OBSTACLE AVOIDANCE AND FEATURE POINT NAVIGATION FOR AN OFFICE DELIVERY ROBOT

BY

MATTHEW STEPHEN PEPICH

B.S., University of Illinois at Urbana-Champaign, 2007

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Electrical and Computer Engineering
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2009

Urbana, Illinois

Adviser:

Professor Mark W. Spong

ABSTRACT

This thesis presents an implementation of an office delivery robot. First, it provides motivation for this type of project and discusses the abilities a successful delivery robot must possess. Next, the robot used for this project is presented, and its relevant hardware is broken down and explained. Special emphasis is placed on the sensing system, which consists of infrared proximity sensors and incremental optical encoders. The investigation of the hardware is followed by a look into the software side of a delivery robot. This includes the user interface, wireless communication scheme, robot controller, and movement methodology. The core of the thesis follows with a discussion of feature point representation, path planning, and obstacle avoidance. Finally, experimental results are provided, and potential areas for future work are proposed.

To My Loving Parents

ACKNOWLEDGMENTS

I would not have made it this far without the support and guidance of many people. I especially want to thank Professor Mark W. Spong for his willingness to advise me over the years. His direction and assistance were invaluable. I also want to thank Daniel Block, my Mechatronics course instructor and the head of the Control Systems Lab. His aid in debugging was a great help, and for that, I am very grateful.

TABLE OF CONTENTS

CHAPTER 1: INTRODUCTION

Robots have become increasingly common in today's society. They build cars, clean homes, and even perform surgery. As the years go by, they grow ever more complex, doing jobs that humans either cannot or do not want to perform. Most robots, though, are used for basic tasks that require little to no human attention. One such task is delivery in an office environment.

In an office, delivery can encompass a great many things. Large companies might need someone to carry interdepartmental mail throughout the building. Perhaps a package needs to be shipped to maintenance, or an employee needs to turn in some paperwork. Regardless of the task, carrying something from one place to another requires little thought, but often a fair amount of time. Automating these deliveries might be a great way to save money in the long term.

If the goal of a delivery robot is to cut costs, it should be designed to be as inexpensive as possible. Systems such as GPS, while effective at tracking position, are expensive. Cheaper ways to aid navigation include following strips of tape along the floor, orienting to colored markers, and following infrared lights. However, all of these require a change to the environment, which company management may not allow. Therefore, this thesis will focus on the issue of low-cost, noninvasive office navigation.

1.1 Project Requirements

An office hallway is a very dynamic environment. People will be walking around, perhaps going to the restroom or stretching their legs. Others might be standing around in groups talking, unaware of their surroundings. There may be a large trash container sitting unattended, or someone standing on a narrow-legged ladder while replacing a light bulb. Whether big or small, moving or stationary, aware of their surroundings or ignorant, all of these are obstacles to a delivery robot.

A delivery robot must deal with not only a variety of obstacles but also a changing environment. For instance, a normally closed door may be open one day, or a surface may have been recently waxed. In cases such as these, the robot may find itself confused. If it cannot deal with a little uncertainty, it could be left wandering around until its battery dies while angry employees prowl the halls in search of their mail.

To perform a task in such an environment, there are several requirements. The most basic of these is a method of communicating with the robot. Dragging hundreds of feet of Ethernet cable behind it would probably be a bad idea, so the robot's link must be wireless. This wireless user interface must allow the user access to important information, such as the robot's position. It also must perform essential functions such as choosing a delivery destination, setting variables, and even turning the robot on.

Another critical requirement is a way to sense the environment. The robot must sense all of those obstacles mentioned earlier in order to avoid them. An inexpensive way to do this is with infrared proximity sensors. Several of these must face forward and be packed tightly enough together to sense narrow objects. In addition, some should face sideways to sense doorways and other openings.

Finally, the robot needs to track its own movement. This can be done by using incremental optical encoders attached to each of the two motor shafts. These encoders keep track of how far the wheels have turned. Combined with other information, the robot is able to estimate its relative position, velocity, and angle.

1.2 Overview of Chapters

The robot used for this thesis will be introduced in Chapter 2. After this brief introduction, a discussion of the I^2C communication protocol used for the infrared proximity sensors will follow. The chapter will conclude with a discussion of those sensors, as well as the incremental optical encoders used for position tracking. Chapter 3 will look at the software needed to run the robot, from the user interface to motion control. The focus here is on robot operation. The navigation component is addressed in Chapter 4. The chapter begins with choosing feature points as a navigation aid. It discusses the entire decision process for setting the movement speed and direction that will help the robot reach its eventual goal. This includes path planning, setting the velocity and turn rates, reaching checkpoints, avoiding obstacles, and updating position. Next, Chapter 5 discusses the performance of this implementation. It demonstrates successful object avoidance, recovery from missed checkpoints, and tolerance to minor positional errors. Finally, Chapter 6 wraps things up and proposes areas for future improvement. The code written for this project can be found in the attached appendices.

CHAPTER 2: ROBOT COMPONENTS

Most of the hardware used for this project was already built. As a result, this section will be decidedly brief. The focus will be on the sensing platform and infrared sensors, which were the primary hardware modifications made during this project.

2.1 Robot Body

The robot used for this thesis is known as a Segbot, with both name and design inspired by the Segway. It was built in the College of Engineering Control Systems Lab (CSL) at the University of Illinois at Urbana-Champaign by a group of students in 2004 under the direction of Dan Block [1]. The initial design had only two wheels – one fixed to each side. Unlike the Segway, however, the Segbot has no counterbalance underneath. To keep from falling over, it rocks back and forth about its equilibrium point, similar to an inverted pendulum.

For this thesis, the Segbot was modified by adding a front platform with a caster wheel underneath. The addition of the platform shifted the center of mass forward and the extra wheel allowed it to stay balanced while stationary. The platform rests about one inch above the ground and is used to mount the infrared sensors. The modified Segbot will be referred to as the Deliverybot.

Other relevant components from the original Segbot include:

- Wireless card model AC4490 from Aerocomm, communicating at 57 600 baud

- LCD screen with two rows capable of displaying 20 characters each
- DSP model TMS320C6713 from Texas Instruments
    - Processor speed of 225 MHz delivering 1800 MIPS and 1350 MFLOPS
    - 512K words of Flash and 16 MB SDRAM
    - Embedded JTAG support via USB

2.2 Sensors and Communication

This section will discuss the two types of sensors that were used: incremental optical encoders and infrared (IR) proximity sensors.

The original Segbot came equipped with an encoder on each wheel. Proximity sensors, though, still needed to be added. To be most effective at avoiding narrow obstacles, four sensors were chosen to face forward. Two sensors were chosen to face right to detect openings and aid in navigation. The only way to add that many sensors to the Segbot was to use the Inter-Integrated Circuit (I^2C) bus.

To understand how the sensors were added, the following subsections will first introduce the I^2C protocol. Once the basic terms and functionality are introduced, the way messages are sent is described. A discussion of the IR sensors and incremental optical encoders will follow.

2.2.1 Overview of the I^2C bus

In order to explain I^2C functionality, it will be beneficial to define some terms. The I^2C bus uses two bidirectional lines, which are shared among all devices that are connected [2]. These two lines are the clock line (SCL) and data line (SDA). The transmitter is the device currently controlling the data line. The receiver is the device waiting to receive that data. The master device is the one that initiates the message and controls the clock line. Finally, a slave is any device being addressed by a master.

However, some of these distinctions are mutable. For instance, suppose the DSP wants to change a setting on an attached IR sensor. Then the DSP will be both the master and the transmitter, and the sensor will be the slave and receiver. If the DSP later wants to know what the IR sensor is reading, the DSP becomes the receiver while the sensor is transmitting its data, but because it is initiating the request, the DSP is still the master.

When no communication is occurring over the I^2C bus, pull-up resistors tie both the clock and the data lines to 5 V. To put a zero on the bus, a connection to ground is made using transistors. When not being pulled low, the transistors are in a high-impedance state. This is critical to prevent devices not being addressed from interfering with communication. Since the bus is pulled high naturally, and since only one device is allowed to communicate at a time, a transmitter being in high impedance is the same as sending a one.

2.2.2 Sending a message over the I^2C bus

When a device wishes to communicate with an attached device, it must first gain control of the bus. This is done by sending a start bit, which is a high-to-low transition of the data line while the clock line is high. Once the start bit is sent over the data bus, the device begins controlling the clock bus, which runs at the default speed of 100 Kbits/sec.

The device must then get the attention of the correct slave. The I^2C protocol requires every device to have its own 7-bit address. To talk to a device, the master sends the desired address over the data bus. Since all communication over the I^2C bus is in 8-bit increments, an additional bit is sent to indicate whether a read or write is desired.

Next, the slave must send an acknowledge bit to say it recognized the message. The master relinquishes control of the data bus by allowing it to be pulled high. If everything went well, the slave will pull the bus low to acknowledge the message.

If the master requested to write information to the slave, it will again take control of the bus and send out the data in 8-bit increments. Each byte must be followed by an acknowledge signal by the slave. This pattern will repeat until a stop condition is met.

On the other hand, if the master sent a request to read, then the slave will maintain control of the bus after the acknowledge signal. In this case, the transmitter/receiver roles are reversed, and the slave transmits one byte at a time while the master acknowledges them. Despite being the receiver, the master still maintains control over the clock line.

Finally, a stop condition must be met. Like to a start bit, the stop bit is a change in the data line while the clock line is high. The difference is that a stop bit is a low-to-high transition. Once the stop bit is sent, the master relinquishes control of the bus.

If the master still had other messages to send, though, it could have sent another start bit instead of a stop bit. After that bit, the entire process would start over with the master able to address a new device. I^2C functionality is summarized in Figure 2.1. Now that the reader has a basic understanding of the I^2C communication protocol, the following section will introduce the sensors that use it.

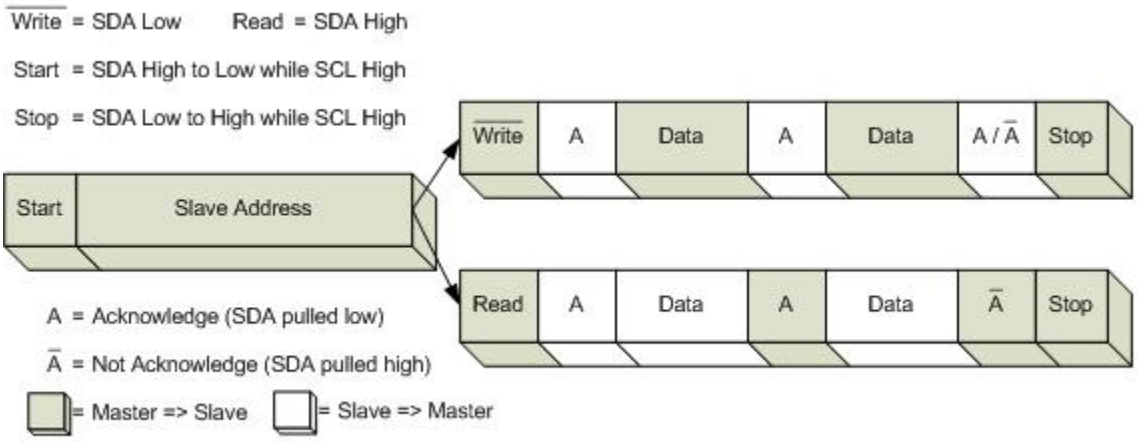


Figure 2.1: Summary of an I^2C message.

2.2.3 Infrared proximity sensors

The proximity sensors used on the Deliverybot can be seen in Figure 2.2 and Figure 2.3. By emitting an infrared beam of light and measuring the strength of the reflection, an IR sensor is able to estimate how far away a target is. These particular sensors are accurate at sensing objects 1 foot away or less. Objects 1 to 2 feet away can have a few inches in error, and anything over 2 feet away is probably unreliable.

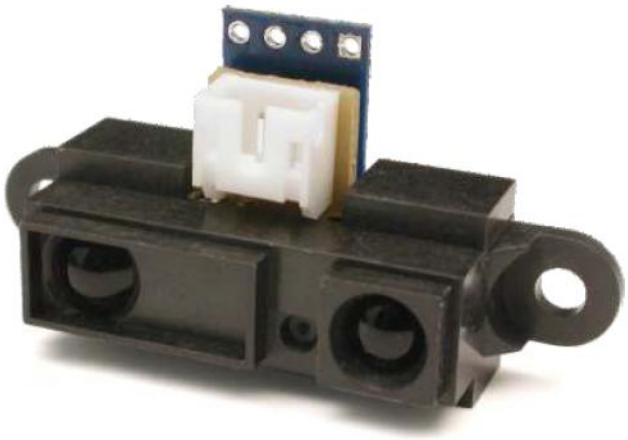


Figure 2.2: Front of the IR sensor used on the Deliverybot.

Figure 2.3: Rear of the IR sensor used on the Deliverybot.

A closer look at Figure 2.3 reveals four inputs near the top of the sensor. As mentioned earlier, these IR sensors communicate over the I^2C bus using the data and clock lines. These are the middle two holes. The two on the outside are the power and ground lines for the sensor. The pinout for the sensor can be seen in Table 2.1.

Table 2.1: Infrared sensor pinout

Pin #	Function
1	GND (signal ground)
2	SCL (I^2C clock)
3	SDA (I^2C data)
4	V_{cc} (+4.5 to +5.5 V)

Theoretically, any number of sensors can be strung together and plugged into to a single connector on the robot. In practice, capacitance limits the number of devices that can be attached to a single bus. A longer bus length can also introduce extra noise to the circuit. To deal with this problem, the sensors were divided into two groups and connected to two separate I^2C ports.

Another practical limitation is the number of available addresses. A seven-bit addressing scheme means there are 2^7 different choices. However, the IR sensor's construction only allows eight different addresses to be used. The address is chosen by soldering connections across three separate jumpers, which can be seen in the lower middle of Figure 2.3. The bottom half of each jumper is connected to ground, while the top half is connected to V_{cc} via a pull-up resistor. Placing a ball of solder across a jumper will bring the voltage on the top half down to zero, and doing so sets one bit of the address. The eight available addresses are listed in Table 2.2, along with the jumper combinations to set them.

Table 2.2: The eight possible addresses for each IR sensor

Device Address	JP1	JP2	JP3
0x20	0	0	0
0x22	0	0	1
0x24	0	1	0
0x26	0	1	1
0x28	1	0	0
0x2A	1	0	1
0x2C	1	1	0
0x2E	1	1	1

1 in the JP1, JP2, or JP3 column indicates a solder jumper. [3]

Initially, all eight available sensors were going to be used. However, several problems arose in adding them to the Deliverybot. First, the pull-up resistors on each jumper number 3 were not functioning properly. Even without a solder ball, they still read low occasionally, which resulted in conflicting addresses. To fix this, the top halves of the empty jumpers were connected to V_{cc} with a small resistor on the offending sensors.

A second problem was that one of the addresses was already being used. Despite the 2^7 different possibilities, the address 0x28 happened to be used by the LCD screen. While this could have been changed, settling for seven working sensors was deemed preferable to modifying the existing Segbot hardware.

Finally, two of the sensors were exceptionally prone to noise. At irregular intervals, they gave readings in the low inches regardless of what they were looking at, most likely due to an internal grounding problem. Because the remaining six sensors were proving sufficient for the task, this issue was left unresolved.

Figure 2.4 shows the final sensor setup from a view looking down from in front of the robot. The rear-facing sensor on the left side of the robot (at the right in the figure) is

the one with stability issues. Because the wires were already cut and connected, it was left on the robot and ignored rather than removed.

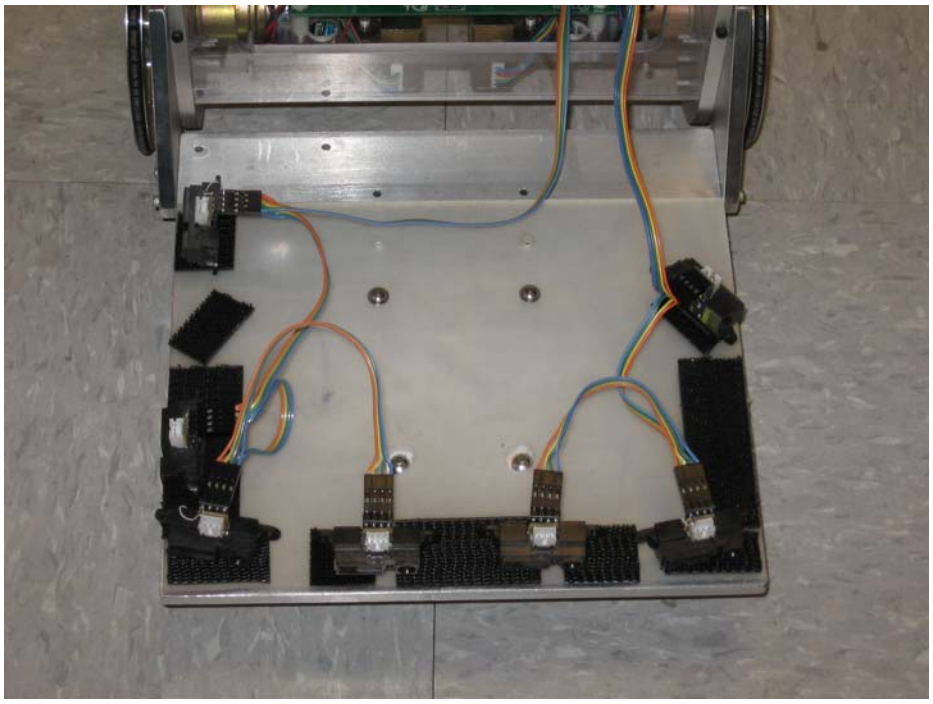


Figure 2.4: Final IR sensor layout.

2.2.4 Incremental optical encoders

To track its position in the world, the Deliverybot uses incremental optical encoders attached to the wheels. This encoder consists of a circular disk with two rows of slits at the far edge. On one side of the disk is a pair of light sources, and on the other side is a pair of photodetectors. A high pulse is produced when the slits line up with the photoemitter-detector pair. By measuring the frequency of the resulting pulse trains, the rate of rotation can be determined. An example of this type of encoder can be seen in Figure 2.5.

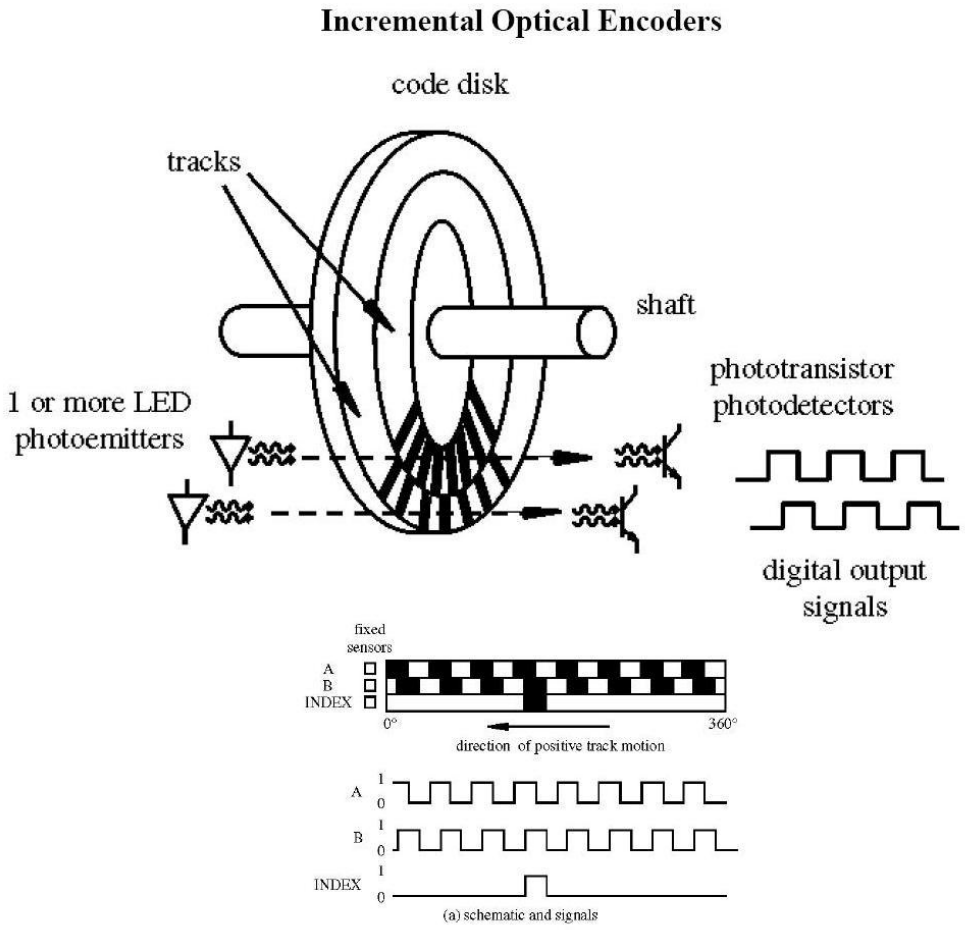


Figure 2.5: Example of an incremental optical encoder [4].

Because slits A and B are offset by one-half the slit width and therefore able to produce pulse trains 90 degrees out of phase from each other, the direction of rotation can be determined as well as the speed. For the moment, assume that clockwise rotation results in a low-to-high transition of A while B is low. Then direction can be determined with a simple method requiring only one D flip-flop. If A is connected to the input line of the flip-flop, and B is connected to the clock line, then anytime the encoder is turning

clockwise, the flip-flop will output a one. Similarly, counterclockwise rotation results in a zero.

In some encoders, an additional track, called an index, is included. This track has only a single slit cut in it, which allows the user to define a hard "home" position. It can also be an easy way to count the number of rotations. However, the encoders that came with the Segbot do not utilize an index, so all calculations made with the encoder values are relative.

The encoders are the primary means in which the robot tracks its position. An internal estimate is kept by the robot, and a change in the encoder readings results in the position estimate being updated. Keeping track of the current position in this manner is called *dead reckoning* and is somewhat prone to errors. Later chapters will propose ways of fixing these errors using the infrared sensors and knowledge of the environment.

CHAPTER 3: SOFTWARE AND OPERATION

Before complicated issues such as path planning and obstacle avoidance can be discussed, some basic functionality must be introduced. This chapter will introduce the user interface. The wireless communication scheme will also be covered, which is required to perform basic tasks such as providing the Deliverybot a destination. After that, the controller will be presented. The Deliverybot uses a type of proportional-integral (PI) controller. Additional checks and compensations are also introduced that are required by the dynamics of Deliverybot. This chapter will conclude by covering the two methods of traveling utilized by the Deliverybot: *goto-xy mode* and *wall following*.

