Settlers of Stepán

Aaron Dorrance, Jack Edwards, Kyle Hilbert, Sarah Holtz, and James Libbey

Table of Contents

1. Introduction (about 3-5 pages)

The problem being solved

The Settlers of Catan is a popular board game that is known amongst a large portion of the student population. In this game, players gather and trade resources to build a colony on an island, competing to become the greatest civilization on the board. However, the game board consists of many pieces which can be hard to keep track of as well as tedious to set up every time a group wants to play a game. Furthermore, people who have never played the game before may struggle with learning the rules during their first run-through of the game. Our goal is to solve the numerous challenges that occur with a traditional board as well as make it easier for new players to learn the rules.

Actual Settlers of Catan game board

High level description of our solution

At the highest level, our solution is to create an electronic board, which will be able to solve the problems associated with traditional boards with board games. Also, our electronic board will be accessible by a GUI, which will allow us to control and solve the problems associated with learning a new game. More specifically, our board was designed around the main aspects of the game Settlers of Catan. In the actual game, the board is composed of 19 hexagons, which have roads and settlements running through them. For our electronic board, each of the 19 hexagons is represented by 19 hexagon shaped PCBs, which display all of the information represented on the traditional board through LEDs, and 7-segment-display, and push buttons. For the roads and settlements running through the hexagons, we use an LED strip, which illuminates differently in order to indicate the presence of a road/city/settlement for each different player around the board. For the GUI, we will create a server using the Raspberry Pi, which can be connected to. Upon connection, a helpful GUI will walk you through how to play the game and explain any rules you might be violating.

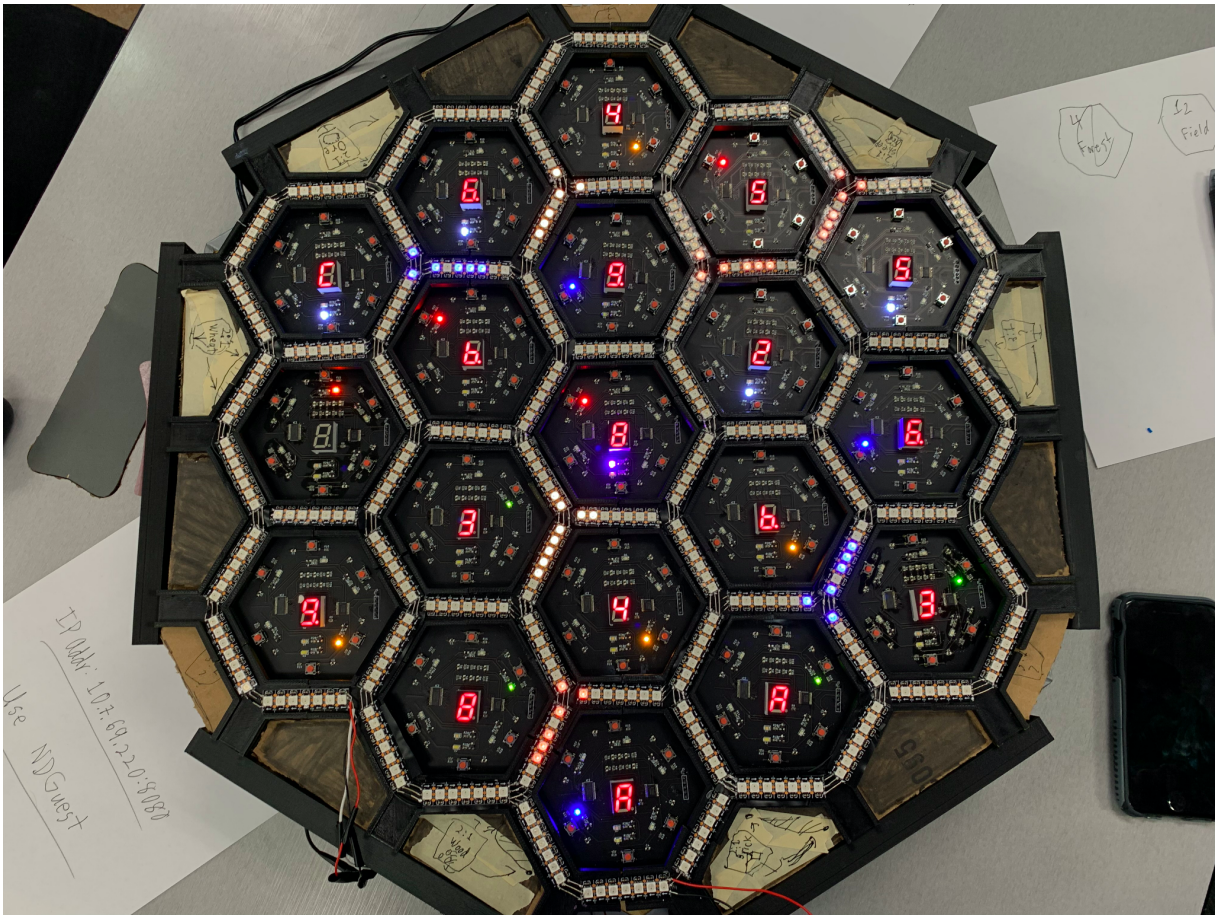


Image of our completed board solution, with a sample game state loaded on.

Basic design results

Overall, the physical board worked quite well. Each of the hexagons was operational and communicated well with the software to display the game state. The LED Strip was able to successfully display the player's roads, settlements, and cities, and the buttons were able to indicate the correct hexagon tiles or sections of the LED strip. However, it wasn't completely perfect. By this, we mean that every once in a while, a section of LEDs on the LED strip would mess up or a hexagon's MOLEX connection would come loose and disconnect the board. Additionally, the complex of wires underneath the board was an absolute nightmare, and our final product definitely wouldn't be able to stand the same wear and tear as a traditional board (but who cares? It looked way cooler).

On the software end, the game logic was completely functional and everything was able to communicate smoothly. When the GUI was implemented, there were some snags here and there with a few features not working perfectly, but overall, it was easy to connect to, looked good, and perfectly communicated any errors to the users and guided gameplay well.

2. Detailed System Requirements

Due to the many sided-nature of this project the design requirements will be categorized by software and hardware. The software requirements center around the game logic run by the Raspberry Pi and the user interface, while the hardware requirements will center around the PCBs, the LED strip, and the game enclosure.

The primary logic and control mechanisms will run on a Raspberry Pi, which will be provided with power and I/O. The system must configure correctly and run the game upon the whole product being powered up. There should be a minimum amount of steps for the end user to power up the product, and minimal configuration should be required to select game parameters and initialize gameplay. The Raspberry Pi should rarely, if ever, need to be accessed to update or modify anything, and never in the course of a typical game.

The game must support up to four players-- this entails four connections to the LAN, four designations for player settlements and pieces on the board, and four sets of game parameters used in the processing. The game program must allow

players to connect, access the game interface via a web browser, and transport data between the players' devices and the system. The web-based portion of the player UI will be intuitive, responsive, and easy to use. The web-based portion of the user interface should not take away from the physical game board itself, so that the design does not resemble an app or computer game more than it does a physical game board. The hybrid of on-board information and web-interface information must be carefully balanced to still make major improvements on the current physical game, while not making it a largely web-based game, which would be something else altogether.

The PCBs will have LEDs that can be configured to show resources and the presence of the robber. They will have a seven segment display to indicate the number of the resources.They will also have buttons that will allow the players to select both which side of the hexagon they'd like to build a road on and which corner of the hexagon they'd like to build a settlement or a city on. The user must be able to interact physically with the board, both by indicating positions on the board with buttons, and reading information conveyed by various displays. This is still the main thrust of the game, where the web control/display interface is secondary. In order to maintain the aesthetic of the original Settlers of Catan board the PCBs must be able to be cut in a hexagonal shape.

The LED strip will have the capacity to indicate roads, settlements, and cities. The roads will be indicated by three LEDs going between the hexagonal PCBs. The settlements and cities will also be specified by the LED strip with one LED at each corner for the settlements and two LEDs at each corner for the cities. In order to accomplish this the LED strip must be able to individually address each LED.

The game board and container should resemble the original Settlers of Catan board and thus should be hexagonal, with each edge being no more than 30cm in length. The width and height must be adequate to fit all of the electronic components and be sturdy enough to hold all of the PCBs and contain all of the wires and support all nineteen of the PCBs.

3. Detailed project description

3.1. System theory of operation (how the whole thing works)

The system initializes by setting up the game board. This involves both creating a random "island" within the code while also representing it through the physical game

board so that players can work with a physical object. Each game space contains three pieces of information: the resource it produces, the number that must be rolled to produce that resource, and whether it has a robber. The resource produced is represented by a colored LED, of which there are 6 colors: one for each of the five resources plus the desert, which does not produce any resources. Likewise, the robber's presence is indicated by a seventh LED. The number that causes resources to be produced is represented in hexadecimal on a 7-segment display (possible numbers range from 2 to 12, as the sum of two dice is used to determine resources).

The system then allows players to enter the game through a GUI that can be accessed through a personal device (phone, computer, tablet). The Raspberry Pi connects to a local network and sets up a LAN, through which any device with a standard web browser can access it. On the screen, players may select their name, color, and where they are in the turn order, with internal logic returning a notification and stopping players from attempting to use a claimed trait for their player profile.

Once all players indicate they are ready through the GUI, they may build their initial settlements in accordance with the rules. The internal game logic goes through a special set of phases allowing players to build in accordance with the set-up rules rather than the standard build procedure. Once this special set of turns passes, the *real* game begins.

The main screen for players during gameplay includes four elements: a "status table" with the player's resources and victory points listed along with who possesses the Longest Road and Largest Army achievements; a list of buttons to perform game actions; a "reset game" option; and a text field listing the build costs of all the available projects. The options change depending on the phase the player is in and whether they are the active player.

On each turn players begin with the trade phase. Players may select a type of trade that they would like to propose (listing the player or bank with which they want to trade and the resources offered). If trading with the bank, the trade is executed to update the player's resource count if the trade is valid. If trading with a player, the player the trade is proposed to may refresh their page to receive a trade notification (it is assumed players will have basic interpersonal communication skills, though we acknowledge that is not a guarantee with engineers). If the trade proposal is accepted by the other players, the internal logic updates the resource count.

Players may select an option to move to the build phase. During the build phase, players may select a project they would like to build. The internal logic returns whether the project is valid. If it is not, the GUI returns a notification, otherwise the player receives a notification telling them to select the location they would like to build their project. The player must then press a button on the physical board before the project expires, which prompts the internal logic to confirm that the location is valid. If valid, the LED strip is updated on the game board and the player's resources are deducted; otherwise the player is informed why the location is invalid.

