Garage Door Controller

Final Project

Authors:

Erik Van Schijndel

Jacob Grace

Instructor:

Jay Herrmann

Due Date:

05/10/2024

Prepared For:

ECE 631 Microcomputer System Design

Kansas State University Computer Engineering Department

Introduction

The purpose of this final lab assignment was a cumulative design of all previous labs in our course culminating in a garage door opener that includes NFC card authentication. Our group also included further developments in our Freeboard software displays as well as a physical 16x16 LED screen to display results of authentication as well as the opening/closing of our garage door. In order to accomplish this, our group utilized all of the previous hardware and software from labs 3 through 8, and tied them together through our MQTT broker in order to achieve our final designed result.

We used several pieces of hardware to configure and validate our final design and implementation which includes the following:

- 2 Raspberry Pi 4s (RPi4)
- 2 NodeMCU ESP32 Development boards (ESP32)
  - ESP32 Onboard blue LED
- 2 SS49E Linear Hall-Effect Sensors (HES)
- 1 RCWL-1601 Ultrasonic Distance Sensor (UDS)
- 1 PN532 NFC/RFID MODULE V4 (PN532)
- 2 NFC Cards
- 1 32x16 Freetronics Dot-Matrix-Display (DMD)
- Macbook & Windows Laptops
- Jumper Wires
- Digilent Analog Discovery 2

Similarly to Lab 7, our RPI4s were configured to the “ece631Lab” wireless network to independently host two separate MQTT servers, one hosted Erik and one by Jacob, for subscribing and publishing NFC authentication, NFC User IDs, UDS distance values, HES hysteresis values, RAG states/flags, LED flash rates and garage door states. Erik’s RPi4 also contains the “Spring24Final_NFC_UID.py” python program containing the logic to initialize and validate our PN532 in order to authenticate our two stored NFC cards. This python program was also configured to automatically run at boot of the RPi4 using supervisord’s config files. The first of our ESP32 boards was configured with the majority of our C code and was wired with jumper cables to accommodate the UDS and HES, while also controlling the onboard blue LED in order to display our garage door state visually through modified flash rates. This ESP32 was also used to determine the garage door’s state itself based on the NFC authentication messages received from our MQTT broker. This ESP32 also sends the hysteresis, distance, LED, and garage door state values to Erik’s MQTT broker. All of these devices were used in conjunction with our MQTT broker in order to display the status of these components visually on freeboard. Erik’s ESP32 was connected via USB to Erik’s Macbook in order to program the ESP32 as well as display the MQTT broker and freeboard visualizations of incoming MQTT JSON data. The second ESP32, which will be noted as Jacob’s ESP32, was used to host the Arduino code for the second HES, as well as the control logic for our 32x16 LED DMD. Jacob also connected his ESP32 via USB to his Windows laptop in order to display his own MQTT broker. Just as in Lab 7, our group also used the Digilent Analog Discovery 2 in order to measure and validate proper PWM waveforms from Erik’s ESP32.

The software utilized in our group's final implementation is as follows:

- Arduino Integrated development environment (IDE)
- Linux operating system
- Nano editor for python
- Apple Terminal application
- PWM Libraries for ESP32
- HiveMQ MQTT broker
- MQTT Communication Protocol
- Freeboard dashboard
- MQTTool iOS Application

The software in this implementation is spread over multiple devices and languages. The Arduino IDE was again the environment in which code for the PWM generation, UDS distance calculations, MQTT subscriptions & publishings, hysteresis value calculations, and LED flash rate modifications are housed on Erik's ESP32. This same environment was used to host hysteresis value calculations, MQTT subscriptions & publishings, and LED Matrix control logic on Jacob's ESP32. Erik's Arduino sketch was written in the C coding language and is saved as "Spring24Final_Erik.ino" in the GitLab repository included in our appendix section below. Jacob's Arduino sketch was written in the same language and is saved as "FinalTest.ino" in the same GitLab repository also included below. The Linux operating system was utilized on both RPi4s to host our MQTT brokers respectively. The Nano editor was utilized on Erik's RPi4 in order to create a python language-based program to initialize and control the NFC card authentication for both stored NFC card values. This aforementioned program is written in python and is saved as "Spring24Final_NFC_UID.py" in the GitLab repository linked below in the appendix section. Erik utilized the Apple terminal program to connect to the RPi4 and write the above listed programs and validate proper communication of our MQTT messaging. Similarly to Lab 7, the PWM libraries for Erik’s ESP32 were used with functions calls such as “ledcSetup()”, “ledcAttachPin()”, and “ledcWrite()” in order to configure the pin outputs and channels for use with our UDS. We also utilized Write for generation of the PWM signal for our UDS sensor. HiveMQ was used as the central hub for data exchanges in our project design utilizing the MQTT messaging protocol. Two servers were used, with an IP of 192.168.1.125 for Erik’s MQTT broker, and 192.168.1.109 for Jacob’s broker. As mentioned previously, Erik’s broker handled the NFC authentication, NFC User IDs, UDS distance values, HES hysteresis values, RAG states/flags, LED flash rates and garage door states from Erik’s ESP32. Jacob’s broker was utilized to handle the HES hysteresis values from Jacob’s ESP32. These two brokers were utilized together to gather information on the Freeboard dashboard that was hosted on Erik’s RPi4. This Freeboard served to display the status and data from all of the aforementioned hardware and software components for a convenient view of all of the devices and the state of our garage door. To accomplish this, specific device-dependent data sources were added to the Freeboard dashboard to support each device and simplify data source utilization. This results in eight total data sources for our Freeboard implementation, defined as NFC Access, NFC UID, Hall Effect Sensor (Erik), RAG State, LED, Garage Door State, RAG2 (Jacob), and Distance Sensor as found in the “dashboard.json” file included in our GitLab repository below in the appendix section. We also utilize the “MQTTool” iOS application as a convenient method for a user to send a simple “Close” message to the "ece631/FinalProject/NFC" topic in order to close their garage door when in an open state. Note that the user does not need to format this close message as a JSON string when sending to the MQTT broker.

