Final Project

MQTT LOCK

ECE 631 - Jay Herrmann

By:

Lock Team

Hamza Ahmed and Jan Mrazek

Introduction

For our final project for this semester's ECE 631 course, Microcomputer Systems Design, we decided to create a conceptually simple remote locking mechanism. During the design and development of this project, we used the knowledge we gained from the lectures and labs during this semester.
Our remote lock project utilized two STM32F4 discovery boards, two Raspberry Pi Zero W microcomputers and a Raspberry Pi 3 Model B microcomputer. As for the software and programming part of our project, we used the C programming language, created and parsed JSON strings and utilized the MQTT protocol in order to send messages among different agents and clients. We also made use of some more complex programming techniques, such as handling of hardware interrupts or circular input and output buffers.

Concept

The entire project is made up of three separate IoT devices. One of these devices is the electronic lock itself, the counterpart to which is a "key device" of sorts. The third part is a server that works as a message-passing agent between the two aforementioned devices.
The "key" component of this project is made up of one user push-button and four LED lights. Three of these LED lights serve the purpose of signalizing when the device is idle, a message is being sent and when a message is being sent. The fourth LED signalizes the status of the lock.
When the user push-button is pressed on the key component, an MQTT message is generated and sent to the MQTT server. This message is a custom JSON string with appropriate commands and status messages contained in itself. The message is also sent on an appropriate MQTT channel and topic, which allows for the server to serve multiple different IoT systems at once without any interference being introduced between them.
Once the MQTT server receives a message, it broadcasts the message along to other devices subscribed to a given channel and topic.
The "lock" component of this project listens to the MQTT channel. Once an appropriate command is sent from the key to toggle the lock and the server

broadcasts this command to the lock component, the lock's engagement status is flipped. The lock then broadcasts its own message via the MQTT channel that informs other agents about its status.
The lock's status is signaled with an orange LED light on the lock itself. Once a status message is posted and received by the key, it too signals the lock's status using an orange LED light as well.

Hardware

For both the lock and key parts of the projects, we used an STM32F4 discovery board in combination with a Raspberry Pi Zero W microcomputer. The discovery board was used to handle all commands and message passing, while the Zero W was used strictly as a bridge between the Discovery board's UART IO and the MQTT server's WiFi connection
For the MQTT server, we used a Raspberry Pi 3 Model B.
We also needed a WiFi router to complete the communication circle. We alternated between using a WiFi AP created on a cell phone and Mr. Herrmann's WiFi router that was used for the lectures and labs during the semester. The table below shows which type of communication was used by each device in the project to communicate amongst them. You can also take a look at the prototype diagram below to get a better idea of how the communication is performed.

Device Name	Communication Method
STM32F407VG Discovery boards	Serial communication via GPIO
Raspberry Pi Zero	Serial via GPIO + Wireless Communication
Raspberry Pi 3 B	Wireless communication

Software

During the code development phase of our project, we utilized the C programming language exclusively.
We made use of a JSON library that was provided to us by Mr. Herrmann called ECE631JSON. The library itself is based on another C-JSON library called JSMN. We used this library to effectively parse JSON strings into messages and then parse individual tokens out of them.
We also used parts of the source code we created for the sixth lab, which included an implementation of a circular buffer which was used to store incoming and outgoing messages.
As for the development environment, we used the System Workbench for STM32, which is based on Eclipse. It is a comprehensive software development solution that lets us develop for ARM based microcontrollers which also includes all components needed for creating, building and debugging embedded applications.

JSON

JSON, or Javascript Object Notation, is a formatting technique that enables programmers to send objects encoded in simple strings in a key-value fashion. Although it is not the most effective way of serializing data, it is probably the most widely used standard. It is compatible with all platforms and programming languages that can manipulate strings, as long as an encoder, decoder and parser function is implemented in some way.

Circular buffer structure

A circular buffer is a data structure that is very similar to a queue with a fixed length. It is circular in a way that if elements are taken out of the front of the buffer, the rest of the elements within the buffer are not shifted forwards, but the positions (indexes) of the head and the tail of the buffer are shifted instead. This technique helps reduce the number of memory accesses and the structure's memory complexity, especially when the items stored in the buffer are very memory-intensive, though this was not the case in our labs and project.

