A Modular Expandable Design for Mobile Robot Control Software

by

Ely C. Wilson

Submitted to the Department of Electrical Engineering and Computer Science
in Partial Fulfillment of the Requirements for the Degrees of

Bachelor of Science in Computer Science and Engineering

and

Master of Engineering in Electrical Engineering and Computer Science

at the

Massachusetts Institute of Technology

May 23, 1997

Author_____
Department of Electrical Engineering and Computer Science
May 23, 1997

Approved by_____
David S. Kang
Technical Supervisor, Draper Laboratory

Certified by_____
John J. Leonard
Thesis Supervisor

Accepted by_____
Arthur C. Smith
Chairman, Department Committee on Graduate Theses

A Modular Expandable Design for Mobile Robot Control Software

by
Ely C. Wilson

Submitted to the Department of Electrical Engineering and Computer Science
on May 23, 1997, in partial fulfillment of the
requirements for the Degrees of
Bachelor of Science in Computer Science and Engineering
and
Master of Engineering in Electrical Engineering and Computer Science

Abstract

This thesis is an investigation into the design and implementation of modular, expandable control software. Control software for mobile robots generally has three common objectives: navigate the robot through its environment, avoid unforeseen obstacles, and perform some task such as exploration, mapping, or data acquisition. Navigating through a real world environment while avoiding obstacles has proven to be a very difficult problem.

The thesis describes the design and implementation of the control software for a Basic UXO Gatherer (BUG) mobile robot. The design attempts to solve the problems associated with control software development by using a modularized, layered structure. The results of the BUG control software are examined to illustrate the strengths and weaknesses of this type of control architecture.

Technical Supervisor, Draper Laboratory: David S. Kang

Thesis Supervisor: John J. Leonard

Acknowledgments

Many thanks to all the members of the SMART team responsible for the BUG microrover, especially Anthony Lorusso and John Thele for putting in the extra time necessary to bring the SMART system into existence. I would like to recognize David Kang for making the entire project possible and John Leonard for all of his help with the writing of this thesis. I greatly appreciate the support given me by my family and friends. Most of all, I thank God, to whom I owe everything.

This thesis was prepared at The Charles Stark Draper Laboratory, Inc., under Contract 15834.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

This thesis is a case study in the development of control software for mobile robots. The contents of the thesis examine the design and implementation of the control software written for the BUG robotic vehicle.

The BUG vehicle and its software were designed and built at the Intelligent Unmanned Vehicle Center (IUVC) of C.S. Draper Laboratory. For more information about the IUVC, see Appendix A.

1.1 Goal of Thesis

The IUVC has produced control software for many different mobile robots. In the past, almost none of the software written has been reused on the next generation of robot. The control software for each robot performs essentially the same functions, so it should be possible to use the same software, with some modifications, for every robot.

Previous control software designs have not been successful in this respect because control software is often design with a particular application or mission in mind. The control software is developed with its purpose in mind and can be very difficult to adapt to other control software problems.

The goal of this thesis is to develop working control software which has the ability to adapt to changing hardware and different missions. The control software should be able to provide all the functionality necessary for the robot to accomplish its mission without sacrificing flexibility and modularity.

This will primarily be accomplished by designing software which is extremely modular and divided into strict sections. The power of the control software to perform many missions will be preserved by organizing the modules into a layered control structure based on Brooks' subsumption control architecture[3]. Unlike subsumption architectures, the control software will remain task-based in order to allow the robot to accomplish specific missions.

This design will be used as a case study to illustrate some of the approaches to solving the problems associated with control software design.

1.2 The EOD Project

This project was funded by Draper Laboratory to support Explosive Ordnance Disposal (EOD). The mission of this vehicle is to aid in the disposal of small unexploded ordnance (UXO).

The UXO our vehicle will handle are the small submunitions used by weapons such as grenade launchers and cluster bombs (Figure 1-1). After a conflict, the surrounding landscape is likely to be littered with unexploded submunitions.

1.2.1 The UXO Dilemma

The current method used by the military to clear UXO is a slow, expensive process which exposes personnel and equipment to considerable risk. Areas suspected of containing UXO are partitioned into sectors with their corners delimited by flags. In each sector, a four- to eight-man sweep team (Figure 1-2) visually scans the area for UXO. This manual sweep of the area is the safest part of the UXO clearing operation since removing or detonating the UXO involves actually handling the UXO.

Once the UXO in an area have been located, most branches of the military use explosive charges to detonate each of the UXO in place. The U.S. Marines may pick up the UXO and carry them to a central location for later detonation. This procedure is much less expensive than detonating the UXO in place, but is also far more dangerous. An unexploded submunition can detonate even if handled with extreme care.

1.2.2 Autonomous Agents

A potential solution to this problem is to have UXO disposal done by inexpensive autonomous robots. If one or more autonomous vehicles are able to clear an area of UXO at a reasonable rate, they would be a feasible alternative to using teams of EOD personnel.

The BUG project is a research effort to see how effective autonomous vehicles can be at locating and disposing of UXO.

1.3 Basic UXO Gatherer

The autonomous vehicle is referred to as a Basic UXO Gatherer (BUG). The goal for the project is to build a BUG which is small, inexpensive, and effective at its mission. Simply stated, the mission is to clear an area of UXO as quickly as possible.

Our strategy for accomplishing the mission is illustrated in the series of figures beginning on page 16. Figure 1-3 is an overhead view of a typical UXO site. The BUG is entering the area from the lower left corner. EOD personnel have spotted UXO at A and B. The BUG will attempt to retrieve UXOs A and B and deposit them at dump site X. All other objects on the map represent rocks, trees, or other obstacles.

A key element of our strategy is to establish "roads" on the map. Roads are areas which are believed to be free of obstacles. It should be safe for the BUG to traverse these areas at high speeds, allowing the BUG to accomplish its mission much more quickly [9]. In Figure 1-4, a number of roads have been designated by thick line segments.

The BUG will try to retrieve UXO A first. It can get halfway there using a nearby road, but it will have to make it the rest of the way through obstacle-filled territory (Figure 1-5). The BUG needs to maneuver its way through these obstacles in order to reach the UXO site.

Once the BUG reaches site A, it will have to perform an area search to determine the exact location of the UXO (Figure 1-6). While performing the search, the BUG may need to avoid additional obstacles. When the UXO is found, the BUG must pick it up and prepare to head for dump site X.

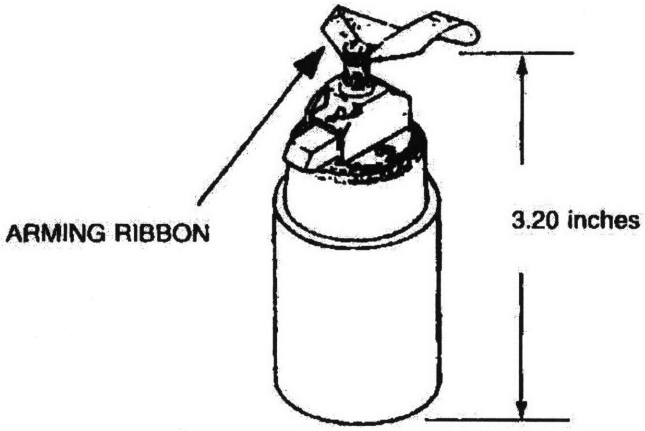

Figure 1-1: A Sample Submunition

Figure 1-2: A Sweep Team Searches for UXO

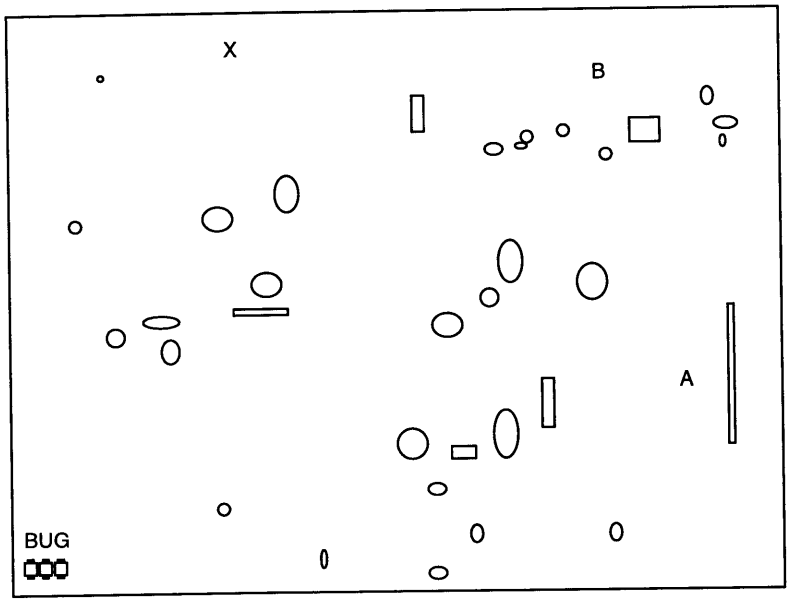

Figure 1-3: A Typical UXO Area

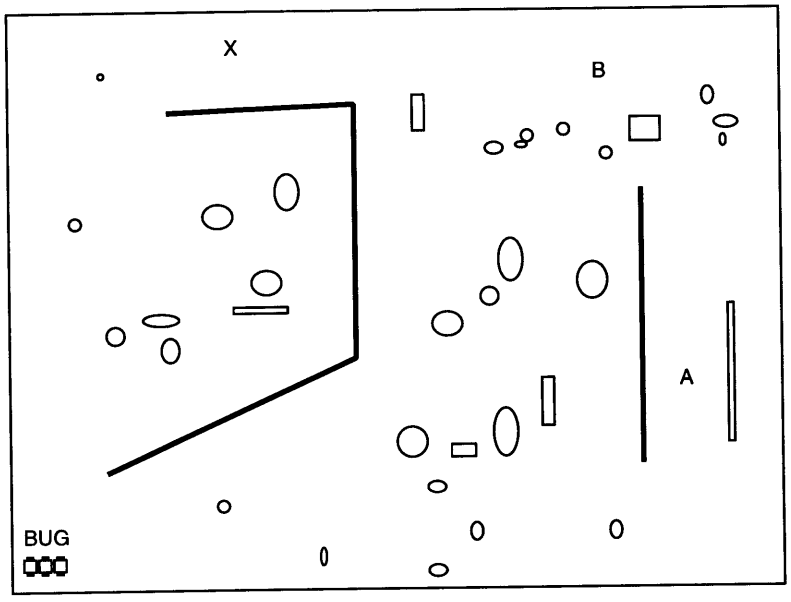

Figure 1-4: Obstacle-Free Roads

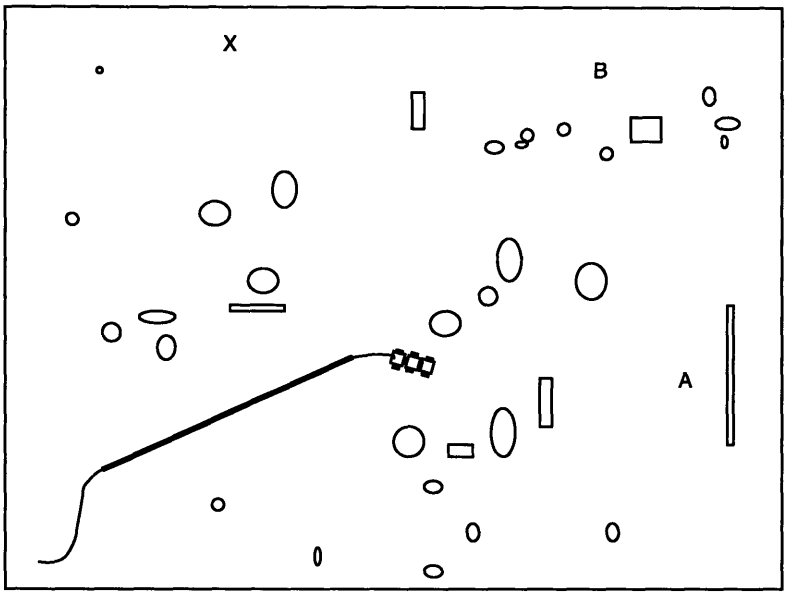

Figure 1-5: BUG Encounters Obstacles Enroute to A

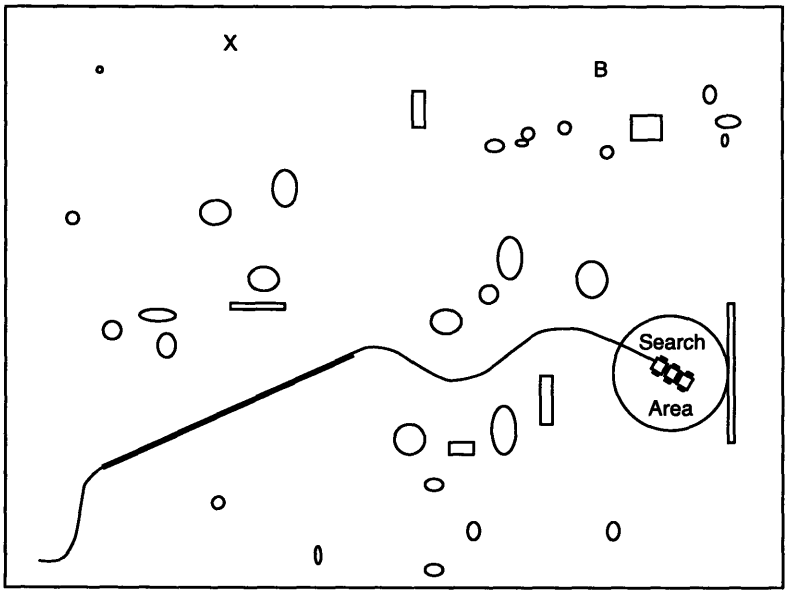

Figure 1-6: BUG Searching for UXO at Site A

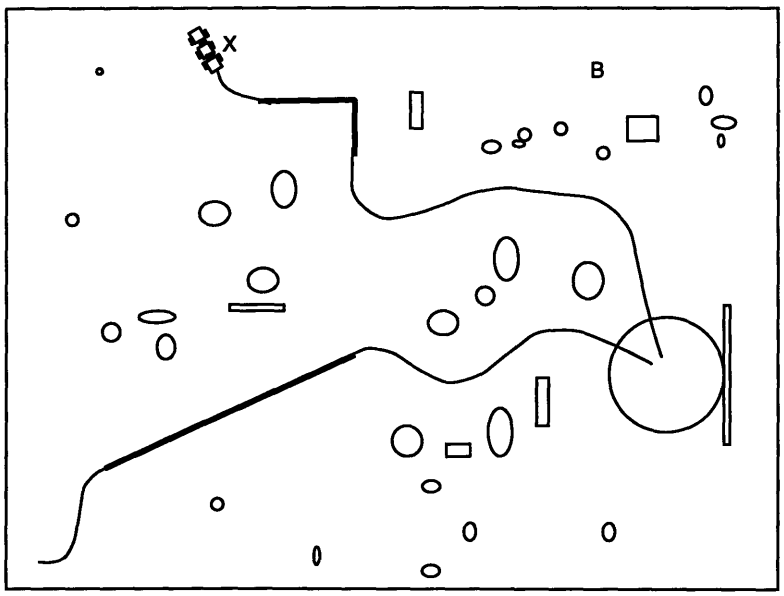

Figure 1-7: BUG Navigates to Dump Site

In Figure 1-7, the BUG has acquired the UXO and is making its way toward site X. Once again, the BUG will need to watch out for obstacles whenever it isn't on a road. After the BUG deposits the UXO at the dump site, it will repeat the whole process with UXO B.

In order to accomplish this mission, the BUG must be able to overcome all of the difficulties described above. The BUG was divided into three major components, each with its own set of associated design problems: mechanical, electrical, and software. The IUVC created three teams dedicated to the development and testing of the major components and two prototype BUGs were constructed. The prototypes, named EOD-1 and EOD-2, are pictured in Figure 1-8.

1.3.1 Mechanical Component

The BUG is required to perform its UXO disposal mission in a variety of terrains and environments, so the mechanical component of the vehicle must be well designed. The biggest design challenges are the development of a chassis which can conquer most terrain and the development of a system for picking up and carrying UXO.

Both BUG prototypes have been equipped with a six-wheeled, flexible chassis. This design allows the vehicle to overcome very large obstacles without difficulty. It is also capable of much higher speeds than tracked vehicle designs.

The BUG will pick up and carry UXO in a special grappler mechanism mounted on the front of the vehicle. The grappler is constructed of a large shovel for holding UXO and a movable rake for sweeping UXO into the mechanism. For more information about the mechanical hardware, see Section B.1 in the Appendix.

