Simulating Basic Flock Behaviors Using Simple Autonomous Agents

Iván Zapata

EEL 5666: Intelligent Machine Design Lab

Dr. Keith L. Doty

TA: Scott Jantz

12/9/96

Table of Contents

Abstract 3
Executive Summary 4
Introduction 5
Integrated System 6
Mobile Platform 7
Actuation 7
Sensors 11
  Infrared Emitters and Detectors: 11
    Working Principles and implementation 12
  Bump sensors 13
Behaviors 14
  Object avoidance 14
  Robot Following 15
  Speed Regulation 16
  Collision Detection 16
Conclusion 18
Appendix 19
  Figures 19
  Source Code 20
    Motor driver routine: servotj.c 20
    Infrared system driver: irtj.c 23
    Main program: chase5.c 26

Abstract

This paper is on research done for the EEL5666: Intelligent Machine Design Laboratory course at the University of Florida. The first objective of this research was to build an autonomous agent platform that would be very simple at the hardware level, but would still provide all the functionality of larger scale, more hardware-intensive robots. The second objective of this project was to use several of these robots to simulate flock behaviors.

Executive Summary

The basic platform for each robot is a Novasoft Talrik Junior™ body. On this platform are mounted an M68HC11E2 processor, two motors, and a sensor suite including infrared detectors, emitters, and bump switches. There is no control hardware for either the motors or any of the sensors, rather, all of the robot's systems are directly controlled by the microprocessor. Reducing the amount of hardware in this fashion greatly minimizes the construction of the robot, which was one of the primary goals of this project. However, simplifying the hardware means that the robot's software must be more sophisticated, since it must deal with every aspect of operation of every subsystem. The author developed software drivers to control the motors and sensor systems, to the point where each robot was able to navigate any given space while avoiding obstacles. After this point, the robots could be programmed to seek each other and follow each other. The robots could essentially travel as a pack, with the leading robot deciding where the pack will go.

Introduction

The two main goals of this project were first to develop a robot that would be very simple to build, and then to program several of these robots to act as a ‘flock’. Three robots took part in this research, Huey, Dewey, and Louie. The basic robot design contains a very minimal amount of hardware: all components such as motors and sensors are directly connected to the robot’s processor. No extra hardware is used to drive, control, or modulate any external system. Because of the reduction in hardware, the software that controls these robots has to keep close control of every aspect relating to every subsystem, as well as to assure that no subsystem interferes with another. The basic flock behavior desired from the robots was for them to be able to travel as a group, and to be able to control the speed and “tightness” at which the group is traveling. The robots were able to recognize each other by detecting each other’s infrared signature, so each robot would follow any infrared bursts that were not its own. While traveling as a group, each robot is able to recognize whether there are other robots in front or behind, and with this information the robot can change its speed to allow robots to catch up to him or to pull away.

Integrated System

One of the goals of this project was to produce somewhat complex behaviors from a very simple robot, for which reason the Talrik Junior ™ (TJ) platform is the basis for all three robots. The TJ body was designed by Dr. Keith Doty with the intention of building a fully autonomous robot which would require an absolute minimum amount of hardware. An M68HC11E2 micro-controller is the brain of each robot, directly controlling all robot behaviors and subsystems. There are only two types of sensors on each TJ: 40kHz infrared detectors and bump sensors, however, these sensors are shared by several subsystems and behaviors. Below (Figure 1: Functional Block Diagram) is a

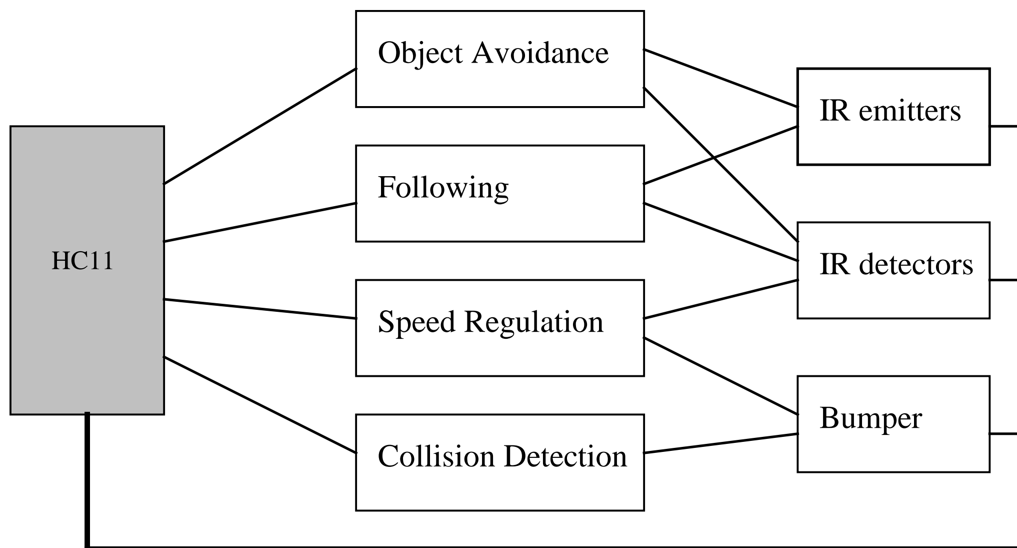
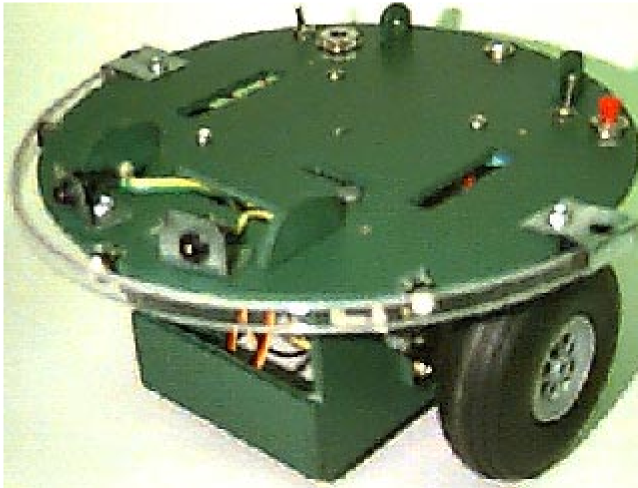


