# Team Cherry Tomato Design Overview

Joseph Miller 504744848
Jorge Hurtado 704595625
Xilai Zhang 804796478
Patrick Chau 404793486
Lab 1B

## Contents

# 1 Introduction

This report provides an overview of team Cherry Tomato's design for a large-scale display system capable of displaying various letters across graduation caps fitted with LEDs. We outline the requirements of our project and cover the motivation behind our design choices, as well as our techniques of implementing them. Additionally, we review the complications we faced throughout the quarter and how we managed to overcome them.

- Allow for 30-80 participants.
- Account for the lack of signal in Pauley Pavilion.
- Account for excess amount of noise with radio and audio communication.
- Noticeable lighting effects within the bright environment of graduation.

# 2 Basic Design

In an effort to meet all the requirements while keeping a relatively low cost, we came up with the following design. Our system consists of client Raspberry Pi Zero W's attached to graduation caps, a hardware token powered by a Raspberry Pi Zero W, and a server running on a Raspberry Pi 4. Each client will be wired to an LED matrix which acts as the display on the top of student's graduation caps. The cap will also hold a button as a means for requesting a position from the server, which will be derived from the hardware token. Our plan of action for the day of graduation will be to pass our token prior to the commencement of the ceremony amongst the students. The token must be incremented by the students as it is passed along as a method of inputting their own position. Each student will press a button to increment the column in the row they are currently in. If they are given the token from someone behind them, they will increment the row they are in instead. Once the token is set to the correct value, which will be displayed by the 7-segment display on the token, the participant will press the button on their cap to initiate a connection to the server. This will allow participants a chance to sync into our server and provide their location info. The 7-segment display additionally serves to display messages regarding whether the hardware is connected and if the pairing is a success. Once the connection succeeds, the token can continue being passed along. Our server allows clients to join the system at any point in time as well as allow for reconnection and recognition of the same client, restoring their prior location and bypassing the need for manually inputting their position again. We broke up the work into four main components, each to be lead by one team member: server (Xilai), client (Jorge), hardware token (Joey), and communications (Patrick).

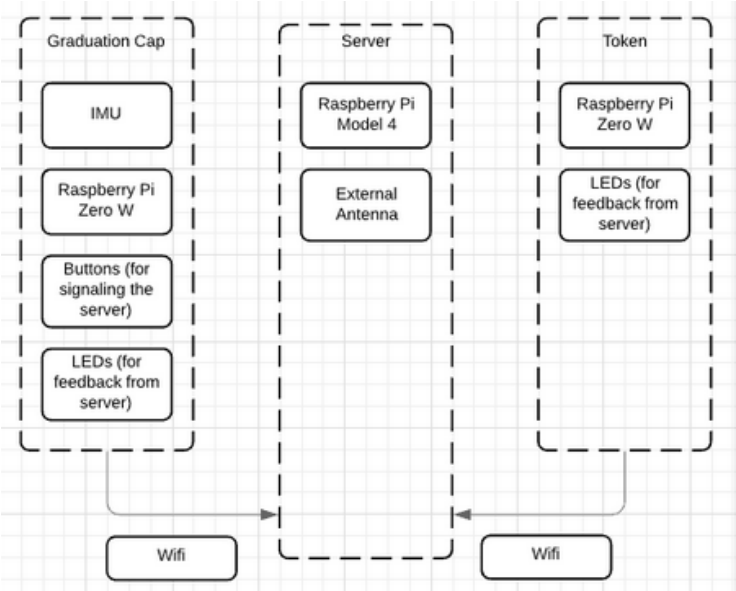


Figure 1: Diagram outlining the overall design for the project. It displays how the individual components communicate together and what physical hardware is needed to implement each device.

# 3 Client (Jorge)

For the first portion of this lab I found that most of my time was spent trying to familiarize myself with the Raspberry Pi interface and available components for the project. Once separated into our groups I was designated to take care of the client. Unfortunately, due to hardware complications our goals for our client were modified throughout the quarter. Our physical outputs were abstracted to print statements which will be replaced once we build our prototype.

I overtook developing the code that would parse out the MAC address of clients added to our system. We opted to use MAC addresses as identifiers for our clients because we realized IP addresses might change. Doing so allowed us to retain the positions of all clients that have joined our server incase they reconnected without the possibility of misidentifying a client. After, verifying my code did parse out the address I setup a practice server client connection. I managed to successfully send my Raspberry Pi's MAC address to my laptop. Upon meeting up with our group, we ended up implementing my function into a client designed for Xilai's server. A snippet of the function code is provided below for reference.

```
def getMAC(interface='wlan0'):
  # Return the MAC address of the specified interface
  try:
    str = open('/sys/class/net/%s/address' %interface).read()
  except:
    str = "00:00:00:00:00:00"
  return str[0:17]
```

Figure 2: Code to obtain MAC Address on Raspberry Pi.

