Autonomous Water Pollutant Cleaning Robot

Abstract:

This product is a floating robot that sucks surrounding water in a large water body (ocean, lake, etc.) and filters the trash which is then stored and transported to the shore. It is also integrated with AI data analysis that can attain very valuable information like where and when are the most waste found on the water. This data can be used to prevent water pollution.

The problem is solved by filtering the surface wastes through suction and then properly disposing them.

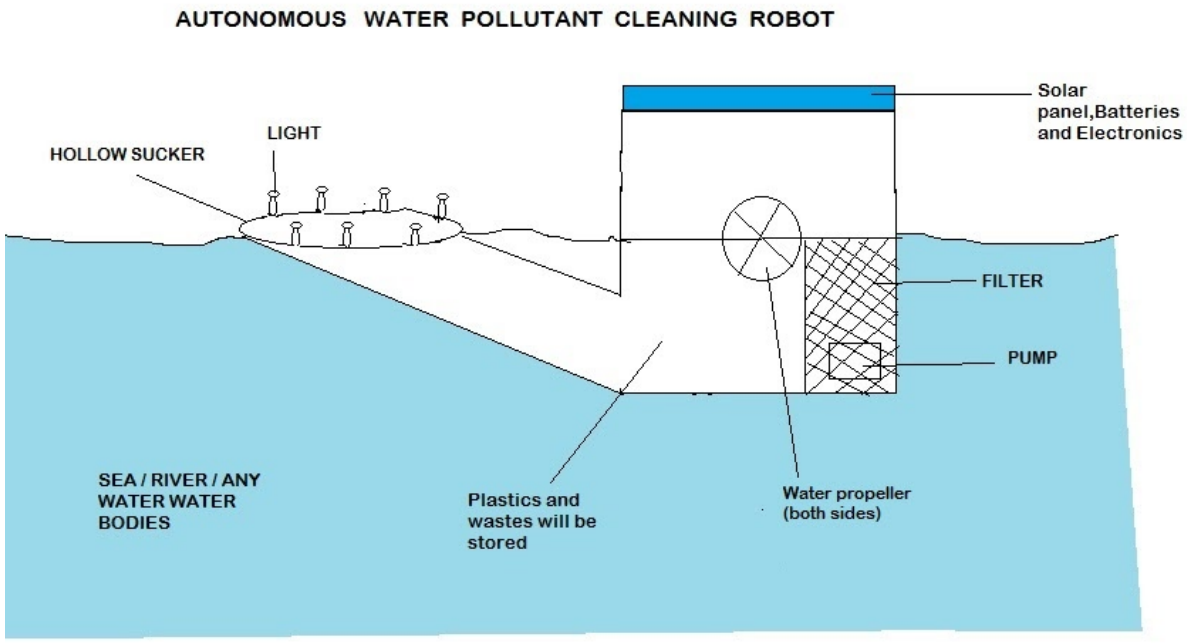
The death and destruction of many species of aquatic animals can be reduced.

Most existing solutions are very large scale. Hence, they are expensive and complex to operate.

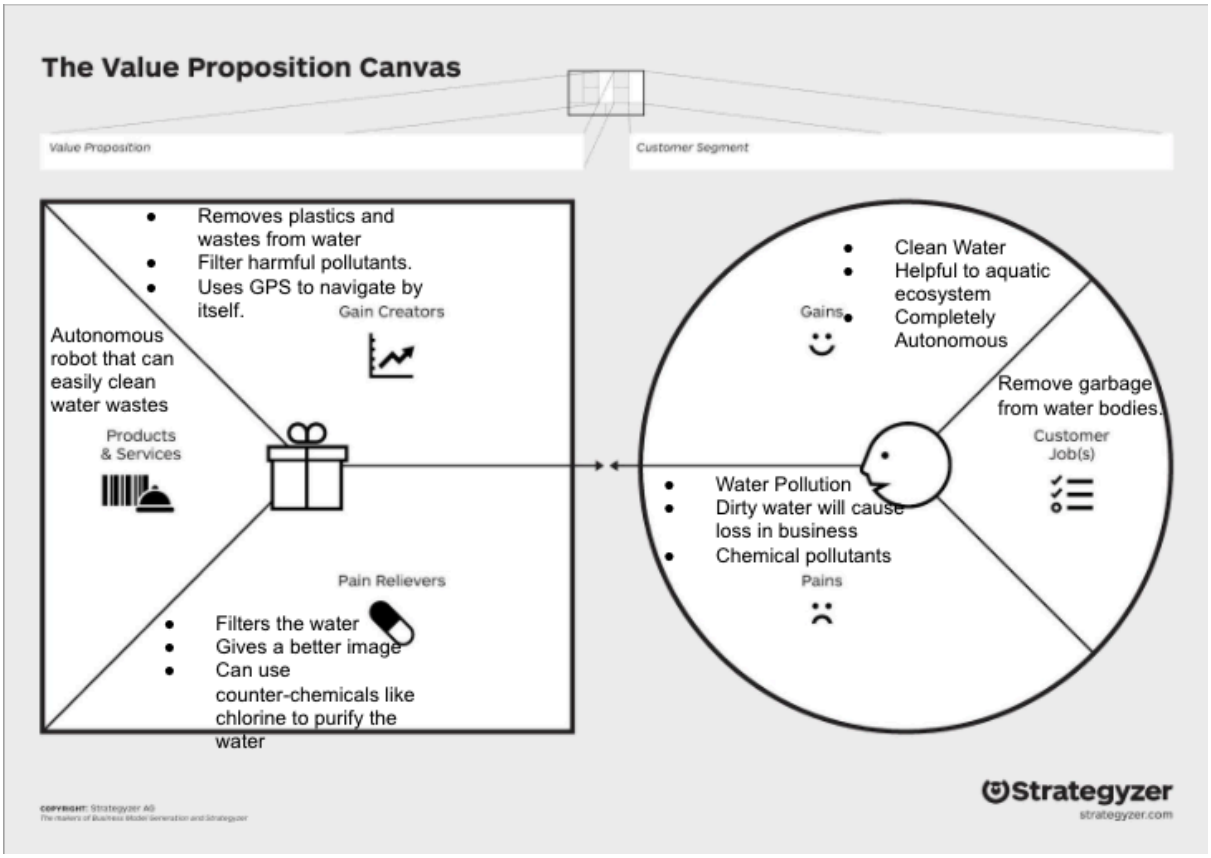
This solution is completely autonomous, compact and uses the power suction to clean the waste.

