University of Florida

Department of Electrical and Computer Engineering

EEL 5666 - Intelligent Machine Design Laboratory

Final Report - Lucid

Student Name: Eugenio Jarosiewicz

Date: 8/4/99

TAs: Scott Jantz

Patrick O'Malley

Instructors: A. Antonio Arroyo

E. M. Schwartz

TABLE OF CONTENTS

ABSTRACT

One day the paradigm shift to practical use of robotics, along with other technologies akin to it such as artificial intelligence, will eventually take place. Robotics has already achieved an established function in situations where the environment is invariable and the tasks are routine, such as assembly line automation. The purpose of this project was to create and research an autonomous mobile robotic agent (herein ‘robot’) which could interact intelligently in a dynamic environment such as a home and which could execute at least one useful purpose, beyond the intrinsic behaviors necessary to keep it from disturbing or in other ways interfering negatively with its environment.

EXECUTIVE SUMMARY

Lucid was created to carry out the humble task of being a sentry that roams between rooms and turns off lights in unoccupied rooms to conserve electricity. I was particularly interested in the programming involved to control the robot - in essence, its 'intelligence'. Hardware design was meant to be a non-issue, as was the interfacing of sensors.

Lucid was constructed on the Mekatronix Talrik II platform. I began the platform with an arrangement of 10 bump sensors and 4 IR emitters/detectors, with plans of adding more IR pairs later. I only later fully realized that the approach to writing the software to control the robot is heavily dependent on the sensor suite available. One insight I had on this was how each sensor is a unique entity, and I had wished to apply a more object-oriented design to the sensory perception code.

Lucid was designed to be a prototype robot in the form it would turn off lights, which was to be a bump switch located at floor level in front of a landmark it could identify. Designing a mechanical device to reach for a light switch and a vision system to locate it on a wall would have been beyond the scope of the project. Past midway through the semester I decided to investigate a different method for my robot to control lights, via a technology called X10. I ended up interfacing an X10 module designed for a PC serial port to the M68HC11 and writing an interrupt handler for it. This deviated me from researching navigation more, but provided me with a lot of experience.

INTRODUCTION

In various forms, both machines and computational devices have been around for centuries. At the merger between the fields of computers and machines is robotics as it stands today. Although it is not a new field, it is still a dawning area and has still not achieved a practical or household use like computers. However, it was not until relatively recently that computers have become widely prevalent and along with the internet finally started the paradigm shift that is being dubbed “The Information Revolution”. With this in mind, I believe that not far from now in the foreseeable future, robotics and other uses of artificial intelligence will allow people to delegate tasks to intelligent machines which they would have otherwise have had to spend their time supervising or administering.

As such, my goal for this project was to create a robot which could navigate between rooms, sense for presence of people or other occupants in the room, and turn off the lights in unoccupied rooms. This paper describes the robot and provides details on the integration of the system platform, actuation, sensors, and behaviors. It also discusses the experiments conducted and results achieved, and of course, any hard, interesting, and embarrassing lessons learned.

INTEGRATED SYSTEM

I developed my robot on the Talrik II platform. It was chosen because of its large sensor expansions and room for even more expansion. I equipped it with the standard bump sensors and 4 IR sensors. As previously explained, I planned on adding more IR's later, but one of the tough lessons I learned was that with the current highly imperative programming style sensor additions don't necessarily scale. I will explain this in detail later. Three ground-facing 'nose' CDS cells were originally meant to follow a line to the location of the light switches, and three 'eye' CDS cells facing forward and upward at approximately 45 degrees were used to detect the light levels in the room. A microphone was used to detect presence of people in a room (given some assumptions that people were awake, etc.). I looked into motion sensors as well, but chose not to use them. Eventually, I added and interfaced an X10 PC serial port module to the microprocessor board and coded the basic functionality to control a light via the protocol defined by the company that produces the X10 modules. This will likewise be covered in more detail shortly. Finally, I added a small piezo buzzer to provide some feedback for testing and debugging purposes, and to sound cheerful in general.

For the software architecture, I began with a classical finite state machine (FSM) approach, mostly because ICC11v5 did not support multitasking. I soon disliked it enough to the point where I was about to switch to IC, when I learned that ICC11v4 had a

multitasking library, which I promptly combined into my code to re-design into a subsumption architecture. This allowed me to modularize my code much more cleanly, and still leave an expandable base to add onto. Many of the benefits of a subsumption architecture have been already discussed, my intent is only to state the opinion that parallel computation enable more complex behaviors to be formed more readily. While such parallel computation can be simulated by questionably clever code in a FSM, it does not lend itself well to modification or understanding.

MOBILE PLATFORM

As I have already mentioned, Lucid was developed on the Mekatronix Talrik II platform. As it is a familiar platform to the people of the Machine Intelligence Lab (MIL) at the University of Florida (UF), I will not go into great detail about it. For people not familiar with it, more information in the form of hyperlinks (or plain url's depending on the format one is reading this) will be available in the references section.

The objectives of this platform choice were to be expandable, and as such, it was slightly large – specifically 10 inches in diameter and 7 inches high. While size is relative and therefore difficult to ascertain a 'good' size for an application, I believe a general rule of thumb is that anything more than necessary is, well, unnecessary. Near the end of the project when I was coding it became apparent to me that with the sensors I had on it I could have chosen a smaller size. As the robot as navigating, the robot's bridge level would occasionally collide with or get stuck on chairs, tables, or other obstacles without triggering any sensor. The best solution to this without adding any sensors would be to have a timeout condition in the code, as suggested in the 6.270 text.

I believe with a ‘certain certainty’ that with the progress of nanotechnology, a bug-sized or even smaller robot will easily be appropriate for such a task. Nevertheless, I am glad with the platform because it does allow me the freedom of expansion in the future without adding more overhead, figuratively and literally.

ACTUATION

The suggested actuators for locomotion in the Talrik II platform are two servos hacked into gearhead motors, which are what I used. The only other actuation on my robot was the X10 PC serial port module, which was used to turn the lights on/off. Since this was a large part of my project, I will provide a short description of X10, followed by specifics about my implementation.

X10 is a technology developed around the 1980’s that can be used to control other electrical devices in a building. The method in which X10 works to control other devices is by sending a low signal over the present wiring in a building to other devices that are X10 aware, ie., listening for such a signal. Part of the technology of X10 is the protocol they have developed communicate via the signals. By sending the proper bit pattern to a X10 communication device, it will specify whether to turn on or off any one of up to 256 devices, to brighten or dim them (for lights), or get status from more advanced modules. Typically one controls X10 via a handheld remote control similar to one for a TV or VCR. The remote sends a radio signal over radio frequencies to a transmitter that is plugged into a wall outlet. This transmitter receives the information, and sends the signal over the wiring of the building via the outlet it is plugged into. There also exists software

that allows one to do this from a computer, via a module that plugs into a PC serial port. This is the piece of equipment (that is marketed under the name ‘Firecracker’) which I added to my robot and interfaced to the microprocessor.

Two things were necessary to communicate through the Firecracker module. The first was to send it the proper signals on two pins on a serial port interface. For this, I purchased a DB9 male connector which I could solder to, and plug the Firecracker into. Specifically, the two pins were Data-Transmit-Ready and Ready-To-Send (DTR and RTS, pins 4 and 7 on a DB9 connector, respectively). The Firecracker also draws its power from the current on one of these two pins, so the protocol for it specifies that one of the two must be high the whole time or otherwise the Firecracker unit will reset itself and the signal must be restarted. Each bit sent to the Firecracker is a combination of one of the two pins high and the other low. Both pins set to high means standby. There was also a specific timing involved. Each bit had to be a minimum of half a millisecond long, separated by another delay period of the same time. In the code, each bit signal is one ms. long, followed by another ms. wait time, making the duration of each command last approximately 80 ms. It was for this reason that I had to write an interrupt driver. Since I have not taken Microprocessor Applications or do not know much about the M68HC11 before coming here, I used a OC line to interrupt, and was using two multiplexed outputs for my pins. If I knew more about the MC, I would have tried to use the parallel subsystem (SPI). I could not use the built-in serial subsystem (SCI) because it adheres to the RS232 standards and sends start and stop bits (to the best of my knowledge).Separately, the X10 protocol defines each command as a 40 bit signal. There

is a 16 bit header, a 16 bit command section, and an 8 bit trailer. For the specifics of the protocol, please see the references.

