

   open_device_paths ();

   record_data_on ();                    /* Open files for data logging       */

   if (LOCATIONLAB)
   {
      open_virtual_world_socket ();   /* open connection to virtual world  */
      if (TRACE && DISPLAYSCREEN)
          printf ("\n[LOCATIONLAB == TRUE, open_virtual_world_socket ()]\n");
      if (strlen (buffer) > 0)
         send_buffer_to_virtual_world_socket (); /* SILENT? send to sound driver */

      strcpy (buffer, " A U V virtual world socket is open");
      send_buffer_to_virtual_world_socket ();  /* buffer containing message sent */
   }

   if (TACTICAL)                         /* must follow opening virtual world */
       open_tactical_socket ();          /* open connection to tactical level */

   strcpy (buffer, " A U V tactical socket is open");

   if (LOCATIONLAB == FALSE)
   {

#if (defined(sun) || defined(sgi))
#else
      /* Dive Tracker Stuff */
      if (DIVETRACKER)
      {
         createdmod();

         /* Fork Dive Tracker Process */
         if( (dt_pid =
            os9fork("/r0/div_trac",0,dt_fork_parmptr,0,0,0,0)) > 0)
         {
            printf("[Dive Tracker Process %d forked]\n",dt_pid);
         }
         else
         {
            printf("[Can't Fork Dive Tracker Process]\n");
         }
      }
#endif

      /* sleep gives 5 minutes to unhook wires and put AUV in the water */
printf("[Starting a 60 second sleep]\n");
      sleep (60);
printf("[Finished a 60 second sleep]\n");
      init_pia   ();
      init_timla (1);
      uncage_dg  ();

      zero_surfaces ();
      command_propellors_off ();
      command_thrusters_off  ();
      thruster_power(1);
```

```
        screw_power(1);
        initialize_adcs();
        zero_gyro_data ();
        strcpy (buffer, "EXECUTION_INITIALIZED");
        send_buffer_to_tactical_socket (); /* message */
        tsleep(10);

    }

    get_control_constants ();              /* announce filename as diagnostic   */

    EMAIL_ENTERED = FALSE;

    /*************************************************************************/
    do    /* while (LOCATIONLAB && LOOPFOREVER)                              */
    {     /*             indefinite repeat loop for long-duration lab testing  */
          /*-----------------------------------------------------------------*/

     if (DISPLAYSCREEN)
     {
       if (LOCATIONLAB && (EMAIL) && (EMAIL_ENTERED == FALSE))
       {
         strcpy (buffer, " Please Enter Your E-mail Address");
         send_buffer_to_virtual_world_socket (); /* buffer containing msg sent */
         printf ("%s *** HERE ***:  ", buffer);
         strcpy (email_address, "");
         gets (email_address);
         EMAIL_ENTERED = TRUE;
         sprintf (buffer, "Thanks");
         printf ("%s ", buffer);
         send_buffer_to_virtual_world_socket (); /* buffer containing msg sent */

         if ((int) (strlen (email_address) > 2))
         {
              sprintf (buffer, "%s\n", email_address);
              printf ("%s\n", buffer);
              send_buffer_to_virtual_world_socket (); /* buffer sent */

              if ((strcmp (email_address, "brutzman") != 0) &&
                  (strcmp (email_address, "BRUTZMAN") != 0) &&
                  (strcmp (email_address, "brutzman@nps.navy.mil") != 0))
              {
                 emailaddressfile = fopen(EMAILADDRESSFILENAME,"a"); /* append */
                 fprintf (emailaddressfile, "%s\n", email_address);
                 fclose  (emailaddressfile);
              }
         }
       }
       else if (LOCATIONLAB == FALSE) /* in water */
       {
         test_alive (10, start_dwell);
         /* Wag fin every 10 seconds for total duration of start_dwell seconds */
         printf("  Position AUV for Directional Gyro Offset Measurement\n");
         printf("  Rate Gyro zero measurement\n");
         printf("  Hit <Enter> on AUV When Ready  *** Here ! ***\n");
        /* answer = getchar (); /* pause */
       }
       printf("\n\nOK!!  Starting the mission.\n");
      }
      parse_mission_script_commands ();     /* read initial script orders      */
                                            /* ignore failure                  */
/* changed false to true so that gyros won't start in lab */
if (LOCATIONLAB) zero_gyro_data (); /* Get daily zeros for gyros*/

      if (SONARINSTALLED) center_sonar ();  /* must have open_device_paths 1st */

      strcpy (buffer, " ,,,,"); /* pause */
      send_buffer_to_virtual_world_socket ();    /* buffer containing msg sent */
      strcpy (buffer, " A U V is starting");
      send_buffer_to_virtual_world_socket ();    /* buffer containing msg sent */

       /* Initialization of closed loop parameters                            */
```

```
      buffer_index            = 0;
      telemetry_records_saved = 0;
      mission_leg_counter     = 0;
      end_test                = FALSE;
      wrap_count              = 0;
      t                       = 0.0;
      dt_time                 = 0.0;

      /*-----------------------------------------------------------------*/
      /*   Main program operational loop code                            */

      if (TRACE && DISPLAYSCREEN)
          printf ("[Starting main program operational loop code... ]");

      nextloopclock = clock () + (int)(dt * (double) CLOCKS_PER_SEC);

      /*-----------------------------------------------------------------*/
      while (end_test == FALSE) /* this is the realtime main operational loop */
                                /* when end_test == TRUE then loop is done    */
      {

         closed_loop_control_module (); /* closed loop code is here <-------- */

      }                         /* end of real-time main operational loop     */


      /* Kill Dive Tracker Process, Unlink Shared Memory, and Cage Gyro */

      if (LOCATIONLAB == FALSE)
      {
          cage_dg();

#if (defined(sun) || defined(sgi))
#else
          /* Kill Divetracker Process */
          if (DIVETRACKER)
          {
             kill(dt_pid,0);

             /* Unlink Shared Memory Module */
             ul_pid = os9exec(os9forkc,argblk[0],argblk,environ,0,0,3);
             ul_pid = os9exec(os9forkc,argblk[0],argblk,environ,0,0,3);
             ul_pid = os9exec(os9forkc,argblk[0],argblk,environ,0,0,3);
          }
#endif
      }


      /*-----------------------------------------------------------------*/

      /* lab version may repeat forever for long-duration testing:        */
      replication_count ++;

      if (LOCATIONLAB && LOOPFOREVER && DISPLAYSCREEN)
      {
          printf ("\n[LOOP FOREVER enabled, next loop is replication %d...]\n",
                  replication_count);
          sprintf (buffer, " LOOP FOREVER enabled, next loop is replication");
          send_buffer_to_virtual_world_socket ();    /* buffer msg sent */
          sprintf (buffer, " %d", replication_count);
          send_buffer_to_virtual_world_socket ();    /* buffer msg sent */
      }

      if (LOCATIONLAB && LOOPFOREVER)
      {
          /* reset amount of time to wait for next command */
          time_next_command = 0.0;
          t                 = 0.0;
          if (DISPLAYSCREEN)
          {
              printf ("\nLoopforever reset time:  [time_next_command = 0.0] ");
```

```
            printf ("  [t = 0.0]\n");
        }
    }

    if (LOOPFILEBACKUP)
    {
        record_data_off ();

#if (defined(sgi) || defined(sun))
        printf ("rm                          output.telemetry.previous\n");
        system ("rm                          output.telemetry.previous"  );
        printf ("cp   mission.output.telemetry output.telemetry.previous\n");
        system ("cp   mission.output.telemetry output.telemetry.previous"  );
        printf ("rm                          output.1_second.previous\n" );
        system ("rm                          output.1_second.previous"   );
        printf ("cp   mission.output.1_second  output.1_second.previous\n" );
        system ("cp   mission.output.1_second  output.1_second.previous"   );
#else
        printf ("del                         output.telemetry.previous\n");
        system ("del                         output.telemetry.previous"  );
        printf ("copy mission.output.telemetry output.telemetry.previous\n");
        system ("copy mission.output.telemetry output.telemetry.previous"  );
        printf ("del                         output.1_second.previous\n" );
        system ("del                         output.1_second.previous"   );
        printf ("copy mission.output.1_second  output.1_second.previous\n" );
        system ("copy mission.output.1_second  output.1_second.previous"   );
#endif
        if (LOCATIONLAB)
        {
            strcpy (buffer, " telemetry data backup complete");
            send_buffer_to_virtual_world_socket ();    /* buffer msg sent */
        }
    }
    else  /* don't bother backing up most recent results                    */
    {
        if (LOCATIONLAB && LOOPFOREVER)
        {
            if (auvtextfile) rewind (auvtextfile);
            if (((TACTICAL == FALSE) || (TACTICALPARSE)) && (auvdatafile != NULL))
            {
                rewind (auvdatafile);
                if (TRACE && DISPLAYSCREEN)
                   printf ("[auvtextfile & auvdatafile rewound to ");
                   printf ("output.data.previous & output.auv.previous]\n");
            }
            strcpy (buffer, " telemetry data backup skipped");
            send_buffer_to_virtual_world_socket ();    /* buffer msg sent */
        }
    }

    if (auvdatafile) fflush (auvscriptfile); /* force completion-file write */
    if    (fclose (auvscriptfile) == 0)
    {
          if (DISPLAYSCREEN)
           printf ("[success closing auvscriptfile mission.script.backup]\n");
    }
    else  if (DISPLAYSCREEN)
           printf ("[failure closing auvscriptfile mission.script.backup]\n");

    /* - - - - - - - - - - - - - - orders - - - - - - - - - - - - - - - - - - */

    if (auvordersfile) fflush (auvordersfile);/* force completion-file write*/
    if (auvordersfile) fclose (auvordersfile);

/*
#if (defined(sgi) || defined(sun))
    if (TRUE && DISPLAYSCREEN)
        printf ("rm                                mission.output.orders\n");
    system     ("rm                                mission.output.orders"  );
        printf ("mv   mission.output.orders.backup mission.output.orders\n");
    system     ("mv   mission.output.orders.backup mission.output.orders"  );
#else
```

```
        printf ("del                                mission.output.orders\n");
    system     ("del                                mission.output.orders"  );
        printf ("copy mission.output.orders.backup mission.output.orders\n");
    system     ("copy mission.output.orders.backup mission.output.orders"  );
        printf ("del  mission.output.orders.backup\n\n");
    system     ("del  mission.output.orders.backup"  );
#endif
*/

#if (defined(sgi) || defined(sun))
  sprintf (buffer, "rm %s.backup\n",                          AUVORDERSFILENAME);
#else
  sprintf (buffer, "del %s.backup\n",                         AUVORDERSFILENAME);
#endif
  if (DISPLAYSCREEN) printf  ("%s\n", buffer);
  system    (buffer);

#if (defined(sgi) || defined(sun))
  sprintf (buffer, "cp   %s %s.backup\n", AUVORDERSFILENAME, AUVORDERSFILENAME);
#else
  sprintf (buffer, "copy %s %s.backup\n", AUVORDERSFILENAME, AUVORDERSFILENAME);
#endif
  if (DISPLAYSCREEN) printf  ("%s\n", buffer);
  system    (buffer);


    /* - - - - - - - - - - - - - e-mail - - - - - - - - - - - - - - - - - - - - */

if (replication_count <= 2)                          /* only send e-mail once */
{
#if (defined(sgi) || defined(sun))
    sprintf (buffer, "rm %s\n",                             AUVEMAILFILENAME);
#else
    sprintf (buffer, "del %s\n",                            AUVEMAILFILENAME);
#endif
    if (DISPLAYSCREEN) printf  ("%s\n", buffer);
    system    (buffer);

#if (defined(sgi) || defined(sun))
    sprintf (buffer, "cp   %s    %s\n", AUVINFOFILENAME,   AUVEMAILFILENAME);
#else
    sprintf (buffer, "copy %s    %s\n", AUVINFOFILENAME,   AUVEMAILFILENAME);
#endif
    if (DISPLAYSCREEN) printf  ("%s\n", buffer);
    system    (buffer);

#if (defined(sgi) || defined(sun))
    sprintf (buffer, "cat  %s >> %s\n", AUVSCRIPTFILENAME, AUVEMAILFILENAME);
#else
    sprintf (buffer, "list %s >> %s\n", AUVSCRIPTFILENAME, AUVEMAILFILENAME);
#endif
    if (DISPLAYSCREEN) printf  ("%s\n", buffer);
    system    (buffer);

#if (defined(sgi) || defined(sun))
    sprintf (buffer, "cat  %s >> %s\n", AUVORDERSFILENAME, AUVEMAILFILENAME);
#else
    sprintf (buffer, "list %s >> %s\n", AUVORDERSFILENAME, AUVEMAILFILENAME);
#endif
    if (DISPLAYSCREEN) printf  ("%s\n", buffer);
    system    (buffer);

    if ((int) (strlen (email_address) >= 3) && (EMAIL))
    {
        sprintf (buffer, "mail %s < %s", email_address,     AUVEMAILFILENAME);
#if (defined(sgi) || defined(sun))
        printf ("%s\n", buffer);
        system    (buffer);
#else
    /* system    (buffer);          /* e-mail not available directly on OS-9 */
        send_buffer_to_virtual_world_socket ();     /* buffer msg sent anyway */
#endif
```

```
      }
} /* end if (replication_count <= 2) */

      /* - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -*/

      /* permit changing the vehicle mission during continuous lab testing    */
      if (LOCATIONLAB && LOOPFOREVER)
      {
         get_control_constants ();
         nextloopclock = clock () + (int)(dt * (double) CLOCKS_PER_SEC);
         record_data_on ();

         strcpy (buffer, " Load mission again");
         send_buffer_to_virtual_world_socket ();
             /* buffer containing message sent */
      }

   } while (LOCATIONLAB && LOOPFOREVER); /* end of lab infinite loop (if any) */
   /**************************************************************************/

   command_propellors_off (); /* all done, turn them off */

   if (TRACE && DISPLAYSCREEN)
       printf ("[all done, send 'kill' message to virtual world dynamics]\n");

   strcpy (buffer, "kill");              /* must start with 'shutdown' to die */
   send_buffer_to_virtual_world_socket ();  /* buffer containing message sent */

   shutdown_virtual_world_socket ();     /* close connection to virtual world */

   close_device_paths ();

   record_data_off ();

   if (TRACE && DISPLAYSCREEN)
       printf ("[finishing main:  fflush (stdout), fflush (stderr)]\n");

   fflush (stdout); /* force completion of screen write */
   fflush (stderr); /* force completion of error  write */

   if (TRACE && DISPLAYSCREEN) printf ("[main exit:  return (0)]\n");

#if (defined(sgi) || defined(sun))
   if (DISPLAYSCREEN) printf ("../gnuplot/gnuplot auv_plot_1_second.gnu\n");
   system ("../gnuplot/gnuplot auv_plot_1_second.gnu"); /* display plotted results */
#endif

   return (0); /* main program exit */

} /* end main program block, execution is complete */

/*****************************************************************************/
/*****************************************************************************/

void closed_loop_control_module ()                 /* executed each time step */

{
     double lateralMult;    /* multiple for lateral thruster voltage */
     int dt_range1, dt_range2;

     if (TRACE && DISPLAYSCREEN)
         printf ("[start  closed_loop_control_module]\n");

     if ((LOCATIONLAB == FALSE) && (HALTSCRIPT == FALSE))
     {
          if ((computer_voltage = read_computer_battery_voltage()) < 20.0)
          {
              HALTSCRIPT = TRUE;
              printf("Low Computer Voltage Detected:  %3.1f\n",computer_voltage);
          }
          if ((motor_voltage = read_motor_gyro_battery_voltage()) < 20.0)
          {
```

```
            HALTSCRIPT = TRUE;
            printf("Low Motor Voltage Detected:  %3.1f\n",motor_voltage);
        }
        if (leak_check ())
        {
            HALTSCRIPT = TRUE;
            printf("Leak Detected\n");
        }
        if (z_kal > 6.0)
        {
            HALTSCRIPT = TRUE;
            printf("Depth Exceeded\n");
        }
        if (DIVETRACKER && (dt_time + 10.0 <= t))
        {
            HALTSCRIPT = TRUE;
            printf("Loss of Dive Tracker for 30 Seconds\n");
        }

    }

    if (HALTSCRIPT)
    {
        execute_shutdown_script();
    }

    /* Speed Control ********************************************************/

    speed = read_speed ();  /* Added by D. Marco 1-12-96 */

    rpm = (port_rpm_command + stbd_rpm_command) / 2.0;

    clamp (& rpm, 700.0, -700.0, "rpm");  /* bound maximum RPM */

    if (TRACE && DISPLAYSCREEN)
        printf ("[clamp (& rpm, 700.0, -700.0, \"rpm\") complete]\n");

    /* Main_Motor RPM Control ***********************************************/

    /* note thruster use does not preclude propeller use                    */

    if (LOCATIONLAB)     /* rpm model assumes instantaneous response */
    {
       port_rpm = port_rpm_command;
       stbd_rpm = stbd_rpm_command;
    }
    else   /* in water => propeller rpms are controlled so read actual value */
    {
       port_rpm = read_port_motor_rpm ();
       stbd_rpm = read_stbd_motor_rpm ();
    }

/* --- NOT YET UPDATED TO CURRENT/OPERATIONAL MARCO AUV HARDWARE/SOFTWARE --- */

/*
if (NOT_YET_REIMPLEMENTED)
{
    main_motor_delta1 = fabs(rpm) - stbd_rpm;
    main_motor_delta2 = fabs(rpm) - port_rpm;

    /* this is reset windup for proportional integral control of motor speed */
    /*    in order to prevent accumulating the integral of speed error       */
/*     if (main_motor_delta1>50.0)  main_motor_delta1 = 50.0;
    if (main_motor_delta2>50.0)  main_motor_delta2 = 50.0;

    main_motor_volt1 = main_motor_volt1 +(rpm/fabs(rpm)*0.2*main_motor_delta1);
    main_motor_volt2 = main_motor_volt2 +(rpm/fabs(rpm)*0.2*main_motor_delta2);

    if (main_motor_volt1 > 1023) main_motor_volt1 = 1023;
    if (main_motor_volt1 < 0)    main_motor_volt1 = 0;
    if (main_motor_volt2 > 1023) main_motor_volt2 = 1023;
    if (main_motor_volt2 < 0)    main_motor_volt2 = 0;
```

```
    send_dac1(main_motor_volt1,RIGHT_MOTOR);
    send_dac1(main_motor_volt2,LEFT_MOTOR);
}
*/
    /* if using virtual world dynamics, network is source of values <<<<<<<< */

       phi   = read_roll_angle      ();    /* read roll angle  */
       cos_phi = cos (radians (phi));
       theta = read_pitch_angle     ();    /* read pitch angle */
       psi   = read_psi             ();    /* Read psi/heading */
       sin_psi = sin (radians (psi));
       cos_psi = cos (radians (psi));

       p     = read_roll_rate_gyro  ();    /* read roll rate   */
       q     = read_pitch_rate_gyro ();    /* read pitch rate  */

       r     = normalize2(psi - psi_im1)/dt; /* differentiate to get r */
       psi_im1 = psi;
/*     r     = read_yaw_rate_gyro   ();    /* Read yaw rate    */

       z     = read_depth           ();    /* Read depth       */
       kalman_z (z);
       if (TRACE) printf ("[z=%5.2f, z_kal=%5.2f]\n", z, z_kal);

       if (fabs (z_kal)      < 0.0001) z_kal      = 0.0;
       if (fabs (z_dot_kal)  < 0.0001) z_dot_kal  = 0.0;
       if (fabs (z_ddot_kal) < 0.0001) z_ddot_kal = 0.0;
   /*  z = z_kal;   no need to overwrite z value, use z_kal when you want a smoothed value
*/
       z_dot = z_dot_kal;
       w     = z_dot_kal;  /* look out!! <<<< */

    /* note:  in laboratory using virtual world, values above are superceded */

    /* estimate X and Y by dead reckoning  - - - - - - - - - - - - - - - - - - */

    /* estimate X and Y with Mathematical Model or Dead Reckoning   */

    if (LOCATIONLAB == FALSE)    /* in-water, perform a valid dead reckon   */
    {
        XY_model_est ( ((port_rpm / 700.0) * 24.0),
                       ((stbd_rpm / 700.0) * 24.0),
                   AUV_bow_lateral, AUV_stern_lateral, /* watch sign shift */
                       AUV_oceancurrent_x, AUV_oceancurrent_y, TRUE );
    }
    else                         /* virtual world providing sensor inputs   */
    {
        x += (speed * dt * cos_psi);
        if (fabs(x) <= 0.0001) x = 0.0;            /* prevent OS-9 gasping */
        y += (speed * dt * sin_psi);
        if (fabs(y) <= 0.0001) y = 0.0;            /* prevent OS-9 gasping */

        x += AUV_oceancurrent_x * dt;
        y += AUV_oceancurrent_y * dt;
    }
    z += AUV_oceancurrent_z * dt;

    if (TRACE)
    {
       printf ("[AUV_oceancurrent_x = %3.1f,",   AUV_oceancurrent_x);
       printf (" AUV_oceancurrent_y = %3.1f,",   AUV_oceancurrent_y);
       printf (" AUV_oceancurrent_z = %3.1f]\n", AUV_oceancurrent_z);
    }
    /* Control laws ****  NOTE:  all k_ constants  must be (+) positive **** */

#if (defined(sun) || defined(sgi))
#else
    /* Update Dive Tracker Ranges */
    if (DIVETRACKER && (CLReaddmod(&dt_range1,&dt_range2)==NEW_DATA) &&
       (dt_range1 < 10000) && (dt_range2 < 10000) &&
       (dt_range2 > 0) && (dt_range1 > 0))
    {
```

```
        divetracker_range1 = (double) dt_range1 / 12.0;
        divetracker_range2 = (double) dt_range2 / 12.0;
        dt_time = t;
        if ((TRACE) && (DISPLAYSCREEN))
          printf("Divetracker Ranges:  %f %f %f\n",t,divetracker_range1,divetracker_range2);
     }
     else
     {
        if (z_kal <= 1.0) dt_time = t;
     }

#endif

     waypoint_distance = sqrt (  (x - x_command) * (x - x_command)
                               + (y - y_command) * (y - y_command));

     /* HOVERCONTROL mode course control  - - - - - - - - - - - - - - - - - - - - */

     /* use WAYPOINTCONTROL (not HOVERCONTROL) until within standoff_distance */
     if   ( (HOVERCONTROL) &&
            (waypoint_distance    > standoff_distance) )
     {
          if  (waypoint_distance   > standoff_distance + 15.0)
          {
              WAYPOINTCONTROL  = TRUE;
              DEADSTICKRUDDER  = FALSE;
              DEADSTICKPLANES  = FALSE;
              port_rpm_command = 700;
              stbd_rpm_command = 700;
              psi_command      = psi_command_hover;
              fprintf (auvordersfile,
"%6.1f  %6.1f %5.1f %5.1f %5.1f %6.1f %6.1f %6.1f %6.1f  %5.1f %5.1f %5.1f %5.1f\n",
                                t, psi_command, x_command, y_command, z_command,
                            port_rpm_command, stbd_rpm_command,
                                rudder_command,   planes_command,
                                   AUV_bow_vertical,
                                 AUV_stern_vertical,
                                   AUV_bow_lateral,
                                  AUV_stern_lateral);
          }
          else
          {
              WAYPOINTCONTROL  = FALSE;
              DEADSTICKRUDDER  = TRUE;
              DEADSTICKPLANES  = TRUE;
           /* port_rpm_command reset needed ?? */
           /* stbd_rpm_command reset needed ?? */
              psi_command      = psi_command_hover;
              fprintf (auvordersfile,
"%6.1f  %6.1f %5.1f %5.1f %5.1f %6.1f %6.1f %6.1f %6.1f  %5.1f %5.1f %5.1f %5.1f\n",
                                t, psi_command, x_command, y_command, z_command,
                            port_rpm_command, stbd_rpm_command,
                                rudder_command,   planes_command,
                                   AUV_bow_vertical,
                                 AUV_stern_vertical,
                                   AUV_bow_lateral,
                                  AUV_stern_lateral);
          }
     }
     else if ((HOVERCONTROL) && (WAYPOINTCONTROL))
          /* restore proper HOVERCONTROL, we are now closer */
     {
          WAYPOINTCONTROL  = FALSE;
          DEADSTICKRUDDER  = TRUE;
          DEADSTICKPLANES  = TRUE;
          rudder_command   = 0.0;
          psi_command      = psi_command_hover;
          port_rpm_command = 0;
          stbd_rpm_command = 0;
              fprintf (auvordersfile,
"%6.1f  %6.1f %5.1f %5.1f %5.1f %6.1f %6.1f %6.1f %6.1f  %5.1f %5.1f %5.1f %5.1f\n",
                                t, psi_command, x_command, y_command, z_command,
```

```
                        port_rpm_command, stbd_rpm_command,
                          rudder_command,   planes_command,
                             AUV_bow_vertical,
                           AUV_stern_vertical,
                              AUV_bow_lateral,
                            AUV_stern_lateral);
}
if ((HOVERCONTROL) && (GPSFIXINPROGRESS == FALSE) &&
     ((    waypoint_distance     > standoff_distance) ||
      (fabs ( depth_error)     > standoff_distance) ||
      (fabs (psi_error) > 10.0 /* degrees */ )))   /* cylinder test */
{
       /* still not at the hoverpoint */
       if (TRACE && DISPLAYSCREEN) printf ("[HOVERCONTROL cylinder test]");
       /* continue until hoverpt reached without further script orders */
       time_next_command = t + 2.0 * dt;
}
/* report STABLE to tactical level once hoverpoint received                  */
if ((HOVERCONTROL) && (REPORTSTABLE)  &&
     ((    waypoint_distance     < standoff_distance) &&
      (fabs ( depth_error)     < standoff_distance) &&
      (fabs (psi_error) < 10.0 /* degrees */ )))
{
       if ((TACTICAL) && (GPSFIXINPROGRESS == FALSE))
       {
          REPORTSTABLE = FALSE;
          if (TRACE && DISPLAYSCREEN)
              printf ("\n[send_buffer_to_tactical_socket (STABLE HOVER)]\n");
          strcpy (buffer, "STABLE HOVER");
          send_buffer_to_tactical_socket (); /* message */
       }
}

/* WAYPOINTCONTROL mode course control - - - - - - - - - - - - - - - - - - - - */
if (WAYPOINTCONTROL)
{
   /* minimum headway speed test & correction  - - - - - - - - - - - - - - - - */
   if (   (port_rpm_command < 200.0)  /* safety check propulsion is positive */
       || (stbd_rpm_command < 200.0))
   {
       if (TRACE && DISPLAYSCREEN)
    printf ("\n[WAYPOINTCONTROL fabs (rpm) < 200.0, too low! reset to 200.0]");
       port_rpm_command = 200.0;
       stbd_rpm_command = 200.0;
   }

   /* note that a reversed x,y calling sequence is necessary                  */
   /*   in order to get correct quadrant alignment                            */
   /*   also try to lead waypoint to account for AUV_oceancurrent set/drift   */

   waypoint_angle=atan2 (y_command - y + AUV_oceancurrent_y * dt,
                         x_command - x + AUV_oceancurrent_x * dt);

   waypoint_angle= normalize (degrees (waypoint_angle));
   psi_command   = waypoint_angle;

   /* If the auv is closer to the waypoint than this value, there */
   /* is a danger that it could enter a death spiral              */
   death_spiral_radius = fabs (sin (radians (normalize2 (waypoint_angle - psi))))
                         * (1.0 / 700.0) * rpm * 15.0;

   if (TRACE && DISPLAYSCREEN)
   {
       printf ("WAYPOINTCONTROL psi_command = %5.1f, ", psi_command);
       printf ("x = %5.1f, y = %5.1f\n", x, y);
   }
   if ((FOLLOWWAYPOINTMODE) && (HOVERCONTROL == FALSE) &&
       (! ((fabs (depth_error) <= standoff_distance) &&
           ((waypoint_distance <= standoff_distance) ||
            (waypoint_distance <= death_spiral_radius) ||
            (detect_death_spiral (FALSE)))) ))
     {
```

```
        if (TRACE && DISPLAYSCREEN)
              printf ("\n[FOLLOWWAYPOINTMODE cylinder test]");
           /* continue until WAYPOINT reached without further script orders */
           time_next_command = t + 2.0 * dt;
        }
        else if ((fabs (depth_error) <= standoff_distance) &&
                 ((waypoint_distance <= standoff_distance) ||
                  (waypoint_distance <= death_spiral_radius) ||
                  (detect_death_spiral (FALSE))))
        {

          WAYPOINTCONTROL = FALSE;
          FOLLOWWAYPOINTMODE = FALSE;

          if (TRACE && DISPLAYSCREEN)
              printf ("\n[FOLLOWWAYPOINTMODE success, WAYPOINT reached]");
              fprintf (auvordersfile,
"%6.1f  %6.1f %5.1f %5.1f %5.1f %6.1f %6.1f %6.1f %6.1f  %5.1f %5.1f %5.1f %5.1f\n",
                              t, psi_command, x_command, y_command, z_command,
                            port_rpm_command, stbd_rpm_command,
                              rudder_command,   planes_command,
                                 AUV_bow_vertical, AUV_stern_vertical,
                                 AUV_bow_lateral,  AUV_stern_lateral);

          /* report STABLE to tactical level once waypoint received              */
        if ((TACTICAL) && (REPORTSTABLE)
             && (GPSFIXINPROGRESS == FALSE))
          {
             REPORTSTABLE = FALSE;
             if (TRACE && DISPLAYSCREEN)
                 printf ("\n[send_buffer_to_tactical_socket (STABLE WAYPOINT)]\n");
             strcpy (buffer, "STABLE WAYPOINT");
             send_buffer_to_tactical_socket (); /* message */
          }
        }
      }
      else if (HOVERCONTROL)
      {
        waypoint_angle=normalize(degrees(atan2(y_command - y, x_command - x)));

        track_angle = normalize (waypoint_angle - psi);
        along_track_distance =   cos (radians(track_angle)) * waypoint_distance;
        cross_track_distance = - sin (radians(track_angle)) * waypoint_distance;

        port_rpm_command =   k_propeller_hover * along_track_distance
                           - k_propeller_current * AUV_oceancurrent_x
                                                 * cos_psi
                           - k_propeller_current * AUV_oceancurrent_y
                                                 * sin_psi
                           - k_surge_hover * u;
        stbd_rpm_command =   k_propeller_hover * along_track_distance
                           - k_propeller_current * AUV_oceancurrent_x
                                                 * cos_psi
                           - k_propeller_current * AUV_oceancurrent_y
                                                 * sin_psi
                           - k_surge_hover * u;

        if (TRACE && DISPLAYSCREEN)
        {
            printf ("\nHOVERCONTROL:\n");
            printf ("psi_command          = %5.1f, ", psi_command);
            printf ("x = %5.1f, y = %5.1f\n", x, y);
            printf ("waypoint_distance    = %5.1f, track_angle = %5.1f\n",
                     waypoint_distance,         track_angle);
            printf ("along_track_distance = %5.1f, ", along_track_distance);
            printf ("cross_track_distance = %5.1f\n", cross_track_distance);
            printf ("port_rpm & stbd_rpm  = %5.1f\n", port_rpm);
        }
      }

     /* Simplified PD rudders/planes control rules: - - - - - - - - - - - - - */
```

```
/* calculate rudders - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - */

delta_rudder =   k_psi *  psi_error

               + (k_r   * r) + (k_v   * v);

/*  tanh not provided under OS-9 C, added at end of this program        */
/*  tanh was change to dtanh toconserve time in approximation           */
if (SLIDINGMODECOURSE)
{
   sigma = k_sigma_r * r + k_sigma_psi * psi_error;

   delta_rudder =  (3.1403 * r) + 81.9712 * eta_steering * dtanh (sigma);
}

/* reduce ordered rudder if excessive roll occurs, may work for many UUVs*/

delta_rudder =  delta_rudder * cos_phi * cos_phi;

if (TRUE && DISPLAYSCREEN && (cos_phi * cos_phi < 0.98))
{
    printf ("\nrudder/planes reduction factor due to roll phi = %6.3f\n",
             cos_phi * cos_phi);
}

depth_error  = (z_kal - z_command);

/* constrain depth_error to +- 15.0 feet to prevent going vertical       */
/*       and enable stable pitch angle even on large depth changes       */

clamp (& depth_error, -15.0, 15.0, "depth_error");               /* feet */

/* calculate planes - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - */

delta_planes = - (k_z * depth_error)

               + (k_theta * theta) + (k_q * q) - (k_w * z_dot_kal);

/* temporary fix to incorrect delta_planes sign above <<<<<<<<<<<<<<<<<< */
if (LOCATIONLAB == FALSE)     delta_planes = - delta_planes;

/* reduce ordered planes if excessive roll occurs, may work for many UUVs*/

delta_planes =  delta_planes * cos_phi * cos_phi;

/* Dead stick means no open loop control of rudders/planes - - - - - - - */

if (DEADSTICKRUDDER)
{
    delta_rudder = rudder_command;
}
if (DEADSTICKPLANES)
{
    delta_planes = planes_command;
}

/* Normalization - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - */

delta_rudder =   normalize2 (delta_rudder);
delta_planes =   normalize2 (delta_planes);

/* constrain planes & rudder orders +/- 22.5 degrees                     */

clamp (& delta_rudder, -22.5, 22.5, "delta_rudder");          /* degrees */
clamp (& delta_planes, -22.5, 22.5, "delta_planes");          /* degrees */

/* watch out for planes/rudder chatter and zero them at 0 rpm if needed! */

/* Simplified lateral/vertical thruster control rules: - - - - - - - - - - */

if      (ROTATECONTROL)  /* open loop rotate  thrusters */
{
```

```
   lateral_thruster_volts =   k_thruster_rotate * rotate_command;
   /* insert * fabs (rotate_command) <<<<<<<<<<< */
}
else if (LATERALCONTROL) /* open loop lateral thrusters */
{
   lateral_thruster_volts = - k_thruster_lateral * (lateral_command);
}
else                              /* heading control is default */
{
   lateral_thruster_volts =   - k_thruster_psi * psi_error

                              - k_thruster_r   * r;
}

vertical_thruster_volts   = - k_thruster_z * (z_kal - z_command)

                              - k_thruster_w * z_dot_kal;


if ((THRUSTERCONTROL) || (HOVERCONTROL) ||
    (ROTATECONTROL) || (LATERALCONTROL))
{
    AUV_bow_vertical    =   vertical_thruster_volts;
    AUV_stern_vertical =   vertical_thruster_volts;

    if      (LATERALCONTROL)
    {
       if (TRACE && DISPLAYSCREEN)
       {
           printf ("(LATERALCONTROL == TRUE, ");
           printf ("lateral_command == %5.2f, ", lateral_command);
           printf ("k_thruster_lateral == %5.2f)\n", k_thruster_lateral);
       }

       AUV_bow_lateral    =     lateral_thruster_volts; /* both positive, */
       AUV_stern_lateral =     lateral_thruster_volts; /* same direction */
    }
    else if (HOVERCONTROL)
    {
       if (TRACE && DISPLAYSCREEN) printf ("(HOVERCONTROL == TRUE)\n");

       AUV_bow_lateral    = - ( - k_thruster_psi * psi_error

                                - k_thruster_r   * r)

                             + k_thruster_hover   * cross_track_distance

                             - k_thruster_current * AUV_oceancurrent_x
                                                  * sin_psi
                             + k_thruster_current * AUV_oceancurrent_y
                                                  * cos_psi

                             + k_sway_hover      * v;

       AUV_stern_lateral =   ( - k_thruster_psi * psi_error

                                - k_thruster_r   * r)

                             + k_thruster_hover   * cross_track_distance

                             - k_thruster_current * AUV_oceancurrent_x
                                                  * sin_psi
                             + k_thruster_current * AUV_oceancurrent_y
                                                  * cos_psi

                             + k_sway_hover       * v;
    }
    else if ((THRUSTERCONTROL) || (ROTATECONTROL))
    {
       if (TRACE && DISPLAYSCREEN)
           printf ("(THRUSTERCONTROL == TRUE) || (ROTATECONTROL == TRUE)\n");
```

```
        AUV_bow_lateral     =  - lateral_thruster_volts; /* negative */
        AUV_stern_lateral   =    lateral_thruster_volts;
    }
    else
    {
        printf ("Thruster control logic error *** \n");
    }

    if (TRACE && DISPLAYSCREEN)
    {
        printf ( "\nThruster control ON.  Pre-clamp calculated values:\n");
        printf (" psi = %6.3f, psi_command = %6.3f\n",
                 psi,          psi_command);
        printf (" AUV_bow_vertical = %6.3f, AUV_stern_vertical = %6.3f\n",
                 AUV_bow_vertical,          AUV_stern_vertical);
        printf (" AUV_bow_lateral  = %6.3f, AUV_stern_lateral  = %6.3f\n",
                 AUV_bow_lateral,           AUV_stern_lateral);
    }
}
else /* thrusters disabled */
{
    if (TRACE && DISPLAYSCREEN)
    {
        printf ( "Thruster control OFF.  Pre-clamp calculated values:\n");
        printf ("vertical_thruster_volts = %6.3f\n",
                vertical_thruster_volts);
        printf ( "lateral_thruster_volts = %6.3f\n",
                 lateral_thruster_volts);
    }
    AUV_bow_vertical   =  0.0;
    AUV_stern_vertical =  0.0;
    AUV_bow_lateral    =  0.0;
    AUV_stern_lateral  =  0.0;
}

if (TRACE && DISPLAYSCREEN)
{
    printf ("Pre-sqrt thruster control calculated values:\n");
    printf ("AUV_bow_vertical   = %6.3f\n", AUV_bow_vertical);
    printf ("AUV_stern_vertical = %6.3f\n", AUV_stern_vertical);
    printf ("AUV_bow_lateral    = %6.3f\n", AUV_bow_lateral);
    printf ("AUV_stern_lateral  = %6.3f\n", AUV_stern_lateral);
}

/* convert to signed sqrt to account for volts-to-thrust relationship    */

/* different multiple required between lab and auv because of polarity   */
/* discrepancy between virtual world and actual auv                       */
if (LOCATIONLAB) lateralMult = 2.0;
else lateralMult = -2.0;

/*  2.0 * sqrt(6.0) = 4.8989      SQR 6 = 2.449 */
AUV_bow_vertical   =  4.8989 * sign (AUV_bow_vertical  )
                             * sqrt (fabs (AUV_bow_vertical  ));
AUV_stern_vertical = 4.8989 * sign (AUV_stern_vertical)
                             * sqrt (fabs (AUV_stern_vertical));
AUV_bow_lateral    = lateralMult * 2.449 * sign (AUV_bow_lateral   )
                             * sqrt (fabs (AUV_bow_lateral   ));
AUV_stern_lateral  = lateralMult * 2.449 * sign (AUV_stern_lateral )
                             * sqrt (fabs (AUV_stern_lateral ));

if (TRACE && DISPLAYSCREEN)
{
    printf ("Post-sqrt thruster control calculated values:\n");
    printf ("AUV_bow_vertical   = %6.3f\n", AUV_bow_vertical);
    printf ("AUV_stern_vertical = %6.3f\n", AUV_stern_vertical);
    printf ("AUV_bow_lateral    = %6.3f\n", AUV_bow_lateral);
    printf ("AUV_stern_lateral  = %6.3f\n", AUV_stern_lateral);
}

/* constrain thruster orders  +/- 24.0 volts == 3820 rpm no-load         */
/* constrain propeller orders +/- 700 rpm no-load                        */
```

```
clamp (&  AUV_bow_vertical,   -24.0, 24.0, "AUV_bow_vertical");
clamp (&  AUV_stern_vertical, -24.0, 24.0, "AUV_stern_vertical");
clamp (&  AUV_bow_lateral,    -24.0, 24.0, "AUV_bow_lateral");
clamp (&  AUV_stern_lateral,  -24.0, 24.0, "AUV_stern_lateral");

clamp (&  port_rpm_command,  -700.0, 700.0, "port_rpm_command");
clamp (&  stbd_rpm_command,  -700.0, 700.0, "stbd_rpm_command");


/* command thruster and propellor orders                                  */

command_motor (AUV_bow_vertical,     BOW_VERTICAL);
command_motor (AUV_stern_vertical, STERN_VERTICAL);
command_motor (AUV_bow_lateral,      BOW_LATERAL);
command_motor (AUV_stern_lateral,  STERN_LATERAL);
command_motor (port_rpm_command,    PORT_PROP);
command_motor (stbd_rpm_command,    STBD_PROP);