Figure 1: Functional Block Diagram

block diagram showing the relationships between the robot's behaviors and sensors.

Mobile Platform

The robot platform is a Talrik Junior™ body, which consists of a rectangular body with a disk-shaped top (see Figure 2: TJ body). The platform is designed to hold two motors



and a single chip computer board (Novasoft MSCC11). The two motors are lined up along the robot's center, so the robot can essentially spin in place. The body also has a compartment for six AA size batteries.

Figure 2: TJ body

Actuation

The only type of actuation present in each robot are two servo motors for locomotion. Since one of the specifications of this project was to have a minimal amount of hardware, servo motors were preferable to regular DC motors for control purposes. DC motors would have required some type of motor driver hardware, since the microprocessor could not possibly supply enough current to drive the motors. Servo motors, on the other hand, do not require any type of driver, they are controlled with a TTL-level signal, and drain power directly off a power supply. Servo motors, however, are usually used in applications that require precision within a limited range of motion, so they must be slightly modified so that they can provide an unlimited range of motion. Motion control in a servo is usually accomplished as follows: a periodic signal is sent to the servo's circuitry, telling it where to move within its range of motion (about 180 degrees). The

servo's shaft will turn to that position and stay there as long as the same signal keeps being fed to the servo. If the signal changes, the shaft will turn to the position specified by the new signal. If no signal is given, the servo won't move at all. The servo circuitry always knows the position of the shaft through a feedback potentiometer. The potentiometer is coupled to the servo's output gear, so that at different positions, the potentiometer will have different resistance. Another characteristic of the servo that is very useful is that it moves at different speeds according to the change of position desired. For instance, if the servo is at 0°, and it is given a signal to go to 180°, it will begin turning at a maximum speed, and as the destination point is approached, speed decreases until it reaches zero. With these principles of operation in mind, consider modifying the servo in the following manner: de-couple the feedback potentiometer from the output gear so that the gear can turn without moving the potentiometer's shaft, then turn the potentiometer shaft to its middle position, and remove any hardware that keeps the servo from turning 360° (in this case, a small stop on one of the gears - see Appendix for Figure). By leaving the potentiometer in its center position, the servo circuitry will always behave as though the servo were in its center position. Therefore, if a signal is sent to the servo telling it to move to its center position, the circuitry will think that the shaft already is at the center position, so the motor will not move. If a signal is given to the servo telling it to move to one of its extremes (0 or 180 degrees), the circuitry will start turning the motors toward that position, but since the potentiometer is no longer attached to any moving parts, it keeps telling the circuitry that the servo is in its center position, so the motor will keep on turning at full speed. Also, if a signal is given to the servo to move to a point close to its center position, it will turn very slowly toward that

position. In this manner, the servo motor can be used just like a DC motor, but without the need for motor drivers. The three-pin connectors from the servo motors were connected to ground, V+ (power supply voltage ~7.2V), and an output pin from the HC11. The servos used in these robots are rated at 5V for supply power, so supplying them with 7.2 volts created some problems. The most evident of these problems is an unstable region of operation when the motor is told to go at a certain speed. Within a speed range, and in only one direction, the motor vibrates pretty hard. I am not certain of the cause of this odd behavior, it may have something to do either with the motor's brushes, or the servo circuitry.

