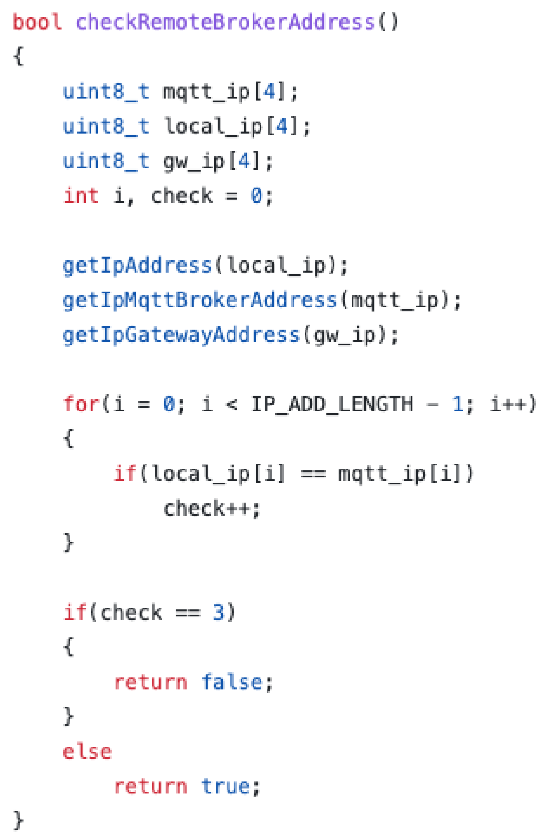
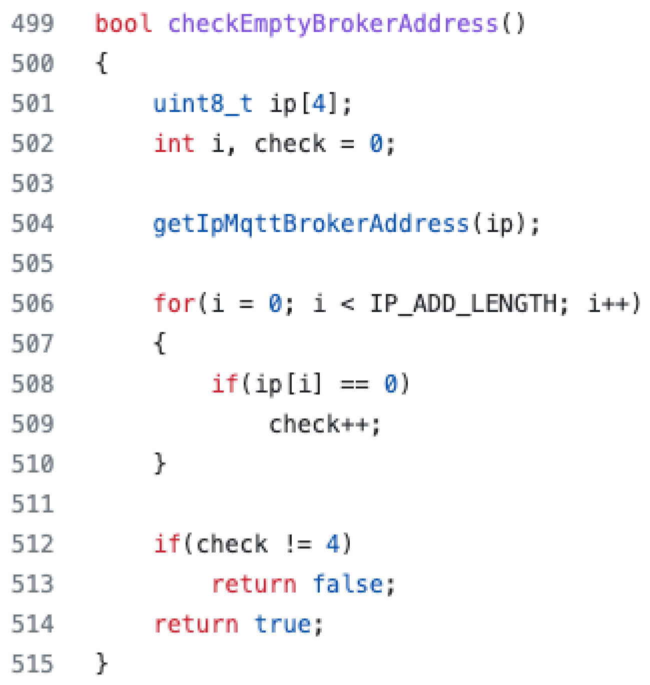
Connection / Device Management: Raquel Reyes & Sho Reagan

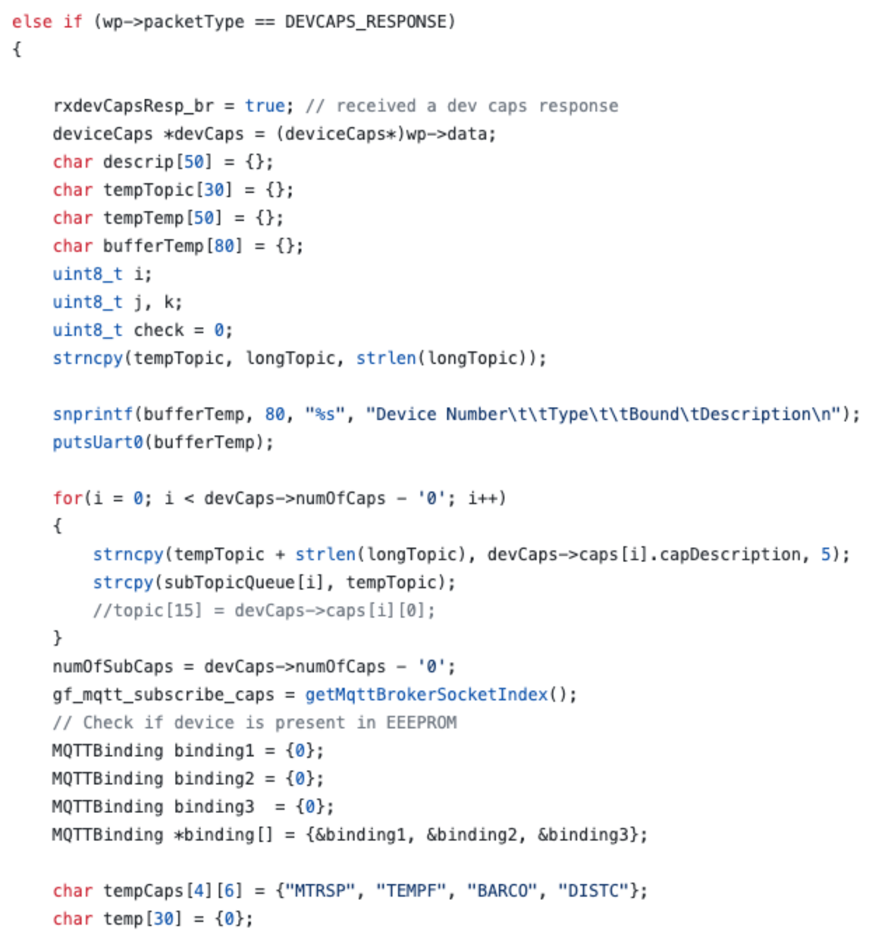


```
bool checkRemoteBrokerAddress()
{
    uint8_t mqtt_ip[4];
    uint8_t local_ip[4];
    uint8_t gw_ip[4];
    int i, check = 0;

    getIpAddress(local_ip);
    getIpMqttBrokerAddress(mqtt_ip);
    getIpGatewayAddress(gw_ip);

    for(i = 0; i < IP_ADD_LENGTH - 1; i++)
    {
        if(local_ip[i] == mqtt_ip[i])
            check++;
    }

    if(check == 3)
    {
        return false;
    }
    else
        return true;
}
```



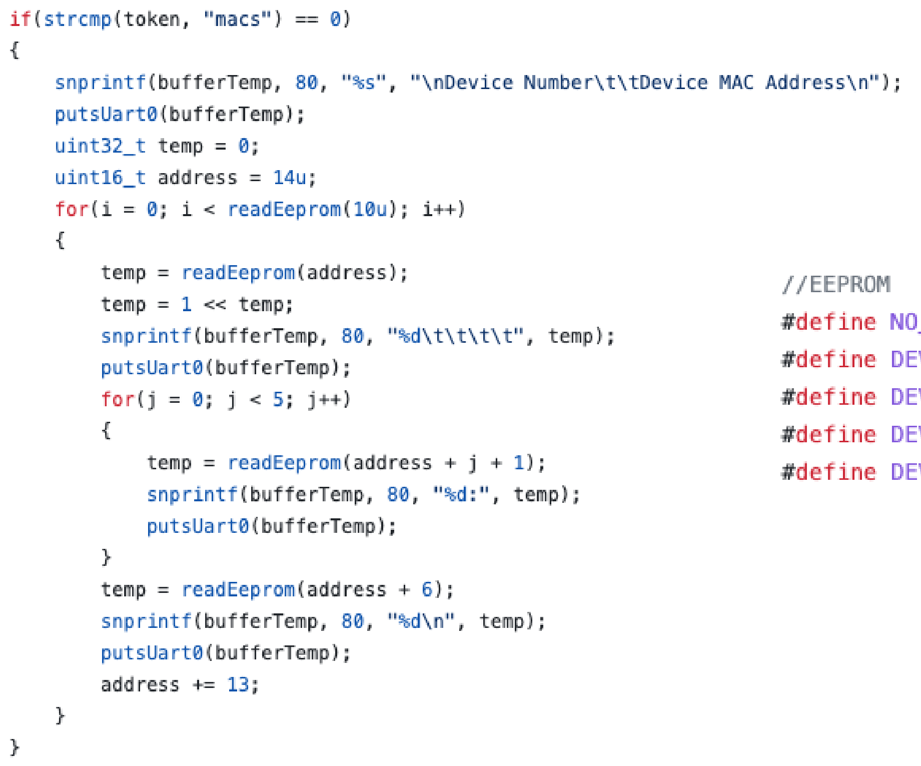
```
bool checkEmptyBrokerAddress()
{
    uint8_t ip[4];
    int i, check = 0;

    getIpMqttBrokerAddress(ip);

    for(i = 0; i < IP_ADD_LENGTH; i++)
    {
        if(ip[i] == 0)
            check++;
    }

    if(check != 4)
        return false;
    return true;
}
```

Utilizing completed Project 1 code as a base, we added the capabilities for not only connecting to a local MQTT broker, but also a remote one, such as Adafruit. This was done by extending the MQTT connect functions to be able to include extra flags and a desired username and password, which were set as defines in the code. We also had to add checks, in the form of the functions *checkEmptyBrokerAddress* and *checkRemoteBrokerAddress*, which returned boolean values depending on whether the broker address stored in eeprom was remote or local using the subnet mask. If there wasn't any valid address imputed, the device would not attempt to auto connect on startup. If there was, then depending on the type of broker address stored, it would either send an ARP request to the gateway IP or broker IP. We also had to make sure that during the processing of the ARP response, to change the current socket's remote IP address back to the broker's IP in the case of a remote broker, since we wanted to still send messages to the broker IP, but keep the gateway MAC.

