



## PROGRESS REPORT FOR ANDREW GAN

### WEEK 1:

**Date:** Aug 28, 2020

**Total hours:** 3

**Description of design efforts:**

Discussed project ideas with team regarding the selection of hardware, API libraries, and task delegation among team members. Volunteered to be responsible for SD card filesystem management and PCB design.

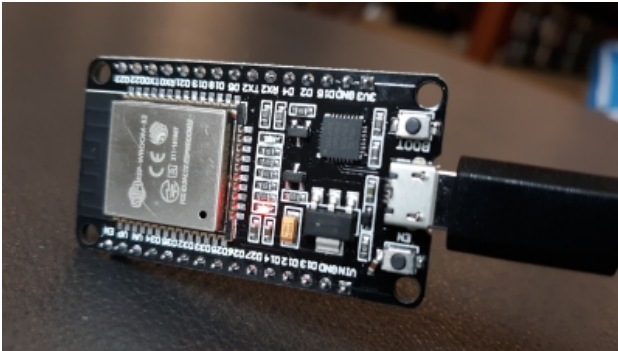
### WEEK 2:

**Date:** Sep 4, 2020

**Total hours:** 8

**Description of design efforts:**

Installed the ESP-IDF development toolchain and successfully built and run a hello world program.



Studied the ESP32 technical reference manual and FAT filesystem interfacing on an ESP32. [link](#)

Research on a FatFs API module that handles the layer between physical disk I/O operations and device driver support. [link](#)

Decided on the MicroSD card, SD card reader and PCB design tools to use for the project.

Written the prototype code for registering the OLED and SD card reader onto the HSPI interface.

```
I (317) cpu_start: Starting scheduler on PRO CPU.  
I (0) cpu_start: Starting scheduler on APP CPU.  
I (328) HSPI: Successfully initialized bus.  
I (1328) OLED: Successfully registered on SPI bus.
```

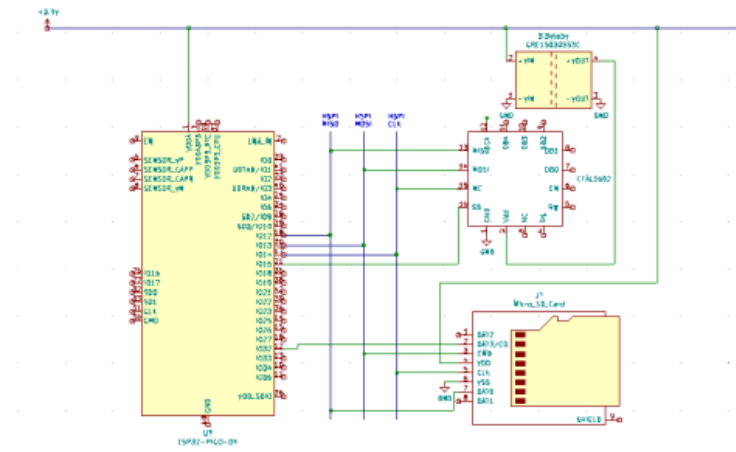
### WEEK 3:

**Date:** Sep 11, 2020

**Total hours:** 10

**Description of design efforts:**

As the hardware engineer of the team, I used Kicad to illustrate a simple HSPI bus connection. Kicad includes tools to design a schematic, as well as a PCB layout that accounts for physical constraints. This allows the team to quickly assemble the components together once the ESP32 and other hardware items arrive. The ESP32 has 40 GPIO pins. However, only certain pins provide the functionality needed for the project such as UART, SPI connection, etc. Therefore, a circuit diagram also allows the team members to reserve the GPIO pins to be used for the components they are individually responsible for, so that there will be a lower chance of pin conflict. Kicad is a free and open source software tool that is preferred over proprietary licensed tools such as Autodesk Eagle.



I also guided my team on how to install the ESP-IDF development toolkit through the Visual Studio Code extensions. This ensures every team member will be developing code for the project on the same platform which prevents platform-specific code from causing errors when different components are put together in the end.

In order to test the SPI functionality of the ESP32, I wrote a piece of code to transmit SPI commands to slave devices, in this case an OLED CFAL 1602C module. There were technical complications that were faced while attempting to do this. Firstly, the OLED module processes 10-bit instruction codes, while the SPI transmits bits in multiples of 8 bits or 1 byte. Therefore, a built-in macro function in the ESP32's SPI library called SPI\_SWAP\_DATA\_TX had to be used to include necessary padding and convert it to an unsigned format that is readily transmittable by the SPI function spi\_device\_polling\_transmit. Due to the inability to obtain the expected results, possibly due to incorrect usage of the SPI driver of the ESP32 or the incorrect connection of pins, an Analog Discovery 2 was used to debug the SPI pins to resolve the problem.

```
I (2328) HSPI: Successfully transmitted 0x00000001, returning 0x00.
I (2328) HSPI: Successfully transmitted 0x00000254, returning 0x00.
I (2328) HSPI: Successfully transmitted 0x00000268, returning 0x00.
I (2328) HSPI: Successfully transmitted 0x00000269, returning 0x00.
I (2338) HSPI: Successfully transmitted 0x00000273, returning 0x00.
I (2348) HSPI: Successfully transmitted 0x00000220, returning 0x00.
I (2358) HSPI: Successfully transmitted 0x00000269, returning 0x00.
I (2358) HSPI: Successfully transmitted 0x00000273, returning 0x00.
I (2368) HSPI: Successfully transmitted 0x00000220, returning 0x00.
I (2378) HSPI: Successfully transmitted 0x00000261, returning 0x00.
I (2378) HSPI: Successfully transmitted 0x00000220, returning 0x00.
I (2388) HSPI: Successfully transmitted 0x00000274, returning 0x00.
I (2398) HSPI: Successfully transmitted 0x00000265, returning 0x00.
I (2398) HSPI: Successfully transmitted 0x00000273, returning 0x00.
I (2408) HSPI: Successfully transmitted 0x00000274, returning 0x00.
```

These milestones were important as they allowed me to be familiar with the ESP32's SPI. The same interface will be used for communicating with the SD card reader Adafruit 254 and the fingerprint scanner. Once the rest of the hardware components arrive, multiple components may be connected to the same SPI bus, with each having its own SS connection to the ESP32, allowing for easier management of the slave devices from the ESP32. I have learned how to better read the ESP32 documentation as well, and where to find the write function to call. It is expected that the SPI will be able to send the correct commands to the OLED module in time for the SMAT next week. The next objectives would include a preview of the complete Kicad schematic including all the peripherals, and the implementation of the SD card filesystem basic read and write operations. I assisted my teammates to get the ESP-IDF development environment set up.

#### WEEK 4:

**Date:** Sep 18, 2020

**Total hours:** 10

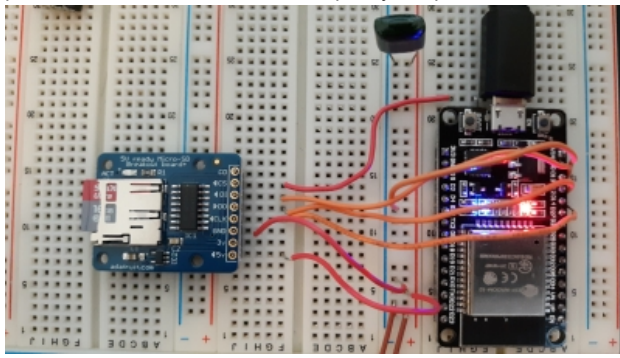
#### Description of design efforts:

One of the goals of this week was to get the OLED module up and running. However, the OLED module did not display any text when connected to the ESP32 via SPI. In order to figure out what went wrong, I requested an ADALM2000 from the lab. The ADALM2000 is a tool that provides features such as oscilloscope, I/O LED, voltmeter, and logic analyzer. After connecting the ADALM2000 to the ESP32, I noticed that while the signals are transmitted correctly, a logic high is represented with a -3.3 V output instead of the more conventional +3.3 V output. CH1 (orange) connected to MOSI, and CH2 (purple) connected to CS. I later realized the ground wire was incorrectly connected and the oscilloscope displayed +3.3 V after the correction. The SPI was tested to be functioning as expected.