It integrates AI and object detection to track the pollutants and clean it. It also provides data as to where most pollutants are found using GPS.

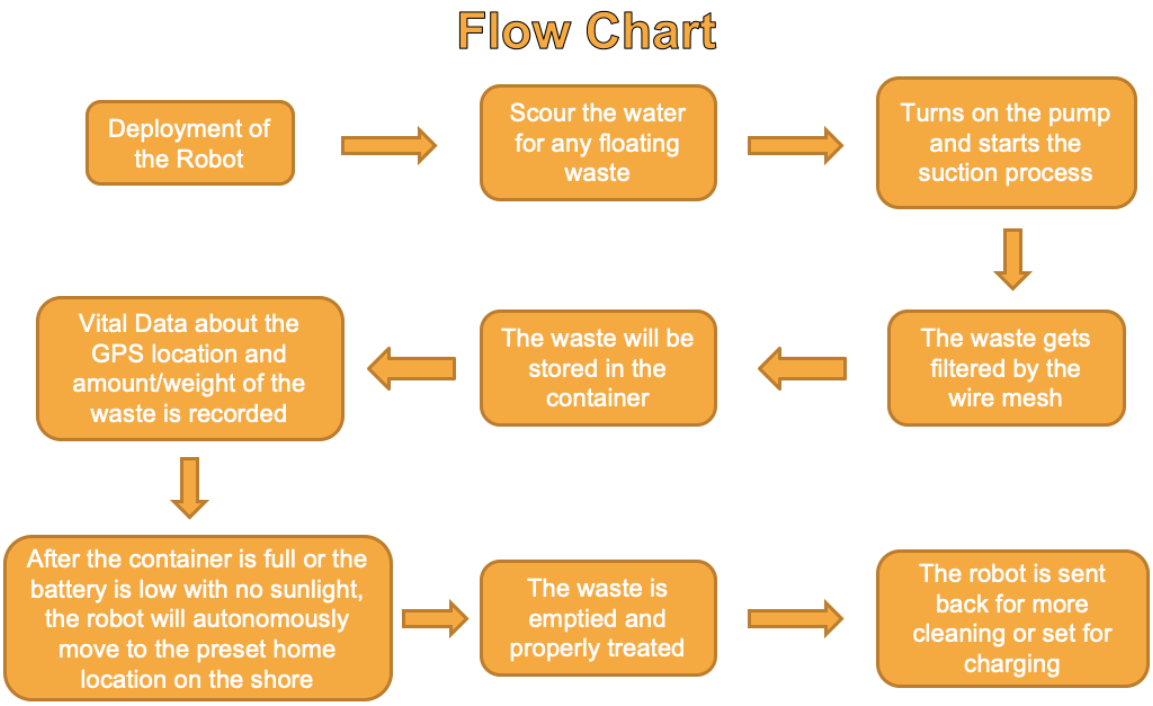
Design Diagram:



Value Proposition Canvas:



Flow Chart:


Flow Chart
Deployment of the Robot
Scour the water for any floating waste
Turns on the pump and starts the suction process
The waste gets filtered by the wire mesh
The waste will be stored in the container
Vital Data about the GPS location and amount/weight of the waste is recorded
After the container is full or the battery is low with no sunlight, the robot will autonomously move to the preset home location on the shore
The waste is emptied and properly treated
The robot is sent back for more cleaning or set for charging