The element indexes are ordered in a circular fashion, meaning that elements can be added past the end of the array (presuming the buffer is not simply completely full) - they are placed back to the beginning of the array, if there is any space left.

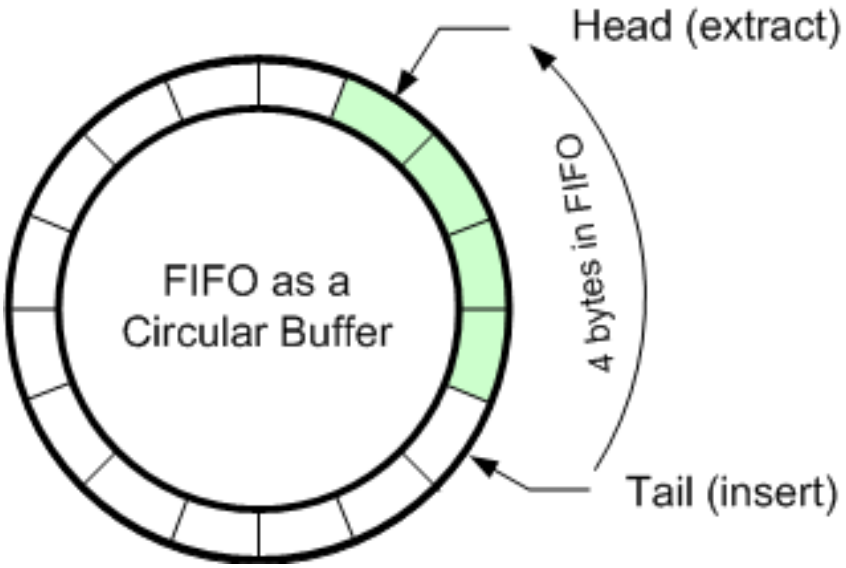


Image source: chegg.com

We had already developed an implementation of a circular buffer structure in one of our labs. This enabled us to simply include the header file of our previous implementation and use without any problems.

Interrupts, sending and receiving messages

The codebase of our project included simple protocols that would enable us to receive and send messages, as well as catch various interrupts produced by the STM board's hardware and firmware.
We made use of USART-related interrupts from the device to parse and store incoming messages. These messages were stored in an RX circular buffer.
When a complete message (ending with a newline) was received successfully, it would be taken out of the incoming circular buffer and sent to a method that would take parse the JSON string contained within the message, and depending on what type of message was received, would perform other steps.

When an outgoing message was to be produced and sent out, it was stored in a TX circular buffer. When such a message was stored in the buffer, an interrupt was enabled that would signal the TX buffer's non-empty status and in turn empty it onto the serial communicator.

