



PAC-MAN

EECE 474 – Team 1

July 24, 2002

PAC-MAN



Submitted to:
Dr. W.G. Dunford

July 24, 2002

Submitted by:
Pauline Pham (#44064988)
Carol Tai (#78264991)
Johnson Bao (#64704984)
Christopher Chou (#82812991)
Jimmy Huang (#4798998)

Table of Contents

List of Tables	iii
List of Figures	iii
Abstract	iii
1.0 Introduction	1
2.0 Chassis	2
2.1 Motor Mounts.....	2
2.2 PCB Mounts	3
2.3 Body	3
3.0 Motors	6
3.1 Motor Selection	6
3.2 Motor Control.....	7
4.0 Radio Frequency Application	9
4.1 RF Considerations	9
4.2 Wireless Input Controller	9
4.2.1 Encoding scheme logic.....	10
4.2.2 Encoder and Decoder	11
4.2.2.1 Error detection and filtering	12
4.2.3 RF Modules	13
5.0 Sensors	16
5.1 Wall detection	16
5.2 Dot Counting Sensor	18
5.3 Pac-Man Tracking.....	19
5.4 Contact Sensor.....	21
5.5 LCD display	21
6.0 Microcontroller and Software	23
6.1 Pac-Man Robot.....	23
6.1.1 Obstacle Sensors	25
6.1.2 LCD Display	26
6.1.3 RF Receiver.....	26
6.1.4.....	27
Stepper Motors	27
6.1.5 Ghost Contact Sensor	29
6.1.6 Maze Dot Sensor	29
6.2 Ghost Robot.....	30
6.2.1 Beacon Sensor	31
7.0 Maze	35
7.1 Physical Dimension.....	35
7.2 Dots	36
8.0 Power	38
8.1 Motor.....	38
8.2 Controller	38
8.3 Maze	38
8.4 Regulators.....	39

9.0	Recommendations	41
10.0	Conclusion.....	43

APPENDICES

- APPENDIX A: Photo Gallery
- APPENDIX B: PCB Layouts
- APPENDIX C: Expense Report
- APPENDIX D: Gantt Chart
- APPENDIX E: Source Code

List of Tables

Table 1. Encoding scheme	11
Table 2. RF signal format for Pac-Man movement	27
Table 3. List of parameters needed to control motor speed.....	27
Table 4. Truth table of the Memory Device	37

List of Figures

Figure 1. Aluminum Corner.....	2
Figure 2. Wheel.....	2
Figure 3. Motor Mounts.....	3
Figure 4. PCB Mount.....	3
Figure 5. Chassis Body	4
Figure 6. Pac-Man South-west view.. ..	4
Figure 7. Front view.....	4
Figure 8. Top view... ..	5
Figure 9. Side view... ..	5
Figure 10. 55M048B2U 12VDC Unipolar Stepper Motor	7
Figure 11. Pin Assignments and internal circuitry of UCN5804.....	8
Figure 12. Wireless Input Controller	10
Figure 13. Encoding scheme logic.....	10
Figure 14. MC145026 Encoder Block Diagram.....	11
Figure 15. MC145027 Decoder Block Diagram.....	12
Figure 16. Encoder and Decoder transmission protocol.....	13
Figure 17. Transmitter Module	14
Figure 18. Receiver Module	14
Figure 19. Transmitter Block Diagram.....	14
Figure 20. Receiver Block Diagram	15
Figure 21. GP2D12	17
Figure 22. Wall Sensor... ..	17
Figure 23. Wall Sensor Circuit	17
Figure 24. Dot Counting Circuit	18
Figure 25. PNA4612.... ..	20
Figure 26. Beacon Receiver Circuit.....	20
Figure 27. IR Beacon Circuit.....	20
Figure 28. Contact Sensor.....	21
Figure 29. LCD display.....	22
Figure 30. Architectural model of Pac-Man software	24
Figure 31. Pin assignments for Pac-Man	25
Figure 32. Model of interface between PIC and obstacle sensors	26
Figure 33. Model of interface between PIC and LCD display	26
Figure 34. Model of interface between PIC and RF receiver	27
Figure 35. Sketch of wheel dimensions	28
Figure 36. Model of interface between PIC and stepper motor drivers.....	29

Figure 37. Model of interface between PIC and Ghost contact sensor.....	29
Figure 38. Model of interface between PIC and maze dot sensor	29
Figure 39. Architectural model of Pac-Man software	30
Figure 40. Pin assignments for the Ghost.....	31
Figure 41. Model of interface between PIC and beacon sensors.....	31
Figure 42. Front beacon measurements of Pac-Man position.....	32
Figure 43. Left beacon measurements of Pac-Man position.....	32
Figure 44. Example of the maze fast flooding algorithm	34
Figure 45. Maze Design.....	35
Figure 46. Dot Circuitry for the Maze (for one dot).....	36
Figure 47. Timing diagram of the Switch (including CD4066, resistor, and capacitor) ..	37
Figure 48. LM2575 Circuit Schematic	39
Figure 49. LM2575 Circuit Block Diagram	39

Abstract

The EECE474 Pac-Man project is a robotic counterpart of the computer game, Pac-Man. As in the original Pac-Man game, the Pac-Man robot, controlled by a player via a RF link, moves around the maze collecting LED “dots” while avoiding contact with the Ghost, which is an autonomous robot capable of tracking Pac-Man’s location in the maze. The microcontrollers onboard the robots are responsible for reading the wall sensors and controlling the stepper motors to perform precision turning and stopping. To simulate the dots being “eaten” by Pac-Man, the maze dot modules, equipped with light sensors are capable of turning off the LEDs once Pac-Man passes through. The number of dots collected by Pac-Man is shown on the LCD display. The tracking mechanism is achieved by mounting IR receivers onto the Ghost to detect Pac-Man’s IR beacon signature and using such information to compute the shortest path to reach Pac-Man in the maze. Contact sensors are also used to detect collisions between Ghost and Pac-Man.

1.0 Introduction

The objective of our EECE474 project is to design a Pac-Man game involving a remote control system that is suitable for children who are six years old and above. The idea of this project comes from the computer game Pac-Man. The rules and the features of our project are similar to the computer game.

Our goals are to implement two wireless and motor running robots and to set up a game setting, while maintaining the same features as in the computer game. The features include the fact that Pac-Man keeps score of the number of dots collected, the Ghost traces Pac-Man and finds the shortest path to it, and that the Ghost travel at a speed faster than Pac-Man.

The project involves two robots – Pac-Man and Ghost. In general, the objective of the game is for players to control Pac-Man's movements within the maze with a wireless controller. Pac-Man must eat all the dots on the maze and avoid contact with the Ghost at the same time. Pac-Man is given three lives. If Pac-Man comes into contact with the Ghost three times before eating all the dots, the game is pronounced over.

Our Pac-Man robot consists of the following systems:

- 1) an RF system that receives directional instructions from users
- 2) a wall detection system
- 3) a score keeping system
- 4) a motor system
- 5) a contact sensor that detect contacts with the Ghost.

The Ghost is an autonomous robot that is:

- 1) able to detect walls
- 2) to automatically traverse the maze
- 3) to locate and catch Pac-Man
- 4) to detect contact with Pac-Man.

Our project is divided into two main components – hardware and software. Both the hardware and the software are further broken down into several individual subsystems. Each subsystem is tested and implemented separately, and integrated together to construct our final project. The testing results and the design processes of each subsystem are discussed in detail in the following sections. In addition, problems encountered and the solutions are described.

2.0 Chassis

The design objective of the chassis is to enable Pac-Man and the Ghost to maneuver easily in the maze. Since most of the robot's weight comes from the batteries and motors, in order to keep the weight to a minimum, the chassis was designed to be as lightweight and small as possible. Therefore, 16-gauge sheet aluminum was chosen to be the main building material for the chassis. The chassis is designed to have three modules: motor mounts, PCB mounts, and the body. This design makes the robot easy to assemble and disassemble. The detail design of each module is described in the following sections.

2.1 Motor Mounts

The robot is designed to accomplish standing 90 degree or 180 degree turns in the maze. In order to do this, the two motors are positioned in the middle of each side of the robot, so simply rotating the two motors in opposite directions achieves the standing turn. To mount the motors on to the chassis body, we made a L-shape aluminum plate (a corner) to attach the motors to the chassis (see Figure 1).

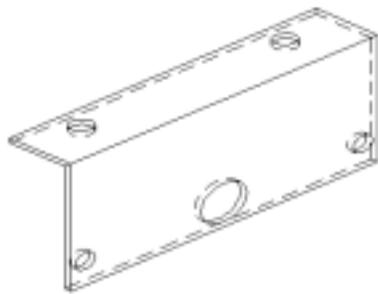


Figure 1. Aluminum Corner

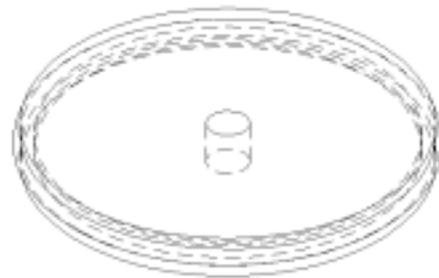


Figure 2. Wheel

The wheels we used, shown in Figure 2, are plastic disks made by the machinist according to the specified dimension. The dimensions of the wheels were chosen so it gives the motor and the dot-counting sensor just enough clearance off the ground. The wheel clamps straight on to the shaft of the motor using a setscrew, so it has a one to one gear ratio. This is why the wheels have to be made just right, otherwise the robot would be moving faster than the desired speed.

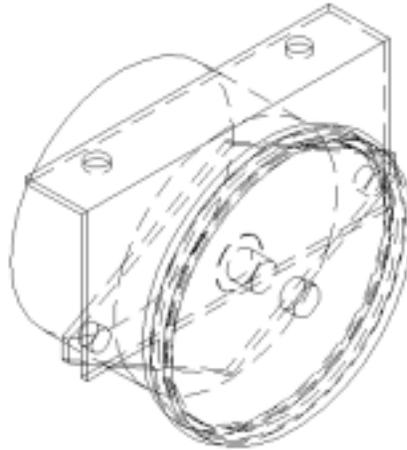


Figure 3. Motor Mounts

2.2 PCB Mounts

In the original design, the PCBs were to be mounted in layers using screws and spacers at each corner. However, after looking at the design carefully, we found this mounting method to be inconvenient in terms of accessing and debugging the PCBs. Therefore, we decided to switch to the wooden slots that we are currently using. This mounting mechanism enables us to slide each of the layers in and out individually for debugging. Just like the motors, the wooden PCB mounts are attached to the main body using an aluminum corner. (See Figure 4)

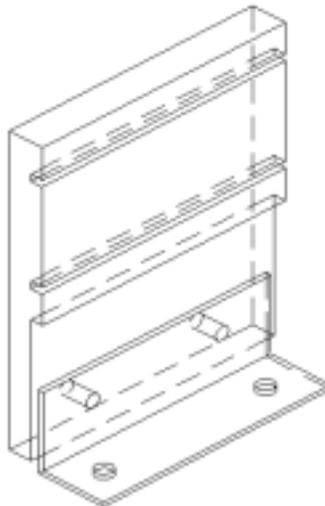


Figure 4. PCB Mount

2.3 Body

The chassis body, as shown in Figure 5, is actually quite simple; it consists of a 10cm x 10cm aluminum plate, two standing casters in the front, and two ball

casters in the back. For any two wheel robot, casters are needed to keep the robot in balance, but usually only one ball caster is needed in the back. For Pac-Man and the Ghost, because the LED dots are planted at the center of each lane, the casters were moved away from the center in order to accommodate the LEDs and sensors. Moreover, having only one caster placed off center may cause an uneven drag and affect the movement of the robot. Therefore, we placed two ball casters in the back to keep the robot balanced. The front standing casters are there to prevent the robot from tilting forward on the brake.

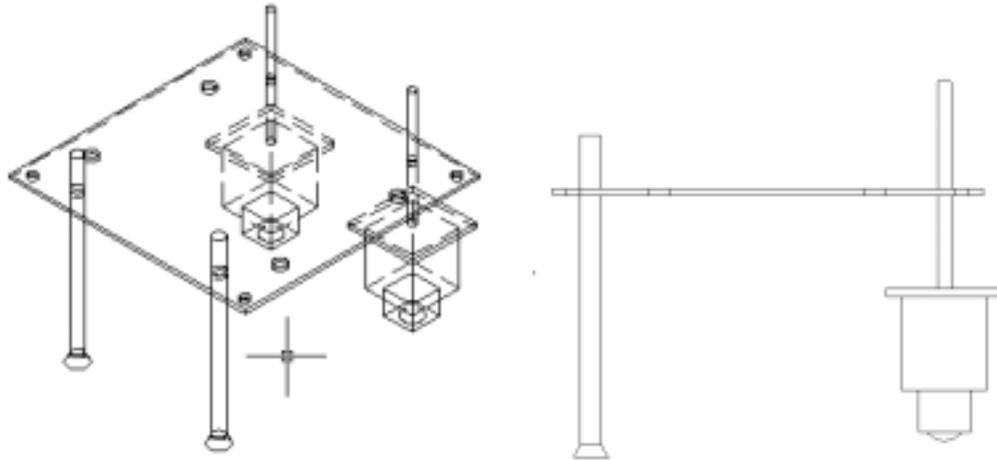


Figure 5. Chassis Body

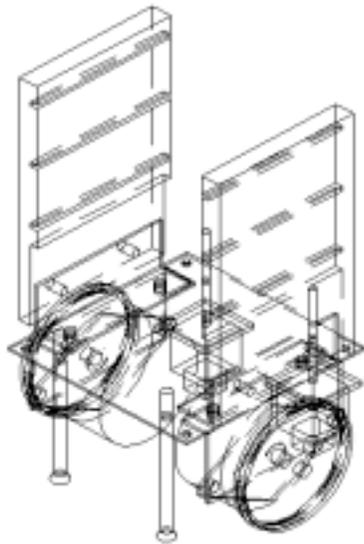


Figure 6. Pac-Man South-west view

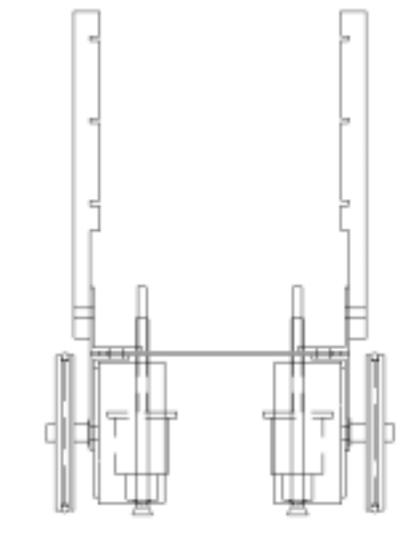


Figure 7. Front view

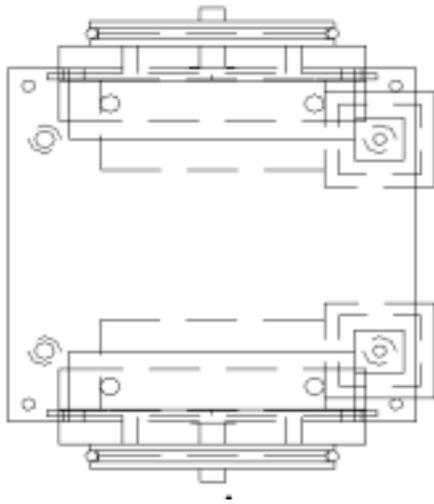


Figure 8. Top view

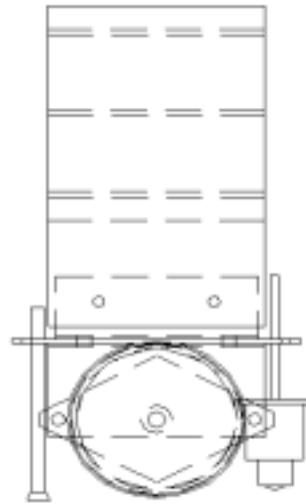


Figure 9. Side view

3.0 Motors

In order to have Pac-Man and the Ghost traverse through the maze efficiently, the movements of the robot need to be precise and easy to control. Our design requires the robots to make near perfect 90 degree and 180 degree turns on a touch button command. Therefore, it is critical to have the right motors for the purpose and a control mechanism, which makes it easy for the microprocessor.

3.1 Motor Selection

For the robots' drive motor, we had to choose from three types of motors: DC, servo, and stepper motors. DC motors are capable of providing high speed and torque for the robot, but they require the proper gearboxes and shaft encoders to achieve the desirable control. As for the servomotors, they have a much simpler control mechanism, but their speed is limited, and in most cases, they are not capable of the full rotation needed for driving purposes.

As a result, we decided to settle with stepper motors. Stepper motors are easy to control with the help of proper translation logic and it can be used to drive the robot without any modifications. However, there is one disadvantage that we did not find out until we started working with them. Although the stepper motors are capable of full rotation, in most cases, they are not designed to provide enough torque for driving small robots. Fortunately, there are special high torque stepper motors that are designed for driving applications. For our robots, we decided to use 12VDC unipolar stepper motors, the 55M048B2U from Thomson Industries Inc.; Figure 10 below shows the dimension of the motor.

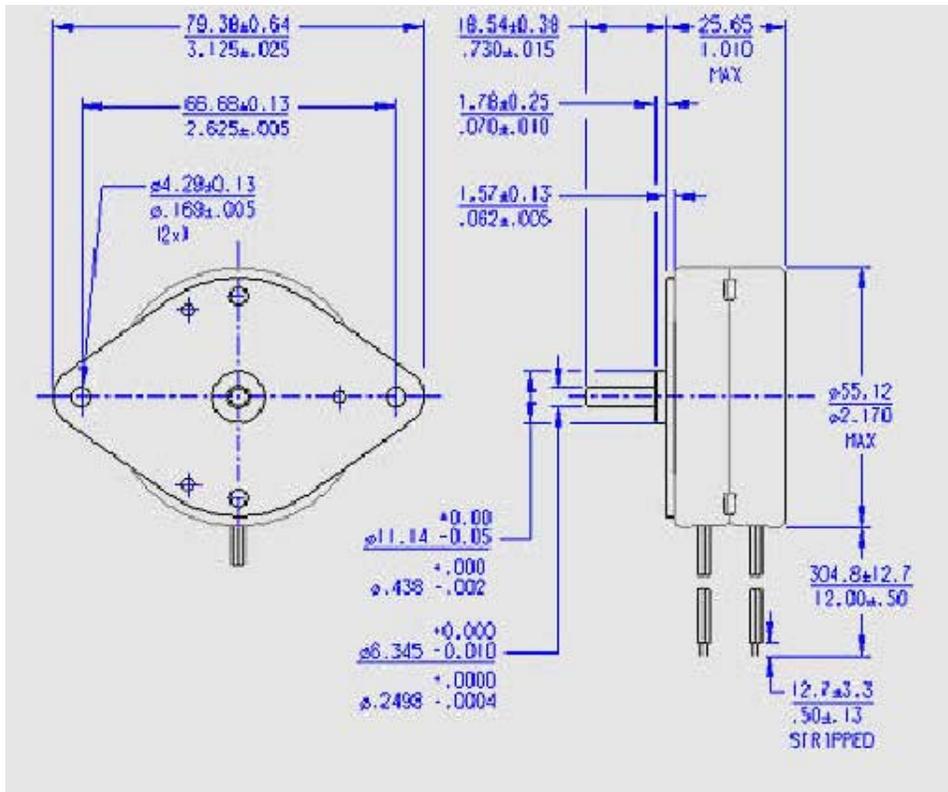


Figure 10. 55M048B2U 12VDC Unipolar Stepper Motor

3.2 Motor Control

Opposite from DC motors, stepper motors have an armature built out of permanent magnets, and surrounded by sets electromagnets that are activated on demand. By activating different sets of coils in a particular sequence, we can move the armature from one position to the next to create the rotation required. The activation sequence can be generated by programming the microprocessor or by logic devices, such as the L297 stepper motor controller, which translates direction and clock signal into corresponding sequences.