SENSORS

The sensors suite on Lucid consists of 10 bump sensrs, 4 IR emitter/detector pairs, 6 CDS cells, and one microphone. The bump sensors where used for collision detection, five in front and five in back. Through use of different resistors and clever wiring, each of the front and three of the rear bump sensors gave a different value, so it was possible to detect from which direction the bump was detected. The general algorithm was to turn away from that direction and move away if anyone was ever activated.

The IR pairs were used for collision avoidance. I had three forward facing ones, and one rear facing one. Two of the three front IR's are mounted outward to detect obstacles at its sides. The IR's were model Sharp GP1U58, hacked into yielding an analog value instead of a digital one. As I mentioned previously, I began with only four of them and planned on adding more later, but after I began coding I realized that the way I was developing the code, the present set of sensors was very intrinsic to the understanding of direction in the robot. Although I did not pursue this due to time constraints, I wanted to research the

possibility of applying object-oriented programming principles to this issue, and thus abstracting the sensors into a separate object. I believe that this would create a much less direct dependence between the navigation code and the sensor code, thus reducing the interaction and increasing the flexibility of each.

One positive thing I did achieve with the IR sensors was to code a self-calibrating functionality for them based largely on fuzzy logic, instead of having certain arbitrary values hard-written in the code. I took a similar approach to the students who were working on the swarm of robots, which was to keep track of the high and low values of the sensors. I do not know how they implemented threshold arbitration in them, but from experimentation I found that a reading of 20 percent of the difference between the high and low readings above the average low was a good threshold.

[Threshold = (high_avg – low_avg) *.20 + low_avg]

One interesting phenomenon came about this, I will explain it along with the behaviors below.

My robot was also equipped with two groups of CDS. The CDS cells were used to detect levels of light in a room, and I used pre-determined values as thresholds. I experimented with having a similar calibration functionality as the IR's, but I found that the readings I needed from them was slightly dependent on the room the robot was in (ie., a bright room vs. a dark room), so it made it a more transient value to track and consider.

Finally, I had several incarnations of my microphone circuit before I used the one on p. 143 of the Mobile Robots text. This final circuit gave me a range of at least 10 ft. for a handclap (granted, a rather inaccurate measure, but I did not have access to equipment to measure amplitude in decibels vs. voltage response). I also used a pre-determined

threshold for the microphone, partially for the same reasons as the CDS, in which it was very varying levels, and secondly, because microphone circuit's sensitivity, it would actually saturate and return lower than expected values for a large amount of noise. The values I have are still not the best (the best would be self-calibrated of course), but merely experimental.

BEHAVIORS with EXPERIMENTS & RESULTS

Since I developed the behaviors by a process of experimentation and research, this section also deals with those topics and the results of the experiements. The behaviors on my robot are stacked in a hierarchy to fit into a subsumption architecture. None of the behaviors actually send commands to the motors, but instead let an 'omniscient behavior function' (OBF) arbitrate between the desired commands, since all the behaviors are running in parallel. Likewise, I created an omniscient motor function which takes a desired motor command and adjusts the speed of the each motor separately so as to not wear out the motors by going from the maximum value in one direction to the maximum in the opposite direction. I used a constant acceleration value to implement this.

I modeled the subsumption architecture from the examples in class notes and in the Mobile Robots text. I assigned priorities to each behavior so that I could have intelligent arbitration of movement. The lowest priority behavior the robot has is a default

‘curiosity’ / cruise task that causes the robot to continue forward if nothing else is interesting. In between this and the next behaviors I planned on implementing a light and/or line follow behaviors, but I ended up not having time for them. Next up is the avoid obstacles behavior. If any of the IR’s exceed their threshold levels, the robot turns away from them. Overriding the obstacle avoidance is the collision detection behavior. As described in the sensor part of this report, the robot is able to determine from which direction a collision was triggered, and thus can turn and move away from that direction. Finally, the microphone listening behavior is given top priority. It is this behavior that also determines whether or not the lights in the room are on or not.

It is difficult to specify how many behaviors my robot contains. I say this due to the fact that through my experimentation, I noticed the formation of two emergent behaviors that were not explicitly coded in. The first of these was a behavior that overrode the default go forward behavior when the robot is first activated and caused it to run away backwards. This was due to the way the IR calibration routine was coded. Since it tracks the high and low values constantly and uses a predetermined percent difference for a threshold, when the robot is first activated the difference and therefore the threshold is minimal. The robot goes backwards until it hits something, and then when it turns, the IR’s eventually pick up some other object near, and ‘learn’ a wider range of values, at which point the robot is not afraid to go forward because it knows that it can sense so far forward/near.

A second emergent behavior that sporadically came up was wall following and corner negotiation. This behavior was even less predicted than the previous, but after analyzing it I understood why it occurred. I designed my navigation and obstacle avoidance code to

try and navigate a room counterclockwise. This was modeled after the algorithm that computers use to solve maze-type puzzles, similar to the Windoze screensaver 3D maze. I noticed that when coming up to a concave corner my robot would either turn in the opposite direction, or slowly navigate the corner correctly by compensating it's direction in small increments away from it in a way that it made a smooth arc close to the corner. This behavior would only happen on occasion, depending on what values at run-time the IR sensors happened to get calibrated to. If I had a way of noticing such a condition via code, or getting the robot to save the values in non-volatile storage, I could have created a more sophisticated long-term learning mechanism.

CONCLUSION

I was able to develop a robot that could wander 'around' and turn off lights based on certain criteria. There are unfortunately many limitations to the robot, many dealing with lack of sensory input, and all with varying degree of effect on the robot. As previously described, the robot can easily get stuck on things that are too high above or too low beneath its IR sensors. Likewise, the robot succumbs to many false positves and occasionally non-triggered responses to the presence of people. The timing of the X10 signal is critical and occasionally is interrupted, causing a fatal disruption in the message. Nearly all of these things can be improved, most with time, and some with money.

I had planned on giving my robot much more intelligent navigation than it currently possesses, but time factors and late additions to my robot precluded me from doing so. Although I successfully got my robot to do its originally intended purpose, I wish I had more time to research navigation. Fortunately with the software architecture in place, to add more cognitive reasoning and functioning, all that is necessary is to create a new task/behavior that monitors the states of the lower behaviors, and arbitrates behaviors accordingly. This would be a meta-behavior, which of course could eventually superceded by more behaviors. From the hardware and electronics standpoint, I gained more experience and knowledge than in any other class I've taken. The only thing I would change is to make the robot as small as possible, sacrificing expandability for size.

APPENDIX A - REFERENCES

BOOKS

Varela, Francisco J. and Paul Bourgine, editors. Towards a Practice of Autonomous Systems, proceedings from the 5th International Conference on Autonomous Agents

Jones, Joseph L., et al. Mobile Robots – Inspiration to Implementation, Second Edition. A K Peters. Natick, MA. 1999

Martin, Fred. The 6.270 Robot Builders Guide. 1992

STORES – PARTS & PRICES (approx,)

• NiCad Rechargeable Batteries $8.00/4-pack Wal-Mart

• LM386 Op-Amp $1.50 Electronics Plus, Gainesville, FL.

