Software Formalization

Year: 2024 **Semester: Spring** **Team: 1** **Project: Dungeon Crawler Board**
Creation Date: February 9th, 2024 **Last Modified:** February 23rd, 2024
Author: Grace Whitaker **Email:** gwhitake@purdue.edu

Assignment Evaluation: See Rubric on Brightspace Assignment

1.0 Utilization of Third Party Software

For this project, our team will be using third party software to develop the computer application and to initialize our hardware in order to save more time for writing the hardware logic. The main hardware logic and the gameplay will be written by the team, except for the LCD and Keypad, whose logic will be borrowed from previous projects and labs from ECE362. Any libraries that are used are included within the softwares listed below. Information about third party software can be found in the following table.

Name	Description	Use	Licensing Info
Unity Personal	"Unity is the world's leading real-time 3D development platform, offering what you need to create and grow amazing games and experiences across a wide range of platforms."[1]	We will use Unity to build our computer application that will send map and character info to the microcontroller.	Unity License
STM32 CubeIDE	"STM32CubeIDE is an advanced C/C++ development platform with peripheral configuration, code generation, code compilation, and debug features for STM32 microcontrollers and microprocessors." [2]	CubeIDE will be used to build our code and send it to the microcontroller. CubeIDE will generate initialization code for our different hardware modules.	SLA0048 STM32 License, Requires st.com account

2D Hex Sprites - Hexagonal Tile Setup	This is a Unity package developed by RRFreelance that provides an easy method to set up hex maps along with editable hex sprites for personal use. [3]	We will use the package's hex image sprites to visually represent the hex map in the Unity application.	Standard Unity Asset Store EULA
ECE362 LCD Code	This is code from Purdue's ECE362 Lab that provides functions for interfacing with an LCD [4]. It was developed based on examples from lcdwiki.com [5].	We will reference the function calls and variables when developing our LCD code.	Open source. No licensing.
ECE362 Keypad Code	This is code from Purdue's ECE362 Lab that provides functions for interfacing with a 4x4 Matrix Keypad [4]. The main.c file contains code for scanning keypad rows through GPIO pins.	We will reference the functions used for scanning keypad rows and columns when developing the functionality for our own keypad.	N/A

2.0 Description of Software Components

Our project has four main software and firmware components. These include the general firmware input/output for displaying and storing game information, the firmware for importing maps and long-term storage, the game logic for running and tracking game data and the game state during gameplay, and a computer application for designing and exporting map and character information. The next section provides an overview of each of these components. The details of these components can be found in their software component diagrams at the end of this document.

2.1 Overview

The general input/output logic involves standard input and output between LEDs, Hall Effect sensors, I2C I/O expanders, an LCD, a Keypad. The LCD and Keypad will be used throughout game initialization and gameplay. The code will display instructions on the LCD and scan the keypad for key press information. The code will output different screens and instructions to the LCD based on the key presses and game state. This key press information will also be used to set variables in the gameplay logic before the game starts. The Hall Effects' output will be used to track and update character positioning in the game logic, and the LED code will set their output colors and brightness based on map hex types, character positions, and field of view. In addition, this LED code will need to select through a set of 16 I2C I/O expanders to select the correct LEDs. The initialization code for these modules will be generated through CubeIDE, but the rest of the logic will be written by the team. Some of these modules also have libraries that we will utilize for development, such as CubeIDE's HAL (Hardware Abstraction Layer) library.

The importing and long-term storage logic involves using a USB module and microSD card module to receive a text file from a connected computer, parse it, and store the information on the microSD card inside the game board. This will allow for a large amount of file space for maps and characters so users can keep their designs for a long time. The USB code will use UART to communicate and receive files from the computer and the code will parse the text file into defined character and map data and save it on the microcontroller. The microSD code will use SPI to receive this data and write it into new files on the microSD. The SPI will also be used to send these files to the microcontroller when needed during the game setup. The initialization for these modules will be generated with CubeIDE, and some libraries will be used for setup.

The gameplay logic will run concurrently with the hardware I/O logic during gameplay. This logic will initialize characters, a game map, and turn order based on information input by the users before gameplay. The game is then run in a series of character turns that continues until either all players are dead or all enemies are dead and chests looted. During each turn, character actions and movements are tracked based on readings from the Hardware I/O system. Movement distances are determined by an algorithm finding the shortest path between starting and ending hexes, and character positions are updated after movement. At the end of each turn, character positions and stats are updated and the map is checked for living players, living enemies, and chests to determine if the game should end or continue. Basic character information and stats will be stored as structs. There will be Character and Hexagon object classes to pass information around about character stats and positioning to determine movement and field of vision to be displayed by a Map object class. This code is entirely written by our team in C++.