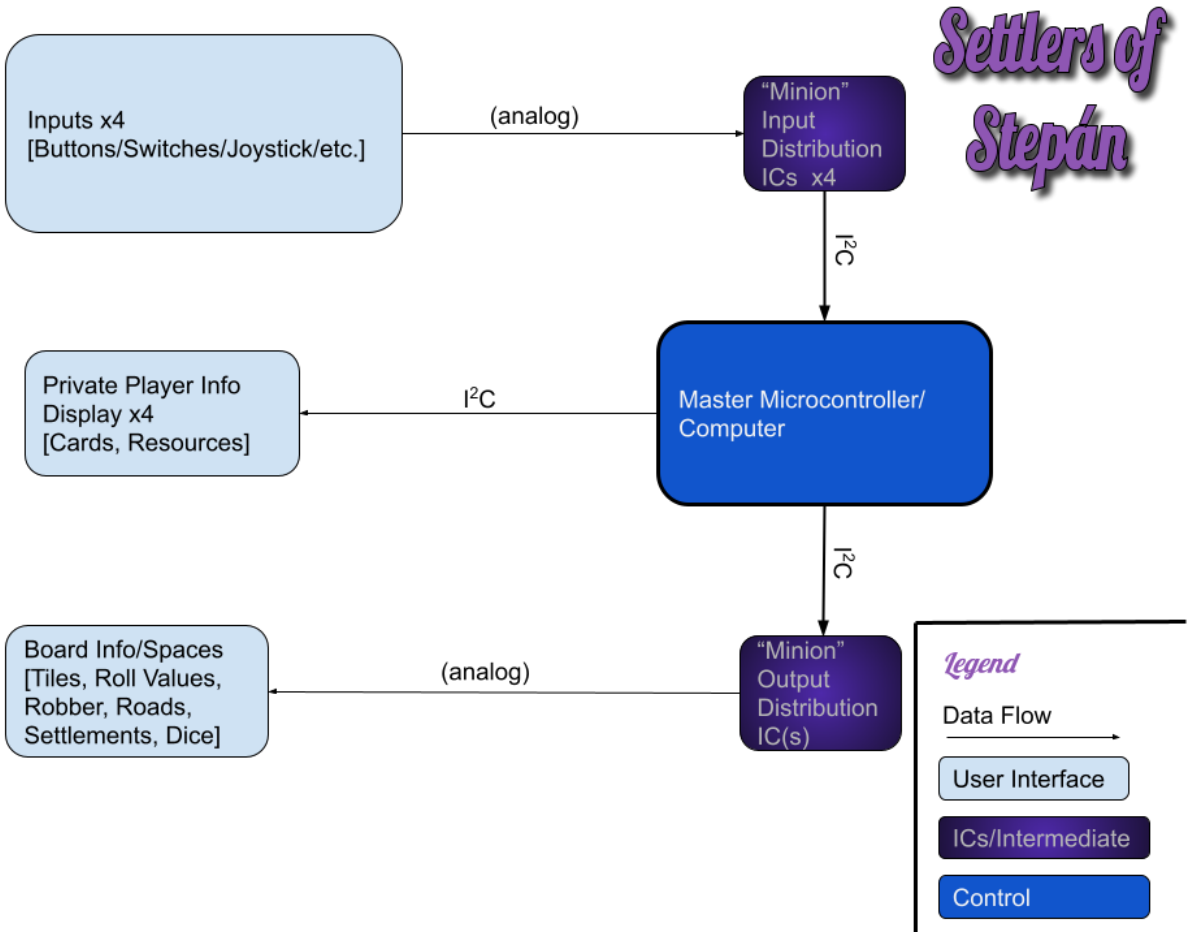
When a player ends the trade phase, the resource die is rolled by the next player through a random number generator and players are allocated resources automatically. When a player updates their page, they receive a notification of what was rolled and which resources they receive. If a "7" is rolled, the robber event is triggered. During a robber event, players with too many resources receive a notification to "pay taxes" by deducting half the resources they own. After taxes are paid, the player who rolled selects a "victim" they would like to steal a resource from in the notification window (the robber really messes around with who has what) and selects a location to move the robber towards on the physical board. If the hexagon is adjacent to the victim, the game updates the robber location and gives the stolen resource at random to the player who moved the robber; otherwise a notification is given and the player must try again with moving the robber.

There are also development cards that players may purchase and play, which utilize many of the same game actions as the existing logic and use internal calls on the API. Similarly, "achievements" that provide bonus points for certain players are determined through algorithms and trackers whenever an action is taken that may update who possesses the achievement.

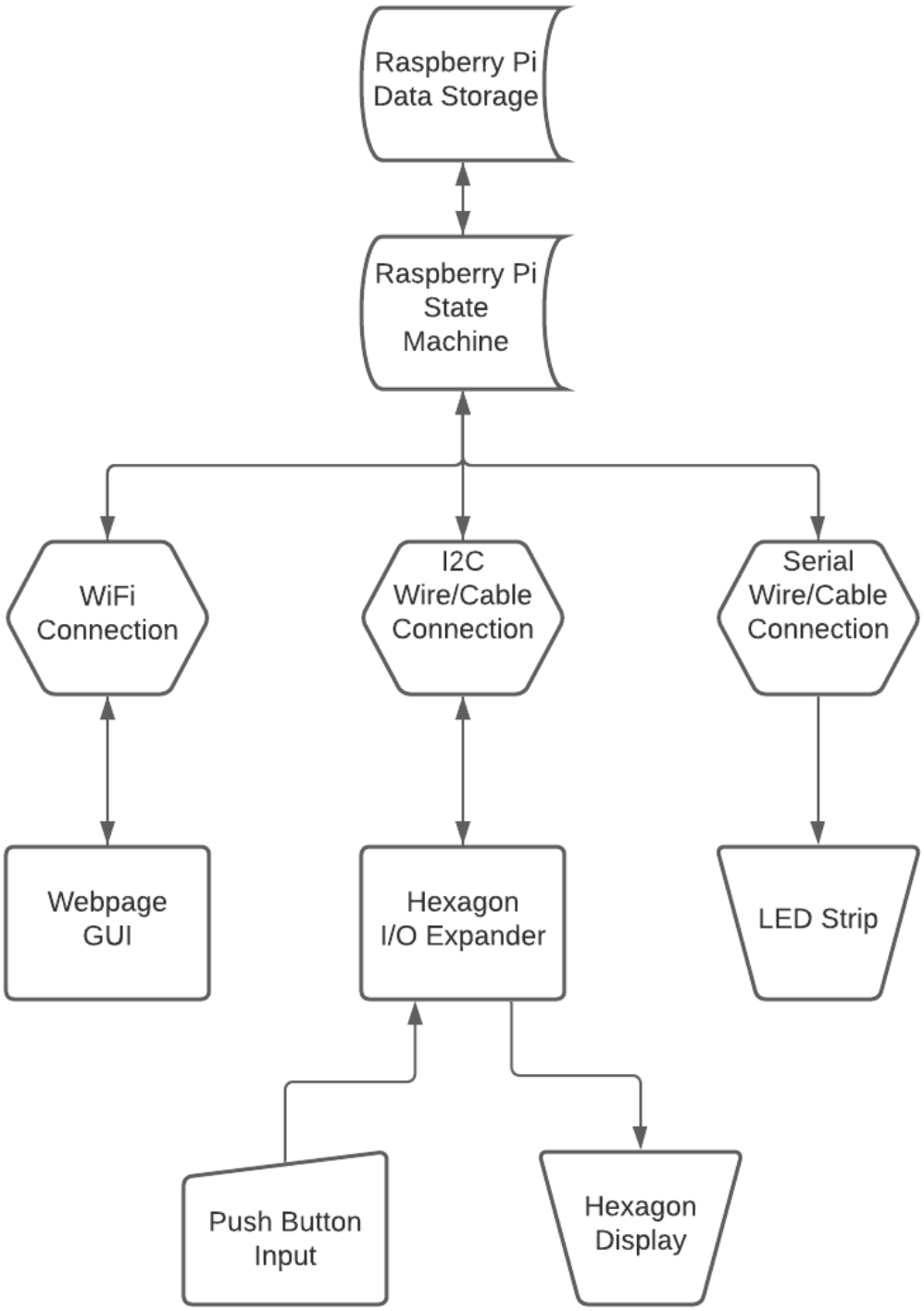
This integration of virtual game logic with the user feedback and input on the physical board and GUI form a cohesive experience where people may play a standard game of The Settlers of Catan.

3.2. System Block diagram

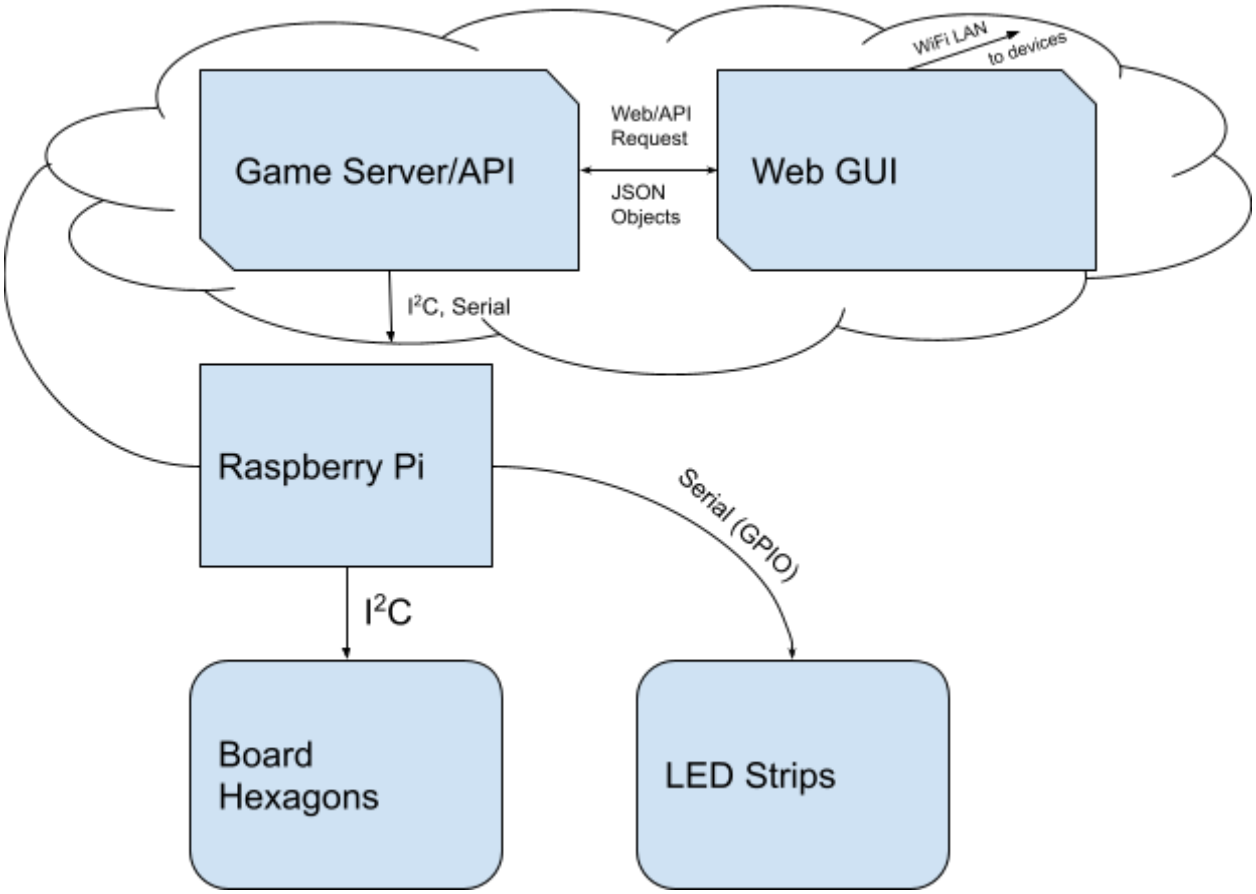
Initial/Prototype Block Diagram, October 2020



Block Diagram from High-Level Design, November 2020



Refined Version of Diagram from Design Review 3, April 2021



3.3. Detailed Design and Operation Web-based Graphical User Interface

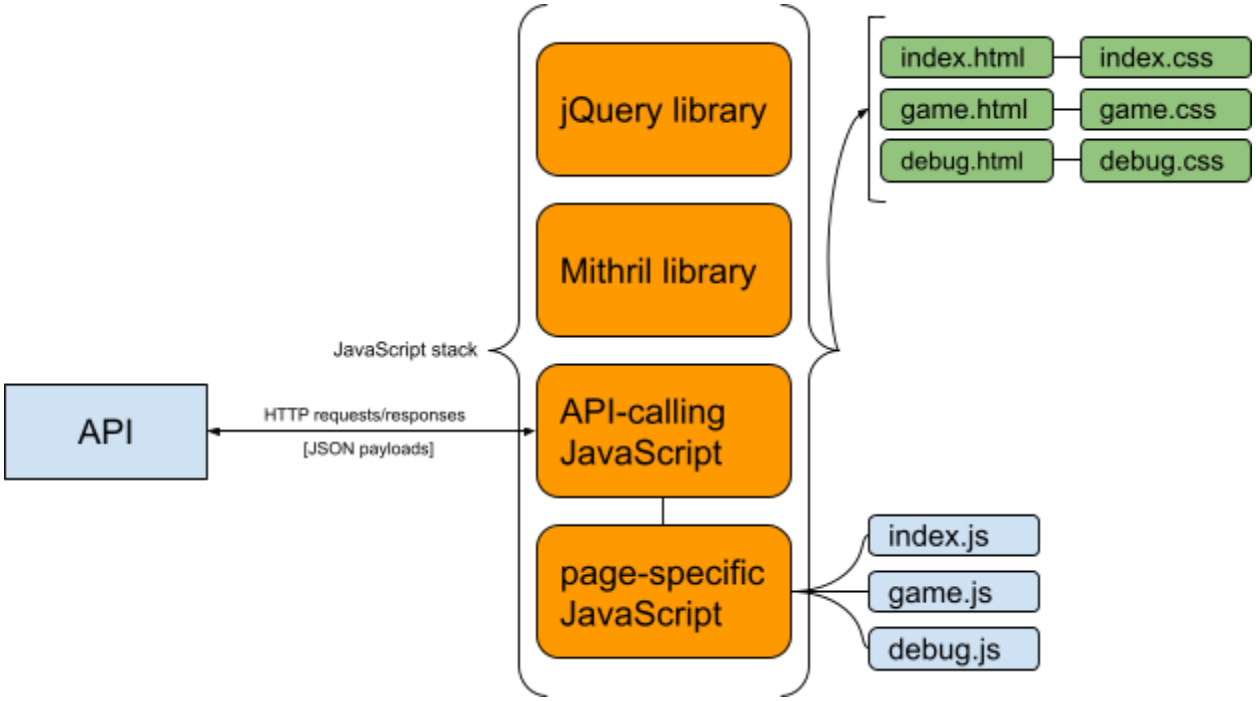
Requirements

From section 4.2.5 of the High-Level Design, this subsystem's requirements are as follows:

- There should be separate GUIs for each player's device in order to distinguish which player is using which device
- The GUI should be friendly to multiple types of devices, including smartphones, tablets, and laptops
- Users should be able to input the following game actions:
    - Roll dice
    - "Maritime trade" with the bank
    - Propose trade to another player
    - Accept trade from another player
    - Build a development
    - Select a development type to build

    - Play a development card
    - Pass turn
- The GUI should display the following information for each player:
    - Number and type of resources owned
    - Development cards owned
    - Who possesses Longest Road and Largest Army
    - Running Victory Point total of player
    - Known Victory Point total of opponents
    - Explanation of mistakes or illegal moves that the player attempts

Flowchart



The above figure shows the function and relation of the files that comprise the Web GUI. The “outside” or “user-facing” side is on the right, while the more internal parts that deal with the API are on the left.

