GeoBuddy

EE459 Spring 2019

Advised by Professor Allan Weber

Team 15

Yi Sui

Brian Suitt

Ling Ye

USC Viterbi

Ming Hsieh Department of Electrical Engineering

Table of Contents

1. Overview

1.1. Background and Motivation

With the increasing popularity with the geocaching activities in large outdoor environment, we were interested in exploring the design of an embedded system used for various latitude and longitude games and activities. After research, we realized the potential of such game device for large outdoor tourism places. Although it is common for people to get an audio guide in indoor facilities like museums, such experience for the outdoor venues is usually provided through a guided tour, and there have been few amusement parks that deployed handheld or smart wristband devices, but these are mostly for a VIP tour experience. Combining our interest in geocaching and outdoor tourism, we decided to pursue the design of a cheaper device for every visitor that would provide an engaging and informative self-guided tour experience incorporating ideas from lat/long game design.

1.2. Product Overview

Currently, most outdoor tour providers spend a large amount of money and resource to hire and train physical people to provide guided tour for their venues. In some cases, such as in many national parks, these tours simply do not exist due to the personnel issues and logistics with coordinating these outdoor experiences. With the intent to help solve this problem in outdoor tours and incorporate ideas from geocaching, GeoBuddy is an integrated all-in-one game device to provide fun and engaging self-guided experience for anything outdoors. We follow the analog of what many indoor museums do, with a handheld device that guides the user through the museum. Our users interact with an intuitive and clear LCD touchscreen that

guides them through the tour, location to location, delivering information or trivia games for each destination.

The reason for a handheld device is that we found many users preferred simply using an integrated device provided to them during tour than going through the hassle of downloading an app for a single experience and worrying about related personal and geolocation data collection. Keeping it all-in-one also allows the tour providers to maintain security and total data control. They can figure out which locations their users spend the most time at, which ones they like the most, and any other pertinent information needed for their review and improvement.

This device is intended to sell to tour providers such as universities, amusement parks and national parks. These providers then distribute these devices to the visitors to use and collect necessary data for better route and activity planning. The following Geobuddy prototype is designed to showcase a tour example designed for the USC Viterbi orientation program, and the prototype features several key locations around the engineering school.

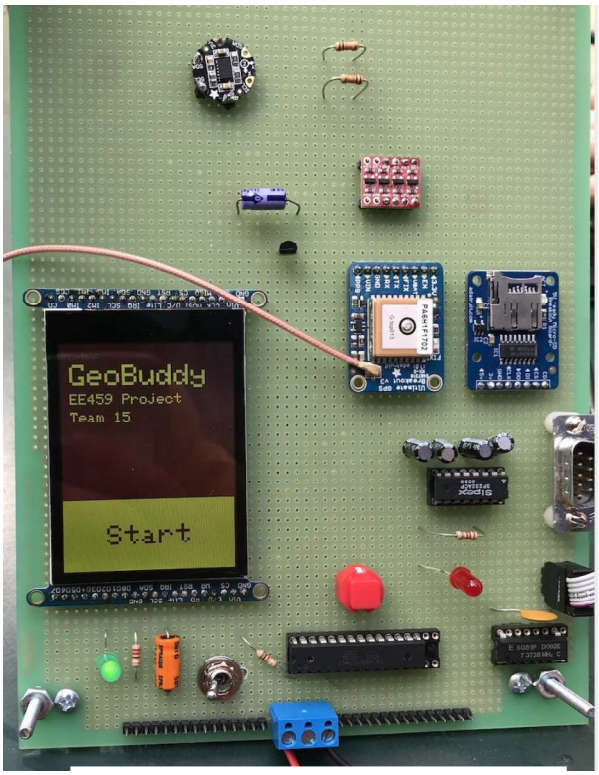

Fig.1 Geobuddy Device

1.3. Design Overview

The overall design is intended to be user-friendly, simple, modular and flexible. The location data for places to visit is all stored in the microcontroller's internal memory, and the

designated location coordinates and related information are pulled out when needed. A GPS module is used to compare user's location with the coordinates of the destination that the user is heading to and update the user through the LCD screen on current heading and distance to the location, along with infographics like the arrow for bearing to guide them. The LCD is a multi-pixel, graphic LCD capable of bright colors and pictures, making the device smooth, and simple to understand. The LCD is also capable of capacitive touch, which was chosen to allow modularity of user interaction and to remove the need for individual buttons to navigate the device. A compass was installed to help direct the user further based on current orientation, but proved having its own issues and was less useful than anticipated.