The computer application will provide players a way to visually create dungeon maps and characters using a graphical user interface. There will be a character page and a map page. The character page will use graphical dropdowns and text boxes where the user can directly set variables associated with a character struct. The map page will be more complex. It will display on the left side a set of six different hex types, and on the right side a hex map with 256 hexes. Each hex on the map will be associated with a MapHex struct that will track the hex's coordinates and type. The user will be able to select a hex type on the left side of the screen and click the hex map to set these structs. The code underneath will keep track of the hex type selected and the data of the individual hexes. Once a map and characters are created, the user can click a button in the application to export the map and character information as a .txt file that will be read by the microcontroller. The info will be translated and written as strings of numbers for easy parsing. The computer application will be developed in C# using Unity, and the code will utilize Unity's default libraries.

2.2 Table of Components and Code Sources

Below is a table detailing the parts of the code that was written by our team (Written Code) and the parts that were used from outside sources (Outside Code). The sources for the listed outside code and libraries can be found in the table in section 1.0.

Component	**Written Code**	**Outside Code**	**Source/Libraries (for outside code)**
General I/O	LED interfacing, Hall Effect interfacing, I2C I/O Expander addressing, LCD interfacing, Text parsing	Initializations, Keypad scanning, LCD text display	CubeIDE (Initializations), ECE362 Lab (Keypad Scanning and LCD text display)
Importing, Storage	Reading from USB, Writing to microSD	Initializations	CubeIDE (Initializations)
Gameplay Logic	Character, map, and hex classes, Pathing algorithms, Turn logic and game state tracking, Character stat updates	N/A	N/A
Application	Converting hex structs for output to a text file, Classes and structs for map hex, characters, and game manager, Passing hex types between classes	GUI Interfacing, Interacting with Unity Components	Unity Engine (GUI and Interacting with Unity Components)

3.0 Testing Plan

All the components will need to be tested thoroughly to ensure correct setup and to minimize the risk of bugs occurring during final gameplay. The tests for these components will be listed below in order of importance (most important to least important).

3.1 General Input/Output

The general input/output is the most important as it is the primary mode of communication information to the user. Each piece of firmware will need to be tested individually before integrating all of them together. The following tests will be needed.

1. Adjust the brightness and color of individual LED lights.
2. Flash LED lights quickly, about 4 times a second.
3. Scan through Hall Effect sensors and set a debug light when a magnet is detected.
4. Print “Hello World” to LCD and clear it after 10 seconds
5. Set a debug light on when a keypad key is pressed, off when no keypad keys are pressed.
6. When a keypad key is pressed, print the number or letter of the key to the LCD.
7. Use the keypad to select a number, then light a debug light at the corresponding I2C I/O address output.
8. Use the keypad to select a row and column, then light the corresponding LED in our LED array.

3.2 Gameplay Logic

The gameplay logic is the second most important since it handles the character, position, and game state data during gameplay. We should first test gameplay in an IDE before testing it on the microcontroller to speed up the testing process.

1. Initialize and print variables of new objects derived from character and map classes.
2. Initialize a game map out of hexagons and print hex coordinates given a row and column.
3. Print the shortest path between two hexagons on the map.
4. Get a character from a hexagon and print their field of view and possible movements.
5. Get a character and print the name of a nearby character and the distance between.
6. Set two characters next to one another and check if they can attack. If they are on opposite teams, they can. If they are on the same team, they cannot.
7. Set a player character next to a set of chests and print the contents of any adjacent chests.
8. Have a character move less than their given movement points. Print the remaining amount of movement points and check if correct.
9. Print the turn order after all characters are added to a game.
10. Print the turn order after an enemy character is in the field of view of a player character.
11. During a character’s turn, print the character whose turn is next and verify with the order.
12. If all players are dead, print “Lose”. When all enemies are dead but chests not looted, print chests remaining. When all enemies are dead and chests looted, print “Win”.

3.3 Importing and Storage

The USB module and microSD module will need to be tested individually. The USB module will receive the text file from the computer, and the microSD module will write file information for long term storage on the microSD. The following tests will be needed:

1. Light a debug light when the USB module receives a file from a computer.
2. Write the text file contents into a buffer and check the buffer size after receiving the file and ensure that the buffer size matches the expected size of the contents.
3. Parse the text file after receiving from USB and check if it matches the original file.
4. Light a debug light when the microSD card is mounted successfully on the SPI.
5. Create a text file, write “hello world”, and save it to the microSD. Insert the microSD into a computer and check if the file exists.
6. Delete the previously created text file. Insert the microSD into a computer and check if the file is now gone.

