Agg Data (LoRa Farms)

Leo Fangmeyer, Peter Harkins, Ricky Pantin, Michael Pitz

Table of Contents

Introduction

Farming is important. The rise of human civilization began with the advent of agriculture. In ancient times, the first farmers allowed for their roaming hunter-gatherer tribes to transform into stationary communities. Because of these farmers providing abundant food, other normal people were able to take up various occupations such as consultants or investment bankers, while the strange ones were able to become engineers. However, farming has never been an easy task. Good and productive agriculture requires the right conditions for different types of crops and livestock. This means that the farmer must always take great care in the upkeep of his land. For most of human history, the condition of farmland had to be closely and painstakingly tracked by individual people working out in the field. This takes valuable time away from the farmer that could be used for things like planning future aspects of the farm, securing business opportunities, or taking part in leisure activities. LoRa Farms seeks to alleviate this problem.

LoRa Farms therefore seeks to implement a low cost and low-power information processing and control system specifically tailored for agricultural applications. This system (and its namesake) relies on LoRa radio modulation for efficient long-range communication. The overarching system consists of a network of devices that interact with each other wirelessly. There are two classes that a device may fall in, either a base station or a field module. The field modules are powered on battery and installed around the user's property, and the base station is plugged into a more permanent power source inside the user's home or workshop, as well as connected to their local WiFi network. Additionally, the field modules are equipped with an array of both digital and analog sensors that monitor various aspects of field conditions, such as light, humidity, temperature, and soil moisture. The modules are also meant to be expandable to the user's preference so that he or she may connect any other desired sensors. The field modules then

send these various sensor readings to the base station after every period of time set by the user, depending on their preference for detail in their data. The readings are sent from the field to the base by means of a LoRa radio packet. Once the message is received, the base station automatically uploads this data to a private webpage that is available to the user. This webpage shows past sensor readings from each field device and has options for the user to manually request a live update from any device. Further, the user has the ability to transmit actuator signals to a field device, which can set off a relay that powers a more complex or cumbersome system like a heat lamp or a retractable shade. Finally, the user has the ability at any time to download past sensor readings into a .csv file, which can be easily manipulated in a spreadsheet program for data analysis.

In its current state, the design for this system meets our expectations fairly well. The most significant aspects, as briefly described above, work generally as intended. The field devices are able to send multiple sensor readings to the base station which has them uploaded to a webpage that is viewable to the user. However, there are some minor hiccups at a few points of integration between the subsystems. Arguably the most noticeable lingering issue is the questionable reliability of LoRa communications between devices. Because of the inclusion of confirmation and acknowledgement messages between the devices, the radio messages will continue to be sent until received, but this does not solve the problem of time between communications. In general, these messages are sent and received fairly quickly, but there is sometimes a noticeable and palpable delay, which could be quite bothersome to the user. A similar downfall of the current iteration of LoRa Farms is that our devices were not able to communicate with each other over two miles away. The furthest successful transmission–without extensive testing–was only around

a half mile. However, this problem could relatively easily be solved with the use of radio repeaters.

Another issue that arises with reliable LoRa communications is the reception of multiple copies of the same message. This is not ordinarily an issue when sending an actuation signal or manually requesting an update to sensor readings, but it can be quite apparent when the base station receives an automatically updated set of readings from the field. In this case, the website will display three or four copies of the same set of readings one after another in the table, which can again be bothersome for the user.

One more issue that still besets LoRa Farms is the accuracy and intelligibility of some of the readings, particularly the battery level, temperature, and humidity. The battery level reading sent from a field module does not seem to be correlated very closely with the actual power (time in correct voltage range) left in the battery. Additionally, the readings from the temperature and humidity sensors are still mostly in their raw form. The user can identify relative differences between readings for these two measurements, but currently they do not correlate to any standard units.

The last major issue with the current iteration of LoRa Farms is that the enclosure for the field devices is not entirely weatherproof. All sides but one are closed off, but the last side merely has a thin piece of plastic that sits in a fairly wide track. The space between the plastic piece and the opening of the board/battery compartment is quite wide, and the track for this piece is wide enough to let water pool up. In its current state, the field device enclosure is still fairly susceptible to adverse weather conditions.

Overall, the current LoRa Farms system works fairly well and can be of some preliminary use to a customer. However, there are some issues that still must be addressed in order to provide a more complete and seamless experience for the user.

Detailed System Requirements

In order to grant smallholding farmers easier access to detailed information about their land, the system must have a network of devices that can communicate via LoRa. The field device should operate with minimal power consumption to allow for long-term use before replacing the power supply (or the entire device). The field device should be able to take in raw sensor data and transmit that data to the base/control device. Either on the user's command or based on programmable logic, the base station should be able to send a signal to the field device that can actuate some process, including but not limited to turning sprinklers on/off, turning lights on/off, opening or closing gates, or dispensing feed for animals. There is also the possibility that the control device will actuate field devices independent of sensor values or user input, such as scheduled feeding times. The field device does not need to do any logic processing, but will be limited to sensing and transmitting data, as well as enacting the actuation when commanded by the control device to some relay device.

Regarding the microcontrollers on each type of device, the field module does not need very much processing power since it will only be receiving and transmitting data. On the other hand, the base station needs a chip that has a bit more processing power along with access to WiFi in order to set up the webpage. Both types of device also need a second integrated chip that will allow communication via LoRa radio modulation.

The base device should be powered from a permanent source such as a standard household receptacle or a USB connection to a computer. also needs to seamlessly communicate with and differentiate between multiple devices in the field. Furthermore, the base station must be able to communicate with field devices that are some distance away, at least two miles. As for the remote device, it must be powered with lithium-ion cells and should last for at least a year or two before needing any replacement or troubleshooting.

The primary user interfaces will be the HTML webpage that will display recorded data. In order to actuate from the base station manually, the HTML page must have a highly accessible UI.

The installation of both the field and base modules must also be relatively simple. The field devices can be placed in predetermined locations from which the property owner wishes to collect data. The circuit board must be in a weatherproof enclosure that protects the device from the elements as well as allow for connections to sensors that must be placed outside the enclosure, such as a soil moisture sensor.

Detailed Project Description

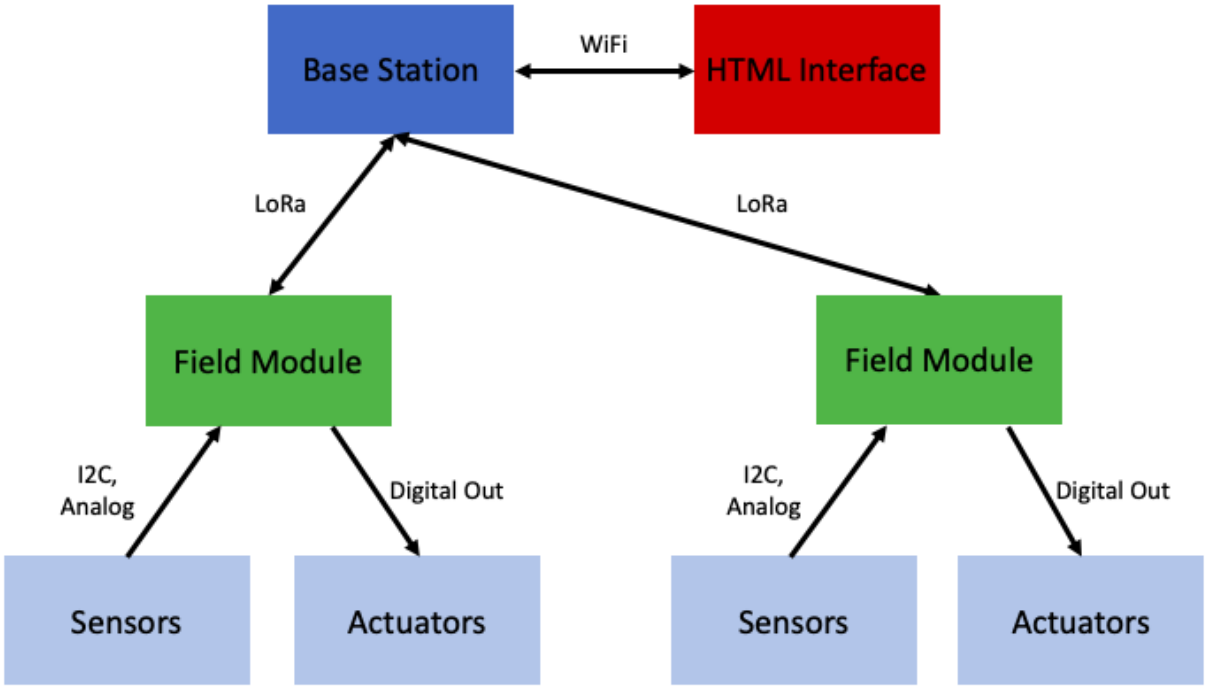
AggData consists of a control system with three primary types of modules. The first module type is the “Base Station” that periodically communicates with the sensor(s) out in the field, or the “Field Module” via LoRa packets. The Base Station receives readings and other information (via LoRa)from the Field Modules and passes it to the web server.

The Field Module devices will contain: (I) a means of monitoring the environment (through use of sensors) (II) a means of communicating with the control center device when necessary (via LoRa) (III) a means of actuating a certain subsystem of the farm.

(I) The Field Module consists of a light sensor, moisture sensor, temperature sensor and a humidity sensor which will be used in order to monitor the environment. (II) The field device's microcontroller periodically sends readings from the sensors to the base station module. It also is meant to send to the base station these readings when one of these values is inappropriate (if the light sensor reading is too low for example). This logic was not implemented in the final system presented; however, the logic would be very similar to that of the timed interrupts used to periodically send information from the field devices to the base module (described later). (III) The field device finally also activates a digital output when a certain system needs to be used given a low or high sensor reading (i.e. turning on the sprinklers in response to a low moisture level). This will be displayed by turning on/off LED's.

The web server consists of a self updating table that displays the periodic sensor readings received by the base station for the appropriate field device module. The user can input which actuators need to be enabled; moreover, the user can also request current sensor readings and the current actuator status of each device to be displayed in a different table. These inputs from the user in the server are passed to the Base Station, which are then sent to the Field Modules via LoRa. Finally, the user can also download all the displayed information into a CSV file.

Subsystems:



The functionality of the overall system is summed up as follows: there are three modules, including a base station and two field modules, that can communicate with each other (though primarily only from field to base and not field-field communications) via LoRa radio protocol. The base station also connects to WiFi to create a user interface (UI) that allows the user to see and control the interactions that the field modules have with incorporated sensors and actuators.

Base

A. Power Supply:

The power supply for the Base Module was to simply plug it into a powered-on computer via micro-USB, though it also could have run from a wall wart or on battery as well. Powering it via USB with the computer allowed for easy debugging and troubleshooting with the serial

monitor, but since the user interface (described later) was established entirely wirelessly, the physical plug-in to the computer is not the sole form of acceptable power for the base module. The power from the wall is then stepped down by a linear voltage regulator to get the voltage to a safe operating Vdd value for the microcontroller.

B. Microcontroller:

The ESP32C3 WROOM02 was the selected microcontroller. It is capable of WiFi connectivity, as well as connecting to a LoRa transceiver for radio transmissions. It also has SPI, Serial, analog in, and I2C capabilities. The SPI allows for use of the RFM95CW for LoRa, while the Serial capabilities allow for serial monitor debugging and troubleshooting, and the analog in and I2C functionality allow for the use of a plethora of potential sensors for environment monitoring. The base station really only needed the SPI for LoRa and WiFi, but the serial proved useful as well throughout the design process. For our board, Tx and Rx were assigned as 21 and 1, respectively; SDA and SCL were 8 and 9; SS = 5; MOSI = 7; MISO = 10; and SCK = 6.

C. LoRa Transceiver:

The LoRa transceiver (RFM95CW) was set up with use of the SPI.h and LoRa.h Arduino libraries. The RFM95CW is a module specifically designed for LoRa transmissions and functionality, with a hand-cut wire antenna used for improving signal quality and reliability.

```
#define ss 5
#define rst -1
#define dio0 20
```

```
// LoRa Setup:
LoRa.setPins(ss,rst,dio0);
while (!LoRa.begin(915E6)) { // waits until LoRa is set up
  Serial.println(".");
  delay(500);
}
LoRa.setSyncWord(0x01); // receive only from paired sender
Serial.println("LoRa Worked");
```

Figure A1. LoRa Pin Assignments; C code

Initially, we set up which pins of the ESP32 connected to specific pins of the RFM95CW LoRa capable chip on the base module, namely, the signal select, reset, and I/O pin used to relay transmission contents to and from our microcontroller. After calling the LoRa.setPins() function from the LoRa library, LoRa is activated, and until it is successfully established, "." is printed to indicate that LoRa is not yet up and running. Then, after LoRa is activated, the SyncWord is set, which is essentially a form of identifier to ensure that the device is only reading data from and sending data to desired devices that have the same SyncWord. After all this is successfully completed, the user is notified via Serial.print().

