A Distributed Control Network for a Mobile Robotics Platform

by

Seán P. Adam

Submitted to the Department of
Electrical Engineering and Computer Science
In Partial Fulfillment of the
Requirements for the Degree of

BACHELOR OF SCIENCE IN ELECTRICAL SCIENCE AND ENGINEERING

and

MASTER OF ENGINEERING
IN ELECTRICAL ENGINEERING AND COMPUTER SCIENCE

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

May 1996

Author____________________________________
Department of Electrical Engineering and Computer Science
May 18, 1996

Approved by________________________________
Dr. David Kang
Technical Supervisor

Certified by________________________________
Professor Gill Pratt
Thesis Supervisor

Accepted by________________________________
F.R. Morgenthaler
Chairman, Department Committee on Graduate Theses

A Distributed Control Network for a Mobile Robotics Platform

by

Seán P. Adam

Submitted to the
Department of Electrical Engineering and Computer Science

May 18, 1996

In Partial Fulfillment of the Requirement for the Degree of
Bachelor of Science in Electrical Science and Engineering
and
Master of Engineering in Electrical Engineering and Computer Science

ABSTRACT

Tests carried out on the Unmanned Vehicle Laboratory's initial prototypes: the MITy series; the MITe series; and Companion, revealed that the traditional architectures normally implemented on mobile robotics platforms are not practical for design, manufacturing, or system control. The major drawback of all of these prototypes was that their control systems were designed using sensor fusion. This required a central controller to assimilate all the sensor data before it could be acted upon. This was undesirable due to the limited amount of processing that could be dedicated to higher level control. A system needed to be designed that possessed: support for distributed processing; a predefined hardware interface to allow for parallel development; attributes allowing for upgradability and adaptability; and plug-and-play capabilities. Research showed that such a system could be built using smart node technology. With a smart network, each sensor and actuator possess its own independent processor and signal conditioning circuitry and acts as a separate node on a local network. By transforming from a traditional direct wired central controller system to a smart node networked system, a more efficient robot architecture was obtained.

Technical Supervisor: Dr. David Kang
Unmanned Vehicle Laboratory Supervisor

Thesis Supervisor: Dr. Gill Pratt
Professor, Electrical Engineering & Computer Science

I dedicate this thesis
to the memory of my
grandfather and namesake

John Patrick Hayes

Husband, Father, and Patriot

He labored to guarantee his children
and his children's children a better life.
Though I never had a chance to know him,
I was raised with stories of his life and his deeds
and I can only hope to one day be the man that he was.

ACKNOWLEDGMENTS

I would like to acknowledge all of the men and women I have had the opportunity and pleasure to work with over the years on this project.

I would like to thank the following: William Kaliardos and Shane Farritor for being there that cold rainy night in Faneuil Hall. Terence Chow and Steve Steiner for putting up with my Irish temper and my ranting and raving about things beyond all of our control on the Companion and EOD prototypes. Jim Dyess, Bryan Koontz, and William McFarland for supporting me when push came to shove and decisions needed to be made.

I would especially like to thank Pehr Anderson, Charles Tung, and Ely Wilson for the help and support they gave me in setting up and using the Echelon network and developing the prototype smart nodes. Without them my research would not have been possible.

I would like to extend my gratitude to Dr. David Kang who gave me the opportunity to research this thesis and who always forced me to work to my potential. I would like to thank Professor Gill Pratt for taking the time to read and critique this paper. I would also like to thank Anne Hunter for her continued support, guidance, and friendship throughout my MIT career.

I give my greatest thanks and love to my mom and dad, my brother Warren, and my grandmother Elizabeth Hayes for always believing in me and for giving me a place that I can always call home.

I also want to thank Jill whose love and support acted as my guiding light throughout this work.

Finally, I want to thank my Lord and Savior for catching me when I stumbled and carrying me when I fell. Without him, none of this would have been possible.

Thank you all.

Animo Non Atitia

This thesis was prepared at The Charles Stark Draper Laboratory, Inc., under IR&D.

Publication of this thesis does not constitute approval by Draper or the sponsoring agency of the findings or conclusions contained herein. It is published for the exchange and stimulation of ideas.

I hereby assign my copyright of this thesis to The Charles Stark Draper Laboratory, Inc., Cambridge, Massachusetts.

Seán P. Ad

Seán Patrick Adam

Permission is hereby granted by The Charles Stark Draper Laboratory, Inc., to the Massachusetts Institute of Technology to reproduce any or all of this thesis.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

INTRODUCTION CHAPTER 1

1.1. Motivation:

In 1990, the Draper Planetary Rover Baseline Experiment (PROBE) Laboratory was started to research, design, and manufacture an autonomous micro-rover for planetary exploration. The initial rovers were designed around specifications set by NASA for its MESUR Pathfinder mission. The first rover prototype (MITy-1) was a proof-of-concept platform. It was equipped with an earthbound sensor suite including three acoustic range finders and a compass. After analyzing the test data from MITy-1, a second prototype was built. Where MITy-1's control system was completely reflexive with no path planning behavior, MITy-2's control system assimilated all of the sensor data to allow for complete path planning behavior. Only sensors that could be used for exploration of the Moon or Mars were employed. These included a laser range finder for long range hazard avoidance, proximity sensors for cliff detection, a gyro and a sun sensor, inclinometers for 3-dimensional navigation, and tachometers for speed control. Both were fully functioning autonomous robot prototypes. Analysis of MITy-2's mechanical platform led to the design of MITy-3. Though never fully autonomous, MITy-3 allowed for research into various methods of steering. To alleviate loading on the processor caused by complex motor control code, MITy-3 implemented its motor control in hardware.

Due to the success of the PROBE Lab's MITy-series, Draper decided that more research should be carried out in the area of robotics and autonomous control systems. The Unmanned Vehicle Laboratory (UVL) was formed in 1994. UVL's task was to investigate possible applications for autonomous and semi-autonomous robots and determine how best to meet the necessary requirements of those applications. The first study dealt with the design of a sensor fusion package that would allow for the autonomization of almost any vehicle. The test platform, known as Companion, was an electric wheelchair. The Companion control system was designed around an 486 Laptop running the higher level path planning control code and a Ampro-386 board to handle the assimilation of all of the sensor data. This was an improvement over the MITy architecture which only utilized a Zilog-180 microprocessor board for data processing and path planning.

1.2. Objective:

The knowledge gained from working with Companion prompted UVL to begin researching the possibility of designing a system which would be easier to build, easier to modify, and able to make use of the best features of the MITy and Companion prototype series. This thesis is divided into three different areas of research and development:

- Research of various proposed control architectures for robot systems. This research also includes studying the current UVL prototypes to determine their advantages and disadvantages.
- Research into the use of smart nodes to improve upon the traditional control architectures normally used in robot design.
- Implementation of a simple smart node system.

The objective of this research is to determine a method by which a more robust and efficient robot platform can be designed and manufactured.

DRAPER UNMANNED VEHICLE LABORATORY PROTOTYPES CHAPTER 2

2.1. The MITy Series -- Robots for Planetary Exploration

In 1990, the newly founded PROBE lab, under the direction of Dr. David Kang, began a system study to determine what was necessary to design and build a robot for planetary exploration. This initial study was completed in June 1992 [1]. It was determined that a successful planetary robot would have to be small, robust, and most importantly completely autonomous. Three prototype robots were designed and built based off the findings of this study: MITy-1, MITy-2, and MITy-3.

The planets considered in the initial study were the Moon and Mars. Though tele-operation is possible for a lunar robot, the time to communicate with a robot on Mars, best case 40 minutes, necessitates a high degree of autonomy for the robots. The robot needs to be able to navigate through an unknown environment while avoiding obstacles and pitfalls; all the while, maintaining a sense of where it has been, where it currently is, and where it wants to go. To accomplish these tasks, the robot requires a complete hazard avoidance and navigation sensor package. The robot also needs a microprocessor capable of processing and fusing all the sensor data for use by the control system. Finally, the robot needs a stable and dependable power supply that will last the entire mission. The MITy prototypes use rechargeable nickel-cadmium batteries and solar panels to restore on-board power.

2.1.1. Basic MITy Structure:

There were a number of design characteristics that were maintained in all the prototypes. Each rover was based upon a three-platform frame interconnected by two flexible steel rods. These rods allowed for flexibility in pitch and roll but not in yaw. The basic MITy structure is shown in Figure 2.1 [2].

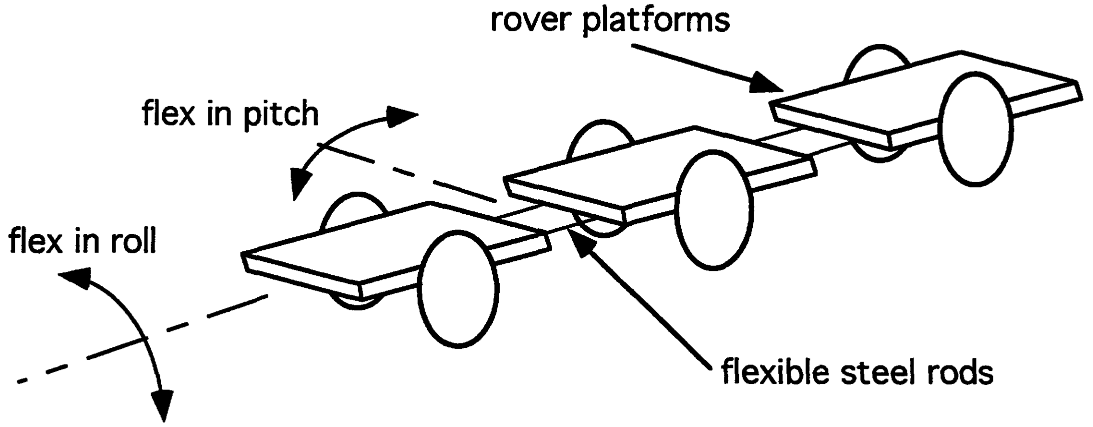


Figure 2-1. Basic MITy Structure

Since the rover is meant to move forward and backing up is only done when necessary to avoid an obstacle, the front platform is dedicated to the hazard avoidance sensors. The middle platform holds the processor package and all the navigational sensors. The rear platform is used to hold battery packs, solar cells, and any mission science packages.

There are two wheels on each platform. Actuation is provided by motors within each of the wheels. MITy-1 and MITy-2 use an Ackerman steering system similar to that used on automobiles. MITy-3 makes use of differential speed control where the wheels are attached to a free pivoting axle [2]. Assuming that the failure of any two motors does not produce drag on the robot, any combination of four wheels will be enough to move the robot.

2.1.2. MITy-1 -- A Proof-of-Concept:

MITy-1 was designed as a proof-of-concept prototype to demonstrate that the PROBE lab was capable of manufacturing a fully autonomous robot. A block diagram of MITy-1 can be seen in Figure 2-2. MITy-1 is 26" long, 13.4" wide, and weighs approximately 8.9 kg.

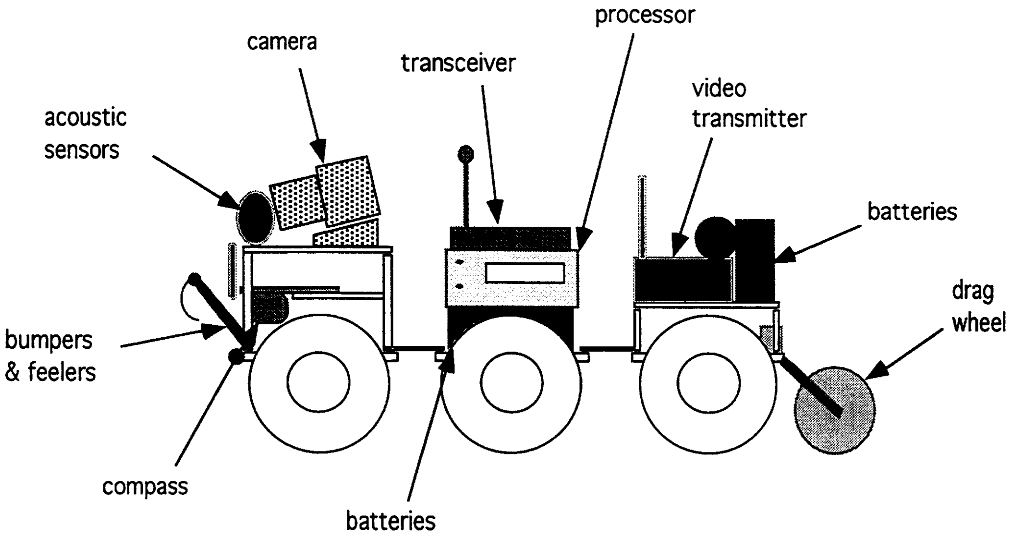


Figure 2-2. MITy-1

To minimize the cost and the development time, only earth-based sensors were implemented on this prototype. Navigation was performed using a drag wheel to measure distance traveled and a compass to determine vehicle heading. Using dead reckoning, these sensors allow the robot to determine its position relative to where it began. Long range hazard avoidance was provided by three Polaroid acoustic range-finders while mechanical feelers and bumpers provided more localized obstacle information. The sonars where mounted on the front platform of the robot and were set at angles to allow full coverage of the robot's forward path. The mechanical feelers were mounted on either side of the front platform and extended out past the wheels. Bumpers where mounted on the front and rear platforms and acted as a last resort if the sonars and feelers failed to detect an obstacle.

A 68C11 board was used as the central controller. Due to the board's limited processing capabilities, a relatively simple reflexive control system was implemented on the robot. The robot also possessed a video and data transmitter to allow control by a PC-based ground station. Initial tests demonstrated that the robot was able to perform simple autonomous tasks without external intervention. When given a destination 100 meters away, the robot was able to arrive within 10 meters of the designated point. This was well within the mission criteria set by NASA's Jet Propulsion Laboratory.

2.1.3. MITy-2 -- A Working Autonomous Prototype:

Using the knowledge and experience gained from designing, building, and testing MITy-1, the development of a more advanced prototype, MITy-2, was initiated. A number of factors were considered during the design of MITy-2. The sensor suite would have to be geared more towards an unknown, hostile, planetary environment. Where MITy-1's sensors allowed only 2-dimensional navigation, MITy-2 would be expected to travel through a 3-dimensional environment. Since the reflexive control implemented on MITy-1 would not be sufficient for traversing the Martian landscape, a more powerful processor would have to be used to implement sensor fusion, path planning, and terrain mapping on board the robot. Also, the MITy-2 prototype would have to be more robust than its predecessor. A picture of MITy-2 can be seen in Figure 2-3.

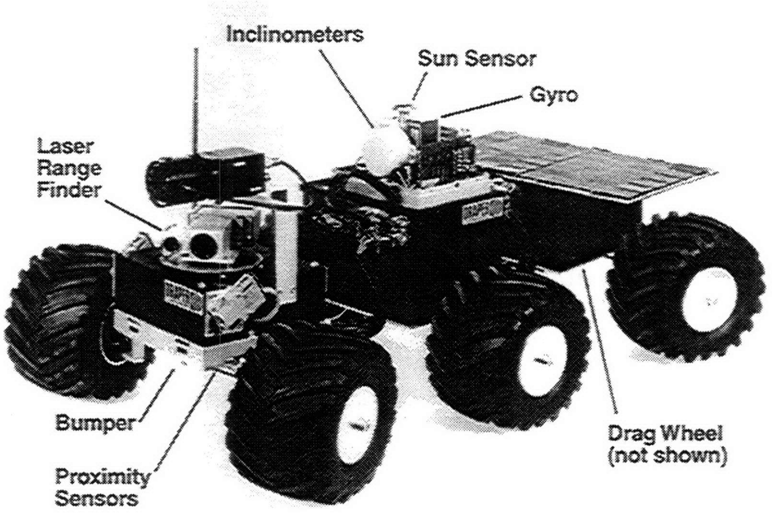


Figure 2-3. MITy-2

It was decided that hazard avoidance would be performed using laser range-finders, infrared proximity sensors, and mechanical bumpers. The most important reason for choosing infrared lasers over acoustical sensors was because acoustic sonars require an atmosphere to act as a medium for the transmission of the sound wave. However, laser range-finders can work in the void of space as well as on Earth. Also, the laser range-finder used for long distance hazard avoidance gave more accurate data than the acoustical sonars used on MITy-1 that suffer from false reflections. Infrared proximity sensors were used for cliff detection to guarantee that the robot did not damage itself by driving off high embankments. The mechanical bumpers act as a last line of defense for the robot. If an obstacle was not detected by the laser range-finder or the control system was unable to maneuver correctly around the object, then the bumpers will be activated and the robot will reverse direction.

For navigation, MITy-2 was outfitted with a mechanical rate-gyro, inclinometers and accelerometers, a sun sensor, and a non-powered drag-wheel. The gyro was used to keep track of the heading of the robot. Unfortunately, the gyro suffers from a large amount of drift that causes errors in navigation information. This was the reason that a sun sensor was also used. The sun sensor monitors the angle of the sun in comparison to the robot. By referencing both the gyro data and the sun sensor data, errors were usually removed. The control system was also designed to deal with the fact that either sensor could return totally inaccurate data that should be ignored. The inclinometers and accelerometers were used to monitor the pitch and roll of the robot's three platforms.

2.1.4. MITy-3-- A System Redesign:

While testing continued on MITy-2, a new design effort was undertaken to upgrade the overall MITy mechanical/electrical system. The third prototype, MITy-3, was never outfitted with any sensors. It was decided that MITy-2 had proven the labs ability to design and build a completely autonomous robot. MITy-3 was mainly an improvement on the hardware structure. Even though MITy-2 had been a step in the right direction for the design of a planetary micro-rover, there were still a number of system characteristics that would never allow the MITy system to be spaced qualified. MITy-3 can be seen in Figure 2-4.

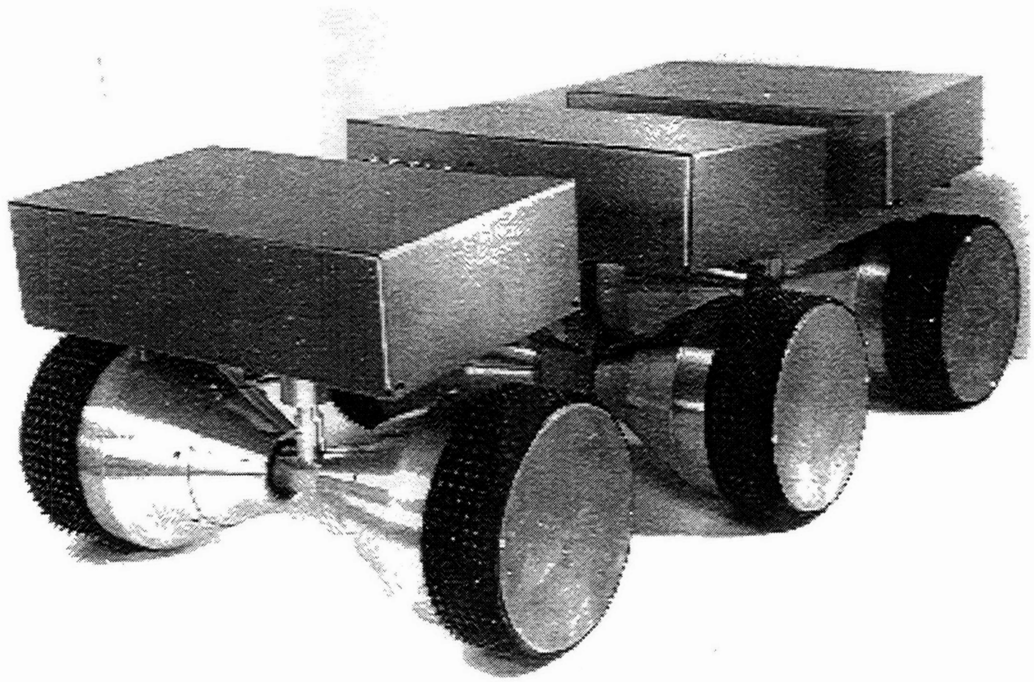


Figure 2-4. MITy-3

Two of the greatest flaws that MITy-2 possessed were the methods by which motor control and steering were performed. On MITy-2, the microprocessor was responsible for outputting pulse-width modulation (PWM) signals to drive each of the motors. Steering was performed by servos normally found on remote control racing cars. These attributes had a negative affect on both the hardware and the software of the robot. Creation of the PWM signals placed a significant load on the processor which reduced the amount of processing that could be directed toward sensor fusion and higher level control. The trade off between sharing processor time with the steering and the control system allowed MITy-2 to have only eight possible motor speeds. The use of "hobby" grade servos led to a number of mechanical failures during testing. It was concluded that a solution needed to be found that would alleviate both problems.