<u>Design Partitioning</u>

There are two primary divisions of our design which have been outlined below. These two primary divisions are the hardware and software involved in our final implementation. Each of these divisions have been expanded upon below with both partner's design philosophy and role in the functionality in our final design.

Hardware:

As listed before, the hardware used in this design was divided between the two group members, Erik and Jacob. The following chart in figure 1 is an image of the hardware utilized by Erik in this design.

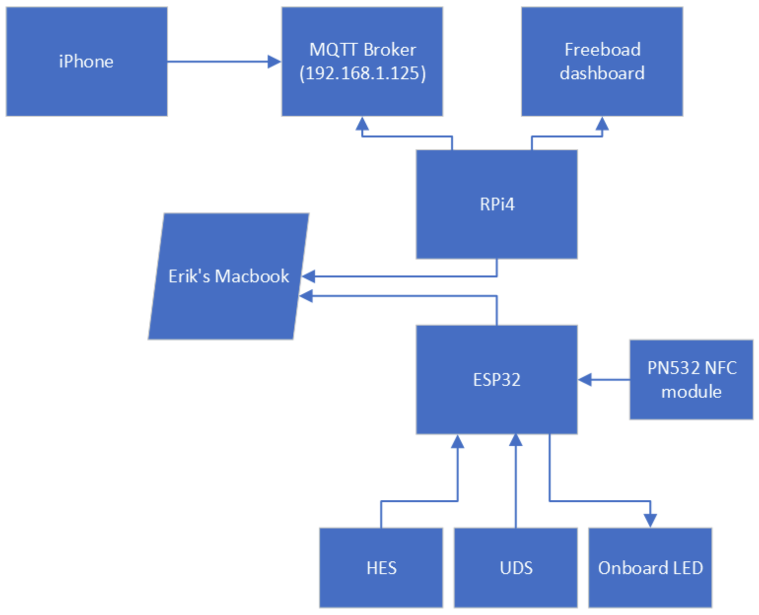


Figure 1: Erik's hardware partitioning in our final project.

As can be seen, there are a total of 8 physical hardware devices used in Erik's portion of the hardware design. These are the HES, UDS, onboard LED, ESP32, PN532 NFC module, RPi4, iPhone, and Macbook. This was chosen in order to consolidate most of the combined previous lab operations and to simplify the wiring that would be needed in order to initialize the use of the LED matrix on Jacob's portion. It was believed initially that our LED Matrix would need more connections than would be available with a further divide in components and made the most sense to our group to proceed.

The second portion of our hardware design partitioning contains Jacob's hardware configuration. This has been presented below in figure 2.

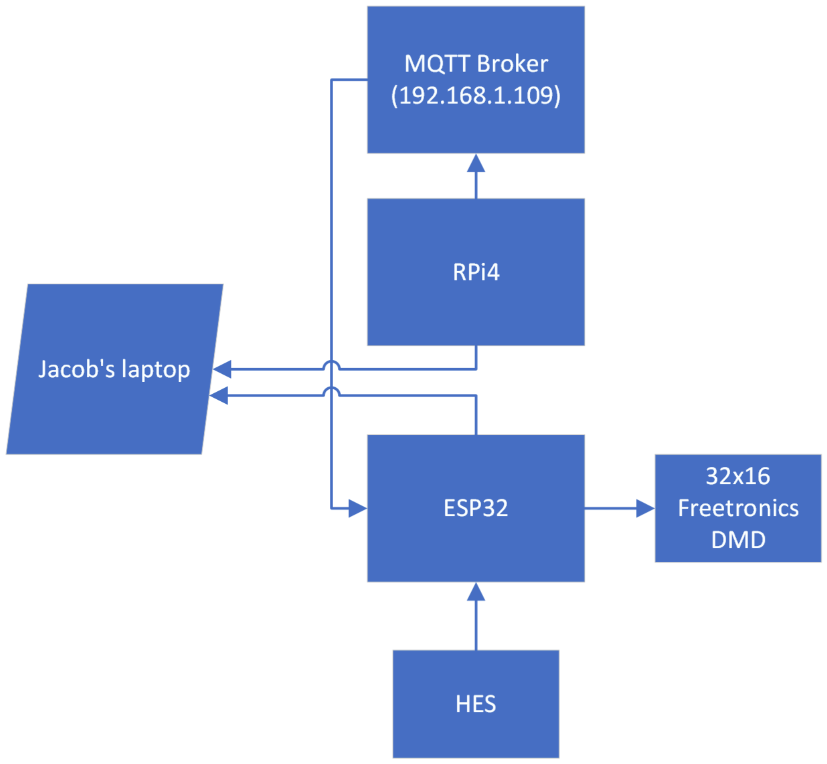


Figure 2: Jacob's hardware partitioning in our final project.