/* Send commands to rudders and planes ***********************************/

command_rudder (delta_rudder);
command_planes (delta_planes);

/* Send command & get reply from sonar ***********************************/

AUV_ST1000_bearing = 0.0;            /* relative bearings of sonar heads */
AUV_ST725_bearing  = 0.0;

/* send telemetry to tactical level and data recording files - - - - - - */
record_data ();

/* read commands from tactical level - - - - - - - - - - - - - - - - - - */
/* if (TACTICAL) read_parallel_port ();[old code] now uses socket*/


/* update simulation clock "t" - - - - - - - - - - - - - - - - - - - - - */

t = t + dt;

fflush (stdout); /* force completion of screen write */
currentloopclock = clock ();

if (TRACE && REALTIME)
    printf("[Unused Loop Time: %5.4f\n",
          (float)(nextloopclock - currentloopclock) / (float) CLOCKS_PER_SEC);

if ((REALTIME) &&
    (currentloopclock < nextloopclock))
{
     if (TRACE && DISPLAYSCREEN)
     {
        printf ("currentloopclock = %ld, nextloopclock = %ld\n",
                currentloopclock,      nextloopclock);
   printf("timestep dt = %5.3f seconds (corresponding clock ticks = %d)\n",
                dt, (int)(dt * (double) CLOCKS_PER_SEC));
        printf ("Busy wait until system clock reaches simulation clock, ");
        printf ("loop duration = %5.3f\n",
                ((double) currentloopclock - (double) nextloopclock)
                 / CLOCKS_PER_SEC);
     }
     while  (currentloopclock < nextloopclock)
     {
         currentloopclock = clock (); /* %%%%% busy wait %%%%% */
     }
     if (TRACE && DISPLAYSCREEN)
     {
         printf ("Busy wait complete, loop+wait duration = %5.3f, ",
                 ((double) currentloopclock - (double) nextloopclock)
                  / CLOCKS_PER_SEC);
         printf ("current clock () = %ld\n", currentloopclock);
     }
}
```

```
    else if ((REALTIME == FALSE) && LOCATIONLAB && DISPLAYSCREEN && TRACE)
         {
             printf ("No busy wait, loop duration = %5.3f, ",
                     ( (double) currentloopclock - (double) nextloopclock)
                     / CLOCKS_PER_SEC);
             printf ("current clock () = %ld\n", currentloopclock);
         }

    nextloopclock = clock () + (int)(dt * (double) CLOCKS_PER_SEC);

    /*   determine if closed_loop_control_module () is done; if so,  - - - - */
    /* repeat call to closed_loop_control_module () for each time step   - - */

    /* this needs to be 3-way logic file | tactical | keyboard <<<<<<<<<<<<< */

    if ((KEYBOARDINPUT) || (TACTICAL))
    {
         parse_mission_script_commands ();     /* get next script orders read */
                                               /* ignore failure              */
         /* we are reading from the mission script file in these cases       */

    }
    else if (auvdatafile == NULL)      /* file never opened, loop and open it */
    {
         end_test = TRUE;
    }
    else if (feof (auvscriptfile) && (t > time_next_command))     /* all done */
    {
         if (TRACE && DISPLAYSCREEN) printf ("end_test set TRUE\n");
         end_test = TRUE;
    }
    else if (t > time_next_command)  /* scriptfile not yet closed, read more */
    {
         if (TRACE && DISPLAYSCREEN)
            printf ("\n[read more from parse_mission_script_commands]\n");
         parse_mission_script_commands ();     /* get next script orders read */
                                               /* ignore failure              */
    }


   /* else not done executing current script command, continue/don't block  */

    if (HALTSCRIPT == FALSE)
       psi_error = normalize2 (psi - psi_command);
       else psi_error = 0.0;

    if (TRACE && DISPLAYSCREEN)
        printf ("\n[time_next_command = %5.1f]\n", time_next_command);

    if (TRACE && DISPLAYSCREEN)
        printf ("\n[finish closed_loop_control_module ()]\n");

    if (end_test)
    {
        command_motor (0.0, BOW_VERTICAL);
        command_motor (0.0, STERN_VERTICAL);
        command_motor (0.0, BOW_LATERAL);
        command_motor (0.0, STERN_LATERAL);
        command_motor (0.0, PORT_PROP);
        command_motor (0.0, STBD_PROP);
    }

    return;

} /* end closed_loop_control_module () */

/*****************************************************************************/

/* The following four functions were added on 12 Dec 95 */
/* They are from Dave Marco's execution code and are used */
/* for speed control of the port propellors */
```

```
int port_speed_control(n_com)
                double n_com;  /* revolutions per second */
{
   double Km_ls = 0.6589,
         e_n,v_ls_spc,
         eta_ls = 10.0,
         phi_ls = 5.0;

   if(fabs(n_com) < 0.25) Int_ls = 0.0;

   e_n = n_com - read_port_motor_rpm () / 60.0;
   Int_ls = Int_ls + dtanh(e_n/phi_ls)*dt;

   v_ls_spc = (1.0/Km_ls)*(n_com + eta_ls*Int_ls);

   v_dls = (int) ((1023.0/48.0)*(v_ls_spc)  + 511.5);

   if(v_dls < 0 )      v_dls  = 0;
   if(v_dls > 1023 )   v_dls  = 1023;
   return(v_dls);

} /* end port_speed_control () */

/*****************************************************************************/

int stbd_speed_control (n_com)
                double n_com;  /* revolutions per second */
{
   double Km_rs = 0.6156,
         e_n,v_rs_spc,
         eta_rs = 10.0,
         phi_rs = 5.0;

   if(fabs(n_com) < 0.25) Int_rs = 0.0;

   e_n = n_com - read_stbd_motor_rpm() / 60.0;

   Int_rs = Int_rs + dtanh(e_n/phi_rs)*dt;

   v_rs_spc = (1.0/Km_rs)*(n_com + eta_rs*Int_rs);

   v_drs  = (int) ((1023.0/48.0)*(v_rs_spc)  + 511.5);

   if(v_drs < 0 )      v_drs  = 0;
   if(v_drs > 1023 )   v_drs  = 1023;
   return(v_drs);

} /* end stbd_speed_control () */

/*****************************************************************************/

/* ---     VERIFIED MATCH CURRENT/OPERATIONAL MARCO AUV HARDWARE/SOFTWARE --- */

double read_depth () /* Return depth in FEET */
{
     int    val      = 0;
     double new_z    = 0.0;  /* zz in dave's execf.c code */
     double z_offset = 0.0;

     if (TRACE && DISPLAYSCREEN) printf ("\n[start  read_depth ()]\n");

     if (LOCATIONLAB && DEADRECKON)
     {
         new_z = z_command;
     }
     else if (LOCATIONLAB)
     {
         new_z = z; /* no change, use virtual world value */
     }
     else /* in-water */
     {
         /* val = adc1(DEPTH_CELL_CH); */ /* Channel 7 */
```

```
        /* 0.0728 = 0.0182*4.0 */
        /* Since A/D now has 0-1023 range instead of 0-4095 */
        /* new_z = 0.0728*( (double) (val - z_val0)) + z_offset; */

        /* adc2 card has 0 - 4095 resolution */
        val = get_adc2 (DEPTH_CELL_CH, 0);
        new_z = 0.0182*( (double) (val - z_val0)) + z_offset;

        /* Calibration for Signal Amp */
        /*new_z = 0.0034285*( (double) (z_val0 - val)) + z_offset;*/
    }

/*
if (ARCHAIC_IGNORE)
{
    z_offset = 0.0;
    val      = get_adc2 (0,0);
    new_z    = 0.002237 * (val - z_val0) + z_offset;    /* new_z (ft) */
/*}
*/

    if (TRACE && DISPLAYSCREEN)
        printf ("\n[finish read_depth (), returns %5.3f]\n", new_z);

    return (new_z + depth_cell_bias);

} /* end read_depth () */

/*****************************************************************************/

/* ---     VERIFIED MATCH CURRENT/OPERATIONAL MARCO AUV HARDWARE/SOFTWARE --- */

double read_psi ()       /* return psi in degrees */

{
    unsigned short psi_bit;
    int psi_bit_int,psi_bit_old_int,delta_psi_bit;
    double angle,tpi;
    double pi = 3.1415927;

    if (TRACE && DISPLAYSCREEN) printf ("[start  read_psi ()]\n");

    if (LOCATIONLAB && DEADRECKON)
    {
        angle = psi_command;
    }
    else  if (LOCATIONLAB)
    {
        psi   = psi; /* no change, use virtual world value */
        angle = psi; /* set up for function return         */
    }
    else /* in-water */
    {
        psi_bit = Read_PortAB(0xFFF00700);
        psi_bit &= 0x3FFF;
        psi_bit_int = psi_bit;
        psi_bit_old_int = psi_bit_old;

        delta_psi_bit = psi_bit_int - psi_bit_old_int;
        psi_bit_old = psi_bit;

        if(abs(delta_psi_bit) > 10000)
        {
           wrap_count = wrap_count - delta_psi_bit/abs(delta_psi_bit);
        }

        angle = start_psi + degrees ((read_heading () -
                          dg_offset + 2.0*pi*((double) wrap_count)));

        if(fabs(angle) < 0.0001) angle = 0.0;
        /*printf("%f %f %f %d %d\n",
               angle,read_heading (),dg_offset,wrap_count,psi_bit); */
```

```
     }

/*
if (ARCHAIC_IGNORE)
{
     /* port needs to be redone:  */
/*     psi_bit = Read_PortAB((struct MFI_PIA *) MFI_BASE);
     psi_bit &= 0x3FFF;
     psi_bit_int = psi_bit;
     psi_bit_old_int = psi_bit_old;

     delta_psi_bit = psi_bit_int - psi_bit_old_int;
     psi_bit_old = psi_bit;

     if(abs(delta_psi_bit) > 10000)
     {
        wrap_count = wrap_count - delta_psi_bit/abs(delta_psi_bit);
     }
     tpi = 2.0 * pi * wrap_count;

     angle = read_heading () - dg_offset + tpi;

     angle = degrees (angle);
}
*/

  if (TRACE && DISPLAYSCREEN)
      printf ("[finish read_psi () returns %5.3f]\n", angle);

  return (normalize (angle));

}  /* end read_psi () */

/*****************************************************************************/

/* ---     VERIFIED MATCH CURRENT/OPERATIONAL MARCO AUV HARDWARE/SOFTWARE --- */

double read_roll_rate_gyro ()    /* Return roll rate in DEGREES/SEC          */
{
     int    val;
     double rate;

     if (TRACE && DISPLAYSCREEN) printf ("[start  read_roll_rate_gyro ()]\n");

     if (LOCATIONLAB)
     {
         rate = p; /* no change, use virtual world value */
         if (fabs (rate) < 0.0001) rate = 0.0;
     }
     else /* in-water */
     {
         val = get_adc2(ROLL_RATE_CH,0);
         /* Next two lines from old method */
         /*val = val >> 2;*/ /* Quick fix for new res */
         /*rate = (roll_rate_0/3.2113 - .31062*val)/57.295779;*/
         rate = degrees (0.07785*(roll_rate_0 - val)/57.295779);
         if(fabs(rate) < 0.0001) rate = 0.0;
     }

/*
if (ARCHAIC_IGNORE)
{
   if (LOCATIONLAB == FALSE) /* not in virtual world, read slots */
/*   {
     val  = get_adc1 (ROLL_RATE_CH);
     rate = (roll_rate_0/3.2113 - .31062*val)/57.295779;
   }
}
*/
     rate = normalize2 (rate);

     if (TRACE && DISPLAYSCREEN)
```

```
        printf ("[finish read_roll_rate_gyro () returns %5.3f]\n", rate);

    return (rate);

}  /* end read_roll_rate_gyro () */

/*****************************************************************************/

/* ---     VERIFIED MATCH CURRENT/OPERATIONAL MARCO AUV HARDWARE/SOFTWARE --- */

double read_pitch_rate_gyro ()    /* Return pitch rate in DEGREES/SEC          */
{
    int    val  = 0;
    double rate;

    if (TRACE && DISPLAYSCREEN) printf ("[start  read_pitch_rate_gyro ()]\n");

     if (LOCATIONLAB)
    {
        rate = q; /* no change, use virtual world value */
        if (fabs (rate) < 0.0001) rate = 0.0;
    }
    else /* in-water */
    {
        val = get_adc2(PITCH_RATE_CH,0);
        /* Next two lines from old method */
        /*val = val >> 2;*/ /* Quick fix for new res */
        /*rate = (pitch_rate_0/13.69399 - .0730001*val)/57.295779;*/
        rate = degrees (0.01825*(pitch_rate_0 - val)/57.295779);
        if(fabs(rate) < 0.0001) rate = 0.0;
    }

/*
if (ARCHAIC_IGNORE)
{
   if (LOCATIONLAB == FALSE) /* not in virtual world, read slots */
/*   {
    val    = get_adc1 (PITCH_RATE_CH);
    rate   = (pitch_rate_0/13.69399 - .0730001*val)/57.295779;
   }
}
*/
    rate = normalize2 (rate);

    if (TRACE && DISPLAYSCREEN)
        printf ("[finish read_pitch_rate_gyro () returns %5.3f]\n", rate);

    return (rate);

}  /* end read_pitch_rate_gyro () */

/*****************************************************************************/

/* ---     VERIFIED MATCH CURRENT/OPERATIONAL MARCO AUV HARDWARE/SOFTWARE --- */

double read_yaw_rate_gyro ()    /* Return yaw rate in DEGREES/SEC              */
{
    int    val  = 0;
    double rate;

    if (TRACE && DISPLAYSCREEN) printf ("[start  read_yaw_rate_gyro ()]\n");

     if (LOCATIONLAB)
    {
        rate = r; /* no change, use virtual world value */
        if (fabs (rate) < 0.0001) rate = 0.0;
    }
    else /* in-water */
    {
        /* Below for adc1 Card */
        /*val = adc1(YAW_RATE_CH);*/ /* Channel 10 */
        /*rate = 2.78*( ((double) yaw_rate_0)/13.653216 -
```

```
                                    0.0732362*( (double) val) )/57.295779;*/

        val = get_adc2(YAW_RATE_CH,0);
        /* Next two lines from old method */
        /*val = val >> 2;*/ /* Quick fix for new res */
        /*rate = 2.78*(yaw_rate_0/13.653216 - .0732362*val)/57.295779;*/
        rate = degrees (0.0509*(yaw_rate_0 - val)/57.295779);
        if(fabs(rate) < 0.0001) rate = 0.0;
    }

/*
if (ARCHAIC_IGNORE)
{
   if (LOCATIONLAB == FALSE) /* not in virtual world, read slots */
/*   {
     val  = get_adc1 (YAW_RATE_CH);
     rate = (yaw_rate_0/13.653216 - .0732362*val)/57.295779;
   }
}
*/
     rate = normalize2 (rate);

     if (TRACE && DISPLAYSCREEN)
         printf ("[finish read_yaw_rate_gyro () returns %5.3f]\n", rate);

     return (rate);

}  /* end read_yaw_rate_gyro () */

/*****************************************************************************/

/* ---     VERIFIED MATCH CURRENT/OPERATIONAL MARCO AUV HARDWARE/SOFTWARE --- */

double read_port_motor_rpm ()    /* Reads rpm from PORT_PROP                 */
{
     int    pulse;
     double local_port_rpm;

     if (TRACE && DISPLAYSCREEN) printf ("[start  read_port_motor_rpm ()]\n");

     local_port_rpm = read_motor (PORT_PROP);

/*
if (ARCHAIC_IGNORE)
{
     pulse    = get_adc1 (LEFT_MOTOR_RPM);
     local_port_rpm = 1.244*pulse - 8.4792;
}
*/
     if (TRACE && DISPLAYSCREEN)
         printf ("[finish read_port_motor_rpm () returns %5.3f]\n",
                   local_port_rpm);

     return (local_port_rpm);

}   /* end read_port_motor_rpm () */

/*****************************************************************************/

/* ---     VERIFIED MATCH CURRENT/OPERATIONAL MARCO AUV HARDWARE/SOFTWARE --- */

double read_stbd_motor_rpm ()    /* Reads rpm from STBD_PROP                 */
{
     int    pulse;
     double local_stbd_rpm;

     if (TRACE && DISPLAYSCREEN) printf ("[start  read_stbd_motor_rpm ()]\n");

     local_stbd_rpm = read_motor (STBD_PROP);

/*
if (ARCHAIC_IGNORE)
```

```
{
     pulse      = get_adc1 (RIGHT_MOTOR_RPM);
     local_stbd_rpm = 1.244*pulse - 8.4792;
}
*/
     if (TRACE && DISPLAYSCREEN)
          printf ("[finish read_stbd_motor_rpm () returns %5.3f]\n",
                    local_stbd_rpm);

     return (local_stbd_rpm);

}   /* end read_stbd_motor_rpm () */

/*****************************************************************************/

/* ---     VERIFIED MATCH CURRENT/OPERATIONAL MARCO AUV HARDWARE/SOFTWARE --- */

double read_motor (motor)       /* Read rpm from single propellor or thruster */
               int motor;
{
/*      motor = 0  Left  Propeller                PORT_PROP       RPM
                1  Right Propeller                STBD_PROP       RPM
                2    Bow Vertical Thruster         BOW_VERTICAL   volts
                3    Bow Lateral  Thruster       STERN_VERTICAL   volts
                4  Stern Vertical Thruster         BOW_LATERAL    volts
                5  Stern Lateral  Thruster       STERN_LATERAL    volts
*/
   int count;
   double freq,rps;
   unsigned char lobyte,hibyte;

   if (TRACE && DISPLAYSCREEN) printf ("[start  read_motor ()]\n");

   if (LOCATIONLAB == FALSE) /* in water */
   {
    switch(motor)
    {
      case PORT_PROP:
         write_tim1a(3,tim_1a_control_reg,17); /* Sel Cntr 1 HOLD Reg. Card 3 */
         lobyte = read_tim1ac1(3,tim_1a_data_reg);
         hibyte = read_tim1ac1(3,tim_1a_data_reg);
         count = (int) (256*hibyte) + (int) lobyte;
         if(v_dls < 512 ) count = -count; /* Account for Direction of Rot. */
         break;

      case STBD_PROP:
         write_tim1a(3,tim_1a_control_reg,18); /* Sel Cntr 2 HOLD Reg. Card 3 */
         lobyte = read_tim1ac1(3,tim_1a_data_reg);
         hibyte = read_tim1ac1(3,tim_1a_data_reg);
         count = (int) (256*hibyte) + (int) lobyte;
         if(v_drs < 512 ) count = -count; /* Account for Direction of Rot. */
         break;

      case BOW_VERTICAL:
         write_tim1a(2,tim_1a_control_reg,17); /* Sel Cntr 1 HOLD Reg. Card 2 */
         lobyte = read_tim1ac1(2,tim_1a_data_reg);
         hibyte = read_tim1ac1(2,tim_1a_data_reg);
         count = (int) (256*hibyte) + (int) lobyte;
         if(v_dbvt < 512 ) count = -count; /* Account for Direction of Rot. */
         break;

      case STERN_VERTICAL:
         write_tim1a(2,tim_1a_control_reg,18); /* Sel Cntr 2 HOLD Reg. Card 2 */
         lobyte = read_tim1ac1(2,tim_1a_data_reg);
         hibyte = read_tim1ac1(2,tim_1a_data_reg);
         count = (int) (256*hibyte) + (int) lobyte;
         if(v_dblt < 512 ) count = -count; /* Account for Direction of Rot. */
         break;

      case BOW_LATERAL:
         write_tim1a(2,tim_1a_control_reg,19); /* Sel Cntr 3 HOLD Reg. Card 2 */
         lobyte = read_tim1ac1(2,tim_1a_data_reg);
```

```
        hibyte = read_tim1ac1(2,tim_1a_data_reg);
        count = (int) (256*hibyte) + (int) lobyte;
        if(v_dsvt < 512 ) count = -count; /* Account for Direction of Rot. */
        break;

     case STERN_LATERAL:
        write_tim1a(2,tim_1a_control_reg,20); /* Sel Cntr 4 HOLD Reg. Card 2 */
        lobyte = read_tim1ac1(2,tim_1a_data_reg);
        hibyte = read_tim1ac1(2,tim_1a_data_reg);
        count = (int) (256*hibyte) + (int) lobyte;
        if(v_dslt < 512 ) count = -count; /* Account for Direction of Rot. */
        break;

     default:
        if (DISPLAYSCREEN)
        printf ("[read_motor () error: illegal motor value (%d)]\n", motor);
        break;
  }

  if(count != 0)
  {
     freq = (1.0/count)*4.0*pow(10.0,6.0); /* F1 (1 Mhz) The 4.0 is in there */
                                           /* as a scale factor from God     */
  }
  else
  {
     /* Sensor Not Counting */
     freq = 0.0;
  }

  /* 500 Counts Per Rev */
  rps = (freq/500.0);
  if((fabs(rps) < 1.0) || (fabs(rps) > 1000.0)) rps = 0.0;

  if (TRACE && DISPLAYSCREEN)
      printf ("[finish read_motor () returns %5.3f]\n", rps);

  return (rps * 60.0); /* convert from per-seconds to per-minutes */
 }
 else /* LOCATIONLAB == TRUE */

      return (rpm);

}  /* end read_motor () */

/*****************************************************************************/

/* ---     VERIFIED MATCH CURRENT/OPERATIONAL MARCO AUV HARDWARE/SOFTWARE --- */

double read_roll_angle ()    /* Return roll angle in DEGREES                  */
{
     int    val;
     double angle;

     if (TRACE && DISPLAYSCREEN) printf ("[start  read_roll_angle ()]\n");

      if (LOCATIONLAB)
     {
         angle = phi; /* no change, use virtual world value */
         if (fabs (angle) < 0.0001) angle = 0.0;
     }
     else /* in-water */
     {
         val = get_adc2 (ROLL_ANGLE_CH,0);
         /* Next three lines from old method */
         /*val = val >> 2;*/ /* Quick fix for new res */
         /* angle = ((516.578 - val)/5.7572)/57.295779;  convert to radians */
         /*angle = (-.1737*val + .1737*roll_0)/57.295779;*/
         angle = 0.043425*(roll_0 - val)/57.295779;
         if (fabs (angle) < 0.0001) angle = 0.0;
     }
```

```
/*
if (ARCHAIC_IGNORE)
{
     val    = get_adc1 (ROLL_ANGLE_CH);
/*   angle = ((516.578 - val)/5.7572)/57.295779;  convert to radians */
/*     angle = (-.1737*val + .1737*roll_0)/57.295779;
}
*/
  angle = normalize2 (angle);

  if (TRACE && DISPLAYSCREEN)
      printf ("[finish read_roll_angle () returns %5.3f]\n", angle);

  return (angle);
}

/*****************************************************************************/

/* ---     VERIFIED MATCH CURRENT/OPERATIONAL MARCO AUV HARDWARE/SOFTWARE --- */

double read_pitch_angle ()     /* Return pitch angle in DEGREES                */
{
     int    val;
     double angle;

     if (TRACE && DISPLAYSCREEN) printf ("[start  read_pitch_angle ()]\n");

      if (LOCATIONLAB)
     {
         angle = theta; /* no change, use virtual world value */
         if (fabs (angle) < 0.0001) angle = 0.0;
     }
     else /* in-water */
     {
         val = get_adc2 (PITCH_ANGLE_CH,0);
         /* Next three lines from old method */
         /*val = val >> 2;*/ /* Quick fix for new res */
         /* angle = ((520.153 - val)/8.340)/57.295779;  convert to radians */
         /*angle = ((-.1199*val + .1199*pitch_0)/57.295779);*/
         angle = degrees (0.02997*(pitch_0 - val)/57.295779);
         if (fabs (angle) < 0.0001) angle = 0.0;
     }

/*
if (ARCHAIC_IGNORE)
{
     val = get_adc1 (PITCH_ANGLE_CH);
/*   angle = ((520.153 - val)/8.340)/57.295779;  convert to radians */
/*     angle = -((-.1199*val + .1199*pitch_0)/57.295779);
}
*/
  angle = normalize2 (angle);

  if (TRACE && DISPLAYSCREEN)
      printf ("[finish read_pitch_angle () returns %5.3f]\n", angle);

  return (angle);
}

/*****************************************************************************/

/* ---     VERIFIED MATCH CURRENT/OPERATIONAL MARCO AUV HARDWARE/SOFTWARE --- */

double read_heading ()

  /* Return heading angle with respect to local magnetic north in radians
       from directional gyro */
{
     unsigned short dg_bit;
     double         angle;

     if (TRACE && DISPLAYSCREEN) printf ("[start  read_heading ()]\n");
```

```
    if      (LOCATIONLAB && (DEADRECKON == FALSE))
    {
        angle = psi;
        if (fabs (angle) < 0.0001) angle = 0.0;
    }
    else if (LOCATIONLAB && (DEADRECKON))
    {
        angle = psi_command;
        if (fabs (angle) < 0.0001) angle = 0.0;
    }
    else /* in-water */
    {
        /*dg_bit = Read_PortAB(MFI_BASE);*/
        dg_bit = Read_PortAB(0xFFF00700);  /* why not a #define here? <<<< */
        /*dg_bit = 10000;*/
        dg_bit &= 0x3FFF;

        angle = (3.8350e-4)*((double) dg_bit);
        /*printf("Angle = %f %d\n",angle,dg_bit);*/
        /*if(fabs(angle) < 0.001) angle = 0.0;*/
    }

/*
if (ARCHAIC_IGNORE)
{
        dg_bit = Read_PortAB(MFI_BASE);
        dg_bit &= 0x3FFF;
        angle = (3.8350e-4) * dg_bit;
}
*/

    angle = normalize (angle);

    if (TRACE && DISPLAYSCREEN)
        printf ("[finish read_heading () returns %5.3f]\n", angle);

    return (angle);
}

/*****************************************************************************/

/* --- NOT YET UPDATED TO CURRENT/OPERATIONAL MARCO AUV HARDWARE/SOFTWARE --- */

double read_speed ()  /* Filter the speed signal                              */
{
   static int old_count1,old_count2;
   static int start = TRUE;
   int count;
   unsigned char lobyte,hibyte;
   double freq;
   double avg_speed;

     if (TRACE && DISPLAYSCREEN)
         printf ("[start  read_speed (), LOCATIONLAB=%d]\n", LOCATIONLAB);

      if (LOCATIONLAB)
     {
         if (TRACE && DISPLAYSCREEN)
            printf ("[finish read_speed () returns %5.3f]\n", speed);

         return (speed); /* from virtual world-paddlewheel speed = u = surge */
     }
     else if (DEADRECKON)
     {
         if (TRACE && DISPLAYSCREEN)
            printf ("[finish read_speed () DEADRECKON returns ");
         avg_speed = (speed_per_rpm * (port_rpm + stbd_rpm) / 2.0);
         if (TRACE && DISPLAYSCREEN)
            printf ("%5.3f]\n", avg_speed);

         return (avg_speed);
     }
```

```
    else
    {   /* I think this is Dave's speed averaging code */
       if(start)
       {
         old_count1 = 0;
         old_count2 = 0;
         start = FALSE;
       }

       write_tim1a(3,tim_1a_control_reg,19);
       lobyte = read_tim1ac1(3,tim_1a_data_reg);
       hibyte = read_tim1ac1(3,tim_1a_data_reg);
       count = (int) (256*hibyte) + (int) lobyte;

       if((old_count1 == count) &&
          (old_count2 == count))
       {
          old_count1 = old_count2;
          old_count2 = count;
          return(0.0);
       }

       old_count1 = old_count2;
       old_count2 = count;

    }

    if (TRACE && DISPLAYSCREEN)
        printf ("[finish read_speed () returns %5.3f]\n", avg_speed);

    if(count != 0)
    {
      freq = (1.0/(2.0*count))* 4.0 *10000.0; /* F3 (10,000 Hz) */
    }
    else
    {
       /* Sensor Not Counting */
       freq = 0.0;
    }

  /* Polyfit for Calibration data in marco:/vault2/marco/AUV/turbo_probe/tp.m */

  if(freq >= 4999.0)
  {
     return(fabs(speed));
  }
  else
  {
    return(0.00000973701619*freq*freq + 0.02934498907499*freq + 0.15845400316984);
  }
}
/*****************************************************************************/

/* ---     VERIFIED MATCH CURRENT/OPERATIONAL MARCO AUV HARDWARE/SOFTWARE --- */

void zero_gyro_data ()
{
     int index, val;
     int save_trace = TRACE;       /* save current TRACE value, restore later */

     if (TRACE && DISPLAYSCREEN) printf ("[start  zero_gyro_data ()]\n");

     /* Marco code has a mode variable for gyro on/off, we assume always on   */

     if (fabs (dg_offset) < 0.001) dg_offset = 0.0;
     /*z_val0      = adc1(DEPTH_CELL_CH);
     yaw_rate_0    = get_adc1(YAW_RATE_CH);*/

     z_val0        = get_adc2(DEPTH_CELL_CH,0);
     pitch_0       = get_adc2(PITCH_ANGLE_CH,0);
     roll_0        = get_adc2(ROLL_ANGLE_CH,0);
     dg_offset     = read_heading ();
```

```
roll_rate_0  = get_adc2(ROLL_RATE_CH,0);
pitch_rate_0 = get_adc2(PITCH_RATE_CH,0);
yaw_rate_0   = get_adc2(YAW_RATE_CH,0);

for (i=0;i<9;++i)
{
    /*z_val0      += adc1(DEPTH_CELL_CH);
    yaw_rate_0   += get_adc1(YAW_RATE_CH);*/

    pitch_0      += get_adc2(11,0);
    roll_0       += get_adc2(12,0);
    roll_rate_0  += get_adc2(9,0);
    pitch_rate_0 += get_adc2(8,0);
    yaw_rate_0   += get_adc2(YAW_RATE_CH,0);
    dg_offset    += read_heading();
    z_val0       += get_adc2(DEPTH_CELL_CH,0);

    tsleep(5);
}

dg_offset    = dg_offset/10.0;
z_val0       = z_val0/10;
pitch_0      = pitch_0/10;
roll_0       = roll_0/10;
roll_rate_0  = roll_rate_0/10;
pitch_rate_0 = pitch_rate_0/10;
yaw_rate_0   = yaw_rate_0/10;

/*psi_bit_old = Read_PortAB(MFI_BASE);*/
psi_bit_old = Read_PortAB(0xFFF00700);
psi_bit_old &= 0x3FFF;

if (TRACE && DISPLAYSCREEN)
{
   printf ("roll_0       = %d\n", roll_0);
   printf ("roll_rate_0  = %d\n", roll_rate_0);
   printf ("pitch_0      = %d\n", pitch_0);
   printf ("pitch_rate_0 = %d\n", pitch_rate_0);
   printf ("yaw_rate_0   = %d\n", yaw_rate_0);
   printf ("z_val0       = %d\n", z_val0);
   printf ("dg_offset    = %f\n", dg_offset);
}


if (ARCHAIC_IGNORE)
{
     pitch_0      = get_adc1(6);
     roll_0       = get_adc1(7);
     roll_rate_0  = get_adc1(9);
     pitch_rate_0 = get_adc1(8);
     yaw_rate_0   = get_adc1(10);
     z_val0       = get_adc2(0,0);
     dg_offset    = read_heading ();

     if (TRACE && DISPLAYSCREEN)
                  printf ("[device averaging for 2 seconds... ]\n");
     for (index=0;index<99;++index)
     {
        pitch_0      += get_adc1(6);
        roll_0       += get_adc1(7);
        roll_rate_0  += get_adc1(9);
        pitch_rate_0 += get_adc1(8);
        yaw_rate_0   += get_adc1(10);
        z_val0       += get_adc2(0,0);

        TRACE         = FALSE;
        dg_offset    += read_heading (); /* this is verbose if TRACEd */
        TRACE         = save_trace;     /* so meanwhile turn off TRACE */
        tsleep (5);                     /* 256ths of a second          */
     }

     pitch_0      = pitch_0/100;
```

```
     roll_0        = roll_0/100;
     roll_rate_0   = roll_rate_0/100;
     pitch_rate_0  = pitch_rate_0/100;
     yaw_rate_0    = yaw_rate_0/100;
     z_val0        = z_val0/100;
     dg_offset     = dg_offset/100.0;

     if (TRACE && DISPLAYSCREEN)
     {
        printf ("pitch_0       = %d\n", pitch_0);
        printf ("roll_0        = %d\n", roll_0);
        printf ("roll_rate_0   = %d\n", roll_rate_0);
        printf ("pitch_rate_0  = %d\n", pitch_rate_0);
        printf ("yaw_rate_0    = %d\n", yaw_rate_0);
        printf ("z_val0        = %d\n", z_val0);
        printf ("dg_offset     = %f\n", dg_offset);
     }
}
     if (TRACE && DISPLAYSCREEN) printf ("[finish zero_gyro_data ()]\n");

     return;

}  /* end zero_gyro_data () */

/*****************************************************************************/

/* ---     VERIFIED MATCH CURRENT/OPERATIONAL MARCO AUV HARDWARE/SOFTWARE --- */

void zero_surfaces ()              /* Initialize all planes & rudders to zero */
{
     if (TRACE && DISPLAYSCREEN)
         printf ("[start  initialize_dacs_zero_actuators ()]\n");

     command_rudder (0.0);
     command_planes (0.0);

/*
if (ARCHAIC_IGNORE)
{
     command_control_surface (BOW_RUDDER_TOP,      0.0);
     command_control_surface (BOW_RUDDER_BOTTOM,   0.0);
     command_control_surface (BOW_PLANE_STBD,      0.0);
     command_control_surface (BOW_PLANE_PORT,      0.0);
     command_control_surface (STERN_RUDDER_TOP,    0.0);
     command_control_surface (STERN_RUDDER_BOTTOM,0.0);
     command_control_surface (STERN_PLANE_STBD,    0.0);
     command_control_surface (STERN_PLANE_PORT,    0.0);

     command_propellors_off ();
}
*/
     if (TRACE && DISPLAYSCREEN)
         printf ("[finish zero_surfaces ()]\n");

     return;

} /* end zero_surfaces () */

/*****************************************************************************/

/* ---     VERIFIED MATCH CURRENT/OPERATIONAL MARCO AUV HARDWARE/SOFTWARE --- */

void initialize_adcs () /* reconsider casts <<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<< */
{                       /* is this code even needed <<<<<<<<<<<<<<<<<<<<<<<<<<<< */
     int i,j,val;

     if (LOCATIONLAB)
     {
        return;
     }
     if (TRACE && DISPLAYSCREEN) printf ("[start  initialize_adcs ()]\n");
```

```
  /* Init_PortA(MFI_BASE,0);
     Init_PortB(MFI_BASE,0);  */

     Init_PortA(0xFFF00700,0);  /* appears to be archaic <<<<<<<<<< */
     Init_PortB(0xFFF00700,0);

     for (i=0;i<4;++i)      /* Stabilize ada-1 card by reading it a few times */
     {
        for(j=0;j<8;++j)
        {
           val = get_adc1 (j);
        }
     }
     if (TRACE && DISPLAYSCREEN) printf ("[finish initialize_adcs ()]\n");

/*
if (ARCHAIC_IGNORE)
{
   if (LOCATIONLAB == FALSE) /* not in virtual world, read slots */
/*   {
     /* Initialize MFI channels:  0 = input port, 1 = output port */

/*     Init_PortA ((struct MFI_PIA *) MFI_BASE, MFI_INPUT_PORT);
     Init_PortB ((struct MFI_PIA *) MFI_BASE, MFI_INPUT_PORT);
   }
}
*/
     return;

} /* end initialize_adcs */

/*******************************************************************************/
/*************************************/
void init_pia ()
{
   if (LOCATIONLAB)
   {
      return;
   }
   via0[ORA_IRA] = 0xFF;
   via0[ORB_IRB] = 0xFF;

   via0[DDRA] = 0xFF;  /* Enable VIA0 for Writing */
   via0[DDRB] = 0xFF;

   via0[ORA_IRA] = 0xFF;
   via0[ORB_IRB] = 0xFF;

   via0a_reg = 0xFF;
   via0b_reg = 0xFF;

   via1[DDRA] = 0x00; /* Enable VIA1 for reading */
   via1[DDRB] = 0x00;

   tsleep(100); /* Let Things Cool Off */
}


/* Initialize tim_1a cards, mode = 0 init encoders only, init = 1 encoders */
/*                          and fins                                       */

void init_tim1a(mode)
   int mode;
{
   int i,j;

   if (LOCATIONLAB)
   {
      return;
   }
/*
   counter 1, Card 1 - front rudder top, rear rudder bottom
```

```
   counter 2, Card 1 - front rudder bottom, rear rudder top

   counter 3, Card 1 - front plane left, rear plane right

   counter 4, Card 1 - front plane right, rear plane left
*/
   if(mode)
   {
      /* Init control surface card 1 */
      write_tim1a(1,tim_1a_control_reg,255); /* reset all board functions */
      write_tim1a(1,tim_1a_control_reg,23);  /* select mastermode register */
      write_tim1a(1,tim_1a_data_reg,176);    /* lobyte enables 8 bit,binary, */
                                             /* fout */
      write_tim1a(1,tim_1a_data_reg,65);     /* hibyte enable fout = 1mhz etc */
      write_tim1a(1,tim_1a_control_reg,249); /* disable write prefetch */

      /* for (i=25;i<=28;i++) Use this if new chip installed */
      for (j=9;j<=12;j++)   /* This is done since signal gets inverted */
                            /* Counters 1-4 Only */
      {
         write_tim1a(1,tim_1a_control_reg,j); /* high output time about 8ms */
         write_tim1a(1,tim_1a_data_reg,0);    /* load all hold registers */
         write_tim1a(1,tim_1a_data_reg,150);  /* lobyte = 0 hibyte = 155 for */
                                              /* 1 mhz */
      }

      for (j=1;j<=4;j++)  /* Counters 1-4 Only */
      {
         write_tim1a(1,tim_1a_control_reg,j); /* program all counter mode */
                                              /* registers see mode j */
         write_tim1a(1,tim_1a_data_reg,98);   /* lobyte = reload from load */
                                              /* or hold, count repeat */
         write_tim1a(1,tim_1a_data_reg,27);   /* higyte = nogate,count on */
                                              /* falling edge 1mhz */
      }
   } /* End if(mode) */

   /* Init speed sensor cards 2 & 3 */

   /*
      counter 1, Card 2 - BOW VERTICAL THRUSTER SPEED

      counter 2, Card 2 - BOW LATERAL THRUSTER SPEED

      counter 3, Card 2 - STERN VERTICAL THRUSTER SPEED

      counter 4, Card 2 - STERN LATERAL THRUSTER SPEED

      counter 1, Card 3 - LEFT SCREW SPEED

      counter 2, Card 3 - RIGHT SCREW SPEED

      counter 3, Card 3 - TURBO PROBE SPEED
   */

   for(i=2;i<=3;++i)  /* Program Master Mode Reg. for Cards 2 & 3 */
   {
      write_tim1a(i,tim_1a_control_reg,0xff); /* Reset All Board Functions */
      write_tim1a(i,tim_1a_control_reg,0x17); /* Select Master Mode Reg. */
      write_tim1a(i,tim_1a_data_reg,0xb0);
      write_tim1a(i,tim_1a_data_reg,0xc1);
   }

   for(j=1;j<=4;++j)     /* Program Counters 1-4, Card 2 */
   {
      write_tim1a(2,tim_1a_control_reg,j);
      write_tim1a(2,tim_1a_data_reg,0xaa);
      write_tim1a(2,tim_1a_data_reg,203); /* Set for F1 (1Mhz) */
   }

   for(j=9;j<=12;++j)  /* Set LOAD Reg 1-4 to Zero, Card 2 */
   {
```

```
      write_tim1a(2,tim_1a_control_reg,j);
      write_tim1a(2,tim_1a_data_reg,0x00);
      write_tim1a(2,tim_1a_data_reg,0x00);
   }

   write_tim1a(2,tim_1a_control_reg,0x4f);  /* Load Counters 1-4 Card 2 */
   write_tim1a(2,tim_1a_control_reg,0x2f);  /* Arm  Counters 1-4 Card 2 */

   write_tim1a(2,tim_1a_aux_gates_reg,0xff); /* SET AUX GATES HIGH TO WORK! */

   for(j=1;j<=3;++j)      /* Program Counters 1-3, Card 3 */
   {
      write_tim1a(3,tim_1a_control_reg,j);
      write_tim1a(3,tim_1a_data_reg,0xaa);