Software control of the servos was regulated by one of the HC11's output compare

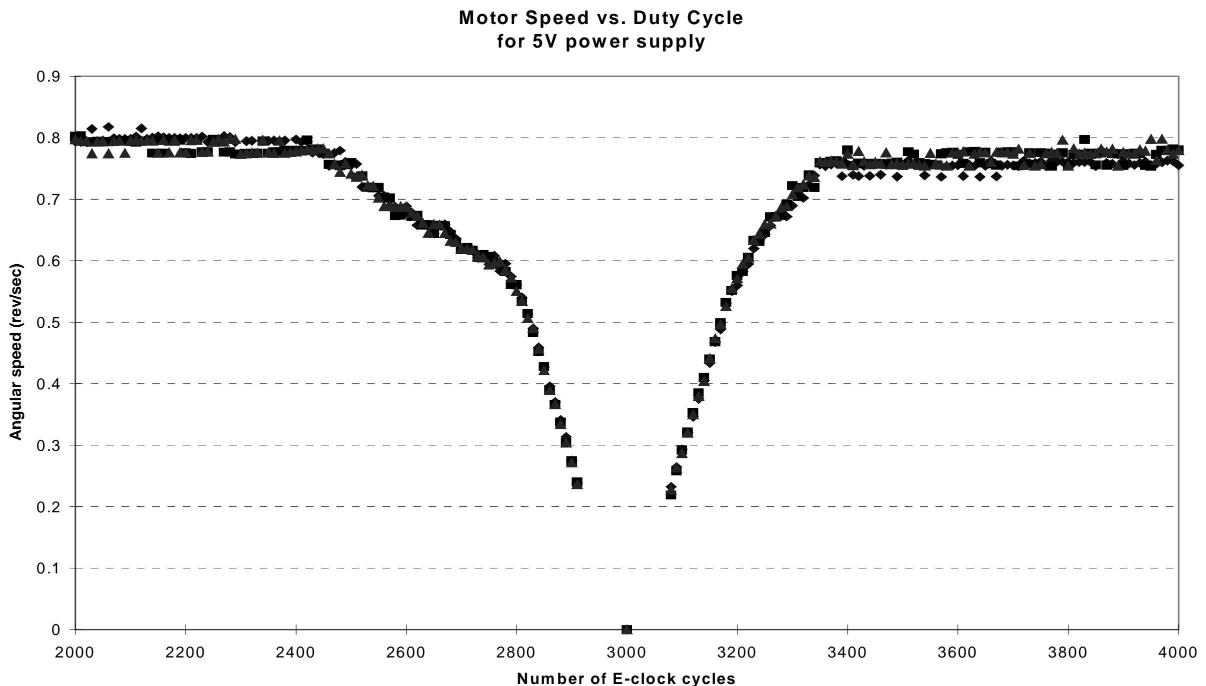


Figure 3: Motor Speed vs. Signal Duty Cycle (5V supply)

timers, and was interrupt-driven. The signal fed to each servo must be periodic, with a

frequency of 50Hz, and a duty cycle ranging from 5% to 10%. A single output compare produces a 100Hz signal which is in fact the two motor signals shifted by 180°. Two bits from one of the HC11's output ports (Port B) control the two motors. As Figure 3: Motor Speed vs. Signal Duty Cycle (5V supply) and Figure 4: Motor Speed vs. Signal Duty Cycle (7.2V supply) show, the speed of each motor depends on the duty cycle of the

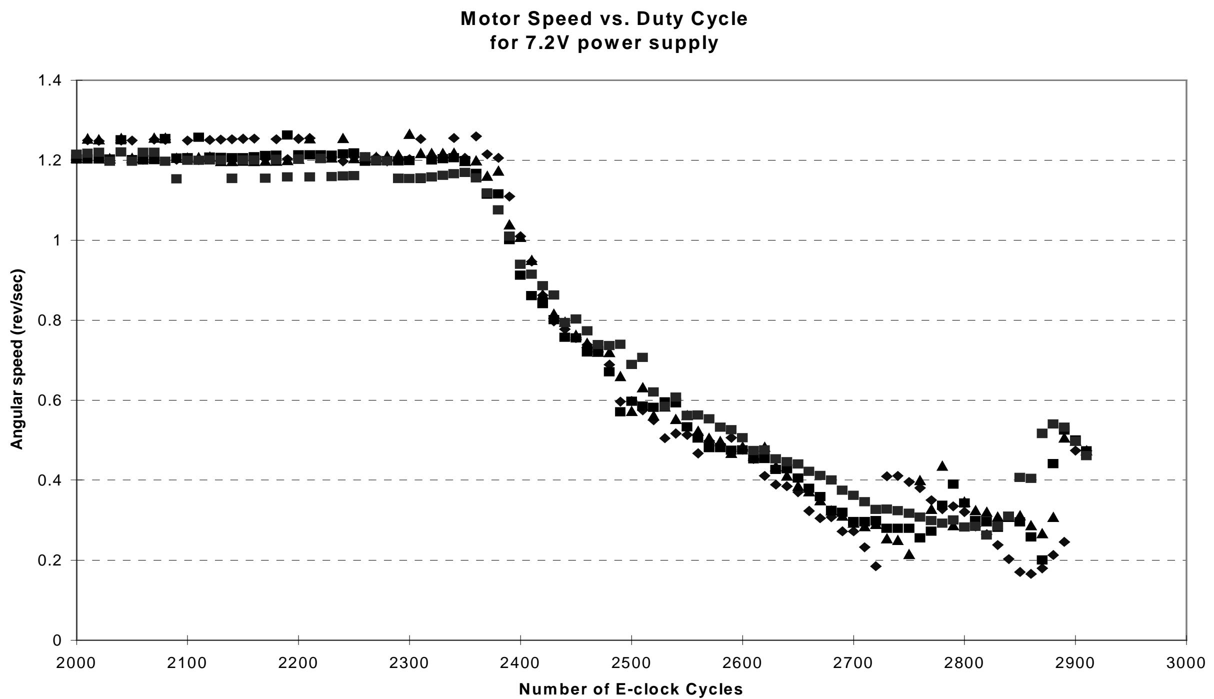


Figure 4: Motor Speed vs. Signal Duty Cycle (7.2V supply)