1.3.2 Electrical Component

Most of the electrical hardware is concerned with providing the vehicle with electrical power and sensors with which it can navigate and detect obstacles. For this purpose, the BUG is equipped with power regulation boards, rate gyroscopes, ultrasonic rangefinders, and a

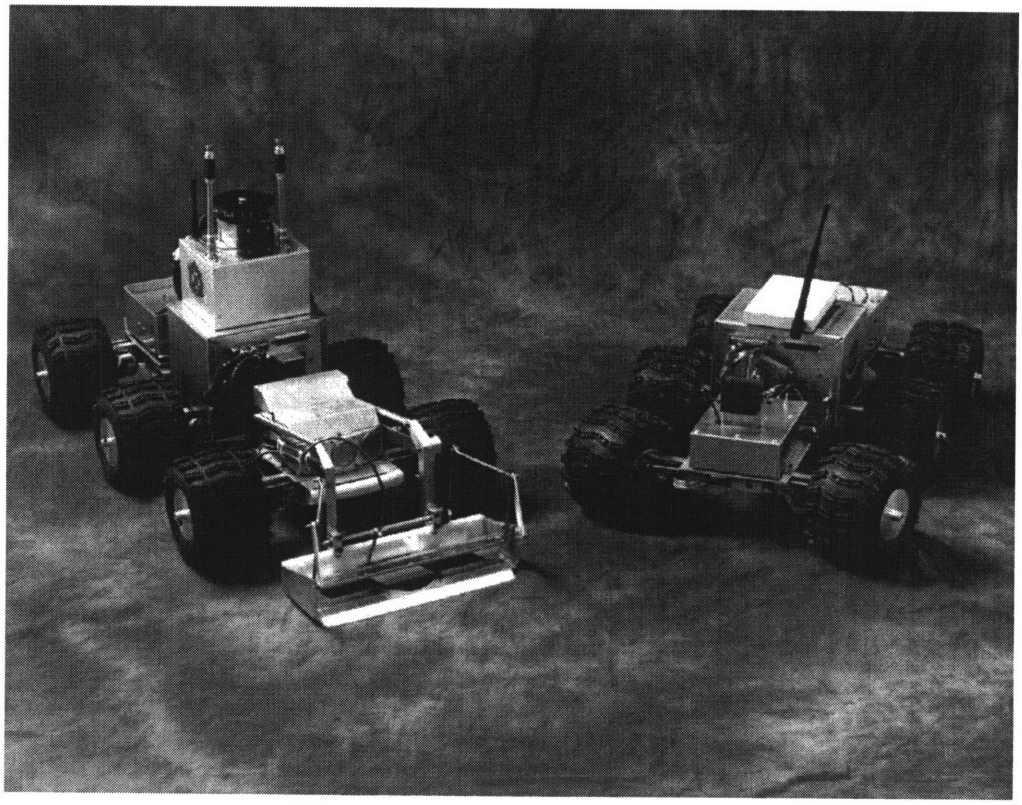

Figure 1-8: EOD-1 & EOD-2

metal detector. More information about the electrical hardware is contained in Section B.2.

Another vital piece of electrical hardware included on vehicle is an embedded processor. This processor will control all of the electrical hardware and will make vital decisions about navigation and obstacle avoidance. As the series of illustrations above show, the vehicle needs to be able to make these decisions to accomplish its mission.

1.3.3 Software Component

The software component of the vehicle is further divided into two separate systems (Figure 1-9). The control station will be located off the vehicle and will direct the BUG's mission under the supervision of a human operator. The control station will communicate the mission directives to the control software running on the BUG's embedded processor. The control software will then attempt to perform its mission autonomously.

This thesis is concerned with the design, development, and testing of the control software for the BUG vehicle. The control software needs to perform the mission described above, but should achieve the overall goal for the thesis as well. The thesis goal is a control software architecture which is flexible and reusable.

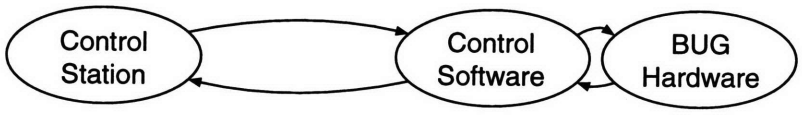

Figure 1-9: BUG Software Components

1.4 Thesis Road Map

The remainder of the thesis is divided into five chapters, which describe the control software problem, the BUG software design, the BUG software implementation, results, and conclusion.

Chapter 2 - The Control Software Problem This chapter is a discussion of how control software is usually designed and describes what issues are currently control software research problems.

Chapter 3 - BUG Software Design This chapter includes a list of goals for the control software and describes what design elements will allow the control software to meet those goals.

Chapter 4 - BUG Software Implementation This chapter shows how the design presented in chapter 3 was developed into a full implementation.

Chapter 5 - Results This chapter examines the software's operation and how well it performed in a series of tests outlined by the the Navy EOD division.

Chapter 6 - Conclusion This chapter summarizes the contents of the thesis and briefly describes which topics related to the BUG control software have the potential for further research.

Chapter 2

The Control Software Problem

Designing control software for mobile robots can be a difficult task. This chapter addresses the problems associated with control software design. The sections are:

- a description of the properties of control software for mobile robots
- a review of popular control software architectures

2.1 Mobile Robot Control Software

Mobile robot control software is the program which translates the operator's abstract commands into specific commands to the robot's hardware. Without control software, a mobile robot is a teleoperated tool at best. It is the control software which gives the robot its intelligence, however limited that intelligence may be.

Typically, the control software runs on an embedded processor which actually resides on the robot. This is because the software needs to be in close communication with the hardware of the robot. The hardware performs only at the direction of the control software, so it isn't efficient for the software to control the robot from a distance.

2.1.1 Responsibilities of Control Software

Control software always has many responsibilities related to the operation of the robot. On a mobile robot, the software will have to direct the vehicle's speed and heading. Autonomous mobile robots will probably have sensors on board for the purposes of determining the robot's location, detecting obstacles, and collecting any data vital to the robot's mission. The control software must operate these sensors and handle the data they provide properly.

The robot is likely to be operating under the supervision of some kind of control station. The control station may wish to send the robot new orders or it may just want to receive any data the robot collects. If a control station is present, the control software needs to be able to communicate with it effectively.

However the robot receives its orders, the control software needs to interpret its orders and carry them out. This is a mobile robot, so the orders will probably involve moving the robot to a new location. To accomplish this task, the control software needs to determine where it is currently located relative to its destination and go there without becoming lost or stuck.

The task of locating the robot relative to some other location and going there without getting lost is referred to here as *navigation*. Keeping the robot from getting stuck is called

obstacle avoidance because obstacles like rocks and bushes are usually responsible for this problem.

2.1.2 Control Software Design

The design of mobile robot control software has become a somewhat hot topic over the last few years as experts argue over which architectures and software designs are the most successful. The different architectures and designs which have been proposed over the years are supposed to help solve the harder problems of control software design.

Some of the software's responsibilities, like controlling the robot's hardware and gathering sensor data, don't present much of a problem to developers regardless of which architecture they use. Navigation and obstacle avoidance, on the other hand, can become frustratingly difficult tasks to perform.

Navigation is a difficult problem because most mobile robots lack a fool-proof method for determining their exact position. Inertial sensors and drag wheels can help the robot determine how it has moved relative to its last position, but the robot's new location will only be an estimation. Some attempts have been made to develop control software which navigates using external objects in the environment, but this kind of navigation is difficult for a computer to perform. The control software needs to be able to recognize an object which it has seen before and must not become confused if an object in its environment moves. All these problems remain largely unsolved despite continued research into robot navigation.

Obstacle avoidance is not necessarily as difficult as navigation, depending on the robot's environment. For example, obstacle avoidance is easy when the robot just has to dodge a small rock (Figure 2-1), but it can become much more difficult when the environment is filled with complex obstacles (Figure 2-2). In Figure 2-1, obstacle avoidance can be as simple as always turning left when sensors detect a rock. In Figure 2-2, the control software has to treat its environment like a life-size maze.

2.2 Control Software Architectures

Researchers and developers have produced many different software designs which make navigation and obstacle avoidance easier. In general, popular control software designs can be divided into one of three types: symbolic, subsumption, or hybrid. Symbolic architectures are the traditional method for dealing with control software problems. Subsumption and hybrid architectures are more recent approaches.

2.2.1 Traditional Symbolic Architecture

The symbolic architecture is the traditional choice for mobile robot control software. One of the earliest successes of the symbolic architecture was the software for Shakey the robot [4].

Shakey's software was based on a language of symbols. Different symbols represented rooms, hallways, and obstacles. Shakey was able to assign every object in its environment a unique symbol based upon its characteristics. A map was built out of these symbols which showed the relationships between all the objects.

The advantage of this approach to control software design is that Shakey was able to make logical conclusions based on the objects around it. Shakey could identify a room

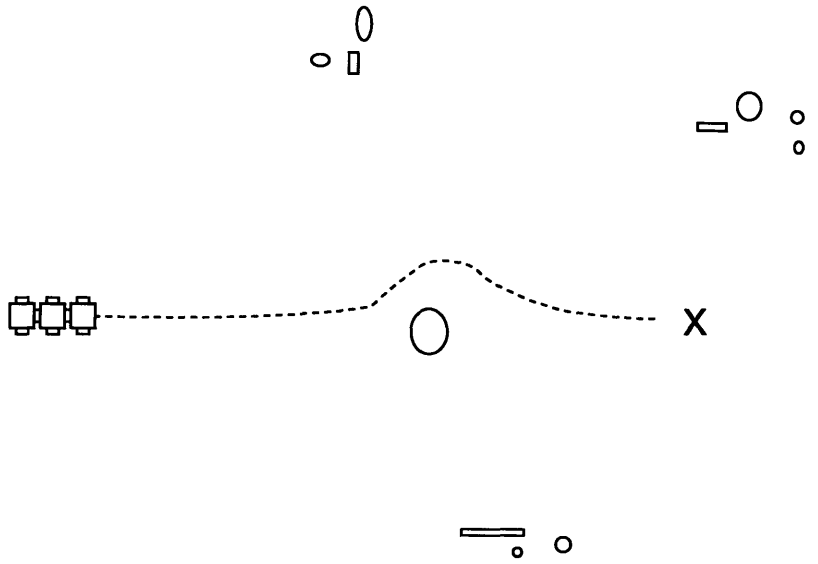

Figure 2-1: Easy Obstacle Avoidance Problem

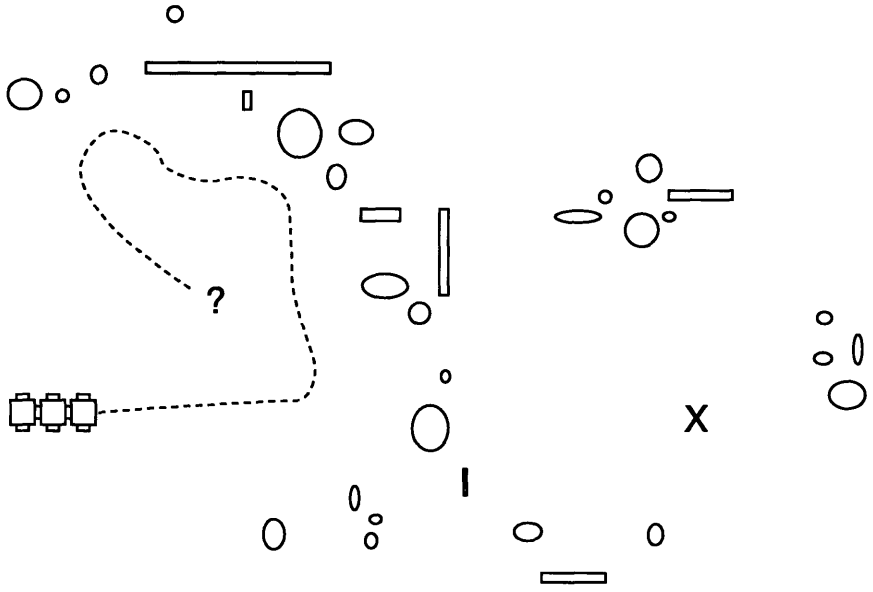

Figure 2-2: Difficult Obstacle Avoidance Problem

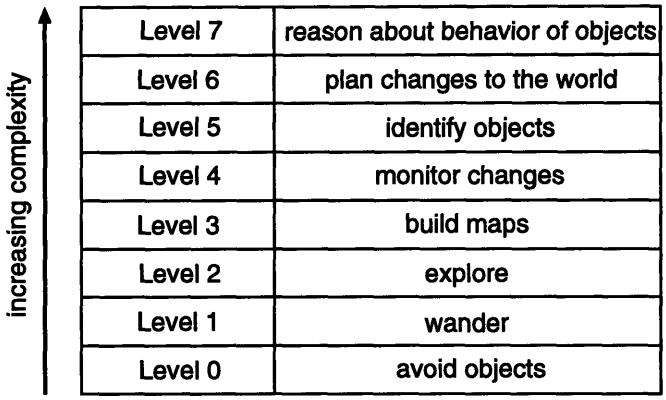

Figure 2-3: Levels of Competence

just by examining the objects inside. Furthermore, Shakey could tell when one object was moved because the rest of the objects were still in the correct location. Using its map, Shakey could describe where a particular object was and how one would get there.

Unfortunately, Shakey was only able to identify objects around it because of severe limitations placed on its environment. All walls were white and the floor had black stripes around the edges so that Shakey could easily identify all of the boundaries. All other objects were painted with different colors so that Shakey could identify them and how they were positioned. Shakey was able to accomplish quite a bit in its simple environment, but it would have been lost in the real world.

2.2.2 Subsumption Architecture

The subsumption architecture was developed in the mid-1980's as an alternative approach to control software design. Traditional symbolic approaches had placed mobile robots with very complex behaviors in very simple environments. Rodney Brooks noticed that evolutionary theory could suggest that complex behaviors in a complex environment are easier to attain once simple behaviors are established in a complex environment.

Thus was born a control architecture based upon simple building blocks which cooperate to form robust low-level behaviors. The behaviors are called robust because they handle real-world environments more reliably than some complex symbolic architectures. By adding more and more building blocks, one can create more and more complex behaviors, thus establishing *levels of competence* (Figure 2-3). Some scientists believe that by building up levels of competence, one will reach human-level intelligence [4].

Unfortunately, it is unclear how difficult it is to construct a subsumption architecture with a high level of competence. The control structure already begins to look very complex at Level 2 (Figure 2-4). So far, implementing the next level of behaviors has proved too difficult.

