Programmable Self-Navigating Lego Car

ECE430 Final Project:
Dr. Song
November 17, 2003

Jeff Keacher
Joey Richey

TABLE OF CONTENTS

1. Introduction

Our final project for ECE430: Microcomputers was to design and construct a small robot capable of following a human-issued directive to a specified location. We chose this project because we would be able to incorporate motors and robotics in a setting slightly more complex than a black-line-following drone.

The robot was designed to allow the destination to be easily input from a keypad and displayed on an LCD. The device, after gathering input from both the user and the GPS, would then navigate to the desired destination. Once at the destination, the robot would stop.

2. Objectives and Specifications

The purpose of our project was to make a robot capable of carrying out navigation to a destination using GPS data. A corollary to this is that our project has a high "cool" factor. Due to complexity issues, we chose not to implement obstacle-avoidance routines.

The robot was required to fulfill several requirements, including:

- The robot shall take input from the user via a keypad.
- The robot shall allow the destination to be specified in terms of distance from the current location, in any direction.
- The robot input distance unit shall be feet.
- The robot shall navigate to the final destination using GPS data.
- The robot shall correct its course on the way to its destination, therefore the robot shall be capable of turning.
- The robot does not need to handle obstacles in its path.
- The robot shall halt once it arrives at its destination
- The robot shall constantly display its current and destination locations while in navigation mode.

3. Strategy and Implementation

We chose to implement our project using a PIC16F877A. We chose this chip because of our previous work with it as well as its easy procurement. For the project, the PIC remained mounted on an X-1 development board. We used this board because it included three components that we deemed essential to our project: a keypad, an LCD display, and a RS-232 serial port.

To navigate to a destination, the robot must be told where the destination is. We programmed the PIC such that this input was gathered using the X-1 keypad and displayed on the LCD. The User Manual elsewhere in this document describes the user interface. Initially, we intended for the user to input the destination in terms of absolute latitude and longitude coordinates. However, this method had two drawbacks: most people are unaware of the latitude and longitude coordinates of anywhere, and the GPS tends to "drift" over a several hundred foot range, leading to poor accuracy. Instead, by inputting a relative distance in feet from the current location, user input is simplified and accuracy is greatly improved.

A Garmin GPS receiver provides location awareness. We chose the Garmin eTrex GPS not only because we already owned one, but also because it incorporates an RS-232 serial port through which location data can be transmitted. The serial data is sent at 9600 bps in 8/N/1 format. Physically connecting the GPS to the X-1 required a special cable, as the GPS has a proprietary plug instead of a standard DB-9 connector. As both the GPS and the X-1 are set up as peripheral devices, we used a null-modem adapter to align the transmit/receive lines in the serial cable. We capture the data in the PIC using interrupts so that serial data is never lost.

The location data can be sent in a variety of formats. We settled on a standard Garmin format known as "Simple Text Output." In Simple Text mode, data is sent from the GPS once per second in a standard, ASCII text sentence. The sentence contains, among other things, the current location and current velocity. By parsing the sentence for these two values, we determine the distance to the target location and the current orientation of the robot. The latitude and longitude data from the GPS is significant to one thousandth of a minute, which, in Indiana, translates roughly to a six-foot accuracy.

One of our requirements was that the robot be able to propel itself to a destination. We chose to fulfill this requirement by building a wheeled chassis powered by electric motors. For ease of construction, we built the chassis out of Legos and powered the wheels using two Mindstorms motors. We included two motors, one for each of the two driven wheels, with each driven wheel on one side of the chassis. These motors are switched on and off by the PIC using IRF540 Power MOSFETs. These transistors were chosen for their ease of implementation (no biasing required) as well as their robust nature (they are rated to nearly 30 amps).

The robot implements a method to correct its course towards the destination. After the user has entered the desired location, the PIC converts the destination to a latitude and longitude coordinate pair. For simplicity, we chose to express that coordinate in terms of minutes, ignoring degrees. As long as the location is not near 0 or 60 minutes (true in Terre Haute), this simplification should have no effect on navigation. After the conversion from feet to coordinates, the PIC opens the serial port and begins parsing the GPS data. All the while the motors remain off. Once good data from the GPS is received, the PIC turns the motors on and begins moving. The PIC continually checks the position information for signs of change. After every few data points, the PIC evaluates its current course of direction. The velocity data from the GPS, in conjunction with an arctangent function, reveals the current direction of motion of the robot. Concurrently, the PIC calculates the direction of travel necessary to propel the robot to the destination. After both the current and required directions are known, the robot subtracts the two angles to calculate the amount of rotation necessary to point itself towards the target. A turning function takes this rotation angle and switches off one motor for a time appropriate to spin the robot towards the correct direction. After the turn, the robot resumes forward direction and repeats the cycle.

There are several conditions under which the device will halt. As mentioned above, the device will not initially move until it knows the destination and is receiving good data from the GPS. If the GPS should stop sending good data, the robot will halt until the data stream resumes. This precaution prevents the robot from following erroneous data or driving "blind." The robot will also stop once it arrives at its destination. After arriving at the destination, the robot will not move again until a new destination is entered.

We encountered several setbacks during development but triumphed over all. In the early stages of the project, we found that enabling code optimization broke our LCD driver. Transferring the code to a new project solved that problem. Calculation of the direction of travel required an arctangent function. One was available in the ANSI math library, but inclusion of that library bloated the size of the program such that it exceeded the bounds of the PIC. Instead, we wrote a custom arctangent function that compiled to roughly 10% of the size of the library version. Perhaps the most frustrating problem was that of stack overflow. After spending a good deal of time experiencing inconsistent errors, we traced the problem to the PIC stack overflowing. This occurred despite the fact that the compiler should have inlined the culprit function.

4. Testing Procedure and Results

We began our testing and debugging by confirming that the robot successfully performed in a static position. We hooked the circuit to an external power supply, inserted LEDs in place of the motors, and moved the GPS receiver to a window (that way the GPS would provide solid data). With this setup, we were able to confirm that the motors would turn on and off at the correct time and that the robot could successfully receive and process the GPS data.

In testing the GPS feed, we compared the position displayed on the X-1 LCD to that displayed on the GPS's internal LCD. The positions matched perfectly. We then confirmed that the destination position was calculated and displayed in accordance with the values entered at the onset of testing. These too matched perfectly.

Satisfied that the robot worked in a static condition, we replaced the LEDs with the Mindstorms motors and mounted the board, a battery and the motors on the Lego chassis. We confirmed that the motors would propel the robot in a straight line.

