Niko Stahl
Project Essay
Prof. Wassell
P31

Project Essay

Introduction and Intent

The goal of this project was to design a digital compass that points in the direction of the user's destination. The device was designed with two use cases in mind. The first use case is for hikers, tourists or anyone else who wants to wander though some geographic area without having to worry about the mental load required to read a detailed map. Smartphones with high-resolution digital maps are clearly useful when precise directions are needed, but sometimes only a general sense of direction is desired to explore a new rural area or an unfamiliar neighborhood. Some significant work has been done on generating schematic maps and this project has taken its inspiration from it (Agrawala, 2001).

The second use case is for bicyclists who put themselves in great danger if they are distracted by high-resolution digital maps while biking in traffic. A small device that at a single glance can give a bicyclists a sense of direction could help bikers safely navigate city traffic.

This paper aims to explain my work so that it could be reproduced by others. Throughout the text I make page references to the ATmega644P's datasheet that we have been using in the course[1]. Unless otherwise specified, please assume that all page references refer to that document. A compressed copy of my code is located in my public DropBox directory (bit.ly/1mb3OcH).

Components Used

ATmega644P - 8-bit Atmel Microcontroller

This is Microcontroller we used throughout the module. The features most relevant to this project are its two universal synchronous/asynchronous transmitter receivers (USARTs). Its operating voltage ranges from 2.27 to 5.5 volt.

CMPS10 - Tilt Compensated Compass Module

There are three options for communicating with this module: Serial, I2C and PWM. The bearing I receive is integer between 0 and 255 where 0 indicates north. The module's operating voltage range from 3.6 to 5.5 volt.

Nokia 5110 - 84x48 Graphic LCD

This module is the screen that graphs the compass bearing. It uses a serial bus interface to communicate with the microcontroller. The operating voltage is 2.7 to 3.3 V.

GPT-635T - GPS antenna module

The GPS provides the current location of the device. The direction and distance to the destination is inferred from the current GPS readings. The operating voltage of the device ranges from 3.3 to 5.5 V.

VARTA Microbattery - LIP 553048CC

[1] http://www.cl.cam.ac.uk/teaching/1314/P31/docs/atmega644p.pdf

This is a small lithium-ion battery (48mmx30mmx5.5mm). Its capacity is 740mAh and its voltage is 3.7V. I use these specs when analysing the prototypes efficiency.

A. Interfacing ATmega644P with the Compass Module

Serial Communication

I chose the serial interface to communicate with the compass module, since I am familiar with serial from the module workbooks. The compass uses a default 9600bps baud rate, 2 stop bits, no parity bits and 8 data bits[2] (8-N-2). It is worth noting that the compass module uses 3.3v-5v signal levels. Atmega644P however uses RS232, which uses +/-9 volts for serial communication. Although the compass module's instructions note that this might be an issue, I did not have any trouble running the module.

The Atmega644P has two serial interfaces, USART0 and USART1 (pg. 171). I use USART0 with an asynchronous clock generation (this is the default) for communications with the compass module. The first attempts at setting the Baud rate failed because I had not noticed that my Makefile was pointing the microcontroller to an external clock source. To correct this I needed to change the fuse settings in the Makefile to use the internal clock. The two relevant lines are:

```
   fuses:
       avrdude -p m644p -P /dev/ttyUSB0 -c avrusb500 -e -U
hfuse:w:0x99:m
       avrdude -p m644p -P /dev/ttyUSB0 -c avrusb500 -e -U
lfuse:w:0x22:m
```

Once I activated the internal clock source (8Mhz), I could set the baud rate by setting an appropriate divisor for USART0 found on pg. 196 of the datasheet. USART0 is initialized with the following function:

```
   void USART0_init(void) {
     //set baud rate to 9600 - see pg. 196 of datasheet
     UBRR0 = 51;
     // Set frame format: 8data, 2stop
     UCSR0C = (1<<USBS0)|(3<<UCSZ00);
     // Enable receiver and transmitter
     UCSR0B = (1<<RXEN0)|(1<<TXEN0);
   }
```

The pin 14 (RXD0) is used for receiving and pin 15 (TXD0) is used for transmitting data with USART0. After connecting those pins to their appropriate counterparts on the compass module I could send send commands to the device[2].