      /* F1 = 203 = 0xCB = 1   Mhz
         F2 = 204 = 0xCC = 100 Khz
         F3 = 205 = 0xCD = 10  Khz
         F4 = 206 = 0xCE = 1   Kz
         F5 = 207 = 0xCF = 100 Hz
      */

      if(j==3)
      {
         /* Turbo Probe */
         write_tim1a(3,tim_1a_data_reg,205); /* Set Counter 3 for F (hz) */
      }
      else
      {
         write_tim1a(3,tim_1a_data_reg,203); /*Set Counters 1-2 for F1 (1Mhz)*/
      }
   }

   for(j=9;j<=11;++j)  /* Set LOAD Reg 1-3 to Zero, Card 3 */
   {
      write_tim1a(3,tim_1a_control_reg,j);
      write_tim1a(3,tim_1a_data_reg,0x00);
      write_tim1a(3,tim_1a_data_reg,0x00);
   }

   write_tim1a(3,tim_1a_control_reg,0x47);  /* Load Counters 1-3 Card 3 */
   write_tim1a(3,tim_1a_control_reg,0x27);  /* Arm  Counters 1-3 Card 3 */

   write_tim1a(3,tim_1a_aux_gates_reg,0xff); /* SET AUX GATES HIGH TO WORK!  */
}


void thruster_power(onoff)
/* A signal inverter has been placed between the pia card and the power
   supplies for the thrusters, so in order to turn them on, bits for these
   must be set low */

   int onoff;
{
   if (LOCATIONLAB)
   {
      return;
   }
   switch(onoff)
   {
      case 0:  /* TURN OFF */
         via0a_reg = via0a_reg | 0x3C; /* Set bits PA2-PA5 High retaining */
                                       /* other bits */
         via0[ORA_IRA] = via0a_reg;
         break;

      case 1:
         via0a_reg = via0a_reg & 0xC3; /* Set bits PA2-PA5  Low retaining */
                                       /* other bits */
         via0[ORA_IRA] = via0a_reg;
         break;
```

```
   }
}

void screw_power(onoff)
/* A signal inverter has been placed between the pia card and the power
   supplies for the thrusters, so in order to turn them on, bits for these
   must be set low */

   int onoff;
{
   if (LOCATIONLAB)
   {
      return;
   }
   switch(onoff)
   {
      case 0:  /* TURN OFF */
         via0a_reg = via0a_reg | 0x03; /* Set bits PA0-PA1 High retaining */
                                       /* other bits */
         via0[ORA_IRA] = via0a_reg;
         break;

      case 1:
         via0a_reg = via0a_reg & 0xFC; /* Set bits PA0-PA1 Low retaining */
                                       /* other bits */
         via0[ORA_IRA] = via0a_reg;
         break;
   }
}


/*************************************/
/*************************************/
/*************************************/


/* ---     VERIFIED MATCH CURRENT/OPERATIONAL MARCO AUV HARDWARE/SOFTWARE --- */

void command_control_surface (angle, surface)

                        double angle;

                        int           surface;

{
  /*  This function sends the desired ANGLE to the specified control SURFACE
      The angle is first normalized to (-45 to 45 degrees), then correction is
      applied for the nonlinearity in the servo control module                */

  /*  Connections are:
          pin 1  = control
          pin 2  = ground
          pin 3  = 5 volts
   useful pulse widths are 600 to 2500 ms
   this program ouputs positive going pulses with a 8 ms delay between pulses
   this program is set up for a 1 MHz board                                  */

     int skip_pulse;
     int pulse,hipulse,lopulse,n,m,dis,larm;
     unsigned char card;

     int volt;           /* archaic */
     double a,b,c,d;     /* archaic */

     int old_pulse1 = -1; /* Init Old Pulses for cont. surfrace servos */
     int old_pulse2 = -1;
     int old_pulse3 = -1;
     int old_pulse4 = -1;

     if (FALSE && DISPLAYSCREEN)
        printf ("[start  command_control_surface ()]\n");
```

```
 if (LOCATIONLAB)
{
    return; /* no action required in virtual world                        */
}

/* pulse = 39.32*angle + 5171;*/ /* angle (deg)*/
pulse = ((int) 2252.87*angle) + 5171; /* Calib for Vehicle Servo angle
                                         (rad) */
hipulse = pulse/256;
lopulse = pulse - (hipulse*256);

skip_pulse = FALSE;

switch(surface)
{
case 1:
   n = 25;
   m = 233;
   dis = 0xC1;
   larm = 0x61;

   if(pulse == old_pulse1) skip_pulse = TRUE;
   old_pulse1 = pulse;
   break;

case 2:
   n = 26;
   m = 234;
   dis = 0xC2;
   larm = 0x62;

   if(pulse == old_pulse2) skip_pulse = TRUE;
   old_pulse2 = pulse;
   break;

case 3:
   n = 27;
   m = 235;
   dis = 0xC4;
   larm = 0x64;

   if(pulse == old_pulse3) skip_pulse = TRUE;
   old_pulse3 = pulse;
   break;

case 4:
   n = 28;
   m = 236;
   dis = 0xC8;
   larm = 0x68;

   if(pulse == old_pulse4) skip_pulse = TRUE;
   old_pulse4 = pulse;
   break;

default:
   printf("Invalid surface code\n");
   break;
}
if(!skip_pulse) /* SKIP resetting of freq out if command angle has not*/
               /* changed. Otherwise servo will chatter at frequency   */
               /* of control loop when command angle does not change   */
{
 write_tim1a(1,tim_1a_control_reg,dis);
 write_tim1a(1,tim_1a_control_reg,n);
 write_tim1a(1,tim_1a_data_reg,lopulse);
 write_tim1a(1,tim_1a_data_reg,hipulse);
 write_tim1a(1,tim_1a_control_reg,m);
 write_tim1a(1,tim_1a_control_reg,larm);
}

/*
```

```
if (ARCHAIC_IGNORE)
{
     a =  1.2487e-4;
     b = -2.9087e-2;
     c =  5.0927;
     d = 500.6576;

     angle = angle*57.295779; /* Convert RADIANS to DEGREES */

/*     if ((angle < -22.92) || (angle > 22.92))
     {
        /* Plane saturated set to +- 45 */
/*        angle = 22.92*angle/fabs(angle);

     }

     volt = a*pow(angle,3.) + b*pow(angle,2.) + c*angle + d;

     send_dac2b (volt,surface);
}
*/
     if (FALSE && DISPLAYSCREEN)
         printf ("[finish command_control_surface ()]\n");

     return;

} /* command_control_surface () */

/*****************************************************************************/

/* ---     VERIFIED MATCH CURRENT/OPERATIONAL MARCO AUV HARDWARE/SOFTWARE --- */

void command_rudder (angle)

  /* Send angular deflection (DEGREES) to rudders.
     Convention (+) angle forward rudder => auv right turn,
                (-) angle forward rudder => auv left turn                    */

     double angle;
{
     if (TRACE && DISPLAYSCREEN) printf ("[start  command_rudder ()]\n");

     /* top/bottom surfaces are slaved due to inadequate DAC card channels    */

     /* positive forward rudder angle pushes bow to right => positive psi rate*/

     angle = radians (angle); /* convert degrees to radians */

     command_control_surface ( angle, BOW_RUDDER_TOP     );
  /* command_control_surface ( angle, BOW_RUDDER_BOTTOM  ); hardware error */
     command_control_surface (-angle, STERN_RUDDER_TOP   );
  /* command_control_surface (-angle, STERN_RUDDER_BOTTOM); hardware error */

/*
if (ARCHAIC_IGNORE)
{
     command_control_surface (BOW_RUDDER_TOP,     -angle);
     command_control_surface (BOW_RUDDER_BOTTOM,   angle);
     command_control_surface (STERN_RUDDER_TOP,    angle);
     command_control_surface (STERN_RUDDER_BOTTOM,-angle);
}
*/
     if (TRACE && DISPLAYSCREEN) printf ("[finish command_rudder ()]\n");

     return;

} /* end command_rudder () */

/*****************************************************************************/

/* ---     VERIFIED MATCH CURRENT/OPERATIONAL MARCO AUV HARDWARE/SOFTWARE --- */
```

```
void command_planes (angle)

  /* Send angular deflection (RADIANS) to bow and stern planes.  Convention:
     (-) bow plane angle => auv dive, (+) bow plane angle => auv rise
     where auv dive = positive depth rate, auv rise = negative depth rate     */

     double angle;
{
     if (TRACE && DISPLAYSCREEN) printf ("[start  command_planes ()]\n");

     /* left/right surfaces are slaved due to inadequate DAC card channels    */

     /* positive planes angle pushes bow up, yields negative depth rate       */

     angle = radians (angle);

     command_control_surface ( angle, BOW_PLANE_STBD  );
  /* command_control_surface (-angle, BOW_PLANE_PORT  ); combined  stern stbd */
     command_control_surface (-angle, STERN_PLANE_STBD);
  /* command_control_surface ( angle, STERN_PLANE_PORT); combined bow stbd    */

/*
if (ARCHAIC_IGNORE)
{
     command_control_surface (BOW_PLANE_STBD,    angle);
     command_control_surface (BOW_PLANE_PORT,   -angle);
     command_control_surface (STERN_PLANE_STBD, -angle);
     command_control_surface (STERN_PLANE_PORT,  angle);
}
*/
     if (TRACE && DISPLAYSCREEN) printf ("[finish command_planes ()]\n");

     return;

} /* end command_planes () */

/*****************************************************************************/

/* ---     VERIFIED MATCH CURRENT/OPERATIONAL MARCO AUV HARDWARE/SOFTWARE --- */

void command_propellors_off ()               /* Turn off both main propellors */
{
     if (TRACE && DISPLAYSCREEN)
         printf ("[start  command_propellors_off ()]\n");

     command_motor (0.0, PORT_PROP);
     command_motor (0.0, STBD_PROP);

/*
if (ARCHAIC_IGNORE)
{
     send_dac1(512, SUPPLY);
     send_dac1(512, RIGHT_MOTOR);
     send_dac1(512, LEFT_MOTOR);
}
*/
     if (TRACE && DISPLAYSCREEN)
         printf ("[finish command_propellors_off ()]\n");

     return;

} /* end command_propellors_off () */

/*****************************************************************************/

/* ---     VERIFIED MATCH CURRENT/OPERATIONAL MARCO AUV HARDWARE/SOFTWARE --- */

void command_thrusters_off ()                /* Turn off both main propellers */
{
     if (TRACE && DISPLAYSCREEN)
         printf ("[start  command_thrusters_off ()]\n");
```

```
     command_motor (0.0, BOW_VERTICAL);
     command_motor (0.0, STERN_VERTICAL);
     command_motor (0.0, BOW_LATERAL);
     command_motor (0.0, STERN_LATERAL);

/*
if (ARCHAIC_IGNORE)
{
     send_dac1(512, SUPPLY);
     send_dac1(512, RIGHT_MOTOR);
     send_dac1(512, LEFT_MOTOR);
}
*/
     if (TRACE && DISPLAYSCREEN)
         printf ("[finish command_thrusters_off ()]\n");

     return;

} /* end command_thrusters_off () */

/*****************************************************************************/

/* ---     VERIFIED MATCH CURRENT/OPERATIONAL MARCO AUV HARDWARE/SOFTWARE --- */

void command_motor (order,     motor)
             double order; int motor;
{
/*      motor = 0  Left  Propeller              PORT_PROP         RPM
                1  Right Propeller             STBD_PROP         RPM
                2    Bow Vertical Thruster       BOW_VERTICAL     volts
                3    Bow Lateral  Thruster     STERN_VERTICAL     volts
                4  Stern Vertical Thruster       BOW_LATERAL      volts
                5  Stern Lateral  Thruster     STERN_LATERAL      volts
*/
     /* use local variables to permit clamping without side effects          */

     int    dac_value      = 0;      /* range         0..1023                 */
     double propellor_rpm  = order;  /* propellors -700.. 700 rpm             */
     double thruster_volts = order;  /* thrusters   -24..  24 volts           */

     if (TRACE && DISPLAYSCREEN)
         printf ("[start  command_motor ()]\n");

     if ((motor == PORT_PROP) || (motor == STBD_PROP))
     {
        clamp (&propellor_rpm, -700.0, 700.0, "command_motor (): propellor_rpm");

        if (motor == PORT_PROP)
            dac_value = port_speed_control (propellor_rpm / 60.0);
        if (motor == STBD_PROP)
            dac_value = stbd_speed_control (propellor_rpm / 60.0);

        if (TRACE && DISPLAYSCREEN)
        {
            if      (motor == PORT_PROP) printf ("[PORT ");
            else if (motor == STBD_PROP) printf ("[STBD ");
            printf ("propellor_rpm = %5.1f, dac_value = %d]\n",
                     propellor_rpm,         dac_value);
        }
     }
     else if (    (motor == BOW_VERTICAL) || (motor == STERN_VERTICAL)
              || (motor == BOW_LATERAL)   || (motor == STERN_LATERAL ))
     {
        clamp (&thruster_volts,-24.0,24.0,"command_motors (): thruster_volts");

        dac_value = (int) ((thruster_volts + 24.0) * 1023.0 / 48.0);

        if (TRACE && DISPLAYSCREEN)
            printf ("[thruster_volts = %5.1f, dac_value = %d]\n",
                      thruster_volts,         dac_value);
     }
     else /* erroneous motor number selected */
```

```
     {
        if (TRACE && DISPLAYSCREEN)
            printf ("[command_motor (): erroneous order/motor (%5.1f/%d)]\n",
                                                order,motor);
        return;
     }

     send_dac2b (dac_value, motor);

/*
if (ARCHAIC_IGNORE)
{
     send_dac1(512, SUPPLY);
     send_dac1(512, RIGHT_MOTOR);
     send_dac1(512, LEFT_MOTOR);
}
*/
     if (TRACE && DISPLAYSCREEN)
         printf ("[finish command_motor ()]\n");

     return;

} /* end command_motor () */

/*****************************************************************************/

/* ---     VERIFIED MATCH CURRENT/OPERATIONAL MARCO AUV HARDWARE/SOFTWARE --- */

void test_alive (interval, local_start_dwell)  /* no longer used by Dave? <<< */

     unsigned int interval;
              int local_start_dwell;
{
     unsigned int iinterval,jinterval;
     double test_delta;

     if (TRACE && DISPLAYSCREEN) printf ("[start  test_alive ()]\n");

     local_start_dwell = local_start_dwell*100;
     interval = interval*100;
     iinterval = local_start_dwell/interval;
     jinterval = 0;
     test_delta = .4; /* Deflect 22.5 degrees */

     while(jinterval < iinterval)
     {
        command_control_surface (BOW_RUDDER_TOP, test_delta);
        tsleep(interval);    /* 256ths of a second */
        test_delta = -test_delta;
        jinterval = jinterval + 1;
     }

     tsleep(200);            /* 256ths of a second */

     if (TRACE && DISPLAYSCREEN) printf ("[finish test_alive ()]\n");

     return;
}

/*****************************************************************************/

/* --- NOT YET UPDATED TO CURRENT/OPERATIONAL MARCO AUV HARDWARE/SOFTWARE --- */

void get_init_avg ()  /* sonar ??? better name needed !!!! <<<< */
{
  int index, rng_sum;

  if (TRACE && DISPLAYSCREEN) printf ("[start  get_init_avg ()]\n");

  rng_sum     = 0;
  range_index = 0;
```

```
  for(index = 0; index < AVG_PTS; ++index)
  {
    via0[ORB_IRB] = (SONAR_SW1 & SONAR_SW3) | SONAR_TRIG2;
    via0[ORB_IRB] = SONAR_SW1 & SONAR_SW3;
    tsleep(5);
    range = get_adc2 (3,0);

    rng_sum += range;
    range_array[index] = range;
    ++range_index;

  }
  avg_rng = (rng_sum/AVG_PTS) * 1.0;

  if (TRACE && DISPLAYSCREEN) printf ("[finish get_init_avg ()]\n");

  return;
}

/*****************************************************************************/

/* --- NOT YET UPDATED TO CURRENT/OPERATIONAL MARCO AUV HARDWARE/SOFTWARE --- */

void get_avg_rng ()
{
   int index, UPDATE_AVG, int_rng_sum;

   if (TRACE && DISPLAYSCREEN) printf ("[start  get_avg_rng ()]\n");

   UPDATE_AVG  = 0;
   int_rng_sum = 0;

   if (((double)range > avg_rng ) ||
       (fabs((double)range - avg_rng) <= MAX_RNG_DIFF) ||
       (bad_rng >= MAX_BAD_PTS))
          {
            range_array[range_index] = range;
            ++range_index;
            UPDATE_AVG = 1;
            if(bad_rng > MAX_BAD_PTS)
              {
                ++bad_updates;
              }
            if(bad_updates >= MIN_NO_PTS)
              {
                bad_rng = 0;
              }
          }
        else
          {
            ++bad_rng;
          }

  if(UPDATE_AVG)
    {
      for(index = range_index - AVG_PTS; index <= range_index; ++index)
        {
          int_rng_sum += range_array[index];
        }

      avg_rng = int_rng_sum/AVG_PTS * 1.0;
    }
    if (TRACE && DISPLAYSCREEN) printf ("[finish get_avg_rng ()]\n");

    return;
}

/*****************************************************************************/
/*
   Lab hardware control changes *FOLLOWING* hardware upgrade 1993:

   Telemetry to tactical level:       serial port /T1 via driver /TT
```

```
   Orders from  tactical level:       parallel port /P via MFI register A

   Sonar:                             interface card device driver /T3

*/
/*****************************************************************************/

/* --- NOT YET UPDATED TO CURRENT/OPERATIONAL MARCO AUV HARDWARE/SOFTWARE --- */

void open_device_paths ()
{
     if (TRACE && DISPLAYSCREEN) printf ("[start  open_device_paths ()]\n");

#if (defined(sgi) || defined(sun))
#else
     /* either /t1 serial port #1 or /tt (high baud rate driver for /t1)      */
     serialpath = open ("/t1", S_IREAD + S_IWRITE);        /* get path number */
     /*                  /tt is device for high baud rate /t1 serial port     */
     if (serialpath <= 0)
     {
        printf ("open_device_paths ():  unable to open serialpath /t1.  ");
        printf ("Exit.\n");
        exit (-1);
     }
     if (TRACE && DISPLAYSCREEN)
        printf ("[serialpath /t1 (normal baud rate) open, path number = %d]\n",
                                                                  serialpath);

     if (SONARINSTALLED)
     {
        sonarpath  = open ("/t3", S_IREAD + S_IWRITE);     /* get path number */
        /*                  /t3 is device for sonar interface card            */
        if (sonarpath <= 0)
        {
           printf ("open_device_paths ():  unable to open sonarpath /t3.  ");
           printf ("Exit.\n");
           exit (-1);
        }
        if (TRACE && DISPLAYSCREEN)
            printf ("[sonarpath  /t3 open, path number = %d]\n", sonarpath);

        tty_mode (sonarpath,1);   /* initialize sonar values */
     }
     else if (TRACE && DISPLAYSCREEN)
             printf ("[sonarpath  /t3 ignored, SONARINSTALLED == FALSE]\n");

     /* other paths:  effectors, depth_sonar, etc. ****************************/

#endif
     if (TRACE && DISPLAYSCREEN) printf ("[finish open_device_paths ()]\n");

     return;
}

/*****************************************************************************/

/* --- NOT YET UPDATED TO CURRENT/OPERATIONAL MARCO AUV HARDWARE/SOFTWARE --- */

void close_device_paths ()
{
     if (TRACE && DISPLAYSCREEN) printf ("[start  close_device_paths ()]\n");

     if (serialpath > 0) close (serialpath);  /* test for open before closing */

     else if (TRACE && DISPLAYSCREEN) printf ("[serialpath was not open!]\n");

     if (SONARINSTALLED)
     {
         if (sonarpath > 0) close (sonarpath);

         else if (TRACE && DISPLAYSCREEN) printf("[sonarpath was not open!]\n");
     }
```

```
     /* other paths:  effectors, depth_sonar, etc. <<<<<<<<<<<<<<<<<<<<<<<<<<< */

     if (TRACE && DISPLAYSCREEN) printf ("[finish close_device_paths ()]\n");

     return;
}
/*****************************************************************************/

/* --- NOT YET UPDATED TO CURRENT/OPERATIONALMARCO AUV HARDWARE/SOFTWARE --- */

/* may be unneeded in next version */

void   read_parallel_port ()   /* loop and display 8 bit data from port A */
{
   static    char  next_char, last_char;
   static    char  current_command [256];
   static    int   index;
   unsigned  char  temp;

return;

   if (TRACE && DISPLAYSCREEN)
   {
       printf ("[start  read_parallel_port (), ");
       if (PARALLELPORTTRACE) printf ("PARALLELPORTTRACE is ON]\n");
       else                   printf ("PARALLELPORTTRACE is OFF]\n");
   }
   /* see initialize_adcs () for Init_PortA & B code                          */

#if (defined(sgi) || defined(sun))
#else
   /* Read PortA parallel port character by character for tactical orders     */
   /* reference:  Walt Landaker's mfi_a3.c in directory /h0/AUV and           */
   /*             page 3-12 of Motorola 6800 Series Manual for 6821 PIA       */
   /*             Programmable Interface Adapter.                             */

   /* Warning!  You may have to reset both computers to get the parallel      */
   /*           port to read & write properly.  Additionally,                 */
   /*           on the 386 you can run PORTFIX to reset parallel port LPT1:   */

   temp = Read_PortA ((struct MFI_PIA *) MFI_BASE);     /* should clear busy! */

   index = 0;

   /* read port status (note sta not stb)                                     */
   PortAFlag = ck_sta ((struct MFI_PIA *) MFI_BASE);

   if (PARALLELPORTTRACE && DISPLAYSCREEN)
       printf ("\n    [time %5.2f read_parallel_port () resumed]", t);

   while (PortAFlag && 0x80)         /* see loop break for alternate exit     */
   {
        /* Note that ck_stb is used in mfi_a3 but ck_sta makes more sense     */
        PortAFlag = ck_sta ((struct MFI_PIA *) MFI_BASE); /* read port status */
        last_char = next_char;       /* read char and reset busy              */
        next_char = Read_PortA((struct MFI_PIA *) MFI_BASE);

        if ((PortAFlag == 0x24) && (last_char == next_char)) break;

        /*  if next_char changed then flag may be messed up, read anyway      */
                                     /* check for ptr strobe                  */
                                     /* break => no character waiting         */
                                     /* control passes outside while loop     */

        else if (next_char == 13)    /* CR indicates end of line              */
        {
             current_command [index]   = 13;  /* CR    /n                     */
             current_command [index+1] = 10;  /* LF    extra, not needed      */
             current_command [index+2] =  0;  /* end of string delimiter      */
             index = 0;

             if (auvtextfile!=NULL)
```

```
            {
                fprintf (auvtextfile, "%s", current_command);
                fflush  (auvtextfile); /* force completion of file write */
            }
            if (DISPLAYSCREEN)
            {
                printf ("\n\n>>> time %5.2f tactical message <<<\n", t);
                printf  ("%s", current_command);
                printf  ("\n");
            }
            /******** insert command processing logic here <<<<<<<<<<<<<<<<< */

        }
        else if (next_char != 10)           /* LF ignored, others appended    */
        {
                current_command [index] = next_char;
                index ++;
                if (PARALLELPORTTRACE && DISPLAYSCREEN)
                {
                   /* print character to screen */
                      printf ("\n %c = %2x, PortAFlag after reading = %4x]  ",
                                  next_char, next_char, PortAFlag);
                }
        }
   } /* end while loop to read characters from port                          */

   if (PARALLELPORTTRACE && DISPLAYSCREEN)
   {
       printf("    [%c = %2x, PortAFlag = %2x, exiting read_parallel_port ()]",
              next_char, next_char, PortAFlag);
   }
#endif

   if (TRACE && DISPLAYSCREEN) printf ("[finish read_parallel_port ()]\n");

   return;

}  /* end read_parallel_port */

/*****************************************************************************/

/* Card arrangement in AUV *PRIOR* to hardware upgrade 1993:

                  -------   -------   -----------------------
                 |       | |       | |          MFI          | <-- Direct
                 |       | |       | |      Reg A and B      |     Gyro
                  -------   -------   -----------------------

                          -----------------------
                         |         adc-2         | <-- Sonars
                         |      ( 0 - 15 )       |
                          -----------------------

                   --------------    -----------------------
  Main motors --> |  ada-1 dac   |  |       ada-1 adc       | <-- Gyros
                  |  ( 0 - 3 )   |  |       ( 0 - 15 )      |
                   --------------    -----------------------

                          -----------------------
                         |        dac-2b         | <-- Planes
                         |      ( 0 - 7 )        |
                          -----------------------


     This file contains the functions which address the A/D D/A cards
     directly in terms of voltages. */

/*****************************************************************************/

/* ---     VERIFIED MATCH CURRENT/OPERATIONAL MARCO AUV HARDWARE/SOFTWARE --- */
```

```
/* this function is full of unreachable statements, ask Dave <<<<<<<<<<<<<<<< */

unsigned char read_tim1ac1 (card, reg)

            unsigned       card;
            unsigned char        reg;
{
   unsigned char data;

   if (TRACE && DISPLAYSCREEN) printf ("[start  read_tim1ac1 ()]\n");

   switch (card)
   {
      case 1:
         return(tim_1ac1[reg]);
         data = tim_1ac1[3];     /* Wait 2uS, one access to address (RTC) */
         break;

      case 2:
         return(tim_1ac2[reg]);
         data = tim_1ac2[3];     /* Wait 2uS, one access to address (RTC) */
         break;

      case 3:
         return(tim_1ac3[reg]);
         data = tim_1ac3[3];     /* Wait 2uS, one access to address (RTC) */
         break;

      default:
         if (DISPLAYSCREEN)
             printf ("[read_tim1ac1 (): invalid card (%d)]\n", card);
         return (0);
  }

  if (TRACE && DISPLAYSCREEN)
      printf ("[finish read_tim1ac1 () = %d]\n", data);

  return (data); /* is this correct ?? */

} /* end read_tim1ac1 () */

/*****************************************************************************/

/* ---     VERIFIED MATCH CURRENT/OPERATIONAL MARCO AUV HARDWARE/SOFTWARE --- */

void write_tim1a (card, reg, value)

   unsigned       card;
   unsigned char        reg, value;
{

   unsigned char data; /* Dummy Data for delay */

   if (LOCATIONLAB) return;   /* avoid bus error writing to restricted memory */

   if (TRACE && DISPLAYSCREEN) printf ("[start  write_tim1a ()]\n");

   switch (card)
   {
      case 1:
         tim_1ac1[reg] = value;
         data = tim_1ac1[3];    /* Wait 2uS, one access address (RTC) */
         break;

      case 2:
         tim_1ac2[reg] = value;
         data = tim_1ac2[3];    /* Wait 2uS, one access address (RTC) */
         break;

      case 3:
         tim_1ac3[reg] = value;
         data = tim_1ac3[3];    /* Wait 2uS, one access address (RTC) */
```

```
        break;

      default:
         if (DISPLAYSCREEN)
         printf ("[write_tim1a () error: illegal card value (%d)]\n", card);
         break;
   }
   if (TRACE && DISPLAYSCREEN) printf ("[finish  write_tim1a ()]\n");

   return;

} /* end write_tim1a () */

/*****************************************************************************/

  /*****************************************************************************
   *  send_dac1(s,ch) -- writes signal 's' to ada-1 dac channel 'ch'
   *                     (allowable channels 0-3)
   *****************************************************************************/

/* --- NOT YET UPDATED TO CURRENT/OPERATIONAL MARCO AUV HARDWARE/SOFTWARE --- */

/* no longer used... */

  void send_dac1 (s,ch)
  int s,ch;
  {
   if (LOCATIONLAB)        /* in virtual world, do not read any slots */
   {
      return;
   }


/*
if (NOT_YET_REIMPLEMENTED)
{

     ch = ch << 2;                          /* offset for G-96 addressing    */
/*     dac1_a[ch] = s >> 2;                   /* write upper 8 bits to MSB     */
/*     dac1_a[ch + DAC_LSB_OFFSET] = s << 6;  /* write lower 2 bits B3,B2      */
/*}*/
     return;
  } /* send_dac1 */

  /*****************************************************************************
   *  send_dac2b (s,ch) -- writes signal 's' to dac2b dac channel 'ch'
   *                     (allowable channels 0-15)
   *****************************************************************************/

/* ---     VERIFIED MATCH CURRENT/OPERATIONAL MARCO AUV HARDWARE/SOFTWARE --- */

 /*******************************************************
   *  dac2b(s,ch) -- writes signal 's' to dac2b dac channel 'ch'
   *                 (allowable channels 0-7)
   *  s = 0    --> - 10 Vdc Output
   *  s = 512  -->    0 Vdc Output
   *  s = 1024 --> + 10 Vdc Output
   *
   * C
   * I
   * R                20*  *19
   * C                18*  *17
   *    Channel 7 --> 16*  *15 <-- Ground
   * B  Channel 6 --> 14*  *13 <-- Ground
   * R  Channel 5 --> 12*  *11 <-- Ground
   * D  Channel 4 --> 10*  *9  <-- Ground
   *    Channel 3 -->  8*  *7  <-- Ground
   * S  Channel 2 -->  6*  *5  <-- Ground
   * I  Channel 1 -->  4*  *3  <-- Ground
   * D  Channel 0 -->  2*  *1  <-- Ground (Bad)
   * E
   *
```

```
   ******************************************************/

void send_dac2b (s,ch)
             int s,ch;
  {
     if (LOCATIONLAB) return; /* avoid bus error writing to restricted memory */

     ch = ch << 2;                              /* offset for G-96 addressing */
     dac2b_a[ch] = s >> 2;                      /* write upper 8 bits to MSB  */
     /*dac2b_a[ch + DAC_LSB_OFFSET] = s << 6;*/ /* write lower 2 bits B3,B2   */
     dac2b_a[ch + 2] = s << 6;

/*
if (ARCHAIC_IGNORE)
{

     ch = ch << 2;                             /* offset for G-96 addressing  */
/*     dac2b_a[ch] = s >> 2;                     /* write upper 8 bits to MSB   */
/*     dac2b_a[ch + DAC_LSB_OFFSET] = s << 6;    /* write lower 2 bits B3,B2    */
/*} */
     return;

} /* send_dac2b */

  /*****************************************************************************
   *  get_adc1(n) -- reads ada-1 adc channel 'n' (channels 0-15)
   *****************************************************************************/

/* ---     VERIFIED MATCH CURRENT/OPERATIONAL MARCO AUV HARDWARE/SOFTWARE --- */

  int get_adc1 (n)
            int n;
  {
     int val;

     if (LOCATIONLAB)        /* in virtual world, do not read any slots */
     {
        return (0);
     }

     /*adc1_a[ADC1_CMD_REG] = n;
     while (adc1_a[ADC1_STATUS_REG] > 20);
     val = adc1_a[ADC1_MSB] << 2;
     val += adc1_a[ADC1_LSB] >> 6;*/

     adc1_a[4] = n;
     while (adc1_a[4] > 20);
     val = adc1_a[0] << 2;
     val += adc1_a[2] >> 6;

/*
if (ARCHAIC_IGNORE)
{
     adc1_a[ADC1_CMD_REG] = n;
     while (adc1_a[ADC1_STATUS_REG] > 20);    /* wait for data */
/*     val = adc1_a[ADC1_MSB] << 2;
     val += adc1_a[ADC1_LSB] >> 6;
     return (val);
}
*/
   return (val);

} /* get_adc1 */

  /*****************************************************************************
   *  get_adc2(n,g); -- Reads adc-2 channel 'n' (0-15)
   *                 with gain 'g' (0 to F => 0 - 1024)
   *****************************************************************************/

/* ---     VERIFIED MATCH CURRENT/OPERATIONAL MARCO AUV HARDWARE/SOFTWARE --- */

  int get_adc2 (n,g)
```

```
            int n,g;
  {
     int val;

     if (LOCATIONLAB)        /* in virtual world, do not read any slots */
     {
        return (0);
     }

     /*adc2_a[ADC2_CH_GAIN] = (n << 4) | g;*/     /* set c&g, start conv */
     /*while((adc2_a[ADC2_STATUS_REG] & 0x7) != 0);*/  /* wait for ready */

     adc2_a[0] = (n << 4) | g;     /* set c&g, start conv */
     while((adc2_a[2] & 0x7) != 0);  /* wait for ready */


/* This adc uses 0 - 4095 to represent full scale input, in order
     to write to the dac (which uses 0 -1023 for full scale) you
     must divide val by 4 or shift right by 2. Use the next line to
     get full resolution.
     val = adc2_a[ADC2_DATA];
     The next line is used for testing purposes only

     val = adc2_a[ADC2_DATA] >> 2;  */

   /*val = adc2_a[ADC2_DATA];*/

     val = adc2_a[1];
     val = val & 0x0FFF;

/*if (ARCHAIC_IGNORE)
{

     adc2_a[ADC2_CH_GAIN] = (n << 4) | g;     /* set c&g, start conv */
/*     while((adc2_a[ADC2_STATUS_REG] & 0x7) != 0);  /* wait for ready */

  /* This adc uses 0 - 4095 to represent full scale input, in order
     to write to the dac (which uses 0 -1023 for full scale) you
     must divide val by 4 or shift right by 2. Use the next line to
     get full resolution.
     val = adc2_a[ADC2_DATA];
     The next line is used for testing purposes only

     val = adc2_a[ADC2_DATA] >> 2;  */
/*
     val = adc2_a[ADC2_DATA];
     val = val & 0x0FFF;

     return(val);
}
*/
   return (val);

} /* get_adc2 */

/*****************************************************************************/
/*
The program code for the Multi-Function Interface originated from 'mfi.c'
Routines include Init_PortA, Init_PortB, Read_PortA, Read_PortB, Read_PortAB

Excerpt of 'mfi.c' comments follows:

  Program example for the Multi-Function-Interface (MFI)
  This example uses the 6821 PIA on the MFI board
  General purpose functions are provided to initialize the PIA
  and read/write data to the ports

  MFI P2 connector definitions are provided by the GESMFI-1 data
  sheet available from GESPAC, Inc.

  6821 device specifics are covered in the 8-bit microprocessor
  & peripheral data book from Motorola Inc.
```

```
  3/1/91        J. Rawlins      RealTime Software Consulting

*/
/*****************************************************************************/

  /*****************************************************************************
   *  Init_PortA(base, dir) -- Initialize Port A of MFI
   *                   dir: 1 = output port, 0 = input port
   *****************************************************************************/

/* ---     VERIFIED MATCH CURRENT/OPERATIONAL MARCO AUV HARDWARE/SOFTWARE --- */

void Init_PortA (base, dir)
  register struct MFI_PIA *base;/* base address of MFI board on G96 bus       */
  int dir;                      /* direction: 1 = output port, 0 = input port */
{
   register short temp;

   if (LOCATIONLAB)        /* in virtual world, do not read any slots */
   {
      return;
   }
   temp = (base->cra & 0x00FF);  /* get current value of control A     */
   temp &= ~4;                   /* clear bit #2 so we can access ddra */
      base->cra = temp;
      if ( dir )                    /* make port A all outputs */
            base->pra = 0x00FF;
      else                          /* port A is all inputs */
            base->pra = 0x0000;
      temp |= 4;                    /* set bit #2 to access data registers */
      base->cra = temp;

/*
if (ARCHAIC_IGNORE)
{
    register short temp;
    temp = base->cra;   /* save contents of control reg. (no-op)              */
/*    base->cra = 0x00;   /* select:  b2 = 0  data direction reg.               */
/*    base->pra = 0x00;   /* set portA: 0 = input                               */
/*    base->cra = 0x24;   /* select: access data reg.s (b2=1)                   */
                         /*  b5=1,b4=0,b3=0(read  w/cal restore)               */
/*}*/

}/* Init_PortA */


  /*****************************************************************************
   *  Init_PortB(base, dir) -- Initialize Port B of MFI
   *                   dir: 1 = output port, 0 = input port
   *****************************************************************************/

/* ---     VERIFIED MATCH CURRENT/OPERATIONAL MARCO AUV HARDWARE/SOFTWARE --- */

void Init_PortB (base, dir)
register struct MFI_PIA *base;  /* base address of MFI board on G96 bus       */
int dir;                        /* direction: 1 = output base, 0 = input base */
{
   register short temp;

   if (LOCATIONLAB)        /* in virtual world, do not read any slots */
   {
      return;
   }
   temp = (base->crb & 0x00FF);  /* get current value of control A       */
   temp &= ~4;                   /* clear bit #2 so we can access ddra   */
   base->crb = temp;
   if ( dir )                    /* make port B all outputs              */
      base->prb = 0x00FF;
   else                          /* port B is all inputs                 */
      base->prb = 0x0000;
   temp |= 4;                    /* set bit #2 to access data registers  */
   base->crb = temp;
```

```
}/* Init_PortB */

  /*****************************************************************************
   *  Read_PortA  (base) -- returns 8 bit value from port A
   *
   *****************************************************************************/

/* --- NOT YET UPDATED TO CURRENT/OPERATIONAL MARCO AUV HARDWARE/SOFTWARE --- */

/* not found! */

unsigned char Read_PortA (base)

register struct MFI_PIA *base;              /* base address of MFI              */
{
        register unsigned short temp;
        temp = base->pra;                   /* read data reg.should reset busy */
        return(temp & 0x00FF);              /* return data to calling program  */
}

  /*****************************************************************************
   *  Read_PortB  (base) -- returns 8 bit value from port B
   *
   *****************************************************************************/

/* --- NOT YET UPDATED TO CURRENT/OPERATIONAL MARCO AUV HARDWARE/SOFTWARE --- */

/* not found! */

unsigned char Read_PortB (base)
register struct MFI_PIA *base;                          /* base address of MFI */
{
        register unsigned short temp;

   if (LOCATIONLAB)        /* in virtual world, do not read any slots */
   {
      return (0);
   }
        temp = base->prb;
        return(temp & 0x00FF);
}

  /*****************************************************************************
   *  Read_PortAB (base) -- return a 16 bit value from ports
   *                        A and B combined then mask off
   *                        the 15 th and 16 th bits.
   * Note: PIA PA0-PA7 is the LSB and PB0-PB7 the MSB
   *****************************************************************************/

/* ---     VERIFIED MATCH CURRENT/OPERATIONAL MARCO AUV HARDWARE/SOFTWARE --- */

unsigned short  Read_PortAB (base)
register struct MFI_PIA *base;      /* base address of MFI                     */
{
        register unsigned short hi,lo,temp;

   if (LOCATIONLAB)        /* in virtual world, do not read any slots */
   {
      return (0);
   }
        lo = (base->pra & 0x00FF);  /* get least significant byte from A      */
        hi = (base->prb & 0x00FF);  /* and most  significant byte from B      */
        temp = ((hi << 8) + lo);    /* shift hi into upper byte of word       */
        return ( temp );            /* return data                            */
}

/*****************************************************************************/

/* --- NOT YET UPDATED TO CURRENT/OPERATIONAL MARCO AUV HARDWARE/SOFTWARE --- */

/* not found! */
```

```
void set_bsyA(base)  /* sets CB2 high (for busy to sending port)                */

register struct MFI_PIA *base;  /* base address of MFI                         */
{
    register short temp;

   if (LOCATIONLAB)        /* in virtual world, do not read any slots */
   {
      return;
   }
    temp = (base->cra & 0xFF);  /* save cra values                            */
    base->cra = 0x38;           /* 8 bit 1=  CR2  high                        */
    base->cra = temp;           /* restore cra values                         */
}

/* --- NOT YET UPDATED TO CURRENT/OPERATIONAL MARCO AUV HARDWARE/SOFTWARE --- */

/* not found! */

/* sets CB2 low (for -busy to sending port) */
void rst_bsyA(base)
register struct MFI_PIA *base;  /* base address of MFI                         */
{
    register short temp;

   if (LOCATIONLAB)        /* in virtual world, do not read any slots */
   {
      return;
   }
    temp = (base->cra & 0xFF);  /* save cra values                            */
    base->cra = 0x30;           /* 8 bit 0= CR2 low                           */
    base->cra = temp;           /* restore cra values                         */
}

/* --- NOT YET UPDATED TO CURRENT/OPERATIONAL MARCO AUV HARDWARE/SOFTWARE --- */

/* not found! */

int ck_sta (base)
register struct MFI_PIA *base;  /* base address of MFI                         */
{
    register unsigned short temp;

   if (LOCATIONLAB)        /* in virtual world, do not read any slots */
   {
      return (0);
   }
    temp = base->cra;           /* save cra values                            */
    return (temp);
}