In the same configuration, we characterized the approximate turning rate by disconnecting one of the motors. That data was used to adjust the timing for the turning subroutine as well as the frequency of turns.

Finally, we connected and mounted the GPS receiver, turned on all of the circuits, and went outside to test the entire device. Unfortunately, the GPS was unable to receive any satellite signals. Confused, we turned the robot off, leaving the GPS on. Almost immediately, the GPS found the satellites and its position. We turned the robot back on, and the GPS quickly lost the satellites.

It appears that the X-1 board emits noise that interferes with the GPS receiver. We attempted to rectify the situation by moving the GPS around on the chassis, shielding the board with aluminum foil, and altering the PIC clock speed. Suspecting that the noise might be going to the GPS via the serial cable, we disconnected the cable. Unfortunately, the noise remained. A slower clock speed combined with extensive foil shielding seemed to help a little bit, but not enough.

We guessed that distancing the GPS from the rest of the robot would decrease the interference. When the GPS was held more than four feet from the board, it captured the satellite signals and calculated position information.

In this configuration, with the GPS held away from the robot by a human, the robot performed as expected. It accepted input, navigated to the destination, and halted upon arrival. Had we access to better shielding, we are confident that the robot would have worked without a human following it around holding the GPS.

5. Bill of Materials

The total cost of this project remained quite low. Our vast collection of electronic devices significantly curtailed our required spending. As shown in Table 1, our total outlay was quite reasonable.

Item	Quantity	Price (each)	Cost to us (total)
PIC16F877A	1	$10	0
X-1 board	1	100	0
IRF540 power MOSFET	2	1	2
Misc. wire	3ft	1	0
Lego Mindstorms motor	2	10	0
Lego chassis	1	20	0
GPS receiver	1	100	0
GPS interface cable	1	20	20
9.6v rechargeable battery	1	20	0
Coaxial barrel-type power connector	1	1	1
		Total	$23

Table 1: Bill of Materials

6. User Manual

To make the car navigate, the user enters the desired number of feet to go in both a north/south and east/west direction. After that data is entered, the car will begin to move. It will continue to drive, occasionally stopping to turn and correct its course, until it reaches its commanded destination. In addition to halting, the car will display a message upon arrival at its destination.

1	2	3	N/E
4	5	6	S/W
7	8	9	
0	B	E	

Figure 1: Keypad Layout

When the car is powered on, the LCD prompts the user to enter the destination. Figure 1 shows the layout of the keys on the car for data input. Table 2 describes the key usage.

Key	**Function**
0-9	Numeric Data Entry
B	Backspace
E	Enter
N/E	Direction; North during Latitude entry, East during Longitude entry
S/W	Direction. South during Latitude entry, West during Longitude entry

Table 2: Keypad Functions

The first information that must be entered is the north/south delta. Use the numeric keypad to enter the number of feet you wish to travel (it must be less than 255). If you make a mistake you can press the backspace key to correct it. After entering the magnitude of the direction, press the North or South button to indicate if the direction is North or South. After all the information is entered correctly press Enter.

You will then be prompted for east/west delta. The procedure to enter this data is the same as for the north/south data, except the directions will be East and West.

After the complete destination is entered, the car will attempt to read a signal from the GPS unit. The screen will begin showing the current and destination positions once the position is known. If the screen remains blank at this point, the GPS is not receiving good position data. Ensure that there is a clear path to the sky if this occurs.

If the data is good, the LCD will display pertinent information, as shown in Table 3 and Table 4. The location information in displayed in terms of minutes of degrees, accurate to one thousandth of a minute. On screen, a decimal point is implied in to the right of the second most significant digit.

The speed number format is the 2's compliment number of the speed, where a negative number indicates the direction is South or West and a positive number means North or East.

AAAAA	BBBBB	EEEEE
CCCCC	DDDDD	EEEEE

Table 3: LCD Status Screen Organization

Number Group	Meaning
AAAAA	Current Latitude coordinate, in thousandths of minutes
BBBBB	Current Longitude coordinate, in thousandths of minutes
CCCCC	Destination Latitude coordinate, in thousandths of minutes
DDDDD	Destination Longitude coordinate, in thousandths of minutes
EEEEE	Current speed in north/south direction
FFFFF	Current speed in east/west direction

Table 4: LCD Screen Symbol Key

The LCD will continually update as the car moves. If for some reason the car looses the GPS signal, it will stop.

Every ten seconds, the car will stop and turn to realign itself with the destination.

After the car has traveled to its desired location, it will stop and display a message on the LCD indicating that it has finished. The car is limited by the accuracy of the GPS, about six feet. The implication of this accuracy limitation is that the car may halt up to six feet from the true destination.

Up to and including the time of arrival at the destination, the robot car's reset button may be pushed to cancel the current navigation and enter a new destination.

7. Conclusions and Recommendations

If we were able to solve the problem of electromagnetic interference from the X-1 board, we feel that we could have made a fully functioning GPS-navigated car. All of the individual parts were implemented and tested successfully separately. When we did simulated driving (by holding the GPS unit away from the car), the car shut off its motors at the correct location.

Although we were unable to make the car fully functional as a self-contained unit, we still implemented all the parts of the program and utilized many of the ideas we covered in class: RS232 communication was used to gather data from the GPS unit; the keypad was used for gathering information from the user; the LCD displayed information useful to the user and for debugging purposes. We were also able to interface with external motors and construct a physical chassis to carry our electronics.

Our project would have benefited if we could had used a board that we could modify. Since we used the X-1 board, which belongs to Rose-Hulman, we were unable to solder anything to it. This meant that many of wires were connected to the board in an insecure manner. Had we been able to solder wires to the X-1, the car would have been more durable.

Were we to do the project over, we would build a complete prototype earlier in the development phase. We built our project in modular components, testing each one as we went along. It was not until the near the end of the project that we mounted all of the parts onto the car as a single unit. If we had assembled the components into a complete system earlier in the project, we would have noticed the electromagnetic interference earlier, which might have allowed us to fix the problem.

