

AD-A274 783



9/

2

NAVAL POSTGRADUATE SCHOOL
Monterey, California

DTIC
ELECTE
JAN 24 1994
S C D



THESIS

**A STRUCTURED PROGRAMMING APPROACH
FOR
COMPLEX AUV MISSION CONTROL**

by

Richard P. Blank

September, 1993

Thesis Advisor:

Anthony J. Healey

Approved for public release; distribution is unlimited.

94

1

21

139

94-01968



REPORT DOCUMENTATION PAGE			Form Approved OMB No. 0704-0188
Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instruction, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188) Washington DC 20503.			
1. AGENCY USE ONLY	2. REPORT DATE 23 September 1993	3. REPORT TYPE AND DATES COVERED Master's Thesis	
4. TITLE AND SUBTITLE A STRUCTURED PROGRAMMING APPROACH FOR COMPLEX AUV MISSION CONTROL		5. FUNDING NUMBERS	
6. AUTHOR(S) BLANK, Richard Peter			
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Naval Postgraduate School Monterey, CA 93943-5000		8. PERFORMING ORGANIZATION REPORT NUMBER	
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)		10. SPONSORING/MONITORING AGENCY REPORT NUMBER	
11. SUPPLEMENTARY NOTES The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government.			
12a. DISTRIBUTION/AVAILABILITY STATEMENT Approved for public release; distribution is unlimited.		12b. DISTRIBUTION CODE A	
13. ABSTRACT Reconfigurability and reliability are two keys for the success of an AUV mission control software. The Strategic layer of our software architecture is the level where control of the mission is accomplished. Here, code may change to meet the requirements of different missions and must therefore be easily reconfigurable. Structured programming is one method of developing this logical control code for the Strategic level. This thesis will show that this approach is a workable alternative to a strict rule based language currently proposed, but may end up with a large number of code lines to consider if missions are changed.			
14. SUBJECT TERMS Structured Programming, Mission Control, Autonomous Vehicles, Robotics.		15. NUMBER OF PAGES 126	
		16. PRICE CODE	
17. SECURITY CLASSIFICATION OF REPORT Unclassified	18. SECURITY CLASSIFICATION OF THIS PAGE Unclassified	19. SECURITY CLASSIFICATION OF ABSTRACT Unclassified	20. LIMITATION OF ABSTRACT UL

Approved for public release; distribution is unlimited.

**A Structured Programming Approach
for
Complex AUV Mission Control**

by

Richard P. Blank
Lieutenant, United States Navy
B.S., Worcester Polytechnic Institute

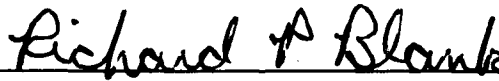
Submitted in partial fulfillment
of the requirements for the degree of

MASTER OF SCIENCE IN MECHANICAL ENGINEERING

from the

NAVAL POSTGRADUATE SCHOOL
September 1993

Author:

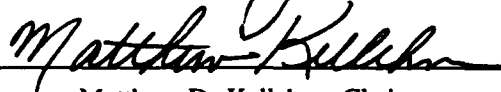


Richard P. Blank

Approved by:



Anthony J. Healey, Thesis Advisor



Matthew D. Kelleher, Chairman
Department of Mechanical Engineering

ABSTRACT

Reconfigurability and reliability are two keys for the success of an AUV mission control software. The Strategic layer of our software architecture is the level where control of the mission is accomplished. Here, code may change to meet the requirements of different missions and must therefore be easily reconfigurable.

Structured programming is one method of developing this logical control code for the Strategic level. This thesis will show that this approach is a workable alternative to a strict rule based language currently proposed, but may end up with a large number of code lines to consider if missions are changed.

Accession For	
NTIS CRA&I	☒
DTIC TAB	☐
Unannounced	☐
Justification	
By	
Distribution /	
Availability Codes	
Dist	Avail and/or Special
A-1	

TABLE OF CONTENTS

I.	INTRODUCTION - THE NEED FOR AUTONOMOUS VEHICLES . . .	1
A.	BACKGROUND	1
B.	THE NEED FOR A MISSION LEVEL CONTROL SYSTEM . .	2
C.	AN EXAMPLE OF A MINE SEARCH MISSION	3
D.	SCOPE OF THE THESIS	5
II.	BACKGROUND IN CONTROL SOFTWARE ARCHITECTURE . . .	6
A.	INTRODUCTION	6
B.	EXPERT SYSTEMS	6
C.	RULE BASED SYSTEMS	8
	1. Search Pattern	8
	2. AND/OR Search Trees	10
	3. Breadth-First-Search	12
	4. Depth-First-Search	12
	5. Inference Engines	13
	6. Chaining Methods	14
D.	STRUCTURED PROGRAMMING	15
	1. Processing Logic Trees (PLT)	16
	2. PLT Rules	17
E.	NEED FOR EASILY RECONFIGURABLE REAL TIME CONTROL CODE	19

F.	DISCUSSION OF VARIOUS SOFTWARE ARCHITECTURES FOR	
	AUV CONTROL	20
	1. Hierarchical Architecture	20
	2. Layered Architecture	23
	3. Hybrid Architectures	26
G.	TRI-LEVEL CONTROL SOFTWARE	27
	1. Strategic Level	30
	2. Tactical Level	30
	3. Execution Level	31
III.	STRATEGIC LEVEL CODE IMPLEMENTATION	33
A.	PROLOG IMPLEMENTATION	33
B.	C CODE IMPLEMENTATION	38
IV.	VALIDATION FOR THE 'FLORIDA MISSION'	44
A.	INTRODUCTION	44
B.	NORMAL OPERATIONAL SCENARIO	45
C.	VEHICLE TYPE FAILURE AND EMERGENCY RECOVERY	48
V.	CONCLUSIONS AND RECOMMENDATIONS	50
A.	CONCLUSIONS	50
B.	RECOMMENDATIONS	50
	APPENDIX A	52
	APPENDIX B	108

LIST OF REFERENCES 114

INITIAL DISTRIBUTION LIST 115

LIST OF TABLES

TABLE I	CHARACTERISTICS OF THE RBM	29
TABLE II	PROLOG CODE FOR THE STRATEGIC LEVEL	34
TABLE III	SAMPLE MISSION LOG OF THE TRANSIT PHASE	47
TABLE IV	SAMPLE OF A TERMINATED MISSION	49

LIST OF FIGURES

Figure 2.1 Inference Engine	9
Figure 2.2 Representation of a Rule Set as an AND/OR Tree	11
Figure 2.3 Basic PLT Structure	16
Figure 3.1 A Prolog Statement	36
Figure 3.2 Mission Transition Diagram	40

ACKNOWLEDGEMENTS

First of all I would like to give special thanks and my sincere gratitude to Dr. Anthony Healey, Dr. Sehung Kwak and David Marco for all of their patience and dedication. Without their vast help I would not have understood the mission control logic or even begun to master C. Dr. Healey provided much insight into the development of an expert system and provided me with the tools necessary to develop the overall logic for a complex mission. He helped me to understand the difficult concepts and made the learning process a truly worthwhile experience. I would also like to thank the Naval Postgraduate School Direct Research Fund for their continued support of the AUV project.

Lastly, and most importantly, I would like to thank my wonderful wife Monica. Her tireless support and encouragement helped make my entire graduate education and thesis research so much more enjoyable.

I. INTRODUCTION - THE NEED FOR AUTONOMOUS VEHICLES

A. BACKGROUND

Completely Autonomous Underwater Vehicles (AUVs) are mobile self-contained instrumentation platforms with the capacity to sense in dynamic and unknown environments, plan an intelligent response, and act in accordance with that input and the mission goal with no human supervision. In this regard the AUV is a mobile robot that to a limited extent emulates human behavior, i.e., the ability to sense, decide and act independently. The class of Unmanned Underwater Vehicles (UUVs) encompass AUVs and Remotely Operated Vehicles (ROVs). ROVs are piloted by a human operator and allow for continuous control by the host. The ROV is controlled continuously using a cable link (tether) for power and communication which makes it dependent on human interaction through a manned surface asset. A spectrum of UUVs are now being developed with less dependence on the human pilot requiring high level commands only typically through acoustic communication and correspondingly increased reliance on onboard computer "intelligence".

Autonomous vehicles are ideally suited for operation in dangerous or environmentally unsafe regions. These regions include, but are not limited to deep water, under ice, heavily

polluted waters, and mine fields. AUVs will be able to operate under relatively unrestricted conditions and have much more freedom than the ROVs.

The overall control capability required for an AUV is immense, spanning from dynamic positioning to vehicle cruise control with obstacle avoidance and automatic fault recovery. Inclusive to this control capability is the coordinated operation of sensors, and the monitoring of the vessel's own engineering systems. The control of the AUV is a continuous process.

This thesis addresses the overall mission control of the AUV. The goal was to build an expert system whereby an "Officer of the Deck" (OOD), represented in computer control code, controls the operation of the AUV. The "OOD" presented here was built using a structured programming approach to encompass a multitude of different situations for a specified mission. It is contended that structured programming is a viable method to control an AUV during a mission, even though a large number of lines of code would be necessary.

B. THE NEED FOR A MISSION LEVEL CONTROL SYSTEM

A mission control package is responsible for the overall planning and safety of a mission. Mission replanning can be accomplished in real time to adjust to new equipment capabilities. The control package attempts to take into account all the possibilities that an AUV might encounter

while permitting the AUV to complete the mission in an acceptable manner.

The primary long term goal and a true measure of success in the AUV's real underwater world will be its reliability. Another area of concern is for AUV mission control software to be easily reconfigurable to a changing mission need by the user. The main idea is to build a highly reliable and reconfigurable expert control system that can be easily modified by the non-expert and used effectively on an actual AUV.

C. AN EXAMPLE OF A MINE SEARCH MISSION

A generic AUV mission may involve the initial planning, an outbound transit, search in the designated area, completion of the desired task, the return transit, and recovery of the vehicle. The National Science Foundation sponsored a workshop at the Florida Atlantic University, headed by Steer, Dunn, and Smith (1992), to discuss the advancement of underwater vehicle technology. The workshop participants realized the importance of an inter-institutional competition and demonstrations in order to share current research concepts and more quickly advance the roles of AUVs. The outcome was a planned exhibition with three sample AUV missions. Each mission scenario posed a separate and viable opportunity to demonstrate the role of an AUV.

The tasks were written in a way that even a partial completion of the scenario would result in much experience being gained from the experimental results. The three tasks were made as realistic as possible for the eventual "real world" application of these vehicles. The missions included search and rescue, pollution source location, and navigation with obstacle avoidance.

Each mission was designed to validate certain aspects of the expected AUV capabilities. The mission specific capabilities which were to be demonstrated were search, surveying, water sampling, obstacle identification and avoidance, and the use of a payload. Furthermore, the vehicle had to maintain the control characteristics (i.e., vehicle stability, and maneuvering capabilities) and navigate in a dynamic environment while monitoring all of the vessel's systems.

The search and rescue mission was specifically defined as: Given the parameters of a search region, the AUV will traverse to the region, locate a subsurface buoy, cut the buoy's mooring line, drop a weight as close to the buoy as possible, return to the launch site, and surface. For the purpose of this thesis, the search and rescue mission was slightly modified to make it more applicable to the Navy. Instead of locating a buoy, the AUV would search for mines in the minefield. Once all the mines were located, the mines would be neutralized and the AUV would return to a predetermined

location. These scenarios have been collectively designated as the "Florida Mission" and will form the basis upon which the subsequent development of mission control code is based and described.

D. SCOPE OF THE THESIS

The scope of this work is to examine control software architecture, to examine the necessary rules in an Artificial Intelligent expert system to conduct a typical search mission, and to provide a translation of mission control logic into executable C code that could be the basis of the higher level of a real time control system for the NPS AUV II vehicle.

II. BACKGROUND IN CONTROL SOFTWARE ARCHITECTURE

A. INTRODUCTION

The goal of the mission control software is to build an expert system that will drive a vehicle to successfully accomplish a mission and which would be easily reconfigurable. This would make it adaptable to changing mission requirements. Various software architectures are available to be implemented in an autonomous underwater vehicle (AUV), each with their advantages and disadvantages. The three types of software control architectures currently being used in the mission control of AUV fall broadly into the classes of hierarchical, behavioral, and hybrids of the two extremes. For the purpose of the NPS II AUV, a hybrid tri-level control software is being designed with plans of implementation as appropriate.

B. EXPERT SYSTEMS

An expert system attempts to emulate the behavior of human expertise. In this respect, an expert system is different than a typical computer program. An algorithmic program operates based on set of commands or actions which result in a specific response or reaction. The user initiates the tasks that the system then performs. Expert systems behave much more as consultants do. They may not know or need to know all the intricate details, but they ask specific questions to

obtain and build required information (compiled knowledge) from the user. The user on the other hand simply responds to the questions. Basically, the expert system queries the user as an ongoing dialogue in the generation of a rule set. Answers following user entered queries may then be found by searching the rule set so developed.

An expert system has the following characteristics as defined by Marcus (1986):

- Experts acquire their knowledge base over time and there is virtually no limit to the amount of knowledge an expert can obtain.
- The knowledge base of an expert can change with time, i.e., it is reconfigurable.
- Experts can infer that certain things are true based on what they know. The normal method of computer programming was to calculate the unknown based on known formulas and determinable variables.
- Experts can utilize the knowledge in different ways which becomes handy when conducting multiple operations. This is important when the knowledge is being used for one thing and also needs to be used in another area.
- Experts apply heuristics or rules of thumb in order to arrive at a conclusion rapidly. This is beneficial in order to minimize the amount of general knowledge required and thus reduce the required computer memory to practically nothing. This keeps the system within certain bounds.

In order for an expert system to be defined as such, it must have the above mentioned characteristics. As the search structure grows, the search mechanism expands into a plethora of possibilities known as the phenomenon of combinatorial explosion. This simply means that as the number of alternative choices increases, new nodes are added

exponentially. This is not a realistic feasibility since computers do not have infinite memory. The search space for the NPS II AUV is not large enough to benefit from the use of heuristics. Practically, then, the system is made to work using a rule based system. The problems are solved using a determined method as described below.

