Mad Max

Final Report

by
Chris Yust

for
Dr. Keith Doty

EEL5934
Intelligent Machines Design Laboratory

University of Florida
Fall, 1995

Table of Contents

Abstract

The development of autonomous agent behaviors depends on the sensor information available to the robot. Often, these behaviors ignore the robot's internal status in favor of data gleaned from the outside environment. Behaviors usually have a hard time obtaining and interpreting this data. Robots find object tracking, mapping, navigation, and other 'trivial' human tasks difficult to implement. They require 'active' sensors such as sonar, which return simple numeric data. Factors such as robot position, angle, and battery status make environmental readings highly temperamental. Currently available sensors (infrared, sonar, light detectors, etc.) do not provide adequate information to implement robust behaviors that depend on the outside environmental state.

Mad Max demonstrates behaviors derived from mostly internal sensors. Internal sensors generally require less complexity, which in turn increases reliability. The data provided by internal sensors may not match the 'information content' of external data. However, the behaviors based on this simple data should form a solid foundation for all autonomous agents as they attempt to interact with a dynamic external environment. Max implements three behaviors based on three sensors. Object avoidance, the only behavior reliant on external IR sensors, follows simple, proved techniques to guide Max through a room. The internal sensors, a wheel encoder and heat monitor, communicate important internal status to self-regulating behaviors. The wheel encoder allows Max to maintain a constant speed on different surfaces. He can also detect inclines and increase or decrease power as necessary to traverse them. The heat sensor determines when Max 'over-exerts' himself, and forces a rest-period to allow cooling. These behaviors mimic a human's adaptability and sense of 'tiredness.'

Executive Summary

Mad Max has again undergone a redefinition of behaviors and design goals since the last report. It was found that, as predicted by the IMDL staff, the wheel encoders required for accurate dead reckoning were beyond the scope of this class. Mapping and navigation were thus eliminated as possible behaviors in the remaining time. In retrospect it is believed that object recognition may play the most important part of navigation.

Max also underwent a significant structural change late in the development process. As described in the Actuators section, the original steering mechanism was replaced by a servo motor. This replacement greatly improved steering reliability and accuracy. The second major problem, the overheating of the motor driver chip, was solved by adding a second parallel driver chip, a heat sink, and a monitoring behavior to rest Max when temperatures reached dangerous levels.

The final Mad Max version successfully implements three behaviors based on three sensor types. The internal sensors proved relatively easy to implement and support as compared to external sensors. Future IMDL robots should give serious consideration, when appropriate, to using the internal data readily available to autonomous agents.

Effort throughout the semester to keep the implementation clean and cosmetically pleasing resulted in a very accessible robot. The EVBU board in particular can be lifted out by removing four screws. Max can be completely disassembled by removing other screw sets. The rechargable batteries can be quickly swapped by unfastening a single rubber band. Finally, all wires connecting the EVBU to Max's body pass through a 20-pin ribbon cable with removable connector.

Introduction

The scope of the Mad Max project was chosen to allow completion by the end of the semester (development cannot be continued in a formal sense due to coursework and time constraints). The resulting behavior set consists of basic yet useful routines that allow Max to roam safely about a room. In addition to the usual obstacle threat, Max's rear drive motor posed two additional problems: 1) high current requirements, and 2) low torque.

The high current requirements required an additional driver chip as described in the Actuator section. A heat-monitoring behavior was developed to preserve the integrity of this circuit. Modifications to the base RC car platform (addition of the battery pack, EVBU, etc.) resulted in a heavier car than the rear drive motor was designed to handle. A constant PWM signal, instead of producing a constant speed, drove Max at a variable speed depending on the surface conditions and incline. A constant-speed behavior superimposed on the PWM signal corrected for the heavier base weight.

Eight major parts compose this paper. The Integrated System section describes the relationship between the behaviors, sensors, and platform. The Mobile Platform section details the radio-controlled car base and controlling microprocessor. Actuation covers the rear drive and front steering mechanisms and their associated problems. The Sensors section depicts the three sensor sets and indicates their use relative to the behaviors. The Behavior section examines each behavior in detail. The Experimental Results and Conclusions summarize the verification and success of each behavior/sensor

combination. Appendix A contains all 'C' source code Max uses to implement his behavior set.

Integrated System

Mad Max follows general structural guidelines given by Brooks [1]. Multiple processes generate behaviors based on current sensor states. The competing behaviors feed information to an arbitration network, which determines Max's resulting action.

Behavior Signal Flow

Mad Max uses three sensors and three behaviors to navigate a room. He dodges obstacles using IR light reflection. Max can maintain a constant, ideal speed using wheel encoders in a feedback loop. Max also keeps from overexerting himself; a thermistor monitors the motor driver for dangerous heat build-up. The arbitration network realizes a simple priority scheme to choose between behaviors: Highest - heat monitoring, and Lowest - obstacle avoidance. Figure 1 shows the signal flow through the behavior modules. Each box corresponds to a separate software process.