control signal sent to that motor. On these two figures, speeds caused by duty cycles between 5% (2000 clock cycles) and 7.5% (3000 clock cycles) are in the counterclockwise direction, and speeds corresponding to 7.5% - 10% duty cycles are clockwise. Figure 4 Figure 4: Motor Speed vs. Signal Duty Cycle (7.2V supply)shows that when the 7.2 power supply is driving the servos, their speed response is different..

While the speed curves in Figure 3 have a downward curvature, the curve in Figure 4 has an upward curvature, which means that a higher voltage supply slightly alters the motor's

speed response. Also, the higher voltage causes the motor to behave erratically at a certain range of speeds, in Figure 4 this range is between 2700 and 2900 clock cycles

Sensors

Infrared Emitters and Detectors:

IR-based sensors form the bulk of each TJ's sensory suite. Infrared emitters and detectors are used by several systems such as collision and object avoidance, recognition of other robots, "following" behavior, and speed regulation, all of which will be explained later. All infrared-light detectors on the robots respond to 40kHz modulated IR light, which results in the need for some type of current modulation. Usually, simple hardware such as a 555 timer or counters are very effective at modulating current through an infrared LED for this type of applications, but, in the spirit of cutting down on hardware, each TJ drives all LED's directly with the HC11 microprocessor. Such a task presents a very relevant timing issue: can the HC11 provide a 40kHz signal without disturbing any processes that may be running? A 40kHz square wave has a period of 25µs, but that period includes an "on" time (when the signal is high) and an "off" time (signal is low), which means that an action must occur every 12.5µs. It is impossible for the HC11 to produce interrupts that fast, which means that if the modulating signal is going to be produced by the processor, all other processes must be put on hold. If the HC11 is going to be producing the 40kHz signal, the processor must produce the signal in such a way that other processes are not kept from functioning properly. A solution to this problem is to produce the IR signal in short, regular bursts, leaving enough time for other processes to proceed normally.

Working Principles and implementation

One of the basic assumptions of this method is that producing a short burst of infrared light will allow the IR detector to provide reliable, repeatable data, just as well as a continuous beam of IR would. In order to check this assumption, the voltage vs. time response of one of the detectors being used is the first factor to be considered.

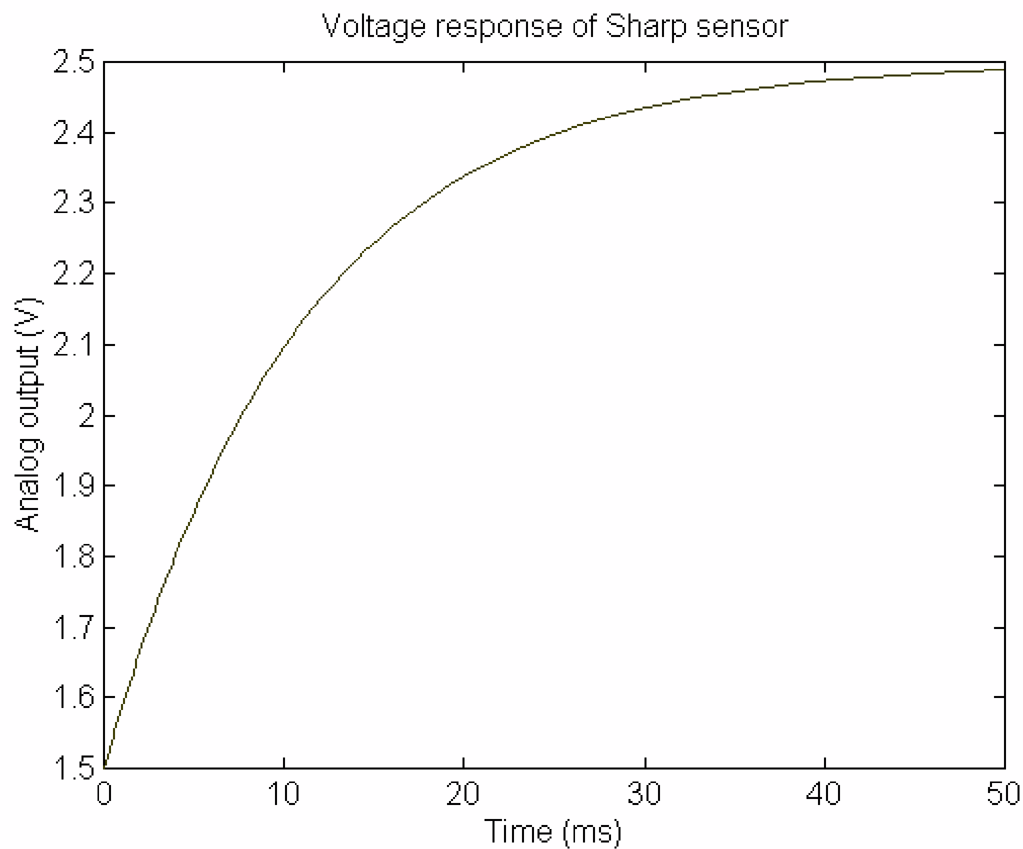


Figure 5 Voltage Response of Sharp IR Sensor

The type of sensor used for this research was a Sharp GP1U58, modified to produce an analog voltage output, rather than a digital output (see "Sensor Hack…"- Doty, de la Iglesia). Because of capacitive effects from the sensor's output stage, the voltage on the sensor's output pin follows a charge curve which is relatively slow. From the time the