Subsumption architectures do have the advantages of being fast and robust, but one can argue that almost any architecture could demonstrate these qualities at the low level of intelligence which subsumption architectures have so far achieved. The element of the subsumption design which I find appealing is the layered control structure (Figure 2-5). In this structure behaviors build upon one another by allowing simple behaviors to operate with occasional interference from the high-level behaviors. This can be a flexible control structure because behaviors can be added to the stack without necessarily destroying the

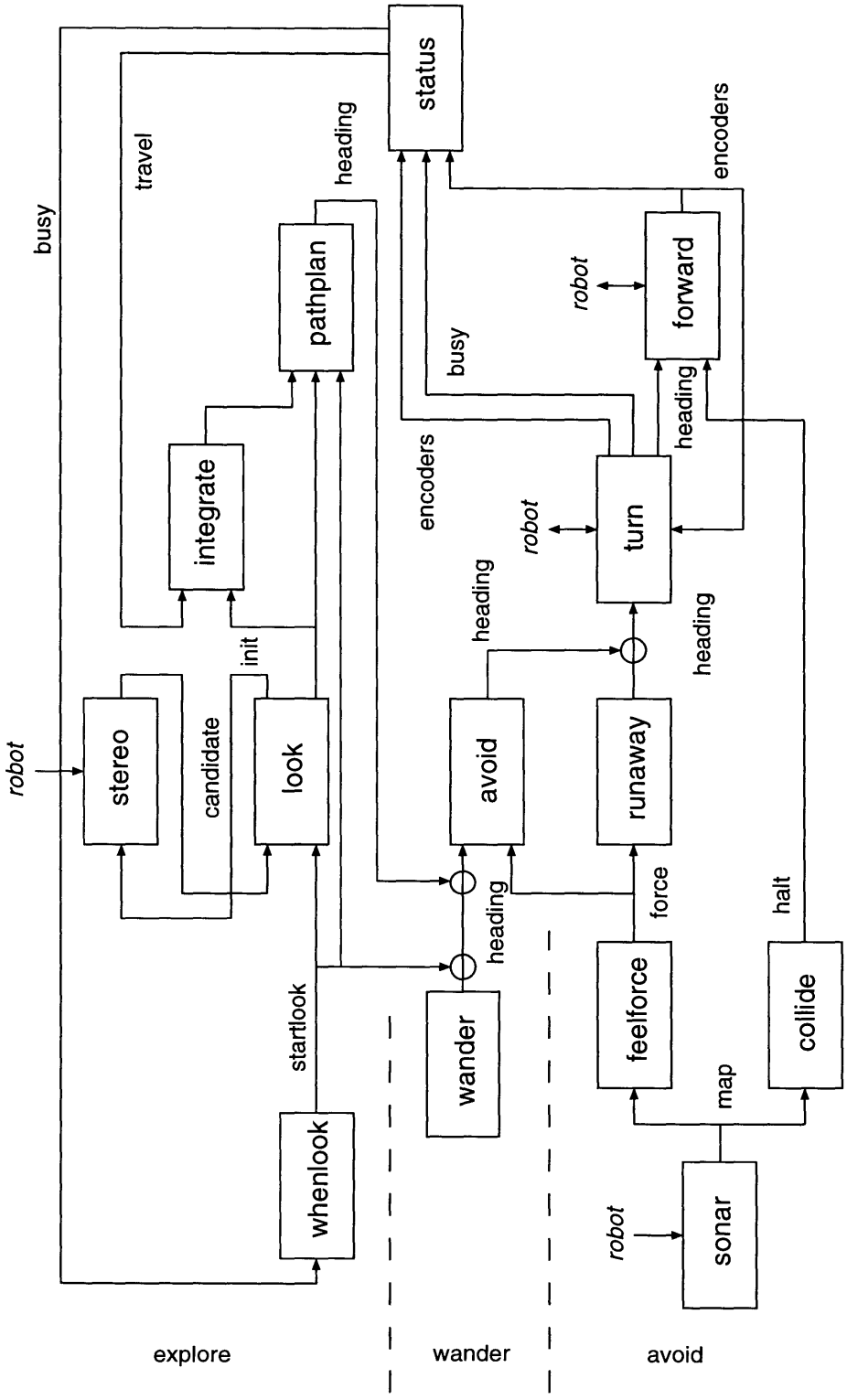

Figure 2-4: A Level 2 Subsumption Architecture [3]

old behaviors already in operation.

For a more detailed discussion of the advantages and disadvantages of a layered control architecture, see "Keeping Layered Control Simple" by James Bellingham [2].

2.2.3 SSS Hybrid Architecture

The popularity of symbolic and subsumption architectures led Connell to develop a hybrid architecture called SSS (Symbolic, Subsumption, and Servo). The SSS architecture combined symbolic control software with a subsumption architecture in the hopes that the hybrid system would benefit from the advantages of both architectures (Figure 2-6).

Complex, task-oriented behaviors were implemented in the symbolic portion of the control architecture. The symbolic portion would issue its commands to the vehicle via the subsumption layer. The subsumption layer, in cooperation with the servo hardware, controlled the behavior of the vehicle, avoiding obstacles if necessary [6].

I don't feel that the SSS hybrid took advantage of the real strengths of the subsumption architecture: flexibility and expandability. A properly designed symbolic architecture can do obstacle avoidance even better than current subsumption designs [11]; there is no need to include a subsumption layer for that purpose. A better hybrid of the symbolic and subsumption architectures is one which is symbolic in nature, but mimics the layered control structure.

2.3 Summary

The control software for a mobile robot has many functions, but navigation and obstacle avoidance are generally the most difficult to implement. Popular control software architectures can be divided into two types: symbolic and subsumption.

Symbolic architectures are most suited to solving the navigation problem, because they are able to build maps of the environment. Symbolic architectures are also better for implementing task-oriented behavior. However, many symbolic architectures are very limited by the environment in which they can operate.

Subsumption architectures are less complex and can accomplish some low-level navigation and obstacle avoidance behaviors. The big strength of the subsumption design is that it can operate in a more complex environment.

A successful control software design might be one which is symbolic in nature, but incorporates the layered control structure used in subsumption architectures. This design could prove to be powerful, flexible, and expandable.

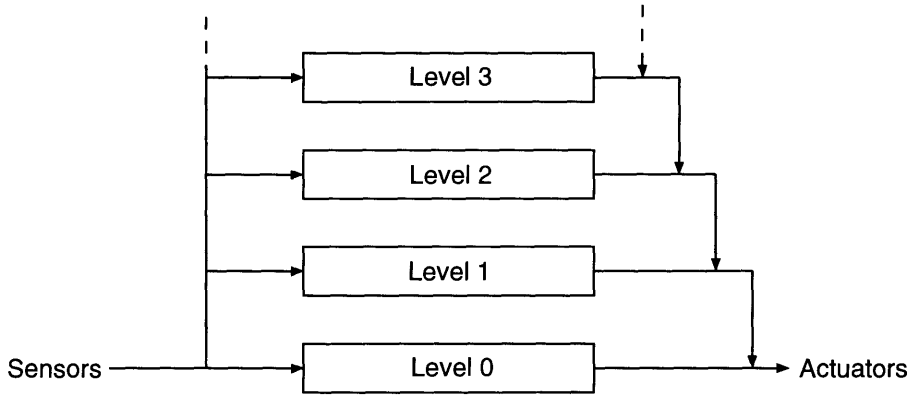

Figure 2-5: Layered Control Structure

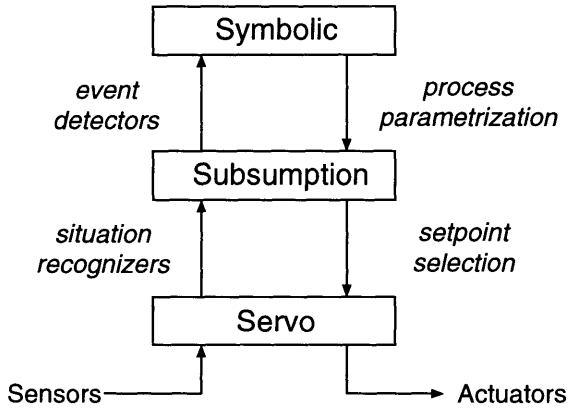

Figure 2-6: SSS Hybrid Architecture [6]

Chapter 3

BUG Software Design

This chapter presents the design of the BUG control software. The main sections of this chapter are:

- an overview of the total software system
- a specification of goals for the design
- a description of past designs which were considered
- the final elements of the control software design

The control software design must be capable of accomplishing the mission described in Section 1.2, in addition to meeting to thesis goals stated in Section 1.1. Specifically, the software must be flexible and expandable enough to adapt to new robots and missions.

3.1 Overall Software System Design

In order to understand the control software's mission, one must look at its role in the overall software system. The control software must regularly interact with the rest of the software and its role in relationship to the other software significantly affects how it must be designed.

The total software system can be viewed as four distinct pieces: user interface, control station, control software, and hardware drivers. These pieces interact to form the vehicle control system. Figure 3-1 is an illustration of how these four parts make up the whole.

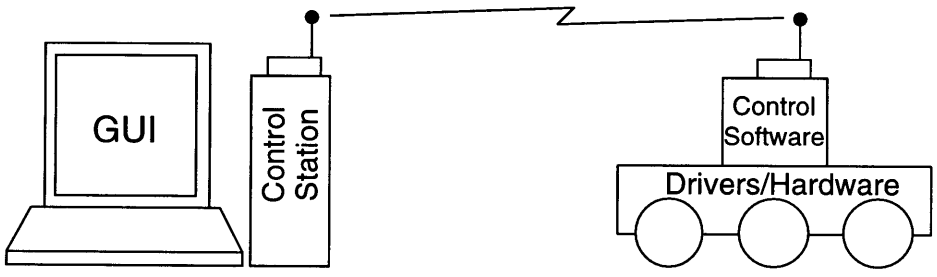

Figure 3-1: BUG Software System

This division of the software system into four pieces illustrates the role of each part of the software in completing the vehicle's mission. The sections which follow describe each piece's role in more detail.

3.1.1 User Interface

The Graphical User Interface (GUI) forms the link between the operator and the control station. The interface does not necessarily need to be graphical, but a GUI helps the operator visualize the vehicle's environment and makes the control station easier to operate.

3.1.2 Control Station

The control station operates behind the GUI in order to link the operator with the vehicle itself. Under the supervision and direction of the operator, the control station performs most of the high-level mission planning. When the control station has a task it would like the vehicle to perform, it communicates its desires to the vehicle via a radio link. The control station receives sensor data from the vehicle through this link and uses it to keep the operator updated with the vehicle's progress.

3.1.3 Control Software

The control software is responsible for taking the high-level directives issued by the control station and executing them on the vehicle. To accomplish this, the control software communicates with the hardware drivers to control the vehicle. The control software can also pass some of the sensor data it receives from the drivers to the control station.

3.1.4 Drivers

The hardware drivers make up the interface between the higher-level software and the vehicle's hardware. The drivers respond to commands from the control software to activate or deactivate hardware, control motors, and take sensor readings.

The drivers could be considered a part of the control software, but they are presented here as a separate system component in order to illustrate their role in the total software system. The control software component described above is designed to be hardware independent. The drivers must be hardware dependent in order to provide an interface between the software and the vehicle.

3.2 Goals for the Software

The first design consideration was to determine what characteristics are desirable in the control software. The characteristics are used as basic criteria which the design should meet. These characteristics can be divided into two categories: those required in order that the vehicle can perform its mission and those which are desirable for other reasons.

3.2.1 Mission-Required Characteristics

Mission-required characteristics directly support the vehicle's mission. The mission can be summarized as:

Find Position on Global Map	
Navigate Roads at High Speeds	
Communication With Control Station	Sensor Data
	New Orders
Semi-autonomous	
Performs Complex Tasks	

Table 3.1: Mission-Required Characteristics

- Proceed to the UXO area using "roads" which have been identified as being free of obstacles.
- Navigate to the general location of a UXO.
- Search the location for UXO.
- If a UXO is found, retrieve it using the vehicle's grappler mechanism.
- Proceed to a UXO dump site using obstacle-free "roads."

To accomplish this mission, the vehicle must be able to locate itself on the global map and navigate the "roads" at high speeds in order to perform its mission efficiently. The vehicle should be in close communication with the control station so that it can transmit sensor data and receive new orders. Finally, the control software must be able to perform complex tasks and should also be semi-autonomous, allowing a single person to oversee multiple vehicles.

3.2.2 Additional Desired Characteristics

There are a number of additional characteristics which are highly desirable in autonomous robot control software. A number of these arise from a desire for software which is easy to write and test. The remainder are the characteristics of software that adapts to changes in the hardware and has potential for expansion.

Making software design easy to implement and test is not easy, but there are a few elements of a design which can make a big difference. The software will be written by more

Ease of Writing And Testing	Separation into Modules
	Coding Standards
Expandability	New Sensors
	New Behaviors
	New Tasks
Potential for Increased Complexity	

Table 3.2: Desirable Characteristics

than one person, so coding standards can help make the integration of the portions written by different people easier. The software should also be separated into modules based on function. This provides natural divisions when writing the code and allows modules to be tested independently.

The control software could conceivably be expanded upon in three ways. One could want to add new sensors to the vehicle, new behaviors to the control software, or new tasks that the vehicle must perform. The addition of any of these items should be supported by the software's design and as easy to perform as possible.

3.3 Control Software Examples

Two previous control software designs were examined during the process of designing the BUG control software. The control software for the MITy-2 microrover [11] was examined because of the similarities between the MITy-2 microrover and the EOD vehicle. Subsumption control architectures were also examined because they claim to be more expandable than some other control software architectures.

By looking at these other designs, we hoped to determine which features were desirable and which features were undesirable. We wanted to take all the lessons learned from these two software architectures and apply them toward the BUG software design.

3.3.1 MITy-2 Architecture

The MITy-2 microrover was built for extraterrestrial exploration, but its mission was very similar to the UXO mission [12]. MITy-2 would navigate a preset course and use on-board hardware to perform a variety of tasks including manipulating rocks and taking scientific data.

The control software for MITy-2 consisted of a handful of tasks operating concurrently on the microrover's processor. These tasks handled such things as navigation, path following, and obstacle detection. The control software was very successful, but it was extremely difficult to understand or modify. The different sections of the software were not clearly divided into modules and those modules which did exist were not designed in a consistent manner.

From the MITy-2 control architecture, we learned that the software needs to be easier to read, more modular and consistent. This is reflected in the desirable characteristics (Table 3.2) chosen for the control software.

3.3.2 Subsumption Architecture

The subsumption control architecture was designed for autonomous mobile robots operating in a dynamic environment. An overview of the subsumption architecture is given in Section 2.2.2. This architecture was considered because in some ways it is easier to expand and adapt to mission changes than other control software architectures.

The main problem with the subsumption architecture is that it is difficult to implement any complex task-based behaviors. The BUG control software should have the potential for expansion and addition of new behaviors exhibited by the subsumption architecture without suffering the lack of complex behaviors.

The design feature that gives a subsumption architecture expandability is its organization into layers of competence. Each layer builds upon the previous layers to produce

a more complex behavior. Unfortunately, the subsumption design forces layers to interact with each other in a very limited fashion. Each layer is also built out of very primitive electronics, which makes it difficult to implement a complex behavior. A more flexible software implementation of this layered approach could allow layers to cooperate and would avoid the limitations of simplistic building blocks.

In order to achieve this goal, the BUG control software will be designed to imitate the layers of competency found in subsumption architectures without being restricted to the limitations of a true subsumption approach.

3.4 Control Software Design

This section specifies the elements of the control software design. This design attempts to take advantage of the strengths of the MITy-2 and subsumption architectures while avoiding the limitations of those designs. It will also include both the necessary and the desirable characteristics described in Section 3.2.

3.4.1 Task Orientation

The control software's entire purpose is centered around carrying out the orders of the operator and the control station. Therefore, the control software needs to be primarily task oriented. This is opposed to the typical subsumption architecture, which is primarily behavior oriented.

For the purposes of this design, a task is a single set of actions which the vehicle must perform. A task would be to navigate to a certain location or to search a given area for UXO. The vehicle's overall mission of clearing UXO is made up of many independent tasks which are determined by the control station. The control software is focused on completing these tasks because successful completion of the individual tasks implies successful completion of the overall mission.

3.4.2 Object-oriented Structure

One of the desired characteristics for the control software is that it be modular. In order to make this separation into modules consistent, the individual modules should be object-oriented in nature. This means that each module only communicates with the rest of the software through a set of publicly available functions. Other modules are not allowed access to the procedures and data contained in an object-oriented module without using the public functions.

The disadvantage of having object-oriented modules is that the modules must be slightly more complex. However, the advantages are that the modules are more self-contained, easier to test individually, and less susceptible to failure due to problems in other modules.

3.4.3 Event-based Architecture

One of the dangers of a flexible and expandable software design is that it may become very difficult to keep track of how the modules need to communicate with each other. When a new module is added to the software or a new task begins operating, it needs to be able to add itself to the communication network.

Our solution to this problem is the use of events and callbacks. Any module can create events which pertain to the function of that module. For example, the module which detects obstacles may wish to create an event which is linked to the detection of an obstacle. Callbacks are the function calls triggered by the occurrence of an event. Other modules can request that they be given a callback when a particular event occurs.

When a module or task wishes to receive data from a certain part of the control software, it can register for a callback with the associated event. As soon as that event occurs, all the callback functions for the event are called. This gives all the modules an opportunity to react to any event which occurs.

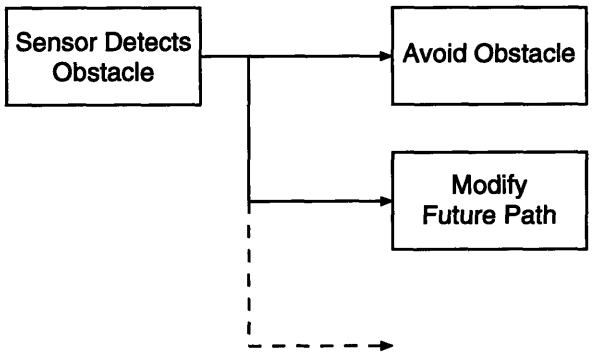

Figure 3-2: Multiple Event Callbacks

Figure 3-2 illustrates the flexibility of an event-based architecture. In this example, the event is the detection of an obstacle. Whenever this event occurs, two callback functions are called. One callback maneuvers the vehicle so that it avoids the obstacle. Another callback modifies the future path of the vehicle so that it will be sure to avoid the obstacle in the future. If other functions wish to be called when the event occurs, they can also register for a callback.