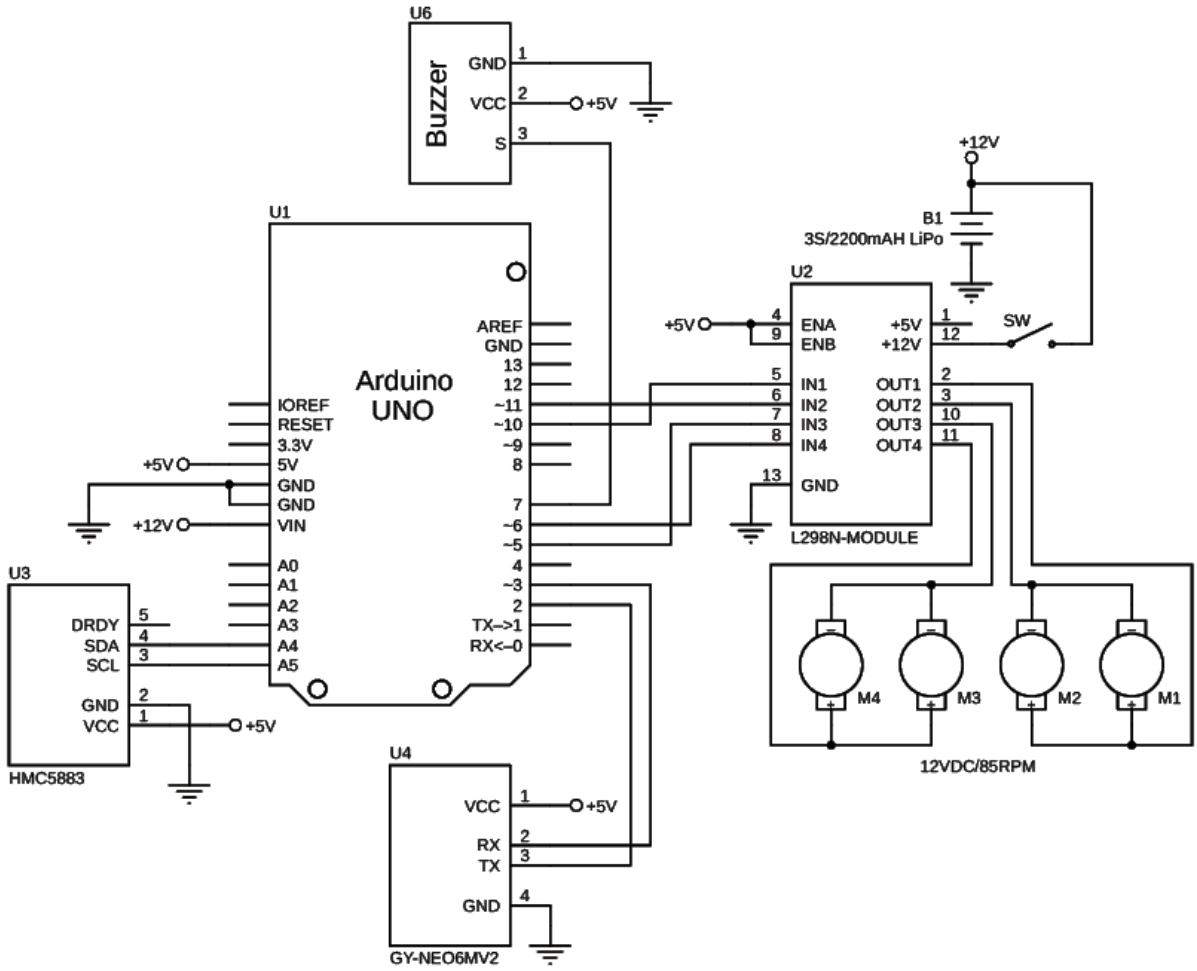
Components Used:

- Arduino Mega 2560 Rev3 Microcontroller
- 12V DC Geared Motors x2
- L298N Motor Driver
- NEO-6M GPS Module with EPROM
- Micro SD Card Module
- 12V 12L/min Water Pump
- 12V 10W Solar Panel
- 12.8V 20AH Li-Po Battery
- Water fin attachments
- Industrial Grade Floatation Rings

