NISTIR 5126

Applying the NIST Real-Time Control System Reference Model to Submarine Automation: A Maneuvering System Demonstration

Hui-Min Huang
Ron Hira
Richard Quintero

*U.S. DEPARTMENT OF COMMERCE
Technology Administration
National Institute of Standards and Technology
Robot Systems Division
Bldg. 220 Rm. B124
Gaithersburg, MD 20899

Anthony Barbera

Advanced Technology and Research Corporation
Laurel Technology Center
14900 Sweitzer Lane
Laurel, MD 20707

NIST

NISTIR 5126

Applying the NIST Real-Time Control System Reference Model to Submarine Automation: A Maneuvering System Demonstration

Hui-Min Huang
Ron Hira
Richard Quintero

*U.S. DEPARTMENT OF COMMERCE
Technology Administration
National Institute of Standards and Technology
Robot Systems Division
Bldg. 220 Rm. B124
Gaithersburg, MD 20899

Anthony Barbera

Advanced Technology and Research Corporation
Laurel Technology Center
14900 Sweitzer Lane
Laurel, MD 20707

February 1993

U.S. DEPARTMENT OF COMMERCE
Ronald H. Brown, Secretary

NATIONAL INSTITUTE OF STANDARDS AND TECHNOLOGY
Raymond G. Kammer, Acting Director

APPLYING THE NIST REAL-TIME CONTROL SYSTEM REFERENCE MODEL TO SUBMARINE AUTOMATION: A MANEUVERING SYSTEM DEMONSTRATION

Hui-Min Huang, Ron Hira, and Richard Quintero

Robot Systems Division
National Institute of Standards and Technology
Gaithersburg, Maryland 20899

Dr. Anthony Barbera

Advanced Technology & Research Corporation
Laurel, Maryland 20707

CONTENTS

APPLYING THE NIST REAL-TIME CONTROL SYSTEM REFERENCE MODEL TO SUBMARINE AUTOMATION: A MANEUVERING SYSTEM DEMONSTRATION

Hui-Min Huang, Ron Hira, and Richard Quintero

Robot Systems Division
National Institute of Standards and Technology
Gaithersburg, Maryland 20899

Dr. Anthony Barbera

Advanced Technology & Research Corporation
Laurel, Maryland 20707

ABSTRACT

The Robot Systems Division (RSD) at the National Institute of Standards and Technology (NIST) has been developing a generic reference model architecture, known as the Real-time Control System (RCS), for the last two decades. This paper demonstrates the application of RCS to submarine automation. The automation of submarine operations involves complex system functionality and requires an enormous amount of intelligence to be built into the software to enable a submarine to operate in an unstructured and often hostile environment semi-autonomously. Software is emerging as a predominant factor in determining the success and performance of modern large and complex intelligent systems. Meanwhile, the fundamental principles and generic approaches of handling software and systems engineering processes are still being explored within the engineering community. RCS attempts to address some fundamental system development issues including a software engineering methodology and a generic architecture. The resolution of these issues can facilitate a unified approach for developing intelligent systems. An open system architecture can also be achieved to serve as a foundation for system integration and coordination. This paper provides an implementation example of the RCS methodology research projects ongoing at NIST RSD.

1. INTRODUCTION

1.1. The DARPA Project and Its Objectives

The National Institute of Standards and Technology (NIST) Robot Systems Division (RSD) has been sponsored by the Defense Advanced Research Projects Agency (DARPA) Submarine Technology Program (STP) for the automation of submarine operations[1]. One objective in this project is for NIST RSD to demonstrate its RCS architecture applied to submarine automation. A series of software demonstrations has been planned for achieving this objective. NIST RSD has been collaborating with the Advanced Technology and Research Corporation (ATR) in this project development effort. This paper reports on the results achieved in our latest project effort called Demonstration #3, or Demo #3.

[1]ARPA Order No. 7829, Amendment No. 000.

The specific objectives for Demo #3 included:

* Converting the existing submarine automation RCS software (Demo #2) to a C language based implementation (see section 2). Laying out the software structure for the entire demonstration system.
* Expanding functionality from Demo #2;
* Demonstrating multiple control modes in RCS by implementing human interactive control and decision aiding capability. In interactive mode, human operators are presented with on screen reports of operational problems, the suggested actions, and the possible effects. The operator then is able to select an appropriate command to address the reported problem.

Figure 1-1 is a screen display showing an animated submarine maneuvering under ice. Detailed discussions for each component of this animation screen will be given throughout this paper.

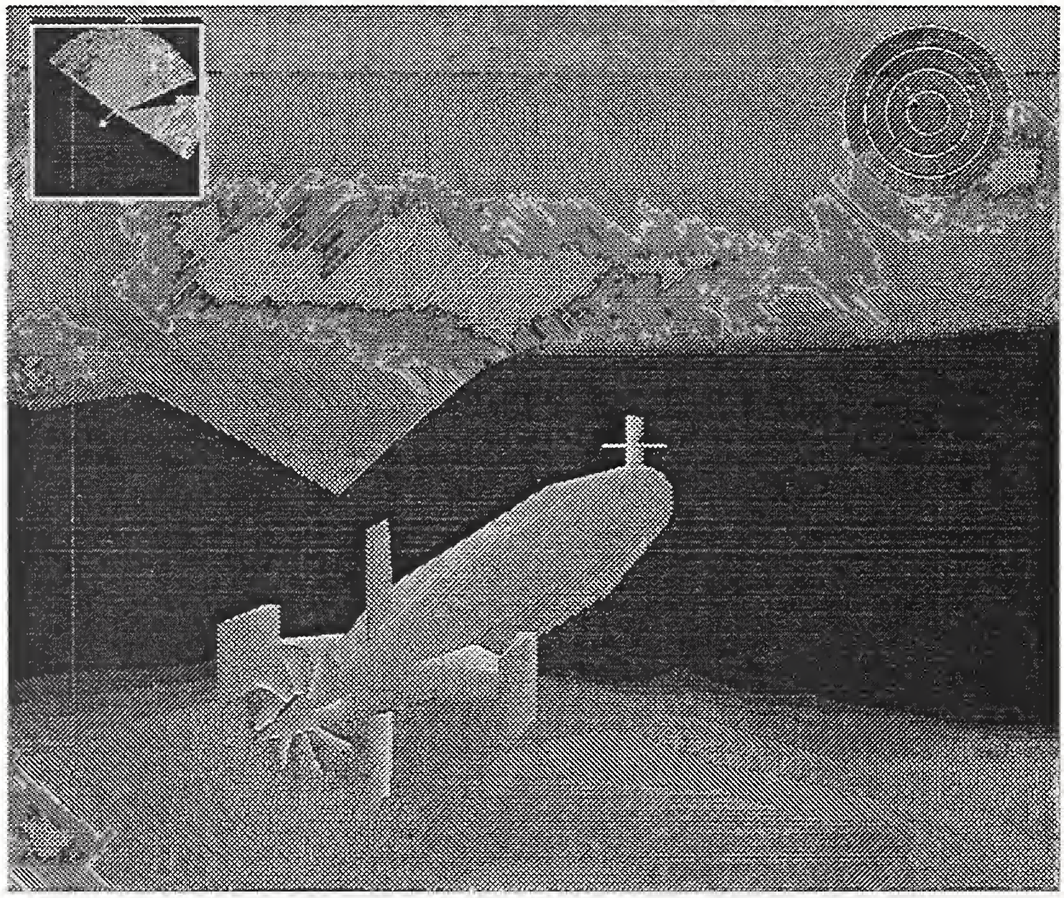


Figure 1-1 Submarine Under Ice Maneuvering

1.2. RCS Architecture

The National Institute of Standards and Technology (NIST) Robot Systems Division has been focusing on the research and development of its generic hierarchical Real-time Control

Systems (RCS) architecture since the late 1970's [Al 91, Ba 84]. RCS provides a reference model for complex hierarchical real-time control systems. It is a generic hierarchical control structure with each level assigned specific responsibilities. For example, RCS specifies an "elementary move" control level to be responsible for a system's kinematics. Controllers are employed at each level to fulfill the level's responsibilities. The controllers assume a generic format which features: sensory processing, world modeling, and behavior generation (previously called task decomposition) functions.

New computer hardware and software technologies have been adopted in RCS during the RCS evolution. The RCS applications include: the NIST Automated Manufacturing Research Facility (AMRF) [Al 82], the Army Field Material-Handling Robot (FMR) project [Jo 91], NASA/NBS Standard Reference Model for Telerobot Control System Architecture (NASREM) [Al 89-1], the DARPA Multiple Autonomous Undersea Vehicles (MAUV) project [Al 88], the Army Robotics Testbed[2] (RT) project [Sz 92, Sz 90], the U.S. Bureau of Mines Coal Mining Automation Project [Hu 91, Hu 90, Al 89-2], and this submarine automation project. Currently, RCS is also being applied to a new problem domain, intelligent vehicle and highway systems (IVHS) [Al 92].

Techniques employed in implementing these projects differed to some extent due to differences in:

* computing environments such as computer hardware, operating systems, programming languages, etc.,
* design and development approaches such as using task trees, finite state machines, object oriented analysis, etc., and
* real-time software execution models including communication mechanisms and sampling rates.

These differences essentially led to the research of different RCS methodologies. In the submarine automation project, the software demonstrations have been developed using an approach originated by Dr. Anthony Barbera of ATR. A comprehensive treatment of the Barbera approach is given in [Qu 92]. This approach basically utilizes controllers, task trees, and state transition diagrams/tables to describe RCS (discussed in detail in section 4). A set of generic controller templates is then used to facilitate implementation (discussed in detail in section 6). This paper is an illustration of how to apply this methodology. Similar discussions on the real-time system representation issue can also be found in [Ko 92] and [Ha 88]. However, they do not offer a generic model for application systems as RCS does (see the model for intelligent machine systems in [Al 91]).

Another control architecture which addresses the real-time embedded system control problem is the Task Control Architecture (TCA) developed by Simmons [Si 90] of Carnegie Mellon University. TCA specifies a generic block structure capturing common capabilities that robotic control systems may possess. The basic system execution model involves message routing (including commands and queries) among all system components via a central control module. As the authors [Si 90] pointed out, this central control module presents a potential bottleneck as system complexity grows. TCA shares the same view as RCS in the use of task trees to describe command chains. In addition, TCA specifies that modules can impose temporal constraints to sequence the planning and execution of system commands. However, from the RCS methodology point of view, the use of state diagrams and state tables, as a step beyond task trees, provides a more robust and systematic method

[2]Originated called the Army Tech-based Enhancement for Autonomous Machines (TEAM) project.

of describing behavioral transition among different system components in both temporal and spatial aspects [Appendix A].

2. PREVIOUS WORK

2.1. Early Demonstrations and the FORTH/Smacro Environment

The earlier demonstrations showed the applicability of RCS to submarine automation. In October of 1990, a concept preview demonstration was implemented. Animation of a lubrication oil fire in the engineering room and user interface allowing an operator to isolate the compartment to extinguish the fire were implemented on a Silicon Graphics Inc. (SGI) workstation. A simple RCS was implemented on an 80386 processor based personal computer (PC) for controlling the simulated submarine.

Demo #1 was presented in February of 1991. A preliminary RCS for ship maneuvering control was implemented along with the animation of the submarine control surfaces. A hierarchy, together with partially implemented displays of command/status numbers and module execution time, could be displayed on the SGI for debugging purposes.

Demo #2 was essentially a work in progress demonstration. It was held in August of 1991. The RCS on the PC had been enhanced to allow for some low level subsystem automatic control such as trim or depth adjustments. The simulation and animation had been expanded to include the real-time computation and graphic display of the submarine sonars, the Cerebellar Model Articulation Controller (CMAC, see sections 3 and 6) neural network, ice keel (jagged underwater ridges and peaks of ice formed when packs of ice plates collide), and sea floor. See section 6 for more detail on these subjects.

The RCS software code for all these early demonstrations was written in a language called Smacro. Smacro was originally developed by Dr. Barbera and M.L. Fitzgerald at NIST in the early 80's. They have continued to evolve Smacro since joining ATR in the late 80's. Smacro was developed using the dictionary based language, FORTH. All processes, subroutines, and variables are defined as words in the dictionary. Data are passed using stacks. The advantages of the FORTH/Smacro language include:

* allowing incremental loading of the code to expedite program testing and prototyping;
* reducing software source code size [Appendices B and C];
* allowing customized operating systems tailored for specific applications [Br 84].

However, the language suffers from some disadvantages which include:

* lacking environmental support, such as user friendly file management or multiple window features;
* employing a block as a program unit. The limitation in the block size (16 lines and 64 characters per line) sometimes discourages documentation within the code (the use of: self-explanatory but longer notations, comments within the code, etc.).

FORTH/Smacro does not seem to be a well supported nor recognized software environment, although it does seem to possess some technological superiority. From the technology transfer (a NIST mission) point of view, it is a deficiency to have standards oriented technologies developed in this kind of environment. As Strassmann [St 92] pointed out, "The rapid deployment of information systems in the future conflicting under unpredictable and often hostile conditions calls for easily repairable software that is constructed from reliable standard components." The FORTH/Smacro language might be regarded as "craft mode" software as opposed to "industrialized" software in Strassmann's terms. Smacro was used in the earlier demonstrations to show the applicability of RCS and to address the RCS implementation issues. These objectives have been achieved judging from the success of the early demonstrations. At this juncture, NIST decided to convert the submarine automation RCS development and control environment to the C language since the C language is a widely accepted and well supported environment. Conversion to C is expected to expedite technology transfer.

2.2. Conversion to Demo #3

The first step in developing Demo #3 was a faithful conversion of the Smacro code to C. Generic templates written in the C language were generated (section 6) based on the Smacro controller templates. All the ship maneuvering controllers are converted. Appendices B and C show a typical conversion for a state table.

3. PROBLEM DOMAIN

3.1. Background

One of the objectives is to develop intelligent control systems designs which can function in unstructured environments while employing deterministic behavior. One of the first and foremost tasks for building intelligent control systems is learning as much detail about the problem domain as possible. The problem domain in this project is a 637 class nuclear powered submarine. Budget constraints and technological advancements have made automation a more salient feature for submarine designers.

A requirement imposed on the automated submarine for this demonstration was that it should be able to operate unmanned or with some high level human supervision. This design objective requires the RCS implementation to span the servo through mission levels as defined by Albus [Al 89-1]. This section will elaborate on some details of the demonstration submarine as well as introduce the mission scenario. A submarine is a very complex system, and as such, work has been confined for this paper to maneuvering automation.

3.2. Mission

The mission for this submarine scenario is to traverse the Bering Strait in covert mode (meaning that avoiding detection by the enemy is of the utmost importance). This mission was chosen because it creates a rich set of possible scenarios that exercise all levels of RCS, particularly in maneuvering control. Much of the submarine's transit through the Bering Strait will be under ice with shallow sea floor depths, which requires the control hierarchy to perform obstacle avoidance. Another phenomenon encountered is changes in sea water density, which forces one to make decisions regarding depth control coupled with signature management (detection avoidance). Fluctuations in water density may require maneuvering mechanism adjustments which jeopardize the submarine's cloak. Of course, the trade-off between safety of the ship and its crew must be weighed against mission stealth requirements. Clearly this mission requires decisions be made while operating in an unstructured environment, one characteristic of intelligent control.

3.3. Maneuvering Mechanisms

As aforementioned, the submarine is an extremely complex system, and as such the demonstration is limited to maneuvering. A submarine has a number of mechanisms for hydrodynamic control of its depth, buoyancy, orientation, and speed. See figure 3-1. The information provided in the paragraphs to follow are presented only as an introduction to the various mechanisms, and should not be interpreted as a comprehensive list. They were chosen because of their important influence on the hydrodynamics of the submarine. The following mechanisms and their respective functions will be analyzed:

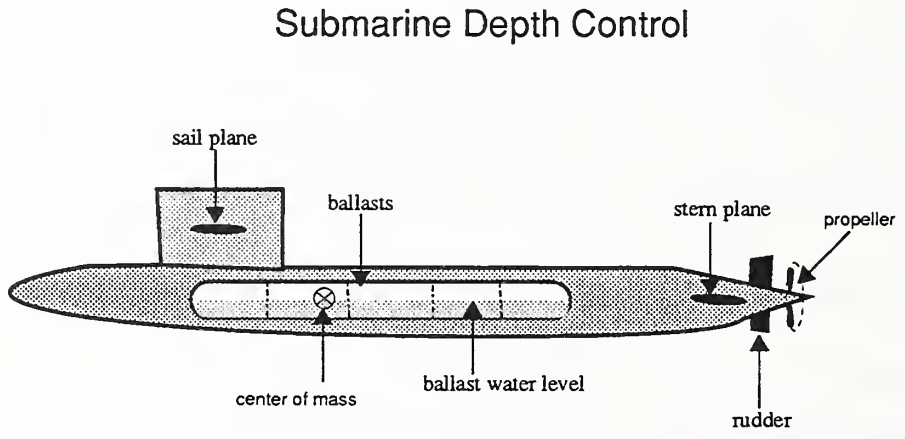


Figure 3-1

- * main ballast tanks
- * variable ballast tanks
- * sail planes
- * stern planes
- * rudder planes
- * turbine

The above system is part of a complex multi-input multi-output (MIMO) system. Many of the mechanisms and their effects on the submarine are inherently interrelated; i.e., a particular mechanism may be used to control the ship's depth and orientation. One such example is that the stern planes and variable ballast tanks each affect both pitch and depth.

The main ballast tanks (MBTs) are used for gross control of the ship's buoyancy. The MBTs are used primarily for submarine submerging and surfacing. If the six tanks are flooded (completely filled) the submarine gains negative buoyancy and the ship submerges. If the tanks are blown (emptied) the submarine gains positive buoyancy and surfaces. The tanks are blown by admitting high-pressure air through valves at their tops. Conversely, they are flooded by allowing the air at the top of the tanks to leave through the vent valves while sea water floods through ports at the bottom of the tanks. When the MBTs are full, and the variable ballast tanks are at their prescribed levels, the submarine is at neutral buoyancy; i.e., the ship is neither sinking nor rising.

The variable ballast tanks are used for small adjustments in buoyancy and orientation. "Variable ballast" as used in this paper refers to any of the following tanks: forward, aft, water round torpedo, auxiliary, and depth control. An example of buoyancy control via variable ballast follows. At larger depths the pressure of the water outside the vessel is much higher and causes the pressure hull to contract. The weight of the ship remains the same; however, the volume of the water it displaces decreases due to the hull contracting. The result is negative buoyancy and the sub will start to sink. Blowing the proper amount of water from the variable ballast tanks, which decreases the weight of the ship, may be sufficient to stabilize the submarine's buoyancy. The variable ballast tanks are also used for trim/orientation control. Weight distribution in a submarine may change after it has been at sea for some time; for example, the supplies which were brought on board might be consumed which results in a change in forward and aft weight distribution. The ship will experience an orientation change; specifically in this case, it will experience a downward pointing pitch (the bow will be at greater depth than the stern). Water may be transferred between the forward and aft trim tanks to resolve the imbalance. The pitch is referred to as its bubble or bubble angle. Ballast tanks may also be used to compensate for improper roll.

The sail planes are located on the conning tower and have a range of $\pm$ 22°. These planes are used for depth control in a variety of methods to be detailed in the next section. The stern planes, also used for depth and orientation control, are located at the rear of the submarine and have a range of $\pm$ 27°. Their distance from the center of mass provides a means for adjusting the pitch of the submarine. The rudder is used for steering the submarine left or right with limits on its range of $\pm$ 37°.

The turbine, which drives the propeller, is bi-directional, enabling the submarine to travel "ahead" or "astern" at a commanded speed. In practice an astern command is only used as an emergency braking procedure and almost never used for backward movement.

The maneuvering information provided in this section is used extensively in section 4 for task decomposition and developing plans for the controller hierarchy.

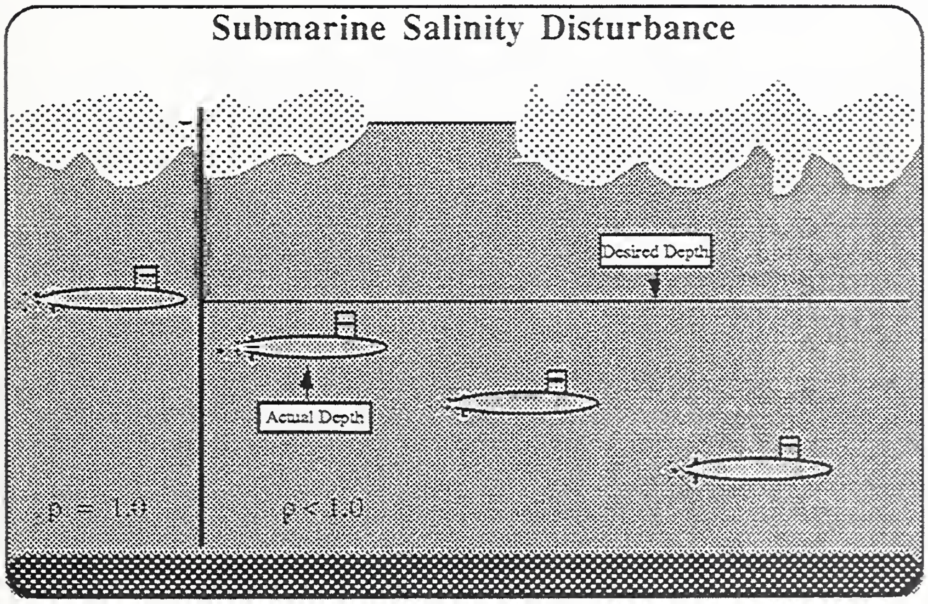


Figure 3-2

3.4. Scenario

One of the initial steps in the RCS design approach consists of developing scenarios, which enable the designer to flesh out all of the details of operation. Former submarine commanders provided detailed information on submarine operations. They were asked about the consequences of certain actions. In addition they provided guidelines for appropriate responses to particular scenarios. Their input was invaluable for obtaining problem domain knowledge, since we at NIST are not submarine experts. Many times "experts" provide high level answers. An RCS design requires not only high level answers, but also all of the low level detail necessary for computer controlled realization.

The scenario for the latest work is to navigate under ice in stealth mode with a sudden salinity change. Salinity gradients may occur from fresh water runoffs, where rivers of fresh water cause the water density, μ, to drop suddenly, see figure 3-2. A drop in the density of the sea water will cause the ship to have negative buoyancy and it will start sinking, which is due to the fact that the submarine weight is now greater than the weight of the water it displaces. Salinity variations frequently occur under ice and create significant problems related to depth control and the signature management system while the ship is operating stealthily. Temperature fluctuations are common in the open ocean, and can cause similar depth control problems. The signature management system is responsible for maintaining an acceptable noise level that keeps the submarine invisible to enemy sonar.

This scenario enabled the demonstration of a number of RCS features to be explained in detail in sections 4 and 6.

3.5. Depth Control

A submarine can control its depth in a number of different ways. The choices are permutations of the aforementioned mechanisms. The operations described assume that the submarine is traveling ahead. All of the operations cause noise to be generated; however, a primary goal is to keep noise to a minimum to avoid enemy detection. Since sudden and large changes in any of the control surfaces will generate significant noise, soft limits are set on their operating range. The likely operations to be ordered by a submarine commander are:

* Ascend/Descend
* Up Bubble/Down Bubble
* Maintain Depth
* Blow Main Ballast

After submerging by venting the MBTs, it is desired that the submarine reach a specified depth; therefore, a Descend command is issued with the desired depth, see figure 3-3. The Descend command requires that both the stern and sail planes point down which provides a means for the submarine to dive without changing its pitch. An Ascend command is analogous, with both planes pointing up, and causes the ship to rise, see figure 3-4.

DESCEND: (DR_DESCEND)

FAIRLY NOISY PROCESS
USE SAIL AND STERN PLANES
DIVE WITHOUT PIVOTING

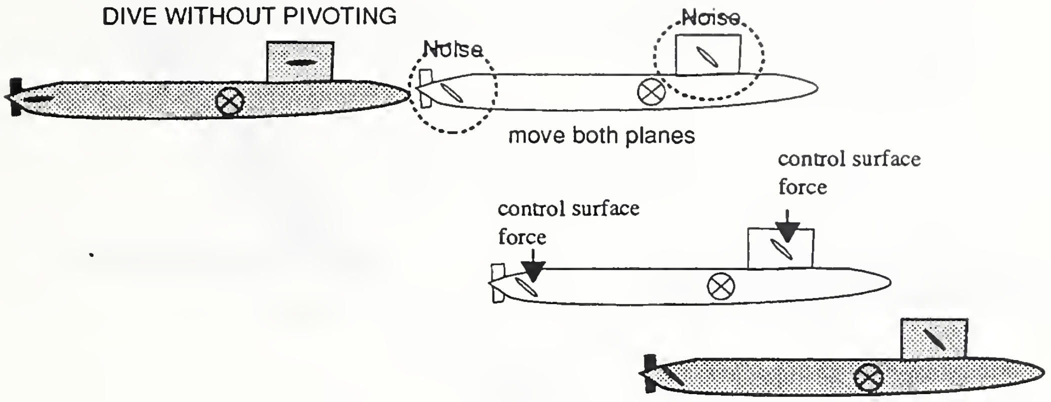


Figure 3-3

ASCEND: (DR_ASCEND)

FAIRLY NOISY PROCESS
USE SAIL AND STERN PLANES
RISE WITHOUT PIVOTING

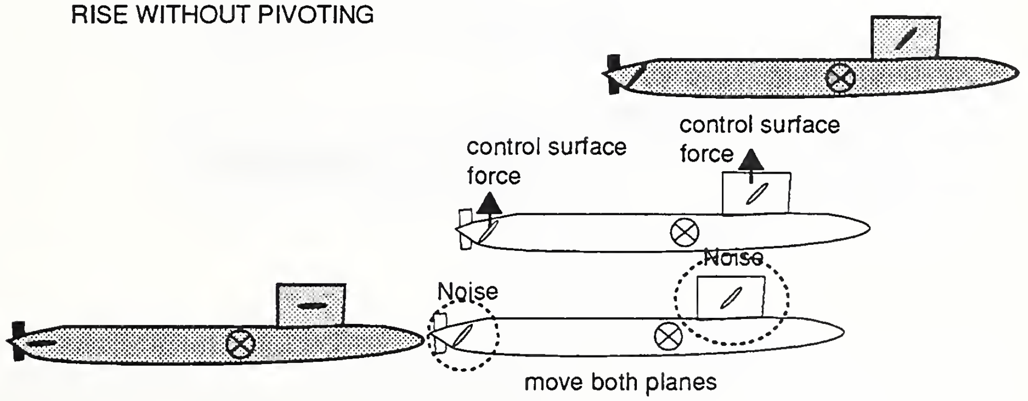


Figure 3-4

If a pitch change is required, an Up Bubble command may be issued, see figure 3-5. The submarine points up, which may be utilized to decrease depth. The term bubble originates from the air bubble type level sensors used to determine the pitch for the submarine. The bubble angle may be changed by changing the stern plane or by pumping water between the Forward and Aft tanks of the variable ballast system. The stern plane effects will be analyzed here. Stern planes pointing down, produce a downward force on the stern of the ship, and because the location of the force is a large enough distance from the center of mass of the ship, a counterclockwise torque results. A Down Bubble command is analogous, with the stern planes pointing up, see figure 3-6. The pitch of the ship may be used in conjunction with propulsion to control depth.