The three actual web pages that make up the UI are implemented in HTML, with their respective CSS documents that describe their visual style. Each page has a “stack” of JavaScript files that make the GUI functional. First, the jQuery and Mithril libraries are imported, followed by a file which contains functions that call the game API, and finally a JavaScript file specific to each web page. This last document implements the GUI behavior specific to each web page.

Design Decisions

First, we chose the trio of HTML, CSS, and JavaScript to build each GUI page because this is the established standard for this--and JavaScript is the go-to language

for making web pages dynamic and able to interact with other applications, which is essential in a user interface.

In the JavaScript, we use two libraries: jQuery and Mithril. jQuery is used to visually modify elements on a web page, which is crucial for creating an intuitive, interactive GUI, needed for a game such as ours. We chose it to implement this part of the UI because it is extremely simple to show, hide, and otherwise modify elements on web pages based on external factors. Secondly, we chose a library called Mithril, which deals extensively with web servers and HTTP requests. This was a relatively straightforward solution to use JavaScript to call API functions and send/receive data, which has to be done using HTTP requests and responses. It has functions that make it very easy to pass data into and receive data from an API such as ours over HTTP.

Using HTTP requests and responses was the obvious way to exchange data between the GUI and the game API and server, since they were implemented in two entirely different languages, processes, and most importantly, devices.

Basic Code Flow

Given the construction of all of the software subsystems, the API cannot initiate changes in the GUI--all updates, data transfers, actions, etc. must be initiated from the GUI side. So, we will start there in describing the functionality of the user interface.

Actions are initiated on the GUI in one of two ways: 1) the user performs an action by doing something with an interactive element on a web page, or 2) an action is initiated subsequent to a user's input automatically by calling another function. For example, the bulk of the JavaScript files concerning each page contain code that runs when a user selects a button on the GUI, which is the majority of the actions in the game. This might use jQuery to show, hide, or modify HTML elements on the page, or call functions related to the API that were defined in other files. Also, actions can occur without the user directly initiating it, such as certain functions being called when the user refreshes their web browser, or when the user does something unrelated, but gives the GUI and opportunity to refresh certain data from the game.

```
$('#reset').on('click', () => {
    $('#confirm').show();
    $('#reset').hide();
    $('#players').hide();
    get_state(static_refresh);
    get_notifications(player,notif_pop);
});
```

Example of GUI changes and other API functions that are called when a GUI element is clicked

As mentioned, the actions related to modifying things from the user's point of view are almost entirely performed as a result of interacting with the GUI. However, when the API needs to be called to send or receive information to and from the game, function calls and passing needs to occur. For example, when a value from the GUI needs to be sent to the server, a jQuery function is used to get this variable and pass it into the relevant API-calling function, located in the general JavaScript file that is common to all pages and contains functions to call each API function. A similar process

occurs when getting information from the API to update the GUI, just in reverse. If variables need to be passed between the page-specific JavaScript files and the API-calling file, they need to be declared properly to be accessible to all the right files.

```
var robber_steal = function(pass) {
    m.request({
        method: "PUT",
        url: "/api/move_robber",
        body: {mover: player, victim: rob_victim},
    })
    .catch(function(e) {
        $('#error_pop').show();
        $('#error_message').text(e.message);
        err = e;
        e = null;
    })
    .then( () => {
        if (err == null) {
            pass();
        }
        err = null;
    })
}
```

Example of a function that is used to call an API function and send data to it

Finally, whenever an API-calling function is activated, several things happen. First, any data that needs to be sent to the API is assembled into a JSON object. An HTTP request is made with the relevant API function as its endpoint, and the JSON object as its “body” or “payload”. If the request returns an error, an error-catching clause prevents the GUI from changing, only allowing the user to see the changes in the GUI they were expecting if the API call was successful. Then, if the API is going to return any data back to the GUI as a result of an API call, the payload of the HTTP response is set to a JavaScript variable, which we can then use in the GUI to display information to the user.

Communication Protocols

There are no communication protocols internal to the GUI itself, only the natural information-passing that is inherent among web systems made up of HTML and JavaScript files. The one communication protocol that the GUI is concerned with is dealing with HTTP requests and responses to and from the API, but again, this is entirely encapsulated in the Mithril JS library. Such communication is entirely internal to the Raspberry Pi anyway, and is not managed by the GUI, so it will not be described here.

Subsystem Testing

As mentioned in Design Review 1, the first “tests” of this subsystem simply involved the linkage of HTML and JS files, mainly concerned with jQuery actions, to ensure that the flow of the web pages and UI elements was ironed out. This prototype testing was done entirely independent of the API or server, since these did not exist yet.

After the server and API were constructed, it was possible to flesh out the GUI's functionality in preparation for the final product by running the server hosting the API on a computer while accessing the GUI from the same machine. The entire software subsystems and communication between them was present in testing this way, which enabled very rigorous testing of the GUI and how it worked together with the API. Basically, this involved running a real game with the API and server, and then methodically testing each necessary function of the game, making sure the correct links and data were present between the GUI and API. We would ensure that the API was receiving the expected data from the GUI, and that the GUI was being properly updated by data it received from the API.

Finally, in the late phases of the project, we ensured the multiplayer functionality of the GUI by hosting the server on a LAN instead of purely within a machine, allowing external devices to interact with it. This phase of testing was simple, with the only concerns having to do with multiplayer interactions, adding players to the game, variables associated with each player, etc.

3.4. Detailed Design and Operation of the Game Logic and API

3.4.1. Requirements

This subsystem needs to provide an API (Application Programming Interface) that allows players to interact with the game. Also, it needs to check all intended user actions against the rules of the game to make sure that no rules are violated. It also interfaces with the hardware to get input and light up all the correct LEDs.

3.4.2. Design

The interface for all the game logic and output is the API. This is all written in Python because it is a simple and easy programming language that has libraries for websites and hardware output that can run on the Raspberry Pi. The server written in Python provides access to the API and website. You can inspect the server code in the server.py file. The API keeps track of the game logic by storing the entire state of the game in a 'state' object. All the API functions interact with the state to check the action against the rules and perform the actions by modifying the state.

There are two main modules. The "logic" module and the "output" module.

The output module handles IO by using I2C to interact with the PCA9685 and PCA9501 as described in the hex subsystem (section 3.5), and interacts with the LED strip as described in the LED strip subsystem (section 3.6).

The "logic" module provides all the functions to verify that the players are following the rules.

The API that is exposed to the website is described below. All of the following functions will return with a 400 error code if the action violates the rules of the game.

The API

```
get_notifications
# takes the name of the player and returns their notifications
# since last call

get_state
# return the state object representing the entire game
```

```
get_player
# takes the name of the player
# returns the object representing that player

new_game
# starts a new game with the default map

shuffle_game
# shuffles the resource tiles and roll numbers

add_player
# takes a name, color, and order (place in turn order)
# adds a player to the game with the appropriate color and
# place in the turn order

player_ready
# takes a name
# notifies the other players that the named player is ready

remove_player
# takes a name
# removes that player from the game

change_player_color
# takes a name and a color
# changes the named players color to the given color

randomize_players
# randomizes the order of the players

build_settlement
# takes a name and position
# builds a settlement at pos for the named player

build_road
# takes a name and position
# builds a road at position for the named player

build_city
# takes a name and position
# builds a city at position for the named player

draw_dev
# takes a name
# draws a dev card for the named player
```

```
end_turn
# takes a name
# performs all the actions required when the named player
# ends their turn including rolling the dice for the next turn

move_robber
# take the mover a position “to” and the victim
# moves the robber to the “to” tile and
# steals a resource from the victim and give it to the mover

pay_taxes
# takes a name and the taxes you pay
# gives the named players taxes to the “bank”

maritime_trade
# takes a name and resource you give and get
# executes a trade with the “bank” with the given exchange

propose_trade
# takes a name and trade
# propose the trade for the other tradee to verify

accept_trade
# takes a name and trade
# executes a trade that has been offered

reject_trade
# takes a name and trade
# rejects a trade that has been offered

play_knight
# takes a name, to, and victim
# plays a knight dev card
# the arguments to this function the same way as the robber

play_build_road
# takes a name and two positions, “one” and “two”
# plays build road dev card by building roads in the
# “one” and “two” positions

play_year_of_plenty
# takes name and two resources, “one” and “two”
# plays year of plenty dev card by giving the named player the
# two named resources

play_monopoly
# takes name and resource
```

```
  # plays the monopoly dev card with the named resource
```

3.4.3. Testing

I wrote a “console” that allowed me to test all the game logic and API on my own computer without having the physical system connected. It allowed me to type in API commands that would execute on the state file and then print an ascii representation of the state file (seen below in Figure 3.4.3.1). I could also directly print any of the keys of the state file with the command “print key”. I also programmed in “cheat codes” so that I could get the board quickly to a desired state for testing.

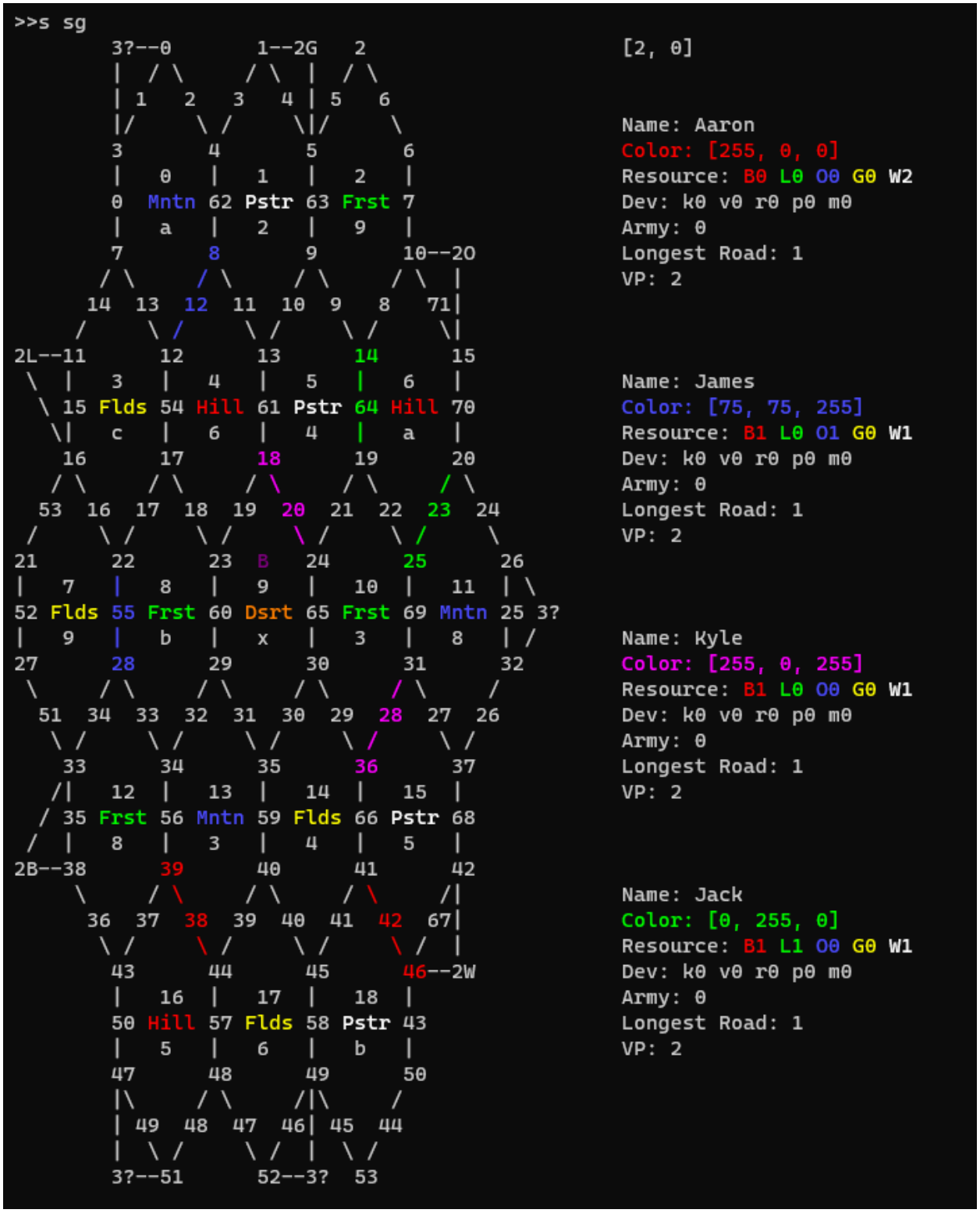


Figure 3.4.3.1 ASCII representation of the game state

3.5. Detailed Design and Operation of Hexagon PCBs

Subsystem requirements. These are the requirements that this particular subsystem must meet. (The combination of all of the subsystem and interface requirements must encompass the overall system requirements.)
This should include a schematic (hardware) or flow chart (software) for the subsystem, the function of the subsystem, the interfaces to other subsystems, etc. There should also be information about why you made the engineering decisions that you did – choice of components, kind of interface, choice of programming language, etc.
Schematics and software need to be described in addition to having a schematic or a code listing. For hardware, you should describe how your circuit works (for example, how your amplifier circuit and filters are designed/operate); for software, you should describe the overall flow of the code (is it interrupt driven, state diagrams, etc.)
If there are communications protocols involved in this subsystem, description and state transition diagrams should be included.
Subsystem Testing. Described how this subsystem was tested to ensure functionality