As can be seen above, there are a total of 5 hardware components present in Jacob's hardware implementation. As previously mentioned, this was selected in order to consolidate all of the jumpers needed and ensure we would have enough GPIO pins for our LED matrix for our final design. This also simplified communication of states for use in changing our LED matrix outputs, as Jacob's ESP32 is able to subscribe and receive the JSON messages of the authentication of NFC cards and garage door states in order to change the display of our LED matrix.

Code:

Following the same structure as the above partition section, the code design partitioning will be split among Erik and Jacob's code designs, describing each file written. The first file to expand upon is the Arduino sketch for Erik's ESP32, named as "Spring24Final_Erik.ino" and outlined below in figure 3.

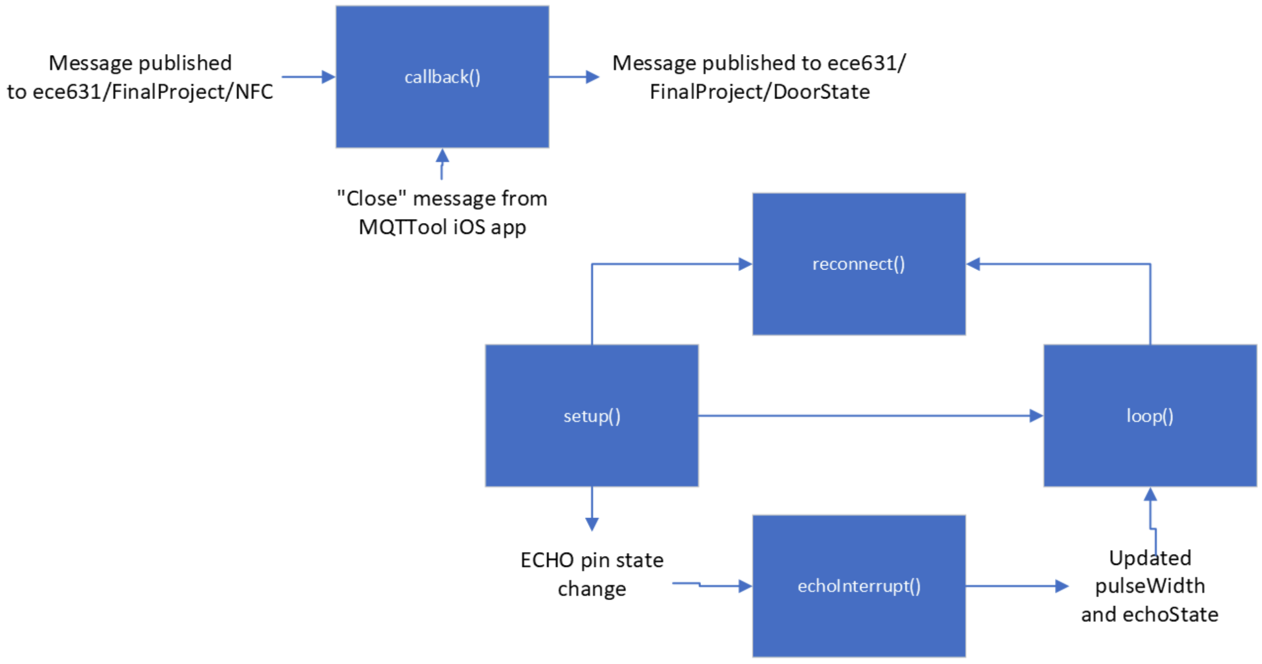

Message published to ece631/FinalProject/NFC
callback()
Message published to ece631/FinalProject/DoorState
"Close" message from MQTTool iOS app
reconnect()
setup()
loop()
ECHO pin state change
echoInterrupt()
Updated pulseWidth and echoState

Figure 3: Flowchart of "Spring24Final_Erik.ino" in our final design.

As can be seen above, there are four primary sections in the aforementioned .ino file. This is similar to the Lab 7 implementation, but contains a slight modification to the reconnect() function, far more control logic in setup() and loop(), and the addition of the callback() function. These are expanded upon below:

- **callback():** This void function was reused from our Lab 5 implementation and is tasked with receiving any messages from the "ece631/FinalProject/NFC" topic for use in changing our garage door state. This function deserializes the JSON message received and checks that it belongs to the NFC topic, before storing to a String variable for use in further checks. The “nfcAuth” string is then used to check that the message received contains an “Authorized” message while the garage door is closed, before storing an opening message to a doc and serializing it for publishing to our "ece631/FinalProject/DoorState" topic and changing our garage door state. If denied, we then store a “closing” message to a doc and serialize it for publishing to the same topic and updating the garage door state. If the input doc does not contain “NFC”, we then store the message as a string called userText, which stores the payload as chars. We then check for a “Close” string input when the garage state is also “Open”, before again storing a “Closing” message and serializing it for publishing to the same DoorState topic and updating our garage state variable.
- **reconnect():** This function was reused from Lab 5 and is called during setup() and in the main loop if connection fails. This is used to connect our ESP32 to the wireless lab network before sending an MQTT message as validation of its success. It has been lightly modified to also subscribe to the "ece631/FinalProject/NFC" topic for use in the above callback() function.
- **setup()**: This function is used to set up our PWM functions, pinModes of the HES, UDS, and LED output. It then saves timers for the HES and UDS, before initializing our state variable for the UDS and attaching our echoInterrupt() function to the defined ECHOPIN. We then calculate the HES offset from pin 36. We begin our WiFi connection afterwards, before serial printing the successful connection IP address. We also set the predefined MQTT server and callback function before calling reconnect(). This function then ends by publishing a

"Closed" message to our MQTT server to initialize the Freeboard display as closed.