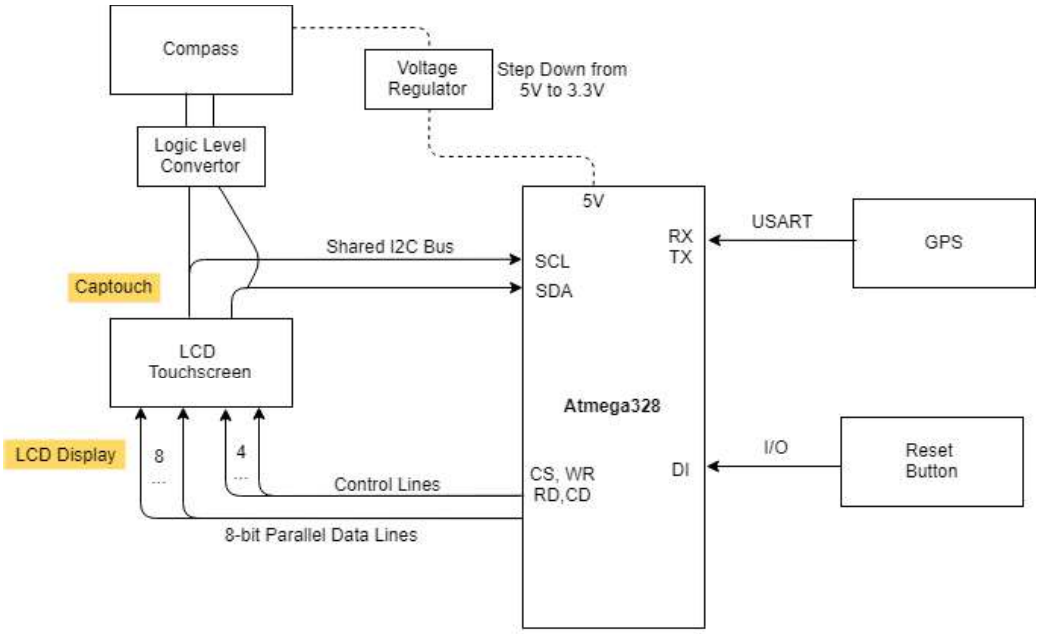

Fig.2 Hardware Block Diagram

The design was intended to be simplistic, but quick to respond to the user. For this we made sure that most devices were timed properly and did not need to be on the same bus, as the compass and GPS can update through their own respective ports, via I2C and Serial respectively. The LCD is wired in 8-bit parallel mode to make sure the user interface was

smooth, as user-friendly is a top priority and we have extra pins to accommodate for 8-bit parallel connections.

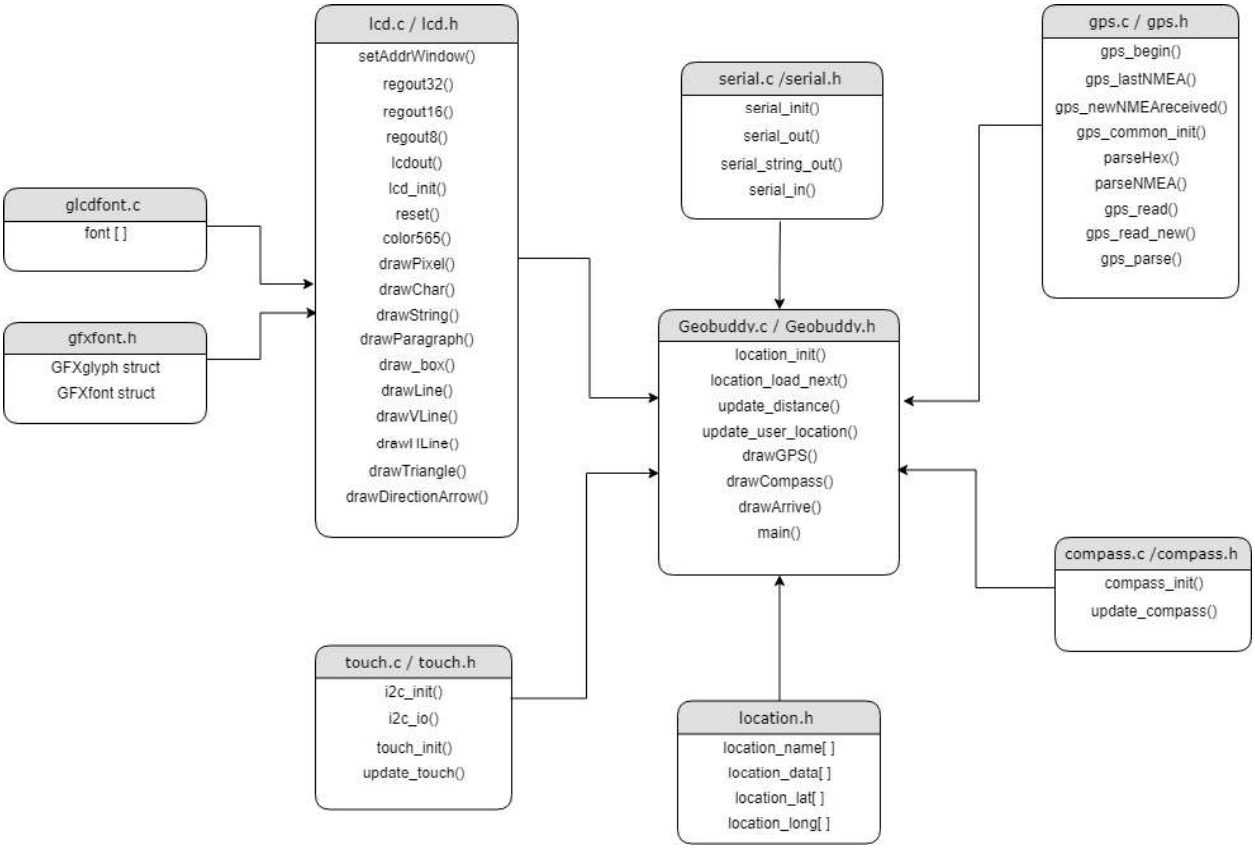

Fig. 3 Software Structure Diagram

As for the software aspect, the entire project is coded in C, as it was found to be easier to directly control the hardware and minimized the program space taken up, allowing us to utilize more function out of our limited RAM and use the extra non-volatile memory for the location data. As shown in Fig.3, we made sure to have the codes modular and easy to adapt or expand based on the different needs of the tour experience and activities. This structure also makes the debugging and testing process for each component easier.

