UNIVERSITY of LIMERICK

"The Design of An Autonomous Mobile Robot Built to Investigate Behaviour Based Control"

by

Mark Christopher Leyden

A Thesis Submitted For The Degree Of
Master of Engineering

Based On Work Done At
The Department of Electronic and Computer Engineering
University of Limerick, Ireland

Supervisors

Dan Toal and Colin Flanagan

Submitted to the University of Limerick, October, 2000

DECLARATION

I hereby declare that this thesis is entirely my own work and has not been submitted as an exercise to any other university.

_____

Mark Leyden

ABSTRACT

Within this thesis, the design of an autonomous mobile robot built as a testbed to investigate behaviour-based control in a real world environment is described. By adopting a behaviour-based control architecture the robot can respond very rapidly to environmental changes by reacting directly to sensor stimuli. A modified form of a behaviour-based control architecture has been developed for this robot. This is based on the standard subsumption architecture with the addition of a concept known as a blackboard. The blackboard allows the sharing of system state and knowledge between behaviours to help them perform their tasks. In addition, the design of a fuzzy logic navigation system is also described. This was designed to overcome one of the limitations of pure behaviour-based systems - that is, the exclusion of interaction between behaviours. Experimentation with the robot has shown that by interacting with each other through the robot's environment, the behaviours can contribute to seemingly intelligent tasks being performed.

ACKNOWLEDGEMENTS

I would like to sincerely thank my supervisors, Dan Toal and Colin Flanagan, for their support and advice in this work. I would also like to thank Danny O'Brien who did a wonderful job in building the chassis for the robot.

Table of Contents

LIST OF FIGURES

1. INTRODUCTION

The field of robotics can be classified into two distinct areas - industrial robotics and mobile robotics. Industrial robots are generally operated in very well structured and controlled environments such as a car assembly plant. In these environments, everything the robot does is preplanned. There is very little variation in the tasks the robot has to perform and how it performs them (Nehmzow, 2000). In contrast, mobile robots must operate in unstructured and dynamic environments. The typical types of application that these robots perform are many. For example, typical duties may include delivering mail around an office building, cleaning floors and performing security duties. Although these environments may be considered structured for the people who live and work in them, this is generally not the case for the mobile robot. To operate successfully in our everyday environments, mobile robots must be capable of dealing with all the uncertainty and variation that exists (Saffiotti, 1997). To build a robot to deal with all this uncertainty can be a complex and challenging task. Having a model of the environment is not sufficient on its own for the robot to operate robustly. Firstly, the model may be inaccurate or incomplete and secondly, the introduction of new features into the environment may render the model invalid. To operate autonomously and robustly, the robot must be capable of responding directly to the environment itself. It must have certain basic capabilities or reflexes built into it. It must be able to navigate, avoid obstacles, build and update maps and be robust towards any environmental changes.

This thesis describes the design of an autonomous mobile robot built as a testbed for behaviour based control and experimentation (Leyden, 2000b). The design of the robot has been heavily influenced by the need for real-time sensing and decision making in order to operate in dynamic and unstructured environments. This has been achieved by adopting a behaviour based control architecture and tightly coupling behaviours with sensor inputs and actuator outputs in a reactive way. A modular hardware control architecture has been implemented which distributes the workload and offloads sensor data stream processing to a dedicated processor.

1.1 Research Contribution

Building a mobile robot can be a complex, expensive and time-consuming task. For these reasons, many researchers working on mobile robots decide to implement their designs in a simulated environment. In such an environment, it is easy to see the results of a newly applied algorithm. It is also easy to modify and fine-tune various parameters. Testing new ideas and methods can be achieved in a fraction of the time it would take to implement on a real robot. For a simulation to be of any practical use, however, it has to be able to model the real world in a very precise manner. This is an extremely complicated thing to do. Also, unless the results of a simulation are validated on a real robot, there is the possibility that a lot of effort will be put into solving problems which do not crop up in the real world. These problems may be associated with the simulation itself and ones that do not appear in reality (Leyden, 2000a).

Due to the limited usefulness of simulations, it was decided to build a real working robot for this thesis. In doing so, experiments can be carried out in the real world giving a true insight into how the robot's control architecture behaves. To operate reliably in unstructured and dynamic environments, a behaviour-based architecture has been adopted. With this type of architecture, the robot's control system consists of a number of behaviours. Each behaviour is designed to perform a particular task such as avoiding obstacles, following a light etc. The behaviours respond directly to environmental cues through the use of the robot's sensors and actuators (Arkin, 1998). With behaviour-based architectures, the system tends to be very robust. A certain amount of redundancy is built into the system. If one behaviour should fail, then the others should still be capable of controlling the robot, although at a lower level. One of the problems with behaviour-based systems, however, is that due to their highly distributed nature, representation and sharing of system states and knowledge between the behaviours is inconvenient. To overcome this shortcoming, a new type of architecture has been developed here. This is known as a behaviour-based blackboard architecture. This introduces the concept of a blackboard, which acts as a central data repository where behaviours can deposit and extract information to help them go about their tasks. A set of behaviours which allows the robot to successfully explore, map, and navigate around the environment have been implemented. In situations when the

robot becomes lost or is unaware of its position, it can perform localisation to re-establish its correct position in a relative fashion.

With behaviour based architectures, each behaviour is designed to perform a specific task such as obstacle avoidance or goal seeking. Each of these behaviours work independently of each other and in parallel. One of the shortcomings of this design, however, is that it excludes interaction between the behaviours. For example, if the obstacle avoidance behaviour encounters an obstacle, it will attempt to manoeuvre the robot around it. How it goes about this is entirely up to itself. It could go either left or right to avoid the obstacle. If at the same time the robot is attempting to travel to some target location, the decision about which way to turn could be important. Turning in one particular direction might move it closer along the desired path towards the target. Turning in the other direction might move it further away. Since the behaviours work independently of each other, it is not possible for one behaviour to know the goals of the other. To help solve this problem, a system has been developed in fuzzy logic that combines the outputs of the obstacle avoidance behaviour and the goal seeking behaviour into one single value (Leyden, 1999). So far, this system has only been tested in simulation. A detailed discussion of this system is given in chapter 5 along with experimental results from tests carried out.

1.2 Structure of the Thesis

Chapter 2 provides a literature review which is relevant to the work discussed in this thesis. A review of mobile robot control architectures is carried out with examples of various types been shown. An examination of mapping techniques for mobile robots is also presented.

Chapter 3 describes the design and construction of the robot. The robot's locomotion system is explained along with the physical design of the robot. Following this, the distributed control architecture which has been implemented is described in detail.

Chapter 4 discusses the control architecture used on the robot and the behaviours which have been implemented.

Chapter 5 describes the design of a fuzzy logic navigation system which has been developed to overcome one of the limitations of pure behaviour based architectures.

Chapter 6 presents experimental results from tests carried out and comments on how successful the robot's control architecture is.

Chapter 7 gives conclusions from the work carried out and presents ideas for future work.

2. BACKGROUND

This chapter provides a literature review which is relevant to the work discussed in this thesis. It provides a review of the theory of robot architectures. It also discusses the different types of architectures available and emphasises the advantages and limitations of each. A review of robot mapping techniques is also discussed.

2.1 Mobile Robot Control Architectures

In order for a mobile robot to function usefully and reliably in unstructured and dynamic environments, it must be able to perceive its surroundings and generate a set of appropriate actions. This requires the use of an underlying architectural framework which is responsible for the sensing and reasoning processes of the robot. The architecture must be able to gather perceptual and state information and generate a set of actions for the robot to follow (see figure 1). An architecture is basically a collection of software building blocks used to construct the robot's control system (Arkin, 1998). When designing the control architecture, a number of key issues must be taken into account. Decisions have to be made on whether the architecture should be centralised or distributed, whether the reasoning should be reactive or deliberative and whether input combination should occur via sensor fusion or arbitration.

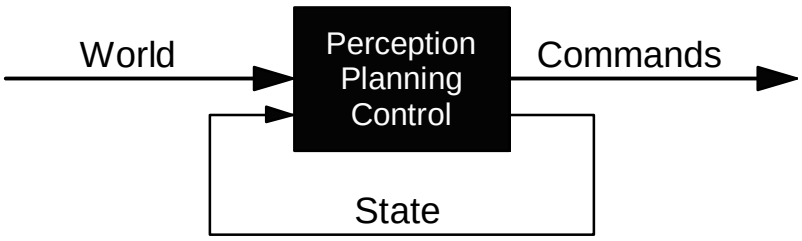


Figure 1 Robot Control Architecture

During the past two decades, two different types of control architectures have dominated the robotics scene. The two architectures differ in the type of reasoning they use. One uses deliberative reasoning to determine what actions should be taken. The other uses reactive

reasoning to tightly couple sensor input with actuator output in a reactive way. Figure 2 shows a comparison of the two methods (Arkin, 1998).

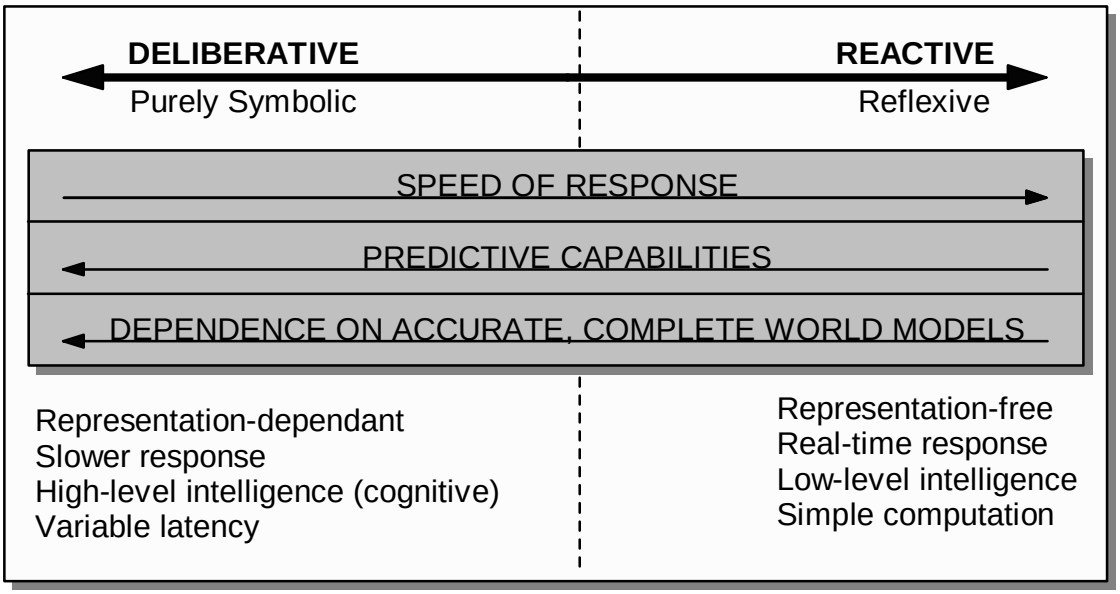


Figure 2 Deliberative versus Reactive Reasoning

2.2 Classical Robot Architectures

The traditional artificial intelligence approach to building a control system for a mobile robot is to break the task into a number of subsystems. These subsystems typically include perception, world modelling, planning, task execution and motor control. The subsystems can be thought of as a series of vertical slices with sensor inputs on the left and actuator outputs on the right. These subsystems form a chain through which information flows from the robot's environment, via the sensors, through the robot and back to the environment through actuators (see figure 3). This type of architecture decomposes a large, complex system into a number of small modules for relative ease of implementation (Albus et al., 1981). Probably the most famous robot built using this type of architecture is Shakey, developed at the Stanford Research Institute in the late 1960s (Nilsson, 1969).

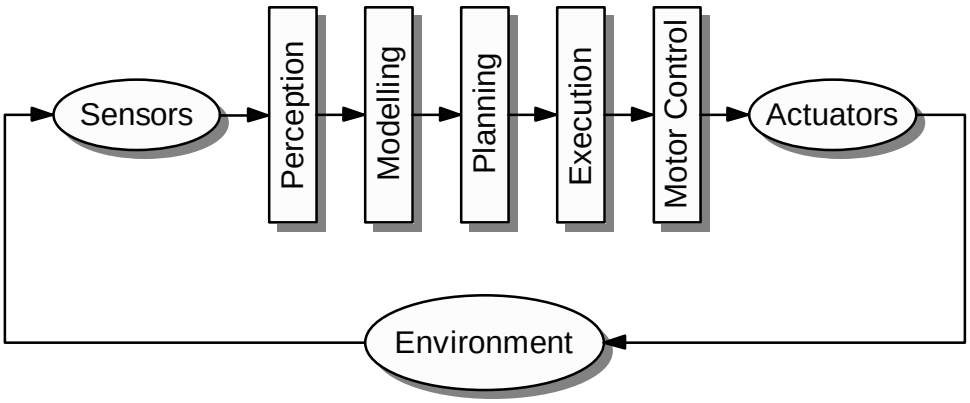


Figure 3 Horizontal Decomposition

The following gives a description of each of the subsystems (Toal et al., 1996).

Perception: This subsystem handles the sensing devices connected to the robot.

World Modelling: This subsystem uses sensor input to determine where the robot is in relation to an internal model of the environment and to update the internal model with new information.

Planning: This subsystem attempts to work out how it will achieve its goals given the current world state and the state of the robot.

Task Execution: This subsystem decomposes the plan into a set of detailed motion commands.

Motor Control: This subsystem interfaces with the robot's actuators to execute motion commands generated by the task execution subsystem.

One of the problems with this type of architecture is that it precludes interaction with low-level sensing and action processes at higer-levels and tends to result in very structured decomposition of problems, leading overall to longer processing times and poor use of sensory information (Orlando, 1984). Its strictly structured decomposition requires an instance of each module to be built in order for the robot to function at all.

The following example highlights the problems with this type of architecture. A robot is instructed to move along a certain path in search of a target. During this time, one of the robot's cameras may detect an obstacle straight ahead along the path, which is not contained

in the world model held by the robot. The information provided by the camera is then processed and the world model is updated. All of this takes time and before the robot is able to respond to the new information contained in the world model, it may already have collided with the obstacle. The problem with this architecture is that planning is a time-consuming task and the world may change during the planning process in a way that invalidates the resulting plan (Kortenkamp et al., 1998). As a result, this type of architecture may only be of use in very structured and highly predictable environments.

2.3 The Symbol System and Physical Grounding Hypotheses

Before discussing reactive and behaviour based architectures, two important hypotheses which are central to the understanding of these architectures will be discussed. These two hypotheses are generally referred to as the *Symbol System Hypothesis* and the *Physical Grounding Hypothesis*.

The symbol system hypothesis (Brooks, 1990) states that intelligence operates on a system of symbols. The implicit idea is that perception and motor interfaces are sets of symbols on which the central intelligence system operates. The robot's sensors deliver a description of entities in the world in terms of symbols to the a central intelligence system which develops an internal model of the world and then drives the actuators based on some belief. The symbols with which these systems reason often have no physical correlation with reality. In other words, they are not grounded by perceptual or actuator processes. The hierarchical architecture described above is based on this type of hypothesis. The Physical grounding hypothesis (Brooks, 1990) is a hypothesis based on the belief that to build an intelligent system, it is necessary to have its representations grounded in the real physical world. This means that a robot must acquire its knowledge from external sensors and not from a set of symbols. Also, all of the robot's goals and desires must be expressed as physical action. Robots based on this hypothesis are generally more robust and can operate in more complex and unstructured environments. Reactive architectures are based on this type of hypothesis.

2.4 Reactive Architectures

To try and overcome the problems associated with the classical type of architecture, a new breed or architecture was developed. This is known as a reactive architecture. Instead of constructing a world model, planning a course of action within that model, and mapping that plan into specific actions, reactive architectures are designed to respond directly to sensor stimuli from environmental cues. This is done by tightly coupling sensor input with actuator output in a reactive way. They are particularly suited to complex and unstructured environments. By sensing the environment at a rapid rate, uncertainty in perception is avoided. Any false readings which are obtained will only have a very limited impact. By continuously acting on the perceived world, any uncertainty on what actions are to be carried out is also avoided. World models are avoided under the belief that "the world is its own best model" (Brooks, 1990). The advantage of reactive architectures is that they have a very fast response time from perceiving an environmental cue to acting on it. This is important in dynamic environments where the robot may come across unforeseen objects.

2.5 Behaviour Based Architectures

A behaviour based architecture is a radically different type of robot control system. Instead of decomposing the architecture into functional modules, the behaviour based architecture decomposes it into task-achieving modules or behaviours. A behaviour in this sense is a routine which performs a certain set of actions in response to a given stimulus from sensors (Arkin, 1998). Each behaviour is designed to perform a particular task such as avoiding obstacles, following a light etc. It does this in response to sensor inputs acting in a similar manner to the way an insect behaves. A behaviour encapsulates the perception, exploration, avoidance, planning and task execution capabilities necessary to achieve one specific aspect of robot control. It is capable of producing meaningful action which in turn can be composed to form levels of competence. Each behaviour realises an individual connection between some kind of sensor data and actuation. The whole system is built step by step from a very low level. Successive levels can be added incrementally to enhance the functionality of the robot.

Control of the robot is distributed across a number of independent behaviours. The interaction between them is what defines the overall behaviour of the robot. Since each behaviour is able to operate independently of each other, if one behaviour fails, the entire control system can still function, albeit at a lower level. This makes the system very robust. Behaviour based architectures have a very fast response time to environmental changes.

Behaviour based architectures have the following general characteristics (Arkin, 1998):

- *Behaviours serve as the basic building blocks for robotic actions.* Each behaviour consists of a senserimotor pair. Information from the sensor determines the motor reflex response.
- *Use of explicit abstract representational knowledge is avoided in the generation of a response.* Purely reactive systems react directly to the world as it is sensed. This avoids the need for intervening abstract representation knowledge.
- *These architectures are inherently modular from a software design perspective.* This allows the competency of the robot to be increased by adding new behaviours, without redesigning or discarding the old ones.
- *Interaction of the behaviours is through the robot's environment.*
- *Behaviours are relatively simple and tend to be more reactive than deliberative.*

Two common architectures which employ this design method are Brooks' subsumption architecture (Brooks, 1986) and Arkin's schema-based architecture (Arkin, 1989). The robot discussed in this thesis uses the subsumption architecture, so a detailed examination of this will be given in the following sections.

2.6 Subsumption

The subsumption architecture was originally developed by Rodney Brooks back in the mid 1980s at MIT (Brooks, 1986). This type or architecture is a purely reactive behaviour based architecture. Unlike classical control which decomposes the control system into a number

of vertical tasks, subsumption decomposes the task into a number of horizontally arranged layers (see figure 4). Each of these layers is capable of implementing a "competence", that is the ability to display a particular behaviour. In addition, each layer encapsulates elements of all the vertical tasks found in the classical system. For example, a layer may have perception, planning, task execution and motor control elements. This allows it to act on information perceived by the robot's sensors and execute a certain set of actions.

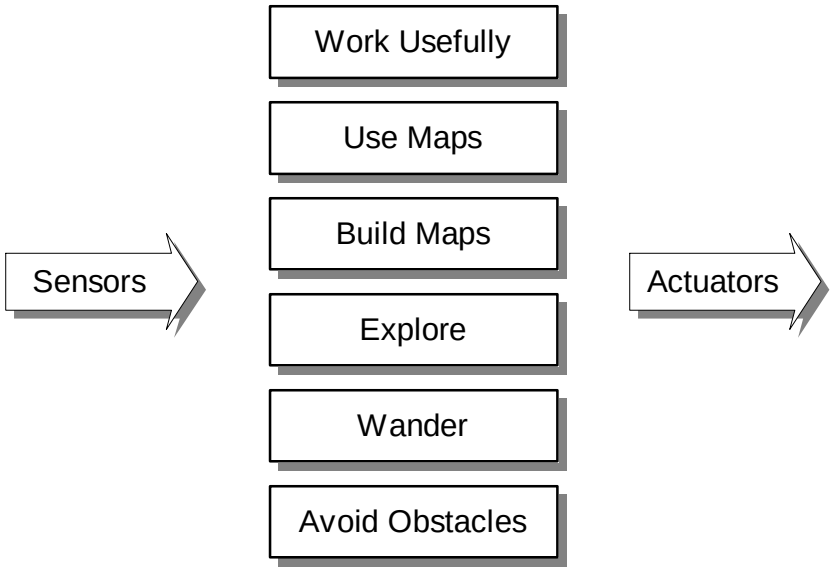


Figure 4 Subsumption Architecture

Each layer in the subsumption architecture implements one behaviour, such as the ability to move away from an obstacle, to follow a light source or to explore the robot's environment. Since the task of controlling the robot is broken down into a number of different behaviours, each behaviour does not have to tackle the whole problem of achieving the overall goal. This greatly simplifies the design of the architecture, since behaviours can be built up incrementally.

2.6.1 Levels of Competence

In subsumption, the control architecture is built up using a number of levels of competence. Each level of competence is an informal specification of how the robot should behave in

any environment it encounters. Higher levels of competence encapsulate all the control layers of lower level competences. The following eight competences are used by Brooks:

Level	Robot's Behaviour
0	Avoid contact with objects (whether the objects are moving or are stationary).
1	Wander aimlessly around without hitting things.
2	Explore the world by seeing places in the distance which look reachable and head for them.
3	Build a map of the environment and plan routes from one place to another.
4	Notice changes in the static environment.
5	Reason about the world in terms of identifiable objects and perform tasks related to certain objects.
6	Formulate and execute plans which involve changing the state of the world in some desirable way.
7	Reason about the behaviour of objects in the world and modify plans accordingly.

The robustness of the robot's control system becomes more acute as the level of competence increases. Take the Explore competence as an example. Since this includes as a subset the Wander and Avoid competencies, it can operate secure in the knowledge that it will not collide with an obstacle. Higher level competencies do not have to be concerned about avoiding obstacles since this is taken care of by a lower level competence.

2.6.2 Layers of Control

The key idea behind levels of competence is that the control system can be built using layers that correspond to each level of competence. Adding a new layer to the control system moves it up to the next level of overall competence.

When designing the control system in Subsumption, the level 0 competence is added first. This endows the robot with an obstacle avoidance ability. This is generally the most

important ability a robot must have and so is added first. A control layer is added which achieves this. Once a reliable and functional level 0 competence has been tested and debugged, the next level of competence is added. This equips the robot with the ability to wonder around without hitting things. A control layer is implemented that lets the robot wander freely. This layer, however, does not have the ability to avoid obstacles. Instead, this is taken care of by layer 0. Together, layer 0 and layer 1 constitute the level 1 competence.

Higher level layers are able to examine data from lower level layers and can also inject data into the internal interfaces of a lower level layer to suppress the normal data flow. Lower level layers always run, however, without being aware of the layers above them. This is shown in figure 5.

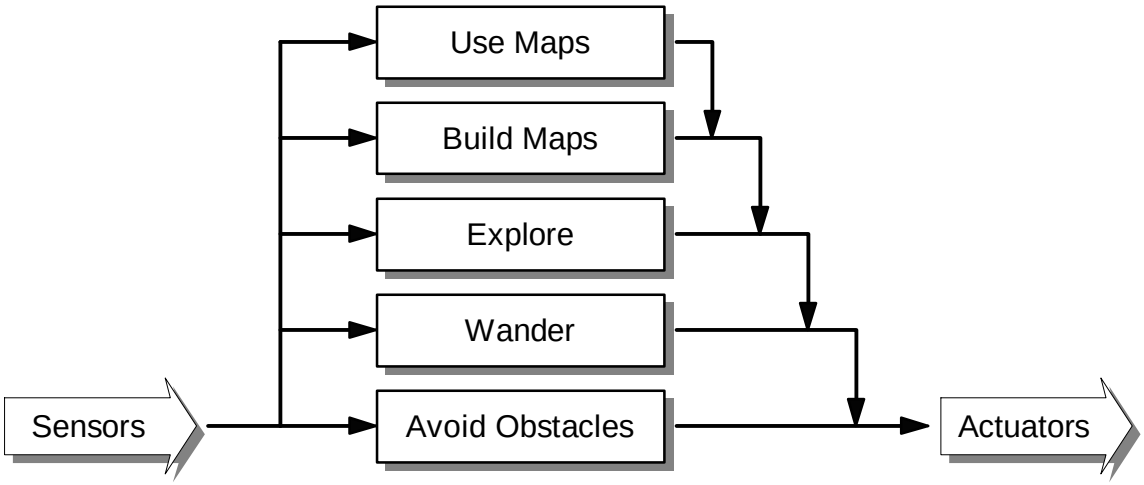


Figure 5 Layers of Control

2.6.3 The Structure of Layers

Each layer in Subsumption contains many of the elements found in classical control such as perception, task execution, motor control and so on. Unlike classical control, however, each layer only has to be concerned with a carrying out a small specific task. Brooks implements each of the layers as finite state machines. Each of these machines has an input and an output. In addition, each input can be suppressed and an output can be inhibited by wires

terminating from other finite state machines. These wires are used by higher level layers to interact with the lower level layers. Figure 6 shows one of these finite state machines.

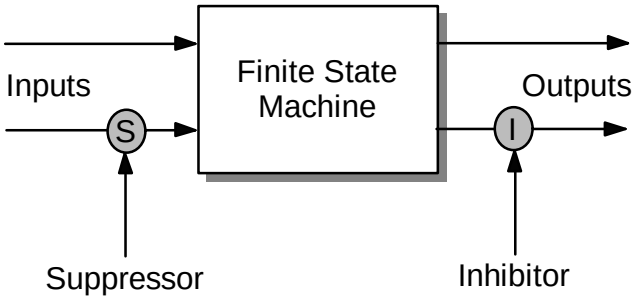


Figure 6 Finite State Machine

The inhibition wire at the output of a module is used to inhibit the output from the module. If a signal is sent along this wire, any output generated by the module will be lost. The suppression wire works in a similar manner. In this case, however, if a signal is sent along the wire, the input to the module will not only be inhibited but will also be replaced with the signal on the suppression wire.

2.6.4 Coordination in Behaviour Based Systems

In behaviour based architectures, a number of behaviours run concurrently. Many of these will try to drive the same actuator at the same time. An example of this can be seen with two simple behaviours. One behaviour is an obstacle avoidance behaviour which attempts to steer the robot away from an obstacle. The other is a light following behaviour which causes the robot to follow a light source. In normal situations (in the absence of obstacles), the light following behaviour controls the robot's motors to track the light source. If an obstacle is encountered, however, the obstacle avoidance behaviour becomes active and tries to steer the robot away from it. At this stage, both behaviours are trying to control the robot's motors, causing a conflict to occur. This is a common characteristic of all behaviour based architectures. To overcome it, an arbitration function has to be implemented. The arbitration function has to select a single behavioural response from a multitude of possible ones. There are a number of ways in which this can be done. In subsumption, a fixed

priority arbitration scheme is used. A higher level layer is able to subsume a lower level one. This is where the name subsumption comes from. Inhibition and suppression are the mechanisms by which conflict resolution between actuator commands from different layers is achieved.

2.6.5 Example Robots Using Subsumption

A number of robots have been built over the years using the subsumption architecture. The following sections give a brief description of some of these.

Allen (Brooks, 1996): This was the first robot built by Brooks to test subsumption. Three behaviours were implemented on it. The first behaviour endowed the robot with the ability to avoid both static and dynamic objects. The second behaviour made the robot wander about at random. The third behaviour was an exploration module which made the robot head for distant places. Although quite a simple robot, it highlighted all the essential features of the subsumption architecture.

Tom and Jerry (Connell, 1987): These two robots were developed to demonstrate how easily it is to implement subsumption on robots with very low computational power. Both of these robot were developed from toy cars and had three infrared proximity sensors mounted at the front and one at the rear. The software for the robot fitted on a single 256 gate programmable array logic chip. Three behaviours were implemented on the robot. The first two were similar to the robot Allen, that is an obstacle avoidance and wander behaviour. The third behaviour allowed the robot to follow a moving target.

Herbert (Brooks et al., 1987): This was a somewhat more ambitious robot. The robot used 24 8-bit processors and 30 infrared proximity sensors. It also had an onboard manipulator with sensors attached to the hand, and a laser range finder to collect three dimensional depth data. The robot had the ability to wander around an office environment and collect soda cans. It could also perform obstacle avoidance and wall following. The remarkable thing about Herbert is that it could do quite complex and useful tasks without there being any internal communication between its behaviour generating modules.

Genghis (Brooks, 1989): This is a six-legged walking robot originally built to develop walking and learning algorithms. Onboard, there are 12 motors, 12 force sensors, 6 pyroelectric sensors, 1 inclinometer and 2 whiskers. The behaviours implemented on the robot give it the ability to walk over rough terrain. The passive infrared sensors are used to detect and follow people. Developing the control system for this robot using the classical hierarchical architecture would be extremely complex. However, developing it in subsumption makes the task much more manageable. Low-level behaviours can be built up gradually, tested and debugged before higher-level behaviours are added to the system.

Squirt (Flynn et al., 1989): This is one of the smallest robots built at the MIT lab. Weighing just 50 grams, it incorporates an 8-bit computer, an onboard power supply, three sensors and a propulsion system. Two behaviours are implemented on the robot. The first behaviour monitors the output from a light sensor, causing the robot to moves in a spiral pattern until a dark area has been located, at which time it stops and remains stationary. The second behaviour only becomes active once a dark area has become established. The output from a pair of microphones is then monitored until a certain sound pattern is detected. Once the pattern is detected, the robot will head in the direction of the sound.

Toto (Mataric, 1989): This robot performed obstacle avoidance, wandering, wall following, path planning and most importantly map-building. Behaviours were implemented on the robot that recognised certain types of landmarks, such as walls corridors and so on. Another set of identical behaviours were also used which lie in waiting for new landmarks to be detected. When this happens, a behaviour allocates itself to represent the landmark that was detected. Behaviours which represent adjacent landmarks have neighbour relationship links activated between them. Together, these group of behaviours represent Toto's map. To plan a path to a particular landmark, the behaviour for that landmark is activated. This activation will spread through the adjacency links between behaviours until it arrives at the one associated with the robot's current position.

2.7 Advantages of Behaviour Based Control

Behaviour based architectures are generally more robust than hierarchical architectures making them more suitable for use in unstructured and dynamic environments. The following lists some of the important advantages which these behaviour based systems exhibit (Nehmzow, 2000).

- In behaviour based systems, there is no functional hierarchy between the different layers. It is not necessary for one layer to call upon another to carry out a specific task. Each of the layers run in parallel and work independently of each other carrying out their intended goal. Each layer can respond directly to environmental changes in a timely manner. There is no central planning module which has to take account of all sub-goals. As a result, no conflict resolution strategy is needed.

- Behaviour based systems are modular making them easy to design, test and debug. When implementing the system, lower levels of competence are added first, such as obstacle avoidance. Once this layer has been thoroughly tested and shown to exhibit the correct behaviour, further layers can be added, increasing the level of competence of the robot.

- A very important advantage of behaviour based systems is that they are robust. In a hierarchical architecture, the failure of one module leads to the failure of the entire system. In behaviour based system, however, the failure of one layer only has a minor effect on the performance of the whole system. This is because the overall behaviour the robot exhibits is the combination of a number of layers of control running concurrently. If one layer fails, the others can still function independently.

2.8 Limitations of Behaviour Based Control

One of the main limitations of behaviour based systems is that they do not cater very easily for the execution of plans (Nehmzow, 2000). Behaviour based system are purely reactive - they respond directly to sensor stimuli. By tightly coupling sensor input with actuator output, they respond very well to environmental cues. However, since behaviour based systems do not have any internal state, they are unable to follow a specified sequence of actions.

2.9 Hybrid Architectures

The traditional hierarchical architecture used on mobile robots maintains an internal world model of the environment, which it then uses to plan and execute a set of actions. In contrast, behaviour based architectures have no internal state and rely on external environmental cues to determine what actions should be taken. An important question arises here. Is a model or map of the robot's environment necessary in order to carry out purposeful and useful tasks? Consider the following example of a delivery robot travelling around an office environment in search of a specific room. If a behaviour-based architecture was used, it could have difficulty in trying to locate the room. At corridor intersections, the robot cannot use its sensors to decide what to do next. There are no environmental cues available to help it reach its goal. Since behaviour based systems determine their actions directly from sensor stimuli, the office environment would have to be extensively modified to provide the robot with a set of available cues. In cases such as this, it is more appropriate for the robot to maintain an internal model of the environment. By accessing the robot's internal state, decisions can be made about what to do next.

The problem with having an internal model of the environment, however, is that it has to be accurate to be of any use. If the environment is changing at a rapid rate, then little value can be placed on the model. Over time, new features may be introduced into the environment or known features may have changed, such as doors closing or opening. Unless the correct state of the environment is held in the model, the robot cannot be expected to generate a correct plan of actions. To overcome this, the introduction of hybrid systems has become

quite popular. These combine the rapid response time of behaviour based systems with the deliberative reasoning capabilities of hierarchical architectures. The low-level reactive components of the system effectively filter out the highly changeable aspects of the world, leaving the higher-level model-based components to deal with the constant or slowly changing features [10].

2.9.1 Example Architectures

The following sections give a brief description of some architectures based on this hybrid approach.

Atlantis (Gat, 1991): This hybrid architecture was developed at the Jet Propulsion Laboratory. It is a three-level architecture consisting of a deliberator, a sequencer and a reactive controller (see figure 7). The deliberator handles planning and world modeling. The sequencer handles initiation and termination of low level activities and addresses reactive-system failures to complete the task. The reactive controller is in charge of managing a collection of primitive activities. The control layer in Atlantis is implemented in ALFA. This is a programming language specially designed to program reactive modules. Within the architecture, conditional sequencing occurs upon the completion of various subtasks or if a failure has been detected. A feature that has been introduced into the system is the notion of "cognizant failure". This gives it the ability to recognize when a task cannot be completed so it can take corrective action. Tasks are described by a list of methods for carrying out the task along with the conditions under which the methods are useful. The deliberator performs such tasks as planning and world modeling. Each of these tasks are initiated and terminated by the sequencer. The output from the deliberator is viewed only as advice to the sequencer, with control of the robot remaining in the hands of the sequencer.

AuRA (Arkin, 1996): This was one of the first architectures developed to demonstrate the hbrid approach. It consists of two components - a deliberative hierarchical planner, based on traditional AI techniques, and a reactive controller, based on schema theory. Figure 8 shows the various components of the AuRA architecture. The hierarchical component

consists of a mission planner, a spatial reasoner and a plan sequencer. The mission planner is in charge of establishing high level goals for the robot. The spatial reasoner uses cartographic knowledge to construct a sequence of navigational path legs that the robot must ececute to complete its mission. The plan sequencer converts each path leg into a set of motor behaviours for execution. The schema controller is in charge of monitoring the behavioural processes at run time.

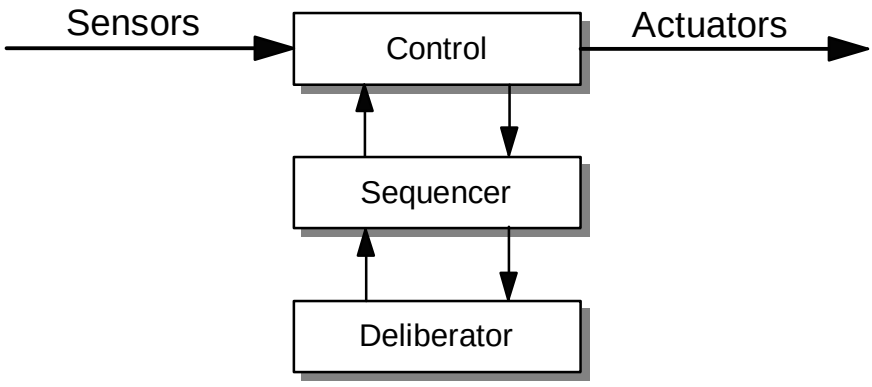


Figure 7 Atlantis Architecture

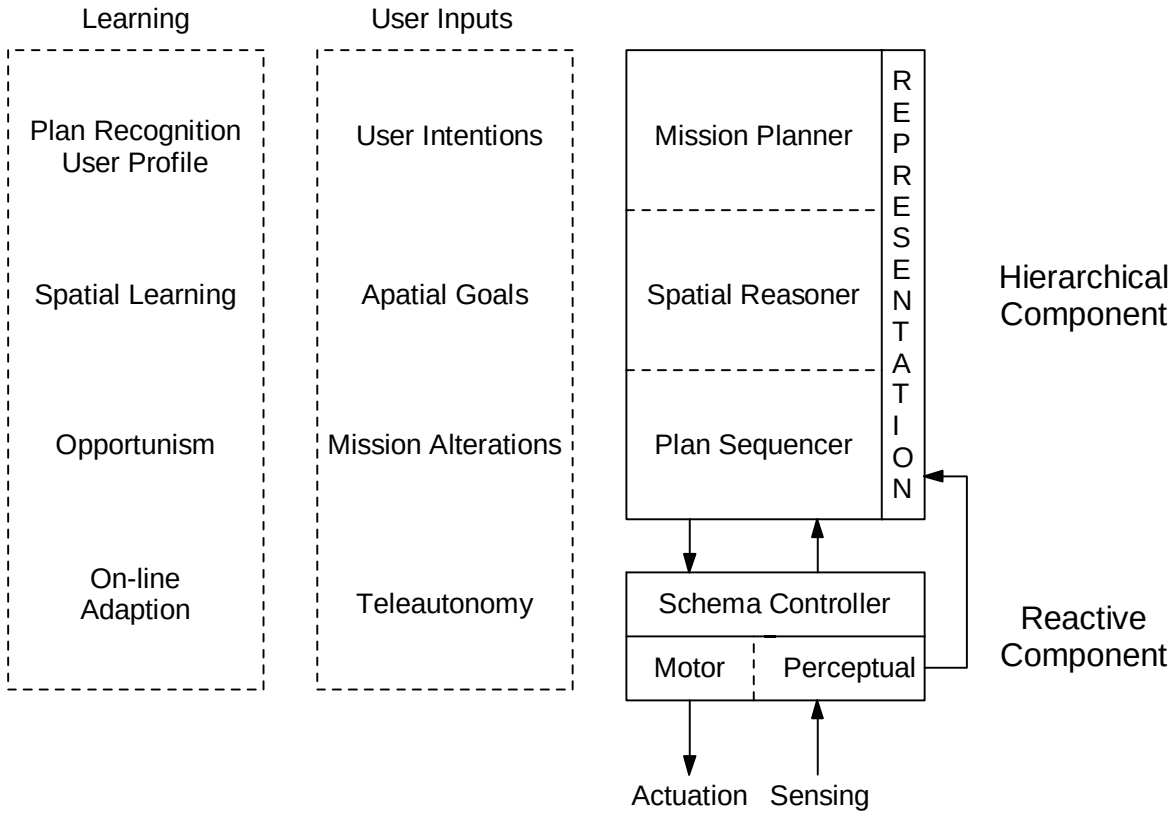


Figure 8 AuRA Architecture

2.10 Mapping

If a mobile robot contains an internal model or map of its environment, it can perform navigational tasks such as travelling to a particular room in an office building. For certain types of robots, having this capability is essential. Office cleaning robots and security robots, for example, both need a map to go about their tasks. A map in this context denotes a one-to-one mapping of the environment onto an internal representation (Nehmzow, 2000). There are a number of ways in which a map can be built up. The tasks a robot has to perform and the environment in which it operates have a strong influence on the type of map which is chosen. The sections that follow review the types of maps available and considers the applications to which they are best suited.

2.10.1 Recognizable Locations

This type of map consists of a number of distinct locations, such as doors, corners, tables and so on, that the robot can recognize as it travels about. If the robot returns to a location contained in the map, it will be able to recognize it. A number of researchers have investigated techniques which allow the robot to recognize these distinct places. Donnett (1992) used a robot with ultrasonic and infrared sensors which measured certain properties of the sensors at various positions in the environment. This allowed the robot to recognize a location if it was subsequently revisited by matching its sensor readings against the stored properties. The robot used a Bayesian process to match the data which computed the propability of the robot being at a particular location.

Nehmzow and Smithers (1991) recognized distinct places by monitoring the movement of the robot. The robot initially adopts a wall-following strategy. If any significant movement occurs, such as turning at a corner, information about the move is used to train a self-organizing neural network. Eventually, after a few circuits of the room, the neural network will be sufficiently trained to be able to recognize individual corners when information about a particular move is presented to the network.

2.10.2 Topological Maps

Topological maps are similar to maps based purely on recognizable locations. However, in this case, the maps are enhanced by the addition of links or paths which show the relationship between different landmarks or features. The map can be considered as a graph with nodes representing distinct locations. Pathways between locations are represented as arcs connecting the appropriate nodes (Kurz, 1996; Zimmer, 1996). A link which establishes the connection between two landmarks has to be identified by the robot. This is done by travelling the corresponding route between the two landmarks. To make constructing the topological more straightforward, the links can be established at the same time as the landmarks are identified. This is the approach used by Mataric's robot (Mataric, 1990). The robot navigates by wall-following and landmarks are identified in the sequence corresponding to their topological relationships.

2.10.3 Metric Topological Maps

In topological maps, no metric or geometric information is stored. This has the advantage that it eliminates the problem of accumulating odometry errors. To make topological maps more useful, however, they are often extended by incorporating some metric information into them such as the length of paths between landmarks. The advantage of adding this information is twofold. Firstly, it can increase the efficiency of the path planner and secondly it can remove any ambiguity between similar landmarks (Lee, 1996). When a robot is travelling between two landmarks, it is usually desirable for the robot to travel along the shortest path. With a topological map, there may be a number of routes between the two landmarks. Having the addition of path length information allows the path planner to choose the shortest route. Path length information is usually obtained from onboard odometry. The problem with odometry, however, is that it is notoriously unreliable, producing errors that gradually accumulate over time. With topological maps, the distance between landmarks is usually small, and the errors that do result are usually acceptable.

2.10.4 Area-Based Metric Maps

This type of map divides the robot's environment into a number of distinct regions or cells. Each of the cells represents a particular location in the environment. Having a large number of cells allows a detailed model or map of the environment to be built up. Associated with each cell is a number representing some property of the cell. Typically, this indicates the occupancy state of the region that the cell represents. For example, the region may be occupied, empty or the state unknown. The map is updated from data coming the robot's sensors such as ultrasonic and infrared range data. A number of maps based on this technique have been developed over the years. Popular examples include Occupancy Grids (Elfes, 1989), Certainty Grids (Moravec, 1988) and Histogram Grids (Borenstein, 1991). In order for a cell to be updated from a sensor reading, it is necessary to determine which cell the data refers to. This is complex to do using sonar sensors due to the wide beam angle. A return reading from a sonar sensor may refer to any obstacle within a 30° range. To

overcome this, various sonar models have been proposed, which translate the sonar readings into occupancy probabilities of the cells within the sonar's beam.

2.11 Choosing a Map

The type of map that is most suitable for a particular robot application is dependent on the environment in which the robot operates and the tasks it is expected to perform. Topological maps are well suited to environments that are dominated by distinctive landmarks which the robot can easily recognize. If the robot's task is to travel between different landmarks, then a topological map is most appropriate. To be effective, however, there must be clear unambiguous paths between the landmarks that the robot can travel. A typical environment where this map can be used quite effectively is an office environment with many doors and corridors.

In contrast, if the robot's application deems it necessary to travel within a large open space occupied by obstacles, then an area-based map is more appropriate. Within this type of environment, there may not many distinct features that the robot can recognize. For such environments, the robot would need to be equipped with a map showing the full metric relationship between all objects in the area. A warehouse robot, for example, would fall into this category.

2.12 Summary

This chapter has provided a literature review which is relevant to the work discussed in this thesis. An important discussion into robot architectures was carried out. It was shown that behaviour based control is more robust and reliable in typical real world environments. The merits of behaviour based control as opposed to hierarchical control were discussed. Despite the major advantages of behaviour based architectures, they do have some limitations - mainly the inability to easily execute plans. This limitation was addressed along with the introduction of the hybrid architecture which is used to overcome it. Examples of both behaviour and hybrid architectures were given. A summary of the types

of maps commonly used mobile robots was also given. It was argued that the type of map most appropriate for a particular robot is dependent on the proposed application and the environment in which the robot is expected to operate.

3. ROBOT DESIGN

This chapter describes both the physical and hardware design of the robot. All of the robot's sub-modules are described in detail and how they interact with each other.

3.1 Locomotion System

Various drive configurations can be used to control the locomotion of a mobile robot. Popular examples include differential drive, synchro drive, leg driven and track driven drive systems. For practical reasons and ease of implementation, differential drive systems incorporating wheels are generally the most preferred option. Such a system also offers a great deal of maneuverability for the robot. One of the disadvantages of wheeled systems, however, is their inability to tackle rough terrain. For such environments, tracks or legs offer a greater degree of flexibility in negotiating obstacles. If the intended work area for a mobile robot is an indoor environment, however, a wheeled system is more than sufficient. On this robot, a differential drive system consisting of two servo motors is used. With a differential drive system, the robot can move in a straight line, follow an arc and turn on the spot, which is useful in negotiating tight corners. To steer the robot, the speed and direction of each motor must be controlled.

When deciding on the type of motors to be used, various parameters have to be taken into account. These include such things as the robot's weight, the speed and acceleration required, wheel diameter and so on. A suitable motor can be chosen based upon these parameters. A detailed calculation is shown below for determining the motors used on this robot:

Mass of Robot = 3 Kg

Wheel Radius = 1.5"

Maximum Speed = 2.08 revs/sec

= 0.5 m/s

= 13.06 rads/sec (2Π * 2.08)

If maximum speed is reached in 0.5 seconds, then the acceleration required is:

Acceleration = 13.06/0.5

= 26.12 rads/sec/sec

Inertia of Robot = mr^2

Where: m is the mass of the robot and r is the wheel radius.

1.5" is equal to 0.0381 m

= 3 * $(0.0381)^2$ = 0.00435 Kg m^2

If it's assumed that each motor only sees half the inertia of the robot, then the inertia for each motor will be 0.00435 / 2 = 0.002175 Kg m^2.

Acceleration Torque = Inertia * Acceleration

= 0.002175 * 26.12

= 0.0568 Nm = 56.8 mNm

Power = Acceleration Torque * Maximum Speed

= 0.0568 * 13.06

= 0.7418 Watts

Using a 30:1 reduction gearbox, the acceleration torque is reduced by a factor of 30. Also the inertia seen by the motors is reduced by a factor of 30^2 = 900.

Acceleration Torque = 0.0568 / 30 = 0.001893 Nm

= 1.89 mNm

Reflected Inertia = 0.002175 / 900

= 0.00000241 Kg m^2

= 2.41 g cm^2

Motors inertia is 24.8 g cm^2.

3.2 Mechanical Deisgn

One of the objectives in designing the robot was to make it small, compact and lightweight so that it could easily be transported and could operate in tight locations. To accomplish this, a cylindrical design using a differential drive system was opted for (see figure 9). The advantage of such a design is that the robot cannot be caught in tight corners since the differential drive system allows the robot to turn on the spot.