C. RULE BASED SYSTEMS

A rule based system uses an inferencing mechanism (engine) to reference a given specified knowledge base or set of rules (McGhee, Byrnes, 1993). The system then utilizes a search and pattern matching technique, to reason and draw conclusions from the knowledge base, working toward satisfaction of a goal condition. A decision is then inferenced based on the current situation. The knowledge base contains the rules, facts, and the initial and goal conditions pertaining to the situations that might be encountered in the mission and a search of the rules is required.

1. Search Pattern

A search pattern is based on one or more logical comparisons which, in an orderly fashion, review the facts (knowledge) in the data base. A proper decision in answer to a query is thus based on the rules and the current facts. This process is known as generate and test. That is, a solution is generated and then tested to verify that it satisfies the initial goal. In the AUV application, the facts

and vehicle conditions are constantly being updated throughout the mission to keep the status or state of the vehicle current.

Known facts are matched to the knowledge base, which in turn leads to new facts. These are in turn applied by the search control mechanism to the knowledge base through the use of operators. Problem solving in a rule based system is shown by the diagram in Figure 2.1.

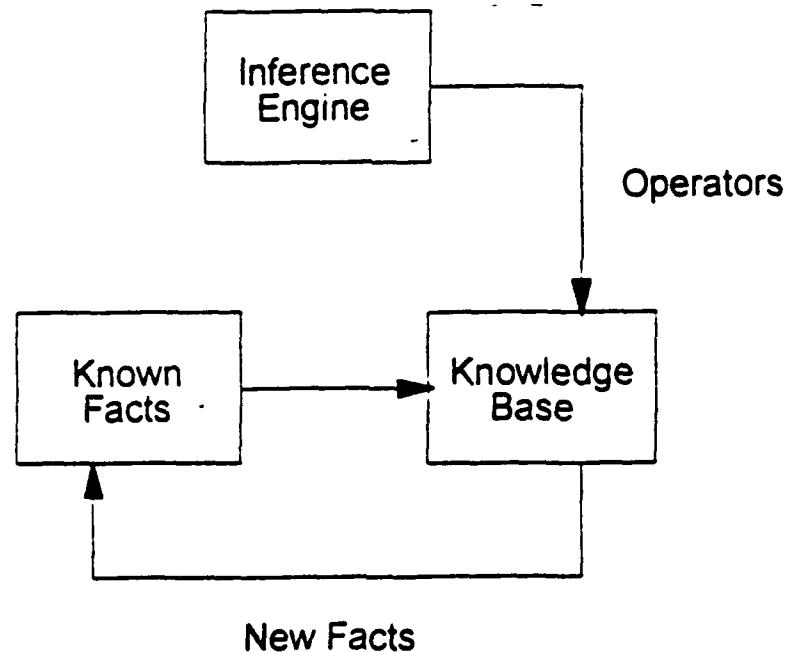


Figure 2.1 Problem Solving in a Rule Based System

A graphical representation of the characteristic structure of the search tree is often useful to provide a road map or outline. This is commonly known as a state graph. In

general, a starting point is branched off to one or more goal points. The goal, for example, may be to proceed to the next waypoint. Each node could represent the possible states of the internal systems. The head node is known as the root node and is the level 0 (primary objective). The successors of level 0 are the levels 1,2,3, etc., and are the steps required to meet the primary objective. Level 1 is known as the goal node. The terminal nodes are known as the "leaves" of the tree. All intermediate nodes are the subgoals of the tree and are levels 2,3, etc., respectively numbered from the root node.

2. AND/OR Search Trees

The general concept of a search tree was further refined by Jackson (1990) to include an AND/OR search tree. A representation of a rule set as an AND/OR tree is shown in Figure 2.2. The branches from a node to its children may be related in two ways. The first is the AND relationship. In order for the goal or subgoal to be satisfied all the subgoals must be achieved. The other possible method to satisfy the requirements for the node is by using the logical OR relationship. This method requires the completion of an alternate route where one branch of many may satisfy the node. Once the problem has been modeled as a state graph a search approach to solve the problem is required. In a logical OR relationship, it is important to prioritize multiple OR

arguments otherwise conflicts could arise. These priorities must then be enforced as the graph is traversed.

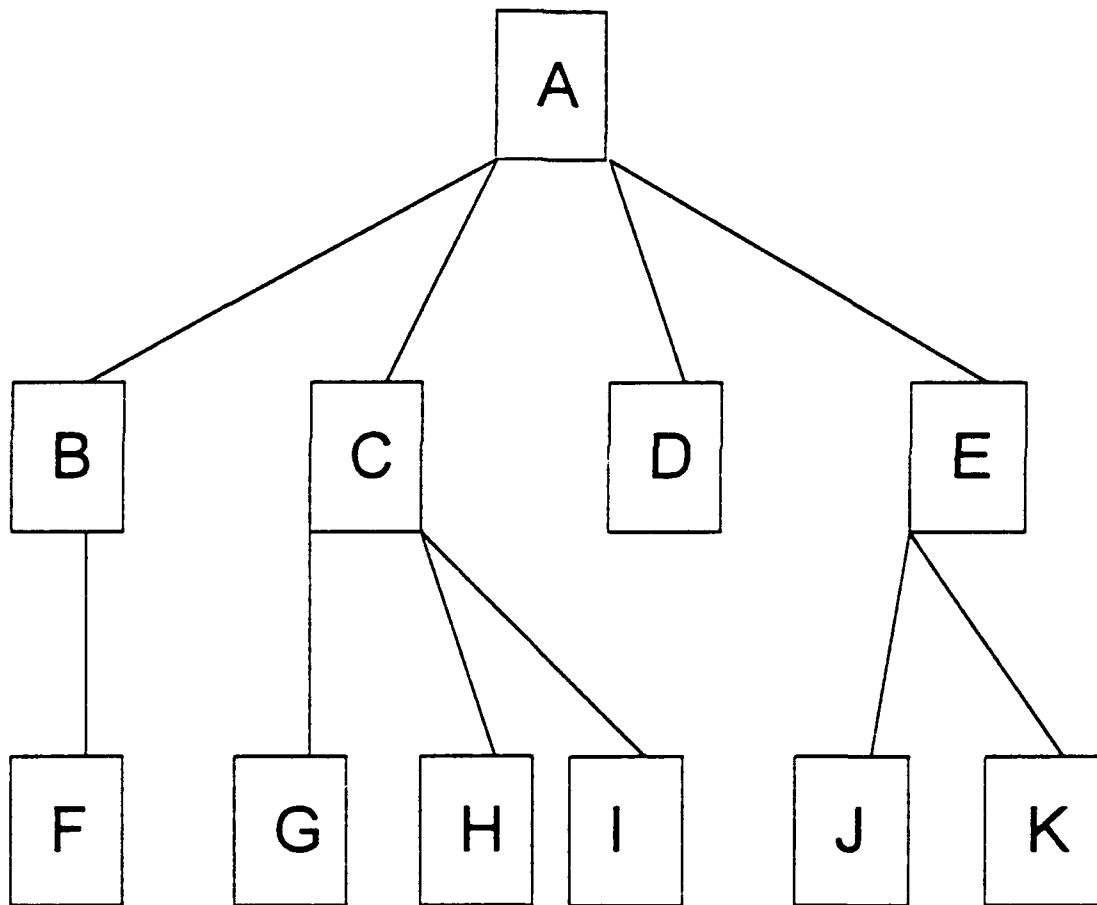


Figure 2.2 AND/OR Search Tree

A blind search method is orderly and methodical and will eventually solve the problem if a solution exists. It is a viable approach especially with small to medium sized problems. Furthermore, as computers continue to become faster such forceful techniques are made more valuable due to their relative simplicity. Two popular methods of blind search are the Breadth-First-Search (BFS) and the Depth-First-Search (DFS).

Heuristics can be used if the state space becomes too large that the blind search method could consume an exorbitant amount of computational processing time. Although the state space associated with the current AUV mission control research is not large enough to benefit from the use of heuristics.

3. Breadth-First-Search

The BFS method examines the nodes moving laterally on the state graph. That is, all the nodes on the current level are searched prior to proceeding to the next level. This continues until the goal has been achieved or all the nodes have been exhausted. The typical process proceeds from the top (level 0) down to the lowest level and from left to right on each level. The advantage of a BFS is that the root goal will always be achieved in the shortest possible path.

4. Depth-First-Search

The typical DFS search begins at the root node and proceeds down to the next level. However, at the next level

it does not proceed laterally, but proceeds along the branch as opposed to the level of the tree. If the desired goal condition is not met it proceeds to the next highest node where an untraversed path is available. This method also continues until the goal has been achieved or traversed as in the BFS. A disadvantage with DFS is that the program may spend much unnecessary effort in a deep subtree, far from a suitable solution that may exist at an upper level. This is also an advantage since these other solutions may not be the desired solutions as is the case with the AUV mission control.

5. Inference Engines

A rule based or an expert system uses the inference engine to implement the search. The inference engine examines the rules in a specific, predetermined order. Once the rules are matched to the current state or situation, the next step is taken, which results in the corresponding action being executed. The function of the inference engine is to determine whether a goal has been satisfied. The inference engine checks the data base to see if the goal is considered to be true. If the goal is not true, then an attempt is made to satisfy the goal.

The conditional part of the rule is the IF part. When the conditional part has been satisfied, the next step can be executed. In some instances more than one rule satisfies the condition. When this occurs the steps are executed according

to some ordered criteria. When the rule has been activated, the actions specified by the rule are carried out. These actions are defined by the rule consequent, or the THEN part of the rule. Each executed action can produce new information which can be used to help find the next rule to be executed. This process of searching, matching and activating rules continues until the goal is satisfied or no new information is being produced.

6. Chaining Methods

Expert systems use this sequence of rules to provide an ordered line of reasoning which the inference engine can use to base its conclusion. It simply provides a sequence of rules to be executed. The inference engine can utilize two approaches to prove the goal: forward and backward chaining. Forward chaining attempts to match one or more of the states from the data base to the rule condition. Backward chaining attempts to match the solution with the goal. In other words, the THEN portion of the rule base is checked, i.e., the data base is examined for the goal. If the goal has been satisfied, the process stops and the rule conditions are asserted as knowledge. Otherwise, the inference engine searches the rule base.

The forward chaining and backward chaining deductive reasoning methods each have their advantages and disadvantages. The choice of which to use is dependent on the

type of application. Forward chaining was utilized in the implementation of the C language in this thesis. In the long run, the user is generally not concerned which chaining control is implemented, as long as the end result is correct.

D. STRUCTURED PROGRAMMING

A structured programming method is another means to use when building an expert system. Combined with the rule based methodologies, it is an alternative solution that can be used in lieu of the strict rule based languages such as PROLOG. Structured programming has been defined as any program whose flow of execution control can be described by using only the three basic control structures, i.e., sequence, selection, and iteration (Jensen, 1981). Structured programming theory deals with converting large and complex flowcharts into standard forms. In this manner they can be represented by iterating and nesting a much smaller and reasonable number of standard control logic structures as the definition suggests.

The core of structured programming lies in the stepwise refinement process. A flowchart is a helpful tool to describe the flow of the execution of the program. The graph is constructed of directed line segments whose direction of flow is indicated by arrows. The line segments begin and end at the nodes similar to the AND/OR Tree. Each node represents a data transformation, decision point or a collection point for the program control paths.

1. Processing Logic Trees (PLT)

Rather than simply flowing in one end of the node and out another, the structured program is modified to allow flow back to the previous node as well. In structured programming this is known as processing a logic tree (PLT). This structure provides a graphical representation of the various parts of the project similar to a flowchart. The PLT permits expansion in detail of the various levels while maintaining the basic program representation. A structured design process is achieved by prohibiting all but a hierarchical approach to the problem solution.

Each node represents the execution of a specified task. The task may be a simple mathematical calculation or a complex function such as "Transit" in the AUV mission. The level of complexity is a function of the abstraction level, or tree level, where the task is defined. The simplest control logic sequence structure of PLT is shown in Figure 2.3.

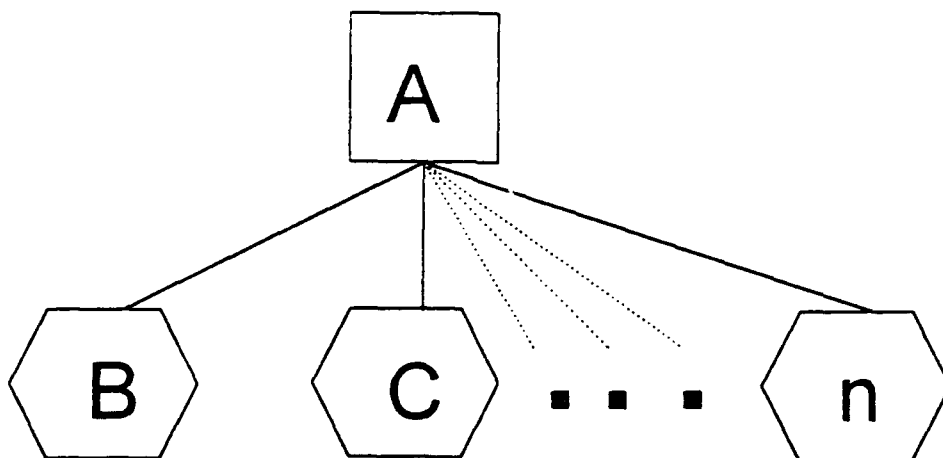


Figure 2.3 Basic Processing Logic Tree Structure

2. PLT Rules

The control logic structure of PLT represents the sequence of operations, i.e., B, C,..., n. Node A collectively represents the sequential tasks of operation {B, C,..., n}. This leads to the fundamental PLT rule:

Rule 1: The processing performed by a node in a processing logic tree is completely defined by its subtrees (subplts).

A simple corollary to this rule states that the processing of a node consists of processing all subtrees of that node.

An obvious follow on to this rule based on this point is:

Rule 2: The Processing of nodes on each level proceeds from left to right. The processing of each node must be complete prior to proceeding to the next node.

This process is fundamental in any ordered tree. Simply stated, the processing of a parent node is not complete until all of its offspring and all of the offspring's offspring, etc., have been processed as well.

The PLT continues in this fashion until the lowest levels have been processed. These lowest levels are identical to the lowest levels on the Rule based method and are similarly called the terminal nodes or "leaves" of the tree. These end nodes contain the most rudimentary tasks and cannot be expanded any further.

