The SCavenger

EE 459, Spring 2025

Professor Weber

Embedded Systems Design Laboratory

Team 3: Jeremy Pogue, Karla Vasquez, Allan Anderson

In collaboration with Otis College of Art and Design

Introduction

In today's world, unwinding from everyday life often means sprawling on a couch and indulging in a familiar pastime. This go-to activity is not necessarily a video game or an app—it can be as simple as reading or shaping cookie dough while baking. While these activities are entertaining and relaxing, they do not get the blood circulating. With the average person spending about 45% of the workday sitting and dedicating 60% to 80%[1] of leisure time to sedentary entertainment, an increasingly inactive lifestyle may lead to various chronic illnesses such as cancer, dementia, and respiratory diseases.

But unwinding does not have to mean turning to sedentary activities. Therefore, *The SCavenger*, a journal-inspired product was designed to reimagine relaxation by turning movement into an adventure. Intended to spark curiosity and connection, The SCavenger uncovers a meaningful story through solving clues and searching for coordinates. Ultimately, young adults are encouraged to engage both mind and body alongside friends, building healthier habits and friendships while enjoying the world around them[2].

Project Overview

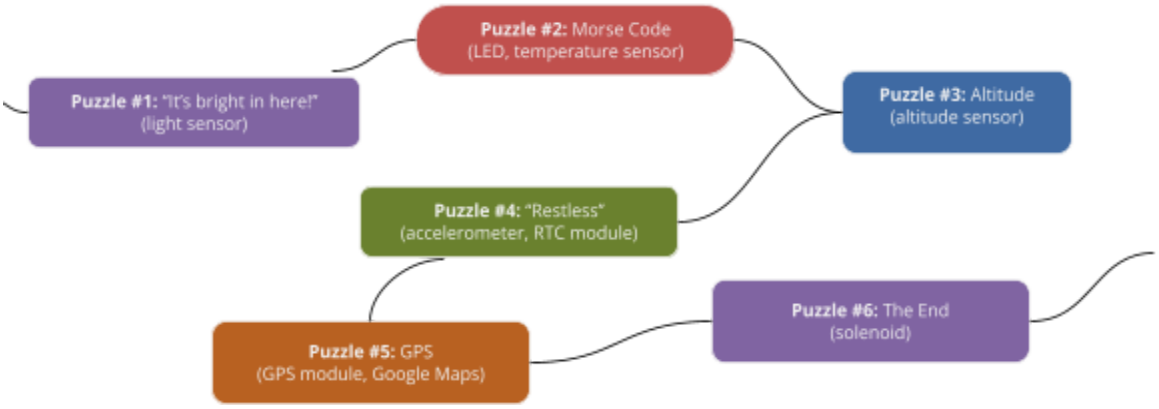


Figure 1: The SCavenger's puzzle gameflow

[1] U.S. Bureau of Labor Statistics. "Sitting and Standing." *Occupational Requirements Survey*, 2023,
[2] Goyal, Jyoti, and Gurseen Rakhra. "Sedentarism and Chronic Health Problems." *Korean journal of family medicine* vol. 45,5 (2024): 239-257. doi:10.4082/kjfm.24.0099

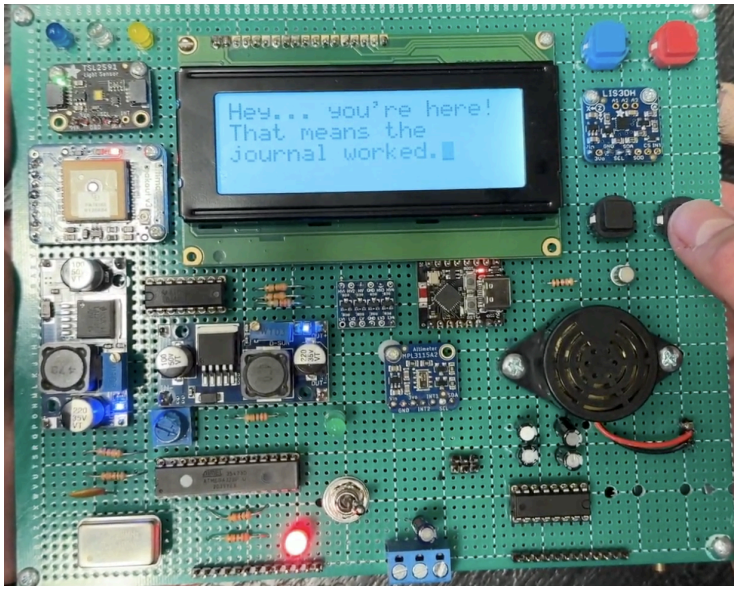
The SCavenger is an interactive adventure journal that integrates problem-solving, physical activity, and challenges that the player must complete in order to unlock the secret compartment found inside the journal. The game follows a chronological storyline, with six puzzles embedded in the game, each requiring interacting with multiple sensors and components. As the journal is portable, this allows the player to complete the challenges required both indoor and outdoor, and can be played either solo or with friends. Each session is unique because after the user completes *The SCavenger*, the player is asked to replace the item inside the secret compartment with another significant object they desire the next player – a friend, perhaps – to find. This enables replayability and helps foster a sense of identity surrounding the game.

The instructions, while explicit, will get increasingly harder to understand or decode. While solving a puzzle, a total of three hints will be available to the player. They will be displayed on the LCD screen upon the player pressing the hint button. Using their provided instructions, the player modifies the device's environment. The device observes this change. If the change was "correct", the player receives a reward. The LCD screen reveals a memory from the narrator's past, and the player proceeds to the next puzzle. If the change was "incorrect", nothing happens.

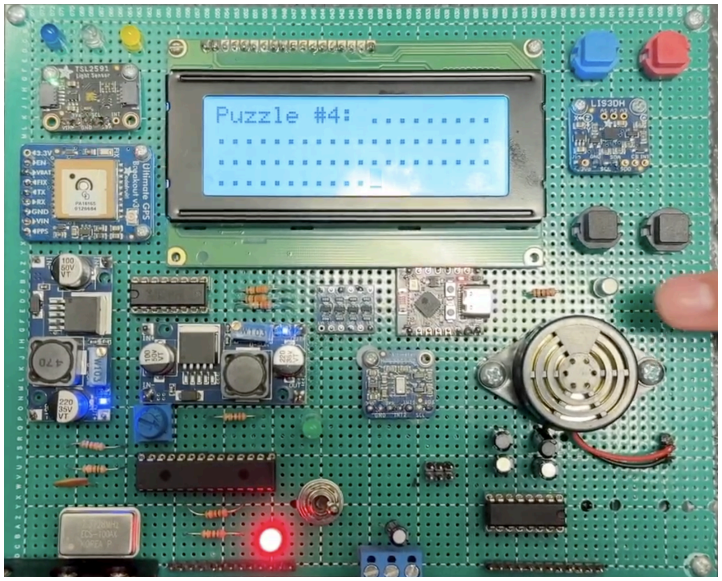
Players will progress through games in a linear order, and each game will follow a similar sequence of events:

1. Upon opening up the "journal", the user will become familiar with the **User Interface.**
    - **Four Buttons** (upper right hand corner of board)
        - **Power** Button
        - **Clue** Button
        - **Previous** Button
        - **Next** button
    - **LCD Screen** (center of board)
        - Displays puzzle prompts, clues, and success messages
    - **Buzzer** (bottom right of board)
        - Plays "victory jingle" and "narrator dialogue"

2. Upon **Powering On**
    - LCD displays a set of instructions
        - Navigate through pressing the next button
        - Retrace steps pressing the previous button
    - Device initializes sensors and sets internal game state

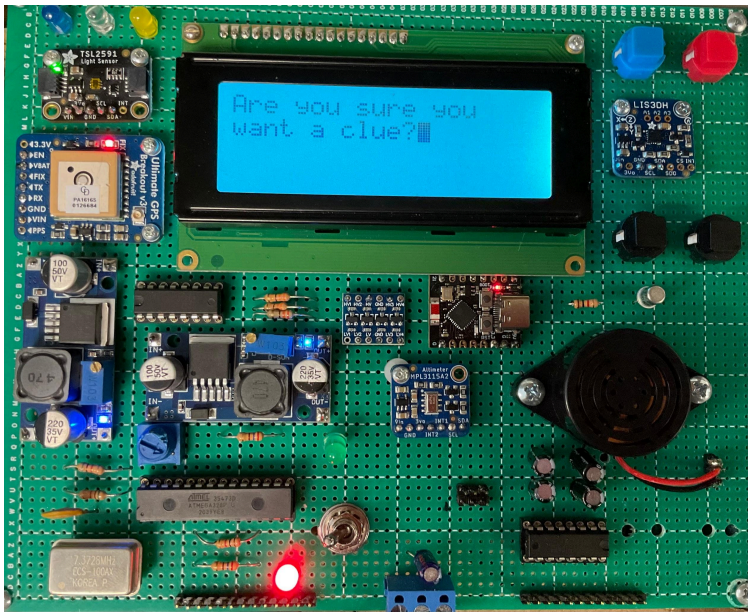


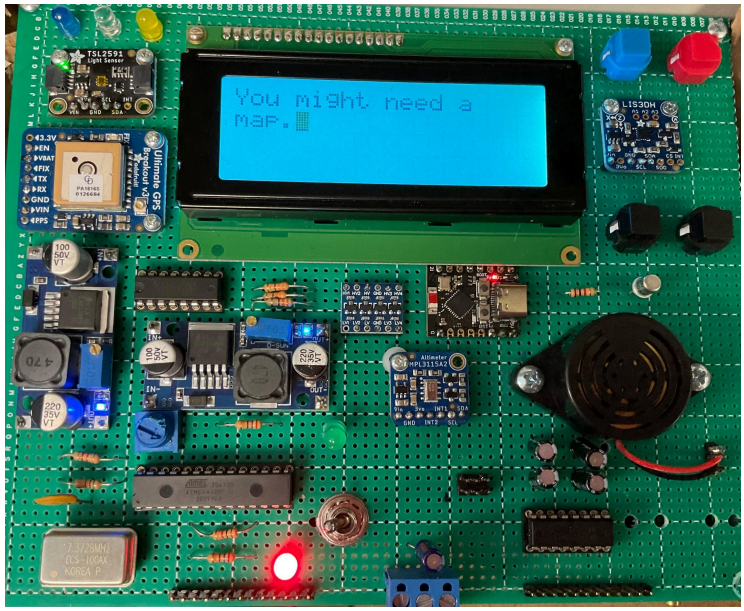
3. **Puzzle Execution**
    - User attempts challenge (e.g., walks to meet a step goal, solves a Morse clue)
    - Sensor data is actively monitored and compared to thresholds
    - Feedback (clue, encouragement, or error) is given via:
        - LCD text
        - Buzzer sounds
        - LED patterns
        - **Clue** button

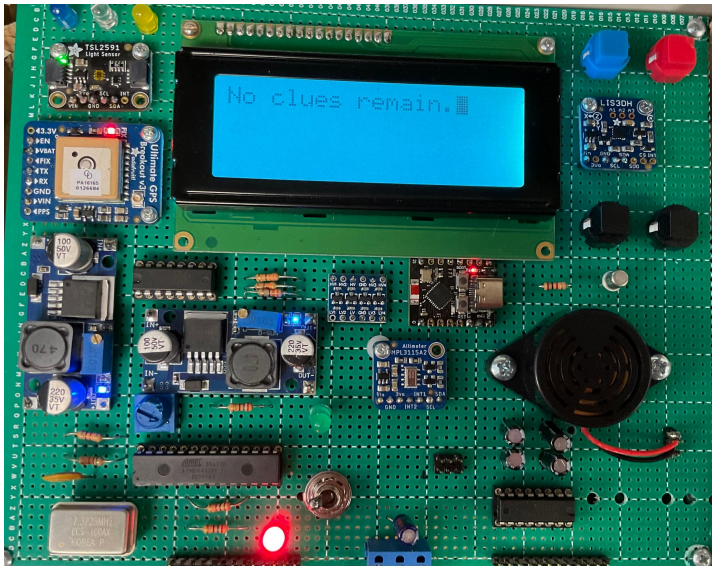


4. **Player Stumped**
    - User presses blue clue button

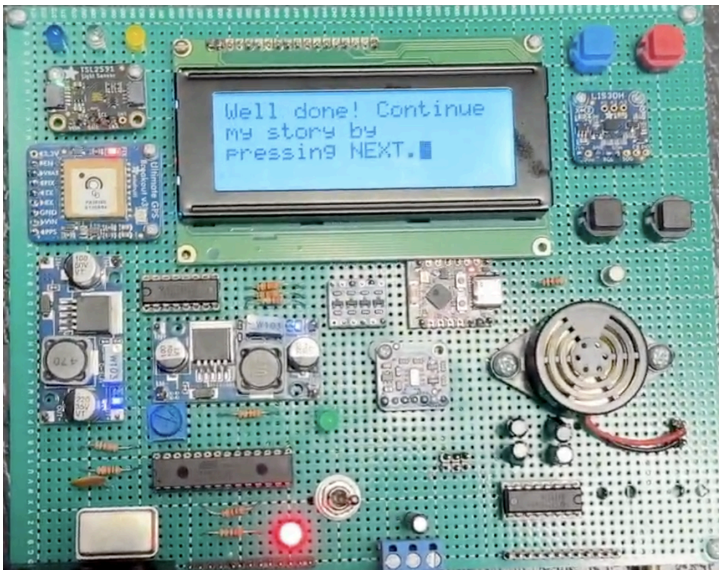
- LCD screen double checks if user wants to use clue
    - A max of 3 clues are given
- Clue is given!







