The Robert Paulson Project

Prepared for:

Dr. Stephen Bruder
Dr. Kevin Wedeward

Written by:

Scott Dearie
Kevin Fisher
Brian Rajala
Steven Wasson

May 7, 2001

Abstract

The following report will show the design and concepts used in the creation of an autonomous robot that could compete in the Trinity College Home Fire-Fighting Robot Contest. The rules of this contest stipulate that the robot must navigate through a maze, find a candle, extinguish the flame and then return to its original location. The robot was required to sense its surroundings, and have closed loop motor control. We were able to accomplish this task with several weeks left in the term. With the extra time available we implemented tone start and arbitrary starting point. They were not required but this greatly reduced our score at the Trinity competition.

Table of Contents

List of Figures and Tables

Figures

Tables

Introduction

Scope

This group was tasked to design and build a robot that could navigate a maze, detect a flame, and extinguish a flame while adhering to the following design criteria:

1. Fully Autonomous
2. Optical isolation between high-power and low-power electronics
3. Demonstrate motor speed and/or position measurements through Altera PLD
4. Closed-loop speed and wall-following control
5. Main program must be in 'C'
6. Neat and clean final design with good connectors
7. Spend no more than $300

We added the following criteria to make our robot more competitive at the international competition:

1. Start off a 3.5KHz tone
2. Start in any room of the maze
3. Return to the room that it started in

Purpose

By reading this paper it will be possible for a person with the following prerequisites to be able to reconstruct our robot and emulate our results.

1. In-depth knowledge of the MC68HC12 micro-controller.
2. Analog and digital circuit design.
3. Principles of linear time-invariant systems.
4. Proficiency in C programming

Hardware design

The hardware functioned as the body, eyes, ears, and mouth of our robot. It allowed our robot to have a very high level of awareness of its surroundings. It was able to see the walls of the maze, look for fire using two methods, and listen for a tone that indicated that it should start. Our hardware consisted of the following components:

Chassis: this is the body of the robot that all other items are attached to.
Tone sensor: this allowed the robot to listen for a tone:
Wall sensors: these allowed the robot to measure the size of rooms and follow walls
Fire sensors: these two devices allowed the robot to see a fire by detecting UV light and visible light.
LED Board: this device allowed our robot to tell us what signals it was receiving from its other components.

Chassis

No matter how good our subsystems were they would be useless without a good chassis to mount them on. We discovered this very early on when we were having problems getting caught on the walls of the maze. The chassis of our robot was designed to fit within the following criteria. It should be able to carry all of our necessary sensors on one plate, have a symmetrical layout, isolate high and low power sensors, be lightweight, and have a low center of gravity with its weight centered more to the rear.

Of all these requirements, the toughest to meet was to have a robot with a low center of gravity and still have most of its weight in the rear. To achieve this we mounted our high power supply in the rear of the robot with the micro-controller above it. This ensured that our robot would not tip forward when it had to make a sudden stop. From this basic design we then placed the rest of our sensors on the robot chassis.

On the bottom of the plate we placed the white line sensor at the front followed by the H-bridge. At the middle of the robot we placed the motors. This ensured that the

robot would still have a symmetrical design. The top plate consisted of the wall and fire sensors. This design allowed us to isolate the high and low power sensors as much as possible; the white line sensor being the exception. The wall and fire sensors are light in comparison to the power supply, so the placement of these sensors had little effect on the distribution of weight.

In order to mount the fan we added a second plate over the HC-12. We also used this second plate to mount our low power supply, tone decoder, fan relay, and LEDs. This plate did throw off the center of gravity some, but not so much that we had to change our design.

After we had a functional design we made our final plate. Having a preliminary plate was very useful because we were able to experiment with sensor placement without harming the final look of the robot. This approach also allowed us to start working with the robot without spending time making a perfectly round plate and to note any defects in our design. The major flaw in our design was that our wheels were on the outside of our chassis. The final plate was constructed from 1/8" Plexiglas. The final plate design is shown below. Note the new placement of the motors.

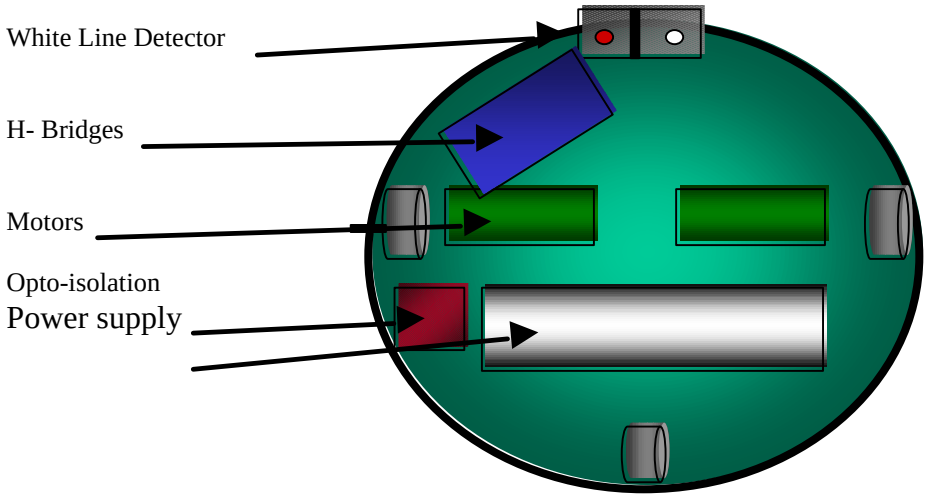

Figure 1: Bottom of Plate 1

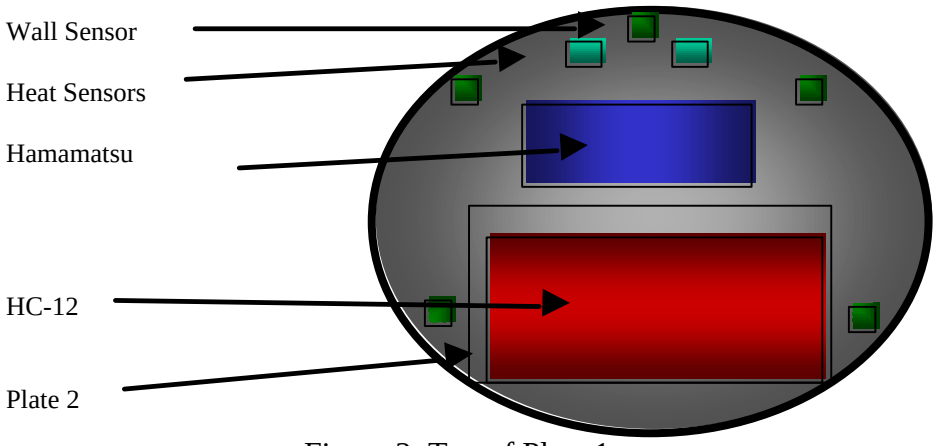

Figure 2: Top of Plate 1

Sensors

All of the following sensors were integrated with the HC-12 to operate as one unit that allowed our robot to sense and react to its environment.

Tone Decoder

The tone decoder will start the robot when the microphone picks up a "smoke alarm" going off. We used a LM567 PLL chip by National Semiconductor. The circuit is active low. When a signal of 3.5 kHz goes into the microphone the logic drops from 5V to 0V. There is a 10k pot on pin 5, which will allow us to adjust the frequency between 2 to 4 kHz. The pot was necessary because the tone generator gets weaker with extended use. C3 and the center frequency determine the bandwidth. C2 and R2 are used to get the center frequency. The equation for center frequency is fo=1/(1.1*R2*C2). The equation for bandwidth is BW=1070sqrt(vi/fo*C3). Where vi is the input voltage which is less than 200mv. We gave it a 10% bandwidth to enable the tone to go low only on a signal near the desired frequency. The tone must be about an inch away for it to activate. This was useful because it prevented other group's tones from starting our robot. Overall the device was useful because it made testing the robot easier and gave us a time reduction in the competitions.

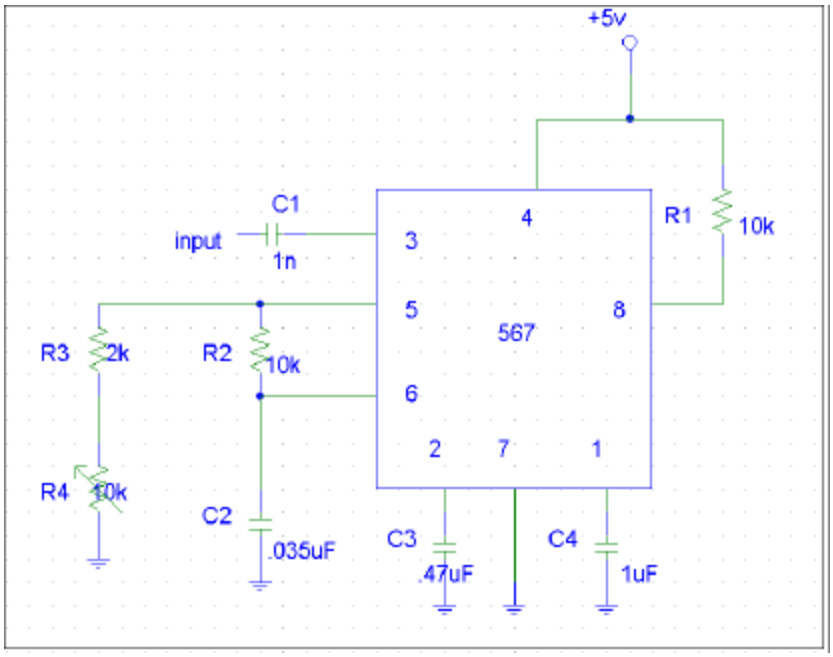

Figure 3: Tone Decoder Circuit

White Line Sensor

While traveling the maze, the robot will encounter white lines at two specific times; when entering or exiting a room and when it is near the candle. In order to observe these lines we chose to look for the transition from dark to light. To do this we used the fact that IR light is reflected by a white surface while it is absorbed by a black surface.

We purchased an IR emitter/detector pair from the local *Radio Shack* to generate and detect the reflected light. The emitter diode was fabricated to only generate a specific frequency of light. Also the detector was designed to only detect the frequency that the diode was emitting. This made it so that the slightest change in frequency would not be detected. This was useful because if the light was reflected back at an angle (as if from a drop of wax) it would not be detected.

The total white line system consisted of the IR emitter/detector matched pair, a Schmitt Trigger, a set of biasing resistors, a 10K pot, and an ambient light shield. The transistor was biased to 5V when it detected IR light this signal was then fed into a

Schmitt Trigger. The Schmitt Trigger was necessary to filter the biased signal into a clean digital signal. The complete schematic can be found below. This sensor worked better than expected and was able to operate in complete darkness and direct light. It was adjustable to detect a white line from 1/8 of an inch to 6 inches off the maze floor.

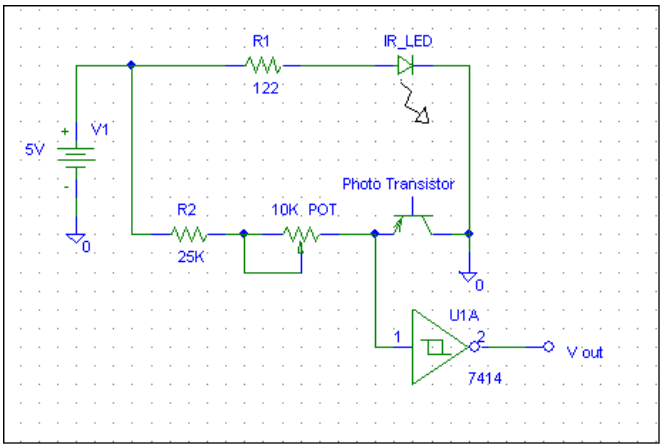

Figure 4: White Line Sensor

Front Wall Sensor

To conserve analog-to-digital ports on the HC12 microcontroller, we decided to digitize our front wall sensor to output a logic "1" or logic "0." With our maze navigation strategy, we did not need to know how far away a wall was in front of us, only if there was a wall in front of us. This data could be sent to the HC12 through a single bit, as opposed to a full eight-bit port. The circuit that we came up with to digitize the front wall sensor's signal uses a comparator and a potentiometer to make the threshold distance adjustable.

Digitizing the front wall sensor proved to be extremely advantageous when we later decided to incorporate two analog right wall sensors; thus using up all our available analog-to-digital ports.

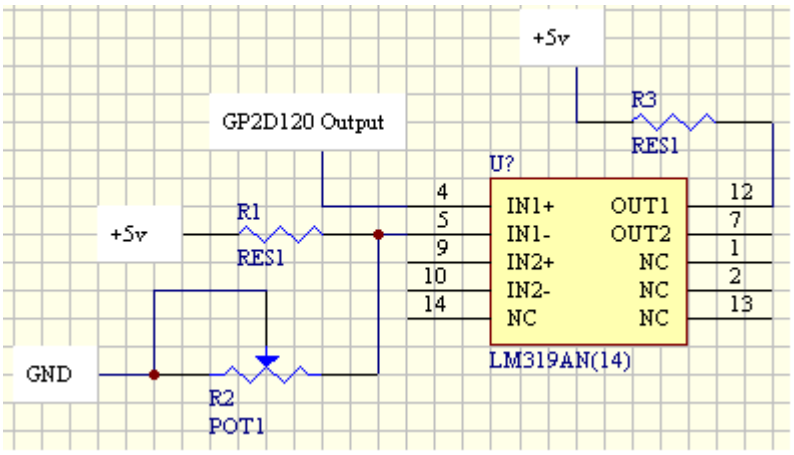

Figure 5: Front Wall Sensor Circuit

Side Wall Sensors

Our robot uses GP2D12 IR sensors to measure its distance from a wall. These sensors are accurate from 9cm to 80cm. This was ideal for our environment because the distances that our robot measured in the hallways of the maze ranged from 12cm to 24cm.

These sensors emit a voltage that is proportional to the distance that the robot is from the wall. The closer the robot is to a wall the higher the voltage. This voltage was then sent to one of the onboard A/D converters and changed into a hexadecimal value. The data sheet for these sensors state that they are accurate starting at 9cm, however we found that the sensors we used were accurate from 11cm as in shown by Figure 6. Even though this is alarming it did not affect our robot because in the worst-case scenario our sensors were 12cm from the wall. Even though the distance from the wall and the voltage are proportional they are not linear. While wall following, however, the portion of the curve that we were operating on was fairly linear. This can be seen on Figure **6**.