In order to keep this process reasonable, each node or subroutine should be strictly limited in size. A manageable size is considered to be a hundred lines or less, often not greater than 30 lines of code. The subroutines should be further limited to one processing function.

The Depth First Search (DFS) can be used in structured programming and is identical to the DFS in rule based programming. The selection criteria allows for the capability to conditionally perform tasks based on a predetermined goal. If the goal is satisfied, the task will be activated.

In this manner a command could be given or a simple question could be asked and a response provided. The program would behave much like the rule based PROLOG and any structured programming language could be used such as PASCAL or C.

The PLT provides a powerful means to represent proper structured programs. A properly structured program is one whose structure can be represented by a hierarchical, nested structure of statements. The one stipulation is that every program block at each nested level, there is only one input and one output path. Strict adherence to this program philosophy, however, can make the program quite complex and also reduces the programs readability, especially for the end user. Without any adherence to structure, though, the program quickly becomes unmanageable and results in increased software life-cycle costs. A well structured program on the other

hand, using conditionals, allows for the judicious use of branching. This further results in a more desirable decrease in the program complexity. In this manner, the PLT complies with the accepted programming philosophy of having only one input path, but having multiple exit paths, when necessary.

E. NEED FOR EASILY RECONFIGURABLE REAL TIME CONTROL CODE

Real time control code is necessary in order to maintain control over the vehicle and its mission in a current stable state. In order to accomplish this the AUV control must be monitored and the data must be continuously updated. One method of achieving this real time control is for the subroutines to be written as simple as possible and be queried often.

The mission profile is initially designed by the programmer to meet the requirements of the end user. In essence an expert system is built. However, a difficulty could arise when these requirements or the mission changes.

The operation of an AUV would normally consist of numerous algorithms of specified actions as it is still a question of research as to how to best organize software modules. These various algorithms are then combined with procedures and rules to encompass the entire mission. The source code is then tested in simulation and eventually installed in the vehicle control computer. The various mission profiles that an AUV can be used for can be quite diverse in their requirements,

goals, and mission specific tasks. Furthermore, the AUV sensors and monitors can change which also result in modifications to the vehicle control software as the mission requirements change.

Therefore, there is an obvious need for great flexibility in the specifications of the vehicle behavior. The goal is to develop a software package that can be easily modified by the end user to meet the changing sensor interfaces and behaviors of the AUV controller.

F. DISCUSSION OF VARIOUS SOFTWARE ARCHITECTURES FOR AUV CONTROL

The importance of the software architecture specification for a control system design cannot be underestimated. Software architecture may mean the method for dividing a complex problem into smaller manageable pieces, the decomposition of program and data according to a set of predetermined criteria, or the breakdown of software components into modules (Zheng, 1990). This thesis will use it as a conceptual design for real time control software.

1. Hierarchical Architecture

Hierarchical control software depends on a structure of hierarchy. In this architecture the control is broken down into successively less complex tasks at the lower levels of a search tree. The various levels are subsequently activated and compiled to form the completed system. This architecture

assumes that the mission planning and actual execution routines are abstracted to the higher levels. The lower levels are responsible for the vehicle specific functions. The overall model must contain, at a minimum, the current state of the vehicle, the current state of the environment, and the current state of the mission.

The hierarchy operates much in the manner of a corporation. Each level receives sensory input directly from the level below it and guidance from the level directly above it. Data elements at the lowest elements or "leaves" consist of the lowest possible structure. This data is then combined with other data at each successively higher level to form abstract data objects.

Another key characteristic of hierarchical architecture is the update frequency of the data. As one moves from the upper hierarchy down toward the "leaves" or terminal nodes there is a faster update rate. At the lower levels the tasks are more simplistic and the amount of data involved is relatively minute. Furthermore, many of the tasks in the terminal nodes involve hardware. The upper levels have a much slower frequency since the quantity of data to be analyzed is much greater and the problems are more complex.

The real time nature arises by the planning horizons of each level. The slower the update frequency, the longer is the lead time required to make corrections or adapt: the

faster the update frequency, the shorter the horizon can be. The length of these horizons is determined by experimentation.

Hierarchical control software architecture utilizes deductive reasoning as in the backward chaining method. The AUV executes its mission by determining if the goal has been satisfied. If the goal is not satisfied, the tasks are divided into smaller, simpler tasks or subgoals. The subgoals are then checked to see if they have been satisfied. If a subgoal fails to be satisfied, an alternative path may be tried. This process continues until all the subgoals of a parent subgoal have been solved. Once these parent goals have been solved, their parent's goals can be solved, etc., until the initial goal has been satisfied. This continues until the goal at level 0 has been satisfied.

Software that relies solely on a hierarchical architecture will have certain disadvantages. These systems are based on "known" assumptions rather than scientific proof. This can result in a breakdown long into the software development. If changes have to be made, one can be quickly sent back to the "drawing board". Another disadvantage is that even the simplest of tasks cannot be performed until the upper hierarchy has been developed. As is the case when numerous personnel are working on a project at different points, incompatibilities can result. Mission logic built this way is often spread throughout the control hierarchy

resulting in a rigid system which is difficult to explain to the user.

2. Layered Architecture

A layered (or behavior-based) architecture (Brooks, 1986 and Bellingham, 1990) is viewed from a behavioral view rather than a function breakdown view. The behavior of the AUV is developed in steps or levels. The desired tasks are first defined. Next, they are grouped according to their level of competence. Every level represents a class of behaviors that the AUV can exhibit. The lowest level represents the most simplistic of behaviors. Each successive level has an increasing level of competence and complexity. The behavior becomes more specific as the top level is approached.

It is important for the control system to be responsive to the higher level goals while continuing to respond to the lower level goals as well. In that regard, each level is a subset of the next higher level. The end objective is to incrementally build a more competent "intelligent" vehicle.

Levels of competence correspond to layers of the control system operating in parallel. A process where each higher level is made to examine and alter the data from the lower level as necessary is known as subsumption. Each upper layer is said to subsume the lower level. Subsume means to

override by including; and to include means to work toward the same goals. Together, the layers achieve the competence required by the top level.

One major difference from the hierarchical architecture is that commands and data are not passed from level to level. The data is distributed among all levels, and each level performs its own sensory processing. The AUV can have the potential to exhibit an intelligent behavior by wiring together multiple layers of control.

The majority of the low level functional requirements of the AUV control can be satisfied using the layered architecture. Multiple concurrent goals can be achieved by having various layers working on different goals. The complexity and time delay of passing data between layers is avoided since the sensory systems are available at each level. Another advantage is that the control system is robust. If one layer fails to produce results, the lower levels still continue to function. Subsumption is extended by adding a new level of competence which is due to the incremental nature of the layering scheme.

Layered architecture reasoning is normally forward in mission control. Forward chaining is therefore the ideal deductive reasoning method for this type of control. Forward chaining is data driven, i.e., computation starts with the existing facts and derives new facts or conclusions from them. This type of system proceeds from conditions known to be true

towards states which the system allows the AUV to establish. The process ends when either no additional facts can be derived from current facts or a goal state is achieved. This method is ideal for layered control entities which rely on the sensory data to determine its behavior.

The primary disadvantage of the layered architecture is that the integration strategy of subsumption can only be validated by trial and error. It is also impossible to verify that mission goals will be achieved. The AUV must behave predictably and reliably in order to meet the users requirements, especially due to the sensitivity and potentially dangerous missions.

Subsumption must address the issue of conflicting commands. If two (or more) commands from two (or more) related behaviors are in direct conflict, the one with the higher priority overrides the lower priority command. Compromise is not a possibility in Subsumption. By permitting preferences for a range of commands, some allowance is made for the selection of commands that can simultaneously satisfy multiple conflicting goals. This is known as cooperation.

Pure subsumption methodology does not explicitly insure that the AUV remain stable. The very nature of an AUV results in operation in an unfriendly or hostile environment. Therefore, it is imperative that the vehicle remains stable. Layered control architectures can get around the stability problem by decoupling the low level control from the layered

behavioral architecture. This results in a pseudo-hybrid system where there is a two level hierarchy with the lowest level being stabilization control of vehicle motion.

3. Hybrid Architectures

The majority of software control architectures for autonomous vehicles in operation today utilize one or the other form of architecture. The two architectures are basically different and therefore very little commonality can be expected to be abstracted in combining the two. However, it is believed that a hybrid architecture will emerge which incorporates the main features of the two types.

The objective of many hybrid architectures is to have explicit world representations at the higher levels of abstraction and utilize layered (behavioral) schemes at the lower levels of abstraction. All three types of architectures are proceeding without any formal methodologies. Hybrids have been developed (Zheng, 1990 and Kwak, 1991) which contain reliable hierarchical systems and real time functionality provided by layered systems. These systems are fairly, easily understood by the non-experts which allows for easier mission and robot reconfiguration, when necessary.

The architecture used in the context of this work will be based on the Rational Behavioral Model (RBM) (Kwak, 1991). The RBM is a three level control architecture, consisting of

a strategic (upper), tactical (middle), and execution (lower) level. Both the strategic and tactical levels represent a true hierarchical approach where commands are passed down and data is passed up between the two levels. In essence there is an isolation of the concise operational doctrine at the strategic level. The lower levels and associated behavior of the lower levels is unconstrained except that it must represent the "inbuilt" capability of the vehicle which must be motion control stable. The true benefit of this type of system is that the global behavior exhibited by the AUV is abstracted to the top, i.e., strategic level. This means that the conflict resolution strategy can be modified by altering this level alone.

The primary advantage of this type of architecture is that the missions can be reconfigured easily. Furthermore, the actual mission specifics are defined at the highest level and how the AUV will respond to problems it encounters are embodied in this rule based strategic level. This thesis will attempt to show that structured programming at the upper level can be used to help make the hybrid architecture a reality.

G. TRI-LEVEL CONTROL SOFTWARE

A tri-level, multiple programming paradigm, software architecture for the control of an AUV is known as a *Rational Behavior Model* (RBM) (Byrnes, 1993). Multiple levels of

organization is nothing new to society. Businesses, governments, and especially the military have used this type of general decision making process for ages. It consists of the upper levels making the majority of the strategic decisions. As the process proceeds down to the lower levels the functions become more specialized and specific. This specifically influences the update rates of the various levels. The higher levels in the hierarchy are characterized by a slower update frequency, broader planning horizons, more abstract sensory information and an increased capability to solve complex problems.

The top level is entirely symbolic and contains no global variables or virtually no memory. The bottom level is synchronous and entirely numerical. The middle level provides an asynchronous interface between the upper and lower levels. The division of command represents that typically found on a manned vehicle, such as a submarine, and consists of the Captain (strategist), his command staff (tacticians) and his crew (executors). The Captain determines the mission by sequencing the goals. The command staff (led by the Officer of the Deck (OOD)) breaks down the tasks to produce commands to the crew to support the achievement of these goals. The crew operate and receive data from the actuators and sensors in response to the commands from the upper levels.

The predominant characteristics of the RBM are shown in Table I (Byrnes, 1993).

TABLE I CHARACTERISTICS OF THE RBM

Strategic Level
- Symbolic computation only; contains mission doctrine/specification - No storage of internal vehicle or external world state variables - Rule based implementation, incorporating rule set, inference engine, and working memory (if required) or a Structured Programming Approach - Non-interruptible, event driven - Directs the Tactical level through asynchronous message passing - Messages may be either commands or queries requiring YES/NO responses - Operates in discrete (Boolean) domain independently of time - Building block: the goal - Abstraction mechanisms: goal decomposition (RBM-B) or rule partitioning (RBM-F); both based on goal driven reasoning
Tactical Level
- Provides asynchronous interface between Strategic and Execution levels - Behaviors (tasks) reside here and may execute concurrently - Behaviors are implemented as methods of objects - Primitive goals activate one or more behaviors - External interface of the model consists of two parts: the behavior activations from the Strategic level and the command/telemetry paths to/from the Execution level - World and Mission models maintained here - Responds to Strategic level with logical TRUE/FALSE - Setpoint, modes, and non-routine data requests are output to the Execution level - Not interruptible except for data transfers; hard deadlines cannot be guaranteed - Operates in discrete event/continuous time domains - Building block: objects with behaviors - Abstraction mechanisms: class and composition hierarchies
Execution Level
- Numeric processing only - Responsible for software to hardware interface, underlying vehicle stability - All synchronous (hard real time) processes reside at this level - Sensor data processed to specification of Tactical level - Servo loops run continuously and concurrently, synchronized by timed interrupts - Operates in continuous space/time domains - Building block: servo loops and signal processing algorithms - Abstraction mechanisms: loop composition, sampling frequency, and data smoothing

1. Strategic Level

The foundation for the Strategic level lies in the top-down decomposition. The mission is broken down into the goal directed reasoning. The goals are then passed directly to the Tactical levels via messages.

The events are goal driven vice data driven as on the Execution level. This level contains no state other than the state of reasoning. The purpose of this is to support determinism and expect predictable, rational responses from the AUV. The Strategic level operates asynchronously in boolean (TRUE/FALSE or YES/NO) space. It acquires the necessary information from the Tactical level by polling during mission execution. The only response it receives is a simple yes or no.

2. Tactical Level

The Tactical level is the middle management level and acts as an interface between the knowledge based Strategic level and the actual vehicle control subsystems of the Execution level. This level may call other routines at the Tactical level or send commands to the Execution level. The data collection and emergency path replanning also occur here. The primary function, however, is to manage the interface between the goals directed by the Strategic level and the actions performed by the Execution level.

The operation of this level occurs in discrete event space and continuous time. The decisions are made based on queries or commands from the upper level which can be received at any time. However, data is only passed up from the Execution level at designated interrupt times. It can detect an event at any time based on stored data received from the lower level. The Tactical level provides information to the Execution level in three types: discrete mode changes, non-routine data requests, and continuous setpoints.

3. Execution Level

The Execution level is where the AUV actually utilizes its sensors and is responsible for the software to hardware interface. This underlies the stability of the AUV. The Execution level controls the motors and control surfaces. In so doing, the depth of the vehicle, heading, and speed are controlled based on commands from the Tactical level. Sensory information as from sonars is also sent back up to the Tactical level to provide data for the upper levels to make their decisions. This also represents the final opportunity for the AUV to avoid danger. Certain situations require the AUV to act without prompting from the upper levels. One type of such inherent reflexive behavior is to "automatically" steer around an obstacle without prior notification (Healey, 1992). Most of this code is based on physical model based control theory and is strictly mathematical in nature. The

systems for the NPS AUV II have been made as robust as possible using sliding mode control theory where feasible.