UP BUBBLE: (DR_UP_BUBBLE)
MODERATELY NOISY PROCESS
USE STERN PLANE
RISE BY PIVOTING

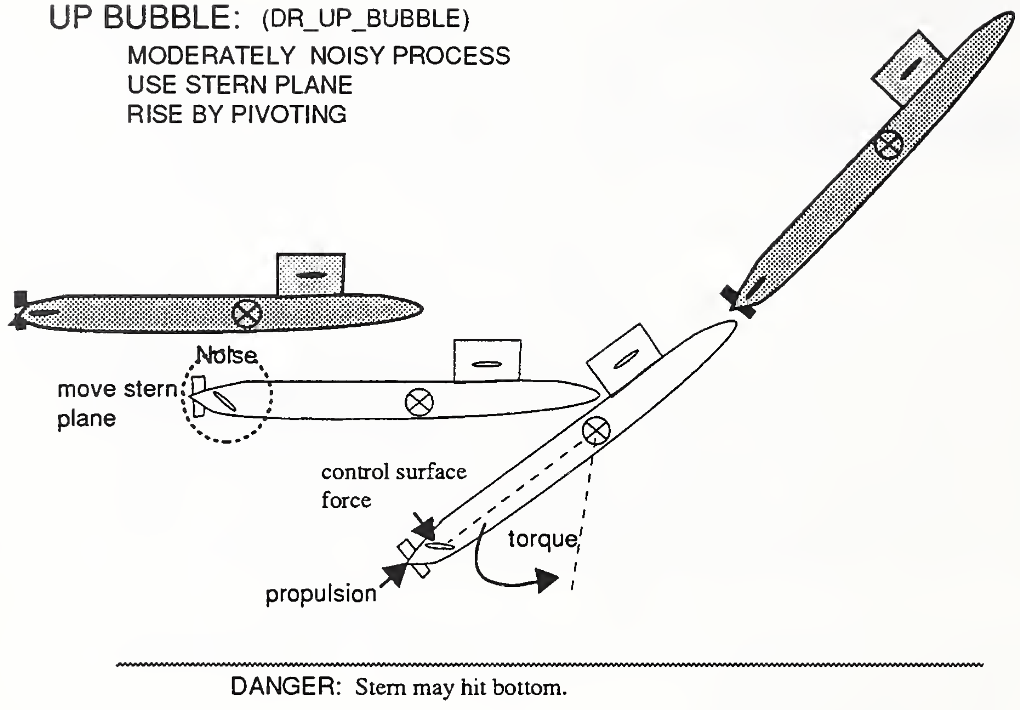


Figure 3-5

DOWN BUBBLE: (DR_DOWN_BUBBLE)
MODERATELY NOISY PROCESS
USE THE STERN PLANE
DIVE BY PIVOTING

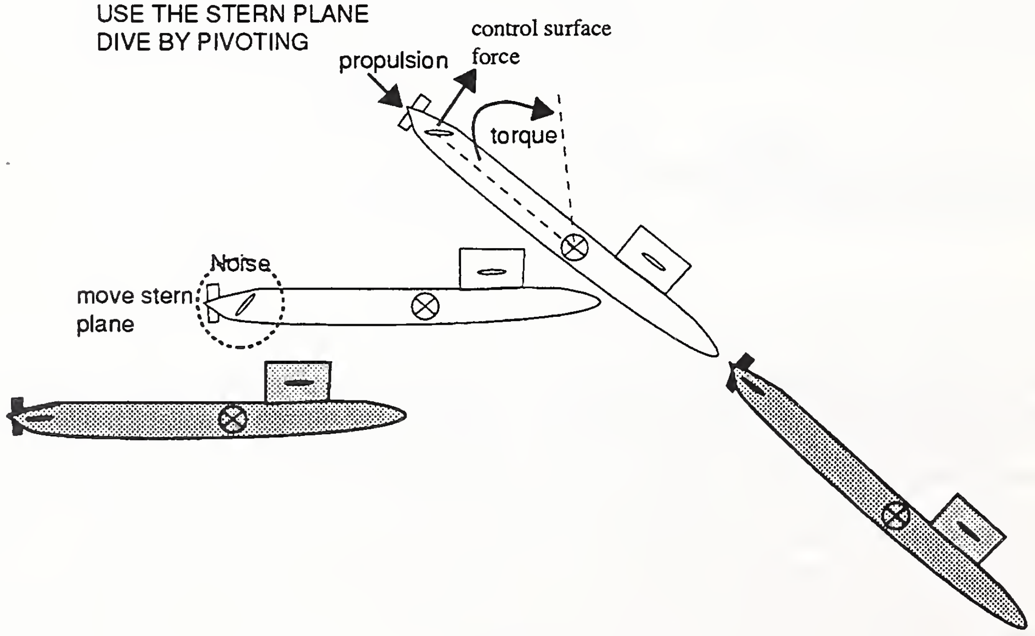


Figure 3-6

MAINTAIN DEPTH: (DR_MAINTAIN_DEPTH)
QUIETEST PROCESS
USE SAIL PLANE
COMPENSATE FOR SMALL DISTURBANCES
MAINTAIN A CONSTANT DEPTH

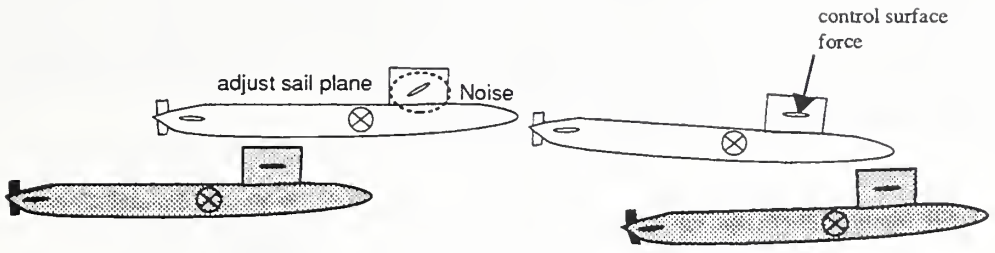


Figure 3-7

Blow Ballast:
VERY NOISY PROCESS
BLOW WATER FROM BALLAST
LIGHTEN SUB AND RISE

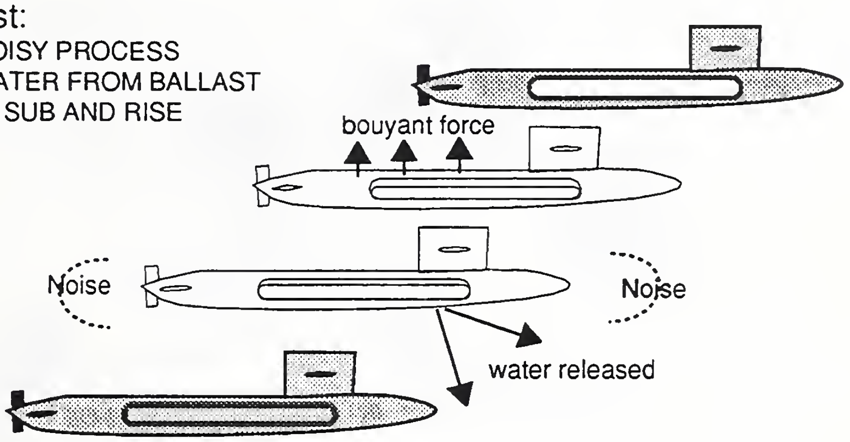


Figure 3-8

Once the submarine has reached a desired depth, it is sustained by the Maintain Depth command, see figure 3-7. The sail planes are used for small changes to maintain a desired depth with little change in submarine orientation. This is the normal operating procedure command.

When it is time to surface, the commander will order a Blow the Main Ballast, which involves blowing the water out of MBTs with pressurized air, see figure 3-8. There are a number of sources and magnitudes of pressurized air. The chosen source depends on the situation.

One important depth control method which was not analyzed but demands a brief mention in this synopsis is hovering via the depth control tanks. It is a complex system in and of itself, and was not included as a possibility in this particular scenario.

3.6. Ice Avoidance Maneuvering

In our demonstration the submarine has a goal point to reach, see figure 3-9; however, there are obstacles in its path, namely ice. If the current sonar returns show ice keels blocking the path of the current heading, then a new course is computed with the ship at its current depth. This is aided automatically by a Cerebellar Model Articulation Controller (CMAC) neural network [Al 75], which stores a map of the ice encountered. An algorithm is then used to compute a new course on a heading which is clear of ice keels. More detail on ice avoidance is presented in sections 4.4, 6.6, & 6.8.

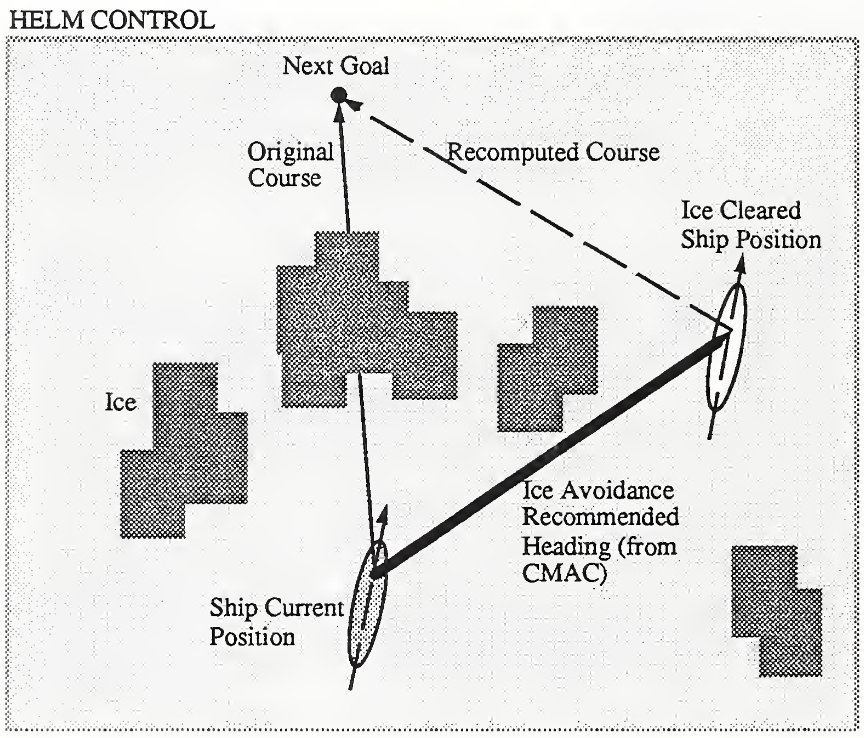


Figure 3-9

3.7. Salinity Problem and Reaction

The problem of salinity perturbations due to fresh water run-offs was introduced earlier. Once a salinity gradient occurs, it must be detected. Because of the sensor arrangement, there is a lag before the disturbance is detected. A detection algorithm was designed and implemented in the control hierarchy. If the problem is severe enough and persists, the Maintain Depth command with its limited plane range will not be adequate to sustain the desired depth. Recall that in our scenario the submarine is operating in stealth mode, and quiet operation is of the utmost importance.

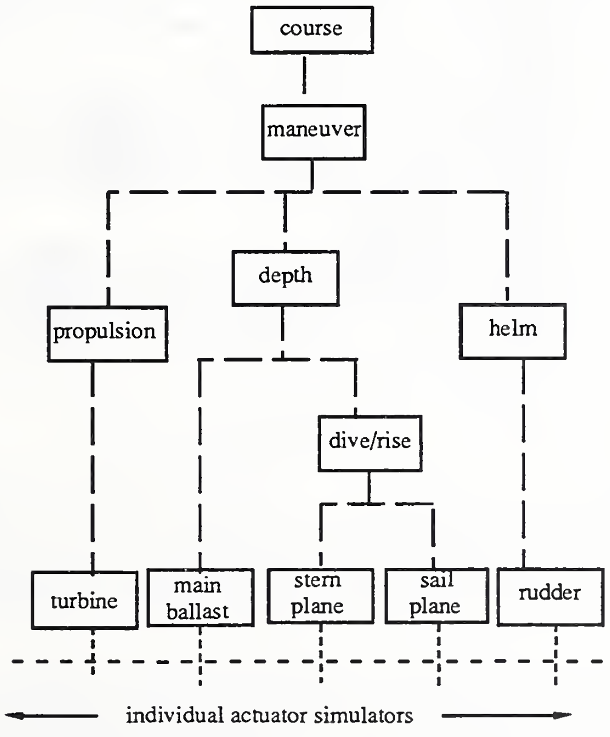
Each command introduced in section 3.4 has a noise level associated with it. In order to avoid cavitation, all maneuvering operations have limits on plane range and rate movements. In this scenario we assume that the order from quietest to noisiest operations is as follows:

* Maintain Depth - limited sail plane movement only (least noisy)
* Up Bubble - stern plane only, changes the bubble angle
* Ascend - both sail and stern plane movements
* Increase Propulsion Speed - provides greater control gain for planes
* Emergency Blow Main Ballast - causes the submarine to surface (most noisy)

The operating choice must be made at a high level of the chain of command. In this software demonstration, two alternative modes are supported. One, supervisor mode, requires the user to input the proper response to the difficulty with a list to choose from. The other, automatic mode, has the control hierarchy respond to the problem depending on certain state variables. State graphs presented in section 4.5 illustrate the salinity problem control implementation. Section 6.5 elaborates on the supervisory and automatic operation modes.

4. RCS REPRESENTATION FOR THE SHIP MANEUVER SYSTEM

After the submarine domain knowledge was acquired and narratively described in the scenario, the next implementation step was to organize and articulate this knowledge in an RCS format. Three representations were used: control hierarchy, task tree, and RCS plans. The development processes for these representations involve domain expert interaction and the rapid prototyping cycle. [Qu 92] and [Hu 92] give an in depth description for these processes. The following sections describe the results in detail.



Note: There are additional ship maneuvering controllers not shown here, including the trim and the depth control tanks.

Figure 4-1 Ship Maneuver Hierarchy (a Simplified View)

4.1. Control Hierarchy, Task Tree, and RCS plans Represented by State Graphs/Tables

In this demonstration, a control hierarchy for the real-time control of the ship maneuvering system has been developed and is shown in figure 4-1. A task tree has been developed to form the command chain. Figure 4-2 shows the resulting task tree mapped on the control hierarchy. The input tasks for each controller can be viewed as a description of the part of the system responsibility the controller shares [Hu 91]. Each task on the task tree, except for the lowest level tasks, corresponds to an RCS plan. RCS plans describe the behavior each controller can perform.

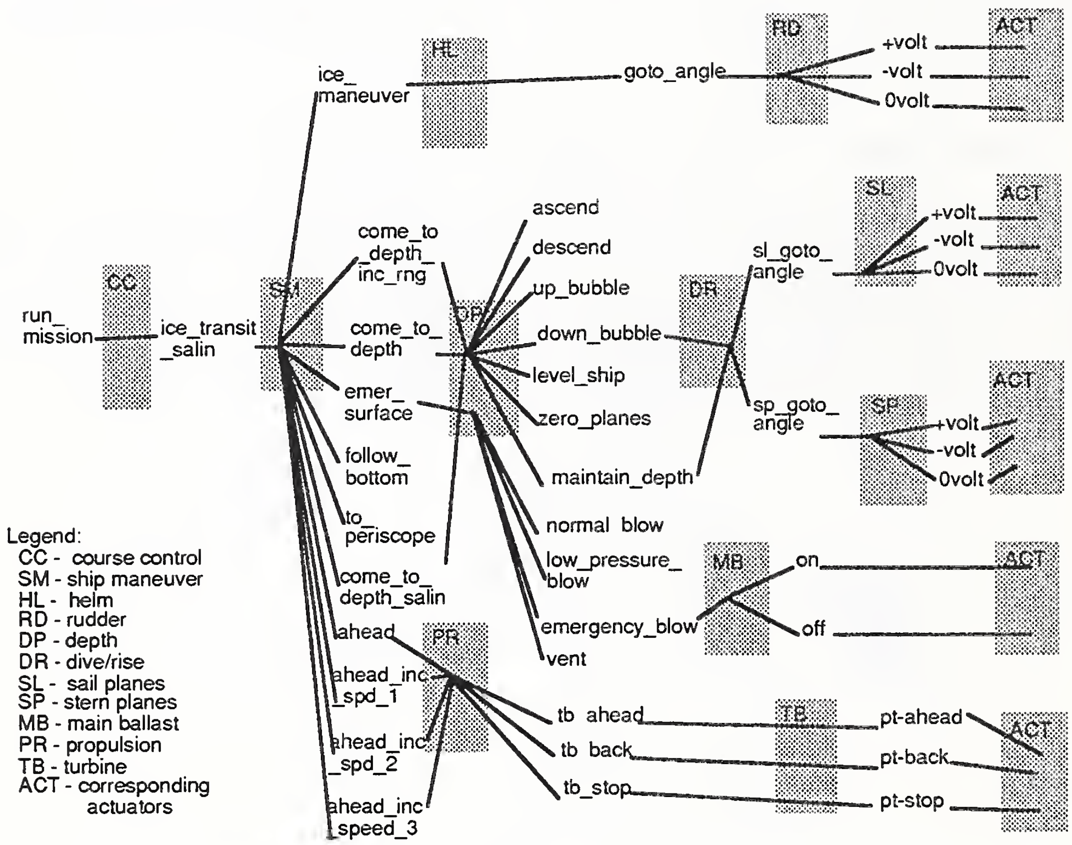


Figure 4-2 Ship Maneuver Task Tree

State transition diagrams and state tables, shown in figures 4-3 through 4-9 in the following sections, are used in this project to describe RCS plans[3]. While [Qu 92] gives an in-depth description of the syntax, a brief summary is provided here to facilitate the understanding of state diagrams. Note that a state diagram typically represents an RCS plan. Therefore, these two terms are used interchangeably in this paper.

* An oval bubble with a (Si) label and a name specify a state, where (i) is the index number. The state (Si) represents one of the finite states that the controller can attain. The state name describes a collective[4] action that the controller is performing. State bubbles are connected by edges which use arrows and

[3]A task is explicitly described by an RCS plan. Task and command are used interchangeably.

[4] Meaning that the state name describes the set of commands associated with that state.

conditions to describe the transition from state to state. This can be seen in figure 4-3 of the next section.

* A round bubble with a (*i) label is referred to as a "don't care[5]" state, where (i) is an index number. The event associated with this state is a prioritized one; i.e., the occurrence of such an event triggers immediate action regardless of the controller's current state.

* A box describes a state transition. The text above the edge starts with an "Evti" label, where (i) is an index number. The description of an event (or a set of events) follows to specify the condition(s) that triggers the particular state transition. The first line of text below the edge, labeled "Job," describes the computation jobs that the controller is required to perform upon the occurrence of the event. The second line, labeled "Cmd," specifies the commands the controller selects for its subordinates. The controller then enters the state the edge points toward. It is not necessary for all the subordinates to receive commands at every state transition. The subordinates that do not receive new commands will continue executing the previous commands.

  There exists a special type of events, namely, significant errors, such as severe drop in the depth of the submarine or an imminent collision with obstacles. These errors could be implemented as prioritized events accompanied by the "don't care" states. Depending on the level of authority the involved controllers have, the compensation action(s) for this type of errors could be initiated in the same plan or by the superior controller. See the RCS task decomposition process as described in [Hu 91] for more information.

* State tables, which complement state diagrams, describe the same information in a tabular format. State diagrams, being graphic, are generally easier to comprehend, whereas state tables have the advantage of providing direct correspondences to the code. Please see section 4.2 and appendices B and C for more information.

* The execution of RCS plans follows a state clock. A controller either stays at a state or transitions to the next state at each state clock control cycle. The decision making process in a control cycle depends only on the current input and the state number of the last cycle. This number is used in conjunction with the state of the world and the state of the input to determine the control action for the current cycle.

A simplified hierarchy is shown at the lower left corner in each state diagram. The controller executing the plan is highlighted.

Note that the state graphs shown in this section apply primarily to the automatic mode system operation. Section 6.5 describes a second mode of operation, interactive mode, which requires a different set of state diagrams to describe the same tasks.

Any given system behavior can be described in multiple ways using the aforementioned notations. One may prefer to use many states and events whereas another may prefer to use very few. Human understandability is one of the dominant factors in determining the number of states and events to use. In a teamwork environment, neither too many nor too

[5]Hatley [Ha 88] uses the same notation but in a different context. In [Ha 88], "don't care" may be used in decision tables when: (1) there exists some combinations of I/O that can not occur or is inconsequential; (2) there are multiple rows in the table that have identical I/O except for one column. This column with differences is marked "don't care," meaning the differences do not cause any effect.

few seem to be the best. A high level person may not want to read a state diagram with a lot of details. On the other hand, to an operator monitoring the performance of a particular RCS controller, it may be helpful to have abundant system execution information conveyed to him, via states and events. In some sense, this is similar to the determination of a sampling rate for discretizing a continuous domain problem.

One may use "out of range" as the only event for a servo controller whereas another may use two events: "below minimal allowance" and "above maximal allowance." When the controller is not performing as expected, the information, "out of range," alone may not be enough for the operator to diagnose the situation. If the operator reads that the controlled variable constantly falls "below minimal allowance," he is better informed. This would help him to find the solution to the problem.

4.2. The Course and the Ship Maneuver Controller Modules

As described in the scenario, the mission for the simulated submarine is to transit the Bering Strait. The highest level control module in the ship maneuvering system RCS is a Course controller (figure 4-1). A human operator designates a mission via a RUN_MISSION command along with starting and goal positions. The Course controller receives this mission (figure 4-2) and calculates a series of intermediate goal points for its subordinate, the Ship Maneuver controller. The primary function for the Ship Maneuver controller is the coordination of its three subordinates, namely, the Depth, Helm, and Propulsion controllers (see figures 4-1 and 4-3). Such coordination is done through evaluating its control goals and the subordinate execution status and issuing appropriate commands to each subordinate. The primary command that Ship Maneuver receives is called ICE_TRANSIT_SALIN[6]. The corresponding ship maneuvering behavior can be described as follows:

* Plan Activation: The ICE_TRANSIT_SALIN plan starts when the Ship Maneuver Controller receives the command, shown as (*0) in figure 4-3, the ship maneuver plan.

* Normal Behavior: Ship Maneuver sends the depth control, helm control, and propulsion control commands to its subordinates, as shown in the shaded box under (Evt0) in the figure. The controller normally remains in state (S1) coordinating the three major ship maneuvering activities. Note that the authors elected to limit the system design to consider only the forward motion of the submarine.

* Error Handling: Errors reported to Ship Maneuver from its subordinates will be accounted for immediately regardless of the controller's current state. In this plan, the errors are described at (Evt1) through (Evt5). Each of them is preceded by a "don't care" state, (*1) through (*5). Algorithms have been implemented to detect the errors (regarded as events). The occurrence of any of these errors implies that the control has been switched to the corresponding "don't care" state regardless of the controller's previous state. The error compensation actions, represented by the jobs and commands listed in the corresponding shaded boxes, are taken. All of these actions lead the controller to the error correction state, (S2).

[6]There is also a simplified version for this command, called ICE_TRANSIT, which does not have the capability of handling the salinity problems and is rarely used.

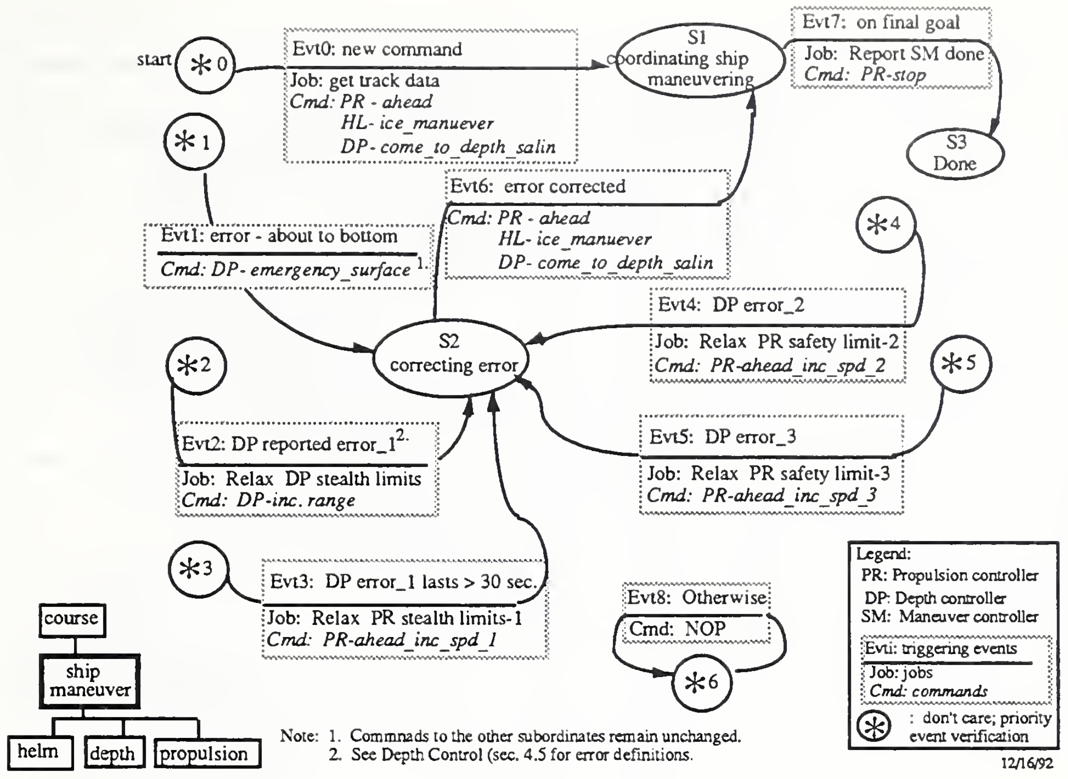


Figure 4-3 The Ship Maneuver Plan

This particular plan describes the Ship Maneuver plan which coordinates the compensation of depth control errors. If the Depth controller (see section 4.5) is unable to handle the error by itself, Ship Maneuver has the authority to change the speed constraints that it imposes on the Propulsion controller and to issue speed increase commands such as AHEAD_INC_SPEED_1 (could also be _2 or _3 depending on the severity of the problem). After a certain period of allowed response time, Depth may find that the depth error is being compensated (section 4.5).

