An Autonomous Navigational Mobile Robot

By: Jay Kraut

A Thesis

presented to the University of Manitoba in partial

fulfillment of the requirements of the degree of Bachelor of Science In the

Department of Electrical and Computer Engineering

University of Manitoba

Winnipeg, Manitoba, Canada

Thesis Advisor: W. Kinsner, Ph. D., P. Eng.

March 7, 2002

An Autonomous Navigational Mobile Robot

By: Jay Kraut

A Thesis

presented to the University of Manitoba in partial

fulfillment of the requirements of the degree of Bachelor of Science In the

Department of Electrical and Computer Engineering

University of Manitoba

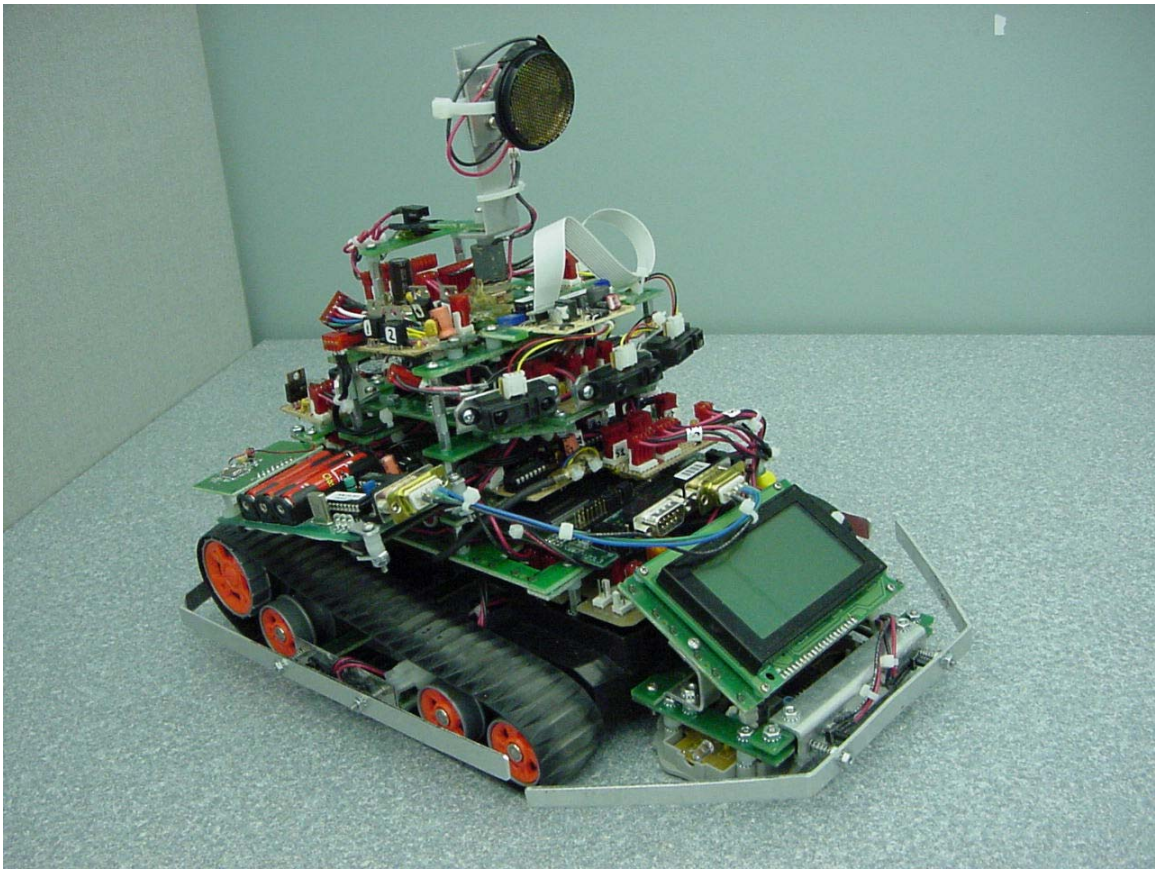
Winnipeg, Manitoba, Canada

Thesis Advisor: W. Kinsner, Ph. D., P. Eng.

March 7, 2002

pp. xi + 58

Visual Abstract



Abstract

There has been some success with autonomous robot navigation in indoor environment over the past few years, but all of the success has come at the price of expensive robot hardware. Even an entry level navigational robot costs well into the thousands of dollars. This paper explores the feasibility of autonomous navigation with a low cost robot. To keep construction costs down, the mobile robot is constructed with inexpensive sensors. A simulator, which is an absolute necessity, is used to develop the AI. A topological map making and navigation approach is used which compensates for the limited sensor arrangement. The robot is tested in an indoor office environment measuring 24 by 16 meters. In the simulator, the robot perfectly maps out and navigates in the area. In the real environment problems occur with the primary sensors. Despite these problems, the robot is still is able to generate a map. Navigating once the map is created, the easier of the two tasks, is reliable as long as the more accurate map that is generated in the simulator is used. This paper shows that despite the limited amount of perception that the sensors provide to the robot it is still possible to perform navigation, and map making.

Acknowledgements

I would first like to thank my advisor, Dr. Kinsner for allowing me pursue a topic on mobile robots, which has long been an interest of mine. I would like to thank Vansco Electronics for teaching me about embedded system design during my sixteen month internship. Furthermore, I wish to acknowledgment the help from many of the engineering staff at Vansco that went who of their way to help me in the first few stages of development. Vansco Electronics also allowed me to stay on location giving me access to their equipment and stockroom, which was invaluable and I am grateful for. I would also like to thank Infineon Semiconductor for donating a 167 evaluation board which serves as the processor of the robot. Lastly, I would like to thank Tyco for giving me samples of the hard to find .1" narrow body headers that serve to connect the robot's electronics to the 167 evaluation board.

Table of Contents

List of Figures

List of Tables

Chapter 1
Introduction

1.1 Introduction

Although the concept of an autonomous navigation mobile robot, one that can travel between two points without any intervention is a simple one, it has yet to be accomplished in a general sense. The majority of current research with these robots is concentrated on an indoor environment with a relatively uniform hallway and room structure. This is the only environment that has consistently yielded successful results because the indoor environment is suited to the perception of an autonomous robot, which is limited by the capability of its sensors. The ability to navigate successfully in indoor environments at a limited sense has been achieved [TBBF98], but only with the cost of expensive hardware. Even an entry level navigational robot, costs well in the thousands of dollars. It is worth exploring a low cost alternative.

1.2 Motivation

The potential usefulness of robots that have the ability of autonomous navigation makes research in this area a worthy field to pursue. With autonomous navigation, self-driving vacuum cleaners and lawn mowers could replace today's self-propelled ones. Even autonomous transportation vehicles could come into being. Although all of the potential is well known, there is not enough ongoing research in this field. The cost of a robot development system remains relatively expensive, due to its low volume, which can be circularly attributed to it high cost. Even though this project strives for results already accomplished by expensive robots, it is a worthwhile exercise to prove that reasonable goals can be accomplished with a low cost robot. This may encourage additional research in this field.

1.3 Goals and Objectives

Even though the goal of an autonomous robot travelling as well as a human in any environment is along time away from occurring, it is possible to develop a robot that can navigate in a restricted environment. The extent to which the environment is restricted, relates to the capability of the robots hardware. That being said, the hardware of the robot consists of inexpensive sensors, which places addition restrictions on the indoor environment in which the robot can navigate. The environment must contain only walls or flat objects at perpendicular or parallel angles to each other. As can be expected, all objects must be detectable with the sensors. The general

layout of the environment must be office-like, consisting of hallways that connect rooms. The floor must be of even grade and must not contain any obstructions. In addition, the environment must be fully static. Even with these restrictions, a large number of office areas fit the requirements. To be able to perform efficient navigation the robot must first create an accurate map. Once the map is created, the robot is told its initial position and then should be able to reliably travel along the map to any location. All of this has to be accomplished at the relatively inexpensive cost of a few hundred dollars.

1.4 Scope

To be able to accomplish navigation with a low cost robot, a robot, a simulator and an AI have to be created. The cost of construction of the robot should not exceed a budget of a few hundred dollars. After the robot is constructed, an application program interface (API) has to be created over a rudimentary operating system that allows software to be developed for the robot. A simulation system has to be created to allow the development of the artificial intelligence (AI) on a personal computer (PC), using Windows ME rather then the actual robot. The simulator should emulate the robot by implementing all of the API functionality. A robust AI then has to be created on the simulator that is immune to the various realities encountered in a real, not simulated environment. The robot with AI is to be tested in a real environment. The overall difficulty is not in the individual creation of each one of the three tasks, but rather in having all of the three parts working together. Even a small problem with the hardware, or the simulator or the AI could cause the robot not to be able to navigate.

1.5 Organization of the Thesis

The thesis requires three distinct tasks to be accomplished to meet the goal, which suggest the logical separation of the thesis into three key sections. Chapter 2 contains information on the development environment, which includes the simulator. It also introduces the API used to program the robot that was first developed out of need when writing the first AI for the simulator. Chapter 3 contains details of how the robot was constructed, and a brief description of the various electronic components. Chapter 4 explains the reasoning behind the selection of the chosen AI, and explains the AI design and how it was implemented. Because each of the above chapters are quite distinct from each other, they all have an introduction which serves as the background for the information contained in the chapter. Results are contained in chapter Chapter 5, and the conclusion in Chapter 6. There is no appendix because of the volume of code. A CD-ROM is included with the thesis that contains all of the code, a chronological picture library of the robot

being built, the schematics, and the video of the robot. It is strongly recommended to view the picture library when reading of Chapter 3. It is also recommended to view the video of the robot working when reading Chapter 5.

Chapter 2
The Robot Development Environment

2.1 Introduction

Because it is exceeding difficult to develop a large program like a robot AI on an embedded system, it is necessary to have a development environment for the AI. The development environment consists of three programs, a simulator, and a map editor, which creates maps for the simulator, and a monitor program once the AI is ported to the robot. A simulator reduces development time in the following ways. First, the robot does not have to be available to test the software. Second it is much easier to track bugs in the code by using a debugger available with a windows compiler, rather then only being able to debug code by monitoring error flags through the LCD or serial output. Lastly, the entire process of compiling code and being able to run it right away is much quicker than having to load the flash (the read only memory) on an embedded processor.

One of the major difficulties of using a simulator is that the code has to be portable for it to be used in the embedded system of the robot. Not only does the code need to be able to compile under the embedded system compiler, but the AI also has to produce the same result. To solve this problem a common application program interface (API) was created. The API is a common set of functions that are implemented both in the simulator and in the embedded system of the robot. Other then the API functions calls, the AI code is of standard C code which makes use of only the standard set of libraries.

Even with a simulator it is still necessary to have the ability to monitor the AI in order to debug a potential problem. A robot monitor must be developed. This program receives input through the wireless connection with the robot, and then represents this information graphically on a PC.

2.2 The Map Editor

Although it is possible to draw and then exports maps using a commercially available drafting package such as AutoCAD, it is necessary to create a map editor. The simulator requires a data structure to store the objects of the map. This data structure has to be fully tested before any simulations are run. By creating a map editor with a reusable data structure, the structure is fully

tested by the time it is imported to the simulator. Also, a custom map editor, while having less functionality overall, it does have features that are specifically targeted for the simulator. These features are not present in drafting packages such as AutoCAD.

2.2.1 The Quad Tree

The key to the map editor is a data structure called a quad tree. A quad tree is used to store objects based on location, but does so in a way that makes the retrieval of objects efficient. The root of a quad tree represents the entire area of the map. The root is divided into four quadrants, each of which is divided repeatedly into quadrants, until the base level of the quad tree is reached. A quad tree is a general structure, which has an arbitrarily size. For the map editor the quad tree contains $128m^2$ of area, which is deemed sufficient for simulation purposes. The quad tree spans seven level deep, to give the base quadrants both a length and height of one metre.

The principle advantage of a quad tree is its fast and efficient object retrievals. In order to access the map for collision detection or rendering, it is advantageous to only retrieve the objects which are in the area required. This is more efficient than accessing all of the objects, and then choosing which ones are needed. The property of a quad tree that makes it efficient to access objects is that each depth level stores a link to any object in its encompassing area. When drawing the map, a recursive search is done to see which quadrants need to be accessed. The search starts at the root node and searches downward until the searched quadrant is completely contain in the search area, or the base quadrant is reached. The size of the base quadrant allows some objects to be retrieved that are not needed, but this inefficiency is minor compared to having to access all of the object in the map.

2.2.2 Other Architecture Details

The map editor is based on using only one primitive object, the line. The advantage of only using lines is that the simulator collision detection only has to solve one set of linear equations. In addition, limiting the type of object to lines makes it easy to render the map in 3D, which is based on drawing triangles. Objects that are built out of lines can easily be converted into walls with height by converting the line into two triangles.

Other then the quad tree the map editor contains one other innovation, the group pointer. The purpose of a group pointer is two-fold. It serves as an interface for different object types, and when drawing it allows for multiple grouping hierarchies. The structure of group pointers is

maintained throughout the map editor, which allows the group hierarchy to be maintained in all functions including the save/load and copy/paste functions.

