University of Florida

Department of Electrical and Computer Engineering

EEL 5666: Intelligent Machines Design Laboratory

Fall 1997

Scout Robot Final Report

By: Mark Atchison

December, 12 1997

mga@cise.ufl.edu

(352) 377-7790

ABSTRACT

Scout is an exploration robot. The primary behavior uses the on-board sonar ranging system to seek out targets that need investigation. The robot controls the servomotor-mounted sonar to range targets without moving its body. Once reaching a target the robot uses a wall-following algorithm to trace a path around the object before seeking another target. The sonar system allows the robot to accurately range distances from 35 feet away giving the searching algorithm excellent range. To carry the robot between obstacles, the powerful DC motors taken from kit racing cars power the Scout Robot's treads through a series of three reduction gears. MOSFET high-current motor drivers complete the basic package to provide a robust platform for an exploration robot.

EXECUTIVE SUMMARY

The heavy construction of Scout provides an excellent base for an exploration robot with room to add additional sensors for improved performance and perception. Due to the heavier construction, Scout uses larger DC motors and motor drivers. The primary sensor used by Scout is the sonar. Scout aims this servo-mounted sensor to measure the distance accurately to the first obstacle. Scout then heads toward the detected obstacle to explore the region. While exploring, the infrared obstacle avoidance is on alert to prevent Scout from colliding with obstacles, people, and other robots. Scout is also equipped with mechanical bump sensors to back up the infrared sensors. The final sensor on-board Scout is the wheel encoders. The front wheels drive these small disks. The robot can use the sonar information with the wheel encoders to read a distance and then drive a distance.

INTRODUCTION

The scout robot is an autonomous search vehicle. The body of the robot is a tracked body taken from a remote controlled bulldozer with an expanded platform to hold the electronics and sensors. The robot's controlling electronics is a 68HC11 evaluation board with the ME11 robot controller expansion kit. The sensor suite allows the robot to seek out targets with its sonar while avoiding obstacles. Scout is able to exhibit several behaviors including searching and obstacle avoidance.

INTEGRATED SYSTEM

The scout robot functions as a searcher. Introduced into a new environment it adapts its sensors to the environment and then seeks out the nearest obstacle to investigate using the sonar. The robot will head toward the obstacle, pausing occasionally to check the target again with the sonar and search for better opportunities. Upon reaching the target, the robot skirts the obstacle using a wall-following algorithm to investigate the perimeter before finding another target. These behaviors provide the groundwork for the final design of a scouting and mapping robot.

MOBILE PLATFORM

The body structure is a tracked vehicle driven by two motors. Figure 1 shows the basic layout of the robot, without the upper sensor

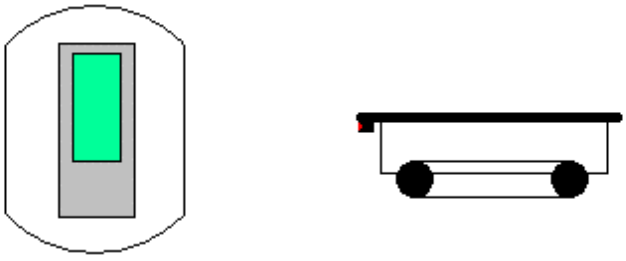

Figure 1: Scout Basic Structure.

suite. The tracks give the robot maneuverability similar to robots with two wheels, but offers additional stability. Additionally, the tracked design gives the robot the ability to traverse rough terrain while seeking its target. The upper platform, rounded at the front and rear to eliminate corners, integrates with the lower aluminum body that holds the motors and reduction gears. The upper platform gives the robot maneuverability similar to a round bodied robot, but with a more vehicular appearance. The electronics ride on top of the main platform under a protective cover – well away from ground contaminates. The original axle, one of the few surviving parts from the original toy, cracked during testing and needed reinforcement. Many other components from the original toy needed replacement, beginning with the motors.