- **loop():** This is the main loop of our sketch which initially checks for wifi connection, and calling reconnect() if it fails. We then initialize sum and index variables before calculating the newest sum and storing our measured pulse width in our moving average filter (MAF). This implementation functions as a circular buffer which stores the 5 most recent distance measurements from our UDS to calculate the moving average. The size is defined as "WINDOW_SIZE_UDS" which can be changed to improve/decrease the smoothness of our filter. We use index to keep the current position within our buffer while also overwriting the oldest measurement present. We update this in a circular fashion using "index = (index + 1) % WINDOW_SIZE_UDS" and set our state machine variable to 0 to indicate a completed measurement. After this, we then check that one second has passed before measuring the distance traveled from microseconds to inches. After the calculation, we round the value to two decimal points before serializing our measurements to be sent across our MQTT communication with the topic as "ece631/Lab7/Distance/SensorID/0" before incrementing a timer. The previously mentioned formula for the distance calculation is as follows:

$$distance = ((sum / WINDOW\_SIZE) * 0.0135039) / 2$$

We then check that a second has passed before updating our timer. We store the analogRead() value of pin 36 and subtract the offset as our value from the HES. After this we use a READINGS array with a size defined as "WINDOW_SIZE_HALL", which is 1 in our case. We subtract this array at "INDEX" from the current sum value and update "VALUE" with the previously read value from the HES. We then update our READINGS array with this newly read value, before updating the index in a similarly circular fashion to above. This is done as: "INDEX = (INDEX+1) % WINDOW_SIZE_HALL;". We then save the hysteresis value as:

$$HYSTERESIS = SUM / WINDOW\_SIZE\_HALL;$$

before updating the hysteresis value with the above calculated one. We then clear our hysteresis doc object and update the message to be the above calculated value. We then enter a conditional check that the hysteresis value is greater than 350 and the garage state is "Opening" in order to emulate a magnet reaching the end of a garage door track. This then updates the garage state to 3 before storing the new state to our garage door doc and changing our LED flash rate to 100 ms. We then publish this to the MQTT broker under the "ece631/FinalProject/DoorState" topic. Conversely, if the garage state is "Closing" and our hysteresis value drops below 50, we then update the door state to "Closed" before storing it to the garage door doc and updating the flash rate to 1000 ms. We similarly publish this garage door state to the same topic as above. We finally check that our defined LEDMillis variable is less than our flash rate, before storing a new LEDMillis time from millis() and flipping the LEDState variable and using digitalWrite() to flip the blue LED's state from on/off. We then store the flash rate to a flash rate doc object and and publish it to the "ece631/FinalProject/LED/Flashrate" topic to be used in our dashboard.

echoInterrupt(): This function was unmodified from my Lab 7 implementation. This interrupt service routine (ISR) is configured to be called whenever there is a state change in the ECHO pin from our UDS. We save the current time initially before reading the current state of the ECHO pin. We use the result of this to determine the difference between a rising edge/falling edge. If we detect a rising edge, we only save the current time. If we detect a falling edge, we then save the current time and set the value of our state machine to 1 before calculating the distance of the pulse by subtracting the rising edge from the falling edge. This calculated pulse width is used above in our MAF as a value for calculating the distance traveled by the UDS ping.

Our design mentions a state machine, which was used in our design in order to ensure independent operation. This state machine has two states:

- **State 0:** In this idle state, “echoState” equals 0 and represents waiting for an echo signal.
- **State 1:** In this state, “echoState” equals 1 and represents a complete signal having been received. This allows us to calculate our distance measurement, filter it, and publish the distance using MQTT.