8. References

Reference Material for ECE430 Microcomputers. Compiled by Jianjian Song. Sept 2003.

http://www.convict.lu/Jeunes/Math/arctan.htm (Polynomial approximation of arctangent)

Appendix A. Commented Source Code

Main.c

```
//
// ECE430 – Self-navigating Lego car
// Jeff Keacher & Joey Richey
// November, 2003
//
// Coded for PIC16F877A
//
//******************
//    Includes
//******************
#include <pic.h>
#include "lcd.h"
#include "key.h"

//******************
//    Constants
//******************

//string literals used throughout program
#define LOCATION_PROMPT       "Destination:"
#define GETTING_SIGNAL        "Acquiring signal..."
#define DESTINATION_REACHED   "At Location!"
```

```
//defines if we are entering the latitude or longitude
#define LATITUDE  0
#define LONGITUDE 1

#define LAT_FOOT_PER_MINUTE         6     //number of feet in 1 minute
of latitude
#define LON_FOOT_PER_MINUTE         9     //number of feet in 1 minute
of longitude at 39 degrees

#define NUM_DIGITS      3     //of how many digits you can enter
#define DIRECTION 3     //the location in the array which holds the
direction

//direction constatns
#define     NORTH 'N'
#define     SOUTH 'S'
#define     EAST  'E'
#define     WEST  'W'

#define BLANK     ' '

#define NUM_LOOPS_PER_TURN    10
#define NO_DATA_WAITING       0
#define SET                         1
#define CLEAR                       0
#define ENABLED               1
#define DISABLED              0
#define CLOCK_SPEED_MHZ       20
#define SERIAL_RATE_BPS       9600
#define SENTENCE_START        '@'
#define NULL_DATA                   '_'
#define FIRST_CHAR                  1
#define ZERO                        0
#define POSITION_LOST         0
#define POSITION_KNOWN        1
#define LAT_MIN_START         17
#define NS_SPEED_DIR          46
#define NS_SPEED_START        49
#define EW_SPEED_DIR          41
#define EW_SPEED_START        44
#define LON_MIN_START         26
#define END_OF_SENTENCE       57
#define ASCII_NUM_OFFSET      0x30
#define MS_PER_DEGREE         175

//******************
//    hardware/software interface
//******************

//ports that the motors are on
#define MOTOR_PORT_CTRL             TRISC
#define MOTOR_PORT_DIRECTION  0b11101011
#define RIGHT_MOTOR                       RC2               //port for
thr right motor
#define LEFT_MOTOR                        RC4               //port for
the left motor
```

```
#define MOTOR_PORT                       PORTC
#define MOTORS_OFF                       0b11101011
#define MOTORS_ON                  0b00010100

//defintions of our keys
#define     BACKSPACE   KEY14
#define RETURN          KEY15
#define NE_KEY          KEY04
#define SW_KEY          KEY08

#define KEY_DIGIT_1     KEY01
#define KEY_DIGIT_2     KEY02
#define KEY_DIGIT_3     KEY03
#define KEY_DIGIT_4     KEY05
#define KEY_DIGIT_5     KEY06
#define KEY_DIGIT_6     KEY07
#define KEY_DIGIT_7     KEY09
#define KEY_DIGIT_8     KEY10
#define KEY_DIGIT_9     KEY11
#define KEY_DIGIT_0     KEY13

//register to get data from the serial port
#define SERIAL_DATA_REG RCREG
#define SERIAL_FLAG           RCIF


//******************
//    Function Prototypes
//******************
char digits_to_num(char *digits);
void num_to_digits(unsigned int num, char* digits);
void init_interrupt(void);
void keypress_callback(char key);
void update_LCD(void);
void strcat(char *dest, char *src);
void gps_add(unsigned int *ret_lat, unsigned int *ret_lon);
void receive_byte(char*);
void initialize_serial(void);
void serial_buffer(void);
void check_position(void);
char ASCIItoNum(char);
void turn_car(int);
int arctan (int,int);
int my_abs (int);
void done (void);
void stop_motors(void);
void start_motors(void);

//******************
//    Global Variables
//******************
//used for when user is entering input
char location[4];
char cur_location;      //the zero based location of the current digit
you are editing
char input_type;
char loopindex;
```

```
//change in latitude and longitude that user entered
char delta_lat;
char delta_lon;

//the lat/lon coordinates of our destination location
bank1 unsigned int dest_lat;
bank1 unsigned int dest_lon;

//our current lat/lon coordintates
unsigned int cur_lat;
unsigned int cur_lon;

int _ns_speed; // Signed north/south velocity (negative number =>
south)
int _ew_speed; // Signed east/west velocity (negative number => west)
char _good_data; // A flag, zero when position stream is lost, and one
when position is known
char _position_updated; // A flag, which when set indicates that the
position has been updated

void main() {

      char digits[6];

      //turn off the motors, just incase they are on
      MOTOR_PORT_CTRL = MOTOR_PORT_DIRECTION;
      MOTOR_PORT &=MOTORS_OFF;

      LCD_init();
      init_interrupt();

      //get location (cannot put this in a function because it
overflows the stack
      for (loopindex=0;loopindex<4;loopindex++) {
            location[loopindex] = BLANK;
      }

      write_char_to_LCD(0, START_LINE_1); // start on line 1
      write_string_to_LCD(LOCATION_PROMPT);
      input_type = LATITUDE;

      while (RBIE == ON) {
            update_LCD();
      }
      //end get location

      //calculate our destination location
      gps_add(&dest_lat, &dest_lon);

      write_char_to_LCD(0, CLEAR_DISPLAY);
      write_string_to_LCD(GETTING_SIGNAL);

      // Initialize the serial port
      initialize_serial();

      while (ON)
      {
            if (_position_updated) {
                  //for latitude: write our current location, followed
by our
                  //destination, follow by our speed
                  write_char_to_LCD(0, START_LINE_1);
                  num_to_digits(cur_lat, digits);
                  write_string_to_LCD(digits);
                  write_char_to_LCD(1, ' ');
                  num_to_digits(cur_lon, digits);
                  write_string_to_LCD(digits);
                  write_char_to_LCD(1, ' ');
                  num_to_digits((unsigned int)_ns_speed, digits);
                  write_string_to_LCD(digits);

                  //for longitude on line 2: write our current
location, followed by our
                  //destination, follow by our speed
                  write_char_to_LCD(0, START_LINE_2);
                  num_to_digits(dest_lat, digits);
                  write_string_to_LCD(digits);
                  write_char_to_LCD(1, ' ');
                  num_to_digits(dest_lon, digits);
                  write_string_to_LCD(digits);
                  write_char_to_LCD(1, ' ');
                  num_to_digits((unsigned int)_ew_speed, digits);
                  write_string_to_LCD(digits);
            }
      check_position();
      }
}

//
// Routine to handle all interrupts and call the appropriate one
//
#pragma interrupt_level 1
void interrupt inter () {
      // interupt handler
      if (RBIF && PEIE != ON) {
            handle_keyevent(keypress_callback);
      }
      // If the serial port has data, call the serial buffer function
      if (SERIAL_FLAG && PEIE)
      {
            serial_buffer();
            SERIAL_FLAG = NO_DATA_WAITING;
      }
}

//
//    Initilize the interrupts that we need for the program
//
void init_interrupt(void) {
      //    key strokes
      PORTB = 0;        //ckear portb
      RBPU = OFF;       //enable PORTB pullup resistors
      TRISB = ENABLE_KEYPAD_INTERRUPT;    // Set RB3-0 as output
      PORTB = ENABLE_KEYPAD_INTERRUPT;

      RBIE = ON;         // turn on RB PORT Change interrupt
      RBIF = OFF;        // Clear interrupt flag
      GIE = ON;          // enable global interupt

}

//
//    This function is called after a key has been pressed (and 
succesfully debounced)
//    The parameter is the key that is pressed as defined in key.h--
KEY01-KEY16
void keypress_callback(char key) {
      char value;

      if (key == NO_KEY_PRESSED) {
            return;
      } else if (key == BACKSPACE) {
            if (cur_location == 0) {
                  location[cur_location] = BLANK;
            } else {
                  location[cur_location - 1] = BLANK;
                  cur_location--;
            }
      } else if (key == RETURN) {
            if (location[DIRECTION] != BLANK) {
                  if (input_type == LATITUDE) {
                        //save the state of longitude
                        delta_lat = digits_to_num(location);
                        for (loopindex=0;loopindex<4;loopindex++) {
                              location[loopindex] = BLANK;
                        }
                        cur_location = 0;
                        input_type = LONGITUDE;
                  } else { //longitude
                        //save the state of the latitude
                        delta_lon = digits_to_num(location);
                        //done getting input, don't listen for key 
presses anymore
                        RBIE = OFF;
                  }
            }
      } else if (key == NE_KEY) {
            if (input_type == LONGITUDE) {
                  location[DIRECTION] = EAST;
            } else {
                  location[DIRECTION] = NORTH;
            }
      } else if (key == SW_KEY) {
            if (input_type == LONGITUDE) {
                  location[DIRECTION] = WEST;
            } else {
                  location[DIRECTION] = SOUTH;
            }
      } else { // a number
```

```
            if ((cur_location < NUM_DIGITS -1) || (location[2] ==
BLANK))  {
                  switch (key) {
                        case KEY_DIGIT_1:
                              value = '1';
                              break;
                        case KEY_DIGIT_2:
                              value = '2';
                              break;
                        case KEY_DIGIT_3:
                              value = '3';
                              break;
                        case KEY_DIGIT_4:
                              value = '4';
                              break;
                        case KEY_DIGIT_5:
                              value = '5';
                              break;
                        case KEY_DIGIT_6:
                              value = '6';
                              break;
                        case KEY_DIGIT_7:
                              value = '7';
                              break;
                        case KEY_DIGIT_8:
                              value = '8';
                              break;
                        case KEY_DIGIT_9:
                              value = '9';
                              break;
                        case KEY_DIGIT_0:
                              value = '0';
                              break;
                        default:
                              value = BLANK;
                  }
                  location[cur_location] = value;
                  cur_location++;
            }
      }
}

//
//    Updates the LCD while input is being entered
//
void update_LCD() {
      write_char_to_LCD(0, START_LINE_2); // start on line 2