Another confusion that arose was that the SPI pin configuration of the ESP32 technical reference manual and the SD SPI example code were different. The reference manual uses GPIO12 for MISO, GPIO13 for MOSI and GPIO14 for CLK while the example code configures the SPI as

GPIO2 for MISO, GPIO15 for MOSI and GPIO14 for CLK. They both worked, which meant the SPI functionality was not restricted to those few pins. All that had to be done was specify the pin numbers correctly during the SPI bus initialization phase in the software.



The SD card was successfully initialized and mounted, and basic file system operations such as creating and writing to a file were possible. Eventhough the OLED module was not needed for the project, it was useful in testing out the SPI to ensure my code works, and to familiarize myself with the ADALM2000 and its corresponding visualization software, Scopy. This powerful tool will be very useful for debugging other interfaces as well, such as UART. With the SD card module operational, I can now write high-level functions that are tailored to the functionality of the project such as searching for the right file given a website id and returning the file which contains the encrypted information. File indexing speed and stability will be important metrics.

```
! (334) cpu_start: Starting scheduler on PRO CPU.
! (0) cpu_start: Starting scheduler on APP CPU.
! (346) HSPI: Successfully initialized bus.
! (3346) gpio: GPIO[15] InputEn: 0 OutputEn: 1 OpenDrain: 0 Pullup: 0 Pulldown: 0 Intr:0
! (3346) sdspi transaction: cmd=52, R1 response: command not supported
! (3386) sdspi transaction: cmd=5, R1 response: command not supported
! (3426) SD-CARD: Successfully mounted SD card.
Name: SC16G
Type: SDHC/SDXC
Speed: 20 MHz
Size: 15193MB
! (3426) SD-CARD: Opening file
! (3436) SD-CARD: File written
```

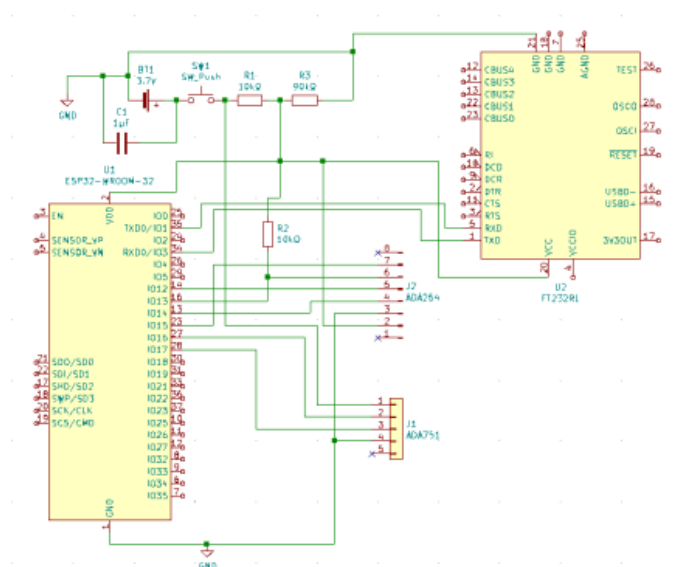
The next steps would be to fully design the first prototype of the PCB, which will include the pull-up resistor for the MOSI pin of the SD SPI, an SD reader, a QFN ESP32 chip, a fingerprint scanner, resistors, capacitors and battery holder. Almost all of the parts will be surface-mounted to make it easier to attach components to the board when it arrives. Another thing to do would be to test and see if the current supplied by the battery will be adequate for all the components in the system.

## WEEK 5:

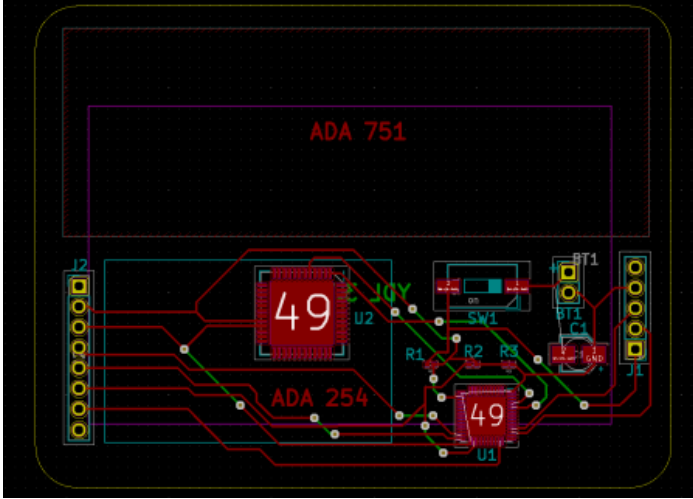
**Date:** Sep 25, 2020

**Total hours:** 9

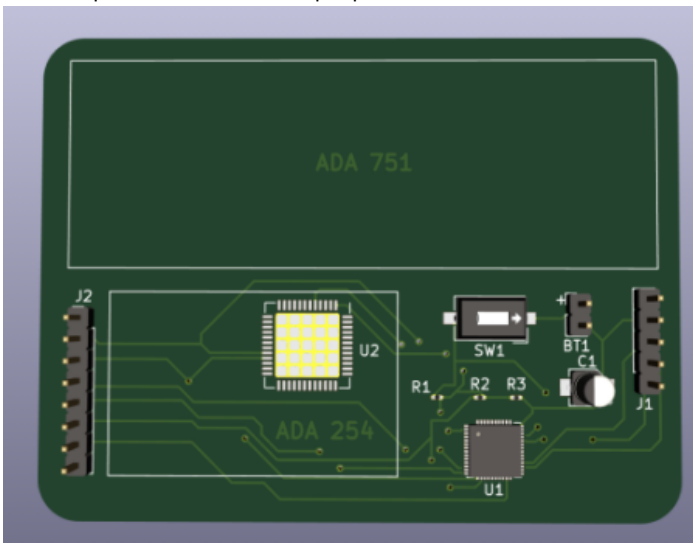
**Description of design efforts:**



This week, I started designing the PCB using KiCAD, which was preferred over Autodesk Eagle due to its open source nature and no need to deal with licensing issues. I surveyed the part numbers of the hardware to be used in our project and added them into the PCB schematic. After that, I had to match the part symbols with their respective footprints so that the size taken up by each piece of hardware can be used to accurately determine the size of the PCB. Placing a development board on the PCB is possible but would result in the PCB being too large. Therefore, for the ESP32 and FTDI, the QFN chips are to be used to reduce the surface area occupied as well as the height of the PCB.



The tool used for schematic layout was eschema whereas the tool used for pcb footprint layout was pcbnew. After the netlist was generated by eschema, it was loaded into pcbnew. The footprints along with the pin connections were generated. The copper trails and vias had to be made on both the front and back layers. The process of connecting the pins was difficult because the trails must not touch other trails to prevent short-circuiting. A 3D preview of the design is shown below. It was later decided that the battery should be placed below the PCB, and that the FTDI can be replaced with its QFN chip equivalent and relocated to below the SD card reader.



With the reduction of the surface area of the PCB to roughly 62mm x 47mm, the device becomes more portable, which is in-line with its function as a password storage tool. With this step completed, the spatial constraint for the project can be visualized. Any components that need to be added later in the project will be able to reserve enough space on the PCB. Throughout the week, I have learned how to design a PCB, and lookup the datasheet for the components to ensure enough room for them. The next steps will include preordering and soldering of components onto the PCB, as well as writing functions that will make the SD card more integrated with other components.