An AUV must have, at a minimum, the following basic control features:

- Steering autopilot for heading control, or for yaw rate control
- Diving autopilot for stable depth changes or control over pitch angle
- Speed control autopilot to adjust the vehicle speed
- A sonar obstacle detection system
- It is desirable to also have a hovering autopilot for maintaining position in a prescribed attitude.

III. STRATEGIC LEVEL CODE IMPLEMENTATION

The mission at the strategic level is formalized as a logical statement to be proved, called a goal statement. The execution of the program is an attempt to solve the problem or goal statement given the assumptions in the logic of the program.

In principle, the mission control logic is only concerned with *what* is to be done, without bothering with *how* this should be accomplished. The *what* is considered the logic portion and the *how* is the control portion. This conforms nicely with the Strategic level of the AUV where the Captain is not concerned with the intricacies of operation, but primarily wants to know what actions are being carried out.

It is at this level that error recovery procedures as well as the conduct of the mission are specified. Error recovery from both types of failure, with mission related tasks or vehicle system related, must be specified.

A. PROLOG IMPLEMENTATION

Strategic level code for the Florida mission using the NPS II AUV has already been developed and implemented in PROLOG (BYRNES, 1993). The actual code is shown in Table II. In this design the RBM was divided into two sections: a mission specific part called the Mission Specification and a vehicle

TABLE II PROLOG CODE FOR THE STRATEGIC LEVEL

```

/*-----MISSION SPECIFICATION FOR SEARCH AND RESCUE-----*/

initialize :- vehicle_ready_for_launch_p(ANS1),ANS1==1,select_first_waypoint(ANS2).
initialize :- alert_user(ANS),fail.

mission :- in_transit_p(ANS1),ANS1==1,transit,!,transit_done_p(ANS2),ANS2==1,fail.
mission :- in_search_p(ANS1),ANS1==1,search,!,search_done_p(ANS2),ANS2==1,fail.
mission :- in_task_p(ANS1),ANS1==1,task,!,task_done_p(ANS2),ANS2==1,fail.
mission :- in_return_p(ANS1),ANS1==1,return,!,return_done_p(ANS2),ANS2==1,
        wait_for_recovery(ANS3).

transit :- waypoint_control.
transit :- surface(ANS1),wait_for_recovery(ANS2).

search :- do_search_pattern(ANS),ANS==1.
search :- surface(ANS1),wait_for_recovery(ANS2).

task :- homing(ANS1),ANS1==1,drop_package(ANS2),ANS2==1,get_gps_fix(ANS3),ANS3==1,
        get_next_waypoint(ANS4),ANS4==1.
task :- surface(ANS1),wait_for_recovery(ANS2).

return :- waypoint_control.
return :- surface(ANS1),wait_for_recovery(ANS2).

/*-----NPS AUV DOCTRINE-----*/

execute_auv_mission :- initialize,repeat,mission.

waypoint_control :- not(critical_system_prob),get_waypoint_status,plan,
        send_setpoints_and_modes(ANS).

get_waypoint_status :-gps_check,reach_waypoint_p(ANS1),ANS1==1,get_next_waypoint(ANS2).
get_waypoint_status.

gps_check :-gps_needed_p(ANS1),ANS1==1,get_gps_fix(ANS1).
gps_check.

plan :- reduced_capacity_system_prob,global_replan.
plan :- near_uncharted_obstacle,local_replan.
plan.

near_uncharted_obstacle :- unknown_obstacle_p(ANS1),ANS1==1,log_new_obstacle(ANS2).

local_replan :- loiter(ANS1),start_local_replanner(ANS2).

global_replan :- loiter(ANS1),start_global_replanner(ANS2).

critical_system_prob :- power_gone_p(ANS),ANS==1.
critical_system_prob :- computer_system_inop_p(ANS),ANS==1.
critical_system_prob :- propulsion_system_p(ANS),ANS==1.
critical_system_prob :- steering_system_inop_p(ANS),ANS==1.

reduced_capacity_system_prob :- diving_system_p(ANS),ANS==1.
reduced_capacity_system_prob :- buoyancy_system_p(ANS),ANS==1.
reduced_capacity_system_prob :- thruster_system_p(ANS),ANS==1.
reduced_capacity_system_prob :- leak_test_p(ANS),ANS==1.

```

independent part called the NPS AUV Doctrine. The first part is particular to the mission, in this case the Florida mission. The second part is particular to the operation of the AUV. The latter portion should not change too often unless additional sensors or equipment are added. The Mission Specification portion will be tailored according to the current mission. Many of the actions, such as transit and return, will almost always be required. However, this is where the majority of the end user interface will have to occur.

The PROLOG rules represent an if-then relation. The rule is divided into a head and a body. The head of the rule corresponds to the then part and the body is the if part. In general, the code can be considered a goal that is divided into subsequent subgoals. If the subgoals are satisfied, then the goal to the left of the ":-" is satisfied. A comma "," represents an AND operator. A logical OR is represented by writing multiple goals with the same head with a defined order of priority. Thus, the goal can be satisfied by meeting all the subgoal requirements in the first rule or the second, etc. A sample rule statement is shown in Figure 3.1.

Mission success or failure in the Strategic level of the RBM is defined by a predetermined sequential order of rules or goals. When a subgoal rule head match is found, the search proceeds to the first subgoal in that rule and then another match is attempted. The inference engine, which is the

goal :- subgoal ₁ , subgoal ₂ , ..., subgoal _n .	
goal	= head
:-	= if
subgoal ₁ , ..., subgoal _n	= body
,	= and

Figure 3.1 A PROLOG Statement

mechanism driving the search, marks each goal to provide a reference should the current inference chain fail. If a match cannot be found (i.e., all branches of a case statement are false), the computation is simply undone to the last match, and a different computation path is attempted, if available. This is known as backtracking (Sterling, 1986). PROLOG uses a technique called lazy evaluation which means it stops the evaluation of a rule when an AND failure or an OR success occurs. Therefore, the sequential order of the rules and subgoals is critical if the desired effect is to be achieved.

Analysis of the PROLOG code reveals that there are no storage of internal or external world state variables at this level. Basically, there are no variables in the rule head. This results in code that can be easily modified to meet the requirements of a mission reconfiguration.

PROLOG uses a repeat function, which when combined with backtracking, allows for the creation of loops. The first time the loop is encountered, the repeat predicate succeeds and the loop can be entered. Repeat is satisfied subsequent times via the use of backtracking. This permits multiple

attempts to satisfy the subgoals lying to the right of the repeat. A cut, "!", blocks the backtracking beyond that point in the rule statement. This insures the strict, iterative execution of the loop. The cut is a beneficial tool to prevent unnecessary search paths and to essentially force the desired sequence of subgoal testing.

The Strategic level program is initiated when a query is made to the PROLOG inference engine. PROLOG then scans heads of the rule set (program) from top to bottom. The first head encountered is the "initialize" rule statement. Next, the first subgoal of this rule statement, "vehicle_ready_for_launch_p(ANS1)", is encountered. This subgoal is a primitive goal in that it cannot be decomposed any further. When a primitive goal is encountered, it either sends a predicate query or a command to the Tactical level.

The query expects either a TRUE or FALSE response from the Tactical level which in turn influences the ensuing reasoning path of the inference engine. A command is a directive that does not require a response, but rather initiates some action at the Tactical level. The primitive goal "in_transit" is a predicate query, since its argument, "ANS1", will either be TRUE or FALSE. The response from the Tactical level is then compared with "ANS1==1". If ANS1 is 1 (signifying TRUE) the subgoal succeeds. The next subgoal is then selected. In this case the command "transit" is sent to the Tactical level to direct the AUV to transit.

If the `ANS1` is 0 (signifying `FALSE`), the PROLOG inference engine commences backtracking and attempts to re-satisfy the subgoal `"in_transit"`. However, this attempt will also fail since there is no other way to satisfy `"in_transit"`. Consequently, the first `"mission"` fails and the subsequent rule is implemented. Subsequently, the next rule statement is implemented. The process continues in an attempt to satisfy the original query.

The four primary phases of the Florida mission are `"transit"`, `"search"`, `"task"`, and `"return"`. Each phase has two rule statements that can satisfy the goal. The first rule statement of each phase can be satisfied by the sequence of steps that would be followed in a normal operation. The alternative statement is a means of having an emergency action if the software or equipment fail. In the case of this mission, the vehicle would be commanded to `"surface"` and `"wait_for_recovery"`. However, this is only if the primary rule statement should fail. It is essential that the desired chain of events occur in the order required.

B. C CODE IMPLEMENTATION

The C code implementation was written using structured programming. In a structured programming language such as C, all of the functions can be broken down into subroutines. The greater the functions are decomposed, the simpler they are to modify. This thesis used the expert system developed for the

PROLOG code as a model for the structured programming approach.

The logic for the implementation is based on if-then relations as is the PROLOG code. Each goal (subroutine) is divided into other subroutines, and these into subsequent subroutines, etc. When the subroutine is at the primitive level it cannot be decomposed any further. Thus, a predicate query is made or a command is sent to the Tactical level. If there is an output from a subroutine it is a simple TRUE or FALSE which corresponds to what is happening and is not concerned with how it happens. This is the beauty of logical programming.

The Strategic level program is initiated when the main program, "ExeAuvMi" (Execute AUV Mission), calls the subroutine "Mission" and simply asks if the mission is complete or not. A generic mission control transition diagram is shown in Figure 3.2. The main program includes a large while loop which is equivalent to the backtracking and repeat combination in PROLOG. "MissionComplete" is initialized as FALSE so that the loop will be entered at least once. The execution of the mission continues until the mission is complete either in its preferred way or in some default fail safe way. If a fault occurs during one of the four phases, the AUV attempts to satisfy the mission in an appropriate default manner. Default fail safe operation for the AUV has

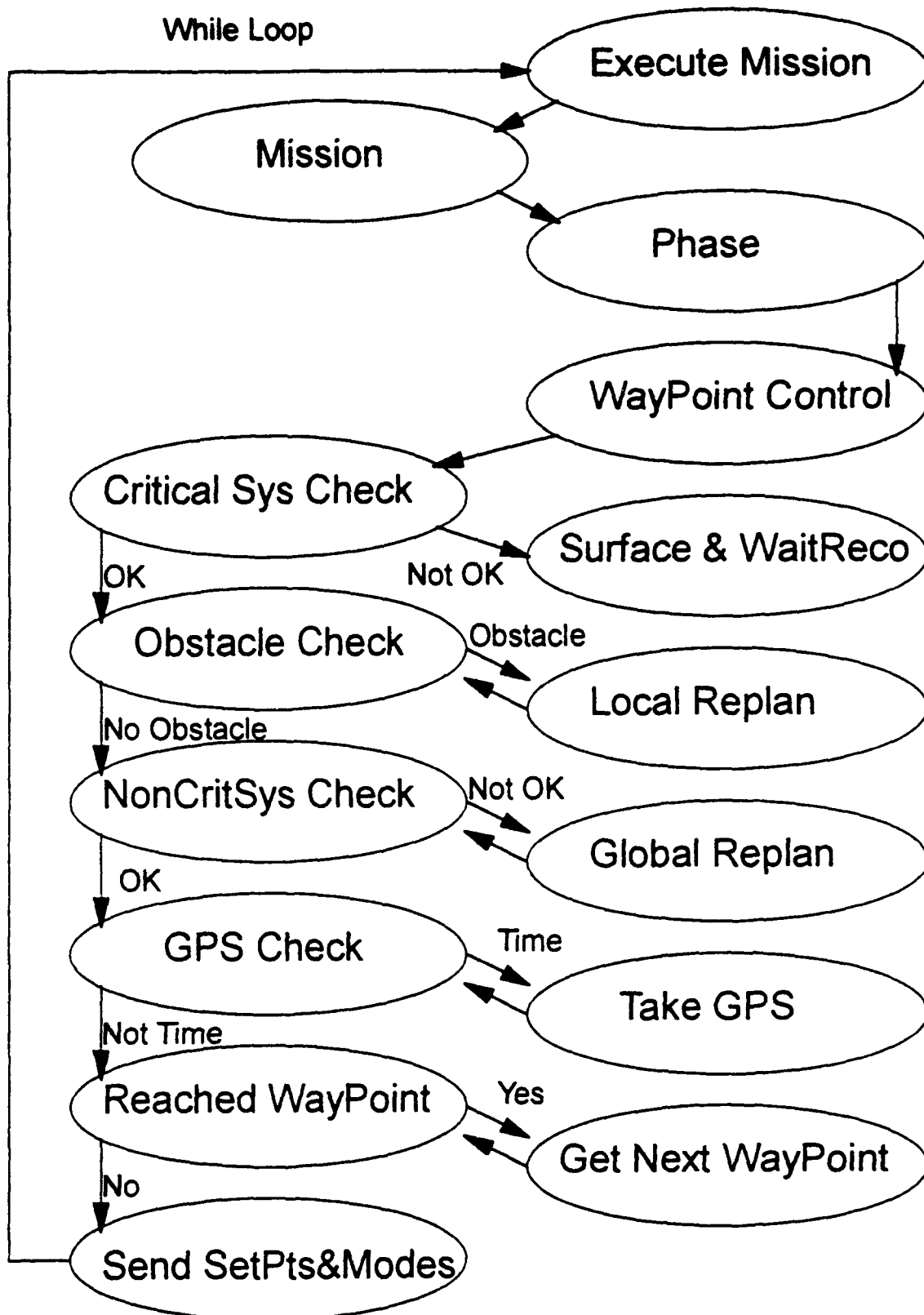


Figure 3.2 Mission Transition Diagram

been defined as a "surface" subroutine and is commanded to "wait for recovery". These alternatives are equivalent to the two OR rule statements for each phase in the PROLOG code.

In general, there are choices of three levels of system problems:

1. No problem
2. Reduced capacity problem
3. Critical system problem

which are the means to force either:

1. No action
2. Replan of mission phases
3. Mission abortion

The decisions are made by objects at the tactical level using system diagnostic techniques with a failure modes and effects analysis (FMEA).