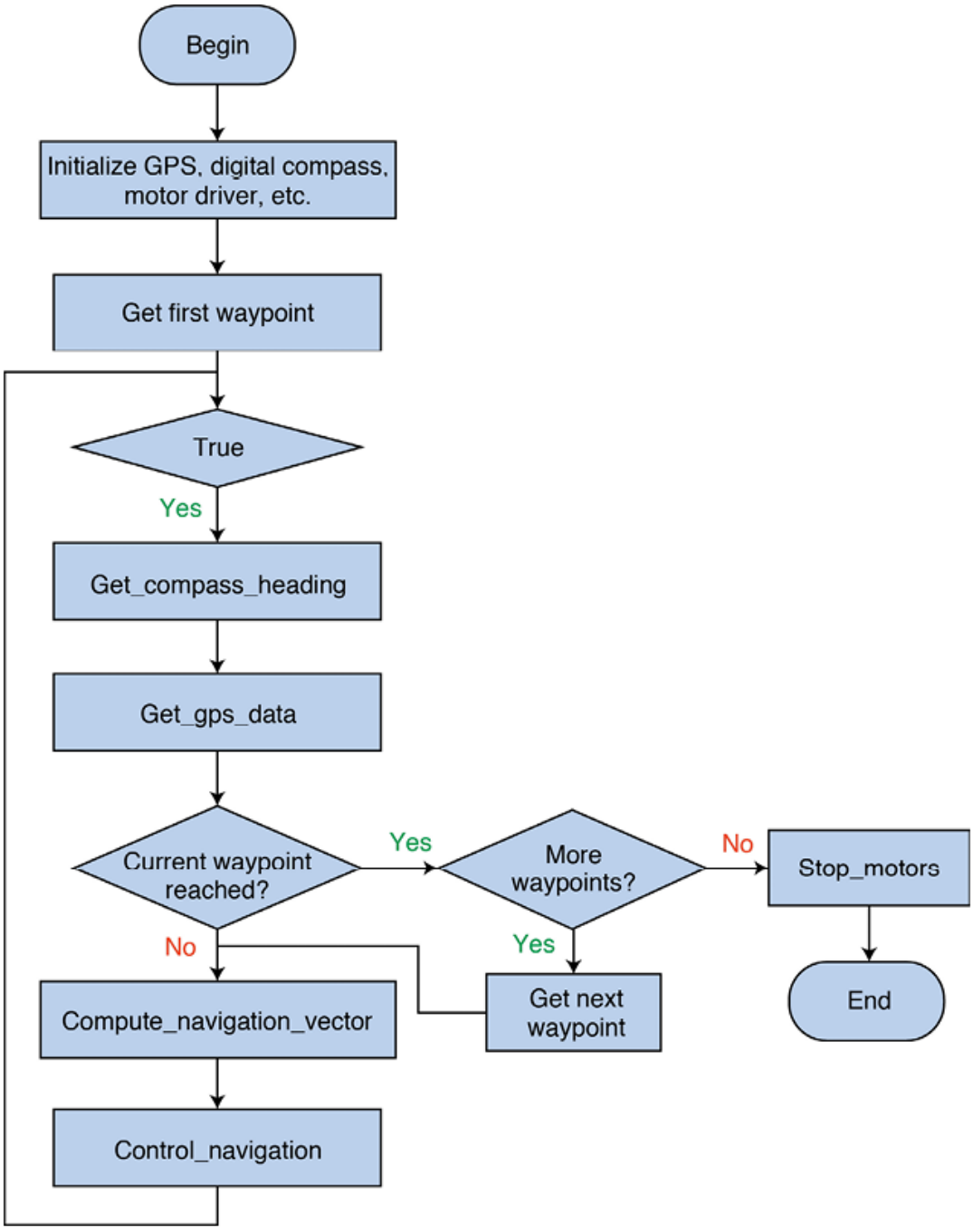
Link

No.	Description	Qty	Unit price	Total price
1	Arduino Mega	1	₹700.00	₹700.00
2	12V DC Geared Motor	2	₹1,133.00	₹2,266.00
3	L298N Motor Driver	1	$125.00	₹125.00
4	NEO-6M GPS Module with EPROM	1	₹700.00	₹700.00
5	Micro SD Card Reader Module	1	₹80.00	₹80.00
6	12V 60W Heavy Duty Water Pump	1	$1,590.00	₹1,590.00
7	12V 10W Solar Panel	1	₹750.00	₹750.00
8	12.8V 20AH Li-Po Battery	1	₹6,250.00	₹6,250.00
9	Enclosure (Acrylic Sheets, 3D Printed Parts, Sealant, etc)	-	$2,000.00	₹2,000.00
10	Floatation	-	₹2,000.00	₹2,000.00
11	Miscellaneous	-	₹1,500.00	₹1,500.00
	Notes:		Subtotal	₹17,961.00
			Adjustments	₹1,000.00
				₹18,961.00

Circuit Diagram:



Step by Step Algorithm:



Program:

I am using the code from an autonomous submarine using an Arduino. It will work perfectly for our own robot with a few changes. This will help it navigate the ocean using GPS and magnetic pointer. The program also includes motor controls that I added.

```
// Ping sensor = 5V, GRND, D11 (for both trigger & echo)
// LCD Display = I2C :  SCL (A5) &  SDA (A4)
// Adafruit GPS = D7 & D8 (GPS Shield, but pins used internally)
// IR Receiver = D5
// Adafruit Magnetometer Adafruit_HMC5883 = I2C :  SCL (A5) &  SDA (A4)
// SD Card (D10, D11, D12, D13)
//


#include <Wire.h>                       // used by: motor driver
#include <Adafruit_MotorShield.h>               // motor driver
#include "utility/Adafruit_PWMServoDriver.h"     // motor driver
#include <NewPing.h>                    // Ping sonar
#include <LiquidCrystal_I2C.h>                // LCD library
#include <Adafruit_Sensor.h>                 // part of mag sensor
#include <Adafruit_HMC5883_U.h>                 // mag sensor
#include <waypointClass.h>                  // custom class to manaage GPS waypoints
#include <Adafruit_GPS.h>                   // GPS
#include <SoftwareSerial.h>                 // used by: GPS
#include <math.h>                       // used by: GPS
#include <moving_average.h>                  // simple moving average class; for Sonar functionality
//#include <PString.h>                    // PString class, for "message" variable; LCD display


// Select optional features
// COMMENT OUT IF NOT DESIRED, don't just change to "NO"
#define USE_GRAPHING YES          // comment out to skip graphing functions in LCD display
#define USE_LCD_BACKLIGHT YES      // use backlight on LCD; commenting out may help in direct sunlight
#define DEBUG YES                // debug mode; uses Serial Port, displays diagnostic information, etc.
//#define USE_IR YES             // comment out to skip using the IR sensor/remote
//#define NO_GPS_WAIT YES         // define this for debugging to skip waiting for GPS fix


// Setup magnemeter  (compass); uses I2C
Adafruit_HMC5883_Unified compass = Adafruit_HMC5883_Unified(12345);
sensors_event_t compass_event;
```

```
// Create the motor shield object with the default I2C address
Adafruit_MotorShield AFMS = Adafruit_MotorShield();


// Setup motor controllers for both drive and steering (turn).
Adafruit_DCMotor *turnMotor = AFMS.getMotor(1);
Adafruit_DCMotor *driveMotor = AFMS.getMotor(3);
#define TURN_LEFT 1
#define TURN_RIGHT 2
#define TURN_STRAIGHT 99


// LCD Display
LiquidCrystal_I2C lcd(0x3F, 20, 4);  // Set the LCD I2C address and size (4x20)
#define LEFT_ARROW 0x7F
#define RIGHT_ARROW 0x7E
#define DEGREE_SYMBOL 0xDF
//char lcd_buffer[20];
//PString message(lcd_buffer, sizeof(lcd_buffer));    // holds message we display on line 4 of LCD


// Ultrasonic ping sensor
#define TRIGGER_PIN  11
#define ECHO_PIN  11
#define MAX_DISTANCE_CM 250                    // Maximum distance we want to ping for (in
CENTIMETERS). Maximum sensor distance is rated at 400-500cm.
#define MAX_DISTANCE_IN (MAX_DISTANCE_CM / 2.5)    // same distance, in inches
int sonarDistance;
NewPing sonar(TRIGGER_PIN, ECHO_PIN, MAX_DISTANCE_CM);     // NewPing setup of pins and
maximum distance.
MovingAverage<int, 3> sonarAverage(MAX_DISTANCE_IN);      // moving average of last n pings, initialize
at MAX_DISTANCE_IN


// Compass navigation
int targetHeading;            // where we want to go to reach current waypoint
int currentHeading;           // where we are actually facing now
int headingError;             // signed (+/-) difference between targetHeading and currentHeading
#define HEADING_TOLERANCE 5     // tolerance +/- (in degrees) within which we don't attempt to turn to
intercept targetHeading


// GPS Navigation
#define GPSECHO false          // set to TRUE for GPS debugging if needed
//#define GPSECHO true          // set to TRUE for GPS debugging if needed
SoftwareSerial mySerial(8, 7);    // digital pins 7 & 8
Adafruit_GPS GPS(&mySerial);
boolean usingInterrupt = false;
float currentLat,
    currentLong,
    targetLat,
    targetLong;
```

```
int distanceToTarget,          // current distance to target (current waypoint)
   originalDistanceToTarget;   // distance to original waypoing when we started navigating to it


// Waypoints
#define WAYPOINT_DIST_TOLERANE  5   // tolerance in meters to waypoint; once within this tolerance,
will advance to the next waypoint
#define NUMBER_WAYPOINTS 5         // enter the numebr of way points here (will run from 0 to (n-1))
int waypointNumber = -1;           // current waypoint number; will run from 0 to (NUMBER_WAYPOINTS -
1); start at -1 and gets initialized during setup()
waypointClass waypointList[NUMBER_WAYPOINTS] = {waypointClass(30.508302, -97.832624),
waypointClass(30.508085, -97.832494), waypointClass(30.507715, -97.832357),
waypointClass(30.508422, -97.832760), waypointClass(30.508518,-97.832665) };


// Steering/turning
enum directions {left = TURN_LEFT, right = TURN_RIGHT, straight = TURN_STRAIGHT} ;
directions turnDirection = straight;


// Object avoidance distances (in inches)
#define SAFE_DISTANCE 70
#define TURN_DISTANCE 40
#define STOP_DISTANCE 12


// Speeds (range: 0 - 255)
#define FAST_SPEED 150
#define NORMAL_SPEED 125
#define TURN_SPEED 100
#define SLOW_SPEED 75
int speed = NORMAL_SPEED;



// IR Receiver
#ifdef USE_IR
  #include "IRremote.h"          // IR remote
  #define IR_PIN 5
  IRrecv IR_receiver(IR_PIN);         // create instance of 'irrecv'
  decode_results IR_results;          // create instance of 'decode_results'
#endif


// IR result codes
#define IR_CODE_FORWARD 0x511DBB