1.4. Cost Analysis

The overall cost of the GeoBuddy comes out to approximately $94.11 per unit, assuming most of that comes from over one thousand units built for mass production. Most of the cost comes from the touchscreen LCD and the GPS. The major components for each unit consists of one Adafruit TFT Capacitive Touch LCD 2090 at a cost of approximately $31.96, one ATmega328p microcontroller at a cost of $1.62, one Adafruit Ultimate GPS 746 at a cost of $31.96, one Adafruit LSM303 Compass chip at a cost of $11.96, one MAX232 Serial Chip for the serial port TTL at a cost of $3.05, one SG51P Clock Oscillator at 7.372 MHz for the ATmega328p at a cost of $1.41, one SparkFun Digital 5v to 3.3v Logic Converter for the compass logic step-down at a cost of $2.95, and one Adafruit LD1117 5v to 3.3v Regulator at $1.00.

Several other miscellaneous parts are needed. One generic through-hole LED is used for a power-on indicator at a total cost of $0.75. A single generic push-button is used as a reset button at a cost of $0.137. Three 10k Ohm resistors are each used for pull-up resistors, one for the reset to the microcontroller, and two more for the SDA and SCL I2C lines for the 3.3v compass, at a total cost of $0.015. One 220 Ohm resistor is used for LED to indicate power-on at a total cost of $0.005. One 47 uF capacitor is used for the 5v to 3.3v regulator at a cost of $0.044, one 10 uF capacitor is used for power regulation at $0.97, one .01 uF capacitor is used for clock oscillator at $0.044, and four 1 uF capacitors are used for the serial port TTL at a total cost of $0.52. Two 3-Pin Male Rectangular Connectors are used for the programming interface to flash code to the microcontroller at a total cost of $1.52. Two 16-Pin Male Rectangular Connectors are used for the Power and Ground buses respectively at a total cost of $0.88. One 28-Pin IC Socket is used for the ATmega328p at a cost of $0.17, and two 16-

Pin IC Sockets are used for the MAX232 chip and the clock oscillator, at a total cost of $1.52. Eight #6x1” bolts and nuts were used for the board at a total cost of $0.35. Approximately fifteen feet of AWG 30 wire was needed at an adjusted total cost of $1.18, and about 6.25” of solder was needed at an adjusted total cost of $0.02.

2. Implementation

2.1. ATmega328p

2.1.1. Hardware

The ATmega328p is an 8-bit AVR RISC-based microcontroller manufactured by Microchip. It contains 32KB ISP flash memory, 1024B EEPROM, 2KB SRAM, 23 general purpose I/O pins and 32 general purpose working registers. In our design, the selected voltage for the system is 5 volts, and the voltage pins are connected to the VCC pin on the ATmega328 microcontroller. One 10 uF capacitor is used to ensure stable power in the device. The microcontroller has both USART and I2C communications capability built in, both of which we take advantage of in our design. The microcontroller gets its clock signal from the SG51P Clock Oscillator, with a .01 uF capacitor connected across the power and ground connections. The oscillators output is then connected to the PB6 pin. The frequency of the clock is 7.3728MHz, which corresponds to a clock period of 135ns.

The LCD utilizes 12 pins of the Atmega328, with 8 data pins and 4 control pins, and has its own internal pull-up resistors. The 8 digital input data pins are PB0, PB1, PD2, PD3, PD4, PD5, PD6, and PD7; the 4 control pins on the LCD module are CS, CD, WR, RD and they are connected to PC3, PC2, PC1, PC0 on the Atmega328 respectively. The capacitive touch is connected via I2C protocol, utilizing PC5 and PC4 for the SDA and SCL pins.

The compass is also connected through I2C protocol to the PC5 and PC4 pins via a bus shared with the capacitive touch. Since the compass operates on 3.3v instead of 5v, a 5v-to-3.3v regulator is used with the 47 uF capacitor to ensure stable stepped-down power for the desired operation of the compass. Two 10k Ohm pull-up resistors are also used after the Sparkfun 5v to 3.3v Digital Logic Converter, one on the SDA line and one on the SCL line to ensure stable signals.

The GPS Tx pin is connected to the microcontrollers PD0 (Rx) pin to receive information. The Rx of the GPS is wired out to the rs232 output only for debugging purposes, but the pin would be utilized later if an EEPROM was added. The MAX232 chip is used here to ensure the TTL signals are properly handled, with the four 1 uF capacitors used to ensure signal integrity. Two of the capacitors are used between the C1+ and C1- pins and C2+ and C2- pins respectively. Two more are used between the VS+ and ground bus and the VS- and ground bus respectively. A reset switch is set up on the pin PC6, the designated reset pin for the microcontroller, with a 10k Ohm resistor used as a pull-up resistor to ensure stability.