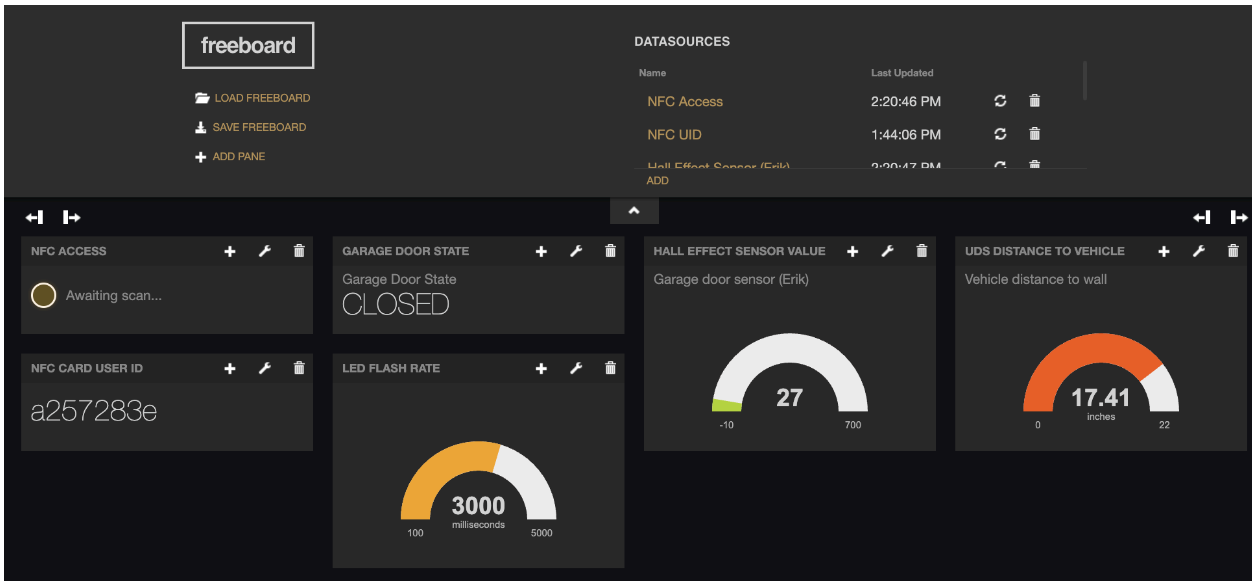
These states are set in the main loop and ISR respectively, and help to ensure consistent and independent function of our code.

This setup made the most sense for most of our control logic for our hardware devices interfacing with the ESP32. It utilized a similar flow and design to our previous lab implementations and allowed for a smooth transition to a multiple device setup. I did not change any initialization from the initial labs from which they were pulled in this setup, other than the primary conditional logic of the callback() function as it pertained to our NFC authentication. We utilized the same window size for the UDS that was found to be the most consistent, and the same window size of one for the HES in order to retain the consistent values read from lab 8.

The second code file pertinent to our design was the use of a python file saved as “Spring24Final_NFC_UID.py” and written by Erik. This file was simply modified from our Lab 4 implementation to store Erik and Jacob’s found UID codes ("a257283e", "5d11273e" respectively). We initialize a bool called “correct” to false for use in conditions of authentication. We begin with a try-except block with an attempt of connecting to our MQTT server, with an exception for ConnectionRefusedError if failure is found. After we connect, we enter a second try block that initializes and stores our MQTTPub client, the PN532 library, the device firmware, and the SAM configuration of our NFC cards. We then store the current time, and initialize a pub timer to 0. We then enter a main While() loop, where we measure that 5 seconds have passed before recording and publishing a heartbeat to identify our software is working. We also store RAG and RAGState JSON messages for use in freeboard, where the RAG indicator flashes yellow while waiting for a card to scan. We then record our last heartbeat as the current time. If no card is swiped, we continue before checking that a second has passed before the last publishing time. If so, we store the user ID and check that this found user ID matches the two stored values of user IDs in the initial array. If so, we construct multiple JSON messages containing an “Authorized” message and updated RAG JSON components for changing the lights to green and indicating success. We then publish this data to our MQTT broker. Similarly, if the card is not found, (correct == False), we then construct multiple JSON messages containing a “Denied” message and updated RAG JSON components for changing the light to red and indicating failure. We then publish this data to our MQTT broker.

The above program was configured using “supervisord” in order to launch on boot of Erik’s RPi4. This decision was made in order to maintain continuity from the previous lab implementation as the design and flow of the authentication made intuitive sense. We also opted to keep the other sensors out of the python implementation for simplicity of the authentication code.

The third code file pertinent to our design is the saved dashboard configuration of our Freeboard application hosted on Erik's RPi4. This file was created by Erik and is saved as "dashboard.json" in our GitLab repository. This file contains 8 data sources that are paired with each of the 7 "widgets" that display data from each device paired through our MQTT broker. The above widgets and data sources have been shown below in a screenshot of our current Freeboard dashboard configuration below in figure 4:



Name	Last Updated
NFC Access	2:20:46 PM
NFC UID	1:44:06 PM
Hall Effect Sensor (Erik)	2:20:47 PM

Figure 4: Freeboard implementation of above widgets/data sources

The above implementation allows us to display all relevant user and device information while parsing MQTT broker messages to relevant devices for state changes. This was decided to be hosted on my laptop as I was the first member of the group to get the RAG widget to properly work on my machine. The types of widgets themselves were rather arbitrary in selection and mainly served as the type of data that we were receiving. As can be seen above, the widgets themselves are organized by the device/state they are beholden to.

The second portion pertains to the code that Jacob has written for his ESP32 and LED DMD combination. This is in the file “FinalTest.ino” as saved in the GitLab repository. This code currently has a few functions, outlined below:

- **triggerScan():** This is an ISR based on the hardware timer initialized from the ESP32 to be utilized for the LED DMD.
- **setup()**: Called during the initial run of the program. Currently utilizes the DMD Arduino library to initialize a timer based on the CPU clock of the ESP32, attach that timer to the above triggerScan ISR, write to it, and enable the timer. This setup also currently initializes the HES pin and collects data to calculate the magnetic offset for the HES.
- **loop():** In this function, an HES value is read with analogRead() from a previously set pin and used in a MAF similarly to Erik’s HES implementation to collect and display the HES values. After these values are collected and stored to an array, they are printed once a second and are compared amongst set integer values in a switch statement in order to determine and modify the LED DMD based on the distance to the vehicle. These modifications are in the form of the setString() commands that differ based on measured distance. In this switch statement, these vary as “DRIVE” for anything less than 50, “SLOW” for anything between 50 and 500, and “STOP” for anything greater than 500. These values are similarly compiled in JSON format messages to be utilized among Erik’s Freeboard dashboard in a similar fashion to the above string messages.

