EEL 5934

Intelligent Machines Design
Laboratory

Clyde

The Exploring House Robot

Jeff Webb

December 9, 1996

Professor: Keith L. Doty

University of Florida

Table of Contents

Abstract

This paper discusses the design of Clyde, an autonomous house robot. Clyde's goal is to explore its world and survive in a cluttered home environment. The robot perceives its surroundings using an array of inexpensive sensors including five forward-looking infrared proximity sensors, an integral bump sensor, wheel encoders, and a compass. The robot has several behaviors that enable it to explore a hostile environment. Clyde can *avoid obstacles*, *escape* from corners, become *frustrated*, and *migrate* in a particular direction. The robot uses a combination of subsumption and neural-net based architectures to arbitrate between these different behaviors.

Executive Summary

Clyde is an autonomous house robot that I designed to explore its surroundings and survive in a cluttered home environment. The robot has a round platform with a dome-like shell that serves as an integral bump sensor. The platform has two drive wheels and a tail-skid. Clyde is controlled by a 68HC11 microprocessor running compiled C software. The robot perceives its surroundings using an array of inexpensive sensors including five forward-looking infrared proximity sensors, eight bump sensors, two wheel encoders, and a compass.

The robot has several behaviors that enable it to explore a hostile environment. Clyde can *avoid obstacles*, *escape* from corners, become *frustrated*, *detect motor stalls*, and *migrate* in a particular direction. The robot uses a combination of subsumption and neural-net based architectures to arbitrate between these different behaviors.

For the most part, Clyde was successful in his role as a house robot. The robot's robust bump sensor design and maneuverable platform allow it to navigate through cluttered environments. The neural net obstacle avoidance and migrate behaviors also work fairly well. The motor stall detection behavior has mixed results. The routine will detect most stalls and wheel slippage, but it sometimes triggers falsely and occasionally does not trigger at all. Overall, Clyde is a successful robot with the potential for future expansion.

Introduction

This paper is about Clyde, an autonomous house robot. Clyde's purpose is to explore its world and survive in the cluttered environment that we humans call home. This is not as simple as it may sound. Although the living room or kitchen of an average home may seem comfortable and secure to human eyes, mobile robots see a world full dangerous obstacles. Chair legs, clothes on the floor, and stairs can be serious hazards to an unsuspecting robot. Clyde's goal is to navigate through such an environment without getting trapped or stuck on any obstacles.

The robot requires several major components in order to accomplish its mission. First of all, Clyde needs a mobile platform capable of navigating through a crowded room. The robot must also have motors to propel itself and sensors to detect its environment. In order to explore a room, Clyde needs behaviors to avoid obstacles and find unknown territory. Finally, the robot needs a method to integrate all of these components into one useful system.

In this report, I will describe the mechanical, electrical, and behavioral systems that enable Clyde to survive in a complex environment. I will first introduce the robot's design at a system level, and then describe each of Clyde's subsystems. Finally, I will discuss each of Clyde's behaviors and evaluate the robot's performance.

Integrated System

Hardware

I chose to use a 68HC11E9 microprocessor as the controller for the robot. I used a standard Motorola EVBU board with Novasoft's ME11 kit, which adds 32k of RAM, motor drivers, and several I/O ports to the system. I designed a battery-backup circuit for the 32k RAM which is described in Appendix B.

Software

I wrote the robot's software in C using the ICC11 compiler. In addition to Clyde-specific code, I wrote several general libraries that can be used for other applications. This includes routines for multi-tasking, time-keeping, motion control, shaft encoders, IR sensors and bump sensors.

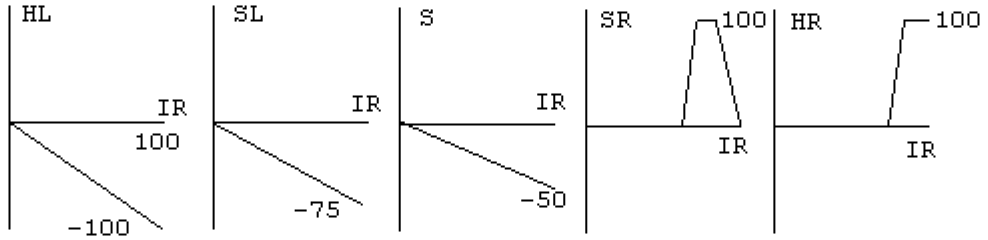
System Overview

Clyde's control system has four main components: sensory input, behavioral desires, behavioral arbitration, and motor actuation. The robot's sensors are sampled by calling the *sampleSensors()* function, which takes readings from the sensors and updates the global sensor output variables. Clyde's behavioral desires are generated by several behavior functions which are called in the main program loop. Each behavior gives its opinion on what the robot's forward speed and turn rate should be, based upon the current sensor values. These opinions are then combined by the behavioral arbitrator. For most of the behaviors, the arbitrator consists of a neural net that sums up the opinions from each sensor and chooses the most popular course of action. A few behaviors use a subsumption arbitrator in which a higher priority behavior completely takes control from the lower level behaviors. Finally, after the behaviors have been arbitrated, the *motionControl()* routine updates the robot's forward speed and turn rate. This routine also smooths the motor response and performs motor speed calculations.

Neural Net

The object avoidance and migrate behaviors are combined using a very simple neural net system. Each behavior has one or more neurons that give opinions on what the robot's current speed and turn rate should be. The robot has five possible forward speeds (fast, medium, slow, very slow, and reverse), and five possible turn rates (hard left, soft left, straight, soft right, and hard right). Each neuron in the net gives an opinion on each one of these possible speeds and turn rates. The benefit of this system is that the system designer can consider each neuron's opinion independently, and then the neural net will combine all the opinions together and arbitrate between them. Figure 1 shows an example of how the opinions for an IR proximity sensor are determined.

Figure 1: Neural Net Transfer Functions



As one can see from the Figure 1, the opinions for this neuron are dependent upon the amount of IR reflection. When no reflection is detected, the neuron has no opinion on the robot’s movement, which allows other neurons to control the robot’s turn rate. As the IR reading increases, however, this neuron gives stronger opinions about where it would like to turn and where it does not want to turn. Since this is a left IR sensor, as the IR reading increases, the robot will tend to turn right in order to avoid the approaching object.

After the opinions for each neuron have been generated, the neuron’s opinions are multiplied by an associated weight factor. This factor allows the system designer to give certain behaviors more influence over the robot’s behavior. For example, the bump sensor neurons have a greater weight than the IR sensor neurons. After this step, the weighted opinions on each subject are added together, and the speed and turn rate with the highest opinions are selected to control the robot.

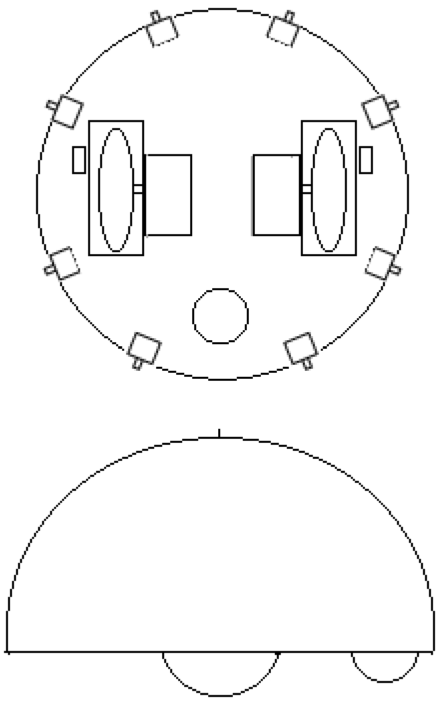
Mobile Platform

The robot chassis is one of the most important features of a mobile robot. The chassis serves as the “body” for the robot, allowing the machine to interact with its environment. Clyde must be able to explore and navigate through a cluttered and changing environment, so the platform must be very maneuverable. The robot must also be able to survive collisions with obstacles, and not get stuck by running over small objects on the floor. This means that the design must be as robust as possible.

In order to satisfy these objectives, I chose to use a round platform with two drive wheels and a tail-skid. A sketch of the platform is shown in Figure 2. One benefit of this chassis design is that the robot can spin in place with very little possibility of bumping into an object during a turn. This makes the robot very maneuverable in tight situations. The platform is 10 ½ inches in diameter, and made out of 1/4 inch thick birch plywood. The plywood is strong, light weight, and very easy to work with.

Figure 2: Mobile Robot Platform

Bottom and Side Views



The platform has two holes for mounting the wheels and drive motors. The two drive motors are modified model airplane servos that turn 3-inch rubber wheels. I mounted the modified servos such that the bottom of the servos are flush with the bottom of the main platform. This configuration allows as much ground clearance as possible without anything protruding below the platform that will cause the robot to get stuck on small objects. Since the robot rides fairly low to the ground, I decided to use a tail-skid instead of a rear caster wheel. The tail-skid, which I constructed from half of a ping-pong ball, is simple, light weight, and effective.

The most interesting feature of this robot platform is the outer shell which serves as a bump sensor. The outer shell is a large plastic bowl which encloses the entire robot. A wire frame supports this dome, which pivots on a single screw at the top of the robot. I mounted ten keyboard switches around the perimeter of the lower platform. When the robot bumps into an object, the shell tilts, and some of the switches are depressed. The goal of this shell is to detect a collision, no matter what part of the robot bumps into an obstacle.

Actuation

Drive Motors

The main actuators used on Clyde are the drive motors. The drive motors must provide adequate torque to maneuver the robot, have a low power consumption, and also have an acceptable top-end speed. In addition to these performance requirements, the motors must also be durable enough to sustain constant use and stresses on the gear train.

The motors used in model-airplane servos meet these requirements, and also have other desirable qualities. The servo motors are small, efficient, and durable. They also provide enough torque and speed for use on small robots. The best features of these motors, however, are the internal gearing, the ease of mounting, and their low price. These motors have been used with much success on many robots created in previous intelligent machine courses.

I modified two servos for continuous rotation using a technique demonstrated by Scott Jantz, one of the lab assistants. This process involved cutting the gear-stops on the internal gears, and then removing all of the electronics from the inside the servos. After this step, I soldered the servo connector wires directly to the motors. The two motors are driven by a SN754410 motor driver chip on the ME11 expansion board. The microprocessor can control the speed and direction of each motor by sending pulse-width modulated (PWM) and select signals to the motor drivers. I used Professor Doty's *motor.c* library file to drive the motors in my ICC11 programs.

Sensors

In order for a robot to navigate and interact with other objects, the robot must have some sense of its environment. Clyde perceives his surroundings using an array of inexpensive sensors. These sensors include infrared proximity sensors, an integral bump sensor, wheel encoders, and a compass.

Infrared Proximity Sensors

Clyde has five forward-looking infrared proximity sensors which serve as the robot's "eyes." Clyde uses these sensors to detect approaching objects and avoid collisions. The five sensors are mounted in a radial pattern on the inside of the robot's shell, which has holes for each of the sensors. This configuration protects the sensors from collisions and provides a wide field of view in front of the robot.