3.5.1. General Requirements and Design

The hexagons needed to fulfill the following requirements:

- The individual tiles must fit within a hexagon to match the geometric shape of the game board
- The design must be reproducible so that 19 identical PCBs may be created (one for each game tile)
- The PCBs must be capable of I^2C communication with the Raspberry Pi
- There must be a way to distinguish the PCBs for the purposes of addressing
- Users must be able to use the board to select any road or settlement location
- Users must be able to use the board to select any tile for moving the robber
- The PCB must be able to indicate which of six distinct resource tiles it may represent
- The PCB must be able to indicate whether the robber is present
- The PCB must indicate what number must be rolled to collect resources from its associated tile

The schematic and board layout of the hexagon PCBs may be found in the appendix.

3.5.2. External Connections

The hexagon includes external connections to power buses (5V and 3.3V), the I^2C bus, and an interrupt wire. This is in the order pictured in Figure 3.5.2.1.

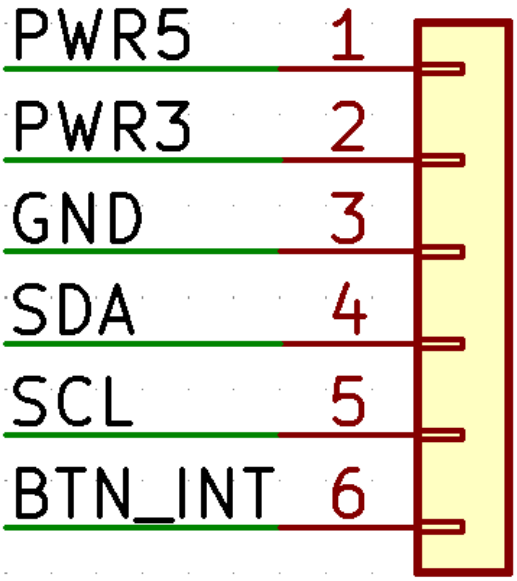


Figure 3.5.2.1: Hexagon Output Connector Order

This connection takes the form of a molex connector that ties each input to its respective bus (power connections to the appropriate power rail, the serial connections to the I^2C bus). The "BTN_INT" connection was omitted as the final software design used polling rather than interrupts to read buttons.

The 5V power is supplied by the power converter that is also supplying the Raspberry Pi, while the 3.3V power is supplied by the Raspberry Pi, with both sources sharing a common ground bus. The I^2C bus establishes a maser-servant system where the Raspberry Pi sends control signals to the PCA devices on the hexagons.

3.5.3. Button Inputs

Design

Standard surface mount resistors and buttons were used for this subsystem due to the ease of use and availability of bulk parts. Originally the subsystem had 12 buttons with an input matrix so that each settlement would have its individual button. However, it was determined that the number of diodes and additional buttons would significantly increase the project cost, so the settlement buttons were omitted in favor of a system where the player may use the buttons of the two adjacent roads to select a settlement. The road buttons were kept instead of the settlement buttons due to a combination of aesthetics and because road building occurs more frequently than settlement selection.

The PCA9501 was selected as a GPIO expander because the Raspberry Pi does not have enough IO pins to support all of the buttons, so instead this simple IC is used to delegate the interpretation of user input through I^2C.

Addressing

Table 3.5.3.1 shows the format used to address the PCA9501 through I^2C protocol.

Table 3.5.3.1: PCA9501 Addressing Format

Fixed	A5	A4	A3	A2	A1	A0	R/W
0	1	A4	A3	A2	A1	A0	

A5 is internally pulled up to HIGH for all PCA9501 by default to differentiate them from the PCA9685 LED drivers. A4-A0 are determined by the pull-up/pull-down network individualized for each hexagon board.

*Note: Although the PCA9501 can have its Fixed bit set to 1 when addressing EEPROM, the hexagon has the WC pin automatically disabled since this application does not utilize EEPROM.

Interpreting Output

When data is read from the PCA9501, an 8-bit value will be read back. Each button sets an individual bit, which is mapped in Figure 3.5.3.1. If multiple buttons are pressed, the sum of all button values will be returned instead.

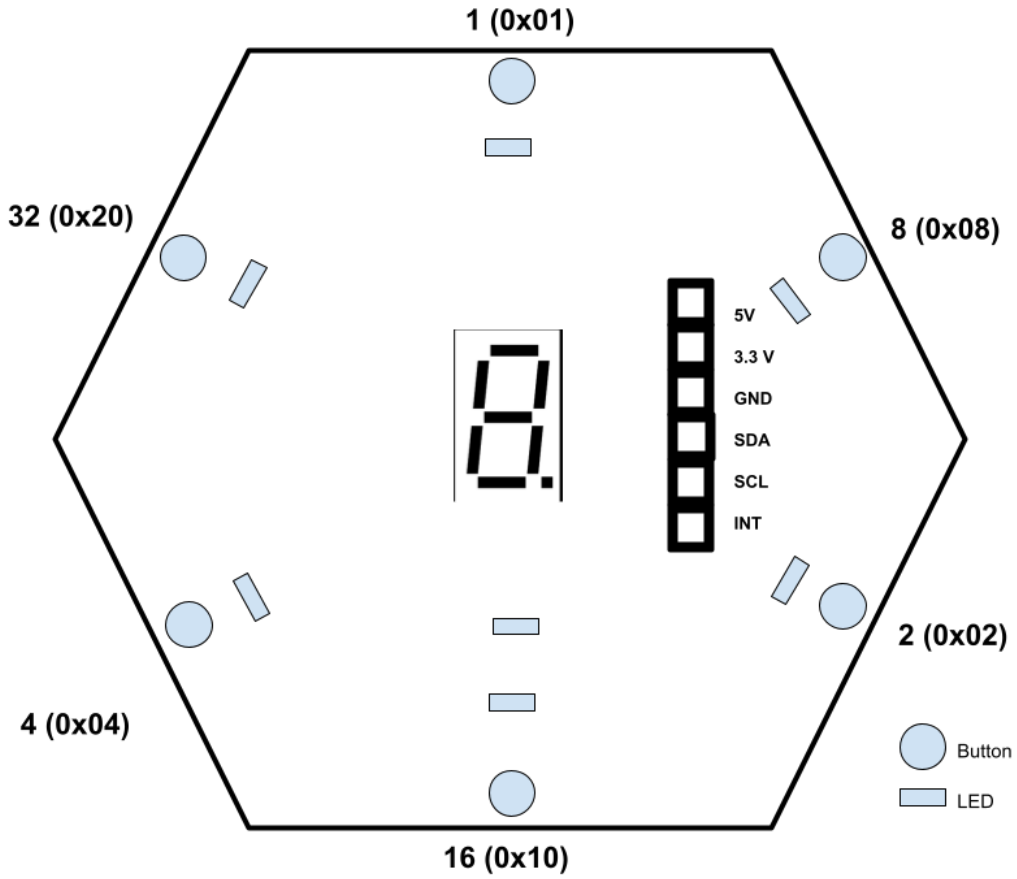


Figure 3.5.3.1: The Hexagon Button Map

3.5.4. LED Output

Design

Since the I^2C bus operates on a 3.3V scale, the PCA9685 must be supplied with V_{DD}=3.3V to properly interpret communications. However, 3.3V is not sufficient to support the voltage drop across green and blue LEDs, which are necessary to provide enough variety in color for the resource LEDs to be functional. Thus, the PCA9685 is used to send a control signal to the gate of an NMOS, which creates a "switch" that can turn on the 5V supply through the appropriate LEDs. Since some colors, such as red, still have low voltage drops, higher resistive loads were added to some of them to maintain similar current and power levels between the resource LEDs.

Surface mount components were selected again due to the ease with which they can be purchased and applied to PCBs in bulk.

Addressing

Table 3.5.4.1 shows the format used to address the PCA9685 through I^2C protocol.

Table 3.5.4.1: PCA9685 Addressing Format

Fixed	A5	A4	A3	A2	A1	A0	R/W
1	0	A4	A3	A2	A1	A0	

A5 is grounded to LOW for all PCA9685 by default to differentiate them from the PCA9501 GPIO expander. A4-A0 are determined by the pull-up/pull-down network individualized for each hexagon board.

LED Assignments

The assignments of LED pins to the different colors is listed in Table 3.5.4.2. The LEDs are also mapped as in Figure 3.5.4.1, which shows the positions of all LEDs on the hexagon.

Table 3.5.4.2: LED Color Assignments

LED Pin	**Color**	**Resource**
LED0	Green	Wood
LED1	Orange	Desert
LED2	Red	Brick
LED3	Blue	Rock
LED4	Violet	Robber
LED5	White	Wool
LED6	Yellow	Wheat

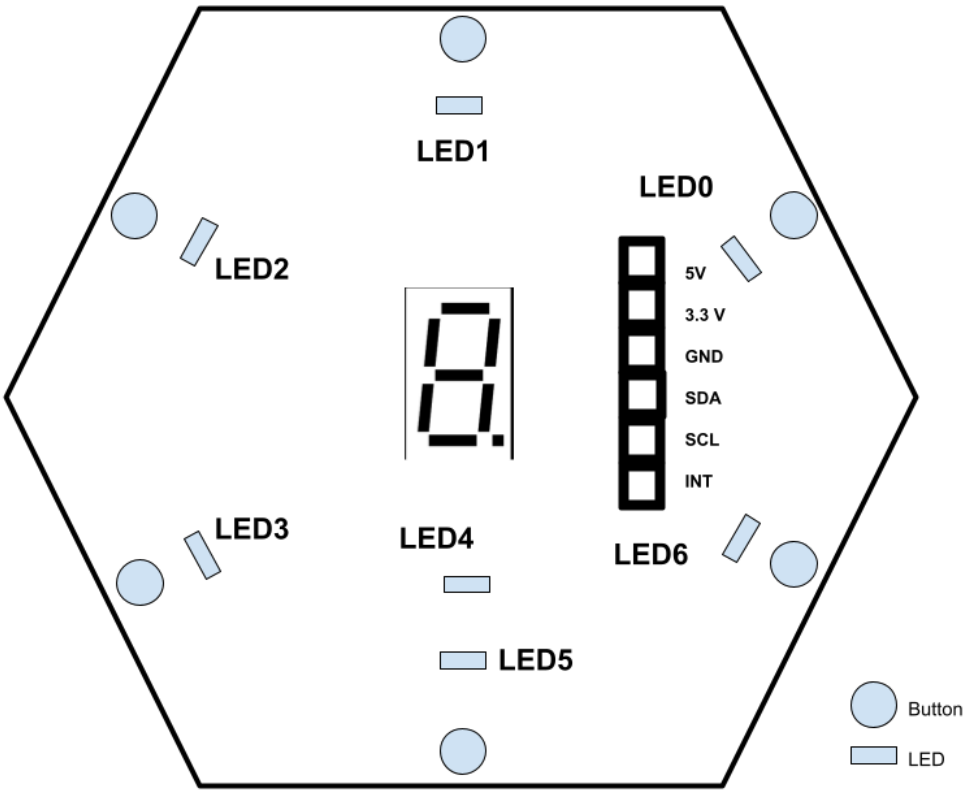


Figure 3.5.4.1: LED Positional Assignments

3.5.5. 7-Segment Display

Design

The 7-segment display is the only through-hole component (aside from the molex connection), this is due to a combination of balancing costs with having a large enough display to be conspicuous, as the numerals must be easily legible. We opted for a red display due to its efficiency and the ability to directly drive individual segments from the PCA9685 rather than setting up a set of NMOS controls like with the resource LEDs. The common cathode variant was selected due to pricing and availability.