The second code file relevant to our design created by Jacob is saved as "Rag2Sender.ino" and is saved in our GitLab repository. This file is currently used to collect and send data for the RAG2 widget used in Erik's Freeboard. The functions relevant in this code have been outlined below:

- **setup_wifi():** This function is called during setup() and is used to configure a connection to Jacob's MQTT broker.
- **reconnect():** Functionally similar to the above description, this function is called during loop() under the condition that the initial MQTT broker connection has failed.
- **setup()**: Called during the initial run of the program. Initializes HES pin and creates a client instance for the host's MQTT broker. Also collects an offset used for reading the HES hysteresis values.
- **loop():** In this function, a HES value is read with analogRead() from a previously set pin and used in a MAF similarly to Erik's HES implementation to collect and display the HES values. After these values are collected and stored to an array, they are printed once a second and are compared amongst set integer values in order to determine a RAG value to be published to Jacob's MQTT broker. These colors/values are determined in the event of a car approaching a wall, where green is clear, yellow is a car approaching, and a red for a car approaching too closely to a wall.

Design Issues / Limitations

There were many issues encountered that our team needed to overcome in our design. A primary concern of our team prior to the finalization of our design was the wiring available on our ESP32 devices for use of all of our prior lab sensors and for our LED Matrix. After determining that 5V input was valid for our UDS, we were able to proceed forward with further testing. We encountered several issues initially with our compilation of the previous lab codes into one Arduino sketch, but these were also repaired and validated to match original lab outputs.

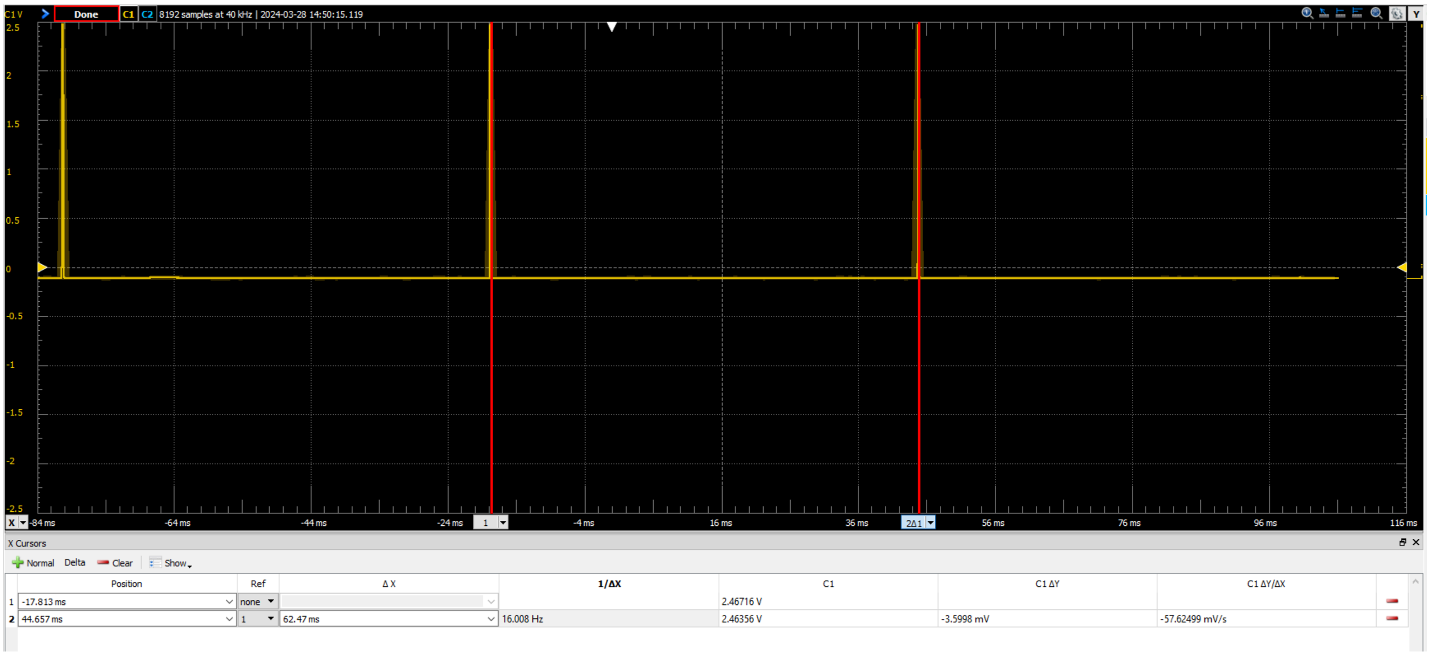
A second primary concern was customizing our Freeboard configuration with the third-party RAG indicator widget. The initial instructions presented on the public repository for the RAG indicator were found to be a bit unclear initially. This was overcome and implemented on Erik's RPi4 for use in the final design.