Figure 9 Robot

The base of the robot consists of a circular sheet of aluminum 30cm in diameter. Located beneath the plate are two 12V DC motors, one on either side. These are positioned in specially designed mounts and connected to each wheel using a rubber pulley transmission system (see figure 10).

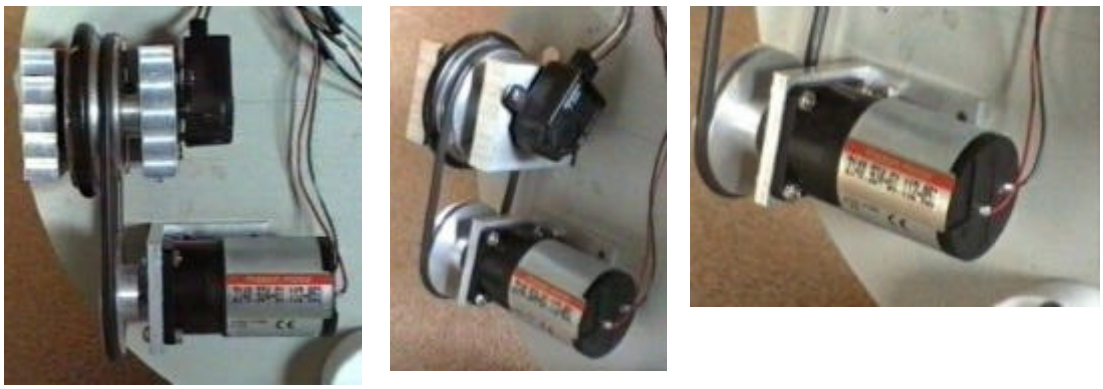


Figure 10 DC Motor and transmission system

Two castor wheels located at the front and back of the base plate are also used in order to further support the robot (see figure 11).

Figure 11 Castor Wheel

Attached to each wheel is an optical incremental shaft encoder that measures wheel displacement as the robot travels about (see figure 12). By attaching the optical encoder to the wheel rather than to the motor itself, a more precise measure of wheel displacement can be obtained. Slippage of the pulley against the motor's drive wheel will not be recorded in this way.

Figure 12 Shaft Encoder

An aluminum frame is located on top of the base plate that allows a number of PCBs to be stacked. These can be easily slided in and out. A total of 5 boards can be used. However, only 3 spaces are occupied on the current design. Located at the bottom of the frame are the 12V battery and the switched mode power supply (see figure 13). On top of the frame is another circular sheet of aluminum 30cm in diameter. This strengthens the structure making it more rigid and also holds the mounts for each of the sonar sensors. The mounts have been designed so as to allow the sensor to be rotated in all three axes (see figure 14). This allows

very accurate positioning of the sensor to be achieved. Two of the sensors are located at the front of the robot, the rest are equally distributed around it. The 3 ultrasonic control boards for the sensors are located on the side of the frame and held in position using some velcro tape. The weight of the robot including the 12V lead acid battery is approximately 3 kg.

Figure 13 Frame where the PCBs are stacked

Figure 14 Holder for Sonar Sensor

3.3 Hardware Design

The electronic control hardware for the robot has been designed in the form of a distributed control architecture, in which a number of separate modules are used to perform complex tasks. Currently, three modules exist, these being a locomotion board, a sonar ranging board and a central controller. There is also a power supply board to control the onboard circuitry. Further modules can be added as required by extra sensory capabilities and implemented behaviours. The advantage of distributing the workload to different modules is that it frees the main processor or controller from performing repetitive albeit complex time-consuming tasks such as the servo control of individual drives. Figure 15 gives a block diagram of the hardware. The following sections give an in-depth discussion of each module.

3.4 Central Controller

A thorough review of various CPU modules was carried out prior to the construction of the robot. This was to determine the various types of features available, the ease of implementation with other hardware, prices and so on. In the end, it was decided to use an embedded 586 CPU module for the robot. While this offers far more processing power than is necessary, it will allow for future enhancement of the robot such as the implementation of relatively sophisticated vision control algorithms. This module has the advantage that it can be easily programmed using traditional PC based programming software such as Borland C++. There is also an abundant source of literature available on how to interface with the PC based architecture. The processor runs at 133 MHz and has 8 MB of RAM and a 2 MB solid state hard disk. While a 2 MB hard disk may seem rather small, it offers sufficient space for the software running on the robot. If necessary, a larger hard disk can always be added in future.

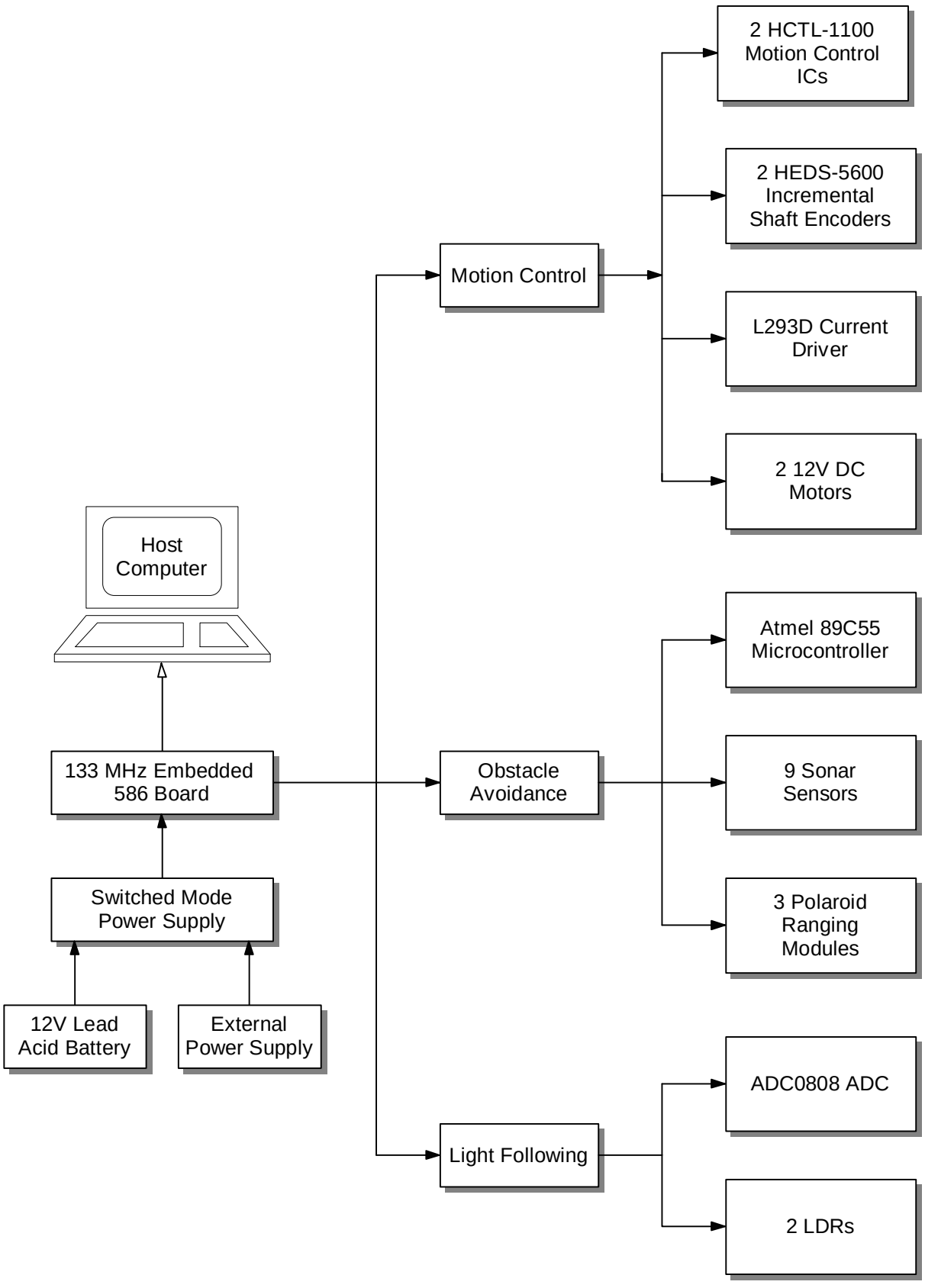


Figure 15 Hardware

The module used is a CMi586DX133 PC/104 module from Real Time Devices. PC/104 is a specification for a compact version of the traditional PC bus optimised for the unique requirements of embedded systems applications. In general, these boards have a much smaller footprint, reduced power consumption and have a self stacking bus, which allows other boards to be stacked on top of them. These features are ideal for the robot, which has to be compact and capable of operating from a low capacity battery.

Feature wise, the CPU module has two RS232/422/485 serial ports, an ECP/EPP parallel port, a floppy and hard disk interface, a keyboard interface and a real time clock. One of the unique features of the module is that it can be operated in virtual mode. This allows you to gain access to the device without having to connect a keyboard, monitor and floppy disk. The module's serial port is connected to the serial port of a host PC and the host PC's keyboard, monitor and floppy disks are made available to the CPU module. When operated in this mode, less hardware has to be connected to the robot, keeping the cost, complexity and weight to a minimum. All software development is carried out on a host PC and is downloaded to the controller over a serial communications link.

3.5 Power Supply Board

The purpose of the power supply board is to provide all of the robot's electronic control circuitry with a regulated 5V output. The robot can take its power from either an onboard 12V lead acid battery or an external power supply. The advantage of using lead acid batteries over other types of rechargeable batteries is that they can provide energy densities of up to 40 Wh/Kg and have a long shelf life. One of their drawbacks, however, is that they are large and heavy relative to other battery technologies. To increase the efficiency of the power being used, a switched mode power supply board was built. This provides a regulated 5V output for the robot's control circuitry. The advantage of using a switched mode power supply is that it offers a far greater efficiency than a standard power supply, resulting in less heat being produced. This avoids having to use a larger heat sink and is

more economical in terms of battery power. This is important since the efficiency of the power supply becomes an important factor when the robot is using the onboard battery.

A switched mode power supply works by rapidly switching an input voltage on and off. By varying the duty cycle of this period, different output voltages can be obtained. The duty cycle refers to the fraction of time a signal is held high within a cycle. For example, a duty cycle of 20% means that the input voltage is on for 20% of the time and off for 80% of the time. With this duty cycle, the output voltage would be one fifth of the input voltage. In order to maintain a fixed output voltage, negative feedback is used to control the duty cycle ratio. To smooth the output voltage, an LC filter is used.

The power supply was built using an SGS Thompson L4960 switching regulator IC. This has been specifically designed for use in switched mode power supplies. It can provide an output voltage from 5V to 40V at up to 2.5A with an efficiency of up to 90%. To build a power supply using the IC, it is necessary to add other components into the design. These include resistors, capacitors, diodes and inductors. To offer protection from reverse polarity at the input to the regulator, a 1N4004 diode is used.

The power supply board has 5 output connectors, four of which provide a regulated 5V output and the other which provides an unregulated 12V output. The 12V output connector is used to power the motors. This connector gets its voltage directly from the battery and has no form of reverse polarity protection. It was decided not to use any protection, since this would have resulted in 0.7V being drop across a diode. While this wouldn't present a problem when using an external power supply which could pump a large voltage into the regulator, it could cause problems when the onboard battery is used. The output from the battery may drop below 12V or near to it at times. If a diode was used, there would be a smaller input voltage to the motors. Since it is desirable to have 12V for the motors or as near to it as possible, it was decided not to use a diode.

3.6 Noise Interference

When designing an electronic system, it is important to consider noise interference. There are several different types of noise, but the most common are EMI (electromagnetic interference) and noise caused by inductive loads such as motors and relays. The presence of noise in an electronic circuit can create fault conditions of varying severity. In the worst case, noise problems can cause catastrophic (or unrecoverable) failure of an electronic circuit. It can cause microprocessors to reset unexpectedly, analog measurements to fail and erroneous signals to trigger.

Sources of EMI include microprocessors, microcontrollers, electrostatic discharges, transmitters and transient power components. Within a microcontroller system, the digital clock circuitry is usually the biggest generator of wide-band noise, which is noise that is distributed throughout the frequency spectrum. These circuits can produce harmonic disturbances up to 300 MHz. Noise can be coupled into a circuit through conductors. If a wire runs through a noisy environment, the wire will pick up the noise inductively and pass it into the rest of the circuit. An example of this type of coupling is found when noise enters a system through the power supply leads. Once the noise is sourced in the power supply lines, it is then conducted to all circuits needing power. Coupling can also occur on circuits that share common impedances. Similarly, it can occur with radiated electric and magnetic fields which are common to all electrical circuits. Whenever current changes, electromagnetic waves are generated. These waves can couple over to nearby conductors and interfere with other signals within the circuit.

It is important to have a good ground layout in a design to minimise ground interference in the system. All ground lines have some finite impedance causing voltage drops to occur. It is these voltage drops that are the cause of the interference in the system. As system frequency increase, so too does the interference. In high frequency digital circuits, current spikes are created when transistors are switched on and off. These spikes can be calculated using the following equation:

$$V = L\frac{di}{dt}$$

Ground noise can be reduced by providing low-impedance pathways for all return signals. There is another type of noise in a circuit called power system noise. This can be decoupled using filters unlike ground noise which cannot. When a logic gate switches, a transient current is produced on the power supply lines. The impedance on the power supply lines along with the sudden current flow creates a voltage drop on the VDD terminal of the chips. The current needed by a chip can be supplied from a nearby decoupling capacitor. This reduces the load on the power supply lines and removes unwanted glitches in the power system. High frequency, low inductance capacitors of around 0.1 μF in size should be used.

These decoupling capacitors are liberally used throughout the main control circuitry of the robot.

3.7 Obstacle Detection and Avoidance

It is important to realize that the environment in which a robot must operate is usually dynamic and constantly changing - that is, the environment has not been specifically engineered for the robot. While the dynamics of the robot can be accurately determined and modelled, the same cannot be said about its environment. Obstacles such as chairs and furniture pose a constant threat to the robot. One way to overcome these problems is to have the robot acquire information about the environment, such as the presence of obstacles, at run-time. There are several ways in which this can be accomplished. Many robots detect an obstacle by making physical contact with it. This is known as tactile sensing. Others determine the distance to an obstacle through non-contact means through the use of a sensor. Depending on the accuracy required, various sensors can be used such as sonar, infrared, laser range finders and vision systems. The quality of information received by these sensors is limited by the sensor's accuracy, it's field of view and the manner in which the robot perceives the information. It is important that these issues are kept in mind when designing and implementing the collision avoidance and detection systems.

3.7.1 Echolocation

The majority of robots use echolocation or sonar (SOund Navigation And Ranging) as a means of detecting or locating obstacles. This is a technique whereby the distance to an object can be determined by using echoes of ultrasonic sound. An ultrasonic pulse is transmitted towards a target which is then reflected. By measuring the time from the moment the pulse was transmitted to the time an echo was received, the distance to the target can be calculated. One of the main advantages of sonar over vision is its fast processing time. It is also quite accurate in most situations. However, it does have some limitations which will be discussed later.

There are three primary characteristics associated with sonar. These include the transmission medium in which the sound travels, the velocity of propagation and the wavelength of the sound. Unlike electromagnetic radiation, sound requires a medium for transmission. Common transmission mediums for sound include air and water. The velocity at which sound waves travel is much slower than light waves, enabling a controller to use the "time-of-flight" principle to estimate an object's distance. The wavelength of a 50 kHz sound wave is 6.872 mm. This is large enough to overcome the roughness of indoor surfaces and yield fairly accurate results.

By measuring the time of flight of a transmitted pulse, the distance to an obstacle can be determined using the following equation:

$$d = \frac{1}{2}vt$$

where: d = distance to the target in meters

v = speed of sound in ms^{-1}

t = time of flight in seconds

The speed of sound is proportional to the square root of the temperature in degrees kelvin. Its speed at a given temperature is:

$$v = 331.6\sqrt{1 + \frac{T}{273.15}}$$

where: v = speed of sound in ms^{-1} for a given temperature

T = temperature in °C

The speed of sound at a standard temperature of 15°C is:

$$v = 331.6\sqrt{1+\frac{15}{273.15}}$$
$$v = 340.58 ms^{-1}$$

Therefore, sound travels at 1117 feet per second at 15°C.

To calculate the distance to an obstacle, the following equation can be used:

$$d = \frac{1}{2}331.6t\sqrt{1+\frac{T}{273.15}}$$

This equation can be simplified by using a standard temperature of 15°C. This results in the following:

$$d = \frac{1}{2}340.58t$$
$$d = 170.29t$$

3.7.2 Limitations and Difficulties of Sonar

One of the limitations of ultrasonic sensors is that they have very poor directional resolution. The ultrasonic signal is typically transmitted out in a cone of 30°. Any object that lies within this cone is capable of producing an echo. All that can be said therefore is that if an object is detected that it lies somewhere within this cone. With sonar, the shortest distance to an object within the cone is measured. This distance may not necessarily be the distance along the centreline of the cone. An example of this is shown in figure 17. Instead

of the sensor measuring the distance to point P, which lies along the centreline, it measures the distance to point D1 instead. Figure 16 shows the typical lobe pattern for a Polaroid ultrasonic sensor. This shows the signal strength of the ultrasonic pulse for different angles within the cone. It can be seen that the signal strength is greatest directly in front of the sensor and reaches a local minimum at about 15° either side of centre. The signal strength then increases again to create weaker side lobes. These side lobes have a sensitivity approximately 10 dB lower than the main central lobe.

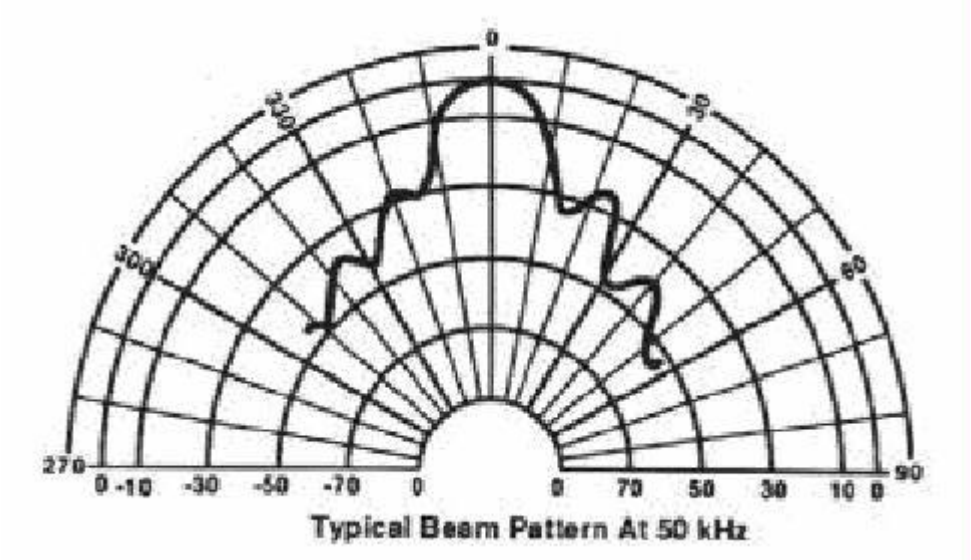


Figure 16 Sonar Beam Pattern

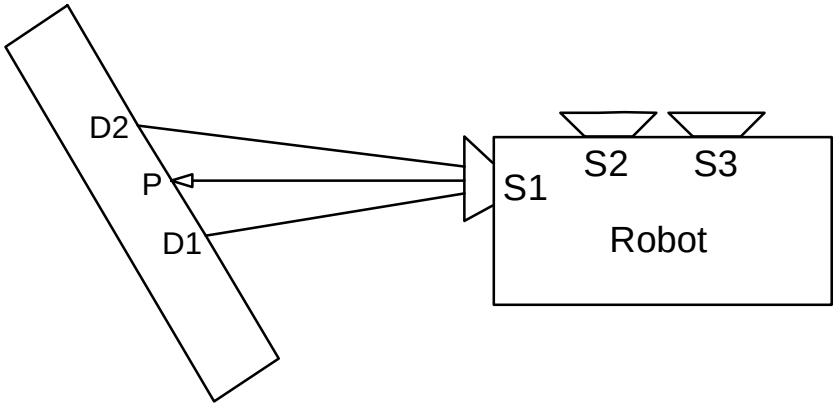


Figure 17 Measureing the Shortest Distance to an Obstacle

The typical wavelength of a 50 KHz ultrasonic beam is 6.87 mm. This is greater than the roughness of most target surfaces resulting in the beams being reflected of these surfaces. This is a phenomenon known as specular reflection. One of the problems with specular reflection, unfortunately, is that it can result in crosstalk. Crosstalk is where an ultrasonic sensor receives the signal transmitted by another sensor and assumes that it was transmitted by itself.

There are two different types of crosstalk - direct crosstalk and indirect crosstalk. Figure 18 shows an example of direct crosstalk. In the figure, sensor 2 lies within the critical path of sensor 1. The critical path is defined as any path of ultrasonic waves transmitted by one sensor and received by one or more sensors resulting in crosstalk. For this example, sensor 2 is fired a short time after sensor 1. As sensor 2 waits for an echo to be received from its own signal, it receives the echo from the signal transmitted by sensor 1. This would result in an incorrect reading being generated by sensor 2. Fortunately, the way to overcome direct crosstalk is quite simple and can be achieved purely through software. The way this is done is explained later in the section on software implementation.

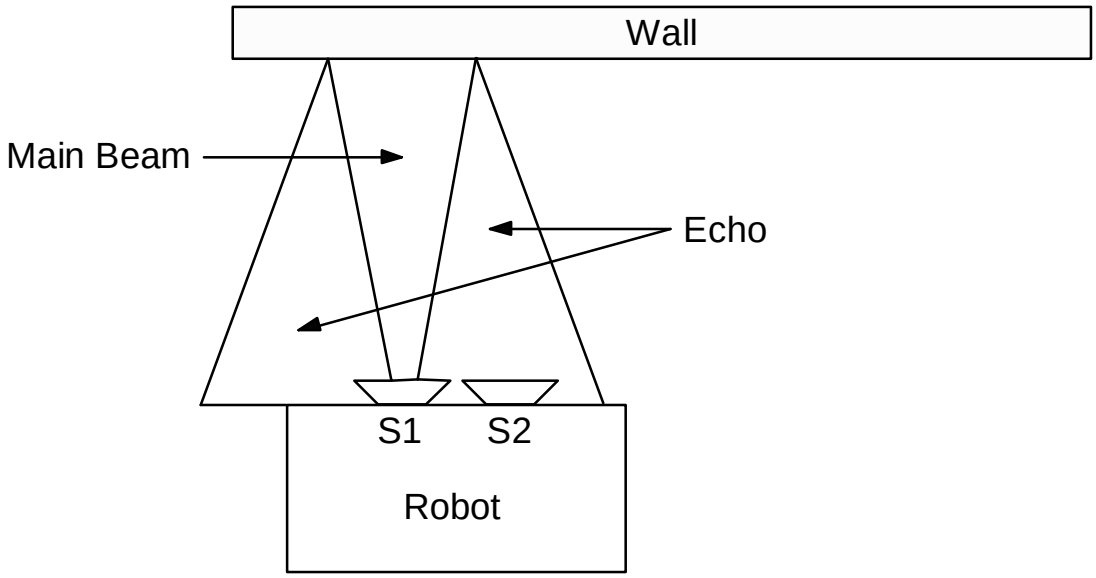


Figure 18 Direct Crosstalk

The other type of crosstalk is known as indirect crosstalk. Figure 19 shows an example of this. This type of crosstalk results from signals being reflected of various surfaces and

being detected by another sensor. In the example, sensor 1 transmits a signal which is reflected of surfaces W1, W2 and W3 and detected by sensors 3 or 4. If at this time sensors 3 or 4 were waiting for their own echoes, they would interpret the reflected signal from sensor 1 as being their own. This would result in an incorrect reading being generated. Unfortunately, indirect crosstalk can never be eliminated completely.

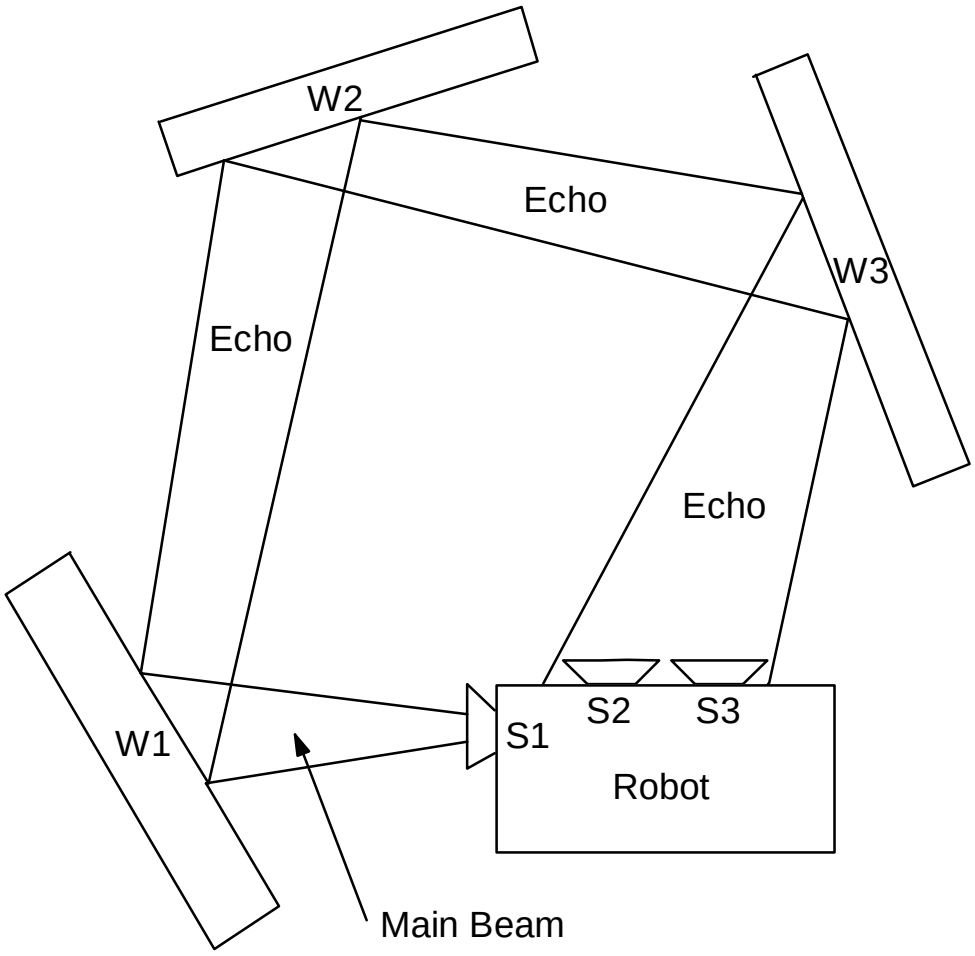


Figure 19 Indirect Crosstalk

3.7.3 Polaroid Ultrasonic Ranging Module

The robot uses a Polaroid ultrasonic ranging module for determining the distance from objects in its vicinity. The module acts as both a transmitter and a receiver. It is designed to transmit an outgoing signal and to function as a receiver for the reflected signal (the echo).

The module consists of an acoustical transducer and a ranging circuit board. The diameter of the transducer determines the acoustical lobe pattern, or acceptance angle, during the transmit and receive operation. A small thin gold-plated plastic foil forms the outer surface of the transducer. Below this foil rests a concentric groove aluminium backplate. These two conductive surfaces act as a capacitor when 400 V is applied to the backplate. When the transducer is acting as a transmitter, the voltage across the capacitor is varied, thus varying the electrostatic force applied on the foil. This change causes the foil to act like a speaker propagating sound waves into the air. The reflected wave applies a varying pressure to the gold foil that alters the capacitance of the conductive surface. This changing capacitance results in variable voltages that can be sampled by a controller. The width of the transmission beam is approximately 30° . Figure 20 shows a picture of the ranging module and the ultrasonic transducer.

Figure 20 Polaroid Ranging Module and Ultrasonic Transducer

When the unit is activated by an INIT signal, the transducer emits a sound pulse, then waits to receive the returning echo from whatever object the sound pulse has struck. The emitted pulse is a high frequency inaudible chirp consisting of sixteen pulses and lasting for a period of approximately 0.5 ms. The pulses consist of multiple frequencies to minimise the effects of sound absorption that occurs in some materials. Another benefit of using multiple frequencies is that if a diffuse reflection occurs, the echo may cause a large enough phase shift that it cancels out other echo signals. In most cases, the first echo is the one that is measured. One of the problems the module has to overcome is the loss in strength of the returning echo as it passes through the air. The further the signal has to travel, the weaker it will be. To overcome this, the module has an amplifier whose gain increases over time.

Apart from power and ground lines, the sonar ranging module contains 3 extra pins which must be used. These include an INIT pin, an ECHO pin and an INHIBIT pin. Each of these is described below.

INIT	This is used as an activation pin controlling the transmission of pules to the ultrasonic transducer. Whenever the pin is brought high, a series of 16 pulses are transmitted.
ECHO	This pin is activated whenever the module receives a returning echo.
BINH	Normally, whenever an ultrasonic pulse is sent by the transducer, a period of 2.38 ms exists before the ranging module can detect a returning echo. This time period is used to eliminate ringing of the transducer from being detected as a returned signal. In this case, the minimum distance which can be measured is 1.5 feet. If distances shorter than this need to be measured (up to 6 inches), the INHIBIT function of the module must be used. The pin is brought high after a time period of approximately 1 ms. This allows distances of around 6 inches to be measured.

Figure 21 shows a typical timing diagram for the Polaroid ranging module.

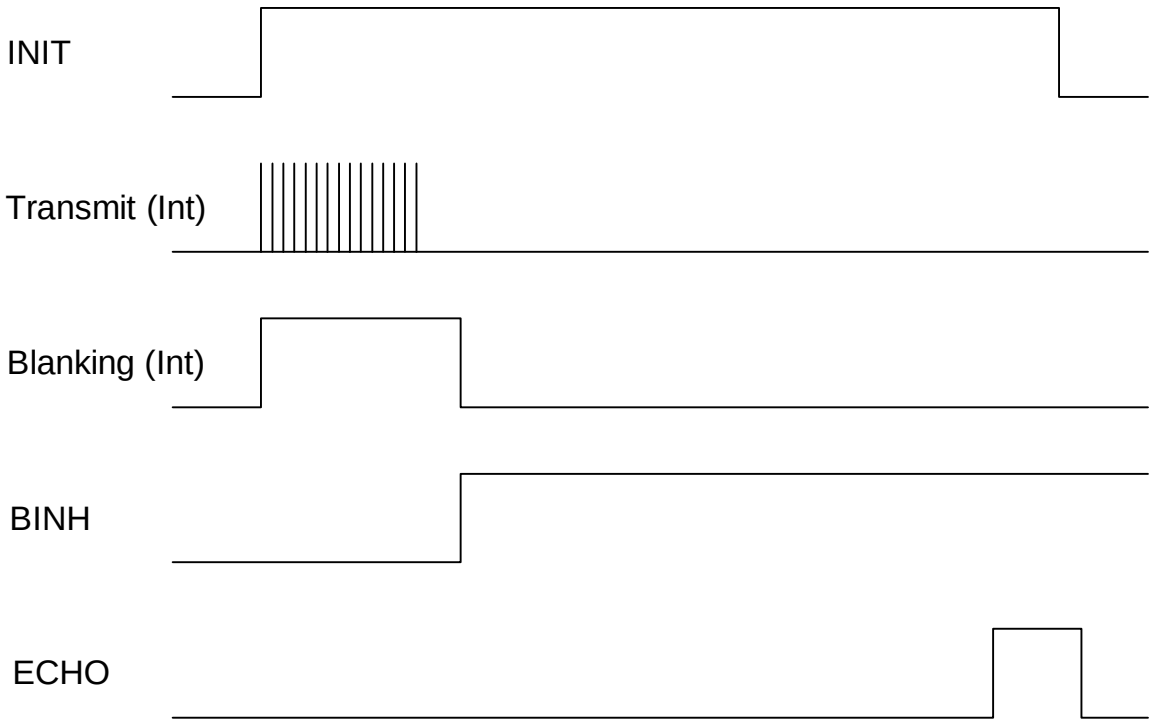


Figure 21 Timing Diagram for a Polaroid Ranging Module

3.7.4 Sonar Ranging Board

A sonar sensor board has been designed and built to perform all the functions necessary for range detection. The board is based upon an 8051 microcontroller and communicates with the main central controller over a low bandwidth serial communications link. Various different versions of the 8051 exist. The one used here is the Atmel 89C55. This is a low power 8-bit microcontroller based on the standard 8051 architecture. It has 16K of EEPROM memory on board, 4 input/output ports, serial communication and timer capabilities and the provision for generating interrupts. Most of these features are used by the sonar board.

A number of tasks have to be performed by the 8051. It has to control various signals going to the Polaroid ranging boards, it must receive confirmation of returning echo signals, it must perform timing calculations and lastly communicate with the main central controller. The ranging detection system for the robot consists of 9 sonar sensors and 3 Polaroid ranging modules. Ideally, each sensor would have its own ranging module. This would be

bulky and expensive, however, and it was decided instead to use just 3 ranging modules and use some form of multiplexing. The multiplexing system is split into 3 groups. Each Polaroid ranging module must control 3 sonar sensors. For this system to work there has to be a way of switching each sonar sensor on and off at various times. This is to avoid all three of them being connected to the ranging module at once. This is done using STP4NB50 power mosfets. One mosfet is used for each sonar sensor. The mosfet can switch the sensor on and off under the control of a GATE signal generated by the 8051. The negative plate of each sensor is connected to the drain of the mosfet. The positive plate is connected to the E1 connector of the ranging module. The source of the mosfet is connected to the E2 connector of the ranging module.

Two ports on the 8051 are used to generate the gate signals to switch the mosfets on and off. Port 0 and Port 2 are used for this purpose. Each of the gates on the mosfets require a pull up resistor to work correctly. Port 2 already has internal pull up resistors on the chip itself, so it is only necessary to add external resistors for Port 0. Through experimentation, it was found that 4.7KΩ resistors seemed to work quite well. In addition to the gate signals, the 8051 must also generate the INIT and BINH signals for each ranging module. This requires the use of 6 output lines. Only two of the pins on Port 0 were used for the gate signals, so the other 6 (pins 0 to 5) can be used for this purpose. Each time an echo is received by a ranging module, the module generates an active high ECHO signal that is read by the 8051. The echo signals for the 3 modules are connected to Port 1 (pins 0 to 2) of the 8051.

3.7.5 Serial Communications

The 8051 microcontroller contains a bi-directional serial I/O system. This consists of an RXD (Receive) and a TXD (Transmit) pin. There are no automatic handshaking signals provided by the 8051. An RS232 serial data link is particularly suitable for slow transmission speeds. It permits a simple bi-directional connection to be achieved by using just the RXD and TXD pins and a ground pin. The RS232 standard requires bipolar levels of +/- 5V to +/- 15V to operate. Since the 8051 can only provide 0V and 5V signals, a driver/receiver is required between the 8051 and the serial port. The driver/receiver

converts the signals from the 8051 into the signals required for serial communication. A MAX232 is used for this purpose.

To use the serial communications system on the 8051, various registers must be configured. These registers specify such things as baud rate, number of data bits and stop bits and parity. The transmission protocol used between the sonar ranging board and the main central controller consists of a baud rate of 2400, eight data bits, one stop bit and no parity (2400 8N1).

The crystal used on the 8051 is an 11.0592 MHz one. This odd value was chosen so that very accurate baud rates could be set. One of the 8051's timers is used as a baud rate generator. It is set up in Mode 2. This is an auto-reload mode which reloads a value into the timer's counter whenever the counter overflows. The reload value is calculated using the following equation:

$$Baud\ Rate = \frac{2^{SMOD}\, x(Oscillator\ Frequency)}{384x(256 - TH1)}$$

SMOD is a bit in the 8051's PCON register. It can be either 0 or 1, thus 2^{SMOD} can be either 1 or 2. This acts as a multiplication factor enabling the baud rate to be doubled. For a baud rate of 600, the reload value TH1 works out at D0**H**. The following is a table listing various baud rates and their equivalent reload values:

Baud Rate	**TH1**
300	A0**H**
600	D0**H**
1200	E8**H**

To initialise serial communications, a value of 52**H** is written to the 8051's SCON register. This sets up the serial communications system in Mode 1. In this mode, ten bits are transmitted (through TXD) or received (through RXD). These ten bits consist of a start bit

(0), eight data bits (LSB first) and a stop bit (1). The 8051's timer must be anabled for communications to commence. This is achieved by setting bit TR1 in the TCON register.

3.7.6 Software Implementation

This section describes the software used on the sonar ranging board. All software development was carried out in C using a special 8051 C compiler. Figure 22 shows a flowchart for the software. At the start of the program is an initialization routine that sets up various registers. Serial communications are set up for a baud rate of 2400, 8 data bits, no parity and one stop bit (2400 8N1). Timer 0 is also set up as a 16 bit timer incrementing every machine cycle. On the 8051 a machine cycle consists of twelve oscillator periods, thus using an 11.0592 MHz clock, a machine cycle lasts for 1.085 μs. The counter therefore increments once every 1.085 μs. This timer is used to measure the time of flight of an ultrasonic pulse.

The 9 sonar sensors used on the robot are split into 3 groups. Each group is controlled by one Polaroid ranging module. The first group consists of the sensors facing 0°, 45° and 90°. The second group consists of the sensors facing 135°, 180° and 225°. Lastly, the third group consists of the sensors facing 270°, 315° and 360°. To switch a sensor on or off, a 1 or a 0 is written to the port pin connected to the gate of the sensor's mosfet. Only one sensor in each group can be on at a time, the other two must always be off. With this arrangement, 3 sensors can always be on at any given time. This brings about another group of sensors, known as firing sensor groups. These are sensors that are grouped together that can all be activated at the same time. In each of these groups, there are 3 sensors. The first group consists of the sensors facing 0°, 135° and 270°. The second group consists of the sensors facing 45°, 180° and 315°. Lastly, the third group consists of the sensors facing 90°, 225° and 360°.

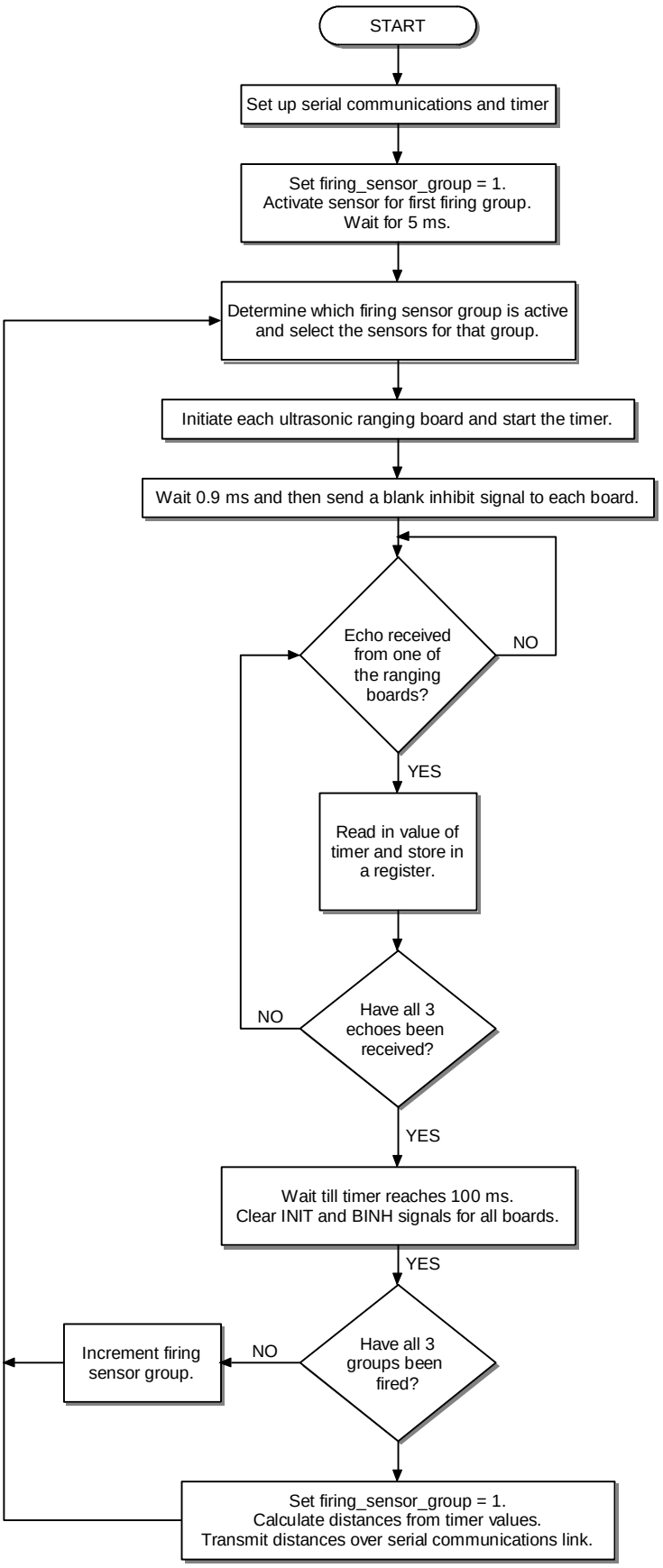


Figure 22 Sonar Software Flowchart

Initially, when the program starts, all the sensors in the first firing group are activated. This consists of the sensors facing 0°, 135° and 270°. Each of these sensors is connected to a different Polaroid ranging module, so this allows all of them to be activated at once. Activating a sensor means writing a 1 to the gate of the sensor's mosfet. This connects the sensor through an electronic switch (the mosfet) to the ranging module. This is done initially at the start of the program, so that one sensor is connected to each ranging module.

The main section of the program consists of a loop that continually repeats itself. The program determines which firing sensor group is currently active and switches on all the sensors for that particular group. All the other sensors are switched off. Each ranging module is then given an INIT signal to initiate it causing an ultrasonic pulse to be transmitted. Also at this time, Timer 0 is cleared and instructed to start counting. In order to prevent ringing of the transducers from being detected as a return signal, the receive input of the ultrasonic ranging module's control IC is inhibited by internal blanking for 2.38 ms after the initiate signal. With this blanking time, only distances of 1.33 ft and over can be measured. To detect objects closer than this, the internal blanking time must be reduced. This is achieved by taking the BINH (blank inhibit) input high. By taking this input high after 0.9 ms, distances of as little as 6 inches can be measured. The next task, therefore, that the program has to perform is to cause a delay of 0.9 ms. Since Timer 1 increments once every 1.085 μs, it is simply a matter of reading in Timer 1's value and waiting till it reaches 830 (1.085 μs * 830 = 0.9 ms). After this short delay, each of the BINH inputs on the ranging modules are activated.

The program then waits until it receives an echo from each ranging module. If an echo is received, the value of Timer 0 is read. Depending on which firing group is active and the module from which the echo came, it can be determined which sonar sensor produced the echo. The value of Timer 0 is then stored in a variable for that particular sensor. Once the echoes from all 3 ranging modules have been received, the INIT and BINH inputs for the modules are brought low. Before this can be done, however, a small time delay must occur. According to the data sheet for the ranging module, the INIT signal must stay high for a period of 100 ms and low for a period of 100 ms. When the echoes are received, the INIT

signals will not have been high for this length of time, so it is necessary to introduce a small time delay to ensure that this timing constraint is met.

The entire procedure described above is repeated for the other 2 firing sensor groups. Once this is completed, the next task is to calculate the distance to the object that each sonar sensor has detected. If the time of flight of the ultrasonic pulse for each sensor is known, it is a simple task of calculating the distance. The following formula is used:

$$d = 170.29t$$

This calculates the distance assuming a standard temperature of 15°C. In the equation, t represents time of flight. When all the distances have been calculated, they are transmitted over a serial communications link to the main central controller. The transmission protocol used is quite simple. It consists of a packet containing a Start character followed by the distance data for all 9 sensors and an End character. The Start character is used in order to ensure synchronization between the program running on the central controller and the start of a packet. The figure below shows a typical transmission packet used.

Start Character	Data	Data	Data	Data	Data	Data	Data	Data	Data	End Character

Figure 23 Typical Transmission Packet

3.8 Locomotion Board

Controlling DC motors can often be a complex and time-consuming task. Rather than have the main central controller on the robot perform this task, a separate locomotion board is used. This greatly reduces the burden placed on the main central controller, freeing it up to perform more useful and important tasks. The locomotion board uses two Hewlett Packard HCTL-1100 motion control ICs and an L293D driver IC. The advantage of using such a device as the HCTL-1100 is that it frees the host processor for other tasks by performing all

the complex and time-intensive functions of digital motion control. The chip is interfaced with a host processor (in this case, the central controller) which specifies the speed, acceleration and move distance for any new motion. Interfacing is done using the central controller's input/output expansion bus.

3.8.1 Controlling The Speed of a DC Motor

One of the easiest ways to control the speed of a motor is by PWM (Pulse Width Modulation). This is a technique whereby the power to the motors is rapidly switched on and off at a very high rate. The natural inductance and resistance of the motor acts as a low pass filter and makes the effective voltage seen by the motor to be the average value of the voltage over time. To control the speed of the motors, it is simply a task of varying the duty cycle of the PWM signal. The duty cycle refers to the fraction of time a signal is held high within a cycle. For example, with a 50% duty cycle, the signal represents a pure square wave while with a 20% duty cycle, the signal is only high for one fifth of the cycle. To implement a PWM control system, it is necessary to decide on the frequency of the PWM control signal. Normally, the frequency should be as high as possible, preferably above 10 KHz. If low frequencies are used, the motors can be quite audible. Also, there will be substantial current and torque variations on the output shaft of the motor. To ensure that a relatively steady current flows in the motors, the following equation can be used:

$2 * \Pi * F * L >> R$	L = Armature Inductance
	R = Armature Resistance
	F = Switching Frequency

For the motors used in the robot, the armature inductance is 1.27 mH, the armature resistance is 10.4 Ω. Therefore, if it's decided that the left hand side of the equation is to be 10 times larger that the right hand side, this results in the switching frequency being equal to13 KHz. This is the ideal frequency at which the PWM control system should work at. However, due to the finite speed of microcontrollers producing the PWM control signal, it

is not always possible to achieve this criteria. A frequency of 10 KHz should be perfectly adequate. The HCTL-1100 motion control processors are able to achieve this.

3.8.2 HCTL-1100 Motion Control IC

The HCTL-1100 is an extremely powerful and flexible motion control IC, which has been designed specifically for performing all the time-intensive functions of digital motion control. The device is capable of accepting high level commands from a host processor and carrying out all the low level functions needed to execute them. The advantage of using such a device is that it greatly frees up valuable processing power on the host processor. The HCTL-1100 is capable of carrying out a variety of different commands. These include the ability to be able to place the robot at any desired position, being able to move the robot at any desired speed and carrying out a trapezoidal profile move while specifying a speed and acceleration.