5. **Puzzle Solved**
    - Game logs puzzle as complete
    - User is notified (e.g., “Puzzle 3 Solved!”, victory jingle)
    - Next puzzle unlocked



6. **Puzzle Unlocked**
    - After all puzzles complete, secret compartment unlock is triggered (e.g., via solenoid and display message)
    - Game ends with closing message

Overview of the Subsystems

The following active components formed the core functional subsystems of *The SCavenger*:

Control and Processing Subsystem

ATmega328P Microcontroller

- Role: Central controller managing game logic, sensor data, and user interactions
- An 8-bit AVR microcontroller that interfaces with all peripherals including sensors, LCD, buzzer, and GPS

Sensing Subsystems

TSL2591 Light Sensor

- Role: Used for brightness-based puzzle (e.g "It's bright in here" puzzle)
- High dynamic range digital light sensor that outputs lux readings over I2C; detects environmental light levels.

LIS3DH Accelerometer

- Role: Used for step-counting in movement-based puzzle (e.g "Restless" puzzle)
- 3-axis accelerometer that communicates via I2C; measures motion and acceleration to detect steps.

MPL3115A2 Altitude/Temperature Sensor

- Role: Used for measuring player's elevation where player must reach a minimum height threshold to proceed (e.g "Altitude" puzzle); also used to meet minimum temperature requirements
- Digital pressure sensor configured to operate in altimeter mode to provide altitude data in meter; also configured in temperature mode

MTK3339 GPS Module

- Role: Used to track player location for geographic challenge triggers
- High-sensitivity GPS receiver that provides NMEA sentence data over UART; tracks latitude, longitude, altitude

Communication Subsystem

Buck Converter (DC-DC Step-Down Regulator)

- Role: Regulates the voltage from four AA batteries (nominal 6V) down to 5V to power the ATmega328P and connected peripherals safely

Power management Subsystem

ESP32-C3

- Role: Intended for WiFi functionality to display web pages of GPS coordinates player can choose from to visit (e.g GPS puzzle)
- Controlled in 4-bit mode using ATmega digital pins; essential for real-time player communication

User Interface and Output Subsystem

20x4 Character LCD (Parallel Interface)

- Role: Displays puzzle clues, hints, and sensor feedback (i.e the narrator)
- Controlled in 4-bit mode using ATmega digital pins; essential for real-time player communication

Buzzer

- Role: Provides audio cues, feedback, or alerts during gameplay
- Small piezoelectric buzzer controlled with PWM

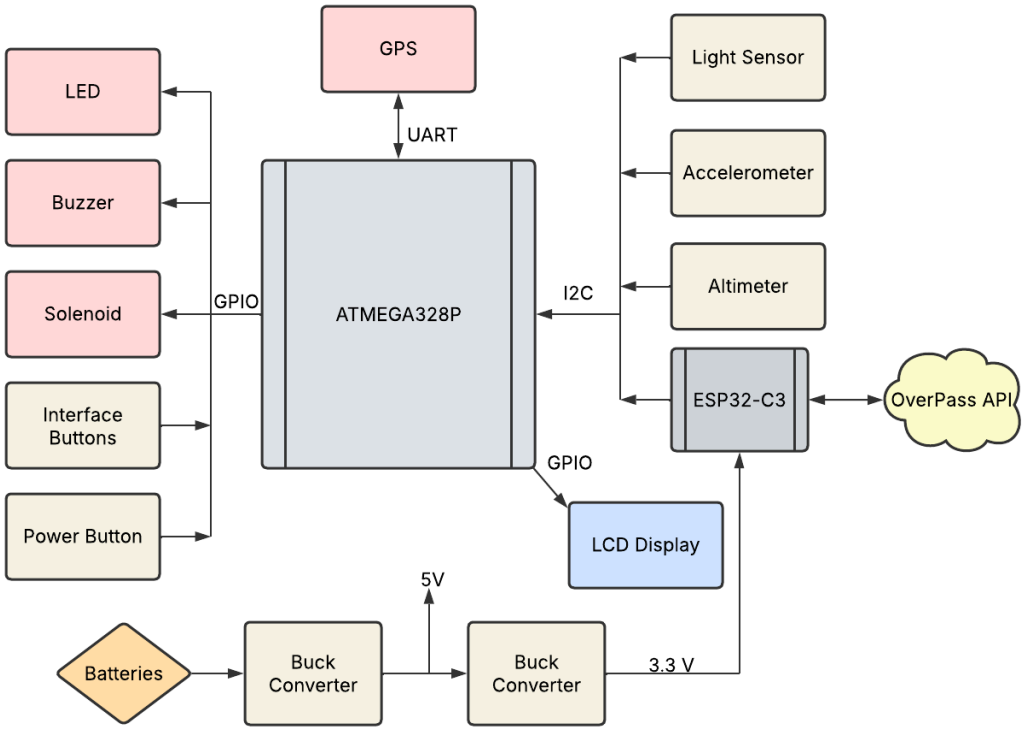
LEDs (Red, Blue, Clear)

- Role: Used for visual effects, clues (e.g Morse code blinking in Freeze Me puzzle)
- Basic digital output components indicating puzzle states or time pressure visually

Push Buttons (Power, Clue, Navigation)

- Role: User inputs to interact with game, request hints, or progress through puzzles
- Debounced in software, these digital inputs trigger specific game state transitions