## WEEK 6:

**Date:** Oct 2, 2020

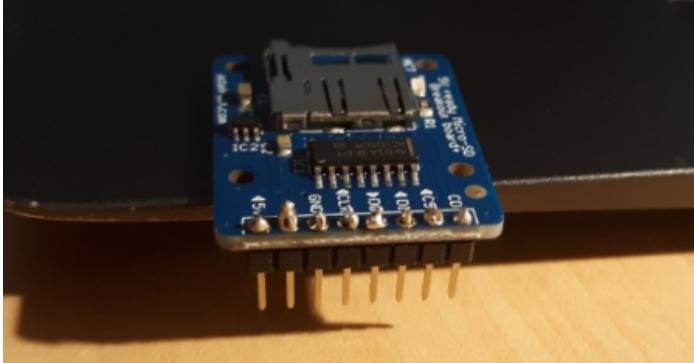
**Total hours:** 9

**Description of design efforts:**

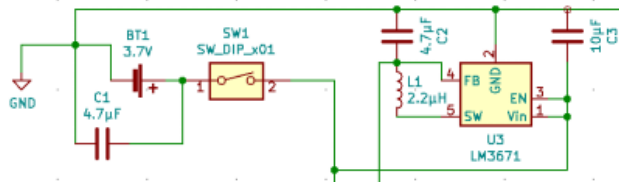


The first issue that was resolved was the instability of the SD card initialization process. The soldering kit was obtained from course staff and soldering was done at home. The SD card breakout board has been soldered and the signals were transmitted more reliably. The soldering process was difficult. When excessive solder was released onto the pin and pad, it was very troublesome to remove it. The temperature of the soldering pen was set to 250 C since the soldering material contains lead and therefore has a lower melting point. I also had to maintain distance between me and the soldering pen so as to not inhale too much fumes. With this experience, I was able to better improve my soldering skills.





During consultation with course staff, many faults with the PCB were pointed out, such as the unreliable voltage divider circuit and the ESP32 QFN chip. The voltage divider circuit does not guarantee a stable supply of current, thus it was decided that an LM3671 buck converter will be used to serve that purpose. The QFN chip requires an external oscillator and UART pin to be programmed, which introduces too much uncertainty to the project as the team is not experienced in doing that. Therefore, it was later decided, after a discussion with John, that a surface mount module of the ESP32 would be used, due to its built in programming components, and relatively small size compared to the development board. It was also decided that the fingerprint scanner will be placed next to the PCB, which will free up valuable space for other components.



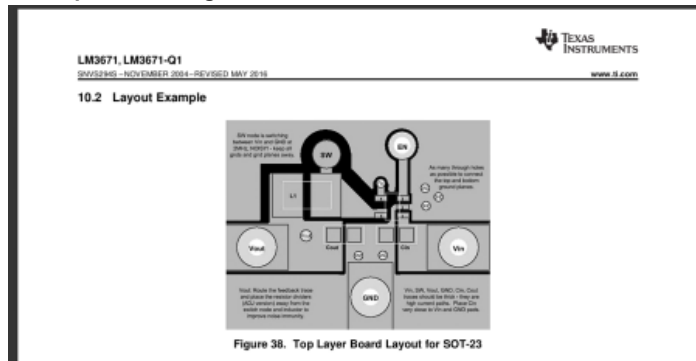
I have learned how to better design the power system such that the current supply will be adequate to drive the entire device. I have also learned to choose between surface mount and through hole modules. As of the end of the week, I am still redesigning the PCB. With John's help in breaking up the breakout boards into discrete components, I will be able to finish up a draft of the footprint of the PCB. Three new components will have to be purchased. They are the USB connector, LM3671 and FT232RL, all of which are surface mount modules. When these new modules arrive, we will be able to test the components together on a breadboard next week before ordering our first revision of the PCB.

## WEEK 7:

**Date:** Oct 9, 2020

**Total hours:** 10

**Description of design efforts:**



This week I planned to focus on finishing up the schematic and PCB layout. As pointed out by course staff during lab hours, the LM3671 layout did not abide by the layout guidelines stated in the LM3671 datasheet. Apart from that, the inductor and capacitor footprints were mismatched with the wrong sizes. The fast switching SW node of the LM3671 causes a lot of noise and had to be isolated far enough from other sensitive components. The capacitors also did not account for the first and second current cycle that flows through the step down converter, which may result in reversing magnetic curls interfering with nearby sensitive components, particularly the RF antenna on the ESP32.

**Table 1. Suggested Inductors and Their Suppliers**

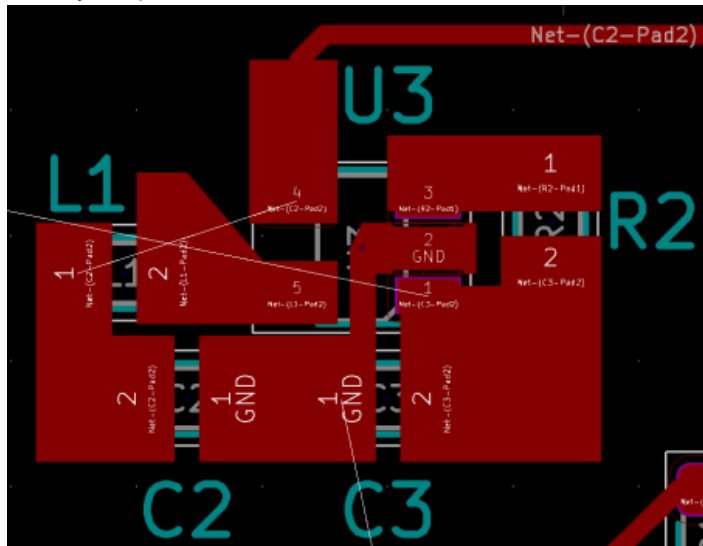
MODEL	VENDOR	DIMENSIONS L x W x H (mm)	D.C.R (maximum)(mΩ)
DO3314-222MX	Coilcraft	3.3 x 3.3 x 1.4	200
LPO3310-222MX	Coilcraft	3.3 x 3.3 x 1	150
ELL5GM2R2N	Panasonic	5.2 x 5.2 x 1.5	53
CDRH2D14NP-2R2NC	Sumida	3.2 x 3.2 x 1.55	94

**Table 2. Suggested Capacitors and Their Suppliers**

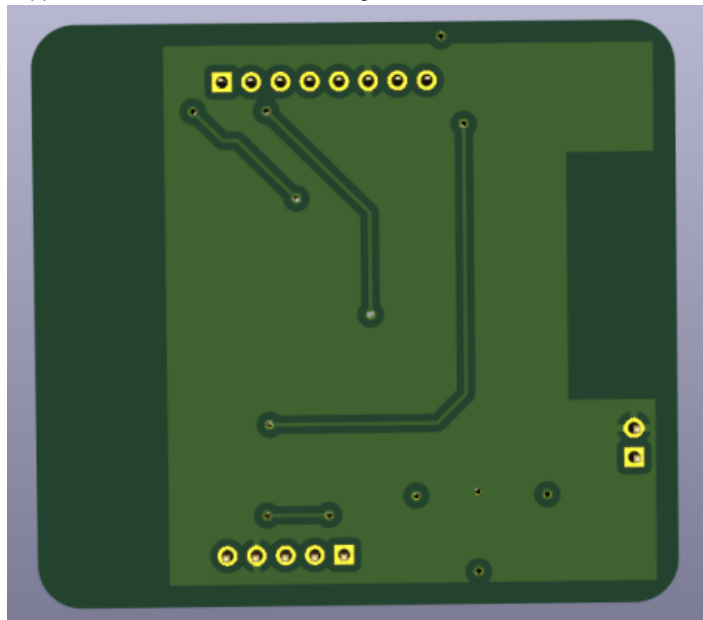
MODEL	TYPE	VENDOR	VOLTAGE RATING (V)	CASE SIZE INCH (mm)
<b>4.7 µF for C<sub>IN</sub></b>				
C2012X5R0J475K	Ceramic, X5R	TDK	6.3	0805 (2012)
JMK212BJ475K	Ceramic, X5R	Taiyo-Yuden	6.3	0805 (2012)
GRM21BR60J475K	Ceramic, X5R	Murata	6.3	0805 (2012)
C1608X5R0J475K	Ceramic, X5R	TDK	6.3	0603 (1608)
<b>10 µF for C<sub>OUT</sub></b>				
GRM21BR60J106K	Ceramic, X5R	Murata	6.3	0805 (2012)
JMK212BJ106K	Ceramic, X5R	Taiyo-Yuden	6.3	0805 (2012)
C2012X5R0J106K	Ceramic, X5R	TDK	6.3	0805 (2012)
C1608X5R0J106K	Ceramic, X5R	TDK	6.3	0603 (1608)

The datasheet of the LM3671 specified the type of inductors and capacitors recommended for use. I corrected the footprint of those components. The previous sizes were extremely small, which would have made the soldering process more difficult. The datasheet also listed a list of guidelines on how to place these components on the PCB in relation to the LM3671. With John assisting in researching the LM3671 circuit layout and studying the LM3671 datasheet, I redid the layout of the step down converter using Kicad, and added mini copper plates to prevent the

power supply from being restricted by the resistivity of the copper trails. I also rearranged some of the components on the PCB and grouped them by their communication interface, namely power systems on the top right, UART at the top, USB at the right, SPI at the bottom and the ESP32 being near the center of the board. Upon John's suggestion, I have replaced the male USB port with the female version and replaced the electrolytic capacitors with ceramic ones.