ACTUATION

Primary Motors

Two large DC (model unknown) motors of the type used in RC racing cars move the robot. These large motors replace the original toy motors that burnt out shortly after the first demo. The large hardware components are required due to the weight of the robot. The motors supply power to the treads through

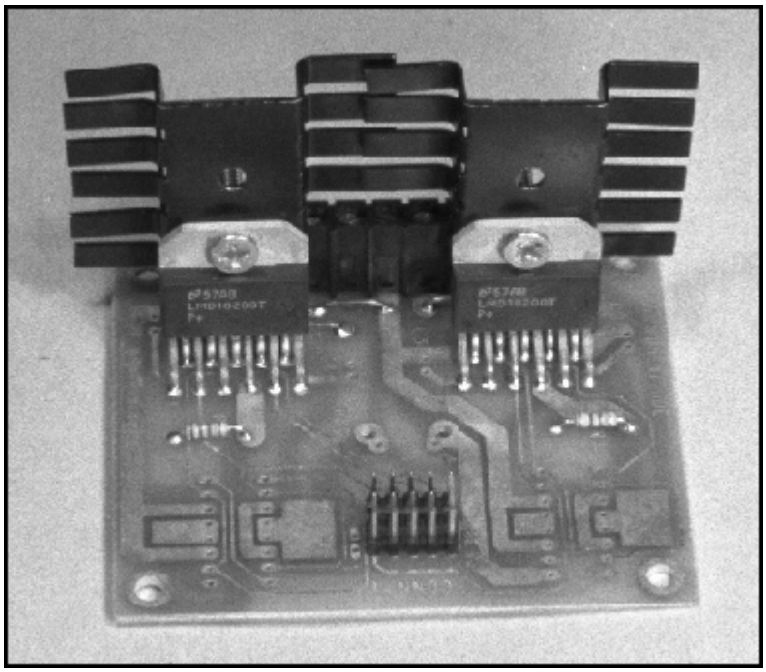

Figure 2: MOSFET Motor Driver.

a series of reduction gears taken from the same vehicle as the motors.

Driver

Figure 2 shows the motor driver used in Scout. Wirz Electronics, 6100 Pershing, #2A St. Louis, MO 63112, Office: (314) 862-3370, supplied the kit and the supporting FAQ. The MOSFET chips on this board are connected in parallel to handle currents up to 6 amps continuous and 10 amps peak. The motor speed and direction is controlled using the same pulse-width management routines commonly available for the 68HC11. This massive motor drive is needed as the original relay-based motor driver design could not withstand the needed power.

Servo Motor

A single servomotor aims the sonar. Using the servo with the sonar, Scout can seek obstacles that require investigation without turning. The Servo is controlled using standard ICC11 library routines through OC3.

SENSORS

Infrared Sensors and Light-Emitting Diodes

The IR receivers used in Scout are the Sharp GP1U58Y 40KHz IR Receivers provided by Novasoft. Figure 3 shows one of the Sharp IR Receivers beside a penny for scale. The address of

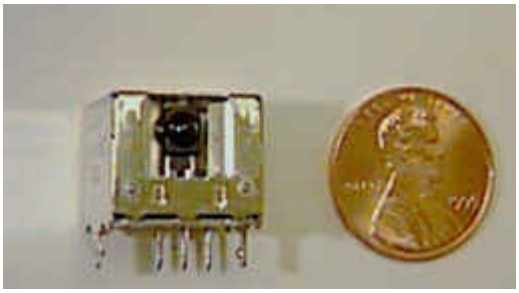

Figure 3: Sharp IR Receiver.

Novasoft is 8822 Trevillian Road, Richmond, VA 23235 and the telephone number is Toll Free U.S. 1-888-MR-ROBOT. These IR receivers, when modified using the Sharp IR hack detailed on the Novasoft Web Site at http://www.mil.ufl.edu/novasoft/, connect to the A/D port of the 68HC11. The analog data provides an indication of the amount of reflected 40KHz light received.

The other part of this sensor is the 40KHz modulated IR emitters mounted in the bumper of Scout. The light the emitted is reflected back when an obstacle is within a foot to a foot and a half depending on ambient conditions. The IR receivers indicate then indicate greater light input by increasing the voltage to the A/D. The robot is then able to maneuver around obstacles.

Bump Sensors

The bump sensors are simple normally on momentary pushbutton switches mounted behind the spring-loaded bumpers of the scout robot. The interface consists of an input buffer memory-mapped at $7000. As pressing the switch disconnects the input lines, the inputs each use a pull-down resistor to ensure a true low reading.

Wheel Encoders

The Wheel Encoder modules shown in Figure 4 connect to the front wheels of the Scout Robot' – one module per wheel. They output a pulse each time the notched disk inside breaks the optical beam. By counting the pulses – one with the pulse accumulation and the other

Figure 4: Wheel Encoder Modules.

with an interrupt routing – the robot calculates the approximate speed and distance traveled.

The modules used are Hewlett-Packard Encoders model number HEDS-5500#I06. Designed to read rotational speed, Hewlett-Packard uses these single piece modules in their own line of printers. The units used in Scout came from Allied Electronics, Inc, 7410 Pebble Dr. Ft. Worth, TX, 76118, Phone (800) 433-5700.