Both Pac-Man and the Ghost are driven by two motors (left and right), which is controlled by the UCN5804 unipolar stepper-motor translator/driver from Allegro Microsystems Inc. Figure 11 below shows the pin assignment and internal circuitry of the UCN5804.

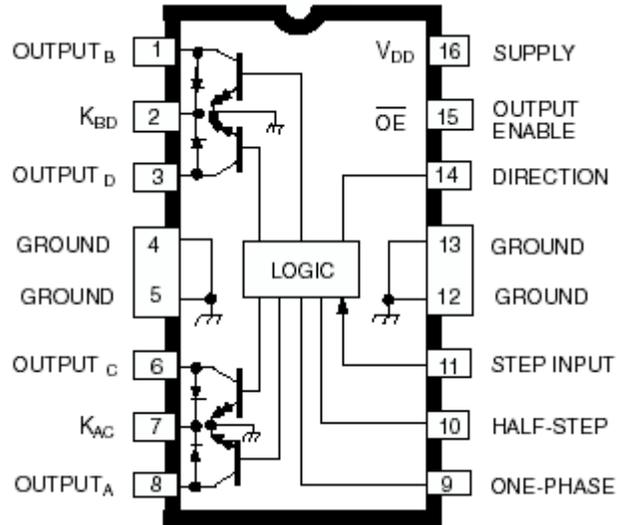


Figure 11. Pin Assignments and internal circuitry of UCN5804

The motors we used can be controlled by combining the L297 with a Darlington transistor array. However, the UCN5804 not only integrated the translation logic with the Darlington transistors, it is also capable of sinking twice as much current (1.25A) than regular Darlington arrays (600mA). Therefore, we chose the UCN5804 as our motor controller to eliminate extra circuitry and to lighten the load on the microprocessor.

4.0 Radio Frequency Application

Since one of the goals in this project is to implement a user controlled Pac-Man, it was essential to have the Pac-Man robot move freely through the maze without having communication wires from the controller suspending from it. Therefore, radio frequency was introduced into our project in order to make the communication between Pac-Man and its user controller wireless. When dealing with radio frequency applications, the transmission protocol and noise factors were key issues taken into consideration in our equipment selection and design.

4.1 RF Considerations

We initially considered a two-way communication link between Pac-Man and the controller. This would have allowed us to control Pac-Man from the PC on the downlink and it would allow Pac-Man to send useful information to the PC (such as data for dot counting and counting lives) on the uplink. We also considered a wireless communication link between Pac-Man and the Ghost. This would have allowed us to send Pac-Man's position coordinates to the Ghost so that the Ghost can track Pac-Man down. However, due to budget restraints and in order to avoid additional programming and synchronization issues between the transmitter and the receiver modules, we decided to use one-way communication between the controller and Pac-Man only and implement the hardware required to count dots and lives on Pac-Man. By simplifying our RF requirements, we were able to design a controller without having to interface it with the PC or a separate microcontroller.

4.2 Wireless Input Controller

A wireless input controller was designed to specifically control the movements of Pac-Man. The controller consists of three subsystems: the encoding scheme logic, the encoder and decoder, and the RF modules. Four momentary SPDT (single-pole-double-throw) pushbuttons were selected as the control buttons such that each time a user pushes a button, the output would go high, else it remains low at all times. From there, an algorithm was devised to encode the output into a specific scheme. This output is then sent to the encoder where it is sent serially to the RF transmitter module. The RF transmitter module sends the data off at 418MHz to the receiver module on Pac-Man. The data is then transferred to the decoder and outputted through three pins to the microcontroller. The following diagram describes the operation of the wireless controller.

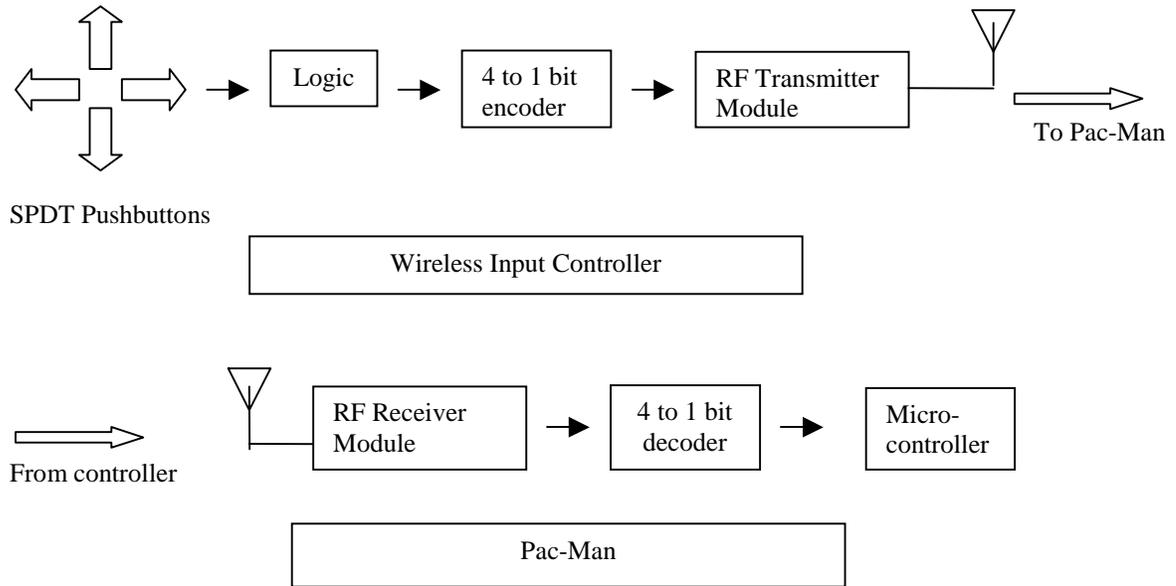


Figure 12. Wireless Input Controller

4.2.1 Encoding scheme logic

There are four movements required to control Pac-Man in the maze: up, down, right, and left. The following logic was designed to meet these specifications.

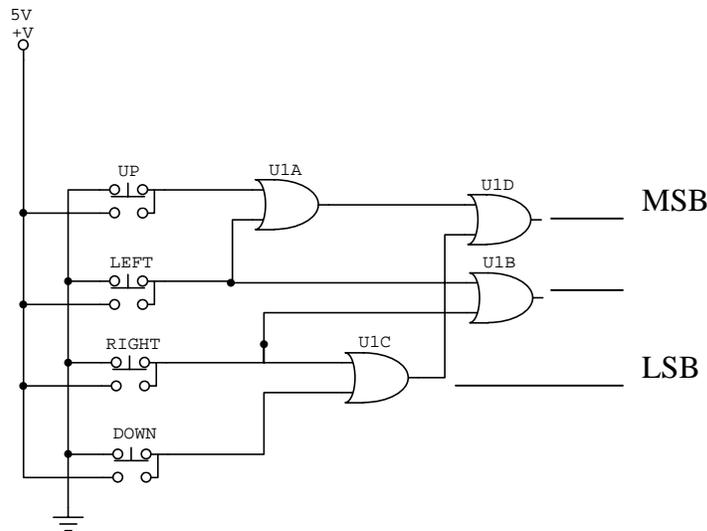


Figure 13. Encoding scheme logic

As each button is pressed, the following outputs are sent to the encoder chip.

Table 1. Encoding scheme

Up	100
Left	101
Right	110
Down	111

The most significant bit was selected to act as an interrupt signal for the micro-controller so that it goes high each time a button is pushed.

4.2.2 Encoder and Decoder

The encoder and decoder chip used in this project is Motorola's MC145026 encoder and MC145027 decoder chips. See Figure 14 and Figure 15 for their block diagrams. Since more than one 474 group was using RF modules at 418MHz, these chips were selected for our application because they performed the necessary error checking and filtering required. In addition, they provided parallel-to-serial and serial-to-parallel data conversion, which was a requirement for the RF modules (the RF modules transmit and receive serial data only).

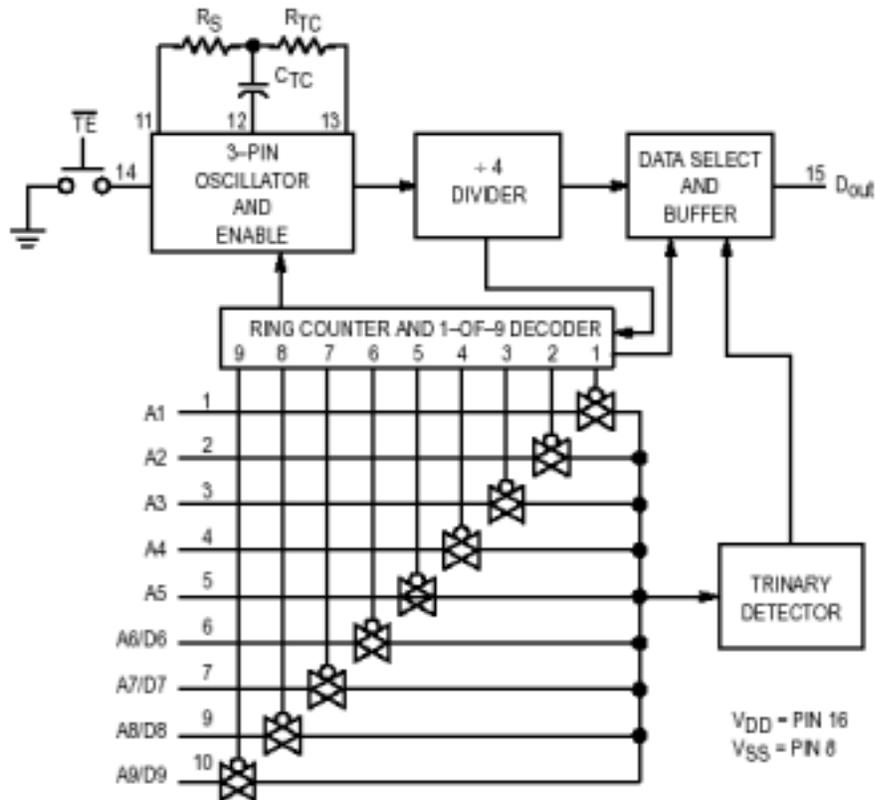


Figure 14. MC145026 Encoder Block Diagram

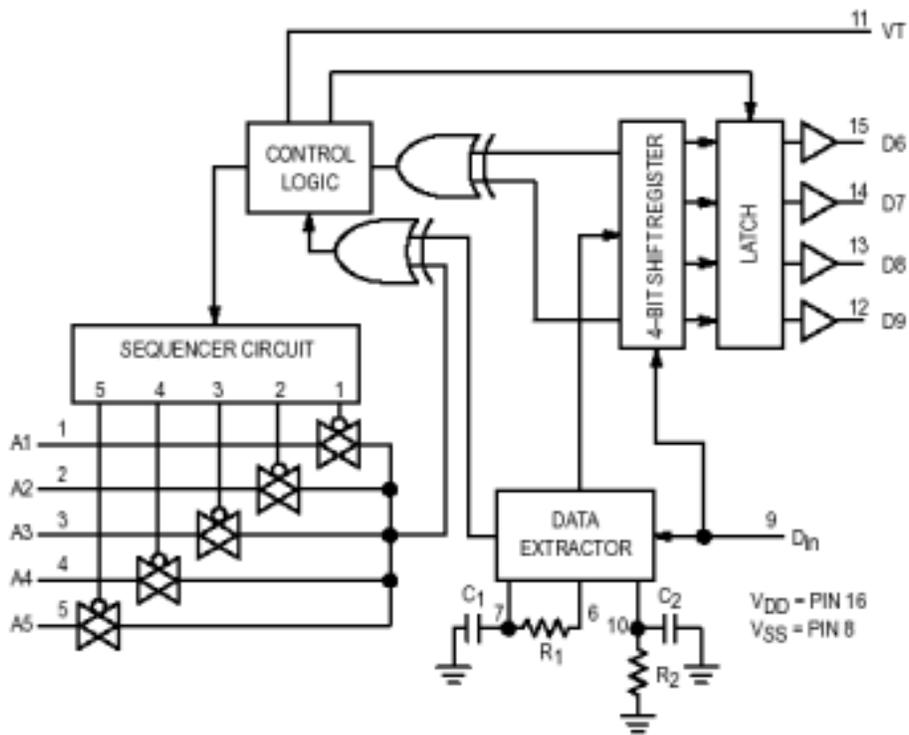


Figure 15. MC145027 Decoder Block Diagram

4.2.2.1 Error detection and filtering

The encoder chip contains nine bits of information. The first five bits contain the address of the encoder and the other remaining four bits contain the data bits. These nine bits of information are sent serially to the RF transmitter module. The RF transmitter module then sends the serial data asynchronously at 418MHz to the receiver module. For this project, only three out of the four data bits were used. The fourth bit was left opened. The decoder receives the serial data via the RF receiver module, unscrambles the data and checks to see if two consecutive addresses are matched to the local address of the decoder. Secondly, it checks to see if the four data bits match the last valid data received. If both conditions are met, the data bits are outputted to the microcontroller.

Both encoder and decoder chip contain an internal RC oscillator. Since the RF modules were tested and found to operate best for input frequencies up to 2kHz, the encoder and decoder clock frequencies were set to approximately 1.7kHz. This resulted in an output frequency of approximately 420Hz to the RF modules. As seen in the figure below, each data bit generated from the encoder

is held for several clock cycles and each transition period amounts to half a clock cycle. Since both chips were set to operate with the same clock frequency, the decoder is able to unscramble and synchronize the data received to its own internal clock. Figure 16 illustrates how the data is unscrambled at the decoder. Note that VT (valid transmission) only goes high once two sets of words have been received from the encoder.

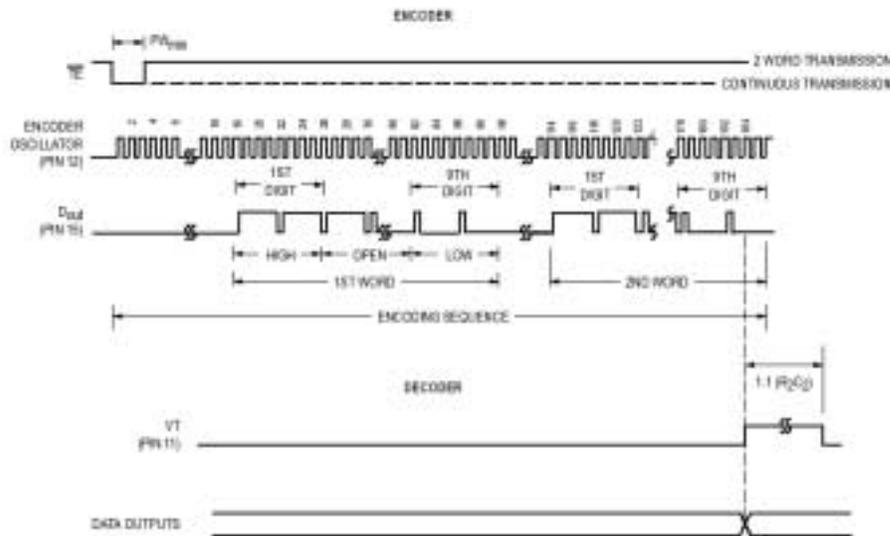


Figure 16. Encoder and Decoder transmission protocol

4.2.3 RF Modules

The RF module was selected based on reliability, size, and easiness of implementation. RF modules made by Abacom, Melex, Ming Microsystems, Ramsey, Linx and RF solutions were researched, however, the modules produced by Linx Technologies was chosen based on previous project success and because it required the least amount of external circuitry. In addition, since our controller's case had a size restraint, the Linx transmitter module was ideal due to its small packaging. See Figure 17 and Figure 18 for their schematics.

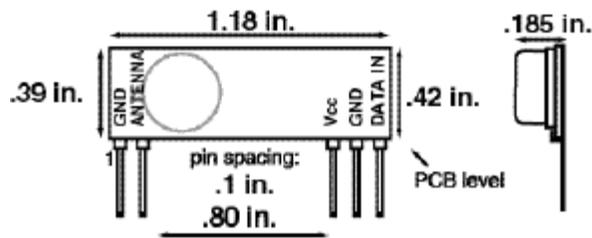


Figure 17. Transmitter Module

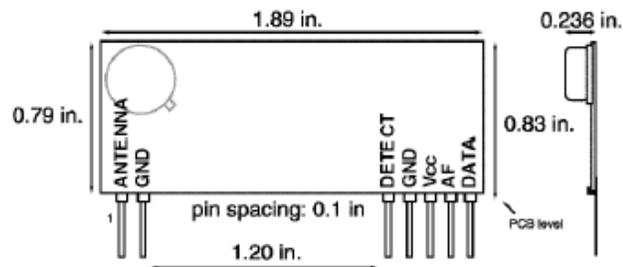


Figure 18. Receiver Module

The Linx modules use precision SAW (Surface Acoustic Wave) techniques and FM/FSK (Frequency Modulation/Frequency Shift Keying) modulation. Figure 19 and Figure 20 show the internal operation of the RF modules. As mentioned before, the Linx modules require no external circuitry other than an external antenna. A 418MHz $\frac{1}{4}$ wave whip antenna was selected as it was recommended to work best with these modules. In addition, the modules recommended that a slow data rate be selected since it increases the performance of the RF modules. As mentioned above, we chose to send a 420Hz signal to the RF transmitter module.