The decision of which of the three levels of system problems to proceed with at the Strategic level is determined by a predetermined sequential order of subroutines. If a FALSE response is received upon calling a subroutine, the next step is attempted. This continues until the steps have been exhausted and the program returns to the while loop. The while loop continues until the mission is complete or it is aborted due to a failure as discussed above.

The "Mission" subroutine, for example, queries the Tactical level with the subroutine "InTransit". If the response, "AUVInTransit", is TRUE the Tactical level is commanded to conduct "Transit". Since no response is expected or required from "Transit" the program continues. The next

query is "IsTransitDone". At least the first time through "TransitComplete" will be FALSE, so the program continues. The different phases, "AUVIn...", are all initialized as TRUE so that the program enters them at least once. The "...Complete" are initialized as FALSE so that the program does not automatically complete the mission the first time through. The next series of queries should all result in a FALSE as well, when the respective subroutines are called. As a result, the while loop continues and starts again with the query is the "AUVInTransit".

The while loop is the primary difference between the rule based programming and structured programming. Whereas PROLOG simply backtracks until it reaches a cut, structured programming must complete the entire loop before it comes back to the same query. This is not as big a problem as it may sound since the only expected response is TRUE or FALSE (1 or 0) and therefore the cycle through the series of queries is virtually instantaneous. Although the repeat in a PROLOG code only goes back to the cut, it does not take much more real computational time to repeat the same question in Structured programming.

There is no storage of internal or external world variables at this level as in the PROLOG code. The lack of memory at the Strategic level is a key characteristic when the mission changes. This allows for easier modification to meet the requirements of a new mission which is one of the long

term goals for mission control software. The Tactical and Execution levels should not have to be changed in the RBM if the mission changes. These levels may have to change if the sensors change or the dynamics of the vehicle is altered, but this would require a change in the hardware of the AUV as well.

Many of the individual subroutines are called by multiple subroutines. Since all the subroutines are linked together, they can be called by any of the others, as long as they have the proper variable in the argument. Thus, it is imperative that commonality be used wherever possible. The AUV systems are checked at each of the individual phases. Therefore, the critical and non-critical system subroutines are called by all the phases. For example, the "NonCriticalSysCheck" subroutine checks if there are any new non-critical system problems. This subroutine is called by the subroutines "WayPCont", "SrchPatt", and "TaskPatt" and the argument "NewNonCriticalSysProb" is the same for all three. This reduces the lines of code drastically. A complete listing of all C subroutines is provided in Appendix A.

IV. VALIDATION FOR THE 'FLORIDA MISSION'

A. INTRODUCTION

The modified "Florida Mission" is a takeoff of the search and rescue mission originally planned for demonstrations off the Florida coast. The mission consists of a transit from the launching site, a search for mines, followed by the neutralization of the mines, and a return transit to a predetermined site. In order to validate the logic of the structured programming two scenarios were conducted, one to test the normal operation and the other to demonstrate the emergency procedure when a major fault is discovered.

The critical systems on the AUV are the power, propulsion, steering, and computer equipment. If any of these systems fail during operation, the AUV is sent into a *Surface* subroutine followed by a *WaitForRecovery* subroutine. The latter two subroutines also define the normal completion of the mission once the AUV has reached its end destination.

The non-critical systems of the AUV are buoyancy, diving, payload, sonar, thruster, and hull integrity (water leaks). If a new non-critical system failure occurs the AUV is sent into a *GlobalReplan* subroutine where the AUV loiters and conducts a global mission replan. This simply means that when one of these systems malfunctions certain phases of the

mission may be shortened or left out completely. It is important to note that the AUV takes action only if a new non-critical system failure occurs. Therefore, the AUV is not sent into a global replan every loop since action has been (or is being) taken on the failure already.

The data is sent to a mission log which records the desired information in order to maintain a running account of the mission status. Normally this would be recorded in a database, but for the purpose of the experiment in this thesis it was recorded in a data file. This method was used in order to demonstrate the proper execution of the mission itself. When the code is implemented on the actual AUV the queries would be made directly to the Tactical level and its database. For the purpose of testing the logic, the queries were made to the user, vice the Tactical level. The set points and modes are sent to the buffer each iterative loop to ensure that the execution level maintains a world state of the vehicle and its position for the autopilot.

B. NORMAL OPERATIONAL SCENARIO

The normal operational scenario consisted of the AUV completing all the phases of the mission without any internal critical equipment failure. A complete sample normal mission log can be found in Appendix B. Many of the steps in each phase were only shown once or twice since only the logic was being tested. In real time, the while loop may be entered

thousands of times before each phase is completed, but repeating the loop a couple times demonstrates that the subroutine logic will function properly. The C code for the first phase (Transit) of a sample mission is shown in Table III. The transit phase includes a new non-critical sonar system failure which requires a global replanning.

TABLE III SAMPLE MISSION LOG OF THE TRANSIT PHASE

Status of the systems follows: 0 is OOC,
1 is OK.

AUV is in transit stage.

Power Status : 0
Propulsion System : 0
Steering System : 0
Computer System : 0

There are no obstacles in the path.

Buoyancy System : 0
Diving System : 0
Payload Status : 0
Sonar System : 0
Thruster System : 0
Water Leak Status : 0

It is not time for a GPS fix.
AUV has not reached a waypoint yet.

The AUV is sending the set points and modes to the buffer.

AUV has not completed the transit stage.

AUV is in transit stage.

Power Status : 0
Propulsion System : 0
Steering System : 0
Computer System : 0

There is an obstacle in the path.
There is an unknown obstacle in the path.
The new obstacle has been logged.

The AUV has commenced loitering to permit time to complete the necessary steps.

The AUV has started the local replanning.

Buoyancy System : 0
Diving System : 0
Payload Status : 0
Sonar System : 1
Thruster System : 0
Water Leak Status : 0

There is a new non-critical system problem.

The AUV has commenced loitering to permit time to complete the necessary steps.

The AUV has started the global replanning.

It is not time for a GPS fix.
AUV has reached the next waypoint.
The AUV has been programmed for the next waypoint and is proceeding in that direction.

The AUV is sending the set points and modes to the buffer.

AUV has not completed the transit stage.

AUV is in transit stage.

Power Status : 0
Propulsion System : 0
Steering System : 0
Computer System : 0

There are no obstacles in the path.

Buoyancy System : 0
Diving System : 0
Payload Status : 0
Sonar System : 0
Thruster System : 0
Water Leak Status : 0

It is time for a GPS fix.
A GPS fix has been taken.

AUV has reached the next waypoint.
The AUV has been programmed for the next waypoint and is proceeding in that direction.

The AUV is sending the set points and modes to the buffer.

AUV has completed the transit stage.

C. VEHICLE TYPE FAILURE AND EMERGENCY RECOVERY

The critical system failure scenario was conducted to ensure that the AUV would take the necessary steps to ensure that the vehicle itself would not be lost if a major component malfunctioned. The scenario consisted of the vehicle originally operating with no equipment failures. Next a diving system failure occurred and a global replan ensued. The next time through the loop a steering system failure occurred which sent the AUV into the *Surface* and *WaitForRecovery* subroutines or essentially termination of the mission. The log of the sample failed mission is shown in Table IV.

TABLE IV SAMPLE OF A TERMINATED MISSION

Status of the systems follows: 0 is OOC,
1 is OK.

AUV is in transit stage.

Power Status : 0
Propulsion System : 0
Steering System : 0
Computer System : 0

There are no obstacles in the path.

Buoyancy System : 0
Diving System : 0
Payload Status : 0
Sonar System : 0
Thruster System : 0
Water Leak Status : 0

It is not time for a GPS fix.
AUV has not reached a waypoint yet.

The AUV is sending the set points and modes to the buffer.

AUV has not completed the transit stage.

AUV is in transit stage.

Power Status : 0
Propulsion System : 0
Steering System : 0
Computer System : 0

There is an obstacle in the path.
There is an unknown obstacle in the path.
The new obstacle has been logged.

The AUV has commenced loitering to permit time to complete the necessary steps.

The AUV has started the local replanning.

Buoyancy System : 0
Diving System : 0
Payload Status : 0
Sonar System : 0
Thruster System : 0
Water Leak Status : 0

There is a new non-critical system problem.

The AUV has commenced loitering to permit time to complete the necessary steps.

The AUV has started the global replanning.

It is not time for a GPS fix.
AUV has reached the next waypoint.
The AUV has been programmed for the next waypoint and is proceeding in that direction

The AUV is sending the set points and modes to the buffer.

AUV has completed the transit stage.

AUV is in the search stage.

The search time has elapsed, the search stage is over.

AUV is in the task stage.

Power Status : 0
Propulsion System : 0
Steering System : 1
Computer System : 0

There is a critical system problem.
Cannot complete programmed mission.

AUV is surfacing.

AUV is waiting for recovery.

V. CONCLUSIONS AND RECOMMENDATIONS

A. CONCLUSIONS

It is the contention of this thesis that the structured programming approach for a complex AUV mission is workable for mission control logic development. This contention has been shown to be true. However, PROLOG rule based code has one primary advantage. A single page of PROLOG code translates into one main program and 52 subroutines in C. Clearly, PROLOG represents a simpler version. Since the long term goal for the Strategic level was to make the code reconfigurable and relatively easy for the user to modify, the rule based programming should be the method of choice.

The single advantage of using C in the Strategic level is that no extra compiler or hardware support would be necessary to install in the AUV. The software is already required since the Execution level also uses C. Extra memory and a PROLOG processor would not be required. The only extra memory required in the structured programming is that some of the variables are initialized to ensure that the subroutines are entered at least once.

B. RECOMMENDATIONS

Structured programming can indeed be used for programming at the Strategic level. However, it lacks the overall

simplicity desired by the user to reprogram. Therefore, the structured programming approach should only be used as an effective backup should the rule based code or related software be unavailable for a real time embedded computer.

APPENDIX A

This appendix contains the entire structured programming code written in C including the main program (ExeAUVMi.C) and all the subroutines. They occur in alphabetical order.

Buoyancy.c

```
/* Subroutine to conduct diagnostics of the buoyancy system
   and determine if there is a new system problem. */

#include <stdio.h>

#define TRUE 1
#define FALSE 0

extern FILE *outfMission;

BuoyancySysDiagnostics(NewBuoyancyProbPtr)

int *NewBuoyancyProbPtr;

/* Main portion of subroutine */
{
    printf("Enter 1 if there is a new buoyancy problem, 0 if not.");
    scanf("%d",&*NewBuoyancyProbPtr);
    fprintf(outfMission,"Buoyancy System    : %d\n",*NewBuoyancyProbPtr);
}
```

Computer.c

```
/* Subroutine to conduct diagnostics of the computer system
   and verify that there are no computer system problems. */

#include <stdio.h>

extern FILE *outfMission;

ComputerSysDiagnostics (ComputerSysProblemPtr)

int *ComputerSysProblemPtr;

/* Main portion of subroutine */
{
    printf("Enter 1 if there is a computer problem, 0 if not.");
    scanf("%d", &*ComputerSysProblemPtr);
    fprintf(outfMission, "Computer System : %d\n", *ComputerSysProblemPtr);
}
```

CritSysC.c

```
/* Subroutine to conduct check of critical systems */

#include <stdio.h>

#define TRUE 1
#define FALSE 0

extern FILE *outfMission;

CriticalSysCheck(NoCriticalSysProblemPtr)

int *NoCriticalSysProblemPtr;

/* Main portion of subroutine */
{
    int PropulSysProblem, SteeringSysProblem, ComputerSysProblem;
    int PowerProblem;

    /* Initialize Status of Systems */
    PowerProblem = FALSE;
    PropulSysProblem = FALSE;
    SteeringSysProblem = FALSE;
    ComputerSysProblem = FALSE;

    /* Call subroutines to conduct system diagnostics and verify
       system is OK */
    PowerCheck(&PowerProblem);
    PropulSysDiagnostics(&PropulSysProblem);
    SteeringSysDiagnostics(&SteeringSysProblem);
    ComputerSysDiagnostics(&ComputerSysProblem);
    fprintf(outfMission, "\n");

    /* Determine if there is any critical system failure */
    if (PowerProblem || PropulSysProblem || SteeringSysProblem ||
        ComputerSysProblem)
    {
        *NoCriticalSysProblemPtr = FALSE;
        fprintf(outfMission, "There is a critical system problem.\n");
        printf("There is a critical system problem.\n");
    }
}
```

DivingSys.c

```
/* Subroutine to conduct diagnostics of the diving system
   and determine if there is a new system problem. */

#include <stdio.h>

#define TRUE 1
#define FALSE 0

extern FILE *outfMission;

DivingSysDiagnostics(NewDivingProbPtr)

int *NewDivingProbPtr;

/* Main portion of subroutine */
{
    printf("Enter 1 if there is a new diving problem, 0 if not.");
    scanf("%d",&*NewDivingProbPtr);
    fprintf(outfMission,"Diving System      : %d\n",*NewDivingProbPtr);
}
```

DropPackage.c

```
/* Subroutine to drop the package for the main task of the mission */
#include <stdio.h>

#define TRUE 1
#define FALSE 0

extern FILE *outfMission;

DropPackage(PackageDropDonePtr)

int *PackageDropDonePtr;

/* Main portion of subroutine */
{
    printf("Enter 1 if the package has been dropped, 0 if not.");
    scanf("%d",&*PackageDropDonePtr);
    if (*PackageDropDonePtr)
    {
        fprintf(outfMission,"AUV has dropped the package.\n\n");
    }
    else
    {
        fprintf(outfMission,"AUV has not dropped the package.\n\n");
    }
}
```

ExeAUVMi.c

```
/* Main program for execution of Florida mission */

#include <stdio.h>

#define TRUE 1
#define FALSE 0

FILE *outfMission;

main()
{
    int MissionComplete;

    /* Initialization */
    MissionComplete = FALSE;

    /* Open file to write the mission information */
    outfMission = fopen("Mission.log", "w");

    fprintf(outfMission, "Status of the systems follows: 0 is OOC, \n");
    fprintf(outfMission, "1 is OK. \n\n");

    /* Conduct while loop until mission is complete */
    while (!MissionComplete)
    {
        Mission(&MissionComplete);
    }
    if (MissionComplete)
    {
        fprintf(outfMission, "Mission completed successfully. \n");
        printf("Mission completed successfully. \n");
    }
    else
    {
        fprintf(outfMission, "Mission not completed! \n");
        printf("Mission not completed!! \n");
    }
}
```