3.1 User Interface

The most basic function the user interface might have is giving the robot a destination and sending it on its way. Although the destination could be set using onboard switches, remote activation might be a useful feature.

Other information might also be useful for the robot to have. The weight of the package, for instance, has an effect on the vehicle's turning performance. Tests have shown that adding extra weight results in a smaller angular change than expected. One solution is to precompute turning coefficients for a wide range of loads. Then whoever loads the robot can first weigh the load and transmit that value to the robot. However, this too could be done onboard with the addition of a weight sensor.

In addition to instructing the robot, the user might also want to receive information from the robot. A user interface could track the robot's position throughout the building. A floor map displaying the robot's coordinates could be made available so that a person expecting a package could track its progress. The display could also throw up flags when it encounters obstacles and other unexpected situations.

3.2 Wireless Communication

In order for information to be passed between the robot and computer, each needs to have access to a wireless card. The Segbot came equipped with an Aerocomm Wireless card, and it was a simple matter to attach a second one to the PC and connect them together. The following subsections discuss how each device transmits and receives information.

3.2.1 Deliverybot transmit

The first step in sending a message is to compile it and convert it to a string. To reduce the effect of errors, only five variables were sent at a time. They were preceded by a packet number (to identify the variables being sent) and separated from each other by the space character. A new packet is sent every 100 msec, which is slow enough to allow the previous message to be received, and fast enough to allow smooth updating.

When the string is finally chosen, it is passed to a function that adds a start character to the beginning of the string and a stop character to the end. These two characters, 253 and 255 (decimal) respectively, are used throughout the communication process to identify the beginning and end of messages. The message is passed to a

function that transmits one character at a time via the wireless card. This function is part of the original Segbot code; no modifications were made for the Deliverybot.

3.2.2 Deliverybot receive

Receiving a message is slightly more complicated. To start, a function waits indefinitely until the wireless card receives a new character. Initially, it ignores any character other than the start character 253. Once that character has been found, each subsequent character is stored in an array. When the stop character 255 is received, the array is null-terminated and a semaphore is posted, letting another function know a message is ready. The array is cleared and the message aborted if 400 characters are received before a stop character is found, so several smaller messages are preferred over one large one.

Once the message has been received and the semaphore posted, another function wakes up and uses the message. The message is broken into segments separated by the space character. By comparing this first segment to various variable names, the function is able to gather information from the PC and store it into variables.

For instance, the message “XY 5 7” can be tokenized twice by the space character. The first string, “XY”, is searched for in a list. When found, the second string token, “5”, is converted into a float and stored as the current *x* position variable. Likewise, “7” is converted and stored as the *y* position. Many things can be done this way, from turning the car on or off to manually driving it using the keyboard.

3.2.3 User interface transmit

The user interface was created using Microsoft Visual Basic. Both sending and receiving data require Visual Basic's Microsoft Comm Control 6.0 object. Adding this object allows serial communication between the user interface and the wireless card.

Once this object is connected to the wireless card, sending a message is as simple as one line of code. If the MSComm object is named SerialCom1, then the line "`SerialCom1.Output = Chr(253) & "XY 5 7" & Chr(255)`" will send a start character, followed by the message "XY 5 7" and a stop character. This will result in the robot's position variables being changed, as mentioned in Section 3.2.2.

3.2.4 User interface receive

When receiving a message, Visual Basic again uses the MSComm object, this time waiting for a receive event to occur. When it does, several things occur. First, the message received is stored in an array. Next, the front characters are removed one at a time until a start character (253) is found. When it is found, the rest of the received characters are stored in an array, which is terminated when a stop character is found (255). If either the start or stop character is never found, then the entire message is discarded.

If everything happens as it is supposed to, then a message sent from the Deliverybot is sitting in an array. Similar to the way the Deliverybot receives messages, the user interface splits the messages into components using the space character. Then by

using the packet number, the remaining five numbers are stored in the appropriate variables.

3.3 Robot Motor Control

The most basic controller imaginable is an on-off switch. If the robot is moving too fast, turn the motors off. If it is moving too slow, turn the motors on. When implemented as a control system, this method is called pulse-width modulation. However, a robot driven this way would always be slightly ahead or behind the desired position, which makes this a poor control system.

Instead of being simply on or off, the motors could always be on proportionally to the error: if the robot is lagging slightly, give it a little extra speed; if it is lagging a lot, then give it a lot of extra speed. Because the speed varies proportional to the error, the robot will have less overshoot and will not oscillate around the target as before. If e is the tracking error and k_p the control gain, the control law for such a proportional controller is:

$$u = k_p e \tag{3.1}$$

Unfortunately, a system with a proportional controller will end up with a steady-state offset when presented with a constant reference input. [5] Turning up the proportional gain will improve this error, but at the expense of damping and eventually even stability. While slight overshoot might be tolerable, instability is unacceptable, so another term must be added to the controller.

One way to remove the steady-state error is to integrate over all past error values. This is called integral control, and when combined with the proportional term, the resulting controller is referred to as a proportional-integral (PI) controller. The PI control equation is:

$$u(t) = k_p e + k_i \int_{t_0}^{t} e(\tau) d\tau \tag{3.2}$$

The following subsections will detail how the delivery robot implements motor control using the function "`PIVelControl (float vref, float turn)`."

3.3.1 Finding velocity, angle, and position

To use Equation (3.2), four things are needed: the proportional gain (k_p), the integral gain (k_i), the error, and the integral of the error. The gains k_p and k_i are constants. For the Deliverybot, they were found by initializing them to zero and increasing them until satisfactory performance was achieved. Finding the error and its integral are more complicated and require knowledge of the current velocity and angle.

The error value is the difference between the actual and desired velocities. To find the actual velocity, the function uses the change in position of the encoders (see Section 2.2.3). Rather than outputting the pulse trains directly, the encoders count the absolute number of pulses that have passed since powering on. This value is then read by the Deliverybot as a 24-bit number and converted to a unit that will be referred to as an encoder tick.

Once the controller knows the number of encoder ticks, the value is converted into a more useful unit. Because the floor in the CSL building uses one-foot-square tiles, the units used to track position are feet. Table 3.1 lists the conversion factors for each motor on the Deliverybot. The slight discrepancy between the two values comes about because the wheels are not identical. They likely have a very slight difference in radius, and this correction, while seemingly minor, is quite useful in reducing dead-reckoned positional error.

Table 3.1: Constants used to convert the values from the encoders to feet

	Approx. Encoder Ticks per Foot
Motor 1	-58.97346
Motor 2	-59.0000

Next, the controller must calculate the robot's velocity. At the end of every function call, the old encoder values are saved. By subtracting the current encoder values from the previous values and then dividing by the time elapsed, the velocity of each wheel can be calculated. This function is called every millisecond, so a constant 1/1000 can be used for the time elapsed. Finally, the center-of-mass velocity is calculated as the average of the two wheels' velocities.

$$
\begin{aligned}
v_1 &= \frac{\left(\mathrm{enc}_1 - \mathrm{enc}_{1\mathrm{old}}\right)}{0.001} \\
v_2 &= \frac{\left(\mathrm{enc}_2 - \mathrm{enc}_{2\mathrm{old}}\right)}{0.001} \\
v_{cm} &= \frac{(v_1 + v_2)}{2}
\end{aligned}
\tag{3.3}
$$

However, this value is not always accurate. Because the numbers of pulses from the encoders are read in as 24-bit values, there will be problems when the number rolls over. A very tiny motion can result in a calculated velocity of many hundreds of tiles per second, and this can cause instability in the controller. To deal with rollover problems, the old velocities are used whenever a new velocity over 100 tiles per second is found.

After the velocity has been found, the next step is calculating the angle. Equation (3.4) shows how this was done. A constant of 0.07 was found by manually turning the robot 90 degrees and measuring the encoder values. Additional code sets the angle between –180 and 180 degrees. The angle is also stored in radians for future use.

$$\theta_{\text{deg}} = \theta_{\text{deg}} + 0.07\left(v_2 - v_1\right) \tag{3.4}$$

The next step is tracking the robot in the *xy*-plane. While the position is not needed for the controller, this is the most logical place to calculate it. By using the angle, the velocity can be separated into its *x* and *y* components.

$$\begin{aligned} v_x &= \frac{\left(v_1 + v_2\right)}{2}\sin\left(\theta_{\text{rad}}\right) \\ v_y &= \frac{\left(v_1 + v_2\right)}{2}\cos\left(\theta_{\text{rad}}\right) \end{aligned} \tag{3.5}$$

Multiplying the velocity by the time elapsed gives the change in position. Adding this to the old position yields the robot's current location.

$$\begin{aligned} x &= x + 0.001v_x \\ y &= y + 0.001v_y \end{aligned} \tag{3.6}$$

The position, velocity, and angle variables are saved for the next iteration.

3.3.2 Checking for legal inputs

The controller function has two inputs: the desired reference velocity and the desired turn rate. However, these inputs are not guaranteed to be realizable. Trying to move at 100 tiles per second will most certainly not work. Likewise, an extremely high turn rate will probably tip the robot over. For both safety and stability, the magnitude of the velocity is limited to 4 feet per second and the turn magnitude is limited to 2.

Another consideration that comes from the Segbot's unique construction is acceleration. Accelerating fast enough can cause the robot to tip over backwards. This is dealt with indirectly by limiting the reference velocity to no more than 1 foot per second plus the current speed.

```
if(vref > velocity + 1.0) vref = velocity + 1.0;
```
(3.7)

3.3.3 Calculating PI coupled control effort

Once the robot velocity, reference velocity, and turn values are known and stable, the error value in Equation (3.2) can be calculated. There will be two error terms, one for each motor. The reference velocity in each term is adjusted proportionally to the error in the turn rate (esteer) before the error in each wheel velocity is calculated (e1, e2). Because the controller is run every millisecond, the integral control can be handled by a

running sum of the error (e1s, e2s). Finally, each of these terms is adjusted by a constant and combined to produce the control effort. The code is listed below, and a block diagram of the controller can be found in Figure 3.1.

```
esteer = v2 - v1 + turn;
e1  =  vref - v1 + Kp_turn*esteer;
e2  =  vref - v2 - Kp_turn*esteer;
e1s =  e1s + e1;
e2s =  e2s + e2;
u1  =  Kp*e1 + Ki*0.001*e1s;
u2  =  Kp*e2 + Ki*0.001*e2s;
```

(3.8)

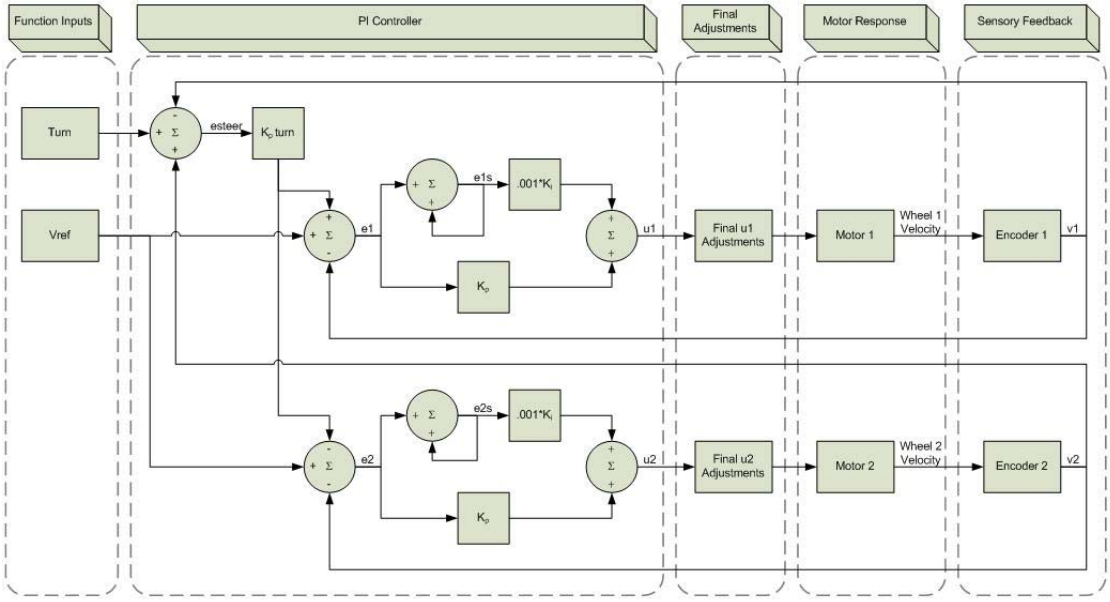


Figure 3.1: The PI controller implemented in the Deliverybot.

3.3.4 Final control effort adjustments

Once the control effort has been calculated, a few final adjustments must be made. First, steps must be taken to control integral windup, which occurs when the robot cannot move fast enough to catch up to its desired position. The integral term gets larger and larger from consistently being behind, and, if left unchecked, results in instability. This can be prevented by reducing the integral term when it gets too large, as in Equation (3.9).

```
if (fabs(u1)>10.0) e1s = e1s * 0.99;
if (fabs(u2)>10.0) e2s = e2s * 0.99;
```
(3.9)

Next, steps can be taken to reduce the effects of friction. Four coefficients were calculated to deal with both static and kinetic friction, while driving both forward and backward. These coefficients were calculated by pushing the robot around with the control effort set to zero. When they are set correctly, the slightest push should send the robot moving in the proper direction at constant speed. To prevent friction compensation from blowing up when the control effort was added, the forward values were reduced to 10% and the backward values to 30%. The code used is included below, with the coefficients listed in Table 3.2.

```
if (v1>= 0.0){u1 = u1 + Vpos*v1 + Cpos;}
else         {u1 = u1 + Vneg*v1 + Cneg;}
if (v2>= 0.0){u2 = u2 + Vpos*v2 + Cpos;}
else         {u2 = u2 + Vneg*v2 + Cneg;}
```
(3.10)

Table 3.2: Friction compensation coefficients

	Forward (-pos)	**Backward (-neg)**
Kinetic (V-)	0.64 * 0.10	0.72 * 0.30
Static (C-)	2.7 * 0.10	-2.6 * 0.30

Finally, the control effort sent to the pulse-width modulators is reduced to lie within the legal limits of ±10, as shown below.

```
if (u1> 10) u1 =  10.0;
if (u1<-10) u1 = -10.0;
if (u2> 10) u2 =  10.0;
if (u2<-10) u2 = -10.0;
```
(3.11)

3.4 Driving Methods

One common method of locomotion is to ignore the surroundings and drive straight toward the destination. The Deliverybot calls this *goto-xy* mode. It was written by a teaching assistant in the University of Illinois Control Systems Lab. The code is a variation of a gradient descent algorithm adapted for path planning use [6]. Unfortunately, this method is made complicated by uncertainty. If the robot is not sure precisely where it is when it starts, then it cannot be any more certain of ending in the correct location.

As an illustration of the danger of uncertainty, assume that the robot has in fact started at the proper coordinates. However, instead of heading due east as expected, it is

angled 10 degrees to the north. If the target is only one foot away, this will result in the robot ending 0.175 feet away from where it thinks it is. As this is only a fraction of the robot's width, this slight error could likely be ignored. If, however, the robot were to travel 10 feet in this direction, it would end up 1.75 feet off target. This second scenario, illustrated in Figure 3.2, is no longer trivial and could be the difference between passing through a doorway and crashing into the wall.

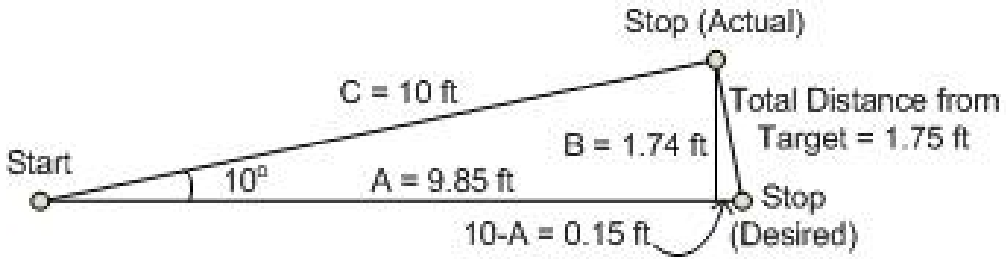


Figure 3.2: The danger of small uncertainties in goto-xy mode.

With this in mind, it seems a poor idea to drive long distances without taking advantage of the surroundings. When people find themselves in a long hallway with low visibility, the first thing they might do is find a wall to use as a reference point. If they follow the wall long enough, they would eventually come to the end of the hallway. This is the idea behind *wall following.*

By using sensors facing perpendicular to the direction of travel, wall following allows the robot to traverse the hallway a set distance away from the wall. Moving in this method is much more error tolerant, since even if the robot's internal angle is slightly off, the wall will always lead in the correct direction.

When wall following, the velocity is set at a constant 1.5 tiles per second. The turn value is proportional to the difference between the ideal wall distance and the actual wall distance. This distance is measured using the rearmost of the two right-facing

sensors. Additionally, both sensors can be used in tandem to provide information that would not be available from one alone (such as when the robot's heading is not parallel to the wall). Without this information, the robot will spend unnecessary time oscillating between too far and too close to the wall before eventually running parallel to it. The final equation is given below, with the ideal wall-following distance set to 11 inches.

$$\text{Turn} = 0.10\left(11 - \text{IR}_{RR}\right) + 0.05\left(\text{IR}_{RR} - \text{IR}_{RF}\right) \tag{3.12}$$

The main problem with wall following is that there is not always a wall to follow. Hallways are filled with doorways, and the robot should not enter every room it comes upon. A successful navigation scheme intelligently combines wall following with goto-xy mode.

CHAPTER 4: NAVIGATION AND DECISION MAKING

This chapter will present a navigation scheme for an office delivery robot. It will begin with a more detailed discussion of the plan of the building. Feature points that can be used for navigation are extracted and used to plot a course from the home position to various destinations.

Once a path is chosen, the next step is taking it. By using the two methods discussed in Section 3.4, the robot will either follow the wall to the next checkpoint or drive directly toward it. However, additional uncertainties must be taken into consideration, so several modifications must be made to these methods before using them.

Next, it must be determined whether the robot has reached the target checkpoint. Depending on the state of the robot, this may entail finding a wall in front of it, detecting an opening on the right, or simply being close to the target coordinate.

Once the foregoing determinations have been made, the robot must check for obstacles. If nothing is in its way, then the robot should continue on to the next checkpoint as planned. However, collisions cannot be tolerated. If an obstacle is detected, new commands must be issued to maneuver the robot around it.

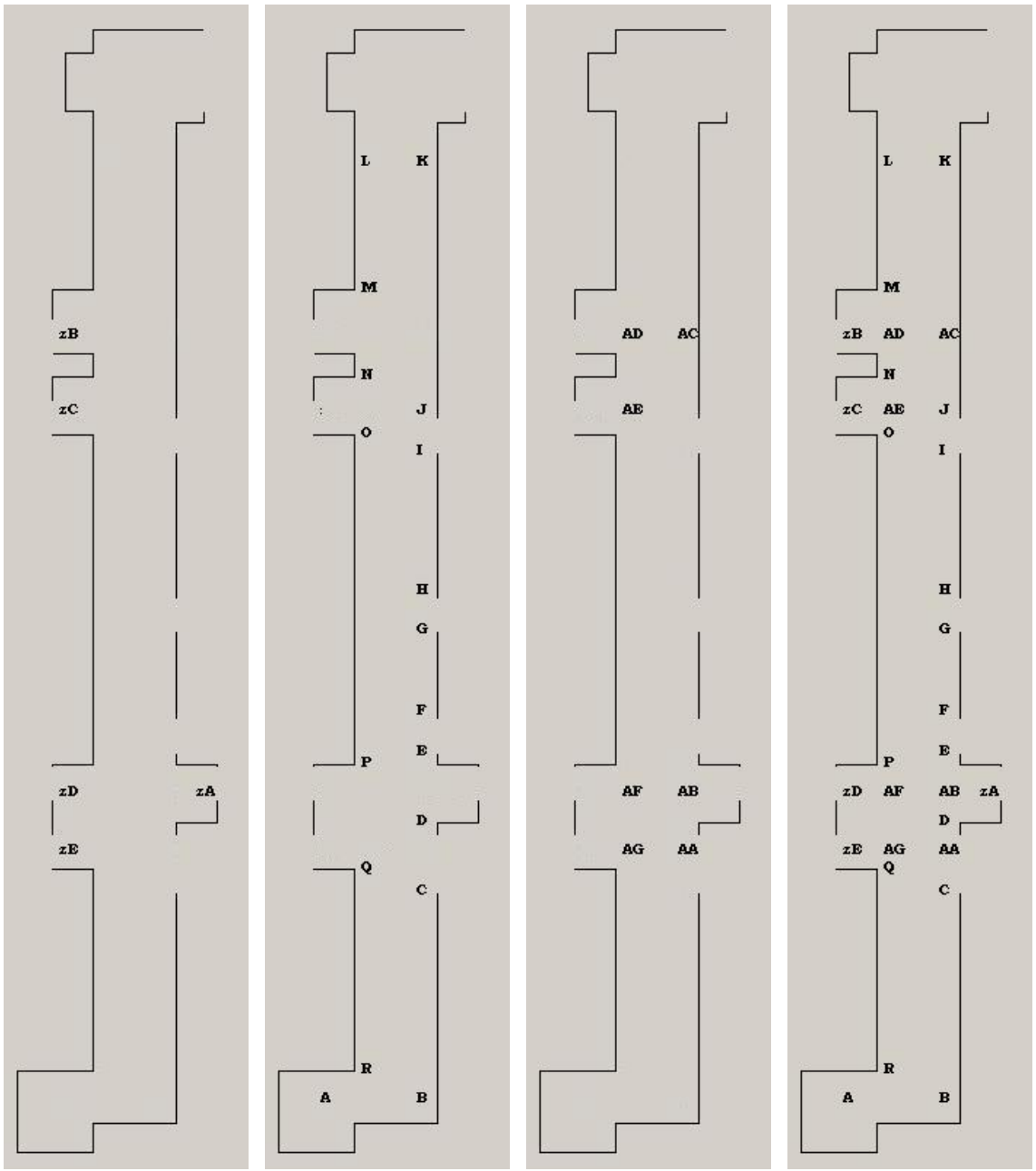
Finally, position tracking through dead reckoning can quickly become unreliable. This chapter will conclude by presenting several methods of updating the robot's position and angle. If done often enough, this updating should prevent large positional and angular errors from building up and causing the robot to become lost.

4.1 Feature Points and Path Selection

For performing tests in the CSL building, five different doorways were chosen as possible destinations (Figure 4.1a). The home location was chosen to be a small opening off the main hallway (point A in Figure 4.1b). To enable the robot to travel from the home position to a destination, checkpoints were created along the way at feature points.