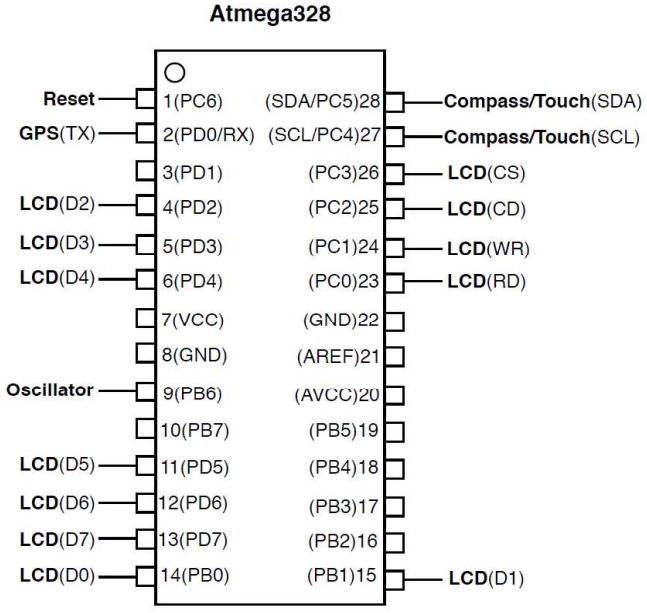

Fig. 4 Atmega328 Pinout Graph

2.1.2. Software

The Atmega328p microcontroller serves as the main control unit in our design. The full program, including the 5 demo locations, takes up 21,220 bytes, approximately 64.8% of the flash memory. The data takes up 1,459 bytes and about 71.2% of the RAM is utilized at maximum runtime. The microcontroller runs the main program in GeoBuddy.c to control all the components used in the design, process data from the sensors, and handle the user interface showing on the LCD display. The entire program is written in the C programming language with the standard C library and the AVR library. Initially when we were testing out the functionality of each components, we had some I2C timing issues, since we have two devices, the capacitive touch and compass, sharing the same I2C bus. To solve this problem, we reviewed our user workflow and noticed that we could separate the I2C data transmission

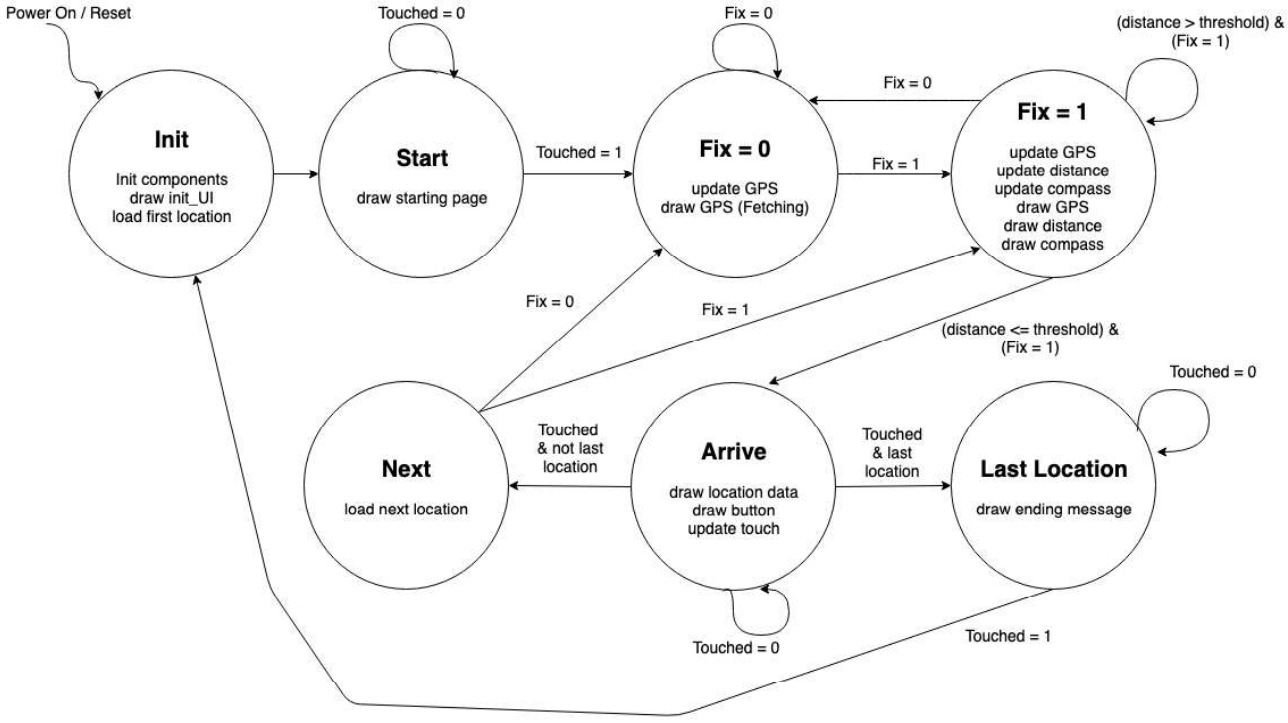

Fig. 5 State Machine in Geobuddy.c

between the microcontroller and multiple slave devices into different steps and states with the

implementation of a state machine. Thus, every feature of GeoBuddy is controlled by a 7-state state machine implemented with an infinite while loop in the main program, as shown in Fig. 5 below.

