Final Report

Creation of a Mobile Robotic Design

Designer: Charles R. Barker, Jr.

EEL5934 Robotics - Fall 1995 - University of Florida
Intelligent Machines Design Laboratory

Instructor: Keith L. Doty
TA's: Erik de la Iglesia
Scott Jantz

December 4, 1995

Table of Contents

Abstract

This report is on the construction of an autonomous robotic platform for EEL5934 - Robotics at the University of Florida. The first construction phase was concerned with developing a platform on which to develop future plans. The second phase was centered on creating a sensor set to allow the robot to develop a view of its environment and react to it. The primary sensors developed were the standard IR collision avoidance sensors, a motor commutator sampler, and an improved version of the "Cyclops" 360° object detection sensor. The third and fourth phases were aimed at creating a behavior set in software that could read and interpret the sensor data in some meaningful way, and then react to it, directing the robot's only actuation system, the drive motors.

Executive Summary

The robot created for this project is built around the design of an improved Cyclops 360° vision system. Every aspect of the robot is centered around this sensor. A simple, low-maintenance platform, protective dome casing, and basic sensor package were made with the sensor in mind.

The platform itself is highly mobile, if not fast. The round-platform, dual-wheel approach makes it nimble enough to rotate in place and navigate areas just slightly larger than it's diameter. The internal structure has a second deck to elevate the Cyclops sensor at midline.

The dome casing contains the IR Emitter banks, as well as easy-access switches, a serial port, and a charging jack. This lets the user perform all downloads and battery charges without having to dismantle the robot.

The sensor suite incudes standard IR sensors along the base for low-level collision avoidance, and a motor commutator noise sampler, designed to aid in straight line navigation, odometry, and stall detection.

The Cyclops System is at the heart of the design, allowing active or passive reading of 40KHz modulated IR in a full 360° sweep of the environment at a sampling rate of better than one full revolution per second. The sensor has none of the limitations of previous implementations, like blind spots, instability, or CPU-intensive routines. Currently, the sensor can reliably take 16 readings per revolution, though this number may be increased at the cost of higher CPU overhead and/or lower revolution speed.

The software package allows for multitasked Sensor Daemons and Behaviors, running under IC. Currently, the Daemon package is essentially complete, collecting and filtering data from the sensors reliably and in real time. The Behavior package is sparse, including Bump 'N Go, Collision Avoidance, and Trapped behaviors. However, these can be easily expanded on with only the addition of behavior code and adjustments to the arbitration system.

Introduction

The goal of this project was to construct an autonomous robotic platform with capabilities that met the requirements of EEL5934 for basic development. The eventual goal for meeting the sensor requirements was to construct a 360-degree "vision" system similar to the one seen on "Cyclops," a robot designed by Rosa Maria Charneca Pasadas and Rui Jorge Ferreira da Costa. The new design would improve upon Cyclops's design to allow both passive and active tracking of other robots, tracking of light sources, and improved 360-degree collision avoidance. The new design would also be more rugged, controllable, and modular than the original.

Integrated System

The robot's hardware systems are integrated as depicted in Figure 1. The input packages, the sensors, are monitored by "daemon" processes, which feed values to the behaviors. The behaviors are monitored by an arbitrator, which in turn sends signals to the output systems, primarily the motor driver system.

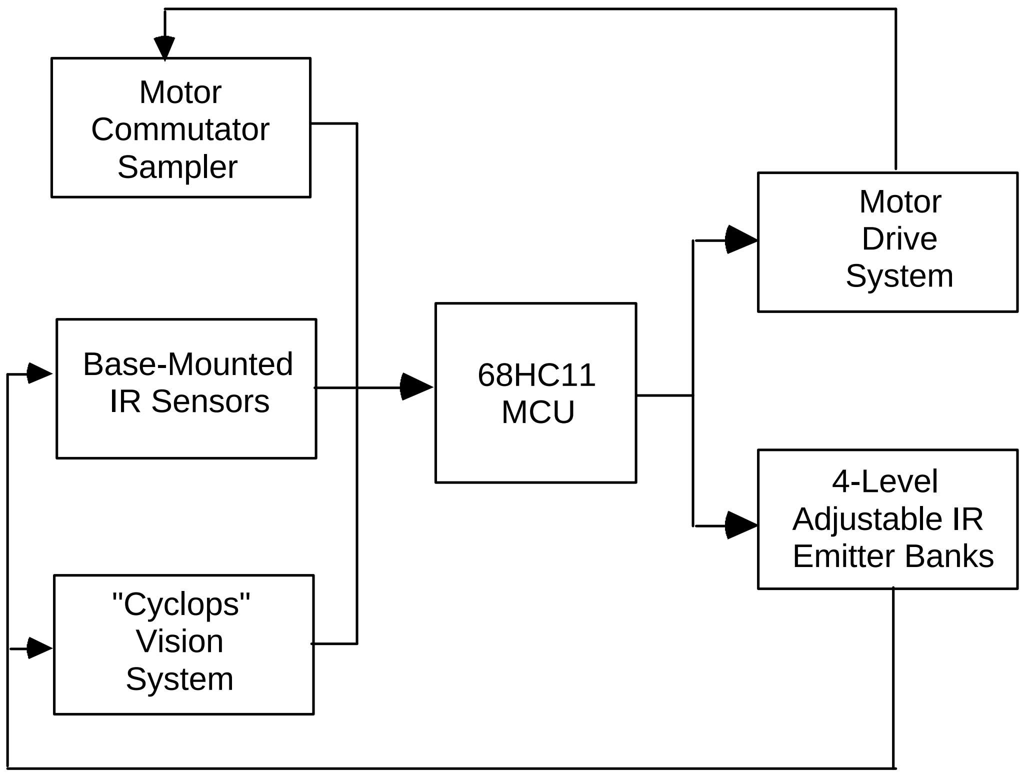

Figure 1.