Additionally, I began developing code for g accelerator and a compass function that would use the data collected from Patrick's log to distinguish whether the participant was standing or walking as part of our standalone lighting function. Additionally, Patrick and I managed to do some test runs so we could acquire data from the IMU and begin looking for identifiers for distinguishing whether individuals are walking or sitting. Unfortunately, I did not finish implementing this as we began to focus on meeting the final requirements.

In the final weeks of the quarter, I worked diligently with Patrick to develop code that would allow our clients to reconnect to our server if they were to lose connection after pairing. We developed a function that would continuously attempt to reconnect once losing connection after pairing. While doing so we found that a simple call to connect would not suffice and we had to create a whole new connection to rejoin the system. In order to also ensure we saved the location information of each client we had to go back and modify Xilai's server code as well. Getting this function to work correctly ended up taking more time than initially expected. We still have to go back so that we can handle more exceptions in order to ensure our server doesn't crash during graduation.

# 4 Server (Xilai)

Our design starts with a definition of the client class:

```
class client:
    def __init__(self):
        self.macId=None
        self.ipAddr = None
        self.row=None
        self.col=None
```

This client class is expected to store the client-specific information for each Raspberry Pi that establishes connection with the server. Currently we have the MAC Address which is unique for each client, the optional IP address, and the row and column information to store the physical location of the client, namely, the Raspberry Pi Zero W's.

```
clientList = []
macToClient=defaultdict(client) # map for mac->client
macToId=defaultdict(int)        # mac to client ID
```

Here is a glance of data structures we are using for the project. The clientList stores all the clients that have connected to the server. Based on this list, we can later do sorting to generate patterns. The macToClient is dictionary that maps the mac ID of the client to the Client class. In this way, as long as we know the MAC Address, we can retrieve all the other attributes of the client. The macToId maps the MAC Address to a unique client ID.

```
print_lock=threading.Lock()      # print locks
addLock=threading.Lock()         # add client locks
```

Since we have multiple clients that are trying to communicate with the server and a shared set of structures, we need locks to prevent race conditions from happening. In this project we generate two types of locks. One of them is the print lock. This lock is for situations when we need to log information for debugging purposes. Another lock is the addLock. This lock is the overall lock on the whole set of shared structure. Whenever we wish to make changes on the shared set of data, we need to acquire this lock first.

```
#Hardware token
hw_token=client()
hw_token_socket=None
curr_row = 0
curr_col= 0
```

We treated the hardware token as a client class. We also store the socket id that we have established when the hardware token talks to the server. This allows for us to deliver messages to the hardware token when any individual client thread is successfully added to the server.

```
size=0
id=0
dummyHead=client()
dummyHead.next=dummyHead
dummyHead.prev=dummyHead
tail=dummyHead
startPointer=dummyHead
```

Our design supports two functions. One of them is to generate a pattern similar to the game snake where a snake loops around the caps and grows larger, the other is to display characters such as UCLA. To generate the snake pattern, we used a linked list structure. If we follow the linked list starting at startPointer, we would be able to get a consecutive segment of client objects to generate the snake pattern.

```
lightUp = []   # list of MACs for clients to light up
interval=1
```

The global array lightUp stores the client objects that we want to be lit up. During each iteration, the server will scan through all the client objects and update them. The interval variable dictates the length of the snake. Alternative algorithms will eventually be used to determine which caps would be the best to light with a particular pattern and this array will allow for an easy way to signal to every cap when their turn to be lit has arrived.

```
def runServer():
    host="192.168.43.190"
    serv=socket.socket(socket.AF_INET,socket.SOCK_STREAM)
    serv.bind((host,8080))
    serv.listen(10) #in case it takes time to acquire lock

    t1 = threading.Thread(target=lookForConnection, args=(serv,))
    t2 = threading.Thread(target=loopThrough, )

    t1.start()
    t2.start()
```

We will begin our server with two threads, one of them goes to the function lookForConnection, which checks whether there are clients that want to establish a connection with the server. The other thread goes to the function loopThrough, which updates information in the lightUp array to indicate whether a specific client class should be lit up.

```
def lookForConnection(serv):
    while True:
        c,addr=serv.accept()
        print_lock.acquire()
        print("connection established")
        print_lock.release()
        start_new_thread(each_client,(c,))
    serv.close()
```

In the lookForConnection function, we further split the program into multiple threads, namely, one thread for each connected client. Each connected client will be directed to the function each_client, which creates a unique thread for said client. We pass the client socket id to the each_client function as well. Along the way, we acquire print lock to log information during the process.