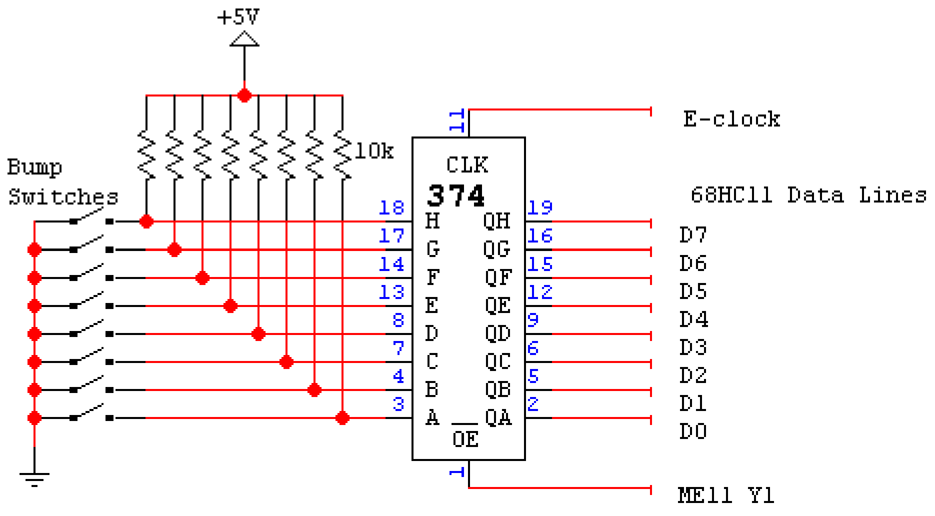
Each sensor consists of a 40kHz modulated infrared LED and a Sharp GP1U58Y infrared detector. The emitters are controlled and modulated by an output latch on the ME11 expansion board. Each emitter is columnated in a black tube to prevent IR leakage from saturating the detectors. I modified the Sharp IR detectors to output analog voltages, as described in lab, and then connected the output of these sensors to the analog inputs of the 68HC11. I then wrote ICC11 routines to sample and normalize the readings from the sensors.

Bump Sensor

Bump sensors are used to detect when the robot has collided with an obstacle and needs to change course. Ideally, most objects will be detected with infrared proximity sensors before a collision occurs, but in an unpredictable environment, bump sensors are a necessity. Since bump sensors are essentially the "last line of defense" for detecting obstacles, I wanted the Clyde's sensor to be very reliable. The resulting design was the pivoting shell configuration described in the *mobile platform* section of this report.

I placed eight SPST keyboard switches around the perimeter of the robot and interfaced them to the micro-controller using a 74HC374 8-bit latch. One terminal of each switch is grounded, and the other terminal is connected to one of the eight latch inputs. I used 10k pull-up resistors on each of the latch inputs. The latch is read as a memory-mapped input at address $4000 by using the Y1 line on the ME11 expansion board as a chip enable for the latch. Data is clocked into the latch by the 68HC11 E-clock. The schematic for the bump sensor interface is shown in Figure 3.

Figure 3: Bump Sensor Interface



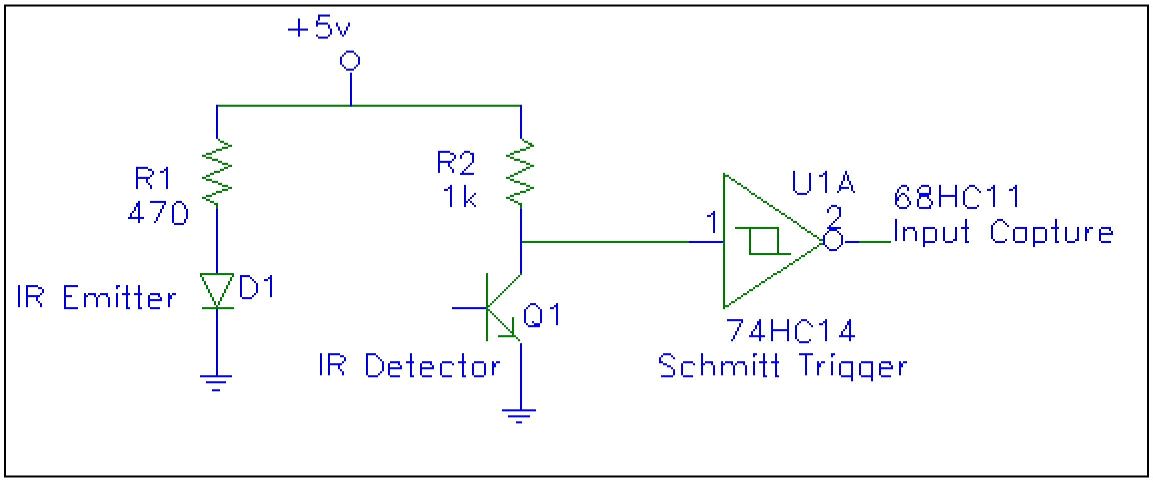
The bump sensor can be polled by a single software read from memory address $4000. Each bit of the data byte corresponds to the state of a bump switch. The switches are active low, so a data value of $FF means that no switches are being depressed. I wrote a set of generic ICC11 functions that allow a user to sample the latch value and determine which bump sensors are active. I also wrote a specific routine for Clyde’s bump sensor configuration that allows the robot to determine the direction of the collision.

Shaft Encoders

Clyde uses a pair of shaft encoders to measure the rotation of the drive wheels. These encoders allow the robot to determine distance it has traveled, the number of degrees it has turned, and the current speed of each wheel. Clyde uses this information to measure forward movement, perform precise rotation, and to determine if the motors have stalled.

The shaft encoders sense wheel rotation by measuring the amount of infrared light reflected off striped cut-outs glued each wheel. The cut-outs are circular pieces of poster-board with 16 alternating black and white stripes painted like pie slices. I glued a cut-out on the outside of each wheel, and then mounted Sharp 2L01 infrared emitter/detector pairs about a centimeter away from each cut-out. I used a 470 ohm current-limiting resistor on the infrared emitter and a 1k resistor on the collector of the detector. I then used a 74HC14 Schmitt-trigger inverter to convert the signals to 0 - 5V square waves. The 68HC11 then uses input capture lines IC2 and IC3 to detect the transitions between the black and white stripes. The schematic for this circuit is shown in Figure 4.

Figure 4: Shaft Encoder Circuit



I wrote an ICC11 library to interpret the data from the shaft encoders. The library includes an initialization routine to set up the system, two interrupt service routines to process the encoder data, and several other functions that allow the user to read, reset, and turn off the encoders.

Compass

Clyde has a compass for measuring rotational motion. This sensor is useful for navigation and calibration purposes. Clyde uses the compass to migrate in a particular direction, to determine if he is trapped, and to detect excessive wheel slippage. The compass, which was designed by myself and Kevin McFarlin, uses an optical encoder from a PC mouse to measure the rotation of a floating magnet. The design is very compact and inexpensive. The details of the compass design are shown in Appendix A.

Behaviors

Object Avoidance Behaviors

IR Avoidance

Clyde uses his five infrared proximity sensors to avoid the large obstacles he encounters. Each sensor has a corresponding neuron in the robot’s neural net that gives an opinion on what it thinks the robot’s current speed and turn rate should be. If a proximity sensor senses a high level of IR reflection, the sensor’s neuron wants the robot to slow down and steer away from the approaching object. The opinions from each neuron are all added into the neural net, and then the turn rate and speed with the highest opinions are selected to control the robot.

Delta IR Avoidance

In addition to responding to the magnitude IR reflection, Clyde also responds to a change in IR reflection on any one sensor. Each proximity sensor has another neuron that gives an opinion based upon the change in IR reflection since the last sample time. This set of opinions causes Clyde to react more quickly to objects when traveling at full speed.

Bump Avoidance

When Clyde's IR sensors fail to detect an object, he must use his bump sensors to navigate around the obstacle. Just like the IR sensors, each bump sensor has a neuron that gives an opinion on the robot's speed and turn rate. The bump sensor opinions, however, have a much greater weight than the IR sensors in the neural net. This allows the bump sensors to have priority over the proximity sensors when a bump has occurred.

Escape Corner Behavior

With only the neural-net based object avoidance, Clyde can function very well under most conditions. Unfortunately, there are a few situations that cause Clyde to get trapped. One of these situations arises when Clyde is facing a dark (non IR-reflective) corner. Since the IR sensors sense no reflection, only the bump sensors have an opinion. This causes the robot to bump on the right side, then on the left side, then on the right side… etc. This is where the *escape corner* behavior comes in. If Clyde encounters the situation described above, when he bumps on the right side for the second time in the same location, he will turn right instead of left. This causes the oscillation to be broken, and Clyde can escape.

Frustrated Behavior

Occasionally, other situations arise where Clyde oscillates or "freezes" and no collisions occur. The *frustrated* behavior allows Clyde to escape from these scenarios as well. When Clyde notices that he has not made much forward progress over a long period of time, and that his heading is approximately the same, Clyde get "frustrated" with his progress and turns a random amount. This action usually "frees" Clyde from his trapped condition, and he can continue exploring.

Motor Stall Detection Behavior

There are some very low objects that Clyde's bump sensors cannot detect. This includes clothing, tools, and other small objects that may be lying on the floor of a typical home. When Clyde encounters one of these objects, the robot may pass over it, the wheels may get stuck and stop turning, or Clyde might get stuck on top of the object, causing a wheel to spin freely. The latter two cases could be very bad situations where the robot is trapped forever. The *stall detection* behavior is designed to detect either of these situations and allow Clyde to escape.

When a wheel spins freely, the motor speed is faster than normal for a given motor duty cycle. When a wheel is stuck, the speed is lower than expected. The *stall detection* behavior takes advantage of this information, and compares the speed of each wheel to an expected speed for the current duty cycle. If the actual speeds are not close to the expected speeds, Clyde will stop, move backwards, and then turn. The difficulty with this scheme is in determining what the expected speeds should be. Ideally, the robot would adaptively change these values to account for different environments, but currently Clyde only calibrates the table on startup.

Performance

Mechanical

The mechanical design of the robot worked very well. The most impressive part of this design is the integrated bump sensor. The sensor consistently detects collisions in the "real world" environment of my cluttered room. The outer shell also provides good protection for the microprocessor and its sensors. A standard robot would much more vulnerable to damage in an unpredictable environment. The shell design does have some disadvantages, however. Mounting the IR sensors, power switches, and serial connector is much more involved than on the standard type robot. The enclosed design also cuts down on the space available to mount sensors and other components. In spite of these deterrents, I believe this design is a very good option for "real world" robots.

Electrical

All of the sensors on the robot proved to be effective. The keyboard switches worked very well as bump sensors. The switches are cheap, fairly sensitive, and very durable. The compass worked well when mounted on the robot, and seemed to be repeatable (see Appendix A). The shaft encoders were also very consistent and reliable. I did not do any detailed mapping using the encoders, but I suspect that they would work fairly well for this purpose. The IR sensors worked well, except for some infrared leakage between the emitters and detectors that caused the ambient readings to rise above the normal values. My battery back-up circuit performed flawlessly throughout the semester. I never had to reload a program due to memory corruption.

Behavioral

In most situations, Clyde's obstacle avoidance was fairly good. Clyde's IR sensors do not always detect dark objects, but the bump sensors detect these objects and allow the robot to continue. As mentioned in the *behaviors* section, the object avoidance occasionally fails and Clyde gets trapped. Fortunately, the *frustrated* and *escape* behaviors usually allow Clyde to get out of these situations. Sometimes these behaviors may cause Clyde to look confused, but they usually succeed in allowing Clyde to escape these traps.

The motor stall detection behavior has demonstrated limited success. Currently, Clyde is able to detect most stall conditions, but sometimes the behavior triggers falsely, and sometimes it does not detect stalls. The main difficulties with this behavior are creating the table of expected speeds (which changes with battery voltage), and avoiding false triggering when the motors are changing speeds.

Conclusion

For the most part, Clyde was successful in his role as a house robot. He is able to navigate through most indoor environments without much difficulty. In very cluttered areas, Clyde sometimes becomes confused, but he rarely gets trapped or stuck on top of an obstacle.

The bump sensor design is one of Clyde's best features, allowing him to survive in environments where other robots may fail. The neural net system also worked very well, and produced some very interesting and complex behaviors. The object avoidance and migrate behaviors integrated well with the neural net, but was very difficult to integrate the other behaviors using the neural net. Because these behaviors require complete control of the robot for an extended period of time, I had to use the subsumption form of arbitration with these routines. The only behavior that I am not really satisfied with is the stall detection behavior. I believe this idea has promise, but it probably needs an adaptive algorithm to calibrate itself.