3.4 Computer Application

The Unity application will handle the creation and export of the text file based on GUI input from the user. It needs to have its GUI interfacing and its text output tested. The following tests will be needed:

1. Print the hex type that is selected when the user clicks on a selection hex on the left side of the hex map page.
2. Print the hex type that is placed when the user clicks on a map hex on the right side of the hex map page.
3. Print the q and r coordinates of each hex on the map page after they are assigned by the game manager.
4. Write the q coordinate, r coordinate, and type of a single hex to a text file and check if the file is correct.
5. Write the coordinates and types of all hexes row by row to a text file and check if the file is correct.
6. When an attribute is selected on the character page, print the name of the character that is being altered.
7. When a character is completed, print the new attributes and name and verify if the changes are correct.
8. Print a character’s information to a text file and check if the file is correct.
9. Print all character and hex information to a text file and check if all information is accurate and written in the correct order and format for parsing.

4.0 Sources Cited:

1] “Main page,” LCD wiki, http://www.lcdwiki.com/Main_Page (accessed Feb. 17, 2024).

[2] “Fall 2023 ECE 362 lab experiments,” f2022-public-labs, https://ece362-purdue.github.io/f2022-public-labs/ (accessed Feb. 17, 2024).

[3] “2d hex sprites - hexagonal tile setup: 2d environments,” Unity Asset Store, https://assetstore.unity.com/packages/2d/environments/2d-hex-sprites-hexagonal-tile-setup-135185 (accessed Feb. 17, 2024).

[4] “STM32CubeIDE,” STMicroelectronics, https://www.st.com/en/development-tools/stm32cubeide.html#overview (accessed Feb. 17, 2024).

[5] U. Technologies, “Unity Personal,” Unity, https://unity.com/products/unity-personal (accessed Feb. 17, 2024).