D. WiFi Capability:

The ESP-32C3-WROOM02 has WiFi capability and an antenna for connecting to wireless networks. In conjunction with the Arduino WiFi.h, WiFiClient.h, and WebServer.h libraries, the base module was connected to a local WiFi network as a WiFi Station. This allows the base station to create its own server with a unique IP address that displays a custom-built html page backed with JavaScript and CSS in order to display desired information and grant the user sufficient functionality.

```
WiFi.mode(WIFI_STA); // connects to existing network
phy_bbpll_en_usb(true);
Serial.println("This boy's a wiFi station");
 WiFi.begin(ssid, password);
 Serial.println("Connecting to ");
 Serial.print(ssid);
 while(WiFi.waitForConnectResult() != WL_CONNECTED){
     Serial.print("."); // wait until wifi connection established
   }
 Serial.println("");
 Serial.print("Connected to ");
 Serial.println(ssid);
 Serial.print("IP address: ");
 Serial.println(WiFi.localIP());
```

Figure A2. Wifi Setup; C Base Station Code

This shows the code used to connect to WiFi. First, the ESP32 is set to Station Mode. The next line (phy_bbpll…) was used to ensure the Serial Monitor JTAG connection was not automatically shut off upon connection to WiFi, which was a challenge that we began to face later on in the project with our own designed boards. Next, the ESP32 connects to the assigned SSID with the necessary password (variables assigned values earlier depending on which network the device is to connect with). The Serial.print's were used to allow us to track the connection to WiFi and ensure that the base module was functioning as desired. After printing the IP Address, we were sure that the module was connected to the internet, and we were able to create a user interface using html after that.

Field

A. Power Supply

The first significant difference between base and field modules is the Power supply. While the base module is generally powered either by wall or computer for steady power and less concern about low power usage, the field module is not in a position of easy power access. Because of this, the field device must run on a battery, which we chose to be a rechargeable 18650 Lithium Ion Cell. This battery generally provides a voltage on the range from 4.2 Volts at full charge down to about 3.7 Volts. This also passes through the linear voltage regulator to drop the voltage to the 3.3V operating Vdd for the ESP32C3.

B. Microcontroller:

This is the same as with the base station. This element of each subsystem is identical in microcontroller choice and configuration.

C. LoRa Transceiver:

Like with the microcontroller, this is the same setup as with the base module; there is no difference between the two in this subsystem element.

D. Sensor Readings and Actuators:

The other critical difference between the field and base modules is the interactions that the field device has with sensor inputs and actuator digital outputs. The setup for sensors took place in two primary ways, one being the simpler setup of an analog input pin–in our case, an analog in pin was used for a soil moisture sensor. It starts with the definition of one pin as an analog read pin (pin 3) then simply requires use of the Arduino *analogRead()* function to read in the value at that pin from the analog sensor.

More complicated than the analog in sensor was the use of I2C sensors, including humidity and temperature sensor: the SHT40-AD1B-R3, with pins 1-4 wired to SDA, SCL, Vdd, GND, respectively, with SDA and SCL wired according to the ESP32 pin assignments (SDA = 8, SCL = 9). The other I2C sensor used is the VEML7700 with pins 1-4 connected to SCL, Vdd, GND, and SDA, respectively.

The software setup for the I2C sensors was also more intensive. The VEML required the SparkFun_VEML7700_Arduino_Library.h, and the SHT40 required the SensirionI2CSht4x.h library, and the use of I2C required the Wire.h library.

```
Wire.begin(); // i2c initialization
Serial.println("Wire began");

if (lightSensor.begin() == false) { // check if VEML7700 is attached properly to the board
  Serial.println("Unable to communicate with VEML7700");
  //    while (1) {
  //      Serial.print("shit");
  //      delay(500);
  //    }
}

Serial.print("Light Sensor began");
lightSensor.setIntegrationTime(VEML7700_INTEGRATION_25ms); // set integration time, modify t
lightSensor.setSensitivityMode(VEML7700_SENSITIVITY_x1); // set sensitivity, again may modif
Serial.println("Light sensor setup good");
```

Figure A3. VEML setup; Field C Code

The light sensor is initialized, alongside I2C, by setting the integration time and sensitivity, and prints the status of the setup to the serial monitor for easier troubleshooting.

```
    // Error checking for SHT-4x:
    uint16_t error;
    char errorMessage[256];
    sht4x.begin(Wire); // begin SHT-4x
    uint32_t serialNumber;
    error = sht4x.serialNumber(serialNumber); // if there is an erro
    if (error) {
        Serial.print("Error trying to execute serialNumber(): ");
        errorToString(error, errorMessage, 256);
        Serial.println(errorMessage);
    } else {
        Serial.print("Serial Number: ");
        Serial.println(serialNumber);
    }
}
```

Figure A4. SHT-4x Setup; Field C Code

This follows a similar procedure to the initialization of the VEML, as the light sensor is introduced to the I2C environment, and will notify the user of success or failure via the serial monitor.

E. Timer Interrupt

Timer interrupts are used in the field modules for a scheduled collection of sensor data, followed by transmission of a packet to the base station.

```
// timer initialization
static hw_timer_t *timer = NULL;
portMUX_TYPE timerMux = portMUX_INITIALIZER_UNLOCKED;
void IRAM_ATTR onTimer() {
  portENTER_CRITICAL_ISR(&timerMux);
  interruptCounter++;
  portEXIT_CRITICAL_ISR(&timerMux);
}
```

Figure A5. Timer Initialization; Field Code

```
// Timer Setup as Interrupt
timer = timerBegin(0, 80, true);
timerAttachInterrupt(timer, &onTimer, true);
timerAlarmWrite(timer, 30000000, true);
timerAlarmEnable(timer);
```

Figure A6. Timer Interrupt Setup; Field Code

The timer is first initialized, then it is set up specifically to be an interrupt timer. Then in the *loop()* function, there is a check to see if the interrupt has triggered, at which point the timer is reset and then data collection is triggered, followed by the packing and writing of the packet containing the newest sensor readings.

```
if (interruptCounter > 0) {
  portENTER_CRITICAL(&timerMux);
  interruptCounter--;
  portEXIT_CRITICAL(&timerMux);

  totalInterruptCounter++;

  Serial.println("An interrupt has occurred. Now going to start sending process. ");
  confirmed = false;
  while (!confirmed) {
    obtainData();
    sendPacket(nack);
  }
```

Figure A7. Interrupt Service; Field Code

LoRa Communications Protocol

A. Packet Design

The radio messages were packetized to ensure that the messages received by the transceivers were sent from sensors within the network. Packets were designed to minimize size, while still carrying a payload with all the desired information.

For the receiver-to-base message, the bytes are as follows:

1. Base ID – address of the basestation

2. Device ID – address of the field device, identifies which device is sending the information
3. Battery level – alerts the user of current battery level on the field device
4. Analog moisture sensor (high byte) – five most significant bits (MSB) of moisture reading
5. Analog moisture sensor (low byte) – seven least significant bits (LSB) of moisture reading
6. Light sensor value (high byte) – seven MSB of lux reading
7. Light sensor value (low byte) – seven LSB of lux reading
8. Humidity sensor value (high byte) – seven MSB of humidity reading
9. Humidity sensor value (low byte) – seven LSB of humidity reading
10. Temperature sensor value (high byte) – seven MSB of temperature reading
11. Temperature sensor value (low byte) – seven LSB of temperature reading
12. Acknowledgment message – identifies if message is a new transmission or an acknowledgement of a received message
13. Actuator 1 – reads in the current Actuator 1 value
14. Actuator 2 – reads in the current Actuator 2 value
15. Actuator 3 – reads in the current Actuator 3 value

For the base to the receiver, the message contains the following bytes:

1. Device ID – specifies which field device the base station is reaching out to
2. Actuator 1 – the value to set Actuator 1 to
3. Actuator 2 – the value to set Actuator 2 to
4. Actuator 3 – the value to set Actuator 3 to

5. Acknowledgment message – identifies if message is a new transmission or an acknowledgement of a received message

The encoding for multi-byte information was done specially, since the server needed to read data as text, meaning that values could only be represented as ASCII characters to the server, which limited them to 7 bits of information, rather than 8 bits. First, the raw sensor value was read in, then processed.

```
firstByteTEMP = int2byte(temperature, 1); // save high byte as first temperature byte
secondByteTEMP = int2byte(temperature, 2); // save low byte as second temperature byte
```

Figure B1. Bit packing the raw values; Field Device

```
byte int2byte(int a, int byteNum) {
  byte returnByte;
  int b = map(a, 0, 65535, 0, 16383);


  if (byteNum == 1) { // if byte is the high byte
    returnByte = (b >> 7); // bit shift rightward the integer 'a' to save as byte
  }
  else if (byteNum == 2) { // if byte is the low byte
    returnByte = (0x007F & b); // save last 7 bits of int
  }

  return returnByte; // return decomposition as a byte
}
```

Figure B2. int2byte(); Field Device

The *int2byte()* function reads in the 16 bit raw value and then breaks it up into 2 bytes, but each with only 7 bits of information. This is because ASCII can only handle 7 bits, and the values passed to the server from the base station will be in text form, requiring them to be represented by 7 bits of information or less to avoid losing information when they are encoded and interpreted as text by the server.

The message from base to field is slightly different, since the field device does not need to read in as much information from the base, but it consists only of actuator values that the base

station wishes to control at the targeted field module. Ergo, the packet send from base to field is smaller, but follows the same principles without the bit-packing necessary for 2 byte chunks of information.

B. Sending:

When a packet is sent using the *sendPacket()* function, most of the values sent in the LoRa packet are global variables, but there is one function argument that determines the message ID, or acknowledge bit of the message. For any transmission that contains data to be interpreted and is not merely an acknowledgement (*ack*) of a previously received message, the device listens for confirmation of receipt of its message from its intended target, meanwhile continuing to alternate between sending and listening until receiving confirmation.

```
void sendPacket(byte acknowledgeYN) {
  LoRa.beginPacket();
  LoRa.write(baseID); // basestation ID -- 0
  LoRa.write(deviceID); // field device ID number -- 1
  LoRa.write(batteryLevel); // battery level reading -- 2
  LoRa.write(firstByteAR); // analog sensor high byte -- 3
  LoRa.write(secondByteAR); // analog sensor low byte -- 4
  LoRa.write(firstByteLIGHT); // light sensor high byte -- 5
  LoRa.write(secondByteLIGHT); // light sensor low byte -- 6
  LoRa.write(firstByteHUM); // humidity sensor high byte -- 7
  LoRa.write(secondByteHUM); // humidity sensor low byte -- 8
  LoRa.write(firstByteTEMP); // temperature sensor high byte -- 9
  LoRa.write(secondByteTEMP); // temperature sensor low byte -- 10
  LoRa.write(acknowledgeYN);  // acknowledgement -- 11
  LoRa.write(act1val); // actuator values -- 12
  LoRa.write(act2val);  //actuator values -- 13
  LoRa.write(act3val); //actuator values -- 14
  LoRa.endPacket();

  Serial.println("Sent!");

  if (acknowledgeYN != ack) { // if written as a data packet, check to
    Serial.println("check send");
    checkSend();
  }
```

Figure B3. sendPacket(); Field Device

The *checkSend()* function that is invoked inside of the *sendPacket()* function is repeated after every time a *nack*, or data-carrying message is sent. *checkSend()* also functions as part of the "Confirmation" element of the LoRa transmission subsystem.

```
void checkSend() { // check to listen for acknowledgement message from basestation for some time
  for (int i = 0; i < 8096; i++) {
    int packetSize = LoRa.parsePacket();
    if (packetSize) {
      Serial.println("packet found in check send");
      while (LoRa.available()) {
        String confirmationMessage = LoRa.readString();
        if (deviceID == confirmationMessage[0]) {
          Serial.println("this message is for me");
          if (confirmationMessage[5] == ack) { // if message contains acknowledgement byte, stop sending
            Serial.print("Confirmation: 0x");
            Serial.println(confirmationMessage[5], BIN);
            confirmed = true; // confirmed = true will stop the field module from continuing transmission
            i = 8096; // when message has been received, exit loop and stop listening for an acknowledgement
          }
        }
      }
    }
  }
}
```

Figure B4. checkSend(); Field Device

checkSend() functions by looping very rapidly through a search for a LoRa packet. Upon receipt, the function ensures that the 0th element of the message read in matches the *deviceID* of the module that received the message. Assuming that this module is the intended recipient, then it checks to make sure that the message is an *ack* message, not carrying significant data but rather simply a confirmation message sent from the recipient of the significant, primary transmission. After confirmation is achieved, the device terminates the loop of *sendPacket()* and *checkSend()*.

There are no significant differences between the sending and receiving functions for the Base and Field modules, nor the protocols they employ for transmission reliability. The primary difference is packet size, and therefore, the array location of the acknowledgment byte.