Feature points are locations in the hallway that are discernible by the IR sensors. If the robot knows it is near a specific feature point, then it can sense when it arrives at it. A map of these points, which consist mainly of corners and doorways, can be seen in Figure 4.1b. Additional points were added (Figure 4.1c) to aid in crossing the hallway and approaching destinations head-on.

As just mentioned, these checkpoints exist as a navigation aid. When moving from the home location to a destination, the robot will pass through every feature point along the way. As each of these points is reached, the robot's position is updated and the next point is targeted. A map of all checkpoints is displayed in Figure 4.1d, and the next point is chosen based on the state diagram in Figure 4.2.



(a) (b) (c) (d)

Figure 4.1: CSL basement maps of (a) destinations, (b) feature points, and additional navigation aids (c). A combined map of all checkpoints can be seen in (d).

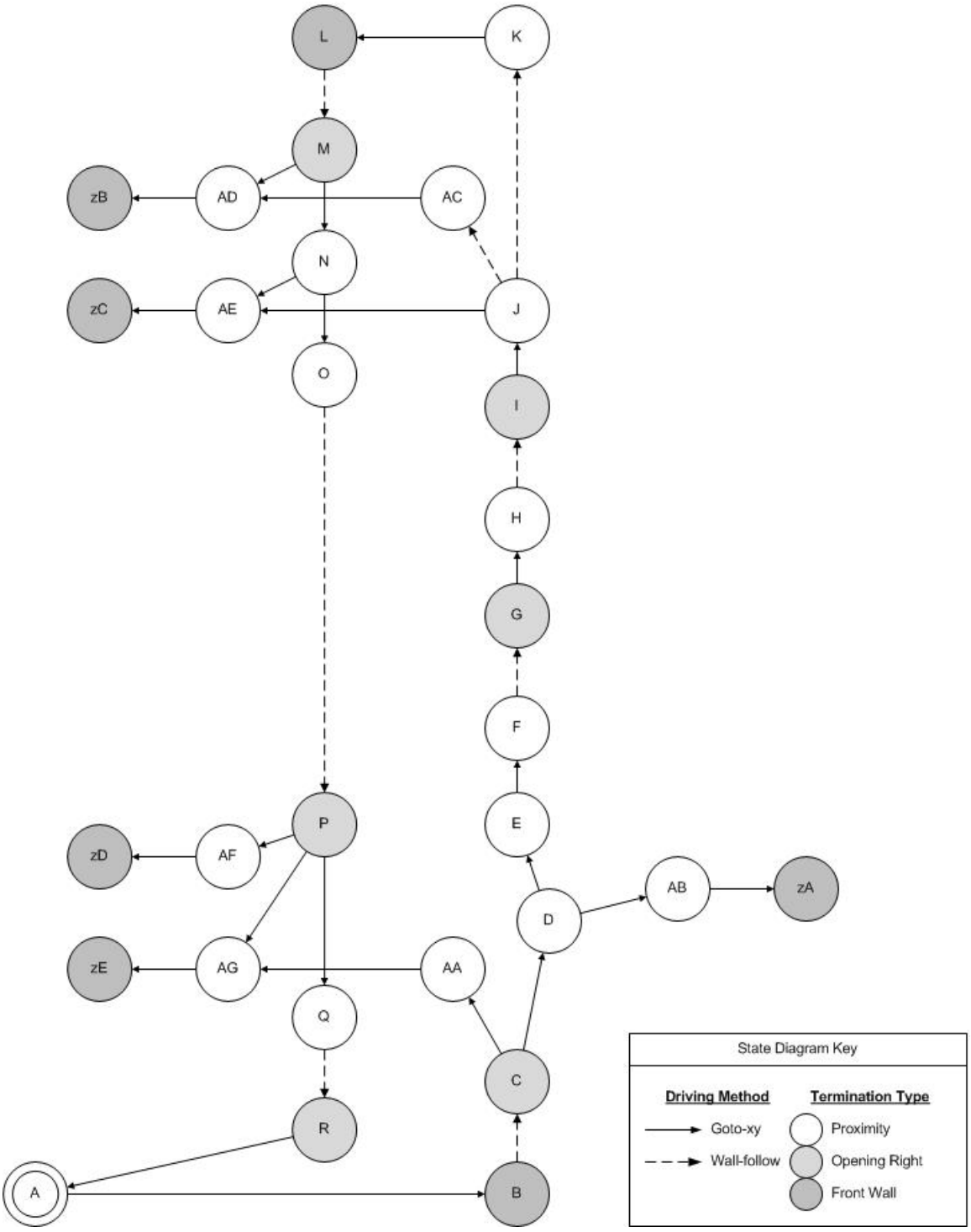


Figure 4.2: State machine for choosing the next checkpoint along the path from home to destination. It also chooses the driving method and termination type

4.2 Set Ideal Reference Velocity and Turn Rate

Once a checkpoint has been chosen, the next step is moving to it. First, the robot must be oriented to face the correct direction. This prevents problems when the robot starts facing a wall, but plans on turning 90 degrees and following it. Turning first and then moving prevents the obstacle detection from triggering on the wall in front of it.

Once it is facing the correct direction, the robot has two movement options: drive directly toward the goal using goto-xy mode, or use the right-facing sensors to implement wall following. In the wall-following case, though, a few changes must be made to what was introduced in Section 3.4.

As the next section will discuss, openings on the right are detected using the difference between the two right-facing sensors. However, if the doorway is not a perfect right angle, the current navigation scheme will partially compensate for the opening before the maximum gap can be detected. The sensors never see the large discrepancy needed to recognize a gradual or shallow opening, and so the checkpoint is missed. To fix this, Equation (3.12) becomes Equation (4.1) when the robot is near a feature point. The robot also reduces its reference velocity as it gets near the destination, which prevents it from overshooting and updating at the wrong location.

$$\text{Turn} = 0.10\left(11 - \text{IR}_{RR}\right) \tag{4.1}$$

Finally, the wall-following algorithm must be prevented from driving the robot in circles. If it tries to wall-follow when not adjacent to a wall, the robot will constantly turn right until a wall is found. To prevent this from leading to circles, if the robot

attempts to drive in the cardinal direction opposite of the current checkpoint, the turn value should be set to zero. This results in the robot driving toward the wall until it is encountered, at which point the wall will direct the robot in the correct direction.

The choice of which driving method to choose depends on whether or not a wall exists along the entire path from the previous to next checkpoint. This is decided using the state machine previously displayed in Figure 4.2 (Section 4.1).

4.3 Arriving at Checkpoints

Now that the Deliverybot is driving in the right direction, the next step is deciding when it reaches the checkpoint. There are three different ways, called termination methods, in which a checkpoint can be reached. The method used depends upon why the target was chosen as a checkpoint in the first place, and is set using the state machine in Figure 4.2.

4.3.1 Front Wall termination

The first termination occurs when the robot meets a wall in front of it. This will occur whenever the hallway makes a left-hand turn or when the destination is reached. In testing, the latter meant stopping at a closed doorway that might normally be a cubicle entrance, but in practice, this could be a delivery bin at the entrance to a cubicle.

One of the dangers with this type of termination is that almost anything can look like it. If someone puts a box down in the robot's way while it is looking for a front wall, then the robot may mistake the obstacle for its goal. It would update its position to the wrong coordinates and possibly become lost.

To fix this, at least partially, the robot should begin looking for a front wall termination only when it is close to it. The robot probably has a small amount of error in its dead reckoning, but it is unlikely to be wrong by more than a foot or so. Therefore, any object sensed to the front can be assumed an obstacle until the robot gets close enough to its destination. For the Deliverybot, "close enough" was about 2 feet. At that point, the robot slows down and declares the next object it encounters to be the wall.

Because feature points were set to be about 1 foot away from the walls, a front termination point should be found when one of the front sensors reads less than 8 inches. Once this happens, the robot should update its position and head to the next checkpoint. Figure 4.3 shows two examples of Front Wall termination.

4.3.2 Opening Right termination

The second type of termination comes about when the robot senses an opening to its right. As illustrated in Figure 4.4, this occurs when it is following a wall and comes upon a doorway, or when the hallway turns right. The front right-facing sensor reads a large value at the same time that the back right-facing sensor still sees the wall. This abrupt change between the two sensors is the termination state being looked for.

In the CSL basement, there were not many openings because most of the doors remained closed at all times. However, even closed doors are noticeable if they are not flush to the wall. In CSL, the closed doors were still almost half a foot farther away than the doorframe. Therefore, the Deliverybot declared the checkpoint reached when the front right sensor read a distance 6 inches or greater than the back right sensor. As with front wall termination, the robot will look for an opening only when within about 2 feet.

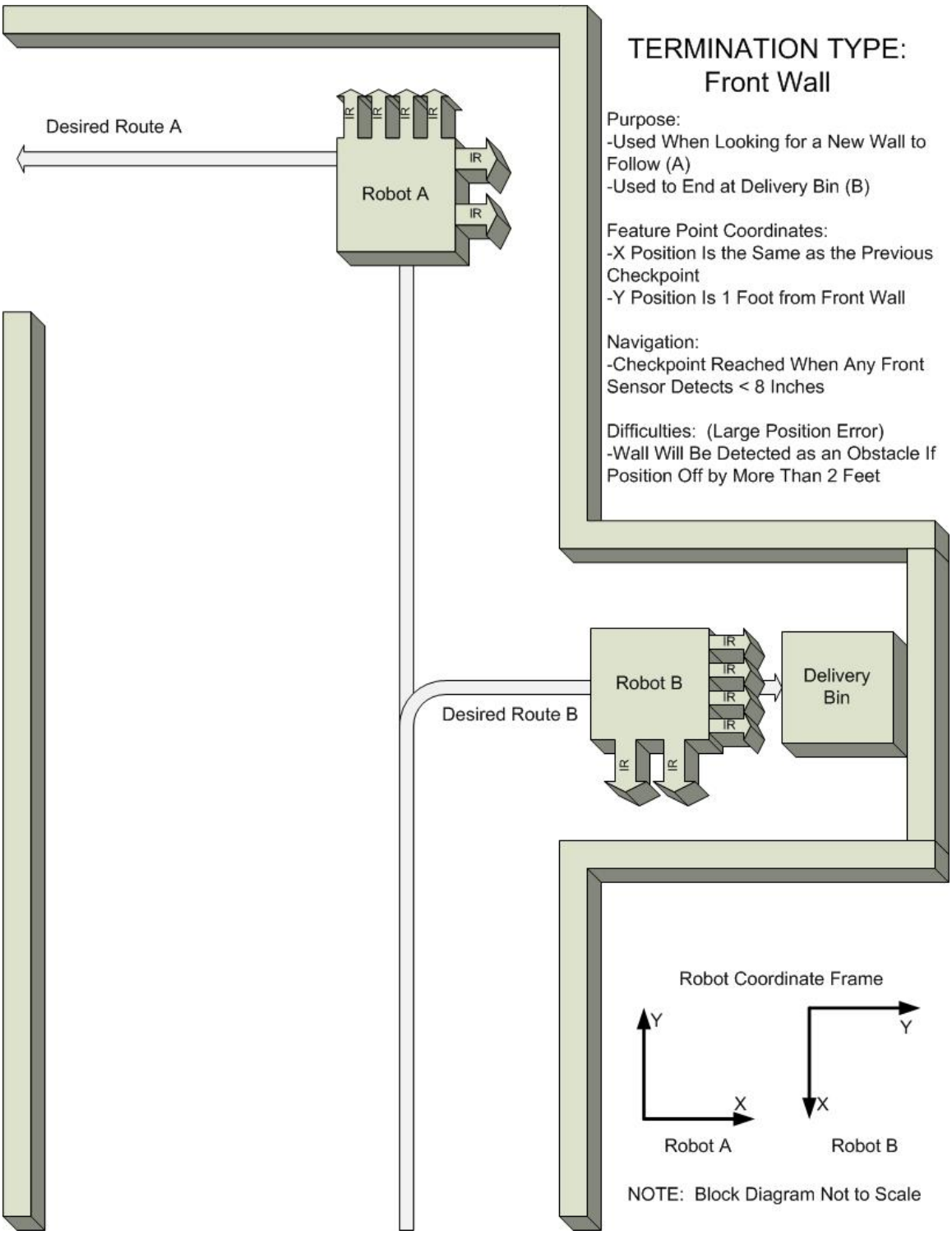


Figure 4.3: Summary of Front Wall termination.

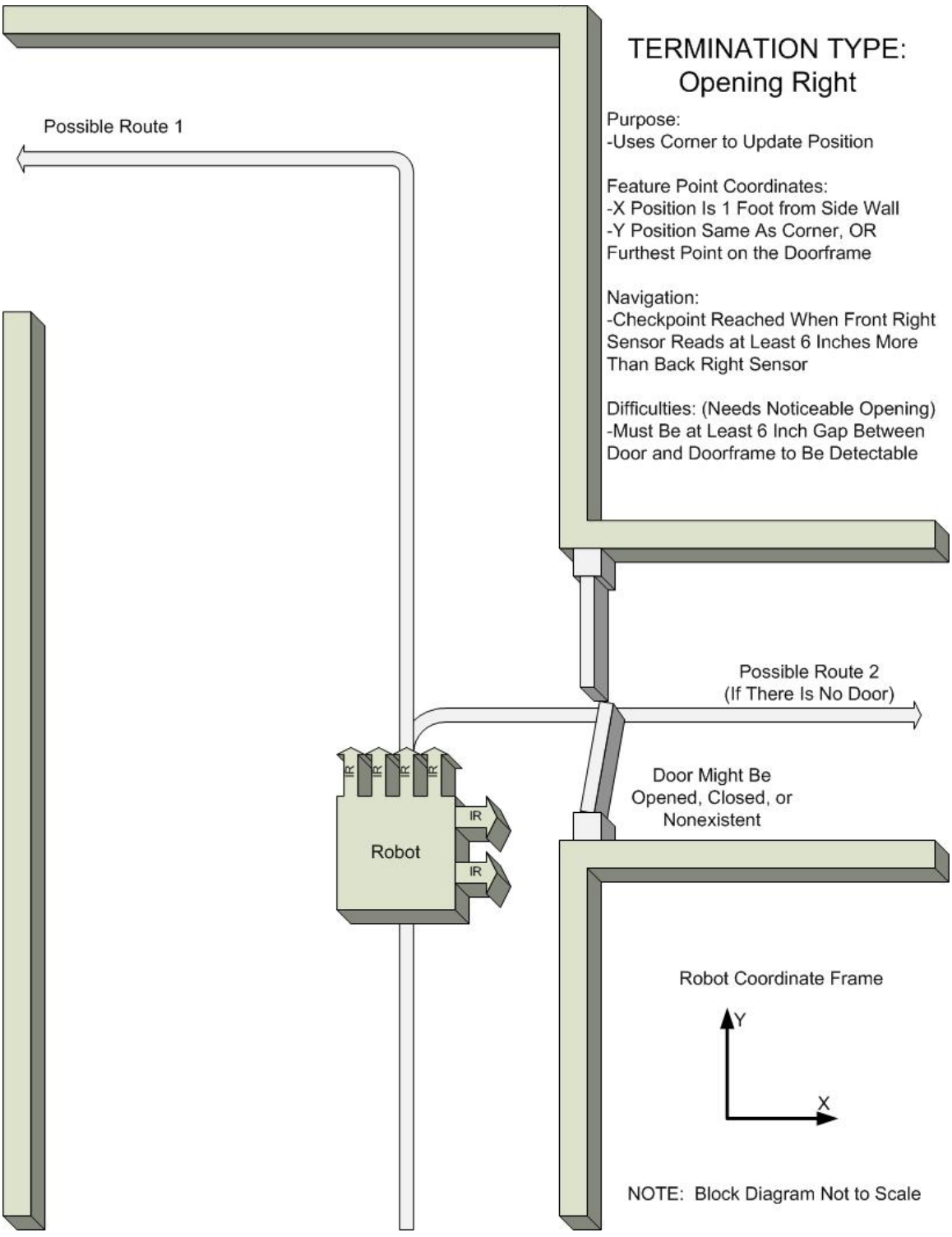


Figure 4.4: Summary of Opening Right termination.

4.3.3 Proximity termination

The third and final termination state is based upon proximity. If the robot thinks it is near the checkpoint, then it assumes it is and moves on. The reason for this termination is that feature points do not exist at every desired location. Sometimes the robot must stop in the middle of a hallway and cross over to the other side, as illustrated in Figure 4.5. These navigation points were introduced in Section 4.1 and shown in Figure 4.1c. They have nothing distinguishable about them that can be sensed, so the robot must estimate when they are reached.

Another instance of proximity termination can be seen in Figure 4.6. In this example, the robot must cross a gap without the aid of a wall to follow. Because of the high occurrence of angular error when leaving from the previous checkpoint, it was decided not to attempt terminating at the adjacent corner. The distances were found to vary widely, especially if an obstacle was encountered along the way. Instead, the robot terminates using the proximity estimate and then slightly updates its position and angle based upon the right sensors' values. For instance, if the sensors read higher than expected, then the angle variable should be decreased slightly and the position set to be slightly farther from the wall.

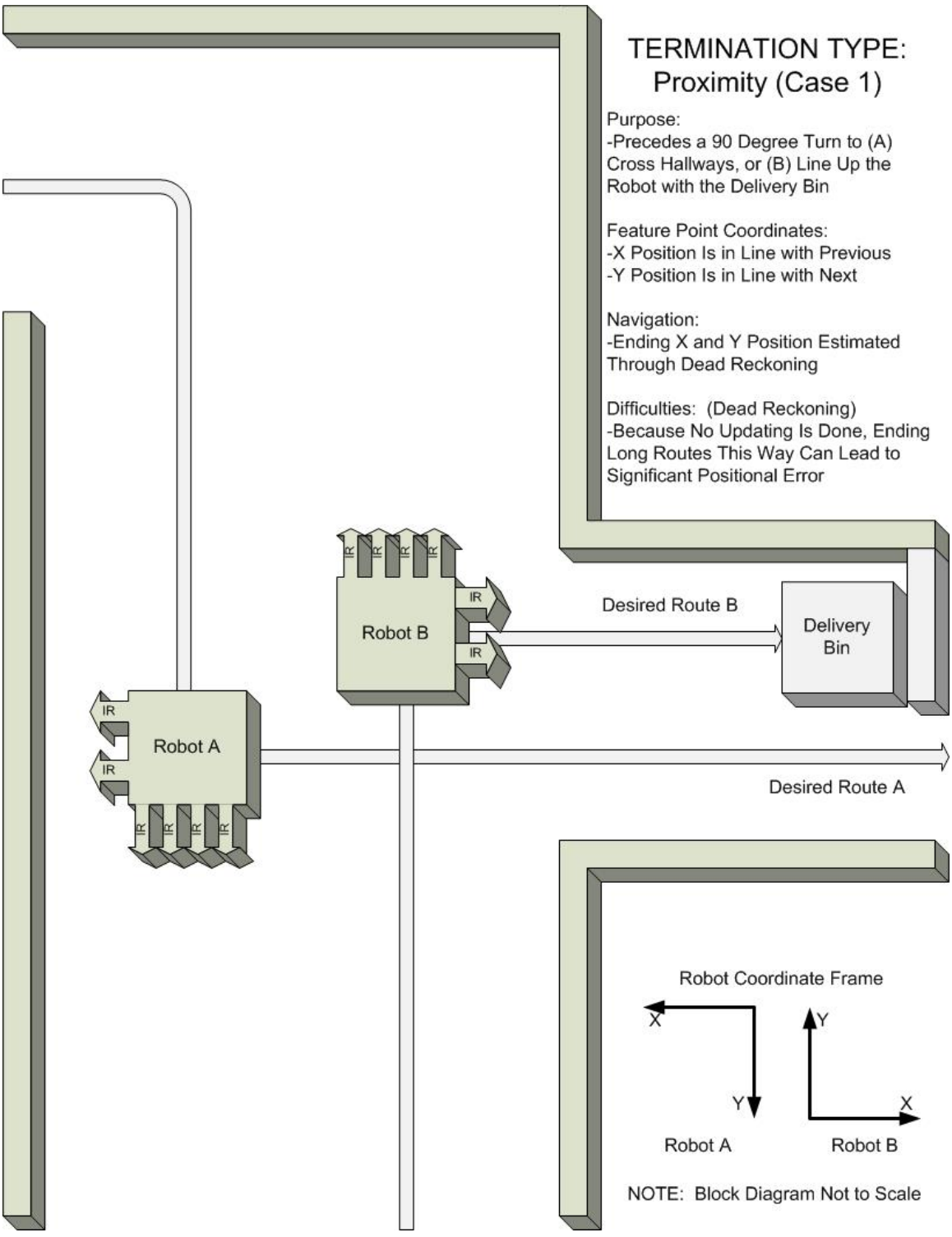


Figure 4.5: Summary of the Proximity termination state used for making 90-degree turns.