Instead of a servo for steering, the MITy-3 front and rear wheels were mounted to axles that could pivoted freely about their center. Steering was performed by using differential speed control. After initial testing of this concept, an additional motor was added to the rear and front platforms to act as a clutch for the free pivoting axle. These motors did not aid in the steering of the robot; however, they locked the axles in place when the correct angle of turn had been achieved.[2] The amount of control necessary for this method of steering required precision motor control. The eight speeds offered by the MITy-2 motor control would not be sufficient to implement differential speed control. A new circuit was designed to allow better control of the motors. The final circuit was able to output 256 different PWM signals. Due to the motors' electrical and mechanical characteristics, these 256 PWM signals translated into approximately 20 noticeable motor speeds. However, these were enough to implement the new steering concepts.

2.2. Companion -- Advanced Robotics Platform for Sensor Fusion:

The early nineties saw a decrease in money being spent by Congress on space research. Consequently, a new source of income needed to be found by UVL. Market research determined that opportunities existed in the area of autonomous and semi-autonomous robotics platforms for Earth based applications. It was decided that the future of UVL lay in researching the area of sensor fusion. Instead of focusing on one particular application, UVL would begin designing a generic robotics platform controller that would have the capability of making any vehicle semi - autonomous to autonomous. A remote control wheelchair frame was used as the mobile test platform for the sensor fusion package. There were a number of differences between the MITy and Companion prototypes. Most importantly, the MITy prototypes had to be small and inexpensive. Consequently, many of the components had to be specially designed to fit these constraints. Fortunately, it was determined that

Companion would not suffer from the same issues. This allowed UVL to outfit the robot with off-the-shelf components that were not available to the MITy series.

The entire idea behind the Companion project was that the problems normally experienced due to sensor fusion could be removed by using multiple processors. Initially it was decided that Companion would possess up to three processors running in parallel just to deal with the fusion of the sensor's data. The higher level control system and path planner were run on a PC laptop. Communication between the laptop and the fusion-processors was carried out through ethernet lines. The robot basically become a mobile LAN. A ground station was also be used to control the robot via a RF modem link.

Companion was completely outfitted with both navigation and hazard avoidance sensors. From the knowledge ascertained from the MITy series, a ring of 24 acoustic sensors was used to allow for obstacle avoidance and environmental mapping. To increase the accuracy of the map created by the acoustic sensors, a sophisticated laser range-finder system was also implemented. The laser system scanned 360° around the robot and had a vertical range equivalent to the height of the robot. Companion also possessed two video cameras that had the same scanning ranges as the laser system. These cameras were used to aid in tele-operation and also for recording necessary information of the surrounding environment. Additional hazard avoidance was supported by bumpers and short range proximity sensors to act as a last line of defense in case an obstacle was not detected by the sonar ring and the laser system. Navigation was performed using dead reckoning based off data from an on-board gyro and the encoders on the front wheels. Future plans include the addition of a GPS system. Figure 2-5 is a picture of the Companion Sensor Fusion prototype.

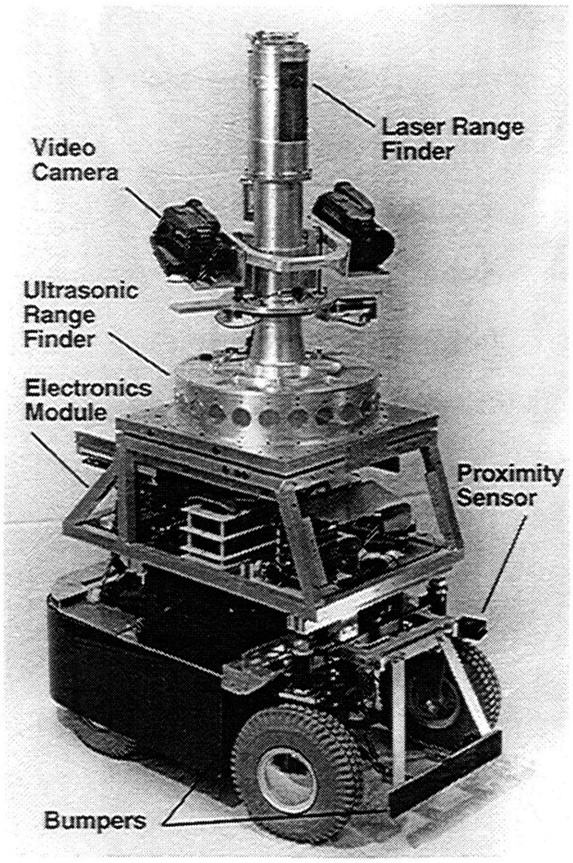


Figure 2-5. Companion Prototype

2.3. MITes -- Nano-Architecture:

Another avenue that UVL began researching was the creation of nano-robots. These robots would be smaller than the MITy series and would be designed for tight areas; such as, nuclear reactors. The objective was to design a simple and inexpensive robot to be used in situations where the robot may be damaged or destroyed. Working with the MITy and Companion prototypes had shown the UVL engineers the importance of modularity and testability for a robot system. It was decided that the MITes would be a bus-based architecture. By defining the hardware interface between the transducers and the central controller, it was hoped that it would become easier to maintain and upgrade the sub-systems on the robot.

The MITes hazard avoidance system was based off a single rotating acoustic sensor and bumpers. The acoustic sensor scanned the entire area in front of the robot to guarantee obstacle avoidance. Since the MITes were designed to be an Earth-based robot, navigation was carried out through dead-reckoning using a compass and encoders on the front wheels. Unlike the MITy series where the robot's platform was specially design, the MITes were built around off-the-shelf RC model race cars.

CONTROL ARCHITECTURE THEORIES CHAPTER 3

3.1. Horizontal and Vertical Decomposition:

The hierarchy of a robot can either be setup in a horizontal or vertical decomposition. Figure 3-1 shows the traditional horizontal decomposition of an autonomous robot's control system.

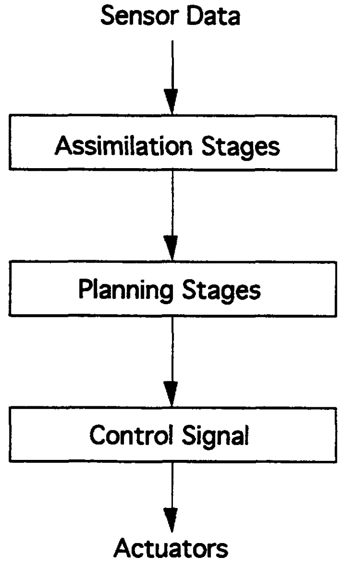


Figure 3-1. Horizontal Decomposition

As can be seen, the sensory data is passed into the Assimilation Stages where the processed data is put into a usable format. The processed information is then sent to the Planning Stages where the control system determines what must be done to achieve the predetermined objectives set by the user. The Planning Stages then issue a Control Signal to the actuators. There are a number of disadvantages to using the horizontal approach. Each layer of the control system requires the assimilated data from the preceding layer. This causes two problems. First, since all sensory data must pass through each consecutive layer (or module), the control system has "an unavoidable delay in the loop between sensing and action." Second, horizontal decomposition places limitations on the flexibility of the system. If modifications to a particular module cause that module's I/O requirements to change, then the I/O of the adjacent modules may also change. At best, these will be the only necessary changes. However, altering the I/O of the adjacent modules may cause a domino affect throughout the control system that requires a reworking of all the layers' interfaces. These are problems that are not normally seen when using a vertical decomposition.

With a vertical decomposition approach, the transducers are connected in parallel. Figure 3-2 shows the traditional vertical decomposition.

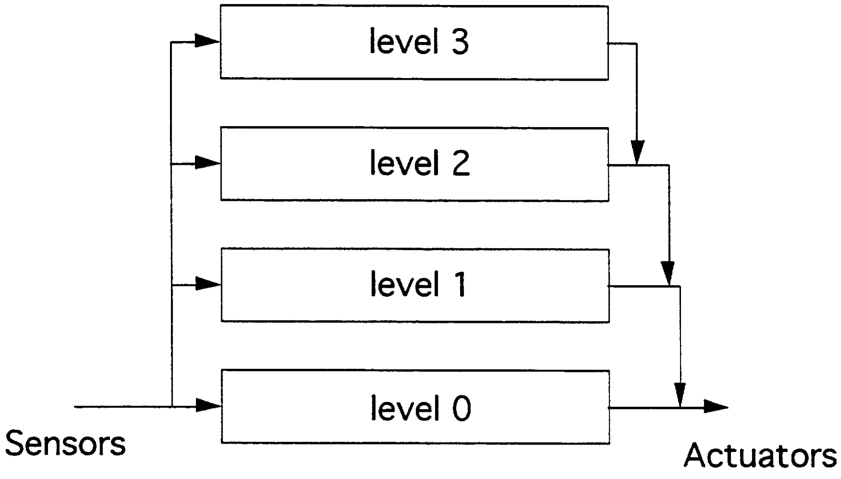


Figure 3-2. Control Layout for a Subsumption Architecture

The sensory data is assimilated at varying levels of complexity; however, unlike the horizontal decomposition, each level of assimilation and planning can be bypassed by the other levels. Vertical decomposition also allows for the possibility of using Command Fusion instead of Sensor Fusion. Command Fusion lacks the generality normally given by Sensor Fusion, but it does allow the control system to exploit low level sensor data for immediacy. This ability to quickly access sensory data allows for reflexive control of the robot. The vertical approach leads to greater flexibility in the architectural design of the control network. Since each layer is connected in parallel, the altering of one assimilation/planning module will have very little affect on the other modules. Not only does this make the system easier to modify but it also gives the ability to connect different assimilation/ planning modules without altering the overall system design.

3.2. Brook's Subsumption:

In 1986, Brooks [3] theorized that the control system of a mobile robot could be divided into different layers of task-achieving behaviors. With each additional layer, the robot achieves a greater sense of competence. The interaction between each layer is very important to the control of the robot. The control system must be responsive to high priority goals, while still servicing necessary "low-level" goals.

This concept of multiple goals is considered one of the four main requirements of an autonomous mobile robot's control system. The next requirement revolves around the use of multiple sensors on the robot. The control system must be able to make decisions based off the outputs of the various on-board sensors. Since some of the sensors may overlap in the physical quantities that they measure, inconsistent readings are possible and the control system must be able to determine which reading, if any, is correct. Also, many sensors' outputs do not have a direct correlation to any physical quantity. The third requirement for the robot's control system is robustness. The control system should be able to handle the failure of sensor systems without a complete loss of the control system's integrity. Finally, extendibility is important for the continued evolution of the robot's control system. Extendibility defines the ability to add more sensors and capabilities to the robot without impairing the speed and performance of the control system. Extendibility can normally be achieved by three basic methods:

- Utilizing previously unused processor power
- Upgrading the processor to achieve a faster overall system
- Adding more processors to carry increased load

The idea of subsumption bases each layer of control off of a level of competence. Traditionally the level of competence of a robot has been divided into five main pieces: sensing, mapping, planning, task execution, and motor control. This hierarchical method requires that each of these items exists to some extent before the robot can move. Any change in a single piece may affect the pieces adjacent to it and require a complete redesign of the robots control architecture. This is regarded as a decomposition of the problem into horizontal slices. Brooks theorized that it is better to decompose the problem vertically where each vertical slice is a different level of competence. These levels of competence were defined as:

0) Avoid contact with objects
1) Wander aimlessly around given environment
2) Explore the given environment
3) Build a map of environment and path plan
4) Monitor changes to the environment
5) Identify objects and carry out tasks based on these items
6) Plan changes to the state of the given environment
7) Reason about behavior of objects in environment

Layers of the control system would be built to correspond to these levels of competence. The robot is initially designed to fulfill the requirements of a zero competence level. The next layer implements a competence level of one. This layer is able to examine data from the level 0 system and is able to suppress level 0's normal data flow. This process is repeated through all the levels of competence and each new control layer subsumes the roles of the lower levels. Hence, the term subsumption architecture. Robustness is achieved by the fact that all the control layers are setup in parallel with each other (see Figure 3-2). The obvious way to handle the problem of extendibility is by giving each layer its own processor.

Each of these processors acts as a finite state machine that runs asynchronously from each other. Because there is no handshaking between the processors, it is possible to lose messages. There is no other form of communication between processors and there is no shared global memory within the system. Input to the modules can be suppressed and outputs can be inhibited which allows for subsumption by higher level layers.

3.3. Payton's Reflexive Control System:

Payton's [4] architecture was designed to take advantage of both the immediate and assimilated data provided by the robot's sensors. Payton argues that the data from the sensors is only useful as long as it does not become obsolete before the control system can use it. This requires that the sensors' data is processed as quickly as possible. The challenges placed upon a control system by constantly changing terrain may be to overwhelming for the processing units unless appropriate tradeoffs between immediacy and assimilation are made.

Both assimilation and immediacy have a number of advantages and disadvantages. The assimilation of data allows for the enhancing the completeness and detail of a constructed world representation with added processing. This will allow for easier high-level planning by the control system. However, assimilation requires a great deal of time to obtain the results. In the time necessary to assimilate the data, the robot may actually have already collided with a potential obstacle. Immediacy allows for a quicker access time to the sensory data. The faster the sensor's output can be used the "more value it has for control." This faster access time to the sensor arrays and planning system basically allows the environment to act as the feedback signal to the control system. Also, the greater the immediacy a control system possesses the easier it is for the system to monitor unexpected changes in the environment. However, in unconstrained environments, robots that employ control systems based around immediacy tend to suffer limited operation. This is due to the fact that "immediate data loses value in loosely constrained environments." It becomes hard to maintain complete control within an environment where moving objects exist. For example, a robot is navigating through an environment where there are people walking in and out of its sensor range. With a reflexive control system, the robot will try to avoid these moving objects. Unfortunately, the robot will soon become disoriented. If all the robot is meant to do is wander throughout the environment and avoid obstacles, then use of immediacy will be fine. However, if the robot is supposed to achieve a specific goal, such as traveling to a particular point, the immediacy can lend itself to errors in the control system. Without extremely sophisticated sensors, it becomes difficult for the robot to maintain position information when reflexive hazard avoidance is being implemented.

Payton's reflexive architecture was designed around four major modules (see Figure 3-3). The Reflexive Planning module requires the highest degree of immediacy while the Mission Planning module requires the highest degree of assimilation. Since the modules are setup in a vertical decomposition approach, the control system is able to perform both immediate and assimilated tasks.

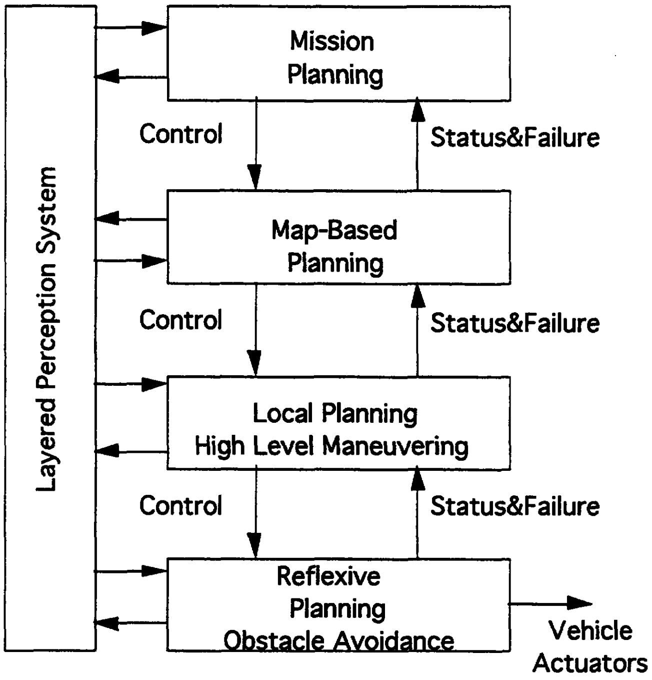


Figure 3-3. Reflexive Control System

The higher levels of the control system pass "constraints and specialization commands" downward while failure and status reports are passed upward by the lower level modules. Payton uses a "common blackboard" approach for communication between modules. When one module needs to communicate to another module, it writes its message to a "blackboard" that is accessible by the entire network. Each command issued to the blackboard is passed through a command arbitration unit which determines the priority of the message and issues them to the actuators. In a highly parallel machine architecture. Due to the vertical decomposition of the architecture, the addition of new modules will not alter the currently existing system to any great degree. Unlike Brook's architecture, the vertical layers are based off of assimilation/ immediacy instead of level of competence. To actually increase the level of competence of Payton's control system would require the addition of both reflexive modules as well as high-level planning modules.

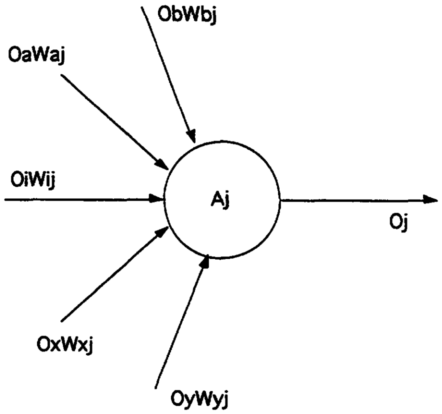
Also, a higher level of competence may require modifications to existing modules.

3.4. Rosenblatt and Payton's Fine Grained Architecture:

Both Rosenblatt and Payton [5] believed that Brook's subsumption architecture [3] advocated principles of system design crucial to the creation of a robot controls system. Like Brooks, they designed their system around the idea of vertical decomposition. However, they also believed that Brook's architecture possessed major practical limitations.

Interestingly enough, one of the major flaws that Rosenblatt and Payton found with the subsumption architecture was the fact that one command could subsume another command. They argued that whenever one command totally inhibited another command all of the data contained by the subsumed command was completely lost. Due to this loss of data, there was no guarantee that the behaviors will be able to perform the functions that were expected of its level of competence. Another drawback evolved from implementing each layer of competence with finite state machines. They argued that these finite state machines possessed internal states that could not be accessed by the other layers. With the addition of each new layer of competence, the demands on the lower layers increased and the need arose to modify the behaviors of the existing layers. These modifications were made impossible by the internal state of the finite state machines.

Rosenblatt and Payton's solution to these problems was to create a system designed around fine-grained behaviors. Each behavior was divided into a collection of very simple decision-makings units. By making them fine-grained, no unit possess any internal state that could not be accessed by the external network and each unit represented a specific concept for the behavior of the robot. Instead of inhibiting commands, they decided that a weight would be applied to each command. Accordingly, no data was ever lost which allowed the system access to all sensory data to make a decision. Where Brook's architecture would have subsumed a command from a lower level of competence, Rosenblatt and Payton's architecture would just attach a lesser weight of importance on a lower layer command. Figure 3-4 shows a behavioral network unit.



Oi is output of unit i
Wij is weight on the link from unit i to j
Oj is output of unit j

Figure 3-4. A Network Unit

All of the inputs to unit **Aj** come from similar units throughout the network. A weight is placed on the links connecting the units together. As can be seen, no data is lost from one unit to another. Each unit takes in its weighted inputs and performs any necessary functions on those inputs. The unit then outputs the results of its processing and the cycle continues throughout the network. Payton and Rosenblatt concluded that their system allowed the selection of commands that best meet the demands of all the system's behaviors.

TRADITIONAL CENTRALIZED CONTROL NETWORKS CHAPTER 4

4.1. MITy Prototype Series: A Direct-wired Transducer Platform:

The traditional design of a mobile autonomous robot revolves around a single central control unit (CCU) that has a central processing unit (CPU) to perform all the sensor fusion, mapping, and planning functions. The CCU normally possesses multiple digital and analog I/O pins. This was the architecture used in the design of the Unmanned Vehicle Lab's MITy prototype series. As Figure 4-1 demonstrates, the architecture resembled a star formation where the CCU is the center of the star and the various sensors and actuators are the rays of that star.