Appendix 1: Software Component Diagram

Legend
Green: Complete
Yellow: In Progress
Red: Not Started

Figure 1: Program Overview

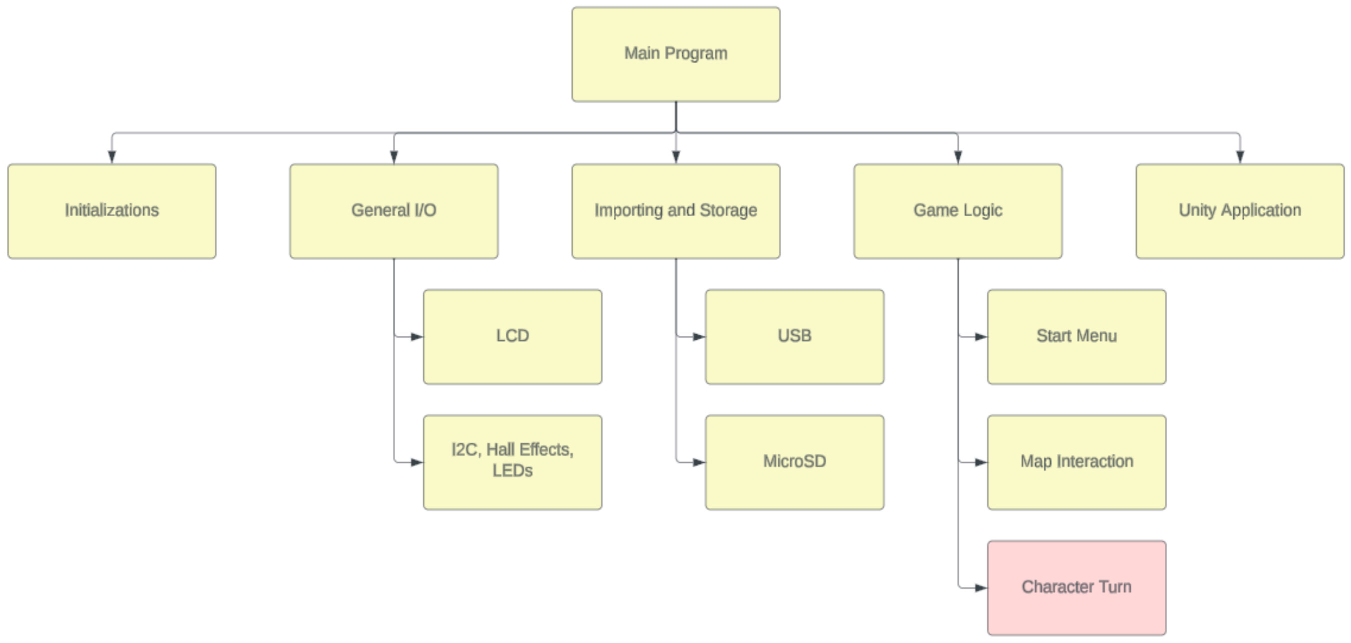


Figure 2: Initialization, LCD, USB, and MicroSD Firmware Structure

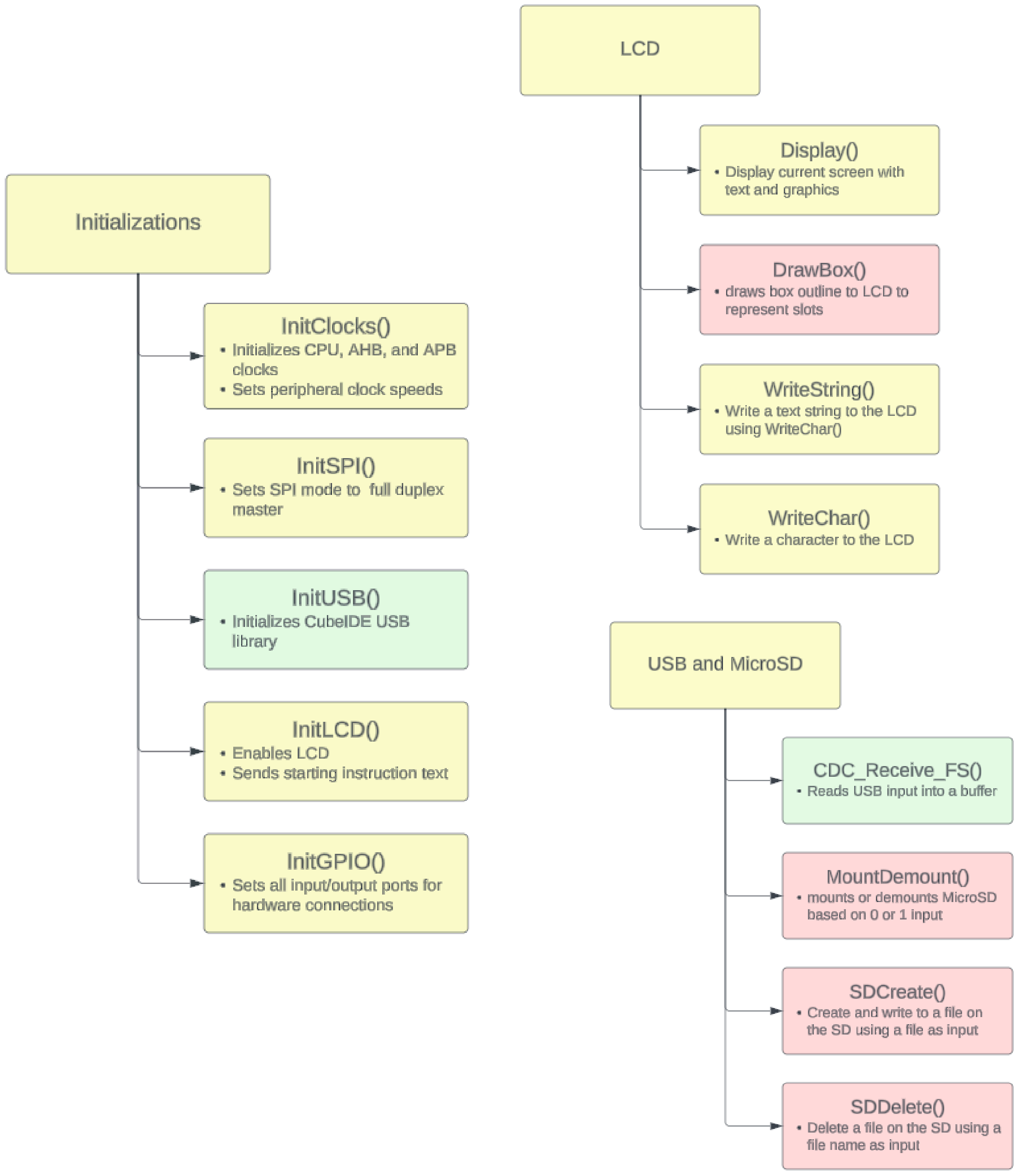
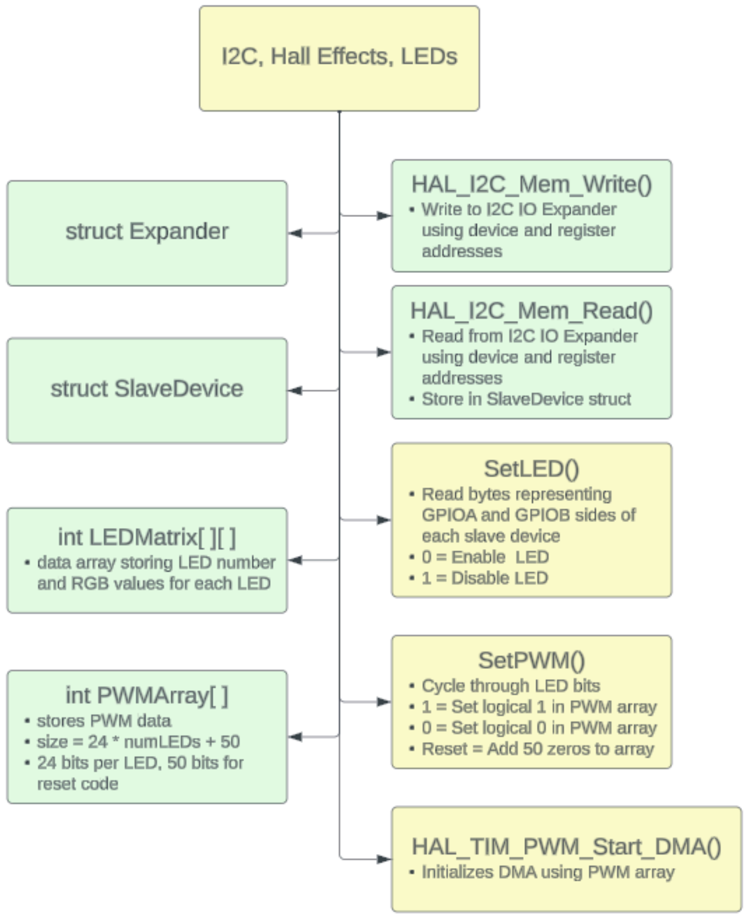
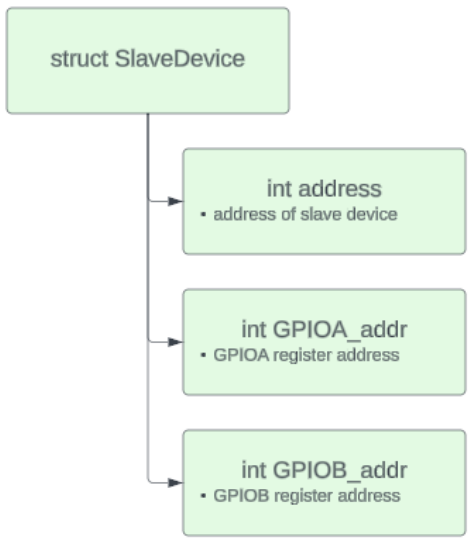