## 4.1 EachClient Function

```
macId=None
while True:

    buffer=None
    c.settimeout(10000) #timeout 10 secs for now,client need to constantly send msg
    try:
        buffer=c.recv(4096).decode('utf-8')
        #print("received message: " + buffer)
```

We start with an empty macId for the thread. Since variables in each thread are defined locally, we can check the status of the current thread by checking the macId variable. If the macId is set, it means we have processed this thread before. If not, the thread is yet to be initialized. In a try loop, we check for response from the client. If we haven't heard back from the client for more than 10 seconds, we assume the client is no longer connected to the server.

```
except IOError or socket.timeout:  # timeout error catch
    print_lock.acquire()
    print("time out occured on ",c)
    print_lock.release()


            if macId:
                target=macToClient[macId]
                addLock.acquire()
                target.prev.next=target.nex
                target.next.prev=target.pre
                del macToClient[macId]
                size-=1

                lightUp.remove(macId)
                addLock.release()

            return #we are out of here
```

In the event of a timeout, we log the timeout message and close the connection. We remove the information of the timed out client from our data structure. Along the way, we will have to acquire the add Lock.

```
try:
    _type,msg=buffer.split(",") # assume client sends in this format
```

In the event that the server does not timeout, we will try to interpret the messages sent from the client. We assume the format of the client message is the type of message, followed by a comma, and followed by the content of the message. If the type of message is start, we create a new client object for the client and add it to our shared data structure. If the type is hwstart, we set up the global variable associated with the hardware token. if type is close, we do clean up for the client. It is important that information is kept in the shared structure so that client can reconnect later. If type of message is position, we will document row and column information for the client in our shared data structure across threads.

# 5 Communications (Patrick)

Our initial consideration is Wifi. It's very easy to work within our framework and we can easily setup connections for all clients to the other server. The usage of Wifi is also preferable given the Raspberry Pis have Wifi cards built in and can even be configured as an access point so this saves somewhat on cost. The specified range for Wifi for our Raspberry Pi Zero W is up to 100m although it should be expected that the range indoors will be much lower. This can be remedied with the purchase of an external antenna but we'll ideally only need 1 for the designated access point. The con here is that we'll have a single access point and many clients all within close proximity to one another, which will lead to significant signal interference. How much is yet to be seen but without knowing more about the conditions, it's hard to say if it'll be a significant problem.

Bluetooth is also considered. Bluetooth transfers over the same frequency band as Wifi. Its strength mainly lies in communication between two points, with expansion up to 7 clients in a master-slave setup. By having a system like these, all the devices are automatically synced to the master's clock, which is very useful. However we desire a system that will be communicating with many clients so while it will work for a simple prototype, there would be no way to scale up using Bluetooth. Additionally the Raspberry Pi doesn't natively support a Bluetooth communication so a shield for the Pi would be necessary, incurring an additional cost. Finally the range matter of In the end, it's determined that the strengths of Bluetooth don't fully align with our goals for this system.

Zigbee is a radio based communication protocol based on the IEEE 802.15.4 specification. It has an upper limit of 100m line-of-sight. Zigbees can be attached to each Raspberry Pi and they would communicate . Their rated over-the-air communication link is 250MB/s which is more than enough for the small data packets that we're planning on sending over our network as detailed in our server-client descriptions. Zigbees have low power consumption, which aids in the long runtime that we're planning on having over the course of the ceremony. The Zigbee architecture requires 3 components - a coordinator, a router and the end devices. This would require extra hardware as these coordinator nodes are separate large components which likely won't last long on a battery pack. Zig-

bee, if the Wifi signal interference proves to be too great, would be our backup communication protocol.

In the end, Wifi was chosen as our desired communication protocol. Wifi was chosen for two main reasons: ease of use and cost effectiveness. To use something like Zigbee would require refitting all the Raspberry Pis with new receiver hardware whereas each Pi already comes pre-equipped with a Wifi chip. Also the ease of using Wifi with the Python framework makes it very attractive for getting a quick prototype up and running. Using TCP/IP we set up a socket based communication between the clients and the server. The clients have no need to communicate between each other, instead placing all the processing load on the server. This decision motivated the need for a better server processing unit. Perhaps if seeking pure robustness in our system and we had ample amounts of time, we would've opted for a Zigbee based network but Wifi has proven to be robust enough for our application that we're experiencing any detriment in performance.