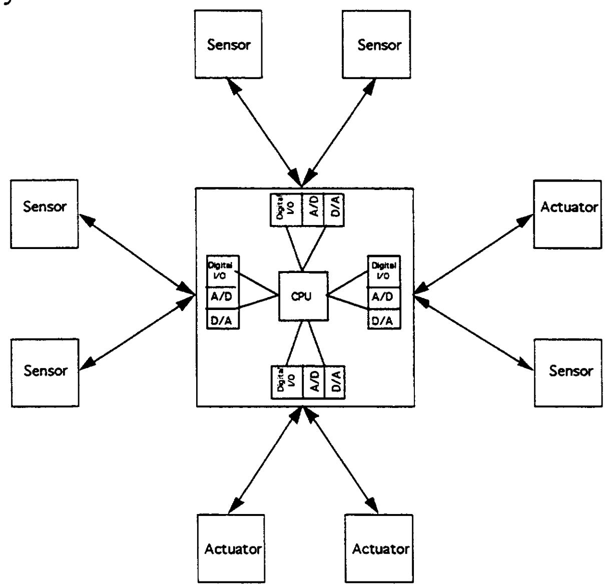


Figure 4-1. Traditional Centralized Processing Architecture

Each transducer is wired directly back to the CCU's I/O pins. As previously mentioned (see Section 2.2), the MITy series used a Zilog Little Giant board as its CCU board. This board possessed a Z180 microprocessor and multiple digital and analog I/O ports. The support and signal conditioning circuitry necessary to allow for more efficient data transmission and easier interfacing between the transducers' I/O and the CPU's I/O was placed out at the transducers.

There are a number of advantages and disadvantages to this architecture. The use of a single CCU board was an inexpensive alternative to having multiple control units. Also, since the hardware and low-level software were geared mainly toward interfacing the transducers to the CCU, the design was fairly straightforward and uncomplicated. Another advantage was that since each transducer was a separate ray of the star, a transducer failure would not cause a catastrophic failure to the electrical system. However, in many regards the disadvantages outweigh the advantages. The direct wired architecture of the MITy series was inflexible and consequently not very modular or expandable.

For example, an acoustic range finder used for hazard avoidance may require a power rail of 5V, two digital I/O lines and possibly some sophisticated signal conditioning circuitry. If a decision was made to upgrade the hazard avoidance sensor to the laser range finder found on MITy-2, then a power rail of 12V would have to be run in addition to the 5V rail, another digital I/O line would have to run, and the signal conditioning circuitry would definitely change. All of these changes can create a logistics nightmare for the layout of the electrical system. First, all the digital I/O lines may already be committed. This means that circuitry must be designed or bought to extend the digital I/O of the CCU. Second, since there may be no space left on the connectors that interconnect the various components of the robot, mechanical alterations to the robot's frame may become necessary to accommodate more connectors. These mechanical changes may require subsequent changes to the electrical system. This domino affect ends up wasting time and resources and usually brings R&D efforts to a stand still.

The low-level software used to control the sensor must be completely rewritten as well since the software-hardware interface for the sonar and laser is completely different. Altering the low-level software may also require changes in the robot's higher-level control code and now software begins experiencing the same problems that were experienced in hardware.

Another drawback of the MITy series architecture was the inability to carry out parallel development of transducers. The only way to achieve parallel development was by completely laying out the robot in advance. This task included: delegating I/O resources to all the transducers; determining the size, shape, and configuration of each transducer on the robot; and routing the necessary power lines to each transducer. Once the robot's layout was finalized, it became difficult to implement any changes.

The limitations experienced with the MITy series permeate all robot designs that utilize a direct-wired transducer platform. The time and the cost necessary to change the layout of the system make this architecture non-versatile and extremely inefficient. The inability to easily upgrade the transducers on the robot guarantees that the system will quickly become obsolete.

4.2. MITe Prototype Series: A Bus-based Transducer Platform:

The MITe series was an attempt to improve upon the MITy architecture. Though the MITes still utilized a single CCU to assimilate the sensor data and to carryout all the higher level control code, a bus-based architecture was introduced to minimize the disadvantages experienced by the MITy series. The signal conditioning circuitry and interface electronics for each transducer were built onto circuit cards that plug into the robot's 8-bit data bus. The MITe possessed eight expansion slots that were accessible by a 3-bit address bus. This allowed access to a minimum of eight different transducers. By allowing each circuit card to support multiple transducers, the MITe could support more than eight transducers. Therefore, unlike the MITy series where the number of transducers was dependent upon the number I/O ports on the CCU, the number of transducers on each MITe is directly related to the size of the support circuitry required by each transducer. Figure 4-2 is a block diagram of the MITe architecture.

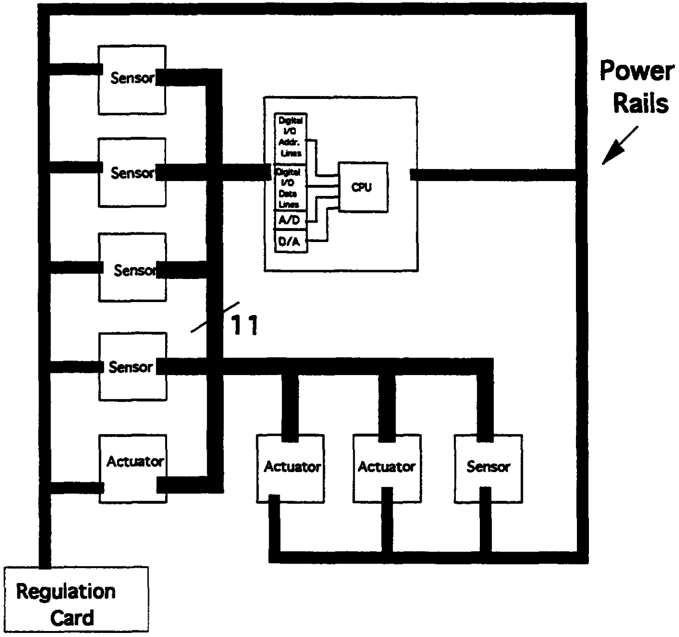


Figure 4-2. MITe Prototype Bus Architecture

Bus architecture gave the MITe series significant advantages over the MITy series. These advantages were mainly due to the standardization of the hardware interface guaranteed by the bus. Each transducer has access to the same digital and analog I/O lines and all the voltage rails. This made the robot's hardware easier to maintain, re-task, and upgrade. It also permitted parallel development and off-line testing of transducer circuitry. However, the bus-based architecture did have some disadvantages that the traditional direct-wired system did not. The bandwidth of the bus was a major limiting factor on how often the transducers could be accessed by the CPU. Also, timing issues became very important to guarantee that the correct information was being accessed at any one time. Though the hardware saw improvements due to the standardized bus, the hardware-software interface had become extremely complex to deal with the transmission of data across the bus. The bus architecture was also susceptible to catastrophic failure due to a transducer failure. For example, if a transducer's bus-card shorted two of its digital bus lines to ground, then the bus would be corrupted making it impossible access any of the other transducers. Like the MITy series, the MITes could suffer from a catastrophic microprocessor failure as well as the fact that the CPU was still loaded down by sensor fusion.

4.3. Companion Prototype:

As previously mentioned (see section 2.4), the removal of the size constraints that had been imposed upon the planetary micro-rover and the inundation of a substantial amount of IR&D funding allowed for the design of a far more sophisticated robot than the MITy or MITe series. Using the knowledge gained through testing the MITy prototypes, it was determined that the Companion should possess some form of distributed processing. The use of multiple processors would alleviate the problems normally associated with trying to carryout sensor fusion and path planning on the same CPU. As Figure 4-3 shows, the Companion prototype had many of the same attributes as the MITy and MITe prototypes.

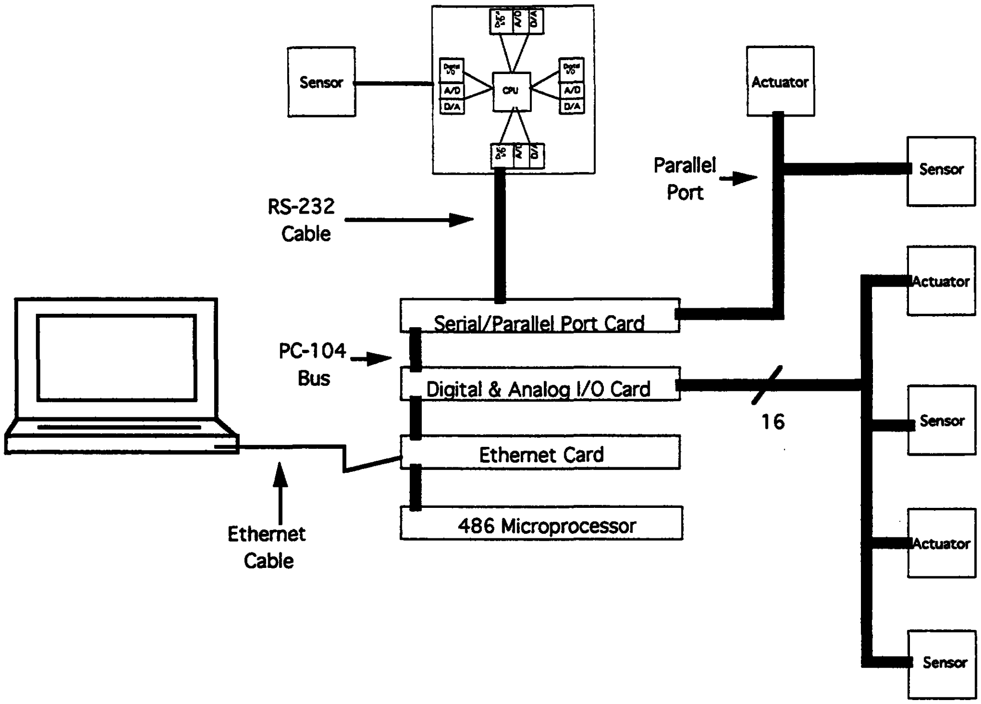


Figure 4-3. Companion Multiple Processor Architecture

Companion's control system was divided between the processing of the sensors' data and the path planning that was carried out after the data had been assimilated. All the sensor fusion occurred on the 486 microprocessor board. This board used a PC-104 bus interface. This allowed for additional peripheral boards to be stacked onto the microprocessor card. Purchasing off-the-shelf computer cards gave an added benefit since software drivers were normally included to handle the interface between the processor and its resources. With the MITe architecture, a custom interface had to be designed to allow the processor to access all the transducers. The commercially supplied drivers greatly simplified the development time for the hardware-software interface on Companion. Figure 4-3 only shows the more common peripheral cards that were used with the Companion prototype. The PC-104 based microprocessor board communicated to a 486 laptop via an ethernet cable. The higher level control code was performed on the laptop. Though full assimilation of sensor data was necessary for control, the utilization of multiple CPUs allowed for more efficient processing. As with the previous prototype series, the hardware signal conditioning occurred out at the transducers. The transducers were interfaced to the PC-104 microprocessor board via the I/O bus provided by the Digital and Analog I/O card. As with the MITy architecture, parallel development of both hardware and software was only possible after the I/O resources had been delegated for each transducer. However, since more than one I/O card can be placed on the PC-104 stack, the robot's transducer suite can easily be expanded. This expansion was limited only by the number of peripheral cards that the microprocessor can support. As additional cards were added to provide GPS and ethernet capabilities, this limitation became a concern. One method to deal with this limitation was to add an additional microprocessor to deal with new transducers. A Zilog Little Giant board was connected to the PC-104 based 486 board via a RS-232 line. The Little Giant was then used to interface a gyro into the Companion's navigation system. This was UVL's first implementation of a smart sensor (see Chapter 5).

Though using commercially available components allowed for a more robust and sophisticated robot, their cost was much higher than the circuits designed by the UVL graduate students. Also, as with the MITy and MITe series, the robot could still suffer a catastrophic processor failure. If either the PC-104 based 486 board or the laptop suffered a failure, the robot would be unable to recover. The ability to upgrade was limited by the fact that new transducers required a re-distribution of I/O resources. Not surprisingly, the fact that most of the electronics were bought commercially and not designed in-house led to a limited understanding of the overall hardware system. Consequently, this made the robot harder to maintain and debug.

4.4. Summary of Advantages/Disadvantages of UVL Systems:

As demonstrated above, all the UVL systems have their own advantages and disadvantages. All share common problems extending from the areas of sensor fusion, catastrophic processor failures, and limited modularity/upgradability. It was important to summarize these points to allow for a better understanding of the benefits of the proposed smart sensor system. The following list gives a breakdown of the various UVL systems:

Direct Wired Transducer Platform (MITy Series):

- Single CPU for all transducer processing
- Hardware signal conditioning carried out at transducer
- Transducers directly wired to I/O pins on CPU board
- Full assimilation of sensor data necessary for control

Bus Based Transducer Platform (MITe Series):

- Single CPU for all transducer processing
- Hardware signal conditioning carried out at transducer
- Transducer circuitry built on cards that plug into I/O bus of CPU
- Full assimilation of sensor data necessary for control

Hierarchical Control Architecture with Multiple Processors (Companion):

- Independent CPU used for all Sensor Fusion
- Independent CPU used for Path Planning and Mapping
- Hardware signal conditioning carried out at transducers
- Sensor Fusion CPU and Planner CPU connected via ethernet
- Full assimilation of sensor data necessary for control; however, multiple CPU's allow for more efficient processing
- Independent CPU used for processing of Gyro information

Table 4-1. UVL System Breakdown

Table 4-1 gives a summary of the advantages and disadvantages of each system.

UVL SYSTEM	ADVANTAGES	DISADVANTAGES
MITy Series	simple to design	non-modular
	inexpensive	non-upgradable
	sensor failure is non-catastrophic to system	difficult to debug
		serial development
		limited I/O resources
		catastrophic processor failure
		processor loaded down by Sensor Fusion
MITe Series	more modular than MITy series	catastrophic sensor failure possible
	retaskable/maintainable	bus width limitations
	parallel development	catastrophic processor failure
	off-line test/debug	processor loaded down by Sensor Fusion
		modularity still limited
Companion Series	efficient use of processing	expensive
	hierarchical layer of abstraction	difficult to debug
	limited parallel development	catastrophic microprocessor failure
	sensor failure is non-catastrophic to system	difficult/expensive to maintain
	more modular/upgradable than MITy and MITe series	limited modularity
		limited upgradability

Table 4-2. Advantages/Disadvantages of UVL Systems

As can be seen from Table 4-1, the disadvantages outweigh the advantages of each system. The greatest concern revolves around the difficulty of upgrading the robot's sensor and actuator suite. These limitations make the robots hard to maintain and therefore hard to market to perspective buyers. The need to design a more efficient and maintainable robot led to the UVL's initial research into the area of smart sensor technology.

SMART MODULES
CHAPTER 5

5.1. Overview of Smart Sensors:

After analyzing the list of the advantages and disadvantages of the UVL systems (see Section 4.4), it was concluded that a more robust and efficient system could be designed if it possessed the following attributes:

- distributed processing to allow for better assimilation of sensor data,
- a predefined hardware interface that would allow for parallel development of transducers,
- transducers should be connected by a data bus that would allow for easier modularity and upgradability,
- transducers should have plug-and-play capabilities to increase system testability and maintainability,
- the failure of any one transducer should not cause a catastrophic system failure,
- and the overall system should be simple and inexpensive to design.

It was already known that the robot's control architecture closely resembled the control systems implemented in test and measurement applications. A central controller normally directs the actions of instruments, sensors, and actuators; polls for any results; and manages the resulting data.[6] This knowledge led to research into how other applications, that utilized a similar control architecture as the robots, dealt with the drawbacks of traditional direct-wired centralized control. It was found that the instrumentation and control system industry had determined that the use of specialized sensor buses was the necessary solution to obtain a more efficient, low cost, high quality product.[7] The idea behind these sensor buses was to turn the traditional control system architecture into a network where the transducers act as nodes.

Accordingly, the nodes need a certain amount of intelligence to allow interface with the sensor network. Nader Najafi of MRL/Sensors coined the term 'smart sensor'. Najafi determined that a smart sensor should: possess all analog and digital signal processing; digitize analog transducer outputs; have a bi-directional bus for communication; contain a specific network address to allow user access and identification; and be able to receive and execute commands over a digital bus.[8] John Eidson and Stan Woods, both of Hewlett-Packard, designed a research prototype of a networked smart sensor system. They defined the hardware design of a smart sensor as including:

- a communication transceiver module for connection to the physical communication medium,
- a common core module that will configure the node, convert transducer signals to digital data with standard units, and manage the flow of data into and out of the node,
- and a transducer interface module that signal conditions the transducer output.

This hardware design has become the industry standard for a smart sensor.[6]

As Figure 5-1 shows, the three major blocks of the smart sensor architecture are the Communication Media Access block, the Control and Configuration block, and the Transducer block.

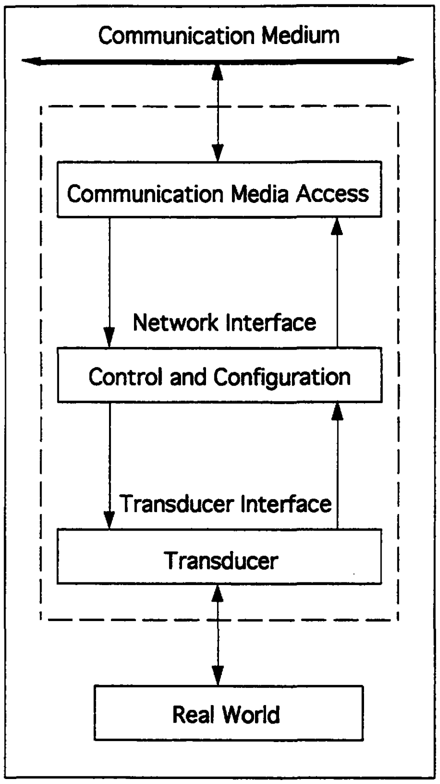


Figure 5-1. Smart Node Architecture

The Communication Media Access block is responsible for handling the low-level protocol between the node and the physical medium. For many of the sensor networks on the market, this block is designed around a media-dependent transceiver. The Control and Configuration block acts as the data path between the application interface with the transducer and communication interface with the physical media. Unlike the Communication Media Access block, this data path, which is illustrated in Figure 5-2, is identical in all smart sensors.

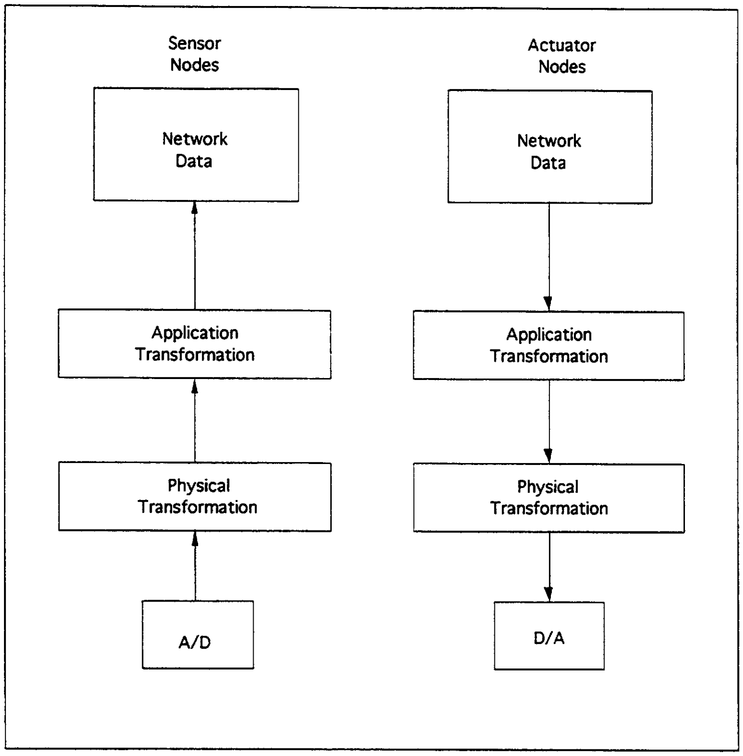


Figure 5-2. Data Path

The importance of the Control and Configuration block lies in the fact that it is responsible for converting the transducer's output to a usable format. The Physical Transformation converts between the digital representation in SI units and the transducer's raw digitized data. The Application Transformation converts between the SI unit representation and the application representation that appears on the network.[6] The application transformation can be as sophisticated as the designer wants; however, even when performing the simplest application transformations, the smart node eliminates much of the processing load normally placed on the central controller due to data digitization, filtering, and signal processing.