LCD
Display()
• Display current screen with text and graphics
DrawBox()
• draws box outline to LCD to represent slots
WriteString()
• Write a text string to the LCD using WriteChar()
WriteChar()
• Write a character to the LCD
Initializations
InitClocks()
• Initializes CPU, AHB, and APB clocks
• Sets peripheral clock speeds
InitSPI()
• Sets SPI mode to full duplex master
InitUSB()
• Initializes CubeIDE USB library
InitLCD()
• Enables LCD
• Sends starting instruction text
InitGPIO()
• Sets all input/output ports for hardware connections
USB and MicroSD
CDC_Receive_FS()
• Reads USB input into a buffer
MountDemount()
• mounts or demounts MicroSD based on 0 or 1 input
SDCreate()
• Create and write to a file on the SD using a file as input
SDDelete()
• Delete a file on the SD using a file name as input

Figure 3: I2C, Hall Effects, and LED Firmware Structure


I2C, Hall Effects, LEDs
struct Expander
struct SlaveDevice
int LEDMatrix[ ][ ]
• data array storing LED number and RGB values for each LED
int PWMArray[ ]
• stores PWM data
• size = 24 * numLEDs + 50
• 24 bits per LED, 50 bits for reset code
HAL_I2C_Mem_Write()
• Write to I2C IO Expander using device and register addresses
HAL_I2C_Mem_Read()
• Read from I2C IO Expander using device and register addresses
• Store in SlaveDevice struct
SetLED()
• Read bytes representing GPIOA and GPIOB sides of each slave device
• 0 = Enable LED
• 1 = Disable LED
SetPWM()
• Cycle through LED bits
• 1 = Set logical 1 in PWM array
• 0 = Set logical 0 in PWM array
• Reset = Add 50 zeros to array
HAL_TIM_PWM_Start_DMA()
• Initializes DMA using PWM array