The map editor is intended to only have two objects, the line and the arc, the arc being an inseparable group of lines. Unfortunately, when developing the simulator, it was necessary to add more objects. The map editor now consists of the line, the arc, three sensor objects, (bumper sensor, sonar, and IR), a robot initial placement object, and two other objects, which support 3D roofs. When developing the simulator, it was convenient to reuse the map editor to draw a representation of the robot used for the simulations. Thus, the three sensor objects were added. The robot initial placement object was later added in order that the robot could start in the same position when the simulator loads a map. Two other objects were necessary to support 3D roofs, the purpose of which are explained in the add depth function section. Each addition of an new object type was time consuming. In retrospect, it would have been useful to have created a base object class that could be extended to support any kind of object. Extending a class is much easier than working through functions and having to add "if statements", to support each new object type.

2.2.3 Drawing Functionality

In addition to serving as a platform to create and test the quad tree, the map editor has to have a user interface. During development of the map editor, time had to be rationed. The simulator could not inherit any of the user interface from the map editor, as such any time spent on adding the user interface had to be justified by saving time when actually drawing the maps. The following (table 2-1) are the features that were deemed necessary to incorporate into the map editor to provide it with a reasonable map editing capability.

Drawing Functions	Draw Line	Draw Arc					
Editing Functions	Cut	Copy		Paste		Undo	Redo
	Move	Rotate		Group		UnGroup	
File Access	Save	Save As		Load		Save, to all lines format	
Model Functions	Sonar	IR Sensor		Bump Sensor		Robot Placement	
Others	Add Depth	Print	Zoom	Pan	Snap (Grid, Polar, EndPoint)		

Table 2-1 Map editor functionality

2.2.4 The Add Depth Function

One feature that the map editor has and that a commercial program like AutoCAD does not is the add depth function. The function extrudes a group of connected lines outwards or inwards, giving an appearance of depth. This function exists to add depth to walls for the 3D version of the simulator. It also adds the information about the wall and corresponding roof, by using a triangle object and a container object, so the roofs can also be rendered in 3D. The add depth function was the most difficult function to implement in the map editor, but the results in 3D justified the time spent.

2.3 The Simulator

2.3.1 Introduction

A simulator has to provide an accurate platform to test the code that is destined for the robot. This requires each component of the robot targeted for simulation to be modelled, and to be reproduced as accurately as possible. Across the simulation, various approximations are made to model the robot. Even with the approximations, the simulator created does an adequate job on accuracy. There are however, a few problems with simulator and these are presented in the difficulties section.

2.3.2 Basic Design

To save time, the simulator and the map program were developmed in tandem with the core of the map editor, the quad tree, forming the core of the simulator. To further save time both the model of the robot and the sonar beam shape were generated by the map editor. Another advantage of reusing the map editor is that it is possible to change the shape of the robot and the shape of the sonar beam by simply altering the model and beam files in the map editor.

Most of the simulator is programmed in C++, and thus is designed using an object-oriented methodology. There is a base robot class, which is responsible for the movement of the robot and the collision detection. Each sensor has its own class, which are children to the base robot class. The simulation is rendered in two ways. Either the Windows GDI (Graphics Device Interface) is used, or DirectX 8.0 (Microsoft's 3D hardware driven API) is used, which allows the simulation to render in 3D if the computer being used has a video card that has 3D hardware support. Each of these options has their own class which makes it possible to switch which rendering is used during run time.

The core of the simulator is the same quad tree class as the map editor. This class is inherited by a new class that, in addition to providing the same rendering functionality as the original quad tree also provides two kinds of collision detection. The two kinds of collision detection are ray based and polygon based. The ray based collision detection finds out the first collision with a ray. This collision detection is used for the IR rangefinders, which could be approximated as a ray without any width to it. The more complex collision detection is polygon based. It is used for both the robot and the sonar collision detection. The algorithm used is far more complex then just checking if the future position of the polygon intersects with an object in the map. If a simple intersecting rectangles scheme is used, it is possible for the sonar beam to pass through an object undetected, if its current position and future position do not intersect the object. To prevent this, the collision detection checks the entire path of the moving object against any potential object in its path. One minor approximation occurs when the robot rotates. The polygon collision detection algorithm only works with linear paths, so the rotation movement is approximated by using piecewise linear movements. In theory this is not a perfect scheme, but in practice, the approximation does not observably affect the accuracy of the collision detection.

2.3.3 3D Rendering Support

To render in 3D is actually quite simple because the simulator is designed to do this in the first place. As mentioned in the map editor section, the basic primitive in the map editor is a line, with the one circular object, the arc, being represented by a group of lines. 3D rendering is based on rendering triangles, and it is easy to give height to a line by forming two triangles. Both the IR rangefinder and Sonar visual output were similarly extruded from their 2D counterparts. Drawing the robot in 3D is simple too. The robot model is drawn in 3D Studio Max 4.0, from which DirectX 8.0 has support to import a mesh. Other then forming an arc out of lines, the only other 3D support required, is rendering the roofs of walls and the tread visualization. It looks better in 3D if walls have depth, something that is not entirely necessary for simulation. Without adding roofs to the wall, the effect of having walls with depth does not look quite right. The add depth function in the map editor, generates the triangle object necessary to draw the roofs correctly. The only other 3D issue is the treads. It is too time consuming to animate the treads in 3D Studio Max and import the animation into the simulator. It was easier just to add a 3D treads effect. This effect adds the feel that the robot is actually moving.

2.3.4 Multi-Threaded Architecture

The simulator uses a multi-threaded architecture. The simulation and the rendering of the simulation are on two different threads. In theory, this makes the simulator run more efficiently, because when either the simulation or the rendering threads needs resources, each process could sleep and let the other one run. The multi-threading, was originally created to solve the problem of run time regulation. In order to maintain accuracy the simulation code has to be executed many times per second. This is because the simulator uses piecewise linear approximation in several of its routines which require the simulation to run many times a second to be accurate. In addition, the robot AI must be run at the rate that the primary sensor refreshes so the AI can make timely decisions. When running the simulation it is ideal to have the individual simulation runs be spaced out as evenly as possible. Running the simulation at 49 times at the start of the second, and once at the end would not produce accurate results. To facilitate even run times, both the simulation and rendering threads are regulated to execute at certain intervals by sleeping for a certain amount of time. Using two threads, allows the rendering and simulation to be run at different rates per second. This is advantageous when running the simulator at a multiple of real-time causing the number of simulations per second to be in the hundreds. At any given time, there is no benefit or rendering faster then 50 times per second, so having them in the same thread would cause the rendering to waste time.

It could be argued that the simulation should be run as fast as the computer could support. However, running the simulation as much as possible causes the computer's response time to slow with respect to user commands, and does not increase the overall accuracy of the simulation. It also could be argued that having two threads is a waste, when it might be possible to regulate both the rendering and simulation at different rates in one thread. This argument does hold some validity. The plan originally was to have both the rendering and simulation in one thread, but it proved difficult to regulate the time properly. However, it might have been worth spending extra time working on using only one thread. Because the simulation and rendering are on two threads, various synchronization problems occurred that had to be solved and took up time to solve. The solutions to these problems did not use semaphores because the Windows operating system is too inefficient in this regard, so other methods had to be devised. The only advantage of multithreading is greater efficiency in scheduling. However, if originally single thread speed regulation worked, before the multithreading solution, time would not be spent working on multithreading for this one advantage.

2.3.5 Difficulties Creating the Simulator

Due to the fact that the simulator inherited the quad tree class from the map editor, most of the simulator implementation was straightforward. There are several problems though that could not be solved in the short amount of time available to implement the simulator. When performing the robot collision detection after a collision, the robot should be moved to the collision point, and be tangent to the object. Finding the collision point is easy, but moving the robot to the collision point proved to be difficult. In order to move the robot tangent to the collision point, the collision point on the robot has to be found. Then the displacement between the collision point of the robot to the collision point on the colliding object has to be translated back to the robots centre, which is reference point used when moving the robot. This is solvable when the robot is moving straight, however when the robot is rotating and moving both the displacement and the rotation have to be solved. After studying the problem, the decision was to work around this problem. After the attempting to move the robot to the collision point, a collision detection scheme is run to make sure the robot does not intersect an object. If it does, the robot is moved back to its pre-collision position. In practice, this work around is not noticeable, because of several factors. First, the AI is designed to avoid collisions, so they do not occur that often. Second, the simulation is run many time a second, and the robot is generally moving slowly, so even if this error does occur, and the robot is moved back to it pre-collision position, it is not noticeable.

A similar problem happened with the sonar beam, which uses the same collision detection scheme as the robot's movement. The problem was not with the rotation, but with the fact that the beam expanded along its path. To move the sonar beam to be tangent to its collision point for rendering purposes, it is necessary to solve both its position and its expansion. After trying and not being able to solve this problem in a reasonable amount of time, the following solution was used. The beam expansion is not recalculated at the collision point, rather the previous beam expansion is used. To compensate for any inaccuracies in the rendering of the sonar beam, the sonar simulation is run more often then the robot simulation, which decreases the error. Even with this "solution" it is sometimes possible to see errors in the visualization of the sonar beams collision with objects, however it does not change the distance calculation.

There is a major problem with the sonar model used. In reality the sonar model is very complex, containing not only beam expansion, but also a complex reflection model explained in [BiTu95]. A sonar is a time of flight sensor, which the return time of a sonar beam is measured, and using the speed of sound constant, distance is derived. However, once a sonar beam hits an object the

sonar beam may reflect back to the detector, or it may reflect in another direction, then eventually travel back to the detector. In addition, there is the potential for objects to absorb the sonar beam, so the beam may never be reflected back to the detector. The sonar model used only takes into account the expanding beam property and is not very accurate. This problem is solved by not using sonar distance directly for important calculations, rather converting the distances to pathway information at the API level.

The proper way to model the optical mouse could not be found. The optical mouse is placed in front of the robot, in the centre between the two treads. To model the optical mouse, an X and Y displacement would have to be generated corresponding to the robot's movement. When the robot's movement consists only of a rotation, the only output of the mouse is a Y displacement. Moving the robot forwards and backwards causes only an X displacement. Moving the robot with both a displacement and a rotation causes both an X and Y output to occur. The model for simulating this so could not be solved or found in the time available. Another problem with using the optical mouse is that if the for translating the X and Y displacement to robot displacement and rotation were complex, it would be difficult to implement from real time processing on the more limited robot processor. As such, the full potential of the optical mouse to be able to precisely tell the displacement and rotation of the robot at all times could not be used.

2.4 The API

2.4.1 Introduction

The AI is required to work in two different platforms, on the simulator on a PC and on the embedded system on the robot. In order to be able to write code on the simulator it is necessary to have a common API to access the robot specific functionality. The ability to flawlessly port the AI from the simulator to the robot has a lot to do with how accurate the API on the simulator resembles the actual robots implementation. But since the API was first developed out of need when writing the first AI on the simulator, the portability depends on how close to the ideal set in the simulator, is the API implemented in by the robots operating system described in the next chapter.

2.4.2 Kernel API Functions

unsigned short KRNLGetTicks()

This function returns a number from 0 to 999, which represents a kernel tick. Each tick is one millisecond. This function is primarily used to limit the amount of times the AI runs per second on the robot, to the amount of times it is run on the simulator per second.

unsigned long KRNLGetTicksL()

This function returns an unsigned long, which represents the amount of ticks since the robot was initialized. Again each tick is one millisecond. This function is used when logging when the side IR rangefinder change values. The amount of time elapsed for a change in distance, helps the follow wall module decide how to adjust the robot to keep it parallel to a wall.

2.4.3 Movement API Functions

void MVGoFoward(char Speed)

This function tells the robot to move straight at a speed factor from 0 to +/-9, either forward or backwards depending on the sign. The output speed is the speed factor multiplied by 5 cm/s on the simulator.

MVGoFowardWA(char speed, char angle)

This function causes the robot to move forward with a rotation angle. This function is used when making slight adjustments to the orientation when trying to move parallel to a wall. The speed variable is the same as the above function. The angle is from 0 to +/- 9, which represents +/- 10 degrees rotation per second, on the simulator.

MVGoFowardT(char Speed, short Ticks)

This function is the same as MVGoFoward(char speed) except the MVisBlocking() call returns true until the amount of ticks specified is traveled. Note, each “tick” is a tick from the shaft encoder which is around 1.2 cm

void MVAllStop()

This function stops the robot, dead stop, no PWM deceleration

void MVTurn(short Ticks)

This function has the robot rotate on its axis. A positive tick count means a counter clockwise direction, a negative tick counter means a clockwise direction. The MVisBlocking() returns true until the robot travels the number of ticks specified by the ticks variable.

BOOL MVIsBlocking()

This function returns true if some movement is occurring that cannot be interrupted. This could either be turning a certain amount of degrees or moving straight a certain amount of ticks.

void MVReset()

This function clears any blocking calls, by resetting the MVisBlocking() to false. It is useful if a collision occurred on a blocking action.

2.4.4 IR Rangefinders API Functions

Front IRs for collision detection

unsigned char IRGetFrontD() Placement at 0 degrees

unsigned char IRGetRFrontD() Placement at + 15 degrees

unsigned char IRGetLFrontD() Placement at – 15 degrees

Side IRs for wall following

unsigned char IRGetLeftD() Placement at – 90 degrees

unsigned char IRGetRightD() Placement at 90 degrees

These functions return the distance in centimetres from the most recent IR reading. The distance returned could be from 10 to 120 in cm or 0xff if no detection is present. In the simulator, a noise model is developed. Depending on the noise settings, this adds some randomness to the output of these functions. The noise model increases the magnitude of the distortion the greater the distance reading is, which is what happens to these sensors in reality.