The USB port was further shifted to the right to account for the size of the micro-USB connector head. After placing the LM3671 as far away as possible from the antenna of the ESP32, I removed the ground plates beneath the ESP32 antenna, USB port and copper trails that transmit the USB signals to the FT232RL. The USB port was further shifted to the right to account for the size of the micro-USB connector head. After placing the LM3671 as far away as possible from the antenna of the ESP32, I removed the ground plates beneath the ESP32 antenna, USB port and copper trails that transmit the USB signals to the FT232RL.



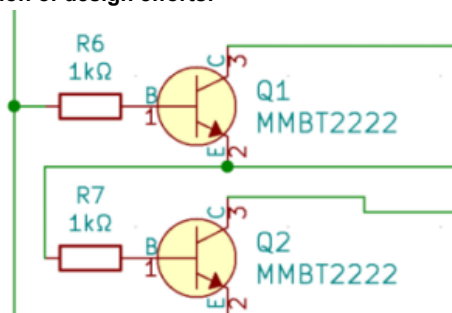
After two weekly meetings and one office hour meeting with course staff, I have gained valuable insights on PCB designs and incorporated the fixes into the latest revision. These steps will greatly prepare the team for midterm review next week, and speed up the finalization of our PCB. Through this rigorous process, I have learned to account for magnetic flux, copper trail resistivity and other EMI concerns that must be taken care of during the design. I have also improved my ability to lookup datasheets for the purpose of choosing the right footprint and layout. The next steps will be the preparation of presentation material for midterm review. On the project side, I will discuss with Ethan regarding the communication between the GUI and SDSPI interface. This will allow the user to extract files from external storage straight from the laptop.

## WEEK 8:

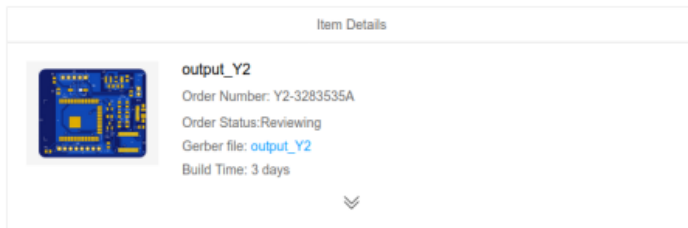
**Date:** Oct 23, 2020

**Total hours:** 12

**Description of design efforts:**



Minor changes were made to the PCB such as the addition of two transistors to the programming pin connections between the ESP32 and FT232 via KiCAD. The transistors were connected between the RTS and DTR pins of the ESP32, and the IO0 and EN pins of the ESP32. This ensures that the ESP32 can be reset and programmed during the powered state. The minimum copper trace width was calculated using 4pcb.com's online calculator to ensure that the resistivity of the trails does not inhibit current supply to the rest of the circuit. Signal and other traces would use the default 0.25mm trace width whereas the trace width for the power supply would be 0.4mm. The gerber files were then exported to JLC for fabrication of the PCB, after which the focus of my work shifted from hardware design to GUI-SD card communication.



After placing the order, I compiled a spreadsheet of all the components that will be needed for the PCB, including the resistors, transistors, inductors, capacitors and the micro USB port. Certain components are available from the ECE shop, while the rest were purchased from Digikey because the project will be using surface mount modules rather than through-hole which is the type available at the shop. This gave us a good estimate of the cost of all mounted components.

Name	Symbol	Component	Unit price	Quantity (purchase 3x more than needed)	Subtotal	Purchase link
Battery pin header	BT1	3.7 V	\$ -	3	\$ -	ECE shop
Capacitor	C1, C3	4.7uF	\$ 0.18	6	\$ 1.08	<a href="https://www.digikey.com/">https://www.digikey.com/</a>
Capacitor	C2, C6	10uF	\$ 0.17	6	\$ 1.02	<a href="https://www.digikey.com/">https://www.digikey.com/</a>
Capacitor	C4, C5	0.1uF	\$ 0.33	6	\$ 1.98	<a href="https://www.digikey.com/">https://www.digikey.com/</a>
Power LED	D1	Green LED	\$ 0.54	3	\$ 1.62	<a href="https://www.digikey.com/">https://www.digikey.com/</a>
USB LED	D2, D3	Blue LED	\$ 1.23	6	\$ 7.38	<a href="https://www.digikey.com/">https://www.digikey.com/</a>
USB port	J1	USB_B_micro	\$ -	3	\$ -	ECE shop
Inductor	L1	2.2uH	\$ 0.98	3	\$ 2.94	<a href="https://www.digikey.com/">https://www.digikey.com/</a>
Resistor	R1, R2	10kΩ	\$ 0.10	6	\$ 0.60	<a href="https://www.digikey.com/">https://www.digikey.com/</a>
Resistor	R3, R4	500Ω	\$ 0.10	6	\$ 0.60	<a href="https://www.digikey.com/">https://www.digikey.com/</a>
Resistor	R5, R6, R7	1kΩ	\$ 0.10	9	\$ 0.90	<a href="https://www.digikey.com/">https://www.digikey.com/</a>
DIP Switch	SW1	SW_DIP_x01	\$ -	3	\$ -	ECE shop
ESP32	U1	ESP32-WROOM-32	\$ 3.80	3	\$ 11.40	<a href="https://www.digikey.com/">https://www.digikey.com/</a>
FT232	U2	FR232RL-REEL	\$ 4.50	3	\$ 13.50	<a href="https://www.digikey.com/">https://www.digikey.com/</a>
Step down converter	U3	LM3671	\$ 1.39	3	\$ 4.17	<a href="https://www.digikey.com/">https://www.digikey.com/</a>
SD pin header	U4	ADA254	\$ -	3	\$ -	ECE shop
Fingerprint pin header	U5	ADA751	\$ -	3	\$ -	ECE shop
Transistors	Q1, Q2	MMBT2222	\$ 0.13	6	\$ 0.78	<a href="https://www.digikey.com/">https://www.digikey.com/</a>
Total					\$ 47.97	

I cooperated with Ethan on processing the incoming UART signals from the desktop GUI. The ESP32 would essentially read the UART signals and compare it with a table of available commands. I wrote the high level functions that will be called when the corresponding commands are received. This was tested by toggling the red and green LEDs from the GUI. Using the same format and with Gary's help, I wrote the code that stores the credentials provided by and retrieves the credentials requested by the GUI. These will be tested over the weekend, the success of which will indicate that our method of handling commands was reliable and can be expanded upon.

```
int cmd_request_credential(char* displayName, char* username) {
    if (!manifestEntryExist(displayName))
        return CMD_FAILURE;
    ManifestEntry* entry = getManifestEntry(displayName, username);
    if (entry == NULL)
        return CMD_FAILURE;
    FILE* fp = fopen("/sdcard/" + displayName);
    if (fp == NULL)
        return CMD_FAILURE;
    fseek(fp, 0, SEEK_END);
    size_t filesize = ftell(fp);
    fseek(fp, 0, SEEK_SET);
    char* buffer = calloc(filesize, sizeof(char));
    fread(buffer, sizeof(char), filesize, fp);
    uart_write_bytes(PORT_NUM, fp, filesize);
    free(buffer);

    return CMD_SUCCESS;
}

int cmd_store_credential(char* displayName, char* username, char* url, char* pw) {
    if (!manifestEntryExist(displayName))
        return CMD_FAILURE;

    addManifestEntry(displayName, username, url);

    char path[256] = {'\0'};
    strcat(path, "/sdcard/");
    strcat(path, displayName);
    FILE* fp = fopen(path, "w");
    if (fp == NULL)
        return CMD_FAILURE;

    fprintf(fp, pw);
    fclose(fp);

    return CMD_SUCCESS;
}
```

To increase the efficiency of indexing the password file for the requested website and login ID, a manifest file was utilized to store unprotected information such as website name, login ID and login url. This also allows the GUI to request for a list of websites which the user has stored credentials. This was done by writing parser and UART scripts via the ESP-IDF framework which will be programmed into the ESP32. This was tested locally without the GUI and was verified to work by manually inspecting the SD card contents on a computer. The SD card showed "Facebook:andrewgan,facebook.com,123456" and a file called "facebook" containing the password "123456" was created in the same directory.