      write_char_to_LCD(1, location[0]);
      write_char_to_LCD(1, location[1]);
      write_char_to_LCD(1, location[2]);
      write_char_to_LCD(1, ' ');
      write_char_to_LCD(1, location[DIRECTION]);

}

//
//converts an interger to an array of characters
```

```
//
void num_to_digits(unsigned int num, char* digits) {
      unsigned int factor = 10000;
      char i = 0;
      digits[5] = '\0';

      while (factor > 0) {
            digits[i] = num/factor + '0';
            num %= factor;
            i++;
            factor /=10;
      }
}

//
//converts a character array to 3 numbers to its corresponding integer
value
//
char digits_to_num(char *digits) {
      int i;
      char factor = 1;
      char retval=0;

      for (i=2;i>=0;i--) {
            if (digits[i] != BLANK) {
                  retval+= (digits[i]-'0') * factor;
                  factor *=10;
            }
      }
      return retval;
}

//
//    Adds the offset provided by the user to the current location, so
that we know where we need to stop at
//
void gps_add(unsigned int *ret_lat, unsigned int *ret_lon) {
      while(cur_lat == 0);
      *ret_lat = cur_lat + (delta_lat/LAT_FOOT_PER_MINUTE);
      *ret_lon = cur_lon + (delta_lon/LON_FOOT_PER_MINUTE);
}

// check_position: Compare the current position to the desired
position.  Adjust course accordingly.
//   Uses _position_updated flag to determine if the position has been
updated since the last time it
//   was called.  This prevents the function from continually
correcting course based on a previous location.
void check_position(void)
{

  int ns_diff; // difference between source and destination
  int ew_diff; // difference between source and destination
  int bearing_dir;  // the direction for the bearing
  int cur_dir; // the current direction
  static char turn_count = 0;
  if (_good_data == CLEAR)
  {
    // We don't have a solid location. Stop the motors.
    stop_motors();
  } else
  {
    // We have a solid location.  Start the motors.
    start_motors();
  }

  // Only check the position if it has changed since last time
  if (_position_updated == SET)
  {
    // Find the angle between the direction we're going and the
direction we need to go
    // First, find the bearing from the current location to the
destination
    ns_diff = dest_lat - cur_lat; // amount north we need to go
    ew_diff = dest_lon - cur_lon; // amount east we need to go

    // Check if we are at our destination
    if (ns_diff == ZERO && ew_diff == ZERO)
    {
      // If we made it, we're done.
      done();
    }

    if (ew_diff != ZERO) // we don't want to divide by zero
    {
      bearing_dir = arctan(ns_diff,ew_diff);
    }

    // Second, find the current direction (this parallels the above as
long as we're relatively
    // close to the equator [things basically square])
    if (_ew_speed != ZERO)
    {
      cur_dir = arctan(_ns_speed,_ew_speed);
    }

    if (++turn_count % NUM_LOOPS_PER_TURN == 0) {
          turn_count = 0;
          // Finally, turn the car to the desired angle by turning it
an amount equal to
          // the difference between the two angles
          turn_car(cur_dir - bearing_dir);
      }
  }

  // Clear the _position_updated flag so that we don't run this again
unnecessarily
  _position_updated = CLEAR;

}

// done: the car has found its destination
void done (void)
{
  // Found the destination

  // Display "Done" on the LCD
  write_char_to_LCD(0, CLEAR_DISPLAY);
  write_string_to_LCD(DESTINATION_REACHED);

  // Stop.
  stop_motors();

  // Loop forever
  while (ON) {};

}

// stop_motors: All stop.
void stop_motors(void)
{
  // stop the motors (both of them)
      MOTOR_PORT &=MOTORS_OFF;
}

// start_motors: All start.
void start_motors(void)
{
      // start the motors (both of them)
      MOTOR_PORT |=MOTORS_ON;
}

// turn_car: turn the car the specified number of degrees
void turn_car(int angle)
{
  unsigned int i;
  // Relate the angle to a time
  // Use the MS_PER_DEGREE constant to determine the number of
miliseconds to be turning

  LEFT_MOTOR = OFF;
  for(i=0;i<MS_PER_DEGREE*angle;i++) {
        wait_15us();
  }
  LEFT_MOTOR = ON;

}

// arctan: Returns a rough approximation of the arctangent of the
parameter, in degrees.
int arctan (int numerator, int denominator)
{
  int work = 0;
  int ratio = 0;
  int need90shift = CLEAR; // set if the result must be shifted by 90
degrees

  if (denominator != ZERO ) // We don't want to divide by zero
  {
    if ( my_abs(denominator) > my_abs(numerator) )
    {
      ratio = my_abs(denominator / numerator);
      need90shift = SET;
    } else
    {
      ratio = my_abs(numerator / denominator);
    }

    // This algorithm is based off a PASCAL routine from
http://www.convict.lu/Jeunes/Math/arctan.htm
    work = ((-150 + 310*ratio - (ratio*ratio/2) - (ratio*ratio/3))/50 +
5)/10;

    if (need90shift == SET)  // If the angle is above 45 degrees, it
needs special treatment
    {
      work = 90 - work;
    }

    if (numerator >= 0 && denominator > 0)
    {
      // first quadrant
      work = 90 - work;
    } else if (numerator >= 0 && denominator < 0 )
    {
      // In second quadrant
      work = 270 + work;
    } else if (numerator < 0 && denominator < 0 )
    {
      // third quadrant
      work = 270 - work;
    } else if (numerator < 0 && denominator > 0 )
    {
      // fourth quadrant
      work = 90 + work;
    }

  } else {
    // Case if the denominator is zero (the angle is 0 or 180 degrees
(compass direction)

    if (numerator >= 0) // Or equal, so that it goes straight
    {
      // Straight ahead
      work = 0;
    } else
    {
      // Otherwise, must be straight behind
      work = 180;
    }

  }
  return work;

}

// my_abs: cheap absolute value function
int my_abs(int input)
```

```
{
  if (input >= 0)
  {
    return input;
  } else
  {
    return -input;
  }
}

// serial_buffer: Read in a sentence of the Garmin "simple text"
position data
void serial_buffer(void)
{
  // Local static variables
  // sentence_position: Current character being worked on in the Simple
Text sentence.  One-index.
  // A value of zero means that the position is not yet known (as when
the device first starts up)
  // The value is also set to zero when the null position character is
received from the GPS ('_')
  // indicating that the GPS has lost its position.
  static unsigned char sentence_position = ZERO;

  // These position variables are kept private until a complete new
location is known, at
  // which point, they are copied into the global position variables
  static unsigned int flip_lat_minutes;
  static unsigned int flip_lon_minutes;
  static char flip_ns_speed; // Signed north/south velocity (negative
number => south)
  static char flip_ns_speed_dir;
  static char flip_ew_speed; // Signed east/west velocity (negative
number => west)
  static char flip_ew_speed_dir;

  // Normal local variables
  char cwork; // Character pulled from serial port that has yet to be
processed

  // Get the byte of data from the serial port
  receive_byte(&cwork);
 // write_char_to_LCD(1, cwork);

  // Check if there is a new sentence
  if (cwork == SENTENCE_START)
  {
    _good_data = POSITION_KNOWN;
    sentence_position = FIRST_CHAR;
  }

  // Check if the position is lost
  if (cwork == NULL_DATA)
  {
    // First, set the flag so that the motors stop
    _good_data = POSITION_LOST;
    // Next, set the sentence position to zero
    sentence_position = ZERO;
  }

  // Convert ASCII to a number
  cwork = ASCIItoNum(cwork);

  // Switch based on sentence position
  switch (sentence_position)
  {
    case LAT_MIN_START:
         flip_lat_minutes = 10000*cwork;
         break;
    case LAT_MIN_START+1:
         flip_lat_minutes += 1000*cwork;
         break;
    case LAT_MIN_START+2:
         flip_lat_minutes += 100*cwork;
         break;
    case LAT_MIN_START+3:
         flip_lat_minutes += 10*cwork;
         break;
    case LAT_MIN_START+4:
         flip_lat_minutes += cwork;
         break;
    case LON_MIN_START:
         flip_lon_minutes = 10000*cwork;
         break;
    case LON_MIN_START+1:
         flip_lon_minutes += 1000*cwork;
         break;
    case LON_MIN_START+2:
         flip_lon_minutes += 100*cwork;
         break;
    case LON_MIN_START+3:
         flip_lon_minutes += 10*cwork;
         break;
    case LON_MIN_START+4:
         flip_lon_minutes += cwork;
         break;
    case EW_SPEED_DIR:
         flip_ew_speed_dir = cwork;
         break;
    case EW_SPEED_START:
         flip_ew_speed = 10 * cwork;
         break;
    case EW_SPEED_START+1:
         flip_ew_speed += cwork;
         break;
    case NS_SPEED_DIR:
         flip_ns_speed_dir = cwork;
         break;
    case NS_SPEED_START:
         flip_ns_speed = 10 * cwork;
         break;
    case NS_SPEED_START+1:
         flip_ns_speed += cwork;
```

```
            break;
      case END_OF_SENTENCE:
            // If we are at the end of the sentence, copy the temporary
position values into the
            // the global position variables
            cur_lat = flip_lat_minutes;
            cur_lon = flip_lon_minutes;
            if (flip_ew_speed_dir == SET)
            {
              // East
              _ew_speed = flip_ew_speed;
            } else
            {
              // West
              _ew_speed = -flip_ew_speed;
            }
            if (flip_ns_speed_dir == SET)
            {
              // North
              _ns_speed = flip_ns_speed;
            } else
            {
              // South
              _ns_speed = -flip_ns_speed;
            }
            _position_updated = SET;
      default:
            break;
  }

  // Increment the sentence position
  sentence_position++;

}

// ASCIItoNum:  Converts the ASCII representation of a number to a
binary representation
// Note: no boundary checking
char ASCIItoNum(char input)
{
  char localWork;

  localWork = input - ASCII_NUM_OFFSET;

  return localWork;
}

// initialize_serial: Sets up the serial port to be able to receive
//
void initialize_serial(void)
{

      // Set the flags
      SPEN = ENABLED;   // serial port
      SYNC = DISABLED;
      BRGH = ENABLED; // high speed
      RX9 = DISABLED;
      CREN = ENABLED;   // continuous mode
      RCIE = ENABLED; // enable interrupts

      // NOTE: No data direction need be specified, as PORTC
      // defaults to being an input

      // Set up the baud rate
      SPBRG = (1000000/SERIAL_RATE_BPS*CLOCK_SPEED_MHZ - 16)/16;

      PEIE = ON;        // peripheral interrupts

      return;
}

// receive_byte: Returns a byte from the serial port in the
//                            function parameter pointer
//
void receive_byte(char* pSerialByte)
{
      // Wait for a byte to be received
      while (SERIAL_FLAG == NO_DATA_WAITING) {};

      // When there's a byte ready, copy it
      *pSerialByte = SERIAL_DATA_REG;
      PORTD = SERIAL_DATA_REG;
      // Clear the byte-received flag
      SERIAL_FLAG = NO_DATA_WAITING;

      return;

} //receive_byte()
```