# 6 Hardware Token (Joey)

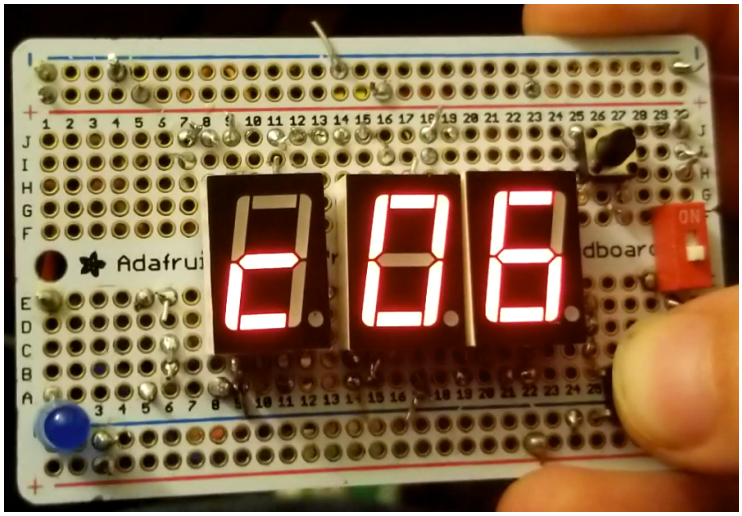


Figure 3: Example of hardware token feedback for inputs. This shows that the user is currently at column 6. By flipping the switch on the right it will swap to displaying the current row position.

The Hardware token itself is a self contained device that allows for both inputs from the user regarding position information and feedback to the user about system status. It will display if it is connecting to the server, if it has successfully added the user as well as the currently inputted row and column number. The seven segment display is capable of outputting messages that will scroll across the screen. One example is when the token is attempting to connect to the server it will display "connecting" across the display.