Init: This state handles the initialization of the device. The specific ports on the microcontroller are set to communicate with the LCD display module, and then the lcd_init() is called to initialize the module. The serial_init() function is called to initialize the USART serial port on the microcontroller for the GPS module. This sets the internal BAUD register by the result of the frequency of 7.372 MHz divided by 16 divided by the BAUD rate of 9600. The Rx and Tx bits of TXEN0 and RXEN0 are enabled in the UCSR0B register, and the UCSR0C register is set up for asynchronous serial, no parity, one stop bit, and 8 data bits. The gps_common_init() function is subsequently called to initialize the GPS module. The i2c_init() function is called to initialize the I2C communication with the compass module and the touch sensing module in the LCD display. After the I2C communication is initialized, the touch_init() and compass_init() functions are called to initialize these two modules. Before this **Init** state ends, it also calls the location_init() function to load the data of the first location which is defined in the location.h header file, sets up UI colors, and draws a blank background. This state transitions to the **Start** state unconditionally.

Start: In this state, the starting page is drawn. The drawString() function is called several times to draw the text “GeoBuddy”, “EE459 Project” and “Team15” in three lines and different font sizes. Then, the draw_box() function is called to draw the yellow background of the “Start” button, and the drawString() function is called again to draw the “Start” text. Every drawString() function call in the main program shares the same “lcd_output_buf” char array

buffer to save memory space. Thus, we call memset(lcd_output_buf, 0, sizeof(lcd_output_buf)) every time after drawString() is called to clear out this shared string buffer. After the start button is drawn, the program enters an infinite while loop. In each iteration of this infinite while loop, the program calls the update_touch() function to fetch data from the touch sensors inside of the touch sensing module through the I2C protocol, and stores the data in the variables: touches(the number of touch points), touchX[0](the X coordinate of the first touch point), and touchY[0](the Y coordinate of the first touch point). One thing to point out is that the touch sensors are able to detect up to 2 touch points, but in our program, we only use the first touch point and ignore the second touch point even if it exists. If a touch point is detected and it falls within the region of the start button, the draw_box() function is called again to draw a blank background and clear out the display. This same method of implementing a touch button on this display would also be used in the other states. When the user presses the start button, the program would break out from the inner infinite while loop handling the touch button, and the state machine would transition to the **Fix = 0** state. Otherwise, the program would move to the next iteration and wait for the user to touch the start button.

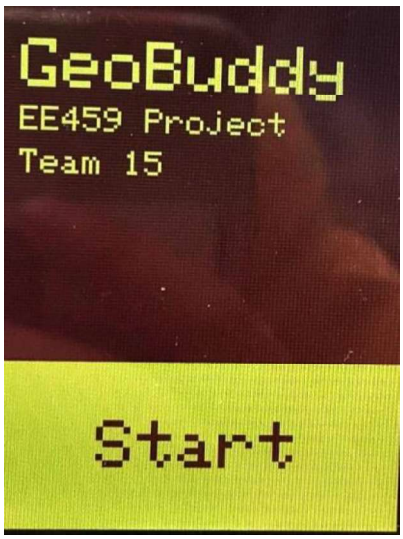

Fig. 6 Welcome Page

Fix = 0: In this state, the gps_read_new() function is called to read data from the GPS module through the USART serial connection. The only data field that the program keeps track of in this state is the "fix" reading. When the GPS module returns fix = 0, it means the GPS module has not localized itself yet. The drawGPS() function is called to draw the GPS data on

the display. In this state, the only data fields to be drawn are “fix”, “fix quality”, and “number of satellites”. The drawGPS() function would utilize drawString() and draw_box() functions to draw and update the GPS data on the display. When the GPS module returns fix = 1, the state machine would move to the **Fix = 1** state. Otherwise, it would stay in the **Fix = 0** state, periodically update the 3 data fields mentioned above and wait for the fix reading to be 1.

Fig. 7 GPS Status Page

Fix = 1: Similar to the **Fix = 0** state, this state first calls the gps_read_new() function to read data from the GPS module. After the GPS data is fetched, the program first checks the value of the fix field. If fix = 0, it means the GPS module fails to localize itself and the GPS data is not valid. In this case, the state machine would move back to the **Fix = 0** state. Otherwise, when fix = 1 it means the GPS data is now valid and ready to be used. The update_user_location() function is called to convert and store the raw data into the variables curr_lat and curr_long. These two variables represent the current user location. Subsequently, the update_distance() function is called to calculate the distance and bearing between the current user location and the current goal location. The drawGPS() function is called to draw and update the following information on the display: the name of the goal location, the coordinate of the goal location, the coordinate of the current user location, the distance between the user and the goal location, and an directional arrow to show roughly which cardinal direction the user should go in order to get close to the goal location. The update_compass() function is called to read the compass heading data from the compass module through the I2C protocol, and the drawCompass() function is used to draw and update the compass heading

information. If the distance between the user and the goal location is less than a predefined threshold (50 feet was used for demo), then the state machine would move to the **Arrive** state. Otherwise, it would stay in this **Fix = 1** state and wait until the user becomes close enough to the goal location. One design decision we made in this state was that we simplified the bearing direction into 8 main cardinal directions and only implemented 8 directional arrows for display instead of showing a very precise and dynamic directional arrow. We made this decision because the user only needs to know the rough direction of the goal location and simplifying the direction arrow display into 8 predefined arrow shapes eliminates the need of calculation every time when the bearing is updated and can save time and memory usage.