2.4.5 Bump Sensor API Function

char BMPGetByte()

This function returns one byte which represents the status of the bumper sensors.

Bump Sensor	Bit altered
RIGHTFRONT	0x01
RIGHTBACK	0x02
REARRIGHT	0x04
REARLEFT	0x08

LEFTFRONT	0x10
LEFTBACK	0x20
FRONTRIGHT	0x40
FRONTLEFT	0x80

Table 2-2 Mapping of the bump sensor output

2.4.6 Shaft Encoder API Functions

These functions returns and resets two independent tick counters, to track distances. Each tick represents around 1.2 cm

short SEGetTick1() Returns the amount of ticks from the first counter
void SEResetTick1() Clear the first tick counter
short SEGetTick2() Returns the amount of ticks from the second counters
void SEResetTick2() Clear the second tick counter

2.4.7 Sonar Sensor API Functions

BOOL SNRIsBlocking()

This function returns true if some sonar command is not completed, and regular execution should not proceed. It is only true if the sonar is currently performing a 3-point scan.

void SNRReset()

This function resets and zeros the sonar, clears any blocking calls.

void SNRSetFowardScan()

This function sets up the sonar to scan continuously at 0 degrees relative to the robot. This is the default setting of the sonar.

*BOOL SNRGetFowardDistance(short *Distance)*

This function returns the distance of the last forward scan in cm. It returns true if the distance is valid, and false otherwise. The sonar distance is from 15 to 1000 cm.

void SNRSet3PointScan()

This function sets up the sonar to scan for the type of pathways of the three directions (forwards, left, right). While scanning, SNRIsBlocking() returns true.

*void SNRGet3PointDistance(short *distance[])*

This function returns an array of the type of pathway encountered at each of three directions, in the following order: right, forward, left. The possible types are hallway, wall and unknown. Unknown usually means a room, but it might also be a short hallway. This function solves the problem of the differences between the simulator and actual sonar model.

2.4.8 Miscellaneous API Functions

void Beep(short numofbeeps, short duration)

This function activates the piezo buzzer accordingly. This function call is similar to the windows system beep call.

void SendNode(unsigned char Node)

This function sends over the information about a node to the robot monitor that is monitoring the robot's progress. This function is only implemented in the embedded system, because it is not required in the simulator.

2.5 The Robot Monitor

During the early days of development, a robot monitor program was developed to properly test the hardware. The job of the robot monitor is to receive data through the serial port and display it to the screen. The earliest version of the monitor contained two big gauges representing how fast the motors were turning. Later versions contained visualizations for all of the sensors. The latest version created is used to debug the AI. It receives current location, and status of the robot. It also contains the same class used in the simulator to display a graph (the map). The graph is used during navigation to communicate with the robot, by relating a position on the graph to a node number, which the robot understands. When creating maps, the AI also sends over information on any new nodes created.

Chapter 3
The Robot

3.1 Background

3.1.1 Introduction

With the advent of cheap processor boards and sensors the popularity of hobbyists building robots has increased over the last few years. Another reason for the increase is the book "Mobile Robots" [JoFl93], that explains how to build a robot on a low budget. The rug warrior robot kit from that book is still one of the least expensive robot kits available. Other then the rug warrior kit, there are several companies producing micro-processing boards for only around $100 that are ready to be used out of the box. More sensors are becoming available, especially infrared (IR) sensors. A few years ago, Sharp introduced a new generation of IR rangefinders that can mesure distance and are inexpensive. In addition, pre-built sensor modules such as digital compasses and single camera computer vision are now available. These modules operate independently of the main processor and are interfaced through standard serial interfaces. Because of the small boom in robot hardware, it has become easier to build a robot that can exhibits simple behaviours. However, much of the new hardware is intended for the hobbyist, and is insufficient alone to construct a navigational robot.

Over the past few years, as companies started to serve robot hobbyist, former researches have started companies that build robots. These researchers had previously been in the forefront of robot development, and are now providing state of the art robots to the research community. In the past, state of the art robots had to be custom built. This is even more expensive than buying one of the robots available today. Picture of increasingly more sophisticated and expensive robots (left to right) are displayed in figure 3-1.

Figure 3-1 Currently available robots, from left to right ordered according to increasing price and capability, the Rug Warrior, ActiveMedia Pioneer-DXe, iRobot Magellen, iRobot B21

The high-end research robot iRobots B21 costs $45000 US. The base model of the robot contains 48 sonar in two arrays, and 24 IR Rangefinders in one array. It has a Pentium class processor. The platform also support accessories such as laser rangefinders and computer vision systems. A more affordable robot is the ActiveMedia Pioneer-DXe, with the Siemens 166 processor, and 8 sonars, at $3495 US. The Pioneer-DXe also allows for accessories and comes with a nice assortment of software. The lowest cost kit available is the rug warrior. For around $600 the kit includes a Motorola HC11 class processor, and its principle object detector is a lowly Sharp IR detector/emitter with a range of 15 cm.

The goal of the hardware design is to surpass the capabilities of the rug-warrior and have capabilities as close to as possible to mid-range research robots such as the ActiveMedia Pioneer-DXe. Although this robot costs in the thousand, for the most part the robots individual components can be bought for a reasonable amount of money.

3.1.2 Micro-Controller

The Micro-Controller used is the Siemens 167 [Siem96], in evaluation board form has been donated by Infineon Semiconducter. This is a high-end micro controller which is of the same family as the one used in the Pioneer-DXe. It runs at 20 MHz and features a pipeline architecture that allows it to run up to one instruction per cycle. While the processor allows up to 16 Megs of memory to be addressed, the evaluation board contains 256 Kbytes of flash read on memory, and 128 Kbytes of RAM. The 167 has 144 pins of which 59 are available for I/O. The 167 contains many hardware modules, such as a 16 channel 10 bit A/D, 5 general timers, a 4 timer 32 register capture compare module, both a synchronous and an asynchronous serial module, 4 channel

PWM (Pulse Width Modulation) Unit, and a CAN (controller area network) module that is not used.

3.1.3 Object Detection Sensors

If cost was not an issue, it would has been possible to buy object detection sensors that have a long maximum range and good accuracy. A laser rangefinder exhibits these properties at a cost of several thousands of dollars. Another expensive sensor that has these properties is a stereo computer vision system, which requires two cameras, and a high end processor that segments the video input into object detection data. Unfortunately, cost is an issue, and the two affordable sensors that are available are a sonar class of sensor, and an IR class of sensor. These sensors have problems with both maximum range and accuracy.

The sonar sensor is a time of flight sensor. It sends out an acoustic beam, which reflects off objects. The distance to an object is determined by measuring the return time of the beam to the robot, and multiplying it by the speed of sound constant divided by two. The most popular of these sensors is the Polaroid sensor kit. This kit comes with the transducer and all of the associated electronics for around $70. The transducer acts as both the emitter and the detector, which leads to a potential ringing problem. Ringing occurs when acoustic energy remains on the transducer after a beam was emitted which could subsequently cause an erroneous detection. The solution is to blank the sensor. This means to not accept any input for a certain amount of time. This limits the sensor to having a minimum distance of 15 cm, which corresponds to its blanking time. The acoustic beam looses its energy the farther it travels to a point where the reflected beam is no longer detectable. This corresponds to its maximum distance of 10 metres.

Figure 3-2 The sonar transducer used in the robot

If the sonar could emit a beam that has a narrow width and reflects straight back to the transducer, the sonar would be a great sensor. However, in reality the sonar beam spreads out at around a 15-

degree angle [BiTu95]. In addition, when the sonar hits an object, the return path depends on the properties of the object hit, and the angle of the object relative to the beam. It is possible for a sonar beam to bounce off an object at an angle that will either take a longer route to return to the transducer, or even not return to the transducer at all. This means that the readings may not represent the actual distance to the nearest object in the path of the beam. The fact that the sonar beam spreads out means that down a straight hallway, the sonar will register a distance relating to the point where the beam expanded and hits the wall at the side of the hallway. To alleviate the inaccuracies of the sonar, multiple readings can be taken at fixed intervals, either by having the sonar transducer rotate around the robot, or by having multiple sonar transducers arrayed together, which could be seen in the robots in figure 3-1. The output of all of the sonar readings can then be fused together, to provide a more accurate picture of any object. However, using multiple sonars is too expensive for this project, so instead the sonar transducer is mounted on a stepper motor, which allows the transducer to be rotated around the robot.

The IR class of sensors on the other hand has a very small beam width of only a few centimetres. The best available IR sensors are from Sharp. The IR used is the GP2D12 produces a non-linear analog voltage output depending on distance as seen in figure 3-3. Since the output of the sensor is non monotonic at 10 cm, this requires that the sensor must never be within 10 cm of an object. Because of this the sensor is mounted at least 10 cm away from the robots bumpers. The range is limited to around 120 cm, which becomes more inaccurate as the distance increases towards the limit of the range. This is due to the fact that there is electrical noise present in the system. The electrical noise becomes more of a nuisance, as the difference of voltages becomes less as the distance becomes greater. Another problem is that the sensor is calibrated for white paper. Object that are of a darker material, or have different reflective properties could cause erroneous output voltages. Despite these limitations, these sensors are still the least expensive at $13.50 each and are the primary sensors used in the robot

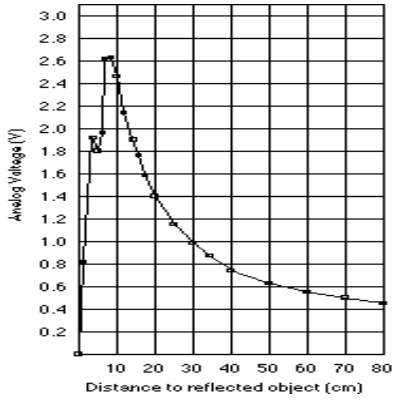


Figure 3-3 Graph of IR Output to Distance

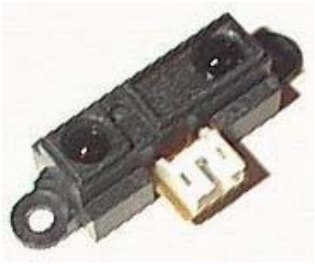


Figure 3-4 Sharp GP2D12

3.1.4 Positional Sensors

It is important that the location of the robot is known at all times. Several different kinds of sensors can be used to tell position. A GPS (Global Position System) sensor could report absolute location, but it is expensive. An inexpensive solution that is common in nearly every mobile robot is a shaft encoder. It usually contains a disk that has alternating white and black segments, and is placed on both drive wheels. An IR detector/emitter pair is placed near the disk, and changes its output when a transition occurs on the disk from a white segment to a black segment. The transition is detected by the processor, which logs the change in distance. The only problem with a shaft encoder sensor is that it reports any movement of the wheels or tracks of a robot. This includes wheels or tracks that are slipping. This can lead to the shaft encoder reporting inaccurate distances. Another inexpensive sensor is an optical sensor. The optical sensor takes pictures of the ground, and reports distance based on the difference between two pictures. The least expensive of this class of sensor is the optical mouse. This sensor is unaffected by wheel slippage, but has the additional problem of a slow update rate. This limits the speed of robot which the mouse reports distance accurately. Another problem is the mouse is adversely affected by certain surface types, in particular a surface that is uniform or reflective.

3.2 The Robots Mechanicals

3.2.1 The Robot Chassis

Building the robot mechanicals from its base components is a difficult job. One of the greatest difficulties is designing the propulsion system. The propulsion system for indoor robots nearly always consists of electric motors. Designing an electric drive system requires knowledge in

choosing a correct motor for the size and weight of the robot. The motor can contain a built in gearbox, but those with a gearbox usually lack power. Therefore, for a mid sized robot, the gearbox has to be built, which would require ability in machining.

Fortunately, there is a much easier way to obtain a ready built chassis. Many mid sized R/C cars contain more then enough power for a mid sized robot. Most of the area in the chassis is available for electronics, after the transmitter board, and plastic cover is removed. The robot chassis used is from a Bedlam R/C toy car from radio shack, which is well suited for use in a robot. The chassis provided a decent sized platform for the robot, that can house all of the electronics. Before the extra weight of the electronics was added, it was a fast toy capable of up to 4 m/s, which means that the motors have sufficient drive power. It is also a skid steer vehicle with tank style track instead of wheels. This has the advantage of being able to turn on its axis, which provides the robot with good mobility. The added traction of the treads, also serves to balance the motors. Even with slightly different amounts of power given to each motor, the robot still goes straight. In addition, with the traction, there is very little slippage and as such using shaft encoders for dead reckoning provides accurate data on the robots position.

