GPS Sensor for Lego NXT

Technology Masterclass - Fall/winter 2008

Martijn ten Bhömer
m.t.bhomer@student.tue.nl
s040198

Contents

1 Introduction

1.1 GPS Sensor

My goal for this technology class is to design a GPS sensor for Lego NXT. Robot navigation is an interesting field and I think that GPS can contribute in making more autonomous robots using Lego. GPS is most successful in outdoor situations over larger distances; the current precision is 10 meters (or at maximum 5 meters). This precision can be interesting enough to develop outdoor applications using Lego. For example to follow a route, to log location information or to even integrate location information with external maps including other information.

This idea is not totally new. In the Lego community there have been attempts[1] to use GPS information with the NXT. Unfortunately the solutions developed are much too complex and specific for certain devices (it involves connecting a GPS receiver through bluetooth with the NXT). Because of these difficulties the demand for a GPS sensor remains unanswered and open for innovation.

My goal is to create an easy to use, plug and play sensor, which works together with Lejos (or any other NXT software platform). This sensor can be the beginning of a whole new range of outdoor Lego applications.

1.2 Challenges

The main challenge of this project is to let a GPS module talk with Lego NXT. The Lego NXT can transmit and receive information using the I^2C protocol (more information about I^2C in chapter 4.2 on page 13). Alas, the GPS module uses a serial protocol (the GPS protocol will be described in chapter 4.1 on page 12) and cannot communicate directly with I^2C . This has been solved by creating an own interface protocol (chapter 4.3 on page 15).

Another challenge was to make optimal use of the intermediary microcontroller between NXT and the GPS module. I wanted to do as much as possible pre-processing on the microcontroller itself instead of on the Lego NXT.

Other challenges are to design the electronics which will facilitate this intermediary protocol (chapter 3.2 on page 7), and of course the whole sensor needs a casing to embody the system (chapter 3.4 on page 9).

[1] for example this RC Car: `http://letsmakerobots.com/node/1865`. Although there is no concrete documentation on how to realize the GPS connection

2 Process Overview

To give an idea about the different steps in the development process, and how these steps followed each other I will introduce the process using photographs.

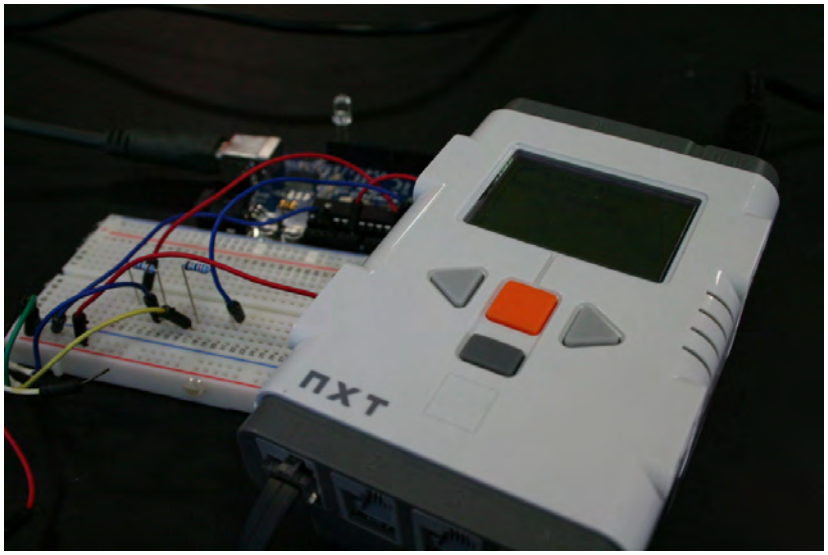

Figure 1: Arduino connected with NXT

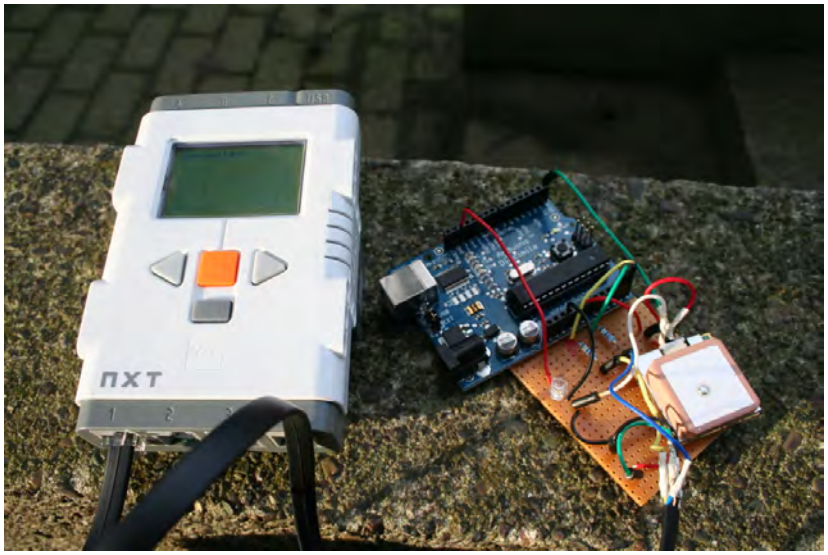

Figure 2: GPS connected with the NXT through the Arduino board

Figure 3: Casing parts

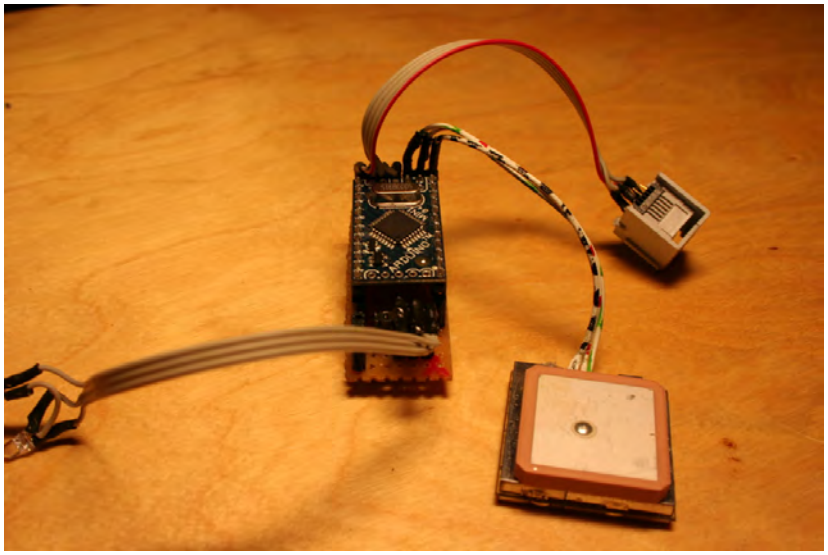

Figure 4: Electronic components

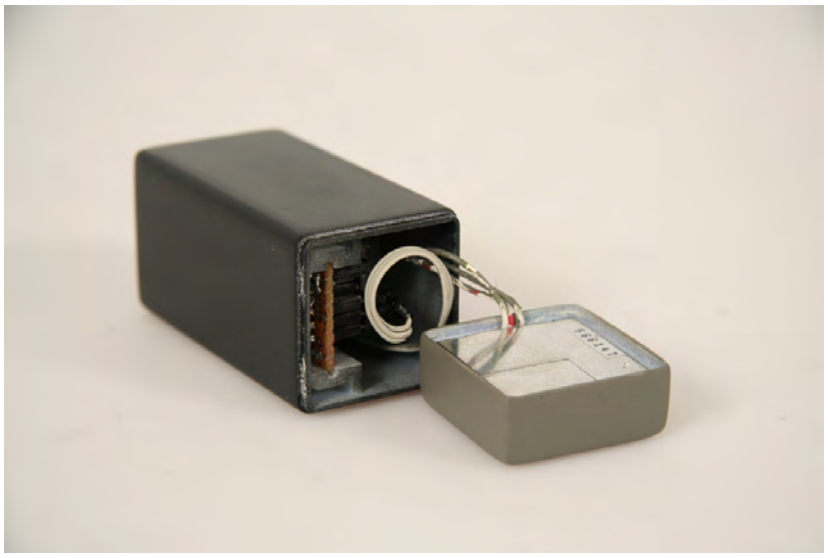

Figure 5: Electronic components in casing

Figure 6: Completed sensor

3 Hardware

3.1 GPS module

3.1.1 Specifications

The GPS module I used is the USGlobalSat EM-406A[2]. This is a relatively cheap module which has 10 meter Positional Accuracy and even 5 meter when it receives a WAAS signal (Wide Area Augmentation System[3]). The hot-start of the module is 1 second (last calculated position and which satellites where in view is still available) warm-start (the last calculated position is available, but not how many satellites were in view) of the module is 38 seconds and cold-start takes 42 seconds (all the information is erased and the position has to be calculated from scratch). An result for the NXT sensor is that the sensor itself will need this time before it can give accurate location information.