LCD.c

```
#include "lcd.h"

//=====================================================
// Initialize_LCD()
//=====================================================
void LCD_init() {
      unsigned int i; //loop count
      TRISD = PORTD_DIRECTION;
      TRISE = PORTE_DIRECTION;
      ADCON1      = 0x02;     // configure all RE pins to be digital
pins
      LCD_RW      = 0;  // write to LCD only
      LCD_E = 0;  // pull down E line

      TMR2ON      = 1;              // Turn on Timer2

      // initialize LCD
      #ifndef DEBUG
            //24.51 millisecond delay
            for(i=0;i<3500;i++) {
                  wait_15us();
            }
      #endif
      write_char_to_LCD(0,0x38);    // write 0x38 to LCD

      #ifndef DEBUG
            //14 millisecond delay
            for(i=0;i<2000;i++) {
                  wait_15us();
            }
      #endif
      write_char_to_LCD(0,0x38);    // write 0x38 to LCD

      #ifndef DEBUG
            //4.9 millisecond delay
            for(i=0;i<700;i++) {
                  wait_15us();
            }
      #endif

      write_char_to_LCD(0,0x38);    // write 0x38 to LCD
      // send actual commands
      write_char_to_LCD(0,0x38);    // 8-bit data, 2 lines, 5x7 font
      write_char_to_LCD(0,0x01);    // clear display, return cursor to
home
      write_char_to_LCD(0,0x0C);    // turn display on, cursor off
      write_char_to_LCD(0,0x06);    // cursor increment
      return;
}

//=====================================================
// wait_15us()
//=====================================================
// use Timer2 to generate 14.80 microsecond delay each time this
routine is called
// at 20 MHz
void wait_15us() {
      #ifndef DEBUG
            TMR2  = 256-53;
            while(TMR2IF==0) {};
      #endif
      return;
}
//=====================================================
// write_string_to_LCD(char *)
//=====================================================
//#pragma interrupt_level 1
void write_string_to_LCD(const char *str) {
      // always write a character to LCD

      char j = 0;
      while (str[j] != '\0') {
            write_char_to_LCD(1, str[j++]);
      }

      return;
} // end write_string_to_LCD()


//=====================================================
// write_char_to_LCD(register, data)
//=====================================================
//#pragma interrupt_level 1
void write_char_to_LCD(char register_select, char data) {
      char i;
      LCD_DATA = data;
      if (register_select == 0) {
            LCD_RS = 0;
      } else {
            LCD_RS = 1; // write data
      }
      #ifndef DEBUG
            // 191.65 usecond delay
            for(i=0;i<40;i++) {
                  wait_15us();
            }
      #endif
      LCD_E = 1;  // pull up E line
      #ifndef DEBUG
            //1.4 millisecond delay
            for(i=0;i<200;i++) {
                  wait_15us();
            }
      #endif
      LCD_E = 0;  // transfer data
      #ifndef DEBUG
            // 43 usecond delay
            for(i=0;i<20;i++) {
                  wait_15us();
            }
      #endif
      return;
}
```