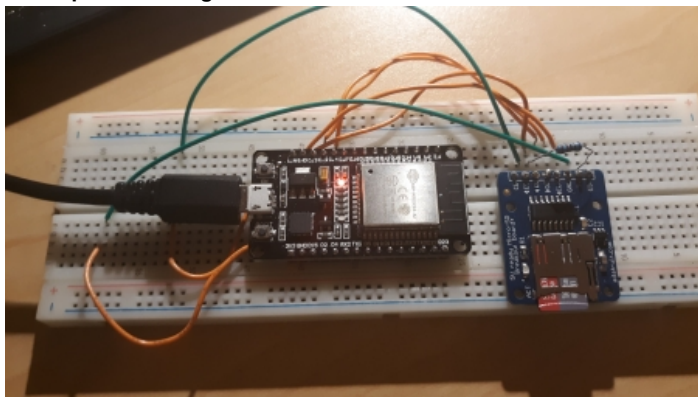
With the successful testing of the aforementioned steps, the project is progressing at a much quicker pace. I have learned how to place a PCB order and work with the UART to perform file I/O operations on the SD card. Future steps include a comprehensive testing for the desktop GUI to perform all available commands and incorporating the AES feature as a middleman between the GUI and SD card.

WEEK 9:

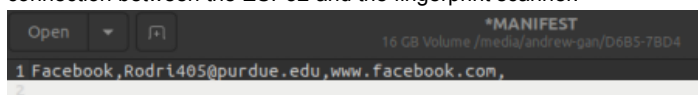
**Date:** Oct 30, 2020

**Total hours:** 9

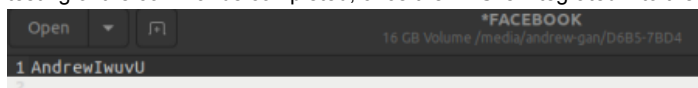
**Description of design efforts:**



This week, Ethan and I implemented, tested and debugged all the available commands, including storing credentials, retrieving credentials, retrieving login entries, deleting credentials and modifying existing credentials. This was done by having the GUI run on Ethan's computer while my computer was used for programming and debugging the code for processing the UART commands. This was done with the ESP-IDF framework and the GUI written in C#. One of the obstacles faced was that the GUI was listening into UART port 0, which is the same port used for console output. The code running on the ESP32 contained print statements which were all written to UART 0. This confused the GUI because the same UART port was used for transmitting commands. The solution was to redirect the console output to UART 1. As of now, UART 0 is reserved for programming and communication with the GUI, UART 1 acts as an output for console debugging, whereas UART 2 establishes a connection between the ESP32 and the fingerprint scanner.



Another minor issue encountered was that the program assumed the existence of a manifest file prior to reading from and writing to it. This caused the program to run into errors and was resolved by checking for the existence of the file before proceeding with the parsing. I also started looking into incorporating the AES encryption into the program. Currently, the data that are stored on the SD card are unencrypted. A high-level AES function which receives a string as input and provides an encrypted string as output can be utilized between the GUI and SD card. With the testing of the commands completed, once the AES is integrated into the project, a functioning prototype of the device will be obtained.



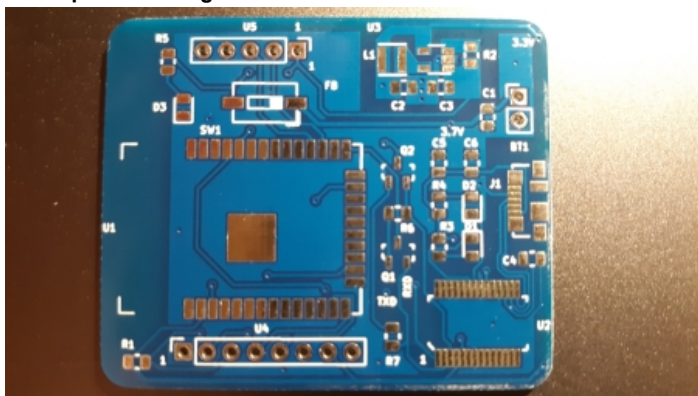
I will be receiving the PCB by this week and will begin soldering the surface mount components onto the PCB. Once that is done, the steps for verifying the PCB will be done as detailed in the lecture videos. This can be accomplished using the ADALM oscilloscope to measure the clock frequency and voltage level of the LM3671 step down converter.

## WEEK 10:

**Date:** Nov 6, 2020

**Total hours:** 11

**Description of design efforts:**



The PCB has finally arrived. Although the PCB was verified by the manufacturer, I thought it would be a good idea to inspect the PCB myself. Upon receiving it, I verified the connections of the copper pours visually before using the ADALM to test the conductivity of the copper pour by providing a voltage to a pad and measuring the voltage of another connected pad. This was done starting with the power and ground pours followed by the signal pours. The ADALM showed that the copper trails were all done accurately as described in the Gerber files sent to the PCB manufacturer. With the PCB verified, the next step would be the soldering of surface mount components onto the pads.

Ethan was responsible for soldering the surface mount components while I assisted in determining the location of the components on the PCB. By the end of the week, we have managed to solder all the resistors onto the PCB and will be testing whether they were solidly connected to the pads. Due to the small size of the 0805 components, we had to resort to using the tip of a pencil to position the components onto the desired pad and solder them with extreme caution so as to not damage the components using the heat from the soldering tool. It was a challenging process but we were able to speed up the process for each component as we become more accustomed to the soldering of 0805 components.



```

W (523) spi_flash: Detected size(4096k) larger than the size in the binary image header(2048k).
I (534) cpu_start: Starting scheduler on PRO CPU.
I (0) cpu_start: Starting scheduler on APP CPU.
I (545) SD-CARD: Successfully initialized SPI bus
I (3555) gpio: GPIO[15]: InputEn: 0 OutputEn: 1 OpenDrain: 0 Pullup: 0 Pulldown: 0 Intr: 0
I (3555) sdspi transaction: cmd=52, R1 response: command not supported
I (3595) sdspi transaction: cmd=5, R1 response: command not supported
I (3635) SD-CARD: Successfully mounted SD card
I (3635) GPIO: Successfully initialized GPIO
I (3645) BTDM_INIT: BT controller compile version [e250020]
I (3645) system api: Base MAC address is not set
I (3645) system api: read default base MAC address from EFUSE
I (3745) phy: phy version: 4370, 4e803b3, Aug 11 2020, 14:18:07, 0, 0
I (4205) SPP-ACCEPTOR DEMO: ESP_SPP_INIT_EVT
I (4295) SPP-ACCEPTOR DEMO: event: 10

```

Other than that, I also integrated the Bluetooth code prepared by Jiahui with the rest of the ESP32 code. Several changes were made to the Bluetooth code such that it is compatible with the code that was already implemented for UART communication. Initially, there were concerns as to how the ESP32 will determine whether UART or Bluetooth will be used by the GUI for communication as there was no connection between the USB port and ESP32 to check for a wired connection. Thankfully, the Bluetooth library on the ESP32 registers callback functions and calls them upon receiving a software interrupt. This enabled the ESP32 to simultaneously handle both modes of communication. With this step, I have gained some insights as to the inner workings of the ESP32 Bluetooth module. However, some strings sent over Bluetooth appear to be corrupted, and so that needs to be improved upon.