References

Robotics and Technical Information

Jones, Joseph L., and Anita M. Flynn, *Mobile Robots: Inspiration to Implementation.* Wellesley, MA: A K Peters, 1993.

ImageCraft ICC11 Users Manual v3.6. Sunnyvale, CA: ImageCraft, 1996.

*Machine Intelligence Lab Web Site. h*ttp://www.mil.ufl.edu

Ideas and Inspiration

Doty, Keith L., *IMDL Class Lectures*. Gainesville, FL: University of Florida, Fall 1996

Rossey, Lee, *IMDL Class Lecture: Neural Nets*. Gainesville, FL: University of Florida, Fall 1996

Electronic Components

All Electronics Corp.

14928 Oxnard Street

Van Nuys, CA 91411

Digikey Corporation

701 Brooks Ave. South

Thief River Falls, MN 56701-0677

Electronics Plus

2026 SW 34th Street

Gainesville FL

Skipper Electronics

3708 Newberry Rd.

Gainesville, FL

Radio Shack

Several Gainesville Locations

Appendix A: Compass Sensor Design

Introduction

The objective of this sensor design is to provide an accurate method for monitoring robot rotation. There are existing sensors for this purpose such as rate gyros and digital compasses. However, these sensors are expensive -- in the range of $100. The sensor design addressed in this report -- a digital compass -- is about one-tenth of the cost of a typical market sensor. Furthermore, because the digital compass will be implemented on a robot, the design needs to be compact. This can accomplished by mounting the sensor in a 35 mm film canister with the interface circuitry on a two inch by one inch printed circuit board.

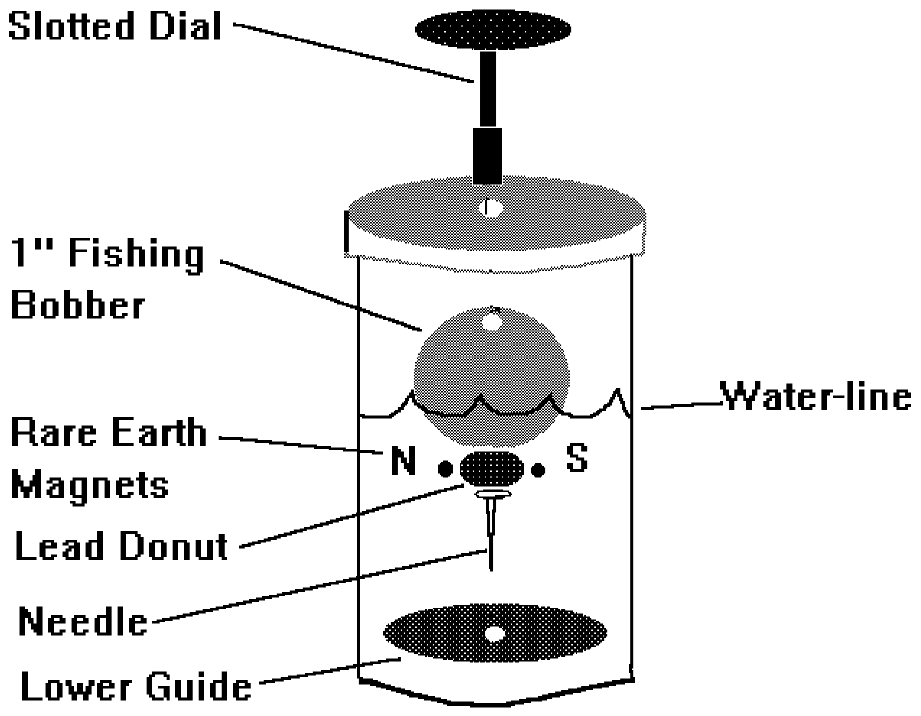
The digital compass design was realized by hacking an inexpensive Logitech PC mouse. The hack allows the use of a precision slotted wheel encoder to monitor the orientation of a rare earth magnet which is suspended in a 35 mm film canister. Another useful feature of the PC mouse hack is the serial interface with the microprocessor. Only one data line is required to transmit data to the 68HC11.

Sensor Description

Mechanical Design

The physical design of the digital compass is light-weight and compact. The magnet and bobber assembly is contained in a 35 mm film canister. A diagram of the sensor is shown in Figure 1. Two rare earth magnets are epoxied to a lead donut, and this lead donut attaches to the bottom of the fishing bobber. The slotted encoder wheel then attaches to the top of the bobber and projects through the lid of the film canister. A lower and upper guide are necessary to center the bobber and the encoder wheel. These guides also serve to restrict any movement perpendicular to the axis of the assembly. The lower guide is fashioned from hard plastic and has a hole in its center to align the needle projecting from the underside of the bobber. The upper guide -- not shown in Figure 1 -- is a guide from the inside of the mouse housing. This guide fits the shaft of the encoder wheel. A second compass was designed using a needle shaft and a slightly different upper guide.

Figure 5: Compass Mechanical Design



On one corner of the PC mouse printed circuit board, an IR emitter/detector pair extends from the board. The IR emitter/detector pair was cut away from the printed circuit board. Then the pair was positioned on the lid of the film canister and in-line with the slots on the encoder wheel. Hot glue held the IR emitter detector pair in place and wires were soldered onto the pair to connect it with the original PC mouse printed circuit board.

Microprocessor Interface

The compass uses the electronics from a Microsoft two button mouse to communicate with the Motorola 68HC11 microprocessor via the processor's Serial Communications Interface (SCI) system. The serial interface makes this sensor very simple to interface, and very universal. The compass was designed for use with a M68HC11 microprocessor, but it can be interfaced to almost any computer system that has an asynchronous serial port.

Serial Protocol

A standard two button mouse uses the serial transmission protocol shown in Figure 2. Each mouse transmission consists of a series of three data bytes that tell the amount and direction of rotation for each of the mouse's two encoder wheels. The mouse only transmits data when movement or a button action has been detected.

Figure 6: Mouse Transmission Protocol

```
     Microsoft Mouse Operation


     Serial UART: 1200 baud, data=7, stop=1, parity=none


     Mouse Protocol of Transmission
             bit:    7  6  5  4  3  2  1  0
     byte 1  (sync)  0  1  L  R y7 y6 x7 x6
     byte 2  (dX)    0  0 x5 x4 x3 x2 x1 x0
     byte 3  (dY)    0  0 y5 y4 y3 y2 y1 y0


     Notes:  - all dx, dy, are two's complement binary
numbers
```

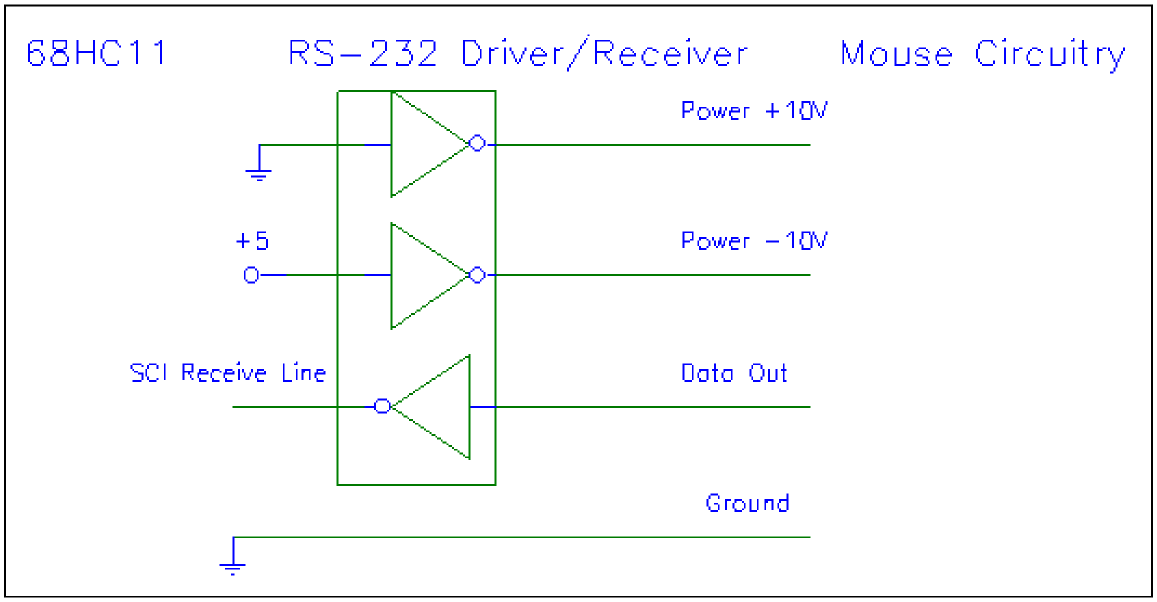
The 68HC11 does not directly support the mouse's "7N1" serial protocol, so the SCI interface uses an "8N1" protocol instead. Fortunately, this protocol mismatch works just fine, and bit 8 of each data byte is always received as a logic "1".

Hardware Interface

The mouse transmits data using RS-232 voltage levels, so we used the MC145407 RS-232 driver/receiver on the Motorola EVBU board for voltage level conversion. The EVBU board uses one driver for serial transmission, and one receiver for serial reception. We placed a mechanical switch in series with the RS-232 receive line so that the user may select the source of the signal entering the 68HC11 SCI receive pin. The switch selects either the compass or the EVBU DB25 connector, which is used for PC communication.

The microprocessor also needs to supply power to the mouse hardware. The mouse uses two RS-232 lines (one +10 V, and one -10 V) as power supplies. We used the two unused line drivers on the MC145407 chip to provide these voltage levels. A block diagram of the hardware interface is shown in Figure 3.

Figure 7: Hardware Interface for Compass Electronics



Software Interface

The software interface for the compass consists of an initialization routine for enabling the compass, an interrupt service routine for handling the incoming data, and an output variable that contains the current heading. The initialization routine configures the SCI system to interrupt the processor on each incoming data byte. This means that no polling is required, and the processor will only be interrupted when the compass heading changes. The interrupt service routine (ISR) keeps track of which data byte is being processed and reads the data byte from the SCI port. The ISR then does some bit manipulation to piece together the change in heading information, and then calculates the new compass orientation.

Experimental Procedure

We tested the compass by rotating it to several headings and measuring the error in the compass readings. This was done by marking headings on a test stand, rotating the compass to each heading, and recording the compass reading. This allowed us to determine the average errors for each heading orientation.

We then tested the compass for cumulative error by rotating the compass many revolutions in one direction. We also checked the compass for tilt error, although we did not do a formal experiment.

Results

The data from the compass tests are shown in Tables 1 and 2. Figures 4 and 5 show plots of the average error in the compass reading for each test heading. The compass seemed to have a large error at some orientations. The cause of this error could be a slight compass tilt, or some magnetic attraction, but we are not certain. The smaller errors in most of the readings are due to friction or encoder resolution (about 2 degrees per pulse of the encoder).