A third design issue presented was use of the LED DMD. This included initial steps to configure the device while also having to wait for new power supply materials to be received from the mail. The shipping took longer than expected, leaving us with scant time for implementing the device into our final design. Another issue present with this LED DMD included providing a proper amount of GPIO pins for use in configuring the device. In order to remedy this our team decided that Erik could use the previous lab devices and connections on his RPi4 and ESP32 which gave Jacob more access to GPIOs required for the screen. A third problem that was encountered occurred when powering the LED DMD the first time for initialization, as it is currently believed the DMD was partially fried from supply voltage/current during initial testing. This left the DMD with the ability to be powered on, but took away the ability to control the LEDS of the DMD themselves, essentially destroying our LED DMD for our purposes. This DMD was still utilized in our designs as aa key output, but did not provide any functionable actions due to the aforementioned damages caused by our chosen power supply.

Testing

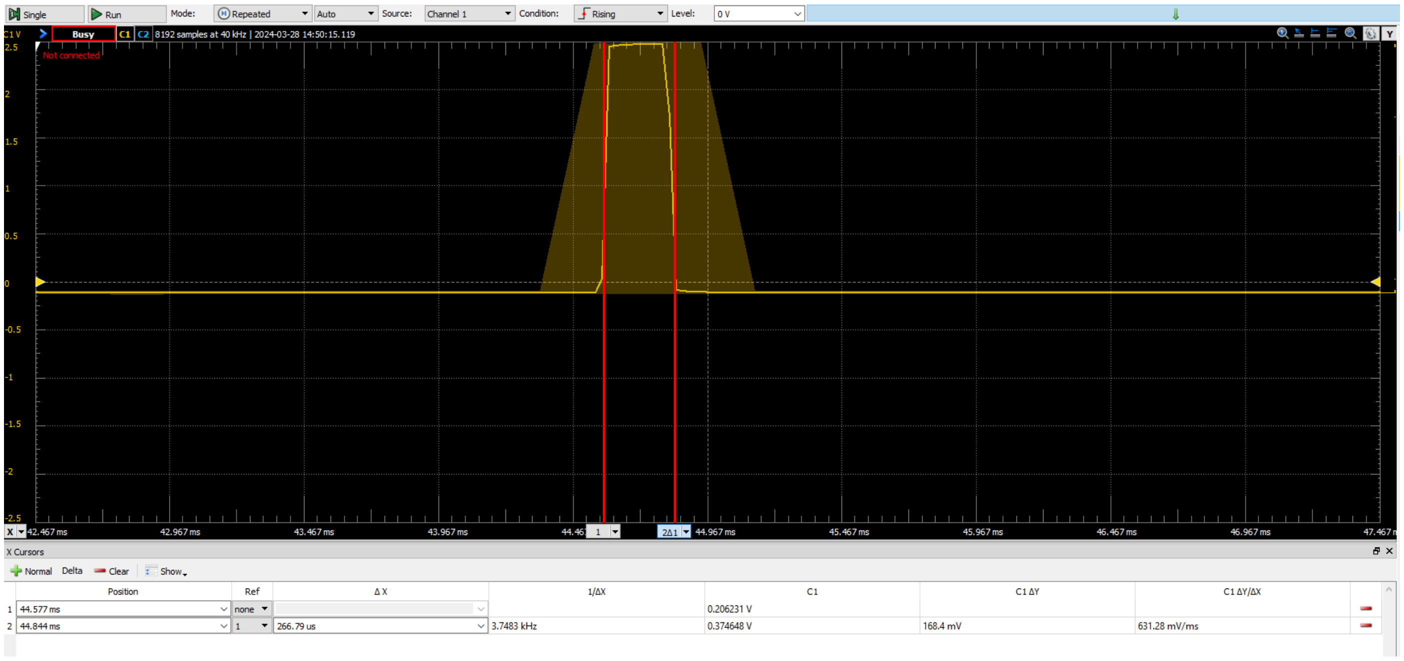
The first test case for our design was ensuring valid parsing of our NFC card authentication. This was done with the help of fellow students in the lab by use of their NFC cards. We were able to validate that NFC UID strings parsed by our python code did differentiate and confirm the stored UID strings as authenticated entries.

Similarly to lab 7, our second test case involved validating the PWM of our UDS sensor for use in the final design. Our specifications provide that we needed a duty cycle of >60 ms, and a pulse of >10us. We have provided the Waveforms program's visualization of our PWM below in the following figures 5 & 6. As can be seen in the figures, our PWM again operated successfully as the duty cycle was measured to be 62.47 ms and the pulse was measured to be 266.79 us.