/*****************************************************************************/

/* --- NOT YET UPDATED TO CURRENT/OPERATIONAL MARCO AUV HARDWARE/SOFTWARE --- */

void center_sonar ()
{
   int  direction,encoder_width;
   char encode;

   if (! SONARINSTALLED)
   {
       printf ("[start/stop center_sonar, SONARINSTALLED false]\n");
       return;
   }

   if (TRACE && DISPLAYSCREEN) printf ("[start  center_sonar ()]\n");

   encoder_width = 0;
   direction = 1;

   /* set_step_size('H'); */          /* '1' = 0.9, '2' = 1.8, '4' = 3.6 */
```

```
   /* Are we inside the Encoder Sensor ? */
   encode = query_sonar_1_reply ('M');    /* Test Head Direction (No Step) */
   if (SONARTRACE && DISPLAYSCREEN)
       printf("center_sonar:  encode = %c\n",encode);

   if ((encode == 't') || (encode == 'T'))
   {
      while( (encode == 't') || (encode == 'T') )
      {
         encode = query_sonar_1_reply ('+'); /* Index Sonar '+' direction */
      }

      /* Outside Encoder Sensor Now */
      direction = -1; /* Reverse Sonar Rotation to Establish Encoder Width */
   }

   while( (encode == 'f') || (encode == 'F') )
   {
      if(direction == 1)
      {
         encode = query_sonar_1_reply ('+'); /* Index Sonar '+' direction */
         if (SONARTRACE && DISPLAYSCREEN) printf("%c\n",encode);
      }
      else
      {
         encode = query_sonar_1_reply ('-'); /* Index Sonar '-' direction */
      }
   }

   /* Found Edge of Encoder */
   while( (encode == 't') || (encode == 'T') )
   {
      encoder_width = encoder_width + 1;

      if(direction == 1)
      {
         encode = query_sonar_1_reply ('+'); /* Index Sonar '+' direction */
      }
      else
      {
         encode = query_sonar_1_reply ('-'); /* Index Sonar '-' direction */
      }
   }
   if (SONARTRACE && DISPLAYSCREEN)
       printf ("center_sonar:  encoder width = %d\n",encoder_width);


   if (TRACE && DISPLAYSCREEN) printf ("[finish center_sonar ()]\n");

   return;

} /* end center_sonar () */

/*****************************************************************************/

/* --- NOT YET UPDATED TO CURRENT/OPERATIONAL MARCO AUV HARDWARE/SOFTWARE --- */

char query_sonar_1_reply (command_char)

   char command_char;

{
   /* code tested & taken from headtest.c (prior version ahead.c)            */

   int index,n,n_bytes;
   char reply,xx[20],c[1];

   if (! SONARINSTALLED)
   {
       printf ("[start/stop query_sonar_1_reply (), SONARINSTALLED false]\n");
       reply = ' ';
       return (reply);
```

```
    }

    if (TRACE && DISPLAYSCREEN) printf ("[start  query_sonar_1_reply ()]\n");

    /* likely string problems here <<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<< */
    c[0] = command_char;
    if (SONARTRACE && DISPLAYSCREEN)
        printf ("query_sonar_1_reply:  command_char = %c\n", command_char);
    n    = write (sonarpath,c,1);  /* write characters to sonarpath /t3 device */
    tsleep(10);

    n_bytes = _gs_rdy (sonarpath);
    if (SONARTRACE && (n_bytes > 1) && DISPLAYSCREEN)
        printf ("query_sonar_1_reply:  lost reply data, n_bytes = %d\n",n_bytes);

    if (n_bytes <= 0)
    {
        printf("query_sonar_1_reply:  bad read, n_bytes = %d\n",n_bytes);
    }
    else
    {
        n = read (sonarpath,xx,n_bytes);  /* n unused?????? */

        if (SONARTRACE && (n_bytes > 0) && DISPLAYSCREEN)
        {
           for (index = 0; index < n_bytes; index++)
           {
                printf("[%c %2d %2x]  ", xx[index], xx[index], xx[index]);
                if ((index+1) % 5 == 0)
                     printf("\n"); /* prevent writing off screen */
           }
           printf("\n");
        }
    }
    reply = xx[0];

    if (TRACE && DISPLAYSCREEN)
        printf ("[finish query_sonar_1_reply () returns %c]\n", reply);

    return (reply);
}

/*****************************************************************************/

/* --- NOT YET UPDATED TO CURRENT/OPERATIONAL MARCO AUV HARDWARE/SOFTWARE --- */

void set_step_size (step_code)

    char step_code;
{
    unsigned int n;
    char         reply;

    if (! SONARINSTALLED)
    {
        printf ("[start/stop set_step_size (), SONARINSTALLED false]\n");
        return;
    }

    if (TRACE && DISPLAYSCREEN) printf ("[start  set_step_size ()]\n");

    if (SONARTRACE && DISPLAYSCREEN) printf ("step code = %c\n",step_code);
    write (sonarpath,step_code,1);
    tsleep (1);  /* 256ths of a second */

    n = read (sonarpath, reply, 1);   /* n unused?????? */

    if (SONARTRACE && DISPLAYSCREEN) printf("step = %c\n",reply);

    if (TRACE && DISPLAYSCREEN) printf ("[finish set_step_size ()]\n");

    return;
```

```
}

/*****************************************************************************/

/* --- NOT YET UPDATED TO CURRENT/OPERATIONAL MARCO AUV HARDWARE/SOFTWARE --- */

void tty_mode (tty_mode_path, mode)

   int   tty_mode_path;
   int   mode;             /* note type specifications differ from headtest.c */
{
   static struct sgbuf old,new;
   static int init = 1;
   int status;

   if (! SONARINSTALLED)
   {
       if (TRACE && DISPLAYSCREEN)
           printf ("[start/stop  tty_mode, SONARINSTALLED false ()]\n");
       return;
   }

   if (init)
   {
      if (TRACE && DISPLAYSCREEN) printf ("[start  tty_mode ()]\n");

      init = 0;
      status = _gs_opt(tty_mode_path, &old);
      status = _gs_opt(tty_mode_path, &new);

      new.sg_class  = 0;
      new.sg_case   = 0;
      new.sg_backsp = 0;
      new.sg_delete = 0;
      new.sg_echo   = 0;
      new.sg_alf    = 0;
      new.sg_nulls  = 0;
      new.sg_pause  = 0;
      new.sg_page   = 0;
      new.sg_bspch  = 0;
      new.sg_dlnch  = 0;
      new.sg_eorch  = 0;
      new.sg_eofch  = 0;
      new.sg_rlnch  = 0;
      new.sg_dulnch = 0;
      new.sg_psch   = 0;
      new.sg_kbich  = 0;
      new.sg_kbach  = 0;
      new.sg_bsech  = 0;
      new.sg_bellch = 0;
      new.sg_parity = 0;
      new.sg_tabcr  = 0;
      new.sg_tabsiz = 0;
      new.sg_tbl    = 0;
      new.sg_col    = 0;
      new.sg_err    = 0;
  }

  if   (mode) _ss_opt (tty_mode_path, &new);
  else        _ss_opt (tty_mode_path, &old);

  if (TRACE && DISPLAYSCREEN) printf ("[finish tty_mode ()]\n");

  return;
} /* tty_mode */

/*****************************************************************************/

void open_virtual_world_socket ()        /* see os9sender.c for original code */
{
   if (LOCATIONLAB == FALSE) /* in water */
   {
```

```
      return;
   }

   if (TRACE && DISPLAYSCREEN)
       printf ("[start open_virtual_world_socket ()]\n");

/* Initialize communications blocks                                        */

/* Initialize both client & server ******************************************/

   /* Signal handlers for termination to override net_open () and net_close ()*/
   /*    signal handlers.  Otherwise you are unable to ^C kill this program.  */

#if (defined(sgi) || defined(sun))
signal (SIGHUP,  shutdown_virtual_world_socket);/* hangup                    */
signal (SIGINT,  shutdown_virtual_world_socket);/* interrupt character       */
signal (SIGKILL, shutdown_virtual_world_socket);/* kill signal from Unix     */
signal (SIGPIPE, shutdown_virtual_world_socket);/* broken pipe from other host*/
signal (SIGTERM, shutdown_virtual_world_socket);/* software termination      */
#endif

/* Initialize sender ********************************************************/

   /* start by finding default/desired remote host to connect to              */
   {

      server_entity = gethostbyname (virtual_world_remote_host_name);
      if (server_entity == NULL)
      {
         if (DISPLAYSCREEN)
         {
           printf("[open_virtual_world_socket:  virtual world remote host\n");
           printf(" (\"%s\") not found]\n",
                   virtual_world_remote_host_name);
           fflush (stdout); /* force completion of screen write */
         }
         /* error message needed on (open) output file   <<<<<<<<<<<<<<<<<<< */
         virtual_world_socket_opened = FALSE;
         exit (1);
      }
      else if (TRACE && DISPLAYSCREEN)
           {
              printf("[open_virtual_world_socket:  virtual world remote host ");
              printf("(\"%s\") located]\n",  virtual_world_remote_host_name);
           }

      /* Client opens server port *************************************/

      /* Fill in structure 'server_address' with the address of the          */
      /*         remote host (i.e. SERVER) that we want to connect with:      */
/*#if defined(sgi)
      bzero ((char *) &server_address, sizeof (server_address));
#endif
*/      server_address.sin_family   = AF_INET;      /* Internet protocol family */

      /* copy server IP address into sockaddr_in struct server_address        */
/*#if defined(sgi)
      bcopy (server_entity->h_addr, &(server_address.sin_addr.s_addr),
             server_entity->h_length);
#else
*/      strncpy(&(server_address.sin_addr.s_addr), server_entity->h_addr,
             server_entity->h_length);
/*#endif*/

      /* make sure port is in network byte order                              */
      server_address.sin_port = htons (AUVSIM1_TCP_PORT_1);

      /* Open TCP (Internet stream) socket                                    */
      if ( (socket_descriptor = socket (AF_INET, SOCK_STREAM, 0)) < 0 )
      {
         if (DISPLAYSCREEN)
         {
```

```
            printf ("[open_virtual_world_socket:  client can't open server");
            printf (" virtual world stream socket]");
         }
         /* error message needed on (open) output file <<<<<<<<<<<<<<<<<<<<<< */
         virtual_world_socket_opened = FALSE;
         exit (1);
      }
      else if (TRACE && DISPLAYSCREEN)
           {
              printf ("[open_virtual_world_socket:  client opened");
              printf (" virtual world server socket successfully]\n");
           }

      /* Connect to the server.  Process will block/sleep until connection is
            is established.  Timeout will return an error.                  */
      if (connect (                  socket_descriptor,
                   (struct sockaddr *) &server_address,
                                 sizeof (server_address)) < 0)
      {
         if (DISPLAYSCREEN)
         {
             printf ("[open_virtual_world_socket:  client can't connect to");
             printf (" virtual world server socket]\n");
         }
         /* error message needed on (open) output file <<<<<<<<<<<<<<<<<<<<<< */
         virtual_world_socket_opened = FALSE;
         exit (1);
      }
      else if (TRACE && DISPLAYSCREEN)
           {
              printf ("[execution client connected to virtual world");
              printf (" server socket successfully]\n");
           }
      virtual_world_socket_opened = TRUE;

   } /* end initialization                                                  */

   socket_stream = socket_descriptor;   /* client */

   if (TRACE && DISPLAYSCREEN)  /* print final info */
   {
     printf("[open_virtual_world_socket CLIENT:  socket_descriptor = %d]\n",
                                                 socket_descriptor);
     printf("[                                   socket_accepted   = %d]\n",
                                                 socket_accepted);
     printf("[                                   socket_stream     = %d]\n",
                                                 socket_stream);
   }

   if (TRACE && DISPLAYSCREEN)
       printf ("[finish open_virtual_world_socket ()]\n");

   return;

}/* end open_virtual_world_socket () */

/*****************************************************************************/

void shutdown_virtual_world_socket ()    /* see os9sender.c for original code */
{
   int  kill_return_value;

   shutdown_signal_received = TRUE;

   if (LOCATIONLAB == FALSE) /* in water */
   {
      return;
   }

   if (virtual_world_socket_opened == FALSE)
   {
       if (TRACE && DISPLAYSCREEN)
```

```
        {
           printf ("[virtual_world_socket_opened FALSE,");
           printf (" shutdown_virtual_world_socket ignored]\n");
        }
        return;
    }
    if (TRACE && DISPLAYSCREEN)
        printf ("[shutdown_virtual_world_socket start ...]\n");

    /* No need to send a message to other side that bridge is going down,      */
    /*    since SIGPIPE signal trigger may shutdown server on other side       */

    if (close (socket_stream) == -1)
    {
        if (TRACE && DISPLAYSCREEN)
         printf ("shutdown_virtual_world_socket close (socket_stream) failed\n");

        /* shutdown () reference:  "Using OS-9 Internet" manual p. 2-55        */

        if (shutdown (socket_stream, 2) == -1)
        {
            if (TRACE && DISPLAYSCREEN)
            {
                printf ("[shutdown_virtual_world_socket shutdown");
                printf (" (socket_stream, 2) failed]\n");
            }

            kill_return_value = kill (socket_stream, SIGKILL);

            if (TRACE && DISPLAYSCREEN)
            {
                printf ("[shutdown_virtual_world_socket kill (socket_stream,");
                printf (" SIGKILL) returned %d]\n", kill_return_value);
            }
        }
    }
    if (TRACE && DISPLAYSCREEN)
        printf ("[shutdown_virtual_world_socket return]\n");

    return;

} /* end shutdown_virtual_world_socket () */

/*****************************************************************************/

void send_buffer_to_virtual_world_socket () /* see os9sender.c for orig. code */
{
    bytes_left        = socket_length;
    bytes_written     = 0;
    ptr_index         = buffer;  /* this global string is the data to be sent  */

    if (LOCATIONLAB == FALSE) /* in water */
    {
       return;
    }

    if (virtual_world_socket_opened == FALSE)
    {
        if (TRACE && DISPLAYSCREEN)
        {
            printf ("[send_buffer_to_virtual_world_socket: ");
            printf ("virtual_world_socket_opened == FALSE, returning]\n");
        }
        return;
    }
    if (TRACE && DISPLAYSCREEN)
        printf ("[send_buffer_to_virtual_world_socket start ...]\n");

    while ((bytes_left > 0) && (bytes_written >= 0))  /* write loop ***********/
    {
        bytes_sent = write (socket_stream, ptr_index, bytes_left);
```

```
        if      (bytes_sent <  0) bytes_written = bytes_sent;
        else if (bytes_sent >  0)
             {
                 bytes_left     -= bytes_sent;
                 bytes_written  += bytes_sent;
                 ptr_index      += bytes_sent;
             }
        if (LOCATIONLAB && TRACE && DISPLAYSCREEN)
        {
              printf ("[record_data send_telemetry_to_server loop");
              printf ("  bytes sent = %d]\n", bytes_sent);
        }
    }
    if   (bytes_written < 0)
    {
        if (LOCATIONLAB && DISPLAYSCREEN)
        {
          printf ("[record_data send_telemetry_to_server () send failed, ");
          printf ("%d bytes_written]\n", bytes_written);
        }
        /* error message needed on (open) output file   <<<<<<<<<<<<<<<<<<<<<<< */
    }
    else if (LOCATIONLAB && TRACE && DISPLAYSCREEN)
         {
             printf ("[record_data send_telemetry_to_server total bytes sent");
             printf (" = %d]\n", bytes_written);
         }

    /* Check termination ******************************************************/

    if (strncmp (buffer, "shutdown", 8) == 0)
    {
        if (TRACE && DISPLAYSCREEN) printf
             ("[send_buffer_to_virtual_world_socket: shutdown_signal_received]\n");
        shutdown_virtual_world_socket ();
    }
    if (TRACE && DISPLAYSCREEN)
        printf ("[send_buffer_to_virtual_world_socket return]\n");

    return;

} /* end send_buffer_to_virtual_world_socket () */

/*****************************************************************************/

void get_string_from_virtual_world_socket ()  /* see os9sender.c for original */
{
    if (LOCATIONLAB == FALSE) /* in water */
    {
       return;
    }

    if (virtual_world_socket_opened == FALSE)
    {
        return;
    }
    if (TRACE && DISPLAYSCREEN)
        printf ("[get_string_from_virtual_world_socket start ...]\n");

    /* listen to remote host, relay to local network/program                  */

    bytes_left        = socket_length;
    bytes_received    = 0;
    ptr_index         = buffer_received; /* buffer_received is where results go */

    while ((bytes_left > 0) && (bytes_received >= 0))  /* read loop ************/
    {
       bytes_read = read (socket_stream, ptr_index, bytes_left);

       if      (bytes_read <  0) bytes_received = bytes_read;
       else if (bytes_read >  0)
       {
```

```
            bytes_left     -= bytes_read;
            bytes_received += bytes_read;
            ptr_index      += bytes_read;
        }
        if (TRACE && DISPLAYSCREEN)
        {
            printf ("[get_string_from_virtual_world_socket receiver block");
            printf (" loop bytes_read = %d]\n", bytes_read);
        }
        /* if nothing is waiting to be read, break out of read loop            */
        if ((bytes_read == 0) && (bytes_received == 0)) break;
    }
    if  (bytes_received < 0)      /* failure     */
    {
         if (TRACE && DISPLAYSCREEN)
         {
            printf ("[get_string_from_virtual_world_socket receiver block read");
            printf (" failed, bytes_received = %d\n", bytes_received);
         }
    }
    else if (bytes_received == 0)  /* no transfer */
    {
         if (TRACE && DISPLAYSCREEN)
         {
           printf("[get_string_from_virtual_world_socket received 0 bytes!!]\n");
         }
    }
    else if (bytes_received > 0)  /* success     */
         {
             if (TRACE && DISPLAYSCREEN)
             {
             printf("[get_string_from_virtual_world_socket received %d bytes]\n",
                      bytes_received);
             }
         }

    /* Check termination ******************************************************/

    if (strncmp (buffer_received, "shutdown", 8) == 0)
    {
        if (TRACE && DISPLAYSCREEN) printf
             ("[get_data_on_virtual_world_socket: shutdown_signal_received]\n");
        shutdown_virtual_world_socket ();
    }
    if (TRACE && DISPLAYSCREEN)
        printf ("[get_string_from_virtual_world_socket return]\n");

    return;

} /* end get_string_from_virtual_world_socket () */

/*****************************************************************************/

void record_data ()  /* this needs to be better partitioned <<<<<<<<<<<<<<<<< */

{
    static int count = 50;

    if (TRACE && DISPLAYSCREEN) printf ("[start  record_data ()]\n");

    system_time              = time (NULL);
    system_tmp               = localtime (&system_time);

    if (TRACE && DISPLAYSCREEN)
        printf ("[OS-9 system time is %02d:%02d:%02d, replication %d]\n",
                 system_tmp->tm_hour, system_tmp->tm_min, system_tmp->tm_sec,
                 replication_count);

    build_telemetry_string (buffer);

    if (TACTICALPARSE == FALSE)                 /* not used by tactical level  */
        send_buffer_to_virtual_world_socket (); /* there it goes               */
```

```
   if (TRACE && DISPLAYSCREEN)              /* telemetry report to screen      */
   {
       printf ("\nsending to virtual world:");
       printf ("\n%s", buffer);
   }
   else if (DISPLAYSCREEN)                  /* partial telemetry report        */
   {
       if (count == 50)
       {
           /* printf("t = %5.1f\n",t); */
           printf(" voltages:  %5.1f    %5.1f\n",computer_voltage,motor_voltage);
           count = 0;
       }
       ++count;

       printf ("sent telemetry to virtual world %5.11f  ", t);
       printf ("(%5.11f %5.11f %5.11f %5.11f %5.11f %5.11f)\n", x,y,z, phi,theta,psi);
   }

   if (LOCATIONLAB)
   {
          get_string_from_virtual_world_socket ();    /* here it comes */

          parse_telemetry_string (buffer_received);
   }
   if (TRACE && LOCATIONLAB && DISPLAYSCREEN)
   {
     printf ("------------------------------------------------------------\n");
   }

   if (((TACTICAL == FALSE) || (TACTICALPARSE)) && (auvdatafile != NULL))
       /* output data to telemetry file  */
   {                                  /* note that unmodified stream is saved */
      if (buffer_size == 0)           /* nothing was received, send auv_state */
                            fprintf (auvdatafile, "%s", buffer);
      else                            /* feedback was received, send uvw_state */
                            fprintf (auvdatafile, "%s", buffer_received);

      if (TRACE && DISPLAYSCREEN)
          printf("[printed to %s telemetry file]\n", AUVDATAFILENAME);
   }

   /* only send/print out every 10th telemetry entry to tactical level        */
   /*    due to serial port bandwidth limitations :-(                         */

   if ((TACTICAL) && TRACE && DISPLAYSCREEN)
          printf ("[sending data to tactical level]\n");

#if defined (sun)
#else
/* --- NOT YET UPDATED TO CURRENT/OPERATIONAL MARCO AUV HARDWARE/SOFTWARE --- */
                          /* writeln (serialpath, buffer, buffer_max); <<<<<<< */
   /* if (TACTICAL) write   (serialpath, buffer, buffer_max); */

      if ((TACTICAL) && TRACE && DISPLAYSCREEN)
          printf ("[write buffer to tactical level serialpath OK]\n");
#endif

   if (TACTICAL)  send_buffer_to_tactical_socket (); /* telemetry */

   if (auvtextfile != NULL)
      /* output data to .auv text file     */
      {
         if (TRACE && DISPLAYSCREEN)
             printf ("[sending data to .auv text file]\n");

          fprintf (auvtextfile, "%s", buffer);
          if (buffer_size != 0) /* feedback was received, also send uvw_state */
             fprintf (auvtextfile, "%s", buffer_received);

          if (TRACE && DISPLAYSCREEN)
              printf ("[fprintf to .auv text file OK]\n");
```

```
      }

   telemetry_records_saved ++;

   if   (((buffer_index +1) % FILEBUFFERSIZE) == 0) buffer_index = 0;
   else                                             buffer_index ++;

   /* need to copy buffer to buffer_array if caching telemetry  <<<<<<<<<<<<<  */

   if (TRACE && DISPLAYSCREEN) printf ("[buffer_index = %d]\n", buffer_index);

   /* - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - */
   /* test code to send data from file serial.d from wr2t1.c                  */
   /* read characters from file, echo characters to screen,                   */
   /*    send characters to serialpath /t1 device                             */
   /*     while ((c[0] = getc(serialtestfile)) != EOF)
          {
              putc   (c[0],stdout);
              writeln (serialpath, c, 1);
          }
   */
   /* - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - */

   if (TRACE && DISPLAYSCREEN) printf ("[finish record_data ()]\n");

   return;
}


/****************************************************************************/


void execute_shutdown_script()
{
   static int phase = 1;
   static double time_next_phase;

if (DISPLAYSCREEN && (phase == 1)) printf("[Shutdown Script Entered]\n");

   switch (phase)
   {
      case 1:
if (TRUE && DISPLAYSCREEN) printf("[Starting Phase 1 of Shutdown Script]\n");
         THRUSTERCONTROL      = TRUE;
         ROTATECONTROL        = FALSE;
         LATERALCONTROL       = FALSE;
         FOLLOWWAYPOINTMODE   = FALSE;
         WAYPOINTCONTROL      = FALSE;
         HOVERCONTROL         = FALSE;
         GPSFIXINPROGRESS = FALSE;
         x_command = x;
         y_command = y;
         z_command = -10;
         psi_command = psi;
         rpm = 0.0;
         port_rpm_command = 0.0;
         stbd_rpm_command = 0.0;
         time_next_phase = t + 20.0;
         phase = 2;
         break;
      case 2:
         if (t >= time_next_phase)
         {
if (TRUE && DISPLAYSCREEN) printf("Starting Phase 2 of Shutdown Script\n");
             THRUSTERCONTROL      = FALSE;
             time_next_phase    = t + 1.0;
             phase = 3;
        }
         break;
      case 3:
         if (t >= time_next_phase)
```

```
        {

if (TRUE && DISPLAYSCREEN) printf("Starting Phase 3 of Shutdown Script\n");
            LOOPFOREVER = FALSE;
            strcpy (buffer, "KILL");
            send_buffer_to_virtual_world_socket (); /* buffer msg sent */
            if (DISPLAYSCREEN) printf ("\n[end_test set TRUE]\n");
            end_test          = TRUE;

            fclose (auvscriptfile);
            auvscriptfilequit = TRUE;
            if (DISPLAYSCREEN)
printf("\n[QUIT condition: (%s backup file) mission.script.backup, file closed]\n",
                                        AUVSCRIPTFILENAME);
            fprintf (auvordersfile,
"%6.1f  %6.1f %5.1f %5.1f %5.1f %6.1f %6.1f %6.1f %6.1f  %5.1f %5.1f %5.1f %5.1f\n",
                          t, psi_command, x_command, y_command, z_command,
                       port_rpm_command, stbd_rpm_command,
                          rudder_command,   planes_command,
                              bow_lateral_thruster_command,
                            stern_lateral_thruster_command,
                             bow_vertical_thruster_command,
                           stern_vertical_thruster_command);
              x_dot                    = 0.0;
              y_dot                    = 0.0;
              z_dot                    = 0.0;
              phi_dot                  = 0.0; /* degrees/sec */
              theta_dot                = 0.0; /* degrees/sec */
              psi_dot                  = 0.0; /* degrees/sec */
              speed                    = 0.0;
              u                        = 0.0;
              v                        = 0.0;
              w                        = 0.0;
              p                        = 0.0; /* degrees/sec */
              q                        = 0.0; /* degrees/sec */
              r                        = 0.0; /* degrees/sec */
              delta_planes             = 0.0; /* degrees     */
              delta_rudder             = 0.0; /* degrees     */
              port_rpm                 = 0;
              stbd_rpm                 = 0;
              vertical_thruster_volts = 0.0;
               lateral_thruster_volts = 0.0;


              phase = 3;

       }
         break;
      default:
         break;
   }
}

/*****************************************************************************/

double read_computer_battery_voltage ()
{
   int    val;
   double voltage;

   val = get_adc1 (COMPUTER_VOLTAGE_CH);

   /* 3.03 Since a voltage divider Circuit by Approx 3 */
   voltage = (10.0/512.0) * ((double) val - 512.0) * 3.03;

   if (TRACE) printf ("read_computer_battery_voltage (): val     = %d\n", val);
   if (TRACE) printf ("read_computer_battery_voltage (): voltage = %f\n",
                      voltage);

   return (voltage);

} /* end read_computer_battery_voltage () */
```

```
/*****************************************************************************/

double read_motor_gyro_battery_voltage ()
{
   int    val;
   double voltage;

   val = get_adc1 (MOTOR_GYRO_VOLTAGE_CH);
   /* 3.03 Since a voltage divider Circuit by Approx 3 */
   voltage = (10.0/512.0) * ((double) val - 512.0) * 3.03;

   if (TRACE) printf ("read_motor_gyro_battery_voltage (): val     = %d\n",val);
   if (TRACE) printf ("read_motor_gyro_battery_voltage (): voltage = %f\n",
                      voltage);

   return (voltage);

} /* end read_motor_gyro_battery_voltage () */

/*****************************************************************************/


int leak_check ()
{
   int    i, adc_value [8], bow_adc_value, stern_adc_value;
   double bow_voltage, stern_voltage;

   LEAK = FALSE;

   for (i=0;i<8;++i)
   {
      adc_value [i] = get_adc1(i);
   }

   bow_adc_value = adc_value[5];
   bow_voltage   = (10.0/512.0)*(bow_adc_value - 512.0);

   if (TRACE) printf ("leak_check ():  bow_voltage = %5.1f\n",bow_voltage);


   if (bow_voltage > 1.7)
   {
      printf("***** BOW LEAK DETECTED ***** bow_voltage = %f\n", bow_voltage);
      LEAK = TRUE;
   }

   stern_adc_value = adc_value [6];
   if (TRACE) printf("stern_adc_value = %d\n",stern_adc_value);

   stern_voltage = (10.0/512.0)*(stern_adc_value - 512.0);

   if (TRACE) printf ("leak_check ():  stern_voltage = %5.1f\n", stern_voltage);

   if(stern_voltage > 1.7)
   {
      printf("***** STERN LEAK DETECTED ***** stern_voltage = %f\n",
             stern_voltage);
      LEAK = TRUE;
   }

   return (LEAK);

} /* end leak_check () */

/*****************************************************************************/

/* Dive Tracker Functions Won't Compile on SGI */
#if (defined(sun) || defined(sgi))
#else

int createdmod()
{
```

```
  mod_data *dat_struct;

  /* Create to data module */
  dt_dmod=CreateMod("DT2CL",sizeof(DT2CLMem),&dat_struct);
  if(dt_dmod == NULL) {
    printf("Data Module exists. Not created\n");
    exit(1);
  }
  dt_dmod->data_status=CL_R;

}

int CLReaddmod(r1_ptr,r2_ptr)
int *r1_ptr;
int *r2_ptr;
{

  static  int r1,r2;

  if (dt_dmod->data_status==DT_W)
    {
      r1=dt_dmod->dtr.r1;
      r2=dt_dmod->dtr.r2;
      dt_dmod->data_status=CL_R;
      *r1_ptr=r1;
      *r2_ptr=r2;
      return(NEW_DATA);
    }
/*****************************************************************************/
/*************************change in code ***********************************/
/*************** Dave Mcclairen needs a negative value to signal that there is
                no new data recorded by the dive tracker. ******************/

/*  *r1_ptr=-1; /*change in code from r1 to -1 to signal no update to tactical*/
/*  *r2_ptr=-1; /*change in code from r2 to -1 to signal no update to tactical*/
  return(OLD_DATA);

}

void *AttachMod(str,data_struct)
char *str;
mod_data **data_struct;
{

/*  Trying to link to the Data Module */
  if ( (*data_struct=modlink(str,ANY))==(mod_data *)-1 ) {
    return(NULL);
  }

/* ... then prepares an auxiliar structure to point to the Data Module */
  return( (void *)((long)((mod_data *)*data_struct)
                        +(long)(*data_struct)->data_offset) );
}

int DettachMod(data_struct)
mod_data *data_struct;
{

  if ( munlink(data_struct)==(mod_data *)-1 ) {
    return(-1);
  }
  return(0);
}

void *CreateMod(str,size,data_struct)
char *str;
unsigned size;
mod_data **data_struct;
{

/* Creates Data Module */
  if( (*data_struct=_mkdata_module
```

```
        (str,size, mkattrevs(MA_REENT | MA_GHOST,0x00),Perm_field)) ==
        (mod_data *)-1
    ){
    return(NULL);
  }

/* ... then prepares an auxiliar structure to point to the Data Module */
  return( (void *)((long)((mod_data *)*data_struct)
                        +(long)(*data_struct)->data_offset) );
}


#endif
/*****************************************************************************/

/* Mathematical Model for Estimating X and Y */

void XY_model_est(v_ls,v_rs,v_blt,v_slt,X_dot_c,Y_dot_c,update_vel)

   double v_ls,v_rs,v_blt,v_slt,X_dot_c,Y_dot_c;
   unsigned short update_vel;
{
   double alpa_x = 0.00375,
          alpa_y = 0.004,
          b_x =    1.33,
          b_y =    17.0;

   double M_x = (435.0 + 43.5) / 32.2,
          M_y = (435.0 + 348.0) / 32.2;

   double f_ls,f_rs,f_blt,f_slt,F_x,F_y;
   double u_ddot,v_ddot,r_ddot;

   f_ls = alpa_x*(v_ls*v_ls)*dsign(v_ls);
   f_rs = alpa_x*(v_rs*v_rs)*dsign(v_rs);

   f_blt =  alpa_y*(v_blt*v_blt)*dsign(v_blt);
   f_slt =  alpa_y*(v_slt*v_slt)*dsign(v_slt);

   F_x = f_ls  + f_rs;
   F_y = f_blt + f_slt;

/* printf("u = %5.1f\n",u);
printf("speed = %5.1f\n",speed); */

   /* Use speed sensor OR mathematical model to estimate speed */
   /* depending on previous speed and speed sensor value       */
   u_ddot = (F_x - b_x*u*fabs(u))/M_x;
   if ((u > 0.2) && (speed >= 0.25))
   {
      u = speed;
   }
   else if ((u < -0.2) && (speed >= 0.25))
   {
      speed = -speed;
      u = speed;
   }
   else
   {
      u = u + dt * (u_ddot);

      if (u > 0.24) u = 0.24;
      else if (u < -0.24) u = -0.24;

      speed = u;
   }

   v_ddot = (F_y - b_y*v*fabs(v))/M_y;
   v  = v + dt*(v_ddot);

   if(!update_vel)
   {
```

```
        u =  x_dot*cos_psi + y_dot*sin_psi;
        v = -x_dot*sin_psi + y_dot*cos_psi;
    }

    /* modify state vector values based on dead reckoning */

    x_dot = u*cos_psi - v*sin_psi + X_dot_c;
    y_dot = u*sin_psi + v*cos_psi + Y_dot_c;

    x = x + dt*x_dot;
    y = y + dt*y_dot;
}


/* Constant gain Kalman filter for depth */
void kalman_z(yk)

  double yk;
{
    double xk1_0,xk1_1,xk1_2;
    double phii[3][3],h[3],b[3],lk[3],res;

    /* a=[0 1 0;0 0 1;0 0 0]; phii=expm(a*0.1); where dt = 0.1 */
    b[0] = 0.0;
    b[1] = 0.0;
    b[2] = 1.0;

    /* phii = [1.0    0.1    0.005
              0.0    1.0      0.1
              0.0    0.0      1.0] */
    phii[0][0] = 1.0;
    phii[0][1] = 0.1;
    phii[0][2] = 0.005;
    phii[1][0] = 0.0;
    phii[1][1] = 1.0;
    phii[1][2] = 0.1;
    phii[2][0] = 0.0;
    phii[2][1] = 0.0;
    phii[2][2] = 1.0;

    /* h = [1 0 0]; */
    h[0] = 1.0;
    h[1] = 0.0;
    h[2] = 0.0;

    if(kal_init_z == 1)
    {
       z_kal = yk;
       z_dot_kal = 0.0;
       z_ddot_kal = 0.0;

       /* xk1=xk; */
       xk1_0 = z_kal;
       xk1_1 = z_dot_kal;
       xk1_2 = z_ddot_kal;
       kal_init_z = FALSE;
    }

    /* set lk = const. Slow Filter */
    lk[0] = 0.2544;
    lk[1] = 0.3727;
    lk[2] = 0.2731;

    /* xk1(:,i)=phii*xk(:,i); */
    xk1_0 = phii[0][0]*z_kal + phii[0][1]*z_dot_kal + phii[0][2]*z_ddot_kal;
    xk1_1 = phii[1][0]*z_kal + phii[1][1]*z_dot_kal + phii[1][2]*z_ddot_kal;
    xk1_2 = phii[2][0]*z_kal + phii[2][1]*z_dot_kal + phii[2][2]*z_ddot_kal;

    res = yk - (h[0]*xk1_0 + h[1]*xk1_1 + h[2]*xk1_2);

    /*  Set res = 0.0 if larger than threshold */
    if(fabs(res) > thres_z)
```

```
    {
       res = 0.0;
    }

    z_kal =  xk1_0 + lk[0]*res;
    z_dot_kal =  xk1_1 + lk[1]*res;
    z_ddot_kal =  xk1_2 + lk[2]*res;
}



/*****************************************************************************/
/*****************************************************************************/
/*****************************************************************************/
/* end of execution.c                                                        */
/*****************************************************************************/
/*****************************************************************************/
```

APPENDIX B - parsefunctions.c SOURCE CODE

```
/*****************************************************************************/
/*
 Program:          parse_functions.c

 Authors:          Don Brutzman

 Revised:          16 February 96

 System:           AUV Gespac 68020/68030, OS-9 version 2.4
 Compiler:         Gespac cc Kernighan & Richie (K&R) C

 Compilation:      ftp>     put parse_functions.c
                   auvsim1> chd execution
     [68020]       auvsim1> make -k2f execution
     [68030]       auvsim1> make      execution

     [Irix ]       fletch>  make execution

 Purpose:          Reduce size of execution.c to allow OS-9 C compiler to work


*/

/*****************************************************************************/
/* parse_functions.c                                                         */

#include "globals.h"
#include "statevector.h"
#include "defines.h"

/*****************************************************************************/

   void          parse_command_line_flags              ();
   int           parse_mission_script_commands         ();
   void          parse_mission_string_commands         ();

   void          print_valid_keywords                  ();

   void          get_control_constants                 ();

extern int       detect_death_spiral                   ();
extern void      send_buffer_to_virtual_world_socket   ();
extern void      clamp                                 ();

/*****************************************************************************/

void parse_command_line_flags (argc, argv)

  int argc; char **argv;     /* command line arguments */
{
   int index;

   if (DISPLAYSCREEN)
   {
       printf ("\n[parse_command_line_flags start: # arguments = %d]\n[", argc);
       for (i = 0; i < argc; i++) printf (" %s", argv[i]);
       printf (" ]\n");
   }

   if (DISPLAYSCREEN) printf ("[parse arguments:  ");
   {
      for (i = 1; i < argc; i++)
      {
         printf ("%s ", argv[i]);

         for (index = 0; index <= (int)strlen (argv[i]); index++)/* uppercase */
              argv[i] [index] = toupper (argv[i] [index]);
```

```
   }
      printf ("]\n");
}

strcpy (buffer, ""); /* initialize for SILENT */

for (i = 1; i < argc; i++)
{
   if      ((strcmp (argv[i],  "HELP") == 0) ||
            (strcmp (argv[i],     "?") == 0) ||
            (strcmp (argv[i],    "/?") == 0) ||
            (strcmp (argv[i],    "-?") == 0))
   {
         if (TRACE && DISPLAYSCREEN) printf ("[print_help] ");
         print_help = TRUE;
   }
   else if ((strcmp (argv[i],  "KEYBOARD")      == 0) ||
            (strcmp (argv[i], "KEY-BOARD")      == 0) ||
            (strcmp (argv[i], "KEYBOARD-INPUT") == 0) ||
            (strcmp (argv[i], "KEYBOARDINPUT")  == 0))
   {
         if (TRACE && DISPLAYSCREEN) printf ("[KEYBOARDINPUT = TRUE] ");
         KEYBOARDINPUT = TRUE;
   }
    else if (strcmp (argv[i],  "TRACE") == 0)
   {
         if (TRACE && DISPLAYSCREEN) printf ("[TRACE = TRUE] ");
         TRACE = TRUE;
   }
   else if ((strcmp (argv[i],  "TRACEOFF") == 0) ||
            (strcmp (argv[i], "TRACE-OFF") == 0) ||
            (strcmp (argv[i],   "NOTRACE") == 0) ||
            (strcmp (argv[i],  "NO-TRACE") == 0))
   {
         if (TRACE && DISPLAYSCREEN) printf ("[TRACE = FALSE] ");
         TRACE = FALSE;
   }
   else if ((strcmp (argv[i],  "LOOPFOREVER") == 0) ||
            (strcmp (argv[i], "LOOP-FOREVER") == 0))
   {
         if (TRACE && DISPLAYSCREEN) printf ("[LOOPFOREVER] ");
         LOOPFOREVER = TRUE;
   }
   else if ((strcmp (argv[i],  "LOOPONCE") == 0) ||
            (strcmp (argv[i], "LOOP-ONCE") == 0))
   {
         if (TRACE && DISPLAYSCREEN) printf ("[LOOPONCE] ");
         LOOPFOREVER = FALSE;
   }
   else if ((strcmp (argv[i],   "LOOPFILEBACKUP") == 0) ||
            (strcmp (argv[i], "LOOP-FILE-BACKUP") == 0))
   {
         if (TRACE && DISPLAYSCREEN) printf ("[LOOPFILEBACKUP] ");
         LOOPFILEBACKUP = TRUE;
   }
   else if ((strcmp (argv[i],   "ENTERCONTROLCONSTANTS") == 0) ||
            (strcmp (argv[i], "ENTER-CONTROL-CONSTANTS") == 0))
   {
         if (TRACE && DISPLAYSCREEN) printf ("[ENTERCONTROLCONSTANTS] ");
         ENTERCONTROLCONSTANTS = TRUE;
   }
   else if ((strcmp (argv[i],    "CONTROLCONSTANTSINPUTFILE") == 0) ||
            (strcmp (argv[i], "CONTROL-CONSTANTS-INPUT-FILE") == 0))
   {
         LOADCONTROLCONSTANTS = TRUE;
         printf ("[%s ", argv[i]);
         printf (CONTROLCONSTANTSINPUTNAME);
         printf ("]\n");
   }
   else   if ((strcmp (argv[i],  "TACTICAL")      == 0) ||
              (strcmp (argv[i],  "TACTICAL-HOST") == 0) ||
              (strcmp (argv[i],  "TACTICALHOST")  == 0))
```