Table 1: Compass #1 Test Data

Actual Heading	Trial 1	Trial 2	Trial 3	Trial 4	Avg. Error
0	358	0	358	0	1
30	24	24	21	26	6.25
60	59	51	53	53	6
90	90	80	80	78	8
120	120	101	103	101	13.75
150	147	130	143	130	12.5
180	178	176	172	176	4.5
210	218	206	206	206	3
240	247	239	237	239	3
270	273	270	268	266	2.5
300	302	298	302	300	1.5
330	331	331	333	331	1.5
360	358	0	0	358	1

Figure 8: Compass #1 Error Plot

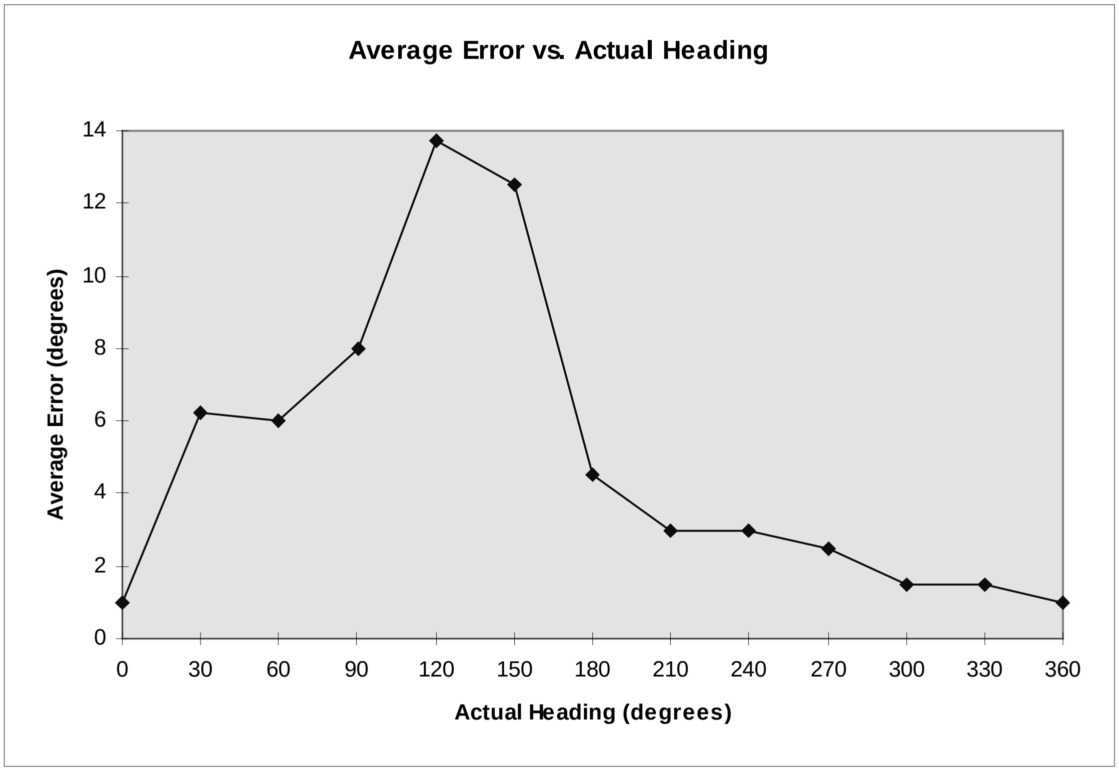
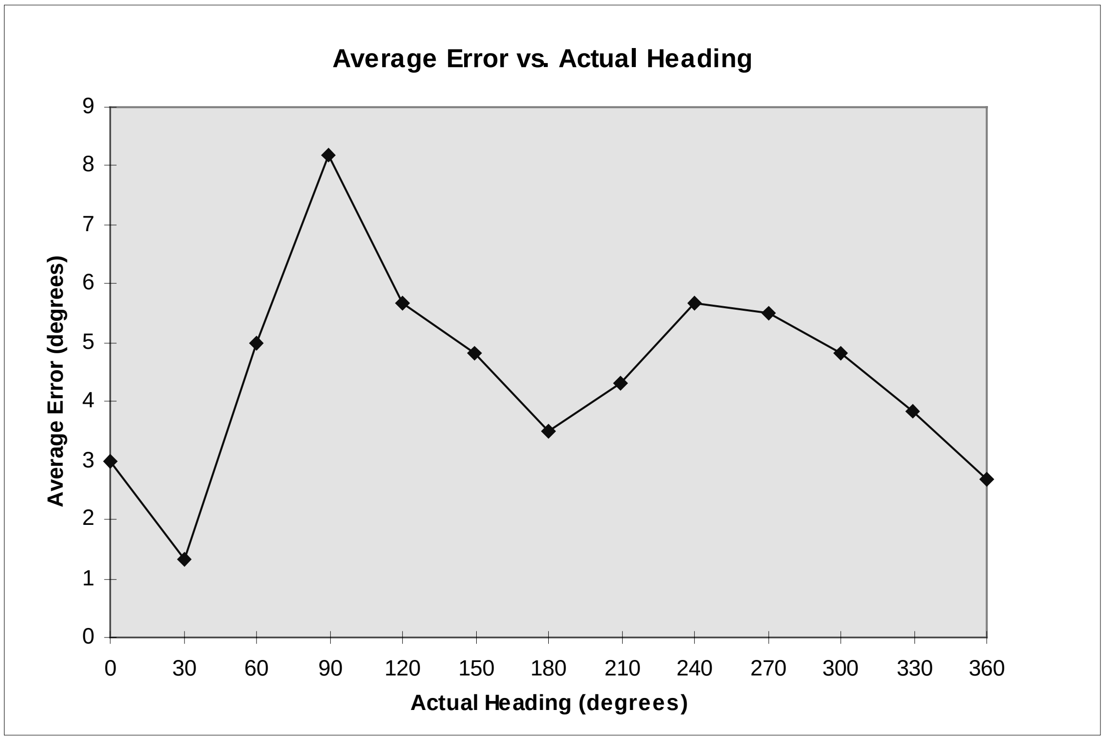


Table 2: Compass #2 Test Data

Actual Heading	Trial 1	Trial 2	Trial 3	Trial 4	Trial 5	Trial 6	Avg. Error
0	0	0	358	358	358	348	3
30	28	32	26	30	30	28	1.33
60	67	63	65	67	70	65	5
90	99	105	95	105	95	99	8.17
120	128	132	124	130	126	122	5.67
150	151	160	151	162	155	149	4.83
180	178	187	183	189	180	178	3.5

210	206	220	210	220	212	206	4.33
240	235	250	233	247	233	243	5.67
270	262	277	266	281	262	273	5.5
300	296	308	296	304	291	304	4.83
330	321	333	321	333	325	327	3.83
360	0	0	358	358	354	354	2.67

Figure 9: Compass #2 Error Plot



The compass showed no cumulative error over many rotations, so it worked well in this respect. The compass has a very large error when it is tilted from vertical, so this is a great consideration in mounting and using the compass. The compass also uses very strong magnets, so it must be mounted away from any type of ferrous materials.

Conclusion

The digital compass design yielded mixed results. The design was successful in several areas. The design was inexpensive – approximately $12 – and very compact. The compass was also easy to interface with the microprocessor and required minimal processing time. However, the compass had some problems with accuracy. This was largely due to friction on the compass guides and imprecise assembly techniques. The performance could be improved with a more refined upper guide design and the use of precision tools to construct the sensor. Although we did not meet our design goals in terms of accuracy, the digital compass should still be useful in many applications.

ICC11 Compass Interface Routine

```
/*************************************************************************/
/* Compass Interface Routines                                            */
/*************************************************************************/

/*-----------------------------------------------------------------------*/
/* Include files                                                         */
/*-----------------------------------------------------------------------*/
#include "serial.c"

/*-----------------------------------------------------------------------*/
/* Global Output Variables                                                       */
/*-----------------------------------------------------------------------*/

/* Current Heading                                                       */
     unsigned int heading=0;

/* Current Speed                                                         */
     int speed = 0;

/*-----------------------------------------------------------------------*/
/* Private Variables                                                     */
/*-----------------------------------------------------------------------*/

/* Current data byte                                                     */
     int byteNumber=1;

/* Current Pulse Count                                                   */
     int pulseCount=0;

/* Pulses per compass revolution                                         */
     pulsesPerRev = 172;

/*-----------------------------------------------------------------------*/
/* Initialization Routine                                                */
```

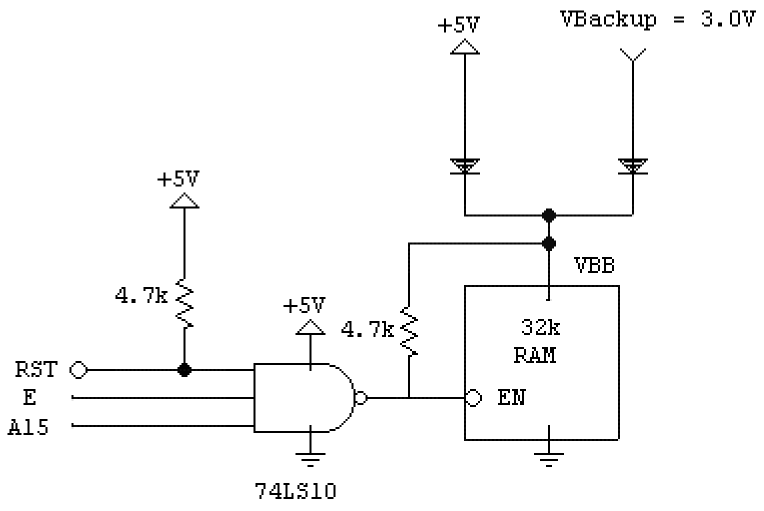
```
/*------------------------------------------------------------------------*/
void initCompass()
  {
/*  Initialize variables                                                  */
      byteNumber = 1;
      pulseCount=0;
      heading = 0;

/*  Set up serial port                                                    */
      BAUD = Baud1200;
      SCCR1 = Prot8N1;
      SCCR2= PollT_IntR;
  }

/*------------------------------------------------------------------------*/
/* Interrupt Service Routine                                              */
/*------------------------------------------------------------------------*/

#define SCI_ISR compassISR
#pragma interrupt_handler compassISR

void compassISR(void)
  {
/*  Check if we have an incoming data byte                                */
      if (SCSR & RDRF)
        {

/*        Change in heading                                               */
            static char dx=0;

/*        Read incoming data byte                                         */
            char inByte = SCDR;

/*        Process current data byte                                       */
            switch (byteNumber)
              {
                case 1 :
                  dx = (inByte & 0x03) << 6;
                  break;
                case 2 :
                  dx = dx + (inByte & 0x7F);
                  pulseCount = pulseCount + dx;
                  if (pulseCount < 0)
                    pulseCount = pulsesPerRev + pulseCount;
                  else if (pulseCount > pulsesPerRev-1)
                    pulseCount = pulseCount - pulsesPerRev;
                  speed = -dx;
                  break;
                case 3 :
                  break;
                default:
                  break;
              };

/*        Update byte counter                                           */
            byteNumber++;
            if (byteNumber>3 || byteNumber < 1)
              byteNumber = 1;
        }
      else
        {
/*        Clear interrupt flag                                          */
            int inByte = SCDR;
        };
  }

/***********************************************************************/
/* Compass Test Program                                                */
/***********************************************************************/

/*--------------------------------------------------------------------*/
/* Include files                                                      */
/*--------------------------------------------------------------------*/
#include <stdio.h>

/*--------------------------------------------------------------------*/
```

```
/* Test Routine                                                          */
/*----------------------------------------------------------------------*/
main()
  {
    int oldPulses;

/*  Set up compass and enable interrupts                                 */
      initCompass();
      printf("The Compass is ready...\n");
      INTR_ON();

/*  Continuously Display Compass Data                                    */
      oldPulses=pulseCount;
      while(1)
        {
          while (oldPulses == pulseCount)
            {};
          oldPulses=pulseCount;
          heading = 360 - (unsigned int)(pulseCount+1)*360/pulsesPerRev;
          printf("%d %d\n", heading, speed);
        };
  }



/*----------------------------------------------------------------------*/
/* Set up interrupt vectors                                              */
/*----------------------------------------------------------------------*/
#include "vectors1.c"
```

Appendix B: Battery Back-up Circuit

I designed a battery back-up circuit for the 32k RAM on ME11 expansion board. The initial design for this circuit came from a New Micros Inc. (NMI) 68HC11 board. I modified this circuit to suit my needs and match the parts that I had available. The final circuit I used (see Figure B-1) looks quite a bit different from the original NMI circuit.