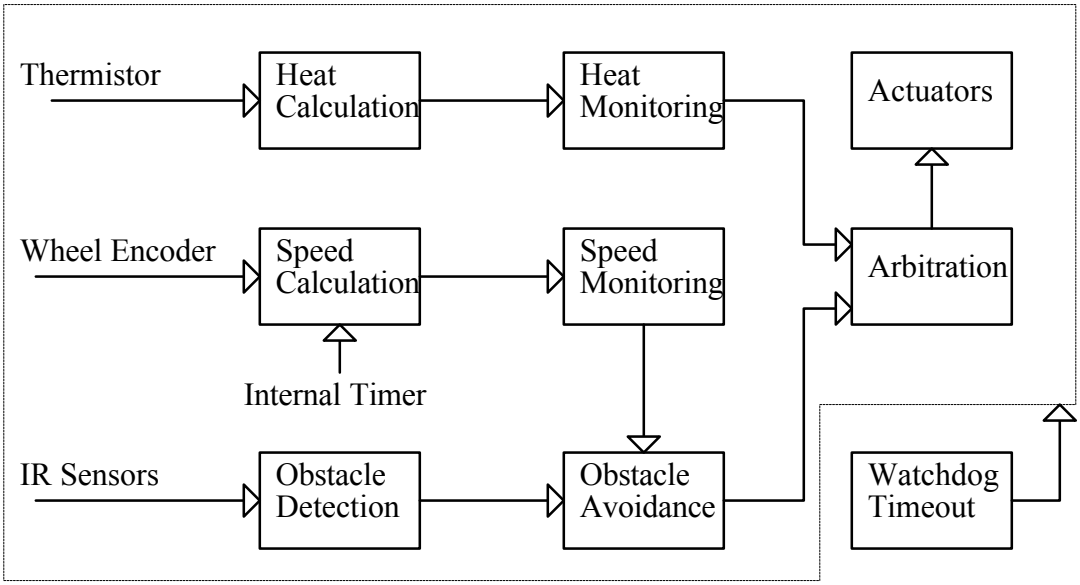


Fig. 1

Behavior Relationships

The object avoidance routine includes Max's "curiosity," driving him forward when no obstacles are present. The speed monitoring behavior works closely with object avoidance to keep Max moving at a desired speed. Whenever the current speed moves lower or higher than the desired speed the monitoring process will tell the object avoidance behavior to speed up or slow down. The self-preservation process provides an overriding cutoff switch to rest Max whenever the motor driver heat rises to dangerous levels.

Mobile Platform

Radio-Controlled Car Base

Max uses a radio-controlled car as a platform base. This platform has the movement restrictions of a regular car, the most limiting of which is a finite turning radius.

The "Invader" RC car by Nikko was chosen for several reasons. It provides left/right steering in both forward and reverse drive. Removing only 3 screws eliminated the plastic body, allowing easy access to the RC circuit board. Finally, the $39.99 price compared favorably to the other RC car choices.

Motorola HC11 Processor

The Motorola HC11 Evaluation Board gives Mad Max processing power. The HC11 was chosen for its simplicity, availability, and familiarity. A provided expansion circuit [3] gives high-level control over motors and sensors. Interactive C (IC), used by the MIT 6.270 course [2], allows programming in a high-level language. IC manages a multitasking environment suitable for the distributed processing module design and also includes routines to drive the expansion circuit.

Actuation

Front Steering Mechanism

Nikko designed the front wheels with a cheap turning mechanism. The front wheels each pivot about their center, moved by a common steering rod about .5cm behind the pivot points. Originally, an electromagnet attracted or repelled a permanent magnet attached to the steering rod, flopping the wheels left or right. The driving signal was measured at 2V across and .12A through the magnet. Some turn granularity was realized by simply driving the electromagnet with the same type of PWM signal as the rear drive motor.

Due to this cheap mechanism, poor rotational torque and unreliability at non-peak power conditions haunted Max for a considerable time. The problem was ultimately resolved by "major surgery." The electromagnet was entirely removed and replaced by a servo motor. Since the servo takes a larger volume of space than the electromagnet, Max's plastic body was altered to accommodate the servo casing. A hot soldering iron was used to melt a large, rectangular hole in the top of Max's body. The downward-pointing servo shaft was then attached to the steering arm. (see Fig. 2).

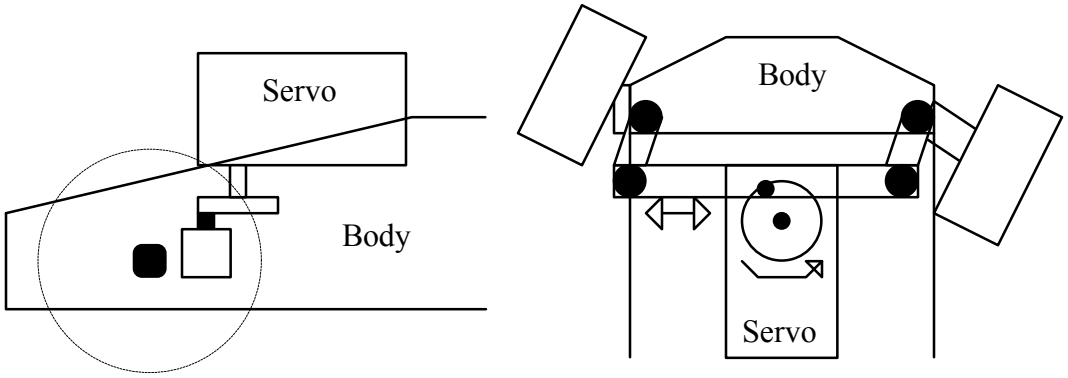


Figure 2.