Sonar Ranging

The sonar, shown in Figure 5, gives the scout robot the ability to detect, avoid, and map obstacles in the environment.

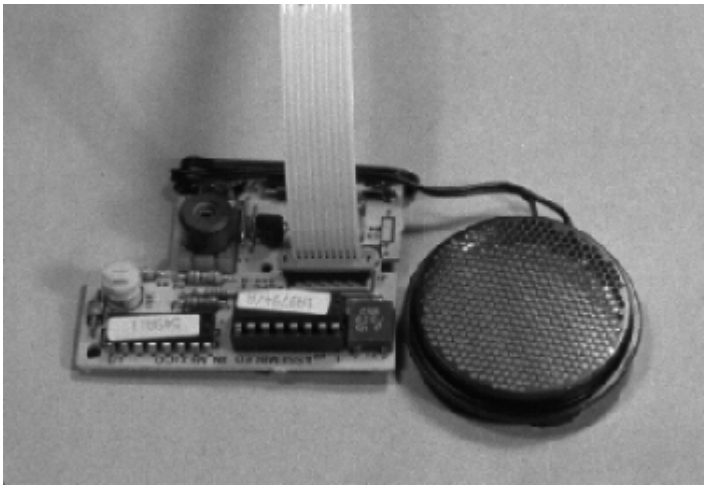

Figure 5: Polaroid Sonar Kit.

The sonar module used is a Polaroid 3500 TTL compatible sonar kit connected to the 68HC11 through memory-mapped control lines and the input capture facility of the 68HC11. The kit consists of a 600 Series Polaroid Transducer and a 6500 Series Sonar Ranging Module. Wirz Electronics, 6100 Pershing, #2A St. Louis, MO 63112, Office: (314) 862-3370, supplied the kit and the supporting FAQ. The complete FAQ can be found on the Wirz Electronics World-Wide-Web site http://www.wirz.com.

Interface

The ranging module drives the electrostatic transducer at 50 kHz from a single 5-volt power supply. Supporting this rate requires high speed switching that creates line noise. As the sonar is a noisy load, drawing current in bursts to emit the pulse, a separate voltage regulator directly from the battery pack must power it. Additionally, the power supply for the sonar is isolated by 100µF and 0.1µF capacitors.

The robot uses two of the inputs of the ranging module: BLNK and INIT. Figure 6 shows a sample-timing diagram for the sonar. By asserting INIT, the robot causes the sonar to emit a pulse. The BLNK input allows the robot to end the eternal blanking period – the period the sonar waits until vibrations from the pulse end - early to detect objects closer than the normal 12-inch limit. When the sonar detects a pulse, it asserts the ECHO line. Input Capture Three detects the rising edge, stops the timer and runs the interrupt routine. The interrupt routine calculates the time difference from emit to receive, places the value into a global variable, and signals availability of data for use by behavior modules by setting a flag.

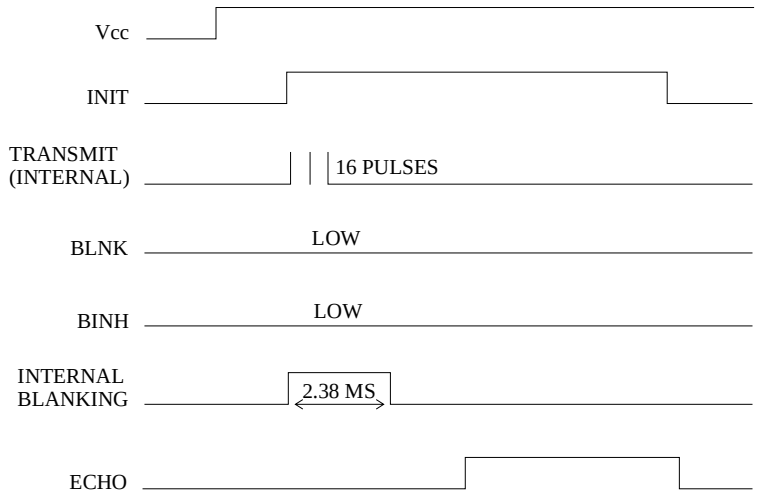

Figure 6: Ranging Module Control Timing without early blanking.