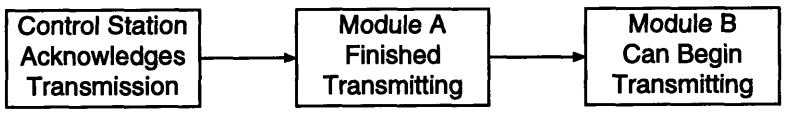

Figure 3-3: Cascading Events

Figure 3-3 shows how callbacks can be used to cascade multiple events. When the control station acknowledges the transmission of a communications packet, an event is triggered. Module A is notified via a callback that this event occurred. Since Module A has no more packets to transmit, it triggers its own event. Module B was waiting for this event to occur, so it had registered a callback with Module A's event. Now that Module A is done, Module B can begin transmitting.

The flexibility of having multiple callbacks for events and the power of using callbacks to cascade multiple events make an event-based architecture a valuable part of the software design.

3.4.4 Scheduler

The control software has multiple operations and behaviors which it is conducting at any given time. These behaviors include updating the vehicle's position on the global map,

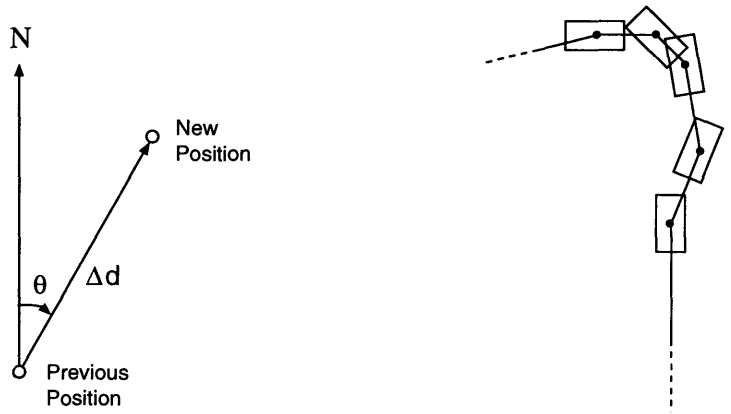

Figure 3-4: Dead Reckoning Navigation

operating sensors, and communicating with the control station. The software should be able to conduct these operations concurrently so that the vehicle does not have to stop moving to communicate with the control station or operate its sensors.

The control software needs to be able to run on many different computer systems, so there is not one single multitasking system which can be used. Instead, the software will use a scheduler to allow the different modules and behaviors to *cooperatively* multitask.

The scheduler gives other modules the ability to request *periodic* and *delayed* callbacks. These callbacks are similar to event callbacks because the modules can give up control of the vehicle and expect to be called back when the time is right. Instead of activating the callbacks when some event occurs, the scheduler waits for a specified amount of time to pass.

A *periodic* callback causes control to be returned to a module every time a specified period of time passes. Periodic callbacks are used when a module needs to periodically perform some function.

A *delayed* callback only returns control to the requesting module once, but the scheduler guarantees that a specified amount of time will pass before the callback occurs. This callback is typically used when a module wants to regain control later to observe the outcome of a single action.

Using these two types of callbacks, the scheduler enables the multiple modules to operate many behaviors concurrently. Because the scheduler only provides cooperative multitasking, the modules need to give up control of the vehicle as quickly as possible so that other modules can take control.

3.4.5 Navigation Strategy

The navigation behavior of the vehicle uses the sensors on the vehicle to determine where the vehicle is, which way it is moving, and how fast it is going. The control software needs a navigation behavior operating on the vehicle so that it can operate semi-autonomously.

The BUG control software will use a dead reckoning navigation strategy. The navigation behavior uses the vehicle's sensors to determine how far the vehicle has gone lately and how much it has turned. This information can be used to estimate the direction the vehicle is pointing and its current position on the global map.

Figure 3-4 shows how this strategy operates. Given the previous position of the vehicle,

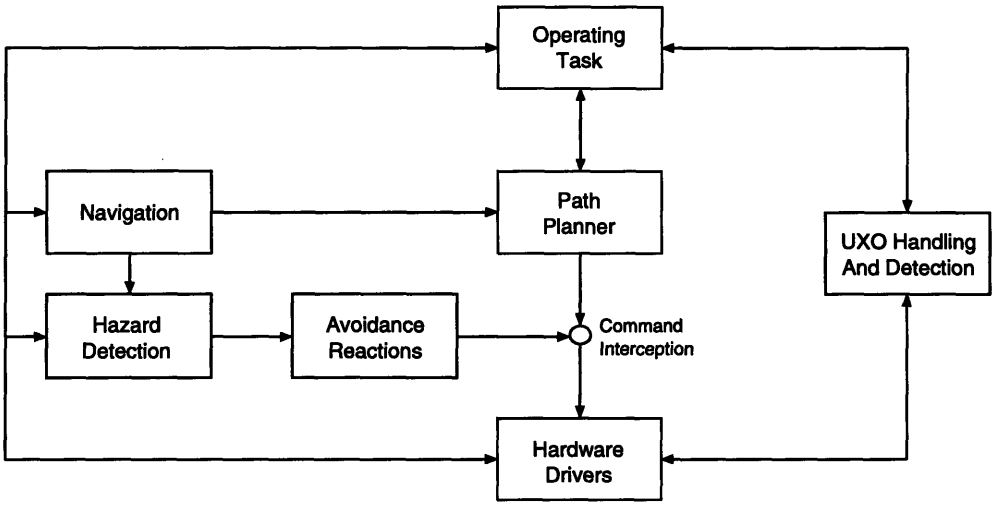

Figure 3-5: Data Flow Diagram

the amount it has turned, and how far it has gone, the current position of the vehicle can be calculated. By repeatedly performing this calculation, the track of the vehicle can be plotted across the map. This strategy is not particularly accurate, for many reasons. Dead reckoning navigation works best if the vehicle does not move large distances between position calculations. Sensor errors such as wheel slippage and gyroscope drift also lead to navigation error.

There are more accurate forms of navigation, but this strategy was chosen because it is fast and requires very little sensor information in order to perform. The computer on-board the vehicle has limited computing power and cannot afford to spend large amounts of its time navigating.

3.4.6 Hazards and Mapping

The BUG vehicle is very susceptible to large rocks, thick shrubs, and other obstacles to its progress. These obstacles and other dangers like cliffs are collectively referred to as *hazards.* Many of the sensors on the vehicle are there to detect hazards. The ultrasonic rangefinders try to detect large obstacles while they are still at a distance and the bumper detects any collisions with objects not detected by the rangefinders.

The BUG software will react to hazards in two ways. It will attempt to avoid any hazard it detects by steering around the hazard. At the same time, it will put the location of the hazard on a map so that it can be taken into considering when planning future paths.

The control software will only handle the immediate reaction of the vehicle to the hazard. Mapping will be done by the control station since this operation can require more computational and storage resources than the vehicle can provide. When the control station plans future paths for the vehicle, it can take the map data it has gathered into account.

3.4.7 Inter-module Data Flow

The flow of data in the control software will be similar in appearance to the data flow for subsumption architectures. Sensor data from the hardware is supplied to all of the simple behaviors, such as navigation and hazard detection. More complex behaviors like path

planning and task control build upon the simple behaviors and combine to produce the final commands to the vehicle. Figure 3-5 shows how the data flow might look.

This design isn't exactly like a subsumption architecture and it isn't supposed to be. The design only tries to use the ideas of separating every behavior and having complex behaviors build on the simple behaviors.

3.4.8 Layer Strategy

In order to insure that there is a clear separation between modules and that simple behaviors are operating without the aid of more complex behaviors, the design will incorporate a layer strategy. This strategy involves ordering modules from highest to lowest and is similar to, but not the same as, layers in subsumption architectures.

Every module will be given its own layer which is a higher level than some layers and a lower level than the other layers. A module is only allowed to call functions in its own layer or a lower layer. If simple behaviors are placed in lower layers than complex behaviors, the layer strategy guarantees that the lower-level behaviors will not be dependent on the higher-levels.

Unfortunately, this means that low-level modules cannot call high-level modules directly. Events allow low-level modules to communicate with the higher levels if upper modules register for callbacks with lower modules. The strict layer strategy can be frustrating at times, but it helps keeps the design in the spirit of the subsumption architecture.

3.4.9 Expandability

The control software may need to be expanded in three key areas: navigation, behaviors, and tasks. Expanding or modifying these areas needs to be as quick and easy as possible. The design described thus far provides this quality. The way this is accomplished is described below.

Complex Navigation

The navigation strategy described in Section 3.4.5 is one of the simplest strategies available. It may become desirable to use a more complex navigation strategy. More intelligent strategies are available which use sensor data about the surrounding environment to track the vehicle's position relative to nearby landmarks. This form of navigation is similar to the way humans usually navigate. We rarely are aware of the exact latitude and longitude of our location; instead, we look at our nearby objects to guess at our location.

The control software design does not directly support complex navigation strategies, but the overall design of the software system allows these strategies to be implemented. Figure 3-1 shows that the control software is in communication with the control station, allowing the two to share data. When it is convenient, the control software can transmit recent sensor data to the control station. If the control station is able to determine an accurate position for the vehicle, it can notify the control software.

This kind of cooperation between the control software and control station allows the vehicle to continue to operate semi-autonomously while the control station uses its extra computational resources to perform complex navigation calculations.

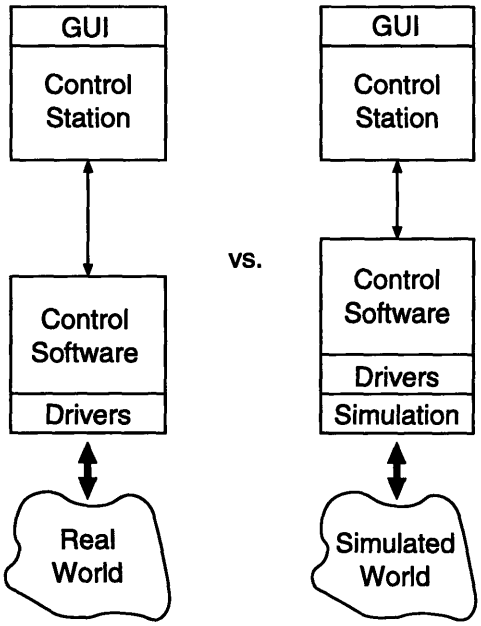

Figure 3-6: Normal vs. Simulated Operation

Additional Behaviors

The software design above only provides the basic behaviors which a mobile robot would need to perform its mission. As the mission changes, it may become necessary to add behaviors to the control software.

The object-oriented nature of the modules makes it easy to add another module to the software. The only difficulty arises when integrating the new module with the rest of the software. The strict layer strategy keeps module dependencies simple and can make the integration process much less difficult.

New Tasks

This design does not actually specify what tasks it will perform. Instead, it creates a task-oriented framework of behaviors which allows any task to accomplish its mission, provided the necessary tasks are in place.

This attitude makes all the tasks the same in the eyes of the control software. The differences between two tasks only lie in the ways they manipulate the behaviors provided by the rest of the software. If all the necessary behaviors are integrated into the software, adding a new task is just as easy as adding the first task. If there is a behavior missing, it can be implemented fairly easily as described above.

3.4.10 Simulation

The final consideration for the design is simulation. The importance of this element of the design becomes evident when the testing and debugging of the software is attempted.

Figure 3-6 shows how a simulated vehicle and environment can be easily incorporated into the design. Normally, the hardware driver portion of the software system is running on the vehicle and interfaces directly with the real world. For simulated operation, the

drivers can be run on a desktop computer and will communicate with a simulated vehicle. The simulated vehicle will operate in a simulated environment. Through the simulation, the driver portion of the software will interface with a simulated environment as if it were operating normally.

This method of simulation is very useful because the main part of the software system is able to operate without knowing whether it is dealing with the real world or a simulated environment.

3.5 Summary

This chapter presented the design for the BUG control software. The mission of the software was first considered and then the goals for the software were set. The successes and failures of other control architectures were examined so that their strengths could be duplicated without including their weaknesses.

The entire software system can be viewed in four parts, with two of the parts running on a desktop computer and two of the parts running on the vehicle. The main goal for the software design is to allow the software to be powerful, expandable, and easy to write and test. The MITy-2 and subsumption control architectures showed some ways in which that goal could be achieved.

The design for the software still leaves some room for minor changes during implementation. The implementation of the BUG software is presented in Chapter 4.

Chapter 4

BUG Software Implementation

This chapter covers the actual implementation of the BUG control software. It is included for use in understanding and modifying the software. The sections cover:

- standards and guidelines used in writing the code
- descriptions of the modules which comprise the software
- ways in which the modules interact
- an overview of what happens during the control software's operation
- expansion of the control software

The control software was implemented in accordance with the design presented in Chapter 3. There are many ways in which the design could be implemented, but we looked for an implementation which is simple and clear.

4.1 Standards and Representations

A major step in the creation of simple, clear control software was to specify standards to which all code must adhere. These standards make the software more portable, more consistent, and more easily understood.

4.1.1 Coding Standards

Programmers often disagree about what standards make code easy to read, but a set of guidelines was chosen to make the control software consistent. These guidelines are:

- Prefix all public functions and global variables with `<filename>_` and prototype them in `<filename>.h`.
- Prefix all private functions and global variables with `_<filename>_`
- Postfix all typedefs with `_t`, structures with `_s`, and values with `_in_<unit>`.
- Put braces on new lines.
- Keep lines shorter than 80 characters.

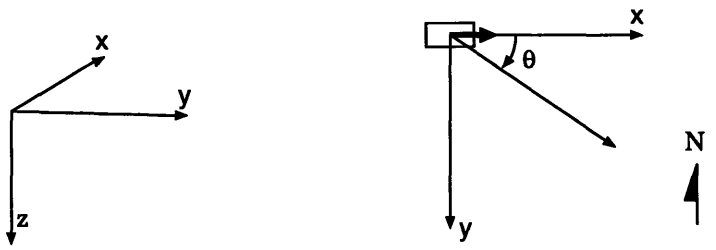

Figure 4-1: Orientation of Axes and Starting Position of Vehicle

- Use spaces to indent. Do not use tabs.
- Use spaces around operators and put one space after commas.
- Make compiler macro names all caps.
- Use the portable variable types defined in the Basic module.

These guidelines are designed to make the code consistently easy to read on any computer. Some guidelines, such as the use of the portable variable types defined in the Basic module, make the software portable between different computer architectures. This is necessary since the software is regularly run on both 32-bit Intel processors and 8-bit Zylog processors.

4.1.2 Internal Representations

It is also important to standardize the internal representation of the physical world. Many parts of the control software interface with the physical world and must communicate with each other. It is much simpler for all of the software to represent the real world in the same fashion instead of trying to convert between multiple representations.

In the case of the BUG control software, the locations of objects in the real world are represented by coordinates on a grid. Different portions of the control software should use the same orientation of axes and origin in order to simplify the integration of the software.

Axes

Position in the world is represented using a three-dimensional cartesian coordinate system. The z-axis points "down" instead of "up" so that angles increase in a clockwise direction. This arrangement of axes is illustrated in Figure 4-1. It is intended to make the coordinate system consistent with compass-style directions and headings.

Initial Position and Orientation

The vehicle is assumed to start at $(0, 0, 0)$ pointing in the x-axis direction. This assumption is made to simplify the control software initialization process. If the vehicle started at a different location or in a different orientation, the control station must update the control software with the correct information.

4.2 Module Descriptions

The control software is separated into multiple modules. Each module is a self-contained set of functions and variables. Modules communicate with each other using calls to public functions. In this way, the modules are object-oriented in nature and can take advantage of the benefits outlined in Section 3.4.2.

The list of all the modules is presented in Figure 4-2 with the modules organized into layers. Each layer consists of a single module. The highest-level layers are at the top of the diagram and the lowest-level layers are at the bottom. The layer strategy is explained in Section 3.4.8.

The following sections describe each of the modules. After an introduction to the purpose of the module, the reason for its location in the layer ordering is given. These sections do not fully explain the implementation of the control software, because it is not necessary to understand exactly how the code was written in order to understand the general features of the implementation. However, the following sections should provide a good foundation for future study of the actual code.

4.2.1 Basic

This module provides the low-level functions and definitions common to all the other modules. This includes more portable data type definitions and functions for manipulating points and vectors in the coordinate space.

Basic is the lowest level module because it is not dependent on any other modules. All modules use Basic to some extent.

4.2.2 Event

The Event module allows the creation and use of events. Most of the control software is event-driven as described in Section 3.4.3. Modules can use the functions in Event to create events and register callbacks with the events of other modules. Whenever an event occurs, all functions registered as callbacks for that event are called.