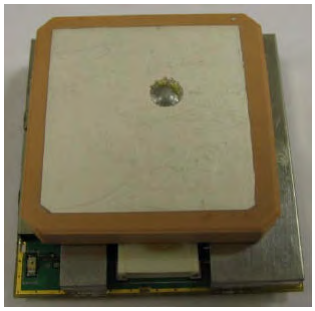

Figure 7: GPS module

3.1.2 Connections

The module has six connections (Figure 8): two ground pins, power input, RX and TX pins and the 1pps pin. The RX and TX (respectively receive channel and transmit channel) are the actual pins which are used for the serial connection. The TX pin is the main transmits channel for outputting navigation and measurement data. The RX pin can be used to send the module specific commands or to change settings. It is for example possible to provide location information to the module to let it find it's current location quicker. I didn't implement this in my Lego sensor, but if quicker start-up time is needed in the future this can be looked at. The 1pps pin provides one pulse-per-second output synchronized to GPS time.

[2] Available at SparkFun Electronics (`http://www.sparkfun.com/`)

[3] A feedback loop existing from an extra geostationary satellite which is connected with multiple ground-stations. These ground-stations are connected to normal orbital satellites and provide an error signal to the geostationary satellite. The geostationary satellite provides this error signal to GPS receivers and an WAAS enabled GPS receiver is able to process this signal. In Europe the equivalent is called EGNOS (European Geostationary Navigation Overlay Service) and is compatible with the North-American WAAS.

For my application I only needed to use the power input, ground connection and the TX output. This is further explained in Chapter 3.3 about the Arduino Shield.

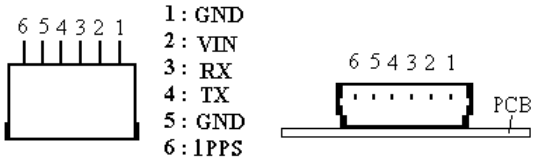

Figure 8: GPS module connections

3.2 Arduino Mini

The Arduino Mini (Figure 9) is a small microcontroller board based on the ATmega168 microcontroller. The board provides a lot of possibilities and features accessible through the Arduino development environment.

For actual implementation in a commercial product this board has some disadvantages, mainly price and size. It should be realized that for an commercial version of this sensor other microcontrollers should be considered. I choose to use this microcontroller during this technology class because of the good development environment and documentation available. I find it also important to build expertise on using this board because of the possible application of it in other projects.

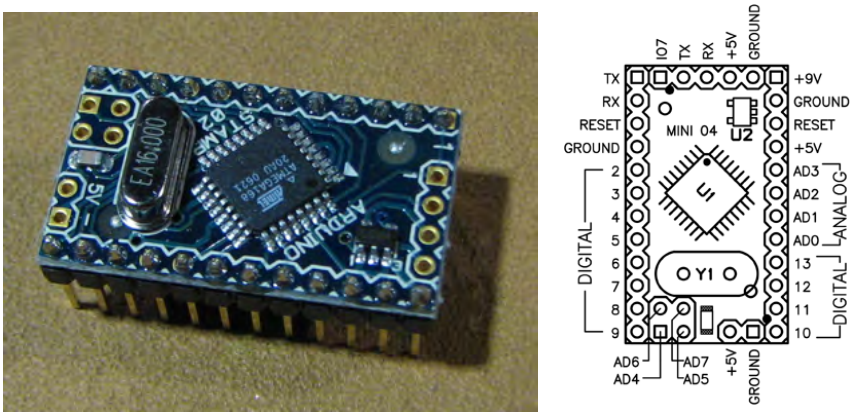

Figure 9: Arduino Mini and pin-layout

3.3 Arduino Shield

Shields are boards that can be plugged on top of the Arduino PCB and extend its capabilities. I developed my electronics in the form of a shield because of the

flexibility and the possibility to easy reproduce such a design. The total circuit is illustrated in figure 10, or for a larger version Appendix A on page 21. In the center the Atmega microcontroller is visible, on the left side the connector to the NXT can be found and on the right side the GPS module. Additionally there are inputs to program the microcontroller and there are two status LED's. I will explain the different connections to the microcontroller part by part.

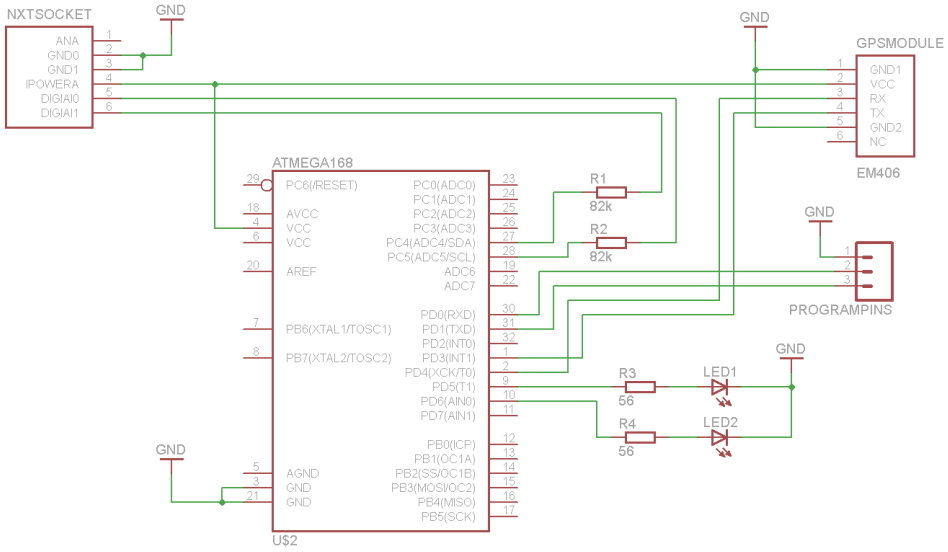

Figure 10: Electrical circuit

3.3.1 NXT Socket

The main power for the microcontroller is supplied by pin 4 of the NXT socket. Pin 2 & 3 are both connected to the ground of the electrical circuit. Pin 5 is the Clock line of the I^2C wiring, and pin 6 is the Data line. Pin 5 & 6 are both connected with pull-up resistors of 82kΩ[4] to pin 4 & pin 5 of the microcontroller which are respectively the clock and data line for this type of microcontroller.

3.3.2 GPS Module

The power for the GPS module is also supplied by pin 4 of the NXT socket. The grounds of the module (pin 1 and 5) are connected to the common ground of the circuit. Pin 3, the receive pin is attached to pin 2 of the microcontroller which has the name digital port 4. Pin 4 of the GPS module is connected to pin 1 of the microcontroller (digital port 3).

[4]This is described in the Lego NXT Hardware Developer Kit `http://mindstorms.lego.com/Overview/nxtreme.aspx`.

3.3.3 Program pins

To program the microcontroller three wires are needed. Pin 1 is the common ground, pin 2 is used to send data and pin 3 receives the data. Pin 2 is connected to the receive pin of the microcontroller (pin 30) and pin 3 is connected to the send pin (pin 31).

3.3.4 LED's

There are two LED connections on the board to give status information. The LED's are connected to pin 9 and pin 10 of the microcontroller. They are connected in series with a resistor of 56Ω. These values can easy be calculated using ohms law $R = \frac{V}{I}$. I used LED's with a desired current of 20 mA and a voltage drop of 3.8V of the 5V source voltage. $R = \frac{5-3.8}{0.02} = 60\Omega$

3.3.5 Finished shield

Figure 11 shows the finished shields. For future versions this shield can easily be produced as a PCB, this will make it possible to miniaturize the design by eliminating the header pins. And will also save a lot of small and frustrating soldering work.