[2] http://www.robot-electronics.co.uk/htm/cmps10ser.htm

Data Transfer

To receive the single byte reading of the compass bearing I needed to first send commands to the compass module via USART0. The following code loads the transfer buffer, after waiting for the Data Register Empty (UDRE0) flag to clear. This ensure that the transmit buffer is empty and prevents overwriting data that has not yet been transmitted (pg. 178):

```
//transmit with usart0
void USART_Transmit(char data)
{
  // Wait for empty transmit buffer
  while ( !( UCSR0A & (1<<UDRE0)) );
  // Put data into buffer, sends the data
  UDR0 = data;
}
```

To receive the value of the bearing, I needed to enable the Receive Enable bit (`RXCIE0)` in the UCSR0B register by setting it to 1 (pg. 180). To prevent blocking code that waits for the compass to respond, I set up an interrupt that is triggered once a complete data frame is received. The interrupt function is below:

```
//receive complete interrupt
ISR(USART0_RX_vect) {
  //store bearing
  cbearing = UDR0;
  //flash LED on/off for debugging purposes
  PORTB |= (1<<PB0);
  _delay_ms(20);
  PORTB &= ~(1<<PB0);
}
```

It is important to note that the interrupt will only trigger if the global interrupts have been enabled by setting the appropriate bits in the SREG register (SREG |= 0x80;). This covers the most essentials components for serial communication with the compass.

B. Interfacing ATmega644P with the LCD Screen

The communication between the microprocessor and the LCD screen requires a serial peripheral interface (SPI). The main difference compared to asynchronous serial communication as described between the compass and the microprocessor is that SPI communication keeps separate lines for the data and the clock signals. Since the clock rate is provided on its own line, it is not necessary to calibrate the devices by explicitly setting the baud rate.

The microprocessor only has one SPI serial interface. Since it already uses it to communicate with the programmer, some extra wiring was needed to also use the same interface for communication with the LCD screen. For example, pin PB5 is the output for the MOSI line of both the programmer and the LCD screen. The clock line also overlaps with the programmer's clock line and terminates at pin PB7. PB3 is used for the to wake up the Nokia screen. See Figure 1 in the appendix for a photograph of the breadboard. On the software side, it is worth noting that the screen must be explicitly told if it is to expect a data or a command. This is done by pulling the screen's D/C pin high or low, respectively. I use Brian Jones' code to take control of the slave select (SS) line:

```
void LCDWrite(uint8_t data_or_command, uint8_t data) {
    //PB2 must be wired to the screen's D/C pin
    if (data_or_command) {
         PORTB |= (1<<PB2) ;                          // D/C
    } else {
         PORTB &= ~(1<<PB2) ;
    }
    //taking control of SS line
    PORTB &= ~(1<<PB3) ;                          // SCE
    //this function is declared in Brian Jones' spi2 library
    //it writes data to the SPDR register
    xmit_spi(data);
    //dropping SS line
    PORTB |= (1<<PB3) ;
}
```

The spi library handles the details of writing the data to the SPDR register. This covers the basics steps needed for communicating with the LCD screen via the SPI.

Interface Design

Due to the simplicity of the Nokia screen and the project's intent, the interface design needs to be efficient and legible. I designed the interface with the intent to directly communicates the change of direction necessary to remain on course to the destination. Figure 2 in the appendix shows the interface when the device is pointed in a direction that requires a right turn of approximately 60 degrees to point back into the direction of the destination. A bar that is drawn to the full extent of the screen suggest a full 180 degrees turn. The distance to the destination is shown in meters in the bottom left corner.

Drawing to the Screen

I use Brian Jones' code from the course website to draw characters to the screen[3]. The graphical interface is every time a new compass bearing is received (updates of the compass

[3] http://www.cl.cam.ac.uk/teaching/1314/P31/nokia/

occur at 75hz[4]). To efficiently update the screen and to prevent flashing, I only update the pixels that change from one iteration to the next. The horizontal bar is drawn as two rows of '|' characters. See Figure 1 in the appendix for an image of the interface. The following method maps the reading to the bar and redraws the the bar to reflect the changed orientation:

```
void updateLine(int bearing)
{
  /*
   this flip is necessary since I want to communicate the
   steering that is necessary rather than the deviation from
   the direct heading to the destination
  */
  bearing = 255-bearing;
  unsigned char  i, j, k;
  uint8_t length, lo, hi;
  if(bearing <= 128) {
   length = 42*bearing /128;
   lo = 42-length;
   hi = 42;
  }
  else {
   length = 42* (255-bearing) /128;
   lo = 42;
   hi = 42+length;
  }
   for(j=0; j<84; j++) {
    //draw two rows of '|'
     for(k=1;k<4;k++) {
         gotoXY(j,k);
         if((j>lo && j<hi) || j == LCD_C_X) {
         //draw
         LCDWrite (LCD_DATA,0xff);
         }
         else {
         //clear
         LCDWrite (LCD_DATA,0x00);
         }
     }
   }
}
```

[4] http://www.robot-electronics.co.uk/htm/cmps10doc.htm

C. Interfacing ATmega644P with the GPS Module

I use the microcontroller's second serial interface to communicate with the GPS module. Although the GPS module's datasheet[5] suggests that the default Baud rate is 38.4K I found this to be inaccurate. I could communicate successfully with the GPS module at a 9600 baud rate using 8-bit data frame with no parity and 1 stop bit (N-8-1). Attempting the communication at 38.4K bps did not succeed.