Arrive: In this state, the draw_box() function is first called to clear out the display with the background color. Then the drawParagraph() function, which is used to break a long char string array into multiple lines and draw them with drawString(), is called to draw the information or short story of the goal location on the LCD display, as the pictures show below. The draw_box() function is called again to draw the yellow background of a touch button. If the current goal location is the last one in the location list, a "Finish" button would be drawn by drawString(). Otherwise a "Next" button would be drawn. Similar to what we did in the **Start** state, an infinite while loop and the update_touch() function would be used to check if the user has pressed the touch button. If the "Next" button is pressed, the state machine would move to the **Next** state. If the "Finish" button is pressed, it would move to the **Last Location** state.

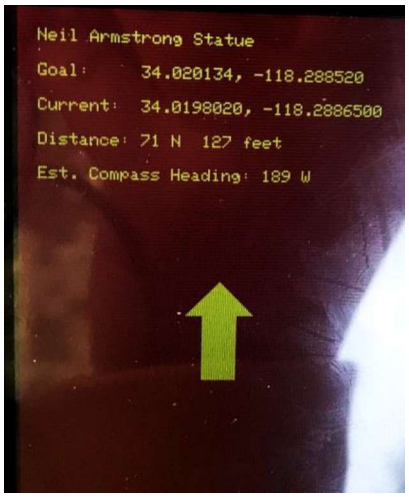

Fig. 8 Direction Page

Next: This state first calls the draw_box() function to clear out the display. Then the location_load_next() function is called to load the data of the next goal location defined in the location.h file. The gps_read_new() function is called to determine whether the fix data field is 0 or not. If the fix = 0, it means the new GPS data is not valid and the state machine would move to the **Fix = 0** state. Otherwise it means the GPS data is valid and the program needs to begin to guide the user to the next goal location so the state machine would move to the **Fix = 1** state.

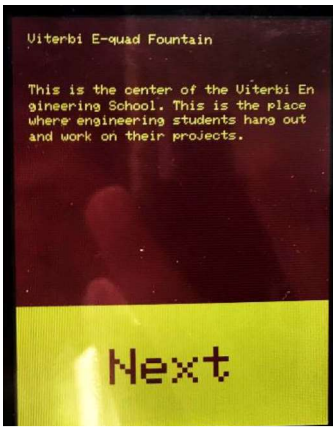

Fig. 9 Information Page

Last Location: In this state, the draw_box() function is first called to clear out the display. The strcpy_P() function is called to copy the ending message into lcd_out_buf, and then the drawParagraph() function is called to draw this ending message on the screen as the picture shows below. Similar to what the program does in the other states, a "Back" button is drawn with the draw_box() and drawString() functions. An infinite while loop is used to check if the user has pressed the touch button using the update_touch() function. When the button is pressed, the state machine would move back to the **Init** state.

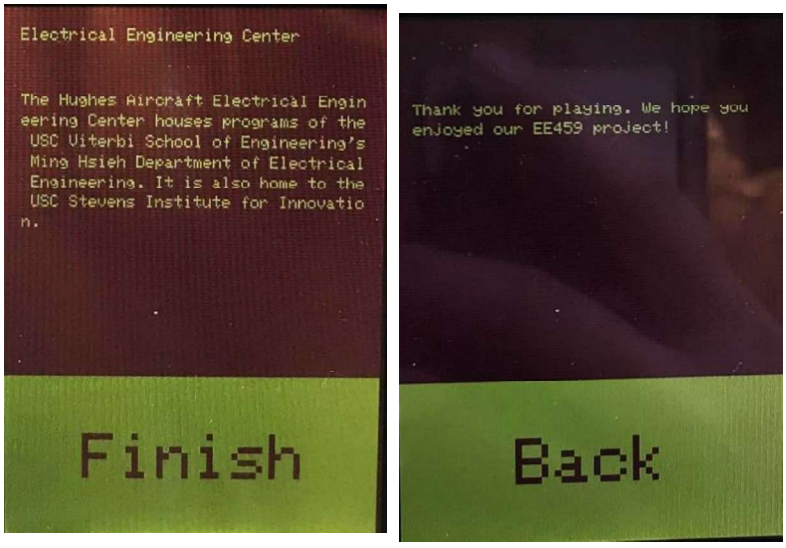

Fig. 10 Finish and Thank You Page

2.2. Adafruit Ultimate GPS 746

2.2.1. Hardware

The Adafruit Ultimate GPS 746 is an all-in-one GPS unit, operating with 66 channels, 10Hz updates, only 20mA current draw, capability to talk with 22 satellites, and complete with its own internal antenna. The GPS communicates the acquired data to other devices via serial. The device was wired in with our 5v bus and to the ATmega328p via the Rx serial port. We only needed the Rx line from the microcontroller to the Tx line from the GPS since we only receive data from the device, not send. This freed up the Tx line on the microcontroller for the serial output that we used for debugging, making for a simpler wiring and no need for a mux to control the line. The received data is in character streams containing the data received from the antenna, including fix, fix quality, and the latitude and longitude values.