The servo proved much easier to control than the electromagnet. The L293 motor driver was removed from the loop entirely, and its PWM enable signal was fed directly to the servo control line. Using the version of IC that supports floating point motor control resolution, the standard "motor(DRIVE,x);" command steered Max through a full turning radius. The actual values of 'x' ranged from 1.0 to 3.0 (of a possible -100.0 to +100.0).

Rear Drive Motor

The Invader frame uses a single bi-directional DC motor to drive the rear axle. The original 4.8V power pack supplied a 3V difference across terminals for full-speed

movement. The radio control unit provided only one speed--fast. The motor drew .3Amps with no load and 1.15A stalled. Using the motor driver circuit supplied by [3] the HC11 can achieve multiple speed levels by varying the Pulsed Width Modulation (PWM) on the motor driver enable pin. The motor's high amperage requirements caused overheating of the original motor driver circuit. This problem was tempered by stacking a second L293 chip in parallel over the first, providing double the current sourcing capabilities. Since the L293 contains two driver circuits per chip, the power requirement was spread across the entire chip by connecting outputs from both circuits to the drive motor.

Sensors

Sharp IR Detectors

Sharp GP1U5-8X IR detectors meet the first requirement of any mobile, autonomous agent--object avoidance. The detectors receive 40kHz modulated IR light reflected from objects in the near vicinity. Mad Max supports 6 Sharp IR sensors, three forward and three reverse, as shown in in Fig. 3.

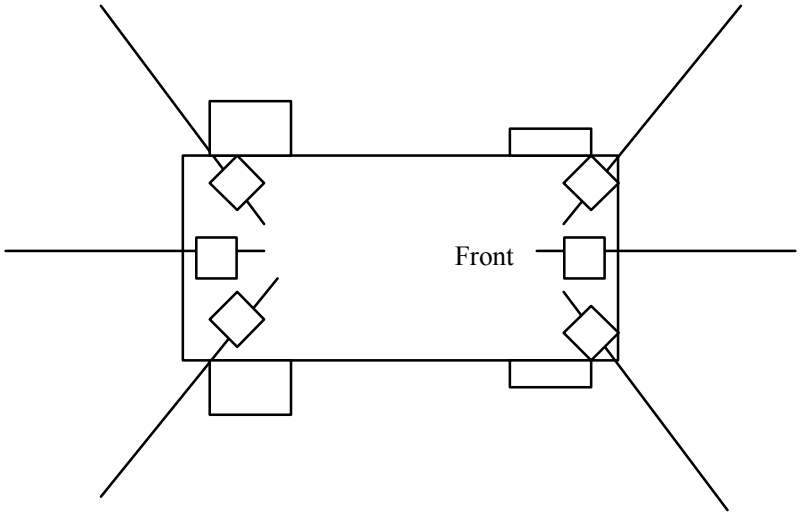


Figure 3.

Wheel Encoders

Mad Max uses a Siemens SFH9000 miniature light reflection emitter/sensor [D7] to encode the rear drive axle. The first attempt to encode the rear wheel used a standard "doughnut" with 36 black and white teeth around the rim of the wheel. This was discarded since 1) Max inherently has a large degree of slop in the wheels and cannot use that level of accuracy, and 2) the sensor was difficult to mount such that it reached all the way out to the rim of the tire. The second solution divided the wheel into quarters with 4 white-out stripes. The axle was marked near the body making it easy to position the sensor securely. The sensor output was run through a standard 74LS04 [D1] inverter to produce a digital pulse whenever a white stripe passes beneath the sensor. This arrangement seems immune to external light sources. Figure 4 depicts the encoder circuit.

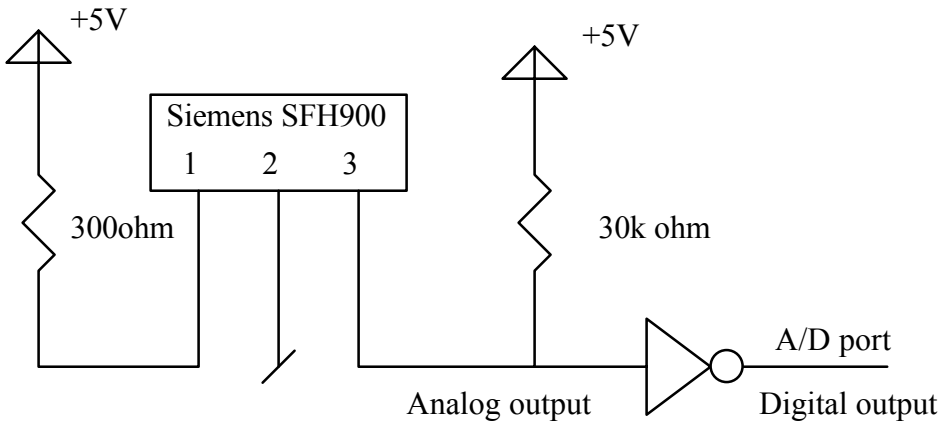


Figure 4.