Figure 6: Characteristic of the GP2D12

Flame Sensors

For short-range fire detection we decided to use two PN168 phototransistors. In normal lighting conditions the PN168's would saturate so they were filtered with tinted Plexiglas. The Plexiglas also served as housing for the sensors. The sides were covered with electrical tape to block out the ambient light. Each PN168 had a 10k pot to adjust the value of the emitter voltage. This allowed us to adjust the sensitivity. When the output for both PN168s were the same the flame was lined up with the fan. If the output was higher for one PN168, the robot would move in that direction until both values were the same. When the robot noticed a white line and the PN168 output values were in the correct range the fan would turn on.

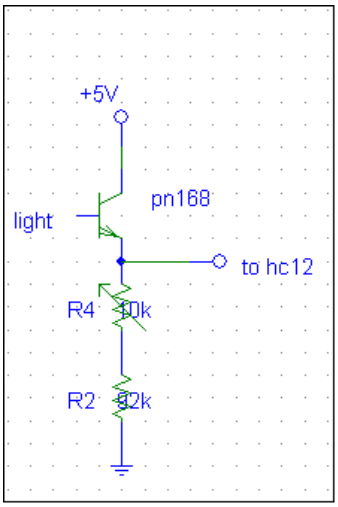

Figure 7: Fire Sensor

Hamamatsu UV Tron & Driving Circuit

From the beginning of our design phase, we chose to utilize the R2868 UV Tron flame sensor from the Hamamatsu Corporation. This sensor can detect a lighter-sized flame at distances greater than five meters. Hamamatsu also sells three types of driving circuit boards (C3704) for use with their R2868 sensor. The difference between these three boards is the supply voltage requirements. We chose to use the C3704-02 driving circuit board because it requires 5v DC to operate – the same as all of our low power circuitry.

The C3704 driving circuit board outputs a 1ms wide pulse if a flame is detected. The detection sensitivity of the circuitry is adjustable on the board itself via a jumper. In using the UV Tron assembly, we had the HC-12 microcontroller look for any pulses from the C3704 driving circuit board within a two-second interval. This two-second interval required us to stop the robot at the entrance to a room in some instances.

Fan Control

The fan control system is comprised of three main components: the fan, a relay, and the fan power supply. The main component of this system is the relay that is used to turn on and off the fan. Initially, we planned on using an Antex solid-state relay that was

opto-isolated. However, the fan assembly that we used drew too much current and fused several relays closed before we tracked down the problem.

The specifications for the DC motor that we used stated that it would only draw a maximum of 2 amps continuous. However when we attached our blade to the motor it drew 4 amps continuous. In order to correct this we had to fabricate our own relay. The key component that was used in this relay was a MJ2955 power transistor. We chose this transistor because it was able to handle 15 amps and 150 watts continuous.

Across this transistor we had a maximum of 17 V at 6 amps. This calculates to 102 watts dissipated well within the given specifications. The complete relay was constructed of 3 transistors and a couple of resistors. The one disadvantage to this design is that we had to connect the high and low power grounds. By doing this, our robots high and low power systems were no longer isolated. We were able to find an isolated relay that met our requirements but did not have the funds or the time to implement this. The complete fan control system is shown below. Note that the system is active high, when Port P5 sends a 5V signal the fan will turn on.

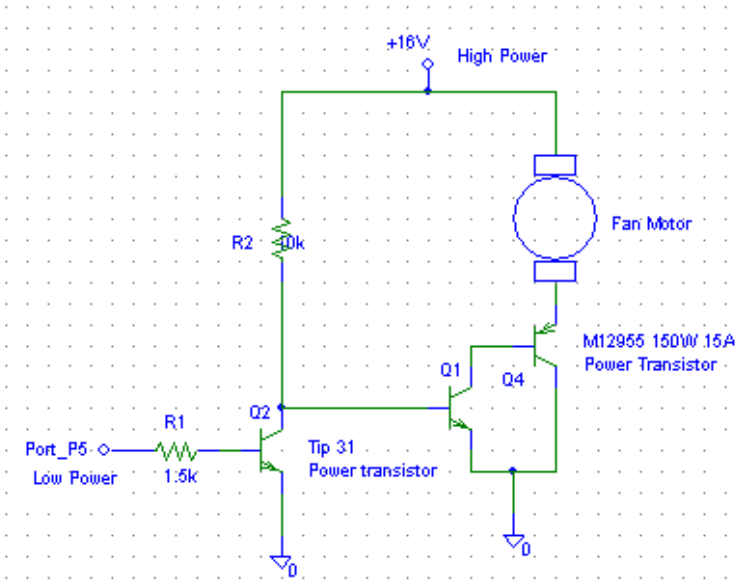

Figure 8: Fan Control System

H Bridges

Allegro 2998

Coming up with our initial design, we chose to use the Allegro 2998 H-Bridge largely because of its cost-effectiveness. With one $7 part, we could control both of our motors. The Allegro 2998 could provide 3A peak current and 2A continuous. Our Maxon 6W green-body motors required a start-up current of less than 2A. This was well within the specifications of the Allegro 2998.

Due to problems that arose with the Maxon 6W green-body motors we had to go with the Maxon 11W motors. These 11W motors had a start-up current of 3.25A – more current than the Allegro 2998 could handle. This problem arose one week before the Trinity College Fire Fighting Home Robot contest was to take place. To resolve this major issue before the international competition, we were forced to borrow an H-Bridge PCB board from Dr. Kevin Wedeward that used the National LMD18200T H-Bridge.

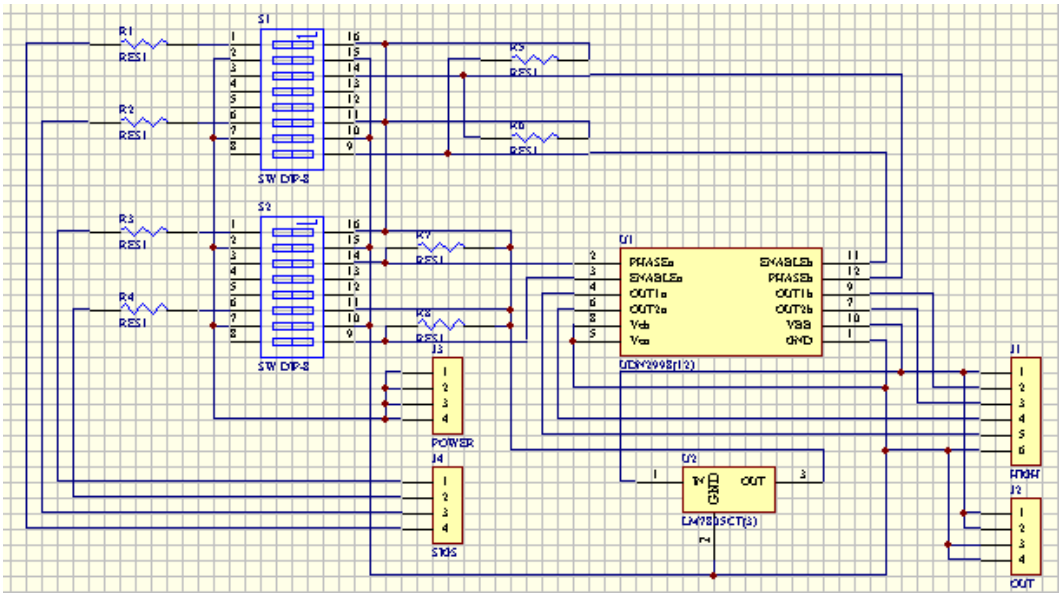

Figure 9: Allegro 2998 H-bridge system with opto-isolation

National LMD18200T

As mentioned previously, we had to use the National LMD18200T H-Bridge chips in our final robot design. The National chips could only control one motor per chip

(as opposed to two motors per chip with the Allegro), so we required the use of two of the National chips. The LMD18200Ts can handle a peak current of 6A and a continuous current of 3A. This limitation was more than enough for the Maxon 11W motors that we ended up using.

Motor Selection

At the start of the semester we analyzed all of the available motors with respect to the torque, start up current, continuous current, encoder design, and speed. This analysis can be found in Appendix A**.** After this analysis we concluded that the motor that suited our needs the best was the *Maxon* 6-watt design. It had a quadrature encoder, a 14:1 gearhead ratio, and a very low startup and continuous current.

As the semester progressed we had to switch motors due to circumstances beyond our control. One of the motors would short and cause our h-bridge to blow. We were persuaded to switch to the *Maxon* 11-watt motor due to supply issues. When we integrated this motor into the robot it performed better than our original choice. During the application of this motor we discovered that it had a better response than the 6-watt motors. The turn in place routines were more accurate as well as the wall following. These motors were shorter and thicker than the *Maxon* 6-watt motors. If we used this motor to begin with our chassis would be smaller. The one draw back to these motors is that our original wheels did not fit the axle. As a consequence of this we had nearly no ground clearance.

LED Board

For the main purpose of testing, we designed a board of LEDs that would light if certain subsystems activated – white line sensor, front wall sensor, and tone decoder. We

also used two lights as power indicators. We also had places wired up for the Hamamatsu sensor and one miscellaneous component, but these were never used.

The LED board proved very useful. We could not tell visually if our high power switch was open or closed without an LED indicator. Having systems wired to LEDs also expedited the calibration process. For example, we could calibrate the tone decoder by sounding the buzzer and turning a potentiometer until the LED reserved for the tone decoder system lit.

There was one minor problem with the LED board. With the low power indicator connected to the LED board, the low power battery drained at a much faster rate. We resolved this issue by only connecting the low power indicator and systems to the LED board when doing calibration testing. This was not as much of an issue with the high power indicator because our high power batteries had a much greater charge capacity. The high power indicator was also the most useful of the LEDs when running the robot in the maze because it was the only way we could tell if the motors were getting power.

Software

The following describes the software developed for this project. Software includes all devices implemented in Altera and the C program for the HC12.

Digital Position and Speed Decoders

We implemented quadrature position and speed decoders into our existing Altera memory expansion board designed in EE 308. The encoder channels were inputted into our Altera chip. A state machine was created to generate a pulse for every state transition of the encoder channels for each motor. Then, 9-bit counters counted these pulses. Simple D flip-flops were used to determine and store the direction of rotation of the

motors. The counters were configured so that if the motors rotated forward, they counted up. Otherwise, they counted down.

The value from the counters is latched into speed registers. The latch signal for this comes from Port P2 of the HC12. This makes it where the latch frequency can be adjusted in software via the HC12. The counters and speed registers are then sign extended to 16-bits and memory mapped into our memory expansion. The left and right motor tick counters are mapped to addresses 0x0400 and 0x0402. The left and right motor speed registers are mapped to address 0x0404 and 0x0406.

Figure 10 on the next page is a block diagram of the digital position and speed decoders and their interface with the HC12. Appendix B contains the Altera code used to implement this.

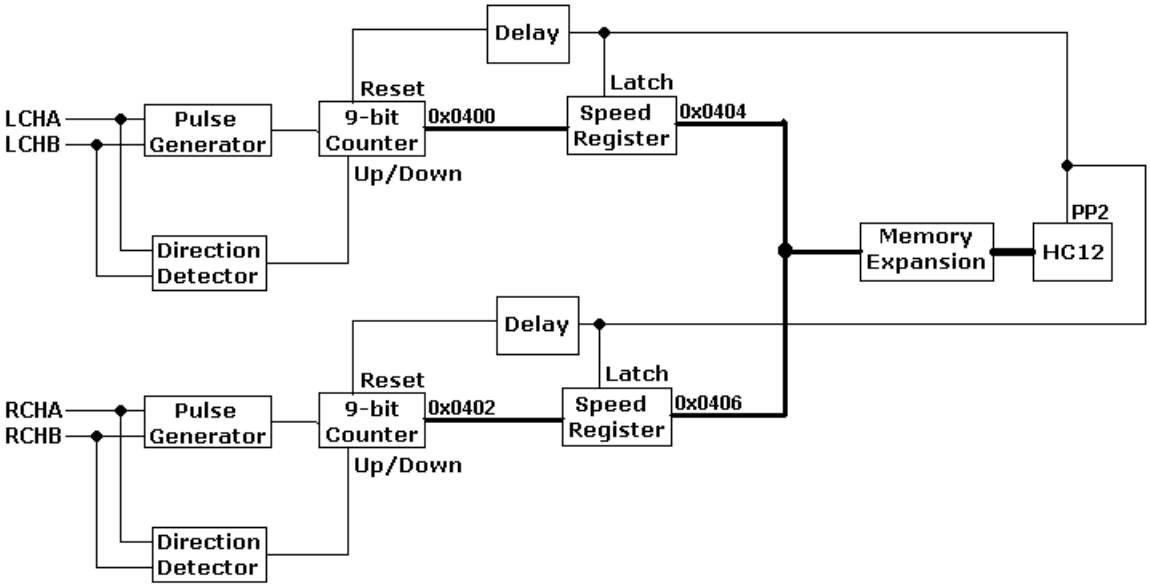

Figure 10: Block diagram of the digital position and speed decoders and the interface between them and the HC12.

HC12 Connections

The table below shows what HC12 resources are being used and where all the devices described earlier connect. It first lists the analog input devices, then the digital input devices, and finally the digital output signals needed by various other devices.

Device	Port	I/O
Left Front Sensor	Port AD2	Input
Left Rear Sensor	Port AD3	Input
Left Infrared Sensor	Port AD4	Input
Right Infrared Sensor	Port AD5	Input
Right Front Sensor	Port AD6	Input
Right Rear Sensor	Port AD7	Input
White Line Sensor	Port DLC0	Input
Front Wall Sensor	Port DLC1	Input
Tone Decoder	Port DLC2	Input
Hamamatsu UV Sensor	Port T7	Input
Left Motor PWM Signal	Port P0	Output
Right Motor PWM Signal	Port P1	Output
Speed Decoder Latch Signal	Port P2	Output
Fan Relay Signal	Port P5	Output
Left Motor Direction Signal	Port P6	Output
Right Motor Direction Signal	Port P7	Output

Table 1: Connections to the HC12

Code Structure

The C code for this project is divided up into four header files and the main C program file. This first header file is named hc12.h and it contains definitions of pointers to various HC12 registers used in this project. The second header file is setup.h which contains functions used to setup various ports on the HC12 to interface with all the devices. The third header file is functions.h. This header file contains functions used to detect the robot's environment, move through the maze, and find and extinguish the flame. The fourth and final header file is navigations.h which contains special functions for navigating through the maze to find the fire, extinguish it, and return to the robot's starting place. The main C program file includes these header files and uses their

functions to carry out the robot's mission of extinguishing the flame. These files are located in Appendix C.