Each sensor and monitoring daemon package produces a distinct data set which is handled by the behavior modules. (Figure 2.) These monitoring daemons are run as independent processes under IC's multitasking system and are explained in detail in the Sensors section.

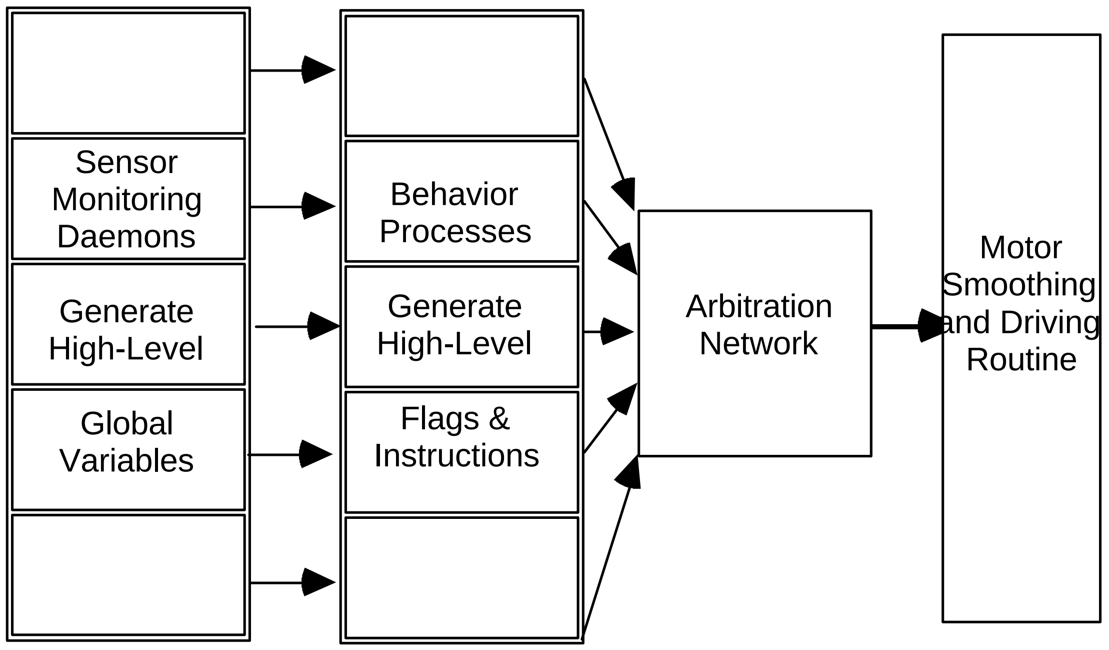

Figure 2.

Each Behavior takes in the sensor package variables and determines a course of action or a flag to raise. These actions and flags are then evaluated by an arbitration network, which is in itself a behavior. From there, the desired action is buffered by a motor driver which directly interfaces with the motors. These behaviors are described in the Behaviors section.

Mobile Platform

Power and MCU Systems

Power to the robot is provided by onboard 8-AA cells arranged in a 9.6V battery. The battery provides both drive motor and MCU power (via a voltage regulator). The battery is rechargeable via a connection jack which is mounted on the rear of the robot. This battery pack can last several hours per charge.

The MC69HC11E9 board from Motorola has been expanded based on the standard design schematic for the course. This design includes a 32K RAM expansion, voltage regulator, and motor / IR LED driver circuits.

Mobile Chassis Design

The design of a robot's platform can greatly enhance the robot's capabilities, allowing it to traverse a multitude of terrains without problems, and likewise make the rest of the design drastically simpler. By making a "worry-free" and robust chassis, the designer spares the need to compensate for shortcomings with extra sensors and algorithms. For example, a wide footprint design with outboard wheels and a properly placed center of gravity eliminates any need for balance sensors and the associated software to read and react to them. In keeping with this philosophy, the goals of the platform design were simplicity, easy access and expandability, and stability.

The base of the robot is a 12-inch diameter circular sheet of lightweight birch plywood with reinforcement cross members supporting the two drive motors. A single castor at the rear of the robot forms the third ground contact. The bulk of the robot's weight lies above this castor to prevent the nose from dragging. The wheels are 2.5-inch shock-absorbing model airplane wheels, and fit into wheel well notches on the sides of the platform.

The goals of the overall design, specifically the Cyclops sensor, brought a number of unknowns into the chassis design at the time it was built. The associated mechanics of the Cyclops sensor could have been bulky and heavy. The Cyclops sensor needed to be placed near the center of the robot to prevent the need for compensation calculations, but the microcontroller and batteries also had to fit onto the board. The base simply didn't have enough room for all of these parts, so a second level was added above the first to mount the drive mechanism and sensors. The LED's for the Cyclops sensors and the numerous switches and jacks needed to be mounted on something as well, so a Fuller-style dome was created from thick cardboard both to mount components on, and to protect the robot itself from damage. This dome was also painted and sealed to increase strength. The dome also serves to block out IR from undesired sources to the Cyclops IR sensors. These sensors would have been susceptible to IR from other robots or reflective surfaces without a shield around them.

Motor Drive

The only actions the robot is capable of are ground movements. There are no plans for additional actuators, though in some respect the motor drive for the Cyclops sensor is an actuator, even if it doesn't react directly with the environment.

A robot's motor drive system can simplify it's control systems in much the same way a well-designed platform can. Here, the goals centered around simple control algorithms and low power consumption. The easier it is for the software to translate desired travel into motor commands, the more instruction cycles are free for other algorithms.

A modification to two Futaba FP-S148 servos allowed continuous rotation. These motors are ideal for the robot's drive system, since they are geared down appropriately and well-sealed for low maintenance. The motors are fed a pulse-width-modulated signal from the 293 motor driver and can rotate forwards and backwards. They use power from the on-board 9.6V battery, and are relatively efficient. The wheels are glued to the servo horns and screwed directly onto the output shaft. With the servos mounted at midline, the robot can steer and spin much like a tank can. The robot can easily turn in any direction without having to back out of a tight position. This, again serves to simplify future coding.