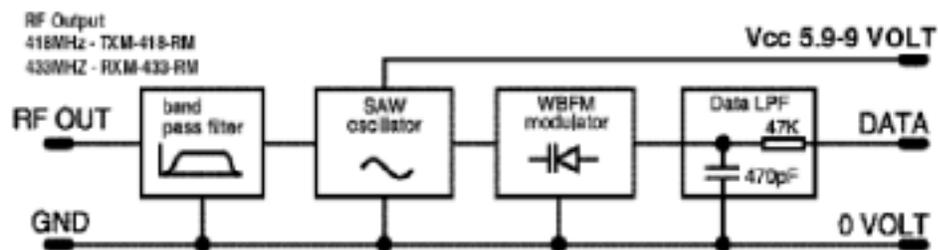


Figure 19. Transmitter Block Diagram

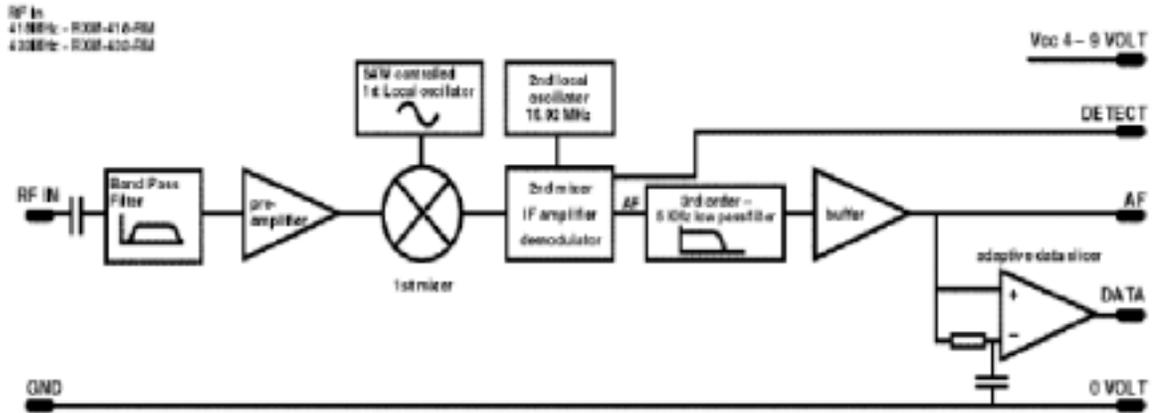


Figure 20. Receiver Block Diagram

The Linx modules have the capability of sending analog or digital data at distances greater than 500 ft. We tested these modules by sending a signal from the function generator from one end of the 474 room to the other end. The modules received the signal instantaneously and mirrored the signal exactly as we varied the frequency. We chose not to test the modules at greater distances since it was unnecessary to do so for our specific application.

5.0 Sensors

Several different types of sensors are used to achieve the following functionalities in the Pac-Man project:

- 1) Wall detection – Pac-Man and Ghost should be able to maneuver around the maze without hitting or scraping the walls, and meanwhile detect openings in the maze. Given the configuration of the robots and the maze, this functionality requires distance sensors that have good resolution within a 15 cm range.
- 2) Dot counting – The “Dots” in the Pac-Man game is realized by the floor mounted LEDs in the maze. Pac-Man should be able to distinguish between the bare floor and an LED light source as it passes over them even when it is not centered within the track. As in the computer game, the dots disappear after eaten by Pac-Man. Our LEDs should also turn off immediately after it has been detected.
- 3) Pac-Man location tracking – Ghost should be able to locate Pac-Man from a give distance within the maze. This functionality requires sensors that have a wide detection angle (approximately 45 degrees) and a long detection range (approximately 120 cm).
- 4) Contact - When Pac-Man and Ghost physically make contact, both robots should be able to detect the event.

Given the above functional requirements, the following sensor designs are investigated as possible solutions.

5.1 Wall detection

We first considered the Sharp GP2d12 IR ranger module (Figure 21), which is equipped with an infrared transmitter and receiver pair. Several features makes GP2D12 an attractive candidate – 1) Small packaging (0.75in x 0.5in) which helps keep the size of our robot under control, 2) GP2D12’s analog output (0.25V to 2.45V) is easily interfaced with the PIC microcontroller’s internal A-D converter, and 3) high immunity to ambient light.

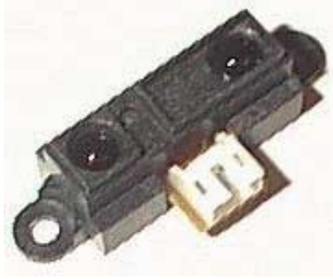


Figure 21. GP2D12

However, it also has a significant drawback. The accurate sensing range is between 80cm to 10cm with the output gradually increasing as the obstacle gets closer. Once the distance is closer than 10cm, the output begins to drop. This poses a serious problem since our robot will not be able to distinguish between a straightaway (wall beyond 10 cm), and a wall 1cm away.

After extensive research, we decided that a sensor design using a pair of CdS photo-resistor and a LED is best suited for our needs. Figure 22 illustrates its operation and Figure 23 is the circuit drawing for the sensor.

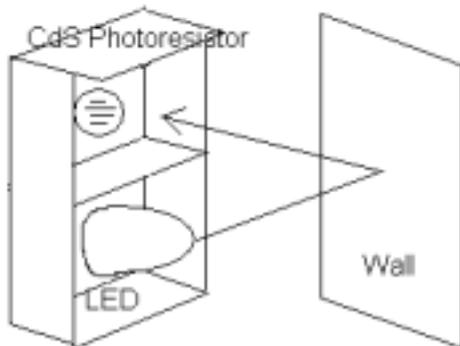


Figure 22. Wall Sensor

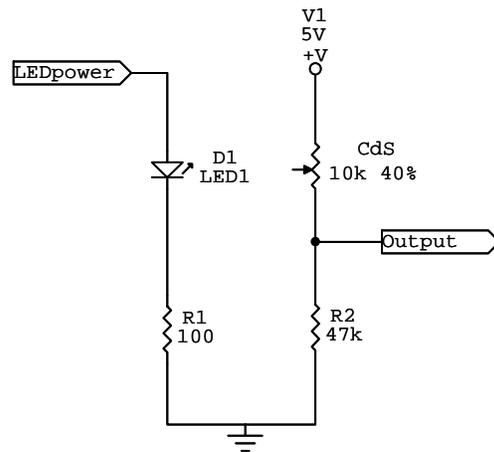


Figure 23. Wall Sensor Circuit

The resistance of the CdS photoresistor is inversely related to the surrounding light intensity. As the light intensity increases, the resistance of the CdS resistor drops. We therefore incorporate the photoresistor in a voltage divider, creating a varying voltage level at the output node as shown in Figure 23. As the sensor approaches the wall, the LED lights up the wall surface. Depending on the amount of light reflected into the CdS cell, the output voltage gradually increases as the sensor moves closer to the wall.

Initial testing of the Figure 22 set-up showed promising results: the voltage output varied from 2.5V when sensor was right in front of the wall to around 1.2V when there was no wall in front of it. However upon further testing, we discovered that varying ambient light intensity on the maze wall resulted in a 0.5V output deviation at the desired stopping distance (wall clearance) of 3 cm. To reduce the interference from the ambient light, we decided to replace the original red LED with a super white LED which has a much greater intensity. Testing revealed that at the desired stopping distance the sensor output already reached 3V, which means that the ambient light now contributes to much less of the overall reflected intensity. Testing also showed that although the output is now consistent with different wall lighting, the output at the desired range still varies around 0.3V from one sensor to another due to the slight differences in the photoresistors. In order to make the sensing distance more accurate across all sensors, software calibration is performed before each round of the game. To reduce the current consumption of the LEDs we decided to modify the circuit to allow modulation to the LED power supply. Instead of leaving the LEDs on all the time, the microcontroller only turns them on right before it polls the sensor data.

5.2 Dot Counting Sensor

Dot counting is an important feature for the Pac-Man game. It allows Pac-Man to detect the dots, which are red LEDs, on the maze and to keep score of the number of dots that it has collected. The circuit is simple and consists of one sensor, which is a photoresistor, three resistors and an op-amp as shown in Figure 24 in the following.

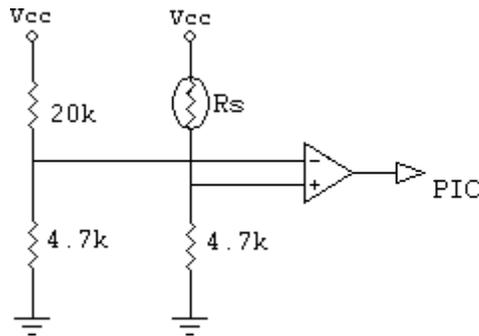


Figure 24. Dot Counting Circuit

The sensor placed at the bottom of Pac-Man, approximately 3cm away from the floor of the maze. When there is no light, the voltage measured across the positive terminal is around 0.6V (see connection diagram above). When a red LED shines to the sensor, the measured voltage is approximately 1.1V. Therefore, the reference voltage is set to 0.96V by connecting a 20k Ω resistor and a 4.7k Ω resistor as shown in the diagram above. Since the LEDs emit a narrow beam of light, the sensor in the center of Pac-Man sometimes cannot detect the

red LED on the maze. Therefore, the dot counting circuit is improved by placing two sensors side by side.

The dot counting system should output a high signal if either one of the sensors or both sensors detect the LED. Therefore, the outputs of both sensors are sent through an OR-gate. The output signal from the OR-gate is then sent to the microcontroller.

5.3 Pac-Man Tracking

As Pac-Man roams round the maze, Ghost should be able to detect Pac-Man's presence when they are within close proximity. Several implementations were considered including radio frequency for transmitting absolute position data, overhead camera for image processing, and sonar for detecting the rebounded signal from Pac-Man. However, we decided that the best way to implement this functionality is to mount an infrared beacon on Pac-Man, and wide-angle infrared receivers on Ghost.

The basic concept is simple: Pac-Man transmits an infrared signal in 360-degree coverage. Depending on the varying signal intensity of the infrared receivers, Ghost should be able to determine the general direction and distance of Pac-Man.

The receiver modules we use are PNA4612 (Figure 25), which detects IR signal modulated at 38kHz. Not only is PNA4612 sensitive to a specific transmitter frequency, but also it is sensitive to an emitter wavelength at around 960nm. These features make PNA4612 particularly attractive since it effectively blocks off IR interference. The beacon detector on Ghost consists of four receiver modules arranged to detect signals from front, left, right and back. Testing results show that PNA4612's detection range is well beyond the 120cm requirement. Upon further testing, however, we discovered an unfavorable output characteristic. The active low output signal becomes an aperiodic pulse train when the IR source is around a 90cm range. The length of the on-period of such a pulsing output is inversely related to the distance between 80cm and 100cm until the output becomes a flat 5V beyond 120cm. In order to convert such an irregular output into a usable signal, we fed the output to a low-pass filter to turn the pulse signal into a smooth analog voltage level, which can be easily interfaced with the microcontroller's A/D converter. (See Figure 26)

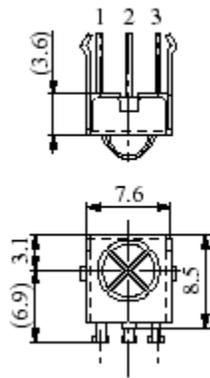


Figure 25. PNA4612

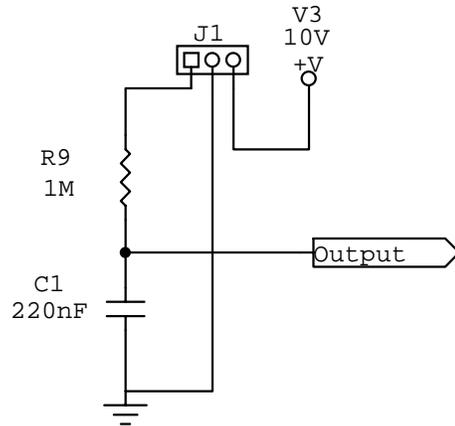


Figure 26. Beacon Receiver Circuit

The IR beacon module on Pac-Man, as shown in Figure 27, consists of a 555 timer used to produce a 38kHz square wave and IR emitters with a maximum output wavelength at 960nm. The resistor and capacitor values chosen were calculated by the following formula where T1 is the On period, and T2 is the Off period of the square wave. The duty circle is set at 60%:

$$T1 = 0.6 * (1/38000)$$

$$T2 = 0.4 * (1/38000)$$

$$T1 = 0.693 * (RA + RB) * C$$

$$T2 = 0.693 * RB * C$$

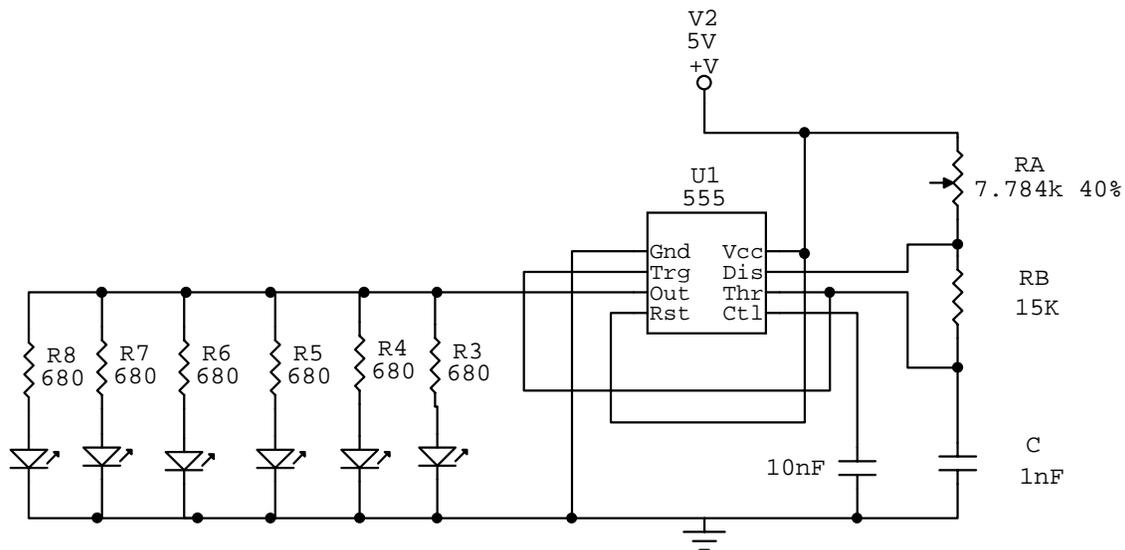


Figure 27. IR Beacon Circuit.

Since the beacon signal needs to cover 360 degrees but each emitter only has a 60-degree transmission angle, we arranged 6 emitters into a circle to form the IR beacon. Both IR beacon and IR receivers are positioned on the top of the robots above the wall, so they are not obstructed as the robots move around the maze.

5.4 Contact Sensor

The contact sensor on both Pac-Man and Ghost uses a simple pull-down circuit to notify the microprocessor upon collision of the two robots. The circuitry is as shown below in Figure 28. The contact sensor is designed to be a ring surrounding the base of the robots. The circular shape was chosen to provide an all-around, large surface area for contact. The ring is constructed with a flexible plastic support with brass strips mounted on it. The conductive strips on the two rings are designed to be offset, so the contact would be detected by both robots. The contact sensor is shown below in Figure 28.

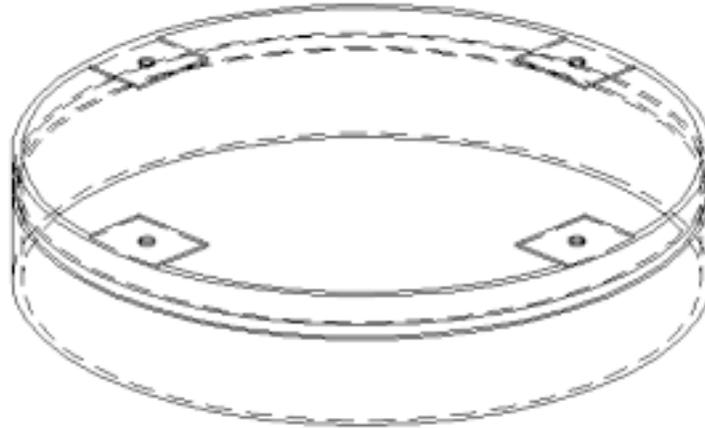


Figure 28. Contact Sensor

5.5 LCD display

The number of dots collected, as well as the number of times contacts are made are shown on the top layer of Pac-Man by an LCD display. Each time a dot is collected, the microcontroller will increment the dot count on the LCD display. The last two digits belong to the dot count, and the first digit is the contact count.

Originally we intended to use LED 7-segment displays; however, due to current consideration, we changed our design to the less current intensive LCD technology. Upon testing, we discovered that the segments on the LCD display fades quickly if we simply supply DC power to the segment pin. The solution is to use a square wave to drive the display segments allowing the LCD to discharge during the off period. Testing also shows that the frequency of the square wave

affects the quality of the display. At frequencies lower than 10Hz, the display flickers and at frequencies higher than 300Hz, the display begins to fade. As shown in Figure 29, we constructed another 555 timer to supply a square wave at around 100 Hz. Meanwhile, CD4026, a decade counter with 7-segment outputs, are used for each digit displayed. The 100Hz square wave is connected to the output enable pin of CD4026, to produce a square wave output that drives the LCD pins.