#define IR_CODE_LEFT 0x52A3D41F
#define IR_CODE_OK 0xD7E84B1B
#define IR_CODE_RIGHT 0x20FE4DBB
#define IR_CODE_REVERSE 0xA3C8EDDB
#define IR_CODE_1 0xC101E57B
#define IR_CODE_2 0x97483BFB
#define IR_CODE_3 0xF0C41643
#define IR_CODE_4 0x9716BE3F
```

```
#define IR_CODE_5 0x3D9AE3F7
#define IR_CODE_6 0x6182021B
#define IR_CODE_7 0x8C22657B
#define IR_CODE_8 0x488F3CBB
#define IR_CODE_9 0x449E79F
#define IR_CODE_STAR 0x32C6FDF7
#define IR_CODE_0 0x1BC0157B
#define IR_CODE_HASHTAG 0x3EC3FC1B


//
// Interrupt is called once a millisecond, looks for any new GPS data, and stores it
SIGNAL(TIMER0_COMPA_vect)
{
  GPS.read();
}


//
// turn interrupt on and off
void useInterrupt(boolean v)
{
  if (v) {
    // Timer0 is already used for millis() - we'll just interrupt somewhere
    // in the middle and call the "Compare A" function above
    OCR0A = 0xAF;
    TIMSK0 |= _BV(OCIE0A);
    usingInterrupt = true;
  } else {
    // do not call the interrupt function COMPA anymore
    TIMSK0 &= ~_BV(OCIE0A);
    usingInterrupt = false;
  }
}



void setup()
{

  // turn on serial monitor
  Serial.begin(115200);       // we need this speed for the GPS


  //
  // Start LCD display
  lcd.begin();          // start the LCD...new version doesn't require size startup parameters
  #ifdef USE_LCD_BACKLIGHT
    lcd.backlight();
  #else
    lcd.noBacklight();
  #endif
  lcd.clear();
  #ifdef USE_GRAPHING
```

```
  createLCDChars();  // initialize LCD with graphing characters
#endif


//
// Start motor drives
AFMS.begin();  // create with the default frequency 1.6KHz

// Set the speed to start, from 0 (off) to 255 (max speed)
driveMotor->setSpeed(NORMAL_SPEED);
turnMotor->setSpeed(255);             // full turn; only option with current RC car chassis


//
// start Mag / Compass
if(!compass.begin())
 {
  #ifdef DEBUG
   Serial.println(F("COMPASS ERROR"));
  #endif
  lcd.print(F("COMPASS ERROR"));
  loopForever();        // loop forever, can't operate without compass
 }


//
// start GPS and set desired configuration
GPS.begin(9600);                        // 9600 NMEA default speed
GPS.sendCommand(PMTK_SET_NMEA_OUTPUT_RMCGGA);    // turns on RMC and GGA (fix data)
GPS.sendCommand(PMTK_SET_NMEA_UPDATE_1HZ);      // 1 Hz update rate
GPS.sendCommand(PGCMD_NOANTENNA);             // turn off antenna status info
useInterrupt(true);                     // use interrupt to constantly pull data from GPS
delay(1000);

//
// Wait for GPS to get signal
#ifndef NO_GPS_WAIT
lcd.setCursor(0, 0);
lcd.print(F("Waiting for GPS"));
unsigned long startTime = millis();
while (!GPS.fix)                // wait for fix, updating display with each new NMEA sentence received
 {
  lcd.setCursor(0, 1);
  lcd.print(F("Wait Time: "));
  lcd.print((int) (millis() - startTime) / 1000);    // show how long we have waited
  if (GPS.newNMEAreceived())
   GPS.parse(GPS.lastNMEA());
 } // while (!GPS.fix)
//delay(1000);
#endif


//
// Start the IR receiver
```

```
 #ifdef USE_IR
  IR_receiver.enableIRIn(); // Start the receiver
  // Wait for operator to press key to start moving
  lcd.setCursor(0, 3);
  lcd.print(F("Press key to start"));
  while(!IR_receiver.decode(&IR_results)) ;   // wait for key press
  IR_receiver.resume();  // get ready for any additional input
 #else
  // if not waiting for user input to start driving, at least give a countdown so they are ready...
  lcd.clear();
  lcd.print(F("GPS Acquired"));
  lcd.setCursor(0, 1);
  lcd.print(F("Starting in..."));
  lcd.setCursor(0, 2);
  for (int i = 10; i > 0; i--)
  {
    lcd.print(i);
    lcd.print(F(" "));
    if (GPS.newNMEAreceived())
        GPS.parse(GPS.lastNMEA());
    delay(500);
  }
 #endif


 //
 // get initial waypoint; also sets the distanceToTarget and courseToTarget varilables
 nextWaypoint();

} // setup()




void loop()
{

   // check for manual kill switch pressed
   #ifdef USE_IR
    checkKillSwitch();
   #endif

   // Process GPS
   if (GPS.newNMEAreceived())            // check for updated GPS information
    {
      if(GPS.parse(GPS.lastNMEA()) )     // if we successfully parse it, update our data fields
        processGPS();
    }

   // navigate
   currentHeading = readCompass();    // get our current heading
   calcDesiredTurn();              // calculate how we would optimatally turn, without regard to obstacles

   // distance in front of us, move, and avoid obstacles as necessary
```

```
   checkSonar();
   moveAndAvoid();

   // update display and serial monitor
   updateDisplay();

}  // loop()




//
// Called after new GPS data is received; updates our position and course/distance to waypoint
void processGPS(void)
{
  currentLat = convertDegMinToDecDeg(GPS.latitude);
  currentLong = convertDegMinToDecDeg(GPS.longitude);

  if (GPS.lat == 'S')          // make them signed
    currentLat = -currentLat;
  if (GPS.lon = 'W')
    currentLong = -currentLong;

  // update the course and distance to waypoint based on our new position
  distanceToWaypoint();
  courseToWaypoint();

}   // processGPS(void)




void checkSonar(void)
{
  int dist;

  dist = sonar.ping_in();               // get distqnce in inches from the sensor
  if (dist == 0)                       // if too far to measure, return max distance;
    dist = MAX_DISTANCE_IN;
  sonarDistance = sonarAverage.add(dist);     // add the new value into moving average, use resulting
average
} // checkSonar()




int readCompass(void)
{
  compass.getEvent(&compass_event);
  float heading = atan2(compass_event.magnetic.y, compass_event.magnetic.x);

  // Once you have your heading, you must then add your 'Declination Angle', which is the 'Error' of the