sensor first detects IR light, it takes about 50 ms for its output to reach a peak value (In Figure 5 about 2.49 V). What this time means is that if only a short IR burst is to be fired, to achieve the sensor's full reading the duration of the burst must be at least 50 ms. If the burst were shorter than 50 ms, the voltage on the detector's output would only partially climb the curve in Figure 5, giving a value lower than the detector's peak value. However, assuming that the burst will always last the same amount of time, the voltage output of the sensor would in fact be repeatable and reliable, albeit not 'correct' for the amount of IR light being detected. A single burst of IR light will certainly not cause the detector to reach full voltage, however, periodic bursts will cause the detector to slowly climb the charge curve until it eventually reaches the peak value.

The software implementation of this method consists of letting an output compare timer produce the IR burst every 20ms, and immediately taking analog readings. These readings are stored in a global array, and this array is not modified until the next time the LED's are fired. Each time a program reads the global array, it will get the last valid detector value.

Each TJ has 3 infrared LED's mounted on its body: 2 facing forwards and one facing back. The function of the back LED is to make the robot visible to other robots which may be following behind. The LED's are wired so that they can be turned on all at the same time or only the back one be turned on. There are also 2 infrared detectors facing forward at a slightly outward angle and one facing back on each robot.

Bump sensors

Each robot is equipped with a bumper that lets the robot know when it has run into an object or when something has run into the robot. The circuitry for the bump sensor is extremely simple: when the robot is bumped, one of three switches will be closed. These switches are connected to an input port on the HC11 with a pull-down resistor, so that when one of the switches is closed, the input pin will go high. Implementing this system in software is as simple as polling the input bit, so that when it goes high, the robot will react accordingly.

Behaviors

Object avoidance

Object avoidance relies on the front IR LED's and detectors to detect obstacles in the robot's path. Object avoidance is the state in which the robot is most of the time. When there is an obstacle in front of the robot, the IR light will bounce off and be perceived by the detectors. The algorithm I used for object avoidance follows the Braitenburg method of each motor being controlled by a single sensor. To explain this in one sentence, each detector's output was normalized by the program to correspond to a motor speed. This can be accomplished very easily by finding the detector's full range of output values (in this case, 84 to 118), determining the range of motor speeds from full forward to full backward (2000 to 4000), and finding a linear relationship between the two. This means that when a detector gets a minimum reading (84), the robot assumes there was nothing in front, and the motor corresponding to that detector moves at full forward speed. When a detector gets a maximum reading (118), the corresponding motor turns at full reverse speed. All detector values in between these maximum and minimum are normalized to motor speeds between full reverse and full forward. One problem with this method is the

"Braitenburg Trap" when both detectors read a value corresponding to zero motor speed, and the robot does not move. To make up for this, the robot has a "boredom" response, so that if the both motors have stayed at a very low speed for a certain period of time, the robot will charge whatever is in front. The reason why the robot will charge, is because Braitenburg traps can be flat walls, a corner, or a very narrow corridor. If the trap presented to the robot is one of the first two, the robot would ram the wall or corner, and its bump sensors will tell it to turn around and leave. However, if the trap happened to be a very narrow pass, the robot will charge, and if it doesn't hit anything after a specified period of time, it will resume obstacle avoidance behavior.

Robot Following

Each TJ will periodically (almost twice a second) look for another TJ. To look for another TJ, the robot turns off its front IR emitters, waits for the infrared detectors to discharge (~100 ms), and then takes a reading from the detectors. If the detectors read anything besides a minimum, the light going into the detectors must be coming from another robot, so the TJ goes into robot following mode. In robot following mode, the two front LED's are off, only the back LED is on, and this is done for two reasons. First, if the front IR emitters were to go on, the robot could not differentiate between its own light and the one coming from the robot in front. Also, if all LED's were turned off while the TJ was following, no other TJ's could follow it, so the back LED must be left on. To determine which way to turn, the robot will take readings from both sensors and take one of the following courses of action: if one reading is higher than a certain threshold, and the other is below, the robot will turn toward the side receiving more IR light. If the readings are close and both above another, lower, threshold, the robot will go straight

ahead, and if both readings are both below this second threshold, the robot will assume it has lost the other TJ, give up following and return to object avoidance.

If the robot's bumper is activated during the 'following' behavior, it will assume it has hit the robot it was following, and will reduce the speed at which it was following. This behavior proved to be harder than anticipated, especially because the hacked IR detectors were not always matched, that is, they produced slightly different outputs. To solve this problem, I used the brute force method of testing detector pairs until I found a pair that was very close.