Event is next in the layer ordering because it provides the events and event-handling that the control software requires. This module also occupies a lower layer because its contents are not very specific to the BUG software. The events implemented here could be integrated into any program.

4.2.3 Sched

The Sched module implements the scheduler. (See Section 3.4.4.) Periodic and delayed callbacks may be requested using the public functions of this module. The scheduler will attempt to activate periodic callbacks before their period elapses. The scheduler guarantees that it will activate delayed callbacks after their delay time has expired.

Sched, like Event, provides very basic functions upon which the rest of the software is built. I consider Sched slightly more specific to the BUG software than Event, so Sched occupies a higher layer position.

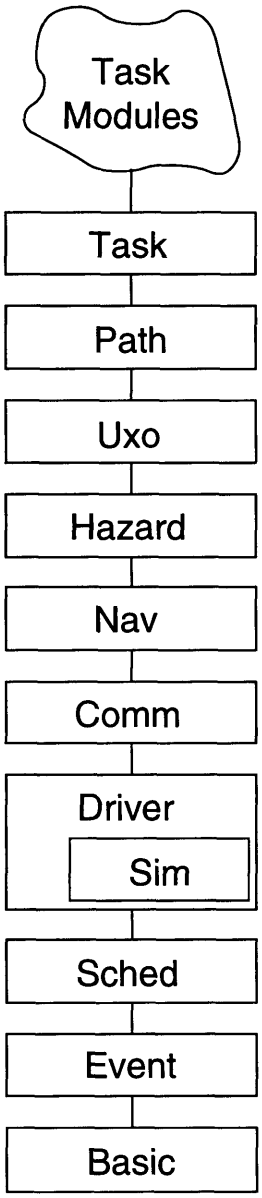

Figure 4-2: Software Modules in Layer Order

4.2.4 Sim

This module is a simulation of the BUG hardware which can be used to debug the control software. When the software is compiled for use in simulation, calls to the functions in Sim replace any commands to the vehicle's hardware. The Sim module keeps track of the simulated vehicle's speed, location, and surrounding physical environment. This allows Sim to simulate the vehicle's sensors as well. The rest of the control software can operate without knowing whether its environment is real or simulated.

Sim is a fairly low-level module, but it needs periodic callbacks in order to update the vehicle's simulated environment. This functionality is provided by Sched, so Sim resides directly above it.

4.2.5 Driver

The Driver module provides the interface between the control software and the vehicle. If the control software is running in a simulated environment, Driver calls functions in the Sim module. If the control software is running on the vehicle in a real environment, Driver sends commands to the vehicle's hardware.

During normal operation, Driver does not require the presence of any of the other modules, but Driver operates differently in a simulated environment. In this case, Driver needs access to Sim. In the layer diagram, Driver is shown enveloping Sim in order to indicate that no modules other than Driver should command the Sim module.

4.2.6 Comm

This module opens communications with the control station and allows other modules to send and receive data. Data is transferred using a simple packet protocol. Two types of packets may be used: reliable and unreliable. Reliable packets are periodically resent until a packet acknowledging the transfer is received. This guarantees that the packet will eventually arrive at its destination. Unreliable packets are sent once and then forgotten. Unreliable packets should be used when possible to reduce the load on the Comm module.

Comm builds upon the functionality of Event and Sched to accomplish its task. It is placed next in the layer order since almost all the remaining modules communicate with the control station in some fashion.

4.2.7 Nav

This module keeps track of the current position of the vehicle. The position is updated using dead-reckoning navigation. This method of navigation is described in Section 3.4.5.

The Nav module uses Driver and Comm to navigate and update the control station with the vehicle's current position and speed. Nav comes before the rest of the modules because the position it calculates is vital state information used by the rest of the software.

4.2.8 Hazard

The Hazard module uses the vehicle's sensors to locate obstacles to the vehicle's movement and other hazards. When a hazard is detected, the other modules are notified and the location of the hazard is sent to the control station.

The vehicle's sensors tend to locate hazards relative to the position of the vehicle, so Hazard uses the position information provided by Nav to determine the coordinates of the

hazards it detects. This requires that Hazard occupy a higher position than Nav in the layer ordering.

4.2.9 Uxo

The Uxo module handles the operations specific to the BUG mission. These operations are the detection of UXO and the manipulation of the grappler hardware. The grappler hardware can be commanded to one of three positions: fully stowed above the front platform, extended above the ground to allow use of the metal detector, and fully extended so that the shovel touches the ground. The rake can also be extended or retracted. When a UXO is detected, other modules are notified via an event.

The detection of UXO is handled by Uxo in the same way detection of hazards is handled by Hazard, so Uxo must also have a higher layer position than Nav. Uxo does not necessarily have to be above Hazard, but I view the management of UXO as a higher-level function than the management of basic obstacles and hazards.

4.2.10 Path

This module maneuvers the vehicle along a given path using the position and heading provided by the Nav module. The types of paths available are:

follow segment The vehicle navigates to a given location by attempting to stay on or near a line segment.

follow point The vehicle navigates to a given location using whatever means necessary.

follow vector The vehicle drives in a specified direction for a specified distance.

follow arc The vehicle maintains a specified rate of turn for a specified distance.

Path is one of the highest-level modules in the control software. Most tasks executed by the vehicle involve following a series of paths and some manipulations of the grappler. In order to accomplish its high-level functions, Path is given a high position in the layer order.

4.2.11 Task

This module manages the task stack. The task stack is a "last-in first-out" style queue for uncompleted tasks. The task on top of the stack is the current task and will execute unless task operation has been suspended. When the current task completes, it "pops" off the stack and the next task becomes active. Tasks can be manually pushed and popped from the control station.

Task is the highest layer because the tasks it manages must have access to functionality provided by the rest of the control software. The modules which actually contain the individual tasks are the only portions of the software which could be considered higher-level than Task.

4.3 Module Interaction

The simple layer ordering keeps module dependencies straightforward, but the diagram of data paths between modules is slightly more complex. Figure 4-3 shows how all the modules communicate.

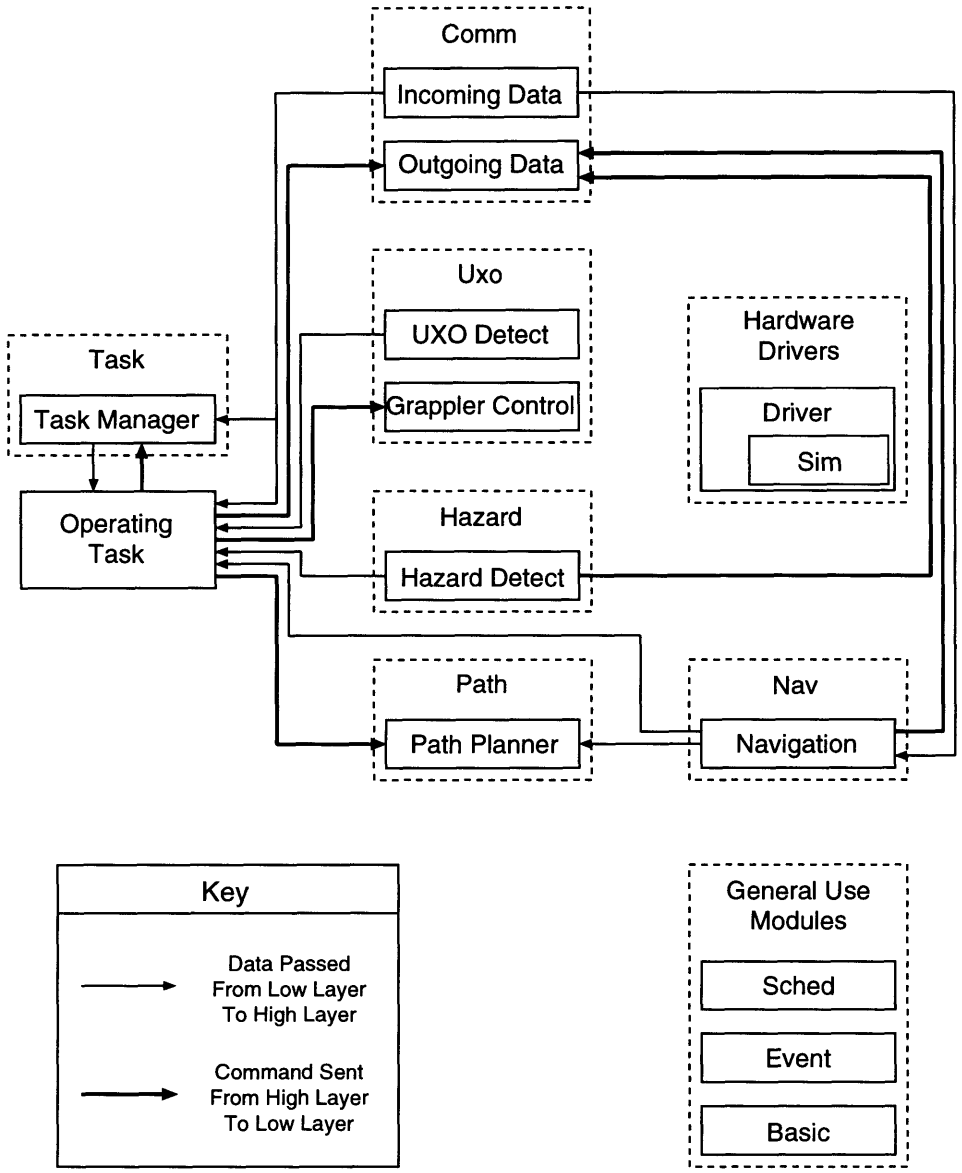

Figure 4-3: Inter-module Data Paths

The Comm and Uxo modules are split into two sections to further clarify the purpose of communications between modules. The arrows indicate the direction that data flows between two modules. A dark arrow indicates that the data is flowing from a higher-level module to a lower-level module. Lighter arrows indicate that the data is flowing from a lower-level module to a higher-level. Since lower layers aren't actually aware of the presence of higher layers, the data flow is accomplished through the use of event callbacks registered by the higher layers.

The diagram shows how the currently operating task has the most control over the operation of the vehicle. The supporting modules provide tools for the operating task and communicate with a limited number of other modules. All of the supporting modules communicate with Driver in order to operate the vehicle's hardware.

The "general use modules" depicted in the lower right-hand corner are used by all the other modules. These three modules provide the foundation upon which the rest of the control software is built.

4.4 Control Software Operation

The diagram picturing module interaction may appear to be extremely complex, but the operation of the integrated software system is actually fairly simple. Each module either implements a set of tools for use by other modules or a behavior which affects the operation of the vehicle. The currently running task acts as the director for the system. It initiates the behaviors of the vehicle necessary to complete the task and modifies or interferes with the operation of behaviors if necessary.

This is where the BUG control software is similar to the subsumption style of autonomous robot control. Low-level modules like Nav and Hazard implement behaviors which operate whenever the vehicle is running. These behaviors seem to be oblivious to the presence of the rest of the control software and provide the foundation for the more complex behaviors. Path is a higher-level behavior which uses that foundation to maneuver the robot from point to point. The tasks themselves are the highest-level behaviors which attempt to achieve a preset goal.

One interesting feature of the BUG control software in comparison to subsumption style control is that the behaviors implemented by individual tasks do not all operate simultaneously. The task stack can be viewed as a list of the vehicle's goals. To accomplish each of the goals, the Task module starts a separate task.

To further clarify the operation of the integrated control software, the follow sections present walk-throughs of two typical tasks. The first task, waypoint, navigates the vehicle along a specified series of path segments. The second task, pickup, uses the grappler mechanism to attempt to retrieve a UXO.

4.4.1 Waypoint Task

When the waypoint task begins, it is given a series of points along the path it should follow. Thanks to the powerful behaviors provided by the Path module, the waypoint task has very little work to do. Its mission is accomplished in the following steps:

1. The waypoint task initiates a *follow segment* Path behavior beginning at the vehicle's current position and ending at the next point in the series. The *follow segment* behavior is given a function to call when the vehicle reaches its destination.

2. The *follow segment* behavior sends commands to Driver which send the vehicle on its way. It also requests a periodic callback from Sched. Using this callback, the behavior monitors the navigation information provided by Nav and makes changes to the vehicle's course as necessary.

3. When the vehicle reaches its destination, the waypoint task is notified via the callback it supplied to Path in Step 1. If there are still more points that the task needs to follow, Steps 1 through 3 are repeated. Otherwise, the waypoint task has completed.

4. The waypoint task notifies the Task module that it is done. The task manager can now pop the waypoint task from the stack and start the next task.

By using the behaviors provided by the rest of the control software, the waypoint task reduces its job to supervision and direction instead of the actual operation of the vehicle.

4.4.2 Pickup Task

The pickup task has a more complex mission than simply following a series of paths. This task needs to manipulate the grappler to retrieve a nearby UXO. However, the control software provides enough built-in behaviors to make this task easy to write.

It is assumed that the UXO was just located using the metal detector and is directly in front of the vehicle. The pickup task performs these steps:

1. Initiate a path-following behavior which backs the vehicle up one foot.

2. Instruct the Uxo module to lower the shovel and extend the rake.

3. Have Path move the vehicle forward one foot.

4. Instruct Uxo to retract the rake. Hopefully the UXO has been raked into the grappler's shovel.

5. Instruct Uxo to bring the shovel into its stowed position.

The pickup task combines the path-following and grappler-manipulation behaviors provided by the control software to attempt to retrieve the UXO. Like the waypoint task, this task is simplified by the use of the functions provided by the underlying layers.

4.5 Expanding the Control Software

One of the main objectives of the design of the the control software is the simple addition of new tasks and behaviors. This section demonstrates how one should approach their creation and integration into the existing software.

4.5.1 New Tasks

New tasks are usually the easiest additions to make to the control software. Not all tasks are as simple as the two tasks outlined above, but they do have several elements in common:

- Every task needs a *start* function which is registered with the task manager when the task initializes. The *start* function does whatever is necessary to put the task in motion. This might include registering callbacks for significant events like UXO detection or communications from the control station. Usually this function will also request a periodic or delayed callback so that the task can monitor its progress or perform additional functions at a later time.

- Each task should also have an *end* function registered with the task manager. This function is called when the task is popped from the task stack. Whether the task successfully completed or was aborted prematurely by the control station, the *end* function should remove all callbacks registered by the task.

- Every task also includes an *init* function which initializes the module. Usually, this only involves registering the task with the task manager.

Different tasks will probably have different paths they want the vehicle to execute and varying commands for the grappler hardware, but individual tasks truly become unique when they register for different event callbacks. Tasks can use these callbacks to respond dynamically to the vehicle's environment.

For example, a task which searches for UXO would probably have a search pattern which the vehicle should navigate. In the event that an obstacle is detected by the vehicle's sensors, the search task may want to modify the search pattern instead of simply allowing the vehicle to avoid the obstacle.

Once all the functions for the new task are completed, addition of the task to the control software is easy. The *init* function for the task is inserted into the list of other *init* functions called when the control software initializes. The new task can then be pushed onto the task stack just like any other task.

4.5.2 New Behaviors

The addition of new behaviors is more complex than the addition of tasks, but it is still done in a modular manner. Each module performs its function by commanding lower-level modules and accepting commands from higher-level modules.

New modules should try to build upon the functions already provided by other modules, if possible. The addition of a new behavior may also involve the creation of multiple new modules which build upon each other to accomplish the desired result. Following these guidelines helps to keep the control software flexible and powerful.

The new module's location in the layer hierarchy should be selected based on which layers it relies upon to function and which layers will need to be at a higher level in order to use it.

4.6 Summary

This chapter has described how the BUG software design was implemented. The implementation is based upon the design from Chapter 3. The main goals for the implementation were to make it easy to write and maintain. The standards and guidelines set forth help the software attain these goals.

The other main thrust for the software was to make it very modular and expandable. The division of the software into object-like modules and the organization of the modules

into a layered hierarchy keeps the software modular and easy to maintain. The modules implement individual behaviors which build upon each other. This idea is borrowed from subsumption control architectures and gives the software powerful expandability.

This implementation was coded in ANSI C and successfully ran on the BUG vehicle. Details about the performance of the software are in Chapter 5.

Chapter 5

Results

Once the software implementation presented in Chapter 4 was written and tested, a series of tasks were created for the UXO retrieval mission. Data were taken on the vehicle's performance while executing these tasks at a series of demonstrations and tests. The sections of this chapter are:

- a list of all tasks implemented for the BUG and how their objectives were achieved
- data showing how the software actually runs
- results from demonstrations and tests which show how the vehicle performed

The information presented in this chapter will show that the control software meets its design goals, stated in Section 3.2. Section 5.1 also shows the range of tasks which were quickly implemented using the foundation provided by the control software.