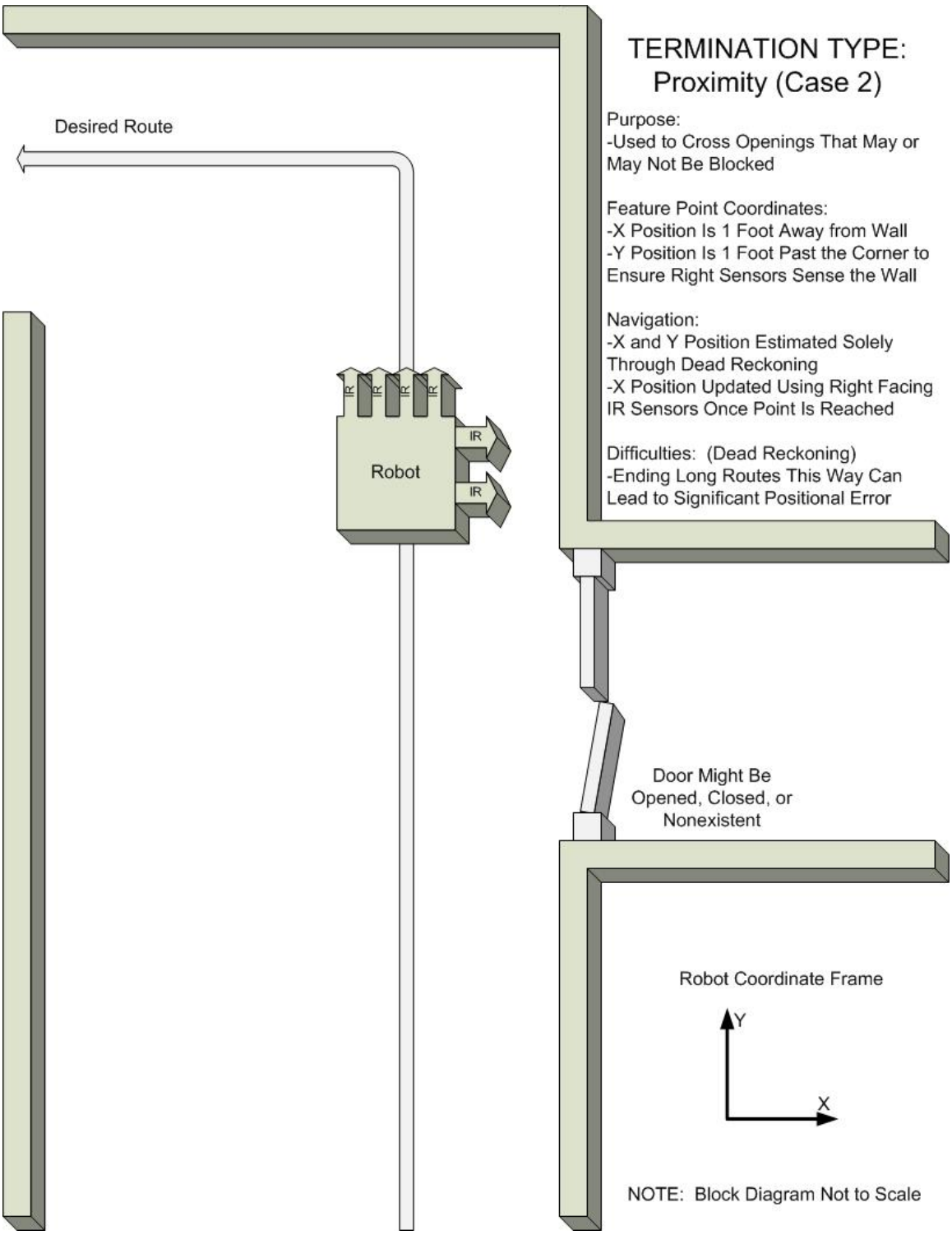


Figure 4.6: Summary of the Proximity termination state used for crossing openings.

4.3.4 Bypassing checkpoints

Unexpected obstacles can appear anywhere. Unfortunately, that includes on top of checkpoints, and dealing with this special case is crucial to successful navigation. For the Deliverybot, this was dealt with by bypassing the unreachable checkpoint.

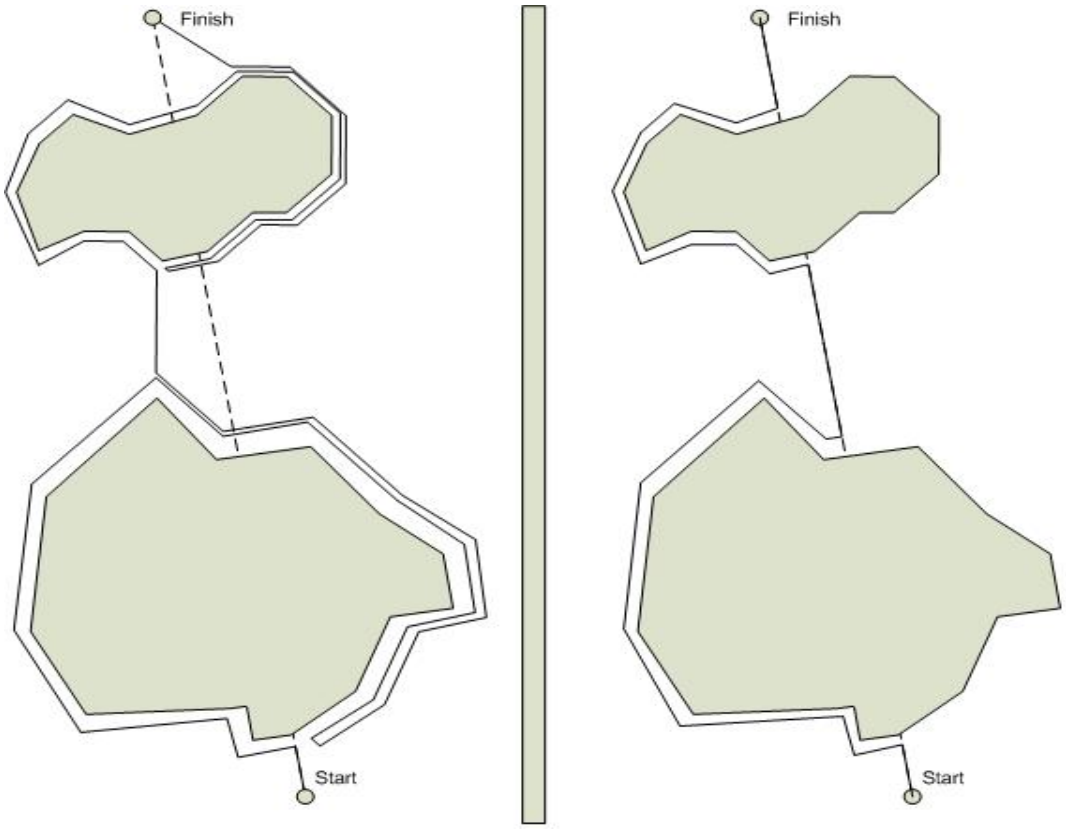
The purpose of checkpoints is to help reach the destination. Missing one becomes irrelevant once the next one is successfully reached. Therefore, a checkpoint should be skipped as soon as reaching it becomes more trouble than help. For the delivery robot, it was decided that once the robot has driven 5 feet farther than it expects the checkpoint to be, it has probably missed it altogether. Once that happens, the point is skipped and the next checkpoint is set as the destination.

4.4 Avoiding Obstacles

The final thing to be done is verify that moving will not result in a collision with an unexpected object. If it will, then a method of getting around the obstacle will be needed. Two common avoidance algorithms are known as Bug1 and Bug2 [7].

The first approach, shown in Figure 4.7a, repeats two steps until the destination is reached. First, drive toward the destination until an obstacle is encountered. Second, circumnavigate the obstacle and return to the point closest to the goal. While this might be useful for a mapping robot, unnecessary turning can cause significant error in the Deliverybot.

The Bug2 algorithm says to circumvent the obstacle until the original path is intersected, at which point the normal path should be followed. This is much better for the Deliverybot, as it requires less turning than Bug1. It is pictured in Figure 4.7b.



(a) (b)

Figure 4.7: Two common avoidance algorithms: (a) Bug1, (b) Bug2.

However, a still simpler algorithm exists that requires even less turning: circumvent an obstacle until the robot has an unhindered path to the target, at which point normal operation resumes. Because there are two navigation methods, this avoidance method has two different patterns. They are shown in Figure 4.8.

To implement this algorithm, a state machine was created. It has six states: *None*, *Approaching*, *Verify 1*, *Clear Obstacle*, *Go Around*, and *Verify 2*. A summary of the states, transitions, and effects can be found in Figure 4.9.

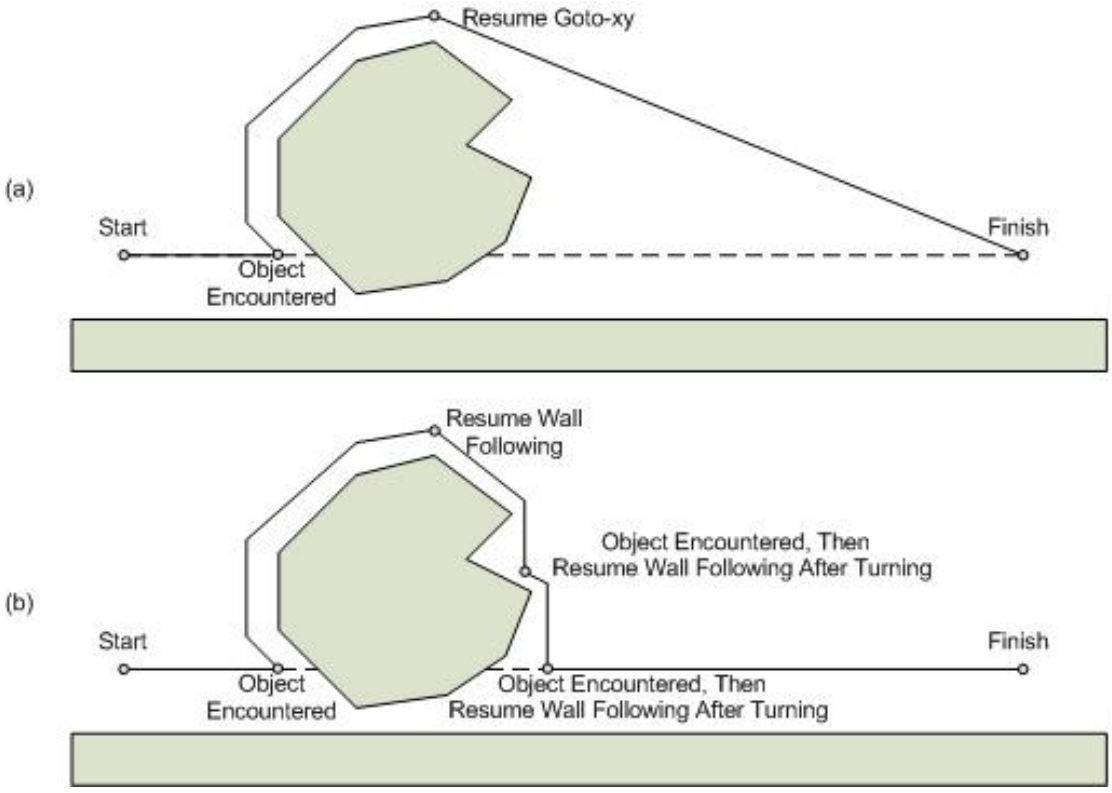


Figure 4.8: The avoidance algorithm used by the Deliverybot when using (a) goto-xy and (b) wall following. Once a clear path exists to the destination, normal operation resumes. Note that while wall following, the robot is not allowed a velocity component in the cardinal direction opposite the destination.

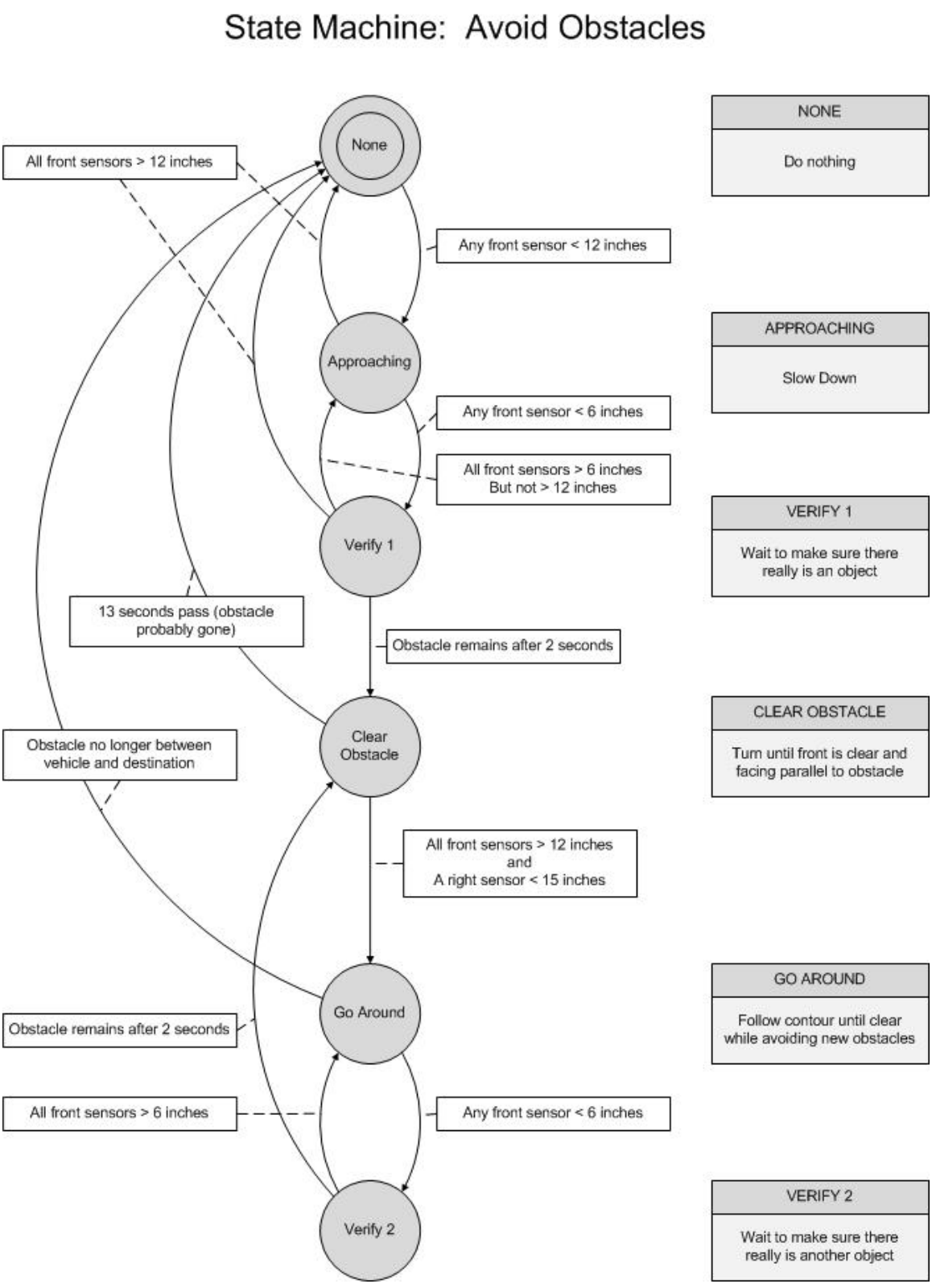


Figure 4.9: Summary of the state machine used for obstacle avoidance.

4.4.1 Obstacle state: None

As its name might imply, being in this state means that there are no obstacles detected. Therefore, it allows the ideal velocity and turn values to be used. This is the desired state for the robot, as it means that everything is working as expected.

This is also the watchdog state. It monitors the front four IR sensors, waiting for one of them to sense something one foot away or less. When this happens, the state changes to Approaching. Because obstacles can be of many shapes and sizes, this happens if any of the front IR sensors sees something. However, there are two exceptions.

The first occurs when the robot is looking for a front wall to change states. This requires the termination state be Front Wall (Section 4.3.1) and that the robot be "close enough" (within 2 feet) to the estimated position. In this case, detecting something in front of the robot is desired, so obstacle avoidance should not kick in.

The second exception occurs when the robot has no forward velocity. This case was added to prevent an obstacle from being detected as the state machine exits a state with front wall termination. This was partially fixed in Section 4.2, where the robot orients itself in the correct direction before moving to the next checkpoint. This addition completely fixes the issue. By the time the robot is moving forward again, the wall should no longer lie in front of it, and obstacle avoidance should never have been triggered.

4.4.2 Obstacle state: Approaching

This state is entered once any of the front sensors detects an object, yet a collision is not yet imminent. It is the equivalent of being on alert. The practical response to this is moving more cautiously, and this was done by reducing the forward velocity to half a tile per second.

One way to leave this state is if the object is no longer detected. It may have moved out of the way, or even been a false positive. When this occurs, all four of the IR sensors should read more than 12 inches again, and the state changes back to None.

The other option is that the obstacle comes closer. If there really is an object, and it is stationary, this should be the result of driving toward it. Once any of the sensors reads less than 6 inches, a collision is imminent and the state machine should move on to the next state, Verify 1.

4.4.3 Obstacle state: Verify 1

The robot keeps track of its position primarily through dead reckoning, and the easiest way to confound this estimation is to turn a lot. Turning can result in wheels slipping and even small angular errors can quickly lead to a large positional error. To mitigate this problem, the robot should try to go around an object only if it must. The Verify 1 state exists to make sure the obstacle will not be moving out of the way on its own.

To do this, the velocity and turn values are both set to zero while the robot waits for 2 seconds. At that point, if an object is still detected within 6 inches, then it will probably not be moving, and the state machine changes to Clear Obstacle. However, if at

any time all of the sensors read more than 6 inches, then it would appear that the object is moving away. In this case, the state regresses back to Approaching or, if they all read more than 12 inches, to None.

4.4.4 Obstacle state: Clear Obstacle

If this state is reached, there is definitely an obstacle and it will not be moving out of the way. The only solution is to go around it. The first step is turning until the obstacle no longer blocks the way. This is done by setting the velocity to zero and the turn value to 0.5. The robot will slowly turn in place until the state is changed.

The state change occurs when two conditions are met. First, all of the front sensors must read more than 12 inches. This makes sure that the robot has cleared the obstacle and can safely drive forward again. The second condition requires that one of the right-facing sensors reads less than 15 inches. This makes sure that the robot has something to circle around when it moves to the next state. When these two conditions are met, the state machine progresses to the state Go Around.

However, a problem occurs if the object moves after this state is reached. The robot can find itself circling indefinitely looking for an object that is no longer there. To deal with this condition, the state uses a counter. If the robot spins for 13 seconds (about one revolution) without meeting the above conditions, it can be assumed that the obstacle is no longer in the way. The state is returned to None, and normal operation resumes.

4.4.5 Obstacle state: Go Around

Once the robot has cleared the obstacle, the situation becomes very similar to right wall following. There is something to the right and the robot wants to follow its contour. However, because the shape is unknown, a lower velocity (half a tile per second) should be observed to let the sensors respond to anything unexpected.

In addition, following at a set distance is no longer very important. All that matters is roughly following the contour of the object until it has been circumvented. To that end, the turn value should be the difference between the two right-facing sensors. If the front sensor reads higher than the rear sensor, the robot should turn right, and vice versa.

However, this can lead to the robot getting too close to the obstacle and even crashing into it. To fix this, a reading from either of the sensors of 6 inches or less will cause the robot to turn slightly to the left, away from the obstacle. Getting too far away from it is not desirable either, so if both sensors read more than 15 inches, the robot should turn slightly to the right.

This state is left once the robot has a clear path to the destination. This is the case when the front sensors do not detect anything and a call to the goto-xy code attempts to turn the robot left. Once this happens, normal navigation may resume and the state is changed to None. This was shown previously in Figure 4.8.

A second way to leave the state is to encounter another obstacle. Once the robot has cleared the first obstacle and begun driving around it, a second obstacle might be found. If any front IR sensor reads 6 inches or less while the first obstacle is being circled, the state should change to Verify 2.

4.4.6 Obstacle state: Verify 2

The Verify 2 state is nearly identical to Verify 1. The only difference is what happens if the object moves out of the way. In Verify 1, no obstacle had yet been certianly found. If whatever triggered the sensors suddenly disappeared, then the robot could resume normal operation. In Verify 2, though, the robot has been interrupted from moving around an obstacle. Therefore, if all front IR sensors read more than 6 inches, the state should revert to Go Around instead of Approaching or None. Everything else about this state is the same as for Verify 1.

4.5 Updating Position

The Deliverybot tracks its position by using the encoders. Unfortunately, dead reckoning is often inaccurate. Poor variable calibration, wheel slippage, and friction can all result in a false position estimate. Fortunately, the Deliverybot knows some information about its environment, and it can use its IR sensors to update its position at key feature points.

4.5.1 Updating along walls

An office building is often filled with long hallways. They are nearly always straight and are usually parallel or perpendicular to every other hallway in the building. If the robot's axes are chosen properly, following a hallway should result in only one axis changing at a

time. If the robot finds this not to be true, then its angle is likely incorrect and can be updated.

For example, say that the positive *y*-axis faces directly north (0 degrees) and that the robot is wall-following in a north-south hallway heading north. If the robot drives in a straight line for several seconds (while wall-following), then it can safely be assumed that the robot has a heading of 0 degrees, and the angle can be updated to reflect this. Encountering obstacles will force the robot to slow down and turn, so updating should only occur when driving straight, without swerving, and moving at full speed.

In addition to updating the angle, the robot's position can also be corrected. Any hallway that runs solely along one axis means that the robot's position along the opposite axis should remain constant when wall-following. While the robot is driving in the *y*-direction, the *x*-position can be set to one foot from the wall, and vice versa.

Finally, if the angle is known, and one of the axes is known, then the other axis can be calculated as well. Provided no obstacles have been encountered, then the distance driven since the last checkpoint (call this *d*) will have been in a straight line. Therefore, driving north means that the *y*-position can be updated to be the former checkpoint's *y*-position plus *d*.

4.5.2 Updating at feature points

In addition to updating along walls, the robot's position can also be corrected at feature points. In Section 4.3, different methods of path termination were discussed. While proximity termination is based solely on the estimated position, the other methods require input from the IR sensors.

In the Opening Right case, the robot does not move to the next checkpoint until an opening on the right side is found. This termination type occurs when the robot is adjacent to the opening while wall-following. This means the position along both axes is known, and the estimated position can be accurately set to the actual position.

For Front Wall termination, the position can be known only in the direction perpendicular to the wall. Therefore, only one variable is updated when this termination state is reached.

CHAPTER 5: EXPERIMENTAL RESULTS

Early on in the project, the Deliverybot attempted to navigate solely by dead reckoning. One look at Figure 5.1a shows clearly that some form of updating would be necessary. The first step was updating along straight walls, and this improved the overall performance immensely (Figure 5.1b). However, this early version had several problems.

First, it failed to account for normally closed doors being opened, such as those for storage and maintenance. Instead, it continued to wall-follow when it reached points such as S and T. One day a door was opened, and the map needed to be redone.

Related to this was a lack of feature points at which to update. The robot looked only for obvious corners such as I and L. This meant that no more updating could be done north of point E. Placing several obstacles along the wall threw off the position so much that the robot could no longer find its target.

When the feature map was redone, several changes were made. First, any potential opening was marked, and goto-xy code was used to cross it. This prevented the robot from entering doorways that it expected to be closed. Second, every depression of more than 6 inches was used to update the position. Third, additional points were added so that the robot made only 90-degree turns, if for no other reason than consistency. A successful run on this new map can be seen in Figure 5.1c.