Note that an A/D port interprets the digital signal output! The 'ticks' were originally fed to the pulse accumulator. However, a severe bouncing problem produced totally unuseable results. Attempts to debounce the analog output by putting a capacitor across

the inverter input resulted in even worse bouncing. Apparently, the bouncing was introduced by the inverter itself as the analog output moved slowly across the TRUE/FALSE threshold. Polling the A/D port every 10ms via a dedicated software process effectively debounced the digital output.

Thermistor

A simple thermistor provides heat feedback on the L293 motor driver circuit. It lies fastened to the side of the heat sink surrounding the dual L293 stack. Resistance nominally starts at 10k ohm (room temperature) and decreases with increased temperature. The circuit of Figure 5 converts temperature to an analog value ranging from 20 (cool) to 40 (very hot).

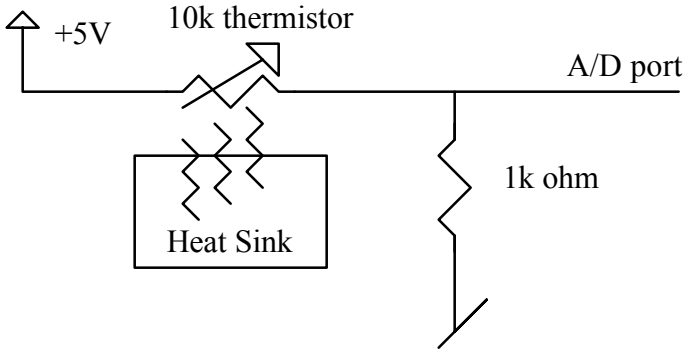


Figure 5.

Behaviors

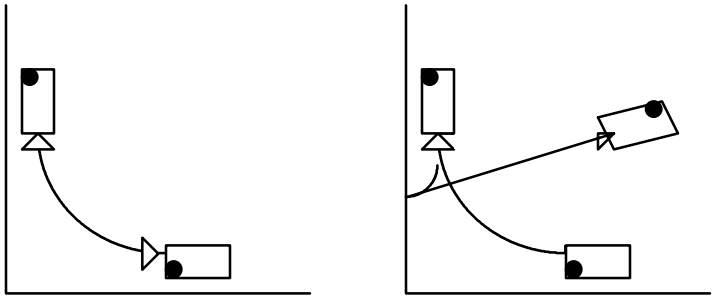
Obstacle Avoidance

The original avoidance behavior simply steered Max left or right away from obstacles. The front wheels were turned proportional to the difference between the analog values of the left and right sensors: iTurnDegree = K * (iAnalogLeft - iAnalogRight).

Max's finite turning radius means there will be some obstacles from which he cannot steer away. Intrusion into a buffer zone (large for the center sensor, small for the side sensors) forced Max to reverse direction. Behavior in the reverse direction mirrored forward motion.

Two processes support this behavior. "UpdateIRData" continually reads the IR sensors, shifts the readings to a zero-base, and stores the result in a global array. On reset, UpdateIRData will read each IR sensor to obtain a "clear" baseline reading (expected to range from 84-86). This individual threshold later subtracts from the raw analog reading to shift each sensor to a common zero-base. The "AvoidObstacles" process monitors the global zero-based sensor array for objects, calculates any necessary steering changes, and reverses direction upon object intrusion into the buffer zone.

A simple modification to this scheme produced a much better 'roaming' pattern. In the reverse direction the left and right IR sensors were logically switched. As a result Max steered towards the nearest object. When the object came close enough, Max reversed direction again and headed forward in a radically new direction. The resulting pattern covered much more area, and as a bonus actually mimicked a real car's three-point turn (see Figure 6).



Original Modified

Figure 6.

Speed Monitoring

Two processes form the speed monitoring behavior. The first, "UpdateSpe ed," monitors the wheel encoder and calculates Max's current speed. The wheel encoder has a resolution of one tick per 1/4 revolution. UpdateSpeed counts the number of ticks per timeslice (500ms) and calculates the number of ticks/second.

The second process, "MonitorSpeed," compares the current ticks/second to an ideal speed (say, 8 ticks/sec or 2 wheel revs/sec). The difference between current and ideal speeds adds to the current motor PWM setting, which ranges from 0 to +100: iCurrentSpeed = iOldSpeed + (IDEAL_TICKS_PER_SEC - iTicksPerSec). Figure 7 visualizes this behavior in a simple feedback structure.

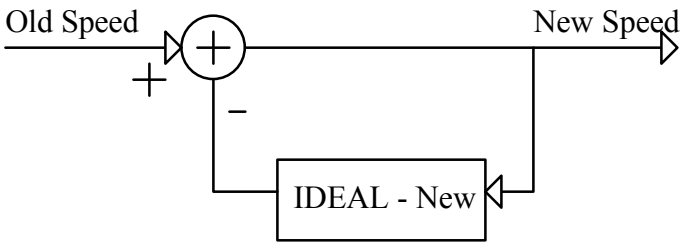


Figure 7.

The AvoidObstacle behavior reads the new speed setting, adds (+/-) direction information, and sends the request to the Arbitration process.