3.2.2 Robot construction Techniques

The first task in constructing the robot is to strip the Bedlam of all of its parts except the main drive motors and associated gears. In addition, some of the platform that is not required is sawed away. A frame is then built to house the electronics. The frame mainly consists of scrap PCB (printed circuit board) from a keyboard assembly. Any assemblies that require to be bent are made out of aluminium. All fabricating is limited to three actions: cutting, drilling, and bending. Fortunately, the chassis contains many screw holes of the same size, so the majority of the structure is held together with screws. Where there are not any screw holes, glue (hot melt or epoxy) is used. The only exception is the sonar. The connection between the stepper motor and the sonar transducer uses a lathed component that was created by a machinist

3.3 The Robots Hardware

3.3.1 The Power Supplies.

There are two power supplies in the robot, one for the motors, and the other for the electronics. The reasons for using two power supplies is for noise issues, and for the fact that when turning on the motors, the motors temporarily draw enough current to drop the voltage of the batteries below

the level required for the electronics thus resetting the system. The R/C car is designed to use a 9.6 volt battery for the motors. This is the voltage of battery is used in the robot for the motors. All of the electronics require 5 volts but a second 9.6 volt battery is used. Firstly, the 167 evaluation board requires at least 8 volts for its linear regulator to step down the voltage to 5 volts, which I was reluctant to by-pass to a direct 5 volt supply. Secondly, using only one type of battery saves costs on spare batteries. Because of this, a voltage regulation PCB was made that steps down the voltage from 9.6 to 5 volts. The primary component on the board is the LM2575T**.** This part provides upwards to 80% efficiency compared to about 50% for a linear regulator.

3.3.2 The H-Bridge PCB

For availability reasons, the MC33186 H-Bridge drivers are used to provide the motors with power. These drivers are in the form of a SMT (Surface Mount Technology) package, so a PCB had to be made to house them. The advantage of using this driver is that it has low on resistance of 150 mΩ, and it has over current protection and temperature monitoring. The drivers are each capable of sustaining up to five amps of drive current, which is sufficient for the motors.

3.3.3 Hardware Design issues

In addition to the H-Bridge driver board and the power regulation board, there are several interfacing boards made out of prototype board, and not etched in PCB. It is quicker to make board with only several components on this using prototype board, than making a PCB. These boards serve to interface the 167 evaluation board to the robots electronics. For the most part, these boards consists of power decoupling, and Schmitt triggers for digit input and output, and decoupling for analog input. The fan board, the stepper driver board, and the LCD board also contain drivers to supply current to external devices.

Whenever possible wires are connected through interconnects instead of being hardwired. All of the wires going to the 167 evaluation board are connected through headers. It is possible to remove the interface board and put in a similar one without much effort. Most of the other connection on the robot use IDC style headers. Hardwiring connection would save space, but using headers whenever possible allows for the possibility of removing boards to repair them without cutting wire.

Nearly all of the components were chosen simply because they were available. I had the advantage of being at a company, Vansco Electronics, with a rather large inventory of electronic

components. The 167 processor was chosen because the evaluation board, along with its compiler are present at Vansco. Through Infineon's sale connection at Vansco, I was able to obtain my own 167 evaluation board that Infineon generously donated. The 128X64 LCD was chosen because it is used on a Vansco product. The H-Bridge drivers were ordered in as samples and remained unused, thus they were available for my use. The only major components that were purchased from outside Vansco were the IR Rangefinders, the optical mouse and the sonar assembly. Fortunately, even though most of the components were chosen because they were available, they proved to be good choices. The one component that was not a good choice was the wireless module, which provides low bandwidth. It is several years old, which on wireless terms are several generations, but it was the only choice possible.

3.3.4 Circuitry Protection

The only circuitry protection explicitly provided for the robot is reverse polarity protection. A NMOS transistor is placed in the batteries ground return path, which prevents current flowing on reverse polarity. There is no transient protection, because the only big inductive loads are the motors, which are directly connected to the battery through the H-Bridge chip. There are no fuses, because in all cases there are "fusible" components that prevent short circuits. The H-Bridge drivers have short circuit protection internally. The regulator on the 5 volt supplies are current limited to around one amp. The 167 evaluation board regulator stops passing current at a certain point. Even so this still leaves the possibility of a short circuit occuring on a connection. To prevent this, each board, even the ones built on prototype board have board coating to prevent contact with foreign objects and corrosion. All wire connection either contain heat shrink, or are taped, or glued that short circuits should not develop over time.

3.4 Hardware Interfacing

In order to take full advantage of the hardware on the robot, software has to be written to allow the hardware to be used to its full potential. Every hardware element must be supported by low level software routines, which form the basis of an operating system. As such, the following section describes how the hardware is interfaced to the robot. A description is also provided of how the Robot API, first described in chapter 2, is implemented over the operating system.

3.4.1 Operating System Architecture

The operating system is based on a multi-priority model. There are two priority levels. Raw sensor input is at the high priority level, while the sensor processing and AI is at the low priority

level. The high priority level is accomplished by routing all of the sensor input to an external interrupt or by placing sensor polling on an interrupt level scheduler. The processing provided by the interrupt level usually consists of saving only the raw data into a queue. Once the raw data is in a queue, it is processed at the lower priority level. The one notable exception is the wireless packeting. This is processed at the interrupt level to speed up the wireless transmissions. This architecture allows the most current sensor data to always be available for processing, even if processing intensive algorithms are being used by the AI. In addition, a system monitor is placed on the high priority level. This ensures that the robot is operating correctly.

3.4.2 Core Operating System Services

3.4.2.1 *The Kernel Scheduler*

Hardware resources used:

- capcom Timer T0
- capcom Register CC1
- capcom compare interrupt

The kernel scheduler is an interrupt driven scheduler. The interrupt occurs every millisecond, at which time each scheduling block is checked to see if any processes should run. None of the processes that are run by the scheduler are processor intensive, otherwise these processes would starve the low priority level of computer cycles. The scheduler also keeps track of the amount of time that has passed since the robot was turned on, in the form of a software tick counter.

For diagnostic purposes, the kernel scheduler contains a function called “blinky” which is automatically added to the scheduler on the initialization. This function toggles one of the evaluation boards LEDs (light emitting diode) on and off twice a second. If the scheduler stops working, the light stops blinking. This is the surest way to tell if the system has ceased working.

3.4.2.2 *A/D Scheduler*

Hardware resources used

- The A/D module
- A/D conversion complete interrupt
- Any pin on port 5, for a total of 16 pins could be used for A/D

This scheduler is used to schedule A/D (analog to digital) conversion. The purpose of the A/D scheduler is to put various devices could be put on the A/D port so their conversions can occur at different rates. It is advantageous to schedule A/D at different rates because the IR rangefinders

require a frequent scheduling rate, and the temperature chip can be scheduled at a slower rate. To start the conversion process the A/D scheduler is initially run by the kernel scheduler. Once a conversion is complete, an interrupt occurs, which not only saves the current output of the A/D, but also continues to run the A/D scheduler.

3.4.2.3 *Wireless Serial Communications*

Hardware Resources

- The Serial Port Module on 167
- RS232 Level IC (on Evaluation Board)
- RFI Wireless serial board
- Receive byte interrupt
- Transmitted byte interrupt
- P3.10 Transmit
- P3.11 Receive

To communicate with the PC the robot employs a wireless serial module. The module requires a packeting protocol for both the PC and embedded side. The wireless module suffers from the problem that it can only send or receive data at one time. If a host sends out a request to transmit and the wireless module is unable to do so, it does not respond to the request. Therefore, the host has to timeout the request and start over, thus costing bandwidth. Because of this, a handshaking protocol is used. The monitor program on the PC is considered the master and initiates a transmission. After the transmission has been successfully received by the robot, it is designated as the robot's time to transmit a message. If the robot does not have data to send, it sends out a blank message. After it has received a message back from the robot, the monitor then could send again, and so on. If for some reason the transmissions has a timeout, the monitor on the PC reinitiates the transmission. In practice, this handshaking scheme allows for the greatest amount of data to be sent over.

The wireless module is limited to sending 32 data bytes in each packet. For efficiency reasons despite the fact that most messages are well below 32 bytes, it is advantageous to be able to send over data at the full 32 bytes at a time. It is also necessary to send out messages larger then 32 bytes. To do this a second protocol is used which uses a separator characters to allow for multiple messages per packet, and for messages larger then 32 bytes. Each message contains both a start character and an end character. When data is received from a packet, this data is placed into queue. The queue is monitored to see if it contains any messages, which are extracted from the queue and then parsed.

When sending messages, message data is placed in a transmit queue, instead of being directly sent to the wireless module. When it is time to transmit the data, the data in the queue is formed into a packet and sent. The queue allows multiple processes to use serial communications without interfering with each other, or having to wait to be able to transmit a message.

To ensure proper operating of the serial module, the system monitor constantly monitors the queues (both transmitting and receiving) checking to see if they get too full. If they do, the queue is then flushed, and the PC monitor program is informed so the amount of data transmitted could then be reduced.

API Functions

- void SendNode(unsigned char Node)

Although there are various status messages, and command messages that go between the robot and the PC monitor, only one is classified as an API message. The SendNode function reliably sends information about a node to the AI monitor. To do this, a function is placed in the kernel scheduler that checks to see if the monitor has acknowledged the reception of a message. If it has not, the node is resent.

3.4.2.4 *The System Monitor*

The system monitor acts as a software watchdog timer to make sure all of the software routines are working correctly. It monitors flags from different modules. If an error flag appears from a module, the monitor takes appropriate action. This is especially important for the serial module when the queues have reached a dangerous size and needs to be flushed to enable quick communication. The system monitor also sends out a status message so the PC monitor program is informed of any problems in any of the modules.

3.4.2.5 *The LCD Routine*

Hardware Resources

- Address, data and control Bus
- Address bus glue logic, and Backlight switch board
- A1, A17, CLK, WR/WHL, AD0-7, P8.1 for the backlight

Note that the entire LCD implementation both hardware and software is courtesy of Vansco Electronics. This is deemed appropriate because the LCD is not core to the robot project, but rather a very useful debugging tool.

3.4.2.6 *Piezo Buzzer*

Hardware Resources

- Interface board (Fan/Piezo)
- P8.3 (active low)

The function call for the piezo buzzer was created to allow processes to dictate multiple beeps and beep duration without directly controlling the piezo output. To do this, the piezo buzzer has a function that is run in the kernel scheduler, which toggles the buzzer on and off.

API Fuctions

- void Beep(short numofbeeps, short duration)

3.4.3 Motor Drivers

3.4.3.1 *The H-Bridge Control*

Hardware Resources

- PWM modules 1 to 4
- H-Bridge PCB consisting of 2 MC33186 H-Bridge driver
- P7.0 – P7.3, P8.0, 8.2

The software simply sets the duty cycle of the PWM (pulse width modulation) channel. There is a reason why four PWM channels are used to drive 2 motors. A DC motor has four modes: forward, reverse, freewheel, and brake. In order to use all four modes, either four PWM channels or two PWM channels with four other pins and some glue logic have to be used. Because there is no other use of the PWM channels all four PWM channels are used.

Each H-Bridge chip has a output status flag. When for whatever reason the chip cannot function (overheated) the flag goes on, which is passed on to the PC monitor via the system message. Due to the large heat sink on the drivers, this has never happened.

API Functions

- void MVGoFoward(char Speed)
- void MVGoFowardWA(char speed, char angle)
- void MVGoFowardT(char Speed, short Ticks)
- void MVAllStop()
- void MVTurn(short Ticks)
- BOOL MVIsBlocking()
- void MVReset()

The API implementation is rather straightforward. When turning on carpet the robot requires around 85% of the PWM duty cycle to start moving. Because of this, the MVTurn command immediately places 85% duty cycle on the PWM channels. When going forward, far less power is required and it is advantageous to provide some acceleration, because turning on the motors with full power could allow the tracks to slip. To provide acceleration, a function is placed in the kernel scheduler. This gradually increments the PWM duty cycle. When using a blocking function, a timeout value is created, which times out the blocking call after a certain amount of time. If this happens, a flag is toggled in the system message, which is sent to the PC monitor.

3.4.3.2 *The H-Bridge Temperature Regulation*

Hardware Resources

- A/D module, (scheduled through the A/D scheduler)
- A/D interface board
- Fan Interface board (Fan/Peizo)
- P5.8 (temperature chip) P2.8 fan turn on (active low)

The temperature chip is placed in the A/D scheduler at a slow rate. Its latest output is recorded, and converted to Celsius. The H-Bridge temperature monitor is placed on the kernel scheduler. If the temperature exceeds a threshold of 40^0C Celsius, the fan is turned on. If the temperature exceeds 100^0C the motors are forced off. After the temperature regulation with the addition of the heat sink was implemented, it was realized that the heat sink used is large compared to the H-Bridge drivers, which results in low temperature even when the motors are stressed.

3.4.4 Sensors

3.4.4.1 *Bump Sensors*

Hardware Resources

- Bump switches (8)
- Bump switch interface board
- P7.4-P7.7 and P8.0-P8.3

The bump sensors are arranged so there are two sensors per side of the robot for a total of eight sensors. All of the bump switches are buffered through Schmitt triggers on the interface board. The switches should be polled periodically by the AI.

API Function

- char BMPGetByte()

3.4.4.2 *The Sonar*

Hardware Resources

- T1,
- CC4 (stepper delay) interrupt
- CC12 (sonar capture) interrupt
- CC13 (sonar timeout) interrupt
- Polaroid 6500 ranging board
- Polaroid instrument grade transducer
- Stepper driving board
- Connector converter board
- General Sonar interfacing board
- P2.4-7, P2.12-14

This is the most complicated device to interface because it consists of a Polaroid 6500 ranging kit, with the transducer placed on a stepper motor. Because delays are necessary when driving the stepper motor, hardware timing is needed to interrupt the processor and inform it to move along the next step on the motor. Hardware timing is also needed to measure the time it takes for a return sonar pulse.