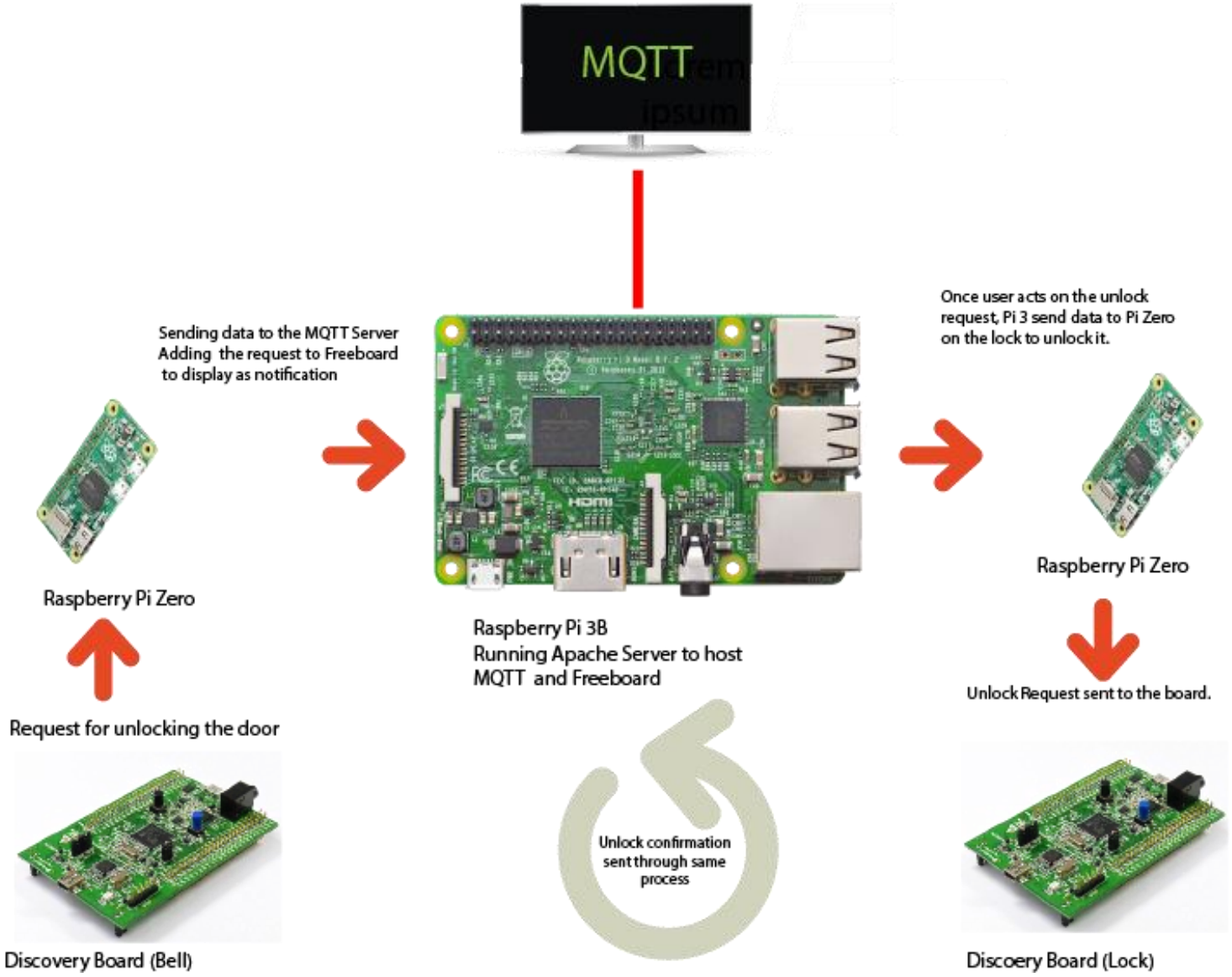
Input

For this project, we required two forms of input. A physical input and a digital input. We used the user push-button on the STM board as a physical input. For the digital input, we used the Arduino IDE's serial monitor in combination with a USB to UART device at first to interact with the board and test various functionalities of the hardware. Using this, we could send commands to the STM board. This enabled us to conduct testing when working separately on the lock and doorbell programs. We later abandoned the USB-UART device when all debugging was done, and we had a reliable serial protocol.

Display

Outputs were a vital part of our application as we used them to receive feedback of our operations. We used four LED lights on each of the STM boards for debugging purposes. Once again, we used Arduino IDE's serial monitor feature combined with a USB-UART device to receive feedback from the serial connection to the STM board at first, and later abandoned it.

Design prototype



Programming

Request

For this project, we were basing our codebase on code we wrote for the sixth lab, and expanded on it. We made use of the circular buffer code, USART interrupts and JSMN library for parsing the JSON (Java Serial Object Notation) strings into messages. This enabled us to serialize the requests to send from the bell to the lock where it gets deserialized. Then the request gets processed and the output results are serialized and sent out.