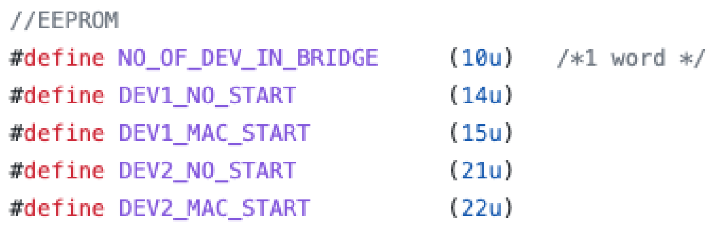


```
else if (wp->packetType == DEVCAPS_RESPONSE)
{

    rxdevCapsResp_br = true; // received a dev caps response
    deviceCaps *devCaps = (deviceCaps*)wp->data;
    char descrip[50] = {};
    char tempTopic[30] = {};
    char tempTemp[50] = {};
    char bufferTemp[80] = {};
    uint8_t i;
    uint8_t j, k;
    uint8_t check = 0;
    strncpy(tempTopic, longTopic, strlen(longTopic));

    snprintf(bufferTemp, 80, "%s", "Device Number\t\tType\t\tBound\tDescription\n");
    putsUart0(bufferTemp);

    for(i = 0; i < devCaps->numOfCaps - '0'; i++)
    {
        strncpy(tempTopic + strlen(longTopic), devCaps->caps[i].capDescription, 5);
        strcpy(subTopicQueue[i], tempTopic);
        //topic[15] = devCaps->caps[i][0];
    }
    numOfSubCaps = devCaps->numOfCaps - '0';
    gf_mqtt_subscribe_caps = getMqttBrokerSocketIndex();
    // Check if device is present in EEEPROM
    MQTTBinding binding1 = {0};
    MQTTBinding binding2 = {0};
    MQTTBinding binding3  = {0};
    MQTTBinding *binding[] = {&binding1, &binding2, &binding3};

    char tempCaps[4][6] = {"MTRSP", "TEMPF", "BARCO", "DISTC"};
    char temp[30] = {0};
```

For device management, we created the UI commands “macs”, “showCaps”, as well as assisting the Tables team with the “bind” and “unbind” commands. We also processed the packet received during a DEVCAPS RESPONSE message and stored it to be able to be passed into the Table team’s functions. While processing, we stored its information into a binding struct and also printed out the device number, function type, and whatever long description about the function the device team decided to send. We chose to format this into an array of structs, where each individual corresponded to a specific capability, since it was decided amongst the teams that aside from the web server, each device would be limited to three capabilities. For the purpose of later auto-subbing to the topics stored in the response packet, we concatenated a short 5 character key to a defined string “uta_iot/feeds/” since this is the syntax for a topic in Adafruit, although we calculated the values to be variable if either the syntax or username was changed. They were stored into a string array we used as a queue to later process the subscribe commands.

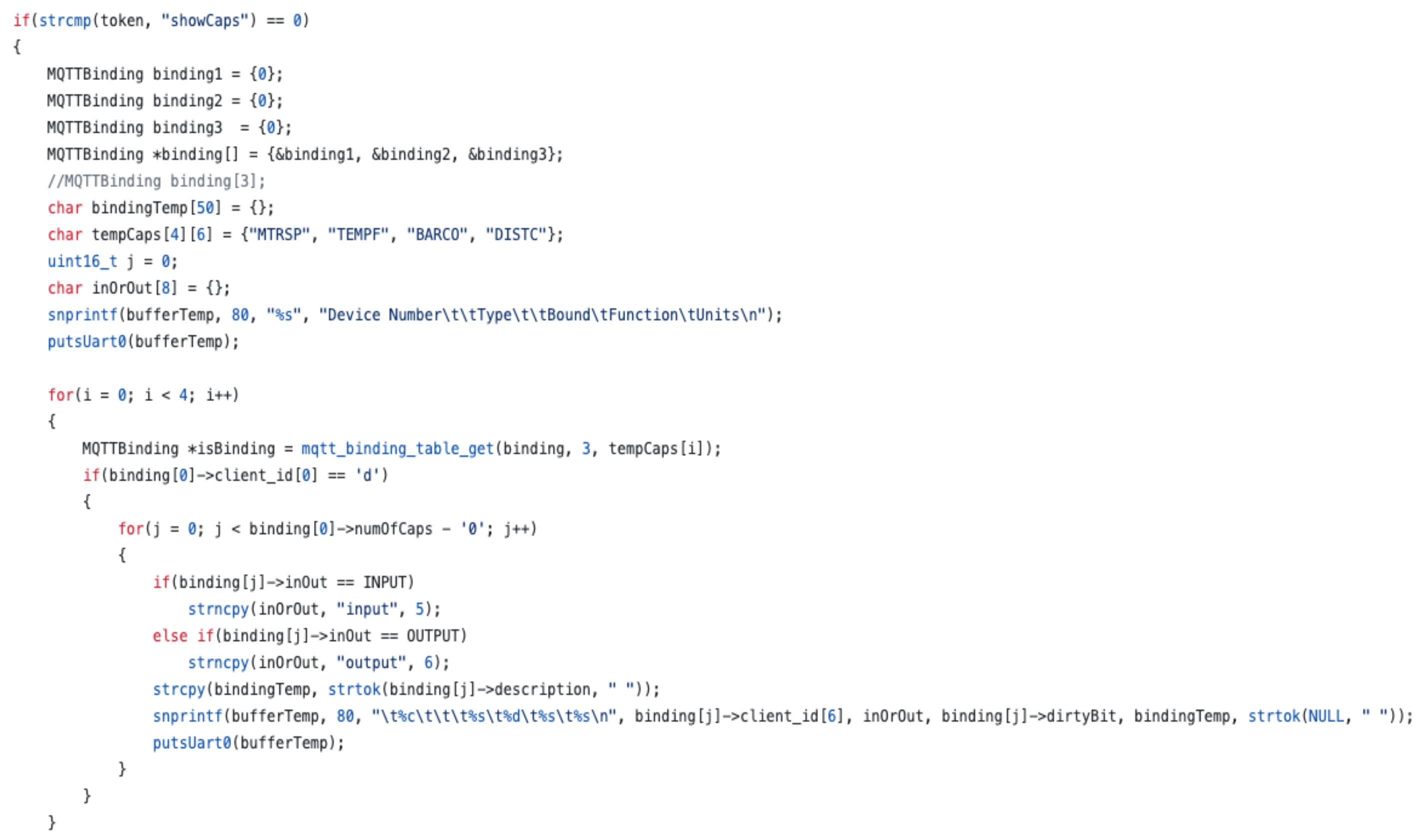


```c
if(strcmp(token, "macs") == 0)
{
    snprintf(bufferTemp, 80, "%s", "\nDevice Number\t\tDevice MAC Address\n");
    putsUart0(bufferTemp);
    uint32_t temp = 0;
    uint16_t address = 14u;
    for(i = 0; i < readEeprom(10u); i++)
    {
        temp = readEeprom(address);
        temp = 1 << temp;
        snprintf(bufferTemp, 80, "%d\t\t\t\t", temp);
        putsUart0(bufferTemp);
        for(j = 0; j < 5; j++)
        {
            temp = readEeprom(address + j + 1);
            snprintf(bufferTemp, 80, "%d:", temp);
            putsUart0(bufferTemp);
        }
        temp = readEeprom(address + 6);
        snprintf(bufferTemp, 80, "%d\n", temp);
        putsUart0(bufferTemp);
        address += 13;
    }
}
```



```c
//EEPROM
#define NO_OF_DEV_IN_BRIDGE     (10u)   /*1 word */
#define DEV1_NO_START           (14u)
#define DEV1_MAC_START          (15u)
#define DEV2_NO_START           (21u)
#define DEV2_MAC_START          (22u)
```

The "macs" command displayed all currently assigned device IDs and their corresponding MAC addresses, and this was done by reading the local EEPROM at the address specified by the wireless team. When a new device connects to the bridge team, they must send their MAC address in order to get assigned a device number, which is sent back in the JOIN RESPONSE message. Based on the addresses defined in local eeprom by the wireless team, we were able to access the number of currently assigned devices and iterate through memory to read first the device ID and then its MAC address. However, the defined values given to us were actually incorrect, as the API team had inadvertently placed a buffer of 6 zeroes in between a previous device's MAC and the next device's ID, which we did not realize until printing out everything stored in that memory region. Once those were accounted for, the display was able to be completed.



```c
if(strcmp(token, "showCaps") == 0)
{
    MQTTBinding binding1 = {0};
    MQTTBinding binding2 = {0};
    MQTTBinding binding3  = {0};
    MQTTBinding *binding[] = {&binding1, &binding2, &binding3};
    //MQTTBinding binding[3];
    char bindingTemp[50] = {};
    char tempCaps[4][6] = {"MTRSP", "TEMPF", "BARCO", "DISTC"};
    uint16_t j = 0;
    char inOrOut[8] = {};
    snprintf(bufferTemp, 80, "%s", "Device Number\t\tType\t\tBound\tFunction\tUnits\n");
    putsUart0(bufferTemp);

    for(i = 0; i < 4; i++)
    {
        MQTTBinding *isBinding = mqtt_binding_table_get(binding, 3, tempCaps[i]);
        if(binding[0]->client_id[0] == 'd')
        {
            for(j = 0; j < binding[0]->numOfCaps - '0'; j++)
            {
                if(binding[j]->inOut == INPUT)
                    strncpy(inOrOut, "input", 5);
                else if(binding[j]->inOut == OUTPUT)
                    strncpy(inOrOut, "output", 6);
                strcpy(bindingTemp, strtok(binding[j]->description, " "));
                snprintf(bufferTemp, 80, "\t%c\t\t\t%s\t%d\t%s\t%s\n", binding[j]->client_id[6], inOrOut, binding[j]->dirtyBit, bindingTemp, strtok(NULL, " "));
                putsUart0(bufferTemp);
            }
        }
    }
```

The “showCaps” command was meant to display all capabilities of each device, whether the function was an input or output, all the different levels you could set it to, and whether the function was bound to a topic or not. It retrieved the information previously stored in the tables, but an issue we did not get the chance to improve was that while information storage into the tables was done correctly, retrieval caused all the capabilities in the resulting struct to be duplicates of the first capability entered for that device. This was mainly due to a limited timeframe, since the API was not completed to be able to deliver fully processable devcaps responses until the afternoon of the testing day, so we were unable to check whether our code was properly parsing or storing the needed information.

While an attempt for an auto-subscribe on a successful connect was made, it ultimately kept sending out incorrectly parsed topics, so they would end up either inadvertently closing the connection from incorrect syntax, or be subscriptions to a non-desired topic.

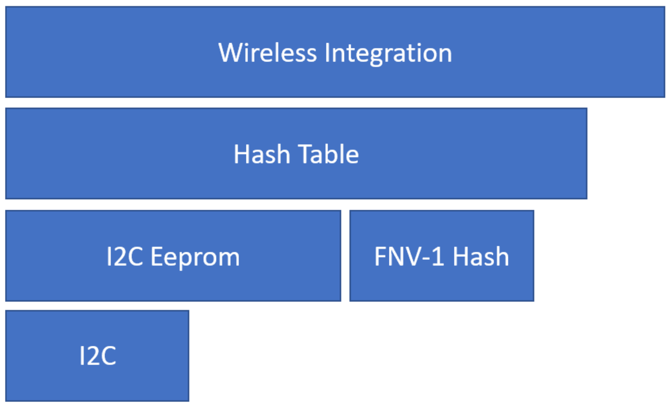
For the “bind” and “unbind” commands, we chose to format it so that a function could only be bound to one specific corresponding topic, and the commands were to set whether the binding was active or not. The parameter was a predetermined 5 character short description keyword that was decided on between the Device and Bridge teams, and we used the first 3 or 4 characters to determine the device name and the last 1 or 2 for the device function in order to retrieve the correct binding. Then we would essentially overwrite the binding by changing what we called the “dirty bit” to denote whether a topic was bound (subscribed) or not, and then writing back into the binding tables. Although this functionality was unable to be tested at the current time, we are confident that with a little more time this would’ve been able to be fleshed out into something more fully functional.