API Functions

- BOOL SNRIsBlocking()
- void SNRReset()
- void SNRSetFowardScan()
- BOOL SNRGetFowardDistance(short *Distance)
- void SNRSet3PointScan()
- void SNRGet3PointDistance(short *distance[])

The sonar has two modes, fixed scan forward, and scan 3 points. The fixed scan mode keeps the sonar at a fixed angle and scans at a maximum rate of 10 times a second, to save power. The 3-point scan scans three angles –90, 0, 90 with a scan arc of +/- 20 degrees at the stepper motors minimum 3.6 degrees per step. When the SNRGet3PointDistance function is called, the information of the content of the path (hallway, wall or unknown), is calculated by looking at the output from each sweep angle. The actual implementation is done by experimentation, and is actually simpler then the simulator implementation.

3.4.4.3 *IR Range Finders*

Hardware Resources

- A/D module, (scheduled through the A/D scheduler)
- Analog Interface board
- Sharp GP2D12 (5)
- Pins 5.0-5.4

These sensors are used to measure distance. There is a problem with accuracy due to the noise. Unfortunately, the de-coupling is done on the A/D board, which does not quite have the same ground as the 167 evaluation board, which receives the signals. This means that the output from the A/D board seems noisy to the micro-controller. However, the noisy A/D board power and ground are also sent to the 167 evaluation board to be used as a A/D reference voltage. This reduces the effect of the noise but does not eliminate it.

API Functions

unsigned char IRGetFrontD()
unsigned char IRGetRFrontD()
unsigned char IRGetLFrontD()
unsigned char IRGetLeftD()
unsigned char IRGetRightD()

Even though the sensors refresh at around 35 ms, to average out the noise, the IR A/D conversions are scheduled at a much quicker rate, that are then averaged. After averaging, these values are converted to distance with linear interpolation and a look up table that is unique for each sensor. The latest values of distance are in an array, which are accessible at any time with the above function calls.

3.4.4.4 *PS/2 Optical Mouse*

Hardware Resources

- The High speed synchronies serial interface and interrupt
- capcom for initial timing issues.
- PS/2 interface board
- P8.4-7 P3.9, P3.13

The interfacing of the PS/2 mouse to the 167 processor is tricky. This is because the PS/2 is a half-duplex synchronous protocol, using one clock line and only one data line. To be able to use only the synchronous serial module to interface the mouse, both the transmit and receive would have to wired to the data line. As well, the protocol requires both lines to be toggled according to the protocol for the slave (the 167) to be able to write a byte into the master (the mouse). While it might be possible to configure the hardware to work as follows. Four digital I/O lines are used initially and hardware timing to emulate a PS/2 adapter thus writing a byte into the mouse to put it into stream mode. After the mouse is in stream mode, the synchronous serial module is turned on in receive mode only, to receive the data stream from the mouse.

Once in stream mode, on the receive interrupt, the received byte is place in the PS/2 queue. A polling function is called from inside of the main loop to look at the queue. To process the PS/2 mouse is difficult because the mouse moves with the robot body. As mentioned before in the simulator difficulties section, to translate X and Y displacement into robot displacement requires complex math. Therefore the mouse can only be used as an anti-slip sensor, to complement the shaft encoders. When the robot is rotating on its axis, the mouse only reports back an Y displacement, which could be used to figure out when the robot has completed a 90 degree turn. This compensates for the use of the problematic shaft encoders when the treads are slipping during a turn. Unfortunately, even with using the newest optical mouse capable of 6000 updates per second, the mouse is not capable of keeping up with the minimum turning speed on carpet. However, if the robot were ever used on a surface that has less traction, therefore more slippage, and a slower minimum turning speed than carpet, it would be used to complement the shaft encoders.

3.4.4.5 *Shaft Encoders*

Hardware Resources

- General Purpose Timer 5 and 6, triggered to count on both edges
- Two shaft encoders made out of a IR emitter/detector pair
- Interface boards (same one used for the bump sensors)
- P5.12 (Timer 6) P5.13 (Timer 5)

Due to the fact that the output of the shaft encoder detector does not have an ideal 0 to 5 volt swing, the output is buffered through Schmitt inverters.

API Functions

- short SEGetTick1()
- void SEResetTick1()
- short SEGetTick2()
- void SEResetTick2()

Each of the two tick counters are independent software counters. They are updated during frequency polling when either of the two timers update.

3.5 Verification

The robot was constructed using the incremental approach. Each hardware component was added individually. After the hardwares associated software was programmed, the component was thoroughly tested to ensure it worked. In addition, since all of the hardware has to work together,

at each stage of adding new components all of the hardware was tested. The latest version of the operating system that has been existence for the latter part of the project has never failed.

Chapter 4
The AI

4.1 Background

4.1.1 Subsumption Based AI

In the past (15 years ago), the hardware of the robot has been far more limited then it is today. At that time Brooks [Broo85], introduced a new concept called the subsumption architecture. This architecture is based upon increasingly sophisticated levels of behaviours. Each increasing level of behaviour is of lower priority than the previous one. The lowest level of behaviour receives the highest priority, and is usually responsible for collision detection. A second level of behaviour may be a wall following module. A high level behaviour can be responsible for activities such as map making, and its inputs consists of the outputs of the lower levels. In this architecture, there is no central control system. An advantage of using this methodology is that only the first few basic levels of behaviour need to be implemented for the robot to have some limited capability, which could match to the limited hardware of a robot. Some ideas from the architecture are still popular today such as the idea of decentralizing control to various modules. However, presently with today's hardware, this architecture is not used often for research purposes but it still is popular with hobbyist.

4.1.2 Metric Based Map Navigation

A metric approach describes how an area is occupied with objects. A popular form of a metric representation is an occupancy map in which an area is represented by a grid, in which each cell can be occupied, empty, or probabilistically occupied. This approach makes it simple to generate a map by simply placing sensor output onto the grid. Some researchers have used sonar sensor output that consists of a ring of 24 sonars, spaced 15 degrees apart from a robot similar to the iRobot B21 [TBBF98]. The output is fused together with a neural net to form the grid. This approach proved very successful in not only generating a map, but also in localizing the current position of the robot based on the previous sensor output. There are other forms of metric representation in which 3D polygon are used to represent an area. The 3D polygons are generated with 2 laser range finders on a Pioneer-DXe [ThBF00],

The main disadvantage of using a metric map is that the map requires a large amount of storage. Making a metric map also requires full sensor coverage. Computational power is also required

when using a probabilistic model that localizes the robot while mapping. Another disadvantage is that to generate a path on a metric based map requires that every element in the map be examined. Thus navigating a metric map also requires a large amount of computation power.

4.1.3 Topological Based Map Navigation

The topological representation differs from the metric representation in that objects themselves are not explicitly stored, rather pathways that are generated based on objects in the area are stored. This means using a topological representation forces the robot to navigate in a predetermined path. There are perhaps many ways to represent a topological map but only two ways are considered for this project. A Voronoi graph [NaCh99] [Chos01] is a collection of points equidistance to two objects. An example would be a hallway that is represented by a line that runs through the centre. The disadvantages of the Voronoi graph is that the fact that the robot could only be used in areas that at all times have equidistance objects. This graph would not work in an area with large opened spaces. Another way to store graphs is by using an ad-hoc technique that stores an area as a collection of nodes, and uses links to connect the nodes together. Only the nodes contain information about the environment, and the purpose of the links is solely to connect the nodes.

The main advantage of a topological map is that once generated it is very easy for a robot to navigate using the map. A computer can calculate a path far quicker using a graph representation then a grid representation, because far less information needs to be considered. One disadvantage is that the robot is forced to follow a predetermined path, which means that the path may not be the shortest distance. In addition, this representation is more suited to an indoor office style environment that is generally uniform. A large room with many small objects is not suited to a topological graph.

4.1.4 Hybrid approach

It is possible to combine both the topological and metric map principles to gain the advantages of both. One of the ways is to use topological maps to link uniform areas such as hallways, and use metric maps in areas such as rooms. Another way is to first generate the map using metric techniques, and then calculate a topological map out of the metric map, that then is used for navigation. This approach is used by Thrun et al that also provides one of the best examples of a

functional autonomous navigation mobile robot, with the robot being able to give tours to people of indoor areas such as museums.

4.2 The Robots AI

4.2.1 Introduction

Originally, because of time requirements, a subsumption based AI was to provide the robot with its ability to navigate by blindly following walls. Subsumption with all of its disadvantages is still the easiest to implement, and not as affected as other methods by the lack of sensor coverage. Using a metric based maps is out of the question, because they require full sensor coverage, usually by a sonar array. The robot contains a rotating sonar, but it takes several seconds for it to scan an area, which is too much time. In addition, the robot lacks the memory for a grid based map with a reasonable amount of resolution. Other metric based schemes such as using lines require less memory but are too computationally expensive to generate. The most used topological approach, the Voronoi graph, requires that the robot be equidistant to two objects. This is not possible due to the limited range of the IR rangefinders. Eventually a topological approach that uses an ad-hoc technique was developed, which would provide better navigational ability then a subsumption AI, while being feasible to implement with the current sensor arrangement.

4.2.2 The AI Design

4.2.2.1 *Introduction*

The topological approach that is implemented is designed to mitigate the problem of the lack of sensor coverage of the robot. The mobile robots principal sensor, the IR rangefinder, has an effective range of only about 80 cm. To be able to localize the robot in an area, the robot is forced to follow predefined paths that are parallel to walls. This constraint may seem limiting, but it is sufficient to navigate in the target area of an office space, in which narrow hallways and offices/cubicles are prevalent.

The topological approach consists of a graph. This graph contains nodes, and links between the nodes. The information in the graph is based on storing relative distances. The robot has a target range of distance to be parallel to the wall to stay within the range of the IR sensors, and not to be to close to collide against the wall. It is ideal not to have a fixed distance to the wall because constant discontinuities would cause the robot to be constantly having to move closer or further

away from the wall. This causes the undesirable effect that the robot is no longer travelling in a straight line.

One of the drawbacks of this topological approach occurs once a robot enters a room. The topological maps provide adequate information to travel from room to room, or cubicle to cubicle. It does not provide a sufficient medium to travel inside a cluttered room. Because of this, a small-enclosed area is defined as an unknown node. This node acts as a path terminator. The robot may travel to an unknown node, but must do so by only traveling at a predefined distance from the nearest decision node. This is to enable the robot to turn around and be able to leave the room. A module can be added to the AI to enable the robot to travel inside a room and be able to navigate its way to the exit, if necessary. However, the principle idea of the project is to travel from room to room, so spending time working on an unknown area module this is not deemed important for this project.

4.2.2.2 *Navigation of the graph*

Once a graph is created of an area, navigation is easy. A shortest path algorithm is used to generate a path from the origin node, to the desired one, and the robot follows the path. The approach is robust to dead reckoning errors because it relies on a constant localization system, in which any landmarks are compared to ones in the direct path in the graph. The robot is then positioned at the location of the landmark that is stored in the map.

4.2.2.3 *Issues of Making The Map*

Mapping an area requires several problems to be solved. The first of these is for the robot to be able to identify hallways from rooms. To identify rooms it would be ideal to have the robot explore every area including rooms, to determine their identity. This would slow down the mapping process and add to the dead reckoning errors. While map making, the robot is not to travel into any rooms. It is essential that the robots sensors, in particular the sonar be used to identify rooms from hallways.

Another problem is that the hallways may be too wide for both sides of the IR sensor to pick up the adjacent walls. Because of this, both sides of the hallways have to be mapped separately. It may seem that this method may lead to two disjoined pathways, with no connectivity. This is true for wide hallways spanning several metres across. However, for hallways of only a few

metres of width, it is possible to make use of the fact that the sonar could identify common decision point across a hallway, then link these two point together.

The navigation relies on the fact that the robot is always localizing against the graph. Without a map and a device which outputs exact position, the only way to localize is to use previous reading of the sensor, to correlate current position based on past position. However, taking full sensor sweeps are slow and correlating requires processor power. This poses some difficulty because only the dead reckoning sensors are used to tell position, and they can produce large errors. When the robot travels in a loop and returns to its starting position, it is possible to "close the loop". Loop closing occurs when the robot has returned to its starting position, and knows the position error incurred during mapping. This error could be propagated backward on the map, to reduce the error, and cause the first node to align, with the last node.

When a robot has completed mapping a loop of an area and returned to its starting position, there may be a position error, due to the dead reckoning system not providing accurate position information. When a loop is completed it is then closed, and the positional error is propagates backwards on the map to reduce positioning errors. Loop closing is a contentious issue. Some robots close loops by having an operator inform the robot when a loop is closed. Some robots have the loops closed during post-processing. A robot that can be called autonomous should be able to close the loops by itself. The difficult part with closing the loop is that if there is a large dead reckoning error, it can be difficult to determine if the robot has returned to its starting position, and whether the loop is to be closed. The problem is solved on assumption. The method used with the robot is that the loop is closed when a similar decision node with a similar sonar sensor output is found within a reasonable distance of the starting node of the loop. This can fail if the dead reckoning error is exceeding large, or the area contains many decision nodes that look the same, in practice however, the method works.