* Completion: When the last goal has been achieved, the controller will enter the done state, (S3).

A state table for the same Ship Maneuver plan is shown in figure 4.4. This table contains the same information as in figure 4-3, but in a tabular format.

IF		THEN		
Event	Current State	Next State	Do Job-List (SP/WM/BG)	
			Computation	Commands
E0: New Command	*0	S1	WM: get track data	PR - ahead HL- ice_manuever DP- come_to_depth_salin
E1: error - about to bottom	*1	S2		DP- emergency_surface
E2: DP reported error_1	*2	S2	WM/BG: Relax DP stealth limits	DP-inc. range
E3: DP error_1 lasts > 30 sec.	*3	S2	WM/BG: Relax PR stealth limits-1	PR-ahead_inc_spd_1
E4: DP error_2	*4	S2	WM/BG: Relax DP stealth limits-2	PR-ahead_inc_spd_2
E5: DP error_3	*5	S2	WM/BG: Relax DP stealth limits-3	PR-ahead_inc_spd_3
E6: Error corrected	S2	S1		PR - ahead HL- ice_manuever DP- come_to_depth_salin
E7: on final goal	S1	S3	BG: Report SM done	PR-stop
E8: otherwise	-	*3		NOP

5/21/92

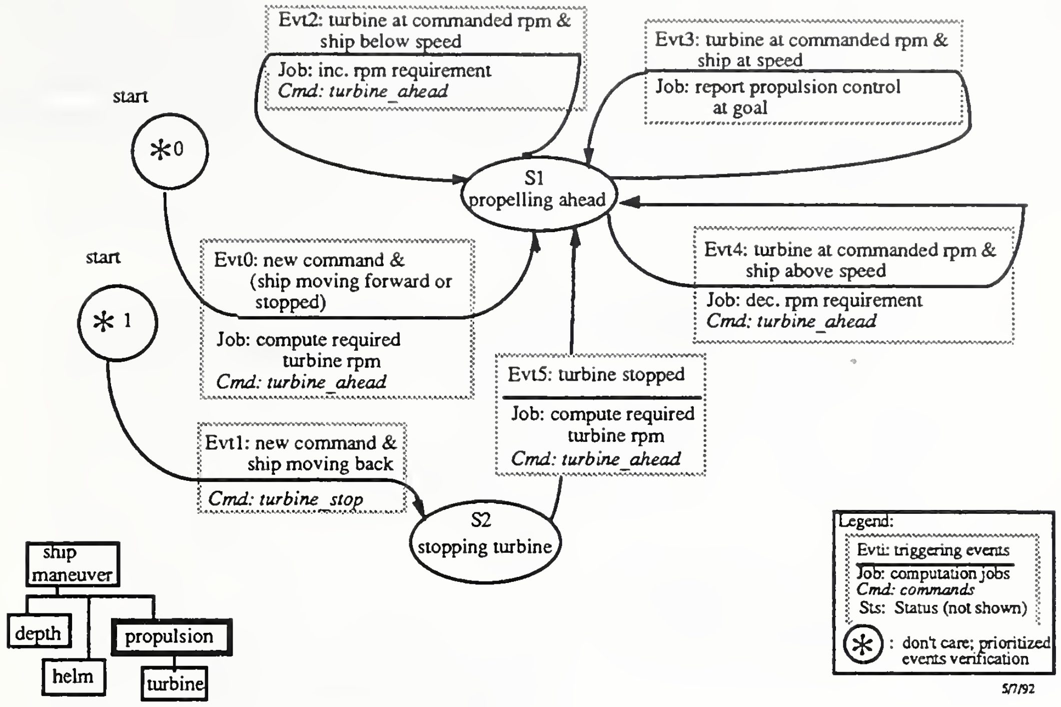
SP: sensory processing
WM: world modeling
BG: behavior generation

Figure 4-4 The Ship Maneuver Plan State Table

4.3. Propulsion Control

The Propulsion controller is responsible for the control of the ship's[7] speed (figure 4-5). The subordinate controller, Turbine, is responsible for maintaining the propeller rpm which the Propulsion controller computes. The submarine can move either forward or backward. The moving ahead behavior can be described as follows:

[7]Even though the submarine has the capability to move backward by reversing the rotation of the propeller, this is rarely done. The propeller is sometimes reversed while the submarine is moving forward to execute an emergency braking procedure.



Evt2: turbine at commanded rpm & ship below speed
Job: inc. rpm requirement
Cmd: turbine_ahead
Evt3: turbine at commanded rpm & ship at speed
Job: report propulsion control at goal
start
*0
S1 propelling ahead
start
Evt0: new command & (ship moving forward or stopped)
Job: compute required turbine rpm
Cmd: turbine_ahead
Evt4: turbine at commanded rpm & ship above speed
Job: dec. rpm requirement
Cmd: turbine_ahead
*1
Evt5: turbine stopped
Job: compute required turbine rpm
Cmd: turbine_ahead
Evt1: new command & ship moving back
Cmd: turbine_stop
S2 stopping turbine
ship maneuver
depth
helm
propulsion
turbine
Legend:
Evti: triggering events
Job: computation jobs
Cmd: commands
Sts: Status (not shown)
* : don't care; prioritized events verification
5/7/92

Figure 4-5 The "Propulsion Ahead at a Desired Speed" Plan

* Plan Activation: This plan is activated when the Propulsion Controller receives the AHEAD command. Either (*0) or (*1) in the figure will be used depending on whether the submarine is currently moving forward, stopped, or moving in reverse.

* Normal Behavior: Propulsion receives, from Ship Maneuver, the ship speed requirements which are functions of the stealth constraints, the under-ice maneuver constraints, etc. Desired ship speed and turbine rpm are computed from these requirements (shown under (Evt0) in the figure). The relationship between the turbine rpm and the ship speed is generally nonlinear due to factors such as variations in the water current profiles.

(Evt1) in figure 4-5 specifies that the turbine has to be stopped if the ship is currently moving in the opposite direction. Otherwise, the Propulsion controller stays in the state (S1) and servos on the desired ship speed. It sends the required turbine rpm to its subordinate, the Turbine controller. The Propulsion controller must recompute the required turbine rpm if the previously commanded rpm fails to keep the ship at the desired speed. (Evt2) and (Evt4) describe such an effect.

* Error Handling: Not implemented in this plan.

* Completion: This servo plan does not have a done state. Instead, whenever the ship speed becomes within the tolerance of a desired speed, a "propulsion control at goal" status will be reported to Ship Maneuver.

The Turbine controller receives and servos on the desired turbine rpm. Figure 4-6 shows a typical primitive servo control state diagram for a regulator problem. The Turbine controller stays in (S1), monitors the rpm control error, and generates the signals for increasing or decreasing the rpm based on the control error. The turbine simulator receives the signals and computes the corresponding rpm, which are sensed and fed back to the Turbine controller. If the rpm is within a pre specified tolerance, an AT_GOAL message is returned as the status to the Propulsion controller. See section 4.1 for the reasons of using multiple events (Evt0 through Evt3).

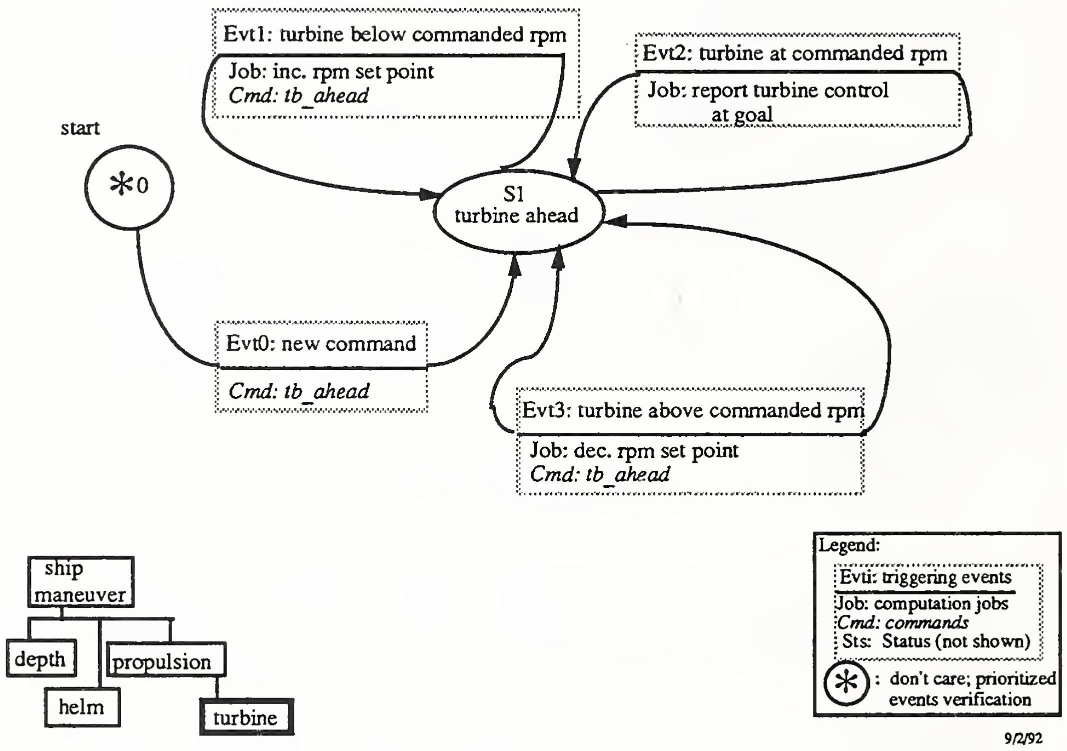


Figure 4-6 The "Turbine Ahead at a Desired rpm" Plan

4.4. Helm Control

The Helm controller is solely responsible for the ship heading control. The control behavior, as shown in figure 4-7, can be described as follows:

* Plan Activation: The ICE_MANEUVER plan starts when the Helm controller receives the command, shown at (*0) in the figure.

* Normal Behavior: A desired course is computed based on the next goal position Helm receives, as shown in the shaded box under (Evt0) in the figure. The required rudder angles are computed based on the course control error. If there is no obstacle (mainly ice keels, as discussed below), the Helm controller will send the computed rudder angles to its subordinate, the Rudder controller, to approach the goal position. (Evt3), (Evt4), and (Evt5) in the figure describe such servoing activities.

* Error Handling: Significant errors will be accounted for immediately regardless of the controller's current state. In this plan, the error entitled "ice problem flag persists for longer than 5 seconds" is regarded as significant. This error is described in the figure as, (*1), a "don't care" state, and a label (Evt1). The detection of this error is performed in the control preprocessing. This detection causes the control to switch to the corresponding "don't care" state no matter what the controller's previous state was. The error compensation actions, represented by the jobs and commands listed in the corresponding shaded boxes, are taken. All these actions lead the controller to the error correction state, (S2).

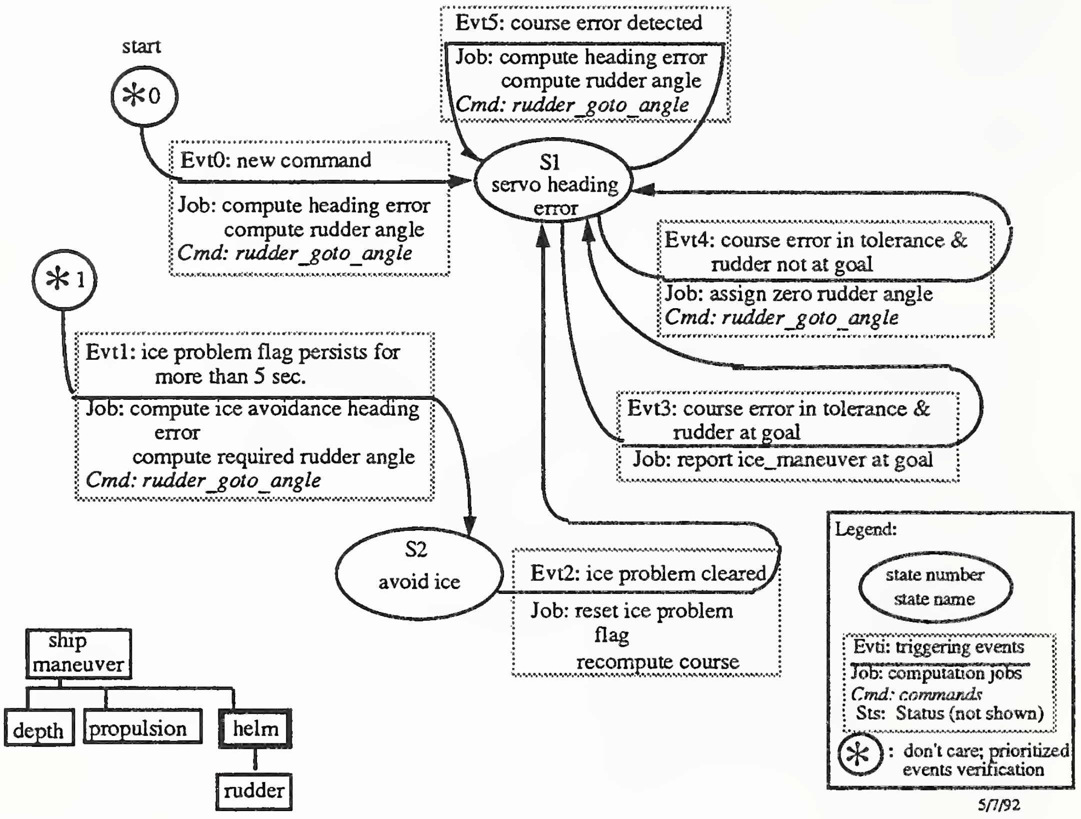
  The Helm controller employs a sensory processing function known as the Cerebellar Model Articulation Controller (CMAC) algorithm (see sections 3 & 6) for developing an ice map. Sonars detect the ice [Hu 92]. CMAC receives the sonar data, generalizes for an estimated ice distribution map, and stores the map in the control system world model. Helm also employs a path planning algorithm (as part of Helm's Behavior Generation function) that computes an ice avoidance recommended heading based on the ice distribution and the desired course. Inside the Helm controller, an ice_problem flag is raised if the ice avoidance recommendation differs from the desired course. The persistence of the ice problem for a predefined period of time is an error condition, (Evt1) and (*1). The desired course will be temporarily omitted and the Rudder controller will be given an ice avoiding rudder angle. See the description under (Evt1). A new course toward the original goal must be computed after the ice has been avoided, described at (Evt2). Under the, "no ice," situations, the ice avoidance recommendations will be consistent with the desired course.

  The computation of the desired rudder angles depends on the size of the heading error. As the error reduces to be within some pre-calculated tolerance, a zero angle command must be sent in advance (before the heading error reaches zero) to account for the inertia of the submarine motion.

* Completion: This servo plan does not have a DONE state. Instead, an "ice_maneuver_at_goal" status, at (Evt3), would be reported to Ship Maneuver once Helm and Rudder are within the specified tolerances. The Helm controller continues in state (S1) monitoring any possible heading deviation or the occurrence of an ice problem.

The Rudder controller always receives a GOTO_ANGLE command together with a targeted angular value for use in servoing. Simulated +VOLT, -VOLT, and 0VOLT signals are sent to the rudder simulator to generate the corresponding rudder angles.

As described in an earlier paper [Hu 91], one important feature a designer may discover as he steps from the higher levels down to the lower levels in an RCS hierarchy is the transition of coordinate systems from global systems with coarse resolutions to local systems with finer resolutions. The Helm controller refers to headings, a measurement global to the world, whereas Rudder refers to rudder angles, local to the submarine's center line. The output of the Rudder controller (voltage) is local to the electro-mechanical rudder mechanism.



start
*0
Evt0: new command
Job: compute heading error
compute rudder angle
Cmd: rudder_goto_angle
Evt5: course error detected
Job: compute heading error
compute rudder angle
Cmd: rudder_goto_angle
S1
servo heading error
Evt4: course error in tolerance & rudder not at goal
Job: assign zero rudder angle
Cmd: rudder_goto_angle
*1
Evt1: ice problem flag persists for more than 5 sec.
Job: compute ice avoidance heading error
compute required rudder angle
Cmd: rudder_goto_angle
Evt3: course error in tolerance & rudder at goal
Job: report ice_maneuver at goal
S2
avoid ice
Evt2: ice problem cleared
Job: reset ice problem flag
recompute course
ship maneuver
depth
propulsion
helm
rudder
Legend:
state number
state name
Evti: triggering events
Job: computation jobs
Cmd: commands
Sts: Status (not shown)
*: don't care; prioritized events verification
5/7/92

Figure 4-7 The Helm Control Plan

One point worth noting is the importance of a consistent design specification across a team of designers. In this implementation, headings are computed in three modules, the Helm controller, the ship simulator, and the CMAC algorithm. Significant debug time was spent in the integration period due to the fact that three modules used different references for their heading computations. All the computed headings have to be normalized before being used.

Future enhancements to this ice avoidance algorithm include adding changing ship depth and speed as some of the options to avoid ice and adding some ice problem severity indices.

4.5. Depth Control

As described earlier, the depth of the submarine can be controlled using either the control surfaces or the ballast. The Depth controller employs a Dive/Rise controller and a Main Ballast controller for these purposes (shown in figure 4-1). The Dive/Rise controller is responsible for using the control surfaces to achieve the desired depth that the Depth controller specifies. The Main Ballast controller is responsible for the buoyancy force control on a gross scale. The main ballast tanks can be blown to surface the ship in emergency situations. There is also variable ballast control on the ship used for such purposes as maintaining the depth or adjusting the trim (pitch angle) for the ship. They are omitted in figure 4-1 since they were not used in the performance of the specified scenario.

Upon receiving a goal, Ship Maneuver typically sends a COME_TO_DEPTH_SALIN[8] command, along with a desired depth, to the Depth controller (figure 4-8). The control behavior can be described as follows:

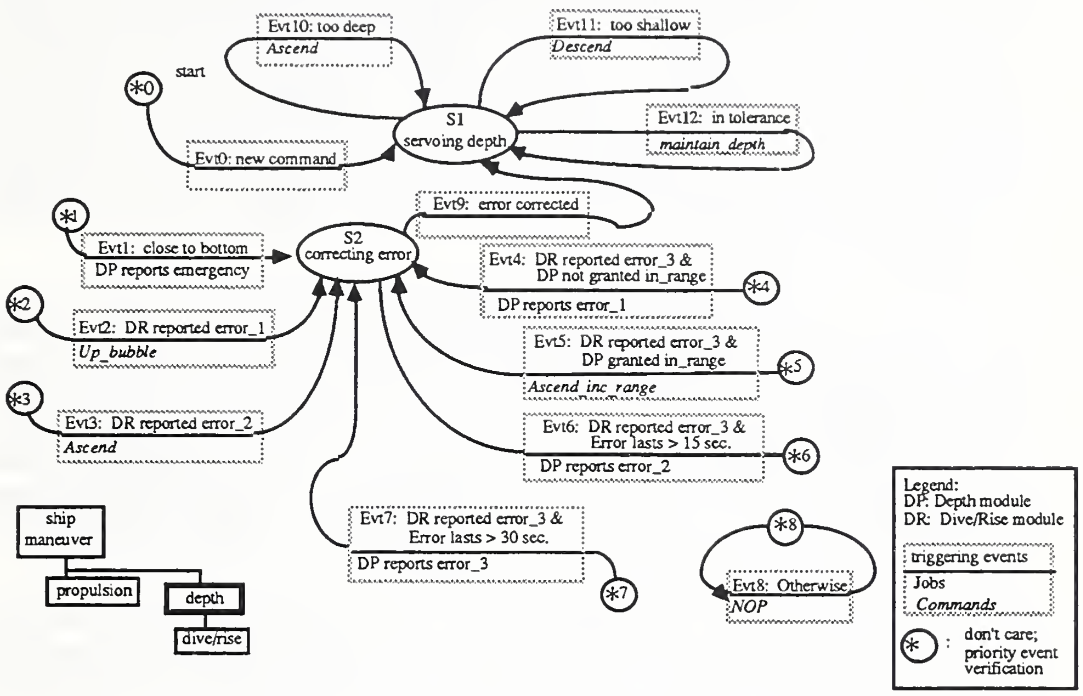


Figure 4-8 The Depth Control Plan

* Plan Activation: The depth control plan is activated when the Depth Controller receives the command, shown at (*0) in the figure.

* Normal Behavior: The Depth controller would normally be in (S1). It selects the ASCEND/DESCEND commands for the Dive/Rise controller to achieve the desired depth (refer to the scenario for a description of the reason that UP-BUBBLE is not being used as a preferred method). After the ship comes within the depth tolerance, the MAINTAIN_DEPTH command will be activated. The Depth controller remains executing the same COME_TO_DEPTH_SALIN command and the Dive/Rise controller remains in the state of maintaining the depth unless an error occurs.

* Error Handling: Errors reported to Depth from its subordinate, Dive/Rise, will be accounted for immediately regardless of the controller's current state. In this plan, these errors are described at (Evt1) through (Evt7). Each of them is preceded by a "don't care" state, (*1) through (*7). The occurrence of any of these errors causes the control to switch to the corresponding "don't care" state no matter what the

[8]The label SALIN in the command name indicates that this command is capable of handling salinity anomalies. There also exists a simplified version called COME_TO_DEPTH which does not have such capability.

controller's previous state was. The error compensation actions, represented by the jobs and commands listed in the corresponding shaded boxes, are taken. All these actions lead the controller to the error correction state, (S2).

The Dive/Rise controller contains a proportional-and-derivative (PD) type of depth error detection algorithm. This algorithm compares the submarine actual vertical speed and depth with the desired values. Whenever the differences exceed a first set of assigned thresholds, an ERROR_1 flag will be reported to Depth, as described at (*2) and (Evt2). The UP_BUBBLE command would be selected for the Dive/Rise controller in this case (see section 3, scenario, for the reason). The Depth controller is now in the state (S2). If ERROR_1 persists for a predefined time or if the speed or depth errors exceed a second set of thresholds (more severe), another ERROR_2 flag, as shown at (*3), will be reported, and the ASCEND command will be activated.

ERROR_3, at (*4), can be received similarly, prompting Depth to report DP_ERR_1 to Ship Maneuver. Ship Maneuver would exercise its authority to relax the stealth/safety constraints and would grant Depth permission to use larger control surface operation ranges (see (*5) here and (*2) in figure 4-3). If the Dive/Rise ERROR_3 error persists, the Depth controller reports DP_ERR_2 and DP_ERR_3 up, shown as (*5) through (*7). Ship Maneuver will relax more stealth/safety constraints and will issue requests to the Propulsion controller to increase speed (see (*3) through (*5) in figure 4-3). The system may re-enter (S1) from (S2) whenever the error is corrected and the normal depth control servo loop resumes.

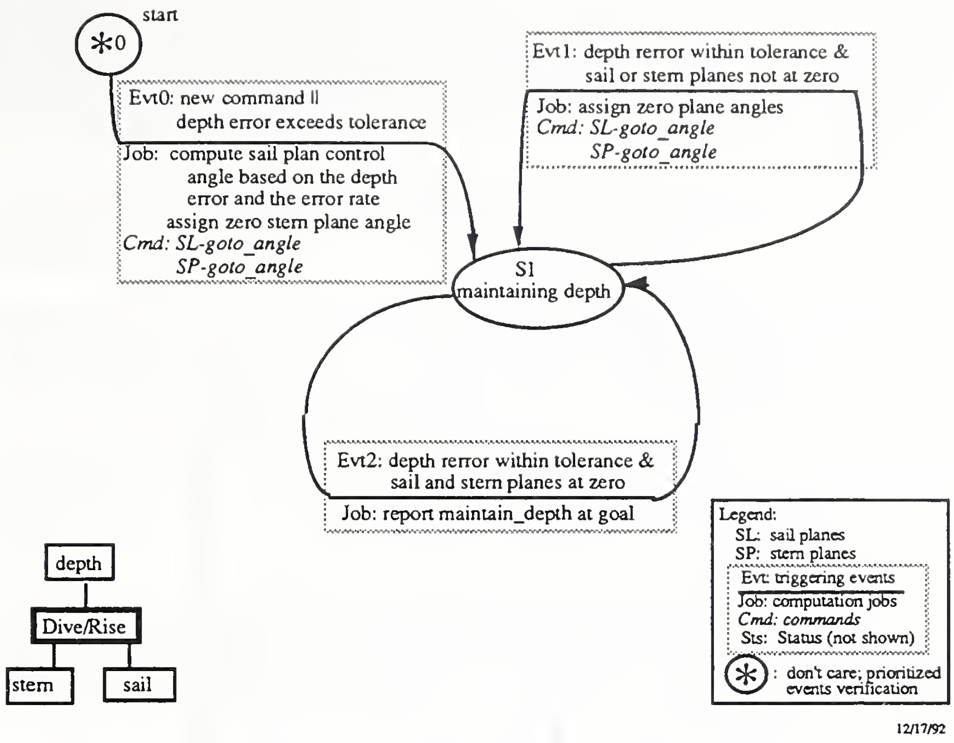
Note that the CLOSE_TO_BOTTOM event, (Evt1) in the figure, takes even higher priority such that whenever it is detected, the error must be reported immediately. Eventually the Ship Maneuver controller will issue an emergency command to blow the main ballast tanks to surface the ship (figure 4-3).