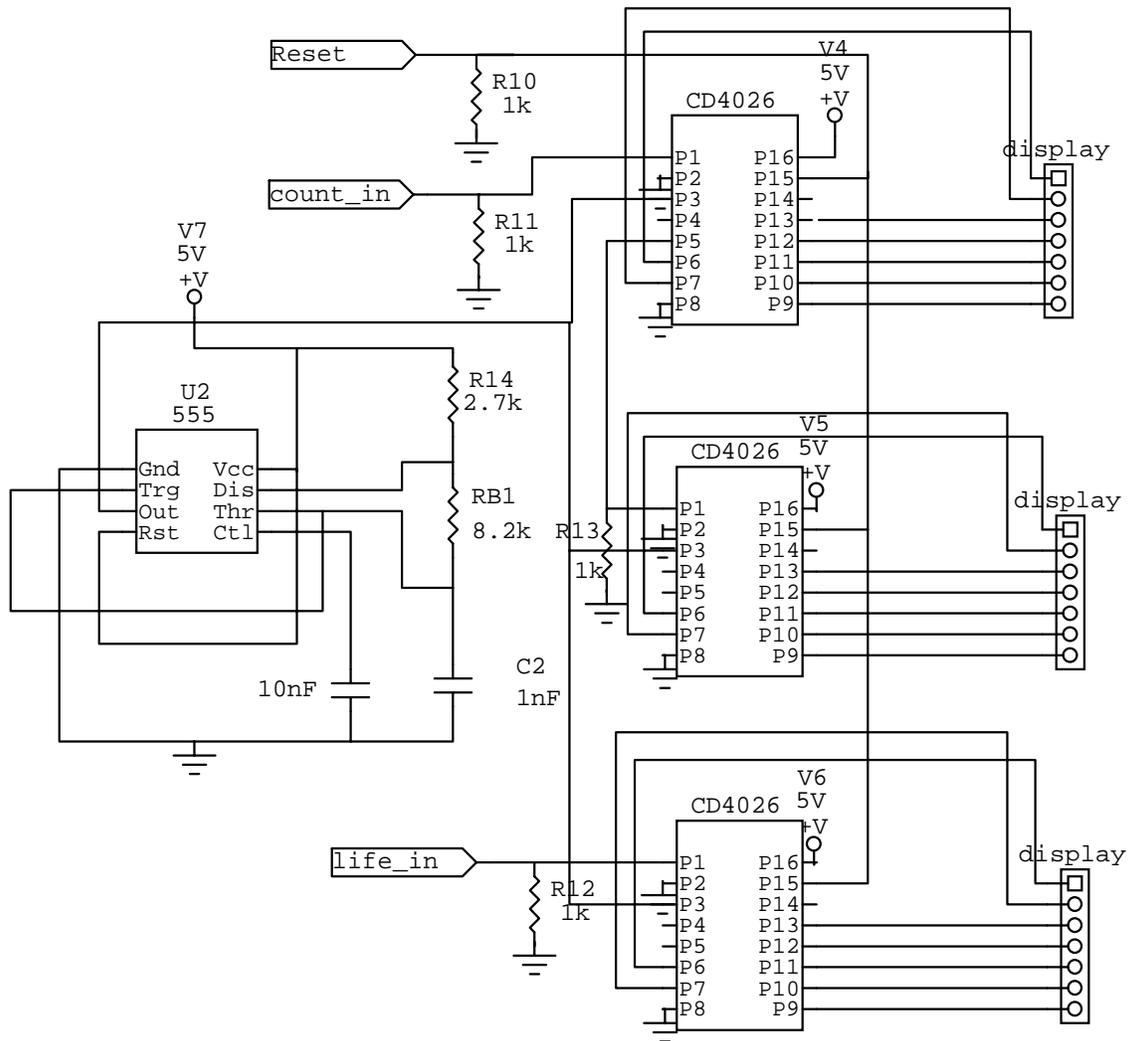


Figure 29. LCD display

6.0 Microcontroller and Software

Our microcontroller of choice was the Microchip PIC16F877. MicroCore-11 microcontrollers using the Motorola HC11 are provided by the lab but upon further consideration, they were rejected for a number of reasons. The project consists of two robots, each of which requires a microcontroller. One MicroCore-11 costs around \$100 and using two would already consume half of our budget constraint of \$400.

Financial issues aside, its functionality were also inadequate for our purposes. Interfacing with RF requires at least 3 pins to decode the four direction signals: up, down, left, and right, and also to indicate the presence of an RF signal. Two stepper motors are used on each robot and each stepper motor driver chip requires 2 pins for stepping and direction, for a total of 4 pins. An LCD display uses 3 pins for counting lives, score, and reset. Ghost contact and maze-dot counting require 1 pin each for a total of 2. Obstacle sensors use LEDs and this requires a minimum of 1 pin to power if we power them all together. This already totals to a requirement of 13 pins, which exceeds the 12 digital pins available on the MicroCore-11.

While searching for a microcontroller with a high pin count and minimal cost, we came across the Microchip PICmicro MCU series, with up to 32 I/O pins in the 40-pin package and a retail price of around \$10, one-tenth of the price of a MicroCore-11. Of the PICmicro series, we picked the PIC16F877 based on its popularity (a large amount of information and support from hobbyists were found on the Internet) and because it had the largest flash memory size of its class.

Drawbacks of the PIC16F877 are that its memory size is slightly small at 8K x 14 words, (although it is expandable externally) and because it is not packaged in a module like the MicroCore-11. Some of the external circuitry required on the PIC16F877 was a clock oscillator, which we chose to be at 4 MHz and a programmer to write to the flash memory. The advantages, however, outweigh the drawbacks for our purposes even though we may start off more slowly because the programming could not begin until we built the programmer.

There are essentially two software designs in the Pac-Man project—one for the Pac-Man robot and one for the Ghost. Though they are very different in functionality and as a result, in implementation, they both play the central role in system integration. Each robot also interfaces to different hardware modules and so these modules are discussed as separate software functions below.

6.1 Pac-Man Robot

The goal of the Pac-Man robot is to receive from the user wireless control signals to move it around the maze. Obstacle sensors guide its navigation. As it traverses through the maze, it collects and counts the dots on the floor and displays it on an

LCD. If it comes in contact with the Ghost robot, it will freeze in shock for a few seconds before recovering.

Six modules are identified in the Pac-Man robot that help it achieve its above goal.

- Obstacle sensors (front, left, right) prevent Pac-Man from running into maze walls and help straighten itself if it is off-centred.
- An LCD display shows the number of dots Pac-Man has collected and the number of times it has been caught by the Ghost.
- RF controls the direction of movement of Pac-Man around the maze.
- Stepper motors perform the actual movement of the robot.
- The maze-dot sensor looks for the LEDs on the floor that it will “collect.”
- The Ghost contact sensor notifies the event of being caught by the Ghost.

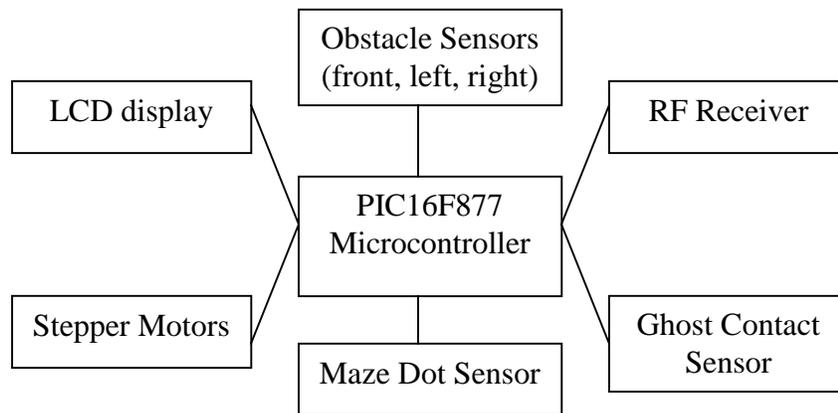


Figure 30. Architectural model of Pac-Man software

Given these modules, the pin assignments they require are laid out as follows. Note that the RF signal, Ghost contact, and maze-dot sensor are events that can happen at any point during the game so the pin assignments are chosen on pins 38-40 such that they are handled by an interrupt service routine. Also, pins 2-9 are analog inputs so that right, left, and front obstacle sensors can estimate the distance from a wall as a function of the voltage input.

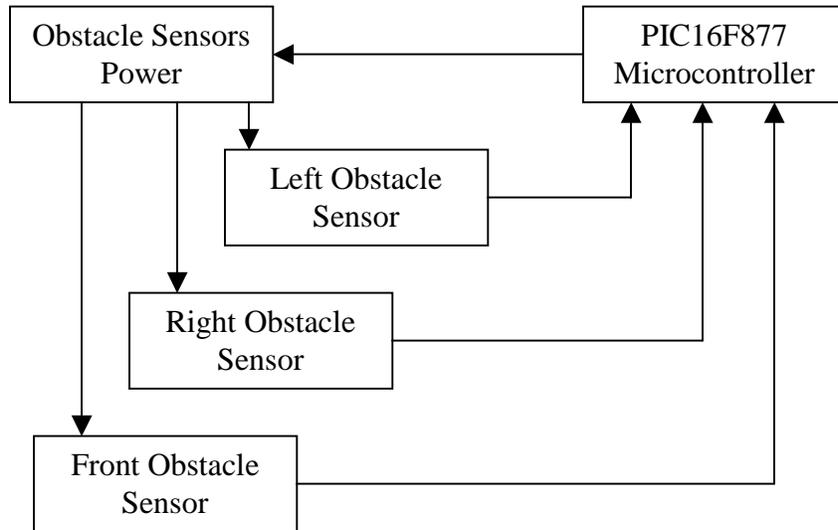


Figure 32. Model of interface between PIC and obstacle sensors

6.1.2 LCD Display

The PIC sends three types of signals to the LCD depending on the event. In the case of a Ghost contact, the PIC sends an LCD lives signal indicating the number of times it has been hit. When Pac-Man runs over a maze dot, it sends an LCD point signal indicating the number of dots it has collected. Finally, an LCD reset signal is for initializing the display when a new game begins.

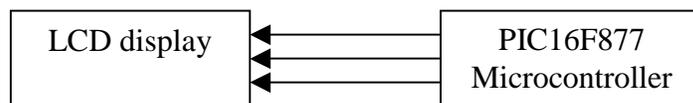


Figure 33. Model of interface between PIC and LCD display

6.1.3 RF Receiver

The RF receiver interfaces with the PIC's interrupt service routine because the user can press any button on the controller at any point in the game. This scenario was chosen because it was vital that the PIC did not miss any of the control signals, which may have resulted if the RF signals were received through polling.

Three bits are used to communicate the four different directions. The format is listed below.

Table 2. RF signal format for Pac-Man movement

	RF Interrupt	RF Bit 1	RF Bit 0
No signal	0	0	0
North	1	0	0
East	1	0	1
West	1	1	0
South	1	1	1

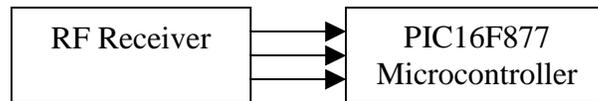


Figure 34. Model of interface between PIC and RF receiver

6.1.4 Stepper Motors

The PIC controls each stepper motor through two pins to the stepper motor driver chip. One pin controls the stepping pulse to the motor, hence controlling the speed, and the other controls the direction of the rotation, used in making stationary turns.

The pulsing code is done through a timer interrupt routine, whose 16-bit timer counts at a rate of 1 MHz (one-quarter of our clock speed). When the interrupt is called, we toggle the stepping pulse pin value from high to low or from low to high to simulate a square wave output.

An interrupt is triggered when this 16-bit timer overflows from 0xFFFF to 0x0000. Hence, an interrupt is by default called at a rate of

$$1 \times 10^6 \frac{\text{counts}}{\text{second}} / 2^{16} \frac{\text{counts}}{\text{interrupt}} = 15.26 \frac{\text{interrupts}}{\text{second}}$$

the interrupt rate by setting the timer value to some number greater than 0x0000 at every interrupt so that it counts from that number up to 0xFFFF instead of 0x0000 every time.

Our target speed for Pac-Man is 20 cm/s. To determine the rate of interrupts, we required some information about the physical robot dimensions. They are summarized in the table below.

Table 3. List of parameters needed to control motor speed

Separation distance of wheels	11.5 cm
Diameter of wheels	6 cm
Degrees per step of stepper motor	7.5

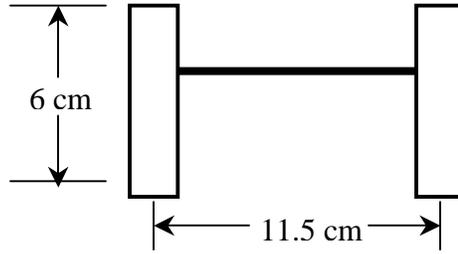
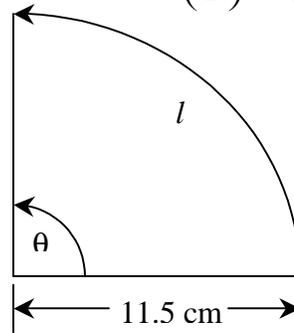


Figure 35. Sketch of wheel dimensions

To make a 90-degree turn, the distance travelled by one wheel was determined. Since the distance between the wheels is 11.5 cm, the distance travelled by the arc length l when θ is 90 degrees can be

calculated. Therefore, $l = r\theta = (11.5)\left(\frac{\pi}{2}\right) = \frac{23}{4}\pi$ cm.



To make a stationary turn, the distance l to be travelled is distributed between the two wheels as $l/2$ because each wheel will turn 45 degrees. One wheel will turn 45 degrees forward, and the other will turn 45 degrees backward. Therefore, each wheel travels $\frac{23}{4} \times \frac{1}{2} = \frac{23}{8}\pi$ cm.

To find the number of stepper pulses needed to turn each wheel 45 degrees, we calculate the number of pulses as

$$\frac{23}{8}\pi \text{ cm} \times \frac{1 \text{ rotation}}{6\pi \text{ cm}} \times \frac{360^\circ}{1 \text{ rotation}} \times \frac{1 \text{ pulse}}{7.5^\circ} = 23 \text{ pulses}$$

Because our target speed is 20 cm/s, we calculate the number of pulses per second required as

$$\frac{20 \text{ cm}}{\text{second}} \times \frac{1 \text{ rotation}}{6\pi \text{ cm}} \times \frac{360^\circ}{1 \text{ rotation}} \times \frac{1 \text{ pulse}}{7.5^\circ} \approx \frac{51 \text{ pulses}}{\text{second}} = \frac{19.6 \text{ ms}}{\text{pulse}}$$

To achieve a speed of 20 cm/s, we need to create a square wave of about 51 Hz. That means we need to toggle the stepper output twice every 19.6 ms or in other words, generate interrupts to toggle at $19.6 / 2 = 9.817$ ms.

We know that the timer counts at 1 MHz so that in 9.817 ms, the timer will do $1 \text{ MHz} \times 9.817 \text{ ms} = 9817$ counts . If we set the timer value to be $0\text{xFFFF} - 9817 = 55718 = 0\text{xD9A6}$ at every interrupt, the timer will overflow every 9.817 ms, creating an interrupt in which we will toggle the stepper output to create a 51 Hz square wave.

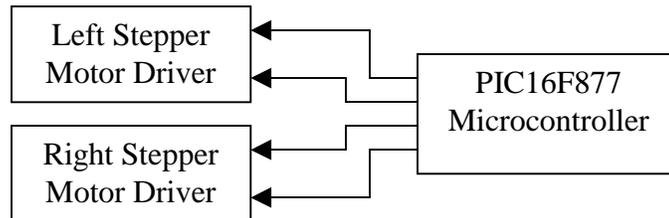


Figure 36. Model of interface between PIC and stepper motor drivers

The diagram above shows how the PIC interfaces to the stepper motors.

6.1.5 Ghost Contact Sensor

The Ghost contact sensor is connected to an interrupt pin on the PIC because the Ghost can catch Pac-Man at any point during the game. This prevents polling which is susceptible to missing the signal and holds up CPU time. The following diagram models how the Ghost contact sensor is connected to the PIC.

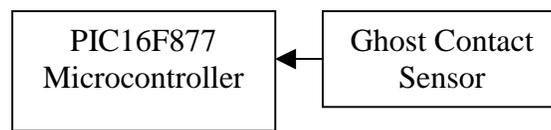


Figure 37. Model of interface between PIC and Ghost contact sensor

6.1.6 Maze Dot Sensor

The maze-dot sensor is connected to an interrupt pin on the PIC for the same reasons as the Ghost contact sensor. The following diagram models how the Maze dot sensor is connected to the PIC.

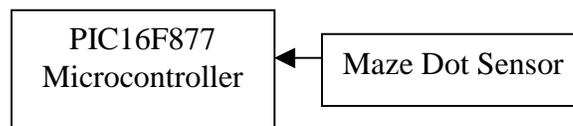


Figure 38. Model of interface between PIC and maze dot sensor

6.2 Ghost Robot

The goal of the Ghost robot is to roam the maze reading its beacon signal to estimate the current location of Pac-Man. Having knowledge of the maze in advance, the Ghost solves for the shortest path to reach Pac-Man and moves to the estimated position. It constantly does this until it contacts Pac-Man at which point it will freeze for a few seconds to allow Pac-Man to escape.

Four modules are identified in the Ghost robot that help it achieve its above goal.

- Obstacle sensors (front, left, right) prevent the Ghost from running into maze walls and help straighten itself if it is off-centred.
- The beacon sensor reads a broadcast signal from Pac-Man and uses it to estimate the current position of Pac-Man
- Stepper motors perform the actual movement of the robot.
- The Pac-Man contact sensor notifies the event of having caught Pac-Man.

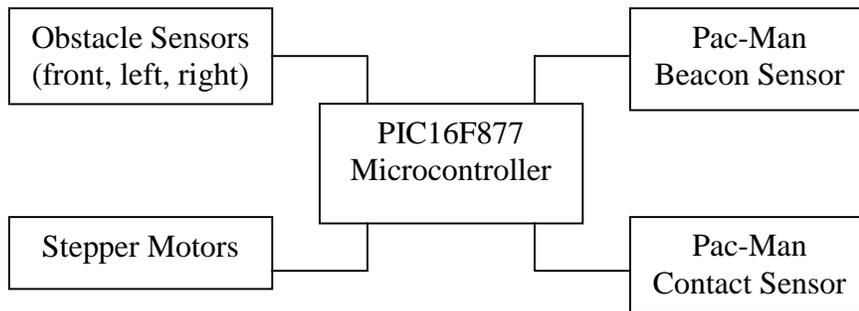


Figure 39. Architectural model of Pac-Man software

Pin assignments for the Ghost are similar to that of Pac-Man except for the modules that the Ghost do not have and the beacon sensor. There are four beacon sensors, one to detect each of the four directions Pac-Man may be away from the Ghost. They use analog pins 7, 8, 9, and 10.

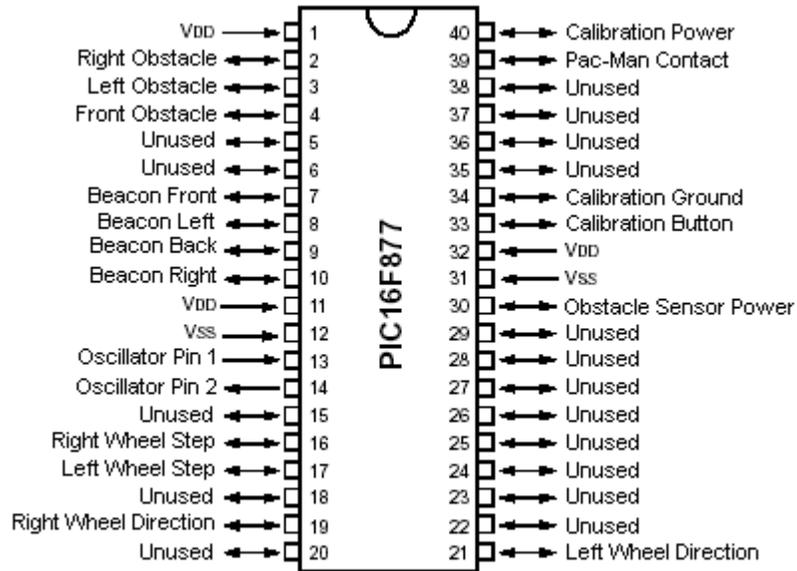


Figure 40. Pin assignments for the Ghost

Because the obstacle sensor, stepper motors, and Pac-Man contact sensor are identical to that of Pac-Man's, they require no repeat explanation here. We will discuss the only thing unique to the Ghost in this section, the beacon sensor.

6.2.1 Beacon Sensor

The PIC receives four different beacon readings from the Pac-Man beacon broadcast signal, one for each of its four directions. It uses these readings to estimate the position of Pac-Man. The interface for these sensors to the PIC is modelled below.