Sensors

Sharp IR Sensors

The robot has a battery of 6 analog-modified Sharp IR sensors mounted in a crossing pattern on the base of the platform. (Figure 3.) These were moved from their positions on top of the robot to the bottom side, mainly to allow for the robot's dome-casing design. These pick up reflections from the four 40-KHz modulated IR LED's mounted under the platform. These four LED's are now controlled by a switching circuit that uses only two lines from the latch to switch resistor banks, much like the method Erik de la Iglesia used. This switching system allows for 3 different brightness levels for the LED's when activated. This ability is probably going to be only marginally useful for the low-level collision avoidance sensors, but the circuit was added primarily for use with the "Cyclops" sensor, making the lower LED bank only a trivial addition.

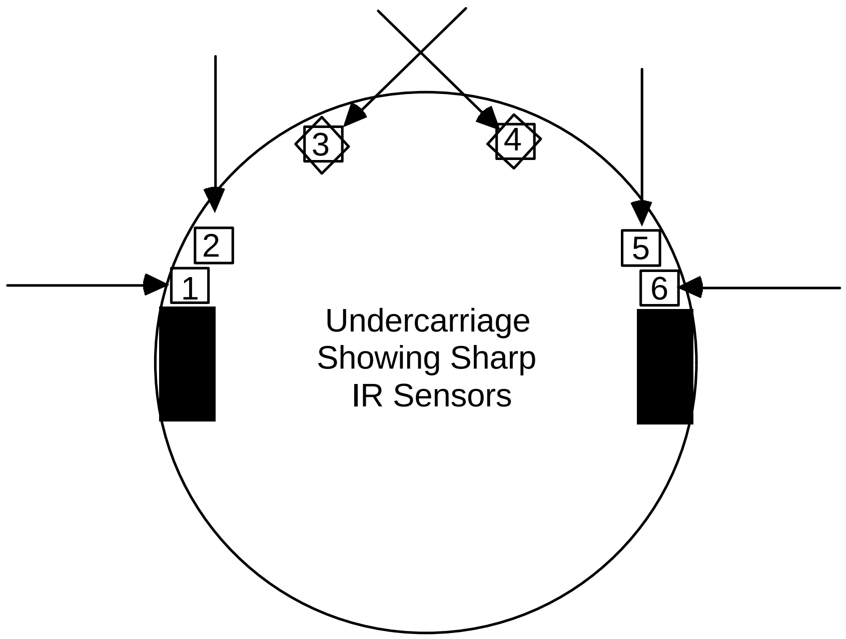

Figure 3.

This sensor arrangement provides ample information about nearby objects. The method is very much "time-tested" by previous semesters of EEL 5934, and is fairly reliable, though less so if the environment contains IR-absorbent materials.

The Base-Mounted IR sensors are polled routinely by the Base IR Monitoring Daemon. This package generates several arrays as output, with each element representing a different sensor. The arrays hold actual "analog" return values, deltas from previous samplings, and a computed factor based on the current actual value and the delta value.

Motor Commutator Sampler Sensor

This circuit is designed as a replacement for the traditional "wheel encoder" system for odometry. The theory behind it and the circuit itself were both first utilized in this class by Erik de la Iglesia. The system relies on the inductive feedback from the point where the motor's coils are switched. When the commutator moves from pickup to pickup, the coils discharge and create a spike. This spike can typically bring havoc upon digital systems, but is used here as a very fast (about 1KHz) pulse representation of the motor's speed. The advantages of this system over the traditional wheel encoders are the lack of additional moving parts and the very short time needed to take a representative sample. At full speed, the motor need only be sampled for about 80ms to get around 70 pulses in the accumulator, which is enough to estimate the motor's actual speed. These pulses are measured with the functions in CASEY3.C, in the appendix. This information can also be used to compare the relative speeds of the two motors to help insure straight-line motion under IC's PWM system, and even allow self-calibration. Testing has also shown that the if the robot were to hit an object, the motor pulse count will fall off significantly, even if the wheels slip some. This was fact was used in the "Bump 'N Go" Behavior much like a low-sensitivity bumper to detect if the robot had stalled or partially stalled a motor, assuming that it had hit an unseen object.

The only anticipated problems were with the PWM signal from the HC11, as this pulse would also appear as a noise spike to the system. Fortunately, the PWM signal is only about 33Hz and therefore can only really create a maximum of 2 or 3 noise spikes for every sample, and would be equally present in both channels if set at the same speed.

The circuit in the appendix shows the concept. A two stage OP Amp circuit is used as a comparator to first generate a pulse and then as a debouncer to clean up low-end noises. The motors are both continually sampled and fed into a simple multiplexer made of a 7400 NAND package. These signals are multiplexed into PA7, the pulse accumulator, and then read as desired from IC. The selector line for the multiplexer comes from one of the data pins that was intended to supply an IR LED. The latch design was changed to provide solid-power lines for the multiplexer selector, the "Cyclops" motor, and the LED intensity selector.

There have been several small problems with the system that were not anticipated. The primary problems were with getting the proper resistor and capacitor values to filter the signal somewhat, but still spike when required. Experiments at low speeds showed the resistors were too high to allow the commutator noise to offset the OP-Amp as desired, so these values were adjusted some. Crosstalk between motors was eliminated by adding large capacitors to the board, but the addition of the multiplexer and bus wire between the circuit and the motors added a good deal of noise to the system. As the system stands, it is only used as part of a "last resort" behavior. A stalled motor can be detected, but the values simply aren't accurate enough to base odometry on.

The Motor Sampler Monitoring Daemon generates data on the number of pulses returned by each motor. The interface-level output is a "Stall Factor" that is based on repeated samplings where the motor pulse output is less than should be expected for the current output of the motor PWM system. Both drive motors have stall factors, which are read in as global variables by any behavior that requires that data.