In use, the HCTL-1100 receives input commands from a host processor and position feedback from an incremental shaft encoder attached to the robot's wheel. An 8-bit bi-directional multiplexed address/data bus is used to interface the HCTL-1100 with the host processor. The device contains two input pins for position feedback from the incremental shaft encoder. A shaft encoder with two output signals 90° out of phase with each other is required. The HCTL-1100 also contains various logic lines to allow it to interface with the host processor. There is also an external clock input which can accept input frequencies between 100 KHz and 2 MHz. The output from the device is a pulse width modulated signal whose duty cycle is proportional to the magnitude of the motor command. There is also one other output signal that specifies the sign/direction of the pulse signal. Figure 24 below shows a block diagram of a control system built using the HCTL-1100.

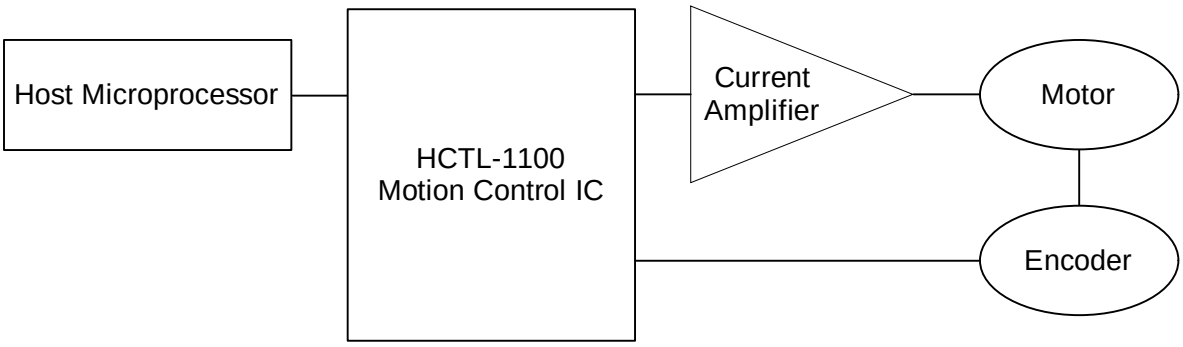


Figure 24 Control System

3.8.3 Incremental Shaft Encoder

To measure movement of the robot's wheel, an incremental shaft encoder is used. An incremental shaft encoder consists of an opaque disc containing a number of transparent slits. On one side of the disc is an infrared LED, on the other side is a infrared phototransistor. The light from the LED passes through the slit and into the phototransistor on the other side. The disc is attached to a shaft on the wheel. As the wheel rotates, the light passing through the slit will be periodically cut off. As each slit passes through the LED/phototransistor pair, an output pulse will be produced. By counting these pulses and knowing the dimensions of the disc and the robot's wheel, distances can be measured. By using only one LED/phototransistor pair, it is not possible to determine whether the direction of rotation is clockwise or counterclockwise. In order to make this distinction, it is necessary to have two LED/phototransistor pairs. The second pair is located so as to produce an output pulse which is 90° out of phase with the first. A phase detector can then be used to detect the leading versus lagging relationship of the two pulses and determine the direction of rotation. By using two LED/phototransistor pairs, the output can be decoded into quadrature counts. The number of pulses for each wheel revolution will then be four times the number of slits in the disk. This is shown in the figure below.

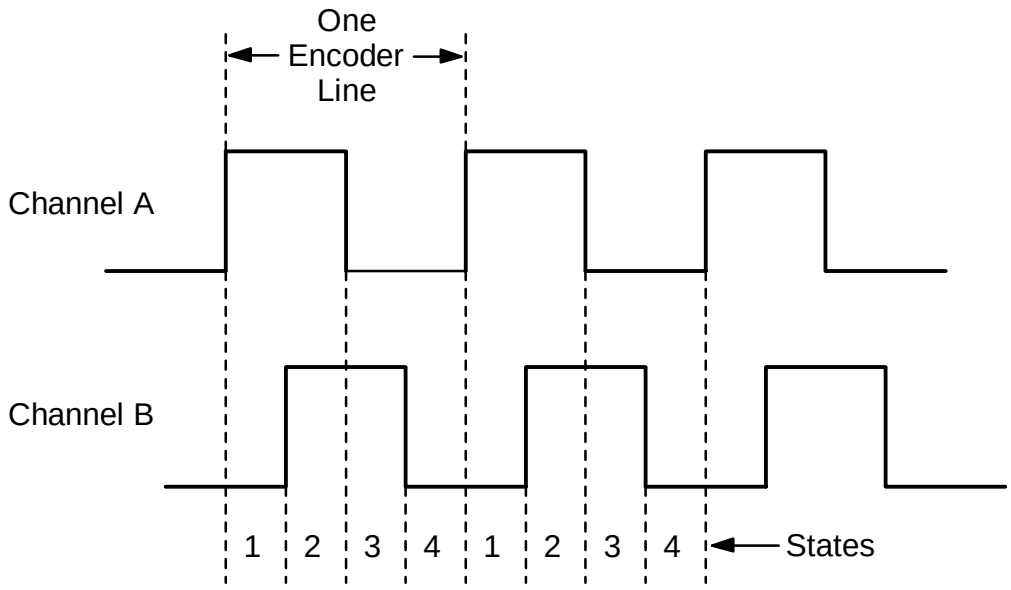


Figure 25 Quadrature Encoder Output Signals

The type of shaft encoder used on the robot is a Hewlett Packard HEDS-5600 two channel optical incremental shaft encoder. A block diagram of this is shown in figure 26. This is a commercially available encoder that contains the wheel and electronic circuitry in the one package. The wheel contains 500 slits. Since the output is decoded into quadrature counts, a total of 2000 pulses can be obtained for each wheel revolution. One pulse corresponds to a travel distance of 0.035mm. Although, in theory, this kind of control resolution allows very accurate distance changes to be measured, this is generally not the case in practice. This is due to mechanical inaccuracies such as friction, backlash and slippage which reduce the effective control resolution.

The HEDS-5600 contains two output pins. These are the outputs from each of the LED/phototransistor pairs. Each output is known as a channel. It is up to the HCTL-1100 to determine the phase relationship of the two channels to determine the direction of rotation and also to count the pulses for measuring movement of the robot's wheel.

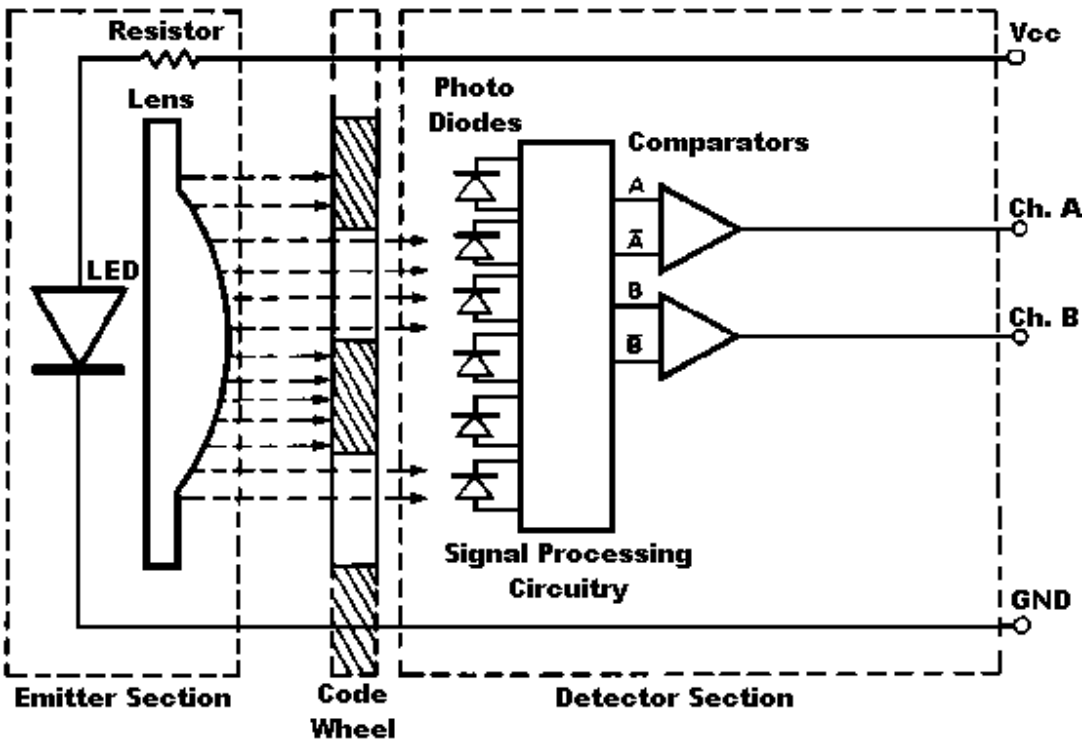


Figure 26 HEDS-5600 Shaft Encoder

3.8.4 Interfacing The HCTL-1100 With The Main Central Controller

The HCTL-1100 contains an 8-bit bi-directional multiplexed address/data bus. The lower 6 bits of this bus are multiplexed between address and data. The upper 2 bits are used for data only. There is also one control bus containing 4 pins that controls the I/O operation of the device. These pins include a read/write line, an address latch enable line, a chip select line and an output enable line. It is not possible to directly interface the HCTL-1100 with the main central controller's I/O expansion bus. This is due to a number of reasons. Instead, an 8255 programmable peripheral interface chip is used. This is a general purpose I/O component for interfacing peripheral equipment with a microprocessor bus. The 8255 contains three 8-bit ports which can act as either inputs or outputs. Two HCTL-1100 controllers are used on the locomotion board, one for each motor. Each address/data bus is connected to a separate port on the 8255. Since the control bus on the HCTL-1100 only contains 4 pins, both of them can be connected to the same 8-bit port on the 8255. One occupies the upper half, the other the lower half.

3.8.5 Address Decoding

Various chips on the locomotion board are mapped into the main central controller's I/O memory address space. By doing this, the devices can be accessed by reading or writing a value to or from a specific memory location. This requires the use of an address decoder. This is a device that is used to select a particular chip whenever a specific memory location is accessed. A 74HCT138 3 to 8 line decoder is used for the address decoding (see figure 27). This device contains 3 binary select inputs (A0, A1 and A2). These inputs determine which one of the normally high outputs will go low. The chip also contains two active low and one active high enable pins.

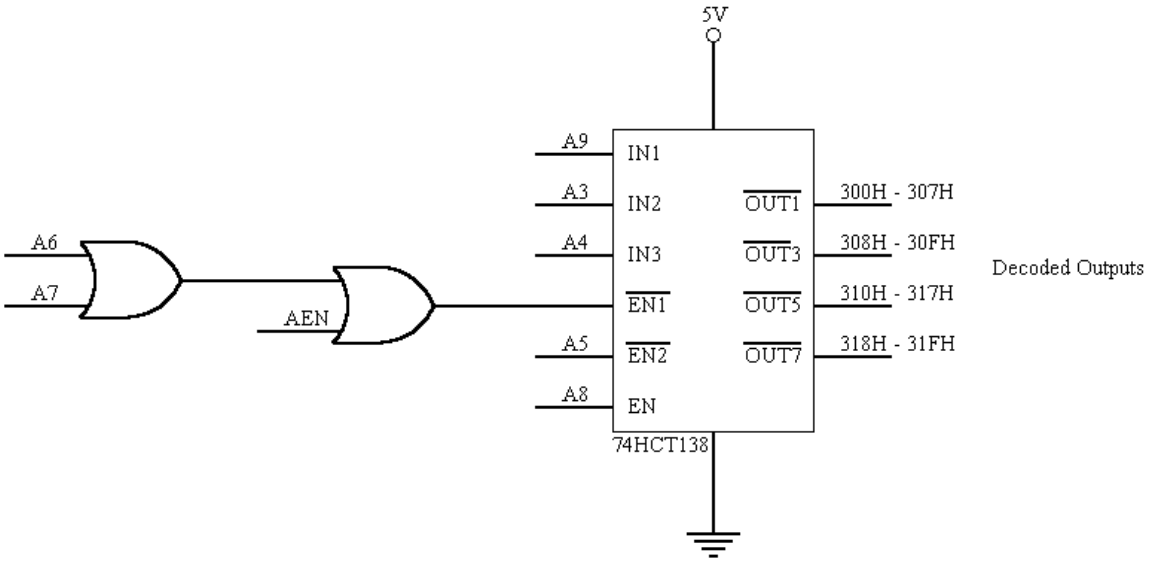


Figure 27 Address Decoding Circuitry

The input/output map for the PC consists of only 1024 addresses, as only the bottom ten address lines are used for input/output mapping. The lower half of the map is reserved for system hardware while the upper half is reserved for the expansion bus. Standard circuits such as serial and parallel ports do not count as system hardware since they fit onto the expansion bus. This means that the 512 addresses for the expansion bus are fairly crowded with few gaps. There is an area, however, starting from address 300H and ending at 31FH which is reserved for prototype cards. This area may be used for mapping external peripherals into the input/output map.

To keep the address decoding as simple as possible, this memory area is split into 4 segments. Each segment is 8 bytes in size. Address lines A3 to A9 and the AEN signal are used for the address decoding. Along with the 74HCT138, 2 OR gates are also used. AEN (address latch enable) is a signal which goes low during processor bus cycles. It is needed to distinguish between normal bus cycles and DMA (direct memory access) cycles. This must be decoded to the low state by the address decoder. Depending on which address or segment is written to, one of the 74HCT138's output pins goes low producing a chip select signal for the device it is connected to. A single device is mapped into each 8-byte segment. Some devices only require four or less memory locations in which to operate. Using 8 bytes for each device may seem wasteful. However, since extra memory space is not required, it is easier to implement the system in this manner.

3.8.6 Using The 8255

The 8255 must be programmed before it can be used. This will determine the manner in which it operates. The three ports (A, B and C) can be configured in a wide variety of ways. The way in which they are configured is determined by software running on the main central controller. There are three modes of operation for the 8255. These include basic input/output, strobed input/output and bi-directional bus. Only the first mode (basic input/output) is used by the robot. In this mode, each of the ports can act either as an input or an output. No handshaking is required, data is simply written to or read from a specified port. Two of the ports (A and B) act as 8-bit ports while port C can act either as one 8-bit port or two 4-bit ports. The outputs to the ports are latched. Therefore, any value written to the port will remain there until it is changed. The way the ports can be configured is very flexible allowing a total of 16 different input/output configurations.

The 8255 contains an 8-bit data bus and a 2-bit address bus. The data bus is connected to the main central controller's data bus. The control signals for the 8255 consist of a READ signal, a WRITE signal and a CHIP SELECT signal. The read and write signals are connected to the main central controller's IOR and IOW pins. The CHIP SELECT signal comes from the 74HCT138 3 to 8 line address decoder chip. The 8255 is mapped into the

main central controller's memory I/O address space at address 6000**H**. The 8255 contains two address pins (A0 and A1). These are connected to the central controller's A0 and A1 pins. The 8255 only occupies four addresses in memory. These include one address for each port and one for the control word. Thus, the device occupies memory locations from 300H to 303H. Once the chip has been properly configured, a port can be written to by simply writing to its specified address in memory. The table below shows the ports and the control word and their corresponding memory address locations.

300H	Port A
301H	Port B
302H	Port C
303H	Control Word

The 8255 has to be configured before it can be used. Configuring it sets up each of the ports as either an input or an output. It also specifies the mode that should be used for each port. To configure the device, a value specifying these various parameters is written to the control register. A block diagram of the control register is shown in figure 28.

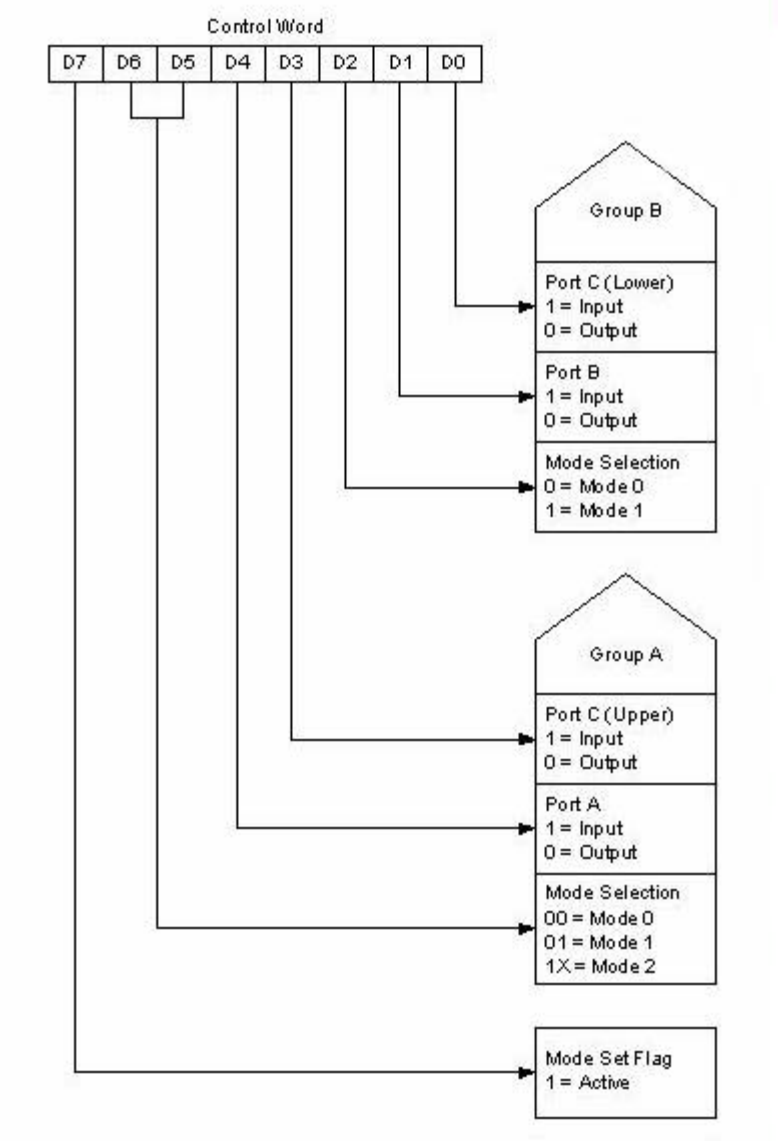


Figure 28 8255 Control Register

3.8.7 L293D Driver Chip

The PWM signal generated by the HCTL-1100 cannot be connected directly to the motors. There are a number of reasons for this. Firstly, the HCTL-1100 doesn't have the required current capability to drive them. Also, it can only give a maximum of 5V out. The motors require 12V to operate. Secondly, motors are inductive devices which could damage the HCTL-1100, although protection diodes could be used to overcome this problem. By using the L293D driver chip, all of these problems can be overcome. The L293D is a quad push-

pull driver capable of delivering currents up to 600 mA per channel. The advantage of the L293D over the L293 is that the L293D has output clamping diodes built in. This saves the addition of having to add them onto the circuit board.

The L293D contains four channels or drivers, each of which is controlled by a TTL compatible logic input. There is also an inhibit input for each pair of drivers along with a separate power supply for both the onboard logic and the motors. The maximum motor voltage is 36V which is far more than adequate. All of this is combined into a small 16-pin device. It is recommended that a heat sink be added to the chip to dissipate excessive heat being produced. It was found, however, that little heat was being produced for the motors used, and so it was decided to leave the heat sink off. The L293D can be used in a number of different ways. The way it is used here is to have each terminal of the motors connected to a separate output channel each. The motors can then be controlled by varying the TTL inputs for each channel. This can be summarized using the following table.

Inputs		**Function**
	C = High and D = Low	Turn Right
VINH = H	C = Low and D = High	Turn Left
	C = D	Fast Motor Stop
VINH	C = X and D = X	Free Running Motor Stop

Note: X = Don't Care

C = Input 2 (Pin 7)

D = Input 1 (Pin 2)

Figure 29 below shows a typical connection for a single motor connected to the L293D.

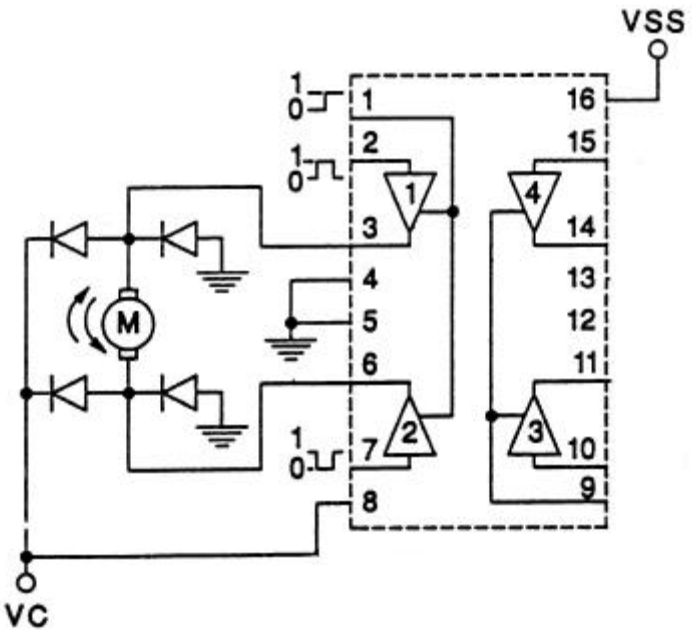


Figure 29 Connecting a single motor to the L293D

The L293D allows the motors to run from a separate power supply. This is an important feature, since the majority of motors require more than 5V in order to produce maximum power and speed. The robot uses two 12V DC motors. The voltage to operate them comes directly from the battery or power supply and is connected to the L293D.

3.8.8 Output From The HCTL-1100

The HCTL-1100 produces two output signals. One is a pulse width modulated signal whose duty cycle is proportional to the magnitude of the motor command. The other is a signal that specifies the sign/direction of the pulse signal. Together, these two signals specify the speed and direction of rotation for the motor. To interface these signals with the L293D, some extra logic circuitry is required. This consists of two AND gates and an inverter. This circuitry converts the PWM signal and the sign signal into the required input format for the L293D. Figure 30 shows this is implemented. The L293D has two TTL inputs to control each motor. If both of these inputs are the same, the motor will come to a fast stop. If the inputs are different, the motor will turn either left or right.

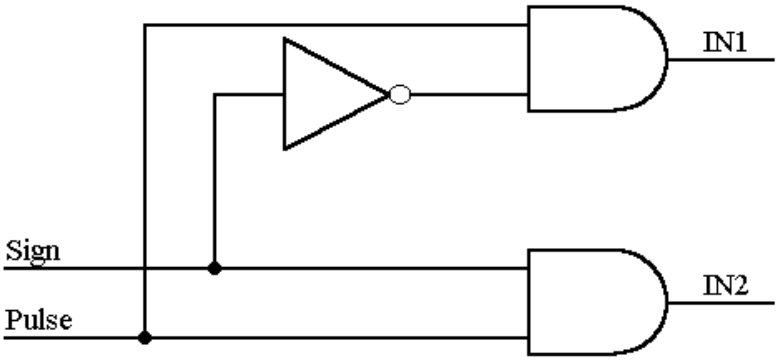


Figure 30 L293D Interface Circuitry

The truth table below shows the operation of the circuit.

PWM	SIGN	IN1	IN2
0	0	0	0
0	1	0	0
1	0	1	0
1	1	0	1

It can be seen from the table that whenever the PWM signal is 0, both inputs to the L293D are also 0. The polarity of the sign signal has no effect. In this state, the motor is not been driven and will come to a fast stop. Whenever, the PWM signal is 1, the inputs to the L293D will have opposite values. This will drive the motor either left or right. The direction is determined by the polarity of the sign signal.

3.8.9 Writing To And Reading From The HCTL-1100

There are 3 different timing configurations that can be used to interface the HCTL-1100 with a host processor. The mode that is used here is the non-overlapped mode. This was chosen because it is the easiest to implement.

To write a value to the HCTL-1100, the following procedure is used. First the address of the register that is to be written to is placed on the HCTL-1100's address bus. ALE is then brought low. After a small time delay, ALE is brought high again, followed by CS low and

R/W low. The value to be written to the register is then placed on the HCTL-1100's data bus. Following this, CS and R/W are brought high again.

To read a value from the HCTL-1100, the following procedure is used. First the address of the register that is to be read is placed on the HCTL-1100's address bus. ALE is then brought low. After a small time delay, ALE is brought high again, followed by CS low. CS is then brought high and OE brought low. The value contained in the register is then placed on the data bus by the HCTL-1100. This can then be read by the host processor.

The address/data and control buses on the HCTL-1100 are connected to the 8255 IC. Therefore, in order to write a value to the HCTL-1100, the value has to be written to the 8255. Similarly, to read a value from the HCTL-1100, the value has to be read from the 8255.

The following is a piece of C code to write a value to some specified register in the HCTL-1100:

```
void Write_To_Register(unsigned int motor, unsigned char reg_address,
                                    unsigned char value)
{
  unsigned int wait;
  outportb(ppi_control, 0x80);

  if (motor == LEFT)
  {
     outportb(ppi_portc, reg_address);
     outportb(ppi_porta, 0xFD);   // ALE low
     for (wait = 0; wait < con; wait++) ;
     outportb(ppi_porta, 0xFA);   // ALE high, CS low, R/W low
     outportb(ppi_portc, value);
     outportb(ppi_porta, 0xFE);   // CS high
     outportb(ppi_porta, 0xFF);   // R/W high
  }
```

```
  else if (motor == RIGHT)
  {
     outportb(ppi_portb, reg_address);
     outportb(ppi_porta, 0xDF);   // ALE low
     for (wait = 0; wait < con; wait++) ;
     outportb(ppi_porta, 0xAF);   // ALE high, CS low, R/W low
     outportb(ppi_portb, value);
     outportb(ppi_porta, 0xEF);   // CS high
     outportb(ppi_porta, 0xFF);   // R/W high
  }
}
```

The following code reads in a value from some specified register in the HCTL-1100:

```
unsigned char Read_Register(unsigned int motor, unsigned char reg_address)
{
  unsigned char reg_value;
  unsigned int wait;

  outportb(ppi_control, 0x80);

  if (motor == LEFT)
  {
     outportb(ppi_portc, reg_address);
     outportb(ppi_porta, 0xFD);   // ALE low
     for (wait = 0; wait < con; wait++) ;
     outportb(ppi_porta, 0xFB);   // ALE high, CS low
     outportb(ppi_porta, 0xFF);   // CS high
     outportb(ppi_porta, 0xF7);   // OE low
     outportb(ppi_control, 0x89);
     reg_value = inportb(ppi_portc);
```

```
    outportb(ppi_porta, 0xFF);   // OE high
  }

  else if (motor == RIGHT)
  {
    outportb(ppi_portb, reg_address);
    outportb(ppi_porta, 0xDF);   // ALE low
    for (wait = 0; wait < con; wait++) ;
    outportb(ppi_porta, 0xBF);   // ALE high, CS low
    outportb(ppi_porta, 0xFF);   // CS high
    outportb(ppi_porta, 0x7F);   // OE low
    outportb(ppi_control, 0x82);
    reg_value = inportb(ppi_portb);
    outportb(ppi_porta, 0xFF);   // OE high
  }

  return reg_value;
}
```

3.8.10 Using The HCTL-1100

In use, the HCTL-1100 receives input commands from the main central controller and position feedback from the incremental shaft encoder attached to the robot's wheel. Internally, the HCTL-1100 is controlled by a bank of 64 8-bit registers, 35 of which are user accessible. These registers contain command and configuration information necessary to properly run the device. A brief description of the most important registers that are used is shown below.

Flag Register: This register consists of a group of flags F0 through F5. Each flag can be individually set or cleared. The flags are used to specify which control mode is to be used and also to specify various other parameters.

Program Counter Register: By writing a value to this register, various functions are executed. These include performing a software reset and executing one of the pre-programmed control modes.

Status Register: This register is used to indicate the status of the HCTL-1100. The lower 4 bits of the register can be written to in order to configure various parts of the HCTL-1100.

PWM Motor Command Register: The 2's-complement value of this register represents the duty cycle and polarity of the PWM command. The PWM signal at the pulse pin has a frequency equal to the external clock divided by 100. The duty cycle is resolved into the 100 clocks. For the 1 MHz clock used on the board, this means that the PWM frequency is equal to 10 KHz.

Actual Position Register: During control moves, the HCTL-1100 keeps track of the position of the motor. This position is stored as a 24-bit number and is located across 3 8-bit registers. By reading this register, the current position of the robot can be obtained. It can also be set to zero or some other value before a new command is issued.

Sample Timer Register: The HCTL-1100 is a digitally sampled data system. While information from the host processor is accepted asynchronously with respect to the control functions, the motor command is computed on a discrete sample time basis. The sample timer is programmable by writing a value to the sample timer register. The sampling period is equal to the following:

$$t = 16(T+1)\left(\frac{1}{frequency\ of\ \ the\ external\ clock}\right)$$

Where T = contents of the sample time register.
With a 1 MHz clock, the sample time for the HCTL-1100 can vary from 128 μS to 4096 μS.

The HCTL-1100 has 4 control modes which are available to the user. These include position control, proportional velocity control, trapezoidal profile control and integral velocity control. The following section describes the function of each control mode and how it is programmed by the user. When the HCTL-1100 is reset, it enters into an initialization/idle mode. In this mode, no commands are being executed by the HCTL-1100 and it is simply waiting for input from the user.

In order to specify which control mode is to be executed, flags F0, F3 or F5 in the flag register must be set. Each flag represents a different control mode. Only one flag can be set at a time. If no flags are set, then the position control mode will be executed. After one of these flags is set, the control mode is entered from the initialization/idle mode by writing a value of 03H to the program counter register.

3.8.10.1 Position Control Mode

To enter this mode, flags F0, F3 and F5 in the flags register must be cleared and a value of 03H written to the program counter register. This mode is used to perform point to point moves with no velocity profiling. The user specifies a 24-bit position command which the controller compares to the 24-bit actual position. After calculating the position error, a motor command is output. The controller will remain locked at a destination until a new command is issued. The actual and command position data is stored as 24-bit 2's-complement data. Position is measured in encoder quadrature counts. For the encoders used on the robot, this means a total of 2000 counts per wheel revolution. This allows very fine position moves to be executed. The following shows an example on how to use this mode:

Example Code to Program Position Moves

```
{ Begin }
    Hard Reset  { HCTL-1100 goes into INIT/IDLE mode }
    Initialize Filter, Timer, Command Position Registers
    Write 03H to Program Counter Register
        { HCTL-1100 is now in Position Mode }
    Write Desired Command Position to Command Position Registers
        { Controller moves to new position }
    Continue writing in new Command Positions
{ End }
```

3.8.10.2 Integral Velocity Mode

To enter this mode, flags F0 and F3 in the flags register must be cleared and flag F5 set to begin the move. Also, a value of 03H must be written to the program counter register. This mode is used to perform continuous velocity profiling. The user specifies a velocity and acceleration and the HCTL-1100 will accelerate up to the desired velocity and maintain it until such time that a new command is issued. Figure 31 shows the capability of the control algorithm. The velocity specified by the user is an 8-bit 2's-complement value. Its units are

quandrature counts/sample time. To convert from rpm to quandrature counts/sample time, the following equation can be used:

$$Vq = (Vr)(N)(t)(0.01667 / rpm - \text{sec})$$

Where: Vq = velocity in quadrature counts/sample time
Vr = velocity in rpm
N = 4 times the number of slots in the codewheel (i.e. quanrature counts)
t = sample time in seconds

The acceleration specified by the user is a 16-bit value. Its units are quadrature counts/sample time squared. To convert from rpm/sec to quadrature counts/sample time squared, the following equation can be used:

$$Aq = (Ar)(N)(t^2)(0.01667 / rpm - \text{sec})$$

Where: Aq = acceleration in quadrature counts/sample time squared
Ar = acceleration in rpm/sec
N = 4 times the number of slots in the codewheel (i.e. quanrature counts)
t = sample time in seconds

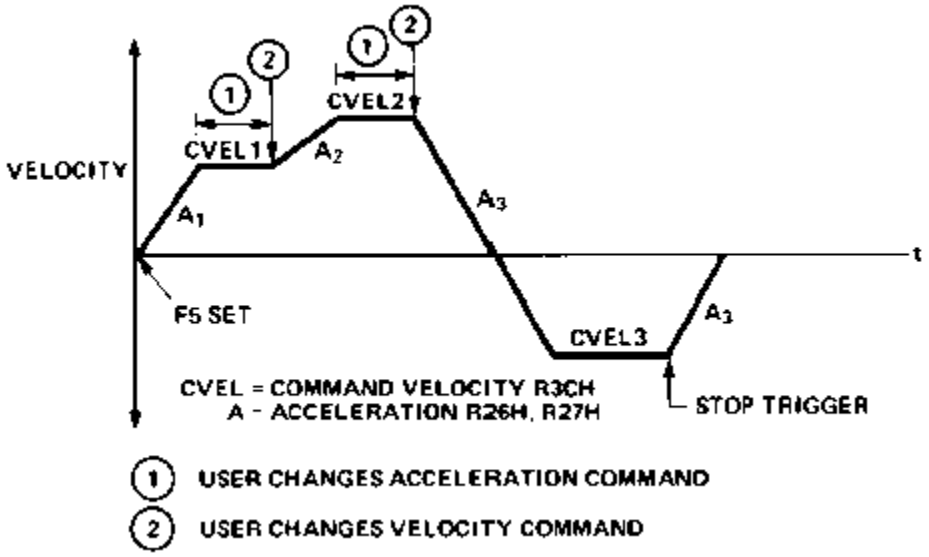


Figure 31 Integral Velocity Mode

The following shows an example on how to use this mode:

Example Code for Programming Integral Velocity Mode

{ Begin }

Hard Reset { HCTL-1100 goes into INIT/IDLE mode }

Initialize Filter, Timer, Command Position Registers

Write 03H to Program Counter Register

{ HCTL-1100 is now in Position Mode }

Write Desired Acceleration (if needed)

Write Desired Maximum Velocity (if needed)

Set Flag F5 in the Flags Register { Integral Velocity Move Begins }

{ System Ramps to Maximum Velocity}

Continue writing in new Accelerations and Velocities

{ End }

3.8.10.3 Trapezoidal Profile Mode

To enter this mode, flags F3 and F5 in the flags register must be cleared and flag F0 set to begin the move. Also, a value of 03H must be written to the program counter register. This mode is used to perform point-to-point position moves while profiling the velocity trajectory to a trapezoid or triangle. The user specifies the desired final position, acceleration and maximum velocity and the controller will produce the necessary profile to carry out the task. If maximum velocity is reached before arriving at the halfway point, the profile generated will be trapezoidal, otherwise it will be triangular. Figure 32 shows the possible trajectories when using this mode. The final position is specified by the user as a 24-bit 2's-complement value. The acceleration is specified as a 16-bit value with units of quadrature counts/per sample time squared. The maximum velocity is a 7-bit scalar value and has the units of quadrature counts/sample time.

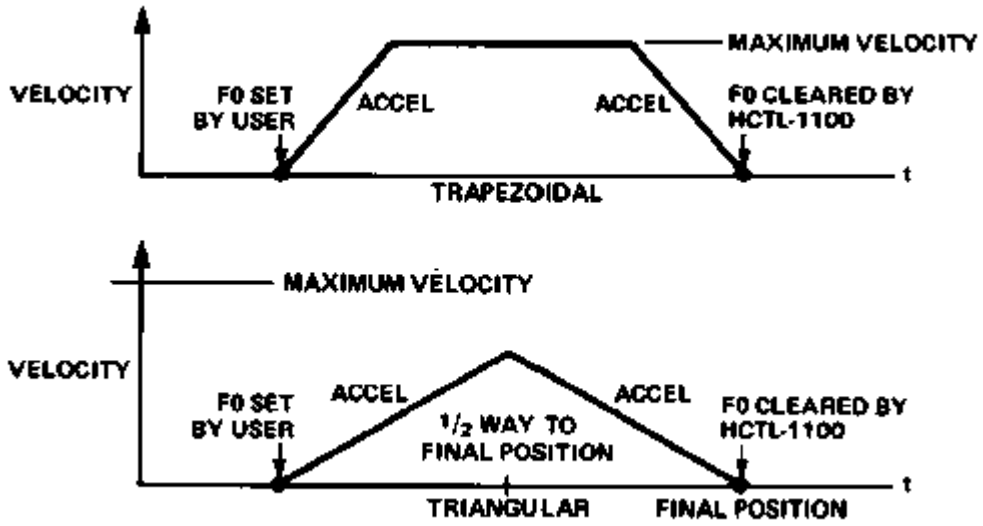


Figure 32 Trapezoidal Profile Mode

3.9 Light Detection Circuitry

A light seeking ability has been implemented on the robot. This allows the robot to seek or follow a light source. To achieve this, the robot uses two light dependant resistors (LDRs). These are devices whose resistance varies with the amount of light falling on them. In complete darkness, the resistance of an LDR is in the order of a few megohms. In bright light, the resistance falls dramatically to a few kilohms. By measuring this resistance, the intensity of the light can be determined. To measure the resistance, the LDR is connected in series with another resistor forming a voltage divider. As the resistance of the LDR changes, a voltage will be dropped across it.

This voltage is read by an analogue to digital converter (ADC). This is a device that converts an analogue input voltage into a digital value. This can then be read by a microprocessor connected to the ADC. There are numerous different types of ADCs available. The type that has been opted for here is a National Semiconductor ADC0808. This was chosen because it allows more than one input voltage to be measured.

3.9.1 ADC0808

The ADC0808 is an 8 input 8-bit ADC capable of measuring input voltages between 0 and 5V with an 8-bit resolution. This allows 256 different values to be measured. Onboard is an 8-channel multiplexer which selects one of 8 possible analogue input voltages. A 3-bit address port is used to select which input to use. There are two voltage reference inputs on the chip. One is connected to 0V and the other to 5V. This allows voltages between 0 and 5V to be measured. The control signals for the ADC0808 consist of a START signal, an ALE (Address Latch Enable) signal, an OE (Output Enable) signal and an EOC (End Of Conversion) signal. Since the ADC0808 is a digitally sampled system, it required a clock to operate. This can have a value between 10 KHz and 1.28 MHz. A 1 MHz clock is used here which comes from the same oscillator used for the HCTL-1100. The ADC0808 is interfaced directly with the main central controller using some additional logic circuitry. This consists of two NOR gates. These get their inputs from the main central controller's IOR and IOW lines and an external chip select signal. The chip select signal comes from the 74HCT138 address decoder. Figure 33 shows how the system is implemented. The device is mapped into the main central controller's memory I/O address space at address 310H. The 3 address lines on the ADC are connected to the main central controller's A0, A1 and A2 address lines. The ADC therefore occupies memory address locations from 310H to 317H. The table below shows each of the analogue inputs and their corresponding memory address locations.

310H	$V_{IN}1$
311H	$V_{IN}2$
312H	$V_{IN}3$
313H	$V_{IN}4$
314H	$V_{IN}5$
315H	$V_{IN}6$
316H	$V_{IN}7$
317H	$V_{IN}8$

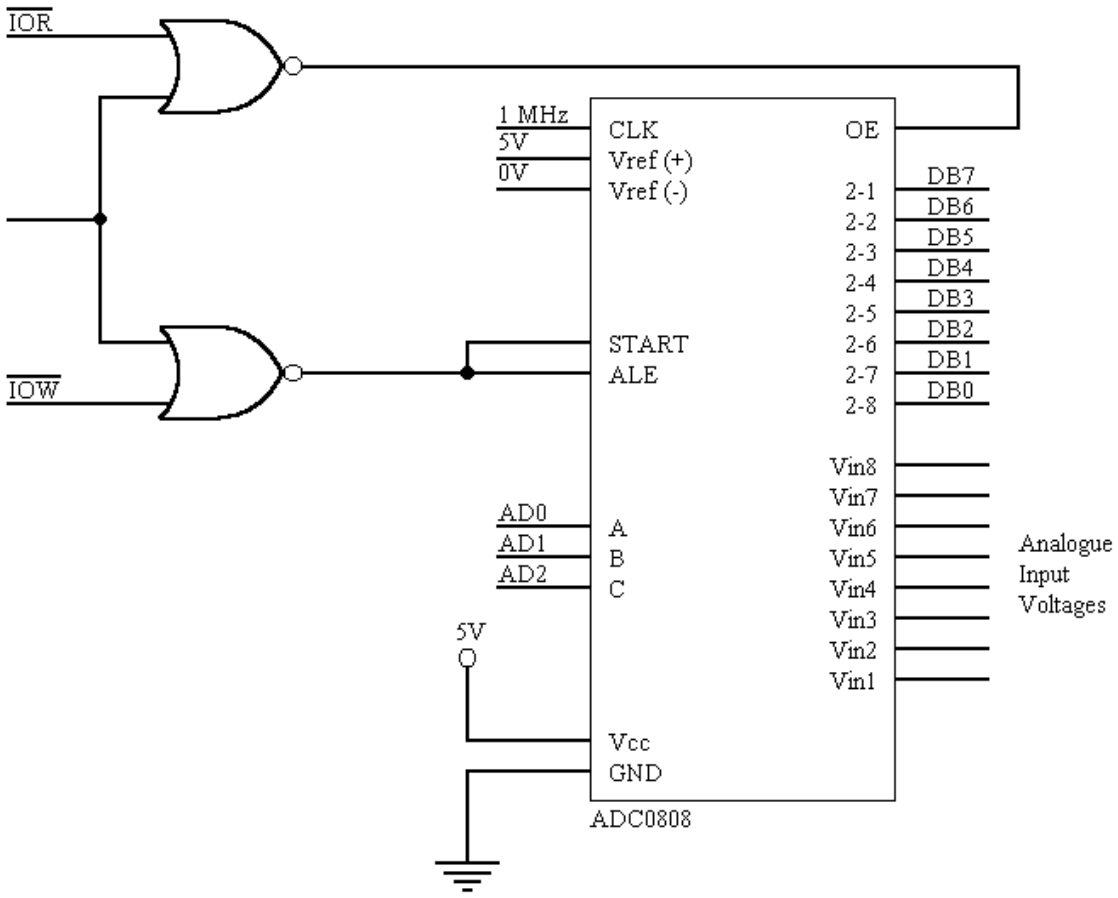


Figure 33 ADC Interface Circuitry

3.9.2 Using the ADC0808

To use the ADC, the address of the analogue input to be measured must first be written to the address port on the chip. Both START and ALE are then triggered. ALE is used to latch the 3 address inputs into internal registers. START is used to initiate a new conversion cycle. Using a 1 MHz clock, the conversion takes approximately 64 uS. After a new conversion begins, the EOC pin goes low to indicate a new conversion cycle is in progress. When the conversion is complete, EOC goes high again. This pin could be used to trigger an interrupt on a microprocessor to let it know when a new conversion is complete. To read in the digital value of the measured voltage, the OE pin is brought high. This places the digital value onto the output port of the ADC. This can then be read by the microprocessor.

As an example, to measure the voltage at input 1 of the ADC, some dummy value would be written to address 6000H. Since the address is used to generate a chip select and for specifying which input to use on the ADC, the value written is irrelevant. By writing to this address, input 1 on the ADC is selected and the ALE and START pins are triggered. This would latch the address and cause a new conversion cycle to begin. Instead of monitoring the EOC pin to determine when the conversion is complete, a small time delay is introduced. This is a simpler solution and requires less hardware. After a 1 mS time delay, the main central controller reads in a value from address 6000H. This causes the OE pin on the ADC to be triggered causing it to place the digital value onto its output port. This is then read in by the main central controller.

The two reference inputs on the ADC0808 are connected to 0V and 5V respectively. This allows the device to measure input voltages between 0 and 5V. With an 8-bit ADC, 256 different voltages can be measured. This means that each bit represents a voltage of 19.53 mV. If the main central controller reads in a value of 178, this would represent a voltage of 3.47V.

3.10 Software Development

The main central controller used on the robot is an embedded 586 CPU module. The advantage of using such a device is that all software development can be carried out using traditional PC based programming languages such as Borland C++. Software for the robot is first developed on a desktop PC. Once this is compiled, it is downloaded to the controller over a serial communications link and stored on the solid state hard disk. From here, it can be executed like any normal program.

The control architecture used on the robot is a modified form subsumption. This is a behaviour-based architecture which tightly couples sensor input to actuator output in a reactive way. A number of behaviours, each of which performs a specific task, all run concurrently. In order to implement this in software, there is a need to have a multitasking ability built in. This allows each of the tasks to run separately. Unfortunately, most microprocessors are sequential machines and do not possess this ability. To overcome it, a

piece of code which simulates this multi-tasking ability can be used instead. There are a number of third-party products which allow standard C programs to implement multitasking. The one which has been opted for here is the MicroC/OS-II real time kernal. To use this kernal, the program is divided into a number of separate logical sections called tasks. These represent both the robot's behaviours and other functions the software has to perform, such as gathering sonar data from the serial port. Each task appears to run simultaneously through the use of preemptive multitasking. The system is driven by a system timer tick that is generated by an interrupt from the central controller's 8254 counter/timer chip. This allocates a certain amount of time for each task to go about its work.

3.11 Summary

This chapter has described both the physical and hardware design of the robot. The electronic control hardware has been designed in the form of a distributed control architecture, in which a number of separate modules are used to perform complex tasks. There are three distinct modules in the system. These consist of a locomotion board, a sonar sensor board and a central controller. Further modules will be added as required by extra sensory capabilities and implemented behaviours to improve the competence of the robot. The advantage of distributing the workload to different modules is that it frees the main controller from performing repetitive albeit complex time-consuming tasks such as the servo control of individual drives. To allow the robot to operate in tight locations, a cylindrical design using a differential drive system was opted for. The differential drive system will allow the robot to turn on the spot, preventing it from getting caught in tight corners.

4. CONTROL ARCHITECTURE

This chapter describes the control architecture used on the robot. In addition a set of behaviours which have been successfully implemented are described in detail. These behaviours allow the robot to explore its environment and generate a map. This map can later be used to travel to specific locations specified by a user. A complete review of topological maps is given to support the mapping technique used.

4.1 Overview

The two mobile robot control architectures that have dominated the robotics scene are the hierarchical architecture and the behaviour-based architecture. The major advantage of behaviour based architectures is that they can respond very rapidly to environmental changes. This makes them very robust in unstructured and dynamic environments. They are also very flexible and require less demanding computational requirements than their equivalent hierarchical architecture. It is for these reasons that a behaviour-based architecture has been adopted here. One of the problems with behaviour-based systems, however, is that they can suffer from system modularity, state representation and the integration of world models. Due to the highly distributed nature of these systems, representation and sharing of system states and knowledge between the behaviours is inconvenient. To overcome these shortcomings with the behaviour-based architecture, a new type of architecture is introduced. This is known as a behaviour-based blackboard architecture. This introduces the concept of a blackboard, which acts as a central data repository where behaviours can deposit and extract information to help them go about their tasks.

4.2 Behaviour-Based Blackboard Architecture

The control architecture implemented on the robot is based on the standard subsumption architecture. This allows the robot to respond very rapidly to environmental changes.