The things to do for next week would include soldering the remaining components onto the PCB by individually testing each system, starting with the power system and ending with the external hardware connections. The Bluetooth connection will have to be rigorously tested to ensure it functions as reliably as the UART. The integration of AES will come next.

## WEEK 11:

**Date:** Nov 13, 2020

**Total hours:** 8

**Description of design efforts:**

```

void aes_encrypt(uint8_t* plaintext, uint8_t* key, uint8_t* ciphertext) {
    uint8_t iv[16];
    memcpy(iv, key, sizeof(iv));

    esp_aes_context ctx;
    esp_aes_init(&ctx);
    esp_aes_setkey(&ctx, key, 256);

    esp_aes_crypt_cbc(&ctx, ESP_AES_ENCRYPT, sizeof(plaintext), iv, (uint8_t*)plaintext, (uint8_t*)ciphertext);

    esp_aes_free(&ctx);
}

void aes_decrypt(uint8_t* ciphertext, uint8_t* key, uint8_t* plaintext) {
    uint8_t iv[16];
    memcpy(iv, key, sizeof(iv));

    esp_aes_context ctx;
    esp_aes_init(&ctx);
    esp_aes_setkey(&ctx, key, 256);

    esp_aes_crypt_cbc(&ctx, ESP_AES_DECRYPT, sizeof(ciphertext), iv, (uint8_t*)ciphertext, (uint8_t*)plaintext);

    esp_aes_free(&ctx);
}

```

Due to the need to prioritize integrating the different subcomponents of the project, as well as a fully soldered and tested PCB not being a requirement for the course this semester, my team and I have decided to put the PCB soldering on hold while we focus on rigorously testing the remaining parts of the project, i.e. AES and fingerprint encryption. I started reading through and understanding the AES encryption and decryption code provided by John. I mainly split up the code into two functions, with one for encryption and the other for decryption. Due to some technical problems with retrieving the image file and converting it into an AES key, the function could not be fully tested at this stage. This was all done within the ESP-IDF framework as usual. Once the feature of downloading and converting the fingerprint character file into an AES key is complete, the encryption function would be called for every read write on protected data. From this step, I have gained an insight on the way AES encryption is handled on the ESP32, that is by installing the required drivers and calling the appropriate library functions.

```

int cmd_request_credential(char* displayName, char* username, int mode) {
    if (getManifestEntry(displayName, username) == NULL)
        return CMD_FAILURE;
    char path[256] = {'\0'};
    strcat(path, "/sdcard/");
    strcat(path, displayName);
    FILE* fp = fopen(path, "r");
    if (fp == NULL)
        return CMD_FAILURE;
    fseek(fp, 0, SEEK_END);
    size_t filesize = ftell(fp);
    fseek(fp, 0, SEEK_SET);

    // retrieve encrypted content
    char* buffer = calloc(filesize + 1, sizeof(char));
    fread(buffer, sizeof(char), filesize, fp);

    uint8_t* aes_key = retrieveFingerprintFile();

    // perform decryption using key generated from fingerprint
    char* plaintext = calloc(filesize + 1, sizeof(char));
    aes_decrypt(buffer, aes_key, plaintext);

    buffer[filesize] = '\n';
    if (mode == UART_MODE)
        uart_write_bytes(PORT_NUM, buffer, filesize + 1);
    else if (mode == BT_MODE)
        btSendData((uint8_t*) buffer, filesize + 1);
    free(buffer);
    fclose(fp);
    return CMD_SUCCESS;
}

```

Another thing that was worked upon was the fixing of the code that sends the commands via Bluetooth. Previously, the header and content of a command packet were sent in separate Bluetooth transactions. I printed out all the byte values received by the ESP32 to inspect the actual bytes transmitted and noticed unintelligible characters at the end. These were the checksum and end of frame bytes that were added to the end of a

transmitted packet. But did not account for that which caused the reading of the Bluetooth packet to be offset by a few bytes. By sending all the information needed in a single Bluetooth packet, Ethan and I managed to simplify the reading and processing of the packet without having to account for multiple header and ending bytes. With the Bluetooth fully debugged and tested, our project now operates reliably on both UART and Bluetooth communication. My knowledge of Bluetooth was very limited prior to working on Bluetooth communication. After debugging and resolving the previous issue with the communication module, I have strengthened my understanding of the packet formatting.

As of now, we have fully met 2 of our PSSCs, namely external storage and sending commands via UART. We have also succeeded in sending commands via Bluetooth without encryption. The next steps will include fully integrating and testing the fingerprint data retrieval and password encryption functionalities.

## WEEK 12:

**Date:** Nov 20, 2020

**Total hours:** 10

**Description of design efforts:**

```
if (checkFingerEnrolled() == 0) {
    if (enrollFinger(0) == -1) {
        ESP_LOGE(TAG, "Fingerprint enrollment failed");
        return;
    }
}
```

```
int cmd_store_credential(char* displayName, char* username, char* url, char* pw) {
    if (authenticateFinger() == 0)
        return CMD_FAILURE;

    // uint8_t* key = NULL;
    // int keysize = 0;
    // if (getCryptoKey(&key, &keysizes) == -1)
    //     return CMD_FAILURE;

    addManifestEntry(displayName, username, url);
    char path[256] = {'\0'};
    strcat(path, "/sdcard/");
    strcat(path, displayName);
    FILE* fp = fopen(path, "w");
    if (fp == NULL)
        return CMD_FAILURE;

    // char* encryptedText = calloc(strlen(pw), sizeof(char));
    // my_aes_encrypt((uint8_t*)pw, key, (uint8_t*)encryptedText);
    fprintf(fp, pw /*encryptedText*/);

    fclose(fp);
    return CMD_SUCCESS;
}
```

This week, my team and I focused on integrating and testing the subsystems of our project together. The first component to be integrated was the fingerprint authentication functionalities. After John was able to thoroughly test and write high-level functions for fingerprint enrollment and matching, I worked on calling these functions in the main loop code. Whenever a command requires access to sensitive information such as credential retrieval, or when the fingerprint data file is needed to generate an encryption key for storing a new credential, the user would be prompted to provide a fingerprint scan. I was able to achieve this by communicating with John who was responsible for cryptography and the fingerprint side of things via Zoom call.

```
void my_rsa_encrypt(uint8_t* plaintext, uint8_t* ciphertext) {
    unsigned char plaintext_buf[RSA_SEND_LEN];
    ciphertext = calloc(RSA_SEND_LEN, sizeof(uint8_t));

    memcpy(plaintext_buf, plaintext, strlen((char*)plaintext));

    TEST_ASSERT_EQUAL(KEYSIZE, (int)client_rsa.len * 8);
    TEST_ASSERT_EQUAL(KEYSIZE, (int)client_rsa.d.n * sizeof(mbedtls_mpi_uint) * 8); // The private exponent

    TEST_ASSERT_EQUAL(0, mbedtls_rsa_public(&client_rsa, plaintext_buf, ciphertext));
}

void my_rsa_decrypt(uint8_t* ciphertext, uint8_t* plaintext) {
    unsigned char ciphertext_buf[RSA_SEND_LEN];
    plaintext = calloc(RSA_SEND_LEN, sizeof(uint8_t));

    memcpy(ciphertext_buf, ciphertext, strlen((char*)ciphertext));

    ciphertext_buf[0] = 0; // Ensure that orig_buf is smaller than rsa.N

    TEST_ASSERT_EQUAL(KEYSIZE, (int)my_rsa.len * 8);
    TEST_ASSERT_EQUAL(KEYSIZE, (int)my_rsa.d.n * sizeof(mbedtls_mpi_uint) * 8); // The private exponent

    TEST_ASSERT_EQUAL(0, mbedtls_rsa_private(&my_rsa, NULL, NULL, ciphertext_buf, plaintext));
}
```

I have also started on satisfying the final PSSC for our project, that is ensuring a secure Bluetooth connection with the GUI. The plan is to utilize the RSA algorithm provided by ESP32's mbedtls (hardware acceleration) library. The library contains high-level functions to generate public-private key pairs, perform encryption and decryption on a string of text. When studying the RSA algorithm to be used, I found the test\_rsa.c example code provided within the ESP-IDF directory rather helpful. The ESP-IDF documentation website on RSA struct types was also useful in understanding the sequence of events that leads to a successful encryption and decryption of data. This step will greatly strengthen the security of the user credentials when sending instructions via the wireless mode of communication. I have also gained a deeper understanding as to how the RSA algorithm works.