Overall, most of our issues came during the integration portion of the project, as discourse amongst the API team as well as several versions of non-working code being sent out to the rest of the teams caused the parts of the bridge team that were a link between the teams to fall behind. There was also inconsistency with the transmission modules themselves, which meant that often times it was random as to when we would receive a device caps response to be able to test on. Due to the time constraints created by the above issues, which had to be fixed or at least temporarily bypassed for our testing purposes, a fair portion of the bridge team’s code was unable to be tested unless it was disconnected from the other teams, such as the connect portion. We are hopeful that if there was a little more cohesion as well as message consistency, the last of the partially faulty requirements would have been fulfilled.

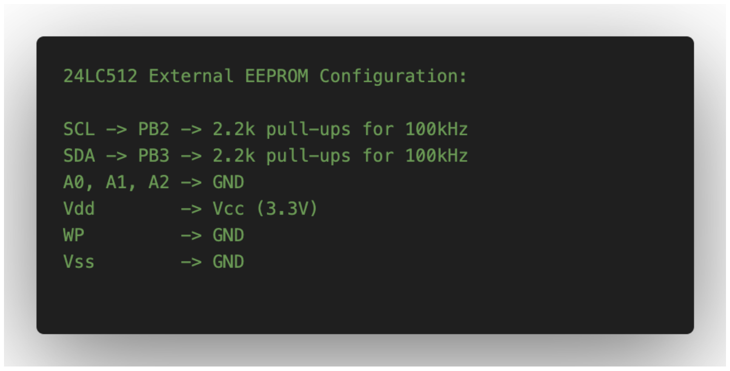
Binding Table: Julian Schneider & Dario Ugalde

The purpose of the MQTT binding table is to manage the relationships between devices and their corresponding topics on a broker (bridge). We developed a set of functions that form a binding table to handle this management. Due to our constrained memory environment we opted for storing the binding table on external EEPROM, allowing for efficient and persistent storage of bindings between clients, topics, device capabilities, etc..

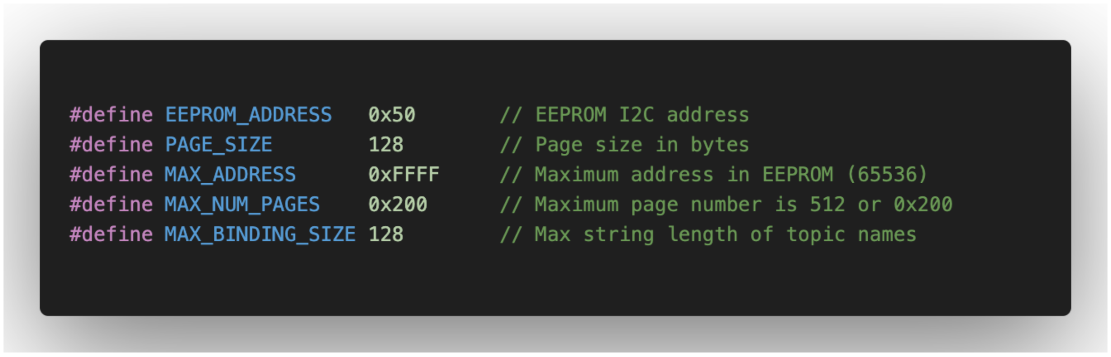
The following image depicts the tables’ team development approach. For the hash table architecture, we opted to take a layered approach as the capabilities from each layer are extended to implement the subsequent layers.



One of our first tasks was to create a library for interfacing with an external eeprom device and solder it onto our board. The device we received is the 24LC512 External I2C EEPROM. Using the address scheme chosen below we had 64K bytes of memory at our disposal.



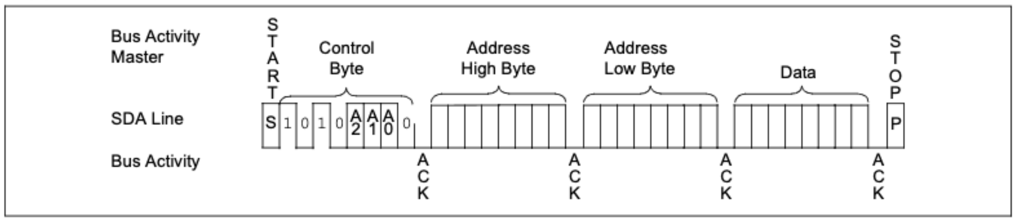
```
24LC512 External EEPROM Configuration:

SCL -> PB2 -> 2.2k pull-ups for 100kHz
SDA -> PB3 -> 2.2k pull-ups for 100kHz
A0, A1, A2 -> GND
Vdd        -> Vcc (3.3V)
WP         -> GND
Vss        -> GND
```



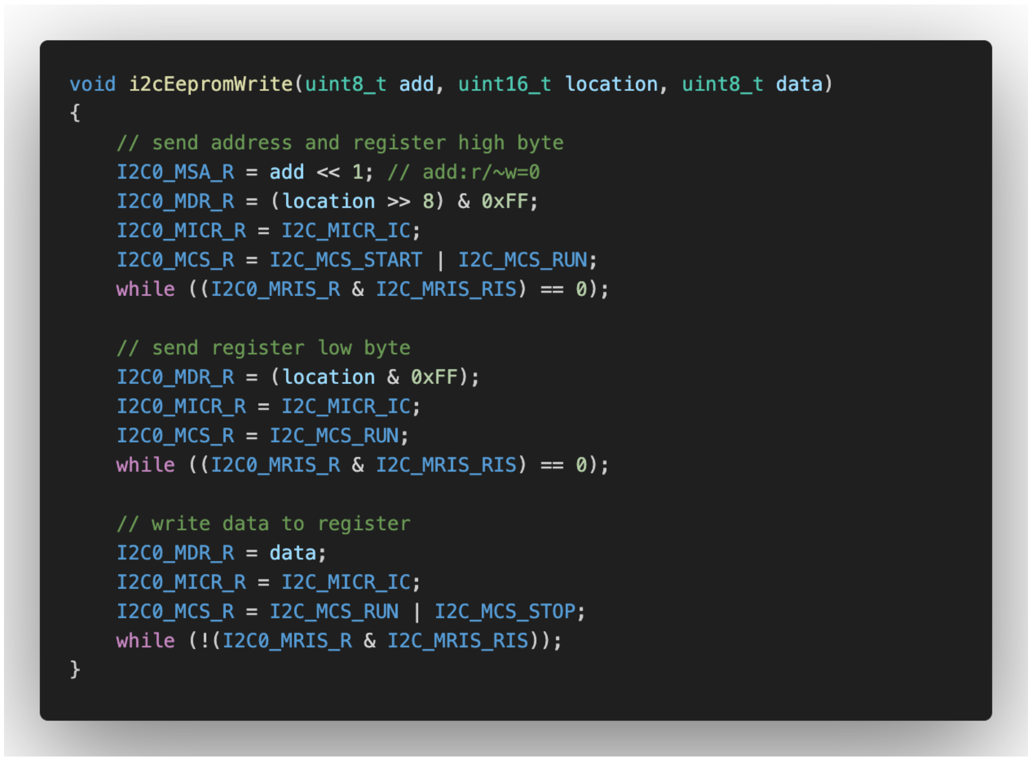
```
#define EEPROM_ADDRESS   0x50        // EEPROM I2C address
#define PAGE_SIZE        128         // Page size in bytes
#define MAX_ADDRESS      0xFFFF      // Maximum address in EEPROM (65536)
#define MAX_NUM_PAGES    0x200       // Maximum page number is 512 or 0x200
#define MAX_BINDING_SIZE 128         // Max string length of topic names
```

Several considerations in regard to the I2C communication protocol were made in order to properly communicate with this device. We can only write a byte at time and we are limited by its 128 byte page write buffer.