Performance

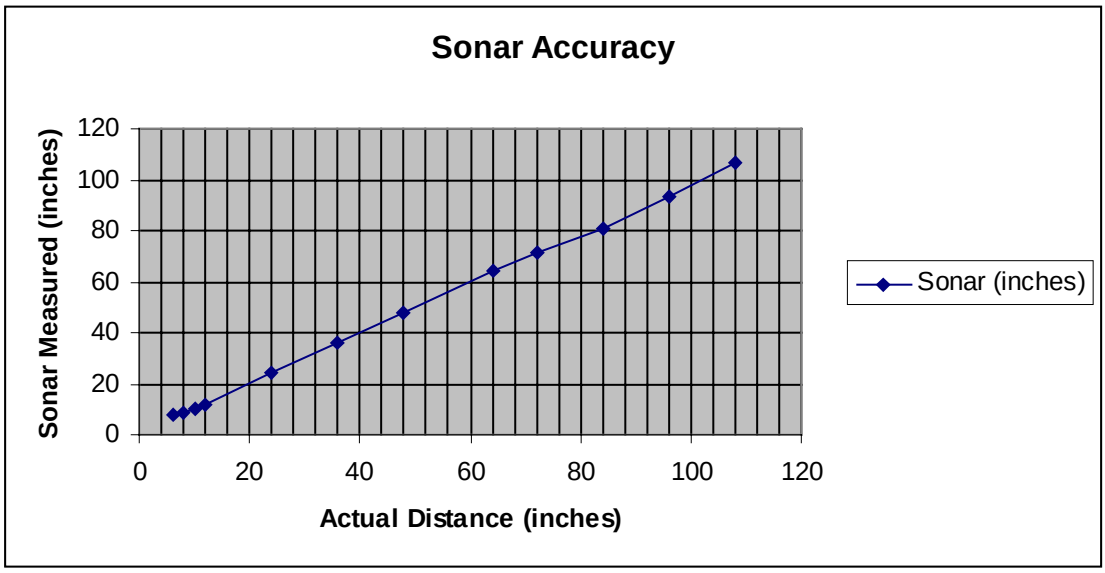

Figure 7: Sonar Accuracy

This module is able to detect obstacles at distances from 6 inches to 35 feet. As Figure 7 shows, the sonar produces consistent accuracy between 10 and 64 inches (between 0.83 and 5.33 feet). Below 10 inches, the sonar reports a distance greater than the actual value. The 6-inch measured distance represents the smallest value the sonar can read in this configuration. The 6-inch distance also corresponds to the distance from the wall to the transducer when the front bumper of the robot strikes the wall. Above 64 inches, the sonar returns values that are smaller than the actual distance. In addition, as the distance increases small variations begin to appear between readings from the same distance.

BEHAVIORS

<u>Obstacle Avoidance</u>

Using the IR system, the robot is always vigilante for obstacles in its path. The robot will turn to avoid obstacles and escape traps. The three forward most Sharp IR receivers provide this function, the remaining two IR receivers function primarily in wall following.

Wall Following

Using the side mounted IR receivers, the robot attempts to follow the wall by maintaining the reading from the IR receivers within a small range. This range detects the wall, but is far enough away to prevent collision. Wall following primarily occurs when reaching a target acquired by the sonar.

Sonar Ranging and Wander

The robot uses the sonar to seek out an object nearby, but at least three feet away to investigate. If the target is too far away, the robot pauses and rechecks the target and attempts to acquire a better target. This function is the first of the scouting routines that give Scout its name.

Sensor Calibration

The sensor calibration routine samples the environment to set the IR levels to the environment. First, Scout engages the bump sensor to get a fixed reference point. The robot then uses its wheel encoders to measure a distance from a fixed object and updates the constants to give a uniform indication from the IR receivers.

CONCLUSION

The sonar provides the robot long-range obstacle detection and avoidance ability. The enhanced knowledge of its environment allows the robot to function as a scout. The strong body and – finally – solid hardware design provide an excellent development platform.