Closed-loop Control

To effectively and efficiently navigate through the maze, we implemented closed-loop control. Four basic functions were implemented: closed-loop speed control to go straight at a desired speed, closed-loop speed control with wall following to go parallel to a wall at a desired speed, turn in place for a desired angle, and flame follow. All of these functions were implemented in a Real Time Interrupt service routine that was executed every 8.192 ms.

For closed-loop speed control, the error between the desired speed and actual speed is added to the present duty cycle. After this is done three times, the desired speeds of the motors are adjusted to correct the robot's orientation to the wall. When the robot is parallel to the wall, the desired speeds are adjusted to correct for the robot's distance to the wall.

Closed-loop speed control is also necessary for turning in place. It ensures that both motors are turning at the same speed in opposite directions. To turn in place for a specific angle, the position decoders are used. The robot turns in place until the number of motor ticks counted by the position decoders reaches a set number for a desired angular rotation.

The final closed-loop control implemented is flame following. This directs the robot to the flame. It does so by adjusting the duty cycles to the motors to correct the robot's orientation to the flame, line it up with the flame, and drive towards the flame. The two PN168 flame sensors determine the orientation of the robot to the flame. If the outputs of the two sensors are equal, then the robot is perfectly lined up with the flame. Otherwise, the flame is off to one of the sides of the robot.

Arbitrary Starting Point

In addition to starting from the home circle, we designed our robot to be capable of starting from any position within any room of the maze. This is done by first determining if the robot is at the home circle by checking if there is a white surface. If there is a white surface, meaning that the robot is on the home circle, the robot will orient itself parallel to the nearest left wall and begin navigating the maze. This means that the robot can start at the home circle at any orientation.

If the robot is in a room, however, it scans the room to find the nearest left wall. Then, it drives towards the nearest left wall that it detected. Next, it will left wall follow through the room until there is no left wall, which occurs at the entrance of the room. It then drives forward until it reaches a front wall. If the robot has crossed a white line while left wall following, then the robot has started from room four and has exited it.

If the robot is in either room one, two, or three, it scans the room from the entrance to determine its size. Then it exits the room. When it reaches a front wall, it checks if there is a right wall. If there is no right wall and the room is large, the robot has started from room 1. If there is a right wall and the room is large, the robot has started from room 2. If there is a right wall and the room is small, the robot has started from room 3. Finally, when the robot is out of the room facing the wall opposite of the room, it makes a 90 degree left turn in place and begins maze navigation.

Maze Navigation

To successfully locate a flame within the maze, our robot must have its behavior pre-programmed. With the limited memory and timeframe, we designed the robot's behavior with simplicity and ease of modification in mind. This led to an extremely modular final set of instructions. Benefits of the modular design included limiting the redundancy of code and easily being able to correct unacceptable behavior. If we wanted

the robot to perform the same actions in many parts of the maze, we could call a single function multiple times. If we discovered problems with the robot's behavior in a specific part of the maze, we could easily correct the error by altering the function written for that part of the maze.

An added bonus of a modular approach was the ease of integrating arbitrary starting point capability. Once the robot discovered its starting position and entered into the hallways of the maze, we could then have our robot run the modular functions in a certain order to successfully search the entire maze for a flame.

No matter where the robot started from, the rooms were searched in the same general order each time. The hierarchy is as follows: room 2, room 1, room 3, and finally room 4. The reasoning behind this order was that the entrances to rooms 2 and 1 are fairly close together, so we could quickly scan those two rooms and have searched half of the maze. We searched room 2 before room 1 because our long-range Hamamatsu fire sensor was mounted to look to the right hand side of the robot. The way the rooms are configured makes it easy to search room 2 and then search room 1 on the way back. Room 4 was our last priority in all cases because the entrance was far away from any of the other rooms. One benefit of using the arbitrary starting point mode was that we only had to search three rooms because the flame could not be placed in the room of origin.

The only problem that we had to deal with in our maze navigation was the length of time it took the Hamamatsu fire sensor to scan a room. To scan the larger rooms (1 and 2) with complete accuracy, our robot had to pause for a second or two at the entrances to be sure there was not a flame present. In most cases this added to our total run time, but it was unavoidable with the nature of the Hamamatsu fire sensor.

Extinguishing Flame and Returning Home

Once a flame has been detected by the Hamamatsu, the robot enters the room in which the flame was detected and begins flame following. The flame follow procedure directs the robot towards the flame without hitting a wall. Once a white surface has been detected and our flame sensors verify the presence of the flame, the fan is turned on and the robot makes a 60-degree sweep to extinguish the flame. Once the flame is extinguished, the robot turns around, exits room, and resumes its maze navigation to return to its starting point. Once the robot has returned to its home, the code is exited.

Production Cost

At the onset of the semester each group was allocated $300 to use in the construction of their robot. We kept a very detailed list of expenditures and we were fairly conscious of this ceiling throughout the semester. Due to the problems with the Maxon 6W green-body motors it was unavoidable that we would go over-budget. We spent much time, effort, and, as one can see from the table, $70 on Allegro 2998 H-Bridge chips in discovering the origin of the problem. In changing to the Maxon 11W motors, we needed to purchase two National LMD18200T H-Bridge chips and an H-Bridge board (listed singularly as “H-Bridge Board” in the table) at $30. Combined, these unavoidable costs account for all but $3.95 (3.8 percent) of the difference.

Item Description	price/ea	quantity	cost
Maxon 11-Watt motor:	$50.00	2	$100.00
H-Bridge (Allegro)UDN2998W	$7.00	10	$70.00
IR distance sensors:0-80cm: GP2D12:	$10.00	4	$40.00
Hamamatsu R2868 UVTron	$35.52	1	$35.52
Antex solid-state opto-isolated2.5A DC relays:	$5.00	5	$25.00
Hamamatsu C3704-02 Driving Circuit	$22.20	1	$22.20
Rechargable 7.2V NiCad pack:	$10.00	1	$10.00
Caster wheel assembly	$10.00	1	$10.00
Schmitt trigger optoisolators	$1.50	6	$9.00
4-pin audio locking conn. female:	$1.50	6	$9.00
IR distance sensors:0-30cm: GP2D120:	$8.00	1	$8.00
crimp contacts: (i.e. pins for above)	$0.10	80	$8.00
wheels (pair)	$6.68	1	$6.68
Wire-wrap DIP socket2x8-pin:	$1.50	4	$6.00
Terminal housing (with pins) 4-pin (pair):	$1.25	4	$5.00
Terminal housing (with pins) 6-pin (pair):	$2.50	2	$5.00
1.5” rubber wheels:	$2.50	2	$5.00
Boxed headers2X5-pin (male)	$1.00	3	$3.00
Perf-board 2.25”X1.8”:	$1.25	2	$2.50
Project enclosure: Small	$2.00	1	$2.00
Perf-board 1.5”X1.75”:	$1.00	2	$2.00
Fuse holder (GMA fuses):	$1.00	2	$2.00
4-pin audio locking conn. male-straight:	$0.50	4	$2.00
4-pin audio locking conn. male-right-ang.:	$1.00	2	$2.00
GMA fuses(1,2,4,6A):	$0.20	8	$1.60
Terminal housing (with pins) 2-pin (pair):	$0.75	2	$1.50
Electret mic (with-leads):	$1.50	1	$1.50
10k Ohm multi-turn pot:	$1.50	1	$1.50
Wire-wrap DIP socket 2x7-pin:	$1.25	1	$1.25
3/16” shrink tubing (price per inch)	$0.10	12	$1.20
3/32” shrink tubing (price per inch)	$0.10	12	$1.20
Tone decoder: NE567	$1.00	1	$1.00
Ribbon Cable per ft.	$0.50	2	$1.00
Plexiglass	$1.00	1	$1.00
Misc. spacers/standoffs 4-40 spacers:	$0.10	8	$0.80
Misc. switches:	$0.50	1	$0.50
GRAND TOTAL			$403.95

Table 2: The development budget for this project

Reproduction Cost

While we were astonished at exactly how costly our robot was to design, we were curious as to how much it would cost to reproduce the robot. In compiling this second price list, we assumed the same cost per component as the electrical engineering department was charging us. In some cases, these prices differ greatly from the retail

value of the components. For the purposes of this theoretical reproduction cost, we also included the prices of items that were donated to us.

Item Description	price/ea	quantity	cost
Maxon 6-Watt 22mm motors (green body):	$50.00	2	$100.00
IR distance sensors:0-80cm: GP2D12:	$10.00	4	$40.00
Hamamatsu R2868 UVTron	$35.52	1	$35.52
Terminal housing (with pins) 4-pin (pair):	$1.25	27	$33.75
Hamamatsu C3704-02 Driving Circuit	$22.20	1	$22.20
Rechargable 7.2V NiCad pack:	$10.00	2	$20.00
Caster wheel assembly	$10.00	1	$10.00
IR distance sensors:0-30cm: GP2D120:	$8.00	1	$8.00
H-Bridge (Allegro)UDN2998W	$7.00	1	$7.00
Schmitt trigger optoisolators	$1.50	4	$6.00
Antex solid-state opto-isolated2.5A DC relays:	$5.00	1	$5.00
1.5” rubber wheels:	$2.50	2	$5.00
Misc. spacers/standoffs 8-32 spacers:	$0.10	35	$3.50
5V regulators: 7805	$0.75	3	$2.25
Fuse holder (GMA fuses):	$1.00	2	$2.00
3.4kHz 9V buzzer:	$1.50	1	$1.50
Electret mic (with-leads):	$1.50	1	$1.50
Reflective IR emitter/detector pair:	$1.50	1	$1.50
1-turn pots10k Ohm	$0.50	3	$1.50
Ribbon Cable per ft.	$0.50	3	$1.50
3/16” shrink tubing (price per inch)	$0.10	12	$1.20
3/32” shrink tubing (price per inch)	$0.10	12	$1.20
Tone decoder: NE567	$1.00	1	$1.00
Plexiglass	$1.00	1	$1.00
Misc. switches:	$0.50	1	$0.50
GRAND TOTAL			$312.62

Table 3: The reproduction budget for this project

High Power Budget

On the high power side of our robot, we were using 16 watts in normal operation – disregarding the fan. At 14.4 volts, the high power systems drew 1.1 amps. Our high power battery could carry a charge of 3.6 amp hours of charge. With these figures, our robot could theoretically run for about 3 hours and 15 minutes. Our high power battery consisted of two 7.2 volt Radio Shack remote control car batteries connected in series.

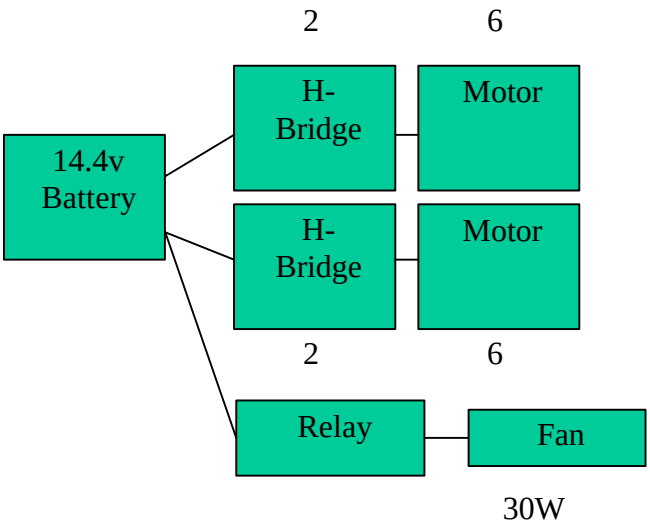

Figure 11: High Power Budget

Low Power Budget

On the low power side of our robot, we were using 3.65 watts in normal operation. At 7.2 volts, the high power systems drew 0.5 amps. Our low power battery could carry a charge of approximately 1.75 amp hours of charge. With these figures, our robot could theoretically run for about 3 hours and 30 minutes. Our low power battery was a Korean cellular phone battery donated by Dr. George Cunningham.

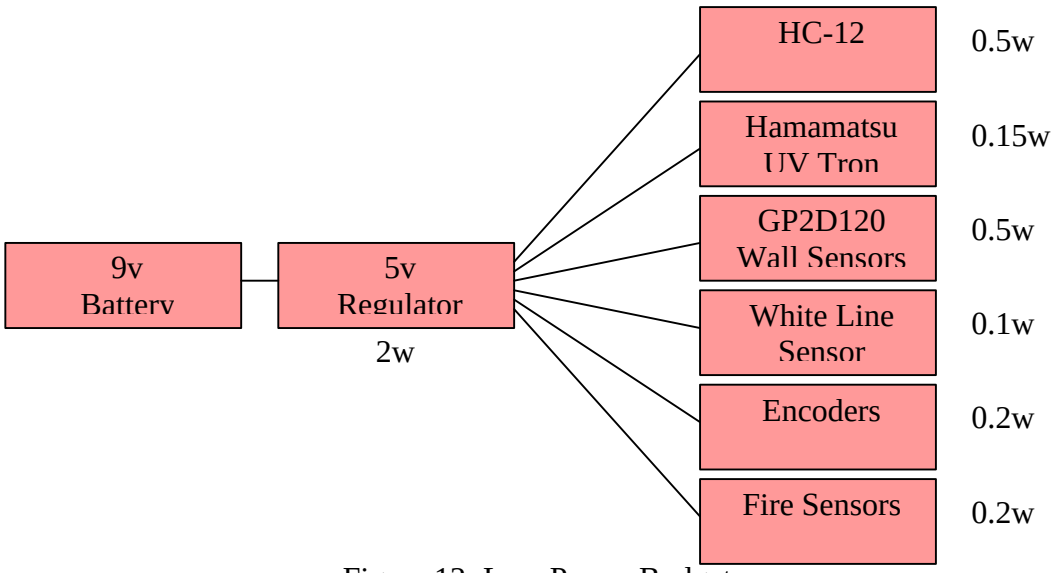

Figure 12: Low Power Budget

Conclusion

This project was a great success. We were able to implement all the required design characteristics and all of our personal goals. Our robot placed 7th at the Trinity Home Fire Fighting Robot Competition and 2nd at the local competition.

If we had the opportunity to develop a 3rd stage of this design we would implement the following improvements:

1. A smaller Chassis- This would allow our robot to travel faster in the maze
2. A better gear-head ratio on the motors so that we could increase the wheel size without speeding up the robot. This would also allow us to do ramps.
3. Add several digital front wall sensors so that we could implement object avoidance.