* Completion: The depth control would normally stay in (S1) to maintain the given desired depth and does not have a completion state.

Note that this plan assumes that the depth control is operated in the automatic mode, see section 6.5 for an interactive mode depth control plan.

The MAINTAIN_DEPTH command (figure 4-9) uses the sail planes to keep the ship within the tolerances of desired depths. The stern plane angles are typically set to zero in this plan. As shown in the figure, with the occurrence of Evt0 (when the depth error exceeds the tolerance), one job for the Dive/Rise controller is to apply the PD control algorithm to compute a sail plane control angle in order to compensate for the depth error. Evt1 shows that the plane angles are reset to zero once the depth becomes within tolerance. Once this is achieved, status is reported (Evt2) and the depth monitoring continues.

All the Dive/Rise commands are decomposed into the SL_GOTO_ANGLE and SP_GOTO_ANGLE commands together with desired plane angles. The Sail Plane and the Stern Plane servo controllers execute the respective commands.



start
*0
Evt0: new command ||
depth error exceeds tolerance
Job: compute sail plan control angle based on the depth error and the error rate
assign zero stem plane angle
Cmd: SL-goto_angle
SP-goto_angle
Evt1: depth rerror within tolerance &
sail or stem planes not at zero
Job: assign zero plane angles
Cmd: SL-goto_angle
SP-goto_angle
S1
maintaining depth
Evt2: depth rerror within tolerance &
sail and stem planes at zero
Job: report maintain_depth at goal
depth
Dive/Rise
stem
sail
Legend:
SL: sail planes
SP: stem planes
Evt: triggering events
Job: computation jobs
Cmd: commands
Sts: Status (not shown)
*: don't care; prioritized events verification
12/17/92

Figure 4-9 The Maintain Depth Plan

5. COMPUTER ENVIRONMENT

5.1. Background

The RCS methodology suggests that the designer map software to the target hardware explicitly. Although the submarine work was a demonstration, careful attention was paid to the mapping of software to hardware, see figure 5-1. The choices made were based on resource availability and performance criteria for a demonstration system. If one developed a real system, the software could be mapped differently.

5.2. Hardware

Two platforms were used in this demonstration. A Gateway 2000 PC compatible 386/33 computer was used as the main development workstation for the controller and simulation. A Silicon Graphics Incorporated (SGI) 4D/220VGX workstation was used for the animation and some environmental simulation. A Bit3 PC to VME Bus extender card set and cable were used to facilitate communication between machines. More detail on data transfer between machines is provided in section 6.4.4.

5.3. Software

5.3.1. Development Software

Microsoft C version 6.00 for the PC was used to develop the RCS control and simulation software. CodeView, the Microsoft source level debugger, was used extensively during the development process, particularly for verifying command traversing. The SGI graphics library (GL) in C was used for the submarine animation and simulation. Software provided by Bit3 was used for the data transfer between the SGI and PC compatible.

Submarine Demo #3 Computer Resources

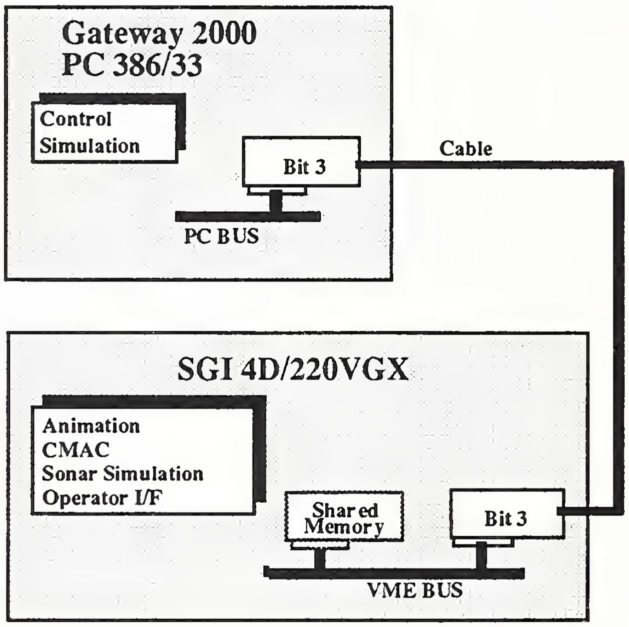


Figure 5-1

5.3.2. RCS Software

The control hierarchy and simulation were implemented on the PC compatible machine, which was operated at a thirty millisecond sample period. The executable software required seven hundred kilobytes of memory on the PC compatible. The animation, CMAC, ice and sea bottom profiles, and sonar simulation were implemented on the SGI workstation, and the executable code requires nearly one megabyte of space. The software distribution was done to ensure that any real-time critical code was executed at the thirty millisecond heartbeat (system cycling time). The sonar simulation and CMAC distribution did not need to operate at the heartbeat rate, which demonstrates the ability of RCS to handle asynchronous communications seamlessly.

6. SYSTEM IMPLEMENTATION

6.1. Overall Software Architecture

One aspect of the RCS methodology is a generic software architecture, which facilitates the development of RCS applications. The research results obtained in demo #3 show that such an architecture may include the following hierarchies: RCS controller hierarchy, human control interface hierarchy, simulation hierarchy, human simulation interface hierarchy, and animation hierarchy. These hierarchies are served by a generic communication mechanism and by partitioned knowledge bases and shared memory. While a proposed layout is shown in figure 6-1, extensive study in future phases of this project is required to verify the format of some aforementioned hierarchies. Descriptions for all the components of the proposed software architecture are given as follows:

* RCS Control Hierarchy (shown in the upper left quadrant of figure 6-1): Executes the plans, as described in section 4, to perform real-time ship maneuvering automatic control.

* Human Control Interface Hierarchy (the upper right quadrant of figure 6-1): Serves as the interface between the above mentioned RCS hierarchy and human operators who use decision aiding in performing interactive control.

* Control System's World Model versus Simulation World Model (shown as part of the central core in figure 6-1): Two distinct world models are used. Each contains sets of state variables, dynamic models, and geometry models to represent the world.

* Communication (depicted by arrowed lines in figure 6-1): Communication is required among all the components described above and is achieved by utilizing a generic shared memory mechanism. There are different types of communication activities in this RCS software structure, including:

  - Command/Status Communication: This type of data is shared only by two adjacent modules along the command chain in the hierarchies, as described in section 6.4.1.
  - World Model Data Communication: This type of data can be shared by all the modules in the hierarchies and therefore is considered as part of the world model [Al 92]. Section 6.4.2 provides more description.
  - Human Computer Interaction. This is described in section 6.4.3.
  - Communication with Other CPU's. This is described in section 6.4.4.

* Global Memory (shown as the central core in figure 6-1): The global memory includes the world models and the memory space that facilitates the aforementioned shared memory based system communication.

* Simulation Hierarchy: Simulation receives actuator commands and computes ship dynamics. Environmental objects/phenomena of concern are also simulated.

* Human Simulation Interface Hierarchy (the lower right quadrant of figure 6-1): Changes the simulation (including ship or environmental) parameters to test the responsiveness of human operators or the RCS ship maneuvering system.

* Animation (at the bottom of figure 6-1): Paints the pictures in real-time based on the data computed in the simulation software.

As reported in the following sections, all the five hierarchies use essentially the same format to implement their software modules. All the hierarchies execute via command passing and shared memory communication. Therefore, in some respect the execution of the whole demonstration can be viewed as five parallel sets of hierarchical control behavior. In the future, the focus of studies should include:

* the format of the two human interaction hierarchies -- would they form a one-to-one correspondence with the control and the simulation hierarchical modules?
* the coordination of the human interactions among multiple hierarchical levels.

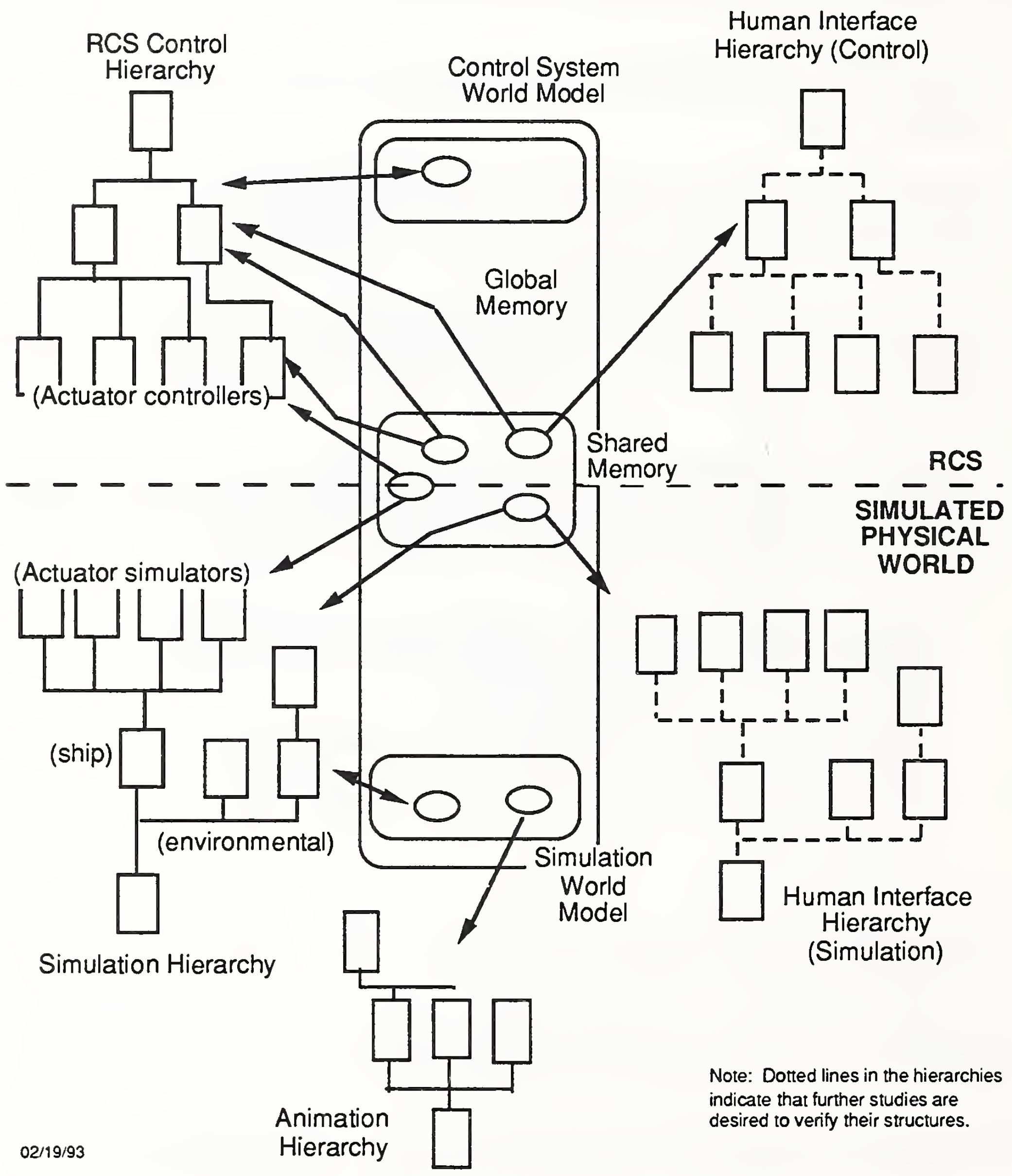


Figure 6-1: Proposed Software Architecture

6.2. Software Structure for the RCS Hierarchy

This section describes the software structure of the implemented RCS hierarchy. First, the main executing program is introduced, then the overhead associated with keyboard interaction and debug displays. Lastly, the generic controller template is introduced as the building block for all of the controller modules. The control software is described in detail in this paper; for information regarding the animation and simulation refer to [Hu 92].

6.2.1. Main Program

The main program organizes the execution of the control modules, simulation modules, debug displays, and animation communication. All of the code was written in C, but other languages have been used successfully, e.g., SMACRO (a superset of FORTH). The main program acts as the scheduler for the cyclically executing system. If multiple CPUs are used for control, each CPU would require a separate main program. A block diagram of the main program for the submarine control is shown in figure 6-2 and a listing of the code is provided in Appendix D. Much of the main program could be copied to other processors verbatim; however, some of the function calls are computer specific. The main program can be thought of as our Real-Time Executive (RTE) [Be 88].

6.2.1.1. Allocate Global and Main Memory

The first step in the main program is to allocate the Global Memory (GM). This memory consists of: world model data, commands, status, simulation model data, and debug statistics. These data structures are defined in a header file and were determined a priori, but this fact does not preclude the designer from utilizing more complex data structures with RCS.

After the GM has been allocated, the valid commands for each module in the hierarchy that may be input are determined. These values are read in from a file that may be updated without the need for recompilation.

One of the most important variables to be set is the state clock interrupt. This constant sets the sample rate of the computer controlled system, because every execution cycle is initiated at the rate of this heartbeat. This makes the maximum execution time of the control system deterministic. In the submarine demonstration, the programmable interval timer for the PC was set at 211 µs, which is fine enough resolution for our control system and the simulation calculations used for dynamic system response. The interrupt causes a timer_counter in the GM to be incremented; therefore, the real-time elapsed is 211 µs X timer_counter. The sample rate is set to an integer multiple of the interrupt time of 211 µs. In this demonstration, the timer_counter increments to 142, approximately 30 ms, before beginning another execution cycle.

The PC user interface is initiated next, by drawing the initial screen.

Once the Global Memory space has been allocated, its initial conditions are set. This allows the designer to set up a number of different "what if" type experiments efficiently. For example, an experiment may begin with the submarine on the surface or submerged. If it was desired to have the initial condition to include the set of all possible states of the system, all state variables would have to reside in the GM, including the intermediate values used for dynamic system response calculations. For a dynamic system as complex as a submarine, this is a difficult problem. In this project, only variables of global interest (ones which are shared between modules) reside in the GM. The world and simulation models are set to the desired values and an INIT command is propagated through the hierarchy on initialization. Commands and status are contained in the GM. The actuator commands are set to their appropriate initial values. For example, it might be desirable to start the experiment with the submarine at zero speed, so the turbine actuator would receive a ZERO-VOLT command.

After all of the initialization is completed, the system is ready to execute.

RCS Templates

Main Program

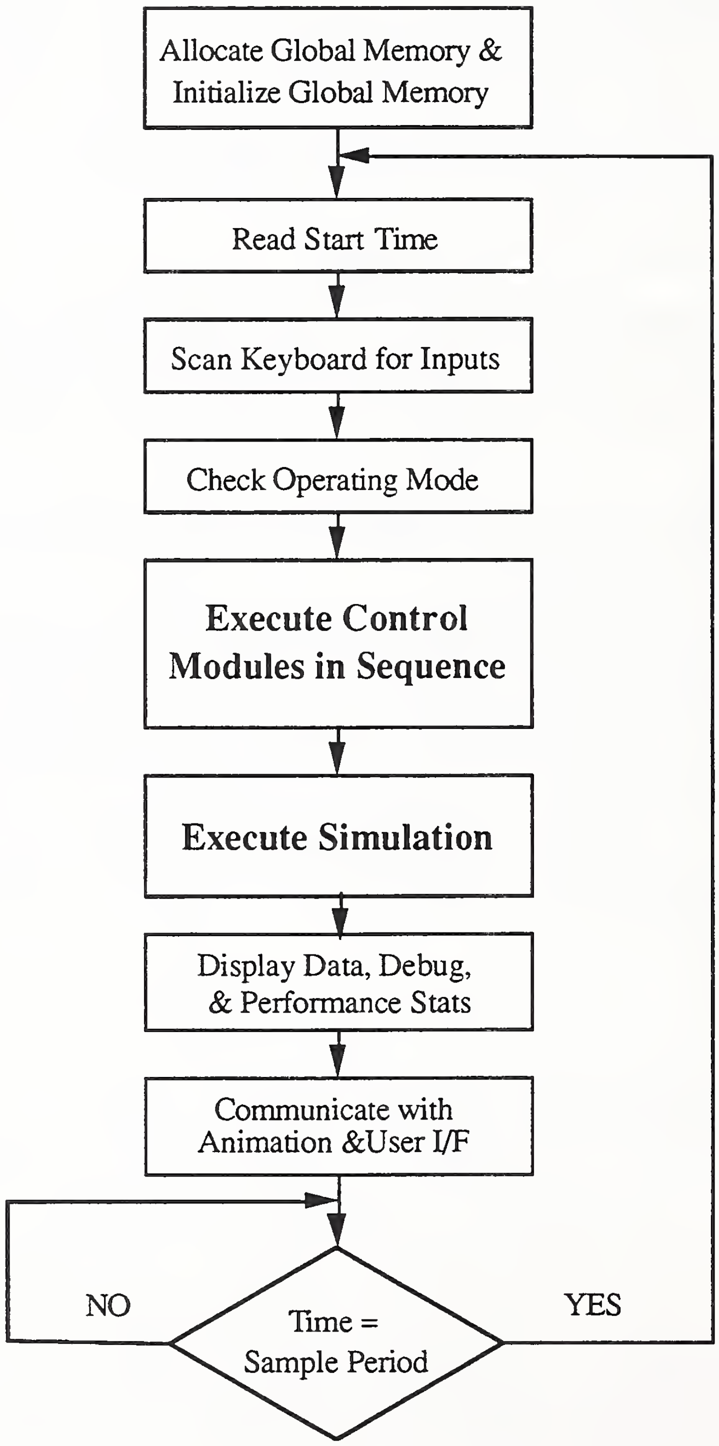

Allocate Global Memory & Initialize Global Memory
Read Start Time
Scan Keyboard for Inputs
Check Operating Mode
Execute Control Modules in Sequence
Execute Simulation
Display Data, Debug, & Performance Stats
Communicate with Animation &User I/F
NO
Time = Sample Period
YES

Figure 6-2

6.2.1.2. Cyclic execution

Figure 6-2 shows the cycle that is executed every heartbeat, beginning with "Read Start Time" and ending with "Time = Sample Period." The cycle repeats approximately every 30 ms in this implementation, which translates into 142 ticks of the timer_counter.

This cycle begins with reading the start time, used for synchronization purposes. The controller modules are run next, then the simulator modules. The communication between the CPUs is run, the debug displayed, and then the execution performance is calculated. Lastly, the program waits for the rest of the 30 ms to elapse. Note that even though there will be time where the CPU is idle, waiting for the 30 ms to elapse, it is necessary to remain on that constant heartbeat. The system should be deterministic, and as such, should have a consistent sampling rate. Varying sample rates affect control response, which is deemed as undesirable.

It should be emphasized that every control module is executed at each cycle. This is true regardless of the planning and execution. This may seem strange at first, but it is a crucial element of the RCS methodology. This approach does not preclude the use of time intensive algorithms; however, algorithms must be organized properly with respect to the hardware available, which is true of any design. In the submarine demonstration, this problem arose with the CMAC and sonar simulation, so those algorithms were assigned to run on separate hardware. The details on CMAC and sonar simulation implementation is presented in [Hu 92].

The sequence of execution for the control modules can be of some importance, particularly when the control is executed on one processor. RCS fully supports a multiple parallel processor design. However, when one designs using a single processor, the ordering of execution for the control modules has interesting consequences. If execution is done top down, i.e., the highest level modules are executed, then the next level and on down to the actuator level, then the commands will propagate from the highest level down to the lowest level in one control cycle. However, status and error information traveling back up the hierarchy will have a lag equal to the number of controllers in that particular thread. If there is an emphasis on fast response to errors, execution sequencing should be bottom up. This choice is left up to the designer.

Only relevant data is transferred between the PC and SGI; i.e., submarine position and orientation. For example, the CMAC recommended heading is transmitted to the PC.

Debug information is calculated and displayed. The actual execution cycle time is then calculated, which should be less than the sample period.

Another debug feature is to single step the cycle, allowing the designer to watch the propagation of commands and execution of the system.

Lastly, the CPU waits for the clock to reach the sample period. After this occurs, the cycle is restarted. This requires that the execution of each cycle *must* take less than 30 ms. If execution time is greater than 30 ms, then extra hardware might be required for proper execution

6.2.2. Overhead

The overhead is presented to provide guidelines for implementations. The implementation may have to be machine specific, but the concepts can be readily employed on most machines.

6.2.2.1. State Clock Timer

The heartbeat is calculated from a hardware programmable interval timer in the PC. The timer runs an interrupt every 211 µs. The interrupt servicing routine (ISR) consists of incrementing the counter. The program returns to normal operation after the ISR. This results in a resolution on time calculations of 211 µs.

Interrupt servicing routines should consist of a minimal number of calculations, and not any program execution. This is one of the tenets of RCS [Qu 92], and ameliorates the non-determinism which may result from traditional real-time programming techniques. Large complex systems can become extremely difficult to manage when many interrupts have to be processed.

6.2.2.2. Keyboard Input

The keyboard on the PC is scanned for command inputs. If a key is hit, it is read into a command buffer. Once the command buffer is full, designated by a carriage return input, action is taken on the command. A maximum of one key is read in per cycle. This operation takes approximately 1 ms to complete, which is small compared to the execution cycle maximum of 30 ms, and will not cause the execution cycle to exceed the 30 ms. This provides the system with *non-blocking I/O*; therefore, the control system execution continues even with keyboard inputs and commands.

Keyboard commands are translated either into debug operations or as commands sent to any controller, with these commands written directly into the GM. See section 6.4 for communications and protocols.

6.2.2.3. Display Mechanism

Debug screens are used on the PC for tracking data, commands, and execution times. However, displaying the data via MS-DOS is extremely time intensive. The solution to the problem was to write the display characters directly to the VGA display card. It still requires a significant amount of time; therefore, the cyclic execution time varies depending on the amount of characters on the debug display. There are a number of screens displaying different debug information.

6.2.3. Generic Controller Template

We, at NIST, believe very strongly that a generic controller template is one of the keys of making RCS a robust, extensible, verifiable, and efficient software design methodology for attacking large scale automation projects. The controller template presented was utilized for all of the software modules developed on the submarine automation project. Once the basic system was coded, extensive revisions were made to add enhanced functionality without scrapping any of the existing code. For example, the supervisory and training sections were added onto an existing piece of code with little or no rewrite. Presented in the next few sections is an analysis of the generic controller template and a sample is provided in Appendix D.

RCS Templates
Generic Controller Module

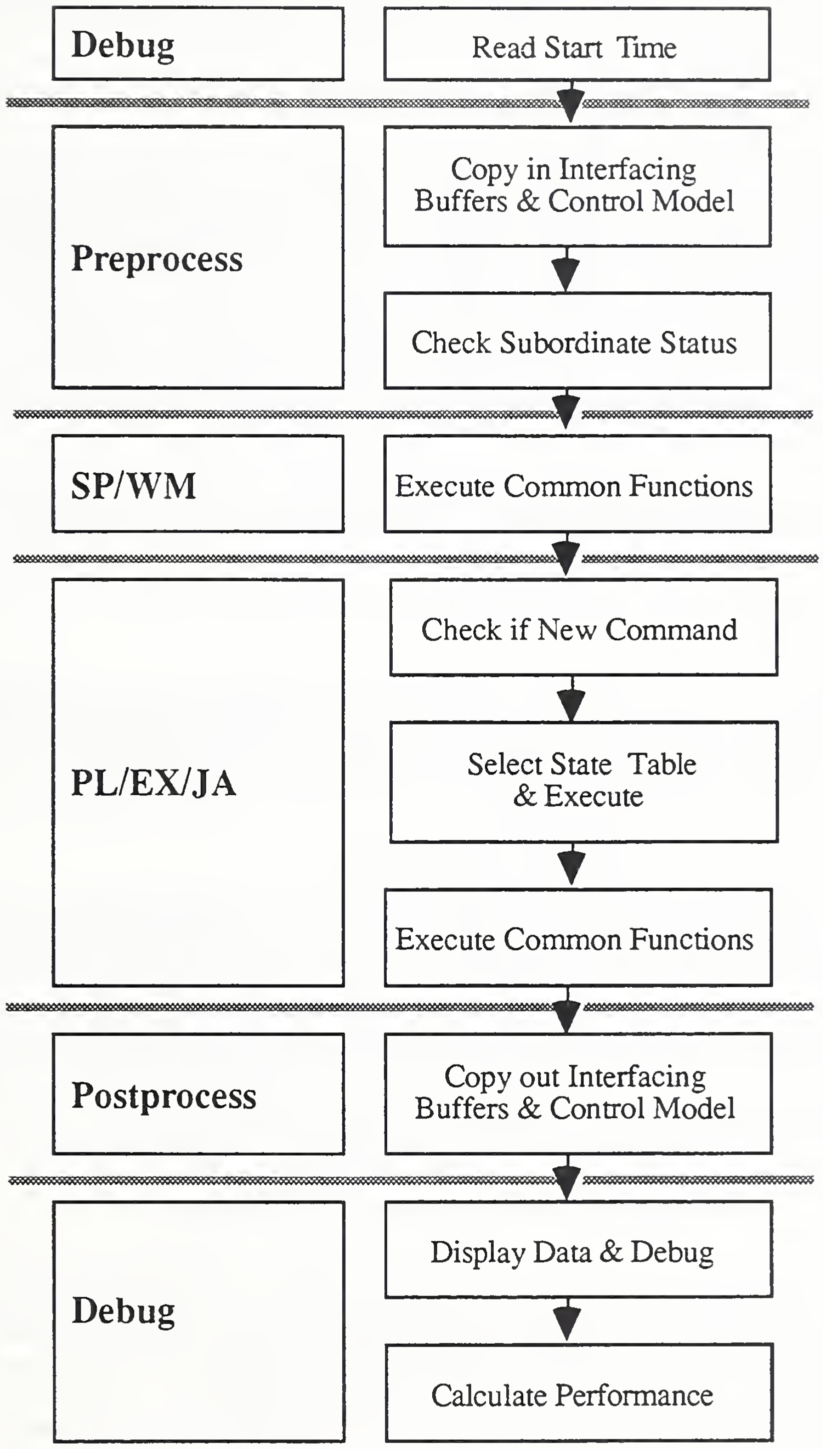


Figure 6-3

6.2.3.1. Time

The first operation the module performs is to read the start time of the module execution and store a copy. This is an important performance measurement/debug feature which provides the designer the ability to flag modules which may be taking an inordinate amount of time to execute. If a module requires a significant amount of execution time, it may be assigned to a separate processor.