• Condenser Microphone Element $1.50 Radio Shack Part # 270-090C

• Piezo Buzzer, 3-20 VDC, 2.7 Khz $2.99 Radio Shack Part # 273-059

• 9-Position Male D-Subminiature Connector $1.00 Radio Shack Part # 276 1537C

• X10 units $6.00 (special) X10, Inc. http://www.x10.com

• All other parts for my robot where purchased from Mekatronix http://www.mekatronix.com through Scott Jantz scott@mil.ufl.edu , or acquired from the lab stocks

APPENDIX B - ERRATA

• Notes about X10

Each day on their web page they have a “special”. They are trying to get people to sample technology, with the hopes that they will buy more of their products. Be careful in what you buy and how much you pay for it. The parts I received were

- CK17A 2pc rf FireCracker Interface Kit
- LM465 Lamp Module
- A remote, which is optional for this application

• ftp://ftp.x10.com/pub/cm17a_proto.txt contains all the x10 protocol information

• Thanks to everyone who helped me with the project

• I left comments in my code wherever useful, and I did not delete bad code that I've commented out so people can learn from my mistakes

APPENDIX C – CODE

```
/* lucid9.h
 * Eugenio Jarosiewicz
 * EEL5666 - IMDL - Intelligent Machines Design Lab
 * Summer 1999, University Of Florida
 */

#ifndef _LUCID_H_
#define _LUCID_H_

#define DEBUG /*Comment out to prevent debug code from being compiled*/

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

```
#include <mtask.h>/*for mtask routines from icc11v4*/
#include <tkbase.h> /*for rest of tk routines*/


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

/***
HACK AREA read: "temporary/test area" ;-)
***/
#define MOTOR_DESIRED_OVERRIDE motortk(LEFT_MOTOR,
desired_left_motor); motortk(RIGHT_MOTOR, desired_right_motor);

/***
Useful Macros
****/
#define max(A,B) ((A) > (B) ? (A) : (B))
#define min(A,B) ((A) < (B) ? (A) : (B))
#define abs(A) ((A) >= 0 ? (A) : (-(A)))
#define DO2X(A) A A

/***
useful for printing
***/
#define CLEAR "\x1b\x5B\x32\x4A\x04" /* clear screen */
#define HOME "\x1b[1;1H" /* Home cursor */


/***
task stuff
***/
#define TASK_INTERVAL_TIME 5
//#define TASK_STACK_SIZE 1024
//#define TASK_STACK_SIZE 1536
#define TASK_STACK_SIZE 2048

/***
sensor stuph
***/

/*** bumper ***/
#define BUMPER_FUZZY_ZERO 12 /* Noise immunity for bumper readings, anything
below this is noise*/
#define BUMPER_FUZZY_RANGE 3 /*value which the bumper readings might vary
by*/
```

```
/* These are the values I get when the bumpers are hit. Note that on Talrick you cannot differentiate
between back right and back middle right, and between back left and back middle left. */
#define BUMPER_BACK_RIGHT 78
#define BUMPER_BACK 44
#define BUMPER_BACK_LEFT 15
#define BUMPER_FRONT_RIGHT 15
#define BUMPER_FRONT_M_RIGHT 22
#define BUMPER_FRONT 44
#define BUMPER_FRONT_M_LEFT 78
#define BUMPER_FRONT_LEFT 127


/*** ir ***/
#define NUM_IR 4 /*I am using this many IR's*/
#define IR_OFFSET 2 /*my first IR starts at array location 2*/
#define IR_FUZZY_INCREASE 10 /*more fuzzy logic values, these are for IR*/
#define IR_THRESHOLD_PERCENT 5 /*percent (actually inverse) used in calculating ir thresholds*/

#define IRE_OUT *(unsigned char *)(0xffb9) /*IR emitter output port driver, the output latch at address 0xffb9 */
#define IRE_ALL_ON 0xff /*Constant for driving all the 40KHz modulated IR emitters on when loaded into IRE_OUT */
#define IRE_ALL_OFF 0x00 /*Constant for turning all the 40KHz modulated IR emitters off when loaded into IRE_OUT */
#define IRE_FOUR 0x03 /*Constant for my four IR's*/


/*easy names into ir array*/
#define ir_left IRDT[2] /*IRDFML*/
#define ir_front IRDT[3] /*IRDFM*/
#define ir_right IRDT[4] /*IRDFMR*/
#define ir_back IRDT[5] /*IRDBM*/

/*
threshold macros
- The first commented out are hard-coded, from previous testing
- Next are based on the low readings added with a predetermined value
- Third are the low readings plus a % of the range

I incrementally developed these, and each seemed to be better
*/
/*
#define ir_left_THRESHOLD 91
#define ir_front_THRESHOLD 98
```

```
#define ir_right_THRESHOLD 88
#define ir_back_THRESHOLD 98
*/
/*
#define ir_left_THRESHOLD (ir_low[0] + IR_FUZZY_INCREASE)
#define ir_front_THRESHOLD (ir_low[1] + IR_FUZZY_INCREASE)
#define ir_right_THRESHOLD (ir_low[2] + IR_FUZZY_INCREASE)
#define ir_back_THRESHOLD (ir_low[3] + IR_FUZZY_INCREASE)
*/
#define ir_left_THRESHOLD (ir_low[0] + (int)((ir_high[0]-
ir_low[0])/IR_THRESHOLD_PERCENT))
#define ir_front_THRESHOLD (ir_low[1] + (int)((ir_high[1]-
ir_low[1])/IR_THRESHOLD_PERCENT))
#define ir_right_THRESHOLD (ir_low[2] + (int)((ir_high[2]-
ir_low[2])/IR_THRESHOLD_PERCENT))
#define ir_back_THRESHOLD (ir_low[3] + (int)((ir_high[3]-
ir_low[3])/IR_THRESHOLD_PERCENT))

/*** microphone ***/
#define MIC IRDT[11] /*microphone signal comes through free ir analog input; #12 on
board hence 11 in array*/
#define MIC_THRESHOLD 10

/*** cds ***/
#define NUM_CDS 6 /*I am using this many cds cells*/

#define nose_left CDS[5] /*NCDSL*/
#define nose_middle CDS[4] /*NCDFM*/
#define nose_right CDS[3] /*NCDSR*/
#define eye_left CDS[2] /*ECDSL*/
#define eye_middle CDS[1] /*ECDSM*/
#define eye_right CDS[0] /*ECDSR*/

#define LIGHT_THRESHOLD 20

/* not sure I will have cds thresholds*/
/*
#define nose_left_THRESHOLD cds_ambient[5]
#define nose_middle_THRESHOLD cds_ambient[4]
#define nose_right_THRESHOLD cds_ambient[3]
#define eye_left_THRESHOLD cds_ambient[2]
#define eye_middle_THRESHOLD cds_ambient[1]
#define eye_right_THRESHOLD cds_ambient[0]
*/
/*
#define nose_left_THRESHOLD (cds_ambient[5] + NOSE_FUZZY_INCREASE)
```

```
#define nose_middle_THRESHOLD (cds_ambient[4] + NOSE_FUZZY_INCREASE)
#define nose_right_THRESHOLD (cds_ambient[3] + NOSE_FUZZY_INCREASE)
#define eye_left_THRESHOLD (cds_ambient[2] + EYE_FUZZY_INCREASE)
#define eye_middle_THRESHOLD (cds_ambient[1] + EYE_FUZZY_INCREASE)
#define eye_right_THRESHOLD (cds_ambient[0] + EYE_FUZZY_INCREASE)
*/

#define FUZZY_LINE_THRESHOLD 3

#define FUZZY_LIGHT_THRESHOLD 3

/***
behaviors
***/
#define OBF omniscient_behavior_function /*just a name that is substituted for the main behavior function
                                         beats typing 'omniscient_behavior_function' each time*/

/***
motor
***/
#define OMF omniscient_motor_function /*just a name that is substituted for the main motor function
                                      beats typing 'omniscient_motor_funtion' each time*/