	Position	Ref	Δ X	1/ΔX	C1	C1 ΔY	C1 ΔY/ΔX
1	-17.813 ms	none			2.46716 V		
2	44.657 ms	1	62.47 ms	16.008 Hz	2.46356 V	-3.5998 mV	-57.62499 mV/s

Figure 5: Waveforms program showing measurement of duty cycle to be ≈62.47ms



Mode: Repeated | Auto | Source: Channel 1 | Condition: Rising | Level: 0 V

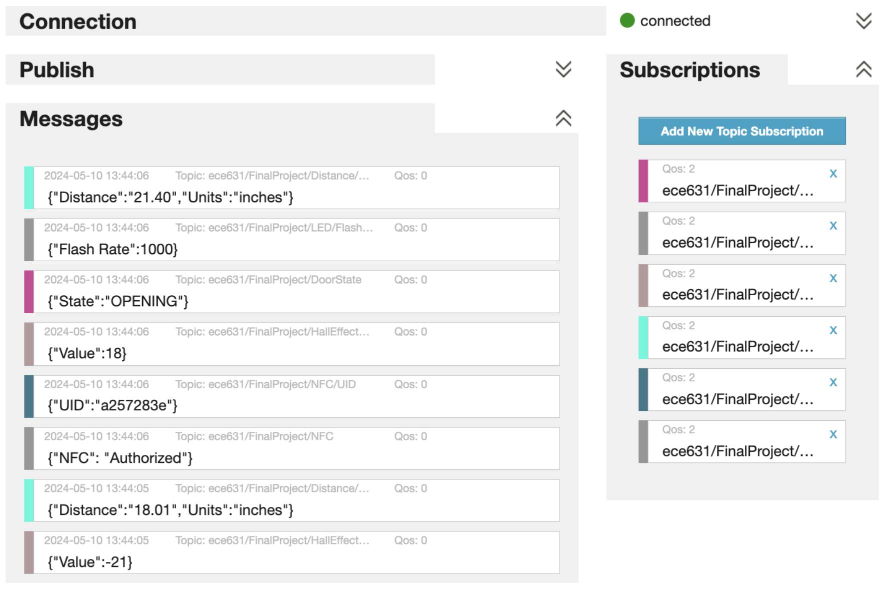
8192 samples at 40 kHz | 2024-03-28 14:50:15.119

	Position	Ref	Δ X	1/ΔX	C1	C1 ΔY	C1 ΔY/ΔX
1	44.577 ms	none			0.206231 V		
2	44.844 ms	1	266.79 us	3.7483 kHz	0.374648 V	168.4 mV	631.28 mV/ms

Figure 6: Waveforms program showing measurement of pulse signal to be ≈266.79us

The third test case was viewing serial outputs of our ESP32 devices to ensure proper function of the LED, the HES of both devices, and UDS. These output statements were acceptably similar as compared to our previous implementations. One key note to remember is that my implementation of the UDS distance calculations still contains a slight deviation to standard values, resulting in few deviations of +/- 2 inches when measuring a distance further than 3 inches.

The fourth test case was ensuring that all topics in our MQTT broker were successfully receiving and publishing required data from our software and hardware implementations. This was validated from each topic individually, tracking the resulting data from measured distances, hysteresis values, and NFC UID strings and authentication statuses. A figure of this validation has been provided below in figure 7.



```
2024-05-10 13:44:06   Topic: ece631/FinalProject/Distance/...   Qos: 0
{"Distance":"21.40","Units":"inches"}
2024-05-10 13:44:06   Topic: ece631/FinalProject/LED/Flash...   Qos: 0
{"Flash Rate":1000}
2024-05-10 13:44:06   Topic: ece631/FinalProject/DoorState   Qos: 0
{"State":"OPENING"}
2024-05-10 13:44:06   Topic: ece631/FinalProject/HallEffect...   Qos: 0
{"Value":18}
2024-05-10 13:44:06   Topic: ece631/FinalProject/NFC/UID   Qos: 0
{"UID":"a257283e"}
2024-05-10 13:44:06   Topic: ece631/FinalProject/NFC   Qos: 0
{"NFC": "Authorized"}
2024-05-10 13:44:05   Topic: ece631/FinalProject/Distance/...   Qos: 0
{"Distance":"18.01","Units":"inches"}
2024-05-10 13:44:05   Topic: ece631/FinalProject/HallEffect...   Qos: 0
{"Value":-21}
```

```
Qos: 2  ece631/FinalProject/...  x
Qos: 2  ece631/FinalProject/...  x
Qos: 2  ece631/FinalProject/...  x
Qos: 2  ece631/FinalProject/...  x
Qos: 2  ece631/FinalProject/...  x
Qos: 2  ece631/FinalProject/...  x
```

Figure 7: MQTT broker validation including NFC UID/authentication, HES values, garage door states, LED flash rate, and UDS measurements.

Our final test case involved validated proper parsing of data in our Freeboard dashboard design. We did this by individually toggling different widgets to ensure the data parsed was being utilized correctly. The largest issue arose with use of our RAG indicator but this was mostly due to being new to the widget's data requirements. The design's validation can be seen above in the previously shown figure 4.

Conclusion

This project required many different iterations and decisions for what would be included and what would need to be left out. There are many different options on the table for what can be done to improve the overall project's efficacy. A major component would be time dedicated to the configuration of our third party modules. Many of these were started later than they should have and resulted in a few last-minute issues that could have been corrected prior. Another method to potentially improve this design could be to find the root cause of the issue of Erik's UDS sensor calculations, as the deviations present in lab 7 still caused a few issues for further use in our design. Another point to consider for potential improvement could be further considerations of feature creep in our design. We began this project with a great deal of confidence and assumptions about what could be added to our final design before really finalizing a solid base to grow outwards from, and we believe this caused a few issues in our final project's design, such as the present mishap with our LED DMD.

Appendix

All code used in this project and design can be found below in the following link to our team's GitLab repository:

https://gitlab.beocat.ksu.edu/s24.jarik/spring24final