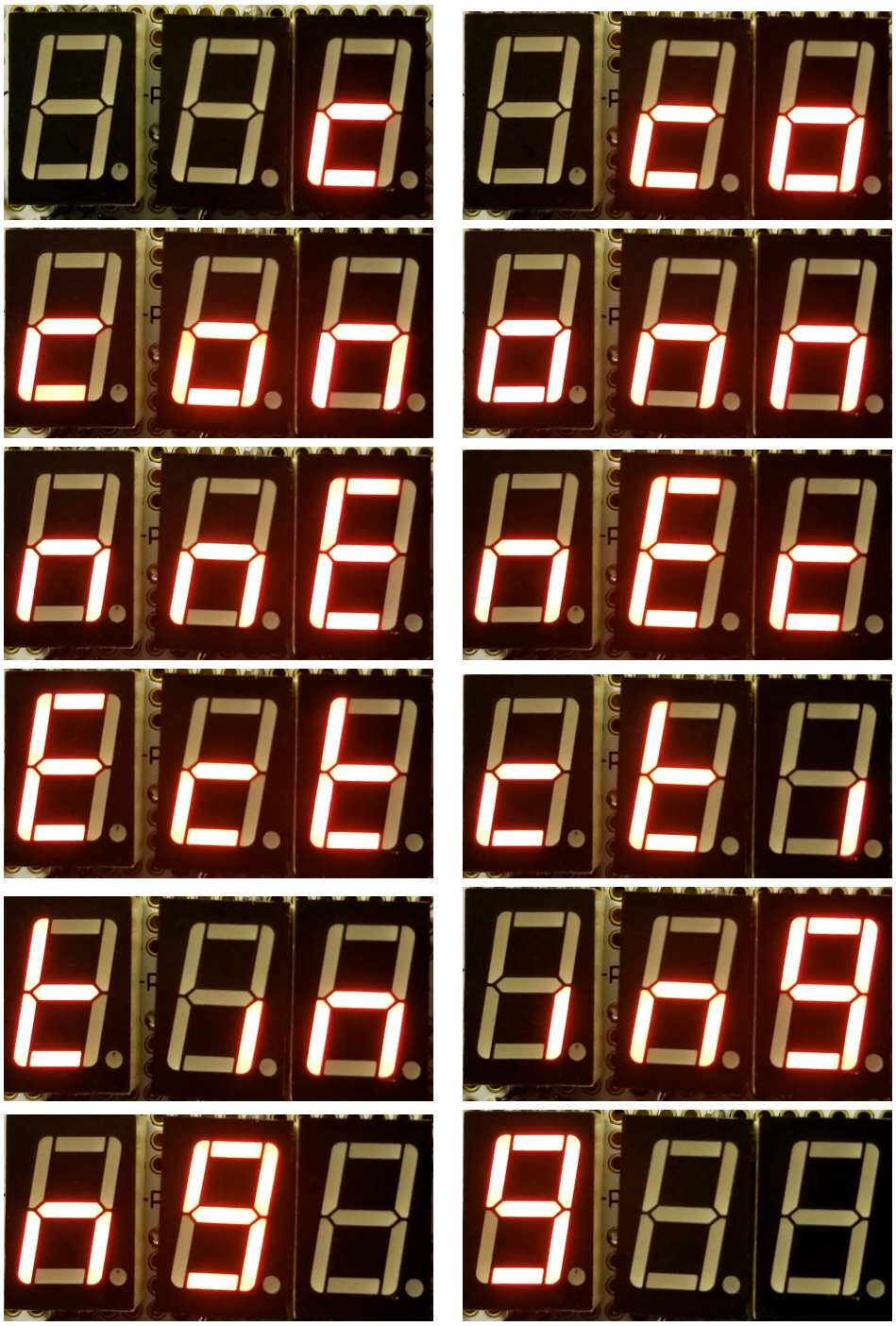

Figure 4: When read left to right top to bottom, these are the individual frames for the word connecting being displayed across the 7 -segment display.

## 6.1 Hardware

The hardware token, as the name implies, is made from hardware. The following parts list and wiring diagram explains how the system can be built in the event that one wishes to use one. In terms of circuit design, there are common cathode 7-segment displays that are connected to a current limiting resistor and the GPIO pins on the Raspberry Pi Zero W. There are also two capacitive debouncing circuits to allow for smooth voltage transitions on the button presses preventing multiple inputs from a single press. Finally there is an LED connected directly to a GPIO pin to a current limiting resistor and ground. This LED was originally to be used for indicating if the system has successfully added a user, but currently that role is taken by the displaying the string "success" across the 7-segment display. The LED will likely instead be used as a way to signal if the token's position info has been modified since the system added a prior user. If no use is found it will be removed from the final design.

### 6.1.1 Parts List

If one wishes to obtain the parts to purchase and assemble their own hardware token the following list can be used.

- 1 x Raspberry Pi Zero W
- 3 x Common Cathode 7 segment displays
- 2 x 1$\mu$F capacitors
- 2 x Push Buttons
- 1 LED
- 1 x Toggle Switch
- 5 x 1k$\Omega$ Resistors and 3 x 10k$\Omega$ Resistors

### 6.1.2 Circuit Schematic

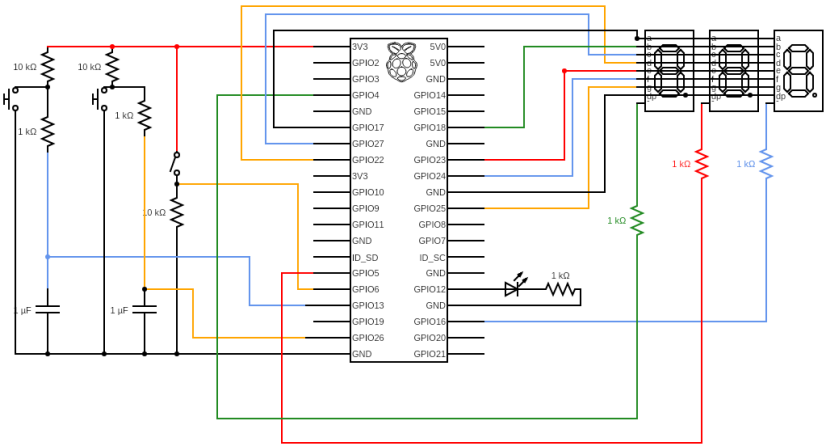


Figure 5: Circuit schematic used for wiring the hardware token.

## 6.2 Code

The hardware token was written in C and has it's own makefile to allow for ease in compilation. It is made up of a total of 8 files in addition to the makefile: hardware.c, hardware.h, globals.h, hardware_token.c, network.c, network.h, signal_handler.c, and signal_handler.h. The header files contain function declarations and global variables. All function declarations are commented to show what inputs/outputs are involved in the function as well as what the function is supposed to do.

```
46  /*****************************************************
47   * Function: attempt_connection
48   *----------------------------------------------------
49   * Attempts to generate a socket fd to the server
50   *----------------------------------------------------
51   * Return Values:
52   * SUCCESSFUL_CONN is returned when connection succeeds
53   * FAILED_CONN is returned when connection fails
54   * ---------------------------------------------------
55   * Inputs:
56   * sockfd will be set to the correct value for the
57   * network file descriptor on success. On fail will be
58   * set to NULL.
59   *****************************************************/
60  int attempt_connection(int* sockfd);
```

Figure 6: Example of comment info above function declarations. This is an example for what the attempt_connection function in networks.h does.