6.2.3.2. Interface buffer

The Generic Controller flow chart is shown in figure 6-3. The structure of the software and communication channels is shown in figure 6-1. All communication is completed via the communication buffers located in the GM. The data in the GM has an important property, it has multiple readers and a single writer. The owner (writer) of a piece of data is predetermined by the designer. The communication buffer is the repository for the inter-module commands, status, and the sensor feedback from the submarine simulation. Actuator simulators are treated as modules. The communication system provides for a multiple processor system to be implemented without major modifications from a single processor design.

Data is generally read in from the GM interface buffers during preprocessing and written out at the end of module execution during post processing. For a more in-depth analysis of the interface buffers, please see section 6.4.

6.2.3.3. Preprocessing

Preprocessing code is executed each cycle. This includes copying information from the interface buffers and control world model. The control world model is the repository in the GM which contains modeling information used by the controllers, and its function is described in more detail in section 6.3.

6.2.3.4. Sensory Processing/World Modeling

Sensory Processing/World Modeling (SP/WM) consists of all filtering and fusing of data within the GM which is used by a module to determine appropriate actions. It is also used to update the current world model for future reference. Execution occurs at each cycle, regardless of the incoming command or outgoing status; however, different algorithms may be completed depending on the state and mode of the module. An example of SP/WM is the pressure sensor transformation to depth. The results of SP/WM are stored in GM so that all of the modules may view the data. Any algorithm that must execute on every control cycle (e.g., path planners) should be implemented as part of preprocessing.

6.2.3.5. Planning/Execution/Job Assignment (PL/EX/JA)

These three sub-modules have been lumped together as Behavior Generation (BG), because in the implementation they are intertwined. Command numbers as well as commands are passed from supervisor modules to their respective subordinates. The numbers serve two purposes, to flag new commands and to detect communications breaks between modules. The current command number is compared to the previous one and if they differ, the subordinate detects a new command. This process is particularly useful when the command received is the same, but the supervisor wants the module to restart the plan. The number is echoed back by the subordinate to the supervisor in its status report, which enables the supervisor to see that the command has been received by the subordinate. If the status number does not match the last number the supervisor sent, the supervisor may assume that the subordinate is still executing the last command sent. However, if this persists, it might be an indication of a communication breakdown. The supervisor then must decide on any corrective actions to be taken. The handshaking protocol provides the system designer the flexibility to shift modules between different processors with ease.

After the command is deemed to be new or the same, the module selects the state table (plan) to execute. The correct plan is selected by the command and state of the system. Once the proper plan has been selected, a plan line is executed. Line execution is module specific and may require a specific algorithm to run or may assign a task to its subordinates. The execution line is determined by the state of the system and variables calculated by the SP/WM functions. Examples of state tables and state graphs are provided in section 4 and Appendices B and C.

6.2.3.6. Post-processing

Post-processing code is also executed each cycle, and includes copying information to the interfacing buffers and control model. It may also include some SP/WM processing before information is posted to the interface buffer. The data posted includes: the status, commands to subordinates, world model data, and actuator commands.

6.2.3.7. Debug

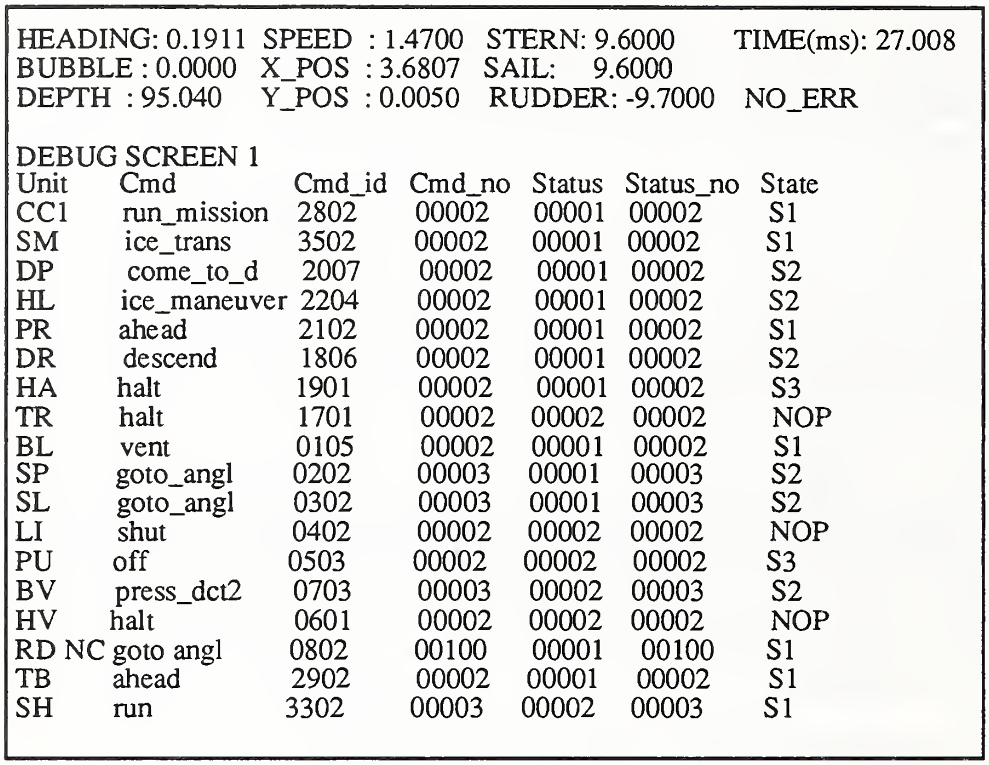
Debug in our generic controller modules consists of tracking:

* commands
* command and status numbers
* status (executing, done, error, etc.)
* current plan line being executed (what state the module is in)
* minimum, maximum, and current execution time performance
* mode (single step or free running).

The commands and status numbers have been described above. The current plan line that was triggered provides information on the execution of a particular plan. It allows the software engineer to debug the logic behind a particular plan and demonstrate which actions are being triggered; i.e., it provides a means for traversing a state graph. The performance calculations are one of the most important metrics of RCS, because they are crucial to enable the designer to recognize bottlenecks in a system.

Once the modules which require the most computing time are flagged, the designer may choose to either leave the system intact, or split the module into more than one. The splitting of the module should be done hierarchically; i.e., the module should be split into more than one level. Splitting a module vertically does not ease the computational burden, but may facilitate the debugging process. An alternative would be to dedicate a special processor for that module.

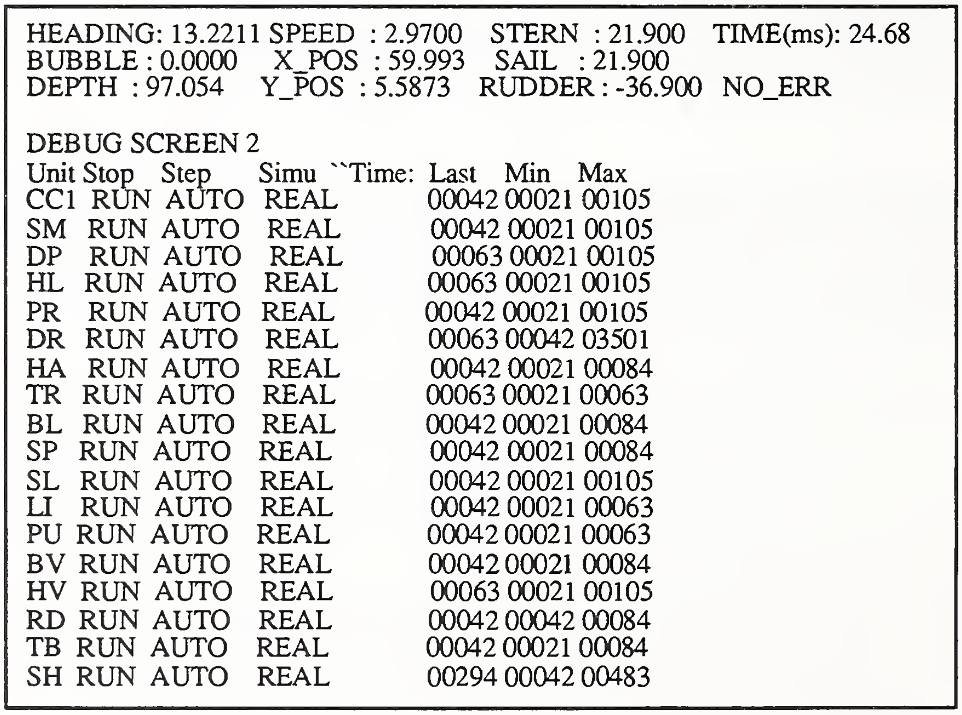
The system may be run in single step mode in order to catch errors in logic and communications, and is preferably done with a simulator.



```
HEADING: 0.1911  SPEED  : 1.4700  STERN: 9.6000     TIME(ms): 27.008
BUBBLE : 0.0000  X_POS  : 3.6807  SAIL:    9.6000
DEPTH  : 95.040  Y_POS  : 0.0050  RUDDER: -9.7000   NO_ERR

DEBUG SCREEN 1
Unit    Cmd           Cmd_id  Cmd_no  Status  Status_no  State
CC1     run_mission   2802    00002   00001   00002      S1
SM      ice_trans     3502    00002   00001   00002      S1
DP      come_to_d     2007    00002   00001   00002      S2
HL      ice_maneuver  2204    00002   00001   00002      S2
PR      ahead         2102    00002   00001   00002      S1
DR      descend       1806    00002   00001   00002      S2
HA      halt          1901    00002   00001   00002      S3
TR      halt          1701    00002   00002   00002      NOP
BL      vent          0105    00002   00001   00002      S1
SP      goto_angl     0202    00003   00001   00003      S2
SL      goto_angl     0302    00003   00001   00003      S2
LI      shut          0402    00002   00002   00002      NOP
PU      off           0503    00002   00002   00002      S3
BV      press_dct2    0703    00003   00002   00003      S2
HV      halt          0601    00002   00002   00002      NOP
RD NC   goto angl     0802    00100   00001   00100      S1
TB      ahead         2902    00002   00001   00002      S1
SH      run           3302    00003   00002   00003      S1
```

Figure 6-4 Diagnostic Display 1



```
HEADING: 13.2211 SPEED  : 2.9700   STERN  : 21.900   TIME(ms): 24.68
BUBBLE : 0.0000   X_POS  : 59.993   SAIL   : 21.900
DEPTH  : 97.054   Y_POS  : 5.5873   RUDDER : -36.900  NO_ERR

DEBUG SCREEN 2
Unit Stop  Step   Simu ``Time: Last  Min   Max
CC1  RUN   AUTO   REAL         00042 00021 00105
SM   RUN   AUTO   REAL         00042 00021 00105
DP   RUN   AUTO   REAL         00063 00021 00105
HL   RUN   AUTO   REAL         00063 00021 00105
PR   RUN   AUTO   REAL         00042 00021 00105
DR   RUN   AUTO   REAL         00063 00042 03501
HA   RUN   AUTO   REAL         00042 00021 00084
TR   RUN   AUTO   REAL         00063 00021 00063
BL   RUN   AUTO   REAL         00042 00021 00084
SP   RUN   AUTO   REAL         00042 00021 00084
SL   RUN   AUTO   REAL         00042 00021 00105
LI   RUN   AUTO   REAL         00042 00021 00063
PU   RUN   AUTO   REAL         00042 00021 00063
BV   RUN   AUTO   REAL         00042 00021 00084
HV   RUN   AUTO   REAL         00063 00021 00105
RD   RUN   AUTO   REAL         00042 00042 00084
TB   RUN   AUTO   REAL         00042 00021 00084
SH   RUN   AUTO   REAL         00294 00042 00483
```

Figure 6-5 Diagnostic Display 2

6.2.4. Diagnostic Displays

Displays are used on the PC as well as the SGI workstation for diagnostic analysis. These diagnostic tools are critical in not only getting a system running, but also to track down bugs in a running system.

The PC displays variables of interest, commands being executed, command and status numbers, plan lines being executed, performance metrics, and execution mode. Please see figures 6-4 and 6-5. The variables of interest consist of primarily of control and simulation model values. The importance of the other diagnostics was described above.

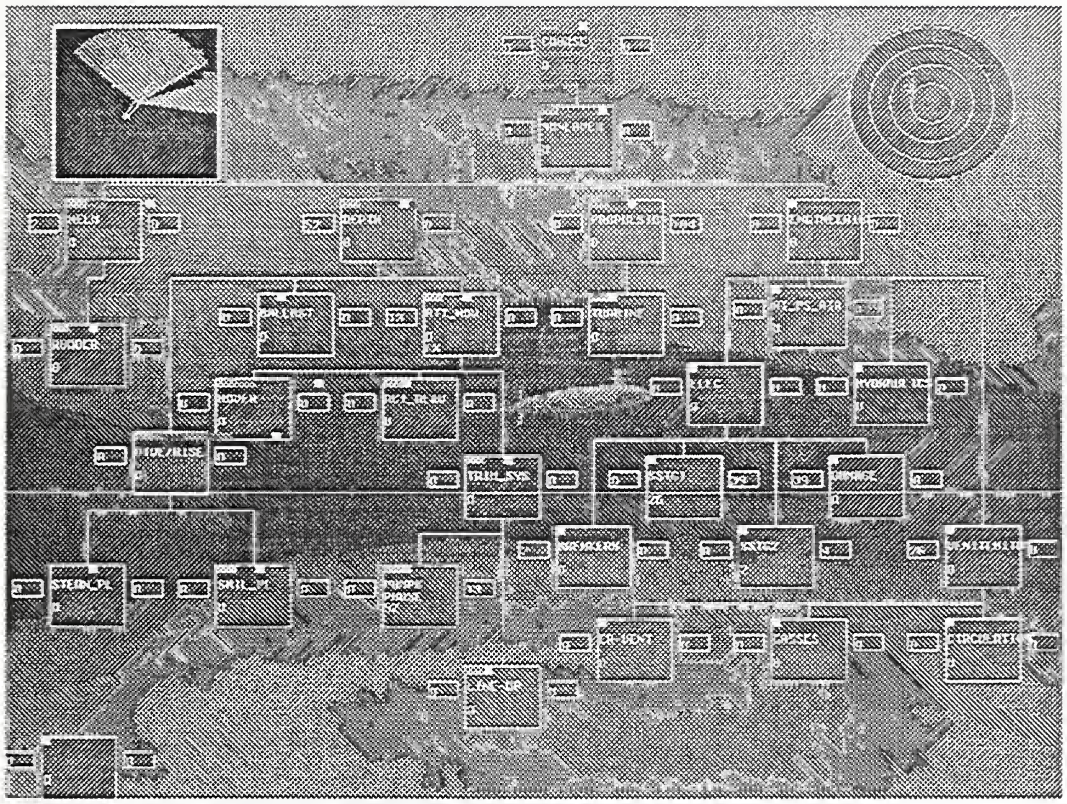


Figure 6-6 A Performance Display on SGI

The SGI workstation enables the designer to create more elaborate pictorial descriptions of diagnostics. The same diagnostics are graphically displayed on the SGI workstation screen in a tree structure. For example, a bar graph is used to display the performance metrics. See figure 6-6.

One of the most important debug recommendations made by the RCS methodology is animation. It conveys an enormous amount of information which can be readily comprehended by a human and is an invaluable debug tool. The animation is discussed in section 6.6.

6.3. Control System World Model and Simulation World Model

According to the Albus Intelligent Machine Systems theory [Al 91], the sensors and actuators act as the interface between an intelligent system and the environment. The intelligent system's perception of the world is described through the use of: state variables, dynamic models, object geometry, etc. At the beginning of a mission, the control system world model may contain some a priori knowledge that may be incomplete, incorrect or too coarse in resolution. For example, the pre-stored map may only indicate major ice distribution areas in the Bering Strait region that might be outdated. A sequence of internal processing, including sensory detection, filtering, comparison, prediction, and fusion, is required to update the system's perception. Discrepancies between the real world and the intelligent machine's perception can be introduced from both the modeling and the processing errors, including:

* There may be environmental changes (unmeasured or disturbance inputs) that are not detected by any sensor employed. The fact that the water salinity is only monitored occasionally during a submarine operation forces the use of an imperfect depth model in the RCS control world model (see sections 3 and 6.7).

* Possible sensory and actuator errors, in the form of noise, biases and failures, may cause distortions in the world models unless the errors are compensated for.

* Some sensory processing or world model prediction models may either contain errors or be greatly simplified. In many cases, the systems engineer will use an order reduced model for high order dynamic systems, and linearized models for nonlinear systems. In Demo #2, one variable ballast control module was unstable due to the existence of an inappropriate inertia model which in turn caused a failure in the depth servo.

Therefore, the control system's world view may differ from the "real environment." In this demonstration, the environment is realized through a separate set of data that the environmental simulators operate on, see section 6.7.

6.4. Shared Memory Model for Communication within a CPU

In this RCS application, the basic principle of maintaining data integrity during communication is a triple buffering mechanism (figure 6-7). At least three copies are kept for any piece of data to be shared. It is defined[9] by the owner module (a controller or a simulator) first. A second copy is defined in the Global Memory (GM). Each of the reader (consumer) modules defines a local copy of the data for itself. Only the owner module can write the data and post it in the GM. The reader modules access the data from the GM. This triple buffering mechanism is simple and replicative. It is used for both input and output and is installed in all the software modules requiring data communication. This mechanism, together with the sequential and cyclic execution model for all the modules, form a simple but rigorous data communication method. The most significant advantages for this execution model include:

* To retain data integrity, user modules always have a local and complete copy of the required data for them to make control decisions. Note that propagation delay may occur in this RCS cyclic and sequential execution model. However, designers can sequence the execution so that the total number of cycles of delay does not destabilize real-time system control. The command/status communication (section 6.4.1) exhibits a rippling effect. If the Dive/Rise controller (figure 4-1) is placed five modules below the Depth controller (figure 6-2) in the Main Program execution

[9]Declaration of data does not necessarily reserve memory. Declarations that reserve memory for the data are called definitions.

schedule, a five cycle delay is required for a depth error to be reported from Dive/Rise to Depth. The designer must make sure that the responding command computed by Depth will come in time for Dive/Rise to compensate for the error. If not, the depth control might be in an unstable state. The world model data communication (section 6.4.2), on the other hand, may require a one cycle delay between the data being posted in the global memory and the access to the data by consumer modules.

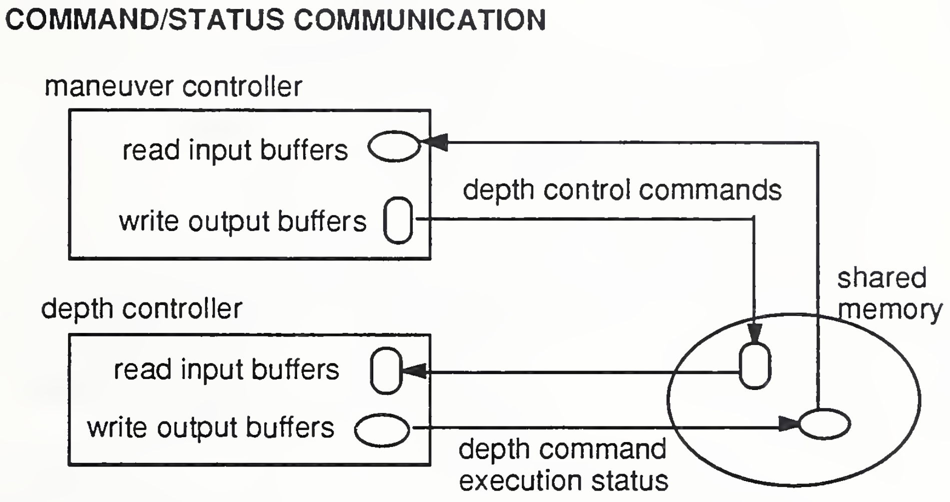


Figure 6-7 Triple buffering data communication

* To enable asynchronous execution (non-blocking). Real-time system control can proceed based on the most recent input data and does not have to pause for the receiving of incoming data. This advantage applies to multiple CPU cases where each can operate on its own cycles.

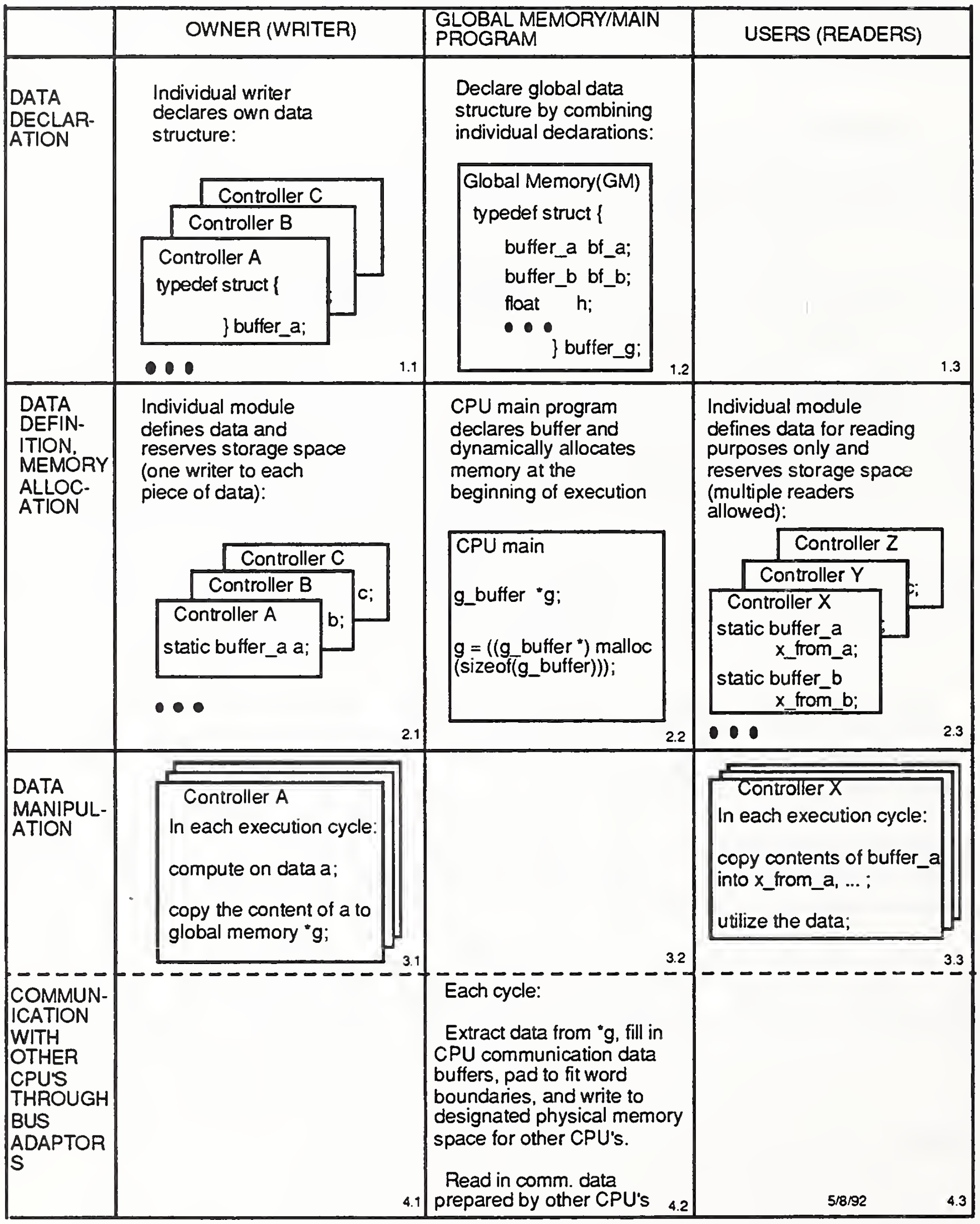
This communication model might also be extended to allow the integration of an RCS application in a heterogeneous environment. An RCS application might communicate using this technique with an Expert System running in parallel.

Figure 6-8 describes the shared memory model for this implementation in detail. The C language syntax is used. There are four stages of communication activities in this implementation (shown as four rows in the figure): data declaration, data definition or memory allocation, data manipulation, and communication with other CPU's. Three columns are shown describing what the owner modules, the global memory, and the consumer modules should do during the four stages. The content of figure 6-8 will be described in the following sections, specifically in sections 6.4.1 and 6.4.4.

6.4.1. Command/Status Communication

An RCS controller is intended to be a stand alone closed-loop controller. Different types of communication are required between a controller and the rest of the system. One type of communication, command/status, occurs only between a controller and its immediate superior and subordinates. RCS retains a rigorous chain of authority. Arbitrary commanding and feedback within the hierarchy are not allowed[10].

TRIPLE BUFFERING DATA COMMUNICATION



	OWNER (WRITER)	GLOBAL MEMORY/MAIN PROGRAM	USERS (READERS)

DATA DECLAR-ATION	Individual writer declares own data structure:
	Controller C
	Controller B
	Controller A
	typedef struct {
	} buffer_a;
	● ● ●
	1.1
		Declare global data structure by combining individual declarations:
		Global Memory(GM)
		typedef struct {
		buffer_a bf_a;
		buffer_b bf_b;
		float h;
		● ● ●
		} buffer_g;
		1.2
			1.3

DATA DEFIN-ITION, MEMORY ALLOC-ATION	Individual module defines data and reserves storage space (one writer to each piece of data): Controller C c; Controller B b; Controller A static buffer_a a; ● ● ● 2.1	CPU main program declares buffer and dynamically allocates memory at the beginning of execution CPU main g_buffer *g; g = ((g_buffer *) malloc (sizeof(g_buffer))); 2.2	Individual module defines data for reading purposes only and reserves storage space (multiple readers allowed): Controller Z ; Controller Y Controller X static buffer_a x_from_a; static buffer_b x_from_b; ● ● ● 2.3
DATA MANIPUL-ATION	Controller A In each execution cycle: compute on data a; copy the content of a to global memory *g; 3.1	3.2	Controller X In each execution cycle: copy contents of buffer_a into x_from_a, ... ; utilize the data; 3.3
COMMUN-ICATION WITH OTHER CPU'S THROUGH BUS ADAPTOR S	4.1	Each cycle: Extract data from *g, fill in CPU communication data buffers, pad to fit word boundaries, and write to designated physical memory space for other CPU's. Read in comm. data prepared by other CPU's 4.2	5/8/92 4.3

Figure 6-8 Triple buffering communication -- from the implementation point of view

[10]World model data can be shared by any controllers without the same constraint.