/*motor powers*/
#define FULL_PWR 100
#define HALF_PWR 50
#define QUARTER_PWR 25
#define NO_PWR 0

/*difference threshold difference between what the motor is moving at and what we want to move*/
#define ACCELERATION_THRESHOLD 30

/*since the right motor is stronger, we compensate the left motor to only a % of the value*/
#define COMPENSATE_RIGHT_MOTOR(A) (int)(A *1.0)
//#define COMPENSATE_RIGHT_MOTOR(A) A

/* these are commands that can be passed to the OMT, names should be sufficiently self descriptive */
/* !!! if this compiler supported enum this wouldn't be so #$%^ painful !!! */
#define STOP 0 /*compelete stop*/
```

```
#define AHEAD 1 /*full forward*/
#define BACK 2 /*full backwards*/
#define LEFT 3 /*turn left*/
#define RIGHT 4 /*turn right*/
#define HALF_AHEAD 5 /*half speed forward*/
#define HALF_BACK 6 /*" " back*/
#define ARC_LEFT 7 /*arc left*/
#define ARC_RIGHT 8 /*" right*/
#define INCREASE_AHEAD 9 /*speed up forward*/
#define INCREASE_BACK 10 /*" " back*/
#define INCREASE_LEFT 11 /*" " left*/
#define INCREASE_RIGHT 12 /*" " right*/
#define DECREASE_AHEAD 13 /*slow down forward*/
#define DECREASE_BACK 14 /*" " back*/
#define DECREASE_LEFT 15 /*" " left*/
#define DECREASE_RIGHT 16 /*" " right*/
#define CLOCKWISE 17 /*rotate clockwise in place*/
#define COUNTERCLOCKWISE 18 /*rotate counter-clockwise in place*/
#define ARC_BACK_LEFT 19 /*arc left and back*/
#define ARC_BACK_RIGHT 20 /*" right and back*/
#define RANDOM 253 /*turn ahead in random direction*/
#define RANDOM_WISE 254 /*rotate in random direction*/
#define UPDATE 255 /*no new instructions, called to update the current motor values*/

/*delays for startup, change directions (ie, speed), etc., testing*/
#define WAIT_LED 300
#define WAIT_MOTOR 5
#define WAIT_READ_SENSOR 1

#define AVOID_OBSTACLES_TIME1 1000
#define COLLISION_WAIT_TIME1 800
#define COLLISION_WAIT_TIME2 700

/*X10 stuff*/
#define DTR 0x40 /* I am passing my X10 signals over the digital out*/
#define RTS 0x80 /* IRE_OUT pins 7 and 8*/

#define MILLISECOND_BIT_DELAY 1 /*each signal bit must be at least .5 millisecs,
we send for a whole one */
#define MICROSECOND_BIT_DELAY 1000 /*same as above, but in microseconds)*/

#define A1_ON 1 /*'on' signal*/
#define A1_OFF 0 /*'off' signal*/

/************************** Prototypes ********************************/
```

```
void main(void);
void init(void);
void calibrate(void);
void translate_move(unsigned int command);
void buzz(unsigned int buzz_time);
void buzzer_task(void);
void read_sensors(void);
void default_task(void);
void avoid_obstacles_task(void);
void collision_escape_task(void);
void line_follow_task(void);
void light_follow_task(void);
void wall_follow_task(void);
void listen_mic_task(void);
#pragma interrupt_handler X10
void X10(void); /*this is actually an interrupt handler now*/
void microsecond_wait(unsigned int);
void OBF(void);
void OMF(void);

#ifdef DEBUG
 void print_debug_task(void);
#endif


/****************************** Globals *********************************/

/* These are sensor values.  low+high are used to calculate range and thresholds.*/
unsigned int ir_low[NUM_IR], ir_high[NUM_IR];
unsigned int ir_ambient[NUM_IR];
unsigned int mic_low, mic_ambient, mic_high;
unsigned int mic_latch;


/* not sure if I'll use these */
unsigned int cds_low[NUM_CDS], cds_ambient[NUM_CDS], cds_high[NUM_CDS];

unsigned int fb, rb; /*front and rear bumpers*/
unsigned int bl, cv; /*battery level & charge voltage*/

unsigned int buzz_time; /*time we want the buzzer to buzz for */

/*behavior variables*/
/* used in the OBT to know what behavior is currently running, and what we want to do
*/
unsigned int current_command;
```

```
unsigned int desired_command;
int current_command_count;

/*motor variables*/
/* these four values represent *speed*, what we are currently at and what we want to be
at, for each left and right motor*/
int current_left_motor;
int current_right_motor;
int desired_left_motor;
int desired_right_motor;

/*tasks*/
/*read_sensors, OBF, and OMF are always running, so don't track their pid*/
int default_task_pid;
int avoid_obstacles_task_pid;
int collision_escape_task_pid;
int line_follow_task_pid;
int light_follow_task_pid;
int wall_follow_task_pid;
int listen_mic_task_pid;
#ifdef DEBUG
 int print_debug_task_pid;
#endif

/*priority flags for each behavior's task*/
int default_task_flag;
int avoid_obstacles_task_flag;
int collision_escape_task_flag;
int line_follow_task_flag;
int light_follow_task_flag;
int wall_follow_task_flag;
int listen_mic_task_flag;
/*print debug has no effect, hence no flag*/

/*commands for each behavior's task*/
unsigned int default_task_command;
unsigned int avoid_obstacles_task_command;
unsigned int collision_escape_task_command;
unsigned int line_follow_task_command;
unsigned int light_follow_task_command;
unsigned int wall_follow_task_command;
unsigned int listen_mic_task_command;
/*print debug has no effect, hence no command*/

/*X10 stuph*/
```

```
/*next two are hard coded commands, I didn't implement the whole X10 protocol.  It's
easy to do, just grab the code from
http://mlug.missouri.edu/~tymm
*/
unsigned char cmd_a1on[5] = { 0xd5, 0xaa, 0x60, 0x00, 0xad };
unsigned char cmd_a1off[5] = { 0xd5, 0xaa, 0x60, 0x20, 0xad };
unsigned int x10_bits_sent; /*global to know how much of the signal we've sent*/
unsigned int x10_command; /*x10 command to send, currently limited to two right
above*/
unsigned char ir_hold; /*to save state of IR's since I have the X10 DTR and RTS signals
on the same mux*/
unsigned char ir_temp; /*temp ir value, for same reason as hold above*/

#endif /*ifdef _LUCID_H_ - wraps *entire* file, minus a few comments*/
```

```
/* lucid9.c
 * Eugenio Jarosiewicz
 * EEL5666 - IMDL - Intelligent Machines Design Lab
 * Summer 1999, University Of Florida
 */

#ifndef _LUCID_C_
#define _LUCID_C_


/***************************** Includes *********************************/
#include "lucid9.h"
/************************* End of includes ******************************/


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

/****************************** main()
************************************/
/*
*This is the main routine.  It starts and does everything.
*For information on an identifier (variable, constant, function name),
*information is generally (clearly not always) located with the declaration, not the use.
*/
 void main(void)
{
        #ifdef DEBUG
                /*print something nice when started ; only if debugging is on*/
                printf(CLEAR);
                printf(HOME);
                printf("\nHello World! My name is Lucid.");
        #endif

        init();

        calibrate();

        /*
        start all the processes
        */

        /*these are always running - base sensing, behavior, and actuation control*/
```

```
	create_process(read_sensors, 0, TASK_INTERVAL_TIME,
TASK_STACK_SIZE);
	create_process(OBF, 0, TASK_INTERVAL_TIME, TASK_STACK_SIZE);
	create_process(OMF, 0, TASK_INTERVAL_TIME, TASK_STACK_SIZE);