```
{
        TACTICAL = TRUE;
        i++;
        if (i >= argc) print_help = TRUE;
        else
        {
            KEYBOARDINPUT = FALSE;
            sscanf (argv[i], "%s",  tactical_remote_host_name);
            if (TRACE && DISPLAYSCREEN)
                printf("[TACTICAL-HOST %s (KEYBOARD-OFF)]", tactical_remote_host_name);
        }
}
else    if ((strcmp (argv[i],  "NO-TACTICAL")  == 0) ||
            (strcmp (argv[i],  "TACTICAL-OFF") == 0))
{
        printf ("[%s]\n", argv[i]);
        TACTICAL = FALSE;
}
else if ((strcmp (argv[i],  "SONARTRACE") == 0) ||
         (strcmp (argv[i], "SONAR-TRACE") == 0))
{
        if (TRACE && DISPLAYSCREEN) printf ("[SONARTRACE] ");
        SONARTRACE = TRUE;
}
else if ((strcmp (argv[i],   "SONARTRACEOFF") == 0) ||
         (strcmp (argv[i], "SONAR-TRACE-OFF") == 0))
{
        if (TRACE && DISPLAYSCREEN) printf ("[SONARTRACEOFF] ");
        SONARTRACE = FALSE;
}
else if ((strcmp (argv[i],  "SONARINSTALLED") == 0) ||
         (strcmp (argv[i], "SONAR-INSTALLED") == 0))
{
        if (TRACE && DISPLAYSCREEN) printf ("[SONARINSTALLED] ");
        SONARINSTALLED = TRUE;
}
else if ((strcmp (argv[i],   "PARALLELPORTTRACE") == 0) ||
         (strcmp (argv[i], "PARALLEL-PORT-TRACE") == 0))
{
        if (TRACE && DISPLAYSCREEN) printf ("[PARALLELPORTTRACE] ");
        PARALLELPORTTRACE = TRUE;
}
  else if ((strcmp (argv[i],  "SILENT") == 0) ||
           (strcmp (argv[i], "SILENCE") == 0))
  {
        if (TRACE && DISPLAYSCREEN) printf ("[SILENT] ");
        /* send to virtual world after socket is open */
        strcpy (buffer, "SILENT"); /* copy current command to buffer */
  }
  else if ((strcmp (argv[i],  "TIMESTEP") == 0) ||
           (strcmp (argv[i], "TIME-STEP") == 0))
  {
        i++;
        if (i >= argc) print_help = TRUE;
        else
        {
            sscanf (argv[i], "%F", &TIMESTEP);
            if (TRACE && DISPLAYSCREEN) printf("[TIMESTEP %f]", TIMESTEP);
            if (TIMESTEP > 0.0) dt = TIMESTEP;
            else if (TRACE && DISPLAYSCREEN)
                    printf("  illegal TIMESTEP value, ignored.");
            if (TRUE && DISPLAYSCREEN) printf("  [dt =  %f]", dt);
        }
  }
  else if ((strcmp (argv[i],  "VIRTUALHOST") == 0) ||
           (strcmp (argv[i], "VIRTUAL-HOST") == 0) ||
           (strcmp (argv[i],  "VIRTUAL")     == 0) ||
           (strcmp (argv[i],  "REMOTE")      == 0) ||
           (strcmp (argv[i],  "REMOTEHOST")  == 0) ||
           (strcmp (argv[i],  "REMOTE-HOST") == 0) ||
           (strcmp (argv[i],  "DYNAMICS")    == 0))
  {
```

```
        i++;
        if (i >= argc) print_help = TRUE;
        else
        {
            sscanf (argv[i], "%s",  virtual_world_remote_host_name);
            if (TRACE && DISPLAYSCREEN)
                printf("[VIRTUAL-HOST %s]", virtual_world_remote_host_name);
        }
    }
    else if ((strcmp (argv[i],  "REALTIME") == 0) ||
            (strcmp (argv[i], "REAL-TIME") == 0))
    {
        if (TRACE && DISPLAYSCREEN) printf ("[REALTIME] ");
        REALTIME = TRUE;
    }
    else if ((strcmp (argv[i],  "NOREALTIME")   == 0)  ||
            (strcmp (argv[i],  "NO-REALTIME")  == 0)  ||
            (strcmp (argv[i],  "NO-REAL-TIME") == 0)  ||
            (strcmp (argv[i],  "NOWAIT")       == 0)  ||
            (strcmp (argv[i], "NO-WAIT")       == 0)  ||
            (strcmp (argv[i],  "NOPAUSE")      == 0)  ||
            (strcmp (argv[i], "NO-PAUSE")      == 0))
    {
        if (TRACE && DISPLAYSCREEN) printf ("[NOWAIT] ");
        REALTIME = FALSE;
    }
    else if ((strcmp (argv[i],  "NOEMAIL")  == 0) ||
             (strcmp (argv[i],  "NO-EMAIL") == 0))
    {
        if (TRACE && DISPLAYSCREEN) printf ("[NO EMAIL]");
        EMAIL = FALSE;
    }
    else   if ((strcmp (argv[i],  "LOCATIONLAB") == 0))
    {
           LOCATIONLAB = TRUE;
    }
    else   if ((strcmp (argv[i],  "LOCATIONWATER") == 0))
    {
           LOCATIONLAB = FALSE;
    }
    else   if ((strcmp (argv[i],  "GYROERROR") == 0) ||
               (strcmp (argv[i], "GYRO-ERROR") == 0) ||
               (strcmp (argv[i], "GYRO_ERROR") == 0))
    {
           i++;
           if (i >= argc)
           {
             print_help = TRUE;
             printf (" warning: invalid GYRO-ERROR command.\n");
           }
           else
           {
             sscanf (argv[i], "%lf", gyro_error);
             if (TRACE && DISPLAYSCREEN)
                 printf("[%s %5.2lf]", argv[i-1], gyro_error);
           }
    }
    else   if ((strcmp (argv[i], "DEPTH-CELL-BIAS")  == 0) ||
               (strcmp (argv[i], "DEPTHCELLBIAS")    == 0) ||
               (strcmp (argv[i], "DEPTH-CELL-ERROR") == 0) ||
               (strcmp (argv[i], "DEPTHCELLERROR")   == 0) ||
               (strcmp (argv[i], "DEPTH-BIAS")       == 0) ||
               (strcmp (argv[i], "DEPTHBIAS")        == 0) ||
               (strcmp (argv[i], "DEPTH-ERROR")      == 0) ||
               (strcmp (argv[i], "DEPTHERROR")       == 0))
    {
           i++;
           if (i >= argc)
           {
             print_help = TRUE;
             printf (" warning: invalid DEPTH-CELL-BIAS command.\n");
           }
```

```
                else
                {
                  sscanf (argv[i], "%lf", & depth_cell_bias);
                  if (TRACE && DISPLAYSCREEN)
                      printf("[%s %5.2lf]", argv[i-1], depth_cell_bias);
                }
        }
        else   if ((strcmp (argv[i], "TETHER")    == 0) ||
                   (strcmp (argv[i], "TETHERED") == 0))
        {
                DISPLAYSCREEN = TRUE;
                LOCATIONLAB   = FALSE;
        }
        else   if ((strcmp (argv[i], "UNTETHER")    == 0) ||
                   (strcmp (argv[i], "UNTETHERED") == 0))
        {
                DISPLAYSCREEN = FALSE;
                LOCATIONLAB   = FALSE;
        }
        else  print_help = TRUE;  /* invalid command line entry parameter found */

   }  /* end for loop through command line parameters                          */

   if (print_help) /* print help string ***************************************/
   {
    printf("\nUsage:  execution \n");
    print_valid_keywords ();
    exit (-1);
   }

   if (TRACE && DISPLAYSCREEN) printf ("\n[parse_command_line_flags complete]\n");

   return;

} /* end parse_command_line_flags () */

/*****************************************************************************/

int parse_mission_script_commands ()    /* get data from file at program start */
                                        /* mission.script.HELP => descriptions */
{
      /* command_buffer is the string to be parsed                            */

      int   index, read_another_line, parameters_read;
   double   parameter1,parameter2,parameter3,parameter4,parameter5,parameter6;
     char   parameter_string [60];
     char   backupcommand [50], new_filename [30];
      int   return_value;

     read_another_line = TRUE;


     /* do not skip to next command in KEYBOARD or script mode until ready  */
     if ((t < time_next_command) && (TACTICAL      == FALSE)
                                 && (TACTICALPARSE == FALSE))
     {
        if (TRACE && DISPLAYSCREEN)
        {
            printf ("\n[skip parse_mission_script_commands () until ");
            printf ("t > time_next_command]\n");
        }
        return (FALSE);
     }

     if (TRACE && DISPLAYSCREEN)
         printf ("\n[start parse_mission_script_commands ()]\n");

     if ((GPSFIXINPROGRESS) && (t >= time_postgps_dive))
     {
              if (TACTICAL) /* execution tell tactical gps-fix done */
              {
                 if (TRACE && DISPLAYSCREEN)
```

```
                    printf
        ("\n[send_buffer_to_tactical_socket (STABLE GPS TIMEOUT)]");
                  strcpy (buffer, "STABLE GPS TIMEOUT");
                  send_buffer_to_tactical_socket (); /* message */
                }
                GPSFIXINPROGRESS  = FALSE;
                read_another_line = FALSE;
      }
      if ((GPSFIXINPROGRESS) && (t >= time_gps_complete) &&
          (t < time_postgps_dive))
      {
                z_command         = previous_z_command;
                time_postgps_dive = t + 30.0;   /* head back to ordered depth */
                time_next_command = time_postgps_dive;
                time_gps_complete = time_postgps_dive + 1.0;
                read_another_line = FALSE;
                if (DISPLAYSCREEN) printf ("\n[GPS-FIX complete.]\n");
      }

      /* Only look at auvscriptfile if we are in script file execution mode   */

      if (   (TACTICALPARSE) /* tactical level internal use           */
          || (KEYBOARDINPUT) /* execution level                       */
          || (TACTICAL))     /* execution level getting tactical comms*/
      {
          /* no auvscriptfile setup required in these modes                  */
      }
      else if (   (auvscriptfile == NULL)/* auvscriptfile not yet opened      */
                || feof (auvscriptfile)   /* auvscriptfile end-of-file, repeat */
                ||    auvscriptfilequit)  /* flag for all done                 */
      {
           if (DISPLAYSCREEN)
           {
              printf ("\n[opening a copy of the auvscriptfile %s]\n",
                      AUVSCRIPTFILENAME);
              fflush (stdout);
           }
#if (defined(sun) || defined(sgi))
           sprintf (backupcommand, "rm      %s.backup", AUVSCRIPTFILENAME);
            printf ("%s\n", backupcommand);
           system  (backupcommand);
           sprintf (backupcommand, "cp   %s %s.backup", AUVSCRIPTFILENAME,
                                                        AUVSCRIPTFILENAME);
            printf ("%s\n", backupcommand);
           system  (backupcommand);
#else
           /* OS-9 */
           sprintf (backupcommand, "del     %s.backup", AUVSCRIPTFILENAME);
            printf ("%s\n", backupcommand);
           system  (backupcommand);
           sprintf (backupcommand, "copy %s %s.backup", AUVSCRIPTFILENAME,
                                                        AUVSCRIPTFILENAME);
            printf ("%s\n", backupcommand);
           system  (backupcommand);
#endif

           sprintf (backupcommand, "%s.backup", AUVSCRIPTFILENAME);
           auvscriptfile = fopen (backupcommand,"r");  /* input file */
           if (auvscriptfile == NULL)
           {
              printf ("AUV execution:  script file %s\n", AUVSCRIPTFILENAME);
              printf
          ("                 (or backup copy %s.backup) not found.\n",
                                        backupcommand);
              printf ("                 Ensure you are in the right directory:\n");
              printf ("                 auvsim1> chd /h0/execution       or\n");
              printf ("                 unix>    cd  ~brutzman/execution\n");
              printf
          ("                 Otherwise ensure you have a %s file.\n",
                                                AUVSCRIPTFILENAME);
              printf ("Exit.\n");
              exit (-1);
```

```
        }
        auvscriptfilequit = FALSE;

    }
    else if (TRACE && DISPLAYSCREEN)
            printf ("\n[auvscriptfile checks out as ready...]\n");

    if (TACTICALPARSE == FALSE)
    {
       /* open auvordersfile  - - - - - - - - - - - - - - - - - - - - - - - - - - */
       sprintf (buffer, "%s", AUVORDERSFILENAME);
       if (auvordersfile == NULL)
       {
           auvordersfile = fopen (buffer,"w");  /* output file */

           if (TRACE && DISPLAYSCREEN)
              printf ("\n[auvordersfile = %x, opened successfully]\n",
                          auvordersfile);

       fprintf (auvordersfile,
"\n\n");
       fprintf (auvordersfile,
"# NPS AUV file %s: commanded propulsion orders versus time\n",
                  AUVORDERSFILENAME);
       fprintf (auvordersfile,
"#\n");
       fprintf (auvordersfile,
"#         timestep:  %4.2f seconds\n", dt);
       fprintf (auvordersfile,
"#\n");
       fprintf (auvordersfile,
"# time  heading North East Depth    rpm   rpm    stern  stern   vertical     lateral \n");
       fprintf (auvordersfile,
"#                x    y     z      port  stbd   plane  rudder  thrusters   thrusters\n");
       fprintf (auvordersfile,
"#                                                             bow/stern   bow/stern\n");
       fprintf (auvordersfile,
"\n");

       }
       else if (TRACE && DISPLAYSCREEN)
               printf ("auvordersfile (%s) = %x\n", AUVORDERSFILENAME,
                          auvordersfile);
       if (auvordersfile == NULL)
       {
            printf ("AUV execution:  %s file open unsuccessful.\n", buffer);
            printf ("                Error.\n");
            printf ("Exit.\n");
            exit (-1);
       }
    }

    while (read_another_line)   /* ********* Parse loop ************ */
    {
       parameter1 = 0.0;
       parameter2 = 0.0;
       parameter3 = 0.0;

       /* Four-way switch: tactical  level parses commands internally, or    */
       /* ^^^^^^^^^^^^^^^  execution level parses commands from              */
       /*                     keyboard | tactical ood | mission.script file  */

       /* each option gets the next order and puts it in command_buffer      */

       if (TACTICALPARSE) /* tactical level internal use             */
       {
              /* command_buffer is already sent and ready */
              read_another_line = FALSE;
       }
       else if (KEYBOARDINPUT) /* this blocks!                       */
       {
              strcpy (buffer, "Enter command");
```

```
    /* send_buffer_to_virtual_world_socket ();      /* buffer msg sent */
       printf ("\n%s *** HERE ***:  ", buffer);
       strcpy (command_buffer, "");
       gets (command_buffer);
}
else if (TACTICAL) /* execution level getting tactical comms  */

        /* get command_buffer string from tactical level, nonblocking */
{
       if (TRACE && DISPLAYSCREEN)
           printf ("\n RECEIVE TACTICAL COMMAND *** HERE ***\n");
       strcpy (command_buffer, "");
       get_string_from_tactical_socket ();
       if (strlen (command_buffer) == 0)  /* no command was received  */
       {
           time_next_command = t + dt;     /* same as STEP command     */
           read_another_line = FALSE;      /*    (prevent blocking)    */
           if (TRACE && DISPLAYSCREEN)
               printf("no tactical command received, STEP & recheck\n");
           break;
       }
}
else /* get command_buffer string from auvscriptfile */
{
    strcpy (command_buffer, "");
    fgets  (command_buffer, 120, auvscriptfile);

    if (feof (auvscriptfile))
    {
        if (DISPLAYSCREEN)
        {
            printf ("\n[EOF condition:  (");
            printf ("%s copy) %s.backup, file closed]\n",
                     AUVSCRIPTFILENAME, AUVSCRIPTFILENAME);
        }
        fclose (auvscriptfile);
        auvscriptfilequit = TRUE;
        read_another_line = FALSE;
        end_test          = TRUE;
        strcpy (command_buffer, "");
        break;
    }
}

/* If Shutdown in Progress, Ignore Commands and Go to Shutdown Script */
if (HALTSCRIPT)
{
      return (FALSE);
}

/* parse the command, if any ---------------------------------------- */

if ((int)(strlen (command_buffer) <= 120) && TRACE && DISPLAYSCREEN)
{
    printf ("strlen (command_buffer) = %d", strlen (command_buffer));
    printf (">>>%s<<<", command_buffer);
}

parameters_read = sscanf (command_buffer, "%s", keyword);

if (TRACE && DISPLAYSCREEN)
{
    printf ("parameters_read = %d, keyword = %s",
             parameters_read,      keyword);
}

for (index=0; index<=(int)strlen (keyword); index++) /* set uppercase */
      keyword [index] = toupper (keyword [index]);

audible_command = TRUE;

if (TRACE && DISPLAYSCREEN)
```

```
        {
            printf (", uppercase  keyword = %s\n",  keyword);
        }

        if      ((parameters_read != 1)          ||
                 (strlen (keyword)         == 0) ||
                 (strlen (command_buffer)  == 0) ||
                 (command_buffer [0]       == '\n'))            /* blank line */
        {
                audible_command = FALSE;
                read_another_line = TRUE;
                if (DISPLAYSCREEN)  printf ("\n");
        }
        else    if  ( keyword [0] == '#')                          /* comment */
        {
                if (DISPLAYSCREEN)  printf ("%s", command_buffer);
                command_buffer [0] = ' ';
        }
        else    if (((keyword [0] == '/') && (keyword [1] == '/')) ||
                    ((keyword [0] == '/') && (keyword [1] == '*'))) /* comment */
        {
                if (DISPLAYSCREEN)  printf ("%s", command_buffer);
                command_buffer [0] = ' ';
                command_buffer [1] = ' ';
        }
        else if ((strcmp (keyword,  "HELP") == 0) ||
                 (strcmp (keyword,  "?")    == 0) ||
                 (strcmp (keyword,  "-?")   == 0) ||
                 (strcmp (keyword,  "/?")   == 0))
        {
                if (DISPLAYSCREEN) printf ("\n[HELP] ");
                print_valid_keywords ();
        }
        else    if      ((strcmp (keyword,  "WAIT") == 0) ||
                         (strcmp (keyword,  "RUN")  == 0))
        {
                parameters_read = sscanf (command_buffer, "%s%lf",
                                                        keyword, & parameter1);
                printf ("\n[%s     %6.2f; ", keyword, parameter1);
                if  ((parameters_read == 2) && (parameter1 >= 0.0))
                {
                     if (TACTICALPARSE) return (FALSE);

                     read_another_line = FALSE;
                     time_next_command = t + parameter1;
                     printf ("time of next command %6.2f]\n",
                                 time_next_command);
                     fprintf (auvordersfile,
"%6.1f  %6.1f %5.1f %5.1f %5.1f %6.1f %6.1f %6.1f %6.1f  %5.1f %5.1f %5.1f %5.1f\n",
                                t, psi_command, x_command, y_command, z_command,
                                port_rpm_command, stbd_rpm_command,
                                rudder_command, planes_command,
                                  bow_lateral_thruster_command,
                                stern_lateral_thruster_command,
                                  bow_vertical_thruster_command,
                                stern_vertical_thruster_command);
                }
                else printf (" warning: illegal time value, ignored\n");
        }
        else    if ((strcmp (keyword,  "TIME")       == 0) ||
                    (strcmp (keyword,  "WAITUNTIL")  == 0) ||
                    (strcmp (keyword,  "PAUSEUNTIL") == 0))
        {
                parameters_read = sscanf (command_buffer, "%s%lf",
                                                        keyword, & parameter1);
                printf ("\n[%s     %6.2f]\n", keyword, parameter1);
                if  (parameters_read == 2)
                {
                     if (TACTICALPARSE) return (FALSE);

                     read_another_line = FALSE;
                     time_next_command = parameter1;
```

```
                    fprintf (auvordersfile,
"%6.1f  %6.1f %5.1f %5.1f %5.1f %6.1f %6.1f %6.1f %6.1f  %5.1f %5.1f %5.1f %5.1f\n",
                            t, psi_command, x_command, y_command, z_command,
                            port_rpm_command, stbd_rpm_command,
                            rudder_command, planes_command,
                              bow_lateral_thruster_command,
                            stern_lateral_thruster_command,
                              bow_vertical_thruster_command,
                            stern_vertical_thruster_command);
                    if  (parameter1 <= t)
                    {
                         t = parameter1;
                         printf (" warning: time value has reset AUV clock,");
                         printf (" velocities reset to zero.\n");
                         u         = 0.0;
                         v         = 0.0;
                         w         = 0.0;
                         p         = 0.0;
                         q         = 0.0;
                         r         = 0.0;
                         x_dot     = 0.0;
                         y_dot     = 0.0;
                         z_dot     = 0.0;
                         phi_dot   = 0.0;
                         theta_dot = 0.0;
                         psi_dot   = 0.0;
                         read_another_line = TRUE;  /* no PDU */
                    }
               }
               else printf (" warning: illegal time value, ignored.\n");
          }
          else if ((strcmp (keyword, "TIMESTEP")  == 0) ||
                   (strcmp (keyword, "TIME-STEP") == 0))  /* different than STEP */
          {
               if (sscanf (command_buffer, "%s%F", keyword, &parameter1) == 2)
               {
                   if (TACTICALPARSE) return (FALSE);

                   if ((parameter1 > 0.0) && (parameter1 <= 5.0))
                   {
                       dt = parameter1;
                       if (DISPLAYSCREEN)
                           printf ("\n[TIMESTEP   %6.2f] ", dt);
                       if (TACTICALPARSE == FALSE)
                       fprintf (auvordersfile, "# timestep:  %4.2f seconds\n", dt);
                   }
                   else print_help = TRUE;
               }
               else print_help = TRUE;
          }
          else if ((strcmp (keyword,  "PAUSE") == 0)     ||
                   (strcmp (keyword, "-PAUSE") == 0))
          {
              if (DISPLAYSCREEN)
              {
                 printf ("\n[PAUSE]\n");
                 strcpy (buffer, " Press any key to continue");
                 send_buffer_to_virtual_world_socket (); /* buffer msg sent */
                 printf ("\n%s *** HERE ***:  ", buffer);
                 answer = getchar (); /* pause */
              }
          }
          else if ((strcmp (keyword,  "REALTIME")  == 0) ||
                   (strcmp (keyword,  "REAL-TIME") == 0))
          {
                 if (DISPLAYSCREEN) printf ("\n[REALTIME] ");
                 REALTIME = TRUE;
          }
          else if ((strcmp (keyword,  "MISSION")   == 0) ||
                   (strcmp (keyword,  "SCRIPT")    == 0) ||
                   (strcmp (keyword,  "FILE")      == 0) ||
                   (strcmp (keyword,  "FILENAME")  == 0))
```

```
        {
                parameters_read = sscanf (command_buffer, "%s%s",
                                                        keyword, new_filename);
                if  (parameters_read == 2)
                {
                    if (DISPLAYSCREEN)
                        printf ("\n[%s   %s]\n",    keyword, new_filename);
#if (defined(sun) || defined(sgi))
                    sprintf (backupcommand, "cp   %s %s", new_filename,
                            AUVSCRIPTFILENAME);
#else
                    sprintf (backupcommand, "copy %s %s", new_filename,
                            AUVSCRIPTFILENAME);
#endif
                    if (DISPLAYSCREEN)
                      printf ("%s\n", backupcommand);
                    system   (        backupcommand);
                    auvscriptfile == NULL; /* force re-read */
                }
                else
                {
                    if (DISPLAYSCREEN)
               printf ("\n[%s]  warning: no filename present, ignored\n", keyword);
                }
        }
        else if ((strcmp (keyword,  "KEYBOARD")       == 0) ||
                 (strcmp (keyword,  "KEYBOARD-ON")    == 0) ||
                 (strcmp (keyword,  "KEYBOARD-INPUT") == 0) ||
                 (strcmp (keyword,  "KEYBOARDINPUT")  == 0))
        {
                if (DISPLAYSCREEN) printf ("\n[%s]\n", keyword);
                KEYBOARDINPUT = TRUE;
        }
        else if ((strcmp (keyword,  "KEYBOARD-OFF") == 0) ||
                 (strcmp (keyword,  "NO-KEYBOARD")  == 0))
        {
                if (DISPLAYSCREEN) printf ("\n[%s]\n", keyword);
                KEYBOARDINPUT = FALSE;
        }
        else if ((strcmp (keyword,  "NOWAIT")      == 0) ||
                 (strcmp (keyword,  "NO-WAIT")     == 0) ||
                 (strcmp (keyword,  "NOREALTIME")  == 0) ||
                 (strcmp (keyword,  "NO-REALTIME") == 0) ||
                 (strcmp (keyword,  "NONREALTIME") == 0) ||
                 (strcmp (keyword,  "NO-PAUSE")    == 0) ||
                 (strcmp (keyword,  "NOPAUSE")     == 0))
        {
                if (DISPLAYSCREEN) printf ("\n[%s]\n", keyword);
                REALTIME = FALSE;
        }
        else   if     ((strcmp (keyword,  "ABORT")    == 0))
        {
                HALTSCRIPT = TRUE;
                if (DISPLAYSCREEN) printf ("\n[%s]\n", keyword);
        }
        else   if     ((strcmp (keyword,  "QUIT")     == 0) ||
                       (strcmp (keyword,  "STOP")     == 0) ||
                       (strcmp (keyword,  "DONE")     == 0) ||
                       (strcmp (keyword,  "EXIT")     == 0) ||
                       (strcmp (keyword,  "REPEAT")   == 0) ||
                       (strcmp (keyword,  "RESTART")  == 0) ||
                       (strcmp (keyword,  "COMPLETE") == 0) ||

                       (strcmp (keyword,  "KILL")     == 0) ||
                       (strcmp (keyword,  "SHUTDOWN") == 0))
        {
                /* note most of these commands don't reset LOOPFOREVER, except */
                /* KILL/SHUTDOWN which terminate the dynamics model connection */

                if ((strcmp (keyword,  "KILL")     == 0) ||
                    (strcmp (keyword,  "SHUTDOWN") == 0))
                {
```

```
                LOOPFOREVER = FALSE;
                strcpy (buffer, "KILL");
                send_buffer_to_virtual_world_socket (); /* buffer msg sent */
            }
            else
            {
                strcpy (buffer, "QUIT");
                send_buffer_to_virtual_world_socket (); /* buffer msg sent */
            }

            printf ("\n[%s]\n", keyword);
            if (TRACE && DISPLAYSCREEN) printf ("\n[end_test set TRUE]\n");
            end_test          = TRUE;
            read_another_line = FALSE;

            if (TACTICALPARSE) return (FALSE);

            fclose (auvscriptfile);
            auvscriptfilequit = TRUE;
            if (DISPLAYSCREEN)
printf("\n[QUIT condition: (%s backup file) mission.script.backup, file closed]\n",
                                        AUVSCRIPTFILENAME);
            fprintf (auvordersfile,
"%6.1f  %6.1f %5.1f %5.1f %5.1f %6.1f %6.1f %6.1f %6.1f  %5.1f %5.1f %5.1f %5.1f\n",
                        t, psi_command, x_command, y_command, z_command,
                      port_rpm_command, stbd_rpm_command,
                        rudder_command,   planes_command,
                            bow_lateral_thruster_command,
                          stern_lateral_thruster_command,
                           bow_vertical_thruster_command,
                         stern_vertical_thruster_command);
            x_dot                   = 0.0;
            y_dot                   = 0.0;
            z_dot                   = 0.0;
            phi_dot                 = 0.0; /* degrees/sec */
            theta_dot               = 0.0; /* degrees/sec */
            psi_dot                 = 0.0; /* degrees/sec */
            speed                   = 0.0;
            u                       = 0.0;
            v                       = 0.0;
            w                       = 0.0;
            p                       = 0.0; /* degrees/sec */
            q                       = 0.0; /* degrees/sec */
            r                       = 0.0; /* degrees/sec */
            delta_planes            = 0.0; /* degrees     */
            delta_rudder            = 0.0; /* degrees     */
            port_rpm                = 0;
            stbd_rpm                = 0;
            vertical_thruster_volts = 0.0;
             lateral_thruster_volts = 0.0;
            return (FALSE);
        }
        else    if ((strcmp (keyword,  "RPM")        == 0) ||
                    (strcmp (keyword,  "SPEED")      == 0) ||
                    (strcmp (keyword,  "PROPS")      == 0) ||
                    (strcmp (keyword,  "PROPELLORS") == 0))
        {
            parameters_read = sscanf (command_buffer, "%s%lf%lf",
                                                keyword,    & parameter1,
                                              & parameter2);
            if  (parameters_read == 3)
            {
                 WAYPOINTCONTROL = FALSE;
                 ROTATECONTROL   = FALSE;
                 HOVERCONTROL    = FALSE;
                 printf ("\n[%s   %6.2f %6.2f]\n",keyword,parameter1, parameter2);
                 port_rpm_command   = parameter1;
                 stbd_rpm_command   = parameter2;
            }
            else
            {
               parameters_read = sscanf (command_buffer, "%s%lf",
```

```
                                                keyword, & parameter1);
            printf ("\n[%s      %6.2f]\n", keyword, parameter1);
            if  (parameters_read == 2)
            {
                 WAYPOINTCONTROL = FALSE;
                 ROTATECONTROL   = FALSE;
                 HOVERCONTROL    = FALSE;
                 port_rpm_command   = parameter1;
                 stbd_rpm_command   = parameter1;
            }
            else printf (" warning: no value, ignored\n");
        }
}
else    if ((strcmp (keyword,  "COURSE")  == 0) ||
            (strcmp (keyword,  "HEADING") == 0) ||
            (strcmp (keyword,  "YAW")     == 0))
{
        parameters_read = sscanf (command_buffer, "%s%lf",
                                                keyword, & parameter1);
        printf ("\n[%s   %6.2f]\n", keyword, parameter1);
        if  (parameters_read == 2)
        {
             DEADSTICKRUDDER = FALSE;
             WAYPOINTCONTROL = FALSE;
             psi_command     = parameter1;
             psi_command_hover = parameter1;
             rotate_command  = 0.0;
             lateral_command = 0.0;
             ROTATECONTROL   = FALSE;
             LATERALCONTROL  = FALSE;

             if (HOVERCONTROL) /* report when stable again */
                 REPORTSTABLE  = TRUE;
        }
        else printf (" warning: no value, ignored\n");
}
else    if ((strcmp (keyword,  "TURN")          == 0) ||
            (strcmp (keyword,  "CHANGE-COURSE") == 0))
{
        parameters_read = sscanf (command_buffer, "%s%lf",
                                                keyword, & parameter1);
        printf ("\n[%s     %6.2f]\n", keyword, parameter1);
        if  (parameters_read == 2)
        {
             DEADSTICKRUDDER = FALSE;
             WAYPOINTCONTROL = FALSE;
             ROTATECONTROL   = FALSE;
             psi_command     = psi_command + parameter1;
        }
        else printf (" warning: no value, ignored\n");
}
else    if (strcmp (keyword,  "RUDDER")          == 0)
{
        parameters_read = sscanf (command_buffer, "%s%lf",
                                                keyword, & parameter1);
        printf ("\n[%s   %6.2f]\n", keyword, parameter1);
        if  (parameters_read == 2)
        {
             DEADSTICKRUDDER = TRUE;
             WAYPOINTCONTROL = FALSE;
             ROTATECONTROL   = FALSE;
             HOVERCONTROL    = FALSE;
             rudder_command  = parameter1;
        }
        else
        {
             printf (" warning: improper value, rudder order ignored\n");
        }
}
else    if (strcmp (keyword,  "DEADSTICKRUDDER") == 0)
{
        parameters_read = sscanf (command_buffer, "%s%lf",
```

```
                                                keyword, & parameter1);
        if  (parameters_read == 2)
        {
             printf ("\n[%s   %6.2f]\n", keyword, parameter1);
             DEADSTICKRUDDER = TRUE;
             WAYPOINTCONTROL = FALSE;
             ROTATECONTROL   = FALSE;
             HOVERCONTROL    = FALSE;
             rudder_command  = parameter1;
        }
        else
        {
             printf ("\n[%s]  ", keyword);
             DEADSTICKRUDDER = TRUE;
             WAYPOINTCONTROL = FALSE;
             ROTATECONTROL   = FALSE;
             rudder_command  = 0.0;
            printf(" warning: improper/missing value, rudder set to 0\n");
        }
}
else    if  (strcmp (keyword,  "DEPTH") == 0)
{
        parameters_read = sscanf (command_buffer, "%s%lf",
                                                keyword, & parameter1);
        printf ("\n[%s    %6.2f]\n", keyword, parameter1);
        if  (parameters_read == 2)
        {
             DEADSTICKPLANES = FALSE;
             z_command  = parameter1;

             if (HOVERCONTROL) /* report when stable again */
                 REPORTSTABLE  = TRUE;
        }
        else printf (" warning: no value, ignored\n");
}
else    if (strcmp (keyword,  "PLANES")          == 0)
{
        parameters_read = sscanf (command_buffer, "%s%lf",
                                                keyword, & parameter1);
        printf ("\n[%s   %6.2f]\n", keyword, parameter1);
        if  (parameters_read == 2)
        {
             DEADSTICKPLANES = TRUE;
             planes_command  = parameter1;
        }
        else printf (" warning: improper value, planes order ignored\n");
}
else    if (strcmp (keyword,  "DEADSTICKPLANES") == 0)
{
        parameters_read = sscanf (command_buffer, "%s%lf",
                                                keyword, & parameter1);
        if  (parameters_read == 2)
        {
             printf ("\n[%s   %6.2f]\n", keyword, parameter1);
             DEADSTICKPLANES = TRUE;
             planes_command  = parameter1;
        }
        else
        {
             printf ("\n[%s]  ", keyword);
             DEADSTICKPLANES = TRUE;
             planes_command  = 0.0;
             printf (" warning: improper value, planes set to 0\n");
        }
}
else    if ((strcmp (keyword,  "THRUSTERS-ON") == 0) ||
            (strcmp (keyword,  "THRUSTERS")    == 0) ||
            (strcmp (keyword,  "THRUSTERON")   == 0) ||
            (strcmp (keyword,  "THRUSTERSON")  == 0))
{
        printf ("\n[%s]\n", keyword);
        THRUSTERCONTROL = TRUE;
```

```
}
else    if ((strcmp (keyword,  "NOTHRUSTER")     == 0) ||
            (strcmp (keyword,  "NOTHRUSTERS")    == 0) ||
            (strcmp (keyword,  "THRUSTERS-OFF") == 0) ||
            (strcmp (keyword,  "THRUSTERSOFF")  == 0))
{
        printf ("\n[%s]\n", keyword);
        THRUSTERCONTROL = FALSE;
        ROTATECONTROL = FALSE;
        LATERALCONTROL = FALSE;
        HOVERCONTROL = FALSE;
}
else    if (strcmp (keyword,  "ROTATE")     == 0)
{
        parameters_read = sscanf (command_buffer, "%s%lf",
                                                  keyword, & parameter1);
        printf ("\n[%s   %6.2f]\n", keyword, parameter1);
        if  (parameters_read == 2)
        {
             THRUSTERCONTROL = TRUE;
             WAYPOINTCONTROL = FALSE;
             HOVERCONTROL    = FALSE;
              rotate_command = parameter1;
             clamp (&rotate_command, -12.0, 12.0, "rotate_command");
             lateral_command = 0.0;
             ROTATECONTROL   = TRUE;
             LATERALCONTROL  = FALSE;
        }
        else printf (" warning: no value, ignored\n");
}
else    if ((strcmp (keyword,  "NOROTATE")     == 0) ||
            (strcmp (keyword,  "ROTATEOFF")    == 0) ||
            (strcmp (keyword,  "ROTATE-OFF")   == 0))
{
        printf ("\n[%s]\n", keyword);
        rotate_command = 0.0;
        ROTATECONTROL  = FALSE;
}
else    if (strcmp (keyword,  "LATERAL")     == 0)
{
        parameters_read = sscanf (command_buffer, "%s%lf",
                                                  keyword, & parameter1);
        printf ("\n[%s   %6.2f]\n", keyword, parameter1);
        if  (parameters_read == 2)
        {
             THRUSTERCONTROL = TRUE;
             WAYPOINTCONTROL = FALSE;
             HOVERCONTROL    = FALSE;
              rotate_command = 0.0;
             lateral_command = parameter1;
             ROTATECONTROL   = FALSE;
             LATERALCONTROL  = TRUE;
        }
        else printf (" warning: no value, ignored\n");
}
else    if ((strcmp (keyword,  "NOLATERAL")     == 0) ||
            (strcmp (keyword,  "LATERALOFF")    == 0) ||
            (strcmp (keyword,  "LATERAL-OFF")   == 0))
{
        printf ("\n[%s]\n", keyword);
        lateral_command = 0.0;
        LATERALCONTROL  = FALSE;
}
else    if ((strcmp (keyword,  "DIVETRACKER1")     == 0) ||
            (strcmp (keyword,  "DIVE-TRACKER1")    == 0) ||
            (strcmp (keyword,  "DIVE-TRACKER-1")   == 0) ||
            (strcmp (keyword,  "DIVE_TRACKER1")    == 0) ||
            (strcmp (keyword,  "DIVE_TRACKER_1")   == 0) )
{
        parameters_read = sscanf (command_buffer, "%s%lf%lf%lf",
                                                 keyword,    & parameter1,
                                              & parameter2, & parameter3);
```

```
        printf ("\n[%s   %6.2f %6.2f %6.2f]\n",   keyword,      parameter1,
                                                 parameter2,   parameter3);
        if  (parameters_read == 4)
        {
             DiveTracker1_x = parameter1;
             DiveTracker1_y = parameter2;
             DiveTracker1_z = parameter3;
        }
        else printf (" warning: invalid DiveTracker1 position, ignored\n");
}
else    if ((strcmp (keyword,  "DIVETRACKER2")    == 0) ||
            (strcmp (keyword,  "DIVE-TRACKER2")   == 0) ||
            (strcmp (keyword,  "DIVE-TRACKER-2")  == 0) ||
            (strcmp (keyword,  "DIVE_TRACKER2")   == 0) ||
            (strcmp (keyword,  "DIVE_TRACKER_2")  == 0) )
{
        parameters_read = sscanf (command_buffer, "%s%lf%lf%lf",
                                                 keyword,    & parameter1,
                                               & parameter2, & parameter3);
        printf ("\n[%s   %6.2f %6.2f %6.2f]\n",   keyword,      parameter1,
                                                 parameter2,   parameter3);
        if  (parameters_read == 4)
        {
             DiveTracker2_x = parameter1;
             DiveTracker2_y = parameter2;
             DiveTracker2_z = parameter3;
        }
        else printf (" warning: invalid DiveTracker2 position, ignored\n");
}
else    if ((strcmp (keyword,  "GPS")       == 0) ||
            (strcmp (keyword,  "GPSFIX")    == 0) ||
            (strcmp (keyword,  "GPS-FIX")   == 0) )
{
   if (TACTICALPARSE == FALSE)
   {
        previous_z_command = z_command;
        if (z_command > 40.0) z_command = 41.0;  /* deep test tank */
        else                  z_command = -standoff_distance - 0.5;
                                                   /* rapid shallow  */
        GPSFIXINPROGRESS  = TRUE;
        time_gps_complete = t + 30.0;
        time_postgps_dive = t + 60.0;
        time_next_command = time_gps_complete;

        /* fixed guestimate of GPS fix interval */
        /* assume GPS-FIX behavior is properly controlled by tactical level */
   }
        if (DISPLAYSCREEN) printf ("\n[GPS-FIX]\n");
}
else    if ((GPSFIXINPROGRESS) &&
            ((strcmp (keyword,  "GPS-COMPLETE")       == 0) ||
             (strcmp (keyword,  "GPS-FIX-COMPLETE")   == 0) ||
             (strcmp (keyword,  "GPSCOMPLETE")        == 0) ||
             (strcmp (keyword,  "GPSFIXCOMPLETE")     == 0)) )
{
        z_command = previous_z_command;
        time_postgps_dive = t + 30.0;   /* head back to ordered depth */
        time_next_command = time_postgps_dive;
        time_gps_complete = time_postgps_dive + 1.0;
        read_another_line = FALSE;
        if (DISPLAYSCREEN) printf ("\n[GPS-FIX complete.]\n");
}
else    if ((strcmp (keyword,  "GYROERROR") == 0) ||
            (strcmp (keyword, "GYRO-ERROR") == 0) ||
            (strcmp (keyword, "GYRO_ERROR") == 0))
{
        parameters_read = sscanf (command_buffer, "%s%lf",
                                                 keyword,    & parameter1);
        printf ("\n[%s   %6.2f]\n",   keyword,      parameter1);
        if  (parameters_read == 2)
        {
             gyro_error = parameter1;
```