Figure B-1: Battery Back-up Circuit



My back-up circuit uses two power supply busses: one for the 32k RAM, and one for the main 68HC11 circuitry. When the main 5V supply is powered by the voltage regulator, the RAM power is supplied via a diode connected to this supply. When the main power supply drops below about 4V, the low-voltage inhibit (LVI) chip on the 68HC11 board holds down the reset (RST) line low. Since the output of the 3-input NAND gate goes high when this occurs, the contents of the 32k RAM will not be destroyed by random memory writes as the battery voltage drops. When the main power supply drops below about 3V, the RAM power will be supplied by the 3V back-up power supply through a second diode. When the main power supply voltage drops low enough that the NAND gate can no longer operate, the chip enable of the RAM will be held high by a 4.7k pull-up resistor. In order for this last condition to occur, an LSTTL NAND gate must be used instead of an HCMOS chip. This has to do with the output resistance of the chips when no power supply is applied.

Appendix C: Main Program Code

```
/**************************************************************************/
/**************************************************************************/
/*                                                                        */
/* Robot Program                                                          */
/*                                                                        */
/* Features:  IR Magnitude Obstacle Avoidance (Neural Net)                */
/*            Delta IR Obstacle Avoidance (NN)                            */
/*            Bump Sensor Object Avoidance (NN)                           */
/*            Frustrated Behavior (Subsumption)                           */
/*            Escape Behavior (Subsumption)                               */
/*            Heading Seek Behavior (NN)                                  */
/*            Heading Repel Behavior (NN)                                 */
/*                                                                        */
/**************************************************************************/
/**************************************************************************/

/*------------------------------------------------------------------------*/
/* Include files                                                          */
/*------------------------------------------------------------------------*/
  #include <multiTsk.h>
  #include <timekeep.h>
  #include <serio.h>
  #include <motcontr.h>
  #include <neural.c>
  #include <function.h>

  #include "sensors.c"
  #include "nnconfig.c"
  #include "move.c"



/*------------------------------------------------------------------------*/
/* Defines                                                                */
/*------------------------------------------------------------------------*/
```

```
  #define motorTimeConstant 4
  #define motorSamplingPeriod 50

/**********************************************************************/
/*                                                                    */
/* IR Opinions                                                        */
/*                                                                    */
/* Description: Sets neural net opinions for obstacle avoidance based on  */
/*              the magnitude of the readings for each IR sensor.         */
/*                                                                    */
/**********************************************************************/

void setIrOpinion()
  {
/*--------------------------------------------------------------------*/
/*  Variables                                                         */
/*--------------------------------------------------------------------*/
      int subject;
      int irIndex;

/*--------------------------------------------------------------------*/
/*  Set IR magnitude opinion for each turn rate subject               */
/*--------------------------------------------------------------------*/
      for (subject=0; subject<numTurnSubjects; subject++)
        {
          for (irIndex=0; irIndex<numIR; irIndex++)
            {
              turnOpinion[subject] += irTurnWeight[irIndex] *
                  pwLinear(ir[irIndex],irTurnTF[irIndex][subject],100);
            };
        };

/*--------------------------------------------------------------------*/
/*  Set IR magnitude opinion for each speed subject                   */
/*--------------------------------------------------------------------*/
          for (subject=0; subject<numSpeedSubjects; subject++)
            {
              for (irIndex=0; irIndex<numIR; irIndex++)
```

```
                {
                 speedOpinion[subject] += irSpeedWeight[irIndex] *
                      pwLinear(ir[irIndex],irSpeedTF[irIndex][subject],100);
                };
            };
  }

/***********************************************************************/
/*                                                                     */
/* Delta IR Opinions                                                   */
/*                                                                     */
/* Description: Sets neural net opinions for obstacle avoidance based on  */
/*              the change in the readings for each IR sensor.         */
/*                                                                     */
/***********************************************************************/


void setDeltaIrOpinion()
  {
/*---------------------------------------------------------------------*/
/*  Variables                                                          */
/*---------------------------------------------------------------------*/
      int subject;
      int irIndex;

/*---------------------------------------------------------------------*/
/*  Set delta IR opinion for this turn rate subject                    */
/*---------------------------------------------------------------------*/
      for (subject=0; subject<numTurnSubjects; subject++)
        {
         for (irIndex=0; irIndex<numIR; irIndex++)
           {
             turnOpinion[subject] += deltaIrTurnWeight[irIndex] *
                  pwLinear(deltaIR[irIndex],deltaIrTurnTF[irIndex][subject]
                  ,100);
            };
        };
```

```
/*-----------------------------------------------------------------------*/
/*  Set delta IR opinion for this speed subject                          */
/*-----------------------------------------------------------------------*/
      for (subject=0; subject<numSpeedSubjects; subject++)
        {
         for (irIndex=0; irIndex<numIR; irIndex++)
           {
             speedOpinion[subject] += deltaIrSpeedWeight[irIndex] *
                  pwLinear(deltaIR[irIndex],deltaIrSpeedTF[irIndex][subject]
                  ,100);
           };
        };
  }

/************************************************************************/
/*                                                                      */
/* Bump Opinions                                                        */
/*                                                                      */
/* Description: Sets neural net opinions for obstacle avoidance based on  */
/*              the bump sensors.                                       */
/*                                                                      */
/************************************************************************/

void setBumpOpinion()
  {
/*-----------------------------------------------------------------------*/
/*  Variables                                                            */
/*-----------------------------------------------------------------------*/
      int subject;
      int bumpIndex;

/*-----------------------------------------------------------------------*/
/*  Adjust motor time constant so robot reacts quickly when bump occurs   */
/*-----------------------------------------------------------------------*/
      if (anyBump())
        setMotorTC(1);
      else
        setMotorTC(motorTimeConstant);
```

```
/*-------------------------------------------------------------------------*/
/*  Set opinions for each bump sensor on each turn rate subject           */
/*-------------------------------------------------------------------------*/
      for (subject=0; subject<numTurnSubjects; subject++)
        {
          for (bumpIndex=0; bumpIndex<numBump; bumpIndex++)
            {
              turnOpinion[subject] += bumpTurnWeight[bumpIndex] *
                   bump[bumpIndex] * bumpTurnOpinion[bumpIndex][subject];
            };
        };

/*-------------------------------------------------------------------------*/
/*  Set opinions for each bump sensor on each speed subject               */
/*-------------------------------------------------------------------------*/
      for (subject=0; subject<numSpeedSubjects; subject++)
        {
          for (bumpIndex=0; bumpIndex<numBump; bumpIndex++)
            {
              speedOpinion[subject] += bumpSpeedWeight[bumpIndex] *
                   bump[bumpIndex] * bumpSpeedOpinion[bumpIndex][subject];
            };
        };

  }

/***************************************************************************/
/*                                                                         */
/* Seek Heading Opinion                                                    */
/*                                                                         */
/* Description: Sets neural net opinions that cause the robot to seek     */
/*              the heading specified by the seekDirection parameter.     */
/*                                                                         */
/***************************************************************************/
void setSeekHeadingOpinion(int seekDirection)
  {
    int diff = headingDiff(getHeading(),seekDirection);
    turnOpinion[hardLeft] +=  headingSeekTurnWeight *
        pwLinear(diff,headingSeekHL,180);
    turnOpinion[softLeft] += headingSeekTurnWeight *
        pwLinear(diff,headingSeekSL,180);
    turnOpinion[straight] += headingSeekTurnWeight *
        pwLinear(diff,headingSeekS,180);
    turnOpinion[softRight] += headingSeekTurnWeight *
        pwLinear(diff,headingSeekSR,180);
    turnOpinion[hardRight] += headingSeekTurnWeight *
        pwLinear(diff,headingSeekHR,180);
  }

/**********************************************************************/
/*                                                                    */
/* Frustrated Opinion                                                 */
/*                                                                    */
/* Description: If not much forward motion is detected in a certain   */
/*              period of time, the robot get frustrated and spins    */
/*              a random amount.                                      */
/*                                                                    */
/**********************************************************************/

/*--------------------------------------------------------------------*/
/*  Private Variables                                                 */
/*--------------------------------------------------------------------*/
  int oldLeftCount, oldRightCount;
  unsigned int lastFrustratedTime=0;
  int lastFrustHeading;

void setFrustratedOpinion()
  {
/*--------------------------------------------------------------------*/
/*  Execute this routine once every 16 seconds                        */
/*--------------------------------------------------------------------*/
      if (getMilliseconds() - lastFrustratedTime > 16000)
          {
            int leftDistance = getEncoder(left) - oldLeftCount;
            int rightDistance = getEncoder(right) - oldRightCount;

            printf("l %d  r %d  \n",leftDistance,rightDistance);

            if ( abs(leftDistance + rightDistance) < 160   &&
                 abs(headingDiff(getHeading(),lastFrustHeading)) < 70)
              {
                printf("I’m frustrated!\n");

                halt();
                delay(1000);
                spin(left,random(135,225));
                halt();
              };
            oldLeftCount = getEncoder(left);