FIGURE 6-1: BYTE WRITE

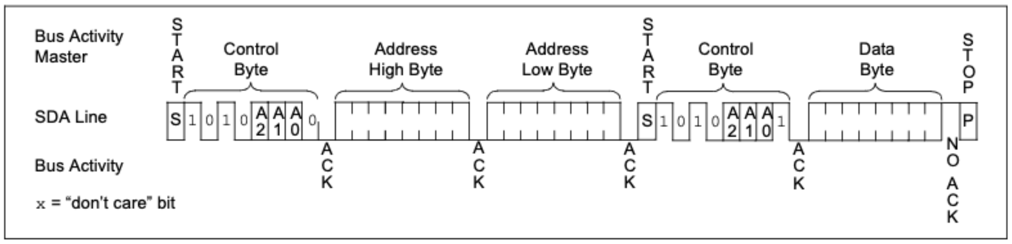


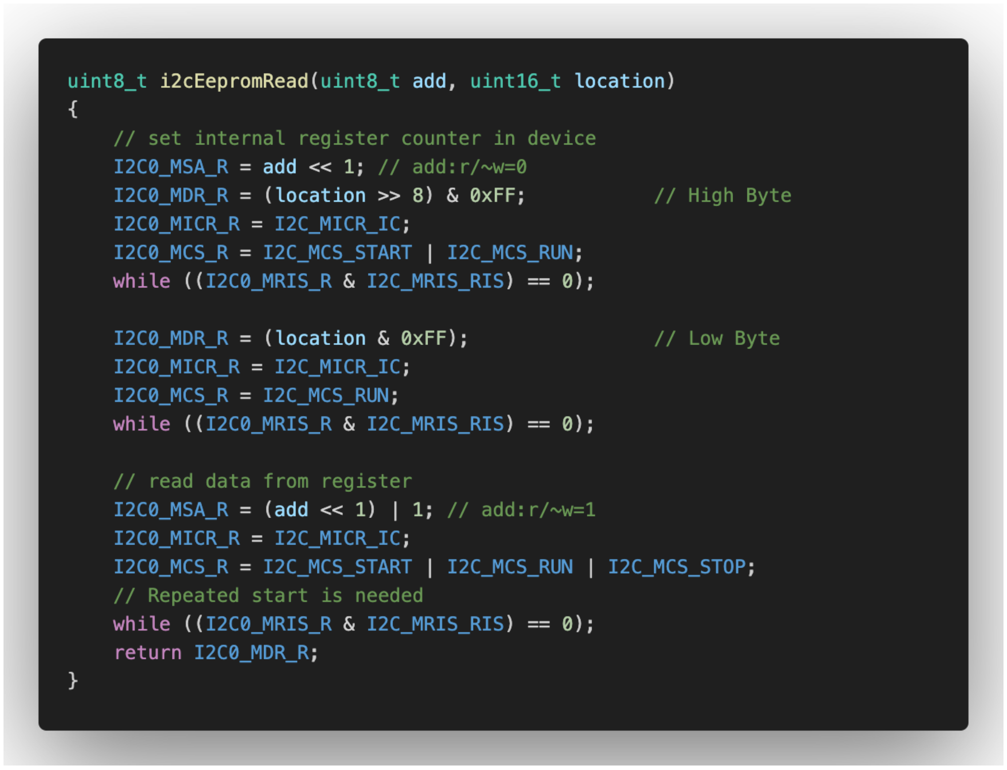
The following code was modified to conform to the 24LC512 byte write specification.



```c
void i2cEepromWrite(uint8_t add, uint16_t location, uint8_t data)
{
    // send address and register high byte
    I2C0_MSA_R = add << 1; // add:r/~w=0
    I2C0_MDR_R = (location >> 8) & 0xFF;
    I2C0_MICR_R = I2C_MICR_IC;
    I2C0_MCS_R = I2C_MCS_START | I2C_MCS_RUN;
    while ((I2C0_MRIS_R & I2C_MRIS_RIS) == 0);

    // send register low byte
    I2C0_MDR_R = (location & 0xFF);
    I2C0_MICR_R = I2C_MICR_IC;
    I2C0_MCS_R = I2C_MCS_RUN;
    while ((I2C0_MRIS_R & I2C_MRIS_RIS) == 0);

    // write data to register
    I2C0_MDR_R = data;
    I2C0_MICR_R = I2C_MICR_IC;
    I2C0_MCS_R = I2C_MCS_RUN | I2C_MCS_STOP;
    while (!(I2C0_MRIS_R & I2C_MRIS_RIS));
}
```

FIGURE 8-2: RANDOM READ



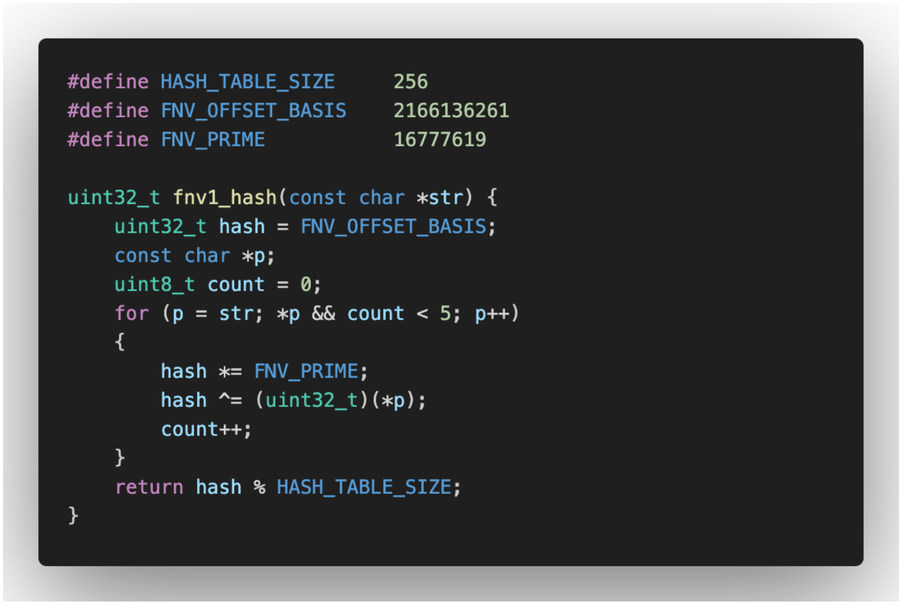


```
uint8_t i2cEepromRead(uint8_t add, uint16_t location)
{
    // set internal register counter in device
    I2C0_MSA_R = add << 1; // add:r/~w=0
    I2C0_MDR_R = (location >> 8) & 0xFF;           // High Byte
    I2C0_MICR_R = I2C_MICR_IC;
    I2C0_MCS_R = I2C_MCS_START | I2C_MCS_RUN;
    while ((I2C0_MRIS_R & I2C_MRIS_RIS) == 0);

    I2C0_MDR_R = (location & 0xFF);                // Low Byte
    I2C0_MICR_R = I2C_MICR_IC;
    I2C0_MCS_R = I2C_MCS_RUN;
    while ((I2C0_MRIS_R & I2C_MRIS_RIS) == 0);

    // read data from register
    I2C0_MSA_R = (add << 1) | 1; // add:r/~w=1
    I2C0_MICR_R = I2C_MICR_IC;
    I2C0_MCS_R = I2C_MCS_START | I2C_MCS_RUN | I2C_MCS_STOP;
    // Repeated start is needed
    while ((I2C0_MRIS_R & I2C_MRIS_RIS) == 0);
    return I2C0_MDR_R;
}
```

The random read ability proved to be the source of most of our initial problems when interacting with the device. Our standard I2C library provided by Dr. Losh needed to be modified in order for us to meet the devices communication requirements. In particular the device required a repeated *start bit* followed by the *control byte*. Once we made those changes to the I2C library we were then able to properly read from EEPROM.

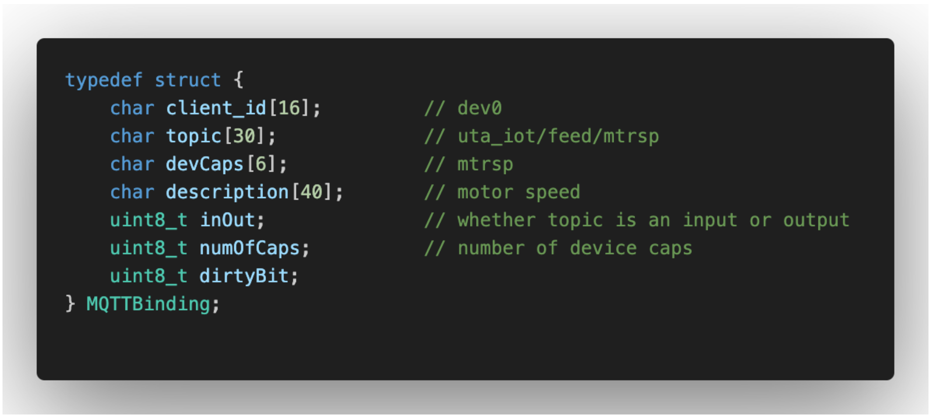
Next is creating the binding table library and deciding on a data structure for managing the bindings. We decided on a hash table as the data structure. Now that we have a desired data structure we now need to decide the algorithm for hashing our data.

FNV-1A hashing algorithm was chosen due to its simplicity and efficiency. It consists of a series of simple multiplication and XOR operations. Another advantage is its reasonably uniform distribution of hashed data across the table which helps reduce collisions. This allows for average search in constant time. Another advantage is that FNV-1A is very deterministic in that it always returns the same hash value for the same input data. This is essential to us since we need the hash table to alway map the same key to the same index. During the integration phase, we uncovered a bug where our implementation of this hash was overrunning the devCaps string resulting in a mismatch of indexes when storing bindings and searching for specific devCaps. We resolved this by limiting the iteration of calculating loops to the length of the devCaps string as shown in the conditional step of the for loop.

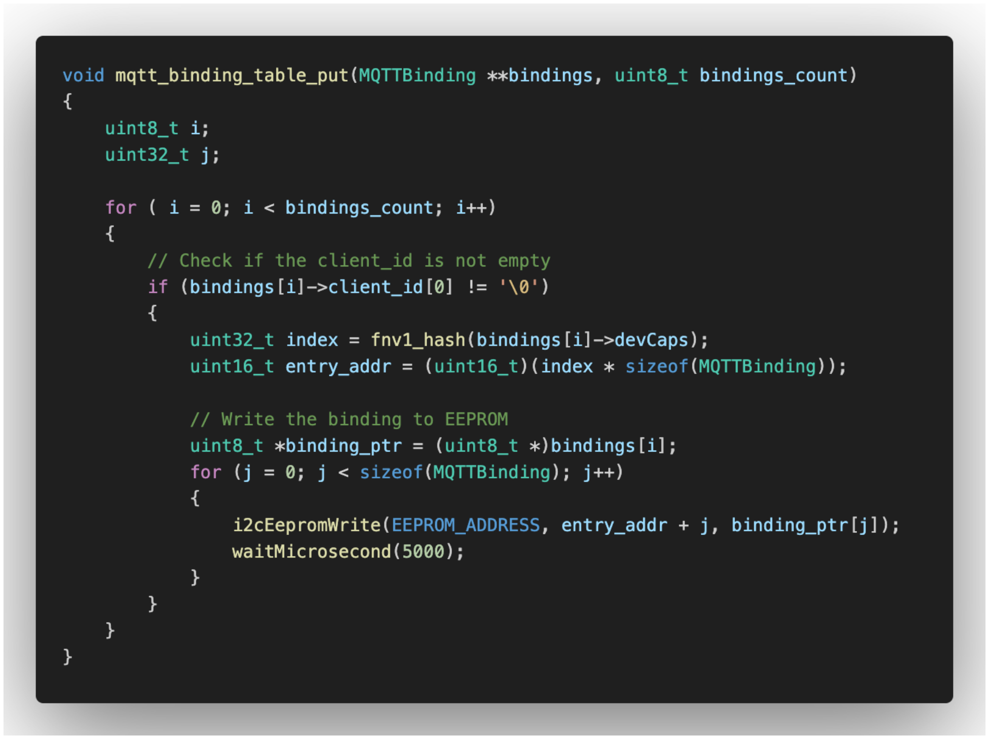


```c
#define HASH_TABLE_SIZE     256
#define FNV_OFFSET_BASIS    2166136261
#define FNV_PRIME           16777619

uint32_t fnv1_hash(const char *str) {
    uint32_t hash = FNV_OFFSET_BASIS;
    const char *p;
    uint8_t count = 0;
    for (p = str; *p && count < 5; p++)
    {
        hash *= FNV_PRIME;
        hash ^= (uint32_t)(*p);
        count++;
    }
    return hash % HASH_TABLE_SIZE;
}
```

The functions we provided the rest of the team with consisted of the following *mqtt_binding_table_put, mqtt_binding_table_get,* and *mqtt_binding_table_remove.* We decided to create a structure which would provide the team with all the useful information one could need from a binding entry.



```c
typedef struct {
    char client_id[16];         // dev0
    char topic[30];             // uta_iot/feed/mtrsp
    char devCaps[6];            // mtrsp
    char description[40];       // motor speed
    uint8_t inOut;              // whether topic is an input or output
    uint8_t numOfCaps;          // number of device caps
    uint8_t dirtyBit;
} MQTTBinding;
```



```c
void mqtt_binding_table_put(MQTTBinding **bindings, uint8_t bindings_count)
{
    uint8_t i;
    uint32_t j;

    for ( i = 0; i < bindings_count; i++)
    {
        // Check if the client_id is not empty
        if (bindings[i]->client_id[0] != '\0')
        {
            uint32_t index = fnv1_hash(bindings[i]->devCaps);
            uint16_t entry_addr = (uint16_t)(index * sizeof(MQTTBinding));

            // Write the binding to EEPROM
            uint8_t *binding_ptr = (uint8_t *)bindings[i];
            for (j = 0; j < sizeof(MQTTBinding); j++)
            {
                i2cEepromWrite(EEPROM_ADDRESS, entry_addr + j, binding_ptr[j]);
                waitMicrosecond(5000);
            }
        }
    }
}
```

Mqtt_binding_table_put **:**

- This function stores a set of MQTT bindings (clientId, devCaps, etc.) in the external EEPROM. It accepts an array of pointers to MQTTBinding structs that have already been filled with data. It uses the FNV-1A hash to create a unique index for each binding based on the devCaps and store that binding in EEPROM.