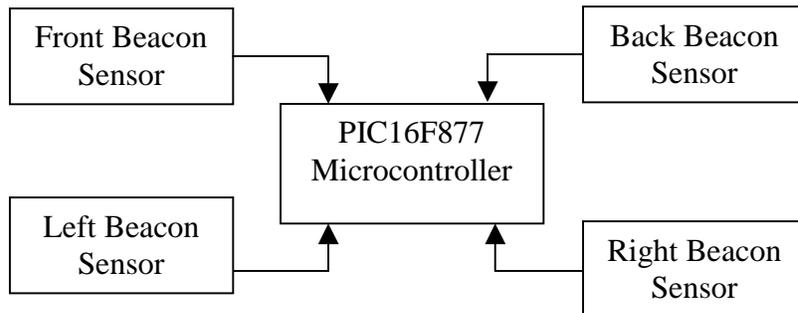


Figure 41. Model of interface between PIC and beacon sensors

The beacon signal strength is a negative, non-linear function of Pac-Man's distance from the Ghost. For a 6x6 maze, we measure and plot the signal strength of the four beacon signals as follows.

4.6 V	4.6 V	4.6 V	4.6 V	4.6 V	4.6 V
4.6 V	4.6 V	4.6 V	4.6 V	4.6 V	3.55 V
4.6 V	4.6 V	4.6 V	4.6 V	4.5 V	0.19 V
4.6 V	4.6 V	4.6 V	4.6 V	0.33 V	0.05 V
4.6 V	4.53 V	4.6 V	4.6 V	0.6 V	0.05 V
4.6 V	4.6 V	4.6 V	4.6 V	4.6 V	Ghost 

Figure 42. Front beacon measurements of Pac-Man position

4.6 V	4.6 V	4.6 V	4.6 V	4.6 V	4.6 V
4.6 V	4.6 V	4.6 V	4.6 V	4.6 V	4.6 V
4.6 V	4.6 V	4.6 V	2.6 V	4.6 V	4.6 V
4.6 V	4.6 V	4.6 V	4.6 V	4.6 V	4.6 V
4.6 V	4.53 V	3.2 V	0.07 V	0.6 V	4.3 V
4.6 V	4.4 V	2.6 V	0.04 V	0.04 V	Ghost 

Figure 43. Left beacon measurements of Pac-Man position

The measurements for the right and back are similar to that of the right and front and so we will rely on the above two measurements to determine Pac-Man's position. The measurements are very crude because the non-linear signal changes very fast for distances of Pac-Man between 2 and 3 cells away but we can generalize a few things about beacon sensors.

- For readings from 0 V to 0.08 V on one sensor and 4.6 V on the other three, we will assume Pac-Man is in direct line-of-sight and is 1 or 2 cells away in the direction of that sensor. We will assume 2 cells away unless that would place it beyond the boundaries of the maze.
- For readings from 0.08 V to 3.8 V on one sensor and 4.6 V on the other three, we will assume Pac-Man is in direct line-of-sight and is 3 cells away in the direction of that sensor.
- For readings from 3.8 V to 4.5 V on one sensor and 4.6 V on the other three, we will assume Pac-Man is in direct line-of-sight and is 4 cells away in the direction of that sensor.

We will discard diagonal signal readings because of the added complexity to the estimation algorithm and justify it by reasoning that because Pac-Man is almost always constantly moving, it will cross the Ghost's direct line-of-sight frequently enough for the estimation to be valid.

From the estimation of Pac-Man's position, the Ghost solves for the shortest route to Pac-Man's position from its own current position using an algorithm called fast flooding, based on the Micromouse Information Centre at <http://micromouse.cannock.ac.uk/maze/fastfloodsolver.htm>. This algorithm marks the target goal, in this case Pac-Man's estimated position, with a flag, and fills accessible neighbours with ascending numbers until all the cells have a number associated with it. The Ghost then retrieves the number associated with its current position and moves to adjacent cells only if they have a lower number associated with it.

The example below illustrates the algorithm. It was taken from the Micromouse Information Centre in the above noted link. In this example, G is the destination goal or in our case, Pac-Man's estimated position, and S is the starting cell or in our case, Ghost's current position. Note how accessible adjacent neighbours are filled with increasing numbers. From this, the robot will move to adjacent cells of decreasing numbers.

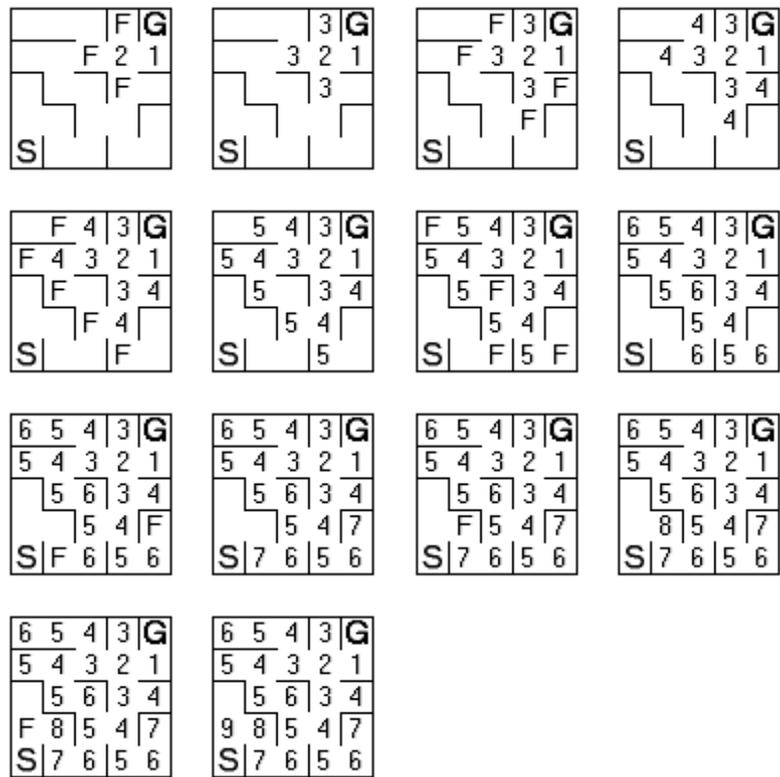


Figure 44. Example of the maze fast flooding algorithm

7.2 Dots

The dots on the maze are implemented with CMOS chips and a simple circuit. The following is the connection diagram for one dot on the maze. In general, a red LED is ‘ON’ at the beginning and after being reset. Then, it is turned off when a sensor corresponded to the red LED senses a super bright white LED that is placed at the bottom off the Pac-Man robot.

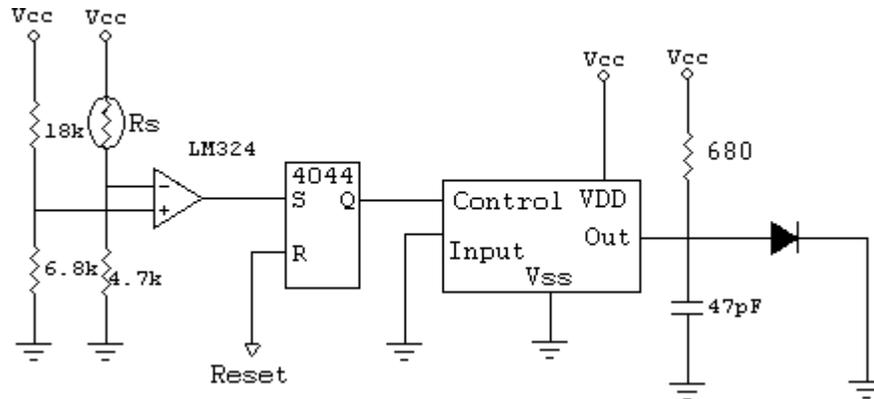


Figure 46. Dot Circuitry for the Maze (for one dot)

The first part of the dot circuit is a comparator. The comparator outputs low when light with certain intensity is sensed; otherwise, it outputs high. As shown in the above diagram, the reference voltage is connected to the positive terminal and the photoresistor along with a 4.7k resistor, is connected to the negative terminal. When the sensor senses light, its resistance decreases. As a result, the voltage to the negative terminal increases. When the negative terminal voltage is higher than the positive terminal voltage, the comparator outputs zero.

$$V^- = \frac{R_s}{4.7k + R_s}$$

The voltage measured across the negative terminal is around 2.4V under the condition that all the lights in the room are turned on. When a super bright white LED is located on top of the photoresistor, the measured negative terminal voltage is in a range from 3.8V-4.1V. Therefore, the reference voltage that is connected to the positive terminal is set to 3.62V. The design is good because it is sensitive only to the light emitted by the super bright white LEDs and it is stable.

The second part of the dot circuit is a memory device. After a light is sensed, it turns off a dot on the maze and saves the “OFF” stage until users reset the dot. A NAND R/S latch (CD4044) is chosen. The following is the truth table for the memory device.

Table 4. Truth table of the Memory Device

	S	R	Q
Start	1	1	0
No light	1	1	0
Light	0	1	1
No light	1	1	1
Light	0	1	1
Reset	0	0→1	0
Start	0	1	1
No light	1	1	1
Reset	1	0→1	0

The output from the comparator is connected to 'S' and a dipswitch is connected to 'R'. 'R' must be high while operating. When 'R' is switched to low, all the red LEDs on the maze will be turn on again.

The last part of the dot circuit is a switch. CD4066, a bilateral switch is chosen. The output of the memory device is connected to 'Control' in the CD4066. When 'Control' is low, the dot is 'ON'; when 'Control' is high, the dot is 'OFF' (see Figure 47 below).



Figure 47. Timing diagram of the Switch (including CD4066, resistor, and capacitor)

As shown in the above diagram, when 'Control' is low, the output of switch is equal to VDD. However, when 'Control' is high, the output voltage never reaches 0V yet the output voltage is low enough to turn off the red LED.

The dots on the maze are powered by a power supply in the lab. In order to make the LEDs brighter, we inputted 12V to the dot circuit.

8.0 Power

8.1 Motor

Our stepper motors require an output voltage of 12V to operate. Since this was our highest voltage requirement, we used 12V as our base voltage and used regulators to supply power to the rest of the circuits. Luckily, we were able to use one regulator to power all of the circuits since they all required the same input voltage of 5V.

For our battery selection, we decided to use a rechargeable 12V Lead-Acid battery with a rating of 1200mA·h on Pac-Man and 10 rechargeable 1.2V Nickel-Hydride AA batteries with a rating of 1600mA·h on the Ghost. We recommend in the future that the AA batteries be used since they have a longer life and are lighter in weight in comparison to the Lead-Acid battery. (The AA batteries were not used on both robots due to our budget restraints.)

While testing the motors at low frequencies, we found that it was essential to have a well-regulated voltage from the battery source. In order to ensure proper operation, we placed a decoupling capacitor across the 5V power and ground pins of the motor driver chips and across the 12V battery terminals.

8.2 Controller

The transmitter module for the controller circuit required a clean and well-regulated voltage to operate efficiently. A low pass filter was added to the transmitter module and a bypass capacitor to the logic ICs in cases when the quality of the power supply is poor. Due to the transmitter's low power consumption (6mA), we were able to use a rechargeable 9V Nickel-Hydride battery to power the module and the LM7805 5V regulator to power the other various components in the controller. The total amount of current consumed by the controller is 10mA, which is well below the battery rating of 150mAh. We found that this was enough to power the controller for several hours.

8.3 Maze

The LEDs and sensors placed on the maze are powered off the power supply since we opted that it would not be necessary to power them off a battery. The voltage supplied to the LEDs, sensors and ICs were set to the maximum voltage rating of 15V. We wanted to maximize the voltage because we found that the brighter the LEDs on the maze, the easier it was for Pac-Man to detect it.

8.4 Regulators

Regulators were added to provide a constant output voltage to the variable load on both robots. On Pac-Man, we used a LM7805 5V regulator to power all the circuitry. This included the RF receiver module, the obstacle sensors, contact sensor, micro-controller, stepper motor driver chips, the LCD, and the beacon circuit. The total amount of current consumed by all circuits is less than 1A, which was enough to meet the regulator's specifications.

On the Ghost, we decided to experiment with a 5V switching voltage regulator to send power to the micro-controller, obstacle sensor, contact sensor and beacon circuits. See Figure 48 for the circuit set-up. The switching regulator contains a 52kHz internal frequency oscillator that allows it to output the input power as a pulse. The regulator controls the pulse duration by using feedback as shown in Figure 49. The width of the pulse changes based on the amount of output power required. When the output power is small, the pulse duration is narrow and when the output power is large, the pulse duration is wider.

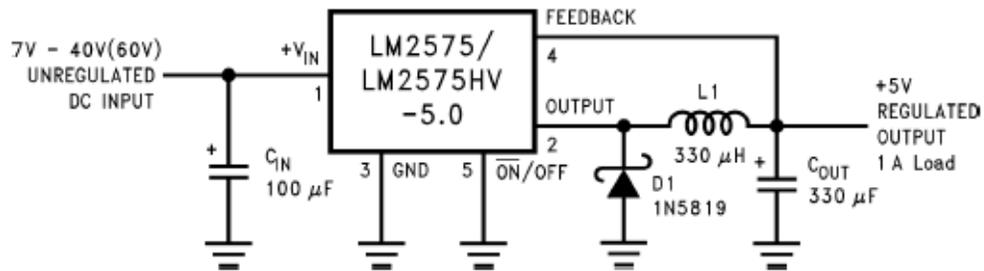


Figure 48. LM2575 Circuit Schematic

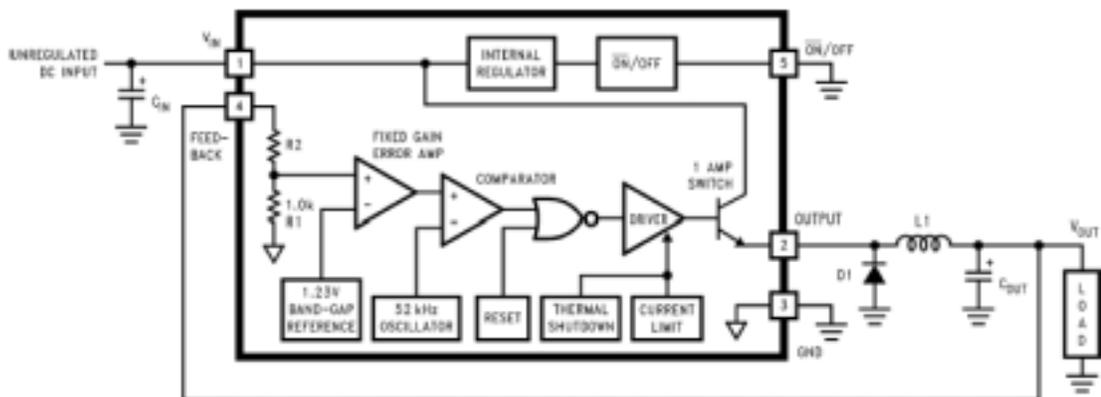


Figure 49. LM2575 Circuit Block Diagram

Due to its switching operation, a switching regulator has higher efficiency than three pin regulators (used on Pac-Man). A three pin regulator has lower efficiency because its input power is equal to its output power at all times whereas for a switching regulator, the output power is usually less than the input power since the amount sent out is determined by the amount of power required by the external circuits. We recommend using a switching regulator on both robots in order to obtain higher efficiency for future considerations. A switching regulator was not implemented on Pac-Man due to time constraints and limited resources.

9.0 Recommendations

RF

In order to remove as much circuitry on Pac-Man as possible, we recommend designing a two-way communication link between the controller and Pac-Man. The LCD displaying the score and lives of Pac-Man could be displayed elsewhere in order to reduce the circuitry on Pac-Man, in addition to conserving power on the robot.

Wall sensors

A plastic casing can be built to improve the stability and consistency of the sensors.

Beacon

Currently there are four IR receivers on Ghost allowing it to detect Pac-Man in four directions. For added tracking capability, two more receivers can be added to achieve 360-degree detection.

LCD

The LCD display used in our Pac-Man project is a simple 3-digit numeric display. For more display options, full-featured LCD modules are available at a reasonable price around \$20.

Microcontroller

Having designed an on-board programming interface directly on the PCB instead of having to take the PIC in and out of the socket would have made reprogramming the PIC with new code easier and faster, and also reduce physical damage to the PIC itself. This is supported through the In-Circuit Serial Programming (ICSP) interface specified by Microchip.

Also, by modulating the obstacle sensor output and filtering the output from 60 Hz, electrical line noise could have reduced the unpredictable interference from ambient room light, and made sensor readings and navigation more reliable.

Finally, putting the obstacle and beacon sensor output through a properly designed non-linear amplifier could have made the sensor output reading more linearly proportional to the distance instead of how we have it dropping inverse square (or cube) to the distance.

Motor

When it comes to choosing motors for your robot, the most important factor is the torque. Therefore, it is best to have estimated how much torque your robot will require before deciding on which motor to use. The required torque can be calculated easily through Newton's second law, $F = ma$, in combination with your desired wheel size. Torque is typically specified using the following three units: g-cm (gram-centimeter), oz-in (ounce-inch), and mNm (milli-Newton-meter).

Power

As mentioned in the report, a switching regulator should be used on both robots in place of the three-pin regulator to increase the efficiency of the power used. In addition, we recommend that a separate battery be used to power the logic circuits and RF modules since these devices, especially the RF, are sensitive to any glitches in the power supply.

10.0 Conclusion

The design and implementation of a Pac-Man game proved to be a challenging and rewarding experience. While keeping the objectives of the EECE474 course in mind, we were able to successfully complete the integration of two robots – Pac-Man and Ghost - using wireless communication, sensors, motors, software, power and circuit design.

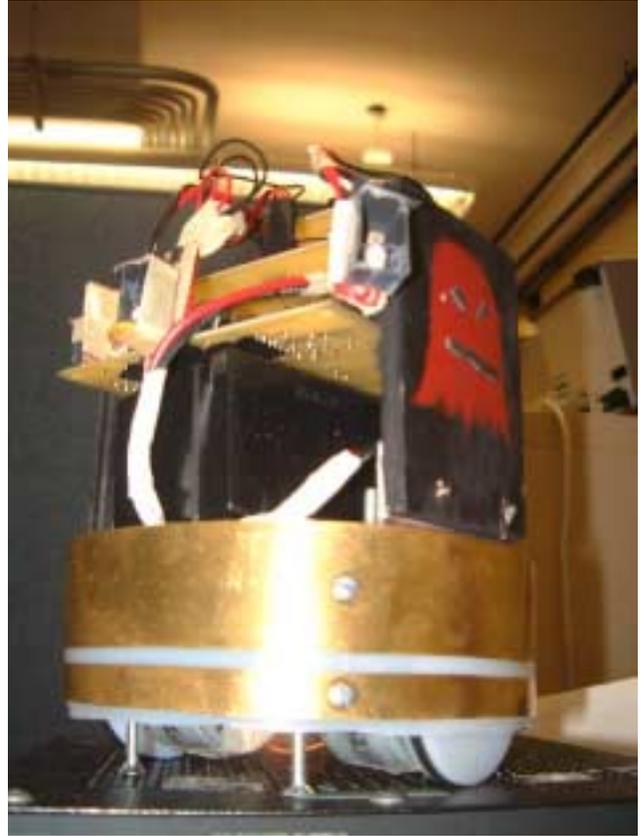
After completing numerous tests and enhancements on our designs, the robot Pac-Man is able to traverse the maze following directional signals sent from a wireless user controller while the Ghost is able to automatically track Pac-Man down to the shortest path from one location to another in the maze. Both robots are capable of aligning themselves to the middle of a lane and complete 90 degree and 180 degree turns while avoiding contact from walls. Moreover, Pac-Man successfully is able to count and keep score of the number of dots it has collected and turn the dots off on the maze once they have been “eaten”.