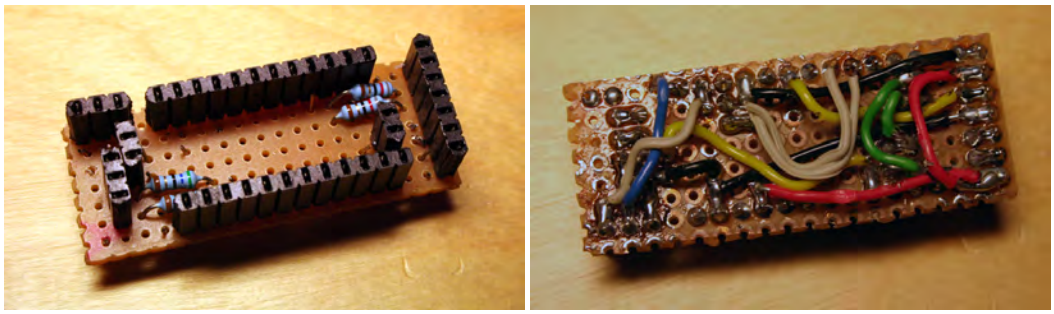

Figure 11: The finished shield with all the inputs (top and bottom)

3.4 Casing

For the casing of the sensor I wanted to meet these requirements:

- Easy to open and close
- Components modular in casing
- Able to connect to Lego
- Professional appearance integrated with Lego style

The technical drawings of this sensor are available in Appendix B on page 22.

Figure 12: Casing parts

Figure 12 shows the casing of the sensor. I created a casing which can be opened from one side. The electronics all fit exactly in the designated spots. For example a rail to slide in the shield and support to fit the NXT socket.

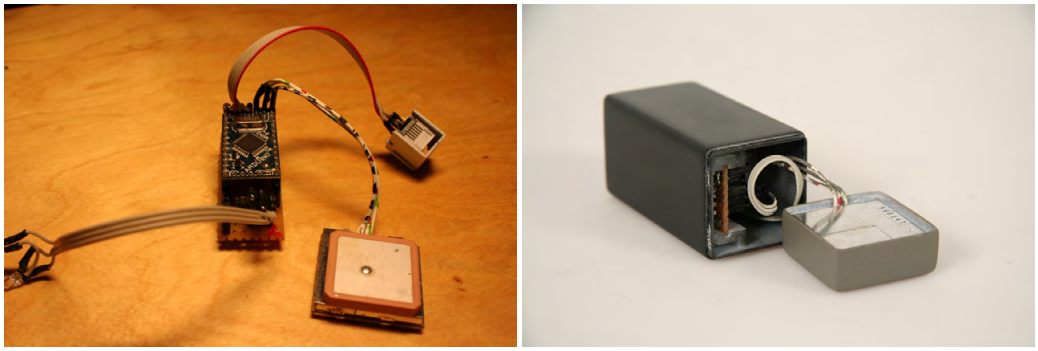

Figure 13: Electronic components modular in casing

Figure 13 shows that the electronics are all modular connected to the shield. This makes it possible to change parts, and to dismantle the shield completely from the sensor casing.

Figure 14: Completed sensor

Figure 14 shows the final sensor and how it can be connected to Lego.

4 Software

4.1 Serial GPS

The GPS protocol is based on the NMEA-0183 standard[5] for interfacing marine electronic devices as defined by the National Marine Electronics Association (NMEA). Within this protocol different output messages can be transferred. One of these messages is the GPRMC message, the recommended Minimum Specific GNSS Data. This message contains the bare essentials of GPS data and contains enough information for this implementation.

This is an example GPRMC output message:

$GPRMC,161229.487,A,3723.2475,N,12158.3416,W,0.13,309.62,120598, ,*10

From this data we can extract these variables:

Name	Example	Units	Description
Message ID	$GPRMC		RMC protocol header
UTC Time	161229.487		hhmmss.sss
Status	A		A=data valid or V=data not valid
Latitude	3723.2475		ddmm.mmmm
N/S Indicator	N		N=north or S=south
Longitude	12158.3416		dddmm.mmmm
E/W Indicator	W		E=east or W=west
Speed Over Ground	0.13	knots	
Course Over Ground	309.62	degrees	True
Date	120598		ddmmyy
Magnetic Variation		degrees	E=east or W=west
Mode	A		A=Autonomous, D=DGPS, E=DR
Checksum	*10		
<CR> <LF>			End of message termination

Table 1: GPRMC variables

One of the design challenges was to use the intermediary microcontroller to it's full extend. By using it to pre-process the GPRMC messages the sensor becomes more usable for direct implementation on the Lego NXT. The extraction of data from the GPRMC output message has been realized with this code:

[5]The reference manual from SiRF can be found on `http://www.newmicros.com/store/product_manual/NMEA_Manual.pdf`

```
for (int i=0;i<300;i++)
{
  //check for the position of the  "," separator
  if (stringGPS[i]==',')
  {
    if(seperatorCounter < 12)
        {
          //and store the location in the seperators array
          seperators[seperatorCounter]=i;
          seperatorCounter++;
        }
  }
}
```

With the positions of the commas stored in the separators array it becomes possible to retrieve the beginning and end point for every variable. The total code for this process can be found in Appendix C on page 24.

4.2 I²C

The I²C bus has been developed in the beginning of the 80's by Philips with the intend to create an easy connection between processor and chips of a television. The advantage of this system is that it only uses two wires, which is much more efficient than the Byte Wide system which has been used until the introduction of I²C.

The two wired system is possible because the data is transmitted serially, one bit at a time. One wire is for sending and receiving data (SDA) and the other is providing a clock (SCL). Important to know is that an I²C bus always has one Master node - in our case the NXT - and can have up to 127 Slave nodes.

On low-level view the I²C protocol looks like figure 15. The transfer starts by sending a start byte (S), the Data line (SDA) is pulled low, while the Clock line (SCL) stays high. Then, SDA sets the transferred bit while SCL is low (blue) and the data is received when SCL rises (green). When the transfer is complete, a stop byte (P) is sent by pulling SDA up, while SCL also remains high.

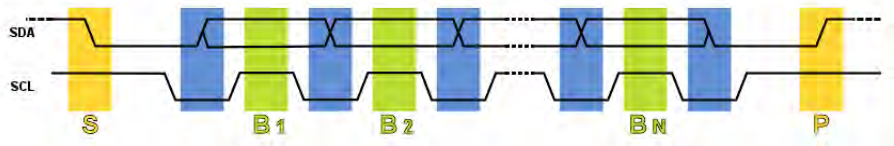

Figure 15: Data and Clock line pulses (picture from Wikipedia)

Luckily on higher level we don't need to deal with the timing of these two

lines. Lejos and Arduino both have their own implementations, which makes this protocol easy to implement.

4.2.1 Lejos I²C implementation

First the I²C classes of Lejos have to be imported.

```
import lejos.nxt.I2CPort;
import lejos.nxt.I2CSensor;
```

The class has to extend the I2CSensor class, and the constructor uses a sensor port as parameter to override the port of the I2CPort class.

```
public class GPSSensor extends I2CSensor {

        public GPSSensor(I2CPort port) {
                super(port); //override the I2CPort class
        }
```

The address of the slave device is set, in this case this is 127.

```
// Start of I2C connection with the sensor,
// the address of the sensor is 127
setAddress(127);
```

The Lejos function to request data from the slave device is getData. The first parameter is the register of the device we would like to request data from, the second is a byte array which stores the actual data and the third parameter is the length of the byte array.

```
//the byte array which will be used to save the data
byte[] readData = new byte[11];
//reading the string (transfered in bytes) from the sensor
getData(0, readData, 11);
```

For the full Lejos code please consult Appendix D.2 on page 40.

4.2.2 Arduino I²C implementation

The Wire library of Arduino[6] has to be imported to be able to use the I²C functions of the microcontroller.

```
#include <Wire.h>
```

Next we give the command to join the I²C bus, and we define the address (which also sets this device as a slave device).

```
Wire.begin(127); //join i2c bus with address 127
```

Then we create an event listener which will response in the event that data is requested by the master. In this case the requestEvent function is called.

[6] The documentation of the library can not be found in the Arduino reference, but in the Wiring reference `http://wiring.org.co/reference/libraries/Wire/Wire_.html`

```
//when data is requested requestEvent is called
Wire.onRequest(requestEvent);
```

When data is requested by the Master node, it is necessary to reply with data. This can be done using the Wire.send function.

```
void requestEvent()
{
  //function that is triggered when the I2C
  //master requests data (in this case the NTX)
  Wire.send(0); //send a zero back to master
}
```

The full Arduino code can be viewed in Appendix C on page 24.

4.3 Interface protocol

The interface protocol makes it possible to send GPS data through I^2C to the NXT. My goal was to let the sensor do the pre-processing and to have a bi-directional protocol. This means that for example the NXT can ask "give me the current latitude" and the sensor replies with "3723.2475", or NXT asks "give me the current time" and the sensor gives "161229.487". Such a bi-directional communication is supported by I^2C. It is for example possible to create multiple register on the slave node, and let the Master poll for a specific register. Unfortunately the Arduino implementation of I^2C is not so sophisticated, it only enables the Master node to send one request, and can in return send data. That's why I made my own protocol within I^2C that does support bi-directional communication. It is based on timing.