	/*these are actual behaviors*/
	default_task_pid = create_process(default_task, 0, TASK_INTERVAL_TIME,
TASK_STACK_SIZE);
	avoid_obstacles_task_pid = create_process(avoid_obstacles_task, 0,
TASK_INTERVAL_TIME, TASK_STACK_SIZE);
	collision_escape_task_pid = create_process(collision_escape_task, 0,
TASK_INTERVAL_TIME, TASK_STACK_SIZE);

	/*these are behaviors for my extra sensors - microphone and X10 interface*/
	create_process(buzzer_task, 0, TASK_INTERVAL_TIME,
TASK_STACK_SIZE);
	listen_mic_task_pid = create_process(listen_mic_task, 0,
TASK_INTERVAL_TIME, TASK_STACK_SIZE);

//	line_follow_task_pid = create_process(line_follow_task, 0,
TASK_INTERVAL_TIME, TASK_STACK_SIZE);
//	light_follow_task_pid = create_process(light_follow_task, 0,
TASK_INTERVAL_TIME, TASK_STACK_SIZE);

	#ifdef DEBUG
//		print_debug_task_pid = create_process(print_debug_task, 0,
TASK_INTERVAL_TIME, TASK_STACK_SIZE);
	#endif

	start_scheduler();
	/*
	* execution does not return from above routine, so it should be last thing called
	* it will preemptively multitaskingly execute in round-robin all the
tasks/processes created previously
	*/
} /*end of main()*/


/****************************** init() ***********************************/
/*
* This routine initializes the base tk modules
* The 4 init_xxx() calls are to Doty's code
*/
void init(void)
{
	#ifdef DEBUG
```

```
		printf("\nInitializing.");
	#endif

	init_analog();
	init_motortk();
	init_clocktk();
	init_serial();

	IRE_OUT = IRE_FOUR;

	current_left_motor = NO_PWR;
	current_right_motor = NO_PWR;
	desired_left_motor = NO_PWR;
	desired_right_motor = NO_PWR;

	current_command = STOP;
	desired_command = STOP;

	current_command_count = 1;
} /*end of init()*/


/****************************** calibrate()
**********************************/
/*
* This routine calibrates the sensors that I am using
* Right now I'm assuming that the robot starts out in an open environment, low sound,
etc
*/
void calibrate(void)
{
	unsigned int i,j,tests=2;

	#ifdef DEBUG
		printf("\nCalibrating.");
	#endif

	/* init values*/
	mic_low = 255;
	mic_ambient = 0;
	mic_high = 0;
	for (i = 0; i < NUM_IR; i++)
	{
		ir_low[i] = 255;
		ir_ambient[i] = 0;
		ir_high[i] = 0;
```

```
}

for (i = 0; i < NUM_CDS; i++)
{
	cds_low[i]=255;
	cds_ambient[i]=0;
	cds_high[i]=0;
}

for (j = 0; j < tests; j++)
{
	IRE_OUT = IRE_ALL_OFF; /*turn off LEDS to get their low values*/
	wait(WAIT_LED); /*wait for emmissions to die down*/

	read_IR();
	for (i = 0; i < NUM_IR; i++)
	{
		if (IRDT[i+IR_OFFSET] < ir_low[i])
		{
			ir_low[i] = IRDT[i+IR_OFFSET];
		}
	}

	IRE_OUT = IRE_FOUR; /*turn on LEDS to get their high values*/
	wait(WAIT_LED); /*wait for emmissions to die down*/

	read_IR();
	for (i = 0; i < NUM_IR; i++)
	{
		ir_ambient[i] += IRDT[i+IR_OFFSET];
		if (IRDT[i+IR_OFFSET] < ir_low[i])
		{
			ir_low[i] = IRDT[i+IR_OFFSET];
		}
		else if (IRDT[i+IR_OFFSET] > ir_high[i])
		{
			ir_high[i] = IRDT[i+IR_OFFSET];
		}
	}

	/*mic is on IR input*/
	mic_ambient += MIC;
	if ( MIC < mic_low )
	{
		mic_low = MIC;
	}
```

```
                else if ( MIC > mic_high )
                {
                        mic_high = MIC;
                }

                read_CDS();
                for (i = 0; i < NUM_CDS; i++)
                {
                        cds_ambient[i] += CDS[i];
                        if ( CDS[i] < cds_low[i] )
                        {
                                cds_low[i] = CDS[i];
                        }
                        else if ( CDS[i] > cds_high[i] )
                        {
                                cds_high[i] = CDS[i];
                        }
                }
        }

        /*get the average of the ambient readings*/
        for (i = 0; i < NUM_IR; i++)
        {
                ir_ambient[i] /= tests;
        }

        mic_ambient /= tests;

        for (i = 0; i < NUM_CDS; i++)
        {
                cds_ambient[i] /= tests;
        }

} /*end of calibrate()*/


/****************** translate_move() ************************/
/*
this routine works by (1) accepting a direction command
it then (2) sets the desired motor rates in response to the command
it then (3) does some sanity checks
*/
void translate_move(unsigned int command)
{
        #ifdef DEBUG
//                printf("\ntranslate_move command %d", command);
```

```
        #endif

        switch (command) {
        case STOP:
//              printf(" aka STOP");
                desired_left_motor = NO_PWR;
                desired_right_motor = NO_PWR;
                break;
        case AHEAD:
//              printf(" aka AHEAD");
                desired_left_motor = FULL_PWR;
                desired_right_motor = FULL_PWR;
                break;
        case BACK:
//              printf(" aka BACK");
                desired_left_motor = -(FULL_PWR);
                desired_right_motor = -(FULL_PWR);
                break;
        case LEFT:
//              printf(" aka LEFT");
                desired_left_motor = NO_PWR;
                desired_right_motor = FULL_PWR;
                break;
        case RIGHT:
//              printf(" aka RIGHT");
                desired_left_motor = FULL_PWR;
                desired_right_motor = NO_PWR;
                break;
        case HALF_AHEAD:
//              printf(" aka HALF_AHEAD");
                desired_left_motor = HALF_PWR;
                desired_right_motor = HALF_PWR;
                break;
        case HALF_BACK:
//              printf(" aka HALF_BACK");
                desired_left_motor = -(HALF_PWR);
                desired_right_motor = -(HALF_PWR);
                break;
        case ARC_LEFT:
//              printf(" aka ARC_LEFT");
                desired_left_motor = QUARTER_PWR;
                desired_right_motor = HALF_PWR;
                break;
        case ARC_RIGHT:
//              printf(" aka ARC_RIGHT");
                desired_left_motor = HALF_PWR;
```

```
                desired_right_motor = QUARTER_PWR;
                break;
        case INCREASE_AHEAD:
        case DECREASE_BACK: /*speeding up forward is same as slowing down
backward*/
//              printf(" aka INCREASE_AHEAD or DECREASE_BACK");
                desired_left_motor += 10;
                desired_right_motor += 10;
                break;
        case INCREASE_BACK:
        case DECREASE_AHEAD: /*speeding up backward is same as slowing down
forward*/
//              printf(" aka INCREASE_BACK or DECREASE_AHEAD");
                desired_left_motor -= 10;
                desired_right_motor -= 10;
                break;
        case INCREASE_LEFT:
        case DECREASE_RIGHT: /*speeding up left is same as slowing down right,
almost*/
//              printf(" aka INCREASE_LEFT or DECREASE_RIGHT");
                desired_left_motor -= 10;
                desired_right_motor += 10;
                break;
        case INCREASE_RIGHT:
        case DECREASE_LEFT: /*speeding up right is same as slowing down left,
almost*/
//              printf(" aka INCREASE_RIGHT or DECREASE_LEFT");
                desired_left_motor += 10;
                desired_right_motor -= 10;
                break;
        case CLOCKWISE:
//              printf(" aka CLOCKWISE");
                desired_left_motor = FULL_PWR;
                desired_right_motor = -(FULL_PWR);
                break;
        case COUNTERCLOCKWISE:
//              printf(" aka COUNTERCLOCKWISE");
                desired_left_motor = -(FULL_PWR);
                desired_right_motor = FULL_PWR;
                break;
        case ARC_BACK_LEFT:
//              printf(" aka ARC_BACK_LEFT");
                desired_left_motor = -(HALF_PWR);
                desired_right_motor = -(FULL_PWR);
                break;
        case ARC_BACK_RIGHT:
```