All POSIX complient function calls that are made have proper error handling associated with them to ensure that the proper error code corresponding to errorno is printed to STDERR along with a message describing what went wrong.

```
long sockfd_flags = fcntl(attempted_sockfd, F_GETFL, NULL);
if(sockfd_flags < 0) {
    fprintf(stderr, "Error obtaining sockfd flags: %s\n", strerror(errno));
    return FAILED_CONN;
}
```

Figure 7: Code showing error handling on POSIX function calls.

### 6.2.1 Makefile

```
CC = gcc
CFLAGS = -Wall -Wextra
LDLIBS = -pthread -lwiringPi
OPS = -g -o
OBJFILES = hardware_token.o hardware.o network.o signal_handler.o
TARGET = hardware_token

all: $(OBJFILES)
    $(CC) $(CFLAGS) $(OPS) $(TARGET) $(OBJFILES) $(LDLIBS)

clean:
    rm -f $(TARGET) $(OBJFILES)
```

Figure 8: Code describing the Makefile.

The Makefile allows for an easy way to continue adding new files to the project and facilitate cleanup and compilation. It supports a make all option that makes the whole project and a clean option that deletes the object files produced.

### 6.2.2 Globals.h

This file contains globals that aren't unique to a type of file. These include delay paramaters that multiple functions use as well as modifiable global variables. The modifiable globals include a pthread mutex for concurrency related problems, row and column paramaters, messages to display on the segments and booleans used to determine when the program has finished, if the server is connected and whether or not there is a message to display.

```
/*************************************************
 * Global Constants
*************************************************/

// Magic Numbers
#define FOREVER 1
#define usec_delay 1
#define ten_usec_delay 10
#define hun_usec_delay 100
#define msec_delay 1
#define ten_msec_delay 10
#define hun_msec_delay 100
#define sec_delay 1000
```

Figure 9: Code describing the Makefile.

```
/*************************************************
 * Modifiable Globals
*************************************************/

// Threads
extern pthread_mutex_t lock;
extern int program_end;

// Rows and Cols
extern int row_pos_ones;
extern int row_pos_tens;
extern int col_pos_ones;
extern int col_pos_tens;

// Server
extern int server_connected;
extern int msg_to_display;
extern char display_msg[20];
```

Figure 10: Code describing the Makefile.

### 6.2.3 Hardware Token.c

This is the main file for the project that ties everything together. In here both of the pthreads are generated and used to start displaying information on the hardware token's display and begin communication with the server.

```
// Init GPIO pin mappings and set signal handlers.
init_pins();
set_sig_handlers();

// Init threads
pthread_t network_thread = 0;
pthread_t hardware_thread = 1;

// Init mutex
pthread_mutex_init(&lock, NULL);

// Run the display
pthread_create(&hardware_thread, NULL, run_display, NULL);

// Attempt communication with server
int sockfd;
while(attempt_connection(&sockfd) && !program_end);

// Communicate with server
pthread_create(&network_thread, NULL, server_communication, &sockfd);

// Wait for threads to finish
pthread_join(hardware_thread, NULL);
pthread_join(network_thread, NULL);

// Close threads
pthread_exit(NULL);

// Close the socket
close(sockfd);
```

Figure 11: Code describing the Makefile.

### 6.2.4 Hardware.c and .h

These files contain the global constants and functions needed to control the GPIO pins corresponding to physical hardware on the Pi. For brevity code snippets will be kept short in this section and will be used sparingly to show more complex implementations of certain functions.

Two easy to get out of the way functions in this file include init_pins and clear_pins. These two do as one would expect and set the wiringPi GPIO pins to their proper start states and clear them all to zero respectively.

The real meat of this file includes the hardware thread function run_display. This function runs while the program hasn't been set to the exit state by the signal handler. Once it is set to exit this thread and the network thread return and all data is properly cleaned up in the main function. If the hardware token is not connected to the server, it will continue to call the display_connecting function that causes the 7 segment display to scroll the string "connecting" across its screen. If the hardware token is connected to the server it instead runs in the main display loop. It first checks if any message is ready to display and displays it if there is. It then reads inputs from the GPIO pins connected to the buttons. If the switch is set to rows then the code for button presses will modify the row variables and if it is set to cols then the column variables will be modified. It simply runs checks to make sure the 1's place overflows at 9 to 0 and 0 underflows to 9. when an underflow occurs the 10's place is modified so that it increments or decrements with similar under/overflow logic. Once this section completes it is time to blink the display. the blink_segment function is called translating the integers to 7 segment bitmaps via the integer_to_display function and either 'r' or 'c' to the bitmap through the character_to_display function.

```
unsigned char integer_to_display(int num_to_display) {

    switch(num_to_display) {
        case 0:
            return 0x3F;
            break;
        case 1:
            return 0x06;
            break;
        case 2:
            return 0x5B;
            break;
        case 3:
            return 0x4F;
            break;
```