The protocol exists from the following steps:

1. Fast data requests with 10ms between each other to decide what kind of data (e.g. 2 x request with 10ms break is latitude, 4 requests is longitude).
2. A pause of 30ms to indicate that the data type has been communicated.
3. A request for the confirmation of the data type.
4. A request to receive byte 35 (character #) to indicate the actual data transfer starts.
5. A request to receive a byte array with the data.

In Lejos this looks like this:

```
// Protocol step 1
for (int h = 1; h <= dataType + 1; h++)
{
  getData(0, readResponse, 1);
  try { Thread.sleep(10); }
  catch (Exception E) { }
```

```
}

// Protocol step 2
try { Thread.sleep(30); }
catch (Exception E) { }

// Protocol step 3
try { getData(0, readResponse, 1); }
catch (Exception E) { readResponse[0] = 0; }
byteChar = readResponse[0];

//convert byte to int
returnInt = (int) byteChar;

//check if the integer received from the sensor is equeal
//to the datatype we intented
if (returnInt == dataType)
{
  data = true;
}

// Protocol step 4
try { getData(0, readResponse, 1);  }
catch (Exception E) { readResponse[0] = 0; }

// 35 is the # sign, means data transfer is starting
if (readResponse[0] == 35)
{

  // Protocol step 5
  try { Thread.sleep(10); }
  catch (Exception E) { }

  // reading the string (transfered in bytes) from the sensor
  try { getData(0, readData, 11); }
  catch (Exception E) { }
}
```

The full code can be viewed in Appendix D.2 on page 40. For the Slave side of the protocol please view the commented requestEvent function code in Appendix C on page 24.

5 Application

The application to demonstrate the Lego NXT GPS Sensor is a robot which can follow a route. In this report I will focus on the formulas which are necessary to implement such an application. The code for my specific application can be found in Appendix D.1 on page 35. All the important GPS functions[7] can be found in the GPSSensor class (Appendix D.2 on page 40).

5.1 Numeric degrees

As explained in Table 1 on page 12 the latitude and longitude from the GPS module are in ddmm.mmmm (d is degrees, and m are minutes) and dddmm.mmmm format respectively. To do calculations with these coordinates we need to translate these to numeric degrees. This is rather simple, by dividing the minutes by 60 we get degrees. Thus, $dd + (mm.mmmm/60)$ for the latitude and $ddd + (mm.mmmm/60)$ for the longitude.

In Lejos this looks like this for the latitude.

```
degrees = Integer.parseInt(x.substring(0, 2));
temp = x.substring(2, 4) + "." + x.substring(4, x.length());
minutes = Float.parseFloat(temp);
minutes = minutes / 60;
decimals = degrees + minutes;
```

5.2 Bearing

The bearing is the angle between two latitude, longitude points. This is needed to navigate the robot to the right direction. The formula for the bearing:

lat1 = start latitude

lat 2 = end latitude

$\Delta long$ = end latitude - start latitude

$\theta = \arctan(\sin(\Delta long) \times \cos(lat2), \cos(lat1) \times \sin(lat2) - \sin(lat1) \times \cos(lat2) \times \cos(\Delta long))$

We need to pay attention that for this formula the numeric degrees need to be calculated to radian degrees first. In Lejos the total function looks like this:

```
//first all the points in degrees are calculated into radians
double lat_from_rad = Math.toRadians(lat_from);
double lat_to_rad = Math.toRadians(lat_to);
double long_from_rad = Math.toRadians(long_from);
double long_to_rad = Math.toRadians(long_to);

//the distance between the two longitude points
double dLong = long_to_rad - long_from_rad;
```

[7] A great collection of important GPS calculations can be found on `http://www.movable-type.co.uk/scripts/latlong.html`.

```
//calculation of the bearing angle
double y = Math.sin(dLong) * Math.cos(lat_to_rad);
double x = Math.cos(lat_from_rad) * Math.sin(lat_to_rad) -
           Math.sin(lat_from_rad) * Math.cos(lat_to_rad)
           * Math.cos(dLong);
return toBearing((float) (Math.atan2(y, x)));
```

5.3 Distance

An important piece of information for a route following robot is to know when it arrived at its destination. For this we need a formula that can perform such a function. There are quite some formulas to calculate distances between to points on earth. Some are based on the model that the Earth is a sphere, and some are more complex. In my application I tried to implement both the Haversine formula and the spherical law of cosines formula.

The spherical law of cosines can be described as this:
lat1 = start latitude
lat 2 = end latitude
long1 = start longitude
long2 = end longitude
R = earth radius (6371 km)
$d = \arccos(\sin(lat1) \times \sin(lat2) + \cos(lat1) \times \cos(lat2) \times \cos(long2 - long1)) \times R$
In Lejos the implementation looks like this:

```
//convert to radians
L1 = Math.toRadians(L1);
L2 = Math.toRadians(L2);
G1 = Math.toRadians(G1);
G2 = Math.toRadians(G2);

double a = Math.pow(Math.sin((L2 - L1) / 2), 2) + Math.cos(L1)
           * Math.cos(L2) * Math.pow(Math.sin((G2 - G1) / 2), 2);

//great circle distance in radians
double angle = Math.toDegrees(Math.sqrt(a));

//each degree on a great circle of Earth is 60 nautical miles (111120m)
double distance = 111120 * angle;

return distance;
```

The problem with Lejos is that the square root function yields inaccurate results. The Math.sqrt() function from Lejos is different than the native Math.sqrt() function of Java. Because a small difference results in a very large difference in distance it is not possible to use this function in Lejos.

Currently I am using a function that looks if the latitude, longitude degrees are in range, and based on this makes a decision if the robot reached its end position.

```
boolean inrange = false;

float difference_lat = (float)Math.abs(lat2 - lat1);
float difference_lon = (float)Math.abs(lon2 - lon1);

if(difference_lat < 0.0001 && difference_lon < 0.0001)
{
  inrange = true;
}

return inrange;
```

In the future a better solution should be found. But for this demonstration application it suffices.

6 Discussion

In summary I designed a working GPS sensor compatible with Lego NXT. Currently the sensor is integrated using Lejos, but other implementations could be made as well, for example the easy to use Lego Mindstorms program. This is all possible because the interfacing between the GPS module and I^2C fully works.

With a package like this the possibilities of navigation and localizing could be further extended. The accuracy can be improved by using additional sensors, for example GSM localizing or accelerometers to measure the exact distance covered.

Physical improvements can be made in the size of the sensor. For example by eliminating the header pins from the Arduino board. An even better solution when considering commercial production is to choose a different microcontroller, smaller in size and less in features.

In the application of the sensor there are some limitations of the Lejos language. Especially its Math functions are currently too limited to do all the necessary calculations. Perhaps other functions exists which are less intensive for the NXT. Another solution is to let the calculations also find place on the microcontroller.

There is a lot to explore with this sensor and many new applications can be found. It will definitely take Lego beyond a toy.