### Test 1 - fingerprint auth. ~~test 1~~

- - perform auth. before certain cmds
- - ensure sensor light turns off
- - check enrollment & unenroll cmds

### Test 2 - AES enc.

- - generate key from fingerprint char file
- - verify SD-card content is unintelligible
- - ensure decryption is correct via GUI

### Test 3 - one way RSA

- - ESP32 sends public key to GUI (unprotected)
- - GUI encrypts cmd using key
- - ESP32 decrypts and exec cmds

### Test 4 - two way RSA

- - same as step 3 but GUI sends public key to ESP32
- - modify RSA context to contain GUI's RSA info

### Test 5 - DH for public key exchange

- - use DH to exchange public keys
- - find a way to use DH generated common key to generate public & private keys

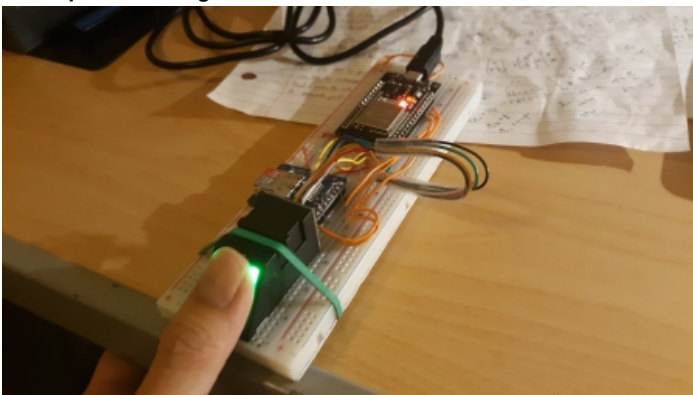
With the various components being integrated, there is a need for a unit testing plan. I have drafted a testing plan to procedurally test each step of integration to prevent the debugging process from being too complex to tackle. The testing plan will include fingerprint authentication, AES, one-way RSA, two-way RSA before implementing the Diffie-Hellman algorithm which has been written by Jiahui Zhu. The testing will be done with Ethan performing operations from the GUI side and the rest of us inspecting the components we developed. This will greatly improve the reliability of the project and prepare the team for PSSC demonstrations and finally, the final presentation for the course. I have also written the user manual which describes the procedures and troubleshooting methods to effectively operate the password storage device.

### WEEK 13:

**Date:** Nov 27, 2020

**Total hours:** 15

**Description of design efforts:**



For the past 2 weeks, the focus has been integration and unit testing. This was done by drafting a testing plan and checking the items off one by one to ensure that not only are the PSSCs met, but that the various components of the project can communicate with one another. The first test was fingerprint authentication. This was done before executing commands that involve sensitive information. Thanks to John, several high level functions were written and unit tested before being added to the main code. Whenever a command that requires fingerprint authentication was made, the fingerprint sensor lit up to request a fingerprint scan and turned off when a match was found, or not found after 3 attempts. The user is allowed 3 chances to provide a matching fingerprint because even if a match was found, the match score may not be high enough for the sensor to generate a hashed key to encrypt or decrypt the user credentials. The ability to enroll and unenroll fingerprint were also tested.

```

I (8342) FINGERPRINT: Fingerprint initialized
E (11852) FINGERPRINT: Failed to generate charFile for buffer 1... resp = 7
I (11872) FINGERPRINT: Matching finger found with ID: 0 and match score: 4
I (13772) FINGERPRINT: Matching finger found with ID: 0 and match score: 90
I (13772) FINGERPRINT: Fingerprint authentication successful
I (14132) FINGERPRINT: Received 13 total packets, 12 data packets and 1 ack packet

```

The next part of integration was AES encryption. Although a sample AES code utilizing mbedtls hardware acceleration was provided in ESP-IDF's installation folder, the code had to be broken into separate functions and unit tested. An issue that I encountered was the presence of garbage values in the plaintext buffer which resulted in an incorrect encryption. I had to print out the encrypted text at the byte level to figure out what went wrong. Another issue was that when the user provided a fingerprint scan, the buffer was loaded with the new fingerprint which was mistakenly used to generate a hashed crypto key instead of the first fingerprint scan, producing incorrect output. The fix was to reload the initial fingerprint into the buffer before any hashing occurs. The encrypted text stored on the SD card was verified to be unintelligible.

```

I (14132) AES: Successfully hashed crypto key
I (14152) FINGERPRINT: Successfully hashed crypto key
I (14152) AES: Successfully encrypted seemslikeitwontbesnowingtoday123
I (14162) CMD: Successfully stored credential for Facebook
E (19442) FINGERPRINT: Failed to generate charFile for buffer 1... resp = 7
I (19532) FINGERPRINT: Matching finger found with ID: 0 and match score: 1405
I (19532) FINGERPRINT: Fingerprint authentication successful
I (19892) FINGERPRINT: Received 13 total packets, 12 data packets and 1 ack packet
I (19892) AES: Successfully hashed crypto key
I (19912) FINGERPRINT: Successfully hashed crypto key
I (19912) AES: Successfully decrypted into seemslikeitwontbesnowingtoday123

```

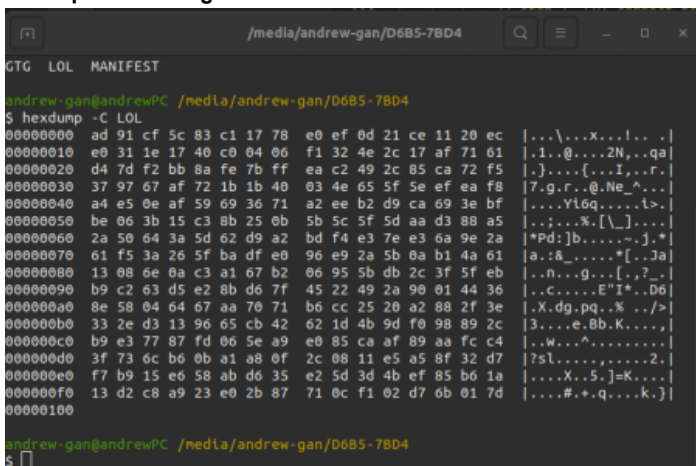
With the fingerprint and AES out of the way, the third phase of integration was the RSA. Due to my unfamiliarity with RSA libraries on C# and the ESP32, I decided to first implemented one way encryption, i.e. having the GUI encrypt using the ESP32's public key and the ESP32 decrypt using it's own private key. I had to implement a handshake sequence upon a successful Bluetooth pairing, which involves the ESP32 confirming the GUI that the virtual COM port was using Bluetooth instead of UART, and then having both parties exchange their public keys before any communication can begin. The output log from the GUI confirms that the public keys were exchanged successfully. The conclusion of 3 out of 5 tests meant we have satisfied 4 out of 5 PSSCs. I believe we are on track to complete the scope of our project by December 5th, 2020.