APPENDIX A – SOURCE CODE

```
/*********************************************
 * Mark G Atchison                           *
 * Scout Robot combined control software file   *
 * Created: 12/11/1997                       *
 *********************************************/

/*********************************************
 * Includes **********************************
 *********************************************/

#include <hc11.h>  /* Standard Register Names */
#include <mil.h>   /* MIL standard functions */

/*********************************************
 * Defines ***********************************
 *********************************************/

#define BHN_COLLISION 0       /* Collision Behavior Number */
#define NUMBER_OF_IR 5          /* Number IR Installed */
#define LEFT_CONSTANT 31        /* Left Constant */
#define RIGHT_CONSTANT 31       /* Right Constant */
#define MIN_SPEED_LEFT -50     /* Left motor minimum */
#define MAX_SPEED_LEFT 50      /* Left motor maximum */
#define MIN_SPEED_RIGHT -50    /* Right motor minimum */
#define MAX_SPEED_RIGHT 50     /* Right motor maximum */
#define LEFT_DIV 5             /* Divide by 32 */
#define RIGHT_DIV 5             /* Divide by 32 */
#define RIGHT_ID 0              /* Right motor ID */
#define LEFT_ID 1               /* Left motor ID */
#define MIN_MOVE_RIGHT 20       /* Min PW needed to move */
#define MIN_MOVE_LEFT 20        /* robot.                */
#define CENTER 3000            /* center the servo      */
#define LEFT 4000               /* left positon          */
#define RIGHT 2000              /* right position        */
#define OFF 0                   /* servo off positon     */
#define SONAR_OUTPUT ADDR4       /* sonar at $4000       */
#define SONAR_BINH 0x01          /* sonar BINH is bit 1 */
#define SONAR_INIT 0x02          /* sonar INIT is bit 2 */
#define SONAR_INPUT PORTA        /* sonar inputs into A */
#define SONAR_ECHO 0x01         /* Echo is A0 line */
#define ON_PERIOD 25            /* rest for 80ms */
#define OFF_PERIOD 10           /* listen for 30ms */
#define CAPTURE_MASK 0xFE       /* TFLG1 Mask */
#define CAPTURE_BIT 1           /* TFLG1 and TCTL2 bit */
#define CAPTURE_REG TIC3        /* Input capture 3 */
#define NUM_BUMP_SENSOR 6       /* Number of bump sensors */
#define FRONT_BMP 0x01  /* bump sensor masks */
#define FRONT_RBMP 0x02
#define FRONT_LBMP 0x1F
```

```
#define BACK_BMP   0x08
#define BACK_LBMP  0x04
#define BACK_RBMP  0x20
#define MAX_CHANGE 10
#define NUMBER_BEHAVIORS 1  /* Number behaviors */
#define NUMBER_PARAMS 5
#define MOT_L 0             /* PARAM ENTRY for Left mot */
#define MOT_R 1               /* PARAM for Right */
#define PRIORITY 2            /* PARAM for PRIORITY */
#define STRENGTH 3            /* PARAM for STRENGTH */
#define TURNS 4               /* PARAM for Turns */
#define IR_THRESHOLD 8        /* IR min value for detect */
#define IR_PANIC 15           /* IR stop everything detect */
#define IR_CHANGE 15          /* Amount IR routine changes */


/*********************************************
 * Global Variables ***************************
 *********************************************/

int act_left;                   /* Current PWM */
int act_right;                  /* Motor Speed */
int ir_values[NUMBER_OF_IR];    /* Results of last IR */
int init_values[NUMBER_OF_IR];  /* Result of first sweep */
unsigned servo_width;           /* Pulse Width for servo */
int sonar_time;                 /* sonar capture time */
int sonar_start_time;           /* sonar start-up time */
int sonar_data_avail;           /* sonar data ready flag */
int bump_sensor_set[NUM_BUMP_SENSOR]; /* Bump sensor */
int times_set[NUM_BUMP_SENSOR];       /* track how long set */

int behaviors[NUMBER_BEHAVIORS * NUMBER_PARAMS];


/*********************************************
 * Wait ***************************************
 *********************************************/

void wait(int length)
{

        int temp;
      temp = length;

        while (temp--)
      {
            /* Waste Time */
      }
}

/*********************************************
 * Device Drivers *****************************
 *********************************************/
```

```
/* Sonar On ***********************************/

void sonar_pulse()
{

      SONAR_OUTPUT = 0;      /* reset sonar */
      SONAR_OUTPUT = 0;