A Electrical circuit

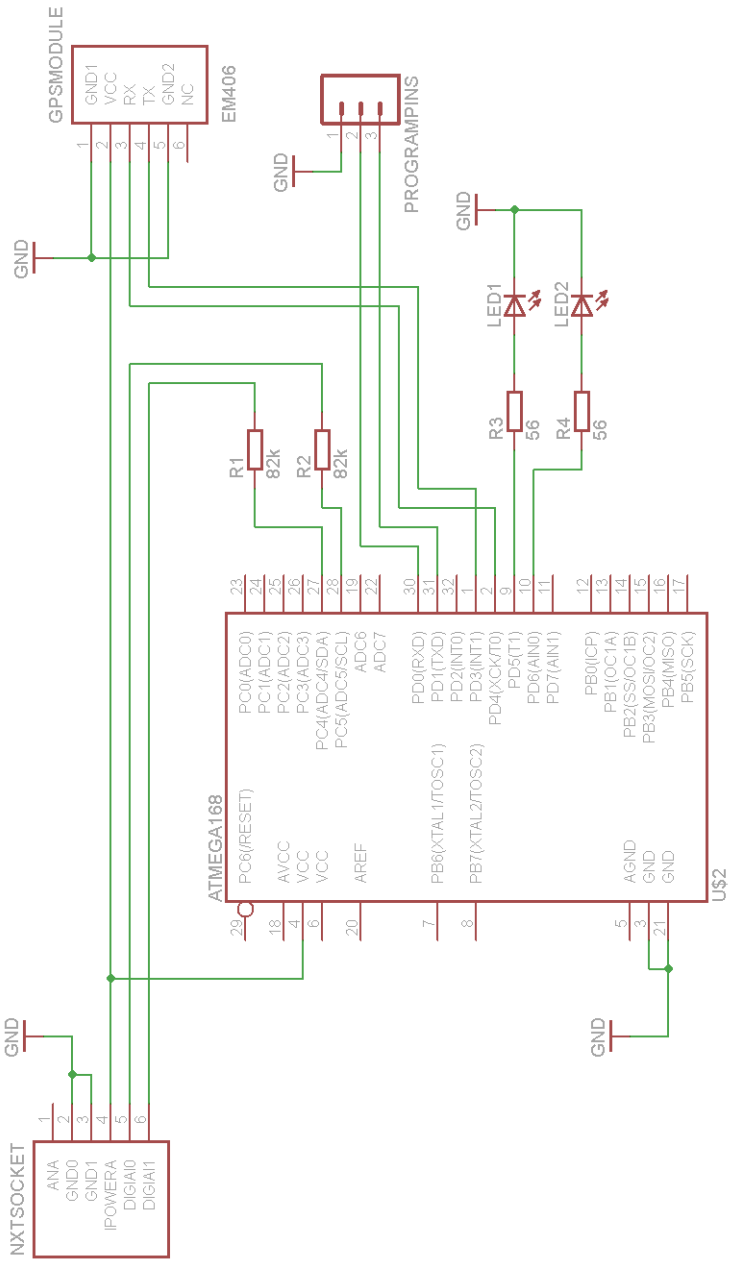

Figure 16: Electrical circuit

B Technical drawing casing

B.1 Casing front

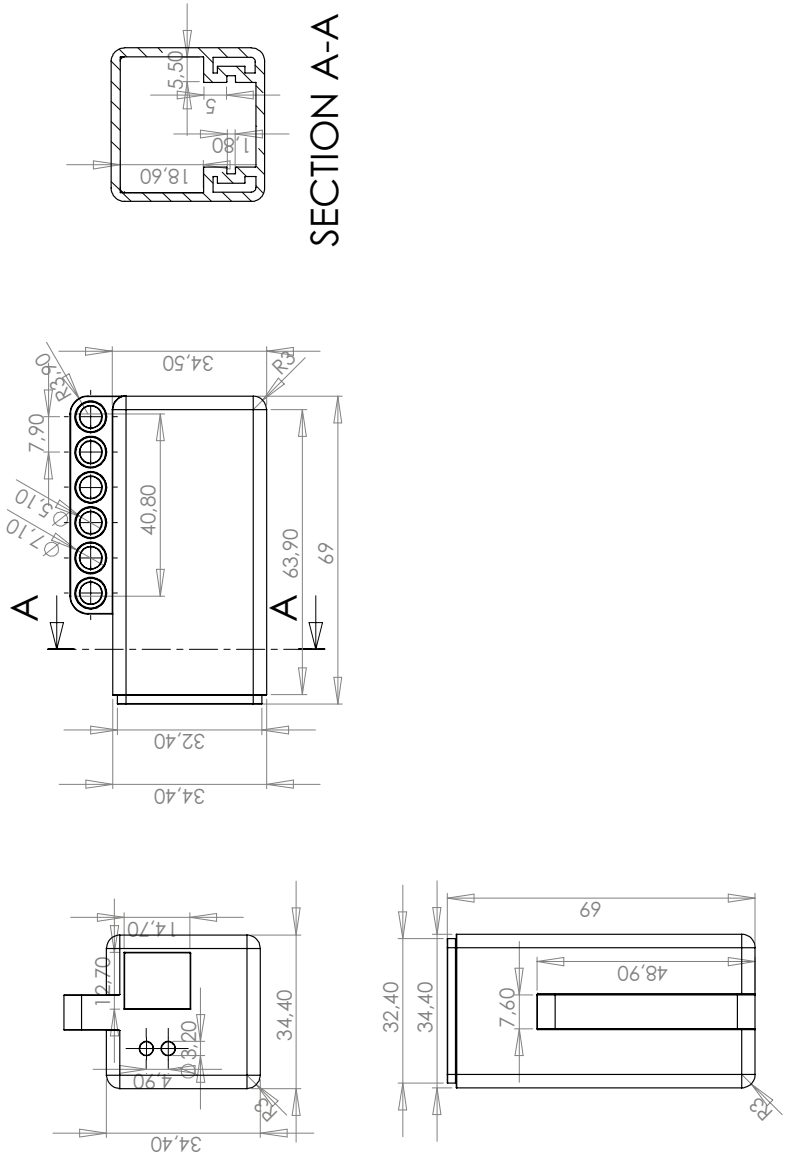

Figure 17: Casing front

B.2 Casing back

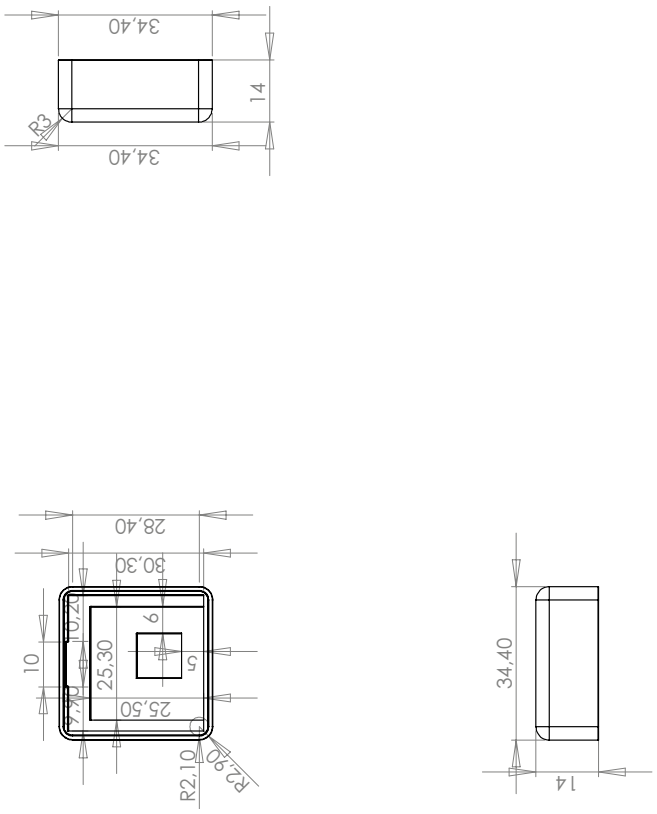

Figure 18: Casing back

B.3 Rendering

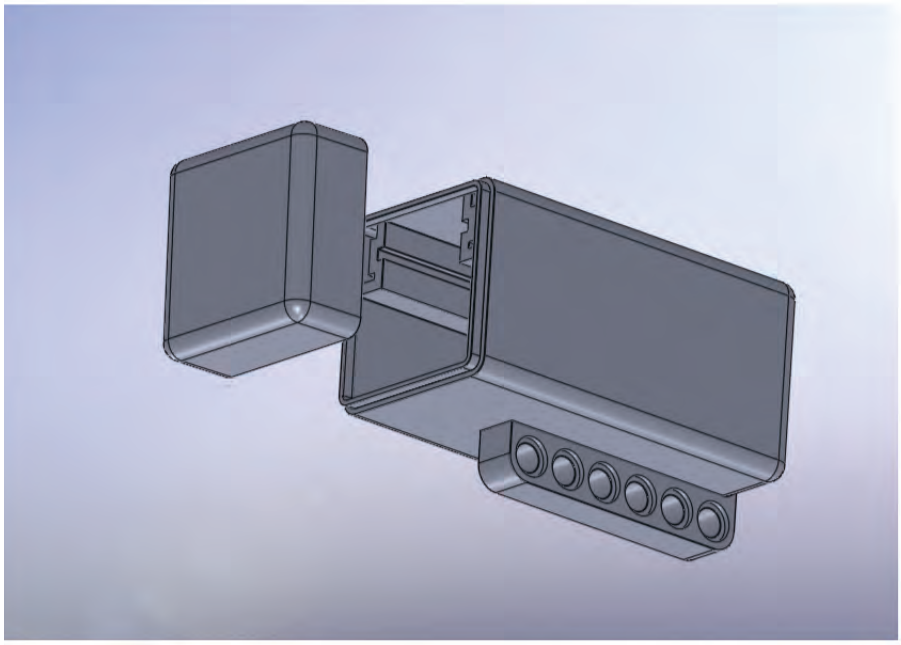

Figure 19: Rendering

C Arduino code

```
#include <Wire.h>
#include <string.h>
#include <ctype.h>
#include <SoftwareSerial.h>

// Software serial TX & RX Pins for the GPS module
#define SoftrxPin 3
#define SofttxPin 4

// Initiate the software serial connection
SoftwareSerial gpsSerial = SoftwareSerial(SoftrxPin, SofttxPin);

int ledPin = 6; //LED test pin
int ledPin2 = 5; //LED test pin

// Enable/disable debug modes
boolean debug_gps = false;
boolean debug_nxt = false;

// Timer variables
unsigned long lastRequest = 0; //the number of milliseconds from when the
program started running
int timePassed = 0; //interval time

// Protocol variables
int transactionNmr = 0; //which data to transfer
boolean alive = false; //variable to check if the connection is alive
boolean confirmed = false; //to check if the datatype is confirmed
boolean transfer = false; //variable to know when to start transfering
boolean dataAvailable = false; //variable that checks if there was already
data from the gps module

// GPS variables
int byteGPS=-1; //byte containing current received byte
char stringGPS[300] = ""; //string containing the total gps string
int stringGPSPosition = 0; //integer to hold the position of the current current
gps character
char stringCheck[7] = "$GPRMC"; //string to check if they gps string is of
the type GPRMC (the string that contains actual information)
int stringCheckCounter = 0;
int seperators[13]; //array to hold the location of the commas
int seperatorCounter = 0; //Integer to store the position of the comma

//GPS Strings
char gps0[7] = "";    //time in UTC (HhMmSs)
char gps1 = ' ';      //status of the data (A=active, V=invalid)
char gps2[11] = "";   //latitude
char gps3 = ' ';      //latitude Hemisphere (N/S)
char gps4[11] = "";   //longitude
char gps5 = ' ';      //longitude Hemisphere (E/W)
char gps6[4] = "";    //velocity (knots)
char gps7[7] = "";    //bearing (degrees)
char gps8[4] = "";    //checksum

void setup()
{
  pinMode(ledPin, OUTPUT);       //initialize LED pins
```

```
  pinMode(ledPin2, OUTPUT);
  blinkBothLED();               // blink leds to indicate the program has
booted

  Serial.begin(9600);           //start serial for output
  gpsSerial.begin(4800);         //start serial for communication with GPS

  Wire.begin(127);              //join i2c bus with address 127
  Wire.onRequest(requestEvent);   //request event

  for (int i=0;i<300;i++){      // Initialize a buffer for received data
    stringGPS[i]=' ';
  }

  Serial.println("Arduino booted");
}

void loop()
{
  //check every 100 ms if the connection is still alive
  //if the time passed is more then 100 ms, then kill connection
  if(millis() - lastRequest > 80 && alive)
  {
    killTransaction();
  }

  //poll the GPS module
  checkGPS();
}

void requestEvent()
{
  //function that is triggered when the I2C master requests data (in this
case the NTX)

  timePassed = millis() - lastRequest; //measure how many timepassed between
this request and last request.
  lastRequest = millis(); // make this request the last request

  //first check if we have an alive connection
  if(!alive)
    //a new transaction starts
  {
    digitalWrite(ledPin2, HIGH); //make the LED pin high to show a transaction
starts
    alive = true; //the connection is alive
    transactionNmr = 0; //reset datatype to zero
    transactionNmr++; //start counting
    Wire.send(0); //send a zero back to master
  }

  else if (alive)
    //a transaction is still running, check which operation to perform based
on the delay time
  {
    if(timePassed < 30 && !confirmed)
      //the datatype transaction is not yet finished
```

```
{
  if(debug_nxt) {
    Serial.println(timePassed);
  }
  Wire.send(0);
  transactionNmr++; //increase the datatype
}

else if(timePassed >= 40 && timePassed < 60 && !confirmed)
  //the transaction number is received, the NXT wants confirmation
{
  transactionNmr = transactionNmr - 1;
  Wire.send(transactionNmr); //send the understood datatype to master
  confirmed = true; //confirmed state
  if(debug_nxt) {
    Serial.println("confirm");
  }
}

else if(confirmed && alive && !transfer )
  //the actual transaction starts
{
  //first send character to indicate the transaction is starting
  Wire.send("#");
  transfer = true;
  if(debug_nxt) {
    Serial.println("start");
  }
}

else if(confirmed && alive && transfer)
  //the actual transaction starts
{
  //send the correct data, depending on the datatype (the transactionNmr)
  switch (transactionNmr) {
  case 0:
    if(dataAvailable)
    {
      Wire.send(gps0);
    }
    else
    {
      Wire.send("error");
    }
    if (debug_nxt) {
      Serial.println("case 0");
    }
    break;
  case 1:
    if(dataAvailable)
    {
      Wire.send(gps1);
    }
    else
    {
      Wire.send("error");
    }
```

```
  if(debug_nxt) {
    Serial.println("case 1");
    Serial.println(gps1);
  }
  break;
case 2:
  if(dataAvailable)
  {
    Wire.send(gps2);
  }
  else
  {
    Wire.send("error");
  }
  if(debug_nxt)
  {
    Serial.println("case 2");
    Serial.println(gps2);
  }
  break;
case 3:
  if(dataAvailable)
  {
    Wire.send(gps3);
  }
  else
  {
    Wire.send("error");
  }
  if(debug_nxt) {
    Serial.println("case 3");
  }
  break;
case 4:
  if(dataAvailable)
  {
    Wire.send(gps4);
  }
  else
  {
    Wire.send("error");
  }
  if(debug_nxt)
  {
    Serial.println("case 4");
    Serial.println(sizeof(gps4));
  }
  break;
case 5:
  if(dataAvailable)
  {
    Wire.send(gps5);
  }
  else
  {
    Wire.send("error");
  }
```

```
    if(debug_nxt) {
      Serial.println("case 5");