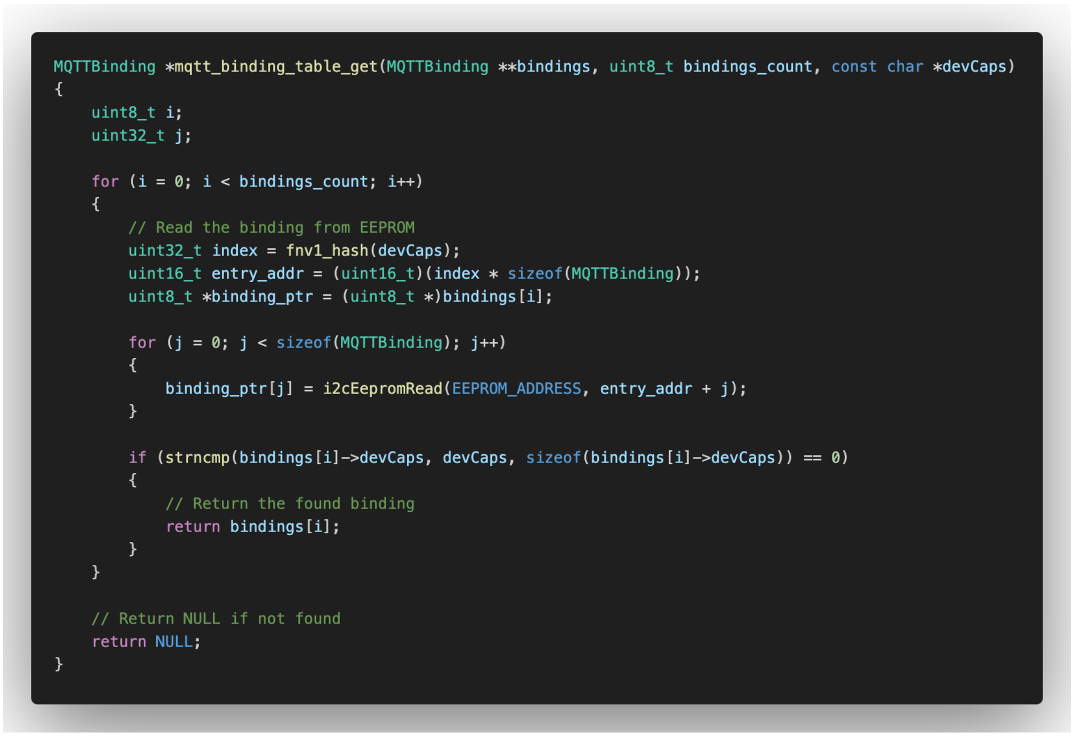
- The function iterates over each binding in the array. It then checks if the *client_id* field of the current binding is empty. If it is empty, the function skips the current iteration and proceeds with the next binding.

- The FNV-1A hash is applied to the *devCaps* of the current binding to generate a unique index. This *index* is later used to determine the address in the external eeprom where the binding will be stored.

- The *entry_addr* is calculated by multiplying the unique hash index by the size of the MQTTBinding struct. This ensures that the memory addresses for different bindings do not overlap, providing a unique address in EEPROM for each binding.

- A pointer to an uint8_t *binding_ptr* is created by casting the current MQTTBinding struct pointer. This point is used to access the individual bytes of the binding when writing to eeprom.

- A loop iterated through each byte of the MQTTBinding struct and the i2cEepromWrite function write each byte to eeprom at the calculated address (entry_addr+current byte offset)



```c
MQTTBinding *mqtt_binding_table_get(MQTTBinding **bindings, uint8_t bindings_count, const char *devCaps)
{
    uint8_t i;
    uint32_t j;

    for (i = 0; i < bindings_count; i++)
    {
        // Read the binding from EEPROM
        uint32_t index = fnv1_hash(devCaps);
        uint16_t entry_addr = (uint16_t)(index * sizeof(MQTTBinding));
        uint8_t *binding_ptr = (uint8_t *)bindings[i];

        for (j = 0; j < sizeof(MQTTBinding); j++)
        {
            binding_ptr[j] = i2cEepromRead(EEPROM_ADDRESS, entry_addr + j);
        }

        if (strncmp(bindings[i]->devCaps, devCaps, sizeof(bindings[i]->devCaps)) == 0)
        {
            // Return the found binding
            return bindings[i];
        }
    }

    // Return NULL if not found
    return NULL;
}
```

Mqtt_binding_table_get:

- This function accepts an array of pointers to MQTTBinding structs, the number of bindings in the array, and a string representing the device capabilities. The function returns a pointer to the first MQTTBinding struct that matches the given devCaps.

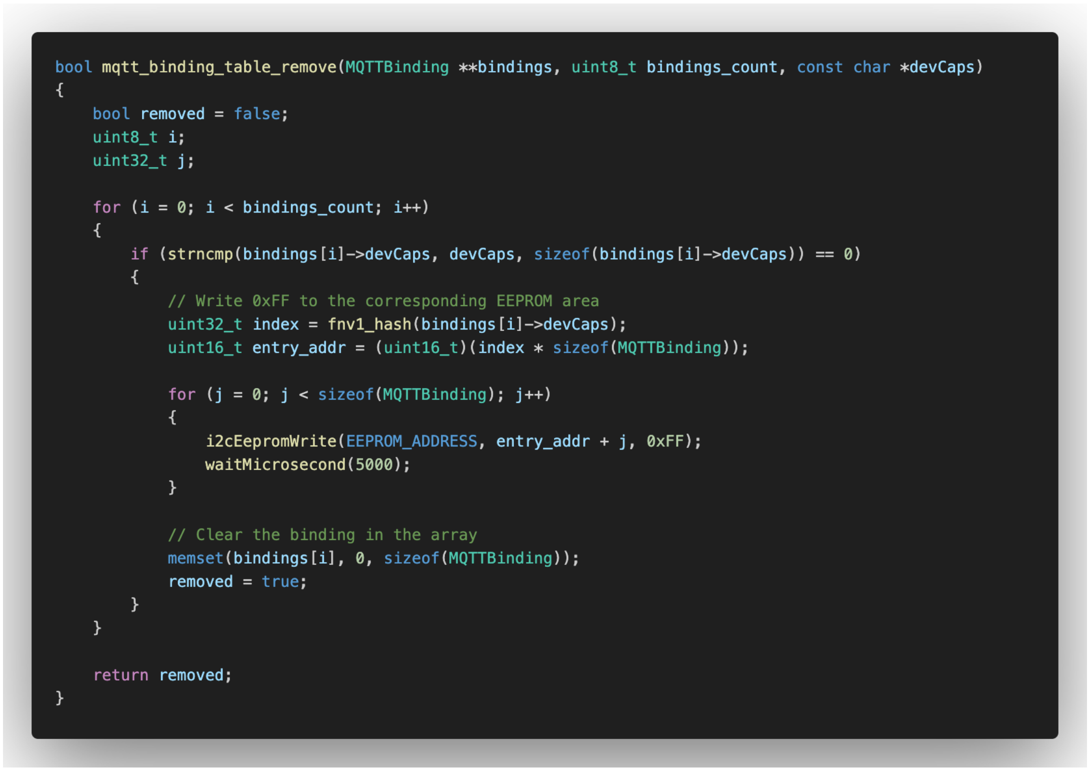
- It iterates through the bindings array, it checks if the devcaps field of the current binding matches the given devCaps parameter.

- The FNV-1A hash is applied to the devCaps parameter to generate a unique index. This index is used to determine where in the external eeprom the binding is stored. The entry_ptr is calculated by multiplying the unique index by the size of the MQTTBinding struct.

- A pointer to an uint8_t *binding_ptr* is created by casting the current MQTTBinding struct pointer. This point is used to access the individual bytes of the binding when reading from eeprom.

- A loop iterates through each byte of the MQTTBinding struct and the i2cEepromRead function reads each byte from eeprom at the calculated address (entry_addr+current byte offset)

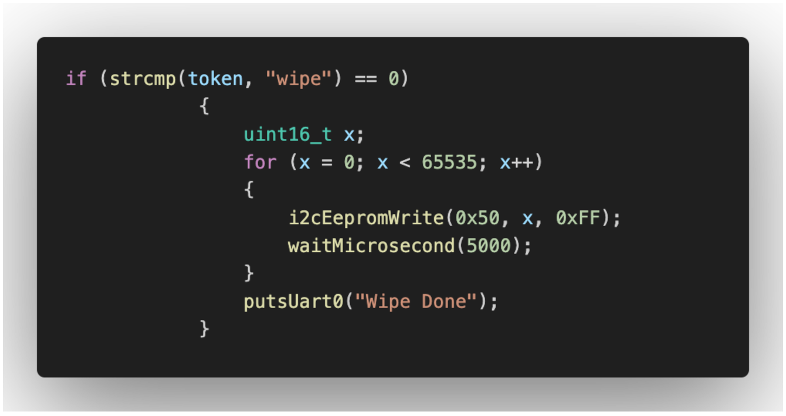
- If no matching binding is found after iterating through all the bindings in the array, the function returns NULL



```c
bool mqtt_binding_table_remove(MQTTBinding **bindings, uint8_t bindings_count, const char *devCaps)
{
    bool removed = false;
    uint8_t i;
    uint32_t j;

    for (i = 0; i < bindings_count; i++)
    {
        if (strncmp(bindings[i]->devCaps, devCaps, sizeof(bindings[i]->devCaps)) == 0)
        {
            // Write 0xFF to the corresponding EEPROM area
            uint32_t index = fnv1_hash(bindings[i]->devCaps);
            uint16_t entry_addr = (uint16_t)(index * sizeof(MQTTBinding));

            for (j = 0; j < sizeof(MQTTBinding); j++)
            {
                i2cEepromWrite(EEPROM_ADDRESS, entry_addr + j, 0xFF);
                waitMicrosecond(5000);
            }

            // Clear the binding in the array
            memset(bindings[i], 0, sizeof(MQTTBinding));
            removed = true;
        }
    }

    return removed;
}
```

Mqtt_binding_table_remove:

- This function operates similarly to the put function, however it writes over the binding with the value 0xFF. This effectively removes the binding from the table and returns the eeprom back to its initial state. This function proved to be buggy and we didn't have time to fix it. To alleviate this we created the function below to manually reset all the addresses in memory.