5.1 Implemented Tasks

A series of tasks were implemented for the control software. When executed under the direction of the control station, these tasks allow the BUG to accomplish its mission. The tasks created for this mission are described in Table 5.1.

Typically, the control station will instruct the vehicle to perform its mission using the following series of tasks:

1. Execute *waypoint* to reach the general location of a UXO.
2. Execute *search* to find the exact location of the UXO.
3. Use *pickup* to acquire the UXO.
4. Follow another *waypoint* task to the dropoff site.
5. Use *dropoff* to deposit the UXO at the site.
6. Return to base or go back to Step 1.

As Table 5.1 indicates, each task was created using the high-level behaviors provided by the control software. Most tasks which mobile robots in our lab perform can be constructed from the behaviors provided.

Task Name	Purpose	Parameters	Behaviors Used
Segment Follow	Navigate path segment	Start and end of segment	Path Nav
Waypoint Follow	Navigate connected series of path segments	Number of waypoints Location of waypoints	Path Nav
Search	Search designated area for UXO	Location of area Search grid type	Path Uxo Nav
Pickup	Acquire UXO	None	Path Uxo Nav
Dropoff	Deposit UXO	None	Path Uxo Nav
Remote Control	Allows direct control by operator	None	Nav

Table 5.1: Tasks Implemented for BUG Control Software

If a new behavior is necessary for a task to accomplish its objective, the behavior will have to be added to the control software as described in Sections 3.4.9 and 4.5.2. In fact, the Uxo module was added to the control software using these methods in order to create the *pickup* and *dropoff* tasks. This achievement is discussed in Section 6.2.1.

5.2 Software Operation

This section describes the actual operation of the control software. Section 5.2.1 includes data on processor usage statistics. Section 5.2.2 is a detailed illustration of exactly how the different modules cooperate to form the full, working control software.

5.2.1 Processor Usage

As described in Chapters 3 and 4, the control software is constructed of many modules which operate concurrently to control the vehicle. Table 5.2 shows approximately how much time each of the major modules spend in control of the processor.

Support modules like Basic, Event, and Task require a negligible amount of attention by the CPU. This is because the functions in these modules are small and are called fairly infrequently.

The majority of the processor cycles are consumed by the Driver module. Driver performs all the low-level operation of the hardware and must be called frequently. In addition, communicating with the hardware involves a lot of very slow I/O. This means that Driver uses a full 56% of the processor's time regardless of what other modules are doing.

Most of the other modules only require a small slice of the processor's time. The Comm module retains control during the processor's idle time in case the control station begins sending large amounts of data to the vehicle. When a task begins running, Path begins using more computing time for its planning algorithms. This results in less idle time for Comm.

Module	CPU Used While Idle	CPU Used During Task
Basic	*	*
Event	*	*
Sched	1%	1%
Driver	56%	56%
Comm	31%	28%
Nav	3%	3%
Hazard	4%	4%
Uxo	2%	2%
Path	2%	6%
Task	*	*

* Indicates Negligible Time in Module

Table 5.2: Percentage of Processor Time Used By Modules

This self-adjusting distribution of processor time demonstrates one of the advantages of concurrent multitasking. Rather than relying on a preemptive kernel to allocate time, each module is allowed to use as much time as necessary to perform its function.

5.2.2 Control Software Walkthrough

Because each module is object-oriented in nature and may be operating concurrently with other modules, the operation of the entire control software system can be viewed as a cooperative effort between the many modules.

Figure 5-1 is a detailed walkthrough of a typical execution of the control software. This example shows the beginning and end of the *waypoint* task's life cycle. As time passes from left to right, the graph indicates how control of the vehicle is passed from module to module.

On the far left, Comm is activated by a periodic callback and receives a command from the control station. The incoming communication indicates that the vehicle should follow a series of waypoints using the *waypoint* task. The figure shows how the control software reacts to this event.

The break in the graph indicates that much time passes before the waypoint task completes. During this period, the various modules use periodic callbacks to perform their functions. In particular, Path uses a periodic callback to monitor the vehicle's progress toward its next waypoint. Whenever a path completes, the *waypoint* task initiates a new path toward the next waypoint.

Finally, the vehicle reaches the last waypoint. The second half of Figure 5-1 shows how the control software responds. Since the *waypoint* task has accomplished its objective, it is popped from the task stack. The BUG is now ready to execute the next task.

5.3 Software Performance

The performance of the control software was tested on two occasions. The first tests were performed in early November, 1996. The final series of tests were performed at a demonstration later in November, 1996.

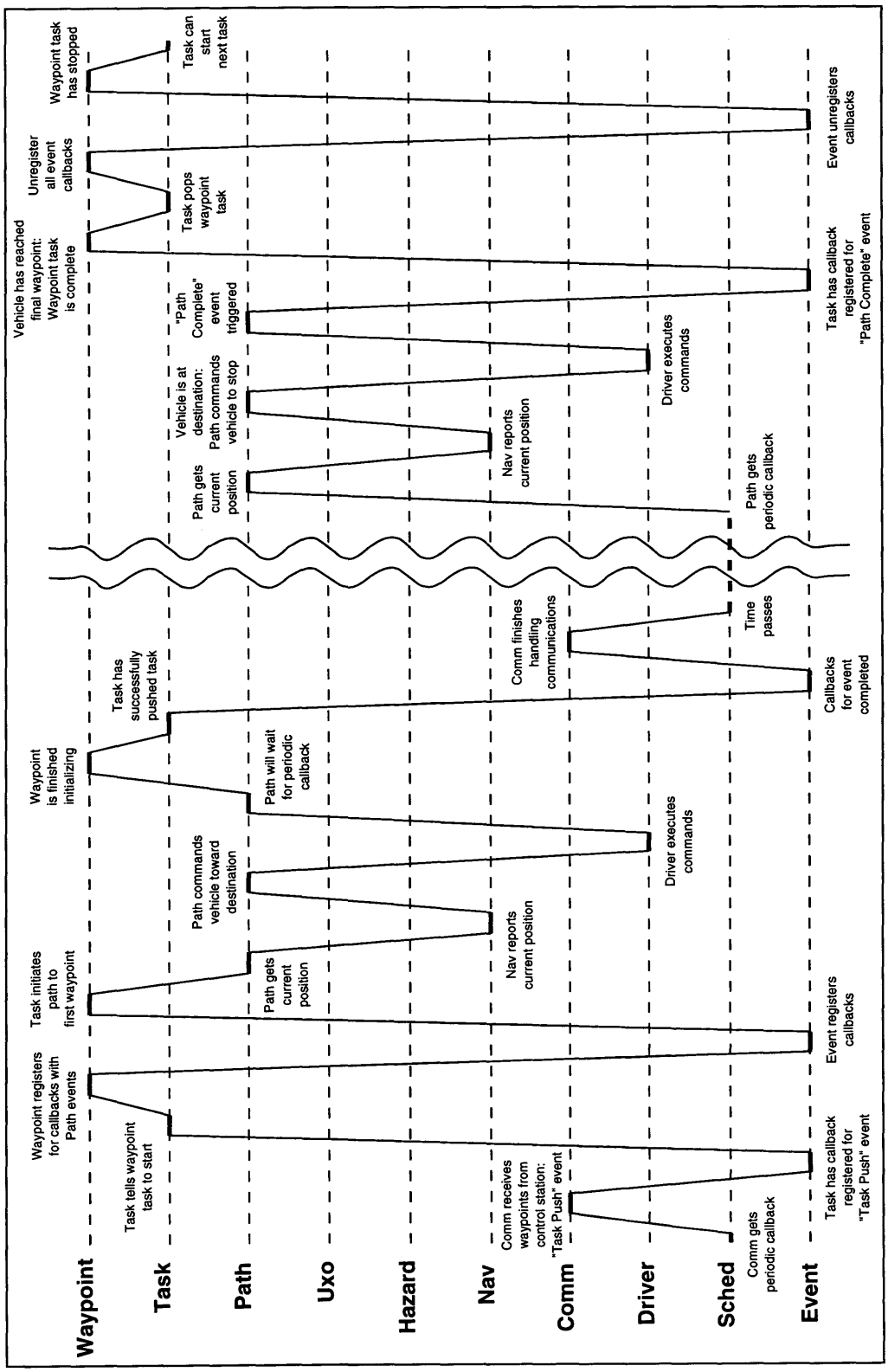

Figure 5-1: Walkthrough of Software Operation

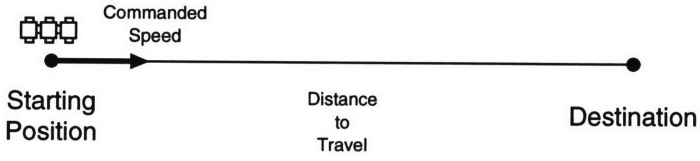

Figure 5-2: Simple Straight Line Test

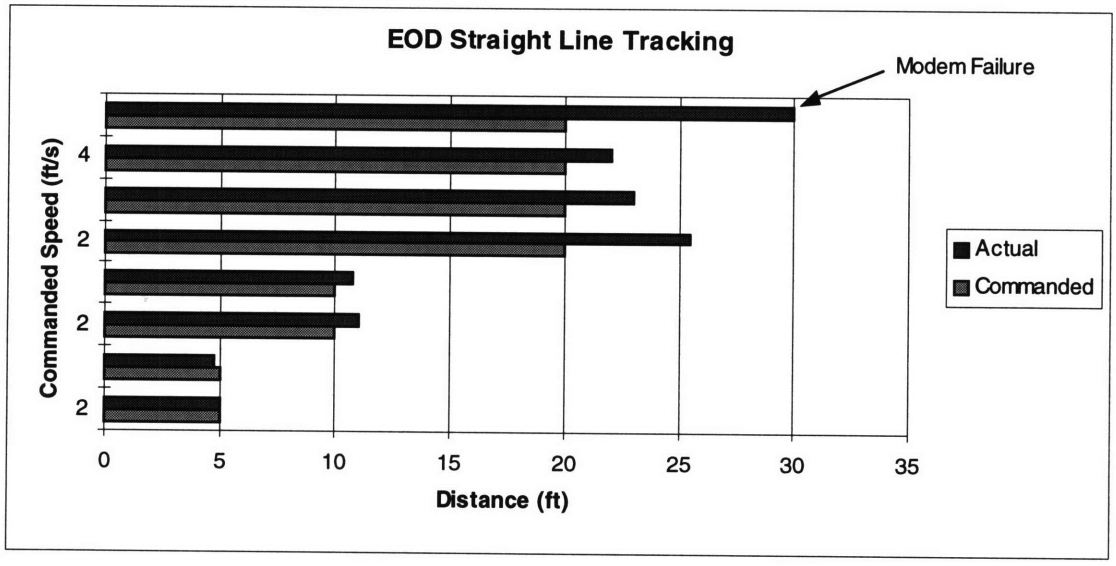

Figure 5-3: Results of Straight Line Tests (Nov '96)

5.3.1 Initial Tests (Nov '96)

The initial tests performed on the BUG were a series of straight-line tests (Figure 5-2). The BUG was instructed to move forward a specified distance and stop as close as possible to its destination. These tests were performed to determine the accuracy of the control software's navigation behavior. The test was repeated at different speeds and for different distances. The results of these tests are charted in Figure 5-3.

In general, the vehicle performed better at slower speeds. This is because the dead reckoning navigation produced errors of the type discussed in Section 3.4.5.

5.3.2 Final Tests (Nov '96)

The graphs in Figure 5-4 show the results of the final tests performed in mid-November. Now that the vehicle had demonstrated its ability to follow a line segment, we decided it was ready to follow a series of waypoints. The two paths the vehicle followed were a five point turn and a trapezoidal circuit of a large area.

The graph on the left shows the result of a five point turn in an enclosed area. The vehicle is alternating between moving forward and backward to perform this turn. The vehicle performed very well on this test and completed the path accurately (within one foot error) for three trials. The vehicle was not able to follow its commanded path exactly, but it remained within an acceptable margin at all times.

The graph to the right displays an attempt at following a trapezoidal path which did a circuit of a large area. In this case, the vehicle was less successful. As the navigation algorithm accumulated error, the vehicle drifted farther and farther from its desired track. This is another example of the limitations of dead reckoning navigation.

5.4 Summary

This chapter presented the results from the BUG control software. The implementation from Chapter 4 was written and tested and a set of tasks were created for the UXO disposal mission.

Data from the operation of the control software shows that the majority of the processor time is used by the Driver module. The other modules cooperatively share the remaining time, with Comm using most of the idle time.

The control software was also subjected to two series of tests to judge its performance. The results of the tests show that the control software functioned well, but a faster processor may help increase the accuracy of the navigation behavior. The success of the software with regard to the thesis goals stated in Section 1.1 is discussed in Chapter 6.

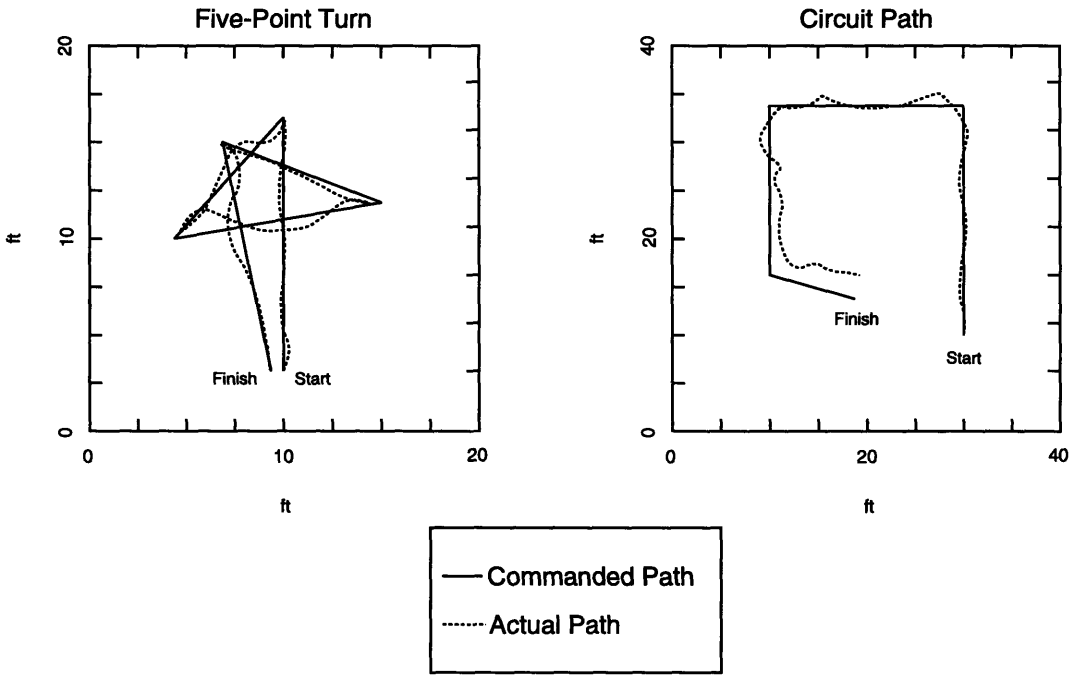

Figure 5-4: Results of Waypoint Test. Note the difference in scale. The vehicle is alternating between forward and backward movement in the Five Point Turn.

Chapter 6

Conclusion

This chapter forms the summary and conclusion for the thesis. The sections include:

- a summary of the contents of the thesis
- conclusions regarding the objectives of the thesis
- a discussion of opportunities for future research

The objective for the thesis was to illustrate the difficulties of control software design by presenting the design process of the BUG control software as a case study.

6.1 Summary of Thesis

The objective for this thesis was to create control software which is flexible and modular enough to be reusable. Most control software performs essentially the same functions: navigation, reactions, and task completion. Therefore, it is very desirable to have a control software system which can be used for many different mobile robots.

Designing the control software to be flexible is essential if the software is to adapt to different mobile robots and their missions. Dividing the software into clearly defined modules helps make the software more flexible and allows small portions of the software to be completely changed without having to redesign the entire software system.

6.1.1 Goals for Software

The control software designed for this thesis controls a mobile robot with a specific mission: the retrieval and disposal of UXO. In order to accomplish this mission, the software design had to meet a number of predefined goals (Section 3.2). Mission-specific goals included: navigation of the vehicle, communication with the control station, and completion of tasks. Additionally, the software was required to be easy to test and expandable.