      SONAR_OUTPUT = 0;
      sonar_data_avail = 0;           /* clear data flag */
      sonar_time = 0;                 /* start cycle */
      SET_BIT(TCTL2,CAPTURE_BIT);     /* enable capture */
      sonar_start_time = TCNT;        /* fetch start time */
      SONAR_OUTPUT = SONAR_INIT;      /* Active sonar */

}

/* Interrupt Code *****************************/


#define TIC3_ISR sonar_isr
#pragma interrupt_handler sonar_isr
void sonar_isr()
{

      if (0xFE & TFLG1) { /* if bit set */
            CLEAR_BIT(TFLG1,CAPTURE_MASK); /* clear flag */
            sonar_time = CAPTURE_REG - sonar_start_time;
            sonar_time << 1; /* eliminate sign */
            SONAR_OUTPUT = 0; /* output low */
            CLEAR_BIT(TCTL2, CAPTURE_BIT); /* disable int */
            sonar_data_avail = 1;

      }
}

/* Analog Init ********************************/

void init_analog(void)
{
  SET_BIT(OPTION,0x80);                  /* Power up A/D */
}

/* Analog Read ********************************/

int analog(int port)
/* Takes one reading from the analog port and returns the value read
 * Inputs : Port
 * Output : None
 * Return Value : Value read from A/D port
 */
{
  ADCTL=port;
```

```
  while(!(ADCTL & 0x80));
  return(ADR1);
}

/* Motor Routines ********************************/

/* Title        motor.c
 * Programmer   Keith L. Doty.
 * Date         June 19, 1996
 * Version      1
 * Description
 *      This module includes motor initialization, motor speed control
 *      and two PWM interrupt drivers motor0 and motor1.
 *      motor0 uses OC2 and motor1 uses OC3.
 *      This module can be used with the TALRIK robot
 *
 */

/**************** Constants ************************************/
#define PERIODM 65,500
#define PERIOD_1PC 655

#define TOC2_ISR motor0
#define TOC3_ISR motor1

#pragma interrupt_handler motor0 motor1
void motor0();
void motor1();

/********************* Data ************************************/
int duty_cycle[2];  /* Specifies the PWM duty cycle for two motors */


/************** Functions ************************************/
void init_motors(void)
/* Function: This routine initializes the motors
 * Inputs:   None
 * Outputs:  None
 * Notes:    This routine MUST be called to enable motor operation!
 */

{
  INTR_OFF();

/* Set OC2 and OC3 to output low */
  SET_BIT(TCTL1,0xA0);
  CLEAR_BIT(TCTL1,0x50);

/* Set PWM duty cycle to 0 first */
   duty_cycle[0] = duty_cycle[1] =0;

/* Enable motor interrupts on OC2 and OC3 */
  SET_BIT(TMSK1,0x60);
```

```
/* Specify PD4 and PD5 as output pins.
 * PD4 controls direction of Motor 1 and PD5 the direction of Motor 0.
 */

  SET_BIT(DDRD,0x30);
  INTR_ON();
}

void motor(int index, int per_cent_duty_cycle)
/* Function: Sets duty cycle and direction of motor specified by index
 * Inputs:   index in [0,1]
 *           -100% <= per_cent_duty_cycle <= 100%
 *           A negative % reverses the motor direction
 * Outputs:  duty_cycle[index]
 *           0 <= duty_cycle[index]<= PERIOD (Typically, PERIOD = 65,500)
 * Notes:    Checks for proper input bounds
 */
{
  if (per_cent_duty_cycle < 0)
     {
      per_cent_duty_cycle = -per_cent_duty_cycle; /* Make positive */
     /* Set negative direction of motors */
      if (index == 0) CLEAR_BIT(PORTD,0x20);
      if (index == 1) CLEAR_BIT(PORTD,0x10);
     }
  else
     {
     /* Set positive direction of motors */
      if (index == 0) SET_BIT(PORTD,0x20);
      if (index == 1) SET_BIT(PORTD,0x10);

     }

/* At this point per_cent_duty_cycle must be a positive number less
 * than 100. If not make it so.
 */
  if (per_cent_duty_cycle > 100) per_cent_duty_cycle = 100;
  duty_cycle[index] = per_cent_duty_cycle*PERIOD_1PC;

}

void motor0 ()
/* Function: This interrupt routine controls the PWM to motor0 using OC2
 * Inputs:   duty_cycle[0] (global)
 * Outputs:  Side effects on TCTL1, TOC2, TFLG1.
 * Notes:    init_motors() assumed to have executed
 */

{
/* Keep the motor off if no duty cycle specified.*/

  if(duty_cycle[0] == 0)
```

```
   {
    CLEAR_BIT(TCTL1, 0x40);
   }
  else
    if(TCTL1 & 0x40)
     {
      TOC2 += duty_cycle[0];                  /* Keep up for width */
      CLEAR_BIT(TCTL1,0x40);                  /*  Set to turn off */
     }
    else
     {
       TOC2 += (PERIODM - duty_cycle[0]);
       SET_BIT(TCTL1,0x40);                 /* Set to raise signal */
     }