Addressing

See Table 3.5.4.2 in Section 3.5.4 for information on addressing the 7-segment display, as it uses the same PCA9685 as the LEDs.

Segment Assignments

Table 3.5.5.1 lists all the sets of LEDs required to create the numerals that are utilized in the game of Catan. In case any custom numerals need to be created, Figure 3.5.5.1 shows the assignments of LED pins to individual segments.

Table 3.5.5.1: 7-Segment Numeral Codes

Numeral	**LEDs to Turn On**
2	LED8, LED9, LED12, LED13, LED15
3	LED9, LED10, LED12, LED13, LED15
4	LED10, LED12, LED14, LED15
5	LED9, LED10, LED13, LED14, LED15
6	LED8, LED9, LED 10, LED11, LED13, LED14, LED15
8	LED8, LED9, LED10, LED12, LED13, LED14, LED15
9	LED9, LED10, LED11, LED12, LED13, LED14, LED15
10 (A)	LED8, LED10, LED12, LED13, LED14, LED15
11 (b.)	LED8, LED9, LED10, LED11, LED14, LED15
12 (C)	LED8, LED9, LED13, LED14

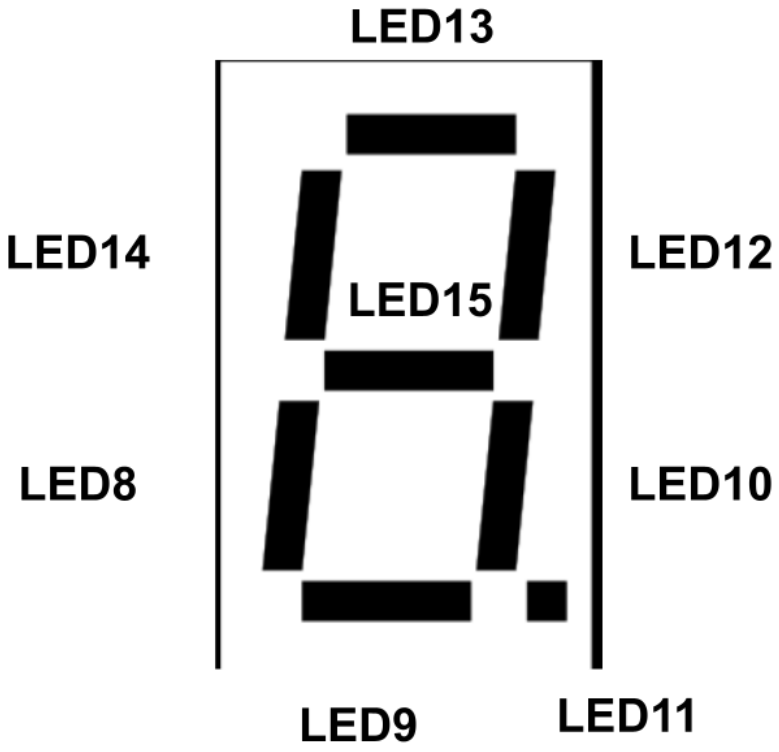


Figure 3.5.5.1: 7-Segment Input Map

3.5.6. Unit Testing

Five tests are conducted to determine whether an individual hexagon was working properly, four of which involve connecting a Raspberry Pi to the appropriate ports.

1. There are pads on the back side of the PCB that are connected to the 5V, 3.3V, and ground planes, which may be inspected through a multimeter.
2. The I^2C communications were scanned to determine whether the PCA9501 and PCA9685 can be identified by the appropriate addresses. Errors are typically the result of solder bridges that create inaccurate addresses or pull SDA or SCL to each other or another node. This may be fixed by visually inspecting the ICs and using soldering equipment to remove the bridges.
3. The LEDs are then tested by executing a test script on the Raspberry Pi. All of the resource LEDs should light up. Individual LEDs may experience faults due to solder bridges at the PCA9685 or faulty connections in the LED/resistor/NMOS segment of the circuit. If none of the LEDs work, this could either indicate an error with the PCA9685 or the 5V power supply, which both must be double-checked in this case.

4. The 7-segment display is then tested by executing a test script that flashes all segments plus the decimal point. Like with the LEDs, an individual segment may have a fault due to a solder bridge on the PCA9685 or the through-hole connection of the display. It is unlikely that none of the segments work if the system already passed the LED test as any systematic error with the PCA9685 should be fixed by that point.
5. The PCA9501 and button inputs must then be tested by a Raspberry Pi script. Once the script is executed, all buttons must be pressed in sequence to confirm that each button triggers a unique value in the console. If an individual button does not work, the pull-down resistor and surface mount connections must be checked on the button along with its orientation. If no button works, the 3.3V supply and PCA9501 must be reinspected.

If the hexagon passes all of the above tests the unit's internal hardware may be assumed functional until it experiences additional soldering or severe jostling, at which point it must be re-tested.

3.6. Detailed operation of LED strip

3.6.1 Subsystem requirements

The LED strip had 4 major requirements. First of all, it had to be able to illuminate at different colors on command in order to indicate the different players. Secondly, it had to be able to indicate the presence of a road/city/settlement between hexagons. Thirdly, it had to be able to be easily differentiated whether someone has a city or a settlement. Lastly, it had to be able to be updated easily during the game.

3.6.2 Engineering decisions

In order to accomplish these subsystem requirements, we had to make several different Engineering decisions.

For the first requirement, the LED strip must have been able to illuminate at different colors. In order to accomplish this, we were debating between getting multiple LEDs of different colors or getting one single RGB LED, which could switch between colors for the players. Due to space constraints between the hexagon tiles, we decided to go with the RGB LED since we would only need to fit one line of LEDs between the tiles.

For the second requirement, the LED strip had to be able to indicate the presence of a road/city/settlement. In order to accomplish this, we felt that it would be easiest to use a single long strip of LEDs, that could each be individually addressed. After doing some research, we found a strip that seemed reasonable for us. It operated with 5V, and we

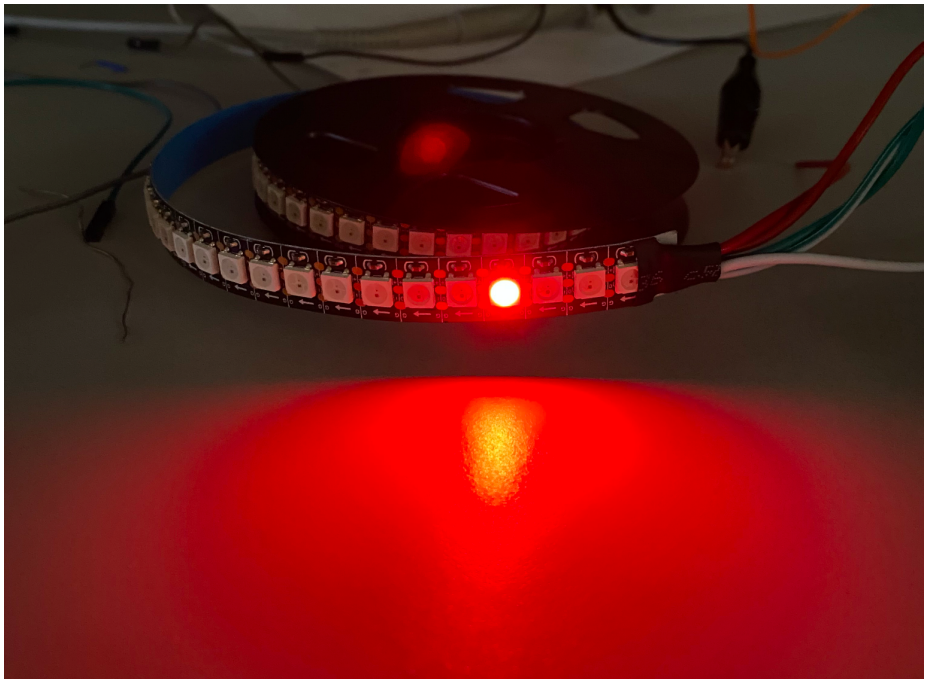
were already planning on using a 5V power bus, and it seemed easy to move around and customize the shape of.

For the third requirement, the strip had to be able to differentiate between city and settlement. In order to solve this, we had to figure out how to divvy up the LEDs between each hexagon. The solution we decided upon was having 7 LEDs between each hexagon. Then, the middle 3 would be used to indicate a road. The outer 2 would be used to indicate cities and settlements. If there was a settlement at an intersection, only 1 of the outer 2 LEDs would illuminate, but once a city was built, the second LED would illuminate. This was considered best since it would be easy to implement but also easy to read and understand in gameplay.

For the fourth requirement, the strip had to be able to be updated easily in the game. To do this, we created a single script that would update the status of the LED strip based on the status of the players. This script could easily be called by the API whenever someone built a road/city/settlement.

3.6.3 Subsystem Testing

The testing of the LED Strip was split into 3 separate phases. First of all, we had to ensure that we could illuminate a single LED on command at different colors. This would allow us to continue to control the entire strip with each LED individually controlled.



Successful test of being able to illuminate an individual LED a color (red)

Once this was done, we had to test if the LED Strip was able to fully illuminate itself when soldered around the board track.

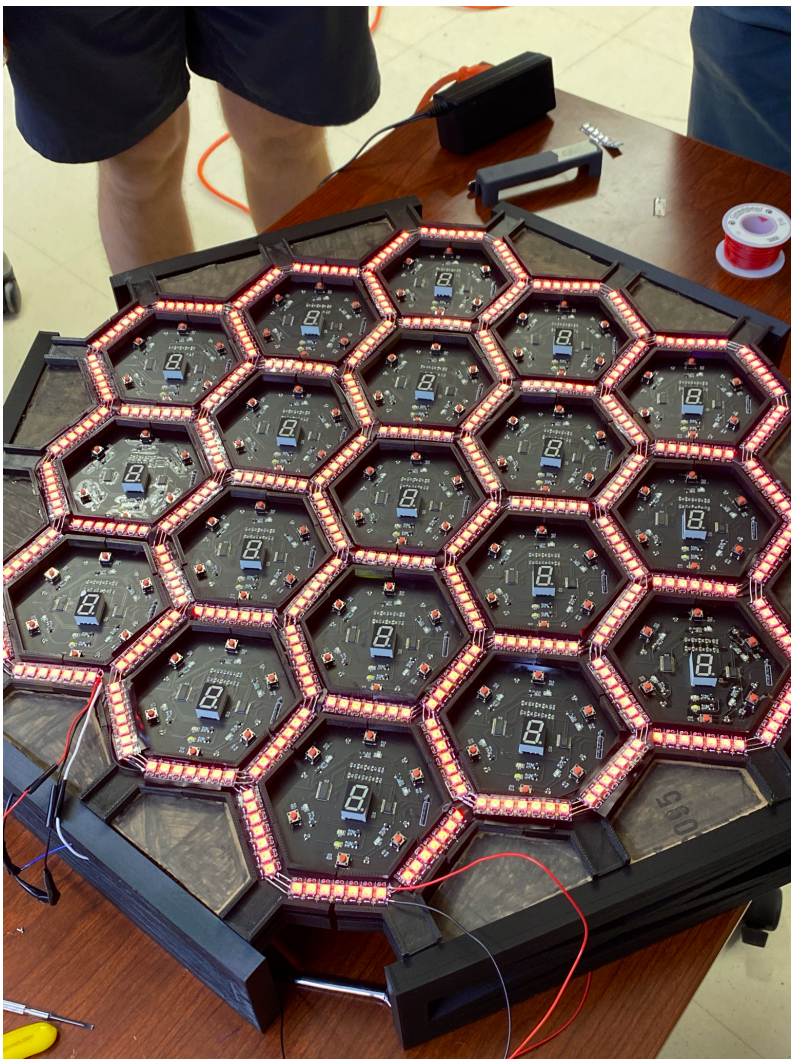
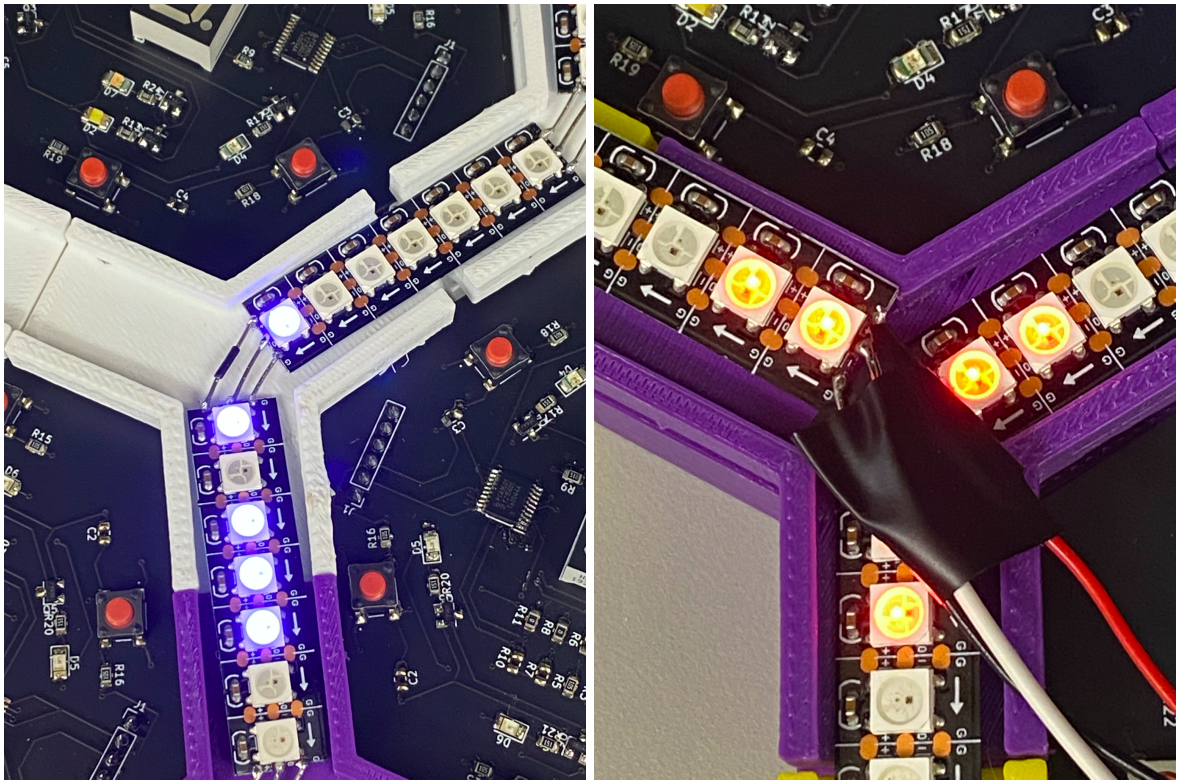