These goals for the BUG software design are stricter than the goals for the thesis, because the BUG software needs to meet its mission requirements. However, the software goals include the thesis goal as part of their requirements.

6.1.2 Software Design

The BUG control software design is described in detail in Chapter 3. The main elements of the design are:

- Task Orientation
- Object-oriented Structure
- Event-based Architecture
- Multi-Tasking Scheduler
- Dead Reckoning Navigation Strategy
- Local Hazard Detection and Remote Mapping
- Subsumption-Style Data Flow
- Layer Strategy

These design elements were further expanded and formed into a software implementation described in Chapter 4. The implementation demonstrated that it is a functioning control software during a series of basic tests. (See Section 5.3.)

6.2 Conclusion

The design for the BUG control software has demonstrated how the problems associated with control software development can be solved. Typically, the biggest problems in the area of mobile robot control software have been navigation and obstacle avoidance. From a more practical standpoint, I feel that flexibility and reusability of control software can be just as important as brilliant navigation or robust obstacle avoidance.

Reusable control software can be achieved by separating the hardware specific code from the higher-level control code. Unfortunately, flexibility isn't as clear an issue. Flexible and powerful control code needs to be able to incorporate powerful navigation and obstacle avoidance strategies.

During the design and testing of the BUG control software, we found that the layered module architecture was sufficient to provide the desired amount of flexibility and reusability. The flexibility and reusability of the software has not fully been put to the test, but the software has already demonstrated these qualities during our continued development of the BUG.

6.2.1 Flexibility

The BUG control software was designed to be flexible and expandable. The ease with which all of the tasks were created reflects these qualities, but the software was able to fully demonstrate this quality when the Uxo module was added to the system.

Software developers at the IUVC were creating the *pickup* and *dropoff* tasks when they discovered that the control software lacked a collection of behaviors necessary for the operation of the grappler. Using the techniques described in Section 4.5.2, the Uxo module was created. The details of the design of Uxo are in Section 4.2.9. This module provides grappler manipulation behaviors which allow high-level tasks to operate the grappler in cooperation with the other behaviors of the vehicle.

The IUVC found the addition of Uxo to be a painless process. This case demonstrates the power and flexibility of the modular, layered architecture.

6.2.2 Reusability

The potential for the BUG control software to be reused for other mobile robots is based in the software's modular and object-oriented design. We have not had the opportunity to move the control software to another platform yet, but the IUVC is already planning to adapt the software for use on the next generation of BUG.

The control software will easily adapt to changes in the hardware on the vehicle because the hardware-dependent portions of the software are isolated in the Driver module. Once a new Driver module is produced which is compatible with the new hardware, the control software will be ready to interface with the vehicle.

If any of the behaviors of the BUG software need to be altered for the new vehicle, the changes will be restricted to those modules which implement the behaviors. If a module needs to be radically altered, the changes may begin to affect parts of the software. In this case, the strict order of module dependencies defined by the layer ordering will help to soften the blow.

6.2.3 Complex Missions

The flexibility and reusability of the software allows it to demonstrate a variety of behaviors on many platforms, but it is also important to examine the ability of the software to handle complex tasks and missions. The BUG control software's ability to accomplish this based in its task manager and its interaction with the control station.

The task manager manages the list of tasks which the software should execute. The tasks themselves are started and halted by the task manager, but they can command it as well. Individual tasks can affect the operation of the BUG outside their own scope by manipulating the task manager. This limited interaction between tasks executed by the vehicle can be used to perform more complex missions.

The control station can also aid in the execution of complex missions by manipulating the task manager remotely. The control station has the advantages of increased processing power, more data storage, and a human supervisor. These advantages can be used to aid the BUG in the performance of its mission.

6.3 Future Research

The highest priority for future research is to write and test obstacle avoidance behaviors for the vehicle. The software is designed to allow flexible obstacle avoidance behaviors similar to those exhibited by subsumption architectures, but the addition of these behaviors could require additional research into the software design.

Due to the limitations of some of our hardware, we were not able to write the control software in an object-oriented language. Developing this design in a language like C++ would enable the construction of *truly* object-oriented modules. These modules would have the advantage of a better defined interface with other modules as well as the power and flexibility of inheritance. By taking advantage of these properties, the control software could become even more like a subsumption architecture. Exactly how this would be accomplished would need to be determined, but a C++ implementation of this design could make addition of new behaviors like obstacle avoidance much easier.

Appendix A

Background of the IUVC

The Intelligent Unmanned Vehicle Center (IUVC) was first established in 1990 as the Planetary Rover Baseline Experiment (PROBE) laboratory. The IUVC represents a cooperation between the C.S. Draper Laboratory and area universities (MIT, Tufts, Boston University, and Northeastern University) to actively foster research and design of intelligent systems including small robotic technologies.

Since the inception of the PROBE lab in 1990, the IUVC has developed a strong background in autonomous robotics and intelligent systems. The IUVC's proficiencies are in the following specialty areas:

- Smart Sensor Technology
- Sensor Fusion
- Teleoperated Robots
- Autonomous Microrovers
- Autonomous Helicopters
- Vorticity-Controlled Undersea Propulsion

These proficiencies have been gained primarily during the development of the MITy series and the Companion system. The IUVC has gained additional skills through the development of other platforms such as the Small Autonomous Aerial Vehicle and Vorticity Controlled Unmanned Underwater Vehicle.

A.1 The MITy Series

The purpose of the MITy project was to develop a small, autonomous robot (dubbed a "microrover") which would be capable of performing scientific missions in extraterrestrial environments. Some potential deployment areas included Earth's moon and the surface of Mars.

MITy microrovers needed to be able to withstand the rigors of space flight and the extreme environments of other planets and moons. In order to perform its mission, a MITy microrover needed to be able to navigate rocky terrain without losing track of its position. These qualities made the MITy series a good starting point for the design of the Basic UXO Gatherer.

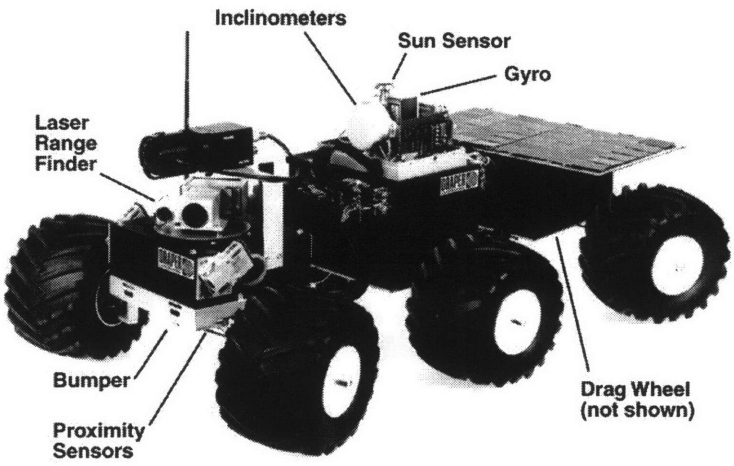

Figure A-1: MITy-2

A.1.1 MITy-1

MITy-1 was the initial prototype of the MITy series. The purpose of MITy-1 was to establish that it was possible to build a small, mobile robot capable of limited autonomous operation. MITy-1 was a success, but its capabilities were very limited [7].

A.1.2 MITy-2

MITy-2 (Figure A-1) built upon the success of its older sibling. This second prototype was fully autonomous and was capable of accomplishing a wide variety of missions [12].

MITy-2 was so capable that it was considered as a solution to the UXO collection problem. Unfortunately, MITy-2 was too slow to provide a viable alternative to sweep teams made up of EOD personnel. The BUGs were designed to be similar to MITy-2, but they were given more powerful motors and a sturdier frame.

A.1.3 MITy-3

MITy-3 was the third experimental prototype in the MITy series. This rover incorporated changes to the mechanical design which improved upon the design of MITy-2. MITy-3 was not as capable as MITy-2, so it was not used as a foundation for the BUG design. However, the lessons learned from MITy-3 were used during the development of the mechanical and electrical components of the BUG.

A.2 Companion

The Companion mobile robot is an autonomous command and control platform designed to supervise the operation of other autonomous vehicles during a mission. Companion is equipped with multiple processors, a laser rangefinder, and a variety of other sensors.

The purpose of Companion is to provide a mobile station with enough computational power to control and coordinate the operation of multiple autonomous vehicles. This would

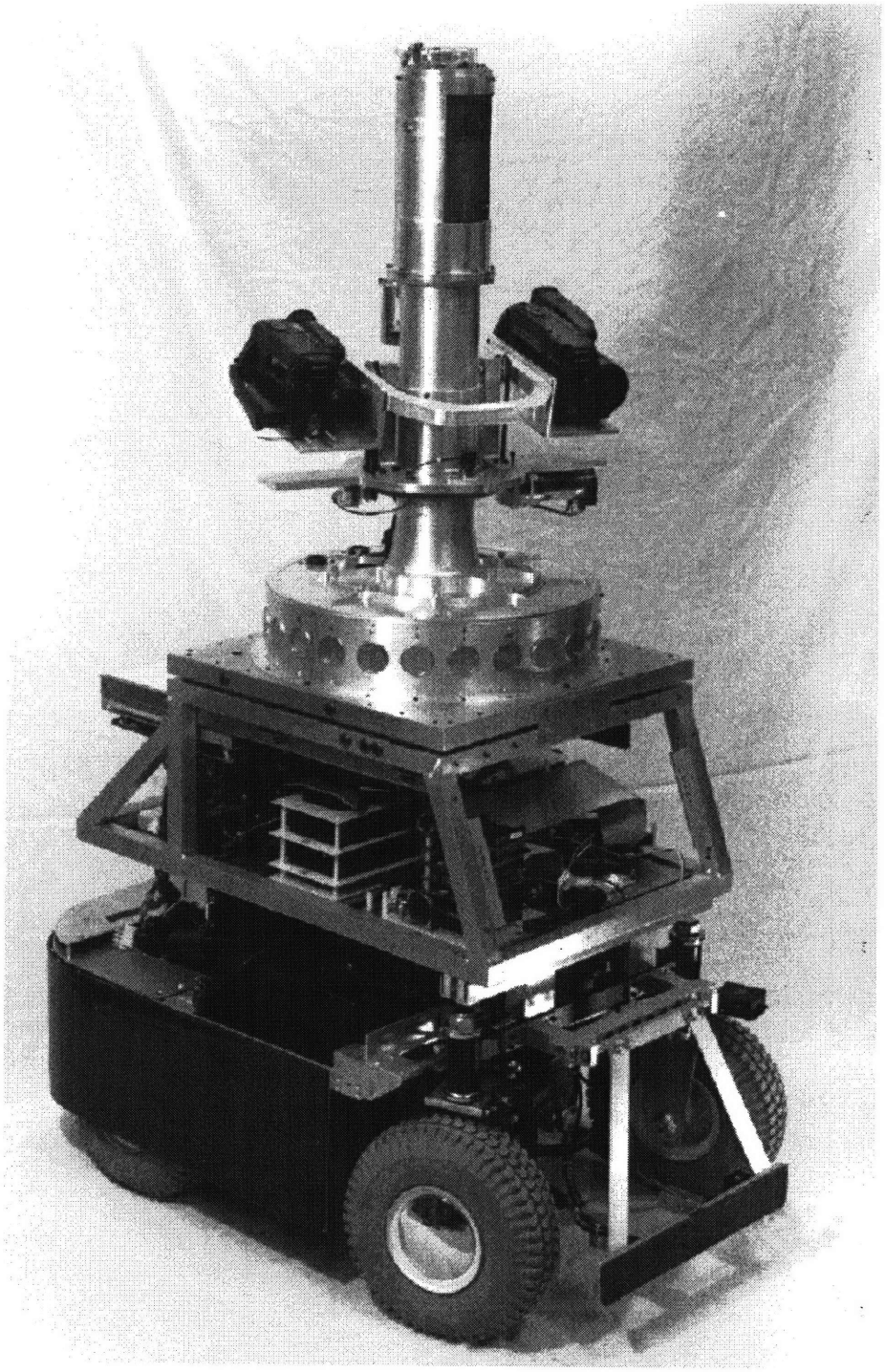

Figure A-2: Companion

Figure A-3: SAAV

allow groups of autonomous agents to operate in concert with minimal human supervision [5] [13].

A.3 Other Achievements

The IUVC has recently incorporated two new autonomous technologies into its family.

A.3.1 Small Autonomous Aerial Vehicle

The Draper Small Autonomous Aerial Vehicle (DSAAV) is currently the most advanced small autonomous helicopter in the nation. The DSAAV is only a few feet in diameter, but it is still capable of taking off, hovering, navigating, and landing under autonomous control [14].

A.3.2 Vorticity Controlled Unmanned Underwater Vehicle

The Vorticity Controlled Unmanned Underwater Vehicle (VCUUV) is a research effort into an underwater propulsion system which mimics the swimming action of a tuna. Tuna are some of the most efficient swimmers in the ocean, and the VCUUV project hopes to achieve that efficiency with a man-made underwater vehicle [1].

Appendix B

EOD-1 and EOD-2 Hardware

EOD-1 and EOD-2 are prototypes for the Basic UXO Gatherer. This appendix describes the hardware that makes up these vehicles. The hardware is separated into mechanical and electrical sections.

B.1 Mechanical Overview

Figure B-1 shows the most recent hardware configuration for EOD-2. The vehicle is equipped with a six-wheel drive flexible frame, front and rear Ackerman steering mechanisms, a modular chassis, and a grappler mechanism for UXO retrieval.

For parallel development purposes, the IUVC has constructed a secondary prototype called EOD-1. EOD-1 shares the same basic mechanical architecture with EOD-2, but lacks a complete sensor suite and grappler mechanism. Table B.1 presents a contrast and comparison of the various components present on each vehicle.

B.1.1 Flexible Frame and Modular Chassis

The vehicle's flexible frame provides a high degree of maneuverability, enabling the rover to traverse rocks, curbs, and uneven terrain (Figure B-2). The frame is constructed of three individual platforms connected by flexible metal rods. The front platform carries the grappler mechanism, a metal detector, sonar rangefinders, and a bumper. The middle platform houses the onboard microprocessor, video camera and transmitter, wireless modem, and additional control hardware. The rear platform contains power regulation circuitry and batteries. (See figure B-5 on page 74 for a detailed schematic diagram of the three platforms.)

The highly modular chassis design allows the operator to swap platform sections with other BUGs. This could become necessary, given the potential for vehicular damage in the mission zone.

B.1.2 Drive Train

Six-wheel drive propulsion also contributes to exceptional maneuverability, producing vehicle speeds up to 6 feet per second. Each aluminum wheel hub, fitted with a knobby rubber tire, is powered by a small (9.8 oz) 12V DC motor with an integrated planetary gearhead and optical encoder. The optical encoders provide feedback used for motor control feedback and autonomous navigation purposes.