            oldRightCount = getEncoder(right);
            lastFrustratedTime = getMilliseconds();
            lastFrustHeading = getHeading();
          };
  }

/**************************************************************************/
/*                                                                        */
/* Escape Opinion                                                         */
/*                                                                        */
/**************************************************************************/



/*------------------------------------------------------------------------*/
/*  Private Variables                                                     */
/*------------------------------------------------------------------------*/
  int bumpReleased;
  int lastBumpHeading, lastBumpHeading2;
  unsigned int lastBumpTime, lastBumpTime2, lastBumpAngle;
  int escaping;

void setEscapeOpinion()
  {
```

```
/*------------------------------------------------------------------------*/
/*  Variables                                                             */
/*------------------------------------------------------------------------*/
      int diff;
      int bumpAngle;

    escaping = 0;
    if (anyBump() == 0)
      bumpReleased = 1;
    if (anyBump() && bumpReleased==1)
      {
            int bumpHeading;

            halt();

            bumpAngle = getBumpAngle();
            bumpHeading = getHeading() + bumpAngle;

             if (bumpHeading > 359)
               bumpHeading = bumpHeading - 360;
             else if (bumpHeading < 0)
               bumpHeading = bumpHeading + 360;

            printf("hdg: %d  %d  %d   \n",bumpHeading,lastBumpHeading,lastBumpHeading2);
            printf("time: %d  %d  %d   \n",getMilliseconds(),lastBumpTime,lastBumpTime2);
            printf("bumpAngle: %d  hdgDiff: %d
\n\n",bumpAngle,headingDiff(bumpHeading,lastBumpHeading));

/*            if (getMilliseconds() - lastBumpTime < 6000
                && abs(headingDiff(bumpHeading,lastBumpHeading)) <= 45
                && abs(bumpAngle-lastBumpAngle) <= 45 )
              {
                escaping = 1;
                 printf("I'm escaping!\n\n");
                if (bumpAngle > 0)
                  spin(left,random(60,180));
                else
                  spin(right,random(60,180));
              }
*/
            if (getMilliseconds() - lastBumpTime2 < 10000
                && abs(headingDiff(bumpHeading,lastBumpHeading2)) < 20)
              {
                escaping = 1;
                printf("I’m escaping!\n\n");
                if (headingDiff(bumpHeading,lastBumpHeading) > 0
                    && (bumpAngle<=90 && bumpAngle >= -90) )
                  spin(right,random(45,225));
                else if (headingDiff(bumpHeading,lastBumpHeading) > 0
                    && (bumpAngle>90 || bumpAngle < -90) )
                  spin(left,random(45,225));
                else if (headingDiff(bumpHeading,lastBumpHeading) < 0
                    && (bumpAngle<=90 && bumpAngle >= -90) )
                  spin(left,random(45,225));
                else if (headingDiff(bumpHeading,lastBumpHeading) < 0
                    && (bumpAngle>90 || bumpAngle < -90) )
                  spin(right,random(45,225));
              };

              lastBumpAngle = bumpAngle;
              lastBumpTime2 = lastBumpTime;
              lastBumpHeading2 = lastBumpHeading;
              lastBumpTime = getMilliseconds();
              lastBumpHeading = bumpHeading;
              bumpReleased = 0;
      };
  }

/***************************************************************************/
/*                                                                         */
/* Stall Opinion                                                           */
/*                                                                         */
/***************************************************************************/

int speedDuty[] =
  {  0,  0,
    10,  0,
    20,  1,
    30,  8,
    40, 11,
    50, 13,
    60, 15,
    70, 16,
    80, 17,
    90, 18,
   100, 18
  };

/*----------------------------------------------------------------------*/
/*  Calibrate the stall speed array                                     */
/*----------------------------------------------------------------------*/
void calibrateStall()
  {
    int i,l,r,l1,r1;

    motor(left,0);
    motor(right,0);

    for (i=0; i<=6; i++)
      {
        motor(left,10*i);
        motor(right,-10*i);
        delay(1000);
        l = getEncoder(left);
        r = getEncoder(right);
        delay(5000);
        motor(left,0);
        motor(right,0);
        l1 = getEncoder(left);
        r1 = getEncoder(right);
        speedDuty[indexMatrix2(speedDuty,i,1)] = (l1-l)/5;
      };

    delay(9000);
    for (i=0; i<=10; i++)
      {
        printf(" %d | %d, %d
\n",i,elementMatrix2(speedDuty,i,0),elementMatrix2(speedDuty,i,1));
      };

  }



/*------------------------------------------------------------------------*/
/*  Monitor Motors for stall condition                                    */
/*------------------------------------------------------------------------*/

int lastStallHeading=0;
unsigned int lastStallTime = 0;

void stallDetect()
  {
/*------------------------------------------------------------------------*/
/*  Execute this section once every 2 seconds                             */
/*------------------------------------------------------------------------*/
      if (getMilliseconds() - lastStallTime > 2000)
        {
/*          int leftChange = getEncoder(left) - lastLeftPulses;
          int rightChange = getEncoder(right) - lastRightPulses;
          int encoderDegrees = (leftChange - rightChange) *
                               360 / pulsesPerTurnRev;



          if (abs(getSpeed(left)-getSpeed(right)) > 20 &&
abs(headingDiff(getHeading(),oldHeading)) < 5)
*/
        int lduty = getAvgDuty(left);
        int rduty = getAvgDuty(right);
        int lduty1 = getOldAvgDuty(left,0);
        int rduty1 = getOldAvgDuty(right,0);
        int tableLspeed = pwLinear(abs(lduty),speedDuty,100);
        int tableRspeed = pwLinear(abs(rduty),speedDuty,100)-1;
        int tableLspeed1 = pwLinear(abs(lduty1),speedDuty,100);
        int tableRspeed1 = pwLinear(abs(rduty1),speedDuty,100)-1;
        int lspeed = abs(getSpeed(left));
        int rspeed = abs(getSpeed(right));
        int lspeed1 = abs(getOldSpeed(left,0));
        int rspeed1 = abs(getOldSpeed(right,0));
        int leftStall = abs(lspeed-tableLspeed) > 2   &&
                        abs(lspeed1-tableLspeed1) > 2 &&
                        abs(lduty-lduty1) <= 10;
        int rightStall = abs(rspeed-tableRspeed) > 2   &&
                         abs(rspeed1-tableRspeed1) > 2 &&
                         abs(rduty-rduty1) <= 10;

       if ( (leftStall || rightStall ) && !anyBump())
            {
              halt();
        printf("Motors are stalled...\n");
        printf("\nleft  : %d  %d  %d  \n",lduty,lspeed,tableLspeed);
        printf("left1 : %d  %d  %d  \n",lduty1,lspeed1,tableLspeed1);
        printf("right : %d  %d  %d  \n",rduty,rspeed,tableRspeed);
        printf("right1: %d  %d  %d  \n",rduty1,rspeed1,tableRspeed1);
              rev();
              delayConditionFalse(random(2000,4000),*anyBumpNow);
              halt();
              spin(left,random(45,135));
            };
        lastStallHeading = getHeading();
        lastStallTime = getMilliseconds();
    };
  }


/***************************************************************************/
/*                                                                         */
/* Neural Net Opinions                                                     */
/*                                                                         */
/***************************************************************************/
```

```
unsigned int et;
unsigned int oldt;

void getOpinionsProcess()
  {

    while(1)
      {
        et = getMilliseconds() - oldt;
        oldt = getMilliseconds();

        sampleSensors();

        setIrOpinion();
        setDeltaIrOpinion();

/*        setSeekHeadingOpinion(0); */
        setEscapeOpinion();
        setFrustratedOpinion();

        stallDetect();

        if (escaping == 0)
            setBumpOpinion();

        neuralNet();
        motionControl();
      };
  }


/***************************************************************************/
/* Display Debugging information                                           */
/***************************************************************************/

void display()
  {
    while(1)
      {
        printf("et: %d  ",et);
        printf("left : %d  %d  %d
",getAvgDuty(left),getSpeed(left),pwLinear(abs(getAvgDuty(left)),speedDuty,100));
        printf("right: %d  %d  %d
\n",getAvgDuty(right),getSpeed(right),pwLinear(abs(getAvgDuty(right)),speedDuty,100));
      };
  }

/***************************************************************************/
/* Robot Test Program                                                      */
/***************************************************************************/

main()
  {

/*  Set up serial port for output                                          */
      initSerial(BAUD_1200,PROT_8N1,POLLED_TRANSMIT);
      printf("Ready...\n");

/*  Initialize I/O devices                                                 */
      initTimeKeeper();
      initMultiTasking(RTI_32ms);
      initSensors();

/*  Initialize Control Processes                                           */
      initMotionControl(motorTimeConstant,200);
      initNeuralTF();

/*      calibrateStall();*/

/*      startProcess(*motionControlProcess,1);*/
/*      startProcess(*stallDetectProcess,1);*/
/*      startProcess(*display,1);*/
/*      startProcess(*motionControlProcess,1);*/
        startProcess(*getOpinionsProcess,4);
```

```
/*  Main Program Loop
                                              */
      motionControlProcess();
/*      getOpinionsProcess();*/

      while(1)
        {
/*          delayConditionFalse(4000,*anyBumpNow);*/

          fwd();
          delay(5000);

          halt();
          delay(1000);

          rev();
          delay(4000);

          halt();
          delay(1000);
        };
  }


/*--------------------------------------------------------------------*/
/* Set up interrupt vectors                                           */
/*--------------------------------------------------------------------*/
#include <vectors.c>

/**********************************************************************/
/*                                                                    */
/* Sensor System                                                      */
/*                                                                    */
/**********************************************************************/

/*--------------------------------------------------------------------*/
/* Includes                                                           */
/*--------------------------------------------------------------------*/
```

```
  #include <ir.c>

  #include <bump.c>

  #include <compass.h>

  #include <encoders.h>


/*-------------------------------------------------------------------------*/

/* Defines                                                                 */

/*-------------------------------------------------------------------------*/


/* Delay between sensor readings in milliseconds                          */

     #define sensorSampleRate 100


/* Names of IR Sensors                                                     */

     #define farRight 0

     #define nearRight 1

     #define center 2

     #define nearLeft 3

     #define farLeft 4


/* Names of Bump Sensors                                                   */

     #define rightRearInner 0

     #define rightRearOuter 1

     #define rightFrontOuter 2

     #define rightFrontInner 3

     #define leftFrontInner 4

     #define leftFrontOuter 5

     #define leftRearOuter 6

     #define leftRearInner 7



/*-------------------------------------------------------------------------*/

/* Init Sensors Process                                                    */

/*-------------------------------------------------------------------------*/