Sensor input is tightly coupled with actuator output in a reactive way allowing the robot to respond directly to environmental cues. To allow interaction and communication between the behaviours to occur, a central data repository known as a blackboard is used. Behaviours are allowed to store and retrieve information from the blackboard to help them execute their tasks. The introduction of the blackboard in this architecture helps alleviate one of the shortcomings of standard behaviour based systems - the sharing of knowledge and system states. In subsumption, the interdependence of behaviours can lead to systems with little flexibility and modularity. The behaviours often need to be able to access and examine the internal state of other behaviours. Adding new behaviours may lead to change of some existing ones. By incorporating a blackboard into the system, new behaviours can easily be integrated without having to adapt the other behaviours.

4.3 Implementation of Simple Low-Level Behaviours

Having built the robot and developed the architectural framework for implementing the robot's control system, a basic set of reflexive behaviours were designed. These initial behaviours were extremely simple and their sole purpose was to demonstrate that the robot could operate reactively based on sensor stimuli. These three behaviours consisted of a cruise behaviour, an obstacle avoidance behaviour and a light following behaviour. In designing a behaviour-based control system, low level behaviours are added first with successive behaviours being added incrementally to enhance the functionality of the robot. Obstacle avoidance is generally the most important ability a robot must have, so this behaviour is added first. This behaviour is the lowest level behaviour since it is essential that the robot avoids obstacles at all costs irrespective of what other tasks it may be performing. In implementing these behaviours, the concept of the blackboard is not introduced. The behaviours are relatively simple and do not require any sharing of knowledge between them. The following sections describe each of these three behaviours in turn and how they interact with each other through the robot's environment.

4.3.1 Obstacle Avoidance Behaviour

To operate successfully in our everyday environments, mobile robots must be capable of dealing with all the uncertainty and variation that exists. To equip them with this capability, they must be provided with a means to detect and avoid obstacles. This is a very important feature of any mobile robot. Colliding with an obstacle could damage or hinder the operation of the robot. It is therefore imperative that it is provided with the best means to prevent a collision from occurring. To endow the robot with this capability, an obstacle avoidance behaviour has been implemented. This behaviour gets its input from a ring of sonar sensors located around the top of the robot. The output from the behaviour is a set of commands for the motors that specify their speed and direction of rotation (see figure 34).

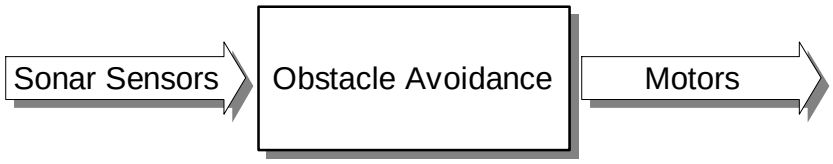


Figure 34 Obstacle Avoidance Behaviour

Each of the sonar sensors return the distance to the object directly in front of them. The behaviour works by assigning weights to each of the sensors and also a condition value (see figure 35). The weights specify the change in speed that should occur in the robot's motors if the sensor detects an obstacle located closer than the condition value. For example, if the sensor facing 45° detects an obstacle located less than 18 inches away, a weight of -3 will be assigned to the left motor and a weight of 2 to the right motor. These values will then be added to the current speed of each motor. This procedure is repeated for each of the sensors in turn resulting in a final speed value for both motors. The weights that have been assigned to each motor have been chosen both experimentally through trial and error and through common sense by determining the relative threat an object poses to the robot. This can be seen in the case of the forward looking sonar sensor. A weight of -9 is assigned to the right motor and a weight of 0 to the left motor. If this sensor detects an obstacle located closer than 20 inches away, the speed of the right motor will be drastically reduced, turning the robot quickly away from the obstacle.. This is important since an obstacle in front of the robot poses more of a threat than any other.

The following equations are used to calculate the change in speed for each motor.

Change in Left Motor Speed = $\sum_{s=1}^{5} Left\_Motor\_Weight(s)$

Change in Right Motor Speed = $\sum_{s=1}^{5} Right\_Motor\_Weight(s)$

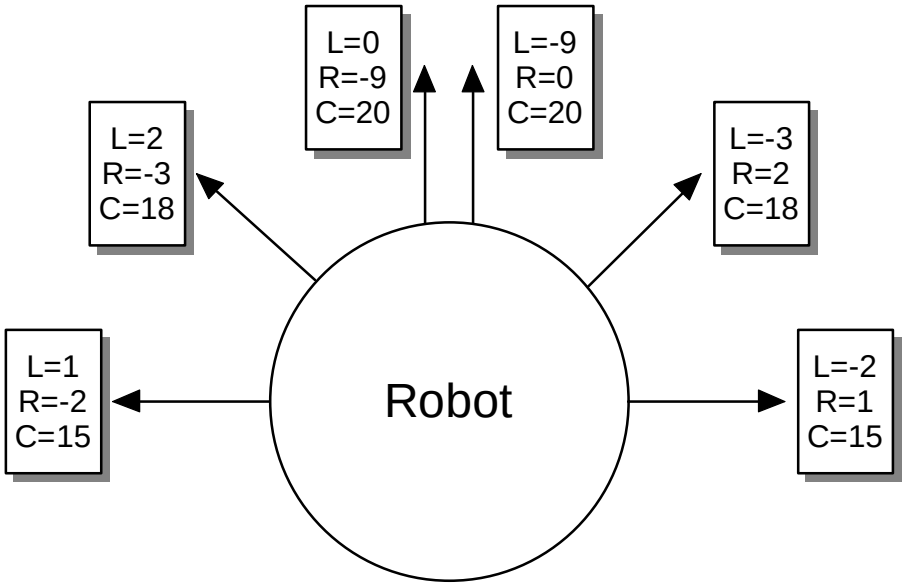


Figure 35 Motor Weights and Condition Values

4.3.2 Cruise Behaviour

This is an extremely simple behaviour whose sole purpose is to move the robot straight ahead. Since the behaviour does not react to any kind of sensor stimuli, there are no inputs to the behaviour. The output is a set of commands for the motors that specify the speed at which to travel (see figure 36). It may seem more logical to endow this behaviour with the ability to steer in a random direction. This is not necessary, however, since in conjunction with the obstacle avoidance behaviour, the robot will naturally turn in the presence of

obstacles. Together, the obstacle avoidance behaviour and the cruise behaviour equip the robot with a very basic ability - to wander around without colliding with objects.

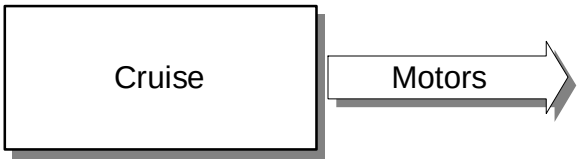


Figure 36 Cruise Behaviour

4.3.3 Light Following Behaviour

This behaviour allows the robot to seek or follow a light source. To achieve this, the robot uses two light dependant resistors (LDRs). These are located on top of the robot pointing forwards. They can determine the direction of a light source relative to the direction in which the robot is facing. By using some simple signal conditioning circuitry, voltages are produced proportional to the intensity of light falling on the LDRs. These voltages are measured using an ADC (analogue to digital converter). The difference between the two voltages is measured and if this is above some preset threshold value, the robot will steer in the direction of the brightest LDR. The input to this behaviour is the voltage from each LDR. The output is a set of commands for the motors that specify their speed and direction of rotation (see figure 37).

Figure 37 Light Following Behaviour

The two reference inputs used on the ADC are connected to 0V and 5V respectively. This allows the device to measure input voltages between 0 and 5V. With an 8-bit ADC, 256 different voltages can be measured. This means that each bit represents a voltage of 19.53 mV. If the main central controller reads in a value of 178, for example, this would represent

a voltage of 3.47V. The preset threshold value chosen is 35. This represents a voltage difference of 683mV (19.53mV * 35). If the difference in the intensity of the light between the two LDRs is greater than this value, the behaviour will attempt to move the robot in the direction of the brighter LDR. The threshold value chosen depends on the type of LDR used. Different LDRs react different to the same amount of light. A certain amount of experimentation is required to come up with a suitable value.

4.3.4 Arbitration Function

In behaviour based architectures, a number of behaviours run concurrently. Many of these may try to drive the same actuator at the same time. This is clearly evident in the behaviours shown here. The three behaviours *obstacle avoidance*, *cruise* and *light following* can all send commands to the motors simultaneously. The problem with this is that a conflict can occur. Unless this is taken care of, erratic behaviour may result. To overcome it, an arbitration function has to be implemented. The arbitration function has to select a single behavioural response from a multitude of possible ones. There are a number of ways in which this can be done. In subsumption, a fixed priority arbitration scheme is used. Each of the behaviours is layered according to their relative importance. Figure 38 shows the layering for the three behaviours described here. As can be seen, the obstacle avoidance behaviour is able to subsume the output of both the light following behaviour and the cruise behaviour. Similarly, the light following behaviour can subsume the output of the cruise behaviour.

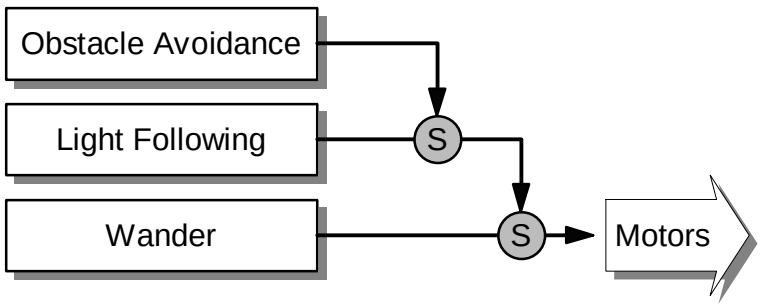


Figure 38 Layering of Behaviours by Importance

The arbitration function selects a single behavioural response to send to the motors. Operation of this function is quite simple. It continuously monitor the output from each of the behaviours and using a fixed priority hierarchy, selects one single output to send to the motors.

4.4 Implementation of Mapping and Navigation Behaviours

The implementation of the *obstacle avoidance*, *cruise* and *light following* behaviours illustrated the robot's capability to react directly to sensor stimuli, and perform seemingly intelligent tasks by interacting with each other through the robot's environment. Together, these three behaviours helped develop the framework for implementing the robot's other behaviours. The next set of behaviours to be developed endow the robot with the ability to explore and map its environment. The robot can build up a topological map of the environment which it can subsequently use to navigate to specific locations specified by the user. The robot explores the environment by following walls or straight edges. To help it detect landmarks for both building and using the map, a set of landmark detection behaviours have been implemented. These can detect both concave and convex corners and doors. Behaviours have also been developed which allow the robot to search for and lock onto a wall and to perform localisation. The behaviours developed in this section all interact and communicate with each other using the blackboard model described earlier. This allows sharing of system state and knowledge between the behaviours. Figure 39 shows the blackboard and how each of the behaviours interacts with it. An arrow leading from a behaviour to the blackboard indicates that the behaviour places information onto the blackboard. Similarly, an arrow leading from the blackboard to a behaviour indicates that the behaviour retrieves information from the blackboard. The following sections give an in-depth discussion of each behaviour and how they interact through the blackboard.

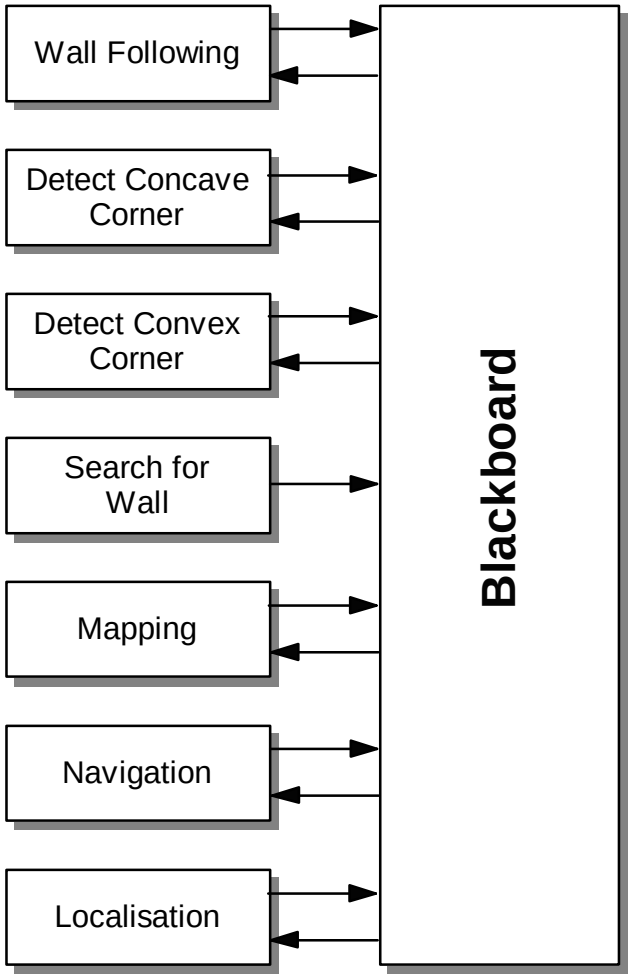


Figure 39 Behaviour-Based Blackboard Architecture

4.4.1 Edge Following Behaviour

The exploration and navigation strategy that the robot uses is based on following the edge or outline of features in the environment. To equip the robot with the ability to do this, an edge following behaviour has been implemented. This causes the robot to travel parallel to an edge at a certain pre-set distance. It is very important that this behaviour be robust in following the edge as accurately as possible, since this is important for the correct operation of the landmark detection behaviours. The inputs to this behaviour are the range readings from the sonar sensors located around the robot and a message from the blackboard telling the behaviour whether it is following a left hand or a right hand edge. The output from the behaviour is a set of commands for the motors that specify their speed and direction of

rotation (see figure 40). The message on the blackboard is originally generated by the *Search For Edge* behaviour, once it has successfully found and locked onto an edge. This behaviour will be explained later.

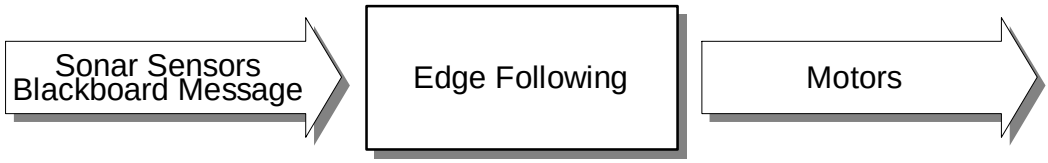


Figure 40 Edge Following Behaviour

For the purpose of this explanation, the procedure for following a left hand edge will be given. To follow a right hand edge, an identical procedure is used, apart from the sonar sensors employed. To follow the left hand edge, two sonar sensors are used - one facing -90° and another facing -45° (see figure 41). The distance returned from the -90° sensor is refererred to as the *side distance* and the distance returned from the -45° sensor is referred to as the *diagonal distance*.

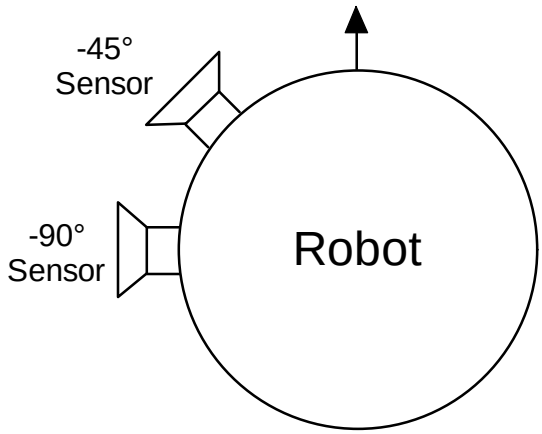


Figure 41 Sonar Sensors used for Edge Following

A fairly simple procedure is used to follow an edge. The behaviour continuously monitors the range readings from the two sonar sensor and turns the robot either towards or away from the edge depending on the values read. The *side distance* indicates how close the robot is to the edge whereas the *diagonal distance* indicates whether the robot is veering in

towards or away from the edge. The following strategy was eventually developed following a lot of experimentation through trial and error.

- If the *side distance* is less than 12 inches, the robot is too close to the edge and should veer away from it.
- If the *side distance* is greater than 11 inches and less than 17 inches, the robot is too far from the edge and should veer towards it.
- If the *side distance* is greater than 11 inches and the *diagonal distance* is less than 18 inches, the robot is too far from the edge but is heading in a direction towards it. It should therefore veer away somewhat to re-establish a correct parallel track.
- If the *side distance* is less than 12 inches and the *diagonal distance* is greater than 17 inches, the robot is close to the edge but is heading in a direction away from it. It should therefore veer towards it somewhat to re-establish a correct parallel track.

For the above strategy to be effective, the robot must initially be parallel to an edge before the procedure can begin. The edge following behaviour itself does not ensure that this requirement is met. Instead, this is left to the *Search For Edge* behaviour. This behaviour will lock onto and ensure that the robot is parallel to the edge. It will then place a message on the blackboard indicating whether a lock onto a left hand or a right hand edge has been established. For the *Edge Following* behaviour, the decision about whether to follow a left or a right hand edge is determined by this message.

4.4.2 Concave Corner Behaviour

The most common features the robot encounters in its environment are straight edges and concave and convex corners. The *Edge Following* behaviour endows the robot with the ability to follow an edge at a certain predefined distance. It cannot, however, deal with a corner if one is encountered. Instead, this ability is provided by the concave and convex corner behaviours. The *Concave Corner* behaviour will turn the robot if a concave corner is detected. The inputs to this behaviour are the range readings from the sonar sensors located around the robot and a message from the blackboard informing the behaviour if the robot is

following a left hand or a right hand edge. The output from the behaviour is a set of commands for the motors that specify their speed and direction of rotation and also a message to be placed on the blackboard for the benefit of the *Mapping* and *Navigation* behaviours (see figure 42).

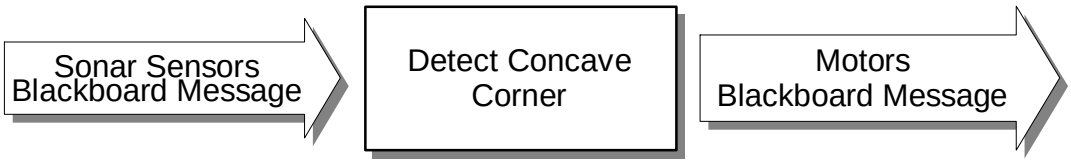


Figure 42 Detect Concave Corner Behaviour

For the purpose of this explanation, it is assumed that the robot is initially following a left hand edge when the corner is detected. A similar procedure is used for a right hand edge. The only difference is the sonar sensors employed.

The behaviour detects a concave corner by monitoring one of the forward looking sonar sensors. This sensor returns the distance to the object directly in front of the robot. If the sensor returns a distance of less than 14 inches, this is an indication that a concave corner has been found (see figure 43). If this occurs, the robot will immediately stop travelling forward. It will then turn through an angle to be parallel with the following edge. The robot turns by differentially driving each of the wheels at the same speed. This allows the robot to turn on the spot resulting in a zero turning radius. To know when to stop turning, the behaviour monitors the distances from the -90° sensor and the -45° sensor. By comparing these two distances, it can be established if the robot is parallel to the edge. When the robot is parallel to an edge, the difference in readings between the -90° sensor and the -45° sensor is approximately three inches (see figure 44). While turning, it would seem appropriate to stop when this value is read. However, in testing, it was found that this resulted in the robot turning through too great an angle. Instead, the robot stops when a difference of one is obtained. Before the robot commences a turn, it is already parallel to an edge. As a result, unless a further condition is introduced, the robot would stop turning the immediate second it starts. To overcome this, the distances from the forward looking sensors are also monitored. In order to stop turning, these must return a distance greater than 18 inches.

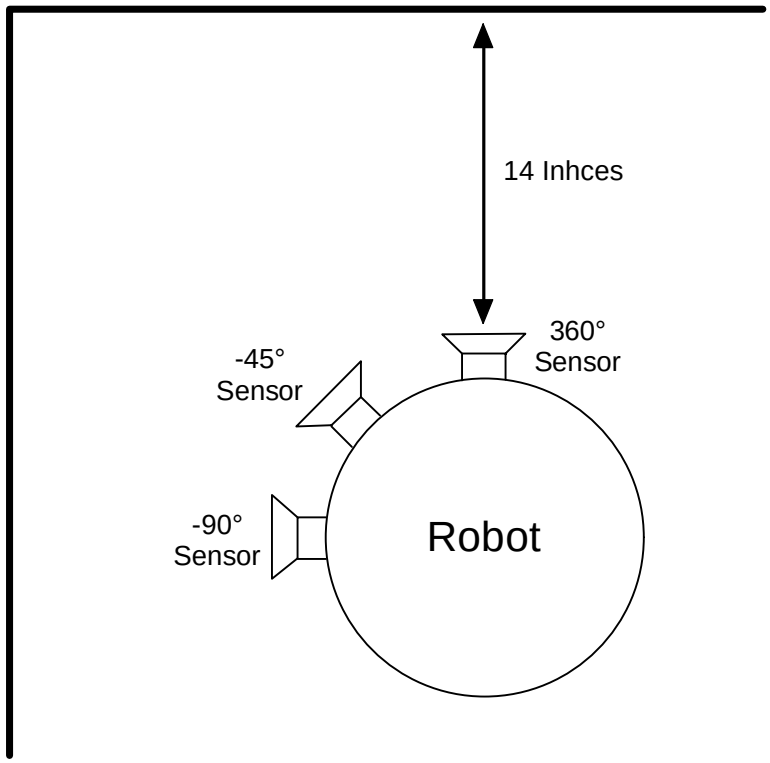


Figure 43 Detecting a Concave Corner

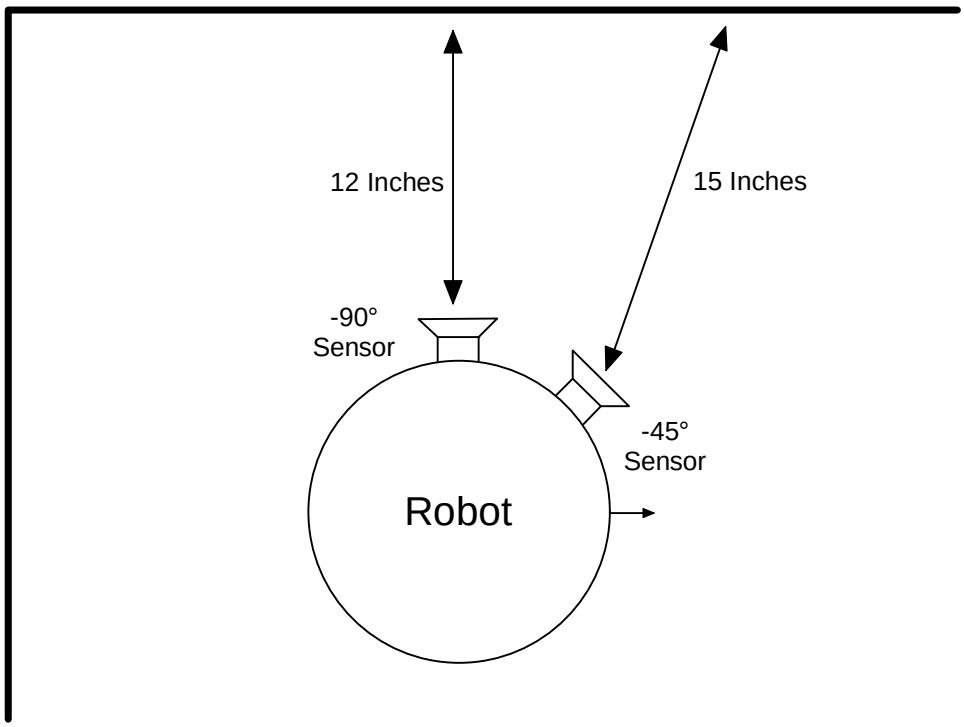


Figure 44 Re-establishing a Parallel Pose to the Following Edge

Once a concave corner is detected, this behaviour will immediately place a message onto the blackboard. This is for the benefit of both the *Mapping* and *Navigation* behaviours. The *Mapping* behaviour will use it to build up a topological map of the environment. The message signifies that a landmark has been detected so it can add a node to the map. This will be explained in detail later on.

4.4.3 Convex Corner Behaviour

The *Convex Corner* behaviour will turn the robot if a convex corner is detected. The inputs to this behaviour are the range readings from the sonar sensors located around the robot and two messages from the blackboard. One of these messages is used to inform the behaviour if the robot is currently following a left hand or a right hand edge. The other message is used to indicate if the navigation behaviour is currently active. The output from the behaviour is a set of commands for the motors that specify their speed and direction of rotation and also a message to be placed on the blackboard for the benefit of the *Mapping* and *Navigation* behaviours (see figure 45). In addition to detecting convex corners, this behaviour can also detect doors.

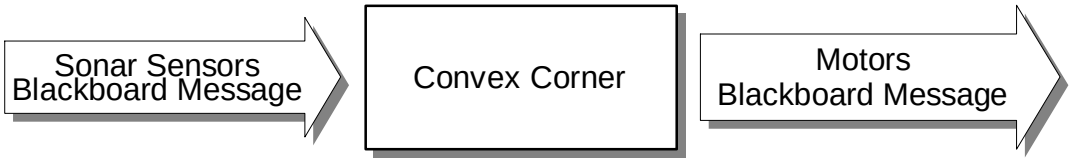


Figure 45 Convex Corner Behaviour

For the purpose of this explanation, it is assumed that the robot is initially following a left hand edge when the corner or door is detected. A similar procedure is used for a right hand edge. The only difference is the sonar sensors employed.

To detect a convex corner or a door, the distance returned from the -90° sensor is monitored. If this returns a distance greater than 20 inches away, this is an indication that a either a convex corner or a door has been found (see figure 46). Which one it is cannot be determined at this stage. Instead the robot must travel straight ahead for a certain

predefined distance. If the robot detects an edge at this point using the -90° sensor, then it is assumed that a door was detected (see figure 47). If, on the other hand, no edge is detected, a convex corner is assumed. In this case, the robot must reverse for a certain distance and then turn. When the robot is turning at this stage, there are no reference cues to tell it when to stop. Instead, it turns through a default angle of 90°. At this stage, the robot will be in a similar position to that shown in figure 48. Having established this position, the robot will then travel straight ahead until an edge is found. This occurs when the distance returned from the -90° sensor is less than 20 inches.

Once a convex corner or door is detected, this behaviour will place a message onto the blackboard. This is for the benefit of both the *Mapping* and *Navigation* behaviours. One of the messages contained in the blackboard which is read by this behaviour is used to indicate if the navigation behaviour is currently active. The purpose of this is simple. If the robot is currently navigating and it knows that the next feature to be detected will be a convex corner, then it can turn immediately at the corner instead of having to travel straight ahead first.

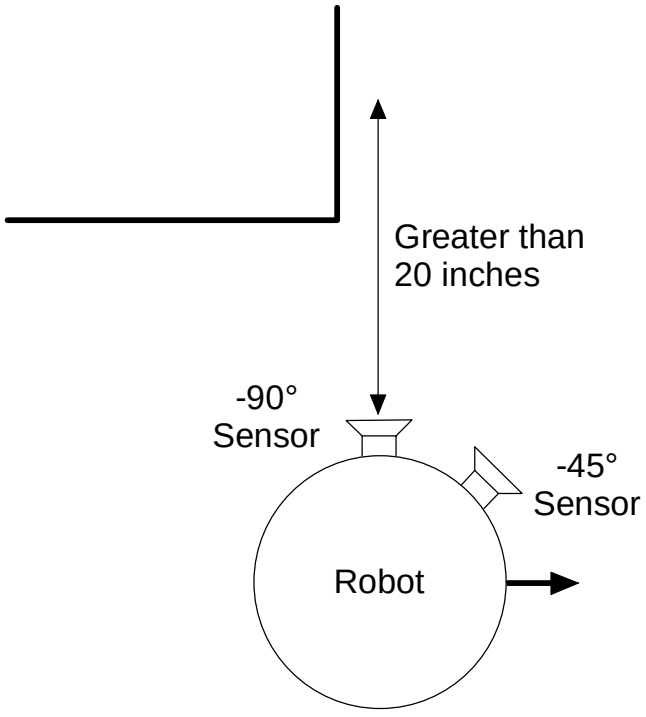


Figure 46 Detecting a Convex Corner or Door

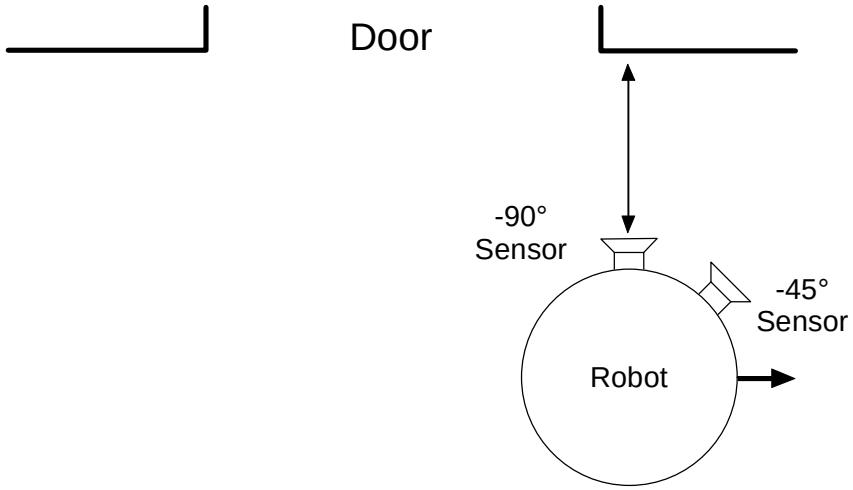


Figure 47 Detecting a Door

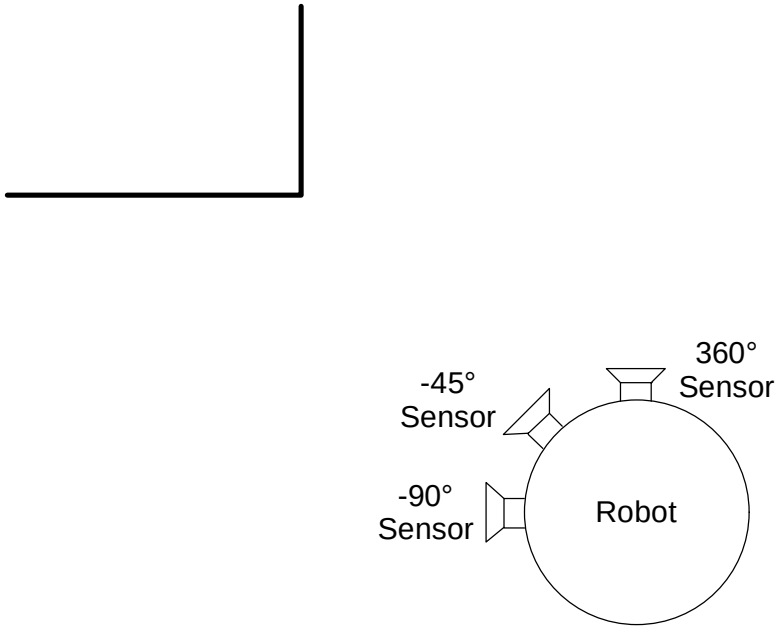


Figure 48 The Robot's Position After Turning at a Convex Corner

4.4.4 Search For Edge Behaviour

The purpose of this behaviour is to lock onto an edge and ensure that the robot is parallel to it. This is important, since when the robot is initially switched on, it could be placed at any random location in the environment. The ability of the *Edge Following* behaviour to be able to accurately follow an edge is based on the assumption that it was initially parallel to an edge in the first place. The *Search For Edge* behaviour ensures that this condition can be met. The inputs to this behaviour are the range readings from the sonar sensors located around the robot. The output is a set of commands for the motors that specify their speed and direction of rotation and also a message to be placed on the blackboard (see figure 49).

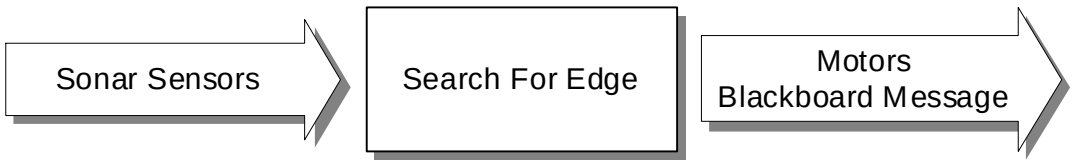


Figure 49 Search For Edge Behaviour

To search for an edge, the robot simply maintains forward motion until one of the sensors returns a distance of less than 16 inches. When this occurs, the robot is in close proximity to the edge but not parallel with it. At this stage, the robot's orientation to the edge must be determined. This is necessary to establish whether the robot will lock onto it from its left hand side or its right hand side. To ensure that the robot is parallel to the edge, a similar procedure is used to the *Concave Corner* behaviour. Following this, a message is placed on the blackboard indicating whether the robot is locked onto the edge from its left hand side or its right hand side. This message will be read by the *Edge Following* and *Convex and Concave Corner* behaviours.

4.4.5 Mapping Behaviour

If a mobile robot contains an internal model or map of its environment, it can perform navigational tasks such as travelling to a particular location in the environment. Having this ability is important for many types of robots. For example, office cleaning robots and

security robots both need a map to go about their tasks. To investigate mapping in the context of a behaviour based architecture, the robot has been equipped with the ability to build up a map of the environment and use it to travel to specific locations. Two behaviours are used to implement these abilities - the Mapping behaviour and the Navigation Behaviour. The Mapping behaviour constructs a map of the environment whereas the Navigation behaviour uses it to travel to locations specified by a user. Before discussing the details of the Mapping behaviour, an explanation of the type of map chosen will first be presented. The robot's intended environment contains many distinctive landmarks which it can easily recognize. As a result, it was decided that a metric topologic map would be used to represent it.

A topological map consists of a number of distinct locations in the environment, such as doors and corners, that the robot can recognize as it travels about. The robot builds up a representation of the environment by storing these distinct locations as nodes in the map. Associated with each node is a link which shows the relationship between different landmarks or features. For example, from one landmark, there may be two paths leading to other landmarks that the robot can travel to. The map can be considered as a graph with nodes representing distinct locations and pathways between the locations represented as arcs connecting the appropriate nodes. A link which establishes the connection between two landmarks has to be identified by the robot. This is done by travelling the corresponding route between the two landmarks. To make constructing the topological map more straightforward, the links can be established at the same time as the landmarks are identified. This is the approach used by the Mapping behaviour which will be described shortly. With a topological map, there is no metric or geometric information stored. The advantage of this is that accumulating odometry errors do not impede the accuracy of the map. One of the problems, however, associated with this is perceptual aliasing. Distinct locations within the environment may appear similar to the robot's sensors. This is especially the case with doors and corners. How can the robot tell one from the other. To overcome this limitation, the introduction of some metric information into the map is often used. The length of paths between landmarks is often stored with each node. The addition of this information can remove any ambiguity between similar landmarks helping the robot to identifying where it is. This path length information is usually obtained from onboard odometry. The problem with odometry, however, is that it is notoriously unreliable,

producing errors that gradually accumulate over time. With topological maps, the distance between landmarks is usually small. Since the metric information stored only shows the length of the path between landmarks, the errors that do result are usually perfectly acceptable. It is important to realize that the errors that do accumulate for one path will not be transferred to another. A topological map which uses some metric information is referred to as a metric topological map. This is the type of map that has been utilized by the robot.

Figure 50 shows an example of an environment where the robot may be expected to operate. This environment consists of 6 distinct landmarks that the robot can recognize (labelled 1 to 6) as it travels about. While exploring the environment, the robot builds up a map in an incremental fashion. As each landmark is detected, the Mapping behaviour will add a node to the map. For each node, the following information is stored.

- Type of Landmark
- Number of Next Node
- Distance to Next Node
- Number of Previous Node
- Distance to Previous Node

The following table shows an example of the nodes constructed for this environment. In this example, it is assumed that the first landmark to be detected by the robot is the one labelled 1 and that it is travelling in a clockwise direction.

	1	2	3	4	5	6
Type of Landmark	Concave	Convex	Concave	Concave	Concave	Concave
Number of Next Node	2	3	4	5	6	1
Distance to Next Node	50	48	45	96	95	48
Number of Previous Node	6	1	2	3	4	5
Distance to Previous Node	48	50	48	45	96	95

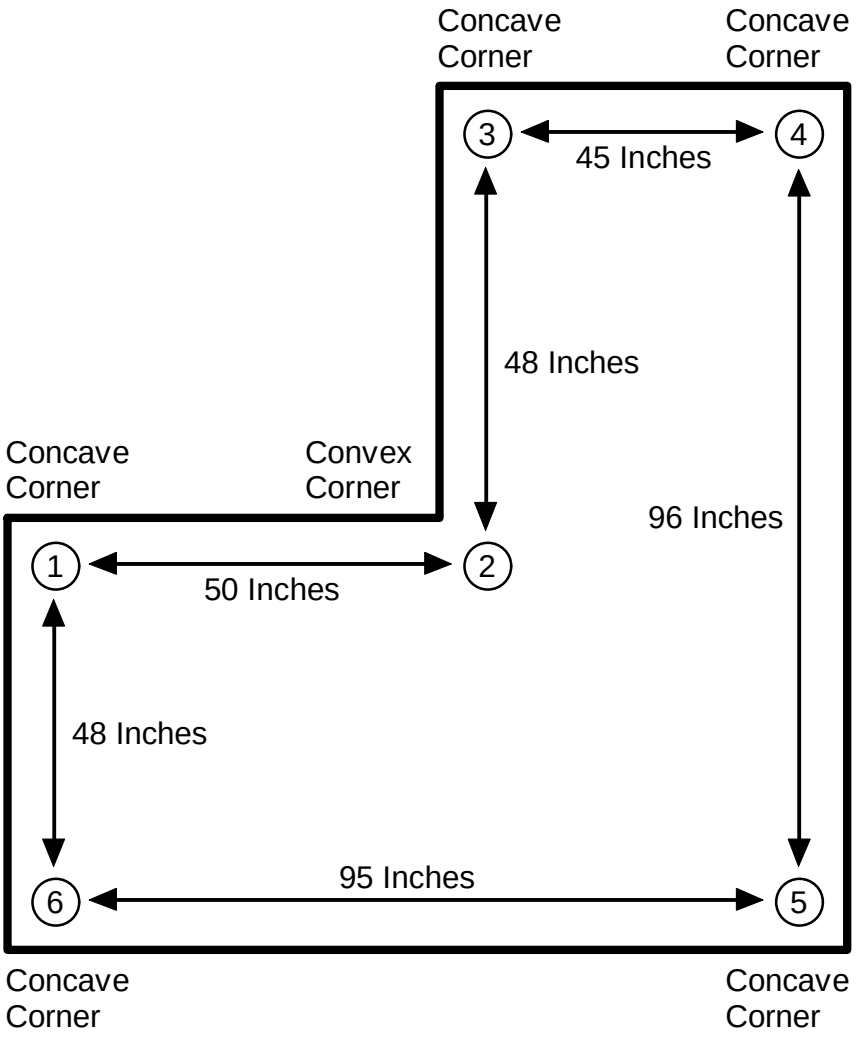


Figure 50 Example Environment

How does the Mapping behaviour known when a landmark has been detected? There are two ways in which this can be done. Firstly, it could continuously monitor the environment itself using the robot's sensor to determine if certain environmental features are present. The second method is to be informed by some other behaviour if a landmark has been found. This second approach has been adopted here. The Mapping behaviour continuously monitors the blackboard for certain messages. These messages are generated by the landmark detection behaviours, namely the concave corner and convex corner behaviours, whenever they detect a landmark. Once a landmark is detected, these behaviours will immediately place the message onto the blackboard for the benefit of the Mapping behaviour.

The message placed onto the blackboard indicates the specific type of landmark detected. For example, the landmark may be a concave corner, a convex corner or a door. Having received this message, the Mapping behaviour will place a node onto the map to represent the landmark. The node contains a number of data fields which must be filled. Firstly, the type of landmark that was detected must be stored. The next item to be stored is the number of the previous node. This is obtained by checking the map to see the number of the last landmark to be detected. The distance to the previous landmark must then be determined. This is done by reading the robot's onboard odometry counter Each time the robot leaves a landmark, it will set the odometry counter to zero. Therefore, whenever the robot arrives at a new landmark, this counter will contain the distance to the previous landmark. A slight problem occurs, however, if this system is used. The counter can only record the distance from the point where the robot originated to the point where it stops. This distance will not be the same as the length of the path between the landmarks. In other words, it may not be the length of the straight edge or wall joining them. See figure 51 for an explanation of this.

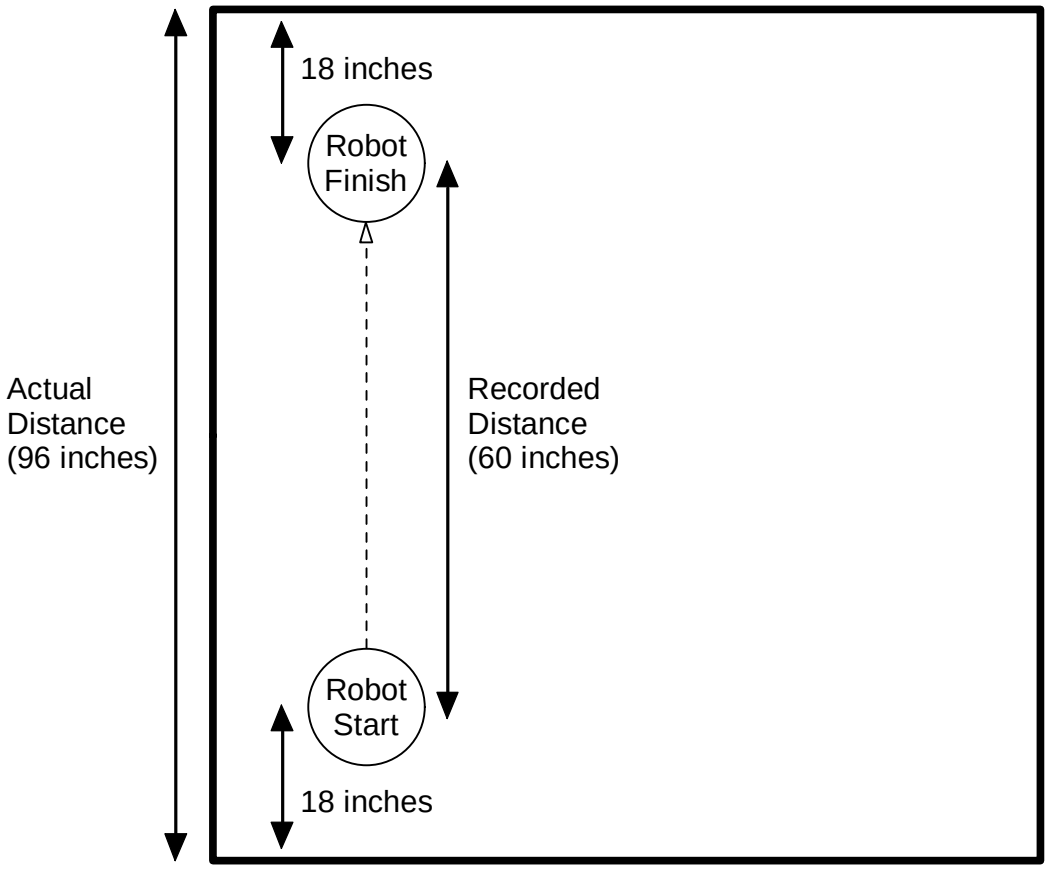


Figure 51 Calculating the Distance Between Landmarks

To overcome this problem, an extra distance must to be added to each reading. From the diagram, it can be seen that a value of 18 inches will have to be added twice. This value represents the distance from each landmark to the centre of the robot. When the robot arrives at a landmark, it will add the range reading from the forward looking sonar sensor to the odometry counter. It will also add a value of 6 inches representing the radius of the robot. Depending on the reading returned by the sensor, a value of 18 inches may not always be obtained. This can vary slightly. Each time the robot leaves a concave corner, it will store the distance from the rear facing sonar sensor to the edge or wall behind the robot. This distance is calculated by the Concave Corner behaviour and stored in the blackboard until such time that the Mapping behaviour needs to access it. Arriving at a landmark, the Mapping behaviour will retrieve this value from the blackboard and add it to the odometry counter plus a value of 6 inches to represent the robot's radius. The final resulting value obtained should give quite an accurate indication of the distance between the two landmarks.

If the robot arrives at a convex corner, a slightly different procedure is used (see figure 52). In this case, the value of 18 will only have to be added once.

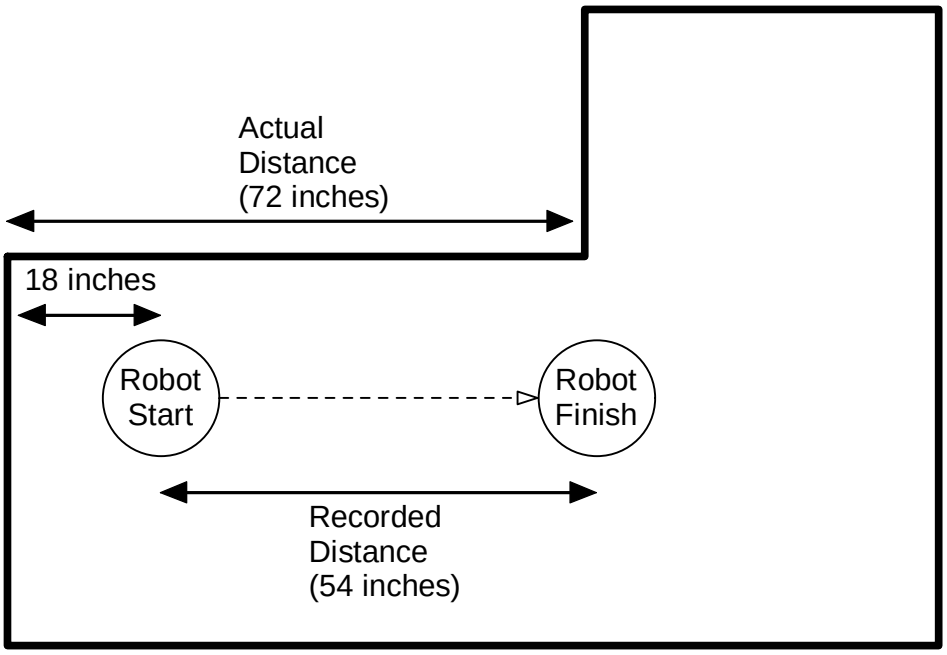


Figure 52 Calculating the Distance Between Landmarks

There are two more data fields to be filled in. These represent the number of the next node and the distance to the next node. The distance to the next node cannot be obtained until such time that the robot travels to it its corresponding landmark. This makes the task of constructing each node in the map a two step process. The information is accumulated by travelling to two different landmarks. At each landmark, the Mapping behaviour will fill in part of the details for the last landmark detected. Referring to the example shown in figure 50, when the robot arrives at node 2, it will know the distance it has travelled from node1. This information can then be used to complete the data field entries for node 1.

As the robot travels about the environment adding each landmark to the map, it will eventually arrive back to place where it started. Some kind of mechanism must be introduced to stop the map making process at this stage. As the robot is mapping the environment, it is continuously comparing detected landmarks with landmarks previously detected. If a match is found, the robot can stop constructing the map.