Pin 15 and 16 are the receive and transmit pins for USART1, and they need to be connected to their counterparts on the GPS module. On the software side, the setup is analogous to the steps required for USART0. In contrast to communicating with the compass module, it is not necessary to send messages from the microcontroller to the GPS module. Instead the GPS automatically starts firing messages once USART1 is initialized.

Processing GPS Sentences

The GPS module sends a variety of NMEA-0183 sentences via serial. Once a new character has been received by the microprocessor the following interrupt routine stores the value in a sentence buffer. The NMEA's convention is to indicate new sentences with a '$' character:

```
ISR(USART1_RX_vect) {
  //read new char from the USART1 receive register
  sentence_char = UDR1;
  if(sentence_char == '$') {
      //new sentence
     //start at beginning of buffer
      char_count = 0;
     //flag that previous sentence is complete sentence
     //next iter of main loop will process input
      process_sentence = 1;
  }
  else {
      /*store character in buffer
        note: char_count is uint_8 and
              length of sentence_buffer is 256
              no modular arithmetic necessary - I allow to overflow
     */
      sentence_buffer[char_count] = sentence_char;
      char_count++;
  }
}
```

[5] http://dlnmh9ip6v2uc.cloudfront.net/datasheets/Sensors/GPS/GP-635T-121130.pdf

The GPS module sends a variety of NMEA-0183 sentences via serial. For simplicity I only parse NMEA records of type GPGLL. The NMEA sentences contain comma-delimited values that must be processed to extract the latitude and longitude coordinates of the reading.

It is essential that a complete sentence is not corrupted before it has been completely processed. Hence, the following lines of the program's main while loop pause the interrupts while extracting coordinates from a complete NMEA sentence stored in the buffer:

```
if(process_sentence == 1) {
    process_sentence = 0;
    //avr-lib routine to disable interrupts
    cli();
    processSentence();
    //avr-lib routine to enable interrupts
    sei();
}
```

The actual parsing of the sentence is a straightforward exercise in counting commas and converting the characters representing coordinates to numeric values. The type of the sentence is determined by simply checking its first 5 characters.

Time To First Fix

"Time To First Fix" (TTFF) is the time needed for a GPS module to identify enough satellites for accurate positioning. The GP-635T's datasheet suggests that a cold start (i.e. looking for satellites without prior knowledge of where to search them) requires 27s. In my experiments, this seems like a severe underestimation. Although I did not have a chance to rigorously test statistics on how long it would take to fix enough satellites, I noticed that it took approximately 5 minutes to to fix satellites when the GPS module was sitting next to the window in the MPhil computer lab. The module never seemed to get a fix, when it was sitting in the lab next to my computer during development. Long TTFFs are a known issue[6] for this device, and I will discuss some strategies on how to minimize the TTTF in the section "Analysis and Optimization".

D. Geometric Calculations

Measuring Distance and Angle to Destination

Now that the current GPS coordinates are known, the program must update estimates of the distance and the heading to the destination. The current implementation prioritizes precision over performance. I use the Haversine formula[7] to calculate the distance between two points along the geodesics of a sphere of the same radius of Earth (6,371km).

The current implementation uses doubles that are rounded to three decimal points. This

[6] see discussion at https://www.sparkfun.com/products/11571
[7] http://mathworld.wolfram.com/GreatCircle.html

convention leads to a tolerance of +/- 8 meters for the distance estimate[8]. This seems reasonable under my assumption that the speed of a bicycle is usually under 5 m/s (18 km/h). At this level of precision, a continuous update of the user's position will be required approximately each second in order to capture the continuity of the trip.