Heat Monitoring

A red LED reports Max's internal temperature to the external world. It blinks on and off (like a heartbeat) with a frequency proportional to temperature. Two processes support this behavior. The "UpdateHeat" process continually reads the thermistor analog output and ensures it falls between HEAT_MIN and HEAT_MAX constants (20 and 40,

for example). It then maps this reading to a value between 0 and 1000 and stores the result in a global variable.

The "MonitorHeat" process checks the current heat against 1000 (HEAT_MAX). When it detects this critical 'heart-attack' stage several things happen. First, the LED stays on solid red to indicate overheating. Second, the Arbitration process receives a signal to stop all motor activity. Third, the MonitorHeat algorithm waits until Max has cooled off by 20% (heat down to 800). When Max cools off the Arbitration block disappears and Max continues on his way.

Experimental Layout and Results

Behavioral Testing

All three behaviors were successfully tested. Max was able to avoid obstacles in an enclosed area. He ramped up to a target ideal speed on surfaces ranging from thick carpet to tile. When Max reached an inclined surface he increased power to the motors, moving onto the surface if possible. The heat monitoring behavior successfully shut down Max whenever high heat levels were detected. A quick touch verified a hot heat sink whenever the LED turned solid red.

Current Status

All basic behaviors have been implemented. Some fine-tuning of constants may be necessary for demonstration purposes; Max has never seen the 'arena' in the MIL lab.

Conclusion

This semester taught many lessons about basic robot construction. Most importantly, "It isn't a paper world!" Sensors and platforms performed differently than expected in almost every instance, resulting in several major design iterations. A robot's behaviors depend entirely on the capabilities of its sensors. More time should have been spent learning about sensor characteristics, perhaps even before platform selection was made.

In retrospect, a platform other than radio-controlled car would have been chosen. Several problems were encountered. First, the finite turning radius made all control algorithms more complex while limiting Max's mobility. Second, the RC body hindered major modifications which may be required by evolving design goals. Third, it proved difficult to mount the EVBU, battery pack, and IR sensors on limited hard-points. On the plus side, the RC car platform provided a very quick initial setup time for a relatively low cost. However, these advantages were outweighed by the disadvantages later in the semester.

Documentation

Books and Papers

[1]. "Elephants Don't Play Chess." Brooks, Rodney. From Designing Autonomous Agents, ed. by Pattie Maes. The MIT Press, c. 1990.

[2]. Martin, Fred et al. "1994 6.270 LEGO Robot Design Competition Course Notes."

[3]. Doty, Keith et al. "EVBU Expansion Schematic." Machine Intelligence Laboratory.

[4]. Doty, Keith. Class lectures in EEL5934 Intelligent Machines Design Laboratory. University of Florida, Fall, 1995.

Referenced Data Sheets

[D1]. Motorola MC74HC04A hex inverter.
[D2]. Motorola MC74HC10 triple 3-input NAND gate.
[D3]. Motorola MC74H138A 1-of-8 decoder.
[D4]. Motorola MC74HC390 dual 4-stage counter with /2 and /5 sections.
[D5]. Motorola MC74HC573A Octal 3-state noninverting latch.
[D6]. Motorola MCM60L256A 32kx8 bit CMOS SRAM.
[D7]. Siemens SFH9000 IR light reflection emitter/sensor.
[D8]. Texas Instruments uA7805 voltage regulator.
[D9]. Texas Instruments L293 quadruple half-H driver.