4.4.6 Localisation Behaviour

When the robot is initially switched on, it could be placed at any random location in the environment, without knowing its whereabouts. This is an undesirable situation. For the robot to perform useful tasks such as travelling to specific locations, it must be aware of its correct position. By using the map, however, the robot can re-establish its correct position in a relative fashion. This is the purpose of the localisation behaviour. This behaviour will build up a local map of the environment and compare it with the global map previously built. The inputs to this behaviour are the messages from the blackboard generated by the landmark detection behaviours, namely the concave corner and convex corner behaviours. The output from the behaviour is a message to be placed onto the blackboard indicating the robot's current position (see figure 53).

Figure 53 Localisation Behaviour

The local map built by this behaviour is constructed in the same manner as the map built by the Mapping Behaviour. The local map, however, consists of only two nodes. These nodes represent the two most recent landmarks detected. The local map is continuously updated as the robot travels about. Also at the same time, a comparison is continuously performed between the local map and the global map. If a match occurs between the two, the behaviour can ascertain the current node that the robot is located at. When comparing the local map with the global map, an exact match will usually never occur. Each time the robot calculates the distance between landmarks, a slightly different value will always be obtained. To take account of this while comparing the maps, a certain error margin is introduced to allow the values to vary slightly. Once the current node is known, a message will be placed onto the blackboard for the benefit of the navigation behaviour. This will indicate the robot's current position.

4.4.7 Navigation Behaviour

This is a very simple behaviour. Its purpose is to travel to specific locations in the environment which have been specified by the user. The inputs to this behaviour are the messages from the blackboard generated by the landmark detection behaviours and the destination node to travel to. The output is a message to be placed onto the blackboard to instruct the motors to stop when the robot arrives at the destination (see figure 54).

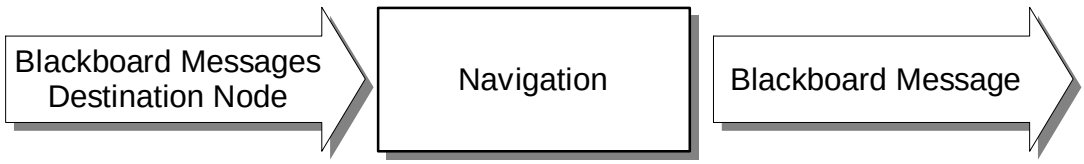


Figure 54 Navigation Behaviour

For this behaviour to work correctly, the robot must initially be aware of its correct position. This is taken care by the localization behaviour. To travel to a specific destination, the behaviour simply keeps track of each landmark encountered. If the destination node matches the current node or landmark, the robot has arrived at the destination and a command is issued to instruct the motors to stop. This ensures that the robot remains stationary at the destination node.

4.4.8 Arbitration Function

Four of the behaviours described in the previous section can all send commands to the motors simultaneously. To prevent a conflict from occurring, the arbitration function has to select a single behaviour output to send to the motors. In keeping with the subsumption architecture, each of the behaviours is layered according to their relative importance (see figure 55).

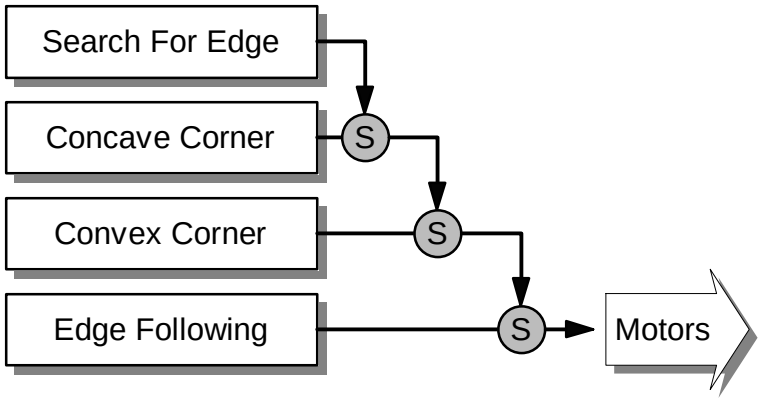


Figure 55 Layering of Behaviours

4.5 Summary

This chapter has described the control architecture used on the robot. A modified form of a behaviour-based architecture has been developed for the robot. This is based on the standard subsumption architecture with the addition of a concept known as a blackboard.

The blackboard allows the sharing of system state and knowledge between behaviours to help them perform their tasks. An initial set of low-level behaviours was developed to demonstrate that the robot could operate reactively based on sensor stimuli. Following the implementation of these, a more complex set of behaviours was developed. These equip the robot with the ability to explore and map its environment. The robot can build up a topological map of the environment which it can subsequently use to navigate to specific locations specified by the user.

5. A FUZZY LOGIC BASED NAVIGATION SYSTEM

5.1 Limitations of Subsumption

One of the limitations associated with subsumption is that the arbitration technique employed only allows a single behaviour to be active at any one time. While this is satisfactory in many situations, there are times when a combination of two behaviours is required. Take, for example, navigating towards a target and avoiding obstacles. Each of these could be implemented as a single behaviour each. So long as no obstacles are detected, the robot will gracefully head towards its target location. If an obstacle is detected, however, the obstacle avoidance behaviour becomes active and steers the robot away from the obstacle. The problem with this is that the obstacle avoidance behaviour has no knowledge about the target location, thus it could steer the robot in any direction to avoid the obstacle. An example of this is shown in figure 56. Here the robot has only a 50% chance of making a correct turn to the left. In many situations, this may work perfectly well, but there are times when it may be desirable for the robot to steer in a direction which takes it closer to its target location. This can be achieved by combining the output of the two behaviours. This is referred to as command fusion. The output from the target following behaviour and the obstacle avoidance behaviour are combined to produce a heading that takes the robot towards its target location while avoiding obstacles. One method used to perform this task was developed by David Payton and Ken Rosenblatt (1989).

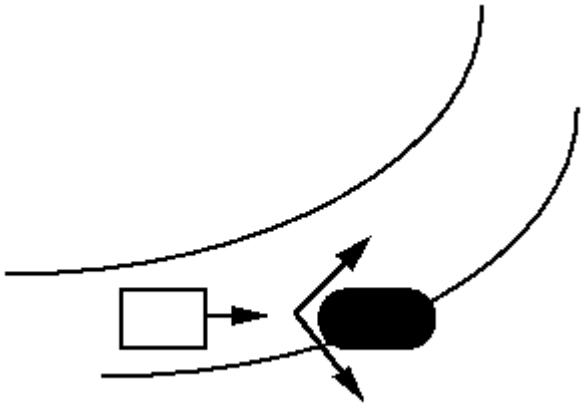


Figure 56 Two possible choices.

5.2 Enhancing Subsumption

5.2.1 Payton and Rosenblatt's Command Fusion Network

This technique came about from Payton and Rosenblatt's observation that a fixed priority based arbitration scheme usually results in loss of information, which makes decision making much more difficult. In their system, instead of each behaviour outputting a single control value, they output a set of nodes. Each node corresponds to a possible control decision. A certain activation level is assigned to each of the nodes, which represents the confidence regarding the control decision. An example of this is shown in figure 58, which represents the situation given in figure 56. This shows one of their networks for combining the two behaviours *Turn-for-Obstacle* and *Track-Road-Edges*. The size and colour of each node represents the behaviours' activation for that command. A node's size represents the magnitude of its activation. Solid black colours represent positive activation while white colours represent negative activation. For example, the *Turn-for-Obstacle* behaviour in figure 57 has a large positive activation level for the "hard left" node and a large negative activation for the "straight ahead" node. To combine the outputs of two behaviours, a weighted sum technique is used to combine the activation strengths of corresponding nodes. The weight associated with a behaviour reflects the degree of importance of the behaviour's suggestion. For example, the *Turn-for-Obstacle* behaviour has a higher weight than the *Track-Road-Edges* behaviour since it is more important to avoid colliding with obstacles than to follow the track. The final control command is the node with the largest positive activation in the combined behaviour based on the winner-take-all selection strategy.

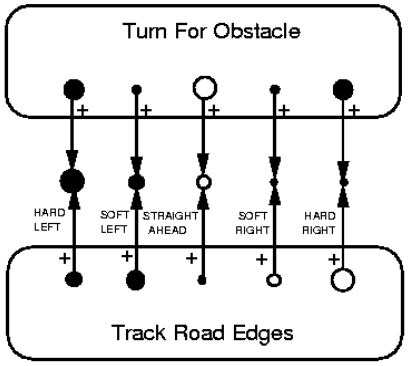


Figure 57 A Payton and Rosenblatt network for fusing two behaviours

5.2.2 Using Fuzzy Logic

Another technique for combining the outputs from two behaviours is by using fuzzy logic. In this system, each behaviour consists of a set of fuzzy control rules and a fuzzy inference module. The output from each behaviour is a fuzzy set. These sets are then combined through a command fusion module and defuzzified to produce a crisp output value. In order to test how successful this procedure is, a navigation system consisting of a target following behaviour and an obstacle avoidance behaviour was developed which uses fuzzy logic to combine the two behaviours.

Before the technique is explained, a brief description of fuzzy logic will be given. Fuzzy logic is a suitable alternative to conventional control systems when a mathematical model for the control system is unavailable. This is particularly true in the case of unstructured and dynamic environments where a large amount of uncertainty exists. In designing a fuzzy logic system, human experience and expertise is employed in the form of heuristic control knowledge. In a fuzzy logic controller, a set of IF-THEN rules are used to capture the relationship between the observed input variables and the output control variables. The output of all the rules are then combined to obtain a fuzzy conclusion for each control variable. These fuzzy conclusions are finally defuzzified resulting in a crisp output value.

5.3 Fuzzy Logic Navigation System

The navigation system being described here consists of two basic behaviours - an obstacle avoidance behaviour and a target following behaviour. Each behaviour has two components. These consist of a set of fuzzy rules and a fuzzy inference module. The fuzzy rules explicitly capture the control strategy of the behaviour in the form of linguistic rules, while the fuzzy inference module implements a fuzzy inference scheme appropriate for the behaviour. The fuzzy control recommendations generated by the two behaviours are fused together and defuzzified to produce a crisp output value. This output value represents the most appropriate direction for the robot to steer towards. Figure 58 shows a block diagram of the controller. The basic procedure for determining the crisp output value consists of the following four steps: (1) The target following behaviour produces a fuzzy set that represents the desired turning directions. (2) The obstacle avoidance behaviour produces a fuzzy set that represents the disallowed turning directions. (3) The command fusion module combines these two fuzzy sets into one output fuzzy set. (4) This output fuzzy set is then defuzzified to produce a crisp output value. The output from each behaviour is maintained in fuzzy set form so as to reduce possible loss of information in command fusion.

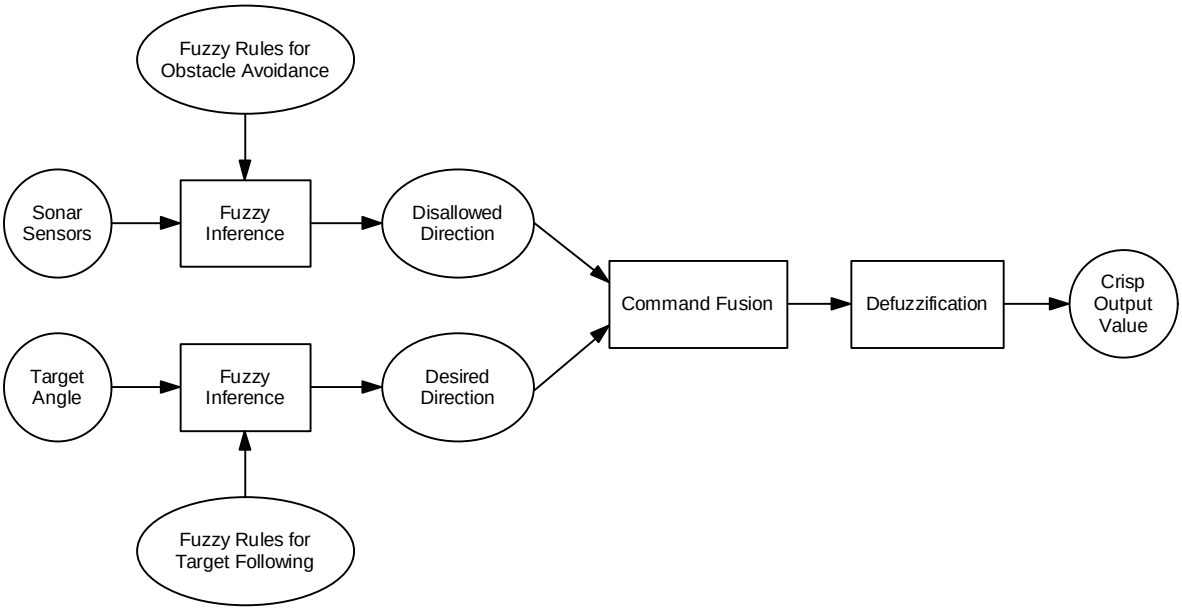


Figure 58 Fuzzy Logic Navigation Controller

As an example on the technique works, the situation shown in figure 59 will be used. In this example, the robot is facing straight ahead. The target following behaviour suggests that the robot should turn to the left to head towards the target. However, to avoid colliding with the obstacle on the left, the robot must go straight ahead for a little while longer.

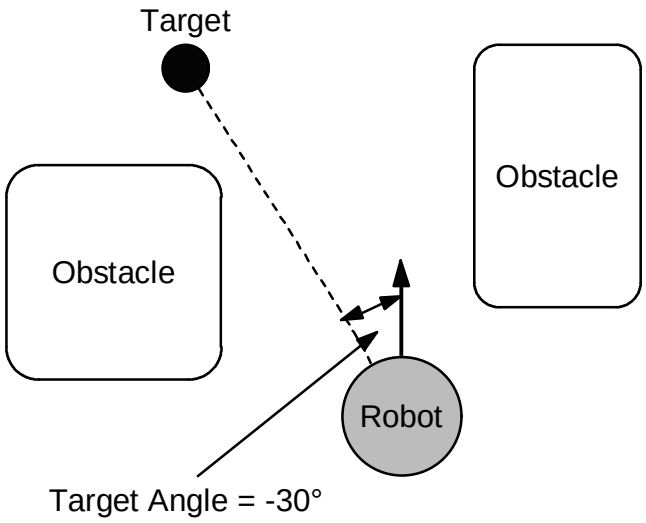


Figure 59 Example

5.3.1 Target Following Behaviour

The input to the target following behaviour is a value representing the angle between the robot's current heading and the location of the target. In order to give the robot a certain amount of flexibility in reaching its target, this specific angle is broadened into a more general desired direction by using a set of fuzzy rules. If this were not done, the robot would not be able to turn in order to avoid any obstacles. For the example shown in figure 59, the robot is facing 0° and the target angle is -36°. To keep the example as simple as possible, only two fuzzy rules are used here by the target following behaviour. These are shown below:

If Target Angle is Around 0° **Then** Desired Direction is Forward

If Target Angle is Around -45° **Then** Desired Direction is Left-Forward

Around 0° and Around -45° are membership functions of the variable Target Angle. Forward and Left-Forward are membership functions of the variable Desired Direction. These are shown in figure 60 below.

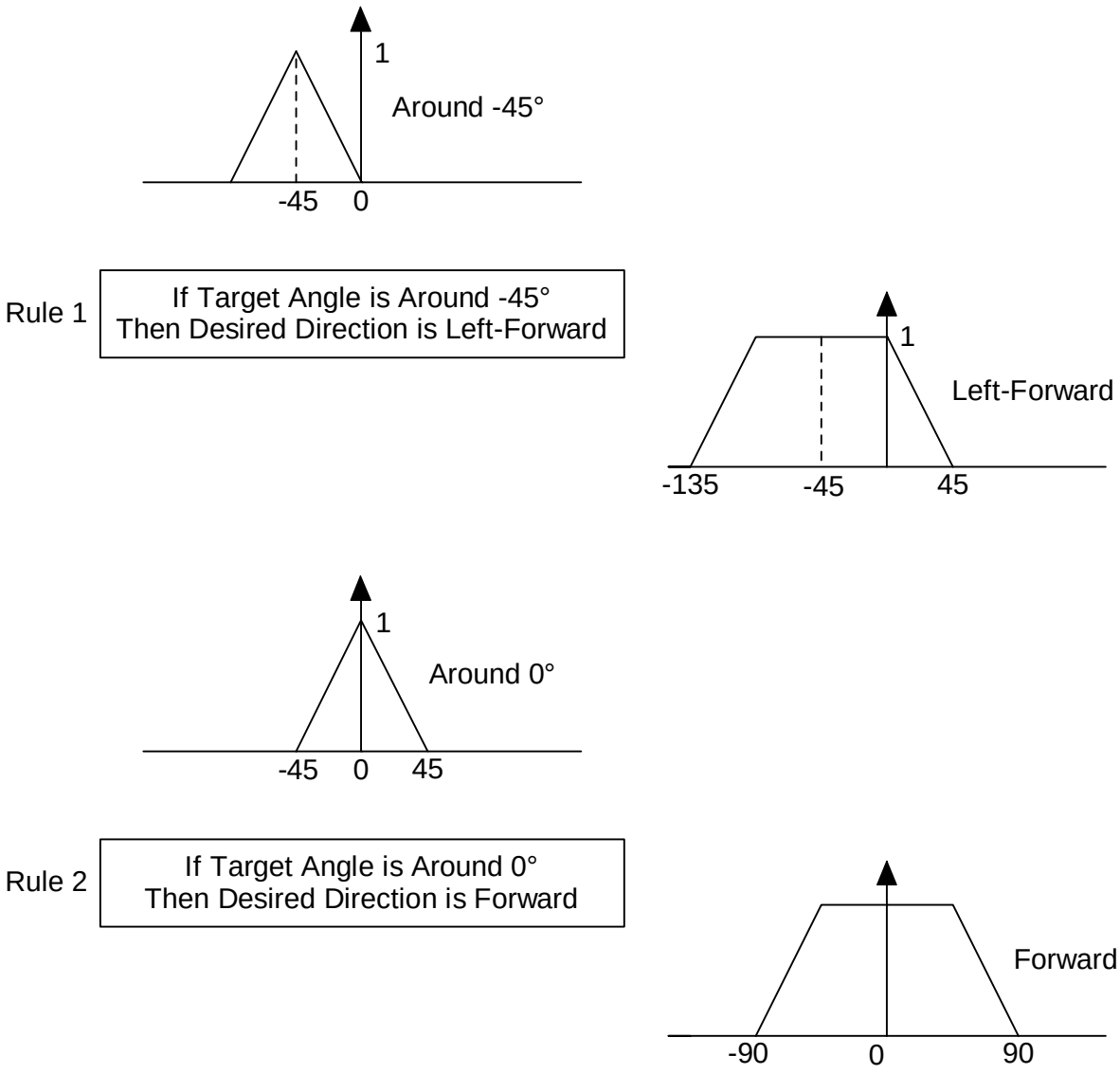


Figure 60 Membership Functions and Fuzzy Rules for Target Following

The fuzzy inference module for the target following behaviour combines the desired directions recommended by the fuzzy rules using a weighted sum technique. An example of this is shown in figure 61 for a target angle of -36° using the two rules given above. The

membership functions of the variable Target Angle are designed such that the sum of their membership values for an angle is exactly one.

For a target angle of -36°, the degree of membership of the function Around -45° is 0.8 and the degree of membership of the function Around 0° is 0.2. These values are used to set the activation levels for the membership functions of the output variable Desired Direction. In the case of rule 1, the activation level for the function Left-Forward is 0.8. For the second rule, the activation level for the function Forward is 0.2.

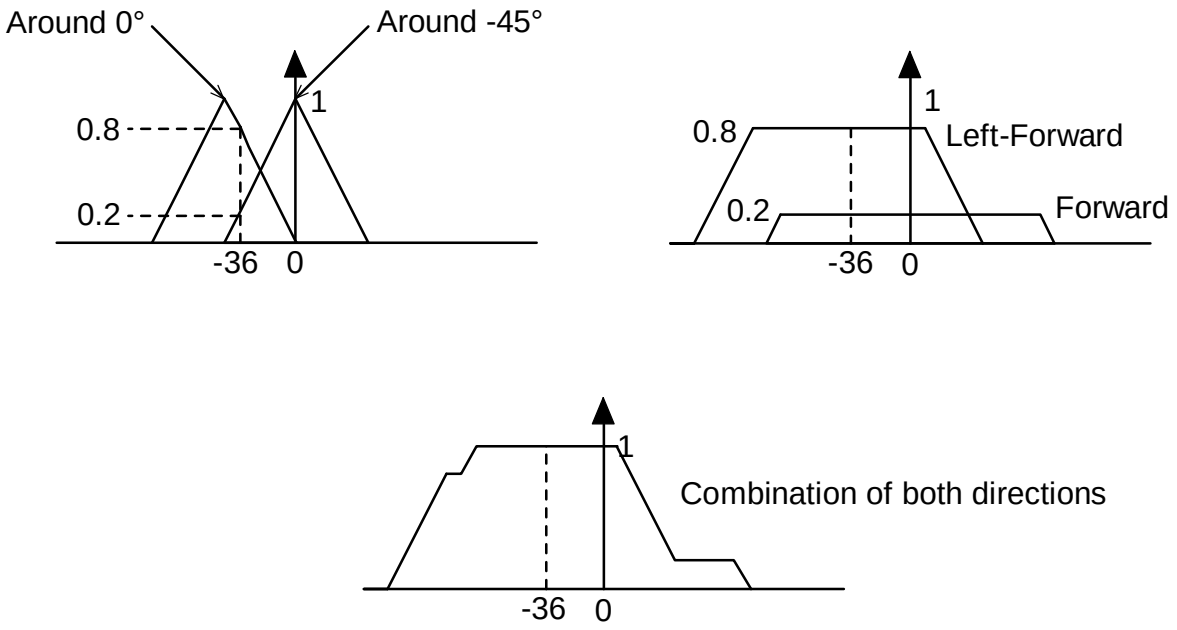


Figure 61 Computing Desired Direction

5.3.2 Obstacle Avoidance Behaviour

This behaviour uses sonar sensors to determine the distance to obstacles located around the robot. This information is then used to generate a fuzzy set that represents the disallowed directions of travel. A disallowed direction is a direction that would result in the robot coming into contact with or passing close to a nearby obstacle. The behaviour has one input variable for each sonar sensor. Additionally, each variable has one membership function

associated with it. This function is called NEAR and represents the degree of how close an obstacle is to the robot. In keeping with the example above, this behaviour only uses two fuzzy rules. These are shown below:

If 0° Sensor Distance is Near **Then** Disallowed Direction is Forward

If -45° Sensor Distance is Near **Then** Disallowed Direction is Left-Forward

Near is a membership function of 0° Sensor Distance and -45° Sensor Distance. Forward and Left-Forward are membership functions of the variable Disallowed Direction. These are shown in figure 62 below.

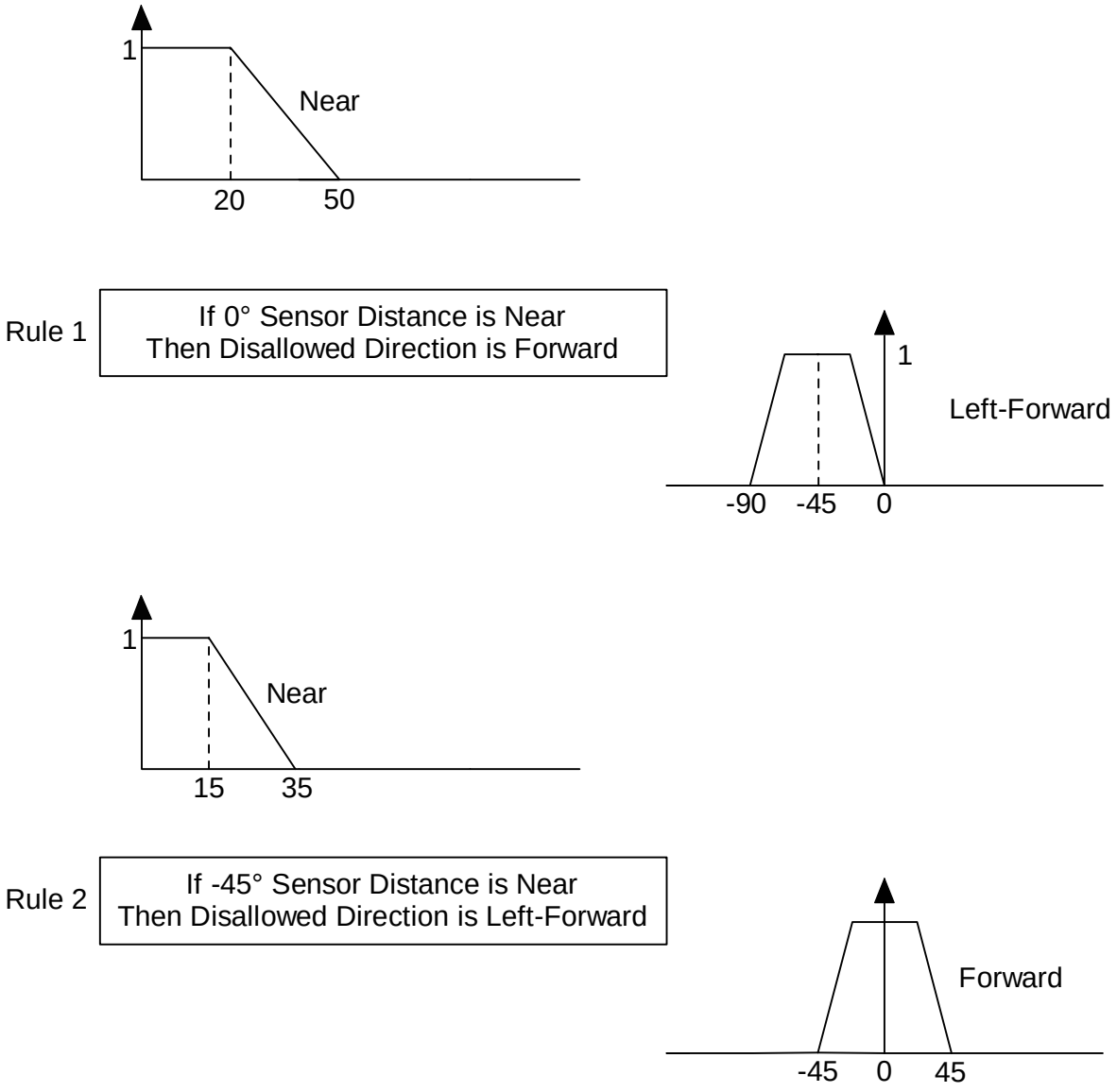
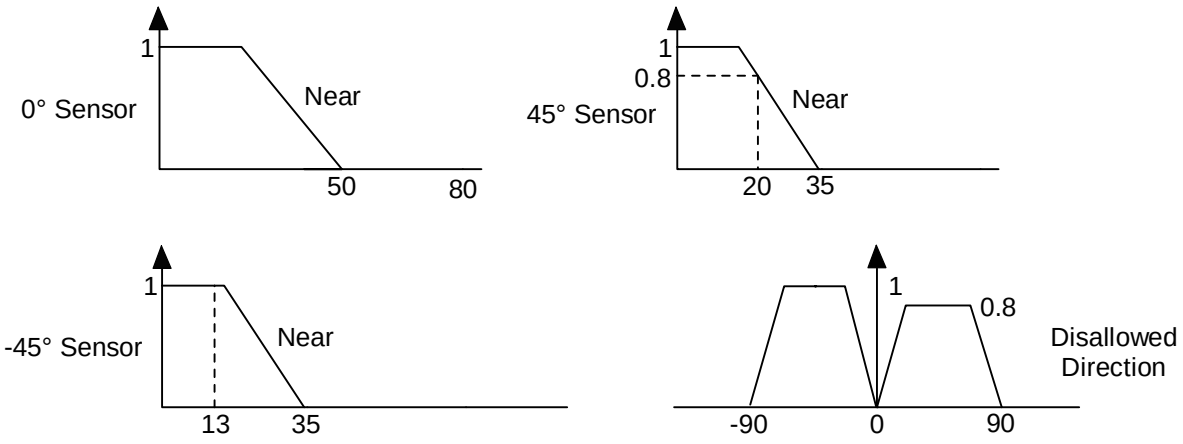


Figure 62 Membership Functions and Fuzzy Rules for Obstacle Avoidance

In the case of the 0° Sensor, for any obstacles located less than 20 inches away, the degree of membership of the function NEAR is 1. For obstacles located greater than 20 inches away and less than 50, the degree of membership gradually decreases to 0. It is worthwhile pointing out that the shape of the membership function NEAR for the 0° Sensor and the -45° Sensor are slightly different to each other. This is due to the fact that obstacles detected at the front of the robot pose more of a threat than ones at the sides. For example, an obstacle detected at a distance of 20 inches by the 0° Sensor is considered closer than an object detected at the same distance by the -45° Sensor.

The fuzzy inference module for the obstacle avoidance behaviour combines the disallowed directions recommended by the fuzzy rules using the MAX operator. An example of this is shown in figure 63 for sensor inputs based on the situation given in figure 59. The MAX operator is used by the fuzzy inference module because it is consistent with the intuition that the degree a travel direction is disallowed should be determined by the sensor that has the strongest opinion about it.



Sensor Angle	Distance Returned	Rule Firing Strength
0°	80	0
-45°	13	1
45°	20	0.8

Figure 63 Calculating Disallowed Direction

5.3.3 Command Fusion

The third component of the fuzzy logic navigation controller is a command fusion module. This combines the two fuzzy sets produced by the target following behaviour and the obstacle avoidance behaviour into one output fuzzy set. This output fuzzy set represents the steering angle for the robot. The steering angle is both *Desired* from the viewpoint of the target following behaviour and *Not Disallowed* from the viewpoint of the obstacle avoidance behaviour. In fuzzy logic, an AND operation is carried out using the MIN operator. The following equation calculates the degree of membership for each angle in the output fuzzy set:

$$\begin{aligned}\mu_{\text{Steering Angle}}(x) &= \mu_{\text{Desired AND Not Disallowed}}(x) \\ &= \min(\mu_{\text{Desired}}(x), \mu_{\text{Not Disallowed}}(x)) \\ &= \min(\mu_{\text{Desired}}(x), 1 - \mu_{\text{Disallowed}}(x))\end{aligned}$$

$\mu_{\text{Desired}}(x)$ is the degree of membership for the angle x in the target following output fuzzy set. $\mu_{\text{Disallowed}}(x)$ is the degree of membership for the angle x in the obstacle avoidance output fuzzy set. $\mu_{\text{Not Disallowed}}(x)$ is the fuzzy complement of $\mu_{\text{Disallowed}}(x)$. The fuzzy set *Disallowed* represents the directions in which the robot should not steer. In contrast to this, the fuzzy set *Not Disallowed* represents the directions in which the robot is allowed to steer. *Not Disallowed* can also be referred to as the *Allowed* direction of travel.

Figure 64 shows the resultant fuzzy set produced by combining the fuzzy sets from each behaviour. This is based on the example shown in figure 59. In this example, even though the target angle is -36°, most of the fuzzy set produced by the command fusion module resides around 0°. This is correct since 0° is the most appropriate angle for the robot to steer given the presence of the obstacle on the left of the robot. The next section gives a detailed description of how the output fuzzy set is defuzzified to produce a crisp value.

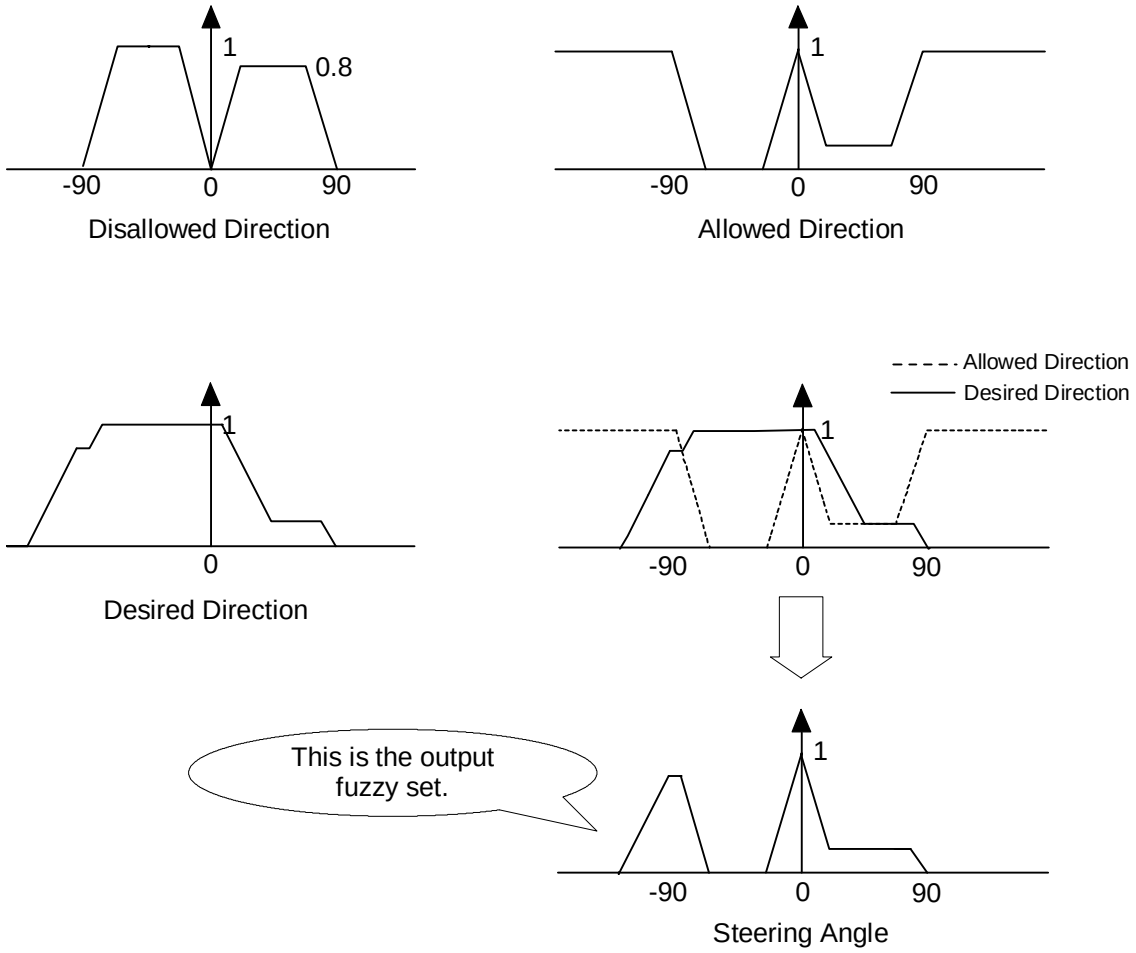


Figure 64 Command Fusion

5.3.4 Defuzzification

The fourth and final component of the fuzzy logic navigation controller is a defuzzification module. This converts the fuzzy output set into a crisp control command which represents the steering angle for the robot. A number of different defuzzification methods exist. Popular ones include the Mean of Maximum (MOM) method and the Centre of Area (COA) method. In order to select a particular method, it is important to understand the linguistic meaning than underlies the defuzzification process.

The MOM method computes the average of the values with the highest degree of membership in the fuzzy set. An example of this is shown in figure 65 below. In this example, the crisp output value produced is 0°. This would steer the robot safely away from

the obstacle. The MOM defuzzification method is basically a winner-take-all arbitration scheme. One of the major drawbacks of using the MOM method on a robot, however, is that it does not use all of the information contained in the fuzzy set. The problem with this is that the crisp values produced will have difficulty in steering the robot smoothly over time.

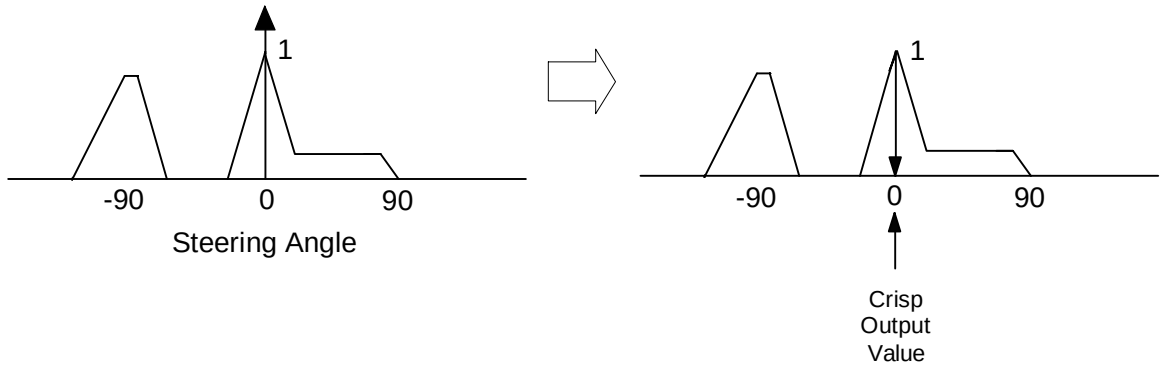


Figure 65 MOM Defuzzification

Another defuzzification method is the Centre of Area method. This computes the centre of gravity of the entire fuzzy set to produce a crisp output value. The following equation is used to calculate this value:

$$Crisp\ Value = \frac{\sum_{k=1}^{n} m(A_k).A_k}{\sum_{k=1}^{n} m(A_k)}$$

In this equation, $\mu(A_k)$ is the degree of membership for the angle A_k. A_k is the angle located at coordinate k along the x-axis. To represent all angles for a complete 360° circle, k must have a value from 1 to 360. Also, A_k may have a value from 1 to 360 or alternatively from -180 to 180.

An example of using the COA method is shown in figure 66 below. In this example, the crisp output value produced is -37°. Clearly if the robot headed in this direction, it would collide with the obstacle on its left. This points out one of the drawbacks of the COA defuzzification method. If the output fuzzy set has a number of peaks in it, then the crisp value produced may be a value that lies in between them.

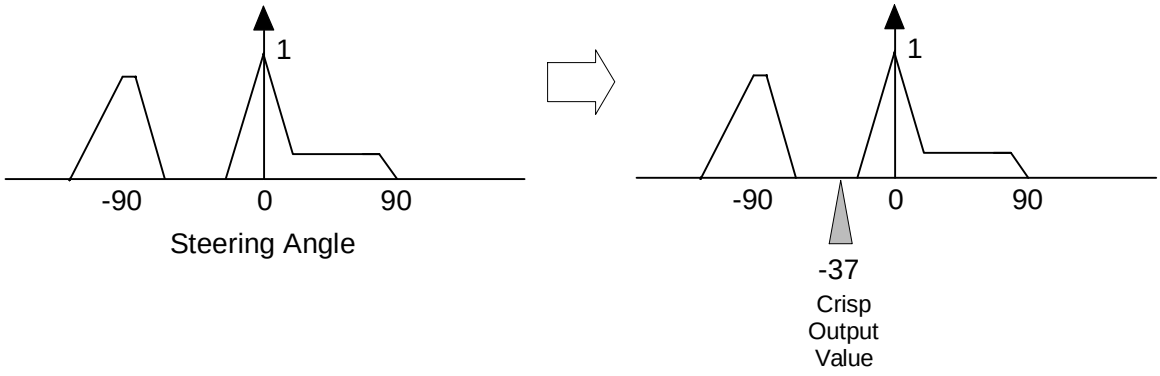


Figure 66 COA Defuzzification

Steering the robot in this direction can often cause it to collide with an obstacle. This can be seen in the example above. The value of -37° lies in between the two peaks in the fuzzy set. Also, the degree of membership for this value is 0 indicating that this is a bad direction in which to steer. The COA method does not ensure that prohibited regions in the fuzzy set are avoided. When defuzzifying the fuzzy set, the COA method comes up with the "best compromise". However, for a mobile robot navigation system, it is more important to come up with the "most plausible result". This requires a different defuzzification technique to be used.

This method is known as Centroid of Largest Area (CLA). This method partitions a multiple peak fuzzy set into several disjoint fuzzy subsets, each of which corresponds to a feasible fuzzy command. The fuzzy subset with the largest area is then selected and defuzzified separately using the COA method. An example of this is shown in figure 67. Here, the fuzzy set consists of two subsets. The subset on the right has the largest area, so this is selected and defuzzified separately using the COA method. This results in a crisp value of 5°. This is a valid steering angle for the robot which will steer it safely around the obstacle on its left.

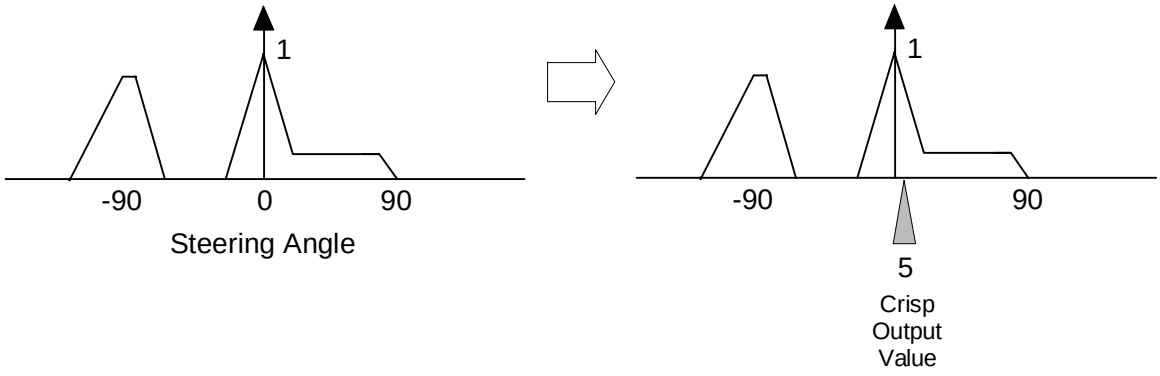


Figure 67 CLA Defuzzification

5.3.5 Implementation

This section describes the implementation of the fuzzy logic navigation controller. The controller has two behaviours - an obstacle avoidance behaviour and a target following behaviour. The input to the target following behaviour is the angle between the robot's current heading and the direction of the target. The obstacle avoidance behaviour gets its input from a group of 8 sonar sensors distributed equally around the robot. The controller consists of four modules - a fuzzification module, a fuzzy inference module, a command fusion module and a defuzzification module. Figure 68 shows a block diagram of the various modules.

The obstacle avoidance behaviour has eight input variables - one for each sonar sensor. Additionally, each variable has one function or term associated with it. This function is called NEAR and represents the degree of how close an obstacle is to the robot. The first task that the controller must perform is to fuzzify the input variables for the obstacle avoidance behaviour. This takes the values from each sonar sensor and determines the degree of membership for the function NEAR in each variable. The shapes of the membership functions NEAR are similar to each other. These shapes were chosen so as to reflect the threat an obstacle poses to the robot. For obstacles located near to the robot, the degree of membership of the function will be one, indicating that the obstacle is extremely close to the robot. For obstacles located further away, the degree of membership gradually

decreases to zero, indicating a safe condition. An example of the membership function NEAR for the forward looking sonar sensor is shown in figure 69.

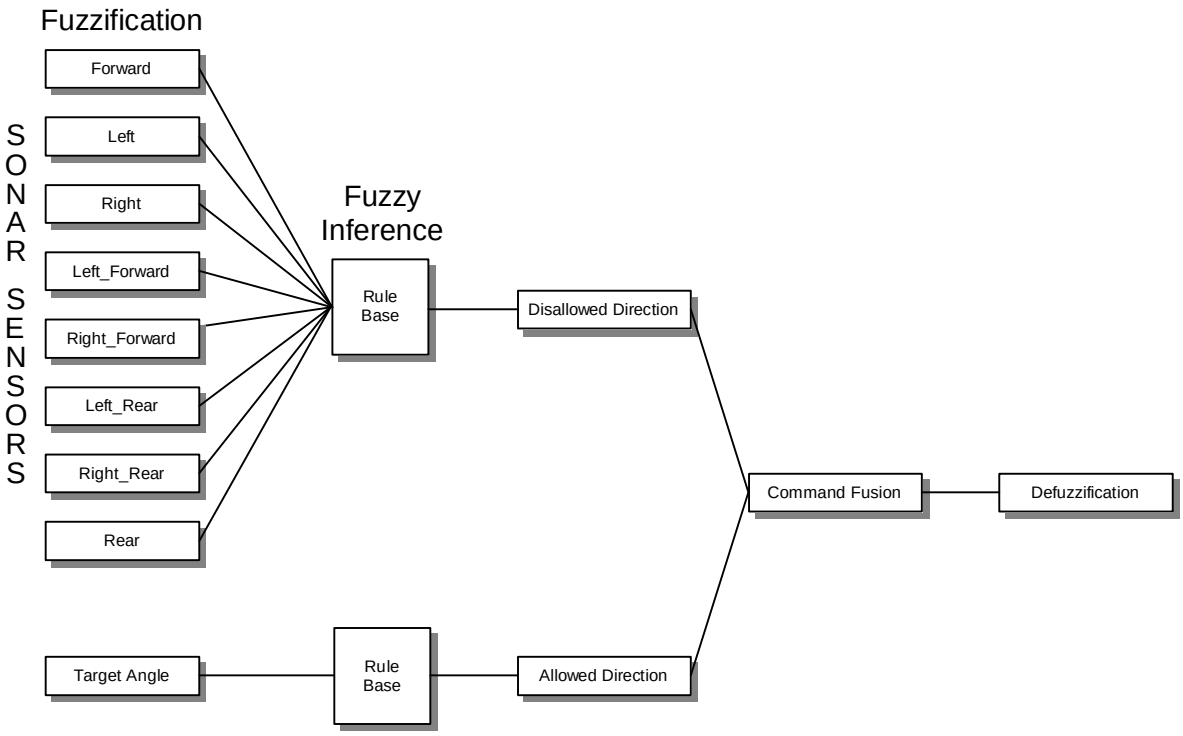


Figure 68 Fuzzy Controller Modules

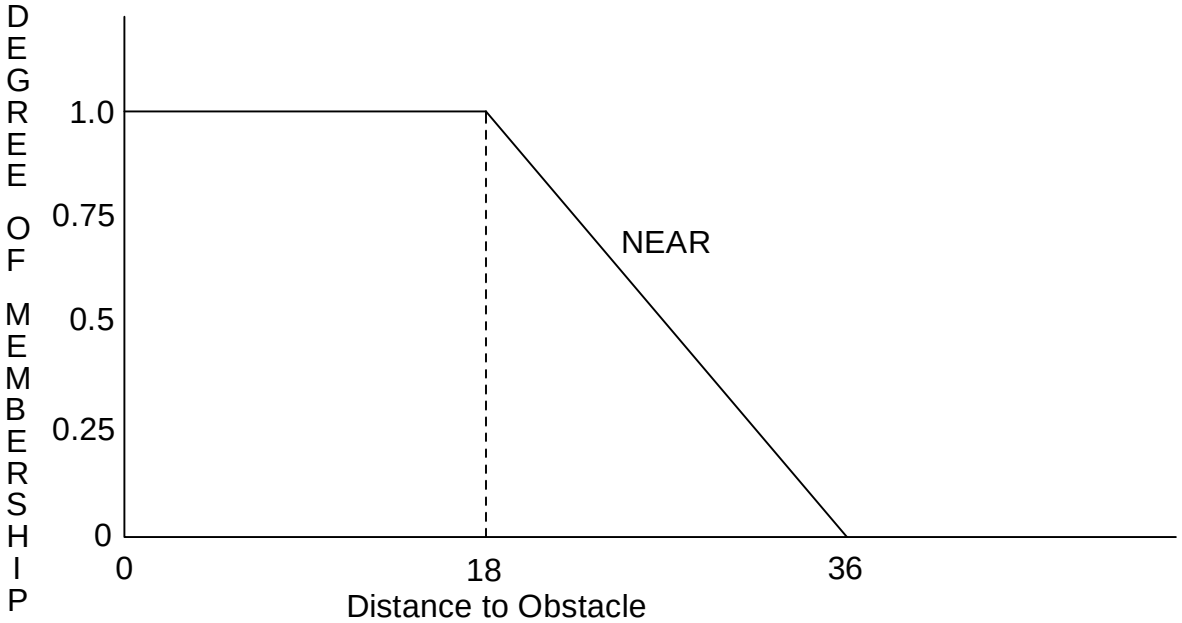


Figure 69 Membership Function Near for Forward Looking Sensor

It can be seem from this function that objects located less than 18 inches away are considered extremely close. Obstacles located between 18 inches and 36 inches away also pose a threat, but not to the same extent. Any obstacle located more than 36 inches away is unimportant.