FoundObs.c

```
/* Subroutine to check if there is an obstacle in the path. */
#include <stdio.h>

#define TRUE 1
#define FALSE 0

extern FILE *outfMission;

FoundObstacle(FoundObstacleAnsPtr)

int *FoundObstacleAnsPtr;

/* Main portion of subroutine */
{
    printf("Enter 1 if AUV has found an obstacle, 0 if not.");
    scanf("%d",&*FoundObstacleAnsPtr);
    if (*FoundObstacleAnsPtr)
    {
        fprintf(outfMission,"There is an obstacle in the path.\n\n");
    }
    else
    {
        fprintf(outfMission,"There are no obstacles in the path.\n\n");
    }
}
```

GetNxtWp.c

```
/* Subroutine to get the next waypoint */
#include <stdio.h>

extern FILE *outfMission;

GetNextWayPoint()

/* Main portion of subroutine */
{
    fprintf(outfMission, "The AUV has been programmed for the next ");
    fprintf(outfMission, "waypoint and is\n");
    fprintf(outfMission, "proceeding in that direction.\n\n");
    printf("The AUV has been programmed for the next waypoint");
    printf(" and is \n");
    printf("proceeding in that direction.\n");
}
```

GlobRepl.c

```
/* Subroutine to conduct global replan */
#include <stdio.h>
extern FILE *outfMission;
GlobalReplan()
/* Main portion of subroutine */
{
    /* Call subroutine to commence loiter to allow for global replan */
    Loiter();

    /* Call subroutine to commence global replan */
    StartGlobalReplanner();
}
```

GPSCheck.c

```
/* Subroutine to check if a GPS fix is necessary and
   take a GPS fix as required. */

#include <stdio.h>

#define TRUE 1
#define FALSE 0

extern FILE *outfMission;

GPSCheck()

/* Main portion of subroutine */
{
    int GPSFixTime;

    /* Initialization */
    GPSFixTime = FALSE;

    /* Subroutine to check if time for GPS fix */
    CheckIfTimeForGPS(&GPSFixTime);

    if (GPSFixTime)
    {
        TakeGPSFix();
    }
}
```

HomeDone.c

```
/* Subroutine to determine if AUV homing phase is completed */
#include <stdio.h>

#define TRUE 1
#define FALSE 0

extern FILE *outfMission;

IsHomingDone (HomingCompletePtr)

int *HomingCompletePtr;

/* Main portion of subroutine */
{
    printf("Enter 1 if the homing is done, 0 if not.");
    scanf("%d",&*HomingCompletePtr);
    if (*HomingCompletePtr)
    {
        fprintf(outfMission,"AUV has completed the homing phase.\n\n");
    }
    else
    {
        fprintf(outfMission,"AUV is still in the homing phase.\n\n");
    }
}
```

Homing.c

```
/* Subroutine to home in on the target */

#include <stdio.h>

#define TRUE 1
#define FALSE 0

extern FILE *outfMission;

Homing()

/* Main portion of subroutine */
{
    int WayPointControlComplete;
    int HaltMission; /* Dummy variable */

    /* Initialization */
    WayPointControlComplete = FALSE;

    WayPointControl(&WayPointControlComplete);

    if (WayPointControlComplete == FALSE)
    /* Unable to complete waypoint control.
       Call subroutine to surface and wait for recovery. */
    {
        fprintf(outfMission, "Cannot complete programmed mission.\n");
        printf("Cannot complete programmed mission.\n");
        Surface();
        WaitForRecovery();

        /* "HaltMission" puts a break in the simulation to imply that
           the mission has been terminated */
        scanf("%d", &HaltMission);
    }
}
```

InReturn.c

```
/* Subroutine to determine if AUV is in the "Return" stage */  
  
#include <stdio.h>  
  
#define TRUE 1  
#define FALSE 0  
  
extern FILE *outfMission;  
  
InReturn(AUVInReturnPtr)  
  
int *AUVInReturnPtr;  
  
/* Main portion of subroutine */  
{  
    printf("Enter 1 if the AUV is in the return stage, 0 if not.");  
    scanf("%d",&*AUVInReturnPtr);  
    if (*AUVInReturnPtr)  
    {  
        fprintf(outfMission,"AUV is in the return stage.\n\n");  
    }  
}
```

InSearch.c

```
/* Subroutine to determine if AUV is in search stage */
#include <stdio.h>

#define TRUE 1
#define FALSE 0

extern FILE *outfMission;

InSearch(AUVInSearchPtr)

int *AUVInSearchPtr;

/* Main portion of subroutine */
{
    printf("Enter 1 if the AUV is in the search mode, 0 if not.");
    scanf("%d",&*AUVInSearchPtr);
    if (*AUVInSearchPtr)
    {
        fprintf(outfMission,"AUV is in the search stage.\n\n");
    }
}
```

InTask.c

```
/* Subroutine to determine if AUV is in the "Task" stage */
#include <stdio.h>

#define TRUE 1
#define FALSE 0

extern FILE *outfMission;

InTask(AUVInTaskPtr)
int *AUVInTaskPtr;

/* Main portion of subroutine */
{
    printf("Enter 1 if the AUV is in the task stage, 0 if not.");
    scanf("%d",&*AUVInTaskPtr);
    if (*AUVInTaskPtr)
    {
        fprintf(outfMission,"AUV is in the task stage.\n\n");
    }
}
```

InTranst.c

```
/* Subroutine to determine if AUV is in transit stage or not */
#include <stdio.h>

#define TRUE 1
#define FALSE 0

extern FILE *outfMission;

InTransit(AUVInTransitPtr)

int *AUVInTransitPtr;

/* Main portion of subroutine */
{
    printf("Enter 1 if the AUV is in transit, 0 if not.");
    scanf("%d",&*AUVInTransitPtr);
    if (*AUVInTransitPtr)
    {
        fprintf(outfMission,"AUV is in transit stage.\n\n");
    }
}
```

IsItTrgt.c

```
/* Subroutine to determine if AUV has found a target */
#include <stdio.h>

#define TRUE 1
#define FALSE 0

extern FILE *outfMission;

IsItTarget(TargetPtr)

int *TargetPtr;

/* Main portion of subroutine */
{
    printf("Enter 1 if the AUV has found a target, 0 if not.");
    scanf("%d",&*TargetPtr);
    if (*TargetPtr)
    {
        fprintf(outfMission,"AUV has found a target.\n\n");
    }
    else
    {
        fprintf(outfMission,"The obstacle is not a target.\n\n");
    }
}
```

LocalRep.c

```
/* Subroutine to conduct local replan */
#include <stdio.h>
extern FILE *outfMission;
LocalReplan()
/* Main portion of subroutine */
{
    /* Call subroutine to commence loiter to allow for global replan */
    Loiter();

    /* Call subroutine to start the local replanner */
    StartLocalReplanner();
}
```

LogObst.c

```
/* Subroutine to log the unknown obstacle. */  
#include <stdio.h>  
extern FILE *outfMission;  
LogNewObstacle()  
/* Main portion of subroutine */  
{  
    fprintf(outfMission, "The new obstacle has been logged.\n\n");  
    printf("The new obstacle has been logged.\n");  
}
```

LogTrgt.c

```
/* Subroutine to log the target */
#include <stdio.h>
extern FILE *outfMission;
LogTarget()
/* Main portion of subroutine */
{
    fprintf(outfMission,"The target has been logged.\n\n");
    printf("The target has been logged.\n");
}
```

Loiter.c

```
/* Subroutine to commence loiter */  
  
#include <stdio.h>  
  
extern FILE *outfMission;  
  
Loiter()  
  
/* Main portion of subroutine */  
{  
    fprintf(outfMission, "The AUV has commenced loitering to permit ");  
    fprintf(outfMission, "time to complete\n");  
    fprintf(outfMission, "the necessary steps.\n\n");  
    printf("The AUV has commenced loitering to permit time to complete\n");  
    printf("the necessary steps.\n");  
}
```

Mission.c

```
/* Subroutine to conduct AUV mission */

#include <stdio.h>

#define TRUE 1
#define FALSE 0

extern FILE *outfMission;

Mission(MissionCompletePtr)

int *MissionCompletePtr;

/* Main portion of subroutine */
{
    int AUVInTransit, TransitComplete, SearchComplete, TaskComplete;
    int ReturnComplete, AUVInSearch, AUVInTask, AUVInReturn;
    int SearchTimeElapsed;

    /* Initialization */
    AUVInTransit      = TRUE;
    AUVInSearch       = TRUE;
    AUVInTask         = TRUE;
    AUVInReturn       = TRUE;
    TransitComplete   = FALSE;
    SearchComplete     = FALSE;
    TaskComplete      = FALSE;
    ReturnComplete    = FALSE;
    SearchTimeElapsed = FALSE;

    /* Call subroutine to find out if in transit */
    InTransit(&AUVInTransit);
    if (AUVInTransit)
    {
        Transit();
        IsTransitDone(&TransitComplete);
    }

    /* Call subroutine to find out if in search mode */
    InSearch(&AUVInSearch);
    if (AUVInSearch)
    {
        HasSearchTimeElapsed(&SearchTimeElapsed);
        /* If the search time has not elapsed */
        if (!SearchTimeElapsed)
        {
            Search();
            IsSearchDone(&SearchComplete);
        }
    }

    /* Call subroutine to find out if in task stage */
    InTask(&AUVInTask);
    if (AUVInTask)
    {
        Task();
        /* Note: Task is done when all the targets have
```

```

        had a package dropped on them */
        IsTaskDone(&TaskComplete);
    }

    /* Call subroutine to find out if in return stage */
    InReturn(&AUVInReturn);
    if (AUVInReturn)
    {
        Return();
        IsReturnDone(&ReturnComplete);

        /* If the return stage is complete, the mission is complete
           and the AUV can surface and wait for pick up. */
        if (ReturnComplete)
        {
            Surface();
            WaitForRecovery();
            *MissionCompletePtr = TRUE;
        }
    }
}

```

NonCritS.c

```
/* Subroutine to conduct check of non-critical systems */

#include <stdio.h>

#define TRUE 1
#define FALSE 0

extern FILE *outfMission;

NonCriticalSysCheck(NewNonCriticalSysProbPtr)

int *NewNonCriticalSysProbPtr;

/* Main portion of subroutine */
{
    int NewBuoyancyProb, NewDivingProb, NewPayloadProb, NewSonarProb;
    int NewThrusterProb, NewWaterLeakProb;

    /* Initialization */

    NewBuoyancyProb = FALSE;
    NewDivingProb = FALSE;
    NewPayloadProb = FALSE;
    NewSonarProb = FALSE;
    NewThrusterProb = FALSE;
    NewWaterLeakProb = FALSE;

    /* Call subroutines to conduct system diagnostics */
    BuoyancySysDiagnostics(&NewBuoyancyProb);
    DivingSysDiagnostics(&NewDivingProb);
    PayloadCheck(&NewPayloadProb);
    SonarSysDiagnostics(&NewSonarProb);
    ThrusterSysDiagnostics(&NewThrusterProb);
    WaterLeakCheck(&NewWaterLeakProb);

    fprintf(outfMission, "\n");
    if (NewBuoyancyProb || NewDivingProb || NewPayloadProb ||
        NewSonarProb || NewThrusterProb || NewWaterLeakProb)
    {
        *NewNonCriticalSysProbPtr = TRUE;
        fprintf(outfMission, "There is a new non-critical system ");
        fprintf(outfMission, "problem.\n\n");
        printf("There is a new non-critical system problem.\n");
    }
}
```

Obstacle.c

```
/* Subroutine to check for unknown obstacles */

#include <stdio.h>

#define TRUE 1
#define FALSE 0

extern FILE *outfMission;

ObstacleCheck(FoundNewObstaclePtr)

int *FoundNewObstaclePtr;

/* Main portion of subroutine */
{
    int FoundObstacleAns, UnknownObstacle;

    /* Initialization */
    FoundObstacleAns = FALSE;
    UnknownObstacle = FALSE;

    /* Call subroutine to determine if there is an object in the way */
    FoundObstacle(&FoundObstacleAns);

    if (FoundObstacleAns)
    {
        /* Call subroutine to check if it is an unknown obstacle */
        CheckIfUnknown(&UnknownObstacle);

        /* If it is an unknown obstacle, it needs to be logged */
        if (UnknownObstacle)
        {
            LogNewObstacle();
            *FoundNewObstaclePtr = TRUE;
        }
        else
        {
            fprintf(outfMission, "The obstacle has been previously ");
            fprintf(outfMission, "logged.\n\n");
            printf("The obstacle has been previously logged.\n");
        }
    }
}
```

PayloadC.c

```
/* Subroutine to conduct diagnostics of the payload
   and determine if there is a new payload problem. */

#include <stdio.h>

#define TRUE 1
#define FALSE 0

extern FILE *outfMission;

PayloadCheck(NewPayloadProbPtr)

int *NewPayloadProbPtr;

/* Main portion of subroutine */
{
    printf("Enter 1 if there is a new payload problem, 0 if not.");
    scanf("%d",&*NewPayloadProbPtr);
    fprintf(outfMission,"Payload Status      : %d\n",*NewPayloadProbPtr);
}
```

PowerChe.c

```
/* Subroutine to check if the power is OK */
#include <stdio.h>
extern FILE *outfMission;
PowerCheck(PowerProblemPtr)
int *PowerProblemPtr;
/* Main portion of subroutine */
{
    printf("Enter 1 if there is a power problem, 0 if not.");
    scanf("%d",&*PowerProblemPtr);
    fprintf(outfMission,"Power Status      : %d\n",*PowerProblemPtr);
}
```

PropulSy.c

```
/* Subroutine to conduct diagnostics of the propulsion system
   and verify that there are no propulsion system problems. */

#include <stdio.h>

extern FILE *outfMission;

PropulSysDiagnostics(PropulSysProblemPtr)

int *PropulSysProblemPtr;

/* Main portion of subroutine */
{
    printf("Enter 1 if there is a propulsion problem, 0 if not.");
    scanf("%d",&*PropulSysProblemPtr);
    fprintf(outfMission,"Propulsion System : %d\n",*PropulSysProblemPtr);
}
```