Cyclops Vision Sensor

This sensor is designed to allow the robot to quickly and efficiently gather reflected IR readings like those from the collision avoidance sensors, but in a full 360° circle around the robot and at a rate of better than one full scan per second. The design I have opted for is the "periscope" style implementation, chosen primarily for it's aesthetics and modularity; The sensor is very simple in design and construction and easy to add to an existing robotic platform. Unfortunately, the original design's variable-gearing motor kit was not available, so the current design uses a Futaba servo with the stop gears clipped to turn the mirror. The servo has worked out well, though, since it is small, readily available, and easy to work with. (Figure 4.)

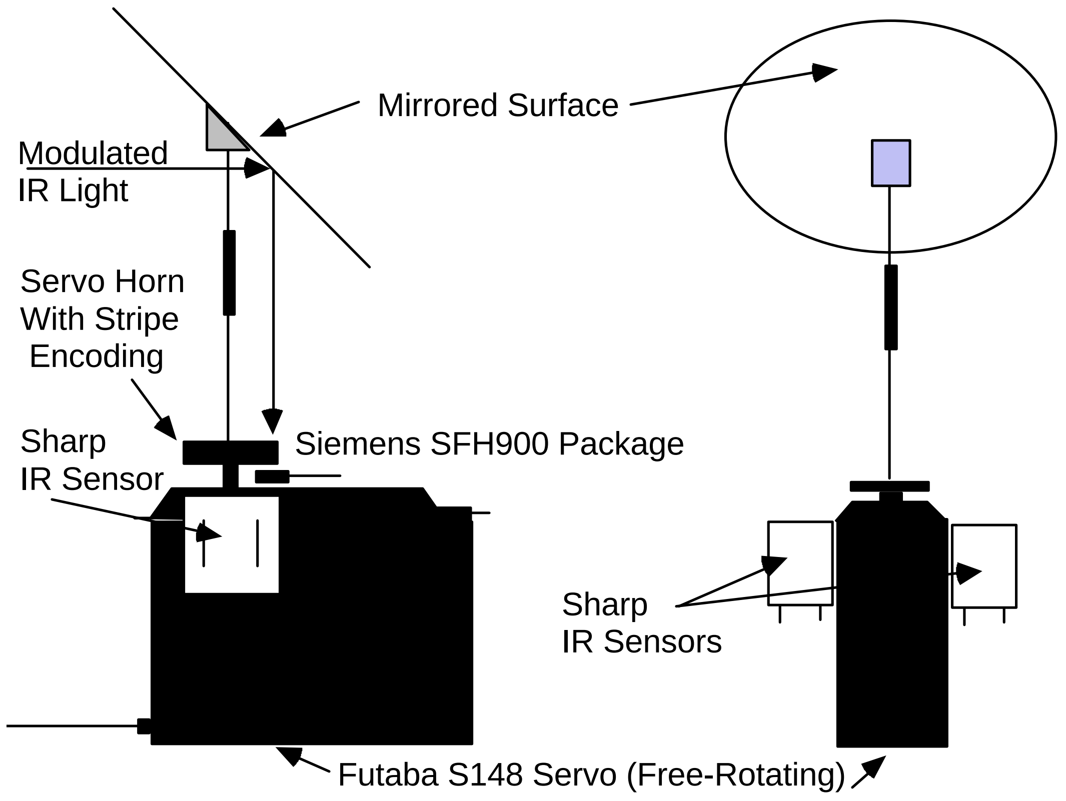

Figure 4.

The mirrored surface rotates, directing the reflected IR into the side-mounted sensors. The Siemens sensor reads a stripe on the servo horn to indicate when the mirror has gone around. This stripe is read to maintain timing and calibration of the servo's speed. A circuit has been designed to select different intensities on two high-mounted banks of LED's to create the modulated IR.

The Sharp IR sensors were hacked to return analog values, just as the base sensors were, but they were also hacked to have a lower time constant so they would react faster to changes in the IR intensity. This meant replacing the internal capacitor with a smaller value. This hack is reported to reduce the range on the sensors, but the IR emitter intensity was increased to compensate.

The eight IR emitters were placed on the perimeter of the dome aiming outward. These LED's had to be modulated at 40KHz and also controlled by the MCU, but power needs were too great for HC logic gates. Transistor switches were added inline with the latch outputs and 40KHz line to provide ample current through the LED's. Preliminary tests practically lit up a darkened room when viewed by a video camera. Resistors in the circuit are socket mounted for easy adjustment.

First impressions indicated that the Cyclops-level emitters would have to be synchronized with the base emitters, which were normally only turned on only for 0.1s durations. Since the Cyclops sensor requires practically constant IR emissions anyhow, it was decided that all of the emitters would remain continuously on. This makes the robot very "intrusive" to other robots, which may be confused by the high-intensity IR bathing the environment, but this side effect cannot be avoided.

The Cyclops sensor's mirror is susceptible to jostling if handled, so the code in CYCLOPS.C was written to help realign the mirror with the Siemens sensor stripe. The motor is pulsed very slowly until the sensor registers. The rotation is so slow because if the motor is shut off from full speed, the inertia of the system causes the mirror to rotate another 30° after shutdown. The motor shaft is thus aligned such that the robot believes the mirror is pointing due forward. The mirror must then be rotated manually into place. Eventually this code could be replaced by an automatic system that compared the returns from the Base IR sensors, though this would probably be unreliable.

The Cyclops sensor is monitored by two separate daemons. The Cyclops Synchronization Daemon monitors the Siemens shaft encoder to determine the rate of rotation and provide synchronization to the other module, the Cyclops Scanning Daemon, which generates return values much like those for the Base-Mounted IR sensors. These values are also in arrays, with each of the elements representing a different position in the rotation. Currently, there are 16 elements to each array, as the 360° sweep is sampled 16 times per revolution.