Key.c

```
#include "key.h"

/**
 *    Handles a generic key event and decides which function press to
call
 */
void handle_keyevent(void (*handle_keypress)(char)) {
      char key;
      int i;
      key = check_key_press();
      if (key == NO_KEY_PRESSED) {
            PORTB = ENABLE_KEYPAD_INTERRUPT;
            RBIF = OFF;             //clear the interrupt
            return;
      }

      //debounce keypress
      for(i=0;i<WAIT_DEBOUNCE;i++) {
            continue;
      }
      //check to see if the key is still pressed
      if (key == check_key_press()) {
            (*handle_keypress) (key);
      }
      PORTB = ENABLE_KEYPAD_INTERRUPT;
      RBIF = OFF;             //clear the interrupt
      return;
}

/**
 * Checks to see if a key is pressed and returns the key pressed or -1
if nothing is pressed
 */
char check_key_press() {
      KEYPAD = KEY01;               //check key 1
      if ((KEY01 ^ KEYPAD) == 0) {
            return KEY01;
      }
      KEYPAD = KEY02;               //check key 2
      if ((KEY02 ^ KEYPAD) == 0) {
            return KEY02;
      }
      KEYPAD = KEY03;               //check key 3
      if ((KEY03 ^ KEYPAD) == 0) {
            return KEY03;
      }
      KEYPAD = KEY04;               //check key 4
      if ((KEY04 ^ KEYPAD) == 0) {
            return KEY04;
      }
      KEYPAD = KEY05;               //check key 5
      if ((KEY05 ^ KEYPAD) == 0) {
            return KEY05;
      }
      KEYPAD = KEY06;               //check key 6
      if ((KEY06 ^ KEYPAD) == 0) {
            return KEY06;
      }
      KEYPAD = KEY07;               //check key 7
      if ((KEY07 ^ KEYPAD) == 0) {
            return KEY07;
      }
      KEYPAD = KEY08;               //check key 8
      if ((KEY08 ^ KEYPAD) == 0) {
            return KEY08;
      }
      KEYPAD = KEY09;               //check key 9
      if ((KEY09 ^ KEYPAD) == 0) {
            return KEY09;
      }
      KEYPAD = KEY10;               //check key 10
      if ((KEY10 ^ KEYPAD) == 0) {
            return KEY10;
      }
      KEYPAD = KEY11;               //check key 11
      if ((KEY11 ^ KEYPAD) == 0) {
            return KEY11;
      }
/*    KEYPAD = w = KEY12;                 //check key 12
      if ((w ^ KEYPAD) == 0) {
            return KEY12;
      }*/
      KEYPAD = KEY13;               //check key 13
      if ((KEY13 ^ KEYPAD) == 0) {
            return KEY13;
      }
      KEYPAD = KEY14;               //check key 14
      if ((KEY14 ^ KEYPAD) == 0) {
            return KEY14;
      }
      KEYPAD = KEY15;               //check key 15
      if ((KEY15 ^ KEYPAD) == 0) {
            return KEY15;
      }
/*    KEYPAD = w = KEY16;                 //check key 16
      if ((w ^ KEYPAD) == 0) {
            return KEY16;
      }*/
      return NO_KEY_PRESSED;
}
```