Tom Ormond, Senior Technical Editor of EDN, confirms that a smart sensor should consist of a transducer combined with signal conditioning and other circuitry to produce data signals that have been logically operated upon to increase their value of information.[8] Ormond hypothesizes that the simplest smart sensor is composed of a transducer with signal conditioning circuitry to filter and scale the output. Using this definition, the transducers implemented on the UVL prototypes can be considered smart sensors. However, as was shown by the earlier research of the UVL, an efficient and cost-effective control system can not be designed using this basic model of a smart sensor. A higher and more effective level of sensor intelligence can be achieved by adding communication capabilities. This includes the addition of A/D and D/A converters at the sensor and actuator nodes respectively, as well as the addition of transducer addressability to allow user access of individual nodes. These features allow for the use of a bi-directional digital bus to interconnect the various transducer nodes and the central controller. Consequently, the computational load on the central control processor will be lessened since the incoming data will be in a predefined digital format.

By placing a microcontroller at the sensor or actuator, a more effective and useful node can be achieved. Varying levels of intelligence can be attained by simply altering the code running on the microcontroller. At the simplest level, the microcontroller can be programmed to carryout all the necessary computations to turn the transducer's raw data into SI units. The microcontroller can also be programmed to monitor the output of the sensor and only transmit the data to the central controller after certain criteria have been met. For example, constantly transmitting the output of a gyro node being used for navigation would be an ineffective use of the network's bandwidth. Instead, the node can be programmed to sample the gyro's output and only transmit data to the central controller periodically. Alternately, more advanced code could be run on the node's microcontroller to transmit the gyro's data only when a significant enough change has occurred to designate an actual course change. By programming all the nodes in this manner, the control architecture can make the best use of the bus' bandwidth.

There are a number of properties that should be met when designing a smart sensor system. As shown in Table 5-1, Eidson and Woods divide these properties into transducer-related, measurement-related, and application-related properties.[6]

Transducer-Related	Measurement-Related	Application-Relate
Physical variable	Timing management	Changing measurement properties by accepting network messages
I/O format	Data management	Communication Pattern
Calibration	Computational characteristic	Managing communication properties
Identity	Identity	Synchronizing node clocks
Operational ranges	Location	Control Models

Table 5-1. Smart Node Properties

It must be understood that for a node to possess these properties it must have a microcontroller. Most of these properties are dependent upon the code running on the microcontroller. Many of the smart sensor networking tools on the market have prewritten protocols that deal with the time management, I/O format and network communication properties. The physical variable: distance, temperature, etc., identity, calibration, and operational ranges are all properties provided by the application program written by the user. By adhering to these design rules, a group of smart nodes can be created and connected into a control system that far exceeds the control architecture on the various UVL prototypes.

5.2. Smart Modules -- A Flexible Control Architecture:

As mentioned above, the reason that UVL began researching the area of smart sensors was to find a system that did not suffer from the same drawbacks as the traditional central control networks. Many of the disadvantages experienced by these systems can be removed by using smart nodes designed around microcontrollers running communication protocol and user-written application code. By standardizing the hardware interface, using a bi-directional digital bus, and the transducer's data format, using SI units, a smart node system becomes modular, expandable, and upgradable. The same argument used to show the drawbacks of the MITy architecture can be used to demonstrate the advantages of a smart sensor system. In Section 4.1, the difficulties surrounding the decision to change from an acoustic range finder to a simple laser range finder were discussed. An entire reworking of the electrical hardware as well as the high level and low level software was required. By using smart nodes, these problems can be avoided. First, since the interface to the bus is the same for all the transducers on the network, there is no reason to rework any of the system hardware. Second, because both devices output the distance to an obstacle and both nodes can be programmed to output their data in meters, the system control code does not have to be changed. The central controller is only expecting to receive some type of obstacle avoidance information across the network. It does not matter that the data comes from an acoustic range finder or a laser range finder. Also, since the bus is run throughout the robot, the only problems that might arise from adding or moving a node are spacing issues.

There are also a number of other advantages for using smart nodes interconnected by an intelligent communication interface. First, the issues involved with developing an entire control system are simplified by the ability to carry out the parallel development of all the system's smart nodes. This is due to the standardization of the nodes' hardware and software interfaces. The only information required by the individual design teams would be the form of the output expected from the node and the input commands that the node will receive. For example, two separate design teams might be responsible for the gyro and laser range finder nodes. The development of the gyro node will not depend at all upon the design decisions of the range finder design team. All the gyro design team needs to know is that their node will output the relative direction of the robot in radians and that the node will receive commands from the central controller dealing with gyro self-calibration, power cycling, and requests for gyro data.

The use of smart nodes adds a level of abstraction between the control code running on the central processor and the transducers. With the traditional architecture, the control code running on the central processor makes calls to device drivers that control the interface with the transducers. These device drivers are an integral part of the hardware/software interface of the overall system. The code that makes up these device drivers is dependent upon the transducer being controlled. The device drivers must change when the transducers are upgraded. These changes to the device drivers usually require a change in the control code running on the central processor.[9] The additional level of abstraction offered by the smart nodes allows for the transducers to be upgraded or modified without any major change to the control system. For example, to determine the distance to a given object, the central controller could send out the request *Distance_to_Object* onto the network. The control system is not concerned with whether or not the range finding device is designed around a laser or a sonar. In many ways, smart sensor technology can be considered the hardware networking equivalent of object-oriented programming.[10] By abstracting away from the transducer interface, the control architecture is able to achieve a level of modularity unattainable by the traditional architectures investigated by UVL.

It is important to summarize the various advantages and disadvantages of the smart sensor systems. Table 7-1 lists these various advantages and the single known disadvantage of the smart sensor network.

ADVANTAGES	DISADVANTAGES
easy to design	bandwidth limitations
parallel development	
easy to maintain	
nearly unlimited upgradability	
distributed/local/efficient microprocessors	
fault tolerant/decentralized processing	
retaskable	
true peer-to-peer control network possible	
different communication mediums possible (i.e. twisted pair, fiber optic, RF, IR)	
application program is media independent	

Table 5-2. Advantages/Disadvantages of Smart Sensor System

Many of the advantages have already been mentioned; however there are a number of important points that have been overlooked so far. The first point is that fault tolerance can be achieved by use of smart nodes. Two ways by which fault tolerance can be implemented in a smart sensor network are by:

1. connecting the nodes in a ring topology to offer alternate paths among the modules in case of a physical break in the bus,
2. redundancy of smart nodes to guarantee that a single failure will not affect the system performance.

This level of fault tolerance is not readily achievable with a traditional direct-wired architecture where redundant modules would either require additional I/O resources or additional hardware to allow access to the redundant nodes.

Another important point revolves around the advantage of distributed/local/efficient processing. The previous discussions concerning the attributes of a smart module system have dealt mainly with easing the processing load on the central controller by conditioning the transducers' data into a more usable format. However, the fact that each node contains a microcontroller can be exploited to design a completely distributed control network where a central controller is unnecessary. Networking protocols can be purchased the support true peer-to-peer communication (see Chapter 7).

The only significant drawback of using a smart sensor network originates from the limitations on the network's bandwidth. To allow successful control of the robot, all of the nodes need to be able to inter-communicate without bogging down the network. To effectively implement a distributed control system, the networking protocol needs be optimized for the transfer of the small packets that control information normally consists.[11] Once again, specialized communication protocols come to the rescue. Support of both acknowledged and unacknowledged transport mechanisms can be used to avoid saturating the network.[6] As will be shown in Chapter 6, a number of corporations have begun developing network protocols especially designed with distributed control in mind.

OPERATING NETWORK PROTOCOL CHAPTER 6

6.1. Sensor/Actuator Buses and Communication Protocols:

One of the most important aspects of a Smart Module System is the network used to interconnect the various modules. A control network replaces the complex wiring from device to device, and a network management tool defines how the devices in the network interoperate. The necessary components for a distributed control system are the communication protocol and the hardware support set. Normally, each distributed control network is viewed as a unique problem that requires a custom communication protocol and custom hardware. This is neither practical nor economic when discussing robotics systems.

Local Area Networks (LANs) have allowed for the design of distributed control networks that are not application dependent. However, LAN based systems can be very expensive. Since each LAN node requires a computer to run the LAN software the minimum cost would be approximately $1000. Most robotics systems are required to be small and inexpensive. Neither criteria would be achievable through the use of a LAN. Also, LAN protocols are optimized for transferring large amounts of data and have no guaranteed maximum delay time for successful transmission of data [11]. The control information sent across the network will consist of small packets. A control system also should be reliable, repeatable, and predictable. Consequently, LAN based systems are not optimized to be used for distributed control networks.

Approximately a decade ago, the industrial measurement and control market began looking for alternatives to centralized control strategies. Design engineers began studying the possibility of creating inexpensive standardized I/O networks that would allow for the quick and easy interconnection of sensors and actuators in the manufacturing environment. There are now a number of different sensor bus protocols used on the market today, including Seriplex, LonWorks, DeviceNet and SDS. Seriplex, which was designed by Automated Process Control, Inc. (APC), makes use of a specialized ASIC chip and does not require any additional controllers to run the protocol. Seriplex requires simple tools to implement a network. The ASIC can be programmed by a specialized hand-held device provided by APC. Both Honeywell Microswitch's Smart Distributed System (SDS) [13] and Allen Bradley's DeviceNet are based off Controller Area Networks (CANs).[14] The CAN architecture was initially designed by Bosch GmbH as a control system for Mercedes-Benz's vehicles. DeviceNet and SDS both require emulators, evaluation boards, and compilers to create the nodes and setup the network. Though the CAN market is maturing, it currently has not achieved a state where the tools are efficient, user-friendly, or abundant. Echelon's LonWorks system is a multi-industry, multi-media, peer-to-peer control network. It is supported by a number of user-friendly Window's based software tools that allow for the creation and simulation of smart nodes and the setup of an entire network. Though currently under development, SDS and DeviceNet have not yet achieved peer-to-peer capability. This makes LonWorks the only commercially available peer-to-peer technology. The ability to implement a peer-to-peer network greatly enhances the idea of a distributed control architecture. This is the main reason that UVL chose the Echelon protocol over SDS and DeviceNet.

6.2. LonWorks Architecture:

6.2.1. Smart Networks:

Echelon proposed creating a local operating network (LON) that would interconnect intelligent devices known as nodes. The network communication is supported by the LonTalk protocol (see Section 6.2). Figure 6-1 shows a generic configuration for a LonWorks Network. The network supports up to 32,385 nodes.

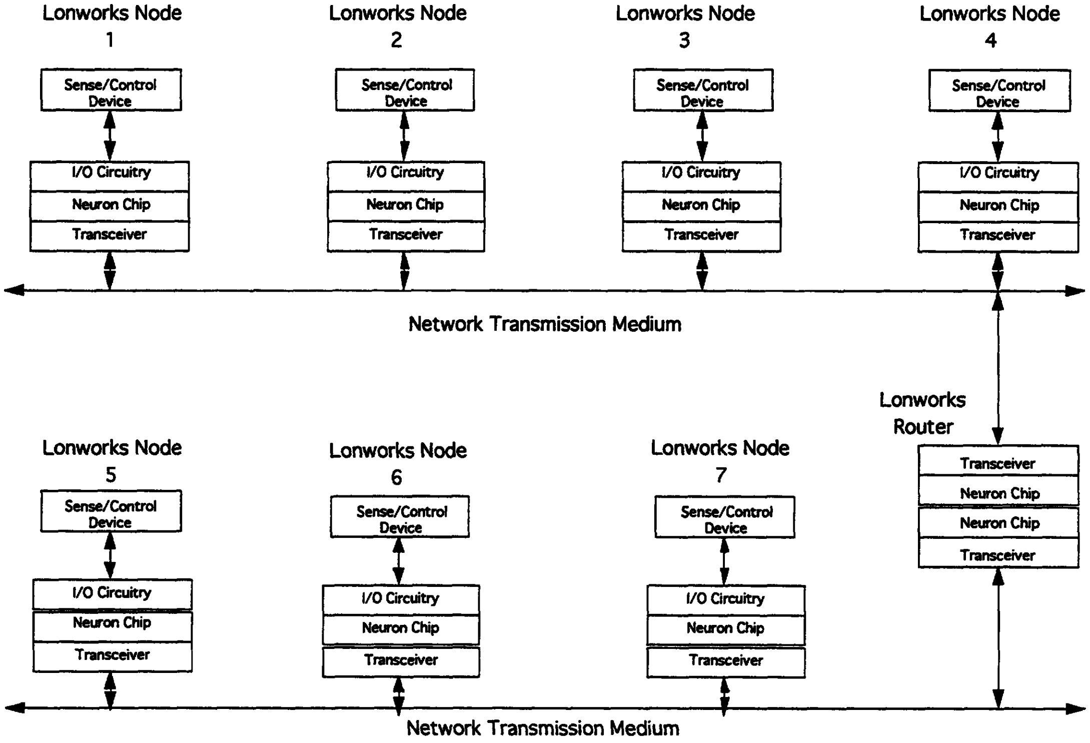


Figure 6-1. LonWorks Network

Routers and gateways are used to create sub-networks to allow for more efficient use of the bus bandwidth. Communication between the nodes is carried out using Network Variables or explicit messages.

6.2.2. Network Variables and Explicit Messages:

Echelon introduced a derivative of ANSI C known as Neuron C. The major modification is that Neuron C supports network variables. Network variables may be specified as either Input or Output objects. When a node assigns a value to a network variable defined as an Output object, the value is propagated across the entire network and any node specifying that network variable as an Input object reads the new value off the bus. For example, in an autonomous submarine, one of the smart sensors might be a pressure sensor to determine if the submarine has a leak. This node would have an output network variable, *sub_pressure* that contained the pressure sensed by the node. Every time the node measures the pressure the value is updated and propagated across the network. An emergency inflation device may be another node on the submarine. One of its input network variables would be the current pressure within the submarine, *current_pressure. Sub_pressure* and *current_pressure* are bound together by the network manager. If a leak occurs, then the pressure within the submarine would begin to drastically change. The pressure sensor would measure this change and send out *Sub_pressure* across the network. The emergency inflation device would read in *current_pressure* and activate.

The use of network variables vastly simplifies the creation of the network. Once the input and output network variables are bound by the network manager, the user does not need to worry about low-level details such as node addressing or request/response/retry processing. All of these issues are handled by the LonTalk protocol. An array of up to 31 bytes of information may be sent as a network variable. If an object larger than 31 bytes needs to be transmitted, then explicit messages may be used. Application programs can create messages of up to 229 bytes of data. To use explicit messages, implicit address connections called message tags must be attached to the data by the user. Consequently, explicit message passing is more difficult to implement than network variables.

6.2.3. LonWorks Smart Nodes:

The LonWorks nodes support local processing and possess I/O hardware to allow for monitoring and control of sensors and actuators. Each node contains the LonTalk protocol in firmware that allows communication between devices on the network. A LonWorks node has three main components: the microcontroller, the transceiver, and the user electronics. The Motorola/Toshiba Neuron Chip is the microcontroller that supports communication, control, scheduling, and the I/O interface. The transceiver provides the interface between the node and the communication medium. The user electronics depends upon the application and the transducers being interfaced with the Neuron Chip's I/O.

Since the LonTalk protocol running on the Neuron Chip is media-independent, the network has been designed to support a variety of communication media. Table 6-1 lists transceiver types and their data communication rates. The wide range of transceiver types gives an added flexibility to the LonWorks plug-and-play capabilities. It also allows for easier test and development of a network. For example, a network can initially be designed using Twisted Pair (78 kbps) as the medium and can then be refit

Transciever Type	Data Rate
EIA-232	39 kbps
Twisted Pair	78 kbps
Twisted Pair	1.25 Mbps
Power-Line	5 kbps
Power-Line	10 kbps
RF (300 MHz)	1200 bps
RF (450 MHz)	4800 bps
RF (900 MHz)	9600 bps
IR	78 kbps
Fiber Optic	1.25 Mbps
Coaxial	1.25 Mbps

Table 6-1. Echelon Transceiver Types

with IR (78 kbps) transceivers without any change to the network's software. The only limitation on changing the network's communication medium lies with the data rate. A control system designed to run with Twisted Pair (1.25 Mbps) as the medium could become ineffective if the medium was changed to RF.

As was previously mentioned, the size and characteristics of the user's application electronics are dependent upon the transducer being interfaced with the network. The application processor on the Neuron Chip (see Section 6.3.1) is normally capable of handling the user's application software; however, for transducers that require computationally intensive processing, a more powerful processor can be used. Though the Neuron Chip still controls the LonTalk protocol, all the application software is run on the new host processor. Figure 6-2 shows a block diagram of two possible configurations for a LonWorks node. The Host-Based configuration would be one possible method for implementing a Mapper/Planner node on the robot.

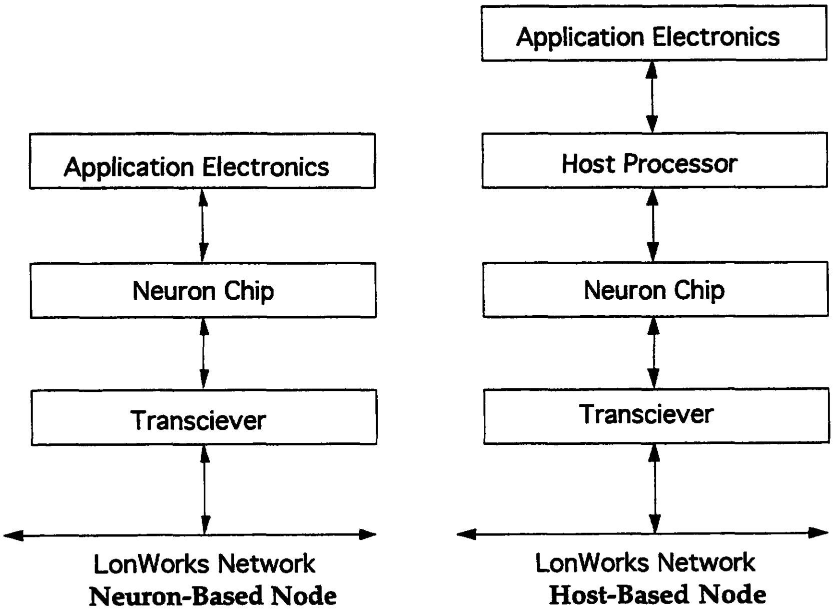


Figure 6-2. Block Diagram of LonWorks Node

6.3. LonTalk Protocol:

6.3.1. Open System Interconnection Standard:

The LonTalk protocol is designed to adhere to the International Standard Organization's (ISO) 7-layer open systems interconnection standard (OSI). Table 6-2 shows the mapping of the LonTalk protocol onto the OSI model. The OSI model defines the criteria for a standard data network.

	OSI Layer	Purpose	Services Provided
7	Application	Application Compatibility	Standard network variable types
6	Presentation	Data interpretation	Network variables, foreign frame transmission
5	Session	Remote actions	Request/response, authentication, network management
4	Transport	End-to-end reliability	Acknowledge and unacknowledged, unicast and multicast, authentication, common ordering, duplicate detection
3	Network	Destination addressing	Addressing, routers, gateways
2	Link	Media access and framing	Framing, data encoding, CRC error checking, predictive CSMA, collision avoidance, priority, collision detection
1	Physical	Electrical interconnect	Media-specific interfaces and modulation schemes

Table 6-2. LonTalk Protocol Mapping onto OSI Model

6.3.2. LonTalk Addressing and Routing:

An address will have a domain address component, a subnet address component, and a node address component. The domain of the network is a collection of nodes on one or more transport mediums, called channels. Multiple channels are connected through routers.

The LonTalk protocol supports the following four types of routers: a repeater, a bridge, a learning router, and a configured router. A repeater simply forwards all packets between the two channels. A bridge only forwards packets that match its domain addressing component. A learning router is able to learn the network topology at a domain/subnet level by monitoring the network traffic. A learning router is always updating its internal routing table. A configured router has a user-programmed internal routing table that will selectively route packets between channels. The throughput of the channel is dependent upon the transceiver's data rate. Typical channel throughputs for 12-byte packets are listed in Table 6-3.

Bit Rate	Peak Number of Packets/sec	Sustained Numbe of Packets/sec
9.766 kbps	45	35
78.125 kbps	400	320
1.25 Mbps	700	560

Table 6-3. Channel Throughput

Communication may only take place between nodes on the same domain. The domain component of the address can be 0, 1, 3, or 6 bytes long; however, the domain component adds overhead to every packet sent over the network. Therefore, use of a 6 byte domain component is not recommended. Each domain may contain up to 255 subnets with each subnet containing up to 127 nodes. Accordingly, the subnet number is 8 bits long and the node number is 7 bits long. Figure 6-3 shows the addressing hierarchy for the LonTalk Protocol. A node's physical location determines which subnet it is assigned.