void initSensors()

  {

    initIR(5, 112, 128, 100, 0x1F);

    initBump(8);

    initCompass();
    initEncoders();
  }

/*------------------------------------------------------------------------*/
/* Sample Sensors                                                         */
/*------------------------------------------------------------------------*/
void sampleSensors()
  {
    sampleIR();
    sampleBump();
  }

/*------------------------------------------------------------------------*/
/* Sample Sensors Process                                                 */
/*------------------------------------------------------------------------*/
void sampleSensorsProcess()
  {
    while(1)
      {
        sampleSensors();
        delay(sensorSampleRate);
      };
  }

/*------------------------------------------------------------------------*/
/* Find Bump Angle Routine                                                */
/*------------------------------------------------------------------------*/
int getBumpAngle()
  {
     int bumpAngle=0;

     if (bump[rightFrontInner] && bump[leftFrontInner])
       bumpAngle = 0;
     else if (bump[rightFrontInner] && bump[rightFrontOuter])
       bumpAngle =   45;
     else if (bump[leftFrontInner] && bump[leftFrontOuter])
       bumpAngle =  - 45;
     else if (bump[rightFrontOuter] && bump[rightRearOuter])
```

```
      bumpAngle =   90;
    else if (bump[leftFrontOuter] && bump[leftRearOuter])
      bumpAngle =  - 90;
    else if (bump[rightRearOuter] && bump[rightRearInner])
      bumpAngle =   135;
    else if (bump[leftRearOuter] && bump[leftRearInner])
      bumpAngle =  - 135;
    else if (bump[rightRearInner] && bump[leftRearInner])
      bumpAngle =   180;
    else if (bump[rightFrontInner])
      bumpAngle =   22;
    else if (bump[leftFrontInner])
      bumpAngle =  - 22;
    else if (bump[rightFrontOuter])
      bumpAngle =   67;
    else if (bump[leftFrontOuter])
      bumpAngle =  - 67;
    else if (bump[rightRearOuter])
      bumpAngle =   112;
    else if (bump[leftRearOuter])
      bumpAngle =  - 112;
    else if (bump[rightRearInner])
      bumpAngle =   157;
    else if (bump[leftRearInner])
      bumpAngle =  - 157;

    return bumpAngle;
 }

/***************************************************************************/
/*                                                                         */
/* Movement Routines                                                       */
/*                                                                         */
/*                                                                         */
/***************************************************************************/

/*-----------------------------------------------------------------------*/
/* Includes                                                                */
```

```
/*-----------------------------------------------------------------------*/
#include <encoders.h>
#include <motcontr.h>

/*-----------------------------------------------------------------------*/
/* Calibration Variables                                                 */
/*-----------------------------------------------------------------------*/
int pulsesPerSpinRev = 45;
unsigned int maxTimePerSpinRev = 12000;

/*-----------------------------------------------------------------------*/
/* Spin Routine                                                          */
/*-----------------------------------------------------------------------*/

void spin(int direction, int degrees)
  {
    int deltaPulses;
    int dLeftPulses=0;
    int dRightPulses=0;

/*  Save current wheel position                                          */
      int startLeftPulses = getEncoder(left);
      int startRightPulses = getEncoder(right);

/*  Set maximum turn time                                                */
      unsigned int maxTurnTime = maxTimePerSpinRev / 360 * degrees;
      unsigned int startTime = getMilliseconds();

/*  Calculate number of pulses needed on each wheel for this turn        */
      deltaPulses = degrees * pulsesPerSpinRev / 360;

/*  Initiate Turn                                                        */
      if (direction == left)
        setDesiredTurnRate(-30);
      else
        setDesiredTurnRate(30);
      setDesiredSpeed(0);
/*  Turn until robot has rotated desired amount                          */
      while(  abs(dLeftPulses) < deltaPulses &&
              abs(dRightPulses) < deltaPulses  &&
              getMilliseconds() - startTime <= maxTurnTime )
       {
          dLeftPulses = getEncoder(left) - startLeftPulses;
          dRightPulses = getEncoder(right) - startRightPulses;
       };

/*  Stop turning                                                         */
      setDesiredTurnRate(0);
  }

/*----------------------------------------------------------------------*/
/* Forward Routine                                                       */
/*----------------------------------------------------------------------*/

void fwd()
  {
    setDesiredSpeed(30);
    setDesiredTurnRate(0);
  }

/*----------------------------------------------------------------------*/
/* Reverse Routine                                                       */
/*----------------------------------------------------------------------*/

void rev()
  {
    setDesiredSpeed(-30);
    setDesiredTurnRate(0);
  }

/*----------------------------------------------------------------------*/
/* Halt Routine                                                          */
/*----------------------------------------------------------------------*/

void halt()
```

```
  {
    setDesiredSpeed(0);
    setDesiredTurnRate(0);
  }
```

```
/*--------------------------------------------------------------------------*/
/* Includes                                                                 */
/*--------------------------------------------------------------------------*/
  #include "neuralTF.c"

/*--------------------------------------------------------------------------*/
/* Defines                                                                  */
/*--------------------------------------------------------------------------*/

/* Turn Rate Defines                                                        */
     #define straight 0
     #define hardLeft 1
     #define softLeft 2
     #define softRight 3
     #define hardRight 4

/* Speed Defines                                                            */
     #define medium 0
     #define slow 1
     #define verySlow 2
     #define fast 3
     #define reverse 4

/*--------------------------------------------------------------------------*/
/* Opinion Transfer Function Arrays                                         */
/*--------------------------------------------------------------------------*/
  int * irTurnTF[maxNumIR][maxNumTurnSubjects];
  int * irSpeedTF[maxNumIR][maxNumSpeedSubjects];
  int irTurnWeight[maxNumIR];
  int irSpeedWeight[maxNumIR];

  int * deltaIrTurnTF[maxNumIR][maxNumTurnSubjects];
  int * deltaIrSpeedTF[maxNumIR][maxNumSpeedSubjects];
  int deltaIrTurnWeight[maxNumIR];
  int deltaIrSpeedWeight[maxNumIR];

  int bumpTurnOpinion[maxNumBump][maxNumTurnSubjects];
  int bumpSpeedOpinion[maxNumBump][maxNumSpeedSubjects];
  int bumpTurnWeight[maxNumBump];
  int bumpSpeedWeight[maxNumBump];

  int headingSeekTurnWeight;

/**********************************************************************/
/*                                                                    */
/* Initialize Neural Net Variables and Opinion Transfer Functions         */
/*                                                                    */
/**********************************************************************/

void initNeuralTF()
  {
    int i;

/*--------------------------------------------------------------------*/
/*  Set up neural net variables                                       */
/*--------------------------------------------------------------------*/

/*    Set the number of subjects for each neural net                        */
        numTurnSubjects = 5;
        numSpeedSubjects = 5;

/*    Set the turn rates for each turn subject                              */
        turnMagnitude[hardLeft] = -35;
        turnMagnitude[softLeft] = -15;
        turnMagnitude[straight] = 0;
        turnMagnitude[softRight] = 15;
        turnMagnitude[hardRight] = 35;

/*    Set the forward speeds for each speed subject                       */
        speedMagnitude[fast] = 50;
        speedMagnitude[medium] = 30;
        speedMagnitude[slow] = 20;
        speedMagnitude[verySlow] = 10;
        speedMagnitude[reverse] = -15;

/*------------------------------------------------------------------------*/
/*  Set neural net turn weights for each neuron                           */
/*------------------------------------------------------------------------*/
        irTurnWeight[farLeft] = 1;
        irTurnWeight[nearLeft] = 3;
        irTurnWeight[center] = 6;
        irTurnWeight[nearRight] = 3;
        irTurnWeight[farRight] = 1;

        deltaIrTurnWeight[farLeft] = 1;
        deltaIrTurnWeight[nearLeft] = 3;
        deltaIrTurnWeight[center] = 6;
        deltaIrTurnWeight[nearRight] = 3;
        deltaIrTurnWeight[farRight] = 1;

        bumpTurnWeight[rightRearInner] = 10;
        bumpTurnWeight[rightRearOuter] = 10;
        bumpTurnWeight[rightFrontOuter] = 15;
        bumpTurnWeight[rightFrontInner] = 15;
        bumpTurnWeight[leftFrontInner] = 15;
        bumpTurnWeight[leftFrontOuter] = 15;
        bumpTurnWeight[leftRearOuter] = 10;
        bumpTurnWeight[leftRearInner] = 10;

        headingSeekTurnWeight = 5;

/*------------------------------------------------------------------------*/
/*  Set neural net speed weights for each neuron                          */
/*------------------------------------------------------------------------*/
```

```
        irSpeedWeight[farLeft] = 1;
        irSpeedWeight[nearLeft] = 1;
        irSpeedWeight[center] = 1;
        irSpeedWeight[nearRight] = 1;
        irSpeedWeight[farRight] = 1;

        deltaIrSpeedWeight[farLeft] = 4;
        deltaIrSpeedWeight[nearLeft] = 4;
        deltaIrSpeedWeight[center] = 4;
        deltaIrSpeedWeight[nearRight] = 4;
        deltaIrSpeedWeight[farRight] = 4;

        bumpSpeedWeight[rightRearInner] = 10;
        bumpSpeedWeight[rightRearOuter] = 10;
        bumpSpeedWeight[rightFrontOuter] = 15;
        bumpSpeedWeight[rightFrontInner] = 15;
        bumpSpeedWeight[leftFrontInner] = 15;
        bumpSpeedWeight[leftFrontOuter] = 15;
        bumpSpeedWeight[leftRearOuter] = 10;
        bumpSpeedWeight[leftRearInner] = 10;

/*-------------------------------------------------------------------------*/
/*  Create transfer functions for IR neurons                               */
/*-------------------------------------------------------------------------*/

/*    Turn rate transfer functions                                         */

        irTurnTF[farLeft][hardLeft]=rampNeg100;
        irTurnTF[farLeft][softLeft]=rampNeg75;
        irTurnTF[farLeft][straight]=rampNeg50;
        irTurnTF[farLeft][softRight]=peak60to80;
        irTurnTF[farLeft][hardRight]=peak80to100;

        irTurnTF[nearLeft][hardLeft]=rampNeg75;
        irTurnTF[nearLeft][softLeft]=rampNeg100;
        irTurnTF[nearLeft][straight]=rampNeg75;
        irTurnTF[nearLeft][softRight]=peak40to60;
        irTurnTF[nearLeft][hardRight]=peak60to100;
        irTurnTF[center][hardLeft]=peak50to100;
        irTurnTF[center][softLeft]=rampNeg75;
        irTurnTF[center][straight]=rampNeg100;
        irTurnTF[center][softRight]=rampNeg75;
        irTurnTF[center][hardRight]=peak50to100;

        irTurnTF[nearRight][hardLeft]=peak60to100;
        irTurnTF[nearRight][softLeft]=peak40to60;
        irTurnTF[nearRight][straight]=rampNeg75;
        irTurnTF[nearRight][softRight]=rampNeg100;
        irTurnTF[nearRight][hardRight]=rampNeg75;

        irTurnTF[farRight][hardLeft]=peak80to100;
        irTurnTF[farRight][softLeft]=peak60to80;
        irTurnTF[farRight][straight]=rampNeg50;
        irTurnTF[farRight][softRight]=rampNeg75;
        irTurnTF[farRight][hardRight]=rampNeg100;

/*    Speed transfer functions                                              */

        irSpeedTF[farLeft][fast]=peak0to10;