Once all of the input variables for the obstacle avoidance behaviour have been fuzzified, the next task for the controller to perform is fuzzy inference. The fuzzy inference process calculates the truth value for the premise of each rule in the fuzzy rule base and applies this to the conclusion part of each rule. The output membership function for the rule is clipped off at a height corresponding to the rule premise's computed degree of truth. The following table shows the eight rules which are used.

If Forward Sensor Distance is Near **Then** Disallowed Direction is Forward
If Left Sensor Distance is Near **Then** Disallowed Direction is Left
If Right Sensor Distance is Near **Then** Disallowed Direction is Right
If Left_Forward Sensor Distance is Near **Then** Disallowed Direction is Left_Forward
If Right_Forward Sensor Distance is Near **Then** Disallowed Direction is Right_Forward
If Left_Rear Sensor Distance is Near **Then** Disallowed Direction is Left_Rear
If Right_Rear Sensor Distance is Near **Then** Disallowed Direction is Right_Rear
If Rear Sensor Distance is Near **Then** Disallowed Direction is Rear

Fuzzy inference consists of two components: aggregation and composition. Aggregation computes the IF part of a fuzzy rule while composition computes the THEN part. Each rule defines an action to be taken in the THEN part. The degree to which this action is valid is given by the truth value in the premise of the rule. Take the following rule as an example:

If Forward Sensor Distance is Near **Then** Disallowed Direction is Forward

If the forward looking sensor detects an obstacle located 27 inches away, then the degree of membership for the function Near will be 0.5. The truth value for the premise of the rule is therefore 0.5. In this situation the disallowed direction of travel is forward and the degree to which this action is considered valid is 0.5.

The obstacle avoidance behaviour has one output variable called Disallowed Direction. Within this variable, there are eight membership functions, each of which is trapezoidal in shape. Each membership function represents a certain disallowed direction of travel and is centred around the direction of one of the sonar sensors. An example of the membership function FORWARD is shown in figure 70.

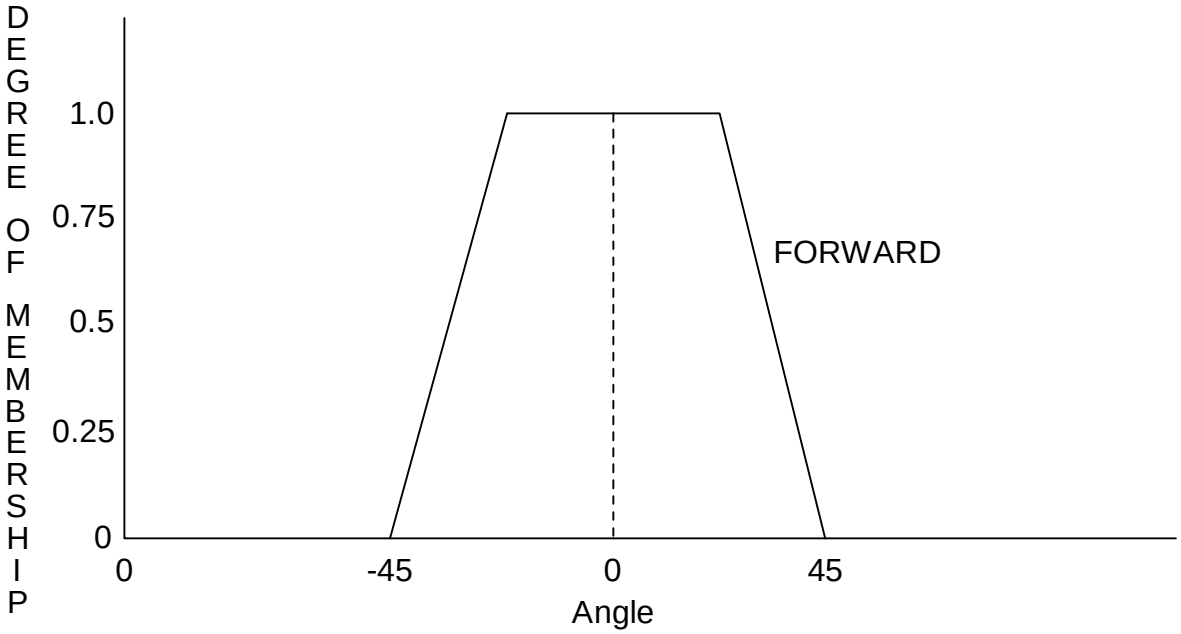


Figure 70 Membership Function Forward

This membership function represents the disallowed direction of travel for the forward looking sonar sensor. If the fuzzy controller considers forward to be a disallowed direction, then any angle between -22.5° and 22.5° will a degree of membership of 1. This indicates a highly disallowed direction. For angles between 22.5° and 45°, the degree of membership gradually decreases to 0. Directions within this region are also considered disallowed, but not to the same extent. All of the other seven membership functions are similar to this one. The table below shows the angle on which each function is centred around:

Membership Function	Centre Value
Forward	0
Left_Forward	-45
Right_Forward	45
Left	-90
Right	90
Left_Rear	-135
Right_Rear	135
Rear	180

The next task for the controller to perform is to fuzzify the input variable for the target following behaviour. This behaviour has only one input variable, which is called Target Angle. This represents the angle between the robot's current heading and the direction of the target. The variable has eight membership functions, which are all triangular in shape. Each membership function represents a certain target angle - for example, 0°, 45° etc. An example of the membership functions 0° and 45° are shown in figure 71. It is important to note that the membership functions are centred around the same points in which the sonar sensors are facing.

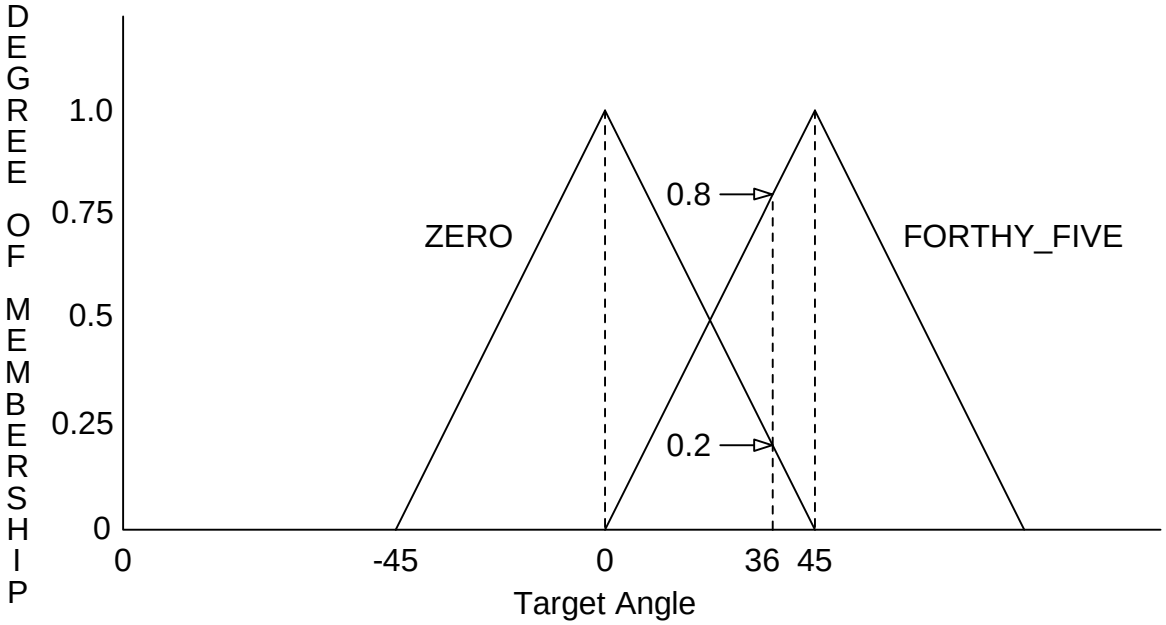


Figure 71 Membership Functions Zero and Forthy_Five

If the input to the target following behaviour happens to be an angle of 36°, then the degree of membership for the function 0° is 0.2 and the degree of membership for the function 45° is 0.8. In this case, the target angle is considered to be more in the 45° direction than in the 0° direction. All of the other six membership functions are similar to these two with the only difference being the centre value of each function. The table below shows each of these values:

Membership Function	Centre Value
0°	0
-45°	-45
45°	45
-90°	-90
90°	90
-135°	-135
135°	135
180°	180

Having fuzzified the input variable for the target following behaviour, the next task for the controller to perform is fuzzy inference. The following table shows the eight rules used by this behaviour.

If Target Angle is 0° **Then** Desired Direction is Forward
If Target Angle is -45° **Then** Desired Direction is Left
If Target Angle is 45° **Then** Desired Direction is Right
If Target Angle is -90° **Then** Desired Direction is Left_Forward
If Target Angle is 90° **Then** Desired Direction is Right_Forward
If Target Angle is -135° **Then** Desired Direction is Left_Rear
If Target Angle is 135° **Then** Desired Direction is Right_Rear
If Target Angle is 180° **Then** Desired Direction is Rear

The target following behaviour has one output variable called Desired Direction. Within this variable, there are eight membership functions, each of which is trapezoidal in shape.

Each membership function represents a certain desired direction of travel for the robot. This variable is an exact copy of the output variable for the obstacle avoidance behaviour and so won't be discussed here. Refer to page 127 for a description of this.

5.3.6 Development

A fuzzy logic development system has been designed in Visual C++ in order to develop and test the fuzzy logic navigation controller. This development package allows a new controller to be developed from scratch based upon a standard template. Initially, there are no rules or memberships functions for any of the behaviours. These must be added by the user. It is possible to add additional input variables for the obstacle avoidance behaviour. This may be done if additional sensors are added to the robot. The target following behaviour has just one input variable, which cannot be changed. In addition, each behaviour also has one output variable. These too cannot be changed. New membership functions can be added for each behaviour. A special membership function editor allows new functions to be added and allows the shape of the membership function to be changed. There is also a rulebase editor which allows new rules to be added and old ones to be modified.

To help test and debug the system, it is possible to simulate the fuzzy controller online. This can be done from two different perspectives. Firstly, it is possible to view the movement of the robot in a simulated environment. This gives an insight into how the robot may behave in a real world environment. During this simulation, the initial starting point for the robot and the finish point can be specified by the user. It is also possible to place various obstacles at different locations around the environment. Secondly, it is possible to view the output from the different modules in the controller. This allows the shape of the output fuzzy sets from each behaviour to be seen along with the fuzzy set generated by the command fusion module. Being able to see this information can help in fine tuning the controller and helps in solving problems which may arise.

5.3.6 Simulation Examples

The following screenshots show the paths followed by the robot during various simulated scenarios. In each case, the robot has a start position and a finish position. The robot travels along a smooth path in reaching its destination. If any obstacles are encountered, the robot will smoothly work its way around them.

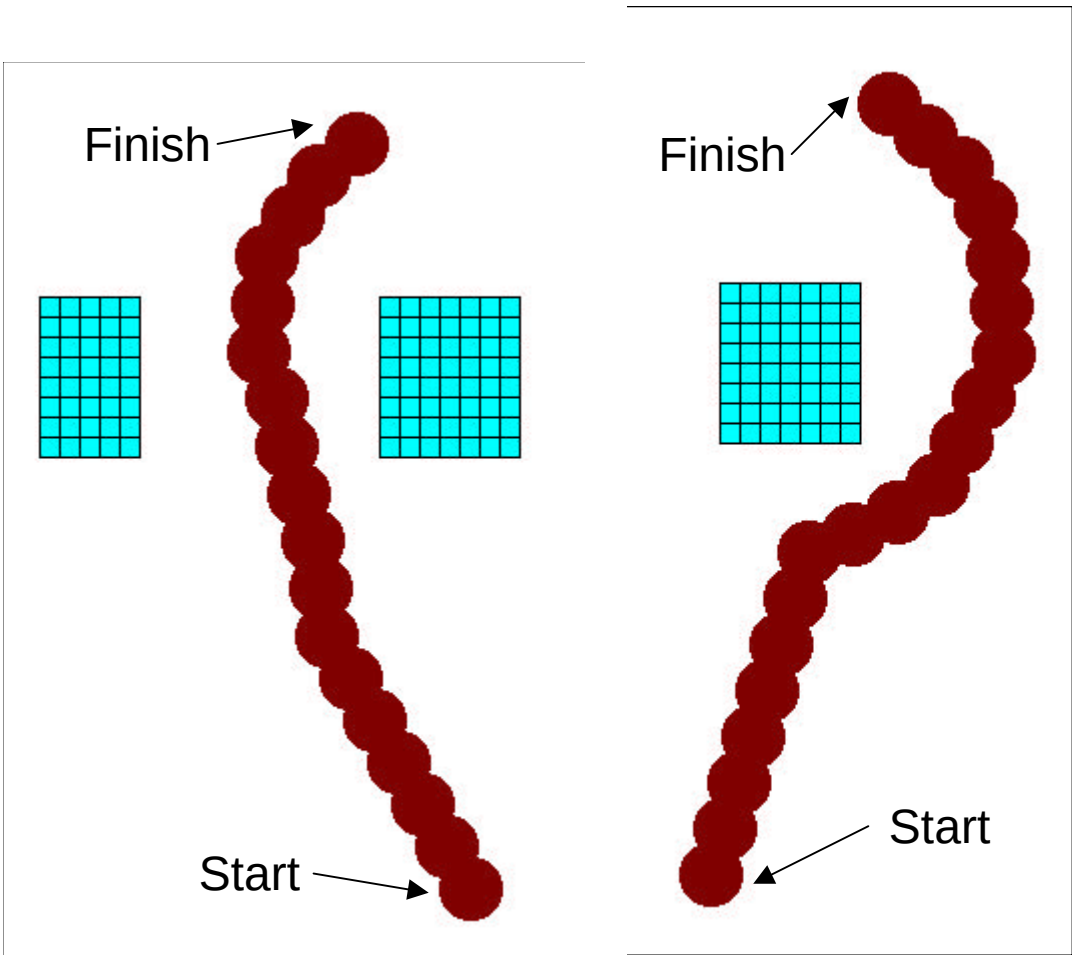


Figure 72 Simulation Example

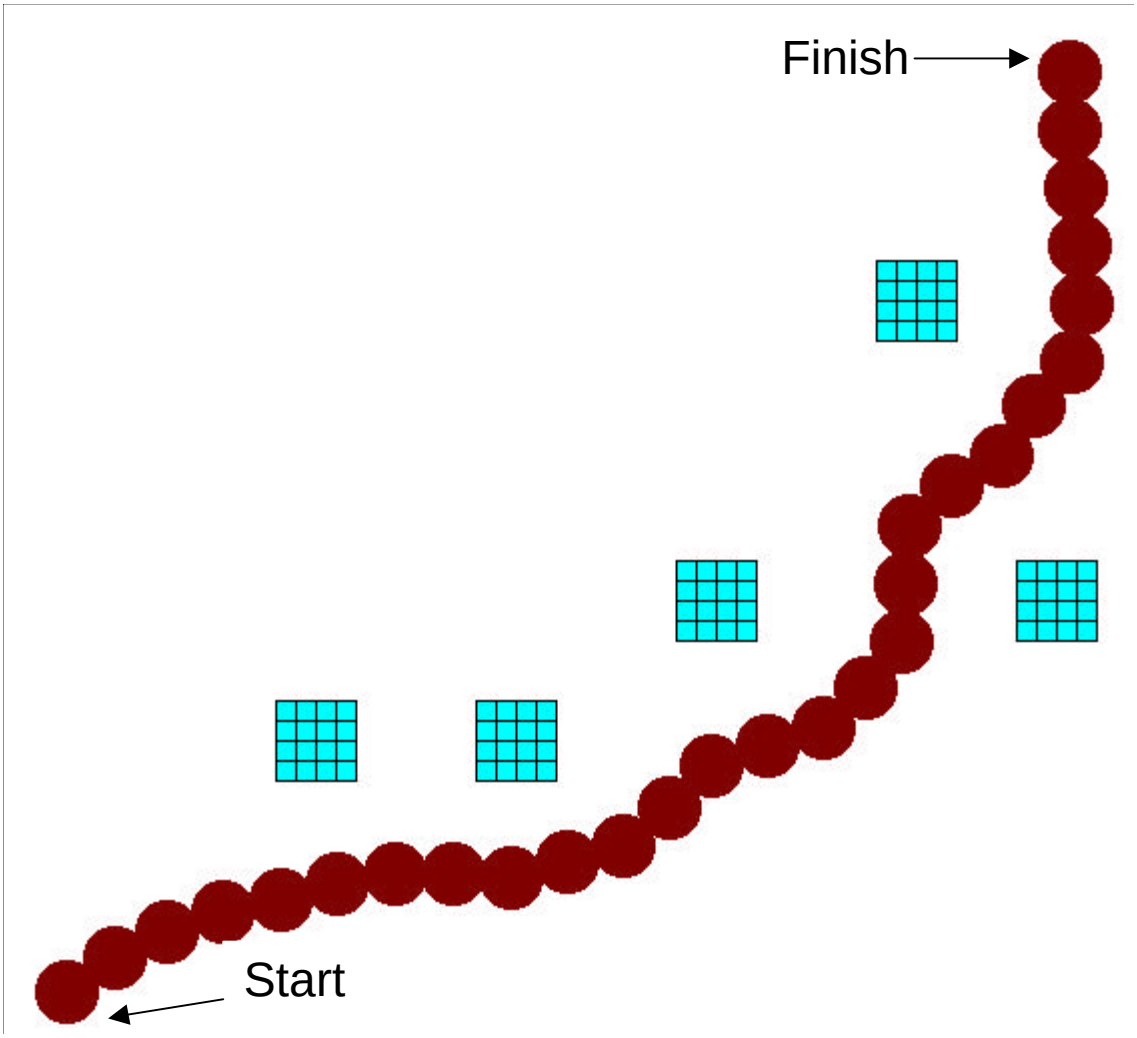


Figure 73 Simulation Example

5.4 Summary

This chapter has described the design of a fuzzy logic navigation system for a mobile robot. The advantage of using fuzzy logic for navigation is that it allows for the easy combination of various behaviours' outputs through a command fusion process. In subsumption, the fixed priority arbitration scheme only allows a single behaviour to be active at any one time. Implementing a navigation system using this basic system could result in a zigzag path being produced. By combining the output of two behaviours, a much smoother path can be produced. The navigation system has been tested in simulation and has shown promising results.

6. EXPERIMENTAL RESULTS

This chapter presents results from tests carried out with the robot. The experiments were designed to investigate how well the behaviours work individually and also how they interact with each other through the robot's environment to perform seemingly intelligent tasks.

6.1 Test Environment

The robot's test environment was a rectangular room approximately 3.5m x 2.5m in size. The room could be customised for performing different types of experiments. A number of straight edges were obtained to construct imaginary walls and corners to test the edge following and landmark detection behaviours.

6.2 Testing the Simple Low-Level behaviours

Behaviour based systems are modular making them easy to design, test and debug. When implementing a control architecture based on this system, lower levels of competence are added first, such as obstacle avoidance. Once this layer has been thoroughly tested and shown to exhibit the correct behaviour, further layers can be added, increasing the level of competence of the robot. The first experiments to be carried out were designed to test the implementation of the simple low-level behaviours. These behaviours were built to demonstrate that the robot could operate reactively based on sensor stimuli. The three behaviours consist of a cruise behaviour, an obstacle avoidance behaviour and a light following behaviour.

6.2.1 Cruise behaviour

The first behaviour to be tested was the cruise behaviour. This is an extremely simple behaviour whose sole purpose is to simply travel straight ahead. To test this behaviour, the robot was placed in the test environment. If it maintained a continuous forward motion, it could reasonably be concluded that the behaviour was performing as expected. As one would expect for such a simple behaviour, it performed flawlessly.

6.2.2 Obstacle Avoidance Behaviour

The next behaviour to be tested was the obstacle avoidance behaviour. This is a somewhat more complex behaviour. In conjunction with this behaviour, the control system also included the cruise behaviour at this stage so the robot could maintain a continuous forward motion in the absence of any obstacles. The first experiment to be carried out using this behaviour was performed in an empty environment with no obstacles present. The only features the robot would encounter that would hinder its progress were the walls around the perimeter of the room. Figure 74 shows an example of the path followed by the robot as it wandered about. It can be seen from the figure that the robot travels straight ahead under normal situations. Only when it is in close proximity to the walls does it turn. To carry out a second experiment for this behaviour, the test environment was prepared by locating a number of obstacles at random around the floor. Figure 75 shows the resulting path followed by the robot in this situation.

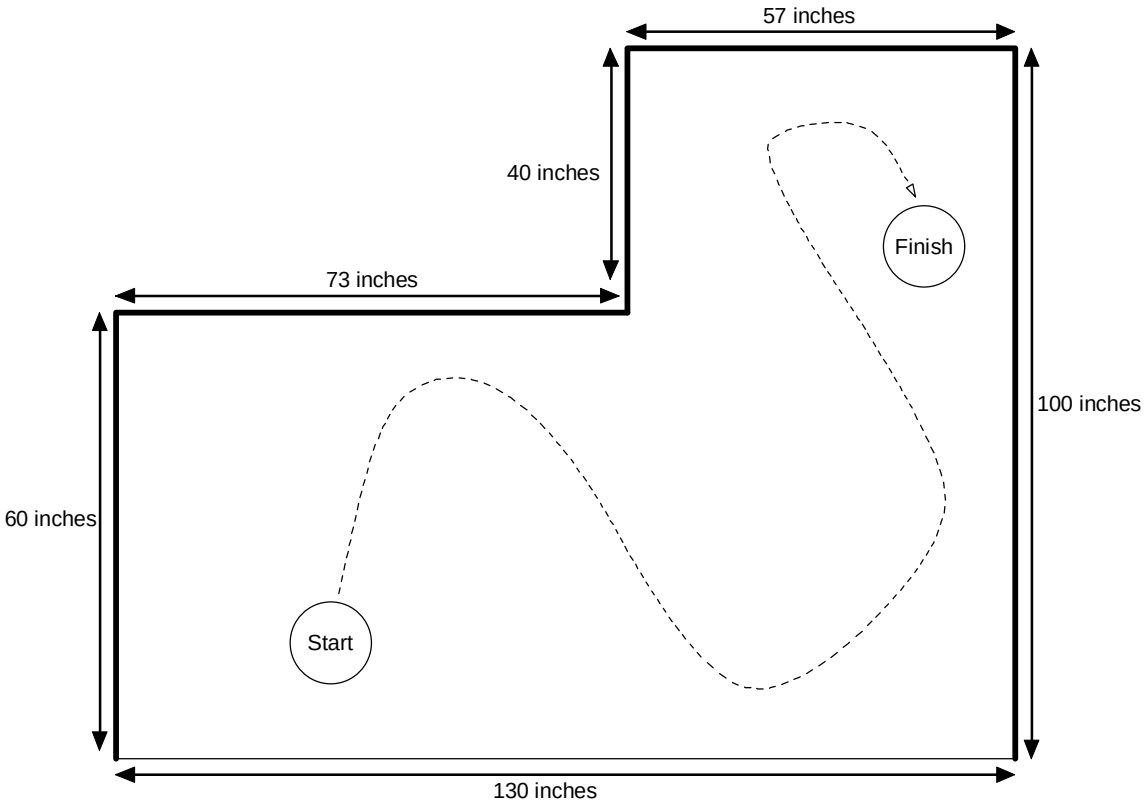


Figure 74 Path Followed by the Robot Under Obstacle Avoidance

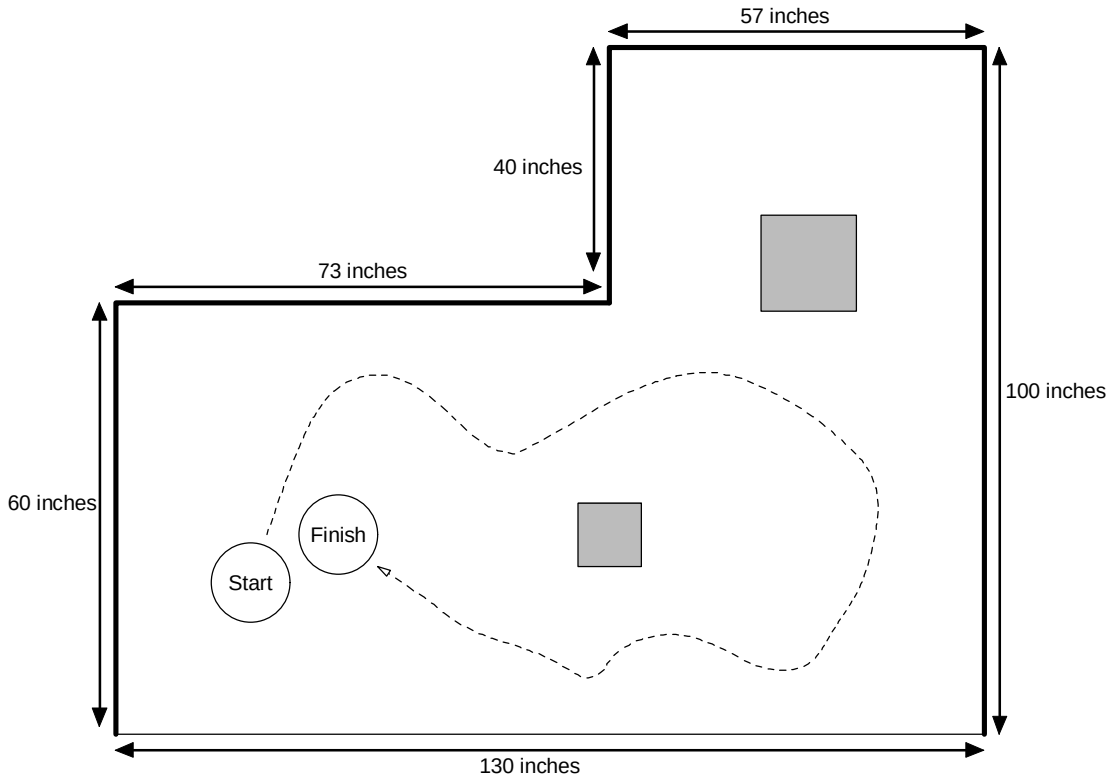


Figure 75 Path Followed by the Robot Under Obstacle Avoidance

6.2.3 Light Following Behaviour

This behaviour allows the robot to seek or follow a light source. In conjunction with this behaviour, the control system also included the cruise behaviour at this stage so the robot could maintain a continuous forward motion in the absence of any light source. The robot was initially placed in the test environment with no light sources present. Under this condition, the cruise behaviour would steer the robot straight ahead. To conduct the experiment, a light source was obtained. This was simply a normal handheld torch. As the robot traveled straight ahead under the influence of the cruise behaviour, the light source was shone into each of the LDRs. In all cases, the robot would attempt to steer in the direction of the brightest LDR. Figure 76 shows an example of the path followed by the robot. Each of the black dots represent the position where the LDRs where stimulated by the light.

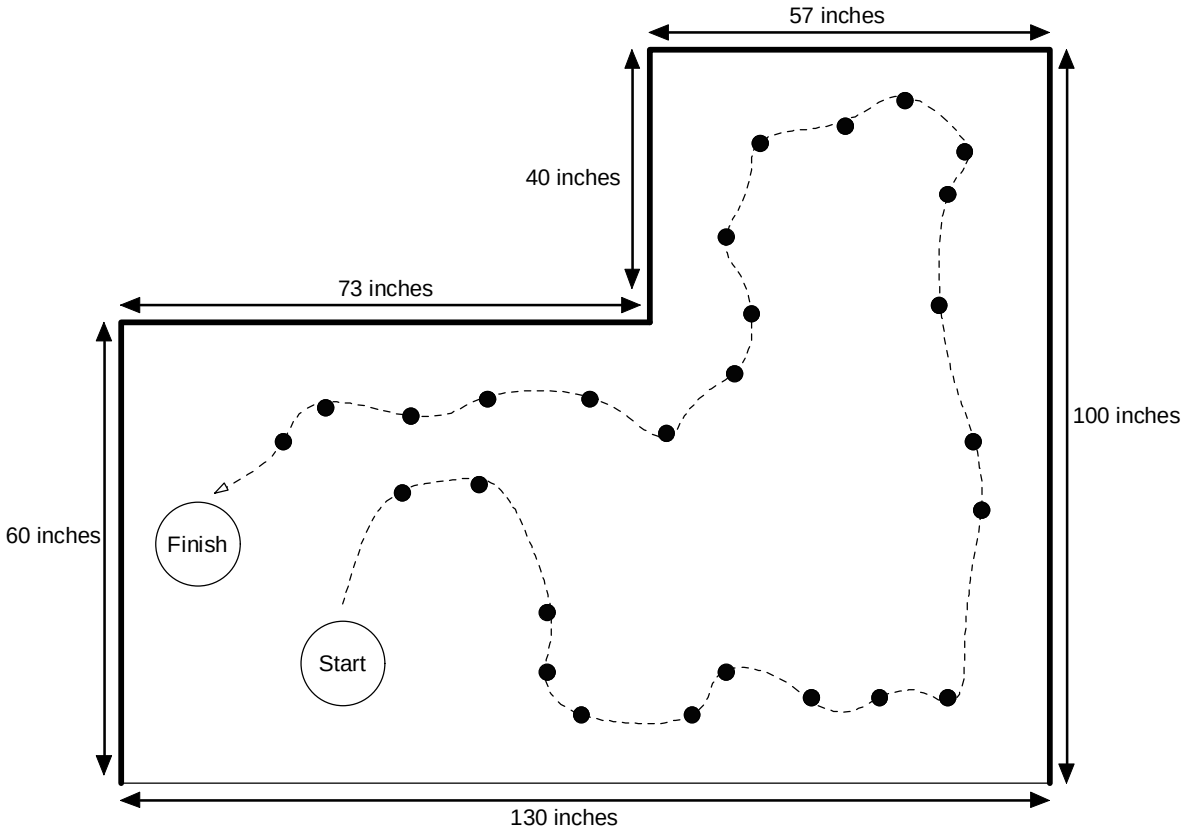


Figure 76 Path Followed by the Robot under Light Following

As another experiment, the obstacle avoidance behaviour was added to the control system increasing the robot's overall competence. During this experiment, the robot performed similar to the experiment described above. If an obstacle was detected, however, the obstacle avoidance behaviour would become active and steer the robot away from it. Since the obstacle avoidance behaviour has a higher priority than the light following behaviour, the robot would avoid obstacles irrespective of whether the LDRs were being stimulated by the light source.

6.3 Testing the Mapping and Navigation Behaviours

The implementation of the *obstacle avoidance*, *cruise* and *light following* behaviours illustrated the robot's capability to react directly to sensor stimuli, and perform seemingly intelligent tasks by interacting with each other through the robot's environment. Following the successful implementation and testing of these behaviours, the next set of behaviours which were developed and tested endow the robot with the ability to explore and map its environment. The robot can build up a map of a previously unknown environment and use it to navigate to certain locations. The first behaviours to be tested were the edge following behaviour and the landmark detection behaviours. It is important that these can operate reliably and robustly in order to ensure that an accurate map can be constructed.

6.3.1 Edge Following Behaviour

This behaviour causes the robot to follow a straight edge at a certain pre-defined distance. One of the assumptions that this behaviour makes is that the robot should initially be placed parallel to an edge before edge following can begin. To satisfy this condition the robot was placed parallel to a wall. The test environment for this experiment contained a number of long straight edges for the robot to follow. In initial experiments carried out with this behaviour, it was found that the range readings returned by the -45° sensor would often be incorrect by quite a large margin. This problem is due to the characteristic of a sonar beam when fired at an angle to a surface. The beam can often be reflected causing an incorrect distance to be obtained. Figure 77 shows an example of a path followed by the robot in this

situation. It can be seem that an oscillatory motion tends to result while following the edge. To overcome this problem, the angle of the sensor was changed to -60°. This has proven to be quite effective resulting in the robot following a smooth path along the edge. Figure 78 shows typical example of this.

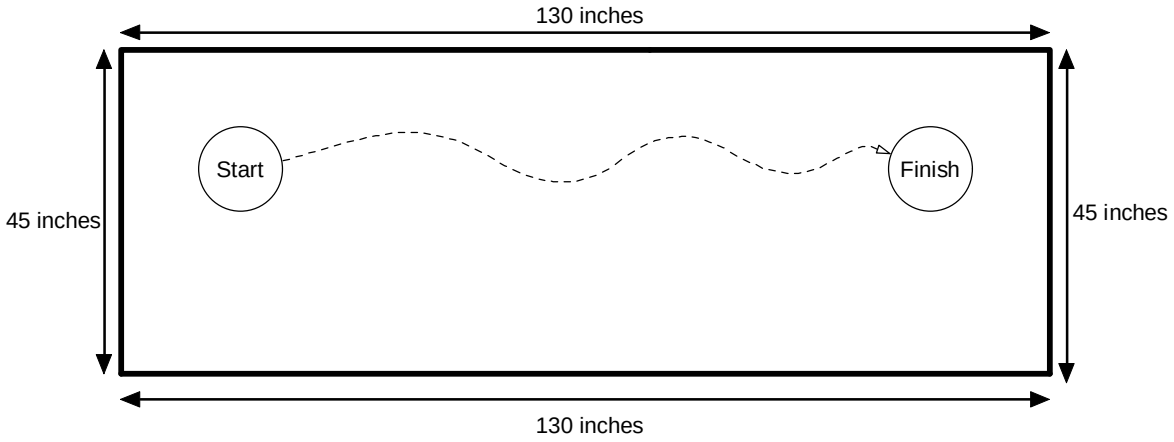


Figure 77 Edge Following

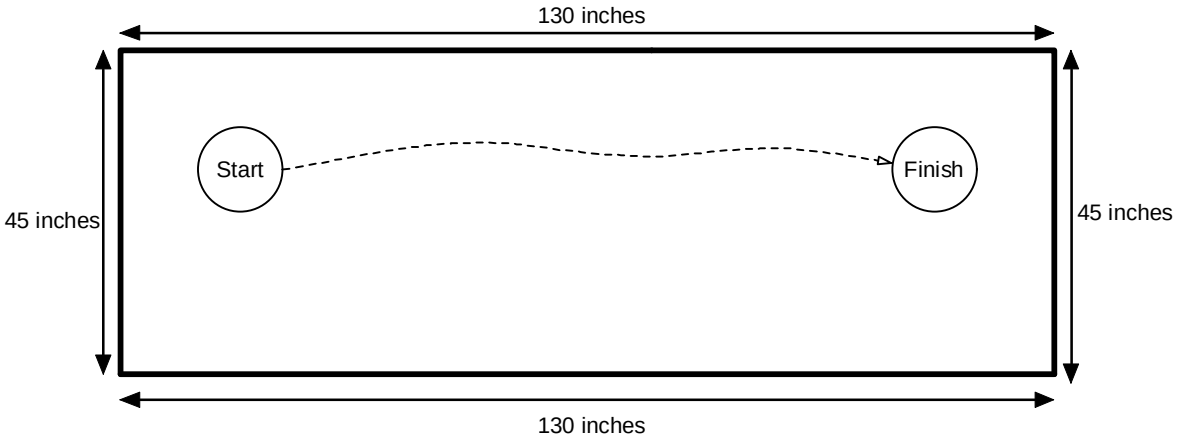


Figure 78 Edge Following

The edge following behaviour was tested for both left and right sided edges. In all cases, the results were the same.

6.3.2 Concave Corner and Convex Corner Behaviours

These behaviours watch out for specific types of landmarks in the robot's environment such as concave and convex corners. If such a feature is detected, the robot will attempt to turn. To test these behaviours, the environment was set up to include a number corners and a door. In general, it was found that the robot could successfully negotiate its way around a corner. Figure 79 shows an example of the path followed by the robot as it works its way around the environment. Whenever the robot detected a convex corner, it would initially travel straight ahead for a certain distance to determine if the feature detected was either a door or a corner. In the example shown, the robot does not find an edge at point A so it backtracks and turns to continue following the edge.

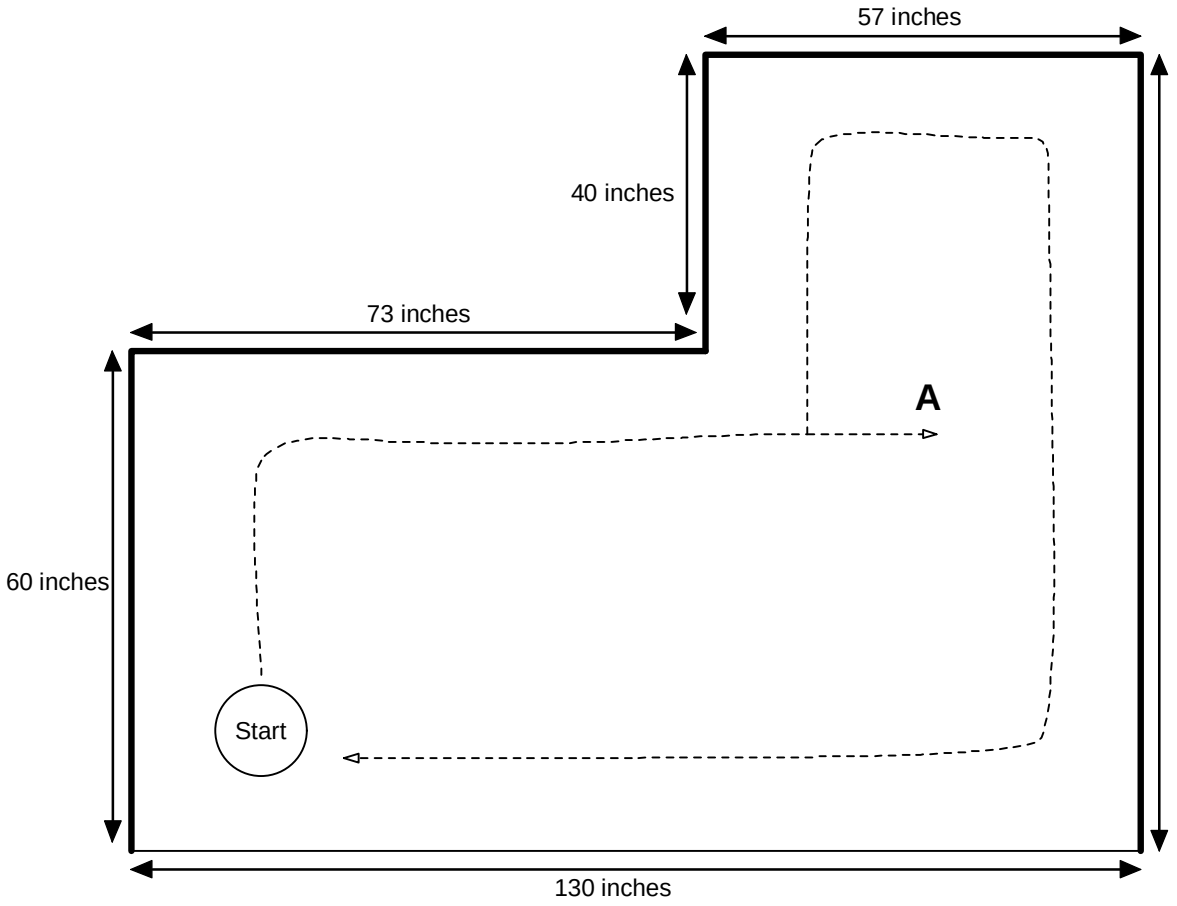


Figure 79 Detecting Corners

In certain situations, it was found that the concave corner behaviour was incapable of turning correctly at a corner. This was due to incorrect readings being returned from the sonar sensors. When this occurred, the corner could be detected a second time. In most cases, the robot would be capable of turning correctly this second time round. However, detecting a corner twice is an undesirable situation. It will decrease the accuracy of the map constructed by the Mapping behaviour. Fortunately, this situation didn't occur very often.

Figure 80 shows an example of an environment containing a door. Here the robot initially follows the straight edge leading to the door. When the door is detected by the convex corner behaviour, the robot will travel straight ahead for a certain distance to determine if this is either a door or a convex corner. At point A, an edge is detected again so this is treated as a door and the robot continues on.

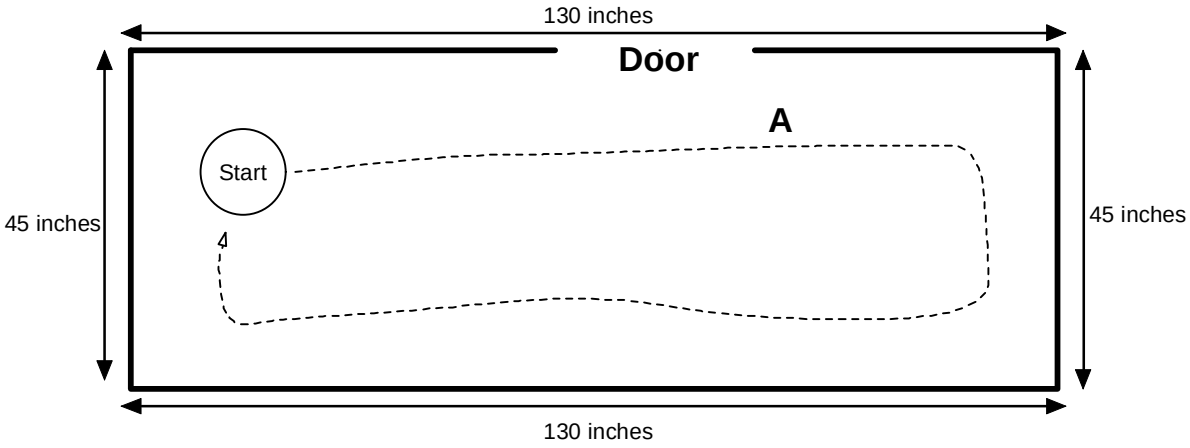


Figure 80 Detecting a Door

6.3.3 Mapping Behaviour

Testing this behaviour was quite straightforward. All that needs to be done is to compare the topological map generated by the behaviour with the actual features and dimensions of the environment. To carry out this experiment, the robot was allowed to map the environment in its entirety. Once a complete map was obtained, the robot would stop and remain stationary. The mapping behaviour was slightly modified for this experiment so it could write the complete map to file in a format that could be easily interpreted. The test

environment shown in figure 81 was used to carry out this experiment. Included in the figure are the node numbers associated with each landmark that the robot detects. The following table shows the nodes constructed for this environment. In this experiment, the first landmark to be detected by the robot is labeled 1 and the robot is travelling in a clockwise direction.

Landmark	1	2	3	4	5	6
Type of Landmark	Concave	Convex	Concave	Concave	Concave	Concave
Number of Next Node	2	3	4	5	6	1
Distance to Next Node	77	37	59	105	138	59
Number of Previous Node	6	1	2	3	4	5
Distance to Previous Node	59	77	37	59	105	138

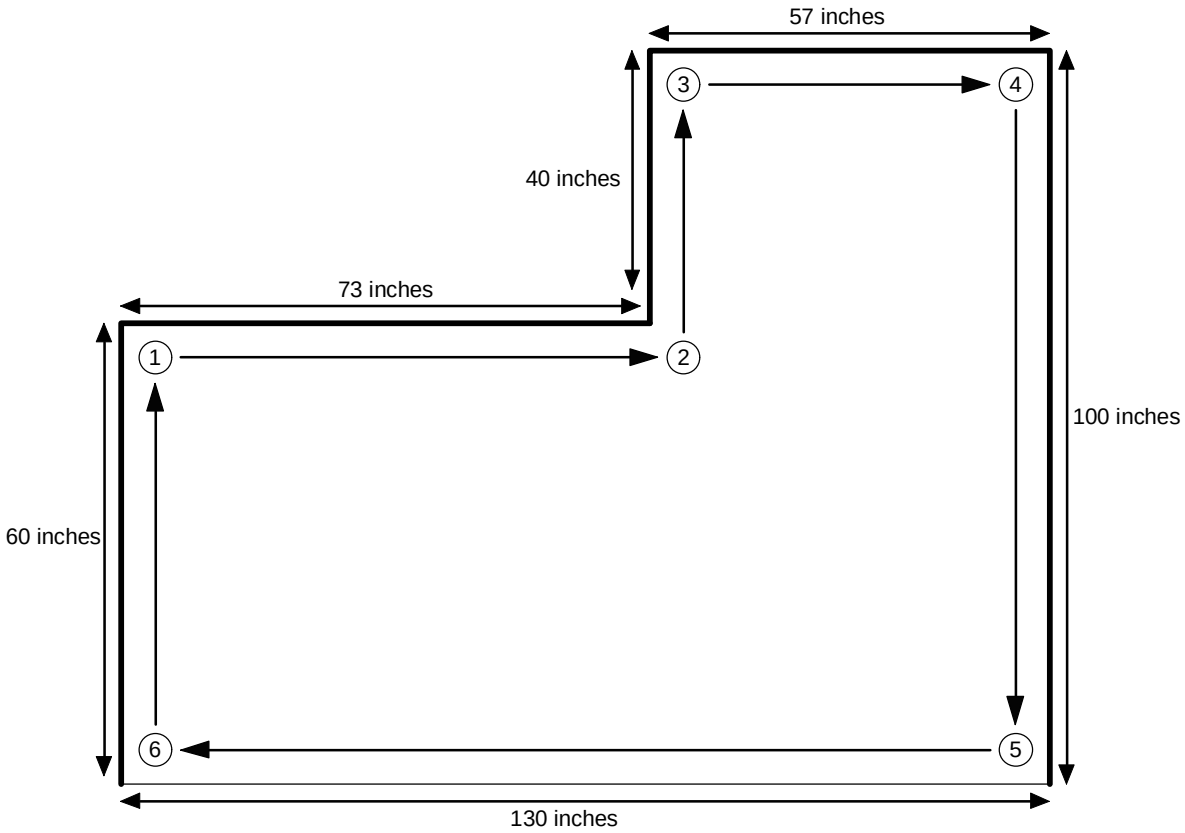


Figure 81 Test Environment for Mapping Behaviour

A number of other experiments were carried out in various types of environment. In all cases, the Mapping behaviour constructed a reasonable map of the environment. The

distance calculated between nodes will never exactly match the real environment values. Errors will always crop up either due to sonar sensor errors or odometry errors.