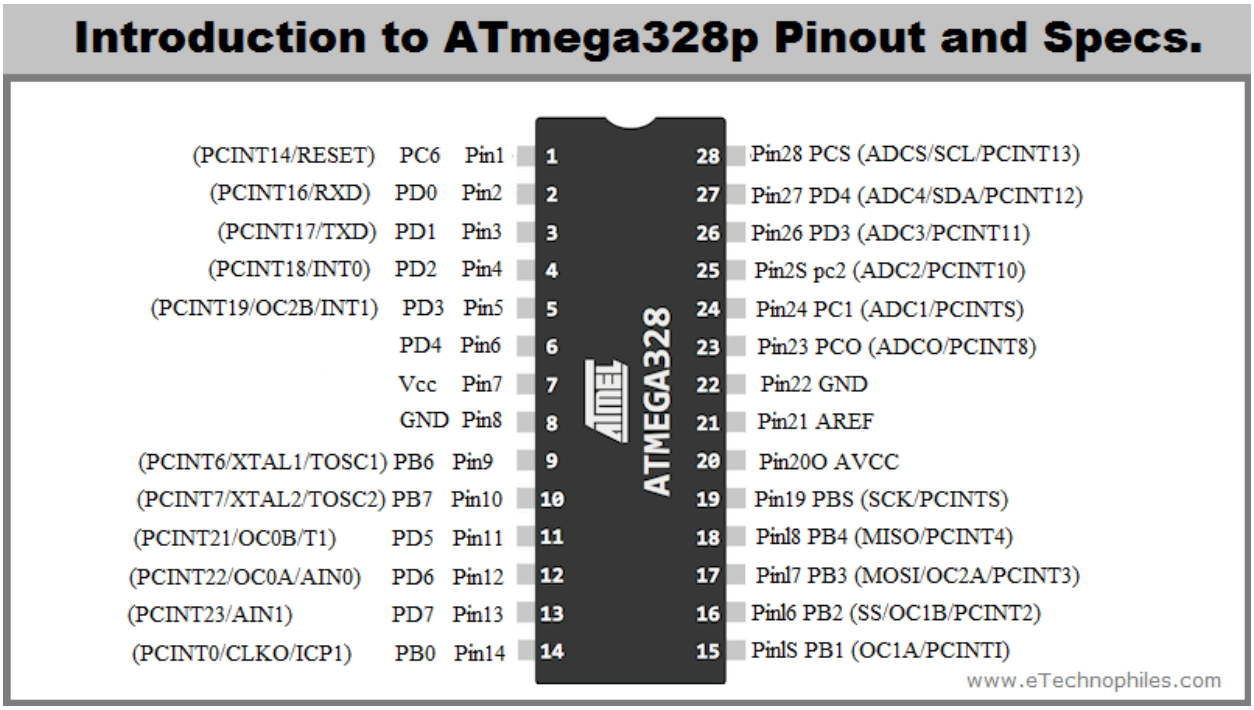
Block Diagram



Hardware Components & Schematics

Hardware I/O

Our project was very exhaustive and used all available I/O pins on the ATmega328. As shown in the above block diagram and schematics (see Appendix B), we primarily used the I2C pins PC5 and PC4 —SCL and SDA respectively— to communicate with all our sensors. We used I2C because of its ease in connecting multiple components using just two bus lines, leaving other pins for our other peripherals. The only exception was the GPS module MTK3339 which only functions using a serial UART interface. Pins PD0 and PD1 are the RX and TX pins respectively and were exclusively used by the GPS.



To give an overview of the specific I/O peripherals and their pins usages:
Inputs:

- Interface Buttons – 4 input pins with pull-up resistors enabled
- Oscillator clock - 1 input pin

Outputs:

- Parallel LCD – 7 output pins (4 data bits, 3 control bits)
- Morse Code LEDs – 3 output pins
- Buzzer – 1 output pin
- Solenoid - 1 output pin

Main components:

- 20x4 characters, parallel, RGB – Adafruit 499
- GPS module (MTK3339) – Adafruit 746
- 3-axis accelerometer (LIS3DH) – Adafruit 2809
- Temperature/altitude sensor (MPL3115A2) – Adafruit 1893
- Light sensor (TSL2591) – Adafruit 1980
- ESP32-C3-Mini-1 – B0D4QD19BB

Component Interactions

The components connected via I2C are the accelerometer, the light sensor, the altimeter, and the ESP32-C3. All of these components await commands from the master microcontroller and then respond with the data saved in their registers. The outlier is the ESP32-C3 which had hardware issues during installation and was not able to acknowledge any of these commands, so its communication was not functional at the time of the prototype demonstration.

The GPS module is special by using a UART serial interface to communicate with the microcontroller. The module is constantly transmitting NMEA sentences to its TX pin with lots of different data points you can selectively parse into readable data. The microcontroller simply reads its RX pin when needed then parses two of the NMEA sentences to get current latitude and longitude.

The LCD is also special by solely using GPIO pins to be controlled. The four data pins are used for the writing of characters onto the screen and directing the position of the cursor. The three control pins are for setting the LCD to Read or Write, interpret the data bits as data to write on screen or commands for the LCD, and for enabling the chip — which is constantly driven high.

Power

The *SCavenger* is almost entirely made up of 5 V components, with the exception of the ESP32-C3 which uses 3.3 V. Given the portable nature of the *SCavenger*, we found batteries to be the better choice of power supply. Specifically, the project uses 4 AA batteries as these are easily accessible and don't require an outlet to recharge. The user can have replacement batteries readily available.

The power supply is made up of a battery holder case that connects the batteries in series, resulting in 6 V output as each battery provides 1.5 V. To bring this to safe digital logic levels, we incorporated a buck converter onto our board that steadily reduces the voltage to 5 V for the power lines.

Regarding the ESP32-C3, a separate buck converter was used to step-down the supply voltage to 3.3 V. This 3.3 V is also used in a bi-directional level-shifter to convert I2C signals coming from the ATmega to safer 3.3 V inputs for the ESP. The ESP I2C signals then also get shifted back to 5 V for the ATmega.

Throughout the board, to maintain **engineering standards**, we connected resistors to load components with low resistance to avoid dangerous currents for the components. We also have a few capacitors to flatten noise from the power supply. Maximum current ever input into the microcontroller was 25 mA to avoid pin damage. This safety standard is also the reason for using

an NPN Bipolar Junction Transistor when activating the solenoid to avoid it drawing 714 mA from the ATmega and use the power supply current instead. Another safety precaution we added was a diode set in parallel with the solenoid but with its cathode set toward the +5V supply to avoid voltage spikes when the solenoid is turned off and releases its magnetic field energy.

Software Architecture & Operation

Overview

The software architecture is designed to orchestrate a linear, puzzle-based interactive journal experience. It handles user input from buttons, controls multiple sensors, manages puzzle logic, and renders story content on the LCD screen. The system is implemented in C for an AVR microcontroller and uses interrupt-driven input, sensor polling, several state machines, and persistent memory to control gameplay progression.

Main Loop and Execution Flow

The core of the software architecture is an event-driven loop that executes approximately every 25 milliseconds. The main loop is responsible for handling button presses, monitoring sensors, updating the LCD display, and evaluating puzzle completion criteria. All peripheral initialization occurs prior to entering this loop, including setup for I2C communication, LCD display, buzzer, button interrupts, LEDs, sensors (light, GPS, accelerometer, altimeter), and the solenoid.

Input from the user is handled using pin change interrupts on four buttons: Power, Clue, Back, and Next. Each interrupt sets a corresponding volatile flag (*power_button_pressed*, for example), which is then debounced and processed in the main loop. Button presses control the power state of the system, navigate dialogue screens, request clues, or confirm puzzle progression depending on the current mode.

The game operates as a finite state machine, alternating between two primary modes: DIALOGUE and PUZZLE. In dialogue mode, narrative text is rendered to the LCD one character at a time using a stateful say_step() function, simulating a typewriter effect. In puzzle mode, sensor data is continuously polled and evaluated against each puzzle's unique conditions. Once a puzzle is solved, a victory sound is played, game state is advanced, and the next dialogue sequence begins.