    }
    break;
  case 6:
    if(dataAvailable)
    {
      Wire.send(gps6);
    }
    else
    {
      Wire.send("error");
    }
    if(debug_nxt) {
      Serial.println("case 6");
    }
    break;
  case 7:
    if(dataAvailable)
    {
      Wire.send(gps7);
    }
    else
    {
      Wire.send("error");
    }
    if(debug_nxt) {
      Serial.println("case 7");
    }
    break;
  case 8:
    if(dataAvailable)
    {
      Wire.send(gps8);
    }
    else
    {
      Wire.send("error");
    }
    if(debug_nxt) {
      Serial.println("case 8");
    }
    break;
  default:
    Wire.send("error");
    if(debug_nxt) {
      Serial.println("else");
    }
    break;
  }

  killTransaction(); //transaction finished, kill the connection
}
else{
  Wire.send(0);

}
```

```
  }
}

void checkGPS() {
  //function to receive GPS data from the module

  digitalWrite(ledPin, LOW); //start by making the LED low

  byteGPS=gpsSerial.read(); //read a byte from the serial port

  if (byteGPS == -1) { //if no data is received, then do nothing
    delay(100);
  }
  else {
    stringGPS[stringGPSPosition]=byteGPS; //if there is serial port data,
it is put in the buffer
    stringGPSPosition++; //and the buffer position is increased

    if (byteGPS==13){ //if the received byte is = to 13, end of transmission
      stringGPS[stringGPSPosition+1] = '\0'; //character to end the string
      seperatorCounter = 0; //reset counters
      stringCheckCounter=0;

      for (int i=1;i<7;i++){ //verifies if the received command starts with
$GPRMC (this is the data we would like to process)
        if (stringGPS[i]==stringCheck[i-1]){
          stringCheckCounter++;
        }
      }

      if(stringCheckCounter==6){ //if yes, continue and process the data

        boolean firstChecksum = false;

        for (int i=0;i<300;i++){
          if (stringGPS[i]==','){    // check for the position of the  ","
separator
            if(seperatorCounter < 12)
            {
              seperators[seperatorCounter]=i; //and store the location in the
seperators array
              seperatorCounter++;
            }
          }


          if (stringGPS[i]=='*' && !firstChecksum){    // ... and the "*"
            seperators[12]=i;
            firstChecksum = true;
          }
        }

        /*
        the separators array indicate the position of:
         0 = the time in UTC (HhMmSs)
         1 = Status of the data (A=active, V=invalid)
         2 = Latitude
```

```
 3 = Latitude Hemisphere (N/S)
 4 = Longitude
 5 = Longitude Hemisphere (E/W)
 6 = Velocity (knots)
 7 = Bearing (degrees)
 8 = date UTC (DdMmAa)
 9 = Magnetic degrees
 12 = Checksum
 */

//check if the data is active
int data = 1;
for (int j=seperators[data];j<seperators[data+1]-1;j++){
  gps1 = stringGPS[j+1];
}

if(gps1 == 'A') //the data is good
{
  if(debug_gps) {
    Serial.println("-------------------");
  }

  dataAvailable = true; //data is availabe, enable this boolean

  digitalWrite(ledPin, HIGH); //enable the LED pin
  emptyStrings(); //empty previous strings

  data = 0; //reset data variables
  int i = 0;

  // Now we will store all the data one by one in seperate variables
  // this makes it possible for the protocol to send the correct variable
  // when requested.

  for (int j=seperators[data];j<seperators[data+1]-1;j++){
    gps0[i] = stringGPS[j+1];
    i++;
  }
  gps0[6] = '\0';

  i = 0;
  data = 2;
  for (int j=seperators[data];j<seperators[data+1]-1;j++) {
    gps2[i] = stringGPS[j+1];
    i++;
  }
  gps2[10] = '\0';

  i = 0;
  data = 3;
  for (int j=seperators[data];j<seperators[data+1]-1;j++) {
    gps3 = stringGPS[j+1];
    i++;
  }

  i = 0;
  data = 4;
```

```
      for (int j=seperators[data];j<seperators[data+1]-1;j++) {
        gps4[i] = stringGPS[j+1];
        i++;
      }
      gps4[10] = '\0';

      i = 0;
      data = 5;
      for (int j=seperators[data];j<seperators[data+1]-1;j++) {
        gps5 = stringGPS[j+1];
      }

      i = 0;
      data = 6;
      for (int j=seperators[data];j<seperators[data+1]-1;j++) {
        gps6[i] = stringGPS[j+1];
        i++;
      }
      gps6[3] = '\0';

      i = 0;
      data = 7;
      for (int j=seperators[data];j<seperators[data+1]-1;j++) {
        gps7[i] = stringGPS[j+1];
        i++;
      }
      gps7[6] = '\0';

      i = 0;
      data = 12;
      for (int j=seperators[data];j<seperators[data+1]-1;j++) {
        gps8[i] = stringGPS[j+1];
        i++;
      }
      gps8[3] = '\0';

      if(debug_gps)
      {
        Serial.println(gps0);
        Serial.println(gps1);
        Serial.println(gps2);
        Serial.println(gps3);
        Serial.println(gps4);
        Serial.println(gps5);
        Serial.println(gps6);
        Serial.println(gps7);
        Serial.println(gps8);
      }
    }
  }

  stringGPSPosition = 0;
  // Reset the buffer
  for (int i=0;i<300;i++){    //
    stringGPS[i]=' ';
  }
}
```

```
  }
}

void emptyStrings()
{
  //function to clear all the strings
  for (int i=0;i<7;i++){    //
    gps0[i]=' ';
  }

  gps1 = ' ';

  for (int i=0;i<11;i++){    //
    gps2[i]=' ';
  }
  gps3 = ' ';

  for (int i=0;i<11;i++){    //
    gps4[i]=' ';
  }

  gps5 = ' ';

  for (int i=0;i<4;i++){    //
    gps6[i]=' ';
  }

  for (int i=0;i<7;i++){    //
    gps7[i]=' ';
  }
  for (int i=0;i<4;i++){    //
    gps8[i]=' ';
  }
}

void blinkBothLED()
{
  //function to blink both LEDs
  digitalWrite(ledPin, HIGH);
  digitalWrite(ledPin2, HIGH);
  delay(500);
  digitalWrite(ledPin, LOW);
  digitalWrite(ledPin2, LOW);
  delay(500);
}

void killTransaction()
{
  // Function to kill the transaction, used when a time out occured.
  // or when the data was succesfuly sent.
  if(debug_nxt) {
    Serial.println("transaction stopped");
  }
  if(debug_nxt) {
    Serial.println(" ");
  }
  digitalWrite(ledPin2, LOW);
```

```
  alive = false;
  confirmed = false;
  transfer = false;
  transactionNmr = 0;
}
```