Behaviors

The first behaviors that were written are simple collision avoidance systems, both using only the basic IR sensors. The first attempt, CASEY.C (using a simple look-and-turn algorithm), was not intended to ever actually be used as a behavior, but only as a demonstration of the robot's sensors. The Dynam.11-based collision avoidance system appears in the appendix as CASEY2.C. The code worked well, but would probably break down when forced to multitask with the processor-intensive Cyclops routines, since it relies on regular sampling intervals. The programs CASEY3.C and CYCLOPS.C are only parts of sensor maintenance routines, and not behaviors themselves.

The first multiple-behavior code was CASEY4.C. This code served as the platform for basic behavior development, as it was designed to be modular. New behavior modules and sensor daemons were added to the structure as they were developed and then tested in place. This code contains the behaviors currently in use, as well as the arbitration system. Though the behavior system is constantly being updated, added to, and tweaked, the following are the three best examples of stable, working behaviors as of this report. These are also the ones seen in the current version of CASEY4.C. Other behaviors include wall following, robot seeking, and object scanning. As of this report, though, these behaviors were either non-functional or very unstable. The robot seeking behavior, for example, requires that the Cyclops sensor enter a "passive" mode. By shutting down IR emitters, it is possible to identify other robots, but this hinders the collision avoidance system. An interprocess communication needs to be established to allow this. Future behaviors, once stabilized, will probably be integrated into a new version of the code with an IR Daemon to allow behaviors to know when the IR system is in passive or active mode.

Bump 'N Go Behavior

This behavior is a last-resort reaction in case of collision or stall. The only input is the motor commutator sensor's stall factors. When the stall factors reach a certain level, the behavior flags the arbitration network to indicate a stalled motor. The flag is in the form of a non-zero value in the "desired motor speed" variable corresponding to the stalled motor, and is equal to 100% in the opposite direction of the current value.

Collision Avoidance Behavior

This behavior is the basic method for avoiding collisions in the environment. It is based solely on the Base IR sensor arrays, and indicates to the arbitration system a general direction in which to travel to avoid collision.

Trapped Behavior

This behavior is based on the Cyclops sensor package, and simply scans the Cyclops return values and compares them to a threshold. If too many of these values exceed the threshold, the robot assumes it is trapped and will alert the arbitration system, which will then shut down the motors and emit a low tone to let the user know of the problem.

Arbitration System

The eventual result of these behaviors is the control of the motors by the Arbitrator. Priority status is given to the "emergency" flags like the Bump 'N Go behavior produces. Barring flags, the robot tends to move forward. The current methods allow the robot to navigate a room with very few collisions, and even fewer "hang-ups" wherein the robot is completely immobilized. (The "Trapped" behavior usually only occurs under forced circumstances, though this would immobilize the robot.) It also seems to find many of the niches in the environment, even if it cannot navigate around in them.

Conclusion

The robot now meets the criteria set by the EEL 5934 syllabus. It has 3 major sensor groups and a behavior system based on those sensors. Although actuation is limited to propulsion in the environment, the robot seems to control this ability well. As for creating a better "Cyclops" sensor, the current design has surpassed some of the limitations of it's predecessor. The Siemens shaft encoder eliminates the need for a synchronizing LED, which created a blind spot in the earlier design. A side effect of this is that the sensors are only sampled as needed, not in a tight , continuous loop as the original system used. This results in far more stable values from the board, as well as lower CPU overhead. The dual-sensor periscope design doesn't have any of the blind spots from the rotation shaft that a normal periscope would have, and is very aesthetically pleasing.

More tests are needed to refine the code as it stands. Most of the coefficients could be replaced by learning algorithms. Some of the code might be best implemented in assembly. A large variety of behaviors could be based on the information provided by the Cyclops sensor, but time has limited such development for this semester. Unfortunately, the hardware end of the project became very time consuming, leaving little time to develop the software to it's fullest potential. Hopefully in the future more complex behaviors can be created. A mapping and navigation routine would be ideal for this sensor set, as would a hunter-seeker routine for chasing other robots (and no doubt terrorizing them with excessive IR). Overall, the robot's design has been very successful for it's basic goals, and shows a great deal of potential for the future.

Documentation

[1] Fred Martin. The 6.270 LEGO Robot Design Competition Course Notes.
Electrical Engineering and Computer Science Department, MIT 1992.
[2] Joseph L. Jones and Anita M. Flynn. Mobile Robots: Inspiration to Implementation.
A. K. Peters 1993.
[3] Pattie Maes. Designing Autonomous Agents.
MIT Press / Elsevier Science Publishers 1990.
[4] M68HC11 Reference Manual.
Motorola 1991.
[5] MC68HC11E9 Technical Data.
Motorola 1991.
[6] Hugh Kenner. Geodesic Math and How To Use It.
University of California Press 1976.
[7] John Prenis. The Dome Builder's Handbook.
Running Press 1973.

Appendices

CASEY.C

```
/*  casey.c

    Attempt at collision avoidance

    Casey Barker
*/

int LEFT =      0;          /* Sensor input values */
int RIGHT =     0;
int FRONTLEFT = 0;
int FRONTRIGHT= 0;
int LTOR =      0;
int RTOL =      0;
int THRESH =    110;            /* Motor init values */
int LMOTOR =    0;
int RMOTOR =    1;
int NORM =      80;
int STOP =      0;


void irscan()
{
 while (1) {
  poke(0x7000, 0xff);
  wait(100);
  LEFT = analog(5);
  RIGHT = analog(4);
  FRONTLEFT = analog(3);
  FRONTRIGHT = analog(0);
  LTOR = analog(2);
  RTOL = analog(1);
  poke(0x7000, 0xff);
  wait(100);
 }
}


void wait(int milli_seconds)    /* Don't busy wait, check timer */
{
  long timer_a;
  timer_a = mseconds() + (long) milli_seconds;
  while(timer_a > mseconds()) {
    defer();
  }
}


void go_right()              /* if object is on left side of robot */
{
   motor(LMOTOR,NORM);
   motor(RMOTOR,STOP);
}

void go_left()               /* if object is on right side of robot */
{
   motor(RMOTOR,NORM);
   motor(LMOTOR,STOP);
}

void go_forward()            /* if no objects detected */
{
   motor(LMOTOR,NORM);
   motor(RMOTOR,NORM);
}

void go_back()               /* if objects in front of robot */
{
   motor(LMOTOR,-1*NORM/2);
   motor(RMOTOR,-1*NORM/2);
}


void main()
{
  beep();
  start_process (irscan());
  while (1) {
         if ((FRONTLEFT > THRESH && FRONTRIGHT >THRESH) || (LTOR >
THRESH && RTOL > THRESH)){
                  go_back();
                  wait (100);
                  go_right();
                  wait (400);
```

```
        }
        else if (LEFT > THRESH | | FRONTLEFT > THRESH | | RTOL >
THRESH) {
                go_right();
        }
        else if (RIGHT > THRESH | | FRONTRIGHT > THRESH | | LTOR >
THRESH) {
                go_left();
        }
        else {go_forward();}
  }
}
```