C. Receiving:

On the receiving side of the system, the devices perform checks every time through the *loop()* function for a packet. Before decoding the message, the device checks to make sure that the message is intended for itself, similar to in *checkSend()* as described above. If the device ID

byte does not match the device ID of the receiver, the receiver exits out of the parsing loop and continues to listen for new transmissions. If the device IDs do match, then the packet is unpacked.

```
void readPacket() {
  int packetSize = LoRa.parsePacket(); // check for incoming packet
  if (packetSize) { // if there is a packet:
    Serial.println("Homie my cracka is a dog (that is to say we gotsa package");
    while (LoRa.available()) {
      String receivedMessage = LoRa.readString(); // save packet as an array of byte
      if (deviceID == receivedMessage[0]) { // first byte within the message contair
        // ****************** CHANGED THESE IF CONDITIONS TOO ****************
        if (receivedMessage[5] == nack) {
          Serial.println("nack message received, setting actuators"); // check if th
          act1val = receivedMessage[1]; // value for actuator 1
          Serial.println(act1val);
          act2val = receivedMessage[2]; // value for actuator 2
          Serial.println(act2val);
          act3val = receivedMessage[3]; // value for actuator 3
          Serial.println(act3val);
          act4val = receivedMessage[4]; // value for actuator 4
          Serial.println(act4val);
```

Figure B5. readPacket() initial processing; Field Code

If the message has reached the intended receiver and is not merely an *ack* message, then it is acted upon, and actuators (for the field device) are adjusted appropriately.

D. Confirmation

Also included in *readPacket()* is the response–the acknowledgment back to the initial sender to indicate that the message has been received and no longer needs to be sent repeatedly.

```
      for (int i = 0; (i <= 15); i++) {
        sendPacket(ack); // send acknowledgement packet 15 times to ensure acknowl
        delay(2); // pause for 2 milliseconds for message to finish sending
      }
    }
    // ************* NEW STUFF ****************
    else if (receivedMessage[5] == req) {
      Serial.println("status request received");
      obtainData();
      for (int i = 0; (i <= 15); i++) {
        Serial.println("Status update sent");
        sendPacket(nack); // send acknowledgement packet 15 times to ensure acknow
        delay(2); // pause for 2 milliseconds for message to finish sending
      }
```

Figure B6. Confirmation send in readPacket(); Field Code

The confirmation is simply designed to send packets targeted specifically for the base station, since in the first iteration of our project, that is the only device that sends messages intended for the field modules, so the target device for all of the field confirmation messages is hard-coded as the *baseID*. Essentially, *sendPacket(ack)* is called 15 times for redundancy. In the event that none of the 15 confirmation messages is received by the initial sender, then the initial sender will simply send again, triggering another 15 confirmation messages being sent, and this loop continues until successful confirmation.

Notice also the *else if* statement involving a *req* in the acknowledgement byte. This is another type of transmission that is neither *ack* nor *nack*, and is explained in greater detail in the server operation, as it is only used as a direct result of server UI use.

The *readPacket()* function is also different for the Base station, but that, too, is detailed later on in the server section.

HTML User Interface

A. Server Setup:

The first step in creating the user interface was to establish a server using the IP address of the ESP32 in WiFi Station Mode.

```
WebServer server(80);
```

Figure C1. Server creation; C code

```
// Server Setup:
server.on("/", handleRoot);      // Display page
server.on("/readADC", handleADC); //check for updated packet values
server.on("/dev_sel", deviceSel); // use this to select which device actuation is being performed on
server.on("/heat_sel", heatSel); // pass the value of the heatlamp actuator
server.on("/shade_sel", shadeSel); //pass value of shade actuator
server.on("/sprink_sel", sprinkSel); //pass value of sprinkler actuator
server.on("/request", updateRequest); // call this function when you want a live update and push the update button

 server.begin(); //Start server
 Serial.println("HTTP server started");
```

Figure C2. C-JS links; C code

Various functions are linked to the server host that define the interaction between the C code and the JavaScript code in the index.h file included in the main base station code file. Each of these links to a particular function in C as well as a function in JavaScript to coordinate passing information between the ESP32 board and the server host. Then, the server is started and the user is notified via serial printout. In our *loop()* function, we call *Server.handleClient()* to ensure constant maintenance of the server display.

Lastly, on the server end, the JS function *getData()* is called on an interval of 100ms:

```
setInterval(function() {
  // Call a function repetatively with 2 Second interval
  getData();
}, 100); //2000mSeconds update rate
```

Figure C3. setInterval; index.h

This *getData()* function is what feeds into the updating of the data log, explained in greater detail below.

B. HTML/CSS Display Page:

Various CSS tools are used in conjunction with HTML in order to construct the webpage. Largely consisting of headings and other assorted text, buttons, and tables, the dashboard is meant to give the farmer or landowner both better information and control of his or her property.

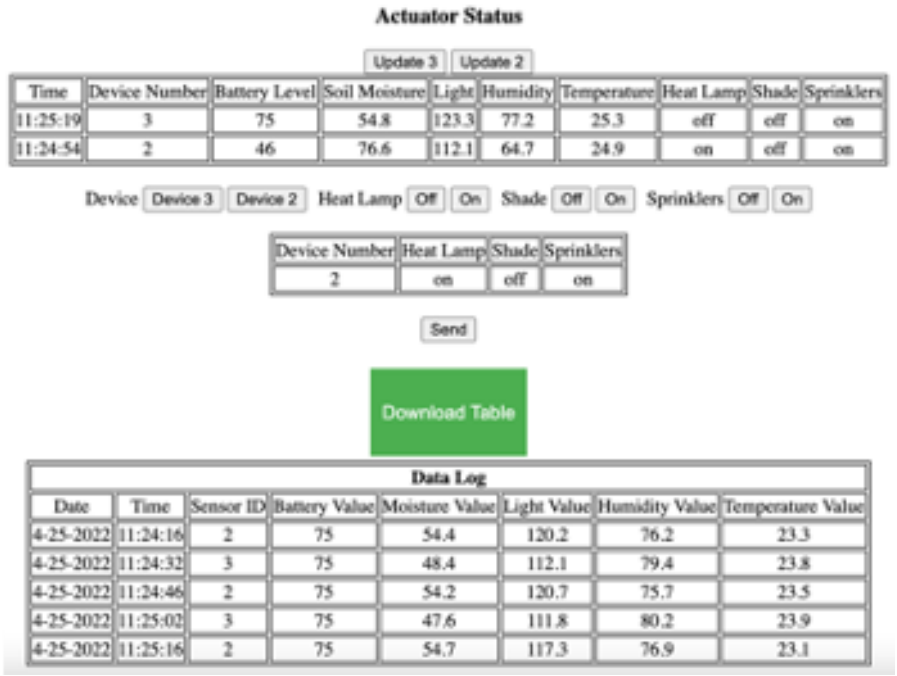


Figure C4. Web page User Interface

C. Data Log Viewing:

The Data Log feature allows a user to view many sensor values recorded over an extended period of time. It consists of a table that indicates the time of the readings, which device the reading is coming from, the battery level of that field device, and each of the pertinent sensor values- in this case, temperature, humidity, light, and soil moisture. It appends a row to the existing table only upon receipt of new information, and a button above it allows for easy downloading of the table as a .CSV file for processing in Excel or similar software. As explained above, the *getData()* function is called every 100ms on the server side:

```
function getData() {
  var xhttp = new XMLHttpRequest(); //some AJAX request has a variable name xhttp
  xhttp.onreadystatechange = function() {            //define a function called onreadystatechange
    if (this.readyState == 4 && this.status == 200) {       //check to see when readyState indicates finished and status


      variable = this.responseText; //variable = testArray
```

Figure C5. Start of getData() JS function; index.h

This establishes the function as an AJAX request, meaning that the server will refresh select elements (as controlled within this function) on this webpage without requiring a refresh of the entire page. The use of AJAX, or Asynchronous Javascript and XML, creates a dynamic webpage that is more convenient to use. At the end of the *getData()* function, the AJAX request is linked to the C function *handleADC* by the server argument *readADC* and the *server.on("/readADC, handleADC)* line above in the C code.

```
xhttp.open("GET", "readADC", true);
xhttp.send();
```

Figure C6. End of getData() (JS function in index.h)

```
void handleADC() {
 server.send(200, "text/plane", testArray); //Send ADC value only to client ajax request
 testArray[0] = 0x69; //arbitrary value that is not 0x01 like the basestation value
 // delay(1000);
}
```

Figure C7. handleADC(); C code

handleADC() passes *testArray* to the server, which is then read into the variable named *variable* in the Javascript function *getData()*. *testArray* is an array full of sensor values received from a field device to be displayed in the table on the base station's server html page. The argument *testArray* is passed to the server and is identified as *this.responseText* on the JS side of things:

```
variable = this.responseText; //variable = testArray

  if(this.responseText.charCodeAt(0) == "0x01") {
    //define variable types as rows and cells
    var newRow = document.createElement("tr");
    //var newCell = document.createElement("td");
    var newDate = document.createElement("td");
    var newTime = document.createElement("td");
    var newID = document.createElement("td");
    var newBat = document.createElement("td");
    var newAnal = document.createElement("td");
    var newLight = document.createElement("td");
    var newHum = document.createElement("td");
    var newTemp = document.createElement("td");
```

Figure C8. getData() table creation; index.h

The conditional is a check to ensure that the packet is original. As shown in the *handleADC* code in Figure C7, after the first time that *testArray* is sent to the server, the 0th element is changed to 0x69, which is not equal to 0x01. Until a new *testArray* is read in, that conditional will reject any repeated actions. Inside the conditional, then, an html element *newRow* is created with the tag characteristic of a row of a table, <tr>. Next, multiple table data entries are created, still without values assigned, but all of type <td>, meaning they can be appended into a row of a table.

```
//populate cells
newDate.innerHTML = get_date();
newTime.innerHTML = get_time();
newID.innerHTML = this.responseText.charCodeAt(1);
newBat.innerHTML = this.responseText.charCodeAt(2);

//processing 2-byte values
var analogValue = Math.round(( (this.responseText.charCodeAt(3) << 7) | (this.responseText.charCodeAt(4)) ) * (65535/16383))/100;
var lightValue = Math.round(( (this.responseText.charCodeAt(5) << 7) | (this.responseText.charCodeAt(6)) ) * (65535/16383))/100;
var humValue = (Math.round(( (this.responseText.charCodeAt(7) << 7) | (this.responseText.charCodeAt(8)) ) * (65535/16383))/100);
var tempValue = (Math.round(( (this.responseText.charCodeAt(9) << 7) | (this.responseText.charCodeAt(10)) ) * (65535/16383))/100);
var tempComb = (this.responseText.charCodeAt(9) << 7) | (this.responseText.charCodeAt(10));
```

Figure C9. Table Cell Population and Bit Unpacking; index.h

The first few cells are then populated with easily available values from the transmission packet received and passed to the server. As shown below, *get_date()* and *get_time()* are simple functions to assign the just-created table cells with values, and the one-byte values (sensor ID and battery value) are read straight into their corresponding cells as well. Additionally, the sensor values that had to be transmitted as two bytes due to LoRa limitations–such as *analogValue, lightValue,* etc–are unpacked and reverted back to their true value, rather than the values they were assigned for transmission.

```
function get_date() {

  var today = new Date();
  var date = (today.getMonth() + 1) + '-' + today.getDate() + '-' + today.getFullYear();
  return date;

}

function get_time() {
  var today = new Date();
  var time = today.getHours() + ":" + today.getMinutes() + ":" + today.getSeconds();
  return time
}
```

Figure C10. get_data() and get_time() functions

After unpacking the encoded bytes, the remainder of the created cells are populated with their corresponding values:

```
//plugging 2-byte values into their cells
newAnal.innerHTML = analogValue;
newLight.innerHTML = lightValue;
newHum.innerHTML = humValue;
newTemp.innerHTML = tempValue;

newRow.append
newRow.append(newDate);
newRow.append(newTime);
newRow.append(newID); //new
newRow.append(newBat);
newRow.append(newAnal);
newRow.append(newLight);
newRow.append(newHum);
newRow.append(newTemp);

document.getElementById("rows").appendChild(newRow); //rows is the name of the body, so we are adding a new row to the body
```

Figure C11. Populating Cells and Appending Row and Table; index.h

After populating each of the created cells, the *newRow* element is populated with each of those cells, one appended after the other to the row. After all the cells have been strung together to make *newRow*, then that *newRow* is appended to the end of the table, which is known by its ID *rows* as seen in the *document.getElementByID()* call. The data log's function, then, is this: any time a new value of *testArray* is passed to the server, the values of *testArray* are unpacked appropriately and assigned to the appropriate cells to build the next row of the data log table before simply adding that row onto the end of the pre-existing table.