A controller communicates with its superior to:

* receive commands
* report its status
* report error messages that require attention from higher authorities.

In a similar vein, this controller also communicates with its subordinates to:

* assign commands
* receive status
* guide error recovery processing, based on the error message received.

As seen in section 6.2, a generic controller template declares a data structure type that is primarily oriented toward communication with its superior. Such a data structure includes:

* the commands the controller may receive, of the enumerated type [Ha 91][11]
* command serial number (see section 6.2.3.5 for its utility)
* the status a controller reports to its superior, of the enumerated type
* command serial number echo attached to status buffer
* performance data[12]
* the sensor data the module receives
* the actuator commands the module sends out.

This data structure declaration activity is indicated in block 1.1 in figure 6-8. Controller A owns a data structure of the type "buffer_a" containing all the information controller A needs to communicate with its superior.

The next step is data definition, shown in block 2.1. A variable "a" of this "buffer_a" type is defined in the heading area of the software templates for a generic controller. This variable is defined as a static type so that its values can be preserved during the entrance and exit of this module in cyclic execution. Appendix D shows an example.

All the individually declared data buffers ("buffer_a," "buffer_b," ...) are combined and declared as a large data structure in the global memory (GM, block 1.2 of figure 6-8). All the reader modules (described below) would access the GM for the required data and do not access the data that the writer module internally keeps (block 3.3 of the same figure).

The heading of the generic controller also defines variables of those types declared by its subordinates (block 2.3). In this way the controller can access the data (block 3.3) to make decisions.

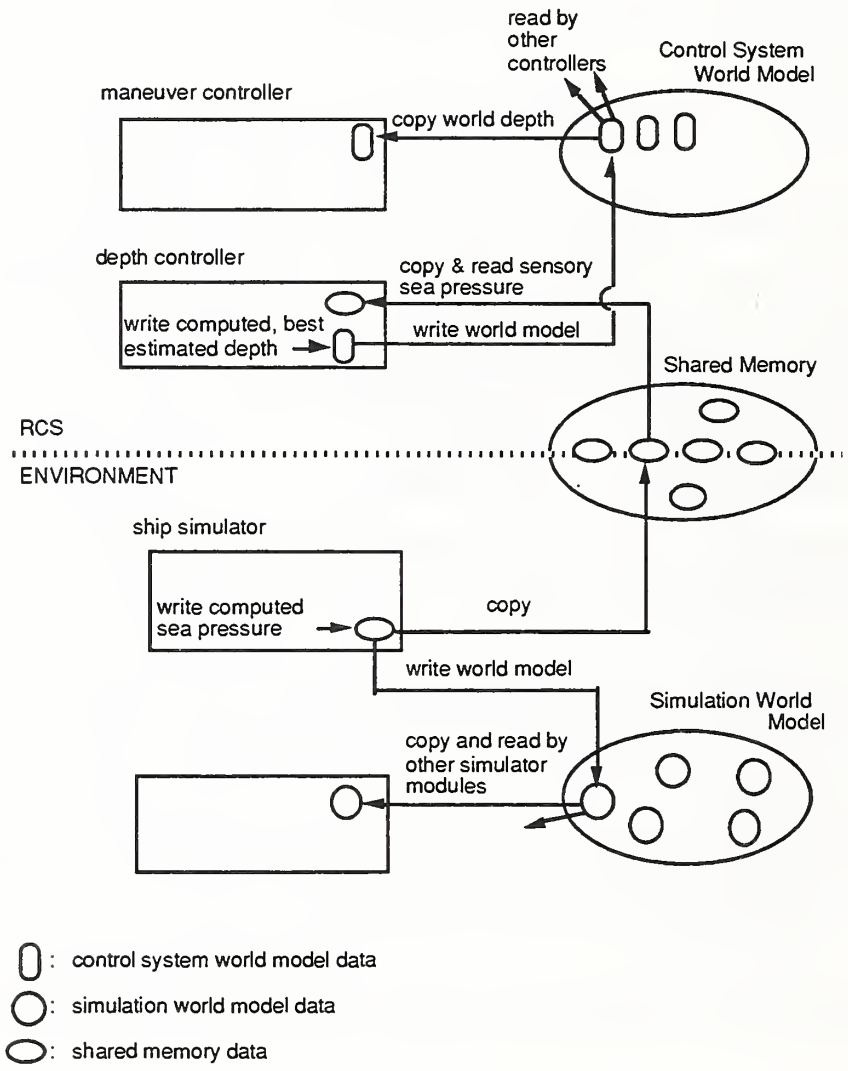
At the beginning of the main program execution, the GM is dynamically allocated (block 2.2 of figure 6-8). The data is filled in during cyclic execution by the owner controllers at each controller's post-processing stage (block 3.1). Prior to the post-processing stage, the owner controllers compute data that they own. These data are defined to be not accessible to the outside of the controllers.

[11]The task names listed in the task tree (figure 4-2) describe all the possible commands each controller can receive or send. Each command is assigned a unique identification number. In this implementation, these numbers are used to facilitate program debugging and keyboard input. For example, the number 2204 needs to be typed in to allow the Helm controller to execute the command ICE_MANEUVER when using the source code debugger.

[12]Currently only the execution time for the module is included in this category.

6.4.2. World Model Data Communication

Besides the command/status communication, the controllers also access the world model for required processing data. The same triple buffering mechanism, shown in figure 6-9, is used, which is consistent with figure 6-8.



read by other controllers
Control System World Model
maneuver controller
copy world depth
depth controller
copy & read sensory sea pressure
write computed, best estimated depth
write world model
Shared Memory
RCS
ENVIRONMENT
ship simulator
write computed sea pressure
copy
write world model
Simulation World Model
copy and read by other simulator modules
: control system world model data
: simulation world model data
: shared memory data

Figure 6-9 World model data communication

The world model includes a state space representation of the perceived world. Two types of state variables may exist:

* sensory oriented data, such as the ship bubble angle,
* internally derived or human input types of data, such as the ship maneuvering modes (Stealth, etc.).

Any data that are required by multiple controllers are defined in the world model and are declared in the GM. The controller that most crucially requires the data "owns" it and is responsible for writing it. In the case of sensory data, the ownership of the data means that this controller possesses a sensor to read the sensory values in. This controller declares and defines a corresponding world state variable (corresponding to blocks 1.1 and 2.1 of figure 6-8) for that particular sensory data. Necessary sensory processing would be performed to derive best estimated values for the defined world state variable [Al 88, Hu 91]. In the GM area a world model data structure is declared (block 1.2 of the same figure) and memory allocated (block 2.2). The owner controllers post the estimates into the GM (block 3.1). All the other concerned controllers declare local copies of the variables, copy the values in, and utilize them for decision making (block 3.3).

6.4.3. Communication between Human Operators and the Control System

As described in section 6.1, a human can interact with the control system to either perform interactive control or to alter the state of the environment. In both cases the same triple buffering model is used to handle the I/O process. Essentially each controller defines its interface data structure and three copies are maintained: within the controller itself, the GM, and the human interface handling module. In this way the control process and the I/O process are non-blocking. The cyclic execution of the real-time control system does not have to pause to wait for the I/O. More detailed and specific discussion will be given in section 6.5.

6.4.4. A Special Case of the Shared Memory Model -- Communication with Other CPU's through a Bus Adapter

The same triple buffering concept also applies to the case of communication with other CPU's (although this is not the only method to perform such communication). As described in section 5, the Bit3 memory mapped bus-to-bus adapters are used to connect the PC bus and the SGI VME bus. A 1M byte dual-ported RAM has been installed on the adapters to physically provide memory space for the shipment of data between these two CPU's[13].

As shown in figure 6-10, the process of writing data from the source computer to the target computer begins with a declaration of the communication data structure at both computers. All the data the target computer requires are included. The two computers use different formats to represent floating point data. Therefore, shared data are declared as long integers or characters to simplify the format conversion problem while retaining data precision. All the data declared in the communication buffer should be "padded" to the 32 bit long word boundaries since the buffer structure may need to be modified (discussed later in this section) on the receiving end and mis-interpretation of data can occur.

The communication process is executed every cycle, and data values are copied from the world models to the aforementioned buffer.

The last step of the writing out data process involves moving the data from the local PC memory to the extended memory. The pointer to the memory space that an outgoing buffer resides at must be represented in terms of the segment[14] and offset[15] numbers [Ha 91]. The actual data moving process is done via the execution of the proper 8086-processor-

---

[13]The PC has 8M of internal RAM, therefore, this external memory is addressed at location (0X800000).

[14]One segment occupies a 64K byte space. Segment numbers can be represented by the higher 16 bits of a "far" type pointer.

[15]Offsets are the relative positions within segments, and can be represented by the lower 16 bits of a pointer.

family interrupt which requires the information including the word count of the buffer, the aforementioned memory location, etc. (Refer to the "movphy" routine in the code.)

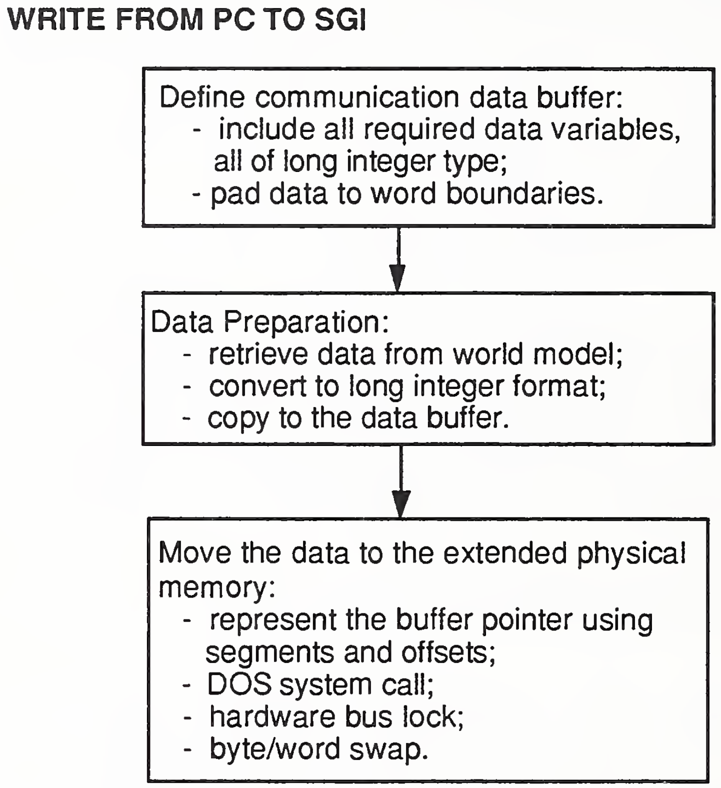

WRITE FROM PC TO SGI
Define communication data buffer:
- include all required data variables, all of long integer type;
- pad data to word boundaries.
Data Preparation:
- retrieve data from world model;
- convert to long integer format;
- copy to the data buffer.
Move the data to the extended physical memory:
- represent the buffer pointer using segments and offsets;
- DOS system call;
- hardware bus lock;
- byte/word swap.

Figure 6-10 Communication with other CPU's Using the Bit3 Adapters

Data integrity is achieved through hardware bus arbitration [Bi 90]. On the receiving end (the SGI workstation), the data buffer needs to be swapped at the word level and followed by another swap at the byte level. This is required due to the different byte ordering on the PC and the SGI, as shown in figure 6-11.

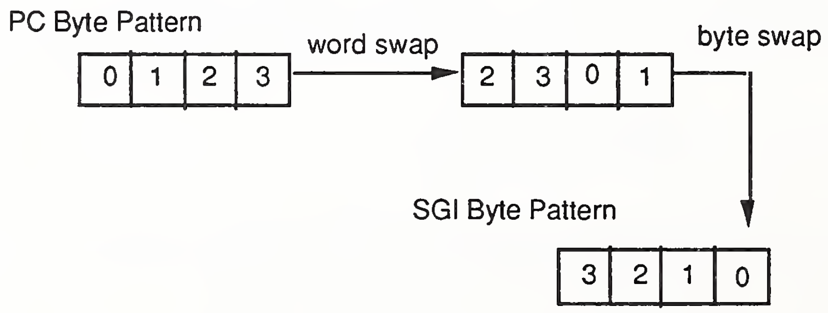


Figure 6-11 Swap of data at SGI upon receiving communication from a PC

Writing from the SGI to the PC can be done similarly. The PC defines an incoming data buffer with proper padding. The data is moved from the SGI local memory to the extended memory before being read into the PC local buffer. Its contents are then copied to the GM.

6.5. Multiple Mode Control -- The Automatic Mode and the Interactive Mode Structures

The RCS architecture [Al 91, Hu 91] specifies that human operators can interact with the control hierarchy at any module at any level to any extent that has been built in. The following describes the different types of control operations and the implementation techniques:

a. Control Modes: Two operation modes have been implemented. In automatic mode, human interaction is not allowed during system execution. In interactive mode, an operator must enter input after errors and the compensation options have been displayed to him. The objectives are to illustrate that authorization is required for a human to intervene with system control and that certain system capabilities can be enabled or disabled only in some control modes. For example, a possible implementation is that, when maneuvering through areas where ice is densely distributed, only in interactive mode may a human operator elect to disable the ice avoidance algorithm and to command the control surfaces directly.

   The human operators can have various degrees of involvement with the system control. The mode in which no human interaction is allowed is called automatic mode. From there on, human interaction types can range from allowing an operator to only respond to error flags, to actively change system parameters or commands, to interject complex human reasoning results to complement preprogrammed machine intelligence, to take over the total control of some subsystems, up to allowing him to take over the control of the entire system. In addition, most of these interaction types can be enabled/disabled either on any controller individually or on any sub-hierarchies. Different passwords could be used for enabling any of these human interaction types. This description suggests that there could be as many control modes as required.

   The interactive mode implementation is rather rudimentary in this demonstration. The user has to select the mode to operate the ship maneuvering system interactively. A re-selection of the automatic mode would prevent him from further interaction with the system control. The current automatic mode may therefore be viewed as a sub-function of the interactive mode. However, the objective here is to create a generic mechanism and software structure to facilitate future full implementation of the multiple RCS control modes.

b. Human Interface: Each controller in the RCS hierarchy can employ an interface module to handle its human I/O. The human interface module and the controllers use the shared memory mechanism to make the communication process non-blocking to the control execution.

c. Automatic Mode Activities: The automatic mode hierarchy, shown partially in figures 6-12 and 6-13, is identical to the one shown in figure 4-1 (section 4.1). A Ship Maneuver console shown on the top of these figures provides the hardware for human computer interface. When a submarine operator turns the interactive mode off, a HALT command will be passed down through the interactive mode hierarchy to inactivate all the modules. The system control will be performed only by the automatic mode hierarchy. The submarine automatically transits through Bering Strait via the plans described in section 4 in the automatic mode. The bottom of figure 6-12 shows that the Dive/Rise module detected and reported an error in the ship depth control. The plan currently being executed in the Depth controller (shown in figure 6-12 as a state table) responds to the error message and selects an error compensation command for the Dive/Rise controller. The depth control resumes without human interaction. The automatic mode might be used when operating the submarine in the wide-open sea.

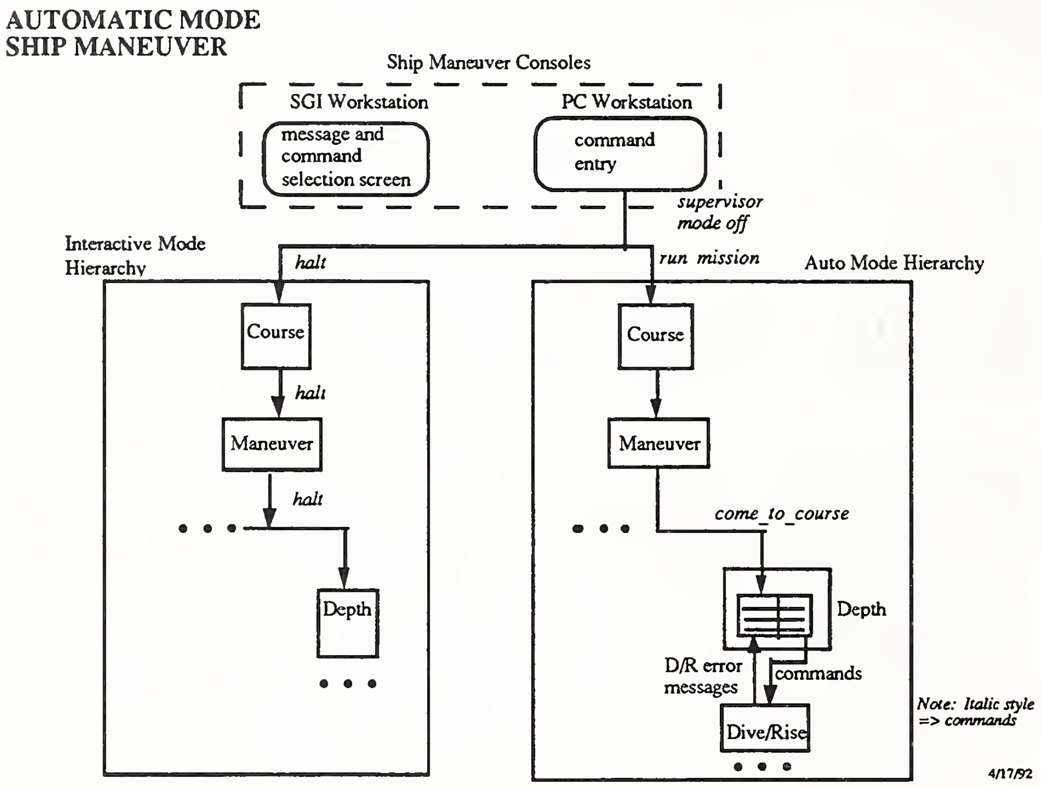


Figure 6-12 Automatic control mode ship maneuvering

d. Interactive Mode Activities: When an operator selects the interactive mode, shown in figure 6-13, a RUN command will activate the entire interactive control hierarchy. The system executes the RUN_MISSION command automatically until a severe depth error occurs. In this situation an error message is sent to the Depth control interface module, as shown at the bottom of figure 6-13. This interface module is responsible for sending and encoding the message to the SGI workstation of the Ship Maneuver console. This module also receives and decodes the human response. The same triple buffering mechanism as described in section 6.4 is used. The command that the operator selects is posted at the GM. The Depth controller copies the information in and executes accordingly. The benefits of this non-blocking communication are to retain data integrity and to allow non-disrupted real-time system control.

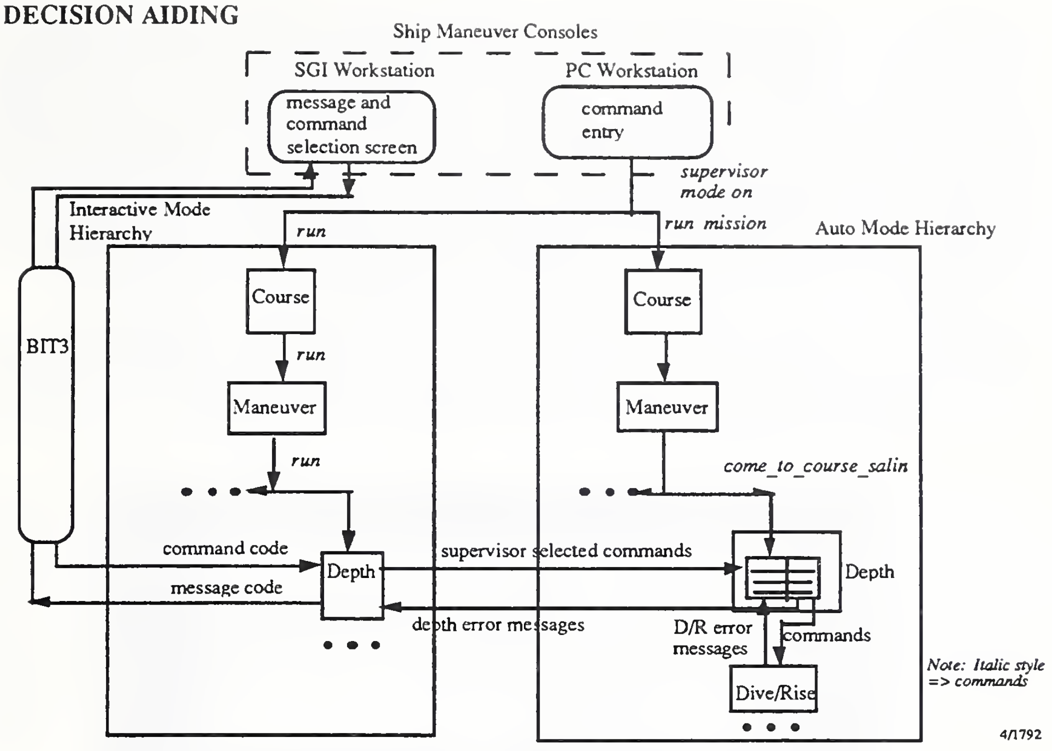


Figure 6-13

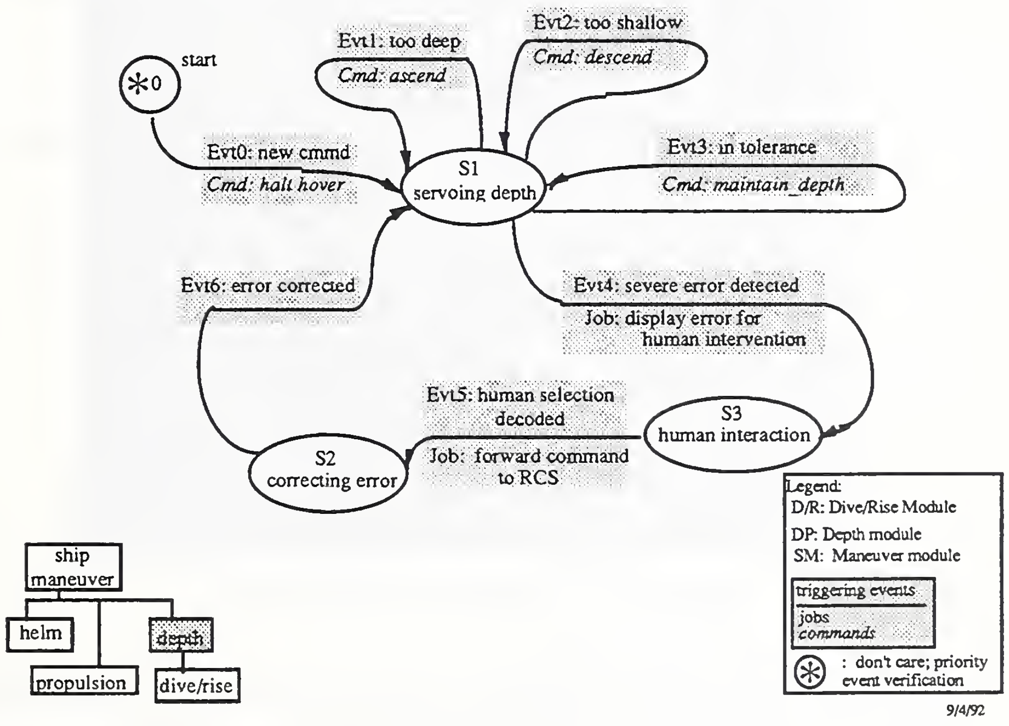
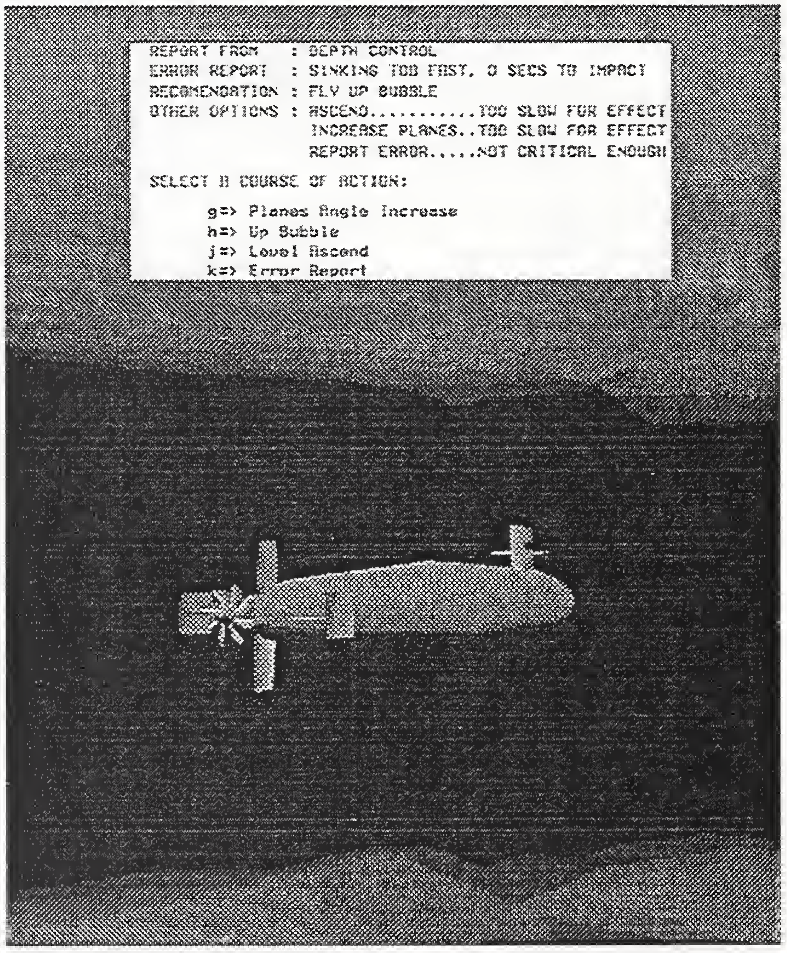


Figure 6-14 The interactive mode depth control plan

A state diagram describing the interactive mode depth control activities is shown in figure 6-14. The normal operation, shown around the state (S1), is the same between this figure and figure 4-8, the automatic mode depth control plan. What is different here is that the error messages are forwarded to and displayed at the Ship Maneuver Console instead. In figure 6-14 the state (S3) describes that the human is attempting to interject his input and such input, once received, would be decoded as a depth control task command. Figure 6-15 demonstrates such operator interaction activities. A message for a severe depth error is displayed. An assessment of the situation, the options, and the recommendation are provided for the operator.