Image of LED Strip fully illuminated while in place around the board

After this, the only thing to test was the LED Strip's ability to represent cities, settlements, and roads correctly from the game API.



Figures showing the correct illumination of a road (3 blue consecutive LEDs on the left), a settlement (2 individual intersection LEDs in blue on left), and a city (2 consecutive connecting LED at the intersection on the right).

3.7. Design of Enclosure

The enclosure was designed to be modular in nature. The top of the board was designed to consist of fifty-four Y-joints that would be able to interlock together. They were designed to encase the PCBs while providing a path for the LED strip to lay on top of them. Due to the symmetry of the hexagonal shape half of them were printed with mae-type connectors while the other half were designed as female-type connectors. The sides of the board were designed to connect to the Y-joints with three of them designed to connect with the male connectors and the other three designed to connect to the female connectors. All pieces were spray painted black in order to give the enclosure a consistent aesthetic. They were designed using the Tinkercad software and printed using the 3D Print Cafe in Stinson-Remick. Images of the 3D designed pieces are provided in the appendix. The base of the enclosure was a 2 x 2 piece of plywood cut into a hexagonal shape that the 3D printed pieces could rest on. The wires were organized through the use of bus bars which would connect to the Pi.

3.8. Interfaces

Many interfaces between subsystems have already been described here, but we will summarize them concisely below:

Hardware

There are two interfaces that connect all of the hardware present in our project. The first is an I^2C bus, which connects the Raspberry Pi (controlling the hardware) with the hexagon PCBs which make up one part of the board. Each hexagon has a unique address for its LED driver chip and for its I/O expander (used to sense button inputs), set with resistors. The data (SDA) and clock (SCL) pins were connected in common among all of the hexagons, which resulted in them being present on a single I^2C bus. Then, knowing the addresses corresponding to the input and output functionality on each hexagon, we could communicate with whichever parts of the board we desired using the Pi.

The second hardware interface connects the LED strip matrix with the Raspberry Pi, which controls it. The strip we chose for this project has a single control data pin, which we connect to a general-purpose I/O pin on the Raspberry Pi. As the strip has its own proprietary serial protocol, taken advantage of by a corresponding Python library described in the LED strip subsection, we simply instruct the serial data made by this code to output on the pin we connected to the LED strip. The LED strip is then controlled by the Pi, using its own serial protocol, using this simple connection.

Software

Two connections comprise the software-related interfaces of this design. First, as described above, the GUI and API send data back and forth using HTTP requests. The API server makes this possible, by hosting all of the API files. The GUI, whose files are also hosted on this server, then makes HTTP requests of the API files resident on it, and the API responds appropriately with data and messages. This server is bound up with the construction of the API itself, and is described in detail in that section.

The other interface is that connecting the server, which hosts the GUI web pages in addition to the API, to the players' devices. This is accomplished by running the server with a specific IP address (the one that the Pi resides on the LAN with), rather than locally on the Pi only. Now, anyone can access the GUI and therefore operate the game, which enables multiplayer play. The default page that a user will see upon accessing the server is set to the "landing page" of the GUI. As above, the functionality for the server to reside publicly on a LAN with a specific IP is implemented in the server.py, which was described elsewhere.

4. System Integration Testing

4.1. Describe how the integrated set of subsystems was tested.

Four tests were conducted to see whether the set of subsystems worked. First, a series of end-to-end tests were conducted to see if a device could connect to the GUI and control the LED strip, 7-segment display, and interpret button input. Then, the full game

was tested to ensure everything worked properly. In general, a legal board layout (according to the rules on amount of each resource and tile number available) needed to be generated upon startup, and then the GUI was used to trigger events where button inputs on the board were used to "build" roads and settlements that would be updated on the LED strip. The systems could be considered fully integrated once build commands were properly interpreted, as that was the primary function conducted through system integration.

4.2. Show how the testing demonstrates that the overall system meets the design requirements

4.2.1. LED Strip End-to-End Test

We first added a temporary UI element to activate a specific LED on the LED strip, to test the end-to-end communication beginning with the web GUI, to the APIs and server, to the Pi's hardware outputs, and finally to the LED strip.

Video demonstration

We can see that the LED strip is totally off at the beginning of the video. Upon clicking a button in the UI, we pan back to the LED strip to see that a certain LED has turned on. This address has been specified in the LED API function, which demonstrates that we can address and control LEDs to suit our needs. You can also briefly see messages on the game server console, indicating that it received data from the web GUI to make this change.

4.2.2. Button Input End-to-End Test

We add another UI element, and upon clicking it, the game waits for a button to be pressed on the board. This "waiting" model is how the actual game will function, since every instance that the board will need to be interacted with, a UI action has to be performed first, so the game will know to wait for a button press, instead of being surprised by a button press at some random phase of the game. Below this test button, we can see text that will indicate what data is returned from the hexagon all the way back to the user interface.

Video Demonstration

We see a button on the test hexagon being pressed, and upon panning back to the screen, the number on the user interface has changed to 32, consistent with the identification of the button being pressed on the board (see Section 4 for confirmation). This confirms the end-to-end functioning and communication of the following

subsystems: hexagon inputs through I^2C, Raspberry Pi I^2C, game APIs/server, and web GUI.

4.2.3. 7-Segment Display End-to-End Test

We add a final test element to the UI to test the output side of the board hexagons. At the beginning of the video, we can see that nothing is lit on the hexagon.

Video Demonstration

After pressing the UI button, we look back to the hexagon to see that the 7-segment display is lit. Of course, we have confirmed that we can control individual LEDs to our needs as well, we just chose to light up the entire thing. This shows the functionality and chain of communication between the web GUI, game APIs, Raspberry Pi I^2C, and hexagon LEDs through I^2C.

4.2.4. Gameplay Test

We were able to successfully generate legal game board states, as can be seen in Figure 4.2.4.1, where the correct amount of each tile and numeral are displayed. The hexagon with no number represents the desert, so the absence of a number is intentional.

Figure 4.2.4.1: A Sample Board State

We were also able to successfully use the GUI to take our turns, building roads and settlements while receiving all of the appropriate notifications when attempting to make illegal actions.

Although we were able to get basic build functionality, much of the game still requires development. One of the first issues was how work needed to be done on the robber logic because it involves a sequence of events that could be broken through an error (taxes, selecting a victim, moving the robber, and resuming the normal turn). In one early iteration, the victim selection window would pop up on top of the tax window if the

player who triggered the robber has too many resources. The robber cannot be moved until the player pays taxes, but taxes cannot be paid since the window was inaccessible. It was also possible to refresh the page and make the robber window disappear, which locks the game since the player is incapable of ending their turn until they move the robber, which is not possible once the notification window is closed. Much of this was circumvented by improving the logic and order in which notifications were generated.

Another problem was that trading did not work as intended, as errors would be returned. It was difficult to trace the errors, but we were able to still demonstrate the game's functionality by using "cheat codes" left over from when the game logic was being tested in isolation. This allowed players to conduct trades, as the internal logic worked, the GUI was simply not compatible. This also applied to the purchasing of development cards from the bank.

Overall, the game was mostly functional based on the tests. The internal game logic worked, as is evidenced by how the cheat codes allowed a moderator to run pieces of the game logic that didn't work through the GUI, and the user input/feedback worked for building roads and settlements on the board, which crosses over between the most systems out of any game action (trading, the only feature that was still broken throughout the demo, primarily resided in the software). With even one additional week just to test the GUI, it is likely that the game would be fully functional since most of the problems stemmed from the interface being in an early iteration.

5. Users Manual/Installation manual

5.1. Installation

Our product will come pre-assembled so the only “installation” necessary will be finding a table to set it on and an outlet to plug it into.

5.2. Setup

All the Code will come preloaded on the Pi and the Pi will be configured as a LAN router. Once the board is plugged in you will be able to go to the wifi settings on your device and select “settlerofstepan” to connect to the board. When connected to the board, your device will not be able to connect to the internet so keep that in mind. Once connected, you can go to your browser and type in “http://settlersofstepan:8080” in to the url bar to access the Game website and start playing.

5.3. How the user can tell if the product is working

As soon as you start a game and start building roads you should be able to tell the hardware is working. If all the hexes light up and you can build roads with the buttons then the hardware should be all working. Otherwise if the game doesn't appear to follow the rules, you get stuck in turn, or the web GUI is non responsive then you have a software or configuration problem. Generally, if you can play the game without problem then the product is working.

5.4. Troubleshooting

If you're getting an error on the GUI and you can't figure out why. It's probably a logic error by the player or the code. The first step should always be to refresh the webpage and see if the problem is fixed. If it persists then move on to these other techniques. The most likely scenario is that you don't understand the rules and are trying to make an illegal action. You can find the rules of the game here: https://www.catan.com/service/game-rules under the Base Game heading for 3&4 players. Once you understand the relevant rules proceed to the next step.
If the action you are trying to take is within the rules. The game is probably stuck in a bad state. The easiest way to get it unstuck or diagnose is with the console. The console will come provided with the Pi. To run it, you must ssh into the Pi ("ssh pi@settlersofstepan" from your local computer command/bash prompt). Once connected to the Pi, you can run the command "sudo python3 console.py" to startup the console. From there you can just press enter to see the game state ASCII version which might clarify some things. If you make any changes using the GUI you have to use the "load" command to see them in the console, and to make changes in the console appear in the GUI you have to run the "save" command. From here you can run any API command or cheat code to get the game unstuck. I will not describe all the options here. From here you will have to look into the code to see your available options and figure stuff out from there. There really is no more generic advice that we can offer.

The most common hardware problem is a hex tile failing to light up properly. This is normally due to a bad connection to the bus bar which is most often attributed to the molex connectors. You can test this with a voltmeter by doing a continuity test between like pins on the other working hex PCB (5V on one hex to 5V on another and so on). The fix for this is normally replacement of the molex connector and the connecting wires.

If you think the LED strip is not working, you can open up the console as described above and run the command "c sa". This will light up the LEDs in the strip red in order. If the wave stops, you probably have a solder connection problem around where it stopped. The fix is normally to resolder the connection between LED strips where the light wave stopped in the "c sa" test.