Figure 12: How integers are converted to 7 deg display outputs

```
unsigned char character_to_display(char char_to_display) {

    switch(char_to_display) {
        case 'r':
            return 0x50;
            break;
        case 'c':
            return 0x58;
            break;
        case 'o':
            return 0x5C;
            break;
        case 'n':
            return 0x54;
            break;
```

Figure 13: How characters are converted to 7 deg display outputs

```
void blink_segment(const int seg, const unsigned char display_char, int time_delay) {

    const unsigned char mask = 0x01;

    // In case pin mappings are changed.
    unsigned int led[NUM_SEGS] = {A,B,C,D,E,F,G};

    // Turn Segment on (Common Cathode display).
    digitalWrite(seg, LOW);

    for (int i = 0; i < NUM_SEGS; i++) {
        unsigned char tmp = (display_char >> i) & mask;
        if(tmp == 0x01) {
            //Blink this LED
            digitalWrite(led[i], HIGH);
            delayMicroseconds(time_delay);
            digitalWrite(led[i], LOW);
        } else {
            //Leave LED off
            delayMicroseconds(time_delay);
            digitalWrite(led[i], LOW);
        }
    }

    // Turn Segment off (Common Cathode display).
    digitalWrite(seg, HIGH);
}
```

Figure 14: A mask is used to determine if the given segment is currently on or off. This is then used to blink the segment on for a small time then off again allowing the illusion of all displays being on at once.

The final function in these files worth noting is the display_message function.

This simply takes in some character array and blinks each segment through the blink_segment function until it completes a full loop. While it is running no inputs from the buttons can affect the display since it would be problematic to have users affect the values they cannot see.

### 6.2.5 Networks.c and .h

As with the hardware code, code snippets will be used sparingly for brevity as the well over two thousand lines of code in these files would be a bit extensive to go over in great detail. These files have two functions. server_communication and attempt_connection. The first of the two is the function the network thread runs through and continues until the program ends. Similar to the hardware thread if a SIGINT or SIGTSTP occur the individual threads return and the program cleans up before exiting. The thread begins by running the following snippet to find the MAC Address of the token. Once it obtains it, it formats a message to deliver to the server:

```c
// Strings
char msg[BUFFER_MAX];
char MAC_ADDR_PATH[BUFFER_MAX];
char MAC_ADDR[18];
char interface[6] = "wlan0";

// Zero out buffers
bzero(msg, sizeof(msg));
bzero(MAC_ADDR_PATH, sizeof(BUFFER_MAX));
bzero(MAC_ADDR, sizeof(MAC_ADDR));

// Write filepath to file containing MAC ADDR
sprintf(MAC_ADDR_PATH, "/sys/class/net/%s/address", interface);

// Open file containing MAC ADDR
int MACfd;
MACfd = open(MAC_ADDR_PATH, O_RDONLY);
if(MACfd != -1) {
    // Read the 17 characters corresponding to MAC ADDR
    read(MACfd, MAC_ADDR, sizeof(char)*17);
} else {
    // Use a hopefully unique identifier for MAC ADDR
    sprintf(MAC_ADDR, "83:13:71:76:13:98");
}

// Write filepath to file containing MAC ADDR
sprintf(msg, "hwstart,%s", MAC_ADDR);
fprintf(stdout, "Token sent: %s\n", msg);

// Deliver MAC_ADDR to server
write(sockfd, msg, sizeof(msg));
```

Figure 15: Code that obtains MAC ADDR. The comments sum up how it works.

It then while the server is connected will continue to read data from the

server and send it current position and column data to keep the connection alive. Once it receives a message back from the server indicating someone has been added, it signals to the hardware thread that it is time to display "success" on the seven segment display. If the token is not connected to the server, it calls attempt_connection.

While the server is attempting to connect to the server, it continues until the program ends to first generate a socket to the server using its IP address, set the socket to nonblocking and use connect in conjunction with select to obtain a timeout of exactly 2 seconds before reattempting communication. This allows for speedy program closes and reconnections if the server ever crashes. The default timeout on connect is way to long to allow for a speedy system. Once the server properly connects, the socket is set to blocking once again and the function returns SUCCESSFUL_CONN (A 1 in the code).