```
        }
        else printf (" warning: invalid GYRO-ERROR command, ignored\n");
}
else   if ((strcmp (keyword, "DEPTH-CELL-BIAS")  == 0) ||
           (strcmp (keyword, "DEPTHCELLBIAS")    == 0) ||
           (strcmp (keyword, "DEPTH-CELL-ERROR") == 0) ||
           (strcmp (keyword, "DEPTHCELLERROR")   == 0) ||
           (strcmp (keyword, "DEPTH-BIAS")       == 0) ||
           (strcmp (keyword, "DEPTHBIAS")        == 0) ||
           (strcmp (keyword, "DEPTH-ERROR")      == 0) ||
           (strcmp (keyword, "DEPTHERROR")       == 0))
 {
        parameters_read = sscanf (command_buffer, "%s%lf",
                                                  keyword,    & parameter1);
        printf ("\n[%s   %6.2f]\n",   keyword,      parameter1);
        if  (parameters_read == 2)
        {
             depth_cell_bias = parameter1;
        }
        else printf (" warning: invalid DEPTH-CELL-BIAS command, ignored\n");
}
else   if ((strcmp (keyword,  "LOCATIONLAB") == 0))
{
       LOCATIONLAB = TRUE;
}
else   if ((strcmp (keyword,  "LOCATIONWATER") == 0))
{
       LOCATIONLAB = FALSE;
}
 else   if ((strcmp (keyword,  "POSITION")    == 0) ||
            (strcmp (keyword,  "LOCATION")    == 0) ||
            (strcmp (keyword,  "FIX")         == 0) )
 /* note this command must be sent to virtual world (AUVsocket.C tests) */
 {
        parameters_read = sscanf (command_buffer, "%s%lf%lf%lf",
                                                  keyword,    & parameter1,
                                                & parameter2, & parameter3);
        printf ("\n[%s   %6.2f %6.2f %6.2f]\n",   keyword,    parameter1,
                                                  parameter2,   parameter3);
        if  (parameters_read == 4)
        {
             x = parameter1;
             y = parameter2;
             z = parameter3;   /* note depth cell will likely update z */
          /* skip line in telemetry file to break point-to-point lines */
             if ((TACTICALPARSE) || (TACTICAL == FALSE))
                fprintf (auvdatafile, "\n");

             if (TRACE)
                 printf ("\nsending fix to virtual world:  [%s]\n", buffer);
             strcpy (buffer, command_buffer); /* copy command to buffer*/
             send_buffer_to_virtual_world_socket (); /* send to vw     */
        }
        else if  (parameters_read == 3)
        {
             x = parameter1;
             y = parameter2;
          /* skip line in telemetry file to break point-to-point lines */
             if ((TACTICALPARSE) || (TACTICAL == FALSE))
                fprintf (auvdatafile, "\n");

             if (TRACE)
                 printf ("\nsending fix to virtual world:  [%s]\n", buffer);
             strcpy (buffer, command_buffer); /* copy command to buffer*/
             send_buffer_to_virtual_world_socket (); /* send to vw     */
        }
        else printf (" warning: invalid x/y/z fix position, ignored\n");


 }
 else   if ((strcmp (keyword,  "ORIENTATION") == 0) ||
            (strcmp (keyword,  "ROTATION")    == 0) )
```

```
{
        parameters_read = sscanf (command_buffer, "%s%lf%lf%lf",
                                                keyword,     & parameter1,
                                              & parameter2, & parameter3);
        printf ("\n[%s   %6.2f %6.2f %6.2f]\n",   keyword,     parameter1,
                                                parameter2,    parameter3);
        if  (parameters_read == 4)
        {
             phi   = parameter1;
             theta = parameter2;
             psi   = parameter3;
        }
        else
        printf (" warning: invalid phi/theta/psi orientation, ignored\n");
}
else    if (strcmp (keyword,  "POSTURE") == 0)
{
        parameters_read = sscanf (command_buffer, "%s%lf%lf%lf%lf%lf%lf",
                                                keyword,     & parameter1,
                                              & parameter2, & parameter3,
                                              & parameter4, & parameter5,
                                              & parameter6);
        printf ("\n[%s   %6.2f %6.2f %6.2f %6.2f %6.2f %6.2f]\n",
                                                keyword,       parameter1,
                                                parameter2,    parameter3,
                                                parameter4,    parameter5,
                                                parameter6);
        if  (parameters_read == 7)
        {
             x     = parameter1;
             y     = parameter2;
             z     = parameter3;
             phi   = parameter4;
             theta = parameter5;
             psi   = parameter6;
             start_psi = parameter6;
             kal_init_z = TRUE;

          /* skip line in telemetry file to break point-to-point lines */
             if ((TACTICAL == FALSE) || (TACTICALPARSE))
                 fprintf (auvdatafile, "\n");
        }
        else
      printf(" warning: invalid posture values (6 required), ignored\n");
}

else if ((strcmp (keyword, "OCEANCURRENT")  == 0) ||
         (strcmp (keyword, "OCEAN-CURRENT") == 0))
{
        parameters_read = sscanf (command_buffer, "%s%lf%lf%lf",
                                                keyword,     & parameter1,
                                              & parameter2, & parameter3);
        if  (parameters_read == 4)
        {
             printf ("\n[%s   %6.2f %6.2f %6.2f]\n", keyword, parameter1,
                                                parameter2, parameter3);
             AUV_oceancurrent_x = parameter1;
             AUV_oceancurrent_y = parameter2;
             AUV_oceancurrent_z = parameter3;
        }
        else if  (parameters_read == 3)
        {
             printf ("\n[%s   %6.2f %6.2f]\n", keyword, parameter1,
                                                       parameter2);
             AUV_oceancurrent_x = parameter1;
             AUV_oceancurrent_y = parameter2;
        }
        else
        {
             printf ("\n warning: improper number of OCEAN-CURRENT ");
             printf ("values, ignored\n");
        }
```

```
}

else   if ((strcmp (keyword,  "CONTINUE") == 0) ||
           (strcmp (keyword,        "GO") == 0))
{
       if (DISPLAYSCREEN) printf ("\n[%s]\n", keyword);
       return (FALSE); /* no action required */
}
else   if ((strcmp (keyword,        "STEP") == 0) ||
           (strcmp (keyword, "SINGLE-STEP") == 0))
{
       if (DISPLAYSCREEN) printf ("\n[%s]\n", keyword);
       time_next_command = t + dt;
       read_another_line = FALSE;
}
else   if ((strcmp (keyword,  "TRACE")    == 0) ||
           (strcmp (keyword,  "TRACE-ON") == 0))
{
       if (DISPLAYSCREEN) printf ("\n[TRACE = TRUE] ");
       TRACE = TRUE;
}
else if ((strcmp (keyword,  "TRACEOFF")  == 0) ||
         (strcmp (keyword,  "TRACE-OFF") == 0) ||
         (strcmp (keyword,  "NOTRACE")   == 0) ||
         (strcmp (keyword,  "NO-TRACE")  == 0))
{
       if (DISPLAYSCREEN) printf ("\n[TRACE = FALSE] ");
       TRACE = FALSE;
}
else if ((strcmp (keyword, "LOOPFOREVER")  == 0) ||
         (strcmp (keyword, "LOOP-FOREVER") == 0))
{
       if (DISPLAYSCREEN) printf ("\n[LOOPFOREVER] ");
       LOOPFOREVER = TRUE;
}
else if ((strcmp (keyword,  "LOOPONCE")  == 0) ||
         (strcmp (keyword,  "LOOP-ONCE") == 0))
{
       if (DISPLAYSCREEN) printf ("\n[LOOPONCE] ");
       LOOPFOREVER = FALSE;
}
else if ((strcmp (keyword,  "LOOPFILEBACKUP") == 0) ||
         (strcmp (keyword,  "LOOP-FILE-BACKUP") == 0))
{
       if (DISPLAYSCREEN) printf ("\n[LOOPFILEBACKUP] ");
       LOOPFILEBACKUP = TRUE;
}
else if ((strcmp (keyword,  "ENTERCONTROLCONSTANTS")   == 0) ||
         (strcmp (keyword,  "ENTER-CONTROL-CONSTANTS") == 0))
{
       if (DISPLAYSCREEN) printf ("\n[ENTERCONTROLCONSTANTS] ");
       ENTERCONTROLCONSTANTS = TRUE;
       get_control_constants ();
       fflush (stdin);
}
else if ((strcmp (keyword,    "CONTROLCONSTANTSINPUTFILE") == 0) ||
         (strcmp (keyword, "CONTROL-CONSTANTS-INPUT-FILE") == 0))
{
       LOADCONTROLCONSTANTS = TRUE;
       get_control_constants ();
       printf ("\n[CONTROLCONSTANTSINPUTFILE %s]",
                 CONTROLCONSTANTSINPUTNAME);
}
else   if ((strcmp (keyword,  "SLIDINGMODECOURSE")   == 0) ||
           (strcmp (keyword,  "SLIDING-MODE-COURSE") == 0))
{
       printf ("\n[%s = TRUE]\n", keyword);
       SLIDINGMODECOURSE = TRUE;
       WAYPOINTCONTROL = FALSE;
       ROTATECONTROL   = FALSE;
       HOVERCONTROL    = FALSE;
```

```
        }
        else   if ((strcmp (keyword,  "SLIDINGMODEOFF")    == 0) ||
                   (strcmp (keyword,  "SLIDING-MODE-OFF") == 0))
        {
               printf ("\n[%s:  SLIDINGMODECOURSE = FALSE]\n", keyword);
               SLIDINGMODECOURSE = FALSE;
        }
        else   if ((strcmp (keyword,  "TACTICAL")       == 0) ||
                   (strcmp (keyword,  "TACTICAL-HOST") == 0) ||
                   (strcmp (keyword,  "TACTICALHOST")  == 0))
        {
              if (sscanf (command_buffer, "%s %s", keyword, parameter_string) == 2)
               {
                   TACTICAL = TRUE;
                   KEYBOARDINPUT = FALSE;
                   strcpy (tactical_remote_host_name, parameter_string);
                   open_tactical_socket ();
                   if (DISPLAYSCREEN)
                                          printf("\n[TACTICAL-HOST  %s  (KEYBOARD-OFF)]",
tactical_remote_host_name);
               }
               else print_help = TRUE;
        }
        else   if ((strcmp (keyword,  "NO-TACTICAL")  == 0) ||
                   (strcmp (keyword,  "TACTICAL-OFF") == 0))
        {
               printf ("\n[%s]\n", keyword);
               TACTICAL = FALSE;
        }
        else if (strcmp (keyword,  "SONARTRACE") == 0)
        {
               if (DISPLAYSCREEN) printf ("\n[SONARTRACE] ");
               SONARTRACE = TRUE;
        }
        else if (strcmp (keyword,  "SONARTRACEOFF") == 0)
        {
               if (DISPLAYSCREEN) printf ("\n[SONARTRACEOFF] ");
               SONARTRACE = FALSE;
        }
        else if (strcmp (keyword,  "SONARINSTALLED") == 0)
        {
               if (DISPLAYSCREEN) printf ("\n[SONARINSTALLED] ");
               SONARINSTALLED = TRUE;
        }
        else if (strcmp (keyword,  "PARALLELPORTTRACE") == 0)
        {
               if (DISPLAYSCREEN) printf ("\n[PARALLELPORTTRACE] ");
               PARALLELPORTTRACE = TRUE;
        }
        else if ((strcmp (keyword,  "AUDIBLE")  == 0) ||
                 (strcmp (keyword,  "AUDIO")    == 0) ||
                 (strcmp (keyword,  "AUDIO-ON") == 0) ||
                 (strcmp (keyword,  "SOUND-ON") == 0) ||
                 (strcmp (keyword,  "SOUNDON")  == 0) ||
                 (strcmp (keyword,  "SOUND")    == 0))
        {
               if (DISPLAYSCREEN) printf ("\n[AUDIBLE] ");
               strcpy (buffer, "AUDIBLE"); /* copy current command to buffer */
               send_buffer_to_virtual_world_socket (); /* send to sound driver */
        }
        else if ((strcmp (keyword,  "SILENT")   == 0) ||
                 (strcmp (keyword,  "SILENCE")  == 0) ||
                 (strcmp (keyword,  "NOSOUND")  == 0) ||
                 (strcmp (keyword,  "NO-SOUND") == 0) ||
                 (strcmp (keyword,  "SOUNDOFF") == 0) ||
                 (strcmp (keyword, "SOUND-OFF") == 0) ||
                 (strcmp (keyword,  "AUDIOOFF") == 0) ||
                 (strcmp (keyword, "AUDIO-OFF") == 0) ||
                 (strcmp (keyword,  "QUIET")    == 0))
        {
               if (DISPLAYSCREEN) printf ("\n[SILENT] ");
               strcpy (buffer, "SILENT");  /* copy current command to buffer */
```

```
            send_buffer_to_virtual_world_socket (); /* send to sound driver */
        }
        else if ((strcmp (keyword, "SOUNDSERIAL")  == 0) ||
                 (strcmp (keyword, "SOUND-SERIAL") == 0))
        {
            if (TRACE && DISPLAYSCREEN) printf ("\n[SOUNDSERIAL ON] ");
            strcpy (buffer, "SOUNDSERIAL");        /* send precise keyword */
            send_buffer_to_virtual_world_socket ();/* send to sound driver */
            audible_command = FALSE;
        }
        else if ((strcmp (keyword, "SOUNDPARALLEL")  == 0) ||
                 (strcmp (keyword, "SOUND-PARALLEL") == 0))
        {
            strcpy (buffer, "SOUNDPARALLEL");     /* send precise keyword */
            send_buffer_to_virtual_world_socket (); /* send to sound driver */
            audible_command = FALSE;
        }
        else if ((strcmp (keyword, "EMAIL")      == 0) ||
                 (strcmp (keyword, "EMAIL-ON")   == 0) ||
                 (strcmp (keyword, "E-MAIL")     == 0) ||
                 (strcmp (keyword, "E-MAIL-ON")  == 0) ||
                 (strcmp (keyword, "EMAILON")    == 0))
        {
            if (TRACE && DISPLAYSCREEN) printf ("\n[EMAIL ON] ");
            EMAIL = TRUE;
        }
        else if ((strcmp (keyword, "EMAILOFF")    == 0) ||
                 (strcmp (keyword, "EMAIL-OFF")   == 0) ||
                 (strcmp (keyword, "E-MAILOFF")   == 0) ||
                 (strcmp (keyword, "E-MAIL-OFF")  == 0) ||
                 (strcmp (keyword, "NO-E-MAIL")   == 0) ||
                 (strcmp (keyword, "NO-EMAIL")    == 0) ||
                 (strcmp (keyword, "NO-E-MAIL")   == 0) ||
                 (strcmp (keyword, "NOEMAIL")     == 0))
        {
            if (TRACE && DISPLAYSCREEN) printf ("\n[EMAIL OFF] ");
            EMAIL = FALSE;
        }
        else if ((strcmp (keyword, "WAYPOINT")     == 0) ||
                 (strcmp (keyword, "WAYPOINT-ON") == 0))
        {
            parameters_read = sscanf (command_buffer, "%s%lf%lf%lf",
                                                  keyword,    & parameter1,
                                                & parameter2, & parameter3);
            if  (parameters_read == 4)
            {
                 printf ("\n[%s   %6.2f %6.2f %6.2f]\n", keyword, parameter1,
                                                  parameter2, parameter3);
                 WAYPOINTCONTROL = TRUE;
                 FOLLOWWAYPOINTMODE = TRUE;
                 HOVERCONTROL     = FALSE;
                 ROTATECONTROL    = FALSE;
                 LATERALCONTROL   = FALSE;
                 REPORTSTABLE     = TRUE;
                 DEADSTICKRUDDER  = FALSE;
                 x_command        = parameter1;
                 y_command        = parameter2;
                 z_command        = parameter3;
            port_rpm_command      = fabs (port_rpm_command); /* ensure fwd  */
            stbd_rpm_command      = fabs (stbd_rpm_command); /* motion only */
                 detect_death_spiral (TRUE);   /* resets static variables  */

            if (TACTICALPARSE == FALSE) fprintf (auvordersfile,
"%6.1f  %6.1f %5.1f %5.1f %5.1f %6.1f %6.1f %6.1f %6.1f  %5.1f %5.1f %5.1f %5.1f\n",
                          t, psi_command, x_command, y_command, z_command,
                       port_rpm_command, stbd_rpm_command,
                         rudder_command,   planes_command,
                             bow_lateral_thruster_command,
                           stern_lateral_thruster_command,
                            bow_vertical_thruster_command,
                          stern_vertical_thruster_command);
            }
```

```
            else if  (parameters_read == 3)
            {
                 printf ("\n[%s   %6.2f %6.2f]\n", keyword, parameter1,
                                                        parameter2);
                 WAYPOINTCONTROL = TRUE;
                 FOLLOWWAYPOINTMODE = TRUE;
                 HOVERCONTROL     = FALSE;
                 ROTATECONTROL    = FALSE;
                 LATERALCONTROL   = FALSE;
                 REPORTSTABLE     = TRUE;
                 DEADSTICKRUDDER  = FALSE;
                 x_command        = parameter1;
                 y_command        = parameter2;
          port_rpm_command        = fabs (port_rpm_command); /* ensure fwd  */
          stbd_rpm_command        = fabs (stbd_rpm_command); /* motion only */
                 detect_death_spiral (TRUE);   /* resets static variables  */

            if (TACTICALPARSE == FALSE) fprintf (auvordersfile,
"%6.1f  %6.1f %5.1f %5.1f %5.1f %6.1f %6.1f %6.1f %6.1f  %5.1f %5.1f %5.1f %5.1f\n",
                         t, psi_command, x_command, y_command, z_command,
                       port_rpm_command, stbd_rpm_command,
                         rudder_command,   planes_command,
                           bow_lateral_thruster_command,
                          stern_lateral_thruster_command,
                           bow_vertical_thruster_command,
                         stern_vertical_thruster_command);
            }
            else
            {
                 WAYPOINTCONTROL = FALSE;
                 printf ("\n warning: improper number of values\n waypoint");
                 printf ("set to current position but otherwise ignored\n");
                 x_command        = x;
                 y_command        = y;
                 z_command        = z;
            if (TACTICALPARSE == FALSE) fprintf (auvordersfile,
"%6.1f  %6.1f %5.1f %5.1f %5.1f %6.1f %6.1f %6.1f %6.1f  %5.1f %5.1f %5.1f %5.1f\n",
                         t, psi_command, x_command, y_command, z_command,
                       port_rpm_command, stbd_rpm_command,
                         rudder_command,   planes_command,
                             bow_lateral_thruster_command,
                          stern_lateral_thruster_command,
                           bow_vertical_thruster_command,
                         stern_vertical_thruster_command);
            }
            if  (FOLLOWWAYPOINTMODE)
            {
            /* continue until WAYPOINT reached without further script orders */
                time_next_command = t + dt;
                read_another_line = FALSE;
            }
        }
        else   if ((strcmp (keyword,  "WAYPOINTFOLLOW")      == 0) ||
                   (strcmp (keyword,  "WAYPOINT-FOLLOW")     == 0) ||
                   (strcmp (keyword,  "WAYPOINTFOLLOWON")    == 0) ||
                   (strcmp (keyword,  "WAYPOINT-FOLLOW-ON") == 0))
        {
            printf ("\n[%s]\n", keyword);
            FOLLOWWAYPOINTMODE = TRUE;
            DEADSTICKRUDDER    = FALSE;
        }
        else   if ((strcmp (keyword,  "WAYPOINTFOLLOWOFF")    == 0) ||
                   (strcmp (keyword,  "WAYPOINT-FOLLOW-OFF") == 0) )
        {
            printf ("\n[%s]\n", keyword);
            FOLLOWWAYPOINTMODE = FALSE;
        }
        else if ((strcmp (keyword, "STANDOFF")            == 0) ||
                 (strcmp (keyword, "STAND-OFF")           == 0) ||
                 (strcmp (keyword, "STANDOFFDISTANCE")    == 0) ||
                 (strcmp (keyword, "STANDOFF-DISTANCE")   == 0) ||
                 (strcmp (keyword, "STAND-OFF-DISTANCE") == 0))
```

```
{
        parameters_read = sscanf (command_buffer, "%s%lf",
                                                  keyword,    & parameter1);
        if  (parameters_read == 2)
        {
             printf ("\n[%s   %6.2f]\n", keyword, parameter1);
             standoff_distance = parameter1;

             if (HOVERCONTROL) /* report when stable again */
                 REPORTSTABLE  = TRUE;
        }
        else
        {
             printf ("\n[%s]\n", keyword);
             printf (" warning: no standoff value provided, ignored");
        }
}
else if ((strcmp (keyword, "HOVEROFF")  == 0) ||
         (strcmp (keyword, "HOVER-OFF") == 0) ||
         (strcmp (keyword, "HOVER_OFF") == 0))
{
        if (DISPLAYSCREEN) printf ("\n[HOVER-OFF] ");
        HOVERCONTROL       = FALSE;
        WAYPOINTCONTROL    = FALSE;  /* explicitly eliminate side effects */
        FOLLOWWAYPOINTMODE = FALSE;
        port_rpm_command   = 0.0;
        stbd_rpm_command   = 0.0;
        rudder_command     = 0.0;
        read_another_line  = FALSE;
}
else if ((strcmp (keyword, "HOVER")    == 0) ||
         (strcmp (keyword, "HOVER-ON") == 0))
{
        parameters_read = sscanf (command_buffer, "%s%lf%lf%lf%lf%lf",
                                                  keyword,    & parameter1,
                                                & parameter2, & parameter3,
                                                & parameter4, & parameter5);
        if  (parameters_read == 6)
        {
             printf ("\n[%s   %6.2f %6.2f %6.2f %6.2f %6.2f]\n",
                                                  keyword, parameter1,
                                                  parameter2, parameter3,
                                                  parameter4, parameter5);
             HOVERCONTROL       = TRUE;
             REPORTSTABLE       = TRUE;
             WAYPOINTCONTROL    = FALSE;
             ROTATECONTROL      = FALSE;
             LATERALCONTROL     = FALSE;
             THRUSTERCONTROL    = TRUE;
             DEADSTICKRUDDER    = TRUE;
           /* FOLLOWWAYPOINTMODE = FALSE; */
             rudder_command     = 0.0;
             x_command          = parameter1;
             y_command          = parameter2;
             z_command          = parameter3;
             psi_command        = parameter4;
             psi_command_hover  = parameter4;
             standoff_distance  = parameter5;
        if (TACTICALPARSE == FALSE) fprintf (auvordersfile,
"%6.1f  %6.1f %5.1f %5.1f %5.1f %6.1f %6.1f %6.1f %6.1f  %5.1f %5.1f %5.1f %5.1f\n",
                        t, psi_command, x_command, y_command, z_command,
                      port_rpm_command, stbd_rpm_command,
                        rudder_command,   planes_command,
                            bow_lateral_thruster_command,
                          stern_lateral_thruster_command,
                           bow_vertical_thruster_command,
                         stern_vertical_thruster_command);
        }
        else if  (parameters_read == 5)
        {
             printf ("\n[%s   %6.2f %6.2f %6.2f %6.2f]\n",
                                                  keyword, parameter1,
```

```
                                                parameter2, parameter3,
                                                parameter4);
                HOVERCONTROL       = TRUE;
                REPORTSTABLE       = TRUE;
                WAYPOINTCONTROL    = FALSE;
                ROTATECONTROL      = FALSE;
                LATERALCONTROL     = FALSE;
                THRUSTERCONTROL    = TRUE;
                DEADSTICKRUDDER    = TRUE;
              /* FOLLOWWAYPOINTMODE = FALSE; */
                rudder_command     = 0.0;
                x_command          = parameter1;
                y_command          = parameter2;
                z_command          = parameter3;
                psi_command        = parameter4;
                psi_command_hover = parameter4;
            if (TACTICALPARSE == FALSE) fprintf (auvordersfile,
"%6.1f  %6.1f %5.1f %5.1f %5.1f %6.1f %6.1f %6.1f %6.1f  %5.1f %5.1f %5.1f %5.1f\n",
                        t, psi_command, x_command, y_command, z_command,
                    port_rpm_command, stbd_rpm_command,
                      rudder_command,   planes_command,
                          bow_lateral_thruster_command,
                        stern_lateral_thruster_command,
                         bow_vertical_thruster_command,
                       stern_vertical_thruster_command);
            }
            else if  (parameters_read == 4)
            {
                printf ("\n[%s   %6.2f %6.2f %6.2f]\n", keyword, parameter1,
                                                parameter2, parameter3);
                HOVERCONTROL     = TRUE;
                REPORTSTABLE     = TRUE;
                WAYPOINTCONTROL = FALSE;
                ROTATECONTROL    = FALSE;
                LATERALCONTROL   = FALSE;
                THRUSTERCONTROL = TRUE;
                DEADSTICKRUDDER = TRUE;
              /* FOLLOWWAYPOINTMODE = FALSE; */
                rudder_command   = 0.0;
                x_command        = parameter1;
                y_command        = parameter2;
                z_command        = parameter3;
                psi_command_hover =  psi_command;
            if (TACTICALPARSE == FALSE) fprintf (auvordersfile,
"%6.1f  %6.1f %5.1f %5.1f %5.1f %6.1f %6.1f %6.1f %6.1f  %5.1f %5.1f %5.1f %5.1f\n",
                        t, psi_command, x_command, y_command, z_command,
                    port_rpm_command, stbd_rpm_command,
                      rudder_command,   planes_command,
                          bow_lateral_thruster_command,
                        stern_lateral_thruster_command,
                         bow_vertical_thruster_command,
                       stern_vertical_thruster_command);
            }
            else if  (parameters_read == 3)
            {
                printf ("\n[%s   %6.2f %6.2f]\n", keyword, parameter1,
                                                        parameter2);
                HOVERCONTROL     = TRUE;
                REPORTSTABLE     = TRUE;
                WAYPOINTCONTROL = FALSE;
                ROTATECONTROL    = FALSE;
                LATERALCONTROL   = FALSE;
                THRUSTERCONTROL = TRUE;
                DEADSTICKRUDDER = TRUE;
             /*  FOLLOWWAYPOINTMODE = FALSE; */
                rudder_command   = 0.0;
                x_command        = parameter1;
                y_command        = parameter2;
                psi_command_hover =  psi_command;
            if (TACTICALPARSE == FALSE) fprintf (auvordersfile,
"%6.1f  %6.1f %5.1f %5.1f %5.1f %6.1f %6.1f %6.1f %6.1f  %5.1f %5.1f %5.1f %5.1f\n",
                        t, psi_command, x_command, y_command, z_command,
```

```
                        port_rpm_command, stbd_rpm_command,
                          rudder_command,   planes_command,
                              bow_lateral_thruster_command,
                            stern_lateral_thruster_command,
                             bow_vertical_thruster_command,
                           stern_vertical_thruster_command);
                }
                else if  (parameters_read == 1)
                {
                     printf ("\n[%s]\n", keyword);
                     HOVERCONTROL     = TRUE;
                     REPORTSTABLE     = TRUE;
                     WAYPOINTCONTROL = FALSE;
                     ROTATECONTROL    = FALSE;
                     LATERALCONTROL   = FALSE;
                     THRUSTERCONTROL = TRUE;
                     DEADSTICKRUDDER = TRUE;
                 /*  FOLLOWWAYPOINTMODE = FALSE; */
                     rudder_command  = 0.0;
                     psi_command_hover =  psi_command;
                if (TACTICALPARSE == FALSE) fprintf (auvordersfile,
"%6.1f  %6.1f %5.1f %5.1f %5.1f %6.1f %6.1f %6.1f %6.1f  %5.1f %5.1f %5.1f %5.1f\n",
                          t, psi_command, x_command, y_command, z_command,
                        port_rpm_command, stbd_rpm_command,
                          rudder_command,   planes_command,
                              bow_lateral_thruster_command,
                            stern_lateral_thruster_command,
                             bow_vertical_thruster_command,
                           stern_vertical_thruster_command);
                }
                else
                {
                     printf ("\n[%s]\n", keyword);
                     printf (" warning: improper number of values, ignored\n");
                }
        }
        else   /* check other possibilities */
        {
                parse_mission_string_commands (command_buffer);
        }

        if (audible_command)
        {
           strcpy (buffer, command_buffer); /* copy current command to buffer */
           send_buffer_to_virtual_world_socket ();    /* send to sound driver */
        }

        if ((print_help) && DISPLAYSCREEN)
        {
           printf ("%s", command_buffer);
           print_valid_keywords ();

           strcpy (buffer, " is an unknown command"); /* copy msg to buffer   */
           send_buffer_to_virtual_world_socket ();    /* send to sound driver */
        }

        return_value = print_help;
        print_help = FALSE;        /* reset value */

        if (TACTICAL)
        {
           time_next_command = t + dt;        /* one command per timestep only */
           read_another_line = FALSE;         /* force acknowledgement and loop*/
           /* TIME and WAIT and RUN commands are not needed in TACTICAL mode   */
        }
        if ((HOVERCONTROL) || (WAYPOINTCONTROL))
        {
           read_another_line = FALSE;         /* force acknowledgement and loop*/
           /* TIME and WAIT and RUN commands are not needed in TACTICAL mode   */
        }
     } /* loop until read_another_line is FALSE) */
```

```
     if (TRACE && DISPLAYSCREEN)
        printf ("\n[end parse_mission_script_commands ()]\n");

     return (return_value);

} /* end parse_mission_script_commands () */

/*****************************************************************************/

void parse_mission_string_commands (command)
     char * command;
{
      int    index;
      int    number_values = 0;
      char   parameter_string [60];

      if (TRACE && DISPLAYSCREEN)
         printf ("\n[parse_mission_string_commands start]\n");

      number_values = sscanf (command_buffer, "%s", keyword);

      for (index = 0; index <= (int) strlen (keyword); index++)
             keyword [index] = toupper (keyword [index]);

      if      (number_values != 1)
      {
             if (DISPLAYSCREEN) printf (" [no parse word found]\n");
             return;
      }
      if ((strcmp (keyword,  "VIRTUALHOST")  == 0) ||
          (strcmp (keyword,  "VIRTUAL")      == 0) ||
          (strcmp (keyword,  "VIRTUAL-HOST") == 0) ||
          (strcmp (keyword,  "REMOTE")       == 0) ||
          (strcmp (keyword,  "REMOTEHOST")   == 0) ||
          (strcmp (keyword,  "REMOTE-HOST")  == 0) ||
          (strcmp (keyword,  "DYNAMICS")     == 0))
      {
           if (sscanf (command, "%s %s", keyword, parameter_string) == 2)
             {
                 strcpy (virtual_world_remote_host_name, parameter_string);
                 if (DISPLAYSCREEN)  printf ("\n[VIRTUAL-HOST %s] ",
                                          virtual_world_remote_host_name);
             }
             else print_help = TRUE;
      }
      else  print_help = TRUE;  /* invalid command line entry parameter found */

      if (TRACE && DISPLAYSCREEN)
          printf ("\n[parse_mission_string_commands complete]\n");

      return;

}/* end parse_mission_string_commands () */

/*****************************************************************************/

void print_valid_keywords ()
{

   if ((TACTICAL) || (TACTICALPARSE)) return;

printf ("\n");
printf ("                  [help] [trace|notrace] [loopforever|looponce]\n");
printf ("                  [wait #] [time #] [timestep (0.0..5.0)] [mission]\n");
printf ("                  [keyboard|keyboard-off] [quit] [kill]\n");
printf ("                  [rpm] [course] [depth] [thrusters|thrusters-off]\n");
printf ("                  [loopfilebackup] [entercontrolconstants]\n");
printf ("                  [rotate] [position|location|fix] [orientation]\n");
printf ("                  [gps|gps-fix] [gps-complete|gps-fix-complete] \n");
printf ("                  [sonartrace|sonartraceoff] [sonarinstalled]\n");
printf ("                  [trace|trace-off] [parallelporttrace] \n");
printf ("                  [remotehost hostname][realtime|nopause] [pause]\n");
```

```
printf ("                    [loop-forever|loop-once][entercontrolconstants]\n\n");
printf ("                    [silence][e-mail|no-email] [waypoint]\n\n");
printf ("See ~/execution/mission.script.HELP for command syntax details.\n");
printf ("\n");

#if (defined(sun) || defined(sgi))
   /* don't pop up help file if TACTICAL is running or invoking this code    */
   if ((HELPFILELAUNCHED == FALSE) && (TACTICAL == FALSE) &&
       (TACTICALPARSE    == FALSE))
   {
      printf ("popping up 'mission.script.HELP' as a zip file...\n");
      system ("zip -v ~/execution/mission.script.HELP");
      HELPFILELAUNCHED = TRUE;
   }
#endif

   return;

} /* end print_valid_keywords */

/*****************************************************************************/

void get_control_constants ()
                                        /* get data from file at program start */
{
     if (TACTICALPARSE) return;

     if (TRACE && DISPLAYSCREEN)
         printf ("\n[start get_control_constants ()]\n");

     if (ENTERCONTROLCONSTANTS)    /* - - - - - - - - - - - - - - - - - - - - - */
     {
         printf("Input start_dwell\n");
         scanf("%d", &start_dwell);

         /* note %F required by OS-9, accepted by SGI as equivalent to %lf   */
         printf("Input  k_psi, k_r, k_v\n");
         scanf("%F %F %F", &k_psi, &k_r, &k_v);

         printf("Input k_z, k_w, k_theta, and k_q\n");
         scanf("%F %F %F %F", &k_z, &k_w, &k_theta, &k_q);

         printf("Input k_thruster_psi,k_thruster_r\n");
         scanf("%F %F", &k_thruster_psi, &k_thruster_r);

         printf("Input k_thruster_rotate\n");
         scanf("%F",  &k_thruster_rotate);

         printf("Input k_thruster_z,k_thruster_w\n");
         scanf("%F %F", &k_thruster_z, &k_thruster_w);

         printf("Input k_propeller_hover, k_surge_hover, k_propeller_current\n");
         scanf("%F %F %F",&k_propeller_hover,&k_surge_hover,
                          &k_propeller_current);

         printf("Input k_thruster_hover, k_sway_hover, k_thruster_current\n");
         scanf("%F %F %F", &k_thruster_hover,
                        &k_thruster_hover, &k_thruster_current);

         printf("Input k_thruster_lateral\n");
         scanf("%F %F", &k_thruster_lateral,
                        &k_thruster_lateral);
     }
     else if (LOADCONTROLCONSTANTS)    /* - - - - - - - - - - - - - - - - - - - */
     {
        if ((controlconstantsinputfile = fopen (CONTROLCONSTANTSINPUTNAME,"r"))
            == NULL)
        {
        printf("AUV execution:  unable to open control constants input file ");
        printf("%s for reading.\n", CONTROLCONSTANTSINPUTNAME);
        printf
        ("                Check ownership permissions in current directory.\n");
```

```
    printf("Exit.\n");
    exit  (-1);
    }

    strcpy (buffer, "Control constants file is");

    if (TRACE && DISPLAYSCREEN)
    {
      printf ("\n[controlconstantsinputfile %s open, pointer = %x]\n",
                CONTROLCONSTANTSINPUTNAME, controlconstantsinputfile);
      send_buffer_to_virtual_world_socket (); /* buffer message sent */
    }

    strcpy (buffer, CONTROLCONSTANTSINPUTNAME);
    if (TRACE && DISPLAYSCREEN)
    {
     send_buffer_to_virtual_world_socket (); /* buffer message sent */
    }

     /* skip remaining header lines in file                                */
     for (i=1;i<=8;i++) fgets (local_buffer, 80, controlconstantsinputfile);

     /* note %F required by OS-9, accepted by SGI as equivalent to %lf   */
     fscanf (controlconstantsinputfile, "%F %F %F", &k_psi, &k_r, &k_v);
     fscanf (controlconstantsinputfile, "%F %F %F %F", &k_z, &k_w,
                                                  &k_theta, &k_q);

     for (i=1;i<=5;i++) fgets (local_buffer, 80, controlconstantsinputfile);
     fscanf(controlconstantsinputfile, "%F %F", &k_thruster_psi,
                                          &k_thruster_r);
     fscanf(controlconstantsinputfile, "%F",    &k_thruster_rotate);

     for (i=1;i<=5;i++) fgets (local_buffer, 80, controlconstantsinputfile);
     fscanf(controlconstantsinputfile, "%F %F", &k_thruster_z,
                                          &k_thruster_w);

     for (i=1;i<=5;i++) fgets (local_buffer, 80, controlconstantsinputfile);
     fscanf(controlconstantsinputfile, "%F %F %F", &k_propeller_hover,
                                             &k_surge_hover,
                                             &k_propeller_current);

     for (i=1;i<=5;i++) fgets (local_buffer, 80, controlconstantsinputfile);
     fscanf(controlconstantsinputfile, "%F %F %F", &k_thruster_hover,
                                             &k_sway_hover,
                                             &k_thruster_current);

     for (i=1;i<=5;i++) fgets (local_buffer, 80, controlconstantsinputfile);
     fscanf(controlconstantsinputfile, "%F",    &k_thruster_lateral);
}
else /* use default initialization values  - - - - - - - - - - - - - - - */
{
     if (TRACE && DISPLAYSCREEN)
        printf ("\n[using default control constant values]\n");

     start_dwell =   1;     /* delay time in seconds */

     k_psi       =    1.00; /* degrees rudder per degree of course error */
     k_r         =    2.00; /* degrees rudder per degree/sec yaw rate    */
     k_v         =    0.00; /* needed ?? */

     k_z         =   15.0;  /* degrees planes per foot of depth error    */
     k_w         =    2.0;
     k_theta     =    4.0;
     k_q         =    1.0;

     rpm               = 400.0;

     k_thruster_psi    =    0.6; /* volts per 1 degree course error       */
     k_thruster_r      =    5.0;
     k_thruster_rotate =    1.5; /* (24V)^2=> 2 # = 16.0 deg/sec empirical*/
                                 /* k_thruster_rotate=(24V / 16 deg/sec)^2*/
     k_thruster_z      =   10.0; /* guesses */
```

```
     k_thruster_w       =   80.0;

     k_propeller_hover =  200.0; /*  200 rpm per one  foot error          */

     k_surge_hover      = 6000.0; /*   60 rpm per 0.01 foot/sec surge      */

                                  /* this value is high to reduce sternway */

     k_propeller_current = 6500.0; /*   experimental */


     k_thruster_hover  =    4.0;
     k_sway_hover      =   40.0;
     k_thruster_current=   40.0;   /*   experimental */

     k_thruster_lateral=   48.0; /* 24 V = 2 # =  0.5  ft/sec empirically */
                                  /* note voltage follows a square law     */
}

if ((TRACE && DISPLAYSCREEN) || (ENTERCONTROLCONSTANTS) ||
                                (LOADCONTROLCONSTANTS))
{
    printf ("\n");
    printf ("\n[k_psi = %5.2f, k_r = %5.2f, k_v = %5.2f, k_z = %5.2f, ",
             k_psi,          k_r,          k_v,          k_z);
    printf ("k_w = %5.2f, k_theta = %5.2f, k_q = %5.2f]\n",
             k_w,          k_theta,          k_q);
    printf ("\n[k_thruster_psi = %5.2f, k_thruster_r = %5.2f, ",
             k_thruster_psi,          k_thruster_r);
    printf ("k_thruster_rotate = %5.2f, ",
             k_thruster_rotate);
    printf ("k_thruster_z = %5.2f, k_thruster_w = %5.2f]\n",
             k_thruster_z,          k_thruster_w);

    printf ("\n[k_propeller_hover = %5.2f, k_surge_hover = %5.2f]\n",
             k_propeller_hover,          k_surge_hover);

    printf ("\n[k_propeller_current = %5.2f]\n",
             k_propeller_current);

    printf ("\n[k_thruster_hover = %5.2f,  k_sway_hover = %5.2f]\n",
             k_thruster_hover,          k_sway_hover);

    printf ("\n[k_thruster_current = %5.2f]\n",
             k_thruster_current);

    printf ("\n[k_thruster_lateral = %5.2f]\n",k_thruster_lateral);
}

if ((controlconstantsoutputfile = fopen (CONTROLCONSTANTSOUTPUTNAME,"w"))
     == NULL)
{
    printf("AUV execution:  unable to open control constants output file ");
    printf("%s for writing.\n", CONTROLCONSTANTSOUTPUTNAME);
    printf
    ("                Check ownership permissions in current directory.\n");
    printf("Exit.\n");
    exit  (-1);
}
if (TRACE && DISPLAYSCREEN)
    printf ("\n[controlconstantsoutputfile %s open, pointer = %x]\n",
             CONTROLCONSTANTSOUTPUTNAME, controlconstantsoutputfile);