```
//                printf(" aka ARC_BACK_RIGHT");
                  desired_left_motor = -(FULL_PWR);
                  desired_right_motor = -(HALF_PWR);
                  break;
        case RANDOM:
//                printf(" aka RANDOM");
                  if (TCNT & 0x0001)
                  {
                        /*right*/
//                      printf("-right");
                        desired_left_motor = FULL_PWR;
                        desired_right_motor = NO_PWR;
                  }
                  else
                  {
                        /*left*/
//                      printf("-left");
                        desired_left_motor = NO_PWR;
                        desired_right_motor = FULL_PWR;
                  }
                  break;
        case RANDOM_WISE:
//                printf(" aka RANDOM_WISE");
                  if (TCNT & 0x0001)
                  {
                        /*clockwise*/
//                      printf("-clockwise");
                        desired_left_motor = FULL_PWR;
                        desired_right_motor = -(FULL_PWR);
                  }
                  else
                  {
                        /*counterclockwise*/
//                      printf("-counterclockwise");
                        desired_left_motor = -(FULL_PWR);
                        desired_right_motor = FULL_PWR;
                  }
                  break;
        case UPDATE: /*no changes to desired values, just might need to update current
below*/
//                printf(" aka UPDATE");
                  break;
        default:
                  printf("\nIllegal move command %d",command);
                  break;
        }
```

```
	if ( desired_left_motor < -(FULL_PWR) ) desired_left_motor = -(FULL_PWR);
	else if ( desired_left_motor > FULL_PWR ) desired_left_motor = FULL_PWR;
	if ( desired_right_motor < -(FULL_PWR) ) desired_right_motor = -
(FULL_PWR);
	else if ( desired_right_motor > FULL_PWR ) desired_right_motor =
FULL_PWR;

} /*end of translate_move()*/


/****************** buzz() ************************/
/*
this routine buzzes for a given amount of time, with busy wait.
this buzz_time variable hides the global one.
*/
void buzz(unsigned int buzz_time)
{
		SET_BIT(PORTA, 0x08);
		wait(buzz_time);
		CLEAR_BIT(PORTA, 0x08);
} /*end of buzz()*/


/****************** buzzer_task() ************************/
/*
this routine buzzes for a given amount of time, but multitasked
*/
void buzzer_task(void)
{
	while(1)
	{
		if ( buzz_time > 0 )
		{
			SET_BIT(PORTA, 0x08);
			wait(buzz_time);
			CLEAR_BIT(PORTA, 0x08);
			buzz_time=0;
		}
		else
		{
			//defer();
		}
	}
} /*end of buzzer_task()*/
```

```
/******************************** read_sensors()
**********************************/
void read_sensors(void)
{
	int i;

	#ifdef DEBUG
		printf("\nRead sensors started.");
	#endif

	while (1)
	{
		read_IR();
		for (i = 0; i < NUM_IR; i++)
		{
		/*always keep track of low+high readings*/
			if ( IRDT[i+IR_OFFSET] < ir_low[i] )
			{
				ir_low[i] = IRDT[i+IR_OFFSET];
			}
			else if ( IRDT[i+IR_OFFSET] > ir_high[i] )
			{
				ir_high[i] = IRDT[i+IR_OFFSET];
			}
		}

		if ( MIC < mic_low )
		{
			mic_low = MIC;
		}
		else if ( MIC > mic_high )
		{
			mic_high = MIC;
		}

		/*this is the base behavior of microphone listening!*/
		if ( MIC < 90 || MIC > 200 ) /*ugh, hard coded.  TODO */
		{
			mic_latch = 1; /*listen_mic_task will clear it.*/
		}

		read_CDS();
		for (i = 0; i < NUM_CDS; i++)
		{
```

```
                        if ( CDS[i] < cds_low[i] )
                        {
                                cds_low[i] = CDS[i];
                        }
                        else if ( CDS[i] > cds_high[i] )
                        {
                                cds_high[i] = CDS[i];
                        }
                }

                fb = front_bumper();
                rb = rear_bumper();
                bl = battery_level();
                cv = charge_volts();
        }
} /*end of read_sensors()*/


/******************************* default_task()
************************************/
void default_task(void)
{
        #ifdef DEBUG
                printf("\nDefault task started.");
        #endif

        while (1)
        {
                default_task_command = AHEAD;
                default_task_flag = 1;
        }
} /*end of default_task()*/


/******************************* avoid_obstacles_task()
************************************/
void avoid_obstacles_task(void)
{
        #ifdef DEBUG
                printf("\nAvoid obstacles task started.");
        #endif

        while (1)
        {
                if ( ir_left > ir_left_THRESHOLD && ir_right > ir_right_THRESHOLD )
                {
```

```
                        avoid_obstacles_task_command = ARC_BACK_RIGHT;
                        avoid_obstacles_task_flag = 1;
                        wait(AVOID_OBSTACLES_TIME1);
                }
/*something is WRONG here, left & right are BACKWARDS!*/
                else if ( ir_left > ir_left_THRESHOLD && ir_right <=
ir_right_THRESHOLD )
                {
//                      avoid_obstacles_task_command = RIGHT;
                        avoid_obstacles_task_command = LEFT;
                        avoid_obstacles_task_flag = 1;
                }
                else if ( ir_left <= ir_left_THRESHOLD && ir_right >
ir_right_THRESHOLD )
                {
//                      avoid_obstacles_task_command = LEFT;
                        avoid_obstacles_task_command = RIGHT;
                        avoid_obstacles_task_flag = 1;
                }
/*              else if ( ir_back > ir_back_THRESHOLD )
                {
                        avoid_obstacles_task_command = AHEAD;
                        avoid_obstacles_task_flag = 1;
                }
*/
                else
                {
                        avoid_obstacles_task_flag = 0;
                }

        }
} /*end of avoid_obstacles_task()*/


/****************************** collision_escape_task()
*********************************/
void collision_escape_task(void)
{
        int hold_fb;
        int hold_rb;

        #ifdef DEBUG
                printf("\nCollision escape task started.");
        #endif

        while (1)
```

```
        {
                hold_fb = fb;
                hold_rb = rb;
                if ( hold_fb > BUMPER_FUZZY_ZERO && hold_rb >
BUMPER_FUZZY_ZERO )
                {
                //something curious is happening
                //TODO
                        collision_escape_task_command = RANDOM_WISE;
                        collision_escape_task_flag = 1;
                        wait(COLLISION_WAIT_TIME1);
                }

                if ( hold_fb > BUMPER_FUZZY_ZERO )
                {
                        if ( abs(hold_fb - BUMPER_FRONT_RIGHT) <=
BUMPER_FUZZY_RANGE
                         || abs(hold_fb - BUMPER_FRONT_M_RIGHT) <=
BUMPER_FUZZY_RANGE)
                        {
//                              collision_escape_task_command = ARC_BACK_RIGHT;
                                collision_escape_task_command = HALF_BACK;
                                collision_escape_task_flag = 1;
                                wait(COLLISION_WAIT_TIME1);
                                collision_escape_task_command =
COUNTERCLOCKWISE;
//                              collision_escape_task_command = ARC_LEFT;
                                wait(COLLISION_WAIT_TIME2);
                        }
                        else if ( abs(hold_fb - BUMPER_FRONT) <=
BUMPER_FUZZY_RANGE )
                        {
                                collision_escape_task_command = ARC_BACK_LEFT;
                                collision_escape_task_flag = 1;
                                wait(COLLISION_WAIT_TIME1);
                                collision_escape_task_command = ARC_RIGHT;
                                wait(COLLISION_WAIT_TIME2);
                        }
                        else
                        {
//                              collision_escape_task_command = ARC_BACK_LEFT;
                                collision_escape_task_command = HALF_BACK;
                                collision_escape_task_flag = 1;
                                wait(COLLISION_WAIT_TIME1);
//                              collision_escape_task_command = ARC_RIGHT;
                                collision_escape_task_command = CLOCKWISE;
```