CASEY2.C

```
/*  casey2.c

    Attempt at collision avoidance using Non-Linear Dynamics

    Casey Barker
*/

int LEFT =      0;          /* Sensor input values */
int RIGHT =     0;
int FLEFT = 0;
int FRIGHT= 0;
int LTOR =      0;
int RTOL =      0;

int LEFTO,RIGHTO,FLEFTO,FRIGHTO,LTORO,RTOLO;


float ABSPEEDR;
float ABSPEEDL;
float SPEEDL=100;
float SPEEDR=100;

float LETFD,RIGHTD,FLEFTD,FRIGHTD,LTORD,RTOLD;
float
ABSLEFTD,ABSRIGHTD,ABSFLEFTD,ABSFRIGHTD,ABSLTORD,ABSRTOLD;

float ca=0.015;
float cb=0.5;
float cd=50;


void irscan()
{
 while (1) {
  poke(0x7000, 0xff);

  RIGHTO=RIGHT;
  LEFTO=LEFT;
  FRIGHTO=FRIGHT;
  FLEFTO=FLEFT;
  LTORO=LTOR;
```

```
RTOLO=RTOL;


wait(90);
LEFT = analog(5);
RIGHT = analog(4);
FRONTLEFT = analog(3);
FRONTRIGHT = analog(0);
LTOR = analog(2);
RTOL = analog(1);
poke(0x7000, 0xff);

RIGHTD=(float)(RIGHTO-RIGHT);
LEFTD=(float)(LEFTO-LEFT);
RTOLD=(float)(RTOLO-RTOL);
LTORD=(float)(LTORO-LTOR);
FLEFTD=(float)(FLEFTO-FLEFT);
FRIGHTD=(float)(FRIGHTO-FRIGHT);

ABSRIGHTD=RIGHTD;
if(RIGHTD<0.0) ABSRIGHTD=-RIGHTD;
ABSLEFTD=LEFTD;
if(LEFTD<0.0) ABSLEFTD=-LEFTD;
ABSFRIGHTD=FRIGHTD;
if(FRIGHTD<0.0) ABSFRIGHTD=-FRIGHTD;
ABSFLEFTD=FLEFTD;
if(FLEFTD<0.0) ABSFLEFTD=-FLEFTD;
ABSRTOLD=RTOLD;
if(RTOLD<0.0) ABSRTOLD=-RTOLD;
ABSLTORD=LTORD;
if(LTORD<0.0) ABSLTORD=-LTORD;

ABSPEEDR=SPEEDR;
if(SPEEDR<0.0) ABSPEEDR=-SPEEDR;
ABSPEEDL=SPEEDL;
if(SPEEDL<0.0) ABSPEEDL=-SPEEDL;

SPEEDL=SPEEDL + ca*ABSPEEDL + cb*ABSLEFTD + cd*LEFTD;
SPEEDR=SPEEDR + ca*ABSPEEDR - cb*ABSRIGHTD + cd*RIGHTD;

if(SPEEDR>100.0)
  SPEEDR=100.0;
if(SPEEDL>100.0)
  SPEEDL=100.0;
  if(SPEEDR<-100.0)
    SPEEDR=-100.0;
  if(SPEEDL<-100.0)
    SPEEDL=-100.0;

  motor(0,SPEEDL);
  motor(1,SPEEDR);

  wait(90);

 }
}


void wait(int milli_seconds)    /* Don't busy wait, check timer */
{
  long timer_a;
  timer_a = mseconds() + (long) milli_seconds;
  while(timer_a > mseconds()) {
    defer();
  }
}



void main()
{
  beep();
  start_process (irscan());

}
```

CASEY3.C

```
 /*
Motor Sensor Routines
*/
void wait (int millisec) {
  long timer_a;
  timer_a = mseconds() + (long) millisec;
  while (timer_a > mseconds()) {
    defer();
    }}

void countinit () {
  poke(0x1026,0x40);
  }

int duration;

int count (int duration)
 {
 poke(0x1027,0x00);
 wait(duration);
 return (peek(0x1027));
 }
```

CYCLOPS.C

```
/*     cyclops.c
       Manual Calibrator Routine
       Casey Barker
*/
/*  ***Wait Function*** */

void wait(int milli_seconds)
 {
  long timer_a;
  timer_a = mseconds() + (long) milli_seconds;
  while(timer_a > mseconds()) {defer();}
 }

void main ()
{
    while ((peek(0x1000)&0x01)==0)
       {
          poke(0x7000,0x80);
          wait (1);
          poke(0x7000,0x00);
          wait (20);
       }
    beep();
}
```