Appendix A: Behavior Code

```
/* Chris Yust, SS# 263-77-4056 */
/* EEL5934 Intelligent Machines Design Laboratory */
/* Fall, 1995 */

/* MADMAX.C : Mad Max's final control program. */

/*********************************************************/
/* 'define' constants */

int DRIVE=1;   /* motors */
int STEER=0;

int TRUE=1;
int FALSE=0;

int FRONT=0;
int REAR=1;

int FORWARD=1;
int STOP=0;
int REVERSE=-1;

int NUM_IR_SENSORS=6;

int IR_FRONT=0;          /* A/D port definitions */
int IR_FRONTLEFT=1;
int IR_FRONTRIGHT=2;
int IR_REAR=3;
int IR_REARLEFT=4;
int IR_REARRIGHT=5;
int WE_DRIVE=6;
int HEAT=7;

int MIDDLE_PAD=25;  /* minimum distance to front sensor */
int SIDE_PAD=35;    /* minimum distance to side sensors */

int MAX_SPEED=50;   /* in motor PWM */
int IDEAL_SPEED=4;  /* in ticks per sec */

float STEERMAX=2.9; /* servo range for full turning */
float STEERMIN=1.1;
float STEERMAP=(STEERMAX-STEERMIN)/200.0;

int LED1_MASK=0x01; /* port 0x7000 bit definitions */
int LED2_MASK=0x02;
int HEAT_MASK=0x04;

int HEAT_MIN=20;    /* minimum and maximum heat values */
int HEAT_MAX=40;

long TIMESLICE=500L;     /* milliseconds per speed update */

float TIMEOUT=60.0;      /* seconds of lifetime */

/*********************************************************/
```

```
/* global variables */

/* for speed tracking */
int giCurrentSpeed;      /* current suggested motor PWM */
int giTicksPerSec;       /* current speed in 1/4rev / sec */
int giSpeedDelta;

/* for arbitration */
int gbHeartAttack;  /* indicates Max has overheated */
int giDriveRequest; /* AvoidObstacles drive suggestion */
int giSteerRequest; /* AvoidObstacles steer suggestion */

/* for obstacle avoidance */
int garriObstacle[6];    /* current IR sensor states */
int garriThreshhold[6];  /* used to convert to zero-base */

/* for Timeout process */
int gbUpdateIRData;      /* Watchdog timer sets these */
int gbUpdateSpeed;       /*   to false, killing all   */
int gbUpdateHeat;        /*   processes when lifetime */
int gbMonitorSpeed;      /*   expires. */
int gbMonitorHeat;
int gbAvoidObstacles;
int gbArbitration;
int gbSetNewDrive;

/* for AvoidObstacles */
int giMiddleSensor;      /* current machine state */
int giLeftSensor;
int giRightSensor;
int giDirection;
int giNewDrive;
int giNewSteer;

/* current heat status */
int giHeat;

/* misc */
int giPort7000;     /* value of bits at port 0x7000 */

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

void main()
{
  int iFrontSensor,iBackSensor;

  /* Take quick reading on front and back sensor.  If */
  /* either one is obstructed, do NOT start Max!      */

  poke(0x7000,LED1_MASK|LED2_MASK);
  wait(100);
  iFrontSensor=analog(IR_FRONT);
  iBackSensor=analog(IR_REAR);
  poke(0x7000,0);

  if ((iFrontSensor>120)||(iBackSensor>120))
```

```
  {
     /* do nothing! */
     poke(0x7000,HEAT_MASK); /* solid red */
  }
  else
  {
    /* sensory input processes */
    start_process(UpdateIRData());
    start_process(UpdateSpeed());
    start_process(UpdateHeat());

    /* behaviors */
    start_process(AvoidObstacles());
    start_process(MonitorSpeed());
    start_process(MonitorHeat());

    /* helper processes */
    start_process(SetNewDrive());
    start_process(TimeOut(TIMEOUT));

    /* behavior arbitration */
    start_process(Arbitration());
  }
}

/*********************************************************/
/*********************************************************/
/* sensory input processes                               */
/*********************************************************/
/*********************************************************/

/*********************************************************/
/* Update IRData                                         */
/*********************************************************/

void UpdateIRData()
{
  int iSensor;

  /* get threshhold value for each IR sensor; */
  /* clear current obstacle */
  for (iSensor=0;iSensor<NUM_IR_SENSORS;iSensor++)
  {
     garriThreshhold[iSensor]=analog(iSensor)+1;
     if (garriThreshhold[iSensor]>100)
     { /* sensor maxxed out */
       garriThreshhold[iSensor]=100;
     }
     garriObstacle[iSensor]=0; /* initialize global data */
  }

  gbUpdateIRData=TRUE;    /* set FALSE by TimeOut to end */

  while (gbUpdateIRData==TRUE)
  {
     /* read all sensors */
     for (iSensor=0;iSensor<NUM_IR_SENSORS;iSensor++)
     {
        garriObstacle[iSensor]=
           analog(iSensor)-garriThreshhold[iSensor];
```

```
        if (garriObstacle[iSensor]<0)
        {
          garriObstacle[iSensor]=0;
        }
     }
     wait(100);     /* wait 100 milliseconds */
  }
}

/*********************************************************/
/* helper functions                                      */

/* none */

/*********************************************************/
/* Update Speed                                          */
/*********************************************************/

void UpdateSpeed()
{
  int iNewDrive=0;
  int iOldDrive;
  int iTicks;

  long lOldTime;
  long lNewTime;

  giTicksPerSec=0;       /* initial speed is standstill */
  iTicks=0;

  gbUpdateSpeed=TRUE;  /* set FALSE by TimeOut to end */

  lNewTime=mseconds();

  while (gbUpdateSpeed==TRUE)
  {
     lOldTime=lNewTime;  /* save start of this timeslice */

     iTicks=0; /* reset ticks count */

     while (lNewTime<(lOldTime+TIMESLICE))
     {
       /* check for ticks */
       iOldDrive=iNewDrive;
       iNewDrive=analog(WE_DRIVE); /* digital value */

       /* tick occurs on a black to white transition  */