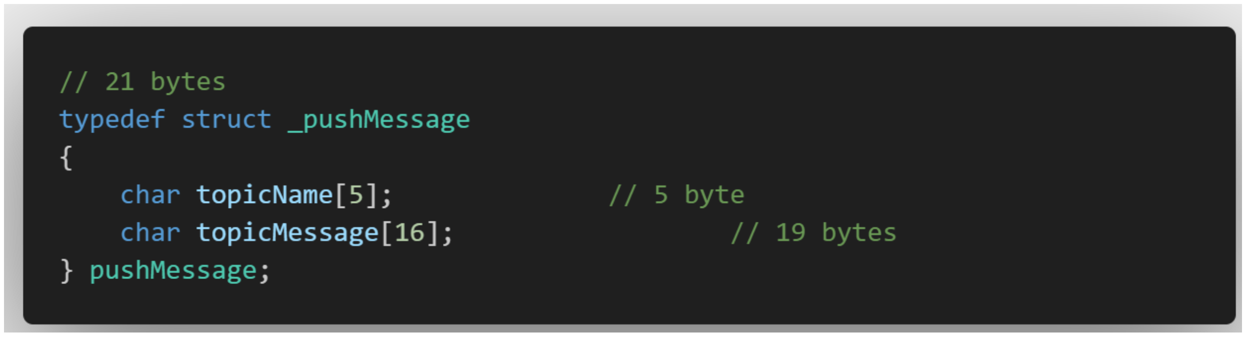
```c
if (strcmp(token, "wipe") == 0)
                {
                    uint16_t x;
                    for (x = 0; x < 65535; x++)
                    {
                        i2cEepromWrite(0x50, x, 0xFF);
                        waitMicrosecond(5000);
                    }
                    putsUart0("Wipe Done");
                }
```

In conclusion, despite encountering integration challenges during the testing phase of our code, we made significant progress in creating a functional MQTT binding table. A primary issue we faced was the difficulty in returning the entire group of struct when reading from the binding table. With additional time to refine and test our code, we are confident that this issue could be resolved. As it stands, we successfully implemented the ability to create bindings in the table, albeit only the first binding per device can be retrieved.

Collaboration with the Connection team and the Forwarding team played a crucial role in implementing the Bind and Unbind commands. However, due to time constraints, we were unable to thoroughly test these commands. Moving forward, we recommend allocating more time for testing and troubleshooting in order to ensure a fully functional and reliable binding table.

Forwarding: Marvin Coopman, Xuan Wang, & Ulysses Chaparro

From a high level perspective, the forwarding team was responsible for routing data from the various devices up to adafruit and messages from adafruit back down to input devices (including the web server). Using the API provided by the wireless API team, the binding tables from the tables team, and push messages from the various devices the forwarding team would forward the various uplink/downlink messages to their proper destination.



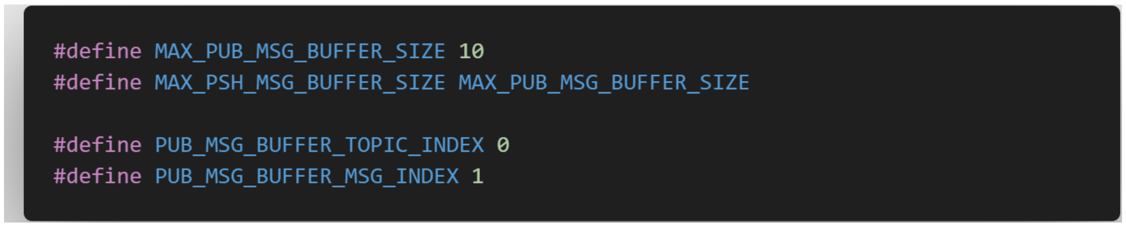
```c
// 21 bytes
typedef struct _pushMessage
{
    char topicName[5];              // 5 byte
    char topicMessage[16];                  // 19 bytes
} pushMessage;
```

Each PUSH message includes a shorthand topic name which acts as a key to the external EEPROM binding tables to find the associated adafruit long topic name. Once the long topic name is retrieved from the binding tables, the publish flag is set and a publish message is queued in a circular buffer to be transmitted during a downtime in the system.

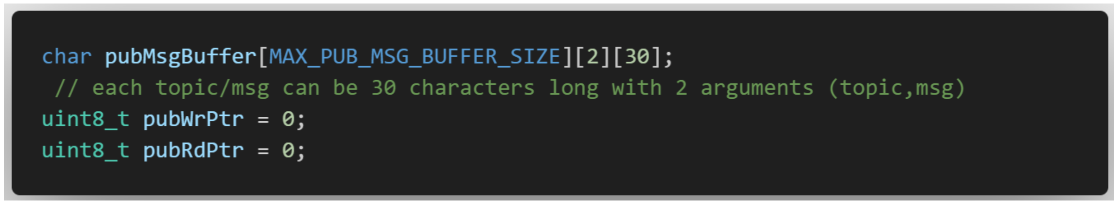
Inversely, for publish messages from adafruit, the long topic name is used as the key to the binding table to identify if the device's capability is configured as an input and should receive the message. If yes, the PUSH flag is set and a PUSH message is queued in a circular buffer and transmitted on the following downlink message from the bridge.

In the network, the web server acts as a unique device that needs to listen to every device and update the data displayed. On every publish message from adafruit, a downlink message will include the web server as a listener if the web server is connected.

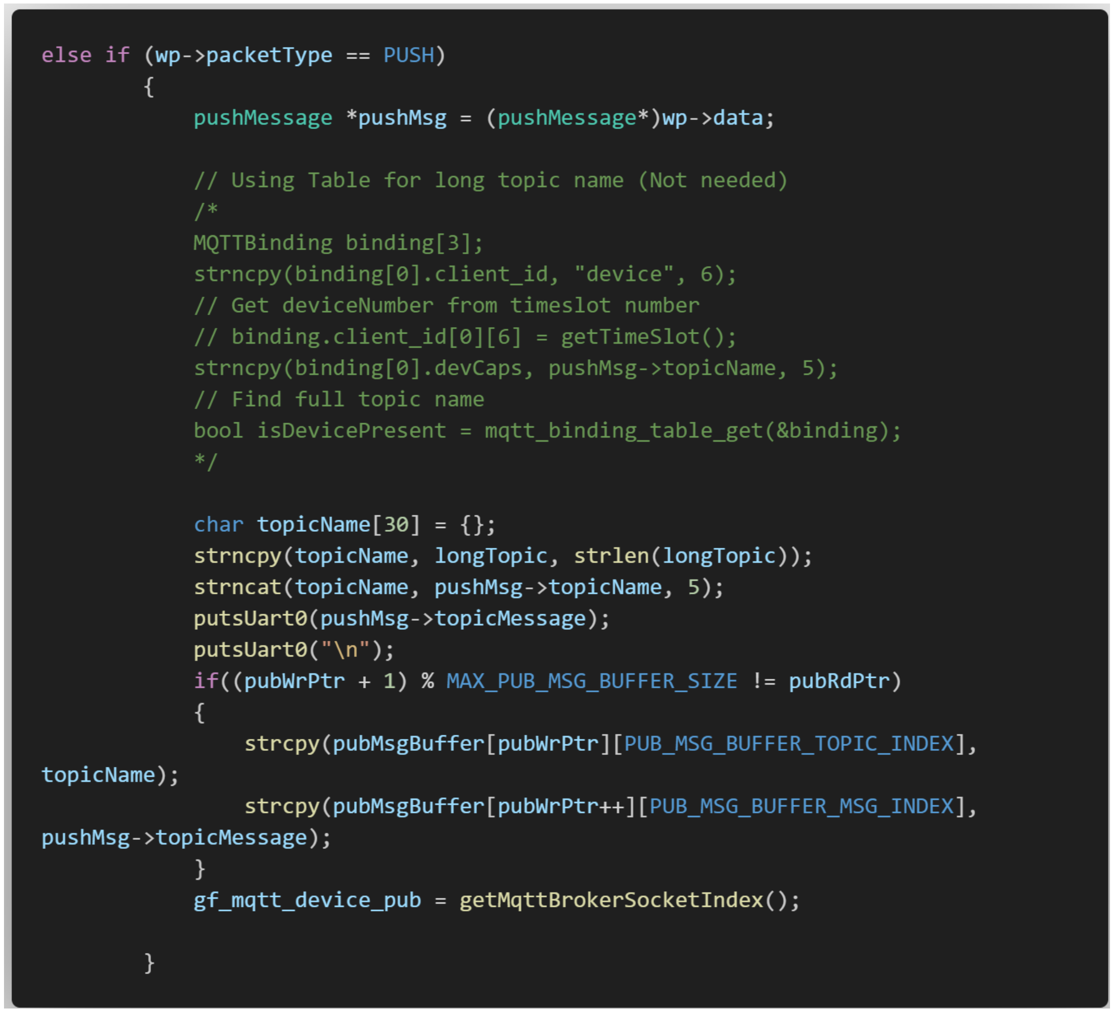
For the uplink message from the device, the following data structures and functions were used.



```
#define MAX_PUB_MSG_BUFFER_SIZE 10
#define MAX_PSH_MSG_BUFFER_SIZE MAX_PUB_MSG_BUFFER_SIZE

#define PUB_MSG_BUFFER_TOPIC_INDEX 0
#define PUB_MSG_BUFFER_MSG_INDEX 1
```



```
char pubMsgBuffer[MAX_PUB_MSG_BUFFER_SIZE][2][30];
 // each topic/msg can be 30 characters long with 2 arguments (topic,msg)
uint8_t pubWrPtr = 0;
uint8_t pubRdPtr = 0;
```



```
else if (wp->packetType == PUSH)
        {
            pushMessage *pushMsg = (pushMessage*)wp->data;

            // Using Table for long topic name (Not needed)
            /*
            MQTTBinding binding[3];
            strncpy(binding[0].client_id, "device", 6);
            // Get deviceNumber from timeslot number
            // binding.client_id[0][6] = getTimeSlot();
            strncpy(binding[0].devCaps, pushMsg->topicName, 5);
            // Find full topic name
            bool isDevicePresent = mqtt_binding_table_get(&binding);
            */

            char topicName[30] = {};
            strncpy(topicName, longTopic, strlen(longTopic));
            strncat(topicName, pushMsg->topicName, 5);
            putsUart0(pushMsg->topicMessage);
            putsUart0("\n");
            if((pubWrPtr + 1) % MAX_PUB_MSG_BUFFER_SIZE != pubRdPtr)
            {
                strcpy(pubMsgBuffer[pubWrPtr][PUB_MSG_BUFFER_TOPIC_INDEX],
topicName);
                strcpy(pubMsgBuffer[pubWrPtr++][PUB_MSG_BUFFER_MSG_INDEX],
pushMsg->topicMessage);
            }
            gf_mqtt_device_pub = getMqttBrokerSocketIndex();

        }
```