6.3.4 Search for Edge Behaviour

The purpose of this behaviour is to lock onto an edge and ensure that the robot is parallel to it. To test this behaviour, the robot was placed at random locations in the environment (see figure 82). From each of these locations, the robot would initially travel straight ahead until an edge was detected. It would then attempt to turn to be parallel with the edge. Depending on the robot's orientation to the edge when it arrives, it will lock onto it either from its left hand side or its right hand side.

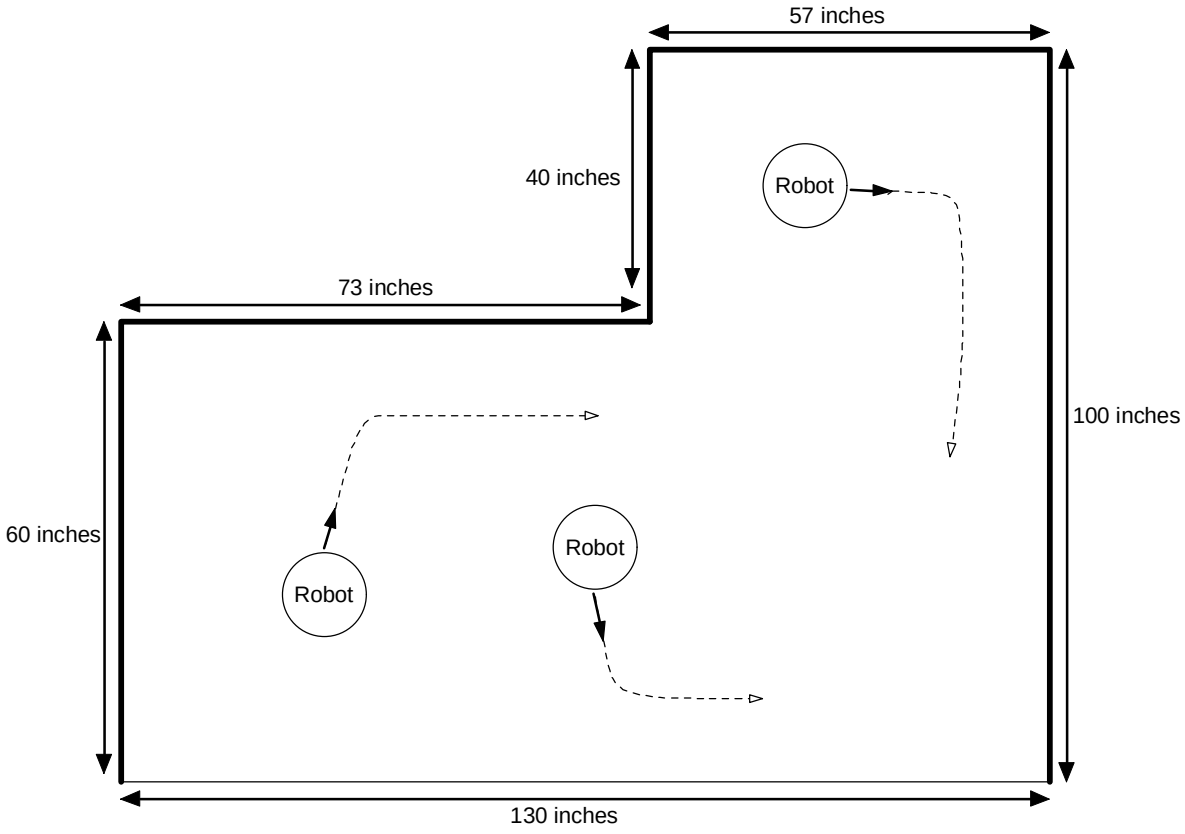


Figure 82 Testing the Search for Edge Behaviour

6.3.5 Localization and Navigation Behaviours

Rather than test each these behaviours individually, they were tested as a pair. The logic behind this is simple. In order for the navigation behaviour to be capable of travelling to specific locations in the environment, it must initially be aware of its starting position. The localization behaviour ensures that this requirement can be met. Before this experiment can be carried out, the Mapping behaviour must have previously constructed a map of the environment for the navigation and localization behaviours to use. For this experiment, the robot's initial starting point is some random location in the environment. When the robot is switched on, it will first lock onto an edge using the Search For Edge behaviour. The localization behaviour will then become active and attempt to determine the current position of the robot. Once the current position is known, the robot can navigation towards its destination. Both of these behaviours were tested in various types of environments. In each case, the destination was specified as a node to travel to. In all cases, the robot was successful in achieving its goal. Figure 83 shows an example of the path taken by the robot.

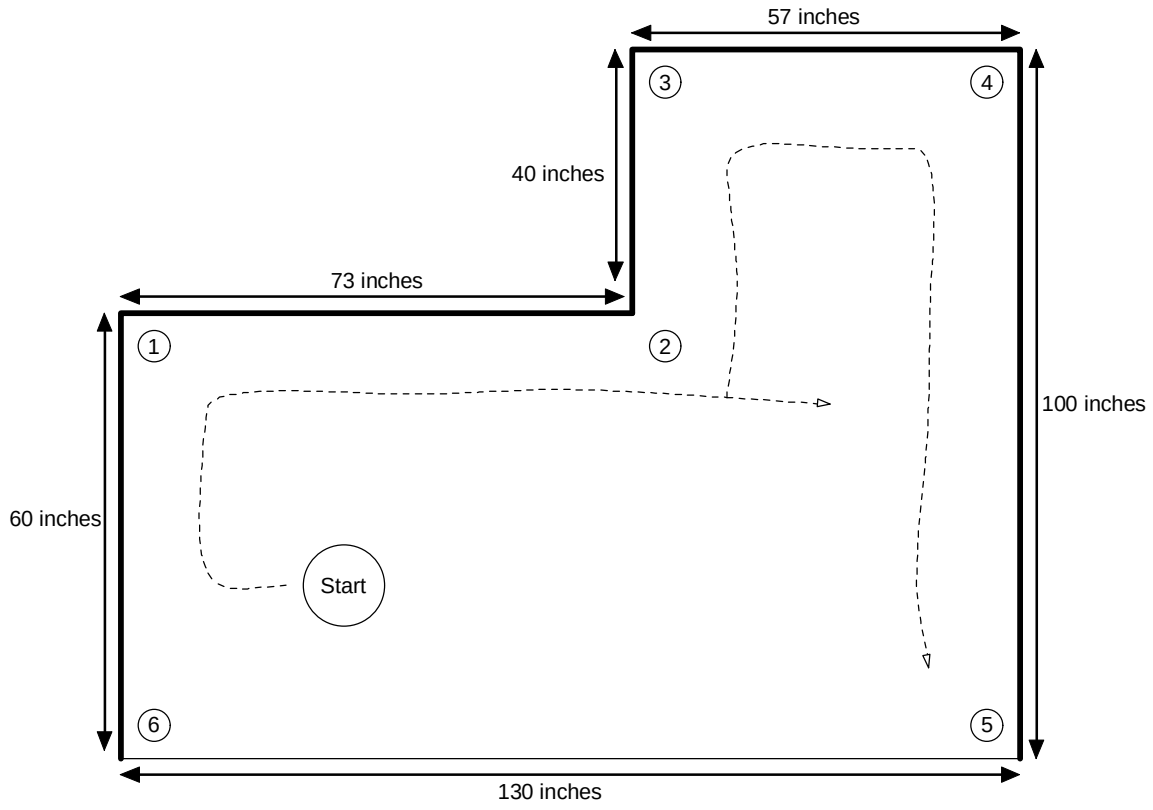


Figure 83 Testing the Localization and Navigation Behaviours

In this example, the robot's starting position is as shown and its destination is node 5. The robot locks onto the wall and follows it around the environment. By the time it reaches node 3, its current position is known. It can then navigate all the way to the destination where it stops and remains stationary.

7. CONCLUSIONS

This chapter discusses the work presented in this thesis and comments on how effectively it met its aims. Future work to enhance and extend the functionality of the robot will also be discussed.

7.1 Discussion

This thesis has described the design of an autonomous mobile robot built as a testbed for behaviour based control and experimentation. By building a real working robot, experiments can be carried out in real world giving a true insight into how the robot behaves. The physical design of the robot was chosen to allow it to operate effectively in most types of environments. This was achieved by using a cylindrical design and using a differential drive system for motion control. The advantage of such a design is that the robot cannot be caught in tight corners since the differential drive system allows it to turn on the spot. From the experiments carried out, this has proven to be quite effective. A modular hardware control architecture has been implemented on the robot which distributes the workload and offloads sensor data stream processing to a dedicated processors.

The design of the robot has been heavily influenced by the need for real-time sensing and decision making in order to operate in dynamic and unstructured environments. This has been achieved by adopting a behaviour based control architecture and tightly coupling behaviours with sensor inputs with actuator outputs in a reactive way. The robot's control system is based on the standard subsumption architecture. An initial set of behaviours were developed and tested to demonstrate that the robot could operate reactively based on sensor stimuli. It was shown that the interaction of these behaviours with each other through the robot's environment resulted in seemingly intelligent tasks being performed.

Following the successful implementation of these behaviours, a modified form of a behaviour based control architecture was developed for the robot. This is based on the standard subsumption architecture with the addition of a concept known as a blackboard.

The blackboard acts as a central data repository where behaviours can deposit and extract information to help them go about their tasks. To test this control system, a set of behaviours were developed to allow the robot to successfully explore, map, and navigate around its environment. In situations when the robot becomes lost or is unaware of its position, it can perform localisation to re-establish its correct position in a relative fashion. Experiments carried out with the robot in a test environment have shown that the system can operate robustly. Some problems were encountered, however, with sonar sensors. Occasionally at a concave corner, the robot could have difficulty in turning correctly. It was also found that during edge following, one of the robot's sonar sensors could give incorrect readings. This problem was overcome, however, by changing the angle of the sensor.

The design of a fuzzy logic navigation system has also been presented in this thesis. Such a design overcomes one of the main limitations of the subsumption architecture. Rather than have a fixed priority arbitration scheme that only allows a single behaviour to be active at any one time, the fuzzy logic controller combines the output of two behaviours. This results in much smoother paths being produced. Tests carried out in simulation have shown this to be the case.

7.2 Future Work

Due to the highly distributed hardware control architecture used on the robot, it is easy to enhance the system by the addition of further modules. A vision system could be implemented by the addition of a camera and extra control circuitry. The high speed and processing power of the main central controller will allow the implementation of relatively sophisticated vision control algorithms. In addition, a collision ring could also be added to the robot. This would take the form of a ring of tactile sensors located around the perimeter of the robot. The purpose of this would be to serve as a failsafe or a backup in the event that the sonar sensors fail to detect an object. A more complex set of landmark detection behaviours could be implemented. For example, behaviours could be implemented to detect corridors and intersections. Also, a more complex mapping system could be introduced. With the present system, each node in the map can contain at most two links to other landmarks. It would be desirable if this could be extended to any number. It would also be

desirable if the robot could dynamically update the map in real time to take account of changing environments.

REFERENCES

[1] Albus, J. S., Barbera, A.J. and Nagel, R.N., 1981. "Theory and Practice of Hierarchical Control" Proceedings of the 23rd IEEE Computer Society International Conference, 18-27.

[2] Arkin, R. C., 1986. "Path Planning for a Vision-Based Autonomous Robot" Proceedings of the SPIE Conference on Mobile Robots, 240-249.

[3] Arkin, R. C., 1989. "Motor Schema-Based Mobile Robot Navigation" International Journal of Robotics Research, 92-112.

[4] Arkin, R. C., 1998. Behaviour-Based Robotics. MIT Press.

[5] Borenstein, J., 1991. "The Vector Field Histogram - Fast Obstacle Avoidance for Mobile Robots" IEEE Transactions on Robotics and Automation, 278-288.

[6] Brooks, R. A., 1986. “A Robust Layered Control System for a Mobile Robot” IEEE Journal of Robotics and Automation, Vol. RA-2, No. 1, 14-23.

[7] Brooks, R. A., 1989. “A Robot That Walks: Emergent Behaviour From a Carefully Evolved Network”, Neural Computation, 253-262.

[8] Brooks, R. A., 1990. “Elephants Don’t Play Chess” Robotics and Autonomous Systems 6, 3-15.

[9] Brooks, R. A., Connell, J. H. and Ning, P., 1987. "Herbert: A second Generation Mobile Robot" MIT AI Memo 984.

[10] Connell, J. H., 1987. "Creature Building with the Subsumption Architecture" IJCAI-87, Milan, 1124-1126.

[11] Donnett, J. G., 1992. Analysis and Synthesis in the Design of Locomotor and Spatial Competences for a Multisensory Mobile Robot. PhD Dissertation, University of Edinburgh.

[12] Elfes, A., 1989. "Using Occupancy Grids for Mobile Robot Perception and Navigation" Computer, 46-57.

[13] Flynn, A. M., Brooks, R. A., Wells, W. M. and Barrett, D. S., 1989. "The World's Largest One Cubic Inch Robot" Proceedings IEEE Micro Electro Mechanical Systems, Salt Lake City, 98-101.

[14] Gat, E., 1991. Reliable Goal-Directed Reactive Control of Autonomous Mobile Robots. PhD Dissertation, Virginia Polytechnic Institute and State University.

[15] Kortenkamp, D., Bonasso, R. P. and Murphy, R., 1998. Artificial Intelligence and Mobile Robots. MIT Press.

[16] Kurz, A., 1996. "Constructing Maps for Mobile Robot Navigaion Based on Ultrasonic Range Data" IEEE Transactions on Systems, Man and Cybernetics, Vol. 26, No. 2, 233-242.

[17] Lee, D., 1996. The Map-Building and Exploration Strategies of a Simple Sonar-Equipped Mobile Robot. Cambridge University Press.

[18] Leyden, M., Toal, D., Flanagan, C., 1999. "A Fuzzy Logic Based Navigation System for a Mobile Robot" Proceedings of 2nd Wismar Symposium on Automatic Control, Germany.

[19] Leyden, M., Toal, D., Flanagan, C., 2000 a. "Pitfalls of Simulation for Mobile Robot Controller Development" MIM 2000, Greece.

[20] Leyden, M., Toal, D., Flanagan, C., 2000 b. "An Autonomous Mobile Robot Built to Investigate Behaviour Based Control" Mechatronics 2000, Atlanta.

[21] Mataric, M. J., 1989. "Qualitative Sonar Based Environment Learning for Mobile Robots." SPIE Mobile Robots, Philadelphia.

[22] Mataric, M. J., 1990. "Navigating With a Rat Brain: A Neurobiologically-Inspired Model for Robot Spatial Representation" Proceedings of Simulation of Adaptive Behaviour, 169-175.

[23] Moravec, H. P., 1988. "Sensor Fusion in Certainty Grids for Mobile Robots" AI Magazine, 61-74.

[24] Nehmzow, U. and Smither, T., 1991. "Using Motor Actions for Location Recognition" Proceeding of the First Conference on Artificial Life, 96-104.

[25] Nehmzow, U., 2000. Mobile Robotics: A Practical Introduction. Springer

[26] Nilsson, N. J., 1969 "A Mobile Automation: An Application of AI Techniques" Proceedings of the First International Joint Conference on Artificial Intelligence, 509-520.

[27] Orlando, N. E., 1984. "An Intelligent Robotics Control Scheme" American Control Conference.

[28] Payton, D., Rosenblatt, J. and Keirsey, D., 1990. “Plan Guided Reaction” IEEE Transactions on Systems, Man and Cybernetics, Vol. 20, No. 6, 1370-1382.

[29] Saffiotti, A., 1997. "Handling Uncertainty in Control of Autonomous Robots" Technical Report TR/IRIDIA/97-8.

[30] Toal, D., Flanagan, C., Jones, C. and Strunz, B., 1996. “Subsumption Architecture for the Control of Robot” IMC-13, Limerick.

[31] Zimmer, U., 1996. "Robust World Modelling and Navigation in a Real World" Neurocompputing, Vol. 13, No. 2-4.

APPENDIX 1

Robot Specifications

Weight	3 kg
Height	30 cm
Width	30 cm
Maximum Speed	0.5 m/s
Power Source	12V Lead Acid Battery or External Power Supply
Motion	2 12V DC Motors
Motion Feedback	2 Incremental Shaft Encoders
Drive System	Differential
Main Controller	586 CPU 8 MB RAM 2 MB Solid State Hard Disk
Sensors	8 Sonar Sensors

APPENDIX 2

Motor Data

The motors used on the robot were 12V DC Motors from Maxon. The table below lists some of the important operating parameters for them.

Assigned power rating	4 Watts
Nominal voltage	12V
No load speed	4090 rpm
Stall torque	31.9 mNm
Speed/torque gradient	129 rpm/mNm
No load current	12.3 mA
Starting Current	1150 mA
Terminal Resistance	10.4 Ω
Max. permissible speed	6400 rpm
Max. continuous current	493mA
Max. continuous torque	13.66 mNm
Max. power output at nominal voltage	3410 mW
Max. efficiency	81%
Torque constant	27.7 mNm/A
Speed constant	345 rpm/V
Rotor inertia	24.8 g/cm^2

APPENDIX 3

Sonar Software Listing

```
/*************************************************************************
*                                                                        *
*                        Sonar  System Control Software                  *
*                                Version 1.0                             *
*                                                                        *
*                             Written By Mark Leyden                     *
*                                20 August 1999                          *
*                                                                        *
*************************************************************************/

#include <at89x52.h>
#include <stdio.h>
#include <math.h>

#define Baud_300  0xA0
#define Baud_600  0xD0
#define Baud_1200 0xE8
#define Baud_2400 0xF4
#define Baud_4800 0xFA
#define Baud_9600 0xFD

/* First Sensor Group */
#define SENSOR_0    P2_0
#define SENSOR_45   P0_6
#define SENSOR_90   P0_7

/* Second Sensor Group */
#define SENSOR_135  P2_1
#define SENSOR_180  P2_2
#define SENSOR_225  P2_3

/* Third Sensor Group */
#define SENSOR_270  P2_4
```

```
#define SENSOR_315  P2_5
#define SENSOR_360  P2_6

/* Initiate signals for the ultrasonic ranging boards */
#define INIT1  P0_0
#define INIT2  P0_2
#define INIT3  P0_1
/* Echo signals from the ultrasonic ranging boards */
#define ECHO1  P1_0
#define ECHO2  P1_2
#define ECHO3  P1_1

/* Blank Inhibit signals for the ultrasonic ranging boards */
#define BINH1  P0_3
#define BINH2  P0_5
#define BINH3  P0_4

#define true   1
#define false  0
#define BOOL   unsigned int

void main(void)
{
  unsigned int blanking_time;
  unsigned int delay_time;
  unsigned int echo_time;

  /* These store the echo times returned from each sensor */
  unsigned int sensor_echo1_time[3];
  unsigned int sensor_echo2_time[3];
  unsigned int sensor_echo3_time[3];

  unsigned int firing_sensor_group;   /* Specifies which group is currenty active */
  BOOL ECHO1_RECEIVED;          /* Indicate if echoes have been received */
  BOOL ECHO2_RECEIVED;
  BOOL ECHO3_RECEIVED;

  unsigned int distance_0;
  unsigned int distance_45;
  unsigned int distance_90;
  unsigned int distance_135;
  unsigned int distance_180;
  unsigned int distance_225;
  unsigned int distance_270;
  unsigned int distance_315;
  unsigned int distance_360;

  /* Set up serial communications */
  SCON = 0x52;
```

```
TMOD = 0x22;
TCON = 0xC0;


TH1 = Baud_2400;
TR1 = 1;  /* Enables timer 1 */
TI = 1;
/* Set up timer zero */
TMOD &= 0xF0;      /* Sets timer 0 up in mode 1 */
TMOD |= 1;         /* 16-bit timer incrementing every machine cycle  */

firing_sensor_group = 1;

/* Activate the sensors for the first firing group */
SENSOR_0 = 1;
SENSOR_45 = 0;
SENSOR_90 = 0;

SENSOR_135 = 1;
SENSOR_180 = 0;
SENSOR_225 = 0;

SENSOR_270 = 1;
SENSOR_315 = 0;
SENSOR_360 = 0;

INIT1 = 0;
INIT2 = 0;
INIT3 = 0;
ECHO1 = 1; /* Writing a one to these pins makes them an input */
ECHO2 = 1;
ECHO3 = 1;
BINH1 = 0;
BINH2 = 0;
BINH3 = 0;

TL0 = 0;
TH0 = 0;
TR0 = 1;
delay_time = 0;

/* Provides a 5 ms delay */
while (delay_time < 4608)
{
  delay_time = (int)TH0;   /* Read in timer's counter and convert to an integer */
  delay_time <<= 8;
  delay_time |= (int)TL0;
}
```

```
TR0 = 0;

while (1)
{
 for (firing_sensor_group = 1; firing_sensor_group < 4; firing_sensor_group++)
 {
  ECHO1_RECEIVED = false;
  ECHO2_RECEIVED = false;
  ECHO3_RECEIVED = false;


  /* The following statements check which firing group is currently active, and then
     select the appropiate sensors for that particular group. This is accomplished by
     writing a logic one to the gate of each sensor's mosfet. */

  if (firing_sensor_group == 1)
  {
   SENSOR_0   = 1; SENSOR_45  = 0; SENSOR_90  = 0;
   SENSOR_135 = 1; SENSOR_180 = 0; SENSOR_225 = 0;
   SENSOR_270 = 1; SENSOR_315 = 0; SENSOR_360 = 0;
  }

  else if (firing_sensor_group == 2)
  {
   SENSOR_0   = 0; SENSOR_45  = 1; SENSOR_90  = 0;
   SENSOR_135 = 0; SENSOR_180 = 1; SENSOR_225 = 0;
   SENSOR_270 = 0; SENSOR_315 = 1; SENSOR_360 = 0;
  }

  else if (firing_sensor_group == 3)
  {
   SENSOR_0   = 0; SENSOR_45  = 0; SENSOR_90  = 1;
   SENSOR_135 = 0; SENSOR_180 = 0; SENSOR_225 = 1;
   SENSOR_270 = 0; SENSOR_315 = 0; SENSOR_360 = 1;
  }

  /* Initiate each ultrasonic ranging board */
  INIT1 = 1;
  INIT2 = 1;
  INIT3 = 1;
  TL0 = 0;
  TH0 = 0;
  TR0 = 1;
  blanking_time = 0;

  /* In order to prevent ringing of the transducers from being detected as a return signal,
     the receive input of the ultrasonic ranging board's control IC is inhibited by internal
     blanking for 2.38 ms after the initiate signal. With this blanking time, only distances
     of 1.33 ft and over can be measured. To detect objects closer than this, the internal
```

```
   blanking time must be reduced. This is achived by taking the BINH (blank inhibit)
   input high. By taking this input high after 0.45 ms, distances of as little as 7 to 8
   inches can be measured. */

while (blanking_time < 830)   /* Provides a 0.45 ms delay */
{
 blanking_time = (int)TH0;   /* Read in timer's counter and convert to an integer */
 blanking_time <<= 8;
 blanking_time |= (int)TL0;
}

BINH1 = 1;
BINH2 = 1;
BINH3 = 1;

while (ECHO1_RECEIVED == false || ECHO2_RECEIVED == false ||
        ECHO3_RECEIVED == false)
{
 if (ECHO1 == 1 && ECHO1_RECEIVED == false)
 {
   echo_time = (int)TH0;   /* Read in timer's counter and convert to an integer */
   echo_time <<= 8;
   echo_time |= (int)TL0;
   sensor_echo1_time[firing_sensor_group - 1] = echo_time;
   ECHO1_RECEIVED = true;
 }

 if (ECHO2 == 1 && ECHO2_RECEIVED == false)
 {
   echo_time = (int)TH0;
   echo_time <<= 8;
   echo_time |= (int)TL0;
   sensor_echo2_time[firing_sensor_group - 1] = echo_time;
   ECHO2_RECEIVED = true;
 }

 if (ECHO3 == 1 && ECHO3_RECEIVED == false)
 {
   echo_time = (int)TH0;   /* Read in timer's counter and convert to an integer */
   echo_time <<= 8;
   echo_time |= (int)TL0;
   sensor_echo3_time[firing_sensor_group - 1] = echo_time;
   ECHO3_RECEIVED = true;
 }
}

delay_time = 0;
while (delay_time < 46080)
{
```

```
      delay_time = (int)TH0;   /* Read in timer's counter and convert to an integer */
      delay_time <<= 8;
      delay_time |= (int)TL0;
    }

    TR0 = 0;
    TL0 = 0;
    TH0 = 0;
    TR0 = 1;
    delay_time = 0;

    while (delay_time < 46080)
    {
      delay_time = (int)TH0;   /* Read in timer's counter and convert to an integer */
      delay_time <<= 8;
      delay_time |= (int)TL0;
    }

    TR0 = 0;    /* Switch timer off */

    INIT1 = 0;
    INIT2 = 0;
    INIT3 = 0;
    BINH1 = 0;
    BINH2 = 0;
    BINH3 = 0;
  }

  distance_0 = (sensor_echo1_time[0] * 13.02) / 1800;
  distance_45 = (sensor_echo1_time[1] * 13.02) / 1800;
  distance_90 = (sensor_echo1_time[2] * 13.02) / 1800;
  distance_135 = (sensor_echo2_time[0] * 13.02) / 1800;
  distance_180 = (sensor_echo2_time[1] * 13.02) / 1800;
  distance_225 = (sensor_echo2_time[2] * 13.02) / 1800;
  distance_270 = (sensor_echo3_time[0] * 13.02) / 1800;
  distance_315 = (sensor_echo3_time[1] * 13.02) / 1800;
  distance_360 = (sensor_echo3_time[2] * 13.02) / 1800;

  printf("S%03d%03d%03d%03d%03d%03d%03d%03d%03dE", distance_0,
         distance_45, distance_90, distance_135, distance_180, distance_225,
         distance_270, distance_315,distance_360);
 }
}
```

APPENDIX 4

Control Software Listing

```
/*************************************************************************
*******
*
*
*                        Robot Control Software Version 1.0
*
*
*
*                              Written By Mark Leyden
*
*                                 3 February 2000
*
*
*
*
*
*
*
*************************************************************************
******/

#include "includes.h"
#include <math.h>

#define  TASK_STK_SIZE      512  /* Size of each task's stacks (# of
WORDs)            */

#define TASK_START_ID           0    /* Application tasks IDs
*/
#define TASK_SONAR_ID           1
#define TASK_MOTOR_ID           2
#define TASK_CRUISE_ID          3
#define TASK_ARBITRATOR_ID      4
#define TASK_LIGHT_ID           5
#define TASK_TARGET_ID          6
#define TASK_EDGE_ID            7
#define TASK_MAP_ID             8
#define TASK_LEFT_WALL_ID       9
#define TASK_CONCAVE_ID         10
#define TASK_CONVEX_ID          11
#define TASK_RIGHT_WALL_ID      12
#define TASK_NAVIGATE_ID          13
#define TASK_LOCALIZATION_ID    14
#define TASK_SEARCH_FOR_WALL_ID 15


#define TASK_START_PRIO           10 /* Application tasks priorities
*/
#define TASK_SONAR_PRIO           11
#define TASK_MOTOR_PRIO           12
```

```
#define TASK_CRUISE_PRIO          13
#define TASK_ARBITRATOR_PRIO      14
#define TASK_LIGHT_PRIO           15
#define TASK_TARGET_PRIO          16
#define TASK_EDGE_PRIO            17
#define TASK_MAP_PRIO             18
#define TASK_LEFT_WALL_PRIO       19
#define TASK_CONCAVE_PRIO         20
#define TASK_CONVEX_PRIO          21
#define TASK_RIGHT_WALL_PRIO      22
#define TASK_NAVIGATE_PRIO        23
#define TASK_LOCALIZATION_PRIO    24
#define TASK_SEARCH_FOR_WALL_PRIO 25

/* Address of com port 2 */
#define PORT1 0x2F8

/* Motor definitions */
#define forward 0
#define reverse 1
#define LEFT    1
#define RIGHT   0

#define true   1
#define false  0

/* Register definitions for the 8255 */
#define ppi_control  0x303
#define ppi_portc    0x302
#define ppi_portb    0x301
#define ppi_porta    0x300
#define ppi_control2 0x30B
#define ppi_portc2   0x30A
#define ppi_portb2   0x309

/* Register definitions for the ADC */
#define adc_in1      0x310
#define adc_in2      0x311

#define con     100
#define convex  0
#define concave 1
#define door   2

unsigned char cruise_left_motor_speed;
unsigned char cruise_right_motor_speed;
unsigned char cruise_left_motor_direction;
unsigned char cruise_right_motor_direction;

unsigned char avoid_left_motor_speed;
unsigned char avoid_right_motor_speed;
unsigned char avoid_left_motor_direction;
unsigned char avoid_right_motor_direction;

unsigned char light_left_motor_speed;
unsigned char light_right_motor_speed;
unsigned char light_left_motor_direction;
unsigned char light_right_motor_direction;
```

```
unsigned char follow_left_motor_speed;
unsigned char follow_right_motor_speed;
unsigned char follow_left_motor_direction;
unsigned char follow_right_motor_direction;

unsigned char concave_left_motor_speed;
unsigned char concave_right_motor_speed;
unsigned char concave_left_motor_direction;
unsigned char concave_right_motor_direction;

unsigned char convex_left_motor_speed;
unsigned char convex_right_motor_speed;
unsigned char convex_left_motor_direction;
unsigned char convex_right_motor_direction;

unsigned char search_left_motor_speed;
unsigned char search_right_motor_speed;
unsigned char search_left_motor_direction;
unsigned char search_right_motor_direction;

unsigned int obstacle_avoidance_behaviour_active;
unsigned int cruise_behaviour_active;
unsigned int light_following_behaviour_active;
unsigned int edge_following_behaviour_active;
unsigned int convex_corner_or_door_behaviour_active;
unsigned int concave_corner_behaviour_active;
unsigned int left_wall_following_behaviour_active;
unsigned int right_wall_following_behaviour_active;
unsigned int search_for_wall_behaviour_active;
unsigned int navigation_behaviour_active;

unsigned int distance[20];
unsigned int convex_corner;
unsigned int concave_corner;
unsigned int door_detected;

unsigned int concave_corner_complete;
unsigned long distance_behind;
unsigned long distance_to_start_of_convex_corner_or_door;

int num_nodes;
int count;
int stop;
int counter;
int node_type[3];
int node_distance_to_previous[3];
int current_node;
int locked_onto_left_wall;
int locked_onto_right_wall;
int map_exists;

/* Helper Functions */
unsigned char Read_Register(unsigned int motor, unsigned char
reg_address);
void Write_To_Register(unsigned int motor, unsigned char reg_address,
unsigned char value);
void Goto_Position(unsigned int motor, unsigned long distance, unsigned
char direction);
```

```
void Set_Integral_Velocity(unsigned int motor, unsigned char velocity,
unsigned char acceleration, unsigned char direction);
void Set_Trapezoid_Profile_Move(unsigned int motor, unsigned long
distance, unsigned char velocity, unsigned char acceleration, unsigned
char direction);
unsigned long Read_Actual_Position(unsigned int motor);
unsigned int _min(unsigned int x, unsigned int y);
void Clear_Motor_Flag_Registers(void);
void Reset_Motor_Counters(void);
unsigned char Lsb2Msb(unsigned char ch);
void Write_Nodes_To_File(void);

/* Task Functions */
void   Acquire_Sonar_Data(void *data);
void   Obstacle_Avoidance(void *data);
void   Cruise(void *data);
void   Arbitrator(void *data);
void   Follow_Light(void *data);
void   Goto_Target(void *data);
void   Mapping(void *data);
void   Follow_Left_Wall(void *data);
void   Follow_Right_Wall(void *data);
void   Detect_Concave_Corner(void *data);
void   Detect_Convex_Corner_Or_Door(void *data);
void   Navigate(void *data);
void   Localization(void *data);
void   Search_For_Wall(void *data);
void   TaskStart(void *data);

OS_STK TaskStartStk[TASK_STK_SIZE];
OS_STK Acquire_Sonar_DataStk[TASK_STK_SIZE];
OS_STK CruiseStk[TASK_STK_SIZE];
OS_STK ArbitratorStk[TASK_STK_SIZE];
OS_STK LightStk[TASK_STK_SIZE];
OS_STK TargetStk[TASK_STK_SIZE];
OS_STK MapStk[TASK_STK_SIZE];
OS_STK LeftWallStk[TASK_STK_SIZE];
OS_STK RightWallStk[TASK_STK_SIZE];
OS_STK ConcaveStk[TASK_STK_SIZE];
OS_STK ConvexStk[TASK_STK_SIZE];
OS_STK NavigateStk[TASK_STK_SIZE];
OS_STK LocalizationStk[TASK_STK_SIZE];
OS_STK Search_For_WallStk[TASK_STK_SIZE];


struct node_structure
{
  int type;
  int next;
  unsigned long distance_to_next;
  int previous;
  unsigned long distance_to_previous;
} node[100];

struct temp_node_structure
{
  int type;
  int next;
  unsigned long distance_to_next;
```

```
  int previous;
  unsigned long distance_to_previous;
} temp_node[100];

void main (void)
{
   /* Set up serial communications for com port two.
      Baud rate is set to 2400. */
   outportb(PORT1 + 1, 0);
   outportb(PORT1 + 3, 0x80);
   outportb(PORT1 + 0, 0x30);
   outportb(PORT1 + 1, 0x00);
   outportb(PORT1 + 3, 0x03);
   outportb(PORT1 + 2, 0xC7);
   outportb(PORT1 + 4, 0x0B);

   /* Initialize variables */
   counter = 0;
   num_nodes = 0;
   stop = 0;
   count = 0;
   distance_behind = 0;
   current_node = -1;
   convex_corner = false;
   concave_corner = false;
   door_detected = false;
   concave_corner_complete = false;
   locked_onto_left_wall = false;
   locked_onto_right_wall = false;
   map_exists = true;

   cruise_behaviour_active = 0;
   obstacle_avoidance_behaviour_active = 0;
   light_following_behaviour_active = 0;
   edge_following_behaviour_active = 0;
   left_wall_following_behaviour_active = 0;
   right_wall_following_behaviour_active = 0;
   concave_corner_behaviour_active = 0;
   convex_corner_or_door_behaviour_active = 0;
   search_for_wall_behaviour_active = 0;
   navigation_behaviour_active = 0;

   OSTimeDlyHMSM(0, 0, 2, 0);

   OSInit();                  /* Initialize uC/OS-II
*/
   PC_DOSSaveReturn();        /* Save environment to return to DOS
*/
   PC_VectSet(uCOS, OSCtxSw); /* Install uC/OS-II's context switch vector
*/

   OSTaskCreateExt(TaskStart, (void *)0, &TaskStartStk[TASK_STK_SIZE-1],
TASK_START_PRIO,
               TASK_START_ID, &TaskStartStk[0], TASK_STK_SIZE, (void *)0,
               OS_TASK_OPT_STK_CHK | OS_TASK_OPT_STK_CLR);
   OSStart();                 /* Start multitasking
*/
}
```

```
void TaskStart(void *data)
{
   data = data;                        /* Prevent compiler warning
*/

   OS_ENTER_CRITICAL();
   PC_VectSet(0x08, OSTickISR);        /* Install uC/OS-II's clock tick
ISR        */
   PC_SetTickRate(OS_TICKS_PER_SEC);  /* Reprogram tick rate
*/
   OS_EXIT_CRITICAL();

   OSStatInit();                       /* Initialize uC/OS-II's statistics
*/

   OSTaskCreateExt(Acquire_Sonar_Data, (void *)0,
&Acquire_Sonar_DataStk[TASK_STK_SIZE-1], TASK_SONAR_PRIO,
               TASK_SONAR_ID, &Acquire_Sonar_DataStk[0], TASK_STK_SIZE,
(void *)0,
               OS_TASK_OPT_STK_CHK | OS_TASK_OPT_STK_CLR);
   /*OSTaskCreateExt(Obstacle_Avoidance, (void *)0,
&MotorStk[TASK_STK_SIZE-1], TASK_MOTOR_PRIO,
               TASK_MOTOR_ID, &MotorStk[0], TASK_STK_SIZE, (void *)0,
               OS_TASK_OPT_STK_CHK | OS_TASK_OPT_STK_CLR);
   OSTaskCreateExt(Cruise, (void *)0, &CruiseStk[TASK_STK_SIZE-1],
TASK_CRUISE_PRIO,
               TASK_CRUISE_ID, &CruiseStk[0], TASK_STK_SIZE, (void *)0,
               OS_TASK_OPT_STK_CHK | OS_TASK_OPT_STK_CLR);*/
   OSTaskCreateExt(Arbitrator, (void *)0, &ArbitratorStk[TASK_STK_SIZE-
1], TASK_ARBITRATOR_PRIO,
               TASK_ARBITRATOR_ID, &ArbitratorStk[0], TASK_STK_SIZE,
(void *)0,
               OS_TASK_OPT_STK_CHK | OS_TASK_OPT_STK_CLR);
   /*OSTaskCreateExt(Follow_Light, (void *)0, &LightStk[TASK_STK_SIZE-1],
TASK_LIGHT_PRIO,
               TASK_LIGHT_ID, &LightStk[0], TASK_STK_SIZE, (void *)0,
               OS_TASK_OPT_STK_CHK | OS_TASK_OPT_STK_CLR);
   OSTaskCreateExt(Goto_Target, (void *)0, &TargetStk[TASK_STK_SIZE-1],
TASK_TARGET_PRIO,
               TASK_TARGET_ID, &TargetStk[0], TASK_STK_SIZE, (void *)0,
               OS_TASK_OPT_STK_CHK | OS_TASK_OPT_STK_CLR);
   OSTaskCreateExt(Edge_Following, (void *)0, &EdgeStk[TASK_STK_SIZE-1],
TASK_EDGE_PRIO,
               TASK_EDGE_ID, &LightStk[0], TASK_STK_SIZE, (void *)0,
               OS_TASK_OPT_STK_CHK | OS_TASK_OPT_STK_CLR);*/
   OSTaskCreateExt(Follow_Left_Wall, (void *)0,
&LeftWallStk[TASK_STK_SIZE-1], TASK_LEFT_WALL_PRIO,
               TASK_LEFT_WALL_ID, &LeftWallStk[0], TASK_STK_SIZE, (void
*)0,
               OS_TASK_OPT_STK_CHK | OS_TASK_OPT_STK_CLR);
   OSTaskCreateExt(Follow_Right_Wall, (void *)0,
&RightWallStk[TASK_STK_SIZE-1], TASK_RIGHT_WALL_PRIO,
               TASK_RIGHT_WALL_ID, &RightWallStk[0], TASK_STK_SIZE, (void
*)0,
               OS_TASK_OPT_STK_CHK | OS_TASK_OPT_STK_CLR);
   OSTaskCreateExt(Detect_Concave_Corner, (void *)0,
&ConcaveStk[TASK_STK_SIZE-1], TASK_CONCAVE_PRIO,
               TASK_CONCAVE_ID, &ConcaveStk[0], TASK_STK_SIZE, (void *)0,
```

```
                OS_TASK_OPT_STK_CHK | OS_TASK_OPT_STK_CLR);
   OSTaskCreateExt(Detect_Convex_Corner_Or_Door, (void *)0,
&ConvexStk[TASK_STK_SIZE-1], TASK_CONVEX_PRIO,
                TASK_CONVEX_ID, &ConvexStk[0], TASK_STK_SIZE, (void *)0,
                OS_TASK_OPT_STK_CHK | OS_TASK_OPT_STK_CLR);
   OSTaskCreateExt(Search_For_Wall, (void *)0,
&Search_For_WallStk[TASK_STK_SIZE-1], TASK_SEARCH_FOR_WALL_PRIO,
                TASK_SEARCH_FOR_WALL_ID, &Search_For_WallStk[0],
TASK_STK_SIZE, (void *)0,
                OS_TASK_OPT_STK_CHK | OS_TASK_OPT_STK_CLR);
   OSTaskCreateExt(Mapping, (void *)0, &MapStk[TASK_STK_SIZE-1],
TASK_MAP_PRIO,
                TASK_MAP_ID, &MapStk[0], TASK_STK_SIZE, (void *)0,
                OS_TASK_OPT_STK_CHK | OS_TASK_OPT_STK_CLR);
   OSTaskCreateExt(Navigate, (void *)0, &NavigateStk[TASK_STK_SIZE-1],
TASK_NAVIGATE_PRIO,
                TASK_NAVIGATE_ID, &NavigateStk[0], TASK_STK_SIZE, (void
*)0,
                OS_TASK_OPT_STK_CHK | OS_TASK_OPT_STK_CLR);
   OSTaskCreateExt(Localization, (void *)0,
&LocalizationStk[TASK_STK_SIZE-1], TASK_LOCALIZATION_PRIO,
                TASK_LOCALIZATION_ID, &LocalizationStk[0], TASK_STK_SIZE,
(void *)0,
                OS_TASK_OPT_STK_CHK | OS_TASK_OPT_STK_CLR);

   while (1)
   {
      OSTimeDlyHMSM(0, 0, 0, 10);
   }
}

/*************************************************************************
*******
*
*
*                                Acquire_Sonar_Data Routine
*
*
*
*  This function reads in the sonar data from the serial port. This data
is    *
*  transmitted from the 8051 every time a complete scan of all the
ultrasonic  *
*  sensors is performed. This occurs approximately 2 times a second. The
data  *
*  is transmitted in the form of a packet. At the start of the packet is
a    *
*  START character denmoted by the letter 'S'. This is used to signify
the    *
*  start of the packet. Following this are 27 bytes which comprise the
sonar   *
*  data. Since 9 ultrasonic sensors are used on the robot, each range
reading  *
*  consists of 3 bytes. For example, if a sensor detected an obstacle at
25    *
*  inches away, the three character bytes transmitted for this value
would be  *
*  '025'. At the end of the packet is an END character which signifies
the end *
```

```
*  of the packet. The function constantly polls the serial port for any
*
*  received characters. Initially, the function waits till a start
character   *
*  is received. Once this is received, the next 27 bytes are stored in an
*
*  array. After this, the function waits till the END character is
received.   *
*  Following this, the function processes the 27 bytes received and
stores     *
*  the range readings for each of the ultrasonic sensors in a series of
global *
*  variables. These variables can then be read by any other function.
*
*
*
*************************************************************************
******/

void Acquire_Sonar_Data(void *data)
{
  int c, ch, counter, start_char;
  char sentence[50];

  data = data;

  start_char = FALSE;
  counter = 0;

  while (1)
  {
   c = inportb(PORT1 + 5);
   if (c & 1)
   {
     ch = inportb(PORT1);

     if (start_char == FALSE && ch == 'S') start_char = TRUE;
     else if (start_char == TRUE)
     {
      sentence[counter++] = ch;
      if (counter == 27) sentence[counter] = '\n';
      if (counter == 28)
      {
        if (ch == 'E')
        {
         sscanf(sentence, "%3d%3d%3d%3d%3d%3d%3d%3d%3d",
               &distance[0], &distance[1], &distance[2],
               &distance[3], &distance[4], &distance[5],
               &distance[6], &distance[7], &distance[8]);
        }
        start_char = FALSE;
        counter = 0;
      }
     }
   }
   OSTimeDlyHMSM(0, 0, 0, 5);
  }
}
```

```
/*************************************************************************
*******
*
*
*                                Obstacle Avoidance Routine
*
*
*
*  This function is the obstacle avoidance behaviour. Depending on which
*
*  sensor detects an obstacle, the speed of the motors is increased and
*
*  decreased by varying amounts. The behaviour will only become active if
an   *
*  obstacle is detected within a certain range.
*
*
*
*************************************************************************
******/

/*void Obstacle_Avoidance(void *data)
{
  unsigned int index;
  unsigned long position;

  unsigned int forward_distance;
  unsigned int left_distance;
  unsigned int right_distance;
  unsigned int left_forward_distance;
  unsigned int right_forward_distance;
  unsigned int reverse_distance;
  unsigned int left_reverse_distance;
  unsigned int right_reverse_distance;

  unsigned int s[9];
  int left_o, right_o;
  int speed_l, speed_r;
  int temp;
  unsigned char speed;
  //int w_l[6] = { 1, 2, 0, -9, -5, -2 };
  //int w_r[6] = { -2, -5, -9, 0, 2, 1 };
  int w_l[6] = { 1, 2, 0, -9, -3, -2 };
  int w_r[6] = { -2, -3, -9, 0, 2, 1 };
  int cond[6] = { 15, 18, 20, 20, 18, 15 };

  data = data;

  //outportb(ppi_control2, 0x83);
  //outportb(ppi_portc2, 0xF0);

  Write_To_Register(LEFT, 0x00, 0x00);
  Write_To_Register(LEFT, 0x02, 0x00);
  Write_To_Register(LEFT, 0x03, 0x00);
  Write_To_Register(LEFT, 0x04, 0x00);
  Write_To_Register(LEFT, 0x05, 0x00);

  Write_To_Register(RIGHT, 0x00, 0x00);
  Write_To_Register(RIGHT, 0x02, 0x00);
```

```
  Write_To_Register(RIGHT, 0x03, 0x00);
  Write_To_Register(RIGHT, 0x04, 0x00);
  Write_To_Register(RIGHT, 0x05, 0x00);

  Write_To_Register(RIGHT, 0x13, 0x00); // Sets actual position counter
to zero
  Write_To_Register(RIGHT, 0x0F, 0xFF); // Sets sample timer period to
4096 uS
  Write_To_Register(RIGHT, 0x05, 0x03);
  Write_To_Register(LEFT, 0x13, 0x00); // Sets actual position counter to
zero
  Write_To_Register(LEFT, 0x0F, 0xFF); // Sets sample timer period to
4096 uS
  Write_To_Register(LEFT, 0x05, 0x03);

  while (1)
  {
   s[0] = distance[6];
   s[1] = distance[7];
   s[2] = distance[8];
   s[3] = distance[0];
   s[4] = distance[1];
   s[5] = distance[2];

   left_o = 0;
   right_o = 0;

   for (index = 0; index < 6; index++)
   {
     if (s[index] < cond[index])
     {
      left_o += w_l[index];
      right_o += w_r[index];
     }
   }

   speed_l = 8 + left_o;
   speed_r = 8 + right_o;

   if (speed_l < 0)
   {
     temp = abs(speed_l);
     avoid_left_motor_speed = (unsigned char)temp;
     avoid_left_motor_direction = reverse;
   }
   else
   {
     avoid_left_motor_speed = (unsigned char)speed_l;
     avoid_left_motor_direction = forward;
   }

   if (speed_r < 0)
   {
     temp = abs(speed_r);
     avoid_right_motor_speed = (unsigned char)temp;
     avoid_right_motor_direction = reverse;
   }
   else
   {
```

```
     avoid_right_motor_speed = (unsigned char)speed_r;
     avoid_right_motor_direction = forward;
   }

   // Determines if the behaviour should become active
   if (!(speed_l == 8 && speed_r == 8))
     obstacle_avoidance_behaviour_active = 1;
   else
     obstacle_avoidance_behaviour_active = 0;

   OSTimeDlyHMSM(0, 0, 0, 10);
  }
}*/

/*************************************************************************
*******
*
*
*                                      Cruise Routine
*
*
*
*  This function is the cruise behaviour. It simply causes the robot to
drive  *
*  around. Since this is the lowest level behaviour, it is always active.
*
*
*
*************************************************************************
******/

/*void Cruise(void *data)
{
  data = data;

  while (1)
  {
   cruise_left_motor_speed = 8;
   cruise_right_motor_speed = 8;
   cruise_left_motor_direction = forward;
   cruise_right_motor_direction = forward;
   cruise_behaviour_active = 1;

   OSTimeDlyHMSM(0, 0, 0, 10);
  }
}*/

/*************************************************************************
*******
*
*
*                                  Follow_Light Routine
*
*
*
*  This function is the light following behaviour. It is used to track a
light *
*  source detected by the robots photcells.
*
```

