**Journal regarding the Summative Computer Engineering Project**

**Journal for documentation of the Summative project for Grade 11 Computer Engineering**

**Benjamin Liu**

**TEJ3M - Grade 11 Computer Engineering**

**Mr. Gordon Roller**

**January 22nd, 2020**

**Day 1: Selection of a Project Idea**

The main thing done was to order arduino parts as well as another chip for the development, if more pins are required. The project idea was also formulated and plans for the following days were drawn up.

**Day 2: First Build**

Today marks the first build of the project, in which I tried to test out the fingerprint sensor and setup a simple version of a keypad lock has been created. During the process, I’ve learned how to use the Keypad library to register multiple keypresses from the buffer into a char array and to then compare every value at each index in the array to a password array that was set up. Effectively, that was the entire working mechanism of the basic keypad lock. I plan to add the network sections and biometrics will be added on in sections in the coming days of development.

The main problem that I encountered was actually to get the fingerprint sensor to work. I spent over 2 hours to find the documentation on the sensor itself (as the manufacturer’s website no longer exists) and to get it to run. As it turns out, after doing some diagnostics work and reinstalling the libraries that it uses (the Adafruit fingerprint sensor library), the sensor turned out to be DOA (dead on arrival). This provided me with the valuable lesson to buy sensors from more reputable manufacturers that actually have documentation and not to just order the cheapest one off of amazon. So, I then had to return the sensor and to order a new one off of Active Tech.

**Day 3: Google Authenticator**

I was thinking of ways to incorporate more interesting features into the lock and I came up with the idea of using google authenticator while logging in to my email. What’s interesting about google authenticator is the fact that it actually has no communication whatsoever with the lock (or account) itself, employing the method of syncing system time. Since the entire basis of incorporating TOTP (Time based One-Time Password algorithm) revolved around system time (normally kept by a Real Time Clock), I had to find a way to put one on my arduino. However, since I didn’t want to wait around to buy and get more components shipped, I decided to employ a library to incorporate a software based real time clock.

This was no challenge, as the documentation had provided the required commands to setup a simple clock running on whatever time it was set up with (by manually entering the date and time). However, that was cumbersome and inaccurate as there would always be discrepancies between what the human entered and the actual time. So, to mitigate that problem, I simply got the arduino to grab the time from the host system/computer it was connected to and to input it into the rtc setup commands. However, it wasn’t as straightforward as I had hoped, as the Time grabbed by the arduino was stored as chars, which I had to then turn into ints. To do so, I employed .toInt(), after taking certain sections of the char array and storing them as strings, to then be able to input them into the RTC.

I then commenced on actually setting up the TOTP, which actually did have a library. See, to synchronize the one-time passwords of the authenticator and the lock, they both share a Base32 encoded secret key, which is used in the calculations for the one-time passwords. This, along with the system time, allow for the lock and authenticator to have the same passwords at the same time.

In the case of the Arduino, some of the calculations for the secret key already had to be done (which I will eventually implement into the Arduino itself(though for the time being, are done via a third-party website)), decoding the secret key and converting it into a hexadecimal array for the Arduino to use. This is then used with the sha1 cryptographic algorithm to generate an HMAC (Hash-based Message Authentication Code). Then, the TOTP can be calculated using the HMAC, the Unix timestamp calculated via the time set into the RTC of the Arduino, and the secret key (in the hex array), to produce a human readable 6 digit code.

One main problem that I’ve realised during this is that I probably won’t have enough pins on my arduino for the LCD shield I plan to implement, so I will probably have to order an Arduino mega off the internet.

Another funny problem I had was that I completely forgot that the unix timestamp had to be calculated with UTC and I spent an hour trying to figure out what was causing the Arduino to generate different codes. I simply had to add 5 to the hour counter of RTC to resolve the problem.

**Day 4: Base32 Encoding and Key generation**

As you may or may not know, google authenticator requires a Base32 encoded Secret Key to be able to function and synchronize with a TOTP enabled device/account. Originally, as I was using a third-party tool online to actually generate this Secret Code and the arduino hex array, it required manual user input to function.

However, I wanted to have the lock mechanism to be completely automatic and require as little user interaction as possible. So, I decided to setup the arduino to be able to run this encryption. The way Base32 encoding works is actually rather simple. It takes an input string and converts each character into its ASCII number. Then, it takes those numbers and converts them into bytes, which are then concatenated and split into chunks of 5 bits. If there are less than 40 bits total (or 5 bytes), then the missing ones are padded on as zeros. Afterwards, the chunks are converted to uppercase letters (A to Z) and numbers (2 to 7 (0 and 1 are not used to to their visual similarity to the letters O and I)) based on the value of each chunk as an integer, which corresponds to a certain symbol in the RFC 4648 alphabet:

|  |  |  |  |
| --- | --- | --- | --- |
| Integer Value | Corresponding symbol | Integer Value | Corresponding symbol |
| 0 | A | 16 | Q |
| 1 | B | 17 | R |
| 2 | C | 18 | S |
| 3 | D | 19 | T |
| 4 | E | 20 | U |
| 5 | F | 21 | V |
| 6 | G | 22 | W |
| 7 | H | 23 | X |
| 8 | I | 24 | Y |
| 9 | J | 25 | Z |
| 10 | K | 26 | 2 |
| 11 | L | 27 | 3 |
| 12 | M | 28 | 4 |
| 13 | N | 29 | 5 |
| 14 | O | 30 | 6 |
| 15 | P | 31 | 7 |

In the case of the arduino hex array, the process would be the same till the ASCII, at which point, the ASCII values would be converted to their corresponding hexadecimal values to be stored in arduino.