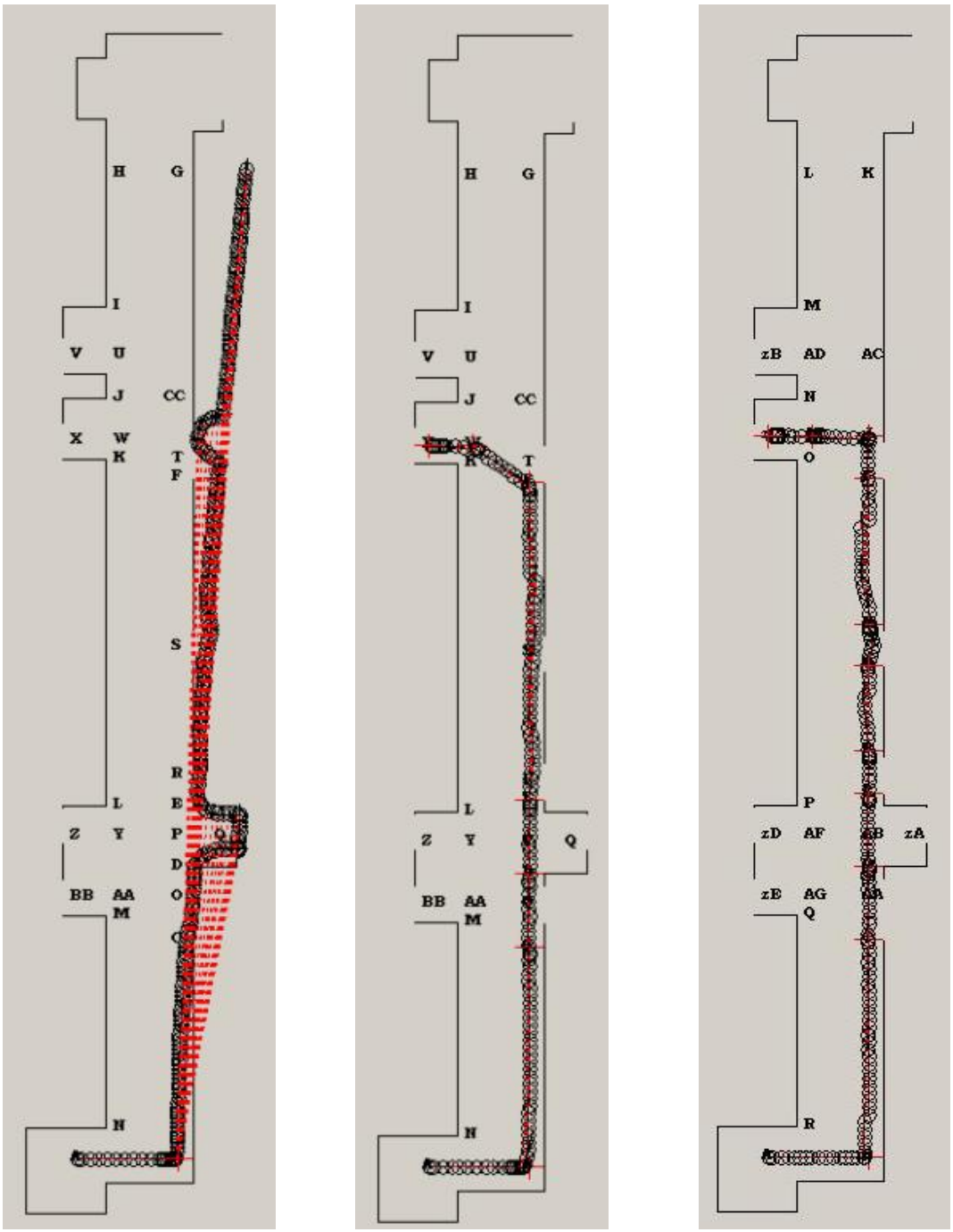


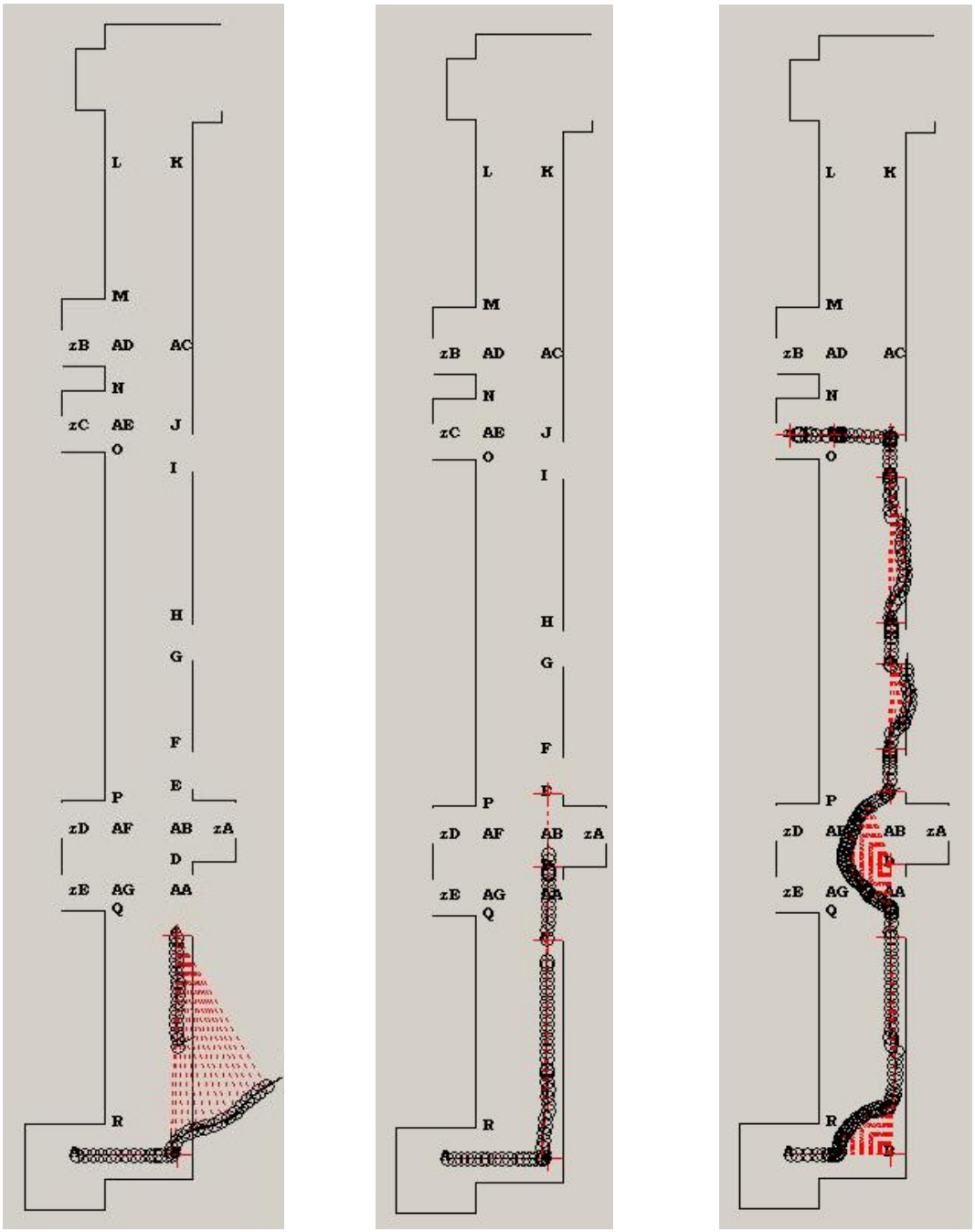
Figure 5.1: (a) This screenshot illustrates drift in the dead reckoning. (b) Updating the position and angle along the walls improved performance dramatically. (c) The final version added additional feature points, and was setup to only use 90-degree turns. This figure shows a successful obstacle-free run using the final version.

To demonstrate the robustness of the Deliverybot, it was presented with obstacles of varying shapes and sizes. It was even picked up and moved slightly to show that small angle and positional error was recoverable. In nearly all of these situations, the robot succeeded in finding the next checkpoint. Below are some specific examples of this.

In the run shown in Figure 5.2a, the Deliverybot was interrupted after reaching the corner checkpoint. With its wheels off the ground, it was rotated 45 degrees to the left before being set down and turned back on. After some initial oscillation, the robot found itself following a straight wall for at least 2 seconds. This led to the angle, *x*-position, and *y*-position to be updated very close to the correct values. It is known to be a successful update because the robot was shut off when it detected the next checkpoint. As can be seen, the northernmost circle was just barely past the checkpoint, which is marked by the red plus sign.

Similar to the previous case, Figure 5.2b demonstrates the Deliverybot's tolerance of positional error. About midway down the wall, where the circle appears darkest, the motors were shut off. The Deliverybot was then picked up and moved forward nearly 2 feet. This caused the robot to reach the corner about 2 feet sooner than anticipated, but as the jump in position indicates, it updated successfully.

A third run, shown in Figure 5.2c, demonstrates what happens when a checkpoint cannot be reached. Twice a checkpoint was surrounded by obstacles, and in both instances, the Deliverybot succeeded in bypassing them. The large amount of driving in the second instance resulted in a minor angular error, but this was corrected as soon as a long, straight section of wall was found.

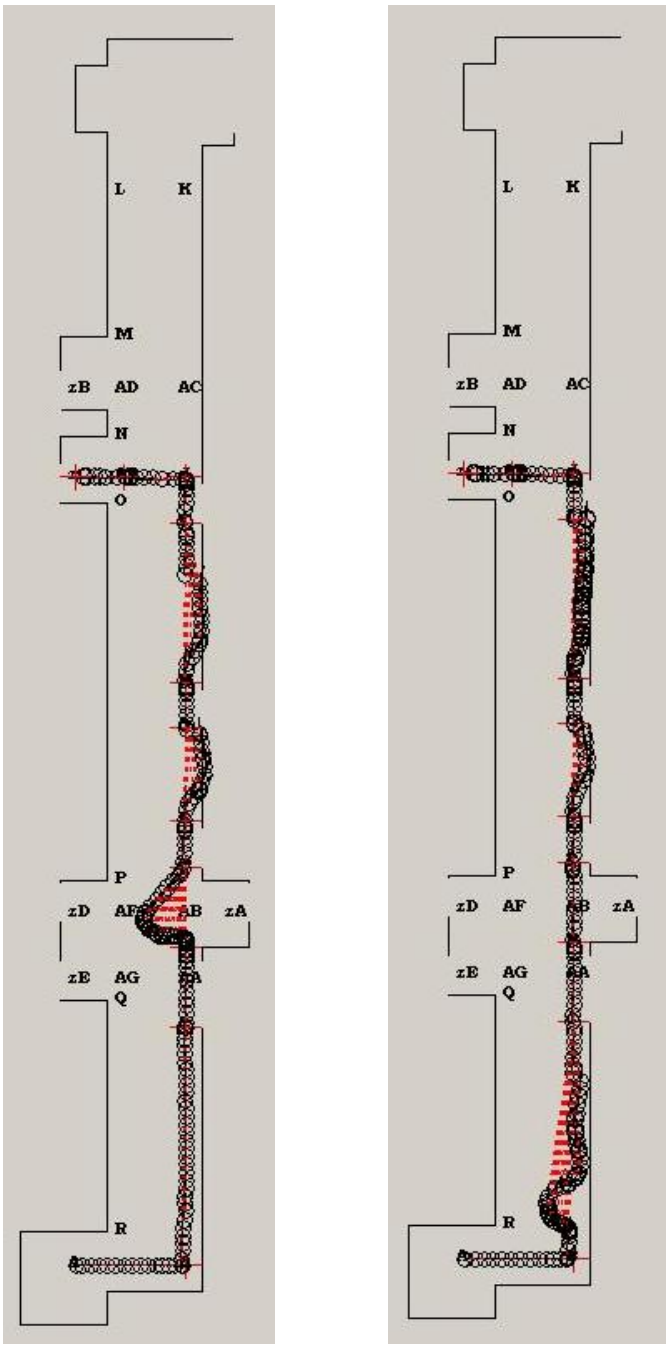


(a) (b) (c)

Figure 5.2: (a) This run demonstrates excellent angular error tolerance when wall-following. (b) Similarly, this run shows tolerance to positional error. (c) In this case, the robot succeeded in bypassing two different checkpoints that were deemed unreachable, and it ended the run in the correct location.

Figure 5.3 illustrates the different ways in which the Deliverybot circumvents obstacles. When in goto-xy mode, the robot drives straight toward the next checkpoint after clearing the object, which is shown in Figure 5.3a. When wall-following, as in Figure 5.3b, the robot returns to the wall before continuing on to the next checkpoint.

Figure 5.3b also shows what happens when the Deliverybot finds itself behind a slow-moving obstacle. North of the doorway halfway down the hall, a person began walking very slowly in the robot's path. The detected obstacle (the person's shoe) did not maintain a set distance for more than two seconds, so the robot never tried to go around it. Instead, the robot followed the person very slowly at a distance of about half a foot. This is apparent because the circles are darker in this region, indicating the robot spent more time there. Because it was not moving at full speed, the robot was also prevented from updating its position and angle. This can be seen at the next checkpoint, where updating the position moved the estimated position about a foot to the left. At this point, the person moved out of the way and the robot successfully found the target.



(a) (b)

Figure 5.3: (a) Behavior in goto-xy mode after an object is cleared. (b) Wall following causes the robot to return to the wall before continuing. This run also shows the result of following behind a slow-moving object.

CHAPTER 6: CONCLUSIONS AND FUTURE WORK

This thesis has analyzed an implementation of an office delivery robot. It briefly discussed the hardware that could be used, such as IR sensors and encoders. It also investigated the need for a user interface, explained a wireless communication scheme, proposed a robot controller, and described two methods of locomotion. Last, it considered feature point representation, path planning, and obstacle avoidance. As demonstrated in the results section, the proposed Deliverybot proved robust to a variety of difficulties that might arise, from angular and positional error to obstacles in inconvenient locations.

However, the current Deliverybot is not perfect. Forcing it to move around and turn excessively without reaching a checkpoint can still cause the robot to become lost (Figure 6.1). Generating this error, though, required active participation by a person, rather than a stationary obstacle. In an office, if people went through this much trouble to confuse the robot, they could just as easily pick it up and move it somewhere else. Despite the unlikelihood of the problem, though, it demonstrates that the robot *can* become lost. Additional work needs to be done to solve this problem.

One proposed method is to declare the robot lost whenever it finds itself in an unreachable position (such as inside a wall). In this lost state, the robot can drive around looking for a feature point, such as a corner. Once a point is found, the robot can place an image of itself at every potential point that matches the one found, and drive around until all but one of these images becomes illegal. The remaining image is assumed to be at the correct location, and the robot resumes normal operation.

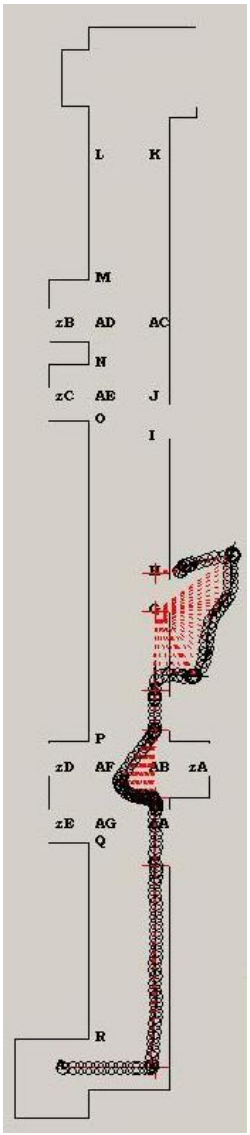


Figure 6.1: This failed run demonstrates that trying hard enough can confuse the robot enough to render it lost. Additional work must be done to deal with this.

Unfortunately, this approach is not as simple as it sounds. It was attempted as part of this thesis, but several problems prevented it from ever succeeding. First, for the robot to get confused enough to become lost, there must be an unusual obstacle present. In practice, this same obstacle could trigger false positives on corner detection. This meant that all of the images being placed were in the wrong locations, and it became lost again immediately after "fixing" itself. Secondly, while driving around eliminating

images, the robot is unable to update its position. By the time only one image remained, even if it was initially the correct one, enough drift had been introduced to render it useless.

Despite this inability to recover from extreme errors, the Deliverybot has proven to be a reliable navigator in typical hallway environments. With the addition of recovery code, a load-carrying platform, and better batteries, this robot is ready to deliver. Autonomous vehicles driving around a populated environment are no longer restricted to movies. They are doable today and could be a cost-effective alternative to delivery by hand.

REFERENCES

[1] GE423 Mechatronics Laboratory at the University of Illinois, "Segbot: Introduction to Mechatronics, University of Illinois at Urbana-Champaign. Introduction," Spring 2004. [Online]. Available: http://coecsl.ece.uiuc.edu/ge423/spring04/group9/index.htm.

[2] Philips Semiconductors, "The I^2C-Bus Specification. Version 2.1," January 2000. [Online]. Available: http://coecsl.ece.uiuc.edu/ge423/datasheets/i2c_spec.pdf.

[3] HVW Technologies, "I2C-It IR Rangefinder." [Online]. Available: http://www.hvwtech.com/products_view.asp?ProductID=665.

[4] GE423 Mechatronics Laboratory at the University of Illinois, "Pointers and Passing Parameters to Functions." [Online]. Available: http://coecsl.ece.uiuc.edu/ge423/lecturenotes/encoders_pwm.pdf.

[5] G. Franklin, J. Powell, and A. Emami-Naeini, *Feedback Control of Dynamic Systems*. Upper Saddle River, NJ: Pearson Prentice Hall, 2006.

[6] M. Spong, S. Hutchinson, and M. Vidyasagar, *Robot Modeling and Control*. Hoboken, NJ: John Wiley & Sons, Inc., 2006.

[7] S. M. LaValle, *Planning Algorithms*. Cambridge, UK: Cambridge University Press, 2006.

APPENDIX A: DELIVERYBOT CODE

Included in this appendix is the code written for the Deliverybot. It uses the core code from the University of Illinois Mechatronics class (GE 423). All relevant additions and modifications to this core code can be found below.