## WEEK 14:

**Date:** Dec 4, 2020

**Total hours:** 20

**Description of design efforts:**



```

/media/andrew-gan/D6B5-7BD4
GTG LOL MANIFEST
andrew-gan@andrewPC: /media/andrew-gan/D6B5-7BD4
$ hexdump -C LOL
00000000 ad 91 cf 5c 83 c1 17 78 e0 ef 0d 21 ce 11 20 ec [...]X...l...
00000010 e0 31 1e 17 40 c0 04 06 f1 32 4e 2c 17 af 71 61 [...]..@....2N...qa
00000020 d4 7d f2 bb 8a fe 7b ff ea c2 49 2c 85 ca 72 f5 [...]....[...I...r...
00000030 37 97 67 af 72 1b 1b 40 03 4e 65 5f 5e ef ea f8 [...]7.g.r...@.Ne_...
00000040 a4 e5 0e af 59 69 36 71 a2 ee b2 d9 ca 69 3e bf [...]....Yl6q....l>...
00000050 be 06 3b 15 c3 8b 25 0b 5b 5c 5f 5d aa d3 88 a5 [...]....X.[\]....
00000060 2a 50 64 3a 5d 62 d9 a2 bd f4 e3 7e e3 6a 9e 2a [...]Pd:]b.....j...
00000070 61 f5 3a 26 5f ba df e0 96 e9 2a 5b 0a b1 4a 61 [...]a:a.....*[...]Ja
00000080 13 08 0e 0a c3 a1 67 b2 06 95 5b db 2c 3f 5f eb [...]..n...g...[,?...
00000090 b9 c2 63 d5 e2 8b d6 7f 45 22 49 2a 90 01 44 36 [...]c.....E"I*..D6
000000a0 8e 58 04 64 67 aa 70 71 b6 cc 25 20 a2 88 2f 3e [...]X.dg.pq..%../>
000000b0 33 2e d3 13 96 65 cb 42 62 1d 4b 9d f0 98 89 2c [...]3....e.Bb.K....
000000c0 b9 e3 77 87 fd 06 5e a9 e0 85 ca af 89 aa fc c4 [...]..w...^.....
000000d0 3f 73 6c b6 0b a1 a8 0f 2c 08 11 e5 a5 8f 32 d7 [...]7sl.....2...
000000e0 f7 b9 15 e6 58 ab d6 35 e2 5d 3d 4b ef 85 b6 1a [...]....X..S.] =K....
000000f0 13 d2 c8 a9 23 e0 2b 87 71 0c f1 02 d7 6b 01 7d [...]....#..+..q...k..
00000100
andrew-gan@andrewPC: /media/andrew-gan/D6B5-7BD4
$

```

In preparation for checking off the project PSSCs by December 5, my team and I worked tirelessly to integrate and test all the functionalities of the device. It was initially believed that the fingerprint and AES operations work flawlessly, a bug was discovered earlier this week regarding the fingerprint sensor's ability to store and retrieve the fingerprint characteristic file. It was discovered that the fingerprint hashed file used to encrypt and decrypt the password was different although the file should have remained the same. After the issue was resolved by John, I proceeded to test the device along with Ethan and confirmed that the fingerprint file was properly hashed and the password retrieval functionality is now stable.

```

int my_rsa_key_send() {
    rsa_pub_info key_to_send = {.divider = '\n', .end = '\n'};
    key_to_send.public_exp = *my_rsa.E.p;
    memcpy(key_to_send.public_mod, my_rsa.N.p, 64 * sizeof(mbedtls_mpi_uint));
    btSendData((uint8_t*)&key_to_send, 0, sizeof(key_to_send));
    ESP_LOGI(TAG, "Successfully sent RSA key pair");
    return RSA_SUCCESS;
}

int my_rsa_key_rcv(uint8_t* data) {
    rsa_pub_info key_to_rcv;
    memcpy(&key_to_rcv, data, sizeof(key_to_rcv));

    client_rsa.E.n = 1;
    client_rsa.E.s = 1;
    client_rsa.E.p = calloc(client_rsa.E.n, sizeof(*(client_rsa.E.p)));
    *client_rsa.E.p = key_to_rcv.public_exp;

    client_rsa.N.n = KEYSIZE / 32;
    client_rsa.N.s = 1;
    client_rsa.N.p = calloc(client_rsa.N.n, sizeof(*(client_rsa.N.p)));
    memcpy(client_rsa.N.p, key_to_rcv.public_mod, 64 * sizeof(mbedtls_mpi_uint));

    client_rsa.len = KEYSIZE / 8;
    client_rsa_received = 1;

    ESP_LOGI(TAG, "Successfully received RSA key pair");
    return RSA_SUCCESS;
}

```

The final PSSC to implement was the ability to secure Bluetooth communication between the ESP32 and GUI using shared-key encryption, namely RSA. I wrote some code that notified both sides of when a Bluetooth handshake occurred and the RSA key exchange was completed. The ESP32 library included the mbedtls toolkit which is used for hardware acceleration of cryptographic operations. The same library was used for AES. I implemented the RSA operation on the ESP32 side by referencing the mbedtls API from the official website and the sample code



provided by the ESP-IDF toolkit. After understanding how RSA was performed, I wrote several high-level functions for key initialization generation, public key exchange, and message encryption and decryption. I conducted a local RSA test by first generating an RSA key pair to encrypt the message, then performing public key exchange, and using the ESP32 decrypt function to return the original text to be printed to the console output. After the RSA functionality was verified, it was tested with the GUI's key exchange and encryption methods.

```
I (13709) RSA: Successfully sent RSA key pair
I (13709) RSA:
I (13719) RSA: 01 00 01 00 00 00 00 00 0a c1 86 a2 fa ba e5 c6
I (13719) RSA: e1 5e e8 8c 3e 44 c0 46 4e 99 52 41 5f e9 a8 d5
I (13729) RSA: 99 55 3a de d8 2e 19 55 d2 79 6b 19 3f 8b c5 3c
I (13739) RSA: 43 66 c6 8d 13 2d 55 9a 58 c0 04 3c 91 8e 8d 67
I (13739) RSA: ed dc 04 d6 70 4d b7 72 22 75 83 70 02 54 64 8d
I (13749) RSA: 14 4c d6 c9 6f 42 ad 92 88 de 13 fb 8b 2e e9 c0
I (13759) RSA: fd aa e7 ae a2 79 43 9a a8 0f 6a cd 61 06 71 7e
I (13759) RSA: 9e 53 e2 fa 23 66 8e e0 9c c7 d0 4c 7b f6 25 ce
I (13769) RSA: 38 95 4a ab d8 9e 76 30 0b 57 3e cf 29 3e 7f c4
I (13779) RSA: 62 78 b8 c6 cb c1 db 41 6d fe 52 86 26 2f b0 2e
I (13779) RSA: e3 db 0d 44 79 ff d3 f1 2f 67 73 86 1e 03 b1 98
I (13789) RSA: 40 3f 1f f5 9b c1 30 29 fe 97 2b a7 40 f7 48 e1
I (13789) RSA: 15 83 8d cd b0 e0 f3 a0 3d 34 c2 a9 ec b2 bd 42
I (13799) RSA: ea 97 26 7e 0a fc 31 7f c0 85 37 54 e3 5d 42 c6
I (13809) RSA: ca c3 d5 e1 c8 0a d2 70 b6 63 50 cb f4 be 82 73
I (13809) RSA: 32 c1 86 af d8 3f ed bc b0 c6 be ea 1a 1d 02 46
I (13819) RSA: c3 f9 78 c9 8a f4 1c 88 b2 0a 00 00 00 00 00 00
I (13829) RSA:
```

public  
exponent

divider

modulus

The GUI's encryption and decryption methods were tested locally, and later the GUI was able to encrypt messages before sending it to the ESP32 to be decrypted. However, the GUI was unable to decrypt messages encrypted by the ESP32. I had the ESP32 print out the public key that was generated by itself and the one that was received from the GUI. The print statements confirmed that the public key exchange was implemented correctly. When code was added to print out the encrypted message on both the ESP32 and GUI using the same original text, it was noticed that the output was different, though it was later found to be expected behavior given that the RSA encryption we were using was the PKCS #1 v1.5 padding standard, which utilizes a PRNG for that purpose. This bug consumed a lot of time to fix but it was later discovered by John that the public key initialization on the ESP32 had issues. After ensuring that two-way RSA communication works as expected, all the functionalities of the device were tested again, including the UART mode. With this step done, all 5 PSSCs were implemented and demonstrated on Dec 4, 2020 at office hours.