/* warning: the file read capability depends on file format/line spacing */
fprintf (controlconstantsoutputfile,
          " ____________________________________________________ \n\n");
fprintf (controlconstantsoutputfile,
          "   AUV execution level control algorithm coefficients \n");
fprintf (controlconstantsoutputfile,
          " ____________________________________________________ \n\n\n");
fprintf (controlconstantsoutputfile,
          "  k_psi    k_r    k_v    k_z    k_w    k_theta     k_q\n\n");
```

```
    fprintf (controlconstantsoutputfile,
             " %5.2f   %5.2f  %5.2f  %5.2f  %5.2f   %5.2f      %5.2f\n\n\n\n",
               k_psi,   k_r,   k_v,   k_z,   k_w,   k_theta,    k_q );

    fprintf (controlconstantsoutputfile,
             "  k_thruster_psi      k_thruster_r    k_thruster_rotate\n\n");
    fprintf (controlconstantsoutputfile,
             "      %5.2f              %5.2f              %5.2f\n\n\n\n",
               k_thruster_psi,     k_thruster_r,   k_thruster_rotate);

    fprintf (controlconstantsoutputfile,
             "  k_thruster_z        k_thruster_w \n\n");
    fprintf (controlconstantsoutputfile,
             "     %5.2f                %5.2f\n\n\n\n",
               k_thruster_z,       k_thruster_w);

    fprintf (controlconstantsoutputfile,
             "  k_propeller_hover   k_surge_hover   k_propeller_current\n\n");
    fprintf (controlconstantsoutputfile,
             "       %5.2f              %5.2f              %5.2f\n\n\n\n",
               k_propeller_hover,  k_surge_hover, k_propeller_current);

    fprintf (controlconstantsoutputfile,
             "  k_thruster_hover    k_sway_hover    k_thruster_current\n\n");
    fprintf (controlconstantsoutputfile,
             "     %5.2f                %5.2f                %5.2f\n\n\n\n",
               k_thruster_hover,   k_sway_hover, k_thruster_current);

    fprintf (controlconstantsoutputfile,
             "  k_thruster_lateral \n\n");
    fprintf (controlconstantsoutputfile,
             "     %5.2f\n\n\n\n",
               k_thruster_lateral);

    fflush (controlconstantsoutputfile); /* force completion of file write */
    fclose (controlconstantsoutputfile);

    if (TRACE && DISPLAYSCREEN)
        printf ("\n[finish get_control_constants ()]\n");

    return;

} /* end get_control_constants () */

/*****************************************************************************/
```

APPENDIX C - globals.c SOURCE CODE

```
/*****************************************************************************/
/*
 Program:          globals.c

 Authors:          Don Brutzman

 Revised:          13 February 96

 System:           AUV Gespac 68020/68030, OS-9 version 2.4
 Compiler:         Gespac cc Kernighan & Richie (K&R) C

 Compilation:      ftp>     put globals.c
                   auvsim1> chd execution
     [68020]       auvsim1> make -k2f execution
     [68030]       auvsim1> make      execution

     [Irix ]       fletch>  make execution

 Purpose:          Allows repeated use of global variables global.c via global.h
                   in order to prevent compiler warnings

                   See statevector.c/statevector.h for other global variables

*/

/*****************************************************************************/

#include "defines.h"
#include "globals.h"

/*****************************************************************************/
/* Program configuration flags                                               */

int    TRACE                    = 0;   /*  1=trace on,     0=trace off       */
int    DISPLAYSCREEN            = 1;   /*  1=screen on,    0=screen off      */

#if (defined(sgi) || defined(sun))
int    LOCATIONLAB              = 1;   /*  1=virtual world,0=actual vehicle  */
#else
int    LOCATIONLAB              = 0;   /*  1=virtual world,0=actual vehicle  */
#endif
int    TACTICAL                 = 0;   /*  1=tactical on,  0=tactical off    */
int    LOOPFOREVER              = 0;   /*  1=repeat execution indefinitely   */
int    LOOPFILEBACKUP           = 1;   /*  1=backup files between replications*/

int    PARALLELPORTTRACE        = 0;   /*  1=trace each char received at port */
int    SONARINSTALLED           = 0;   /*  1=sonar head available for query   */
int    SONARTRACE               = 0;   /*  1=trace on,     0=trace off        */
int    ENTERCONTROLCONSTANTS    = 0;   /*  1=manual entry, 0=default values   */
int    LOADCONTROLCONSTANTS     = 1;   /*  1=  file entry, 0=default values   */
int    REALTIME                 = 1;   /*  1=1 second real-time waits, 0=none */

int    DIVETRACKER              = 0;   /*  1=no dive tracker means abort      */

int    DEADRECKON               = 0;   /*  1=dead reckon navigate, 0=regular  */
int    DEADSTICKRUDDER          = 0;   /*  1=use ordered rudder, 0 = control  */
int    DEADSTICKPLANES          = 0;   /*  1=use ordered planes, 0 = control  */
int    SLIDINGMODECOURSE        = 0;   /*  1=use sliding mode,   0 = control  */

int    THRUSTERCONTROL          = 0;   /*  1=use thrusters, 0=use propellers  */
int    ROTATECONTROL            = 0;   /*  1=use thrusters to rotate in place */
int    LATERALCONTROL           = 0;   /*  1=use thrusters for lateral motion */
int    FOLLOWWAYPOINTMODE       = 0;   /*  1=   go to WAYPOINT without WAITs  */
int    WAYPOINTCONTROL          = 0;   /*  1=   go to WAYPOINT                */
int    HOVERCONTROL             = 0;   /*  1=hover at WAYPOINT                */

int    LEAK                     = 0;   /*  1=water leak in progress           */
```

```
int    HALTSCRIPT                = 0;   /*  1=automatic shutdown criteria met  */

#if defined(sgi)
int    EMAIL                     = 1;   /*  1=send e-mail, 0=don't send e-mail */
#else
int    EMAIL                     = 0;   /*  can't send email via OS-9 directly */
#endif
int    EMAIL_ENTERED             = 0;   /*  flag for first time through        */

int    NOT_YET_REIMPLEMENTED     = 0;   /*  code in block needs reverification */
int    ARCHAIC_IGNORE            = 0;   /*  code in block not valid, commented */

double TIMESTEP                  = 0.15; /*  time of a single closed loop       */
                                         /*  add code to warn if exceeded <<<<  */
                                         /*  units are seconds                  */

int    TACTICALPARSE             = 0;   /*  1=tactical level parsing commands  */
int    KEYBOARDINPUT             = 0;   /*  1=read keyboard vice mission file  */
int    HELPFILELAUNCHED          = 0;   /*  1=mission.script.HELP already shown*/
int    GPSFIXINPROGRESS          = 0;   /*  1=wait GPS-FIX & restore z_command */
int    REPORTSTABLE              = 0;   /*  1=tell when stable hover/waypt/gps */

/*****************************************************************************/
/* files and paths                                                           */

FILE * auvscriptfile;
FILE * auvordersfile;
FILE * auvdatafile;
FILE * auvtextfile;
FILE * controlconstantsinputfile;
FILE * controlconstantsoutputfile;
FILE * emailaddressfile;

/* FILE • serialtestfile;  */

 int   serialpath = 0;

 int   sonarpath  = 0;

/*****************************************************************************/
/* Variables and data structures                                             */

/* buffers of full strings for byte transfer to tactical level & disk file   */
/* 'buffer' usually < 256, intentionally oversized in case of overflow error */

   time_t          system_time             = 0;
   struct tm      *system_tmp              = 0;

   /* dac:  digital-analog converter                                         */
   /* adc:  analog-digital converter                                         */

   /*  4 Channels of DAC ADA-1 DAC -- updated                                */
   unsigned char  *dac1_a  = (unsigned char  *) DAC1_ADDR;

   /*  8 Channels of DAC DAC-2B -- updated                                   */
   unsigned char  *dac2b_a = (unsigned char  *) DAC2B_ADDR;

   /* 16 Channels of ADC ADA-1 -- updated                                    */
   unsigned char  *adc1_a  = (unsigned char  *) ADC1_ADDR;

   /* 16 Channels of ADC ADC-2 -- updated                                    */
   unsigned short *adc2_a  = (unsigned short *) ADC2_ADDR;

   unsigned char *via0    = (unsigned char *) VIA0_ADDR;
   unsigned char *via1    = (unsigned char *) VIA1_ADDR;

   unsigned char via0a_reg, via0b_reg;

   int             telemetry_records_saved = 0;
   int             mission_leg_counter     = 0;
   int             replication_count       = 1;
   int             end_test                = FALSE;
```

```
int             wrap_count              = 0;

double          dt                      = 0.15; /* units are seconds */
double          rpm                     = 0.0; /* +-700 rpm == +-2 ft/sec */
                                               /* (steady state)          */

double          dt_time                 ;      /* dive tracker time */

double          computer_voltage        = 24.0;
double          motor_voltage           = 24.0;

double          vertical_thruster_volts = 0.0; /* intermediate calculation */
double           lateral_thruster_volts = 0.0; /* intermediate calculation */

double          main_motor_delta1       = 0.0;
double          main_motor_delta2       = 0.0;
int             main_motor_volt1        = 512;
int             main_motor_volt2        = 512;

unsigned short  psi_bit_old             = 0;

double          dg_offset               = 0.0;

double          start_psi               = 0.0;   /* initial heading in degrees */

/* Used to estimate the X and Y position of the AUV */
double          X_est                   = 0.0;
double          Y_est                   = 0.0;
double          X_dot_est               = 0.0;
double          Y_dot_est               = 0.0;
double          u_est                   = 0.0;
double          v_est                   = 0.0;

/* control coefficients are based on standard units (degrees/feet/seconds) */
double          k_psi                   = 0.0;
double          k_r                     = 0.0;
double          k_v                     = 0.0;

double          k_z                     = 0.0;
double          k_w                     = 0.0;
double          k_theta                 = 0.0;
double          k_q                     = 0.0;

double          k_thruster_psi          = 0.0;
double          k_thruster_r            = 0.0;
double          k_thruster_rotate       = 0.0;
double          k_thruster_lateral      = 0.0;
double          k_thruster_z            = 0.0;
double          k_thruster_w            = 0.0;

double          k_propeller_hover       = 0.0;
double          k_surge_hover           = 0.0;
double          k_propeller_current     = 0.0;

double          k_thruster_hover        = 0.0;
double          k_sway_hover            = 0.0;
double          k_thruster_current      = 0.0;

double          k_sigma_r               = 12.0;
double          k_sigma_psi             = 28.87;
double          eta_steering            =  0.1;
double          sigma                   =  0.0;

int             mission_legs_total      = 0;

/*      values initialized in parse_mission_script_commands () */
double          time_next_command       = 0.0; /* units are seconds */
double          time_gps_complete       = 0.0; /* units are seconds */
double          time_postgps_dive       = 0.0; /* units are seconds */
double                psi_command       = 0.0; /* degrees      */
double                psi_command_hover = 0.0; /* degrees      */
double                  x_command       = 0.0; /* feet         */
```

```
double                y_command        = 0.0; /* feet          */
double                z_command        = 0.0; /* feet          */
double          stbd_rpm_command       = 0.0; /* -700..700     */
double          port_rpm_command       = 0.0; /* -700..700     */
double            planes_command       = 0.0; /* degrees       */
double            rudder_command       = 0.0; /* degrees       */
double            rotate_command       = 0.0; /* degrees/sec   */
double           lateral_command       = 0.0; /* ft/sec        */

double     bow_lateral_thruster_command = 0.0; /* volts -24..24 */
double   stern_lateral_thruster_command = 0.0; /* volts -24..24 */
double    bow_vertical_thruster_command = 0.0; /* volts -24..24 */
double  stern_vertical_thruster_command = 0.0; /* volts -24..24 */

double          previous_z_command      = 0.0; /* feet          */

double          gyro_error              = 0.0; /* degrees       */

double          waypoint_distance       = 0.0; /* feet          */
double          waypoint_angle          = 0.0; /* degrees       */

double          track_angle             =  0.0; /* degrees       */
double          along_track_distance    =  0.0; /* feet          */
double          cross_track_distance    =  0.0; /* feet          */
double          standoff_distance       =  1.0; /* feet          */

double          death_spiral_radius     =  15.0; /* feet         */

double          depth_error                   ; /* feet          */
double          depth_cell_bias         =  0.0; /* feet          */
double          psi_error               =  0.0; /* degrees       */

/* psi i-1 for differentiation of r needed because of busted gyro */
double          psi_im1                 = 0.0;  /* degrees       */

/* values used by kahlman depth filter */
int             kal_init_z              = TRUE;
double          thres_z                 = 1.0;
double          z_kal                   = 0.0;
double          z_dot_kal               = 0.0;
double          z_ddot_kal              = 0.0;

int             roll_rate_0             = 0;
int             pitch_rate_0            = 0;
int             yaw_rate_0              = 0;
int             roll_0                  = 0;
int             pitch_0                 = 0;
int             z_val0                  = 0;
int             sw1                     = 0;
int             error                   = 0;
int             range                   = 0;
int             bad_rng                 = 0;
int             bad_updates             = 0;
int             range_index             = 0;
double          range1                  = 0.0;
double          range2                  = 0.0;
double          error1                  = 0.0;
double          error2                  = 0.0;
double          avg_rng                 = 0.0;
int             k_range                 = 0;
int             range_array [3000];

int             pointer                 = NULL;
int             speed_array [11];

int             PortAFlag               = 0.0;

int             tick                    = 0;
int             curr_tick               = 0;
int             tick1                   = 0;
int             tick2                   = 0;
int             i                       = 0;
```

```
int             mask                    = 0x0000ffff;
long            davedate                = 0;
long            davetime                = 0;
double          value                   = 0.0;
short           day                     = 0;
char            answer                  = ' ';
int             start_dwell             = 0;

int             socket_descriptor       = 0;
int             socket_accepted         = 0;
int             socket_stream           = 0;

int             tactical_socket_descriptor = 0;
int             tactical_socket_accepted   = 0;
int             tactical_socket_stream     = 0;

int             socket_length           = MAXBUFFERSIZE;
int             buffer_max              = MAXBUFFERSIZE;

/* char         buffer_array [FILEBUFFERSIZE][256]; -- not implemented     */

char            buffer                  [MAXBUFFERSIZE + 10];
char            local_buffer            [MAXBUFFERSIZE + 10];

int             buffer_size             = 0;
int             buffer_index            = 0;
int             variables_parsed        = 0;

char            buffer_received         [MAXBUFFERSIZE + 10],
                virtual_world_remote_host_name [60],
                     tactical_remote_host_name [60],
                command_buffer          [MAXBUFFERSIZE + 10];

int             bytes_received          = 0;
int             bytes_read              = 0;
int             bytes_written           = 0;
int             bytes_left              = 0;
int             bytes_sent              = 0;

FILE          * netstat_fileptr;

struct sockaddr_in  server_address;

struct hostent *server_entity;

char            email_address [81];

int             shutdown_signal_received    = FALSE;
int             virtual_world_socket_opened = FALSE;
int                  tactical_socket_opened = FALSE;

char   * ptr_index;

int      print_help                 = FALSE;

double          speed_per_rpm = 2.0 / 700.0 ; /* steady state:
                                                 2.0 feet/sec per 700 rpm  */
                                              /* -700..700   */

clock_t   nextloopclock       = 0;
clock_t   currentloopclock       = 0;

int    audible_command    = TRUE;

int    auvscriptfilequit  = FALSE;

double AUV_oceancurrent_x = 0.0; /* Ocean current rate along North-axis */
double AUV_oceancurrent_y = 0.0; /* Ocean current rate along  East-axis */
double AUV_oceancurrent_z = 0.0; /* Ocean current rate along Depth-axis */

double DiveTracker1_x;           /* DiveTracker1 transducer x (feet)    */
double DiveTracker1_y;           /* DiveTracker1 transducer y (feet)    */
```

```
   double DiveTracker1_z;           /* DiveTracker1 transducer z (feet)    */

   double DiveTracker2_x;           /* DiveTracker2 transducer x (feet)    */
   double DiveTracker2_y;           /* DiveTracker2 transducer y (feet)    */
   double DiveTracker2_z;           /* DiveTracker2 transducer z (feet)    */

/*****************************************************************************/

   /* Dave's cats and dogs */

   unsigned char *tim_1ac1 = TIM_1AC_1;
   unsigned char *tim_1ac2 = TIM_1AC_2;
   unsigned char *tim_1ac3 = TIM_1AC_3;

   unsigned char tim_1a_data_reg      = TIM_1AC_DATA_REG;
   unsigned char tim_1a_control_reg   = TIM_1AC_CONTROL_REG;
   unsigned char tim_1a_aux_gates_reg = TIM_1AC_AUX_GATES_REG;

#ifndef sgi

   char *argblk[] = {
            "unlink",
             "DT2CL",
              0,
   };

   char *dt_fork_parmptr;
   int dt_pid;
   int ul_pid;

#endif
```

APPENDIX D - statevector.c SOURCE CODE

```
/*****************************************************************************/
/*
 Program:          statevector.c

                   State vector (telemetry variables) common definition

 Authors:          Don Brutzman and Mike Burns

 Revised:          27 January 96

 System:           AUV Gespac 68020/68030, OS-9 version 2.4
 Compiler:         Gespac cc Kernighan & Richie (K&R) C

 Compilation:      ftp>     put statevector.c
                   auvsim1> chd execution
     [68020]       auvsim1> make -k2f execution
     [68030]       auvsim1> make      execution

     [Irix ]       fletch>  make execution

 Purpose:          Allows repeated use of global variables in statevector.c
                   via statevector.h in order to prevent compiler warnings

                   See globals.c/globals.h for other global variables

 Religion:         All distance units are feet, all time units are seconds,
                   all rotational units are degrees.  This is only required
                   when transmitting values externally (socket/text/file).

                   Deciding factors are consistency and human readability.
                   Computational performance is not an issue.

                   Anyone who disagrees has to put up with an endless argument
                   from Don who will not be persuaded to accept any variations!

*/

/*****************************************************************************/

#include "defines.h"

/*****************************************************************************/

   int             STATEVECTORSIZE         = 37;  /* how many variables follow*/

   char            keyword [300];                 /* auv_state or uvw_state   */

   double          t                       = 0.0; /* units are seconds */
   double          x                       = 0.0; /* feet        */
   double          y                       = 0.0; /* feet        */
   double          z                       = 2.0; /* feet        */
   double          phi                     = 0.0; /* degrees     */
   double          theta                   = 0.0; /* degrees     */
   double          psi                     = 0.0; /* degrees     */
   double          x_dot                   = 0.0; /* feet/sec    */
   double          y_dot                   = 0.0; /* feet/sec    */
   double          z_dot                   = 0.0; /* feet/sec    */
   double          phi_dot                 = 0.0; /* degrees/sec */
   double          theta_dot               = 0.0; /* degrees/sec */
   double          psi_dot                 = 0.0; /* degrees/sec */
   double          speed                   = 0.0; /* feet/sec (paddlewheel) */
                                                  /*      possibly averaged */
   double          u                       = 0.0; /* feet/sec    */
   double          v                       = 0.0; /* feet/sec    */
   double          w                       = 0.0; /* feet/sec    */
   double          p                       = 0.0; /* degrees/sec */
   double          q                       = 0.0; /* degrees/sec */
```

```
double          r                       = 0.0; /* degrees/sec */
double          delta_planes            = 0.0; /* degrees     */
double          delta_rudder            = 0.0; /* degrees     */
double          port_rpm                = 0  ; /* -700..700   */
double          stbd_rpm                = 0  ; /* -700..700   */


/* +- 24V <=> +-2 lb, + Volts moves thruster in + direction, all identical */
double AUV_bow_vertical   = 0.0; /* thruster  rpm                          */
double AUV_stern_vertical = 0.0; /* thruster  rpm                          */
double AUV_bow_lateral    = 0.0; /* thruster  rpm                          */
double AUV_stern_lateral  = 0.0; /* thruster  rpm                          */

/* warning: do not use leading zero with bearings or else read as octal    */

double AUV_ST1000_bearing =   0.0;/* ST_1000 conical pencil bearing degrees*/
double AUV_ST1000_range   =  10.0;/* ST_1000 conical pencil range   feet   */
double AUV_ST1000_strength=  20.0;/* ST_1000 conical pencil strength dB    */

double AUV_ST725_bearing  =  90.0;/* ST_725  1 x 24 sector bearing  degrees*/
double AUV_ST725_range    =  20.0;/* ST_725  1 x 24 sector range    feet   */
double AUV_ST725_strength =  10.0;/* ST_725  1 x 24 sector strength  dB    */

double divetracker_range1 =  -1.0;/* feet range to divetracker unit 1      */
double divetracker_range2 =  -1.0;/* feet range to divetracker unit 2      */
                                  /* negative range means invalid return   */
                                  /* future:  divetracker_heading1 & 2     */

/*****************************************************************************/
```

APPENDIX E - external_functions.c SOURCE CODE

```
/*****************************************************************************/
/*
 Program:          external_functions.c

 Authors:          Don Brutzman

 Revised:          13 February 96

 System:           AUV Gespac 68020/68030, OS-9 version 2.4
 Compiler:         Gespac cc Kernighan & Richie (K&R) C

 Compilation:      ftp>     put external_functions.c
                   auvsim1> chd execution
     [68020]       auvsim1> make -k2f execution
     [68030]       auvsim1> make      execution

     [Irix ]       fletch>  make execution

 Purpose:          Reduce size of execution.c to allow OS-9 C compiler to work

                   Make functions globally available for tactical level
                      in order to guarantee compatibility
*/

/*****************************************************************************/
/* external_functions.c                                                      */

#include "globals.h"
#include "statevector.h"
#include "defines.h"

/*****************************************************************************/
/* OS-9 ~ specific function compatibility (mostly stubs to permit compilation */

#if (defined(sgi) || defined(sun))

void    tsleep   (unsigned svalue) { /* null body */}

void    _sysdate (format, time, date, day, tick)
                  int format, *time, *date, *tick; short *day;
                 { /* null body */}

double  pow      (xx, yy)
                  double xx, yy;
                 {return exp (yy * log (xx));}


int     _gs_rdy  (path) int path; { return 0; } /* bytes waiting on path */

int     _gs_opt  (path, buffer)
                  int path; struct sgbuf *buffer;
                 { return 0; }

int     _ss_opt  (path, buffer)
                  int path; struct sgbuf *buffer;
                 { return 0; }
#endif

/*****************************************************************************/

   double         degrees                          ();
   double         radians                          ();
   double         normalize                        ();
   double         normalize2                       ();
   double         radian_normalize                 ();
   double         radian_normalize2                ();
   void           clamp                            ();
```

```
#if (defined(sgi) || defined(sun))
#else
   double         atan2                                  ();
   double         sinh                                   ();
   double         cosh                                   ();
   double         tanh                                   ();
#endif

   double         sign                                   ();

   void           build_telemetry_string                 ();
   void           parse_telemetry_string                 ();

   void           open_tactical_socket                   ();
   void           shutdown_tactical_socket               ();
   void           send_buffer_to_tactical_socket         ();
   void           get_string_from_tactical_socket        ();

   void           record_data_on                         ();
   void           record_data_off                        ();

   void           cage_dg                                ();
   void           uncage_dg                              ();

   int            detect_death_spiral                    ();

   double         dsign                                  ();
   double         dtanh                                  ();


/*****************************************************************************/

double degrees (rads)     /* radians input */
        double  rads;
{
   return rads * 180.0 / PI;
}
/*****************************************************************************/

double radians (degs)     /* degrees input*/
        double  degs;
{
   return degs * PI / 180.0;
}
/*****************************************************************************/

double normalize (degs)     /* degrees input*/
        double  degs;
{
   double result = degs;

   while (result <    0.0) result += 360.0;
   while (result >= 360.0) result -= 360.0;

   return result;
}
/*****************************************************************************/

double normalize2 (degs)     /* degrees input*/
        double  degs;
{
   double result = degs;

   while (result <= -180.0) result += 360.0;
   while (result >   180.0) result -= 360.0;

   return result;
}
/*****************************************************************************/

double radian_normalize (rads)     /* radians input*/
        double  rads;
```

```
{
   double result = rads;

   while (result <       0.0) result += 2.0 * PI;
   while (result >= 2.0 * PI) result -= 2.0 * PI;

   return result;
}
/*****************************************************************************/

double radian_normalize2 (rads)     /* radians input*/
        double  rads;
{
   double result = rads;

   while (result <= - PI) result += 2.0 * PI;
   while (result >    PI) result -= 2.0 * PI;

   return result;
}
/*****************************************************************************/

void clamp (clampee, absolute_min, absolute_max, name)
            double * clampee;
            double   absolute_min;
            double   absolute_max;
            char   * name;
{
   double new_value, local_min, local_max;

   if ((absolute_max == 0.0) && (absolute_min == 0.0)) return;    /* no clamp */

   if (absolute_max >= absolute_min) /* ensure min & max used in proper order */
   {
       local_min = absolute_min;
       local_max = absolute_max;
   }
   else
   {
       local_min = absolute_max;
       local_max = absolute_min;
   }
   if ((* clampee) > local_max)
   {
       new_value = local_max;

       if (TRACE && DISPLAYSCREEN)
           printf ("[clamping %s from %5.3f to %5.3f]\n",
                           name, * clampee, new_value);

       * clampee = new_value;
   }
   if ((* clampee) < local_min)
   {
       new_value = local_min;

       if (TRACE && DISPLAYSCREEN)
           printf ("[clamping %s from %5.3f to %5.3f]\n",
                           name, * clampee, new_value);

       * clampee = new_value;
   }
}
/*****************************************************************************/
#if (defined(sgi) || defined(sun))
#else

/* thanks to Michael Olberg    Oct 20, 94    olberg@bele.oso.chalmers.se      */

double atan2 (y, x)
             double y; double x;
{
```

```
    if (TRACE && DISPLAYSCREEN)
          printf ("[atan2 (%5.3f, %5.3f)]\n", y, x);

    if (x == 0.0) {
        if (y < 0.0)      return(-PI/2.0);
        else              return( PI/2.0);
    } else {
        if (x < 0.0) {
            if (y < 0.0) return(atan(y/x)-PI);
            else         return(atan(y/x)+PI);
        } else           return(atan(y/x));
    }
}
    /* as to the tanh you will simply have to use */

double sinh (x)
       double x;
{
    return (exp(x) - exp(-x))/2.0;
}

double cosh (x)
       double x;
{
    return (exp(x) + exp(-x))/2.0;
}

double tanh (x)
       double x;
{
    return sinh(x)/cosh(x);
}

#endif

/*****************************************************************************/

double sign (x)
       double x;
{
    if      (x > 0.0) return  1.0;
    else if (x < 0.0) return -1.0;
    else              return  0.0;
}

/*****************************************************************************/

void build_telemetry_string (telemetry_buffer_ptr)

     char * telemetry_buffer_ptr;

{   /* this function parses global string buffer_received in execution.c      */

   if (TRACE && DISPLAYSCREEN) printf ("[begin build_telemetry_string ()]\n");

   for (i = 0; i <= buffer_max; i++) buffer [i] = '\0';        /* zero buffer */

   /* auv_state means state vector from auv that is going to virtual world    */

   if (LOCATIONLAB) z = z - depth_cell_bias;

   buffer_size = sprintf (telemetry_buffer_ptr,
 " auv_state %5.3f %5.3f %5.3f %5.3f %5.3f %5.3f %5.3f %5.3f %5.3f %5.3f %5.3f %5.3f %5.3f
%5.3f %5.3f %5.3f %5.3f %5.3f %5.3f %5.3f %5.3f %5.3f %5.3f %5.3f %5.3f %5.3f %5.3f %5.3f
%5.3f %5.3f %5.3f %5.3f %5.3f %5.3f %5.3f %5.3f %5.3f %5.3f %5.3f %5.3f %5.3f %5.3f %5.3f \n",
           t,x,y,z,
           normalize2 (phi  ),
           normalize2 (theta),
           normalize  (psi  ),
           speed,
           u, v, w,
           normalize2 (p),
```

```
            normalize2 (q),
            normalize2 (r),
            x_dot, y_dot, z_dot,
            normalize2 (phi_dot  ),
            normalize2 (theta_dot),
            normalize2 (psi_dot  ),
            normalize2 (delta_rudder),
            normalize2 (delta_planes),
            port_rpm,                stbd_rpm,
            AUV_bow_vertical,        AUV_stern_vertical,
            AUV_bow_lateral,         AUV_stern_lateral,
            AUV_ST1000_bearing,      AUV_ST1000_range,    AUV_ST1000_strength,
            AUV_ST725_bearing,       AUV_ST725_range,     AUV_ST725_strength,
            divetracker_range1,      divetracker_range2);

   if (LOCATIONLAB) z = z + depth_cell_bias;

   if (buffer_size > buffer_max)     /* sprintf buffer overflow condition      */
   {
       if (DISPLAYSCREEN)
           printf ("Buffer overflow, buffer_size = %d, reduced to %d !!!!!!\n",
                                     buffer_size, buffer_max);
       buffer_size  =      buffer_max;
   }

   if (TRACE && DISPLAYSCREEN) printf ("[buffer_size is %d]\n", buffer_size);


   /* other state variables & timing constraints can be added <<<<<<<<<<<<<<< */

   if (TRACE && DISPLAYSCREEN) printf ("[finish build_telemetry_string ()]\n");

   return;
}
/*****************************************************************************/

void parse_telemetry_string (passed_buffer_ptr)

     char * passed_buffer_ptr;

{   /* this function parses global string buffer_received in execution.c      */

    /* temporary hold variables */

    double  AUV_time_temp,
            AUV_x_temp,                  AUV_y_temp,            AUV_z_temp,
            AUV_phi_temp,                AUV_theta_temp,        AUV_psi_temp,
            AUV_speed_temp,
            AUV_u_temp,                  AUV_v_temp,            AUV_w_temp,
            AUV_p_temp,                  AUV_q_temp,            AUV_r_temp,
            AUV_x_dot_temp,              AUV_y_dot_temp,        AUV_z_dot_temp,
            AUV_phi_dot_temp,            AUV_theta_dot_temp,    AUV_psi_dot_temp,
            AUV_delta_rudder_temp,       AUV_delta_planes_temp,
            AUV_port_rpm_temp,           AUV_stbd_rpm_temp,
            AUV_bow_vertical_temp,       AUV_stern_vertical_temp,
            AUV_bow_lateral_temp,        AUV_stern_lateral_temp,
            AUV_ST1000_bearing_temp,     AUV_ST1000_range_temp,
            AUV_ST1000_strength_temp,    AUV_ST725_bearing_temp,
            AUV_ST725_range_temp,        AUV_ST725_strength_temp,
            AUV_divetracker_range1_temp, AUV_divetracker_range2_temp;

   if (TRACE && DISPLAYSCREEN) printf ("[begin parse_telemetry_string ()]\n");

   /* update and output AUV state variables                                  */
   /* note that if dead reckoning is used, values will not change            */

   /* note %F required by OS-9, accepted by SGI as equivalent to %lf          */

   if (TRACE && DISPLAYSCREEN)
       printf ("from telemetry buffer:\n%s", passed_buffer_ptr);

   variables_parsed = sscanf(passed_buffer_ptr,
```

```
/*        "%s %lf %lf %lf %lf %lf %lf %lf %lf %lf %lf %lf %lf %lf %lf %lf %lf %lf %lf %lf
%lf %lf %lf %lf %lf %lf %lf %lf %lf %lf %lf %lf %lf %lf %lf %lf %lf %lf %lf %lf %lf %lf\n",*/
          "%s %F %F %F %F %F %F %F %F %F %F %F %F %F %F %F %F %F %F %F %F %F %F %F %F %F
%F %F %F %F %F %F %F %F %F %F\n",
            keyword,                    &AUV_time_temp,
           &AUV_x_temp,                 &AUV_y_temp,              &AUV_z_temp,
           &AUV_phi_temp,               &AUV_theta_temp,          &AUV_psi_temp,
           &AUV_speed_temp,
           &AUV_u_temp,                 &AUV_v_temp,              &AUV_w_temp,
           &AUV_p_temp,                 &AUV_q_temp,              &AUV_r_temp,
           &AUV_x_dot_temp,             &AUV_y_dot_temp,          &AUV_z_dot_temp,
           &AUV_phi_dot_temp,           &AUV_theta_dot_temp,      &AUV_psi_dot_temp,
           &AUV_delta_rudder_temp,      &AUV_delta_planes_temp,
           &AUV_port_rpm_temp,          &AUV_stbd_rpm_temp,
           &AUV_bow_vertical_temp,      &AUV_stern_vertical_temp,
           &AUV_bow_lateral_temp,       &AUV_stern_lateral_temp,
           &AUV_ST1000_bearing_temp,    &AUV_ST1000_range_temp,
                                        &AUV_ST1000_strength_temp,
           &AUV_ST725_bearing_temp,     &AUV_ST725_range_temp,
                                        &AUV_ST725_strength_temp,
           &AUV_divetracker_range1_temp, &AUV_divetracker_range2_temp);

   if (variables_parsed == STATEVECTORSIZE)  /* transfer OK, keep new values */
   {
      t                    =        (AUV_time_temp);
      x                    =        (AUV_x_temp);
      y                    =        (AUV_y_temp);
      z                    =        (AUV_z_temp);
      phi                  =        (AUV_phi_temp);
      theta                =        (AUV_theta_temp);
      psi                  =        (AUV_psi_temp);
      speed                =        (AUV_speed_temp);
      u                    =        (AUV_u_temp);
      v                    =        (AUV_v_temp);
      w                    =        (AUV_w_temp);
      p                    =        (AUV_p_temp);
      q                    =        (AUV_q_temp);
      r                    =        (AUV_r_temp);
      x_dot                =        (AUV_x_dot_temp);
      y_dot                =        (AUV_y_dot_temp);
      z_dot                =        (AUV_z_dot_temp);
      phi_dot              =        (AUV_phi_dot_temp);
      theta_dot            =        (AUV_theta_dot_temp);
      psi_dot              =        (AUV_psi_dot_temp);
      delta_rudder         =        (AUV_delta_rudder_temp);
      delta_planes         =        (AUV_delta_planes_temp);
      port_rpm             =        (AUV_port_rpm_temp);
      stbd_rpm             =        (AUV_stbd_rpm_temp);
      AUV_bow_vertical     =        (AUV_bow_vertical_temp);
      AUV_stern_vertical   =        (AUV_stern_vertical_temp);
      AUV_bow_lateral      =        (AUV_bow_lateral_temp);
      AUV_stern_lateral    =        (AUV_stern_lateral_temp);
      AUV_ST1000_bearing   =normalize(AUV_ST1000_bearing_temp);
      AUV_ST1000_range     =        (AUV_ST1000_range_temp);
      AUV_ST1000_strength  =        (AUV_ST1000_strength_temp);
      AUV_ST725_bearing    =normalize(AUV_ST725_bearing_temp);
      AUV_ST725_range      =        (AUV_ST725_range_temp);
      AUV_ST725_strength   =        (AUV_ST725_strength_temp);
      divetracker_range1   =        (AUV_divetracker_range1_temp);
      divetracker_range2   =        (AUV_divetracker_range2_temp);
   }
   else if ((variables_parsed != STATEVECTORSIZE) && (variables_parsed != -1))
   {
       strcpy (command_buffer, passed_buffer_ptr);
       parse_mission_script_commands ();

   /*    if (DISPLAYSCREEN)
  printf ("\nGarble problem in buffer_received !!! variables parsed = %d\n%s\n",
              variables_parsed, passed_buffer_ptr);
       TRACE = TRUE; */
   }
/*
```

```
   if (TRACE && DISPLAYSCREEN)
   {
       printf ("\nfrom telemetry buffer state variables:");
       printf ("\n %s t=%5.3f x=%5.3f y=%5.3f z=%5.3f ",
           keyword,              t,
           x,                    y,                    z);
       printf ("phi=%5.3f theta=%5.3f psi=%5.3f ",
           phi,                  theta,                psi);
       printf ("paddlewheel speed=%5.3f ",
           speed);
       printf ("u=%5.3f v=%5.3f w=%5.3f p=%5.3f q=%5.3f r=%5.3f ",
           u,                    v,                    w,
           p,                    q,                    r);
       printf ("x_dot=%5.3f y_dot=%5.3f z_dot=%5.3f ",
           x_dot,                y_dot,                z_dot);
       printf ("phi_dot=%5.3f theta_dot=%5.3f psi_dot=%5.3f ",
           phi_dot,              theta_dot,            psi_dot);
       printf ("delta_rudder=%5.3f delta_planes=%5.3f ",
           delta_rudder,         delta_planes);
       printf ("port_rpm=%5.3f stbd_rpm=%5.3f ",
           port_rpm,             stbd_rpm);
       printf ("bow_vertical=%5.3f stern_vertical=%5.3f ",
           AUV_bow_vertical,     AUV_stern_vertical);
       printf ("bow_lateral=%5.3f stern_lateral=%5.3f ",
           AUV_bow_lateral,      AUV_stern_lateral);
       printf ("ST1000 b/r/s %5.3f %5.3f %5.3f, ST725 b/r/s %5.3f %5.3f %5.3f",
           AUV_ST1000_bearing,   AUV_ST1000_range,    AUV_ST1000_strength,
           AUV_ST725_bearing,    AUV_ST725_range,     AUV_ST725_strength);
       printf ("divetracker_range1=%5.3f divetracker_range2=%5.3f ",
           divetracker_range1,   divetracker_range2);
       printf (", [current time %d %d %d] \n",
           system_tmp->tm_hour, system_tmp->tm_min, system_tmp->tm_sec);
   }*/

   /* keep all telemetry variables in degrees */
   phi          = normalize2 (phi  );
   theta        = normalize2 (theta);
   psi          = normalize  (psi  );
   phi_dot      = normalize2 (phi_dot);
   theta_dot    = normalize2 (theta_dot);
   psi_dot      = normalize2 (psi_dot);
   p            = normalize2 (p);
   q            = normalize2 (q);
   r            = normalize2 (r);
   delta_rudder = normalize2 (delta_rudder);
   delta_planes = normalize2 (delta_planes);

   if (TRACE && DISPLAYSCREEN) printf ("[finish parse_telemetry_string ()]\n");

   return;
}
/*****************************************************************************/

void open_tactical_socket ()        /* see os9sender.c for original code */
{
   if (TRACE && DISPLAYSCREEN)
       printf ("[start open_tactical_socket ()]\n");

/* Initialize communications blocks                                          */

/* Initialize both client & server ******************************************/

   /* Signal handlers for termination to override net_open () and net_close ()*/
   /*    signal handlers.  Otherwise you are unable to ^C kill this program.  */

#if (defined(sgi) || defined(sun))
signal (SIGHUP,  shutdown_tactical_socket);/* hangup                     */
signal (SIGINT,  shutdown_tactical_socket);/* interrupt character        */
signal (SIGKILL, shutdown_tactical_socket);/* kill signal from Unix      */
signal (SIGPIPE, shutdown_tactical_socket);/* broken pipe from other host*/
signal (SIGTERM, shutdown_tactical_socket);/* software termination       */
#endif
```

```
/* Initialize sender ********************************************************/

   /* start by finding default/desired remote host to connect to              */
   {

      server_entity = gethostbyname (tactical_remote_host_name);
      if (server_entity == NULL)
      {
         if (DISPLAYSCREEN)
         {
           printf("[open_tactical_socket:  tactical remote host\n");
           printf(" (\"%s\") not found]\n",
                   tactical_remote_host_name);
           fflush (stdout); /* force completion of screen write */
         }
         /* error message needed on (open) output file   <<<<<<<<<<<<<<<<<<< */
         tactical_socket_opened = FALSE;
         exit (1);
      }
      else if (TRACE && DISPLAYSCREEN)
           {
              printf("[open_tactical_socket: tactical remote host ");
              printf("(\"%s\") located]\n",  tactical_remote_host_name);
           }

      /* Client opens server port ********************************************/

      /* Fill in structure 'server_address' with the address of the           */
      /*         remote host (i.e. SERVER) that we want to connect with:      */