One further concern of map making is how to explore an area to map it. To minimize dead reckoning errors, the smallest possible loop should be traversed and then closed. This is done by using the convention, of always turning right if possible into a hallway to form a loop. After the first loop is closed, a decision node that has a link to a yet to be explorer hallway can be chosen randomly, and explored. This method can be used until there are no more decision nodes to explore, and then the map making is done. In practice, this is rarely done for several reasons. The robot does not know the entire layout of the area, so it can be very inefficient to choose an

area to explore at random. In addition, the area can contain stairs or other hazards that can jeopardize the safety of the robot. Therefore, after closing the first loop, the robot waits for a command that instructs which area to explorer next, and generate its map.

4.3 AI Implementation

4.3.1 Introduction

Performing map making and navigation are similar. When map making, the sensor data is used to generate a map, and when navigating, the sensor data is used to localize against a map. Because of this, both the map making and navigation are contained in one program. The program contains three levels, with only the top level differing for the two tasks. The first level consists of the raw sensor data gathering level. The only function of this level is to gather sensor information for the higher levels. The mid level performs the basic movement activities of the robot such as follow a wall and turn. This level defers all decisions to the top level. The top level is the decision level. This level looks at the exit conditions of the mid-level, and makes decisions. This system may look a lot like subsumption because the lower level modules take control. However this is not the case because all decisions are centralized at the decision making level.

The following sections present all the modules which make up both the map making and navigation programs. The low level module consists of the IR processing, and the sonar processing which is implemented in the API. The mid level modules are the follow wall module and the turn module. The top level modules, are the map making decision module, and the navigation decision module.

4.3.2 IR logging Module

This module is responsible for producing the IR distance values for the rest of the AI, and for logging any changes in distance. The first task of this module is to obtain the most recent IR distance values by calling on the IR API functions. These values are stored with past IR distance readings. The value of the distance that the rest of the AI uses is then produced. The distance could just be the raw readings, but if noise is an issue, filtering can occur based on the past IR readings.

After the present value of the distance is produced, this value is checked against the previous value to see if it is different. If it is different, its value and the timestamp are logged by using the KRNGetTicksL API Function. A flag is toggled which states a new value is available. The purpose of the flag is to inform the follow wall module that the robot is drifting away or towards the wall.

4.3.3 Sonar Processing Module

Although the sonar processing is done in the API level, it is considered a member of the AI sensor gathering level. It is necessary to implement this module in the API because of the profound differences between the simulators sonar model, and the real sonar on the robot. The simulator is not programmed to account for the sonar reflection model, and as such does not provide an accurate representation of the actual sonar output. Instead of attempting to improve the simulator's sonar model to work more like the real sonar, it was decided to work around the problem. Instead of having the sonar output distance, it processes to distinguish an this as to one of three types. An area is either a hallway, a wall, or an unknown area. This serves to classify rooms and short hallways. This information is then used by the decision module to decide which direction to explorer next when map making.

4.3.4 Follow Wall Module

The navigation scheme is based on following walls, and a module exists to accomplish this. The follow wall module, must keep the robot parallel to the wall at all times. This goal is complicated by the fact that the wall although the wall is restricted to 90 degrees changes in direction, it may contain many discontinuities, such a file cabinet placed near the wall. This module has the ability to compensate for this. In addition, this module must accept starting conditions at which the robot does not start parallel to the wall, and needs to be immediately adjusted.

In terms of sensors, the module uses either the left or the right IR to track distance to the wall. Changes in sensor readings are already monitored by the low level IR module, which informs the module. The time stamp of the previous reading is compared to the new one, and the time it takes for the distance to change is calculated, and then processed. If the rate of change is slow it is indicative of the robot moving further or closer to a wall. If the rate of change is fast it indicates that the IRs have detected a discontinuity in the wall.

If the rate of change is slow, a correctional movement is generated. The rate of change is scaled with a constant, which dictates how much of a correction is made to the robots movement. One problem is that the robot can oscillate about the wall instead of travelling straight. Note that after the first oscillation is logged, the robot at some point must be parallel to the wall, and any large initial error in not being parallel is reduced. Because of this, the remaining error is somewhat small, so dampening parameters can be introduced to reduce further oscillation. In the simulator at least, after a few oscillations, the robot travels approximately parallel to the wall. Furthermore, the absolute distance to the wall is monitored because it is possible for the robot to gradually oscillate itself towards or away from the wall. A minimum and maximum distance is set, to force the robot in a certain range.

When a rapid rate of change is detected, the follow wall module enters a transitory state. In this state, the robot is forced to go straight without any adjustment. When the rate of change finally settles, it is likely that a discontinuity in the wall is found. However, if the starting and ending distances of the transitory state are close to the same value, it is judged that the noise caused the module to enter the transitory state, and the module returns to its normal state. If the distances are sufficiently different, the follow wall module logs the discontinuity information, and exits, so the decision level, could make a decision.

The module is also responsible for monitoring the possibility of a forward collision. If a forward object is detected the module informs the decision level about the possible collision and exits.

4.3.5 Turning Module

The decision level could order the robot to make one of two turn types. One is a blind turn in which the robot turns 90 or 180 degrees oblivious to its surrounding environment. This turn is accomplished by calling the API function MVTurn, which is a blocking call and does not allow processing to continue until the turn is complete. The other type of turn that the robot can make is a programmed turn. In this case when called, the robot not only turns but also moves until a wall contact is found. The decision level defers control to the turn module to accomplish this turn.

4.3.6 Unknown Area Modules

When travelling either to or from an unknown area, this module is used for navigation. The problem with an unknown area is that the area is likely to be a room, which consist of objects

spaced at irregular intervals that are not easily represented by the topological graph. Because of this, it would be ideal for this module to use a metric form of navigation to traverse a room. A properly implemented module would enable the robot to exit a room no matter what its initial placement. The robot would also be able to travel inside a room, to a certain location in it. To enable this action without the use of a map, the robot could take full coverage sonar sweeps, to build an idea of what the room looks like. It would store the data temporarily to finds its way out or to a certain point in a room. Unfortunately doing this would haved require extra development time, which was not available. The current unknown modules (both entering and exiting a room) are simple. When entering a room the robot simply turns into the direction of the room, and goes forward up to a metre while trying to avoid objects. The exit room routine does the opposite motion. This routine is problematic because, if the robot is knocked off course, it may not be able to find the exit.

4.3.7 Decision Level

The decision level is the highest level of the AI. It makes all decisions. The decision level is called upon to make decisions when the lower level modules have reached an exit condition. It also either generates a map or follows the map for navigation purposes. There are two decision modules, the navigation module, and the mapping module. Each receives the exactly the same inputs from the lower level modules The navigation module is responsible for moving across nodes in the graph. The mapping module is responsible for generating the maps.

4.3.7.1 *The Mapping Module*

The mapping module works on the principles of creating maps described in the AI Design section. The module lets the follow wall module operate the robot until that module produces an exit condition. If the exit condition is a detection of a discontinuity where the wall is still detected, albeit at a different distance, a landmark node is added to the graph representing the size of the discontinuity, and the direction of it. In this case the follow wall module is re-enabled. If the follow wall module exits with the condition of ceasing to detect a wall the decision module must make a decision. First the discontinuity is logged, as a landmark node that contains the information of losing wall contact, with the direction of the lost contact. Then the decision module moves forward a little to allow the sonar to take a three point scan of the area. The sonar scan tells the module, which ways are possible to travel. By convention if a hallway is detected on the wall following side of the robot, the decision module will turn into the hallway. If not, the robot proceeds straight. If the robot is unable to go straight or turn into its wall following side, it

turns to the opposite wall following side. If a turn is called upon, the turn module is told which direction to execute the turn, and then it is enabled. As soon as the turn module exits, a landmark node is recorded the gain of the wall detection, and then the follow wall module is enabled again.

It is correct to say that the above system is not able to take care of all situations. The above system is based on always trying to turn towards the wall following side. This could be called mapping the inside loop. Although an office like area is primarily composed of inside loops, there is always at least one outside loop, where the robot is required to try to turn away from its wall following side. The problem with implementing this is that the robot would have to be able to classify an opening such as a hallway away from its primary wall following side. In narrow hallways, it may not always be possible for the outside IR to have the range to pick up a discontinuity on the other side of the hallway. Because of this, the robot needs to constantly use its sonar to scan in at least a 90 degree arc away from its wall following side. The problem with doing so is that a 90 degree scan is not particularly fast, and the entire mapping scheme needs to be slowed down. While the navigation module would not have a problem with the outside turn concept, additional work would have to be done to be able to use this idea from mapping. Unfortunately, the time it would take to implement this had to be taken into consideration, and as such, outside loop mapping is not supported.

Another consideration is the fact that each decision node could only have four links, one in each direction to other nodes. Because of this it is not possible to have only one decision node at a junction. When making a map of a hallway both sides of the hallway are explored and mapped separately. The junction cannot link to both of them, because they are in the same direction. To solve this problem, the decision node junction is cloned so there are two nodes, which are then able to link to both sides of the hallway. The two nodes are then connected together.

As mentioned in the AI design section, once the first loop is completed, the map module waits for user input before mapping the next loop. The robot operator specifies a decision node to start the next mapping loop, the direction in which to travel, and the side of the robot to use for wall following. The robot is ambidextrous in terms of being able to use its right or left side to follow a wall.

4.3.7.2 *The Navigation Module*

The navigation module's task is to have the robot move from point A to point B using the map. The module is called either from the mapping module to start making a map, or from a user, to move to a different point. This module, receives the same input as the mapping module, however instead of creating new nodes on the graph, it uses the information to localize the robot on the already created map. The route to take to move from point A to point B is generated by a helper function, which implements a shortest path algorithm.

The navigator is rather simple. It stores several pieces of information such as the current node, the immediate destination node, and the goal node. It also keeps track of direction, and distance travelled since the last node. Information about the route that is generated by a shortest path algorithm is stored in a stack. Once the follow wall module identifies a landmark, this landmark is correlated against the current destination node. If it is the same, the destination node becomes the current node, and the next destination node is popped from the stack. If the landmark is different, the stack is searched, to see if a similar landmark exists between the top of the stack, and the next decision node stored on a stack. If there is, it is assumed that the robot is at that point, and the current position is updated accordingly. This situation can occur because it is possible a landmark's size can be close to the threshold for recording landmarks, and yet not be recognized every single time. As soon as the stack is empty, the current node is the goal node and the navigation module stops the robot, pending further instructions.

Several ambiguities occur which are treated as special cases. When starting out, before turning on the follow wall module the robot must be in the correct direction. This is accomplished by performing a blink turn if necessary. When travelling from decision node to decision node, the navigation is blind, because there are no landmarks to localize the robot. In this case, the robot uses the distance between the nodes as a basis of how far the robot has to travel. In the case of travelling to an unknown node, perhaps to enter a room, the unknown travelling routine is turned on. When travelling from an unknown node, back to its parent decision node, a similar routine is used. Both of these routines are overly simplistic and provide only bare functionality of travelling to and from a room. As such, in practice, even in the simulator, errors occur when using these unknown node routines.

4.3.8 Topological Graph Implementation

If it had been designed for a PC, the topological graph would have been implemented using dynamic memory allocation. However, because the AI is intended for an embedded system, is was decided to pre-allocate the memory required in the form of an array, and then create a custom memory allocation system. The system is rather simplistic with each node in the array being initially linked together to form a chain of free nodes that can be allocated. When a node is allocated, the free node pointer moves to the next free node in the array. When nodes are deleted, they are simply added back to the free node link. Using a custom memory allocation scheme allows for perfect segmentation of the available memory. Efficient memory allocation is important because the embedded system memory model contains only enough memory for several hundred nodes.

Each node contains the same structure despite its type, so some of the information stored in a node is interpreted differently for each type of node. Three types of nodes, (landmark, decision, unknown) contain common information such as either specifying their node type, position, index number, and for links which either connect to another node or are specified as null. The unknown node, by convention is unexplored, so only one of its links, links back to its parent decision node. The unknown node contains no more additional information. The landmark node must contain one and only one piece of landmark information. In the same memory area, the decision node stores information on paths not traversed yet as if they are hallways, unknown areas, or walls. During a mapping loop, all nodes on the loop contains the ID of the loop to distinguish it from nodes not on the loop. This is used for processing when a loop is closed.

4.3.9 Closing Loop algorithm

The close loop algorithm is used when the map-making decision level has indicated that the robot is in the same position as the one it started mapping the loop. The reason this is necessary is that the position of the nodes is determined by the dead reckoning sensors, which are not entirely accurate. When the robot has reached the start of the loop, there is probably some error in the difference of its reported position and the position of the starting node. Because of this error, a close loop algorithm is developed to propagate error backwards in the nodes in the loop to eliminate the error. Closing the loop also can occur as soon the robot reaches a similar decision node that is in the same position of a previous mapped loop. Closing the loop becomes even more important in this case to keep the positions of the second and later loops consistent with the positions of previous loops.