Note that, the options that the human operators can select to compensate for the errors can not go beyond the scope of the tasks defined in the task tree (section 4.1). The operator may select only the pre-defined tasks, as shown in figure 6-15. However, he can, in addition,



```
REPORT FROM    : DEPTH CONTROL
ERROR REPORT   : SINKING TOO FAST, 0 SECS TO IMPACT
RECOMENDATION  : FLY UP BUBBLE
OTHER OPTIONS  : ASCEND..........TOO SLOW FOR EFFECT
                  INCREASE PLANES..TOO SLOW FOR EFFECT
                 REPORT ERROR.....NOT CRITICAL ENOUGH

SELECT A COURSE OF ACTION:

      g=> Planes Angle Increase
      h=> Up Bubble
      j=> Level Ascend
      k=> Error Report
```

Figure 6-15 An Operator Interaction Screen

- judge the severity of the situation,
- make trade-off studies between factors such as ship obstacle avoidance safety, sea bottom safety, stealth requirements, etc., and
- adjust the parameter values for the selected task commands in order to expedite system response.

One example is that the operator can increase the propulsion speed to try to compensate for severe depth dropping error with the understanding that the higher speed increases the possibility for the ship to either hit ice keels or be detected.

e. Hierarchical Considerations for the Future: In this demonstration, only the Depth controller has the human interaction capability which all the other controllers can also have. However, conflicts may arise when multiple human operators are interacting simultaneously with their corresponding controllers. A human Depth control officer may select one task, his subordinate, the human Dive/Rise control officer, may select a totally unrelated task. Some prioritization schemes may be used to solve this problem. This issue will be carefully examined in the next phase of this project.

6.6. Simulator Structure

The simulator hierarchy is shown in figure 6-16. This figure is drawn to fit directly in the bottom of figure 4-1, the RCS hierarchy. The RCS actuator controllers send commands to their respective simulators. The simulators compute the actuator movements accordingly and send the computed values back to the original controllers via the simulated sensors. The same values also are used by the ship simulator for the computation of the ship dynamics[16]. The simulated ship state is fed back to the required higher level RCS controllers via simulated sensors. More detailed discussion is presented in the following sections.

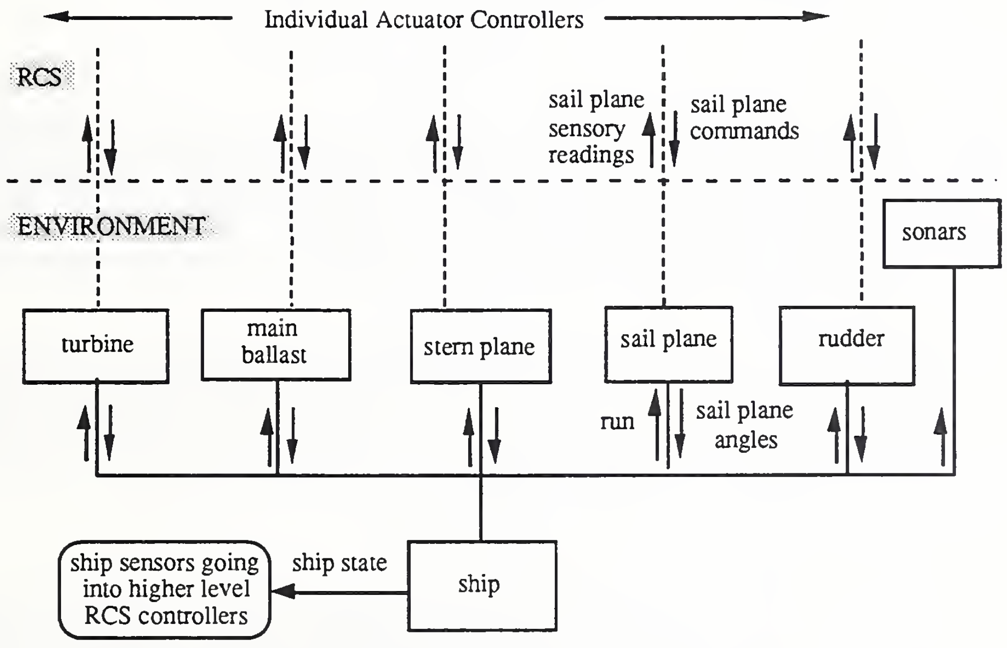


Figure 6-16 The simulation hierarchy

[16]Actually the values are copied to the simulation world model, which allows the ship simulator to copy the data, as shown in figure 6.9.

The sonar simulator in this figure is an exception. It does not send any data to the ship simulator. Rather, the sonar data are sent to the Helm control sensory processing routines (CMAC). The submarine is equipped with 14 forward looking sonars. Sonar data are simulated by extending 14 sets of five vectors (figure 6-17), each set simulating a sonar pinging in its fixed direction relative to the ship's center line, from the ship to the full sonar operating range (approximately 900 meters). Intercepts with the simulated ice keels (see section 6.6.3 and [Hu 92]) are collected as the sonar detections. As mentioned earlier, ice keels mean the ridges and protrusions of ice built up underwater (as a result of pack ice collisions).

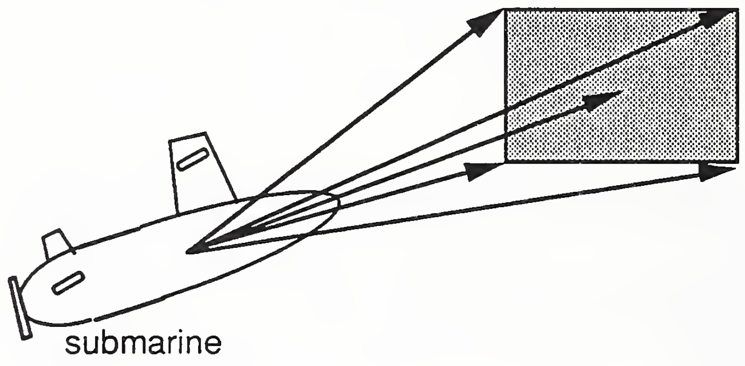


Figure 6-17 Simulation of a forward looking sonar

6.6.1. Actuators

The template for the actuator simulators is similar to the one used for the generic controllers, as shown in figure 6-18. Essentially the simulator copies in the command stored in the GM and the current actuator value stored in the simulation world model. The simulator then computes a new actuator value according to the incoming command. At the post-processing phase, the simulator copies the values back to the world model as well as the GM interface buffers.

These simulated actuators generally do not perform any closed loop control function. The desired actuator positions are servoed by the lowest level RCS controllers. The actuators normally receive and respond to commands such as ON, OFF, INCREASE_VOLT, DECREASE_VOLT, ZERO_VOLT, etc. Dynamic models typically exist in the simulators to translate those commands to actuator positions.

6.6.2. Physical System

The template for the ship system simulator, shown in figure 6-19, assumes the format of a generic controller. This software module copies the incoming commands. Three commands have been implemented in this demonstration: INITIALIZATION, HALT, AND EXECUTION. All the required data then are copied in including the current ship state and the computed actuator positions.

The decision processing essentially involves a sequential execution of all the actuator simulators and a computation of the ship dynamics. Currently the ship dynamics includes the modeling of the position, speed, depth, bubble angle, and heading for the ship. This

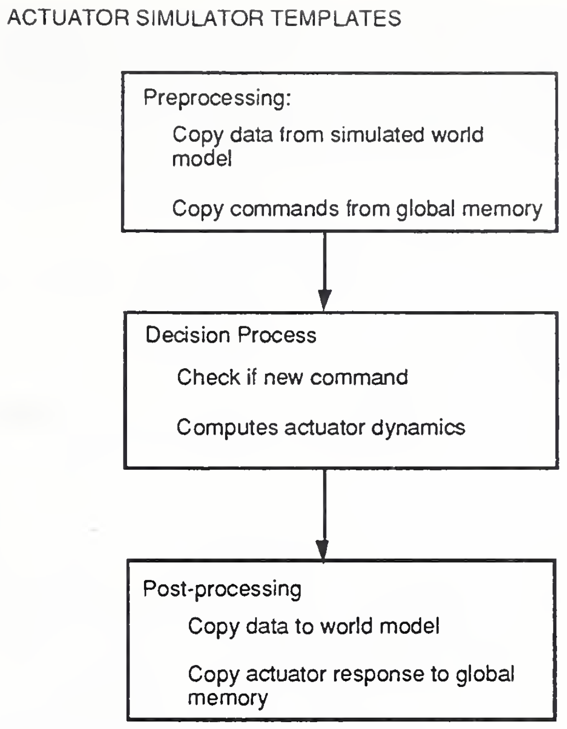

ACTUATOR SIMULATOR TEMPLATES
Preprocessing:
Copy data from simulated world model
Copy commands from global memory
Decision Process
Check if new command
Computes actuator dynamics
Post-processing
Copy data to world model
Copy actuator response to global memory

Figure 6-18 The actuator simulator templates

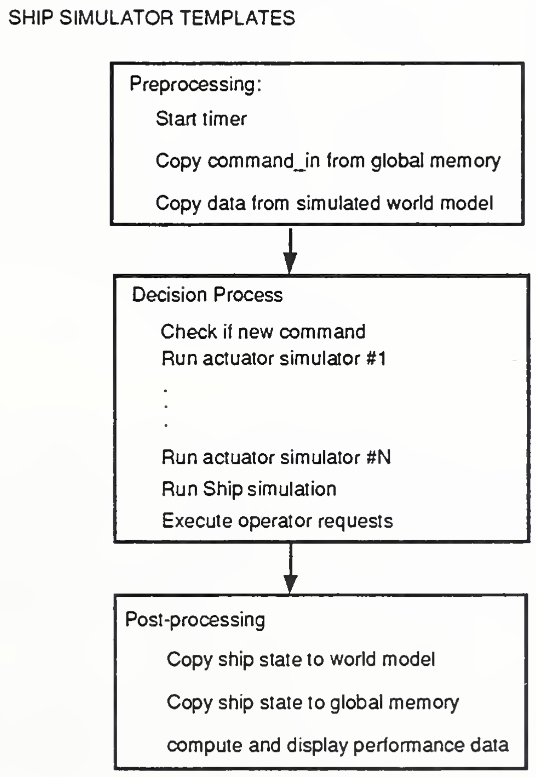

SHIP SIMULATOR TEMPLATES
Preprocessing:
Start timer
Copy command_in from global memory
Copy data from simulated world model
Decision Process
Check if new command
Run actuator simulator #1
.
.
.
Run actuator simulator #N
Run Ship simulation
Execute operator requests
Post-processing
Copy ship state to world model
Copy ship state to global memory
compute and display performance data

Figure 6-19 The ship system simulator templates

simulated ship depth can be regarded as the actual depth. On the other hand, the control system may employ a depth model to compute the ship depth based on the sea pressure sensory values. This depth may be referred to as the perceived depth. The two depth values may differ in some situations (see section 3, when the submarine runs into some fresh water pockets).

6.6.3. Environmental

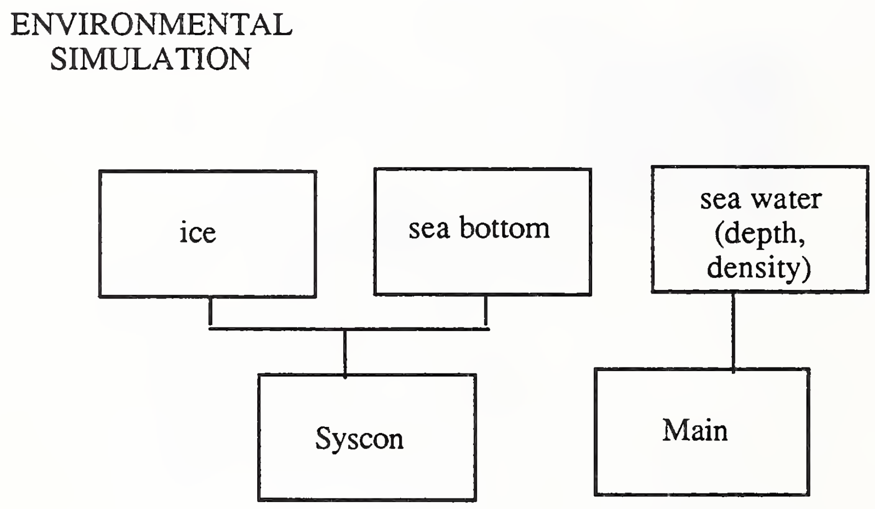


Figure 6-20 The environmental simulation software structure

The environmental simulation software structure is shown in figure 6-20. The ice keels are fractally generated (see section 6.8) prior to the system cyclic execution. The sea bottom is generated similarly but the bottom looking sonar is not currently integrated. These modules are controlled by the main routine on the SGI, SYSCON. Fresh water run-off pockets in the sea water are simulated through the computation of the true ship depth and the water density. Unlike the depth simulation in the ship system, in this environmental simulation the depth model is sensitive to water density changes.

6.7. Operator Interaction with the Simulators

As shown in the lower right quadrant of figure 6-1, the software architecture specifies that any software unit, be it controller or simulator, be subject to human interaction. Operators can intervene in the simulation state space. These simulation operators inject changes to the environment as well as to the physical ship system. For example, they may change the ship's bubble angle. These changes are expected to be detected by the RCS sensor systems. The ship maneuvering RCS is then expected to respond to the changes and update its world model. An operator interaction hierarchy can be implemented, which would be parallel to the simulation hierarchy, as seen in figure 6-16. This simulation interaction hierarchy is conceptually distinguished from the interactive control hierarchy (section 6.5). If the RCS sensory system is not able to detect and respond to the environmental changes effectively, the RCS design needs to be modified. Each simulator would have a human interface module. They can be executed cyclically along with the controller and simulator modules in the main program.

Currently an operator can inject changes either from the graphic user interface (GUI) implemented on the SGI IRIS workstation or through the predefined keyboard command strings. The GUI includes slider bar control for all the control surfaces. In other words, one can click the mouse on the slider bars to steer the control surfaces. The numerical values for the affected control surfaces will show on the GUI screen, while the actual data are sent, using the same triple buffering mechanism, to the simulation modules. The simulated submarine will then respond to the GUI requests.

The GUI also includes a switch for changing the sea water density. One can click on the switch and change the density for an amount up to 10 percent. The submarine would rise or sink accordingly.

All these intervention capabilities are also implemented in the code as keyboard command strings. One can type in, from the PC workstation (see figure 6-12), commands such as "CHANGE_DENSITY 0.95."

6.8. Animation

Animation is a very powerful design tool used by the RCS methodology because it enables the user to visualize the resultant actions of controllers. In the submarine demonstration, animation was used extensively for design and debug of the control algorithms. Animating enhances the human understandability of a process; i.e., a human can comprehend many more variables pictorially than in other formats, e.g., tabular. RCS focuses on machines that do work and the human understandability of complex control systems; therefore, animation is a logical step. It enables one to debug the algorithms that are used to control the machine and may avoid costly mistakes that occur when transferring software from simulated to real systems, sometimes referred to as off-line programming (OLP). For a detailed analysis of OLP, see Tarnoff [Ta 92]. A brief description of the animation is provided here. For a detailed analysis of the submarine animation, please see [Hu 92].

6.8.1. Software Structure

Although the animation is not a controller itself, the same principles of RCS software modularity and cyclic execution are used. The animation software executes at a slower rate than the controller software on the PC, but this is not of grave consequence. Humans can process information at a much slower rate and in far less detail than a computer can; therefore, the animation update rate is fast enough for a person to comprehend the scene.

6.8.2. Submarine Model

Care was taken to maintain scale of dimensions for the submarine model and its size. For example, the relative lengths of the height, width, and length of the submarine are true. Please see figure 6-21. Much heuristic information was provided by our domain experts. For example, the time it takes for the submarine to make a ninety degree turn at a certain speed. These parameters were used to make the animation look and feel realistic. The speed of the submarine was increased to allow for short demonstrations.

Modeling was used to determine the ship's depth, speed, and orientation. Although modeling was kept as simple as possible, most of the important parameters were included to make the simulation realistic. One of the aims of future work is to integrate higher fidelity simulation with the current control structure.

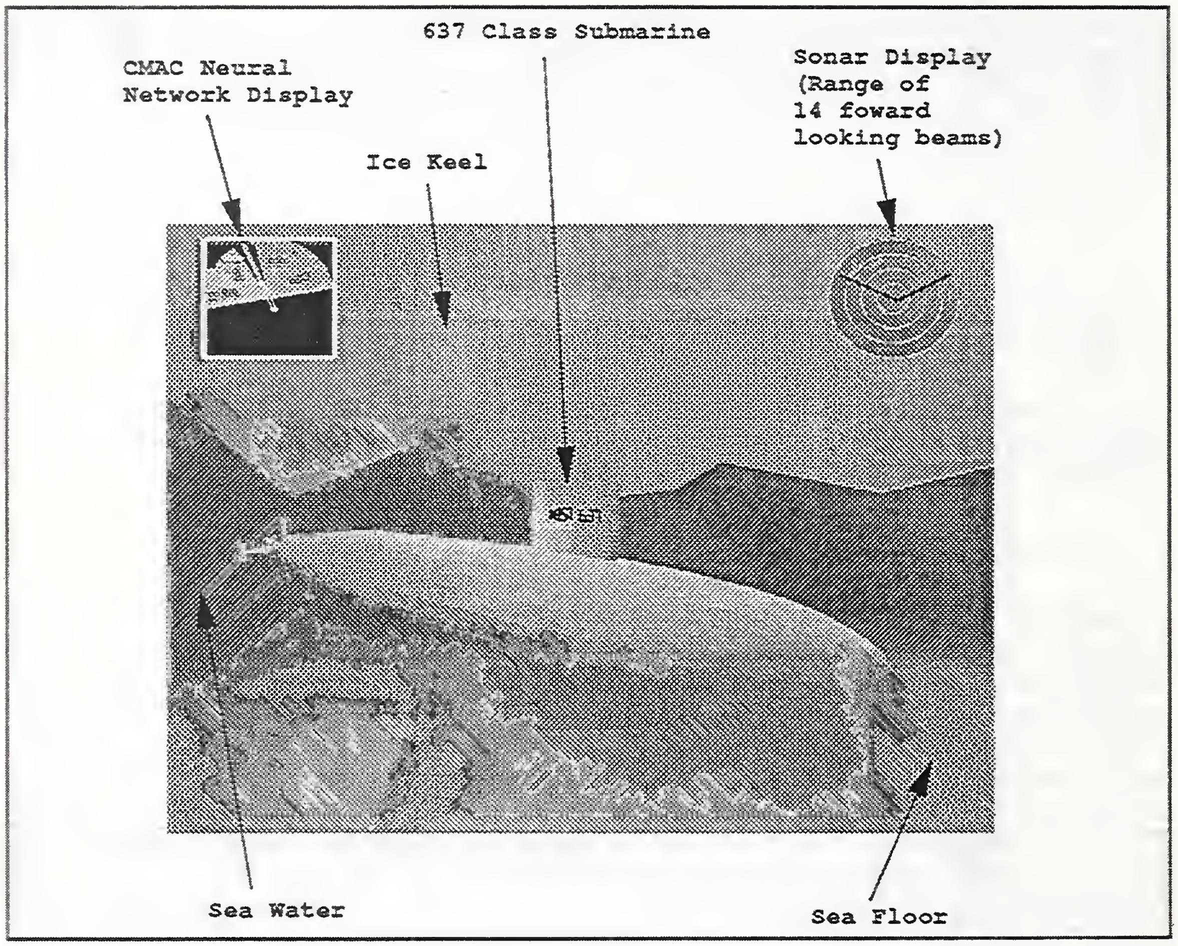

637 Class Submarine
CMAC Neural Network Display
Sonar Display (Range of 14 foward looking beams)
Ice Keel
Sea Water
Sea Floor

Figure 6.21 Animation of a Submarine Transiting Undersea

6.8.3. Ice keel and Sea bottom

The ice keel and sea bottom profiles, shown in figure 6-21, were generated using fractals [La 91], which allowed a complex structure of indefinite size and extent without the need for long and complex mappings.

6.8.4. Current Sonar Display

Ice detection sonar on a 637 class submarine consists of fourteen forward looking beams, one downward looking, and one upward looking. Under stealth operation, these beams are short range (high frequency) to avoid detection. Current simulation sonar beams are calculated and displayed on the SGI workstation. See the upper right corner of figure 6-21. These values are updated every four seconds as in an actual submarine. Because the submarine can only see a short distance ahead with any detail, it is necessary to generate a map of the ice profile.

6.8.5. Estimated Ice Map and Ice Avoidance Recommendations

A two dimensional map of ice encountered is stored by a Cerebellar Model Articulation Controller (CMAC) neural network. The CMAC neural net provides an efficient means for storing sparse data. The network is trained by current sonar data. A local map, of the ice formations is queried and displayed. The local map is provided as an input for heading control and ice avoidance maneuvering. Please see the upper left corner of figure 6-21. For more information on CMAC, see [Al 75]. For more information on how CMAC was implemented in this work, see [Hu 92].

6.8.6. Environmental Intervention Slider Bar Control Input

A graphical user interface (GUI) is used for environmental intervention and the ability to change various parameters in the system. Please see figure 6-22. One example of this is the salinity perturbations mentioned previously. The GUI was designed initially as button and slider bar controls. These controls are to be enhanced in future work. The GUI allows one to “fly” the submarine as well as providing a means for a trainer to inject faults. The GUI local variables reside on the SGI, and those variables are copied to GM via the Bit3 bus adapter card. The GUI screen not only provides a means for interaction, but also another method of tracking variables.

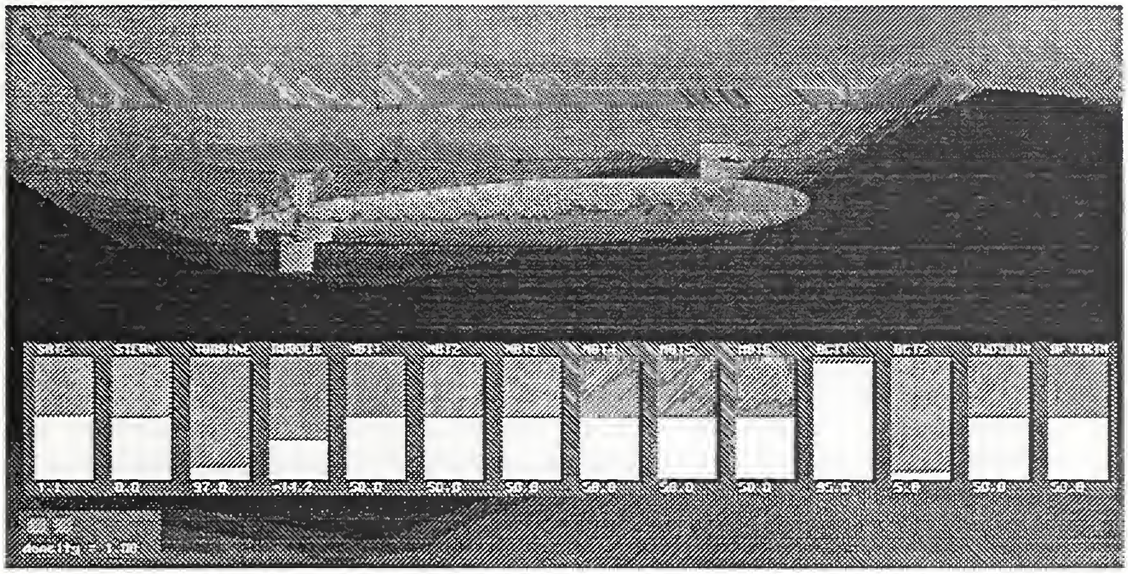


Figure 6-22 Slider Bar Control Input

7. FUTURE DEMONSTRATION DIRECTIONS

The demonstration model presented in this paper will be expanded and enhanced with a focus on RCS methodology solutions for submarine automation. Specifically for demo #4, scheduled for the fall of 1992, we plan on the following enhancements:

* Submarine automation RCS with expanded functionality in planning, decision aiding and multi-mode operation;
* Expanded human interface for information display and operator input;
* Improved C templates;

We also plan to develop a scenario for demonstrating the RCS methodology applied to ship systems automation.

Achieving these goals will necessitate formalizing the RCS methodology, creating portable code, and enhancing the software structure. They will also demonstrate RCS software robustness, verifiability, extensibility, and efficiency.

Planned research for the longer term in the area of ship systems consists of:

* Preliminary planning, decision aiding, and multi-mode operations including:
    ** Detect and locate failures and anomalies
    ** Reconfigure machinery line-ups in real-time
* Expanded human interface for operator input, simulation, control, and use as training aid
* Animation and simulation.

These goals will highlight the versatility provided by the RCS design, provide generic and reusable software, and demonstrate high level control of a complex system. To enhance the RCS design methodology, a Computer Aided Software Engineering tool is being developed by RSD.

8. SUMMARY

The technical objectives established for Demo #3, as described in section 1, were accomplished. The specific achievements include:

* Implemented a submarine automation model.
* Established a first version C language based generic RCS development environment containing:
    - a set of generic templates for future RCS developments;
    - layout of a generic, modularized, and extensible software structure featuring multiple parallel hierarchies: control, simulation, animation, control interaction, and environmental interaction;
    - a set of conceptually separated and distributed world models in the software structure serving multiple hierarchies.
* Demonstrated the extensibility of the RCS architecture by adding new functionalities (including the automatic control of the salinity problem) to the existing RCS application, Demo #2.
* Demonstrated multiple control modes in RCS execution, namely, the automatic mode and the interactive mode. A rudimentary human decision aiding capability was implemented.
* Devised a generic operator interaction handling mechanism.