Originally, I planned to find a library to do the conversions for me, though my search for one came up empty handed, as the only \*functioning\* cryptography libraries that encoded to Base32 were for the ESP32 cryptography module, which I didn’t want to buy. So, I decided that I would do all the conversions manually in the code.

To begin with, I converted single characters in a test string to an integer (as chars are able to be directly changed it their ASCII value when converted to an int). From there, I was able to use itoa to convert the number to a base 2 number (binary) and store it in a char array. However, I wasn’t able to get the values in that array to transfer over to a larger one, as it ended up infinitely looping my code. I ended up finding out that I could have simply employed a memcpy function to copy over a certain length of data from one array to another.

**Day 5: Base32 Encoding and Key generation (part 2)**

Over the course of yesterday and today, I looked up on more ways I could copy over the values between arrays and decided to streamline the process by simply using the strcpy (since I don’t need to know the length of the data anymore(for this part of the process)) and strcat, which would copy over and concatenate the data together.

I also implemented a way to find out how much padding I would need for each byte (as the arduino would only provide up till the first high bit (first ‘1’) in each byte, resulting in most to either have a data length of 6 or 7). This could be done with a simple Serial.write command, which would return me the length of the data as an integer. Then by employing an if….else if loop, I’d be able to add the correct amount of padding based on the length of the data.

Once that was done, I used a very crude way to copy data over to the chuck arrays, by manually copying each section to each array, line by line.

After that, the actual conversion to Base32 was much simpler, as it was really just a matter of comparing the data in the arrays to a string. If the data match, then a value would get concatenated to a string that stored the secret key. It would then get displayed once the process was over.

One funny mistake was the fact that I miscounted and accidentally only provided 14 char arrays for the 16 chunks that had to be stored, which originally resulted in my Secret Code missing 2 values. I also ended up making a funny indexing mistake as I was setting up the main array to store the binary, further causing funny errors.

The next step that I plan to do is to optimize the actual code, especially the copying of chunks.

**Day 6: Base32 Encoding and Key generation (part 3)**

I decided that to optimize and clear up space, I would fix the inefficient copying of data chunks. I implemented a basic pointer system where a for loop runs for a certain amount of iterations, and each iteration pointed to a different char array within the pointer array that the “chunk” (a substring of the total concatenated data) would be copied to. Then, to advance the index of the data I wanted to copy by 5 spaces, I simply added a counter system that would add 5 to itself at the end of each for loop iteration, thus resulting in each char array to have the chunk one after the other.

I also used this day to get rid of much of the redundant code I had from before, especially the serial.println statements that I was using to debug the code. I also ended up getting rid of much of the arrays and variables that I no longer used, which were just taking up space.

Last but not least, I also implemented a custom input function that would come in use for me later, during the setup of the lock, as it allows the user to choose their 10 digit passcode and configures their secret key for Google authenticator.

On another note, the delivery for the new fingerprint scanner arrived today. It was much like the last, and originally didn’t function when I tried wiring it up and running a test script on it. Turns out that the wires were no making good contact and I had to strip the cable and twist it together with the cut end of one of the male to male connectors. I had also incorrectly configured the hardware serial port it used to communicate with the chip and accidentally configured the serial port next to where I had it connected (Since on the arduino mega, Serial1 is pins 16 and 17, which are marked TX2 and RX2 on the PCB).

Other than that, it was not a long day for development, though I did end up having to push the scheduled day of the ASCII to Hexadecimal conversion back 2 days (washing machine broke down, had to replace a ball bearing in the motor).

**Day 7: Base32 Encoding and Key generation (part 4)**

I received the new part, the linear servo, today. However, like many of my ordered parts, it did not go to plan. The linear servo I had ordered turned out to be way too small and was actually meant for RC planes. At this point, I had given up on finding or creating a linear servo to act as a lock for the safe, so I decided to order a 5V solenoid from amazon.

In terms of code, I spent the day trying to find a way to convert an inputted passcode from a char array to an unsigned 8bit integer array that could be used for the TOTP function. I ended up trying to find a method to manually convert the array for hours until I realized that I could have simply just directly converted from char to uint8\_t. This is because in the arduino (as is in any computer), the value is stored as binary and only converted to human readable when called to do so by the human. So, all I simply had to do was to redesignate the array type as uint8\_t (effectively changing the types of the values in the array from char to uint8\_t).

This effectively allowed me to complete the final section of the TOTP and base lock authentication, allowing me to proceed to the next portion of the development, implementing the fingerprint scanner.