APPENDICES

APPENDIX A: Photo Gallery



Pac-Man



Ghost

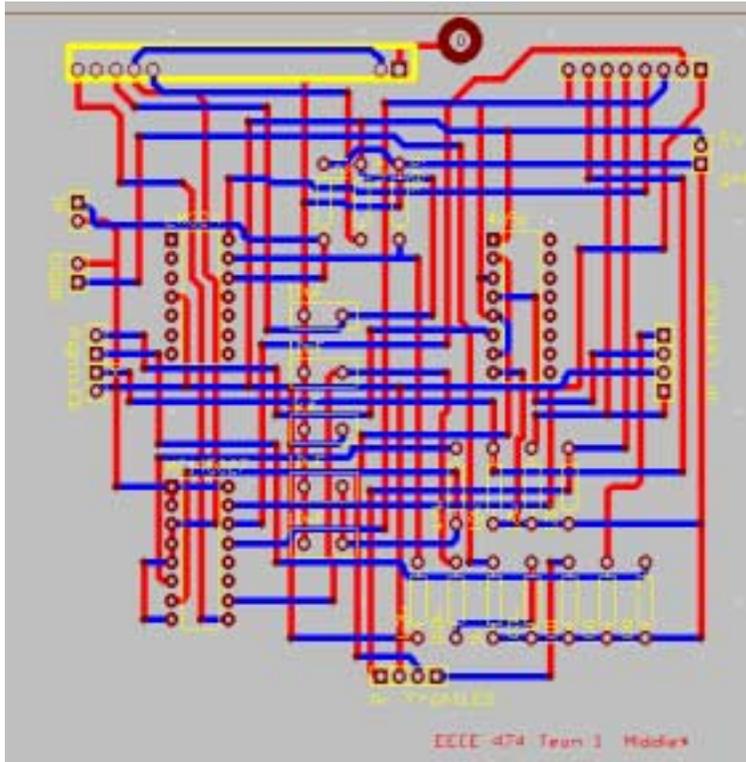


Wireless Controller

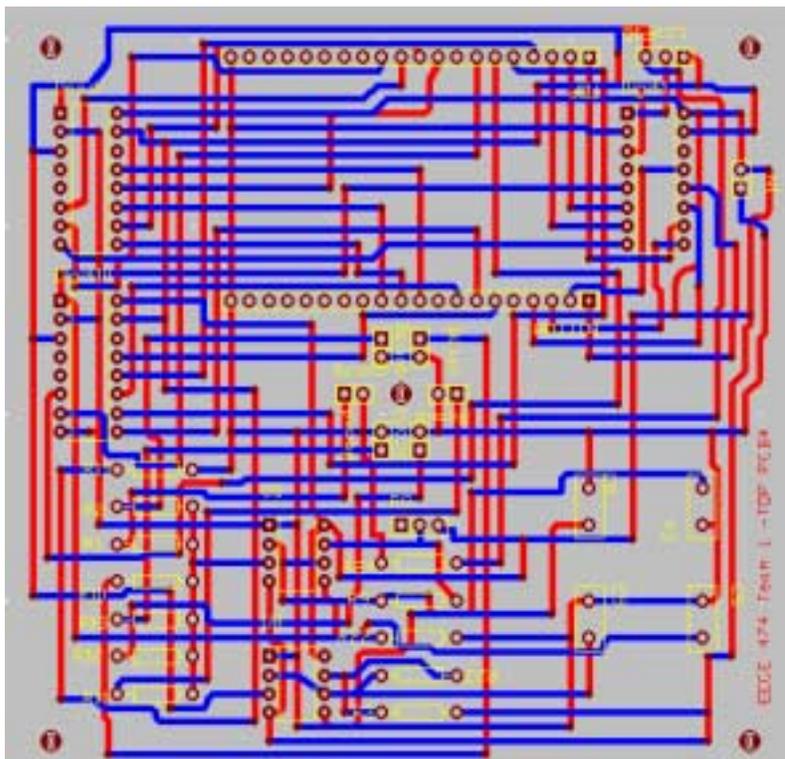


The game setting

Middle Layer (RF module, obstacle sensors, dot counting)

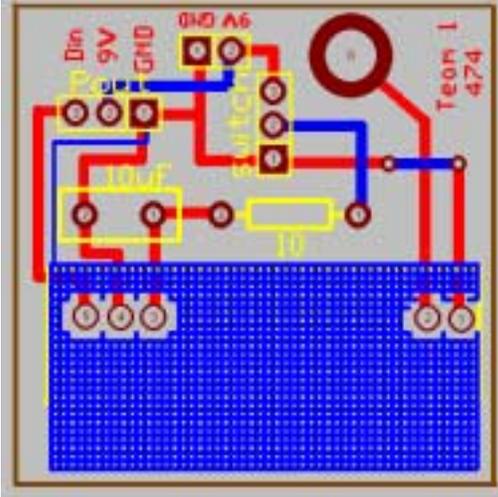


Top Layer (beacon, LCD)

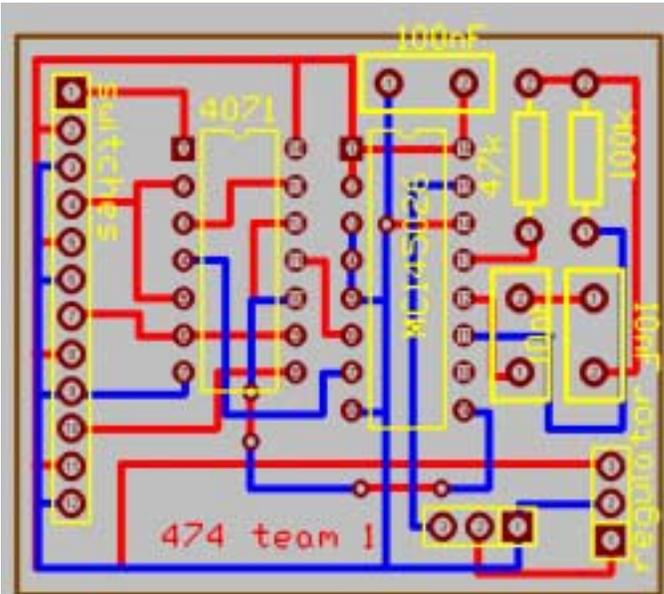


Controller

RF module



Encoder and Logic IC

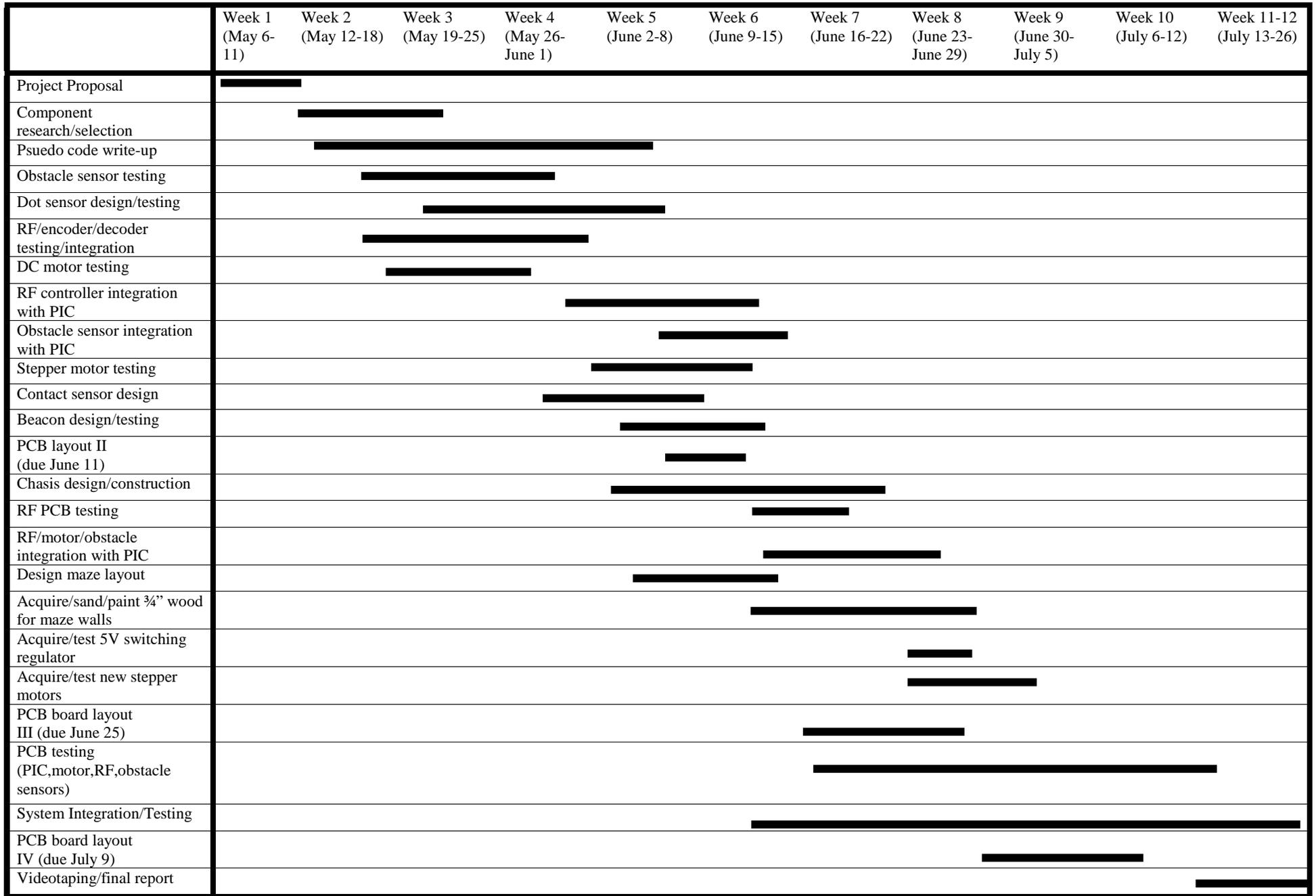


APPENDIX C: Expense Report

Quantity	Part	Source	Price	Amount
2	¼ Wave Whip 418MHz Antenna	DigiKey	\$3.29	\$6.58
1	RF Transmitter 418MHz	DigiKey	\$36.41	\$36.41
1	RF Receiver 418MHz	DigiKey	\$59.46	\$59.46
4	Stepper Motor 12VDC	DigiKey	\$32.15	\$128.60
1	LM2575T-5.0 Switch Regulator	DigiKey	\$4.27	\$4.27
6	Photo IC infrared 38kHz	DigiKey	\$2.93	\$17.58
4	SPDT Momentary Switches	Active Electronics	\$7.49	\$29.96
2	PIC16F877	Active Electronics	\$10.79	\$21.58
4	UCN5804B Stepper Motor Driver	HVW Technologies Inc.	\$10.00	\$40.00
1	12V Lead Acid Battery	RadioShack	\$10.05	\$10.05
9	LEDs	Lee's Electronic Components	\$3.42	\$30.78
30	Photoresistors	Lee's Electronic Components	\$1.00	\$30.00
*	Passive components	EE Department		\$40.00
**	Active components	EE Department		\$30.00
TOTAL				\$485.27
10	AA 1.2V Batteries	Donated		\$40.00
2	2 castors	Donated		\$15.00
TOTAL				\$55.00
GRAND TOTAL				\$540.27
*	resistors,capacitors,inductors,diodes,LEDs			
**	MC145026, MC145027, LM555, CD4071, LM324,LM7805, CD4044, CD4066			

-Cost of donated items were estimated

APPENDIX D: Gantt chart



APPENDIX E: Source Code

Pac-Man Header File

```
/*
 * EECE 474 Summer Semester 2002
 * Team 1 Pac-Man Microcontroller Code
 * =====
 * Pac-Man code header file
 */

// RF Constants
#define RF_0          PIN_B0 // RF LSB pin
#define RF_1          PIN_B1 // RF MSB pin
#define RF_ON         PIN_B7 // RF incoming signal interrupt pin

// Obstacle Sensor Constants
// Note: higher sensor readings mean closer obstacle
#define SENSOR_LED    PIN_D7 // input pin powering sensor LEDs
#define SENSOR_VALID_DELAY 10 // time in ms for sensor read valid from sensor LED on
#define OBSTACLE_LEFT 1 // analog channel AN0
#define OBSTACLE_RIGHT 0 // analog channel AN1
#define OBSTACLE_FRONT 2 // analog channel AN2

// Other Sensor Constants
#define CONTACT        PIN_B6 // Ghost contact sensor pin
#define DOT_COUNT      PIN_B5 // maze dot eating/counting sensor pin

// Motor Control Constants
#define R_WHEEL        PIN_C1 // right wheel motor stepping pin
#define L_WHEEL        PIN_C2 // left wheel motor stepping pin
#define RIGHT_ANGLE_TURN 43 // number of pulses on one wheel to turn 90 degrees
#define BACK_TURN      84
#define L_WHEEL_DIR    PIN_D2 // left wheel direction pin
#define R_WHEEL_DIR    PIN_D0 // right wheel direction pin
#define L_FORWARD      1 // left wheel direction pin low is forward
#define L_BACKWARD     0 // left wheel direction pin high is backward
#define R_FORWARD      0 // right wheel direction pin high is forward
#define R_BACKWARD     1 // right wheel direction pin low is backward
#define ADJUST_AMT     1 // number of pulses to slow wheel down by when adjusting for off-
centre

// LCD Constants
#define LCD_RESET      PIN_D4
#define LCD_LIVES      PIN_D6
#define LCD_POINTS     PIN_D5
#define DOT_COUNT_EXPIRE 20000

#define CELL_SIZE      102 // number of pulses to traverse one step

enum direction { N, E, S, W };
enum turn { RIGHT, LEFT, BACK, NONE };

// Function Prototypes
void turn_right();
void turn_left();
void turn_backwards();
void stepper_timer();
void signal_change();
void read_obstacles( int &left, int &front, int &right );
```

Pac-Man Source Code

```
/*
*****
*/
EECE 474 Summer Semester 2002
/*
*****
*/
Team 1 Pac-Man Microcontroller Code
/*
*****
*/
This program is for controlling the Pac-Man robot. It is designed
for a Microchip PIC16F877 running with an oscillator frequency of 4
MHz. It interfaces with the obstacle, Ghost contact, and dot-eating
sensors, and radio frequency control to control the robot's stepper
motors for movement and the LCD display for user interface.
/*
*****
*/

#include <16F877.h>
#include "Pac-Man.h"

//device adc=10
#use delay (clock=4000000)
#fuses XT, NOWDT, NOPROTECT

int ob_left, ob_right, ob_front;
short rf_enable = 0;
short turn_enable = 0;
int turn_counter = 0;
short turn_180 = 0;
short stop = 0;
int adjust_counter = 0;
short adjust_left = 0, adjust_right = 0;
short dot_count_timer = 0;
int dot_count_counter = 0;
direction current_orient = N;
short initial_start = 0;
int cell_counter = 0;
short turn_ready = 0; //used for pre-sigaled turns
int turn_delay_count = 0; // used to delay pre-sigaled turns
turn next_turn = NONE;

int calibration_stage = 0;
int FRONT_OBSTACLE = 105;
int MAX_RIGHT = 0;
int MAX_LEFT = 0;
int MIN_RIGHT = 0;
int MIN_LEFT = 0;
int MIDDLE_RIGHT = 0;
int MIDDLE_LEFT = 0;
#define NUM_AVERAGES 1

void read_obstacles( int &left, int &front, int &right )
{
    // turn on sensor LEDs
    output_bit(SENSOR_LED, 1);
    delay_ms(SENSOR_VALID_DELAY);

    set_adc_channel( OBSTACLE_LEFT );
    delay_us(10);
    left = read_adc();

    set_adc_channel( OBSTACLE_RIGHT );
    delay_us(10);
    right = read_adc();

    set_adc_channel( OBSTACLE_FRONT );
    delay_us(10);
    front = read_adc();

    // turn off sensor LEDs
    output_bit(SENSOR_LED, 0);
}

void turn_right()
{
    turn_enable = 1;
    turn_180 = 0;
    turn_counter = 0;
    output_bit(L_WHEEL_DIR, L_FORWARD);
    output_bit(R_WHEEL_DIR, R_BACKWARD);
}

void turn_left()
{
    turn_enable = 1;
    turn_180 = 0;
    turn_counter = 0;
    output_bit(L_WHEEL_DIR, L_BACKWARD);
    output_bit(R_WHEEL_DIR, R_FORWARD);
}

void turn_backwards()
{
    turn_counter = 0;
}
```

```

    turn_enable = 1;
    turn_180 = 1;
    output_bit(L_WHEEL_DIR, L_BACKWARD);
    output_bit(R_WHEEL_DIR, R_FORWARD);
}

short left_wheel = 0, right_wheel = 0;

#INT_TIMER1
void stepper_timer()
{
    if (initial_start)
    {
        set_timer1(0xD9A6); // sets timer to interrupt in 9.8ms (for 20cm/s)

        output_bit(L_WHEEL, left_wheel);
        output_bit(R_WHEEL, right_wheel);

        if (!turn_enable)
        {
            output_bit(L_WHEEL_DIR, L_FORWARD);
            output_bit(R_WHEEL_DIR, R_FORWARD);

            if ( (stop == 0) && (turn_ready == 1) ) // turn-delay in progress
            {
                turn_delay_count++;
            }

            if( !(turn_ready) )
            {
                if ((ob_right < MIN_RIGHT) && (next_turn == RIGHT)) //right side is open and a
                right turn is registered
                {
                    turn_delay_count = 0;
                    turn_ready = 1;
                }
                else if ((ob_left < MIN_LEFT) && (next_turn == LEFT)) //left side is open and a
                left turn is registered
                {
                    turn_delay_count = 0;
                    turn_ready = 1;
                }
            }
            else
            {
                if( turn_delay_count == 75)
                {
                    stop = 1;
                    turn_delay_count = 0;
                }
            }
        }

        if ( stop )
        {
            switch (next_turn)
            {
                case LEFT:
                    turn_left();
                    break;
                case RIGHT:
                    turn_right();
                    break;
                case BACK:
                    turn_backwards();
                    break;
            }
        }
        else if (next_turn == BACK)
        {
            turn_backwards();
        }
    }
    else
    {
        turn_counter++;
        if (!turn_180)
        {
            if (turn_counter == RIGHT_ANGLE_TURN)
            {
                turn_enable = 0;
                turn_180 = 0;
                turn_counter = 0;
                next_turn = NONE;
                turn_ready = 0;
            }
        }
        else
        {
            if (turn_counter == BACK_TURN)
            {
                turn_enable = 0;
                turn_180 = 0;

                turn_counter = 0;
            }
        }
    }
}