2.2.2. Software

Once the GPS is initialized in the main program, it can start reading data. When the gps_read() function is called, the device listens on the Rx line until it has received an entire NMEA, the string of characters that completely describes one of the types of GPS data. This is checked to make sure it is new data since we last read with the gps_newNMEAreceived() function, and then called into the gps_parse(char* nmea) function. This function first runs a checksum to ensure that the data is not corrupted, and if that checks out, it parses for the type of data received. There are many types, such as GPGGA, GPGLL, GPRMC, GPGSA, GPGSV, GPVTG, and GPGST. We are only interested in the GPGGA and the GPRMC for this device, so we are looking for data streams that start with a “$GPGGA” or a “$GPRMC” and then splitting those strings to get each individual fields data. As we split the string into multiple

substrings, we check to make sure there is a field, and if there is, we store it as current data and move to the next one. With latitude and longitude, the data is stored in various stages, with degrees and minutes calculated and stored separately, and then an overall latitude and longitude is constructed out of these. These float structures are used to compare with our location in the main program. The gps_read() function is called periodically, with the gps_parse() function being called to update the fields as well.

2.3. Adafruit TFT Capacitive Touch LCD 2090

2.3.1. Hardware

The Adafruit 2.8" TFT Capacitive Touchscreen LCD is a graphic LCD capable of 240x320 pixel individual RGB control. Technically, the screen is only rated for single-touch, although we were able to detect two touches on most occasions. The display has built in RAM buffering to reduce the workload on the microcontroller, with both an 8-bit digital or 5-bit SPI mode to choose from to interface with the screen. We chose to the 8-bit digital mode due to the speedy response and ease of implementation since available pins were not an issue for us. The touch communicates over I2C, using only two pins for the SCL and SDA lines. The device is capable of running at both 3.3v and 5v, but for simplicity we run it at 5v as that is the main bus we used.

2.3.2. Software

The software of the captive touch LCD is broken into two files to enable easy debugging and testing. The lcd.c/lcd.h contains codes for all the display functions and capacities. It defines the basic functions that interact with the registers and direct outputs of the LCD as well as high-level functions to draw specific colored geometric items, such as lines, triangles and

boxes, and print out character and paragraph with different colors and fonts. The i2c.c/i2c.h includes the I2C protocol implementation and functions for detecting points of touch on the screen and its position.

Within lcd.h, the pin addresses for the 8 data pins and 4 control pins (CD, RD, CS, WR) are defined. The reset is connected to the overall reset pin of the atmega328 chip. The CD pin select either 0 for the command register(address) or 1 for the data register. RD and WR controls whether to read and write operation of the LCD. CS is the chip enable. After defining the pins, several pin bitwise operations are defined to manipulate the above pins, such as LCD_RD_Active and LCD_CD_Data, for readable coding for functions later in the lcd.c. The last part of the lcd.h defines the readable names for several necessary internal of the LCD module.

For the lcd.c, lcd_out() is the fundamental function which writes a byte to the LCD. With lcd_out and bitwise operations defined in lcd.h, the regout8(), regout16() and regout32() each output a different length register address and data value to the LCD. The regout32() function outputs an 8-bit register address and 32-bit data value to the LCD; regout16() function outputs a 16-bit register address and 16-bit data value; regout8() outputs an 8-bit register address and 8-bit data value to the LCD. These three register output functions satisfy the different functions' need to output different combination of data and address to the LCD. The setAddrWindow() function takes two pairs of the coordinates and sets the LCD address window on the screen. The color565() changes 3 RGB byte values into a 16-bit color value used as input for the LCD display. The lcd_init() function initializes the overall LCD with its display, power and memory. The reset() function resets the entire LCD. These functions are above are the bottom-level implementation need to start and reset the LCD as well as writing to the specific register for display purpose.

The following functions draws shapes and prints out paragraphs with the help of the functions defined above. The drawpixel() function draws one pixel with given coordinates and color. The draw_box() function utilized two set of coordinates on the diagonal ends of the rectangle to draw a rectangle. The drawLine uses two sets of coordinates and the color to draw a straight line. The drawTriangle function utilizes the drawLine() function and the coordinates for the three points to draw the triangle. Utilizing the drawTriangle() and draw_box() functions, the drawDirectionArrow() function could draw 8 different directions (N, S, W, E, NE, SW, NW, SE) based on the data input. Aside from displaying shapes on the screen, the capacity of printing character, string and paragraph is implemented as well. The gfxfont.h defines two structs for the font, and the glcdfont.c is an open-source standard ASCII 5*7 font file, which could be replaced with any open-source or self-created font file in the future to change the font shown upon the screen. These fonts are stored within the program space of the microcontroller and can be accessed through the inclusion of the <avr/pgmspace.h>. The drawChar can draw character with different colors and sizes. The character with size=1 utilizes the drawPixel() function and these with size larger than 1 are drawn through a pixelated way with the draw_box() function. The drawString is built on top of drawChar and prints out the string with the first character at the location (x, y), and the drawParagragh breaks a long string into multiple lines and then uses drawString() function to draw out each line.

The i2c.h and i2c.c are used for the capacitive touch sensing. The touch uses the I2C protocol to communicate with the microcontroller. The I2C protocol are initiated with the correct prescaler and baud rate. The touchscreen is initialized through the touch_init(), which contains the definition of the device address and register needed to access to get the correct address. The update_touch() functions update two points of touch detected with their respective

coordinates on the screen. In the main program, these points of touch would compare with the touch area defined and incur the desired next steps.