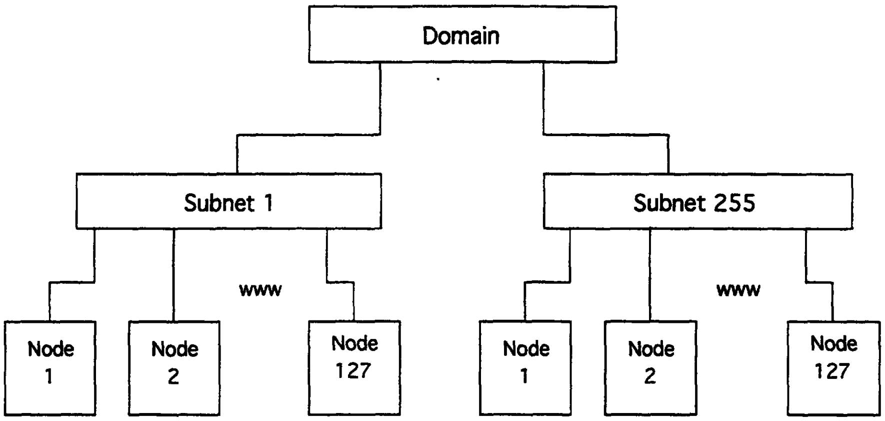


Figure 6-3. LonTalk Addressing Hierarchy

However, the location of a node often has little to do with the function of the node. It is for this reason that the LonTalk protocol supports group addressing. A node may be a member of up to 15 different groups. Group addressing allows efficient use of the network bandwidth through the use of one-to-many network variable connections. For example, a house may have 50 lights that are members of the same group. A single network variable called *dim_lights* can be sent out across the network using group addressing. Though all 50 lights have been dimmed, the network only experienced overhead due to the transmission of a single network variable instead of 50 separate network variables. The group address component is 1 byte long, so a domain may contain up to 256 groups.

6.3.3. LonTalk Communication Services:

The LonTalk protocol supports the following types of message services: acknowledged, request/response, unacknowledged-repeated , and unacknowledged. For acknowledged service, a message can be sent to either a single node or a group of nodes. Upon receipt of the message, all the receiving nodes send an acknowledgment across the network. If one of the nodes does not acknowledge receipt of the message, the sending node times out and re-tries the transaction to that node. The request/response service is just as reliable as the acknowledged service. With this service, a message is sent to a node or group of nodes and a response is expected from each receiver. The difference between acknowledged messages and request/response messages is that the responses may include data. With repeated-unacknowledged service, a message is sent to a node or group of nodes multiple times. The sending node does not expect any response from the receiving the nodes. The disadvantage of this method is the fact that if the message is not received then it is lost and the control system will have no way of knowing about it. However, this service is affective when the number of responses or acknowledgments may overload the network. Unacknowledged messages are the least reliable. They are sent only once and no response is expected from the receiving nodes. This service should only be used when the message that is being sent is not critical to the application.

The protocol supports a collision avoidance algorithm that allows the network to carry close to its maximum capacity under overload conditions. Accordingly, network throughput does not suffer degradation due to excess collisions. When a collision is detected, the protocol cancels the transmission of the damaged packet and then re-transmits it. Though the collision avoidance algorithm is transceiver dependent, all of the transceivers mentioned in this chapter support the collision algorithm.

LonTalk supports priority message passing. Only nodes that have been assigned a priority time slot by the network management device can transmit prioritized packets. When a node generates a priority packet, it is placed ahead of any non-priority packets in the transmission queue. Upon reaching a router, the priority packet will be moved to the head of the router's transmission queue. 126 levels of priority are supported by the protocol with priority 2 being the highest priority and priority 127 being the lowest. The protocols support of priority packets and collision detection permits bounded response time within the system.

Another feature of the LonTalk protocol guarantees security access to individual nodes. By supporting authenticated messages, nodes can prevent unauthorized access. 48-bit keys are distributed to the nodes on a network during installation. The sender and receiver of an authenticated message both must posses the same key. Upon receiving an authenticated message, a node generates a random challenge to the sender to provide authentication. The sender then carries out a transformation on the challenge by using its internal authentication key and the data from the original packet. The receiver carries out its own transformation on the challenge it generated and compares it to the sender's reply. Even if the challenge and response are intercepted, the transformation carried out on the challenge makes it difficult to determine the authentication key.

6.4. Motorola/Toshiba Neuron Chip:

Motorola has currently designed two VLSI devices to support the LonWorks operating network. These are the MC143150 and the MC143120. Both chips contain a limited amount of internal EEPROM and RAM. The main differences between these two chips is that the MC143150 does not have internal ROM but is instead able to access 64 Kbytes of external memory, while the MC143120 possesses 10 Kbytes of internal ROM but is unable to access external memory.

The Neuron Chip contains three 8-bit pipelined processors: one for Media Access Control, one for Network Protocol Control, and one for Application Device Control. Figure 6-4 is a block diagram of the MC143150.

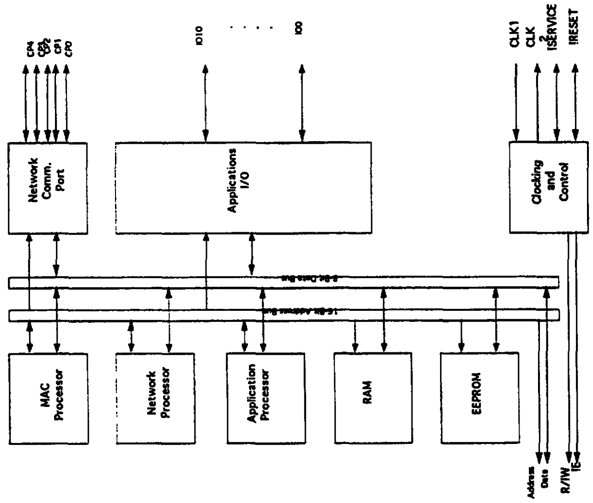


Figure 6-4. Block Diagram of MC143150

The MC143150 possesses control lines to access external memory. The block diagram for the MC143120 is identical except for the external memory control lines and an additional block to represent the on-chip 10 Kbytes of ROM.

The Neuron Chip also has selectable input clock rates ranging from 625 kHz to 10 MHz. The average on-chip RAM and EEPROM size is 2 Kbytes and 512 bytes respectively. As Figure 6-4 shows, the Neuron Chip also have 11 programmable I/O pins (IO0-IO10). Table 6-4 is a partial lists of the various I/O objects supported by the Neuron Chip. The frequency and timer I/O are supported by two 16-bit timer/counters.

MODE	I/O Object
Direct Modes	Bit I/O
	Byte I/O
	Leveldetect Input
	Nibble I/O
Parallel Modes	Muxbus I/O
	Master/Slave A
	Slave B
Serial Modes	Bitshift I/O
	Serial I/O
	NeuroWire I/O
Timer/Counter Input Modes	Dualslope Input
	Infrared Input
	Period Input
	Pulsecount Input
	Quadrature Input
	Totalcount Input
Timer/Counter Output Modes	Edgedivide Output
	Frequency Output
	Oneshot Output
	Pulsecount Output
	Pulsewidth Output
	Triac Output

Table 6-4. Neuron Chip I/O Objects

The main programming language used to write applications is a derivative of ANSI C known as Neuron C which supports network variables. Fifteen timers can be declared in software by the user's application program. The programs running on the application processor are totally event driven. Therefore, *when* statements are used to execute the application code running on the Neuron Chip. All of these features make the Neuron Chip an effective microcontroller for the LonWorks architecture.

PROPOSED SYSTEM ARCHITECTURE CHAPTER 7

7.1. Distributed Intelligence:

As discussed in Chapter 3, there currently exists a number of different theories on how a robot's control system should be designed. Brooks [3] theorized that the control system of a mobile robot could be divided into different layers of task-achieving behaviors. With each additional layer, the robot achieves a greater sense of competence. The interaction between each layer is very important to the control of the robot. The control system must be responsive to high priority goals, while still servicing necessary "low-level" goals. These higher levels would become responsible for subsuming the tasks of the lower layers. Brook's control system experienced problems originating from inadequate command fusion and the fact his control layers had internal state variables that the network could not access. These system characteristics made it very difficult to modify the existing system. Payton and Rosenblatt [5] endeavored to improve upon Brook's subsumption architecture. They proposed an architecture possessing modules that were so fine-grained they did not have internal state. Another improvement upon Brook's architecture was that new modules did not completely subsume the function of existing modules, but instead biased the decisions toward different alternatives. Payton [4] also proposed a method to create a reflexive control architecture. The architecture takes advantage of both the immediate and assimilated data provided by the robot's sensors. Payton argues that the data from the sensors is only useful as long as it does not become obsolete before the control system can use it. This requires that the sensors' data is processed quickly. The challenges placed upon a control system by constantly changing terrain may be overwhelming for the processing units unless appropriate tradeoffs between immediacy and assimilation are made.

Though these control methodologies were researched during the development of the MITy and MITe prototypes, their use of a single central processor did not allow for the implementation of the advanced control systems suggested by Brooks and Payton. The use of multiple processors on Companion to divide the tasks of sensor fusion and higher level control and path planning was a step towards realizing the proposed control systems. Unfortunately, the need for sensor fusion still limited the speed at which the sensor data could be processed. It was decided that the only way to achieve the necessary speed in processing was to design a distributed control network. One way to do this was the use of smart sensors coupled with a central processor for higher level control.

As previously discussed, smart sensor technology allows all the processing to occur at the sensor and actuator nodes. By processing the data at the transducers and only passing formatted data across the network, a level of abstraction is added that allows the implementation of command fusion, or data fusion, instead of sensor fusion. With the traditional control architectures, command fusion is only implemented after all the sensor's data has been assimilated and normally the same processor is responsible for both functions. If the data from the sensors is being processed at the sensor and the central processor is only responsible for fusing the commands being transmitted by the sensors, then the control system can conceivably have access to the all the sensors' information before it becomes obsolete.

The attributes of a distributed control system designed using smart sensors coupled with a central processor to implement command fusion are many. As discussed in Chapters 5 and 6, a LonWorks smart sensor architecture decreases the overall complexity of the system. From a hardware perspective, the wiring harness is simplified to a two wire bus. Also, the hardware interface between the transducer nodes and the wiring harness as well as between the transducers and the neuron node has been universally defined and maintained throughout the robot. Another advantage derives from the fact that each transducer module can be made self-sufficient by supplying each with its on power supply. From the software side, the added abstraction provided by the network variables allows for the design of a higher level control system without knowing what code is actually running at the nodes. Accordingly, application software is written at the node level without affecting the overall control system. Consequently, a total parallel design effort can be carried out on the entire system in both hardware and software -- something inconceivable for the traditional robot architectures first implemented by UVL.

7.2. Sensor-Actuator Communication Control:

Up to this point, the overall suggestion has been to utilize smart nodes to increase the central controller's efficiency and reduce many of negative aspects of initial system design and manufacturing. However, the theory of smart nodes can be taken to a higher level. If the smart nodes can communicate to the central processor via network variables, then why can't they be designed to communicate to each other instead? The basic distributed control system discussed in section 7.1 is setup as a master-slave network where the central processor interfaces with each smart node but no peer-to-peer interaction occurs between the nodes. Use of the Echelon LonWorks protocol (see Chapter 6) allows for the implementation of a peer-to-peer control system. In his article 'Autonomous Control with Peer-to-Peer I/O

Networks', Lawrence Gibson stated that "LonWorks is currently the only commercial implementation of peer-to-peer technology."[15]

Peer-to-peer networking was one of the main reasons for using the LonWorks protocol. UVL began researching the area of smart nodes to determine a way to reduce the amount of processing time being spent by the central controller on data manipulation and low level control. In a master-slave arrangement, the use of smart nodes decreases the amount of data manipulation being performed on the central controller. Unfortunately, the central controller is still responsible for carrying out all the low level control that determines what each sensor and actuator will do. However, by using the smart nodes in a peer-to-peer configuration, it is possible to relieve the central controller of the task of performing low level control. The central controller can be delegated to a supervisory role over the transducers. This allows the central controller to spend most of its processing power on high level control.

Consider a very simple example where a robot has been designed to navigate through a given environment without hitting any obstacles. This was the same example used by Brook's in his discussion of subsumption. In a master-slave configuration, the central controller would receive a network variable, *obstacle_detected,* from the hazard avoidance sensor. *Obstacle _detected* contains information about whether an obstacle lays in the path of the robot. The controller would then use this information to determine what the appropriate action should be to avoid the obstacle. Once this decision is made, the controller sends out the network variable *turn_robot*. This prompts the steering node to alter the course of the robot.

In a peer-to-peer configuration, the central controller does not play a role in the above scenario. The controller is responsible for sending out the command for the robot to start roving. When an obstacle is detected, the hazard avoidance node sends out the network variable *obstacle_detected;* however, instead of the controller acting upon this data, the steering actuator node would deal with it directly. This is achieved by increasing the complexity of the application code on the respective nodes. The hazard avoidance algorithm is now programmed either on the hazard avoidance sensor node or the steering actuator node. It is totally up to the system designers to decide which node the algorithm should be programmed onto.

Because every node on the network receives the network variables being transmitted, it is possible to have all the low level navigation and hazard avoidance carried out by sensor-actuator communication. The central controller, or even a separate ground station, can be responsible for setting way-points for the robot. Once the way-point is set, the robot traverses its way to the designated point using only sensor-actuator communication. For example, the higher level control system may decide that the robot should travel to a location exactly 100 feet in front of the robot's current position. As in the previous example, if an obstacle is detected then the hazard avoidance node will inform the steering actuator node to steer around the obstacle. The steering actuator node will output a network variable, *steer_angle,* that tells the system what angle the robot should turn. The compass node on the robot receives *steer_angle* and monitors how far the robot has turned. Once the robot has achieved the heading that it needs to steer around the obstacle, the compass node will output a network variable *stop_turn.* Once around the obstacle, the robot returns to its normal course. All of this is done without any interference by the central controller.

In many ways, this can be compared to the way that a human being interfaces with the environment. When the brain, the human's central controller, makes a decision, the body begins to respond to the higher level commands from the brain. For example, you just printed out a copy of your thesis from the computer on your desk and you must now go and retrieve it from the printer down the hall. Your brain will make the decision to get up and go down the hall to the printer. Once you are in motion, your brain may begin thinking about totally different things. You are still able to navigate through the cluttered lab, the narrow lab door, and down the twisting hallway to the printer. You are able to avoid obstacles and navigate without any conscious thought. It is not an uncommon situation to arrive at that printer and not remember the trip there. This is because your sensory system has taken over the low level control of your body. Your eyes are still taking in information and your actuators, arms and legs, react correctly to what is perceived to be an obstacle. If the obstacle is too large to get around easily, then your brain may "come back on line" and take over with higher level control.

7.3. Competence through Sensor Priority (future work):

Brook's [3] proposed adding different layers of task-achieving behaviors to obtain greater levels of robot competence. A similar idea can be implemented through smart sensor technology. As described in detail in the preceding chapters, each node can be programmed to receive network variables, as well as output network variables. Up to this point, these network variables have not been assigned any priority in regards to each other. However, if these network variables were assigned a specific priority, then a system could be designed that had varying levels of competence dependent upon the sensors and actuators it possessed. Such a system can be seen in Figure 7-1.

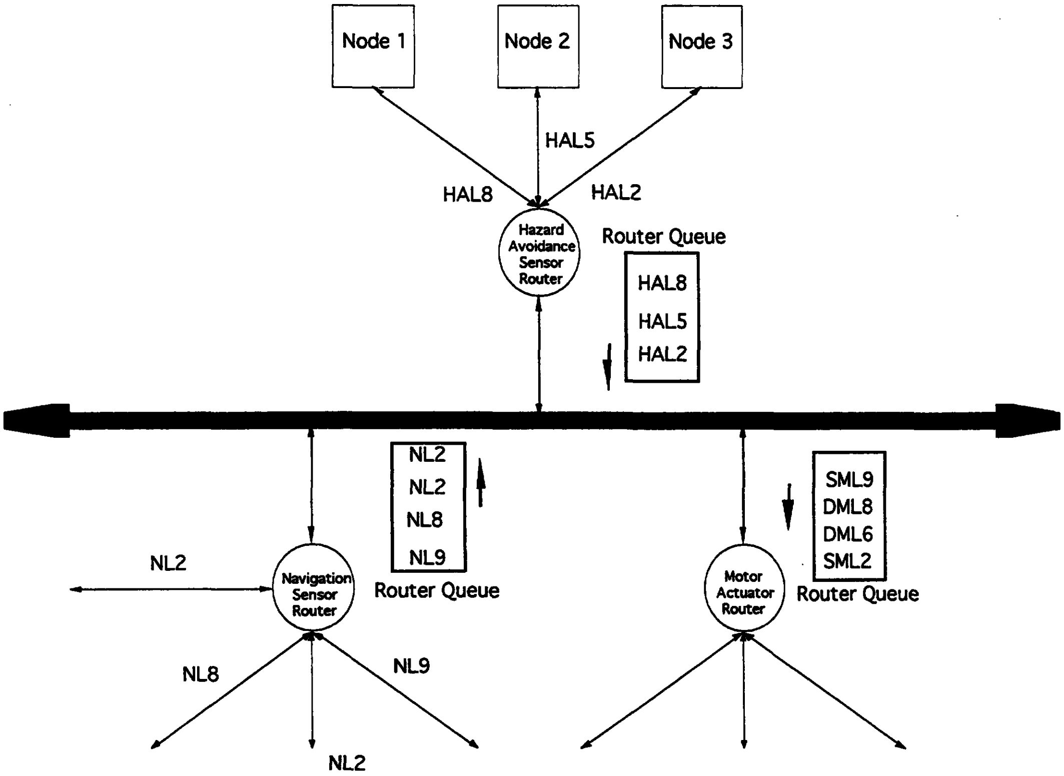


Figure 7-1. Priority-Level based Smart Network

As discussed in Section 6.3.3, the Echelon protocol supports the use of routers and priority level network variables. The network variables can be assigned priorities from 2 to 127 with 2 being the highest priority. When a high priority variable reaches a router, it is immediately moved to the front of the queue. In the system portrayed in Figure 7-1, the various types of sensors and actuators are grouped together under Hazard Avoidance (HA) Sensors, Navigation (N) Sensors, and Motor (DM or SM) Actuators. The network variables passing to and from the nodes have been labeled HAL#, NL#, DML#, and SML# respectively. The numbers attached as a suffix to these variables represent the priority levels. Even if HAL8 and HAL5 arrive before HAL2, HAL2 will move to the top of the queue and will be transmitted first across the network.

By utilizing the priority capability of the Echelon protocol, a varying level of competence can be ascertained. The more important a sensor is to the robot's control system, the higher priority its network variables should be given. Consider a robot that possesses both an acoustic and a laser range finder. The laser range finder will return more accurate data than the acoustic sensor. Consequently, the network variables from the laser node should have a higher priority than the variables from the acoustic sensor node. The robot can work without the laser range finder. However, with the laser node, the robot achieves a higher level of competence. Without the protocols support of priority variables, there would be no guarantee that the laser's data would be transmitted ahead of the sonar's data.

By creating network variables that represent either hazard avoidance, navigation, or motor control, another level of abstraction has been added. The code running on the nodes can be written to take advantage of this abstraction and the various priorities attached to each variable. The nodes that require heading information can be programmed to ignore lower priority heading variables if a high priority heading variable had been acted upon within a finite amount of time. The same idea can be used for any category of network variables. The use of priority level network variables at the protocol level as well as the application code level has not been implemented for this thesis. These characteristics fall outside the scope of this thesis, the programming capability of this author, and the financial resources of this project.

7.4. Multiple Robot-Node Network (future work):

There is an additional feature of smart sensor networks that has not yet been discussed. This feature is currently only supported by the Echelon protocol and hardware. In Section 6.2.3. LonWorks Smart Nodes, it was noted that a number of different transceivers could be used with the Echelon network. What is important for a multiple robot-node network is the ability to make use of RF transceivers. Figure 7-2 represents a block diagram of a two robot network.