LCD.h

```
#ifndef LCD_H
#define LCD_H

#include <pic.h>

/**   Constants   **/
#define FREQUENCY  20   /* Crystal frequency in MHz*/

/**   Pin Configuration **/

#define PORTD_DIRECTION 0x00  // LCD data pins
#define PORTE_DIRECTION 0x00  // LCD control pins
#define LCD_DATA        PORTD
#define LCD_RS                RE0
#define LCD_E                 RE1
#define LCD_RW                RE2         // 0 for writing to LCD

#define     NUM_LCD_DIGITS    20    //number of digits on the LCD

//constants for write_char_to_lcd
#define START_LINE_1          0x80  //set cursor at line 1
#define START_LINE_2          0xC0  //set cursor at line 2
#define CLEAR_DISPLAY         0x01  //clear display, return cursor to
home


/**   function prototypes     **/
/**
  *   Initilize the LCD
  */
void LCD_init();
```

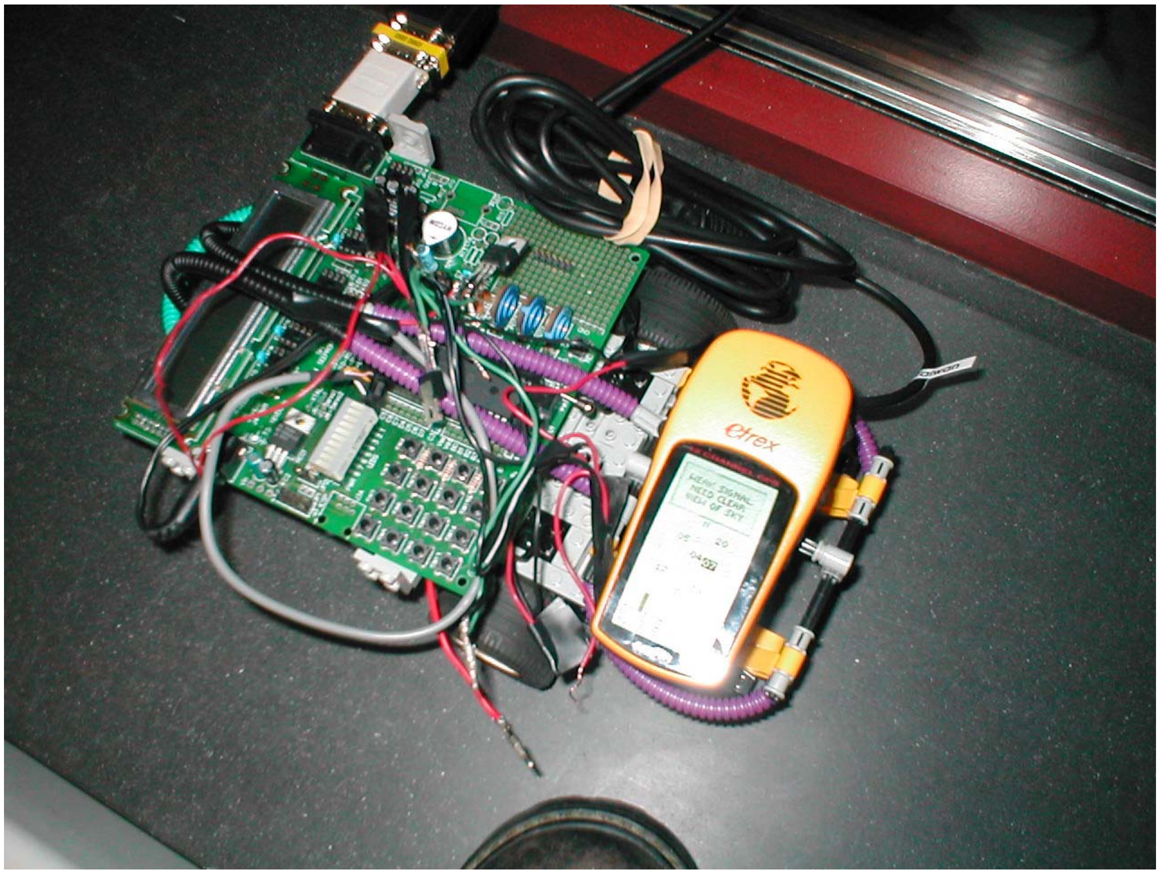
```
/**
  *   Delay loop for 14.80us at 20 Mhz used for LCD initilzation
  */
void wait_15us();

/**
  * Writes a character to the LCD
  * @param register_select
  *         (0 for command, 1 for data)
  * @param data
  *         The character to be written to the LCD
  */
void write_char_to_LCD(char register_select, char data);

/**
  *   Writes a string to the LCD
  * @param str
  *         A null-terminated string to be written to the LCD
  */
void write_string_to_LCD(const char *);
#endif
```