Communication

Once we got the basic communication between the two discovery boards working, we wanted to make the communication between the two boards wireless. For this part of the project, we used two Raspberry Pi Zero W microcomputers (RPI-Zero). We connected each RPI-Zero to the GPIO Pins on each of the Discovery boards. The program running on the STM Discovery boards contained a very simple state machine with 5 distinct states; 4 of which were used for the WiFi and MQTT setup of the Pi Zero W itself. Every time another configuration step was completed (verified by monitoring the Zero W's serial output and parsing the IP address and status messages), the state machine transitioned into the next state. The fifth and final state transformed the STM board into the main part of the whole project – the lock and key components.
These boards were programmed with an automated Wi-Fi setup script, that, when started, first sends messages to the RPI-Zero itself to configure network settings and MQTT subscriptions.
Next, we used a Raspberry Pi 3 Model B to setup an MQTT Server to handle communication between the Bell and Lock. Once the MQTT server was setup, we had the PI-Zeros publish and subscribe to a topic called lock. This enabled us to automate the entire process of network setup and make the whole project essentially plug and play.

Troubleshooting

While testing the automatic setup feature we ran into a log of errors. In our first iterations of the project's code, we had to manually set up the PI-Zeroes before we could start communicating between the STM boards. Where we had to manually connect the PI-Zeroes with the use of a USART cable, and we had to make use of a serial monitor to enter the commands for Wi-Fi setup and to subscribe to the MQTT Server to post messages. The next biggest issue that we ran into was with an old version of the ECE631JSON library, which had a multitude of bugs that prevented us from parsing JSON strings properly. That took a lot of effort to figure out.

We also encountered an issue where if the user push-button was pressed briefly, it would generate hundreds of interrupts. We fixed this issue by developing a custom debouncing method that would only allow the button-press to be registered once every 1000 1ms ticks, or once every second. This way the push-button and whole project's functionality was actually working properly and reliably.

Another problem we encountered while developing this project was the fact that the RPI-Zero takes an extended amount of time to connect to a WiFi network once it is instructed to do so. This problem is further exaggerated when a multitude of devices is connected to the same wireless router, which is what was happening during testing in the lab. This problem was not critical per se, but it proved to be annoying while we were testing and debugging the project.

Testing

During the development phase of our lock project, we of course tested our source code on the STM boards. This proved to be a somewhat tedious process, as we could only use the STM boards in the lab, which was not always a practical solution.

At first, we made sure that the codebase we used from the sixth lab worked correctly and used the circular buffer structure properly, as losing data from the buffers or unreliable communication would be very hard to debug.

We then continued to program the key part of the project and verified that the push-button and MQTT message publishing worked correctly. We mostly tested this behaviour using a USB-UART device connected to the GPIO pins on the STM board.

After the key part of the project was tested to be reliable and fully functional, we moved on to its counterpart. This proved to be a relatively simple task, as we could simply recycle all of the communication and JSON parsing code while only changing some static strings. We also did not have to worry about the lock part's push-button functionality, as it was not designed to have one.

After both parts of the project were completed, we continued working on the automatic setup of the serial bridge's WiFi connection and MQTT topic subscriptions. This proved to be a tedious task as we had to restart the Zero W with every new build of our source code, which took a relatively long time (3-5 minutes) each time.

Once the automatic setup function was working correctly, we tested the functionality of the whole system many times just to verify that everything worked reliably and there no hidden bugs or inconsistencies in the source code or communication link.

We did not have to test the Pi 3 or the MQTT server at all, since those were verified to be working properly in one of the previous lab assignments. The MQTT server itself was pretty much plug-and-play.

Conclusion

This project definitely did hit a home run when we talk about the learning experience. This project makes use of every lab that we have worked on in this class. From setting up a Raspberry Pi with an MQTT server, to making use of communications form and to the GPIO pins on the STM discovery board. The most challenging part of this project was serializing the data with JSON and passing it to the circular buffer for communications. Our recommendations for this class are to invest in a router which would be capable of servicing a great number of clients at once with no hiccups, as each student is going to use 3 devices that need to connect to the router. All in all, we thought working on this project was a good experience that helped us put the knowledge we gained during the course to practice. If we wanted to expand on this project or start a whole new one, our existing codebase could serve as a good starting point. One fairly simple expansion to this project would be an integration with the Freeboard service that could tie into the MQTT messaging protocol.

Appendix

The entire codebase of our ECE 631 project, including this writeup in .docx and .pdf formats, can be found on K-State's Beocat Gitlab at https://gitlab.beocat.ksu.edu/s18.LockTeam/lock-project