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

/*void Follow_Light(void *data)
{
  unsigned char chl;
  unsigned char chr;
  int diff;

  data = data;

  outportb(ppi_control2, 0x93);
  outportb(ppi_portc2, 0xF0);

  //outportb(ppi_control2, 0x91);
  //outportb(ppi_portc2, 0xF0);

  while (1)
  {
   //outportb(ppi_portb2, 0x55);
   outportb(adc_in1, 0x00);
   OSTimeDlyHMSM(0, 0, 0, 1);
   chl = inportb(adc_in1);
   chl = Lsb2Msb(chl);

   OSTimeDlyHMSM(0, 0, 0, 10);

   outportb(adc_in2, 0x00);
   OSTimeDlyHMSM(0, 0, 0, 1);
   chr = inportb(adc_in2);
   chr = Lsb2Msb(chr);

   diff = (int)chl - (int)chr;

   printf("num1 = %d  num2 = %d ..\n", (int)chl, (int)chr);
   OSTimeDlyHMSM(0, 0, 0, 333);

   // If the right photocell detects a light, then turn right
   if (abs(diff) > 35 && chl < chr)
   {
     light_left_motor_speed = 10;
     light_right_motor_speed = 5;
     light_left_motor_direction = forward;
     light_right_motor_direction = reverse;

     light_following_behaviour_active = 1;
   }

   // If the left photocell detects a light, then turn left
   else if (abs(diff) > 35 && chl > chr)
   {
     light_left_motor_speed = 5;
     light_right_motor_speed = 10;
     light_left_motor_direction = reverse;
     light_right_motor_direction = forward;

     light_following_behaviour_active = 1;
```

```
   }

   else
     light_following_behaviour_active = 0;

   OSTimeDlyHMSM(0, 0, 0, 10);
  }
}*/

/*************************************************************************
*******
*
*
*                                  Goto_Target Routine
*
*
*
*  This function is the target seeking behaviour. It drives the robot
towards  *
*  a predefined target location.
*
*
*
*************************************************************************
******/

/*void Goto_Target(void *data)
{
  int x1, y1, x2, y2;
  double heading;
  double wheel;
  //double cur_heading;
  double distance;
  double angle;

  data = data;

  //OSTimeDlyHMSM(0, 0, 5, 0);

  outportb(ppi_control2, 0x83);
  outportb(ppi_portc2, 0xF0);

  while (1)
  {
   Clear_Motor_Flag_Registers();

   //Write_To_Register(RIGHT, 0x13, 0x00); // Sets actual position
counter to zero
   Write_To_Register(RIGHT, 0x0F, 0xFF); // Sets sample timer period to
4096 uS
   Write_To_Register(RIGHT, 0x05, 0x03);
   //Write_To_Register(LEFT, 0x13, 0x00); // Sets actual position counter
to zero
   Write_To_Register(LEFT, 0x0F, 0xFF); // Sets sample timer period to
4096 uS
   Write_To_Register(LEFT, 0x05, 0x03);

   x1 = 0;
   y1 = 0;
```

```
   x2 = 48;
   y2 = 48;

   if (x2 > x1 && y1 == y2) heading = 90.;
   else if (x2 < x1 && y1 == y2) heading = 270.;
   else if (x1 == x2 && y1 < y2) heading = 0.;
   else if (x1 == x2 && y1 > y2) heading = 180.;

   else
   {
      angle = atan(((double)(y2 - y1) / (double)(x2 - x1)));
      angle = angle * 57.3;
      if (x2 > x1 && y2 > y1) heading = 90. - angle;
      else if (x2 < x1 && y2 < y1) heading = 270. - angle;
      else if (x2 < x1 && y2 > y1) heading = 270. - angle;
      else if (x2 > x1 && y2 < y1) heading = 90. - angle;
   }

   if (heading < 0) heading = heading + 360;
   //if (cur_heading < heading) heading -= cur_heading;
   //else heading += cur_heading;

   if (heading < 0) heading = 0 - heading;

   distance = sqrt((double)(x2 - x1)*(double)(x2 - x1) + (double)(y2 -
y1)*(double)(y2 - y1));
   distance = distance / 8.29576 * 2000;
   //wheel = (14.82439 / 360 * heading) / 8.29576 * 2000;
   //heading = 45.0;
   //distance = 50;
   //printf("Heading = %f\n", heading);
   if (heading >= 0 && heading <= 180)
   {
      wheel = (31.0 / 360 * heading) / 8.29576 * 2000;
      Set_Trapezoid_Profile_Move(LEFT, (unsigned long)wheel, 8, 15,
forward);
      Set_Trapezoid_Profile_Move(RIGHT, (unsigned long)wheel, 8, 15,
reverse);
   }
   else
   {
      wheel = (31.0 / 360 * (360 - heading)) / 8.29576 * 2000;
      Set_Trapezoid_Profile_Move(LEFT, (unsigned long)wheel, 8, 15,
reverse);
      Set_Trapezoid_Profile_Move(RIGHT, (unsigned long)wheel, 8, 15,
forward);
   }

   while ((inportb(ppi_portc2) & 0x05) != 0x00) ;
   Set_Trapezoid_Profile_Move(LEFT, (unsigned long)distance, 8, 15,
forward);
   Set_Trapezoid_Profile_Move(RIGHT, (unsigned long)distance, 8, 15,
forward);
   while ((inportb(ppi_portc2) & 0x05) != 0x00) ;

   while (1) ;
   //exit(0);

   OSTimeDlyHMSM(0, 0, 0, 10);
```

```
  }
}*/

/************************************************************************
*******
*
*
*                                  Arbitrator Routine
*
*
*
*  This function is the arbitrator. This is used to decide which beaviour
*
*  should be active at any given time. Since subsumption is used as the
*
*  control architecture, a priority based arbitration mehanism is used.
Higher *
*  level behaviours are given higher precedence over lower level
behaviours.   *
*
*
************************************************************************
******/

void Arbitrator(void *data)
{
  INT32U clk;
  INT32U clk_start;

  data = data;

  OSTimeDlyHMSM(0, 0, 2, 0);
  Clear_Motor_Flag_Registers();
  Reset_Motor_Counters();

  while (1)
  {
   if (search_for_wall_behaviour_active == true)
   {
      Set_Integral_Velocity(LEFT, search_left_motor_speed, 20,
search_left_motor_direction);
      Set_Integral_Velocity(RIGHT, search_right_motor_speed, 20,
search_right_motor_direction);
   }

   else if (concave_corner_behaviour_active == true)
   {
      Set_Integral_Velocity(LEFT, concave_left_motor_speed, 20,
concave_left_motor_direction);
      Set_Integral_Velocity(RIGHT, concave_right_motor_speed, 20,
concave_right_motor_direction);
   }

   else if (convex_corner_or_door_behaviour_active == true)
   {
      Set_Integral_Velocity(LEFT, convex_left_motor_speed, 20,
convex_left_motor_direction);
      Set_Integral_Velocity(RIGHT, convex_right_motor_speed, 20,
convex_right_motor_direction);
```

```
   }

   else if (left_wall_following_behaviour_active == true)
   {
      Set_Integral_Velocity(LEFT, follow_left_motor_speed, 20,
follow_left_motor_direction);
      Set_Integral_Velocity(RIGHT, follow_right_motor_speed, 20,
follow_right_motor_direction);
   }

   else if (right_wall_following_behaviour_active == true)
   {
      Set_Integral_Velocity(LEFT, follow_left_motor_speed, 20,
follow_left_motor_direction);
      Set_Integral_Velocity(RIGHT, follow_right_motor_speed, 20,
follow_right_motor_direction);
   }

   OSTimeDlyHMSM(0, 0, 0, 10);
  }
}


unsigned long Read_Actual_Position(unsigned int motor)
{
  unsigned long temp, position;
  unsigned char reg1, reg2, reg3;

  reg3 = Read_Register(motor, 0x14);
  reg2 = Read_Register(motor, 0x13);
  reg1 = Read_Register(motor, 0x12);
  temp = (long)reg2;
  temp <<= 8;
  temp |= (long)reg3;
  position = (long)reg1;
  position <<= 16;
  position |= temp;
  return (position / 241);
}

/*************************************************************************
*******
*
*
*                               Write_To_Register Routine
*
*
*
*  This function writes an 8-bit value to a specified register of the
*
*  HCTL-1100. The function takes three parameters - the desired motor to
use   *
*  (either left or right), the address of the register to write to and
the     *
*  8-bit value to be written. The non-overlapped mode of interfacing is
used   *
*  used when writing a value to the controller.
*
```

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

void Write_To_Register(unsigned int motor, unsigned char reg_address,
unsigned char value)
{
  unsigned int wait;
  outportb(ppi_control, 0x80);

  if (motor == LEFT)
  {
   outportb(ppi_portc, reg_address);
   outportb(ppi_porta, 0xFD);   // ALE low
   for (wait = 0; wait < con; wait++) ;
   outportb(ppi_porta, 0xFA);   // ALE high, CS low, R/W low
   outportb(ppi_portc, value);
   outportb(ppi_porta, 0xFE);   // CS high
   outportb(ppi_porta, 0xFF);   // R/W high
  }

  else if (motor == RIGHT)
  {
   outportb(ppi_portb, reg_address);
   outportb(ppi_porta, 0xDF);   // ALE low
   for (wait = 0; wait < con; wait++) ;
   outportb(ppi_porta, 0xAF);   // ALE high, CS low, R/W low
   outportb(ppi_portb, value);
   outportb(ppi_porta, 0xEF);   // CS high
   outportb(ppi_porta, 0xFF);   // R/W high
  }
}

/*************************************************************************
*******
*
*
*                                Read_From_Register Routine
*
*
*
*  This function reads an 8-bit value from a specified register of the
*
*  HCTL-1100. The function takes two parameters - the desired motor to
use     *
*  (either left or right) and the address of the register to read from.
The    *
*  8-bit value read from the controller is stored as a return value to
the     *
*  calling function. The non-overlapped mode of interfacing is used used
when  *
*  writing a value to the controller.
*
*
*
*************************************************************************
******/
```

```
unsigned char Read_Register(unsigned int motor, unsigned char
reg_address)
{
  unsigned char reg_value;
  unsigned int wait;

  outportb(ppi_control, 0x80);

  if (motor == LEFT)
  {
   outportb(ppi_portc, reg_address);
   outportb(ppi_porta, 0xFD);   // ALE low
   for (wait = 0; wait < con; wait++) ;
   outportb(ppi_porta, 0xFB);   // ALE high, CS low
   outportb(ppi_porta, 0xFF);   // CS high
   outportb(ppi_porta, 0xF7);   // OE low
   outportb(ppi_control, 0x89);
   reg_value = inportb(ppi_portc);
   outportb(ppi_porta, 0xFF);   // OE high
  }

  else if (motor == RIGHT)
  {
   outportb(ppi_portb, reg_address);
   outportb(ppi_porta, 0xDF);   // ALE low
   for (wait = 0; wait < con; wait++) ;
   outportb(ppi_porta, 0xBF);   // ALE high, CS low
   outportb(ppi_porta, 0xFF);   // CS high
   outportb(ppi_porta, 0x7F);   // OE low
   outportb(ppi_control, 0x82);
   reg_value = inportb(ppi_portb);
   outportb(ppi_porta, 0xFF);   // OE high
  }

  return reg_value;
}

/*************************************************************************
*******
*
*
*                              Position Control Routine
*
*
*
*  This routine sets up the HCTL-1100 to perform position control. This
*
*  performs point to point position moves with no velocity profiling. The
user *
*  specifies a 24-bit position command which the controller compares to
the    *
*  24-bit actual position. The position error is calculated, the full
digital  *
*  lead compensation is applied and the motor command is output. The
*
*  controller will remain position locked at a destination until a new
*
*  position command is given. The function takes three parameters - the
*
```

```
*  desired motor to use (either left or right), the distance to move and
the   *
*  direction in which to move (either forward or reverse).
*
*
*
*  Registers used: OOH  Flag register (all flags should be cleared)
*
*                  0CH  Command Position MSB (two's complement)
*
*                  0DH  Command Position (two's complement)
*
*                  0EH  Command Position LSB (two's complement)
*
*
*
*  Parameters:     motor        - LEFT or RIGHT
*
*                  distance     - Distance to move in encoder quadrature
*
*                                 counts. One wheel revolution is equal
to     *
*                                 2000 counts.
*
*                  direction    - forward or reverse
*
*
*
*************************************************************************
******/

void Goto_Position(unsigned int motor, unsigned long distance, unsigned
char direction)
{
  unsigned long temp;

  if (direction == reverse) distance = 0xFFFFFF - distance;
  temp = distance >> 16;
  temp &= 0x0000FF;
  Write_To_Register(motor, 0x0C, (unsigned char)temp);
  temp = distance >> 8;
  temp &= 0x0000FF;
  Write_To_Register(motor, 0x0D, (unsigned char)temp);
  temp = distance & 0x0000FF;
  Write_To_Register(motor, 0x0E, (unsigned char)temp);
}


/*************************************************************************
*******
*
*
*                           Integral Velocity Routine
*
*
*
*  This routine sets up the HCTL-1100 to perform integral velocity
control.    *
```

```
*  This performs continuous velocity profiling. The function takes four
*
*  parameters - the desired motor to use (either LEFT or RIGHT),
velocity,     *
*  acceleration and the direction of rotation for the motor (either
forward or *
*  reverse).
*
*
*
*  Registers used: OOH  Flag register (F5 set to begin move)
*
*                  26H  Acceleration LSB
*
*                  27H  Acceleration MSB
*
*                  3CH  Command Velocity (two's complement)
*
*                  OFH  Sample Timer
*
*
*
*  Parameters:     motor        - LEFT or RIGHT
*
*                  velocity     - A value between 0 and 15
*
*                  acceleration - A value between 0 and 20
*
*                  direction    - forward or reverse
*
*
*
*************************************************************************
******/

void Set_Integral_Velocity(unsigned int motor, unsigned char velocity,
unsigned char acceleration, unsigned char direction)
{
  Write_To_Register(motor, 0x27, 0x00);
  Write_To_Register(motor, 0x26, acceleration);

  if (direction == forward)
  {
   Write_To_Register(motor, 0x3C, velocity);
  }
  else if (direction == reverse)
  {
   Write_To_Register(motor, 0x3C, 0xFF - velocity);
  }

  Write_To_Register(motor, 0x00, 0x0D);
}

/*************************************************************************
*******
*
*
*                          Trapezoidal Profile Routine
*
```

```
*
*
*  This routine sets up the HCTL-1100 to perform a trapeziodal profile
move.   *
*  This performs point to point position moves while at the same time
*
*  profiling the velocity trajectory to a trapezoid or triangle. The
function  *
*  takes five parameters - the desired motor to use (either LEFT or
RIGHT),    *
*  the distance to move, acceleration, maximum velocity and the direction
in   *
*  which to move (either forward or reverse). The HCTL-1100 computes the
*
*  necessary profile to conform to the command data. If maximum velocity
is    *
*  reached before the distance halfway point, the profile will be
trapezoidal, *
*  otherwise the profile will be triangular.
*
*
*
*  Registers used: OOH  Flag register (F0 set to begin move)
*
*                  26H  Acceleration LSB
*
*                  27H  Acceleration MSB
*
*                  28H  Maximum Velocity
*
*                  29H  Final Position LSB
*
*                  2AH  Final Position
*
*                  2BH  Final Position MSB
*
*                  13H  Position Counter
*
*                  0CH  Command Position MSB (two's complement)
*
*                  0DH  Command Position (two's complement)
*
*                  0EH  Command Position LSB (two's complement)
*
*                  OFH  Sample Timer
*
*
*
*  Parameters:     motor        - LEFT or RIGHT
*
*                  distance     - Distance to move in encoder quadrature
*
*                                 counts. One wheel revolution is equal
to     *
*                                 2000 counts.
*
*                  velocity     - A value between 0 and 15
*
```

```
*                   acceleration - A value between 0 and 20
*
*                   direction    - forward or reverse
*
*
*
*******************************************************************
******/

void Set_Trapezoid_Profile_Move(unsigned int motor, unsigned long
distance, unsigned char velocity, unsigned char acceleration, unsigned
char direction)
{
  unsigned long temp;

  Write_To_Register(motor, 0x0C, 0x00); // Sets the command position to 0
  Write_To_Register(motor, 0x0D, 0x00);
  Write_To_Register(motor, 0x0E, 0x00);

  Write_To_Register(motor, 0x13, 0x00); // Sets actual position counter
to zero

  Write_To_Register(motor, 0x27, 0x00);
  Write_To_Register(motor, 0x26, acceleration);

  Write_To_Register(motor, 0x28, velocity);

  if (direction == reverse) distance = 0xFFFFFF - distance;
  temp = distance >> 16;
  temp &= 0x0000FF;
  Write_To_Register(motor, 0x2B, (unsigned char)temp);
  temp = distance >> 8;
  temp &= 0x0000FF;
  Write_To_Register(motor, 0x2A, (unsigned char)temp);
  temp = distance & 0x0000FF;
  Write_To_Register(motor, 0x29, (unsigned char)temp);

  Write_To_Register(motor, 0x00, 0x08);
}


unsigned int _min(unsigned int x, unsigned int y)
{
  return ((x < y) ? x : y);
}

void Clear_Motor_Flag_Registers(void)
{
   Write_To_Register(LEFT, 0x00, 0x00);
   Write_To_Register(LEFT, 0x02, 0x00);
   Write_To_Register(LEFT, 0x03, 0x00);
   Write_To_Register(LEFT, 0x04, 0x00);
   Write_To_Register(LEFT, 0x05, 0x00);

   Write_To_Register(RIGHT, 0x00, 0x00);
   Write_To_Register(RIGHT, 0x02, 0x00);
   Write_To_Register(RIGHT, 0x03, 0x00);
   Write_To_Register(RIGHT, 0x04, 0x00);
   Write_To_Register(RIGHT, 0x05, 0x00);
```

```
}

void Reset_Motor_Counters(void)
{
   Write_To_Register(RIGHT, 0x13, 0x00); // Sets actual position counter
to zero
   Write_To_Register(RIGHT, 0x0F, 0xFF); // Sets sample timer period to
4096 uS
   Write_To_Register(RIGHT, 0x05, 0x03);
   Write_To_Register(LEFT, 0x13, 0x00); // Sets actual position counter
to zero
   Write_To_Register(LEFT, 0x0F, 0xFF); // Sets sample timer period to
4096 uS
   Write_To_Register(LEFT, 0x05, 0x03);
}

unsigned char Lsb2Msb(unsigned char ch)
{
   unsigned char num;

   num = 0;

   if ((ch & 0x01) == 0x01) num = num | 0x80;
   if ((ch & 0x02) == 0x02) num = num | 0x40;
   if ((ch & 0x04) == 0x04) num = num | 0x20;
   if ((ch & 0x08) == 0x08) num = num | 0x10;
   if ((ch & 0x10) == 0x10) num = num | 0x08;
   if ((ch & 0x20) == 0x20) num = num | 0x04;
   if ((ch & 0x40) == 0x40) num = num | 0x02;
   if ((ch & 0x80) == 0x80) num = num | 0x01;

   return num;
}

void Detect_Concave_Corner(void *data)
{
   int diff;

   data = data;
   OSTimeDlyHMSM(0, 0, 2, 0);

   while (1)
   {
      if (distance[8] < 14 && left_wall_following_behaviour_active ==
true)
      {
         concave_corner = true;
         concave_left_motor_speed = 2;
         concave_left_motor_direction = forward;
         concave_right_motor_speed = 2;
         concave_right_motor_direction = reverse;
         concave_corner_behaviour_active = true;

         diff = distance[7] - distance[6];
         while ((diff < 1) || distance[8] < 18 || distance[0] < 18)
         {
            diff = distance[7] - distance[6];
            OSTimeDlyHMSM(0, 0, 0, 10);
         }
```

```
         concave_left_motor_speed = 0;
         concave_left_motor_direction = forward;
         concave_right_motor_speed = 0;
         concave_right_motor_direction = forward;
         OSTimeDlyHMSM(0, 0, 1, 0);
         concave_corner_complete = true;

         Clear_Motor_Flag_Registers();
         Reset_Motor_Counters();
      }

      else if (distance[8] < 14 && right_wall_following_behaviour_active
== true)
      {
         concave_corner = true;
         concave_left_motor_speed = 2;
         concave_left_motor_direction = reverse;
         concave_right_motor_speed = 2;
         concave_right_motor_direction = forward;
         concave_corner_behaviour_active = true;

         diff = distance[1] - distance[2];
         while ((diff < 1) || distance[8] < 18 || distance[0] < 18)
         {
            diff = distance[1] - distance[2];
            OSTimeDlyHMSM(0, 0, 0, 10);
         }

         concave_left_motor_speed = 0;
         concave_left_motor_direction = forward;
         concave_right_motor_speed = 0;
         concave_right_motor_direction = forward;
         OSTimeDlyHMSM(0, 0, 1, 0);
         concave_corner_complete = true;

         Clear_Motor_Flag_Registers();
         Reset_Motor_Counters();
      }

      else concave_corner_behaviour_active = false;

      OSTimeDlyHMSM(0, 0, 0, 10);
   }
}

void Detect_Convex_Corner_Or_Door(void *data)
{
   int diff;
   data = data;
   OSTimeDlyHMSM(0, 0, 2, 0);

   while (1)
   {
      if (distance[6] > 20 && distance[7] > 25 &&
left_wall_following_behaviour_active == true)
      {
         distance_to_start_of_convex_corner_or_door =
Read_Actual_Position(LEFT);
```

```
         if (navigation_behaviour_active == true && node[current_node +
1].type == convex)
         {
            convex_corner = true;

            convex_left_motor_speed = 3;
            convex_left_motor_direction = forward;
            convex_right_motor_speed = 3;
            convex_right_motor_direction = forward;
            convex_corner_or_door_behaviour_active = true;
            OSTimeDlyHMSM(0, 0, 2, 0);
            convex_left_motor_speed = 0;
            convex_left_motor_direction = reverse;
            convex_right_motor_speed = 0;
            convex_right_motor_direction = reverse;
            OSTimeDlyHMSM(0, 0, 0, 500);
            convex_left_motor_speed = 0;
            convex_left_motor_direction = forward;
            convex_right_motor_speed = 4;
            convex_right_motor_direction = forward;
            OSTimeDlyHMSM(0, 0, 3, 500);
            convex_left_motor_speed = 3;
            convex_left_motor_direction = forward;
            convex_right_motor_speed = 3;
            convex_right_motor_direction = forward;
            while (distance[6] >= 20) OSTimeDlyHMSM(0, 0, 0, 10);
            //OSTimeDlyHMSM(0, 0, 5, 0);

            Clear_Motor_Flag_Registers();
            Reset_Motor_Counters();
            continue;
         }

         else if (navigation_behaviour_active == true &&
node[current_node + 1].type == door)
         {
            door_detected = true;

            convex_left_motor_speed = 3;
            convex_left_motor_direction = forward;
            convex_right_motor_speed = 3;
            convex_right_motor_direction = forward;
            convex_corner_or_door_behaviour_active = true;
            OSTimeDlyHMSM(0, 0, 9, 500);
            convex_left_motor_speed = 0;
            convex_left_motor_direction = reverse;
            convex_right_motor_speed = 0;
            convex_right_motor_direction = reverse;
            OSTimeDlyHMSM(0, 0, 0, 500);

            Clear_Motor_Flag_Registers();
            Reset_Motor_Counters();
            continue;
         }

         convex_left_motor_speed = 3;
         convex_left_motor_direction = forward;
         convex_right_motor_speed = 3;
```

```
convex_right_motor_direction = forward;
convex_corner_or_door_behaviour_active = true;
OSTimeDlyHMSM(0, 0, 9, 500);
convex_left_motor_speed = 0;
convex_left_motor_direction = reverse;
convex_right_motor_speed = 0;
convex_right_motor_direction = reverse;
OSTimeDlyHMSM(0, 0, 0, 500);

if (distance[6] > 20)
{
   convex_left_motor_speed = 3;
   convex_left_motor_direction = reverse;
   convex_right_motor_speed = 3;
   convex_right_motor_direction = reverse;
   OSTimeDlyHMSM(0, 0, 5, 0);
   convex_left_motor_speed = 0;
   convex_left_motor_direction = reverse;
   convex_right_motor_speed = 0;
   convex_right_motor_direction = reverse;
   OSTimeDlyHMSM(0, 0, 0, 500);

   convex_corner = true;

   convex_left_motor_speed = 0;
   convex_left_motor_direction = forward;
   convex_right_motor_speed = 4;
   convex_right_motor_direction = forward;
   OSTimeDlyHMSM(0, 0, 3, 500);
   convex_left_motor_speed = 3;
   convex_left_motor_direction = forward;
   convex_right_motor_speed = 3;
   convex_right_motor_direction = forward;
   while (distance[6] >= 20) OSTimeDlyHMSM(0, 0, 0, 10);

   //OSTimeDlyHMSM(0, 0, 5, 0);

   Clear_Motor_Flag_Registers();
   Reset_Motor_Counters();
}

else
{
   door_detected = true;
   while (distance[6] < 20)
   {
      convex_left_motor_speed = 2;
      convex_left_motor_direction = reverse;
      convex_right_motor_speed = 2;
      convex_right_motor_direction = reverse;
      OSTimeDlyHMSM(0, 0, 0, 10);
   }
   convex_left_motor_speed = 3;
   convex_left_motor_direction = forward;
   convex_right_motor_speed = 3;
   convex_right_motor_direction = forward;
   while (distance[6] >= 20) OSTimeDlyHMSM(0, 0, 0, 10);
   //OSTimeDlyHMSM(0, 0, 2, 0);
```

```
            Clear_Motor_Flag_Registers();
            Reset_Motor_Counters();
         }
      }

      else if (distance[2] > 20 && distance[1] > 25 &&
right_wall_following_behaviour_active == true)
      {
         distance_to_start_of_convex_corner_or_door =
Read_Actual_Position(LEFT);

         if (navigation_behaviour_active == true && node[current_node +
1].type == convex)
         {
            convex_corner = true;

            convex_left_motor_speed = 3;
            convex_left_motor_direction = forward;
            convex_right_motor_speed = 3;
            convex_right_motor_direction = forward;
            convex_corner_or_door_behaviour_active = true;
            OSTimeDlyHMSM(0, 0, 2, 0);
            convex_left_motor_speed = 0;
            convex_left_motor_direction = reverse;
            convex_right_motor_speed = 0;
            convex_right_motor_direction = reverse;
            OSTimeDlyHMSM(0, 0, 0, 500);
            convex_left_motor_speed = 4;
            convex_left_motor_direction = forward;
            convex_right_motor_speed = 0;
            convex_right_motor_direction = forward;
            OSTimeDlyHMSM(0, 0, 3, 500);
            convex_left_motor_speed = 3;
            convex_left_motor_direction = forward;
            convex_right_motor_speed = 3;
            convex_right_motor_direction = forward;
            while (distance[2] >= 20) OSTimeDlyHMSM(0, 0, 0, 10);
            //OSTimeDlyHMSM(0, 0, 5, 0);

            Clear_Motor_Flag_Registers();
            Reset_Motor_Counters();
            continue;
         }

         else if (navigation_behaviour_active == true &&
node[current_node + 1].type == door)
         {
            door_detected = true;

            convex_left_motor_speed = 3;
            convex_left_motor_direction = forward;
            convex_right_motor_speed = 3;
            convex_right_motor_direction = forward;
            convex_corner_or_door_behaviour_active = true;
            OSTimeDlyHMSM(0, 0, 9, 500);
            convex_left_motor_speed = 0;
            convex_left_motor_direction = reverse;
            convex_right_motor_speed = 0;
            convex_right_motor_direction = reverse;
```

```
   OSTimeDlyHMSM(0, 0, 0, 500);

   Clear_Motor_Flag_Registers();
   Reset_Motor_Counters();
   continue;
}

convex_left_motor_speed = 3;
convex_left_motor_direction = forward;
convex_right_motor_speed = 3;
convex_right_motor_direction = forward;
convex_corner_or_door_behaviour_active = true;
OSTimeDlyHMSM(0, 0, 9, 500);
convex_left_motor_speed = 0;
convex_left_motor_direction = reverse;
convex_right_motor_speed = 0;
convex_right_motor_direction = reverse;
OSTimeDlyHMSM(0, 0, 0, 500);

if (distance[2] > 20)
{
   convex_left_motor_speed = 3;
   convex_left_motor_direction = reverse;
   convex_right_motor_speed = 3;
   convex_right_motor_direction = reverse;
   OSTimeDlyHMSM(0, 0, 5, 0);
   convex_left_motor_speed = 0;
   convex_left_motor_direction = reverse;
   convex_right_motor_speed = 0;
   convex_right_motor_direction = reverse;
   OSTimeDlyHMSM(0, 0, 0, 500);

   convex_corner = true;

   convex_left_motor_speed = 4;
   convex_left_motor_direction = forward;
   convex_right_motor_speed = 0;
   convex_right_motor_direction = forward;
   OSTimeDlyHMSM(0, 0, 3, 500);
   convex_left_motor_speed = 3;
   convex_left_motor_direction = forward;
   convex_right_motor_speed = 3;
   convex_right_motor_direction = forward;
   while (distance[2] >= 20) OSTimeDlyHMSM(0, 0, 0, 10);
   //OSTimeDlyHMSM(0, 0, 5, 0);

   Clear_Motor_Flag_Registers();
   Reset_Motor_Counters();
}

else
{
   door_detected = true;
   while (distance[2] <= 20)
   {
      convex_left_motor_speed = 2;
      convex_left_motor_direction = reverse;
      convex_right_motor_speed = 2;
      convex_right_motor_direction = reverse;
```

```
               OSTimeDlyHMSM(0, 0, 0, 10);
            }
            convex_left_motor_speed = 3;
            convex_left_motor_direction = forward;
            convex_right_motor_speed = 3;
            convex_right_motor_direction = forward;
            while (distance[2] >= 20) OSTimeDlyHMSM(0, 0, 0, 10);
            //OSTimeDlyHMSM(0, 0, 2, 0);

            Clear_Motor_Flag_Registers();
            Reset_Motor_Counters();
         }
      }

      else convex_corner_or_door_behaviour_active = false;

      OSTimeDlyHMSM(0, 0, 0, 10);
   }
}

void Follow_Left_Wall(void *data)
{
   int left_o, right_o;
   int temp;

   data = data;
   OSTimeDlyHMSM(0, 0, 2, 0);

   while (1)
   {
      if (locked_onto_left_wall == true)
      {
         if (stop == 1)
         {
            follow_left_motor_speed = 0;
            follow_left_motor_direction = forward;
            follow_right_motor_speed = 0;
            follow_right_motor_direction = forward;
            left_wall_following_behaviour_active = true;
         }

         else
         {
            left_o = 3;
            right_o = 3;

            if (distance[6] < 12) { left_o += 1; right_o -= 1; }
            if (distance[6] > 11 && distance[6] < 17) { left_o -= 1;
right_o += 1; }
            if (distance[6] > 11 && distance[7] < 18) { left_o += 1;
right_o -= 1; }
            if (distance[6] < 12 && distance[7] > 17) { left_o -= 1;
right_o += 1; }

            if (distance[6] > 16 && distance[7] > 22) { left_o = 3;
right_o = 3; }
            if (distance[6] > 11 && distance[6] < 17 && distance[7] > 30)
{ left_o = 3; right_o = 3; }
```

```
            temp = abs(left_o);
            follow_left_motor_speed = (unsigned char)temp;
            temp = abs(right_o);
            follow_right_motor_speed = (unsigned char)temp;

            if (left_o >= 0) follow_left_motor_direction = forward;
            else follow_left_motor_direction = reverse;
            if (right_o >= 0) follow_right_motor_direction = forward;
            else follow_right_motor_direction = reverse;

            left_wall_following_behaviour_active = true;
         }
      }

      else left_wall_following_behaviour_active = false;

      OSTimeDlyHMSM(0, 0, 0, 10);
   }
}

void Follow_Right_Wall(void *data)
{
   int left_o, right_o;
   int temp;

   data = data;
   OSTimeDlyHMSM(0, 0, 2, 0);

   while (1)
   {
      if (locked_onto_right_wall == true)
      {
         if (stop == 1)
         {
            follow_left_motor_speed = 0;
            follow_left_motor_direction = forward;
            follow_right_motor_speed = 0;
            follow_right_motor_direction = forward;
            right_wall_following_behaviour_active = true;
         }
         else
         {
            left_o = 3;
            right_o = 3;

            if (distance[2] < 12) { left_o -= 1; right_o += 1; }
            if (distance[2] > 11 && distance[2] < 17) { left_o += 1;
right_o -= 1; }
            if (distance[2] > 11 && distance[1] < 18) { left_o -= 1;
right_o += 1; }
            if (distance[2] < 12 && distance[1] > 17) { left_o += 1;
right_o -= 1; }

            if (distance[2] > 16 && distance[1] > 22) { left_o = 3;
right_o = 3; }
            if (distance[2] > 11 && distance[2] < 17 && distance[7] > 30)
{ left_o = 3; right_o = 3; }

            temp = abs(left_o);
```

```
            follow_left_motor_speed = (unsigned char)temp;
            temp = abs(right_o);
            follow_right_motor_speed = (unsigned char)temp;

            if (left_o >= 0) follow_left_motor_direction = forward;
            else follow_left_motor_direction = reverse;
            if (right_o >= 0) follow_right_motor_direction = forward;
            else follow_right_motor_direction = reverse;

            right_wall_following_behaviour_active = true;
         }
      }

      else right_wall_following_behaviour_active = false;

      OSTimeDlyHMSM(0, 0, 0, 10);
   }
}

/************************************************************************
*******
*
*
*                                    Mapping Routine
*
*
*
************************************************************************
******/

void Mapping(void *data)
{
   unsigned long distance_travelled;
   int index;
   data = data;

   while (1)
   {
      if (map_exists == true)
      {
         OSTimeDlyHMSM(0, 0, 0, 10);
         continue;
      }

      if (concave_corner_complete == true)
      {
         distance_behind = distance[4];
         concave_corner_complete = false;
      }

      if (convex_corner == true || concave_corner == true ||
door_detected == true)
      {
         distance_travelled = Read_Actual_Position(LEFT);
         /*if (door_detected == true) distance_travelled =
distance_travelled + distance_behind + 8 - 32;
         if (concave_corner == true) distance_travelled =
distance_travelled + (distance[8] + distance_behind + 15);
```

```
         if (convex_corner == true) distance_travelled =
distance_travelled + distance_behind + 8;*/
         if (concave_corner == true) distance_travelled =
distance_travelled + (distance[8] + distance_behind + 15);
         if (convex_corner == true || door_detected == true)
distance_travelled = distance_to_start_of_convex_corner_or_door +
distance_behind + 8;

         if (convex_corner == true) node[num_nodes].type = convex;
         else if (concave_corner == true) node[num_nodes].type = concave;
         else if (door_detected == true) node[num_nodes].type = door;
         if (num_nodes != 0)
         {
            if (node[num_nodes - 1].type == door) distance_travelled =
distance_travelled + 32 - distance_behind - 8;
            if (node[num_nodes - 1].type == convex) distance_travelled =
distance_travelled - distance_behind - 8;
            node[num_nodes - 1].next = num_nodes;
            node[num_nodes - 1].distance_to_next = distance_travelled;
            node[num_nodes].previous = num_nodes - 1;
            node[num_nodes].distance_to_previous = distance_travelled;
         }

         node_type[count] = node[num_nodes].type;
         node_distance_to_previous[count++] = distance_travelled;

         //if (num_nodes > 4) { Write_Nodes_To_File(); stop = 1; return;
}

         if (num_nodes >= 4 && count == 2)
         {
              for (index = 0; index < num_nodes - 4; index++)
              {
               if (node_type[0] == node[index].type &&
                  node_type[1] == node[index + 1].type &&
                  (float)node_distance_to_previous[0] >=
(float)node[index].distance_to_previous*0.75 &&
                  (float)node_distance_to_previous[0] <=
(float)node[index].distance_to_previous*1.25 &&
                  (float)node_distance_to_previous[1] >=
(float)node[index + 1].distance_to_previous*0.75 &&
                  (float)node_distance_to_previous[1] <=
(float)node[index + 1].distance_to_previous*1.25)
                {
                  node[0].previous = num_nodes - 3;
                  node[0].distance_to_previous = node[num_nodes -
2].distance_to_previous;
                  node[num_nodes - 3].next = 0;
                  num_nodes -= 2;
                  //if (nodes_written == 0)
                  Write_Nodes_To_File();
                  //nodes_written = 1;
                  stop = 1;
                  return;
               }
              }
         }
         if (count == 2)
         {
```

```
            node_type[0] = node_type[1];
            node_distance_to_previous[0] = node_distance_to_previous[1];
            count = 1;
         }

         num_nodes++;
         convex_corner = false;
         concave_corner = false;
         door_detected = false;

         Clear_Motor_Flag_Registers();
         Reset_Motor_Counters();
      }

      OSTimeDlyHMSM(0, 0, 0, 10);
   }
}


void Write_Nodes_To_File(void)
{
   FILE *fptr;
   int index;
   char *types[3] = { "Convex", "Concave", "Door" };

   fptr = fopen("nodes.txt", "w");

   for (index = 0; index < num_nodes; index++)
   {
      fprintf(fptr, "%s NN=%d NND=%ld  PN=%d PND=%ld\n",
types[node[index].type], node[index].next,
                                          node[index].distance_to_next,
                                          node[index].previous,

node[index].distance_to_previous);
   }

   fclose(fptr);

   fptr = fopen("location.map", "wb");
   fwrite(&num_nodes, sizeof(int), 1, fptr);

   for (index = 0; index < num_nodes; index++)
   {
      fwrite(&node[index], sizeof(node[index]), 1, fptr);
   }

   fclose(fptr);
}

void Navigate(void *data)
{
   FILE *fptr;
   int index, destination_node;
   data = data;

   destination_node = 2;

   if ((fptr = fopen("location.map", "rb")) != NULL)
```

```
   {
      fread(&num_nodes, sizeof(int), 1, fptr);

      for (index = 0; index < num_nodes; index++)
      {
         fread(&node[index], sizeof(node[index]), 1, fptr);
      }

      fclose(fptr);
      map_exists = true;
   }
   else map_exists = false;

   while (1)
   {
      if (locked_onto_right_wall == true)
         destination_node = num_nodes - 1 - destination_node;

      if (navigation_behaviour_active == true)
      {
         if (convex_corner == true || concave_corner == true ||
door_detected == true)
         {
            current_node++;
            if (current_node == num_nodes) current_node = 0;
            convex_corner = false;
            concave_corner = false;
            door_detected = false;
         }

         if (current_node == destination_node)
         {
            stop = 1;
            return;
         }
      }

      OSTimeDlyHMSM(0, 0, 0, 10);
   }
}

void Localization(void *data)
{
   unsigned long distance_travelled;
   int num, index;
   int first, second;
   data = data;
   num = 0;

   while (1)
   {
      if ((convex_corner == true || concave_corner == true ||
door_detected == true) && current_node == -1 && map_exists == true)
      {
         num++;
         if (num == 1)
         {
            convex_corner = false;
            concave_corner = false;
```

```
            door_detected = false;
            continue;
         }
         distance_travelled = Read_Actual_Position(LEFT);
         /*if (door_detected == true) distance_travelled =
distance_travelled + distance_behind + 8 - 32;
         if (concave_corner == true) distance_travelled =
distance_travelled + (distance[8] + distance_behind + 15);
         if (convex_corner == true) distance_travelled =
distance_travelled + distance_behind + 8;*/
         if (concave_corner == true) distance_travelled =
distance_travelled + (distance[8] + distance_behind + 15);
         if (convex_corner == true || door_detected == true)
distance_travelled = distance_to_start_of_convex_corner_or_door +
distance_behind + 8;

         if (convex_corner == true) node_type[count] = convex;
         else if (concave_corner == true) node_type[count] = concave;
         else if (door_detected == true) node_type[count] = door;

         if (count != 0)
         {
            if (node_type[count - 1] == door) distance_travelled =
distance_travelled + 32 - distance_behind - 8;
         }

         node_distance_to_previous[count++] = distance_travelled;

         if (count == 2)
         {
              for (index = 0; index < num_nodes; index++)
              {
               first = index;
               second = index + 1; if (second == num_nodes) second = 0;
// wrap around to start
               if (node_type[0] == node[first].type &&
                  node_type[1] == node[second].type &&
                  (float)node_distance_to_previous[0] >=
(float)node[first].distance_to_previous*0.75 &&
                  (float)node_distance_to_previous[0] <=
(float)node[first].distance_to_previous*1.25 &&
                  (float)node_distance_to_previous[1] >=
(float)node[second].distance_to_previous*0.75 &&
                  (float)node_distance_to_previous[1] <=
(float)node[second].distance_to_previous*1.25)
               {
                  current_node = index + 1;
                  if (current_node == num_nodes) current_node = 0;
                  navigation_behaviour_active = true;
                  break;
               }
              }
         }

         if (count == 2)
         {
            node_type[0] = node_type[1];
            node_distance_to_previous[0] = node_distance_to_previous[1];
            count = 1;
```

```
         }

         convex_corner = false;
         concave_corner = false;
         door_detected = false;
      }

      OSTimeDlyHMSM(0, 0, 0, 10);
   }
}


void Search_For_Wall(void *data)
{
   int diff, temp;
   int index, dex;

   data = data;
   OSTimeDlyHMSM(0, 0, 2, 0);

   while (1)
   {
      if (locked_onto_left_wall == false && locked_onto_right_wall ==
false)
      {
         search_left_motor_speed = 3;
         search_left_motor_direction = forward;
         search_right_motor_speed = 3;
         search_right_motor_direction = forward;

         search_for_wall_behaviour_active = true;

         while (distance[8] > 15 && distance[6] > 15 && distance[7] > 15
               && distance[0] > 15 && distance[2] > 15 && distance[1] >
15) OSTimeDlyHMSM(0, 0, 0, 10);

         if ((distance[8] < 15 || distance[6] < 15 || distance[7] < 15)
&& (distance[7] < distance[1]))
         {
            search_left_motor_speed = 2;
            search_left_motor_direction = forward;
            search_right_motor_speed = 2;
            search_right_motor_direction = reverse;

            diff = distance[7] - distance[6];
            while ((diff < 1) || distance[8] < 18 || distance[0] < 18)
            {
               diff = distance[7] - distance[6];
               OSTimeDlyHMSM(0, 0, 0, 10);
            }

            search_left_motor_speed = 0;
            search_left_motor_direction = forward;
            search_right_motor_speed = 0;
            search_right_motor_direction = forward;
            OSTimeDlyHMSM(0, 0, 1, 0);
            locked_onto_left_wall = true;
         }
```

```
         else if ((distance[0] < 15 || distance[2] < 15 || distance[1] <
15) && (distance[1] < distance[7]))
         {
            search_left_motor_speed = 2;
            search_left_motor_direction = reverse;
            search_right_motor_speed = 2;
            search_right_motor_direction = forward;

            diff = distance[1] - distance[2];
            while ((diff < 1) || distance[8] < 18 || distance[0] < 18)
            {
               diff = distance[1] - distance[2];
               OSTimeDlyHMSM(0, 0, 0, 10);
            }

            search_left_motor_speed = 0;
            search_left_motor_direction = forward;
            search_right_motor_speed = 0;
            search_right_motor_direction = forward;
            OSTimeDlyHMSM(0, 0, 1, 0);
            locked_onto_right_wall = true;
            if (map_exists == true)
            {
              for (index = 0; index < num_nodes; index++)
              {
               temp_node[index].type = node[num_nodes - 1 - index].type;
               //dex = num_nodes - 2 - index; if (dex == -1) dex =
num_nodes - 1;
               temp_node[index].next = node[index].next;
               dex = num_nodes - 2 - index; if (dex == -1) dex =
num_nodes - 1;
               temp_node[index].distance_to_next =
node[dex].distance_to_next;
               temp_node[index].previous = node[index].previous;
               dex = num_nodes - index; if (dex == num_nodes) dex = 0;
               temp_node[index].distance_to_previous =
node[dex].distance_to_previous;
              }

              for (index = 0; index < num_nodes; index++)
              {
               if (temp_node[index].type != door &&
temp_node[temp_node[index].next].type == door)
                  temp_node[index].distance_to_next -= 32;
               if (temp_node[index].type == door &&
temp_node[temp_node[index].next].type != door)
                  temp_node[index].distance_to_next += 32;

               if (temp_node[index].type == door &&
temp_node[temp_node[index].previous].type != door)
                  temp_node[index].distance_to_previous -= 32;
               if (temp_node[index].type != door &&
temp_node[temp_node[index].previous].type == door)
                  temp_node[index].distance_to_previous += 32;
              }

              for (index = 0; index < num_nodes; index++)
              {
               node[index].type = temp_node[index].type;
```

```
                node[index].next = temp_node[index].next;
                node[index].distance_to_next =
temp_node[index].distance_to_next;
                node[index].previous = temp_node[index].previous;
                node[index].distance_to_previous =
temp_node[index].distance_to_previous;
              }
            }
         }
      }

      else search_for_wall_behaviour_active = false;

      OSTimeDlyHMSM(0, 0, 0, 10);
   }
}
```

APPENDIX 5

Published Papers

1. Leyden, M., Toal, D., Flanagan, C., 1999. "A Fuzzy Logic Based Navigation System for a Mobile Robot" Proceedings of 2nd Wismar Symposium on Automatic Control, Germany.

2. Leyden, M., Toal, D., Flanagan, C., 2000. "Pitfalls of Simulation for Mobile Robot Controller Development" MIM 2000, Greece.

3. Leyden, M., Toal, D., Flanagan, C., 2000. "An Autonomous Mobile Robot Built to Investigate Behaviour Based Control" Mechatronics 2000, Atlanta.