6. To-Market Design Changes

The two biggest problems with this iteration of the design are the mess of wires we had underneath creating unreliable connections and the LED strip being an absolute pain to assemble. We'll address both these problems with a new physical design here. The idea is to put male pin headers on the bottom of all the sides of the hex PCB then those will slot into a female pin header on a connector that just connects one hex PCB side to another hex PCB side. This connector PCB would route 5V, 3.3V, GND, SDA, and SCL between all the PCBs and connect each internal side of a hex PCB to another hex PCB. This way power could still be routed in parallel to all the PCBs without the mess of wires. This also makes our physical assembly cleaner because all the hex PCBs would just sit in these connector PCBs and the connector PCBs could be screwed down into a piece of wood or HDPE (side view of this arrangement shown in figure 6.1).

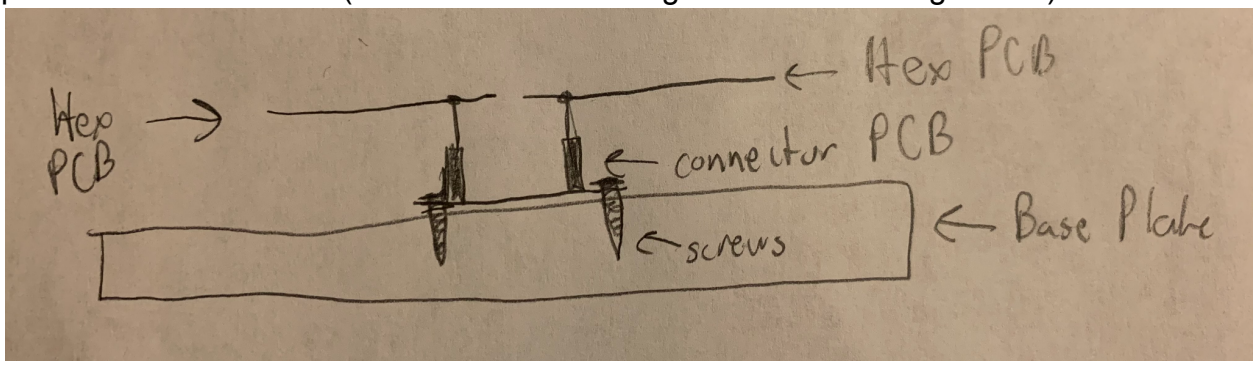


Figure 6.1 Side view of Proposed Hex PCB Interconnect

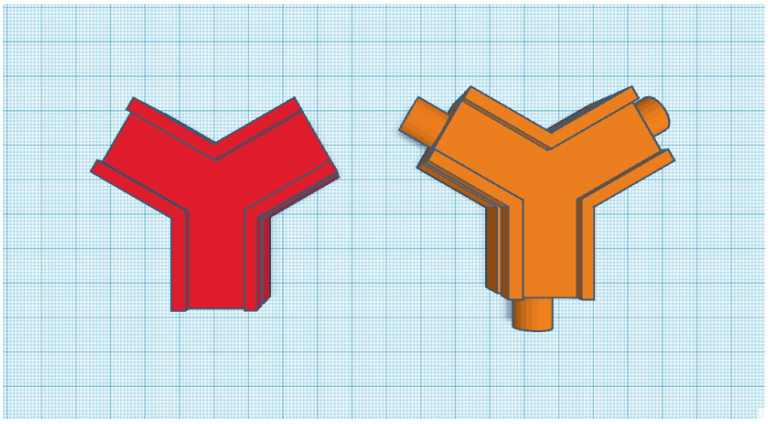
The LED strip problem will be addressed by using surface mount RGB LEDs on the hex to represent roads and settlements. This would require 3 LED drivers per chip with a creative routing of wires and transistor placements. The add on problem this presents is that we would run out of I2C address. To fix this I would add a PIC on the hex to provide each chip an individual I2C address and then use the PIC to send signals out to everything on each hex. We could also use this to add another GPIO expander (or a bigger one) so we could have a button for each road and settlement.
We also have to fix a number of bugs in the GUI and game logic that we just didn't quite have time to fix before the presentation.

7. Conclusions

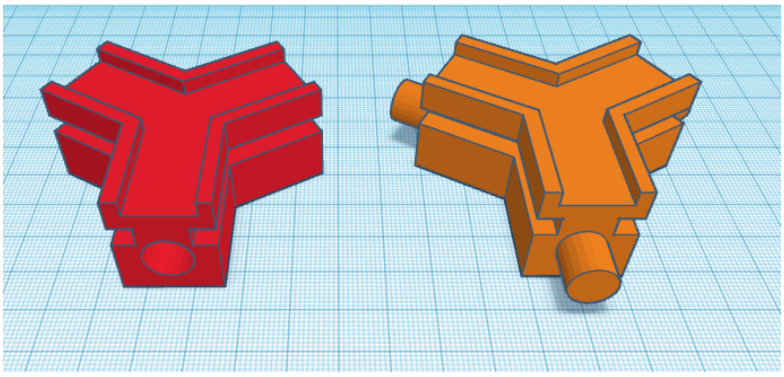
Throughout the process of building this project there are many conclusions that the group has come to agree on. The main conclusion is that in a future iteration of the project the implementation of the roads would be carried out through LEDs added on to the PCBs rather than a separate strip that needs to be cut up and re-soldered into a hexagon pattern. Another major conclusion is that testing should be conducted as early in the process as possible. The importance of prototyping cannot be expressed enough, and the physical aspects of the design are absolutely not a negligible factor. Ultimately the main conclusion is that to the extent that components of the project can be designed, prototyped, and tested, they should.

8. Appendices

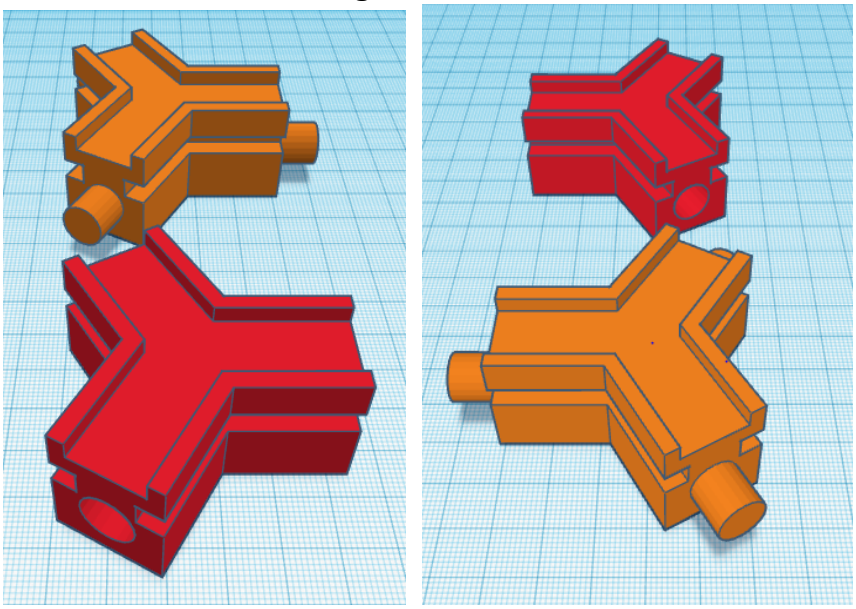
Hardware Schematics



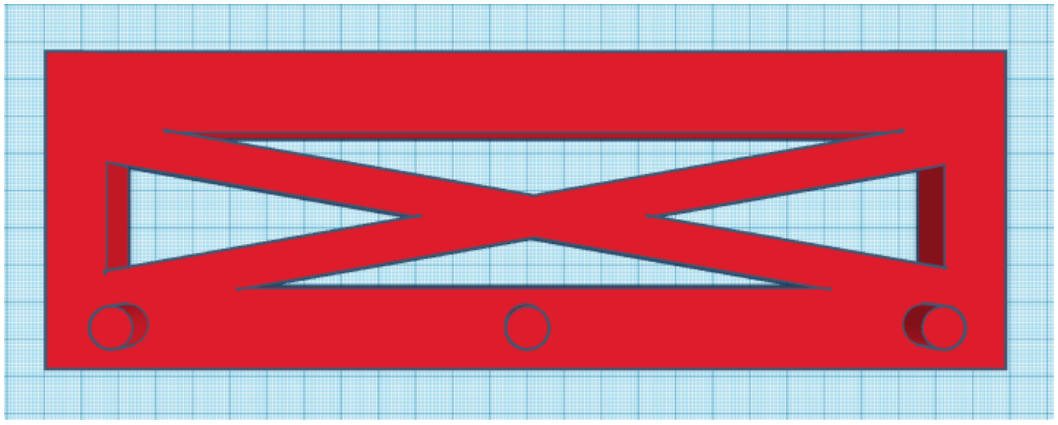
Top View of Y-Joint



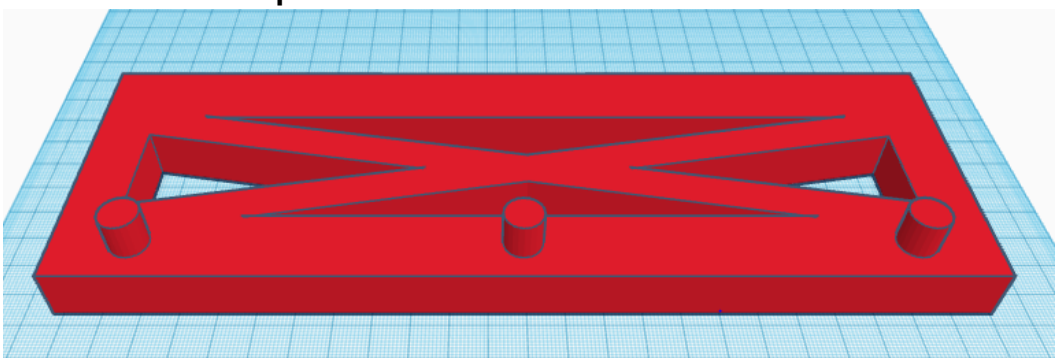
Front Angle View of Y-Joint



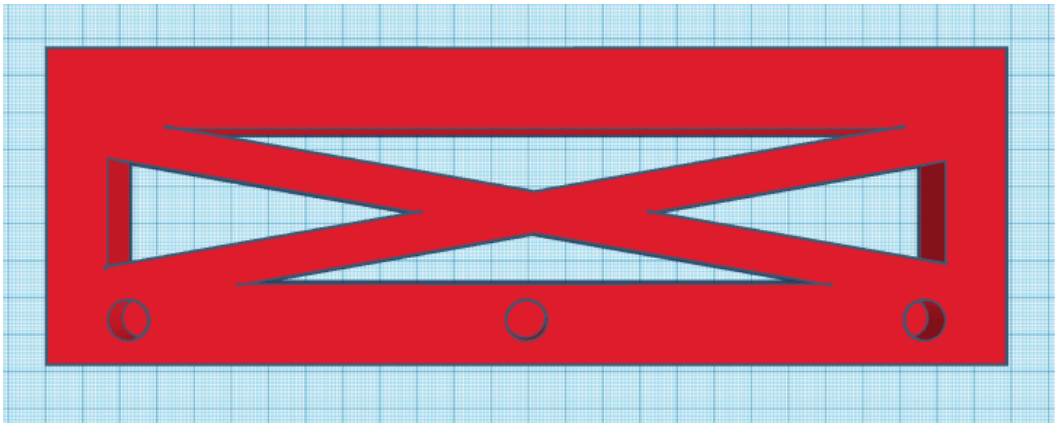
Left: Side View of Female Y-joint
Right: Side View of Male Joint