Persistent game progress is stored in EEPROM and loaded on startup, ensuring that players can resume from their last completed puzzle. Additionally, the LCD display is only refreshed when a *display_dirty* flag is set, minimizing unnecessary writes and improving visual performance.

Puzzle logic is tightly integrated into the loop, with each puzzle occupying a distinct *if* block that checks relevant conditions. For example, Puzzle 1 monitors light level transitions, Puzzle 2 transmits Morse code to LEDs and checks temperature, and Puzzle 5 compares live GPS coordinates to a predefined target. The loop structure provides consistent, low-latency control while remaining simple, readable, and modular.

To ensure smooth operation across multiple input/output devices, time-sensitive events such as solenoid activation, Morse flashing, and step counter decay are tied to the consistent loop interval, avoiding reliance on additional timers.

State Management

The system operates as a finite state machine, tracking game progression and user interaction through a global *GameState* structure (see Appendix A2). This structure contains several key fields:

- *puzzle_index*: Tracks the current puzzle, ranging from 0 to 5.
- *mode*: Indicates whether the system is in DIALOGUE or PUZZLE mode.
- *dialogue_index*: Tracks the current line of dialogue within a given puzzle.
- *current_screen*: Determines what content should be displayed on the LCD (prompt, clue menu, specific clue, etc.).
- *clue_progress*: Tracks how many clues have been unlocked for the current puzzle.
- *clue_menu_open*: Indicates whether the clue menu is currently active.
- *puzzle_complete*: Flags whether the current puzzle has been solved.
- *power_on*: Tracks the system's power state.

These fields together define the active "scene" of the game and determine how input events and sensor readings are interpreted. All transitions between puzzles, screens, and modes are driven by this centralized state.

Input System

User interaction is driven by four physical buttons: Power, Clue, Back, and Next. Each button is connected to a GPIO pin configured with a pin change interrupt. When a button is pressed, its associated interrupt sets a volatile flag (e.g., *power_button_pressed*), which is then processed and debounced in the main loop.

This interrupt-driven approach ensures responsive input handling without busy-wait polling, allowing the microcontroller to react to button presses in real time within the constraints of a single-threaded execution model.

To account for development needs and player testing, a manual reset sequence was implemented. If the Power and Clue buttons are held simultaneously at any time, the system resets the EEPROM-stored puzzle index to zero, effectively starting the game from the beginning. This behavior provides a way to replay the full experience without completing the final puzzle.

Excluding this unique combination, the buttons behave according to the following state diagram.

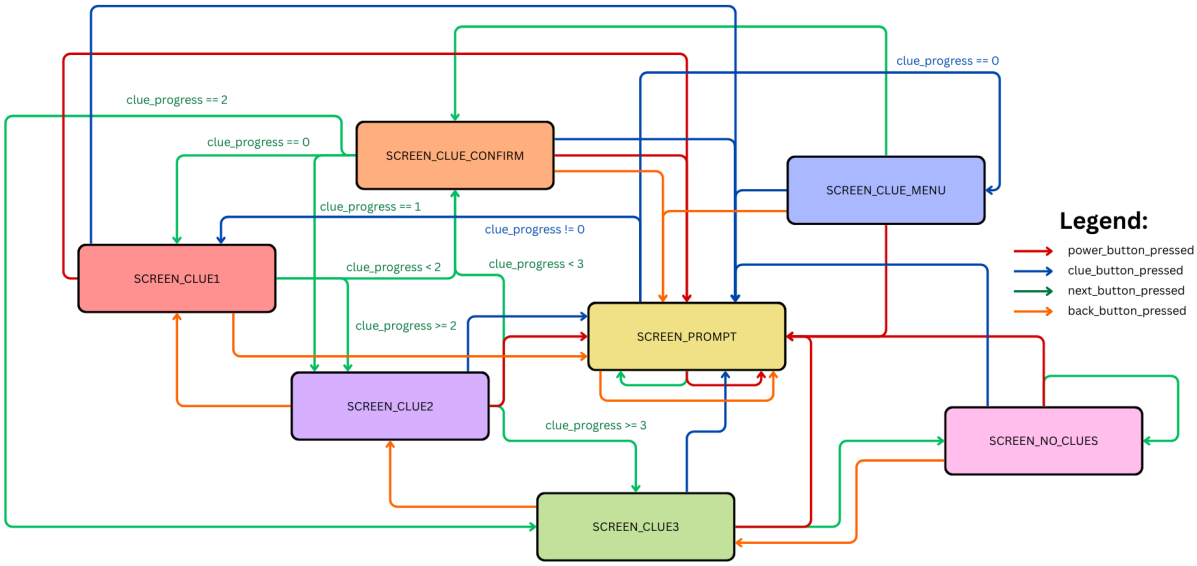


Figure 1. State machine diagram detailing the behavior of the Power, Clue, Next, and Back buttons. For state transitions with more than one condition, the extra conditions are labeled on the transition arrows themselves.

Display & Dialogue System

The user interface is rendered on a character-based LCD, which displays both story content and puzzle prompts. The display system is controlled through a *DialogueState* structure (see Appendix A3), which manages:

- *fullText*: A pointer to the current string being displayed.
- *charIndex*: The position of the next character to render.
- *row, col*: Cursor row and column.
- *finished*: A finished flag indicating whether the current line is complete.

In dialogue mode, story text is revealed one character at a time using the *say_step()* function, simulating a typewriter effect. This function wraps words across lines, skips redundant spaces, and plays soft “voice clicks” for each character using the buzzer for added immersion.

Text content is stored in flash memory (PROGMEM) and loaded into RAM when needed using helper functions like *init_say_from_progmem()* and *copy_progmem_string()*. This approach is necessary to allow large narrative scripts while avoiding overflow of the ATmega328p's available RAM.

In puzzle mode, the display is used to show prompts, clues, and puzzle-specific visualizations (e.g., a live dot counter for the step puzzle or GPS coordinates). The display is only updated when a *display_dirty* flag is set, reducing flicker and avoiding unnecessary writes.

Puzzle Logic & Sensor Integration

Each puzzle in the system is associated with a unique real-world interaction, monitored using one or more onboard sensors. Puzzle logic is evaluated continuously in the main loop, with each puzzle occupying a distinct conditional block gated by *puzzle_index*, *game.mode*, and *game.current_screen*. Once the completion criteria are met, a victory sound is played, the current puzzle is marked as complete, and the system transitions to the next dialogue sequence.

The following summarizes the logic and sensor integration for each puzzle:

Puzzle 1: Light Sensor Trigger

This introductory puzzle teaches the player about environmental interaction. Using a TSL2591 digital light sensor, the system records an initial brightness level and waits for the player to cross a threshold by turning a light on or off. A change from above to below (or vice versa) the *BRIGHT_THRESHOLD* triggers success.

The TSL2591 digital light sensor communicates over I²C and reports the measured intensity of full-spectrum and infrared light in two channels. The raw channel data acquired from the TSL2591 was converted to a real-world lux measurement using the module's reported integration time, analog gain, and lux formula denominator.