With the implementation of these four changes we feel that this project would be ready for public distribution.

Appendix A: Motor Selection Criteria

Motor	Ratio	Voltage	No load Current	Starting current	speed w/gh	Toque w/GH	power	Quad
Max 6 green	14:1	18	26	1.28	680	182	5.32	Y
Max 6 gray	30:1	9	78	4.9	346	262	9.6	Y
Max 5	14:1	12	42		510			N
Max 11	10:1	18	37	3.25	920	182	13.6	Y

Appendix B: Altera Code

```
% This code is for HC12 memory expansion, as well as quatrature position and speed
   determination. Position and speed are memory mapped to addresses 0x0400 through
   0x0407 %

SUBDESIGN memexp2
(
   E                 : INPUT;  % E-Clock %
   R_W               : INPUT;  % R/W Line %
   RESETn            : INPUT;  % Reset line %
   LSTRBn            : INPUT;  % Let's us tell if 8-bit or 16-bit access %

   PA[7..0]          : BIDIR;  % Address and Data (15-8) from HC12 %
   PB[7..0]          : BIDIR;  % Address and Data (7-0) from HC12 %

   WEn               : OUTPUT; % Write Enable to memory %
   OEn               : OUTPUT; % Output Enable to memory %
   CS_MEM_ODDn       : OUTPUT; % Chip Select for odd addresses %
   CS_MEM_EVENn      : OUTPUT; % Chip Select for even addresses %
   A[15..1]          : OUTPUT; % Demultiplexed address bits %

   LCHA              : INPUT;  % Left Motor Channel A Encoder %
   LCHB              : INPUT;  % Left Motor Channel B Encoder %
   RCHA              : INPUT;  % Right Motor Channel B Encoder %
   RCHB              : INPUT;  % Right Motor Channel B Encoder %
   LTCHclk           : INPUT;  % Latch clock for speed determination %
)

VARIABLE
   demux[15..0]      : DFF;   % Demuliplexed address internal %
   CS_MEM            : NODE;  % Tell's when address in range of memory %

   IDB[15..0]        : NODE;  % Internal data bus to send to HC12 %
   PA_OE             : NODE;  % High when we want to send data to HC12 on %
                              % the HC12's Port A %
   PB_OE             : NODE;  % High when we want to send data to HC12 on %
                              % the HC12's Port B %
   XX                : DFF;   % Used to allow RESETn to be connected to pin 20 %

   LCNT[8..0]        : DFF;   % Internal 9-bit counter for left motor speed %
   RCNT[8..0]        : DFF;   % Internal 9-bit counter for right motor speed %
   LSPD[8..0]        : DFF;   % Internal 9-bit latched speed register for left motor %
   RSPD[8..0]        : DFF;   % Internal 9-bit latched speed register for right motor %
   LDIR              : DFF;   % Internal direction bit for left motor %
   RDIR              : DFF;   % Internal direction bit for right motor %
   CRESETn           : NODE;  % Internal reset for counters to determine speed %
```

```
    DFLCHA          :  NODE;  % Digitally filtered LCHA %
    DFLCHB          :  NODE;  % Digitally filtered LCHB %
    DFRCHA          :  NODE;  % Digitally filtered RCHA %
    DFRCHB          :  NODE;  % Digitally filtered RCHB %
    LTP             :  NODE;  % Left Transition Pulse %
    RTP             :  NODE;  % Right Transition Pulse %
    DFLTCH          :  NODE;  % Digitally filtered latch clock %

    % State machine to filter LCHA %
    LCHAF: MACHINE WITH STATES (LA10, LA20, LA11, LA21);

    % State machine to filter LCHB %
    LCHBF: MACHINE WITH STATES (LB10, LB20, LB11, LB21);

    % State machine to filter RCHA %
    RCHAF: MACHINE WITH STATES (RA10, RA20, RA11, RA21);

    % State machine to filter RCHB %
    RCHBF: MACHINE WITH STATES (RB10, RB20, RB11, RB21);

    LPG: MACHINE WITH STATES (LXX, LXnX);  % State machine to generate %
                                           % left transition pulse %

    RPG: MACHINE WITH STATES (RXX, RXnX);  % State machine to generate %
                                           % right transition pulse %

    % State machine to filter LTCHclk %
    LTCHF: MACHINE WITH STATES (LTCH10, LTCH20, LTCH11, LTCH21);

    % State machine to reset counters after speed has been latched %
    CRST: MACHINE WITH STATES (C0, C1, C2, C3);

BEGIN
        % The following state machines have E as there clock and the complement of
        RESETn as thee reset %
        LCHAF.clk = E;
        LCHAF.reset = !RESETn;

        LCHBF.clk = E;
        LCHBF.reset = !RESETn;

        RCHAF.clk = E;
        RCHAF.reset = !RESETn;

        RCHBF.clk = E;
        RCHBF.reset = !RESETn;
        LPG.clk = E;
        LPG.reset = !RESETn;
```

```
RPG.clk = E;
RPG.reset = !RESETn;

LTCHF.clk = E;
LTCHF.reset = !RESETn;

% Left direction flip-flop uses left channel B to clock in left channel A.
   This is used to determine direction %
LDIR.clk = DFLCHB;
LDIR.clrn = RESETn;
LDIR.d = DFLCHA;

% Right direction flip-flop uses right channel A to clock in right channel B.
   This is used to determine direction %
RDIR.clk = DFRCHA;
RDIR.clrn = RESETn;
RDIR.d = DFRCHB;

% The counter reset state machine %
CRST.clk = E;
CRST.reset = !RESETn;

% The motor tick counters count pulses generated by the state transition of
   the quadrature encoders %

% Tick counter for left motor %
LCNT[].clk = LTP;
LCNT[].clrn = CRESETn;
% Count up if rotating forward %
IF (LDIR.q == 0) THEN
        LCNT[].d = LCNT[].q + 1;
% Count down if rotating backwards %
ELSE
        LCNT[].d = LCNT[].q - 1;
END IF;

% Tick counter for right motor %
RCNT[].clk = RTP;
RCNT[].clrn = CRESETn;
% Count up if rotating forward %
IF (RDIR.q == 0) THEN
        RCNT[].d = RCNT[].q + 1;
% Count down if rotating backwards %
ELSE
        RCNT[].d = RCNT[].q - 1;
END IF;
```

```
% Counter are latched into speed registers. Latch for the speed registers is derived
   from the LTCHclk %

% Left speed registers %
LSPD[].clk = DFLTCH;
LSPD[].clrn = RESETn;
LSPD[].d = LCNT[].q;

% Right speed registers %
RSPD[].clk = DFLTCH;
RSPD[].clrn = RESETn;
RSPD[].d = RCNT[].q;

% Table for the Counter reset state machine %
TABLE
CRST,     DFLTCH          =>    CRST,  CRESETn;
C0,         0             =>    C0,      1;
C0,         1             =>    C1,      1;
C1,         0             =>    C0,      1;
C1,         1             =>    C2,      1;
C2,         X             =>    C3,      0;
C3,         0             =>    C0,      1;
C3,         1             =>    C3,      1;
END TABLE;

% Table for the left channel A filter %
TABLE
LCHAF,     LCHA      =>    LCHAF,     DFLCHA;
LA10,        0       =>     LA10,        0;
LA10,        1       =>     LA20,        0;
LA20,        0       =>     LA10,        0;
LA20,        1       =>     LA11,        0;
LA11,        0       =>     LA21,        1;
LA11,        1       =>     LA11,        1;
LA21,        0       =>     LA10,        1;
LA21,        1       =>     LA11,        1;
END TABLE;
```

```
% Table for the left channel B filter %
TABLE
LCHBF,      LCHB        =>    LCHBF,      DFLCHB;
LB10,         0         =>     LB10,         0;
LB10,         1         =>     LB20,         0;
LB20,         0         =>     LB10,         0;
LB20,         1         =>     LB11,         0;
LB11,         0         =>     LB21,         1;
LB11,         1         =>     LB11,         1;
LB21,         0         =>     LB10,         1;
LB21,         1         =>     LB11,         1;
END TABLE;

% Table for the right channel A filter %
TABLE
RCHAF,      RCHA        =>    RCHAF,      DFRCHA;
RA10,         0         =>     RA10,         0;
RA10,         1         =>     RA20,         0;
RA20,         0         =>     RA10,         0;
RA20,         1         =>     RA11,         0;
RA11,         0         =>     RA21,         1;
RA11,         1         =>     RA11,         1;
RA21,         0         =>     RA10,         1;
RA21,         1         =>     RA11,         1;
END TABLE;

% Table for the right channel B filter %
TABLE
RCHBF,      RCHB        =>    RCHBF,      DFRCHB;
RB10,         0         =>     RB10,         0;
RB10,         1         =>     RB20,         0;
RB20,         0         =>     RB10,         0;
RB20,         1         =>     RB11,         0;
RB11,         0         =>     RB21,         1;
RB11,         1         =>     RB11,         1;
RB21,         0         =>     RB10,         1;
RB21,         1         =>     RB11,         1;
END TABLE;
```

```
% Table for the left state transition pulse generator %
TABLE
LPG,   DFLCHA,     DFLCHB      =>     LPG,   LTP;
LXX,       0,          0       =>     LXX,     0;
LXX,       0,          1       =>     LXnX,    1;
LXX,       1,          0       =>     LXnX,    1;
LXX,       1,          1       =>     LXX,     0;
LXnX,      0,          0       =>     LXX,     1;
LXnX,      0,          1       =>     LXnX,    0;
LXnX,      1,          0       =>     LXnX,    0;
LXnX,      1,          1       =>     LXX,     1;
END TABLE;

% Table for the right state transition pulse generator %
TABLE
RPG,   DFRCHA,     DFRCHB      =>     RPG,   RTP;
RXX,       0,          0       =>     RXX,     0;
RXX,       0,          1       =>     RXnX,    1;
RXX,       1,          0       =>     RXnX,    1;
RXX,       1,          1       =>     RXX,     0;
RXnX,      0,          0       =>     RXX,     1;
RXnX,      0,          1       =>     RXnX,    0;
RXnX,      1,          0       =>     RXnX,    0;
RXnX,      1,          1       =>     RXX,     1;
END TABLE;

% Table for the LTCHclk filter %
TABLE
LTCHF,        LTCHclk      =>     LTCHF,        DFLTCH;
LTCH10,         0          =>     LTCH10,         0;
LTCH10,         1          =>     LTCH20,         0;
LTCH20,         0          =>     LTCH10,         0;
LTCH20,         1          =>     LTCH11,         0;
LTCH11,         0          =>     LTCH21,         1;
LTCH11,         1          =>     LTCH11,         1;
LTCH21,         0          =>     LTCH10,         1;
LTCH21,         1          =>     LTCH11,         1;
END TABLE;
```

```
%
************************************************************************
%
% Address decoding and demultiplexing                                  %
% Latch address on rising edge of E clock                              %
%
************************************************************************
%

demux[15..8].d = PA[7..0];
demux[7..0].d = PB[7..0];
demux[15..0].clk = E;
A[15..1] = demux[15..1].q;              % Don't need LSB of address %

% Enable writes when E high and R/W low %

IF (E == VCC) & (R_W == GND) THEN
      WEn = GND;
ELSE
      WEn = VCC;
END IF;

% Enable reads when E high and R/W high %

IF (E == VCC) & (R_W == VCC) THEN
      OEn = GND;
ELSE
      OEn = VCC;
END IF;

% Access external memory when addresses in the range 0x1000 to 0x7fff %

IF (E==VCC) & ((demux[15..0].q >= H"1000") & (demux[15..0].q <= H"7fff")) THEN
         CS_MEM = VCC;
ELSE
         CS_MEM = GND;
END IF;

% Access even memory locations when address is even %

IF (CS_MEM == VCC) & (demux0.q == GND) THEN
         CS_MEM_EVENn = GND;
ELSE
         CS_MEM_EVENn = VCC;
END IF;
```

```
% Access odd memory locations when LSTRBn is low %

IF (CS_MEM == VCC) & (LSTRBn == GND) THEN
        CS_MEM_ODDn = GND;
ELSE
        CS_MEM_ODDn = VCC;
END IF;

% A read from address 0x0400 reads the Left Motor Tick Counter %
% A read from address 0x0402 reads the Right Motor Tick Counter %
% A read from address 0x0404 reads the Left Motor Speed Register %
% A read from address 0x0406 reads the Right Motor Speed Register %

IF (demux[15..0].q == H"0400") # (demux[15..0].q == H"0401") THEN
        % Sign extend left count register to 16 bits %
        IF(LDIR.q == 0) THEN
                IDB[15..9] = B"0000000";
        ELSE
                IDB[15..9] = B"1111111";
        END IF;
        IDB[8..0] = LCNT[8..0].q;
ELSIF (demux[15..0].q == H"0402") # (demux[15..0].q == H"0403") THEN
        % Sign extend right count register to 16 bits %
        IF(RDIR.q == 0) THEN
                IDB[15..9] = B"0000000";
        ELSE
                IDB[15..9] = B"1111111";
        END IF;
        IDB[8..0] = RCNT[8..0].q;

ELSIF (demux[15..0].q == H"0404") # (demux[15..0].q == H"0405") THEN
        % Sign extend left speed register to 16 bits %
        IF(LDIR.q == 0) THEN
                IDB[15..9] = B"0000000";
        ELSE
                IDB[15..9] = B"1111111";
        END IF;
        IDB[8..0] = LSPD[8..0].q;
ELSIF (demux[15..0].q == H"0406") # (demux[15..0].q == H"0407") THEN
        % Sign extend right speed register to 16 bits %
        IF(RDIR.q == 0) THEN
                IDB[15..9] = B"0000000";
        ELSE
                IDB[15..9] = B"1111111";
        END IF;
        IDB[8..0] = RSPD[8..0].q;
ELSE
        IDB[15..0] = H"0000";
END IF;
```

```
%  If reading from address 0x0400, 0x0402, 0x0404, or 0x0406, we need to drive the %
%  Port A lines when E is high %

IF (R_W==VCC) & (E == VCC) &
        ((demux[15..0].q == H"0400") # (demux[15..0].q == H"402")
        # (demux[15..0].q == H"404") # (demux[15..0].q == H"406"))
THEN
        PA_OE = VCC;
ELSE
       PA_OE = GND;
END IF;

%  If reading from address 0x0400 - 0x0407, we need to drive the Port B lines when E is
     high %

IF  ((R_W == VCC) & (E == VCC) & (LSTRBn == GND))
                        & ((demux[15..0].q == H"0400") # (demux[15..0].q == H"0401")
                        # (demux[15..0].q == H"402") # (demux[15..0].q == H"403")
                        # (demux[15..0].q == H"404") # (demux[15..0].q == H"405")
                        # (demux[15..0].q == H"406") # (demux[15..0].q == H"407"))
THEN
        PB_OE = VCC;
ELSE
        PB_OE = GND;
END IF;

%  Here's where we put the internal data bus values onto Port A %

PA[7] = TRI(IDB[15], PA_OE);
PA[6] = TRI(IDB[14], PA_OE);
PA[5] = TRI(IDB[13], PA_OE);
PA[4] = TRI(IDB[12], PA_OE);
PA[3] = TRI(IDB[11], PA_OE);
PA[2] = TRI(IDB[10], PA_OE);
PA[1] = TRI(IDB[9],  PA_OE);
PA[0] = TRI(IDB[8],  PA_OE);

%  Here's where we put the internal data bus values onto Port B %

PB[7] = TRI(IDB[7], PB_OE);
PB[6] = TRI(IDB[6], PB_OE);
PB[5] = TRI(IDB[5], PB_OE);
PB[4] = TRI(IDB[4], PB_OE);
PB[3] = TRI(IDB[3], PB_OE);
PB[2] = TRI(IDB[2], PB_OE);
PB[1] = TRI(IDB[1], PB_OE);
PB[0] = TRI(IDB[0], PB_OE);
```

```
% We have to do the following to have pins mapped correctly %
XX.clk = E;
XX.d = VCC;
XX.clrn  = !RESETn;

END;
```