It should be noted that the simple act of propagating the error backwards on the nodes in the loop does not eliminate all of the error. Actually, it reduces the effect of random errors, but the map-making scheme contains a persistent error, that has not been eliminated. When following a wall the robot is not parallel to the wall at all times especially when it had just gained contact and is oscillating when attempting to align itself parallel to the wall. The distance traveled due to the oscillations is counted as part of the total distance. In theory, this oscillatory error is consistent, and one property of a loop is that the robot travels in one direction the same distance as it does in opposite directions. This means that the oscillatory errors in both directions should be same and should not create any recognizable errors. To reduce the error due to this effect, it is possible to rescale the entire graph. It is not, however, necessary to account for this error because when navigating, a goal node is specified by node not position.

4.3.10 Shortest Path Algorithm

Once the navigator is called upon to navigate, the first thing the module does is call the shortest route function to generate the best route to navigate to the goal node. The algorithm contains two parts. The first part is to find out the distance to all other nodes in the graph, based on the position on the starting node. The first part of the algorithm consists of setting the weight of the starting node to 0, and the weight of all other nodes to a large number. Then add all paths out of the starting node to a queue. An item is then extracted from the queue, and the algorithm continues until there are no more entries in the queue. An item is removed from the queue and its path is traversed while adding the new weights to the path. If in the pathway the distance of the next node is less than the current distance, this pathway is complete thus extract the next item of the queue. If a decision node is reached, place all of the potential paths into the queue, and then extract the next item from the queue. The second stage requires a simple traversal where the algorithm starts at the goal node, and works its way back to the starting node, by constantly picking the path with the lower distance. Every time a node is traversed, it is added to the stack, which is later retrieved while navigating the path.

Chapter 5
Results and Discussion

5.1 Descriptions of the Testing Area

The environment in which the AI was tested was Vansco Electronics engineering floor, as shown in figure 5-1. Figure 5-1 is generated by the map editor and is an accurate representation of the engineering floor. However, the map contains only the inside area of the engineering floor. In reality, the engineering floor contains an outside loop containing several cubicles and rooms. This is excluded because as mentioned in the AI design section, mapping an outside loop is not supported.

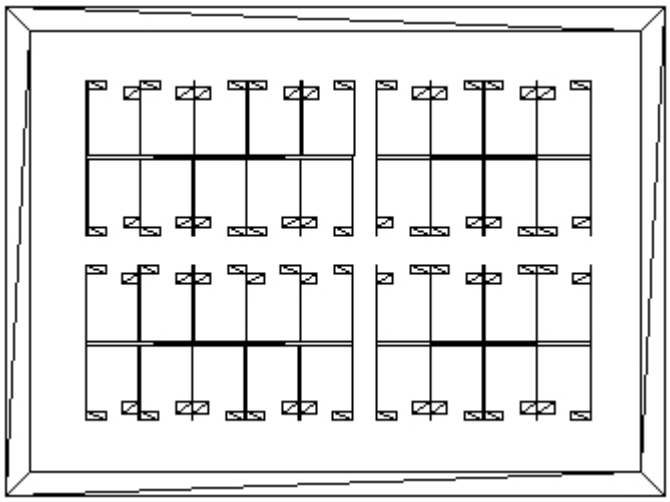


Figure 5-1 The map of the target area

Vansco Electronics engineering floor was chosen as the testing area since it was convenient to test the robot at the same place it was built. It was also chosen because it represents a very challenging area to generate a topological graph. This is because the entrance of each cubicle consists of a bookshelf and a filing cabinet that are not inline to one another. This is not the ideal situation for topological mapping which is better suited to long even hallways, rather then short staggered hallways. Problems are caused because the bookshelves and filing cabinets are short so there is not a lot of distance for the robot to align itself to the wall. If the robot can successfully work in this area, it can probably work in most other areas, that the criteria specified in section 1.3.

5.2 Simulator Results

5.2.1 Results of Mapping in Target Area

To be able for the map making to work on the real robot, the map making program has to work in the simulator consistently and accurately. Fortunately, it does exactly that, as seen by figure 5-2. The generated topological graph directly corresponds to the map of figure 5-1.

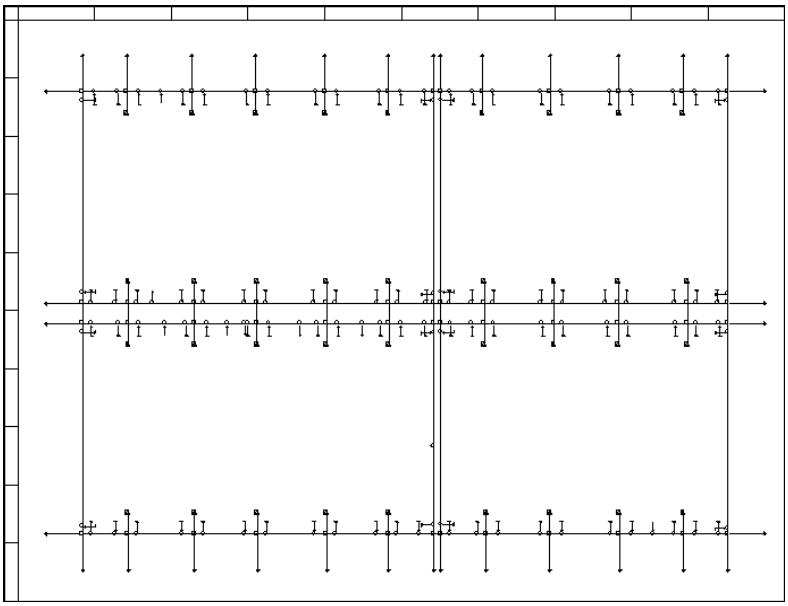


Figure 5-2 The graph made in the simulator by the robot.

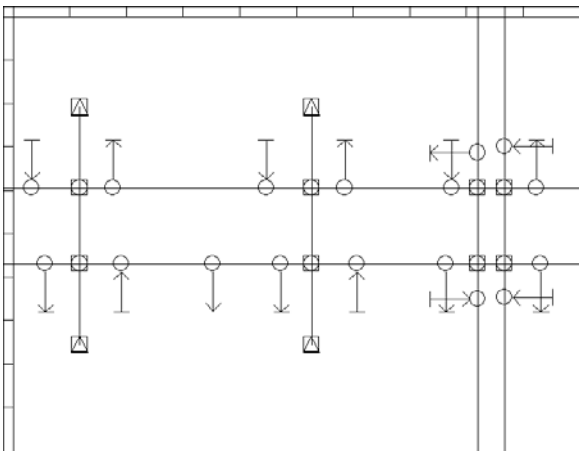


Figure 5-3 The close up of the graph.

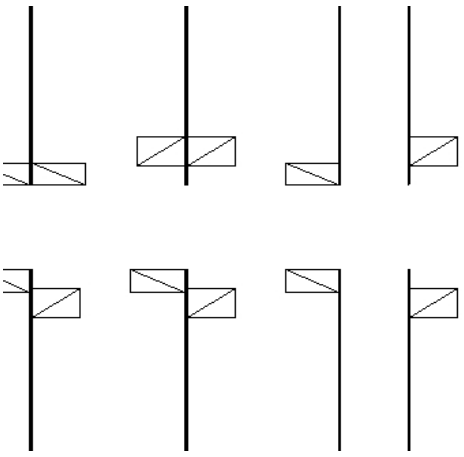


Figure 5-4 The close up of the map.

Since figure 5-3 is the first visual of the topological graph an explanation is in order. The close-up of the graph in figure 5-3 is of the centre area of the map of figure 5-1 is shown in more detail in figure 5-4. Each symbol of the graph is a node, with the lines being the links connecting the nodes. A decision node is represented by the square. The landmark node is represented by the circle, with the arrow showing information about the landmark. An arrow that points towards the wall means that the wall appears closer to the robot. An arrow that points further away means that the wall appears further away from the robot. Note that the arrow direction is relative to the direction the wall was mapped and as such they are treated the same when navigating. An arrow that has a perpendicular line on it means that the robot has either lost, or gained contact with the wall. In the case of a change in detection distance, the length of the arrow represents the magnitude of the change in distance. A node represented with a triangle in it, is an unknown node and represents a cubicle. An unexplored hallway from a decision node is represented by a very long arrow.

Notice that there are four decision nodes in the centre. As noted in the AI implementation these nodes were all cloned from one to allow pathways to exist on both sides of the hallway. Also, notice that they are connected together symmetrically. This occurs because of the way the close loop algorithm works, with basing the location of a cloned decision node to align with the previous one.

5.2.2 Results of Navigating in Target Area

With the graph that is generated by the robot simulation being accurate, the navigation works reliably in the simulator. The only problem is the one noted by the earlier discussion about navigating in an unknown (cubicle) area. Since the robot is travelling blindly in an unknown node, it is possible that the robot could get stuck and cannot find the exit. Without considering unknown nodes, the robot could navigate successfully to any point in the map nearly 100% of the time.

5.3 Robot Results

5.3.1 Results of porting the code

There are approximately 4500 lines of code that form the AI. These lines of code had to be ported from the simulator to the embedded system. Overall, the port went smoothly. There were some minor compilation issues, due to the fact that Visual C++ 6.0 is a more advanced and forgiving compiler than the embedded system one. After the code was ported, some additional

code was required to duplicate the message passing system from the simulator, to be sent over the wireless. In all, only two days were spent porting the code, and most of the time was spent on adding and testing the communication between the robot, and the AI monitor on a PC.

5.3.2 First Mapping Results

The first task in generating a map for the entire floor is to map out one loop. The bottom right corner was attempted as shown in figure 5-5. Note that figure 5-5 contains some changes in the layout because two of the cubicles were converted to one, a few days before the mapping attempt. The robot was allowed to map the area, without a single change in code. The results were not as good as expected.

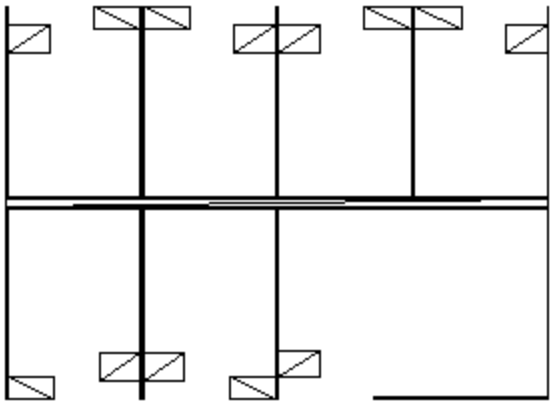


Figure 5-5 Target area to map

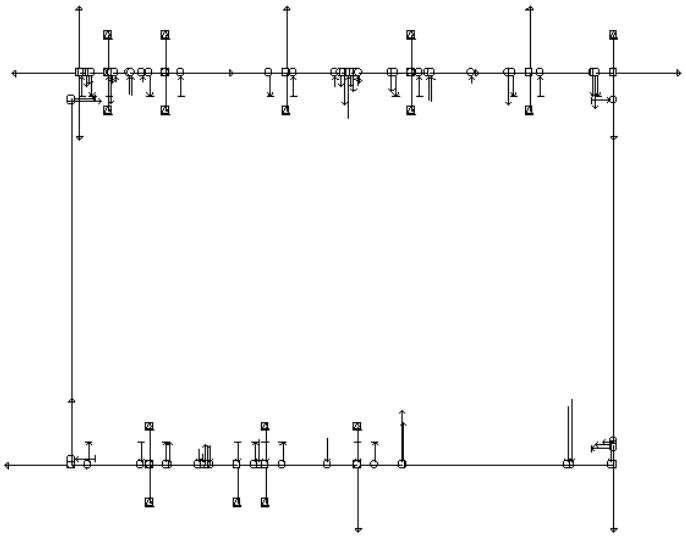


Figure 5-6 The first try at mapping a loop.

As can be seen in figure 5-6, the first graph displayed before the loop was closed has the general shape of the area, but the accuracy is low. There are both erroneous decision nodes and landmark nodes. Several major problems with the IR rangefinders caused the errors in the graph in figure 5-6. First is that the IR rangefinders performs very poorly on the black reflective surfaces of the filing cabinets. The second problem has to do with that fact that the beam width of the IR rangefinders is about 1 cm. This combined with the fact that there are gaps in the cubicles wall caused many erroneous readings to occur. There is also a problem when first detecting or losing the detection of a wall. The IR rangefinders detection transition is not smooth and when losing a wall contact, the IR may first loses the wall detection, but then may redetect the wall in the next reading, and then lose the detection again. This affect is visible by noting that near decision nodes there are always multiple landmark detections.

5.3.3 Second mapping results

To fix the problems found after the first mapping results, three minor changes were introduced in the code. The first is that the averaging for the IR sensors were increased to averaging 10 samples over 70ms. This is a two fold increase in averaging from the first mapping attempt. In addition, a gap filter was introduced, so that small gaps are ignored. The gap filter delays the output of the gain or loss of detection until several gains or loss of detections occur consecutively. This allows a gap of a few centimetres to be ignored. Third, the threshold distance for a landmark node to be reported was increased, which eliminated many of the smaller landmark nodes which may be noise related. There was one minor change to the environment, with the realization that no matter how good the IR filtering is, the IR rangefinders still do not output the correct distance when detecting the black reflective filing cabinets. To solve this problem white paper was taped onto the filing cabinets, and used to fill in some of the larger gaps that existed between objects. The results are shown in figure 5-7 and figure 5-8.