Puzzle 2: Morse Code and Temperature

Three LEDs flash a Morse code message ("FREEZE ME") using a timed update loop. Meanwhile, the MPL3115A2 barometric sensor is used to monitor ambient temperature. The puzzle is solved when the temperature rises above a defined threshold (e.g., by applying body heat), reinforcing the message's meaning.

The MPL3115A2 barometric sensor communicates over I^2C and, by specifying a particular read location, reports the ambient temperature in degrees Celsius using a 12-bit signed format.

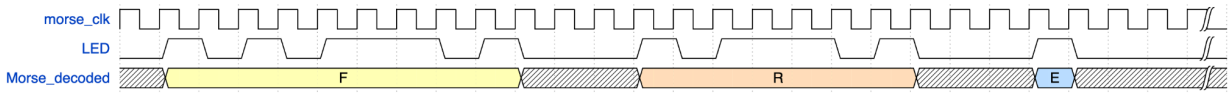


Figure 2. Timing diagram illustrating the first three letters of the Morse code message: ‘F’, ‘R’, and ‘E’. The morse_clk signal, shown for reference, represents a 250 ms timing unit derived from the system’s 25 ms main loop. Each Morse element (dot, dash, and spacing) aligns with this timing interval for clarity.

Puzzle 3: Altitude Challenge

To complete this puzzle, the player must increase their altitude. The MPL3115A2 sensor measures barometric pressure to estimate elevation. The system records a baseline and continuously checks for a gain of at least *ALTITUDE_THRESHOLD* meters. This encourages physical movement such as walking up stairs or a hill. As the player moves, their corresponding gain is printed on the LCD.

The MPL3115A2 reports altitude in meters, using a 20-bit signed format.

Puzzle 4: Step Tracker

The LIS3DH accelerometer detects steps by analyzing spikes in vertical (Z-axis) acceleration. A simple step detection function increments a counter, which decays over time to encourage the player to keep moving. A dot-based visualization on the LCD provides feedback as the player approaches the *STEP_GOAL*. When the screen becomes filled with dots, the puzzle is complete.

Puzzle 5: GPS Navigation

This puzzle prompts the player to travel to a new location. After instructing the user to go outside, the device uses the Adafruit Ultimate GPS module to acquire an initial fix. It then generates a target location based on that first reading and continuously monitors the player’s live coordinates. Once the player comes within a small threshold of the target, the puzzle is marked as complete.

The Adafruit Ultimate GPS module communicates with the microcontroller via UART and continuously streams NMEA sentences. To extract the user's coordinates, the system parses two specific sentence types: $GPRMC and $GPGGA, both of which consistently provided valid latitude and longitude data during field testing at the University of Southern California. Each sentence is parsed according to its standardized format to obtain precise geographic coordinates, which are then compared against the generated target location.

Puzzle 6: Secret Compartment Finale

Once the player reaches the final puzzle, the system activates a solenoid to unlock a physical compartment. The solenoid is driven using a 2N2222 NPN transistor, which acts as an electrical

switch to safely control the component. It remains energized for two seconds before turning off, after which the system powers down, signaling the end of the narrative journey.

Persistence

To preserve user progress in the case of power loss, the system uses the AVR microcontroller's built-in EEPROM. The current puzzle index is stored at EEPROM address 100 and is read during system initialization. This allows players to resume their journey without redoing previously completed puzzles.

When a puzzle is solved, the updated puzzle index is written back to EEPROM using *eeprom_update_byte()*, which avoids unnecessary writes by only updating if the value has changed. On startup, the game reads the stored value using *eeprom_read_byte()* and validates it against the total number of puzzles to guard against corruption or invalid memory values.

EEPROM persistence ensures a more user-friendly experience and removes the need for external storage. Additionally, only a single progress variable, *game.puzzle_index*, must be stored, minimizing any concern of EEPROM memory overflow.

Testing, Debugging, & Validation

A major component of *The SCavenger* involved integrating multiple I2C-based sensors, including the TSL2591 light sensor, MPL3115A2 altimeter, and LIS3DH accelerometer. Each presented unique debugging challenges.

- For the TSL2591, we ran into the common issue of our sensor responding to I2C read commands, but failing to acknowledge writes. Eventually, we discovered this was due to incorrect initialization sequence and overlooked delays in the setup process.
- The MPL3115A2 altimeter initially returned erratic altitude values, often starting at negative numbers that increasingly grew smaller despite rising above the surface. We noticed that we needed to wait for a proper stabilization period after power-up, and ensure digits were output correctly onto the LCD screen in C.
- LIS3DH accelerometer needed to reliably detect user steps, and had to pay careful attention as to how we expected the device to be held (with the accelerometer lying flat on the board), and whether we should configure movement to be in the x,y, or z direction, or simply, account for all.
- ESP32-C3 would send back an ACK signal to any commands with empty buffers in testing. For an as-of-yet undetermined reason, when data was inserted into the write buffers, the ESP would no longer even send an ACK back, let alone actually process the data. Given this issue and the lack of more time to address it, the prototype uses a replacement puzzle that doesn't use the ESP at all.

There were also some mechanical and memory capacity situations as the ATmega would keep disconnecting from its socket, and the RAM was overloaded during the GPS puzzle. Therefore, Flash memory was used to maintain the loads of dialogue being used by the LCD screen.

Engineering standards were maintained for this aspect of development, with each hardware component being developed and tested in isolation before being integrated into the full system. This modular approach enabled incremental debugging and reduced unintended cross-subsystem interface. We also designed our game logic to handle sensor errors gracefully, utilizing LCD feedback to aid in debugging and usability.

Adaptive Design

As development progressed, certain hardware components and ideas were phased out in favor of simpler or more reliable solutions. For example, the DS3231 RTC module was initially planned for a time-based challenge. However, we determined the RTC was unnecessary and removed it from the design when realizing that the ATmega came with its own timing module.

Another chip replacement we did was from the ESP8266 that was originally proposed and worked on with the ESP32-C3. The reason was that the first toolchain we found and used ended up being outdated and so flashing was an issue. Then we found another toolchain which was slightly more updated yet the flashing process was incredibly difficult and confusing. For these reasons we opted for a much newer ESP model that even had I2C functionality — instead of UART which would've needed a mux for our ATmega to drive alongside the GPS. The ESP32 had up-to-date support and a toolchain that facilitated flashing, thus making the software a non-issue when developing.