ReachdWP.c

```
/* Subroutine to check if the AUV has reached a waypoint */

#include <stdio.h>

#define TRUE 1
#define FALSE 0

extern FILE *outfMission;

ReachedWayPoint (ReachedWayPointAnsPtr)

int *ReachedWayPointAnsPtr;

/* Main portion of subroutine */
{
    printf("Enter 1 if the AUV has reached a waypoint, 0 if not.");
    scanf("%d",&*ReachedWayPointAnsPtr);
    if (*ReachedWayPointAnsPtr)
    {
        fprintf(outfMission,"AUV has reached the next waypoint.\n\n");
    }
    else
    {
        fprintf(outfMission,"AUV has not reached a waypoint yet.\n\n");
    }
}
```

Return.c

```
/* Subroutine to conduct AUV return stage */

#include <stdio.h>

#define TRUE 1
#define FALSE 0

extern FILE *outfMission;

Return()

/* Main portion of subroutine */
{
    int WayPointControlComplete;
    int HaltMission; /* Dummy variable */

    /* Initialization */
    WayPointControlComplete = FALSE;

    WayPointControl(&WayPointControlComplete);

    if (WayPointControlComplete == FALSE)
    /* Unable to complete the return.
    Call subroutine to surface and wait for recovery. */
    {
        fprintf(outfMission, "Cannot complete programmed mission.\n\n");
        printf("Cannot complete programmed mission.\n");
        Surface();
        WaitForRecovery();

        /* "HaltMission" puts a break in the simulation to imply that
        the mission has been terminated */
        scanf("%d", &HaltMission);
    }
}
```

RtrnDone.c

```
/* Subroutine to determine if AUV return stage is done */
#include <stdio.h>

#define TRUE 1
#define FALSE 0

extern FILE *outfMission;

IsReturnDone(ReturnCompletePtr)

int *ReturnCompletePtr;

/* Main portion of subroutine */
{
    printf("Enter 1 if the return is done, 0 if not.");
    scanf("%d",&*ReturnCompletePtr);
    if (*ReturnCompletePtr)
    {
        fprintf(outfMission,"AUV has completed the return stage.\n\n");
    }
    else
    {
        fprintf(outfMission,"AUV has not completed the return ");
        fprintf(outfMission,"stage.\n\n");
    }
}
```

Search.c

```
/* Subroutine to conduct AUV search stage */

#include <stdio.h>

#define TRUE 1
#define FALSE 0

extern FILE *outfMission;

Search()

/* Main portion of subroutine */
{
    int SearchPatternComplete;
    int HaltMission; /* Dummy variable */

    /* Initialization */
    SearchPatternComplete = FALSE;

    DoSearchPattern(&SearchPatternComplete);

    if (SearchPatternComplete == FALSE)
    /* Unable to complete search.
    Call subroutine to surface and wait for recovery. */
    {
        fprintf(outfMission, "Cannot complete programmed mission.\n\n");
        printf("Cannot complete programmed mission.\n");
        Surface();
        WaitForRecovery();

        /* "HaltMission" puts a break in the simulation to imply that
        the mission has been terminated */
        scanf("%d", &HaltMission);
    }
}
```

SendSPAM.c

```
/* Subroutine to send set points and modes */  
#include <stdio.h>  
  
extern FILE *outfMission;  
  
SendSetPointsAndModes()  
  
/* Main portion of subroutine */  
{  
    fprintf(outfMission, "The AUV is sending the set points and modes");  
    fprintf(outfMission, " to the buffer.\n\n");  
    printf("The AUV is sending the set points and modes to the ");  
    printf("buffer.\n");  
}
```

Sonar.c

```
/* Subroutine to conduct diagnostics of the sonar system
   and determine if there is a new system problem. */

#include <stdio.h>

#define TRUE 1
#define FALSE 0

extern FILE *outfMission;

SonarSysDiagnostics(NewSonarProbPtr)

int *NewSonarProbPtr;

/* Main portion of subroutine */
{
    printf("Enter 1 if there is a new sonar problem, 0 if not.");
    scanf("%d",&*NewSonarProbPtr);
    fprintf(outfMission,"Sonar System      : %d\n",*NewSonarProbPtr);
}
```

SrchDone.c

```
/* Subroutine to determine if AUV search stage is done */

#include <stdio.h>

#define TRUE 1
#define FALSE 0

extern FILE *outfMission;

IsSearchDone(SearchCompletePtr)

int *SearchCompletePtr;

/* Main portion of subroutine */
{
    printf("Enter 1 if the search is done, 0 if not.");
    scanf("%d",&*SearchCompletePtr);
    if (*SearchCompletePtr)
    {
        fprintf(outfMission,"AUV has completed the search stage.\n\n");
    }
    else
    {
        fprintf(outfMission,"AUV has not completed the search ");
        fprintf(outfMission,"stage.\n\n");
    }
}
```

SrchPatt.c

```

/* Subroutine to conduct waypoint control in search stage */

#include <stdio.h>

#define TRUE 1
#define FALSE 0

extern FILE *outfMission;

DoSearchPattern(SearchPatternCompletePtr)

int *SearchPatternCompletePtr;

/* Main portion of subroutine */
{
    int NoCriticalSysProblem,WayPStatusComplete,FoundNewObstacle;
    int NewNonCriticalSysProb,Target;
    /* Initialization */
    NoCriticalSysProblem = TRUE;
    WayPStatusComplete = FALSE;
    FoundNewObstacle = FALSE;
    NewNonCriticalSysProb = FALSE;
    Target = FALSE;

    /* Check if there is a critical system problem */
    CriticalSysCheck(&NoCriticalSysProblem);
    /* Note: if there is a critical system problem,
       SearchPatternCompletePtr remains FALSE and subroutine is done. */
    if (NoCriticalSysProblem)
    {
        /* Check if there are any new obstacles that need to be avoided */
        ObstacleCheck(&FoundNewObstacle);
        if (FoundNewObstacle)
        {
            IsItTarget(&Target);
            if (Target)
            {
                LogTarget();
            }
            else
            {
                LocalReplan();
            }
        }
        /* Check if there is a non-critical system problem */
        NonCriticalSysCheck(&NewNonCriticalSysProb);
        if (NewNonCriticalSysProb)
        {
            GlobalReplan();
        }
        /* Conduct check of the waypoint status */
        WayPointStatus();
        SendSetPointsAndModes();
        *SearchPatternCompletePtr = TRUE;
    }
}

```

SrchTEla.c

```
/* Subroutine to check if the search time has elapsed */

#include <stdio.h>

#define TRUE 1
#define FALSE 0

extern FILE *outfMission;

HasSearchTimeElapsed(SearchTimeElapsedPtr)

int *SearchTimeElapsedPtr;

/* Main portion of subroutine */
{
    printf("Enter 1 if the search time has elapsed, 0 if not.");
    scanf("%d",&*SearchTimeElapsedPtr);
    if (*SearchTimeElapsedPtr)
    {
        fprintf(outfMission,"The search time has elapsed, ");
        fprintf(outfMission,"the search stage is over.\n\n");
    }
    else
    {
        fprintf(outfMission,"The search time has not elapsed, ");
        fprintf(outfMission,"continuing the search.\n\n");
    }
}
```

Steering.c

```
/* Subroutine to conduct diagnostics of the steering system
   and verify that there are no steering system problems. */

#include <stdio.h>

extern FILE *outfMission;

SteeringSysDiagnostics(SteeringSysProblemPtr)

int *SteeringSysProblemPtr;

/* Main portion of subroutine */
{
    printf("Enter 1 if there is a steering problem, 0 if not.");
    scanf("%d",&*SteeringSysProblemPtr);
    fprintf(outfMission,"Steering System : %d\n",*SteeringSysProblemPtr);
}
```

StrtGRep.c

```
/* Subroutine to start the global replanner.  This subroutine is used
   when a non-critical system has failed. */

#include <stdio.h>

extern FILE *outfMission;

StartGlobalReplanner()

/* Main portion of subroutine */
{
    fprintf(outfMission, "The AUV has started the global replanning.\n\n");
    printf("The AUV has started the global replanning.\n");
    /* This is necessary when a non-critical system fails */
}
```

StrtLRep.c

```
/* Subroutine to start local replanner. This subroutine is used
   when an obstacle must be avoided. */

#include <stdio.h>

extern FILE *outfMission;

StartLocalReplanner()

/* Main portion of subroutine */
{
    fprintf(outfMission, "The AUV has started the local replanning.\n\n");
    printf("The AUV has started the local replanning.\n");
    /* This is necessary when an obstacle needs to be avoided */
}
```

Surface.c

```
/* Subroutine to surface */  
  
#include <stdio.h>  
  
#define TRUE 1  
#define FALSE 0  
  
extern FILE *outfMission;  
  
Surface()  
  
/* Main portion of subroutine */  
{  
    fprintf(outfMission, "AUV is surfacing.\n\n");  
    printf("AUV is surfacing.\n");  
}
```

TakeGPSF.c

```
/* Subroutine to take a GPS fix */  
#include <stdio.h>  
extern FILE *outfMission;  
TakeGPSFix()  
/* Main portion of subroutine */  
{  
    fprintf(outfMission, "A GPS fix has been taken.\n\n");  
    printf("A GPS fix has been taken.\n");  
}
```

Task.c

```
/* Subroutine to conduct AUV task stage */

#include <stdio.h>

#define TRUE 1
#define FALSE 0

extern FILE *outfMission;

Task()

/* Main portion of subroutine */
{
    int TaskPatternComplete;
    int HaltMission; /* Dummy variable */

    /* Initialization */
    TaskPatternComplete = FALSE;

    DoTaskPattern(&TaskPatternComplete);

    if (TaskPatternComplete == FALSE)
    /* Unable to complete task.
    Call subroutine to surface and wait for recovery. */
    {
        fprintf(outfMission, "Cannot complete programmed mission.\n\n");
        printf("Cannot complete programmed mission.\n");
        Surface();
        WaitForRecovery();

        /* "HaltMission" puts a break in the simulation to imply that
        the mission has been terminated */
        scanf("%d", &HaltMission);
    }
}
```

TaskDone.c

```
/* Subroutine to determine if AUV task stage is done */
#include <stdio.h>

#define TRUE 1
#define FALSE 0

extern FILE *outfMission;

IsTaskDone(TaskCompletePtr)

int *TaskCompletePtr;

/* Main portion of subroutine */
{
    printf("Enter 1 if the task is done, 0 if not.");
    scanf("%d",&*TaskCompletePtr);
    if (*TaskCompletePtr)
    {
        fprintf(outfMission,"AUV has completed the task stage.\n\n");
    }
    else
    {
        fprintf(outfMission,"AUV has not completed the task ");
        fprintf(outfMission,"stage.\n\n");
    }
}
```

TaskPatt.c

```
/* Subroutine to conduct waypoint control in task stage */

#include <stdio.h>

#define TRUE 1
#define FALSE 0

extern FILE *outfMission;

DoTaskPattern(TaskPatternCompletePtr)

int *TaskPatternCompletePtr;

/* Main portion of subroutine */
{
    int NoCriticalSysProblem,WayPStatusComplete,FoundNewObstacle;
    int NewNonCriticalSysProb;
    int HomingComplete,PackageDropDone;

    /* Initialization */
    NoCriticalSysProblem = TRUE;
    WayPStatusComplete = FALSE;
    FoundNewObstacle = FALSE;
    NewNonCriticalSysProb = FALSE;
    HomingComplete = FALSE;
    PackageDropDone = FALSE;

    /* Check if there is a critical system problem */
    CriticalSysCheck(&NoCriticalSysProblem);

    /* Note: if there is a critical system problem,
    TaskPatternCompletePtr remains FALSE and subroutine is done. */
    if (NoCriticalSysProblem)
    {
        /* Check if there are any new obstacles that need to be avoided */
        ObstacleCheck(&FoundNewObstacle);
        if (FoundNewObstacle)
        {
            LocalReplan();
        }

        /* Home in on target */
        Homing();

        /* Check if homing stage is complete */
        IsHomingDone(&HomingComplete);
        if (HomingComplete)
        {
            /* Conduct actual task */
            DropPackage(&PackageDropDone);
        }

        if (PackageDropDone)
        {
            TakeGPSFix();
        }
    }
}
```

```

/* Check if there is a non-critical system problem */
NonCriticalSysCheck(&NewNonCriticalSysProb);
if (NewNonCriticalSysProb)
{
    GlobalReplan();
}

/* Conduct check of the waypoint status */
WayPointStatus();
SendSetPointsAndModes();
*TaskPatternCompletePtr = TRUE;
}
}

```

Thruster.c

```
/* Subroutine to conduct diagnostics of the thruster system
   and determine if there is a new system problem. */

#include <stdio.h>

#define TRUE 1
#define FALSE 0

extern FILE *outfMission;

ThrusterSysDiagnostics(NewThrusterProbPtr)

int *NewThrusterProbPtr;

/* Main portion of subroutine */
{
    printf("Enter 1 if there is a new thruster problem, 0 if not.");
    scanf("%d",&*NewThrusterProbPtr);
    fprintf(outfMission,"Thruster System    : %d\n",*NewThrusterProbPtr);
}
```

TimeFGPS.c

```
/* Subroutine to check if it is time for "GPSfix" */
#include <stdio.h>

#define TRUE 1
#define FALSE 0

extern FILE *outfMission;

CheckIfTimeForGPS(GPSFixTimePtr)

int *GPSFixTimePtr;

/* Main portion of subroutine */
{
    printf("Enter 1 if it is time for a GPS fix, 0 if not.");
    scanf("%d",&*GPSFixTimePtr);
    if (*GPSFixTimePtr)
    {
        fprintf(outfMission,"It is time for a GPS fix.\n");
    }
    else
    {
        fprintf(outfMission,"It is not time for a GPS fix.\n");
    }
}
```

TranDone.c

```
/* Subroutine to determine if AUV transit stage is done */
#include <stdio.h>

#define TRUE 1
#define FALSE 0

extern FILE *outfMission;

IsTransitDone(TransitCompletePtr)

int *TransitCompletePtr;

/* Main portion of subroutine */
{
    printf("Enter 1 if the transit stage is complete, 0 if not.");
    scanf("%d",&*TransitCompletePtr);
    if (*TransitCompletePtr)
    {
        fprintf(outfMission,"AUV has completed the transit stage.\n\n");
    }
    else
    {
        fprintf(outfMission,"AUV has not completed the transit ");
        fprintf(outfMission,"stage.\n\n");
    }
}
```