magnetic field in your location.
  // Find yours here: http://www.magnetic-declination.com/
```

```
  // Cedar Park, TX: Magnetic declination: 4° 11' EAST (POSITIVE);  1 degreee = 0.0174532925 radians

  #define DEC_ANGLE 0.069
  heading += DEC_ANGLE;

  // Correct for when signs are reversed.
  if(heading < 0)
    heading += 2*PI;

  // Check for wrap due to addition of declination.
  if(heading > 2*PI)
    heading -= 2*PI;

  // Convert radians to degrees for readability.
  float headingDegrees = heading * 180/M_PI;

  return ((int)headingDegrees);
}  // readCompass()



void calcDesiredTurn(void)
{
    // calculate where we need to turn to head to destination
    headingError = targetHeading - currentHeading;

    // adjust for compass wrap
    if (headingError < -180)
      headingError += 360;
    if (headingError > 180)
      headingError -= 360;

    // calculate which way to turn to intercept the targetHeading
    if (abs(headingError) <= HEADING_TOLERANCE)      // if within tolerance, don't turn
      turnDirection = straight;
    else if (headingError < 0)
      turnDirection = left;
    else if (headingError > 0)
      turnDirection = right;
    else
      turnDirection = straight;

}  // calcDesiredTurn()




void moveAndAvoid(void)
{

    if (sonarDistance >= SAFE_DISTANCE)       // no close objects in front of car
        {
          if (turnDirection == straight)
            speed = FAST_SPEED;
```

```
        else
          speed = TURN_SPEED;
        driveMotor->setSpeed(speed);
        driveMotor->run(FORWARD);
        turnMotor->run(turnDirection);
        return;
      }

    if (sonarDistance > TURN_DISTANCE && sonarDistance < SAFE_DISTANCE)    // not yet time to turn, but
slow down
      {
       if (turnDirection == straight)
        speed = NORMAL_SPEED;
       else
        {
          speed = TURN_SPEED;
          turnMotor->run(turnDirection);      // alraedy turning to navigate
         }
       driveMotor->setSpeed(speed);
       driveMotor->run(FORWARD);
       return;
      }

    if (sonarDistance <  TURN_DISTANCE && sonarDistance > STOP_DISTANCE)  // getting close, time to
turn to avoid object
      {
       speed = SLOW_SPEED;
       driveMotor->setSpeed(speed);      // slow down
       driveMotor->run(FORWARD);
       switch (turnDirection)
       {
        case straight:               // going straight currently, so start new turn
         {
          if (headingError <= 0)
            turnDirection = left;
          else
            turnDirection = right;
          turnMotor->run(turnDirection);  // turn in the new direction
          break;
         }
        case left:                   // if already turning left, try right
         {
          turnMotor->run(TURN_RIGHT);
          break;
         }
        case right:                  // if already turning right, try left
         {
          turnMotor->run(TURN_LEFT);
          break;
         }
       } // end SWITCH

       return;
      }
```

```
    if (sonarDistance <  STOP_DISTANCE)         // too close, stop and back up
     {
       driveMotor->run(RELEASE);          // stop
       turnMotor->run(RELEASE);           // straighten up
       turnDirection = straight;
       driveMotor->setSpeed(NORMAL_SPEED);  // go back at higher speet
       driveMotor->run(BACKWARD);
       while (sonarDistance < TURN_DISTANCE)       // backup until we get safe clearance
        {
          if(GPS.parse(GPS.lastNMEA()) )
            processGPS();
          currentHeading = readCompass();    // get our current heading
          calcDesiredTurn();              // calculate how we would optimatally turn, without regard to
obstacles
          checkSonar();
          updateDisplay();
          delay(100);
        } // while (sonarDistance < TURN_DISTANCE)
       driveMotor->run(RELEASE);       // stop backing up
       return;
      } // end of IF TOO CLOSE

}  // moveAndAvoid()




void nextWaypoint(void)
{
  waypointNumber++;
  targetLat = waypointList[waypointNumber].getLat();
  targetLong = waypointList[waypointNumber].getLong();

  if ((targetLat == 0 && targetLong == 0) || waypointNumber >= NUMBER_WAYPOINTS)    // last waypoint
reached?
   {
     driveMotor->run(RELEASE);    // make sure we stop
     turnMotor->run(RELEASE);
     lcd.clear();
     lcd.println(F("* LAST WAYPOINT *"));
     loopForever();
   }

   processGPS();
   distanceToTarget = originalDistanceToTarget = distanceToWaypoint();
   courseToWaypoint();

}  // nextWaypoint()
```

```
// returns distance in meters between two positions, both specified
// as signed decimal-degrees latitude and longitude. Uses great-circle
// distance computation for hypothetical sphere of radius 6372795 meters.
// Because Earth is no exact sphere, rounding errors may be up to 0.5%.
// copied from TinyGPS library
int distanceToWaypoint()
{

  float delta = radians(currentLong - targetLong);
  float sdlong = sin(delta);
  float cdlong = cos(delta);
  float lat1 = radians(currentLat);
  float lat2 = radians(targetLat);
  float slat1 = sin(lat1);
  float clat1 = cos(lat1);
  float slat2 = sin(lat2);
  float clat2 = cos(lat2);
  delta = (clat1 * slat2) - (slat1 * clat2 * cdlong);
  delta = sq(delta);
  delta += sq(clat2 * sdlong);
  delta = sqrt(delta);
  float denom = (slat1 * slat2) + (clat1 * clat2 * cdlong);
  delta = atan2(delta, denom);
  distanceToTarget =  delta * 6372795;

  // check to see if we have reached the current waypoint
  if (distanceToTarget <= WAYPOINT_DIST_TOLERANE)
    nextWaypoint();

  return distanceToTarget;
}  // distanceToWaypoint()