  CLEAR_FLAG(TFLG1,0x40);                         /* Clear OC2F interrupt
Flag */
}

void motor1()
/* Function: This interrupt routine controls the PWM to motor1 using OC3
 * Inputs:   duty_cycle[1] (global)
 * Outputs:  Side effects on TCTL1, TOC2, TFLG1.
 * Notes:    init_motors() assumed to have executed
 */

{
 /* Keep the motor off if no duty cycle specified.*/

  if(duty_cycle[1] == 0)
   {
    CLEAR_BIT(TCTL1, 0x10);
   }
  else
    if(TCTL1 & 0x10)
     {
      TOC3 += duty_cycle[1];                /* Keep up for width */
      CLEAR_BIT(TCTL1,0x10);                /*  Set to turn off */
     }
    else
     {
      TOC3 += (PERIODM - duty_cycle[1]);
      SET_BIT(TCTL1,0x10);                  /* Set to raise signal */
     }
  CLEAR_FLAG(TFLG1,0x20);                         /* Clear OC3F interrupt
Flag */
}

/* Servo Routines ******************************/

/* servo.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
** Sept. 12 1996 change OC1 back to OC5 for SCCB
**
** ----------------------------------------------------------------
**
** Modified: 07 November 1997
**
** Modified for scout robot by Mark Atchison.  The scout robot needs
** only one servo to operate the sonar turrent. servo1_hand() removed.
**
**/
#include <mil.h>
#include <hc11.h>
unsigned width;

#pragma interrupt_handler servo_hand
#define PERIOD 40000
#define TOC4_ISR servo_hand

void init_servo(void)
/* This routine initalizes the servos. */
{
    INTR_OFF();
   /* SET_BIT(OC1M,0x80); */
    SET_BIT(TCTL1,0x08);        /* Set OC4 to lower line */
    CLEAR_BIT(TCTL1,0x04);

    width = 0;                  /* Set PWM's to 0 first */
    SET_BIT(TMSK1,0x10);
    INTR_ON();
}

void servo(unsigned newwidth)
/* Sets a servo to a certain pulse width */
{
  width = newwidth;
}

void servo_hand ()
{
    if(width == 0)
    {
      CLEAR_BIT(TCTL1, 0x04);
    } else if(TCTL1 & 0x04) {
      TOC4 += width;          /* Keep up for width */
      CLEAR_BIT(TCTL1,0x04);  /*  Set to turn off */
    } else {
      TOC4 += (PERIOD - width);
      SET_BIT(TCTL1,0x04);          /*   Set to raise signal */
    }
    CLEAR_FLAG(TFLG1,0x10);           /* Turn off OC4 interrupt */
}
```

```
/*********************************************
 * IR Read ***********************************
 *********************************************/

void init_ir()
{
      int port;

      for (port=0; port < NUMBER_OF_IR; port++)
            init_values[port] = analog(port);
}

void read_ir()
{
      int port;

      /* Read all IR Ports */

      for (port = 0; port < NUMBER_OF_IR; port++) {
            ir_values[port] = (analog(port) - init_values[port]);
      }
}

/*********************************************
 * Update Bump *******************************
 *********************************************/

void read_bump()
{
      if (FRONT_BMP & ADDR4)
            bump_sensor_set[1] = 1;
      else
            bump_sensor_set[1] = 0;

      if (FRONT_LBMP & ADDR4)
            bump_sensor_set[0] = 1;
      else
            bump_sensor_set[0] = 0;

      if (FRONT_RBMP & ADDR4)
            bump_sensor_set[2] = 1;
      else
            bump_sensor_set[2] = 0;

      if (BACK_BMP & ADDR4)
            bump_sensor_set[4] = 1;
      else
            bump_sensor_set[4] = 0;

        if (BACK_RBMP & ADDR4)
            bump_sensor_set[5] = 1;
```

```
      else
            bump_sensor_set[5] = 0;

        if (BACK_LBMP & ADDR4)
            bump_sensor_set[3] = 1;
      else
            bump_sensor_set[3] = 0;
}


/*******************************************
 * Update Motors ******************************
 *******************************************/

void update_motors(int request_left, int request_right)
{

      /* temporary variable used in calculations */

      int post_value_left, post_value_right;
      int temp_act_left, temp_act_right, temp;

      /* Calculate update */

      temp_act_left = (LEFT_CONSTANT * act_left) + request_left;
      temp_act_left >> LEFT_DIV;

      temp_act_right = (RIGHT_CONSTANT * act_right) + request_right;
      temp_act_right >> RIGHT_DIV;

      /* Limit Change */
      temp = temp_act_right - act_right;
      if ((abs(temp)) > MAX_CHANGE)
            if (temp > 0)
                  temp_act_right = act_right + MAX_CHANGE;
            else
                  temp_act_right = act_right - MAX_CHANGE;