Appendix C: HC12 Code

```
/* This is the HC12 header file. It contains definitions of pointers to various registers. */
#ifndef __HC12_H
#define __HC12_H    1
#define _BASE    0x0000

#define LCNT   (* (int *)(_BASE+0x400))
#define RCNT   (* (int *)(_BASE+0x402))
#define LSPD   (* (int *)(_BASE+0x404))
#define RSPD   (* (int *)(_BASE+0x406))

#define RTICTL (* (unsigned char *)(_BASE+0x14))
#define RTIFLG (* (unsigned char *)(_BASE+0x15))
#define INTCR  (* (unsigned char *)(_BASE+0x1e))

#define PWCLK  (* (unsigned char *)(_BASE+0x40))
#define PWPOL  (* (unsigned char *)(_BASE+0x41))
#define PWEN   (* (unsigned char *)(_BASE+0x42))
#define PWSCAL1 (* (unsigned char *)(_BASE+0x46))
#define PWPER0 (* (unsigned char *)(_BASE+0x4c))
#define PWPER1 (* (unsigned char *)(_BASE+0x4d))
#define PWPER2 (* (unsigned char *)(_BASE+0x4e))
#define PWDTY0 (* (unsigned char *)(_BASE+0x50))
#define PWDTY1 (* (unsigned char *)(_BASE+0x51))
#define PWDTY2 (* (unsigned char *)(_BASE+0x52))
#define PWCTL  (* (unsigned char *)(_BASE+0x54))
#define PORTP  (* (unsigned char *)(_BASE+0x56))
#define DDRP   (* (unsigned char *)(_BASE+0x57))

#define ATDCTL2 (* (unsigned char *)(_BASE+0x62))
#define ATDCTL4 (* (unsigned char *)(_BASE+0x64))
#define ATDCTL5 (* (unsigned char *)(_BASE+0x65))
#define ADR2H   (* (unsigned char *)(_BASE+0x74))
#define ADR3H   (* (unsigned char *)(_BASE+0x76))
#define ADR4H   (* (unsigned char *)(_BASE+0x78))
#define ADR5H   (* (unsigned char *)(_BASE+0x7a))
#define ADR6H   (* (unsigned char *)(_BASE+0x7c))
#define ADR7H   (* (unsigned char *)(_BASE+0x7e))

#define PACTL  (* (unsigned char *)(_BASE+0xa0))
#define PAFLG  (* (unsigned char *)(_BASE+0xa1))
#define PACNT  (* (unsigned int  *)(_BASE+0xa2))

#define PORTDLC (* (unsigned char *)(_BASE+0xfe))
#define DDRDLC  (* (unsigned char *)(_BASE+0xff))

#endif
```

```
/* This is the setup header file. This file contains functions that setup various subsystems
   of the robot */

#include <hc12.h>

/* This function performs basic setup of the PWM system */
void PWM_setup()
{
  // Choose 8-bit mode
  PWCLK = PWCLK & ~0xC0;
  // Choose left-aligned
  PWCTL = PWCTL & ~0x08;
  // Choose high during duty cycle polarity
  PWPOL = PWPOL | 0x0F;
}

/* This function sets up the latch for the digital speed decoders in Altera. The latch
   frequency is about 198.4 Hz. */
void LATCH_setup()
{
  // Choose clock mode for channel 2
  PWPOL = PWPOL | 0x40;

  // Select N = 5 and M = 2
  PWCLK |= 0x05;
  PWSCAL1 = 2;

  // Select period of 210 and duty of 105
  PWPER2 = 209;
  PWDTY2 = 104;

  PWEN |= 0x04;        // Enable PWM Channel 2
}
```

```
/* This function sets up the H-Bridge PWM and direction signals */
void HB_setup()
{
  // Select clock mode 1 for Channels 1 and 0
  PWPOL = PWPOL & ~0x30;
  // Select N = 4 for Channels 1 and 0
  PWCLK = (PWCLK & ~0x18) | 0x20;

  // Select period of 200 for Channels 1 and 0
  PWPER1 = 199;
  PWPER0 = 199;

  PWEN |= 0x03;        //Enable PWM Channels 1 & 0

  // Enable Direction Bits for Output
  DDRP |= 0xC0;

  // Initialize the H-bridge duty cycles to 0
  PWDTY1 = 0;
  PWDTY0 = 0;
}

/* This function sets up the A/D Port. Four distance sensors, two on the left & two
    on the right, and two flame sensors plug into the A/D Port. */
void AD_setup()
{
  ATDCTL2 = 0x80;   //Power up A/D
  ATDCTL4 = 0x01;   //Select A/D clock
  ATDCTL5 = 0x40;   //8-channel mode

  //Scan continuously on all channels
  ATDCTL5 |= 0x30;
}

/* This function sets up bits 0-2 of Port DLC as inputs. The Whiteline sensor, Frontwall
    sensor, and Tone Decoder plug into Port DLC. */
void DLC_setup()
{
  DDRDLC &= ~0x07;
}
```

```
/* This is the functions header file */

#include <hc12.h>

#define forward 0
#define backwards 1
#define left 2
#define right 3

#define Small_Room 0
#define Large_Room 1

signed int desired_speed, ldesired_speed, rdesired_speed;
signed int desired_dist, desired_orient;

signed int ldty, rdty;
signed int lserr, rserr, curr_orient;

signed int fsum, fdiff;
unsigned char fire = 0, fireout = 0;
unsigned char flame =0, flameh = 0, flamegh = 0;

unsigned char ws = 0, wlc = 0, InRoom = 0, room;
unsigned char room_size, room_start;
unsigned int dist_sum;
unsigned char smallest_dist, smallest_turns;

unsigned char hallway_nav_complete = 0;
unsigned char leftwall_nav_complete = 0;
unsigned char room_exited = 0;

unsigned int counter;
unsigned char i, cc;
unsigned int desired_ticks, lticks, rticks;

/* This function enables the RTI interrupt */
void enableRTI()
{
  RTICTL = 0x84;       // Enable RTI for 8ms
  RTIFLG = 0x80;       // Clear RTI flag
  _asm("cli");         // Clear I bit in CCR
}

/* This function disables the RTI interrupt */
void disableRTI()
{
  RTICTL = 0x00;    // Clear RTI Control Register
  _asm("sei");      // Set I bit in CCR
}
```

```
/* This is a delay function */
void delay(int num)
{
  int y = num;
  int x;

  while(y > 0)
  {
    // Inner loop delays for 1 ms
    x = 1333;
    while (x > 0)
    {
      x = x - 1;
    }
    y = y - 1;
  }
}

/* This function returns a 1 if there is a left wall and a 0 if not */
unsigned char LeftWall()
{
  return ((ADR2H > 30) && (ADR3H > 30));
}

/* This function returns a 1 if there is a right wall and a 0 if not */
unsigned char RightWall()
{
  return ((ADR6H > 30) && (ADR7H > 30));
}

/* This function returns a 1 if there is a front wall and a 0 if not */
unsigned char FrontWall()
{
  return ((PORTDLC & 0x02) == 0x02);
}

/* This function returns a 1 if there is a white  surface and a 0 if not */
unsigned char WhiteSurface()
{
  return ((PORTDLC & 0x01) == 0x01);
}

/* This function returns a 1 if the tone detector has been triggered and a 0 if not */
unsigned char Tone()
{
  return ((PORTDLC & 0x04) == 0x00);
}
```

```
/* This function returns a 1 if there is a flame present and a 0 if not */
unsigned char AFlame()
{
  return ((ADR4H > 40) || (ADR5H > 40));
}

/* This function returns a 1 if the robot is about parallel to a left wall and a 0 if
     it is not */
unsigned char LInRange()
{
  unsigned char r;
  if(((ADR2H < desired_dist - 10) && (ADR3H < desired_dist - 10))
        || ((ADR2H > desired_dist) && (ADR3H > desired_dist)))
    r = 1;
  else
    r = 0;

  return r;
}

/* This function returns a 1 if the robot is about parallel to a right wall and a 0 if
     it is not */
unsigned char RInRange()
{
  unsigned char r;
  if(((ADR6H < desired_dist - 10) && (ADR7H < desired_dist - 10))
        || ((ADR6H > desired_dist) && (ADR7H > desired_dist)))
    r = 1;
  else
    r = 0;

  return r;
}

/* This is the error and exit function. This function is called when everything hits
     the fan. */
void error()
{
  // Set H-bridge duty cycles to 0
  PWDTY1 = 0;
  PWDTY0 = 0;
  // Exit and return to DBug-12
  exit(0);
}
```

```
/* This function waits for a tone to be detected and then undetected before exiting */
void Wait_for_Tone()
{
  while(!Tone()) {}      //Wait until tone is detected
  delay(200);
  while(Tone()) {}       //Wait until tone is undetected
}

/* This function counts white lines */
void Count_Whitelines()
{
  // Check for White Surface
  if(WhiteSurface())
    ws = 1;

  // If a white surface has been seen and now the surface is black, count whiteline
  if(!WhiteSurface() && ws)
  {
    wlc++;
    ws = 0;
  }
}

/* This function sets the direction of the motors */
void Set_Direction(unsigned char dir)
{
  if(dir == forward)
    PORTP = (PORTP & ~0x80) | 0x40;
  if(dir == backwards)
    PORTP = (PORTP & ~0x40) | 0x80;
  if(dir == left)
    PORTP |= 0xC0;
  if(dir == right)
    PORTP &= ~0xC0;
}

/* This function sets the duty cycle of the H-bride
   PWM signals */
void Set_DTY(unsigned char LDTY, unsigned char RDTY)
{
  PWDTY1 = RDTY;
  PWDTY0 = LDTY;
}
```

```
/* This function performs a Hamamatsu scan */
void Ham_Scan()
{
  // Set RTI condition code to 128. Reset RTI counter and pulse accumulator
  cc = 128;
  counter = 0;
  PACNT = 0;

  // Wait until flame has been detected or counter has reached 400.
  while(!flameh && (counter < 400))
  {
    if(PACNT >= 1)
    {
      flameh = 1;
      PORTP |= 0x10;
    }
    else
    {
      flameh = 0;
      PORTP &= ~0x10;
    }
  }
  PACNT = 0;
}

/* This function turns the fan off */
void fan_off()
{
  PORTP = PORTP & ~0x20;//Make bit 5 of Port P low
  DDRP &= ~0x20;              //Disable bit 5 of Port P
}

/* This function turns the fan on */
void fan_on()
{
  DDRP |= 0x20;         //Enable bit 5 of Port P
  PORTP |= 0x20;        //Make bit 5 of Port P high
}
```

```
/* This function sets up closed-loop speed control */
void go(signed int speed)
{
  desired_speed = speed;  // Set desired speed

  cc = 0;          //Set RTI condition code to 0

  // Set Direction
  if(desired_speed >= 0)
    Set_Direction(forward);
  else
    Set_Direction(backwards);

  // Set initial duty cycles to the desired speed
  Set_DTY((unsigned char)desired_speed, (unsigned char)desired_speed);
}

/* This function sets up closed-loop speed control with left wall following */
void golw(signed int speed, signed char dist, signed char orient)
{
  // Only go forward when wall following
  if(speed >= 0)
    desired_speed = speed;
  else
    desired_speed = -speed;

  // Set desired speed for left and right motors
  ldesired_speed = desired_speed;
  rdesired_speed = desired_speed;

  // Set desired distance and orientation from wall
  desired_dist = dist;
  desired_orient = orient;

  cc = 1;          // Set RTI condition code to 1
  i = 0;  // Reset i

  Set_Direction(forward);       //Go Forward

  // Set initial duty cycles to the desired speeds
  Set_DTY((unsigned char)ldesired_speed, (unsigned char)rdesired_speed);//Set Duty
Cycle
}
```

```
/* This function sets up closed-loop speed control with right wall following */
void gorw(signed int speed, signed char dist, signed char orient)
{
  // Only go forward when wall following
  if(speed >= 0)
    desired_speed = speed;
  else
    desired_speed = -speed;

  // Set desired speed for left and right motors
  ldesired_speed = desired_speed;
  rdesired_speed = desired_speed;

  // Set diesired distance and orientation to wall
  desired_dist = dist;
  desired_orient = orient;

  cc = 2;           // Set condition code to 2
  i = 0;  // Reset i

  Set_Direction(forward);       //Go forward

  // Set initial duty cycles to the desired speeds
  Set_DTY((unsigned char)ldesired_speed, (unsigned char)rdesired_speed);
}
```

```
/* This sets up closed-loop speed control with left wall following while performing a
   hamamatsu scan */
void goscan()
{
 // Set desired speed to 60
 desired_speed = 60;
 ldesired_speed = desired_speed;
 rdesired_speed = desired_speed;

 // Set desired distance from the wall to 80.
 desired_dist = 80;
 // Set desired orientation to wall to 0.
 desired_orient = 0;

 PACNT = 0;  // Reset pulse accumulator
 cc = 1;          // Set RTI condition code to 0
 counter = 0;   // Reset RTI counter

 Set_Direction(forward);       // Go forward

 // Set initial duty cycles to the desired speed
 Set_DTY((unsigned char)desired_speed, (unsigned char)desired_speed);

 // Wait until a flame has been detected, a front wall, or the counter has reached 100
 while(!FrontWall() && !flamegh && (counter <= 100))
 {
  if(PACNT > 0)
    flamegh = 1;
 }
}
```

```
// This function turns in place 45 degrees */
void tip45(unsigned int dir)
{
  // Set desired speed to 100
  ldesired_speed = 100;
  rdesired_speed = 100;

  // Set desired number of motor ticks to 600
  desired_ticks = 600;

  // Set direction
  Set_Direction(dir);
  // Set initial duty cycles to the desired speeds
  Set_DTY((unsigned char)ldesired_speed, (unsigned char)rdesired_speed);

  if(dir == left)
  {
    // Negate desired left motor speed
    ldesired_speed = -ldesired_speed;
    rticks = 0;    // Reset right motor tick counter
    cc = 3;        // Set RTI condition code to 3

    // Turn in place left by 45 degrees
    while(rticks <= desired_ticks)
    {
      if(rticks > desired_ticks - 1)
        break;
    }
  }

  if(dir == right)
  {
    // Negate desired right motor speed
    rdesired_speed = -rdesired_speed;
    lticks = 0;    // Reset left motor tick counter
    cc = 4;        //Set RTI condition code to 4

    // Turn in place right by 45 degrees
    while(lticks <= desired_ticks)
    {
      if(lticks > desired_ticks - 1)
        break;
    }
  }
```

```
  // Set duty cycles and desired speeds to 0
  Set_DTY(0, 0);
  ldesired_speed = 0;
  rdesired_speed = 0;
}

/* This function turns in place 90 degrees */
void tip90(unsigned int dir)
{
  // Set desired speeds to 100
  ldesired_speed = 100;
  rdesired_speed = 100;

  // Set desired number of ticks to 1300
  desired_ticks = 1300;

  // Set direction
  Set_Direction(dir);
  // Set initial duty cycles to the desired speeds
  Set_DTY((unsigned char)ldesired_speed, (unsigned char)rdesired_speed);

  if(dir == left)
  {
    // Negate desired left motor speed
    ldesired_speed = -ldesired_speed;