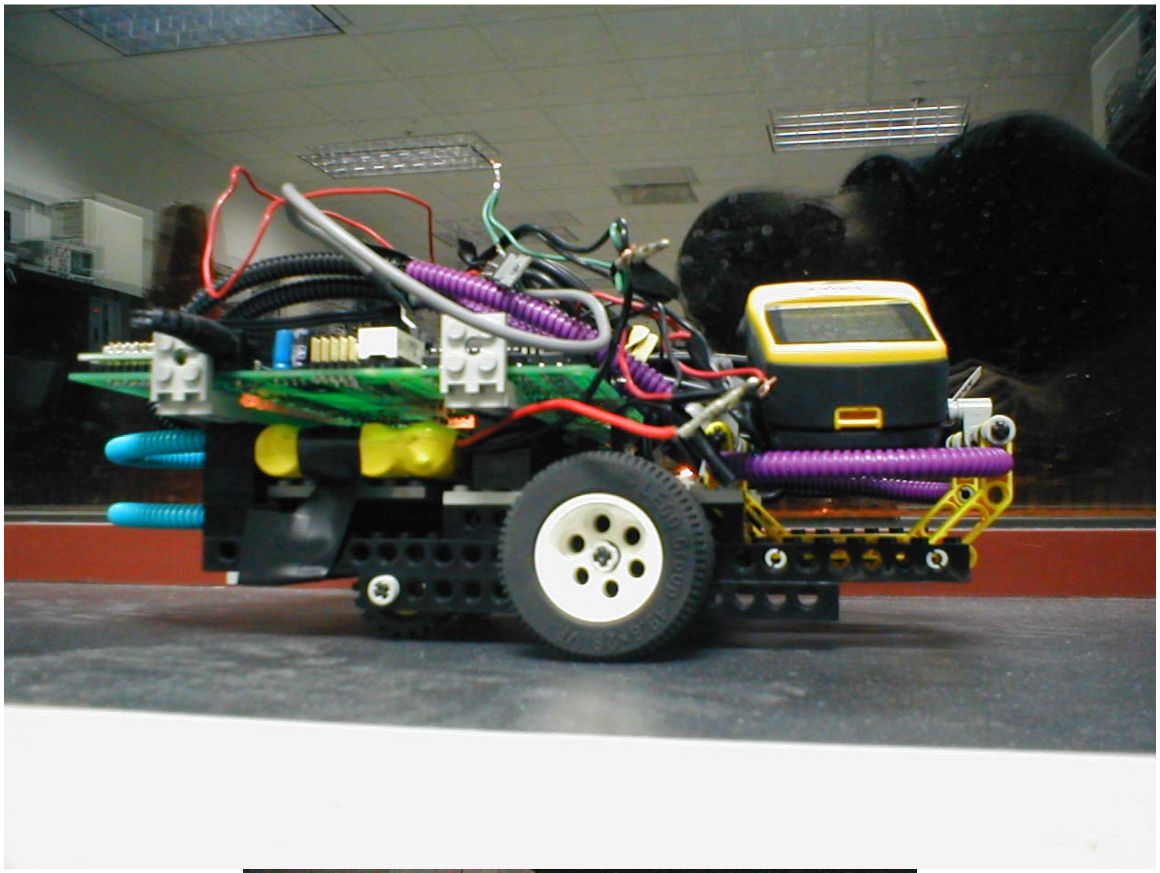
Key.h

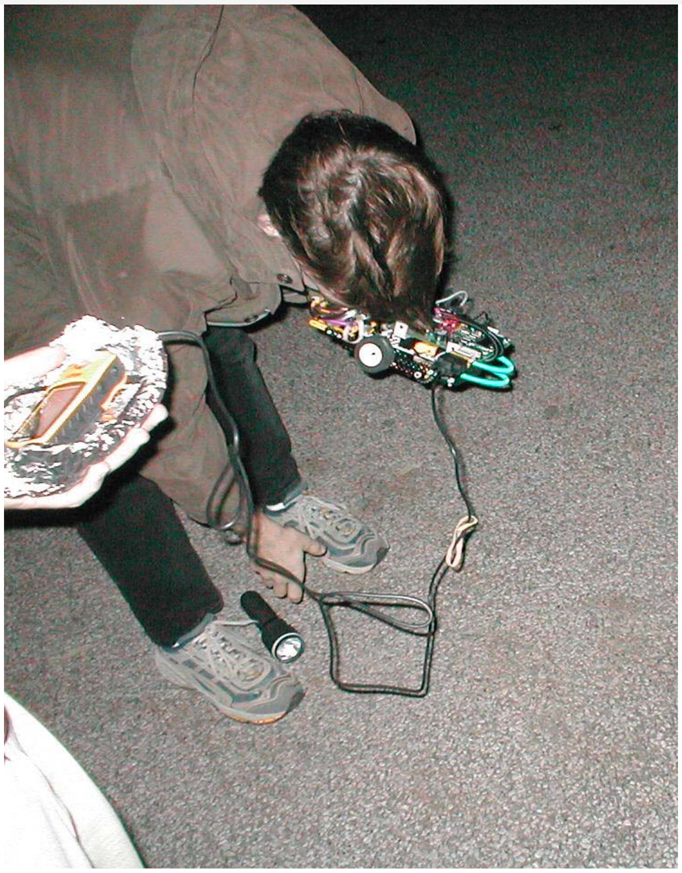
```
#ifndef KEY_H
#define KEY_H

#include <pic.h>

//need for keypressing
#define ENABLE_KEYPAD_INTERRUPT     0b11110000
#define KEYPAD    PORTB


#define ON  1
#define OFF 0

//constants for keypad
#define NO_KEY_PRESSED  0
#define KEY01     0b11101110
#define KEY02     0b11011110
#define KEY03     0b10111110
#define KEY04     0b01111110
#define KEY05     0b11101101
#define KEY06     0b11011101
#define KEY07     0b10111101
#define KEY08     0b01111101
#define KEY09     0b11101011
#define KEY10     0b11011011
#define KEY11     0b10111011
#define KEY12     0b01111011
#define KEY13     0b11100111
#define KEY14     0b11010111
#define KEY15     0b10110111
#define KEY16     0b01110111
```

```
#define WAIT_DEBOUNCE              10000 //delay for debounce keypress


void handle_keyevent(void (*handle_keypress)(char));
char check_key_press(void);

#endif
```

Appendix B.   Photographs







Appendix C. Hardware Schematic

The following page is a hardware schematic of our project.