**Day 8: Fingerprint Sensor and Actuator**

The fingerprint sensor was coded and setup to run today. It turns out that the fingerprint sensor actually has a built-in separate chip along with enough EEPROM to store around 100 fingerprints. At one point, my arduino started to heat up and turned itself off, which was odd and for me, was worrying. I had initially thought I had fried the chip until I started to inspect the circuitry. The problem revealed itself to be that the VCC and ground wires of the fingerprint sensor were touching where they had been stripped and twisted together with the male to male connecter cables, effectively shorting the circuit. Since I had no electrical tape on hand, I decided to use scotch tape to cover and isolate the exposed wiring. Interestingly enough it worked as a solution.

The Adafruit library was actually very simple to use, as the keywords for the function return integers for different types of states/problems. For example, by setting the default of an integer variable to -1, we can then get an integer output from the fingerprint sensor at each step of the enrollment and comparison process and compare it against preset conditions. Since the FINGERPRINT\_OK condition (where the process has been correctly performed) returns a zero, we can compare it to what we received from the fingerprint sensor as output. If the returned value of the fingerprint sensor is equal to that of the condition, things are going smoothly. However, if it doesn’t equal the FINGERPRINT\_OK condition, it can be compared to other conditions to diagnose the problem.

By using an entire afternoon, I was able to write up some code and implement a way to ask for your name and customize it to your fingerprint. I then combined the script with my lock and TOTP script and tested it to confirm it was working.

I then proceeded to work on the actuator. This was much easier as I simply connected one wire to the ground on the breadboard and the other to the collector pin of a NPN transistor. The base pin was then connected to a wire which led to pin 43 on the arduino mega, which is set as an output. The third pin on the emitter pin was connected to the 5V power rail on the breadboard. When the safe is unlocked, pin 43 is set to a high state and closes the circuit of the solenoid, allowing the bolt to slide back and the lock to open.

I also used the day to change up the way to set the software RTC, allowing it to become fully automated by grabbing the system time at startup.

**Day 9: Code optimization**

I mostly spent the day cleaning up and optimizing the code, getting rid of unused functions, arrays and variables. I also used the day to setup an LCD TFT shield for the project, to allow displaying qr codes and to allow the safe to run without a connection to a computer.

However, the library for the TFT was only compatible with the arduino UNO and the version for the arduino MEGA was outdated and did not function. During this time, my other LCD, the 16x2 module, died and I could no longer get the lcd shield to light back up. I used a multimeter and verified that there was no continuity (reading stayed at 1) between the vcc and vss pins for the backlight of the LCD, meaning that at least that part of the board was fried.

**Day 10: NodeMCU Networking / Blynk**

I used this day to setup the networking and serial communication of the 2 chips. Since the networking chip I am using is the NodeMCU 12-E, it actually is a ESP-8266 Wifi chip soldered onto a standalone custom arduino chip, which actually has a 80MHz processor, which is much higher than the 16MHz clock speed of the arduino Mega and 4MB of flash, which is much higher than the 256kb of the arduino Mega.

However, since the NodeMCU wasn’t compatible with some of the libraries and doesn't have nearly enough pins for my project, I simply used serial communication to allow it to communicate with the arduino Mega while running its own script. The NodeMCU chip was setup with software serial and was connected to the Serial 2 ports of the arduino Mega. Initially, the connection was only tested before I tried using it to send data to my phone over the web.

Blynk was also connected to the NodeMCU chip, which allowed it to directly communicate with my phone over Blynk’s servers, meaning that as long as both the NodeMCU and my phone had access to the internet, I could receive information about my safe’s status anywhere in the world. I also managed to implement a remote locking system to disable the lock if anyone tried to access it and also a remote password changing function as well. On top of that, I implemented a way to receive real time notifications on your phone regarding actions performed on the lock as well as emails.

**Day 11: Finalization / Physical Setup**

On the final day of development, I spent most of my time on the physical creation of the safe itself. I carved out a cardboard box and used duct tape to secure the modules to the sides. I also used the day to optimize the performance of some of the code, cutting out sections that required serial input from the computer as it was slow and glitchy.

One such example was the fingerprint scanner, which kept checking for a response from the fingerprint sensor even after the initial calibration. This caused it to often think the fingerprint scanner timed out, even if it was actually still functioning fine. To fix this issue, I simply added an if loop that only runs the calibration the first time the machine boots up.

I also had to change the time setting on my laptop that would be used for the presentation at school. Since the system time is the basis for the software RTC, I had to change the time of the PC forward 2 minutes as it was initially setup incorrectly and never bothered to correct itself based on network based time.

For the design of the safe, I reorganized some of the modules to be able to fit it all onto one of the doors on the side of the safe and to allow the wires connected to the chips enough space to flex and bend as the safe door closes and opens.

Though quite a few modifications have been made since the initial conceptual idea of the safe, the core functionality and usage of the safe is still relatively similar to the original idea. One of the main changes was moving from a text message to an app in terms of mobile communication. This also changes out the GPRS shield for a Wifi chip.