2.4. Adafruit Compass and Accelerometer LSM303

2.4.1. Hardware

The compass is an Adafruit LSM303 triple-axis accelerometer and magnetometer, capable of measuring acceleration and magnetic fields in three dimensions. The chip is interfaced with I2C to the ATmega328p, as such it only needs four connections in the SDA, SCL, 3.3v Power, and Ground. Since the device requires an overall 3.3v source, a voltage regulator and digital logic converter was needed to send and receive data on the I2C bus. Once these were installed, the compass stored the data from the magnetometer and accelerometer in internal registers until polled from the main program.

The magnetometer proved to be a very sensitive instrument and unfortunately did not quite work the way we wanted it to. The magnetic fields generated by the signals traveling in the wires across our board gave off slight magnetic fields, but enough to be detectable and warp the measurements on the device. We found that when the compass was angled more in line with the wires, we got better readings, but that only worked for North, as when we turned the device the wires caused significant interference, resulting of results that were, at best, off by ten degrees, and at worst 80 degrees.

2.4.2. Software

The compass needs to be initialized through compass_init(), particularly the magnetometer. As the magnetometer has its own specific address, this must be used to talk to the device and differentiate it from the accelerometer also on board. The gain on the magnetometer must first

be sent across to get appropriate readings, with the internal software in the main program also updating. From there, the device is polled to ensure it is working properly, and then data can start to be read. The raw 16-bit x, y, and z data stored in the magnetometer needs to be pulled into the program. This is done using one function in update_compass() and reads in each 8 bits part of the coordinate data at a time. Each coordinate is pulled in as the high and low 8 bits and then combined into one 16-bit signed integer using bit shifting. These values then need to be adjusted by the gain we have set for the magnetometer and then by the Earth's magnetic influence to get the proper readings. From here, the actual x and y coordinates are used in the atan2(y, x) function to determine the compass heading. The atan2(y, x) function specifically retains the signs of each coordinate to ensure an accurate heading, as the tangent function can reflect a false positive in the third quadrant otherwise.

3. Future work

3.1. Full Product Integration

For the full product integration, we would need to build a shielded PCB to house all the electronics, provide the device with a power source, and then position the compass for minimal interference. With these, we feel we could make the compass accurate enough to get measurements to guide the user. After that, more modularity would be programmed into the state machine to allow trivia, data collection, and multiple game modes, like scavenger hunts, and user-selected locations.

On the packaging side, we would need someone to fabricate a cover that would be ergonomic to the hand and easy to work with, that would fit everything we needed inside.

Further testing would be done to ensure that this did not block the magnetometer and allowed full range of use of the device

3.2. Improvements

There are a few things we could do differently. By swapping out the touch LCD and utilizing a few more pins for buttons, we could half our cost on the LCD, a significant $15. While this is less modular than using the touch screen for buttons, it would make the device more appealing in cost-effectiveness. We could also use the accelerometer to assist with the magnetometer to increase accuracy and retain that as we move, comparing where we should have moved to the compass readings.

Storage will also be something that needs to be improved upon, as the program space is not optimal for large data sets. An EEPROM would be best suited for this, as we could just use the serial lines to connect and it would take only one additional pin to use for the multiplexor to separate it from the GPS.

4. Appendix

4.1. Part list and Cost Analysis

Components	Price (100+)	Notes
ATmega328 MCU	$1.62	Digikey Part #: ATMEGA328-PU-ND
2.8" TFT LCD with Cap Touch Breakout Board	$31.96	Adafruit 2090
Ultimate GPS Breakout Board	$31.96	Adafruit 746
Compass chip	$11.96	Adafruit Flora LSM303
MAX232 Serial Chip	$3.05	Mouser Electronics Part #: 700-MAX232EPE
SG51P Clock Oscillator	$1.41	Mouser Electronics Part #: 732-531P8.0000 MCR
Sparkfun Digital 5v to 3.3v Logic Converter	$2.95	Mouser Electronics Part #: 474-BOB-12009
5v to 3.3v Regulator	$1.00	Adafruit LD1117
Through-Hole LED	$.75	Digikey Part #: 754-1615-ND
Push Button	$.137	Mouser Electronics Part #: 611-PTS645SL43SMTR92
3x 10k Ohm Resistors	$.015	Digikey Part #: CF14JT10K0TR-ND

220 Ohm Resistor	$.005	Digikey Part #: CF14JT220RTR-ND
47 uF Capacitor	$.044	Mouser Electronics Part #: 594-D471K20Y5PH63L6R
10 uF Capacitor	$.97	Digikey Part #: 490-7518-3-ND
.01 uF Capacitor	$.044	Mouser Electronics Part #: 594-D103M29Z5UH6UJ5R
4x 1 uF Capacitor	$.52	Digikey Part #: 399-13923-2-ND
2x 3-Pin Male Rectangular Connectors	$1.52	Digikey Part #: 732-5316-ND
2x 16-Pin Male Rectangular Connectors	$.88	Digikey Part #: 732-5327-ND
28-Pin IC Socket	$.17	Digikey Part #: ED3050-5-ND
2x 16-Pin IC Sockets	$1.52	Digikey Part #: ED3316-ND
8x #6x1” Nuts and Bolts	$.35	Home Depot
15’ 30 AWG Wire	$1.18	Adafruit 3164
6.25” Solder	$.02	Digikey Part #: 82-140-ND
Total	**$94.11**	

4.2. Schematic Diagrams