File #1: user_Main.c

```
// STANDARD ANSI INCLUDES
#include <std.h> // DSP/BIOS standard include file
#include <hwi.h>
#include <swi.h>
#include <log.h> // LOG_printf calls
#include <mem.h> // MEM_alloc calls
#include <que.h> // QUE functions
#include <sem.h> // Semaphore functions
#include <sys.h>
#include <tsk.h> // TASK functions
#include <rtdx.h> // RTDX functions
#include <math.h> // sinf,cosf,fabsf, etc.
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

// DSP INCLUDES
#include <c6x.h>         // register defines, in c:\ti\c6000\cgtools\include
#include <c6x11dsk.h>    // TI header in the directory c:\ti\c6000\dsk6x11\include
#include <fastrts67x.h>  // TI's real-time math library, in c:\ti\c6700\mthlib\include

// COECSL INCLUDES       // all COECSL functions are usually in the directory f:\C6713DSK\include
#include <c6xdskdigio.h> // COECSL functions for daughter card, encs, PWM
#include <max3100uart.h> // COECSL functions for communication to max serial chip
#include <dac7724.h>     // COECSL functions for the DAC7724 chip
#include <ad7864.h>      // COECSL functions for the AD7864 chip
#include <RCservo.h>     // COECSL functions to set up PWM ch3 and ch4 to drive RC servos
#include <switch_led.h>  // COECSL functions for turning off LEDs, monitoring switches
#include <dspvisioncolor50Hz_cLCD.h>
#include <i2c.h>
#include <edma.h>
#include <sharpir.h>
#include <dsk6713.h>
#include <user_ColorVisionFuncs.h>
#include <user_UARTFuncs.h>
```

```
#include <user_PIFuncs.h>
#include <user_IR_UltraFuncs.h>
#include <atmel_pwrboard2.h>
#include <color_LCD.h>
#include <xy.h>
#include <statemachine.h>


// INITIALIZING and EXTERNING VARIABLES
int timeint = 0;                        // Timer, counts up every millisecond
int ledstate= 0;                        // Reads the 4 switches in as binary

// IRs controlled by DSP's GPIO
extern int ir1,ir2,ir3,ir4,ir5,ir6,ir7,ir8; // Read by the functions in "user_IR_UltraFuncs.c"
char IR1_s[2] = "00";                   //      IR values
char IR2_s[2] = "00";                   //      converted
char IR3_s[2] = "00";                   //      into strings
char IR4_s[2] = "00";                   //      so that the
//char IR5_s[2] = "00";                 //      displayed
char IR6_s[2] = "00";                   //      values have
char IR7_s[2] = "00";                   //      a constant
char IR8_s[2] = "00";                   //      two digits
int IR_L =0, IR_F1 =0, IR_F2 =0, IR_F3 =0;  // Sensors arranged based on position instead of
int IR_F4=0, IR_RF =0, IR_RR =0;        //  address, which makes using them easier

extern volatile int new_irdata;         // Flags for the TSK
float Rmotor = 0,Lmotor = 0;            // These are the motor encoders (Enc1 and Enc2)
float Enc3 = 0, Enc4 = 0;               // These track the mouse position on LCD screen
int Enc3int=0,Enc4int=0;                // Convert to integers because correspond to pixels

// Wireless Communication Variables
int packet = 1;                         // Rotate through groups of variables to transmit
char sendbuff[100];                     // Buffer to store a string before sending it
extern far SEM_Obj SEM_UART1MessageReady;   // Semaphore indicating a new message has been fully
received
extern char UART1MessageArray[400];         // Buffer holding newly received messages

// Controller Variables
extern float Kp,Ki,Kp_turn,Vpos,Vneg,Cpos,Cneg;    // Constants
int car_on = 0;                         // Highest level motor control
int manual = 0;                         // Manual mode allows control from VB
float vref=0,turn=0;                    // Control Variables for PI Control
float vref_manual=0,turn_manual=0;      // Control Variables for Manual PI Control
extern float destX, destY;              // X and Y target from State Machine
float destX_manual=0, destY_manual = 0;     // Manual values set from VB
extern int cstate;                      // Current target state
extern int obstacle_state;              // State used to get around obstacles
extern float target_radius,turn_thres;  // Constants used with
extern float target_radius_near;        //      GOTO_XY function
int target_user = -1;                   // Robot gets the target user from VB
int go_home = 0;                        // Varible set from VB, starts the return trip

// Absolute Positioning Variables
extern float Angle, Angle_rad, vx, vy;  // Angle in degrees, then radians, x and y velocities
extern float velocity, Xcurrent, Ycurrent;  // Self-explanatory
```

```
// Temporary and/or Testing Variables
float temp1  = 0.0;                             // NOT IN USE
float temp2  = 0.0;                             // NOT IN USE
float temp3  = 0.0;                             // NOT IN USE
float temp4  = 0.0;                             // NOT IN USE
float temp5  = 0.0;                             // NOT IN USE

int test_LOST = 0;                              // mislabeled, set to 1 if out of bounds

extern int goto_method, termination, obstacle_state, almost_there;

////////////////////////////////////////////////////////////////////////////////////////////
// The Main() function is used primarily for performing initializations.
void main(void) {

   int tmp_row,tmp_col,tmp_ratio,tmp_laser_start;
   Init_C6713DSK();                 // set up DSP to talk to external 8 Meg of SDRAM.
   Init_UART1and2(2);               // configure multi-buffered McBSP for SPI,
                                    //     tell max3100 what baudrate to use.
   Init_Wireless();
   Init_pwm(1);                     // zero chip 1's two PWM channels.
   Init_encoders(1);              // initialize encoders to starting zero position.
   Init_RCservo();                  // setup second PWM chip to command RC Servos.
   Init_encoders(2);              // initialize encoders to starting zero position.
     Init_i2c();
     Init_LCD(120);
     Init_SharpIRs();

     Init_DAC7724();
     ICR = 0x0080;                  // clear pending ints
     IER |= 0x0080;                 // enable int 7

#ifdef USEVISION                    // VISION CODE NOT BEING USED
   tmp_row = 0;
   tmp_col = 0;
   tmp_ratio = 4;                   // Start the camera out initially in X4 compression mode.
                                    //     In this mode row and col do not matter.
   set_image_start(&tmp_row,&tmp_col,&tmp_ratio);
   tmp_laser_start = 41;
     set_laser_start(&tmp_laser_start);
     Init_Memory_for_Vision_Func();
   Init_colorvision();            // uncomment if your app uses the camera.
     LCDcolor_Init_OnlyCanCallinMain();
#endif

   enable_edma();                   // needs to be called to enable interrupt for both the camera
                                    //     and the i2c bus
   set_LEDstate(get_switchstate()); // Set LEDs to Switch State

     SmallSprintf(sendbuff,"0 X X X X X"); // check if it should be on, and what the target is
     WirelessSend(sendbuff,strlen(sendbuff));
}// END main

////////////////////////////////////////////////////////////////////////////////////////////
// This function is called every millisecond to prepare the ADC to be read
void prdStartADC(void) {
```

```
    ADC7864_Start();
}// END prdStartADC

////////////////////////////////////////////////////////////////////////////////////////////////
// This function does all the real work
void ADC_INT7_Func(void) {

    timeint++;

    // ************ Get data from encoders and ADC (mostly unnecessary) ******************
    read_encoders(1,&Rmotor,&Lmotor,g_standard_GearMotor); // *UNNEEDED* if change names
    read_encoders(2,&Enc3,&Enc4,g_standard_GearMotor);        // Read cursor from attached mouse
    Enc3int = (int) (Enc3*(1000/PI));                          // Converts to pixels
    Enc4int = (int) (Enc4*(1000/PI));                          // Converts to pixels
  ledstate = get_switchstate();                              // Gets the switch state

    // ********************** Get data from IR sensors and Camera ************************
    // If new IR data is ready, save it into our variables
    if (new_irdata == 1) {
        if(ir1 >= 30) {     SmallSprintf(IR1_s,"--");  }    else if(ir1 > 9) {   SmallSprintf(IR1_s,"%d",ir1);
    }    else {     SmallSprintf(IR1_s,"0%d",ir1);      }
        if(ir2 >= 30) {     SmallSprintf(IR2_s,"--");  }    else if(ir2 > 9) {   SmallSprintf(IR2_s,"%d",ir2);
    }    else {     SmallSprintf(IR2_s,"0%d",ir2);      }
        if(ir3 >= 30) {     SmallSprintf(IR3_s,"--");  }    else if(ir3 > 9) {   SmallSprintf(IR3_s,"%d",ir3);
    }    else {     SmallSprintf(IR3_s,"0%d",ir3);      }
        if(ir4 >= 30) {     SmallSprintf(IR4_s,"--");  }    else if(ir4 > 9) {   SmallSprintf(IR4_s,"%d",ir4);
    }    else {     SmallSprintf(IR4_s,"0%d",ir4);      }
        // No ir5
        if(ir6 >= 30) {     SmallSprintf(IR6_s,"--");  }    else if(ir6 > 9) {   SmallSprintf(IR6_s,"%d",ir6);
    }    else {     SmallSprintf(IR6_s,"0%d",ir6);      }
        if(ir7 >= 30) {     SmallSprintf(IR7_s,"--");  }    else if(ir7 > 9) {   SmallSprintf(IR7_s,"%d",ir7);
    }    else {     SmallSprintf(IR7_s,"0%d",ir7);      }
        if(ir8 >= 30) {     SmallSprintf(IR8_s,"--");  }    else if(ir8 > 9) {   SmallSprintf(IR8_s,"%d",ir8);
    }    else {     SmallSprintf(IR8_s,"0%d",ir8);      }

        // IR_L    IR_F1       IR_F2       IR_F3       IR_F4       IR_RF       IR_RR
        // IR6     IR7         IR8         IR4         IR3         IR2         IR1
        IR_L = ir6; IR_F1 = ir7; IR_F2 = ir8; IR_F3 = ir4; IR_F4 = ir3; IR_RF = ir2; IR_RR = ir1;

        new_irdata = 0;
    }

    // ************** Call statemachine code / Make navigation decisions ****************
    if((go_home == 0) && (target_user != -1)) {
        goto_target();
    }
    else if(go_home == 1) {
        target_user = 0;
        goto_target();
    }
    else {
        vref = 0;
        turn = 0;
    } // At this point, 'vref' and 'turn' are known

    // ***************** Set manual values when driving via the mouse *********************
```

```
    if(manual == 2) {
        destX = destX_manual;
        destY = destY_manual;
        xy_control(&vref,&turn,turn_thres, Xcurrent, Ycurrent, destX, destY, Angle_rad, target_radius,
target_radius_near);
        avoid_obstacles();
    }

    // ******************** Send the proper commands to the motors ************************
    if(car_on == 1) {
        if(manual == 1) {                           // This is the keyboard driving mode
            PIVelControl(vref_manual,turn_manual);
        }
        else {                                      // For normal and mouse driving modes
            PIVelControl(vref,turn);
        }
    }
    else {                                          // If the car is off, don't move
        PIVelControl(0,0);
        out_PWM(1,1,0);
        out_PWM(1,2,0);
    }

    // ************ Output to the LCD depending on the value on the switches **************
    if(0 == (timeint%200)){
        switch(ledstate){
            case 0:
                LCDPrintfLine1("cstate:%d, close: %d",cstate, almost_there);
                LCDPrintfLine2("IR:%s %s %s %s %s %s",IR7_s,IR8_s,IR4_s,IR3_s,IR2_s,IR1_s);
            break;
            case 1:
                LCDPrintfLine1("L  F1 F2 F3 F4 RF RR");
                LCDPrintfLine2("%s %s %s %s %s %s
%s",IR6_s,IR7_s,IR8_s,IR4_s,IR3_s,IR2_s,IR1_s);
            break;
            case 4:
                LCDPrintfLine1("EncL:%0.1f Enc3:%0.1f",Lmotor, Enc3 );
                LCDPrintfLine2("EncR:%0.1f Enc4:%0.1f",Rmotor, Enc4 );
            break;
            case 9:
                LCDPrintfLine1("Outputing Vision Data");
                LCDPrintfLine2("on Comm 2");
            break;
            case 10:
                LCDPrintfLine1("Echoing ADC to DAC");
                LCDPrintfLine2("");
            break;
            case 12:
                LCDPrintfLine1("V=%.1f,X=%.1f,Y=%.1f",velocity, Xcurrent, Ycurrent);
                LCDPrintfLine2("A=%.3f,x=%.1f,y=%.1f", Angle_rad, vx, vy);
            break;
            case 13:
                LCDPrintfLine1("V=%.1f,X=%.1f,Y=%.1f",velocity, Xcurrent, Ycurrent);
                LCDPrintfLine2("A=%.1f", Angle);
            break;
            case 14:
```

```
                    LCDPrintfLine1("%.3f, %.3f %d",Vpos, Vneg, target_user);
                    LCDPrintfLine2("%.2f, %.2f %d",Cpos, Cneg, cstate);
                break;
                case 15:
                    LCDPrintfLine1("t1= t2= t3= t4= t5=");
                    LCDPrintfLine2("%.1f,%.1f,%.1f,%.1f,%.1f", temp1, temp2, temp3, temp4, temp5);
                break;
                default:
                    LCDPrintfLine1("Change Switches State");
                    LCDPrintfLine2("Undefined State");
                break;
            }// end switch(ledstate)
        }// endif print to display

        // ********************** Send Information to the VB Interface ************************
        if (0 ==(timeint%100)) {
            if(packet == 1) {
                SmallSprintf(sendbuff,"%d %.1f %.1f %.1f %.1f %.1f", packet, vref, turn, Angle, Xcurrent,
Ycurrent);
                packet = 2;
            }
            else if(packet == 2) {
                SmallSprintf(sendbuff,"%d %.1f %.1f %.1f %.1f %d", packet, vx, vy, destX, destY,
target_user);
                packet = 3;
            }
            else if(packet == 3) {
                SmallSprintf(sendbuff,"%d %s %s %s %s %s", packet, IR6_s,IR7_s,IR8_s,IR4_s,IR3_s);
                packet = 4;
            }
            else {
                SmallSprintf(sendbuff,"%d %s %s %d %d %d", packet, IR2_s,IR1_s, goto_method,
termination, obstacle_state);
                packet = 1;
            }
            WirelessSend(sendbuff,strlen(sendbuff));
        }// endif send wireless
}// end ADC_INT7_Func


////////////////////////////////////////////////////////////////////////////////////////////////
// This function acts upon information received from VB
void NetReceiveMessageTask(void){

    char *temp;

    while(1){
        SEM_pend(&SEM_UART1MessageReady,SYS_FOREVER);
        temp = strtok(UART1MessageArray, " ");

        if(strcmp(temp,"vref")==0) {
            temp = strtok('\0', " ");
            vref_manual = atof(temp);
        }
        else if(strcmp(temp,"turn")==0) {
            temp = strtok('\0', " ");
```

```
        turn_manual = atof(temp);
    }
    else if(strcmp(temp,"t1")==0) {
        temp = strtok('\0', " ");
        temp1 = atof(temp);
    }
    else if(strcmp(temp,"t2")==0) {
        temp = strtok('\0', " ");
        temp2 = atof(temp);
    }
    else if(strcmp(temp,"t3")==0) {
        temp = strtok('\0', " ");
        temp3 = atof(temp);
    }
    else if(strcmp(temp,"t4")==0) {
        temp = strtok('\0', " ");
        temp4 = atof(temp);
    }
    else if(strcmp(temp,"t5")==0) {
        temp = strtok('\0', " ");
        temp5 = atof(temp);
    }
    else if(strcmp(temp,"Kp")==0) {
        temp = strtok('\0', " ");
        Kp = atof(temp);
    }
    else if(strcmp(temp,"Ki")==0) {
        temp = strtok('\0', " ");
        Ki = atof(temp);
    }
    else if(strcmp(temp,"Kpt")==0) {
        temp = strtok('\0', " ");
        Kp_turn = atof(temp);
    }
    else if(strcmp(temp,"On")==0) {
        car_on = 1;
    }
    else if(strcmp(temp,"Off")==0) {
        car_on = 0;
    }
    else if(strcmp(temp,"Man0")==0) {
        manual = 0;
    }
    else if(strcmp(temp,"Man1")==0) {
        manual = 1;
    }
    else if(strcmp(temp,"Man2")==0) {
        manual = 2;
    }
    else if(strcmp(temp,"XY")==0) {
        temp = strtok('\0', " ");
        Xcurrent = atof(temp);
        temp = strtok('\0', " ");
        Ycurrent = atof(temp);
    }
    else if(strcmp(temp,"Angle")==0) {
```

```
            temp = strtok('\0', " ");
            Angle = atof(temp);
        }
        else if(strcmp(temp,"destXY")==0) {
            temp = strtok('\0', " ");
            destX_manual = atof(temp);
            temp = strtok('\0', " ");
            destY_manual = atof(temp);
        }
        else if(strcmp(temp,"target")==0) {
            temp = strtok('\0', " ");
            target_user = atoi(temp);
            go_home = 0;
        }
        else if(strcmp(temp,"MAIN")==0) {
            temp = strtok('\0', " ");
            car_on = atoi(temp);
            temp = strtok('\0', " ");
            target_user = atoi(temp);
        }
        else if(strcmp(temp,"return")==0) {
            go_home = 1;
        }
    }
}
```

File #2: user_IR_UltraFuncs.c

```
// STANDARD ANSI INCLUDES
#include <std.h> // DSP/BIOS standard include file
#include <hwi.h>
#include <swi.h>
#include <log.h> // LOG_printf calls
#include <mem.h> // MEM_alloc calls
#include <que.h> // QUE functions
#include <sem.h> // Semaphore functions
#include <sys.h>
#include <tsk.h> // TASK functions
#include <rtdx.h> // RTDX functions
#include <math.h> // sinf,cosf,fabsf, etc.
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

// DSP INCLUDES
#include <c6x.h>        // register defines, in c:\ti\c6000\cgtools\include
#include <c6x11dsk.h>   // TI header in the directory c:\ti\c6000\dsk6x11\include
#include <fastrts67x.h>  // TI's real-time math library, in c:\ti\c6700\mthlib\include

// COECSL INCLUDES      // all COECSL functions are usually in the directory f:\C6713DSK\include
#include <c6xdskdigio.h> // COECSL functions for daughter card, encs, PWM
#include <max3100uart.h> // COECSL functions for communication to max serial chip
#include <dac7724.h>     // COECSL functions for the DAC7724 chip
#include <ad7864.h>     // COECSL functions for the AD7864 chip
#include <RCservo.h>     // COECSL functions to set up PWM ch3 and ch4 to drive RC servos
```

```
#include <switch_led.h>  // COECSL functions for turning off LEDs, monitoring switches
#include <dspvisioncolor50Hz_cLCD.h>
#include <i2c.h>
#include <edma.h>
#include <sharpir.h>
#include <dsk6713.h>
#include <user_ColorVisionFuncs.h>
#include <user_UARTFuncs.h>
#include <user_PIFuncs.h>
#include <user_IR_UltraFuncs.h>
#include <atmel_pwrboard2.h>

volatile int new_irdata = 0;
unsigned char ir_buff[1];
int ir1=0, ir2=0, ir3=0, ir4=0, ir5=0, ir6=0, ir7=0, ir8=0;
int RD_IR_ADDR = 0x20;

void getIRs(void) {

      while(1) {
            if (new_irdata == 0) {
           if ( i2c_stdrecv(RD_IR_ADDR,1,ir_buff,1) == 0 ) {
                        switch(RD_IR_ADDR) {
                              case 0x20:    RD_IR_ADDR = 0x22;    ir1 = ir_buff[0];   break;
                              case 0x22:    RD_IR_ADDR = 0x24;    ir2 = ir_buff[0];   break;
                              case 0x24:    RD_IR_ADDR = 0x26;    ir3 = ir_buff[0];   break;
                              case 0x26:    RD_IR_ADDR = 0x2A;    ir4 = ir_buff[0];   break;
                              // case 0x28: conflicts with LCD screen
                              case 0x2A:    RD_IR_ADDR = 0x2C;    ir6 = ir_buff[0];   break;
                              case 0x2C:    RD_IR_ADDR = 0x2E;    ir7 = ir_buff[0];   break;
                              case 0x2E:    RD_IR_ADDR = 0x20;    ir8 = ir_buff[0];   new_irdata = 1;    break;
                              default:  RD_IR_ADDR = 0x20;     break;
                        }
           } else {
              TSK_sleep(5);
           }
       } else {
           TSK_sleep(10);
       }
      }     // end while
}

void getI2CSensors(void) {
      // Not used
}

void atmel_IRs_task(void) {
      // Not used
}

void atmel_RCservo_task(void) {
      // Not used
}
```

File # 3: user_PIFuncs.c

```
// STANDARD ANSI INCLUDES
#include <std.h> // DSP/BIOS standard include file
#include <hwi.h>
#include <swi.h>
#include <log.h> // LOG_printf calls
#include <mem.h> // MEM_alloc calls
#include <que.h> // QUE functions
#include <sem.h> // Semaphore functions
#include <sys.h>
#include <tsk.h> // TASK functions
#include <rtdx.h> // RTDX functions
#include <math.h> // sinf,cosf,fabsf, etc.
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

// DSP INCLUDES
#include <c6x.h>         // register defines, in c:\ti\c6000\cgtools\include
#include <c6x11dsk.h>    // TI header in the directory c:\ti\c6000\dsk6x11\include
#include <fastrts67x.h>  // TI's real-time math library, in c:\ti\c6700\mthlib\include

// COECSL INCLUDES       // all COECSL functions are usually in the directory f:\C6713DSK\include
#include <c6xdskdigio.h> // COECSL functions for daughter card, encs, PWM
#include <max3100uart.h> // COECSL functions for communication to max serial chip
#include <dac7724.h>     // COECSL functions for the DAC7724 chip
#include <ad7864.h>      // COECSL functions for the AD7864 chip
#include <RCservo.h>     // COECSL functions to set up PWM ch3 and ch4 to drive RC servos
#include <switch_led.h>  // COECSL functions for turning off LEDs, monitoring switches
#include <dspvisioncolor50Hz_cLCD.h>
#include <i2c.h>
#include <edma.h>
#include <sharpir.h>
#include <dsk6713.h>
#include <user_ColorVisionFuncs.h>
#include <user_UARTFuncs.h>
#include <user_PIFuncs.h>

// globals only used by PIVelCopntrol
float encoder1 = 0.0F;
float encoder2 = 0.0F;
float p1_old   = 0.0F;
float p2_old   = 0.0F;
float v1       = 0.0F;
float v2       = 0.0F;
float v1_old   = 0.0F;
float v2_old   = 0.0F;
float u1       = 0.0F;
float u2       = 0.0F;
float e1       = 0.0F;
float e2       = 0.0F;
float e1s      = 0.0F;
float e2s      = 0.0F;
float esteer   = 0.0F;
```

```
// Tunable gains
// Closed-loop coupled velocity control
float Kp       = 3;
float Ki       = 25;
float Kp_turn  = 2;

// Friction Compensation
float Vpos =  0.64*0.1;
float Vneg =  0.72*0.3;
float Cpos =  2.7*0.1;
float Cneg = -2.6*0.3;

// Absolute Positioning Variables
float velocity       = 0.0F;
float vx         = 0.0F;
float vy         = 0.0F;
float Xcurrent      = 0.0F;
float Ycurrent      = 0.0F;
float Angle    = 0.0F;
float Angle_rad = 0.0F;

// Temporary Variables
extern float temp1, temp2, temp3, temp4, temp5;


////////////////////////////////////////////////////////////////////////////////////////////
// This function is told how fast to turn and move forward, then commands the motors to do so.
void PIVelControl(float vref, float turn) {

    float x1,x2;
    read_encoders(1, &encoder1, &encoder2, g_standard_GearMotor);

      ////////////////////////////////////////////////////////////////
      // Find Position, Angle, and Velocity

    // Calculate position (tiles) from encoder readings
    x1 = -(encoder1/59.0)*1.00045;
    x2 = -(encoder2/59.0)*1.00000;

    // Calculate velocity (tiles/sec) from position
    v1 = (x1 - p1_old)/0.001;
    v2 = (x2 - p2_old)/0.001;

    // Fix the flip-over problem with the encoders...
    if ((-100.0F>v1)||(100.0F<v1))  v1 = v1_old;
    if ((-100.0F>v2)||(100.0F<v2))  v2 = v2_old;

      // Center of mass velocity
      velocity = (v1+v2)/2;

      // Track the angle relative to the starting configuration, wrap -180 to 180
      Angle = Angle + (v2-v1)*0.07;
      while(Angle > 180) {
            Angle = Angle - 360;
      }
```

```
  while(Angle < -180) {
      Angle = Angle + 360;
  }
  Angle_rad = Angle * PI / 180;

  // Track velocity in X and Y directions
  vx = velocity*sin(Angle_rad);
  vy = velocity*cos(Angle_rad);

  // Calculate absolute posistion from past data
  Xcurrent = Xcurrent + vx*0.001;
  Ycurrent = Ycurrent + vy*0.001;

  ////////////////////////////////////////////////////////////////
// Save old positions and velocities
p1_old = x1;
p2_old = x2;

v1_old = v1;
v2_old = v2;

  ////////////////////////////////////////////////////////////////
  // Set limits so robot does not tip over or break

  // Don't accelerate too fast, limit based on momentum
  if(vref > velocity + 1.0) {
      vref = velocity + 1.0;
  }

  // Assert maximum velocity and turn rate
  if ( vref > 4 )
      vref = 4;
  if ( vref < -4 )
      vref = -4;

  if ( turn > 2 )
      turn = 2;
  if ( turn < -2 )
      turn = -2;

  ////////////////////////////////////////////////////////////////
// Calculate PI Coupled Control Effort
esteer = v2 - v1 + turn;
e1     = vref - v1 + Kp_turn*esteer;
e2     = vref - v2 - Kp_turn*esteer;
e1s    = e1s + e1;
e2s    = e2s + e2;
u1     = Kp*e1 + Ki*0.001*e1s;
u2     = Kp*e2 + Ki*0.001*e2s;

  ////////////////////////////////////////////////////////////////
  // Perform Final Checks and Adjustments

// Check for integral windup
if (fabs(u1)>10.0) e1s = e1s * 0.99;
if (fabs(u2)>10.0) e2s = e2s * 0.99;
```

```
   // Friction compensation
   if (v1>= 0.0) {
      u1 = u1 + Vpos*v1 + Cpos;
   }
   else {
      u1 = u1 + Vneg*v1 + Cneg;
   }

   if (v2>= 0.0) {
      u2 = u2 + Vpos*v2 + Cpos;
   }
   else {
      u2 = u2 + Vneg*v2 + Cneg;
   }

   // Final check to make sure within range
   if (u1> 10)  u1 =  10.0;
   if (u1<-10)  u1 = -10.0;
   if (u2> 10)  u2 =  10.0;
   if (u2<-10)  u2 = -10.0;

     ////////////////////////////////////////////////////////////////
   // Send PWM command to motors
   out_PWM(1,1,u1);
   out_PWM(1,2,-u2);
}
```