Speed Regulation

At regular time intervals, the robot will "look back" by reading its third IR detector to see if any other robots are following. Each TJ will only look back after it has traveled in a straight line for a given amount of time, that way it will know it is not looking at an obstacle it may have just avoided. If the TJ determines that another robot is, in fact, following, it will slightly reduce speed and allow the other robot to catch up to him. When the TJ is hit from behind by the other robot, it will resume normal speed. This behavior was added to keep the robots traveling in a tight pack.

Collision Detection

The collision detection behavior/system is connected to all other behaviors. During collision avoidance, when a front bump is detected, the robot will back up, turn for a random amount of time, then resume collision avoidance. During the 'following' behavior, if a front bump is detected, the robot knows it has caught up to the robot in

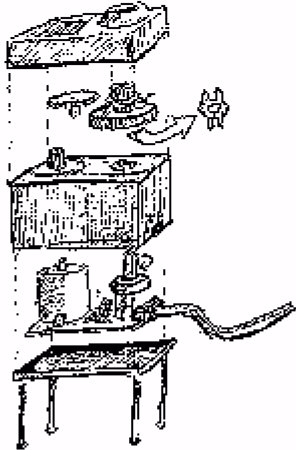
front. Finally when a robot has determined that another TJ is following it, it will slow down until it feels a bump from the back, then resume normal speed.

Conclusion

This project established the feasibility of building a simple robot which contains no circuitry besides a microcontroller chip, in this case, an M68HC11E2. All three robots successfully followed each other and traveled in a straight line, although this behavior proved to be more complex than expected. With a more sophisticated sensor suite, the group behavior could be greatly enhanced to include elements such as more advanced robot to robot communication, and the completion of complex group goals. The 'following' behavior is very effective, but could definitely be improved to provide a smoother response. After using the motor and infrared driver routines which I wrote, I am convinced that those could also be optimized either by translating them directly to assembly language, or by finding tighter algorithms that are as effective. Code size was an issue to keep in mind throughout this project, since the E2 microcontroller has only 2K of EEPROM and 256 bytes of RAM on board.

Appendix

Figures



The figure to the left shows how to modify servo motors so that they can be operated as regular DC motors.

Source Code

Motor driver routine: servotj.c

```
/* servotj.c

•	Written by Isaac Green for MIL (Dr. K. Doty, Director)
•	The servo's use OC4 and OC5.  This module was written for the
•	Thomas robot, with hopes of extending it for general use in the
•	furture.
•	Started : 13 OCT 1995 change OC5 to OC1 for Talrik robot

November 2, 1996:

Modified code to utilize a single OC interrupt.  OC2 will control the

timing for both servo motors, and PORTB will provide the I/O pins for

the PWM signals.  This code is intended for use with a Talrik Jr using

a single chip computer board (MSCC11).  Changes by Ivan Zapata

* /

/**************** Includes ***************************************/
#include <hc11.h>
#include <mil.h>

/**************** Constants **************************************/
#define PERIOD 40000
#define HALFPERIOD 20000


#pragma interrupt_handler servo_hand
void servo_hand();

unsigned width[2], current_width;
char signal_state;
char servomask[2];

/* current_width was added in case width[] is changed while the signal is

being produced, which produces abnormal (shaking) behavior in the servos

* /

void init_servos(void)
```

```
/* This routine initalizes the servos. */
{

INTR_OFF();

  CLEAR_BIT(TCTL1,0xC0);           /* Interrupt will not affect OC2 pin
*/

  width[0] = width[1] = 0;            /* Set PWM's to 0 first */
  TOC2 = 0;                        /* First OC2 int will occur at TCNT=0 */
    current_width = 0;
  signal_state = 0;                /* Initial state of signals */

  PORTB = PORTB & 0x3F;             /* Motors start turned off */
    servomask[0] = 0x40;
    servomask[1] = 0x80;

    SET_BIT(TMSK1,0x40);       /* Enable OC2 interrupt */

    INTR_ON();

}

void servo(int index, unsigned newwidth)
/* Sets a servo to a certain pulse width */
{
/*  if (index)

    width[1] = newwidth;

    else

    width[0] = newwidth;*/

    asm("ldab %index \n"

    "asrb \n"
    "ldy #_width \n"
    "aby \n"
    "ldd %newwidth \n"
    "std 0,Y");


}

void servo_hand ()


{
    char odd;
    int index;
    unsigned int pwidth;


/*
    signal_state = 0  -> Turn on servo0
    signal_state = 1  -> Turn off servo0
```

```
    signal_state = 2  -> Turn on servo1
    signal_state = 3  -> Turn off servo1
*/

    signal_state &= 0x03;        /* Only use last 2 bits */
    index = signal_state;
    asm("ldaa %index \n"
        "lsra \n"
        "staa %index");

    odd = signal_state & 0x01;

    pwidth = *(width+index);

    if ((pwidth == 0)&&(!(odd)))
   {
TOC2 += HALFPERIOD;

    signal_state++;
   }
    else
   {
    if (!(odd))
    TOC2 += pwidth;
    else if (odd)
    TOC2 += PERIOD - current_width;
    PORTB ^= *(servomask+index);
    current_width = pwidth;
   }

    signal_state++;

    /* Signal 0 will go on bit 6 of PortB, and Signal 1 will be bit 7 */

    CLEAR_FLAG(TFLG1,0x40);        /* Clear OC2I flag */
}
```