When a PUSH message is received from the wireless network, it is parsed and sent through the binding tables to find the long topic name used by adafruit. Due to the limited time to test integration, a shortcut was done to create the long topic name for adafruit. The shorthand name used by the wireless network would be appended to the end of *uta_iot/feeds/* to create the long topic name. The publish message buffer holds up to 10 messages, each with a long topic name and message to be sent to adafruit. The functions to interface with the buffer from ethernet.c are below:

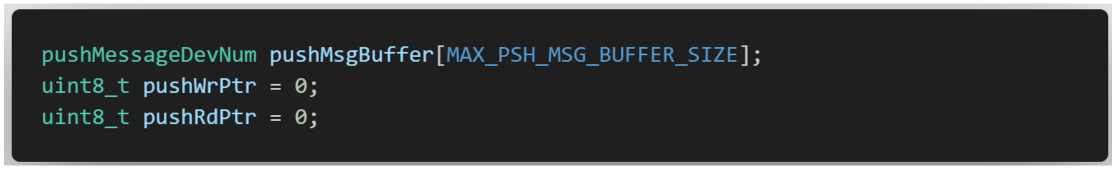
```
bool readPubMsgBuffer(char pubMsg[1][2][30])
{
    if(pubRdPtr == pubWrPtr)
        return false;


    strcpy(pubMsg[0][PUB_MSG_BUFFER_TOPIC_INDEX], pubMsgBuffer[pubRdPtr][
PUB_MSG_BUFFER_TOPIC_INDEX]);
    strcpy(pubMsg[0][PUB_MSG_BUFFER_MSG_INDEX], pubMsgBuffer[pubRdPtr][
PUB_MSG_BUFFER_MSG_INDEX]);
    pubRdPtr = (pubRdPtr + 1) % MAX_PUB_MSG_BUFFER_SIZE;
    return true;


}
bool isPubMsgBufferEmpty(void)
{
    return pubRdPtr == pubWrPtr;
}
```

The *readPubMsgBuffer* returns a boolean based on if a read was successful and saves the publish message into a pointer passed into pubMsg. This function is called when the publish message flag is set. The *queuePubMsgBuffer* function is missing, but is done when a PUSH message is received.

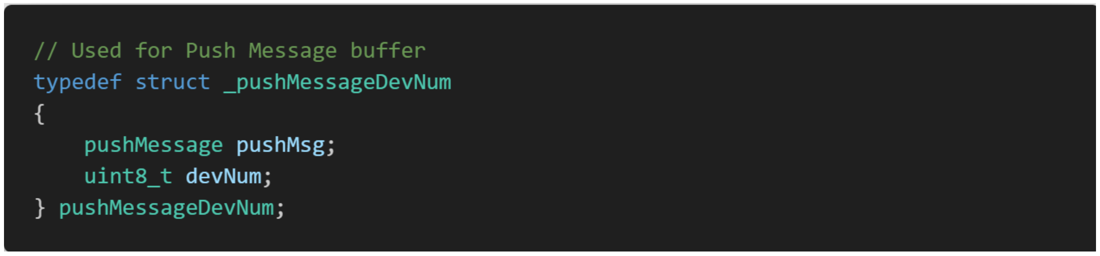
```
if(gf_mqtt_device_pub)
    {
        char publishMsg[2][30];
        bool isValidRead = readPubMsgBuffer(&publishMsg);
        if(isValidRead)
        {
            sendMqttMessage(data, sockets[gf_mqtt_device_pub], PUBLISH, mqttFlags
, (void *)publishMsg, 30, 2);
            // if(isPubMsgBufferEmpty())
                gf_mqtt_device_pub = 0;
        }
        else
            gf_mqtt_device_pub = 0;

    }
```

This will read the next available publish message in the buffer and transmit it to adafruit. The flag would be set to false if the buffer is empty or the current read was invalid.

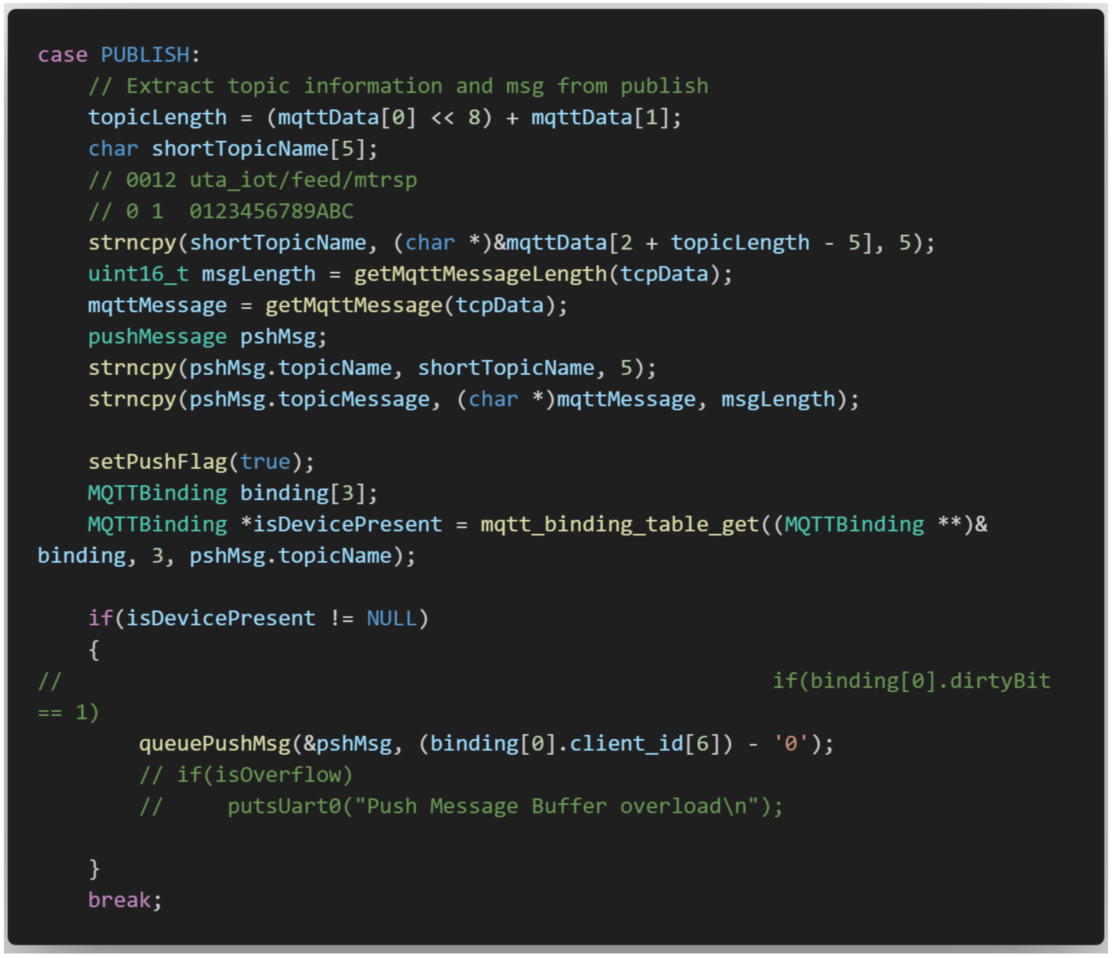
For downlink messages, the following data structure and functions were used:



```
pushMessageDevNum pushMsgBuffer[MAX_PSH_MSG_BUFFER_SIZE];
uint8_t pushWrPtr = 0;
uint8_t pushRdPtr = 0;
```

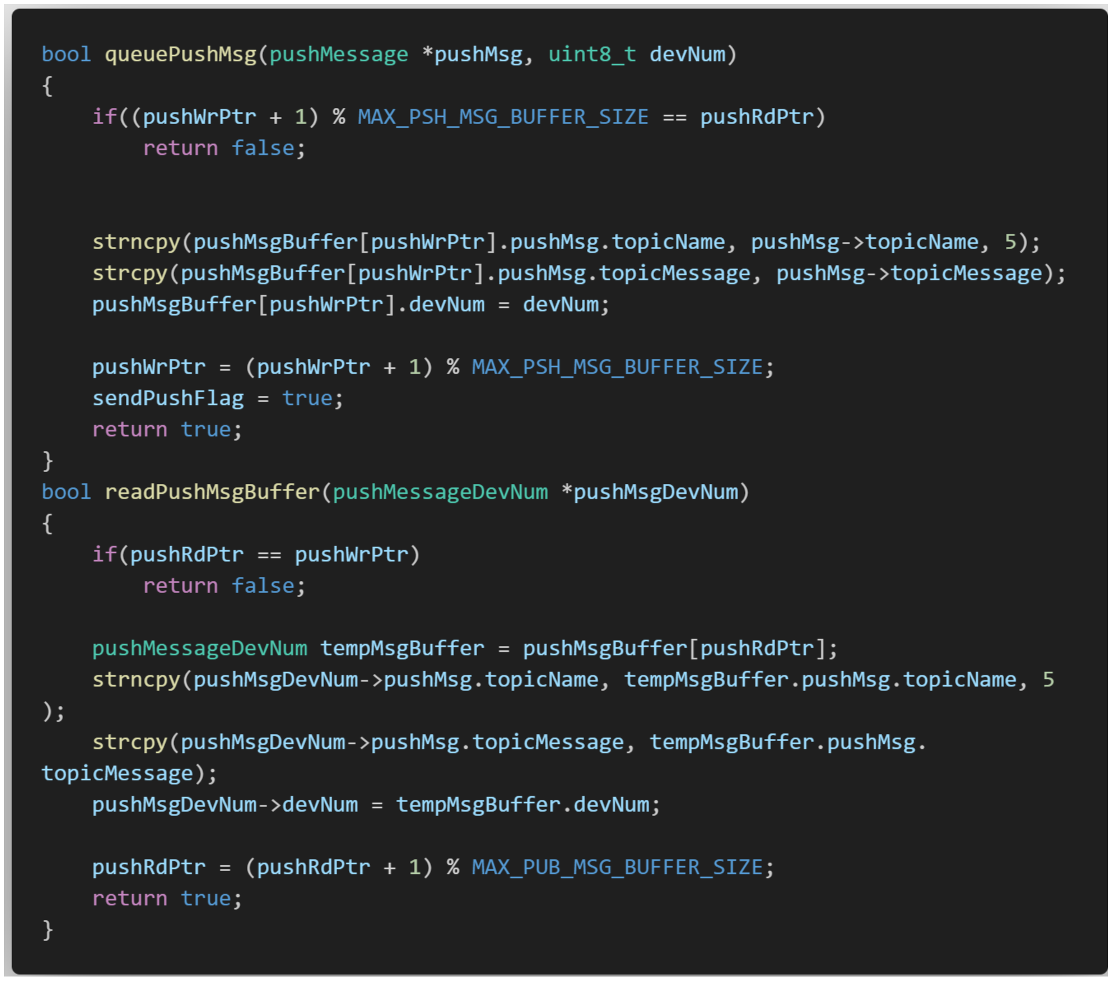


```
// Used for Push Message buffer
typedef struct _pushMessageDevNum
{
    pushMessage pushMsg;
    uint8_t devNum;
} pushMessageDevNum;
```