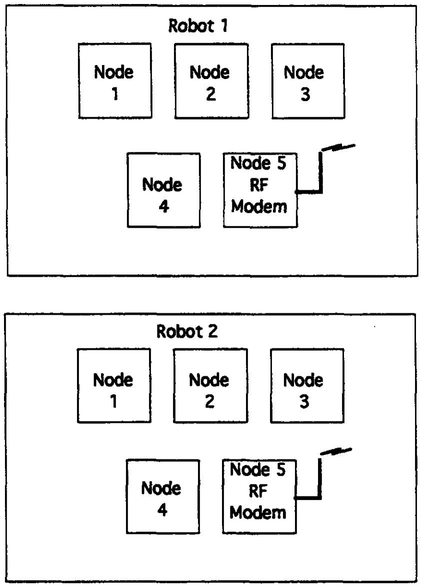


Figure 7-2. Two Robot Network

Each robot is designed around a smart network. Until this point, the discussion has revolved around individual networks where each robot would be acting independently. However, the use of RF modems now allows the possibility of inter-robot communication. By using RF modems, both robots now become sub-networks within a larger overall network. It is totally up to the designer whether any or all the robot's network variables are transmitted to another robot or even a stationary ground station. Consider a situation where multiple robots are being used to search for radioactive materials. A sophisticated ground station is being used to monitor where each of the robots are in relation to some fixed origin. Robot 3 discovers radioactive material in one quadrant of the search area. It transmits to the other robots that it has found radioactive material and where this material is located in reference to itself. The other robots are able to determine where they are in relation to robot 3 via information from the ground station. They can now all converge on the quadrant where the material has been detected and carry out a more thorough search.

RF modems also open up another interesting feature of the smart network systems. System failure is a common concern of all robot designers. With a traditional centralized control architecture, if the main controller suffers a hard-fault then a catastrophic failure occurs. By using RF modems with smart networks, failure of the central controller no longer means failure of the entire robot. Since the RF modem node acts exactly as any other node on the network, it is possible to transmit commands from a ground station to emulate the central controller. This ability can also lead to the removal of the central controller from the robot all together and allowing the ground station to carry out all the higher level control of the system. The RF modem would allow the ground station to monitor the I/O of all the transducers on the robot and consequently knowing the state of the robot at all times. Multiple RF modems could be used for each robot to reduce any bandwidth problems that may be experienced by trying to transmit all the network variables to the ground station. Multiple RF modems would also add a level of fault tolerance in case one of the modems failed. Once again, this issue has not yet been tested; however, it is a major point of interest for UVL and research will be undertaken to study the viability of a multiple robot-node network.

IMPLEMENTATION OF SMART SENSOR ARCHITECTURE
CHAPTER 8

8.1. Smart Module Demonstration:

8.1.1. Explanation of Demonstration Control System:

The intention of this thesis was to research the possibility of implementing a distributed control network on an autonomous robot using smart sensor technology. Accordingly, a working demonstration was designed to simulate an autonomous mobile robot. Early on it was decided that an actual robot would not be designed using smart sensors until this thesis was concluded. For the demonstration to be considered successful certain criteria needed to be met. This criteria called for:

- The design of nodes to support sensors and actuators normally used on UVL prototypes (Includes application code and node-transducer interface)
- The setup of an Echelon LonWorks network. (Includes the system protocol and node-network interface)
- The interconnection of transducer nodes to create a simple control system. (Includes the intercommunication between application code running on each node)

As has been explained in the preceding chapters, smart sensors are a viable option for the creation of a distributed control network and consequently UVL has begun researching into the design of a number of future robots with this technology.

Due to the limited amount of time and resources, only three smart nodes were designed and built by the completion of this thesis. It was decided that for the demonstration to be worthwhile, a hazard avoidance node, a navigational node, and an actuator node should be designed and manufactured. Hazard avoidance is performed using an acoustic sensor range-finder. The on-board navigation sensor is a three axis rate-gyro. The actuator node is designed to control up to four submarine thrusters. These three nodes are connected together into a working distributed control network to simulate a very simple autonomous mobile robot. Figure 8-1 shows a diagram of the demonstration system.

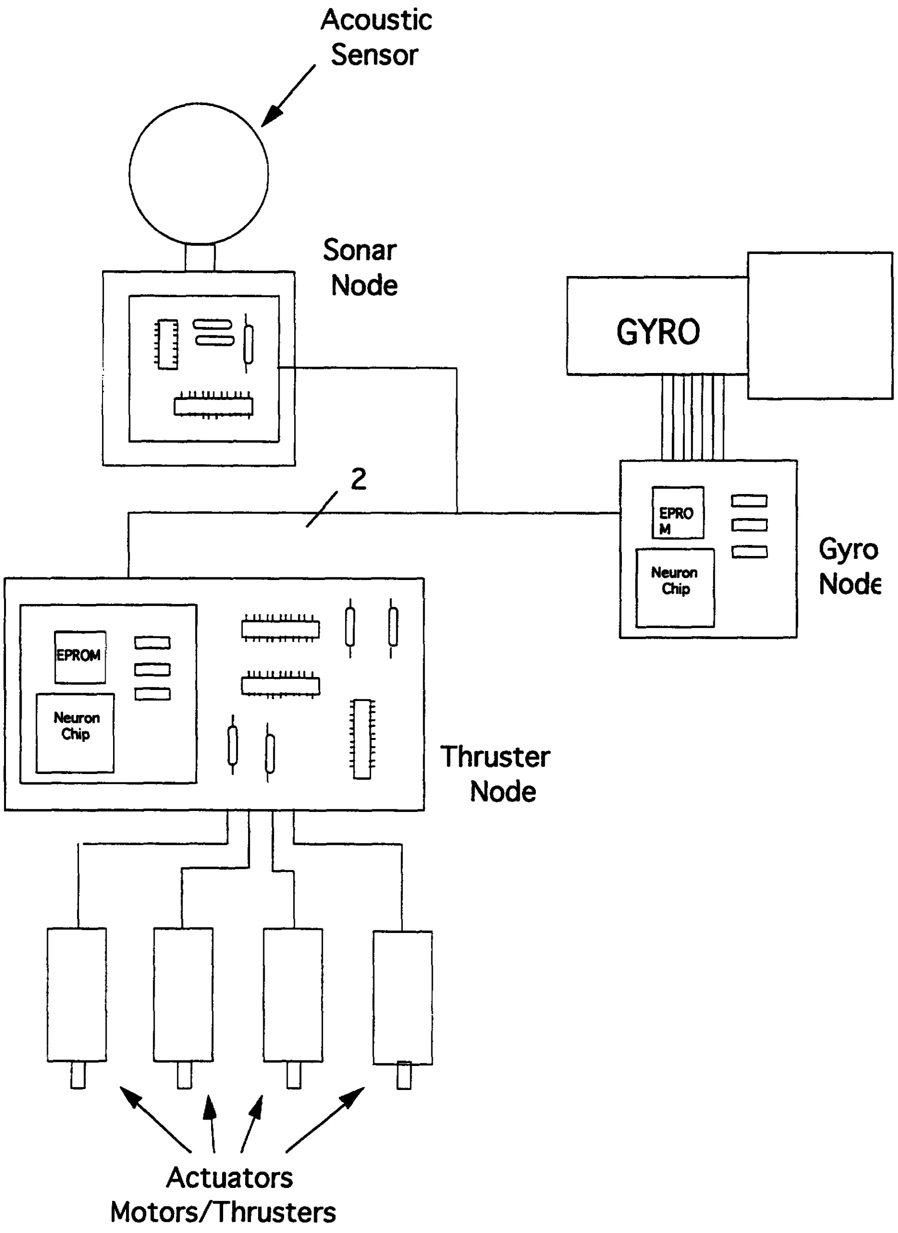


Figure 8-1. Demonstration Network

A central controller acting in a supervisory mode is being emulated by a PC computer running the LonWorks' software. This allows for the manipulation of network variables by the user.

As previously mentioned, the demonstration network emulates a mobile robot with a simple control system. The control code running on the demonstration simulates what Brooks [3] referred to as the lowest level of competence for a robot. The objective of the control system would be to allow a robot to wander through a 2D environment without hitting any obstacles. The acoustic sensor scans the entire region in front of the robot.
The algorithm running on the sonar node divides this front area into the following quadrants.

Quadrant 1: 90° to 45°
Quadrant 2: 45° to 0°
Quadrant 3: 0° to -45°
Quadrant 4: -45° to -90°

It then finds the closest object in each quadrant. Figure 8-2 shows how the two imaginary hemispheres exist in front of the robot. The center of each hemisphere is the sonar.

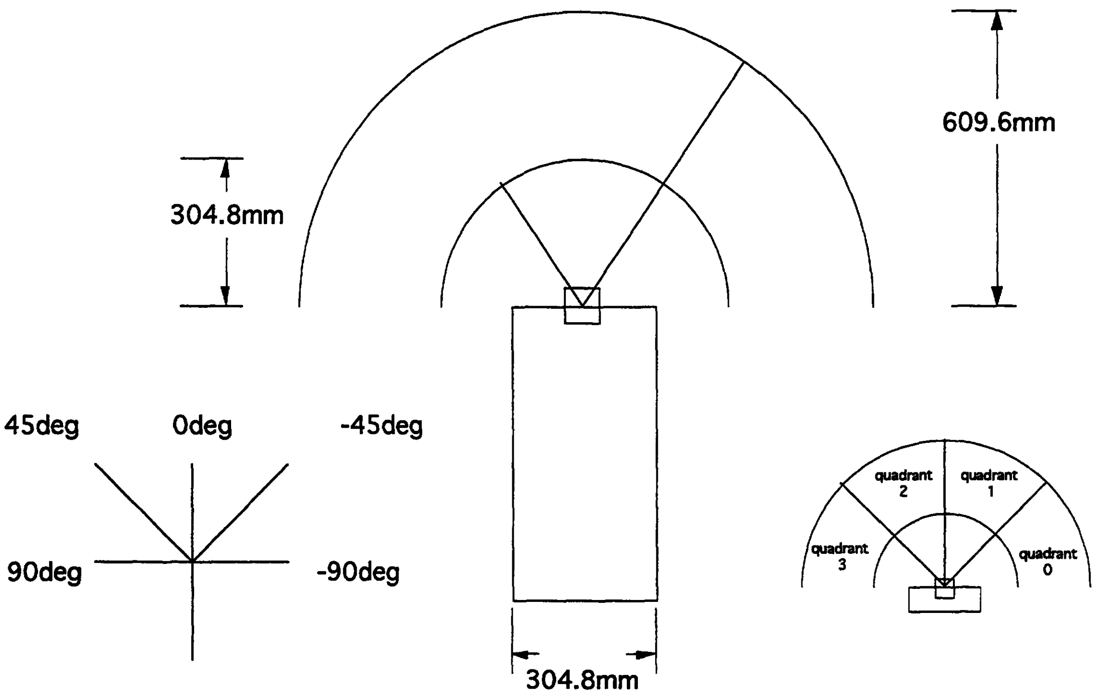


Figure 8-2. Obstacle Avoidance Hemispheres

The inner sphere has a radius of 304.8mm while the outer sphere has a radius of 609.6mm. An object is only considered to be an obstacle if it falls within the outer sphere. Therefore, if no objects are detected within the outer sphere, then the sonar node will not transmit any obstacle data. However, if objects do fall within the outer sphere then the quadrant data will be transmitted across the network. This was done to conserve the bandwidth of the network. The only component of the hazard avoidance algorithm that actually occurs on the sonar node is the safety-stop feature. If an object is detected within the inner sphere (as shown in Figure 8-3), it is considered to be unavoidable.

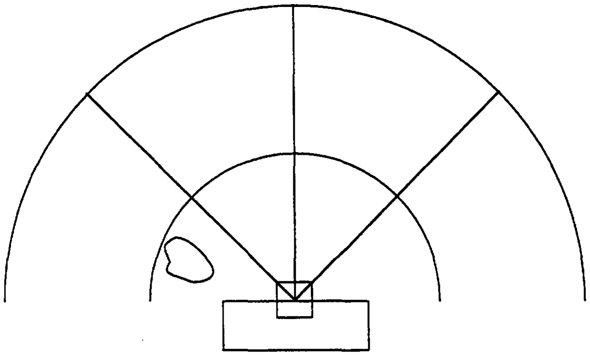
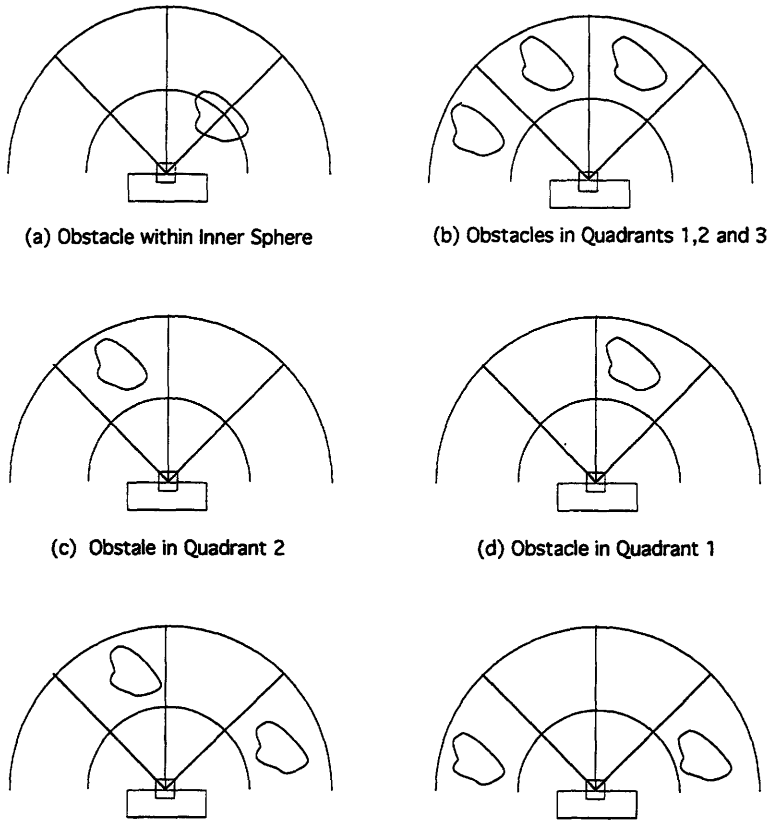


Figure 8-3. Obstacle within Safety-Radius

The safety-stop algorithm sends out a command to stop the robot from preceding forward. This portion of the algorithm is executed as soon as an object is detected. The sonar then continues the scan and outputs the quadrant data.

The main hazard avoidance algorithm resides on the thruster node. The thruster node also allows direct control of each individual thruster by the central controller, external ground station, or other nodes. The thruster node reads the quadrant data off the network whenever they are updated by the sonar node. If an obstacle is determined to be within the inner sphere but the sonar node has not successfully performed the safety-stop algorithm, it will be executed on the thruster node instead (see Figure 8-4(a)). This was added to deal with the possibility of the stop command being lost from the sonar node. A very simple algorithm is used for obstacle avoidance. The quadrant data is first checked to see if more than three obstacles had been detected. As can be seen in Figure 8-4(b), if more than two of the quadrants contain obstacles then there is no way that the robot can proceed safely.



(a) Obstacle within Inner Sphere

(b) Obstacles in Quadrants 1,2 and 3

(c) Obstale in Quadrant 2

(d) Obstacle in Quadrant 1

(e) Obstacles in Quadrants 0 and 2

(f) Obstacles in Quadrants 0 and 3

Figure 8-4. Possible Hazards Encountered by Robot

For the situations when the robot can not avoid the obstacles successfully, a special avoidance routine is called that backs the robot up along a straight path for 1219mm. The distance-traveled information will be input by the user through the PC computer. The routine then commands the robot to precede forward at either an angle of -45° or 45°. The routine alternates between these two angles (i.e. if the last time the routine was called the robot turned to 45° then the robot will turn to -45° the next time the routine is called). There are four situations that can occur to force this special avoidance routine to be called. The first two have already been discussed: an obstacle within the safety radius and three or more obstacles in front of the robot. The next two situations require obstacles to occur in alternating quadrants. An example of this can be seen in Figure 8-4(e). The reason for these last two cases originates from the assumption that the robot does not possess a turning radius that allows it to avoid both obstacles.

Only three other hazard situations exist. If an obstacle is detected in quadrant 1, then the robot will turn counterclockwise 90°. Similarly, if an obstacle is detected in quadrant 2, then the robot will turn clockwise 90°. These are shown in Figure 8-4 (d) and (c) respectively. The final situation is does not require any avoidance routine to be executed. This can be seen in Figure 8-4(f). Since the robot has been modeled as being 304.8mm, any obstacles occurring in quadrants 0 and 3 will not impede the forward motion of the robot.

The gyro node is responsible for making sure that the robot turns to the correct angles when commanded by the hazard avoidance algorithm. The gyro's algorithm is relatively simple. It takes in the commanded heading change and compares the current heading of the robot to the desired heading. When the heading of the robot is within 30° of the desired heading, the gyro will output a slow-turn command that will inform the turning-thrusters to slow to their minimum speed. This allows the robot to initially turn quickly for large angles and to slow the turn as the robot approaches the correct heading. The thruster node also possesses a slow-turn routine that guarantees that if the magnitude of the initial turn angle is less than 30°, then it will turn at its slowest speed.

8.1.2. Smart Nodes and Network Variables:

As mentioned above, the demonstration revolves around three smart nodes: a sonar node; a gyro node; and a thruster node, along with a PC computer running the LonWorks software. The user-code on each node can be divided into two categories. The first is the code necessary for interfacing with the transducer. These device drivers deal with the signals occurring at the neuron chips I/O pins. The second category is the code that makes up the control system. Most of this code depends upon the transfer of network variables between the various nodes. When the smart network is being set up as a distributed control architecture, it becomes important to understand what control code is running on each node. The actual in-and-outs do not need to be known but the information that will be passed throughout the network should be predefined. This allows for the parallel development of individual nodes. It also allows for the modification of code running on each node without affecting the overall control system. Consequently, Pehr Anderson, a MIT CS. student, was able to modify and improve the author's code without changing the demonstration control system. The code running on each node can be seen in Appendix A.

To better understand the demonstration control code, it is important to see what the various network variables are and how they are bounded with each node. Table 8-1 lists each nodes network variables and the network variables to which they are connected.

NODE	NETWORK VARIABLE	BOUND TO
SONAR		
INPUT VARIABLES	nviSonarPower	Central Controller
	nviMotorStop	THRUSTER: nvoThruster_Stop
	nviSTEP_NUM	Central Controller
OUTPUT VARIABLES	nvoInQuadrants[0]	THRUSTER: nviObstacles[0]
	nvoInQuadrants[1]	THRUSTER: nviObstacles[1]
	nvoInQuadrants[2]	THRUSTER: nviObstacles[2]
	nvoInQuadrants[3]	THRUSTER: nviObstacles[3]
	nvoSafety_Stop	THRUSTER: nviSafety_Stop

Table 8-1. Demonstration Network Variables

NODE	NETWORK VARIABLE	BOUND TO
THRUSTER		
INPUT VARIABLES	nvi00Thruster	Central Controller
	nvi01Thruster	Central Controller
	nvi02Thruster	Central Controller
	nvi03Thruster	Central Controller
	nviObstacles[0]	SONAR: nvoInQuadrants[0]
	nviObstacles[1]	SONAR: nvoInQuadrants[1]
	nviObstacles[2]	SONAR: nvoInQuadrants[2]
	nviObstacles[3]	SONAR: nvoInQuadrants[3]
	nviSlow_Turn	GYRO: nvoSlow_Turn
	nviStop_Turn	GYRO: nvoStop_Turn
	nviSteer_Angle	Central Controller
	nviSafety_Stop	SONAR: nvoSafety_Stop
	nviRobot_Speed	Central Controller
	nviStop_BckUp	Central Controller
OUTPUT VARIABLES	nvoTurn_Angle	GYRO: nviHeading_Change
	nvoBack_Up	Central Contoller
	nvoThruster_Stop	SONAR: nviMotor_Stop
GYRO		
INPUT VARIABLES	nviHeading_Change	THRUSTER: nvoTurn_Angle
	nviRecalibrate	Central Controller
	nviClipping	Central Controller
OUTPUT VARIABLES	nvoSlow_Turn	THRUSTER: nviSlow_Turn
	nvoStop_Turn	THRUSTER: nviStop_Turn
	nvoAngle[0]	Central Controller
	nvoAngle[1]	Central Controller
	nvoAngle[2]	Central Controller

Table 8-1. Demonstration Network Variables (Cont.)