Cost Analysis

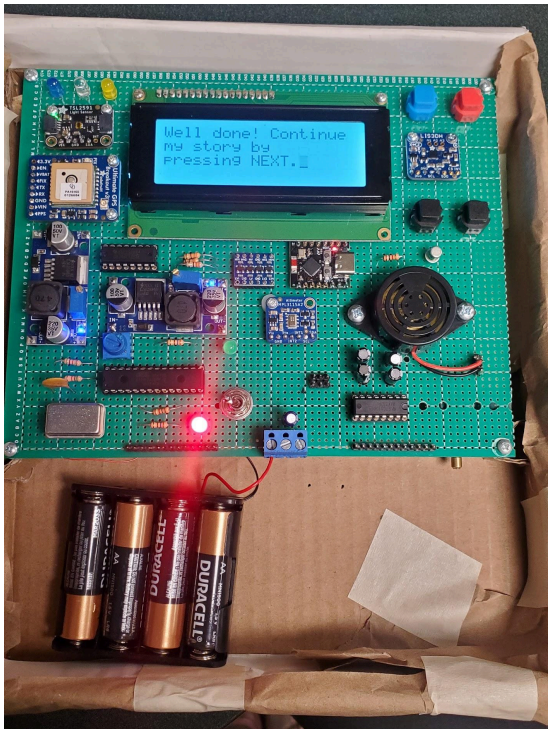
The cost analysis is divided into two sections. First, the bulk as-built production cost is calculated based on the current prototype. Then, a more realistic production cost is estimated, accounting for design changes that would occur in actual manufacturing. In both estimates, prices assume bulk purchasing at 1,000-unit scale from vendors such as Digi-Key and Amazon.

Several design choices would be modified if *The SCavenger* were produced in large quantities. Firstly, the protoboard would be replaced with a printed circuit board. A custom PCB layout consolidates many discrete components, improves testability, and reduces the device's footprint, which could reduce assembly time and error rate. Additionally, many of the sensors would be exchanged for more compact and reliable alternatives. The table below reflects these changes and gives a more accurate estimate of the true production cost.

Notably, this estimate excludes non-recurring costs such as PCB layout and firmware development, which are assumed to be amortized over full production. It also does not include the cost of labor for assembly, flashing, and testing.

The complete cost analysis can be found in Appendices C1 and C2. In summary, the total cost per unit is approximately **$98.81** for the current prototype and **$46.92** for a theoretical production-ready version, based on bulk component pricing and design optimizations.

Final Assembly



Appendix

Appendix A1: Code (ESP32-C3 Firmware)

The ESP32 had issues communicating with our microcontroller through I2C due to hardware issues. Despite this, we did develop the code that would be used in the final product assuming the hardware issue is solved.

```
// --- App Main ---
void app_main(void)
{
    ESP_ERROR_CHECK(nvs_flash_init());
    esp_netif_init();
    esp_event_loop_create_default();
    wifi_event_group = xEventGroupCreate();

    esp_netif_create_default_wifi_ap();
    esp_netif_create_default_wifi_sta();

    wifi_init_config_t cfg = WIFI_INIT_CONFIG_DEFAULT();
    esp_wifi_init(&cfg);

    esp_event_handler_register(WIFI_EVENT, ESP_EVENT_ANY_ID, &wifi_event_handler, NULL);
    esp_event_handler_register(IP_EVENT, IP_EVENT_STA_GOT_IP, &wifi_event_handler, NULL);

    start_wifi_ap();
    server = start_webserver();
}
```

Figure A1.1. Main logic

The above code is the programming that the ESP runs on startup. It initializes its needed general network interface then creates two separate interfaces to go into Access Point mode or Station mode. It also creates handlers that will deal with wifi events like a user connecting to the Access Point or the ESP connecting to the user's mobile hotspot.

By default, it powers up to Access Point mode and starts the web server the user can connect to and give their mobile hotspot credentials.

```c
// --- Overpass API Query ---
static void query_overpass_api(double lat, double lon)
{
    const char *url = "https://overpass-api.de/api/interpreter";

    char post_data[512];
    snprintf(post_data, sizeof(post_data),
            "[out:json];node(around:2000,%.6f,%.6f)[\"amenity\"=\"cafe\"];out;",
        lat, lon);

    esp_http_client_config_t config = {
        .url = url,
        .method = HTTP_METHOD_POST,
        .cert_pem = (const char *)_binary_isrg_root_x1_pem_start,
        .timeout_ms = 10000,
        .buffer_size = 8192,
        .buffer_size_tx = 8192,
    };
```

Figure A1.2. API Interaction

For API calls, the project uses Overpass API to search for nearby locations tagged with the 'cafe' amenity tag. The code above shows the url of the API and how the query was created to be sent. You can also note the HTTP Client configuration to properly handle sending the query and receiving the server response.

Appendix A2: Code (GameState)

```c
typedef struct {
    uint8_t puzzle_index;
    SystemMode mode;
    uint8_t clue_progress;
    PuzzleScreen current_screen;
    uint8_t dialogue_index;
    uint8_t power_on;
    uint8_t puzzle_complete;
    float initial_light;
    uint8_t clue_menu_open;
    int32_t base_altitude;
    float target_latitude;
    float target_longitude;
} GameState;

extern GameState game;
```

Figure A2. The GameState structure tracks player progression, the device's current mode, and other relevant state details.

Appendix A3: Code (DialogueState)

```
59  typedef struct {
60      const char* fullText;
61      int charIndex;
62      int row;
63      int col;
64      uint8_t finished;
65  } DialogueState;
```

Figure A3. The DialogueState structure facilitates and tracks printing of a long dialogue string.