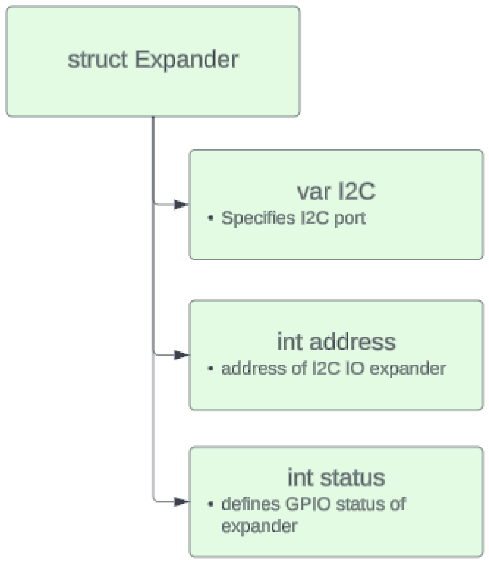


Figure 4: Important Character Structs

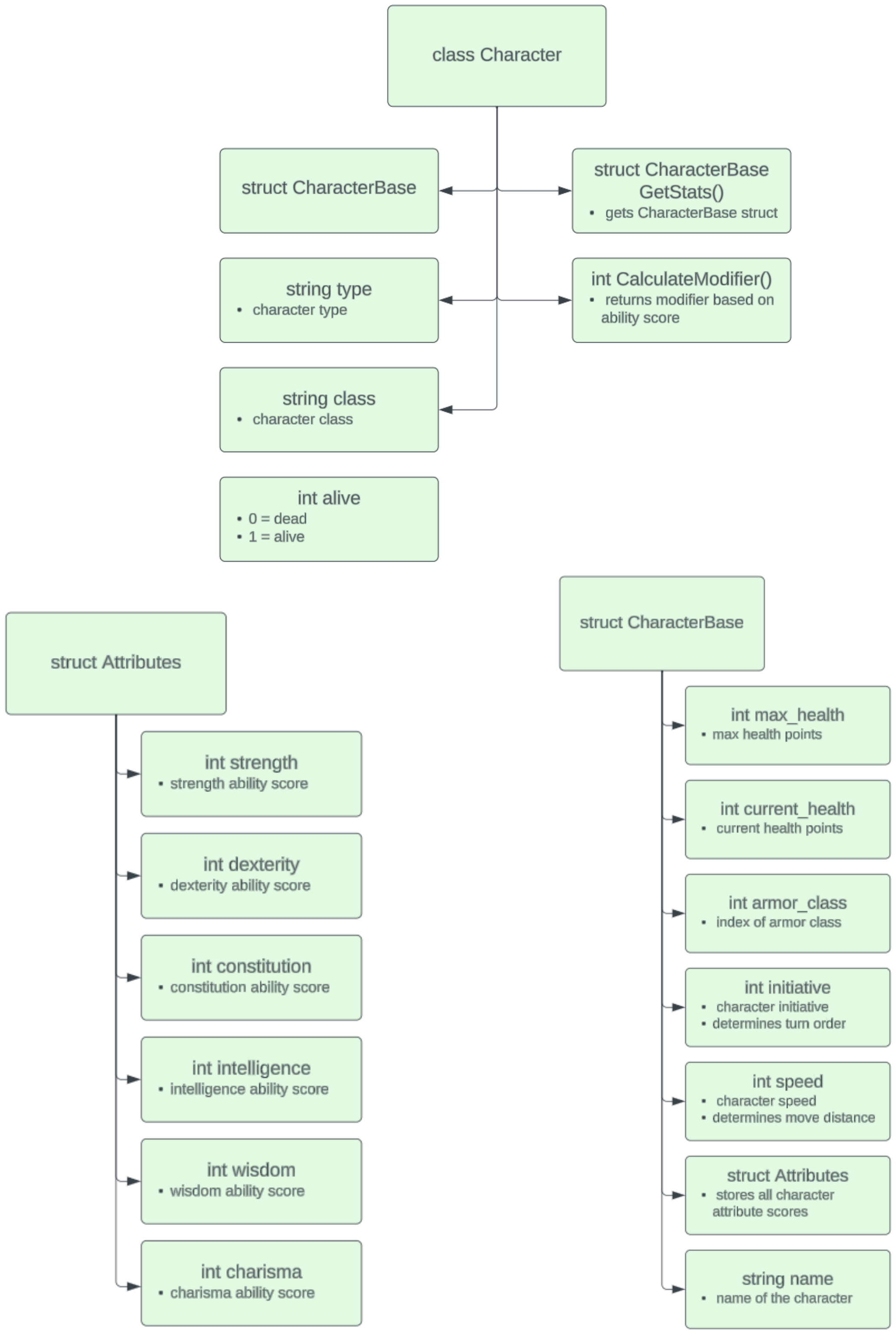