      temp = temp_act_left - act_left;
      if ((abs(temp)) > MAX_CHANGE)
            if (temp > 0)
                  temp_act_left = act_left + MAX_CHANGE;
            else
                  temp_act_left = act_left - MAX_CHANGE;

      act_left = temp_act_left;
      act_right = temp_act_right;


      /* Bounds check */

      if (act_left > MAX_SPEED_LEFT)
            act_left = MAX_SPEED_LEFT;
```

```
      else if (act_left < MIN_SPEED_LEFT)
            act_left = MIN_SPEED_LEFT;

      if (act_right > MAX_SPEED_RIGHT)
            act_right = MAX_SPEED_RIGHT;
      if (act_right < MIN_SPEED_RIGHT)
            act_right = MIN_SPEED_RIGHT;

      /* Prevent motor strain and overheat by
         setting motors to zero in range where
         the robot is not moving              */

      if ((abs(act_right)) < MIN_MOVE_RIGHT)
            post_value_right = 0;
        else post_value_right = - act_right;

      if ((abs(act_left)) < MIN_MOVE_LEFT)
            post_value_left = 0;
        else post_value_left = act_left;

      /* Post update to motors */

        motor(RIGHT_ID, post_value_right);
        motor(LEFT_ID, post_value_left);

}


/*******************************************
 * Collision ***********************************
 *******************************************/

void collision( )
{

      int loop, mot_l, mot_r;

      mot_l = act_left;
      mot_r = act_right;

      read_ir(); /* Get new IR reading */

        behaviors[BHN_COLLISION * NUMBER_PARAMS + PRIORITY] = 0;

        for (loop = 1; loop < NUMBER_OF_IR-1; loop++) {
            if (ir_values[loop] > IR_THRESHOLD) {
                        behaviors[BHN_COLLISION * NUMBER_PARAMS + PRIORITY] =
70;
                  if (loop = 0) {
                        mot_l -= IR_CHANGE;
                        mot_r -= IR_CHANGE;
                                SERVO(LEFT);
                  }
```

```
                if (loop = 2) {
                      mot_r -= IR_CHANGE;
                }
                if (loop = 3) {
                      mot_l -= IR_CHANGE;
                }
      }

        for (loop = 1; loop < NUMBER_OF_IR-1; loop++) {
            if (ir_values[loop] > IR_PANIC)
                      behaviors[BHN_COLLISION * NUMBER_PARAMS + PRIORITY] =
100;
                if (loop = 2) {
                      mot_l -= IR_CHANGE;
                      mot_r -= IR_CHANGE;
                }
                if (loop = 1) {
                      mot_r -= IR_CHANGE;
                }
                if (loop = 3) {
                      mot_l -= IR_CHANGE;
                }
            }
      }

        behaviors[BHN_COLLISION * NUMBER_PARAMS + MOT_L] = mot_l;
        behaviors[BHN_COLLISION * NUMBER_PARAMS + MOT_R] = mot_r;
}

/*******************************************
 * Initalize***********************************
 *******************************************/

/* Initalizes the needed interrupt routines for the scout robot */

void scout_init ()
{
      INTR_OFF();

      ADDR7 = 0xFF;   /* Turn on IR emmitters */

      init_motors();  /* Initalize the motor routines */
      init_servo();   /* Initalize the servo routine */
      init_analog();  /* Initalize the analog routines */
      init_ir();      /* Take first IR sample */

      INTR_ON();

        servo(CENTER);

}

/* behav_init *********************************/
```

```
void behav_init()
{
        int temp;
        for (temp=0; temp < (NUMBER_PARAMS * NUMBER_BEHAVIORS); temp++)
        {
                behaviors[temp] = 0;
        }
}

/*******************************************
 * Main ************************************
 *******************************************/

void main()
{
        int behavior, max_value, max_index, temp;
       scout_init();
        behav_init();


      while (1) {

            max_value = 0;

                  collision();

                for (behavior=0; behavior < NUMBER_BEHAVIORS; behavior++) {

                        temp =  behaviors[behavior * NUMBER_PARAMS +
STRENGTH] * behaviors[behavior * NUMBER_PARAMS + PRIORITY];
                  if (temp > max_value) {
                        max_value = temp;
                        max_index = behavior;
                  }

                update_motors(behaviors[max_index * NUMBER_PARAMS + MOT_L],
behaviors[max_index * NUMBER_PARAMS + MOT_R]);

            }


        }

}
```