```
                        wait(COLLISION_WAIT_TIME2);
                }
        }
        else
        {
                collision_escape_task_flag = 0;
        }

        if ( hold_rb > BUMPER_FUZZY_ZERO )
        {
                if ( abs(hold_rb - BUMPER_BACK_RIGHT) <=
BUMPER_FUZZY_RANGE )
                {
                        collision_escape_task_command =
COUNTERCLOCKWISE;
                        collision_escape_task_flag = 1;
                        wait(COLLISION_WAIT_TIME1);
                        collision_escape_task_command = ARC_LEFT;
                        wait(COLLISION_WAIT_TIME2);
                }
                else if ( abs(hold_rb - BUMPER_BACK_LEFT) <=
BUMPER_FUZZY_RANGE )
                {
                        collision_escape_task_command = CLOCKWISE;
                        collision_escape_task_flag = 1;
                        wait(COLLISION_WAIT_TIME1);
                        collision_escape_task_command = ARC_RIGHT;
                        wait(COLLISION_WAIT_TIME2);
                }
                else
                {
                        collision_escape_task_command = AHEAD;
                        collision_escape_task_flag = 1;
                        wait(COLLISION_WAIT_TIME1);
                }
        }
        else
        {
                collision_escape_task_flag = 0;
        }
    }
} /*end of collision_escape_task()*/


/****************************** line_follow_task()
*********************************/
```

```
void line_follow_task(void)
{
	int nose_delta;

	#ifdef DEBUG
		printf("\nLine follow task started.");
	#endif

	while (1)
	{
		nose_delta = nose_left - nose_right;
		if ( abs(nose_delta) >= FUZZY_LINE_THRESHOLD )
		{
			if ( nose_delta > 0 )
			{
				line_follow_task_command = LEFT;
			}
			else
			{
				line_follow_task_command = RIGHT;
			}
			line_follow_task_flag = 1;
		}
		else
		{
			line_follow_task_flag = 0;
		}
	}
} /*end of line_follow_task()*/


/****************************** light_follow_task()
*********************************/
void light_follow_task(void)
{
	int eye_delta;

	#ifdef DEBUG
		printf("\nLight follow task started.");
	#endif

	while (1)
	{
//printf("\n le %d re %d", eye_left, eye_right);

		eye_delta = eye_left - eye_right;
```

```
                if ( abs(eye_delta) >= FUZZY_LIGHT_THRESHOLD )
                {
                        if ( eye_delta > 0 )
                        {
                                light_follow_task_command = ARC_LEFT;
                                light_follow_task_flag = 1;
                        }
                        else
                        {
                                light_follow_task_command = ARC_RIGHT;
                                light_follow_task_flag = 1;
                        }
                }
                else
                {
                        light_follow_task_flag = 0;
                }
        }
} /*end of light_follow_task()*/


/******************************** listen_mic_task()
**********************************/
/*
Every minute, stop and listen to microphone (so no interference from noise of motors,
mic is sensitive)
if there is no noise, then
if we think the lights are on, send an x10 command to turn them off.
else, we emit a buzz just for us to know we are in this part of the program, and also turn
light on
*/
  void listen_mic_task(void)
{
        #ifdef DEBUG
                printf("\nListen mic task started.");
        #endif

        while (1)
        {
                if ( mic_latch )
                {
                        #ifdef DEBUG
                                printf("\nHmm, people...");
                        #endif

                        buzz(25); /*buzz to let peeople know*/
```

```
                        mic_latch = 0; /*clear flag*/

                        ir_temp=ir_hold=IRE_OUT; /*save IR state*/

                        if ( eye_middle < LIGHT_THRESHOLD ) /* this thresh is hard
coded, TODO someday*/
                        {
                                x10_command = A1_ON; /*set up command to be
executed,
                                                                        in this case
turn the light on since the eye reads low*/
                                buzz(25); /*buzz twice if turning on to distinguish*/
                        }
                        else
                        {
                                x10_command = A1_OFF; /*set up command to be
executed,
                                                                        in this case
turn the light off since the eye reads >= threshold*/
                                /*we've buzzed once already, that's the human signal for
off*/
                        }

                        x10_bits_sent = 0; /*set up location to start sending from*/

                        /*place firecracker in wait mode, both DTR=RTS=1*/
                        SET_BIT(ir_temp, DTR);
                        SET_BIT(ir_temp, RTS);
                        IRE_OUT = ir_temp;

                        INTR_OFF(); /*time to set up timing interrupts*/
                        // do nothing for oc1 CLEAR_BIT(TCTL1,0x30); /* Interrupt will
not affect OC1 pin */

                        TOC1 += CLOCK_TICK; /*interrupt in 1 millisec, (1msec = 2000
clock cycles for 8MHz crystal), defined in clocktk*/
                        SET_BIT(TMSK1,0x80); /* Enable OC1 interrupt */

                        INTR_ON(); /*we are ready to go*/

                        wait(100); /*by the end of this the x10 interrupt handler will be
called, and when we return the signal will
                                                have been auto-magikally sent;-)*/

                        IRE_OUT = ir_hold; /*restore IR's back to what they were*/
                }
```

```
                else
                {
                        wait(5);
                        //defer();
                }
        }
} /*end of listen_mic_task()*/


/****************************** X10() ***********************************/
/*
This routine receives an on/off message.
It then sends out the appropriate message via the two lines I have set up connecting to the
X10 'Firecracker' module
Due to memory requirements, it is not a full implementation of X10, only the two
messages I need.
I must also turn interrupts off inside the routine because timing is important.
A better implementation instead of busy wait would be instead *using* interrupts, instead
of disabling them...
This is just proof-of-concept.
*/
void X10(void)
{
        unsigned int xbyte;
        unsigned int xbit;
        unsigned char signal_bit;

        #ifdef DEBUG
        //      printf("\nX10. command=%d.", x10_command);
        #endif

        if ( x10_bits_sent < 80 )
        {
                TOC1 += CLOCK_TICK;     /* Set the timer to interrupt again in 1msec.
*/

                if ( (x10_bits_sent%2) == 1) /*odd count, send a wait*/
                {
                        SET_BIT(ir_temp, DTR);
                        SET_BIT(ir_temp, RTS);
                        IRE_OUT = ir_temp;
                }
                else /*even count, send next bit of the command*/
                {
                        xbyte = (x10_bits_sent/2)/8;
                        xbit = 7-((x10_bits_sent/2)%8);
```

```
//                      printf("\nB%d b%d",xbyte,xbit);

                        if ( x10_command == A1_ON)
                        {
                                signal_bit = cmd_a1on[xbyte] & (1<<xbit);
                        }
                        else if (x10_command == A1_OFF)
                        {
                                signal_bit = cmd_a1off[xbyte] & (1<<xbit);
                        }

                        if ( signal_bit )
                        {
                                SET_BIT(ir_temp, RTS);
                                CLEAR_BIT(ir_temp, DTR);
                        }
                        else
                        {
                                CLEAR_BIT(ir_temp, RTS);
                                SET_BIT(ir_temp, DTR);
                        }

                        IRE_OUT = ir_temp;
                }
                ++x10_bits_sent;
        }
        else
        {
                CLEAR_BIT(TMSK1,0x80); /* Should be finished sending x10
command, disable further OC3 interrupts */
        }