D. Live Data Viewing:

Live data viewing follows a similar path to the table update, this time through the function *requestReadings()*, which is passed a device number as its argument. *requestReadings()* is called by a button push, either the “Update 2” or “Update 3” Button:

```
<h3> Actuator Status </h3>
<button class = "button" onclick = "requestReadings(3)";> Update 3 </button>
<button class = "button" onclick = "requestReadings(2)";> Update 2 </button>
```

Figure C12. Update Buttons Code; index.h

```
function requestReadings(deviceNumber) {
  var request = new XMLHttpRequest();
    request.onreadystatechange = function() {
      if(this.readyState == 4 && this.status == 200) {
```

Figure C13. requestReadings(); index.h

This function is linked via “request” to the *updateRequest()* C function in Figure C2, and passes the argument *deviceNumber* as an input to the server and the JS backing it to the C code.

```
request.open("GET", "request?device="+deviceNumber, "true");
request.send();
```

Figure C14. Link between requestReadings() (index.h) and updateRequest (C code); index.h

In the linked C function, *updateRequest()*, the variable *deviceNumber* is saved as the argument passed from the server, *deviceNumber*, the initial argument passed into the function *requestReadings()*.

```
void updateRequest() {
  String deviceNumber = server.arg("device");
  if(deviceNumber == "3"){ Serial.println("Requesting from Device 3");
    deviceID = 0x03;
  }
  else if (deviceNumber == "2"){ Serial.println("Requesting from Device 2");
    deviceID = 0x02;
  }
  confirmed = false;
  while(!confirmed){
    packetWrite(req);
    Serial.println("Status Request Sent");
  }

  server.send(200, "text/plane", statusUpdate);
  // delay(1000);
}
```

Figure C15. updateRequest(); C code

The argument *deviceNumber* that gets passed from server to the C code here is dependent on which button the user pushes: the “Update 2” or “Update 3” Button, each corresponding to the device with the respective ID number. *updateRequest()* checks which device the user has requested readings from and updates the global variable *deviceID* accordingly. After that, the function implements the standard packet send protocol, with the message identifier as a *req* message to show that the base station is sending the field device a request for readings. In the sending protocol, *packetWrite()*, there is a call of the *checkSend()* function, which upon recognition of a response to a *req* message, updates the variable *statusUpdate*, which is then passed from the Base station to the server to then be read and acted upon back in the *requestReadings()* JS function.

```
if (this.responseText.charCodeAt(1) == 2) {
  document.getElementById("levelBat2").innerHTML = this.responseText.charCodeAt(2);
  document.getElementById("moistSoil2").innerHTML = Math.round(( (this.responseText.charCodeAt(3) << 7) | (this.responseText.charCode
  document.getElementById("light2").innerHTML = Math.round(( (this.responseText.charCodeAt(5) << 7) | (this.responseText.charCodeAt(6
  document.getElementById("humidity2").innerHTML = (Math.round(( (this.responseText.charCodeAt(7) << 7) | (this.responseText.charCode
  document.getElementById("temp2").innerHTML = (Math.round(( (this.responseText.charCodeAt(9) << 7) | (this.responseText.charCodeAt(1
  if(this.responseText.charCodeAt(12) == 0){
    document.getElementById("heatLamp2Stat").innerHTML = "off";
  }
  else if (this.responseText.charCodeAt(12) >= 1){
    document.getElementById("heatLamp2Stat").innerHTML = "on";
  }
  if(this.responseText.charCodeAt(13) == 0){
    document.getElementById("shade2Stat").innerHTML = "off";
  }
  else if (this.responseText.charCodeAt(13) >= 1){
    document.getElementById("shade2Stat").innerHTML = "on";
  }
  // document.getElementById("shade2Stat").innerHTML = this.responseText.charCodeAt(13);
  if(this.responseText.charCodeAt(14) == 0){
    document.getElementById("sprink2Stat").innerHTML = "off";
  }
  else if (this.responseText.charCodeAt(14) >= 1){
    document.getElementById("sprink2Stat").innerHTML = "on";
  }
```

Figure C16. requestReadings() table update; index.h

After the user has pushed an “update” button on the server, thereby sending a request from the base station to the specified field module, the base station receives a packet from the field module and passes that message to the server. If *this.responseText* (the argument passed to the server) is a packet regarding device 2, then the device 2 values are all updated–not just sensor values, but the update function also reports the current status of the actuators associated with that device. Likewise, the analogous actions are carried out for a device 3 request.

E. Controls/Actuation:

The actuation capabilities are quite similar to the live update request functionality. The user can push a number of buttons to assign multiple values to be passed from the server to the base station, which then sends those values to the field device, which then reads that packet and turns on/off certain actuators onboard that module (represented by LEDs in our circumstance).

```
<div class="wrap-flex", style = "text-align: center;">
    <div class = "box", style = "margin: 0 5px;">
      <label for = "deviceNum">Device</label>
      <!-- <select name = "deviceNum" id ="deviceNum">
        <label> Device Number </label>
      </select> -->
        <button onclick="device_select(3)"; > Device 3 </button>
        <button onclick= "device_select(2)";> Device 2 </button>
      </select>
    </div>
    <br>
    <div class = "box", style = "margin: 0 5px;">
      <label for = "heatLamp">Heat Lamp </label>
      <!-- <select name = "heatLamp" id ="heatLamp"> -->
        <!-- <label> Heat Lamp Status </label> -->
        <button onclick = "heatLamp_set(0)";> Off </button>
        <button onclick = "heatLamp_set(1)";> On </button>
      </select>
    </div>
```

Figure C17. Actuator Control Buttons; index.h

By use of these buttons (and the ones for shade and sprinklers), the user can assign the desired actuator states and then click the “Send” button to call the *pass()* function:

```
function pass(){
 var dev = new XMLHttpRequest();
  dev.onreadystatechange = function() {
    if (this.readyState == 4 && this.status == 200) {
//      selectedDevice;
//      heatLampLevel;
//      shadeActLevel;
//      sprinklerActLevel;

    }
  };
  dev.open("GET", "dev_sel?state="+selectedDevice, true);
  dev.send();
  var heat = new XMLHttpRequest();
  heat.open("GET", "heat_sel?state="+heatLampLevel, true);
  heat.send();
  var shade = new XMLHttpRequest();
  shade.open("GET", "shade_sel?state="+shadeActLevel, true);
  shade.send();
  var sprink = new XMLHttpRequest();
  sprink.open("GET", "sprink_sel?state="+sprinklerActLevel, true);
  sprink.send();
}
```

Figure C18. Pass() Function; index.h

The *pass()* function links to 4 different C functions, each being passed a different variable that is assigned in JavaScript by the button options described above. Each of these 4 C functions follows closely along with the *sprinkSel()* function.

```
void sprinkSel() {
  String selectedSprink = server.arg("state");
  Serial.print("sprink: ");
  Serial.println(selectedSprink);
  if(selectedSprink == "1"){
    act3val = 0xFF;
  }
  else{
    act3val = 0x00;
  }

  confirmed = false;
  while(!confirmed){
  packetWrite(nack);
  Serial.println("NACK message sent");
  }

}
```

Figure 18. sprinkSel() function; index.h

The *sprinkSel()* function, like the *heatSel, deviceSel, and shadeSel* functions, reads the corresponding server argument "state" to determine what value to assign to the respective actuator value. Since *sprinkSel()* is called last, all the actuator values have been assigned for the desired device by the other 3 linked C functions, and then *sprinkSel()* includes the send protocol, this time with a *nack* message, indicating that the message has information that needs to be read in and interpreted (i.e. it is not simply an acknowledgement of receipt of a message from another device). The targeted field device will receive the message, send confirmation to the base station, and then assign the actuator values according to the values sent in the packet from the base station as assigned by the user's control via the server.

System Integration Testing

The integrated set of subsystems was tested by having a field device read from both an analog and a digital sensor and sending those measurements to the table on the HTML webpage. A moisture sensor (analog) and a light sensor (digital) were used in the test. The readings from these sensors were automatically sent to the base station over LoRa which utilized our packet design, as described above. The field module kept transmitting this message until it received a confirmation message from the base station.

On the base station side of things, the device was powered on and looking for packets from the field module. Upon receipt, the base would check to ensure it was the desired recipient, and send confirmation of receipt (via *ack* message) to the field device. After receiving a data-carrying message, the base station then would pass values to the server, where they would be unpacked via JavaScript and displayed on the server.

This meant that the system successfully read in sensor values to the field device. Repeated successes meant that the system displayed sufficient reliability in LoRa transmissions. Serial prints throughout ensured that the system was functioning according to expectations and not succeeding just by chance. The display on the webpage meant success in creating a UI, especially with the dynamic table that appended with each new reading. By updating on a timed interrupt from the base, that allows the owner control of how frequently he or she wants updates on their property. Additionally, the download button on the webpage means that the system has offline data storage. The update buttons provide live feedback optionality, and the actuator control also grants the owner power over the system and proves the system is not merely for passive information collection, but that it can also function as a truly useful tool and time-saver for farmers and homeowners, too.

User's Manual/Installation Manual

The setup and installation of LoRa Farms is quite simple. When the user brings the system home, almost all functionality will already be present without any additional effort. However, there is a small number of actions the user must take before the entire system is up and running.

Firstly, the field devices must be connected to a computer via the mini-USB port on the circuit board. Using an integrated development environment such as Arduino, the user should adjust the interrupt length in order to specify the time interval at which they want the field devices to transmit to the base station. Then, the field devices, in their enclosures, must be installed at the user's preferred locations around his or her property. It is recommended that these enclosures be fastened to a post in some way so as to be in a feasible position for all sensors to work properly. This is also the time where the user should install any additional sensors that he would like connected to the field device (Note: if additional sensors are installed, they must be programmed into the field device code). The user must also ensure that an 18650 lithium ion cell is properly installed in the battery holder so the field device is able to read sensors and transmit messages.

Next, the user should return to his home and pick a fitting location to install the base station. The most convenient location would be connected to a desktop computer via the mini-USB port on the circuit board. This would allow the base station to have both continuous power and the ability to be programmed, which is necessary for basic functionality. However, once programmed, the base station could also feasibly be connected to a standard wall receptacle.

Once the base station is connected to a power source, it is ready to be programmed. This is most easily done with an integrated development environment such as Arduino. Two things in the code must be adjusted by the user in order for them to see their desired functionality: the SSID variable and the WiFi password variable. These variables must be changed in order to match the user's local WiFi network. Otherise, the base station will not be able to establish a web server that can interface with the user through HTML. LoRa Farms should now be fully functional and transmitting sensor readings from the field.

The user will know if the product is working if they can first access the HTML webpage and then see if the table on the site is automatically updating itself with new sensor values after every time interval (which the user established at the beginning of the setup). If this is all happening, then the system is most likely working correctly. One type of issue that could arise is if some sensors, or the board itself, on a field device begin to malfunction and false readings are sent to the base station. This issue would be more difficult for the user to realize, but it could be noticed if they are familiar with the typical range of correct readings. Another issue that might be relatively commonplace is that the webpage stops updating sensor values from one or more field devices. In this case, the battery for the aforementioned device is probably depleted and must be recharged or replaced.

For the user to troubleshoot LoRa Farms, they must take a base station and field module and connect them to computers so that their serial prints can be viewed and so that the devices can be reprogrammed if necessary. The most intuitive troubleshooting process is for the user to insert serial prints in different blocks of code for both devices to ensure that each block is being run through.

To-Market Design Changes

There are a number of changes to our design that would be ideal to add before the product goes to market. First and foremost, the current iteration of the circuit board would need to be redesigned. As it stands, for LoRa Farms to be functional it needs a number of additional jumper cables soldered onto the board in order to achieve both I2C sensor capabilities and LoRa radio communication. The next iteration of the board design would need these connections already traced in order to eventually bring down production costs. In the same vein, we might also consider designing separate boards for the field and base modules in order to have them more specialized. This would be especially useful for the base station which has no need for sensors on its board. For the field device, a new board could be designed on which it is even easier to connect additional sensors and actuators.

Another design change would be to include with the LoRa Farms package a number of repeater modules that would be used in order to facilitate the communication between base and field modules that are further away from each other than the current half-mile range.

Another change that should be implemented before going to market would be a more spacious and effectively weatherproof enclosure for the field devices. In its current state, the enclosure is quite cramped, which makes it difficult for the user to take out the board and battery in order to replace or troubleshoot. More space inside the enclosure would help alleviate this problem. Additionally, the current enclosure is not entirely weatherproof. The main reason for this is the removable plastic panel that allows the user to replace and troubleshoot the field device. This panel must have a tighter fit or have a better seal before LoRa Farms can be put to market.

One final change that would be applied to LoRa Farms before it is brought to market would be to grant the user the ability to change, from the server, the time interval at which sensor readings are transmitted to the base station. In its current state, the user must reprogram the device themselves in order to change the time interval, and this can be a very cumbersome process.

Some troubleshooting may be necessary. The use of the console in the webpage and serial monitor would help best with this, as they can indicate by their printouts (or lack thereof) how the system is functioning. And as elementary and cliché as this sounds, often the trick may just be to disconnect the devices from power and then reconnect. At times, the RFM95CW LoRa Module seemed to simply stop working, despite receiving no error messages and the serial monitor indicating (seemingly) normal functionality, until it would become clearly apparent that the LoRa transmission simply were not being found at all. The solution here, more often than not, was simply to power off, then back on.