    rticks = 0; // Reset right motor tick counter
    cc = 3;         // Set RTI condition code to 3

    // Turn in place left by 90 degrees
    while(rticks <= desired_ticks)
    {
      if(rticks > desired_ticks - 1)
        break;
    }
  }
  if(dir == right)
  {
    // Negate desired right motor speed
    rdesired_speed = -rdesired_speed;
    lticks = 0; // Reset left motor tick counter
    cc = 4;         // Set RTI condition code to 4

    // Turn in place right by 90 degrees
    while(lticks <= desired_ticks)
    {
      if(lticks > desired_ticks - 1)
        break;
    }
  }
```

```
  // Set duty cycles and desired speeds to 0
  Set_DTY(0, 0);
  ldesired_speed = 0;
  rdesired_speed = 0;
}

/* This function turns in place 180 degrees */
void tip180()
{
  // Set desired speeds to 100
  ldesired_speed = 100;
  rdesired_speed = 100;

  // Set desired number of ticks to 3500
  desired_ticks = 3500;

  Set_Direction(right);// Turn right
  // Set initial duty cycles to the desired speeds
  Set_DTY((unsigned char)ldesired_speed, (unsigned char)rdesired_speed);

  // Negate desired right motor speed
  rdesired_speed = -rdesired_speed;
  lticks = 0;      // Reset left motor tick counter
  cc = 4;          // Set RTI condition code to 4

  // Turn in place right by 180 degrees
  while(lticks <= desired_ticks)
  {
    if(lticks > desired_ticks - 1)
      break;
  }

  // Set duty cycles and desired speeds to 0
  Set_DTY(0, 0);
  ldesired_speed = 0;
  rdesired_speed = 0;
}
```

```
/* This function orients the robot parallel to the left wall */
void orient_left()
{
  // Set desired speeds to 60
  ldesired_speed = 60;
  rdesired_speed = 60;

  // Set inital duty cycle to desired speeds
  Set_DTY((unsigned char)ldesired_speed, (unsigned char)rdesired_speed);


  if(ADR2H > ADR3H)
  {
    Set_Direction(right);          // Turn right
    // Negate desired right motor speed
    rdesired_speed = -rdesired_speed;
    cc = 4;         // Set RTI condition code to 4

    // Turn right until almost parallel to wall
    while(ADR2H > ADR3H + 25);
  }
  else
  {
    Set_Direction(left); // Turn left
    // Negate diesired left motor speed
    ldesired_speed = -ldesired_speed;
    cc = 3;         // Set RTI condition code to 3

    // Turn left until almost parallel to wall
    while(ADR3H > ADR2H + 25);
  }

  // Set duty cycles and desired speeds to 0
  Set_DTY(0, 0);
  ldesired_speed = 0;
  rdesired_speed = 0;
}
```

```
/* This function orients the robot parallel to the right wall */
void orient_right()
{
  // Set desired speeds to 60
  ldesired_speed = 60;
  rdesired_speed = 60;

  // Set initial duty cycles to desired speeds
  Set_DTY((unsigned char)ldesired_speed, (unsigned char)rdesired_speed);

  if(ADR7H > ADR6H)
  {
    Set_Direction(right);          // Turn right
    // Negate desired right motor speed
    rdesired_speed = -rdesired_speed;
    cc = 4;         // Set RTI condition code to 4

    // Turn until almost parallel to wall
    while(ADR7H > ADR6H + 25);
  }
  else
  {
    Set_Direction(left); // Turn left
    // Negate desired left motor speed
    ldesired_speed = -ldesired_speed;
    cc = 3;         // Set RTI condition code to 3

    // Turn until almost parallel to wall
    while(ADR6H > ADR7H + 25);
  }

  // Set duty cycles and desired speeds to 0
  Set_DTY(0, 0);
  ldesired_speed = 0;
  rdesired_speed = 0;
}
```

```
/* This function scan a room to see if robot is in a small or large room. It also checks if
   the robot is at the home circle. */
void Scan_Room()
{
 unsigned char c;

 // Initializations
 room_start = 128;
 room = 128;
 counter = 0;
 wlc = 0;

 // Check for a white surface
 if(!WhiteSurface())
 {
  smallest_dist = ADR2H;
  smallest_turns = 0;

  // Orient to nearest wall if one is present
  if((ADR2H >= 80) || (ADR3H >= 80))
   orient_left();
  if((ADR6H >= 80) || (ADR7H >= 80))
   orient_right();

  // Scan to find nearest left wall.
  for(c = 0; c < 8; c++)
  {
   // Make eight left 45 degree turns to find the nearest wall.
   tip45(left);
   delay(100);
   // Store the smallest distance and number of turns to face robot towards the wall that
   // is the closest
   if(smallest_dist < ADR2H)
   {
    smallest_dist = ADR2H;
    smallest_turns = c;
   }
  }
  Set_DTY(0, 0);       // Set duty cycles to 0

  // Turn the robot until it is facing the closest left wall
  for(c = 0; c < smallest_turns; c++)
  {
   tip45(left);
   delay(100);
  }
```

```
go(30);        // Go towards the nearest wall
while((ADR2H < 80) && (ADR3H < 60) && (ADR6H < 80) && !FrontWall() &&
          !WhiteSurface());

// If there is a white surface, turn around and left wall follow
if(WhiteSurface())
{
  tip180();                // Turn 180 degrees
  delay(100);
  golw(60, 100, 0);    // Left Wall Follow
  // Left wall follow until there is a front wall and turn right 90 degrees
  while(!FrontWall())
  tip90(right);
  delay(100);
  // Orient parallel to left wall
  orient_left();
}

// If left rear sensor sees a wall and there is no front wall, turn left 45 degrees
if((ADR3H >= 60) && !FrontWall())
  tip45(left);

// If right front sensor sees a wall and there is no front wall, turn around
if((ADR6H >= 80) && !FrontWall())
{
  tip180();              // Turn 180 degrees
  delay(100);
  // Orient parallel to left wall
  orient_left();
}

// If left front sensor sees a wall and there is no front wall, orient parallel to left wall
if((ADR2H >= 80) && !FrontWall())
  orient_left();

// If there is front wall, turn right 90 degrees
if(FrontWall())
{
  tip90(right);
  delay(100);
}

counter = 0; // Reset RTI counter
golw(60, 100, 0);    // Left wall follow
```

```
// Left wall follow until there is a front wall or counter is 100. Count white lines while
// doing this.
while(!FrontWall() && (counter < 100))
  Count_Whitelines();

// If there is a front wall, turn right 90 degrees and then left wall follow
if(FrontWall())
{
  tip90(right);
  delay(100);
  golw(60, 100, 0);
}

// Continue left wall following until there is no left wall. Count white line while doing
// this.
while(LeftWall())
{
  Count_Whitelines();

  // If there is a front wall, turn right 90 degrees, orient robot parallel to left wall, and
  // continue left wall following
  if(FrontWall())
  {
    tip90(right);
    delay(100);
    golw(60, 100, 0);
    delay(200);
  }
}

// After left wall has disappeared, go forward until there is a front wall.
go(30);
while(!FrontWall());
go(0);

dist_sum = 0;

// If no white lines have been counted, scan room to determine its size
if(wlc == 0)
{
  // Make eight 45 degree turns, summing up the the values from the distance sensors
  for(c = 0; c < 8; c++)
  {
    tip45(right);
    delay(100);
    dist_sum = dist_sum + ADR2H + ADR3H + ADR6H + ADR7H;
  }
  Set_DTY(0, 0);     // Set duty cycles to 0
```

```
      // Determine room size
      if(dist_sum > 1700)
        room_size = Small_Room;
      else
        room_size = Large_Room;
    }
    // If a white line has been counted, then robot has started from room 4
    else
    {
      room_start = 4;
      room = 4;
    }
  }
  // If a white surface was detected initially, then robot has started from the home circle
  else
  {
    room = 0;
    room_start = 0;
    orient_left();
  }
}

/* This function exits the room after the room scan and after putting out the flame */
void Exit_Room()
{
  // If fire is not out, then exiting room after performing room scan
  if(!fireout)
  {
    tip90(left);  // Turn left 90 degrees
    delay(100);

    // Oreint parallel to right wall
    orient_right();
    delay(100);

    // If not in room 4, go forward until there is a front wall
    if(room_start != 4)
    {
      // Perform final check on the size of room if it is a large room by checking the value
      // from the left rear sensor
      if((room_size == Large_Room) && (ADR3H > 25))
           room_size = Small_Room;

      // Right wall follow out of the room
      gorw(50, 100, 0);
      while(!FrontWall() && RightWall());
```

```
  // If no front wall and no right wall, go straight until there is a front wall
  if(!FrontWall() && !RightWall())
  {
   delay(200);
   go(40);
   while(!FrontWall());
  }
  go(0);

  // If there is no right wall, then robot has started in room 1
  if(!RightWall())
  {
   room_start = 1;
   room = 1;
  }
  else
  {
      // If there is a right wall and the room is small, then robot has started in
      // room 3
   if(room_size == Small_Room)
   {
    room_start = 3;
    room = 3;
   }
   // If there is a right wall and the room is large, then robot has started in room 4
   else
   {
      room_start = 2;
      room = 2;
   }
  }


  // Turn left 90 degrees and orient parallel to right wall
  tip90(left);
  orient_right();
 }
}
```

```
// Exit room after outing flame
else
{
  // Procedures for exiting rooms 1 and 2
  if(room < 3)
  {
    // Backup
    go(-40);
    delay(200);

    tip45(right);          // Turn right 45 degrees
    delay(100);

    // Go towards a wall
    go(40);
    while((ADR2H < 80) && (ADR6H < 80) && !FrontWall());

    // If there is a front wall, turn right 90 degrees and orient parallel to left wall
    if(FrontWall())
    {
        tip90(right);
        delay(100);
        orient_left();
    }

    // If right front sensor sees wall and left front sensor sees no wall and there is no front
    // wall, turn robot around and orient parallel to left wall
    if((ADR6H >= 80) && (ADR2H < 80) && !FrontWall())
    {
      orient_right();
      tip180();
      delay(100);
      orient_left();
    }

    // If left front sensor sees wall and there is no front wall, orient parallel to left wall
    if((ADR2H >= 80) && !FrontWall())
      orient_left();


    // Left wall follow
    golw(50, 100, 0);
    delay(200);
```

```
// If no left wall, go forward to a left wall
if(!LeftWall())
{
     go(40);
     while((ADR2H < 80) && !FrontWall());

     // If there is front wall, turn right 90 degrees and orient parallel to left wall
     if(FrontWall())
     {
       tip90(right);
       orient_left();
     }

     // If left front sensor sees a wall, orient parallel to left wall
     if(ADR2H >= 80)
       orient_left();
}

// Left wall follow until there is no left wall
while(LeftWall())
{
     // If there is a front wall, turn right 90 degrees, oreint parallel to left wall, and
     // continue left wall following
  if(FrontWall())
  {
    tip90(right);
    delay(100);
    golw(50, 100, 0);
    delay(200);
  }
}

// After left wall has disappeared, go straight until there is a front wall
go(40);
while(!FrontWall());
go(0);

// Once a front wall is seen, turn left 90 degrees and orient parallel to right wall
tip90(left);
delay(100);
orient_right();

// Go straight out of room until there is a front wall
gorw(50, 100, 0);
while(!FrontWall() && RightWall());
```

```
  // If no front wall and no right wall, go straight until there is a front wall
  if(!FrontWall() && !RightWall())
  {
    go(40);
    while(!FrontWall());
  }
  go(0);

  // Turn left 90 degrees and orient parallel to right wall
  tip90(left);
  orient_right();
}
// Procedures for exiting rooms 3 and 4
else
{
  // Procedures for exiting room 3
  if(room == 3)
  {
    // Turn 45 degrees and go towards a wall
      tip45(right);
      delay(100);
      go(30);

      while((ADR2H < 80) && !FrontWall());

      // If there is front wall, turn right 90 degrees and orient parallel to the left wall
      if(FrontWall())
      {
        tip90(right);
        delay(100);
        orient_left();
      }

      // If left fron sensor sees a wall, orient parallel to left wall
      if(ADR2H >= 80)
        orient_left();
```

```
      // Left wall follow until there is no left wall
      golw(40, 100, 0);
      while(ADR2H >= 50)
      {
        // If there is a front wall, turn right 90 degrees and continue left wall following
        if(FrontWall())
        {
          tip90(right);
          delay(100);
          golw(40, 100, 0);
          delay(200);
        }
      }

      // After left wall has disappeared, go straight until there is a front wall
      go(40);
      while(!FrontWall());
      go(0);

      // Turn left 90 degrees and orient parallel to right wall
      tip90(left);
      delay(100);
      orient_right();

      // Go straight out of room until there is a front wall
      gorw(40, 100, 0);
      while(!FrontWall());

      // Turn left 90 degrees and orient parallel to the right wall
      tip90(left);
      delay(100);
      orient_right();
}

// Procedures for exiting room 4
else
{
      // Turn left 45 degrees and go towards a wall
      tip45(left);
      delay(100);
      go(30);

      while((ADR6H < 80) && !FrontWall());
```

```
        // If there is a front wall, turn left 90 degrees and orient parallel to right wall
        if(FrontWall())
        {
          tip90(left);
          delay(100);
          orient_right();
        }