As can be seen from Table 8-1, there are a number of network variables that directly interface to the central controller. Some of these are due to the lack of an actual node to give distance data. In a real system, this node would exist and there would be no reason for the central controller to interfere with the control system. Even though in this demonstration each network variable is only bound to one other network variable, a real system would possess network variables that are bound to multiple sources. This is a helpful feature of the LonWorks architecture, since it allows the central controller to have direct access to all the system's network variables.

8.2. Poseidon - A Smart Sensor Surf-Zone Vehicle (Future Work):

The first practical implementation of a smart node architecture will be Poseidon prototype. Poseidon will be an autonomous submarine meant to perform reconnaissance missions in the surf-zone region. The nodes used in the demonstration network will be used on Poseidon. Initially, a traditional hierarchical control system with a central controller carrying out all of the mission and path planning will be implemented. The nodes will only be responsible for conditioning the transducers' I/O into a more useful form. The system will have no reflexive control and will implement sensor fusion. However, once the smart network is fully operational a transition will begin to a semi-distributed control network with the central controller carrying out only higher level control and supervising node interaction. This will allow for reflexive control and the implementation of command fusion instead of sensor fusion.

There are a number of reasons for following this plan. First, the UVL engineers are still far more familiar with the traditional control architectures implemented on the previous prototypes. Second, an evaluation of each system can be made to determine whether the advantages theorized in this paper are actually practical. The Poseidon prototype should be complete by July, 1996. The rapid design and manufacturing stage can be contributed to the ease of laying out the 2-wire bus for the Echelon architecture. The Poseidon will also possess a smart RF modem which will allow a ground station to monitor all of the robot's systems and provide an additional level of fault tolerance in case the central controller fails.

CONCLUSIONS AND RECOMMENDATIONS CHAPTER 9

9.1 Smart Sensor Technology:

The original motivation for this thesis was to determine a more efficient way to design and manufacture a mobile robotics platform. Research on the past UVL prototypes revealed major drawbacks in the traditional system architectures that they were designed around. The thesis' objective evolved to include the finding of an architecture to provide a robust control system and a more practical final product. Practicality was defined as support for future upgradability, modularity, and system adaptability. As this thesis has shown, a system designed using smart nodes possesses the necessary attributes to achieve the system described above.

Three different implementations of smart node networks have been discussed in this paper. The first and simplest system is designed using smart nodes to condition the transducers' data into a more usable format. Sensor fusion is performed by a central controller responsible for path planning, mapping, and system control. The advantages of this system over the traditional architectures are:

1. a 2-wire bus interconnecting each of the transducers on the robot,
2. a predefined hardware interface between the node and the communication medium,
3. a predefined hardware interface between the transducer and the node,
4. and the transmission of pre-formatted data to the central controller from each sensor.

Item 1 allows for easier manufacturing of the robot since the only wiring necessary for the robot would be the running of the 2-wire bus and power lines. Items 2 and 3 allow for the parallel development of individual nodes.

Item 4 decreases the amount of processing carried out by the central controller to assimilate the sensors' data. This is a definite win over the traditional architectures. The next implementation has the smart nodes performing low level control through peer-to-peer interaction. The central controller is now performing command fusion instead of sensor fusion. By relieving the central controller of the responsibility of carrying out data assimilation and low level control, it is now able to dedicate its processing time to higher level control, such as path planning and mapping. This opens up the possibility to implement the control theories of Brooks, Payton, and Rosenblatt which require some form of distributed processing to achieve varying levels on robot intelligence.[3, 4, 5] The final and most advanced configuration does not possess a central controller. Control is performed through peer-to-peer communication between the system nodes.

Research shows that the first two implementations are achievable in the near term. As discussed in Chapter 8, a number of nodes have been built and tested. A simple test system has been designed and studied. Initial tests reveal that smart nodes are a viable solution to the alleviate the drawbacks experienced by traditional control systems. It is recommended that research continue in this area. Further research should be carried out to determine the correct balance between control by the central processing unit and peer-to-peer communication.

9.2 Echelon LonWorks Networking Protocol:

In Chapter 6, a number of different networking tools were discussed to allow for the implementation of a smart node system. After investigating each of these protocols, it was decided to use the Echelon LonWorks Protocol. There were a number of reasons for this choice. From a technical and theoretical standpoint, the most important reason was due to LonWorks support of peer-to-peer interaction between the nodes. From a practical standpoint, the reason was due to Echelon's reputation as being the most user-friendly protocol on the market.

Working with the Echelon networking tools revealed that they are not as user-friendly as was first believed. The protocol possesses a number of kinks that caused UVL to waist a large amount of time and money to work around. Echelon's documentation, as well as the user-interface leaves a lot to be desired. Unfortunately, Echelon's normal response to technical problems required a transfer of funds to obtain a correct answer. It took months for the UVL staff to figure out how to setup a working smart node network and even that process is not totally understood at this moment. If this is the most user-friendly protocol on the market, then this author does not want to see what it would take to get one of the other protocols working.

This author's conclusion about the Echelon system is that you should only use it if you have the money to support it. Once all of the various pieces have been purchased and setup correctly, the system should work without any problems. Basically, it must be looked at as an investment of money that will pay off in the long term. It would certainly have taken UVL far longer to try and develop their own networking protocol and in the end the same amount of money would have been wasted to obtain the same result. It is recommended that further research be carried out in the areas of smart sensor protocols to determine if a better supplier can be found. Otherwise, a closer relationship should be made with the Echelon support engineers so that the LonWorks protocol can evolve into a more user-friendly system.

9.3 Future Research:

Draper has begun researching other areas where smart sensor technology can be used. These areas include both military and industrial applications. The Unmanned Vehicle Laboratory will possess a working smart sensor prototype by July 1996. This will be the Poseidon surf-zone vehicle. Design of a smart sensor based micro-rover will begin by June 1996 and should be completed by January 1997. Upon completion of this vehicle, tests can be carried out to compare the performance of the traditional control architecture with the distributed control architecture theorized in this paper.

REFERENCES

1. Gilbert, John. Design of a Micro-Rover for a Moon/Mars Mission. MS Thesis, MIT December 1992, CSDL-T-1163.
2. Farritor, Shane. Dimensional Kinematic Simulation, Steering System and Scientific Instrument Deployment Mechanism for a Planetary Micro-Rover. MS Thesis, MIT June 1994, CSDL-T-1209.
3. Brooks, R.A. A Robust Layered Control System for a Mobile Robot. IEEE Journal of Robotics and Automation, Vol. RA-2, No. 1, March 1986.
4. Payton, D.W. An Architecture for Reflexive Autonomous Vehicle Control. Proceedings of the IEEE Conference on Robotics and Automation, April 1986.
5. Rosenblatt, K.J., Payton, D.W. A Fine-Grained Alternative to the Subsumption Architecture for Mobile Robot Control. Proceedings of the IEEE International Joint Conference on Neural Networks, Vol II, June 1989
6. Eidson, J.C., Woods, S.P. A Research Prototype of a Networked Smart Sensor System. Hewlett-Packard, 1995.
7. Tapperson, G. Fieldbus: Migrating Control to Field Devices. ISA Paper #94-569, 1994.
8. Ormond, T. Smart Sensors Tackle Tough Environment. EDN, October 1993.
9. Technical discussion with Steven Steiner, MIT CS graduate student.
10. Technical discussion with Dr. David Kang, UVL Supervisor.
11. Smith, S. An Approach to Intelligent Distributed Control for Autonomous Underwater Vehicles. IEEE, 1994.
12. Leonard, M. Creating the Truly Smart Sensor. EDN, October 1994.
13. Gonia, P., Postma, S. Object-Oriented Device Modeling for Flexible Interoperability. Honeywell Microswitch.

14. Dalstra, C.J. Tools Development for Non-hardware Specific Applications. 3rd IEEE/NIST Smart Sensor Workshop, May 1995.

15. Gibson, L. Autonomous Control with Peer-to-Peer I/O Networks. Sensors Magazine, September 1995.

APPENDIX A

A.1 Demonstration Sonar Code:

```
// Sean P. Adam
//3/21/96
// Final Radar Code for Demonstration Network


// This code completely controls the stepper motor and sonar
// system.
//
// Special Features:
//                  1. Only scans hemisphere in front of robot.
//                  2. Has a 25ms timeout timer to handle
//                     missed return from acoustic sensor

#pragma set_node_sd_string "@0,1,3.Sonar transducer and stepper motor."

#include <snvt_rq.h>
#include <snvt_lev.h>

mtimer repeating tmECHOLATE;                        // timeout if no Echo from Sonar
const unsigned long int TIME = 25;
const unsigned long int ONE_FOOT = 3048;            // 1ft -> 304.8mm
const unsigned long int TWO_FOOT = 6096;            // 2ft -> 609.6mm
const unsigned long int SAFETY_FACTOR = 0;          // 0 objects within 609.6mm
const unsigned long int CONV_FACTOR = 7;
const unsigned long int OFFSET_FACTOR = 617;        // corresponds to delay of 40
const unsigned long int SCALE_FACTOR = 10;


int servo_index;                        // index for servo direction
int STEP_DIR = 1;                       // 1-> counterclockwise 0-> clockwise
int temp;
unsigned long int Sum_Quadrants;
unsigned long int SonarMap [40];        // array of sonar points
unsigned long int InQuadrants[4];       // array of closest object in each quadrant
unsigned int Obstacles[4];              // quadrant obstacle marker array

network input  sd_string("@0|1.") SNVT_obj_request nvi00Request;
network output sd_string("@0|2.") SNVT_obj_status nvo00Status;

network input  sd_string("@2|1.") SNVT_lev_disc nviSonarPower;          // On/Off
network output sd_string("@3|1.") SNVT_length_mil nvoInQuadrants[4];    // Obstacle info
network output sd_string("@4|1.") SNVT_lev_disc nvoSafety_Stop;         // Emergency
                                                                        // Stop
network input sd_string("@5|1.") SNVT_lev_disc nviThrusterStop;         // Thrusters are
                                                                        // stopped
network input sd_string("@6|1.") SNVT_count nviSTEP_NUM = 10;           // #servo steps

void send_click(void);                          // fire sonar
void move_stepper(void);                    // move stepper
void find_sonar_vectors(void);              // find closest obstacles

////////////////////////////////////////////////////////////////////////////
// Written by Charles Tung

IO_1 output pulsecount invert clock(6) ioStep_Pulse;   // pulse # of steps
IO_4 output bit ioStep_Dir;                            // direction of travel
IO_5 input ontime mux invert clock(1) ioEcho;          // measure ECHO time
IO_6 output bit ioInit;                                // send pulse
IO_7 output bit ioBinh;                                // start listening


////////////////////////////////////////////////////////////////////////////

when (reset)
{
        io_out (ioStep_Pulse, 0);                      // stop motor
        io_out (ioStep_Dir, STEP_DIR);                 // set direction
        send_click();                                  // clear first measurement
}

when (wink)
{
        move_stepper();
        send_click();
}

////////////////////////////////////////////////////////////////////////////

when (nv_update_occurs(nvi00Request))
{
  if (nvi00Request.object_id > 3)
    nvo00Status.invalid_id = TRUE;
  else {
    nvo00Status.invalid_id = FALSE;
    nvo00Status.invalid_request = FALSE;
    nvo00Status.object_id = nvi00Request.object_id;

  // TODO: Replace the following with your own application code

    switch (nvi00Request.object_request) {
    case RQ_NORMAL:
    case RQ_UPDATE_STATUS:                  // just update status
      break;
    case RQ_REPORT_MASK:                    // report possible error bits
       nvo00Status.invalid_id = TRUE;
       nvo00Status.invalid_request = TRUE;
      break;
    default:                                // reject all other requests
       nvo00Status.invalid_request = TRUE;
      break;
    }
  }
```

```
}

////////////////////////////////////////////////////////////////////////////

// Code assumes that nviStep_Num will be set
// when servo is in starting position
when (nv_update_occurs(nviSTEP_NUM))
{
        servo_index = 0;
}

when (nv_update_occurs(nviSonarPower))
{
        if (nviSonarPower == ST_ON)
        {
                io_out (ioStep_Dir, STEP_DIR);          // set direction
                tmECHOLATE = TIME;                      // set timer
                servo_index = 0;                        // initialize servo position
                nvoSafety_Stop = ST_OFF;                // turn off Safety stop
                move_stepper();                         // step servo
                send_click();                           // fire sonar
        }
        else
        {
                tmECHOLATE = 0;                         // turn off timer
        }
}

// if thrusters are stopped then reset safety_stop
when (nv_update_occurs(nviThrusterStop))
{
        if (nviThrusterStop == ST_ON)
        {
                nvoSafety_Stop = ST_OFF;
        }
}

////////////////////////////////////////////////////////////////////////////

when (io_update_occurs (ioEcho))
{
        // input sonar returns into array
        SonarMap[servo_index] = ((input_value - OFFSET_FACTOR) / SCALE_FACTOR) *
                                        CONV_FACTOR;
        if (nviSonarPower == ST_ON)
        {

                // check safety radius
                if ((SonarMap[servo_index] < ONE_FOOT) && (nviThrusterStop ==
                                                                ST_OFF))
                {
                        nvoSafety_Stop = ST_ON;
                }
                if(STEP_DIR)
                {
                        // increment servo if endpoint not reached
                        if (servo_index < (200/nviSTEP_NUM)) servo_index++;
                        else
                        {
                                find_sonar_vectors();               // call obstacle function
                                servo_index = (200/nviSTEP_NUM);    // reset servo position
                                STEP_DIR = !STEP_DIR;               // change Step_Dir
                                io_out(ioStep_Dir, STEP_DIR);       // set servo direction
                        }
                }
                else
                {
                        // decrement servo if endpoint not reached
                        if (servo_index > 0) servo_index–;
                        else
                        {
                                find_sonar_vectors();        // call function to find closest
                                servo_index = 0;             // call function to find closest
                                STEP_DIR = !STEP_DIR;               // change Step_Dir
                                io_out(ioStep_Dir, STEP_DIR);       // set servo direction
                        }
                }
                move_stepper();                 // move stepper
                send_click();                   // fire sonar
                tmECHOLATE = TIME;              // reset timer
        }
}

/////////////////////////////////////////////////////////////////////////////

when(timer_expires(tmECHOLATE))
{
        send_click();                   // fire sonar
}

/////////////////////////////////////////////////////////////////////////////
// Written by Charles Tung

// Function to fire sonar
void send_click()
{
        io_out (ioInit, 0);                     // reset Init
        io_out (ioBinh, 0);                     // reset Binh
        delay (1000);
        io_out (ioInit, 1);                     // send pulse
        delay (40);
        io_out (ioBinh, 1);                     // start listening
}

// Function to more stepper
void move_stepper()
{
        io_out (ioStep_Pulse, nviSTEP_NUM);
```

```
}

//////////////////////////////////////////////////////////////////////////

void find_sonar_vectors(void)
{
        int step;
        int i;
        Sum_Quadrants = 0;                              // initialize sum of closest obstacles

        // Search each quadrant. 200 steps for servo to go from 0 degrees to 180 degrees
        // Divide hemisphere into quadrants based off of STEP_NUM

        // Quadrant 0
        temp = 0;
        for(step = 1; step < ((200/nviSTEP_NUM)/4)-1; step++)
        {
                if(SonarMap[step] < SonarMap[temp])
                {
                        temp = step;
                }
                InQuadrants[0] = SonarMap[temp];
        }

        // Quadrant 1
        temp = ((200/nviSTEP_NUM)/4)-1;
        for(step=((200/nviSTEP_NUM)/4); step < ((200/nviSTEP_NUM)/2)-1; step++)
        {
                if(SonarMap[step] < SonarMap[temp])
                {
                        temp = step;
                }
                InQuadrants[1] = SonarMap[temp];
        }

        // Quadrant 2
        temp = ((200/nviSTEP_NUM)/2)-1;
        for(step=((200/nviSTEP_NUM)/2); step < (3*(200/nviSTEP_NUM)/4)-1; step++)
        {
                if(SonarMap[step] < SonarMap[temp])
                {
                        temp = step;
                }
                InQuadrants[2] = SonarMap[temp];
        }

        // Quadrant 3
        temp = (3*(200/nviSTEP_NUM)/4)-1;
        for(step=(3*(200/nviSTEP_NUM)/4); step < (200/nviSTEP_NUM)+1; step++)
        {
                if(SonarMap[step] < SonarMap[temp])
                {
                        temp = step;
                }
        InQuadrants[3] = SonarMap[temp];
    }
    for(i=0; i < 4; i++)
    {
        // if obstacle within two feet then set flag
        if(InQuadrants[i] < TWO_FOOT)
        {
            Obstacles[i] = 1;
        }
        else
        {
            Obstacles[i] = 0;
        }
        Sum_Quadrants += Obstacles[i];
    }

    // if there are any obstacles then output data
    if(Sum_Quadrants > SAFETY_FACTOR)
    {
        for(i=0; i < 4; i++)
        {
            nvoInQuadrants[i] = InQuadrants[i];
            Obstacles[i] = 0;
        }
    }
}

// End of file rdr10.nc
```