Conclusions

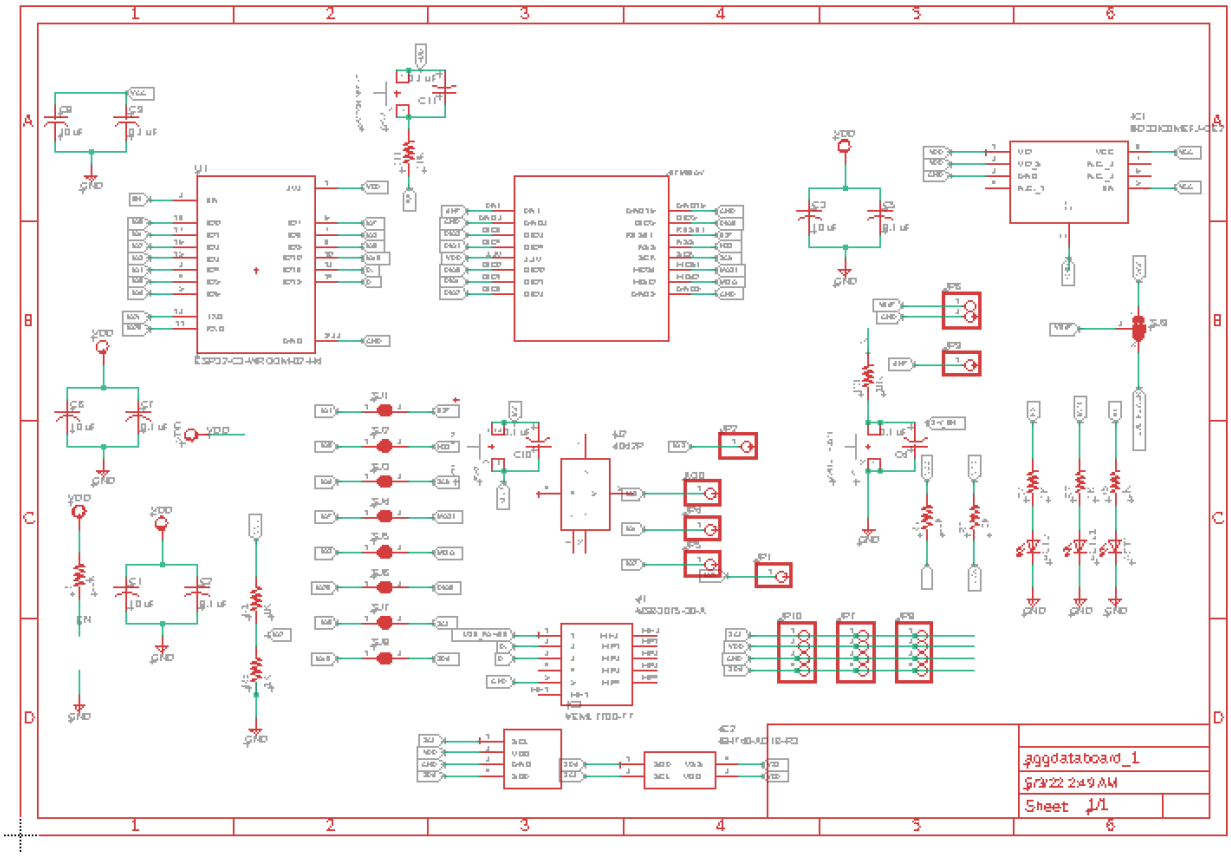
The design of LoRa Farms, the radio-assisted agriculture data system, was much more difficult than any one of us anticipated. The most difficult aspect of this design process was figuring out the best ways to troubleshoot. The chips used were prone to many bugs that were not well documented, forcing us to spend a lot of time figuring out detours rather than getting over a roadblock. The ESP32C3-WROOM-02 was difficult to use. We had JTAG issues, and trouble with serial ports, and even programming the devices at times. It was definitely insightful to learn how to more effectively troubleshoot and circumvent challenges when things did not go to plan, but it definitely could have been smoother, and we could have made better use of our time and

resources had we faced less challenges that forced us to redirect completely rather than make small adjustments.

Thorough, consistent documentation was very helpful throughout the design process, especially in a group project where tasks were often juggled and picked up unfinished by someone else. Overall, despite the challenges and some shortcomings in the “polish” department, we believe that we successfully designed and implemented a functional prototype that could feasibly be used by a smallholding farmer, albeit with some hiccups along the way.

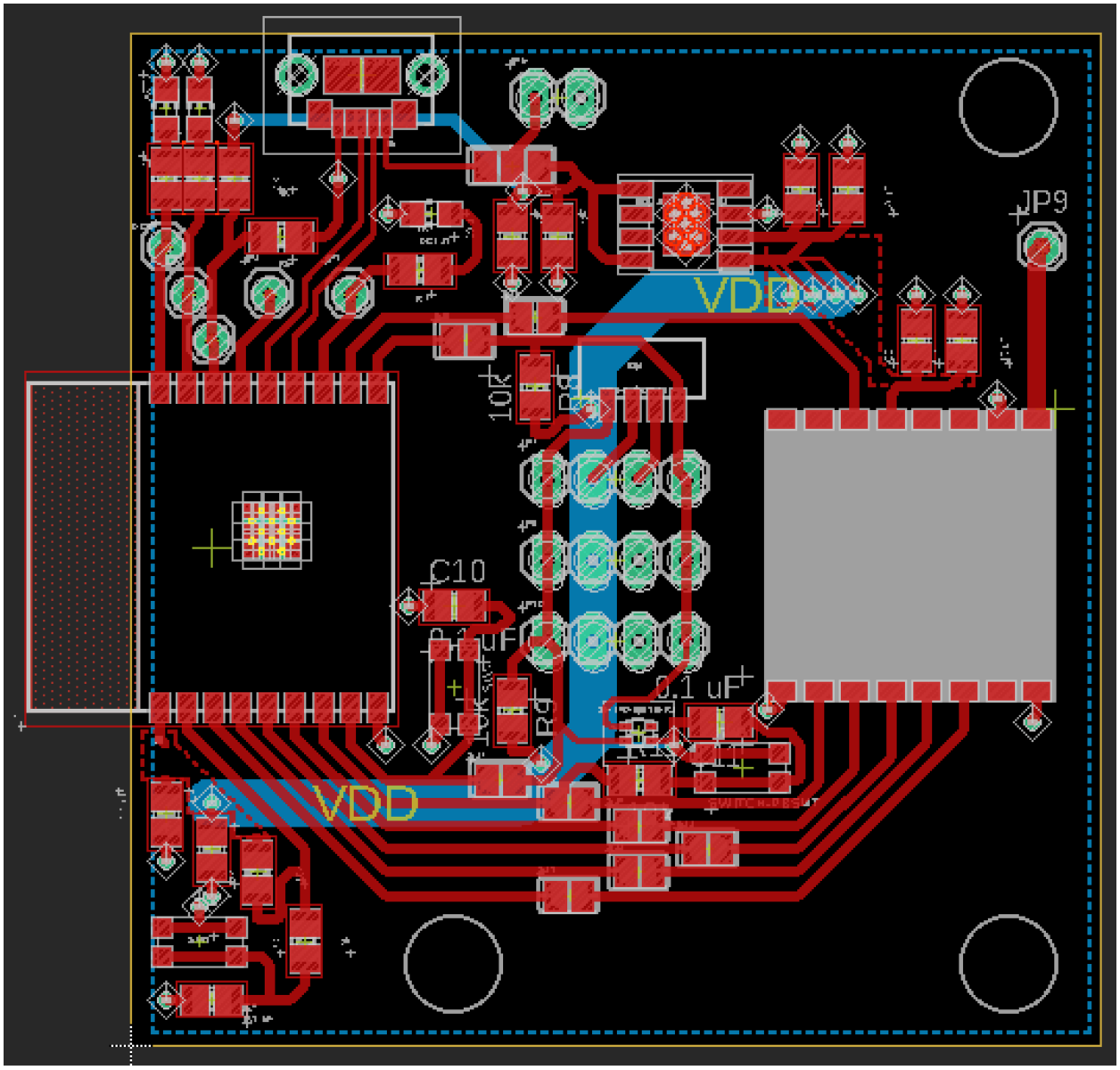
Appendices

Schematic:



Note- some pins were changed manually by us after receiving the board- the pins described in the documentation text are the final pins used.

Board Layout:



Datasheets:
ESP32-C3-WROOM-02:
https://www.espressif.com/sites/default/files/documentation/esp32-c3-wroom-02_datasheet_en.pdf
ESP32 Series:
https://www.espressif.com/sites/default/files/documentation/esp32-c3_datasheet_en.pdf
RFM95CW:
https://cdn.sparkfun.com/assets/1/5/d/6/6/RFM95CW_Specification_EN_V1.0.pdf
VEML7700 (Light Sensor):
https://www.vishay.com/docs/84286/veml7700.pdf
SHT-4x:
https://www.mouser.com/datasheet/2/682/Datasheet_SHT4x-1917879.pdf

Code:

Field Device:

```
#include <SPI.h>
#include <LoRa.h>
#include <Arduino.h>
#include <SensirionI2CSht4x.h>
#include <Wire.h>
#include <SparkFun_VEML7700_Arduino_Library.h>
#include <esp32-hal-timer.h>

// Ambient Light Sensor
VEML7700 lightSensor;
// Temp / Humidity
SensirionI2CSht4x sht4x;

// Pin selections for LoRa SPI -- need to be changed for our board
#define ss 5 // 5 // 5 //16
#define rst -1 //4
#define dio0 20 //1 //20 // 26
// Pin selections for actuators
#define act1 0
#define act2 1
#define act3 1
#define act4 1

// Other pin selections
```

```
#define batteryPin 2
#define arPin 3

// for testing
int count = 0;

// incoming variables
byte deviceID = 0x03; //02
byte act1val = 0xFF;
byte act2val = 0xFF;
byte act3val;
byte act4val;
byte ack = 0xAA;
byte nack = 0x00;
byte req = 0x99;
byte firstByteAR;
byte secondByteAR;
byte firstByteLIGHT;
byte secondByteLIGHT;
byte firstByteTEMP;
byte secondByteTEMP;
byte firstByteHUM;
byte secondByteHUM;

// outgoing variables
byte baseID = 0x01; //01
byte batteryLevel;
int light;
int temperature;
int humidity;
int i2c2Val = 0xFFFF;
int i2c3Val = 0x4344;
int i2c4Val = 0x2941;

bool confirmed;
bool sht4xPresent = false;
bool luxPresent = false;

float luxRaw;
int lux;
```

```
String sensorReading;
byte acknowledge;

// timer variables
volatile int interruptCounter;
int totalInterruptCounter;

extern "C" void phy_bbpll_en_usb(bool en);

// timer initialization
static hw_timer_t *timer = NULL;
portMUX_TYPE timerMux = portMUX_INITIALIZER_UNLOCKED;
void IRAM_ATTR onTimer() {
  portENTER_CRITICAL_ISR(&timerMux);
  interruptCounter++;
  portEXIT_CRITICAL_ISR(&timerMux);
}

void setup() {
  // Serial Setup:
  phy_bbpll_en_usb(true);
  vTaskDelay(200);
  Serial.begin(115200); // Begin serial connection
  Serial.println("start");
  // Timer Setup as Interrupt
  timer = timerBegin(0, 80, true);
  timerAttachInterrupt(timer, &onTimer, true);
  timerAlarmWrite(timer, 30000000, true);
  timerAlarmEnable(timer);
  Serial.println();
  Serial.println("Got here at least");

  pinMode(act1, OUTPUT);
  pinMode(act2, OUTPUT);
  digitalWrite(act1, HIGH);
  digitalWrite(act2, HIGH);

  // Pin Mode Setup:
//  ledcSetup(0, 5000, 8); // setup led channel (channel, frequency, resolution)
```

```
//  ledcAttachPin(act1, 0); // attach pin to led channel (pin, channel)
//  ledcWrite(act1, 255); // analog write equivalent (pin, magnitude)
//  ledcSetup(1, 5000, 8);
//  ledcAttachPin(act2, 1);
//  ledcWrite(act2, 255);
//  ledcSetup(2, 5000, 8);
//  ledcAttachPin(act3, 0);
//  ledcWrite(act3, 255);
//  ledcSetup(3, 5000, 8);
//  ledcAttachPin(act4, 0);
//  ledcWrite(act4, 255);


  Serial.println("Setup Pin Mode");

  // LoRa Setup:
  //LoRa.setPins(ss,rst,dio0); // sets pins used for lora -- ss is chip select, rst is reset (may be
useful for periodic reset of radio) and dio0 is used to communicate between radio and esp32
  LoRa.setPins(ss, rst, dio0);
  while (!LoRa.begin(915E6)) { // waits until LoRa is set up
    Serial.println(".");
    delay(500);
  }

  Serial.println("LoRa.setPins executed");
  Serial.println("LoRa worked");
  LoRa.setSyncWord(0x01); // receive only from paired sender
  Serial.println("LoRa set sync");
  // Light Sensor Setup:
  if (luxPresent == true) {

    Wire.begin(); // i2c initialization
    Serial.println("Wire began");

    if (lightSensor.begin() == false) { // check if VEML7700 is attached properly to the board
      Serial.println("Unable to communicate with VEML7700");
      //    while (1) {
      //      Serial.print("shit");
      //      delay(500);
      //    }
```

```
    }

    Serial.print("Light Sensor began");
    lightSensor.setIntegrationTime(VEML7700_INTEGRATION_25ms); // set integration time, modify to daylight conditions to maximize resolution without overflow
    lightSensor.setSensitivityMode(VEML7700_SENSITIVITY_x1); // set sensitivity, again may modify to maximize resolution
    Serial.println("Light sensor setup good");
    // Temp/Humidity Setup
    // Wire.begin(); // i2c initialization
  }

  delay(100);

//    // Error checking for SHT-4x:
//    uint16_t error;
//    char errorMessage[256];
//    sht4x.begin(Wire); // begin SHT-4x
//    uint32_t serialNumber;
//    error = sht4x.serialNumber(serialNumber); // if there is an error message, display, otherwise display serial number of device
//    if (error) {
//        Serial.print("Error trying to execute serialNumber(): ");
//        errorToString(error, errorMessage, 256);
//        Serial.println(errorMessage);
//    } else {
//        Serial.print("Serial Number: ");
//        Serial.println(serialNumber);
//    }
}




void loop() {
  readPacket(); // scan for received transmission

  if (interruptCounter > 0) {
    portENTER_CRITICAL(&timerMux);
    interruptCounter--;
```

```
    portEXIT_CRITICAL(&timerMux);

    totalInterruptCounter++;

    Serial.println("An interrupt has occurred. Now going to start sending process. ");
    confirmed = false;
    while (!confirmed) {
      obtainData();
      sendPacket(nack);
    }


  }

  // check for send condition
  //  int button = analogRead(0);
  //  if (button < 100) {
  //    delay(50);
  //    Serial.println("Packet Send Requested by User");
  //    confirmed = false;
  //    while (!confirmed) {
  //    obtainData();
  //    sendPacket(0x00);
  //    }
  //  }
}

void readPacket() {
  int packetSize = LoRa.parsePacket(); // check for incoming packet
  if (packetSize) { // if there is a packet:
    Serial.println("Homie my cracka is a dog (that is to say we gotsa package");
    while (LoRa.available()) {
      String receivedMessage = LoRa.readString(); // save packet as an array of bytes
      if (deviceID == receivedMessage[0]) { // first byte within the message contains the device
ID. If the ID matches the field device message, then the field device takes the message,
otherwise, skip
        // ****************** CHANGED THESE IF CONDITIONS TOO ****************
        if (receivedMessage[5] == nack) {
```

```
        Serial.println("nack message received, setting actuators"); // check if the message is an acknowledgement message. If acknowledgement message, then do not respond. If not an acknowledgement message, it is a data message -- save data and respond
        act1val = receivedMessage[1]; // value for actuator 1
        Serial.println(act1val);
        act2val = receivedMessage[2]; // value for actuator 2
        Serial.println(act2val);
        act3val = receivedMessage[3]; // value for actuator 3
        Serial.println(act3val);
        act4val = receivedMessage[4]; // value for actuator 4
        Serial.println(act4val);

                if (act1val > 100) {
                  act1val = 1;
                }
                else {
                  act1val = 0;
                }
                if (act2val > 100) {
                  act2val = 1;
                }
                else {
                  act2val = 0;
                }


                digitalWrite(act1,act1val);
                digitalWrite(act2,act2val);

        for (int i = 0; (i <= 15); i++) {
          sendPacket(ack); // send acknowledgement packet 15 times to ensure acknowledge is received by base station
          delay(2); // pause for 2 milliseconds for message to finish sending
        }
      }
      // ************* NEW STUFF ****************
      else if (receivedMessage[5] == req) {
        Serial.println("status request received");
        obtainData();
        for (int i = 0; (i <= 15); i++) {
```

```
          Serial.println("Status update sent");
          sendPacket(nack); // send acknowledgement packet 15 times to ensure acknowledge is
received by base station
          delay(2); // pause for 2 milliseconds for message to finish sending
         }
        }

      }
    }
  }
}

void obtainData() {
  Serial.println("obtaining Data");
  checkLight();
  checkTemperatureHumidity();
  checkAR(arPin);
  checkBattery();
}

void sendPacket(byte acknowledgeYN) {
  LoRa.beginPacket();
  LoRa.write(baseID); // basestation ID -- 0
  LoRa.write(deviceID); // field device ID number -- 1
  LoRa.write(batteryLevel); // battery level reading -- 2
  LoRa.write(firstByteAR); // analog sensor high byte -- 3
  LoRa.write(secondByteAR); // analog sensor low byte -- 4
  LoRa.write(firstByteLIGHT); // light sensor high byte -- 5
  LoRa.write(secondByteLIGHT); // light sensor low byte -- 6
  LoRa.write(firstByteHUM); // humidity sensor high byte -- 7
  LoRa.write(secondByteHUM); // humidity sensor low byte -- 8
  LoRa.write(firstByteTEMP); // temperature sensor high byte -- 9
  LoRa.write(secondByteTEMP); // temperature sensor low byte -- 10
  LoRa.write(acknowledgeYN);  // acknowledgement -- 11
  LoRa.write(act1val); // actuator values -- 12
  LoRa.write(act2val);  //actuator values -- 13
  LoRa.write(act3val); //actuator values -- 14
  LoRa.endPacket();

  Serial.println("Sent!");
```

```
  if (acknowledgeYN != ack) { // if written as a data packet, check to ensure the base station received the message by listening for acknowledgement
    Serial.println("check send");
    checkSend();
  }
}

void checkBattery() {
  int batteryLevel_12bit = analogRead(batteryPin); // record 12-bit ADC value that corresponds to battery voltage divider
  batteryLevel = map(batteryLevel_12bit, 0, 4095, 0, 100); // map value to appropriate value, eventually will be able to calibrate and map to battery level percentage
}

void checkAR(int pin) {
  int arVal = analogRead(pin); // read value from analog sensor
  Serial.print("Moisture Value: ");
  Serial.println(arVal);
  firstByteAR = int2byte(arVal, 1); // save first 4 bits to high byte

  secondByteAR = int2byte(arVal, 2); // save last 8 bits to low byte
  Serial.print("Moisture Value first byte: ");
  Serial.println(firstByteAR);
  Serial.print("Moisture Value second byte: ");
  Serial.println(secondByteAR);
}

void checkLight() {

  if (luxPresent == true) {
    luxRaw = lightSensor.getLux(); // obtain lux value as float from light sensor
    lux = round(luxRaw); // round lux value and save as an int for easier workability
  }

  else {
    luxRaw;
    lux = round(random(40, 100));
  }
```

```
  Serial.println("New transmission ----------------------------------------------");
  Serial.print("Raw lux: ");
  Serial.println(luxRaw);
  // light = round(luxRaw*100);
  light = lux;
  firstByteLIGHT = int2byte(light, 1); // save high byte as first byte
  secondByteLIGHT = int2byte(light, 2); // save low byte as second byte
  // Print statements for testing
  Serial.print("Lux: ");
  Serial.println(map(lux, 0, 65535, 0, 16383)); // change this from 0,65535 to 0,4096?
  Serial.print("First byte light: ");
  Serial.println(firstByteLIGHT, HEX);
  Serial.print("Second byte light: ");
  Serial.println(secondByteLIGHT, HEX);
}


void checkSend() { // check to listen for acknowledgement message from basestation for some
time
  for (int i = 0; i < 8096; i++) {
    int packetSize = LoRa.parsePacket();
    if (packetSize) {
      Serial.println("packet found in check send");
      while (LoRa.available()) {
        String confirmationMessage = LoRa.readString();
        if (deviceID == confirmationMessage[0]) {
          Serial.println("this message is for me");
          if (confirmationMessage[5] == ack) { // if message contains acknowledgement byte, stop
sending
            Serial.print("Confirmation: 0x");
            Serial.println(confirmationMessage[5], BIN);
            confirmed = true; // confirmed = true will stop the field module from continuing
transmission
            i = 8096; // when message has been received, exit loop and stop listening for an
acknowledgement
          }
        }
      }
    }
  }
```

```
}

void checkTemperatureHumidity() {
  uint16_t error;
  char errorMessage[256];

  float temperatureRaw;
  float humidityRaw;

  if (sht4xPresent == true) {

    error = sht4x.measureHighPrecision(temperatureRaw, humidityRaw); // check for either an
error or for sensor readings
  }

  if (sht4xPresent == true) {
    if (error) {
      errorToString(error, errorMessage, 256); // if there is an error message, convert to error
message
    }
    else { // if no error:
      temperature = round(temperatureRaw * 100); // multiply temperature by 100 and round, to
save 2 decimal places but remain an int
      humidity = round(humidityRaw * 100); // multiply humidity by 100 and round, to save 2
decimal places but remain an int


      // Print for checking:
      Serial.print("Temperature:");
      Serial.print(map(temperature, 0, 65535, 0, 16383));
      Serial.print("\t");
      Serial.print("Humidity:");
      Serial.println(map(humidity, 0, 65535, 0, 16383));
    }
  }
  else {
    temperature = round(random(210, 250));
    humidity = round(random(40, 80));
  }
  // Prints for checking:
```

```
  Serial.print("Raw Temperature: ");
  Serial.println(temperatureRaw);
  Serial.print("Raw humidity");
  Serial.println(humidityRaw);

  firstByteTEMP = int2byte(temperature, 1); // save high byte as first temperature byte
  secondByteTEMP = int2byte(temperature, 2); // save low byte as second temperature byte

  firstByteHUM = int2byte(humidity, 1); // save high byte as first humidity byte
  secondByteHUM = int2byte(humidity, 2); // save low byte as second humidity byte

  // Print for checking:
  Serial.print("First byte temperature: ");
  Serial.println(firstByteTEMP, HEX);
  Serial.print("Second byte temperature: ");
  Serial.println(secondByteTEMP, HEX);
  Serial.print("First byte humidity: ");
  Serial.println(firstByteHUM, HEX);
  Serial.print("Second byte humidity: ");
  Serial.println(secondByteHUM, HEX);

}

byte int2byte(int a, int byteNum) {
  byte returnByte;
  int b = map(a, 0, 65535, 0, 16383);


  if (byteNum == 1) { // if byte is the high byte
    returnByte = (b >> 7); // bit shift rightward the integer 'a' to save as byte
  }
  else if (byteNum == 2) { // if byte is the low byte
    returnByte = (0x007F & b); // save last 7 bits of int
  }

  return returnByte; // return decomposition as a byte
}
```