CASEY4.C

```
/*  casey4.c

   First Integrated Behavior Platform.

   Casey Barker
*/


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


int irscan_pid;
int motor_count_pid;
int motor_driver_pid;
int cysynch_pid;
int cyscan_pid;

int trapped_pid;
int coll_av_pid;
int bumpngo_pid;
int arbitrate_pid;


int l_motorid=1;
int r_motorid=0;

int l_motor = 0;
int r_motor = 0;

int l_motor_desired = 0;        /*Arbitrated Values*/
int r_motor_desired = 0;        /*These are the ones the Motor Driver reacts to*/

int bg_l_motor_desired = 0;     /*Bump 'N Go Values*/
int bg_r_motor_desired = 0;

int ca_l_motor_desired = 0;     /*Collision Avoidance Values*/
int ca_r_motor_desired = 0;

int rightside=4;    /*IR Sensor Analog Port Values*/
int leftside=5;
int leftfront=3;
int rightfront=0;
int leftdiag=2;
int rightdiag=1;
int rcyclops=6;
int lcyclops=7;

int base[6];        /*Base-Mounted IR Sensor Array (Current Values)*/
int base_old[6];    /*Base-Mounted IR Sensor Array (Previous Values)*/
int base_delta[6];  /*Base-Mounted IR Sensor Array (Delta from Last Values)*/
int base_factor[6]; /*Calculated Collision Potential*/

int cyclops[16];             /*Cyclops Sensor Data Array (Current Values)*/
int cyclops_old[16];         /*Cyclops Sensor Data Array (Previous Values)*/
int cyclops_delta[16];        /*Cyclops Sensor Data Array (Delta from Last
Values)*/
int cyclops_factor[16];       /*Calculated Object / Robot Potential*/
int cyclops_count;           /*The Current position in the Cyclops array*/

int thresh = 10;             /*Threshhold for lowest factor responded to*/
int lower_limit = 80;        /*Lowest Probable IR value, subtracted from current IR
reading for factor calculation*/

float motordelta=0.05;       /*Motor Adjustment Coefficient*/

int pa_duration=100;         /*Pulse Acc. Wait Time for Motor Sensor*/
int left_count;             /*Motor Count for Left Motor*/
int right_count;             /*Motor Count for Right Motor*/
int left_sf;               /*Left "Stall Factor" (0-3)*/
int right_sf;               /*Right "Stall Factor" (0-3)*/
float pulsefactor=1.0;       /*Multiplied by current motor speed to determine what
ideal pulse return should be*/

long cyperiod;               /*Total Rotational Time*/
long cydelta;               /*Time Slice between Reads*/
long oldcydelta;             /*Previous Time Slice*/
long time;                 /*Time of Siemens Hit*/
long prevtime;               /*Previous Time of Siemens Hit*/
long icydelta = 40L;          /*Guess of what cydelta should be*/
int goodrun;                /*Is the current pass good?*/

int trap_flag;               /*Am I Trapped?*/
int trapthresh = 93;          /*Trapped Threshhold*/
int trap_count;
int port;                  /*Current Value of 0x7000 MMIO port)*/

/*  ************************************************  */
/*  *******************Functions*******************  */
/*  ************************************************  */


/*  ***Wait Function***  */
void wait(int milli_seconds)
 {
  long timer_a = mseconds() + (long) milli_seconds;
  while(timer_a > mseconds()) {defer();}
 }


/*  ***Float Wait Function***  */
void fwait(float milli_seconds)
 {
  long timer_a = mseconds() + (long) milli_seconds;
  while(timer_a > mseconds()) {defer();}
 }

/*  ***Absolute Value Function***  */

int abs(int n)
{
if (n>0) return n;
if (n<0) return -n;
else return 0;
}

/*  ***Pulse Accumulator Initialization Function***  */
void pa_init ()
 {
  poke(0x1026,0x40);
 }


/*  ***Motor Smoothing Function***  */
void motor_smooth()
{
   l_motor = (int) ((float) l_motor +  (motordelta * (float) (l_motor_desired -
l_motor)));
```

```
   r_motor = (int) ((float) r_motor +  (motordelta * (float) (r_motor_desired -
r_motor)));
}


/*  *************************************************  */
/*  ***********Sensor & Monitoring Daemons***********  */
/*  *************************************************  */

/*  ***Motor Sensor Daemon***  */
void motor_count()
{
   while(1)
   {
      port = (port | 0x40);
      poke(0x7000,port);
      poke (0x1027,0x00);
      wait (pa_duration);
      left_count = peek(0x1027);
      if ((left_count< (int) ((float) l_motor * pulsefactor))&&((abs(l_motor))>15))
      {
         if (left_sf<3)
         {
            left_sf = left_sf +1;
         }
      }
      else
      {
         if (left_sf>0)
         {
            left_sf = left_sf -1;
         }
      }

      port = (port - (port & 0x40));
      poke(0x7000,port);
      poke (0x1027,0x00);
      wait (pa_duration);
      right_count = peek(0x1027);
      if ((right_count< (int) ((float) r_motor * pulsefactor))&&((abs(r_motor))>15))
      {
         if (right_sf<3)
         {
            right_sf = right_sf +1;
            }
        }
        else
        {
            if (right_sf>0)
            {
                right_sf = right_sf -1;
            }
        }
    }
}



/* ***Motor Driving Daemon*** */
void motor_driver()
{
    while(1)
    {
        motor_smooth();
        motor(l_motorid,l_motor);
        motor(r_motorid,r_motor);
    }
}

/* ***Cyclops IR Scanning Daemon*** */
void cyscan()
{
    int n=cyclops_count;

    while(1)
    {
        if ((n != cyclops_count)&&(goodrun==1))
        {
            n=cyclops_count;
            cyclops_old[n] = cyclops[n];
            cyclops[n]=((analog(lcyclops) + analog(rcyclops))/2);
            cyclops_delta[n] = cyclops[n] - cyclops_old[n];
            cyclops_factor[n] = (cyclops[n]-lower_limit) * cyclops_delta[n];
        }
        else defer();
    }
}
```

```
/*  ***Cyclops Synchronizer Daemon***  */
void cysynch()
{
time = mseconds();
cydelta = icydelta;
while (1)