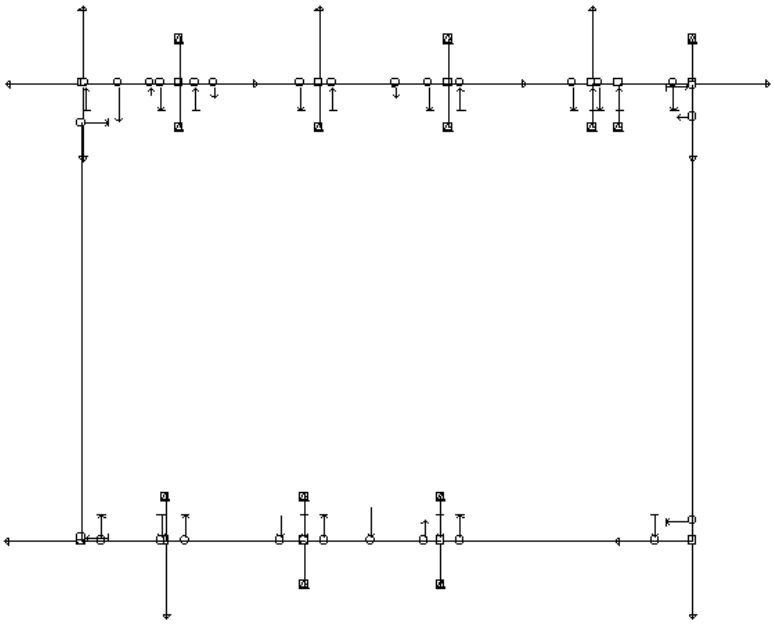


Figure 5-7 Results before the loop is closed

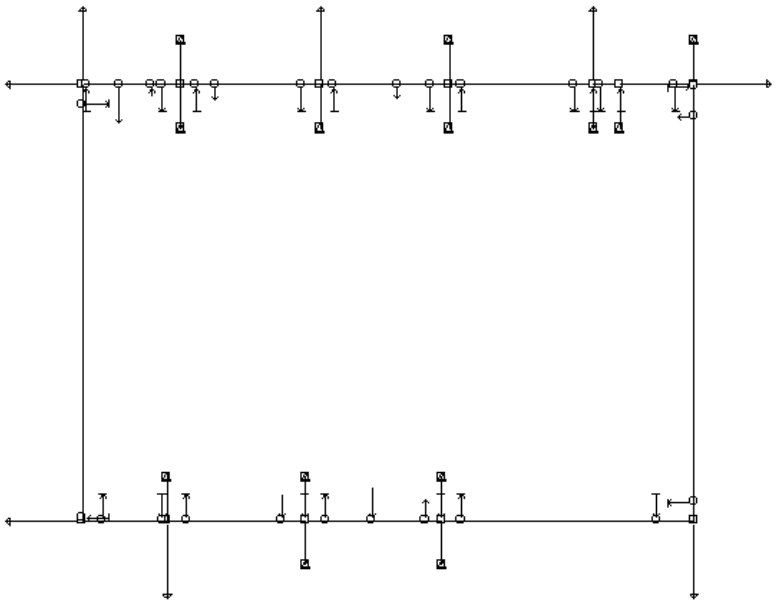


Figure 5-8 Results after the loop is closed

The results shown in figure 5-8 and 5-9 are quite good. There are still some erroneous landmark nodes when comparing the graph generated by the robot to the one generate by the simulator. However, in some cases the extra landmarks represent objects not in the map used in the

simulator that exist in the engineering floor. There is one small error in that some landmark nodes are to close to the decision nodes. The reason for this is that the sensor lag filter that was introduced caused landmarks to be reported late. Also the robot may have been slightly angled when at the decision node, which would cause it to detect the landmark sooner. There is also an extra decision node at the top right of the graph, which is attributed to an erroneous IR detection. There are also some linking errors with some of the nodes not quite linking properly. This is probably caused by an error in the code that exists, perhaps because of the differences in compilers.

As for the dead reckoning error there is a gap of about 20 cm that is closed by the close loop routines. It is a little surprising to see the error in the vertical direction. The vertical direction featured two straightaways, while the horizontal, direction features a staggered area. However, the close loop error is due to random error, not a persistent one, so it is possible that the errors in the horizontal direction exactly offset each other. As for the total error, the dx is the graph is 1026 cm and the dy is 730. The actual distance is 970 by 710. The error rate is around 5% in the X direction, with its staggered features and 3% in the Y direction which featured a straightway. This error mainly represents the fact that the robot is not moving perfectly parallel with the wall.

As for reliability during mapping, it was unfortunately low. Much of this was caused by problems with the IR rangefinders. First, there were phantom forward detections so the forward IR rangefinders were shut off. Then there were too many oscillations when the robot was attempting to align itself to the wall, which may be caused by noisy readings. The oscillations occasionally caused a collision, and often caused near misses with the wall. However, this may also be due to the fact that the motor look up table could be incorrect in comparison to the one used in the simulator. Another reliability problem is the fact that the sonar API function SNRGet3PointDistance did not always work properly when mapping. Sometimes cubicles would be classified as hallways, and the robot would turn into the cubicle. This occurs when the robot is near its maximum distance away from the hallway on the wall following side, and when the cubicle was clean. During map making the clean cubicles had their chair placed slightly closer to the doorway, to guarantee that the cubicle would be not be classified as a hallway. The loop was attempted to be mapped several times, with the robot only successfully returning to the starting position without a collision about half of the time.

5.3.4 Navigation results

Due to reliability issues in making the map of the one loop, and the time required to place white paper on all of the filing cabinets and large gaps in the engineering floor, it was decided to use the graph already generated by the simulator, which is accurate (figure 5-2). The navigator was run with only a few minor code changes. The unknown node routine was changed allowing the robot to travel deeper into the cubicle. The simulator generated graph is not entirely accurate due to the map used not being entirely accurate, and the gaps between the decision nodes are overstated. This would cause the robot to blindly travel too far between decision nodes.
Because of this, some of the decision node to decision node travel rules was changed especially with regards to shortening the distance traveled in four-decision node cluster in the centre (see figure 5-3).

The results were quite good. After initially changing some distance constants to solve the problems as mentioned in the above paragraph, the navigator proved to be very reliable. The robot was successfully able to navigate to its destination node nearly every time. It came as no surprise that the robot can navigate far better then it can make maps. Using a correct map allows for errors in the IR readings to be referenced against the map, and ignored if they are incorrect. Also, the robot is continuously moving when navigating, rather then stopping constantly when mapping the area to take sonar readings. The fact the robot is continuously moving seems to improve the overall alignment of the robot being parallel to the wall.

Chapter 6
Conclusion and Recommendations

6.1 The Feasibility of a Low Cost Robot

In addition that the robot is to make maps and navigate in them, the other goal was to achieve this at a low cost. It is difficult to gauge the entire cost of constructing the robot. The major components that were purchased are the IR rangefinders, the sonar assembly, the optical mouse, the robot chassis, and the batteries. In total, perhaps $300 was spent building the robot. If all of the electronics had to be purchased, instead of some being donated the cost would be far higher. The evaluation board is worth around $300, the LCD around $100, the wireless module around $200 and other miscellaneous parts around $200. This brings the cost to about $1100. While this price may seem expensive note that the $1100 is the retail purchasing cost. In comparison to retail prices, electronic components can be purchased at half-price or lower if purchasing in bulk through a distributor. That would make the costs of the robots components below $600.

The stated goal of the hardware design is attaining the capabilities of the Pioneer-DXe that costs around $4000 US, at a lower cost. As it turns out the robot constructed is nearly the same as the Pioneer-DXe. The processor is of the same family, and the sensors arrangement is very similar, with a forward sensor array for collision detection and a sensor on each of the sides for wall following. The goal of achieving the capabilities of the Pioneer-DXe at a low cost is accomplished. However, the low cost could be entirely attributed to the countless hours spent designing and constructing the robot, and the donations of parts.

An interesting observation is that the actual cost of producing the Pioneer-DXe is probably below $1000 US. The rest of the price reflects the cost of engineering time, and the low volume. Even if the Pioneer-DXe were purchased in enough volume to justify lowering the cost, the cost would be reduced to perhaps $2000 US. This still is too high of a cost. In order for low cost navigation robots to be produced in volume, a single sensor module, which in addition to providing all of the perception, could also be used as the main processor, would have to be available at a cost in the low hundreds of dollars. It would also have to be possible to adapt this module to control various different robot chassis.

6.2 Performance of the Robot

The robot is able to navigate reliably using a map generated by the simulator, while being close to being able to generate maps of the same quality as the simulator can. All of this is accomplished in a challenging environment. The performance shows that in the case of the IR rangefinders limited range, good software can overcome inexpensive hardware. It also shows that the detection problem of the inexpensive IR rangefinders cause problems that cannot be solved with software.

6.3 Recommendations

6.3.1 Timeline issues

Even though this project was started in August, leaving around 7 months to complete the project, time was always an issue. Around half the time was dedicated to constructing the robot, and the rest was spent on writing software. Because of time considerations, many ideas to improve the software had to be left as ideas, and are listed in this chapter. It is not recommended for an individual to pursue both the hardware and software design as an undergraduate thesis. Many of the recommendations for the AI would have been implemented if there were more time. In hindsight, it might have been worth to do an all software thesis, and not worry about being limited to the capabilities of inexpensive sensors. It is possible in the simulator, to give the robot a full sonar array, and increase the range on the IR rangefinders to several metres. However, there is still a certain satisfaction in seeing the AI work using a real robot.

6.3.2 Increase Use of Sonar

In retrospect, it should have been obvious why all of the expensive research robots use sonars instead of IR rangefinders as their primary sensors. If the robot was designed again, it is very likely that multiple sonars would be used. If only one set of electronics and multiple transducers are used, they are not much more expensive then using IR rangefinders,. However, sonars also have problems.

Another recommendation is to use the rotating sonar in tandem with the side IR rangefinder to generate side distance. The sonar is not affected by the colour of the filing cabinets, or by small gaps in walls. The sonar would have some erroneous detections, but this can be compensated by also using the IR. The sonar was intended for use as a forward collision sensor, while following walls. The reason is that the three front IRs have a beam width of 1cm, and it is possible for them

not to pick up a narrow object that is directly in front of the robot. The odds of a narrow poll obstructing the robot in the middle of a hallway are small. If more time were available, the sonar would be programmed to assist the Side IR rangefinders.

6.3.3 Use of Machine Learning

Often machine learning techniques such as neural networks and Bayesian probability are used in autonomous robots for sensor interpretation and automatically tuning constants in modules. The sonar SNRGet3PointDistance API Functions could incorporate machine learning to match the sonar distance output to pathway identification. Machine learning could also be used to act as a smart filter for the IR output. The follow wall module could benefit with machine learning that could fine-tune the adjustments and dampening parameters, for aligning the robot to be parallel to a wall. The robot would greatly benefit if elements of machine learning were added.

6.3.4 Addition of Hybrid Mapping Techniques

Due to time constraints, it was necessary to simplify the navigation once inside an unknown (cubicle) area. A good addition to this project would be the ability to switch to a metric form of navigation once inside a cubicle. A further extension would have the robot make metric maps of unknown areas, which would be referenced once the unknown area is entered. This would allow the robot to be able travel to a precise location in the cubicle.

6.3.5 Adding Refinement

Due to time constraints, many of the modules were designed overly simplistic and lack refinement. Experiments in the simulator test the robots ability to map outside loops. The follow wall module could make use of the Front Side (+/- 15 degrees) IR Rangefinders, to make adjustments to reduce collisions. The robot could be given the ability to fully recover from a collision, which would require a better collision functionality than exists now.

References

[BiTu95] T. Bilgic and I. Turksen, “Model based localization for an autonomous mobile robot,”Proceedings of the 1995 IEEE Conference on Systems, Man and Cybernetics, pp. 6

[Broo95] R. Brooks, “A robust layered control system for a mobile robot,” A. I. Memo 864, Massachusetts institute of technology artificial intelligence laboratory, pp28, Sept, 1985

[Chos01] H. Choset, “Topological simultaneous localization and mapping (SLAM): toward exact localization without explicit localization,” IEEE transactions on robotics and automation, vol. 17, no.2, April 2001.

[JoFl93] J. Jones and A. Flynn, “Mobile Robots: Inspiration to Implementation,” A K Peters, pp. 457,1993.

[NaCh99] Keiji Nagatani and Howie Choset, “Toward robust sensor based exploration by constructing reduced generalized voronoi graph,” Proceedings of the 1999 IEEE/RSJ International Conference on on Intelligent Robots and Systems, pp. 1687-1692, 1999.

[Siem96] “Siemens 16 bit microcontrollers C167 Derivatives User’s Manual”, Siemens, 1996.

[TBBF98] S. Thrun, A. Bucken, W. Burgard, D. Fox, T. Frohlinghaus, D. Hennig, T. Hofmann, M. Krell, and T. Schimdt, “Map learning and high-speed navigation in RHINO,” MIT/AAAI Press, pp. 24, 1998.

[ThBF00] S. Thrun, W. Burgard, and D.Fox,”A real-time algorithm for mobile robot mapping with application to multi-robot and 3d mapping,”IEEE Conference on Robotics and Automation,” IEEE Press, pp. 8, May 2000.

Appendix

Please see Appendix.htm on the CD-ROM, or view web page at www.worldminesweepernetwork.com/robot.htm as of March 2002.