I use the initial heading for the route between two spherical coordinates to estimate the direction towards the destination[9]. Note that the interface must communicate the heading with respect to the device's current orientation. To calculate the heading with respect to the current orientation some conversion is necessary to bring the compass reading [0-255] and the heading [-180 - 180 degrees] onto the same scale. The natural evolution of the project led me to convert the heading to the compass's 0-255 scale. After calculating the clockwise difference between the optimal heading and the current bearing, I send that value to the updateLine function:

```
/*
  angleToDest255:
      the angle to the destination in global coords on a 0-255 scale
  cbearing:
      the current compass bearing in global coords on a 0-255 scale
*/
if(angleToDest255>cbearing) {
      updateLine(angleToDest255-cbearing);
}
else {
      updateLine(255-cbearing+angleToDest255);
}
```

The Haversine formula and the heading calculations include several trigonometric functions, many of which require the avr-lib's mathematics library. To link the math library during compilation, I needed to add the "-lm" flag to the Makefile's two lines that call the avr-gcc compiler[10]:

```
avr-gcc ${CPPFLAGS} -Os -mmcu=${MCU} -o $@ -c $^ -lm
...
avr-gcc -Os -mmcu=${MCU} -o $@ $^ -lm
```

E. Analysis & Optimization

[8] This estimate was made using the haversine formula to measure the distance between coordinates <0.00005,0.00005> and <0.00000,0.00000> which are the farthest points apart that would be below the current precision.
[9] The formula I use is sometimes referred to as the "forward azimuth" and is described here: http://www.movable-type.co.uk/scripts/latlong.html
[10] http://www.nongnu.org/avr-libc/user-manual/FAQ.html#faq_libm

Reducing Time To First Fix

The biggest issue with the prototype is the long duration of the GPS module's TTFF. The reason the TTFF is so long in this case is that the GPS module has no prior knowledge of nearby satellites and must therefore search the entire GPS band for a signal. This is called a "cold start" and can only be resolved, if knowledge of previously available satellites is stored. Furthermore, the stored information is only valid if device's location has not significantly changed since the last fix.

The solution suggested by the manufacturer is to put the GPS module into a low power state while it is not being used[11]. The device's power saving mechanism is controlled via any general purpose in/out (GPIO) pin on the microcontroller. The GPIO pin needs to be wired to connect with the module's PWR_CTRL pin (pin 6). Pulling the pin high or low changes the GPS's state to ON or SLEEP, respectively.

This feature could be used in tandem with the microcontroller's sleep functionality[12] in order to minimize the TTFF. When the prototype goes to sleep, it first pulls down the GPS module's PWR_CTRL and then puts the microcontroller to sleep. On wake-up, the microcontroller could then initialize a warm start of the GPS module.

Power Consumption

Observing the power supply display suggests that the operation of the device draws approximately 100mA at 3.6V. This means that the Varta lithium-ion battery that the prototype runs with would be depleted in 7.4 hours. To analyze the power consumption of each individual module I observed the change in current after removing each one individually from the board. What I found is that the GPS module is by far the greatest consumer (~70 mA) followed by the compass (~25mA). Disconnecting the Nokia screen did not cause a noticeable change in current.

These observations motivate the need to decrease the GPS module's power consumption. A first reaction might to suggest that the GPS module should be shut off periodically. However, since this leads to another warm or cold start of the module, I suspect this is a not viable solution since it could take the GPS module a full 30ms[11] to recover its satellites.

An alternative method to reduce the GPS module's power consumption would be to only have it send the NMEA sentences that the device actually uses for its calculations. Since I am only using the GPGLL format, I could shut off the other five formats using the designated commands[13]. I have not been able to test the potential benefit of this approach.

The default update frequency of the GPS is 1Hz and unfortunately this is the lowest available update frequency for this device. To reduce the cost of getting a GPS signal, it might be worthwhile to search for another, more efficient device.

Computational Complexity of Geometric Calculations

The geometric calculations are currently using high precision and expensive formulas that account for the curvature of the earth. Since the use cases that I've discussed probably won't

[11] http://dlnmh9ip6v2uc.cloudfront.net/datasheets/Sensors/GPS/GP-635T-121130.pdf
[12] http://www.atmel.com/Images/doc8267.pdf
[13] http://www.gui-soft.com/public_share/ublox-commands.pdf

include trips of more than 20km, I could save computational cost by modeling the earth's surface as a plane. This would significantly reduce the use of trigonometric functions. For example, using a vectorized implementation of the dot product, the formula for calculating the angle between two vectors requires only one call to acos(). Since trigonometric functions are computationally expensive, this will increase the efficiency of the program. The potential improvements when assuming a planar geometry are summarized in the following table:

Metric	Trigonometric Functions for Spherical Geom.	Trigonometric Functions for Planar Geom.
Angle b/t two vectors	8	1
Distance b/t two vectors	7	0

Furthermore, I could optimize the space needed to store geographic coordinates. Since latitude and longitude are in the ranges [-90,90] and [180,180], respectively, I could use 32-bit integers to maintain my three decimal point precision, as long as multiply the current coordinates by 10^3.