D Java code

D.1 Main Lejos class

```
package gps;

import compasNavigator;
import lejos.navigation.CompassNavigator;
import lejos.navigation.CompassPilot;
import lejos.nxt.*;

public class i2c_test extends Thread {
    //variable to store the current position
    private static double[] currentPos = new double[] { 0.0, 0.0 };

    //the array with points to travel to
    private static double[][] travelTo = new double[][] {
            { 51.181465, 5.9576449 }, { 51.181652, 5.9582815 } };

    //current destination point from the destination array
    public static int gotoPoint = 0;

    //temporary strings to store the latitude and longitude
    String latitude;
    String longitude;

    //floats to store the latitude longitude when they have been typecasted
    float latitude_decimals = 0.0f;
    float longitude_decimals = 0.0f;

    //variables to see if the robot is in rang of its destination
    public static boolean inrange = false;
    double distance = 0.0f;

    //the angle between the current and desitnation point
    public static float bearing = 0.0f;
    int distance_meters = 0;

    //variable to check if we are using old or new data
    boolean buffer = false;

    //initiate classes
    public static GPSSensor gps;
    public static CompassPilot pilot;
    public static CompassNavigator navigator;

    public static void main(String[] args) {
        LCD.clear();
        LCD.drawString("calibrating", 0, 0);
        LCD.refresh();

        gps = new GPSSensor(SensorPort.S1); // create a new GPSSensor object

        //Compass navigator variables
        float WheelDiameter = 5.6f;
        float TrackWidth = 18;

        //Full speed ahead
        Motor.A.setPower(100);
        Motor.B.setPower(100);
```

```
        //Create the Compass navigator object, this will help us navigating
        //using the digital compass.
        pilot = new CompassPilot(SensorPort.S2, WheelDiameter, TrackWidth,
                Motor.A, Motor.B);
        navigator = new CompassNavigator(pilot);
        navigator.calibrateCompass();

        //Create and start the navigation thread
        Thread gpsThread = new i2c_test();
        gpsThread.start();

        while (true) {
            // all the LCD output and buttons
            LCD.drawString("heading " + pilot.getAngle(), 0, 5);
            LCD.refresh();

            if (Button.LEFT.isPressed()) {
                LCD.clear();
                LCD.drawString("calibrating", 0, 0);
                LCD.refresh();
                navigator.calibrateCompass();
            }

            if (Button.ESCAPE.isPressed()) {
                System.exit(0);
            }
        }
    }

    public void run() {
        while (true) {
            // read the latitude information
            latitude = gps.doReading(2);

            // pause for 200 ms
            try {
                Thread.sleep(200);
            } catch (Exception E) {
            }

            //read the longitude data
            longitude = gps.doReading(4);

            //convert latitude and longitude to decimal degrees
            latitude_decimals = gps.toDecimalDegrees(latitude, "lat");
            longitude_decimals = gps.toDecimalDegrees(longitude, "long");

            buffer = false;

            if (latitude_decimals > 0) {
                // has to be larger then zero
                currentPos[0] = latitude_decimals;
            } else {
                // otherwise we use data from the buffer
                buffer = true;
            }
            if (longitude_decimals > 0) {
```

```
    currentPos[1] = longitude_decimals;
} else {
    buffer = true;
}

// calculate data using the spherical law of cosines,
// unfortunately the square root function of Lejos is not good
// enough.
// distance = gps.calculateDistance(currentPos[0],
// currentPos[1],
// travelTo[0][0], travelTo[0][1]);
// distance_meters = (int)(Math.abs(distance));

// therefore (for now) use the alternate inRange function)
inrange = gps.inRange(currentPos[0], currentPos[1],
        travelTo[gotoPoint][0], travelTo[gotoPoint][1]);

// calculate the bearing to the destination
bearing = (int) Math.abs(gps.calculateBearing(currentPos[0],
        currentPos[1], travelTo[gotoPoint][0],
        travelTo[gotoPoint][1]));

// print everything on the screen
LCD.clear();
LCD.drawString("lat " + currentPos[0], 0, 0);
LCD.drawString("long " + currentPos[1], 0, 1);
LCD.drawString("inrange " + inrange, 0, 3);
LCD.drawString("bearing " + bearing, 0, 4);
LCD.drawString("buffer " + buffer, 0, 6);
LCD.refresh();

// let the robot rotate to the calculated bearing using the digital
// compass
pilot.rotateTo((int) bearing);

if (!inrange) {
// if we didn't arrive yet we will drive for two seconds

    Motor.A.forward();
    Motor.B.forward();

    try {
        Thread.sleep(2000);
    } catch (Exception E) {
    }

    Motor.A.stop();
    Motor.B.stop();
} else {
// if we did arrive, then goto the next point in the destination
// array
    if (gotoPoint == 0) {
        gotoPoint = 1;
    } else {
        gotoPoint = 0;
    }
}
```

```
                LCD.clear();
                LCD.drawString("lat " + currentPos[0], 0, 0);
                LCD.drawString("long " + currentPos[1], 0, 1);
                LCD.drawString("inrange " + inrange, 0, 3);
                LCD.drawString("bearing " + bearing, 0, 4);
                LCD.drawString("point " + gotoPoint, 0, 6);
                LCD.refresh();

                try {
                    Thread.sleep(1000);
                } catch (Exception E) {
                }
            }
        }
    }
```