        CLEAR_FLAG(TFLG1,0x80); /* Clear OC3I flag */
} /*end of X10()*/


/****************** OBF aka microsecond_wait() ************************/
/*does a hard busy wait for a given # of microseconds*/
void microsecond_wait(unsigned int microsecs)
{
        unsigned int mark = TCNT + (microsecs*2);

        if(mark >= microsecs) /* Check for unsigned integer overflow */
                while (TCNT < mark); /* Wait for timer to count microsecs. */
        else /* Compensate for overflow */
```

```
	{
		while (0 != TCNT);
		while (TCNT < mark);
	}
} /*end of microsecond_wait()*/

/****************** OBF aka omniscient_behavior_function()
************************/
/*omniscient_behavior_function - is in charge of behaviors & associated states; behavior
arbitrator*/
void OBF(void)
{
	#ifdef DEBUG
		printf("\nOBF task started");
	#endif

	while (1)
	{
		if (default_task_flag)
		{
			desired_command = default_task_command;
		}
/*		if (line_follow_task_flag)
		{
			desired_command = line_follow_task_command;
		}
		if (light_follow_task_flag)
		{
			desired_command = light_follow_task_command;
		}
*/		if (avoid_obstacles_task_flag)
		{
			desired_command = avoid_obstacles_task_command;
		}
		if (collision_escape_task_flag)
		{
			desired_command = collision_escape_task_command;
		}

		if (listen_mic_task_flag)
		{
			desired_command = listen_mic_task_command;
		}


/*
```

```
                if (listen_mic_task_flag)
                {
                        printf("\nlisten %d",listen_mic_task_command);
                }
                else
*//*            if (collision_escape_task_flag)
                {
                        printf("\ncollision %d",collision_escape_task_command);
                }
                else if (avoid_obstacles_task_flag)
                {
                        printf("\navoid %d",avoid_obstacles_task_command);
                }
                else if (light_follow_task_flag)
                {
                        printf("\nlight %d",light_follow_task_command);
                }
                else if (line_follow_task_flag)
                {
                        printf("\nline %d",line_follow_task_command);
                }
                else if (default_task_flag)
                {
                        printf("\ndefault %d",default_task_command);
                }
*/

                /*Higher up functions will determine if current behavior has been active
for too long.  can we say TODO ?*/
                if ( desired_command == current_command)
                {
                        ++current_command_count;
                }
                else
                {
                        current_command = desired_command;
                        current_command_count = 1;
                }

                translate_move(current_command);
                wait(1);
        }

} /*end of OBF aka omniscient_behavior_function()*/
```

```
/****************** OMF aka omniscient_motor_function()
************************/
/*
this routine works by (1) caculates the difference between what the motors currently are
and what they want to be
it then (2) sets the *actual* motor values carefully
*/
void OMF(void)
{
	int right_motor_diff, left_motor_diff; /*difference between current and desired
motor values*/

	#ifdef DEBUG
		printf("\nOMF task started");
	#endif

	while (1)
	{
		right_motor_diff = current_right_motor - desired_right_motor;
		left_motor_diff = current_left_motor - desired_left_motor;

		/*do calculations on diffs here so we don't work motor too much*/
		if ( abs(right_motor_diff) < ACCELERATION_THRESHOLD )
		{
			current_right_motor = desired_right_motor;
		}
		else
		{
			if ( right_motor_diff > 0 )
			{
				current_right_motor -= ACCELERATION_THRESHOLD;
			}
			else
			{
				current_right_motor +=
ACCELERATION_THRESHOLD;
			}
		}

		if ( abs(left_motor_diff) < ACCELERATION_THRESHOLD )
		{
			current_left_motor = desired_left_motor;
		}
		else
		{
			if ( left_motor_diff > 0 )
```

```
                        {
                                current_left_motor -= ACCELERATION_THRESHOLD;
                        }
                        else
                        {
                                current_left_motor += ACCELERATION_THRESHOLD;
                        }
                }

                if ( current_left_motor < -(FULL_PWR) ) current_left_motor = -
(FULL_PWR);
                else if ( current_left_motor > FULL_PWR ) current_left_motor =
FULL_PWR;
                if ( current_right_motor < -(FULL_PWR) ) current_right_motor = -
(FULL_PWR);
                else if ( current_right_motor > FULL_PWR ) current_right_motor =
FULL_PWR;


/* Doh, as nice as this is, it screws up timing everywhere...hmmm
#ifdef DEBUG
                motortk(LEFT_MOTOR, current_left_motor/2);
                motortk(RIGHT_MOTOR,
COMPENSATE_RIGHT_MOTOR(current_right_motor/2));
#else
*/
                motortk(LEFT_MOTOR, current_left_motor);
                motortk(RIGHT_MOTOR, current_right_motor);
//#endif
                right_motor_diff = current_right_motor - desired_right_motor;
                motortk(RIGHT_MOTOR,
COMPENSATE_RIGHT_MOTOR(current_right_motor));

                wait(WAIT_MOTOR);

/*              if ( (current_right_motor == desired_right_motor) && (current_left_motor
== desired_left_motor) )
                {
                        wait(10);
                }
*/
        }
} /*end of OMF aka omniscient_motor_function()*/


#ifdef DEBUG
```

```
/****************************** print_debug_task()
*********************************/
void print_debug_task(void)
{
        printf("\nprint_debug_task started");

        while ( 1 )
        {
                printf("\n***** IR *****");
                printf("\nwhich value low high thresh");
                printf("\nl %d %d %d %d", ir_left, ir_low[0], ir_high[0],
ir_left_THRESHOLD);
                printf("\nf %d %d %d %d", ir_front, ir_low[1], ir_high[1],
ir_front_THRESHOLD);
                printf("\nr %d %d %d %d", ir_right, ir_low[2], ir_high[2],
ir_right_THRESHOLD);
                printf("\nb %d %d %d %d", ir_back, ir_low[3], ir_high[3],
ir_back_THRESHOLD);

/*              printf("\nl %d thresh %d f %d thresh %d r %d thresh %d b %d thresh
%d",
                 ir_left, ir_left_THRESHOLD, ir_front, ir_front_THRESHOLD,
                 ir_right, ir_right_THRESHOLD, ir_back, ir_back_THRESHOLD );
*/
                printf("\n***** BUMP *****");
                printf("\nfb %d rb %d",fb, rb);

/*
                printf("\n***** CDS *****");
                printf("\nnl %d thresh %d nm %d thresh %d nr %d thresh %d",
                 nose_left, nose_left_THRESHOLD, nose_middle,
nose_middle_THRESHOLD, nose_right, nose_right_THRESHOLD);
                printf("\nel %d thresh %d em %d thresh %d er %d thresh %d", \
                 eye_left, eye_left_THRESHOLD, eye_middle,
eye_middle_THRESHOLD, eye_right, eye_right_THRESHOLD);
*/

                /*mic is on IR12/11, read_IR already called above, so info is recent, can
use that */
                printf("\n***** MIC *****");
                printf("\nmic %d, mic_ambient %d thresh %d", MIC, mic_ambient,
MIC_THRESHOLD);

                printf("\n***** MOTORS *****");
                printf("\ncurrent : l %d r %d", current_left_motor, current_right_motor );
                printf("\ndesired : l %d r %d", desired_left_motor, desired_right_motor );
```

```
                printf("\n***** BEHAVIORS *****");
                printf("\ncurrent : %d desired : %d", current_command, desired_command
);

                printf("\n***** LEVELS *****");
                printf("\nbl %d cv %d",bl, cv);

                wait(5);
                //defer();
        }
} /* end of print_debug_task()*/
#endif /*ifdef DEBUG*/


/************************** End of Functions ******************************/


#endif /*ifdef _LUCID_C_ - wraps *entire* file, minus a few comments*/
```