        irSpeedTF[farLeft][medium]=peak10to30;
        irSpeedTF[farLeft][slow]=peak30to60;
        irSpeedTF[farLeft][verySlow]=peak60to80a;
        irSpeedTF[farLeft][reverse]=peak80to100a;

        irSpeedTF[nearLeft][fast]=peak0to10;
        irSpeedTF[nearLeft][medium]=peak10to30;
        irSpeedTF[nearLeft][slow]=peak30to60;
        irSpeedTF[nearLeft][verySlow]=peak60to80a;
        irSpeedTF[nearLeft][reverse]=peak80to100a;

        irSpeedTF[center][fast]=peak0to10;
        irSpeedTF[center][medium]=peak10to30;
        irSpeedTF[center][slow]=peak30to60;
        irSpeedTF[center][verySlow]=peak60to80a;
        irSpeedTF[center][reverse]=peak80to100a;
        irSpeedTF[nearRight][fast]=peak0to10;

        irSpeedTF[nearRight][medium]=peak10to30;

        irSpeedTF[nearRight][slow]=peak30to60;

        irSpeedTF[nearRight][verySlow]=peak60to80a;

        irSpeedTF[nearRight][reverse]=peak80to100a;


        irSpeedTF[farRight][fast]=peak0to10;

        irSpeedTF[farRight][medium]=peak10to30;

        irSpeedTF[farRight][slow]=peak30to60;

        irSpeedTF[farRight][verySlow]=peak60to80a;

        irSpeedTF[farRight][reverse]=peak80to100a;




/*------------------------------------------------------------------------*/

/*  Create transfer functions for delta IR neurons                        */

/*------------------------------------------------------------------------*/


/*    Turn rate transfer functions                                        */


        deltaIrTurnTF[farLeft][hardLeft]=deltaIrTf0;

        deltaIrTurnTF[farLeft][softLeft]=deltaIrTf1;

        deltaIrTurnTF[farLeft][straight]=rampNeg50;

        deltaIrTurnTF[farLeft][softRight]=peak60to80;

        deltaIrTurnTF[farLeft][hardRight]=peak80to100;


        deltaIrTurnTF[nearLeft][hardLeft]=deltaIrTf3;

        deltaIrTurnTF[nearLeft][softLeft]=deltaIrTf2;

        deltaIrTurnTF[nearLeft][straight]=rampNeg75;

        deltaIrTurnTF[nearLeft][softRight]=peak40to60;

        deltaIrTurnTF[nearLeft][hardRight]=peak60to100;


        deltaIrTurnTF[center][hardLeft]=peak50to100;

        deltaIrTurnTF[center][softLeft]=rampNeg75;

        deltaIrTurnTF[center][straight]=rampNeg100;

        deltaIrTurnTF[center][softRight]=rampNeg75;

        deltaIrTurnTF[center][hardRight]=peak50to100;
        deltaIrTurnTF[nearRight][hardLeft]=peak60to100;
        deltaIrTurnTF[nearRight][softLeft]=peak40to60;
        deltaIrTurnTF[nearRight][straight]=rampNeg75;
        deltaIrTurnTF[nearRight][softRight]=deltaIrTf2;
        deltaIrTurnTF[nearRight][hardRight]=deltaIrTf3;

        deltaIrTurnTF[farRight][hardLeft]=peak80to100;
        deltaIrTurnTF[farRight][softLeft]=peak60to80;
        deltaIrTurnTF[farRight][straight]=rampNeg50;
        deltaIrTurnTF[farRight][softRight]=deltaIrTf1;
        deltaIrTurnTF[farRight][hardRight]=deltaIrTf0;


/*    Speed transfer functions                                          */

        deltaIrSpeedTF[farLeft][fast]=peak0to10;
        deltaIrSpeedTF[farLeft][medium]=peak10to30;
        deltaIrSpeedTF[farLeft][slow]=peak30to60;
        deltaIrSpeedTF[farLeft][verySlow]=peak60to80a;
        deltaIrSpeedTF[farLeft][reverse]=peak80to100a;

        deltaIrSpeedTF[nearLeft][fast]=peak0to10;
        deltaIrSpeedTF[nearLeft][medium]=peak10to30;
        deltaIrSpeedTF[nearLeft][slow]=peak30to60;
        deltaIrSpeedTF[nearLeft][verySlow]=peak60to80a;
        deltaIrSpeedTF[nearLeft][reverse]=peak80to100a;

        deltaIrSpeedTF[center][fast]=peak0to10;
        deltaIrSpeedTF[center][medium]=peak10to30;
        deltaIrSpeedTF[center][slow]=peak30to60;
        deltaIrSpeedTF[center][verySlow]=peak60to80a;
        deltaIrSpeedTF[center][reverse]=peak80to100a;

        deltaIrSpeedTF[nearRight][fast]=peak0to10;
        deltaIrSpeedTF[nearRight][medium]=peak10to30;
        deltaIrSpeedTF[nearRight][slow]=peak30to60;
        deltaIrSpeedTF[nearRight][verySlow]=peak60to80a;
        deltaIrSpeedTF[nearRight][reverse]=peak80to100a;
        deltaIrSpeedTF[farRight][fast]=peak0to10;
        deltaIrSpeedTF[farRight][medium]=peak10to30;
        deltaIrSpeedTF[farRight][slow]=peak30to60;
        deltaIrSpeedTF[farRight][verySlow]=peak60to80a;
        deltaIrSpeedTF[farRight][reverse]=peak80to100a;

/*-----------------------------------------------------------------------*/
/*  Set bump opinions                                                    */
/*-----------------------------------------------------------------------*/

/*    Turn Opinions                                                      */

        bumpTurnOpinion[rightRearInner][hardLeft] = 10;
        bumpTurnOpinion[rightRearInner][softLeft] = 50;
        bumpTurnOpinion[rightRearInner][straight] = 40;
        bumpTurnOpinion[rightRearInner][softRight] = -10;
        bumpTurnOpinion[rightRearInner][hardRight] = -30;

        bumpTurnOpinion[rightRearOuter][hardLeft] = 20;
        bumpTurnOpinion[rightRearOuter][softLeft] = 70;
        bumpTurnOpinion[rightRearOuter][straight] = -10;
        bumpTurnOpinion[rightRearOuter][softRight] = -60;
        bumpTurnOpinion[rightRearOuter][hardRight] = -90;

        bumpTurnOpinion[rightFrontOuter][hardLeft] = 40;
        bumpTurnOpinion[rightFrontOuter][softLeft] = 90;
        bumpTurnOpinion[rightFrontOuter][straight] = -80;
        bumpTurnOpinion[rightFrontOuter][softRight] = -90;
        bumpTurnOpinion[rightFrontOuter][hardRight] = -100;

        bumpTurnOpinion[rightFrontInner][hardLeft] = 100;
        bumpTurnOpinion[rightFrontInner][softLeft] = 50;
        bumpTurnOpinion[rightFrontInner][straight] = -90;
        bumpTurnOpinion[rightFrontInner][softRight] = -100;
        bumpTurnOpinion[rightFrontInner][hardRight] = -80;

        bumpTurnOpinion[leftFrontInner][hardLeft] = -80;
        bumpTurnOpinion[leftFrontInner][softLeft] = -100;
        bumpTurnOpinion[leftFrontInner][straight] = -90;
        bumpTurnOpinion[leftFrontInner][softRight] = 50;
        bumpTurnOpinion[leftFrontInner][hardRight] = 100;

        bumpTurnOpinion[leftFrontOuter][hardLeft] = -100;
        bumpTurnOpinion[leftFrontOuter][softLeft] = -90;
        bumpTurnOpinion[leftFrontOuter][straight] = -80;
        bumpTurnOpinion[leftFrontOuter][softRight] = 90;
        bumpTurnOpinion[leftFrontOuter][hardRight] = 40;

        bumpTurnOpinion[leftRearOuter][hardRight] = 20;
        bumpTurnOpinion[leftRearOuter][softRight] = 70;
        bumpTurnOpinion[leftRearOuter][straight] = -10;
        bumpTurnOpinion[leftRearOuter][softLeft] = -60;
        bumpTurnOpinion[leftRearOuter][hardLeft] = -90;

        bumpTurnOpinion[leftRearInner][hardRight] = 10;
        bumpTurnOpinion[leftRearInner][softRight] = 50;
        bumpTurnOpinion[leftRearInner][straight] = 40;
        bumpTurnOpinion[leftRearInner][softLeft] = -10;
        bumpTurnOpinion[leftRearInner][hardLeft] = -30;

/*    Speed Opinions                                                         */

        bumpSpeedOpinion[rightRearInner][fast] = 50;
        bumpSpeedOpinion[rightRearInner][medium] = 100;
        bumpSpeedOpinion[rightRearInner][slow] = 50;
        bumpSpeedOpinion[rightRearInner][verySlow] = 0;
        bumpSpeedOpinion[rightRearInner][reverse] = -100;

        bumpSpeedOpinion[rightRearOuter][fast] = 40;
        bumpSpeedOpinion[rightRearOuter][medium] = 100;
        bumpSpeedOpinion[rightRearOuter][slow] = 60;
        bumpSpeedOpinion[rightRearOuter][verySlow] = 10;
        bumpSpeedOpinion[rightRearOuter][reverse] = -90;

        bumpSpeedOpinion[rightFrontOuter][fast] = -90;
        bumpSpeedOpinion[rightFrontOuter][medium] = -80;
        bumpSpeedOpinion[rightFrontOuter][slow] = -10;
        bumpSpeedOpinion[rightFrontOuter][verySlow] = 0;
        bumpSpeedOpinion[rightFrontOuter][reverse] = 70;

        bumpSpeedOpinion[rightFrontInner][fast] = -100;
        bumpSpeedOpinion[rightFrontInner][medium] = -90;
        bumpSpeedOpinion[rightFrontInner][slow] = -50;
        bumpSpeedOpinion[rightFrontInner][verySlow] = -10;
        bumpSpeedOpinion[rightFrontInner][reverse] = 100;

        bumpSpeedOpinion[leftFrontInner][fast] = -100;
        bumpSpeedOpinion[leftFrontInner][medium] = -90;
        bumpSpeedOpinion[leftFrontInner][slow] = -50;
        bumpSpeedOpinion[leftFrontInner][verySlow] = -10;
        bumpSpeedOpinion[leftFrontInner][reverse] = 100;

        bumpSpeedOpinion[leftFrontOuter][fast] = -90;
        bumpSpeedOpinion[leftFrontOuter][medium] = -80;
        bumpSpeedOpinion[leftFrontOuter][slow] = -10;
        bumpSpeedOpinion[leftFrontOuter][verySlow] = 0;
        bumpSpeedOpinion[leftFrontOuter][reverse] = 70;

        bumpSpeedOpinion[leftRearOuter][fast] = 40;
        bumpSpeedOpinion[leftRearOuter][medium] = 100;
        bumpSpeedOpinion[leftRearOuter][slow] = 60;
        bumpSpeedOpinion[leftRearOuter][verySlow] = 10;
        bumpSpeedOpinion[leftRearOuter][reverse] = -90;

        bumpSpeedOpinion[leftRearInner][fast] = 50;
        bumpSpeedOpinion[leftRearInner][medium] = 100;
        bumpSpeedOpinion[leftRearInner][slow] = 0;
        bumpSpeedOpinion[leftRearInner][verySlow] = 50;
        bumpSpeedOpinion[leftRearInner][reverse] = -100;
}

/*------------------------------------------------------------------------*/
/* Define Transfer Functions (Piece-wise linear coordinates)              */
```

```
/*----------------------------------------------------------------------------*/
```

```
int headingSeekHL[] =
  {
    -179, -50,
    -135, -50,
    -90,    0,
     90,    0,
    135,  100,
    180,  100
  };

int headingSeekSL[] =
  {
    -179,   0,
    -135,   0,
    -90,  -50,
    -30,  -50,
      0,    0,
     30,  100,
     90,  100,
    135,    0,
    180,    0
  };

int headingSeekS[] =
  {
    -179,   0,
     -20,   0,
     -10,   25,
      10,   25,
      20,    0,
     180,    0
  };

int headingSeekSR[] =
  {
    -179,   0,
    -135,   0,
    -90,  100,
    -30,  100,
      0,    0,
     30,  -50,
     90,  -50,
    135,    0,
    180,    0
  };

int headingSeekHR[] =
  {
    -179, 100,
    -135, 100,
    -90,    0,
     90,    0,
    135,  -50,
    180,  -50
  };

int headingRepelHL[] =
  {
    -179, 0,
     -1,  0,
     0, -30,
    180, 0
  };

int headingRepelSL[] =
  {
    -179, 0,
     -1,  0,
     0, -50,
     90, 0,
    180, 0
  };

int headingRepelS[] =
  {
    -179, 0,
     -45, 0,
     0,  -100,
     45, 0,
    180, 0
  };

int headingRepelSR[] =
  {
    -179, 0,
     -90, 0,
     0,  -50,
     1,  0,
    180, 0
  };

int headingRepelHR[] =
  {
   -179, 0,
     0, -30,
     1,  0,
     180, 0
  };

int headingRepelSlow[] =
  {
   -179, 0,
    -40, 0,
    -30, 100,
    -20, 100,
    -10, 0,
     10, 0,
     20, 100,
     30, 100,
     40, 0,
     180, 0
  };

int headingRepelVerySlow[] =
  {
   -179, 0,
    -20, 0,
    -10, 100,
     10, 100,
     20, 0,
     180, 0
  };

int zero[] =
  {
     0,   0,
   100,   0
  };

int deltaIrTf1[] =
  {
  -100,   0,
   -50,   0,
   -40, 100,
   -15, 100,
     0,   0,
   100, -75
  };

int deltaIrTf0[] =
  {
  -100, 100,
   -50, 100,
   -40,   0,
     0,   0,
   100,-100
  };

int deltaIrTf2[] =
  {
  -100,   0,
   -60, 100,
   -30, 100,
     0,   0,
   100,-100
  };

int deltaIrTf3[] =
  {
  -100, 100,
   -60, 100,
   -30,   0,
     0,   0,
   100, -75
  };

int rampNeg100[] =
  {
     0,   0,
   100,-100
  };

int rampNeg75[] =
  {
     0,   0,
   100, -75
  };

int rampNeg50[] =
  {
     0,   0,
   100, -50
  };

int peak60to80[] =
  {
     0,   0,
    60, 100,
    80, 100,
   100,   0
  };

int peak80to90[] =
  {
     0,   0,
    80, 100,
    90, 100,
   100,   0
  };

int peak80to100[] =
  {
     0,   0,
    80, 100,
   100, 100
  };

int peak90to100[] =
  {
     0,   0,
    90, 100,
   100, 100
  };

int peak40to60[] =
  {
     0,   0,
    40, 100,
    60, 100,
   100,   0
  };

int peak50to100[] =
  {
     0,   0,
    50, 100,
   100, 100
  };
```

```
int peak60to100[] =
  {
     0,   0,
    60, 100,
   100, 100
  };

int peak0to10[] =
  {
     0, 10,
    10, 10,
    20,   0,
   100,   0
  };

int peak10to30[] =
  {
     0,   0,
    10, 10,
    30, 10,
    40,   0,
   100,   0
  };

int peak30to60[] =
  {
     0,   0,
    20,   0,
    30,  30,
    60,  30,
    70,   0,
   100,   0
  };

int peak60to80a[] =
  {
     0,   0,
    50,   0,
    60, 100,
    80, 100,
    90,   0,
   100,   0
  };

int peak80to100a[] =
  {
     0,   0,
    70,   0,
    80,  50,
   100,  50
  };
```