D.2 GPSSensor Class

```
package gps;

import lejos.nxt.I2CPort;
import lejos.nxt.I2CSensor;

public class GPSSensor extends I2CSensor {

    public GPSSensor(I2CPort port) {
        super(port); //override the I2CPort class
    }

    public String gpsTransaction(int dataType) {
        //variables
        String message = ""; // string to hold the output message of this
                             // function
        boolean data = false; // boolean to check if the correct datatype
                              // was received by the sensor
        int returnInt = 0; // integer that holds the data type received by
                           // arduino

        byte[] readResponse = new byte[1]; // byte array to store the response
                                           // from the sensor
        byte[] readData = new byte[11]; // byte array to store the actual
                                        // data from the sensor
        byte byteChar; // single byte used as a buffer

        // Start of I2C connection with the sensor, the address of the sensor
        // is 127
        setAddress(127);

        /*
         * The I2C implementation of Arduino is not able to use registers,
         *therefore I wrote a protocol which makes it possible to fetch specific
         * data from the GPS sensor. Steps of the protocol:
         *
         * 1. Fast readings within 10ms of each other to decide what kind
        * of data (e.g. 2 x reading with 10ms break is latitude, 4x reading
         * is longitude)
         * 2. A pause of 30ms to indicate the dataType has been
         * transfered.
         * 3. Read the sensor to receive an confirmation of the datatype.
         * 4. Read the sensor to receive byte 35 (character #) to
         * indicate data transfer starts.
         * 5. Receive a byte array with the
         * actual data.
         */

        // Protocol step 1
        for (int h = 1; h <= dataType + 1; h++) {
            getData(0, readResponse, 1);
            try {
                Thread.sleep(10);
            } catch (Exception e) {
            }
        }

        // Protocol step 2
```

```
try {
    Thread.sleep(30);
} catch (Exception E) {
}

// Protocol step 3
try {
    getData(0, readResponse, 1);
} catch (Exception e) {
    readResponse[0] = 0;
}

byteChar = readResponse[0];
returnInt = (int) byteChar; // convert byte to int

if (returnInt == dataType) { // check if the integer received from
                             // the sensor is equeal to datatype we
                             // want
    data = true;
} else {
    message = "error"; // if the step failed, error message will be
                       // sent
    data = false;
}

if (data) { // if there was no error continue with the protocol
    try {
        Thread.sleep(10);
    } catch (Exception e) {
    }

    // Protocol step 4
    try {
        getData(0, readResponse, 1);
    } catch (Exception e) {
        readResponse[0] = 0;
    }

    if (readResponse[0] == 35) // 35 is the # sign, means data transfer
                               // is starting here after
    {
        // Protocol step 5
        try {
            Thread.sleep(10);
        } catch (Exception e) {
        }

        try {
            getData(0, readData, 11); // reading the string
                                      //(transfered in bytes) from
                                      // the sensor
        } catch (Exception e) {
            for (int i = 0; i < readData.length; i++) {
                readData[i] = 0;
            }
        }
```

```
            for (int j = 0; j < readData.length; j++) {

                String tempstring = ""; // creating a string of the bytes,
                                        // by typecasting the bytes to
                                        // chars
                tempstring += (char) readData[j];

                try {
                    int tempint = Integer.parseInt(tempstring);
                    // for current implementation only integers will be
                    // used, check if it is a string
                    message += tempstring;
                } catch (Exception e) {
                }
            }
        } else {
            message = "error";
        }
    }

    return (message);
}

public String doReading(int dataType) {
    /* this function is a "rescue function" for the actual "gpsTransaction"
     * function. If the gpsTransaction fails, this function will repeat
     * for a fixed amount of times.
     */

    String message = ""; // variable to hold the message by the sensor
    boolean succes = false;
    int counter = 0; // counter to see how many times we tried

    while (!succes) {
        message = gpsTransaction(dataType); // the call to the gps sensor
                                            // transaction function

        if (counter >= 5) { // if we didn't receive a result for five
                            // times, we stop trying for this call
            message = "timeout";
        }

        if (message.equals("error")) {
        // if an error is received by the  gps transaction function, then
        // we didn't succeed
            succes = false;
            message = "error";
        } else {
        // otherwise we succeeded and we can stop trying.
            succes = true;
        }

        counter++;

        try {
            /*
             * pause for a short while before trying again.
```

```
                * Keep in mind that the total repetitions of this function
                * shouldn't be larger then the time between sensor readings.
                */
               Thread.sleep(100);
           } catch (Exception e) {
           }
       }

       return (message);
   }

   public float toDecimalDegrees(String x, String type) {
       /*
        * The latitude and longitude variables from the gps sensor are in
        * the degrees, minutes format.
        * To be able to do calculations with it we need to convert them to
        * decimal degrees.
        * We can do this by splitting the degrees and minutes part, and diving
        * the minutes part by 60.
        */
       float degrees = 0.0f;
       float minutes = 0.0f;
       float decimals = 0.0f;
       String temp;

       if (type.equals("lat")) {
           try {
               degrees = Integer.parseInt(x.substring(0, 2));
               temp = x.substring(2, 4) + "." + x.substring(4, x.length());
               minutes = Float.parseFloat(temp);
               minutes = minutes / 60;
               decimals = degrees + minutes;
           } catch (Exception e) {
               decimals = 0.0f;
           }
       } else if (type.equals("long")) {
           try {
               degrees = Integer.parseInt(x.substring(0, 3));
               temp = x.substring(3, 5) + "." + x.substring(5, x.length());
               minutes = Float.parseFloat(temp);
               minutes = minutes / 60;
               decimals = degrees + minutes;
           } catch (Exception e) {
               decimals = 0.0f;
           }
       }

       return decimals;
   }

   public float calculateBearing(double lat_from, double long_from,
           double lat_to, double long_to) {
       /*
        * The bearing is the angle between two latitude, longitude points.
        * this is needed to navigate the robot to the right direction.
        */
```

```
    // first all the points in degrees are calculated into radians
    double lat_from_rad = Math.toRadians(lat_from);
    double lat_to_rad = Math.toRadians(lat_to);
    double long_from_rad = Math.toRadians(long_from);
    double long_to_rad = Math.toRadians(long_to);

    // the distance between the two longitude points
    double dLong = long_to_rad - long_from_rad;

    // calculation of the bearing angle
    double y = Math.sin(dLong) * Math.cos(lat_to_rad);
    double x = Math.cos(lat_from_rad) * Math.sin(lat_to_rad)
            - Math.sin(lat_from_rad) * Math.cos(lat_to_rad)
            * Math.cos(dLong);
    return toBearing((float) (Math.atan2(y, x)));
}

public double calculateDistance(double lat1, double lon1, double lat2,
        double lon2) {
    /* calculation of the distance using two latitude longitude points
     * using the spherical law of cosines. This formula uses a spherical
     * model of the earth.
     */

    double L1 = lat1;
    double G1 = lon1;
    double L2 = lat2;
    double G2 = lon2;

    // convert to radians
    L1 = Math.toRadians(L1);
    L2 = Math.toRadians(L2);
    G1 = Math.toRadians(G1);
    G2 = Math.toRadians(G2);

    double a = Math.pow(Math.sin((L2 - L1) / 2), 2) + Math.cos(L1)
            * Math.cos(L2) * Math.pow(Math.sin((G2 - G1) / 2), 2);

    // great circle distance in radians
    double angle = Math.toDegrees(Math.sqrt(a));

    // each degree on a great circle of Earth is 60 nautical miles (111120
    // meters)
    double distance = 111120 * angle;

    return distance;
}

public boolean inRange(double lat1, double lon1, double lat2,
        double lon2) {
    /* a method to see if the robot is in range of a latitude longitude
     * point that uses less computing power. This is necessary because
     * both the Haversine and the spherical law of cosines can not be
     * computed by Lejos.
     */
    boolean inrange = false;
```

```
            float difference_lat = (float)Math.abs(lat2 - lat1);
            float difference_lon = (float)Math.abs(lon2 - lon1);

            if(difference_lat < 0.0001 && difference_lon < 0.0001)
            {
                inrange = true;
            }

            return inrange;
        }

        private float toBearing(float rad) // convert radians to degrees (as
        // bearing: 0...360)
        {
            return (float) ((int) (Math.toDegrees(rad)) % 360);
        }

}
```