Base- C Code:

```
// Ajax headers
```

```
#include <Wire.h>
#include <Arduino.h>
#include <WiFi.h>
#include <WiFiClient.h>
#include <WebServer.h>
#include "index.h"
// LoRa headers
#include <SPI.h>
#include <LoRa.h>


// Pin selections for LoRa SPI -- need to be changed for our board
#define ss 5
#define rst -1
#define dio0 20
// Pin selections for actuators
#define LED1 0
#define LED2 1
#define LED3 1

WebServer server(80);
const char* ssid = "ND-guest";
const char* password = "";

// Variable Declarations
bool LED1status = LOW;
bool LED2status = LOW;
bool LED3status = LOW;
bool messageComp = 0;
bool confirmed = false;

int message2send = 0;
int lastMessage = 0;

byte deviceID = 0x02;
byte baseID = 0x01;
byte act1val = 1;
byte act2val = 0; // set to zero for now since we don't have capability to change other things
byte act3val = 0;
byte act4val = 0;
```

```
byte ack = 0xAA;
byte nack = 0x00;
byte req = 0x99;

// uint8_t part1con;

int sensorID;
int checkBattery;
int analogSensor;
int lightSensor;
int i2c2Val;
int i2c3Val;
int i2c4Val;
int analSense;
int vemlRead;
int humRead;

String testServer = "a";
String testArray;
String statusUpdate;


// Functions for server

void handleRoot() {
 String s = MAIN_page; //Read HTML contents
 server.send(200, "text/html", s); //Send web page
}

void handleADC() {
 server.send(200, "text/plane", testArray); //Send ADC value only to client ajax request
 testArray[0] = 0x69; //arbitrary value that is not 0x01 like the basestation value
 // delay(1000);
}


void act1Control() {
  String state = "ON";
  String act_state = server.arg("state");
  deviceID = 0x02;
```

```
  if(act_state == "1"){
    act1val = 0xFF;
    state = "ON";
    Serial.print("on ");
  }
  else {
    act1val = 0x00;
    state = "OFF";
    Serial.print("off ");
  }
  server.send(200, "text/plane", state);

  confirmed = false;
  while (!confirmed) {
    packetWrite(nack);
    //delay(500)
    Serial.println("NACK message sent");
  }

  Serial.println("Packet Sent");
}

void updateRequest() {
  String deviceNumber = server.arg("device");
  if(deviceNumber == "3"){ Serial.println("Requesting from Device 3");
    deviceID = 0x03;
  }
  else if (deviceNumber == "2"){ Serial.println("Requesting from Device 2");
    deviceID = 0x02;
  }
  confirmed = false;
  while(!confirmed){
    packetWrite(req);
    Serial.println("Status Request Sent");
  }

  server.send(200, "text/plane", statusUpdate);
  // delay(1000);
}
```

```
void sprinkSel() {
  String selectedSprink = server.arg("state");
  Serial.print("sprink: ");
  Serial.println(selectedSprink);
  if(selectedSprink == "1"){
    act3val = 0xFF;
  }
  else{
    act3val = 0x00;
  }

  confirmed = false;
  while(!confirmed){
  packetWrite(nack);
  Serial.println("NACK message sent");
  }

}

void deviceSel() {
  String selectedDevice = server.arg("state");
  Serial.print("device: ");
  Serial.println(selectedDevice);
  if(selectedDevice == "3"){
    deviceID = 0x03;
  }
  else if(selectedDevice == "2"){
    deviceID = 0x02;
  }

}

void heatSel() {
  String selectedHeat = server.arg("state");
  Serial.print("heat: ");
  Serial.println(selectedHeat);
  if(selectedHeat == "1"){
      act1val = 0xFF;
      Serial.println("act1 on");
```

```
    }
    else{
      act1val = 0x00;
      Serial.println("act1 off");
    }

}

void shadeSel() {
  String selectedShade = server.arg("state");
  Serial.print("shade: ");
  Serial.println(selectedShade);
  if(selectedShade == "1"){
    act2val = 0xFF;
  }
  else{
    act2val = 0x00;
  }

}


extern "C" void phy_bbpll_en_usb(bool en);

void setup() {
  // Serial Setup:
  Serial.begin(115200); // Begin serial connection
  Serial.println("starting");
  // delay(5000);

 // WiFi Setup:
 phy_bbpll_en_usb(true);

 WiFi.mode(WIFI_STA); // connects to existing network
 phy_bbpll_en_usb(true);
 Serial.println("This boy's a wiFi station");
  WiFi.begin(ssid, password);
  Serial.println("Connecting to ");
  Serial.print(ssid);
  while(WiFi.waitForConnectResult() != WL_CONNECTED){
```

```
    Serial.print("."); // wait until wifi connection established
   }
  Serial.println("");
  Serial.print("Connected to ");
  Serial.println(ssid);
  Serial.print("IP address: ");
  Serial.println(WiFi.localIP());
  phy_bbpll_en_usb(true);

  // Server Setup:
  server.on("/", handleRoot);      // Display page
  server.on("/readADC", handleADC); //check for updated packet values
  server.on("/dev_sel", deviceSel); // use this to select which device actuation is being performed
on
  server.on("/heat_sel", heatSel); // pass the value of the heatlamp actuator
  server.on("/shade_sel", shadeSel); //pass value of shade actuator
  server.on("/sprink_sel", sprinkSel); //pass value of sprinkler actuator
  server.on("/request", updateRequest); // call this function when you want a live update and push
the update button

   server.begin(); //Start server
   Serial.println("HTTP server started");

  // Pin Mode Setup:
  pinMode(LED1,OUTPUT);
  pinMode(LED2,OUTPUT);
  pinMode(LED3,OUTPUT);
  digitalWrite(LED1,HIGH); // need to change from digital to analog write (everything)
  digitalWrite(LED2,HIGH);
  digitalWrite(LED3,HIGH);

  // LoRa Setup:
  LoRa.setPins(ss,rst,dio0);
  while (!LoRa.begin(915E6)) { // waits until LoRa is set up
    Serial.println(".");
    delay(500);
  }
  LoRa.setSyncWord(0x01); // receive only from paired sender
  Serial.println("LoRa Worked");
}
```

```
void loop() {

  // put your main code here, to run repeatedly:
  server.handleClient(); //packetWrite happens here when server UI is utilized
  delay(1);
  readPacket();
  // ****
//  int button = analogRead(0);
//  if (button < 100) {
//    delay(50);
//    Serial.print(button);
//    confirmed = false;
//    while (!confirmed) {
//      packetWrite(0x00);
//    }
//  }
}

// prepare basestation to field device packet
void packetWrite(byte acknowledgeYN) {

  Serial.println("packetWrite called");
  Serial.print("act1 val: ");
  Serial.println(act1val);
  Serial.print("act2 val: ");
  Serial.println(act2val);
  Serial.print("act3 val: ");
  Serial.println(act3val);

  // create packet
  LoRa.beginPacket();
  LoRa.write(deviceID);
  LoRa.write(act1val);
  LoRa.write(act2val);
  LoRa.write(act3val);
  LoRa.write(act4val);
  LoRa.write(acknowledgeYN);
  LoRa.endPacket();
```

```
  // delay(100);

  //check for confirmation
  if (acknowledgeYN != ack) {
//    if(acknowledgeYN == req){
//      for (int i = 1; i <= 10; i++){
//        checkSend();
//      }
//    }
    checkSend();
  }
}



void readPacket() {
  int packetSize = LoRa.parsePacket();
  // Serial.println("entered readPacket");
  if (packetSize) { Serial.println("Package found");
    while (LoRa.available()) {
      String message = LoRa.readString();
      Serial.println("Homies we gotsta package");
      if (baseID == message[0]) {
        if (message[11] != ack) { //needs to be 11c
          testArray = message;
          int deviceIDnum = message[1];
          deviceID = deviceIDnum;
          // testServer = deviceID;
          int batteryval = message[2];

          int analsens1 = message[3];
          int analsens2 = message[4];
          analSense = bitShift(analsens1, analsens2);

          int vemlread1 = message[5];
          int vemlread2 = message[6];
          vemlRead = bitShift(vemlread1, vemlread2);

          int humread1 = message[7];
          int humread2 = message[8];
```

```
        humRead = bitShift(humread1, humread2);

        int tempread1 = message[9];
        int tempread2 = message[10];
        int tempRead = bitShift(tempread1, tempread2);

        Serial.println(message);
        Serial.println("Base Station ID");
        Serial.println(baseID);
        Serial.println("Device ID");
        Serial.println(deviceIDnum);
        Serial.println("Battery Value");
        Serial.println(batteryval);
        Serial.println("Analog Sensor Reading");
        Serial.println(analsens1);
        Serial.println(analsens2);
        Serial.println("VEML Light Reading");
        Serial.println(vemlread1);
        Serial.println(vemlread2);
        Serial.println("Moisture Reading");
        Serial.println(humread1);
        Serial.println(humread2);
        Serial.print("Temp hiByte: ");
        Serial.println(tempread1, HEX);
        Serial.println("Temp loByte: ");
        Serial.println(tempread2, HEX);
        Serial.print("Temp Val: ");
        Serial.print(tempRead);

        for(int i = 1; i < 14; i++) {packetWrite(ack);}

      }
    }
  }
 }
}


// Functions for Transmission Reliability
```

```
//checkSend is used for confirming that a sent message has been received
// both base station and field device can use checkSend to ensure that
// their message has been received

void checkSend() {

Serial.println("Entered checkSend");
 for (int i = 0; i < 15000 ; i++) { //was 8096
    int packetSize = LoRa.parsePacket();
    if (packetSize) { Serial.println("Packet Received");
      while (LoRa.available()) {
        String confirmationMessage = LoRa.readString();

        if (baseID == confirmationMessage[0]) {
          Serial.println("Base ID correct");
          if (confirmationMessage[11] == ack) { //needs to be 11
            Serial.println("confirmation");
            confirmed = true;
            Serial.println("Confirmed set to true! Stop sending");
            i = 15000;
          }
          else if (confirmationMessage[11] == nack){ Serial.println("request complete");
            statusUpdate = confirmationMessage;
            confirmed = true;
            i = 15000;
          }
        }
      }
    }
  }
}


int bitShift(int a, int b) {

  int shifted = ((a << 7) | b);
  return shifted;
}
```

```
/* Packet Protocol -- 15 bytes in packet -- field to basestation
 *  Address - 1 bytes
 *  AnalogIn - 3 bytes (2 AnalogIn, each 12-bit adc)
 *  I2C - 8 bytes (2 bytes per actuator, 16-bit resolution)
 */

// Example: { 0x01 0x8A932F 0x2179ABCD10006969 }

/* Packet Protocol -- basestation to field
 *  Address - 1 bytes -- intended receiver
 *  Actuator - 3 bytes (1 bytes per actuator, 8-bit desired PWM)
 */

 // Example: { 0x03 0x001111 }
```