   {
      if (peek(0x1000) & 1)
      {
         goodrun = 1;
         prevtime = time;
         time = mseconds();
         cyperiod = time - prevtime;
         oldcydelta = cydelta;
         cydelta = ((long) (((float) cyperiod  /  16.0) * (1.0 / 3.0)) + (long) ((float)
oldcydelta * (2.0 / 3.0)));
         start_process (tone(500.0, 0.05));
         if (cydelta > (long) (1.5 * (float) icydelta))
         {
            cydelta = oldcydelta;
            goodrun = 0;
            wait(300);
         }
         else
         {
            cyclops_count=0;
            lwait (cydelta - 2L);

            cyclops_count=1;
            while ((cyclops_count<15)&&(((peek(0x1000))&0x01) != 1))
            {
               lwait (cydelta);
               cyclops_count = cyclops_count + 1;
            }
         }
         cyclops_count = 15;
      }
   }
}

/*  ***Long Wait Function (SPECIFICALLY for CYCLOPS Synchronization!)***
*/
void lwait(long milli_seconds)
 {
  long timer_a = mseconds() + milli_seconds;
  while((timer_a > mseconds())&&((((peek(0x1000))&0x01) !=
1)||(cyclops_count<2))) {defer();} /* Doesn't wait if Siemens Sensor goes high */
 }



/*  ***Baseline IR Scanning Daemon***  */
void irscan()
{

int n;

while (1)
   {
   for (n=0; n<6; n++)
      {
      base_old[n] = base[n];        /* Stores old data */
      }

   base[rightside]  = analog(rightside);
   base[leftside]   = analog(leftside);
   base[rightfront] = analog(rightfront);
   base[leftfront]  = analog(leftfront);
   base[rightdiag]  = analog(rightdiag);
   base[leftdiag]   = analog(leftdiag);

   for (n=0; n<6; n++)
      {
      base_delta[n] = base[n] - base_old[n];   /* Calculates Change in IR */
      }

   for (n=0; n<6; n++)
      {
      base_factor[n] = (base[n]-lower_limit) * base_delta[n];   /* Calculates Object
Collision Factor */
      }
   wait(80);
   }
}

/*  ************************************************  */
/*  ********************Behaviors********************  */
/*  ************************************************  */
/*  ***Bump 'N Go Behavior***  */

void bumpngo()
{

while (1)
{
   if (left_sf>1)
   {
      if (l_motor>0)
      {
         bg_l_motor_desired = -100;
      }
      if (l_motor<0)
      {
         bg_l_motor_desired = 100;
      }
   }
   else
   {
      bg_l_motor_desired = 0;
   }

   if (right_sf>1)
   {
      if (r_motor>0)
      {
         bg_r_motor_desired = -100;
      }
      if (r_motor<0)
      {
         bg_r_motor_desired = 100;
      }
   }
   else
   {
      bg_r_motor_desired = 0;
   }

}

}
```

```
/*  ***Collision Avoidance Behavior***  */

void coll_av()
{
int largest, sensor, n;

while (1)
{
   largest = 0;
   sensor = 0;
   for (n=0; n<6; n++)
   {
      if (largest<base_factor[n])
      {
         largest = base_factor[n];
         sensor = n;
      }
   }

   if (largest<thresh)
      {
         sensor = 6;
         ca_l_motor_desired = 100;
         ca_r_motor_desired = 100;
      }
   if (sensor==rightside)
      {
         ca_l_motor_desired = 50;
         ca_r_motor_desired = 100;
      }
   if (sensor==leftside)
      {
         ca_l_motor_desired = 100;
         ca_r_motor_desired = 50;
      }
   if (sensor==rightfront)
      {
         ca_l_motor_desired = -100;
         ca_r_motor_desired = 100;
      }
   if (sensor==leftfront)
      {
         ca_l_motor_desired = 100;
```

```
            ca_r_motor_desired = -100;
        }
    if (sensor==rightdiag)
        {
            ca_l_motor_desired = -50;
            ca_r_motor_desired = -100;
        }
    if (sensor==leftdiag)
        {
            ca_l_motor_desired = -100;
            ca_r_motor_desired = -50;
        }
    defer();

    }
}

/*  ***Shutdown if Trapped Behavior***  */

void trapped()
{
int n;

while (1)
{
    trap_count = 0;

    wait (1000);

    for (n=0; n<16; n++)
        {
            if (cyclops[n] > trapthresh)
                {
                    trap_count = trap_count + 1;
                }
        }
    if (trap_count > 14)
        {
            trap_flag = 1;
        }
    else
        {
            trap_flag = 0;
        }
```

```
}
}

/*  ***Robot Detection & Tracking Behavior***  */

/*  ***Behavior Arbitration***  */

void arbitrate ()
{
while (1)
{
   if (trap_flag == 1)
      {
         l_motor_desired = r_motor_desired = 0;
         tone (300.0, 0.5);
      }
   else
      {
         if (bg_l_motor_desired != 0)
            {
               l_motor_desired = bg_l_motor_desired;
               wait (1000);
            }
         else
            {
            l_motor_desired = ca_l_motor_desired;
            }

        if (bg_r_motor_desired != 0)
        {
            r_motor_desired = bg_r_motor_desired;
            wait (1000);
        }
        else
        {
            r_motor_desired = ca_r_motor_desired;
        }
      }
}

}

/*  **************************************************  */
/*  ******************Main Function*****************  */
/*  *************************************************  */

void main()
{
    beep();
    port = 0b10001111;
    poke(0x7000,port);
    pa_init ();

    cysynch_pid        = start_process (cysynch());
    motor_driver_pid   = start_process (motor_driver());
    motor_count_pid    = start_process (motor_count());
    irscan_pid         = start_process (irscan());
    cyscan_pid         = start_process (cyscan());

    bumpngo_pid        = start_process (bumpngo());
    coll_av_pid        = start_process (coll_av());
    trapped_pid        = start_process (trapped());

    wait(2000);        /*Lets System Stabilize*/

    tone(1000.0, 0.5);
    arbitrate_pid      = start_process (arbitrate());

}
```