A.2 Demonstration Thruster Code:

```
// Sean P. Adam
// Modified by Pehr Anderson
//3/21/96
// Thruster Code for Demonstration Network

//////////////////////////////////////////////////////////////

#pragma set_node_sd_string "@0,3,3,3,3.Thruster Node:  -10 to 10 volt output"
#pragma enable_io_pullups

#include <snvt_rq.h>
#include <snvt_lev.h>

//////////////////////////////////////////////////////////////

const long int  HALFPI = 157;
const long int THREEHALFPI = -157;

const unsigned long int ONE_FOOT = 3048;
const unsigned long int TWO_FOOT = 6096;

// Thruster Speed Settings
const int STOP_SPEED = 128;
const int POS_MAX_SPEED = 255;
const int POS_HALF_SPEED = 224;
const int POS_MIN_SPEED = 192;
const int NEG_MIN_SPEED = 64;
const int NEG_HALF_SPEED = 32;
const int NEG_MAX_SPEED = 0;

// Thruster Numbering
const int FOR_THRUST_ONE = 0;
const int FOR_THRUST_TWO = 1;
const int TURN_THRUST_ONE = 2;
const int TURN_THRUST_TWO = 3;

//////////////////////////////////////////////////////////////

network input  sd_string("@0|1.") SNVT_obj_request nvi00Request;
network output sd_string("@0|2.") SNVT_obj_status nvo00Status;

network input  sd_string("@1|1.") int nvi00Thruster;          // for external control
network input  sd_string("@2|1.") int nvi01Thruster;          // for external control
network input  sd_string("@3|1.") int nvi02Thruster;          // for external control
network input  sd_string("@4|1.") int nvi03Thruster;          // for external control

network input  sd_string("@5|1.") SNVT_length_mil nviObstacles[4] = {10000, 10000, 10000,
                                                                     10000};
                                                  // input from InQuadrants on sonar
```

```
network output sd_string("@6|1.") SNVT_angle_vel nvoTurn_Angle;  // output to Gyro
network input  sd_string("@7|1.") SNVT_lev_disc nviSlow_Turn;     // input from Gyro
network input  sd_string("@8|1.") SNVT_lev_disc nviStop_Turn;     // input from Gyro
network input  sd_string("@9|1.") SNVT_angle_vel nviSteer_Angle;  // input from Planner
network output sd_string("@10|1.") SNVT_lev_disc nvoBack_Up;      // output to distance
                                                                  // sensor
network input  sd_string("@11|1.") SNVT_lev_disc nviStop_BckUp;   // input from distance
                                                                  // sensor
network input  sd_string("@12|1.") SNVT_lev_disc nviSafety_Stop;  // input from sonar
network input  sd_string("@13|1.") int nviRobot_Speed;            // set forward speed

network output sd_string("&14|1.") SNVT_lev_disc nvoThruster_Stop = ST_OFF;
                                                    // output thrusters are stopped
//////////////////////////////////////////////////////////////

void stop_thruster(int thruster);
void set_thruster(int thruster, int voltage);     // set thruster speed
void turn_robot(long int turn_angle);             // turn robot
void all_stop(void);                              // stop all thrusters

//////////////////////////////////////////////////////////////

int i;
int num_obstacles = 0;                            // number of obstacles
unsigned long int Obstacles[4];                   // obstacle array
long int Angle_Of_Turn;
long AFTER_BCKUP_ANGLE = 78;                      // Equivalent to PI/4
int safety_breached = 0;                          // Safety flag

//////////////////////////////////////////////////////////////
// Written by Charles Tung and Pehr Anderson

IO_0 output byte ioVoltage = 128;                 // IO0-7 Voltage on thruster
IO_8 output bit ioWR = 1;                         // Write enable
IO_9 output bit ioLo = 0;                         // Lo address bit
IO_10 output bit ioHi = 0;                        // Hi address bit

//////////////////////////////////////////////////////////////

when (reset)
{
        all_stop();
}



when (wink)
{
        set_thruster(FOR_THRUST_ONE,NEG_MAX_SPEED);
        set_thruster(FOR_THRUST_TWO,NEG_MIN_SPEED);
        set_thruster(TURN_THRUST_ONE,POS_MIN_SPEED);
        set_thruster(TURN_THRUST_TWO,POS_MAX_SPEED);
}
```

```
////////////////////////////////////////////////////////////////

when(nv_update_occurs(nvi00Request))
{
   if (nvi00Request.object_id > 5)
      nvo00Status.invalid_id = TRUE;
   else {
      nvo00Status.invalid_id = FALSE;
      nvo00Status.invalid_request = FALSE;
      nvo00Status.object_id = nvi00Request.object_id;

      switch (nvi00Request.object_request) {
      case RQ_NORMAL:
      case RQ_UPDATE_STATUS:                   // just update status
        break;
      case RQ_REPORT_MASK:                     // report possible error bits
         nvo00Status.invalid_id = TRUE;
         nvo00Status.invalid_request = TRUE;
        break;
      default:                                 // reject all other requests
         nvo00Status.invalid_request = TRUE;
        break;
      }
   }
}

////////////////////////////////////////////////////////////////
// Written by Charles Tung

// Set Thruster 0
when(nv_update_occurs(nvi00Thruster))
{
        set_thruster(FOR_THRUST_ONE, nvi00Thruster);
}

// Set Thruster 1
when(nv_update_occurs(nvi01Thruster))
{
        set_thruster(FOR_THRUST_TWO, nvi01Thruster);
}

// Set Thruster 2
when(nv_update_occurs(nvi02Thruster))
{
        set_thruster(TURN_THRUST_ONE, nvi02Thruster);
}

// Set Thruster 3
when(nv_update_occurs(nvi03Thruster))
{
        set_thruster(TURN_THRUST_TWO, nvi03Thruster);
}

////////////////////////////////////////////////////////////////
```

```
// Set robot speed
when (nv_update_occurs(nviRobot_Speed))
{
        set_thruster(FOR_THRUST_ONE, nviRobot_Speed);
        set_thruster(FOR_THRUST_TWO, nviRobot_Speed);
        nvoThruster_Stop = ST_OFF;
}

// if sonar detects obstacle within one foot then stop robot
when (nv_update_occurs(nviSafety_Stop))
{
        if(nviSafety_Stop == ST_ON)
        {
                set_thruster(FOR_THRUST_ONE, STOP_SPEED);
                set_thruster(FOR_THRUST_TWO, STOP_SPEED);
                nvoThruster_Stop = ST_ON;
                safety_breached = 1;
        }
        else
        {
                safety_breached = 0;
        }
}

// When obstacle information is sent
when(nv_update_occurs(nviObstacles))
{
        num_obstacles = 0;
        for(i=0; i<4; i++)
        {
                // input obstacle information
                Obstacles[i] = nviObstacles[i];

                // if obstacle within safety radius but safety flag
                // not set then stop robot
                if((Obstacles[i] < ONE_FOOT) && (safety_breached == 0))
                {
                        set_thruster(FOR_THRUST_ONE, STOP_SPEED);
                        set_thruster(FOR_THRUST_TWO, STOP_SPEED);
                        nvoThruster_Stop = ST_ON;
                        safety_breached = 1;            // set safety flag
                        Obstacles[i] = 3;               // force robot to backup
                }
                else
                {
                        if(Obstacles[i] < TWO_FOOT)
                        {
                                Obstacles[i] = 1;       // mark quadrant as having obstacle
                        }
                        else
                        {
                                Obstacles[i] = 0;       // mark quadrant as empty
                        }
```

```
            }
            num_obstacles += Obstacles[i];          // count number of obstacles
        }

        // if more than two obstacles then stop robot
        // and invoke back up routine
        if(num_obstacles > 2)
        {
            for(i=0; i<4; i++)
            {
                stop_thruster(i);
            }
            nvoBack_Up = ST_ON;                             // initiate back up routine

            set_thruster(FOR_THRUST_ONE, NEG_MIN_SPEED);
            set_thruster(FOR_THRUST_TWO, NEG_MIN_SPEED);
        }
        else
        {
            // if obstacles are unavoidable then stop robot and invoke
            // back up routine
            if((Obstacles[1]&&Obstacles[2]) || (Obstacles[0]&&Obstacles[2]) ||
                        (Obstacles[1]&&Obstacles[3]))
            {
                for(i=0; i<4; i++)
                {
                    stop_thruster(i);
                }
                nvoBack_Up = ST_ON;
                set_thruster(FOR_THRUST_ONE, NEG_MIN_SPEED);
                set_thruster(FOR_THRUST_TWO, NEG_MIN_SPEED);
            }

            // if obstacle in quadrant 1 turn clockwise 90 degrees
            else if(Obstacles[1])
            {
                turn_robot(THREEHALFPI);
            }

            // if obstacle in quadrant 2 turn counterclockwise 90 degrees
            else if(Obstacles[2])
            {
                turn_robot(HALFPI);
            }
        }
    }

    when(nv_update_occurs(nviSlow_Turn))
    {
        if(nviSlow_Turn == ST_ON)
        {
            set_thruster(FOR_THRUST_ONE, POS_MIN_SPEED);
            set_thruster(FOR_THRUST_TWO, POS_MIN_SPEED);
```

```
                // turn slowly counterclockwise
                if(Angle_Of_Turn > 0)
                {
                        set_thruster(TURN_THRUST_ONE, POS_MIN_SPEED);
                        set_thruster(TURN_THRUST_TWO, POS_MIN_SPEED);
                }

                // turn slowly clockwise
                else
                {
                        set_thruster(TURN_THRUST_ONE, NEG_MIN_SPEED);
                        set_thruster(TURN_THRUST_TWO, NEG_MIN_SPEED);
                }
        }
}


when(nv_update_occurs(nviStop_Turn))
{
        if(nviStop_Turn == ST_ON)
        {
                // stop turning
                stop_thruster(TURN_THRUST_ONE);
                stop_thruster(TURN_THRUST_TWO);
                nvoThruster_Stop = ST_OFF;              // output thrusters are stopped
        }
}


when(nv_update_occurs(nviSteer_Angle))
{
        turn_robot(nviSteer_Angle);
}


// stop backing up
when(nv_update_occurs(nviStop_BckUp))
{
        // start forward slowly
        set_thruster(FOR_THRUST_ONE, POS_MIN_SPEED);
        set_thruster(FOR_THRUST_TWO, POS_MIN_SPEED);
        safety_breached = 0;                            // reset safety flay
        nvoBack_Up = ST_OFF;                            // turn off backup command
        turn_robot(AFTER_BCKUP_ANGLE);
        AFTER_BCKUP_ANGLE = -AFTER_BCKUP_ANGLE;
                                                        // alternate between 45deg and -45deg
}


//////////////////////////////////////////////////////////////////
// Written by Charles Tung
// Edited by Sean P. Adam

// stop all thrusters
```

```
void all_stop()
{
        set_thruster(FOR_THRUST_ONE, STOP_SPEED);
        set_thruster(FOR_THRUST_TWO, STOP_SPEED);
        set_thruster(TURN_THRUST_ONE, STOP_SPEED);
        set_thruster(TURN_THRUST_TWO, STOP_SPEED);
        nvoThruster_Stop = ST_ON;                          // Announce the stoppage
}

void set_thruster(int thruster, int voltage)
{
        if (nvoThruster_Stop == ST_ON)                     // Clear the Stopped signal
                nvoThruster_Stop = ST_OFF;

        io_out(ioVoltage, voltage);
        switch(thruster)
        {
                case 0:
                        io_out(ioHi,0);
                        io_out(ioLo,1);
                        break;
                case 1:
                        io_out(ioHi,1);
                        io_out(ioLo,1);
                        break;
                case 2:
                        io_out(ioHi,1);
                        io_out(ioLo,0);
                        break;
                case 3:
                        io_out(ioHi,0);
                        io_out(ioLo,0);
                        break;
        }
        io_out(ioWR,0);                         // pulse WR low
        delay(1000);
        io_out(ioWR,1);
        delay(1000);
}

//////////////////////////////////////////////////////////////////

void stop_thruster(int thruster)
{
        io_out(ioVoltage, 128);

        switch(thruster)
        {
                case 0:
                        io_out(ioHi,0);
                        io_out(ioLo,1);
                        break;
                case 1:
                        io_out(ioHi,1);
                        io_out(ioLo,1);
                        break;
                case 2:
                        io_out(ioHi,1);
                        io_out(ioLo,0);
                        break;
                case 3:
                        io_out(ioHi,0);
                        io_out(ioLo,0);
                        break;
        }
        io_out(ioWR,0);                         // pulse WR low
        delay(1000);
        io_out(ioWR,1);
        delay(1000);
}

void turn_robot(long int turn_angle)
{
        nvoTurn_Angle = turn_angle;             // output turn angle
        Angle_Of_Turn = turn_angle;

        // if new heading == current heading
        if(turn_angle == 0)
        {
                set_thruster(FOR_THRUST_ONE,POS_MIN_SPEED);
                set_thruster(FOR_THRUST_TWO, POS_MIN_SPEED);
                stop_thruster(TURN_THRUST_ONE);
                stop_thruster(TURN_THRUST_TWO);
        }
        else
        {
                // if angle within 30deg then turn slowly and proceed slowly
                if(abs(turn_angle) < 52)
                {
                        set_thruster(FOR_THRUST_ONE, POS_MIN_SPEED);
                        set_thruster(FOR_THRUST_TWO, POS_MIN_SPEED);
                        if(turn_angle > 0)
                        {
                                set_thruster(TURN_THRUST_ONE, POS_MIN_SPEED);
                                set_thruster(TURN_THRUST_TWO, POS_MIN_SPEED);
                        }
                        else
                        {
                                set_thruster(TURN_THRUST_ONE, NEG_MIN_SPEED);
                                set_thruster(TURN_THRUST_TWO, NEG_MIN_SPEED);
                        }
                }

                // else travel at half-speed
                else
                {
                        set_thruster(FOR_THRUST_ONE, POS_HALF_SPEED);
                        set_thruster(FOR_THRUST_TWO, POS_HALF_SPEED);
```

```
                    if(turn_angle > 0)
                    {
                            set_thruster(TURN_THRUST_ONE, POS_HALF_SPEED);
                            set_thruster(TURN_THRUST_TWO, POS_HALF_SPEED);
                    }
                    else
                    {
                            set_thruster(TURN_THRUST_ONE, NEG_HALF_SPEED);
                            set_thruster(TURN_THRUST_TWO, NEG_HALF_SPEED);
                    }
            }
    }
    if(safety_breached == 1)
    {
            stop_thruster(FOR_THRUST_ONE);
            stop_thruster(FOR_THRUST_TWO);
            safety_breached = 0;
    }
}

// End of file thruster.nc
```

A.3 Demonstration Gyro Code:

```
// Pehr Anderson
// Modified by Sean P. Adam
//3/21/96
// Gyro Code for Demonstration Network

////////////////////////////////////////////////////////////////////

// This code polls a LTC1294 (12 bit, 8 channel A/D converter).
// The first three channels are the X,Y, and Z gyro rotation
// channels.

////////////////////////////////////////////////////////////////////

#pragma set_node_sd_string "@0,1,1,1,1,1,1,1,5 Gyro Integration Node"

#pragma enable_io_pullups

#include <snvt_rq.h>
#include <snvt_lev.h>

////////////////////////////////////////////////////////////////////

#define GYRO_POLL  10                // Poll gyro every 10ms
#define INTEGRAL_DIVISOR  10         // dt
#define DRIFT_THRESHHOLD  4          // threshold for maximum drift
#define ANGLE_TOL  9                 // Angle tolerance ~= 5 degrees

// Scale the output to radians (180 degrees ~= 3.14 rad)
#define PI         314
#define XSCALE(x) (x/(14800/PI))              // 1480 average for 180 degrees
#define YSCALE(y) (y/(20384/PI))              // 2038.4 average for 180 degrees
#define ZSCALE(z) (z/(12000/PI))              // 1200 average for 180 degrees

// Define constants to access LTC1294
#define LTC_START             0x80            // Always include this bit
#define LTC_UNIPOLAR          0x04            // Unipolar Samples from 0V to +5V
#define LTC_ACTIVE            0x01            // ~Power Shutdown
#define LTC_MSBF              0x02            // Most Significant Bit First
#define LTC_SGL               0x40            // Single-Ended Channel
                                              // Selection
#define LTC_POWERDOWN   LTC_START             // For Power Shutdown send
                                              // this byte only

// Single ended channels are out of order, this fixes that...
//                    0x01           0x02 & 0x04

// Define macro to access LTC1294
#define LTC_ADDR(x)   (((0x01 & x) << 5) | ((0x06 & x) << 2) | LTC_START | LTC_ACTIVE)

////////////////////////////////////////////////////////////////////
```

```
network input  sd_string("@0|1.") SNVT_obj_request nvi00Request;
network output sd_string("@0|2.") SNVT_obj_status nvo00Status;

network output sd_string("@1|1.Angle") SNVT_angle_vel nvoAngle[3];  // XYZ Angle Data
network input sd_string("@2|1.Recalibrate") SNVT_lev_disc nviRecalibrate;  // Recal Gyro
network input sd_string("@3|1.Heading") SNVT_angle_vel nviHd_Cng_y;    // Turn Angle
network output sd_string("@4|1.Turn") SNVT_lev_disc nvoSlow_Turn = ST_OFF;
network output sd_string("@5|2.Turn") SNVT_lev_disc nvoStop_Turn = ST_OFF;


//////////////////////////////////////////////////////////////////

void update(void);                            // function recalculates drift offset
void  recalibrate(void);                      // function recals Gyro
long int analog_to_digital(short int addr);   // function reads A/D

//////////////////////////////////////////////////////////////////

long int Integral[3];                // variable to hold integral of gyro rate
long int Offset[3];                  // variable to hold offset drift
long int cal_avg;                    // variable to hold avgerage offset
long int new_heading_y;              // variable for new robot heading_y dir
short int slow_turn_flag = 0;        // slow_turn command given
short int turn_occur_flag = 0;       // turn occurring
mtimer repeating GyroPoll;           // period of gyro polling
mtimer repeating GyroUpdate;         // Period of network variable updates
mtimer repeating AnalogPoll;         // period of analog channel polling

//////////////////////////////////////////////////////////////////

IO_0 output bit ADC_cs = 1;                   // Initialize the Chip Select to off
IO_4 input bit ioSwitch;                      // IO4 Switch on LTM-10 Eval Board
IO_8 neurowire master select (IO_0) serial_io;
IO_8 output bitshift numbits (8) clockedge (+) ADC_group_control;

//////////////////////////////////////////////////////////////////

when (reset)
{
        recalibrate  ();             // Reset the Integrator & recalibrate
        GyroPoll = GYRO_POLL;        // Set Gyro Polling Period
        GyroUpdate = 300;            // Set Period of network variable updates
        AnalogPoll = 1000;           // Set Gyro Polling Period (NV not updated)
}


when (wink)
{
        update();
}

//////////////////////////////////////////////////////////////////////////////
```

```
when (nv_update_occurs(nvi00Request))
{
  if (nvi00Request.object_id > 10)
    nvo00Status.invalid_id = TRUE;
  else {
    nvo00Status.invalid_id = FALSE;
    nvo00Status.invalid_request = FALSE;
    nvo00Status.object_id = nvi00Request.object_id;

  // TODO: Replace the following with your own application code

    switch (nvi00Request.object_request) {
    case RQ_NORMAL:
    case RQ_UPDATE_STATUS:          // just update status
      break;
    case RQ_REPORT_MASK:          // report possible error bits
      nvo00Status.invalid_id = TRUE;
      nvo00Status.invalid_request = TRUE;
      break;
    default:                              // reject all other requests
      nvo00Status.invalid_request = TRUE;
      break;
    }
  }
}

////////////////////////////////////////////////////////////////////////////////

// This task is called when the IO_4 switch state is changed
// It causes the integrating gyro to reset and re-calibrate
when  (io_changes(ioSwitch))
{
        recalibrate  ();
}

// Recalibrate gyro
when  (nv_update_occurs(nviRecalibrate))
{
        switch  (nviRecalibrate)
        {
                case ST_ON:
                        recalibrate();
                        break;
                case ST_HIGH:
                        // TODO
                        // Calibrate Divisors for each channel

                        // Prompt the user to rotate the X, Y, & Z channels
                        // by flashing the LED once, twice, and three times
                        // in rapid succession
                        nviRecalibrate  =  ST_OFF;
                        break;
                case ST_LOW:
                case ST_OFF:
                        break;
        }
}


// When heading data is updated
when (nv_update_occurs(nviHd_Cng_y))
{
        new_heading_y = nviHd_Cng_y;          // set new heading
        slow_turn_flag = 0;                   // reset slow turn flag
        turn_occur_flag = 1;                  // set turn occurring flag
        nvoSlow_Turn = ST_OFF;                // set Slow_Turn off
        nvoStop_Turn = ST_OFF;                // set Stop_Turn off

        // if angle between current heading and new heading
        // is less than 30 degrees set slow_turn_flag
        if(abs(new_heading_y - YSCALE(Integral[1])) < 52)
        {
                slow_turn_flag = 1;
        }

}

// Perform the integration
when (timer_expires(GyroPoll))
{
        static short i;
        static long instant;
        for  (i=0;i<3;++i)
        {
                instant = (analog_to_digital(i) - Offset[i]) / INTEGRAL_DIVISOR;
                if (abs (instant) > DRIFT_THRESHHOLD)
                        Integral[i] += instant;
        }
}

// Update the network varibles (this is slower than sampling)
when (timer_expires(GyroUpdate))
{
                nvoAngle[0] = XSCALE(Integral[0]);          // output x displacement

                nvoAngle[1] = YSCALE(Integral[1]);          // output y displacement
                nvoAngle[2] = ZSCALE(Integral[2]);          // output z displacement

                // if robot is turning
                if(turn_occur_flag)
                {
                        // if slow turn flag not set and difference angle
                        // less than 30 degrees then output Slow_Turn
                        if(!slow_turn_flag && abs(new_heading_y - YSCALE(Integral[1])) <
                                                                52)
                        {
                                nvoSlow_Turn = ST_ON;
                                slow_turn_flag = 1;
                        }

                        // if difference angle within tolerance stop turn
                        if(abs(new_heading_y - YSCALE(Integral[1])) < ANGLE_TOL)
                        {
                                nvoStop_Turn = ST_ON;
                                turn_occur_flag = 0;
                        }
                }
}

/////////////////////////////////////////////////////////////////////

long int analog_to_digital(short int addr)
{
        static long adc_data;
        io_out(ADC_cs, 0);                          // Activate ADC
        addr = LTC_SGL | LTC_ADDR(addr);
        io_out(ADC_group_control, addr);            // send addr to ADC
        io_in(serial_io, &adc_data, 16);            // get converted data

        // If the LTC1294 chip select is not deactivated,
        // further inputs on the serial channel will be ignored

        io_out(ADC_cs, 1);                          // De-activate ADC

        // right justify and account for negative inputs
        adc_data = adc_data >> 3;
        if (adc_data & 0x0800)                      // Handle negative data
                return ((0x07ff & adc_data) - 0x0800);
        return (adc_data & 0x07ff);
}


void recalibrate(void)
{
        short i, j;
        for (i=0;i<3;++i)
        {
                // Reset
                cal_avg = 0;
                Integral[i] = 0;

                // Re-Calibrate (this may need to be more robust)
                // todo: fix to wait for signal to settle out
                // read drift and average
                for(j=0; j<100;j++)
                {
                        cal_avg += analog_to_digital(i);
                }
                Offset[i] = cal_avg/100;
        }
}
```

```
// Function just reads drift, averages, and resets offset
void update(void)
{
        short i;
        cal_avg = 0;

        for(i=0; i<100; i++)
                cal_avg += analog_to_digital(i);
        Offset[i] = cal_avg/100;
}

// End of file gyro2.nc
```