File #4: user_statemachine.c

```
// STANDARD ANSI INCLUDES
#include <std.h> // DSP/BIOS standard include file
#include <hwi.h>
#include <swi.h>
#include <log.h> // LOG_printf calls
#include <mem.h> // MEM_alloc calls
#include <que.h> // QUE functions
#include <sem.h> // Semaphore functions
#include <sys.h>
#include <tsk.h> // TASK functions
#include <rtdx.h> // RTDX functions
#include <math.h> // sinf,cosf,fabsf, etc.
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

// DSP INCLUDES
#include <c6x.h>         // register defines, in c:\ti\c6000\cgtools\include
#include <c6x11dsk.h>    // TI header in the directory c:\ti\c6000\dsk6x11\include
#include <fastrts67x.h>  // TI's real-time math library, in c:\ti\c6700\mthlib\include

// COECSL INCLUDES       // all COECSL functions are usually in the directory f:\C6713DSK\include
#include <c6xdskdigio.h> // COECSL functions for daughter card, encs, PWM
#include <max3100uart.h> // COECSL functions for communication to max serial chip
```

```
#include <dac7724.h>     // COECSL functions for the DAC7724 chip
#include <ad7864.h>     // COECSL functions for the AD7864 chip
#include <RCservo.h>     // COECSL functions to set up PWM ch3 and ch4 to drive RC servos
#include <switch_led.h>  // COECSL functions for turning off LEDs, monitoring switches
#include <dspvisioncolor50Hz_cLCD.h>
#include <i2c.h>
#include <edma.h>
#include <sharpir.h>
#include <dsk6713.h>
#include <user_ColorVisionFuncs.h>
#include <user_UARTFuncs.h>
#include <user_PIFuncs.h>
#include <user_IR_UltraFuncs.h>
#include <atmel_pwrboard2.h>
#include <color_LCD.h>
#include <xy.h>
#include <statemachine.h>

// State Variables
int cstate                  = ptA;
int next_pt             = ptA;
int pstate                  = ptA;
int obstacle_state      = NONE;
int obstacle_counter    = 0;
int go_direction        = 0;
int goto_method         = STRAIGHT;
int almost_there        = 0;
int termination         = PROXIMITY;
int opening_counter     = 0;
float close_enough          = 2;
float target_radius     = 0.25;
float target_radius_near= 0.25;
float turn_thres        = 1;
float destX             = 0;
float destY             = 0;
float destX_old         = 0;
float destY_old         = 0;
int going_straight          = 0;
float dist_from         = 0;

extern float vref, turn, Xcurrent, Ycurrent, velocity, Angle_rad, Angle;
extern int car_on;
extern int target_user;
extern int go_home;
extern int IR_R, IR_F1, IR_F2, IR_F3, IR_F4, IR_RF, IR_RR;

float test_point= 0;
float my_temp1   = 0;
float my_temp2   = 0;
float my_temp3   = 0;

// FEATURE POINTS
//'alphabet = Array("A", "B", "C", "D", "E", "F", "G","H", "I", "J", "K", "L", "M",  "N",  "O", "P", "Q",
"R", "AA", "AB", "AC", "AD", "AE", "AF", "AG", "zA", "zB", "zC", "zD", "zE")
float featureX[30] = {0, 7,  7,  7,  7,  7,  7, 7,  7,  7,  7,  3,  3,  3,  3,  3,  3,  3,  7,  7,  7,  3,
3,  3,  3,  10,  0,  0,  0,  0};
```

```
float featureY[30] = {0,  0,  18,  24,  30,33.5, 40.5, 44,  56,59.5,  81,  81,  70, 62.5, 57.5,  29,  20, 2.5, 21.5,
26.5,   66,   66, 59.5, 26.5, 21.5, 26.5,   66, 59.5, 26.5, 21.5};


////////////////////////////////////////////////////////////////////////////////////////////////
// Helper function to update the destination, as well as set the method for getting there.
// Once the robot is 'close enough', always drive straight towards destination.
void set_dest(int dest, int method, int end_type) {
      destX_old     = featureX[cstate];
      destY_old     = featureY[cstate];
      destX         = featureX[dest];
      destY         = featureY[dest];
      next_pt  = dest;
      termination   = end_type;
      goto_method= method;
}

////////////////////////////////////////////////////////////////////////////////////////////////
// Function that returns the distance the passed coordinate is from the current location.
float current_dist_from(float myX, float myY) {
      my_temp1 = Xcurrent - myX;
      my_temp2 = Ycurrent - myY;
      my_temp1 = my_temp1 * my_temp1;
      my_temp2 = my_temp2 * my_temp2;
      my_temp3 = sqrtsp(my_temp1 + my_temp2);
      return my_temp3;
}

////////////////////////////////////////////////////////////////////////////////////////////////
// This function returns a 1 if the robot is facing the correct cardinal direction (only N or S)
int correct_direction(void) {
      if(go_direction == NORTH) {
            if((Angle < 90) && (Angle > -90)) {
                  return 1;
            }
            else {
                  return 0;
            }
      }
      else {            // if(go_direction == SOUTH) {
            if((Angle < 90) && (Angle > -90)) {
                  return 0;
            }
            else {
                  return 1;
            }
      }
}

////////////////////////////////////////////////////////////////////////////////////////////////
// Highest level navigation controller that calls all necessary functions.
void goto_target(void) {

      // First, figure out which cardinal direction the target is in.
      // This will be more complicated in a bigger building.
      if(featureY[target_user] > Ycurrent - 5) {
```

```
        go_direction = NORTH;
    }
    else {
        go_direction = SOUTH;
    }

    // Second, decide which checkpoint the robot is heading towards, and how it will get there.
    choose_path();

    // Third, set ideal velocity and turn rate.
    set_ideal();

    //test_point = current_dist_from(destX, destY);
    if(current_dist_from(destX, destY) < close_enough) {
        almost_there = 1;
    }

    // Fourth, determine if the robot has arrived at the checkpoint.
    check_next_state();

    // Fifth, naviagate around any obstacles in the robot's path.
    if((almost_there == 1) && (termination == FRONT_WALL)) {
        // Do nothing, because any obstacle seen should be the front wall.
    }
    else {
        avoid_obstacles();
    }
}

////////////////////////////////////////////////////////////////////////////////////////////////
// This function chooses the path from the current location to a target location
// Now this assumes that the target must be zA, zB, zC, zD, or zE
void choose_path(void) {

    switch(cstate) {
        case ptA:     set_dest(ptB, STRAIGHT, FRONT_WALL);     break;
        case ptB:     set_dest(ptC, WALL, OPENING_RIGHT);      break;
        case ptC:
            if(target_user == ptzE) {
                set_dest(ptAA, STRAIGHT, PROXIMITY);
            }
            else {
                set_dest(ptD, STRAIGHT, PROXIMITY);
            }
            break;
        case ptD:
            if((target_user == ptzA) || (target_user == ptzD)) {
                set_dest(ptAB, STRAIGHT, PROXIMITY);
            }
            else {
                set_dest(ptE, STRAIGHT, PROXIMITY);
            }
            break;
        case ptE:     set_dest(ptF, STRAIGHT, PROXIMITY);      break;
        case ptF:set_dest(ptG, WALL, OPENING_RIGHT);       break;
        case ptG:     set_dest(ptH, STRAIGHT, PROXIMITY);      break;
```

```
case ptH:     set_dest(ptI, WALL, OPENING_RIGHT);        break;
case ptI: set_dest(ptJ, STRAIGHT, PROXIMITY);          break;
case ptJ:
    if(target_user == ptzC) {
        set_dest(ptAE, STRAIGHT, PROXIMITY);
    }
    else if(target_user == ptzB) {
        set_dest(ptAC, WALL, PROXIMITY);
    }
    else {
        set_dest(ptK, WALL, PROXIMITY);
    }
    break;
case ptK:     set_dest(ptL, STRAIGHT, FRONT_WALL);     break;
case ptL:     set_dest(ptM, WALL, OPENING_RIGHT);      break;
case ptM:
    if(target_user == ptzB) {
        set_dest(ptAD, STRAIGHT, PROXIMITY);
    }
    else {
        set_dest(ptN, STRAIGHT, PROXIMITY);
    }
    break;
case ptN:
    if(target_user == ptzC) {
        set_dest(ptAE, STRAIGHT, PROXIMITY);
    }
    else {
        set_dest(ptO, STRAIGHT, PROXIMITY);
    }
    break;
case ptO:     set_dest(ptP, WALL, OPENING_RIGHT);        break;
case ptP:
    if(target_user == ptzD) {
        set_dest(ptAF, STRAIGHT, PROXIMITY);
    }
    else if(target_user == ptzE) {
        set_dest(ptAG, STRAIGHT, PROXIMITY);
    }
    else {
        set_dest(ptQ, STRAIGHT, PROXIMITY);
    }
    break;
case ptQ:     set_dest(ptR, WALL, OPENING_RIGHT);        break;
case ptR:     set_dest(ptA, STRAIGHT, PROXIMITY);        break;

case ptAA:    set_dest(ptAG, STRAIGHT, PROXIMITY);break;
case ptAB:    set_dest(ptzA, STRAIGHT, FRONT_WALL);    break;
case ptAC:    set_dest(ptAD, STRAIGHT, PROXIMITY);break;
case ptAD:    set_dest(ptzB, STRAIGHT, FRONT_WALL);    break;
case ptAE:    set_dest(ptzC, STRAIGHT, FRONT_WALL);    break;
case ptAF:    set_dest(ptzD, STRAIGHT, FRONT_WALL);    break;
case ptAG:    set_dest(ptzE, STRAIGHT, FRONT_WALL);    break;

case ptzA:    car_on = 1;   break;
case ptzB:    car_on = 1;   break;
```

```
            case ptzC:     car_on = 1;   break;
            case ptzD:     car_on = 1;   break;
            case ptzE:     car_on = 1;   break;

            case LOST:  robot_lost();                                   break;

            default:  car_on = 0;   break;    // Should not be here
        } // case statement
} // end choose_path()

////////////////////////////////////////////////////////////////////////////////////////////
// This function sets the 'vref' and 'turn' values in an ideal situation.
// Once close enough, change states and head towards next checkpoint
void set_ideal(void) {

    if(current_dist_from(destX_old, destY_old) < 0.5) {
        xy_control(&vref,&turn,turn_thres, Xcurrent, Ycurrent, destX, destY, Angle_rad, target_radius,
target_radius_near);
        if(( turn < 0.1 ) && ( turn > -0.1)) {
            // get going in the right direction
        }
        else {
            vref = 0;
        }
    }
    else {
        switch(goto_method) {
            case STRAIGHT:
                // Simply drive straight towards destination.
                xy_control(&vref,&turn,turn_thres, Xcurrent, Ycurrent, destX, destY, Angle_rad,
target_radius, target_radius_near);
                break;

            case WALL:
                // Follow the right wall, which will presumably lead to the next checkpoint.

                if(almost_there == 0) {
                    turn = (11-IR_RR)*0.1 + (IR_RR - IR_RF)*0.05;
                }
                else {
                    turn = (11-IR_RR)*0.1;
                }

                // Prevent Robot from driving in circles in the middle of the hallway
                if((correct_direction() == 0) && (turn < 0)) {
                    turn = 0;
                }

                // Constant forward velocity, slow down when near target (but don't stop)
                vref = current_dist_from(destX, destY) + 0.5;
                if(vref > 1.5) {
                    vref = 1.5;
                }

                // If going straight for 2 seconds, and have not hit any obstacles, update position + angle
                if((IR_RR <= 12) && (IR_RR >= 10) && (velocity > 1.25)) {
```

```
                        going_straight++;
                        if((turn == 0) && (going_straight > 2000)) {
                            dist_from = current_dist_from(destX_old, destY_old);
                            if(go_direction == NORTH) {
                                Angle = 0;                          // Heading North
                                Xcurrent = destX_old;               // X coordinate of previous
checkpoint
                                Ycurrent = destY_old + dist_from;  // All distance travelled was straight
north
                            }
                            else {
                                Angle = 180;                        // Heading South
                                Xcurrent = destX_old;               // X coordinate of previous
checkpoint
                                Ycurrent = destY_old - dist_from;   // All distance travelled was straight
south
                            }
                        }
                    }
                    else {
                        going_straight = 0;
                    }
                    break;

                default:  car_on = 0;   break;    // Should not be here
            } // end case
        } // end if
} // end set_ideal()

////////////////////////////////////////////////////////////////////////////////////////////
// This function checks if the robot should advance to the next checkpoint.
void check_next_state(void) {
    float vref_temp = 0;
    float turn_temp = 0;

    if(almost_there == 1) {
        switch(termination) {
            case FRONT_WALL:
                if((IR_F1 <= 8) || (IR_F2 <= 8) || (IR_F3 <= 8) || (IR_F4 <= 8)) {
                    set_next_state();
                    vref = 0;
                    turn = 0;
                }
                else {
                    vref = 1.0;
                    turn = 0;
                }
                break;
            case OPENING_RIGHT:
                if((IR_RF - IR_RR >= 6) || (opening_counter > 0)) {
                    opening_counter++;
                    vref = 0;
                    turn = 0;
                    if(opening_counter > 2000) {
                        Xcurrent = destX;
                        Ycurrent = destY;
```

```
                        set_next_state();
                        opening_counter = 0;
                    }
                }
                break;
            case PROXIMITY:
                if(xy_control(&vref_temp,&turn_temp,turn_thres, Xcurrent, Ycurrent, destX, destY,
Angle_rad, target_radius, target_radius_near)) {
                    // Do not do this at every proximity termination, just D through J
                    if(( IR_RR > 12 ) && (next_pt < ptD) && (next_pt > ptJ)) {
                        dist_from = (IR_RR - 12)/12;
                        // Only setup for driving north
                        Xcurrent = destX - dist_from;
                        Ycurrent = destY - dist_from/10;
                    }
                    set_next_state();
                }
                break;
            default:  car_on = 0;   break;    // Should not be here
        } // end switch(termination)

        // If we drove past the checkpoint, move on.
        if(destX > 5) {                         // Check if going north or south
            if(Ycurrent > destY + 1.5) {
                set_next_state();
            }
        }
        else {
            if(Ycurrent < destY - 1.5) {
                set_next_state();
            }
        }
    } // end if(almost_there)

    // If very far ahead, move on.
    if(destX > 5) {
        if(Ycurrent > destY + 5) {
            set_next_state();
        }
    }
    else {
        if(Ycurrent < destY - 5) {
            set_next_state();
        }
    }
} // end check_next_state()

////////////////////////////////////////////////////////////////////////////////////////////////
// This function makes all needed changes to advance to the next state.
void set_next_state(void) {
    pstate = cstate;
    cstate = next_pt;
    almost_there = 0;
} // end set_next_state()

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

```
// This function will use IR sensor information to navigate around obstacles.
void avoid_obstacles(void) {
	float vref_temp = 0;
	float turn_temp = 0;

	switch(obstacle_state) {
		case NONE:
			// What happens:
			//  nothing

			// When to change states:
			if((vref == 0) || ((termination == FRONT_WALL) && (almost_there == 1))) {
				// We're expecting a wall, or not moving forwards, so do nothing here.
			}
			else {
				if((IR_F1 <= 12) || (IR_F2 <= 12) || (IR_F3 <= 12) || (IR_F4 <= 12)) {
					obstacle_state = APPROACHING;
				}
			}
			break;

		case APPROACHING:
			// What happens:
			if(vref > 0.5) {
				vref = 0.5;
			}

			// When to change states:
			if((IR_F1 > 12) && (IR_F2 > 12) && (IR_F3 > 12) && (IR_F4 > 12)) {
				obstacle_state = NONE;
			}
			if((IR_F1 <= 6) || (IR_F2 <= 6) || (IR_F3 <= 6) || (IR_F4 <= 6)) {
				obstacle_state = VERIFY1;
				obstacle_counter = 0;
			}
			break;

		case VERIFY1:
			// What happens:
			vref = 0;
			turn = 0;
			obstacle_counter++;

			// When to change states:
			if((IR_F1 > 6) && (IR_F2 > 6) && (IR_F3 > 6) && (IR_F4 > 6)) {
				obstacle_state = APPROACHING;
			}
			else if((IR_F1 > 12) && (IR_F2 > 12) && (IR_F3 > 12) && (IR_F4 > 12)) {
				obstacle_state = NONE;
			}
			else if(obstacle_counter > 2000) {
				obstacle_state = CLEAR_OBSTACLE;
				obstacle_counter = 0;
			}
			break;
```

```
        case CLEAR_OBSTACLE:
            // What happens:
            vref = 0;
            turn = 0.5;

            // When to change states:
            if((IR_F1 > 12) && (IR_F2 > 12) && (IR_F3 > 12) && (IR_F4 > 12) && ((IR_RF <= 15) ||
(IR_RR <= 15))) {
                obstacle_state = GO_AROUND;
            }
            if(obstacle_counter > 13000) {
                obstacle_state = NONE;
            }
            break;


        case GO_AROUND:
            // What happens:
            vref = 0.5;
            turn = (IR_RR - IR_RF)*0.1;

            if((IR_RR > 15) && (IR_RF > 15)) {
                turn = -0.5;
            }
            if((IR_RR <= 6) || (IR_RF <= 6)) {
                turn = 0.5;
            }

            // When to change states:
            xy_control(&vref_temp,&turn_temp,0, Xcurrent, Ycurrent, destX, destY, Angle_rad, 0, 0);
            if(turn_temp > 0.0) {               // was -0.25
                obstacle_state = NONE;
            }
            if((IR_F1 <= 6) || (IR_F2 <= 6) || (IR_F3 <= 6) || (IR_F4 <= 6)) {
                obstacle_state = VERIFY2;
                obstacle_counter = 0;
            }
            break;

        case VERIFY2:
            // What happens:
            vref = 0;
            turn = 0;
            obstacle_counter++;

            // When to change states:
            if((IR_F1 > 6) && (IR_F2 > 6) && (IR_F3 > 6) && (IR_F4 > 6)) {
                obstacle_state = GO_AROUND;
            }
            else if(obstacle_counter > 2000) {
                obstacle_state = CLEAR_OBSTACLE;
                obstacle_counter = 0;
            }
            break;

        default:  obstacle_state = NONE;    break;
```

```
    } // end obstacle state machine
} // end avoid_obstacles()


////////////////////////////////////////////////////////////////////////////////////////////////
// FIXME: Could not get this concept to work
// ONCE WE HAVE DECIDED WE ARE LOST, WE MUST FIND A LANDMARK
void robot_lost(void) {
    // Code was not worth saving
} // end robot_lost()

////////////////////////////////////////////////////////////////////////////////////////////////
// Compares a given point with what is known of the map. Returns a 1 if inside walls, else a 0.
float check_if_legal(float x, float y) {
    int legal = 1;

    if( x > 11 || x < -3.5 ) {
        legal = 0;
    }
    if( y > 92.5 || y < -4.5 ) {
        legal = 0;
    }
    if( x > 2 && y < -2 ) {
        legal = 0;
    }
    if( x > 8 && ( y < 24 || (y > 29 && y < 84.4))) {
        legal = 0;
    }
    if( x < -1 && y > 2.5 ) {
        legal = 0;
    }
    if( x < 0 && y > 70 ) {
        legal = 0;
    }
    if( x < 2 && ((y > 2.5 && y < 20) || (y > 29 && y < 57.5) || (y > 62.5 && y < 64.5) || (y > 70 && y <
85.5) || (y > 90.5))) {
        legal = 0;
    }
    return legal;
}
```

APPENDIX B: VB INTERFACE CODE

The code used for the user interface is included below. Figure B.1 on page 97 shows the final version of the user interface.

Filename: VB Interface

```
Option Explicit  'Force you to declare all your variables
Dim WorkingOnAMessage As Boolean
Dim ByteNumber As Integer
Dim vref As Single
Dim turn As Single
Dim Angle As Single
Dim Xcurrent As Single
Dim Ycurrent As Single
Dim vx As Single
Dim vy As Single
Dim destX As Single
Dim destY As Single
Dim target_user As Single
Dim Chars_received As String
Dim temp1_flag As Single
Dim temp2_flag As Single
Dim temp3_flag As Single
Dim temp4_flag As Single
Dim temp5_flag As Single
'Dim global_button As Single
Dim global_x As Single
Dim global_y As Single
Dim Begin As Single
Dim num_features As Single
Dim num_features_primary As Single
Dim num_features_secondary As Single
Dim LR As Single
Dim LM As Single
Dim LF As Single
Dim MF As Single
Dim RF As Single
Dim RM As Single
Dim RR As Single
Dim Serial_time As Single
Dim Serial_start As Single
Dim Lost As Single
Dim goto_method As Single
```

```
Dim termination As Single
Dim obstacle_state As Single

Private Sub Map_options_Click(Index As Integer)
   Dim i As Single

   If (Map_options(3).Value = 0) Then
      For i = 0 To num_features_primary Step 1
         pt_lbl(i).Visible = False
      Next
   End If

   If (Map_options(4).Value = 0) Then
      For i = num_features_primary To num_features_secondary Step 1
         pt_lbl(i).Visible = False
      Next
   End If

   If (Map_options(5).Value = 0) Then
      For i = num_features_secondary To num_features Step 1
         pt_lbl(i).Visible = False
      Next
   End If


   If (Map_options(0).Value = 1) Then
      Map_options(3).Enabled = True
      Map_options(4).Enabled = True
      Map_options(5).Enabled = True
      For i = num_features + 1 To pt_lbl.Count - 1 Step 1
         pt_lbl(i).Visible = False
      Next
   End If

   If (Map_options(3).Value = 1) Then
      For i = 0 To num_features_primary - 1 Step 1
         pt_lbl(i).Visible = True
      Next
   End If
   If (Map_options(4).Value = 1) Then
      For i = num_features_primary To num_features_secondary - 1 Step 1
         pt_lbl(i).Visible = True
      Next
   End If
   If (Map_options(5).Value = 1) Then
      For i = num_features_secondary To num_features Step 1
         pt_lbl(i).Visible = True
      Next
   End If

   If (Map_options(0).Value = 0) Then
      Map_options(3).Enabled = False
      Map_options(4).Enabled = False
      Map_options(5).Enabled = False
      For i = 0 To num_features Step 1
         pt_lbl(i).Visible = False
```

```
      Next
   End If

   If (Map_options(2).Value = 0) Then
      Map.MousePointer = 12
   End If

   If (Map_options(2).Value = 1) Then
      Map.MousePointer = 2
   End If

End Sub

Private Sub RETURN_Click()
   SerialCom1.Output = Chr(253) & "return " & Chr(255)
End Sub

Private Sub CountdownTimer_Timer()
   If Begin = 1 Then
      If Val(Countdown) <= 1 Then
         On_button.Value = True
         Countdown.Text = 0
      Else
         Countdown.Text = Val(Countdown) - 1
      End If
   End If
End Sub

Private Sub Driver_Click()
   SerialCom1.Output = Chr(253) & "Man1" & Chr(255)
   Label12.Enabled = True
   Vscrollval.Enabled = True
   Label11.Enabled = True
   HScrollVal.Enabled = True
   VScroll1.Enabled = True
   HScroll1.Enabled = True
   Label24.Enabled = False
   GotoX_lbl.Enabled = False
   Label20.Enabled = False
   GotoY_lbl.Enabled = False
   GotoX_lbl.Caption = "X"
   GotoY_lbl.Caption = "X"
   VScroll1.Value = 0
   HScroll1.Value = 0
End Sub