#if defined(sgi)
      bzero ((char *) &server_address, sizeof (server_address));
#endif
      server_address.sin_family   = AF_INET;      /* Internet protocol family */

      /* copy server IP address into sockaddr_in struct server_address        */
#if defined(sgi)
      bcopy (server_entity->h_addr, &(server_address.sin_addr.s_addr),
             server_entity->h_length);
#else
      strncpy(&(server_address.sin_addr.s_addr), server_entity->h_addr,
             server_entity->h_length);
#endif

      /* make sure port is in network byte order                               */
      server_address.sin_port = htons (AUVSIM1_TCP_PORT_2);

      /* Open TCP (Internet stream) socket                                     */
      if ( (tactical_socket_descriptor = socket (AF_INET, SOCK_STREAM, 0)) < 0 )
      {
         if (DISPLAYSCREEN)
         {
             printf ("[open_tactical_socket: client can't open server");
             printf (" tactical stream socket]");
         }
         /* error message needed on (open) output file <<<<<<<<<<<<<<<<<<<<<< */
         tactical_socket_opened = FALSE;
         exit (1);
      }
      else if (TRACE && DISPLAYSCREEN)
           {
              printf ("[open_tactical_socket: client opened");
              printf (" tactical server socket successfully]\n");
           }

      /* Connect to the server.  Process will block/sleep until connection is
            is established.  Timeout will return an error.                  */
      if (connect (                  tactical_socket_descriptor,
                   (struct sockaddr *) &server_address,
                                 sizeof (server_address)) < 0)
      {
         if (DISPLAYSCREEN)
```

```
        {
            printf ("[open_tactical_socket: client can't connect to");
            printf (" tactical server socket, tactical comms ignored]\n");
        }
        /* error message needed on (open) output file <<<<<<<<<<<<<<<<<<<<<< */
        tactical_socket_opened = FALSE;

        if      (LOCATIONLAB) exit (1);
        else if (DISPLAYSCREEN) printf ("[continuing anyway]");
      }
      else
      {
           if (TRACE && DISPLAYSCREEN)
           {
              printf ("[execution client connected to tactical");
              printf (" server socket successfully]\n");
           }
           tactical_socket_opened = TRUE;
      }

   } /* end initialization                                                    */

   tactical_socket_stream = tactical_socket_descriptor;   /* client */

   if (TRACE && DISPLAYSCREEN)  /* print final info */
   {
     printf("[open_tactical_socket CLIENT: tactical_socket_descriptor = %d]\n",
                                          tactical_socket_descriptor);
     printf("[                            tactical_socket_accepted   = %d]\n",
                                          tactical_socket_accepted);
     printf("[                            tactical_socket_stream     = %d]\n",
                                          tactical_socket_stream);
   }

   if (TRACE && DISPLAYSCREEN)
       printf ("[finish open_tactical_socket ()]\n");

   return;

}/* end open_tactical_socket () */

/*****************************************************************************/

void shutdown_tactical_socket ()    /* see os9sender.c for original code */
{
   int  kill_return_value;

   shutdown_signal_received = TRUE;

   if (tactical_socket_opened == FALSE)
   {
       if (TRACE && DISPLAYSCREEN)
       {
          printf ("[tactical_socket_opened FALSE,");
          printf (" shutdown_tactical_socket ignored]\n");
       }
       return;
   }
   if (TRACE && DISPLAYSCREEN)
       printf ("[shutdown_tactical_socket start ...]\n");

   /* No need to send a message to other side that bridge is going down,      */
   /*    since SIGPIPE signal trigger may shutdown server on other side       */

   if (close (tactical_socket_stream) == -1)
   {
       if (TRACE && DISPLAYSCREEN)
        printf ("shutdown_tactical_socket close (tactical_socket_stream) failed\n");

       /* shutdown () reference:  "Using OS-9 Internet" manual p. 2-55        */

       if (shutdown (tactical_socket_stream, 2) == -1)
```

```
        {
            if (TRACE && DISPLAYSCREEN)
            {
                printf ("[shutdown_tactical_socket shutdown");
                printf (" (tactical_socket_stream, 2) failed]\n");
            }

            kill_return_value = kill (tactical_socket_stream, SIGKILL);

            if (TRACE && DISPLAYSCREEN)
            {
                printf ("[shutdown_tactical_socket kill (tactical_socket_stream,");
                printf (" SIGKILL) returned %d]\n", kill_return_value);
            }
        }
    }
    if (TRACE && DISPLAYSCREEN)
        printf ("[shutdown_tactical_socket return]\n");

    return;

} /* end shutdown_tactical_socket () */

/*****************************************************************************/

void send_buffer_to_tactical_socket () /* see os9sender.c for orig. code */
{
    if (HALTSCRIPT) return;

    bytes_left        = socket_length;
    bytes_written     = 0;
    ptr_index         = buffer;  /* this global string is the data to be sent  */

    if (tactical_socket_opened == FALSE)
    {
        if (TRACE && DISPLAYSCREEN)
        {
            printf ("[send_buffer_to_tactical_socket: ");
            printf ("tactical_socket_opened == FALSE, returning]\n");
        }
        return;
    }
    if (TRACE && DISPLAYSCREEN)
        printf ("[send_buffer_to_tactical_socket start ...]\n");

    while ((bytes_left > 0) && (bytes_written >= 0))  /* write loop ***********/
    {
        bytes_sent = write (tactical_socket_stream, ptr_index, bytes_left);

        if      (bytes_sent <  0) bytes_written = bytes_sent;
        else if (bytes_sent >  0)
             {
                 bytes_left     -= bytes_sent;
                 bytes_written  += bytes_sent;
                 ptr_index      += bytes_sent;
             }
        if (LOCATIONLAB && TRACE && DISPLAYSCREEN)
        {
              printf ("[record_data send_telemetry_to_server loop");
              printf ("  bytes sent = %d]\n", bytes_sent);
        }
    }
    if   (bytes_written < 0)
    {

        HALTSCRIPT = TRUE;     /* loss of socket comms with tactical level */

        if (LOCATIONLAB && DISPLAYSCREEN)
        {
          printf ("[record_data send_telemetry_to_server () send failed, ");
          printf ("%d bytes_written]\n", bytes_written);
        }
```

```
        /* error message needed on (open) output file   <<<<<<<<<<<<<<<<<<<<<< */
    }
    else if (LOCATIONLAB && TRACE && DISPLAYSCREEN)
         {
             printf ("[record_data send_telemetry_to_server total bytes sent");
             printf (" = %d]\n", bytes_written);
         }

    /* Check termination ******************************************************/

    if (strncmp (buffer, "shutdown", 8) == 0)
    {
        if (TRACE && DISPLAYSCREEN) printf
             ("[send_buffer_to_tactical_socket: shutdown_signal_received]\n");
        shutdown_tactical_socket ();
    }
    if (TRACE && DISPLAYSCREEN)
        printf ("[send_buffer_to_tactical_socket return]\n");

    return;

} /* end send_buffer_to_tactical_socket () */

/*****************************************************************************/

void get_string_from_tactical_socket ()  /* see os9sender.c for original */
{
    if (tactical_socket_opened == FALSE)
    {
        if (TRACE && DISPLAYSCREEN)
        printf ("[get_string_from_tactical_socket not open, ignored]\n");
        return;
    }
    if (TRACE && DISPLAYSCREEN)
        printf ("[get_string_from_tactical_socket start ...]\n");

    /* listen to remote host, relay to local network/program                  */

    bytes_left        = socket_length;
    bytes_received    = 0;
    ptr_index         = command_buffer; /* command_buffer is where results go */

    while ((bytes_left > 0) && (bytes_received >= 0))  /* read loop ************/
    {
       bytes_read = read (tactical_socket_stream, ptr_index, bytes_left);

       if      (bytes_read <  0) bytes_received = bytes_read;
       else if (bytes_read >  0)
       {
            bytes_left     -= bytes_read;
            bytes_received += bytes_read;
            ptr_index      += bytes_read;
       }
       if (TRACE && DISPLAYSCREEN)
       {
           printf ("[get_string_from_tactical_socket receiver block");
           printf (" loop bytes_read = %d]\n", bytes_read);
       }
       /* if nothing is waiting to be read, break out of read loop            */
       if ((bytes_read == 0) && (bytes_received == 0)) break;
    }
    if  (bytes_received < 0)      /* failure     */
    {
         if (TRACE && DISPLAYSCREEN)
         {
            printf ("[get_string_from_tactical_socket receiver block read");
            printf (" failed, bytes_received = %d\n", bytes_received);
         }
    }
    else if (bytes_received == 0)  /* no transfer */
    {
         if (TRACE && DISPLAYSCREEN)
```

```
        {
          printf("[get_string_from_tactical_socket received 0 bytes!!]\n");
        }
    }
    else if (bytes_received > 0)  /* success     */
        {
            if (TRACE && DISPLAYSCREEN)
            {
            printf("[get_string_from_tactical_socket received %d bytes]\n",
                    bytes_received);
            }
        }

    /* Check termination ******************************************************/

    if (strncmp (command_buffer, "shutdown", 8) == 0)
    {
        if (TRACE && DISPLAYSCREEN) printf
            ("[get_data_on_tactical_socket: shutdown_signal_received]\n");
        shutdown_tactical_socket ();
    }
    if (TRACE && DISPLAYSCREEN)
        printf ("[get_string_from_tactical_socket finish]\n");

    return;

} /* end get_string_from_tactical_socket () */

/*****************************************************************************/

void record_data_on ()
{
     if (TRACE && DISPLAYSCREEN) printf ("[start  record_data_on ()]\n");

     /* Open files for writing                                              */

     if ((TACTICALPARSE) || (TACTICAL == FALSE))
       if ((auvdatafile = fopen (AUVDATAFILENAME,"w")) == NULL)
       {
         printf("record_data_on () unable to open output file %s for writing.",
                AUVDATAFILENAME);
         printf
        ("                Check ownership permissions in current directory.\n");
         printf("Exit.\n");
         exit  (-1);
       }
     if (TRACE && DISPLAYSCREEN && (auvdatafile != NULL))
         printf ("[auvdatafile %s open,   pointer = %x]\n",
                              AUVDATAFILENAME, auvdatafile);

     if ((TACTICALPARSE) || (TACTICAL == FALSE) || (auvdatafile != NULL))
     {
     fprintf (auvdatafile, "# auvdatafile %s shows %d ", AUVDATAFILENAME,STATEVECTORSIZE);
     fprintf (auvdatafile, "state vector variables at %3.1f intervals.\n\n", dt);

     fprintf (auvdatafile, "# state                                          paddle ");
     fprintf (auvdatafile, "                                  ");
     fprintf (auvdatafile, "                  phi   theta psi   ");
     fprintf (auvdatafile, "delta  delta  port  stbd  ");

     fprintf (auvdatafile, "bow_  stern bow_  stern  ");
     fprintf (auvdatafile, "_ST1000 sonar__     ");
     fprintf (auvdatafile, "_ST725  sonar__    ");
     fprintf (auvdatafile, "Dive  Dive");
     fprintf (auvdatafile, "\n");

     fprintf (auvdatafile, "# vector   t     x      y      z   phi   theta psi   speed ");
     fprintf (auvdatafile, "u     v     w     p     q     r     ");
     fprintf (auvdatafile, "x_dot y_dot z_dot _dot  _dot  _dot  ");
     fprintf (auvdatafile, "rudder plane  rpm   rpm   ");

     fprintf (auvdatafile, "vrtcl vrtcl latrl latrl  ");
```

```
     fprintf (auvdatafile, "bng   range  dB     ");
     fprintf (auvdatafile, "bng   range  dB    ");
     fprintf (auvdatafile, "Trk1  Trk2");
     fprintf (auvdatafile, "\n\n");
     }

     if ((auvtextfile = fopen (AUVTEXTFILENAME,"w")) == NULL)
     {
        printf("record_data_on () unable to open output file %s for writing.  ",
                 AUVTEXTFILENAME);
        printf
        ("                Check ownership permissions in current directory.\n");
        printf("Exit.\n");
        exit  (-1);
     }
     if (TRACE && DISPLAYSCREEN)
         printf ("[auvtextfile %s open, pointer = %x]\n",
                              AUVTEXTFILENAME, auvtextfile);

     fprintf (auvtextfile, "# auvtextfile %s shows %d ", AUVDATAFILENAME,STATEVECTORSIZE);
     fprintf (auvtextfile, "state vector variables at %3.1f intervals.\n\n", dt);

     fprintf (auvtextfile, "# state                                                  paddle ");
     fprintf (auvtextfile, "                                          ");
     fprintf (auvtextfile, "                  phi   theta psi   ");
     fprintf (auvtextfile, "delta  delta  port  stbd  ");

     fprintf (auvtextfile, "bow_  stern bow_  stern  ");
     fprintf (auvtextfile, "_ST1000 sonar__     ");
     fprintf (auvtextfile, "_ST725  sonar__    ");
     fprintf (auvtextfile, "Dive  Dive");
     fprintf (auvtextfile, "\n");

     fprintf (auvtextfile, "# vector   t     x      y      z   phi   theta psi   speed ");
     fprintf (auvtextfile, "u     v     w     p     q     r     ");
     fprintf (auvtextfile, "x_dot y_dot z_dot _dot  _dot  _dot  ");
     fprintf (auvtextfile, "rudder plane  rpm   rpm   ");

     fprintf (auvtextfile, "vrtcl vrtcl latrl latrl  ");
     fprintf (auvtextfile, "bng   range  dB     ");
     fprintf (auvtextfile, "bng   range  dB    ");
     fprintf (auvtextfile, "Trk1  Trk2");
     fprintf (auvtextfile, "\n\n");

     if (LOOPFOREVER)
         fprintf (auvtextfile, "# Mission replication #%d\n",
                              replication_count);

/* testing code from wr2t1.c, not currently in use         */
/* serial.d is a telemetry test file to check connectivity */
/*   if ((serialtestfile = fopen ("serial.d", "r")) <= 0)
     {
      printf("record_data_on () can't open test file serial.d\n");
      printf("Exit.\n");
      exit  (-1);
     }
*/
     if (TRACE && DISPLAYSCREEN) printf ("[finish record_data_on ()]\n");

     return;
}

/*****************************************************************************/

void record_data_off ()

{
     if  (TRACE && DISPLAYSCREEN) printf ("[start  record_data_off ()]\n");

     if  ((auvdatafile != NULL) && TRACE && DISPLAYSCREEN)
     {
          printf ("[flushing and closing auvdatafile %s   %x]\n",
```

```
                                                AUVDATAFILENAME, auvdatafile);
            fflush (stdout); /* force completion of screen write */

        }
        if  (auvdatafile != NULL)
        {
            if (TRACE && DISPLAYSCREEN) printf ("[auvdatafile flushed]\n");
            fflush (stdout); /* force completion of screen write */

            fflush (auvdatafile); /* force completion of file write */

            fclose (auvdatafile);
            if (TRACE && DISPLAYSCREEN) printf ("[auvdatafile closed]\n");
            fflush (stdout); /* force completion of screen write */
        }
        else if ((TRACE && DISPLAYSCREEN) &&
                ((TACTICAL == FALSE) || (LOCATIONLAB)))
                printf ("[auvdatafile was not open!!]\n");

        if  (TRACE && DISPLAYSCREEN)
        {
            printf ("[flushing and closing auvtextfile %s %x]\n",
                                                AUVTEXTFILENAME, auvtextfile);
            fflush (stdout); /* force completion of screen write */
        }
        if  (auvtextfile != NULL)
        {
            if (TRACE && DISPLAYSCREEN) printf ("[auvtextfile flushed]\n");
            fflush (stdout); /* force completion of screen write */
            fflush (auvtextfile); /* force completion of file write */
            fclose (auvtextfile);
            if (TRACE && DISPLAYSCREEN) printf ("[auvtextfile closed]\n");
            fflush (stdout); /* force completion of screen write */
        }
        else if (TRACE && DISPLAYSCREEN) printf ("[auvtextfile was not open!!]\n");

/*      fclose (serialtestfile);  /* serial port test file */

        if (TRACE && DISPLAYSCREEN)
        {
            printf ("[finish record_data_off ()]\n");
            fflush (stdout);  /* force completion of screen write */
        }
        return;
}


/*****************************************************************************/


int detect_death_spiral (reset_statics)
    int reset_statics;
{
    static int turn_direction = 0;
    static double psi_old     = 0.0;
    static double start_psi   = 0.0;

    if (TRACE && DISPLAYSCREEN) printf ("[start detect_death_spiral ()]\n");

    /* reset static variables; don't check for spiral */
    if (reset_statics)
    {
        turn_direction = 0;
        if (TRACE && DISPLAYSCREEN) printf ("[finish detect_death_spiral ()]\n");
        return (FALSE);
    }

    /* Turn direction changed, reset static variables */
    if ((dsign (psi_dot) != turn_direction) || (turn_direction == 0))
    {
        turn_direction = dsign (psi_dot);
        start_psi = psi;
```

```
        psi_old = psi;
        if (TRACE && DISPLAYSCREEN) printf ("[finish detect_death_spiral ()]\n");
        return (FALSE);
    }

    /* Same turn direction, check for full circle */

    /* Right Hand Turn */
    if (turn_direction == 1)
    {
        if (((psi > start_psi) && (psi_old < start_psi)) ||
            ((psi_old > 330.0) && (psi > start_psi) && (psi_old > psi)) ||
            ((psi_old > 330.0) && (start_psi > psi_old) && (psi < start_psi)))
        {
            if (TRACE && DISPLAYSCREEN)
            {
                printf ("[Right Hand Death Spiral Detected]\n");
                printf ("[finish detect_death_spiral ()]\n");
            }
            return (TRUE);
        }
    }

    /* Left Hand Turn */
    else if (turn_direction == -1)
    {
        if (((psi < start_psi) && (psi_old > start_psi)) ||
            ((psi_old < 30.0) && (psi < start_psi) && (psi_old < psi)) ||
            ((psi_old < 30.0) && (start_psi < psi_old) && (psi > start_psi)))
        {
            if (TRACE && DISPLAYSCREEN)
            {
                printf ("[Left Hand Death Spiral Detected]\n");
                printf ("[finish detect_death_spiral ()]\n");
            }
            return (TRUE);
        }
    }

    /* No Spiral Detected */
    psi_old = psi;
    if (TRACE && DISPLAYSCREEN) printf ("[finish detect_death_spiral ()]\n");
    return (FALSE);

} /* end int detect_death_spiral () */


/*****************************************************************************/

void cage_dg () /* dg = directional gyro */

{
    /* Low TRUE Logic */
    /* Setting (Cage DG) Low and (UnCage DG) High will Cage the DG */
    via0b_reg = via0b_reg & 0xFE; /* Set bits PB0 Low retaining */
                                  /* other bits */
    via0[ORB_IRB] = via0b_reg;

    /*via0b_reg = via0b_reg | 0x02;*/ /* Set bit PB1 High retaining */
                                      /* other bits */
    /* Using PB3 Pin 44/19 since 48/23 crapped out */
    via0b_reg = via0b_reg | 0x08; /* Set bit PB3 High retaining */
    via0[ORB_IRB] = via0b_reg;

    printf("Waiting 20 sec. for Gyro to Cage\n");
    tsleep(2000);  /* Wait 20 seconds MAX for Caging */

    return;

}  /* end cage_dg () */
```

```
/*****************************************************************************/

void uncage_dg () /* dg = directional gyro */

{
   /* Low TRUE Logic */
   /* Setting (Cage DG) Hi and (UnCage DG) Low will UnCage the DG */

   via0b_reg = via0b_reg | 0x01; /* Set bit PB0 High retaining */
                                 /* other bits */
   via0[ORB_IRB] = via0b_reg;

   /*via0b_reg = via0b_reg & 0xFD;*/ /* Set bits PB1 Low retaining */
                                     /* other bits */
   /* Using PB3 Pin 44/19 since 48/23 crapped out */
   via0b_reg = via0b_reg & 0xF7; /* Set bits PB3 Low retaining */
   via0[ORB_IRB] = via0b_reg;

   tsleep(100); /* Wait 1 second for UnCaging */

   return;

}  /* end uncage_dg () */

/*****************************************************************************/

double dsign(value)

   double value;
{
   if(value == 0.0) return(1.0);
   if(value > 0.0)  return(1.0);
   if(value < 0.0)  return(-1.0);
}

/*****************************************************************************/

double dtanh(value)

   double value;
{
   if(fabs(value) > 1.0)
   {
      return(dsign(value));
   }
   else
   {
      return(value);
   }

/*   double epv,emv;

   epv = exp(value);
   emv = exp(-value);

   return( (epv - emv)/(epv + emv) );
*/
}


/*****************************************************************************/
/*****************************************************************************/
/* end of external_functions.c                                               */
/*****************************************************************************/
/*****************************************************************************/
```

APPENDIX F - mission.script.HELP

```
//-----------------------------------------------------------------------------//

        ~/execution/mission.script.HELP                26 April 96

        Mission script syntax for NPS AUV execution level control
        & tactical level in the NPS AUV Underwater Virtual World.

        Don Brutzman                          brutzman@nps.navy.mil

//-----------------------------------------------------------------------------//

This file describes how to change and create NPS AUV mission script files.
Files and the 'execution' program are in the ~/execution subdirectory.

       To run a new mission, copy an existing mission file over file
        'mission.script' or edit the mission.script file for a new mission.

       Example:       unix> cd execution
                      unix> cp  mission.script.siggraph  mission.script
                      unix> execution  virtualhost fletch.cs.nps.navy.mil

Script commands are read by AUV execution level (execution.c)
       from the "mission.script" default file at the start of each mission.

Some of the following commands will also work when invoked from the command
       line upon execution.

Here are script keywords (and synonyms) that are currently recognized:

         |            |
---------+------------+-------------------------------------------------------
Keywords | Parameters |  Description
Synonyms | [optional] |  (units in feet-degrees-seconds as appropriate)
---------+------------+-------------------------------------------------------
         |            |
         |            |
HELP                  Provide a list of available keywords
?                     (as specified in this HELP file).
/?
-?

VIRTUALHOST   hostname        tells execution level to open socket to virtual world
VIRTUAL              hostname which is already running and waiting on 'hostname'
REMOTEHOST    hostname        VIRTUALHOST is a command line switch.  Example:
REMOTE        hostname        unix> execution  virtualhost fletch.stl.nps.navy.mil
DYNAMICS      hostname

TACTICALHOST  hostname        tells execution level to open socket to tactical level
TACTICAL      hostname        which is already running and waiting on 'hostname'
STRATEGICHOST hostname        TACTICAL/STRATEGIC is a command line switch.  Example:
STRATEGIC     hostname        unix> execution  tacticalhost fletch.stl.nps.navy.mil

//                    comments follow on this line which are not executed
/*                    note comments will still be spoken if AUDIO-ON
#                     pound sign also indicates a comment if in first column

WAIT                  #      Wait  (or  run)  for  #  seconds  (letting  the  robot  execute)

RUN            #      prior to reading from the script file again

                        If in TACTICAL mode, execution ignores WAIT commands.

TIME           #      Wait (or run) until robot clock time #
WAITUNTIL       #     (letting the robot execute its current orders)
PAUSEUNTIL     #      prior to reading from the script file again
```

```
                        If in TACTICAL mode, execution ignores TIME commands.

TIMESTEP        #       change default execution level time step interval
TIME-STEP       #       from default of 0.1 sec to # sec

PAUSE                   temporarily stop execution until <enter> is pressed

REALTIME                run execution level code in real-time
REAL-TIME               (busy wait at the end of each timestep if time remains)

NOREALTIME              run execution level code as quickly as possible
NO-REALTIME
NONREALTIME
NOWAIT
NO-WAIT
NOPAUSE
NO-PAUSE

MISSION     filename    Replace 'mission.script' with 'filename' and start
SCRIPT      filename    the new mission.  Read tactical commands for execution
FILE        filename    level from filename.

TELEMETRY   filename    Playback prerecorded telemetry data from filename.
                         Consider using with NOSCRIPT if no script file present.
                        dynamics should be run with selection
                              E dEad_reckon_test_with_execution_level

NOSCRIPT                Ignore script command file.  Selectively used
                        in combination with TELEMETRY data file playback.

KEYBOARD                read script commands from keyboard
KEYBOARD-ON

KEYBOARD-OFF            read script commands from mission.script file
NO-KEYBOARD

QUIT                    do not execute any more commands in this script, but
STOP                    repeat the mission again if LOOP-FOREVER is set
DONE
EXIT
REPEAT
RESTART
COMPLETE
<eof> marker

KILL                    same as QUIT but also shuts down socket to virtual world
SHUTDOWN                'dynamics' process.

RPM             # [##]  Set ordered rpm values to # for both propellers
SPEED           # [##]  [ or independently set left & right rpm values
PROPS           # [##]    to # and ## respectively]
PROPELLORS      # [##]  maximum propellor speed is +- 700 rpm => 2 ft/sec

COURSE          #       Set new ordered course (commanded yaw angle)
HEADING                 #
YAW             #

TURN            #       Change ordered course by # degrees
CHANGE-COURSE   #               (positive # to starboard, negative # to port)

RUDDER          #       Force rudder to remain at # degrees

DEADSTICKRUDDER         [#]     Force rudder to remain at 0 [or #] degrees

DEPTH           #       Set new ordered depth (commanded z)

PLANES          #       Force planes to remain at # degrees

DEADSTICKPLANES         [#]     Force planes to remain at 0 [or #] degrees

PITCH           #       Set new ordered pitch (commanded theta angle).
THETA           #       Only effective during HOVERCONTROL.
```

```
THRUSTERS-ON            Enable vertical and lateral thruster control
THRUSTERS
THRUSTERON
THRUSTERSON

NOTHRUSTER              Disable vertical and lateral thruster control
NOTHRUSTERS
THRUSTERS-OFF
THRUSTERSOFF

ROTATE          #       open loop lateral thruster rotation control
                        at # degrees/sec

NOROTATE                disable open loop lateral thruster rotation control
ROTATEOFF
ROTATE-OFF

LATERAL                 #       open loop lateral thruster translation control
                        at # ft/sec
                        (positive is to starboard, maximum is 0.5 ft/sec)

NOLATERAL               disable open loop lateral thruster translation control
LATERALOFF
LATERAL-OFF

DIVETRACKER1 # ## ###   Position of DiveTracker transducer 1
DIVETRACKER2 # ## ###   Position of DiveTracker transducer 2
                        Still need to incorporate bearing to DiveTrackers.

GPS                     Proceed to shallow depth, take Global Positioning
GPSFIX                  System (GPS) fix, restore ordered depth when done.
GPS-FIX                         Control (thrusters, propellers/planes, combined)
                        is not modified.  Maximum fix time is 30 seconds,
                        at which time execution returns to previously
                        ordered depth.

GPS-COMPLETE            GPS fix complete, resume previously ordered depth.
GPS-FIX-COMPLETE

GYRO-ERROR    #         Degrees of error measured for gyrocompass.
GYROERROR     #         [GYRO + ERROR = TRUE]

DEPTH-CELL-BIAS  #        Feet of bias error measured for depth cell.
DEPTHCELLBIAS    #        [DEPTH CELL Z + BIAS = TRUE Z]
DEPTH-CELL-ERROR #
DEPTHCELLERROR   #

LOCATION-LAB            Vehicle is operating in lab using virtual world.

LOCATION-WATER          Vehicle is operating in water without the virtual world.

POSITION      # ## [###]       reset vehicle dead reckon position to (x, y) or
LOCATION      # ## [###]       (x, y, z) = (#, ##, ###) at current clock time
FIX           # ## [###]       This is a navigational position fix.  Receipt of a
                        POSITION/LOCATION/FIX command resets the execution
                        level dead-reckon position.  Note that depth value z
                         will likely be reset by depth cell if operational.

ORIENTATION  # ## ###  reset vehicle orientation to
ROTATION     # ## ###  (phi, theta, psi) = (#, ##, ###)

POSTURE #a #b #c #d #e #f
                        reset vehicle dead reckon posture to
                        (x, y, z, phi, theta, psi) = (#a, #b, #c, #d, #e, #f)

OCEANCURRENT  #x #y [#z] Ocean current rate along North-axis, East-axis and
OCEAN-CURRENT #x #y [#z]      [optional] Depth-axis (feet/sec)
                         (this is cartesian version of parametric set and drift)

CONTINUE                continue reading script & executing, no action performed
GO
```

```
STEP                    loop for another timestep prior to reading script again.
SINGLE-STEP             Only useful in execution keyboard mode.

TRACE                   enable  verbose print statements in execution level
TRACE-ON

TRACEOFF                disable verbose print statements in execution level
TRACE-OFF
NOTRACE
NO-TRACE

LOOPFOREVER             repeat current mission when done.
LOOP-FOREVER            each repetition is called a 'replication.'

LOOPONCE                do not LOOPFOREVER, stop when end of script is reached
LOOP-ONCE

LOOPFILEBACKUP          back up output files during each loop replication
LOOP-FILE-BACKUP                to permit inspection while new files are written
                        the backup files are in execution directory:
                        output.telemetry.previous & output.1_second.previous

ENTERCONTROLCONSTANTS           start a keyboard dialog to enter
ENTER-CONTROL-CONSTANTS                 revised control algorithm coefficients

CONTROLCONSTANTSINPUTFILE       read revised control algorithm coefficients
CONTROL-CONSTANTS-INPUT-FILE    from file "control.constants.input"

SLIDINGMODECOURSE       Sliding mode course control algorithm (not yet working)
SLIDING-MODE-COURSE

SLIDINGMODEOFF          Disable sliding mode course control algorithm
SLIDING-MODE-OFF

SONARTRACE              Enable  verbose print statements in execution sonar code

SONARTRACEOFF           Disable verbose print statements in execution sonar code

SONARINSTALLED          Sonar interface hardware cards are installed, use them

SONAR725  #b #r #p      Set the bearing (#b), range (#r), and power (#p) of the
SONAR-725 #b #r #p      ST-725 sonar. In virtual world, bearing is necessary for
SONAR_725 #b #r #p      sonar model.  In water, this stores data in the state
ST725     #b #r #p      vector for replay and examination.

SONAR1000  #b #r #p     Set the bearing (#b), range (#r), and power (#p) of the
SONAR-1000 #b #r #p     ST1000 sonar. In virtual world, bearing is necessary for
SONAR_1000 #b #r #p     sonar model.  In water, this stores data in the state
ST1000     #b #r #p     vector for replay and examination.

PARALLELPORTTRACE       enable trace statements for parallel port communications

AUDIBLE                         enable  text-to-speech audio output
AUDIO
AUDIO-ON
SOUND-ON
SOUNDON
SOUND

SILENT                  disable text-to-speech audio output
SILENCE
NOSOUND
SOUNDOFF
SOUND-OFF
AUDIOOFF
AUDIO-OFF
QUIET

SOUNDSERIAL             tell virtual world to pause while playing back sound
SOUND-SERIAL            (default)

SOUNDPARALLEL           tell virtual world to play sounds as parallel processes
```

```
SOUND-PARALLEL          (this may cause garbles if speeches play simultaneously)

EMAIL                   ask  user for electronic mail address at mission start,
EMAIL-ON                send user an  electronic mail report  at mission finish
E-MAIL
E-MAIL-ON
EMAILON

EMAILOFF                disable electronic mail address query feature
EMAIL-OFF
E-MAILOFF
E-MAIL-OFF
NO-E-MAIL
NO-EMAIL
NO-E-MAIL
NOEMAIL

WAYPOINT    #X #Y [#Z]        Point towards waypoint with coordinates (#X, #Y)
WAYPOINT-ON #X #Y [#Z]        (depth #Z optional).  Leave waypoint control by
                        ordering course, rudder, sliding-mode, rotate or
                        lateral thruster control.

                         If in TACTICAL mode, execution reports STABLE when done.

WAYPOINTFOLLOW          Set mode to arrive at each waypoint before reading the
WAYPOINT-FOLLOW                next mission script command, i.e. continue towards each
WAYPOINTFOLLOWON        waypoint for however long it takes to reach the standoff
WAYPOINT-FOLLOW-ON      distance before pausing to read the next command.

WAYPOINTFOLLOWOFF       Disables WAYPOINTFOLLOW mode
WAYPOINT-FOLLOW-OFF

STANDOFF          #     Change standoff distance for WAYPOINT-FOLLOW and HOVER
STAND-OFF         #     control
STANDOFFDISTANCE   #
STANDOFF-DISTANCE  #
STAND-OFF-DISTANCE #

HOVER      [#X #Y] [#Z]       Hover using thrusters and propellers for longitudinal
                        and lateral positioning at specified or previous
                        waypoint

HOVER      [#X #Y] [#Z]       [#orientation]  [#standoff-distance]

                        Uses WAYPOINT control until within #standoff-distance
                        of HOVER point (#X, #Y, #Z), then switches to
                        HOVER control with [optional] final #orientation

                        Full speed (700 RPM) port & starboard is used if
                        AUV distance to WAYPOINT is > #standoff-distance + 10',
                        then slows to 200 RPM until within #standoff-distance,
                        then HOVER control.

                        HOVER without parameters is the preferred method of
                        slowing since backing down with negative propellers may
                        result in large sternway and severe depth excursions.

                         If in TACTICAL mode, execution reports STABLE when done.

HOVEROFF                Turn off HOVER mode
HOVER-OFF
HOVER_OFF

TETHER                  command line switch only, used for in-water runs
TETHERED                set DISPLAYSCREEN=TRUE and LOCACTIONLAB=FALSE

UNTETHER                command line switch only, used for in-water runs
UNTETHERED              set DISPLAYSCREEN=FALSE and LOCACTIONLAB=FALSE

BENCH-TEST              Simplified command-line parameter for quick
BENCHTEST               switch setting during Russ's control and prop testing.
BENCH
```

//----------//

APPENDIX G - OBTAINING AND OPERATING CURRENT SOFTWARE

The Center for Autonomous Underwater Vehicle Research (CAUVR) at NPS maintains an Internet web site that is for the general public. The web site is located at `http://www.cs.nps.navy.mil/research/auv/` and contains an abundants of current information. The avenues that can be taken from this site are: Briefing Notes, The Phoenix AUV, Network Monitoring Page, Current Phoenix AUV software, NPS Underwater Virtual World, Underwater Robotics Laboratories on the WWW, Phoenix AUV Photos, Testing Status, Research Center Personnel, Thesis Work, Papers in Hypertext, Publication Abstracts, and Anonymous ftp Server.

The address `http://www.stl.nps.navy.mil/~brutzman/dissertation/execution/auv-uvw.GUIDE` describes in detail what equipment is needed to run the software, how to download the most current version of the software, how to setup computers for running the software, and also gives sample script file missions to run. Because this is ongoing research, these files are updated regularly. The email address (`auvrg@cs.nps.navy.mil`) is used by the AUV research group to rapidly disseminate all information from meetings and conferences to all research team members. This address is available to anyone that is interested in underwater robotics. By emailing the research group, all interested personnel can be put onto the AUV research group email list and have all traffic forwarded.

LIST OF REFERENCES

Brutzman, Donald P., *NPS AUV Integrated Simulator*, Master's Thesis, Naval Postgraduate School, Monterey, CA, March 1992.

Brutzman, Donald P., *A Virtual World for an Autonomous Undersea Vehicle*, Ph.D. Dissertation, Naval Postgraduate School, Monterey, CA, December 1994. Available at *http://www.cs.nps.navy.mil/research/auv*

Brutzman, Donald P., *From Virtual World to Reality: Designing an Autonomous Underwater Robot*, Proceedings of the Autonomous Vehicles in Mine Countermeasures Symposium, Naval Postgraduate School, Monterey CA, April 1995. Available at *http://www.cs.nps.navy.mil/research/auv*

Byrnes, R.B., *The Rational Behavior Model: A Multi-Paradigm, Tri-Level Software Architecture for the Control of Autonomous Vehicles*, Ph.D. Dissertation, Naval Postgraduate School, Monterey, CA, March 1993.

Byrnes, R.B., *The Rational Behavior Software Architecture for Intelligent Ships, An Approach to Mission and Motion Control*, Naval Engineers Journal, March 1996.

Campbell, Michael, *Real-Time Sonar Classification for Autonomous Underwater Vehicles*, Master's Thesis, Naval Postgraduate School, Monterey, CA, March 1996.

Davis, Duane, *Precision Maneuvering and Control of the Phoenix Autonomous Underwater Vehicle (AUV) for Entering a Recovery Tube*, Master's Thesis, Naval Postgraduate School, Monterey, CA, September 1996. Available at *http://www.cs.nps.navy.mil/research/auv*

Healey, A.J., *Mission Planning, Execution, and Data Analysis for the NPS AUV II Autonomous Underwater Vehicle*, Proceedings of the First IARP Workshop on Mobile Robots for Subsea Environments, Monterey, California, October, 1990

Healey, A.J., *Research on Autonomous Vehicles at the Naval Postgraduate School*, Naval Research Reviews, Office of Naval Research, Washington, DC, volume XLIV number 1, Spring 1992.

Kelly/Pohl, *A Book on C*, third edition, Benjamin/Cummings Publishing Company, Inc., Redwood City, CA, 1995.

Kwak, S.H., McGhee, R.B. and Bihari, T.E., *Rational Behavior Model: A Tri-Level Multiple Paradigm Architecture for Robot Vehicle Control Software*, technical report NPS-CS-92-003, Naval Postgraduate School, Monterey CA, March 1992.

Leonhardt, Brad, *Mission Planning and Mission Control Software for the Phoenix Autonomous Underwater Vehicle (AUV) Implementation and Experimental Study*, Master's Thesis, Naval Postgraduate School, Monterey, CA, March 1996. Available at *http://www.cs.nps.navy.mil/research/auv*

Marco, David, *Autonomous Control of Underwater Robots and Local Area Navigation*, Ph.D. Dissertation, Naval Postgraduate School, Monterey, CA, September 1996.

Marco, D.B., Healey, A.J., McGhee, R.B., *Autonomous Underwater Vehicles: Hybrid Control of Mission and Motion*, Autonomous Robots 3, 169-186, Kluwer Acedemic Publishers, Norwell, MA, 1996.

McClarin, David, *Kalman-Filtering of Navigation Data for the Phoenix Autonomous Underwater Vehicle*, Master's Thesis, Naval Postgraduate School, Monterey, CA, March 1996.

Rogers, Charles Ray, *A Study of 3-D Visualization and Knowledge-Based Mission Planning and Control for the NPS Model 2 Autonomous Underwater Vehicle*, Master's Thesis, Naval Postgraduate School, Monterey, CA, December 1989.

Scrivener, Arthur, *Acoustic Underwater Navigation of the Autonomous Underwater Vehicle using the DiveTracker System*, Master's Thesis, Naval Postgraduate School, Monterey, CA, March 1996.

INITIAL DISTRIBUTION LIST

		No. Copies
1.	Defense Technical Information Center 8725 John J. Kingman Rd., STE 0944 Ft. Belvoir, VA 22060-6218	2
2.	Dudlley Knox Library, Naval Postgraduate School 411 Dyer Rd. Monterey, CA 93943-5101	2
3.	Computer Technology Programs, Code CS Naval Postgraduate School Monterey, CA 93943-5000	1
4.	Dr. Ted Lewis, Code CS Chair, Computer Science Department Naval Postgraduate School Monterey, CA 93943-5100	1
5.	Dr. Robert McGhee, Code CS/Mz Computer Science Department Naval Postgraduate School Monterey, CA 93943-5100	2
6.	Dr. Donald P. Brutzman, Code UW/Br Undersea Warfare Academic group Naval Postgraduate School Monterey, CA 93943-5100	2
7.	Dr. Anthony J. Healey, Code ME/Hy Mechanical Engineering Department Naval Postgraduate School Monterey, CA 93943-5100	1
8.	CDR Michael Holden, USN, Code CS/Hm Computer Science Department Naval Postgraduate School Monterey, CA 93943-5100	1

9. LT Michael L. Burns 2
11 Dean Ave.
Dracut, MA 01826

10. David Marco, Code ME/MA 1
Mechanical Engineering Department
Naval Postgraduate School
Monterey, CA 93943-5000

11. Russell Whalen, Code CS/Wh 1
Computer Science Department
Naval Postgraduate School
Monterey, CA 93943-5000

12. Mr. Norman Caplan 1
National Science Foundation
BES, Room 565
4201 Wilson Blvd.
Arlington, VA 22230

13. Dr. James Bellingham 1
Underwater Vehicles Laboratory, MIT Sea Grant College Program
292 Main Street
Massachusetts Institute of Technology
Cambridge Massachusetts 02142