Infrared system driver: irtj.c

```
#include <hc11.h>
#include <mil.h>
#include <analog.h>

/* irmod2() modulates PORTB, bit 0 at 40kHz */

#define PERIOD 40000

#pragma interrupt_handler irread;
void irread();

int analog_value[8], current_mode, discharge_count;

/* void init_ir()

Will initialize the interrupt-driven ir readings for TJ.  OC3 will be used for timing control

* /

void init_ir()

{
    int i;
INTR_OFF();

    for (i = 0; i < 9; i++)
    analog_value[i] = 0;
    current_mode = 0;         /* Fire IR LED's before reading port */
    discharge_count = 0;
TOC3 = 10000;

    CLEAR_BIT(TCTL1, 0x30);  /* Interrupt will not affect OC3 pin */
    SET_BIT(TMSK1, 0x20);    /* Turn on OC3 interrupt */
    INTR_ON();
}

void discharge()

{
    discharge_count = 7;
```

```
    analog_value[7] = 0;
    analog_value[6] = 0;
}

void ir_mode(int mode)

{
    current_mode = mode;
}

int ir_value(int num)

{
    int i;
    i = analog_value[num];
    return i;
}

/* irmod2() modulates PORTB, bit 0 at 40kHz */
void irmod2()
{
  asm("ldaa    #1\n"
      "ldy    #255\n"          /* 2*(# of IR pulses) */
      "staa    4100\n");         /* 4100 = Port B */
  asm("loop : ldaa 4100\n"      /* 4 cycles - necessary */
      "eora #1\n"                /* 2 cycles - necessary */
      "staa 4100\n"            /* 4 cycles - necessary */
    "nop\n"
    "nop\n"
    "nop\n"          /* 8 cycles */
    "next : nop");
  asm("dey\n"                               /* 4 cycles - necessary */
      "bne loop");                          /* 3 cycles - necessary */
    /* total = 25 cycles = 40 kHz*/

    asm("ldaa 4100");          /* Clear Port B - Turn LEDs off */

    asm("anda #254");

    asm("staa 4100");

}

/* irmod3() modulates PORTB, bit 1 at 40kHz */
void irmod3()
{
  asm("ldaa    #3\n"
      "ldy    #255\n"          /* 2*(# of IR pulses) */
      "staa    4100\n");         /* 4100 = Port B */
  asm("loop : ldaa 4100\n"      /* 4 cycles - necessary */
      "eora #3\n"                /* 2 cycles - necessary */
      "staa 4100\n"            /* 4 cycles - necessary */
    "nop\n"
    "nop\n"
    "nop\n"          /* 8 cycles */
    "next : nop");
  asm("dey\n"                               /* 4 cycles - necessary */
```

```
    "bne loop");                    /* 3 cycles - necessary */
  /* total = 25 cycles = 40 kHz*/
  asm("ldaa 4100");          /* Clear Port B - Turn LEDs off */
  asm("anda #253");
  asm("staa 4100");
}

/*Calls the modulator and then reads the analog ports */

/* Care should be taken since it also kills interrupts

  for that period of time*/
/* irread() is the interrupt handler */

void irread()

{
  if ((!(current_mode))&&(!(discharge_count)))
  irmod2();
  else if ((current_mode)&&(!(discharge_count)))
  irmod3();
  analog_value[7] = analog(7);
  analog_value[6] = analog(6);

  if (discharge_count)
 {
  discharge_count = discharge_count - 1;
  analog_value[7] = 0;
  analog_value[6] = 0;
 }

TOC3 += PERIOD;

  CLEAR_FLAG(TFLG1, 0x20);
}
```