F. Future Work

Along with the optimizations mentioned in the previous section, I would like to add functionality to the device that allows a user to easily set new destination coordinates. A simple way to record the current location would be to attach a button that sends a command to the device to store the current coordinates in volatile memory[14].

A more flexible interface would provide a keypad for the user to punch in new geographic coordinates. The LCD screen would be a suitable visual interface to provide feedback during data entry.

Ultimately, it would be desirable to select a geographic point in a smartphone's mapping application, and to send its coordinates to the device. This communication could probably occur via bluetooth.

References

Agrawala, M., and Stolte, C. 2001. Rendering effective route maps: improving usability through generalization. In Proc. of ACM SIGGRAPH 2001, 241–249.

Appendix

Figures

[14] Thanks to Brian Jones for this idea.

Niko Stahl
Project Essay
Prof. Wassell
P31

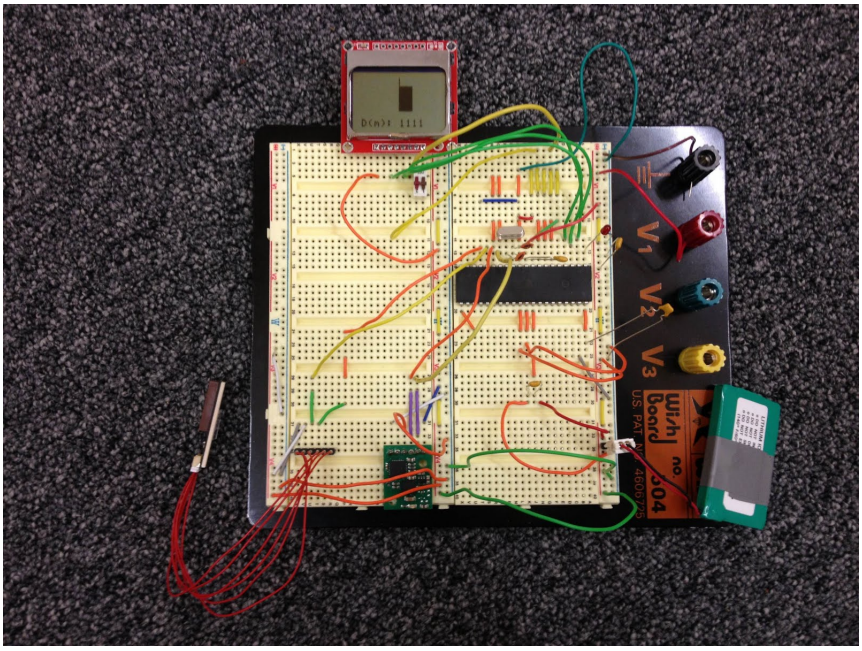

Figure 1: The prototype.

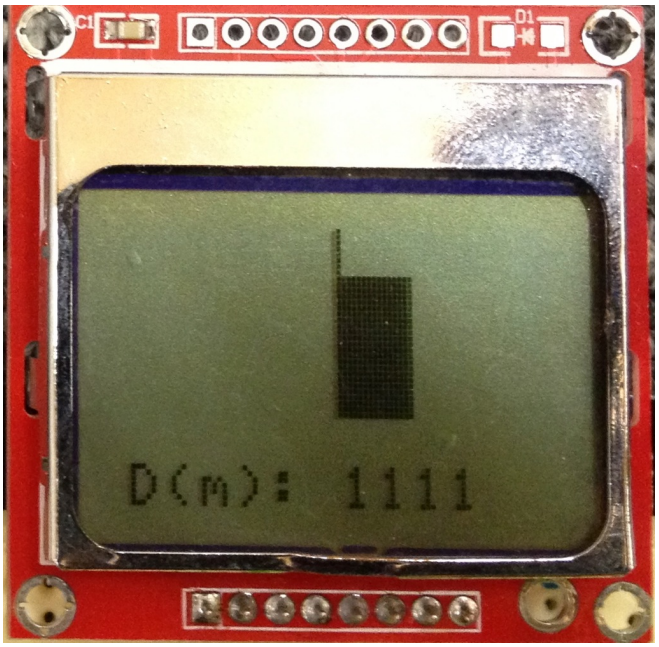

Figure 2: The interface indicating that the distance to the destination is 1,111 meter. A right turn of approximately 60 degrees is required to steer exactly towards the destination.