        // If right front sensor sees a wall, orient parallel to right wall
        if(ADR2H >= 80)
          orient_right();

        // Right wall follow until there is no right wall
        gorw(30, 100, 0);
        while(ADR6H >= 50)
        {
          // If there is a front wall, turn left 90 degrees and continue right wall following
       if(FrontWall())
       {
        tip90(left);
        delay(100);
        gorw(30, 100, 0);
        delay(200);
       }
      }

      // Once right wall has disappeared, go straight until there is front wall
      go(30);
      while(!FrontWall());
      go(0);

      // Turn right 90 degrees and orient parallel to left wall
      tip90(right);
      delay(100);
      orient_left();

      // Go straight out of room until there is a front wall
      go(30);
      while(!FrontWall());

      // Turn right 90 degrees and orient parallel to left wall
      tip90(right);
      delay(100);
      orient_left();
    }
   }
  }
}
```

```
/* This function sets up flame follow */
void Flame_Follow()
{
  Set_Direction(forward);       // Go forward
  while(!fire && !room_exited && AFlame())
  {
    cc = 5;         // Set RTI condition code to 5

    if(FrontWall())
    {
      // If there is a front wall and a right wall, turn left 90 degrees and orient parallel to
      // right wall
      if(RightWall())
      {
        tip90(left);
            orient_right();
      }
      // If there is a front wall and a left wall, turn rightt 90 degrees and orient parallel to
      // left wall
      else
      {
        tip90(right);
            orient_left();
      }

      delay(100);
      Set_Direction(forward);    // Go forward
    }
  }
}
```

```
/* This function extinguishes the flame */
void Flame_Out()
{
  if(fire)
  {
    // Turn fan on
    fan_on();
    delay(500);

    // Set desired speeds to 40
    ldesired_speed = 40;
    rdesired_speed = 40;
    // Set desired ticks to 1000
    desired_ticks = 1000;

    Set_Direction(right);          // Turn right
    // Set initial duty cycles to the desired speeds
    Set_DTY((unsigned char)ldesired_speed, (unsigned char)rdesired_speed);

    // Negate desired right motor speed
    rdesired_speed = -rdesired_speed;
    lticks = 0;    // Set left motor tick counter to 0
    cc = 4;        // Set RTI condition code to 4

    // Turn in place right
    while(lticks <= desired_ticks)
    {
      if(lticks > desired_ticks - 1)
        break;
    }

    // Set desired speeds to 40
    ldesired_speed = 40;
    rdesired_speed = 40;
    // Set desired number of ticks to 2000
    desired_ticks = 2000;

    Set_Direction(left); // Turn left
    // Set duty cycles to desired speeds
    Set_DTY((unsigned char)ldesired_speed, (unsigned char)rdesired_speed);

    // Negate desired left motor speed
    ldesired_speed = -ldesired_speed;
    rticks = 0;    // Reset right motor tick counter
    cc = 3;        //Set RTI condition code to 3
```

```
    // Turn in place left
    while(rticks <= desired_ticks)
    {
      if(rticks > desired_ticks - 1)
        break;
    }

    // Set desired speeds to 40
    ldesired_speed = 40;
    rdesired_speed = 40;
    // Set desired number of ticks to 1000
    desired_ticks = 1000;

    Set_Direction(right);          // Turn right
    // Set duty cycles to desired speeds
    Set_DTY((unsigned char)ldesired_speed, (unsigned char)rdesired_speed);

    // Negate desired right motor speed
    rdesired_speed = -rdesired_speed;
    lticks = 0;    // Reset left motor tick counter
    cc = 4;        // Set RTI condition code to 4

    // Turn in place right
    while(lticks <= desired_ticks)
    {
      if(lticks > desired_ticks - 1)
        break;
    }

    // Set duty cycles and desired speeds to 0
    Set_DTY(0, 0);
    ldesired_speed = 0;
    rdesired_speed = 0;

    // Turn fan off
    fan_off();
    delay(300);
  }
}
```

```
/*  The RTI interrupt takes care of six functions based on the following condition codes:

    cc = 0 -- Closed-Loop Speed Control
    cc = 1 -- Closed-Loop Speed & Left Wall Follow Control
    cc = 2 -- Closed-Loop Speed & Right Wall Follow Control
    cc = 3 -- Turn Left in Place
    cc = 4 -- Turn Right in Place
    cc = 5 -- Flame Follow
*/
@interrupt void rti_isr()
{
 // Closed-loop speed control
 if(cc == 0)
 {
  // Calculate speed errors
  rserr = desired_speed - RSPD;
  lserr = desired_speed - LSPD;

  // Control equation for forward direction
  if(desired_speed >= 0)
  {
   rdty = rdty + rserr/4;
   ldty = ldty + lserr/4;
  }
  // Control equation for backwards direction
  else
  {
   rdty = rdty - rserr/4;
   ldty = ldty - lserr/4;
  }

  // Keep right motor duty cycle within 0 and 200
  if(rdty >= 199) rdty = 199;
  if(rdty <= 0) rdty = 0;

  // Keep left motor duty cycle within 0 and 200
  if(ldty >= 199) ldty = 199;
  if(ldty <= 0) ldty = 0;

  // Set duty cycle
  if(desired_speed == 0)
   Set_DTY(0, 0);
  else
   Set_DTY((unsigned char)ldty, (unsigned char)rdty);
 }
```

```
// Closed-loop speed & left wall following
if(cc == 1)
{
  // Calculate speed error
  rserr = rdesired_speed - RSPD;
  lserr = ldesired_speed - LSPD;

  // Adjust the left and right duty cycles
  rdty = ((9*rdty)/10) + rserr/3;
  ldty = ((9*ldty)/10) + lserr/3;

  // Keep right motor duty cycle within 0 and 200
  if(rdty >= 199) rdty = 199;
  if(rdty <= 19) rdty = 19;

  // Keep left motor duty cycle within 0 and 200
  if(ldty >= 199) ldty = 199;
  if(ldty <= 19) ldty = 19;

  // Set duty cycles
  Set_DTY((unsigned char)ldty, (unsigned char)rdty);

  // When i = 3, make wall following adjustments
  if(i == 3)
  {
    // Calculate orientation error
    curr_orient = ADR3H - ADR2H;

    // Adjust desired speeds to correct orientation
    rdesired_speed = desired_speed - (2*(desired_orient - curr_orient))/4;
    ldesired_speed = desired_speed + (2*(desired_orient - curr_orient))/4;

    // If oriented somewhat parallel, adjust distance
    if(LInRange())
    {
      rdesired_speed = rdesired_speed + (2*(desired_dist - (ADR3H)))/4;
      ldesired_speed = ldesired_speed - (2*(desired_dist - (ADR3H)))/4;
    }

    i = 0;        // Reset i
  }
}
```

```
// Closed-loop speed and right wall following
if(cc == 2)
{
  // Calculate speed error
  rserr = rdesired_speed - RSPD;
  lserr = ldesired_speed - LSPD;

  // Adjust duty cycles for speed error
  rdty = ((9*rdty)/10) + rserr;
  ldty = ((9*rdty)/10) + lserr;

  // Keep right motor duty cycle within 0 and 200
  if(rdty >= 199) rdty = 199;
  if(rdty <= 19) rdty = 19;

  // Keep left motor duty cycle within 0 and 200
  if(ldty >= 199) ldty = 199;
  if(ldty <= 19) ldty = 19;

  // Set duty cycles
  Set_DTY((unsigned char)ldty, (unsigned char)rdty);

  // When i = 3, make wall following adjustments
  if(i == 3)
  {
    // Calculate orientation error
    curr_orient = ADR6H - ADR7H;

    // Adjust desired speeds to correct orientation
    rdesired_speed = desired_speed - (2*(desired_orient - curr_orient))/4;
    ldesired_speed = desired_speed + (2*(desired_orient - curr_orient))/4;

    // If oriented somewhat parallel, adjust distance
    if(RInRange())
    {
      rdesired_speed = rdesired_speed - (3*(desired_dist - (ADR7H)))/8;
      ldesired_speed = ldesired_speed + (3*(desired_dist - (ADR7H)))/8;
    }

    i = 0;        // Reset i
  }
}
```

```
// Closed-loop speed control left turn in place
if(cc == 3)
{
  // Calculate speed errors
  rserr = rdesired_speed - RSPD;
  lserr = ldesired_speed - LSPD;

  // Adjust duty cycles to correct speed
  rdty = PWDTY1 + rserr;
  ldty = PWDTY0 - lserr;

  // Keep right motor duty cycle within 0 and 200
  if(rdty >= 199) rdty = 199;
  if(rdty <= 19) rdty = 19;

  // Keep left motor duty cycle within 0 and 200
  if(ldty >= 199) ldty = 199;
  if(ldty <= 19) ldty = 19;

  // Set duty cycles
  Set_DTY((unsigned char)ldty, (unsigned char)rdty);

  // Count right motor ticks
  if(RCNT >= 0)
    rticks = rticks + RCNT;
}
```

```
// Closed-loop speed control right turn in place
if(cc == 4)
{
  // Calculate speed error
  rserr = rdesired_speed - RSPD;
  lserr = ldesired_speed - LSPD;

  // Adjust duty cycles to correct speed
  rdty = PWDTY1 - rserr;
  ldty = PWDTY0 + lserr;

  // Keep right motor duty cycle within 0 and 200
  if(rdty >= 199) rdty = 199;
  if(rdty <= 19) rdty = 19;

  // Keep left motor duty cycle within 0 and 200
  if(ldty >= 199) ldty = 199;
  if(ldty <= 19) ldty = 19;

  // Set duty cycle
  Set_DTY((unsigned char)ldty, (unsigned char)rdty);

  if(LCNT >= 0)
    lticks = lticks + LCNT;
}
```

```
// Flame Follow
if(cc == 5)
{
  // Calculate flame intensity and orientation to flame
  fsum = ADR4H + ADR5H;
  fdiff = ADR4H - ADR5H;

  // If left wall and flame is to the left, follow left wall at a distance of 80
  if((ADR2H > 80) && (fdiff > 0))
  {
    rdty = 50 + (1*(80 - ADR3H)/8);
    ldty = 50 - (1*(80 - ADR3H)/8);
  }
  // If right wall and flame is to the right, follow right wall at a distance of 80
  else if((ADR6H > 80) && (fdiff < 0))
  {
    rdty = 50 - (1*(80 - ADR7H)/8);
    ldty = 50 + (1*(80 - ADR7H)/8);
  }
  // If flame and no left or right wall, follow flame
  else
  {
    rdty = 50 + fdiff;
    ldty = 50 - fdiff;
  }

  // Keep left and right motor duty cycles within 0 and 200
  if (rdty > 199) rdty = 199;
  if (ldty > 199) ldty = 199;
  if (rdty < 0 ) rdty = 0;
  if (ldty < 0 ) ldty = 0;

  // Set duty cycle
  if(fire)
    Set_DTY(0, 0);
  else
    Set_DTY(ldty, rdty);

  // If there is a white surface and a flame, set fire variable to indicate that it is time to
  //out the flame
  if(WhiteSurface() && (fsum > 200))
    fire = 1;
```

```
    // If there is a white surface and no flame, then the robot has exited the room
    if(WhiteSurface() && (fsum < 200))
      room_exited = 1;
  }

  i++;                    // Increment i
  counter++;              // Increment counter
  RTIFLG = 0x80;          // Reset RTI flag
}
```

```
/* This is the navigations header file. It contains functions used to navigate through
   through the maze */

#include <functions.h>

/* This is the room 0 (home circle) navigation code. This code will return the robot to
   room 1 if the fire is out and it started from room 1, return it to room 3 if the fire is out
   and it started fromroom 3, or, otherwise, bring the robot to the center of the maze
   facing rooms 1 and 2. */
void room0()
{
  // Go to center of maze
  if(room_start != 1)
  {
    // Left wall follow until there is no left wall
    golw(100, 100, 0);
    delay(100);
    while(LeftWall());
    // Go straight until there is a front wall
    go(80);
    while(!FrontWall());

    // Turn left 90 degrees, facing towards rooms 1 and 2 if room started from is not room
    // 3. This is room 1 position.
    if(room_start != 3)
    {
      tip90(left);
      delay(100);
      orient_right();
      room = 1;             // Set room = 1
    }
```

```
    // Return to room 3 if fire is out
    else
    {
      if(fireout)
      {
        tip90(right);
        delay(100);
        orient_left();
        // Left wall follow until there is a front wall
        golw(80, 100, 0);
        delay(100);
        while(!FrontWall());
        tip90(right);
        delay(100);
        orient_left();
        // Enter room 3 and stop
        golw(80, 100, 0);
        while(!WhiteSurface());
        delay(500);
        Set_DTY(0, 0);
        disableRTI();
        exit(0);
      }
    }
  }

  // Return to room 1
  else
  {
    if(fireout)
    {
      // Left wall follow into room 1 and stop
      golw(80, 100, 0);
      delay(500);
      while(!WhiteSurface());
      delay(500);
      Set_DTY(0, 0);
      disableRTI();
      exit(0);
    }
  }
}
```

```
/* This is room 1 navigation code. It will scan room 2 for a flame while it brings the robot
   to room 2 position, which is at the entrance of room 2 facing towards room 1. It will
   also return the robot to room 2 if the fire is out and the robot started from room 2. */
void room1()
{
 // Right wall follow until there is no right wall
 gorw(70, 110, 0);
 delay(400);
 while(RightWall());

 // Hamamatsu scan room 2
 if((room_start != 2) && !fireout)
 {
  goscan(); // Left wall follow hamamatsu scan
  // Go straight until there is a front wall
  go(60);
  while(!FrontWall());
  delay(50);
  // If no flame detected, scan in place
  if(!flamegh)
  {
   go(0);
   orient_left();
   delay(100);
   tip45(right);          // Look into room 2
   Set_DTY(0, 0);
   Ham_Scan();
  }

  // Determine the existence of a flame
  if((flameh == 1) || (flamegh == 1))
   flame = 1;
  else
   flame = 0;
 }

 // If robot didn't start from room 2 and there is no flame, turn around
 if((room_start != 2) && !flame)
 {
  if(!fireout)
  {
   tip45(right);
   tip90(right);
  }
```

```
    else
    {
      go(40);
      while(!FrontWall());
      tip180();
    }
    delay(100);
    orient_right();
  }
  room = 2;      // Set room = 2

  // Return to room 2
  if((room_start == 2) && fireout)
  {
    // Right wall follow until there is a front wall