Transit.c

```
/* Subroutine to conduct AUV transit */

#include <stdio.h>

#define TRUE 1
#define FALSE 0

extern FILE *outfMission;

Transit()

/* Main portion of subroutine */
{
    int WayPointControlComplete;
    int HaltMission; /* Dummy variable */

    /* Initialization */
    WayPointControlComplete = FALSE;

    WayPointControl(&WayPointControlComplete);

    if (WayPointControlComplete == FALSE)
    /* Unable to complete waypoint control.
       Call subroutine to surface and wait for recovery. */
    {
        fprintf(outfMission, "Cannot complete programmed mission.\n\n");
        printf("Cannot complete programmed mission.\n");
        Surface();
        WaitForRecovery();

        /* "HaltMission" puts a break in the simulation to imply that
           the mission has been terminated */
        scanf("%d", &HaltMission);
    }
}
```

UnKnwnOb.c

```
/* Subroutine to check if the AUV has encountered a new obstacle. */
#include <stdio.h>

#define TRUE 1
#define FALSE 0

extern FILE *outfMission;

CheckIfUnknown(UnknownObstaclePtr)

int *UnknownObstaclePtr;

/* Main portion of subroutine */
{
    printf("Enter 1 if there is an unknown obstacle, 0 if not.");
    scanf("%d",&*UnknownObstaclePtr);
    if (*UnknownObstaclePtr)
    {
        fprintf(outfMission,"There is an unknown obstacle in the ");
        fprintf(outfMission,"path.\n");
        printf(outfMission,"There is an unknown obstacle in the path.\n");
    }
}
```

WaitReco.c

```
/* Subroutine to wait for recovery */  
#include <stdio.h>  
#define TRUE 1  
#define FALSE 0  
  
extern FILE *outfMission;  
  
WaitForRecovery()  
  
/* Main portion of subroutine */  
{  
    fprintf(outfMission, "AUV is waiting for recovery.\n");  
    printf("AUV is waiting for recovery.\n");  
}
```

WaterLea.c

```
/* Subroutine to conduct determine if there is a new water leak. */
#include <stdio.h>

#define TRUE 1
#define FALSE 0

extern FILE *outfMission;

WaterLeakCheck(NewWaterLeakProbPtr)

int *NewWaterLeakProbPtr;

/* Main portion of subroutine */
{
    printf("Enter 1 if there is a new water leak, 0 if not.");
    scanf("%d",&*NewWaterLeakProbPtr);
    fprintf(outfMission,"Water Leak Status : %d\n",*NewWaterLeakProbPtr);
}
```

WayPCont.c

```
/* Subroutine to conduct waypoint control */

#include <stdio.h>

#define TRUE 1
#define FALSE 0

extern FILE *outfMission;

WayPointControl(WayPointControlCompletePtr)

int *WayPointControlCompletePtr;

/* Main portion of subroutine */
{
    int NoCriticalSysProblem, WayPStatusComplete, FoundNewObstacle;
    int NewNonCriticalSysProb;

    /* Initialization */
    NoCriticalSysProblem = TRUE;
    WayPStatusComplete = FALSE;
    FoundNewObstacle = FALSE;
    NewNonCriticalSysProb = FALSE;

    /* Check if there is a critical system problem */
    CriticalSysCheck(&NoCriticalSysProblem);

    /* Note: if there is a critical system problem,
       WayPointControlCompletePtr remains FALSE and subroutine is done. */
    if (NoCriticalSysProblem)
    {
        /* Check if there are any new obstacles that need to be avoided */
        ObstacleCheck(&FoundNewObstacle);
        if (FoundNewObstacle)
        {
            LocalReplan();
        }

        /* Check if there is a non-critical system problem */
        NonCriticalSysCheck(&NewNonCriticalSysProb);
        if (NewNonCriticalSysProb)
        {
            GlobalReplan();
        }

        /* Conduct check of the waypoint status */
        WayPointStatus();
        SendSetPointsAndModes();
        *WayPointControlCompletePtr = TRUE;
    }
}
```

WayPStat.c

```
/* Subroutine to get the status of the AUV wrt the waypoints */

#include <stdio.h>

#define TRUE 1
#define FALSE 0

extern FILE *outfMission;

WayPointStatus()

/* Main portion of subroutine */
{
    int ReachedWayPointAns;

    /* Initialization */
    ReachedWayPointAns = FALSE;

    /* Subroutine to take GPS fix when necessary */
    GPScheck();

    /* Subroutine to determine if the AUV has reached the next waypoint */
    ReachedWayPoint(&ReachedWayPointAns);
    if (ReachedWayPointAns)
    /* Subroutine to get the next waypoint */
    {
        GetNextWayPoint();
    }
}
```

APPENDIX B

Example of a Normal Mission Log:

Status of the systems follows: 0 is OOC,
1 is OK.

AUV is in transit stage.

Power Status : 0
Propulsion System : 0
Steering System : 0
Computer System : 0

There are no obstacles in the path.

Buoyancy System : 0
Diving System : 0
Payload Status : 0
Sonar System : 0
Thruster System : 0
Water Leak Status : 0

It is not time for a GPS fix.
AUV has not reached a waypoint yet.

The AUV is sending the set points and modes to the buffer.

AUV has not completed the transit stage.

AUV is in transit stage.

Power Status : 0
Propulsion System : 0
Steering System : 0
Computer System : 0

There is an obstacle in the path.

There is an unknown obstacle in the path.
The new obstacle has been logged.

The AUV has commenced loitering to permit time to complete the necessary steps.

The AUV has started the local replanning.

Buoyancy System : 0
Diving System : 0
Payload Status : 0
Sonar System : 1
Thruster System : 0
Water Leak Status : 0

There is a new non-critical system problem.

The AUV has commenced loitering to permit time to complete

the necessary steps.

The AUV has started the global replanning.

It is not time for a GPS fix.

AUV has reached the next waypoint.

The AUV has been programmed for the next waypoint and is proceeding in that direction.

The AUV is sending the set points and modes to the buffer.

AUV has not completed the transit stage.

AUV is in transit stage.

Power Status : 0
Propulsion System : 0
Steering System : 0
Computer System : 0

There are no obstacles in the path.

Buoyancy System : 0
Diving System : 0
Payload Status : 0
Sonar System : 0
Thruster System : 0
Water Leak Status : 0

It is time for a GPS fix.

A GPS fix has been taken.

AUV has reached the next waypoint.

The AUV has been programmed for the next waypoint and is proceeding in that direction.

The AUV is sending the set points and modes to the buffer.

AUV has completed the transit stage.

AUV is in the search stage.

The search time has not elapsed, continuing the search.

Power Status : 0
Propulsion System : 0
Steering System : 0
Computer System : 0

There is an obstacle in the path.

There is an unknown obstacle in the path.
The new obstacle has been logged.

AUV has found a target.

The target has been logged.

Buoyancy System : 0

Diving System : 0
Payload Status : 1
Sonar System : 0
Thruster System : 0
Water Leak Status : 0

There is a new non-critical system problem.

The AUV has commenced loitering to permit time to complete the necessary steps.

The AUV has started the global replanning.

It is not time for a GPS fix.
AUV has reached the next waypoint.

The AUV has been programmed for the next waypoint and is proceeding in that direction.

The AUV is sending the set points and modes to the buffer.

AUV has not completed the search stage.

AUV is in the search stage.

The search time has not elapsed, continuing the search.

Power Status : 0
Propulsion System : 0
Steering System : 0
Computer System : 0

There is an obstacle in the path.

There is an unknown obstacle in the path.
The new obstacle has been logged.

The obstacle is not a target.

The AUV has commenced loitering to permit time to complete the necessary steps.

The AUV has started the local replanning.

Buoyancy System : 0
Diving System : 0
Payload Status : 0
Sonar System : 0
Thruster System : 0
Water Leak Status : 0

It is time for a GPS fix.
A GPS fix has been taken.

AUV has reached the next waypoint.

The AUV has been programmed for the next waypoint and is proceeding in that direction.

The AUV is sending the set points and modes to the buffer.

AUV has completed the search stage.

AUV is in the task stage.

Power Status : 0
Propulsion System : 0
Steering System : 0
Computer System : 0

There are no obstacles in the path.

Power Status : 0
Propulsion System : 0
Steering System : 0
Computer System : 0

There is an obstacle in the path.

There is an unknown obstacle in the path.
The new obstacle has been logged.

The AUV has commenced loitering to permit time to complete the necessary steps.

The AUV has started the local replanning.

Buoyancy System : 0
Diving System : 0
Payload Status : 0
Sonar System : 0
Thruster System : 0
Water Leak Status : 0

It is not time for a GPS fix.
AUV has reached the next waypoint.

The AUV has been programmed for the next waypoint and is proceeding in that direction.

The AUV is sending the set points and modes to the buffer.

AUV is still in the homing phase.

Buoyancy System : 0
Diving System : 0
Payload Status : 0
Sonar System : 0
Thruster System : 0
Water Leak Status : 0

It is not time for a GPS fix.
AUV has reached the next waypoint.

The AUV has been programmed for the next waypoint and is proceeding in that direction.

The AUV is sending the set points and modes to the buffer.

AUV has not completed the task stage.

AUV is in the task stage.

Power Status : 0
Propulsion System : 0
Steering System : 0
Computer System : 0

There is an obstacle in the path.

The obstacle has been previously logged.

Power Status : 0
Propulsion System : 0
Steering System : 0
Computer System : 0

There is an obstacle in the path.

The obstacle has been previously logged.

Buoyancy System : 0
Diving System : 0
Payload Status : 0
Sonar System : 0
Thruster System : 0
Water Leak Status : 0

It is time for a GPS fix.
A GPS fix has been taken.

AUV has reached the next waypoint.

The AUV has been programmed for the next waypoint and is proceeding in that direction.

The AUV is sending the set points and modes to the buffer.

AUV has completed the homing phase.

AUV has dropped the package.

A GPS fix has been taken.

Buoyancy System : 0
Diving System : 0
Payload Status : 0
Sonar System : 0
Thruster System : 0
Water Leak Status : 0

It is not time for a GPS fix.
AUV has reached the next waypoint.

The AUV has been programmed for the next waypoint and is proceeding in that direction.

The AUV is sending the set points and modes to the buffer.

AUV has completed the task stage.

AUV is in the return stage.

Power Status : 0

Propulsion System : 0
Steering System : 0
Computer System : 0

There is an obstacle in the path.

There is an unknown obstacle in the path.
The new obstacle has been logged.

The AUV has commenced loitering to permit time to complete the necessary steps.

The AUV has started the local replanning.

Buoyancy System : 0
Diving System : 0
Payload Status : 0
Sonar System : 0
Thruster System : 0
Water Leak Status : 0

It is not time for a GPS fix.
AUV has not reached a waypoint yet.

The AUV is sending the set points and modes to the buffer.

AUV has not completed the return stage.

AUV is in the return stage.

Power Status : 0
Propulsion System : 0
Steering System : 0
Computer System : 0

There are no obstacles in the path.

Buoyancy System : 0
Diving System : 0
Payload Status : 0
Sonar System : 0
Thruster System : 0
Water Leak Status : 0

It is not time for a GPS fix.
AUV has reached the next waypoint.

The AUV has been programmed for the next waypoint and is proceeding in that direction.

The AUV is sending the set points and modes to the buffer.

AUV has completed the return stage.

AUV is surfacing.

AUV is waiting for recovery.
Mission completed successfully.

LIST OF REFERENCES

Bellingham, J.G., "State Configured Layered Control", paper presented at the International Advanced Robotics Programme, 1st Workshop on Mobile Robots for Subsea Environments, Monterey, CA, 23-26 October 1990).

Brooks, R.A., "A Robust Layered Control System For A Mobile Robot", *IEEE Journal of Robotics and Automation*, v.RA-2, pp.14-23, 1986.

Byrnes, R.B., *The Rational Behavior Model: A Multi-Paradigm, Tri-Level Software Architecture for the Control of Autonomous Vehicles*, Ph.D. Dissertation, Naval Postgraduate School, Monterey, CA, March 1993.

Byrnes, R.B., et. al., "Rational Behavior Model: An Implemented Tri-Level Multilingual Software Architecture For Control Of Autonomous Underwater Vehicles", paper to appear in Proceedings of 9th International Symposium on Unmanned, Untethered Submersible Technology, Durham, NH, 1993.

Jackson, P., *Introduction To Expert Systems*, 2d ed., Addison-Wesley, 1990.

Jensen, T.W., *Structured Programming*, IEEE-0018-9162/81/0300-0031, March 1981.

Kwak, S.H., and McGhee, R.B., *Rational Behavior Model: A Hierarchical Multiple Paradigm Architecture For Robot Vehicle Control Software*, Naval Postgraduate School, 1991.

Marcus, C., *Prolog Programming*, Addison-Wesley, 1986.

Sterling, L., and Shapiro, E., *The Art of Prolog*, The MIT Press, 1986.

Zheng, X., Jackson, E., and Kao, M., "Object-Oriented Software Architecture For Mission-Configureable Robots", paper presented at the International Advanced Robotics Programme, 1st Workshop on Mobile Robots for Subsea Environments, Monterey, CA, 23-26 October 1990).

INITIAL DISTRIBUTION LIST

	No. Copies
1. Defense Technical Information Center Cameron Station Alexandria VA 22304-6145	2
2. Library, Code 052 Naval Postgraduate School Monterey CA 93943-5002	2
3. Dr. A.J. Healey, Code ME/HY AUV Project Department of Mechanical Engineering Naval Postgraduate School Monterey, CA 93940	1
4. Dr. S. Kwak, Code CS/KW Department of Computer Science Naval Postgraduate School Monterey, CA 93940	1
5. Dr. R. McGhee, Code CS/MZ Department of Computer Science Naval Postgraduate School Monterey, CA 93940	1
6. NPS Naval Engineering Code 34 Monterey, CA 93943-5100	1
7. Lt. Richard Blank 3116 Waterside Lane Alexandria, VA 22309	1