Besides reporting accomplishments, this paper also serves a more important role. As stated earlier, part of our mission at NIST is technology transfer. This report fits into this mission by illustrating the RCS development procedure and presenting a specific application example.

The computer source code for this implementation can be made available upon request to the authors.

ACKNOWLEDGMENTS

The authors extend their appreciation to Ms. M.L. Fitzgerald, Mr. Clyde Findley, Mr. Philip Feldman, Mr. Nat Frampton, and Mr. Mark Routson of the Advanced Technology and Research Corporation for their participation in various aspects of this project.

REFERENCES

[Al 92] Albus, J.S., Juberts, M., Szabo, S., "RCS: A Reference Model Architecture for Intelligent Vehicle and Highway Systems," ISATA 92, Florence, Italy, June 1992.

[Al 91] Albus, J.S., "A Theory of Intelligent Systems," CONTROL AND DYNAMIC SYSTEMS, ADVANCES IN THEORY AND APPLICATIONS book series chapter, Volume 46, Academic Press, 1991.

[Al 89-1] Albus, J.S., McCain, H.G., and Lumia, R., "NASA/NBS Standard Reference Model for Telerobot Control System Architecture (NASREM)," NBS Technical Note 1235, National Bureau of Standards, U. S. Department of Commerce, April, 1989.

[Al 89-2] Albus, J., Quintero, R., Huang, H., and Roche, M., "Mining Automation Real-Time Control System Architecture Standard Reference Model (MASREM)", NIST Technical Note 1261 Volume 1, National Institute of Standards and Technology, U. S. Department of Commerce, May 1989.

[Al 88] Albus, J.S., "System Description and Design Architecture for Multiple Autonomous Undersea Vehicles", NIST Technical Note 1251, National Institute of Standards and Technology, U. S. Department of Commerce, September 1988.

[Al 82] Albus, J.S., McLean, C., Barbera, A., and Fitzgerald, M., "An Architecture for Real-Time Sensory-interactive Control of Robots in a Manufacturing Environment," 4th IFAC/IFIP Symposium on Information Control Problems in a Manufacturing Technology, Gaithersburg, MD, Oct. 1982.

[Al 75] Albus, J.S., "Data Storage in the Cerebellar Model Articulation Controller (CMAC)," Journal of Dynamic Systems, Measurements, and Control, September 1975.

[Ba 84] Barbera, A. J., et al., "RCS: The NBS Real-Time Control Systems," Robotics 8 Conference and Exposition, Detroit, MI, June 1984.

[Be 88] Bennett, S., Real-time Computer Control, Prentice Hall, Englewood Cliffs, NJ, 1988.

[Bi 90] Bit3 Computer Corporation, Bus Adaptor Products Users Manual, Minneapolis, MN, 1990

[Br 84] Brodie, L., Thinking FORTH, Prentice Hall, Englewood Cliffs, New Jersey, 1984.

[Co 91] Coad, P. and Yourdon, E., Object Oriented Analysis, Yourdon Press Computing Series, Pretince Hall, Inc., Englewood Cliffs, New Jersey, 1991.

[Ha 91] Harbison, S.P. and Steele, G.L. Jr., C A Reference Manual, Prentice Hall Software Series, Englewood Cliffs, New Jersey, 1991.

[Ha 88] Hatley, D.J. and Pirbhai, I.A., Strategies for Real-Time System Specification, Dorset House Publishing Co., Inc., N.Y., N.Y., 1988.

[Hu 92] Huang, H., Hira, R., and Feldman, P., "A Submarine Simulator Driven by A Hierarchical Real-Time Control System Architecture," NISTIR 4875, NIST, 1992. Order through NTIS, Order Number PB92-213354/AS.

[Hu 91] Huang, H., Quintero, R., and Albus, J.S., "A Reference Model, Design Approach, and Development Illustration toward Hierarchical Real-Time System Control for Coal Mining Operations," CONTROL AND DYNAMIC SYSTEMS, ADVANCES IN THEORY AND APPLICATIONS book series chapter, Volume 46, Academic Press, 1991.

[Hu 90] Huang, H., and Quintero, R., "Task Decomposition Methodology for the Design of a Coal Mining Automation Hierarchical Real-Time Control System," The Fifth IEEE International Symposium on Intelligent Control, Philadelphia, PA, 1990.

[Jo 91] Johnson, D.W., Szabo, S., McClellan, H.W., DeBellis, W.B., "Towards an Autonomous Heavy Lift Robot for Field Applications", 8th International Symposium on Automation and Robotics in Construction, 3-5 June 1991, Stuttgart Germany.

[Ko 92] Kowal, J.A., Behavior Models, Specifying User's Expectations, Prentice Hall, Englewood Cliffs, New Jersey, 1992.

[La 91] Lauwerier, H., Fractals, Princeton University Press, Princeton, NJ, 1991.

[Qu 92] Quintero, R. and Barbera, A.J., An RCS Methodology for Developing Intelligent Control Systems, NISTIR 4936, 1992.

[Si 90] Simmons, R., et al., "Autonomous Task Control for Mobile Robots," Proceedings of the Fifth International Symposium on Intelligent Control, Philadelphia, PA, September, 1990.

[St 92] Strassmann, P.A., "The Use of the Ada Computer Language: The DoD Context," Cross Talk, the Monthly Technical Report of the United States Air Force Software Technology Support Center, Hill AFB, Utah, February, 1992.

[Sz 92] Szabo, S., Scott, H.A., Murphy, K.N., Legowik, S.A., Bostelman, R.V., "High-Level Mobility Controller for a Remotely Operated Unmanned Land Vehicle," Journal of Intelligent and Robotic Systems, 5: 63-77, 1992.

[Sz 90] Szabo, S., Scott, H., Murphy, K., Legowik, S., "Control System Architecture for a Remotely Operated Unmanned Land Vehicle, proceedings of Fifth IEEE International Symposium on Intelligent Control," September 5-7, 1990, Philadelphia, PA.

[Ta 92] Tarnoff, N., Jacoff, A., Lumia, R, "Graphical Simulation for Sensor Based Robot Programming," Journal of Robotic Systems, 5:49-62, 1992.

APPENDIX A: Comparison between Task Control Architecture (TCA) and RCS:

The task control architecture (TCA), developed by Simmons [Si 90] of Carnegie Mellon University, deals with the same real-time embedded system control problem domain as RCS does. TCA specifies a generic block structure capturing common capabilities that robotic control systems may possess. System capabilities, including: hierarchical planning, concurrent planning, execution and perception, coordination of multiple tasks, error recovery, and reaction to changes, etc., are the main blocks specified in TCA's control structure. These "capability modules" are tied to the physical sensory and actuator systems via a central control module. System execution is facilitated through message routing (including commands) among all the modules served by this central control module.

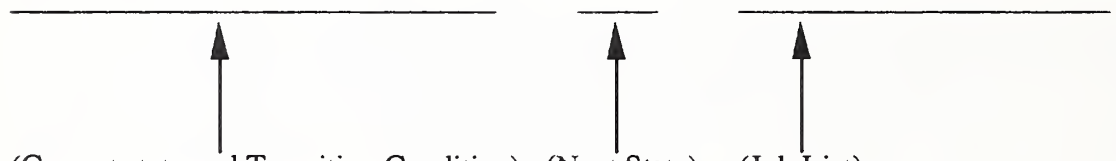
TCA employs a central control module as the heart of the system execution, which, as the authors [Si 90] point out, presents a potential bottleneck as the system complexity grows. On the other hand, RCS features generic controller nodes in distributed environments, which, in our view, should be better suited for dealing with complex real-time control system problems. TCA shares the same view as RCS in the use of task trees to describe command chains. In addition, TCA specifies that modules can impose temporal constraints to sequence the planning and execution of system commands. In the RCS methodology, state diagrams and state tables are used (as a step beyond task trees) to provide a more robust and systematic method of describing the transition of system behavior among different subsystems in both the temporal and the spatial aspects.

The TCA central control module contains a scheduler to arbitrate the resources and to handle messages. This is in contrast to our implementation which contains a deterministic execution sequence (user specified) coupled with non-blocking communication. This execution model ensures determinism (users know exactly what the system state is at any instance of time), concurrency (the system executes and does not wait for the incoming messages), and data integrity.

APPENDIX B: A Propulsion Ahead State Table in Smacro

```
PRDEF        Routine        AHEAD        STATE-TABLE

$- new-command  moving-ahead      |  S1      CALC-FWD-PROP-SPEED
                                             prop-speed => rpm
                                             tb-Ahead  tb# INC
$- new-command  moving-back       |  S2      tb-Stop tb# INC
$- S2 tb-done                     |  S1      CALC-FWD-PROP-SPEED
                                             prop-speed => rpm
                                             tb-Ahead  tb# INC
$- S1 tb-done below-speed         |          prop-speed INC
                                             prop-speed => rpm
                                             tb-Ahead  tb# INC
$- S1 tb-done above-speed         |          prop-speed DEC
                                             prop-speed => rpm
                                              tb-Ahead
                                             tb# INC
$- S1    tb-done  at-speed        |  S1      done
$- default   NOP   END-ST
End-routine
```


(Current state and Transition Condition) (Next State) (Job List)

APPENDIX C: A Propulsion Ahead State Table in C

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

PRPSE: This is the state table for the AHEAD command of this level.

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

static void pr_ahead(void)
{
    ST_BGN
        new_command &&
        (ship_dir == MOVING_AHEAD || ship_dir == STOPPED)
        THEN
            pr_cur_state = S1;
            calc_fwd_prop_speed();
            tb_co.command = TB_AHEAD;
            tb_co.command_num ++;
            tb_co.rpm = prop_speed;
    ST
        new_command &&
        ship_dir == MOVING_BACK
        THEN
            pr_cur_state = S2;
            tb_co.command = TB_STOP;
            tb_co.command_num ++;
    ST
        pr_cur_state == S2      &&
        tb_si.status == TB_DONE
        THEN
            pr_cur_state = S1;
            calc_fwd_prop_speed();
            tb_co.command = TB_AHEAD;
            tb_co.command_num ++;
            tb_co.rpm = prop_speed;
    ST
        pr_cur_state == S1      &&
        tb_si.status == TB_DONE     &&
        sub_speed_status == BELOW_SPEED
        THEN
            ++ prop_speed;
            tb_co.command = TB_AHEAD;
            tb_co.command_num ++;
            tb_co.rpm = prop_speed;
    ST
        pr_cur_state == S1      &&
        tb_si.status == TB_DONE     &&
        sub_speed_status == ABOVE_SPEED
        THEN
            -- prop_speed;
            tb_co.command = TB_AHEAD;
            tb_co.command_num ++;
            tb_co.rpm = prop_speed;
    ST
        pr_cur_state == S1      &&
        tb_si.status == TB_DONE     &&
        sub_speed_status == AT_SPEED
        THEN
            pr.so.status = PR_AT_GOAL;
    DEFAULT
    ST_END
}
```

APPENDIX D: Generic Templates

```
/*
* Example of Generic Controller Module for RCS on a PC Compatible in
* Microsoft C v6.00
* This is the Template used for the RCS Software Demonstration
* The following prefixes are used:
      PR - this module
      TB - subordinate to this module
      SM - supervisor of this module

* The following are non-standard C conventions used in this source
  code:

  #define ST_BGN            if(
  #define THEN         ){
  #define ST              )else if(
  #define DEFAULT         )else {
  #define ST_END            }

  #define CASE_BGN      if(
  #define CASE         )else if(
  #define CASE_END     }

  #define COPY_BUFFER(x,y,z)  (memcpy((char *)(x),(char *)(y),(z)))

  dprintf( ) is a specialized routine for writing to the VGA display
           rapidly.

* The template may be easily extended by adding plans and Sensory
* Processing & World Modeling functions
* Search for !!! for places where the plan in Appendix C can be added
*/

/*
* Include Files
*/
#include "global.h"     /* global definitions */

/*
* Definitions and Macros
*/
#define PR_DEBUG_LINE                 5   /* line used for debug screen */

/*
* Private Global Variables
*/
/* Start Time - Used for Performance Metrics */
static unsigned                   pr_cycle_start_time;

/* Initialize Current State */
static int                        pr_cur_state = S0;

/* Declare Local Copies of Interface Buffers */
static PR_BUFFER                  pr;     /* command from above */

/* Commands to and Status from Subordinates, TB */
static TB_COMMAND                 tb_co;
static TB_STATUS                  tb_si;

/*
* World Model Data Local Copies
*/
static enum OPER_MODE             w_op_mode;

/*
* Function Prototypes - Listed by Category
*/

/* Primary Controller Module Routine, calls all others */
void pr_controller();

/* Behavior Generation - Planning, Execution, & Job Assignment */
static void pr_decision_process     (void);
static void pr_check_if_new_command (void);

/* Sensory Processing and World Modelling */
static void pr_pre_process          (void);
static void pr_post_process         (void);

/* Debug Functions for Display */
static void pr_print_in_data        (void);
static void pr_print_out_data       (void);

/* Commands which may be received - State table plans */
static void pr_init                 (void);
static void pr_halt                 (void);
/* !!! Declare more commands here !!! */

/*QQ*******************************************************************
**
RTNS:  None.
PRPSE: External Routine used by main to execute this task.  This is the
       only entrance to this task. Note that no parameters are passed
       explicitly in the function call. All parameter passing is done
       in the pre_process by copying the interface buffers.
ATHR:  David G. Quigley
CRTD:  10/09/91
MDFD:  Ron Hira & Hui-Min Huang 11/20/92
NOTES: None.
PRBLM: None.
OTHER: None.
*********************************************************************
*/
void pr_controller()
{
/*  READ START TIME, COPY IN INTERFACING BUFFERS AND CONTROL MODEL */
/*  CHECK SUBORDINATE STATUS                                       */
        pr_pre_process();

/*  EXECUTE COMMON FUNCTIONS SENSORY PROCESSING & WORLD MODELLING  */
        /* Function calls made for SP & WM */


/*  CHECK IF NEW COMMAND                                           */
        pr_check_if_new_command();

/*  BG/JA/PL/EX -SELECT STATE TABLE AND EXECUTE                    */
        pr_decision_process();

/*  EXECUTE COMMON FUNCTIONS SENSORY PROCESSING & WORLD MODELLING  */
        /* Function calls made for SP & WM */

/*  COPY OUT INTERFACING BUFFERS & CONTROL MODEL, DISPLAY DATA &   */
/*  DEBUG, CALCULATE PERFORMANCE                                   */
        pr_post_process();
}

/*QQ*******************************************************************
**
```

```
RTNS:  None.
PRPSE: This routine's function is to plan the control via a specific
         state table based on the command in as well as other inputs and
         sensor data Behavior Generation.
ATHR:  David G. Quigley
CRTD:  10/09/91
MDFD:  None
NOTES: None.
PRBLM: None.
OTHER: None.
************************************************************************
*/
static void pr_decision_process(void)
{
/* DEBUG - for running modes */
        /*
        * If run flag in not set return
        */
        if (pr.model.dont_run == TRUE)
        {
                return;
        }

        /*
        * If single step flag is set but single step number has not changed
        *     return
        */
        if (
        (pr.model.single_step == TRUE) &&
        (pr.model.single_step_num == pr.perfo.single_step_num))
        {
                return;
        }

        /*
        * If single step flag is set and single step number has changed
        *     cycle once.
        */
        if (
        (pr.model.single_step == TRUE) &&
        (pr.model.single_step_num != pr.perfo.single_step_num))
        {
                pr.perfo.single_step_num = pr.model.single_step_num;
        }


/* BEHAVIOR GENERATION STATE TABLE - PLANNING, EXECUTION, AND JOB ASSIGNMENT */
        /*
         * Execute the proper command
         */
        ST_BGN
                pr.ci.command == PR_INIT
                THEN
                        dprintf(PR_DEBUG_LINE,8,"init     ");
                        pr_init();
        ST
                pr.ci.command == PR_HALT
                THEN
                        dprintf(PR_DEBUG_LINE,8,"halt     ");
                        pr_halt();

/* !!! Commands may be added here in the same format as above  !!! */
        DEFAULT
                dprintf(PR_DEBUG_LINE,8,"INVALID   ");
        ST_END
}

/*QQ*********************************************************************
**
RTNS:  None.
PRPSE: This routine handles all of the preprocessing this level.  It
         includes:  Reading the global interface buffers,
         Checking executing status of children;
ATHR:  Hui-Min Huang
CRTD:  11/21/91
MDFD:  None
NOTES: None.
PRBLM: None.
OTHER: None.
************************************************************************
*/
static void pr_pre_process(void)
{
/* DEBUG - Display */
        dprintf(PR_DEBUG_LINE,1,"PR");

/* Performance */
        /*
        * Starting time for PR controller module
        */
        pr_cycle_start_time = W->timer_counter;

/* Copying Interfacing Buffers */
        /*
        * Get incoming commands from superior module.
        */
        COPY_BUFFER(&pr,&(G->pr_buf),sizeof(PR_BUFFER));

        /*
        * Get status back from subordinates (Turbine - TB)
        */
        COPY_BUFFER(&tb_si,&(G->tb_buf.so),sizeof(TB_STATUS));
        COPY_BUFFER(&tb_co,&(G->tb_buf.ci),sizeof(TB_COMMAND));

        /*
        * Copy in world model data to local copies
        */
        COPY_BUFFER(&w_op_mode,&(W->pr_w_op_mode),sizeof(enum OPER_MODE));

/* DEBUG  */
        /*
        * If the subordinate status back echo number does not
        * match its command number, assume the subordinate is still executing.
        */
        if (tb_si.status_num != tb_co.command_num)
        {
                tb_si.status = TB_EXECUTING;
        }


        /*
        * Debug Display - Execute print_in_data
        */
        pr_print_in_data();
}


/*QQ*********************************************************************
```

```
**
RTNS:  None.
PRPSE: This is the state table (plan) for the INIT command of this level.
ATHR:  Hui-Min Huang
CRTD:  10/14/91
MDFD:  None.
NOTES: None.
PRBLM: None.
OTHER: None.
*************************************************************************
*/
static void pr_init(void)
{
        /* Plan for INIT command */
        ST_BGN
                new_command     /* conditions */
                THEN
                        /* Set current state of PR control */
                        pr_cur_state = S1;
                        /* Command and command number to subordinate TB */
                        tb_co.command = TB_INIT;
                        tb_co.command_num ++;
        ST
                pr_cur_state == S1      &&
                tb_si.status == TB_DONE
                THEN
                        pr_cur_state = NOP;
                        pr.so.status = PR_DONE;
        DEFAULT
        ST_END
}

/*QQ*********************************************************************
**
RTNS:  None.
PRPSE: This is the state table (plan) for the HALT command of this level.
ATHR:  Hui-Min Huang
CRTD:  10/14/91
MDFD:  commented out other actuator stuff 12/31/91
MDFD:  None
NOTES: None.
PRBLM: None.
OTHER: None.
*************************************************************************
*/
static void pr_halt(void)
{
        /* Plan for HALT command */
        ST_BGN
                new_command
                THEN
                        pr_cur_state = S1;
                        tb_co.command = TB_HALT;
        ST
                pr_cur_state == S1              &&
                tb_si.status == TB_DONE
                THEN
                        pr_cur_state = NOP;
                        pr.so.status = PR_DONE;
        DEFAULT
        ST_END
}

/* !!! Extra Plans may be added here, in similar format as above. !!! */
```

```
/*QQ*********************************************************************
**
RTNS:  None.
PRPSE: This routine handles the writing out of all interface buffers as
        well as any other required Sensory Processing & World Modeling.
ATHR:  David G. Quigley
CRTD:  10/09/91
MDFD:  None
NOTES: None.
PRBLM: None.
OTHER: None.
*************************************************************************
*/
static void pr_post_process(void)
{
/* Copy Interface Buffers */
        /*
        * Write status back to superior module contained in the PR_BUFFER
        */
        COPY_BUFFER(&(G->pr_buf.so),&pr.so,sizeof(PR_STATUS));
        COPY_BUFFER(&(G->pr_buf.perfo),&pr.perfo,sizeof(PR_PERFORMANCE));

        /*
        * Write commands to subordinate TB
        */
        COPY_BUFFER(&(G->tb_buf.ci),&tb_co,sizeof(TB_COMMAND));

        /*
        * Write world data
        */
        COPY_BUFFER(&(W->pr_w_op_mode),&w_op_mode,sizeof(enum OPER_MODE));

/* DEBUG */
        /*
        * Calculate performance metrics for module execution
        */
        pr.perfo.last_cycle_time = (W->timer_counter-pr_cycle_start_time) * 21;

        if (pr.perfo.min_cycle_time == 0)
        {
                pr.perfo.min_cycle_time = 0xFFFF;
        }
        if (pr.perfo.last_cycle_time > pr.perfo.max_cycle_time)
        {
                pr.perfo.max_cycle_time = pr.perfo.last_cycle_time;
        }
        if (pr.perfo.last_cycle_time < pr.perfo.min_cycle_time)
        {
                pr.perfo.min_cycle_time = pr.perfo.last_cycle_time;
        }

        /*
        * Display print out data
        */
        pr_print_out_data();
}

/*QQ*********************************************************************
**
RTNS:  None.
PRPSE: This routine checks to see if the command receieved from above is
        a new command.  It checks this by comparing the current command
        in number with the last command in number it received (stored
        in status out number). If they are different, it is assumed that
        it is a new command.
ATHR:   David G. Quigley
CRTD:   10/09/91
MDFD:   None
NOTES:  None.
PRBLM:  None.
OTHER:  None.
*************************************************************************
*/
static void pr_check_if_new_command(void)
{
        if (pr.ci.command_num != pr.so.status_num)
        {
                dprintf(PR_DEBUG_LINE,5,"NC");
                pr.so.status_num = pr.ci.command_num;
                new_command = TRUE;
                pr_cur_state = S0;
                pr.so.status = PR_EXECUTING;
        }
        else
        {
                new_command = FALSE;
                dprintf(PR_DEBUG_LINE,5,"  ");
        }
}


/*QQ*********************************************************************
**
RTNS:   None.
PRPSE:  This routine prints the command and command number received this
        cycle to the screen if the Debug for this level is active.
ATHR:   Hui-Min Huang
CRTD:   11/21/91
MDFD:   None
NOTES:  None.
PRBLM:  None.
OTHER:  None.
*************************************************************************
*/
static void pr_print_in_data(void)
{
        dprintf(PR_DEBUG_LINE,35,"%5.5u",pr.ci.command_num);
        dprintf(PR_DEBUG_LINE,30,"%4.4d",pr.ci.command);

        dprintf(PR_DEBUG_LINE+20,1,"PR");
        if (pr.model.dont_run == TRUE)
        {
                dprintf(PR_DEBUG_LINE+20,10,"STOP");
        }
        else
        {
           dprintf(PR_DEBUG_LINE+20,10,"RUN ");
        }

        if (pr.model.single_step == TRUE)
        {
                dprintf(PR_DEBUG_LINE+20,16,"SINGLE");
        }
        else
        {
                dprintf(PR_DEBUG_LINE+20,16,"AUTO  ");
        }

        if (pr.model.simulate == TRUE)
        {
                dprintf(PR_DEBUG_LINE+20,23,"SIMU");
        }
        else
        {
           dprintf(PR_DEBUG_LINE+20,23,"REAL");
        }
}

/*QQ*********************************************************************
**
RTNS:    None.
PRPSE:   This routine prints the status, status number, and state matched
          for this cycle to the screen if the Debug for this level is active.
ATHR:    Hui-Min Huang
CRTD:    11/21/91
MDFD:    None
NOTES:   None.
PRBLM:   None.
OTHER:   None.
*************************************************************************
*/
static void pr_print_out_data(void)
{
        dprintf(PR_DEBUG_LINE,48,"%5.5u",pr.so.status_num);
        dprintf(PR_DEBUG_LINE,43,"%4.4d",pr.so.status);
        if (pr_cur_state == NOP)
        {
                dprintf(PR_DEBUG_LINE,55,"NOP   ",pr_cur_state);
        }
        else
        {
                dprintf(PR_DEBUG_LINE,55,"S%d   ",pr_cur_state);
        }
        if ((pr.model.dont_run == TRUE) ||
                (pr.model.single_step == TRUE))
        {
                dprintf(PR_DEBUG_LINE+20,35,"
        }
        else
        {
                dprintf(PR_DEBUG_LINE+20,35,"%5.5u  %5.5u  %5.5u",

pr.perfo.last_cycle_time,

pr.perfo.min_cycle_time,

pr.perfo.max_cycle_time);
        }
}
```

```
/*
* PR buffer definitions for the Generic Controller Template
*/
enum pr_commands
{
        PR_INIT = 2100,
        PR_HALT,
/* !!! Extra commands (plans) may be defined here !!! */
};

enum pr_responses
{
        PR_NOT_READY = 0,
        PR_EXECUTING,
        PR_DONE,
        PR_ERROR,
};

/*
* Module buffer interface structures:
*/
typedef struct
{
    unsigned                command_num;
    enum pr_commands        command;
/* !!! Command parameters may be added here, e.g., ship_speed !!! */
} PR_COMMAND;

typedef struct
{
     unsigned                  status_num;
     enum pr_responses         status;
     unsigned                  error_num;
} PR_STATUS;


typedef struct
{                                 /* pr mode structure */
     boolean                  dont_run;
     boolean                  single_step;
     boolean                  simulate;
     unsigned                 single_step_num;
} PR_MODE;

typedef struct
{                                 /* pr performance */
     unsigned                 last_cycle_time;
     unsigned                 min_cycle_time;
     unsigned                 max_cycle_time;
     unsigned                 single_step_num;
} PR_PERFORMANCE;


typedef struct
{                                 /* pr command-response buffer */
     PR_COMMAND                ci;
     PR_STATUS                 so;
     PR_MODE                   model;
     PR_PERFORMANCE            perfo;
/* Sensors and Actuators may be added here */
} PR_BUFFER;
```