Appendix B: Schematics

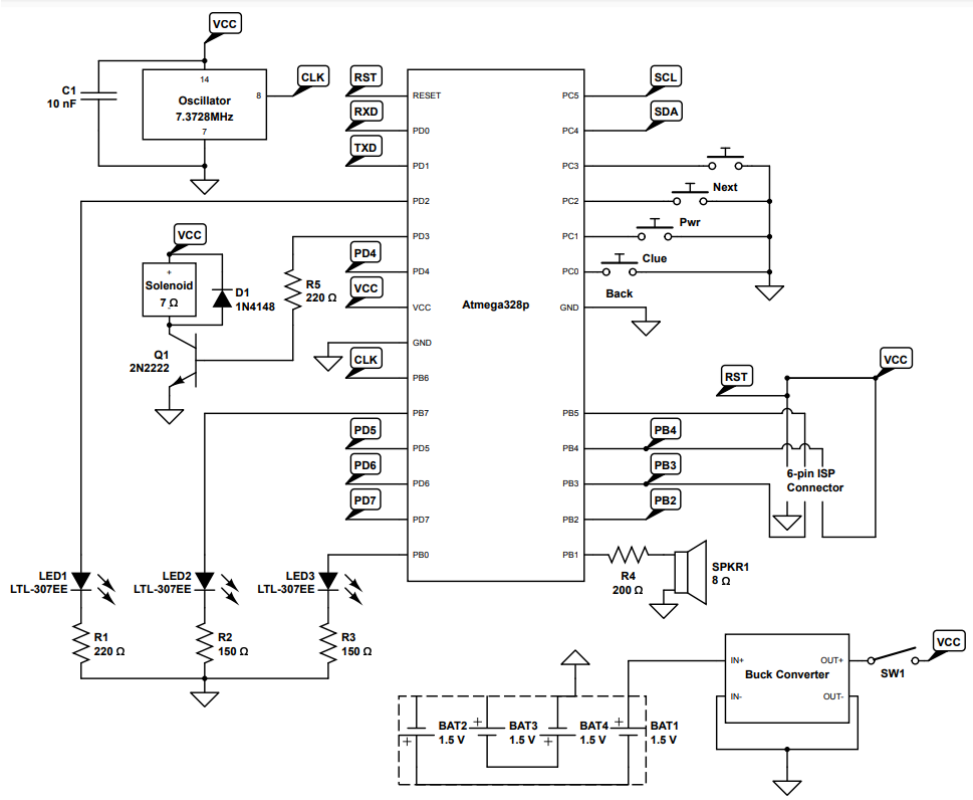


Figure B1. Microcontroller and GPIO

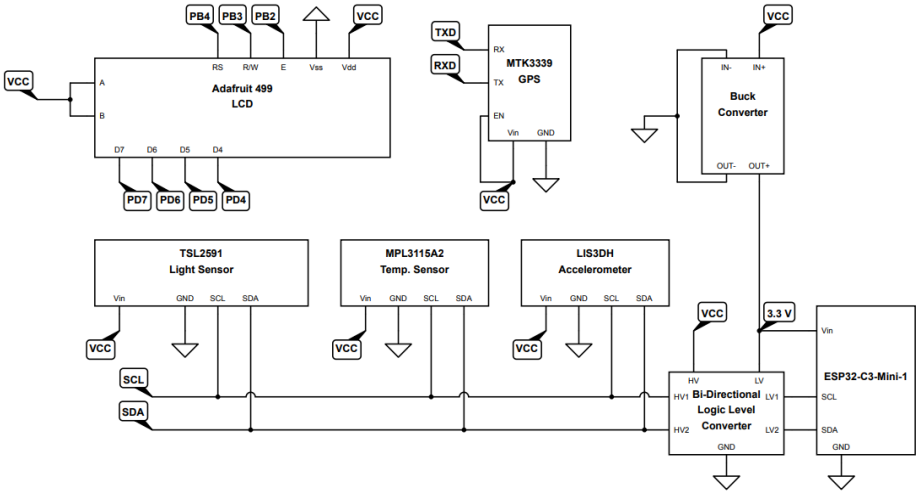


Figure B2: Major Peripherals

Appendix C1: Parts List & Cost Details (Prototype)

Part Name	Quantity	Cost ($)
ATmega328p Microcontroller	1	2.19[3]
ECS-100AX-072 Oscillator	1	1.79[4]
MD42004A6W-FPT RGB LCD	1	12.33[5]
Piezo Buzzer	1	2.00[6]
LED	3	0.33[7]
TSL2591 Light Sensor	1	6.95[8]
MPL3115A2 Altitude/Temperature Sensor	1	9.95[9]
LIS3DH Accelerometer	1	4.95[10]
Adafruit Ultimate GPS Breakout v3 Module	1	29.95[11]
Solenoid	1	6.43[12]
2N2222A Transistor	1	2.69[13]
Pushbutton	4	2.01[14]
Battery Holder	1	0.97[15]
DC-DC Step Down Buck Converter	1	1.00[16]
Resistor	8	0.08[17]

[3] Price from Digi-Key, accessed May 4, 2025.
[4] Price from Digi-Key, accessed May 4, 2025.
[5] Price from Digi-Key, accessed May 4, 2025.
[6] Price from Amazon, accessed May 4, 2025.
[7] Price from Digi-Key, accessed May 4, 2025.
[8] Price from Digi-Key, accessed May 4, 2025.
[9] Price from Digi-Key, accessed May 4, 2025.
[10] Price from Digi-Key, accessed May 4, 2025.
[11] Price from Digi-Key, accessed May 4, 2025.
[12] Price from Digi-Key, accessed May 4, 2025.
[13] Price from Digi-Key, accessed May 4, 2025.
[14] Price from Digi-Key, accessed May 4, 2025.
[15] Price from Digi-Key, accessed May 4, 2025.
[16] Price from Amazon, accessed May 4, 2025.
[17] Price from Digi-Key, accessed May 4, 2025.

10µF Electrolytic Capacitor	1	0.08[18]
0.01µF Ceramic Capacitor	1	0.12[19]
Protoboard	1	6.95[20]
IC Socket	1	0.20[21]
Screws, nuts, washers	14	7.84[22]
Total Cost ($)	**98.81**	

Table C1. Parts list and cost analysis for current prototype.

Appendix C2: Parts List & Cost Details (Realistic Product)

Part Name	**Quantity**	**Cost ($)**
ATmega328p Microcontroller	1	2.19
ECS-100AX-072 Oscillator	1	1.79
MD42004A6W-FPTLRGB LCD	1	12.33
Piezo Buzzer	1	2.00
LED	3	0.33
LTR-329ALS-01 Optical Sensor	1	0.42[23]
MPL3115A2ST1 Barometric Sensor	1	2.82[24]
MXC4005XC Accelerometer	1	0.61[25]
TESEO-LIV3R RF Receiver (GPS)	1	8.42[26]
Solenoid	1	6.43
2N2222A Transistor	1	2.69

[18] Price from Digi-Key, accessed May 4, 2025.
[19] Price from Digi-Key, accessed May 4, 2025.
[20] Price from ProtoSupplies, accessed May 4, 2025.
[21] Price from Digi-Key, accessed May 4, 2025.
[22] Price from Digi-Key, accessed May 4, 2025.
[23] Price from Digi-Key, accessed May 7, 2025.
[24] Price from Digi-Key, accessed May 7, 2025.
[25] Price from Digi-Key, accessed May 7, 2025.
[26] Price from Digi-Key, accessed May 7, 2025.

Pushbutton	4	2.01
Battery Holder	1	0.97
UC3526ADWTR Buck	1	1.63[27]
Resistor	8	0.08
10µF Electrolytic Capacitor	1	0.08
0.01µF Ceramic Capacitor	1	0.12
Custom PCB	1	2.00[28]
Total Cost ($)	**46.92**	

Table C2. Parts list and cost analysis for realistic products.

[27] Price from Digi-Key, accessed May 7, 2025.
[28] Price from AllPCB, accessed May 7, 2025.

Division of Work

Task	Jeremy	Karla	Allan
System design	20%	60%	20%
Component selection	40%	20%	40%
Hardware design	50%	25%	25%
Software design	60%	20%	20%
Documentation	20%	30%	50%
Project report (oral)	25%	40%	35%
Project report (written)	25%	35%	40%

By signing my name below, I agree to the division of work as listed above.

Jeremy Pogue 	Karla Vasquez 
Allan Anderson