Figure 5: Important Game and Map Structs

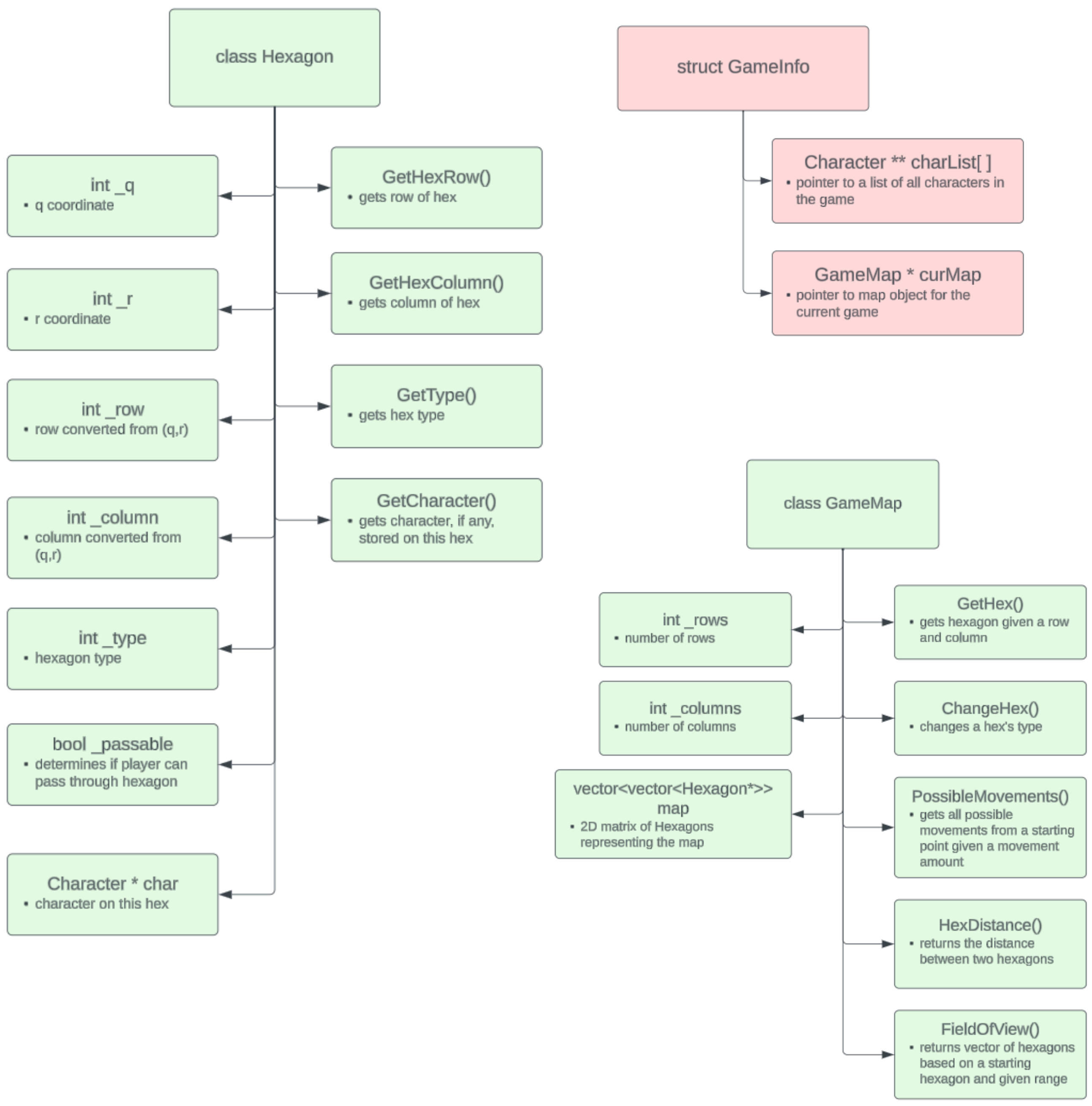
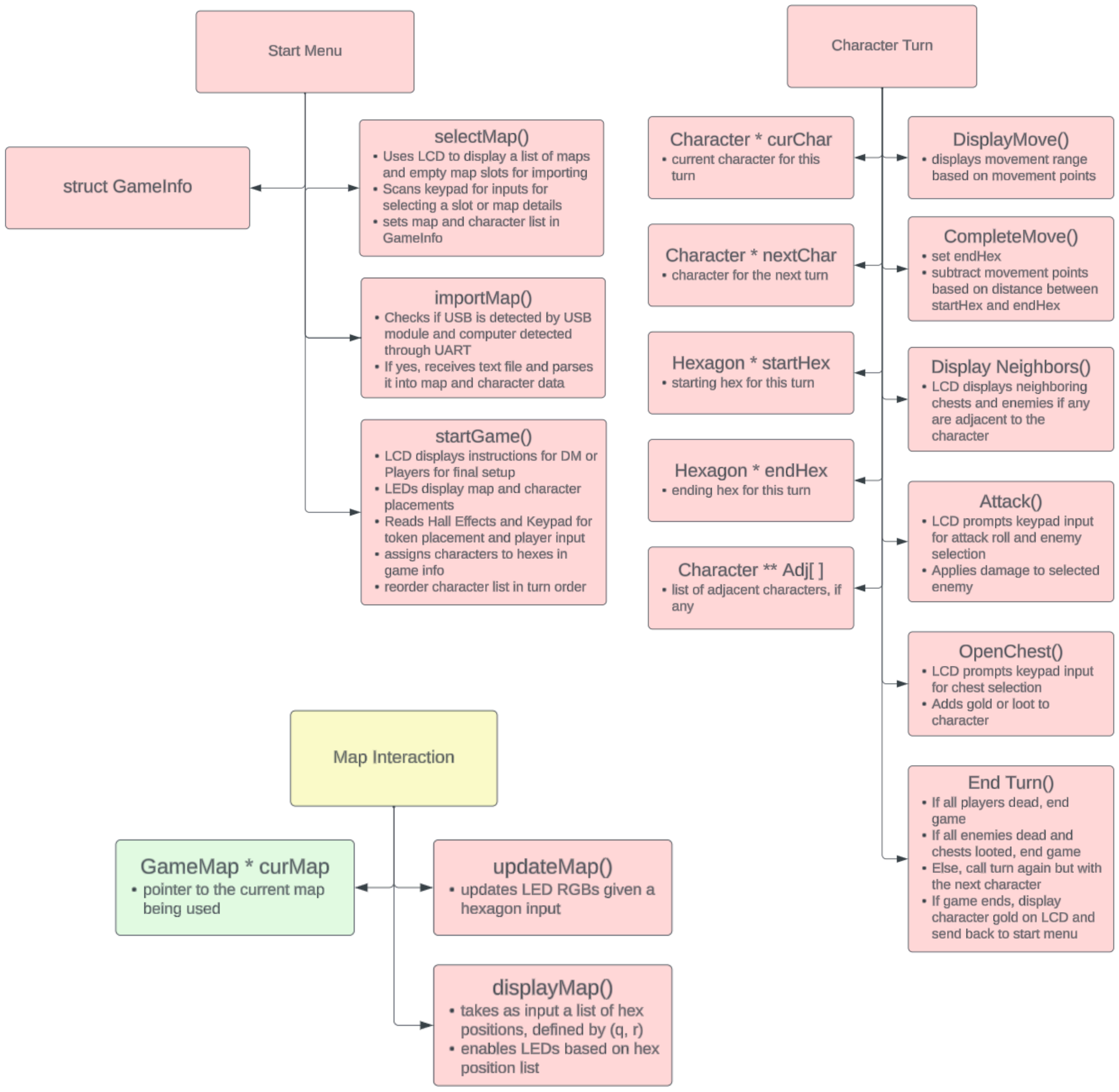