       if ((iOldDrive<128)&&(iNewDrive>128))
       {
         iTicks=iTicks+1;
         wait(10); /* debounce padding time */
       }

       lNewTime=mseconds();   /* update current time */
       defer();
     }

     /* do speed calculation */
     giTicksPerSec=iTicks*1000/(int)(lNewTime-lOldTime);
```

```
    /* check for timer wraparound */
    if (giTicksPerSec<0) giTicksPerSec=0;
  }

  /* clean up process */

}

/*********************************************************/
/* helper functions                                      */

/* none */

/*********************************************************/
/* Update Heat                                           */
/*********************************************************/

void UpdateHeat()
{
  int iHeat;

  gbUpdateHeat=TRUE;  /* set FALSE by TimeOut to end */

  while (gbUpdateHeat==TRUE)
  {
    /* expected range: HEAT_MIN -> HEAT_MAX */

    iHeat=analog(HEAT);  /* read thermistor */
    if (iHeat < HEAT_MIN) iHeat=HEAT_MIN;
    if (iHeat > HEAT_MAX) iHeat=HEAT_MAX;

    iHeat=iHeat-HEAT_MIN;            /* move to zero base */
    iHeat=(HEAT_MAX-HEAT_MIN)-iHeat; /* reverse 'sense' */

    /* now 0 = heart attack, (HEAT_MAX-HEAT_MIN) = cool */

    /* stretch 0->(MAX-MIN) range to 0->1000 */
    /* note: keep (MAX-MIN) <=32 to prevent overflow! */
    iHeat=(iHeat*1000)/(HEAT_MAX-HEAT_MIN);

    giHeat=iHeat;   /* set global variable */

    wait(1000); /* check new heat every second */
  }

  /* clean up this process */

}

/*********************************************************/
/* helper functions                                      */

/* none */

/*********************************************************/
/*********************************************************/
/* behavior processes                                    */
/*********************************************************/
/*********************************************************/
```

```
/*********************************************************/
/* Avoid Obstacles                                       */
/*********************************************************/


void AvoidObstacles()
{
  int bChange;
  int iMiddleSensor,iLeftSensor,iRightSensor;

  int iNewSpeed=0;
  int iNewSteer=0;

  /* straighten out wheels */
  motor(STEER,(STEERMAX-STEERMIN)/2.0);

  SetDirection(FORWARD);

  gbAvoidObstacles=TRUE;  /* set FALSE by TimeOut to end */

  while (gbAvoidObstacles==TRUE)
  {
    /* check for imminent collisions */

    /* extract sensor data in this direction */
    iMiddleSensor=garriObstacle[giMiddleSensor];
    iLeftSensor  =garriObstacle[giLeftSensor  ];
    iRightSensor =garriObstacle[giRightSensor ];

    bChange=FALSE;
    if (iMiddleSensor>MIDDLE_PAD) bChange=TRUE;
    if (iLeftSensor  >SIDE_PAD  ) bChange=TRUE;
    if (iRightSensor >SIDE_PAD  ) bChange=TRUE;

    if (bChange==TRUE)
    {
      iNewSpeed=0;  /* stop drive motor in this direction */

      if (giDirection==FORWARD) SetDirection(REVERSE);
      else                      SetDirection(FORWARD);

      /* Let sensor data settle */
      wait(1000);
    }

    /* wandering around algorithm */

    /* speed calculation - trivial */
    iNewSpeed=giCurrentSpeed;

    /* set actual speed request in global variable */
    giDriveRequest=iNewSpeed*giDirection;

    /* steering calculation */
    if (iabs(iLeftSensor-iRightSensor)>3)
    {
      iNewSteer=5*(iLeftSensor-iRightSensor);
      if (iNewSteer>100) iNewSteer=100;
      if (iNewSteer<-100) iNewSteer=-100;
    }
```

```
    else iNewSteer=0;

    /* set actual steer request in global variable */
    giSteerRequest=iNewSteer;

    defer();

  }

  /* clean up this process */
  iNewSpeed=0;
  giNewDrive=0;
  giSteerRequest=0;

}

/*********************************************************/
/* helper functions                                      */

void SetDirection(int iDirection)
{
  ResetSpeed();

  if (iDirection==FORWARD)
  {
    /* switch LED bits */
    giPort7000=(giPort7000&(~LED2_MASK))|LED1_MASK;
    poke(0x7000,giPort7000);   /* turn on front IR LEDS */

    giLeftSensor=IR_FRONTLEFT;
    giMiddleSensor=IR_FRONT;
    giRightSensor=IR_FRONTRIGHT;
    giDirection=FORWARD;
  }
  else /* giDirection=REVERSE */
  {
    /* switch LED bits */
    giPort7000=(giPort7000&(~LED1_MASK))|LED2_MASK;
    poke(0x7000,giPort7000);   /* turn on rear IR LEDS */

    giLeftSensor=IR_REARLEFT;
    giMiddleSensor=IR_REAR;
    giRightSensor=IR_REARRIGHT;
    giDirection=REVERSE;
  }
}

/*********************************************************/
/* MonitorSpeed                                          */
/*********************************************************/

void MonitorSpeed()
{

  giCurrentSpeed=0; /* start Max at standstill */

  gbMonitorSpeed=TRUE;  /* set FALSE by TimeOut to end */

  while (gbMonitorSpeed==TRUE)
  {
```

```
    /* delta between old and ideal speed, in ticks/sec */
    giSpeedDelta=IDEAL_SPEED-giTicksPerSec;

    /* new (current speed) is old speed plus delta */
    giCurrentSpeed=giCurrentSpeed+giSpeedDelta;

    if (giCurrentSpeed>MAX_SPEED) giCurrentSpeed=MAX_SPEED;
    if (giCurrentSpeed<0)         giCurrentSpeed=0;

    wait((int)TIMESLICE); /* allow change to take effect */
  }

  /* clean up this process */

}

/*********************************************************/
/* helper functions                                      */

/* none */

/*********************************************************/
/* MonitorHeat                                           */