Main program: Chase9.c

```
/*
  10.28.96
  This program was written for a Talrik Junior platform in order to
  achieve collision avoidance. Ivan Zapata
 */

#include <servotj.h>
#include <hc11.h>
#include <mil.h>
#include <irtj.h>
#include <analog.h>
#include <vectors.h>


#define ZEROL  3000   /* Servo signals */
#define ZEROR  3000
#define FORWL  2250
#define FORWR  3500
#define BACKL  3500
#define BACKR  2400
#define RVOFF 6148
#define LVOFF -498
#define SENSORMIN 85
#define SENSORMAX 118
#define LEFT_SERVO 0
#define RIGHT_SERVO 1
#define SELF  0
#define OTHER 1
#define ATTENTIONSPAN 5
#define BACKTH1 100
#define BACKTH2 88



void turn(void);
void follow(void);
void catch_up(void);

char backpeak, followed, bored, lastback;
int vdec, straight, rv, lv;

int main(void)
{
  int rval, lval, attentionflag;
  int trapped;

  int i, j;
  char c;


  init_servos();
```

```
 init_analog();
 init_ir();
 ir_mode(SELF);

 DDRC = 0x02;          /* PORTC is input except for bit 2*/

 backpeak = 0;
 followed = 0;
 lastback = 0;
 bored = 0;
 vdec = 0;
 straight = 0;
 trapped = 0;

 attentionflag = ATTENTIONSPAN;

 while(1)
 {

/* servo0 = left, servo1 = right */
/* PE6 is on the right, PE7 is on the left */



   rval = ir_value(6);           /* Get sensor readings */
   lval = ir_value(7);

   if (rval < SENSORMIN) rval = SENSORMIN;
   if (lval < SENSORMIN) lval = SENSORMIN;

   rv = -(rval << 5) + RVOFF;    /* Calculate indexes */
   lv = (lval << 5) + LVOFF;


   catch_up();


   servo(RIGHT_SERVO, rv);          /* Set actual servo speed */
   servo(LEFT_SERVO, lv);

   attentionflag--;

   if(PORTC & 0x20)
   {
     servo(LEFT_SERVO, BACKL);
     servo(RIGHT_SERVO, BACKR);
     for(i = 0; i < 25000; i++);
     turn();
     ir_mode(SELF);
   }

   if (!(attentionflag))
   {
     discharge();
     ir_mode(OTHER);
```

```
      do
      {
        rval = ir_value(6);
      }
      while (rval == 0);
      for(i = 0; i < 6000; i++);
      rval = ir_value(6);
      lval = ir_value(7);

      if ((rval > 98)||(lval > 98))
      {
       follow();
      }


      ir_mode(SELF);
      attentionflag = ATTENTIONSPAN;
  }

  if((rv > 2900) && (rv < 3100) && (lv > 2900) && (lv < 3100))
    trapped++;

  if (trapped > 255)
  {
    trapped = 0;
    ir_mode(OTHER);
  }


  for(i = 1; i < 2500; i++);




  }
}

void catch_up()
{
  int i, j;

    i = ir_value(2);
    if ((i > BACKTH1)&&(!(followed)))
    {
      bored = 0;
      j = 0;
      backpeak = 0;
      while(!(followed)&&(!(bored)))
      {
        servo(RIGHT_SERVO, rv);
        servo(LEFT_SERVO, lv);
        rv--;
        lv++;
        for (i = 0; i < 50; i++);
        j++;
```

```
        if (j > 3900) bored = 1;

        if (rv < (ZEROR + 100)) rv = ZEROR+100;
        if (lv > (ZEROL - 100)) lv = ZEROL-100;
        if (PORTC & 0x01) followed = 1;
        lastback = i;

      }
      if (bored) followed = 0;

    }

    i = ir_value(2);
    if (i != lastback)
      lastback -= i;
    if ((i > backpeak)&&(followed))
    {
      backpeak = i;
    }
    else if((lastback > 0)&&(followed))   /* decreasing */
    {
      vdec += 100;
      rv -= vdec;    /* decrease speed */
      lv += vdec;
      if (rv < (ZEROR + 100)) rv = ZEROR + 100;
      if (lv > (ZEROL - 100)) lv = ZEROL - 100;

    }
    else if ((lastback < 0)&&(followed))
    {
      vdec = 0;
    }
    if (i < BACKTH2)
    {
      vdec = 0;
      followed = 0;
    }

    lastback = i;


}


void follow(void)
{
  int r, l, i, j;
 /* int ract, lact, rlast, llast;*/
  SET_BIT(PORTC, 0x02);
  l = ir_value(6);
  r = ir_value(7);

  rv = FORWR;    /* Calculate indexes */
/*  rlast = FORWR;*/
  lv = FORWL;
```

```
/*  llast = FORWL; */


  while ((r > 90) || (l > 90))
  {
    if ((l > 110)&&(r < 105))
    {
/*      rv -= (r-l) <<  2;*/
      lv = 3500 - (r >> 5) + 2050;
      rv = FORWR;
    }
    else if ((r > 110)&&(l < 105))
    {
/*      lv += (r-l) << 2 ;*/
      rv = l >> 5;
      lv = FORWL;
    }
    else
   /* if ((r > 110)&&(l > 110))
      if ((r < 110)&&(l < 110))*/
      {
        rv = FORWR;
        lv = FORWL;
      }

    if (rv < ZEROR) rv = ZEROR;
    if (lv > ZEROL) lv = ZEROL;

 /*   ract = rlast + ((rv - rlast) >> 4);
    lact = llast + ((lv - llast) >> 4);*/

    servo(RIGHT_SERVO, rv);
    servo(LEFT_SERVO, lv);


/*    rlast = ract;
    llast = lact;*/



    l = ir_value(6);
    r = ir_value(7);
  }
  CLEAR_BIT(PORTC, 0x02);
}


void turn()
{
 int i;
 unsigned rand;

 rand = TCNT;

 if (rand & 0x0001)
```

```
 {
  servo(RIGHT_SERVO, FORWR);
  servo(LEFT_SERVO, BACKL);
 }
 else
 {
  servo(RIGHT_SERVO, BACKR);
  servo(LEFT_SERVO, FORWL);
 }
 for (i = 0; i < rand; i++);
}
```