    gorw(80, 100, 0);
    while(!FrontWall());
    tip90(right);
    delay(100);
    orient_left();
    // Left wall follow into room 2 and stop
    golw(80, 100, 0);
    delay(500);
    Set_DTY(0, 0);
    disableRTI();
    exit(0);
  }
}

/* This is the room 2 navigation code. It scans room 1 for a fire and then goes to either
    room 3 or room 4. If the robot started from room 3, this code will take it to room 4.
    Otherwise, the robot goes to room 3 */
void room2()
{
  // Right wall follow until there is no right wall
  gorw(40, 100, 0);
  delay(350);
  while(RightWall());
```

```
// Return to room 1 if robot started from there and the fire is out
if((room_start == 1) && fireout)
{
  // Right wall follow into room 1 and stop
  gorw(50, 100, 0);
  delay(1500);
  Set_DTY(0, 0);
  disableRTI();
  exit(0);
}

// Hamamatsu scan room 1
if((room_start != 1) && !fireout)
{
  // Hammamatsu scan while left wall following
  goscan();
  // If no flame detected, scan in place
  if(!flamegh)
  {
    orient_left();
    delay(50);
    tip45(right);          // Look into room 1
    Set_DTY(0, 0);
    Ham_Scan();
  }

  // Determine the existence of flame
  if((flameh == 1) || (flamegh == 1))
    flame = 1;
  else
    flame = 0;

  // If no flame turn left 45 degree to line up with left wall
  if(!flame)
    tip45(left);
}
```

```
if(flame)
{
  if(flameh == 1)
    go(0);
  // If flame was detected during the goscan, left wall follow until there is a full left wall.
  // This is how the robot will know that it lined up at the entrance of room 1
  else
  {
    golw(40, 80, 0);
    while(!LeftWall());
    delay(300);
  }
  room = 1;
}
// If no flame, move on
else
{
  golw(80, 100, 0);    // Left wall follow
  delay(800);

  // Go to room 3 position
  if(room_start != 3)
  {
    // Left wall follow until there is no left wall
    while(LeftWall());
    // Go straight a little bit and turn left 90 degrees
    go(60);
    delay(400);
    tip90(left);
    delay(100);
    // Go straight a little bit
    go(40);
    delay(400);
    // Left wall follw a little bit
    golw(60, 100, 0);
    delay(600);
    // Hamamatsu scan room 3
    goscan();
    // Go straight until there is a front wall
    go(60);
    while(!FrontWall());

    flame = flamegh;
```

```
      // If no flame turn around and set room = 3
      if(!flame)
      {
        tip180();
        delay(100);
        orient_right();
      }
      room = 3;
    }

    // Go to room 4 position if room start is 3
    else
    {
      // Left wall follow until there is no left wall
      while(LeftWall());
      // Go straight until there is a right wall
      go(80);
      while(!RightWall());
      // Right wall follow until there is no right wall
      gorw(80, 100, 0);
      delay(400);
      while(RightWall());
      // Go straight until there is a front wall
      go(80);
      while(!FrontWall());

      // Face room 4 and set room = 4
      tip90(right);
      delay(100);
      orient_left();
      room = 4;
    }
  }
}
```

```
/* This is the room 3 navigation code. It wil bring the robot to room 4 if it started from
   either room 1 or 2. It will return to room 4 if robot started from room 4 and the fire is
   out. If robot started from room 3, this will take the robot to room 1 position. */
void room3()
{
 // Right wall follow until there is no right wall
 gorw(40, 100, 0);
 delay(400);
 gorw(80, 100, 0);
 while(RightWall());
 // Go straight until there is a front wall
 go(60);
 while(!FrontWall());

 // Go to room 4 position if room start is 1 or 2
 if(room_start < 3)
 {
  tip90(left);
  delay(100);
  orient_right();
  // Right wall follow until there is no right wall
  gorw(60, 100, 0);
  delay(200);
  while(RightWall());
  // Left wall follow until there is a front wall
  golw(60, 100, 0);
  while(!FrontWall());
  // Face room 4 and set room = 4
  tip90(right);
  delay(100);
  orient_left();
  room = 4;
 }

 // Go towards center of maze
 else
 {
  // Face rooms 1 and 2
  tip90(right);
  delay(100);
  orient_left();
  // Left wall follow
  golw(60, 80, 0);
  delay(100);
```

```
// Go to room 1 position if room start is not 4
if(room_start != 4)
{
  // Left wall follow until there is no left wall
  while(LeftWall());
  // Go straight until there is a right wall and set room = 1
  go(60);
  while(!RightWall());
  Set_DTY(0, 0);
  room = 1;
}

//Enter room 4 if fire is out
else
{
  //Left wall follow until there is no left wall
  golw(80, 100, 0);
  delay(1000);
  while(LeftWall());
  // Go straight until there is a front wall
  go(60);
  while(!FrontWall());

  // Return to room 4 if fire is out
  if(fireout)
  {
    // At home circle, face room 4
    tip90(left);
    delay(100);
    // Right wall follow for a little bit
    gorw(80, 80, 0);
    delay(500);
    // Left wall follow into room 4 and stop
    golw(80, 100, 0);
    delay(500);
    while(!WhiteSurface());
    delay(400);
    Set_DTY(0, 0);
    disableRTI();
    exit(0);
  }
```

```
      // At home circle, turn around and begin left wall follow navigation
      else
      {
          tip180();
          delay(100);
          orient_left();
          golw(80, 100, 0);
          // Hallway navigation is complete
          hallway_nav_complete = 1;
      }
    }
  }
}

/* This is the room 4 navigation code. If starting from room 4, this take the robot to room
    1 position. Otherwise, this scans room 4 and then returns to home circle */
void room4()
{
  // Drive by room 4 if not started from there
  if(room_start != 4)
  {
    // Left wall follow for a little bit
    golw(80, 100, 0);
    delay(200);

    // Hammamatsu scan room 4 while wall following
    while(!flamegh && !RightWall() && !fireout)
      goscan();
    flame = flamegh;

    // If no flame, return to home circle
    if(!flamegh)
    {
      // Left wall follow until there is a front wall
      golw(90, 100, 0);
      while(!FrontWall());
      tip90(right);          // Turn right
      delay(100);
      orient_left();
```

```
    // Left wall follow until there is a front wall
    golw(80, 100, 0);
    delay(200);
    while(!FrontWall());
    tip90(right);          // Turn right
    delay(100);
    orient_left();
    Set_DTY(0, 0);
    // Robot is at home circle now. Set room = 0
    room = 0;
    if(room_start == 0)
    {
        // Hallway navigation is compete
        hallway_nav_complete = 1;
        // Exit if fire is out
        if(fireout)
          exit(0);
    }

    // If room start is 1, 2, or 3 and fire is not out, the hallway navigation is complete
    if(((room_start < 4) && (room_start != 0)) && !fireout)
        hallway_nav_complete = 1;
  }
}

// If room start is 4, go to room 1 position
else
{
  // Right wall follow until there is a front wall
  gorw(80, 100, 0);
  while(!FrontWall());
  tip90(left);   // Turn left
  delay(100);
  orient_right();
  // Right wall follow until there is no right wall
  gorw(80, 90, 0);
  delay(100);
  while(RightWall());
  // Go straight until there is a left wall
  go(60);
  while(!LeftWall());
  // Left wall follow until there is no left wall
  golw(60, 80, 0);
  delay(100);
  while(LeftWall());
```

```
   // Go straight until there is a right wall and
   // set room = 1
   go(60);
   while(!RightWall());
   Set_DTY(0, 0);
   room = 1;
  }
}

/* This function performs the hallway navigation to find and extinguish the flame and
    return home */
void Hallway_Nav()
{
 fireout = 0;    // Initialize fireout

 // Run the proper room navigation codes until the hallway navigation is complete
 while(hallway_nav_complete == 0)
 {
  if((room == 0) && !flame)
    room0();
  if((room == 1) && !flame)
    room1();
  if((room == 2) && !flame)
    room2();
  if((room == 3) && !flame)
    room3();
  if((room == 4) && !flame)
    room4();

  // If flame has been detected, enter room
  if(flame)
  {
    // Procedures for entering rooms 1, 2, and 3 and putting out the flame
    if(room != 4)
    {
        wlc = 0;
        counter = 0;
        tip90(right);    // Turn right
        delay(100);
        // If room is 2 or 3, orient parallel to left wall
        if((room == 2) || (room == 3))
          orient_left();
```

```
   // If room = 1, go straight until robot has crossed the white line or counter reaches
   // 150
   if(room == 1)
   {
     go(40);
     while((wlc == 0) && (counter < 150))
       Count_Whitelines();
   }

   counter = 0;     // Reset RTI counter

   // Left wall follow until robot has crossed a white line or counter reaches 150
golw(40, 100, 0);
while((wlc == 0) && (counter < 150))
     Count_Whitelines();

 // Find flame
 while(!room_exited && !fire)
 {
  // Left wall follow while there is no flame
  golw(80, 100, 0);
  while(!AFlame())
  {
       // If there is a front wall, turn right, orient parallel to left wall, and continue left
       // wall following
       if(FrontWall())
       {
         tip90(right);
        delay(100);
        orient_left();
        golw(80, 100, 0);
       }
  }
  // If there is a flame, follow it
  Flame_Follow();
}
```

```
  // Blow out the fire
  while(fire)
  {
    go(20);
    delay(150);
    Flame_Out();
    go(0);
    fsum = ADR4H + ADR5H;
    // Check if fire still exists
    if(fsum < 100)
    {
      flame = 0;
      flamegh = 0;
      flameh = 0;
      fire = 0;
      fireout = 1;
    }
  }

  // Exit room
  if(room_exited)
    flame = 0;
  else
    Exit_Room();
}

// Procedures for entering room 4 and putting out the flame
else
{
  // Right wall follow into room 4 and then make a 45 degree right turn
  gorw(40, 80, 0);
  delay(800);
  tip45(right);
  // Go straight until a white line has been seen or the counter has reached 100
  go(60);
  counter = 0;
  wlc = 0;
  while((counter < 100) && (wlc == 0))
    Count_Whitelines();
```

```
// Right wall follow until a white line has been seen
gorw(40, 100, 0);
while(wlc == 0)
  Count_Whitelines();

while(!room_exited && !fire)
{
  // Right wall follow while there is no flame
  gorw(60, 100, 0);
  while(!AFlame())
  {
    // If there is a front wall, turn left, orient parallel to right wall, and continue right
    // wall following
    if(FrontWall())
    {
      tip90(left);
      delay(100);
      orient_right();
      gorw(60, 100, 0);
    }
  }
  // If there is a flame, follow it
  Flame_Follow();
}

// Blow out fire until it is out
while(fire)
{
  go(20);
  delay(150);
  Flame_Out();
  go(0);
  fsum = ADR4H + ADR5H;
  // Check if fire is out
  if(fsum < 100)
  {
    flame = 0;
    flamegh = 0;
    flameh = 0;
    fire = 0;
    fireout = 1;
  }
}
```

```
      // Exit room
      if(room_exited)
        flame = 0;
      else
        Exit_Room();
    }
   }
  }
}

/* This is the left wall follow navigation code. This code will left wall follow through the
   entire maze, entering every room, to find the candle */
void LWF_Nav()
{
 wlc = 0;
 golw(70, 100, 0);      // Left wall follow
 while(!fire)
 {
  while(!AFlame())
  {
   Count_Whitelines();
   // If there is a front wall, turn right, orient parallel to left wall, and continue left wall
   // following
   if(FrontWall())
   {
    tip90(right);
    delay(100);
    orient_left();
    golw(70, 100, 0);
   }
   // If white line counter equals 7 and a white surface has been detected, robot is at the
   // home circle. Turn around and left wall follow into room 4
   if((wlc == 7) && WhiteSurface())
   {
    tip180();
    orient_right();
    golw(70, 100, 0);
   }
   // If over 10 whitelines have been counter, give up
   if(wlc >= 10)
    error();
  }
  // If in a room and there is a flame, follow it
  if((wlc != 0) && AFlame())
   Flame_Follow();
 }
```

```
  room = wlc/2;                    // Determine room
  // Blow out fire until it is out
  while(fire)
  {
    go(20);
    delay(150);
    Flame_Out();
    go(0);
    fsum = ADR4H + ADR5H;
    // Check if fire is out
    if(fsum < 100)
    {
      flame = 0;
      flamegh = 0;
      flameh = 0;
      fire = 0;
      fireout = 1;
    }
  }
  Exit_Room();
  hallway_nav_complete = 0;
}
```

```
/* This is the main program file */

#include <setup.h>
#include <navigation.h>

main()
{
  INTCR &= ~0x60;   // Disable IRQ
  PWM_setup();      // Setup PWM
  LATCH_setup();    // Setup latch
  HB_setup();       // Setup H-bridge
  AD_setup();       // Setup A/D ports
  DLC_setup();      // Setup Port DLC
  HAM_setup();      // Setup hamamatsu

  Wait_for_Tone();
  delay(250);
  enableRTI();

  // Scan and exit room
  Scan_Room();
  if(room_start != 0)
    Exit_Room();
  Set_DTY(0, 0);
  disableRTI();

  enableRTI();
  // If room start has been determined, begin hallway navigation
  if(room_start != 128)
  {
    Hallway_Nav();
    // If hallway navigation is complete and the fire is not out, begin left wall following
    if(hallway_nav_complete && !fireout)
    {
      LWF_Nav();
      Hallway_Nav();
    }
  }
  // Left wall follow through maze if room start was not correctly determined
  else
    LWF_Nav();
}
```