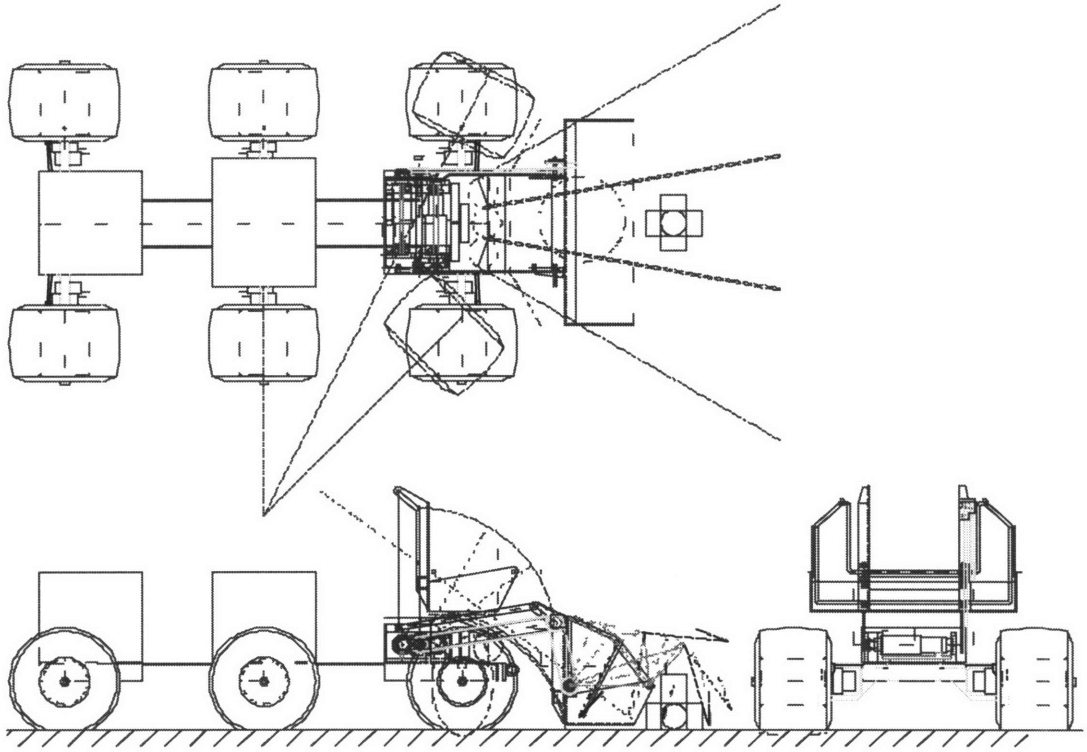

Figure B-1: The EOD-2 Basic UXO Gatherer

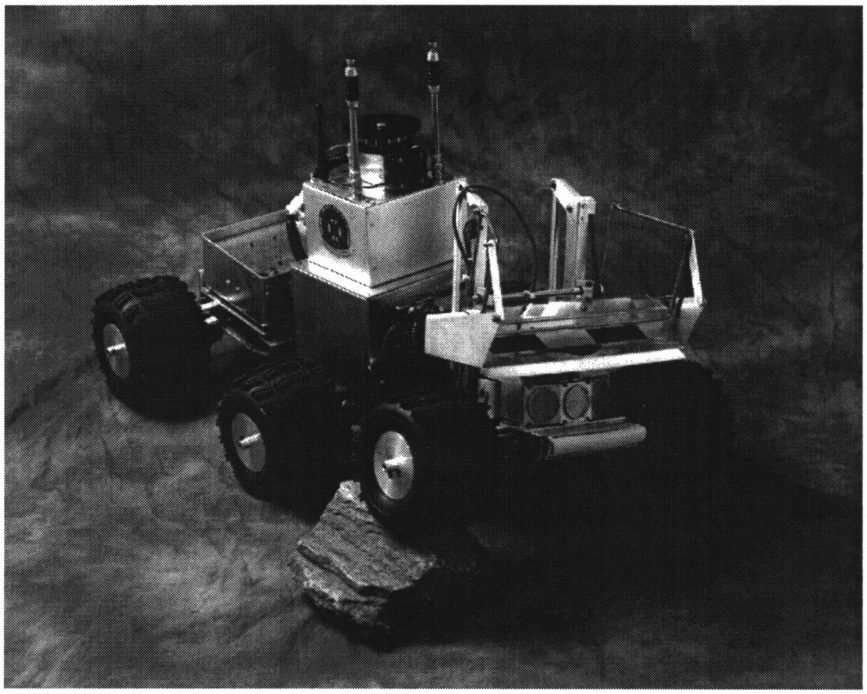

Figure B-2: EOD-2 Demonstrates Its Flexible Chassis

Component/Capability	EOD-1	EOD-2
six-wheeled, three-platform mechanical architecture	●	●
six drive wheel motors with integrated encoders	●	●
grappler mechanism for retrieval of UXO		●
12 MHz Z-World Little Giant Microprocessor with 512KB SRAM and PIO96 Expansion Board	●	●
Systron-Donner micro-mechanical gyroscope		●
Video camera and transmitter	●	●
Front bump sensors		●
Local Positioning System		●
Polaroid sonar ranging module array		●
Proxim wireless modem	●	●
Metal detector		●

Table B.1: Components of EOD-1 and EOD-2

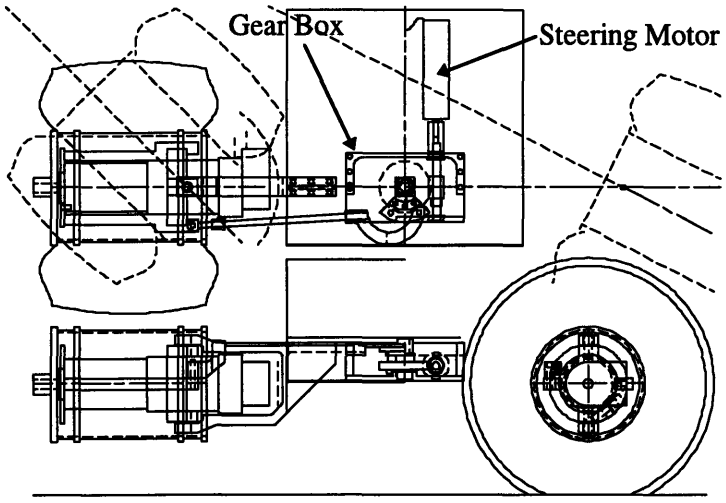

Figure B-3: Ackerman Steering System

B.1.3 Steering System

At the heart of the Ackerman steering system are two 24V DC motors, also equipped with integrated planetary gearheads and optical encoders. The gearhead output shafts are coupled to doubly threaded worm gears which are mounted in aluminum gear boxes (Figure B-3) near the front and read of the BUG. The worm gears mate with worm wheel inside the gear boxes, providing an overall steering ratio of 30:1.

The mechanical linkages providing the Ackerman steering, combined with both front and rear "crab" steering, yield a tight turning radius.

B.1.4 Grappler Mechanism

The grappler mechanism (Figures B-1 and B-4), positioned on the front platform, serves a dual purpose. It is used to both detect and acquire UXO during performance of the vehicle's mission.

The grappler detects potential UXO using a metal detector embedded in its base. The BUG extends the grappler and activates the metal detector when it is searching for UXO. Upon detection of a UXO, the grappler mechanism is used to scoop up the UXO for transport to the ordnance disposal area.

The grappler is driven by two 24V DC motors equipped with integrated gearheads and optical encoders. These encoders provide feedback to the control system and allows autonomous operation of the grappler. One of the 24V motors is used to actuate the scoop/shovel linkage, while the other is used to drive the rake. The rake is used to sweep UXO into the scoop/shovel during the acquisition process.

Vehicle	Dim. (in)	Weight	Top Speed
EOD-1	29 x 17 x 8	26 lbs.	6 ft/s
EOD-2	29 x 17 x 16	36 lbs.	6 ft/s

Table B.2: Physical Characteristics of EOD-2

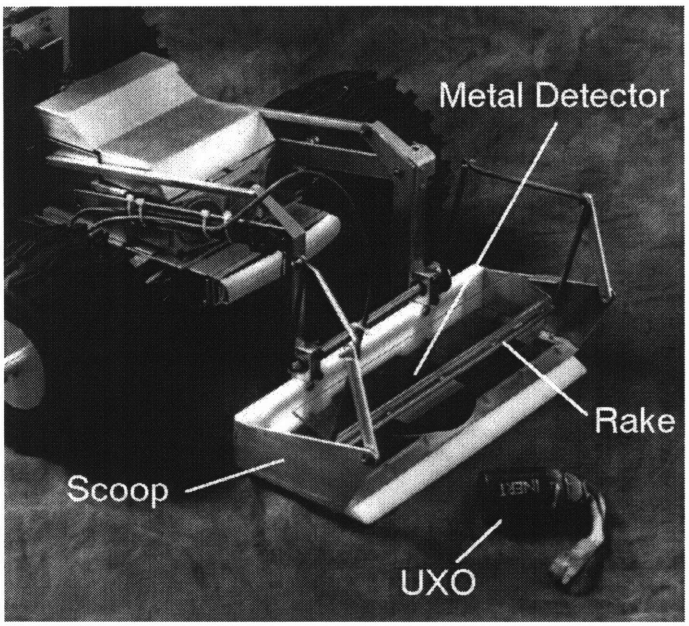

Figure B-4: The Grappler Mechanism

B.2 Electrical Overview

Figure B-5 presents a schematic diagram of the basic electrical hardware layout of the BUG electrical system. Sensors and various electrical hardware are distributed among the three chassis platforms, as discussed in section B.1.1.

B.2.1 Sonar Rangefinders

Located at the front of the rover are three Polaroid ultrasonic ranging modules. The ranging modules work by emitting a series of sound pulses and measuring the time elapsed until the echo returns to the transducer. The time measured can then be multiplied by the speed of sound at ambient conditions to calculate the round-trip distance to the nearest object.

The operation of the ranging module and the calculations for measuring the distance to the nearest object are handled locally by a Basic Stamp. This frees the main processor to perform other tasks while waiting for an echo to reach the ranging module.

B.2.2 Bumper

Also located at the front of the rover is a bump sensor. The bump sensor is a small plate spring mounted in front of two electrical switches such the depressing the plate causes the switches to close. The bumper signal is resistor-tied high and the switch is tied to ground such that when the switch is depressed, the signal is pulled low.

B.2.3 Motor Encoders

Attached to each motor shaft is a rotary optical encoder. The encoders serve to measure the rotation of the motor shafts. The signal from each encoder is decoded by an HCTL 2016 quadrature decoder which tracks the angular position of the motor shaft with a 16-bit

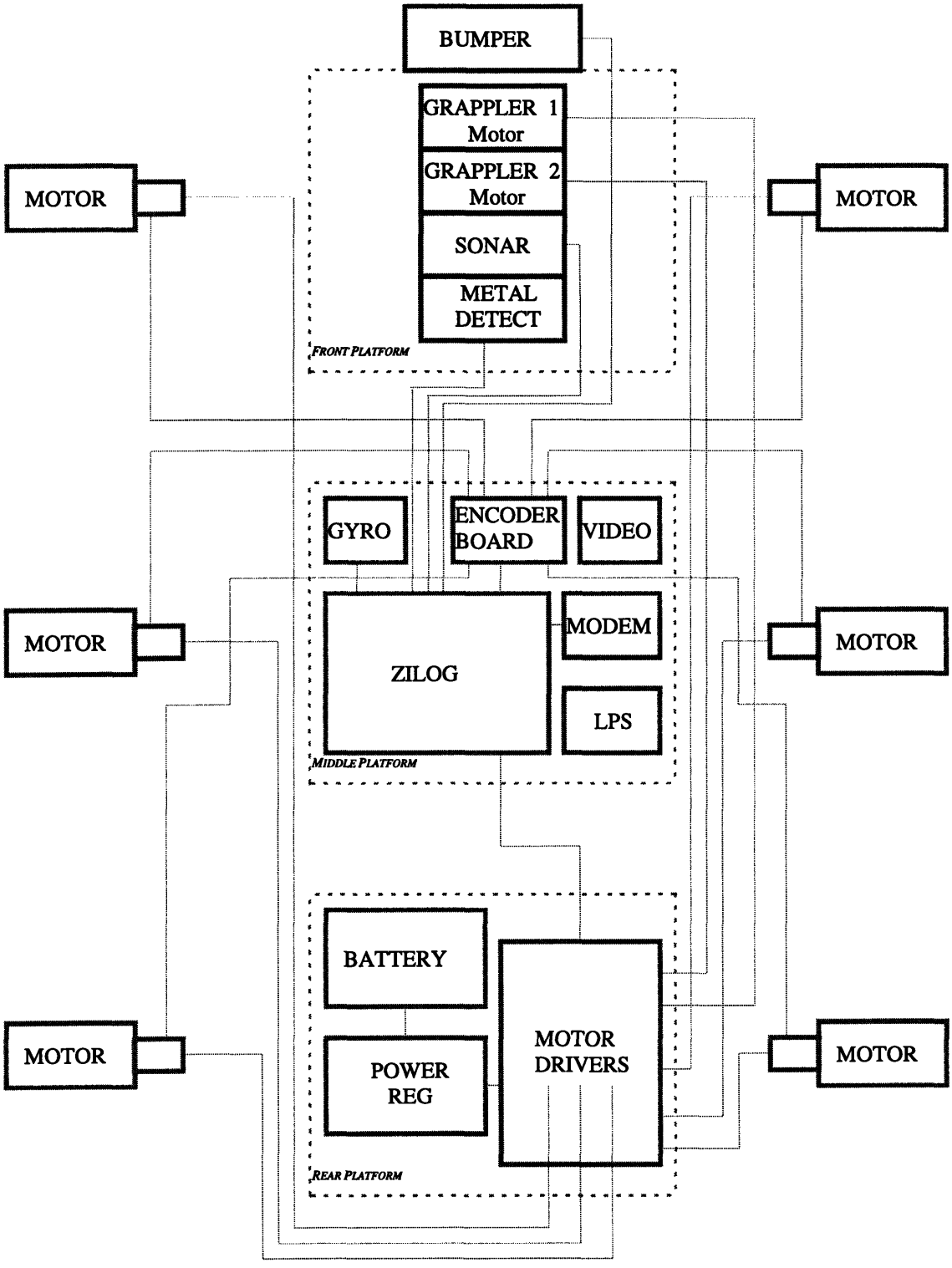

Figure B-5: Components of the Three BUG Platforms

counter. Position can be determined by multiplying the angular rotation of the drive motors by the wheel radius.

Encoders are also used on the steering and grappler motors to determine steering angle and grappler location, respectively. When the BUG first activates, the steering position can be initialized using a pair of photodiode sensors. The position of the grappler can be initialized using two microswitches which are depressed when the grappler reaches its "stowed" position.

B.2.4 Gyro

A micro-mechanical angular rate sensor is used to track changes in heading. A rate gyro offers much higher bandwidth than a magnetic compass and is not affected by abnormalities in magnetic fields.

The rate gyro outputs an analog voltage proportional to the rate of rotation. This analog voltage is low-pass filtered to anti-alias the signal and digitized into a 12-bit digital signal. The digital signal is then locally integrated using a PIC16C84 microcontroller. The integrated signal is scaled and passed to the main processor as the relative heading of the vehicle.

Integrating the signal on a dedicated microcontroller frees the main processor from having to commit its resources to constantly processing the gyro signal.

B.2.5 Onboard Microprocessor

The BUG's main processor is provided by a Z-World Little Giant embedded controller board. The Little Giant is based on a 12 MHz Z180 microprocessor. The Z180 is supported by 512 KB of SRAM, 16 digital I/O lines, 2 serial ports, an 8-channel A/D converter, and a 12-bit D/A converter. The Little Giant is programmed using Z-World's variant of the C programming language called Dynamic C.

Bibliography

[1] Jamie Anderson, Peter Kerrebrock, and Michael Triantafyllou. Concept design of a Flexible-Hull unmanned undersea vehicle. In *Seventh International Offshore and Polar Engineering Conference*, May 1997.

[2] J. G. Bellingham, T. R. Consi, R. Beaton, and W. Hall. Keeping layered control simple. In *Proceedings AUV '90*, 1990.

[3] Rodney A. Brooks. A robust layered control system for a mobile robot. *IEEE Journal of Robotics and Automation*, 1986.

[4] Rodney A. Brooks. Elephants don't play chess. *Robotics and Autonomous Systems*, 6:3–15, 1990.

[5] Terence Y. Chow. Software architecture, path planning, and implementation for an autonomous robot. Master's thesis, Massachusetts Institute of Technology, Department of Mechanical Engineering, May 1996.

[6] Jonathan H. Connell. SSS: A hybrid architecture applied to robot navigation. *IEEE Conference on Robotics and Automation*, pages 2719–2724, 1992.

[7] John E. Gilbert. Design of a micro-rover for a Moon/Mars mission. Master's thesis, Massachusetts Institute of Technology, Department of Mechanical Engineering, December 1992.

[8] Bryan Koontz, Charles Tung, and Ely Wilson. Small autonomous robotic technician. *Mine Lines - Topics in the Art of Warfare*, 3(2), September 1996.

[9] Bryan S. Koontz. A multiple vehicle mission planner to clear unexploded ordnance from a network of roadways. Master's thesis, Massachusetts Institute of Technology, Department of Mechanical Engineering, May 1997.

[10] Benjamin Kuipers, Richard Froom, Wan-Yik Lee, and David Pierce. The semantic hierarchy approach to robot learning. *AAAI Fall Symposium Series*, pages 90–97, October 1992.

[11] Eric J. Malafeew. An autonomous control system for a planetary micro-rover. Master's thesis, Massachusetts Institute of Technology, Department of Mechanical Engineering, May 1993.

[12] Steven Y. Schondorf. Systems engineering for a mars micro-rover. Master's thesis, Massachusetts Institute of Technology, Department of Aeronautics and Astronautics, June 1992.

[13] Steve J. Steiner. Mapping and sensor fusion for an autonomous vehicle. Master's thesis, Massachusetts Institute of Technology, Department of Electrical Engineering and Computer Science, June 1996.

[14] Christian A. Trott. Electronics design for an autonomous helicopter. Master's thesis, Massachusetts Institute of Technology, Department of Electrical Engineering and Computer Science, June 1997.