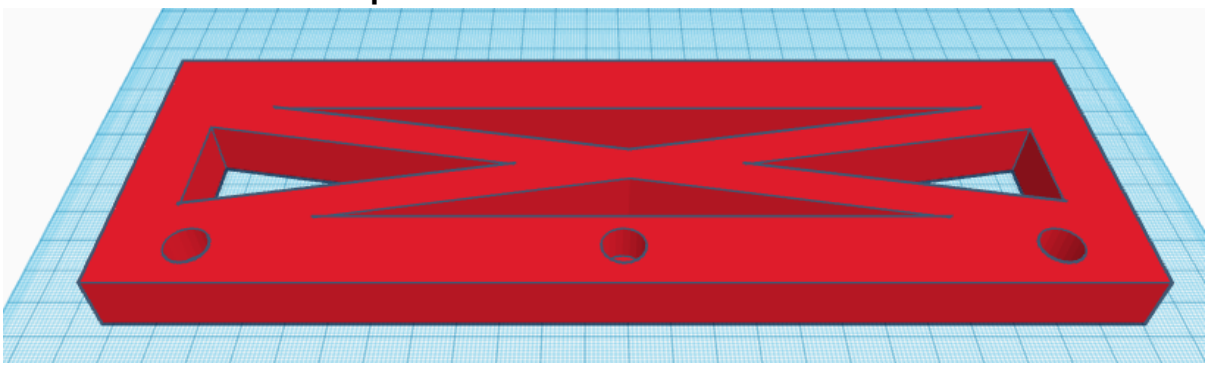
Top View of Female Y-Joint Side Piece



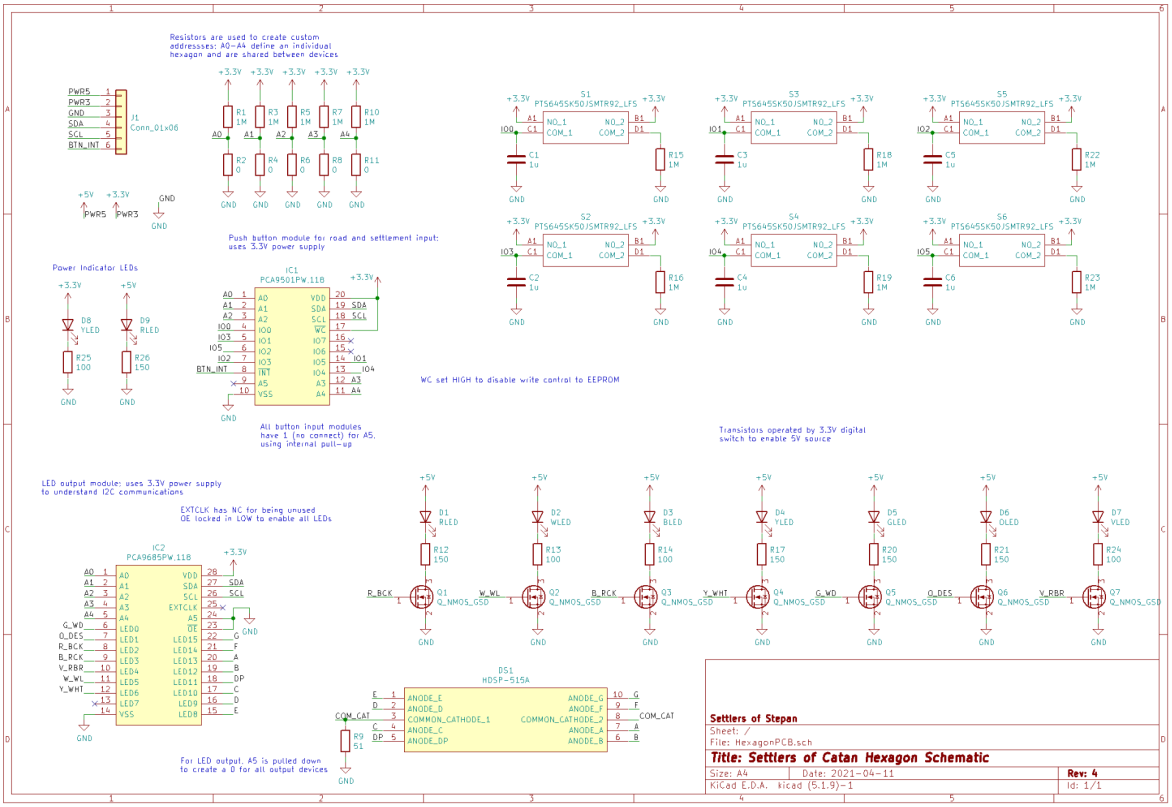
Front Angle View of Female Y-joint Side Piece



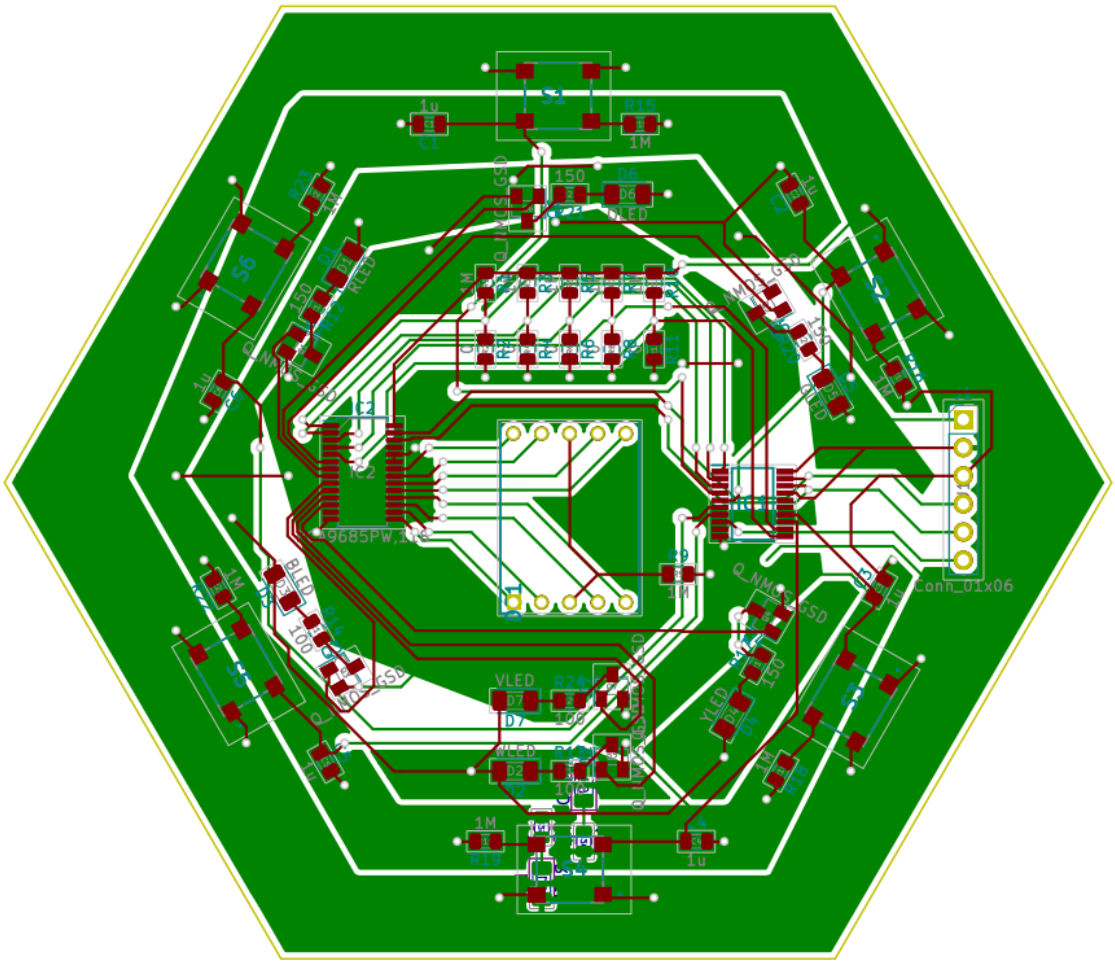
Top View of Male Y-Joint Side Piece



Front Angle View of Male Y-Joint Side Piece



Schematic for the Hexagon PCBs



The PCB Layout for the Hexagons

Software Listings

Below is a listing of all files, with their hierarchy and description. Non-functional files (like images and prototype code) are omitted.

- [debug] - This directory contains accessory files that the console uses to debug the game with.
    - These are all files used to experiment with an ASCII layout of the board. None of them are used in the final product.
- [defaults] - This directory contains default states for the game state and logic that are used to initialize new games.
    - *default_hexes.json*
        - This describes the default parameters for each hexagon in the initial state.
    - *default_state.json*

        - This is the “empty” state file that is contains all of the game information for a new game with no players or items on the board.
    - *developments.json*
        - The standard list of development cards that can be bought in the game.
    - *fill_state.py*
        - This script populates the state file with information from files containing other information about the game state.
    - *harbors.json*
        - This describes the default parameters for each harbor in the initial state.
- [logic] - This directory contains files that implement the entire logic of the game.
    - *build.py*
        - This implements the logic for the build phase of a turn and associated actions.
    - *development.py*
        - This file includes the logic for buying and using development cards.
    - *misc.py*
        - This file includes logic that does not fall into any major category, including shuffling the board for alternate games.
    - *player.py*
        - This file implements all logic involving players, including their addition, removal, and sending notifications.
    - *resources.py*
        - This file includes all logic dealing with game resources, and the first phase of every turn which involves receiving resources and rolling.
    - *trade.py*
        - This file encapsulates logic involving domestic and maritime trading.
    - *vp.py*
        - This file encapsulates logic involving each player’s victory points, and the end of the game.
- [output] - This directory contains files necessary to control hardware.
    - *gpio.py*
        - This file contains functions that deal with button inputs from the hexagon PCBs and relate it to the rest of the game logic.
    - *led_strip.py*
        - This file contains functions that operate the LED strip and receive instructions from the game logic to do so.
    - *process.py*
        - This file weaves together all of the API’s interactions with the hardware, and in concert with the flow of the game logic, ensures that the hardware is communicated with in an efficient way with reasonable timing behavior.
    - *pwm.py*

        - This file contains functions that operate the hexagon PCBs' LEDs and allow the game logic to do so.
    - *seven_segment.py*
        - This files contains functions that operate the hexagon PCBs' 7-segment displays and connect their operation to the rest of the game logic.
- [www] - This directory contains the web GUI and associated scripts, and is the root directory of the web server that the game is hosted on.
    - *404.html*
        - This is a static page to which the user is redirected if they enter an invalid address or try to call a nonexistent API endpoint.
    - *debug.css*
        - This is the CSS stylesheet for the Debug page.
    - *debug.html*
        - This is the Debug page itself, describing its content and structure, including UI elements.
    - *debug.js*
        - This contains the JavaScript code specific to the Debug page, related to UI elements that reside on this page.
    - *game.css*
        - This is the CSS stylesheet for the Game page.
    - *game.html*
        - This is the Game page itself, describing its content and structure, including UI elements.
    - *game.js*
        - This contains the JavaScript code specific to the Game page, related to UI elements that reside on this page.
    - *index.css*
        - This is the CSS stylesheet for the Index page.
    - *index.html*
        - This is the Index page itself, describing its content and structure, including UI elements.
    - *index.js*
        - This contains the JavaScript code specific to the Index page, related to UI elements that reside on this page.
    - *jquery.min.js*
        - This is the jQuery library, which is imported into each web page to enable dynamic control of UI elements.
    - *mithril.js*
        - This is the Mithril JS library, which is used to make and receive HTTP requests and responses to and from the server.
    - *mithril_func.js*
        - This is a file in which resides functions that call API endpoints, and send/receive data to and from the API.
    - *notify.json*

        - This file contains a list of notifications waiting to be sent to the game players.
- *api.py*
    - This file contains the main logic of the API, with a listing of its endpoints and functions, and handles parsing inputs to the API from the outside, as well as sending back data.
- *console.py*
    - This is a Python script which enables the game to be modified and analyzed from the command-line, useful for debugging and testing.
- *print_state.py*
    - This performs a stand-alone function that is also encapsulated in the console--printing out the state of the game in a visual format within the command line.
- *server.py*
    - This is the game server itself, taken from a popular free HTTP server called BaseHTTP, which simply serves files and handles requests and responses. It can run locally or on a LAN, and comes with a number of configuration options.
- *state.json*
    - This is the all-important file that contains the state of the game, on which everything else operates by either reading data from or changing.
- *test_7seg.py*
    - This is a test script that flashes all of the 7-segment display on a particular hexagon PCB, whose address is given as an argument. This was used to test each PCB.
- *test_gpio.py*
    - This is a test script that reads button inputs on a particular hexagon PCB, whose address is given as an argument. This was used to test each PCB.
- *test_leds.py*
    - This is a test script that flashes all of the LEDs on a particular hexagon PCB, whose address is given as an argument. This was used to test each PCB.
- *upload.sh*
    - This is a shell script which is used to quickly and easily update the files on the Raspberry Pi from another computer using rsync.

Components & Datasheets

This section will include links to datasheets of major components of our design. It does not include common components such as individual LEDs, resistors, capacitors, or connection-related components. It also does not include the Raspberry Pi Zero W, which does not have a datasheet.

- NXP Semiconductors PCA9685 -- LED driver on each hexagon PCB
- NXP Semiconductors PCA9501 -- GPIO expander on each hexagon PCB
- Broadcom HDSP-515A -- 7-segment LED display on each hexagon PCB

- ON Semiconductor NDS355AN -- transistor to drive hexagon LEDs
- MEAN WELL RSD-30G-5 -- voltage regulator for 5V power bus
- Qualtek QFWB-65-12-US01 -- power supply
- BTF-LIGHTING WS2812B -- LED strip
    - This datasheet is for each individual LED, not the entire strip. However, the part number is identical.