Private Sub GotoXY_Click()
   SerialCom1.Output = Chr(253) & "Man2" & Chr(255)
   Label12.Enabled = False
   Vscrollval.Enabled = False
   Label11.Enabled = False
   HScrollVal.Enabled = False
   VScroll1.Enabled = False
   HScroll1.Enabled = False
   Label24.Enabled = True
   GotoX_lbl.Enabled = True
```

```
    Label20.Enabled = True
    GotoY_lbl.Enabled = True
    VScroll1.Value = 0
    HScroll1.Value = 0
End Sub

Private Sub HScroll1_Change()
    HScrollVal.Caption = -HScroll1.Value / 100
    SerialCom1.Output = Chr(253) & "turn " & HScrollVal.Caption & Chr(255)
End Sub

Private Sub Driver_KeyDown(keycode As Integer, shift As Integer)
    If keycode = vbKeyA Then
        HScroll1.Value = -VScroll1.LargeChange * 1.5
    ElseIf keycode = vbKeyD Then
        HScroll1.Value = VScroll1.LargeChange * 1.5
    ElseIf keycode = vbKeyS Then
        VScroll1.Value = VScroll1.LargeChange * 1.5
    ElseIf keycode = vbKeyW Then
        VScroll1.Value = -VScroll1.LargeChange * 1.5
    End If
End Sub

Private Sub Driver_KeyUp(keycode As Integer, shift As Integer)
    If keycode = vbKeyA Then
        HScroll1.Value = 0
    ElseIf keycode = vbKeyD Then
        HScroll1.Value = 0
    ElseIf keycode = vbKeyS Then
        VScroll1.Value = 0
    ElseIf keycode = vbKeyW Then
        VScroll1.Value = 0
    End If
End Sub

Private Sub Map_MouseDown(button As Integer, shift As Integer, X As Single, Y As Single)
    If (Map_options(2).Value = 1) Then
        X = Round(X, 1)
        Y = Round(Y, 1)
        If button = vbRightButton Then
            X_lbl.Caption = CStr(X)
            Y_lbl.Caption = CStr(Y)
            'SerialCom1.Output = Chr(253) & "X " & X & Chr(255)
            'SerialCom1.Output = Chr(253) & "Y " & Y & Chr(255)
            SerialCom1.Output = Chr(253) & "XY " & X & " " & Y & Chr(255)
            global_x = X
            global_y = Y
        ElseIf button = vbLeftButton Then
            If GotoXY.Value = True Then
                GotoX_lbl.Caption = CStr(X)
                GotoY_lbl.Caption = CStr(Y)
                'SerialCom1.Output = Chr(253) & "destX " & X & Chr(255)
                'SerialCom1.Output = Chr(253) & "destY " & Y & Chr(255)
                SerialCom1.Output = Chr(253) & "destXY " & X & " " & Y & Chr(255)
            End If
        End If
```

```
    End If
End Sub

Private Sub Map_MouseUp(button As Integer, shift As Integer, X As Single, Y As Single)
    Dim temp1 As Variant
    Dim temp2 As Variant
    Dim temp3 As Variant
    If (Map_options(2).Value = 1) Then
        If button = vbRightButton Then
            X = Round(X, 1)
            Y = Round(Y, 1)
            temp1 = (X - global_x)
            temp2 = (Y - global_y)
            If temp1 > 0 Then
                If temp2 > temp1 Then
                    temp3 = 0
                ElseIf temp2 < -temp1 Then
                    temp3 = 180
                Else
                    temp3 = 90
                End If
            Else
                If temp2 > -temp1 Then
                    temp3 = 0
                ElseIf temp2 < temp1 Then
                    temp3 = 180
                Else
                    temp3 = -90
                End If
            End If

            'Angle = temp3
            Angle_lbl.Caption = CStr(temp3)
            SerialCom1.Output = Chr(253) & "Angle " & temp3 & Chr(255)
        End If
    End If
End Sub

Private Sub Off_button_Click()
    SerialCom1.Output = Chr(253) & "Off" & Chr(255)
    VScroll1.Value = 0
    HScroll1.Value = 0
    Begin = 0
End Sub

Private Sub On_button_Click()
    SerialCom1.Output = Chr(253) & "On" & Chr(255)
    Begin = 0
End Sub

Private Sub ManEnable_Click()
    ManControlFrame.Enabled = True
    Wait.Enabled = True
    Driver.Enabled = True
    GotoXY.Enabled = True
    SerialCom1.Output = Chr(253) & "Man1" & Chr(255)
```

```
End Sub

Private Sub ManDisable_Click()
   ManControlFrame.Enabled = False
   Wait.Enabled = False
   Driver.Enabled = False
   GotoXY.Enabled = False
   SerialCom1.Output = Chr(253) & "Man0" & Chr(255)
   Wait.Value = True
   Off_button.Value = True
End Sub

Private Sub RedrawTimer_Timer()
   If Map_options(1).Value = 1 Then
      Map.Cls
   End If
   Map.Circle (Val(X_lbl.Caption), Val(Y_lbl.Caption)), 0.5
   Map.Line (Xcurrent, Ycurrent)-(Xcurrent + vx, Ycurrent + vy)
   If destX_lbl.Caption = "X" Then
      'Do nothing
   Else
      Map.DrawStyle = vbDot
      Map.Line (Val(destX_lbl.Caption), Val(destY_lbl.Caption))-(Val(X_lbl.Caption),
Val(Y_lbl.Caption)), RGB(255, 0, 0)
      Map.DrawStyle = vbSolid
      Map.Line (Val(destX_lbl.Caption) - 1, Val(destY_lbl.Caption))-(Val(destX_lbl.Caption) + 1,
Val(destY_lbl.Caption)), RGB(255, 0, 0)
      Map.Line (Val(destX_lbl.Caption), Val(destY_lbl.Caption) - 1)-(Val(destX_lbl.Caption),
Val(destY_lbl.Caption) + 1), RGB(255, 0, 0)
   End If
End Sub

Private Sub START_Click()
   If Val(Countdown) = 0 Then
      On_button.Value = True
   Else
      Begin = 1
   End If

   'Select_User.Enabled = False
   SerialCom1.Output = Chr(253) & "target " & Select_User.ItemData(Select_User.ListIndex) & Chr(255)
   'temp1.Text = Select_User.ItemData(Select_User.ListIndex)
   'temp1.Text = Select_User.ListIndex

End Sub

Private Sub STOP_Click()
   Off_button.Value = True
   Begin = 0
End Sub

Private Sub Form_Load()
   Dim featureX As Variant
   Dim featureY As Variant
   Dim alphabet As Variant
   Dim i As Single
```

```
SerialCom1.PortOpen = True 'Enable the serial port when form is loaded
Map.AutoRedraw = True

Map.Line (-3.5, -4.5)-(2, -4.5)
Map.Line (2, -4.5)-(2, -2)
Map.Line (2, -2)-(8, -2)
Map.Line (8, -2)-(8, 18)            '(8,20)
'Map.Line (8,18 )-(8,23 )
Map.Line (8, 23)-(8, 24)
Map.Line (8, 24)-(11, 24)
Map.Line (11, 24)-(11, 26)       '(11,24.05)
'Map.Line (11,24.05 )-(11,28.95 )
Map.Line (11, 28.95)-(11, 29)
Map.Line (11, 29)-(8, 29)
Map.Line (8, 29)-(8, 30)
'Map.Line (8,30 )-(8,33 )
Map.Line (8, 33)-(8, 40.5)
'Map.Line (8,40.5 )-(8,43.5 )
Map.Line (8, 43.5)-(8, 56)
'Map.Line (8,56 )-(8,59 )
Map.Line (8, 59)-(8, 81)
Map.Line (8, 81)-(8, 84)
Map.Line (8, 84)-(8, 84.5)
Map.Line (8, 84.5)-(10, 84.5)
Map.Line (10, 84.5)-(10, 85.5)

Map.Line (-3.5, -4.5)-(-3.5, 2.5)
Map.Line (-3.5, 2.5)-(2, 2.5)
Map.Line (2, 2.5)-(2, 20)
Map.Line (2, 20)-(-1, 20)
Map.Line (-1, 20)-(-1, 20.05)
'Map.Line (-1,20.05 )-(-1,23 )
Map.Line (-1, 23)-(-1, 26)          '(-1,24)
'Map.Line (-1,24 )-(-1,28.95 )
Map.Line (-1, 28.95)-(-1, 29)
Map.Line (-1, 29)-(2, 29)
Map.Line (2, 29)-(2, 57.5)
Map.Line (2, 57.5)-(-1, 57.5)
Map.Line (-1, 57.5)-(-1, 57.55)
'Map.Line (-1,57.55 )-(-1,60.5 )
Map.Line (-1, 60.5)-(-1, 62.5)
Map.Line (-1, 62.5)-(2, 62.5)
Map.Line (2, 62.5)-(2, 64.5)
Map.Line (2, 64.5)-(-1, 64.5)
Map.Line (-1, 64.5)-(-1, 64.55)
'Map.Line (-1,64.55 )-(-1,67.5 )
Map.Line (-1, 67.5)-(-1, 70)
Map.Line (-1, 70)-(2, 70)
Map.Line (2, 70)-(2, 78)
Map.Line (2, 78)-(2, 81)
Map.Line (2, 81)-(2, 85.5)
Map.Line (2, 85.5)-(0, 85.5)
Map.Line (0, 85.5)-(0, 90.5)
Map.Line (0, 90.5)-(2, 90.5)
Map.Line (2, 90.5)-(2, 92.5)
```

```
    Map.Line (2, 92.5)-(10, 92.5)

    alphabet = Array("A", "B", "C", "D", "E", "F", "G", "H", "I", "J", "K", "L", "M", "N", "O", "P", "Q",
"R", "AA", "AB", "AC", "AD", "AE", "AF", "AG", "zA", "zB", "zC", "zD", "zE")
    featureX = Array(0, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 3, 3, 3, 3, 3, 3, 3, 7, 7, 7, 3, 3, 3, 3, 10, 0, 0, 0, 0)
    featureY = Array(0, 0, 18, 24, 30, 33.5, 40.5, 44, 56, 59.5, 81, 81, 70, 62.5, 57.5, 29, 20, 2.5, 21.5, 26.5,
66, 66, 59.5, 26.5, 21.5, 26.5, 66, 59.5, 26.5, 21.5)
    num_features = UBound(featureX)
    num_features_primary = 18
    num_features_secondary = 25

    For i = 0 To num_features Step 1
        pt_lbl(i).Left = featureX(i) - 0.5
        pt_lbl(i).Top = featureY(i) + 1
        pt_lbl(i).Visible = True
        'Select_User.AddItem (alphabet(i))
        'Select_User.ItemData(i) = i
    Next
    For i = num_features + 1 To pt_lbl.Count - 1 Step 1
        pt_lbl(i).Visible = False
    Next

    Select_User.AddItem ("zA")
    Select_User.ItemData(0) = 25
    Select_User.AddItem ("zB")
    Select_User.ItemData(1) = 26
    Select_User.AddItem ("zC")
    Select_User.ItemData(2) = 27
    Select_User.AddItem ("zD")
    Select_User.ItemData(3) = 28
    Select_User.AddItem ("zE")
    Select_User.ItemData(4) = 29


    temp1_flag = 0
    temp2_flag = 0
    temp3_flag = 0
    temp4_flag = 0
    temp5_flag = 0
    Off_button.Value = True
    ManDisable.Value = True
    Wait.Value = True
    Begin = 0

    Map.AutoRedraw = False
End Sub



Private Sub SerialCom1_OnComm()
    Dim charBuff As String
    Dim Reading As Integer
    Dim FoundStartFlag As Boolean
    Dim splitstrings As Variant
    Dim i As Single
```

```
   Select Case SerialCom1.CommEvent
      Case 2  ' Event messages... only care about event 2 (receive)
         charBuff = SerialCom1.Input
         Do While (Len(charBuff) > 0)
            If WorkingOnAMessage = False Then  ' Waiting for start signal... 253 in a byte
               FoundStartFlag = False
               Do While ((Len(charBuff) > 0) And (FoundStartFlag = False))
                  If (Asc(Mid(charBuff, 1, 1)) = 253) Then
                     WorkingOnAMessage = True
                     FoundStartFlag = True
                     Chars_received = ""
                  End If
                  ByteNumber = 0
                  charBuff = Mid(charBuff, 2, Len(charBuff) - 1) ' Truncate buffer by one
               Loop
            End If

            Do While (Len(charBuff) > 0) ' Only enter here if have good data left
               ByteNumber = ByteNumber + 1  'Increment our byte number
               Reading = Asc(Mid(charBuff, 1, 1))  'Get integer value of current character
               charBuff = Mid(charBuff, 2, Len(charBuff) - 1) ' Truncate buffer by one
               If Reading = 255 Then 'The stop byte
                  'DoAnother = charBuff    Commented out but left as a reminder Note above
                  WorkingOnAMessage = False
                  ByteNumber = 0
                  charBuff = ""

                  """""""""""""""" DO STUFF WITH NEW DATA """""""""""""""

                  splitstrings = Split(Chars_received, " ", -1, 1)
                  If UBound(splitstrings) = 5 Then

                     If CStr(Val(splitstrings(0))) = "0" Then
                        If (On_button.Value) Then
                           SerialCom1.Output = Chr(253) & "MAIN 1 " &
Select_User.ItemData(Select_User.ListIndex) & Chr(255)
                        Else
                           SerialCom1.Output = Chr(253) & "MAIN 0 " &
Select_User.ItemData(Select_User.ListIndex) & Chr(255)
                        End If

                     End If

                     If CStr(Val(splitstrings(0))) = "1" Then
                        vref = Val(splitstrings(1))
                        turn = Val(splitstrings(2))
                        Angle = Val(splitstrings(3))
                        Xcurrent = Val(splitstrings(4))
                        Ycurrent = Val(splitstrings(5))
                        vref_lbl.Caption = CStr(vref)
                        turn_lbl.Caption = CStr(turn)
                        Angle_lbl.Caption = CStr(Angle)
                        X_lbl.Caption = CStr(Xcurrent)
                        Y_lbl.Caption = CStr(Ycurrent)

                     End If
```

```
If CStr(Val(splitstrings(0))) = "2" Then
   vx = Val(splitstrings(1))
   vy = Val(splitstrings(2))
   destX = Val(splitstrings(3))
   destY = Val(splitstrings(4))
   target_user = Val(splitstrings(5))
   vx_lbl.Caption = CStr(vx)
   vy_lbl.Caption = CStr(vy)
   destX_lbl.Caption = CStr(destX)
   destY_lbl.Caption = CStr(destY)
   target_user_lbl.Caption = CStr(target_user)

End If

If CStr(Val(splitstrings(0))) = "3" Then
   LR = Val(splitstrings(1))
   LM = Val(splitstrings(2))
   LF = Val(splitstrings(3))
   MF = Val(splitstrings(4))
   RF = Val(splitstrings(5))
   LR_lbl.Caption = CStr(LR)
   LM_lbl.Caption = CStr(LM)
   LF_lbl.Caption = CStr(LF)
   MF_lbl.Caption = CStr(MF)
   RF_lbl.Caption = CStr(RF)

End If


If CStr(Val(splitstrings(0))) = "4" Then
   RM = Val(splitstrings(1))
   RR = Val(splitstrings(2))
   goto_method = Val(splitstrings(3))
   termination = Val(splitstrings(4))
   obstacle_state = Val(splitstrings(5))
   RM_lbl.Caption = CStr(RM)
   RR_lbl.Caption = CStr(RR)
   'g_m_lbl.Caption = CStr(goto_method)
   'term_lbl.Caption = CStr(termination)
   'obstacle_lbl.Caption = CStr(obstacle_state)

   For i = 0 To 1 Step 1
     method_opt(i).BackColor = &H8080FF
     method_opt(i).FontBold = False
   Next
   For i = 0 To 3 Step 1
     terminate_opt(i).BackColor = &H8080FF
     terminate_opt(i).FontBold = False
   Next
   For i = 0 To 5 Step 1
     obstacle_opt(i).BackColor = &H8080FF
     obstacle_opt(i).FontBold = False
   Next

   method_opt(goto_method).BackColor = &H80FF80
   terminate_opt(termination).BackColor = &H80FF80
```

```
                    obstacle_opt(obstacle_state).BackColor = &H80FF80

                    method_opt(goto_method).FontBold = True
                    terminate_opt(termination).FontBold = True
                    obstacle_opt(obstacle_state).FontBold = True


                End If

                'Map.Cls     'UNCOMMENT THIS LINE TO CLEAR SCREEN
                'Map.Circle (Xcurrent, Ycurrent), 0.5
                'Map.Line (Xcurrent, Ycurrent)-(Xcurrent + 4 * vx, Ycurrent + 4 * vy)

              """""""""""""" END OF DO STUFF WITH NEW DATA """""""""""
              End If

            Else
              Chars_received = Chars_received & Chr(Reading)
            End If 'Ends Check for stop byte
          Loop  'Ends looping through good data in charBuffer...

        Loop
    End Select

End Sub

Private Sub temp1_Change()
    temp1_flag = 1
    temp1.BackColor = &HC0FFFF
End Sub

Private Sub temp2_Change()
    temp2_flag = 1
    temp2.BackColor = &HC0FFFF
End Sub

Private Sub temp3_Change()
    temp3_flag = 1
    temp3.BackColor = &HC0FFFF
End Sub

Private Sub temp4_Change()
    temp4_flag = 1
    temp4.BackColor = &HC0FFFF
End Sub

Private Sub temp5_Change()
    temp5_flag = 1
    temp5.BackColor = &HC0FFFF
End Sub


Private Sub Update_Button_Click()
    If temp1_flag = 1 Then
        SerialCom1.Output = Chr(253) & "t1 " & temp1.Text & Chr(255)
        temp1.BackColor = &H80000005
```

```
        temp1_flag = 0
    End If
    If temp2_flag = 1 Then
        SerialCom1.Output = Chr(253) & "t2 " & temp2.Text & Chr(255)
        temp2.BackColor = &H80000005
        temp2_flag = 0
    End If
    If temp3_flag = 1 Then
        SerialCom1.Output = Chr(253) & "t3 " & temp3.Text & Chr(255)
        temp3.BackColor = &H80000005
        temp3_flag = 0
    End If
    If temp4_flag = 1 Then
        SerialCom1.Output = Chr(253) & "t4 " & temp4.Text & Chr(255)
        temp4.BackColor = &H80000005
        temp4_flag = 0
    End If
    If temp5_flag = 1 Then
        SerialCom1.Output = Chr(253) & "t5 " & temp5.Text & Chr(255)
        temp5.BackColor = &H80000005
        temp5_flag = 0
    End If


End Sub

Private Sub VScroll1_Change()
    Vscrollval.Caption = -VScroll1.Value / 100
    SerialCom1.Output = Chr(253) & "vref " & Vscrollval.Caption & Chr(255)
End Sub

Private Sub Wait_Click()
    Label12.Enabled = False
    Vscrollval.Enabled = False
    Label11.Enabled = False
    HScrollVal.Enabled = False
    VScroll1.Enabled = False
    HScroll1.Enabled = False
    Label24.Enabled = False
    GotoX_lbl.Enabled = False
    Label20.Enabled = False
    GotoY_lbl.Enabled = False
    GotoX_lbl.Caption = "X"
    GotoY_lbl.Caption = "X"
    VScroll1.Value = 0
    HScroll1.Value = 0
End Sub
```

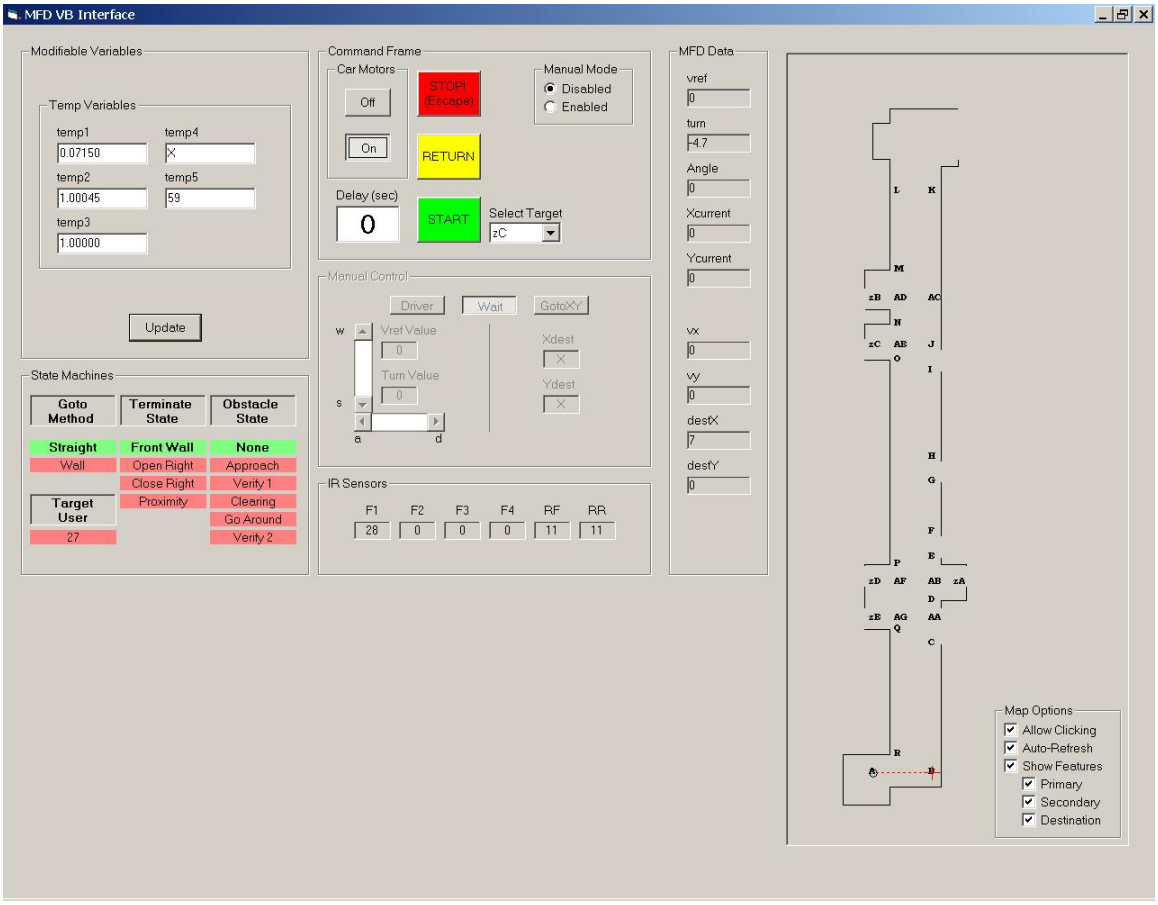


Figure B.1: Final version of the user interface.