```

```

        next_turn = NONE;
        turn_ready = 0;
    }
}
if (!(stop || turn_enable) && initial_start)
{
    if (!adjust_left)
    {
        if(left_wheel)
            left_wheel = 0;
        else
            left_wheel = 1;
    }
    else
    {
        adjust_counter++;
        if (adjust_counter == ADJUST_AMT)
        {
            adjust_left = 0;
            adjust_counter = 0;
        }
    }

    if (!adjust_right)
    {
        if(right_wheel)
            right_wheel = 0;
        else
            right_wheel = 1;
    }
    else
    {
        adjust_counter++;
        if (adjust_counter == ADJUST_AMT)
        {
            adjust_right = 0;
            adjust_counter = 0;
        }
    }
}
else
{
    left_wheel = 0;
    right_wheel = 0;
}

if ( input(DOT_COUNT) )
{
    output_bit(LCD_POINTS, 1);
    dot_count_timer = 1;
}

if (dot_count_timer)
{
    dot_count_counter++;
    if (dot_count_counter == DOT_COUNT_EXPIRE)
    {
        output_bit(LCD_POINTS, 0);
        dot_count_timer = 0;
        dot_count_counter = 0;
    }
}
}

}

#INT_RB
void signal_change()
{
    short rfl, rf0;
    int i;
    if ( input(RF_ON) )
    {
        if( !initial_start )
        {
            // calibration mode
            if (calibration_stage == 0) // move robot in front of wall
            {
                // calibrate nearest front sensor distance
                FRONT_OBSTACLE = 0;
                for ( i = 0; i < NUM_AVERAGES; i++ )
                {
                    output_bit(SENSOR_LED, 1);
                    delay_ms(SENSOR_VALID_DELAY*10);
                    set_adc_channel( OBSTACLE_FRONT );
                    delay_us(10);
                    FRONT_OBSTACLE += read_adc();
                    output_bit(SENSOR_LED, 0);
                }

                FRONT_OBSTACLE /= NUM_AVERAGES;
                calibration_stage++;
            }
        }
    }
}

```

```

else if (calibration_stage == 1)
{
    // calibrate nearest acceptable left and farthest acceptable right
    // (before off-centre adjustment takes effect)

    MAX_LEFT = 0;
    MIDDLE_RIGHT = 0;

    for ( i = 0; i < NUM_AVERAGES; i++ )
    {
        output_bit(SENSOR_LED, 1);
        delay_ms(SENSOR_VALID_DELAY*10);

        set_adc_channel( OBSTACLE_LEFT );
        delay_us(10);
        MAX_LEFT += read_adc();

        set_adc_channel( OBSTACLE_RIGHT );
        delay_us(10);
        MIDDLE_RIGHT += read_adc();

        output_bit(SENSOR_LED, 0);
    }

    MAX_LEFT /= NUM_AVERAGES;
    MIDDLE_RIGHT /= NUM_AVERAGES;

    calibration_stage++;
}

effect)
else if (calibration_stage == 2)
{
    // calibrate nearest acceptable right (before off-centre adjustment takes
    MAX_RIGHT = 0;
    MIDDLE_LEFT = 0;

    for ( i = 0; i < NUM_AVERAGES; i++ )
    {
        output_bit(SENSOR_LED, 1);
        delay_ms(SENSOR_VALID_DELAY*10);

        set_adc_channel( OBSTACLE_RIGHT );
        delay_us(10);
        MAX_RIGHT += read_adc();

        set_adc_channel( OBSTACLE_LEFT );
        delay_us(10);
        MIDDLE_LEFT += read_adc();

        output_bit(SENSOR_LED, 0);
    }

    MAX_RIGHT /= NUM_AVERAGES;
    MIDDLE_LEFT /= NUM_AVERAGES;

    calibration_stage++;
}

else if (calibration_stage == 3)
{
    // calibrate far left (no wall on left sensor)
    MIN_LEFT = 0;

    for ( i = 0; i < NUM_AVERAGES; i++ )
    {
        output_bit(SENSOR_LED, 1);
        delay_ms(SENSOR_VALID_DELAY*10);

        set_adc_channel( OBSTACLE_LEFT );
        delay_us(10);
        MIN_LEFT += read_adc();

        output_bit(SENSOR_LED, 0);
    }

    calibration_stage++;
}

else if (calibration_stage == 4)
{
    // calibrate far right (no wall on right sensor)

    MIN_RIGHT = 0;

    for ( i = 0; i < NUM_AVERAGES; i++ )
    {
        output_bit(SENSOR_LED, 1);
        delay_ms(SENSOR_VALID_DELAY*10);

        set_adc_channel( OBSTACLE_RIGHT );
        delay_us(10);

```

```

        MIN_RIGHT += read_adc();

        output_bit(SENSOR_LED, 0);
    }

    calibration_stage++;
}
else if ( calibration_stage == 5)
{
    initial_start = 1 ; //facing north initially

    rf1 = input(RF_1);
    rf0 = input(RF_0);

    if ( !rf1 && !rf0 ) // north
    {
        next_turn = NONE;
    }
    else if ( rf1 && !rf0 ) // east
    {
        next_turn = RIGHT;
    }
    else if ( !rf1 && rf0 ) // west
    {
        next_turn = LEFT;
    }
    else // south
    {
        next_turn = BACK;
    }
}
}
else
{
    rf_enable = 1;

    rf1 = input(RF_1);
    rf0 = input(RF_0);

    if ( !rf1 && !rf0 ) // north
    {
        switch (current_orient)
        {
            case N:
                // move forward
                next_turn = NONE;
                break;

            case E:
                // turn left
                next_turn = LEFT;
                break;

            case S:
                // turn backwards
                next_turn = BACK;
                break;

            case W:
                // turn right
                next_turn = RIGHT;
                break;
        }
        current_orient = N;
    }
    else if ( rf1 && !rf0 ) // west
    {
        switch (current_orient)
        {
            case N:
                // turn left
                next_turn = LEFT;
                break;

            case E:
                // turn backwards
                next_turn = BACK;
                break;

            case S:
                // turn right
                next_turn = RIGHT;
                break;

            case W:
                // move forward
                next_turn = NONE;
                break;
        }
        current_orient = W;
    }
    else if ( !rf1 && rf0 ) // east
    {
        //
        stop = 1;
        switch (current_orient)
        {
            case N:
                // turn right
                next_turn = RIGHT;
                break;

            case E:

```

```

        // move forward
        next_turn = NONE;
        break;
    case S:
        // turn left
        next_turn = LEFT;
        break;
    case W:
        // turn backwards
        next_turn = BACK;
        break;
    }
    current_orient = E;
}
else // south
{
    switch (current_orient)
    {
    case N:
        // turn backwards
        next_turn = BACK;
        break;
    case E:
        // turn right
        next_turn = RIGHT;
        break;
    case S:
        // move forward
        next_turn = NONE;
        break;
    case W:
        // turn left
        next_turn = LEFT;
        break;
    }
    current_orient = S;
}
}
else
{
    rf_enable = 0;
}
}

if ( input(CONTACT) )
    stop = 1;
else
    stop = 0;

if ( input(DOT_COUNT) )
{
    output_bit(LCD_POINTS, 1);
    dot_count_timer = 1;
}
}

main()
{
    output_bit(LCD_RESET, 1);
    delay_ms(1);
    output_bit(LCD_RESET, 0);

    setup_timer_1(T1_INTERNAL|T1_DIV_BY_1); // setup interrupts

    enable_interrupts(INT_TIMER1);
    enable_interrupts(INT_RB);
    enable_interrupts(GLOBAL);

    setup_port_a(ALL_ANALOG);
    setup_adc(adc_clock_internal);

    output_bit(L_WHEEL_DIR, L_FORWARD); // initial direction forward
    output_bit(R_WHEEL_DIR, R_FORWARD); // initial direction forward

    output_bit(SENSOR_LED, 0);

    while( initial_start == 0 );

    for( ; ; )
    {
        read_obstacles( ob_left, ob_front, ob_right );

        if (ob_front > FRONT_OBSTACLE)
        {
            stop = 1;
            cell_counter = CELL_SIZE;
        }
        else
        {
            stop = 0;
        }

        if ( !turn_enable )
        {

```

```
if ( (ob_left > MAX_LEFT) || (ob_right > MAX_RIGHT) )
{
    if( ((signed int)(ob_left - MAX_LEFT)) < ((signed int)(ob_right - MAX_RIGHT)) )
    {
        // adjust to right
        adjust_right = 1;
        adjust_left = 0;
    }
    else // closer to right
    {
        // adjust to left
        adjust_left = 1;
        adjust_right = 0;
    }
}
else
{
    adjust_right = 0;
    adjust_left = 0;
}
}
}
```

Ghost Header File

```
/*
*****
*/
/* EECE 474 Summer Semester 2002 */
/*
*****
*/
/* Team 1 Pac-Man Microcontroller Code */
/* ===== */
/*
*/
/* Ghost code header file */
/*
*****
*/

// Obstacle Sensor Constants
// Note: higher sensor readings mean closer obstacle
#define SENSOR_LED PIN_D7
#define SENSOR_VALID_DELAY 10 // time in ms for sensor read valid from sensor LED on

#define OBSTACLE_LEFT 1 // analog channel for left sensor
#define OBSTACLE_FRONT 2 // analog channel for front sensor
#define OBSTACLE_RIGHT 0 // analog channel for right sensor

// Other Sensor Constants
#define CONTACT PIN_B6

// Motor Control Constants
#define R_WHEEL PIN_C1
#define L_WHEEL PIN_C2
#define RIGHT_ANGLE_TURN 43 // number of pulses on one wheel to turn 90 degrees
#define BACK_TURN 92
#define L_WHEEL_DIR PIN_D2 // left wheel direction pin
#define R_WHEEL_DIR PIN_D0 // right wheel direction pin
#define L_FORWARD 1 // left wheel direction pin low is forward
#define L_BACKWARD 0 // left wheel direction pin high is backward
#define R_FORWARD 0 // right wheel direction pin high is forward
#define R_BACKWARD 1 // right wheel direction pin low is backward
#define ADJUST_AMT 1 // number of pulses to slow wheel down by when adjusting for
off-centre

// Beacon Sensor Constants
#define BEACON_FRONT 4 // analog channel AN4
#define BEACON_LEFT 5 // analog channel AN5
#define BEACON_BACK 6 // analog channel AN6
#define BEACON_RIGHT 7 // analog channel AN7

// Maze Constants
#define CELL_SIZE 102 // number of pulses to traverse one cell

//temp
//define CELL_SIZE 2 // number of pulses to traverse one cell

#define NUM_CELLS 36 // number of cells in maze
#define NUM_X_CELLS 6 // number of cells in east-west direction
#define NUM_Y_CELLS 6 // number of cells in north-south direction
#define NORTH 0x01
#define EAST 0x02
#define SOUTH 0x04
#define WEST 0x08
#define INITIAL_X_POS 2 // startup x coordinate for Ghost
#define INITIAL_Y_POS 2 // startup y coordinate for Ghost

#define CALIBRATION_POWER PIN_B7
#define CALIBRATION_GND PIN_B1
#define CALIBRATION_BUTTON PIN_B0

enum direction { N, E, S, W };

// Beacon Constants
enum Pac-Man_dir { PACFRONT, PACBACK, PACLEFT, PACRIGHT };
#define NEAR 4
#define MIDDLE 179
#define FAR 214

// Function Prototypes
void estimate_Pac-Man( signed int &x, signed int &y, int front, int back, int left, int right );
void solve( int x, int y ); // calculates shortest way to reach (x,y)
// from current position and writes list
// of moves in memory bank 3 starting with offset 0

void go_forward();
void turn_right();
void turn_left();
void turn_backwards();
void stepper_timer();
void signal_change();
void read_obstacles( int &left, int &front, int &right );
void read_beacon( int &front, int &left, int &back, int &right );
void move( direction dir );
short wall_exists( int x, int y, direction dir );
short is_neighbour( int x, int y, direction dir );
int point_to_cell_number( signed int x, signed int y );
int beacon_minval( int front, int left, int back, int right );
```

Ghost Source Code

```

/*****
/* EECE 474 Summer Semester 2002
/*****
/*
/* Team 1 Ghost Microcontroller Code
/* =====
/*
/* This program is for controlling the Ghost robot. It is designed for
/* a Microchip PIC16F877 running with an oscillator frequency of 4 MHz
/* It interfaces with the obstacle, Pac-Man contact, and Pac-Man beacon-
/* finding sensors. The Ghost estimates the position of Pac-Man using
/* the beacon-finding sensors and with knowledge of its current
/* position, uses a flood-filling maze-solving algorithm to find the
/* shortest path to get to Pac-Man.
/*
/*****

#include <16F877.h>
#include "Ghost.h"

#define PIC16F877 *16
// #device adc=10
#include delay (clock=4000000)
#fuses XT, NOWDT, NOPROTECT

int turn_counter = 0;
int cell_step_counter = 0;
int adjust_counter = 0;
direction current_orient = N;

short next_move_ready = 1;
short turn_enable = 0;
short turn_l80 = 0;
short stop = 1;
short adjust_left = 0, adjust_right = 0;
short backturn_dir = 0;
short initial_start = 0;

int calibration_stage = 0;
int FRONT_OBSTACLE = 245;
int MAX_RIGHT = 0;
int MAX_LEFT = 0;
int MIN_RIGHT = 0;
int MIN_LEFT = 0;
int MIDDLE_RIGHT = 0;
int MIDDLE_LEFT = 0;

short pacpos_unknown = 0;

int maze[NUM_CELLS] =
{
    NORTH | WEST,
    NORTH | SOUTH,
    NORTH,
    NORTH | SOUTH,
    NORTH | SOUTH,
    NORTH | EAST,
    WEST | EAST,
    WEST | NORTH,
    SOUTH,
    NORTH | SOUTH,
    NORTH | EAST,
    WEST | EAST,
    WEST | EAST,
    WEST | EAST,
    NORTH | WEST | SOUTH,
    NORTH,
    0,
    EAST,
    SOUTH,
    SOUTH,
    NORTH,
    EAST | SOUTH,
    WEST | EAST,
    WEST | EAST,
    NORTH | WEST,
    NORTH | SOUTH,
    0,
    NORTH | SOUTH,
    EAST | SOUTH,
    WEST | EAST,
    WEST | SOUTH,
    NORTH | SOUTH,
    SOUTH,
    NORTH | SOUTH,
    NORTH | SOUTH,
    EAST | SOUTH
};

// - - - - - --> increasing x
// | - - - - |
// | | - - | |

```

```

// | | | _ |
// | - - - | |
// | | - - - |
// | - - - - - |
//
// ||
// \ /
// increasing y

int current_x_pos = INITIAL_X_POS;
int current_y_pos = INITIAL_Y_POS;

short wall_exists( int x, int y, direction dir )
{
    int cell_no, cell_walls;
    cell_no = (y*NUM_X_CELLS) + x;

    if ((maze[cell_no] | NORTH) && (dir == N))
        return 1;
    else if ((maze[cell_no] | EAST) && (dir == E))
        return 1;
    else if ((maze[cell_no] | SOUTH) && (dir == S))
        return 1;
    else if ((maze[cell_no] | WEST) && (dir == W))
        return 1;
    return 0;
}

void move( direction dir ) // move direction one cell
{
    switch (dir)
    {
        case N:
            switch (current_orient)
            {
                case N:
                    // move forward
                    go_forward();
                    break;

                case E:
                    // turn left
                    turn_left();
                    break;

                case S:
                    // turn backwards
                    turn_backwards();
                    break;

                case W:
                    // turn right
                    turn_right();
                    break;
            }
            current_orient = N;
            break;

        case E:
            switch (current_orient)
            {
                case N:
                    // turn right
                    turn_right();
                    break;

                case E:
                    // move forward
                    go_forward();
                    break;

                case S:
                    // turn left
                    turn_left();
                    break;

                case W:
                    // turn backwards
                    turn_backwards();
                    break;
            }
            current_orient = E;
            break;

        case W:
            switch (current_orient)
            {
                case N:
                    // turn left
                    turn_left();
                    break;

                case E:
                    // turn backwards
                    turn_backwards();
                    break;

                case S:
                    // turn right
                    turn_right();
                    break;

                case W:
                    // move forward

```

```

        go_forward();
        break;
    }
    current_orient = W;
    break;
case S:
    switch (current_orient)
    {
    case N:
        // turn backwards
        turn_backwards();
        break;
    case E:
        // turn right
        turn_right();
        break;
    case S:
        // move forward
        go_forward();
        break;
    case W:
        // turn left
        turn_left();
        break;
    }
    current_orient = S;
    break;
}
}

void estimate_Pac-Man( signed int &x, signed int &y, int front, int back, int left, int right )
{
    Pac-Man_dir pacdir;
    int minvalue;
    int cells_away = 0;
    int random;

    pacdir = PACFRONT;
    minvalue = front;

    if ( minvalue > back )
    {
        minvalue = back;
        pacdir = PACBACK;
    }
    if ( minvalue > left )
    {
        minvalue = left;
        pacdir = PACLEFT;
    }
    if ( minvalue > right )
    {
        minvalue = right;
        pacdir = PACRIGHT;
    }

    if ( minvalue < FAR )
    {
        if ( minvalue <= NEAR )
            cells_away = 2;
        else if ( minvalue <= MIDDLE )
            cells_away = 3;
        else if ( minvalue <= FAR )
            cells_away = 4;

        switch( pacdir )
        {
        case PACFRONT:
            switch( current_orient )
            {
            case N:
                x = current_x_pos;
                y = current_y_pos - cells_away;
                break;
            case S:
                x = current_x_pos;
                y = current_y_pos + cells_away;
                break;
            case E:
                x = current_x_pos + cells_away;
                y = current_y_pos;
                break;
            case W:
                x = current_x_pos - cells_away;
                y = current_y_pos;
                break;
            }
            break;
        case PACLEFT:
            switch( current_orient )
            {
            case N:

```

```

        x = current_x_pos - cells_away;
        y = current_y_pos;
        break;
    case S:
        x = current_x_pos + cells_away;
        y = current_y_pos;
        break;
    case E:
        x = current_x_pos;
        y = current_y_pos - cells_away;
        break;
    case W:
        x = current_x_pos;
        y = current_y_pos + cells_away;
        break;
    }
    break;
case PACRIGHT:
    switch( current_orient )
    {
    case N:
        x = current_x_pos + cells_away;
        y = current_y_pos;
        break;
    case S:
        x = current_x_pos - cells_away;
        y = current_y_pos;
        break;
    case E:
        x = current_x_pos;
        y = current_y_pos + cells_away;
        break;
    case W:
        x = current_x_pos;
        y = current_y_pos - cells_away;
        break;
    }
    break;
case PACBACK:
    switch( current_orient )
    {
    case N:
        x = current_x_pos;
        y = current_y_pos - cells_away;
        break;
    case S:
        x = current_x_pos;
        y = current_y_pos + cells_away ;
        break;
    case E:
        x = current_x_pos - cells_away;
        y = current_y_pos;
        break;
    case W:
        x = current_x_pos + cells_away;
        y = current_y_pos;
        break;
    }
    break;
}
else
{
    random = get_timer1();
    x = (random/10)%6;
    y = random%6;
    // random Pac-Man position estimate
}
}

short is_neighbour( int x, int y, direction dir )
{
    int walls;
    walls = maze[point_to_cell_number(x,y)];

    if ( dir == N )
    {
        if ( walls & NORTH )
            return 0;
        else
            return 1;
    }

    else if ( dir == E )
    {
        if ( walls & EAST )
            return 0;
        else
            return 1;
    }

    else if ( dir == S )
    {

```

```

        if ( walls & SOUTH )
            return 0;
        else
            return 1;
    }
else // dir == W
{
    if ( walls & WEST )
        return 0;
    else
        return 1;
}
}

void solve( int x, int y )
{
    // maze solving algorithm to get to point pos from current position with fast flooding
    int queue_x[NUM_CELLS], queue_y[NUM_CELLS]; // holds queue of cells to calculate distance
    int goal_distance[NUM_CELLS]; // holds each cell's distance from point pos
    int neigh_cell_x, neigh_cell_y;
    int queue_count = 0, queue_index = 0, curr_cell_dist = 0; // for solving maze
    int i;
    int curr_pos_dist, neigh_pos_dist; // for creating roadmap
    int curr_cell_x, curr_cell_y;

    // initialize frontier array
    for ( i = 0; i < NUM_CELLS; i++ )
    {
        goal_distance[i] = 0xFF;
    }
    goal_distance[point_to_cell_number(x,y)] = curr_cell_dist;

    // add target point pos to queue
    queue_x[0] = x;
    queue_y[0] = y;
    queue_count++;

    while(queue_count > 0)
    {
        queue_count--;
        curr_cell_x = queue_x[queue_index];
        curr_cell_y = queue_y[queue_index];
        curr_cell_dist = goal_distance[point_to_cell_number(curr_cell_x, curr_cell_y)];
        queue_index++;

        if ( curr_cell_dist != 0xFF )
        {
            // check all neighbours for accessibility
            if ( is_neighbour(curr_cell_x, curr_cell_y, N) )
            {
                neigh_cell_x = curr_cell_x;
                neigh_cell_y = curr_cell_y - 1;

                if ( goal_distance[point_to_cell_number(neigh_cell_x, neigh_cell_y)] == 0xFF )
                {
                    goal_distance[point_to_cell_number(neigh_cell_x, neigh_cell_y)] =
curr_cell_dist + 1;

                    queue_x[queue_index + queue_count] = neigh_cell_x;
                    queue_y[queue_index + queue_count] = neigh_cell_y;
                    queue_count++;
                }
            }

            if ( is_neighbour(curr_cell_x, curr_cell_y, E) )
            {
                neigh_cell_x = curr_cell_x + 1;
                neigh_cell_y = curr_cell_y;

                if ( goal_distance[point_to_cell_number(neigh_cell_x, neigh_cell_y)] == 0xFF )
                {
                    goal_distance[point_to_cell_number(neigh_cell_x, neigh_cell_y)] =
curr_cell_dist + 1;

                    queue_x[queue_index + queue_count] = neigh_cell_x;
                    queue_y[queue_index + queue_count] = neigh_cell_y;
                    queue_count++;
                }
            }

            if ( is_neighbour(curr_cell_x, curr_cell_y, S) )
            {
                neigh_cell_x = curr_cell_x;
                neigh_cell_y = curr_cell_y + 1;

                if ( goal_distance[point_to_cell_number(neigh_cell_x, neigh_cell_y)] == 0xFF )
                {
                    goal_distance[point_to_cell_number(neigh_cell_x, neigh_cell_y)] =
curr_cell_dist + 1;

                    queue_x[queue_index + queue_count] = neigh_cell_x;
                    queue_y[queue_index + queue_count] = neigh_cell_y;
                    queue_count++;
                }
            }
        }
    }
}

```

```

curr_cell_dist + 1;

        }
    }

    // create roadmap
    curr_pos_dist = goal_distance[point_to_cell_number(current_x_pos, current_y_pos)];
    curr_cell_x = current_x_pos;
    curr_cell_y = current_y_pos;

    for ( i= 0; curr_pos_dist > 0; i++ )
    {
        // check north
        neigh_cell_x = curr_cell_x;
        neigh_cell_y = curr_cell_y - 1;

        if ( (neigh_cell_x < NUM_X_CELLS) && (neigh_cell_y < NUM_Y_CELLS) )
        {
            if ( goal_distance[point_to_cell_number(neigh_cell_x, neigh_cell_y)] < curr_pos_dist )
            {
                // add to roadmap
                curr_pos_dist = goal_distance[point_to_cell_number(neigh_cell_x, neigh_cell_y)];
                write_bank(3, i, NORTH);
                curr_cell_x = neigh_cell_x;
                curr_cell_y = neigh_cell_y;
                continue;
            }
        }

        // check east
        neigh_cell_x = curr_cell_x + 1;
        neigh_cell_y = curr_cell_y;

        if ( (neigh_cell_x < NUM_X_CELLS) && (neigh_cell_y < NUM_Y_CELLS) )
        {
            if ( goal_distance[point_to_cell_number(neigh_cell_x, neigh_cell_y)] < curr_pos_dist )
            {
                // add to roadmap
                curr_pos_dist = goal_distance[point_to_cell_number(neigh_cell_x, neigh_cell_y)];
                write_bank(3, i, EAST);
                curr_cell_x = neigh_cell_x;
                curr_cell_y = neigh_cell_y;
                continue;
            }
        }

        // check south
        neigh_cell_x = curr_cell_x;
        neigh_cell_y = curr_cell_y + 1;

        if ( (neigh_cell_x < NUM_X_CELLS) && (neigh_cell_y < NUM_Y_CELLS) )
        {
            if ( goal_distance[point_to_cell_number(neigh_cell_x, neigh_cell_y)] < curr_pos_dist )
            {
                // add to roadmap
                curr_pos_dist = goal_distance[point_to_cell_number(neigh_cell_x, neigh_cell_y)];
                write_bank(3, i, SOUTH);
                curr_cell_x = neigh_cell_x;
                curr_cell_y = neigh_cell_y;
                continue;
            }
        }

        // check west
        neigh_cell_x = curr_cell_x - 1;
        neigh_cell_y = curr_cell_y;

        if ( (neigh_cell_x < NUM_X_CELLS) && (neigh_cell_y < NUM_Y_CELLS) )
        {
            if ( goal_distance[point_to_cell_number(neigh_cell_x, neigh_cell_y)] < curr_pos_dist )
            {
                // add to roadmap
                curr_pos_dist = goal_distance[point_to_cell_number(neigh_cell_x, neigh_cell_y)];
                write_bank(3, i, WEST);
                curr_cell_x = neigh_cell_x;
                curr_cell_y = neigh_cell_y;
                continue;
            }
        }
    }
    write_bank(3, i, 0);

    // clean up

```

```

        queue_count = 0, queue_index = 0, curr_cell_dist = 0;
    }

int point_to_cell_number( signed int x, signed int y )
{
    int pos_y;
    int pos_x;

    pos_y = y;
    pos_x = x;

    if (x > 5)
        pos_x = 5;
    else if (x < 0)
        pos_x = 0;

    if (y > 5)
        pos_y = 5;
    else if (y < 0)
        pos_y = 0;

    return (pos_y*NUM_X_CELLS) + pos_x;
}

void read_obstacles( int &left, int &front, int &right )
{
    // turn on sensor LEDs
    output_bit(SENSOR_LED, 1);
    delay_ms(SENSOR_VALID_DELAY);

    set_adc_channel( OBSTACLE_LEFT );
    delay_us(10);
    left = read_adc();

    set_adc_channel( OBSTACLE_FRONT );
    delay_us(10);
    front = read_adc();

    set_adc_channel( OBSTACLE_RIGHT );
    delay_us(10);
    right = read_adc();

    // turn off sensor LEDs
    output_bit(SENSOR_LED, 0);
}

void read_beacon( int &front, int &left, int &back, int &right )
{
    set_adc_channel( BEACON_FRONT );
    delay_us(10);
    front = read_adc();

    set_adc_channel( BEACON_LEFT );
    delay_us(10);
    left = read_adc();

    set_adc_channel( BEACON_BACK );
    delay_us(10);
    back = read_adc();

    set_adc_channel( BEACON_RIGHT );
    delay_us(10);
    right = read_adc();
}

void go_forward()
{
    cell_step_counter = 0;
    stop = 0;
    next_move_ready = 0;
    turn_enable = 0;
    turn_180 = 0;
    turn_counter = 0;

    output_bit(L_WHEEL_DIR, L_FORWARD);
    output_bit(R_WHEEL_DIR, R_FORWARD);
}

void turn_right()
{
    cell_step_counter = 0;
    stop = 0;
    next_move_ready = 0;
    turn_enable = 1;
    turn_180 = 0;
    turn_counter = 0;

    output_bit(L_WHEEL_DIR, L_FORWARD);
    output_bit(R_WHEEL_DIR, R_BACKWARD);
}

void turn_left()
{
    cell_step_counter = 0;
    stop = 0;
}

```

```

        next_move_ready = 0;
        turn_enable = 1;
        turn_180 = 0;
        turn_counter = 0;

        output_bit(L_WHEEL_DIR, L_BACKWARD);
        output_bit(R_WHEEL_DIR, R_FORWARD);
    }

void turn_backwards()
{
    cell_step_counter = 0;
    stop = 0;
    next_move_ready = 0;
    turn_enable = 1;
    turn_180 = 1;
    turn_counter = 0;

    if (backturn_dir)
    {
        backturn_dir = 0;
        output_bit(L_WHEEL_DIR, L_BACKWARD);
        output_bit(R_WHEEL_DIR, R_FORWARD);
    }
    else
    {
        backturn_dir = 1;
        output_bit(L_WHEEL_DIR, L_FORWARD);
        output_bit(R_WHEEL_DIR, R_BACKWARD);
    }
}

short left_wheel = 0, right_wheel = 0;

int beacon_minval( int front, int left, int back, int right )
{
    int minvalue;
    minvalue = front;

    if( left < minvalue )
    {
        minvalue = left;
    }

    if( back < minvalue )
    {
        minvalue = back;
    }

    if( right < minvalue )
    {
        minvalue = right;
    }

    return minvalue;
}

#INT_TIMER1
void stepper_timer()
{
    if (initial_start)
    {
        set_timer1(0xE152); // sets timer to interrupt in 7.853981634ms (for 64 Hz
wave)

        output_bit(L_WHEEL, left_wheel);
        output_bit(R_WHEEL, right_wheel);

        if (!turn_enable && !next_move_ready)
        {
            if ( cell_step_counter++ >= CELL_SIZE )
            {
                cell_step_counter = 0;
                next_move_ready = 1;
                stop = 1;
            }
            output_bit(L_WHEEL_DIR, L_FORWARD);
            output_bit(R_WHEEL_DIR, R_FORWARD);
        }
        else if (turn_enable)
        {
            turn_counter++;
            if (!turn_180)
            {
                if (turn_counter == RIGHT_ANGLE_TURN)
                {
                    turn_enable = 0;
                    turn_180 = 0;
                    turn_counter = 0;
                }
            }
            else
            {
                if (turn_counter == BACK_TURN)
                {

```

```

        turn_enable = 0;
        turn_180 = 0;
        turn_counter = 0;
    }
}

if (!stop || turn_enable)
{
    if (!adjust_left || turn_enable)
    {
        if(left_wheel)
            left_wheel = 0;
        else
            left_wheel = 1;
    }
    else
    {
        adjust_counter++;
        if (adjust_counter == ADJUST_AMT)
        {
            adjust_left = 0;
            adjust_counter = 0;
        }
    }

    if (!adjust_right || turn_enable )
    {
        if(right_wheel)
            right_wheel = 0;
        else
            right_wheel = 1;
    }
    else
    {
        adjust_counter++;
        if (adjust_counter == ADJUST_AMT)
        {
            adjust_right = 0;
            adjust_counter = 0;
        }
    }
}
else
{
    left_wheel = 0;
    right_wheel = 0;
}
}

#INT_RB
void signal_change()
{
    if ( input(CONTACT) )
        stop = 1;
    else
        stop = 0;
}

main()
{
    int i;
    int ob_left, ob_right, ob_front;
    int beacon_l, beacon_f, beacon_r, beacon_b;
    signed int Pac-Man_pos_x, Pac-Man_pos_y;
    int next_move;

    write_bank(3, 0, 0); // initializing memory bank 3 which will hold list of moves calculated from solve

    output_bit(SENSOR_LED, 0);

    setup_timer_1(T1_INTERNAL|T1_DIV_BY_1); // setup interrupts

    output_bit(CALIBRATION_POWER, 1);
    output_bit(CALIBRATION_GND, 0);

    enable_interrupts(INT_TIMER1);
    enable_interrupts(INT_RB);
    enable_interrupts(GLOBAL);

    setup_port_a(ALL_ANALOG);
    setup_adc(adc_clock_internal);

    output_bit(L_WHEEL_DIR, L_FORWARD); // initial direction forward
    output_bit(R_WHEEL_DIR, R_FORWARD); // initial direction forward

    while( initial_start == 0 )
    {
        if ( input(CALIBRATION_BUTTON) )
        {
            while (input(CALIBRATION_BUTTON));

            // calibration mode
            if (calibration_stage == 0) // move robot in front of wall

```

```

    {
        // calibrate nearest front sensor distance
        FRONT_OBSTACLE = 0;
        output_bit(SENSOR_LED, 1);
        delay_ms(SENSOR_VALID_DELAY*10);
        set_adc_channel( OBSTACLE_FRONT );
        delay_us(10);
        FRONT_OBSTACLE += read_adc();
        output_bit(SENSOR_LED, 0);

        calibration_stage++;
    }
else if (calibration_stage == 1)
{
    // calibrate nearest acceptable left and farthest acceptable right
    // (before off-centre adjustment takes effect)
    MAX_LEFT = 0;

    output_bit(SENSOR_LED, 1);
    delay_ms(SENSOR_VALID_DELAY*10);

    set_adc_channel( OBSTACLE_LEFT );
    delay_us(10);
    MAX_LEFT += read_adc();

    output_bit(SENSOR_LED, 0);

    calibration_stage++;
}
else if (calibration_stage == 2)
{
    // calibrate nearest acceptable right (before off-centre adjustment takes
effect)
    MAX_RIGHT = 0;
    MIDDLE_LEFT = 0;

    output_bit(SENSOR_LED, 1);
    delay_ms(SENSOR_VALID_DELAY*10);

    set_adc_channel( OBSTACLE_RIGHT );
    delay_us(10);
    MAX_RIGHT += read_adc();

    output_bit(SENSOR_LED, 0);

    calibration_stage++;
}
else if (calibration_stage == 3)
{
    // calibrate far left (no wall on left sensor)
    MIN_LEFT = 0;

    output_bit(SENSOR_LED, 1);
    delay_ms(SENSOR_VALID_DELAY*10);

    set_adc_channel( OBSTACLE_LEFT );
    delay_us(10);
    MIN_LEFT += read_adc();

    output_bit(SENSOR_LED, 0);

    calibration_stage++;
}
else if (calibration_stage == 4)
{
    // calibrate far right (no wall on right sensor)
    MIN_RIGHT = 0;

    output_bit(SENSOR_LED, 1);
    delay_ms(SENSOR_VALID_DELAY*10);

    set_adc_channel( OBSTACLE_RIGHT );
    delay_us(10);
    MIN_RIGHT += read_adc();

    output_bit(SENSOR_LED, 0);

    calibration_stage++;
}
else if (calibration_stage == 5)
{
    initial_start = 1;
}
}
}
delay_ms(5000);
for( ; ; )

```

```

{
    // Look for Pac-Man's beacon signal
    read_beacon( beacon_f, beacon_l, beacon_b, beacon_r );

    if (beacon_minval(beacon_f, beacon_l, beacon_b, beacon_r) > FAR)
        pacpos_unknown = 1;

    estimate_Pac-Man(Pac-Man_pos_x, Pac-Man_pos_y, beacon_f, beacon_b, beacon_l, beacon_r);
    solve(Pac-Man_pos_x, Pac-Man_pos_y);

    next_move = read_bank(3, 0);
    for ( i = 0; next_move != 0; i++ )
    {
        if (pacpos_unknown)
        {
            read_beacon( beacon_f, beacon_l, beacon_b, beacon_r );
            if (beacon_minval(beacon_f, beacon_l, beacon_b, beacon_r) < FAR)
                break;
        }

        next_move = read_bank(3, i);
        if (next_move == NORTH)
        {
            current_y_pos--;
            move( N );
        }
        else if (next_move == EAST)
        {
            current_x_pos++;
            move( E );
        }
        else if (next_move == SOUTH)
        {
            current_y_pos++;
            move( S );
        }
        else if (next_move == WEST)
        {
            current_x_pos--;
            move( W );
        }

        while (!next_move_ready) // while not ready, adjust position (make all below inside the
while loop)
        {
            read_obstacles( ob_left, ob_front, ob_right );

            if (ob_front > FRONT_OBSTACLE)
            {
                stop = 1;
            }
            else
                stop = 0;

            if ( !turn_enable )
            {
                if ( (ob_left > MAX_LEFT) || (ob_right > MAX_RIGHT) )
                {
                    if( ((signed int)(ob_left - MAX_LEFT)) < ((signed
int)(ob_right - MAX_RIGHT)) )
                    {
                        // adjust to right
                        adjust_right = 1;
                        adjust_left = 0;
                    }
                    else // closer to right
                    {
                        // adjust to left
                        adjust_left = 1;
                        adjust_right = 0;
                    }
                }
            }
        }

        // recalibrate Ghost's known position if currently facing a wall
        if ( wall_exists(current_x_pos, current_y_pos, current_orient) )
        {
            while ( ob_front < FRONT_OBSTACLE ) // if no front wall found when there should
be one
            {
                // move forward until there is a front obstacle
                read_obstacles( ob_left, ob_front, ob_right );
                go_forward();
            }
            stop = 1;
        }
    }
    stop = 1;
}
}

```