/*********************************************************/

void MonitorHeat()
{
   int iPeriod;

   gbMonitorHeat=TRUE;  /* set FALSE by TimeOut to end */

   while (gbMonitorHeat==TRUE)
   {
      if (giHeat!=0) /* if not at 'heart attack' status */
      {
        gbHeartAttack=FALSE;

        /* calculate pulse period, in ms */
        /* remember: giHeat 0->1000 */
        iPeriod=giHeat*1;     /* slowest is 1 second */
        iPeriod=iPeriod/2; /* LED on, off for 1/2 period */

        /* blink pulse LED */

        /* turn off heat LED */
        giPort7000=giPort7000&(~HEAT_MASK);
        poke(0x7000,giPort7000);
        wait(iPeriod);

         /* turn on  heat LED */
        giPort7000=giPort7000|( HEAT_MASK);
        poke(0x7000,giPort7000);
        wait(iPeriod);
      }
      else  /* heart attack! */
      {
        /* stop MAX */
        gbHeartAttack=TRUE;

        /* solid red light */
```

```
        giPort7000=giPort7000|( HEAT_MASK);
        poke(0x7000,giPort7000);

        while(giHeat>800) /* wait until Max cools 20% */
        {
          defer();
        }
      }
   }

  /* clean up this process */
  giPort7000=giPort7000&(~HEAT_MASK); /* turn off heat LED */
  poke(0x7000,giPort7000);

}

/*********************************************************/
/* helper functions                                      */

/* none */

/*********************************************************/
/*********************************************************/
/* helper processes                                      */
/*********************************************************/
/*********************************************************/

/*********************************************************/
/* SetNewDrive                                           */
/*********************************************************/

void SetNewDrive()
{

  int iCurrentDrive=0;

  giNewDrive=0;
  motor(DRIVE,(float)giNewDrive);

  gbSetNewDrive=TRUE;   /* set FALSE by TimeOut to end */

  while (gbSetNewDrive==TRUE)
  {
    /* if need to change speed */
    if (giNewDrive!=iCurrentDrive)
    {
      /* if opposite directions */
      if ((giNewDrive*iCurrentDrive)<0)
      {
        motor(DRIVE,0.0);  /* kill current direction */
        wait(1000);  /* wait for intertia to fade */

        /* ramp up to new speed */
        iCurrentDrive=giNewDrive;
        motor(DRIVE,(float)iCurrentDrive);

      }
      else /* same direction, different speed */
      {
        iCurrentDrive=giNewDrive;  /* change immediately */
```

```
        motor(DRIVE,(float)iCurrentDrive);
      }
    }
    defer();

  }

  /* clean up this process */
  motor(DRIVE,0.0);

}

/*********************************************************/
/* helper functions                                      */

/* none */

/*********************************************************/
/* TimeOut                                               */
/*********************************************************/

void TimeOut(float fLifeSeconds)
{
  float fStartTime;
  float fEndTime;
  float fNow;

  reset_system_time();

  fStartTime=seconds();  /* get start time */
  fEndTime=fStartTime+fLifeSeconds; /* calculate end time */

  while (seconds()<fEndTime)  /* wait until lifetime ends */
  {
    wait(1000);  /* wait a second */
  }

  /* time's up; kill all processes and shut down Max */
  gbUpdateIRData=FALSE;
  gbUpdateSpeed=FALSE;
  gbUpdateHeat=FALSE;

  gbAvoidObstacles=FALSE;
  gbMonitorSpeed=FALSE;
  gbMonitorHeat=FALSE;

  gbSetNewDrive=FALSE;
  gbArbitration=FALSE;

}

/*********************************************************/
/* helper functions                                      */

/* none */

/*********************************************************/
/*********************************************************/
/* Arbitration                                           */
/*********************************************************/
```

```
/*********************************************************/

void Arbitration()
{

  gbArbitration=TRUE;   /* set FALSE by TimeOut to end */

  while (gbArbitration==TRUE)
  {
    if (gbHeartAttack==TRUE)
    {
      ResetSpeed();

      /* new drive (another process handles motor cmd) */
      giNewDrive=0;

      /* set new steer */
      giNewSteer=0;
      SetSteer();
    }
    else
    {
      /* new drive (another process handles motor cmd) */
      giNewDrive=giDriveRequest;

      /* set new steer */
      giNewSteer=giSteerRequest;
      SetSteer();

    }
  }

  /* clean up this process */
  giNewDrive=0;
  giNewSteer=0;
  SetSteer();

}

/*********************************************************/
/* helper functions                                      */

void SetSteer()
{
   float fNewSteer;

   fNewSteer=(float)(giNewSteer+100);
   fNewSteer=(fNewSteer*STEERMAP)+STEERMIN;
   motor(STEER,fNewSteer);
}

/*********************************************************/
/*********************************************************/
/* misc. functions                                       */
/*********************************************************/
/*********************************************************/

void wait(int ms)
{
  long timer;
```

```
  timer=mseconds() + (long)ms;
  while (timer>mseconds())
  {
    defer();
  }
}


float fabs(float fNum)
{
  if (fNum>=0.0) return(fNum);
  else return(-fNum);
}

int iabs(int iNum)
{
  if (iNum>=0) return(iNum);
  else return(-iNum);
}

void ResetSpeed()
{
  giDriveRequest=0;
  giCurrentSpeed=0; /* force Max to ramp up again */
  giTicksPerSec=0;
}
```