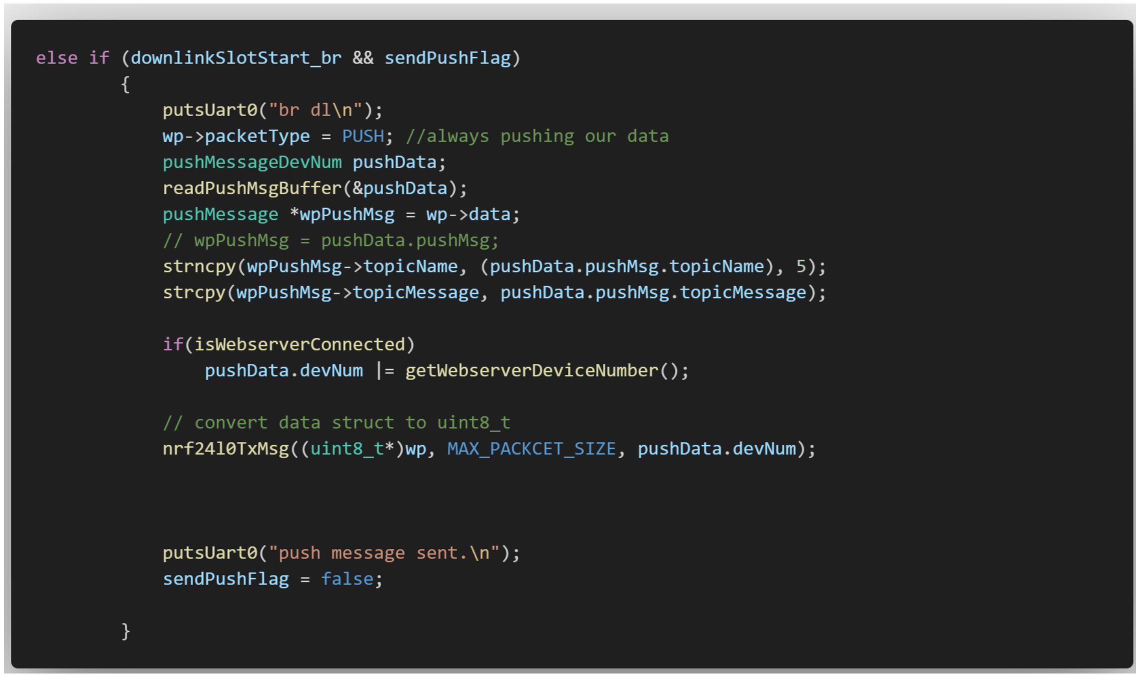
```
case PUBLISH:
    // Extract topic information and msg from publish
    topicLength = (mqttData[0] << 8) + mqttData[1];
    char shortTopicName[5];
    // 0012 uta_iot/feed/mtrsp
    // 0 1  0123456789ABC
    strncpy(shortTopicName, (char *)&mqttData[2 + topicLength - 5], 5);
    uint16_t msgLength = getMqttMessageLength(tcpData);
    mqttMessage = getMqttMessage(tcpData);
    pushMessage pshMsg;
    strncpy(pshMsg.topicName, shortTopicName, 5);
    strncpy(pshMsg.topicMessage, (char *)mqttMessage, msgLength);

    setPushFlag(true);
    MQTTBinding binding[3];
    MQTTBinding *isDevicePresent = mqtt_binding_table_get((MQTTBinding **)&
binding, 3, pshMsg.topicName);

    if(isDevicePresent != NULL)
    {
//                                                  if(binding[0].dirtyBit
== 1)
        queuePushMsg(&pshMsg, (binding[0].client_id[6]) - '0');
        // if(isOverflow)
        //     putsUart0("Push Message Buffer overload\n");

    }
    break;
```

When a publish message is received from adafruit, the shorthand topic name is extracted using the same strategy of taking off the last 5 characters of the long topic name. The shorthand name is then used as a key for the *mqtt_binding_table_get* function to find the device number associated with the topic. The dirty bit in the binding table identifies if the device is an input and should receive the message. No testing was able to be done with using the dirtyBit in the binding table, but the logic is there for it. The PUSH message buffer holds up to 10 messages, each with a short topic name, message, and destination device. The functions to interface with the buffer are below:



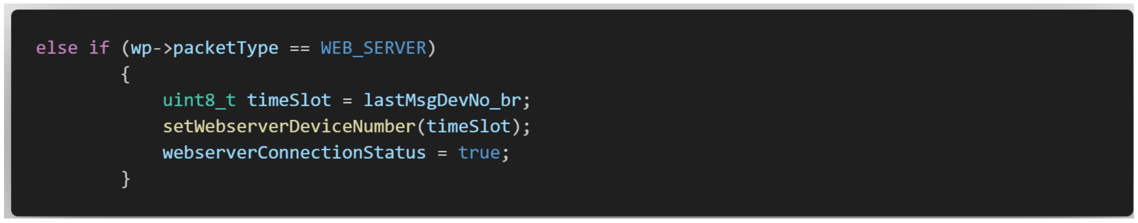
```
bool queuePushMsg(pushMessage *pushMsg, uint8_t devNum)
{
    if((pushWrPtr + 1) % MAX_PSH_MSG_BUFFER_SIZE == pushRdPtr)
        return false;


    strncpy(pushMsgBuffer[pushWrPtr].pushMsg.topicName, pushMsg->topicName, 5);
    strcpy(pushMsgBuffer[pushWrPtr].pushMsg.topicMessage, pushMsg->topicMessage);
    pushMsgBuffer[pushWrPtr].devNum = devNum;

    pushWrPtr = (pushWrPtr + 1) % MAX_PSH_MSG_BUFFER_SIZE;
    sendPushFlag = true;
    return true;
}
bool readPushMsgBuffer(pushMessageDevNum *pushMsgDevNum)
{
    if(pushRdPtr == pushWrPtr)
        return false;

    pushMessageDevNum tempMsgBuffer = pushMsgBuffer[pushRdPtr];
    strncpy(pushMsgDevNum->pushMsg.topicName, tempMsgBuffer.pushMsg.topicName, 5
);
    strcpy(pushMsgDevNum->pushMsg.topicMessage, tempMsgBuffer.pushMsg.
topicMessage);
    pushMsgDevNum->devNum = tempMsgBuffer.devNum;

    pushRdPtr = (pushRdPtr + 1) % MAX_PUB_MSG_BUFFER_SIZE;
    return true;
}
```

Both functions return a boolean based if the read/write was successful (if the buffer is empty/full).



```
else if (downlinkSlotStart_br && sendPushFlag)
        {
            putsUart0("br dl\n");
            wp->packetType = PUSH; //always pushing our data
            pushMessageDevNum pushData;
            readPushMsgBuffer(&pushData);
            pushMessage *wpPushMsg = wp->data;
            // wpPushMsg = pushData.pushMsg;
            strncpy(wpPushMsg->topicName, (pushData.pushMsg.topicName), 5);
            strcpy(wpPushMsg->topicMessage, pushData.pushMsg.topicMessage);

            if(isWebserverConnected)
                pushData.devNum |= getWebserverDeviceNumber();

            // convert data struct to uint8_t
            nrf24l0TxMsg((uint8_t*)wp, MAX_PACKCET_SIZE, pushData.devNum);



            putsUart0("push message sent.\n");
            sendPushFlag = false;

        }
```

Based on the timing provided by the wireless team, a PUSH message is read out of the buffer and sent out to the destination device. If the web server is connected, the destination device bit number is OR'ed with the web server device bit number to have the webserver read the message sent. The web server is identified when a WEB_SERVER wireless packet is received and its device number is maintained by the following code.



```
else if (wp->packetType == WEB_SERVER)
        {
            uint8_t timeSlot = lastMsgDevNo_br;
            setWebserverDeviceNumber(timeSlot);
            webserverConnectionStatus = true;
        }
```

In conclusion, for the forwarding team in the IoT project, many integration issues occurred with very little time for debugging primarily due to the inconsistency in push messages from the devices and wireless packetType parsing done by the wireless team (wp->packetType ==_____). Despite this, with only the barcode scanner device, the code was able to route the device's message up to adafruit and back with no problems. For the final demo, the temperature sensor was able to have its data forward to adafruit too. No integration with the web server messages were tested, but we have hope it would've worked with more consistent devices. Interfacing with the binding tables also proved difficult, but methods were used to circumvent the issues such as extracting the shorthand topic name from the long topic name. There was also an issue that occurred where too many publish messages would be made and adafruit would force close the connection when many devices were connected to the network. Little to no debugging was able to be done on this issue due to the time constraint and low consistency in the system in general, but we think it had to do with the transmission speeds of the web server and devices. For clarity, Ulysses originally had the wireless Device Management role. He provided the initial structure of the device capabilities, which was then modified for integration purposes. He decided to join the bridge forwarding team, as he was told by the wireless API team that the DM tasks had been completed by them already. Over time, it became clear that the wireless DM role's tasks had to be implicitly done by several teams, as many updates and changes were made to the devices code and API code.