Base- index.h (including html, javascript):

```
const char MAIN_page[] PROGMEM = R"=====(
<!DOCTYPE html>
<html>
<style>
card {
    max-width: 1000px;
    height: 25px;
    padding: 0px;
    box-sizing: border-box;
    color: white;
    margin:20px;
    box-shadow: 0px 2px 18px -4px rgba(0,0,0,0.75);
}
.card1{background: #0f5f45;}
.card2{background: #ffffff;}

.card3 {
    max-width: 1000px;
    height: 500px;
    padding: 30px;
    box-sizing: border-box;
    background: #0f5f45;
    margin:20px;
    box-shadow: 0px 2px 18px -4px rgba(0,0,0,0.75);
```

```
}

.button0 {
  border: none;
  color: white;
  padding: 30px;
  text-align: center;
  text-decoration: none;
  display: inline-block;
  font-size: 16px;
  margin: 4px 2px;
  cursor: pointer;
}

.button1 {background-color: #4CaF50;}
.button2 {background-color: #008CBA;}


table, th, td {
  border: 1px solid black;
  text-align: center;
}

.wrap-flex { display: flex; justify-content: center; margin: 0 -5px}

</style>

<head>
<title> LoRa Farms </title>
</head>


<body>
<center>
<div class="card card1">
  <h1>Welcome to LoRa Farms!</h1><br>
  <h2> Farm Control Dashboard </h2> <br>

</div>
```

```
<br>

<h3> Actuator Status </h3>
<button class = "button" onclick = "requestReadings(3)";> Update 3 </button>
<button class = "button" onclick = "requestReadings(2)";> Update 2 </button>
<br>
<table>
  <tr>
    <td> Time </td>
    <td> Device Number </td>
    <td> Battery Level </td>
    <td> Soil Moisture </td>
    <td> Light </td>
    <td> Humidity </td>
    <td> Temperature </td>
    <td> Heat Lamp </td>
    <td> Shade </td>
    <td> Sprinklers </td>
  </tr>
  <tr>
    <td id = "time" rowspan = "2"> -- </td>
    <td > 3 </td>
    <td id = "levelBat3"> -- </td>
    <td id = "moistSoil3"> -- </td>
    <td id = "light3"> -- </td>
    <td id = "humidity3"> -- </td>
    <td id = "temp3"> -- </td>
    <td id = "heatLamp3Stat"> -- </td>
    <td id = "shade3Stat"> -- </td>
    <td id = "sprink3Stat"> -- </td>
  </tr>
  <tr>
    <td > 2 </td>
    <td id = "levelBat2"> -- </td>
    <td id = "moistSoil2"> -- </td>
    <td id = "light2"> -- </td>
    <td id = "humidity2"> -- </td>
    <td id = "temp2"> -- </td>
    <td id = "heatLamp2Stat"> -- </td>
    <td id = "shade2Stat"> -- </td>
```

```
    <td id = "sprink2Stat"> -- </td>
</table>



<br>


<div class="wrap-flex", style = "text-align: center;">
    <div class = "box", style = "margin: 0 5px;">
      <label for = "deviceNum">Device</label>
      <!-- <select name = "deviceNum" id ="deviceNum">
        <label> Device Number </label>
      </select> -->
        <button onclick="device_select(3)"; > Device 3 </button>
        <button onclick= "device_select(2)";> Device 2 </button>
      </select>
    </div>
    <br>
    <div class = "box", style = "margin: 0 5px;">
      <label for = "heatLamp">Heat Lamp </label>
      <!-- <select name = "heatLamp" id ="heatLamp"> -->
        <!-- <label> Heat Lamp Status </label> -->
        <button onclick = "heatLamp_set(0)";> Off </button>
        <button onclick = "heatLamp_set(1)";> On </button>
      </select>
    </div>

    <div class = "box", style = "margin: 0 5px;">
      <label for = "shadeAct">Shade</label>
      <!-- <select name = "shadeAct" id ="shadeAct">
        <label> Shade Value </label> -->
        <button onclick = "shadeAct_set(0)";> Off </button>
        <button onclick="shadeAct_set(1)";> On </button>
      </select>
    </div>

    <div class = "box", style = "margin: 0 5px;">
      <label for = "sprinklerAct">Sprinklers</label>
      <!-- <select name = "sprinklerAct" id ="sprinklerAct">
```

```
      <label> Sprinkler Value </label> -->
      <button onclick = "sprinklerAct_set(0)";> Off </button>
      <button onclick = "sprinklerAct_set(1)";> On </button>
    </select>
  </div>
 </div>

 <br>

 <table>
  <tr>
    <td id = "deviceHead"> Device Number</td>
    <td id = "heatLampHead"> Heat Lamp </td>
    <td id = "shadeActHead"> Shade </td>
    <td id = "sprinklerActHead"> Sprinklers </td>
  </tr>
  <tr>
    <td id = "deviceCell"> -- </td>
    <td id = "heatLampCell"> -- </td>
    <td id = "shadeActCell"> -- </td>
    <td id = "sprinklerActCell"> -- </td>
  </tr>
 </table>

 <br>

 <button class = "button" onclick = "pass()";> Send </button>

 <br>
 <br>


<button class = "button0 button1", style = "height:75px; padding:10px" onclick=
"table_to_CSV()"> Download Table </button>
<br>


<table>
  <tr>
    <th colspan = "8"> Data Log </th>
```

```
  </tr>
  <tbody id ="rows">
    <tr>
      <td id = "date"> Date </td>
      <td id = "time"> Time </td>
      <td id = "device_id"> Sensor ID </td>
      <td id = "battery_value" > Battery Value </td>
      <td id = "moisture_value" > Moisture Value </td>
      <td id = "light_value" > Light Value </td>
      <td id = "humidity_value" > Humidity Value </td>
      <td id = "temperature_value" > Temperature Value </td>
    </tr>
   </tbody>
</table>



</center>
<script>

setInterval(function() {
  // Call a function repetatively with 2 Second interval
  getData();
}, 100); //2000mSeconds update rate

function getData() {
  var xhttp = new XMLHttpRequest(); //some AJAX request has a variable name xhttp
  xhttp.onreadystatechange = function() {            //define a function called onreadystatechange
    if (this.readyState == 4 && this.status == 200) {       //check to see when readyState indicates
finished and status indicates success

      variable = this.responseText; //variable = testArray

        if(this.responseText.charCodeAt(0) == "0x01") {
          //define variable types as rows and cells
          var newRow = document.createElement("tr");
          //var newCell = document.createElement("td");
          var newDate = document.createElement("td");
          var newTime = document.createElement("td");
          var newID = document.createElement("td");
          var newBat = document.createElement("td");
```

```
        var newAnal = document.createElement("td");
        var newLight = document.createElement("td");
        var newHum = document.createElement("td");
        var newTemp = document.createElement("td");

        //populate cells
        newDate.innerHTML = get_date();
        newTime.innerHTML = get_time();
        newID.innerHTML = this.responseText.charCodeAt(1);
        newBat.innerHTML = this.responseText.charCodeAt(2);

        //processing 2-byte values
        var analogValue = Math.round(( (this.responseText.charCodeAt(3) << 7) |
(this.responseText.charCodeAt(4)) ) * (65535/16383))/100;
        var lightValue = Math.round(( (this.responseText.charCodeAt(5) << 7) |
(this.responseText.charCodeAt(6)) ) * (65535/16383))/100;
        var humValue = (Math.round(( (this.responseText.charCodeAt(7) << 7) |
(this.responseText.charCodeAt(8)) ) * (65535/16383))/100);
        var tempValue = (Math.round(( (this.responseText.charCodeAt(9) << 7) |
(this.responseText.charCodeAt(10)) ) * (65535/16383))/100);
        var tempComb = (this.responseText.charCodeAt(9) << 7) |
(this.responseText.charCodeAt(10));
        console.log("combined bytes:");
        console.log(tempComb);
        var tempUnmap = tempComb * 4;
        console.log("unmapped:");
        console.log(tempUnmap);
        var tempRound = Math.round(tempUnmap);
        var temp2meNice = tempRound/100;

        console.log("temp2meNice: ");
        console.log(temp2meNice);
        console.log("temp value: ");
        console.log(tempValue);
        console.log("hi_byte:");
        console.log(this.responseText.charCodeAt(9));
        console.log("lo_byte:");
        console.log(this.responseText.charCodeAt(10));
```

```
        //plugging 2-byte values into their cells
        newAnal.innerHTML = analogValue;
        newLight.innerHTML = lightValue;
        newHum.innerHTML = humValue;
        newTemp.innerHTML = tempValue;

        newRow.append
        newRow.append(newDate);
        newRow.append(newTime);
        newRow.append(newID); //new
        newRow.append(newBat);
        newRow.append(newAnal);
        newRow.append(newLight);
        newRow.append(newHum);
        newRow.append(newTemp);

        document.getElementById("rows").appendChild(newRow); //rows is the name of the
body, so we are adding a new row to the body
      }

   }
  };
  xhttp.open("GET", "readADC", true);
  xhttp.send();
}

function requestReadings(deviceNumber) {
  var request = new XMLHttpRequest();
   request.onreadystatechange = function() {
     if(this.readyState == 4 && this.status == 200) {

       document.getElementById("time").innerHTML = get_time();
       status = this.responseText;
       if (this.responseText.charCodeAt(1) == 2) {
         document.getElementById("levelBat2").innerHTML = this.responseText.charCodeAt(2);
         document.getElementById("moistSoil2").innerHTML = Math.round((
(this.responseText.charCodeAt(3) << 7) | (this.responseText.charCodeAt(4)) ) *
(65535/16383))/100;
```

```
        document.getElementById("light2").innerHTML = Math.round((
(this.responseText.charCodeAt(5) << 7) | (this.responseText.charCodeAt(6)) ) *
(65535/16383))/100;
        document.getElementById("humidity2").innerHTML = (Math.round((
(this.responseText.charCodeAt(7) << 7) | (this.responseText.charCodeAt(8)) ) *
(65535/16383))/100) + 35;
        document.getElementById("temp2").innerHTML = (Math.round((
(this.responseText.charCodeAt(9) << 7) | (this.responseText.charCodeAt(10)) ) *
(65535/16383))/100) + 1.5;
        if(this.responseText.charCodeAt(12) == 0){
          document.getElementById("heatLamp2Stat").innerHTML = "off";
        }
        else if (this.responseText.charCodeAt(12) >= 1){
          document.getElementById("heatLamp2Stat").innerHTML = "on";
        }
        if(this.responseText.charCodeAt(13) == 0){
          document.getElementById("shade2Stat").innerHTML = "off";
        }
        else if (this.responseText.charCodeAt(13) >= 1){
          document.getElementById("shade2Stat").innerHTML = "on";
        }
        // document.getElementById("shade2Stat").innerHTML =
this.responseText.charCodeAt(13);
        if(this.responseText.charCodeAt(14) == 0){
          document.getElementById("sprink2Stat").innerHTML = "off";
        }
        else if (this.responseText.charCodeAt(14) >= 1){
          document.getElementById("sprink2Stat").innerHTML = "on";
        }
        //document.getElementById("sprink2Stat").innerHTML =
this.responseText.charCodeAt(14);
      }

      else if (this.responseText.charCodeAt(1) == 3) {
        document.getElementById("levelBat3").innerHTML = this.responseText.charCodeAt(2);
        document.getElementById("moistSoil3").innerHTML = Math.round((
(this.responseText.charCodeAt(3) << 7) | (this.responseText.charCodeAt(4)) ) *
(65535/16383))/100;
```

```
        document.getElementById("light3").innerHTML = Math.round((
(this.responseText.charCodeAt(5) << 7) | (this.responseText.charCodeAt(6)) ) *
(65535/16383))/100;
        document.getElementById("humidity3").innerHTML = (Math.round((
(this.responseText.charCodeAt(7) << 7) | (this.responseText.charCodeAt(8)) ) *
(65535/16383))/100) + 35;
        document.getElementById("temp3").innerHTML = (Math.round((
(this.responseText.charCodeAt(9) << 7) | (this.responseText.charCodeAt(10)) ) *
(65535/16383))/100) + 1.5;
        //document.getElementById("heatLamp3Stat").innerHTML =
this.responseText.charCodeAt(12);
        if(this.responseText.charCodeAt(12) == 0){
          document.getElementById("heatLamp3Stat").innerHTML = "off";
        }
        else if (this.responseText.charCodeAt(12) == 1){
          document.getElementById("heatLamp3Stat").innerHTML = "on";
        }
        //document.getElementById("shade3Stat").innerHTML =
this.responseText.charCodeAt(13);
        if(this.responseText.charCodeAt(13) == 0){
          document.getElementById("shade3Stat").innerHTML = "off";
        }
        else if (this.responseText.charCodeAt(13) == 1){
          document.getElementById("shade3Stat").innerHTML = "on";
        }
        //document.getElementById("sprink3Stat").innerHTML =
this.responseText.charCodeAt(14);
        if(this.responseText.charCodeAt(14) == 0){
          document.getElementById("sprink3Stat").innerHTML = "off";
        }
        else if (this.responseText.charCodeAt(14) == 1){
          document.getElementById("sprink3Stat").innerHTML = "on";
        }
       }

     }
   };
  request.open("GET", "request?device="+deviceNumber, "true");
  request.send();
}
```

```
function send(act1_sts){
 var xhttp = new XMLHttpRequest();
  xhttp.onreadystatechange = function() {
    if (this.readyState == 4 && this.status == 200) {
      document.getElementById("act1").innerHTML =
      this.responseText;
    }
  };
  xhttp.open("GET", "act1_set?state="+act1_sts, true);
  xhttp.send();
}

function pass(){
 var dev = new XMLHttpRequest();
  dev.onreadystatechange = function() {
    if (this.readyState == 4 && this.status == 200) {
//      selectedDevice;
//      heatLampLevel;
//      shadeActLevel;
//      sprinklerActLevel;

    }
  };
  dev.open("GET", "dev_sel?state="+selectedDevice, true);
  dev.send();
  var heat = new XMLHttpRequest();
  heat.open("GET", "heat_sel?state="+heatLampLevel, true);
  heat.send();
  var shade = new XMLHttpRequest();
  shade.open("GET", "shade_sel?state="+shadeActLevel, true);
  shade.send();
  var sprink = new XMLHttpRequest();
  sprink.open("GET", "sprink_sel?state="+sprinklerActLevel, true);
  sprink.send();
}


function get_date() {
```

```
  var today = new Date();
  var date = (today.getMonth() + 1) + '-' + today.getDate() + '-' + today.getFullYear();
  return date;

}

function get_time() {
  var today = new Date();
  var time = today.getHours() + ":" + today.getMinutes() + ":" + today.getSeconds();
  return time
}

function table_to_CSV() {

  //variable to store final csv
  var csv_data = [];

  //get each row's data
  var rows = document.getElementsByTagName('tr');
  for(var i = 0; i < rows.length; i++) {
    // get each column data
    var cols = rows[i].querySelectorAll('td, th');

    //store rows as csv data type
    var csvrow =[];
    for(var j = 0; j < cols.length; j++){

      //get text data of each cell of a row
      // and push it to csvrow
      csvrow.push(cols[j].innerHTML);
    }

    //now combine columns with commas as delimiters
    csv_data.push(csvrow.join(","));
  }

  //combine each row data with newline char as delimiters
  csv_data = csv_data.join("\n");

  //call download function to download created table
```

```
  download_CSV_file(csv_data);

}


function download_CSV_file(csv_data) {

  //Create CSV file object and feed our csv_data into it
  CSV_file = new Blob([csv_data], {type: "text/csv"});

  //create to temporary link to initiate download process
  var temp_link = document.createElement('a');

  //download CSV
  temp_link.download = "Sensor Data.csv";
  var url = window.URL.createObjectURL(CSV_file);
  temp_link.href = url;

  //shouldnt display this link
  temp_link.style.display = "none";
  document.body.appendChild(temp_link);

  //automatically click link to trigger download
  temp_link.click();
  document.body.removeChild(temp_link);

}

var counter = 0;
var startDelay = 0;

var selectedDevice = 0;

function device_select(x) {console.log("called function");
  selectedDevice = x;
  console.log(selectedDevice);
  document.getElementById("deviceCell").innerHTML = x;
}

var heatLampLevel;
```

```
function heatLamp_set(x) {
  heatLampLevel = x;
  if(x == 1){ y = "on";}
    else if (x == 0) {y = "off";}

  document.getElementById("heatLampCell").innerHTML = y;

}

var shadeActLevel;

function shadeAct_set(x) {
  shadeActLevel = x;
  if(x == 1){ y = "on";}
    else if (x == 0) {y = "off";}
  document.getElementById("shadeActCell").innerHTML = y;

}

var sprinklerActLevel;

function sprinklerAct_set(x) {
  sprinklerActLevel = x;
  if(x == 1){ y = "on";}
    else if (x == 0) {y = "off";}
  document.getElementById("sprinklerActCell").innerHTML = y;

}


</script>
</body>
</html>
)=====";
```