### 6.2.6 Signal Handler.c and .h

These files handle the signal handling needed for the hardware token to work properly without crashing. The need for these functions is mostly due to the SIGPIPE signal. When the TCP socket connecting the token and the server is severed, a SIGPIPE is delivered from the OS to the program and if it isn't handled, the program is immediately stopped. Whenever the SIGPIPE comes through, the signal handler sets a global variable signaling to the hardware and network threads that the token is no longer connected. This causes connecting to display on the token while the network thread gets to work generating a new socket to attempt reconnection. SIGTSTP and SIGINT are two other signals handled by the signal handler. These are important for the program to properly exit while testing. This allows for all LEDs to properly shut off and network communication to successfully close out.

```
if (signal(SIGINT, sigterm_handler) == SIG_ERR) {
    fprintf(stderr, "signal failed with error: %s\n", strerror(errno));
}

if (signal(SIGTSTP, sigtstp_handler) == SIG_ERR) {
    fprintf(stderr, "signal failed with error: %s\n", strerror(errno));
}

if (signal(SIGPIPE, sigpipe_handler) == SIG_ERR) {
    fprintf(stderr, "signal failed with error: %s\n", strerror(errno));
}
```

Figure 16: Code used to set signal handlers for various signals.

## 6.3 Future Plans

The hardware token is still not 100% complete. There are a few small touches that will make the token more presentable, intuitive and robust.

### 6.3.1 3D Printed Case

A 3D printed case will allow for printed instructions to remind users how to use the system as well as make the increment and decrement buttons more clear to the user. It also allows all the internals to be hidden away from the user and provides padding in case the device is dropped during the graduation ceremony. It also allows for the token to have a more aesthetically pleasing design giving it a much more professional look and feel.

### 6.3.2 PCB

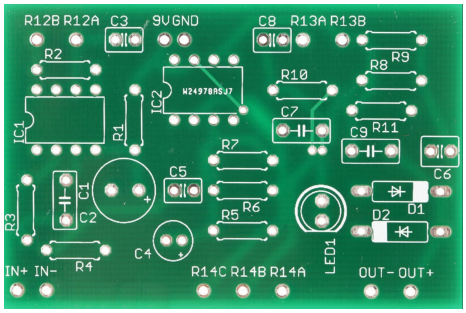


Figure 17: Sample printed circuit board that shows how easily components could be soldered together without worrying about cutting and soldering connecting wires.

Having a printed circuit board for soldering the components for the hardware token will significantly reduce the difficulty in connecting all of the components together and allow for a more compact design. This will make development of a 3D printed case easier and allow for case designs to be more compact as well.

### 6.3.3 Using RFID to Initialize Position Data


Figure 18: RFID reader that is currently planned to be integrated as a means of connecting oneself to the server.

Currently the system supports having each individual cap having a button that will be pressed when someone wishes to tell the server to poll the hardware token and input their position into the system. This leads to a few issues involving people accidentally inputting themselves into the system when they don't have the hardware token. These issues can easily be circumvented by use of RFID integrated into the client caps that reads from the hardware token to decide whether to transmit or not. Since RFID is only accurate up to a few centimeters, there would be no way to false input oneself into the system. All that would be required from a user would be to pull the hardware token up to the graduation cap to send one's current position to the server. RFID can be obtained very cheaply on Amazon, a pack of 5 readers and tokens run for a little over $10.

# 7 Conclusion

| | **Projected** | **Actual** |
|---|---|---|
| W6 | Order components | Ordered components |
| W8 | Working prototype hat w/ LEDs<br>Successful communication between client and server | Set up server as access point<br>Prototype hardware token |
| W9 | Integrate hat and wireless communication<br>Determine mounting for hardware | Had 3 clients communicating to server at same time |
| W10 | Finalize final presentation and prototype demo | Had hardware token & 3 clients communicating to server with token passing position of clients to server |
| W11 | Final Presentation | Final presentation |

Figure 19: Chart of the expected vs. actual timeline we followed this quarter.

Overall the project this quarter has gone successfully this quarter. While we were unable to meet our original timeline exactly, despite the complications, we managed to follow it pretty closely and adapted our goals according to the setbacks that arose (mostly related to the lack of hardware for the caps). This allowed us to continue moving forward throughout the quarter and ensured everyone always had something to work on. Despite the setbacks we faced throughout the quarter we managed to adhere closely to our initial timeline. Although a challenging project, we were able to divide the work equally amongst our group. We have learned to work as an effective unit, not afraid to ask one another for help when we need it.