class Hexagon
int _q
• q coordinate
int _r
• r coordinate
int _row
• row converted from (q,r)
int _column
• column converted from (q,r)
int _type
• hexagon type
bool _passable
• determines if player can pass through hexagon
Character * char
• character on this hex
GetHexRow()
• gets row of hex
GetHexColumn()
• gets column of hex
GetType()
• gets hex type
GetCharacter()
• gets character, if any, stored on this hex
struct GameInfo
Character ** charList[ ]
• pointer to a list of all characters in the game
GameMap * curMap
• pointer to map object for the current game
class GameMap
int _rows
• number of rows
int _columns
• number of columns
vector<vector<Hexagon*>> map
• 2D matrix of Hexagons representing the map
GetHex()
• gets hexagon given a row and column
ChangeHex()
• changes a hex's type
PossibleMovements()
• gets all possible movements from a starting point given a movement amount
HexDistance()
• returns the distance between two hexagons
FieldOfView()
• returns vector of hexagons based on a starting hexagon and given range

Figure 6: Main Gameplay Software Structure


Start Menu
struct GameInfo
selectMap()
• Uses LCD to display a list of maps and empty map slots for importing
• Scans keypad for inputs for selecting a slot or map details
• sets map and character list in GameInfo
importMap()
• Checks if USB is detected by USB module and computer detected through UART
• If yes, receives text file and parses it into map and character data
startGame()
• LCD displays instructions for DM or Players for final setup
• LEDs display map and character placements
• Reads Hall Effects and Keypad for token placement and player input
• assigns characters to hexes in game info
• reorder character list in turn order
Character Turn
Character * curChar
• current character for this turn
Character * nextChar
• character for the next turn
Hexagon * startHex
• starting hex for this turn
Hexagon * endHex
• ending hex for this turn
Character ** Adj[ ]
• list of adjacent characters, if any
DisplayMove()
• displays movement range based on movement points
CompleteMove()
• set endHex
• subtract movement points based on distance between startHex and endHex
Display Neighbors()
• LCD displays neighboring chests and enemies if any are adjacent to the character
Attack()
• LCD prompts keypad input for attack roll and enemy selection
• Applies damage to selected enemy
OpenChest()
• LCD prompts keypad input for chest selection
• Adds gold or loot to character
End Turn()
• If all players dead, end game
• If all enemies dead and chests looted, end game
• Else, call turn again but with the next character
• If game ends, display character gold on LCD and send back to start menu
Map Interaction
GameMap * curMap
• pointer to the current map being used
updateMap()
• updates LED RGBs given a hexagon input
displayMap()
• takes as input a list of hex positions, defined by (q, r)
• enables LEDs based on hex position list