// returns course in degrees (North=0, West=270) from position 1 to position 2,
// both specified as signed decimal-degrees latitude and longitude.
// Because Earth is no exact sphere, calculated course may be off by a tiny fraction.
// copied from TinyGPS library
int courseToWaypoint()
{
  float dlon = radians(targetLong-currentLong);
  float cLat = radians(currentLat);
  float tLat = radians(targetLat);
  float a1 = sin(dlon) * cos(tLat);
  float a2 = sin(cLat) * cos(tLat) * cos(dlon);
  a2 = cos(cLat) * sin(tLat) - a2;
  a2 = atan2(a1, a2);
  if (a2 < 0.0)
  {
    a2 += TWO_PI;
  }
  targetHeading = degrees(a2);
```

```
  return targetHeading;
}   // courseToWaypoint()



// converts lat/long from Adafruit degree-minute format to decimal-degrees; requires <math.h> library
double convertDegMinToDecDeg (float degMin)
{
  double min = 0.0;
  double decDeg = 0.0;

  //get the minutes, fmod() requires double
  min = fmod((double)degMin, 100.0);

  //rebuild coordinates in decimal degrees
  degMin = (int) ( degMin / 100 );
  decDeg = degMin + ( min / 60 );

  return decDeg;
}



//
// Uses 4 line LCD display to show the following information:
// LINE 1: Target Heading; Current Heading;
// LINE 2: Heading Error; Distance to Waypoint;
// LINE 3: Sonar Distance; Speed;
// LINE 4: Memory Availalble; Waypoint X of Y;
void updateDisplay(void)
{

  static unsigned long lastUpdate = millis();       // for controlling frequency of LCD updates
  unsigned long currentTime;

  // check time since last update
  currentTime = millis();
  if (lastUpdate > currentTime)   // check for time wrap around
    lastUpdate = currentTime;

  if (currentTime >= lastUpdate + 500 )   // limit refresh rate
  {
    lastUpdate = currentTime;

    // line 1
    lcd.clear();
    lcd.print(F("tH= "));
    lcd.print(targetHeading, DEC);
    lcd.write(DEGREE_SYMBOL);
    lcd.print(F(" cH= "));
    lcd.print(currentHeading, DEC);
    lcd.write(DEGREE_SYMBOL);
```

```
  // line 2
  lcd.setCursor(0, 1);
  lcd.print(F("Err "));
  if (headingError < 0)
    lcd.write(LEFT_ARROW);
  lcd.print(abs(headingError), DEC);
  if (headingError > 0)
    lcd.write(RIGHT_ARROW);
  lcd.print(F(" Dist "));
  lcd.print(distanceToTarget, DEC);
  lcd.print(F("m "));
  #ifdef USE_GRAPHING
    lcd.write(map(distanceToTarget, 0, originalDistanceToTarget, 0, 7));    // show tiny bar graph of
distance remaining
  #endif

  // line 3
  lcd.setCursor(0, 2);
  lcd.print(F("Snr "));
  lcd.print(sonarDistance, DEC);
  #ifdef USE_GRAPHING
    lcd.write(map(sonarDistance, 0, MAX_DISTANCE_IN, 0, 7));
  #endif
  lcd.print(F(" Spd "));
  lcd.print(speed, DEC);
  #ifdef USE_GRAPHING
    lcd.write(map(speed, 0, 255, 0, 7));
  #endif

  // line 4
  lcd.setCursor(0, 3);
  lcd.print(F("Mem "));
  lcd.print(freeRam(), DEC);
  lcd.print(F(" WPT "));
  lcd.print(waypointNumber + 1, DEC);
  lcd.print(F(" OF "));
  lcd.print(NUMBER_WAYPOINTS - 1, DEC);


   #ifdef DEBUG
    //Serial.print("GPS Fix:");
    //Serial.println((int)GPS.fix);
    Serial.print(F("LAT = "));
    Serial.print(currentLat);
    Serial.print(F(" LON = "));
    Serial.println(currentLong);
    //Serial.print("Waypint LAT =");
    //Serial.print(waypointList[waypointNumber].getLat());
    //Serial.print(F(" Long = "));
    //Serial.print(waypointList[waypointNumber].getLong());
    Serial.print(F(" Dist "));
    Serial.print(distanceToWaypoint());
    Serial.print(F("Original Dist "));
```

```
    Serial.println(originalDistanceToTarget);
    Serial.print(F("Compass Heading "));
    Serial.println(currentHeading);
    Serial.print(F("GPS Heading "));
    Serial.println(GPS.angle);

    //Serial.println(GPS.lastNMEA());

    //Serial.print(F("Sonar = "));
    //Serial.print(sonarDistance, DEC);
    //Serial.print(F(" Spd = "));
    //Serial.println(speed, DEC);
    //Serial.print(F("  Target = "));
    //Serial.print(targetHeading, DEC);
    //Serial.print(F("  Current = "));
    //Serial.print(currentHeading, DEC);
    //Serial.print(F("  Error = "));
    //Serial.println(headingError, DEC);
    //Serial.print(F("Free Memory: "));
    //Serial.println(freeRam(), DEC);
   #endif

  } //  if (currentTime >= lastUpdate + 500 )

}  // updateDisplay()



//
// Display free memory available
//#ifdef DEBUG
  int freeRam ()   // display free memory (SRAM)
    {
     extern int __heap_start, *__brkval;
     int v;
     return (int) &v - (__brkval == 0 ? (int) &__heap_start : (int) __brkval);
  } // freeRam()
//#endif




// end of program routine, loops forever
void loopForever(void)
{
  while (1)
    ;
}




//
// Graphing (mini-inline bar graph for LCD display)
```

```
#ifdef USE_GRAPHING
  void createLCDChars(void)
   {
   int lvl = 0;
   byte arry[8];
   for (int a = 7; a >= 0; a--)
    {
    for (int b = 0; b <= 7; b++)
    {
       if (b >= lvl)
         arry[b] = B11111;      // solid
       else
         //arry[b] = B00000;    // blank row
         arry[b] = B10001;      // hollow but with sides
     }
    lcd.createChar(a, arry);
    lvl++;
   }
  } // createLCDChars(void)
#endif




//
// Implement an IR "kill switch" if selected in configuration options
#ifdef USE_IR
  void checkKillSwitch(void)
  {
   if(IR_receiver.decode(&IR_results))     // check for manual "kill switch"
      {
        turnMotor->run(RELEASE);
        driveMotor->run(RELEASE);

        lcd.clear();
        lcd.print(F("Press to resume"));
        delay(1000);

        IR_receiver.resume();
        while(!IR_receiver.decode(&IR_results)) ;  // wait for key press
        IR_receiver.resume();  // get ready for any additional input
      }
  }  // checkKillSwitch()
#endif
```